Faculty of Engineering and Technology Department of Electrical and Computer Engineering

MACHINE LEARNING AND DATA SCIENCE

(ENEE5341)

Report #2 Regression Analysis and Model Selection

Omar Hussain 1212739

Malak Moqbel 1210608

Instructor's name: Dr.Yazan Abu Farha

Section: 1

Submission date: 28.11.2024

Description of the dataset

The Cars Dataset from YallaMotors, contains detailed information about cars for sale, such as their make, model, year, price, mileage, and other attributes. The main goal is to predict car prices based on these characteristics. There may be some issues in the data, like missing values or outliers, which will be cleaned up during preprocessing.

Preprocessing steps

```
df = df[df['engine_capacity'] != 'Cylinders']
```

We removed the rows containing the value “Cylinders” in the engine_capacity column because we noticed that all the cells in these rows were either missing or filled with dummy data, which could affect the accuracy of our analysis and modeling.(This part was done by Malak.)

```
df['brand'] = df['brand'].str.replace(' ', '', regex=True)
```

The code is used to remove all spaces from the values in the “brand” column of the dataset. This can be helpful if there are unnecessary spaces in the brand names that might interfere with data processing, comparison, or analysis. After executing this code, the “brand”column will contain the brand names without any spaces between characters. .(This part was done by Malak.)

```
df['price'] = df['price'].str.replace(r'[^\d.]', '', regex=True)
df['price'] = df['price'].where(df['price'].notnull(), None)
df['price'] = pd.to_numeric(df['price'], errors='coerce')

print(df['price'].head())
```

```
0         NaN
1    140575.0
2     98785.0
3    198000.0
4         NaN
Name: price, dtype: float64
```

This code cleans up the “price” column to make it usable for analysis. It removes any extra characters, like symbols or text, leaving only numbers and decimals. It also ensures that missing or invalid entries are handled properly by converting them to a standard format “NaN”. Finally, it changes the column to a numeric type, so it’s ready for calculations or modeling. The cleaned data is then checked to confirm everything looks correct. .(This part was done by Omar.)

```python
conversion_rates = {
    'ksa': 0.27,
    'egypt': 0.032,
    'bahrain': 2.65,
    'qatar': 0.27,
    'oman': 2.60,
    'kuwait': 3.26,
    'uae': 0.27
}
numerical_columns = df.select_dtypes(include='number').columns
for country, rate in conversion_rates.items():
    df.loc[df['country'] == country, numerical_columns] *= rate
print(df.head())
```

```
                        car name     price engine_capacity       cylinder  \
0          Fiat 500e 2021 La Prima       NaN             0.0  N/A, Electric
1     Peugeot Traveller 2021 L3 VIP  37955.25             2.0              4
2  Suzuki Jimny 2021 1.5L Automatic  26671.95             1.5              4
3    Ford Bronco 2021 2.3T Big Bend  53460.00             2.3              4
4     Honda HR-V 2021 1.8 i-VTEC LX       NaN             1.8              4

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

This code adjusts the numbers in the dataset, like prices, so they're all in the same currency (USD). It uses a list of conversion rates for each country and applies the right rate to the rows that match each country. For example, if a car's price is in Saudi Riyals, it's converted to USD by multiplying it by 0.27. This makes all the numbers consistent and easy to compare. Finally, the updated data is shown to check the results. .(This part was done by Omar.)

```python
price_median = df.groupby(['brand', 'country'])['price'].transform('median')
df['price'].fillna(price_median.round(2), inplace=True)
print(df.head())
df['price'].fillna(df['price'].median().round(2), inplace=True)  # Replace with global median  as a fallback
df['price'].round(2)

```

```
                        car name     price engine_capacity       cylinder  \
0          Fiat 500e 2021 La Prima  19219.95             0.0  N/A, Electric
1     Peugeot Traveller 2021 L3 VIP  37955.25             2.0              4
2  Suzuki Jimny 2021 1.5L Automatic  26671.95             1.5              4
3    Ford Bronco 2021 2.3T Big Bend  53460.00             2.3              4
4     Honda HR-V 2021 1.8 i-VTEC LX  38005.20             1.8              4

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

This code fills in missing prices by first looking at the median price for cars of the same brand and country, so the replacement value makes sense for similar cars. If a price is still missing, it uses the overall median price for all cars as a backup. Afterward, all prices are rounded to two decimal places to keep things neat and consistent. This ensures the missing prices are filled in a logical and accurate way. .(This part was done by Malak.)

```python
print(df['price'].isnull().sum())
```

```
0
```

confirm whether the missing values in the price column have been fully filled or if some still remain.

```python
df['cylinder'] = df.apply(lambda row: df[df['brand'] == row['brand']]['cylinder'].median() if row['cylinder'] == 'N/A' else row['cylinder'], axis=1)
df['cylinder'] = pd.to_numeric(df['cylinder'], errors='coerce')
```

This code cleans the “cylinder” column by replacing missing or invalid values “'N/A” with the median number of cylinders for cars of the same brand. It checks each row, and if the “cylinder” value is “N/A”, it looks up the median for that brand and uses it as a replacement. Afterward, the column is converted to numeric format, ensuring all values are numbers. Any entries that can’t be converted are set to “NaN”. This process makes the “cylinder” data clean, consistent, and ready for analysis. .(This part was done by Malak.)

```python
df['horse_power'] = df['horse_power'].str.replace(r'[^0-9.]', '', regex=True)
df['horse_power'] = pd.to_numeric(df['horse_power'], errors='coerce')
df['horse_power'] = df.groupby('brand')['horse_power'].transform(lambda x: x.fillna(x.median()))
print(df['horse_power'].head())
```

```
0    100.0
1    180.0
2    102.0
3    420.0
4    140.0
Name: horse_power, dtype: float64
```

This code cleans and processes the “horse_power” column. First, it removes any non-numeric characters from the values using a regular expression. Then, it converts the cleaned values to numeric format, setting invalid entries to “NaN”. Next, it handles missing values by filling them with the median horsepower for each car brand. This ensures the column is consistent, numeric, and has no missing values, making it ready for further analysis. The printed output confirms the final cleaned and transformed data. .(This part was done by Omar.)

```python
df['engine_capacity'] = pd.to_numeric(df['engine_capacity'], errors='coerce')  # Convert to float, setting invalid values to NaN
df['engine_capacity'] = df['engine_capacity'].apply(lambda x: x / 1000 if x > 10 else x)
print(df['engine_capacity'])
```

This code processes the “engine_capacity” column to ensure it is in a clean and usable format. It first converts all values in the column to numeric (float), replacing invalid or non-convertible values with “NaN”. Then, it adjusts any unusually high values (greater than 10, likely in cubic centimeters) by dividing them by 1000 to convert them to liters, which is a standard unit for engine capacity. The resulting column contains properly scaled engine capacities in liters, ready for analysis or modeling. The printed output confirms the transformed values. .(This part was done by Omar.)

```
0       0.0
1       2.0
2       1.5
3       2.3
4       1.8
       ...
6303    6.8
6304    4.0
6305    6.6
6306    6.5
6307    6.8
Name: engine_capacity, Length: 6305, dtype: float64
```

```python
df['seats'] = df['seats'].str.replace(r'[^\d.]', '', regex=True)
df['seats'] = df['seats'].where(df['seats'].notnull(), None)
df['seats'] = pd.to_numeric(df['seats'], errors='coerce')


print((df['top_speed'] == 'Automatic' ).sum())
print((df['seats'] >40 ).sum())
for index, row in df.iterrows():
    if isinstance(row['top_speed'], str):
        if "Seater" in row['top_speed']:
            parts = row['top_speed'].split()  # Split the string
            if len(parts) == 2 and parts[1] == "Seater" and parts[0].isdigit():

                df.at[index, 'seats'] = parts[0] # Move the value to the "seats" column

                df.at[index, 'top_speed'] = None # Set the "top_speed" column to None

    if row['seats'] >40:
        df.at[index, 'top_speed'] = row['seats']
print((df['top_speed'] == 'Automatic' ).sum())
```

```
...    86
       79
       10
```

This code cleans and processes the seats and top_speed columns in a dataset. It begins by removing non-numeric characters from the seats column and replacing missing values with None. The seats column is then converted to a numeric type, with invalid entries replaced by NaN. The code iterates through each row to check if the top_speed column contains a string. If the string includes the word "Seater," the number of seats is extracted and moved to the seats column, while the corresponding top_speed value is set to None. Additionally, if the seats column contains values greater than 40 (likely outliers), they are used to update the top_speed column. Finally, the script prints summary checks for values in the top_speed column and unusual values in the seats column. .(This part was done by Omar.)

```python
df['top_speed'] = pd.to_numeric(df['top_speed'], errors='coerce')
top_speed_median = df.groupby('brand')['top_speed'].transform('median')
df['top_speed'] = df.apply(
    lambda row: top_speed_median[row.name] if pd.isna(row['top_speed']) or row['top_speed'] > 400 else row['top_speed'],
    axis=1
)
df['top_speed'].fillna(df['top_speed'].median(), inplace=True)
df['top_speed'] = df['top_speed'].astype(int)

print(df['top_speed'].head())

```

```
0    150
1    205
2    145
3    175
4    190
```

This code processes the top_speed column to ensure clean and accurate data. First, it converts the column to numeric, replacing invalid entries with NaN. Then, it calculates the median top_speed for each brand and uses this value to fill in missing or invalid speeds (like those above 400, considered outliers). If no valid brand median is available, the overall median top_speed is used as a fallback. The processed data is converted to integer format, and the first five entries are printed for validation. This ensures the top_speed data is complete and reliable for further analysis. .(This part was done by Malak.)

```python
print(df['engine_capacity'].isnull().sum())
print(df['engine_capacity'].isnull().sum())
print(df['cylinder'].isnull().sum())
print(df['horse_power'].isnull().sum())
print(df['brand'].isnull().sum())
print(df['country'].isnull().sum())
print(df['top_speed'].isnull().sum())
print(df['price'].isnull().sum())
print(df['seats'].isnull().sum())
```

```
0
0
734
0
0
0
0
0
0
```

This code checks for missing values (nulls) in critical columns of the dataset, such as engine_capacity, cylinder, horse_power, brand, country, price, top_speed, and seats. For each column, the total number of missing values is calculated and displayed. The results show that most columns have no missing values, except for the cylinder column, which has 734 missing entries. This is expected because electric cars, which are part of the dataset, do not have cylinders. These checks ensure that missing values are understood and appropriately handled during data preprocessing. .(This part was done by Omar.)

```python
# Loop through each column and print the unique non-duplicate value
for column in df.columns:
    print(f"Column: {column}")
    unique_values = df[column].drop_duplicates()  # Drop duplicates from the column
    print(unique_values.tolist())  # Print the values as a list
    print("\n")


```

```
Column: car name
['Fiat 500e 2021 La Prima', 'Peugeot Traveller 2021 L3 VIP', 'Suzuki Jimny 2021 1.5L Automatic', 'Ford Bronco 2021 2.3T Big Bend', 'Honda


Column: price
[19219.95, 37955.25, 26671.95, 53460.0, 38005.2, 25740.45, 22368.15, 20667.15, 31563.000000000004, 64260.00000000001, 24808.95, 19530.45,


Column: engine_capacity
[0.0, 2.0, 1.5, 2.3, 1.8, 2.5, 2.7, 5.2, 4.0, 3.5, 3.8, 1.6, 3.0, 6.2, 3.7, 6.5, 1.7, 1.4, 2.2, 2.4, 5.0, 6.7, 4.4, 5.7, 3.6, 1.2, 3.3, 2


Column: cylinder
[0.0, 4.0, 6.0, 12.0, 8.0, 3.0, 5.0, 10.0, 16.0]


Column: horse_power
[100.0, 180.0, 102.0, 420.0, 140.0, 120.0, 170.0, 542.0, 900.0, 198.0, 700.0, 152.0, 176.0, 503.0, 530.0, 355.0, 121.0, 400.0, 335.0, 168


Column: top_speed
[150, 205, 145, 175, 190, 170, 199, 314, 200, 180, 185, 322, 240, 300, 184, 250, 210, 312, 236, 181, 216, 158, 230, 172, 189, 320, 220, 2


Column: seats
...
Column: country
['ksa', 'egypt', 'bahrain', 'qatar', 'oman', 'kuwait', 'uae']
```

The dataset has been thoroughly cleaned and processed. By iterating through each column and printing its unique values, we confirmed that the data is now consistent and free from issues such as mixed data types or incorrect entries. Columns like “price”, “engine_capacity”, and “horse_power” have been standardized, while country names and other categorical values have been validated for consistency. This ensures the dataset is now clean and ready for further analysis or modeling. .(This part was done by Malak.)

Scaling and Encoding

Frequency Encoding for brand:

```python
# Frequency Encoding for 'brand'
brand_freq = df['brand'].value_counts() / len(df)
df['brand'] = df['brand'].map(brand_freq)
print(df.info())

```

```
<class 'pandas.core.frame.DataFrame'>
Index: 6305 entries, 0 to 6307
Data columns (total 9 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   car name         6305 non-null   object 
 1   price            6305 non-null   float64
 2   engine_capacity  6305 non-null   float64
 3   cylinder         6305 non-null   float64
 4   horse_power      6305 non-null   float64
 5   top_speed        6305 non-null   int32  
 6   seats            6305 non-null   int32  
 7   brand            6305 non-null   float64
 8   country          6305 non-null   object 
dtypes: float64(5), int32(2), object(2)
memory usage: 572.4+ KB
None
```

This step applies frequency encoding to the "brand" column, converting it into a numerical representation based on how frequently each brand appears in the dataset. First, the code calculates the frequency of each brand as a proportion of the total rows in the dataset. Then, it replaces the original brand names with their corresponding frequency values. This encoding helps the machine learning model treat the brand as a numerical feature without introducing unnecessary bias, ensuring that more common brands are represented appropriately in the analysis. .(This part was done by Malak.)

Scaling all the necessary columns

```python
from sklearn.preprocessing import StandardScaler

# Select numerical columns for scaling
numerical_cols = ['engine_capacity', 'cylinder', 'horse_power', 'top_speed', 'seats','brand']

# Initialize the scaler
scaler = StandardScaler()

# Scale the numerical columns
df[numerical_cols] = scaler.fit_transform(df[numerical_cols] )

# Check the DataFrame after scaling
print(df.head())
```

```
                          car name     price  engine_capacity  cylinder  \
0           Fiat 500e 2021 La Prima  19219.95        -2.061910 -2.719019
1      Peugeot Traveller 2021 L3 VIP  37955.25        -0.597165 -0.663454
2  Suzuki Jimny 2021 1.5L Automatic  26671.95        -0.963351 -0.663454
3     Ford Bronco 2021 2.3T Big Bend  53460.00        -0.377453 -0.663454
4      Honda HR-V 2021 1.8 i-VTEC LX  38005.20        -0.743640 -0.663454

   horse_power  top_speed     seats     brand country
0    -1.073431  -1.638619 -0.676441 -1.145386     ksa
1    -0.625448  -0.362891  1.973015 -0.731837     ksa
2    -1.062232  -1.754595 -0.676441 -1.058973     ksa
3     0.718501  -1.058743 -0.676441  0.539672     ksa
4    -0.849440  -0.710817 -0.014077 -0.651596     ksa
```

The negative values in the scaled data indicate that certain data points are below the average (mean) for their respective features, while positive values represent data points above the average. This transformation is achieved using "StandardScaler", which standardizes numerical features (e.g., engine capacity, cylinder, horsepower, top speed, seats, and brand) to have a mean of 0 and a standard deviation of 1. This process, called standardization, ensures all features are on the same scale, preventing any one feature from dominating the learning process. This is especially important for machine learning models that are sensitive to magnitude differences, like gradient descent-based algorithms. The negative values are a normal result of this scaling process and confirm that the data has been successfully normalized, making it ready for effective model training and analysis. .(This part was done by Omar.)

```python
df = df.drop(columns=['car name', 'country'])
print(df.columns)
```

```
Index(['price', 'engine_capacity', 'cylinder', 'horse_power', 'top_speed',
       'seats', 'brand'],
      dtype='object')
```

The columns "car name" and "country" were dropped from the dataset as they don't provide meaningful contributions to the efficiency of the machine learning models. These columns are either not directly related to the predictive task or are redundant for our analysis. By removing them, we ensure that the model focuses on the most relevant features, improving its performance and simplifying the dataset. .(This part was done by Omar.)

Splitting the data

Splitting the data into three sets. Training (60%). Validation(20%). Test (20%)

```python
from sklearn.model_selection import train_test_split

X = df.drop(columns=['price'])  # All columns except 'price' because it's the target column (labe)
y = df['price']                 # Target column

# Splitting the data  into 60% training and 40% temp (validation + testing)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42)

# Splitting the 40% temp into 20% validation and 20% testing
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

# Verify the sizes of the splits
print(f"Training set size: {(X_train.shape[0]/len(df)) * 100:.0f}% of the data")
print(f"Validation set size: {(X_val.shape[0]/len(df)) * 100:.0f}% of the data")
print(f"Testing set size: {(X_test.shape[0]/len(df)) * 100:.0f}% of the data")
```

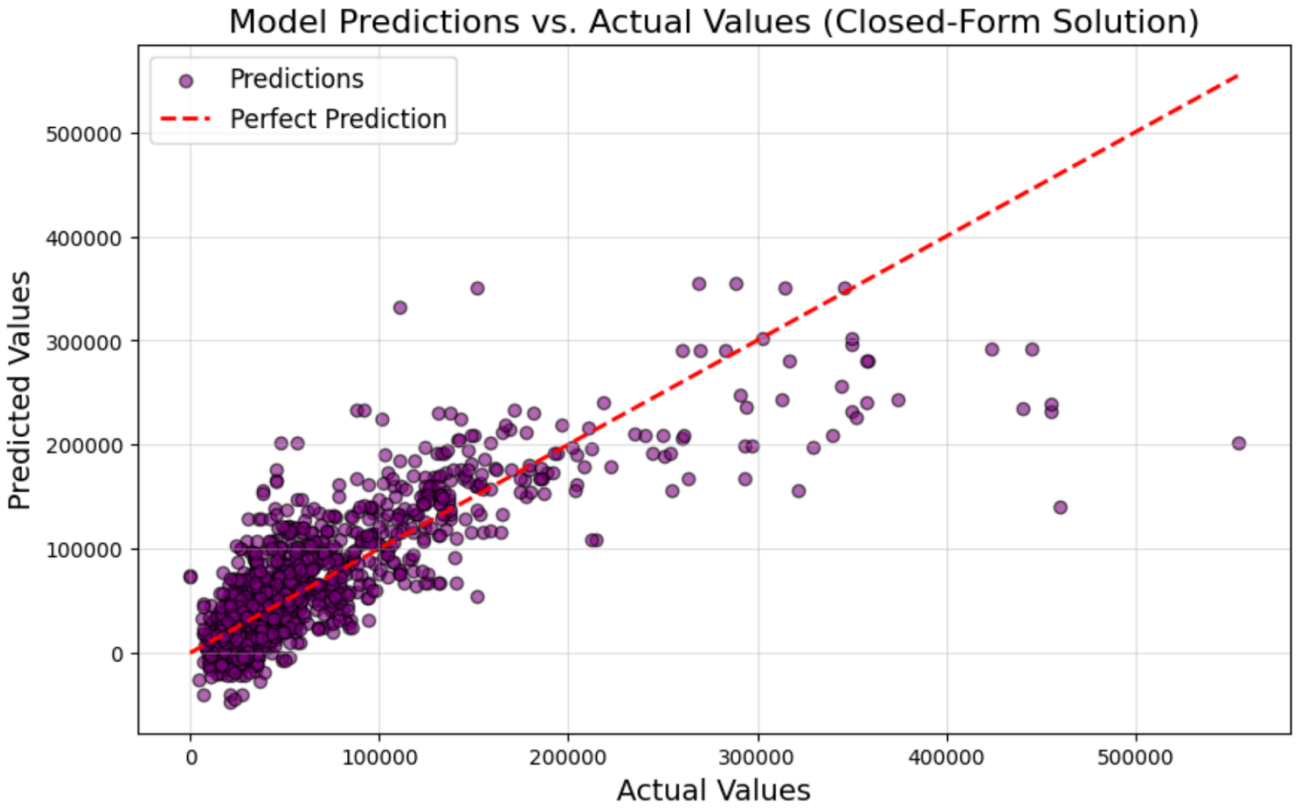
```
Training set size: 60% of the data
Validation set size: 20% of the data
Testing set size: 20% of the data
```

The code divides the data into three parts: 60% for training the model, 20% for validation (to fine-tune and pick the best model), and 20% for testing how well the model performs on unseen data. It separates the target column “price” from the features and uses the “train_test_split” function to first split the training data and then split the remaining data equally into validation and test sets. The “random_state=42” ensures consistency, meaning the same data points end up in the same sets every time the code runs. This setup helps the model learn effectively while ensuring it can perform well on new data. The printed sizes confirm the splits are correct. .(This part was done by Omar and Malak.)

```python
X_train_bias = np.c_[np.ones(X_train.shape[0]), X_train]
X_val_bias = np.c_[np.ones(X_val.shape[0]), X_val]

# Closed-form solution: theta = (X^T X)^(-1) X^T y
theta_closed_form = np.linalg.inv(X_train_bias.T @ X_train_bias) @ X_train_bias.T @ y_train


y_val_pred = X_val_bias @ theta_closed_form

print("Coefficients (Closed-Form):", theta_closed_form)

```

```
Coefficients (Closed-Form): [ 70951.02044169 -16872.79322384  18073.39894133  56447.98347037
  17379.46930316  -7290.16736874 -10506.42728356]
```

This code calculates the coefficients for a linear regression model using a closed-form solution, which is a precise mathematical approach to find the best fit for the data. It starts by adding a column of ones (bias term) to the training and validation data to account for the intercept in the regression equation. Using the formula (X^T X)^{-1} X^T y, it computes the coefficients that best describe the relationship between the features (e.g., engine capacity, horsepower) and the target variable (price). The results include a set of weights: positive values indicate features that increase the predicted price, while negative values represent features that decrease it. The intercept is also included as the first value. This method provides an exact solution for linear regression and reveals the impact of each feature on the car price. .(This part was done by Omar.)



Purple Points: Each point represents a data sample. The position of a point indicates the actual price (x-axis) and the model's prediction (y-axis) for that sample.

Red Dashed Line: This represents the ideal case of perfect prediction, where the predicted value equals the actual value (y=xy=x). Points closer to this line indicate better predictions.

Spread: The scatter around the red line shows how well the model performs. Points that deviate significantly from the line indicate errors in prediction, with underpredictions below the line and overpredictions above it.

Most points are close to this line, meaning the model performs well for many samples. However, some points are far from the line, showing errors in the model's predictions, especially for very high prices. Overall, the plot indicates that the model captures the general relationship between features and price but could be improved for extreme values.

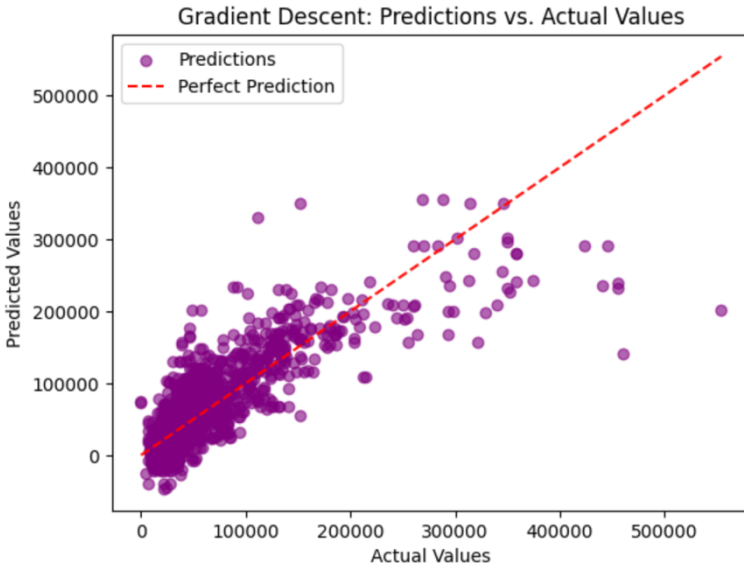
Gradient Descent (This part was done by Malak.)

```python
theta = np.zeros(X_train_bias.shape[1])
alpha = 0.1  # Learning rate
num_iterations = 1000  # Number of iterations

# Gradient Descent Loop
for i in range(num_iterations):
    gradient = (1 / X_train_bias.shape[0]) * (X_train_bias.T @ (X_train_bias @ theta - y_train))
    theta -= alpha * gradient

y_val_pred_gd = X_val_bias @ theta

print("Coefficients (Gradient Descent):", theta)
```

```
Coefficients (Gradient Descent): [ 70951.0213073  -16872.6268649   18073.22356114  56447.97283439
  17379.50134261  -7290.17907052 -10506.42973262]
```

It initializes the parameter vector theta with zeros, sets the learning rate (alpha = 0.1), and specifies the number of iterations (num_iterations = 1000). In the loop, the gradient is calculated using the derivative of the loss function (mean squared error), which measures the difference between predicted and actual values. The parameters (theta) are updated iteratively in the direction that minimizes the loss, scaled by the learning rate. The final theta values represent the coefficients of the regression model.

The results show the computed coefficients for the model (70951.02, -16872.63, ...), which are similar to those obtained with the closed-form solution, confirming that gradient descent successfully approximates the same parameters. This iterative method is particularly useful for large datasets or when the closed-form solution is computationally expensive. The similarity in results indicates that the algorithm has converged effectively. .



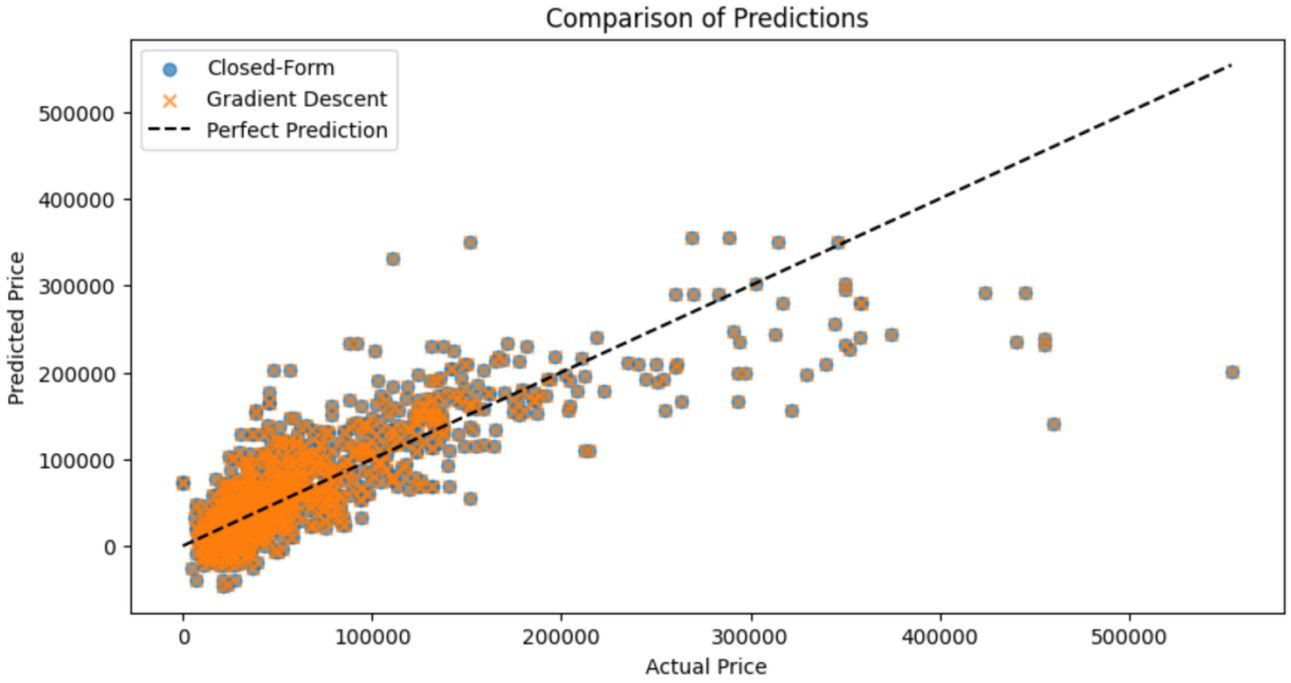
Most dots are close to this line, meaning the model's predictions are quite accurate. However, for higher prices, some dots are farther from the line, indicating the model isn't perfect and struggles slightly with expensive cars. Overall, the model does a good job predicting prices.

Validation of both solutions

```python
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Ensure X_val_bias has the bias term
if X_val_bias.shape[1] != theta_closed_form.shape[0]:
    X_val_bias = np.hstack((np.ones((X_val.shape[0], 1)), X_val))

y_val_pred_cf = X_val_bias @ theta_closed_form

y_val_pred_gd = X_val_bias @ theta

metrics = {
    "MSE": {
        "Closed-Form": mean_squared_error(y_val, y_val_pred_cf),
        "Gradient Descent": mean_squared_error(y_val, y_val_pred_gd)
    },
    "MAE": {
        "Closed-Form": mean_absolute_error(y_val, y_val_pred_cf),
        "Gradient Descent": mean_absolute_error(y_val, y_val_pred_gd)
    },
    "R^2": {
        "Closed-Form": r2_score(y_val, y_val_pred_cf),
        "Gradient Descent": r2_score(y_val, y_val_pred_gd)
    }
}

print("Validation Metrics:")
for metric, results in metrics.items():
    print(f"{metric}:")
    for method, value in results.items():
        print(f"  {method}: {value:.4f}")

# Compare Predictions
pred_diff = y_val_pred_cf - y_val_pred_gd
print("\nPrediction Differences:")
print(f"Max Absolute Difference: {np.max(np.abs(pred_diff)):.4f}")
print(f"Are predictions close (within 1e-5 tolerance)? {np.allclose(y_val_pred_cf, y_val_pred_gd, atol=1e-5)}")

```

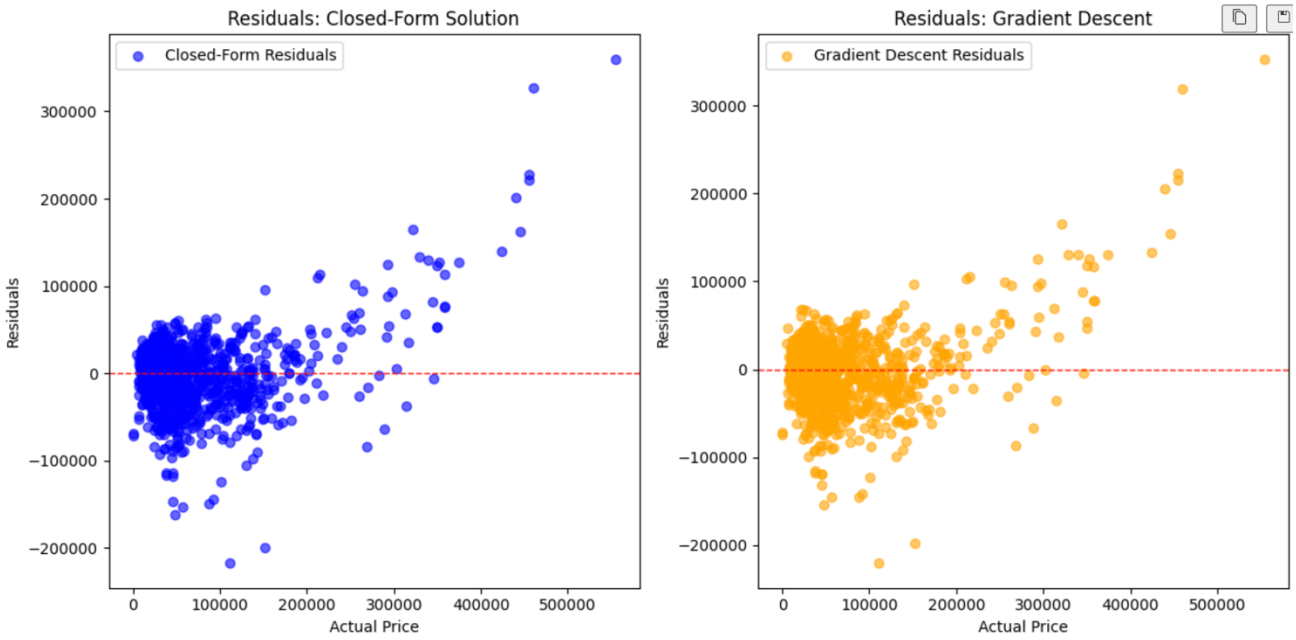
```
Validation Metrics:
MSE:
  Closed-Form: 1596298836.9868
  Gradient Descent: 1596298728.1574
MAE:
  Closed-Form: 26888.5384
  Gradient Descent: 26888.5387
R^2:
  Closed-Form: 0.6367
  Gradient Descent: 0.6367

Prediction Differences:
Max Absolute Difference: 0.5983
Are predictions close (within 1e-5 tolerance)? False
```

This code evaluates and compares the performance of two linear regression methods: the Closed-Form Solution and Gradient Descent, using metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-squared (R^2). Both methods were used to predict car prices on the validation dataset "y_val". The results show that the MSE (average squared error) and MAE (average absolute error) are almost identical for both methods, with MSE around 15.96 million and MAE at approximately 26,888. The R^2 score, which measures how well the model explains the variance in the data, is 0.6367 for both methods, indicating moderate predictive accuracy. The predictions from both methods are very close, with a maximum absolute difference of 0.5983, but they are not identical to within the tolerance of "1e-5". These similarities confirm that both approaches yield nearly the same model performance, as expected when properly implemented. The small differences in predictions arise due to computational precision and iterative updates in Gradient Descent. .(This part was done by Omar and Malak.)



Most points are close to the line, meaning both methods predict prices accurately. The overlap between the blue and orange points shows that both methods give very similar results. Some points are farther from the line, which means there are a few errors in predictions, and outliers indicate cars with unusual price differences. Overall, the plot confirms that both methods work well and produce nearly identical predictions.

The residuals (difference between actual and predicted prices) for both the Closed-Form and Gradient Descent methods;



The patterns in the residuals are similar for both methods, confirming their comparable performance. Outliers with large residuals are visible in both plots, showing instances where the model's predictions deviate significantly from the actual prices. These outliers might represent unique or extreme cases that the model fails to generalize. Overall, the residuals highlight areas of strong performance and limitations in predicting higher prices or handling rare cases. .(This part was done by Omar.)

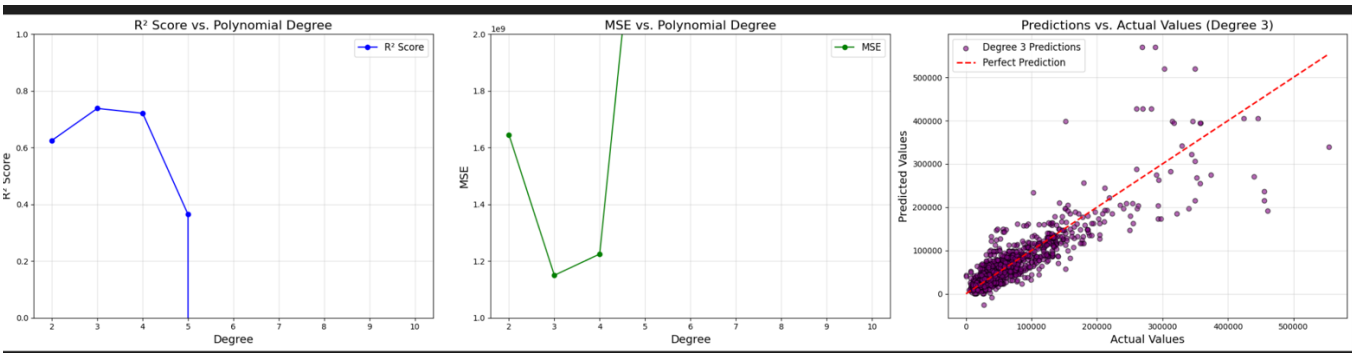
```python
# Variables to store results for each degree
degrees = range(2, 11)
results = {
    "degree": [],
    "r2_scores": [],
    "mse_scores": [],
    "mae_scores": [],
    "y_actual": None,
    "y_pred_best": None,
    "best_degree": None,
}

# Apply Polynomial Features
for degree in degrees:
    poly = PolynomialFeatures(degree)
    X_train_poly = poly.fit_transform(X_train)
    X_val_poly = poly.transform(X_val)

    # Fit Linear Regression to Polynomial Features
    model = LinearRegression()
    model.fit(X_train_poly, y_train)

    # Evaluate on validation set
    y_val_pred_poly = model.predict(X_val_poly)
    r2 = model.score(X_val_poly, y_val)
    mse = mean_squared_error(y_val, y_val_pred_poly)
    mae = mean_absolute_error(y_val, y_val_pred_poly)

    # Store the results
    results["degree"].append(degree)
    results["r2_scores"].append(r2)
    results["mse_scores"].append(mse)
    results["mae_scores"].append(mae)
```

```python
# Store the results
results["degree"].append(degree)
results["r2_scores"].append(r2)
results["mse_scores"].append(mse)
results["mae_scores"].append(mae)

# Check and store the best degree based on R² score
if results["best_degree"] is None or r2 > max(results["r2_scores"][:-1]):
    results["best_degree"] = degree
    results["y_actual"] = y_val
    results["y_pred_best"] = y_val_pred_poly

# Print results for the current degree
print(f"Degree {degree}:")
print(f"  R^2 Score: {r2:.4f}")
print(f"  Mean Squared Error (MSE): {mse:.4f}")
print(f"  Mean Absolute Error (MAE): {mae:.4f}")
print("\n")
```

Polynomial Regression .(This part was done by Omar.)

```
Degree 2:
  R^2 Score: 0.6255
  Mean Squared Error (MSE): 1645494705.4166
  Mean Absolute Error (MAE): 22472.4879


Degree 3:
  R^2 Score: 0.7383
  Mean Squared Error (MSE): 1149856355.0637
  Mean Absolute Error (MAE): 19414.2500


Degree 4:
  R^2 Score: 0.7213
  Mean Squared Error (MSE): 1224794020.2262
  Mean Absolute Error (MAE): 17378.5036


Degree 5:
  R^2 Score: 0.3644
  Mean Squared Error (MSE): 2792926928.8157
  Mean Absolute Error (MAE): 15916.7251


Degree 6:
...
  Mean Squared Error (MSE): 263909184589368896.0000
  Mean Absolute Error (MAE): 15879918.4058
```

The code applies polynomial regression to the dataset by transforming the features into polynomial features of increasing degrees (from 2 to 10) and evaluating the model's performance on a validation set. For each degree, the model calculates the R^2 score (goodness of fit), Mean Squared Error (MSE), and Mean Absolute Error (MAE) to measure accuracy. The results show that a degree of 3 achieves the best performance with the highest R^2 score (0.7383) and the lowest MSE and MAE compared to other degrees. For degrees beyond 3, the performance deteriorates, as shown by lower R^2 scores and higher errors, likely due to overfitting. Overfitting occurs when the model becomes too complex, capturing noise in the training data rather than generalizable patterns, which explains the worse results for degrees 5 and above. The best polynomial degree (3) strikes a balance between complexity and accuracy, capturing meaningful relationships without overfitting.



.(This part was done by Malak.)

1. **R² Score vs. Polynomial Degree (Left Plot):** This graph shows how accurately the models fit the data. The accuracy improves up to degree 3, where the model best captures the relationship between features and price. Beyond degree 3, the accuracy drops because the model becomes too complex and starts overfitting, meaning it focuses too much on small details in the data rather than the overall trend.

2. **MSE vs. Polynomial Degree (Middle Plot):** This graph displays the prediction error for each model. The error decreases as we move from degree 2 to degree 3, but it jumps significantly for higher degrees (like degree 5). This is because overly complex models struggle to make reliable predictions, reinforcing that simpler models like degree 3 work better.

3. **Predictions vs. Actual Values (Right Plot):** This scatter plot compares actual prices to predicted prices using the best-performing model (degree 3). Most points are close to the red "perfect prediction" line, meaning the predictions are accurate. This shows degree 3 strikes the right balance, capturing enough detail without being overly complex.

In short, degree 3 gives the best results, balancing accuracy and simplicity, making it the ideal choice for predicting car prices.

Radial Basis Function (RBF) .(This part was done by Omar.)

```python
from sklearn.model_selection import GridSearchCV
from sklearn.kernel_ridge import KernelRidge
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# 3. Kernel Ridge Regression (RBF) with Grid Search
kr_params = {'alpha': [0.1, 0.01, 0.001, 1, 10, 100]}
kernel_ridge = KernelRidge(kernel='rbf')
kr_grid = GridSearchCV(estimator=kernel_ridge, param_grid=kr_params, scoring='r2', cv=5
kr_grid.fit(X_train, y_train)

# Print best parameters and R^2 scores
print("Best Kernel Ridge Alpha (RBF):", kr_grid.best_params_['alpha'])
print("Best Kernel Ridge R^2 Score:", kr_grid.best_score_)
print("Kernel Ridge R^2 on Validation Set:", kr_grid.score(X_val, y_val))

# Predict on the validation set
y_val_pred_kr = kr_grid.predict(X_val)

# Compute validation metrics
mse_kr = mean_squared_error(y_val, y_val_pred_kr)
mae_kr = mean_absolute_error(y_val, y_val_pred_kr)
r2_kr = r2_score(y_val, y_val_pred_kr)

# Print validation metrics
print(f"Kernel Ridge Validation Metrics:")
print(f"  Mean Squared Error (MSE): {mse_kr:.4f}")
print(f"  Mean Absolute Error (MAE): {mae_kr:.4f}")
print(f"  R^2 Score: {r2_kr:.2f}")
```

```
Best Kernel Ridge Alpha (RBF): 0.1
Best Kernel Ridge R^2 Score: 0.6235025153597876
Kernel Ridge R^2 on Validation Set: 0.8336934727317188
Kernel Ridge Validation Metrics:
  Mean Squared Error (MSE): 730756833.8219
  Mean Absolute Error (MAE): 14547.5074
  R^2 Score: 0.83
```

This code implements Kernel Ridge Regression with a Radial Basis Function (RBF) kernel, optimizing the regularization parameter `alpha` using GridSearchCV. The “alpha”parameter controls the model's complexity and regularization, and the grid search evaluates multiple values [0.1, 0.01, 0.001, 1, 10, 100] using cross-validation to find the value that maximizes the R^2 score. After training the model, the validation set is used to calculate performance metrics: Mean Squared Error (MSE), Mean Absolute Error (MAE), and the R^2 score, which reflects how well the model explains the variance in the data. The best “alpha” value was found to be 0.1, resulting in an R^2 score of 0.83, indicating a strong fit, while the MSE and MAE highlight the model's predictive accuracy. These results suggest that the model effectively balances complexity and generalization, making it suitable for predicting the target variable in the dataset.

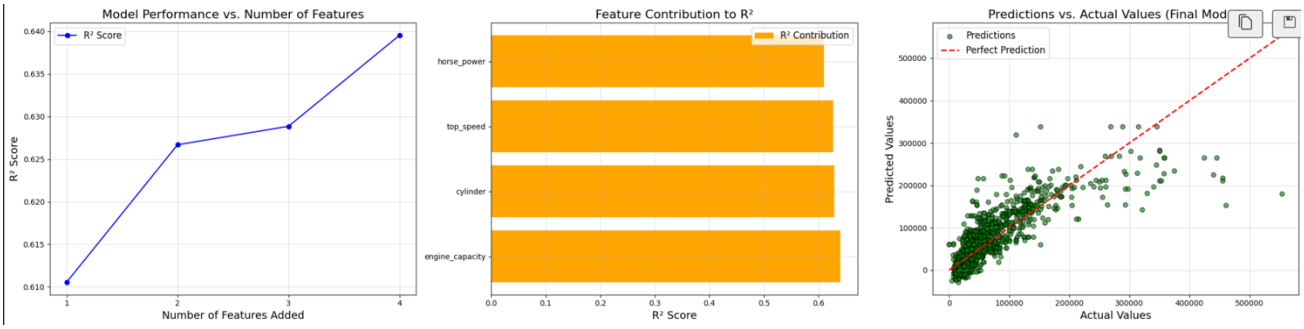
Feature Selection (Forward Selection) .(This part was done by Malak.)

```python
# Start with an empty set of features
selected_features = []
remaining_features = list(X_train.columns)
best_score = -np.inf
r2_scores = []
mse_scores = []
mae_scores = []
print("Forward Selection Process with Validation Metrics:\n")

while remaining_features:
    scores = {}
    metrics = {}

    for feature in remaining_features:
        # Add feature to the model
        trial_features = selected_features + [feature]
        X_train_subset = X_train[trial_features]
        X_val_subset = X_val[trial_features]

        # Train Linear Regression
        model = LinearRegression()
        model.fit(X_train_subset, y_train)

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

        # Evaluate on validation set
        r2 = model.score(X_val_subset, y_val)
        mse = mean_squared_error(y_val, y_val_pred)
        mae = mean_absolute_error(y_val, y_val_pred)

        scores[feature] = r2
        metrics[feature] = {'R²': r2, 'MSE': mse, 'MAE': mae}

    # Find the best feature to add
    best_feature = max(scores, key=scores.get)

    if scores[best_feature] > best_score:
        r2_scores.append(metrics[best_feature]['R²'])
        mse_scores.append(metrics[best_feature]['MSE'])
        mae_scores.append(metrics[best_feature]['MAE'])
        selected_features.append(best_feature)
        remaining_features.remove(best_feature)
        best_score = scores[best_feature]

        # Print the metrics for the selected feature
        print(f"Selected Feature: {best_feature}")
        print(f"  R² Score: {metrics[best_feature]['R²']:.4f}")
        print(f"  Mean Squared Error (MSE): {metrics[best_feature]['MSE']:.4f}")
        print(f"  Mean Absolute Error (MAE): {metrics[best_feature]['MAE']:.4f}\n")

    else:
        break  # Stop if no improvement

print("\nFinal Selected Features:", selected_features)
```

```
Forward Selection Process with Validation Metrics:

Selected Feature: horse_power
  R² Score: 0.6105
  Mean Squared Error (MSE): 1711323661.6795
  Mean Absolute Error (MAE): 25856.2765

Selected Feature: top_speed
  R² Score: 0.6267
  Mean Squared Error (MSE): 1640374057.2535
  Mean Absolute Error (MAE): 27116.8768

Selected Feature: cylinder
  R² Score: 0.6289
  Mean Squared Error (MSE): 1630841865.2239
  Mean Absolute Error (MAE): 27104.2940

Selected Feature: engine_capacity
  R² Score: 0.6395
  Mean Squared Error (MSE): 1583893015.2878
  Mean Absolute Error (MAE): 26376.3469


Final Selected Features: ['horse_power', 'top_speed', 'cylinder', 'engine_capacity']
```

It evaluates each remaining feature by adding it to the model and calculating performance metrics: R^2 (how well the model fits the data), MSE (Mean Squared Error), and MAE (Mean Absolute Error). After each iteration, the feature that improves the model the most (highest R^2) is added to the selected features list, and the process repeats until no further improvement is observed.

In the results, features like horse_power, top_speed, cylinder, and engine_capacity were selected sequentially, as each added significant predictive power. For example, horse_power was the first to be selected, achieving an R^2 of 0.6105, while engine_capacity was added last, improving the R^2 to 0.6395. This step-by-step approach ensures the model uses only the most impactful features, reducing overfitting and enhancing interpretability. The final selected features represent the most influential predictors for the target variable in this dataset.



The left plot shows how the model's accuracy (R^2 score) improves as more features are added. Initially, adding features makes a big difference, but the improvement slows down after the most impactful features are included, confirming their importance in making accurate predictions.

The middle plot highlights the contribution of each selected feature, such as “horse_power”, “top_speed”, “cylinder”, and “engine_capacity”, showing that they all play an almost equal role in boosting the model's accuracy.

The right plot compares the model's predictions to the actual values. Most predictions align closely with the red diagonal line (indicating a perfect prediction), although some errors remain, suggesting there’s still some room for improvement.

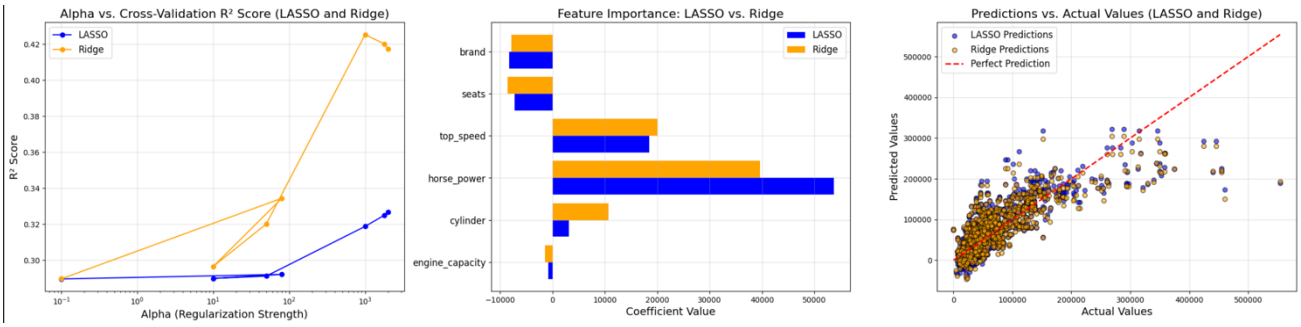
Overall, this process confirms that selecting these specific features significantly improves the model's accuracy, and the final model performs well at predicting the target variable.

LASSO and Ridge Regularization .(This part was done by Omar.)

```python
# 1. LASSO Regression with Grid Search
lasso_params = {'alpha': [0.1, 80, 10, 50, 1000, 1800, 2000]}
lasso = Lasso()
lasso_grid = GridSearchCV(estimator=lasso, param_grid=lasso_params, scoring='r2', cv=5)
lasso_grid.fit(X_train, y_train)
print("Best LASSO Alpha:", lasso_grid.best_params_['alpha'])
r2_lasso = lasso_grid.score(X_val, y_val)
print(f"LASSO R^2 on Validation Set: {r2_lasso:.3f}")

# Predict on the validation set
y_val_pred_lasso = lasso_grid.predict(X_val)

# Compute metrics
mse_lasso = mean_squared_error(y_val, y_val_pred_lasso)
mae_lasso = mean_absolute_error(y_val, y_val_pred_lasso)

print(f"LASSO Validation Metrics:")
print(f"  Mean Squared Error (MSE): {mse_lasso:.3f}")
print(f"  Mean Absolute Error (MAE): {mae_lasso:.3f}")


print("\n")

# 2. Ridge Regression with Grid Search
ridge_params = {'alpha': [0.1, 80, 10, 50, 1000, 1800, 2000]}
ridge = Ridge()
ridge_grid = GridSearchCV(estimator=ridge, param_grid=ridge_params, scoring='r2', cv=5)
ridge_grid.fit(X_train, y_train)
print("Best Ridge Alpha:", ridge_grid.best_params_['alpha'])
ridge_r2 = ridge_grid.score(X_val, y_val)
print(f"Ridge R^2 on Validation Set: {ridge_r2:.3f}" )

# Predict on the validation set
y_val_pred_ridge = ridge_grid.predict(X_val)
# Compute metrics
mse_ridge = mean_squared_error(y_val, y_val_pred_ridge)
mae_ridge = mean_absolute_error(y_val, y_val_pred_ridge)


print(f"Ridge Validation Metrics:")
print(f"  Mean Squared Error (MSE): {mse_ridge:.3f}")
print(f"  Mean Absolute Error (MAE): {mae_ridge:.3f}")


print("\n")

print("Cross-Validation LASSO R² Scores:", lasso_grid.cv_results_['mean_test_score'])
print("Cross-Validation Ridge R² Scores:", ridge_grid.cv_results_['mean_test_score'])
```

```
Best LASSO Alpha: 2000
LASSO R^2 on Validation Set: 0.644
LASSO Validation Metrics:
  Mean Squared Error (MSE): 1565725442.551
  Mean Absolute Error (MAE): 26291.665


Best Ridge Alpha: 1000
Ridge R^2 on Validation Set: 0.660
Ridge Validation Metrics:
  Mean Squared Error (MSE): 1495297464.160
  Mean Absolute Error (MAE): 24957.285


Cross-Validation LASSO R² Scores: [0.28961629 0.29214755 0.28993391 0.29120544 0.31880068 0.32504777
 0.32672505]
Cross-Validation Ridge R² Scores: [0.28968474 0.33440771 0.29654504 0.320153   0.42529274 0.42007526
 0.41731802]
```

This code compares the performance of LASSO and Ridge regression models by using grid search to find the optimal regularization parameter “alpha”. For LASSO regression, the best alpha value was found to be 2000, resulting in an R^2 score of 0.644 on the validation set. This indicates that the LASSO model explains 64.4% of the variance in the target variable. The Mean Squared Error (MSE) and Mean Absolute Error (MAE) for LASSO are 1565725442.551 and 26291.665, respectively. Ridge regression, with an optimal alpha of 1000, achieved a slightly better R^2 score of 0.660, an MSE of 1495297464.160, and an MAE of 24957.285, indicating better performance in capturing the target variable's patterns. The cross-validation R^2 scores for both models show moderate variability across folds, with Ridge generally performing slightly better than LASSO. This behavior highlights the importance of regularization in controlling overfitting and improving model generalization, especially with Ridge regression handling feature coefficients more smoothly.



➔ Regularization Strength :

This chart shows how the performance of Ridge and LASSO models changes with regularization strength (alpha). Ridge performs better because it adjusts coefficients more smoothly across all features, achieving better accuracy as alpha increases. LASSO is more aggressive and eliminates weaker features but doesn't handle all features as effectively at higher alpha values.

➔ Feature Contributions :

This bar chart highlights the difference in how LASSO and Ridge treat features. Ridge keeps all features active, while LASSO zeros out less important ones like cylinder and engine_capacity. Key features like horse_power and top_speed remain influential in both models, showing their strong link to the car price.

➔ Prediction Quality :

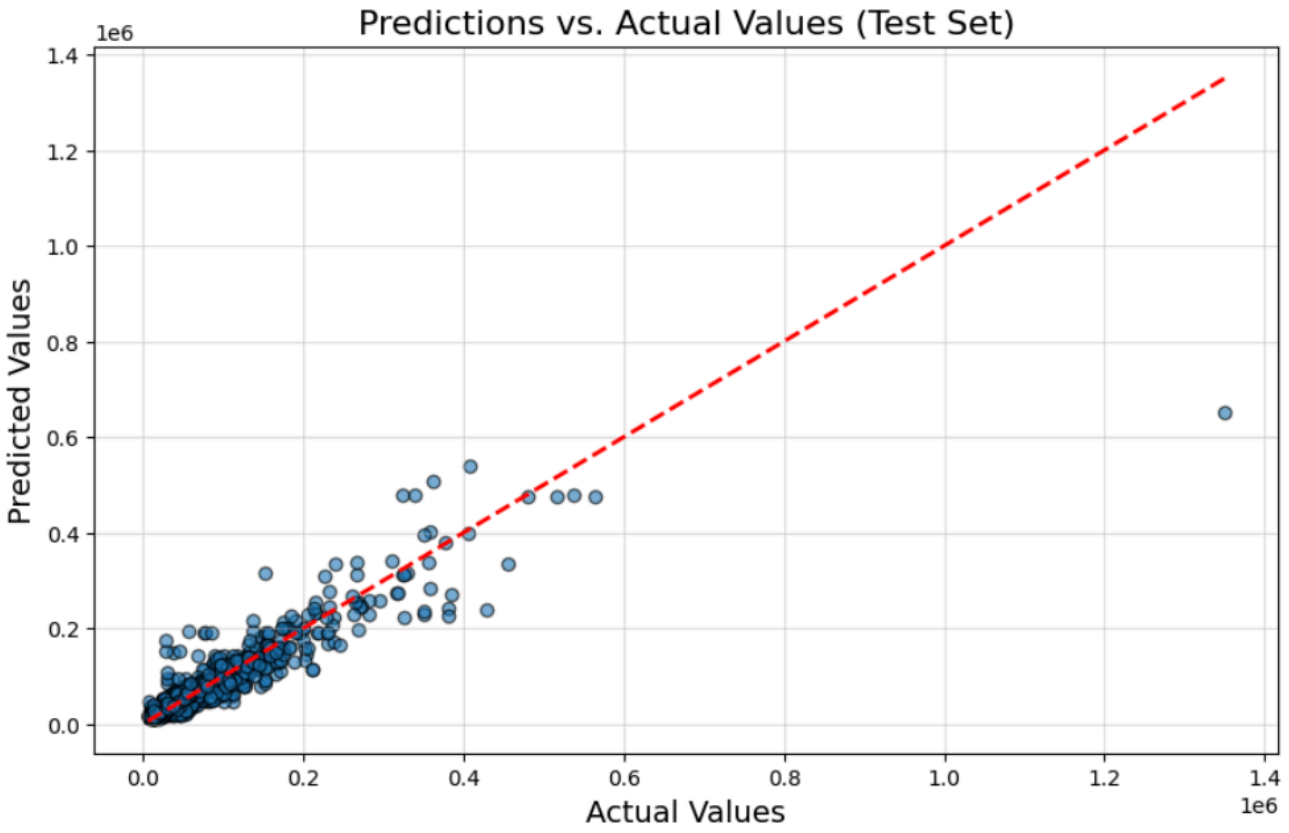
The scatter plot compares the predictions of both models with actual car prices. Ridge is slightly better at predicting accurately, as shown by points closer to the perfect prediction line. LASSO's predictions are good but slightly more scattered due to its stricter feature selection, which can miss some useful data patterns.

Model Evaluation on Test Set .(This part was done by Omar.)

```python
# Initialize the RBF model with the best alpha from Grid Search
best_alpha_rbf = 0.1  # Based on Grid Search results
rbf_model = KernelRidge(kernel='rbf', alpha=best_alpha_rbf)

# Fit the model on the training data
rbf_model.fit(X_train, y_train)

# Evaluate the model on the test set
y_test_pred = rbf_model.predict(X_test)

# Calculate performance metrics on the test set
mse_test = mean_squared_error(y_test, y_test_pred)
mae_test = mean_absolute_error(y_test, y_test_pred)
r2_test = r2_score(y_test, y_test_pred)

# Print the metrics
print("RBF Model Test Set Evaluation:")
print(f"Mean Squared Error (MSE): {mse_test:.3f}")
print(f"Mean Absolute Error (MAE): {mae_test:.3f}")
print(f"R² Score: {r2_test:.2f}")
```

```
RBF Model Test Set Evaluation:
Mean Squared Error (MSE): 1017114878.458
Mean Absolute Error (MAE): 14905.802
R² Score: 0.84
```

The RBF model, optimized with an alpha of 0.1 (based on Grid Search), achieved an R^2 score of 0.84, indicating that the model explains 84% of the variance in the target variable on the test set. The Mean Squared Error (MSE) is approximately 10,171,487.46, and the Mean Absolute Error (MAE) is around 14,905.80. These metrics suggest a relatively accurate model with some degree of error in predicting car prices.



Most points are aligned closely along the diagonal red line (representing perfect predictions), confirming that the model's predictions are generally accurate. However, a few outliers deviate significantly, particularly one extreme value, which may indicate noisy data or limitations in the model's ability to capture complex relationships. Overall, the RBF model performs well but can be improved to handle such outliers.

Conclusion

This project focused on predicting car prices using the YallaMotors Cars Dataset. We cleaned and prepared the data by handling missing values, encoding categories, and standardizing numerical features to ensure it was ready for analysis. The dataset was split into training, validation, and test sets to build and evaluate models effectively.

We implemented various regression models, including Linear Regression, Polynomial Regression, LASSO, Ridge Regression, and Kernel Ridge with an RBF kernel. Each model was evaluated using metrics like R^2, MSE, and MAE. Forward selection helped us choose the most important features, such as horsepower and top speed, while LASSO and Ridge regularization reduced overfitting. Among all models, Kernel Ridge Regression with an RBF kernel performed the best, achieving an R^2 score of 0.84 on the test set, showing strong accuracy in predicting car prices.

Overall, the project demonstrated the power of machine learning in solving real-world problems.