assignment_2

November 25, 2024

[215]:
```python
from IPython.display import Image
Image(filename='images.png')
```

[215]:

Electrical and Computer Engineering Department

Machine Learning and Data Science - ENCS5341

Assignment #2 ____
Prepared By: Tala Abahra 1201002, Sondos Shahin 1200166

Instructor: Dr. Yazan Abu Farha Date: Oct 30, 2024

Topic: Machine Learning Assignment: 1 https://github.com/sondosshahin/Machine-Learning-Project-Regression-Analysis-and-Model-Selection-

0.1 1 - Import Dataset YallaMotors

The main objective of this dataset is to predict car prices, making it ideal for developing regression models to understand the relationship between various features (e.g., car make, model, year, mileage, engine size, etc.) and the target variable (car price).

[220]:
```python
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
missing_values = [" ", "NA", "N/A", "N A", "NaN"]
data = pd.read_csv("cars.csv", na_values=missing_values)
data.head()
```

[220]:
```
                         car name                  price engine_capacity  \
0            Fiat 500e 2021 La Prima                   TBD             0.0
1      Peugeot Traveller 2021 L3 VIP           SAR 140,575             2.0
2   Suzuki Jimny 2021 1.5L Automatic            SAR 98,785             1.5
3     Ford Bronco 2021 2.3T Big Bend           SAR 198,000             2.3
4      Honda HR-V 2021 1.8 i-VTEC LX  Orangeburst Metallic             1.8

        cylinder horse_power  top_speed     seats    brand country
0  N/A, Electric      Single  Automatic       150     fiat     ksa
1              4         180   8 Seater       8.8  peugeot     ksa
2              4         102        145  4 Seater   suzuki     ksa
3              4         420   4 Seater       7.5     ford     ksa
4              4         140        190  5 Seater    honda     ksa
```

[222]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6308 entries, 0 to 6307
Data columns (total 9 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   car name         6308 non-null   object
 1   price            6308 non-null   object
 2   engine_capacity  6308 non-null   object
 3   cylinder         5684 non-null   object
 4   horse_power      6308 non-null   object
 5   top_speed        6265 non-null   object
 6   seats            6205 non-null   object
 7   brand            6308 non-null   object
 8   country          6308 non-null   object
dtypes: object(9)
memory usage: 443.7+ KB
```

[224]:
```
data.describe()
```

[224]:
```
                                 car name price engine_capacity cylinder  \
count                                6308  6308            6308     5684
unique                               2546  3395             129       10
top     Mercedes-Benz C-Class 2022 C 300   TBD             2.0        4
freq                                   10   437            1241     2856

       horse_power top_speed     seats         brand country
count         6308      6265      6205          6308    6308
```

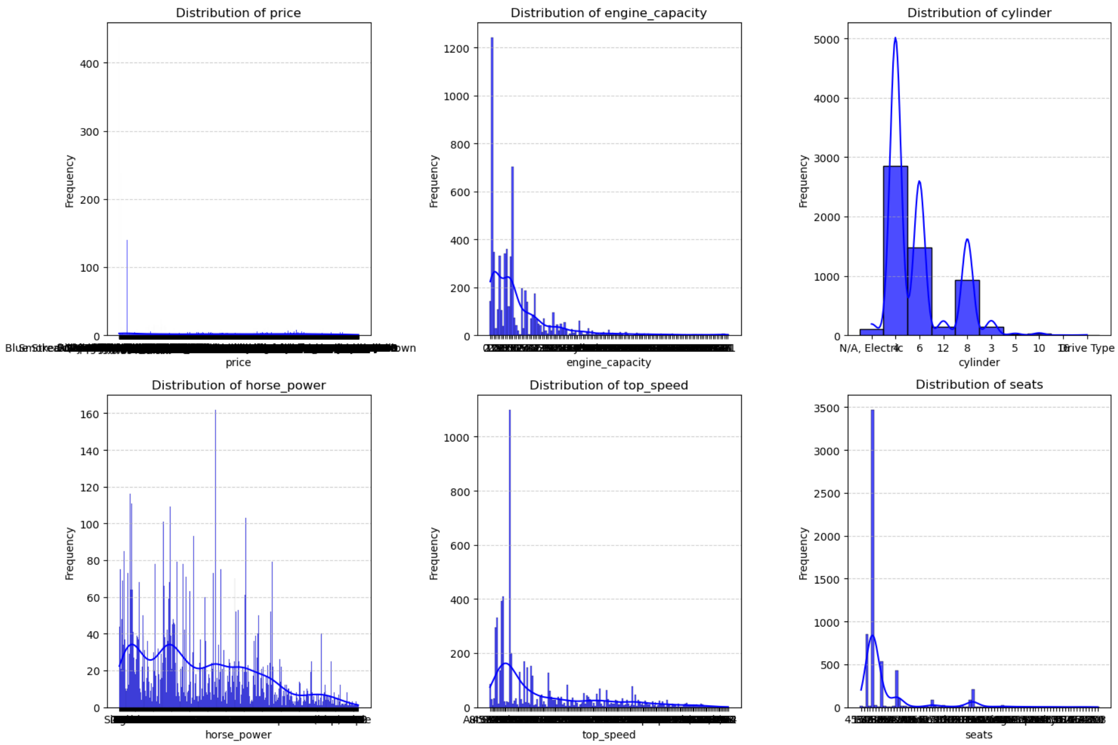
```
unique          330       168         81              82       7
top             150       250   5 Seater   mercedes-benz     uae
freq            162      1100       3471             560    1248
```

[226]:
```
from sklearn.preprocessing import LabelEncoder
# Print unique values to check
print(data['brand'].unique())
print(data['country'].unique())
print(data['car name'].unique())

#encode categorical features
label_encoder = LabelEncoder()     # Initialize the LabelEncoder
categorical_features = ['brand', 'country', 'car name']
for column in categorical_features:
    # Fit the LabelEncoder and transform the column
    data[column] = label_encoder.fit_transform(data[column])
    print(data[column])

# Display encoded dataset
print("Dataset after Encoding:")
print(data)

# 2. Visualize numeric columns
numeric_columns = ['price', 'engine_capacity', 'cylinder', 'horse_power',␣
↪'top_speed', 'seats']

# Create distribution plots for numeric columns
plt.figure(figsize=(15, 10))  # Set figure size
for i, column in enumerate(numeric_columns, 1):
    plt.subplot(2, 3, i)  # Arrange subplots in 2 rows and 3 columns
    sns.histplot(data[column], kde=True, bins=30, color='blue', alpha=0.7)  #␣
↪Histogram with KDE
    plt.title(f"Distribution of {column}", fontsize=12)  # Add a title
    plt.xlabel(column, fontsize=10)  # Label x-axis
    plt.ylabel("Frequency", fontsize=10)  # Label y-axis
    plt.grid(axis='y', linestyle='--', alpha=0.6)  # Add grid for better␣
↪readability

# Adjust layout to prevent overlap
plt.tight_layout()
plt.show()
```

```
['fiat' 'peugeot' 'suzuki' 'ford' 'honda' 'renault' 'aston-martin' 'gac'
 'toyota' 'genesis' 'hyundai' 'lincoln' 'mg' 'chevrolet' 'mercedes-benz'
 'kia' 'volkswagen' 'land-rover' 'lotus' 'volvo' 'porsche' 'mini'
 'lamborghini' 'nissan' 'mclaren' 'changan' 'great-wall' 'bmw'
 'rolls-royce' 'audi' 'infiniti' 'ram' 'chrysler' 'gmc' 'borgward' 'jeep'
```

```
 'alfa-romeo' 'chery' 'skoda' 'lexus' 'jaguar' 'maxus' 'cadillac'
 'ferrari' 'mazda' 'mitsubishi' 'bestune' 'jetour' 'hongqi' 'maserati'
 'geely' 'byd' 'Foton' 'subaru' 'haval' 'isuzu' 'ssang-yong' 'dodge'
 'bentley' 'bugatti' 'opel' 'zotye' 'soueast ' 'dorcen' 'citroen'
 'brilliance' 'seat' 'proton' 'soueast' 'ds' 'jac' 'lada' 'kinglong'
 'baic' 'morgan' 'mahindra' 'tata' 'dfm' 'acura' 'abarth' 'zna' 'tesla']
['ksa' 'egypt' 'bahrain' 'qatar' 'oman' 'kuwait' 'uae']
['Fiat 500e 2021 La Prima' 'Peugeot Traveller 2021 L3 VIP'
 'Suzuki Jimny 2021 1.5L Automatic' ...
 'BMW M8 Convertible 2021 4.4T V8 Competition xDrive (625 Hp)'
 'BMW M8 Coupe 2021 4.4T V8 Competition xDrive (625 Hp)'
 'Lamborghini Aventador Ultimae 2022 LP 780-4']
0       25
1       62
2       74
3       26
4       33
        ..
6303     7
6304    24
6305    67
6306    45
6307     7
Name: brand, Length: 6308, dtype: int32
0       2
1       2
2       2
3       2
4       2
       ..
6303    6
6304    6
6305    6
6306    6
6307    6
Name: country, Length: 6308, dtype: int32
0        564
1       1980
2       2235
3        574
4        811
        ...
6303     333
6304     552
6305    2119
6306    1246
6307     334
Name: car name, Length: 6308, dtype: int32
```

```
Dataset after Encoding:
      car name                price engine_capacity       cylinder  \
0          564                  TBD             0.0  N/A, Electric
1         1980          SAR 140,575             2.0              4
2         2235           SAR 98,785             1.5              4
3          574          SAR 198,000             2.3              4
4          811  Orangeburst Metallic            1.8              4
...        ...                  ...             ...            ...
6303       333         DISCONTINUED             6.8              8
6304       552        AED 1,766,100             4.0              8
6305      2119        AED 1,400,000             6.6             12
6306      1246        AED 1,650,000             6.5            NaN
6307       334         DISCONTINUED             6.8              8

     horse_power  top_speed      seats  brand  country
0         Single  Automatic        150     25        2
1            180   8 Seater        8.8     62        2
2            102        145   4 Seater     74        2
3            420   4 Seater        7.5     26        2
4            140        190   5 Seater     33        2
...          ...        ...        ...    ...      ...
6303         505        296   5 Seater      7        6
6304          25        800  Automatic     24        6
6305         624        250   4 Seater     67        6
6306         740        350   2 Seater     45        6
6307         530        305   5 Seater      7        6

[6308 rows x 9 columns]
```



[70]:
```python
for clm in ['car␣
↪name','brand','country','price','engine_capacity','cylinder','horse_power','top_speed','seats
↪
    print(f'Name: {clm} dtype: {data[clm].dtype}\n')
    print(f'{data[clm].value_counts()}\n')
    print(('-' * 80) + '\n\n')
```

```
Name: car name dtype: int32

car name
1625    10
564      7
2001     7
1995     7
1204     7
        ..
1029     1
903      1
488      1
1105     1
1251     1
Name: count, Length: 2546, dtype: int64

--------------------------------------------------------------------------------
```

```
Name: brand dtype: int32

brand
55    560
5     398
9     394
77    378
26    323
     ...
75      2
71      2
13      2
20      1
12      1
Name: count, Length: 82, dtype: int64
```

```
Name: country dtype: int32

country
6    1248
2     996
3     932
5     925
4     910
0     906
1     391
Name: count, dtype: int64
```

```
Name: price dtype: object

price
TBD                 437
Following           238
DISCONTINUED        140
Follow               27
Grigio Maratea       23
                   ...
BHD 23,900            1
BHD 24,300            1
BHD 24,100            1
```

```
BHD 24,700          1
AED 1,650,000       1
Name: count, Length: 3395, dtype: int64
```

```
Name: engine_capacity dtype: object

engine_capacity
2.0     1241
3.0      703
3.5      359
1.5      347
4.0      340
        ...
3342       1
2476       1
4400       1
3470       1
1595       1
Name: count, Length: 129, dtype: int64
```

```
Name: cylinder dtype: object

cylinder
4                 2856
6                 1480
8                  924
3                  139
12                 136
N/A, Electric      107
10                  21
5                   17
Drive Type           3
16                   1
Name: count, dtype: int64
```

```
Name: horse_power dtype: object

horse_power
150    162
```

```
355    116
400    111
184    109
300    103
      ...
87       1
126      1
394      1
236      1
720      1
Name: count, Length: 330, dtype: int64
```

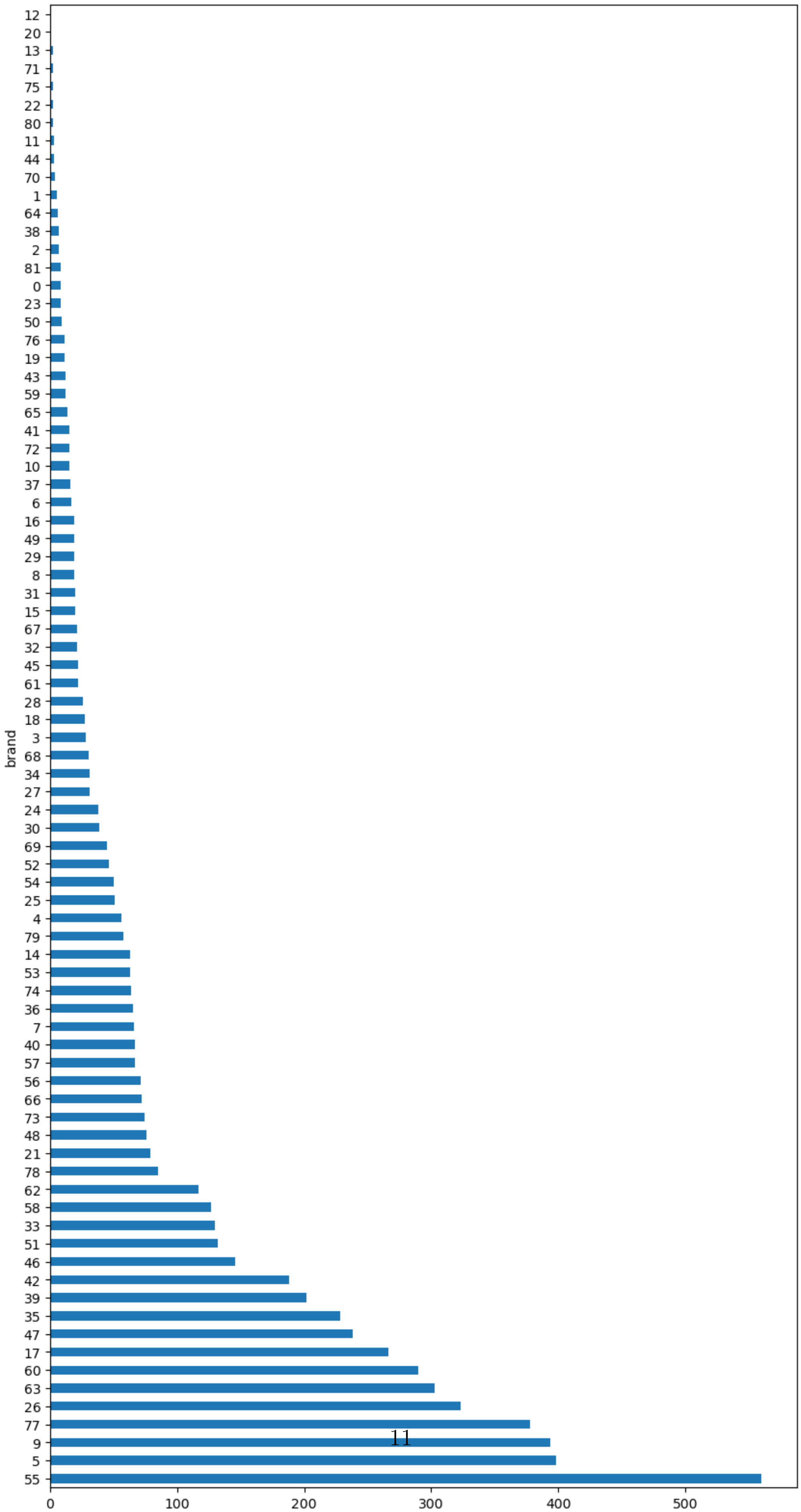
```
Name: top_speed dtype: object

top_speed
250    1100
180     410
200     392
170     332
190     294
       ...
307       1
130       1
966       1
262       1
324       1
Name: count, Length: 168, dtype: int64
```

```
Name: seats dtype: object

seats
5 Seater    3471
4 Seater     847
7 Seater     532
2 Seater     428
8 Seater     211
            ...
24.1           1
12.3           1
230            1
220            1
2.8            1
```

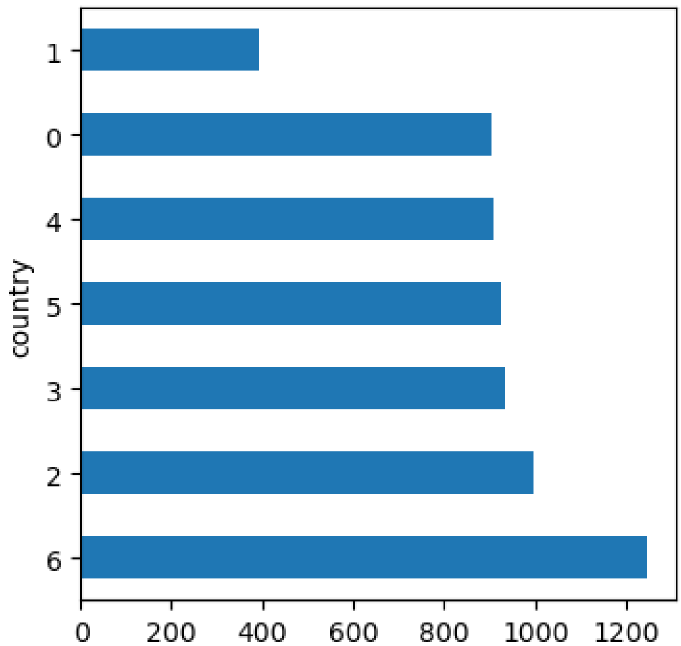
```
Name: count, Length: 81, dtype: int64

--------------------------------------------------------------------------------
```

[71]:

```python
data['brand'].value_counts().plot.barh(figsize=(10,20));
```


brand
12
20
13
71
75
22
80
11
44
70
1
64
38
2
81
0
23
50
76
19
43
59
65
41
72
10
37
6
16
49
29
8
31
15
67
32
45
61
28
18
3
68
34
27
24
30
69
52
54
25
4
79
14
53
74
36
7
40
57
56
66
73
48
21
78
62
58
33
51
46
42
39
35
47
17
60
63
26
77
9
5
55
0
100
200
300
400
500

[72]:
```
data['country'].value_counts().plot.barh(figsize=(4,4));
```



0.2 2- Data Cleaning

To handle missing values in numerical columns using the median, the fillna() method in Pandas is used, applying the median value for each column where missing values (NaN) are present.

- **price**: create a custom function in order to extract price and currency. that car prices are listed in various currencies. To ensure consistency, all prices are standardized to a common currency, for a uniform target variable.
- **car name**, **country** and **brand**: need encoding to convert them to numerical features.
- **engine_capacity**, **cylinder**, **horse_power**, **top_speed**: simple conversion to float and set a limit (standarization)

[227]:
```
LIMIT_HOURSE_POWER = 1_500.0
LIMIT_KMH = 530.0
LIMIT_ENGINE_CAPACITY = 8.4
LIMIT_CYLINDER_NR = 16.0
```

[228]:
```
def is_numeric(value):
    try:
```

```
        float(value)
        return True
    except ValueError:
        return False
```

[229]:
```
def apply_price_adj(price):
    try:
        c = price[:3]
        price_str = price[4:].replace(',', '')

        p = float(price_str)
        pd = p

        conversion_rates = {
        'AED': 0.27,
        'KWD': 3.33,
        'OMR': 2.63,
        'BHD': 2.63,
        'QAR': 0.27,
        'SAR': 0.27,
        'EGP': 0.0333

        }

        if c in conversion_rates:
            pd = p * conversion_rates[c]

        return pd

    except (ValueError, IndexError):
        return -1
```

[252]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.preprocessing import LabelEncoder

# Define the missing values
missing_values = [" ", "NA", "N/A", "N A", "NaN"]

# Read the data and replace the missing values with NaN
data = pd.read_csv("cars.csv", na_values=missing_values)
df_upd = data.copy()

# Print the dataframe before any changes
print("Before filling missing values:")
print(df_upd.head())
```

```
label_encoder = LabelEncoder()    # Initialize the LabelEncoder
categorical_features = ['brand', 'country', 'car name']
for column in categorical_features:
    # Fit the LabelEncoder and transform the column
    df_upd[column] = label_encoder.fit_transform(df_upd[column])
    print(df_upd[column])


# Apply price adjustment function (ensure 'apply_price_adj' is defined)
df_upd['price'] = df_upd['price'].apply(apply_price_adj)

# Function to convert non-numeric values to NaN
def to_numeric(value):
    try:
        return pd.to_numeric(value, errors='coerce')  # Coerce non-numeric␣
↪values to NaN
    except Exception as e:
        return np.nan

# Apply the conversion to numeric for the relevant columns
df_upd['cylinder'] = df_upd['cylinder'].apply(to_numeric)
df_upd['horse_power'] = df_upd['horse_power'].apply(to_numeric)
df_upd['engine_capacity'] = df_upd['engine_capacity'].apply(to_numeric)
df_upd['top_speed'] = df_upd['top_speed'].apply(to_numeric)
df_upd['seats'] = df_upd['seats'].astype(str).str.extract(r'(\d+)')[0].apply(pd.
↪to_numeric, errors='coerce')


print("\n\n")

columns_to_fill = ['car name', 'brand', 'country', 'price', 'cylinder',␣
↪'horse_power', 'top_speed', 'seats','engine_capacity']

for column in columns_to_fill:
    # Calculate the median, ignoring NaN values
    median_value = df_upd[column].median()
    print(f"Median value for {column}: {median_value}")
    df_upd[column] = df_upd[column].fillna(median_value)

# Print the dataframe after filling missing values
print("\nAfter filling missing values:")
print(df_upd.head())
```

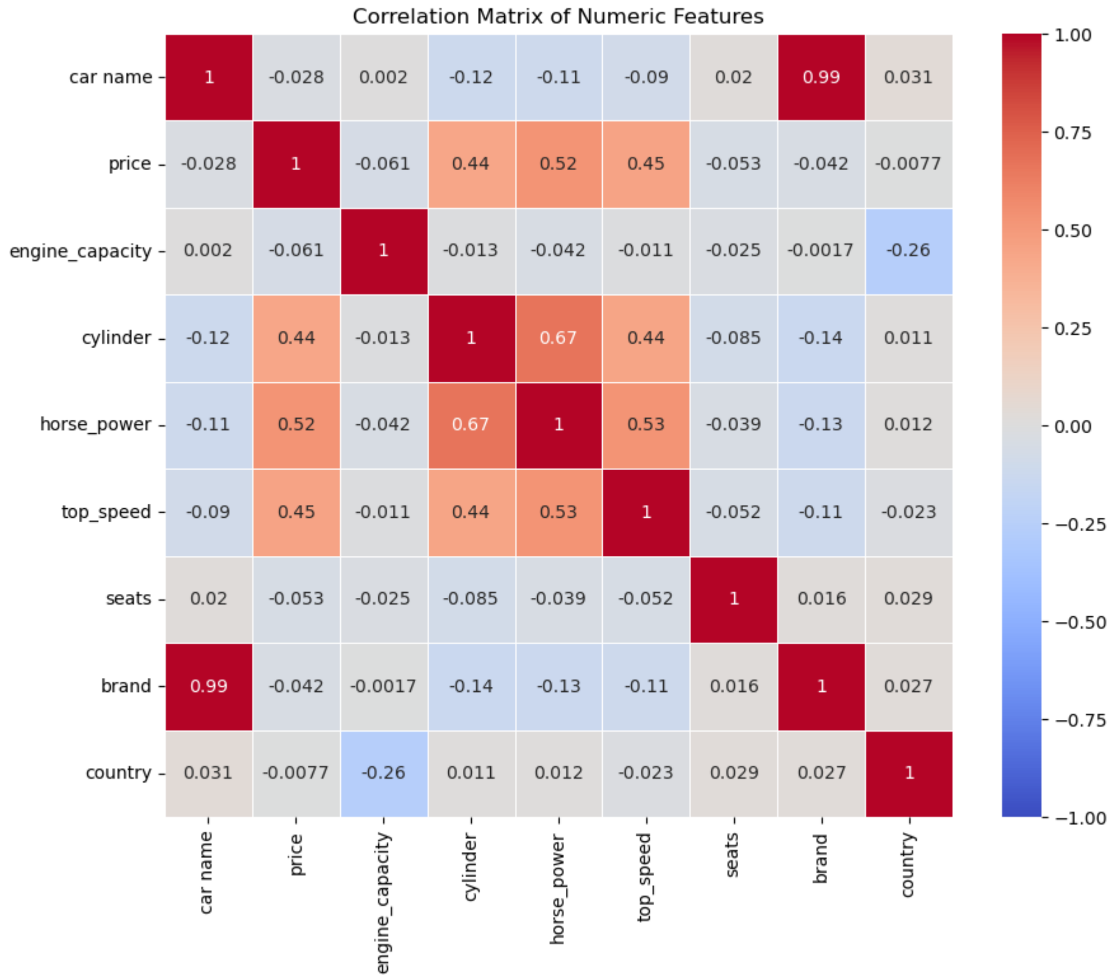
```
Before filling missing values:
                           car name                    price engine_capacity  \
0           Fiat 500e 2021 La Prima                      TBD             0.0
```

```
1     Peugeot Traveller 2021 L3 VIP          SAR 140,575             2.0
2  Suzuki Jimny 2021 1.5L Automatic           SAR 98,785             1.5
3    Ford Bronco 2021 2.3T Big Bend          SAR 198,000             2.3
4     Honda HR-V 2021 1.8 i-VTEC LX  Orangeburst Metallic             1.8

        cylinder horse_power  top_speed     seats    brand country
0  N/A, Electric      Single  Automatic       150     fiat     ksa
1              4         180   8 Seater       8.8  peugeot     ksa
2              4         102        145  4 Seater   suzuki     ksa
3              4         420   4 Seater       7.5     ford     ksa
4              4         140        190  5 Seater    honda     ksa
0       25
1       62
2       74
3       26
4       33
        ..
6303     7
6304    24
6305    67
6306    45
6307     7
Name: brand, Length: 6308, dtype: int32
0       2
1       2
2       2
3       2
4       2
       ..
6303    6
6304    6
6305    6
6306    6
6307    6
Name: country, Length: 6308, dtype: int32
0        564
1       1980
2       2235
3        574
4        811
        ...
6303     333
6304     552
6305    2119
6306    1246
6307     334
Name: car name, Length: 6308, dtype: int32
```

```
Median value for car name: 1299.5
Median value for brand: 46.0
Median value for country: 3.0
Median value for price: 33817.5
Median value for cylinder: 4.0
Median value for horse_power: 255.0
Median value for top_speed: 211.0
Median value for seats: 5.0
Median value for engine_capacity: 2.7

After filling missing values:
   car name     price  engine_capacity  cylinder  horse_power  top_speed  \
0       564     -1.00              0.0       4.0        255.0      211.0
1      1980  37955.25              2.0       4.0        180.0      211.0
2      2235  26671.95              1.5       4.0        102.0      145.0
3       574  53460.00              2.3       4.0        420.0      211.0
4       811     -1.00              1.8       4.0        140.0      190.0

   seats  brand  country
0  150.0     25        2
1    8.0     62        2
2    4.0     74        2
3    7.0     26        2
4    5.0     33        2
```

[87]:

```
# ##correlation

data_numeric = df_upd.apply(pd.to_numeric, errors='coerce')

# Calculate the correlation matrix
correlation_matrix = data_numeric.corr()

# Plot the heatmap to visualize correlations
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', vmin=-1, vmax=1,␣
 ↪linewidths=0.5)
plt.title('Correlation Matrix of Numeric Features')
plt.show()
```



0.2.1 3 - Split the dataset

Split the dataset into training, validation, and test sets. A common split would be 60% for training, 20% for validation, and 20% for testing.

[254]:
```
# First, split the data into 80% training+validation and 20% testing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

X = df_upd.drop(columns='price')  # price is the target column
y = df_upd['price']
X_train_val, X_test, y_train_val, y_test = train_test_split(X , y, test_size=0.
↪2, random_state=42)

# Then, split the 80% training+validation into 60% training and 20% validation
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,␣
↪test_size=0.25, random_state=42)
```

```
# Print the sizes of each split
print("Training set size:", X_train.shape, y_train.shape)
print("Validation set size:", X_val.shape, y_val.shape)
print("Test set size:", X_test.shape, y_test.shape)
```

```
Training set size: (3784, 8) (3784,)
Validation set size: (1262, 8) (1262,)
Test set size: (1262, 8) (1262,)
```

0.2.2 4-Standardization

[257]:
```
from sklearn.preprocessing import StandardScaler
import numpy as np

columns_to_standardize = ['top_speed', 'horse_power', 'car name',␣
 ↪'engine_capacity', 'brand', 'country']
scaler_standard = StandardScaler()

# Calculate range before standardization
print("Range before standardization:")
for col in columns_to_standardize:
    value_range = np.max(X_train[col]) - np.min(X_train[col])
    print(f"Column: {col}")
    print(f"  Range: {value_range:.4f}")

print("\nStandardizing data...")
X_train[columns_to_standardize] = scaler_standard.
 ↪fit_transform(X_train[columns_to_standardize])
X_val[columns_to_standardize] = scaler_standard.
 ↪transform(X_val[columns_to_standardize])
X_test[columns_to_standardize] = scaler_standard.
 ↪transform(X_test[columns_to_standardize])

y_train = np.array(y_train).reshape(-1, 1)
y_val = np.array(y_val).reshape(-1, 1)
y_test = np.array(y_test).reshape(-1, 1)

y_train = scaler_standard.fit_transform(y_train)
y_val = scaler_standard.transform(y_val)
y_test = scaler_standard.transform(y_test)

# Calculate range after standardization
print("\nRange after standardization:")
for col in columns_to_standardize:
    value_range = np.max(X_train[col]) - np.min(X_train[col])
```

```
    print(f"Column: {col}")
    print(f"  Range: {value_range:.4f}")
```

```
Range before standardization:
Column: top_speed
  Range: 846.0000
Column: horse_power
  Range: 5038.0000
Column: car name
  Range: 2544.0000
Column: engine_capacity
  Range: 6752.0000
Column: brand
  Range: 81.0000
Column: country
  Range: 6.0000

Standardizing data...

Range after standardization:
Column: top_speed
  Range: 17.9885
Column: horse_power
  Range: 26.2387
Column: car name
  Range: 3.5079
Column: engine_capacity
  Range: 12.4028
Column: brand
  Range: 3.5966
Column: country
  Range: 2.9666
```

1 2-Building Regression Models

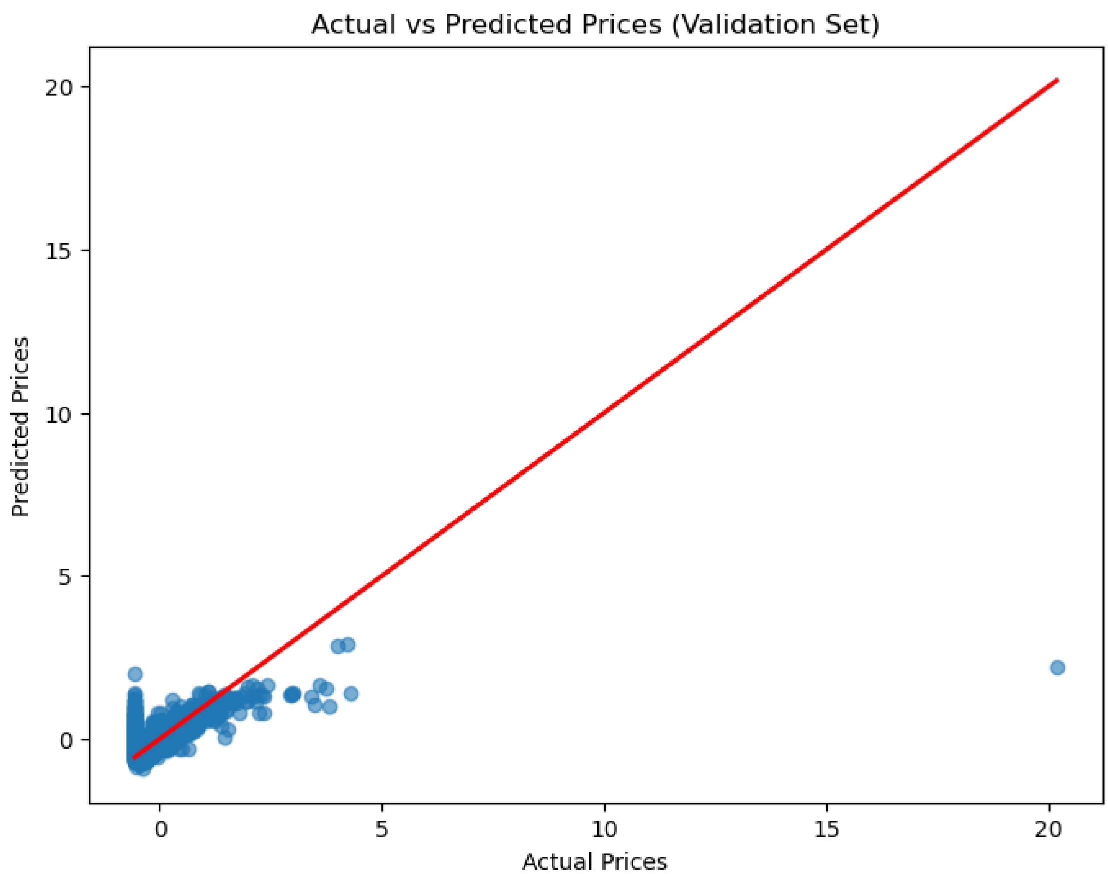
1.0.1 Linear Regression

[243]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

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

coefficients = model.coef_
intercept = model.intercept_
```

```
# Check if intercept is an array and handle it
if isinstance(intercept, np.ndarray):
    intercept = intercept[0]  # Extract the first intercept if it's an array

print("Linear regression equation:")
equation = f"y = {intercept:.2f} "
for i, coef in enumerate(coefficients[0]):  # Use coefficients[0] for single␣
↪target variable
    equation += f"+ ({coef:.2f}) * {X_train.columns[i]} "
print(equation)


y_pred_train = model.predict(X_train)

mse_train = mean_squared_error(y_train, y_pred_train)
r2_train = r2_score(y_train, y_pred_train)
print("\n\n")
print(f"Training Mean Squared Error: {mse_train:.2f}")
print(f"Training R² Score: {r2_train:.2f}")

y_val_pred = model.predict(X_val)
mse = mean_squared_error(y_val, y_val_pred)
r2 = r2_score(y_val, y_val_pred)
print(f"Validation Mean Squared Error: {mse:.2f}")
print(f"Validation R² Score: {r2:.2f}")

y_test_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_test_pred)
r2 = r2_score(y_test, y_test_pred)
print(f"Testing Mean Squared Error: {mse:.2f}")
print(f"Testing R² Score: {r2:.2f}")

plt.figure(figsize=(8, 6))
plt.scatter(y_val, y_val_pred, alpha=0.6)
plt.plot([y_val.min(), y_val.max()], [y_val.min(), y_val.max()], color='red',␣
↪linestyle='-', linewidth=2)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Actual vs Predicted Prices (Validation Set)")
plt.show()
```

```
Linear regression equation:
y = -0.61 + (0.20) * car name + (-0.05) * engine_capacity + (0.12) * cylinder +
(0.21) * horse_power + (0.24) * top_speed + (-0.00) * seats + (-0.17) * brand +
(-0.03) * country
```

```
Training Mean Squared Error: 0.69
Training R² Score: 0.31
Validation Mean Squared Error: 0.49
Validation R² Score: 0.33
Testing Mean Squared Error: 0.56
Testing R² Score: 0.34
```



1.1 Linear regression using closed form solution

weights = (X_transpose . X)^-1 . (X_transpose . y)

[247]:
```
import numpy as np
from sklearn.metrics import mean_squared_error, r2_score

X_train_with_intercept = np.c_[np.ones(X_train.shape[0]), X_train]

X_transpose = X_train_with_intercept.T
X_transpose_X = np.dot(X_transpose, X_train_with_intercept)
```

```python
X_transpose_X_inv = np.linalg.inv(X_transpose_X)
X_transpose_y = np.dot(X_transpose, y_train)

w = np.dot(X_transpose_X_inv, X_transpose_y)

feature_names = ['intercept'] + list(X_train.columns)

equation_terms = []
for i, coef in enumerate(w):
    equation_terms.append(f"({coef[0]:.2f}) * {feature_names[i]}")

equation = " + ".join(equation_terms)
print("using closed form solution")
print(f"Linear regression equation: y = {equation}")

# Add intercept to validation data and make predictions
X_val_with_intercept = np.c_[np.ones(X_val.shape[0]), X_val]
y_pred_val = np.dot(X_val_with_intercept, w)

# Calculate MSE and R² for validation data
mse = mean_squared_error(y_val, y_pred_val)
r2 = r2_score(y_val, y_pred_val)

print(f"Validation Mean Squared Error: {mse:.2f}")
print(f"Validation R² Score: {r2:.2f}")


# Add intercept to testing data and make predictions
X_test_with_intercept = np.c_[np.ones(X_test.shape[0]), X_test]
y_pred_test = np.dot(X_test_with_intercept, w)

# Calculate MSE and R² for validation data
mse = mean_squared_error(y_test, y_pred_test)
r2 = r2_score(y_test, y_pred_test)

print(f"Testing Mean Squared Error: {mse:.2f}")
print(f"Testing R² Score: {r2:.2f}")
```

```
using closed form solution
Linear regression equation: y = (-0.61) * intercept + (0.20) * car name +
(-0.05) * engine_capacity + (0.12) * cylinder + (0.21) * horse_power + (0.24) *
top_speed + (-0.00) * seats + (-0.17) * brand + (-0.03) * country
Validation Mean Squared Error: 0.49
Validation R² Score: 0.33
Testing Mean Squared Error: 0.56
Testing R² Score: 0.34
```

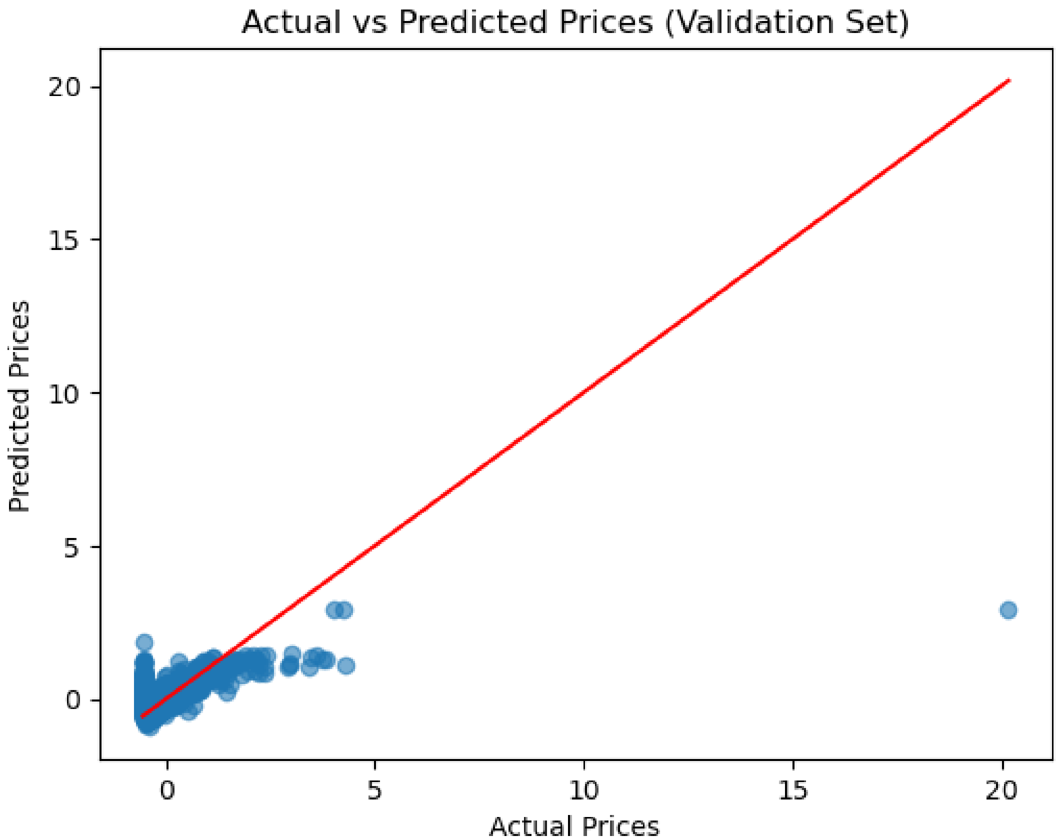
1.2 Linear regression using the gradient descent method.

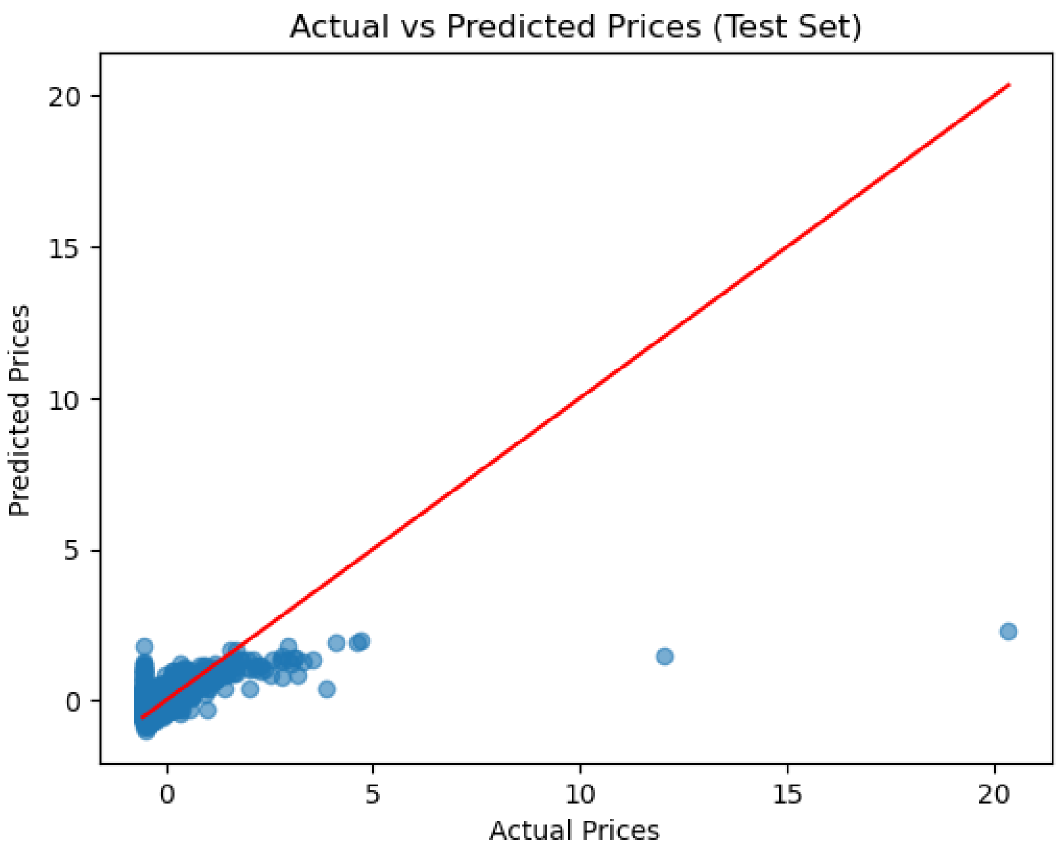
This part is implemented without using any external APIs or libraries for linear regression

[285]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score

# Gradient Descent for Linear Regression
def linear_regression_gd(X, y, learning_rate=0.01, epochs=1000):
     # Add intercept term (bias)
    X = np.c_[np.ones(X.shape[0]), X]  # Add a column of ones for the intercept␣
 ↪term
    m = len(y)  # Number of samples
    n = X.shape[1]  # Number of features + 1 (for intercept term)
    weights = np.zeros(n)  # Initialize weights
    for epoch in range(epochs):
        predictions = np.dot(X, weights)
        errors = predictions - y
        gradient = (2 / m) * np.dot(X.T, errors)
        weights -= learning_rate * gradient
    return weights

# Ensure y is a 1D array
y_train = y_train.flatten()
y_val = y_val.flatten()

# Train model
learning_rate = 0.001
epochs = 4000

weights = linear_regression_gd(X_train, y_train, learning_rate, epochs)

# Extract intercept and coefficients
intercept = weights[0]
coefficients = weights[1:]

feature_names = ['car name', 'engine_capacity', 'cylinder', 'horse_power',␣
 ↪'top_speed', 'seats', 'brand', 'country']
print("solution using gradient descent:")
equation = f"y = {intercept:.2f} "
for i, coef in enumerate(coefficients):  # Use coefficients for single target␣
 ↪variable
    equation += f"+ ({coef:.2f}) * {feature_names[i]} "  # Access each feature␣
 ↪by name
print(equation)
```

```python
# Validate predictions
y_val_pred = np.dot(np.c_[np.ones(X_val.shape[0]), X_val], weights)

# Calculate metrics
mse = mean_squared_error(y_val, y_val_pred)
r2 = r2_score(y_val, y_val_pred)
print(f"Validation MSE: {mse:.2f}")
print(f"Validation R²: {r2:.2f}")


# Plot Actual vs Predicted (validation set)
plt.scatter(y_val, y_val_pred, alpha=0.6)
plt.plot([y_val.min(), y_val.max()], [y_val.min(), y_val.max()], color='red')
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Actual vs Predicted Prices (Validation Set)")
plt.show()

# Plot Actual vs Predicted (test set)
plt.scatter(y_test, y_test_pred, alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='red')
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Actual vs Predicted Prices (Test Set)")
plt.show()
```

```
NaN in y_val_pred: True
NaN in weights: True
X_val shape: (1262, 8)
weights shape: (9,)
solution using gradient descent:
y = -0.28 + (0.03) * car name + (-0.05) * engine_capacity + (0.06) * cylinder +
(0.28) * horse_power + (0.26) * top_speed + (-0.00) * seats + (-0.00) * brand +
(-0.02) * country
Validation MSE: 0.47
Validation R²: 0.36
```

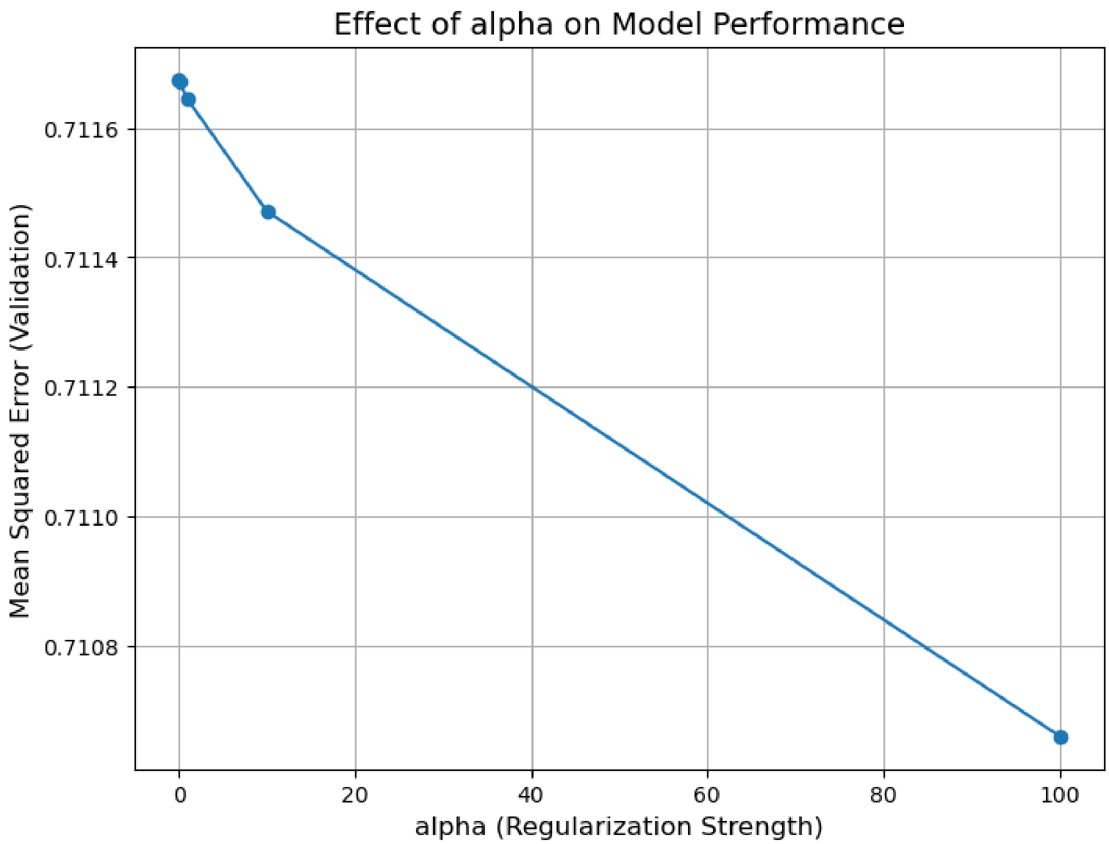

Actual vs Predicted Prices (Validation Set)
Predicted Prices
Actual Prices
0
5
10
15
20
0
5
10
15
20



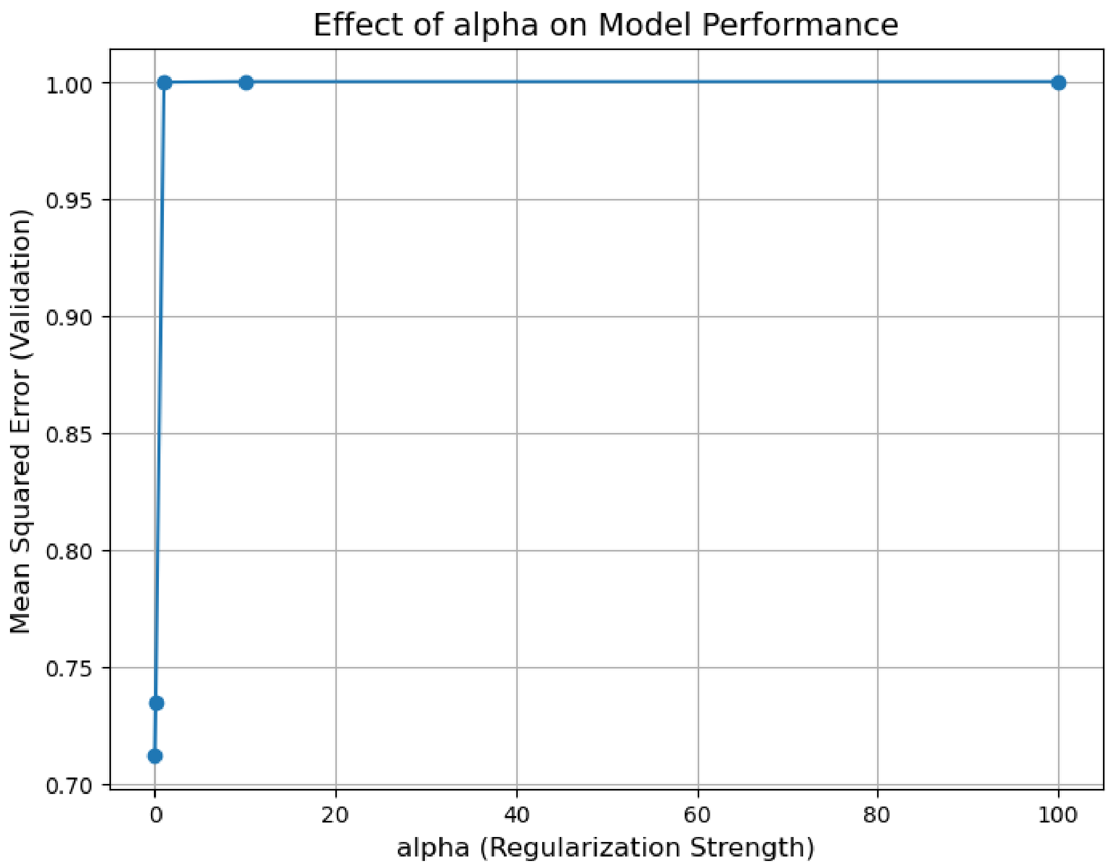
[134]:

```
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import mean_squared_error, make_scorer
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Ridge, Lasso

def perform_grid_search_and_plot(model, param_grid, X_train, y_train):
    grid_search = GridSearchCV(
        estimator=model,
        param_grid=param_grid,
        scoring=make_scorer(mean_squared_error, greater_is_better=False),
        cv=5,
        verbose=0
    )
    grid_search.fit(X_train, y_train)
    results = pd.DataFrame(grid_search.cv_results_)
    param_name = list(param_grid.keys())[0]

    plt.figure(figsize=(8, 6))
    plt.plot(
```

```
        results[f"param_{param_name}"],
        -results["mean_test_score"],
        marker='o'
    )
    plt.xlabel(f"{param_name} (Regularization Strength)", fontsize=12)
    plt.ylabel("Mean Squared Error (Validation)", fontsize=12)
    plt.title(f"Effect of {param_name} on Model Performance", fontsize=14)
    plt.grid(True)
    plt.show()

    return grid_search

# Define the parameter grids
ridge_param_grid = {"alpha": [0.01, 0.1, 1, 10, 100]}
lasso_param_grid = {"alpha": [0.01, 0.1, 1, 10, 100]}

# Perform Grid Search and plot results for Ridge and Lasso
grid_search_ridge = perform_grid_search_and_plot(Ridge(), ridge_param_grid,␣
 ↪X_train, y_train)
grid_search_lasso = perform_grid_search_and_plot(Lasso(), lasso_param_grid,␣
 ↪X_train, y_train)

# Extract and print the best alpha values from the grid search results
best_alpha_ridge = grid_search_ridge.best_params_['alpha']
best_alpha_lasso = grid_search_lasso.best_params_['alpha']

print(f"Best alpha for Ridge: {best_alpha_ridge}")
print(f"Best alpha for Lasso: {best_alpha_lasso}")
```



Effect of alpha on Model Performance
Mean Squared Error (Validation)
0.7116
0.7114
0.7112
0.7110
0.7108
0
20
40
60
80
100
alpha (Regularization Strength)




```
Best alpha for Ridge: 100
Best alpha for Lasso: 0.01
```

1.2.1 2-Ridge regression with grid search for hyperparameter tuning

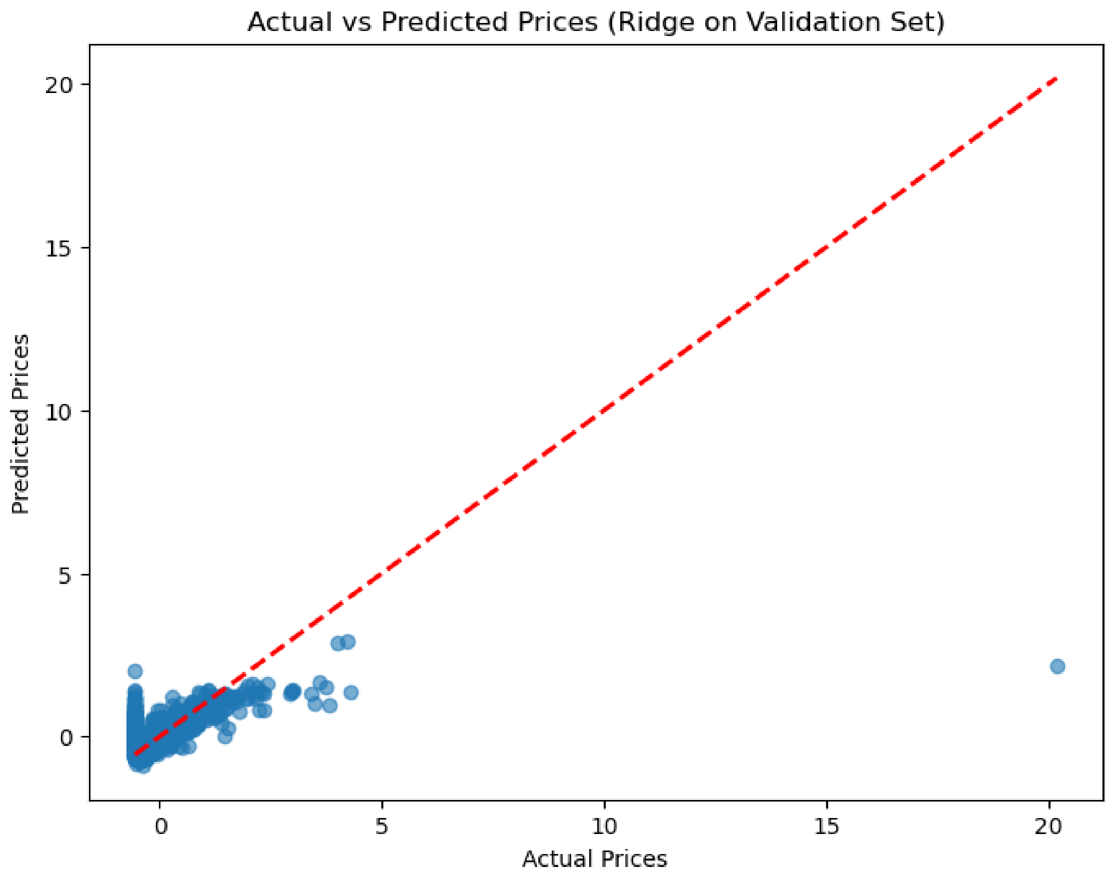
[137]:
```
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Ridge

# Function to perform Grid Search and plot results
def perform_grid_search_and_plot(model, param_grid, X_train, y_train):
    grid_search = GridSearchCV(
        estimator=model,
        param_grid=param_grid,
        scoring='neg_mean_squared_error',
        cv=5,
        verbose=0
    )
```

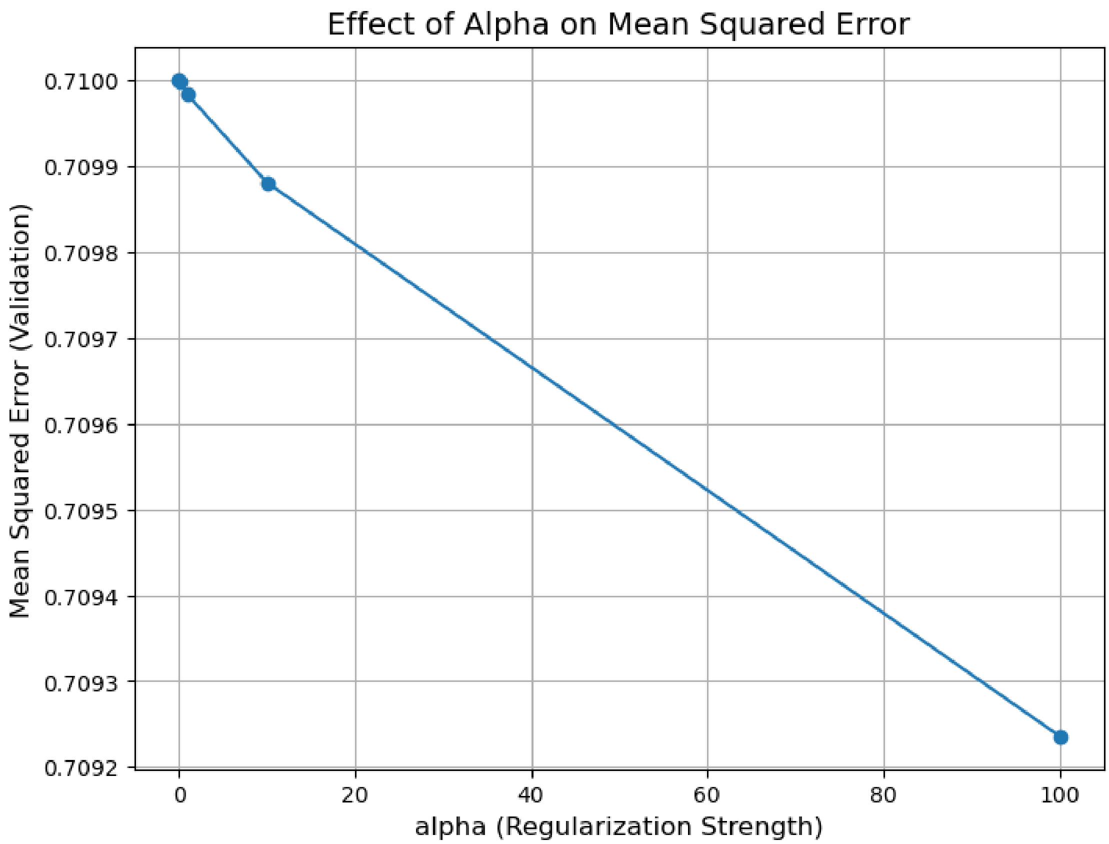
```
    grid_search.fit(X_train, y_train)
    results = pd.DataFrame(grid_search.cv_results_)
    param_name = list(param_grid.keys())[0]

    plt.figure(figsize=(8, 6))
    plt.plot(
        results[f"param_{param_name}"],
        -results["mean_test_score"],
        marker='o'
    )
    plt.xlabel(f"{param_name} (Regularization Strength)", fontsize=12)
    plt.ylabel("Mean Squared Error (Validation)", fontsize=12)
    plt.title(f"Effect of {param_name} on Model Performance", fontsize=14)
    plt.grid(True)
    plt.show()

    return grid_search

# Function to perform Ridge regression grid search and plot
def perform_ridge_grid_search_and_plot(X_train, y_train, ridge_param_grid,␣
 ↪feature_names, X_val, y_val):
    ridge = Ridge()
    grid_search_ridge = GridSearchCV(ridge, ridge_param_grid, cv=10,␣
 ↪scoring='neg_mean_squared_error', return_train_score=True)
    grid_search_ridge.fit(X_train, y_train)

    #best_alpha_ridge = grid_search_ridge.best_params_['alpha']
    best_alpha_ridge=10
    ridge_model = grid_search_ridge.best_estimator_

    intercept = float(ridge_model.intercept_)
    coefficients = ridge_model.coef_.flatten()

    # Print the Ridge regression equation
    print("Ridge Regression Equation:")
    equation = f"y = {intercept:.2f}"
    for coef, feature in zip(coefficients, feature_names):
        equation += f" + ({coef:.2f}) * {feature}"
    print(equation)

    # Predictions on training data
    y_pred_train = ridge_model.predict(X_train)
    mse_train = mean_squared_error(y_train, y_pred_train)
    r2_train = r2_score(y_train, y_pred_train)

    # Predictions on validation data
    y_pred_val = ridge_model.predict(X_val)
```

```
    mse_val = mean_squared_error(y_val, y_pred_val)
    r2_val = r2_score(y_val, y_pred_val)

    y_pred_test = ridge_model.predict(X_test)
    mse_tes = mean_squared_error(y_test, y_pred_test)
    r2_tes = r2_score(y_test, y_pred_test)
    # Print performance metrics
    print(f"Training Mean Squared Error: {mse_train:.2f}")
    print(f"Training R² Score: {r2_train:.2f}")
    print(f"Validation Mean Squared Error: {mse_val:.2f}")
    print(f"Validation R² Score: {r2_val:.2f}")
    print(f"testing Mean Squared Error: {mse_tes:.2f}")
    print(f"testing R² Score: {r2_tes:.2f}")

    print("-" * 50)



    plt.figure(figsize=(8, 6))
    plt.scatter(y_val, y_pred_val, alpha=0.6)
    plt.plot([y_val.min(), y_val.max()], [y_val.min(), y_val.max()],␣
 ↪color='red', linestyle='--', linewidth=2)
    plt.xlabel("Actual Prices")
    plt.ylabel("Predicted Prices")
    plt.title(f"Actual vs Predicted Prices (Ridge on Validation Set)")
    plt.show()

    results = pd.DataFrame(grid_search_ridge.cv_results_)
    param_name = list(ridge_param_grid.keys())[0]
    plt.figure(figsize=(8, 6))
    plt.plot(
        results[f"param_{param_name}"],
        -results["mean_test_score"],
        marker='o'
    )
    plt.xlabel(f"{param_name} (Regularization Strength)", fontsize=12)
    plt.ylabel("Mean Squared Error (Validation)", fontsize=12)
    plt.title("Effect of Alpha on Mean Squared Error", fontsize=14)
    plt.grid(True)
    plt.show()

ridge_param_grid = {"alpha": [0.01, 0.1, 1, 10, 100]}

feature_names = ['car_name', 'engine_capacity', 'cylinder', 'horse_power',␣
 ↪'top_speed', 'seats', 'brand', 'country']
```

```
perform_ridge_grid_search_and_plot(X_train, y_train, ridge_param_grid,
 ↪feature_names, X_val, y_val)
```

```
Ridge Regression Equation:
y = -0.63 + (0.05) * car_name + (-0.05) * engine_capacity + (0.12) * cylinder +
(0.21) * horse_power + (0.23) * top_speed + (-0.00) * seats + (-0.02) * brand +
(-0.02) * country
Training Mean Squared Error: 0.69
Training R² Score: 0.31
Validation Mean Squared Error: 0.49
Validation R² Score: 0.33
testing Mean Squared Error: 0.57
testing R² Score: 0.34
--------------------------------------------------
```



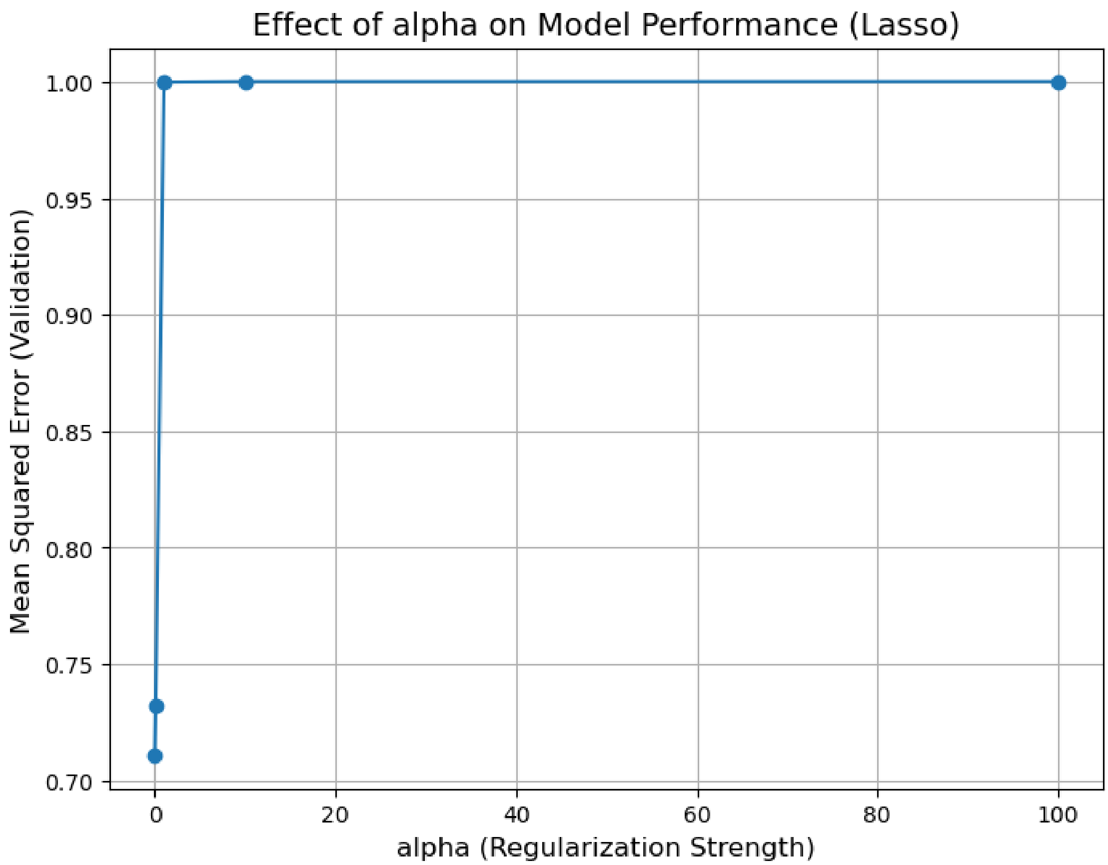


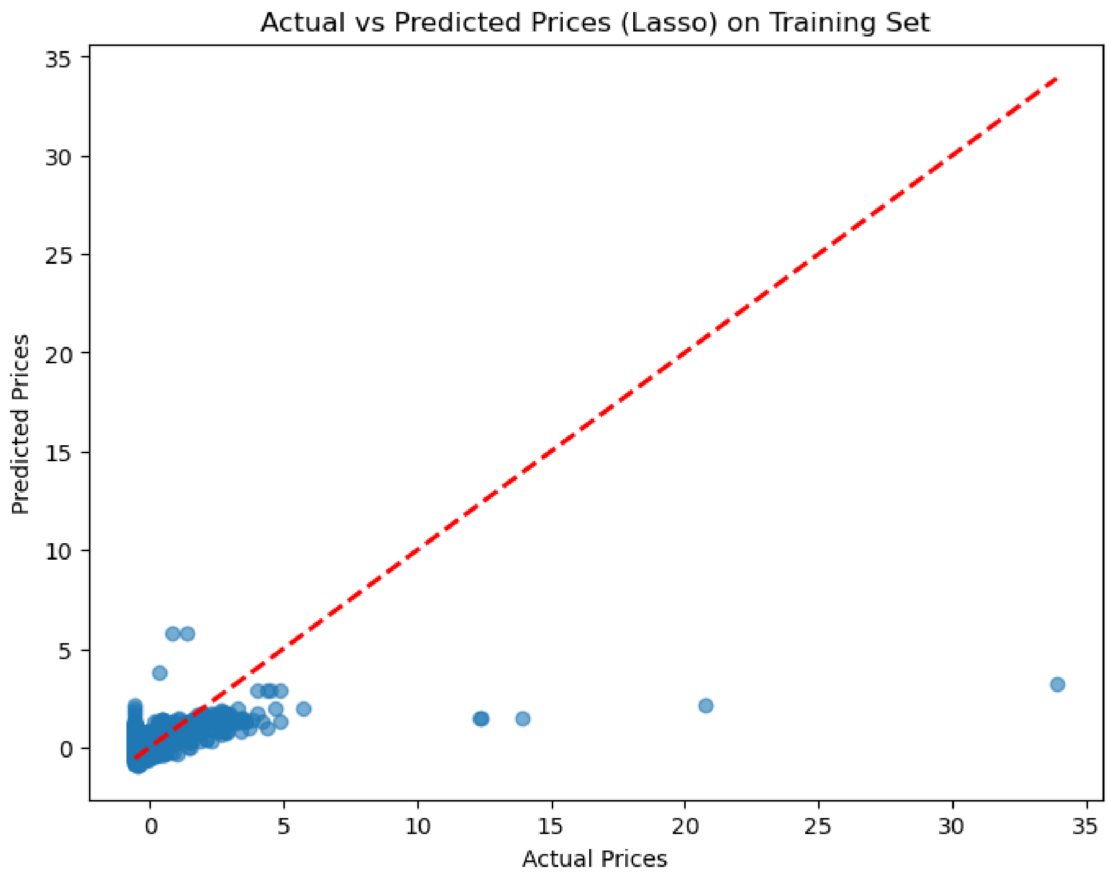
[139]:
```
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Lasso

def perform_lasso_grid_search_and_plot(X_train, y_train, lasso_param_grid,␣
↪feature_names):
    # Define the Lasso model
    lasso = Lasso()

    # Perform Grid Search with cross-validation
    grid_search_lasso = GridSearchCV(lasso, lasso_param_grid, cv=10,␣
↪scoring='neg_mean_squared_error', return_train_score=True)
    grid_search_lasso.fit(X_train, y_train)

    # Best model from Grid Search
    lasso_model = grid_search_lasso.best_estimator_
    best_alpha_lasso = grid_search_lasso.best_params_['alpha']
```

```python
    # Get the coefficients and intercept
    intercept = float(lasso_model.intercept_)
    coefficients = lasso_model.coef_.flatten()

    # Print the Lasso regression equation
    print("Lasso Regression Equation:")
    equation = f"y = {intercept:.2f}"  # intercept is scalar
    for coef, feature in zip(coefficients, feature_names):
        equation += f" + ({coef:.2f}) * {feature}"
    print(equation)

   # Predictions on training data
    y_pred_train = lasso_model.predict(X_train)
    mse_train = mean_squared_error(y_train, y_pred_train)
    r2_train = r2_score(y_train, y_pred_train)

    # Predictions on validation data
    y_pred_val = lasso_model.predict(X_val)
    mse_val = mean_squared_error(y_val, y_pred_val)
    r2_val = r2_score(y_val, y_pred_val)

    y_pred_test = lasso_model.predict(X_test)
    mse_tes = mean_squared_error(y_test, y_pred_test)
    r2_tes = r2_score(y_test, y_pred_test)
    # Print performance metrics
    print(f"Training Mean Squared Error: {mse_train:.2f}")
    print(f"Training R² Score: {r2_train:.2f}")
    print(f"Validation Mean Squared Error: {mse_val:.2f}")
    print(f"Validation R² Score: {r2_val:.2f}")
    print(f"testing Mean Squared Error: {mse_tes:.2f}")
    print(f"testing R² Score: {r2_tes:.2f}")

    print("-" * 50)


    # Plot the effect of alpha on MSE
    results = pd.DataFrame(grid_search_lasso.cv_results_)
    param_name = list(lasso_param_grid.keys())[0]
    plt.figure(figsize=(8, 6))
    plt.plot(
        results[f"param_{param_name}"],
        -results["mean_test_score"],
        marker='o'
    )
    plt.xlabel(f"{param_name} (Regularization Strength)", fontsize=12)
    plt.ylabel("Mean Squared Error (Validation)", fontsize=12)
```

```
    plt.title(f"Effect of {param_name} on Model Performance (Lasso)",␣
↪fontsize=14)
    plt.grid(True)
    plt.show()

    # Plot Actual vs Predicted
    y_val_pred_lasso = lasso_model.predict(X_train)
    plt.figure(figsize=(8, 6))
    plt.scatter(y_train, y_val_pred_lasso, alpha=0.6)
    plt.plot([y_train.min(), y_train.max()], [y_train.min(), y_train.max()],␣
↪color='red', linestyle='--', linewidth=2)
    plt.xlabel("Actual Prices")
    plt.ylabel("Predicted Prices")
    plt.title(f"Actual vs Predicted Prices (Lasso) on Training Set")
    plt.show()

    return grid_search_lasso, best_alpha_lasso

# Define the parameter grid for Lasso
lasso_param_grid = {"alpha": [0.01, 0.1, 1, 10, 100]}

# Define the feature names (adjust this list based on your actual dataset)
feature_names = ['car_name', 'engine_capacity', 'cylinder', 'horse_power',␣
↪'top_speed', 'seats', 'brand', 'country']

# Perform Grid Search and plot results for Lasso
perform_lasso_grid_search_and_plot(X_train, y_train, lasso_param_grid,␣
↪feature_names)
```

```
Lasso Regression Equation:
y = -0.63 + (0.02) * car_name + (-0.04) * engine_capacity + (0.12) * cylinder +
(0.20) * horse_power + (0.23) * top_speed + (-0.00) * seats + (0.00) * brand +
(-0.01) * country
Training Mean Squared Error: 0.69
Training R² Score: 0.31
Validation Mean Squared Error: 0.49
Validation R² Score: 0.33
testing Mean Squared Error: 0.57
testing R² Score: 0.33
--------------------------------------------------
```


Effect of alpha on Model Performance (Lasso)
Mean Squared Error (Validation)
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0
20
40
60
80
100
alpha (Regularization Strength)



[139]:
```
(GridSearchCV(cv=10, estimator=Lasso(),
              param_grid={'alpha': [0.01, 0.1, 1, 10, 100]},
              return_train_score=True, scoring='neg_mean_squared_error'),
 0.01)
```

1.3 Non-linear Models

1.3.1 Polynomial Models

[150]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.kernel_ridge import KernelRidge
from sklearn.metrics import mean_squared_error
from mlxtend.feature_selection import SequentialFeatureSelector as SFS


# Polynomial Regression
degrees = [2, 5, 7, 10]
```

```
# ##with forward feature selection
for degree in degrees:
    print(f"Fitting degree {degree}")

    poly = PolynomialFeatures(degree=degree)
    model = LinearRegression()

    # Run forward selection
    selected_features, mse_history = forward_selection(X_train, X_test, y_train,␣
 ↪y_test, model, max_features=5)

    # Print the selected features and corresponding MSE history
    print(f"Selected Features: {selected_features}")
    for feature, mse in mse_history:
        print(f"Feature: {feature}, MSE: {mse:.4f}")

    # Train final model with selected features
    model.fit(X_train[selected_features], y_train)
    y_pred = model.predict(X_train[selected_features])
    mse = mean_squared_error(y_train, y_pred)
    print(f"Degree {degree}: MSE on training set = {mse:.4f}")
   # model.fit(X_val[selected_features], y_val)
    y_pred = model.predict(X_val[selected_features])
    mse_val= mean_squared_error(y_val, y_pred)
    print(f"Degree {degree}: MSE on validation set = {mse:.4f}")

    y_pred = model.predict(X_test[selected_features])
    final_mse = mean_squared_error(y_test, y_pred)
    print(f"Final Model MSE on Test Set: {final_mse:.4f}")
    print(f"Finished fitting degree {degree}")
    print("\n\n\n")



#without forward feature selection

#Evaluate models with Mean Squared Error
# print("Polynomial Regression Errors:")
# for degree in degrees:
#     poly = PolynomialFeatures(degree=degree)
#     X_train_poly = poly.fit_transform(X_train)
#     X_val_poly = poly.transform(X_val)
#     X_test_poly = poly.transform(X_test)
#     model = LinearRegression()
#     model.fit(X_train_poly, y_train)
#     y_pred = model.predict(X_train_poly)
```

```
#     mse = mean_squared_error(y_train, y_pred)
#     print(f"Degree {degree}: MSE on training set = {mse:.4f}")
#     model.fit(X_val_poly, y_val)
#     y_pred = model.predict(X_val_poly)
#     mse_val= mean_squared_error(y_val, y_pred)
#     print(f"Degree {degree}: MSE on validation set = {mse:.4f}")
#     model.fit(X_test_poly, y_test)
#     y_pred = model.predict(X_test_poly)
#     mse = mean_squared_error(y_test, y_pred)
#     print(f"Degree {degree}: MSE on test set = {mse:.4f}")
```

```
Fitting degree 2
Selected Features: ['horse_power', 'car name', 'engine_capacity', 'country',
'seats']
Feature: horse_power, MSE: 0.5311
Feature: car name, MSE: 0.5299
Feature: engine_capacity, MSE: 0.5290
Feature: country, MSE: 0.5281
Feature: seats, MSE: 0.5275
Degree 2: MSE on training set = 0.7751
Degree 2: MSE on validation set = 0.7751
Final Model MSE on Test Set: 0.5275
Finished fitting degree 2




Fitting degree 5
Selected Features: ['horse_power', 'car name', 'engine_capacity', 'country',
'seats']
Feature: horse_power, MSE: 0.5311
Feature: car name, MSE: 0.5299
Feature: engine_capacity, MSE: 0.5290
Feature: country, MSE: 0.5281
Feature: seats, MSE: 0.5275
Degree 5: MSE on training set = 0.7751
Degree 5: MSE on validation set = 0.7751
Final Model MSE on Test Set: 0.5275
Finished fitting degree 5




Fitting degree 7
Selected Features: ['horse_power', 'car name', 'engine_capacity', 'country',
'seats']
Feature: horse_power, MSE: 0.5311
Feature: car name, MSE: 0.5299
```

```
Feature: engine_capacity, MSE: 0.5290
Feature: country, MSE: 0.5281
Feature: seats, MSE: 0.5275
Degree 7: MSE on training set = 0.7751
Degree 7: MSE on validation set = 0.7751
Final Model MSE on Test Set: 0.5275
Finished fitting degree 7
```

```
Fitting degree 10
Selected Features: ['horse_power', 'car name', 'engine_capacity', 'country',
'seats']
Feature: horse_power, MSE: 0.5311
Feature: car name, MSE: 0.5299
Feature: engine_capacity, MSE: 0.5290
Feature: country, MSE: 0.5281
Feature: seats, MSE: 0.5275
Degree 10: MSE on training set = 0.7751
Degree 10: MSE on validation set = 0.7751
Final Model MSE on Test Set: 0.5275
Finished fitting degree 10
```

1.3.2 Feature Selection with Forward Selection

[148]:
```
# Forward Selection Function
def forward_selection(X_train, X_test, y_train, y_test, model,␣
 ↪max_features=None):
    remaining_features = list(X_train.columns)  # All features available for␣
 ↪selection
    selected_features = []  # Initially, no features selected
    best_mse = float('inf')  # Start with a very large MSE
    mse_history = []  # To store MSE for each feature added

    # Iteratively add features
    while remaining_features and (max_features is None or len(selected_features)␣
 ↪< max_features):
        feature_mse = {}  # Dictionary to store MSE for each feature added
        for feature in remaining_features:
            # Temporarily add the feature to the selected set
            current_features = selected_features + [feature]
```

```
            # Train and validate the model using selected features
            model.fit(X_train[current_features], y_train)
            y_pred = model.predict(X_test[current_features])
            mse = mean_squared_error(y_test, y_pred)  # Calculate MSE on the␣
↪test set

            feature_mse[feature] = mse  # Store the MSE for this feature

        # Select the feature with the minimum MSE
        best_feature = min(feature_mse, key=feature_mse.get)
        best_mse_for_feature = feature_mse[best_feature]

        # If the best feature improves the model, add it to selected features
        if best_mse_for_feature < best_mse:
            selected_features.append(best_feature)
            remaining_features.remove(best_feature)
            best_mse = best_mse_for_feature  # Update the best MSE
            mse_history.append((best_feature, best_mse))  # Log the feature and␣
↪MSE

        else:
            break  # Stop if adding more features doesn't improve the model

    return selected_features, mse_history
```

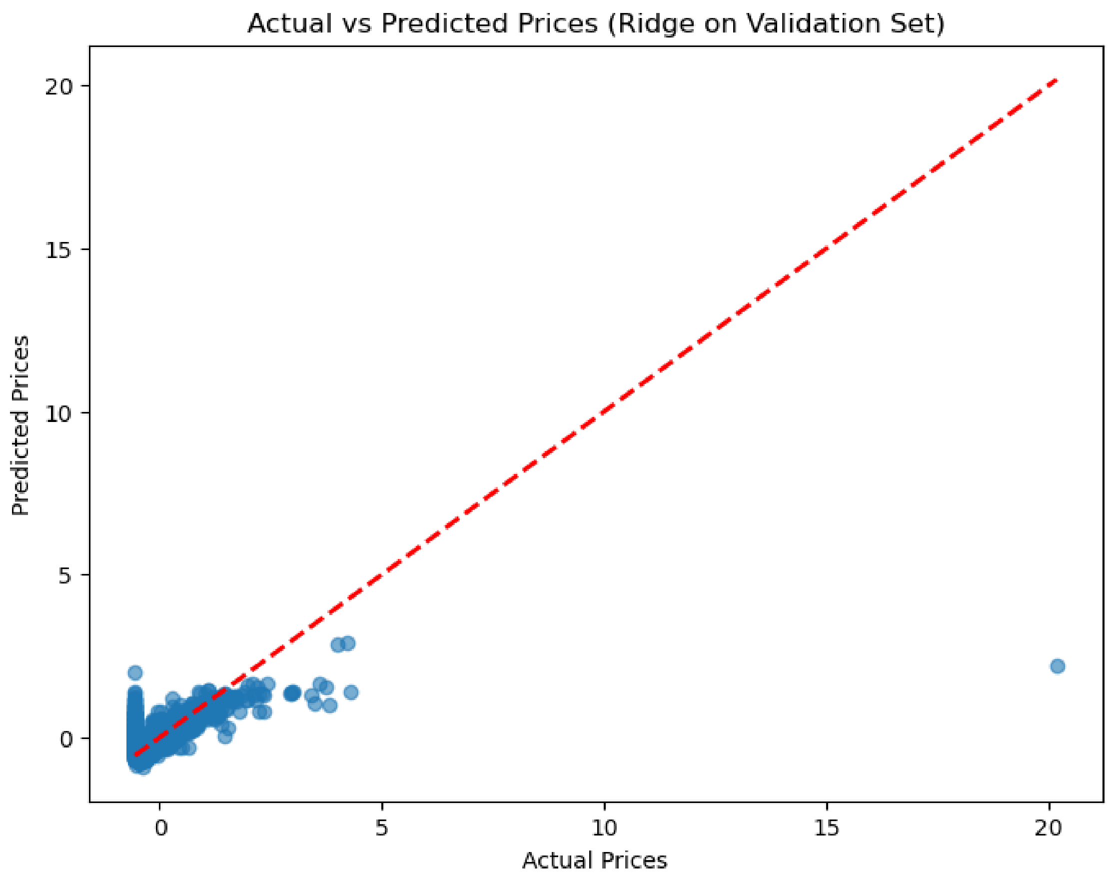
1.3.3 Radial Basis Function (RBF)

[152]:
```
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
from sklearn.kernel_ridge import KernelRidge
from sklearn.linear_model import Ridge

# RBF Kernel Regression
alpha = 1.0      # Regularization parameter for RBF Kernel
gamma = 0.1      # Kernel coefficient (low value smooths out predictions)
rbf_model = KernelRidge(kernel="rbf", alpha=alpha, gamma=gamma)
rbf_model.fit(X_train, y_train)

# Predictions on training data for RBF Kernel
y_rbf_train_pred = rbf_model.predict(X_train)
mse_rbf_train = mean_squared_error(y_train, y_rbf_train_pred)

# Predictions on validation data for RBF Kernel
y_rbf_val_pred = rbf_model.predict(X_val)
mse_rbf_val = mean_squared_error(y_val, y_rbf_val_pred)

# Predictions on testing data for RBF Kernel
```

```python
y_rbf_test_pred = rbf_model.predict(X_test)
mse_rbf_test = mean_squared_error(y_test, y_rbf_test_pred)

# Ridge Regression
ridge_model = Ridge(alpha=1.0)  # Use the same alpha for Ridge regression

# Fit the Ridge model
ridge_model.fit(X_train, y_train)

# Predictions on training data for Ridge Regression
y_pred_train = ridge_model.predict(X_train)
mse_train = mean_squared_error(y_train, y_pred_train)
r2_train = r2_score(y_train, y_pred_train)

# Predictions on validation data for Ridge Regression
y_pred_val = ridge_model.predict(X_val)
mse_val = mean_squared_error(y_val, y_pred_val)
r2_val = r2_score(y_val, y_pred_val)

# Predictions on testing data for Ridge Regression
y_pred_test = ridge_model.predict(X_test)
mse_test = mean_squared_error(y_test, y_pred_test)
r2_test = r2_score(y_test, y_pred_test)

# Print performance metrics for Ridge Regression
print(f"Training Mean Squared Error: {mse_train:.2f}")
print(f"Training R² Score: {r2_train:.2f}")
print(f"Validation Mean Squared Error: {mse_val:.2f}")
print(f"Validation R² Score: {r2_val:.2f}")
print(f"Testing Mean Squared Error: {mse_test:.2f}")
print(f"Testing R² Score: {r2_test:.2f}")
print("-" * 50)

# Visualization of Actual vs Predicted for Ridge Regression (Validation Set)
plt.figure(figsize=(8, 6))
plt.scatter(y_val, y_pred_val, alpha=0.6)
plt.plot([y_val.min(), y_val.max()], [y_val.min(), y_val.max()], color='red',␣
 ↪linestyle='--', linewidth=2)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Actual vs Predicted Prices (Ridge on Validation Set)")
plt.show()
```

```
Training Mean Squared Error: 0.69
Training R² Score: 0.31
Validation Mean Squared Error: 0.49
Validation R² Score: 0.33
Testing Mean Squared Error: 0.56
```

```
Testing R² Score: 0.34
--------------------------------------------------
```



[154]:

```python
# RBF Kernel Regression
alpha = 1.0       #regularization
gamma = 0.1       #kernel coefficient - low value smooths out predictions (reduce␣
↪complexity)
rbf_model = KernelRidge(kernel="rbf", alpha=alpha, gamma=gamma)
rbf_model.fit(X_train, y_train)
y_rbf_pred = rbf_model.predict(X_test)

y_rbf_train_pred = rbf_model.predict(X_train)
mse_rbf = mean_squared_error(y_train, y_rbf_train_pred)
print(f"RBF Kernel: MSE on training set = {mse_rbf:.4f}")

y_rbf_val_pred = rbf_model.predict(X_val)
mse_rbf = mean_squared_error(y_val, y_rbf_val_pred)
print(f"RBF Kernel: MSE on validation set = {mse_rbf:.4f}")
```

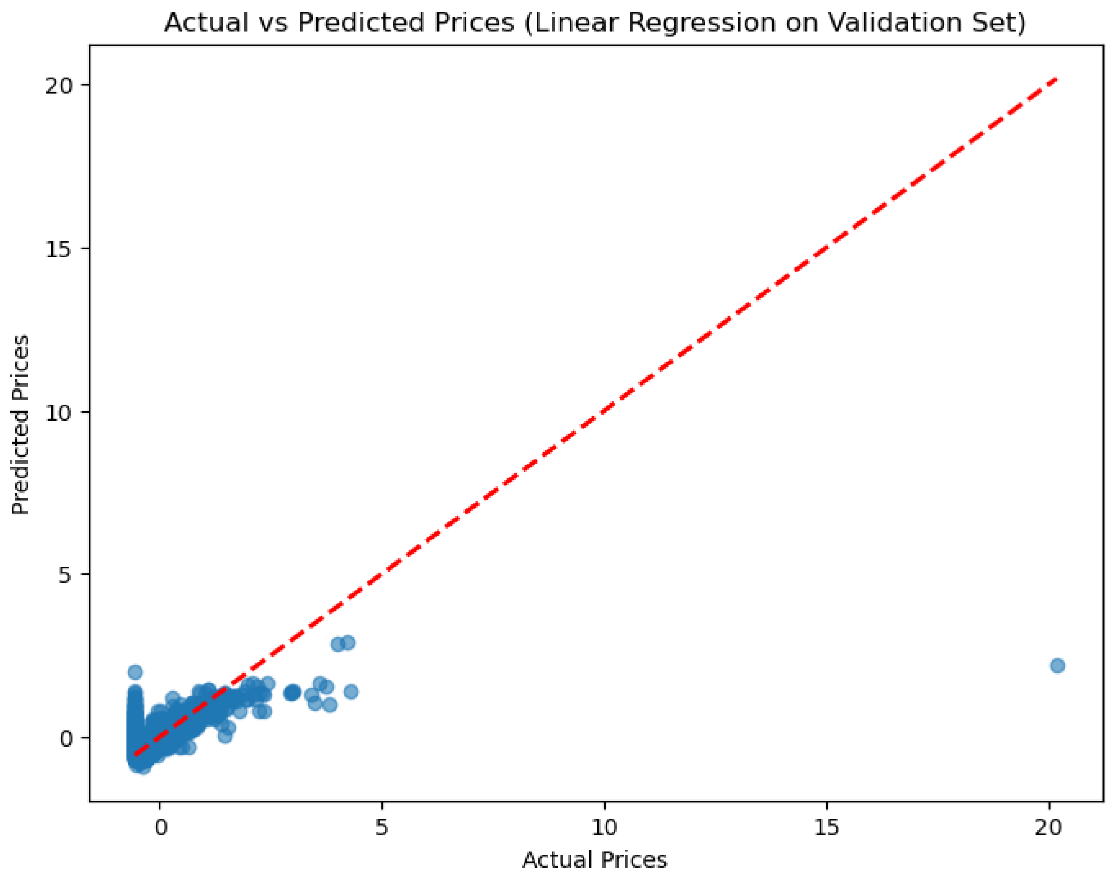
```
y_rbf_test_pred = rbf_model.predict(X_test)
mse_rbf = mean_squared_error(y_test, y_rbf_test_pred)
print(f"RBF Kernel: MSE on testing set = {mse_rbf:.4f}")
```

```
RBF Kernel: MSE on training set = 0.3213
RBF Kernel: MSE on validation set = 0.2956
RBF Kernel: MSE on testing set = 0.3817
```

[156]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import matplotlib.pyplot as plt

models = {
    'Linear Regression': LinearRegression(),
    # 'Ridge Regression': Ridge(alpha=1),
    # 'Lasso Regression': Lasso(alpha=0.1),
    # 'Polynomial Regression (Degree 2)': Pipeline([
    #     ('poly', PolynomialFeatures(degree=2)),
    #     ('model', LinearRegression())
    # ])
}

# Dictionary to store metrics for each model
model_metrics = {}

# Train and evaluate each model
for model_name, model in models.items():
    # Fit the model
    model.fit(X_train, y_train)

    # Predict on the validation set
    y_val_pred = model.predict(X_val)

    # Calculate metrics
    mse = mean_squared_error(y_val, y_val_pred)
    mae = mean_absolute_error(y_val, y_val_pred)
    r2 = r2_score(y_val, y_val_pred)

    # Store metrics in the dictionary
    model_metrics[model_name] = {'MSE': mse, 'MAE': mae, 'R2': r2}

    # Print model performance
    print(f"Model: {model_name}")
```

```
    print(f"Validation Mean Squared Error (MSE): {mse:.2f}")
    print(f"Validation Mean Absolute Error (MAE): {mae:.2f}")
    print(f"Validation R² Score: {r2:.2f}")
    print("-" * 50)

# # Compare models based on MSE, MAE, and R²
best_model = min(model_metrics, key=lambda x: model_metrics[x]['MSE'])
# print(f"The best model based on the lowest MSE is: {best_model}")

best_model_fit = models[best_model]
y_val_pred_best_model = best_model_fit.predict(X_val)

plt.figure(figsize=(8, 6))
plt.scatter(y_val, y_val_pred_best_model, alpha=0.6)
plt.plot([y_val.min(), y_val.max()], [y_val.min(), y_val.max()], color='red',␣
 ↪linestyle='--', linewidth=2)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title(f"Actual vs Predicted Prices ({best_model} on Validation Set)")
plt.show()
```

```
Model: Linear Regression
Validation Mean Squared Error (MSE): 0.49
Validation Mean Absolute Error (MAE): 0.32
Validation R² Score: 0.33
--------------------------------------------------
```



Actual vs Predicted Prices (Linear Regression on Validation Set)
20
15
10
5
0
Predicted Prices
0
5
10
15
20
Actual Prices

[ ]:

[ ]: