CO2 EMISSIONS (EDA and ML)

Linear Regression and Regularization (Ridge-Lasso-GridSearchCV)

Duygu Jones | Data Scientist | July 2024
Follow me: duygujones.com | Linkedin | GitHub | Kaggle | Medium | Tableau

Project Description

Objective: The goal of this project is perform an EDA to prepare the data for a Machine Learning Models, focusing on building a Linear Model.

The project consists of two main phases:

1. **Exploratory Data Analysis (EDA)**:
    - For the dataset of Co2 Emission by Vehicle in Canada, performed an Exploratory Data Analysis (EDA) to understand the general structure of the dataset, summarize key statistical insights, and explore relationships between independent variables and the target variable.
2. **Machine Learning Model**: Preparing the data and building models, including Simple and Multiple Linear Regression, Polynomial Regression, and Regularization techniques.
    - **Simple Linear Regression Model**:
    - **Multiple Linear Regression Model**:
    - **Polynomial Regression Model**:
    - **Scaling the Data**:
    - **Regularization**:
        - Ridge regression was evaluated with cross-validation and GridSearchCV was used to choose the best alpha value.
        - Lasso regression was also evaluated similarly, with GridSearchCV used for parameter tuning.
    - **Final Model and Prediction**: The final model was built and predictions were made.

- **Feature Importances**: The importance of features was analyzed using Ridge and Lasso regression techniques.

As this dataset was compiled for analytical and educational purposes, steps suitable for beginners will be followed to ensure a thorough understanding.

About the Dataset

- This dataset contains information about various vehicles' carbon dioxide (CO2) emissions and fuel consumption.
- In the context of Machine Learning (ML), this dataset is often used to predict CO2 emissions based on vehicle characteristics or to analyze fuel efficiency of vehicles.
- The goal could be to predict CO2 emissions or fuel consumption based on the features of the vehicles.
- There are total 7385 rows and 12 columns.

The columns in the dataset can be described as follows:

1. **Make**: The brand of the vehicle.
2. **Model**: The model of the vehicle.
3. **Vehicle Class**: The class of the vehicle (e.g., compact, SUV).
4. **Engine Size(L)**: The engine size in liters.
5. **Cylinders**: The number of cylinders in the engine.
6. **Transmission**: The type of transmission (e.g., automatic, manual).
7. **Fuel Type**: The type of fuel used (e.g., gasoline, diesel).
8. **Fuel Consumption City (L/100 km)**: Fuel consumption in the city (liters per 100 kilometers).
9. **Fuel Consumption Hwy (L/100 km)**: Highway (out-of-city) fuel consumption.
10. **Fuel Consumption Comb (L/100 km)**: Combined (city and highway) fuel consumption.
11. **Fuel Consumption Comb (mpg)**: Combined fuel consumption in miles per gallon. (efficiency-> less fule long way)
12. **CO2 Emissions(g/km)**: CO2 emissions in grams per kilometer.

NOTE:

11. **Fuel Consumption Comb (mpg)**:

- High mpg value: The vehicle operates more efficiently, consumes less fuel, and thus produces less CO2 emissions.
- Low mpg value: The vehicle consumes more fuel and produces more CO2 emissions.
  - Therefore, there is a negative relationship between "Fuel Consumption Comb (mpg)" and "CO2 Emissions."
  - As fuel efficiency increases (mpg value increases), CO2 emissions decrease.
  - This explains why environmentally friendly vehicles have high mpg values and produce fewer CO2 emissions.

Model

The "Model" column includes terms that identify specific features or configurations of vehicles:

- `4WD/4X4` : Four-wheel drive. A drive system where all four wheels receive power.
- `AWD` : All-wheel drive. Similar to 4WD but often with more complex mechanisms for power distribution.
- `FFV` : Flexible-fuel vehicle. Vehicles that can use multiple types of fuel, such as both gasoline and ethanol blends.
- `SWB` : Short wheelbase.
- `LWB` : Long wheelbase.
- `EWB` : Extended wheelbase.

Transmission

The "Transmission" column indicates the type of transmission system in the vehicle:

- `A` : Automatic. A transmission type that operates without the need for the driver to manually change gears.
- `AM` : Automated manual. A version of a manual transmission that is automated.
- `AS` : Automatic with select shift. An automatic transmission that allows for manual intervention.
- `AV` : Continuously variable. A transmission that uses continuously varying ratios instead of fixed gear ratios.
- `M` : Manual. A transmission type that requires the driver to manually change gears.
- `3 - 10` : Number of gears in the transmission.

Fuel Type

The "Fuel Type" column specifies the type of fuel used by the vehicle:

- `X` : Regular gasoline.
- `Z` : Premium gasoline.
- `D` : Diesel.
- `E` : Ethanol (E85).
- `N` : Natural gas.

Vehicle Class

The "Vehicle Class" column categorizes vehicles by size and type:

- `COMPACT` : Smaller-sized vehicles.
- `SUV - SMALL` : Smaller-sized sports utility vehicles.
- `MID-SIZE` : Medium-sized vehicles.
- `TWO-SEATER` : Vehicles with two seats.
- `MINICOMPACT` : Very small-sized vehicles.
- `SUBCOMPACT` : Smaller than compact-sized vehicles.
- `FULL-SIZE` : Larger-sized vehicles.
- `STATION WAGON - SMALL` : Smaller-sized station wagons.
- `SUV - STANDARD` : Standard-sized sports utility vehicles.
- `VAN - CARGO` : Vans designed for cargo.
- `VAN - PASSENGER` : Vans designed for passenger transportation.
- `PICKUP TRUCK - STANDARD` : Standard-sized pickup trucks.
- `MINIVAN` : Smaller-sized vans.
- `SPECIAL PURPOSE VEHICLE` : Vehicles designed for special purposes.
- `STATION WAGON - MID-SIZE` : Mid-sized station wagons.
- `PICKUP TRUCK - SMALL` : Smaller-sized pickup trucks.

This dataset can be used to understand the fuel efficiency and environmental impact of vehicles. Machine learning models can use these features to predict CO2 emissions or perform analyses comparing the fuel consumption of different vehicles.

Table of Contents

EXPLORATORY DATA ANALYSIS(EDA)

Import and Read the Dataset

In [3]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

import plotly.express as px
import scipy.stats as stats

%matplotlib inline

from scipy import stats
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler

from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge, RidgeCV
from sklearn.linear_model import Lasso, LassoCV
from sklearn.linear_model import ElasticNet, ElasticNetCV

from sklearn.model_selection import cross_val_score, cross_validate

from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error

from yellowbrick.regressor import ResidualsPlot, PredictionError

import warnings
warnings.filterwarnings("ignore")
```

In [4]:
```
co2 = pd.read_csv('co2.csv')
df = co2.copy()
```

Understanding the Data

In [5]:
```
df.head()
```

Out[5]:

	Make	Model	Vehicle Class	Engine Size(L)	Cylinders	Transmission	Fuel Type	Fuel Consumption City (L/100 km)	Fuel Consumption Hwy (L/100 km)	Consu Comb
0	ACURA	ILX	COMPACT	2.0	4	AS5	Z	9.9	6.7	
1	ACURA	ILX	COMPACT	2.4	4	M6	Z	11.2	7.7	
2	ACURA	ILX HYBRID	COMPACT	1.5	4	AV7	Z	6.0	5.8	
3	ACURA	MDX 4WD	SUV - SMALL	3.5	6	AS6	Z	12.7	9.1	
4	ACURA	RDX AWD	SUV - SMALL	3.5	6	AS6	Z	12.1	8.7	

In [6]:
```
# Display random 5 sample

df.sample(5)
```

Out[6]:

	Make	Model	Vehicle Class	Engine Size(L)	Cylinders	Transmission	Fuel Type	Fuel Consumption City (L/100 km)	Consu Hw
3495	CHEVROLET	COLORADO	PICKUP TRUCK - SMALL	2.5	4	A6	X	12.0	
5297	PORSCHE	BOXSTER	TWO-SEATER	2.0	4	AM7	Z	10.5	
4765	FORD	FLEX AWD	SUV - STANDARD	3.5	6	AS6	X	14.7	
6430	TOYOTA	C-HR	COMPACT	2.0	4	AS7	X	8.7	
5425	VOLKSWAGEN	GOLF R	COMPACT	2.0	4	AM7	Z	10.6	

In [7]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7385 entries, 0 to 7384
Data columns (total 12 columns):
 #   Column                            Non-Null Count  Dtype
---  ------                            --------------  -----
 0   Make                              7385 non-null   object
 1   Model                             7385 non-null   object
 2   Vehicle Class                     7385 non-null   object
 3   Engine Size(L)                    7385 non-null   float64
 4   Cylinders                         7385 non-null   int64
 5   Transmission                      7385 non-null   object
 6   Fuel Type                         7385 non-null   object
 7   Fuel Consumption City (L/100 km)  7385 non-null   float64
 8   Fuel Consumption Hwy (L/100 km)   7385 non-null   float64
 9   Fuel Consumption Comb (L/100 km)  7385 non-null   float64
 10  Fuel Consumption Comb (mpg)       7385 non-null   int64
 11  CO2 Emissions(g/km)               7385 non-null   int64
dtypes: float64(4), int64(3), object(5)
memory usage: 692.5+ KB
```

In [8]:
```
# Check out the missing values

missing_count = df.isnull().sum()
value_count = df.isnull().count()
missing_percentage = round(missing_count / value_count * 100, 2)
missing_df = pd.DataFrame({"count": missing_count, "percentage": missing_percentage})
missing_df
```

Out[8]:

	count	percentage
Make	0	0.0
Model	0	0.0
Vehicle Class	0	0.0
Engine Size(L)	0	0.0
Cylinders	0	0.0
Transmission	0	0.0
Fuel Type	0	0.0
Fuel Consumption City (L/100 km)	0	0.0
Fuel Consumption Hwy (L/100 km)	0	0.0
Fuel Consumption Comb (L/100 km)	0	0.0
Fuel Consumption Comb (mpg)	0	0.0
CO2 Emissions(g/km)	0	0.0

In [9]:
```
# Check out the duplicated values!!!!!!!

df.duplicated().sum()
```

Out[9]: 1103

In [10]:
```
duplicated_rows = df[df.duplicated(keep=False)]

duplicated_rows
```

Out[10]:

	Make	Model	Vehicle Class	Engine Size(L)	Cylinders	Transmission	Fuel Type	Fuel Consumption City (L/100 km)	Con Hv
4	ACURA	RDX AWD	SUV - SMALL	3.5	6	AS6	Z	12.1	
5	ACURA	RLX	MID-SIZE	3.5	6	AS6	Z	11.9	
12	ALFA ROMEO	4C	TWO-SEATER	1.8	4	AM6	Z	9.7	
13	ASTON MARTIN	DB9	MINICOMPACT	5.9	12	A6	Z	18.0	
15	ASTON MARTIN	V8 VANTAGE	TWO-SEATER	4.7	8	AM7	Z	17.4	
...	...	...	...	...	...	...	...	...	
7356	TOYOTA	Tundra	PICKUP TRUCK - STANDARD	5.7	8	AS6	X	17.7	
7365	VOLKSWAGEN	Golf GTI	COMPACT	2.0	4	M6	X	9.8	
7366	VOLKSWAGEN	Jetta	COMPACT	1.4	4	AS8	X	7.8	
7367	VOLKSWAGEN	Jetta	COMPACT	1.4	4	M6	X	7.9	
7368	VOLKSWAGEN	Jetta GLI	COMPACT	2.0	4	AM7	X	9.3	

2102 rows × 12 columns

Some duplicate rows are identical, while others have slight variations. Possible reasons for duplicates include:

1. **Data Entry Errors**: Same data entered multiple times.
2. **Different Periods**: Performance of the same model recorded over different years.
3. **Model Updates**: Comparing different versions of the same model.

Understanding the context and data collection methods is crucial to determine the cause of duplicates. Whether to drop duplicates depends on the analysis purpose:

- For unique observations, duplicates can be dropped.
- For analyzing changes over time or variations, keeping duplicates might be more appropriate.

- However, since I will be using a linear model, I do not plan to delete these rows because duplicate rows contain similar values. Linear regression tries to find the general trend of the data points and the fact that duplicate rows contain similar values does not lead to a major change in the model's predictions.

In [11]:
```
# Let's observe unique values

def get_unique_values(df):

    output_data = []

    for col in df.columns:

        # If the number of unique values in the column is less than or equal to 5
        if df.loc[:, col].nunique() <= 10:
            # Get the unique values in the column
            unique_values = df.loc[:, col].unique()
            # Append the column name, number of unique values, unique values, and data type to t
            output_data.append([col, df.loc[:, col].nunique(), unique_values, df.loc[:, col].dty
        else:
            # Otherwise, append only the column name, number of unique values, and data type to
            output_data.append([col, df.loc[:, col].nunique(),"-", df.loc[:, col].dtype])

    output_df = pd.DataFrame(output_data, columns=['Column Name', 'Number of Unique Values', ' U

    return output_df
```

In [12]:
```
get_unique_values(df)
```

Out[12]:

	Column Name	Number of Unique Values	Unique Values	Data Type
0	Make	42	-	object
1	Model	2053	-	object
2	Vehicle Class	16	-	object
3	Engine Size(L)	51	-	float64
4	Cylinders	8	[4, 6, 12, 8, 10, 3, 5, 16]	int64
5	Transmission	27	-	object
6	Fuel Type	5	[Z, D, X, E, N]	object
7	Fuel Consumption City (L/100 km)	211	-	float64
8	Fuel Consumption Hwy (L/100 km)	143	-	float64
9	Fuel Consumption Comb (L/100 km)	181	-	float64
10	Fuel Consumption Comb (mpg)	54	-	int64
11	CO2 Emissions(g/km)	331	-	int64

- The columns Engine Size(L), Cylinders, Fuel Consumption City (L/100 km), Fuel Consumption Hwy (L/100 km), Fuel Consumption Comb (L/100 km), Fuel Consumption Comb (mpg) and CO2 Emissions(g/km) are numerical and continuous in nature.
- The columns Make, Model, Vehicle Class, Transmission and Fuel Type are categorical in nature.

In [13]:
```
# Basic statistics summary of Numerical features

df.describe().T
```

Out[13]:

	count	mean	std	min	25%	50%	75%	max
Engine Size(L)	7385.0	3.160068	1.354170	0.9	2.0	3.0	3.7	8.4
Cylinders	7385.0	5.615030	1.828307	3.0	4.0	6.0	6.0	16.0
Fuel Consumption City (L/100 km)	7385.0	12.556534	3.500274	4.2	10.1	12.1	14.6	30.6
Fuel Consumption Hwy (L/100 km)	7385.0	9.041706	2.224456	4.0	7.5	8.7	10.2	20.6
Fuel Consumption Comb (L/100 km)	7385.0	10.975071	2.892506	4.1	8.9	10.6	12.6	26.1
Fuel Consumption Comb (mpg)	7385.0	27.481652	7.231879	11.0	22.0	27.0	32.0	69.0
CO2 Emissions(g/km)	7385.0	250.584699	58.512679	96.0	208.0	246.0	288.0	522.0

In [14]:
```
# Basic statistics summary of Object features

df.describe(include= 'object').T
```

Out[14]:

	count	unique	top	freq
Make	7385	42	FORD	628
Model	7385	2053	F-150 FFV 4X4	32
Vehicle Class	7385	16	SUV - SMALL	1217
Transmission	7385	27	AS6	1324
Fuel Type	7385	5	X	3637

Rename the Columns

In [15]:
```
df.rename(columns={ 'Make': 'make',
                    'Model': 'model',
                    'Vehicle Class': 'vehicle_class',
                    'Engine Size(L)': 'engine_size',
                    'Cylinders': 'cylinders',
                    'Transmission': 'transmission',
                    'Fuel Type': 'fuel_type',
                    'Fuel Consumption City (L/100 km)': 'fuel_cons_city',
                    'Fuel Consumption Hwy (L/100 km)': 'fuel_cons_hwy',
                    'Fuel Consumption Comb (L/100 km)': 'fuel_cons_comb',
                    'Fuel Consumption Comb (mpg)': 'fuel_cons_comb_mpg',
                    'CO2 Emissions(g/km)': 'co2'
                    }, inplace=True)
```

In [16]:
```
df.columns
```

Out[16]:
```
Index(['make', 'model', 'vehicle_class', 'engine_size', 'cylinders',
       'transmission', 'fuel_type', 'fuel_cons_city', 'fuel_cons_hwy',
       'fuel_cons_comb', 'fuel_cons_comb_mpg', 'co2'],
      dtype='object')
```

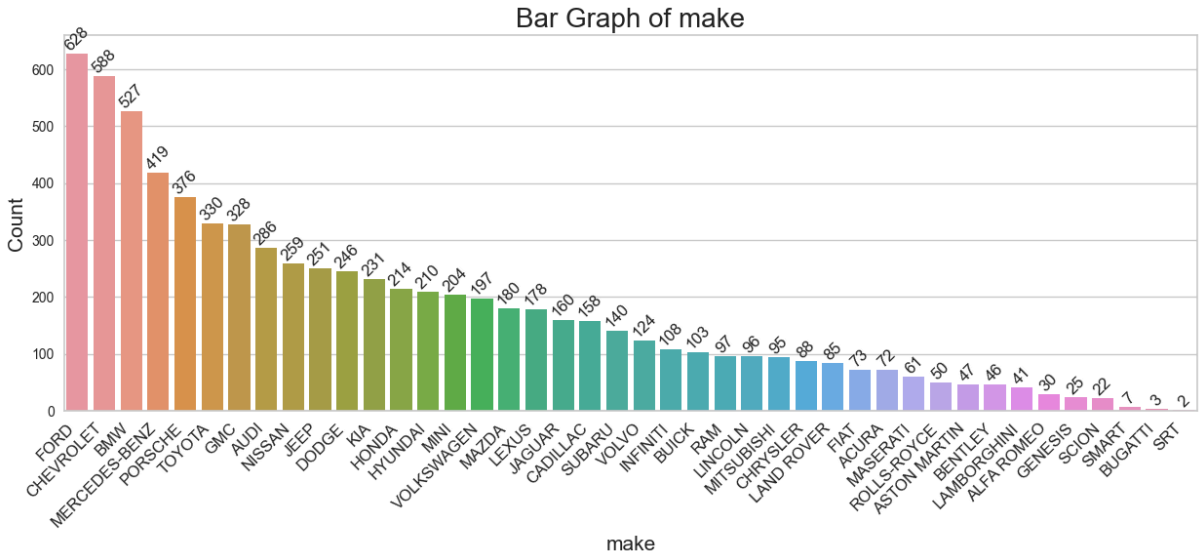
Data Visualisation

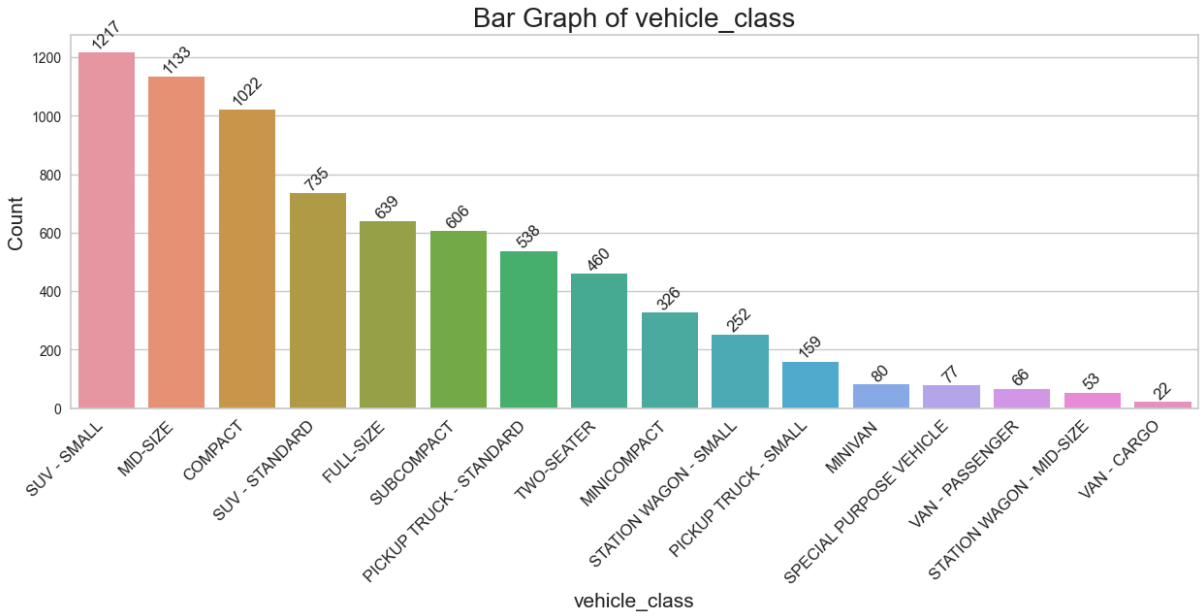
Categorical Features

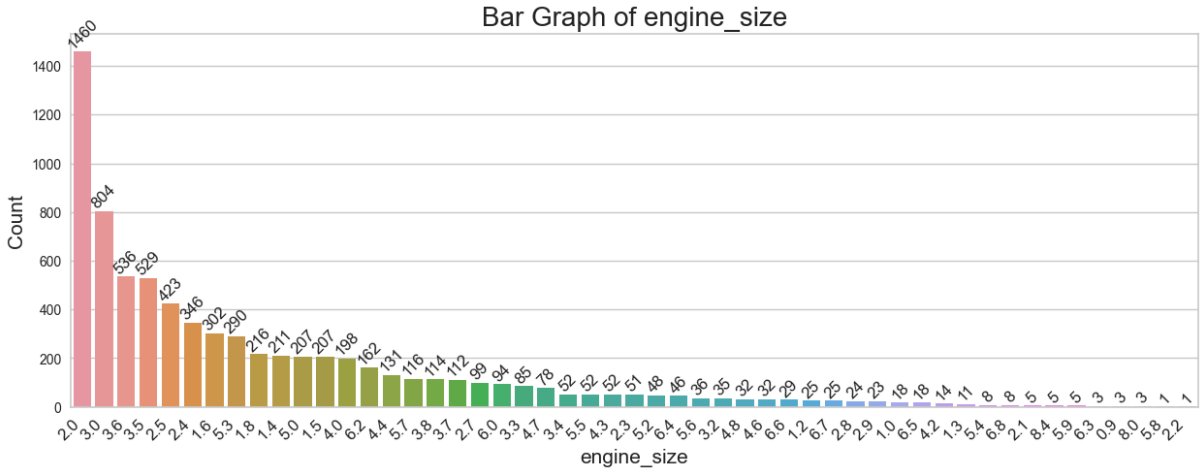
Distribution of Categorical Features

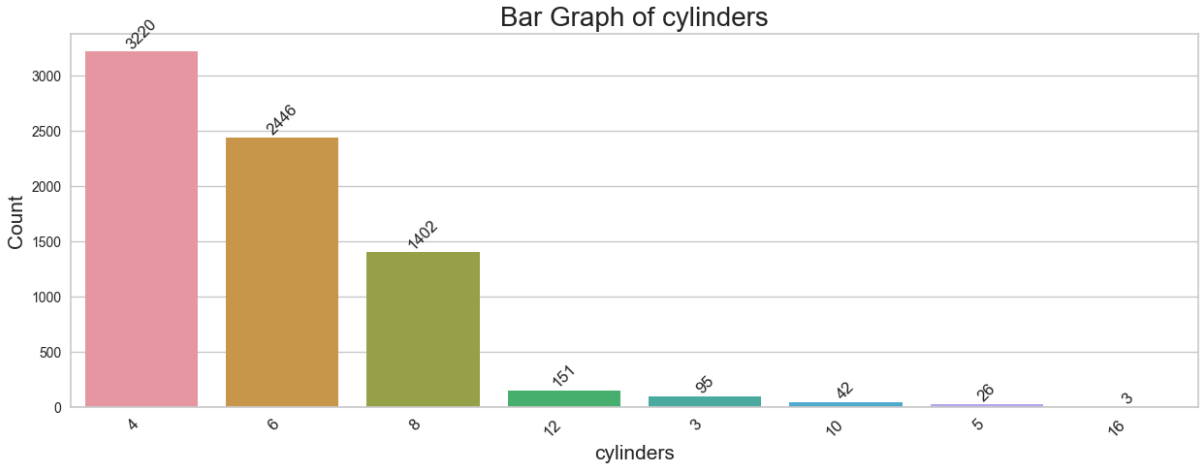
In [17]:
```
import plotly.graph_objects as go
import plotly.express as px
```

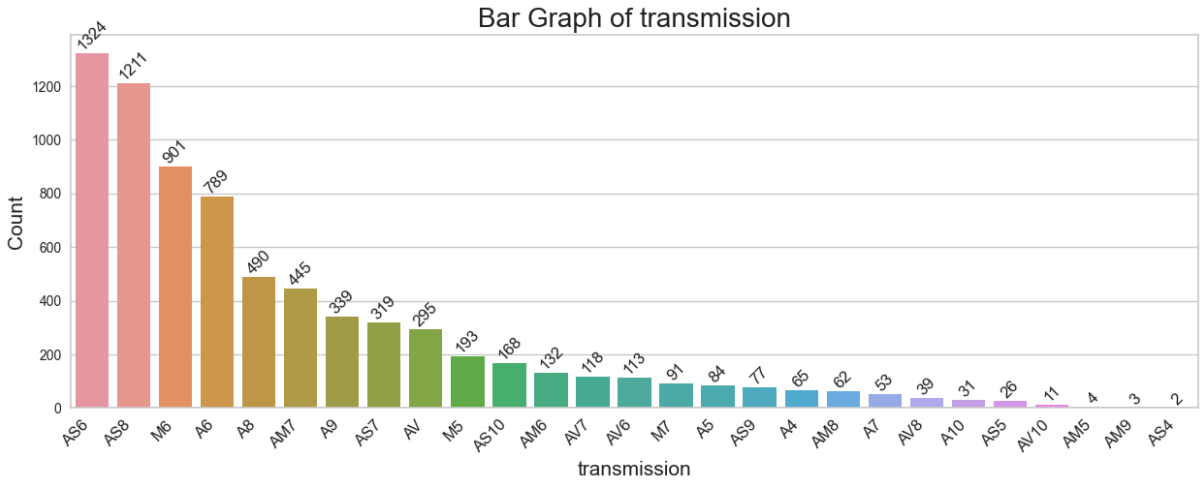
In [18]:
```
# Let's look at the distribution of our categorical characteristics with a bar graph

def plot_bar_graphs(df, columns):
    for column in columns:
        plt.figure(figsize=(15, 5))
        ax = sns.countplot(x=column, data=df, order=df[column].value_counts().index)
        ax.bar_label(ax.containers[0],rotation=45)
        plt.xlabel(column, fontsize=15)
        plt.ylabel('Count', fontsize=15)
        plt.title(f'Bar Graph of {column}', fontsize=20)
        plt.xticks(rotation=45, ha='right', fontsize=12)
        plt.show()

cat_features = ['make','vehicle_class', 'engine_size', 'cylinders', 'transmission', 'fuel_type']

plot_bar_graphs(df, cat_features)
```

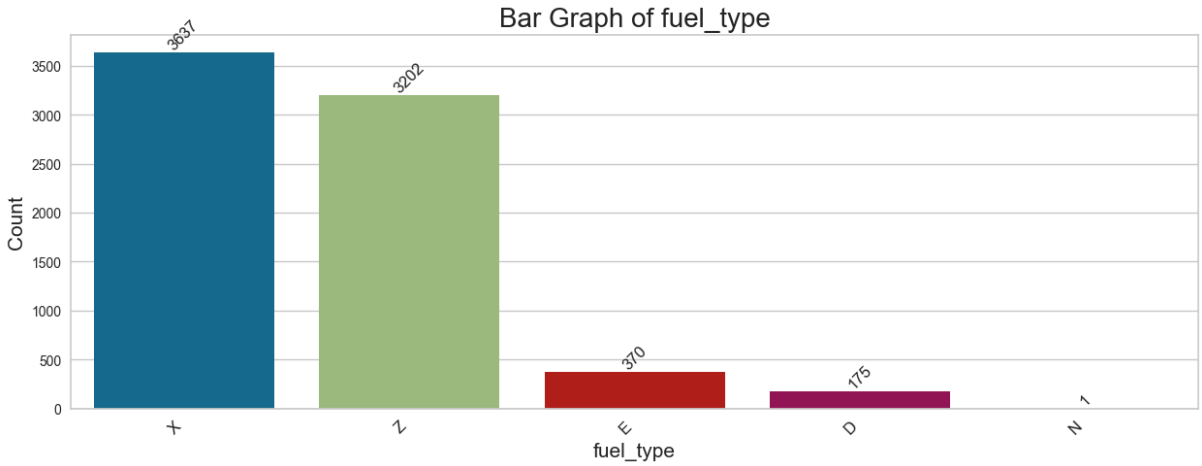



Bar Graph of vehicle_class
Count
vehicle_class
SUV - SMALL
MID-SIZE
COMPACT
SUV - STANDARD
FULL-SIZE
SUBCOMPACT
PICKUP TRUCK - STANDARD
TWO-SEATER
MINICOMPACT
STATION WAGON - SMALL
PICKUP TRUCK - SMALL
MINIVAN
SPECIAL PURPOSE VEHICLE
VAN - PASSENGER
STATION WAGON - MID-SIZE
VAN - CARGO
1217
1133
1022
735
639
606
538
460
326
252
159
80
77
66
53
22


Bar Graph of engine_size
Count
engine_size
1460
804
536
529
423
346
302
290
216
211
207
207
198
162
131
116
114
112
99
94
85
78
52
52
52
51
48
46
36
35
32
32
29
25
25
24
23
18
18
14
11
8
8
5
5
5
3
3
3
1
1


Bar Graph of cylinders
Count
cylinders
4
6
8
12
3
10
5
16
3220
2446
1402
151
95
42
26
3

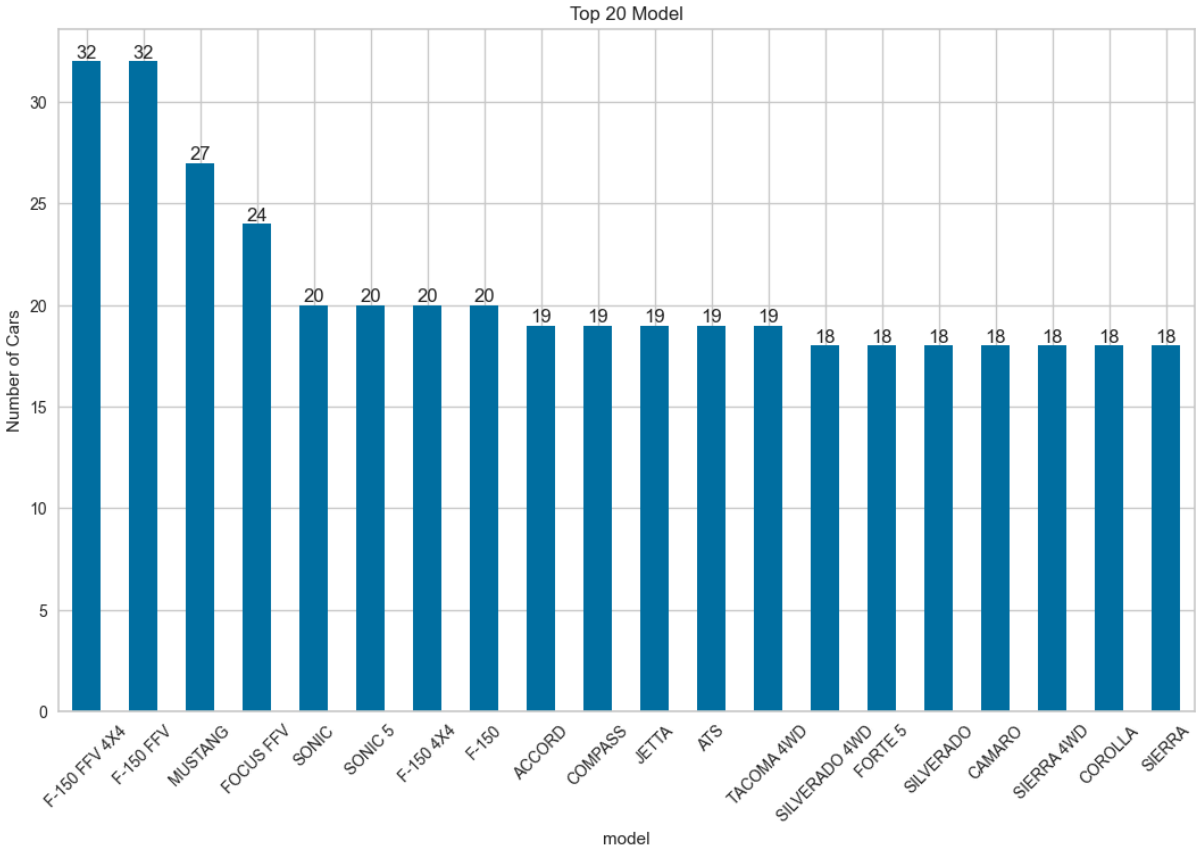




In [19]:
```
# Model Feature

df.model.unique()
```

Out[19]:
```
array(['ILX', 'ILX HYBRID', 'MDX 4WD', ...,
       'Tacoma 4WD D-Cab TRD Off-Road/Pro', 'Atlas Cross Sport 4MOTION',
       'XC40 T4 AWD'], dtype=object)
```

In [20]:
```
fig = plt.figure(figsize = (10,6))
ax = fig.add_axes([0,0,1,1])
counts = df.model.value_counts().sort_values(ascending=False).head(20)
counts.plot(kind = "bar")
plt.title('Top 20 Model')
plt.xlabel('model')
plt.ylabel('Number of Cars')
plt.xticks(rotation = 45)
ax.bar_label(ax.containers[0], labels=counts.values, fontsize=12);
```

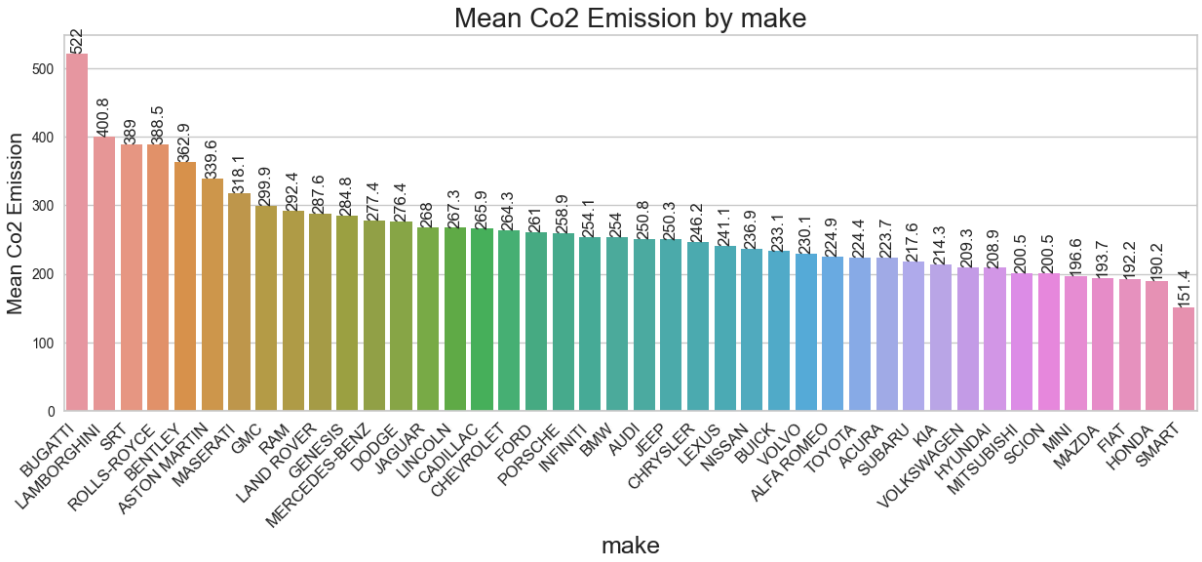


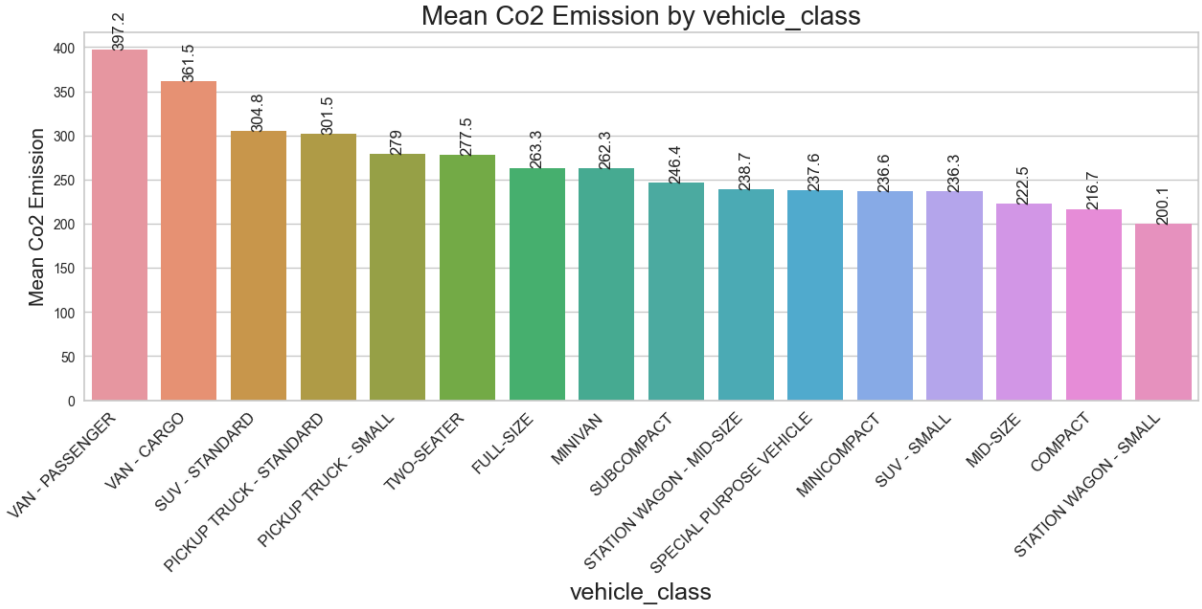
As can be seen from the graphs above:

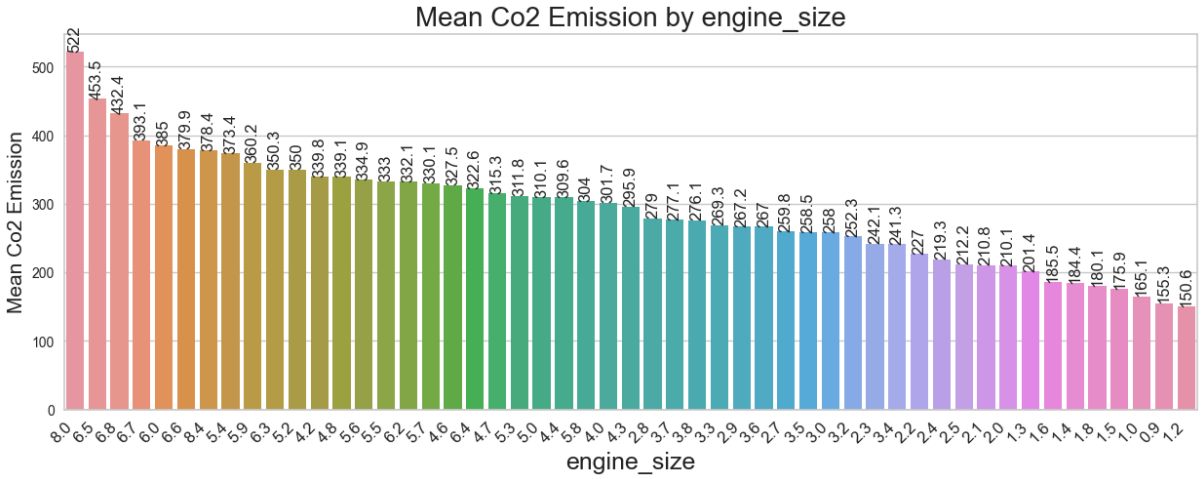
- The number of vehicles consuming diesel, ethanol and natural gas fuel in the data set is very small.
- Widespread use of AS6, AS8, M6, A6, A9 as transmission options
- 4, 6, 8 are commonly used as cylinders option
- Engine Size (L) with 2.0 and 3.0 options in density
- The dataset is generally dominated by smaller sized vehicles

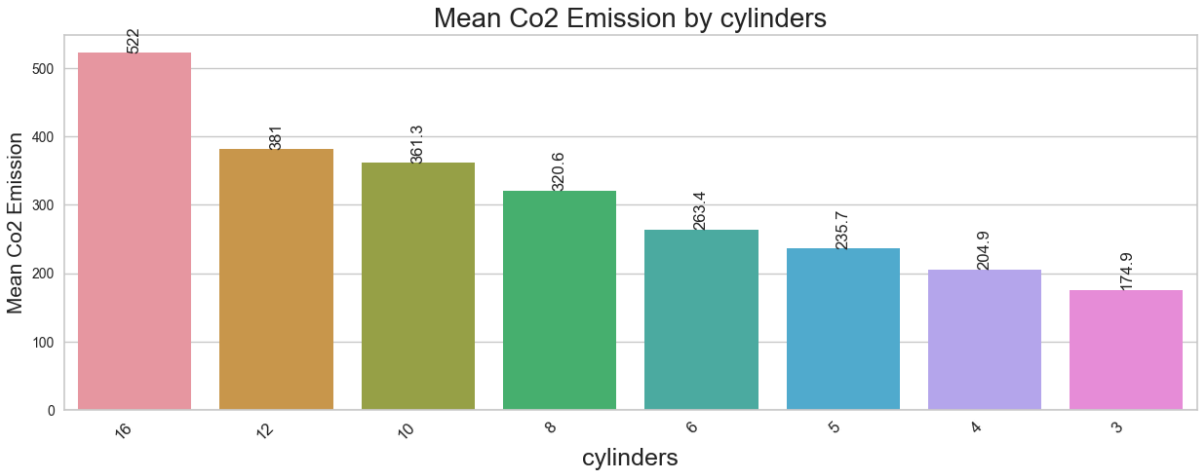
Target Variable vs Categorical Features

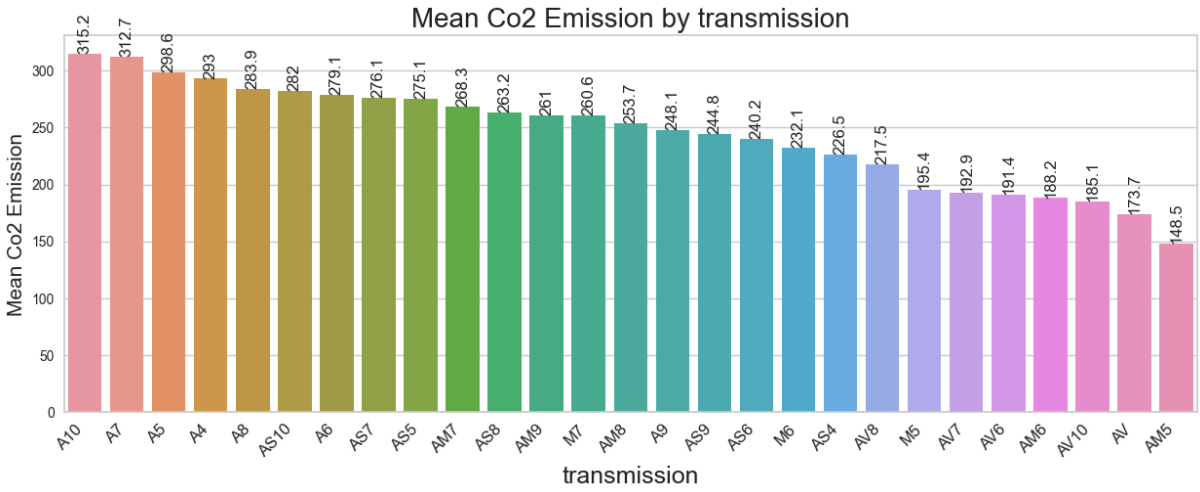
In [21]:
```
# Let's look at the relationship between our categorical attributes and the target variable

def plot_bar_with_co2(df, columns):
    for column in columns:
        plt.figure(figsize=(15, 5))
        grouped_data = df.groupby(column)['co2'].mean().round(1).reset_index()
        grouped_data_sorted = grouped_data.sort_values(by='co2', ascending=False)
        ax = sns.barplot(x=column, y='co2', data=grouped_data_sorted, order=grouped_data_sorted[
        ax.bar_label(ax.containers[0],rotation=90)
        plt.xlabel(column, fontsize=18)
        plt.ylabel('Mean Co2 Emission', fontsize=15)
        plt.title(f'Mean Co2 Emission by {column}', fontsize=20)
        plt.xticks(rotation=45, ha='right', fontsize=12)
        plt.show()

plot_bar_with_co2(df, cat_features)
```

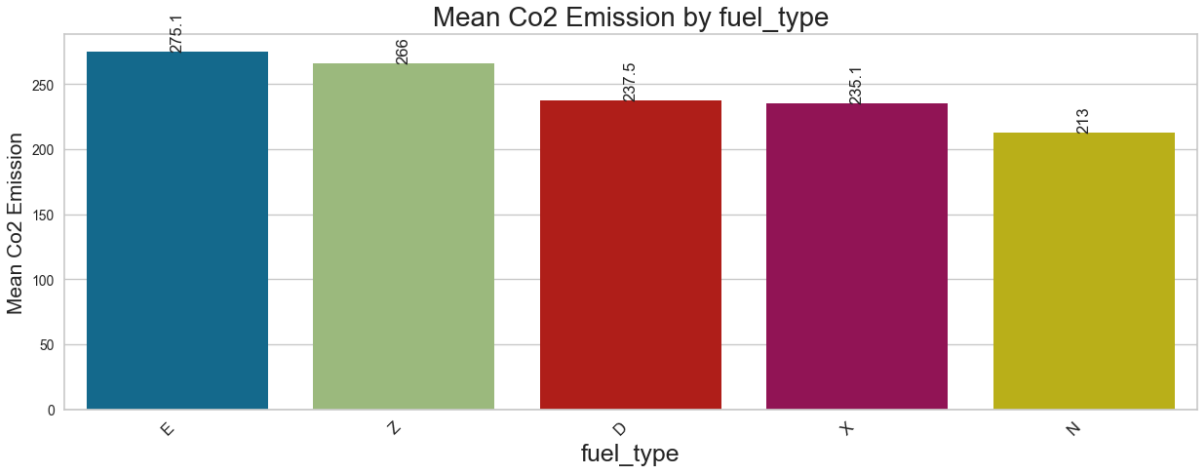

Mean Co2 Emission by make
Mean Co2 Emission
make
BUGATTI 522
LAMBORGHINI 400.8
SRT 389
ROLLS-ROYCE 388.5
BENTLEY 362.9
ASTON MARTIN 339.6
MASERATI 318.1
GMC 299.9
RAM 292.4
LAND ROVER 287.6
GENESIS 284.8
MERCEDES-BENZ 277.4
DODGE 276.4
JAGUAR 268
LINCOLN 267.3
CADILLAC 265.9
CHEVROLET 264.3
FORD 261
PORSCHE 258.9
INFINITI 254.1
BMW 254
AUDI 250.8
JEEP 250.3
CHRYSLER 246.2
LEXUS 241.1
NISSAN 236.9
BUICK 233.1
VOLVO 230.1
ALFA ROMEO 224.9
TOYOTA 224.4
ACURA 223.7
SUBARU 217.6
KIA 214.3
VOLKSWAGEN 209.3
HYUNDAI 208.9
MITSUBISHI 200.5
SCION 200.5
MINI 196.6
MAZDA 193.7
FIAT 192.2
HONDA 190.2
SMART 151.4


Mean Co2 Emission by vehicle_class
Mean Co2 Emission
vehicle_class
VAN - PASSENGER 397.2
VAN - CARGO 361.5
SUV - STANDARD 304.8
PICKUP TRUCK - STANDARD 301.5
PICKUP TRUCK - SMALL 279
TWO-SEATER 277.5
FULL-SIZE 263.3
MINIVAN 262.3
SUBCOMPACT 246.4
STATION WAGON - MID-SIZE 238.7
SPECIAL PURPOSE VEHICLE 237.6
MINICOMPACT 236.6
SUV - SMALL 236.3
MID-SIZE 222.5
COMPACT 216.7
STATION WAGON - SMALL 200.1


Mean Co2 Emission by engine_size
Mean Co2 Emission
engine_size
8.0 522
6.5 453.5
6.8 432.4
6.7 393.1
6.0 385
6.6 379.9
8.4 378.4
5.4 373.4
5.9 360.2
6.3 350.3
5.2 350
4.2 339.8
4.8 339.1
5.6 334.9
5.5 333
6.2 332.1
5.7 330.1
4.6 327.5
6.4 322.6
4.7 315.3
5.3 311.8
5.0 310.1
4.4 309.6
5.8 304
4.0 301.7
4.3 295.9
2.8 279
3.7 277.1
3.8 276.1
3.3 269.3
2.9 267.2
3.6 267
2.7 259.8
3.5 258.5
3.0 258
3.2 252.3
2.3 242.1
3.4 241.3
2.2 227
2.4 219.3
2.5 212.2
2.1 210.8
2.0 210.1
1.3 201.4
1.6 185.5
1.4 184.4
1.8 180.1
1.5 175.9
1.0 165.1
0.9 155.3
1.2 150.6







As can be seen from the graphs above:

- Bugatti has the highest average Co2 emissions
- Large-volume vehicles have high CO2 emission averages
- C02 emission averages of high volume and cylinders engines are also high
- Ethanol is the fuel with the highest average CO2

ANOVA Test for Categorical Features

In [22]:
```
# Perform ANOVA test for each categorical feature
anova_results = {}
categorical_features = df.select_dtypes(include=['object']).columns

for feature in categorical_features:
    groups = [df["co2"][df[feature] == category].values for category in df[feature].unique()]
```

```
    anova_results[feature] = stats.f_oneway(*groups)

# Display the ANOVA results
for feature, result in anova_results.items():
    print(f"ANOVA result for {feature}:")
    print(f"F-statistic: {result.statistic}, p-value: {result.pvalue}")
    print()
```

```
ANOVA result for make:
F-statistic: 106.8000265413262, p-value: 0.0

ANOVA result for model:
F-statistic: 58.405091462865016, p-value: 0.0

ANOVA result for vehicle_class:
F-statistic: 266.0228094521597, p-value: 0.0

ANOVA result for transmission:
F-statistic: 103.70394951088048, p-value: 0.0

ANOVA result for fuel_type:
F-statistic: 148.94555963595639, p-value: 1.062810397301377e-122
```

NOTE:

- The p-values for each of the Make, Model, Vehicle Class, Transmission, and Fuel Type variables are much smaller than 0.05, indicating that these variables create statistically significant differences in `co2_emissions`.

Label Encoding the Categorical Features

In [23]:

```
from sklearn.preprocessing import LabelEncoder

# Copy the original dataframe to avoid modifying it directly
df_labeled = df.copy()

# List of categorical columns
categorical_columns = ['make', 'model', 'vehicle_class', 'transmission', 'fuel_type']

# Apply Label Encoding to each categorical column
label_encoders = {}
for column in categorical_columns:
    le = LabelEncoder()
    df_labeled[column] = le.fit_transform(df_labeled[column])
    label_encoders[column] = le

# Display the first few rows of the labeled dataframe
print(df_labeled.head())
```

```
   make  model  vehicle_class  engine_size  cylinders  transmission  \
0     0   1057              0          2.0          4            14
1     0   1057              0          2.4          4            25
2     0   1058              0          1.5          4            22
3     0   1233             11          3.5          6            15
4     0   1499             11          3.5          6            15

   fuel_type  fuel_cons_city  fuel_cons_hwy  fuel_cons_comb  \
0          4             9.9            6.7             8.5
1          4            11.2            7.7             9.6
2          4             6.0            5.8             5.9
3          4            12.7            9.1            11.1
4          4            12.1            8.7            10.6

   fuel_cons_comb_mpg  co2
0                  33  196
1                  29  221
2                  48  136
3                  25  255
4                  27  244
```

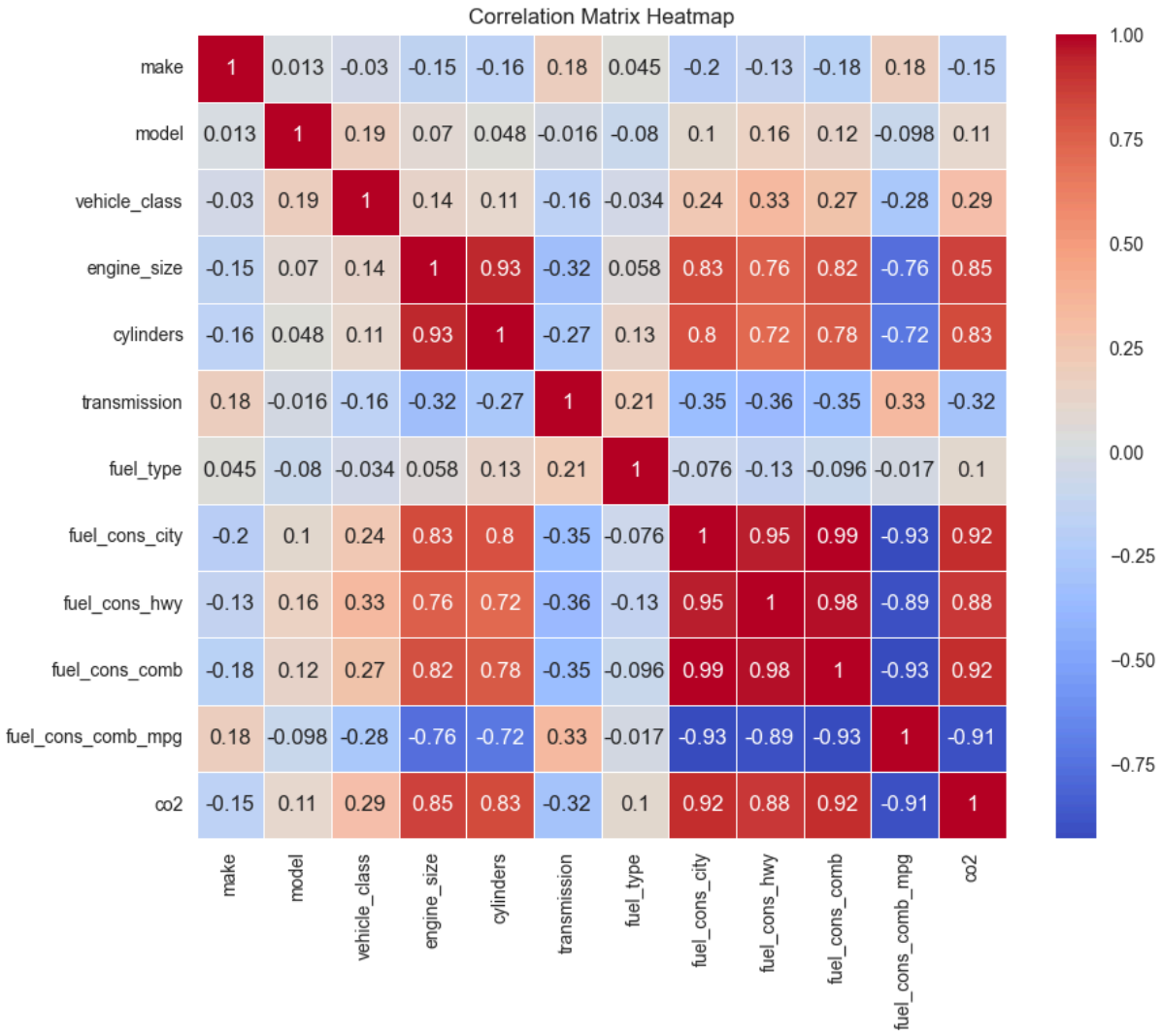
In [24]:
```
df_labeled.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7385 entries, 0 to 7384
Data columns (total 12 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   make                7385 non-null   int32
 1   model               7385 non-null   int32
 2   vehicle_class       7385 non-null   int32
 3   engine_size         7385 non-null   float64
 4   cylinders           7385 non-null   int64
 5   transmission        7385 non-null   int32
 6   fuel_type           7385 non-null   int32
 7   fuel_cons_city      7385 non-null   float64
 8   fuel_cons_hwy       7385 non-null   float64
 9   fuel_cons_comb      7385 non-null   float64
 10  fuel_cons_comb_mpg  7385 non-null   int64
 11  co2                 7385 non-null   int64
dtypes: float64(4), int32(5), int64(3)
memory usage: 548.2 KB
```

In [25]:
```
#df = df_labeled.copy()
```

Correlation Matrix

In [26]:
```
correlation_matrix = df_labeled.corr()

plt.figure(figsize=(10,8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Matrix Heatmap')
plt.show()
```

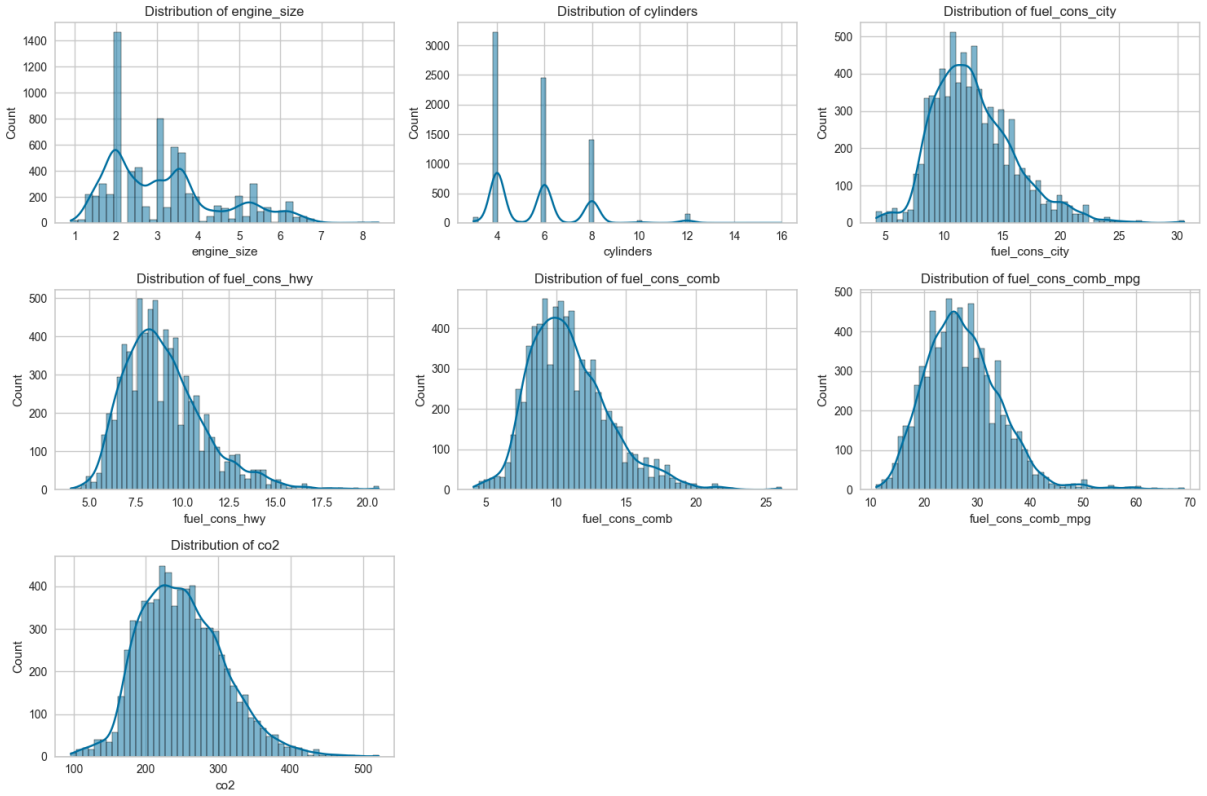


- The correlation matrix shows that features like `engine_size`, `cylinders`, and fuel consumption metrics (city, highway, combined) have strong positive correlations with `co2_emissions`.
- Notably, `fuel_cons_mpg` has a strong negative correlation with CO2 emissions, indicating that higher fuel efficiency results in lower emissions.
- From a multicollinearity perspective, `fuel_cons_city`, `fuel_cons_hwy`, and `fuel_cons_comb` are highly intercorrelated, suggesting potential redundancy.
  - It may be beneficial to select just one or combine them into a single metric to avoid multicollinearity.
- Key predictors for a CO2 emissions model include `engine_size`, `cylinders`, and fuel consumption metrics.
- Categorical features such as `make`, `model`, `vehicle_class`, `transmission`, and `fuel_type` are also important due to their significant impact on CO2 emissions.
  - Managing multicollinearity among these highly correlated features is crucial to ensure model stability and performance.

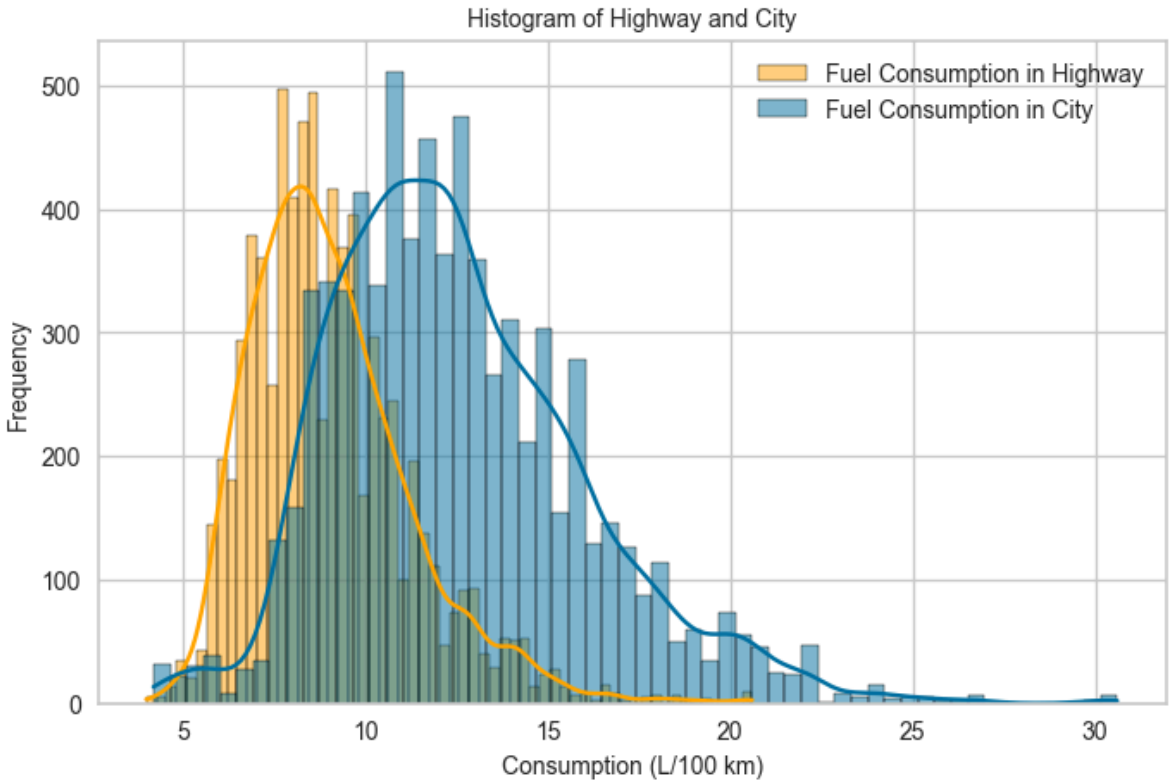
Numerical Features

Distribution of Numerical Features

In [27]:
```
numerical_df = df.select_dtypes(include=['number'])

plt.figure(figsize=(15, 10))

num_vars = len(numerical_df.columns)

for i, var in enumerate(numerical_df.columns, 1):
    plt.subplot((num_vars // 3) + 1, 3, i)
    sns.histplot(data=df, x=var, kde=True)
    plt.title(f'Distribution of {var}')

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



In [28]:
```
# Consumption of Highway and City

plt.figure(figsize=(8, 5))
sns.histplot(data=df, x="fuel_cons_hwy", kde=True, label = "Fuel Consumption in Highway",color =
sns.histplot(data=df, x="fuel_cons_city", kde=True, label = "Fuel Consumption in City")
plt.xlabel('Consumption (L/100 km)', fontsize=10)
plt.ylabel('Frequency', fontsize=10)
plt.title(f'Histogram of Highway and City', fontsize=10)
plt.legend()
plt.show()
```



1. **Engine Size (engine_size):** The distribution is skewed with several peaks, indicating different types of vehicles with varying engine sizes.
2. **Cylinders:** The distribution is concentrated around specific values (4, 6, 8), reflecting common engine types in the dataset.
3. **Fuel Consumption City (fuel_cons_city):** The distribution is approximately normal, which is beneficial for model learning.
4. **Fuel Consumption Hwy (fuel_cons_hwy):** Similar to city fuel consumption, this also shows an approximately normal distribution.
5. **Fuel Consumption Combined (fuel_cons_comb):** The combined fuel consumption distribution is normal, making it a useful variable for modeling.
6. **Fuel Consumption MPG (fuel_cons_mpg):** This shows a normal distribution and has an inverse relationship with other fuel consumption metrics, which is expected.
7. **CO2 Emissions (co2):** The distribution is nearly normal, which is advantageous for regression models.

Overall Evaluation:

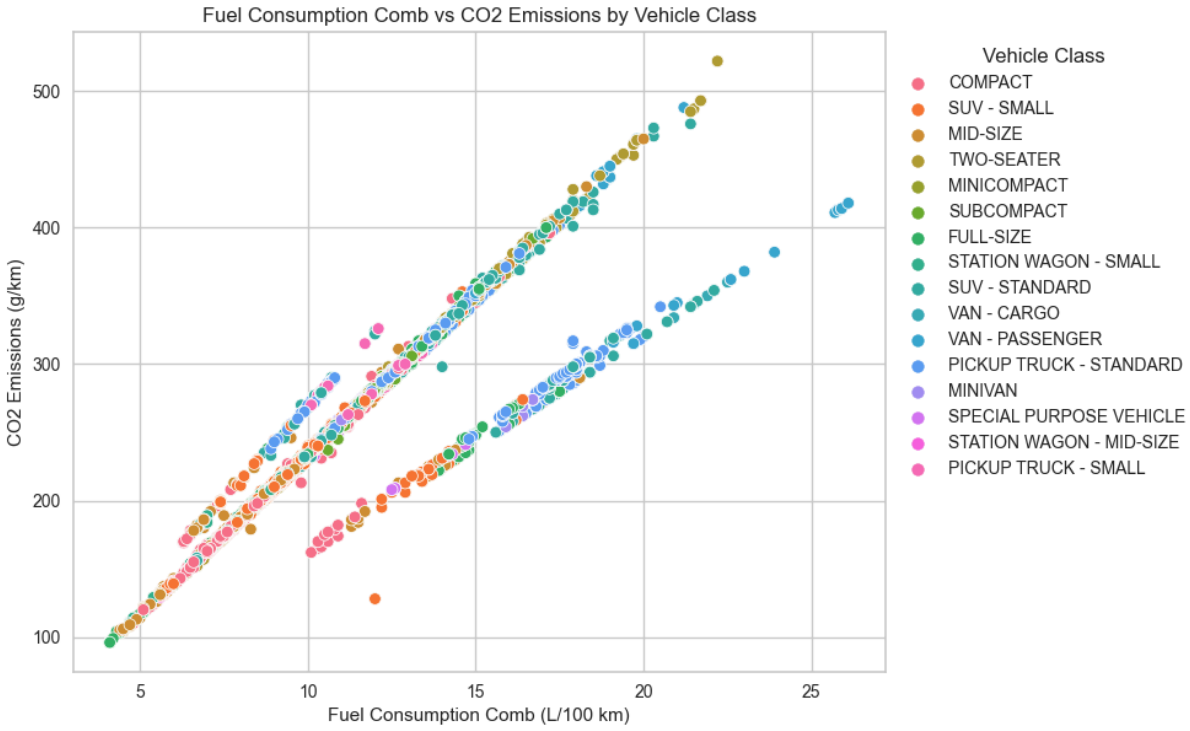
- Most features exhibit normal or skew-normal distributions, which are suitable for modeling.
- Consider multicollinearity, especially among `fuel_cons_City`, `fuel_cons_Hwy`, and `fuel_cons_Comb`.
- Features like `cylinders`, `engine_size`, and `fuel_cons_mpg` provide important information about vehicle performance and efficiency, making them valuable for the model.

Target Variable vs Numerical Features

In [29]:
```
# Target vs Fuel Consumption Combined (city+hwy)
# Hue: Vehicle Class

plt.figure(figsize=(10,6))
sns.scatterplot(data=df,x='fuel_cons_comb',y='co2',hue='vehicle_class')
plt.legend(bbox_to_anchor=(1, 1), loc='upper left', title='Vehicle Class')
plt.tight_layout()

plt.title('Fuel Consumption Comb vs CO2 Emissions by Vehicle Class')
```

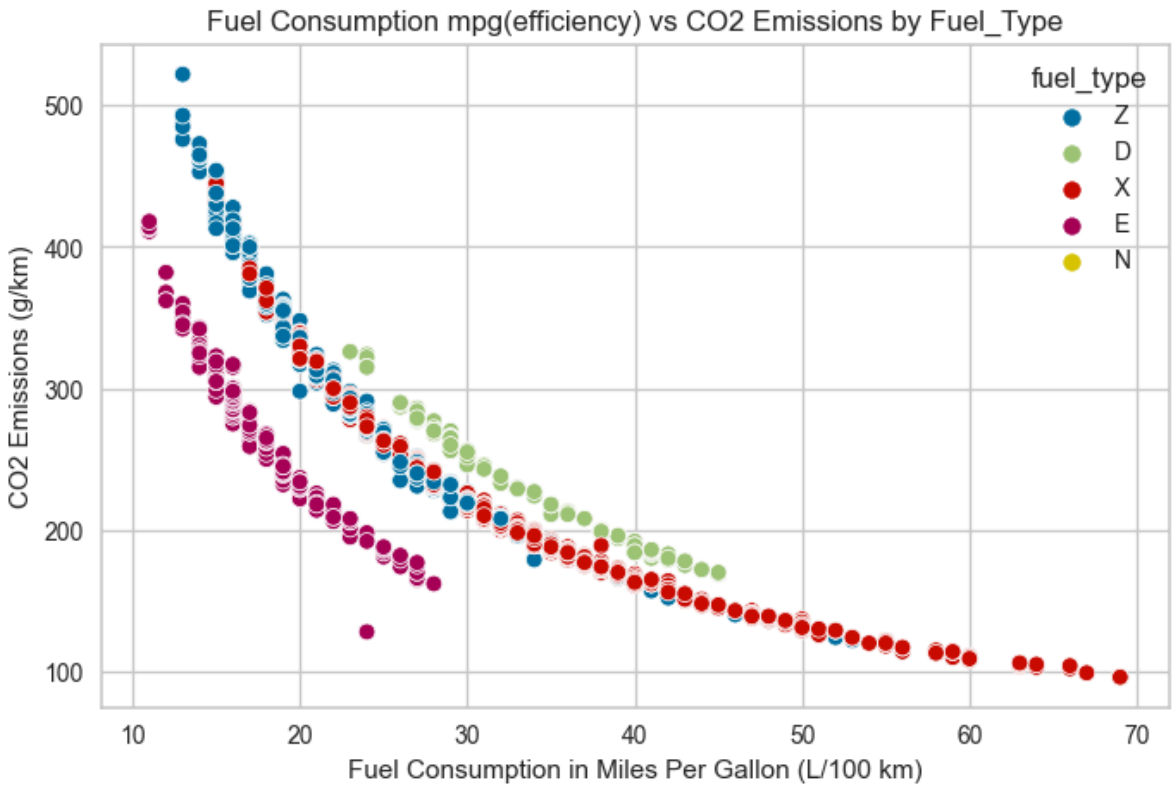
```
plt.xlabel('Fuel Consumption Comb (L/100 km)')
plt.ylabel('CO2 Emissions (g/km)')

# Vehicle class makes impact on overall fuel efficiency of vehicle as well lead to more emission
# Two-Seater,Mid Size, Passenger van, Cargo van, Pick Up Truck have most emission with lowest fu
# Mini Van, SPV, Station Wagon are more fuel efficienct with low emission
```

Out[29]: Text(86.47222222222221, 0.5, 'CO2 Emissions (g/km)')



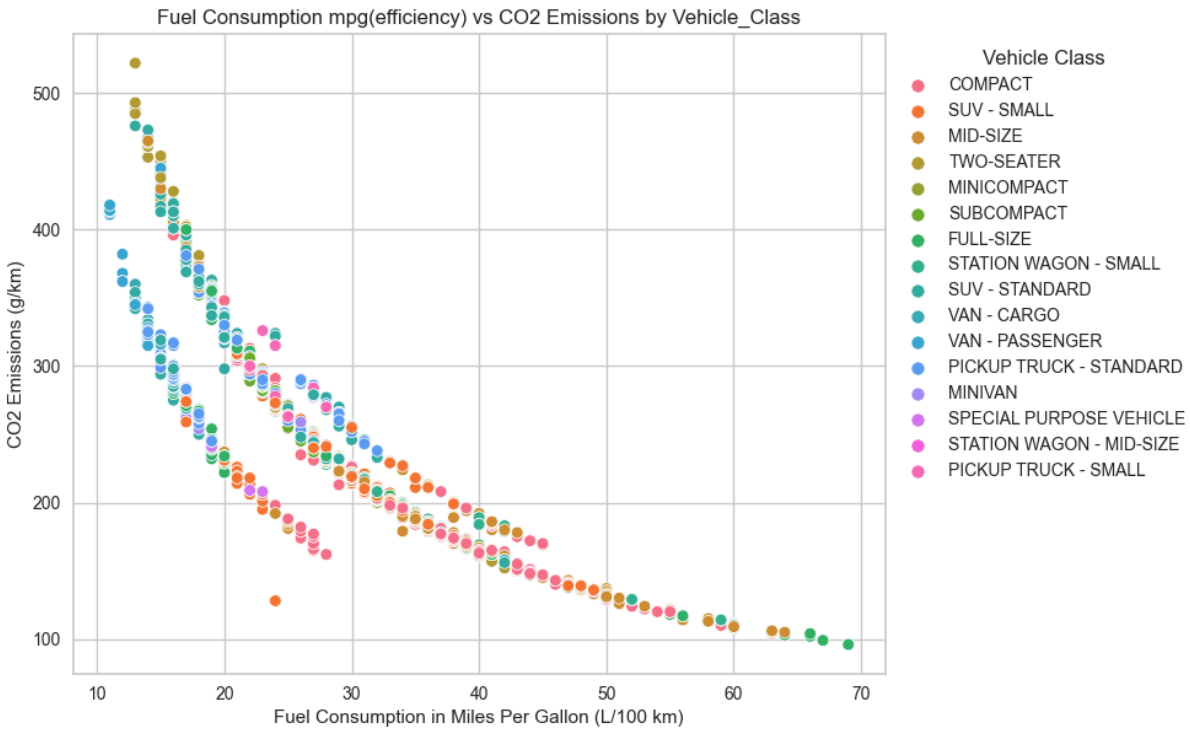
In [30]:
```
# Target vs Fuel Consumption in Miles Per Gallon (mpg)
# Hue: Fuel Type

plt.figure(figsize=(8,5))
sns.scatterplot(data=df,x='fuel_cons_comb_mpg',y='co2',hue='fuel_type')

plt.title('Fuel Consumption mpg(efficiency) vs CO2 Emissions by Fuel_Type')
plt.xlabel('Fuel Consumption in Miles Per Gallon (L/100 km)')
plt.ylabel('CO2 Emissions (g/km)')
```

Out[30]: Text(0, 0.5, 'CO2 Emissions (g/km)')

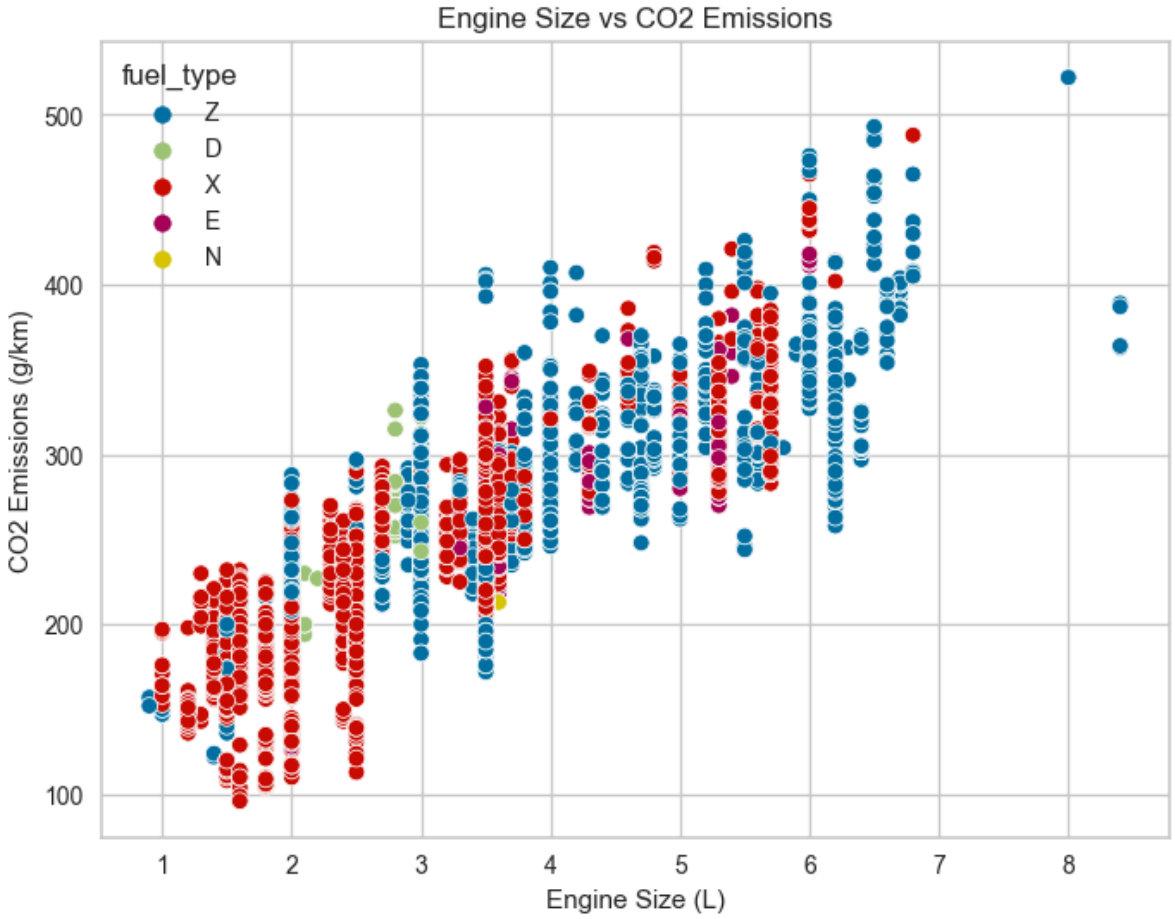


In [31]:
```
# Target vs Fuel Consumption in Miles Per Gallon
# Hue: Vehicle Class

plt.figure(figsize=(10,6))
sns.scatterplot(data=df, x='fuel_cons_comb_mpg',y='co2',hue='vehicle_class')
plt.legend(bbox_to_anchor=(1, 1), loc='upper left', title='Vehicle Class')
plt.tight_layout()

plt.title('Fuel Consumption mpg(efficiency) vs CO2 Emissions by Vehicle_Class')
plt.xlabel('Fuel Consumption in Miles Per Gallon (L/100 km)')
plt.ylabel('CO2 Emissions (g/km)')
```

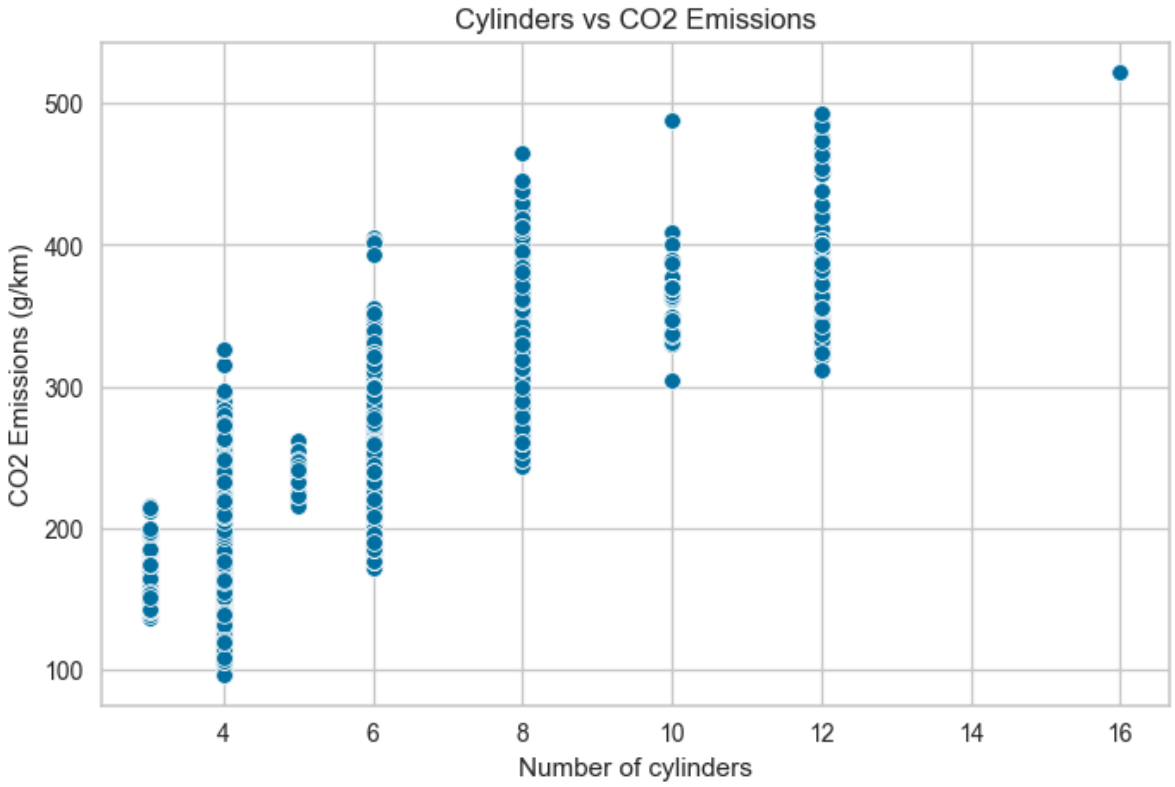
Out[31]:
```
Text(86.47222222222221, 0.5, 'CO2 Emissions (g/km)')
```



In [32]:
```
# Target vs Engine Size

plt.figure(figsize=(8, 6))
sns.scatterplot(x='engine_size', y='co2', data=df, hue='fuel_type')
plt.title('Engine Size vs CO2 Emissions')
plt.xlabel('Engine Size (L)')
plt.ylabel('CO2 Emissions (g/km)')
plt.show()
```

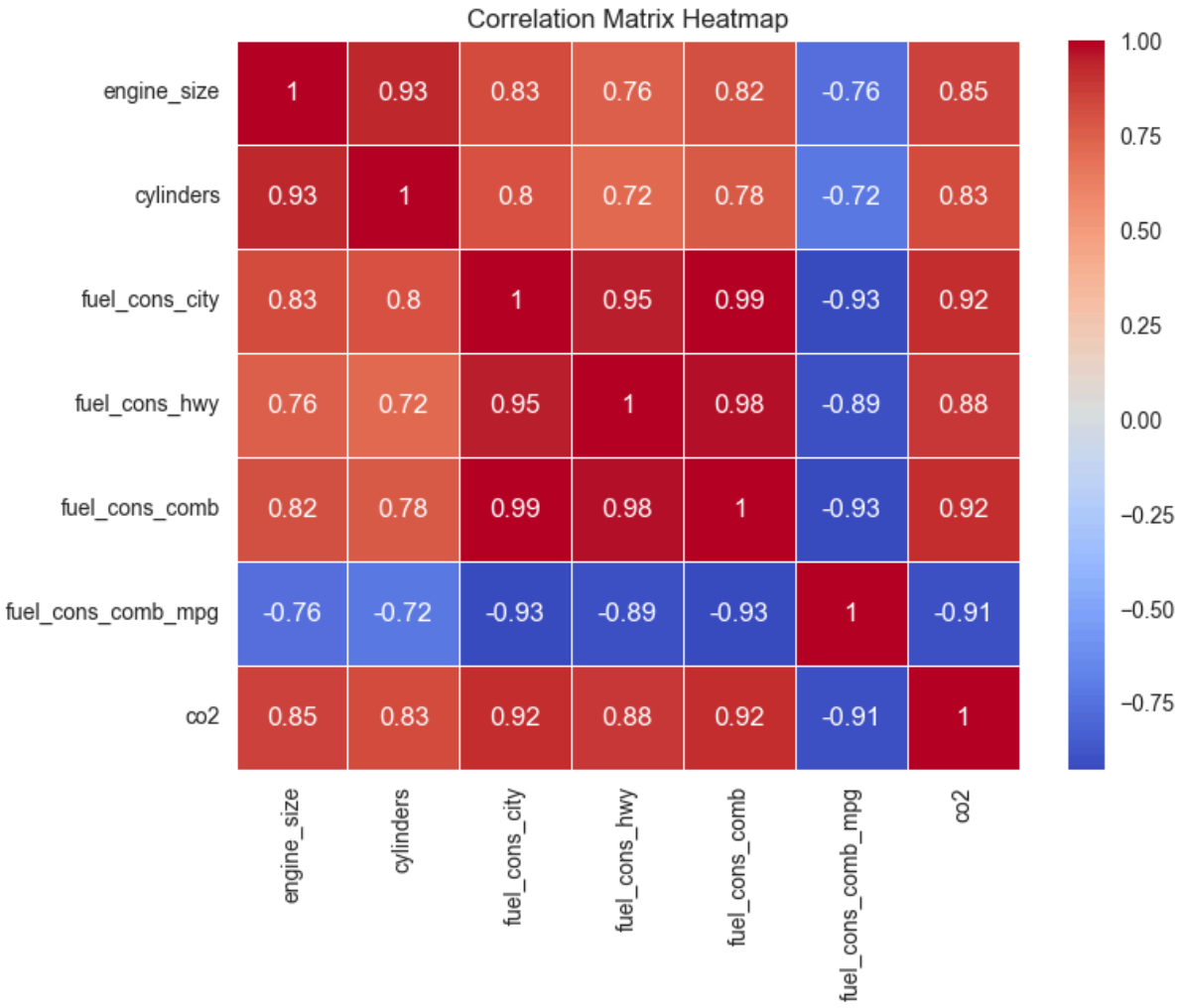


In [33]:
```
# Target vs Cylinders Number

plt.figure(figsize=(8,5))
sns.scatterplot(data=df,x='cylinders',y='co2')

plt.title('Cylinders vs CO2 Emissions')
plt.xlabel('Number of cylinders')
plt.ylabel('CO2 Emissions (g/km)')
plt.show()
# Vehicles with more cylinders tend to have higher emission
```



Correlations of Numerical Features

In [34]:
```
correlation_matrix = df.corr(numeric_only=True)

plt.figure(figsize=(8,6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Matrix Heatmap')
plt.show()
```



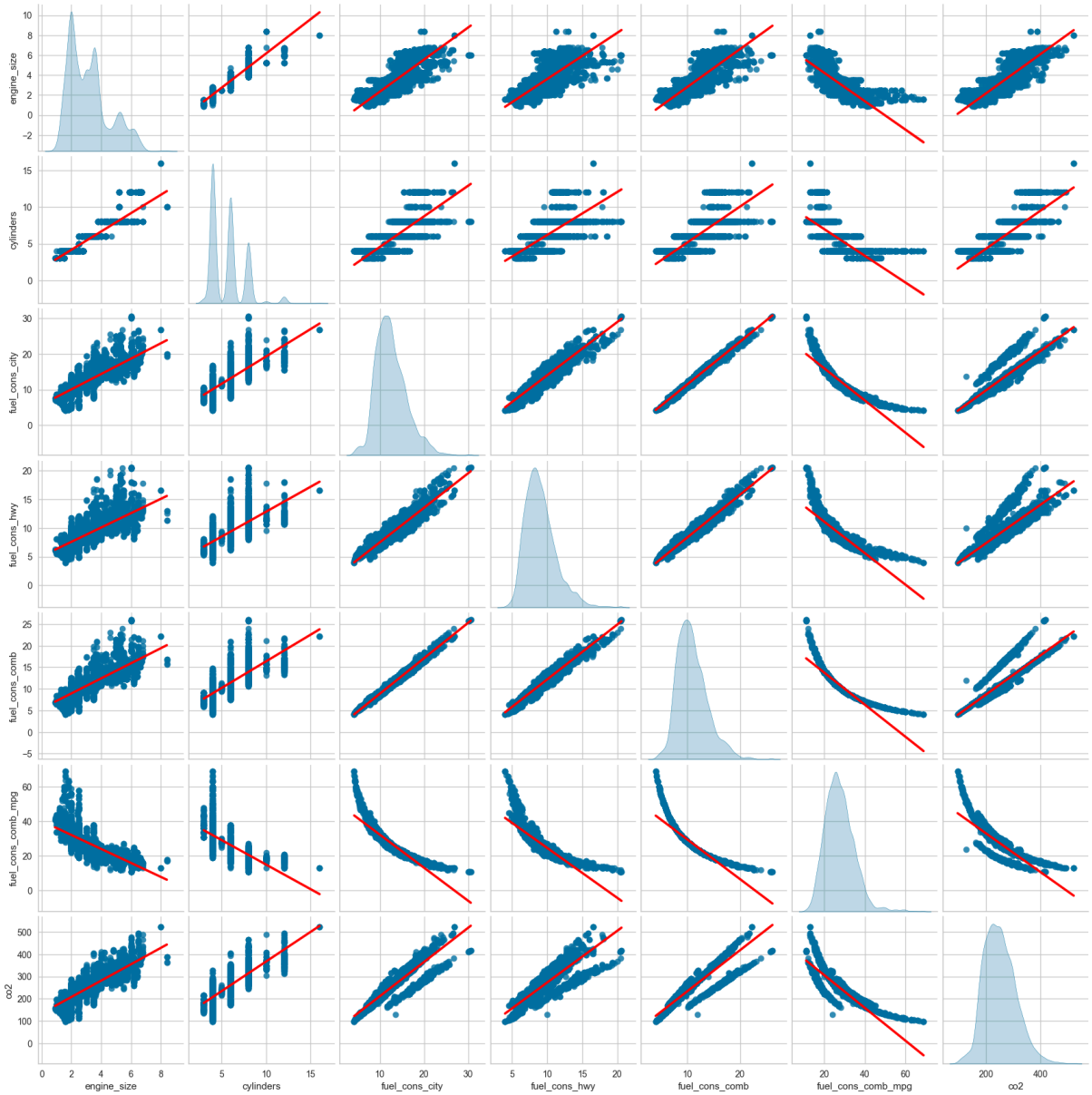
The correlation matrix shows strong multicollinearity among several features. Specifically:

- `fuel_cons_city`, `fuel_cons_hwy`, and `fuel_cons_comb` have very high correlations (0.95 and above), indicating they carry almost identical information.
- `engine_size` and `cylinders` are also highly correlated (0.93).
- `co2_emissions` has strong positive correlations with `fuel_cons_comb`, `fuel_cons_city`, and `engine_size` (0.85 and above), indicating these features are important for predicting CO2 emissions.
- `fuel_cons_mpg` shows high negative correlations with other fuel consumption measures (-0.93 and above), as higher mpg indicates lower fuel consumption.

These findings suggest that removing some highly correlated features can help reduce multicollinearity and improve model performance.

In [35]:
```
# Pairplot for the dataframe

sns.pairplot(df,
             kind="reg",
             diag_kind="kde",
             plot_kws={"line_kws": {"color": "red"}}
            )
```

Out[35]:
```
<seaborn.axisgrid.PairGrid at 0x20bde6a0d90>
```



The pairplot shows the relationships and distributions between various numerical features in the dataset. Here's a brief analysis:

1. **Engine Size (engine_size):**
   - Positively correlated with `cylinders`, `fuel_cons_city`, `fuel_cons_hwy`, `fuel_cons_comb`, and `co2`.
   - Larger engines tend to have more cylinders and higher fuel consumption.
2. **Cylinders:**
   - Strong positive correlation with `engine_size` and fuel consumption metrics.
   - As the number of cylinders increases, fuel consumption and CO2 emissions also increase.
3. **Fuel Consumption City (fuel_cons_City):**
   - High positive correlation with `fuel_cons_hwy`, `fuel_cons_comb`, and `co2`.
   - Vehicles that consume more fuel in the city tend to consume more on highways and produce higher CO2 emissions.
4. **Fuel Consumption Hwy (fuel_cons_Hwy):**
   - Similar correlations as city fuel consumption, showing strong positive relationships with `fuel_cons_city`, `fuel_cons_comb`, and `co2`.
5. **Fuel Consumption Combined (fuel_cons_Comb):**

- Very high correlation with both city and highway fuel consumption.
- Indicative of overall vehicle efficiency.

6. **Fuel Consumption MPG (fuel_cons_mpg):**

   - Shows a strong negative correlation with other fuel consumption metrics and `co2`.
   - *__Higher MPG values indicate better fuel efficiency and lower CO2 emissions.__*
   - fuel efficiency increases (mpg value increases), CO2 emissions decrease.

7. **CO2 Emissions (co2):**

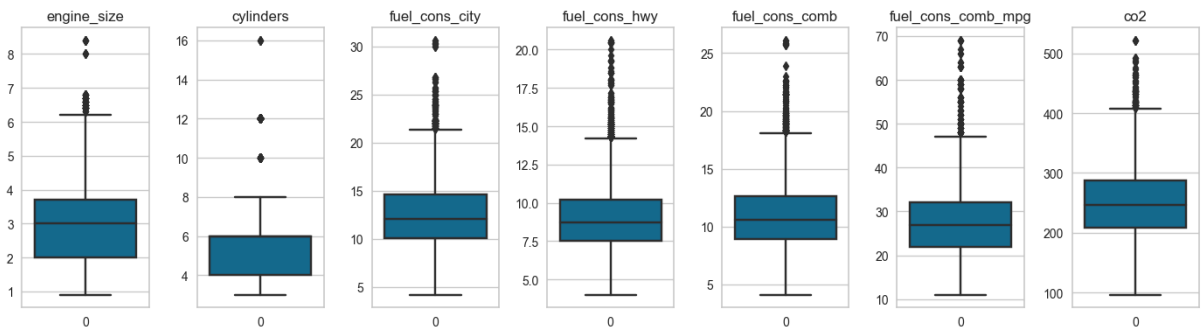
   - Strongly correlated with `engine_size`, `cylinders`, `fuel_cons_city`, `fuel_cons_hwy`, and `fuel_cons_comb`.
   - *__Vehicles with larger engines, more cylinders, and higher fuel consumption emit more CO2__*.

Overall Evaluation:

- The pairplot reveals strong relationships among features, particularly between engine size, fuel consumption, and CO2 emissions.
- The negative correlation between MPG and other features highlights its importance in representing fuel efficiency.
- Suggest focusing on features like `fuel_cons_comb`, `engine_size`, and `fuel_cons_mpg` for predictive modeling, while considering multicollinearity.

Outlier Analysis

In [36]:
```
# Checking Outliers

# Initialize the subplot counter
x = 0

# Create a figure with specified size
plt.figure(figsize=(16, 4))

# Loop through each numerical column and create a boxplot
for col in df.select_dtypes(include=['number']).columns:
    x += 1
    plt.subplot(1, 8, x)
    sns.boxplot(data=df[col])
    plt.title(col)

# Show the plots
plt.tight_layout()  # Adjust subplots to fit in the figure area.
plt.show()
```



- We do not intervene with outliers at the moment, but we can take action later according to the model's forecasting performance.

In [37]:
```
df.describe().T
```

Out[37]:

	count	mean	std	min	25%	50%	75%	max
engine_size	7385.0	3.160068	1.354170	0.9	2.0	3.0	3.7	8.4
cylinders	7385.0	5.615030	1.828307	3.0	4.0	6.0	6.0	16.0
fuel_cons_city	7385.0	12.556534	3.500274	4.2	10.1	12.1	14.6	30.6
fuel_cons_hwy	7385.0	9.041706	2.224456	4.0	7.5	8.7	10.2	20.6
fuel_cons_comb	7385.0	10.975071	2.892506	4.1	8.9	10.6	12.6	26.1
fuel_cons_comb_mpg	7385.0	27.481652	7.231879	11.0	22.0	27.0	32.0	69.0
co2	7385.0	250.584699	58.512679	96.0	208.0	246.0	288.0	522.0

Skewness

In [38]:

```
# Calculate skewness for numeric features

# A skewness value greater than 1 indicates positive skewness,
# a skewness value less than -1 indicates negative skewness,
# and a skewness value close to zero indicates a relatively symmetric distribution.

num_cols= df.select_dtypes('number').columns

skew_limit = 0.75               # define a limit above which we will log transform
skew_vals = df[num_cols].skew()


# Showing the skewed columns
skew_cols = (skew_vals
             .sort_values(ascending=False)
             .to_frame()
             .rename(columns={0:'Skew'})
             .query('abs(Skew) > {}'.format(skew_limit)))
skew_cols
```

Out[38]:

	Skew
cylinders	1.110415
fuel_cons_hwy	1.079217
fuel_cons_comb_mpg	0.977034
fuel_cons_comb	0.893316
engine_size	0.809181
fuel_cons_city	0.809005

MACHINE LEARNING

- The objective of creating and using a model with the CO2 emission dataset is to build machine learning algorithms capable of accurately predicting vehicle CO2 emissions based on their characteristics.
- By examining variables such as engine size, number of cylinders, and fuel consumption, the aim is to develop models that can evaluate the environmental impact of different vehicles and guide policy decisions aimed at reducing carbon emissions.
- Additionally, these models can support automotive manufacturers in designing more fuel-efficient vehicles and help consumers make informed choices when selecting vehicles with lower carbon

footprints.

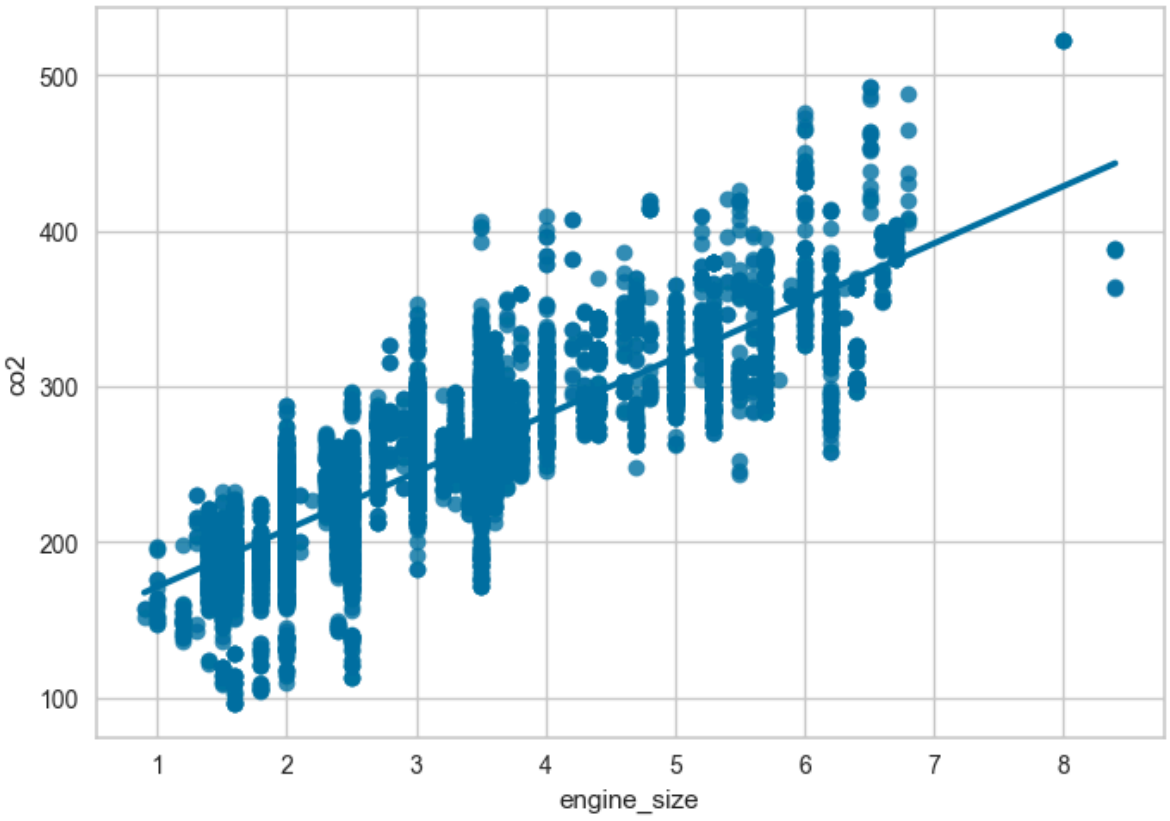
- Ultimately, the goal is to harness data-driven insights to mitigate the environmental impact of transportation and promote sustainable development.
- *Evaluating model accuracy on both training and test sets is essential to determine whether the model is overfitting or underfitting the data, addressing the bias-variance tradeoff effectively.*

Simple Linear Regression Model

- This simple linear regression model was built using only `engine_size` as the predictor and the target variable `co2`, without any data manipulation. This is often referred to as a "vanilla model."
- *As this dataset was compiled for analytical and educational purposes, steps suitable for beginners will be followed to ensure a thorough understanding.*

In [39]:
```
# Check the correlation between independent feature (engine_size) and target variable (co2_emiss

sns.regplot(x = 'engine_size', y = 'co2', data=df, ci=None)
```

Out[39]: <Axes: xlabel='engine_size', ylabel='co2'>



Splitting the Data

In [40]:
```
# Split the selected independent feature (engine_size) and target variable (co2_emissions) for S

X = df[['engine_size']]
y = df['co2']
```

Train | Test Split

In [41]:
```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [42]:
```
# Display the shapes of the resulting datasets

print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

```
X_train shape: (5908, 1)
X_test shape: (1477, 1)
y_train shape: (5908,)
y_test shape: (1477,)
```

Model

In [43]:
```
from sklearn.linear_model import LinearRegression

model_simple_lin_reg = LinearRegression()
```

Training the Model

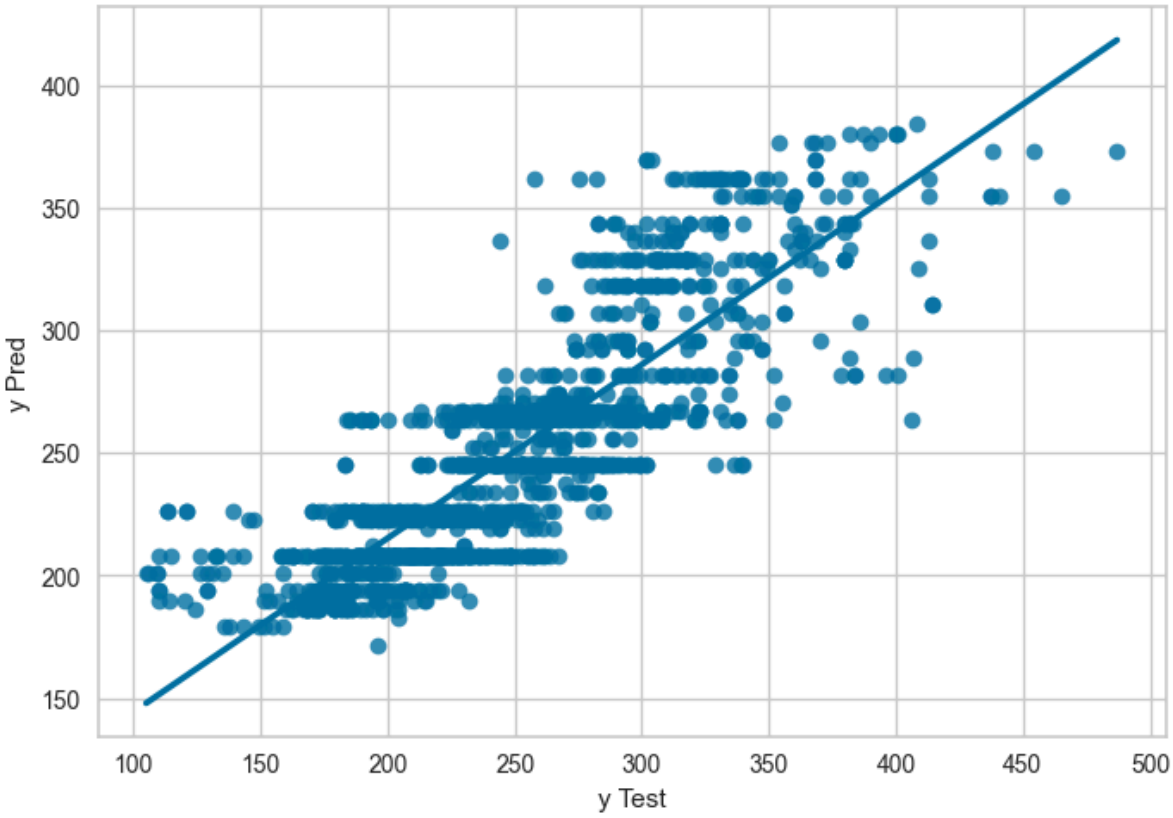
In [44]:
```
model_simple_lin_reg.fit(X_train, y_train)
```

Out[44]:
```
▾ LinearRegression
LinearRegression()
```

Predicting Test Data

In [45]:
```
# Predict using the model on the test data

y_train_pred = model_simple_lin_reg.predict(X_train)
y_pred = model_simple_lin_reg.predict(X_test)
```

In [46]:
```
sns.regplot(x=y_test, y=y_pred, ci=None)

plt.xlabel('y Test')
plt.ylabel('y Pred')

plt.show()
```



Evaluating the Model

Performance Metrics

In [47]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score


def train_val(y_train, y_train_pred, y_test, y_pred, i):

    scores = {
    i+"_train": {"R2" : r2_score(y_train, y_train_pred),
    "mae" : mean_absolute_error(y_train, y_train_pred),
    "mse" : mean_squared_error(y_train, y_train_pred),
    "rmse" : np.sqrt(mean_squared_error(y_train, y_train_pred))},

    i+"_test": {"R2" : r2_score(y_test, y_pred),
    "mae" : mean_absolute_error(y_test, y_pred),
    "mse" : mean_squared_error(y_test, y_pred),
    "rmse" : np.sqrt(mean_squared_error(y_test, y_pred))}
    }

    return pd.DataFrame(scores)
```

In [48]:
```
slr_score = train_val(y_train, y_train_pred, y_test, y_pred, 'linear')
slr_score
```

Out[48]:

	linear_train	linear_test
R2	0.724528	0.723812
mae	23.374987	22.927177
mse	941.716358	949.985253
rmse	30.687397	30.821831

- The difference between MAE and RMSE indicates outlier effect

In [49]:
```
avg_co2= df['co2'].mean()
avg_co2
```

Out[49]: 250.58469871360867

In [50]:
```
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
rmse
```

Out[50]: 30.821830791462336

In [51]:
```
rmse/avg_co2
```

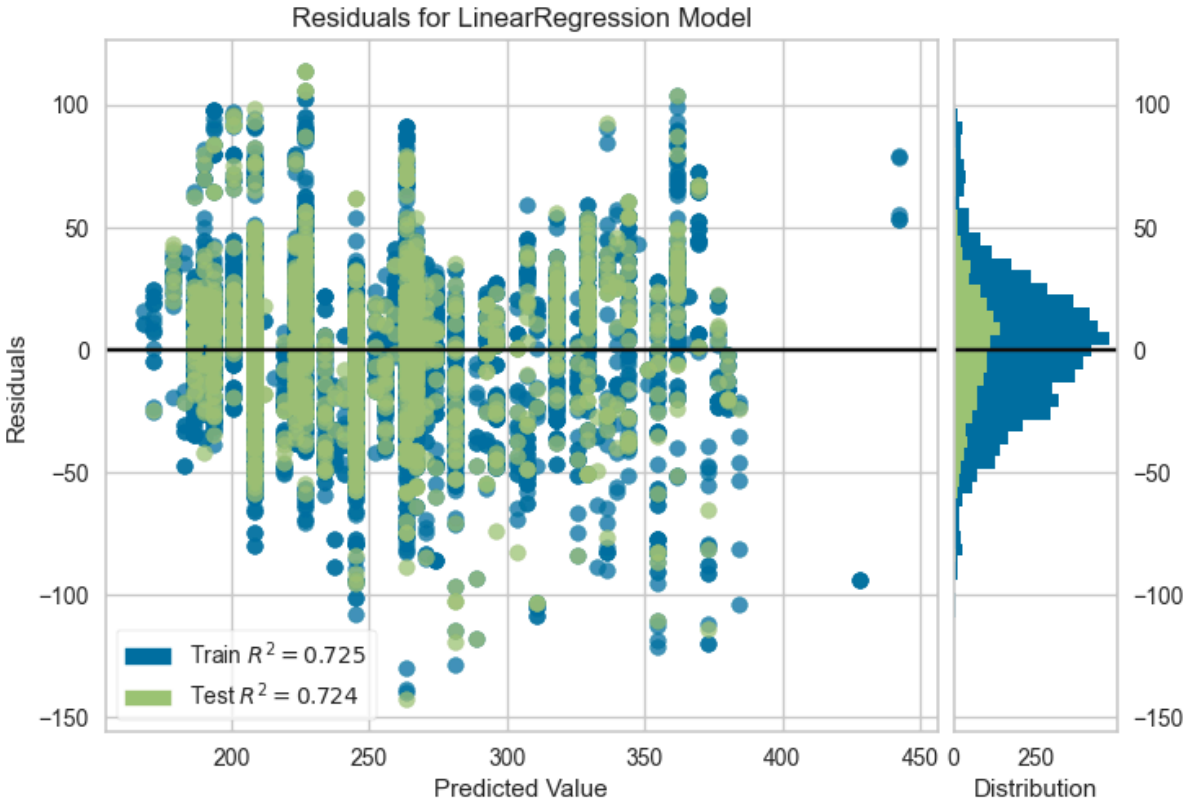
Out[51]: 0.12299965221215829

- To determine how much the error deviates from the mean of the target label.
- According to the RMSE metric, our model has an average error rate of 12%.
- Prefer the RMSE metric because it penalizes poor predictions.

Residuals Distributions

In [52]:
```
# !pip install yellowbrick
```

In [53]:
```
from yellowbrick.regressor import ResidualsPlot

# linear model and visualizer

model = LinearRegression()
visualizer = ResidualsPlot(model)
visualizer.fit(X_train, y_train)  # Fit the training data to the visualizer
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
visualizer.show()                 # Finalize and render the figure
```



Out[53]: <Axes: title={'center': 'Residuals for LinearRegression Model'}, xlabel='Predicted Value', ylabel='Residuals'>

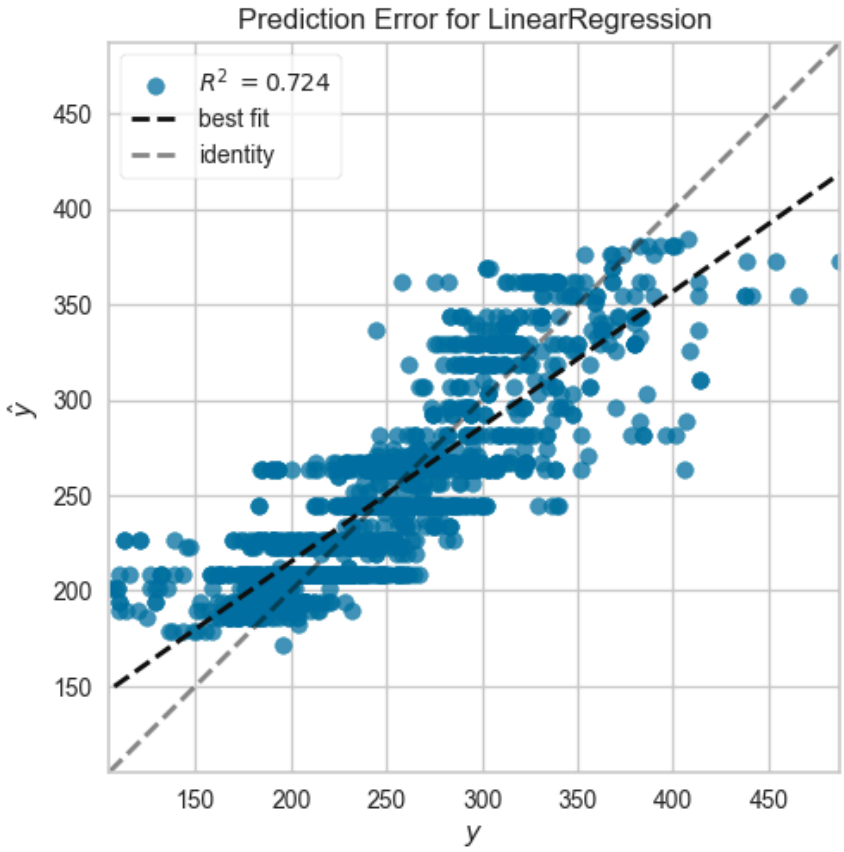
In [54]:
```
# Comparing Actual y_test, Predicted_y and Residuals

my_dict = {"Actual": y_test, "pred": y_pred, "residual": y_test - y_pred}
compare = pd.DataFrame(my_dict)
compare.head(20)
```

Out[54]:

	Actual	pred	residual
7261	253	244.893111	8.106889
4489	344	296.150160	47.849840
1539	322	362.052081	-40.052081
3532	297	329.101120	-32.101120
6418	308	281.505289	26.494711
3703	406	263.199200	142.800800
5976	242	244.893111	-2.893111
4332	216	208.280933	7.719067
5015	246	222.925804	23.074196
2087	223	226.587022	-3.587022
2126	283	233.909457	49.090543
4161	326	281.505289	44.494711
4814	274	266.860418	7.139582
486	251	263.199200	-12.199200
6607	322	296.150160	25.849840
1128	382	288.827724	93.172276
5159	248	244.893111	3.106889
5391	193	263.199200	-70.199200
6643	204	186.313626	17.686374
6003	211	222.925804	-11.925804

Prediction Error For LinearRegression

In [55]:
```
from yellowbrick.regressor import PredictionError

model = LinearRegression()

visualizer = PredictionError(model)
visualizer.fit(X_train, y_train)  # Fit the training data to the visualizer
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
visualizer.show()                 # Finalize and render the figure
```



Out[55]: <Axes: title={'center': 'Prediction Error for LinearRegression'}, xlabel='y', ylabel='$\\hat{y}$'>

- The graph shows the prediction error for a linear regression model with an (R^2) value of 0.724, indicating that 72.4% of the variance in the target variable is explained by the model.
- The points are mostly close to the best fit line, suggesting reasonable accuracy.
- However, the model tends to underpredict higher values.

Conclusion

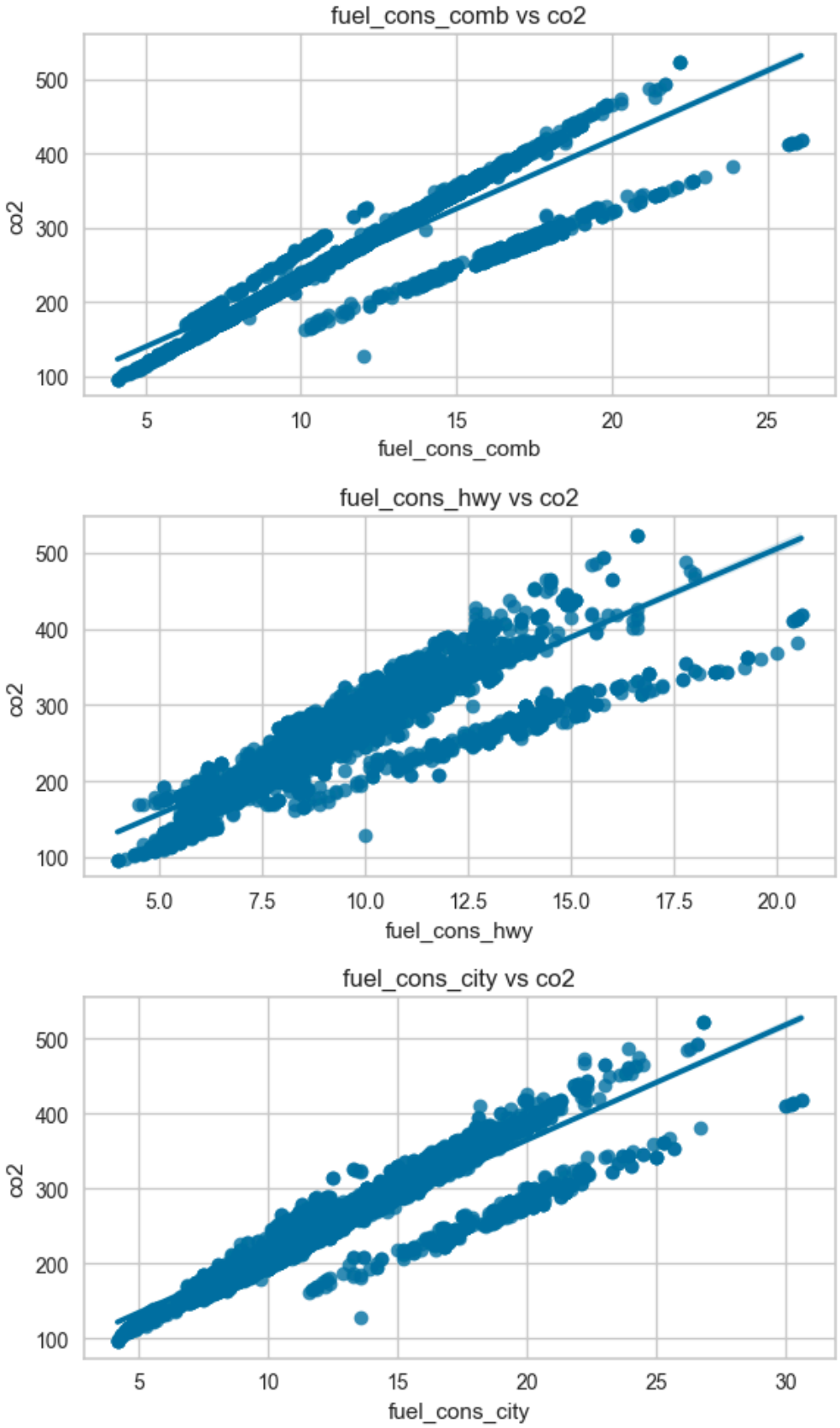
- This simple linear regression model was built using only engine size as the predictor and the target variable, without any data manipulation.
- This is often referred to as a "vanilla model."
- The model explains 72.4% of the variance in the target variable, indicating a good fit but with room for improvement.

Multiple Linear Regression Model

- We will now create a multiple linear regression model using 'engine_size', 'fuel_cons_comb', 'fuel_cons_hwy', and 'fuel_cons_city' as the independent variables and the target variable.
- This model aims to capture the relationship between multiple predictors and the target variable for better prediction accuracy and insights.

In [56]:
```
# See the relationship between independent features and target variable

independent_variables = ['fuel_cons_comb', 'fuel_cons_hwy','fuel_cons_city' ]
target_variable = 'co2'

plt.figure(figsize=(6,10))

for i, var in enumerate(independent_variables):
    plt.subplot(3,1, i + 1)
```

```
    sns.regplot(x=df[var], y=df[target_variable])
    plt.title(f'{var} vs {target_variable}')

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



Splitting the Data

In [57]:
```
X = df[["engine_size", "cylinders", "fuel_cons_comb", "fuel_cons_comb_mpg"]]
y = df["co2"]
```

In [58]:
```
X.head()
```

Out[58]:

	engine_size	cylinders	fuel_cons_comb	fuel_cons_comb_mpg
0	2.0	4	8.5	33
1	2.4	4	9.6	29
2	1.5	4	5.9	48
3	3.5	6	11.1	25
4	3.5	6	10.6	27

In [59]:
```
# Check Multicolinarty between features

def color_custom(val):
    if val > 0.90 and val < 0.99:
        color = 'red'
    elif val >= 1:
        color = 'blue'
    else:
        color = 'black'
    return f'color: {color}'
```

In [60]:
```
pd.DataFrame(X).corr().style.map(color_custom)
```

Out[60]:

	engine_size	cylinders	fuel_cons_comb	fuel_cons_comb_mpg
engine_size	1.000000	0.927653	0.817060	-0.757854
cylinders	0.927653	1.000000	0.780534	-0.719321
fuel_cons_comb	0.817060	0.780534	1.000000	-0.925576
fuel_cons_comb_mpg	-0.757854	-0.719321	-0.925576	1.000000

Train | Test Split

In [61]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

In [62]:
```
print("X Train features shape: {}\ny Train features shape: {}\nX Test features shape: {}\ny Test
      .format(X_train.shape, y_train.shape, X_test.shape, y_test.shape))
```

```
X Train features shape: (5169, 4)
y Train features shape: (5169,)
X Test features shape: (2216, 4)
y Test features shape: (2216,)
```

Model

In [63]:
```
from sklearn.linear_model import LinearRegression

model_multi_lin_reg = LinearRegression()
```

Training the Model

In [64]:
```
model_multi_lin_reg.fit(X_train, y_train)
```

Out[64]:
```
▾ LinearRegression
LinearRegression()
```

Predicting Test Data

In [65]:
```
y_pred = model_multi_lin_reg.predict(X_test)
y_train_pred = model_multi_lin_reg.predict(X_train)
```

Evaluating the Model

Evaluating: Performance- Error Metrics

In [66]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
# rmse fonk yoktur, mse karekoku alinarak manuel hesapla

def train_val(y_train, y_train_pred, y_test, y_pred, i):

    scores = {
    i+"_train": {"R2" : r2_score(y_train, y_train_pred),
    "mae" : mean_absolute_error(y_train, y_train_pred),
    "mse" : mean_squared_error(y_train, y_train_pred),
    "rmse" : np.sqrt(mean_squared_error(y_train, y_train_pred))},

    i+"_test": {"R2" : r2_score(y_test, y_pred),
    "mae" : mean_absolute_error(y_test, y_pred),
    "mse" : mean_squared_error(y_test, y_pred),
    "rmse" : np.sqrt(mean_squared_error(y_test, y_pred))}
    }

    return pd.DataFrame(scores)
```

In [67]:
```
mlr_score = train_val(y_train, y_train_pred, y_test, y_pred, "multi")
mlr_score

# As can be seen, high R2 scores were obtained with only 4 features. (excluding categorical feat
```

Out[67]:

	multi_train	multi_test
R2	0.903877	0.900108
mae	11.436689	11.514614
mse	330.829164	337.406607
rmse	18.188710	18.368631

Cross Validation for Multiple Linear

In [68]:
```
from sklearn.metrics import get_scorer_names

scorers = get_scorer_names()
print(scorers)
```

```
['accuracy', 'adjusted_mutual_info_score', 'adjusted_rand_score', 'average_precision', 'balanced_
accuracy', 'completeness_score', 'explained_variance', 'f1', 'f1_macro', 'f1_micro', 'f1_sample
s', 'f1_weighted', 'fowlkes_mallows_score', 'homogeneity_score', 'jaccard', 'jaccard_macro', 'jac
card_micro', 'jaccard_samples', 'jaccard_weighted', 'matthews_corrcoef', 'max_error', 'mutual_inf
o_score', 'neg_brier_score', 'neg_log_loss', 'neg_mean_absolute_error', 'neg_mean_absolute_percen
tage_error', 'neg_mean_gamma_deviance', 'neg_mean_poisson_deviance', 'neg_mean_squared_error', 'n
eg_mean_squared_log_error', 'neg_median_absolute_error', 'neg_negative_likelihood_ratio', 'neg_ro
ot_mean_squared_error', 'normalized_mutual_info_score', 'positive_likelihood_ratio', 'precision',
'precision_macro', 'precision_micro', 'precision_samples', 'precision_weighted', 'r2', 'rand_scor
e', 'recall', 'recall_macro', 'recall_micro', 'recall_samples', 'recall_weighted', 'roc_auc', 'ro
c_auc_ovo', 'roc_auc_ovo_weighted', 'roc_auc_ovr', 'roc_auc_ovr_weighted', 'top_k_accuracy', 'v_m
easure_score']
```

In [69]:
```
from sklearn.model_selection import cross_validate, cross_val_score

model = LinearRegression()
scores = cross_validate(model, X_train, y_train,
                        scoring = ['r2', 'neg_mean_absolute_error','neg_mean_squared_error', \
                                   'neg_root_mean_squared_error'], cv = 10, return_train_score=
```

In [70]:
```
pd.DataFrame(scores, index = range(1,11))
```

Out[70]:

	fit_time	score_time	test_r2	train_r2	test_neg_mean_absolute_error	train_neg_mean_absolute_erro
1	0.004081	0.002005	0.917568	0.902321	-10.966634	-11.49584
2	0.003000	0.002000	0.898006	0.904529	-11.705880	-11.44524
3	0.003002	0.002001	0.892331	0.904956	-11.329495	-11.40312
4	0.002997	0.001998	0.920210	0.902001	-10.763197	-11.49100
5	0.002001	0.002000	0.875318	0.906981	-12.949277	-11.24134
6	0.003003	0.001999	0.894262	0.904821	-11.368600	-11.38024
7	0.003001	0.001997	0.915373	0.902485	-10.922760	-11.50830
8	0.002000	0.003001	0.917338	0.902337	-11.034622	-11.54371
9	0.002005	0.002002	0.898096	0.904445	-11.276306	-11.46786
10	0.004577	0.002007	0.902799	0.903967	-12.168696	-11.38218

In [71]:
```
scores = pd.DataFrame(scores, index=range(1,11))
scores.iloc[:, 2:].mean()
```

Out[71]:
```
test_r2                            0.903130
train_r2                           0.903884
test_neg_mean_absolute_error     -11.448547
train_neg_mean_absolute_error    -11.435887
test_neg_mean_squared_error     -331.701024
train_neg_mean_squared_error    -330.783996
test_neg_root_mean_squared_error  -18.171359
train_neg_root_mean_squared_error -18.186934
dtype: float64
```

- The fact that this score obtained after Cross Validation and Train-test score are compatible indicates that the model has generalization ability.

Comparing the Scores Multi & Simple Linear Regression

In [72]:
```
pd.concat([slr_score, mlr_score], axis=1)

# Concatinated simple linear ve multiple linear models scores
```

Out[72]:

	linear_train	linear_test	multi_train	multi_test
R2	0.724528	0.723812	0.903877	0.900108
mae	23.374987	22.927177	11.436689	11.514614
mse	941.716358	949.985253	330.829164	337.406607
rmse	30.687397	30.821831	18.188710	18.368631

In [73]:
```
print("train RMSE:", 20.762905/df["co2"].mean())
print("CV RMSE:", 20.929894/df["co2"].mean())
```

```
train RMSE: 0.08285783252763476
CV RMSE: 0.08352422996074718
```

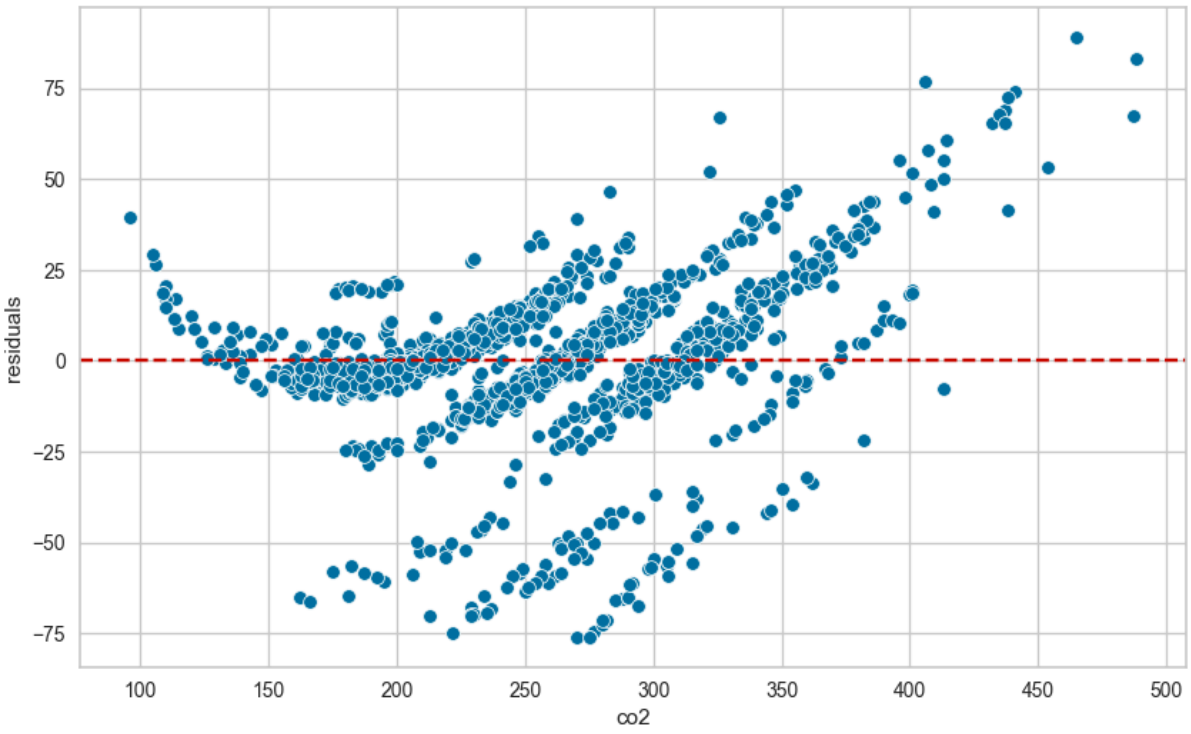
In [74]:
```
mlr_df = pd.DataFrame(model_multi_lin_reg.coef_, index=X_train.columns, columns = ["mlr_coef"])
mlr_df
```

Out[74]:

	mlr_coef
engine_size	4.513895
cylinders	7.044066
fuel_cons_comb	5.855570
fuel_cons_comb_mpg	-3.258171

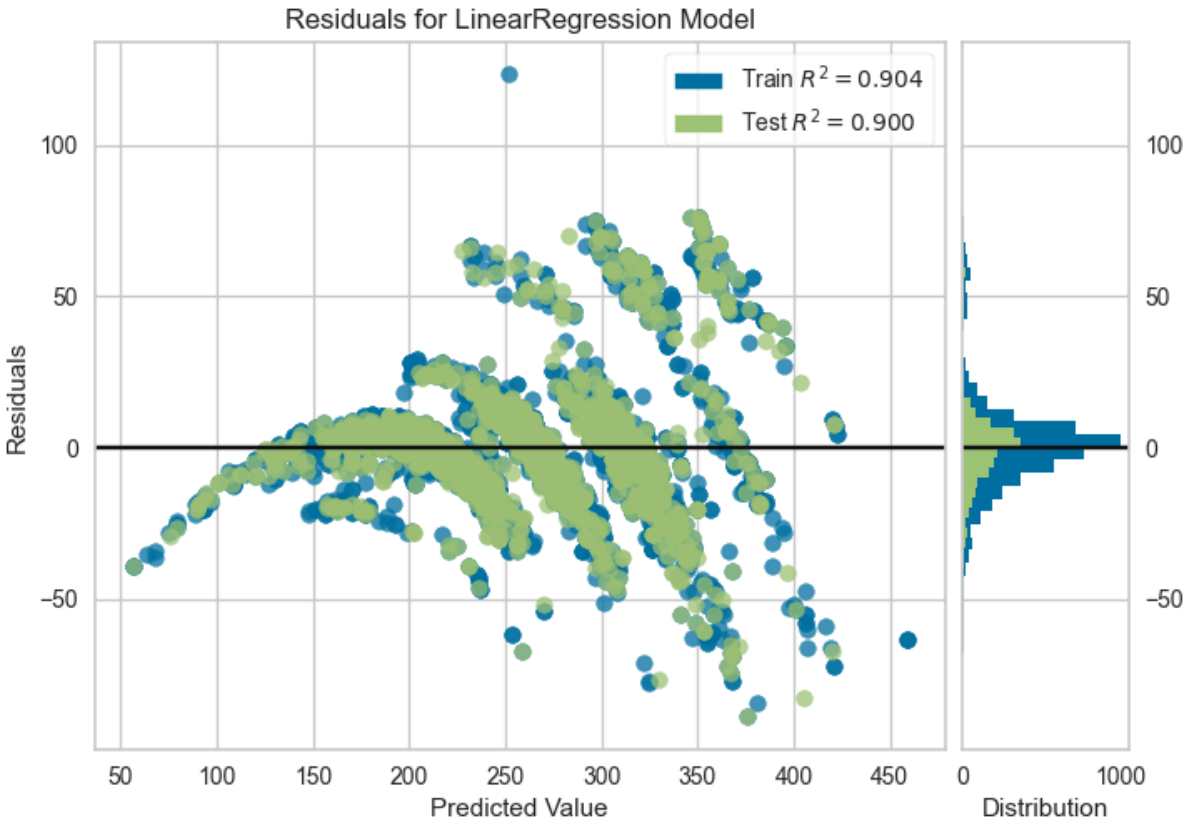
Evaluating: Residuals Distributions

In [75]:
```
residuals = y_test - y_pred

plt.figure(figsize = (10,6))
sns.scatterplot(x = y_test, y = residuals) #-residuals
plt.axhline(y = 0, color ="r", linestyle = "--")
plt.ylabel("residuals");
```



In [76]:
```
from yellowbrick.regressor import ResidualsPlot


model = LinearRegression()
```

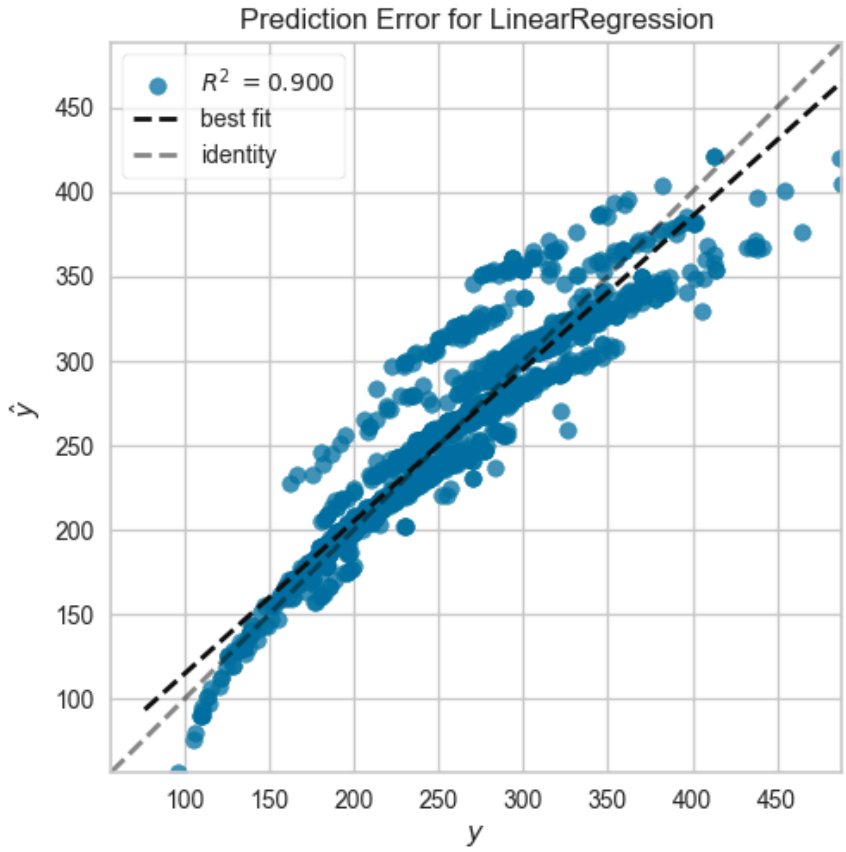
```
visualizer = ResidualsPlot(model)

visualizer.fit(X_train, y_train)
visualizer.score(X_test, y_test)
visualizer.show();                 # Finalize and render the figure
```



- The residual plot indicates that the linear regression model has good performance, with (R^2) values of 0.904 for the training set and 0.900 for the test set.
- Most residuals are close to zero, indicating accurate predictions.
- However, there are systematic errors as residuals increase with predicted values, suggesting the model may be biased for higher values.
- Additionally, the spread of residuals indicates the variance of errors is not constant. This suggests potential areas for model improvement.

Evaluating: Prediction Error for LinearRegression

In [77]:
```
from yellowbrick.regressor import PredictionError

model = LinearRegression()

visualizer = PredictionError(model)
visualizer.fit(X_train, y_train)  # Fit the training data to the visualizer
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
visualizer.show()                 # Finalize and render the figure
```



Out[77]: <Axes: title={'center': 'Prediction Error for LinearRegression'}, xlabel='y', ylabel='$\\hat{y}$'>

- This graph shows the prediction error for a linear regression model with an (R^2) value of 0.900, indicating that 90% of the variance in the target variable is explained by the model.
- The points are closely aligned with the best fit line, suggesting accurate predictions.
- However, there is a slight deviation from the identity line, especially at higher values, indicating that the model may slightly underpredict or overpredict in certain ranges.
- Overall, the model performs well with a high degree of accuracy.

Polynomial Features

In [78]:
```
from sklearn.preprocessing import PolynomialFeatures

def poly(d):

    train_rmse_errors = []
    test_rmse_errors = []
    number_of_features = []

    for i in range(1, d):
        polynomial_converter = PolynomialFeatures(degree = i, include_bias =False)
        poly_features = polynomial_converter.fit_transform(X)

        X_train, X_test, y_train, y_test = train_test_split(poly_features, y, test_size=0.3, rar

        model = LinearRegression(fit_intercept=True)
        model.fit(X_train, y_train)

        train_pred = model.predict(X_train)
        test_pred = model.predict(X_test)

        train_RMSE = np.sqrt(mean_squared_error(y_train,train_pred))
        test_RMSE = np.sqrt(mean_squared_error(y_test,test_pred))
```

```
        train_rmse_errors.append(train_RMSE)
        test_rmse_errors.append(test_RMSE)

        number_of_features.append(poly_features.shape[1])

    return pd.DataFrame({"train_rmse_errors": train_rmse_errors, "test_rmse_errors":test_rmse_er
                         "number of features":number_of_features}, index=range(1,d))
```

In [79]:
```
poly(8)

# The poly(8) function creates polynomial regression models of degrees 1 to 8,
# evaluates their training and test RMSE, and returns a DataFrame summarizing these errors and
# the number of features used at each degree.
```

Out[79]:

	train_rmse_errors	test_rmse_errors	number of features
1	18.314047	18.087420	4
2	15.788285	15.680230	14
3	14.314294	13.914205	34
4	12.916365	12.699119	69
5	12.093931	12.490365	125
6	11.139029	13.456785	209
7	19.939588	116.936156	329

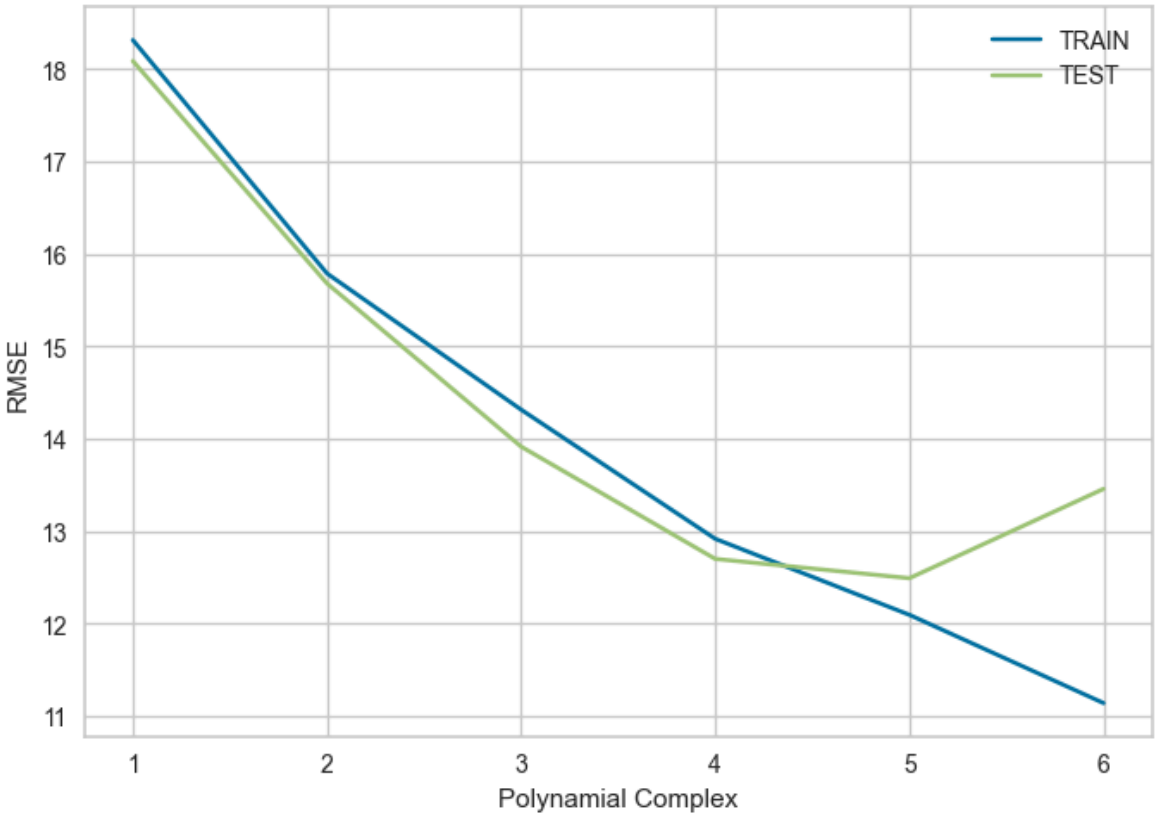
This table shows RMSE and feature counts for polynomial regression models of different degrees.

Key insights:

- **Performance Improvement**: RMSE decreases up to the 6th degree, indicating better performance.
- **Optimal Degree**: The 6th degree has the lowest test RMSE (13.456785).
- **Overfitting**: The 7th degree shows overfitting with a high test RMSE (116.936156).
- **Feature Count**: The 4th degree model, with 69 features, achieves similar performance to the 5th degree model (125 features), making it more efficient and cost-effective.

The 4th degree is preferable for reducing computational cost while maintaining performance.

In [80]:
```
plt.plot(range(1,7), poly(7)["train_rmse_errors"], label = "TRAIN")
plt.plot(range(1,7), poly(7)["test_rmse_errors"], label = "TEST")
plt.xlabel("Polynamial Complex")
plt.ylabel("RMSE")
plt.legend()
plt.show()
```



Poly(degree=4)

In [81]:
```
# Selected degree=4

poly_converter = PolynomialFeatures(degree = 4, include_bias=False)
```

Model

In [82]:
```
# Poly linear model

poly_lin_reg = LinearRegression()
```

Train | Test Split

In [83]:
```
X_train, X_test, y_train, y_test = train_test_split(poly_converter.fit_transform(X), y,
                                                    test_size = 0.2, random_state = 42)
```

Training the Model

In [84]:
```
poly_lin_reg.fit(X_train, y_train)
```

Out[84]:
```
LinearRegression()
```

Predicting Test Data

In [85]:
```
y_train_pred = poly_lin_reg.predict(X_train)
y_pred = poly_lin_reg.predict(X_test)
```

Evaluating the Model

In [86]:
```
poly_mlr_score = train_val(y_train, y_train_pred, y_test, y_pred, "poly(4)")
poly_mlr_score
```

Out[86]:

	poly(4)_train	poly(4)_test
R2	0.953385	0.947646
mae	6.042713	6.426362
mse	159.355150	180.078849
rmse	12.623595	13.419346

Comparison of Multiple Linear & Poly Multiple Linear Regression

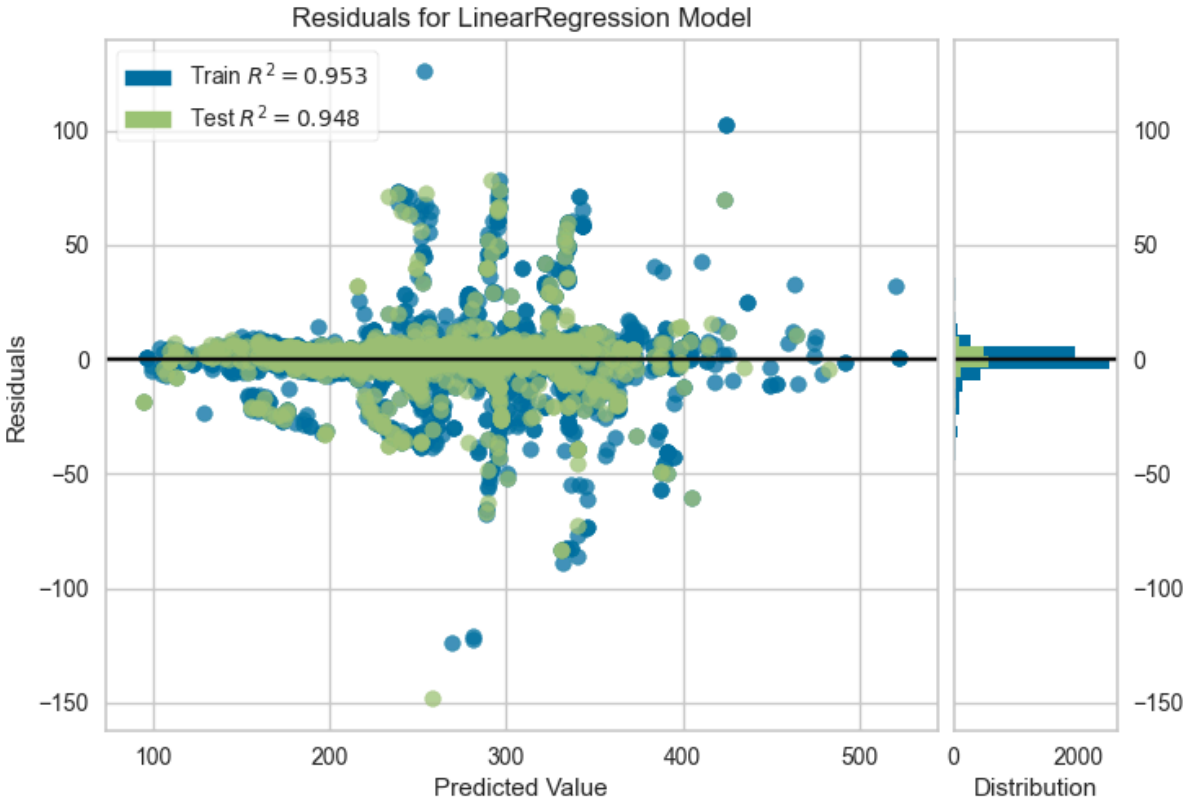
In [87]:
```
result = pd.concat([mlr_score, poly_mlr_score], axis=1)
result
```

Out[87]:

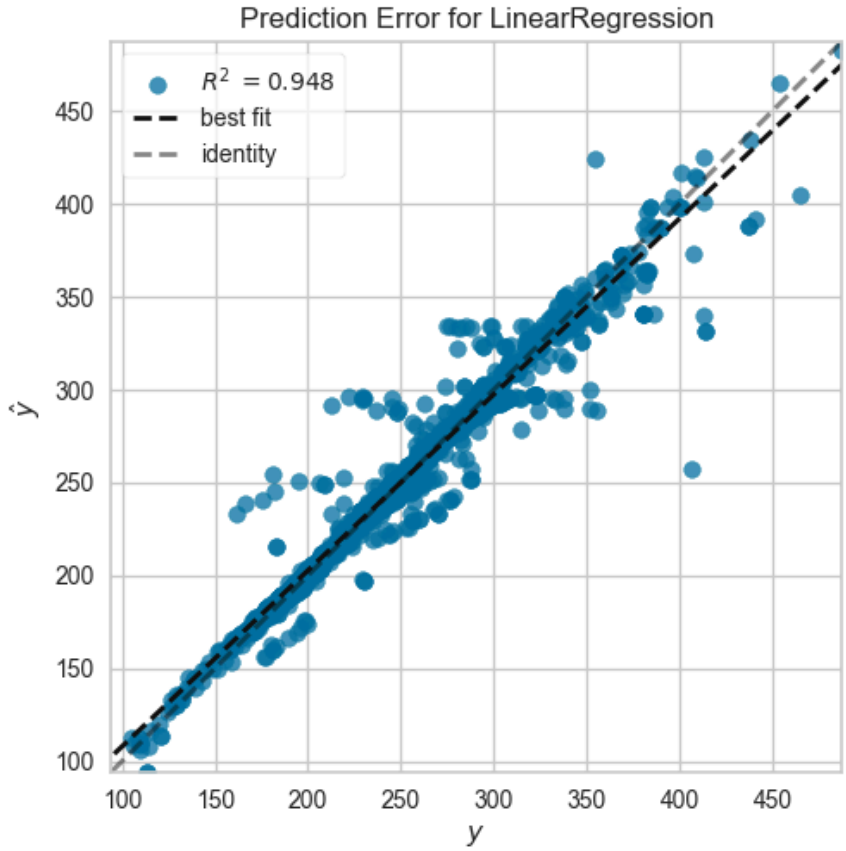
	multi_train	multi_test	poly(4)_train	poly(4)_test
R2	0.903877	0.900108	0.953385	0.947646
mae	11.436689	11.514614	6.042713	6.426362
mse	330.829164	337.406607	159.355150	180.078849
rmse	18.188710	18.368631	12.623595	13.419346

- The polynomial regression model (degree 4) performs better than the multiple linear regression model, achieving higher (R^2) values
- and lower MAE, MSE, and RMSE on both training and test sets, indicating better overall performance and accuracy.

In [88]:
```
# Distributions of Poly_Multi Regression Residuals

poly_lin_reg = LinearRegression()
visualizer = ResidualsPlot(poly_lin_reg)

visualizer.fit(X_train, y_train)
visualizer.score(X_test, y_test)
visualizer.show();
```



In [89]:
```
# Prediction Error for Poly_Multi LinearRegression

poly_lin_reg = LinearRegression()
visualizer = PredictionError(poly_lin_reg)

visualizer.fit(X_train, y_train)
visualizer.score(X_test, y_test)
visualizer.show();
```



Scaling the Data

In [90]:
```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler

# StandardScaler

scaler = StandardScaler()
```

In [91]:
```
X_train.shape
```

Out[91]:
```
(5908, 69)
```

In [92]:
```
scaler.fit(X_train)
```

Out[92]:
```
StandardScaler

StandardScaler()
```

In [93]:
```
X_train_scaled = scaler.transform(X_train)
X_train_scaled
```

Out[93]:
```
array([[-0.12149345,  0.20534188, -0.37681659, ...,  1.49112427,
         0.11090642, -0.1077625 ],
       [ 0.6141664 ,  0.20534188,  0.59257065, ..., -0.88383211,
        -0.7146818 , -0.4535469 ],
       [-0.12149345,  0.20534188,  0.24636092, ..., -0.45682398,
        -0.50862809, -0.38225914],
       ...,
       [ 0.24633648,  0.20534188, -0.20371173, ..., -0.45682398,
        -0.17471027, -0.2363027 ],
       [-0.48932338, -0.88513888, -0.82688924, ...,  0.46538028,
         0.6363709 ,  0.24212651],
       [ 0.6141664 ,  1.29582264,  0.31560287, ...,  1.02340697,
        -0.48713916, -0.38225914]])
```

In [94]:
```
X_test_scaled = scaler.transform(X_test)
X_test_scaled
```

Out[94]:
```
array([[-0.12149345,  0.20534188, -0.03060686, ...,  0.34185348,
        -0.27672051, -0.29073711],
       [ 0.90843035,  1.29582264,  1.2849901 , ..., -0.91425102,
        -0.98206772, -0.52952408],
       [ 2.23261809,  1.29582264,  1.04264329, ..., -0.70108997,
        -0.89522984, -0.50782853],
       ...,
       [-1.29854922, -0.88513888, -1.10385702, ..., -0.45682398,
         1.06274987,  0.60302082],
       [ 0.24633648,  0.20534188, -0.27295367, ...,  1.02340697,
        -0.02542564, -0.17547199],
       [-0.85715331, -0.88513888, -0.96537313, ..., -1.06618224,
         0.73923453,  0.35208087]])
```

In [95]:
```
pd.DataFrame(X_train_scaled).agg(['mean','std']).round(2)
```

Out[95]:

	0	1	2	3	4	5	6	7	8	9	...	59	60	61	62	63	64	65	66	67	(
mean	0.0	-0.0	0.0	-0.0	0.0	-0.0	0.0	0.0	-0.0	0.0	...	0.0	-0.0	0.0	0.0	0.0	-0.0	-0.0	-0.0	0.0	0
std	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1

2 rows × 69 columns

In [96]:
```
pd.DataFrame(X_test_scaled).agg(['mean','std']).round(2)
```

Out[96]:

	0	1	2	3	4	5	6	7	8	9	...	59	60	61	62	63
mean	-0.02	-0.02	-0.02	0.03	-0.02	-0.02	-0.02	-0.00	-0.02	-0.02	...	0.01	-0.02	-0.02	0.00	0.03
std	0.98	0.98	1.01	1.03	0.97	0.97	0.98	0.98	0.97	0.98	...	1.02	0.94	0.98	1.01	1.06

2 rows × 69 columns

Regularization

Ridge Regression

In [97]:
```
# Set the Ridge Regression Model

from sklearn.linear_model import Ridge

ridge_model = Ridge(alpha=1, random_state=42) # default alpha=1
```

In [98]:
```
# Fit the model

ridge_model.fit(X_train_scaled, y_train)
```

Out[98]:
```
Ridge(alpha=1, random_state=42)
```

In [99]:
```
# Prediction

y_pred = ridge_model.predict(X_test_scaled)
y_train_pred = ridge_model.predict(X_train_scaled)
```

In [100…
```
ridge_score = train_val(y_train, y_train_pred, y_test, y_pred, "ridge")
ridge_score
```

Out[100…

	ridge_train	ridge_test
R2	0.936997	0.927938
mae	7.652171	8.404607
mse	215.378640	247.865625
rmse	14.675784	15.743749

In [101…
```
ridge_model.coef_[:20]
```

Out[101…
```
array([ 23.53473428,  21.81177982,  36.25291809, -48.15273428,
         9.39307922, -20.15183773, -11.79747111,  -8.44492693,
        22.84914613,  21.2414396 , -45.89679452, -46.50500418,
         1.95549588,  24.56310834,  11.56138468,  -8.88385905,
        11.4720219 ,   2.99571656,  -2.67187074, -29.493438  ])
```

In [102…
```
result = pd.concat([result, ridge_score], axis=1)
result
```

Out[102…

	multi_train	multi_test	poly(4)_train	poly(4)_test	ridge_train	ridge_test
R2	0.903877	0.900108	0.953385	0.947646	0.936997	0.927938
mae	11.436689	11.514614	6.042713	6.426362	7.652171	8.404607
mse	330.829164	337.406607	159.355150	180.078849	215.378640	247.865625
rmse	18.188710	18.368631	12.623595	13.419346	14.675784	15.743749

Cross Validation for Ridge with Alpha=1

In [103…

```
model = Ridge(alpha=1)
scores = cross_validate(model, X_train_scaled, y_train,
                    scoring=['r2', 'neg_mean_absolute_error','neg_mean_squared_error','neg_root_
                        cv=10, return_train_score=True)
```

In [104…

```
scores = pd.DataFrame(scores, index = range(1, 11))
scores.iloc[:,2:].mean()
```

Out[104…

```
test_r2                            0.935583
train_r2                           0.936798
test_neg_mean_absolute_error      -7.741835
train_neg_mean_absolute_error     -7.687446
test_neg_mean_squared_error      -219.792111
train_neg_mean_squared_error     -216.053887
test_neg_root_mean_squared_error  -14.782462
train_neg_root_mean_squared_error -14.698160
dtype: float64
```

In [105…

```
ridgeCV_score = train_val(y_train, y_train_pred, y_test, y_pred, "ridgeCV")
ridgeCV_score
```

Out[105…

	ridgeCV_train	ridgeCV_test
R2	0.936997	0.927938
mae	7.652171	8.404607
mse	215.378640	247.865625
rmse	14.675784	15.743749

In [106…

```
rm = Ridge(alpha=1).fit(X_train_scaled, y_train)
rm.coef_
```

Out[106…

```
array([ 23.53473428,  21.81177982,  36.25291809, -48.15273428,
          9.39307922, -20.15183773, -11.79747111,  -8.44492693,
         22.84914613,  21.2414396 , -45.89679452, -46.50500418,
          1.95549588,  24.56310834,  11.56138468,  -8.88385905,
         11.4720219 ,   2.99571656,  -2.67187074, -29.493438  ,
        -25.92919119, -55.00731808,  32.57263267, -18.88823925,
         18.03713269,  28.127431  , -10.13832476, -18.82012788,
         35.17502176,  13.39583867, -61.73947536,  28.4628938 ,
        -44.73051695,  41.32117708, -30.79311154,  -8.82004599,
         66.1051035 , -15.66256156,   2.90007861,   2.8575353 ,
        -13.17019181,  42.10532967,  14.22162041,   6.11491563,
         -1.01654173, -10.0296938 ,  -6.06762709, -32.36687191,
        -13.02702283,   1.04328498, -11.22597825, -12.41801011,
         12.9786839 ,  10.68898143, -26.69883855,  14.5411835 ,
         -7.14864989,  12.55756406,  31.16072789,   9.71332932,
         -2.45418754,  22.74877625, -16.18768069,  40.42328897,
         75.14917693, -57.57968339,   1.14865078,  25.90298166,
        -55.11753146])
```

In [107…

```
rm_df = pd.DataFrame(ridge_model.coef_,  columns = ["ridge_coef_1"])
rm_df
```

Out[107…

	ridge_coef_1
0	23.534734
1	21.811780
2	36.252918
3	-48.152734
4	9.393079
…	…
64	75.149177
65	-57.579683
66	1.148651
67	25.902982
68	-55.117531

69 rows × 1 columns

GridSearchCV for Ridge; Choosing best alpha value

In [108…
```
from sklearn.model_selection import GridSearchCV
```

In [109…
```
alpha_space = np.linspace(0.01, 1, 50)
alpha_space
```

Out[109…
```
array([0.01      , 0.03020408, 0.05040816, 0.07061224, 0.09081633,
       0.11102041, 0.13122449, 0.15142857, 0.17163265, 0.19183673,
       0.21204082, 0.2322449 , 0.25244898, 0.27265306, 0.29285714,
       0.31306122, 0.33326531, 0.35346939, 0.37367347, 0.39387755,
       0.41408163, 0.43428571, 0.4544898 , 0.47469388, 0.49489796,
       0.51510204, 0.53530612, 0.5555102 , 0.57571429, 0.59591837,
       0.61612245, 0.63632653, 0.65653061, 0.67673469, 0.69693878,
       0.71714286, 0.73734694, 0.75755102, 0.7777551 , 0.79795918,
       0.81816327, 0.83836735, 0.85857143, 0.87877551, 0.89897959,
       0.91918367, 0.93938776, 0.95959184, 0.97979592, 1.        ])
```

In [110…
```
ridge_model = Ridge()
```

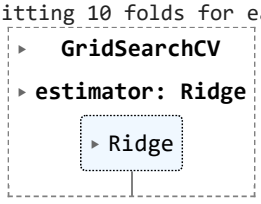
In [111…
```
param_grid = {"alpha":alpha_space}
```

In [112…
```
grid_ridge = GridSearchCV(estimator=ridge_model,
                          param_grid=param_grid,
                          scoring='neg_root_mean_squared_error',
                          cv=10,
                          verbose=1,
                          return_train_score=True)
```

In [113…
```
grid_ridge.fit(X_train_scaled, y_train)
```

```
Fitting 10 folds for each of 50 candidates, totalling 500 fits
```

Out[113…



In [114…
```
grid_ridge.best_params_
```

Out[114…
```
{'alpha': 0.01}
```

In [115…
```
grid_ridge.best_index_
```

Out[115…
```
0
```

In [116…
```
pd.DataFrame(grid_ridge.cv_results_).loc[0, ["mean_test_score", "mean_train_score"]]
```

Out[116…
```
mean_test_score    -13.999045
mean_train_score    -13.82016
Name: 0, dtype: object
```

In [117…
```
grid_ridge.best_score_
```

Out[117…
```
-13.99904474803903
```

In [118…
```
y_pred = grid_ridge.predict(X_test_scaled)
y_train_pred = grid_ridge.predict(X_train_scaled)
```

In [119…
```
ridgeGrid_score = train_val(y_train, y_train_pred, y_test, y_pred, "ridgeGrid")
ridgeGrid_score
```

Out[119…

	ridgeGrid_train	ridgeGrid_test
R2	0.944163	0.936900
mae	7.015601	7.623278
mse	190.880188	217.040191
rmse	13.815940	14.732284

In [120…
```
pd.concat([ridge_score, ridgeGrid_score], axis=1)
```

Out[120…

	ridge_train	ridge_test	ridgeGrid_train	ridgeGrid_test
R2	0.936997	0.927938	0.944163	0.936900
mae	7.652171	8.404607	7.015601	7.623278
mse	215.378640	247.865625	190.880188	217.040191
rmse	14.675784	15.743749	13.815940	14.732284

In [121…
```
rgm = Ridge(alpha=0.01).fit(X_train_scaled, y_train)
rgm.coef_
```

Out[121…
```
array([ 148.39184628,  -75.539314  ,    1.89073059,  -39.2025673 ,
         80.34305849, -144.4062634 ,  173.68518471, -168.7889454 ,
        150.97510173,   11.56968096, -102.9757516 , -294.03955376,
        -33.72089117,  158.21866864, -320.35270253, -127.63245589,
        246.17252803,   -1.27365154,  -10.75809842, -247.52584117,
        -82.61895431,  -93.47045404,  271.25978543,   92.3647927 ,
        128.09208909,  195.27977936,  -81.51564726,  -36.46598937,
        144.61310434,  197.14779166,   17.45337141,  231.29930997,
       -170.15034968, -199.58120167, -210.87517716,  228.60016708,
        140.85750453,  171.97473442,  112.84838809,  -43.660485  ,
        137.25103388,  627.4177336 , -249.20137435,  -54.64930227,
        -37.38520592, -233.61726129,  133.66930241, -166.89405436,
       -348.29370689,  102.16266973, -682.52353428,   76.31481337,
        -43.6210163 ,  -30.0612061 ,  -49.09925845,  -70.8037206 ,
       -121.43848718,  214.72161791,   44.6031287 ,   31.98253493,
         18.84673054,  165.83388604,  -65.83764861,  -92.61352443,
        352.84175776, -272.18974268,   33.83073756,  172.50092204,
         41.48031392])
```

In [122…
```
rgm_df = pd.DataFrame(rgm.coef_,  columns = ["ridge_coef_0.01"])
rgm_df
```

Out[122…

	ridge_coef_0.01
0	148.391846
1	-75.539314
2	1.890731
3	-39.202567
4	80.343058
…	…
64	352.841758
65	-272.189743
66	33.830738
67	172.500922
68	41.480314

69 rows × 1 columns

In [123…
```
pd.concat([rm_df, rgm_df], axis = 1)
```

Out[123…

	ridge_coef_1	ridge_coef_0.01
0	23.534734	148.391846
1	21.811780	-75.539314
2	36.252918	1.890731
3	-48.152734	-39.202567
4	9.393079	80.343058
…	…	…
64	75.149177	352.841758
65	-57.579683	-272.189743
66	1.148651	33.830738
67	25.902982	172.500922
68	-55.117531	41.480314

69 rows × 2 columns

Lasso Regression

In [124…
```
from sklearn.linear_model import Lasso
```

In [125…
```
# Set the Lasso Regression Model

lasso_model = Lasso(alpha=1)
```

In [126…
```
# Fit the Model

lasso_model = lasso_model.fit(X_train_scaled, y_train)
lasso_model
```

Out[126…

```
Lasso
Lasso(alpha=1)
```

In [127…

```
y_pred = lasso_model.predict(X_test_scaled)
y_train_pred = lasso_model.predict(X_train_scaled)
```

In [128…

```
ls_score = train_val(y_train, y_train_pred, y_test, y_pred, "lasso")
ls_score
```

Out[128…

	lasso_train	lasso_test
R2	0.908211	0.903149
mae	11.471661	11.815889
mse	313.786812	333.132733
rmse	17.714029	18.251924

In [129…

```
lasso_model.coef_
```

Out[129…

```
array([  0.        ,   0.        ,   0.        , -31.55638666,
         0.        ,   0.        ,   0.        ,   0.        ,
         0.        ,  15.41126594,  -0.        ,   0.        ,
         0.        ,  -0.        ,   0.        ,   0.        ,
         0.        ,   0.        ,   0.        ,   0.        ,
         0.        ,   0.        ,   3.90259382,  -0.        ,
         0.        ,   0.        ,   0.        ,   0.        ,
         0.        ,  -0.        ,  -0.        ,   0.        ,
        -0.        ,   0.        ,   0.        ,   0.        ,
         4.32641728,   0.        ,   0.        ,   0.        ,
         0.        ,   0.        ,   0.        ,  -0.        ,
         0.        ,   0.        ,   0.        ,   0.        ,
         0.        ,  -0.        ,   0.        ,   0.        ,
         0.        ,  -0.        ,   0.        ,   0.        ,
         0.        ,   0.        ,   0.        ,  -0.        ,
         0.        ,   3.53742748,  -0.        ,   0.        ,
        -0.        ,   0.        ,   0.        ,  -0.        ,
         0.43177847])
```

In [130…

```
pd.concat([ridge_score, ridgeGrid_score, ls_score], axis=1)
```

Out[130…

	ridge_train	ridge_test	ridgeGrid_train	ridgeGrid_test	lasso_train	lasso_test
R2	0.936997	0.927938	0.944163	0.936900	0.908211	0.903149
mae	7.652171	8.404607	7.015601	7.623278	11.471661	11.815889
mse	215.378640	247.865625	190.880188	217.040191	313.786812	333.132733
rmse	14.675784	15.743749	13.815940	14.732284	17.714029	18.251924

In [131…

```
lsm_df = pd.DataFrame(lasso_model.coef_, columns = ["lasso_coef_1"])
```

In [132…

```
pd.concat([rm_df, rgm_df, lsm_df], axis = 1)
```

Out[132…

	ridge_coef_1	ridge_coef_0.01	lasso_coef_1
0	23.534734	148.391846	0.000000
1	21.811780	-75.539314	0.000000
2	36.252918	1.890731	0.000000
3	-48.152734	-39.202567	-31.556387
4	9.393079	80.343058	0.000000
...	...	...	...
64	75.149177	352.841758	-0.000000
65	-57.579683	-272.189743	0.000000
66	1.148651	33.830738	0.000000
67	25.902982	172.500922	-0.000000
68	-55.117531	41.480314	0.431778

69 rows × 3 columns

GridSearchCV for Lasso; Choosing best alpha value

In [133…
```
#Set the Lasso model

lasso_model = Lasso()
```

In [134…
```
param_grid = {"alpha":alpha_space} #defined above

grid_lasso = GridSearchCV(estimator=lasso_model,
                          param_grid=param_grid,
                          scoring='neg_root_mean_squared_error',
                          cv=5,
                          verbose=1,
                          return_train_score=True)
```

In [135…
```
# Fit the gridS Lasso Model

grid_lasso.fit(X_train_scaled, y_train)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
```

Out[135…

GridSearchCV
estimator: Lasso
Lasso

In [136…
```
grid_lasso.best_params_
```

Out[136…
```
{'alpha': 0.01}
```

In [137…
```
grid_lasso.best_index_
```

Out[137…
```
0
```

In [138…
```
pd.DataFrame(grid_lasso.cv_results_).loc[0, ["mean_test_score", "mean_train_score"]]
```

Out[138…
```
mean_test_score    -15.409405
mean_train_score   -15.296438
Name: 0, dtype: object
```

In [139…
```
grid_lasso.best_score_
```

Out[139… -15.409405447365188

In [140…
```
lsm = Lasso(alpha=0.01).fit(X_train_scaled, y_train)
lsm.coef_
```

Out[140…
```
array([ 3.93142732e+01,  2.10912481e+01,  1.14983007e+01, -2.53020981e+01,
       -2.27897244e+01, -1.12054793e+01, -1.09530392e+01, -4.93984805e+00,
        1.71996540e+01,  0.00000000e+00, -6.63218532e+00, -2.34753709e+01,
        3.33849133e+00,  0.00000000e+00,  1.52022208e+01,  0.00000000e+00,
        0.00000000e+00,  1.85606738e-01,  0.00000000e+00, -2.58258572e+00,
       -2.21179175e+00, -9.78175881e+00,  5.93328094e+00, -3.45177471e+00,
        5.71666927e+00,  8.34475432e+00, -0.00000000e+00, -4.71000807e+00,
        9.55470786e+00,  0.00000000e+00, -2.58754254e+01,  2.66309706e+01,
       -1.54022343e+01,  4.21110401e+01, -3.57094223e+00, -3.81286016e+00,
        2.19816905e+01, -3.36639023e+00, -2.44340158e+01,  5.80324585e-01,
       -0.00000000e+00,  2.13417112e+01,  0.00000000e+00, -0.00000000e+00,
       -1.63854747e+01, -0.00000000e+00, -2.47907933e+00, -0.00000000e+00,
       -0.00000000e+00, -8.65255730e+00,  7.38152165e+00, -1.30119602e+00,
       -1.85016334e+00,  1.13035941e+01, -3.19931209e-04,  1.80901870e+01,
       -2.20597669e+00,  0.00000000e+00,  7.96346759e+00, -6.55602118e+00,
       -0.00000000e+00,  0.00000000e+00, -6.77542798e+00,  9.69209206e+00,
        1.73040576e+01, -3.68066420e+01,  0.00000000e+00, -2.28353745e+01,
       -1.48381049e+01])
```

In [141…
```
lgm_df = pd.DataFrame(lsm.coef_, columns=["lasso_coef_0.01"])
```

In [142…
```
pd.concat([rm_df, rgm_df, lsm_df, lgm_df], axis = 1)
```

Out[142…

	ridge_coef_1	ridge_coef_0.01	lasso_coef_1	lasso_coef_0.01
0	23.534734	148.391846	0.000000	39.314273
1	21.811780	-75.539314	0.000000	21.091248
2	36.252918	1.890731	0.000000	11.498301
3	-48.152734	-39.202567	-31.556387	-25.302098
4	9.393079	80.343058	0.000000	-22.789724
…	…	…	…	…
64	75.149177	352.841758	-0.000000	17.304058
65	-57.579683	-272.189743	0.000000	-36.806642
66	1.148651	33.830738	0.000000	0.000000
67	25.902982	172.500922	-0.000000	-22.835374
68	-55.117531	41.480314	0.431778	-14.838105

69 rows × 4 columns

In [143…
```
# Prediction

y_pred = grid_lasso.predict(X_test_scaled)
y_train_pred = grid_lasso.predict(X_train_scaled)
```

In [144…
```
lsg_score = train_val(y_train, y_train_pred, y_test, y_pred, "grid_lasso")
lsg_score
```

Out[144…

	grid_lasso_train	grid_lasso_test
R2	0.931439	0.923899
mae	8.526906	9.113804
mse	234.378526	261.758631
rmse	15.309426	16.178956

In [145…
```
pd.concat([ridge_score, ridgeGrid_score, ls_score, lsg_score], axis=1)
```

Out[145…

	ridge_train	ridge_test	ridgeGrid_train	ridgeGrid_test	lasso_train	lasso_test	grid_lasso_train	gr
R2	0.936997	0.927938	0.944163	0.936900	0.908211	0.903149	0.931439	
mae	7.652171	8.404607	7.015601	7.623278	11.471661	11.815889	8.526906	
mse	215.378640	247.865625	190.880188	217.040191	313.786812	333.132733	234.378526	
rmse	14.675784	15.743749	13.815940	14.732284	17.714029	18.251924	15.309426	

Final Model and Prediction

Model

In [146…
```
# Scale the Data

final_scaler = StandardScaler()
X_scaled = final_scaler.fit_transform(poly_converter.fit_transform(X))
```

In [147…
```
# Set the Final Model

final_model = Lasso(alpha=0.01)  #grid_lasso (Lasso model built over 28 features with our best a
```

In [148…
```
# Fit the Final Model

final_model.fit(X_scaled, y)
```

Out[148…
```
Lasso(alpha=0.01)
```

In [149…
```
# Prediction of the Final Model

final_model.predict(X_scaled)
```

Out[149…
```
array([202.32031288, 226.03676396, 129.64073402, ..., 234.05166746,
       229.09360749, 238.89697646])
```

In [150…
```
final_model.coef_
```

Out[150…
```
array([ 36.72826234,  22.89748574,   8.57593383, -25.15151208,
       -21.80654934, -11.32249581,  -9.90626373,  -4.29136716,
        15.90883766,   0.        ,  -7.12051605, -21.17772776,
         3.43262021,  -0.55717158,  16.5599249 ,   0.        ,
         0.        ,   0.        ,  -1.2027637 ,  -3.82215834,
        -2.8531007 ,  -9.16232562,   5.88947659,  -2.72866572,
         4.71066454,  10.77547276,  -0.        ,  -1.09327297,
         7.95584525,   0.        , -27.45868298,  25.01927537,
       -15.37992859,  38.6166295 ,  -1.03958338,  -3.18606714,
        19.57316841,  -4.68043557, -23.86059701,   0.        ,
        -0.10805156,  18.68248114,   0.        ,  -0.        ,
       -15.39914546,  -0.        ,  -2.98029356,  -0.        ,
        -0.        ,  -8.59617613,   6.66861927,   0.        ,
        -0.95286808,   9.7364982 ,  -0.        ,  15.99084398,
        -1.48506904,   0.77492297,   8.60489526,  -6.80889799,
         0.        ,   0.        ,  -6.32833095,  10.65882153,
        17.2356954 , -34.71311705,   0.        , -21.57025617,
       -12.54795801])
```

In [151…
```
final_model.intercept_
```

Out[151…
```
250.58469871360683
```

In [152…
```
X.head()
```

Out[152…

	engine_size	cylinders	fuel_cons_comb	fuel_cons_comb_mpg
0	2.0	4	8.5	33
1	2.4	4	9.6	29
2	1.5	4	5.9	48
3	3.5	6	11.1	25
4	3.5	6	10.6	27

In [153…
```
df.loc[[4]]
```

Out[153…

	make	model	vehicle_class	engine_size	cylinders	transmission	fuel_type	fuel_cons_city	fuel_cons_
4	ACURA	RDX AWD	SUV - SMALL	3.5	6	AS6	Z	12.1	

Predicting

In [154…
```
new_data = [[3.5, 12.1, 8.7, 10.6]]
new_data
```

Out[154…
```
[[3.5, 12.1, 8.7, 10.6]]
```

In [155…
```
poly_sample = poly_converter.transform(new_data)
poly_sample
```

Out[155…
```
array([[3.50000000e+00, 1.21000000e+01, 8.70000000e+00, 1.06000000e+01,
        1.22500000e+01, 4.23500000e+01, 3.04500000e+01, 3.71000000e+01,
        1.46410000e+02, 1.05270000e+02, 1.28260000e+02, 7.56900000e+01,
        9.22200000e+01, 1.12360000e+02, 4.28750000e+01, 1.48225000e+02,
        1.06575000e+02, 1.29850000e+02, 5.12435000e+02, 3.68445000e+02,
        4.48910000e+02, 2.64915000e+02, 3.22770000e+02, 3.93260000e+02,
        1.77156100e+03, 1.27376700e+03, 1.55194600e+03, 9.15849000e+02,
        1.11586200e+03, 1.35955600e+03, 6.58503000e+02, 8.02314000e+02,
        9.77532000e+02, 1.19101600e+03, 1.50062500e+02, 5.18787500e+02,
        3.73012500e+02, 4.54475000e+02, 1.79352250e+03, 1.28955750e+03,
        1.57118500e+03, 9.27202500e+02, 1.12969500e+03, 1.37641000e+03,
        6.20046350e+03, 4.45818450e+03, 5.43181100e+03, 3.20547150e+03,
        3.90551700e+03, 4.75844600e+03, 2.30476050e+03, 2.80809900e+03,
        3.42136200e+03, 4.16855600e+03, 2.14358881e+04, 1.54125807e+04,
        1.87785466e+04, 1.10817729e+04, 1.35019302e+04, 1.64506276e+04,
        7.96788630e+03, 9.70799940e+03, 1.18281372e+04, 1.44112936e+04,
        5.72897610e+03, 6.98013180e+03, 8.50452840e+03, 1.03618392e+04,
        1.26247696e+04]])
```

In [156…
```
scaled_sample = final_scaler.transform(poly_sample)
scaled_sample
```

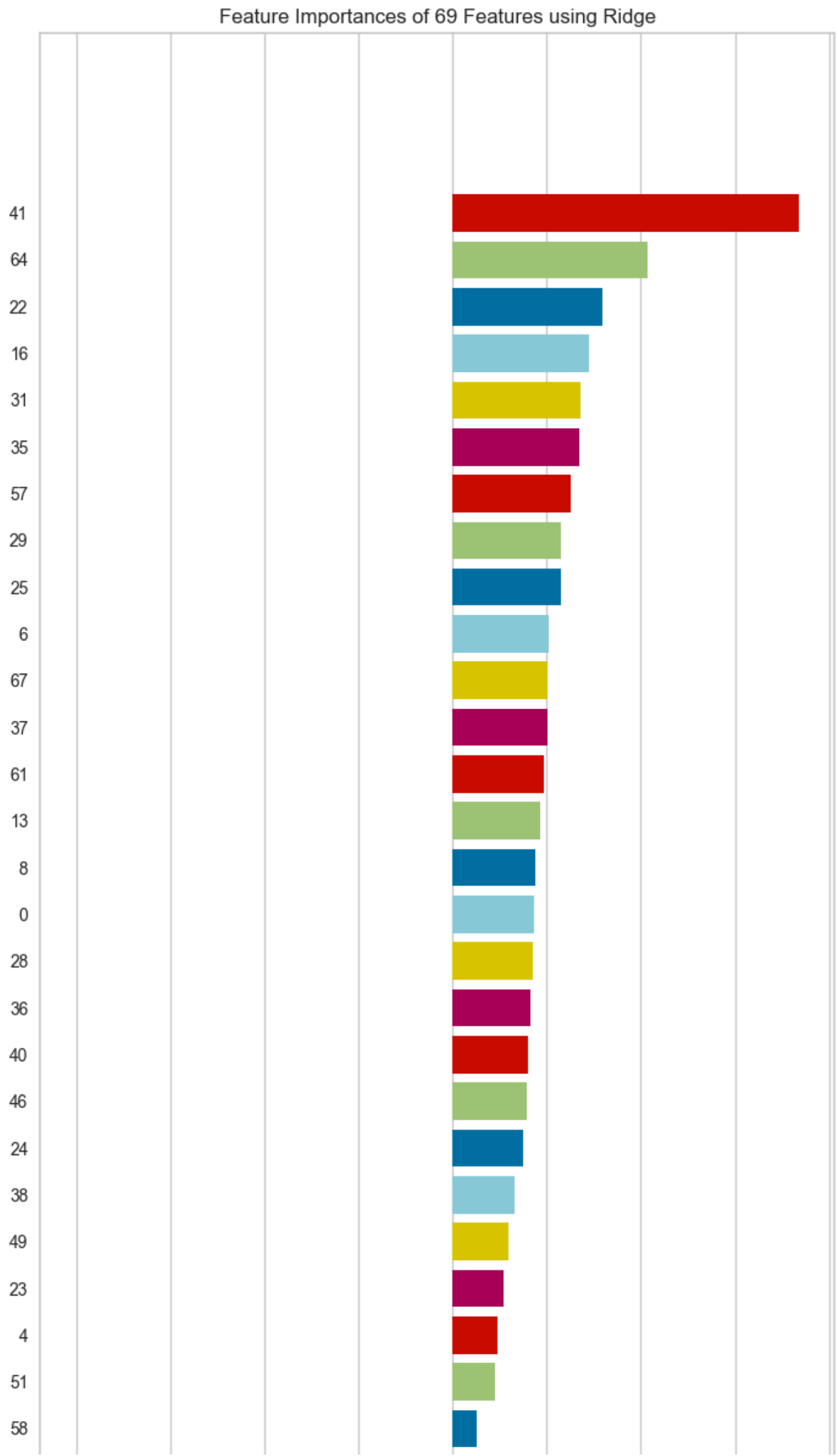
Out[156…
```
array([[ 2.51043221e-01,  3.54722175e+00, -7.86592975e-01,
        -2.33449629e+00,  4.20258727e-02,  1.45294115e+00,
        -2.90439294e-01, -2.07731384e+00,  4.50420741e+00,
         1.03559282e+00, -6.02276288e-01, -7.35575271e-01,
        -5.86081736e+01, -1.51383251e+00, -1.21983313e-01,
         6.42965545e-01, -2.74015663e-01, -7.88482810e-01,
         2.25843619e+00,  4.34757532e-01, -9.07753134e-02,
        -4.63284330e-01, -1.48898696e+00, -2.43349716e+00,
         5.31668308e+00,  1.94600141e+00,  1.74285042e+00,
         1.28123365e-01, -9.07934976e-01, -1.71633442e+00,
        -6.40070374e-01, -2.80600957e+00, -3.31347594e+00,
        -9.69374967e-01, -2.24618757e-01,  2.00279972e-01,
        -3.05132015e-01, -5.30763799e-01,  1.09483757e+00,
         9.52080499e-02, -1.39833961e-01, -3.96621308e-01,
        -7.63124967e-01, -1.51617548e+00,  2.77494239e+00,
         9.31811464e-01,  8.04691124e-01, -4.98680683e-02,
        -4.03703334e-01, -1.41414686e+00, -4.77209892e-01,
        -1.09113584e+00, -3.29381883e+00, -1.38000323e+00,
         5.77400261e+00,  2.49733740e+00,  2.68615873e+00,
         6.91422749e-01,  5.22911093e-01, -6.17173882e-01,
        -2.15421078e-01, -8.20999587e-01, -3.72357855e+00,
        -1.12598342e+00, -5.23585156e-01, -1.43931294e+00,
        -3.88729669e+01, -1.67710672e+00, -6.15808251e-01]])
```

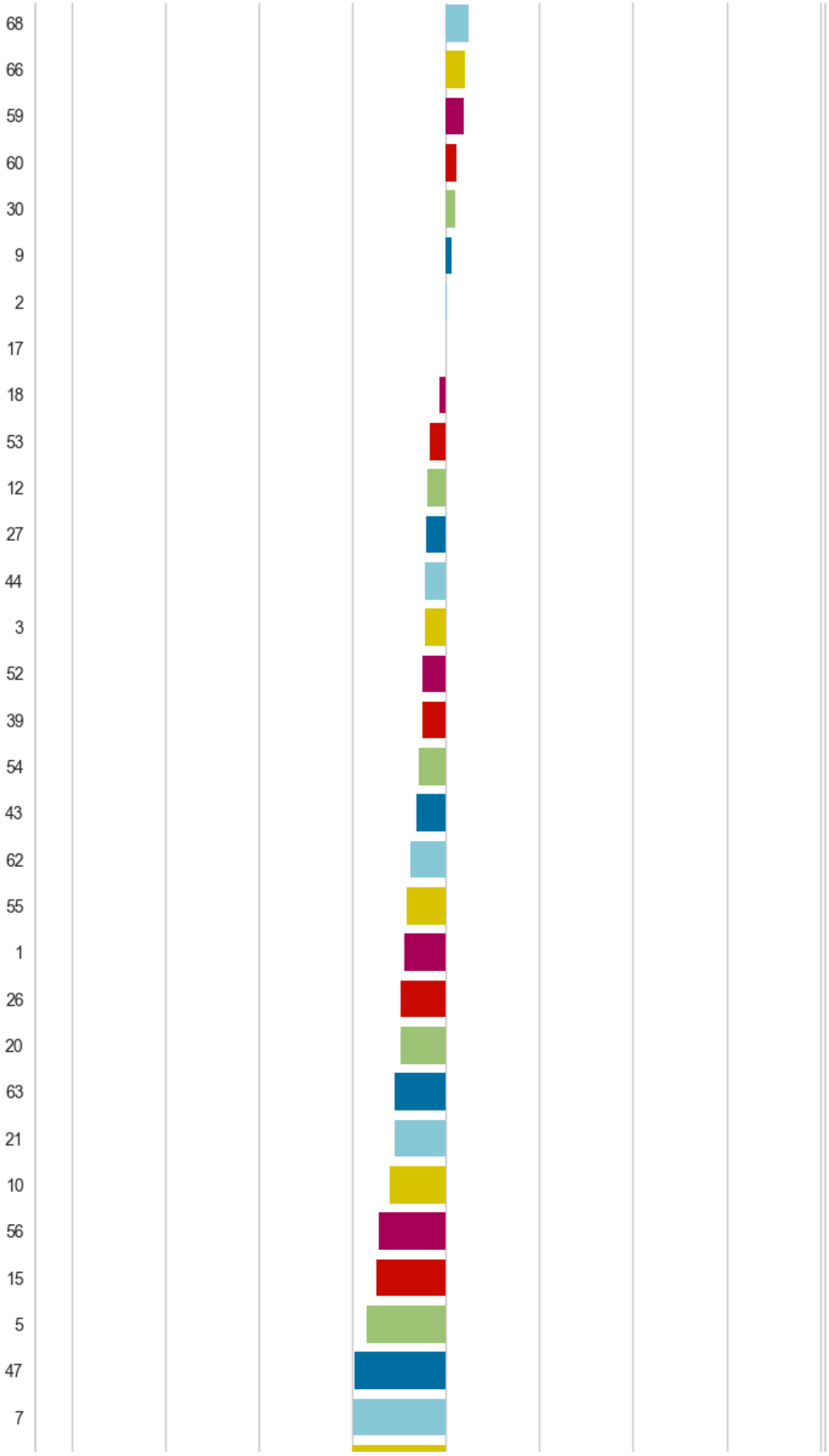
In [157…
```
final_model.predict(scaled_sample)
```

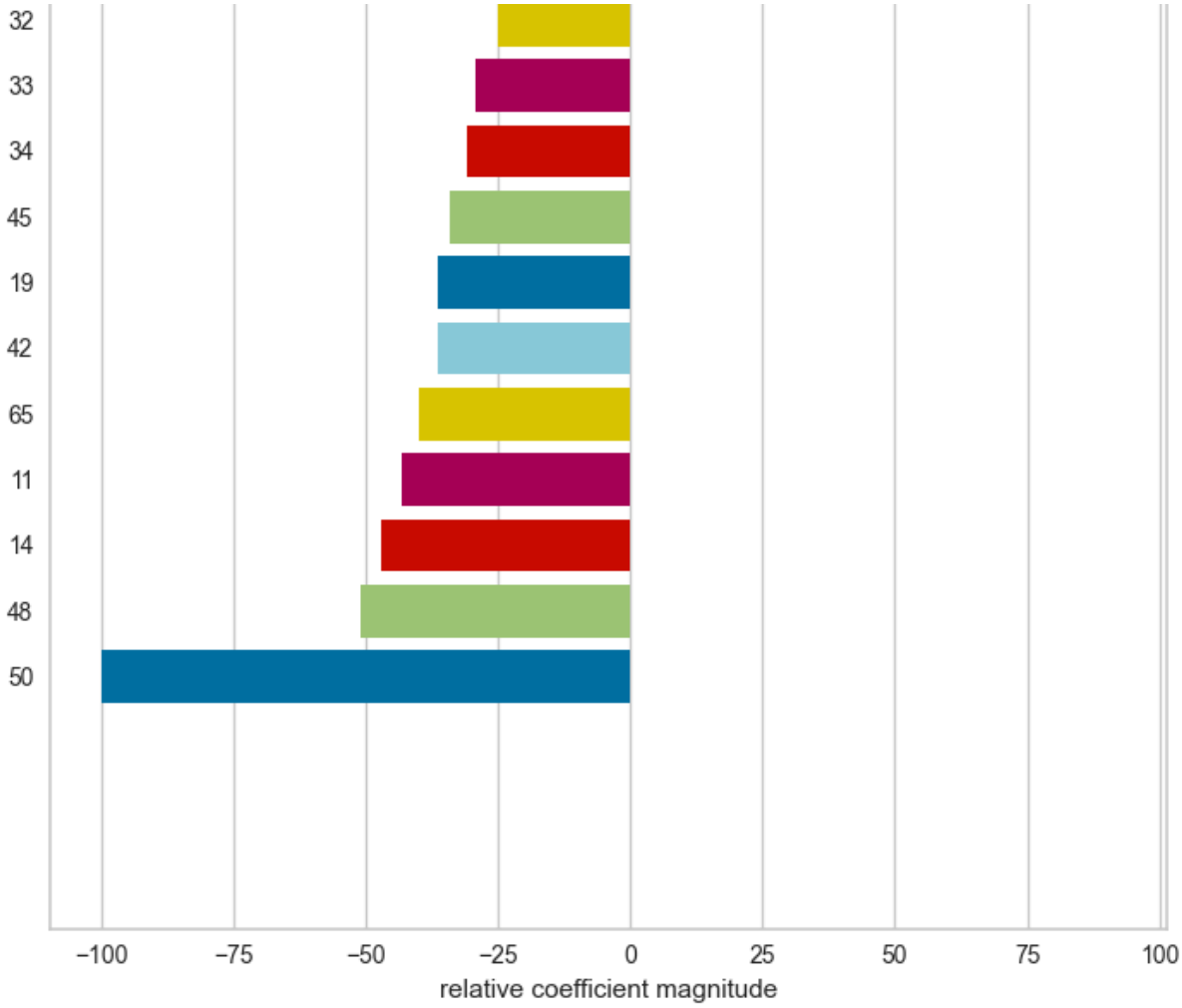
Out[157…
```
array([331.8565249])
```

Feature importances with Ridge

In [158…
```
from yellowbrick.model_selection import FeatureImportances
from yellowbrick.features import RadViz

model = Ridge(alpha=0.01)

viz = FeatureImportances(model, labels=pd.DataFrame(X_train).columns)
visualizer = RadViz(size=(720, 3000))
viz.fit(X_train_scaled, y_train)
viz.show();
```


Feature Importances of 69 Features using Ridge
41
64
22
16
31
35
57
29
25
6
67
37
61
13
8
0
28
36
40
46
24
38
49
23
4
51
58

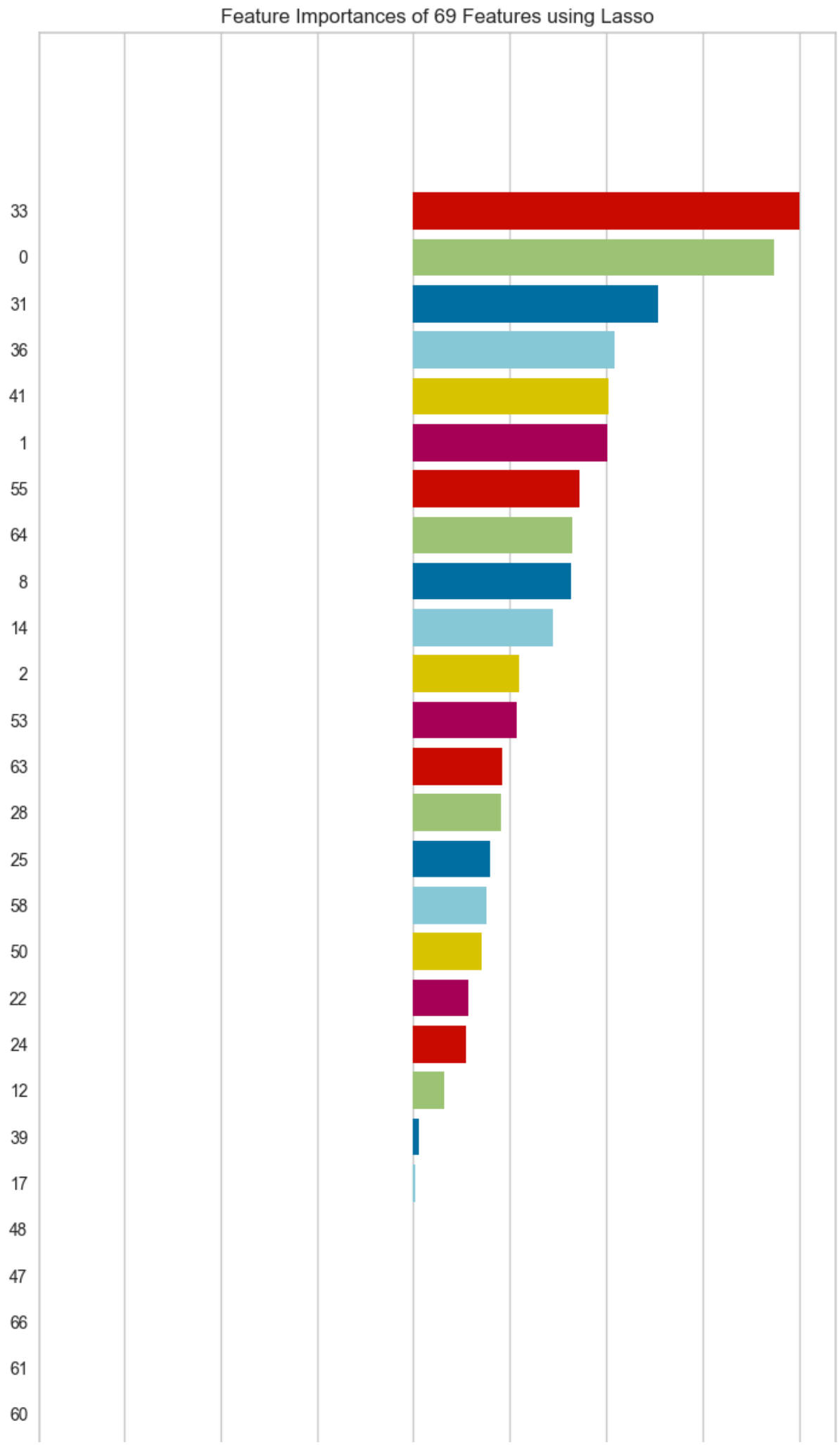

68
66
59
60
30
9
2
17
18
53
12
27
44
3
52
39
54
43
62
55
1
26
20
63
21
10
56
15
5
47
7

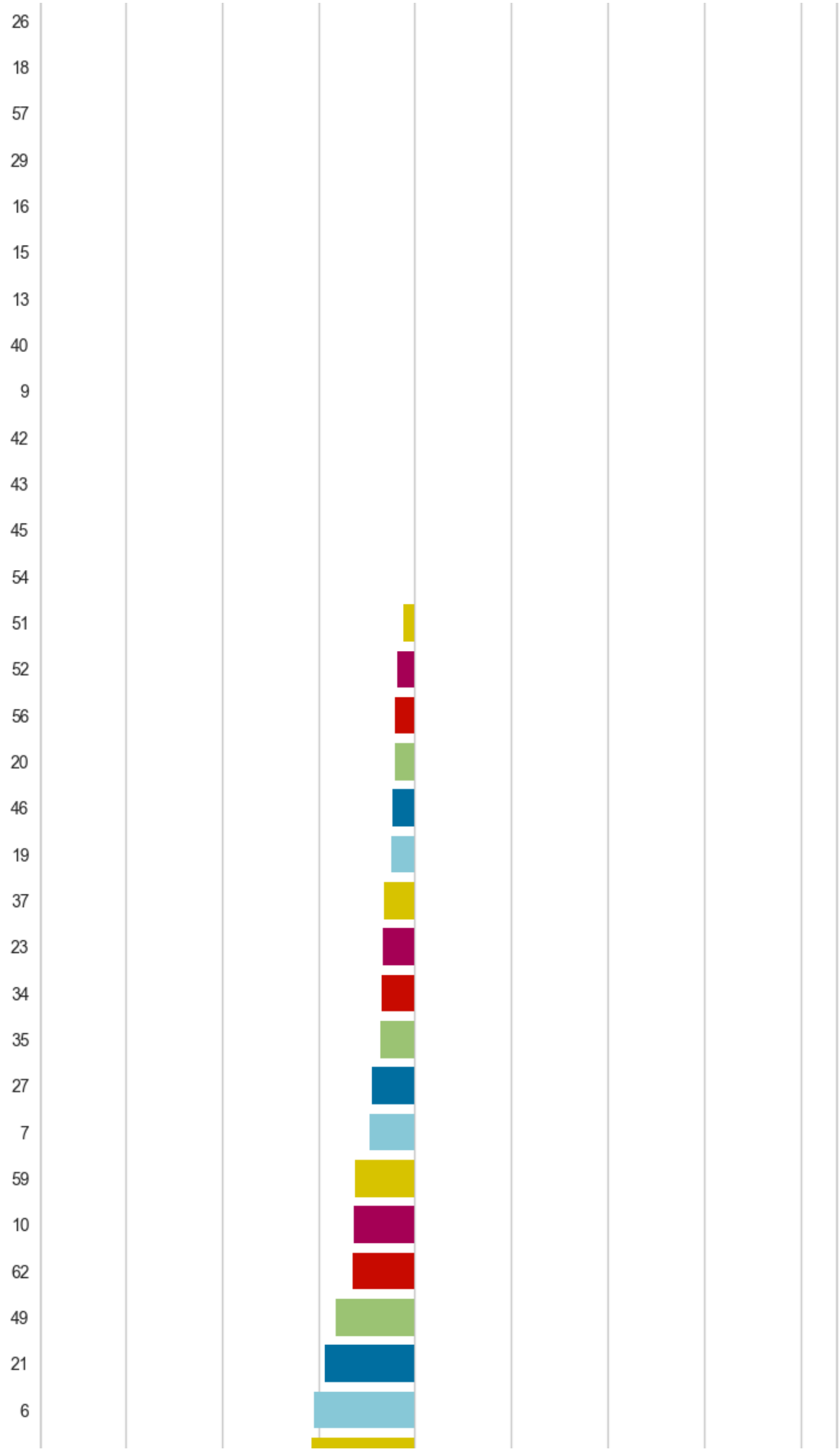


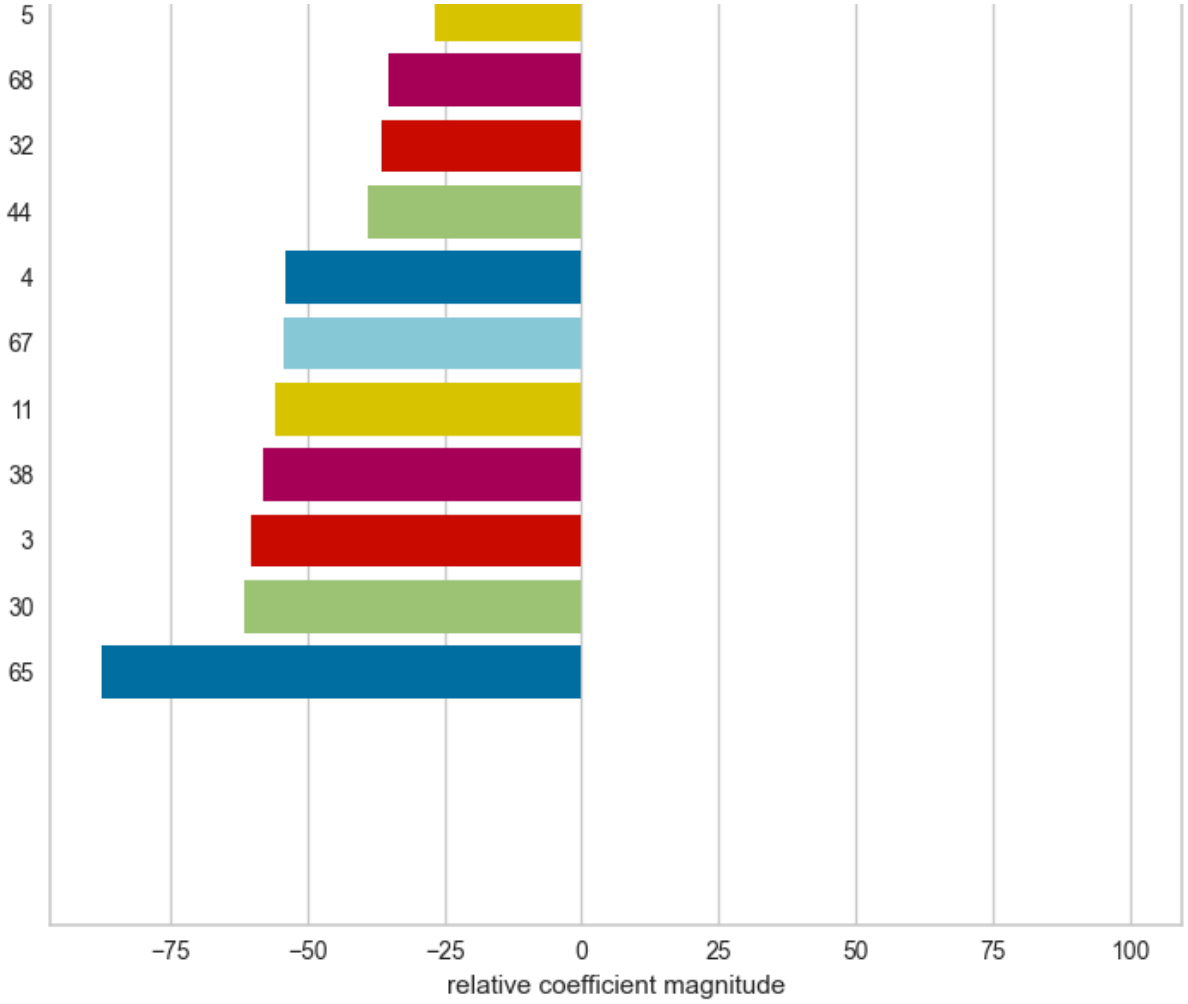
Feature importances with Lasso

In [159…

```
from yellowbrick.model_selection import FeatureImportances
from yellowbrick.features import RadViz

model = Lasso(alpha=0.01)

viz = FeatureImportances(model, labels=pd.DataFrame(X_train).columns)
visualizer = RadViz(size=(720, 3000))
viz.fit(X_train_scaled, y_train)
viz.show();

# Lasso modeli için en iyi scoru veren alpha=0.01 hyperparametresiyle modeli FeatureImportances
# fonksiyonunun içine veriyoruz.  labels olarak da df'in columns isimlerini veriyoruz.

# Lasso üzerinden feature selection yapabiliriz
```


Feature Importances of 69 Features using Lasso
33
0
31
36
41
1
55
64
8
14
2
53
63
28
25
58
50
22
24
12
39
17
48
47
66
61
60


26
18
57
29
16
15
13
40
9
42
43
45
54
51
52
56
20
46
19
37
23
34
35
27
7
59
10
62
49
21
6



Thank you...

Duygu Jones | Data Scientist | 2024

Follow me: duygujones.com | Linkedin | GitHub | Kaggle | Medium | Tableau

In [ ]: