
Practice Hands-on case study: Linear Regression

Welcome to the Hands-on case study on Linear Regression. In this case study, we aim to construct a linear model that explains the relationship a car's mileage (mpg) has with its other attributes

---

Dataset:

There are 8 variables in the data:

- mpg: miles per gallon
- cyl: number of cylinders
- disp: engine displacement (cu. inches) or engine size
- hp: horsepower
- wt: vehicle weight (lbs.)
- acc: time taken to accelerate from O to 60 mph (sec.)
- yr: model year
- car name: car model name
- Also provided are the car labels (types)
- Missing data values are marked by series of question marks.

Import Libraries

In [33]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
#%matplotlib inline
import seaborn as sns
```

In [34]:
```
!pwd
```

```
/Users/obaozai/Data/GitHub/Regression and Prediction
```

Load and review data

In [35]:
```
data = pd.read_csv("auto_mpg.csv")
data.shape
```

Out[35]:
```
(398, 9)
```

In [36]:
```
data.head()
```

Out[36]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	
0	18.0	8	307.0	130	3504	12.0	70	1	ch c
1	15.0	8	350.0	165	3693	11.5	70	1	
2	18.0	8	318.0	150	3436	11.0	70	1	pl s
3	16.0	8	304.0	150	3433	12.0	70	1	re
4	17.0	8	302.0	140	3449	10.5	70	1	

In [37]:
```
data.isnull().sum()
```

Out[37]:
```
mpg             0
cylinders       0
displacement    0
horsepower      0
weight          0
acceleration    0
model year      0
origin          0
car name        0
dtype: int64
```

In [38]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   mpg           398 non-null    float64
 1   cylinders     398 non-null    int64  
 2   displacement  398 non-null    float64
 3   horsepower    398 non-null    object 
 4   weight        398 non-null    int64  
 5   acceleration  398 non-null    float64
 6   model year    398 non-null    int64  
 7   origin        398 non-null    int64  
 8   car name      398 non-null    object 
dtypes: float64(3), int64(4), object(2)
memory usage: 28.1+ KB
```

In [39]:
```
#dropping/ignoring car_name
data = data.drop('car name', axis=1)
# Also replacing the categorical var with actual values
data['origin'] = data['origin'].replace({1: 'america', 2: 'europe', 3: 'asia
data.head(20)
```

Out[39]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin
0	18.0	8	307.0	130	3504	12.0	70	america
1	15.0	8	350.0	165	3693	11.5	70	america
2	18.0	8	318.0	150	3436	11.0	70	america
3	16.0	8	304.0	150	3433	12.0	70	america
4	17.0	8	302.0	140	3449	10.5	70	america
5	15.0	8	429.0	198	4341	10.0	70	america
6	14.0	8	454.0	220	4354	9.0	70	america
7	14.0	8	440.0	215	4312	8.5	70	america
8	14.0	8	455.0	225	4425	10.0	70	america
9	15.0	8	390.0	190	3850	8.5	70	america
10	15.0	8	383.0	170	3563	10.0	70	america
11	14.0	8	340.0	160	3609	8.0	70	america
12	15.0	8	400.0	150	3761	9.5	70	america
13	14.0	8	455.0	225	3086	10.0	70	america
14	24.0	4	113.0	95	2372	15.0	70	asia
15	22.0	6	198.0	95	2833	15.5	70	america
16	18.0	6	199.0	97	2774	15.5	70	america
17	21.0	6	200.0	85	2587	16.0	70	america
18	27.0	4	97.0	88	2130	14.5	70	asia
19	26.0	4	97.0	46	1835	20.5	70	europe

Create Dummy Variables

Values like 'america' cannot be read into an equation. Using substitutes like 1 for america, 2 for europe and 3 for asia would end up implying that european cars fall exactly half way between american and asian cars! we dont want to impose such an baseless assumption!

So we create 3 simple true or false columns with titles equivalent to "Is this car America?", "Is this care European?" and "Is this car Asian?". These will be used as independent variables without imposing any kind of ordering between the three regions.

In [40]:
```
data = pd.get_dummies(data, columns=['origin'])
data.head()
```

Out[40]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin_am
0	18.0	8	307.0	130	3504	12.0	70	
1	15.0	8	350.0	165	3693	11.5	70	
2	18.0	8	318.0	150	3436	11.0	70	
3	16.0	8	304.0	150	3433	12.0	70	
4	17.0	8	302.0	140	3449	10.5	70	

Dealing with Missing Values

In [41]:
```
#A quick summary of the data columns
data.describe()
```

Out[41]:

	mpg	cylinders	displacement	weight	acceleration	model yea
count	398.000000	398.000000	398.000000	398.000000	398.000000	398.000000
mean	23.514573	5.454774	193.425879	2970.424623	15.568090	76.010050
std	7.815984	1.701004	104.269838	846.841774	2.757689	3.697627
min	9.000000	3.000000	68.000000	1613.000000	8.000000	70.000000
25%	17.500000	4.000000	104.250000	2223.750000	13.825000	73.000000
50%	23.000000	4.000000	148.500000	2803.500000	15.500000	76.000000
75%	29.000000	8.000000	262.000000	3608.000000	17.175000	79.000000
max	46.600000	8.000000	455.000000	5140.000000	24.800000	82.000000

In [42]:
```
# hp is missing cause it does not seem to be reqcognized as a numerical colu
data.dtypes
```

Out[42]:
```
mpg              float64
cylinders          int64
displacement     float64
horsepower        object
weight             int64
acceleration     float64
model year         int64
origin_america      bool
origin_asia         bool
origin_europe       bool
dtype: object
```

Q.2 The method used to check whether an entry of a column is a numerical value or is it missing?

In [43]:
```
# if the string is made of digits store True else False hint: use isdigit()
# hpIsDigit = pd.DataFrame(data.horsepower.str._________())
# data[hpIsDigit['horsepower'] == False]   # from temp take only those rows


# Assuming `data` is your existing DataFrame and it has a column named 'hors
hpIsDigit = pd.DataFrame(data.horsepower.str.isdigit())

# Rename the column if needed
hpIsDigit.columns = ['hpIsDigit']

print(hpIsDigit)
```

```
     hpIsDigit
0         True
1         True
2         True
3         True
4         True
..         ...
393       True
394       True
395       True
396       True
397       True

[398 rows x 1 columns]
```

In [44]:
```
# Replace missing values represented by '?' with NaN
data = data.replace('?', np.nan)

# Check if 'horsepower' column exists
if 'horsepower' in data.columns:
    # Create a new column 'hpIsDigit' to store True if the string is made of
    data['hpIsDigit'] = data['horsepower'].str.isdigit()
    print(data)
else:
    print("Column 'horsepower' does not exist in the DataFrame.")
```

```
      mpg  cylinders  displacement horsepower  weight  acceleration  \
0    18.0          8         307.0        130    3504          12.0
1    15.0          8         350.0        165    3693          11.5
2    18.0          8         318.0        150    3436          11.0
3    16.0          8         304.0        150    3433          12.0
4    17.0          8         302.0        140    3449          10.5
..    ...        ...           ...        ...     ...           ...
393  27.0          4         140.0         86    2790          15.6
394  44.0          4          97.0         52    2130          24.6
395  32.0          4         135.0         84    2295          11.6
396  28.0          4         120.0         79    2625          18.6
397  31.0          4         119.0         82    2720          19.4

     model year  origin_america  origin_asia  origin_europe hpIsDigit
0            70            True        False          False      True
1            70            True        False          False      True
2            70            True        False          False      True
3            70            True        False          False      True
4            70            True        False          False      True
..          ...             ...          ...            ...       ...
393          82            True        False          False      True
394          82           False        False           True      True
395          82            True        False          False      True
396          82            True        False          False      True
397          82            True        False          False      True

[398 rows x 11 columns]
```

There are various ways to handle missing values. Drop the rows, replace missing values with median values etc. of the 398 rows 6 have NAN in the hp column. We could drop those 6 rows - which might not be a good idea under all situations

In [45]:
```python
#instead of dropping the rows, lets replace the missing values with median v

# Replace missing values represented by '?' with NaN
data['horsepower'] = data['horsepower'].replace('?', np.nan)

# Convert the 'horsepower' column to numeric, forcing non-numeric values to
data['horsepower'] = pd.to_numeric(data['horsepower'], errors='coerce')

# Calculate the median of the 'horsepower' column, skipping NaN values
median_hp = data['horsepower'].median()

# Replace NaN values with the median
data['horsepower'] = data['horsepower'].fillna(median_hp)

# Create a new column 'hpIsDigit' to store True if the string is made of dig
data['hpIsDigit'] = data['horsepower'].apply(lambda x: str(x).isdigit())

print(data)
```

```
      mpg  cylinders  displacement  horsepower  weight  acceleration  \
0    18.0          8         307.0       130.0    3504          12.0
1    15.0          8         350.0       165.0    3693          11.5
2    18.0          8         318.0       150.0    3436          11.0
3    16.0          8         304.0       150.0    3433          12.0
4    17.0          8         302.0       140.0    3449          10.5
..    ...        ...           ...         ...     ...           ...
393  27.0          4         140.0        86.0    2790          15.6
394  44.0          4          97.0        52.0    2130          24.6
395  32.0          4         135.0        84.0    2295          11.6
396  28.0          4         120.0        79.0    2625          18.6
397  31.0          4         119.0        82.0    2720          19.4

     model year  origin_america  origin_asia  origin_europe  hpIsDigit
0            70            True        False          False      False
1            70            True        False          False      False
2            70            True        False          False      False
3            70            True        False          False      False
4            70            True        False          False      False
..          ...             ...          ...            ...        ...
393          82            True        False          False      False
394          82           False        False           True      False
395          82            True        False          False      False
396          82            True        False          False      False
397          82            True        False          False      False

[398 rows x 11 columns]
```

Filling the missing values with median value

In [46]:
```
# replace the missing values with median value.
# Note, we do not need to specify the column names below
# every column's missing value is replaced with that column's median respect

medianFiller = lambda x: x.fillna(x.median())
data = data.apply(medianFiller,axis=0)



data['horsepower'] = data['horsepower'].astype('float64')  # converting the
```

In [47]:
```
data.head()
```

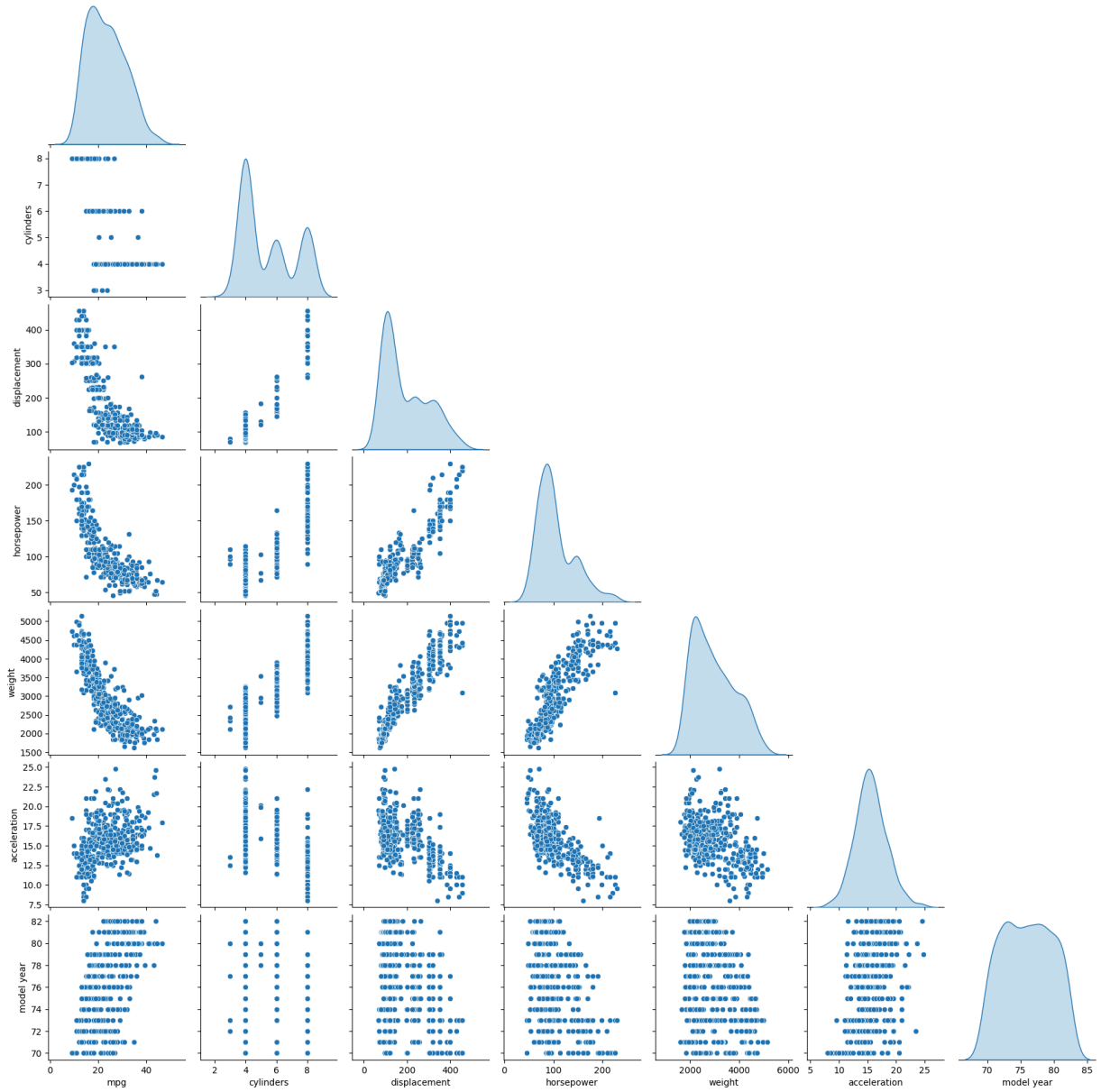
Out[47]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin_am
0	18.0	8	307.0	130.0	3504	12.0	70	
1	15.0	8	350.0	165.0	3693	11.5	70	
2	18.0	8	318.0	150.0	3436	11.0	70	
3	16.0	8	304.0	150.0	3433	12.0	70	
4	17.0	8	302.0	140.0	3449	10.5	70	

BiVariate Plots

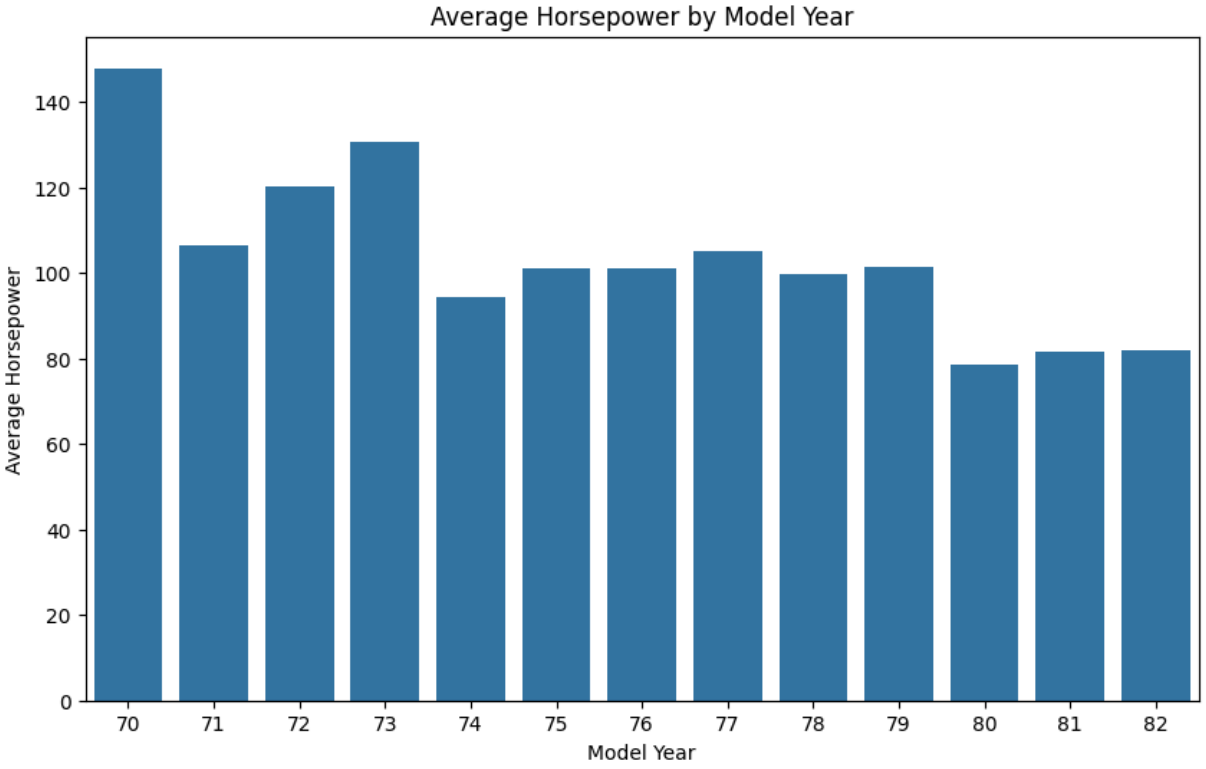
A bivariate analysis among the different variables can be done using scatter matrix plot. Seaborn libs create a dashboard reflecting useful information about the dimensions. The result can be stored as a .png file.

In [48]:
```
# Replace missing values represented by '?' with NaN
data['horsepower'] = data['horsepower'].replace('?', np.nan)

# Convert the 'horsepower' column to numeric, forcing non-numeric values to
data['horsepower'] = pd.to_numeric(data['horsepower'], errors='coerce')

# Calculate the median of the 'horsepower' column, skipping NaN values
median_hp = data['horsepower'].median()

# Replace NaN values with the median
data['horsepower'] = data['horsepower'].fillna(median_hp)

# Select the first 7 columns for the pairplot
data_attr = data.iloc[:, 0:7]

# Plot the pairplot with density curves on the diagonal
sns.pairplot(data_attr, diag_kind='kde', corner=True)

# Show the plot
plt.show()
```

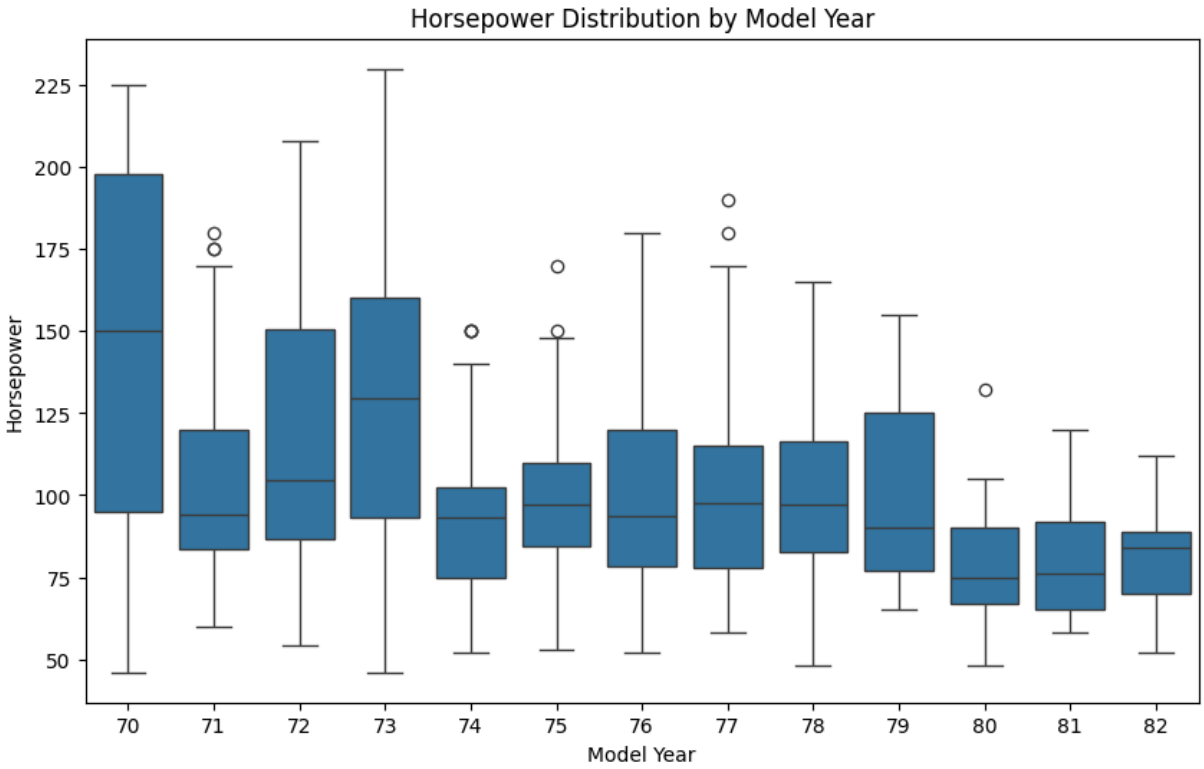


In [49]:
```
# Replace missing values represented by '?' with NaN
data['horsepower'] = data['horsepower'].replace('?', np.nan)

# Convert the 'horsepower' column to numeric, forcing non-numeric values to
data['horsepower'] = pd.to_numeric(data['horsepower'], errors='coerce')

# Calculate the median of the 'horsepower' column, skipping NaN values
median_hp = data['horsepower'].median()

# Replace NaN values with the median
data['horsepower'] = data['horsepower'].fillna(median_hp)

# Check if 'model year' column exists
if 'model year' in data.columns:
    # Group by 'model year' and calculate the mean 'horsepower'
    mean_hp_by_year = data.groupby('model year')['horsepower'].mean().reset_

    # Create a bar plot
    plt.figure(figsize=(10, 6))
```

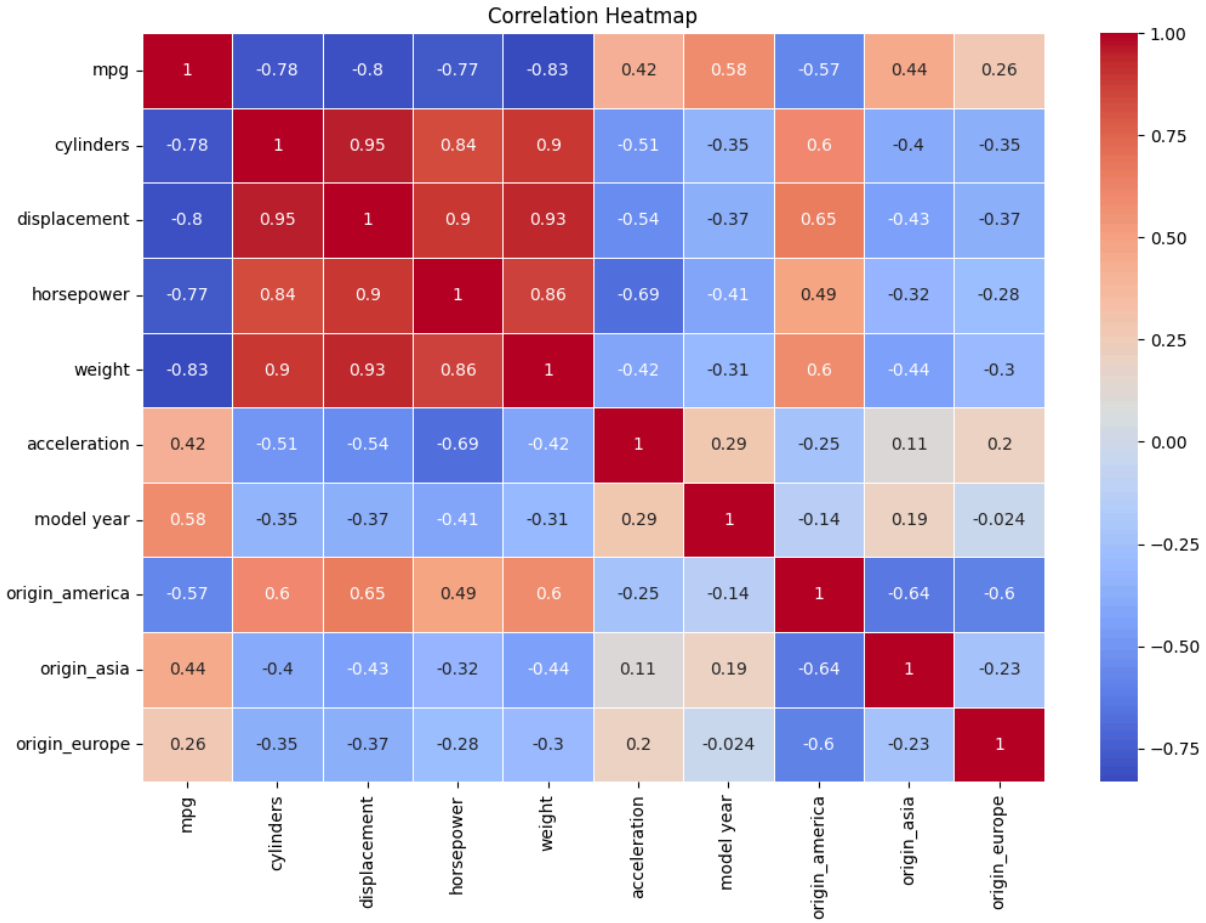
```
    sns.barplot(x='model year', y='horsepower', data=mean_hp_by_year)
    plt.xlabel('Model Year')
    plt.ylabel('Average Horsepower')
    plt.title('Average Horsepower by Model Year')
    plt.show()
else:
    print("Column 'model year' does not exist in the DataFrame.")
```



In [50]:
```
# Replace missing values represented by '?' with NaN
data['horsepower'] = data['horsepower'].replace('?', np.nan)

# Convert the 'horsepower' column to numeric, forcing non-numeric values to
data['horsepower'] = pd.to_numeric(data['horsepower'], errors='coerce')

# Calculate the median of the 'horsepower' column, skipping NaN values
median_hp = data['horsepower'].median()

# Replace NaN values with the median
data['horsepower'] = data['horsepower'].fillna(median_hp)

# Check if 'model year' column exists
if 'model year' in data.columns:
    # Create a box plot
    plt.figure(figsize=(10, 6))
    sns.boxplot(x='model year', y='horsepower', data=data)
    plt.xlabel('Model Year')
    plt.ylabel('Horsepower')
    plt.title('Horsepower Distribution by Model Year')
    plt.show()
else:
    print("Column 'model year' does not exist in the DataFrame.")
```



In [51]:
```
# Replace missing values represented by '?' with NaN
data['horsepower'] = data['horsepower'].replace('?', np.nan)

# Convert the 'horsepower' column to numeric, forcing non-numeric values to
data['horsepower'] = pd.to_numeric(data['horsepower'], errors='coerce')

# Calculate the median of the 'horsepower' column, skipping NaN values
median_hp = data['horsepower'].median()

# Replace NaN values with the median
data['horsepower'] = data['horsepower'].fillna(median_hp)

# Select only the numeric columns for the correlation heatmap
numeric_cols = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',

# Calculate the correlation matrix
corr_matrix = data[numeric_cols].corr()

# Create a heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap')
plt.show()
```



Observation between 'mpg' and other attributes indicate the relationship is not really linear. However, the plots also indicate that linearity would still capture quite a bit of useful information/pattern. Several assumptions of classical linear regression seem to be violated, including the assumption of no Heteroscedasticity

Split Data

In [52]:
```
# lets build our linear model
# independant variables
X = data.drop(columns = {'mpg','origin_europe'})
# the dependent variable
y = data['mpg']
```

In [53]:
```
X
```

Out[53]:

	cylinders	displacement	horsepower	weight	acceleration	model year	origin_americ
0	8	307.0	130.0	3504	12.0	70	Tru
1	8	350.0	165.0	3693	11.5	70	Tru
2	8	318.0	150.0	3436	11.0	70	Tru
3	8	304.0	150.0	3433	12.0	70	Tru
4	8	302.0	140.0	3449	10.5	70	Tru
...	...	...	...	...	...	...	
393	4	140.0	86.0	2790	15.6	82	Tru
394	4	97.0	52.0	2130	24.6	82	Fals
395	4	135.0	84.0	2295	11.6	82	Tru
396	4	120.0	79.0	2625	18.6	82	Tru
397	4	119.0	82.0	2720	19.4	82	Tru

398 rows × 9 columns

In [54]:
```
y
```

Out[54]:
```
0      18.0
1      15.0
2      18.0
3      16.0
4      17.0
       ...
393    27.0
394    44.0
395    32.0
396    28.0
397    31.0
Name: mpg, Length: 398, dtype: float64
```

In [55]:
```
# Sklearn package's model_selection have a function train_test_split() is us
from sklearn.model_selection import train_test_split


# Split X and y into training and test set(out of sample data) in 70:30 rati

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, ra
```

Q.3 & 4 Create linear regression model using statsmodels OLS and interpretate coefficient

In [56]:
```
# Import libraries for building linear regression model
from statsmodels.graphics.gofplots import ProbPlot
from statsmodels.formula.api import ols
```

```python
import statsmodels.api as sm

# Define the target variable and features
X_train = data[['cylinders', 'displacement', 'horsepower', 'weight', 'accele
y_train = data['mpg']

# Add the intercept to data
X_train_ols = sm.add_constant(X_train)
X_test_ols = sm.add_constant(X_test) # added to fix error in cell bellow

# Create the model
model1 = ols('mpg ~ cylinders + displacement + horsepower + weight + acceler

# Get the model summary
print(model1.summary())
```

OLS Regression Results

Dep. Variable:	mpg	R-squared:	0.809
Model:	OLS	Adj. R-squared:	0.806
Method:	Least Squares	F-statistic:	275.5
Date:	Sat, 18 Jan 2025	Prob (F-statistic):	4.75e-137
Time:	08:46:22	Log-Likelihood:	-1053.5
No. Observations:	398	AIC:	2121.
Df Residuals:	391	BIC:	2149.
Df Model:	6		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-15.0292	4.717	-3.186	0.002	-24.303	-5.755
cylinders	-0.2517	0.331	-0.761	0.447	-0.902	0.398
displacement	0.0069	0.007	0.938	0.349	-0.008	0.021
horsepower	0.0028	0.014	0.204	0.839	-0.024	0.029
weight	-0.0070	0.001	-10.546	0.000	-0.008	-0.006
acceleration	0.0930	0.100	0.929	0.354	-0.104	0.290
Q("model year")	0.7577	0.052	14.532	0.000	0.655	0.860

Omnibus:	36.844	Durbin-Watson:	1.216
Prob(Omnibus):	0.000	Jarque-Bera (JB):	56.980
Skew:	0.620	Prob(JB):	4.24e-13
Kurtosis:	4.378	Cond. No.	8.47e+04

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 8.47e+04. This might indicate that there

```
are
strong multicollinearity or other numerical problems.
```

- Not all the variables are statistically significant to predict the outcome variable. To check which are statistically significant or have predictive power to predict the target variable, we need to check the `p-value` against all the independent variables.
- **Interpreting the Regression Results:**

1. **Adjusted. R-squared**: It reflects the fit of the model.
    - R-squared values range from 0 to 1, where a higher value generally indicates a better fit, assuming certain conditions are met.
2. **coeff**: It represents the change in the output Y due to a change of one unit in the variable (everything else held constant).
3. **std err**: It reflects the level of accuracy of the coefficients.
    - The lower it is, the more accurate the coefficients are.
4. **P >|t|**: It is p-value.
    - Pr(>|t|) : For each independent feature there is a null hypothesis and alternate hypothesis

    Ho : Independent feature is not significant

    Ha : Independent feature is significant
    - A p-value of less than 0.05 is considered to be statistically significant.
5. **Confidence Interval**: It represents the range in which our coefficients are likely to fall (with a likelihood of 95%).

- To be able to make statistical inferences from our model, **we will have to test the significance of the regression coefficients and linear regression assumptions.**

Checking the performance of the model on the train and test data set

In [57]:
```
X_test_ols
```

Out[57]:

	const	cylinders	displacement	horsepower	weight	acceleration	model year	origin
174	1.0	6	171.0	97.0	2984	14.5	75	
359	1.0	4	141.0	80.0	3230	20.4	81	
250	1.0	8	318.0	140.0	3735	13.2	78	
274	1.0	5	131.0	103.0	2830	15.9	78	
283	1.0	6	232.0	90.0	3265	18.2	79	
...	...	...	...	...	...	...	...	
382	1.0	4	108.0	70.0	2245	16.9	82	
39	1.0	8	400.0	175.0	4464	11.5	71	
171	1.0	4	134.0	96.0	2702	13.5	75	
271	1.0	4	156.0	105.0	2745	16.7	78	
247	1.0	4	85.0	70.0	2070	18.6	78	

120 rows × 10 columns

In [58]:

```python
# RMSE
def rmse(predictions, targets):
    return np.sqrt(((targets - predictions) ** 2).mean())


# MAPE
def mape(predictions, targets):
    return np.mean(np.abs((targets - predictions)) / targets) * 100


# MAE
def mae(predictions, targets):
    return np.mean(np.abs((targets - predictions)))


# Model Performance on test and train data
def model_pref(olsmodel, x_train, x_test, y_train,y_test):

    # Insample Prediction
    y_pred_train = olsmodel.predict(x_train)
    y_observed_train = y_train

    # Prediction on test data
    y_pred_test = olsmodel.predict(x_test)
    y_observed_test = y_test

    print(
        pd.DataFrame(
            {
                "Data": ["Train", "Test"],
                "RMSE": [
```

```
                    rmse(y_pred_train, y_observed_train),
                    rmse(y_pred_test, y_observed_test),
                ],
                "MAE": [
                    mae(y_pred_train, y_observed_train),
                    mae(y_pred_test, y_observed_test),
                ],
                "MAPE": [
                    mape(y_pred_train, y_observed_train),
                    mape(y_pred_test, y_observed_test),
                ],
            }
        )
    )


# Checking model performance
model_pref(model1, X_train_ols, X_test_ols, y_train,y_test)
```

```
    Data      RMSE       MAE       MAPE
0  Train  3.414176  2.631155  12.142630
1   Test  3.121403  2.406793  11.164196
```

Observations:

- RMSE, MAE, and MAPE of train and test data are not very different, indicating that the **model is not overfitting and has generalized well.**

Question 5: Performing cross validation and comparing its average performance to OLS performance

In [59]:
```
# import the required function
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

# build the regression model using Sklearn Linear regression
linearregression = LinearRegression()

# Perform cross-validation
cv_Score11 = cross_val_score(linearregression, X_train, y_train, cv=10) # cv
cv_Score12 = cross_val_score(linearregression, X_train, y_train, cv=10, scor

print("RSquared: %0.3f (+/- %0.3f)" % (cv_Score11.mean(), cv_Score11.std() *
print("Mean Squared Error: %0.3f (+/- %0.3f)" % (-1 * cv_Score12.mean(), cv_
```

```
RSquared: 0.608 (+/- 0.546)
Mean Squared Error: 13.648 (+/- 17.616)
```

Get model Coefficients in a pandas dataframe with column 'Feature' having all the features and column 'Coefs' with all the corresponding Coefs. Write the regression equation.

In [60]:
```
coef = model1.params
```

```
coef
```

Out[60]:
```
Intercept          -15.029189
cylinders           -0.251750
displacement         0.006909
horsepower           0.002765
weight              -0.006984
acceleration         0.093028
Q("model year")      0.757657
dtype: float64
```

In [61]:
```
# Let us write the equation of the fit
Equation = "log (car_mileage) ="
print(Equation, end='\t')
for i in range(len(coef)):
    print('(', coef[i], ') * ', coef.index[i], '+', end = ' ')
```

```
log (car_mileage) =     ( -15.029188840182897 ) *  Intercept + ( -0.25174966
88027132 ) *  cylinders + ( 0.00690927480908097 ) *  displacement + ( 0.0027
646512674675897 ) *  horsepower + ( -0.006984066613400087 ) *  weight + ( 0.
09302770239320923 ) *  acceleration + ( 0.7576574674575892 ) *  Q("model yea
r") +
```

```
/var/folders/q0/xfs5xjxx50xdjh4tzn1psdnw0000gn/T/ipykernel_2786/1038366294.p
y:5: FutureWarning: Series.__getitem__ treating keys as positions is depreca
ted. In a future version, integer keys will always be treated as labels (con
sistent with DataFrame behavior). To access a value by position, use `ser.il
oc[pos]`
  print('(', coef[i], ') * ', coef.index[i], '+', end = ' ')
```

Building Decision Tree

In [62]:
```
#importing Decision tree regressor using sklearn

from sklearn.tree import DecisionTreeRegressor
```

In [63]:
```
# splitting the data in 70:30 ratio of train to test data
# separate the dependent and indepedent variable
Y1 = data['mpg']
X1 = data.drop(columns = {'mpg','origin_europe'})

X_train1, X_test1, y_train1, y_test1 = train_test_split(X1, Y1, test_size=0.
```

Question 6: Building Decision tree and Checking its performance

This code will:

Import the DecisionTreeRegressor from sklearn.tree. Define the Decision Tree Regressor with a specified random_state for reproducibility. Fit the Decision Tree Regressor to the training dataset (X_train1 and y_train1).

In [64]:
```
# Import the required function
from sklearn.tree import DecisionTreeRegressor

# Defining the Decision Tree Regressor
dt = DecisionTreeRegressor(random_state=1)

# Fitting Decision Tree Regressor to train dataset
dt.fit(X_train1, y_train1)
```

Out[64]:
```
DecisionTreeRegressor(random_state=1)
```

Checking model perform on the train and test dataset

In [65]:
```
model_pref(dt, X_train1, X_test1,y_train1,y_test1)
```

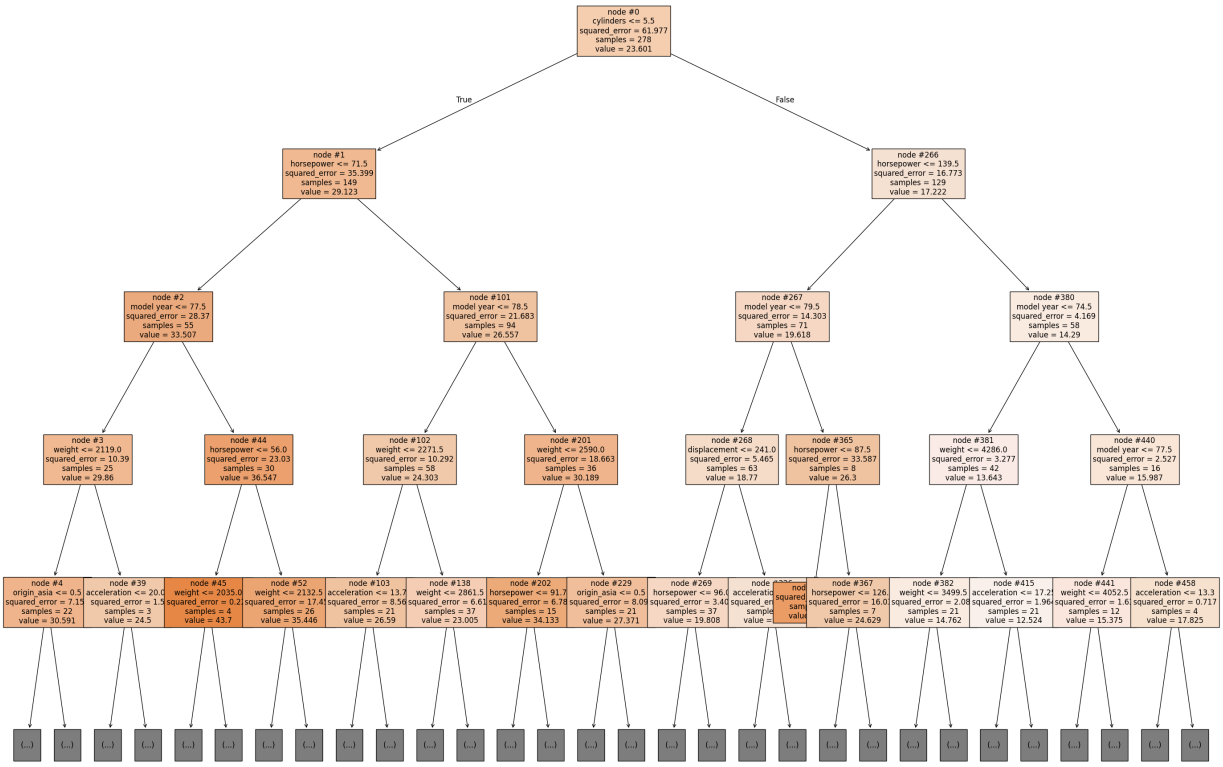
	Data	RMSE	MAE	MAPE
0	Train	0.00000	0.000000	0.000000
1	Test	4.18962	2.739167	11.850942

Observations:

- **The model seem to overfit the data** as rmse, mae and mape value of train data is 0, but that value for test data is much higher.

In [66]:
```
from sklearn.tree import plot_tree
```

In [67]:
```
features = list(X1.columns)

plt.figure(figsize=(35,25))
plot_tree(dt, max_depth=4, feature_names=features,filled=True,fontsize=12,nc
plt.show()
```



Let's plot the feature importance for each variable in the dataset and analyze the variables

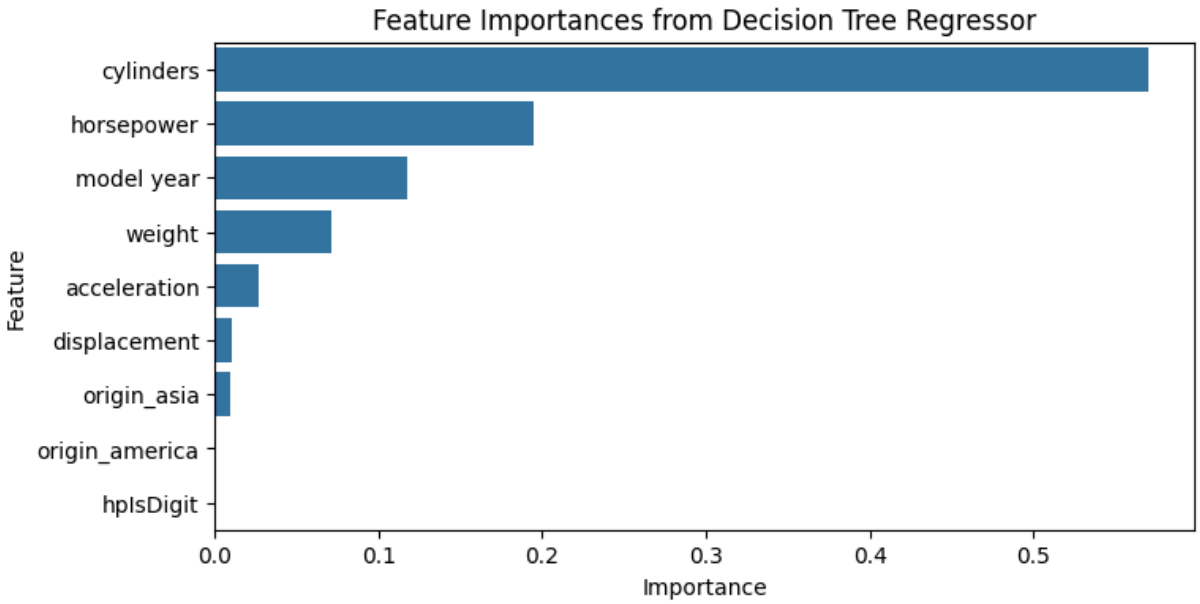
Checking Feature importance

This code will:

Import the necessary libraries for data manipulation and plotting. Find the feature importances from the fitted Decision Tree Regressor using the feature_importances_ attribute. Create a DataFrame for the feature importances and sort them in descending order. Plot the feature importances using a bar plot.

In [68]:
```
# Find feature importances from the decision tree
importances = dt.feature_importances_

# Create a DataFrame for the feature importances
columns = X1.columns
importance_df = pd.DataFrame(importances, index=columns, columns=['Importanc

# Plot the feature importances
plt.figure(figsize=(8, 4))
sns.barplot(x=importance_df.Importance, y=importance_df.index)
plt.xlabel('Importance')
plt.ylabel('Feature')
```

```
plt.title('Feature Importances from Decision Tree Regressor')
plt.show()
```



Building Random Forest

In [69]:
```
#importing random forest regressor usinf sklearn

from sklearn.ensemble import RandomForestRegressor
```

Parameters for regression

n_estimators: The number of trees in the forest.

min_samples_split: The minimum number of samples required to split an internal node:

max_depth The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

max_features{“auto”, “sqrt”, “log2”, 'None'}: The number of features to consider when looking for the best split.

- If “auto”, then max_features=sqrt(n_features).
- If “sqrt”, then max_features=sqrt(n_features) (same as “auto”).
- If “log2”, then max_features=log2(n_features).
- If None, then max_features=n_features.

This code will:

Import the RandomForestRegressor from sklearn.ensemble. Define the Random Forest Regressor with 100 estimators and a specified random_state for reproducibility. Fit the Random Forest Regressor to the training dataset (X_train1 and y_train1)

In [70]:
```
# Import the required function
from sklearn.ensemble import RandomForestRegressor

# Defining the Random Forest Regressor
rf = RandomForestRegressor(n_estimators=100, random_state=1)

# Hyperparameters, we have randomly chosen them for now but we can tune thes

# Fitting the Random Forest Regressor to the training dataset
rf.fit(X_train1, y_train1)
```

Out[70]:
```
RandomForestRegressor
RandomForestRegressor(random_state=1)
```

Q.7 Check performance of Random Forest

This code will:

Use the score method of the Random Forest Regressor to calculate the R^2 score on the test dataset. Print the R^2 score, which indicates how well the model predicts the target variable on the test dataset.

In [71]:
```
# checking model performance on test dataset
rf_score = rf.score(X_test1, y_test1)
print(f"R^2 Score on Test Dataset: {rf_score:.3f}")
```

```
R^2 Score on Test Dataset: 0.864
```

In [72]:
```
model_pref(rf, X_train1, X_test1,y_train1,y_test1)
```

```
    Data      RMSE       MAE      MAPE
0  Train  1.030134  0.718651  3.036687
1   Test  2.815511  1.960367  8.515241
```

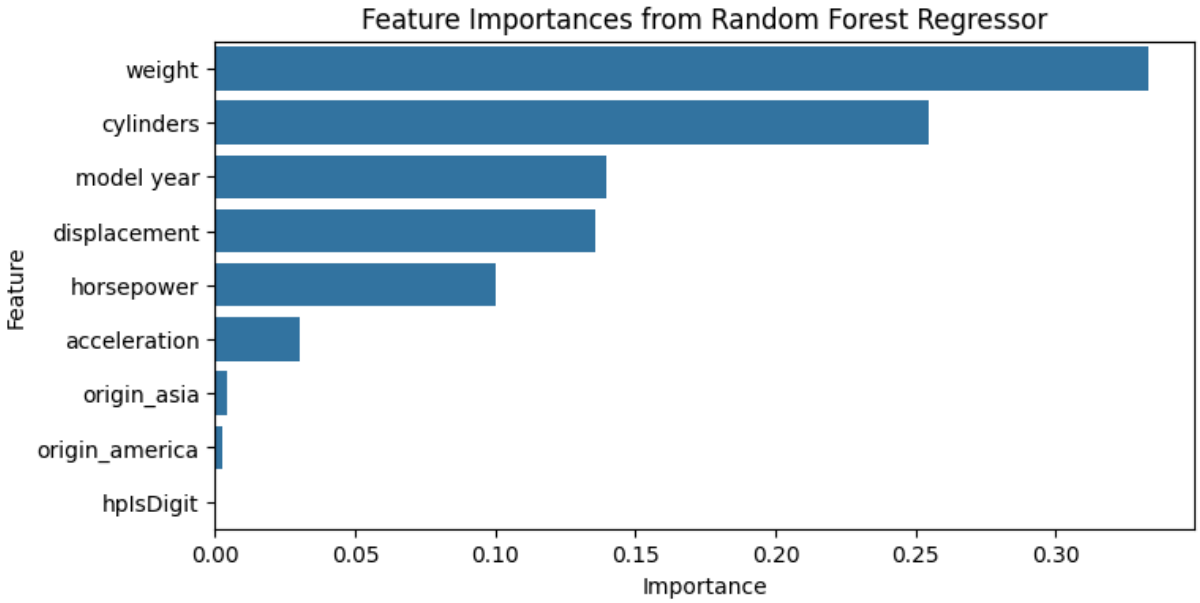
Question 8 & 9: Checking the feature importance of each variable in Random Forest and comparing to Decision Tree

This code will:

Use the feature_importances_ attribute of the Random Forest Regressor to get the feature importances. Create a DataFrame for the feature importances and sort them in

descending order. Plot the feature importances using a bar plot.

In [73]:
```
# print feature importance Random forest and complete the code
importances = rf.feature_importances_

columns = X1.columns
importance_df = pd.DataFrame(importances, index=columns, columns=['Importanc

plt.figure(figsize=(8, 4))
sns.barplot(x=importance_df.Importance, y=importance_df.index)
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importances from Random Forest Regressor')
plt.show()
```



Question 10: Comparing results of three model

In [74]:
```
print("Linear Regression")
model_pref(model1, X_train_ols, X_test_ols,y_train,y_test)
print("Decision tree")
model_pref(dt, X_train1, X_test1,y_train1,y_test1)
print("Random Forest")
model_pref(rf, X_train1, X_test1,y_train1,y_test1)
```

```
Linear Regression
    Data      RMSE       MAE       MAPE
0  Train  3.414176  2.631155  12.142630
1   Test  3.121403  2.406793  11.164196
Decision tree
    Data     RMSE       MAE       MAPE
0  Train  0.00000  0.000000   0.000000
1   Test  4.18962  2.739167  11.850942
Random Forest
    Data      RMSE       MAE      MAPE
0  Train  1.030134  0.718651  3.036687
1   Test  2.815511  1.960367  8.515241
```

Based on the provided observations, we can analyze the performance of the Linear Regression, Decision Tree, and Random Forest models on both the training and test datasets. Here are the observations:

Linear Regression Train Data: RMSE: 3.414 MAE: 2.631 MAPE: 12.143%

Test Data: RMSE: 3.121 MAE: 2.407 MAPE: 11.164% Decision Tree

Train Data: RMSE: 0.000 MAE: 0.000 MAPE: 0.000%

Test Data: RMSE: 4.190 MAE: 2.739 MAPE: 11.851% Random Forest

Train Data: RMSE: 1.030 MAE: 0.719 MAPE: 3.037%

Test Data: RMSE: 2.816 MAE: 1.960 MAPE: 8.515%

Observations:

Linear Regression:

The model performs reasonably well on both the training and test datasets. The RMSE, MAE, and MAPE values are relatively close between the training and test datasets, indicating that the model generalizes well to unseen data. Decision Tree:

The model perfectly fits the training data (RMSE, MAE, and MAPE are all 0), which is a clear sign of overfitting.

The performance on the test data is significantly worse compared to the training data, with higher RMSE, MAE, and MAPE values. This indicates that the model does not generalize well to unseen data. Random Forest:

The model performs well on both the training and test datasets.

The RMSE, MAE, and MAPE values are lower compared to the Linear Regression and Decision Tree models, indicating better performance.

The difference between the training and test performance is smaller compared to the Decision Tree, indicating that the Random Forest model generalizes better to unseen data.

Conclusion: Linear Regression provides a good balance between simplicity and performance, with reasonable generalization to unseen data.

Decision Tree suffers from overfitting, as it perfectly fits the training data but performs poorly on the test data.

Random Forest offers the best performance among the three models, with lower error metrics and better generalization to unseen data.

Based on these observations, the Random Forest model is the best choice for this dataset, as it provides the most accurate predictions and generalizes well to new data.

The variance in model performance due to modifying the `train_test_split` parameters can be assessed by changing the `test_size` and `random_state` values. Here's how each parameter affects the variance:

1. **`test_size`**:
    - This parameter determines the proportion of the dataset to include in the test split. Changing the `test_size` can affect the variance in model performance because different splits can lead to different training and test sets, which can impact the model's ability to generalize.
    - A smaller `test_size` means more data for training, which can lead to better model performance on the training set but might not provide a robust estimate of the model's performance on unseen data.
    - A larger `test_size` means less data for training, which can lead to higher variance in model performance because the model has less data to learn from.
2. **`random_state`**:
    - This parameter controls the shuffling applied to the data before applying the split. Changing the `random_state` will result in different splits of the data, which can lead to different training and test sets.
    - Different splits can lead to different model performance metrics, introducing variance in the results.

To quantify the variance in model performance due to different splits, you can perform multiple train-test splits with different `random_state` values and calculate the performance metrics for each split. Here's an example of how to do this:

This code will:

1. Perform multiple train-test splits with different `random_state` values.
2. Fit the Random Forest model to each split.
3. Calculate the performance metrics (RMSE, MAE, R^2) for each split.
4. Calculate the mean and standard deviation of the performance metrics to quantify the variance.

By analyzing the mean and standard deviation of the performance metrics, you can understand how much variance to expect in model performance due to different train-test splits.

In [75]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_scor
import numpy as np

# Define the number of iterations for different random states
n_iterations = 10

# Store the performance metrics for each iteration
rmse_list = []
mae_list = []
r2_list = []

# Perform multiple train-test splits with different random states
for i in range(n_iterations):
    # Split the data
    X_train1, X_test1, y_train1, y_test1 = train_test_split(X1, Y1, test_siz

    # Fit the Random Forest model
    rf = RandomForestRegressor(n_estimators=100, random_state=1)
    rf.fit(X_train1, y_train1)

    # Predict on the test set
    y_pred = rf.predict(X_test1)

    # Calculate performance metrics
    rmse = np.sqrt(mean_squared_error(y_test1, y_pred))
    mae = mean_absolute_error(y_test1, y_pred)
    r2 = r2_score(y_test1, y_pred)

    # Store the metrics
    rmse_list.append(rmse)
    mae_list.append(mae)
    r2_list.append(r2)

# Calculate the mean and standard deviation of the performance metrics
rmse_mean = np.mean(rmse_list)
rmse_std = np.std(rmse_list)
mae_mean = np.mean(mae_list)
mae_std = np.std(mae_list)
r2_mean = np.mean(r2_list)
r2_std = np.std(r2_list)

print(f"RMSE: {rmse_mean:.3f} (+/- {rmse_std:.3f})")
```

```
print(f"MAE: {mae_mean:.3f} (+/- {mae_std:.3f})")
print(f"R^2: {r2_mean:.3f} (+/- {r2_std:.3f})")
```

```
RMSE: 2.896 (+/- 0.166)
MAE: 2.025 (+/- 0.093)
R^2: 0.861 (+/- 0.018)
```

The relationship between test size and model performance is nuanced. Here are some key points to consider:

Test Size and Model Performance:

Smaller Test Size: When the test size is small, more data is available for training the model. This can lead to better performance on the training set because the model has more data to learn from. However, the test set might not be representative of the overall data distribution, leading to unreliable estimates of model performance on unseen data. Larger Test Size: When the test size is large, less data is available for training the model. This can lead to slightly worse performance on the training set because the model has less data to learn from. However, the test set is more likely to be representative of the overall data distribution, leading to more reliable estimates of model performance on unseen data. Overfitting and Underfitting:

Overfitting: If the model performs significantly better on the training set than on the test set, it might be overfitting. Overfitting occurs when the model learns the noise and details in the training data, which do not generalize to unseen data. Underfitting: If the model performs poorly on both the training set and the test set, it might be underfitting. Underfitting occurs when the model is too simple to capture the underlying patterns in the data. Balancing Test Size:

The goal is to find a balance where the test size is large enough to provide a reliable estimate of model performance on unseen data, but not so large that the training set becomes too small to train an effective model. A common practice is to use a test size of 20-30% of the total data. This provides a good balance between having enough data for training and having a representative test set. Example: Evaluating Different Test Sizes You can evaluate the impact of different test sizes on model performance by performing multiple train-test splits with different test sizes and comparing the performance metrics.

Here is an example:

This code will:

Evaluate the impact of different test sizes on model performance. Perform train-test splits with different test sizes. Fit the Random Forest model to each split. Calculate the performance metrics (RMSE, MAE, R^2) for each split. Print the performance metrics for each test size. By analyzing the results, you can determine the optimal test size that

provides a good balance between training data and a representative test set, leading to reliable estimates of model performance on unseen data.

This code will:

Evaluate the impact of different test sizes on model performance. Perform train-test splits with different test sizes. Fit the Random Forest model to each split. Calculate the performance metrics (RMSE, MAE, R^2) for each split. Print the performance metrics for each test size. By analyzing the results, you can determine the optimal test size that provides a good balance between training data and a representative test set, leading to reliable estimates of model performance on unseen data.

In [76]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_scor
import numpy as np

# Define different test sizes to evaluate
test_sizes = [0.1, 0.2, 0.3, 0.4, 0.5]

# Store the performance metrics for each test size
results = []

# Perform train-test splits with different test sizes
for test_size in test_sizes:
    # Split the data
    X_train1, X_test1, y_train1, y_test1 = train_test_split(X1, Y1, test_siz

    # Fit the Random Forest model
    rf = RandomForestRegressor(n_estimators=100, random_state=1)
    rf.fit(X_train1, y_train1)

    # Predict on the test set
    y_pred = rf.predict(X_test1)

    # Calculate performance metrics
    rmse = np.sqrt(mean_squared_error(y_test1, y_pred))
    mae = mean_absolute_error(y_test1, y_pred)
    r2 = r2_score(y_test1, y_pred)

    # Store the metrics
    results.append((test_size, rmse, mae, r2))

# Print the results
for test_size, rmse, mae, r2 in results:
    print(f"Test Size: {test_size:.2f} | RMSE: {rmse:.3f} | MAE: {mae:.3f} |
```

```
Test Size: 0.10 | RMSE: 2.057 | MAE: 1.421 | R^2: 0.911
Test Size: 0.20 | RMSE: 2.341 | MAE: 1.692 | R^2: 0.903
Test Size: 0.30 | RMSE: 2.816 | MAE: 1.960 | R^2: 0.864
Test Size: 0.40 | RMSE: 2.821 | MAE: 2.036 | R^2: 0.866
Test Size: 0.50 | RMSE: 3.071 | MAE: 2.201 | R^2: 0.850
```

Based on the results, we can observe the impact of different test sizes on the model performance:

Observations:

1. **Test Size: 0.10**
    - RMSE: 2.057
    - MAE: 1.421
    - R^2: 0.911
    - **Observation:** The model performs very well with a small test size, showing the lowest RMSE and MAE, and the highest R^2. However, this might not be a reliable estimate of model performance on unseen data due to the small test set.
2. **Test Size: 0.20**
    - RMSE: 2.341
    - MAE: 1.692
    - R^2: 0.903
    - **Observation:** The model still performs well, with slightly higher RMSE and MAE, and a slightly lower R^2 compared to the 10% test size. This test size provides a more reliable estimate of model performance.
3. **Test Size: 0.30**
    - RMSE: 2.816
    - MAE: 1.960
    - R^2: 0.864
    - **Observation:** The performance metrics indicate a further increase in RMSE and MAE, and a decrease in R^2. This test size provides a good balance between training and test data.
4. **Test Size: 0.40**
    - RMSE: 2.821
    - MAE: 2.036
    - R^2: 0.866
    - **Observation:** The performance metrics are similar to the 30% test size, indicating that the model's performance is relatively stable with this test size.
5. **Test Size: 0.50**

- RMSE: 3.071
- MAE: 2.201
- R^2: 0.850
- **Observation:** The model's performance metrics show the highest RMSE and MAE, and the lowest R^2. This indicates that the model has less data to train on, which might affect its ability to generalize.

Conclusion:

- **Smaller Test Sizes (10-20%):** These provide lower RMSE and MAE, and higher R^2, but might not be as reliable for estimating model performance on unseen data due to the smaller test set.
-
- **Moderate Test Sizes (30-40%):** These provide a good balance between training and test data, with relatively stable performance metrics. They are likely to provide more reliable estimates of model performance on unseen data.
-
- **Larger Test Sizes (50%):** These provide higher RMSE and MAE, and lower R^2, indicating that the model has less data to train on, which might affect its ability to generalize.

Recommendation:

- A test size of **20-30%** is generally recommended as it provides a good balance between having enough data for training and having a representative test set for reliable performance estimation. In this case, a test size of **30%** seems to provide a good balance with reasonable performance metrics.

You can further fine-tune the test size and other hyperparameters based on your specific dataset and model requirements to achieve the best performance.

From the dataset study, there are several additional insights and analyses that can be performed to gain a deeper understanding of the data and the models. Here are some suggestions:

1. Feature Importance Analysis:

Random Forest Feature Importance: We already looked at feature importance for the Random Forest model. This can help identify which features are most influential in predicting the target variable (mpg). Comparison Across Models: Compare feature importance across different models (e.g., Decision Tree, Random Forest) to see if certain features consistently show high importance.

3. Residual Analysis:

Residual Plots: Plot the residuals (difference between actual and predicted values) to check for patterns. Ideally, residuals should be randomly distributed, indicating a good fit. Heteroscedasticity: Check for heteroscedasticity (changing variance of residuals) which can indicate issues with the model.

4. Model Comparison:

Cross-Validation: Perform cross-validation to compare the performance of different models (e.g., Linear Regression, Decision Tree, Random Forest) more robustly. Hyperparameter Tuning: Use techniques like Grid Search or Random Search to tune hyperparameters and improve model performance.

5. Correlation Analysis:

Correlation Matrix: Analyze the correlation matrix to understand the relationships between different features. High correlation between features can indicate multicollinearity, which might affect model performance. Pair Plots: Use pair plots to visualize relationships between pairs of features and the target variable.

6. Distribution Analysis:

Histograms and Density Plots: Plot histograms and density plots for each feature to understand their distributions. This can help identify skewness, outliers, and the need for transformations. Box Plots: Use box plots to visualize the spread and identify outliers in the data.

7. Interaction Effects:

Interaction Terms: Explore interaction effects between features. Interaction terms can be added to the model to capture the combined effect of multiple features on the target variable.

8. Model Diagnostics:

VIF (Variance Inflation Factor): Calculate VIF to check for multicollinearity among features. High VIF values indicate multicollinearity, which can be addressed by removing or combining features. QQ Plots: Use QQ plots to check if the residuals follow a normal distribution, which is an assumption for certain models like Linear Regression.

9. Predictive Performance:

ROC Curve and AUC: For classification tasks, plot the ROC curve and calculate the AUC to evaluate model performance. Precision-Recall Curve: For imbalanced datasets, use precision-recall curves to evaluate model performance.

10. Sensitivity Analysis:

Sensitivity to Test Size: As we did, analyze how sensitive the model performance is to different test sizes. Sensitivity to Random State: Analyze how sensitive the model performance is to different random states in train-test splits.

11. Domain-Specific Insights:

Domain Knowledge: Use domain knowledge to interpret the results. For example, understanding why certain features are important for predicting mpg can provide valuable insights. Policy Implications: If applicable, analyze the policy implications of the findings. For example, understanding the impact of vehicle weight on fuel efficiency can inform regulations and standards. Example: Residual Analysis This code will:

Predict the target variable on the test set using the Random Forest model. Calculate the residuals (difference between actual and predicted values). Plot the residuals to check for patterns and distribution. By performing these additional analyses, you can gain a deeper understanding of the dataset, the relationships between features, and the performance and behavior of different models. This can help in making more informed decisions and improving model performance.

From the dataset study, there are several additional insights and analyses that can be performed to gain a deeper understanding of the data and the models. Here are some suggestions:

1. **Feature Importance Analysis:**

- **Random Forest Feature Importance:** We already looked at feature importance for the Random Forest model. This can help identify which features are most influential in predicting the target variable (`mpg`).
- **Comparison Across Models:** Compare feature importance across different models (e.g., Decision Tree, Random Forest) to see if certain features consistently show high importance.

2. **Residual Analysis:**

- **Residual Plots:** Plot the residuals (difference between actual and predicted values) to check for patterns. Ideally, residuals should be randomly distributed, indicating a good fit.
- **Heteroscedasticity:** Check for heteroscedasticity (changing variance of residuals) which can indicate issues with the model.

3. **Model Comparison:**

- **Cross-Validation:** Perform cross-validation to compare the performance of different models (e.g., Linear Regression, Decision Tree, Random Forest) more robustly.
- **Hyperparameter Tuning:** Use techniques like Grid Search or Random Search to tune hyperparameters and improve model performance.

4. **Correlation Analysis:**

- **Correlation Matrix:** Analyze the correlation matrix to understand the relationships between different features. High correlation between features can indicate multicollinearity, which might affect model performance.
- **Pair Plots:** Use pair plots to visualize relationships between pairs of features and the target variable.

5. **Distribution Analysis:**

- **Histograms and Density Plots:** Plot histograms and density plots for each feature to understand their distributions. This can help identify skewness, outliers, and the need for transformations.
- **Box Plots:** Use box plots to visualize the spread and identify outliers in the data.

6. **Interaction Effects:**

- **Interaction Terms:** Explore interaction effects between features. Interaction terms can be added to the model to capture the combined effect of multiple features on the target variable.

7. **Model Diagnostics:**

- **VIF (Variance Inflation Factor):** Calculate VIF to check for multicollinearity among features. High VIF values indicate multicollinearity, which can be addressed by removing or combining features.
- **QQ Plots:** Use QQ plots to check if the residuals follow a normal distribution, which is an assumption for certain models like Linear Regression.

8. **Predictive Performance:**

- **ROC Curve and AUC:** For classification tasks, plot the ROC curve and calculate the AUC to evaluate model performance.
- **Precision-Recall Curve:** For imbalanced datasets, use precision-recall curves to evaluate model performance.

9. **Sensitivity Analysis:**

- **Sensitivity to Test Size:** As we did, analyze how sensitive the model performance is to different test sizes.
- **Sensitivity to Random State:** Analyze how sensitive the model performance is to different random states in train-test splits.

10. Domain-Specific Insights:

- **Domain Knowledge:** Use domain knowledge to interpret the results. For example, understanding why certain features are important for predicting `mpg` can provide valuable insights.
- **Policy Implications:** If applicable, analyze the policy implications of the findings. For example, understanding the impact of vehicle weight on fuel efficiency can inform regulations and standards.

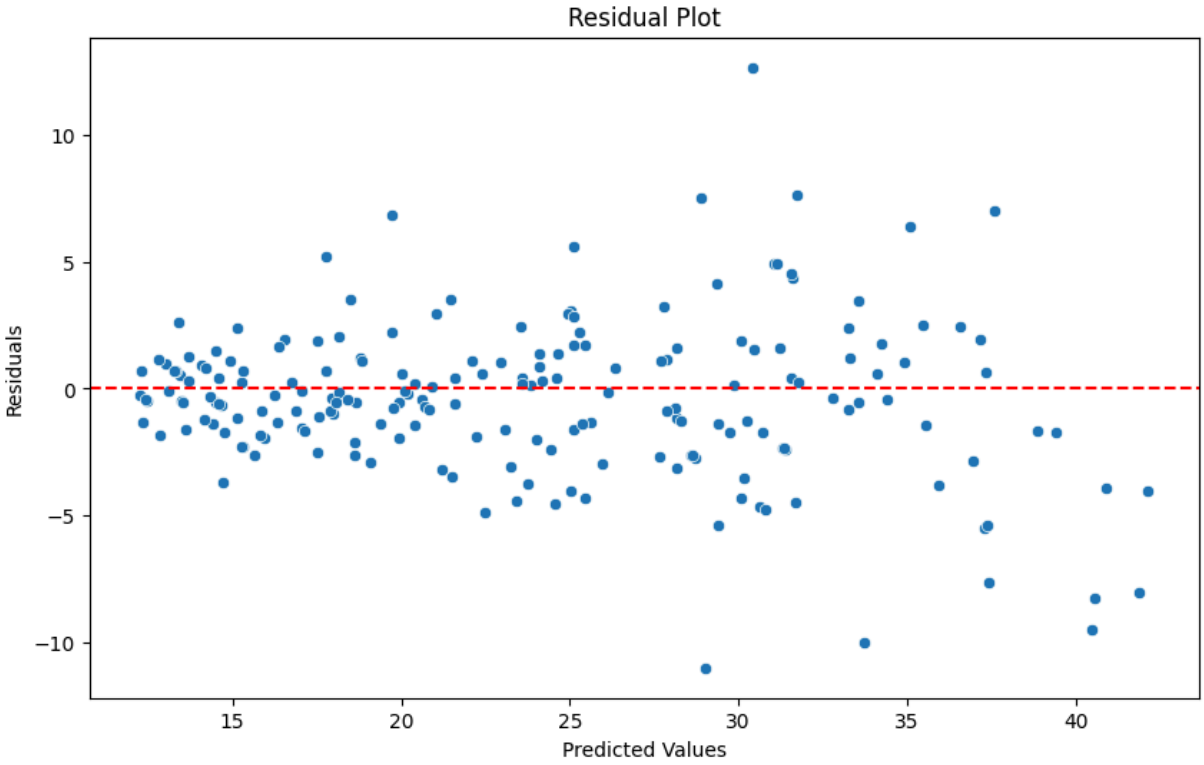
This code will:

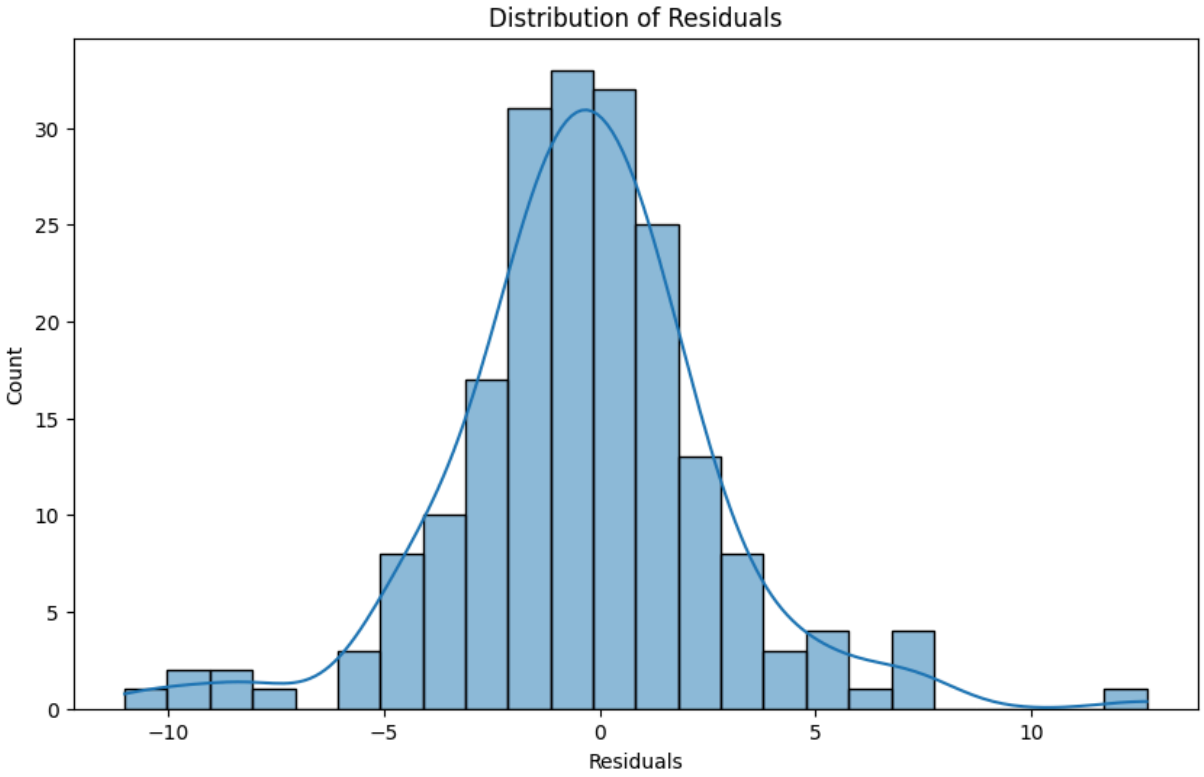
1. Predict the target variable on the test set using the Random Forest model.
2. Calculate the residuals (difference between actual and predicted values).
3. Plot the residuals to check for patterns and distribution.

By performing these additional analyses, you can gain a deeper understanding of the dataset, the relationships between features, and the performance and behavior of different models. This can help in making more informed decisions and improving model performance.

In [77]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Predict on the test set
y_pred = rf.predict(X_test1)

# Calculate residuals
residuals = y_test1 - y_pred

# Plot residuals
plt.figure(figsize=(10, 6))
sns.scatterplot(x=y_pred, y=residuals)
plt.axhline(0, color='red', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residual Plot')
plt.show()

# Plot distribution of residuals
plt.figure(figsize=(10, 6))
sns.histplot(residuals, kde=True)
plt.xlabel('Residuals')
```

```
plt.title('Distribution of Residuals')
plt.show()
```





Indeed, the field of data science is vast and encompasses a wide range of topics and techniques. It can be overwhelming, especially for those who are new to the field or even for experienced practitioners who are venturing into new areas. However, it's important

to remember that data science is a collaborative and iterative process. Here are some strategies to manage the complexity and continue growing your expertise:

1. **Focus on Fundamentals:**

- **Statistics and Probability:** A strong foundation in statistics and probability is crucial for understanding data distributions, hypothesis testing, and model evaluation.
- **Linear Algebra and Calculus:** These mathematical concepts are the backbone of many machine learning algorithms.
- **Programming Skills:** Proficiency in programming languages like Python or R is essential for implementing data science workflows.

2. **Learn Incrementally:**

- **Start Simple:** Begin with simpler models and techniques (e.g., linear regression, decision trees) before moving on to more complex ones (e.g., ensemble methods, deep learning).
- **Build on Knowledge:** Gradually expand your knowledge by learning new techniques and tools as you become comfortable with the basics.

3. **Practical Experience:**

- **Projects:** Work on real-world projects to apply what you've learned. This helps in understanding the practical challenges and nuances of data science.
- **Competitions:** Participate in data science competitions (e.g., Kaggle) to gain experience and learn from others.

4. **Collaborate and Network:**

- **Peer Learning:** Collaborate with peers to share knowledge and learn from each other.
- **Mentorship:** Seek mentorship from experienced data scientists who can provide guidance and insights.
- **Community:** Engage with the data science community through forums, meetups, and conferences.

5. **Continuous Learning:**

- **Courses and Tutorials:** Take online courses and follow tutorials to keep up with new developments and techniques.
- **Books and Research Papers:** Read books and research papers to deepen your understanding of specific topics.

- **Blogs and Articles:** Follow data science blogs and articles to stay updated with industry trends and best practices.

6. Tools and Libraries:

- **Familiarize with Tools:** Learn to use popular data science tools and libraries (e.g., Pandas, Scikit-Learn, TensorFlow, PyTorch) to streamline your workflow.
- **Automated Tools:** Explore automated machine learning (AutoML) tools that can help with model selection and hyperparameter tuning.

7. Domain Knowledge:

- **Understand the Domain:** Gain knowledge about the specific domain you are working in (e.g., finance, healthcare) to make more informed decisions and interpretations.
- **Interdisciplinary Approach:** Data science often intersects with other fields. An interdisciplinary approach can provide valuable insights and innovative solutions.

This example demonstrates an incremental learning approach, starting with a simple linear regression model and then moving on to a more complex random forest model. By gradually building on your knowledge and experience, you can manage the complexity of data science and continue to grow your expertise.

In [78]:
```
#  Example: Incremental Learning Approach

# Start with a simple linear regression model
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

# Load and preprocess the data (assuming data is already loaded and preproce
X = data.drop(columns=['mpg', 'origin_europe'])
y = data['mpg']

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, rar

# Initialize and fit the linear regression model
linear_model = LinearRegression()
linear_model.fit(X_train, y_train)

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

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Linear Regression - MSE: {mse:.3f}, R^2: {r2:.3f}")
```

```
# Once comfortable, move on to more complex models like Random Forest
from sklearn.ensemble import RandomForestRegressor

# Initialize and fit the random forest model
rf_model = RandomForestRegressor(n_estimators=100, random_state=1)
rf_model.fit(X_train, y_train)

# Predict on the test set
y_pred_rf = rf_model.predict(X_test)

# Evaluate the model
mse_rf = mean_squared_error(y_test, y_pred_rf)
r2_rf = r2_score(y_test, y_pred_rf)

print(f"Random Forest - MSE: {mse_rf:.3f}, R^2: {r2_rf:.3f}")
```

```
Linear Regression - MSE: 9.161, R^2: 0.843
Random Forest - MSE: 7.927, R^2: 0.864
```

In [79]:
```
data
```

Out[79]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin_
0	18.0	8	307.0	130.0	3504	12.0	70	
1	15.0	8	350.0	165.0	3693	11.5	70	
2	18.0	8	318.0	150.0	3436	11.0	70	
3	16.0	8	304.0	150.0	3433	12.0	70	
4	17.0	8	302.0	140.0	3449	10.5	70	
...	...	...	...	...	...	...	...	
393	27.0	4	140.0	86.0	2790	15.6	82	
394	44.0	4	97.0	52.0	2130	24.6	82	
395	32.0	4	135.0	84.0	2295	11.6	82	
396	28.0	4	120.0	79.0	2625	18.6	82	
397	31.0	4	119.0	82.0	2720	19.4	82	

398 rows × 11 columns

While the implementation of machine learning models in Python can be straightforward thanks to powerful libraries like Scikit-Learn, TensorFlow, and PyTorch, the underlying theory and concepts are indeed complex. Understanding these theories is crucial for making informed decisions, interpreting results, and improving models.

Key Theoretical Concepts in Data Science and Machine Learning Statistics and Probability:

Descriptive Statistics: Mean, median, mode, variance, standard deviation, etc. Inferential Statistics: Hypothesis testing, confidence intervals, p-values, etc. Probability Distributions: Normal distribution, binomial distribution, Poisson distribution, etc. Linear Algebra:

Vectors and Matrices: Operations, transformations, eigenvalues, eigenvectors. Matrix Decompositions: Singular value decomposition (SVD), principal component analysis (PCA). Calculus:

Differentiation: Derivatives, gradients, optimization. Integration: Area under curves, cumulative distributions. Optimization:

Gradient Descent: Learning rate, convergence, stochastic gradient descent (SGD). Convex Optimization: Convex functions, local and global minima. Machine Learning Algorithms:

Supervised Learning: Linear regression, logistic regression, decision trees, random forests, support vector machines (SVM), neural networks. Unsupervised Learning: K-means clustering, hierarchical clustering, PCA, t-SNE. Reinforcement Learning: Markov decision processes, Q-learning, policy gradients. Model Evaluation:

Metrics: Accuracy, precision, recall, F1-score, ROC-AUC, mean squared error (MSE), mean absolute error (MAE). Cross-Validation: K-fold cross-validation, leave-one-out cross-validation. Feature Engineering:

Feature Selection: Removing irrelevant or redundant features. Feature Extraction: Creating new features from existing ones. Normalization and Scaling: Standardizing data to improve model performance. Regularization:

L1 and L2 Regularization: Preventing overfitting by adding penalty terms to the loss function. Dropout: Regularization technique for neural networks. Example: Understanding Linear Regression Let's take linear regression as an example. While the implementation is simple, the theory involves understanding several concepts:

Model Equation:

The linear regression model is defined as $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n + \epsilon$, where β are the coefficients and ϵ is the error term. Ordinary Least Squares (OLS):

The goal is to minimize the sum of squared residuals (differences between observed and predicted values). This involves solving for the coefficients β that minimize the cost function $J(\beta) = \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$. Assumptions:

Linearity: The relationship between the features and the target is linear. Independence: Observations are independent of each other. Homoscedasticity: Constant variance of

residuals. Normality: Residuals are normally distributed. Interpretation:

Coefficients: Each (\beta) represents the change in the target variable for a one-unit change in the corresponding feature, holding all other features constant. R-squared: Proportion of variance in the target variable explained by the features. Example: Linear Regression Implementation Conclusion While the implementation of machine learning models in Python can be straightforward, understanding the underlying theory is essential for:

Making informed decisions about model selection and evaluation. Interpreting the results and understanding the limitations of the models. Improving model performance through techniques like feature engineering, regularization, and hyperparameter tuning. Continuous learning and practice, along with a solid understanding of the theoretical concepts, are key to becoming proficient in data science and machine learning.

In [80]:
```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

# Load and preprocess the data (assuming data is already loaded and preproce
X = data.drop(columns=['mpg', 'origin_europe'])
y = data['mpg']

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, rar

# Initialize and fit the linear regression model
linear_model = LinearRegression()
linear_model.fit(X_train, y_train)

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

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Linear Regression - MSE: {mse:.3f}, R^2: {r2:.3f}")
```

```
Linear Regression - MSE: 9.161, R^2: 0.843
```

The result `Linear Regression - MSE: 9.161, R^2: 0.843` provides insights into the performance of the linear regression model. Lets break down the deeper meaning of these metrics:

Mean Squared Error (MSE)

- **Definition:** MSE is the average of the squared differences between the actual and

predicted values. It is calculated as: [\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2] where (y_i) are the actual values, (\hat{y}_i) are the predicted values, and (n) is the number of observations.

- `**Interpretation:**`
  - A lower MSE indicates that the model's predictions are closer to the actual values.
  - In this case, an MSE of 9.161 means that, on average, the squared difference between the actual and predicted values is 9.161. This value is in the units of the target variable squared.
  - MSE is sensitive to outliers because it squares the errors, giving more weight to larger errors.

R-squared (R^2)

- **Definition:** R^2, also known as the coefficient of determination, measures the proportion of the variance in the dependent variable that is predictable from the independent variables. It is calculated as: [R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}*i)^2}{|sum*{i=1}^{n} (y_i - \bar{y})^2}] where (\bar{y}) is the mean of the actual values.

- **Interpretation:**

  - R^2 ranges from 0 to 1, where 0 indicates that the model explains none of the variance in the target variable, and 1 indicates that the model explains all the variance.
  - An R^2 of 0.843 means that 84.3% of the variance in the target variable (`mpg`) is explained by the independent variables in the model.
  - A higher R^2 indicates a better fit of the model to the data.

Deeper Meaning and Insights

1. **Model Fit:**
   - An R^2 of 0.843 suggests that the linear regression model provides a good fit to the data,

explaining a significant portion of the variance in the target variable. However, it also indicates that there is still 15.7% of the variance that is not explained by the model, which could be due to factors not included in the model or inherent randomness.

2. **Error Magnitude:**
   - An MSE of 9.161 indicates the average squared error between the actual and predicted values.

While MSE provides a sense of the error magnitude, it is not as interpretable as R^2 in terms of explaining the proportion of variance.

3. **Model Limitations:**
    - Despite a high R^2, the model may still have limitations. For example, it might not capture non-linear relationships or interactions between variables. Additionally, the presence of outliers or multicollinearity among the independent variables can affect the model's performance.
4. **Practical Implications:**
    - In practical terms, the results suggest that the linear regression model is useful for predicting `mpg` based on the given features. However, there may be room for improvement by exploring more complex models, feature engineering, or addressing potential issues like multicollinearity.

Next Steps To gain further insights and potentially improve the model, consider the following steps:

1. **Residual Analysis:**
    - Analyze the residuals to check for patterns, heteroscedasticity, and normality. Residual plots can help identify issues with the model.
2. **Feature Engineering:**
    - Explore creating new features or transforming existing ones to capture non-linear relationships or interactions.
3. **Model Comparison:**
    - Compare the linear regression model with other models (e.g., polynomial regression, decision trees, random forests) to see if they provide better performance.
4. **Cross-Validation:**
    - Use cross-validation to assess the model's performance more robustly and ensure that the results are not due to a particular train-test split.
5. **Hyperparameter Tuning:**
    - For more complex models, perform hyperparameter tuning to optimize the model's performance.

By performing these additional analyses and steps, you can gain a deeper understanding of the model's performance and identify areas for improvement.

In [ ]: