MUHAMMAD HUZAIFA OWAIS

Understanding AI (771763_C23_T3A)

Summative Assignment:

Exercise 1: Analysing Second Hand Car Sales Data with Supervised and Unsupervised Learning Models

In [2]:

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

In [3]:

```
df = pd.read_csv ("car_sales_data_24.csv")
df.head()
```

Out[3]:

	Manufacturer	Model	Engine size	Fuel type	Year of manufacture	Mileage	Price
0	Ford	Fiesta	1.0	Petrol	2002	127300	3074
1	Porsche	718 Cayman	4.0	Petrol	2016	57850	49704
2	Ford	Mondeo	1.6	Diesel	2014	39190	24072
3	Toyota	RAV4	1.8	Hybrid	1988	210814	1705
4	VW	Polo	1.0	Petrol	2006	127869	4101

PART (a)

Compare regression models that predict the price of a car based on a single numerical input feature. Based on your results, which numerical variable in the dataset is the best predictor for a car’s price, and why? For each numerical input feature, is the price better fit by a linear model or by a non-linear (e.g. polynomial) model?

Linear Regression

1) Price vs Mileage

In [4]:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

#Seperating independent and dependent variable
x = df["Mileage"]
y = df["Price"]
x = x.to_numpy().reshape(-1, 1)

#Splitting into test train
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 42)

#Standardization
scale = StandardScaler()
scale.fit(x)
x_train_scaled = scale.transform(x_train)
x_test_scaled = scale.transform(x_test)

#Applying linear Regression
Price_linear = LinearRegression()
Price_linear.fit(x_train_scaled, y_train)

#Prediction
Price_pred = Price_linear.predict(x_test_scaled)
```

```
#Calculating error
mae = mean_absolute_error(y_test, Price_pred)
mse = mean_squared_error(y_test, Price_pred)
rmse = np.sqrt(mse)
R2 = r2_score(y_test, Price_pred)
print(mae,mse,rmse,R2)
```

```
7964.784670024687 162468566.87254104 12746.315815659875 0.4013139100884707
```

2) Price vs Engine Size

In [5]:

```
#Seperating independent and dependent variable
x = df["Engine size"]
y = df["Price"]
x = x.to_numpy().reshape(-1, 1)

#Splitting into test train
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 42)

#Standardization
scale = StandardScaler()
scale.fit(x)
x_train_scaled = scale.transform(x_train)
x_test_scaled = scale.transform(x_test)

#Applying linear Regression
Price_linear = LinearRegression()
Price_linear.fit(x_train_scaled, y_train)

#Prediction
Price_pred = Price_linear.predict(x_test_scaled)

#Calculating error
mae = mean_absolute_error(y_test, Price_pred)
mse = mean_squared_error(y_test, Price_pred)
rmse = np.sqrt(mse)
R2 = r2_score(y_test, Price_pred)
print(mae,mse,rmse,R2)
```

```
10817.491562557905 230499154.45279127 15182.198604049128 0.15062562461380213
```

3) Price vs Year of manufacture

In [6]:

```
#Seperating independent and dependent variable
x = df["Year of manufacture"]
y = df["Price"]
x = x.to_numpy().reshape(-1, 1)

#Splitting into test train
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 42)

#Standardization
scale = StandardScaler()
scale.fit(x)
x_train_scaled = scale.transform(x_train)
x_test_scaled = scale.transform(x_test)

#Applying linear Regression
Price_linear = LinearRegression()
Price_linear.fit(x_train_scaled, y_train)

#Prediction
Price_pred = Price_linear.predict(x_test_scaled)

#Calculating error
mae = mean_absolute_error(y_test, Price_pred)
mse = mean_squared_error(y_test, Price_pred)
rmse = np.sqrt(mse)
R2 = r2_score(y_test, Price_pred)
print(mae,mse,rmse,R2)
```

```
7031.0392086748125 132678999.94793086 11518.637069893766 0.5110865244812854
```

Polynomial Regression

1) Price vs Mileage

In [7]:

```python
from sklearn.preprocessing import PolynomialFeatures

#Seperating independent and dependent variable
x = df["Mileage"]
y = df["Price"]
x = x.to_numpy().reshape(-1, 1)

# Splitting into test train
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

# Standardization
scale = StandardScaler()
scale.fit(x)
x_train_scaled = scale.transform(x_train)
x_test_scaled = scale.transform(x_test)

# Dictionary to store errors for different degrees
error_metrics = {}

# Loop to check errors for polynomial degrees from 2 to 9
for degree in range(2, 10):
    poly = PolynomialFeatures(degree=degree, include_bias=False)
    x_train_poly = poly.fit_transform(x_train_scaled)
    x_test_poly = poly.transform(x_test_scaled)

    #Fitting Model
    Price_poly = LinearRegression()
    Price_poly.fit(x_train_poly, y_train)

    #Prediction
    Price_pred = Price_poly.predict(x_test_poly)

    #Calculating error
    mae = mean_absolute_error(y_test, Price_pred)
    mse = mean_squared_error(y_test, Price_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_test, Price_pred)

    error_metrics[degree] = {'MAE': mae, 'MSE': mse, 'RMSE': rmse, 'R2': r2}
    print(f'Degree: {degree}, MAE: {mae}, MSE: {mse}, RMSE: {rmse}, R2: {r2}')
```

```
Degree: 2, MAE: 6409.911605271255, MSE: 129620312.1626197, RMSE: 11385.091662460154, R2: 0.5223575898060919
Degree: 3, MAE: 5815.669418610494, MSE: 122123243.4158437, RMSE: 11050.93857624065, R2: 0.5499837999721879
Degree: 4, MAE: 5719.6716155491085, MSE: 120800573.84612861, RMSE: 10990.931436694918, R2: 0.5548577512119925
Degree: 5, MAE: 5698.012248246026, MSE: 120626997.26255809, RMSE: 10983.032243536303, R2: 0.5554973696201464
```

```
Degree: 6, MAE: 5697.243777600846, MSE: 120618137.87224893, RMSE: 10982.62891443797, R2: 0.5555300158965633
Degree: 7, MAE: 5697.246650908994, MSE: 120618112.38919675, RMSE: 10982.627754285253, R2: 0.5555301097999512
Degree: 8, MAE: 5698.213368390338, MSE: 120615246.34783071, RMSE: 10982.497272835113, R2: 0.5555406709757651
Degree: 9, MAE: 5700.3653375402955, MSE: 120619907.69554448, RMSE: 10982.709487897077, R2: 0.5555234942129599
```

2) Price vs Engine Size

In [8]:

```
#Seperating independent and dependent variable
x = df["Engine size"]
y = df["Price"]
x = x.to_numpy().reshape(-1, 1)

# Splitting into test train
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

# Standardization
scale = StandardScaler()
scale.fit(x)
x_train_scaled = scale.transform(x_train)
x_test_scaled = scale.transform(x_test)

# Dictionary to store errors for different degrees
error_metrics = {}

# Loop to check errors for polynomial degrees from 2 to 9
for degree in range(2, 10):
    poly = PolynomialFeatures(degree=degree, include_bias=False)
    x_train_poly = poly.fit_transform(x_train_scaled)
    x_test_poly = poly.transform(x_test_scaled)

    #Fitting Model
    Price_poly = LinearRegression()
    Price_poly.fit(x_train_poly, y_train)

    #Prediction
    Price_pred = Price_poly.predict(x_test_poly)

    #Calculating error
    mae = mean_absolute_error(y_test, Price_pred)
    mse = mean_squared_error(y_test, Price_pred)
    rmse = np.sqrt(mse)
```

```
    r2 = r2_score(y_test, Price_pred)

    error_metrics[degree] = {'MAE': mae, 'MSE': mse, 'RMSE': rmse, 'R2': r2}
    print(f'Degree: {degree}, MAE: {mae}, MSE: {mse}, RMSE: {rmse}, R2: {r2}')
```

```
Degree: 2, MAE: 10807.262347148684, MSE: 230326165.9994691, RMSE: 15176.5004529855, R2: 0.1512630758002863
Degree: 3, MAE: 10802.86898273087, MSE: 230076036.26779428, RMSE: 15168.257522464282, R2: 0.15218478757450593
Degree: 4, MAE: 10801.446336622119, MSE: 230012047.3259903, RMSE: 15166.148071477817, R2: 0.1524205826584638
Degree: 5, MAE: 10801.138446122159, MSE: 230001460.47862178, RMSE: 15165.799038580913, R2: 0.15245959450166247
Degree: 6, MAE: 10802.342121404878, MSE: 230086961.99933666, RMSE: 15168.617669363832, R2: 0.15214452696519276
Degree: 7, MAE: 10801.317345438729, MSE: 230130261.5671455, RMSE: 15170.044876899525, R2: 0.15198497087723473
Degree: 8, MAE: 10801.067555182659, MSE: 230124725.03884053, RMSE: 15169.86239353675, R2: 0.15200537262353186
Degree: 9, MAE: 10802.550555267559, MSE: 230127220.21159396, RMSE: 15169.944634427442, R2: 0.15199617807440668
```

3) Price vs Year of manufacture

In [9]:

```
#Seperating independent and dependent variable
x = df["Year of manufacture"]
y = df["Price"]
x = x.to_numpy().reshape(-1, 1)

# Splitting into test train
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

# Standardization
scale = StandardScaler()
scale.fit(x)
x_train_scaled = scale.transform(x_train)
x_test_scaled = scale.transform(x_test)

# Dictionary to store errors for different degrees
error_metrics = {}

# Loop to check errors for polynomial degrees from 2 to 9
for degree in range(2, 10):
    poly = PolynomialFeatures(degree=degree, include_bias=False)
    x_train_poly = poly.fit_transform(x_train_scaled)
    x_test_poly = poly.transform(x_test_scaled)

    #Fitting Model
    Price_poly = LinearRegression()
```

```
    Price_poly.fit(x_train_poly, y_train)

    #Prediction
    Price_pred = Price_poly.predict(x_test_poly)

    #Calculating error
    mae = mean_absolute_error(y_test, Price_pred)
    mse = mean_squared_error(y_test, Price_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_test, Price_pred)

    error_metrics[degree] = {'MAE': mae, 'MSE': mse, 'RMSE': rmse, 'R2': r2}
    print(f'Degree: {degree}, MAE: {mae}, MSE: {mse}, RMSE: {rmse}, R2: {r2}')
```

```
Degree: 2, MAE: 5387.109074986957, MSE: 105993894.20194325, RMSE: 10295.33361295025, R2: 0.60941940157544
Degree: 3, MAE: 5186.868941344731, MSE: 103043508.14527172, RMSE: 10151.034831251034, R2: 0.6202913820821918
Degree: 4, MAE: 5162.883981090708, MSE: 102720854.73230511, RMSE: 10135.129734359847, R2: 0.621480338899649
Degree: 5, MAE: 5160.772689003934, MSE: 102654671.46530674, RMSE: 10131.864165360032, R2: 0.6217242199290662
Degree: 6, MAE: 5161.63220527173, MSE: 102627580.92078595, RMSE: 10130.527178818778, R2: 0.6218240468216457
Degree: 7, MAE: 5160.95024310547, MSE: 102626149.86664064, RMSE: 10130.456547788981, R2: 0.6218293201629892
Degree: 8, MAE: 5159.1645625747915, MSE: 102633206.52261665, RMSE: 10130.804830940959, R2: 0.6218033168452062
Degree: 9, MAE: 5159.318482863797, MSE: 102640880.42667185, RMSE: 10131.183564947969, R2: 0.6217750390084431
```

PART (b):

Consider regression models that take multiple numerical variables as input features to predict the price of a car. Does the inclusion of multiple input features improve the accuracy of the model's prediction compared to the single-input feature models that you explored in part (a)?

Linear Model

Price vs Engine Size, Year of Manufacture & Mileage

In [10]:

```
#Seperating independent and dependent variables
feature_names = ["Engine size", "Year of manufacture", "Mileage"]
```

```
x = df[feature_names]
y = df["Price"]

#Splitting into test train
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 42)

#Standardization
scale = StandardScaler()
scale.fit(x)
x_train_scaled = scale.transform(x_train)
x_test_scaled = scale.transform(x_test)

#Applying linear Regression
Price_linear = LinearRegression()
Price_linear.fit(x_train_scaled, y_train)

#Prediction
Price_pred = Price_linear.predict(x_test_scaled)

#Calculating error
mae = mean_absolute_error(y_test, Price_pred)
mse = mean_squared_error(y_test, Price_pred)
rmse = np.sqrt(mse)
R2 = r2_score(y_test, Price_pred)
print(mae,mse,rmse,R2)
```

```
6091.4581416562205 89158615.76017143 9442.38400829851 0.671456306417368
```

Polynomial Regression

Price vs Engine Size, Year of Manufacture & Mileage

In [11]:

```
#Seperating independent and dependent variables
feature_names = ["Engine size", "Year of manufacture", "Mileage"]
x = df[feature_names]
y = df["Price"]

# Splitting into test train
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
```

```python
# Standardization
scale = StandardScaler()
scale.fit(x)
x_train_scaled = scale.transform(x_train)
x_test_scaled = scale.transform(x_test)

# Dictionary to store errors for different degrees
error_metrics = {}

# Loop to check errors for polynomial degrees from 2 to 9
for degree in range(2, 10):
    poly = PolynomialFeatures(degree=degree, include_bias=False)
    x_train_poly = poly.fit_transform(x_train_scaled)
    x_test_poly = poly.transform(x_test_scaled)

    #Fitting Model
    Price_poly = LinearRegression()
    Price_poly.fit(x_train_poly, y_train)

    #Prediction
    Price_pred = Price_poly.predict(x_test_poly)

    #Calculating error
    mae = mean_absolute_error(y_test, Price_pred)
    mse = mean_squared_error(y_test, Price_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_test, Price_pred)

    error_metrics[degree] = {'MAE': mae, 'MSE': mse, 'RMSE': rmse, 'R2': r2}
    print(f'Degree: {degree}, MAE: {mae}, MSE: {mse}, RMSE: {rmse}, R2: {r2}')
```

```
Degree: 2, MAE: 3196.8249339763493, MSE: 29310960.84095311, RMSE: 5413.959811538419, R2: 0.8919910178614003
Degree: 3, MAE: 2323.5575068835483, MSE: 19627044.755805485, RMSE: 4430.24206514785, R2: 0.9276756180745401
Degree: 4, MAE: 2206.880841573635, MSE: 18490866.37703282, RMSE: 4300.100740335372, R2: 0.9318623614189501
Degree: 5, MAE: 2183.1372857333718, MSE: 18302200.042324003, RMSE: 4278.107062980543, R2: 0.932557584577491
Degree: 6, MAE: 2182.7075191089743, MSE: 18259118.146071818, RMSE: 4273.068937669016, R2: 0.9327163385599364
Degree: 7, MAE: 2169.106483752251, MSE: 18205554.818261806, RMSE: 4266.7967866142635, R2: 0.932913715935182
Degree: 8, MAE: 2142.468862644927, MSE: 18177921.926438417, RMSE: 4263.557426192172, R2: 0.9330155413422571
Degree: 9, MAE: 2180.3678368088176, MSE: 17996342.78383326, RMSE: 4242.209658165572, R2: 0.9336846486593731
```

PART (c)

In parts (a) and (b) you only considered models that use the numerical variables from the dataset as inputs. However, there are also several categorical variables in the dataset that are likely to affect the price of the car. Now train a regression model that uses all relevant input variables (both categorical and numerical) to predict the price (e.g. a Random Forest Regressor model). Does this improve the accuracy of your results?

In [12]:

```
# Applying Label Encoder
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df_encoded = df.copy()
df_encoded['Manufacturer']= le.fit_transform(df_encoded['Manufacturer'])
df_encoded['Model']= le.fit_transform(df_encoded['Model'])
df_encoded['Fuel type']= le.fit_transform(df_encoded['Fuel type'])
```

Linear Model

In [13]:

```
#Seperating numerical and categorical features
numerical_features = ["Engine size","Year of manufacture","Mileage"]
numerical = df_encoded [numerical_features]
categorical_features = ["Manufacturer","Model","Fuel type"]
categorical = df_encoded[categorical_features]

#Standardizing numerical features
scale = StandardScaler()
scale.fit(numerical)
T_numerical = scale.transform(numerical)

# Convert transformed numerical features back to a dataframe
T_numerical_df = pd.DataFrame(T_numerical, columns=numerical_features, index=df_encoded.index)

# Combine scaled numerical features and categorical features
# Seperating dependent and independent variables
x = pd.concat([T_numerical_df, categorical], axis=1)
y = df["Price"]

# Splitting into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
```

```python
# Applying Linear Regression
Price_linear = LinearRegression()
Price_linear.fit(x_train, y_train)

# Prediction
Price_pred = Price_linear.predict(x_test)

#Calculating error
mae = mean_absolute_error(y_test, Price_pred)
mse = mean_squared_error(y_test, Price_pred)
rmse = np.sqrt(mse)
R2 = r2_score(y_test, Price_pred)
print(mae,mse,rmse,R2)
```

```
6076.345864707946 89013685.25867541 9434.706421435456 0.6719903658783444
```

Polynomial Regression

In [14]:

```python
#Seperating numerical and categorical features
numerical_features = ["Engine size","Year of manufacture","Mileage"]
numerical = df_encoded [numerical_features]
categorical_features = ["Manufacturer","Model","Fuel type"]
categorical = df_encoded[categorical_features]

#Standardizing numerical features
scale = StandardScaler()
scale.fit(numerical)
T_numerical = scale.transform(numerical)

# Convert transformed numerical features back to a dataframe
T_numerical_df = pd.DataFrame(T_numerical, columns=numerical_features, index=df_encoded.index)

# Combine scaled numerical features and categorical features
# Seperating dependent and independent variables
x = pd.concat([T_numerical_df, categorical], axis=1)
y = df["Price"]

# Splitting into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
```

```
# Dictionary to store errors for different degrees
error_metrics = {}

# Loop to check errors for polynomial degrees from 2 to 9
for degree in range(2, 10):
    poly = PolynomialFeatures(degree=degree, include_bias=False)
    x_train_poly = poly.fit_transform(x_train)
    x_test_poly = poly.transform(x_test)

    #Fitting Model
    Price_poly = LinearRegression()
    Price_poly.fit(x_train_poly, y_train)

    #Prediction
    Price_pred = Price_poly.predict(x_test_poly)

    #Calculating error
    mae = mean_absolute_error(y_test, Price_pred)
    mse = mean_squared_error(y_test, Price_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_test, Price_pred)

    error_metrics[degree] = {'MAE': mae, 'MSE': mse, 'RMSE': rmse, 'R2': r2}
    print(f'Degree: {degree}, MAE: {mae}, MSE: {mse}, RMSE: {rmse}, R2: {r2}')
```

```
Degree: 2, MAE: 2989.4438661874756, MSE: 25326013.742123663, RMSE: 5032.495776662278, R2: 0.9066752884438702
Degree: 3, MAE: 1706.9864621812537, MSE: 9433213.403289303, RMSE: 3071.353676034283, R2: 0.9652392228451989
Degree: 4, MAE: 899.265718308596, MSE: 2975008.7550537162, RMSE: 1724.821369027447, R2: 0.9890372864529976
Degree: 5, MAE: 332.1832412037725, MSE: 482200.17559067335, RMSE: 694.4063476025211, R2: 0.9982231237510361
Degree: 6, MAE: 92.27557870086031, MSE: 48016.07426618837, RMSE: 219.1257042571418, R2: 0.9998230638928582
Degree: 7, MAE: 19.05443913142482, MSE: 2824.1275087367544, RMSE: 53.14252072245684, R2: 0.9999895932740211
Degree: 8, MAE: 2.6860089860479013, MSE: 80.26166720047691, RMSE: 8.958887609546004, R2: 0.9999997042409825
Degree: 9, MAE: 0.6059637446682433, MSE: 13.796991329839027, RMSE: 3.714430148736011, R2: 0.9999999491589854
```

Decision Tree Regressor Model

In [15]:

```
#Seperating numerical and categorical features
numerical_features = ["Engine size","Year of manufacture","Mileage"]
numerical = df_encoded [numerical_features]
categorical_features = ["Manufacturer","Model","Fuel type"]
categorical = df_encoded[categorical_features]
```

```
#Standardizing numerical features
scale = StandardScaler()
scale.fit(numerical)
T_numerical = scale.transform(numerical)

# Convert transformed numerical features back to a dataframe
T_numerical_df = pd.DataFrame(T_numerical, columns=numerical_features, index=df_encoded.index)

# Combine scaled numerical features and categorical features
# Seperating dependent and independent variables
x = pd.concat([T_numerical_df, categorical], axis=1)
y = df["Price"]

# Splitting into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

# Applying Decision Tree Regressor Model
from sklearn.tree import DecisionTreeRegressor
Price_DT = DecisionTreeRegressor(random_state=42)
Price_DT.fit(x_train, y_train)

# Prediction
Price_pred = Price_DT.predict(x_test)

#Calculating error
mae = mean_absolute_error(y_test, Price_pred)
mse = mean_squared_error(y_test, Price_pred)
rmse = np.sqrt(mse)
R2 = r2_score(y_test, Price_pred)
print(mae,mse,rmse,R2)
```

```
486.1968 1139991.6378 1067.703909237013 0.9957992050443717
```

Random Forest Regressor model

In [16]:

```
#Seperating numerical and categorical features
numerical_features = ["Engine size","Year of manufacture","Mileage"]
numerical = df_encoded [numerical_features]
categorical_features = ["Manufacturer","Model","Fuel type"]
categorical = df_encoded[categorical_features]
```

```python
#Standardizing numerical features
scale = StandardScaler()
scale.fit(numerical)
T_numerical = scale.transform(numerical)

# Convert transformed numerical features back to a dataframe
T_numerical_df = pd.DataFrame(T_numerical, columns=numerical_features, index=df_encoded.index)

# Combine scaled numerical features and categorical features
# Seperating dependent and independent variables
x = pd.concat([T_numerical_df, categorical], axis=1)
y = df["Price"]

# Splitting into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

# Applying Random Forest Regressor Model
from sklearn.ensemble import RandomForestRegressor
Price_RF = RandomForestRegressor(random_state=42)
Price_RF.fit(x_train, y_train)

# Prediction
Price_pred = Price_RF.predict(x_test)

#Calculating error
mae = mean_absolute_error(y_test, Price_pred)
mse = mean_squared_error(y_test, Price_pred)
rmse = np.sqrt(mse)
R2 = r2_score(y_test, Price_pred)
print(mae,mse,rmse,R2)
```

```
332.270401 475731.2266336499 689.7327211562823 0.9982469614067221
```

Part (d)

Develop an Artificial Neural Network (ANN) model to predict the price of a car based on all the available information from the dataset. How does its performance compare to the other supervised learning models that you have considered? Discuss your choices for the architecture of the neural network that you used, and describe how you tuned the hyperparameters in your

model to achieve the best performance.

In [145]:

```
#Seperating numerical and categorical features
numerical_features = ["Engine size","Year of manufacture","Mileage"]
numerical = df_encoded [numerical_features]
categorical_features = ["Manufacturer","Model","Fuel type"]
categorical = df_encoded[categorical_features]

#Standardizing numerical features
scale = StandardScaler()
scale.fit(numerical)
T_numerical = scale.transform(numerical)

# Convert transformed numerical features back to a dataframe
T_numerical_df = pd.DataFrame(T_numerical, columns=numerical_features, index=df_encoded.index)

# Combine scaled numerical features and categorical features
# Seperating dependent and independent variables
x = pd.concat([T_numerical_df, categorical], axis=1)
y = df["Price"]

# Splitting into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
```

Constructor Stage

In [43]:

```
from keras.models import Sequential
model = Sequential()

from keras.layers import Dense, Dropout
# Input Layer
model.add(Dense(units = 64, input_dim = (6), activation = "relu"))
# Dropout of 20%
model.add(Dropout(0.2))
# First Dense layer with 64 neurons
model.add(Dense(units = 64, activation = "relu"))
# Output layer
model.add(Dense(units = 1, activation = "linear"))
model.summary()
```

```
C:\Python312\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	448
dropout (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 64)	4,160
dense_2 (Dense)	(None, 1)	65

Total params: 4,673 (18.25 KB)

Trainable params: 4,673 (18.25 KB)

Non-trainable params: 0 (0.00 B)

Compilation Stage

In [147]:

```
model.compile(optimizer="adam", loss='mean_squared_error', metrics=['mean_squared_error'])

from keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(monitor='val_loss', patience = 20)

history = model.fit(x = x_train, y = y_train, batch_size = None, epochs = 200, verbose = "auto", validation_split = 0.1, c
allbacks = [early_stopping])
```

```
Epoch 1/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - loss: 378895296.0000 - mean_squared_error: 378895296.0000 - val_loss: 12961
3424.0000 - val_mean_squared_error: 129613424.0000
Epoch 2/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 105290880.0000 - mean_squared_error: 105290880.0000 - val_loss: 48027
508.0000 - val_mean_squared_error: 48027508.0000
Epoch 3/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 55315580.0000 - mean_squared_error: 55315580.0000 - val_loss:
```

```
37292876.0000 - val_mean_squared_error: 37292876.0000
Epoch 4/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 42053372.0000 - mean_squared_error: 42053372.0000 - val_loss:
33571216.0000 - val_mean_squared_error: 33571216.0000
Epoch 5/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 36499760.0000 - mean_squared_error: 36499760.0000 - val_loss:
31513434.0000 - val_mean_squared_error: 31513434.0000
Epoch 6/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 37168468.0000 - mean_squared_error: 37168468.0000 - val_loss:
29576058.0000 - val_mean_squared_error: 29576058.0000
Epoch 7/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 33593724.0000 - mean_squared_error: 33593724.0000 - val_loss:
27831226.0000 - val_mean_squared_error: 27831226.0000
Epoch 8/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 32735074.0000 - mean_squared_error: 32735074.0000 - val_loss:
26408984.0000 - val_mean_squared_error: 26408984.0000
Epoch 9/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 30422526.0000 - mean_squared_error: 30422526.0000 - val_loss:
25220242.0000 - val_mean_squared_error: 25220242.0000
Epoch 10/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 29379414.0000 - mean_squared_error: 29379414.0000 - val_loss:
24276382.0000 - val_mean_squared_error: 24276382.0000
Epoch 11/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 28546594.0000 - mean_squared_error: 28546594.0000 - val_loss:
23531300.0000 - val_mean_squared_error: 23531300.0000
Epoch 12/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 28367408.0000 - mean_squared_error: 28367408.0000 - val_loss:
22977040.0000 - val_mean_squared_error: 22977040.0000
Epoch 13/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 27219088.0000 - mean_squared_error: 27219088.0000 - val_loss:
22561830.0000 - val_mean_squared_error: 22561830.0000
Epoch 14/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 26157988.0000 - mean_squared_error: 26157988.0000 - val_loss:
22198502.0000 - val_mean_squared_error: 22198502.0000
Epoch 15/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 25977544.0000 - mean_squared_error: 25977544.0000 - val_loss:
21757858.0000 - val_mean_squared_error: 21757858.0000
Epoch 16/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 26356740.0000 - mean_squared_error: 26356740.0000 - val_loss:
21276700.0000 - val_mean_squared_error: 21276700.0000
Epoch 17/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 25495298.0000 - mean_squared_error: 25495298.0000 - val_loss:
20932020.0000 - val_mean_squared_error: 20932020.0000
Epoch 18/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 25312138.0000 - mean_squared_error: 25312138.0000 - val_loss:
```

```
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 25312138.0000 - mean_squared_error: 25312138.0000 - val_loss:
20586142.0000 - val_mean_squared_error: 20586142.0000
Epoch 19/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 24068952.0000 - mean_squared_error: 24068952.0000 - val_loss:
20244498.0000 - val_mean_squared_error: 20244498.0000
Epoch 20/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 25435440.0000 - mean_squared_error: 25435440.0000 - val_loss:
19906196.0000 - val_mean_squared_error: 19906196.0000
Epoch 21/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 24430522.0000 - mean_squared_error: 24430522.0000 - val_loss:
19571860.0000 - val_mean_squared_error: 19571860.0000
Epoch 22/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 23763900.0000 - mean_squared_error: 23763900.0000 - val_loss:
19322466.0000 - val_mean_squared_error: 19322466.0000
Epoch 23/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 23930098.0000 - mean_squared_error: 23930098.0000 - val_loss:
19006682.0000 - val_mean_squared_error: 19006682.0000
Epoch 24/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 23031364.0000 - mean_squared_error: 23031364.0000 - val_loss:
18757978.0000 - val_mean_squared_error: 18757978.0000
Epoch 25/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 22649536.0000 - mean_squared_error: 22649536.0000 - val_loss:
18414646.0000 - val_mean_squared_error: 18414646.0000
Epoch 26/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 22212704.0000 - mean_squared_error: 22212704.0000 - val_loss:
18271062.0000 - val_mean_squared_error: 18271062.0000
Epoch 27/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 23196296.0000 - mean_squared_error: 23196296.0000 - val_loss:
17872970.0000 - val_mean_squared_error: 17872970.0000
Epoch 28/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 22146766.0000 - mean_squared_error: 22146766.0000 - val_loss:
17640438.0000 - val_mean_squared_error: 17640438.0000
Epoch 29/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 20389390.0000 - mean_squared_error: 20389390.0000 - val_loss:
17383518.0000 - val_mean_squared_error: 17383518.0000
Epoch 30/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 21483050.0000 - mean_squared_error: 21483050.0000 - val_loss:
17161054.0000 - val_mean_squared_error: 17161054.0000
Epoch 31/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 21456716.0000 - mean_squared_error: 21456716.0000 - val_loss:
16900508.0000 - val_mean_squared_error: 16900508.0000
Epoch 32/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 20778748.0000 - mean_squared_error: 20778748.0000 - val_loss:
16711114.0000 - val_mean_squared_error: 16711114.0000
Epoch 33/200
```

```
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 20801162.0000 - mean_squared_error: 20801162.0000 - val_loss:
16477082.0000 - val_mean_squared_error: 16477082.0000
Epoch 34/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 19962808.0000 - mean_squared_error: 19962808.0000 - val_loss:
16270247.0000 - val_mean_squared_error: 16270247.0000
Epoch 35/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - loss: 20378512.0000 - mean_squared_error: 20378512.0000 - val_loss:
16095829.0000 - val_mean_squared_error: 16095829.0000
Epoch 36/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - loss: 21142626.0000 - mean_squared_error: 21142626.0000 - val_loss:
15918395.0000 - val_mean_squared_error: 15918395.0000
Epoch 37/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 18941614.0000 - mean_squared_error: 18941614.0000 - val_loss:
15727140.0000 - val_mean_squared_error: 15727140.0000
Epoch 38/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 20200640.0000 - mean_squared_error: 20200640.0000 - val_loss:
15632103.0000 - val_mean_squared_error: 15632103.0000
Epoch 39/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 19899786.0000 - mean_squared_error: 19899786.0000 - val_loss:
15454259.0000 - val_mean_squared_error: 15454259.0000
Epoch 40/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 19323584.0000 - mean_squared_error: 19323584.0000 - val_loss:
15198816.0000 - val_mean_squared_error: 15198816.0000
Epoch 41/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 19026532.0000 - mean_squared_error: 19026532.0000 - val_loss:
15038705.0000 - val_mean_squared_error: 15038705.0000
Epoch 42/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 18914724.0000 - mean_squared_error: 18914724.0000 - val_loss:
14912343.0000 - val_mean_squared_error: 14912343.0000
Epoch 43/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 17818742.0000 - mean_squared_error: 17818742.0000 - val_loss:
14732544.0000 - val_mean_squared_error: 14732544.0000
Epoch 44/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 18710880.0000 - mean_squared_error: 18710880.0000 - val_loss:
14642329.0000 - val_mean_squared_error: 14642329.0000
Epoch 45/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 18550570.0000 - mean_squared_error: 18550570.0000 - val_loss:
14521997.0000 - val_mean_squared_error: 14521997.0000
Epoch 46/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 17846574.0000 - mean_squared_error: 17846574.0000 - val_loss:
14309961.0000 - val_mean_squared_error: 14309961.0000
Epoch 47/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 18173456.0000 - mean_squared_error: 18173456.0000 - val_loss:
14250495.0000 - val_mean_squared_error: 14250495.0000
Epoch 48/200
```

```
Epoch 48/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 18646886.0000 - mean_squared_error: 18646886.0000 - val_loss:
14132212.0000 - val_mean_squared_error: 14132212.0000
Epoch 49/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 18312002.0000 - mean_squared_error: 18312002.0000 - val_loss:
14022414.0000 - val_mean_squared_error: 14022414.0000
Epoch 50/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 17297838.0000 - mean_squared_error: 17297838.0000 - val_loss:
13947327.0000 - val_mean_squared_error: 13947327.0000
Epoch 51/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 17259256.0000 - mean_squared_error: 17259256.0000 - val_loss:
13818722.0000 - val_mean_squared_error: 13818722.0000
Epoch 52/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 17379784.0000 - mean_squared_error: 17379784.0000 - val_loss:
13738175.0000 - val_mean_squared_error: 13738175.0000
Epoch 53/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 17400128.0000 - mean_squared_error: 17400128.0000 - val_loss:
13652334.0000 - val_mean_squared_error: 13652334.0000
Epoch 54/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 17234172.0000 - mean_squared_error: 17234172.0000 - val_loss:
13426674.0000 - val_mean_squared_error: 13426674.0000
Epoch 55/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 16845758.0000 - mean_squared_error: 16845758.0000 - val_loss:
13373503.0000 - val_mean_squared_error: 13373503.0000
Epoch 56/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 16748254.0000 - mean_squared_error: 16748254.0000 - val_loss:
13267866.0000 - val_mean_squared_error: 13267866.0000
Epoch 57/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 18278786.0000 - mean_squared_error: 18278786.0000 - val_loss:
13186918.0000 - val_mean_squared_error: 13186918.0000
Epoch 58/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 16448729.0000 - mean_squared_error: 16448729.0000 - val_loss:
13141576.0000 - val_mean_squared_error: 13141576.0000
Epoch 59/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 17618412.0000 - mean_squared_error: 17618412.0000 - val_loss:
13031518.0000 - val_mean_squared_error: 13031518.0000
Epoch 60/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 16377126.0000 - mean_squared_error: 16377126.0000 - val_loss:
13010991.0000 - val_mean_squared_error: 13010991.0000
Epoch 61/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 16882588.0000 - mean_squared_error: 16882588.0000 - val_loss:
12913496.0000 - val_mean_squared_error: 12913496.0000
Epoch 62/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 16117073.0000 - mean_squared_error: 16117073.0000 - val_loss:
12711369.0000 - val_mean_squared_error: 12711369.0000
```

```
Epoch 63/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 15806957.0000 - mean_squared_error: 15806957.0000 - val_loss: 12764417.0000 - val_mean_squared_error: 12764417.0000
Epoch 64/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 15942357.0000 - mean_squared_error: 15942357.0000 - val_loss: 12576039.0000 - val_mean_squared_error: 12576039.0000
Epoch 65/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - loss: 16523815.0000 - mean_squared_error: 16523815.0000 - val_loss: 12630679.0000 - val_mean_squared_error: 12630679.0000
Epoch 66/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 16513278.0000 - mean_squared_error: 16513278.0000 - val_loss: 12826533.0000 - val_mean_squared_error: 12826533.0000
Epoch 67/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 15833097.0000 - mean_squared_error: 15833097.0000 - val_loss: 12498541.0000 - val_mean_squared_error: 12498541.0000
Epoch 68/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 16472651.0000 - mean_squared_error: 16472651.0000 - val_loss: 12235050.0000 - val_mean_squared_error: 12235050.0000
Epoch 69/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 16723111.0000 - mean_squared_error: 16723111.0000 - val_loss: 12285715.0000 - val_mean_squared_error: 12285715.0000
Epoch 70/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 16413818.0000 - mean_squared_error: 16413818.0000 - val_loss: 12050882.0000 - val_mean_squared_error: 12050882.0000
Epoch 71/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 15713731.0000 - mean_squared_error: 15713731.0000 - val_loss: 11977798.0000 - val_mean_squared_error: 11977798.0000
Epoch 72/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 16465308.0000 - mean_squared_error: 16465308.0000 - val_loss: 11919193.0000 - val_mean_squared_error: 11919193.0000
Epoch 73/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 15611903.0000 - mean_squared_error: 15611903.0000 - val_loss: 11789452.0000 - val_mean_squared_error: 11789452.0000
Epoch 74/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 16789574.0000 - mean_squared_error: 16789574.0000 - val_loss: 11924704.0000 - val_mean_squared_error: 11924704.0000
Epoch 75/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 16108772.0000 - mean_squared_error: 16108772.0000 - val_loss: 11667996.0000 - val_mean_squared_error: 11667996.0000
Epoch 76/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 15653172.0000 - mean_squared_error: 15653172.0000 - val_loss: 11577510.0000 - val_mean_squared_error: 11577510.0000
Epoch 77/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 15882510.0000 - mean_squared_error: 15882510.0000 - val_loss: 11559448.0000 - val_mean_squared_error: 11559448.0000
```

```
Epoch 78/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 15144196.0000 - mean_squared_error: 15144196.0000 - val_loss: 11524817.0000 - val_mean_squared_error: 11524817.0000
Epoch 79/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 15414196.0000 - mean_squared_error: 15414196.0000 - val_loss: 11452253.0000 - val_mean_squared_error: 11452253.0000
Epoch 80/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 16004956.0000 - mean_squared_error: 16004956.0000 - val_loss: 11374822.0000 - val_mean_squared_error: 11374822.0000
Epoch 81/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 15485283.0000 - mean_squared_error: 15485283.0000 - val_loss: 11302598.0000 - val_mean_squared_error: 11302598.0000
Epoch 82/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 16417457.0000 - mean_squared_error: 16417457.0000 - val_loss: 11323660.0000 - val_mean_squared_error: 11323660.0000
Epoch 83/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 15204822.0000 - mean_squared_error: 15204822.0000 - val_loss: 11184668.0000 - val_mean_squared_error: 11184668.0000
Epoch 84/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 15406834.0000 - mean_squared_error: 15406834.0000 - val_loss: 11135073.0000 - val_mean_squared_error: 11135073.0000
Epoch 85/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 15904915.0000 - mean_squared_error: 15904915.0000 - val_loss: 11033255.0000 - val_mean_squared_error: 11033255.0000
Epoch 86/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 15690254.0000 - mean_squared_error: 15690254.0000 - val_loss: 11095066.0000 - val_mean_squared_error: 11095066.0000
Epoch 87/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 14656690.0000 - mean_squared_error: 14656690.0000 - val_loss: 11077103.0000 - val_mean_squared_error: 11077103.0000
Epoch 88/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 14362342.0000 - mean_squared_error: 14362342.0000 - val_loss: 11073340.0000 - val_mean_squared_error: 11073340.0000
Epoch 89/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 16039119.0000 - mean_squared_error: 16039119.0000 - val_loss: 10942331.0000 - val_mean_squared_error: 10942331.0000
Epoch 90/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 14643550.0000 - mean_squared_error: 14643550.0000 - val_loss: 10962202.0000 - val_mean_squared_error: 10962202.0000
Epoch 91/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 14346751.0000 - mean_squared_error: 14346751.0000 - val_loss: 10767043.0000 - val_mean_squared_error: 10767043.0000
Epoch 92/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 14223226.0000 - mean_squared_error: 14223226.0000 - val_loss:
```

```
10745873.0000 - val_mean_squared_error: 10745873.0000
Epoch 93/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 14712922.0000 - mean_squared_error: 14712922.0000 - val_loss:
10767493.0000 - val_mean_squared_error: 10767493.0000
Epoch 94/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 14404439.0000 - mean_squared_error: 14404439.0000 - val_loss:
11086861.0000 - val_mean_squared_error: 11086861.0000
Epoch 95/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 14432423.0000 - mean_squared_error: 14432423.0000 - val_loss:
10567673.0000 - val_mean_squared_error: 10567673.0000
Epoch 96/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 15850610.0000 - mean_squared_error: 15850610.0000 - val_loss:
10567076.0000 - val_mean_squared_error: 10567076.0000
Epoch 97/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 14792662.0000 - mean_squared_error: 14792662.0000 - val_loss:
10486125.0000 - val_mean_squared_error: 10486125.0000
Epoch 98/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 14480800.0000 - mean_squared_error: 14480800.0000 - val_loss:
10488737.0000 - val_mean_squared_error: 10488737.0000
Epoch 99/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 13475851.0000 - mean_squared_error: 13475851.0000 - val_loss:
10318141.0000 - val_mean_squared_error: 10318141.0000
Epoch 100/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 15500211.0000 - mean_squared_error: 15500211.0000 - val_loss:
10426721.0000 - val_mean_squared_error: 10426721.0000
Epoch 101/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 13342592.0000 - mean_squared_error: 13342592.0000 - val_loss:
10342835.0000 - val_mean_squared_error: 10342835.0000
Epoch 102/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - loss: 14887703.0000 - mean_squared_error: 14887703.0000 - val_loss:
10317464.0000 - val_mean_squared_error: 10317464.0000
Epoch 103/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 13999754.0000 - mean_squared_error: 13999754.0000 - val_loss:
10276030.0000 - val_mean_squared_error: 10276030.0000
Epoch 104/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 14165111.0000 - mean_squared_error: 14165111.0000 - val_loss:
10237182.0000 - val_mean_squared_error: 10237182.0000
Epoch 105/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 13963613.0000 - mean_squared_error: 13963613.0000 - val_loss:
10123234.0000 - val_mean_squared_error: 10123234.0000
Epoch 106/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - loss: 14249213.0000 - mean_squared_error: 14249213.0000 - val_loss:
10090098.0000 - val_mean_squared_error: 10090098.0000
Epoch 107/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 15272936.0000 - mean_squared_error: 15272936.0000 - val_loss:
```

```
10035314.0000 - val_mean_squared_error: 10035314.0000
Epoch 108/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 12930546.0000 - mean_squared_error: 12930546.0000 - val_loss:
9880137.0000 - val_mean_squared_error: 9880137.0000
Epoch 109/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 13920505.0000 - mean_squared_error: 13920505.0000 - val_loss:
9875526.0000 - val_mean_squared_error: 9875526.0000
Epoch 110/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 14348366.0000 - mean_squared_error: 14348366.0000 - val_loss:
9814615.0000 - val_mean_squared_error: 9814615.0000
Epoch 111/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 14601977.0000 - mean_squared_error: 14601977.0000 - val_loss:
9812013.0000 - val_mean_squared_error: 9812013.0000
Epoch 112/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 13395628.0000 - mean_squared_error: 13395628.0000 - val_loss:
9891115.0000 - val_mean_squared_error: 9891115.0000
Epoch 113/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 13934667.0000 - mean_squared_error: 13934667.0000 - val_loss:
9746708.0000 - val_mean_squared_error: 9746708.0000
Epoch 114/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 13909221.0000 - mean_squared_error: 13909221.0000 - val_loss:
9660739.0000 - val_mean_squared_error: 9660739.0000
Epoch 115/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 13609021.0000 - mean_squared_error: 13609021.0000 - val_loss:
9732660.0000 - val_mean_squared_error: 9732660.0000
Epoch 116/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 13980483.0000 - mean_squared_error: 13980483.0000 - val_loss:
9701354.0000 - val_mean_squared_error: 9701354.0000
Epoch 117/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 14102671.0000 - mean_squared_error: 14102671.0000 - val_loss:
9852305.0000 - val_mean_squared_error: 9852305.0000
Epoch 118/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 13122481.0000 - mean_squared_error: 13122481.0000 - val_loss:
9475129.0000 - val_mean_squared_error: 9475129.0000
Epoch 119/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 13107684.0000 - mean_squared_error: 13107684.0000 - val_loss:
9621198.0000 - val_mean_squared_error: 9621198.0000
Epoch 120/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 13021440.0000 - mean_squared_error: 13021440.0000 - val_loss:
9631452.0000 - val_mean_squared_error: 9631452.0000
Epoch 121/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 13398034.0000 - mean_squared_error: 13398034.0000 - val_loss:
9481927.0000 - val_mean_squared_error: 9481927.0000
Epoch 122/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 14467775.0000 - mean_squared_error: 14467775.0000 - val_loss:
```

```
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 14407775.0000 - mean_squared_error: 14407775.0000 - val_loss:
9307798.0000 - val_mean_squared_error: 9307798.0000
Epoch 123/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 12647826.0000 - mean_squared_error: 12647826.0000 - val_loss:
9344118.0000 - val_mean_squared_error: 9344118.0000
Epoch 124/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 13026524.0000 - mean_squared_error: 13026524.0000 - val_loss:
9508804.0000 - val_mean_squared_error: 9508804.0000
Epoch 125/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 13468590.0000 - mean_squared_error: 13468590.0000 - val_loss:
9240776.0000 - val_mean_squared_error: 9240776.0000
Epoch 126/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 13267729.0000 - mean_squared_error: 13267729.0000 - val_loss:
9275700.0000 - val_mean_squared_error: 9275700.0000
Epoch 127/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 12838248.0000 - mean_squared_error: 12838248.0000 - val_loss:
9183686.0000 - val_mean_squared_error: 9183686.0000
Epoch 128/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 12689611.0000 - mean_squared_error: 12689611.0000 - val_loss:
9102109.0000 - val_mean_squared_error: 9102109.0000
Epoch 129/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 12919139.0000 - mean_squared_error: 12919139.0000 - val_loss:
9155464.0000 - val_mean_squared_error: 9155464.0000
Epoch 130/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 13197884.0000 - mean_squared_error: 13197884.0000 - val_loss:
9156728.0000 - val_mean_squared_error: 9156728.0000
Epoch 131/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 13142161.0000 - mean_squared_error: 13142161.0000 - val_loss:
9027016.0000 - val_mean_squared_error: 9027016.0000
Epoch 132/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 12080051.0000 - mean_squared_error: 12080051.0000 - val_loss:
8987503.0000 - val_mean_squared_error: 8987503.0000
Epoch 133/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 13194030.0000 - mean_squared_error: 13194030.0000 - val_loss:
9091035.0000 - val_mean_squared_error: 9091035.0000
Epoch 134/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 12454509.0000 - mean_squared_error: 12454509.0000 - val_loss:
9127282.0000 - val_mean_squared_error: 9127282.0000
Epoch 135/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 12716765.0000 - mean_squared_error: 12716765.0000 - val_loss:
8851858.0000 - val_mean_squared_error: 8851858.0000
Epoch 136/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 12174280.0000 - mean_squared_error: 12174280.0000 - val_loss:
8869642.0000 - val_mean_squared_error: 8869642.0000
Epoch 137/200
```

```
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 12767742.0000 - mean_squared_error: 12767742.0000 - val_loss:
8861542.0000 - val_mean_squared_error: 8861542.0000
Epoch 138/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 12287494.0000 - mean_squared_error: 12287494.0000 - val_loss:
8793091.0000 - val_mean_squared_error: 8793091.0000
Epoch 139/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 12911429.0000 - mean_squared_error: 12911429.0000 - val_loss:
8755130.0000 - val_mean_squared_error: 8755130.0000
Epoch 140/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 12163009.0000 - mean_squared_error: 12163009.0000 - val_loss:
8705048.0000 - val_mean_squared_error: 8705048.0000
Epoch 141/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 13132169.0000 - mean_squared_error: 13132169.0000 - val_loss:
8710859.0000 - val_mean_squared_error: 8710859.0000
Epoch 142/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 12066699.0000 - mean_squared_error: 12066699.0000 - val_loss:
8727563.0000 - val_mean_squared_error: 8727563.0000
Epoch 143/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11847936.0000 - mean_squared_error: 11847936.0000 - val_loss:
8579244.0000 - val_mean_squared_error: 8579244.0000
Epoch 144/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 11683853.0000 - mean_squared_error: 11683853.0000 - val_loss:
8536422.0000 - val_mean_squared_error: 8536422.0000
Epoch 145/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11584072.0000 - mean_squared_error: 11584072.0000 - val_loss:
8531380.0000 - val_mean_squared_error: 8531380.0000
Epoch 146/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 12354460.0000 - mean_squared_error: 12354460.0000 - val_loss:
8494659.0000 - val_mean_squared_error: 8494659.0000
Epoch 147/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11984417.0000 - mean_squared_error: 11984417.0000 - val_loss:
8499909.0000 - val_mean_squared_error: 8499909.0000
Epoch 148/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 12076731.0000 - mean_squared_error: 12076731.0000 - val_loss:
8547111.0000 - val_mean_squared_error: 8547111.0000
Epoch 149/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 11818714.0000 - mean_squared_error: 11818714.0000 - val_loss:
8451477.0000 - val_mean_squared_error: 8451477.0000
Epoch 150/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 11381672.0000 - mean_squared_error: 11381672.0000 - val_loss:
8416459.0000 - val_mean_squared_error: 8416459.0000
Epoch 151/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11574459.0000 - mean_squared_error: 11574459.0000 - val_loss:
8364302.5000 - val_mean_squared_error: 8364302.5000
Epoch 152/200
```

```
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11939306.0000 - mean_squared_error: 11939306.0000 - val_loss: 8326606.0000 - val_mean_squared_error: 8326606.0000
Epoch 153/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 12005495.0000 - mean_squared_error: 12005495.0000 - val_loss: 8406899.0000 - val_mean_squared_error: 8406899.0000
Epoch 154/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11644770.0000 - mean_squared_error: 11644770.0000 - val_loss: 8207045.5000 - val_mean_squared_error: 8207045.5000
Epoch 155/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 12034070.0000 - mean_squared_error: 12034070.0000 - val_loss: 8303274.0000 - val_mean_squared_error: 8303274.0000
Epoch 156/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 12056707.0000 - mean_squared_error: 12056707.0000 - val_loss: 8197144.5000 - val_mean_squared_error: 8197144.5000
Epoch 157/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11833318.0000 - mean_squared_error: 11833318.0000 - val_loss: 8209888.5000 - val_mean_squared_error: 8209888.5000
Epoch 158/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 11776071.0000 - mean_squared_error: 11776071.0000 - val_loss: 8141954.5000 - val_mean_squared_error: 8141954.5000
Epoch 159/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11629622.0000 - mean_squared_error: 11629622.0000 - val_loss: 8188189.0000 - val_mean_squared_error: 8188189.0000
Epoch 160/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11348456.0000 - mean_squared_error: 11348456.0000 - val_loss: 8180851.5000 - val_mean_squared_error: 8180851.5000
Epoch 161/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11542731.0000 - mean_squared_error: 11542731.0000 - val_loss: 8212714.0000 - val_mean_squared_error: 8212714.0000
Epoch 162/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 11571930.0000 - mean_squared_error: 11571930.0000 - val_loss: 8088850.0000 - val_mean_squared_error: 8088850.0000
Epoch 163/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 12189130.0000 - mean_squared_error: 12189130.0000 - val_loss: 8145771.5000 - val_mean_squared_error: 8145771.5000
Epoch 164/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11917451.0000 - mean_squared_error: 11917451.0000 - val_loss: 8144653.0000 - val_mean_squared_error: 8144653.0000
Epoch 165/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11229196.0000 - mean_squared_error: 11229196.0000 - val_loss: 8049368.0000 - val_mean_squared_error: 8049368.0000
Epoch 166/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 10882743.0000 - mean_squared_error: 10882743.0000 - val_loss: 8063255.0000 - val_mean_squared_error: 8063255.0000
```

```
Epoch 167/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11396616.0000 - mean_squared_error: 11396616.0000 - val_loss:
8025964.0000 - val_mean_squared_error: 8025964.0000
Epoch 168/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11809816.0000 - mean_squared_error: 11809816.0000 - val_loss:
7984471.0000 - val_mean_squared_error: 7984471.0000
Epoch 169/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 10897444.0000 - mean_squared_error: 10897444.0000 - val_loss:
7981669.0000 - val_mean_squared_error: 7981669.0000
Epoch 170/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 12123254.0000 - mean_squared_error: 12123254.0000 - val_loss:
8005788.0000 - val_mean_squared_error: 8005788.0000
Epoch 171/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 10951219.0000 - mean_squared_error: 10951219.0000 - val_loss:
8002266.0000 - val_mean_squared_error: 8002266.0000
Epoch 172/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11336783.0000 - mean_squared_error: 11336783.0000 - val_loss:
7926403.5000 - val_mean_squared_error: 7926403.5000
Epoch 173/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11028846.0000 - mean_squared_error: 11028846.0000 - val_loss:
7877289.5000 - val_mean_squared_error: 7877289.5000
Epoch 174/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11144858.0000 - mean_squared_error: 11144858.0000 - val_loss:
7880564.0000 - val_mean_squared_error: 7880564.0000
Epoch 175/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11086166.0000 - mean_squared_error: 11086166.0000 - val_loss:
7934778.0000 - val_mean_squared_error: 7934778.0000
Epoch 176/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 10908454.0000 - mean_squared_error: 10908454.0000 - val_loss:
7874781.5000 - val_mean_squared_error: 7874781.5000
Epoch 177/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11456641.0000 - mean_squared_error: 11456641.0000 - val_loss:
7955118.0000 - val_mean_squared_error: 7955118.0000
Epoch 178/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11625339.0000 - mean_squared_error: 11625339.0000 - val_loss:
7830333.5000 - val_mean_squared_error: 7830333.5000
Epoch 179/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 10876964.0000 - mean_squared_error: 10876964.0000 - val_loss:
7847815.0000 - val_mean_squared_error: 7847815.0000
Epoch 180/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11261592.0000 - mean_squared_error: 11261592.0000 - val_loss:
7940408.5000 - val_mean_squared_error: 7940408.5000
Epoch 181/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11156824.0000 - mean_squared_error: 11156824.0000 - val_loss:
7967353.5000 - val_mean_squared_error: 7967353.5000
```

```
Epoch 182/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11033865.0000 - mean_squared_error: 11033865.0000 - val_loss: 7784658.5000 - val_mean_squared_error: 7784658.5000
Epoch 183/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 10607882.0000 - mean_squared_error: 10607882.0000 - val_loss: 7786443.5000 - val_mean_squared_error: 7786443.5000
Epoch 184/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11112293.0000 - mean_squared_error: 11112293.0000 - val_loss: 7825180.5000 - val_mean_squared_error: 7825180.5000
Epoch 185/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 10731379.0000 - mean_squared_error: 10731379.0000 - val_loss: 7694101.0000 - val_mean_squared_error: 7694101.0000
Epoch 186/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11394338.0000 - mean_squared_error: 11394338.0000 - val_loss: 7799073.0000 - val_mean_squared_error: 7799073.0000
Epoch 187/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11073818.0000 - mean_squared_error: 11073818.0000 - val_loss: 7760615.5000 - val_mean_squared_error: 7760615.5000
Epoch 188/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 10669009.0000 - mean_squared_error: 10669009.0000 - val_loss: 7749135.5000 - val_mean_squared_error: 7749135.5000
Epoch 189/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 10393162.0000 - mean_squared_error: 10393162.0000 - val_loss: 7768706.0000 - val_mean_squared_error: 7768706.0000
Epoch 190/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 11075825.0000 - mean_squared_error: 11075825.0000 - val_loss: 7714211.0000 - val_mean_squared_error: 7714211.0000
Epoch 191/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11391660.0000 - mean_squared_error: 11391660.0000 - val_loss: 7667861.0000 - val_mean_squared_error: 7667861.0000
Epoch 192/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 10613619.0000 - mean_squared_error: 10613619.0000 - val_loss: 7536234.0000 - val_mean_squared_error: 7536234.0000
Epoch 193/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 10827295.0000 - mean_squared_error: 10827295.0000 - val_loss: 7762051.5000 - val_mean_squared_error: 7762051.5000
Epoch 194/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11357185.0000 - mean_squared_error: 11357185.0000 - val_loss: 7539802.0000 - val_mean_squared_error: 7539802.0000
Epoch 195/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 10728633.0000 - mean_squared_error: 10728633.0000 - val_loss: 7545015.0000 - val_mean_squared_error: 7545015.0000
Epoch 196/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 10793211.0000 - mean_squared_error: 10793211.0000 - val_loss: 7508379.0000 - val_mean_squared_error: 7508379.0000
```

```
7508379.0000 - val_mean_squared_error: 7508379.0000
Epoch 197/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 11314533.0000 - mean_squared_error: 11314533.0000 - val_loss: 7493165.0000 - val_mean_squared_error: 7493165.0000
Epoch 198/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 10362986.0000 - mean_squared_error: 10362986.0000 - val_loss: 7482450.5000 - val_mean_squared_error: 7482450.5000
Epoch 199/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 10832178.0000 - mean_squared_error: 10832178.0000 - val_loss: 7323806.0000 - val_mean_squared_error: 7323806.0000
Epoch 200/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 10094064.0000 - mean_squared_error: 10094064.0000 - val_loss: 7364849.5000 - val_mean_squared_error: 7364849.5000
```

In [148]:

```
#Prediction
y_pred = model.predict(x_test)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
```

In [149]:

```
#Calculating error
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
R2 = r2_score(y_test, y_pred)
print(mae,mse,rmse,R2)
```

```
1460.8939770690918 7557536.63523477 2749.097421925016 0.972150981426239
```

HyperParameter Turning using Random Search

In [152]:

```
import keras_tuner as kt
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import Adam
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from keras.callbacks import EarlyStopping

#Seperating numerical and categorical features
numerical_features = ["Engine size","Year of manufacture","Mileage"]
numerical = df_encoded [numerical_features]
categorical_features = ["Manufacturer","Model","Fuel type"]
categorical = df_encoded[categorical_features]

#Standardizing numerical features
scale = StandardScaler()
scale.fit(numerical)
T_numerical = scale.transform(numerical)

# Convert transformed numerical features back to a dataframe
T_numerical_df = pd.DataFrame(T_numerical, columns=numerical_features, index=df_encoded.index)

# Combine scaled numerical features and categorical features
# Seperating dependent and independent variables
x = pd.concat([T_numerical_df, categorical], axis=1)
y = df["Price"]

# Splitting into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

# Define the function to build the model
def build_model(hp):
    model = Sequential()

    # Adding hidden layers
    for i in range(hp.Int('num_layers', 1, 3)):  # Number of hidden layers
        model.add(Dense(
            units=hp.Int(f'units_{i}', min_value=32, max_value=128, step=32),  # Number of neurons
            activation="relu",  # Activation function
            input_dim= (6)
        ))
        model.add(Dropout(hp.Choice("dropout",[0.1, 0.2, 0.3])))  # Dropout rate

    # Output layer
    model.add(Dense(units=1, activation='linear'))

    # Compile the model
```

```python
    model.compile(
        optimizer=Adam(learning_rate=hp.Choice('learning_rate', [1e-2, 1e-3])),  # Learning rate
        loss='mean_squared_error',
        metrics=['mean_squared_error']
    )
    return model

# Create a Keras Tuner RandomSearch instance
tuner = kt.RandomSearch(
    build_model,
    objective='val_mean_squared_error',
    max_trials=100,  # Number of different combinations to try
    executions_per_trial= 1,
    directory='dir_new',
    project_name='hyperparameter_tuning_01'
)

# Perform the hyperparameter search
tuner.search(x_train, y_train, epochs=50, batch_size=None, validation_split=0.1, callbacks=[EarlyStopping(monitor='val_lo
ss', patience=20)])
```

```
Trial 100 Complete [00h 01m 35s]
val_mean_squared_error: 28533948.0

Best val_mean_squared_error So Far: 1040276.625
Total elapsed time: 02h 43m 52s
```

In [153]:

```python
tuner.results_summary()
```

```
Results summary
Results in dir_new\hyperparameter_tuning_01
Showing 10 best trials
Objective(name="val_mean_squared_error", direction="min")

Trial 083 summary
Hyperparameters:
num_layers: 2
units_0: 128
dropout: 0.1
learning_rate: 0.01
units_1: 128
units_2: 128
Score: 1040276.625
```

```
Trial 045 summary
Hyperparameters:
num_layers: 3
units_0: 128
dropout: 0.1
learning_rate: 0.01
units_1: 96
units_2: 64
Score: 1439859.0

Trial 059 summary
Hyperparameters:
num_layers: 2
units_0: 128
dropout: 0.1
learning_rate: 0.01
units_1: 128
units_2: 96
Score: 1445542.0

Trial 060 summary
Hyperparameters:
num_layers: 3
units_0: 128
dropout: 0.1
learning_rate: 0.01
units_1: 64
units_2: 96
Score: 1485919.0

Trial 054 summary
Hyperparameters:
num_layers: 3
units_0: 128
dropout: 0.1
learning_rate: 0.01
units_1: 128
units_2: 96
Score: 1487169.0

Trial 014 summary
Hyperparameters:
num_layers: 3
units_0: 128
```

```
dropout: 0.1
learning_rate: 0.01
units_1: 96
units_2: 32
Score: 1712630.625

Trial 068 summary
Hyperparameters:
num_layers: 2
units_0: 96
dropout: 0.1
learning_rate: 0.01
units_1: 64
units_2: 128
Score: 1826229.0

Trial 053 summary
Hyperparameters:
num_layers: 3
units_0: 128
dropout: 0.1
learning_rate: 0.01
units_1: 64
units_2: 32
Score: 2030829.25

Trial 013 summary
Hyperparameters:
num_layers: 2
units_0: 128
dropout: 0.2
learning_rate: 0.01
units_1: 64
units_2: 96
Score: 2407031.0

Trial 057 summary
Hyperparameters:
num_layers: 3
units_0: 96
dropout: 0.1
learning_rate: 0.01
units_1: 64
units_2: 96
```

Score: 2591306.25

Parameters Obtained from Random Search

In [156]:

```
#Seperating numerical and categorical features
numerical_features = ["Engine size","Year of manufacture","Mileage"]
numerical = df_encoded [numerical_features]
categorical_features = ["Manufacturer","Model","Fuel type"]
categorical = df_encoded[categorical_features]

#Standardizing numerical features
scale = StandardScaler()
scale.fit(numerical)
T_numerical = scale.transform(numerical)

# Convert transformed numerical features back to a dataframe
T_numerical_df = pd.DataFrame(T_numerical, columns=numerical_features, index=df_encoded.index)

# Combine scaled numerical features and categorical features
# Seperating dependent and independent variables
x = pd.concat([T_numerical_df, categorical], axis=1)
y = df["Price"]

# Splitting into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

from keras.models import Sequential
model = Sequential()

from keras.layers import Dense, Dropout
model.add(Dense(units = 128, input_dim = (6), activation = "relu"))
model.add(Dropout(0.1))
model.add(Dense(units = 128, activation = "relu"))
model.add(Dense(units = 64, activation = "relu"))
model.add(Dense(units = 1, activation = "linear"))
model.summary()

adam_optimizer = Adam(learning_rate = 0.01)
model.compile(optimizer=adam_optimizer, loss='mean_squared_error', metrics=['mean_squared_error'])

from keras.callbacks import EarlyStopping
```

```
early_stopping = EarlyStopping(monitor='val_loss', patience = 20)

history = model.fit(x = x_train, y = y_train, batch_size = None, epochs = 200, verbose = "auto", validation_split = 0.1, c
allbacks = [early_stopping])

#Prediction
y_pred = model.predict(x_test)

#Calculating error
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
R2 = r2_score(y_test, y_pred)
print(mae,mse,rmse,R2)
```

```
C:\Python312\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
dense_10 (Dense)	(None, 128)	896
dropout_3 (Dropout)	(None, 128)	0
dense_11 (Dense)	(None, 128)	16,512
dense_12 (Dense)	(None, 64)	8,256
dense_13 (Dense)	(None, 1)	65

Total params: 25,729 (100.50 KB)

Trainable params: 25,729 (100.50 KB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 89450456.0000 - mean_squared_error: 89450456.0000 - val_loss:
15215399.0000 - val_mean_squared_error: 15215399.0000
Epoch 2/200
```

```
Epoch 2/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 17984438.0000 - mean_squared_error: 17984438.0000 - val_loss: 22306578.0000 - val_mean_squared_error: 22306578.0000
Epoch 3/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 14626770.0000 - mean_squared_error: 14626770.0000 - val_loss: 12544962.0000 - val_mean_squared_error: 12544962.0000
Epoch 4/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 14253807.0000 - mean_squared_error: 14253807.0000 - val_loss: 7154402.5000 - val_mean_squared_error: 7154402.5000
Epoch 5/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 9353363.0000 - mean_squared_error: 9353363.0000 - val_loss: 4443579.5000 - val_mean_squared_error: 4443579.5000
Epoch 6/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 7744595.0000 - mean_squared_error: 7744595.0000 - val_loss: 3523892.5000 - val_mean_squared_error: 3523892.5000
Epoch 7/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 5925279.5000 - mean_squared_error: 5925279.5000 - val_loss: 3870617.2500 - val_mean_squared_error: 3870617.2500
Epoch 8/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 5772067.5000 - mean_squared_error: 5772067.5000 - val_loss: 4706299.5000 - val_mean_squared_error: 4706299.5000
Epoch 9/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 5624439.0000 - mean_squared_error: 5624439.0000 - val_loss: 2549421.7500 - val_mean_squared_error: 2549421.7500
Epoch 10/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 4111541.2500 - mean_squared_error: 4111541.2500 - val_loss: 3088496.2500 - val_mean_squared_error: 3088496.2500
Epoch 11/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 4050307.0000 - mean_squared_error: 4050307.0000 - val_loss: 2371021.5000 - val_mean_squared_error: 2371021.5000
Epoch 12/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 3552041.7500 - mean_squared_error: 3552041.7500 - val_loss: 2641693.7500 - val_mean_squared_error: 2641693.7500
Epoch 13/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 4284135.5000 - mean_squared_error: 4284135.5000 - val_loss: 1227347.8750 - val_mean_squared_error: 1227347.8750
Epoch 14/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 3129267.2500 - mean_squared_error: 3129267.2500 - val_loss: 1502659.2500 - val_mean_squared_error: 1502659.2500
Epoch 15/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 3343840.0000 - mean_squared_error: 3343840.0000 - val_loss: 5103408.0000 - val_mean_squared_error: 5103408.0000
Epoch 16/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 3152800.2500 - mean_squared_error: 3152800.2500 - val_loss: 1796231.1250 - val_mean_squared_error: 1796231.1250
```

```
Epoch 17/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 3256544.5000 - mean_squared_error: 3256544.5000 - val_loss:
2274519.0000 - val_mean_squared_error: 2274519.0000
Epoch 18/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 3113113.7500 - mean_squared_error: 3113113.7500 - val_loss:
1637829.0000 - val_mean_squared_error: 1637829.0000
Epoch 19/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2981688.2500 - mean_squared_error: 2981688.2500 - val_loss:
2774809.5000 - val_mean_squared_error: 2774809.5000
Epoch 20/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2977179.0000 - mean_squared_error: 2977179.0000 - val_loss:
980998.2500 - val_mean_squared_error: 980998.2500
Epoch 21/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 3157494.0000 - mean_squared_error: 3157494.0000 - val_loss:
2272847.2500 - val_mean_squared_error: 2272847.2500
Epoch 22/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2850443.2500 - mean_squared_error: 2850443.2500 - val_loss:
1943925.6250 - val_mean_squared_error: 1943925.6250
Epoch 23/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2988249.2500 - mean_squared_error: 2988249.2500 - val_loss:
1684810.0000 - val_mean_squared_error: 1684810.0000
Epoch 24/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 2800195.5000 - mean_squared_error: 2800195.5000 - val_loss:
2541660.5000 - val_mean_squared_error: 2541660.5000
Epoch 25/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2889537.7500 - mean_squared_error: 2889537.7500 - val_loss:
1523431.6250 - val_mean_squared_error: 1523431.6250
Epoch 26/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2422113.5000 - mean_squared_error: 2422113.5000 - val_loss:
2379297.0000 - val_mean_squared_error: 2379297.0000
Epoch 27/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 2711265.7500 - mean_squared_error: 2711265.7500 - val_loss:
1972085.6250 - val_mean_squared_error: 1972085.6250
Epoch 28/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 2560140.2500 - mean_squared_error: 2560140.2500 - val_loss:
4373387.0000 - val_mean_squared_error: 4373387.0000
Epoch 29/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2655799.0000 - mean_squared_error: 2655799.0000 - val_loss:
1415040.0000 - val_mean_squared_error: 1415040.0000
Epoch 30/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2670892.7500 - mean_squared_error: 2670892.7500 - val_loss:
1857539.2500 - val_mean_squared_error: 1857539.2500
Epoch 31/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2472076.7500 - mean_squared_error: 2472076.7500 - val_loss:
1336035.1250 - val_mean_squared_error: 1336035.1250
```

```
Epoch 32/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2379229.7500 - mean_squared_error: 2379229.7500 - val_loss:
946278.0625 - val_mean_squared_error: 946278.0625
Epoch 33/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2142464.5000 - mean_squared_error: 2142464.5000 - val_loss:
1652163.1250 - val_mean_squared_error: 1652163.1250
Epoch 34/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2175563.7500 - mean_squared_error: 2175563.7500 - val_loss:
1070811.2500 - val_mean_squared_error: 1070811.2500
Epoch 35/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 2044567.0000 - mean_squared_error: 2044567.0000 - val_loss:
701671.2500 - val_mean_squared_error: 701671.2500
Epoch 36/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2261950.5000 - mean_squared_error: 2261950.5000 - val_loss:
991492.9375 - val_mean_squared_error: 991492.9375
Epoch 37/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2108139.2500 - mean_squared_error: 2108139.2500 - val_loss:
586335.1875 - val_mean_squared_error: 586335.1875
Epoch 38/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2125695.2500 - mean_squared_error: 2125695.2500 - val_loss:
529777.1875 - val_mean_squared_error: 529777.1875
Epoch 39/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1753975.3750 - mean_squared_error: 1753975.3750 - val_loss:
1807639.6250 - val_mean_squared_error: 1807639.6250
Epoch 40/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 3380227.0000 - mean_squared_error: 3380227.0000 - val_loss:
1842242.3750 - val_mean_squared_error: 1842242.3750
Epoch 41/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1929574.6250 - mean_squared_error: 1929574.6250 - val_loss:
1572208.3750 - val_mean_squared_error: 1572208.3750
Epoch 42/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1793748.7500 - mean_squared_error: 1793748.7500 - val_loss:
888658.1875 - val_mean_squared_error: 888658.1875
Epoch 43/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1670647.7500 - mean_squared_error: 1670647.7500 - val_loss:
1146297.1250 - val_mean_squared_error: 1146297.1250
Epoch 44/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1694000.2500 - mean_squared_error: 1694000.2500 - val_loss:
775783.5625 - val_mean_squared_error: 775783.5625
Epoch 45/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1644553.0000 - mean_squared_error: 1644553.0000 - val_loss:
1251415.1250 - val_mean_squared_error: 1251415.1250
Epoch 46/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1661752.8750 - mean_squared_error: 1661752.8750 - val_loss:
806461.4375 - val_mean_squared_error: 806461.4375
```

```
806461.4375 - val_mean_squared_error: 806461.4375
Epoch 47/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1571267.6250 - mean_squared_error: 1571267.6250 - val_loss:
617993.1875 - val_mean_squared_error: 617993.1875
Epoch 48/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1617691.3750 - mean_squared_error: 1617691.3750 - val_loss:
3339050.2500 - val_mean_squared_error: 3339050.2500
Epoch 49/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1794601.2500 - mean_squared_error: 1794601.2500 - val_loss:
980010.3750 - val_mean_squared_error: 980010.3750
Epoch 50/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1718850.8750 - mean_squared_error: 1718850.8750 - val_loss:
1297190.1250 - val_mean_squared_error: 1297190.1250
Epoch 51/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1271121.8750 - mean_squared_error: 1271121.8750 - val_loss:
2085128.2500 - val_mean_squared_error: 2085128.2500
Epoch 52/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 1384278.7500 - mean_squared_error: 1384278.7500 - val_loss:
2951445.7500 - val_mean_squared_error: 2951445.7500
Epoch 53/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1475912.6250 - mean_squared_error: 1475912.6250 - val_loss:
2085794.3750 - val_mean_squared_error: 2085794.3750
Epoch 54/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1286037.8750 - mean_squared_error: 1286037.8750 - val_loss:
3290280.5000 - val_mean_squared_error: 3290280.5000
Epoch 55/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1375237.3750 - mean_squared_error: 1375237.3750 - val_loss:
2283306.5000 - val_mean_squared_error: 2283306.5000
Epoch 56/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1255332.0000 - mean_squared_error: 1255332.0000 - val_loss:
498593.2500 - val_mean_squared_error: 498593.2500
Epoch 57/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1464300.6250 - mean_squared_error: 1464300.6250 - val_loss:
1820308.8750 - val_mean_squared_error: 1820308.8750
Epoch 58/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1160864.3750 - mean_squared_error: 1160864.3750 - val_loss:
2100757.0000 - val_mean_squared_error: 2100757.0000
Epoch 59/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1169024.6250 - mean_squared_error: 1169024.6250 - val_loss:
2021904.2500 - val_mean_squared_error: 2021904.2500
Epoch 60/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1193656.1250 - mean_squared_error: 1193656.1250 - val_loss:
2423895.0000 - val_mean_squared_error: 2423895.0000
Epoch 61/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1367581.1250 - mean_squared_error: 1367581.1250 - val_loss:
```

```
2829742.2500 - val_mean_squared_error: 2829742.2500
Epoch 62/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1051124.7500 - mean_squared_error: 1051124.7500 - val_loss: 1469840.6250 - val_mean_squared_error: 1469840.6250
Epoch 63/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1279472.5000 - mean_squared_error: 1279472.5000 - val_loss: 1438999.8750 - val_mean_squared_error: 1438999.8750
Epoch 64/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1126599.7500 - mean_squared_error: 1126599.7500 - val_loss: 4019581.0000 - val_mean_squared_error: 4019581.0000
Epoch 65/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1120569.6250 - mean_squared_error: 1120569.6250 - val_loss: 1588503.7500 - val_mean_squared_error: 1588503.7500
Epoch 66/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1274936.5000 - mean_squared_error: 1274936.5000 - val_loss: 1310876.7500 - val_mean_squared_error: 1310876.7500
Epoch 67/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1004033.5625 - mean_squared_error: 1004033.5625 - val_loss: 1951977.6250 - val_mean_squared_error: 1951977.6250
Epoch 68/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1201488.3750 - mean_squared_error: 1201488.3750 - val_loss: 958041.1250 - val_mean_squared_error: 958041.1250
Epoch 69/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1004543.5000 - mean_squared_error: 1004543.5000 - val_loss: 1343647.5000 - val_mean_squared_error: 1343647.5000
Epoch 70/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 966714.8750 - mean_squared_error: 966714.8750 - val_loss: 1113139.2500 - val_mean_squared_error: 1113139.2500
Epoch 71/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1098061.7500 - mean_squared_error: 1098061.7500 - val_loss: 2950213.7500 - val_mean_squared_error: 2950213.7500
Epoch 72/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1074784.1250 - mean_squared_error: 1074784.1250 - val_loss: 3385348.5000 - val_mean_squared_error: 3385348.5000
Epoch 73/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1019918.3750 - mean_squared_error: 1019918.3750 - val_loss: 2627133.5000 - val_mean_squared_error: 2627133.5000
Epoch 74/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1056050.3750 - mean_squared_error: 1056050.3750 - val_loss: 4303462.0000 - val_mean_squared_error: 4303462.0000
Epoch 75/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1007149.6250 - mean_squared_error: 1007149.6250 - val_loss: 3152230.0000 - val_mean_squared_error: 3152230.0000
Epoch 76/200
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1124362.3750 - mean_squared_error: 1124362.3750 - val_loss:
```

```
1125/1125 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1124562.5750 - mean_squared_error: 1124562.5750 - val_loss:
2143996.0000 - val_mean_squared_error: 2143996.0000
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
899.5124879440308 2136269.6177298366 1461.5983092935749 0.9921280145645142
```

PART (e)

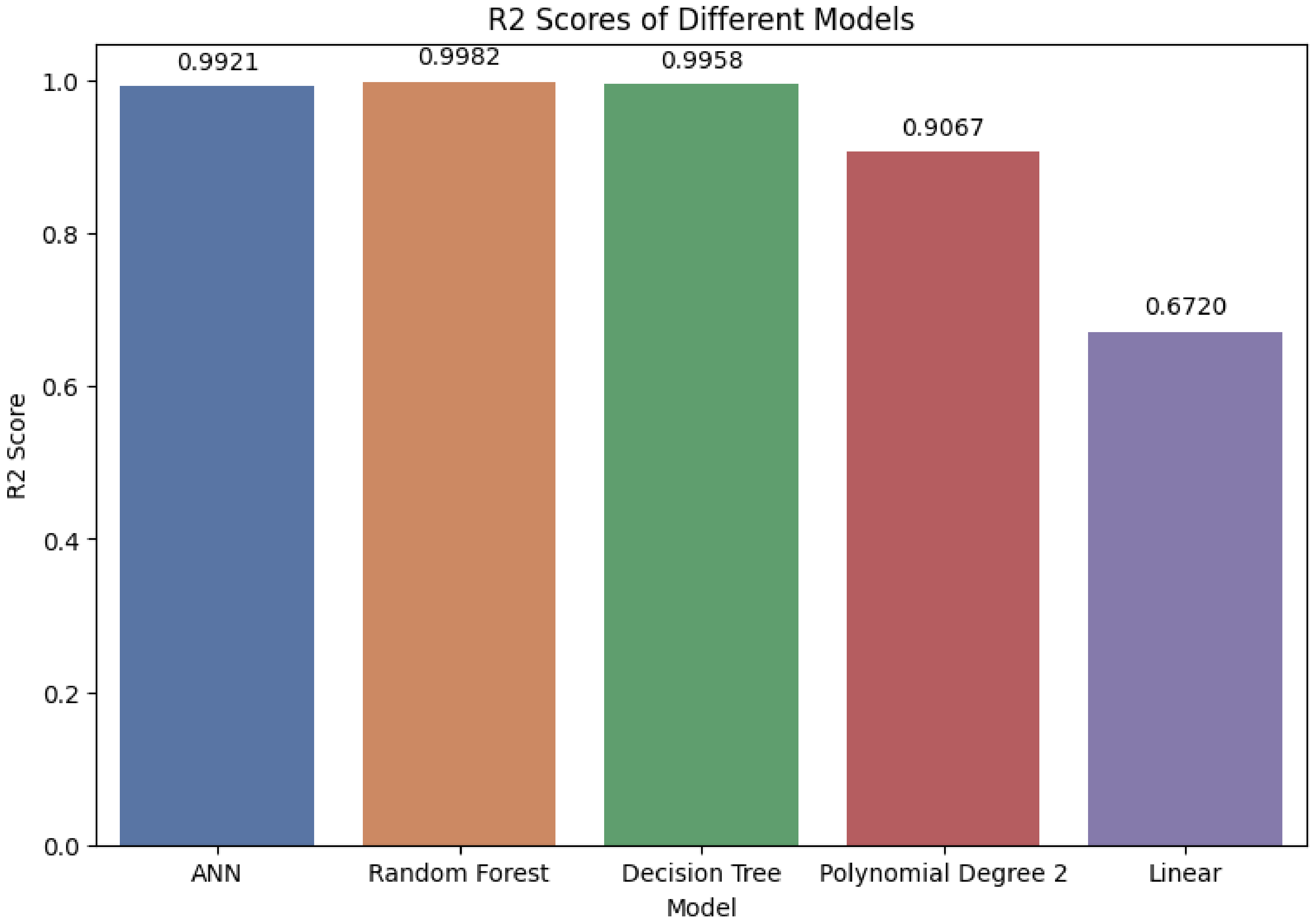
Based on the results of your analysis, what is the best model for predicting the price of a car and why? You should use suitable figures and evaluation metrics to support your conclusions.

In [60]:

```python
data = {
    'Model': ['ANN', 'Random Forest', 'Decision Tree', 'Polynomial Degree 2', 'Linear'],
    'R2 Score': [0.9921280145645142, 0.9982469614067221, 0.9957992050443717, 0.9066752884438702, 0.6719903658783444]
}

# Create a DataFrame
df_model = pd.DataFrame(data)

#Plotting
plt.figure(figsize=(9, 6))
barplot = sns.barplot(x='Model', y='R2 Score', data=df_model, palette='deep')
plt.title('R2 Scores of Different Models')
plt.xlabel('Model')
plt.ylabel('R2 Score')
# Display the R2 Score values on top of the bars
for p in barplot.patches:
    barplot.annotate(format(p.get_height(), '.4f'),
                     (p.get_x() + p.get_width() / 2., p.get_height()),
                     ha='center', va='center',
                     xytext=(0, 10),
                     textcoords='offset points')
plt.show()
barplot.figure.savefig("Model R2 Score")
```

```
C:\Users\hp\AppData\Local\Temp\ipykernel_4604\3903277179.py:11: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue`
and set `legend=False` for the same effect.
```

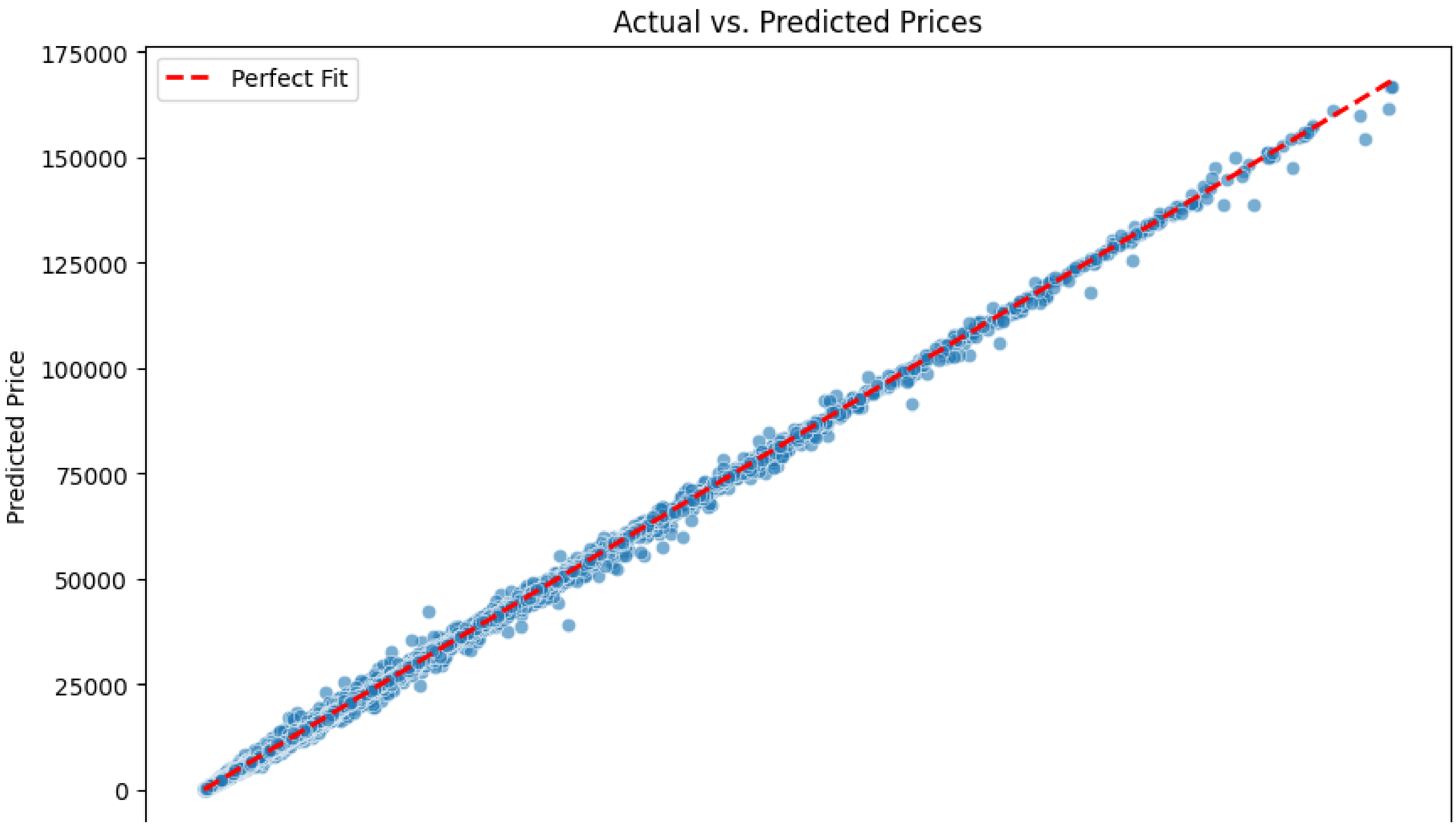
```
barplot = sns.barplot(x='Model', y='R2 Score', data=df_model, palette='deep')
```



In [54]:

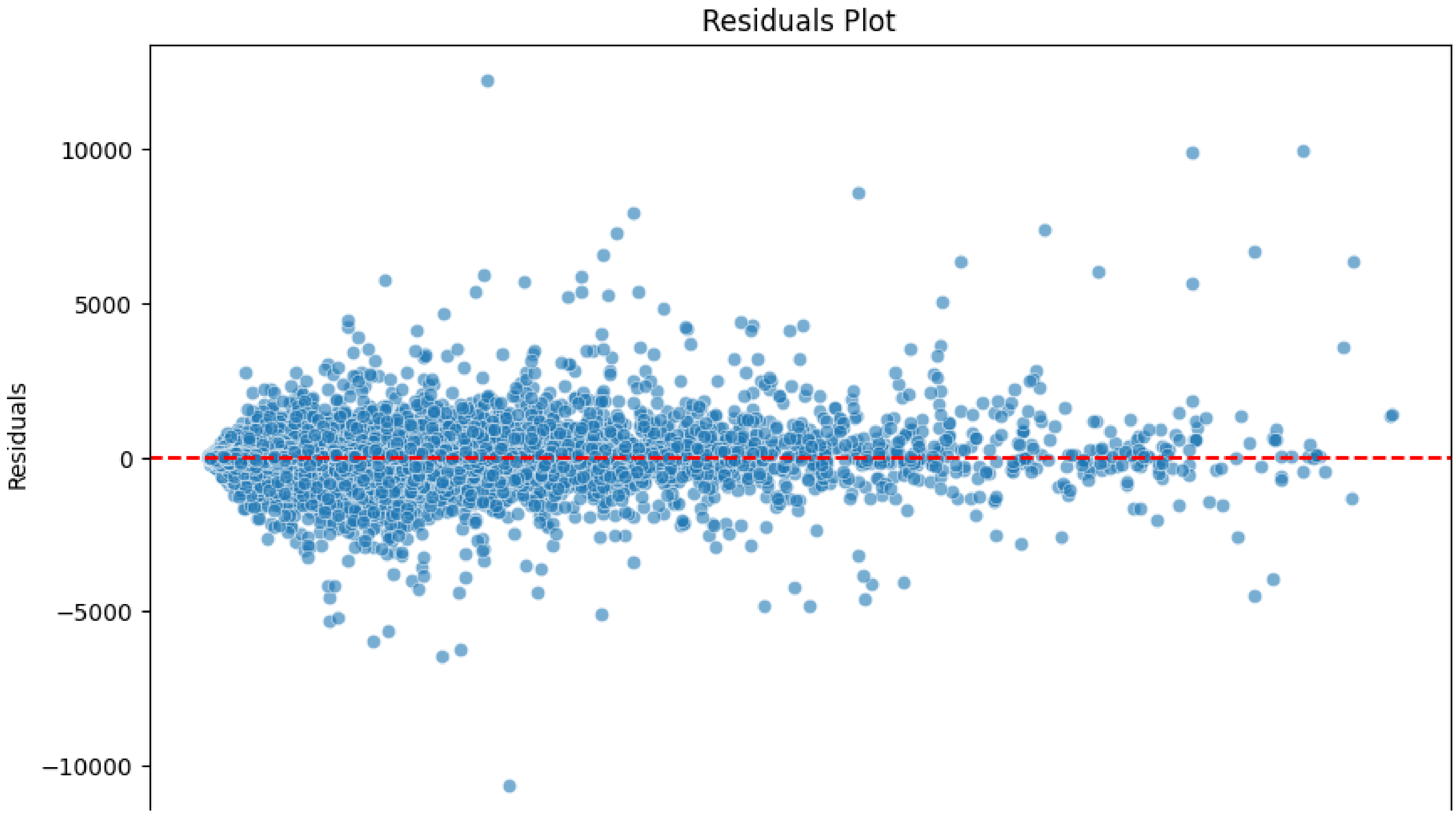
```
Total_Price_Pred = Price_RF.predict(x)

data = {
    'Actual Price': y,
    'Predicted Price': Total_Price_Pred
}
```

```
df = pd.DataFrame(data)

# plotting
plt.figure(figsize=(10, 6))
sns.scatterplot(x='Actual Price', y='Predicted Price', data=df, alpha=0.6)
plt.plot([df['Actual Price'].min(), df['Actual Price'].max()],
         [df['Actual Price'].min(), df['Actual Price'].max()],
         color='red', linestyle='--', linewidth=2, label='Perfect Fit')

plt.title('Actual vs. Predicted Prices')
plt.xlabel('Actual Price')
plt.ylabel('Predicted Price')
plt.legend()
plt.savefig("Actual vs Predicted")
plt.show()
```



In [55]:

```
residuals = y - Total_Price_Pred
plt.figure(figsize=(10, 6))
sns.scatterplot(x=Total_Price_Pred, y=residuals, alpha=0.6)
plt.axhline(y=0, color='red', linestyle='--')
plt.title('Residuals Plot')
plt.xlabel('Predicted Price')
plt.ylabel('Residuals')
plt.savefig("Residual Plot")
plt.show()
```



PART (f)

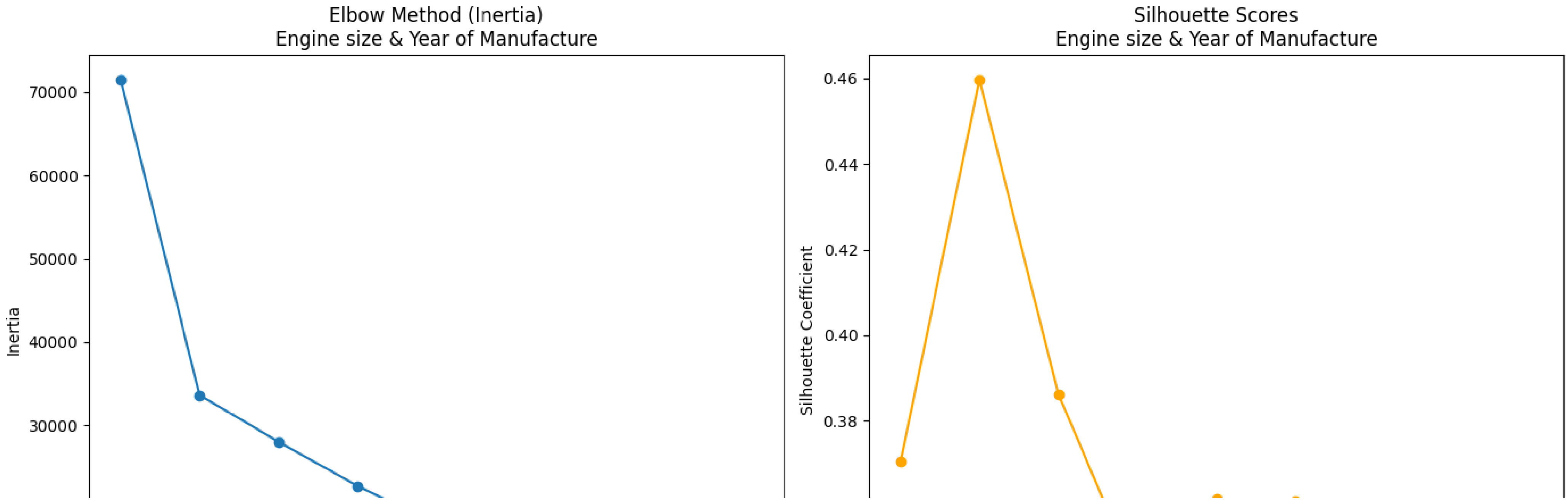
Consider different combinations of the numerical variables in the dataset to use as input features for the clustering algorithm. In each case, what is the optimal number of clusters (k) to use and why? Which combination of variables produces the best clustering results? Use appropriate evaluation metrics to support your conclusions.

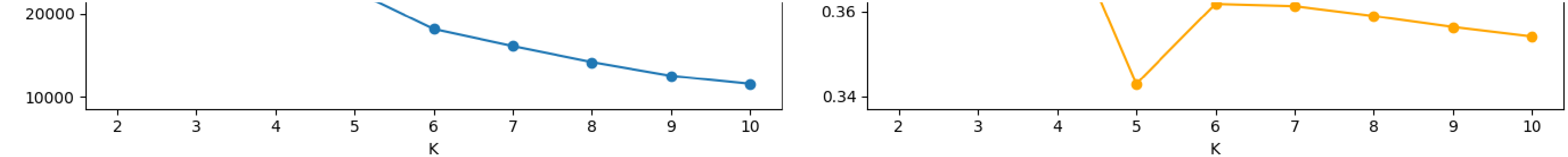
1) Engine Size and Year of Manufacturer

In [38]:

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

#Seperating features
features = ["Engine size", "Year of manufacture"]
x = df[features]

# Standardization
scale = StandardScaler()
scale.fit(x)
x_scaled = scale.transform(x)

#Checking optimal number of k
Inertia = []
Silhouette_Scores = []
for k in range(2,11):
    kmeans = KMeans(n_clusters = k, random_state = 42)
    kmeans.fit(x_scaled)
    inertia = kmeans.inertia_
    Inertia.append(inertia)
    silhouette_avg = silhouette_score(x_scaled, kmeans.labels_)
    Silhouette_Scores.append(silhouette_avg)
```

```python
# Elbow Graph
k_range = range(2, 11)

plt.figure(figsize=(14, 6))

# Plot Inertia (Elbow Method)
plt.subplot(1, 2, 1)
plt.plot(k_range, Inertia, 'o-', label='Inertia')
plt.xlabel("K")
plt.ylabel("Inertia")
plt.title("Elbow Method (Inertia)\nEngine size & Year of Manufacture")
plt.xticks(k_range)

# Plot Silhouette Scores
plt.subplot(1, 2, 2)
plt.plot(k_range, Silhouette_Scores, 'o-', color='orange', label='Silhouette Score')
plt.xlabel("K")
plt.ylabel("Silhouette Coefficient")
plt.title("Silhouette Scores\nEngine size & Year of Manufacture")
plt.xticks(k_range)

# Show the plots
plt.tight_layout()
plt.savefig("Engine & Year K_S graph")
plt.show()
```





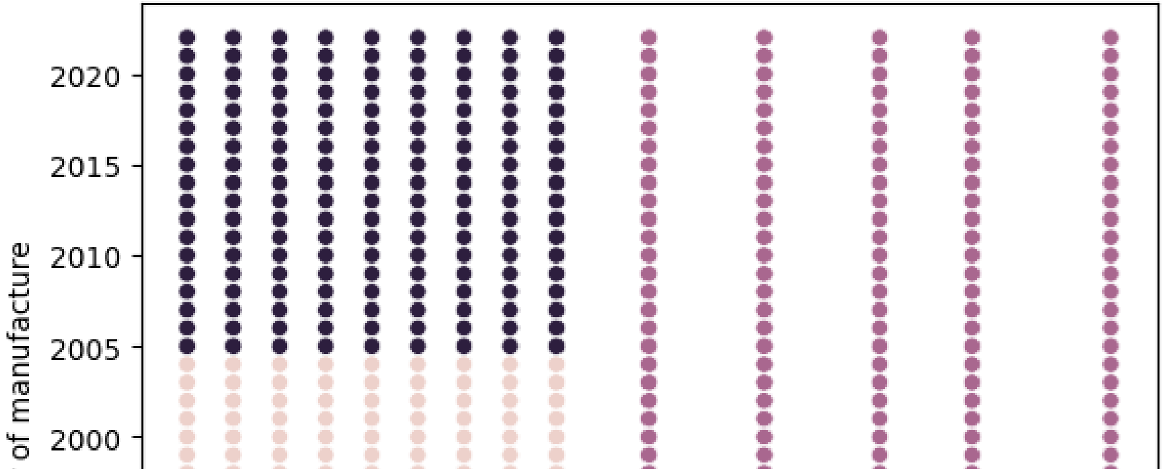
In [31]:

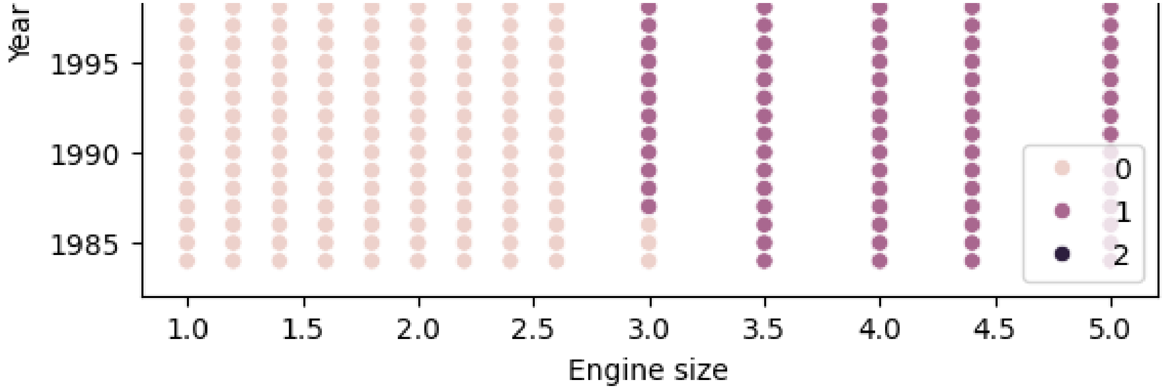
```
from sklearn.metrics import davies_bouldin_score

#Using optimal k obtained from elbow graph
kmeans = KMeans(n_clusters = 3, random_state = 42)
kmeans.fit(x_scaled)

#Prediction
cluster_labels_pred = kmeans.predict(x_scaled)

#Evaluation Metrics
db_score = davies_bouldin_score(x_scaled, cluster_labels_pred)
s_score = silhouette_score(x_scaled, cluster_labels_pred)
print("Should be close to 0: ",db_score)
print("Should be close to 1: ",s_score)
```

```
Should be close to 0:  0.7525001635420162
Should be close to 1:  0.4596955533294489
```

In [32]:

```
Engine_Year = sns.scatterplot(data = df, x = "Engine size", y = "Year of manufacture", hue = cluster_labels_pred)
```



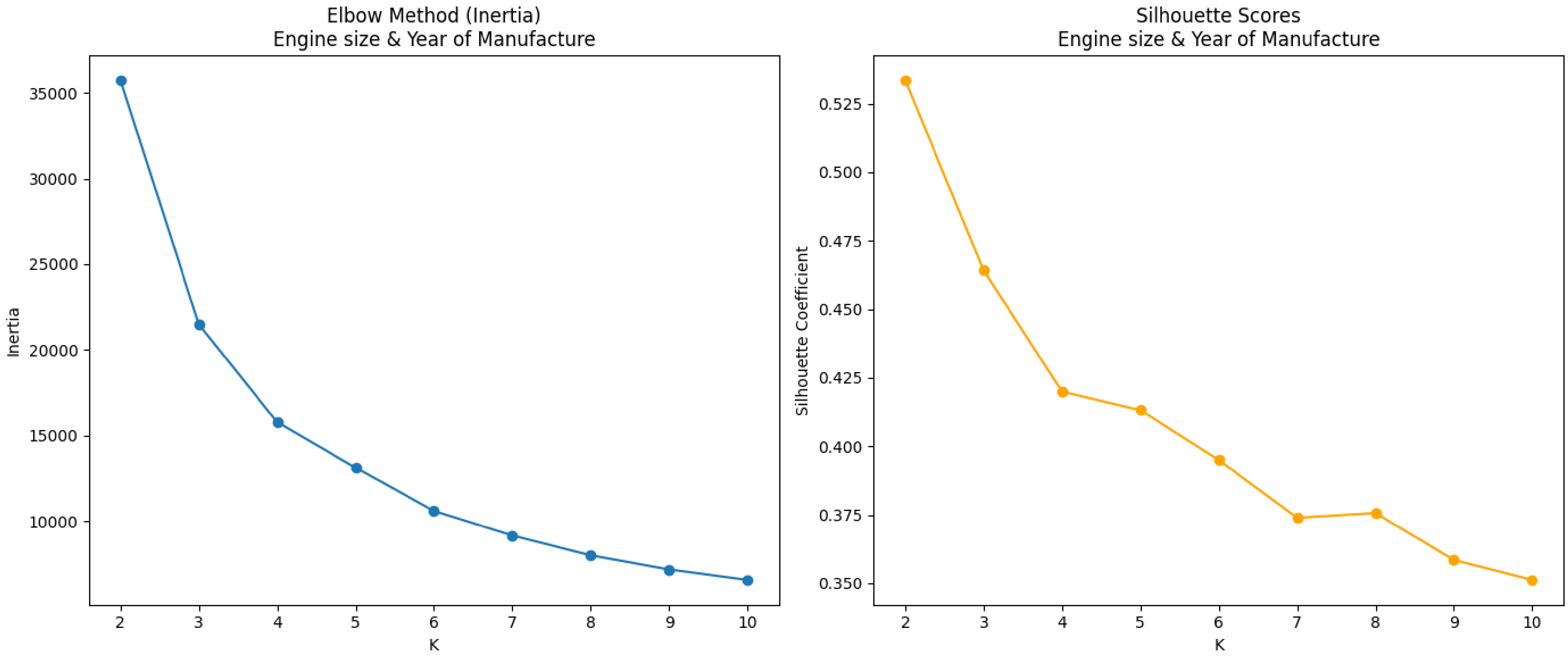


2) Year of Manufacturer and Mileage

In [35]:

```
#Seperating features
features = ["Year of manufacture", "Mileage"]
x = df[features]

# Standardization
scale = StandardScaler()
scale.fit(x)
x_scaled = scale.transform(x)

#Checking optimal number of k
Inertia = []
Silhouette_Scores = []
for k in range(2,11):
    kmeans = KMeans(n_clusters = k, random_state = 42)
    kmeans.fit(x_scaled)
    inertia = kmeans.inertia_
    Inertia.append(inertia)
    silhouette_avg = silhouette_score(x_scaled, kmeans.labels_)
    Silhouette_Scores.append(silhouette_avg)

# Elbow Graph
k_range = range(2, 11)

plt.figure(figsize=(14, 6))

# Plot Inertia (Elbow Method)
```

```
plt.subplot(1, 2, 1)
plt.plot(k_range, Inertia, 'o-', label='Inertia')
plt.xlabel("K")
plt.ylabel("Inertia")
plt.title("Elbow Method (Inertia)\nEngine size & Year of Manufacture")
plt.xticks(k_range)

# Plot Silhouette Scores
plt.subplot(1, 2, 2)
plt.plot(k_range, Silhouette_Scores, 'o-', color='orange', label='Silhouette Score')
plt.xlabel("K")
plt.ylabel("Silhouette Coefficient")
plt.title("Silhouette Scores\nEngine size & Year of Manufacture")
plt.xticks(k_range)

# Show the plots
plt.tight_layout()
plt.savefig("Year & Mileage K_S graph")
plt.show()
```



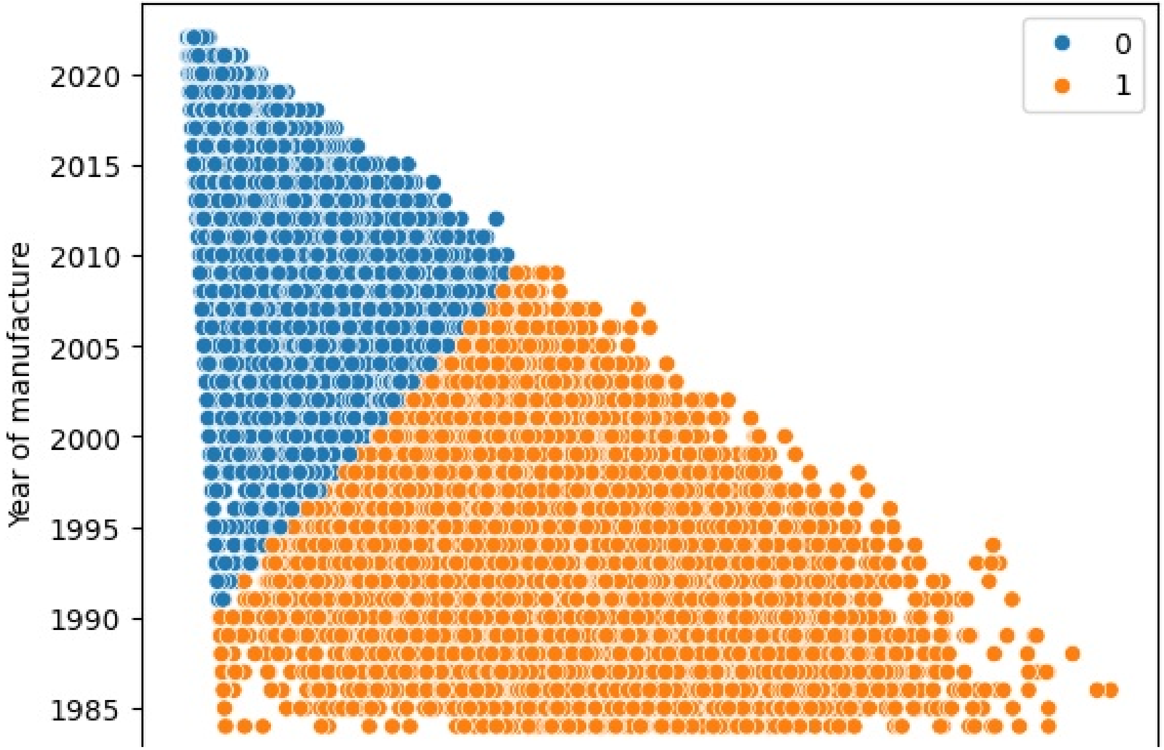
In [36]:

```
#Using optimal k obtained from elbow graph
kmeans = KMeans(n_clusters = 2, random_state = 42)
kmeans.fit(x_scaled)

#Prediction
cluster_labels_pred = kmeans.predict(x_scaled)

#Evaluation Metrics
db_score = davies_bouldin_score(x_scaled, cluster_labels_pred)
s_score = silhouette_score(x_scaled, cluster_labels_pred)
print("Should be close to 0: ",db_score)
print("Should be close to 1: ",s_score)
```

```
Should be close to 0:  0.6588265230316139
Should be close to 1:  0.5334978926912568
```

In [37]:

```
Year_Mileage = sns.scatterplot(data = df, x = "Mileage", y = "Year of manufacture", hue = cluster_labels_pred)
```

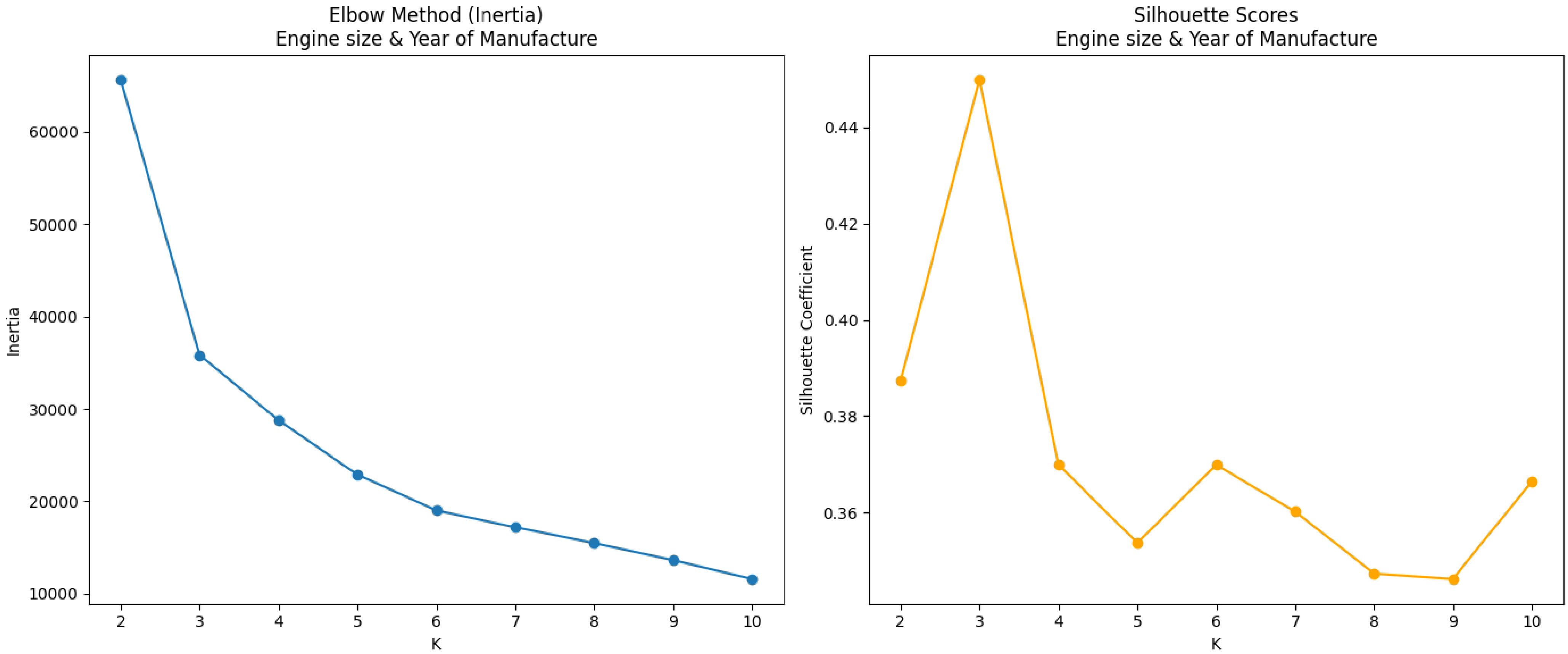


3) Mileage and Engine Size

In [39]:

```
#Seperating features
features = ["Mileage", "Engine size"]
x = df[features]

# Standardization
scale = StandardScaler()
scale.fit(x)
x_scaled = scale.transform(x)

#Checking optimal number of k
Inertia = []
Silhouette_Scores = []
for k in range(2,11):
    kmeans = KMeans(n_clusters = k, random_state = 42)
    kmeans.fit(x_scaled)
    inertia = kmeans.inertia_
    Inertia.append(inertia)
    silhouette_avg = silhouette_score(x_scaled, kmeans.labels_)
    Silhouette_Scores.append(silhouette_avg)

# Elbow Graph
k_range = range(2, 11)

plt.figure(figsize=(14, 6))

# Plot Inertia (Elbow Method)
plt.subplot(1, 2, 1)
plt.plot(k_range, Inertia, 'o-', label='Inertia')
plt.xlabel("K")
plt.ylabel("Inertia")
plt.title("Elbow Method (Inertia)\nEngine size & Year of Manufacture")
plt.xticks(k_range)

# Plot Silhouette Scores
```

```
plt.subplot(1, 2, 2)
plt.plot(k_range, Silhouette_Scores, 'o-', color='orange', label='Silhouette Score')
plt.xlabel("K")
plt.ylabel("Silhouette Coefficient")
plt.title("Silhouette Scores\nEngine size & Year of Manufacture")
plt.xticks(k_range)

# Show the plots
plt.tight_layout()
plt.savefig("Mileage & Engine K_S graph")
plt.show()
```



In [26]:

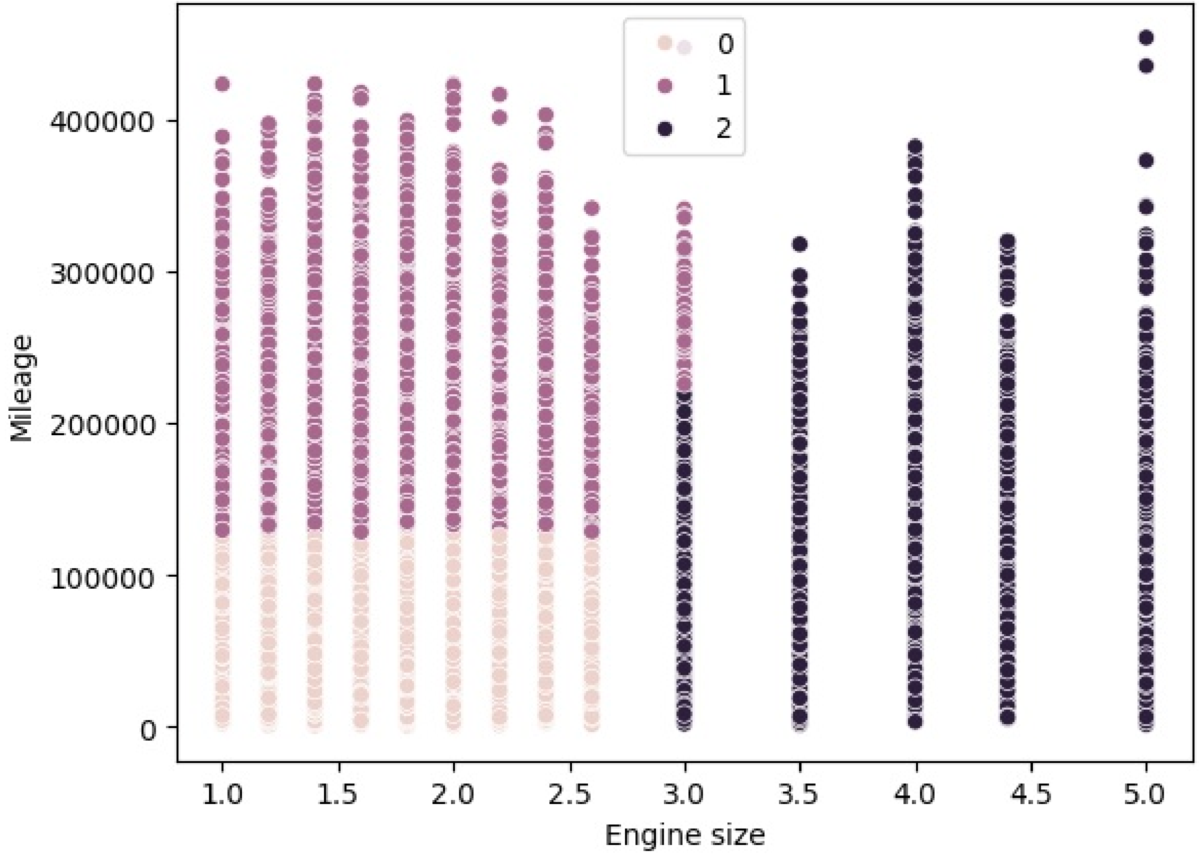
```
#Using optimal k obtained from elbow graph
kmeans = KMeans(n_clusters = 3, random_state = 42)
kmeans.fit(x_scaled)
```

```
#Prediction
cluster_labels_pred = kmeans.predict(x_scaled)

#Evaluation Metrics
db_score = davies_bouldin_score(x_scaled, cluster_labels_pred)
s_score = silhouette_score(x_scaled, cluster_labels_pred)
print("Should be close to 0: ",db_score)
print("Should be close to 1: ",s_score)
```

```
Should be close to 0:  0.7767410574042378
Should be close to 1:  0.4498399871492221
```

In [27]:

```
Mileage_Size = sns.scatterplot(data = df, x = "Engine size", y = "Mileage", hue = cluster_labels_pred)
```

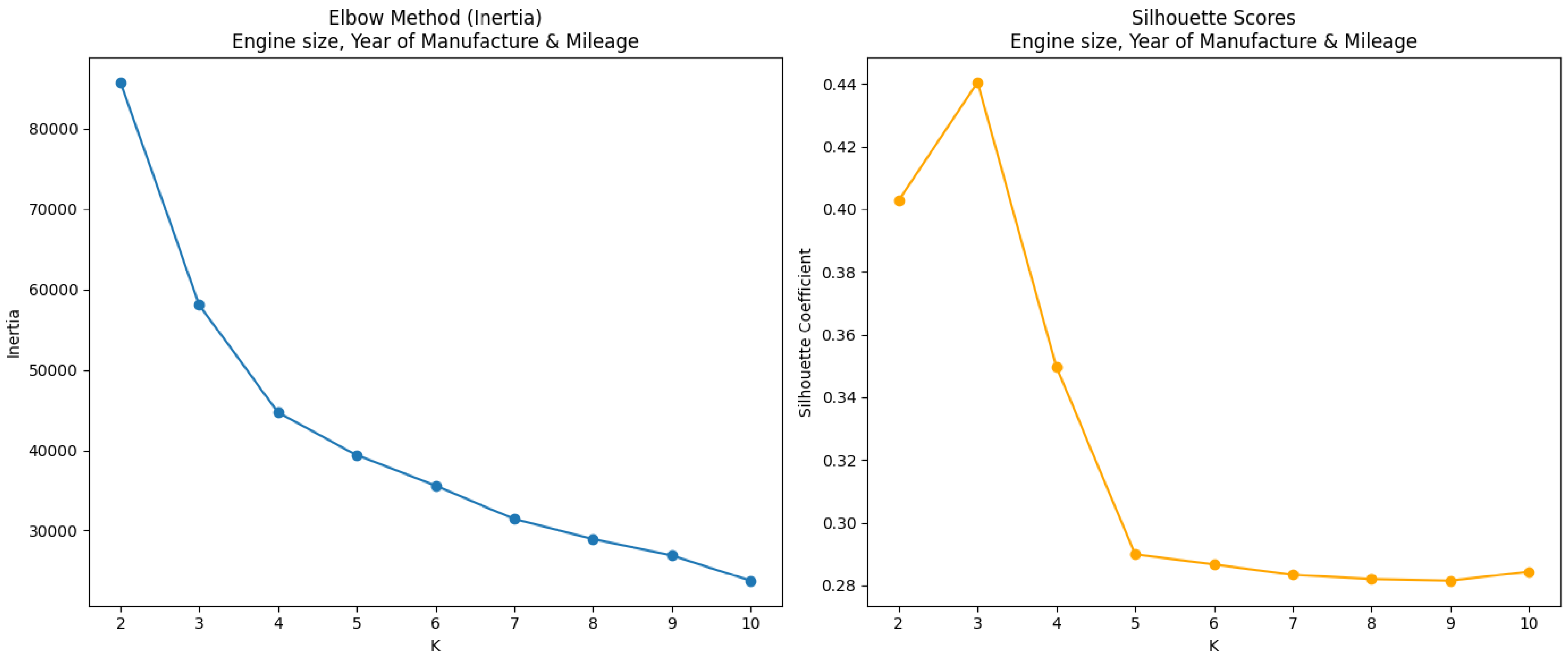


4) Engine Size, Year of Manufacture & Engine Size

In [40]:

```
#Seperating features
features = ["Year of manufacture", "Mileage", "Engine size"]
x = df[features]

# Standardization
scale = StandardScaler()
scale.fit(x)
x_scaled = scale.transform(x)

#Checking optimal number of k
Inertia = []
Silhouette_Scores = []
for k in range(2,11):
    kmeans = KMeans(n_clusters = k, random_state = 42)
    kmeans.fit(x_scaled)
    inertia = kmeans.inertia_
    Inertia.append(inertia)
    silhouette_avg = silhouette_score(x_scaled, kmeans.labels_)
    Silhouette_Scores.append(silhouette_avg)

# Elbow Graph
k_range = range(2, 11)

plt.figure(figsize=(14, 6))

# Plot Inertia (Elbow Method)
plt.subplot(1, 2, 1)
plt.plot(k_range, Inertia, 'o-', label='Inertia')
plt.xlabel("K")
plt.ylabel("Inertia")
plt.title("Elbow Method (Inertia)\nEngine size, Year of Manufacture & Mileage")
plt.xticks(k_range)

# Plot Silhouette Scores
plt.subplot(1, 2, 2)
plt.plot(k_range, Silhouette_Scores, 'o-', color='orange', label='Silhouette Score')
plt.xlabel("K")
plt.ylabel("Silhouette Coefficient")
plt.title("Silhouette Scores\nEngine size, Year of Manufacture & Mileage")
plt.xticks(k_range)

# Show the plots
```

```
plt.tight_layout()
plt.savefig("Mileage, Engine & Year K_S graph")
plt.show()
```



In [41]:

```
#Using optimal k obtained from elbow graph
kmeans = KMeans(n_clusters = 3, random_state = 42)
kmeans.fit(x_scaled)

#Prediction
cluster_labels_pred = kmeans.predict(x_scaled)

#Evaluation Metrics
db_score = davies_bouldin_score(x_scaled, cluster_labels_pred)
s_score = silhouette_score(x_scaled, cluster_labels_pred)
print("Should be close to 0: ",db_score)
print("Should be close to 1: ",s_score)
```

```
Should be close to 0:  0.8054720418101068
Should be close to 1:  0.4403994757844225
```

PART (g):

Compare the results of the k-Means clustering model from part (f) to at least one other clustering algorithm. Which algorithm produces the best clustering? Use suitable evaluation metrics to justify your answer.

Using Gaussian Mixture Models

In [42]:

```
from sklearn.mixture import GaussianMixture

def process_features(features, n_components, df):
    # Separating features
    x = df[features]

    # Standardization
    scale = StandardScaler()
    scale.fit(x)
    x_scaled = scale.transform(x)

    # Using optimal number of components
    gmm = GaussianMixture(n_components=n_components, random_state=42)
    gmm.fit(x_scaled)

    # Prediction
    gaussian_labels_pred = gmm.predict(x_scaled)

    # Evaluation Metrics
    db_score = davies_bouldin_score(x_scaled, gaussian_labels_pred)
    s_score = silhouette_score(x_scaled, gaussian_labels_pred)
    print(f"Features: {features}")
    print("Davies-Bouldin Score (Should be close to 0): ", db_score)
    print("Silhouette Score (Should be close to 1): ", s_score)
```

```python
# Process each set of features
process_features(["Engine size", "Year of manufacture"], 3, df)
process_features(["Mileage", "Year of manufacture"], 2, df)
process_features(["Engine size", "Mileage"], 3, df)
process_features(["Engine size", "Mileage", "Year of manufacture"], 3, df)
```

```
Features: ['Engine size', 'Year of manufacture']
Davies-Bouldin Score (Should be close to 0):  0.7537376543788059
Silhouette Score (Should be close to 1):  0.45985179228782713
Features: ['Mileage', 'Year of manufacture']
Davies-Bouldin Score (Should be close to 0):  0.7085385674366645
Silhouette Score (Should be close to 1):  0.47469157897147196
Features: ['Engine size', 'Mileage']
Davies-Bouldin Score (Should be close to 0):  0.7847224817819551
Silhouette Score (Should be close to 1):  0.4469170178653802
Features: ['Engine size', 'Mileage', 'Year of manufacture']
Davies-Bouldin Score (Should be close to 0):  0.8778846277473331
Silhouette Score (Should be close to 1):  0.39037818073737385
```