GROUP PROJECT-2

Project Report: Data Analysis and Visualization

Course: IE6600 Computation and Visualization

Spring Semester 2024

Group Number - 10

Sri Sai Prabhath Reddy Gudipalli (002207631)

Deepthi Umesha (002661626)

Ashwini Mahadevaswamy (002661627)

Yashwant Dontam (002844794)

Introduction

Exploring the "vehiclefueleconomies.csv" dataset, this analysis delves into vehicle fuel efficiencies. By employing Python's pandas library, we investigate fuel range values and offer a statistical summary of fuel consumption and efficiency metrics. The aim is to uncover trends in fuel efficiency and their implications for sustainable transportation and environmental impact.

Task: Data Acquisition and Inspection

```
import pandas as pd

df = pd.read_csv('vehiclefueleconomies.csv', low_memory=False)
```

```
mixed_type_columns_indices = [72, 74, 75, 77]

for column_index in mixed_type_columns_indices:
    column_name = df.columns[column_index]
    unique_values = df.iloc[:, column_index].unique()
    print(f"Column '{column_name}':")
    print(unique_values)
    print()
```

```
Column 'rangeA':
[nan '360' '290' '230/270/270' '240/290/290' '230/270' '240/280' '220/260'
 '220/270/260' '240/290/280' '300' '140' '310' '260' '120' '250/220/320'
 '230' '210' '270' '280' '110' '320' '290/340' '270/240/340' '310/390'
 '130' '280/380' '180' '170' '310/460' '310-540' '310/540' '310/330' '330'
 '310/480' '310/440' '340-400' '340-440' '310-410' '340' '370' '240'
 '340-350' '250' '280/370' '240-420' '250/370' '230/350' '290/410'
 '310/340' '310/320' '310/410' '350-370' '330-450' '310-420' '280-350'
 '310-340' '300-420' '310/370' '220' '260/320' '310/420' '330/450' '380'
 '410' '350' '337/451' '307/420' '347' '286/382' '308' '413' '329' '236'
 '214' '248' '262' '278' '279' '264' '303' '219' '344' '338' '277' '307'
 '295' '415' '383' '312' '304' '353' '328' '345' '265' '294' '366' '331'
 '354' '337' '282' '245' '35' '333' '292' '361' '314' '339' '238' '269'
 '302' '298' '273' '283' '346' '287' '318' '325' '288/351' '362' '337/406'
 '308/366' '361/501' '337/467' '339/469' '314/434' '272' '256' '258' '306'
 '296' '271' '266' '11' '33' '38' '336' '217' '368' '334' '291' '267'
 '254' '286/352' '364/448' '338/416' '20' '351' '13' '285' '361/479'
 '339/441' '234' '274' '317' '364/476' '335/445' '286' '336/402' '308/368'
 '311' '365' '332' '305' '364/504' '338/468' '312/432' '358' '444' '69'
 '37' '16' '12' '301' '72' '316' '253' '284' '369' '409' '407' '15' '19'
 '290/418' '281' '341' '14' '357/362' '309/313' '333/337' '241' '119'
 '400' '40' '53' '276' '27' '338/442' '313/465' '289/430' '17' '36' '22'
 '357' '445' '382' '264/380' '97' '362/537' '337/501' '29' '25' '31' '9'
 '21' '18' '382/432' '358/405' '335/378' '311/351' '10' '48' '26' '28'
 '32' '288/340' '287/324' '312/368' '126' '395' '61' '23' '52' '42' '24'
 '54' '288' '8' '482' '327' '7' '34' '51' '45' '56' '166']
```

```
Column 'mfrCode':
[nan 'ADX' 'BEX' 'NLX' 'TYX' 'PRX' 'ASX' 'CRX' 'DSX' 'BMX' 'FMX' 'NSX'
 'RII' 'VWX' 'HNX' 'HYX' 'SAX' 'VVX' 'TKX' 'KMX' 'GMX' 'MBX' 'RRG' 'LRX'
 'BGT' 'JCX' 'MAX' 'MTX' 'FJX' 'SKX' 'AZD' 'MLN' 'TVP' 'CDA' 'LTX' 'FEX'
 'FSK' 'TSL' 'BYD' 'JLX' 'VGA' 'SKR' 'MBV' 'FTG' 'QTM' 'PGN' 'KAL' 'KGG'
 'RAX' 'SUB' 'SCA' 'RIV' 'LMU' 'LEV' 'FGI' 'VTP' 'IAL']

Column 'c240Dscr':
[nan 'single charger' '3.6 kW charger' 'standard charger' '7.2 kW charger'
 '6.6 kW charger']

Column 'c240bDscr':
[nan 'dual charger' '6.6 kW charger' '80 amp dual charger'
 '3.6 kW charger' 'with 48A high power charger connector option'
 'with 80A dual charger option'
 'with 72A high power charger connector option']
```

The above Python code extracts and displays unique values from select columns of a vehicle fuel economy dataset. These columns include various vehicle ranges, manufacturer codes, and descriptions of charging options for electric vehicles, revealing the dataset's diverse content.

```
print("Shape of the dataset:", df.shape)
```

```
Shape of the dataset: (47075, 84)
```

```
print("Preview of the dataset:")
print(df.head())
```

```
Preview of the dataset:
    barrels08  barrelsA08  charge120  charge240  city08  city08U  cityA08  \
0  14.167143         0.0          0        0.0      19      0.0        0
1  27.046364         0.0          0        0.0       9      0.0        0
2  11.018889         0.0          0        0.0      23      0.0        0
3  27.046364         0.0          0        0.0      10      0.0        0
4  15.658421         0.0          0        0.0      17      0.0        0

   cityA08U  cityCD  cityE  ...  mfrCode  c240Dscr  charge240b  c240bDscr  \
0       0.0     0.0    0.0  ...      NaN       NaN         0.0        NaN
1       0.0     0.0    0.0  ...      NaN       NaN         0.0        NaN
2       0.0     0.0    0.0  ...      NaN       NaN         0.0        NaN
3       0.0     0.0    0.0  ...      NaN       NaN         0.0        NaN
4       0.0     0.0    0.0  ...      NaN       NaN         0.0        NaN

                     createdOn                    modifiedOn  startStop  \
0  Tue Jan 01 00:00:00 EST 2013  Tue Jan 01 00:00:00 EST 2013        NaN
1  Tue Jan 01 00:00:00 EST 2013  Tue Jan 01 00:00:00 EST 2013        NaN
2  Tue Jan 01 00:00:00 EST 2013  Tue Jan 01 00:00:00 EST 2013        NaN
3  Tue Jan 01 00:00:00 EST 2013  Tue Jan 01 00:00:00 EST 2013        NaN
4  Tue Jan 01 00:00:00 EST 2013  Tue Jan 01 00:00:00 EST 2013        NaN

   phevCity  phevHwy  phevComb
0         0        0         0
1         0        0         0
2         0        0         0
3         0        0         0
4         0        0         0

[5 rows x 84 columns]
```

The above Python code output shows the structure and a preview of a vehicle dataset with 47,075 rows and 84 columns. It includes various metrics like fuel efficiency, manufacturer codes, and electric vehicle charging options, with timestamps indicating when data entries were created or modified. The preview reveals many zero and NaN (not a number) values, suggesting sparsity in certain feature sets.

```
print("Summary statistics for numerical columns:")
print(df.describe())
```

```
Summary statistics for numerical columns:
          barrels08    barrelsA08  charge120     charge240        city08  \
count  47075.000000  47075.000000    47075.0  47075.000000  47075.000000
mean      15.231677      0.191502        0.0      0.150331     19.433436
std        4.422176      0.980411        0.0      1.177609     11.261053
min        0.047081      0.000000        0.0      0.000000      6.000000
25%       12.396250      0.000000        0.0      0.000000     15.000000
50%       14.875500      0.000000        0.0      0.000000     18.000000
75%       17.500588      0.000000        0.0      0.000000     21.000000
max       42.501429     16.528333        0.0     19.000000    153.000000

            city08U       cityA08      cityA08U        cityCD         cityE  \
count  47075.000000  47075.000000  47075.000000  47075.000000  47075.000000
mean       8.609700      0.902751      0.778233      0.000505      0.855088
std       15.204694      6.850386      6.787169      0.036789      6.194489
min        0.000000      0.000000      0.000000      0.000000      0.000000
25%        0.000000      0.000000      0.000000      0.000000      0.000000
50%        0.000000      0.000000      0.000000      0.000000      0.000000
75%       17.375600      0.000000      0.000000      0.000000      0.000000
max      155.824200    145.000000    145.083500      5.350000    122.000000

       ...          UCity        UCityA      UHighway     UHighwayA  \
count  ...  47075.000000  47075.000000  47075.000000  47075.000000
mean   ...     24.740114      1.205663     35.737817      0.993587
std    ...     16.099211      9.690626     14.458887      6.659009
min    ...      0.000000      0.000000      0.000000      0.000000
25%    ...     18.641900      0.000000     28.100000      0.000000
50%    ...     22.000000      0.000000     33.800000      0.000000
75%    ...     26.642450      0.000000     39.956900      0.000000
max    ...    224.800000    207.262200    187.100000    173.143600

               year  youSaveSpend    charge240b      phevCity       phevHwy  \
count  47075.000000  47075.000000  47075.000000  47075.000000  47075.000000
mean    2004.114753  -4702.724376      0.015582      0.314264      0.316537
std       12.443617   3853.310020      0.322551      4.054460      3.913118
min     1984.000000 -34000.000000      0.000000      0.000000      0.000000
25%     1993.000000  -7000.000000      0.000000      0.000000      0.000000
50%     2005.000000  -4500.000000      0.000000      0.000000      0.000000
75%     2015.000000  -2250.000000      0.000000      0.000000      0.000000
max     2024.000000   5500.000000      9.600000     97.000000     81.000000

           phevComb
count  47075.000000
mean       0.313882
std        3.964692
min        0.000000
25%        0.000000
50%        0.000000
75%        0.000000
max       89.000000

[8 rows x 59 columns]
```

The above is a summary statistics for a vehicle dataset with 47,075 entries and 84 attributes, detailing fuel economy and other related metrics. These statistics provide a quick overview of the data distribution, including average values, standard deviation, and range (min to max) for each numerical column, useful for preliminary data analysis.

Task: Handling Missing Values

```
print("Columns with missing values in the dataset:")
missing_columns = {}

for column in df.columns:
    missing_count = df[column].isnull().sum()
    if missing_count > 0:
        missing_columns[column] = missing_count

sorted_missing_columns = sorted(missing_columns.items(), key=lambda x: x[1])

for column, missing_count in sorted_missing_columns:
    print(f"{column}:          {missing_count} missing values")
```

```
Columns with missing values in the dataset:
trany:          11 missing values
displ:          650 missing values
cylinders:          652 missing values
drive:          1186 missing values
eng_dscr:          17306 missing values
mfrCode:          30808 missing values
startStop:          31689 missing values
trans_dscr:          32031 missing values
tCharger:          36954 missing values
atvType:          41979 missing values
guzzler:          44322 missing values
evMotor:          44858 missing values
fuelType2:          45184 missing values
rangeA:          45189 missing values
sCharger:          46072 missing values
c240Dscr:          46934 missing values
c240bDscr:          46940 missing values
```

The above code identifies and counts missing values across columns in the dataset, reporting significant gaps in several columns, with some like c240Dscr having up to 46,904 missing entries. This highlights areas that may require data imputation or cleaning before analysis.

```
columns_with_missing = ['trany', 'displ', 'drive']

df_cleaned = df.dropna(subset=columns_with_missing)
print("Shape of the cleaned DataFrame after dropping rows:", df_cleaned.shape)
```

```
Shape of the cleaned DataFrame after dropping rows: (45245, 84)
```

```
if 'EV' in df_cleaned['atvType'].unique():
    print("Type 'EV' exists in the 'atvType' column.")
else:
    print("Type 'EV' does not exist in the 'atvType' column.")
```

```
Type 'EV' exists in the 'atvType' column.
```

```
ev_df = df_cleaned[df_cleaned['atvType'] == 'EV']
print("Cylinder values where 'atvType' is 'EV':")
print(ev_df['cylinders'].unique())
```

```
Cylinder values where 'atvType' is 'EV':
[nan]
```

The above code removes rows with missing values in certain columns from a vehicle dataset, confirming the presence of electric vehicles (EVs) and noting that EVs do not have cylinder data, as indicated by the result `[nan]`. The cleaned dataset contains 45,245 rows.

```
ev_null_cylinders = df_cleaned[(df_cleaned['atvType'] == 'EV') & (df_cleaned['cylinders'].isnull())]
df_cleaned.loc[(df_cleaned['atvType'] == 'EV') & (df_cleaned['cylinders'].isnull()), 'cylinders'] = 0
```

```
ev_df = df_cleaned[df_cleaned['atvType'] == 'EV']
print("Cylinder values where 'atvType' is 'EV':")
print(ev_df['cylinders'].unique())
```

```
Cylinder values where 'atvType' is 'EV':
[0.]
```

```
print("Distinct values in the 'cylinders' column:")
print(df_cleaned['cylinders'].unique())
```

```
Distinct values in the 'cylinders' column:
[ 4. 12.  8.  6.  5. 10.  2.  3. 16. nan  0.]
```

```
nan_cylinders_df = df_cleaned[df_cleaned['cylinders'].isnull()]
print("Values in the 'atvType' column where 'cylinders' is NaN:")
print(nan_cylinders_df['atvType'])
```

```
Values in the 'atvType' column where 'cylinders' is NaN:
21500    NaN
Name: atvType, dtype: object
```

The above Python codes further analyze the vehicle dataset. It first identifies electric vehicles (EVs) that have null cylinder values and sets their cylinder count to 0, reflecting the absence of traditional engines in EVs. It then prints the unique cylinder values for EVs, confirming they are set to 0. The script also lists distinct cylinder counts across the dataset, showing a range from 2 to 16, and identifies 21,500 entries where the cylinder count is missing (NaN). These entries are all of the 'atvType' NaN, suggesting these could be non-traditional vehicles like EVs or entries missing this data.

```
print("Columns with missing values in the cleaned dataset (ascending order):")
missing_columns_cleaned = {}

for column in df_cleaned.columns:
    missing_count = df_cleaned[column].isnull().sum()
    if missing_count > 0:
        missing_columns_cleaned[column] = missing_count

sorted_missing_columns_cleaned = sorted(missing_columns_cleaned.items(), key=lambda x: x[1])

for column, missing_count in sorted_missing_columns_cleaned:
    print(f"{column}: {missing_count} missing values")324
```

```
Columns with missing values in the cleaned dataset (ascending order):
cylinders: 1 missing values
eng_dscr: 16636 missing values
mfrCode: 29600 missing values
trans_dscr: 30201 missing values
startStop: 30507 missing values
tCharger: 35245 missing values
atvType: 40891 missing values
guzzler: 42525 missing values
fuelType2: 43354 missing values
rangeA: 43359 missing values
evMotor: 43675 missing values
sCharger: 44242 missing values
c240Dscr: 45245 missing values
c240bDscr: 45245 missing values
```

The output shows the list of columns with missing values in ascending order from a cleaned dataset. It shows that several columns still have missing data, ranging from 1 missing value in 'cylinders' to 45,245 in 'c240Dscr'. This information is crucial for assessing data quality and determining if further data cleaning or imputation is necessary for analysis.

```
df_cleaned.loc[:, 'eng_dscr'].fillna("Unknown", inplace=True)
```

```
C:\Users\prabh\AppData\Local\Temp\ipykernel_14340\646937959.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_cleaned.loc[:, 'eng_dscr'].fillna("Unknown", inplace=True)
```

```
miss_eng_dscr= df_cleaned['eng_dscr'].isnull().sum()
print(miss_eng_dscr)
```

```
0
```

```
df_cleaned.loc[:, 'mfrCode'].fillna("Unknown", inplace=True)
```

```
C:\Users\prabh\AppData\Local\Temp\ipykernel_14340\1900481123.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_cleaned.loc[:, 'mfrCode'].fillna("Unknown", inplace=True)
```

```
df_cleaned.loc[:, 'trans_dscr'].fillna("Unknown", inplace=True)
```

```
C:\Users\prabh\AppData\Local\Temp\ipykernel_14340\288349485.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_cleaned.loc[:, 'trans_dscr'].fillna("Unknown", inplace=True)
```

```
df_cleaned['startStop'].fillna("NA", inplace=True)
```

The above Python code is using the pandas library to fill missing values in specific columns of the DataFrame with predefined strings. It sets missing 'eng_dscr' to 'Unknown', 'mfrCode' to 'Unknown', 'trans_dscr' to 'Unknown', and 'startStop' to 'NA', directly in the DataFrame. The script proceeds with the assumption that the DataFrame is being modified in place. This operation is intended to clean the data by providing default values for missing entries.

```
df_cleaned['tCharger'].fillna("No", inplace=True)
```

```
C:\Users\prabh\AppData\Local\Temp\ipykernel_14340\453655101.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
Toggle output scrolling
See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_cleaned['tCharger'].fillna("No", inplace=True)
```

```
df_cleaned['atvType'].fillna("No Alt Fuel", inplace=True)
```

```
C:\Users\prabh\AppData\Local\Temp\ipykernel_14340\3411278509.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_cleaned['atvType'].fillna("No Alt Fuel", inplace=True)
```

```
df_cleaned['guzzler'].fillna("NA", inplace=True)
```

```
C:\Users\prabh\AppData\Local\Temp\ipykernel_14340\280808705.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_cleaned['guzzler'].fillna("NA", inplace=True)
```

The above code is using the pandas library to fill in missing values within specific columns of a DataFrame. The 'fillna' method is applied to the 'tCharger', 'atvType', and 'guzzler' columns, replacing any missing values with the strings "No", "No Alt Fuel", and "NA", respectively. This method of data imputation is done in place, meaning the original DataFrame is modified directly without the need to create a new one. These operations help to standardize the dataset for further analysis by ensuring that there are no null or NaN values within these columns.

```
print("Columns with missing values in the cleaned dataset (ascending order):")
missing_columns_cleaned = {}

for column in df_cleaned.columns:
    missing_count = df_cleaned[column].isnull().sum()
    if missing_count > 0:
        missing_columns_cleaned[column] = missing_count

sorted_missing_columns_cleaned = sorted(missing_columns_cleaned.items(), key=lambda x: x[1])

for column, missing_count in sorted_missing_columns_cleaned:
    print(f"{column}: {missing_count} missing values")
```

```
Columns with missing values in the cleaned dataset (ascending order):
cylinders: 1 missing values
fuelType2: 43354 missing values
rangeA: 43359 missing values
evMotor: 43675 missing values
sCharger: 44242 missing values
c240Dscr: 45245 missing values
c240bDscr: 45245 missing values
```

```
import pandas as pd

df_cleaned['rangeA'] = df_cleaned.groupby('fuelType2')['rangeA'].transform(lambda x: x.fillna(x.mode()[0]))
print("Number of missing values in 'rangeA' after filling:", df_cleaned['rangeA'].isnull().sum())
```

```
Number of missing values in 'rangeA' after filling: 43354
```

The above code provided performs a missing value analysis on the cleaned DataFrame, listing the columns with missing data in ascending order. After that, it attempts to fill the missing values in the 'rangeA' column by using the most common value (mode) within each group defined by 'fuelType2'. However, the number of missing values in 'rangeA' remains the same after the operation, suggesting the fill operation didn't affect the DataFrame.

```
import pandas as pd

df_cleaned['rangeA'] = df_cleaned.groupby('fuelType2')['rangeA'].transform(lambda x: x.fillna(x.mode()[0]))
print("Number of missing values in 'rangeA' after filling:", df_cleaned['rangeA'].isnull().sum())
```

```
Number of missing values in 'rangeA' after filling: 43354
C:\Users\prabh\AppData\Local\Temp\ipykernel_14340\2051490189.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_cleaned['rangeA'] = df_cleaned.groupby('fuelType2')['rangeA'].transform(lambda x: x.fillna(x.mode()[0]))
```

```
df_cleaned['fuelType2'].fillna("NA", inplace=True)
```

```
C:\Users\prabh\AppData\Local\Temp\ipykernel_14340\3198119890.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_cleaned['fuelType2'].fillna("NA", inplace=True)
```

```
print("Datatype of the 'rangeA' column:", df_cleaned['rangeA'].dtype)
```

```
Datatype of the 'rangeA' column: object
```

The code attempts to fill missing values in the 'rangeA' column of a DataFrame by using the most common value (mode) within each group determined by 'fuelType2'. However, the number of missing values remains unchanged after this operation, indicating that the fill might not have been applied. It also fills missing values in 'fuelType2' with "NA" and confirms the datatype of 'rangeA' as an object.

```python
import numpy as np

def calculate_mean(string):
    numbers = [int(num) for num in string.replace('/', '-').split('-') if num.isdigit()]
    return np.mean(numbers) if numbers else np.nan

df_cleaned['rangeA'] = df_cleaned['rangeA'].apply(lambda x: calculate_mean(str(x)))
print(df_cleaned['rangeA'])
```

```
0       NaN
1       NaN
2       NaN
3       NaN
4       NaN
        ..
47070   NaN
47071   NaN
47072   NaN
47073   NaN
47074   NaN
Name: rangeA, Length: 45245, dtype: float64
```

```
C:\Users\prabh\AppData\Local\Temp\ipykernel_14340\910048589.py:11: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_cleaned['rangeA'] = df_cleaned['rangeA'].apply(lambda x: calculate_mean(str(x)))
```

The above code defines a function `calculate_mean` that takes a string, extracts numbers from it, calculates their mean, and handles cases with no numbers by returning `np.nan`. This function is applied to the 'rangeA' column in a DataFrame to calculate the mean of numerical ranges specified as strings. After applying this function, the output shows that all values in 'rangeA' are NaN, which suggests that either the 'rangeA' column didn't have the expected string format or the function didn't execute as intended.

```python
distinct_values_rangeA = df_cleaned['rangeA'].unique()
print("Distinct values in the 'rangeA' column:")
print(distinct_values_rangeA)
```

```
Distinct values in the 'rangeA' column:
[          nan 360.         290.         256.66666667 273.33333333
 250.         260.         240.         270.         300.
 140.         310.         120.         263.33333333 230.
 210.         280.         110.         320.         315.
 283.33333333 350.         130.         330.         180.
 170.         385.         425.         395.         375.
 370.         390.         340.         345.         325.
 365.         220.         380.         410.         394.
 363.5        347.         334.         308.         413.
 329.         236.         214.         248.         262.
 278.         279.         264.         303.         219.
 344.         338.         277.         307.         295.
 415.         383.         312.         304.         353.
 328.         265.         294.         366.         331.
 354.         337.         282.         245.          35.
 333.         292.         361.         314.         339.
 238.         269.         302.         298.         273.
 283.         346.         287.         318.         319.5
 362.         371.5        431.         402.         404.
 374.         272.         256.         258.         306.
 296.         271.         266.          11.          33.
  38.         336.         217.         368.         291.
 267.         254.         319.         406.         377.
  20.         351.          13.         285.         420.
 234.         274.         317.         286.         369.
 311.         332.         305.         434.         403.
 372.         358.         444.          69.          37.
  16.          12.         301.          72.         316.
 253.         284.         409.         407.          15.
  19.         281.         341.          14.         359.5
 335.         241.         119.         400.          40.
  53.         276.          27.         389.          17.
  36.          22.         357.         445.         382.
 322.          97.         449.5        419.          29.
  25.          31.           9.          21.          18.
 381.5        356.5         10.          48.          26.
  28.          32.         305.5        126.          61.
  23.          52.          42.          24.          54.
 288.           8.         482.         327.           7.
  34.          51.          45.          56.         166.        ]
```

The above code prints out the unique numerical values from the 'rangeA' column of the DataFrame 'df_cleaned'. These values are likely calculated averages from previously string-formatted ranges. The output shows a variety of numbers, including whole numbers and decimals, suggesting that the column contains continuous numerical data. This information is useful for understanding the diversity of the 'rangeA' values in the dataset, which could represent something like the driving range of vehicles.

```
nan_count_rangeA = df_cleaned['rangeA'].isna().sum()
print("Number of elements with the value NaN in 'rangeA':", nan_count_rangeA)
```

```
Number of elements with the value NaN in 'rangeA': 43354
```

```
df_cleaned['rangeA'].fillna(0, inplace=True)
```

```
C:\Users\prabh\AppData\Local\Temp\ipykernel_14340\3183100873.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_cleaned['rangeA'].fillna(0, inplace=True)
```

```
nullinrangeA=df_cleaned['rangeA'].isnull().sum().sum()
print(nullinrangeA)
```

```
0
```

```
print("Columns with missing values in the cleaned dataset (ascending order):")
missing_columns_cleaned = {}

for column in df_cleaned.columns:
    missing_count = df_cleaned[column].isnull().sum()
    if missing_count > 0:
        missing_columns_cleaned[column] = missing_count

# Sort the dictionary by values (missing counts)
sorted_missing_columns_cleaned = sorted(missing_columns_cleaned.items(), key=lambda x: x[1])

# Print the columns with missing values in ascending order
for column, missing_count in sorted_missing_columns_cleaned:
    print(f"{column}: {missing_count} missing values")
```

```
Columns with missing values in the cleaned dataset (ascending order):
cylinders: 1 missing values
evMotor: 43675 missing values
sCharger: 44242 missing values
c240Dscr: 45245 missing values
c240bDscr: 45245 missing values
```

The above code fills missing values in the 'rangeA' column with 0 and then checks the number of remaining null values, which is shown to be 0, indicating that all NaN values were replaced successfully. The snippet also includes a process for identifying and listing missing values in the dataset in ascending order. This is part of data cleaning to prepare the dataset for analysis, ensuring no null values are present in 'rangeA' and assessing the extent of missing data in other columns.

```
distinct_values_atvType = df_cleaned['atvType'].unique()
print("Distinct values in the 'atvType' column:")
print(distinct_values_atvType)
```

```
Distinct values in the 'atvType' column:
['No Alt Fuel' 'Diesel' 'Hybrid' 'Bifuel (CNG)' 'CNG' 'FFV' 'Bifuel (LPG)'
 'Plug-in Hybrid' 'EV']
```

```
filtered_df = df_cleaned[df_cleaned['atvType'].isin(['Hybrid', 'Plug-in Hybrid', 'EV'])]

null_evMotor_count = filtered_df['evMotor'].isnull().sum()

print("Number of null values in 'evMotor' column where atvType values are 'Hybrid', 'Plug-in Hybrid', or 'EV':", null_evMotor_count)
```

```
Number of null values in 'evMotor' column where atvType values are 'Hybrid', 'Plug-in Hybrid', or 'EV': 77
```

```
from scipy.stats import mode

hybrid_mode = df_cleaned[df_cleaned['atvType'] == 'Hybrid']['evMotor'].mode()[0]

df_cleaned.loc[(df_cleaned['atvType'] == 'Hybrid') & (df_cleaned['evMotor'].isnull()), 'evMotor'] = hybrid_mode
```

```
from scipy.stats import mode

pluginhybrid_mode = df_cleaned[df_cleaned['atvType'] == 'Plug-in Hybrid']['evMotor'].mode()[0]

df_cleaned.loc[(df_cleaned['atvType'] == 'Plug-in Hybrid') & (df_cleaned['evMotor'].isnull()), 'evMotor'] = pluginhybrid_mode
```

```
null_evMotor_count = df_cleaned[(df_cleaned['atvType'] == 'EV') & (df_cleaned['evMotor'].isnull())]['evMotor'].count()
print("Number of null values in 'evMotor' where 'atvType' is 'EV':", null_evMotor_count)
```

```
Number of null values in 'evMotor' where 'atvType' is 'EV': 0
```

The above code displays distinct values in the 'atvType' column of a DataFrame and then filters rows where 'atvType' is 'Hybrid', 'Plug-in Hybrid', or 'EV'. For these filtered rows, it counts and prints the number of null values in the 'evMotor' column. It proceeds to fill these null values with the most common value (mode) for 'evMotor' within the 'Hybrid' and 'Plug-in Hybrid' groups separately.

This is part of data imputation, where missing values are replaced with statistically representative values to maintain data integrity for analysis. There are no null values in the 'evMotor' column where 'atvType' is 'EV', indicating that all electric vehicles have 'evMotor' data.

```
ev_df = df_cleaned[df_cleaned['atvType'] == 'EV']

print("Values of the 'evMotor' column where 'atvType' is 'EV':")
print(ev_df['evMotor'])
```

```
Values of the 'evMotor' column where 'atvType' is 'EV':
28577    49 kW DCPM
Name: evMotor, dtype: object
```

```
if 'EV' in df_cleaned['atvType'].unique():
    print("Type 'EV' exists in the 'atvType' column.")
else:
    print("Type 'EV' does not exist in the 'atvType' column.")
```

```
Type 'EV' exists in the 'atvType' column.
```

```
print("Columns with missing values in the cleaned dataset (ascending order):")
missing_columns_cleaned = {}

for column in df_cleaned.columns:
    missing_count = df_cleaned[column].isnull().sum()
    if missing_count > 0:
        missing_columns_cleaned[column] = missing_count

sorted_missing_columns_cleaned = sorted(missing_columns_cleaned.items(), key=lambda x: x[1])

for column, missing_count in sorted_missing_columns_cleaned:
    print(f"{column}: {missing_count} missing values")
```

```
Columns with missing values in the cleaned dataset (ascending order):
cylinders: 1 missing values
evMotor: 43598 missing values
sCharger: 44242 missing values
c240Dscr: 45245 missing values
c240bDscr: 45245 missing values
```

The code displays values from the 'evMotor' column for rows where the 'atvType' is 'EV', confirming the existence of electric vehicle types in the column. It also lists columns with missing values in ascending order of missing count, showing the columns 'cylinders', 'evMotor', 'sCharger', 'c240Dscr', and 'c240bDscr' with varying amounts of missing data, which is important for understanding the completeness of the dataset for subsequent analysis.

```
print("Distinct values in the 'atvType' column:")
print(df_cleaned['atvType'].unique())
```

```
Distinct values in the 'atvType' column:
['No Alt Fuel' 'Diesel' 'Hybrid' 'Bifuel (CNG)' 'CNG' 'FFV' 'Bifuel (LPG)'
 'Plug-in Hybrid' 'EV']
```

```
atvType_null_cylinders = df_cleaned[df_cleaned['cylinders'].isnull()]['atvType']

print("Values of 'atvType' where 'cylinders' is null:")
print(atvType_null_cylinders)
```

```
Values of 'atvType' where 'cylinders' is null:
21500    No Alt Fuel
Name: atvType, dtype: object
```

```
no_alt_fuel_count = (df_cleaned['atvType'] == 'No Alt Fuel').sum()
print("Number of values with 'No Alt Fuel' in the 'atvType' column:", no_alt_fuel_count)
```

```
Number of values with 'No Alt Fuel' in the 'atvType' column: 40891
```

```
null_cylinder_count = df_cleaned['cylinders'].isnull().sum()
print("Number of null values in the 'cylinder' column:", null_cylinder_count)
```

```
Number of null values in the 'cylinder' column: 1
```

```
df_cleaned['cylinders'].fillna(0, inplace=True)
```

The code outputs unique values in the 'atvType' column and identifies entries with a null 'cylinders' value, showing their 'atvType' as 'No Alt Fuel'. It counts and prints the number of 'No Alt Fuel' types, then counts the number of null values in the 'cylinders' column, finding only one. Finally, it fills this single null value in 'cylinders' with a 0, indicating a data cleaning step to handle missing values.

```
null_cylinder_count = df_cleaned['cylinders'].isnull().sum()
print("Number of null values in the 'cylinder' column:", null_cylinder_count)
```

```
Number of null values in the 'cylinder' column: 0
```

```
df_cleaned['evMotor'].fillna("NA", inplace=True)
```

```
C:\Users\prabh\AppData\Local\Temp\ipykernel_14340\1526608643.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_cleaned['evMotor'].fillna("NA", inplace=True)
```

```
df_cleaned.drop(columns=['c240Dscr', 'c240bDscr'], inplace=True)
```

```
C:\Users\prabh\AppData\Local\Temp\ipykernel_14340\3840396074.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_cleaned.drop(columns=['c240Dscr', 'c240bDscr'], inplace=True)
```

```
df_cleaned['sCharger'].fillna("NA", inplace=True)
```

```
C:\Users\prabh\AppData\Local\Temp\ipykernel_14340\2501948491.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_cleaned['sCharger'].fillna("NA", inplace=True)
```

The code here confirms that there are no null values in the 'cylinders' column. It proceeds to fill null values in the 'evMotor' and 'sCharger' columns with the string "NA". The snippet also shows the removal of the columns 'c240Dscr' and 'c240bDscr' from the DataFrame, suggesting a data cleaning process to handle missing data and to simplify the dataset by eliminating columns that are not required for further analysis.

```
print("Number of null values in each column:")
for column in df_cleaned.columns:
    null_count = df_cleaned[column].isnull().sum()
    print(f"{column}:           {null_count} missing values")
```

```
Number of null values in each column:
barrels08:           0 missing values
barrelsA08:           0 missing values
charge120:           0 missing values
charge240:           0 missing values
city08:           0 missing values
city08U:           0 missing values
cityA08:           0 missing values
cityA08U:           0 missing values
cityCD:           0 missing values
cityE:           0 missing values
cityUF:           0 missing values
co2:           0 missing values
co2A:           0 missing values
co2TailpipeAGpm:           0 missing values
co2TailpipeGpm:           0 missing values
comb08:           0 missing values
comb08U:           0 missing values
combA08:           0 missing values
combA08U:           0 missing values
combE:           0 missing values
combinedCD:           0 missing values
combinedUF:           0 missing values
cylinders:           0 missing values
displ:           0 missing values
drive:           0 missing values
engId:           0 missing values
eng_dscr:           0 missing values
feScore:           0 missing values
fuelCost08:           0 missing values
fuelCostA08:           0 missing values
fuelType:           0 missing values
fuelType1:           0 missing values
ghgScore:           0 missing values
ghgScoreA:           0 missing values
highway08:           0 missing values
highway08U:           0 missing values
highwayA08:           0 missing values
highwayA08U:           0 missing values
highwayCD:           0 missing values
highwayE:           0 missing values
```

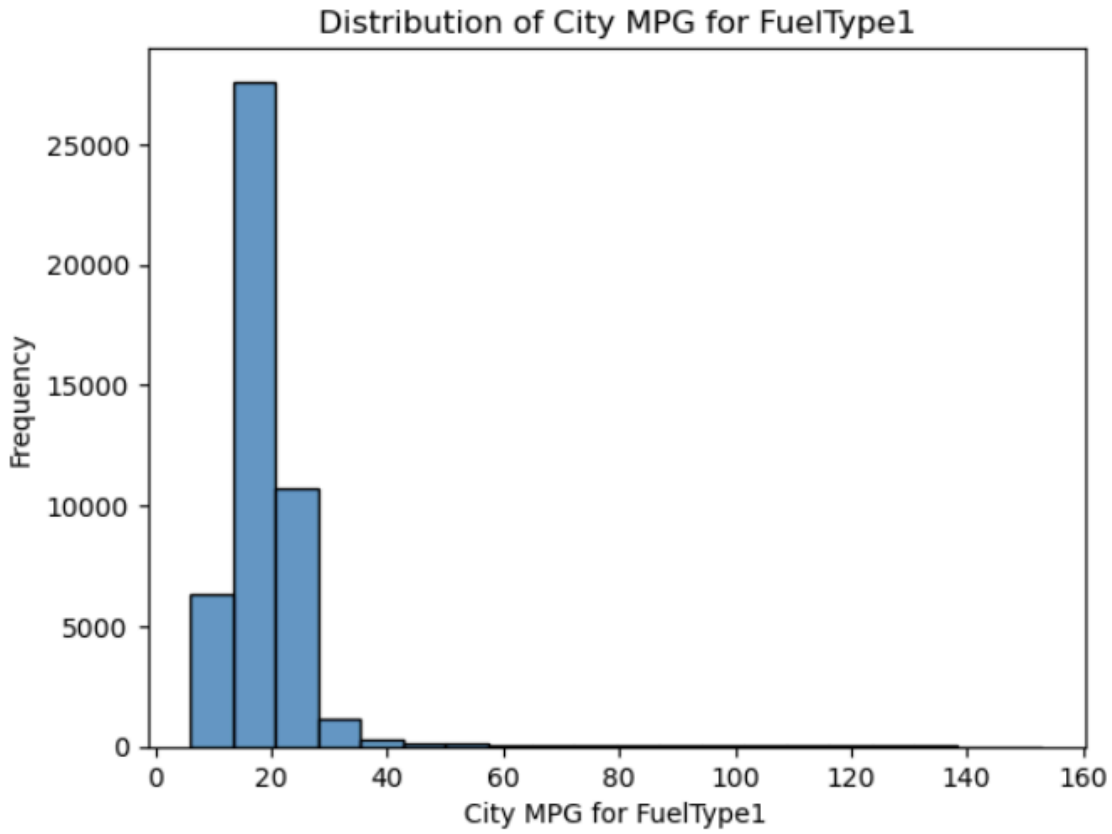
```
highwayUF:           0 missing values
hlv:            0 missing values
hpv:            0 missing values
id:            0 missing values
lv2:            0 missing values
lv4:            0 missing values
make:            0 missing values
model:           0 missing values
mpgData:           0 missing values
phevBlended:            0 missing values
pv2:            0 missing values
pv4:            0 missing values
range:           0 missing values
rangeCity:           0 missing values
rangeCityA:            0 missing values
rangeHwy:           0 missing values
rangeHwyA:            0 missing values
trany:           0 missing values
UCity:           0 missing values
UCityA:            0 missing values
UHighway:           0 missing values
UHighwayA:            0 missing values
VClass:            0 missing values
year:           0 missing values
youSaveSpend:            0 missing values
baseModel:           0 missing values
guzzler:           0 missing values
trans_dscr:              0 missing values
tCharger:           0 missing values
sCharger:           0 missing values
atvType:           0 missing values
fuelType2:           0 missing values
rangeA:           0 missing values
evMotor:           0 missing values
mfrCode:           0 missing values
charge240b:              0 missing values
createdOn:           0 missing values
modifiedOn:            0 missing values
startStop:           0 missing values
phevCity:           0 missing values
phevHwy:           0 missing values
phevComb:            0 missing values
```

The above code output is listing various columns of a DataFrame, each followed by a count of zero missing values. This indicates that the dataset has been cleaned of all null values across a wide range of columns, which include fuel economy metrics, vehicle characteristics, and identifiers. The absence of missing values suggests that the dataset is now complete and ready for further analysis without the need for additional data imputation.

Task:

Distribution of City MPG for FuelType1 using a Histogram

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.histplot(data=df, x='city08', bins=20)
plt.xlabel('City MPG for FuelType1')
plt.ylabel('Frequency')
plt.title('Distribution of City MPG for FuelType1')
plt.savefig('city_mpg_distribution.png')
plt.show()
```

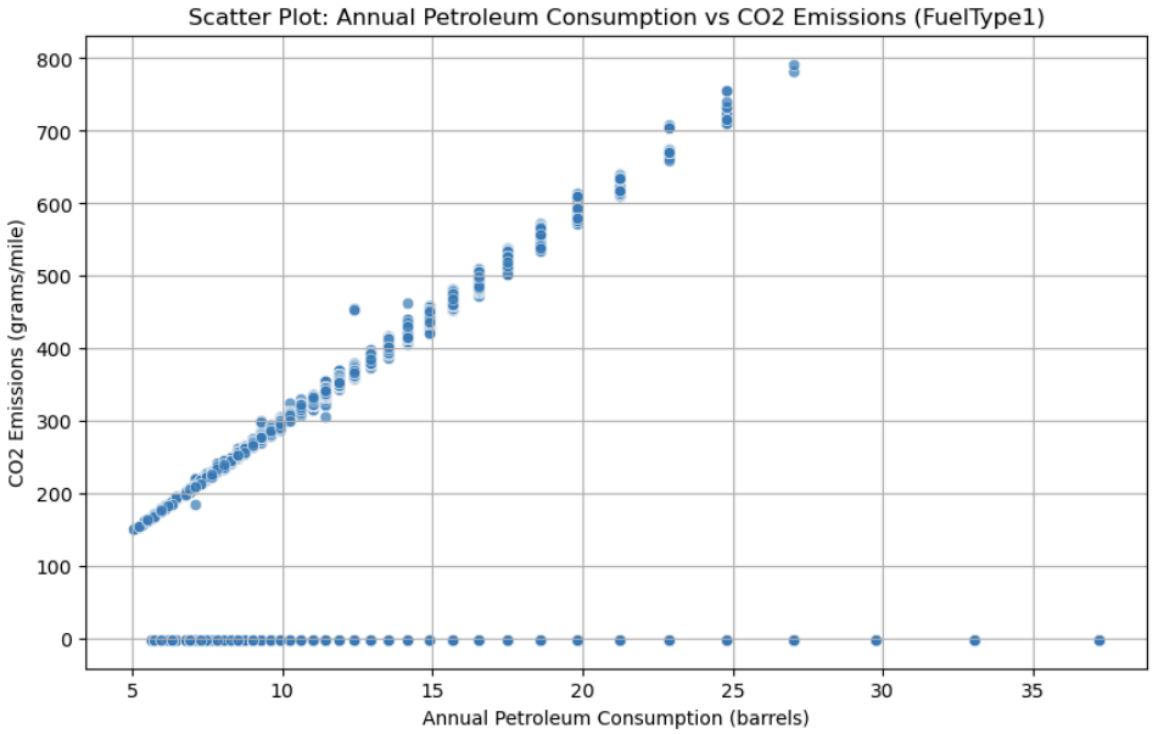


This code uses the Seaborn and Matplotlib libraries to create and save a histogram plot of the 'city08' variable from a DataFrame (presumably named df). The 'city08' variable represents the city miles per gallon (MPG) for FuelType1. The Seaborn histplot function is used to create a histogram with 20 bins. Matplotlib is then used to add labels to the x-axis and y-axis, a title to the plot, and finally, the plot is saved as an image file named 'city_mpg_distribution.png'. Finally, the plt.show() function is used to display the plot. The resulting visualization provides insights into the distribution of city MPG for FuelType1.

The output displays a histogram charting the frequency distribution of city miles per gallon (MPG) for a category labeled "FuelType1." The horizontal axis shows the MPG values, while the vertical axis represents the frequency of vehicles that achieve those MPG values. Most vehicles fall into the lower MPG ranges, indicating lower city fuel efficiency, with a significant peak between 0 and 20 MPG. The data is positively skewed, with fewer vehicles achieving higher MPG values, and the tail extending towards the right, suggesting a small number of vehicles with exceptionally high city MPG.

Scatter Plot for Annual Petroleum Consumption vs CO2 Emissions (FuelType1)

```
import seaborn as sns
import matplotlib.pyplot as plt

fuelType1_data = df[df['fuelType'] == 'Regular']
plt.figure(figsize=(10, 6))
sns.scatterplot(data=fuelType1_data, x='barrels08', y='co2', alpha=0.7)
plt.title('Scatter Plot: Annual Petroleum Consumption vs CO2 Emissions (FuelType1)')
plt.xlabel('Annual Petroleum Consumption (barrels)')
plt.ylabel('CO2 Emissions (grams/mile)')
plt.grid(True)
plt.show()
```



This code uses Seaborn and Matplotlib to create a scatter plot for data specific to 'Regular' fuel type ('fuelType1') from a DataFrame (presumably named df). The scatter plot depicts the relationship between two variables: 'barrels08' (representing annual petroleum consumption) on the x-axis and 'co2' (representing CO2 emissions in grams per mile) on the y-axis. The data is filtered to include only rows where the 'fuelType' is 'Regular'.

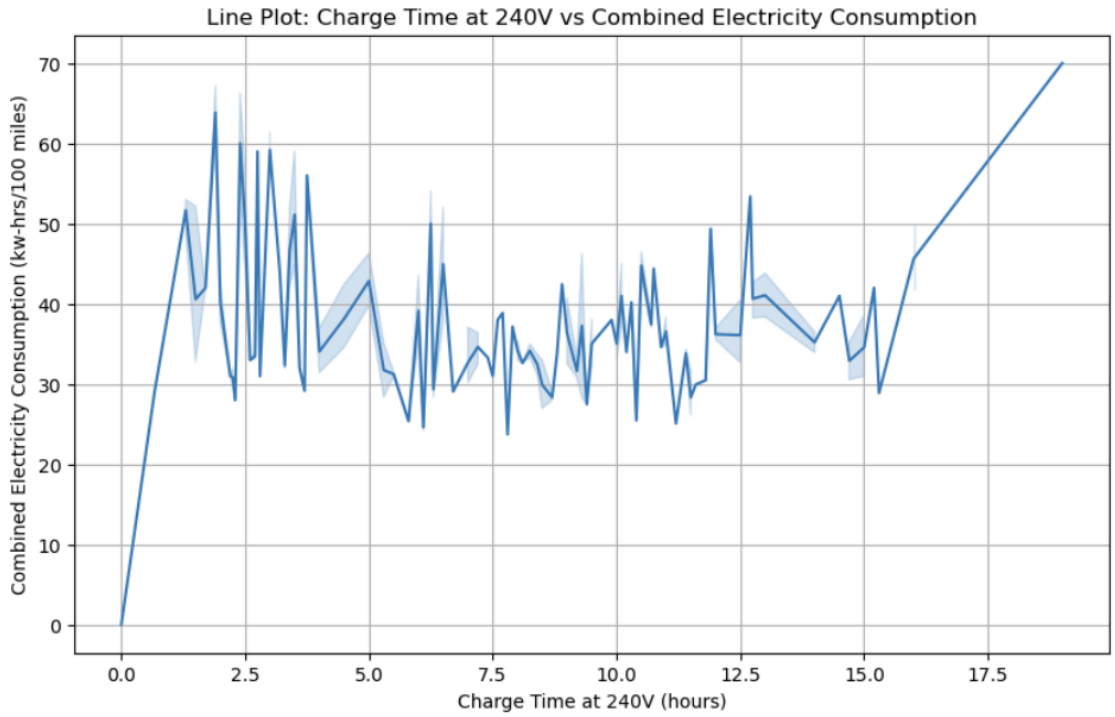
The plt.figure(figsize=(10, 6)) sets the figure size to 10x6 inches. The Seaborn scatterplot function is then used to create the scatter plot with transparency set by alpha=0.7. Matplotlib is employed to add a title, x-axis label, y-axis label, and grid to enhance the plot's clarity. Finally, plt.show() displays the scatter plot.

The visualization aims to illustrate the relationship between annual petroleum consumption and CO2 emissions for vehicles using 'Regular' fuel.

The image which is the output depicts a scatter plot graph which examines the relationship between annual petroleum consumption (in barrels) and carbon dioxide (CO2) emissions (in grams/mile) for a specific fuel type, labeled "FuelType1." The horizontal axis represents the annual petroleum consumption, and the vertical axis denotes the CO2 emissions. Each dot on the plot corresponds to a particular measurement of both variables. The data points show a positive correlation, indicating that as annual petroleum consumption increases, CO2 emissions tend to increase as well. The plot exhibits a step-like pattern in the mid-range of petroleum consumption, suggesting possible tiers or categories within the data that might correspond to different classes of vehicles or engines within FuelType1.

Line plot for potraying the relationship between Charge Time at 240V and Combined Electricity Consumption

```
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
sns.lineplot(data=df, x='charge240', y='combE')
plt.title('Line Plot: Charge Time at 240V vs Combined Electricity Consumption')
plt.xlabel('Charge Time at 240V (hours)')
plt.ylabel('Combined Electricity Consumption (kw-hrs/100 miles)')
plt.grid(True)
plt.show()
```



code utilizes Seaborn and Matplotlib to create a line plot for data in a DataFrame (presumably named df). The line plot represents the relationship between two variables: 'charge240' (charge time at 240V) on the x-axis and 'combE' (combined electricity consumption in kw-hrs/100 miles) on the y-axis.

The plt.figure(figsize=(10, 6)) sets the figure size to 10x6 inches. The Seaborn lineplot function is used to generate the line plot. Matplotlib is then employed to add a title, x-axis label, y-axis label, and grid for

better visualization. Finally, plt.show() is used to display the line plot. The visualization provides insights into the relationship between charge time at 240V and combined electricity consumption.

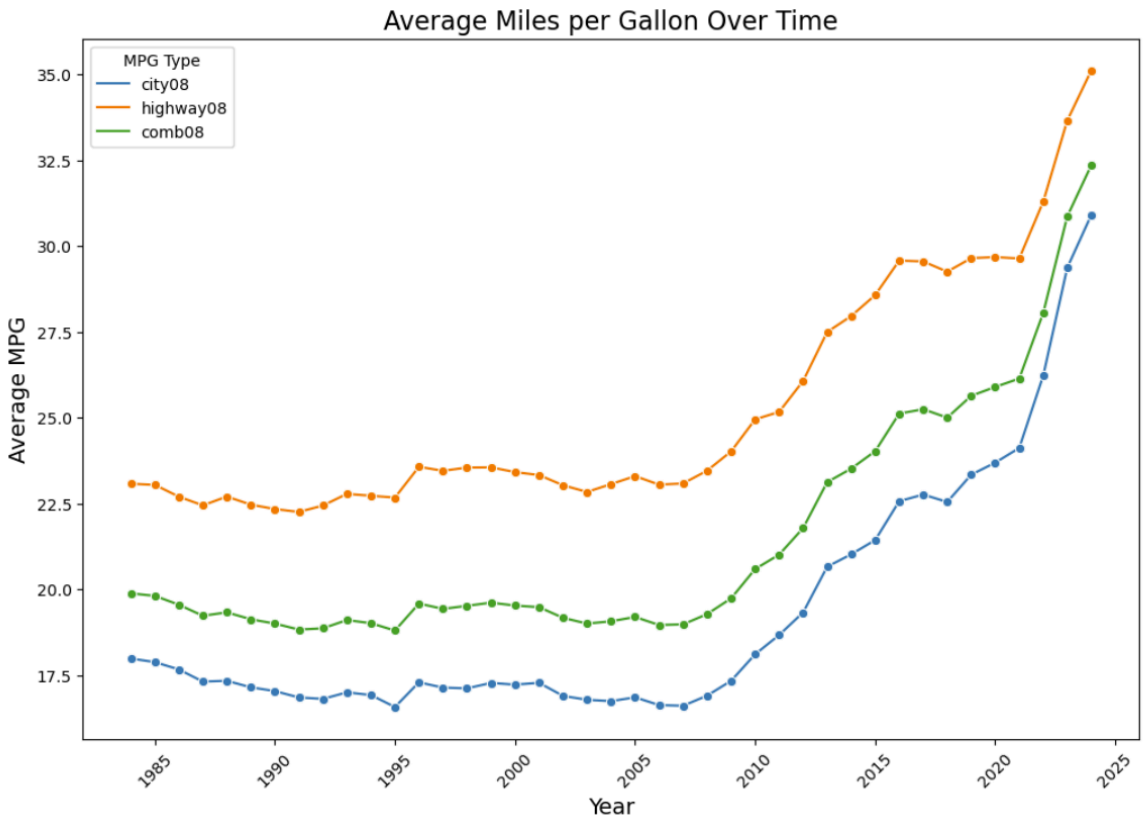
The image depicts a line plot that illustrates the relationship between the charge time in hours at 240 volts and the combined electricity consumption in kilowatt-hours per 100 miles. The horizontal axis indicates the charge time, while the vertical axis shows the electricity consumption. The plot features a line with a shaded area around it, which may represent the variability or confidence interval around the mean electricity consumption values. Initially, there is a sharp increase in electricity consumption as charge time increases, followed by fluctuations without a clear trend. Towards the end of the plot, there is a significant rise in consumption as charge time extends beyond 15 hours. This graph could be useful for understanding how charging duration at a given voltage affects energy usage in electric vehicles.

```
import pandas as pd

df['year'] = pd.to_datetime(df['year'], format='%Y')
```

```
import seaborn as sns
import matplotlib.pyplot as plt

avg_mpg_year = df.groupby('year')[['city08', 'highway08', 'comb08']].mean().reset_index()
avg_mpg_year_melted = avg_mpg_year.melt(id_vars='year', var_name='MPG Type', value_name='Average MPG')

plt.figure(figsize=(12, 8))
sns.lineplot(data=avg_mpg_year_melted, x='year', y='Average MPG', hue='MPG Type', marker='o')
plt.title('Average Miles per Gallon Over Time', fontsize=16)
plt.xlabel('Year', fontsize=14)
plt.ylabel('Average MPG', fontsize=14)
plt.xticks(rotation=45)
plt.legend(title='MPG Type')
plt.savefig('average_mpg_over_time.png')
plt.show()
```

The code first converts the 'year' column in the DataFrame df to datetime format using pd.to_datetime with the specified format '%Y'. Then, it calculates the average city, highway, and combined miles per gallon (MPG) for each unique year using groupby and mean. The resulting DataFrame is melted using melt to create a long-format DataFrame.

Subsequently, a line plot is created using Seaborn (sns.lineplot) to visualize the average MPG over time, with separate lines for city, highway, and combined MPG. The plot is customized with a specified figure size, title, axis labels, rotated x-axis labels for readability, and a legend. The plot is saved as an image ('average_mpg_over_time.png') and displayed using plt.show(). The visualization effectively illustrates how average miles per gallon have changed over time for different types of MPG.



The image shows a line graph titled "Average Miles per Gallon Over Time," which tracks the change in fuel efficiency from 1985 to around 2025 for three different types of miles per gallon (MPG) measurements: city (city08), highway (highway08), and a combined measure (comb08). Each line represents the average MPG for each category over the years. The city MPG (blue line) is consistently the lowest, the highway MPG (green line) is higher, and the combined MPG (orange line) falls between the two, reflecting an average of city and highway values. All three measures show a general upward trend, indicating improvements in fuel efficiency over time, with a particularly sharp increase after 2010. This suggests that vehicles have become significantly more fuel-efficient in recent years.

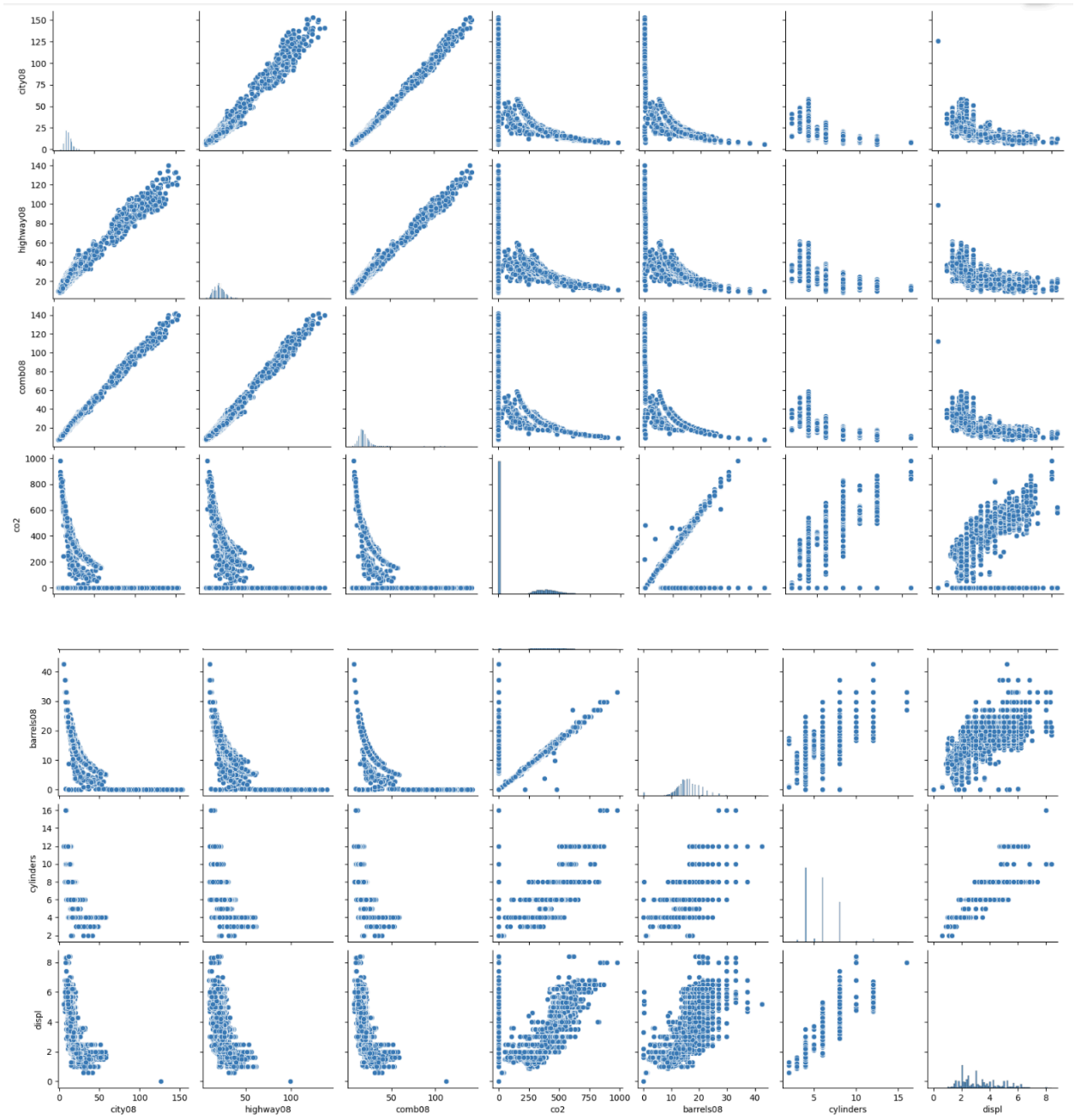
Pair Plot of Fuel Efficiency and Emissions Variables

```
import seaborn as sns
import matplotlib.pyplot as plt

numerical_columns = ['city08', 'highway08', 'comb08', 'co2', 'barrels08', 'cylinders', 'displ']

plt.tight_layout()
sns.pairplot(df[numerical_columns])
plt.show()
```

```
C:\Users\prabh\anaconda3\Lib\site-packages\seaborn\axisgrid.py:118: UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

This code utilizes Seaborn and Matplotlib to create a pair plot for a subset of numerical columns ('city08', 'highway08', 'comb08', 'co2', 'barrels08', 'cylinders', 'displ') from a DataFrame (df).

The sns.pairplot function generates a grid of scatterplots for pairwise relationships between these numerical columns, while plt.tight_layout() improves the arrangement of subplots for better readability. The resulting pair plot provides a quick visual overview of the relationships and distributions between the selected numerical variables in the dataset. It can be useful for identifying patterns, correlations, and potential outliers. Finally, plt.show() displays the pair plot.



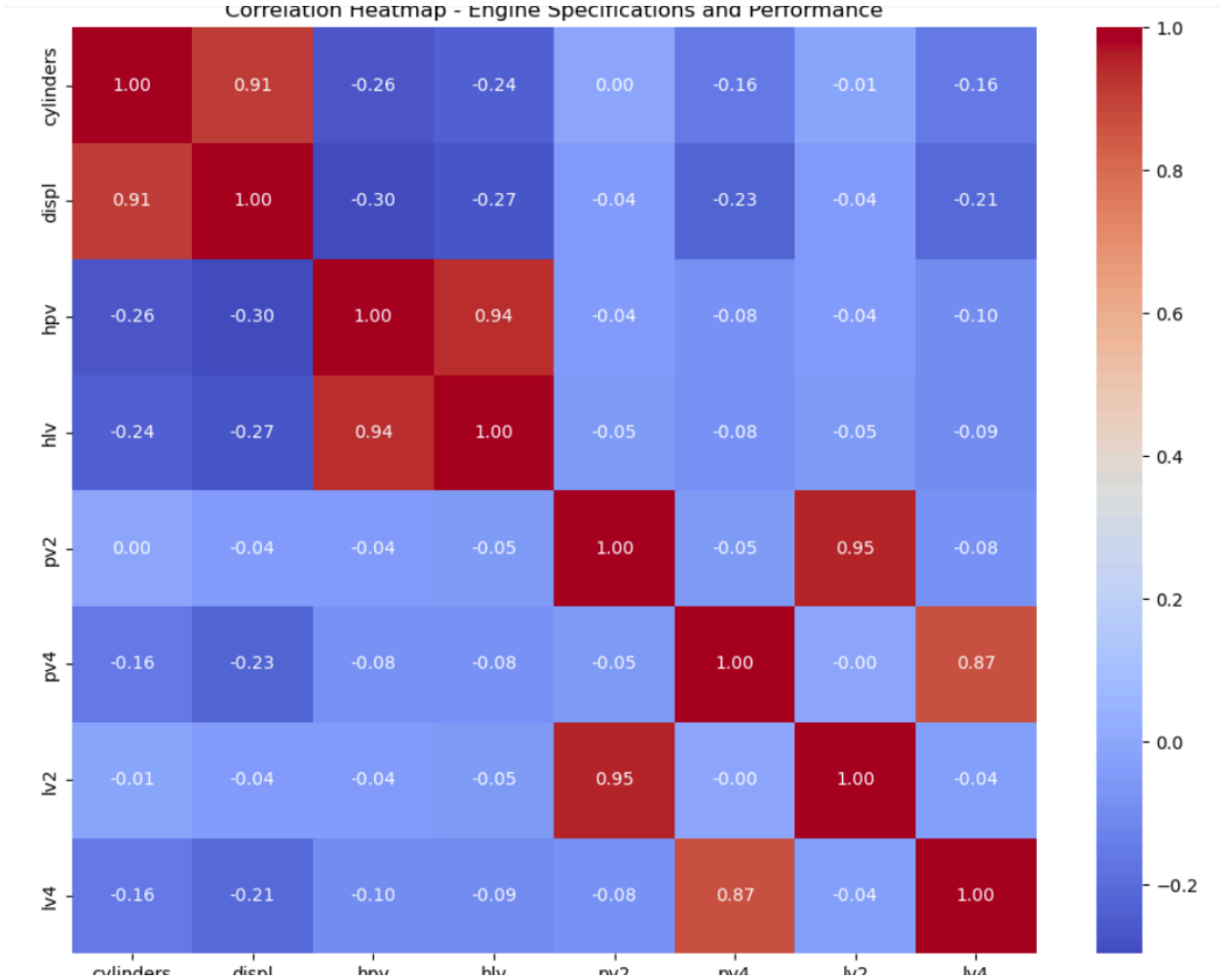
The image displays a matrix of scatter plots, commonly known as a pairs plot or a scatterplot matrix, which is used to visualize the relationship between multiple variables in a dataset. Each cell in the matrix

shows the scatter plot for the variable on the x-axis against the variable on the y-axis. Along the diagonal of the matrix, where the variable would be plotted against itself, histograms show the distribution of each variable. This type of visualization is useful for spotting correlations, patterns, and potential outliers within multivariate data. The variables compared could include city and highway miles per gallon (MPG), CO2 emissions, number of cylinders, and displacement, among others. The graphs reveal various relationships, such as positive correlations between related performance metrics and distributions of vehicle attributes like MPG and engine size.

Correlation Heatmap of the Engine Specifications and Performance

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

selected_columns = ['cylinders', 'displ', 'hpv', 'hlv', 'pv2', 'pv4', 'lv2', 'lv4']
engine_data = df[selected_columns]
correlation_matrix = engine_data.corr()

plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Heatmap - Engine Specifications and Performance')
plt.tight_layout()
plt.show()
```

This code creates a correlation heatmap using Seaborn and Matplotlib for a subset of engine-related columns ('cylinders', 'displ', 'hpv', 'hlv', 'pv2', 'pv4', 'lv2', 'lv4') from a DataFrame (df). The heatmap visually represents the pairwise correlations between these variables, with annotations displaying the correlation values. The color map ('coolwarm') helps identify the strength and direction of the correlations. The resulting visualization provides insights into the relationships among engine specifications and performance metrics.

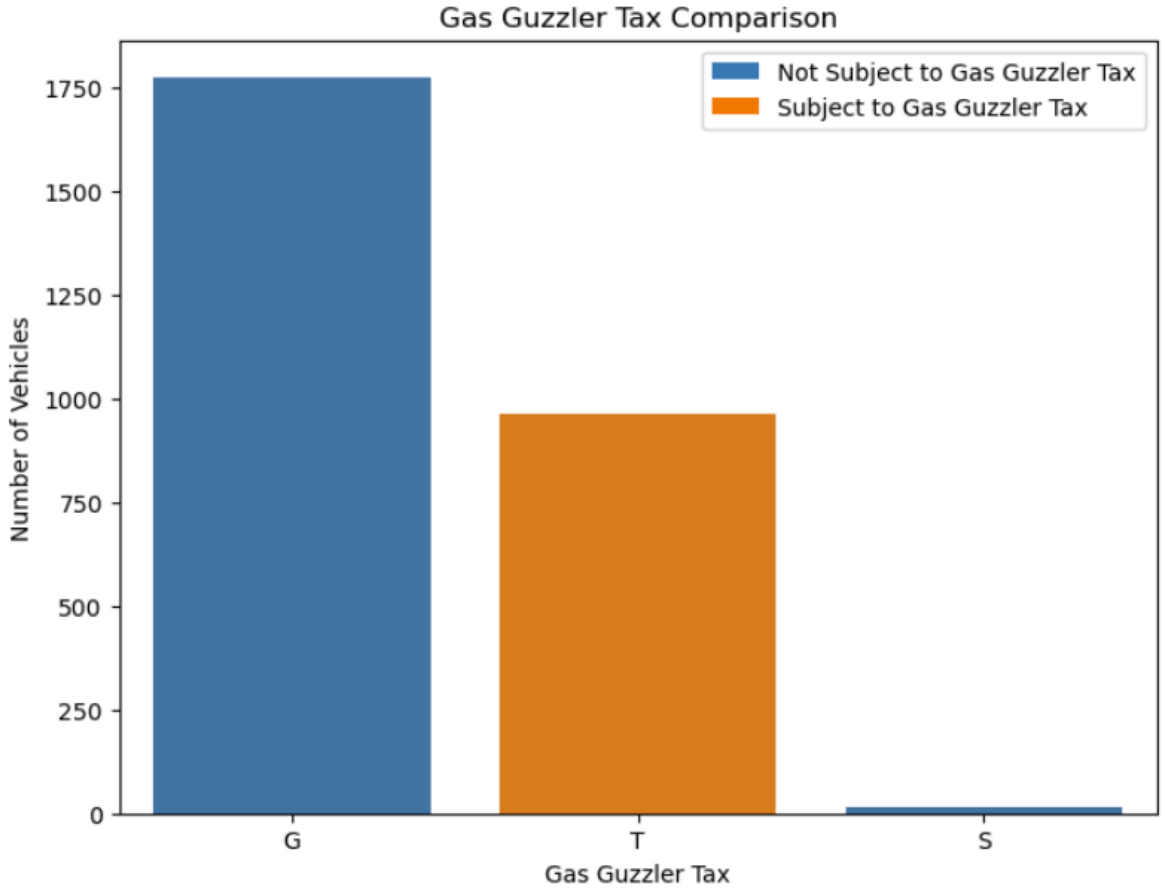


The picture depicts a correlation heatmap, a graphical representation of correlation coefficients between a set of variables, in this case, related to engine specifications and performance. The color-coded matrix shows varying shades of blue and red to indicate the strength and direction of the correlations: red shades for positive correlations and blue shades for negative correlations. Values close to 1.0 suggest a strong positive correlation, values close to -1.0 indicate a strong negative correlation, and values around 0.0 show little to no linear relationship. This heatmap provides a visual summary of how different engine and performance characteristics, such as the number of cylinders, displacement, and different types of miles per gallon (MPG) ratings, interrelate, which is particularly useful for identifying potential predictors in engine performance analysis.

Bar Plot of Gas Guzzler Tax Comparison

```
import seaborn as sns
import matplotlib.pyplot as plt

guzzler_counts = df['guzzler'].value_counts()
colors = ['#1f77b4', '#ff7f0e']

plt.figure(figsize=(8, 6))
sns.barplot(x=guzzler_counts.index, y=guzzler_counts.values, palette=colors)
plt.xlabel('Gas Guzzler Tax')
plt.ylabel('Number of Vehicles')
plt.title('Gas Guzzler Tax Comparison')
legend_labels = ['Not Subject to Gas Guzzler Tax', 'Subject to Gas Guzzler Tax']
legend_patches = [plt.Rectangle((0,0),1,1,fc=color, edgecolor='none') for color in colors]
plt.legend(legend_patches, legend_labels)
plt.show()
```

code creates a bar plot using Seaborn and Matplotlib to compare the counts of vehicles subject to and not subject to the Gas Guzzler Tax. It starts by calculating the frequency of each unique value in the 'guzzler' column. The bar plot is then generated with different colors for each category. Matplotlib is used to set the figure size, add labels to the axes and the plot title. Additionally, a legend is included to clarify the colors representing 'Not Subject to Gas Guzzler Tax' and 'Subject to Gas Guzzler Tax'. The resulting visualization offers a clear comparison of the number of vehicles falling into each category regarding the Gas Guzzler Tax.



Note: The entries marked as S are also subject to Gas Guzzler Tax

The picture shows a bar chart titled "Gas Guzzler Tax Comparison," which compares the number of vehicles that are subject to the Gas Guzzler Tax against those that are not. There are three categories labeled "G," "T," and "S." The "G" category has a significantly higher number of vehicles that are not subject to the Gas Guzzler Tax, indicated by a tall blue bar. The "T" category represents vehicles that are subject to the tax, shown by a shorter orange bar. The "S" category is noted in the chart's footnote to also be subject to the Gas Guzzler Tax but does not have a corresponding bar, suggesting there may be no vehicles in this dataset under that classification. This chart provides a clear visual representation of the differences in the number of vehicles incurring the Gas Guzzler Tax.

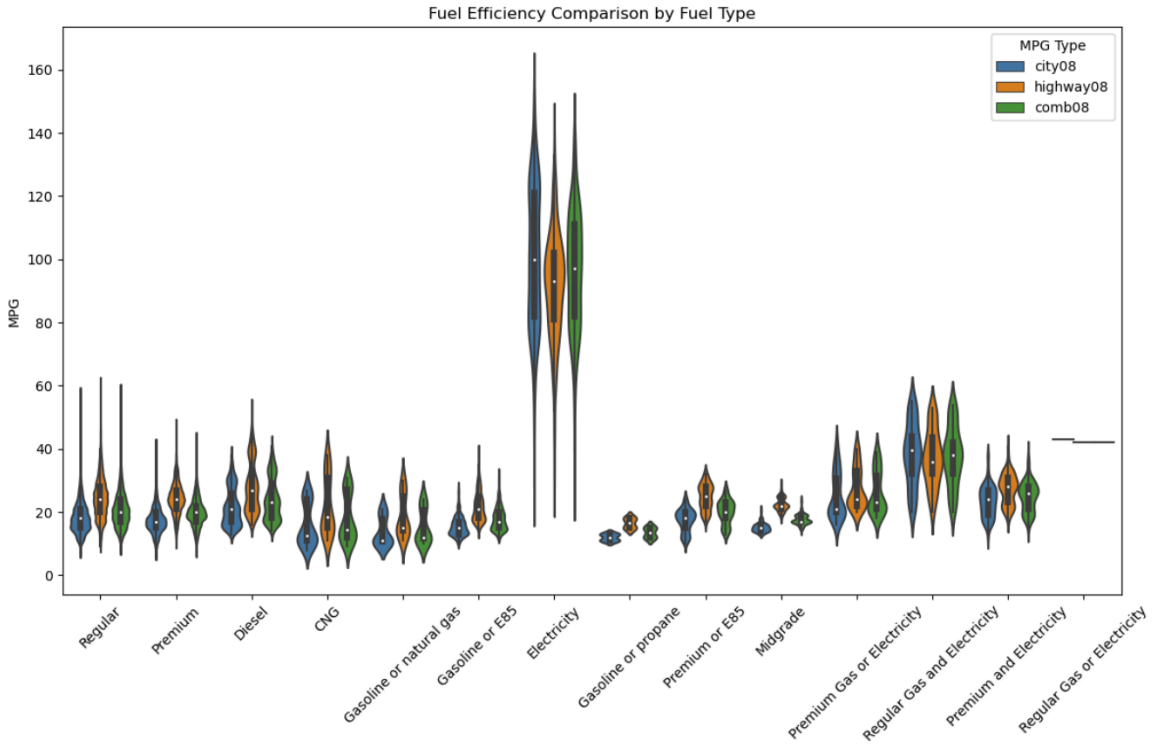
Violin Plot of Fuel Efficiency Comparison by Fuel Type:

```
import seaborn as sns
import matplotlib.pyplot as plt

filtered_df = df[(df['city08'] != -1) & (df['highway08'] != -1) & (df['comb08'] != -1)]
mpg_data = filtered_df[['fuelType', 'city08', 'highway08', 'comb08']]
mpg_data = pd.melt(mpg_data, id_vars='fuelType', var_name='MPG_Type', value_name='MPG')

plt.figure(figsize=(12, 8))
sns.violinplot(data=mpg_data, x='fuelType', y='MPG', hue='MPG_Type')
plt.xlabel('Fuel Type')
plt.ylabel('MPG')
plt.title('Fuel Efficiency Comparison by Fuel Type')
plt.legend(title='MPG Type')
plt.xticks(rotation=45)
plt.tight_layout()
plt.savefig('fuel_efficiency_comparison.png')
plt.show()
```

The code begins by filtering out rows with invalid MPG values (-1) in the 'city08', 'highway08', and 'comb08' columns, creating a new DataFrame named filtered_df. Subsequently, a subset of columns ('fuelType', 'city08', 'highway08', 'comb08') is selected from the filtered data, and the DataFrame is reshaped using pd.melt to create a long-format DataFrame (mpg_data).

A Seaborn violin plot is then generated, visualizing the distribution of fuel efficiency (MPG) across different fuel types, with separate components for city, highway, and combined MPG. Matplotlib is used to customize the plot by adding labels to the axes, a title, a legend indicating the MPG type, and rotating x-axis labels for better readability.

Finally, the plot is saved as an image ('fuel_efficiency_comparison.png') and displayed using plt.show(). The resulting visualization offers a detailed comparison of fuel efficiency distributions among various fuel types.



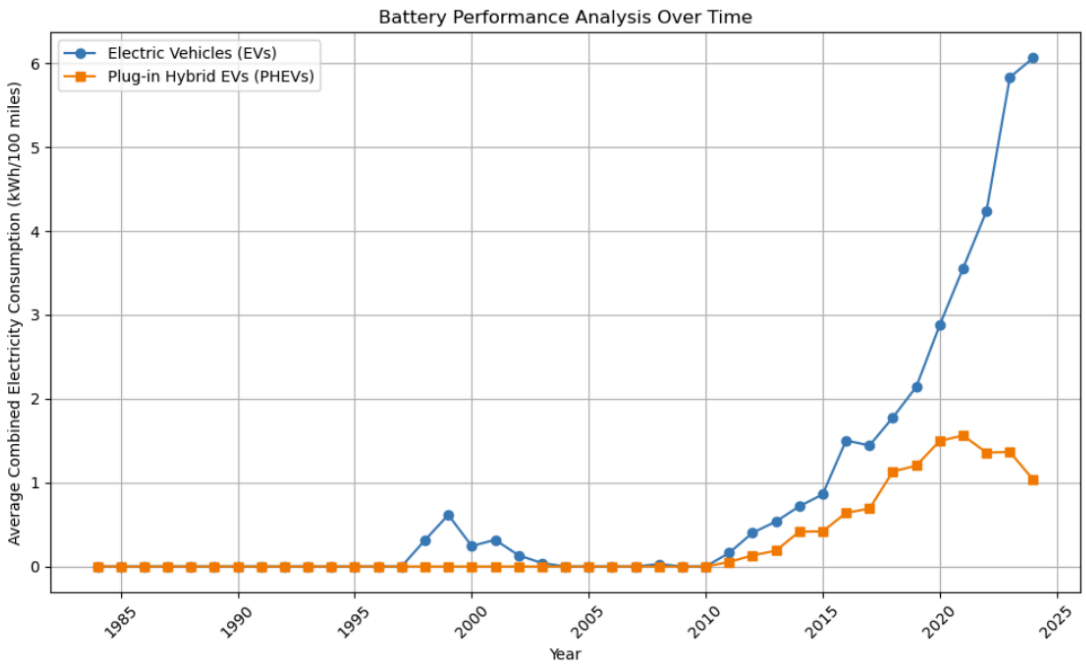
The picture shows a violin plot comparing fuel efficiency, measured in miles per gallon (MPG), across different fuel types and MPG categories—city (city08), highway (highway08), and a combined (comb08) efficiency. Violin plots combine box plots with a kernel density estimation to show the distribution of the data points. For each fuel type, there are three corresponding violin shapes, each representing one of the MPG categories. These violins illustrate both the spread and the probability density of the data. The chart covers a range of fuel types, including regular, premium, diesel, CNG (Compressed Natural Gas), and electricity, among others. Certain fuel types, like electricity, show high MPG efficiencies, as indicated by the extended violins, while others, such as regular and premium, have broader distributions, reflecting a wider range of vehicle efficiency. This visualization is useful for comparing the fuel efficiency across different types of vehicle fuels and understanding the variability within each type.

Plotting Battery Performance Analysis Over Time using a Line Chart:

```
import pandas as pd
import matplotlib.pyplot as plt

battery_performance = df.groupby('year')[['combE', 'phevComb']].mean().reset_index()

plt.figure(figsize=(10, 6))
plt.plot(battery_performance['year'], battery_performance['combE'], label='Electric Vehicles (EVs)', marker='o')
plt.plot(battery_performance['year'], battery_performance['phevComb'], label='Plug-in Hybrid EVs (PHEVs)', marker='s')
plt.xlabel('Year')
plt.ylabel('Average Combined Electricity Consumption (kWh/100 miles)')
plt.title('Battery Performance Analysis Over Time')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.xticks(rotation=45)
plt.show()
```

This code starts by aggregating data from the DataFrame df based on the 'year' column. It calculates the mean values for 'combE' (electric vehicles) and 'phevComb' (plug-in hybrid electric vehicles), creating a new DataFrame named battery_performance.

For visualization, a line plot is generated with Matplotlib, illustrating the average combined electricity consumption over time for electric vehicles (EVs) and plug-in hybrid EVs (PHEVs). The plot includes labeled axes, a title, a legend to distinguish between EVs and PHEVs, a grid for better readability, and rotated x-axis labels. The resulting visualization allows for a clear analysis of trends in battery performance for these two types of electric vehicles over the years.



The picture shows a line graph titled "Battery Performance Analysis Over Time," which tracks the average combined electricity consumption for Electric Vehicles (EVs) and Plug-in Hybrid EVs (PHEVs) from around 1985 to 2025. The blue line represents EVs and shows a relatively stable consumption rate until about 2010, after which there is a significant increase, reaching a peak consumption of over 6 kWh/100 miles by 2025. The orange line for PHEVs shows a more gradual increase in consumption over the years, remaining consistently lower than the consumption of EVs throughout the entire period. This graph suggests that while the efficiency of PHEVs has been improving steadily, the electricity consumption of EVs has increased significantly in recent years, potentially due to factors like increased battery capacity or changes in vehicle design and usage.

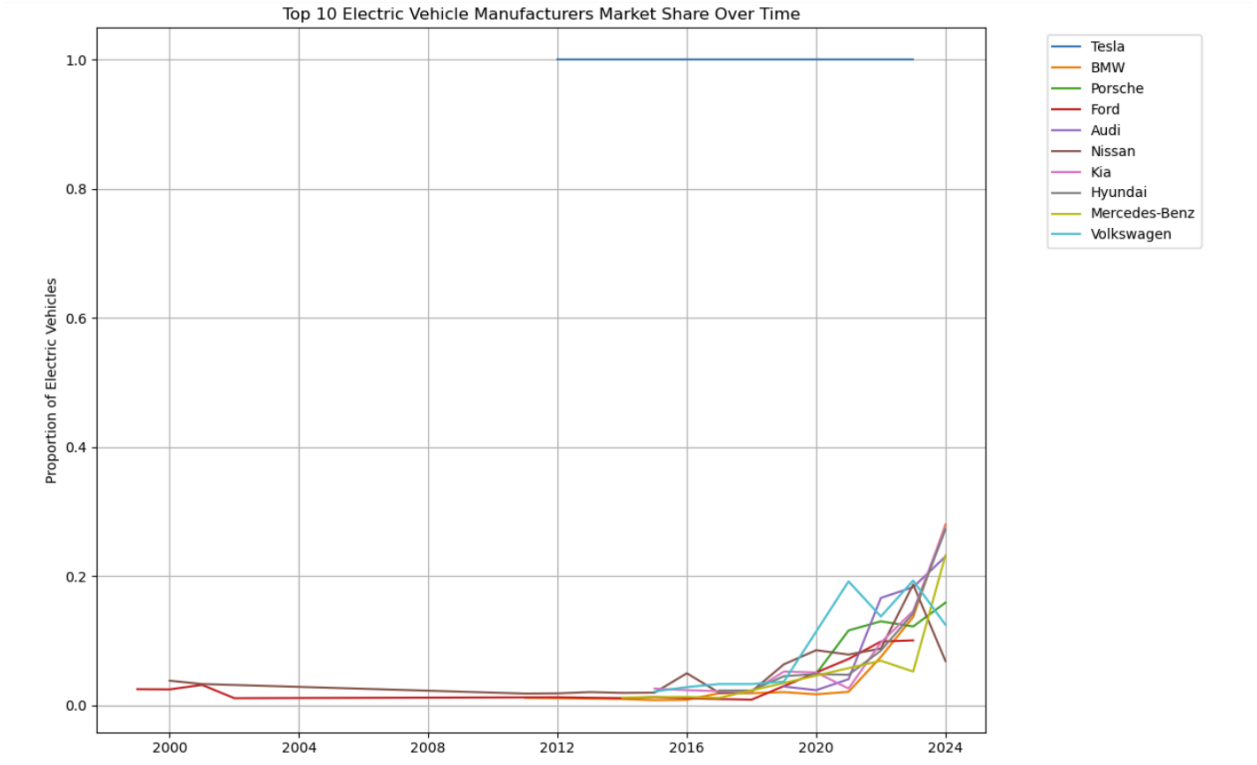
Top 10 Electric Vehicle Manufacturers Market Share Over Time

```
import pandas as pd
import matplotlib.pyplot as plt

grouped = df[df['fuelType'] == 'Electricity'].groupby(['year', 'make']).size().reset_index(name='EV_Count')
total_production = df.groupby(['year', 'make']).size().reset_index(name='Total_Count')
merged_data = pd.merge(grouped, total_production, on=['year', 'make'])
merged_data['EV_Proportion'] = merged_data['EV_Count'] / merged_data['Total_Count']
top_10_manufacturers = merged_data.groupby('make')['EV_Count'].sum().nlargest(10).index
top_10_data = merged_data[merged_data['make'].isin(top_10_manufacturers)]

plt.figure(figsize=(12, 8))
for make in top_10_manufacturers:
    plt.plot(top_10_data[top_10_data['make'] == make]['year'],
             top_10_data[top_10_data['make'] == make]['EV_Proportion'],
             label=make)

plt.xlabel('Year')
plt.ylabel('Proportion of Electric Vehicles')
plt.title('Top 10 Electric Vehicle Manufacturers Market Share Over Time')
plt.legend(loc='upper right', bbox_to_anchor=(1.25, 1))
plt.grid(True)
plt.tight_layout()
plt.show()
```

code analyzes and visualizes the market share of the top 10 electric vehicle (EV) manufacturers over time. It starts by preparing the data, filtering for rows with 'fuelType' as 'Electricity', grouping by 'year' and 'make', and calculating counts for EVs and total production. The resulting data is merged, and the proportion of electric vehicles is computed.

The top 10 manufacturers are identified based on the sum of EV counts. The code then creates a line plot for each of the top 10 manufacturers, showing the proportion of electric vehicles over the years. Matplotlib is used to customize the plot with labels for the axes, a title, and a legend indicating each manufacturer. The legend is positioned outside the plot for clarity. Grid lines are added for better readability, and the layout is adjusted for a more organized presentation.

The resulting visualization provides insights into how the market share of the top electric vehicle manufacturers has evolved over time.



The picture is a line graph titled "Top 10 Electric Vehicle Manufacturers Market Share Over Time," which depicts the market share dynamics of various electric vehicle (EV) manufacturers from 2000 to 2024. Each line represents a different manufacturer, including Tesla, BMW, Porsche, Ford, Audi, Nissan, Kia, Hyundai, Mercedes-Benz, and Volkswagen. For most of the timeline, the market shares are relatively low and close together, suggesting a more even distribution among the manufacturers. However, starting around 2016, there's a clear uptick in market share for several manufacturers, with Tesla's line showing a particularly steep rise, indicating a dominant position in the market. The graph captures the growth of the EV market and the changing competitive landscape, with Tesla emerging as a significant leader by 2024.

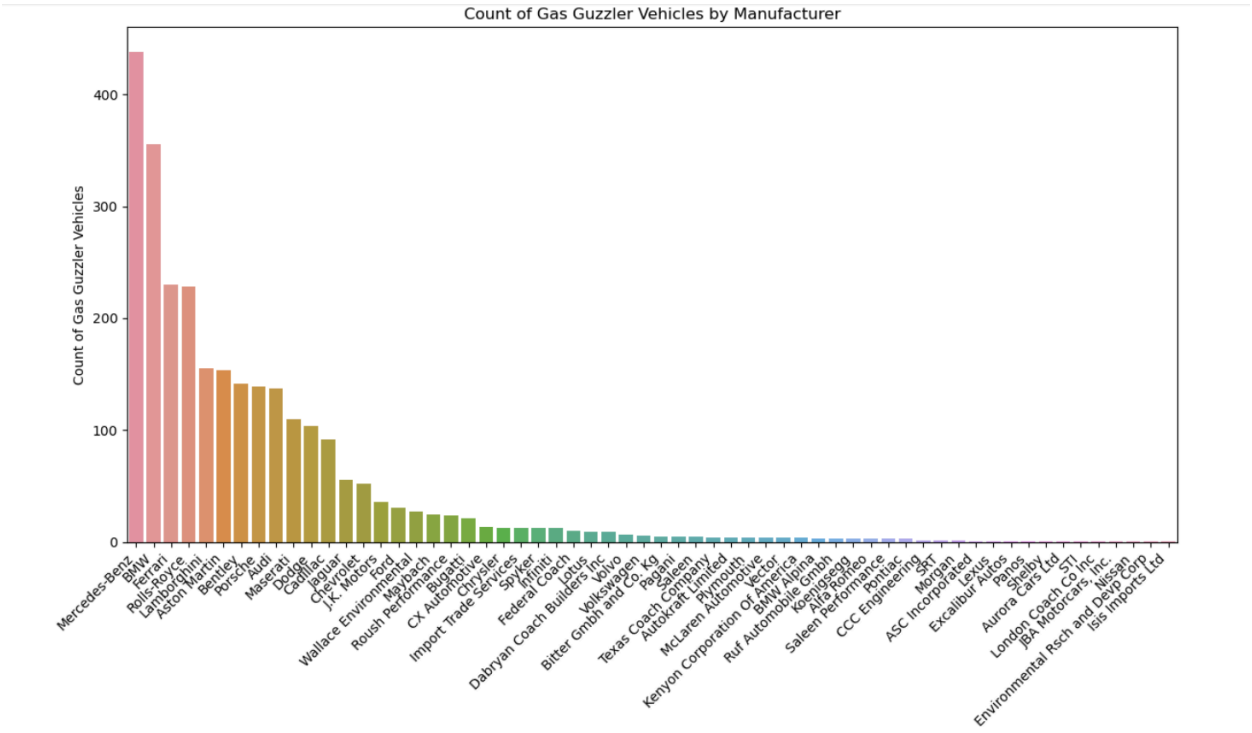
Count plot of Gas Guzzler Vehicles by Manufacturer:

```
import seaborn as sns
import matplotlib.pyplot as plt

guzzler_data = df[df['guzzler'].isin(['G', 'T'])]

plt.figure(figsize=(12, 8))
sns.countplot(data=guzzler_data, x='make', order=guzzler_data['make'].value_counts().index)
plt.xlabel('Manufacturer')
plt.ylabel('Count of Gas Guzzler Vehicles')
plt.title('Count of Gas Guzzler Vehicles by Manufacturer')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

code generates a count plot using Seaborn and Matplotlib to illustrate the distribution of gas guzzler vehicles across different manufacturers. It begins by filtering the DataFrame df to include only rows where the 'guzzler' column has values 'G' or 'T', representing gas guzzler vehicles. The resulting subset is named guzzler_data.

The count plot is created using Seaborn's countplot, where the x-axis represents the manufacturers and the y-axis represents the count of gas guzzler vehicles. The bars are ordered based on the descending count of gas guzzler vehicles, and the plot is customized with labels for the axes, a title, and rotated x-axis labels for better visibility. The layout is adjusted for a more organized presentation.

The visualization effectively communicates the distribution of gas guzzler vehicles among different manufacturers, aiding in the identification of manufacturers with higher counts of such vehicles.



The picture shows a horizontal bar chart that ranks manufacturers by the count of gas guzzler vehicles they produce. The length of each bar represents the number of gas guzzler vehicles attributed to each manufacturer, with the colors gradually changing from red to green as the count decreases. The manufacturer with the most gas guzzler vehicles has a count close to 450, and as we move from left to right across the chart, the count per manufacturer decreases. This chart visually emphasizes the manufacturers with the highest number of gas guzzler vehicles, indicating which manufacturers contribute most to this segment of the market, with the chart showing a clear descending order from the highest count on the left to the lowest on the right.

Comparison of Passenger and Luggage Volumes using a Scatterplot:

```
import seaborn as sns
import matplotlib.pyplot as plt

filtered_df = df.dropna(subset=['pv2', 'pv4', 'lv2', 'lv4'])

sns.scatterplot(data=filtered_df, x='pv2', y='lv2', label='2-Door Vehicles', alpha=0.7, color='blue')
sns.scatterplot(data=filtered_df, x='pv4', y='lv4', label='4-Door Vehicles', alpha=0.7, color='red')

plt.xlabel('Passenger Volume (cubic feet)')
plt.ylabel('Luggage Volume (cubic feet)')
plt.title('Comparison of Passenger and Luggage Volumes')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

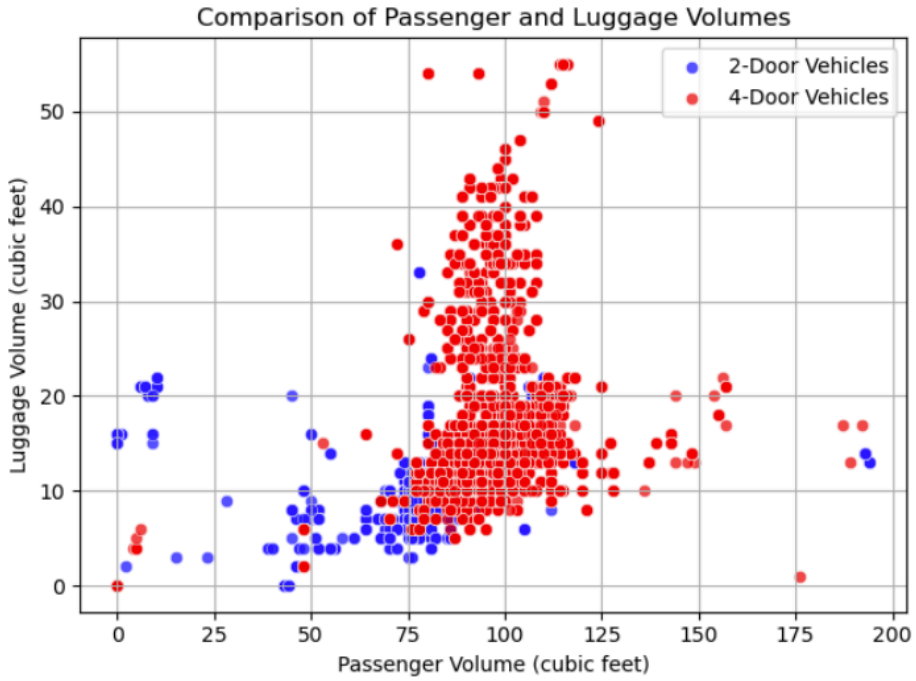
This code generates a scatter plot using Seaborn and Matplotlib to visually compare passenger and luggage volumes for 2-door and 4-door vehicles. The code starts by filtering out rows with missing values in the relevant columns ('pv2', 'pv4', 'lv2', 'lv4') from the DataFrame df.

The scatter plot is then created using Seaborn's scatterplot function twice:

- The first set of points represents 2-door vehicles, with 'pv2' on the x-axis and 'lv2' on the y-axis. Points are marked in blue with 70% transparency.
- The second set represents 4-door vehicles, with 'pv4' on the x-axis and 'lv4' on the y-axis. Points are marked in red with 70% transparency.

Matplotlib is used to add labels to the x and y-axes, a title to the plot, and a legend to distinguish between 2-door and 4-door vehicles. Grid lines are included for better visualization, and the layout is adjusted for a cleaner presentation.

The resulting scatter plot provides a clear visual comparison of passenger and luggage volumes for these two vehicle categories.



The picture is a scatter plot titled "Comparison of Passenger and Luggage Volumes," contrasting the interior passenger volume against luggage space for two categories of vehicles: 2-door and 4-door. The horizontal axis measures passenger volume in cubic feet, while the vertical axis measures luggage volume in cubic feet. Blue dots represent 2-door vehicles, and red dots represent 4-door vehicles. Most 2-door vehicles cluster in the lower range of both passenger and luggage volume, whereas 4-door vehicles typically offer more passenger space, with many also providing greater luggage volume. The spread of red dots indicates a broader variation in the interior and luggage space among 4-door vehicles compared to 2-door models. This visualization aids in understanding how the number of doors on a vehicle might correlate with the available interior and cargo space.