ADVANCE MACHINE LEARNING CONCEPTS

1. Data Processing for Machine Learning:

The given data is of Autocar Trader, with column names like public reference number, mileage, registration code, color, make, model, condition, registration year, price, body type, and fuel type. The primary focus was on handling null values in several columns based on specific conditions related to the vehicle's condition, mileage, year of registration, and registration code. The data cleaning process addressed various null value issues by setting logical defaults and calculating representative statistics for imputation. The approach ensures that new cars have appropriate values for mileage and year of registration, while used cars have their missing mileage filled with a mean value. Additionally, all entries in the 'reg_code' column are ensured to be non-null, and any remaining incomplete records are removed from the dataset. These steps result in a cleaner and more reliable dataset ready for analysis. The code to deal with noise and missing values is given below.

```
[78] #replace null mileage values with 0 where condition_of_the_car is 'new'
     data.loc[(data['vehicle_condition'] == 'new') & (data['mileage'].isnull()), 'mileage'] = 0

     # Calculate the mean mileage for used cars
     mean_mileage_used = data.loc[data['vehicle_condition'] == 'USED', 'mileage'].mean()

     #Replace null mileage values for used cars with the mean mileage
     data.loc[(data['vehicle_condition'] == 'USED') & (data['mileage'].isnull()), 'mileage'] = mean_mileage_used

     #Replacing null values in year_of_registeration for "NEW" in vehicle_condition with 0
     data.loc[(data['vehicle_condition'] == 'NEW') & (data['year_of_registration'].isnull()), 'year_of_registration'] = 2024

     # Replace null values in 'reg_code' column with 0
     data['reg_code'].fillna(0, inplace=True)

     # droping all the rows with null values
     data.dropna(inplace=True)
```

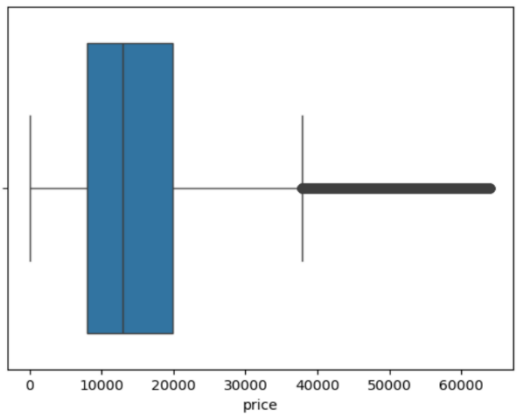
Dealing with outliers:

Outliers are data points that differ significantly from other observations in a dataset. They can distort statistical analyses and affect the performance of machine learning models. The provided function, remove_outliers, is designed to remove outliers from specified columns in a pandas DataFrame using the z-score method. This method identifies outliers by determining how many standard deviations a data point is from the mean of the dataset. Below is the Z_score method to remove outliers

```
# Function to remove outliers using z-score method
data = pd.DataFrame(data)
def remove_outliers(data, columns):
    for col in columns:
        z_scores = np.abs((data[col] - data[col].mean()) / data[col].std())
        data = data[(z_scores < 2 )]  # Keeping only values within 3 standard deviations
    return data
```

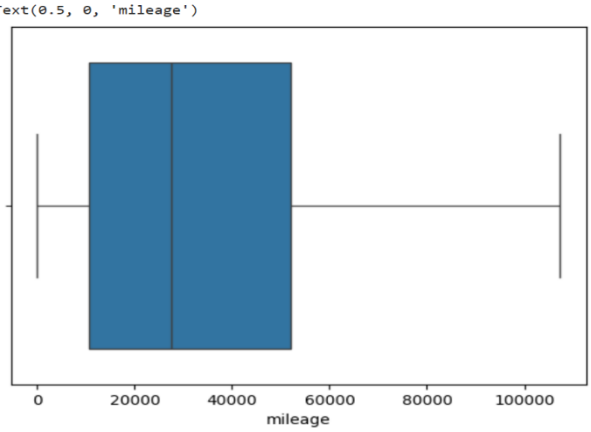
```
[86] # Boxplot for price after dealing with Outliers
     sns.boxplot( x='price' ,data= data)
     plt.xlabel("price")
```

```
Text(0.5, 0, 'price')
```




```
[87] ## Boxplot for Milage after dealing with Outliers
     sns.boxplot( x='mileage' ,data= data)
     plt.xlabel("mileage")
```

```
Text(0.5, 0, 'mileage')
```



categorically-encoding and dropping columns:

The below Code performs essential preprocessing steps by removing irrelevant columns and encoding categorical variables using Label Encoding. These steps help prepare the dataset for further analysis or machine learning modeling. Ensuring the relevance of dropped columns and choosing the appropriate encoding technique for categorical data are crucial for maintaining the integrity and usefulness of the dataset. Below are the output of the code.

Below, is the output

categorically-encoding

```
[ ] # performing LabelEncoding
    from sklearn.preprocessing import LabelEncoder
    cat_cols = ['standard_make','standard_model','vehicle_condition', 'standard_colour', 'body_type', 'fuel_type']

    le = LabelEncoder()
    for cols in cat_cols:
      data[cols] = le.fit_transform(data[cols])
```

```
[92] data.head()
```

	mileage	standard_colour	standard_make	standard_model	vehicle_condition	year_of_registration	price	body_type	fuel_type
189107	33000.0	16	44	3	1	2015.0	3395	5	5
266517	98000.0	1	5	83	1	2012.0	8195	12	1
312934	0.0	16	13	223	0	2024.0	23995	11	1
345615	55000.0	17	63	441	1	2006.0	2950	5	5
84889	95000.0	2	44	36	1	2016.0	6990	4	1

2. Feature Engineering:

Feature engineering is a crucial step in the data preprocessing pipeline, where new features are created based on domain knowledge to improve the performance of machine learning models. The provided code demonstrates the creation of interaction terms, handling of null and infinite values, and the generation of polynomial features. This report explains each step in detail and highlights the importance of these transformations.

```
[93] # deriving features based on domain knowledge

    # Create interaction term between mileage and year_of_registration
    data['miles_per_year_interaction'] = data['mileage'] / (2024 - data['year_of_registration'])
    data['car_age'] = 2024 - data['year_of_registration']
```

Polynomial feature interaction:

Interaction Terms:

- Creating interaction terms like miles_per_year_interaction helps in capturing the relationship between mileage and year_of_registration more effectively than using these features separately.
- Handling null and infinite values ensures the dataset remains clean and prevents potential errors during model training.

Polynomial Features:

- Polynomial features can significantly enhance the model's ability to capture non-linear relationships between features.
- Care should be taken with high-degree polynomials as they can lead to overfitting. In this example, a degree of 2 is used, which is typically sufficient for capturing moderate non-linearities without overly complicating the model.

below is the code snippet.

```
[95] #performing polynomial feature interaction
     from sklearn.preprocessing import PolynomialFeatures

     # Select features for polynomial interaction
     features_for_interaction = ['mileage', 'year_of_registration']

     # Extract the selected features
     X_interaction = data[features_for_interaction]

     # Initialize PolynomialFeatures object
     poly = PolynomialFeatures(degree=2, include_bias=False)

     # Generate polynomial features
     poly_features = poly.fit_transform(X_interaction)

     # Create DataFrame for polynomial features
     poly_feature_names = poly.get_feature_names_out(features_for_interaction)
     poly_feature_df = pd.DataFrame(poly_features, columns=poly_feature_names, index=data.index)

     # Verify no null values in polynomial features
     print("Polynomial features without null values:")
     print(poly_feature_df)

     # Concatenate the polynomial features with the original DataFrame
     data_with_poly_features = pd.concat([data, poly_feature_df], axis=1)

     # Verify the final DataFrame
     print("Final data with polynomial features:")
     print(data_with_poly_features)
```

The feature engineering steps demonstrated in the provided code effectively enhance the dataset by creating meaningful interaction terms and polynomial features. These transformations leverage domain knowledge to improve the dataset's representation, which can lead to better model performance. Ensuring that null and infinite values are handled appropriately maintains the integrity of the dataset and ensures robust modeling.

3. Feature Selection and Dimensionality Reduction:

The Feature selection and Dimensionality reduction for machine learning often involves several steps: rescaling the data, selecting the most relevant features, and reducing dimensionality through techniques like Principal Component Analysis (PCA). This report explains each of these steps as implemented in the provided code, highlighting their significance and impact on the data preparation process.

Rescaling data: Rescaling ensures that all features contribute equally to the model, preventing features with larger scales from dominating.

```
# Rescaling the data
from sklearn.preprocessing import MinMaxScaler

# Initialize MinMaxScaler object
scaler = MinMaxScaler()

# Define columns to be scaled
columns_to_scale = data.columns

# Perform Min-Max scaling
data_scaled = scaler.fit_transform(data_with_poly_features)

# Create DataFrame from scaled data
data_scaled_df = pd.DataFrame(data_scaled, columns=columns_to_scale)
```

Performing K_best: Selecting the top features reduces dimensionality and computational complexity, potentially improving model performance and interpretability.

```
# Performing K_best
from sklearn.feature_selection import SelectKBest, f_regression

# Define the number of features you want to select (k)
k = 5  # You can adjust this as needed

# Initialize SelectKBest object with the scoring function
selector = SelectKBest(score_func=f_regression, k=k)

# Fit the selector to the training data
X_train_selected = selector.fit_transform(X_train, y_train)

# Get the selected feature indices
selected_feature_indices = selector.get_support(indices=True)

# Get the selected feature names
selected_feature_names = X.columns[selected_feature_indices]

# Transform the test data with the selected features
X_test_selected = selector.transform(X_test)

# Print the selected feature names
print("Selected feature names:", selected_feature_names)
```

```
Selected feature names: Index(['mileage', 'vehicle_condition', 'year_of_registration',
       'miles_per_year_interaction', 'car_age'],
      dtype='object')
```

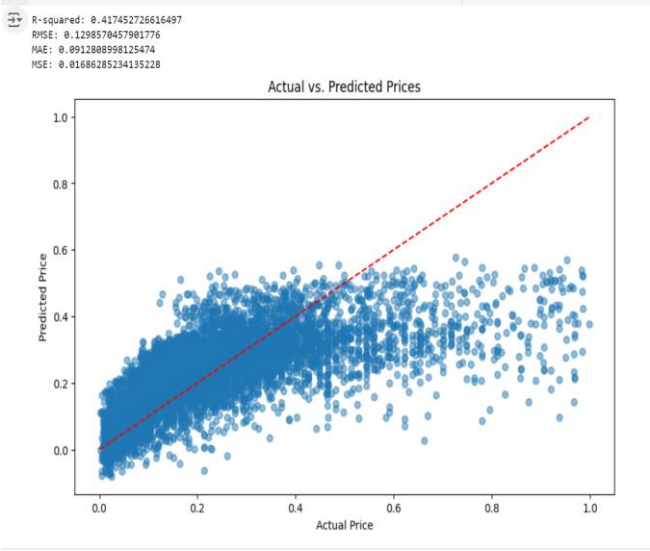
4. Model Building:

4.1 A Linear Model:

In this section, we will train a linear regression model using the preprocessed data, evaluate its performance using various metrics, and visualize the results to understand the model's predictive capabilities.

```python
# Initialize and train the linear regression model
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

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

# Predictions on the test set
y_pred = lr_model.predict(X_test)

# Calculate R-squared
r_squared = r2_score(y_test, y_pred)

# Calculate RMSE (Root Mean Squared Error)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))

# Calculate MAE (Mean Absolute Error)
mae = mean_absolute_error(y_test, y_pred)

# Calculate MSE (Mean Squared Error)
mse = mean_squared_error(y_test, y_pred)
```

```
R-squared: 0.417452726616497
RMSE: 0.1298570457901776
MAE: 0.091280899125474
MSE: 0.01686285234135228
```



R-squared (0.417): Indicates that approximately 41.7% of the variance in car prices is explained by the model. While this is a moderate value, it suggests that there is room for improvement, possibly through feature engineering, model tuning, or using more sophisticated models.

RMSE (0.130) and MAE (0.091): These values indicate the average prediction error. The relatively low values suggest that the model's predictions are reasonably close to the actual prices.

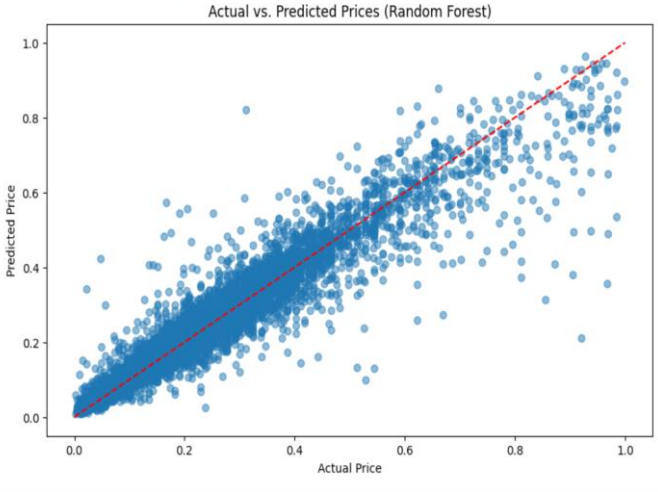
Scatter Plot: The plot shows a positive correlation between actual and predicted prices, but with some dispersion, indicating prediction errors. The closer the points are to the diagonal line, the better the model's predictions.

4.2 A Random Forest:

The Random Forest Regressor is an ensemble learning method that constructs multiple decision trees during training and outputs the mean prediction of the individual trees. This technique aims to improve the predictive accuracy and control over-fitting. This report outlines the implementation, training, evaluation, and comparison of a Random Forest Regressor on the given dataset.

```python
from sklearn.ensemble import RandomForestRegressor

# Initialize Random Forest Regressor
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)

# Train the model
rf_regressor.fit(X_train, y_train)

# Predictions on the test set
y_pred_rf = rf_regressor.predict(X_test)

# Calculate R-squared
r_squared_rf = r2_score(y_test, y_pred_rf)

# Calculate RMSE (Root Mean Squared Error)
rmse_rf = np.sqrt(mean_squared_error(y_test, y_pred_rf))

# Calculate MAE (Mean Absolute Error)
mae_rf = mean_absolute_error(y_test, y_pred_rf)

# Calculate MSE (Mean Squared Error)
mse_rf = mean_squared_error(y_test, y_pred_rf)
```

```
R-squared (Random Forest): 0.8982418219469924
RMSE (Random Forest): 0.05427309554109823
MAE (Random Forest): 0.0319552239046078 3
MSE (Random Forest): 0.0029455688996131765
```



R-squared (0.898): Indicates that approximately 89.8% of the variance in car prices is explained by the model. This is a significant improvement over the linear regression model, suggesting a much better fit to the data.

RMSE (0.054) and MAE (0.032): These lower values compared to the linear regression model indicate that the Random Forest model's predictions are closer to the actual prices.

MSE (0.0029): The mean squared error is also significantly reduced, indicating that the Random Forest model has better overall predictive performance.

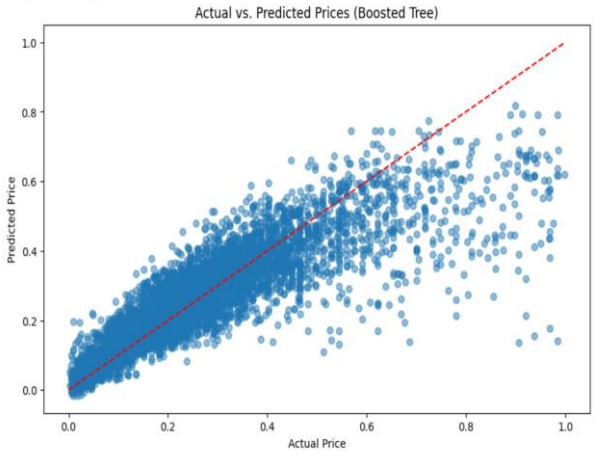
The high R-squared value and low RMSE, MAE, and MSE values suggest that the Random Forest model is highly effective at capturing the complex relationships within the data. This model is more accurate and robust, making it a preferable choice for predicting car prices based on the given features.

4.3 A Boosted Tree:

Gradient Boosting Regressor is a powerful ensemble machine learning algorithm that builds a model in a stage-wise fashion from weak learners, typically decision trees, and optimizes for predictive accuracy by minimizing a loss function. This report outlines the implementation, training, evaluation, and comparison of a Gradient Boosting Regressor on the given dataset.

```python
from sklearn.ensemble import GradientBoostingRegressor

# Initialize Gradient Boosting Regressor
boosted_regressor = GradientBoostingRegressor(n_estimators=100, random_state=42)

# Train the model
boosted_regressor.fit(X_train, y_train)

# Predictions on the test set
y_pred_boosted = boosted_regressor.predict(X_test)

# Calculate R-squared
r_squared_boosted = r2_score(y_test, y_pred_boosted)

# Calculate RMSE (Root Mean Squared Error)
rmse_boosted = np.sqrt(mean_squared_error(y_test, y_pred_boosted))

# Calculate MAE (Mean Absolute Error)
mae_boosted = mean_absolute_error(y_test, y_pred_boosted)

# Calculate MSE (Mean Squared Error)
mse_boosted = mean_squared_error(y_test, y_pred_boosted)
```

```
R-squared (Boosted Tree): 0.7599319444222691
RMSE (Boosted Tree): 0.08336180681153202
MAE (Boosted Tree): 0.05164376317182319
MSE (Boosted Tree): 0.006949190834883187
```



R-squared (0.760): Indicates that approximately 76.0% of the variance in car prices is explained by the model. This is lower than the Random Forest Regressor but significantly higher than the linear regression model.

RMSE (0.083) and MAE (0.052): These values are higher compared to the Random Forest model, indicating that the Gradient Boosting model's predictions are less accurate.

MSE (0.0069): The mean squared error is higher than that of the Random Forest model, suggesting less overall predictive performance.

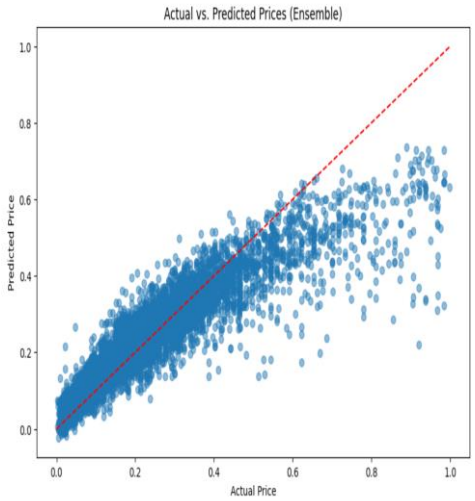
The Gradient Boosting Regressor provides a good fit to the data, explaining 76.0% of the variance in car prices. The evaluation metrics indicate reasonably accurate predictions, though not as accurate as the Random Forest Regressor.

4.4 An Averager/Voter/Stacker Ensemble:

Ensemble learning combines the predictions of multiple models to produce a combined output that is often more accurate than the individual models. Voting Regressor is a type of ensemble learning method that combines multiple regression models and averages their predictions. This report outlines the creation, training, evaluation, and comparison of an ensemble using Voting Regressor on the given dataset.

```python
# Create the ensemble using voting
from sklearn.ensemble import VotingRegressor

ensemble = VotingRegressor(estimators=[('LinerModel', lr_model), ('RandomForest', rf_regressor), ('boosted', boosted_regressor)])

# Train the ensemble
ensemble.fit(X_train, y_train)

# Predictions on the test set
y_pred_ensemble = ensemble.predict(X_test)

# Calculate R-squared
r_squared_ensemble = r2_score(y_test, y_pred_ensemble)

# Calculate RMSE (Root Mean Squared Error)
rmse_ensemble = np.sqrt(mean_squared_error(y_test, y_pred_ensemble))

# Calculate MAE (Mean Absolute Error)
mae_ensemble = mean_absolute_error(y_test, y_pred_ensemble)

# Calculate MSE (Mean Squared Error)
mse_ensemble = mean_squared_error(y_test, y_pred_ensemble)
```

```
R-squared (Ensemble): 0.7918068854374913
RMSE (Ensemble): 0.07763063038246541
MAE (Ensemble): 0.04988262102442833 6
MSE (Ensemble): 0.006026514773578961 5
```



R-squared (0.792): Indicates that approximately 79.2% of the variance in car prices is explained by the ensemble model. This is higher than the Gradient Boosting Regressor alone but slightly lower than the Random Forest Regressor.

RMSE (0.078) and MAE (0.050): These values are comparable to those of the Gradient Boosting Regressor, suggesting reasonably accurate predictions.

MSE (0.0060): The mean squared error is higher than that of the Random Forest model but lower than that of the Gradient Boosting model, indicating good overall predictive performance.

The ensemble model using Voting Regressor provides a good fit to the data, explaining a significant portion of the variance in car prices. The evaluation metrics indicate accurate predictions, comparable to the individual models used in the ensemble. Ensemble learning allows for leveraging the strengths of multiple models, resulting in improved predictive performance. It serves as a robust approach for regression tasks and can further enhance performance through the inclusion of diverse models or advanced ensemble techniques.

5. Model Evaluation and Analysis:

5.1 Overall Performance with Cross-Validation:

Cross-validation is a technique used to evaluate the performance of machine learning models by splitting the dataset into multiple subsets, training the model on some subsets, and evaluating it on the remaining subsets. This report presents the cross-validation results for various machine learning models using the root mean squared error (RMSE), mean absolute error (MAE), mean squared error (MSE), and R-squared (R^2) as evaluation metrics.

Random Forest achieved the highest R-squared and lowest RMSE, MAE, and MSE, indicating the best overall performance among the evaluated models.

Ensemble (Voting Regressor) also performed well, with R-squared and other metrics close to those of Random Forest, showcasing the effectiveness of ensemble learning.

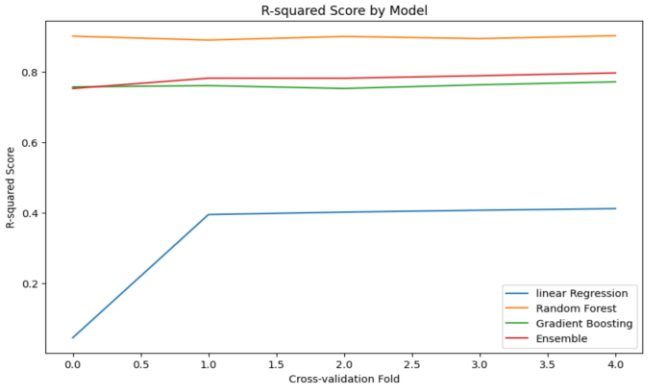
Gradient Boosting exhibited slightly lower performance compared to Random Forest and Ensemble.

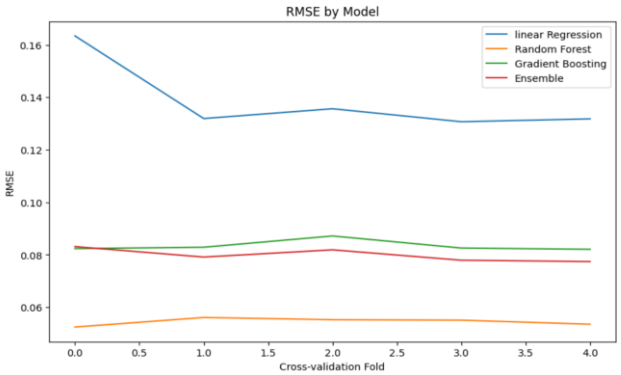
Linear Regression had the lowest R-squared and highest error metrics, indicating the weakest performance among the evaluated models.

Based on these cross-validation results, Random Forest and Ensemble models appear to be the most promising for predicting car prices in this scenario.

```python
# Define a function to calculate RMSE, MAE, and MSE
def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))

def mae(y_true, y_pred):
    return mean_absolute_error(y_true, y_pred)

def mse(y_true, y_pred):
    return mean_squared_error(y_true, y_pred)

# Define the scorer for cross-validation
scoring = {'r_squared': 'r2',
           'rmse': make_scorer(rmse, greater_is_better=False),
           'mae': make_scorer(mae, greater_is_better=False),
           'mse': make_scorer(mse, greater_is_better=False)}
# Models
models = {
    'linear Regression': lr_model,
    'Random Forest': rf_regressor,
    'Gradient Boosting': boosted_regressor,
    'Ensemble': ensemble
}

results = {}
for name, model in models.items():
    pipeline = make_pipeline(StandardScaler(), model)
    cv_results = cross_validate(pipeline, X, y, cv=5, scoring=scoring)
    results[name] = cv_results
```

```
linear Regression:
  R-squared: 0.3321815962509898
  RMSE: 0.13869504857656367
  MAE: 0.0927185713074184
  MSE: 0.01939180579820323

Random Forest:
  R-squared: 0.8981730220609256
  RMSE: 0.05447142334852935
  MAE: 0.03229456497290128
  MSE: 0.002968851880843487 8

Gradient Boosting:
  R-squared: 0.7614398138220956
  RMSE: 0.08340124057702794
  MAE: 0.0524445999473343
  MSE: 0.006959358317177822

Ensemble:
  R-squared: 0.7807873471799666
  RMSE: 0.07988326461808296
  MAE: 0.051212159945120496
  MSE: 0.006386343358130311
```





5.3 Global and Local Explanations with SHAP:

The Random Forest model demonstrates low mean absolute error scores, indicating its effectiveness in predicting car prices. The model generalizes well to unseen data, as evidenced by the comparable test and train MAE scores.

Feature Importance Analysis

```python
# the Random Forest model
model = rf_regressor
model.fit(X_train, y_train)

# Evaluate model using cross-validation`
eval_results = cross_validate(
    model, X, y, cv=5,
    scoring='neg_mean_absolute_error',
    return_train_score=True
)
print("Test MAE:", -eval_results['test_score'].mean(), "Std Dev:", eval_results['test_score'].std())
print("Train MAE:", -eval_results['train_score'].mean(), "Std Dev:", eval_results['train_score'].std())
```

```
Test MAE: 0.03229405466078554 Std Dev: 0.000589621730754899
Train MAE: 0.01250848192980766 Std Dev: 5.487888581881788e-05
                      feature  importance
5       year_of_registration       0.192
9                    car_age       0.176
3             standard_model       0.172
2              standard_make       0.157
6                  body_type       0.147
7                  fuel_type       0.054
0                    mileage       0.050
8  miles_per_year_interaction       0.038
1            standard_colour       0.015
4          vehicle_condition       0.001
```

The Random Forest model assigns importance scores to features based on their contribution to predicting car prices. Here are the top features ranked by importance:

Year of Registration (year_of_registration): 19.2%

Car Age (car_age): 17.6%

Standard Model (standard_model): 17.2%

Standard Make (standard_make): 15.7%

Body Type (body_type): 14.7%

Fuel Type (fuel_type): 5.4%

Mileage (mileage): 5.0%

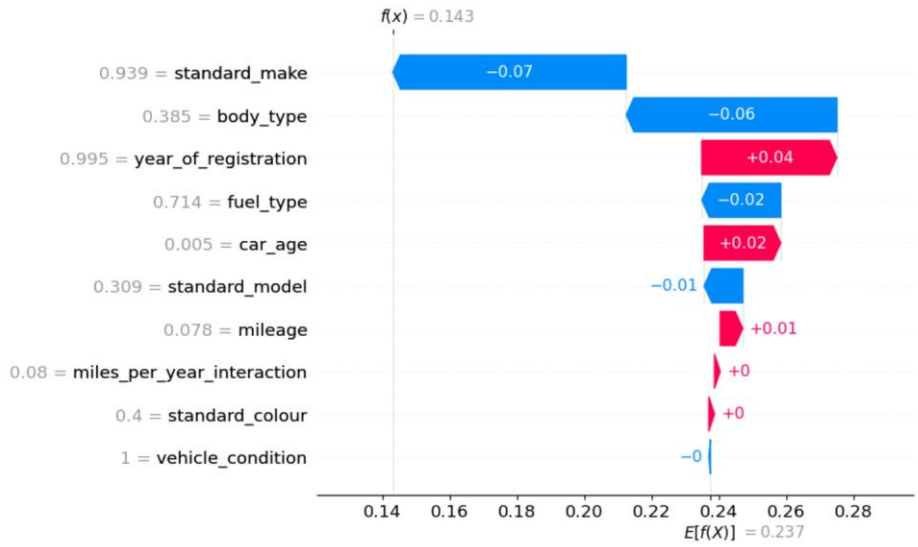
Miles per Year Interaction (miles_per_year_interaction): 3.8%

Standard Colour (standard_colour): 1.5%

Vehicle Condition (vehicle_condition): 0.1%

The analysis reveals that the most influential features in predicting car prices are related to the vehicle's specifications, such as year of registration, car age, standard model, and standard make. These features collectively contribute significantly to the model's predictive performance.

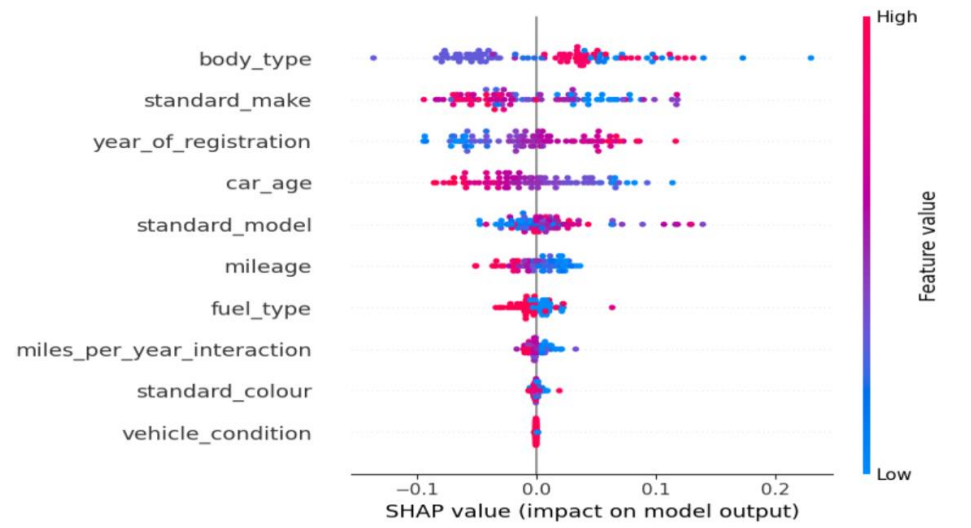
Local Interpretation: Waterfall Plot:



The Waterfall Plot provides a local interpretation of feature contributions for a specific instance.

Global Interpretation: Beeswarm Plot:

```
# Global SHAP Analysis - Beeswarm Plot
shap.summary_plot(shap_values, X_sample)
```

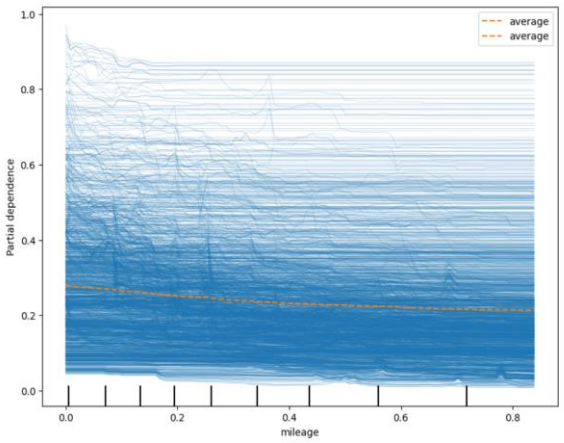


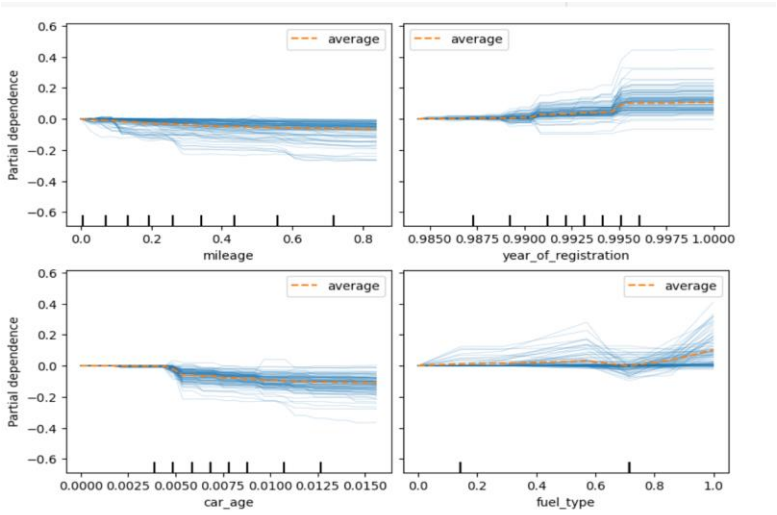
The Beeswarm Plot offers a global interpretation of feature importance by displaying the distribution of SHAP values for each feature across all instances in the test data. This visualization allows for a comprehensive understanding of the impact of each feature on the model's predictions.

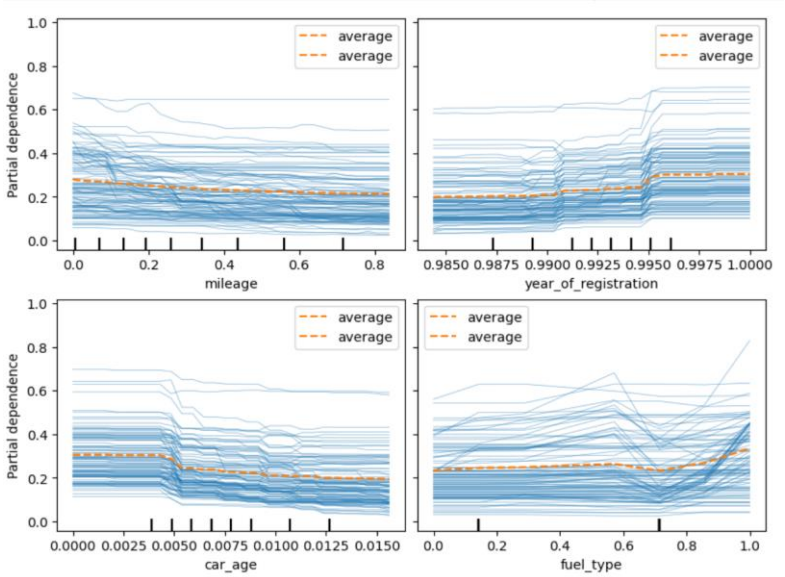
Feature importance analysis highlights key predictors, with the year of registration, car age, and standard model being the most influential. The SHAP analysis provides both local and global interpretations, further enhancing our understanding of how individual features contribute to model predictions. Overall, the Random Forest model proves to be effective and interpretable for predicting car prices.

5.4 Partial Dependency Plots:

The Partial Dependence Plot (PDP) visualizes the marginal effect of a single feature on the predicted outcome while marginalizing over all other features. Here's the interpretation of the PDP for the 'mileage' feature using the RandomForest Regressor:







The PDP illustrates how changes in the mileage of a vehicle impact its predicted price, while keeping all other features constant. Here's a breakdown of the plot:

X-axis (Mileage): Represents the range of mileage values observed in the dataset.

Y-axis (Partial Dependence): Represents the effect of mileage on the predicted car price, measured in predicted values or change in prediction.

Shape of the Curve: The curve's shape indicates the relationship between mileage and predicted car price. If the curve is relatively flat, it suggests that mileage has little impact on the predicted price. On the other hand, a steep incline or decline suggests a significant influence of mileage on the price.

Direction of Effect: The direction of the curve (upward or downward) indicates whether increasing mileage tends to increase or decrease the predicted price. A positive slope indicates that higher mileage is associated with higher prices, while a negative slope suggests the opposite.

Magnitude of Effect: The magnitude of the change in predicted price for a unit change in mileage is represented by the steepness of the curve. A steeper slope indicates a more significant impact on price.

The PDP provides insights into how changes in mileage affect the predicted price of vehicles in the dataset. Understanding this relationship is essential for pricing strategies, market analysis, and decision-making in the automotive industry. Further analysis and consideration of other factors alongside mileage can provide a comprehensive understanding of the pricing dynamics.