regression_gjrich

April 12, 2025

1 Final Project: Regression Analysis

Submission: GitHub Repository with Jupyter Notebook and Peer Review

---

1.1 Overview

Businesses and organizations often need to understand the relationships between different factors to make better decisions. For example, a company may want to predict the fuel efficiency of a car based on its weight and engine size or estimate home prices based on square footage and location. Regression analysis helps identify and quantify these relationships between numerical features, providing insights that can be used for forecasting and decision-making.

This project demonstrates your ability to apply regression modeling techniques to a real-world dataset. You will: - Load and explore a dataset. - Choose and justify features for predicting a target variable. - Train a regression model and evaluate performance. - Compare multiple regression approaches. - Document your work in a structured Jupyter Notebook. - Conduct a peer review of a classmate's project.

1.2 Dataset Options

Select one dataset from the list below. If you get good results, you can try the process on a suitable dataset of your own. Suitable datasets contain **numerical features** and a **numerical target variable** for regression.

1. Auto MPG Dataset (Predict fuel efficiency based on engine specs and weight)
   - UCI Auto MPG Dataset

1.3 Section 1. Import and Inspect the Data

1.3.1 1.0 Import necessary libraries

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
```

```python
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from scipy.stats import ks_2samp, wasserstein_distance, energy_distance

import math
import time

#using this variable allows us to
state_setter=432
```

1.3.2 1.1 Load the dataset and display the first 10 rows.

I used utils/convert_to_csv.py to convert our original auto-mpg.data file to a csv for easy consumption by Pandas. Pandas also had trouble with the data set due to some corrupted values, so convert_to_csv converts any mismatched values in auto-mpg.data to blanks when generating auto-mpg.csv.

[2]:
```python
# Load the dataset
df = pd.read_csv('data/auto-mpg.csv')

# Display the first 10 rows
print("First 10 rows of the dataset:")
display(df.head(10))
```

```
First 10 rows of the dataset:

    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0  3504.0          12.0
1  15.0          8         350.0       165.0  3693.0          11.5
2  18.0          8         318.0       150.0  3436.0          11.0
3  16.0          8         304.0       150.0  3433.0          12.0
4  17.0          8         302.0       140.0  3449.0          10.5
5  15.0          8         429.0       198.0  4341.0          10.0
6  14.0          8         454.0       220.0  4354.0           9.0
7  14.0          8         440.0       215.0  4312.0           8.5
8  14.0          8         455.0       225.0  4425.0          10.0
9  15.0          8         390.0       190.0  3850.0           8.5

   model_year  origin                   car_name
0          70       1  chevrolet chevelle malibu
1          70       1          buick skylark 320
2          70       1         plymouth satellite
3          70       1              amc rebel sst
4          70       1                ford torino
5          70       1           ford galaxie 500
6          70       1           chevrolet impala
7          70       1          plymouth fury iii
```

```
8          70       1           pontiac catalina
9          70       1         amc ambassador dpl
```

1.3.3 1.2 Check for missing values and display summary statistics.

[3]:
```python
# Check for missing values
print("\nMissing values in each column:")
print(df.isnull().sum())
```

```
Missing values in each column:
mpg             0
cylinders       0
displacement    0
horsepower      6
weight          0
acceleration    0
model_year      0
origin          0
car_name        0
dtype: int64
```

Horsepower is missing 6 values. We will address this in section 2.

[4]:
```python
# Get summary statistics
print("\nSummary statistics:")
print(df.describe(include='all').T)
```

```
Summary statistics:
              count unique         top freq         mean         std     min  \
mpg           398.0    NaN         NaN  NaN    23.514573    7.815984     9.0
cylinders     398.0    NaN         NaN  NaN     5.454774    1.701004     3.0
displacement  398.0    NaN         NaN  NaN   193.425879  104.269838    68.0
horsepower    392.0    NaN         NaN  NaN   104.469388    38.49116    46.0
weight        398.0    NaN         NaN  NaN  2970.424623  846.841774  1613.0
acceleration  398.0    NaN         NaN  NaN     15.56809    2.757689     8.0
model_year    398.0    NaN         NaN  NaN     76.01005    3.697627    70.0
origin        398.0    NaN         NaN  NaN     1.572864    0.802055     1.0
car_name        398    305  ford pinto    6          NaN         NaN     NaN

                  25%     50%     75%     max
mpg              17.5    23.0    29.0    46.6
cylinders         4.0     4.0     8.0     8.0
displacement   104.25   148.5   262.0   455.0
horsepower       75.0    93.5   126.0   230.0
weight        2223.75  2803.5  3608.0  5140.0
acceleration   13.825    15.5  17.175    24.8
model_year       73.0    76.0    79.0    82.0
```

```
origin              1.0      1.0      2.0      3.0
car_name            NaN      NaN      NaN      NaN
```

Some summary statistics for the info. note that unique, top, and freq are generated for categorical features whereas mean, std, and min are generated for numerical features.

[5]:
```python
# Check data types
print("\nData types:")
print(df.dtypes)

# Check unique values in categorical columns
print("\nUnique values in 'origin' column:")
print(df['origin'].unique())
```

```
Data types:
mpg             float64
cylinders         int64
displacement    float64
horsepower      float64
weight          float64
acceleration    float64
model_year        int64
origin            int64
car_name         object
dtype: object

Unique values in 'origin' column:
[1 3 2]
```

We have mostly numerical columns with a couple string columns.

1.3.4 Reflection 1: What do you notice about the dataset? Are there any data issues?

The 'origin' column contains only integers (1, 2, and 3), but likely represents categorical information about manufacturing locations that should be properly encoded. The dataset spans model years from the 1970s to early 1980s based on the 'model_year' column, making this a historical dataset that might not reflect current automotive technology. These issues will need to be resolved through appropriate data cleaning and transformation steps before proceeding with modeling. The engine is misfiring on this dataset, but with some fine-tuning, we'll have it purring in no time.

1.4 Section 2. Data Exploration and Preparation

1.4.1 2.1 Handle missing values and clean data

First, let's examine our missing values more closely and create a plan to handle them. We already identified that the 'horsepower' column has 6 missing values, and they appear to be represented as '?' characters. Let's double check that all got taken care of with our earlier work.

[6]:

```python
# Function to check for mismatched data types in the dataframe
def check_data_type_mismatches(df):
    print("Checking for data type mismatches in each column...")

    # Dictionary to store results
    mismatches = {}

    # Check each column
    for column in df.columns:
        # Get the data type of the column
        dtype = df[column].dtype

        # Check for mismatches based on the data type
        if dtype == 'int64' or dtype == 'float64':
            # For numeric columns, check for non-numeric values
            non_numeric_count = 0
            non_numeric_indices = []

            for i, value in enumerate(df[column]):
                # Try to convert to float to see if it's numeric
                try:
                    float(value)
                except (ValueError, TypeError):
                    # If conversion fails, it's not numeric
                    non_numeric_count += 1
                    if non_numeric_count <= 5:  # Limit to first 5 examples
                        non_numeric_indices.append(i)

            if non_numeric_count > 0:
                mismatches[column] = {
                    'expected_type': dtype,
                    'mismatch_count': non_numeric_count,
                    'example_indices': non_numeric_indices
                }

    # Print results
    if mismatches:
        print("\nMismatched data types found:")
        for column, info in mismatches.items():
            print(f"\nColumn: {column}")
            print(f"Expected type: {info['expected_type']}")
            print(f"Mismatches found: {info['mismatch_count']}")

            # Print examples
            print("Examples of mismatched values:")
            for idx in info['example_indices']:
                print(f"  Row {idx}: '{df.loc[idx, column]}'")
```

```python
    else:
        print("\nNo data type mismatches found!")

    return mismatches

# Run the function on our dataframe
mismatches = check_data_type_mismatches(df)
```

```
Checking for data type mismatches in each column…

No data type mismatches found!
```

Looks good to go.

2.1.1 Impute or drop missing values Now that we've properly identified the missing values, we'll impute them using the median value of the horsepower column. This is a reasonable approach for this small number (6 out of ~400) of missing values in a numerical feature.

[7]:
```python
# Create an imputer for the horsepower column
horsepower_imputer = SimpleImputer(strategy='median')

# Fit the imputer on the horsepower data and transform it
df['horsepower'] = horsepower_imputer.fit_transform(df[['horsepower']])

# Verify that missing values have been imputed
print("Missing values after imputation:")
print(df.isnull().sum())
```

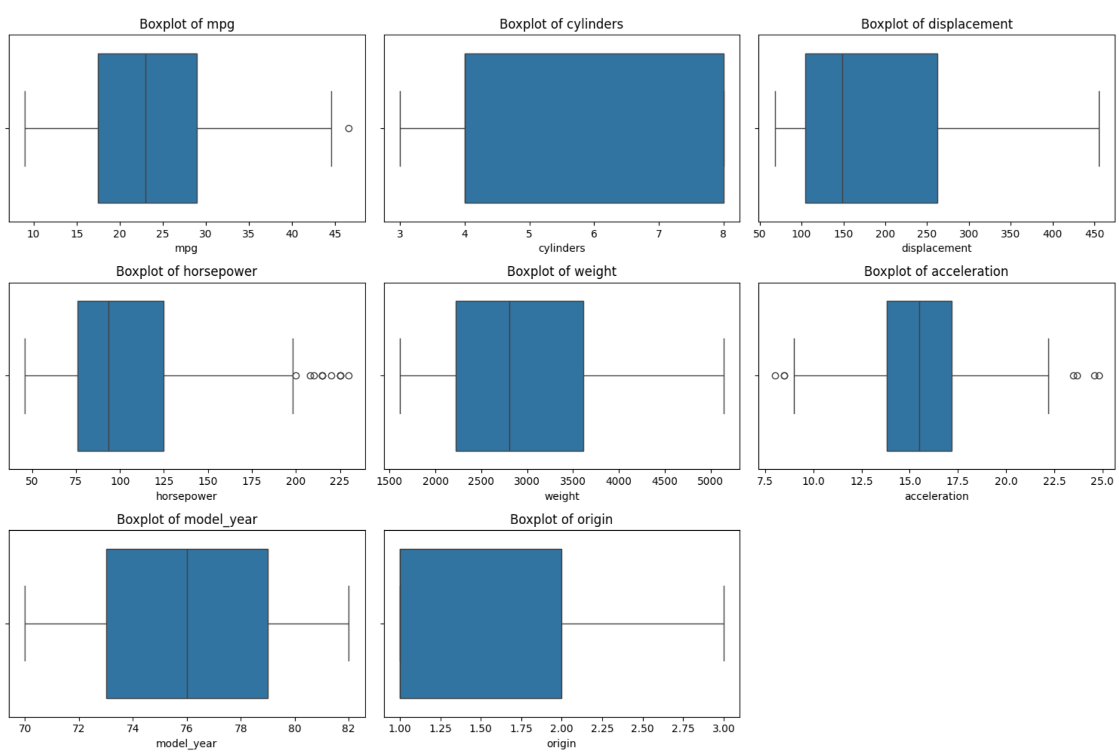
```
Missing values after imputation:
mpg             0
cylinders       0
displacement    0
horsepower      0
weight          0
acceleration    0
model_year      0
origin          0
car_name        0
dtype: int64
```

Worked well!

2.1.2 Remove or transform outliers Let's identify potential outliers in our numerical columns using boxplots. This will help us determine if any data points are significantly outside the normal range and might need to be addressed.

[8]:
```python
# Create boxplots for all numerical columns to identify outliers
plt.figure(figsize=(15, 10))
```

```python
# Select only numerical columns
numerical_cols = df.select_dtypes(include=['int64', 'float64']).columns

# Create boxplots for each numerical column
for i, column in enumerate(numerical_cols):
    plt.subplot(3, 3, i+1)
    sns.boxplot(x=df[column])
    plt.title(f'Boxplot of {column}')
    plt.tight_layout()

plt.show()

# Calculate the IQR and identify outliers for each numerical column
for column in numerical_cols:
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1

    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    outliers = ((df[column] < lower_bound) | (df[column] > upper_bound)).sum()

    if outliers > 0:
        print(f"Column '{column}' has {outliers} outliers")
```



```
Column 'mpg' has 1 outliers
Column 'horsepower' has 11 outliers
Column 'acceleration' has 7 outliers
```

2.1.3 Convert categorical data to numerical format using encoding

The 'origin' column is currently represented as integers (1, 2, 3) but it's actually a categorical feature representing the car's manufacturing region.

By examining a few example rows, we can see that 1 is america, 2 is Europe, and 3 is Asia. Here are some example rows evidencing that:

mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
18.0	8	307.0	130.0	3504.0	12.0	70	1	"chevrolet chevelle malibu"
15.0	8	350.0	165.0	3693.0	11.5	70	1	"buick skylark 320"
18.0	8	318.0	150.0	3436.0	11.0	70	1	"plymouth satel-lite"
26.0	4	97.00	46.00	1835.0	20.5	70	2	"volkswagen 1131 deluxe sedan"
25.0	4	110.0	87.00	2672.0	17.5	70	2	"peugeot 504"
24.0	4	107.0	90.00	2430.0	14.5	70	2	"audi 100 ls"
25.0	4	113.0	95.00	2228.0	14.0	71	3	"toyota corona"
27.0	4	97.00	88.00	2130.0	14.5	70	3	"datsun pl510"
35.0	4	72.00	69.00	1613.0	18.0	71	3	"datsun 1200"

Let's encode it properly using one-hot encoding.

[9]:
```python
# First, let's understand what the origin values represent
print("Value counts for 'origin':")
print(df['origin'].value_counts())

# Convert 'origin' to a more meaningful categorical representation
origin_mapping = {1: 'america', 2: 'europe', 3: 'asia'}
df['origin_name'] = df['origin'].map(origin_mapping)
```

```python
# Apply one-hot encoding to the 'origin' column
origin_encoded = pd.get_dummies(df['origin_name'], prefix='origin')

# Join the encoded columns to the original dataframe
df = pd.concat([df, origin_encoded], axis=1)

# Drop the original 'origin' and 'origin_name' columns
df = df.drop(['origin', 'origin_name'], axis=1)

# Display the first few rows to confirm the encoding
print("\nFirst 5 rows after encoding 'origin':")
display(df.head())
```

```
Value counts for 'origin':
origin
1    249
3     79
2     70
Name: count, dtype: int64

First 5 rows after encoding 'origin':
```

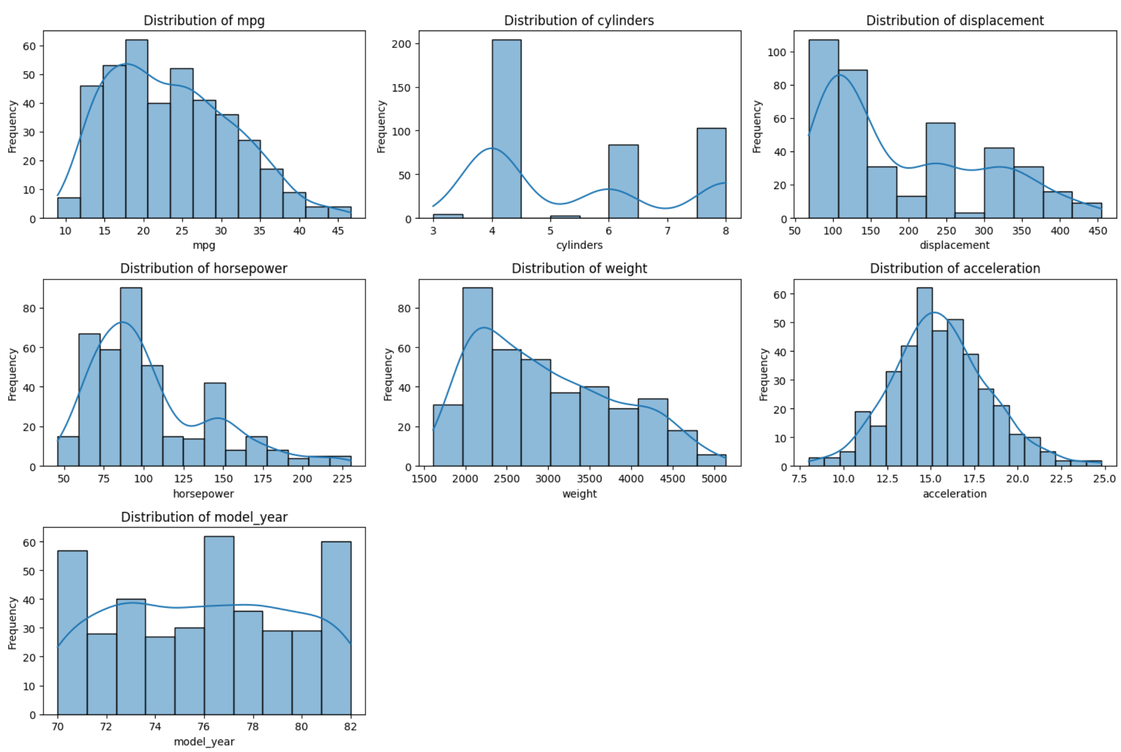
```
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0  3504.0          12.0
1  15.0          8         350.0       165.0  3693.0          11.5
2  18.0          8         318.0       150.0  3436.0          11.0
3  16.0          8         304.0       150.0  3433.0          12.0
4  17.0          8         302.0       140.0  3449.0          10.5

   model_year                   car_name  origin_america  origin_asia  \
0          70  chevrolet chevelle malibu            True        False
1          70          buick skylark 320            True        False
2          70         plymouth satellite            True        False
3          70              amc rebel sst            True        False
4          70                ford torino            True        False

   origin_europe
0          False
1          False
2          False
3          False
4          False
```

1.4.2 2.2 Explore data patterns and distributions

Now let's visualize our dataset's key features to better understand their distributions and relationships, starting with histograms of our numerical features.

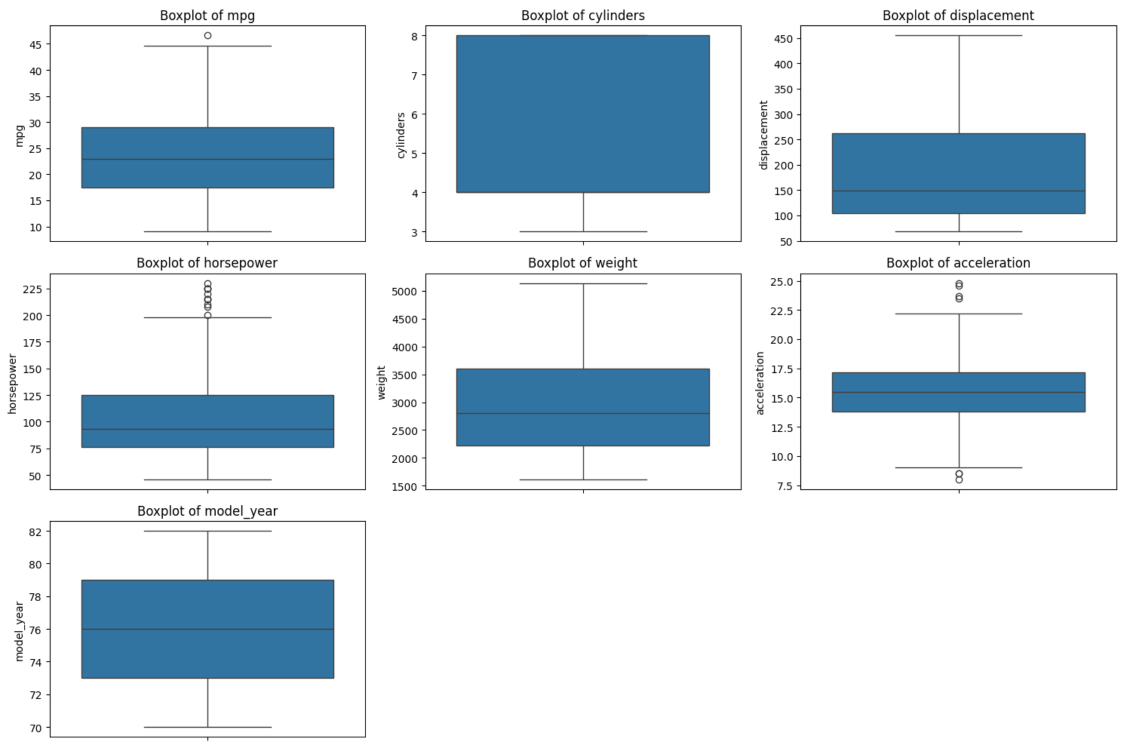
2.2.1 Create histograms, boxplots, and count plots for categorical variables (as applicable). Let's create visualizations to understand the distributions of our features. These plots will help us identify patterns and relationships in the data that might influence our modeling approach.

[10]:
```python
# Create histograms for numerical features
plt.figure(figsize=(15, 10))
numerical_features = ['mpg', 'cylinders', 'displacement', 'horsepower',
 ↪'weight', 'acceleration', 'model_year']

for i, feature in enumerate(numerical_features):
    plt.subplot(3, 3, i+1)
    sns.histplot(df[feature], kde=True)
    plt.title(f'Distribution of {feature}')
    plt.xlabel(feature)
    plt.ylabel('Frequency')

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



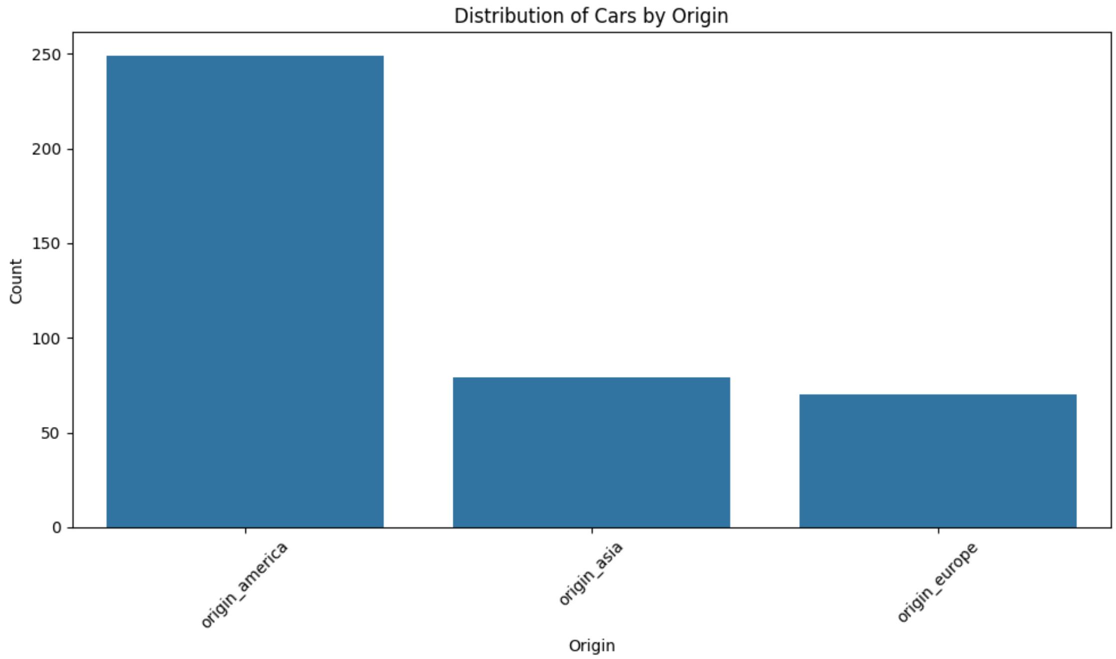
[11]:
```python
# Create boxplots for numerical features
plt.figure(figsize=(15, 10))
for i, feature in enumerate(numerical_features):
    plt.subplot(3, 3, i+1)
```

```
    sns.boxplot(y=df[feature])
    plt.title(f'Boxplot of {feature}')
    plt.ylabel(feature)

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



[12]:

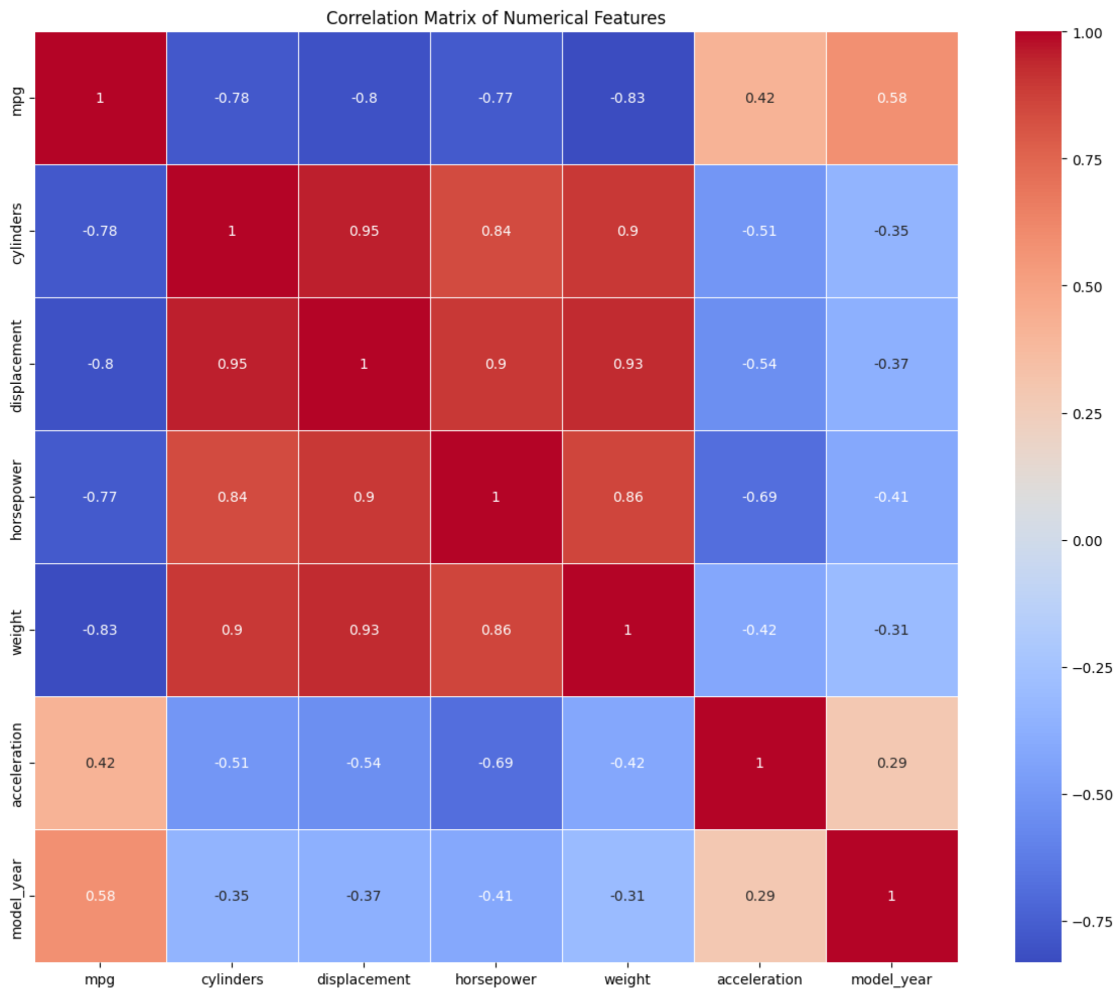
```
# Count plots for categorical variables (the encoded origin columns)
plt.figure(figsize=(10, 6))
origin_counts = df[['origin_america', 'origin_asia', 'origin_europe']].sum()
sns.barplot(x=origin_counts.index, y=origin_counts.values)
plt.title('Distribution of Cars by Origin')
plt.xlabel('Origin')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

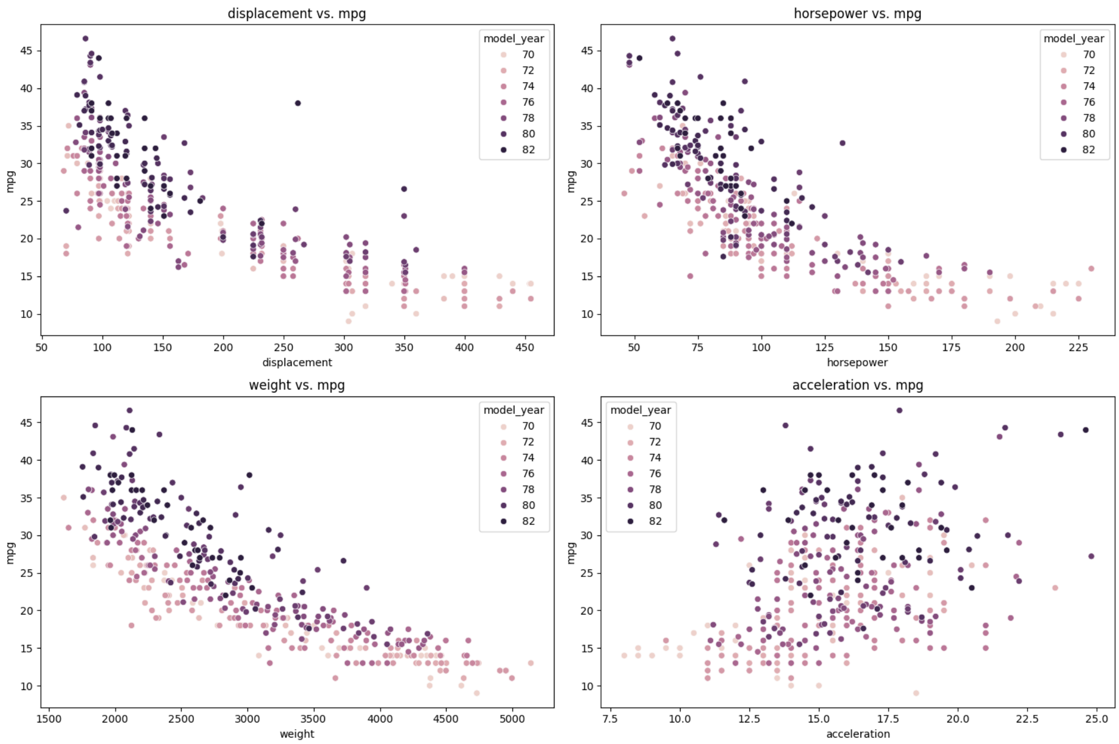


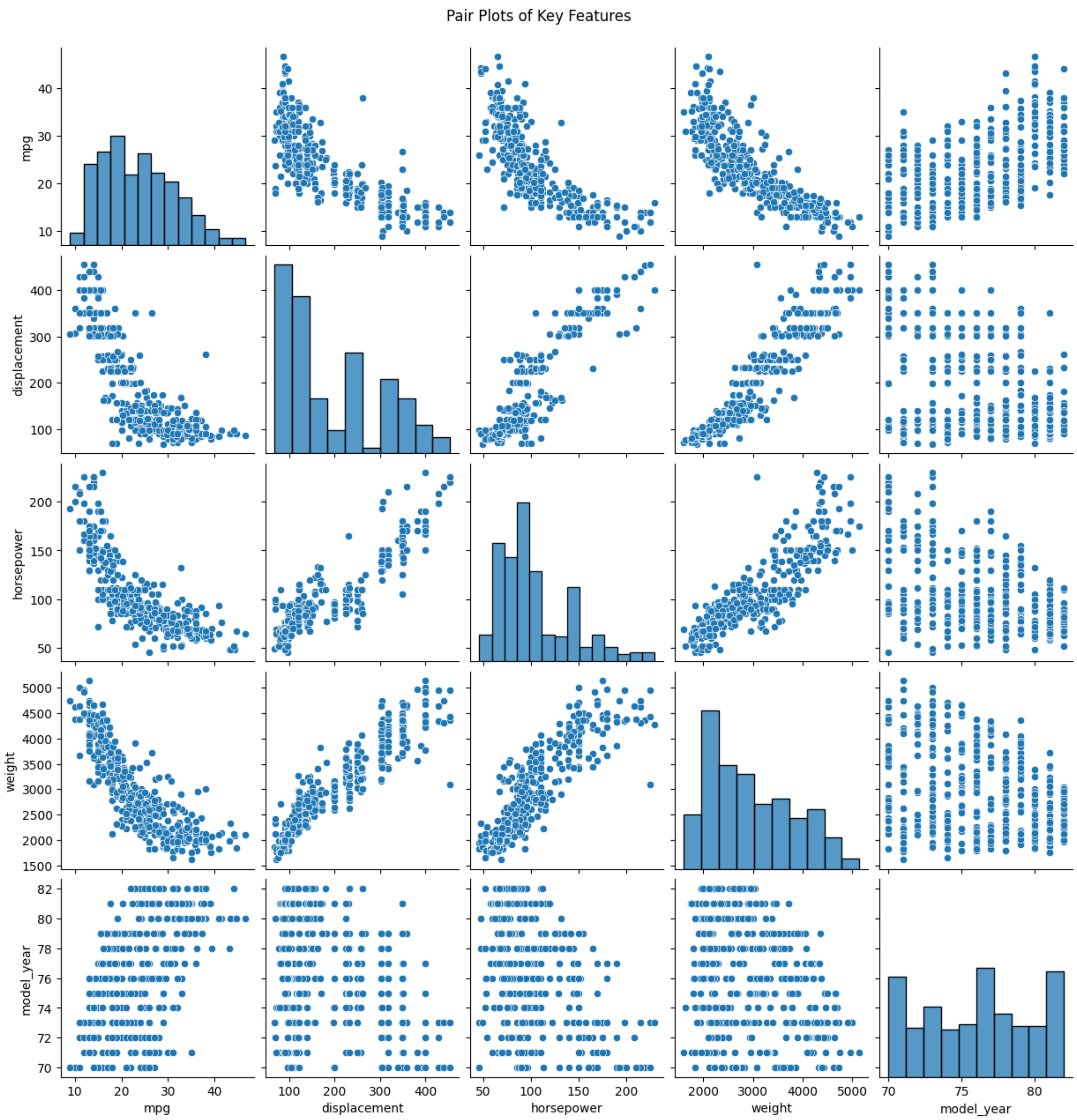
2.2.2 Identify patterns, outliers, and anomalies in feature distributions. Now, let's examine relationships between features and identify potential outliers. Correlation analysis and scatter plots will help us understand how features relate to our target variable (mpg) and to each other.

[13]:
```
# Correlation matrix to identify relationships between numerical features
plt.figure(figsize=(12, 10))
correlation_matrix = df[numerical_features].corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Matrix of Numerical Features')
plt.tight_layout()
plt.show()

# Scatter plots of important features vs. target variable (mpg)
plt.figure(figsize=(15, 10))
features_to_plot = ['displacement', 'horsepower', 'weight', 'acceleration']

for i, feature in enumerate(features_to_plot):
    plt.subplot(2, 2, i+1)
    sns.scatterplot(x=df[feature], y=df['mpg'], hue=df['model_year'])
    plt.title(f'{feature} vs. mpg')
    plt.xlabel(feature)
    plt.ylabel('mpg')

plt.tight_layout()
```

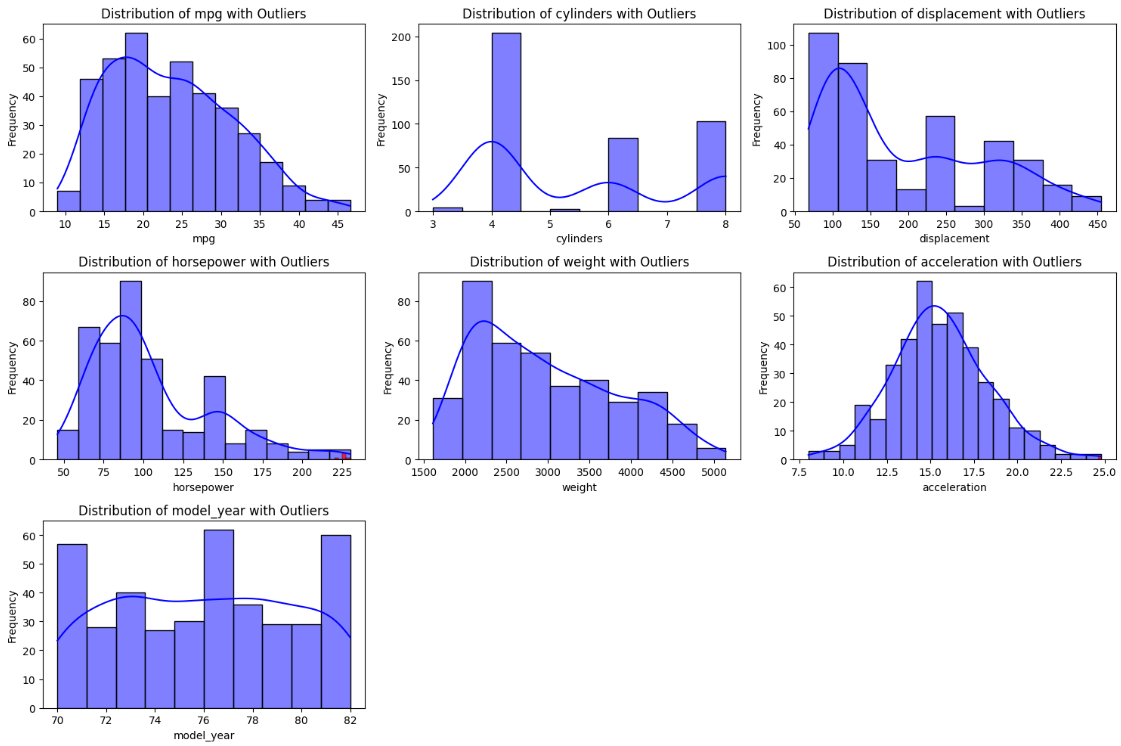
```python
plt.show()

# Pair plots for key features to identify patterns and outliers
sns.pairplot(df[['mpg', 'displacement', 'horsepower', 'weight', 'model_year']])
plt.suptitle('Pair Plots of Key Features', y=1.02)
plt.show()

# Calculate and show potential outliers using z-score method
from scipy import stats

plt.figure(figsize=(15, 10))
for i, feature in enumerate(numerical_features):
    plt.subplot(3, 3, i+1)
    z_scores = np.abs(stats.zscore(df[feature]))
    outliers = (z_scores > 3)

    # Plot histograms with outliers highlighted
    sns.histplot(df[feature], kde=True, color='blue', alpha=0.5)
    if outliers.any():
        sns.histplot(df[feature][outliers], color='red', alpha=0.7)

    plt.title(f'Distribution of {feature} with Outliers')
    plt.xlabel(feature)
    plt.ylabel('Frequency')

    # Print number of outliers
    outlier_count = outliers.sum()
    if outlier_count > 0:
        print(f"{feature} has {outlier_count} outliers (z-score > 3)")

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


Correlation Matrix of Numerical Features
mpg
cylinders
displacement
horsepower
weight
acceleration
model_year
1
-0.78
-0.8
-0.77
-0.83
0.42
0.58
0.95
0.84
0.9
-0.51
-0.35
0.93
-0.54
-0.37
0.86
-0.69
-0.41
-0.42
-0.31
0.29
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75


displacement vs. mpg
horsepower vs. mpg
weight vs. mpg
acceleration vs. mpg
model_year
70
72
74
76
78
80
82
mpg
displacement
horsepower
weight
acceleration



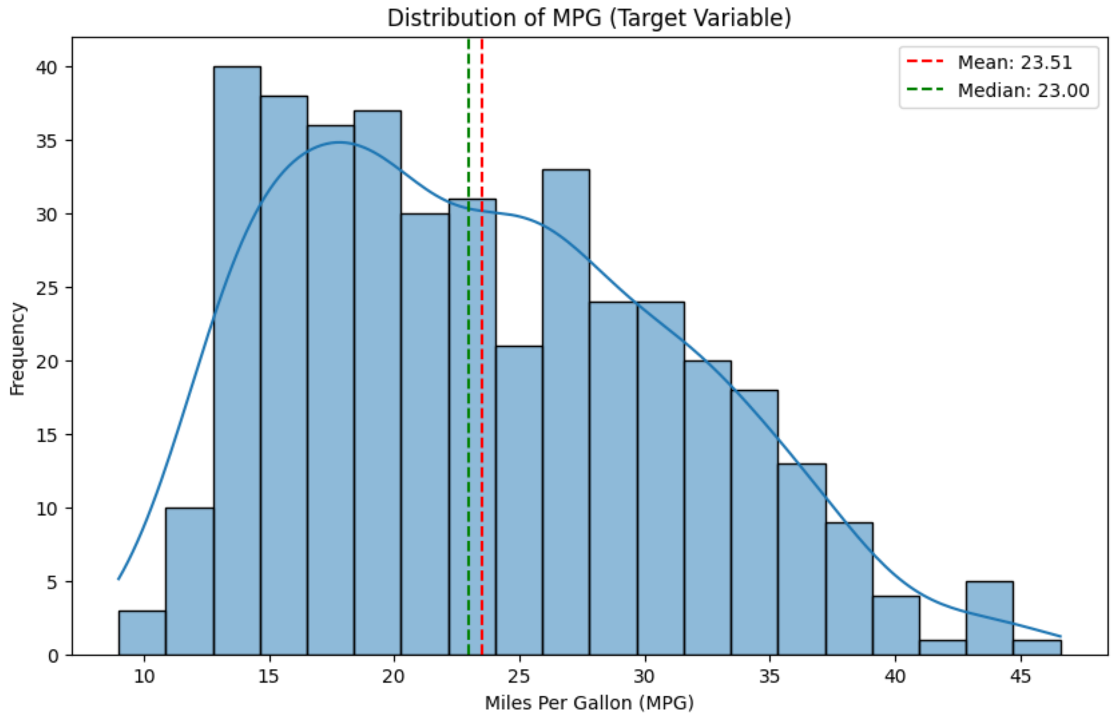

```
horsepower has 5 outliers (z-score > 3)
acceleration has 2 outliers (z-score > 3)
```

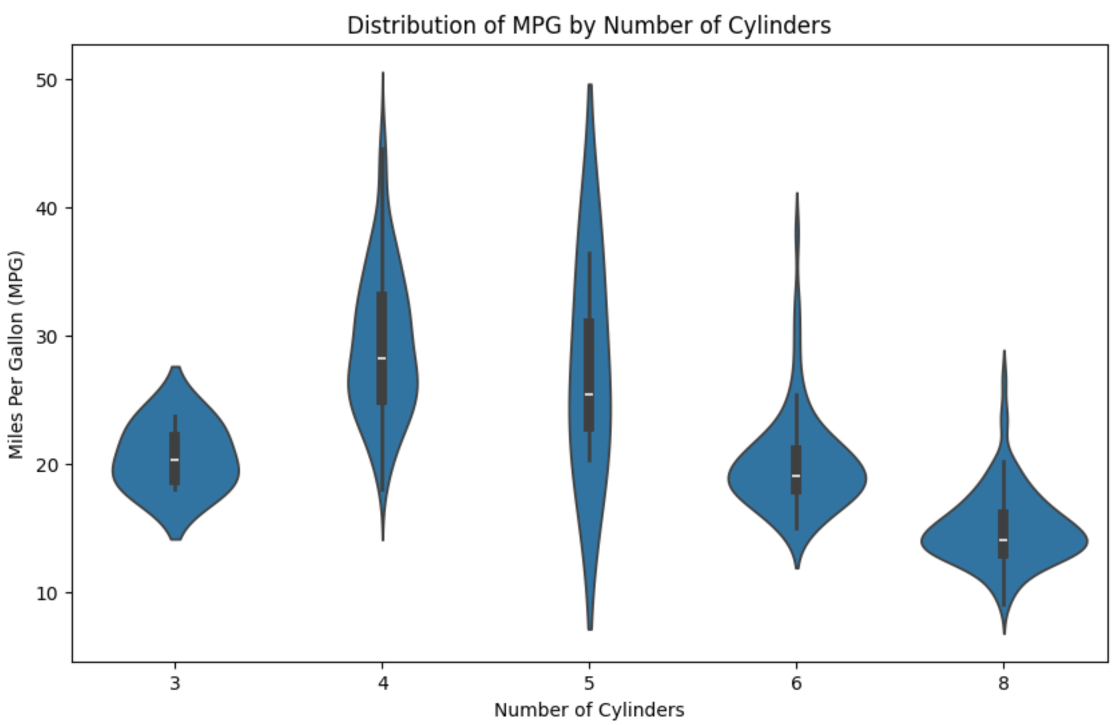


2.2.3 Check for class imbalance in the target variable (as applicable). While our target variable (mpg) is continuous for regression, we'll examine its distribution to ensure it's well-represented across its range and doesn't have concentrated values that could bias our model.

[14]:
```
# Examine the distribution of the target variable (mpg)
plt.figure(figsize=(10, 6))
sns.histplot(df['mpg'], bins=20, kde=True)
plt.title('Distribution of MPG (Target Variable)')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Frequency')
plt.axvline(df['mpg'].mean(), color='red', linestyle='--', label=f'Mean:␣
  ↪{df["mpg"].mean():.2f}')
plt.axvline(df['mpg'].median(), color='green', linestyle='--', label=f'Median:␣
  ↪{df["mpg"].median():.2f}')
plt.legend()
plt.show()

# Create a violin plot of mpg by number of cylinders
plt.figure(figsize=(10, 6))
sns.violinplot(x='cylinders', y='mpg', data=df)
plt.title('Distribution of MPG by Number of Cylinders')
plt.xlabel('Number of Cylinders')
plt.ylabel('Miles Per Gallon (MPG)')
```

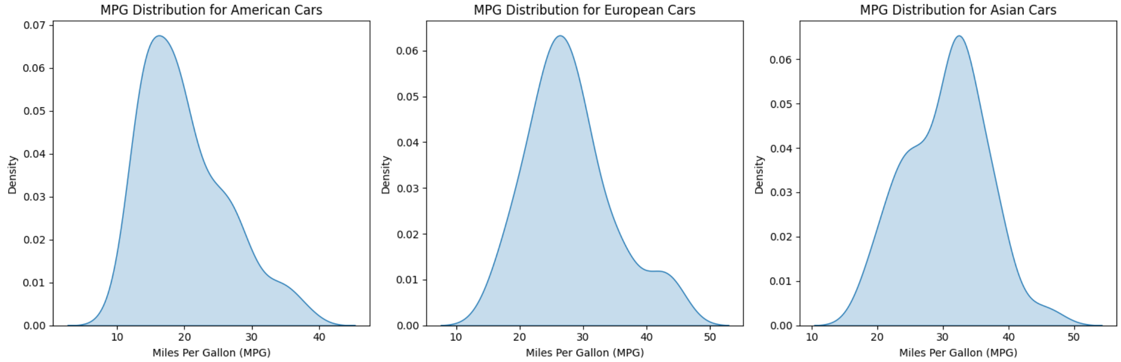
```python
plt.show()
```





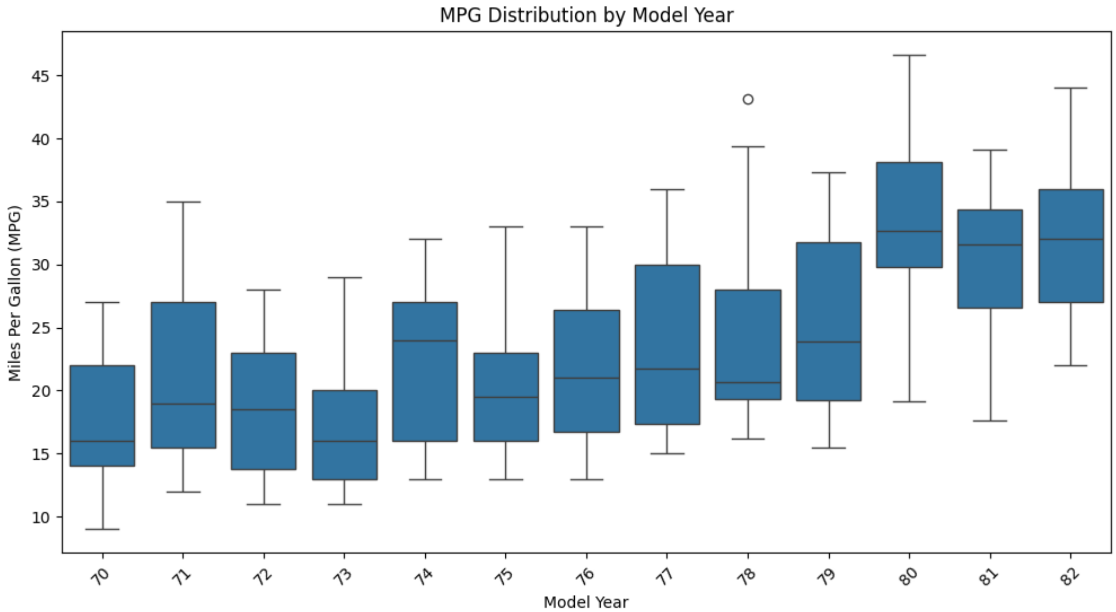
[15]:

```
# Create mpg distribution by origin
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
sns.kdeplot(df[df['origin_america'] == True]['mpg'], fill=True, label='America')
plt.title('MPG Distribution for American Cars')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')

plt.subplot(1, 3, 2)
sns.kdeplot(df[df['origin_europe'] == True]['mpg'], fill=True, label='Europe')
plt.title('MPG Distribution for European Cars')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')

plt.subplot(1, 3, 3)
sns.kdeplot(df[df['origin_asia'] == True]['mpg'], fill=True, label='Asia')
plt.title('MPG Distribution for Asian Cars')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')

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



[16]:

```
# Let's also look at MPG over the years to detect any trends
plt.figure(figsize=(12, 6))
sns.boxplot(x='model_year', y='mpg', data=df)
plt.title('MPG Distribution by Model Year')
plt.xlabel('Model Year')
plt.ylabel('Miles Per Gallon (MPG)')
plt.xticks(rotation=45)
plt.show()
```



1.4.3 2.3 Feature selection and engineering

2.3.1 Create new features (as applicable).

[17]:
```python
# Feature engineering: Extract car make from car_name

# Extract the car make (first word) from the car_name column
df['make'] = df['car_name'].str.split().str[0]

# Display the first 10 rows with the new 'make' column
print("First 10 rows with make column added:")
display(df.head(10))
```

```
First 10 rows with make column added:

    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0  3504.0          12.0
1  15.0          8         350.0       165.0  3693.0          11.5
2  18.0          8         318.0       150.0  3436.0          11.0
3  16.0          8         304.0       150.0  3433.0          12.0
4  17.0          8         302.0       140.0  3449.0          10.5
5  15.0          8         429.0       198.0  4341.0          10.0
6  14.0          8         454.0       220.0  4354.0           9.0
7  14.0          8         440.0       215.0  4312.0           8.5
8  14.0          8         455.0       225.0  4425.0          10.0
9  15.0          8         390.0       190.0  3850.0           8.5

   model_year                   car_name  origin_america  origin_asia  \
0          70  chevrolet chevelle malibu            True        False
```

```
1          70         buick skylark 320            True        False
2          70        plymouth satellite            True        False
3          70             amc rebel sst            True        False
4          70               ford torino            True        False
5          70          ford galaxie 500            True        False
6          70          chevrolet impala            True        False
7          70         plymouth fury iii            True        False
8          70          pontiac catalina            True        False
9          70         amc ambassador dpl           True        False

   origin_europe       make
0          False  chevrolet
1          False      buick
2          False   plymouth
3          False        amc
4          False       ford
5          False       ford
6          False  chevrolet
7          False   plymouth
8          False    pontiac
9          False        amc
```

2.3.2 Transform or combine existing features to improve model performance (as applicable).

Transform Makes

[18]:

```python
# Define mapping dictionary for make standardization
make_mapping = {
    'chevy': 'chevrolet',
    'chevroelt': 'chevrolet',
    'vokswagen': 'volkswagen',
    'vw': 'volkswagen',
    'toyouta': 'toyota',
    'mercedes': 'mercedes-benz',
    'maxda': 'mazda',
    'hi': 'hindustan',    # 'hi' appears to be Hindustan Motors
    'capri': 'ford'       # Capri was a model made by Ford
}

# Apply the mapping to standardize makes
df['make'] = df['make'].replace(make_mapping)

# Show the updated make counts
make_counts = df['make'].value_counts()
print("Updated make counts after standardization:")
print(make_counts)
```

```python
# Display the first 10 rows to verify changes
print("\nFirst 10 rows with standardized make column:")
display(df.head(10))
```

```
Updated make counts after standardization:
make
ford             52
chevrolet        47
plymouth         31
amc              28
dodge            28
toyota           26
datsun           23
volkswagen       22
buick            17
pontiac          16
honda            13
mazda            12
mercury          11
oldsmobile       10
fiat              8
peugeot           8
audi              7
volvo             6
chrysler          6
renault           5
saab              4
opel              4
subaru            4
mercedes-benz     3
bmw               2
cadillac          2
hindustan         1
triumph           1
nissan            1
Name: count, dtype: int64

First 10 rows with standardized make column:
```

```
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0  3504.0          12.0
1  15.0          8         350.0       165.0  3693.0          11.5
2  18.0          8         318.0       150.0  3436.0          11.0
3  16.0          8         304.0       150.0  3433.0          12.0
4  17.0          8         302.0       140.0  3449.0          10.5
5  15.0          8         429.0       198.0  4341.0          10.0
6  14.0          8         454.0       220.0  4354.0           9.0
7  14.0          8         440.0       215.0  4312.0           8.5
```

```
8  14.0           8          455.0       225.0  4425.0           10.0
9  15.0           8          390.0       190.0  3850.0            8.5

   model_year                   car_name  origin_america  origin_asia  \
0          70  chevrolet chevelle malibu            True        False
1          70          buick skylark 320            True        False
2          70         plymouth satellite            True        False
3          70              amc rebel sst            True        False
4          70                ford torino            True        False
5          70           ford galaxie 500            True        False
6          70           chevrolet impala            True        False
7          70          plymouth fury iii            True        False
8          70           pontiac catalina            True        False
9          70         amc ambassador dpl            True        False

   origin_europe       make
0          False  chevrolet
1          False      buick
2          False   plymouth
3          False        amc
4          False       ford
5          False       ford
6          False  chevrolet
7          False   plymouth
8          False    pontiac
9          False        amc
```

[19]:
```
print("All the unique makes")
print(make_counts)
```

```
All the unique makes
make
ford              52
chevrolet         47
plymouth          31
amc               28
dodge             28
toyota            26
datsun            23
volkswagen        22
buick             17
pontiac           16
honda             13
mazda             12
mercury           11
oldsmobile        10
fiat               8
peugeot            8
```

```
audi              7
volvo             6
chrysler          6
renault           5
saab              4
opel              4
subaru            4
mercedes-benz     3
bmw               2
cadillac          2
hindustan         1
triumph           1
nissan            1
Name: count, dtype: int64
```

[20]:

```
# Create a function to check if each make has exactly one origin
def check_make_origin_consistency(df):
    # Get unique makes
    makes = df['make'].unique()

    # Dictionary to store results
    make_origins = {}
    inconsistent_makes = {}

    # Check each make
    for make in makes:
        # Get data for this make
        make_data = df[df['make'] == make]

        # Check which origins this make has
        has_america = make_data['origin_america'].any()
        has_asia = make_data['origin_asia'].any()
        has_europe = make_data['origin_europe'].any()

        # Count number of origins
        origin_count = sum([has_america, has_asia, has_europe])

        # Store the origin
        if has_america:
            make_origins[make] = 'America'
        elif has_asia:
            make_origins[make] = 'Asia'
        elif has_europe:
            make_origins[make] = 'Europe'

        # If more than one origin, this make is inconsistent
        if origin_count > 1:
```

```
            inconsistent_makes[make] = {
                'America': has_america,
                'Asia': has_asia,
                'Europe': has_europe
            }

    return make_origins, inconsistent_makes

# Run the check
make_origins, inconsistent_makes = check_make_origin_consistency(df)

# Show results
print("Number of unique makes:", len(make_origins))
print("\nOrigin countries by make:")
for make, origin in sorted(make_origins.items()):
    print(f"{make}: {origin}")

if inconsistent_makes:
    print("\nWARNING: The following makes have inconsistent origins:")
    for make, origins in inconsistent_makes.items():
        print(f"{make}: {origins}")
else:
    print("\nAll makes have consistent origins!")

# Create a correlation matrix visualization
plt.figure(figsize=(12, 10))
# Create a crosstab between make and origin
make_origin_crosstab = pd.crosstab(df['make'], [df['origin_america'],
 ↪df['origin_asia'], df['origin_europe']])
print("\nCrosstab of make and origin:")
display(make_origin_crosstab)

# Create a heatmap showing make-origin relationships
plt.figure(figsize=(15, 12))
# Define a palette that has True as red and False as white/light
# First, create a temporary DataFrame with just the origins
origin_df = df.groupby('make')[['origin_america', 'origin_asia',
 ↪'origin_europe']].first()
sns.heatmap(origin_df, cmap=['white', 'blue'], cbar=False)
plt.title('Origin by Make (Blue indicates True)')
plt.ylabel('Make')
plt.tight_layout()
plt.show()
```

```
Number of unique makes: 29

Origin countries by make:
```

```
amc: America
audi: Europe
bmw: Europe
buick: America
cadillac: America
chevrolet: America
chrysler: America
datsun: Asia
dodge: America
fiat: Europe
ford: America
hindustan: America
honda: Asia
mazda: Asia
mercedes-benz: Europe
mercury: America
nissan: Asia
oldsmobile: America
opel: Europe
peugeot: Europe
plymouth: America
pontiac: America
renault: Europe
saab: Europe
subaru: Asia
toyota: Asia
triumph: Europe
volkswagen: Europe
volvo: Europe

All makes have consistent origins!

Crosstab of make and origin:

origin_america False       True
origin_asia    False True  False
origin_europe  True  False False
make
amc                0     0    28
audi               7     0     0
bmw                2     0     0
buick              0     0    17
cadillac           0     0     2
chevrolet          0     0    47
chrysler           0     0     6
datsun             0    23     0
dodge              0     0    28
fiat               8     0     0
```

```
ford               0    0   52
hindustan          0    0    1
honda              0   13    0
mazda              0   12    0
mercedes-benz      3    0    0
mercury            0    0   11
nissan             0    1    0
oldsmobile         0    0   10
opel               4    0    0
peugeot            8    0    0
plymouth           0    0   31
pontiac            0    0   16
renault            5    0    0
saab               4    0    0
subaru             0    4    0
toyota             0   26    0
triumph            1    0    0
volkswagen        22    0    0
volvo              6    0    0

<Figure size 1200x1000 with 0 Axes>
```

[ ]:
```python
# Get value counts of each make
make_counts = df['make'].value_counts()


# Identify makes with fewer than 10 instances
rare_makes = make_counts[make_counts < 10]
print("Makes with fewer than 5 instances:")
print(rare_makes)

# Calculate the total number of cars with these rare makes
rare_makes_count = sum(rare_makes)
total_cars = len(df)
rare_percentage = (rare_makes_count / total_cars) * 100

print(f"\nTotal cars in dataset: {total_cars}")
print(f"Cars with rare makes (< 5 instances): {rare_makes_count}")
print(f"Percentage of dataset with rare makes: {rare_percentage:.2f}%")
```

```
Makes with fewer than 5 instances:
make
fiat             8
peugeot          8
audi             7
volvo            6
chrysler         6
renault          5
saab             4
opel             4
subaru           4
mercedes-benz    3
bmw              2
cadillac         2
hindustan        1
triumph          1
nissan           1
Name: count, dtype: int64

Total cars in dataset: 398
Cars with rare makes (< 5 instances): 62
Percentage of dataset with rare makes: 15.58%
```

[40]:
```python
# Store the original dataframe size
original_size = len(df)

# Get the list of rare makes (makes with fewer than 5 instances)
rare_makes_list = rare_makes.index.tolist()
```

```python
# Filter out rows with rare makes
df_filtered = df[~df['make'].isin(rare_makes_list)]

# Calculate the new size
new_size = len(df_filtered)
removed_rows = original_size - new_size

# Display results
print(f"Original dataset size: {original_size} rows")
print(f"After removing rare makes: {new_size} rows")
print(f"Removed {removed_rows} rows ({(removed_rows/original_size)*100:.2f}% of␣
↪data)")

# Check the distribution of makes in the filtered dataset
print("\nMake distribution in filtered dataset:")
make_distribution = df_filtered['make'].value_counts()
print(make_distribution)

# Display the first few rows of the filtered dataset
print("\nFirst 5 rows of filtered dataset:")
display(df_filtered.head())

# Optionally, reassign to the original variable if you want to continue with␣
↪the filtered dataset
# df = df_filtered.copy()
```

```
Original dataset size: 398 rows
After removing rare makes: 336 rows
Removed 62 rows (15.58% of data)

Make distribution in filtered dataset:
make
ford          52
chevrolet     47
plymouth      31
amc           28
dodge         28
toyota        26
datsun        23
volkswagen    22
buick         17
pontiac       16
honda         13
mazda         12
mercury       11
oldsmobile    10
```

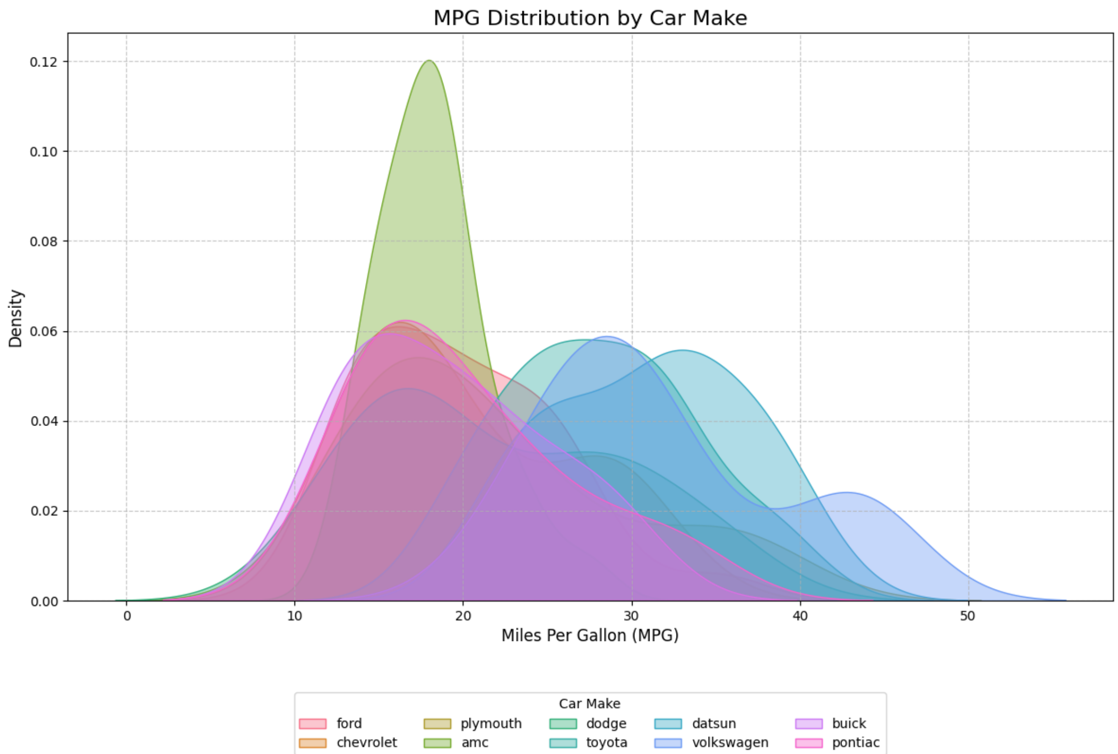
```
Name: count, dtype: int64

First 5 rows of filtered dataset:
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0  3504.0          12.0
1  15.0          8         350.0       165.0  3693.0          11.5
2  18.0          8         318.0       150.0  3436.0          11.0
3  16.0          8         304.0       150.0  3433.0          12.0
4  17.0          8         302.0       140.0  3449.0          10.5

   model_year                   car_name  origin_america  origin_asia  \
0          70  chevrolet chevelle malibu            True        False
1          70          buick skylark 320            True        False
2          70         plymouth satellite            True        False
3          70              amc rebel sst            True        False
4          70                ford torino            True        False

   origin_europe       make
0          False  chevrolet
1          False      buick
2          False   plymouth
3          False        amc
4          False       ford
```

[23]:

```python
# Get the most common makes (to avoid too many colors in the plot)
top_makes = df['make'].value_counts().head(10).index.tolist()
print(f"Top 10 makes: {top_makes}")

# Filter dataset to include only the top makes for better visualization
filtered_df = df[df['make'].isin(top_makes)]

# Create a figure with adequate size
plt.figure(figsize=(12, 8))

# Use a color palette that distinguishes between different makes
colors = sns.color_palette("husl", len(top_makes))

# Plot KDE curves for each make
for i, make in enumerate(top_makes):
    make_data = filtered_df[filtered_df['make'] == make]
    sns.kdeplot(make_data['mpg'], fill=True, color=colors[i], alpha=0.4,␣
 ↪label=make, common_norm=False)

# Add plot details
plt.title('MPG Distribution by Car Make', fontsize=16)
plt.xlabel('Miles Per Gallon (MPG)', fontsize=12)
plt.ylabel('Density', fontsize=12)
```

```python
plt.grid(True, linestyle='--', alpha=0.7)

# Add a legend at the bottom
plt.legend(title='Car Make', bbox_to_anchor=(0.5, -0.15), loc='upper center',␣
↪ncol=5)

# Adjust layout to make room for the legend
plt.tight_layout()
plt.subplots_adjust(bottom=0.2)

# Display the plot
plt.show()

# Optionally, add some statistics for each make
print("\nMPG Statistics by Make:")
for make in top_makes:
    make_data = filtered_df[filtered_df['make'] == make]
    print(f"{make.capitalize()}:")
    print(f"  Count: {len(make_data)}")
    print(f"  Average MPG: {make_data['mpg'].mean():.2f}")
    print(f"  Min-Max: {make_data['mpg'].min():.1f}-{make_data['mpg'].max():.
↪1f}")
    print()
```

```
Top 10 makes: ['ford', 'chevrolet', 'plymouth', 'amc', 'dodge', 'toyota',
'datsun', 'volkswagen', 'buick', 'pontiac']
```




```
MPG Statistics by Make:
Ford:
  Count: 52
  Average MPG: 19.80
  Min-Max: 10.0-36.1

Chevrolet:
  Count: 47
  Average MPG: 20.22
  Min-Max: 10.0-34.0

Plymouth:
  Count: 31
  Average MPG: 21.70
  Min-Max: 13.0-39.0

Amc:
  Count: 28
  Average MPG: 18.25
  Min-Max: 13.0-27.4

Dodge:
```

```
  Count: 28
  Average MPG: 22.06
  Min-Max: 11.0-36.0

Toyota:
  Count: 26
  Average MPG: 28.17
  Min-Max: 19.0-39.1

Datsun:
  Count: 23
  Average MPG: 31.11
  Min-Max: 22.0-40.8

Volkswagen:
  Count: 22
  Average MPG: 31.84
  Min-Max: 22.0-44.3

Buick:
  Count: 17
  Average MPG: 19.18
  Min-Max: 12.0-30.0

Pontiac:
  Count: 16
  Average MPG: 20.01
  Min-Max: 13.0-33.5
```

2.3.3 Scale or normalize data (as applicable).

1.4.4 Reflection 2: What patterns or anomalies do you see? Do any features stand out? What preprocessing steps were necessary to clean and improve the data? Did you create or modify any features to improve performance?

1.5 Section 3. Feature Selection and Justification

1.5.1 3.1 Choose features and target

Model Year As a numerical feature:

Preserves the chronological relationship between years Implicitly models the steady improvement in fuel efficiency over time Simpler model with fewer parameters

As a categorical feature:

Allows for non-linear relationships between specific years and mpg May capture regulatory changes that happened in specific years Increases model complexity significantly

Since the data shows a fairly steady upward trend in mpg by model year (as seen in your boxplots), treating it as a numerical feature is likely sufficient and won't cause overfitting. The linear relationship appears reasonably strong.

3.1.1 Select two or more input features We are going to use model year, weight, and make.

3.1.2 Select a target variable (as applicable) The assignment indicated we ought to use MPG. I was hoping I could feed data and use it to predict the origin of the car or the make, and I might in the future as time allows.

3.1.3 Justify your selection with reasoning.

1.5.2 3.2 Define X and y

3.2.1 Assign input features to X

[24]:
```python
# 3.2.1 Assign input features to X

# Select the features we want to use for our model
# Convert 'make' to a categorical variable using one-hot encoding
make_encoded = pd.get_dummies(df['make'], prefix='make', drop_first=True)

# Combine the encoded make columns with our other numerical features
X = pd.concat([
    df[['weight', 'model_year']],
    make_encoded
], axis=1)

# Display the first few rows of our feature set
print("First 5 rows of our features (X):")
display(X.head())

# Shape of the feature matrix
print(f"\nShape of X: {X.shape}")
print(f"Number of features: {X.shape[1]}")
```

```
First 5 rows of our features (X):

   weight  model_year  make_audi  make_bmw  make_buick  make_cadillac  \
0  3504.0          70      False     False       False          False
1  3693.0          70      False     False        True          False
2  3436.0          70      False     False       False          False
3  3433.0          70      False     False       False          False
4  3449.0          70      False     False       False          False

   make_chevrolet  make_chrysler  make_datsun  make_dodge  …  make_peugeot  \
0            True          False        False       False  …         False
1           False          False        False       False  …         False
2           False          False        False       False  …         False
```

```
3           False          False         False       False  ...         False
4           False          False         False       False  ...         False

   make_plymouth  make_pontiac  make_renault  make_saab  make_subaru  \
0          False         False         False      False        False
1          False         False         False      False        False
2           True         False         False      False        False
3          False         False         False      False        False
4          False         False         False      False        False

   make_toyota  make_triumph  make_volkswagen  make_volvo
0        False         False            False       False
1        False         False            False       False
2        False         False            False       False
3        False         False            False       False
4        False         False            False       False

[5 rows x 30 columns]
```

```
Shape of X: (398, 30)
Number of features: 30
```

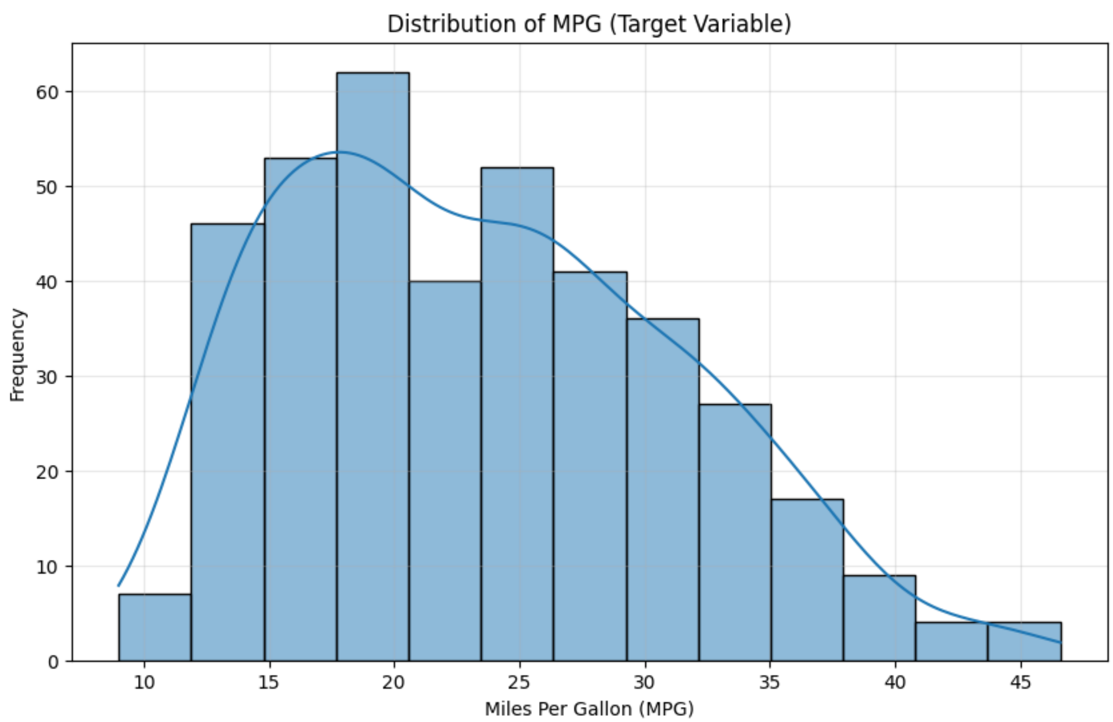
3.2.2 Assign target variable to y (as applicable)

[25]:
```python
# 3.2.2 Assign target variable to y

# Our target variable is 'mpg'
y = df['mpg']

# Display the first few values of our target
print("First 5 values of our target (y):")
display(y.head())

# Basic statistics of our target variable
print("\nBasic statistics of MPG (target variable):")
print(y.describe())

# Visualize the distribution of our target variable
plt.figure(figsize=(10, 6))
sns.histplot(y, kde=True)
plt.title('Distribution of MPG (Target Variable)')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Frequency')
plt.grid(True, alpha=0.3)
plt.show()
```

```
First 5 values of our target (y):
```

```
0    18.0
```

```
1    15.0
2    18.0
3    16.0
4    17.0
Name: mpg, dtype: float64

Basic statistics of MPG (target variable):
count    398.000000
mean      23.514573
std        7.815984
min        9.000000
25%       17.500000
50%       23.000000
75%       29.000000
max       46.600000
Name: mpg, dtype: float64
```



1.5.3 Reflection 3: Why did you choose these features? How might they impact predictions or accuracy?

1.6 Section 4. Train a Model (Linear Regression)

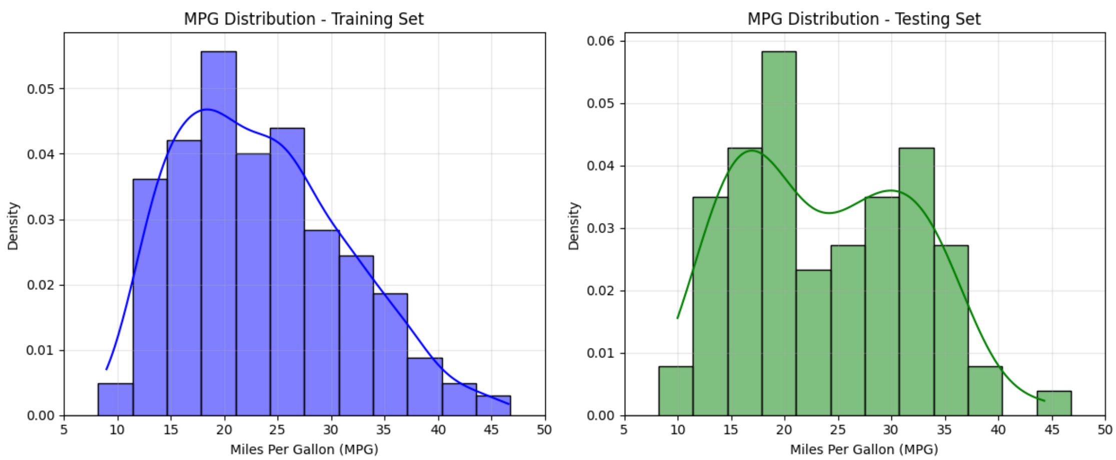
1.6.1 4.1 Split the data into training and test sets using train_test_split (or StratifiedShuffleSplit if class imbalance is an issue).

4.1.1 Initial Random Split

[26]:
```python
# 4.1 Split the data into training and test sets

# Set a random seed for reproducibility
np.random.seed(state_setter)

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=state_setter)

# Display the shapes of our training and testing sets
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}")

# Create a better comparison of MPG distributions with same bins and␣
 ↪normalization
plt.figure(figsize=(12, 5))

# Define common binning
bins = np.linspace(5, 50, 15)  # 15 bins from 5 to 50 MPG

plt.subplot(1, 2, 1)
sns.histplot(y_train, bins=bins, kde=True, color='blue', stat='density')
plt.title('MPG Distribution - Training Set')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')
plt.grid(True, alpha=0.3)
plt.xlim(5, 50)

plt.subplot(1, 2, 2)
sns.histplot(y_test, bins=bins, kde=True, color='green', stat='density')
plt.title('MPG Distribution - Testing Set')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')
plt.grid(True, alpha=0.3)
plt.xlim(5, 50)

plt.tight_layout()
```

```python
plt.show()

# Compare statistical measures
print("\nTraining set MPG statistics:")
print(y_train.describe())
print("\nTesting set MPG statistics:")
print(y_test.describe())
```

```
X_train shape: (318, 30)
X_test shape: (80, 30)
y_train shape: (318,)
y_test shape: (80,)
```




```
Training set MPG statistics:
count    318.000000
mean      23.443396
std        7.762728
min        9.000000
25%       17.500000
50%       22.750000
75%       28.325000
max       46.600000
Name: mpg, dtype: float64

Testing set MPG statistics:
count    80.000000
mean     23.797500
std       8.067876
min      10.000000
25%      16.875000
50%      23.850000
```

```
75%       31.000000
max       44.300000
Name: mpg, dtype: float64
```

These distributions are concerningly different. Let's tackle it with a stratified test-train-split…but stratify it across what?

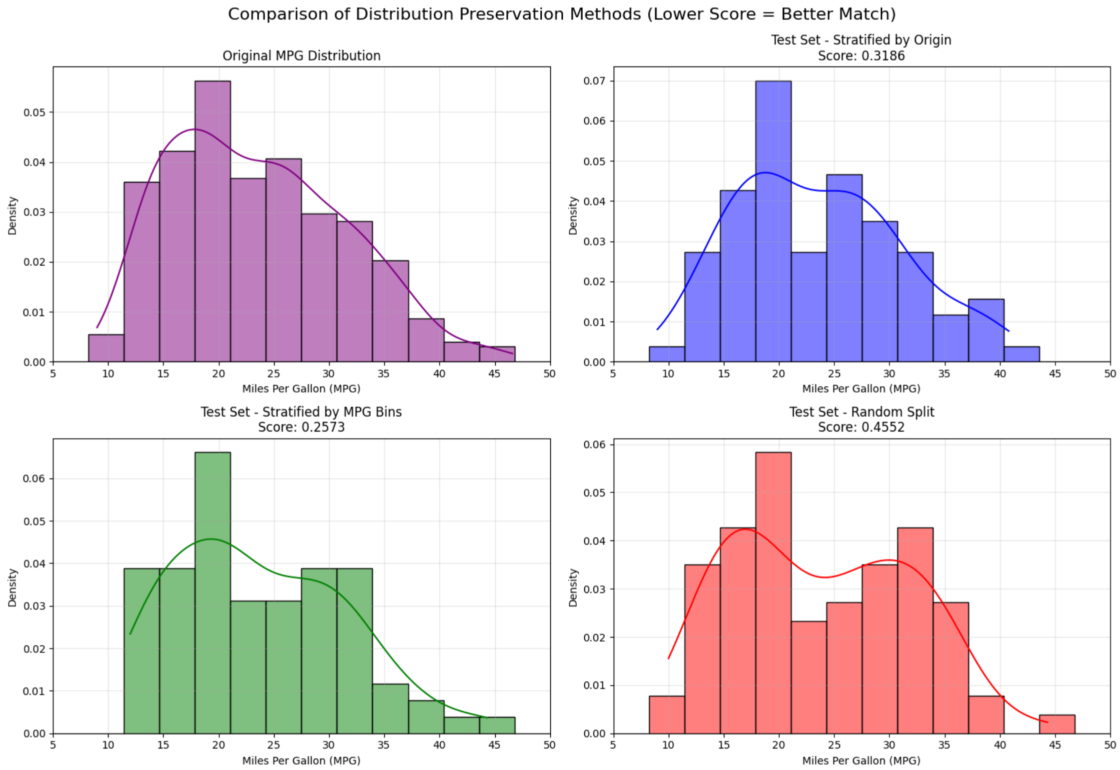
4.1.2 Stratified Test Train Split - Stratification Comparison Per our research earlier, there are too many small 'make' bins, so let's consider origin instead. Let's also look at separating MPG into Bins and stratifying according to the bins. We will them compare mpg distributions produced by these 2 methods against the random split we've done here and the original data set's mpg distribution

[27]:
```python
# Function to evaluate similarity between distributions
def compare_distributions(original, sample, method_name):
    # Calculate statistical distances
    ks_stat, ks_pval = ks_2samp(original, sample)
    w_distance = wasserstein_distance(original, sample)
    e_distance = energy_distance(original, sample)

    # Calculate basic statistics and their differences
    orig_stats = original.describe()
    sample_stats = sample.describe()

    # Calculate absolute differences in key statistics
    mean_diff = abs(orig_stats['mean'] - sample_stats['mean'])
    std_diff = abs(orig_stats['std'] - sample_stats['std'])

    # Quartile differences
    q1_diff = abs(orig_stats['25%'] - sample_stats['25%'])
    median_diff = abs(orig_stats['50%'] - sample_stats['50%'])
    q3_diff = abs(orig_stats['75%'] - sample_stats['75%'])


    # Return a composite score
    # Weighted sum of distances
    composite_score = (ks_stat * 0.3) + (w_distance * 0.3) + (e_distance * 0.1)␣
 ↪+ \
                      (mean_diff * 0.1) + (std_diff * 0.1) + \
                      (q1_diff * 0.03) + (median_diff * 0.04) + (q3_diff * 0.03)

    """The score is a weighted combination of several statistical distance␣
 ↪metrics:

    Kolmogorov-Smirnov statistic (30% weight): Measures the maximum difference␣
 ↪between two cumulative distribution functions. A value of 0 means identical␣
 ↪distributions.
```

```
    Wasserstein distance (30% weight): Also known as the "Earth Mover's␣
↪Distance," it measures how much "work" it would take to transform one␣
↪distribution into another. Lower values mean distributions are more similar.
    Energy distance (10% weight): Another statistical distance metric that's␣
↪sensitive to differences in both shape and location of distributions.
    Differences in key statistics (30% total weight):

    Mean difference (10%)
    Standard deviation difference (10%)
    Quartile differences (10% total: 3% for Q1, 4% for median, 3% for Q3)

    These components capture different aspects of distribution similarity:

    The statistical distance metrics (KS, Wasserstein, Energy) capture overall␣
↪shape differences
    The mean and standard deviation differences capture central tendency and␣
↪spread
    The quartile differences capture structural details of the distribution"""


    return {
        'method': method_name,
        'ks_stat': ks_stat,
        'ks_pval': ks_pval,
        'w_distance': w_distance,
        'e_distance': e_distance,
        'mean_diff': mean_diff,
        'std_diff': std_diff,
        'q1_diff': q1_diff,
        'median_diff': median_diff,
        'q3_diff': q3_diff,
        'composite_score': composite_score
    }

# Original data
y_original = df['mpg']

# Prepare features (same as before)
make_encoded = pd.get_dummies(df['make'], prefix='make', drop_first=True)
X = pd.concat([df[['weight', 'model_year']], make_encoded], axis=1)

# Method 1: Stratify by origin
strat_var_origin = np.where(df['origin_america'], 'america',
                   np.where(df['origin_europe'], 'europe', 'asia'))

X_train_origin, X_test_origin, y_train_origin, y_test_origin = train_test_split(
    X, y_original,
```

```
    test_size=0.2,
    random_state=state_setter,
    stratify=strat_var_origin
)

# Method 2: Stratify by 5 binned MPG values
mpg_bins = pd.qcut(df['mpg'], q=5, labels=False, duplicates='drop')

X_train_mpg, X_test_mpg, y_train_mpg, y_test_mpg = train_test_split(
    X, y_original,
    test_size=0.2,
    random_state=state_setter,
    stratify=mpg_bins
)

# Method 3: Regular random split (as baseline)
X_train_rand, X_test_rand, y_train_rand, y_test_rand = train_test_split(
    X, y_original,
    test_size=0.2,
    random_state=state_setter
)

# Compare distributions
origin_results = compare_distributions(y_original, y_test_origin, "Origin")
mpg_results = compare_distributions(y_original, y_test_mpg, "MPG Bins")
random_results = compare_distributions(y_original, y_test_rand, "Random")
# Visualize all distributions
plt.figure(figsize=(15, 10))

# Define common binning
bins = np.linspace(5, 50, 15)

# Original distribution
plt.subplot(2, 2, 1)
sns.histplot(y_original, bins=bins, kde=True, color='purple', stat='density')
plt.title('Original MPG Distribution')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')
plt.grid(True, alpha=0.3)
plt.xlim(5, 50)

# Origin stratification
plt.subplot(2, 2, 2)
sns.histplot(y_test_origin, bins=bins, kde=True, color='blue', stat='density')
plt.title(f'Test Set - Stratified by Origin\nScore:␣
 ↪{origin_results["composite_score"]:.4f}')
plt.xlabel('Miles Per Gallon (MPG)')
```

```
plt.ylabel('Density')
plt.grid(True, alpha=0.3)
plt.xlim(5, 50)

# MPG bins stratification
plt.subplot(2, 2, 3)
sns.histplot(y_test_mpg, bins=bins, kde=True, color='green', stat='density')
plt.title(f'Test Set - Stratified by MPG Bins\nScore:␣
↪{mpg_results["composite_score"]:.4f}')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')
plt.grid(True, alpha=0.3)
plt.xlim(5, 50)

# Random split
plt.subplot(2, 2, 4)
sns.histplot(y_test_rand, bins=bins, kde=True, color='red', stat='density')
plt.title(f'Test Set - Random Split\nScore: {random_results["composite_score"]:.
↪4f}')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')
plt.grid(True, alpha=0.3)
plt.xlim(5, 50)

plt.tight_layout()
plt.suptitle('Comparison of Distribution Preservation Methods (Lower Score =␣
↪Better Match)', fontsize=16, y=1.02)
plt.show()
```



This suggests mpg bins is the best, and visually it definitely appears closest. Given the small set of the data however, let's cross-validate across 10 splits to confirm we achieve the same results.

Note - the score represents the dissimilarity between the test set distribution and the original distribution. Lower scores indicate test sets that more closely match the original MPG distribution, which is why lower is better, and 0 would mean the distributions are identical.

If curious for more detail, review the comments and calculations in the python code.

4.1.3 Cross Validated Stratification Factor Comparison

[28]:
```python
# Function to evaluate similarity between distributions
def compare_distributions(original, sample, method_name):
    # Calculate statistical distances
    ks_stat, ks_pval = ks_2samp(original, sample)
    w_distance = wasserstein_distance(original, sample)
    e_distance = energy_distance(original, sample)

    # Calculate basic statistics and their differences
    orig_stats = original.describe()
    sample_stats = sample.describe()

    # Calculate absolute differences in key statistics
    mean_diff = abs(orig_stats['mean'] - sample_stats['mean'])
    std_diff = abs(orig_stats['std'] - sample_stats['std'])
```

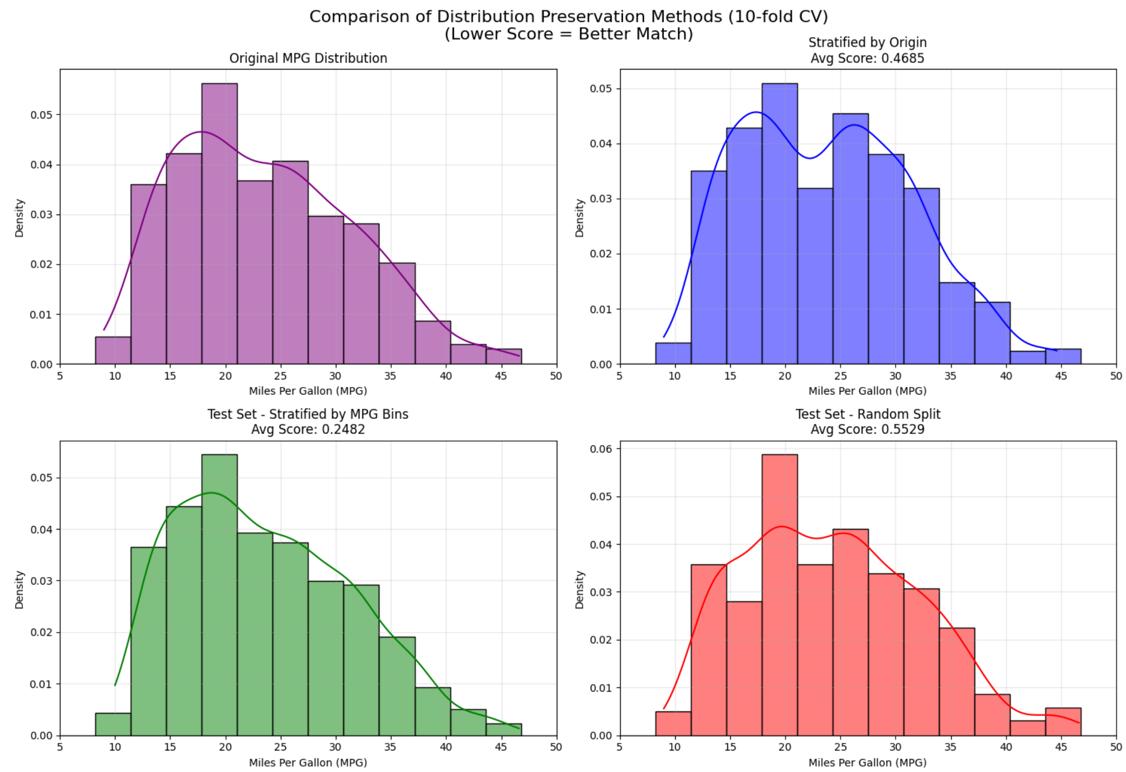
```
    # Quartile differences
    q1_diff = abs(orig_stats['25%'] - sample_stats['25%'])
    median_diff = abs(orig_stats['50%'] - sample_stats['50%'])
    q3_diff = abs(orig_stats['75%'] - sample_stats['75%'])

    # Return a composite score
    # Weighted sum of distances
    composite_score = (ks_stat * 0.3) + (w_distance * 0.3) + (e_distance * 0.1)␣
↪+ \
                      (mean_diff * 0.1) + (std_diff * 0.1) + \
                      (q1_diff * 0.03) + (median_diff * 0.04) + (q3_diff * 0.03)

    return {
        'method': method_name,
        'ks_stat': ks_stat,
        'ks_pval': ks_pval,
        'w_distance': w_distance,
        'e_distance': e_distance,
        'mean_diff': mean_diff,
        'std_diff': std_diff,
        'q1_diff': q1_diff,
        'median_diff': median_diff,
        'q3_diff': q3_diff,
        'composite_score': composite_score
    }

# Original data
y_original = df['mpg']

# Prepare features (same as before)
make_encoded = pd.get_dummies(df['make'], prefix='make', drop_first=True)
X = pd.concat([df[['weight', 'model_year']], make_encoded], axis=1)

# Number of cross-validation iterations
n_iterations = 10

# Initialize dictionaries to store results and test set distributions
results_origin = []
results_mpg = []
results_random = []

# Initialize arrays to accumulate histograms for averaging
all_y_test_origin = []
all_y_test_mpg = []
all_y_test_random = []
```

```
# Cross-validation loop
for i in range(n_iterations):
    # Set random state based on state_setter + iteration
    random_state = state_setter + i

    # Method 1: Stratify by origin
    strat_var_origin = np.where(df['origin_america'], 'america',
                    np.where(df['origin_europe'], 'europe', 'asia'))

    X_train_origin, X_test_origin, y_train_origin, y_test_origin =␣
 ↪train_test_split(
        X, y_original,
        test_size=0.2,
        random_state=random_state,
        stratify=strat_var_origin
    )


    mpg_bins = pd.qcut(df['mpg'], q=5, labels=False, duplicates='drop')




    X_train_mpg, X_test_mpg, y_train_mpg, y_test_mpg = train_test_split(
        X, y_original,
        test_size=0.2,
        random_state=random_state,
        stratify=mpg_bins
    )

    # Method 3: Regular random split (as baseline)
    X_train_rand, X_test_rand, y_train_rand, y_test_rand = train_test_split(
        X, y_original,
        test_size=0.2,
        random_state=random_state
    )

    # Evaluate distributions
    results_origin.append(compare_distributions(y_original, y_test_origin,␣
 ↪"Origin"))
    results_mpg.append(compare_distributions(y_original, y_test_mpg, "MPG␣
 ↪Bins"))
    results_random.append(compare_distributions(y_original, y_test_rand,␣
 ↪"Random"))

    # Store test set distributions for later averaging
    all_y_test_origin.append(y_test_origin)
```

```python
    all_y_test_mpg.append(y_test_mpg)
    all_y_test_random.append(y_test_rand)

# Calculate average scores for each method
def average_results(results_list):
    avg_results = {
        'ks_stat': np.mean([r['ks_stat'] for r in results_list]),
        'ks_pval': np.mean([r['ks_pval'] for r in results_list]),
        'w_distance': np.mean([r['w_distance'] for r in results_list]),
        'e_distance': np.mean([r['e_distance'] for r in results_list]),
        'mean_diff': np.mean([r['mean_diff'] for r in results_list]),
        'std_diff': np.mean([r['std_diff'] for r in results_list]),
        'q1_diff': np.mean([r['q1_diff'] for r in results_list]),
        'median_diff': np.mean([r['median_diff'] for r in results_list]),
        'q3_diff': np.mean([r['q3_diff'] for r in results_list]),
        'composite_score': np.mean([r['composite_score'] for r in results_list])
    }
    return avg_results

avg_origin = average_results(results_origin)
avg_mpg = average_results(results_mpg)
avg_random = average_results(results_random)

# Function to create a combined histogram representation across all CV␣
↪iterations
def create_averaged_histogram(all_samples, bins=15):
    # Concatenate all samples across CV iterations
    combined = pd.concat(all_samples)
    return combined

# Create averaged histograms
avg_hist_origin = create_averaged_histogram(all_y_test_origin)
avg_hist_mpg = create_averaged_histogram(all_y_test_mpg)
avg_hist_random = create_averaged_histogram(all_y_test_random)

# Visualize the average distributions
plt.figure(figsize=(15, 10))

# Define common binning
bins = np.linspace(5, 50, 15)

# Original distribution
plt.subplot(2, 2, 1)
sns.histplot(y_original, bins=bins, kde=True, color='purple', stat='density')
plt.title('Original MPG Distribution')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')
```

```
plt.grid(True, alpha=0.3)
plt.xlim(5, 50)

# Origin stratification (average across CV)
plt.subplot(2, 2, 2)
sns.histplot(avg_hist_origin, bins=bins, kde=True, color='blue', stat='density')
plt.title(f'Stratified by Origin\nAvg Score: {avg_origin["composite_score"]:.
↪4f}')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')
plt.grid(True, alpha=0.3)
plt.xlim(5, 50)

# MPG bins stratification (average across CV)
plt.subplot(2, 2, 3)
sns.histplot(avg_hist_mpg, bins=bins, kde=True, color='green', stat='density')
plt.title(f'Test Set - Stratified by MPG Bins\nAvg Score:␣
↪{avg_mpg["composite_score"]:.4f}')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')
plt.grid(True, alpha=0.3)
plt.xlim(5, 50)

# Random split (average across CV)
plt.subplot(2, 2, 4)
sns.histplot(avg_hist_random, bins=bins, kde=True, color='red', stat='density')
plt.title(f'Test Set - Random Split\nAvg Score: {avg_random["composite_score"]:.
↪4f}')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')
plt.grid(True, alpha=0.3)
plt.xlim(5, 50)

plt.tight_layout()
plt.suptitle('Comparison of Distribution Preservation Methods (10-fold␣
↪CV)\n(Lower Score = Better Match)', fontsize=16, y=1.02)
plt.show()

# Create a summary table of average results
methods = [
    {'method': 'Origin', **avg_origin},
    {'method': 'MPG Bins', **avg_mpg},
    {'method': 'Random', **avg_random}
]
comparison_df = pd.DataFrame(methods)
comparison_df = comparison_df.set_index('method')
```

```python
# Sort by composite score (lower is better)
comparison_df = comparison_df.sort_values('composite_score')

print("\n--- Summary of Average Distribution Similarity Across 10 Splits ---")
print(comparison_df[['ks_stat', 'w_distance', 'mean_diff', 'std_diff', 'composite_score']])

# Determine the winner
winner = comparison_df.index[0]
print(f"\nBest stratification method: {winner}")

# Show standard deviation of scores to see consistency
std_origin = np.std([r['composite_score'] for r in results_origin])
std_mpg = np.std([r['composite_score'] for r in results_mpg])
std_random = np.std([r['composite_score'] for r in results_random])

print("\n--- Standard Deviation of Scores Across 10 Splits ---")
print(f"Origin: {std_origin:.4f}")
print(f"MPG Bins: {std_mpg:.4f}")
print(f"Random: {std_random:.4f}")
```




```
--- Summary of Average Distribution Similarity Across 10 Splits ---
```

```
          ks_stat  w_distance  mean_diff  std_diff  composite_score
method
MPG Bins  0.043109    0.459575   0.167256  0.204233         0.248155
Origin    0.078392    0.829066   0.505710  0.416851         0.468472
Random    0.087827    0.974399   0.707552  0.371907         0.552930

Best stratification method: MPG Bins

--- Standard Deviation of Scores Across 10 Splits ---
Origin: 0.1263
MPG Bins: 0.0452
Random: 0.1821
```

This clearly indicates the mpg binning method gave the lowest score, and a distribution closest to the original. No surprise given the distribution of mpg itself is what we're interested in!

Stratifying the split across a greater number of bins should obviously help it match the the original distribution, but let's double check that and also confirm there aren't major performance implications. With our smaller data set, it probably won't have performance implications, but let's double check and confirm a larger number of bins is ideal

4.1.4 Bin Count Stratification Fine Tuning

[29]:
```python
# Function to evaluate similarity between distributions (same as before)
def compare_distributions(original, sample, method_name):
    # Calculate statistical distances
    ks_stat, ks_pval = ks_2samp(original, sample)
    w_distance = wasserstein_distance(original, sample)
    e_distance = energy_distance(original, sample)

    # Calculate basic statistics and their differences
    orig_stats = original.describe()
    sample_stats = sample.describe()

    # Calculate absolute differences in key statistics
    mean_diff = abs(orig_stats['mean'] - sample_stats['mean'])
    std_diff = abs(orig_stats['std'] - sample_stats['std'])

    # Quartile differences
    q1_diff = abs(orig_stats['25%'] - sample_stats['25%'])
    median_diff = abs(orig_stats['50%'] - sample_stats['50%'])
    q3_diff = abs(orig_stats['75%'] - sample_stats['75%'])

    # Return a composite score (lower is better)
    composite_score = (ks_stat * 0.3) + (w_distance * 0.3) + (e_distance * 0.1) + \
                      (mean_diff * 0.1) + (std_diff * 0.1) + \
                      (q1_diff * 0.03) + (median_diff * 0.04) + (q3_diff * 0.03)
```

```python
    return {
        'method': method_name,
        'ks_stat': ks_stat,
        'ks_pval': ks_pval,
        'w_distance': w_distance,
        'e_distance': e_distance,
        'mean_diff': mean_diff,
        'std_diff': std_diff,
        'q1_diff': q1_diff,
        'median_diff': median_diff,
        'q3_diff': q3_diff,
        'composite_score': composite_score
    }

# Original data
y_original = df['mpg']

# Prepare features
make_encoded = pd.get_dummies(df['make'], prefix='make', drop_first=True)
X = pd.concat([df[['weight', 'model_year']], make_encoded], axis=1)

# Number of cross-validation iterations
n_iterations = 10

# Different bin counts to test
bin_counts = [3, 5, 8, 15, 25]

# Dictionary to store results for different bin counts
bin_results = {}
bin_times = {}
all_distributions = {}

# Baseline: Origin stratification for comparison
all_y_test_origin = []
results_origin = []
origin_times = []

# Run cross-validation for origin stratification as baseline
for i in range(n_iterations):
    random_state = state_setter + i

    # Measure time for origin stratification
    start_time = time.time()

    # Method: Stratify by origin
    strat_var_origin = np.where(df['origin_america'], 'america',
                      np.where(df['origin_europe'], 'europe', 'asia'))
```

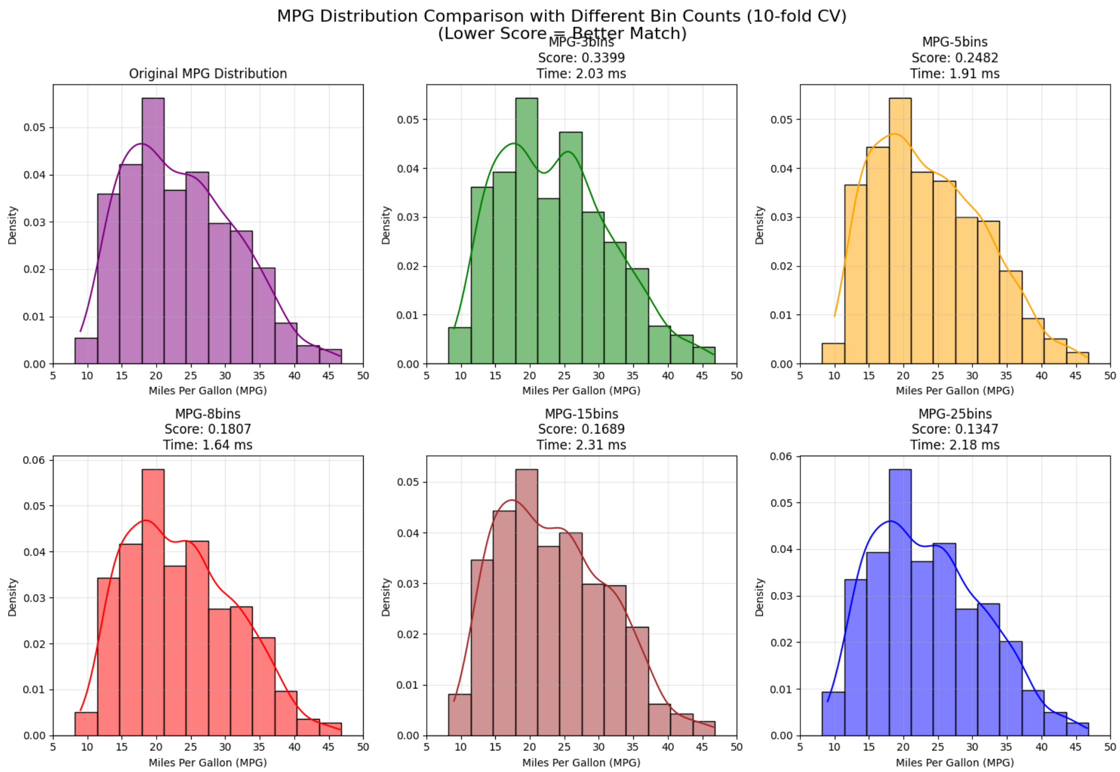
```
    X_train_origin, X_test_origin, y_train_origin, y_test_origin =␣
↪train_test_split(
        X, y_original,
        test_size=0.2,
        random_state=random_state,
        stratify=strat_var_origin
    )

    end_time = time.time()
    origin_times.append(end_time - start_time)

    # Evaluate distribution
    results_origin.append(compare_distributions(y_original, y_test_origin,␣
↪"Origin"))
    all_y_test_origin.append(y_test_origin)

# Average origin results
avg_origin = {
    'ks_stat': np.mean([r['ks_stat'] for r in results_origin]),
    'w_distance': np.mean([r['w_distance'] for r in results_origin]),
    'mean_diff': np.mean([r['mean_diff'] for r in results_origin]),
    'std_diff': np.mean([r['std_diff'] for r in results_origin]),
    'composite_score': np.mean([r['composite_score'] for r in results_origin])
}
avg_origin_time = np.mean(origin_times)

# Run cross-validation for each bin count
for bin_count in bin_counts:
    method_name = f"MPG-{bin_count}bins"
    bin_results[method_name] = []
    bin_times[method_name] = []
    all_distributions[method_name] = []

    for i in range(n_iterations):
        random_state = state_setter + i

        # Measure time
        start_time = time.time()

        # Create bins and stratify
        mpg_bins = pd.qcut(df['mpg'], q=bin_count, labels=False,␣
↪duplicates='drop')


        X_train_bins, X_test_bins, y_train_bins, y_test_bins = train_test_split(
            X, y_original,
```

```python
            test_size=0.2,
            random_state=random_state,
            stratify=mpg_bins
        )

        end_time = time.time()
        bin_times[method_name].append(end_time - start_time)

        # Evaluate distribution
        bin_results[method_name].append(compare_distributions(y_original,␣
 ↪y_test_bins, method_name))
        all_distributions[method_name].append(y_test_bins)

# Calculate averages for each bin count
avg_results = {}
for method_name in bin_results:
    avg_results[method_name] = {
        'ks_stat': np.mean([r['ks_stat'] for r in bin_results[method_name]]),
        'w_distance': np.mean([r['w_distance'] for r in␣
 ↪bin_results[method_name]]),
        'mean_diff': np.mean([r['mean_diff'] for r in␣
 ↪bin_results[method_name]]),
        'std_diff': np.mean([r['std_diff'] for r in bin_results[method_name]]),
        'composite_score': np.mean([r['composite_score'] for r in␣
 ↪bin_results[method_name]])
    }

# Average times
avg_times = {method: np.mean(times) for method, times in bin_times.items()}

# Function to create a combined histogram
def create_averaged_histogram(all_samples):
    return pd.concat(all_samples)

# Create averaged histograms
avg_hist_origin = create_averaged_histogram(all_y_test_origin)
avg_hist_bins = {method: create_averaged_histogram(dists)
                 for method, dists in all_distributions.items()}

# Visualize the distributions
plt.figure(figsize=(15, 10))
bins = np.linspace(5, 50, 15)

# Visualize the distributions
plt.figure(figsize=(15, 10))
bins = np.linspace(5, 50, 15)
```

```
# Original distribution
plt.subplot(2, 3, 1)
sns.histplot(y_original, bins=bins, kde=True, color='purple', stat='density')
plt.title('Original MPG Distribution')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')
plt.grid(True, alpha=0.3)
plt.xlim(5, 50)

# Different bin counts
colors = ['green', 'orange', 'red', 'brown', 'blue']  # Added blue as the 5th␣
↪color
subplot_positions = [2, 3, 4, 5, 6]  # Using all positions in the 2x3 grid

for i, ((method, dist), color, pos) in enumerate(zip(avg_hist_bins.items(),␣
↪colors, subplot_positions)):
    plt.subplot(2, 3, pos)
    sns.histplot(dist, bins=bins, kde=True, color=color, stat='density')
    plt.title(f'{method}\nScore: {avg_results[method]["composite_score"]:.
↪4f}\nTime: {avg_times[method]*1000:.2f} ms')
    plt.xlabel('Miles Per Gallon (MPG)')
    plt.ylabel('Density')
    plt.grid(True, alpha=0.3)
    plt.xlim(5, 50)

plt.tight_layout()
plt.suptitle('MPG Distribution Comparison with Different Bin Counts (10-fold␣
↪CV)\n(Lower Score = Better Match)', fontsize=16, y=1.02)
plt.show()

# Create a summary table
summary_data = []

# Add origin as baseline
summary_data.append({
    'Method': 'Origin',
    'Score': avg_origin['composite_score'],
    'Time (ms)': avg_origin_time * 1000,
    'KS Stat': avg_origin['ks_stat'],
    'W-Distance': avg_origin['w_distance'],
    'Mean Diff': avg_origin['mean_diff'],
    'Std Diff': avg_origin['std_diff']
})

# Add bin methods
for method in avg_results:
    summary_data.append({
```

```
        'Method': method,
        'Score': avg_results[method]['composite_score'],
        'Time (ms)': avg_times[method] * 1000,
        'KS Stat': avg_results[method]['ks_stat'],
        'W-Distance': avg_results[method]['w_distance'],
        'Mean Diff': avg_results[method]['mean_diff'],
        'Std Diff': avg_results[method]['std_diff']
    })

summary_df = pd.DataFrame(summary_data)
```

```
<Figure size 1500x1000 with 0 Axes>
```



For our small data set, using a big bin count still worked out! We'll use 25 bins in our final stratification.

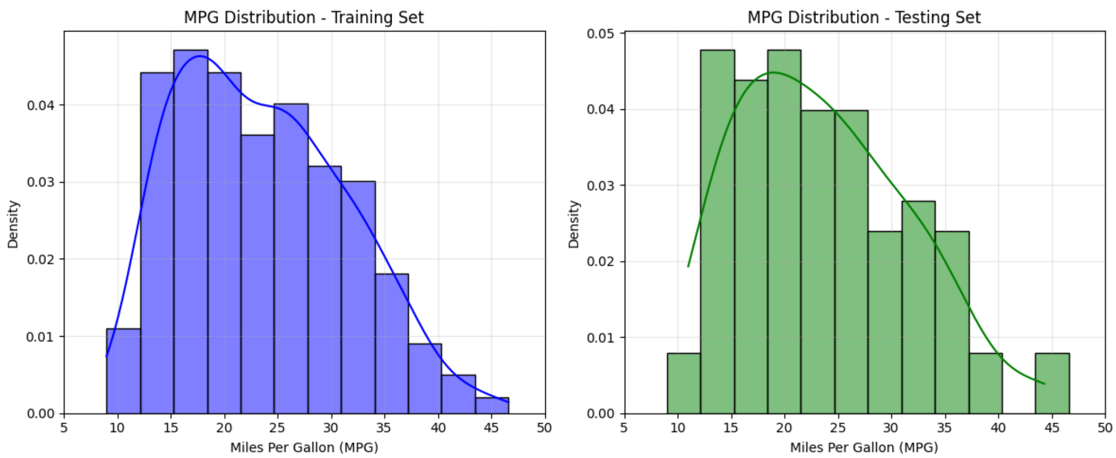
A couple notes: - The 15 bin and 25 bin distributions were mapped to 12 bin diagrams for easy comparison across splits. - Bin Counts over 25, some bins only had a single sample and thus couldn't be stratified. 25 turned out to be the max we could support, which conveniently does not have performance implications for our data set.

4.1.5 Stratified Test Train Split We are finally ready to split the data! We will split it across 25 bins for mpg to keep the distribution equal.

[30]:
```python
# 4.1.5 Test Train Split using MPG stratification with 25 bins

# Original data
y = df['mpg']

# Prepare features
make_encoded = pd.get_dummies(df['make'], prefix='make', drop_first=True)
X = pd.concat([df[['weight', 'model_year']], make_encoded], axis=1)

# Create 25 mpg bins for stratification (our optimal bin count from earlier␣
↪analysis)
mpg_bins = pd.qcut(df['mpg'], q=25, labels=False, duplicates='drop')

# Split the data with stratification
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=state_setter,
    stratify=mpg_bins
)

# Display the shapes of our training and testing sets
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}")

# Visualize our final train/test distributions with the same number of bins
plt.figure(figsize=(12, 5))

# Define a fixed set of bins for both plots - using exactly 12 bins
fixed_bins = np.linspace(y.min(), y.max(), 13)  # 13 edges = 12 bins

plt.subplot(1, 2, 1)
sns.histplot(y_train, bins=fixed_bins, kde=True, color='blue', stat='density')
plt.title('MPG Distribution - Training Set')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')
plt.grid(True, alpha=0.3)
plt.xlim(5, 50)

plt.subplot(1, 2, 2)
sns.histplot(y_test, bins=fixed_bins, kde=True, color='green', stat='density')
plt.title('MPG Distribution - Testing Set')
plt.xlabel('Miles Per Gallon (MPG)')
plt.ylabel('Density')
plt.grid(True, alpha=0.3)
```

```
plt.xlim(5, 50)

plt.tight_layout()
plt.show()

# Compare statistical measures to verify balance
print("\nTraining set MPG statistics:")
print(y_train.describe())
print("\nTesting set MPG statistics:")
print(y_test.describe())

# Calculate distribution similarity score between original and test set
similarity_score = compare_distributions(y, y_test, "Final Split")
print(f"\nFinal test set similarity score: {similarity_score['composite_score']:
    ↪.4f}")
```

```
X_train shape: (318, 30)
X_test shape: (80, 30)
y_train shape: (318,)
y_test shape: (80,)
```




```
Training set MPG statistics:
count    318.00000
mean      23.50566
std        7.80725
min        9.00000
25%       17.50000
50%       23.00000
75%       29.00000
max       46.60000
Name: mpg, dtype: float64
```

```
Testing set MPG statistics:
count    80.000000
mean     23.550000
std       7.899928
min      11.000000
25%      17.750000
50%      22.750000
75%      29.125000
max      44.300000
Name: mpg, dtype: float64

Final test set similarity score: 0.1176
```

1.6.2 4.2 Train model using Scikit-Learn model.fit() method

[31]:

```python
# 4.2 Train model using Scikit-Learn's Linear Regression

# Initialize the linear regression model
lr_model = LinearRegression()

# Train the model on the training data
start_time = time.time()
lr_model.fit(X_train, y_train)
training_time = time.time() - start_time

print(f"Model trained in {training_time:.4f} seconds")

# Get the coefficients and intercept
print("\nModel Parameters:")
print(f"Intercept: {lr_model.intercept_:.4f}")

# Display some of the most influential coefficients
coefficients = pd.DataFrame({
    'Feature': X_train.columns,
    'Coefficient': lr_model.coef_
})

# Sort coefficients by absolute value (to find most influential features)
coefficients['Abs_Coefficient'] = np.abs(coefficients['Coefficient'])
sorted_coeffs = coefficients.sort_values('Abs_Coefficient', ascending=False)

print("\nTop 10 most influential features:")
display(sorted_coeffs.head(10))

# Make predictions on the training and test sets
y_train_pred = lr_model.predict(X_train)
```

```python
y_test_pred = lr_model.predict(X_test)

# Create a dataframe to store actual vs predicted values for the test set
results_df = pd.DataFrame({
    'Actual': y_test,
    'Predicted': y_test_pred,
    'Residual': y_test - y_test_pred
})

# Display the first few rows of predictions vs. actual values
print("\nSample of Actual vs. Predicted values (Test Set):")
display(results_df.head(10))

# Plot actual vs predicted for test set
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_test_pred, alpha=0.7)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
plt.xlabel('Actual MPG')
plt.ylabel('Predicted MPG')
plt.title('Actual vs. Predicted MPG (Test Set)')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

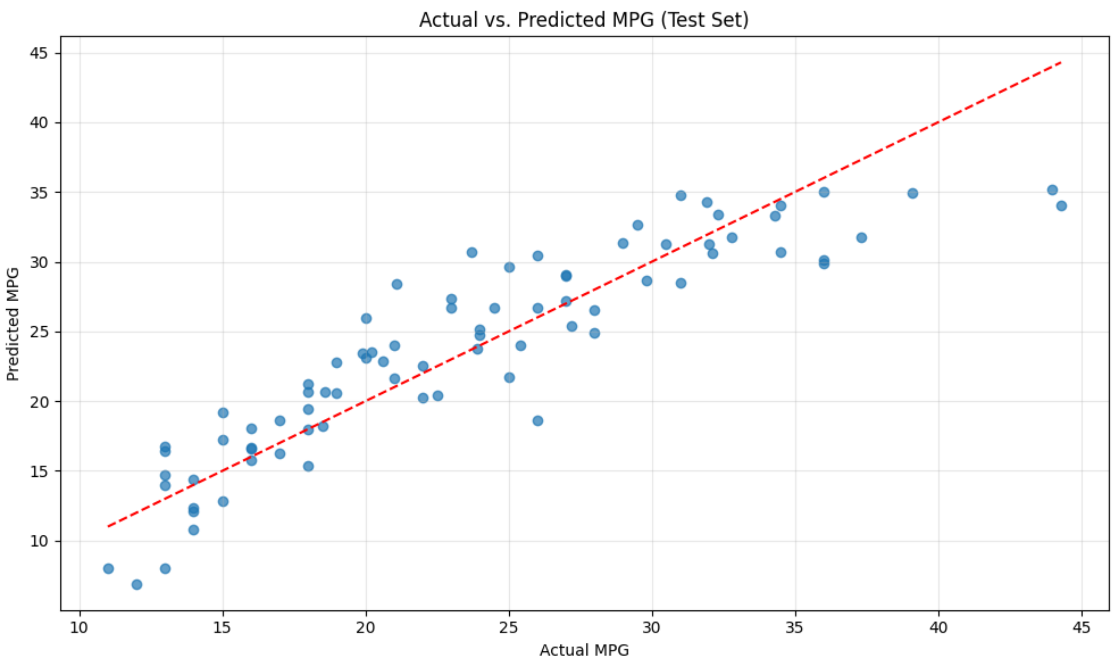
```
Model trained in 0.0048 seconds

Model Parameters:
Intercept: -15.6403

Top 10 most influential features:
```

	Feature	Coefficient	Abs_Coefficient
27	make_triumph	8.295360	8.295360
23	make_renault	5.584860	5.584860
13	make_honda	5.196054	5.196054
8	make_datsun	5.127233	5.127233
28	make_volkswagen	4.894003	4.894003
2	make_audi	4.737821	4.737821
5	make_cadillac	4.445454	4.445454
15	make_mercedes-benz	4.040828	4.040828
10	make_fiat	3.609072	3.609072
22	make_pontiac	3.556101	3.556101

```
Sample of Actual vs. Predicted values (Test Set):
```

	Actual	Predicted	Residual
171	24.0	25.141424	-1.141424
142	26.0	30.472593	-4.472593

```
304    37.3  31.771171  5.528829
67     11.0   8.055217  2.944783
311    32.1  30.619889  1.480111
87     13.0  14.675906 -1.675906
292    18.5  18.248122  0.251878
43     13.0   8.054850  4.945150
75     14.0  14.384008 -0.384008
0      18.0  15.376790  2.623210
```



1.6.3 4.3 Evalulate performance

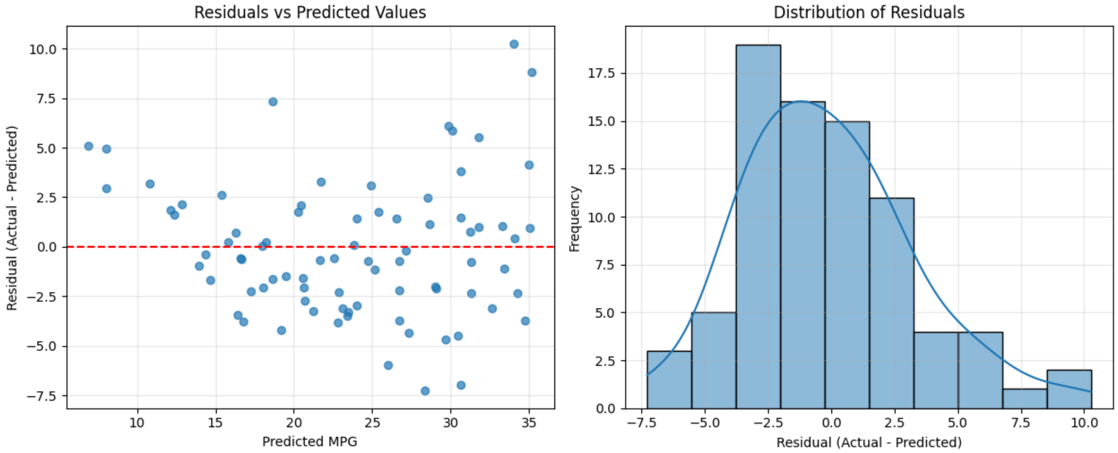
4.3.1 Regression: R^2, MAE, RMSE

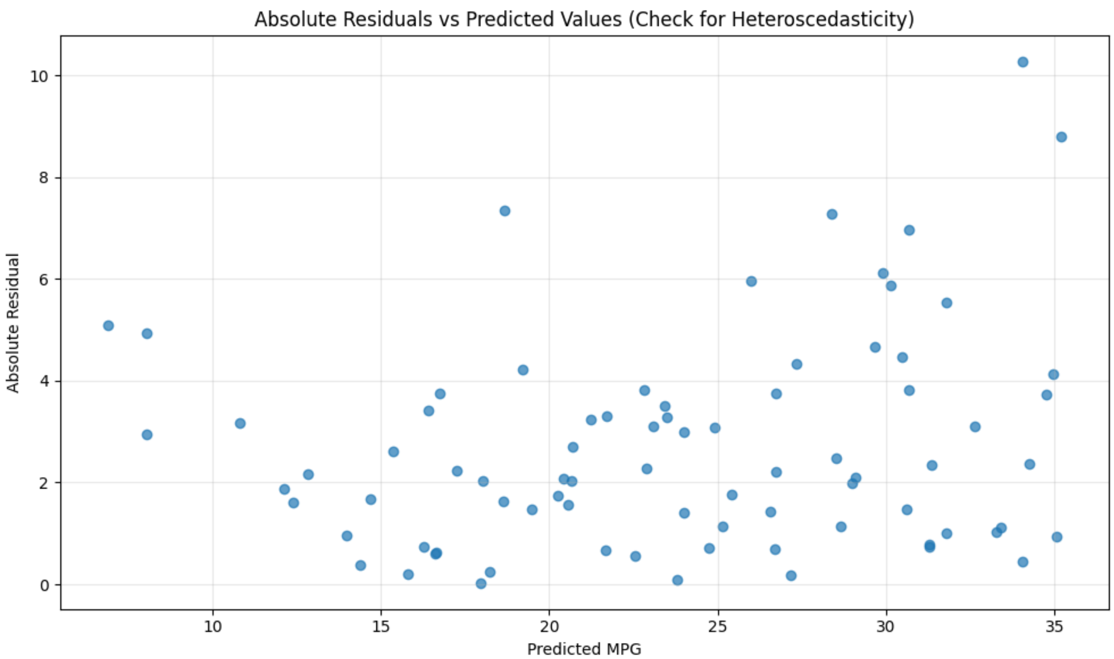
[32]:
```
# 4.3.1 Evaluate performance with R^2, MAE, and RMSE

# Function to calculate and display performance metrics
def evaluate_model(y_true, y_pred, dataset_name):
    # Calculate metrics
    r2 = r2_score(y_true, y_pred)
    mae = mean_absolute_error(y_true, y_pred)
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))

    # Calculate additional metrics
    # Mean Absolute Percentage Error (MAPE)
    mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100

    # Median Absolute Error
```

```
    med_ae = np.median(np.abs(y_true - y_pred))

    # Print metrics
    print(f"\n--- {dataset_name} Metrics ---")
    print(f"R² Score: {r2:.4f}")
    print(f"Mean Absolute Error (MAE): {mae:.4f}")
    print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")
    print(f"Mean Absolute Percentage Error (MAPE): {mape:.2f}%")
    print(f"Median Absolute Error: {med_ae:.4f}")

    return {
        'R2': r2,
        'MAE': mae,
        'RMSE': rmse,
        'MAPE': mape,
        'MedianAE': med_ae
    }

# Evaluate on training set
train_metrics = evaluate_model(y_train, y_train_pred, "Training Set")

# Evaluate on test set
test_metrics = evaluate_model(y_test, y_test_pred, "Test Set")

# Compare training vs test metrics to check for overfitting
metrics_comparison = pd.DataFrame({
    'Training': [train_metrics['R2'], train_metrics['MAE'], train_metrics['RMSE']],
    'Testing': [test_metrics['R2'], test_metrics['MAE'], test_metrics['RMSE']]
}, index=['R² Score', 'Mean Absolute Error (MAE)', 'Root Mean Squared Error (RMSE)'])

print("\n--- Training vs Testing Performance ---")
display(metrics_comparison)

# Visualize residuals
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.scatter(y_test_pred, results_df['Residual'], alpha=0.7)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel('Predicted MPG')
plt.ylabel('Residual (Actual - Predicted)')
plt.title('Residuals vs Predicted Values')
plt.grid(True, alpha=0.3)

plt.subplot(1, 2, 2)
```

```python
sns.histplot(results_df['Residual'], kde=True)
plt.xlabel('Residual (Actual - Predicted)')
plt.ylabel('Frequency')
plt.title('Distribution of Residuals')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Check for heteroscedasticity (if variance of residuals changes with predicted␣
↪values)
plt.figure(figsize=(10, 6))
plt.scatter(y_test_pred, np.abs(results_df['Residual']), alpha=0.7)
plt.xlabel('Predicted MPG')
plt.ylabel('Absolute Residual')
plt.title('Absolute Residuals vs Predicted Values (Check for␣
↪Heteroscedasticity)')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

```
--- Training Set Metrics ---
R² Score: 0.8391
Mean Absolute Error (MAE): 2.3302
Root Mean Squared Error (RMSE): 3.1269
Mean Absolute Percentage Error (MAPE): 10.58%
Median Absolute Error: 1.7095

--- Test Set Metrics ---
R² Score: 0.8134
Mean Absolute Error (MAE): 2.6810
Root Mean Squared Error (RMSE): 3.3907
Mean Absolute Percentage Error (MAPE): 12.21%
Median Absolute Error: 2.1878

--- Training vs Testing Performance ---

                                Training   Testing
R² Score                        0.839085  0.813446
Mean Absolute Error (MAE)       2.330232  2.681005
Root Mean Squared Error (RMSE)  3.126887  3.390734
```





1.6.4 Reflection 4: How well did the model perform? Any surprises in the results?

1.7 Section 5. Improve the Model or Try Alternates (Implement Pipelines)

1.7.1 5.1 Implement Pipeline 1: Imputer → StandardScaler → Linear Regression.

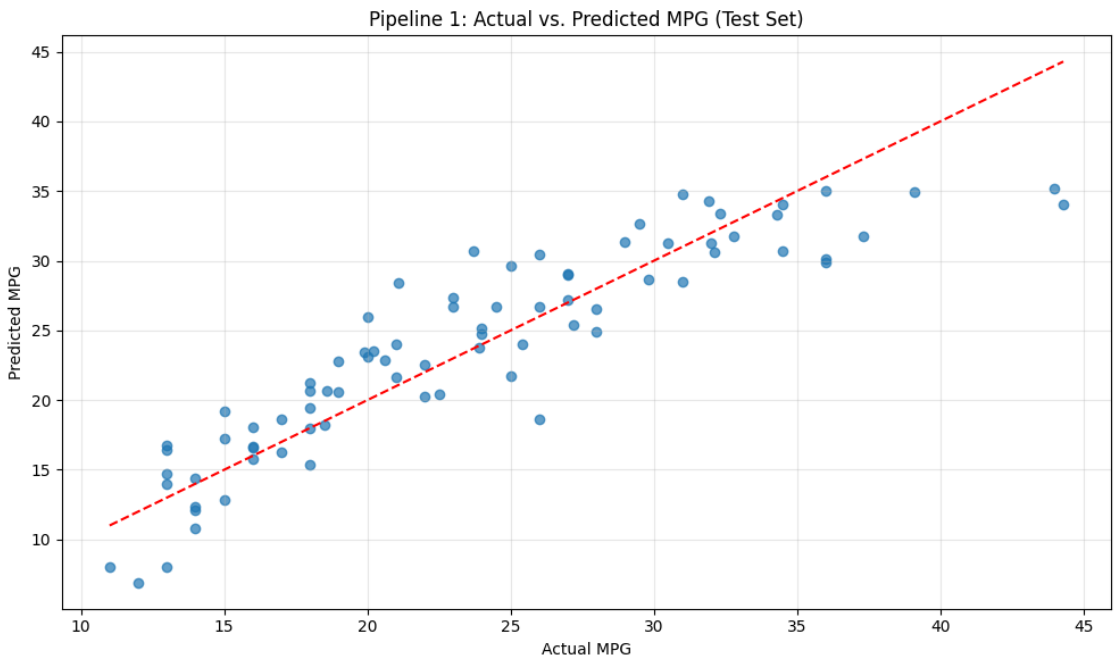
[33]:
```
# 5.1 Implement Pipeline 1: Imputer → StandardScaler → Linear Regression

print("### 5.1 Pipeline 1: Imputer → StandardScaler → Linear Regression ###")
```

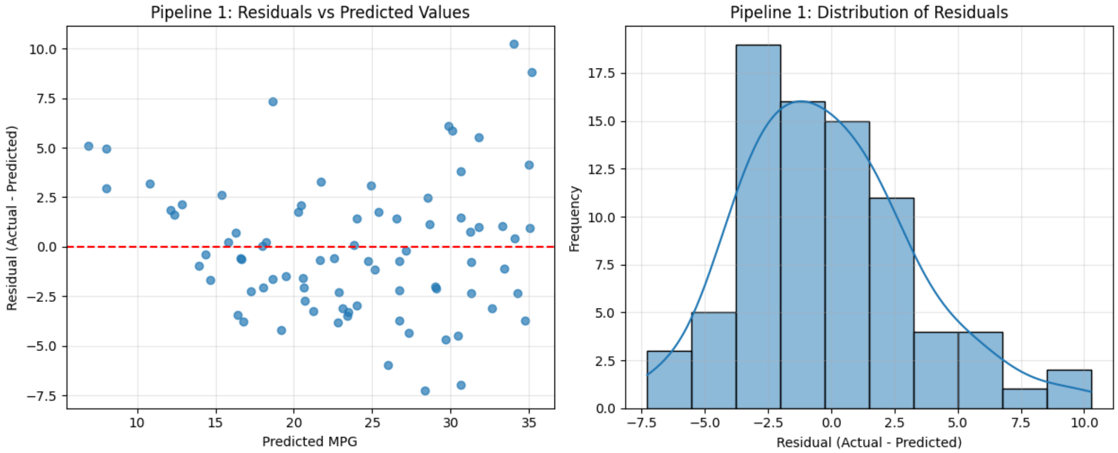
```
# Create a pipeline with imputer, scaler, and linear regression
pipeline1 = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler()),
    ('regressor', LinearRegression())
])

# Train the pipeline on the training data
start_time = time.time()
pipeline1.fit(X_train, y_train)
pipeline1_training_time = time.time() - start_time

print(f"Pipeline 1 trained in {pipeline1_training_time:.4f} seconds")

# Make predictions on training and test sets
y_train_pred_pipeline1 = pipeline1.predict(X_train)
y_test_pred_pipeline1 = pipeline1.predict(X_test)

# Evaluate pipeline1 performance
train_metrics_pipeline1 = evaluate_model(y_train, y_train_pred_pipeline1,␣
 ↪"Pipeline 1 (Training Set)")
test_metrics_pipeline1 = evaluate_model(y_test, y_test_pred_pipeline1,␣
 ↪"Pipeline 1 (Test Set)")

# Get the linear regression model from the pipeline
linear_model = pipeline1.named_steps['regressor']

# Display some information about the model
print("\nPipeline 1 Model Parameters:")
print(f"Intercept: {linear_model.intercept_:.4f}")

# Display top coefficients (note: these are scaled coefficients now)
coefficients_pipeline1 = pd.DataFrame({
    'Feature': X_train.columns,
    'Coefficient': linear_model.coef_
})

# Sort coefficients by absolute value
coefficients_pipeline1['Abs_Coefficient'] = np.
 ↪abs(coefficients_pipeline1['Coefficient'])
sorted_coeffs_pipeline1 = coefficients_pipeline1.sort_values('Abs_Coefficient',␣
 ↪ascending=False)

print("\nTop 10 most influential features (after scaling):")
display(sorted_coeffs_pipeline1.head(10))

# Visualize actual vs predicted for test set
```

```python
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_test_pred_pipeline1, alpha=0.7)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
plt.xlabel('Actual MPG')
plt.ylabel('Predicted MPG')
plt.title('Pipeline 1: Actual vs. Predicted MPG (Test Set)')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

# Create a dataframe to store actual vs predicted values for the test set
results_df_pipeline1 = pd.DataFrame({
    'Actual': y_test,
    'Predicted': y_test_pred_pipeline1,
    'Residual': y_test - y_test_pred_pipeline1
})

# Visualize residuals
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.scatter(y_test_pred_pipeline1, results_df_pipeline1['Residual'], alpha=0.7)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel('Predicted MPG')
plt.ylabel('Residual (Actual - Predicted)')
plt.title('Pipeline 1: Residuals vs Predicted Values')
plt.grid(True, alpha=0.3)

plt.subplot(1, 2, 2)
sns.histplot(results_df_pipeline1['Residual'], kde=True)
plt.xlabel('Residual (Actual - Predicted)')
plt.ylabel('Frequency')
plt.title('Pipeline 1: Distribution of Residuals')
plt.grid(True, alpha=0.3)

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

```
### 5.1 Pipeline 1: Imputer → StandardScaler → Linear Regression ###
Pipeline 1 trained in 0.0066 seconds

--- Pipeline 1 (Training Set) Metrics ---
R² Score: 0.8391
Mean Absolute Error (MAE): 2.3302
Root Mean Squared Error (RMSE): 3.1269
Mean Absolute Percentage Error (MAPE): 10.58%
Median Absolute Error: 1.7095
```

```
--- Pipeline 1 (Test Set) Metrics ---
R² Score: 0.8134
Mean Absolute Error (MAE): 2.6810
Root Mean Squared Error (RMSE): 3.3907
Mean Absolute Percentage Error (MAPE): 12.21%
Median Absolute Error: 2.1878

Pipeline 1 Model Parameters:
Intercept: 23.5057

Top 10 most influential features (after scaling):
            Feature  Coefficient  Abs_Coefficient
0            weight    -5.005370         5.005370
1        model_year     2.627028         2.627028
8       make_datsun     1.328100         1.328100
28  make_volkswagen     0.969078         0.969078
13       make_honda     0.949537         0.949537
26      make_toyota     0.790859         0.790859
21    make_plymouth     0.732874         0.732874
22     make_pontiac     0.729538         0.729538
23     make_renault     0.622415         0.622415
14       make_mazda     0.580426         0.580426
```





1.7.2 5.2 Implement Pipeline 2: Imputer → Polynomial Features (degree=3) → StandardScaler → Linear Regression.

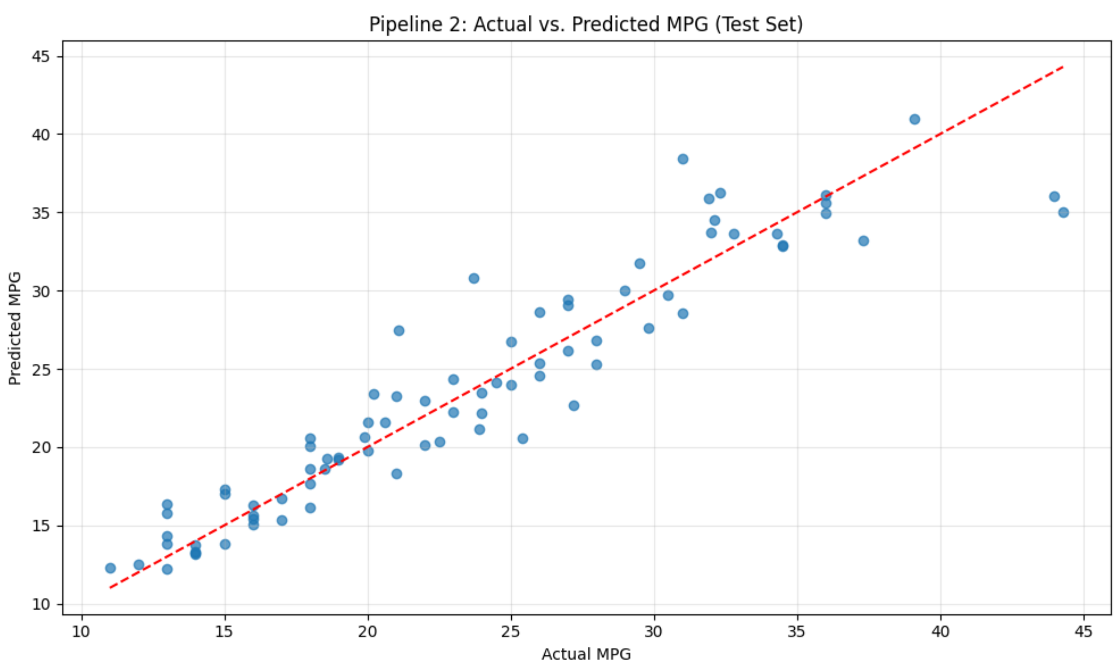
[ ]:
```
# 5.2 Implement Pipeline 2: Imputer → Polynomial Features (degree=3) →␣
↪StandardScaler → Linear Regression

print("\n### 5.2 Pipeline 2: Imputer → Polynomial Features (degree=3) →␣
↪StandardScaler → Linear Regression ###")

# Create a pipeline with imputer, polynomial features, scaler, and linear␣
↪regression
pipeline2 = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('poly', PolynomialFeatures(degree=3, include_bias=False)),
    ('scaler', StandardScaler()),
    ('regressor', LinearRegression())
])

# Use all features including make dummies
# No need to restrict to only numerical features since you mentioned your␣
↪processor can handle it
X_train_full = X_train.copy()  # Use all features
X_test_full = X_test.copy()    # Use all features

# Print feature count to understand the scale
print(f"Number of input features: {X_train_full.shape[1]}")
print(f"Expected polynomial features (degree=3): {int((X_train_full.shape[1] +␣
↪3) * (X_train_full.shape[1] + 2) * (X_train_full.shape[1] + 1) / 6) - 1}")

# Train the pipeline on the training data with all features
```

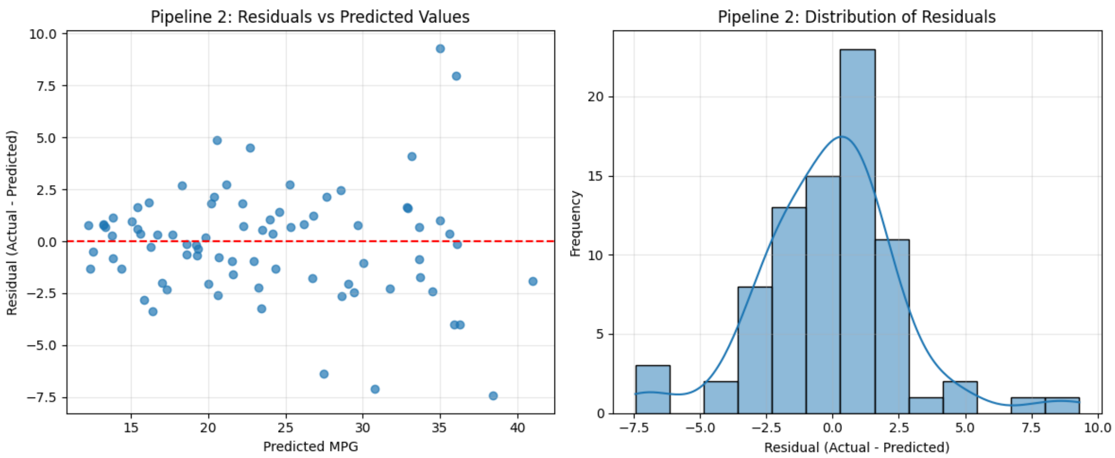
```python
start_time = time.time()
pipeline2.fit(X_train_full, y_train)
pipeline2_training_time = time.time() - start_time

print(f"Pipeline 2 trained in {pipeline2_training_time:.4f} seconds")

# Make predictions on training and test sets
y_train_pred_pipeline2 = pipeline2.predict(X_train_full)
y_test_pred_pipeline2 = pipeline2.predict(X_test_full)

# Evaluate pipeline2 performance
train_metrics_pipeline2 = evaluate_model(y_train, y_train_pred_pipeline2,␣
 ↪"Pipeline 2 (Training Set)")
test_metrics_pipeline2 = evaluate_model(y_test, y_test_pred_pipeline2,␣
 ↪"Pipeline 2 (Test Set)")

# Get all feature names for reference
# Since we're using all features, we need to get all column names
feature_names = X_train_full.columns.tolist()

# Get the polynomial feature names
poly_features = pipeline2.named_steps['poly'].
 ↪get_feature_names_out(feature_names)

# Get the linear regression model from the pipeline
linear_model2 = pipeline2.named_steps['regressor']

# Display model information
print("\nPipeline 2 Model Parameters:")
print(f"Intercept: {linear_model2.intercept_:.4f}")
print(f"Number of features after polynomial transformation:␣
 ↪{len(poly_features)}")

# Display top coefficients
coefficients_pipeline2 = pd.DataFrame({
    'Feature': poly_features,
    'Coefficient': linear_model2.coef_
})

# Sort coefficients by absolute value
coefficients_pipeline2['Abs_Coefficient'] = np.
 ↪abs(coefficients_pipeline2['Coefficient'])
sorted_coeffs_pipeline2 = coefficients_pipeline2.sort_values('Abs_Coefficient',␣
 ↪ascending=False)

print("\nTop 10 most influential polynomial features (after scaling):")
```

```python
display(sorted_coeffs_pipeline2.head(10))

# Visualize actual vs predicted for test set
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_test_pred_pipeline2, alpha=0.7)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
plt.xlabel('Actual MPG')
plt.ylabel('Predicted MPG')
plt.title('Pipeline 2: Actual vs. Predicted MPG (Test Set)')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

# Create a dataframe to store actual vs predicted values for the test set
results_df_pipeline2 = pd.DataFrame({
    'Actual': y_test,
    'Predicted': y_test_pred_pipeline2,
    'Residual': y_test - y_test_pred_pipeline2
})

# Visualize residuals
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.scatter(y_test_pred_pipeline2, results_df_pipeline2['Residual'], alpha=0.7)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel('Predicted MPG')
plt.ylabel('Residual (Actual - Predicted)')
plt.title('Pipeline 2: Residuals vs Predicted Values')
plt.grid(True, alpha=0.3)

plt.subplot(1, 2, 2)
sns.histplot(results_df_pipeline2['Residual'], kde=True)
plt.xlabel('Residual (Actual - Predicted)')
plt.ylabel('Frequency')
plt.title('Pipeline 2: Distribution of Residuals')
plt.grid(True, alpha=0.3)

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

```
### 5.2 Pipeline 2: Imputer → Polynomial Features (degree=3) → StandardScaler →
Linear Regression ###
Pipeline 2 trained in 0.0065 seconds

--- Pipeline 2 (Training Set) Metrics ---
R² Score: 0.8641
```

```
Mean Absolute Error (MAE): 2.0394
Root Mean Squared Error (RMSE): 2.8732
Mean Absolute Percentage Error (MAPE): 8.85%
Median Absolute Error: 1.5128

--- Pipeline 2 (Test Set) Metrics ---
R² Score: 0.8852
Mean Absolute Error (MAE): 1.9200
Root Mean Squared Error (RMSE): 2.6601
Mean Absolute Percentage Error (MAPE): 8.11%
Median Absolute Error: 1.3850

Pipeline 2 Model Parameters:
Intercept: 23.5057
Number of features after polynomial transformation: 9

Top 10 most influential polynomial features (after scaling):
               Feature  Coefficient  Abs_Coefficient
4         model_year^2  1939.275717      1939.275717
1           model_year  -993.425044       993.425044
8         model_year^3  -939.229965       939.229965
3    weight model_year   202.428275       202.428275
7  weight model_year^2  -115.576841       115.576841
0               weight  -110.444340       110.444340
6  weight^2 model_year    21.550534        21.550534
5             weight^3    -4.455134         4.455134
2             weight^2    -1.937355         1.937355
```





1.7.3 5.3 Compare performance of all models across the same performance metrics

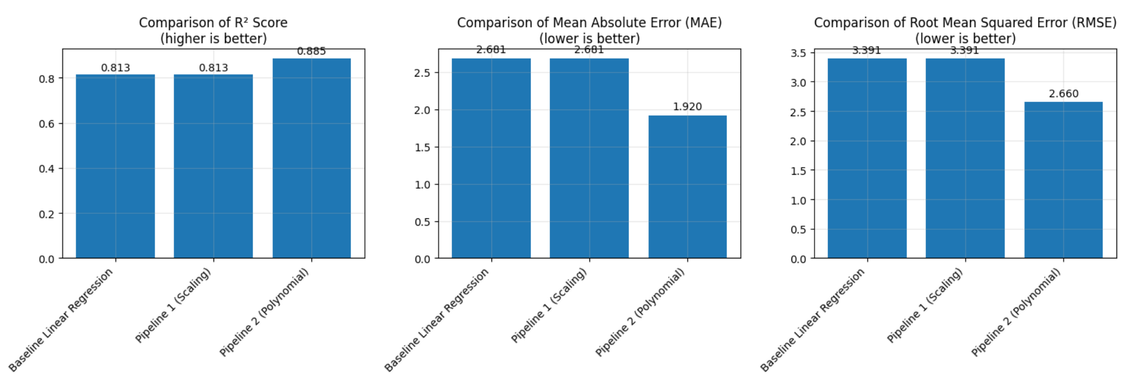
[35]:
```python
# 5.3 Compare performance of all models across the same performance metrics

print("\n### 5.3 Compare Performance of All Models ###")

# Create a comparison dataframe for test metrics
models_comparison = pd.DataFrame({
    'Baseline Linear Regression': [test_metrics['R2'], test_metrics['MAE'], test_metrics['RMSE'], test_metrics['MAPE']],
    'Pipeline 1 (Scaling)': [test_metrics_pipeline1['R2'], test_metrics_pipeline1['MAE'], test_metrics_pipeline1['RMSE'], test_metrics_pipeline1['MAPE']],
    'Pipeline 2 (Polynomial)': [test_metrics_pipeline2['R2'], test_metrics_pipeline2['MAE'], test_metrics_pipeline2['RMSE'], test_metrics_pipeline2['MAPE']]
}, index=['R² Score', 'Mean Absolute Error (MAE)', 'Root Mean Squared Error (RMSE)', 'Mean Absolute Percentage Error (MAPE)'])

print("\n--- Test Set Performance Comparison ---")
display(models_comparison)

# Training time comparison
training_times = pd.DataFrame({
    'Model': ['Baseline Linear Regression', 'Pipeline 1 (Scaling)', 'Pipeline 2 (Polynomial)'],
    'Training Time (seconds)': [training_time, pipeline1_training_time, pipeline2_training_time]
```

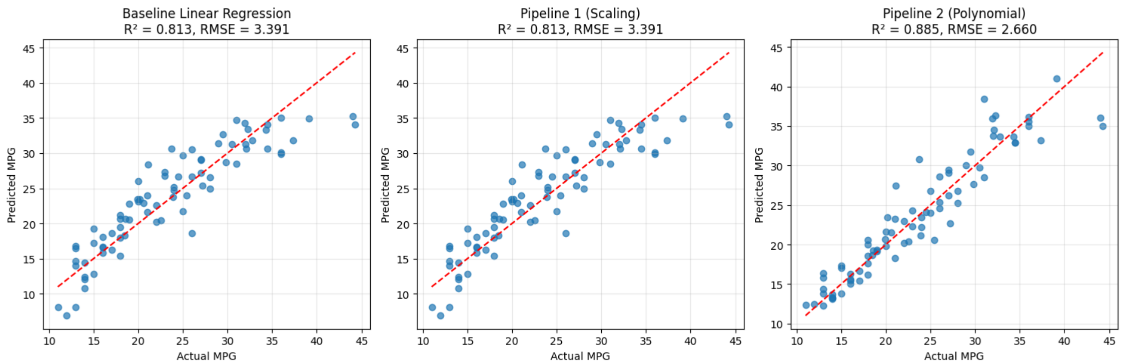
```
})

print("\n--- Training Time Comparison ---")
display(training_times)

# Visualize performance comparison - fixed version
metrics_to_plot = ['R² Score', 'Mean Absolute Error (MAE)', 'Root Mean Squared␣
↪Error (RMSE)']
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# Plot each metric
for i, metric in enumerate(metrics_to_plot):
    values = models_comparison.loc[metric].values
    model_names = models_comparison.columns

    # For R², higher is better
    if metric == 'R² Score':
        axes[i].bar(range(len(model_names)), values)
        axes[i].set_title(f'Comparison of {metric}\n(higher is better)')
    # For error metrics, lower is better
    else:
        axes[i].bar(range(len(model_names)), values)
        axes[i].set_title(f'Comparison of {metric}\n(lower is better)')

    axes[i].set_xticks(range(len(model_names)))
    axes[i].set_xticklabels(model_names, rotation=45, ha='right')
    axes[i].grid(True, alpha=0.3)

    # Add value labels on top of bars
    for j, v in enumerate(values):
        axes[i].text(j, v + (0.01 if metric == 'R² Score' else 0.05),
                    f'{v:.3f}', ha='center', va='bottom')

plt.tight_layout()
plt.show()

# Create a scatter plot of all models predictions vs actual values
plt.figure(figsize=(15, 5))

# Base model
plt.subplot(1, 3, 1)
plt.scatter(y_test, y_test_pred, alpha=0.7)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
plt.xlabel('Actual MPG')
plt.ylabel('Predicted MPG')
plt.title(f'Baseline Linear Regression\nR² = {test_metrics["R2"]:.3f}, RMSE =␣
↪{test_metrics["RMSE"]:.3f}')
```

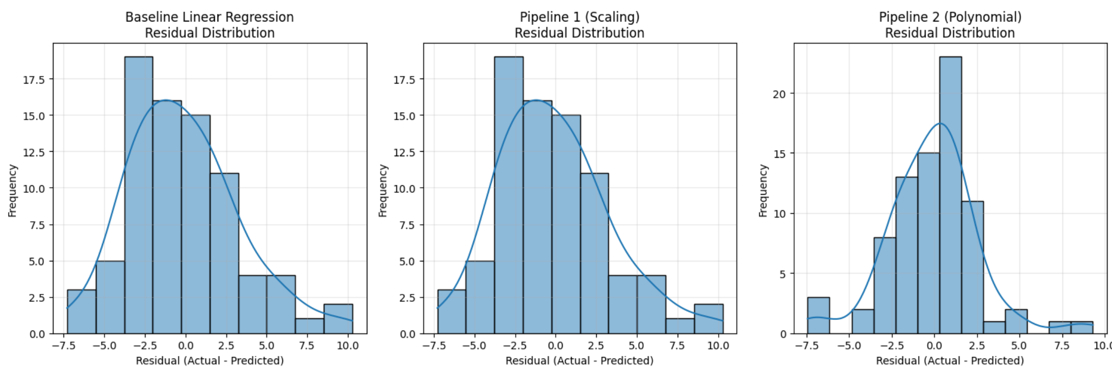
```
plt.grid(True, alpha=0.3)

# Pipeline 1
plt.subplot(1, 3, 2)
plt.scatter(y_test, y_test_pred_pipeline1, alpha=0.7)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
plt.xlabel('Actual MPG')
plt.ylabel('Predicted MPG')
plt.title(f'Pipeline 1 (Scaling)\nR² = {test_metrics_pipeline1["R2"]:.3f}, RMSE = {test_metrics_pipeline1["RMSE"]:.3f}')
plt.grid(True, alpha=0.3)

# Pipeline 2
plt.subplot(1, 3, 3)
plt.scatter(y_test, y_test_pred_pipeline2, alpha=0.7)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
plt.xlabel('Actual MPG')
plt.ylabel('Predicted MPG')
plt.title(f'Pipeline 2 (Polynomial)\nR² = {test_metrics_pipeline2["R2"]:.3f}, RMSE = {test_metrics_pipeline2["RMSE"]:.3f}')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Residual comparison
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# Base model residuals
sns.histplot(y_test - y_test_pred, kde=True, ax=axes[0])
axes[0].set_xlabel('Residual (Actual - Predicted)')
axes[0].set_ylabel('Frequency')
axes[0].set_title('Baseline Linear Regression\nResidual Distribution')
axes[0].grid(True, alpha=0.3)

# Pipeline 1 residuals
sns.histplot(y_test - y_test_pred_pipeline1, kde=True, ax=axes[1])
axes[1].set_xlabel('Residual (Actual - Predicted)')
axes[1].set_ylabel('Frequency')
axes[1].set_title('Pipeline 1 (Scaling)\nResidual Distribution')
axes[1].grid(True, alpha=0.3)

# Pipeline 2 residuals
sns.histplot(y_test - y_test_pred_pipeline2, kde=True, ax=axes[2])
axes[2].set_xlabel('Residual (Actual - Predicted)')
axes[2].set_ylabel('Frequency')
axes[2].set_title('Pipeline 2 (Polynomial)\nResidual Distribution')
```

```python
axes[2].grid(True, alpha=0.3)

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

```
### 5.3 Compare Performance of All Models ###

--- Test Set Performance Comparison ---
                                       Baseline Linear Regression  \
R² Score                                                 0.813446
Mean Absolute Error (MAE)                                2.681005
Root Mean Squared Error (RMSE)                           3.390734
Mean Absolute Percentage Error (MAPE)                   12.206060

                                       Pipeline 1 (Scaling)  \
R² Score                                           0.813446
Mean Absolute Error (MAE)                          2.681005
Root Mean Squared Error (RMSE)                     3.390734
Mean Absolute Percentage Error (MAPE)             12.206060

                                       Pipeline 2 (Polynomial)
R² Score                                              0.885180
Mean Absolute Error (MAE)                             1.920014
Root Mean Squared Error (RMSE)                        2.660116
Mean Absolute Percentage Error (MAPE)                 8.114776

--- Training Time Comparison ---
                        Model  Training Time (seconds)
0  Baseline Linear Regression                 0.004824
1        Pipeline 1 (Scaling)                 0.006558
2     Pipeline 2 (Polynomial)                 0.006480
```







5.3.1 Polynomial optimization

[36]:
```
# 5.3.1 Optimizing Polynomial Degree (Testing degrees 3, 4, 6, and 9)

print("\n### 5.3.1 Optimizing Polynomial Degree ###")

# Degrees to test
poly_degrees = [3, 4, 6, 9]

# Dictionary to store results for each degree
poly_results = {}
poly_train_metrics = {}
poly_test_metrics = {}
poly_training_times = {}
poly_predictions = {}

# Only use numerical features to avoid feature explosion
X_train_numeric = X_train[['weight', 'model_year']]
X_test_numeric = X_test[['weight', 'model_year']]

# Test each polynomial degree
```

```
for degree in poly_degrees:
    print(f"\nTesting Polynomial Degree {degree}")

    # Create pipeline with the current degree
    poly_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy='median')),
        ('poly', PolynomialFeatures(degree=degree, include_bias=False)),
        ('scaler', StandardScaler()),  # Important to scale after polynomial␣
↪transformation
        ('regressor', LinearRegression())
    ])

    # Train the pipeline
    start_time = time.time()
    poly_pipeline.fit(X_train_numeric, y_train)
    train_time = time.time() - start_time
    poly_training_times[degree] = train_time

    print(f"Training time: {train_time:.4f} seconds")

    # Get the polynomial feature names
    poly_features = poly_pipeline.named_steps['poly'].
↪get_feature_names_out(['weight', 'model_year'])
    print(f"Number of features after polynomial transformation:␣
↪{len(poly_features)}")

    # Make predictions
    y_train_pred = poly_pipeline.predict(X_train_numeric)
    y_test_pred = poly_pipeline.predict(X_test_numeric)

    # Store predictions for later visualization
    poly_predictions[degree] = {
        'train': y_train_pred,
        'test': y_test_pred
    }

    # Evaluate performance
    train_metrics = evaluate_model(y_train, y_train_pred, f"Degree {degree}␣
↪(Training Set)")
    test_metrics = evaluate_model(y_test, y_test_pred, f"Degree {degree} (Test␣
↪Set)")

    # Store results
    poly_results[degree] = poly_pipeline
    poly_train_metrics[degree] = train_metrics
    poly_test_metrics[degree] = test_metrics
```

```
### 5.3.1 Optimizing Polynomial Degree ###

Testing Polynomial Degree 3
Training time: 0.0038 seconds
Number of features after polynomial transformation: 9

--- Degree 3 (Training Set) Metrics ---
R² Score: 0.8641
Mean Absolute Error (MAE): 2.0394
Root Mean Squared Error (RMSE): 2.8732
Mean Absolute Percentage Error (MAPE): 8.85%
Median Absolute Error: 1.5128

--- Degree 3 (Test Set) Metrics ---
R² Score: 0.8852
Mean Absolute Error (MAE): 1.9200
Root Mean Squared Error (RMSE): 2.6601
Mean Absolute Percentage Error (MAPE): 8.11%
Median Absolute Error: 1.3850

Testing Polynomial Degree 4
Training time: 0.0032 seconds
Number of features after polynomial transformation: 14

--- Degree 4 (Training Set) Metrics ---
R² Score: 0.8704
Mean Absolute Error (MAE): 2.0598
Root Mean Squared Error (RMSE): 2.8059
Mean Absolute Percentage Error (MAPE): 8.95%
Median Absolute Error: 1.5475

--- Degree 4 (Test Set) Metrics ---
R² Score: 0.8825
Mean Absolute Error (MAE): 1.9355
Root Mean Squared Error (RMSE): 2.6910
Mean Absolute Percentage Error (MAPE): 8.18%
Median Absolute Error: 1.5177

Testing Polynomial Degree 6
Training time: 0.0044 seconds
Number of features after polynomial transformation: 27

--- Degree 6 (Training Set) Metrics ---
R² Score: 0.8759
Mean Absolute Error (MAE): 2.0216
Root Mean Squared Error (RMSE): 2.7461
Mean Absolute Percentage Error (MAPE): 8.75%
```

```
Median Absolute Error: 1.5371

--- Degree 6 (Test Set) Metrics ---
R² Score: 0.8764
Mean Absolute Error (MAE): 1.9547
Root Mean Squared Error (RMSE): 2.7598
Mean Absolute Percentage Error (MAPE): 8.27%
Median Absolute Error: 1.5049

Testing Polynomial Degree 9
Training time: 0.0088 seconds
Number of features after polynomial transformation: 54

--- Degree 9 (Training Set) Metrics ---
R² Score: 0.8949
Mean Absolute Error (MAE): 1.8234
Root Mean Squared Error (RMSE): 2.5270
Mean Absolute Percentage Error (MAPE): 7.82%
Median Absolute Error: 1.3453

--- Degree 9 (Test Set) Metrics ---
R² Score: 0.8741
Mean Absolute Error (MAE): 1.9616
Root Mean Squared Error (RMSE): 2.7854
Mean Absolute Percentage Error (MAPE): 8.27%
Median Absolute Error: 1.2692
```

[37]:
```python
# Create a comparison dataframe of test metrics
poly_comparison = pd.DataFrame({
    f'Degree {degree}': [
        poly_test_metrics[degree]['R2'],
        poly_test_metrics[degree]['MAE'],
        poly_test_metrics[degree]['RMSE'],
        poly_test_metrics[degree]['MAPE'],
        poly_training_times[degree]
    ] for degree in poly_degrees
}, index=['R² Score', 'Mean Absolute Error (MAE)', 'Root Mean Squared Error␣
↪(RMSE)',
          'Mean Absolute Percentage Error (MAPE)', 'Training Time (seconds)'])

print("\n--- Polynomial Degree Comparison (Test Set Metrics) ---")
display(poly_comparison)
```

--- Polynomial Degree Comparison (Test Set Metrics) ---

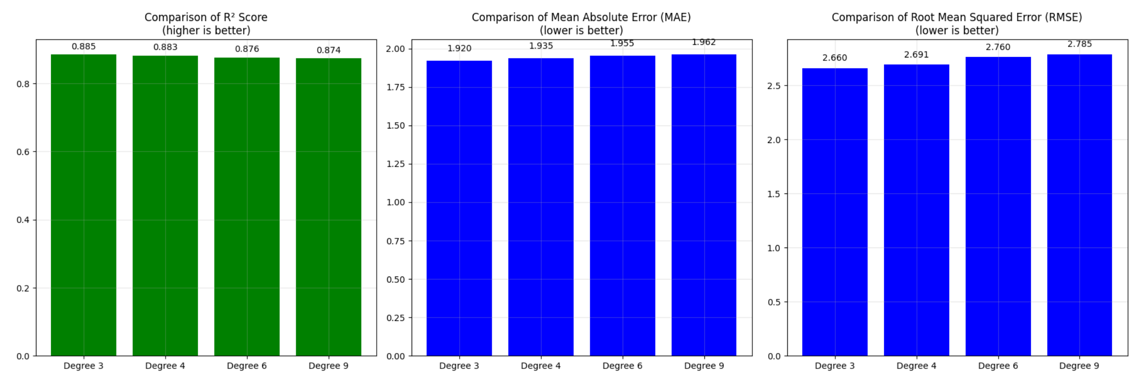
	Degree 3	Degree 4	Degree 6	Degree 9
R² Score	0.885180	0.882503	0.876412	0.874109

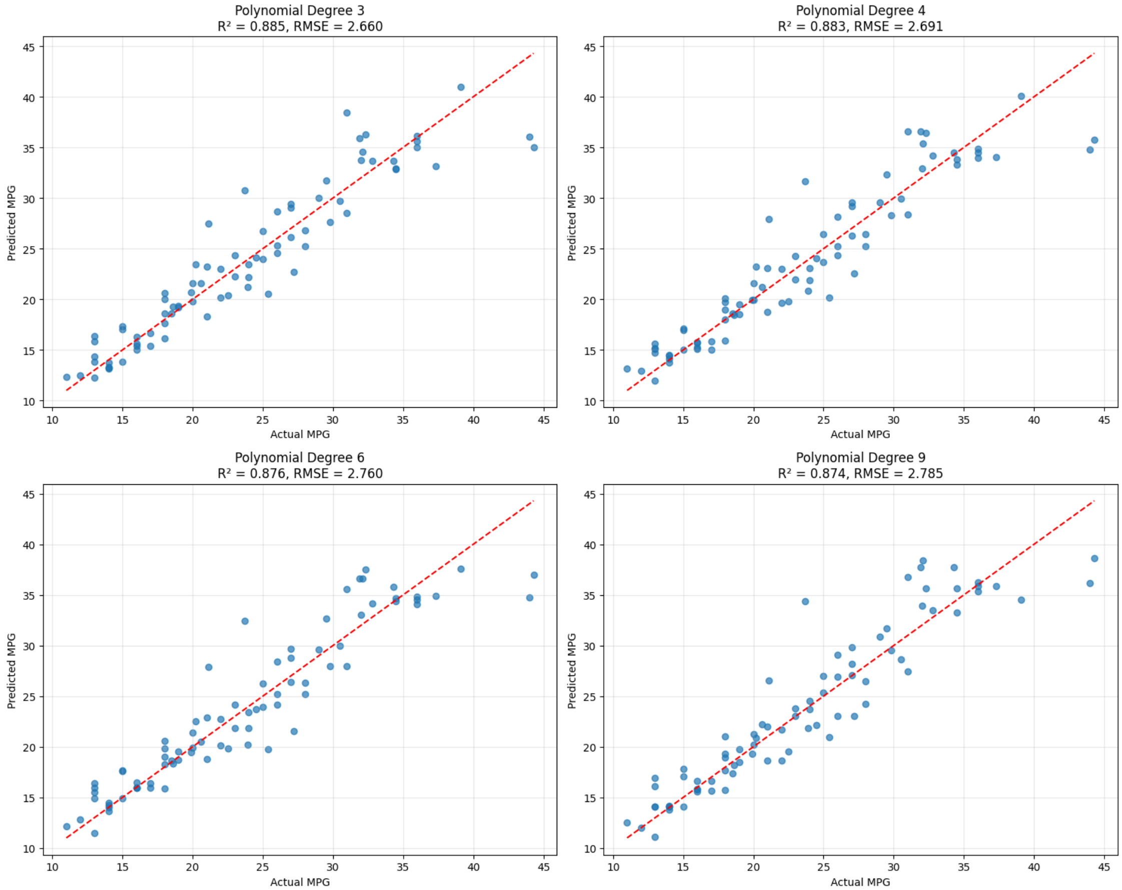
```
Mean Absolute Error (MAE)             1.920014  1.935474  1.954713  1.961635
Root Mean Squared Error (RMSE)        2.660116  2.690950  2.759813  2.785412
Mean Absolute Percentage Error (MAPE) 8.114776  8.184074  8.274599  8.265061
Training Time (seconds)               0.003828  0.003169  0.004433  0.008789
```

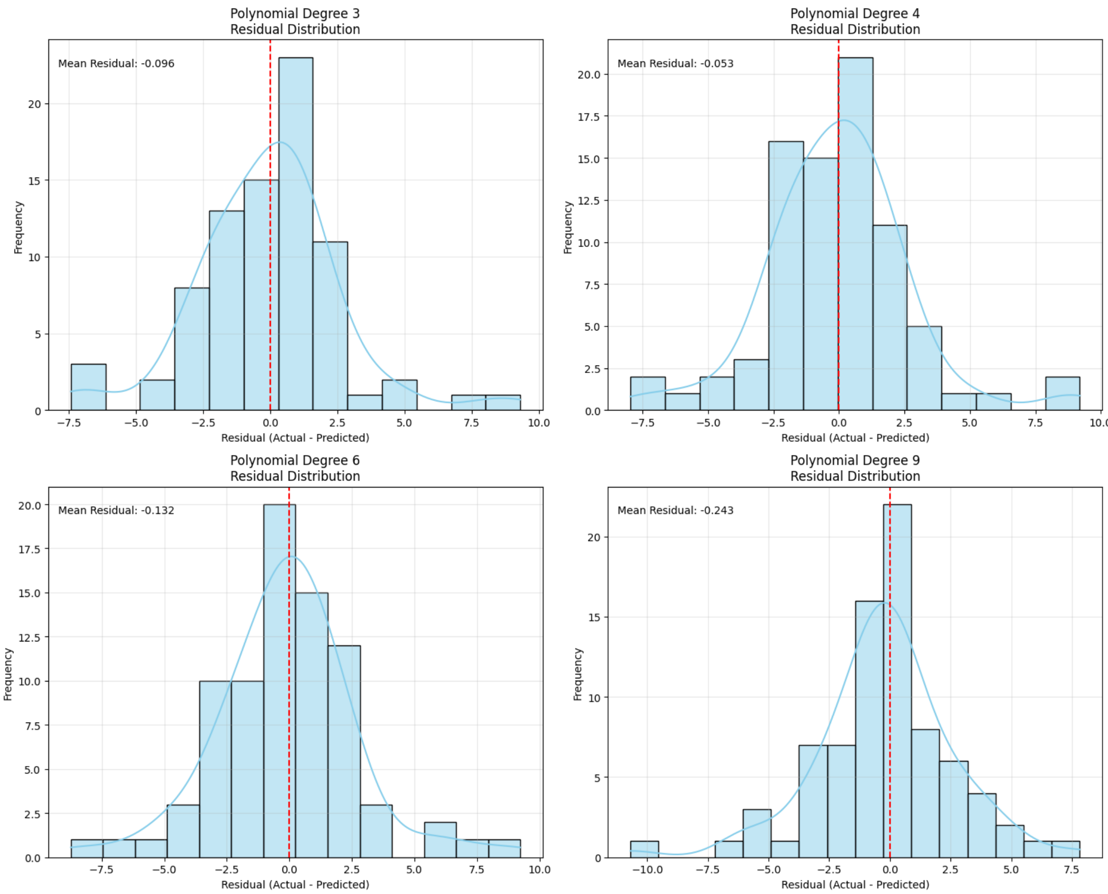
[38]:
```python
# Visualize metrics comparison
metrics_to_plot = ['R² Score', 'Mean Absolute Error (MAE)', 'Root Mean Squared␣
↪Error (RMSE)']
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

for i, metric in enumerate(metrics_to_plot):
    values = poly_comparison.loc[metric].values
    x_pos = range(len(poly_degrees))

    # For R², higher is better
    if metric == 'R² Score':
        axes[i].bar(x_pos, values, color='green')
        axes[i].set_title(f'Comparison of {metric}\n(higher is better)')
    # For error metrics, lower is better
    else:
        axes[i].bar(x_pos, values, color='blue')
        axes[i].set_title(f'Comparison of {metric}\n(lower is better)')

    axes[i].set_xticks(x_pos)
    axes[i].set_xticklabels([f'Degree {d}' for d in poly_degrees])
    axes[i].grid(True, alpha=0.3)

    # Add value labels on top of bars
    for j, v in enumerate(values):
        axes[i].text(j, v + (0.01 if metric == 'R² Score' else 0.05),
                    f'{v:.3f}', ha='center', va='bottom')

plt.tight_layout()
plt.show()


# Plot predictions vs actual values for all degrees
fig, axes = plt.subplots(2, 2, figsize=(15, 12))
axes = axes.flatten()

for i, degree in enumerate(poly_degrees):
    axes[i].scatter(y_test, poly_predictions[degree]['test'], alpha=0.7)
    axes[i].plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],␣
↪'r--')
    axes[i].set_xlabel('Actual MPG')
    axes[i].set_ylabel('Predicted MPG')
```

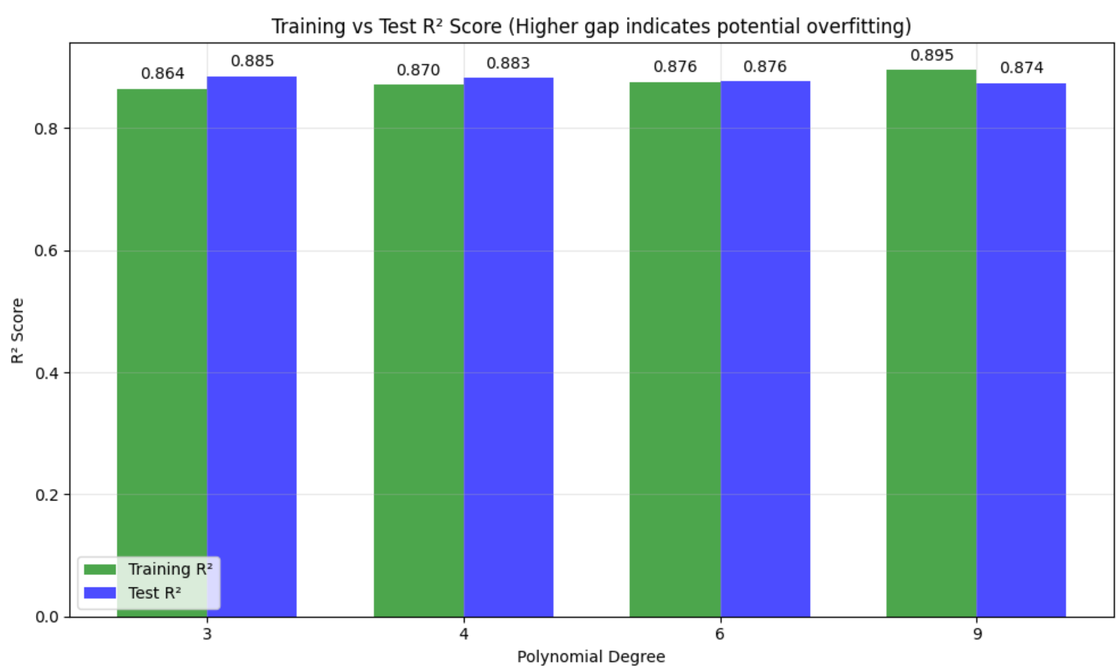
```
    axes[i].set_title(f'Polynomial Degree {degree}\nR² =␣
↪{poly_test_metrics[degree]["R2"]:.3f}, RMSE =␣
↪{poly_test_metrics[degree]["RMSE"]:.3f}')
    axes[i].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Plot residual distributions for all degrees
fig, axes = plt.subplots(2, 2, figsize=(15, 12))
axes = axes.flatten()

for i, degree in enumerate(poly_degrees):
    residuals = y_test - poly_predictions[degree]['test']
    sns.histplot(residuals, kde=True, ax=axes[i], color='skyblue')
    axes[i].set_xlabel('Residual (Actual - Predicted)')
    axes[i].set_ylabel('Frequency')
    axes[i].set_title(f'Polynomial Degree {degree}\nResidual Distribution')
    axes[i].grid(True, alpha=0.3)

    # Add vertical line at 0
    axes[i].axvline(x=0, color='red', linestyle='--')

    # Calculate and display mean residual
    mean_residual = residuals.mean()
    axes[i].text(0.02, 0.95, f'Mean Residual: {mean_residual:.3f}',
                 transform=axes[i].transAxes, verticalalignment='top')

plt.tight_layout()
plt.show()

# Check for signs of overfitting by comparing R² on training vs test sets
plt.figure(figsize=(10, 6))
train_r2 = [poly_train_metrics[d]['R2'] for d in poly_degrees]
test_r2 = [poly_test_metrics[d]['R2'] for d in poly_degrees]

x = range(len(poly_degrees))
width = 0.35

plt.bar([i - width/2 for i in x], train_r2, width, label='Training R²',␣
↪color='green', alpha=0.7)
plt.bar([i + width/2 for i in x], test_r2, width, label='Test R²',␣
↪color='blue', alpha=0.7)

plt.xlabel('Polynomial Degree')
plt.ylabel('R² Score')
```

```
plt.title('Training vs Test R² Score (Higher gap indicates potential␣
 ↪overfitting)')
plt.xticks(x, [f'{d}' for d in poly_degrees])
plt.legend()
plt.grid(True, alpha=0.3)

# Add value labels
for i, v in enumerate(train_r2):
    plt.text(i - width/2, v + 0.01, f'{v:.3f}', ha='center', va='bottom')
for i, v in enumerate(test_r2):
    plt.text(i + width/2, v + 0.01, f'{v:.3f}', ha='center', va='bottom')

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




Polynomial Degree 3
R² = 0.885, RMSE = 2.660
Polynomial Degree 4
R² = 0.883, RMSE = 2.691
Polynomial Degree 6
R² = 0.876, RMSE = 2.760
Polynomial Degree 9
R² = 0.874, RMSE = 2.785
Predicted MPG
Actual MPG


Polynomial Degree 3
Residual Distribution
Mean Residual: -0.096
Frequency
Residual (Actual - Predicted)
Polynomial Degree 4
Residual Distribution
Mean Residual: -0.053
Frequency
Residual (Actual - Predicted)
Polynomial Degree 6
Residual Distribution
Mean Residual: -0.132
Frequency
Residual (Actual - Predicted)
Polynomial Degree 9
Residual Distribution
Mean Residual: -0.243
Frequency
Residual (Actual - Predicted)


Training vs Test R² Score (Higher gap indicates potential overfitting)
0.864
0.885
0.870
0.883
0.876
0.876
0.895
0.874
R² Score
Training R²
Test R²
Polynomial Degree

1.7.4 Reflection 5: Which models performed better? How does scaling impact results?

1.8 Section 6. Final Thoughts & Insights

1.8.1 6.1 Summarize findings.

1.8.2 6.2 Discuss challenges faced.

1.8.3 6.3 If you had more time, what would you try next?

1.8.4 Reflection 6: What did you learn from this project?