Manchester Metropolitan University

Coursework Specification (1CWK100)

6G7V0017 Advanced Machine Learning

Student ID: 22547202

Neha Alam Hussain

22/23 Semester 2

Contents

Notebook: https://colab.research.google.com/drive/1mDF1Cc7AhDiJWnWMEfta0g24Q_F6PAu8?usp=sharing

1. Data Processing for Machine Learning

Detection of Erroneous, Missing Values and Outliers

```
# it shows us which column has null values
df.isnull().sum()
```

```
public_reference          0
mileage                 127
reg_code              31857
standard_colour        5378
standard_make             0
standard_model            0
vehicle_condition         0
year_of_registration  33311
price                     0
body_type               837
crossover_car_and_van     0
```

Mileage: Our dataset contains 127 missing values for mileage. Additionally, there are erroneous mileage values, such as 999,999. According to AutoTrader's website, the maximum mileage option for selling or buying a vehicle is 200,000 miles. This suggests that the platform considers cars with over 200,000 miles as outliers or less relevant for their customers. There are also zero values for used cars, which could be erroneous.

Registration Code: The reg_code column has 608 missing values for used vehicles. Other null values mostly belong to new vehicles, which do not have a registration code assigned yet.

Standard Color: A total of 5,378 records have missing standard_colour values.

Year of Registration: Our dataset contains 2,062 records without a year of registration, despite their condition being marked as 'USED'. There are also a few erroneous values like 1007 and 1015.

Body Type: There are 837 records with null values in the body_type column.

Fuel Type: The fuel_type column contains 601 records with null values.

Price: While there are no missing values in the price column, the maximum value is 9,999,999, which is exceptionally high. Only six rows have such high values. The minimum value is 120.

Dealing with Erroneous, Missing Values and Outliers

First, we will remove the six records with a price of 9,999,999 due to the erroneous price and the missing year of registration and registration code.

```
df.drop(df[df['price'] == 9999999].index, inplace=True)
df.reset_index(drop = True)
```

Next, we will address missing values in the dataset using mean, median, and mode, depending on the data type and skewness. Categorical variables will be filled using the mode value of the respective column. If the feature is numeric and the data is not skewed, we will use the mean to fill null values. Otherwise, we will employ the median. We have three categorical columns (standard_colour, body_type, fuel_type) with missing values. We will determine the mode for each column and fill the missing values accordingly.

```
df.fillna({ 'standard_colour': str(df['standard_colour'].mode()[0]), 'body_type': str(df['body_type'].mode()[0]), 'fuel_type': str(df['fuel_type'].mode()[0])},inplace=True)
```

For the year of registration, we will fill missing values using the reg_code, but to do the vice versa we also need the months in addition to the year of registration to fill reg_code since the reg_code age identifier changes twice a year (on the 1st of March and September).

```
# Fixing year_of_registration column
reg_code_mapping = {
    'A': 1983, 'B': 1984, 'C': 1985, 'D': 1986, 'E': 1987, 'F': 1988,'G': 1989, 'H': 1990, 'J': 1991, 'K': 1992, 'L': 1993, 'M': 1994,
    'N': 1995, 'P': 1996, 'R': 1997, 'S': 1998, 'T': 1999, 'V': 2000,'W': 2000, 'X': 2000, 'Y': 2001,
    '02': 2002,'03': 2003,'04': 2004,'05': 2005,'06': 2006,'07': 2007,'08': 2008,'09': 2009,
    '10': 2010,'11': 2011,'12': 2012,'13': 2013,'14': 2014,'15': 2015,'16': 2016,'17': 2017,'18': 2018,'19': 2019,'20': 2020,
    '51': 2001,'52': 2002, '53': 2003,'54': 2004,'55': 2005,'56': 2006,'57': 2007,'58': 2008,'59': 2009,
    '60': 2010,'61': 2011,'62': 2012,'63': 2013,'64': 2014,'65': 2015,'66': 2016,'67': 2017,'68': 2018,'69': 2019,'70': 2020
}

for index, row in df.iterrows():
    if row['reg_code'] in reg_code_mapping:
        df.at[index, 'year_of_registration'] = reg_code_mapping[row['reg_code']]
```

But here we are assuming that the cars are registered between March-August and filling the reg_code with year_of_registration as this is the closest value and better than filling with mode of the column if we have year_of_registration.

```
[101] def year_to_reg_code(x):
        str_x = str(x)
        decimal_position = str_x.find('.')
        if decimal_position != -1:
            return (str_x[decimal_position - 2: decimal_position])
        else:
            return None
```

```
[102] for index, row in df.iterrows():
        if pd.isnull(row['reg_code']) and row['vehicle_condition'] == 'USED' and pd.notnull(row['year_of_registration']):
            df.at[index, 'reg_code'] = year_to_reg_code(row['year_of_registration'])
```

After analyzing the numerical columns (mileage, year_of_registration), we found that they are skewed. Therefore, we will fill the remaining null values with the median for used cars.

```
for index, row in df.iterrows():
    if pd.isnull(row['year_of_registration']) and row['vehicle_condition'] == 'USED':
        df.at[index, 'year_of_registration'] = 2016
```

```
[227] df.fillna({'mileage': df['mileage'].median()},inplace = True)
```

Finally, we will fill the remaining values in the year_of_registration column for new vehicles with the current year, which will help us calculate the age of the vehicle.

```
[117] df['reg_code']=df['reg_code'].fillna('23')
      df['year_of_registration']=df['year_of_registration'].fillna(2023)
```

By following these steps, we have successfully handled all missing and erroneous values in our dataset.

```
df[df.isna().any(axis=1)]
```

public_reference	mileage	reg_code	standard_colour	standard_make	standard_model	vehicle_condition	year_of_registration	price	body_type	crossover_car_and_van	fuel_type

Next, we handle outliers in mileage and price columns of our dataset using the interquartile range method. This is crucial for the robustness of the model. We will filter out records that are outliers.

```
[121] # Calculate Interquartile Range for mileage and price to deal with outliers
      mileage_Q1 = df['mileage'].quantile(0.25)
      mileage_Q3 = df['mileage'].quantile(0.75)
      mileage_IQR = mileage_Q3 - mileage_Q1

      price_Q1 = df['price'].quantile(0.25)
      price_Q3 = df['price'].quantile(0.75)
      price_IQR = price_Q3 - price_Q1

      # Find the lower and upper bounds for outliers
      mileage_lower_bound = mileage_Q1 - 1.5 * mileage_IQR
      mileage_upper_bound = mileage_Q3 + 1.5 * mileage_IQR

      price_lower_bound = price_Q1 - 1.5 * price_IQR
      price_upper_bound = price_Q3 + 1.5 * price_IQR

      # Filter out the outliers in mileage and price
      df = df[(df['mileage'] > mileage_lower_bound) & (df['mileage'] < mileage_upper_bound) & (df['price'] > price_lower_bound) & (df['pr
```

Encoding Categorical Columns

```
def custom__onehot_encoding(data, column):
    # Create a copy of the DataFrame to avoid modifying the original one
    data_copy = data.copy()

    # Find the top 10 most frequent categories for the given column
    top_10_occurring_cat = data_copy[column].value_counts().sort_values(ascending=False).head(10).index

    # Create 10 binary variables for each category
    for cat in top_10_occurring_cat:
        data_copy[f"{column}_{cat}"] = np.where(data_copy[column] == cat, 1, 0)

    return data_copy
```

```
for col in categorical_features:
  df = custom__onehot_encoding(df, col)

df.drop(categorical_features, axis=1, inplace=True)
```

Now to encode the categorical columns, custom one hot encoding has been used as the sklearn onehotencoder was increasing the dimensionality since we have multiple features with multiple categories so its not efficient to use that. After analyzing we have customized the encoder to take the top 10 categories and apply onehot encoding method on it.

Rescaling Data

We are using MinMax to normalize values in mileage and price columns. The idea is to normalize the values between 0 to 1.

```
dataset_normalized=df.copy(deep=True)
dataset_normalized['mileage'] = (dataset_normalized['mileage'] - dataset_normalized['mileage'].min()) / (dataset_normalized['mileage'].max() - dataset_normalized['mileage'].min
dataset_normalized['price'] = (dataset_normalized['price'] - dataset_normalized['price'].min()) / (dataset_normalized['price'].max() - dataset_normalized['price'].min())
```

Split data into predictors and target

```
[133] X = dataset_normalized.drop('price', axis=1)
      y = dataset_normalized['price']
```

Train Test split

```
# Split data into train, validation, and test sets (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

2. Feature Engineering

Derive Features Based on Domain Knowledge

We have created a new feature called "mileage_type" to classify vehicles based on their mileage as low, medium, or high. This is important since vehicles with lower mileage tend to sell at higher prices. To encode these categories, we will use 0 for low, 1 for medium, and 2 for high.

Before setting specific thresholds, we conducted an analysis of the dataset after initial data processing, including handling missing values and removing erroneous records. We used the "describe" function on the mileage feature to understand the data dispersion. Considering that the average mileage of the vehicles is approximately 0.29 (normalized), we determined the following thresholds to categorize the mileage as low, medium, or high.

```
def categorize_mileage(mileage):
    if mileage < 0.3:
        return 0  #low
    elif mileage >= 0.3 and mileage < 0.6:
        return 1  #medium
    else:
        return 2  #high
```

Vehicle age is an influential factor in determining its price, with newer cars generally commanding higher prices than older ones. We created a new feature, "vehicle_age," using the "year_of_registration" column to reflect this relationship. We used 2023 as the base year to calculate vehicle age, subtracting the registration year from it. For example, a vehicle registered in 2021 would have an age of 2 years, while new vehicles would be assigned a value of 0.

```
[285] def vehicle_age_calculation(year_of_registration):
          if pd.isnull(year_of_registration):
              return 0
          else:
              return (2023-year_of_registration)
```

Polynomial and Interaction Features

```
[71] poly_int_pipe = Pipeline(
         steps=[
             ("poly_int", PolynomialFeatures(degree=2, include_bias=False))  ]
     ).set_output(transform='pandas')
     poly_int_pipe
```

```
Pipeline
PolynomialFeatures
```

We have created a pipeline to create polynomial and interaction features which is called in the custom transformer.

```
def polyintfeatures(df):
    poly_int_features=poly_int_pipe.fit_transform(df[numeric_features])
    poly_int_features.drop(numeric_features,axis=1, inplace=True)
    poly_int_features = poly_int_features.reset_index(drop=True)
    df = df.reset_index(drop=True)
    df = pd.concat([df,poly_int_features], axis=1)
    return df
```

```
[75] class FeatureEngineering(BaseEstimator, TransformerMixin):
       def __init__(self):
         print('\n>>>>>>>init() called.\n')

       def fit(self, X, y = None):
         return self

       def transform(self, X, y = None):
         X_trans = X.copy() # creating a copy to avoid changes to original dataset
         X_trans['mileage_type'] = X_trans['mileage'].apply(categorize_mileage)
         X_trans['vehicle_age'] = X_trans['year_of_registration'].apply(vehicle_age_calculation)
         X_trans = polyintfeatures(X_trans)
         return X_trans
```

The transformer applies two additional functions to the dataset, creating "mileage_type" and "vehicle_age" features and generating polynomial and interaction features. This transformer will be utilized later in the pipeline for feature scaling and dimensionality reduction. We created these functions, pipelines, and transformers to ensure scalability, as the same pre-processing steps must be applied to the test set.

3. Feature Selection and Dimensionality Reduction

In this section of the pipeline, we focus on reducing the number of features in the dataset to improve model interpretability, reduce the risk of overfitting, and decrease computational complexity. The pipeline includes custom transformers, drop_features and remove_correlated_features, as well as scikit-learn's built-in VarianceThreshold transformer.

Removing Features Based on Domain Knowledge

The drop_features custom transformer, is designed to remove specific features from the dataset that are deemed irrelevant or redundant. In this case, we drop the columns 'public_reference', 'reg_code', and 'crossover_car_and_van'. By doing so, we eliminate potential noise in the dataset and allow the model to focus on more relevant features. A custom transformer is created to drop the values which will be called in the pipeline later.

```
[77] class drop_features(BaseEstimator, TransformerMixin):
       def __init__(self):
         print('')

       def fit(self, X, y = None):
         return self

       def transform(self, X, y = None):
         X_trans = X.copy() # creating a copy to avoid changes to original dataset
         X_trans = X.drop(['public_reference', 'reg_code', 'crossover_car_and_van'], axis=1)
         return X_trans
```

Automated selection and Dimensionality reduction

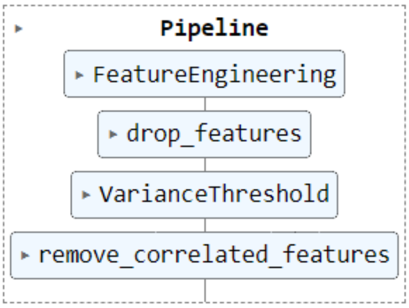
We explored various feature selection and dimensionality reduction techniques to optimize our model's performance and reduce computational complexity. These techniques include Variance Threshold, correlation-based feature removal, SelectKBest, Ridge regression, Recursive Feature Elimination (RFE), SelectFromModel, and Principal Component Analysis (PCA). By evaluating these methods, we aim to efficiently select the most relevant features and improve the model's overall performance.

```
class remove_correlated_features(BaseEstimator, TransformerMixin):
  def __init__(self):
    print('')

  def fit(self, X, y = None):
    return self

  def transform(self, X, y = None):
    X_trans = X.copy() # creating a copy to avoid changes to original dataset
    corr_matrix = X_trans.corr().abs()
    upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(np.bool))
    to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]
    return X_trans.drop(X_trans.columns[to_drop], axis=1)
```

```
[79] pipe= Pipeline([
        ('feature_engineering',FeatureEngineering() ),
        ('drop_features', drop_features()),
        ('feature_selection_variance', VarianceThreshold()),
      ('feature_selection_correlation', remove_correlated_features()),
        #("featsel", SelectKBest(f_regression, k=10)),
        #    ("regr", Ridge(alpha=100))
      #('feature_selection_rfe', RFE(estimator=LinearRegression(), n_features_to_select=8, step=1)),
      #('feature_selection_model', SelectFromModel(estimator=LassoCV())),
       # ('dimensionality_reduction', PCA(n_components=4))
  ])
```



4. Model Building

As we have a regression problem at hand, we will be using linear regression, random forest, gradient boosted regressor, stacking regressor, and voting regressor algorithms on our dataset. We will be using three evaluation metrics: mean absolute error (MAE), mean squared error (MSE) and r-squared method. The evaluation metrics will help us to decide which model performs best on our dataset. The lower the MAE and MSE values, the better the model has performed. In the case of the r-squared method, a value closer to 1 means that model has fitted better.

```
[83] def evaluate_model(model, X, y, cv=5):
         scores = cross_val_score(model, X, y, cv=cv, scoring='neg_mean_squared_error')
         rmse_scores = np.sqrt(-scores)
         return rmse_scores.mean(), rmse_scores.std()
```

A Linear Model

```
[84] linear_model = LinearRegression()
     linear_model.fit(X_train_transformed, y_train)
     linear_rmse_mean, linear_rmse_std = evaluate_model(linear_model, X_train_transformed, y_train)
     print(f"Linear Model: Mean RMSE: {linear_rmse_mean}, Standard Deviation: {linear_rmse_std}")
```

```
Linear Model: Mean RMSE: 0.14929622116717173, Standard Deviation: 0.0006586743843254114
```

```
linear_y_pred = linear_model.predict(X_test_transformed)
linear_mae = mean_absolute_error(y_test, Linear_y_pred)
linear_mse = mean_squared_error(y_test, Linear_y_pred)
linear_rmse = mean_squared_error(y_test, Linear_y_pred, squared=False)
linear_r2 = r2_score(y_test, Linear_y_pred)
print("Mean Absolute Error:", linear_mae)
print("Mean Squared Error:", linear_mse)
print("Root Mean Squared Error:", linear_rmse)
print("R-squared:", linear_r2)
```

```
Mean Absolute Error: 0.11015289500293725
Mean Squared Error: 0.022176762359500373
Root Mean Squared Error: 0.14891864342485925
R-squared: 0.518584132999736
```

The Linear Model has an RMSE of 0.1493 and an R-squared value of 0.5186. This model has the highest RMSE and the lowest R-squared value among all models, indicating that it is the least accurate model in predicting the target variable (price). It is likely that the relationship between the features and the target variable is not linear, which is why the linear model's performance is relatively poor.

A Random Forest

```
random_forest = RandomForestRegressor(n_estimators=100, random_state=42)
random_forest.fit(X_train_transformed, y_train)
random_forest_rmse_mean, random_forest_rmse_std = evaluate_model(random_forest, X_train_transformed, y_train)
print(f"Random Forest: Mean RMSE: {random_forest_rmse_mean}, Standard Deviation: {random_forest_rmse_std}")
```

```
Random Forest: Mean RMSE: 0.05892254881960918, Standard Deviation: 0.0004695506028487416
```

```
[87] rf_y_pred = random_forest.predict(X_test_transformed)
     rf_mae = mean_absolute_error(y_test, rf_y_pred)
     rf_mse = mean_squared_error(y_test, rf_y_pred)
     rf_rmse = mean_squared_error(y_test, rf_y_pred, squared=False)
     rf_r2 = r2_score(y_test, rf_y_pred)
     print("Mean Absolute Error:", rf_mae)
     print("Mean Squared Error:", rf_mse)
     print("Root Mean Squared Error:", rf_rmse)
     print("R-squared:", rf_r2)
```

```
Mean Absolute Error: 0.03802442897389503
Mean Squared Error: 0.0033525956595036465
Root Mean Squared Error: 0.05790160325503644
R-squared: 0.9272214437816778
```

The Random Forest model has an RMSE of 0.0589 and an R-squared value of 0.9272. This model significantly outperforms the Linear Model, with a much lower RMSE and a higher R-squared value. The Random Forest model is able to capture complex relationships between the features and the target variable, which may not be adequately modelled by a linear model. This indicates that there may be non-linear relationships and interactions between the features in the dataset.

A Boosted Tree

```
boosted_tree = GradientBoostingRegressor(n_estimators=100, random_state=42)
boosted_tree.fit(X_train_transformed, y_train)
boosted_tree_rmse_mean, boosted_tree_rmse_std = evaluate_model(boosted_tree, X_train_transformed, y_train)
print(f"Boosted Tree: Mean RMSE: {boosted_tree_rmse_mean}, Standard Deviation: {boosted_tree_rmse_std}")
```

```
Boosted Tree: Mean RMSE: 0.10089791522353724, Standard Deviation: 0.0007024001930368758
```

```
boostedtree_y_pred = boosted_tree.predict(X_test_transformed)
boosted_mae = mean_absolute_error(y_test, boostedtree_y_pred)
boosted_mse = mean_squared_error(y_test, boostedtree_y_pred)
boosted_rmse = mean_squared_error(y_test, boostedtree_y_pred, squared=False)
boosted_r2 = r2_score(y_test, boostedtree_y_pred)
print("Mean Absolute Error:", boosted_mae)
print("Mean Squared Error:", boosted_mse)
print("Root Mean Squared Error:", boosted_rmse)
print("R-squared:", boosted_r2)
```

```
Mean Absolute Error: 0.06945796203314109
Mean Squared Error: 0.010109618403134629
Root Mean Squared Error: 0.10054659816788745
R-squared: 0.7805391684462037
```

The Boosted Tree model has an RMSE of 0.1009 and an R-squared value of 0.7805. This model performs better than the Linear Model but not as well as the Random Forest model. Like the Random Forest model, the Boosted Tree model is also capable of capturing non-linear relationships and interactions between the features, but it seems that the Random Forest model is better suited for this dataset.

An Averager/Voter/Stacker Ensemble

Stacking Regressor

[90]
```
# Stacking Ensemble
estimators = [('linear', linear_model),
            # ('rf', best_rf),
              ('rf', random_forest),
              ('gb', boosted_tree)]
stacking_regressor = StackingRegressor(estimators=estimators, final_estimator=LinearRegression())
stacking_regressor.fit(X_train_transformed, y_train)
stacking_rmse_mean, stacking_rmse_std = evaluate_model(stacking_regressor, X_train_transformed, y_train)
print(f"Stacking Ensemble: Mean RMSE: {stacking_rmse_mean}, Standard Deviation: {stacking_rmse_std}")
```

```
Stacking Ensemble: Mean RMSE: 0.058611360969216285, Standard Deviation: 0.0004647477491735175
```

```
stackingRegr_y_pred = stacking_regressor.predict(X_test_transformed)
stackingRegr_mae = mean_absolute_error(y_test, stackingRegr_y_pred)
stackingRegr_mse = mean_squared_error(y_test, stackingRegr_y_pred)
stackingRegr_rmse = mean_squared_error(y_test, stackingRegr_y_pred, squared=False)
stackingRegr_r2 = r2_score(y_test, stackingRegr_y_pred)
print("Mean Absolute Error:", stackingRegr_mae)
print("Mean Squared Error:", stackingRegr_mse)
print("Root Mean Squared Error:", stackingRegr_rmse)
print("R-squared:", stackingRegr_r2)
```

```
Mean Absolute Error: 0.03800079215902855
Mean Squared Error: 0.0033190767559690823
Root Mean Squared Error: 0.057611429039463015
R-squared: 0.9279490762351625
```

The Stacking Ensemble model has an RMSE of 0.0586 and an R-squared value of 0.9279. This model performs very similarly to the Random Forest model, with a slightly lower RMSE and a slightly higher R-squared value. The Stacking Ensemble model combines the predictions of several base models and aims to reduce both the bias and the variance of the predictions. However, in this case, the improvement over the Random Forest model is marginal.

Voting Regressor

```
voting_model = VotingRegressor([('linear', linear_model), ('rf', random_forest), ('gb', boosted_tree)])
voting_model.fit(X_train_transformed, y_train)
voting_rmse_mean, voting_rmse_std = evaluate_model(voting_model, X_train_transformed, y_train)
print(f"Stacking Ensemble: Mean RMSE: {voting_rmse_mean}, Standard Deviation: {voting_rmse_std}")
```

```
Stacking Ensemble: Mean RMSE: 0.08933492777005225, Standard Deviation: 0.0003421684523968673
```

[93]
```
votingRegr_y_pred = voting_model.predict(X_test_transformed)
votingRegr_mae = mean_absolute_error(y_test, votingRegr_y_pred)
votingRegr_mse = mean_squared_error(y_test, votingRegr_y_pred)
votingRegr_rmse = mean_squared_error(y_test, votingRegr_y_pred, squared=False)
votingRegr_r2 = r2_score(y_test, votingRegr_y_pred)
print("Mean Absolute Error:", votingRegr_mae)
print("Mean Squared Error:", votingRegr_mse)
print("Root Mean Squared Error:", votingRegr_rmse)
print("R-squared:", votingRegr_r2)
```

```
Mean Absolute Error: 0.06299268146649002
Mean Squared Error: 0.00788994122837756
Root Mean Squared Error: 0.08882534113853748
R-squared: 0.8287241917703407
```

The Voting Ensemble model has an RMSE of 0.0893 and an R-squared value of 0.8287. This model performs better than the Linear Model and the Boosted Tree model, but not as well as the Random Forest and Stacking Ensemble models. The Voting Ensemble model combines the predictions of several base models by averaging their predictions or using majority voting. The performance of this model suggests that it can capture some of the complex relationships between the features and the target variable, but not as effectively as the Random Forest and Stacking Ensemble models.

5. Model Evaluation and Analysis

```
[109] models = [('Linear Model', linear_model), ('Random Forest', random_forest), ('Boosted Tree', boosted_tree), ('Stacking Ensemble', stack
```

```
[108] scores = [Linear_y_pred, rf_y_pred, boostedtree_y_pred, stackingRegr_y_pred, votingRegr_y_pred]
```

Overall Performance with Cross-Validation

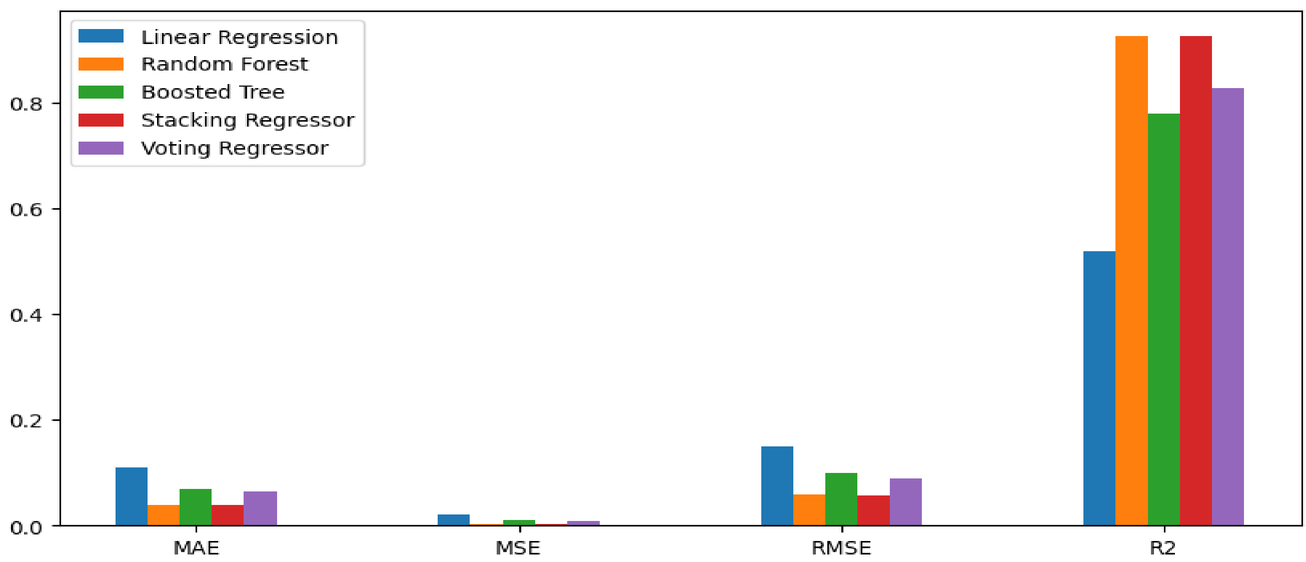
```
for (name, model), y_pred in zip(models, scores):
    mae = mean_absolute_error(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    r2 = r2_score(y_test, y_pred)
    print(f"{name} Cross-Validation Scores:")
    print("Mean Absolute Error:", mae)
    print("Mean Squared Error:", mse)
    print("Root Mean Squared Error:", rmse)
    print("R-squared:", r2)
    print('---------------------------------------------')
```

The cross-validation scores reveal that the Random Forest and Stacking Ensemble models perform the best, with the highest R-squared values of 0.9272 and 0.9279, respectively, indicating that they can explain around 92.7% of the variation in the target variable (price). These models also exhibit the lowest Root Mean Squared Error (RMSE) values, demonstrating their superior predictive accuracy.

The Boosted Tree and Voting Regressor models have moderately good performance, with R-squared values of 0.7805 and 0.8287, respectively. While their RMSE values are higher than those of the Random Forest and Stacking Ensemble models, they still perform reasonably well in predicting vehicle prices.

The Linear Model has the weakest performance among all the models, with the lowest R-squared value of 0.5186 and the highest RMSE of 0.1489. This suggests that the linear model is not able to capture the underlying patterns in the data as effectively as the other models and can only explain about 51.86% of the variation in the target variable.

```
Linear Model Cross-Validation Scores:
Mean Absolute Error: 0.11015289500293725
Mean Squared Error: 0.022176762359500373
Root Mean Squared Error: 0.14891864342485925
R-squared: 0.518584132999736
---------------------------------------------
Random Forest Cross-Validation Scores:
Mean Absolute Error: 0.03802442897389503
Mean Squared Error: 0.0033525956595036465
Root Mean Squared Error: 0.05790160325503644
R-squared: 0.9272214437816778
---------------------------------------------
Boosted Tree Cross-Validation Scores:
Mean Absolute Error: 0.06945796203314109
Mean Squared Error: 0.010109618403134629
Root Mean Squared Error: 0.10054659816788745
R-squared: 0.7805391684462037
---------------------------------------------
Stacking Ensemble Cross-Validation Scores:
Mean Absolute Error: 0.03800079215902855
Mean Squared Error: 0.0033190767559690823
Root Mean Squared Error: 0.057611429039463015
R-squared: 0.9279490762351625
---------------------------------------------
voting Regressor Cross-Validation Scores:
Mean Absolute Error: 0.06299268146649002
Mean Squared Error: 0.00788994122837756
Root Mean Squared Error: 0.08882534113853748
R-squared: 0.8287241917703407
---------------------------------------------
```

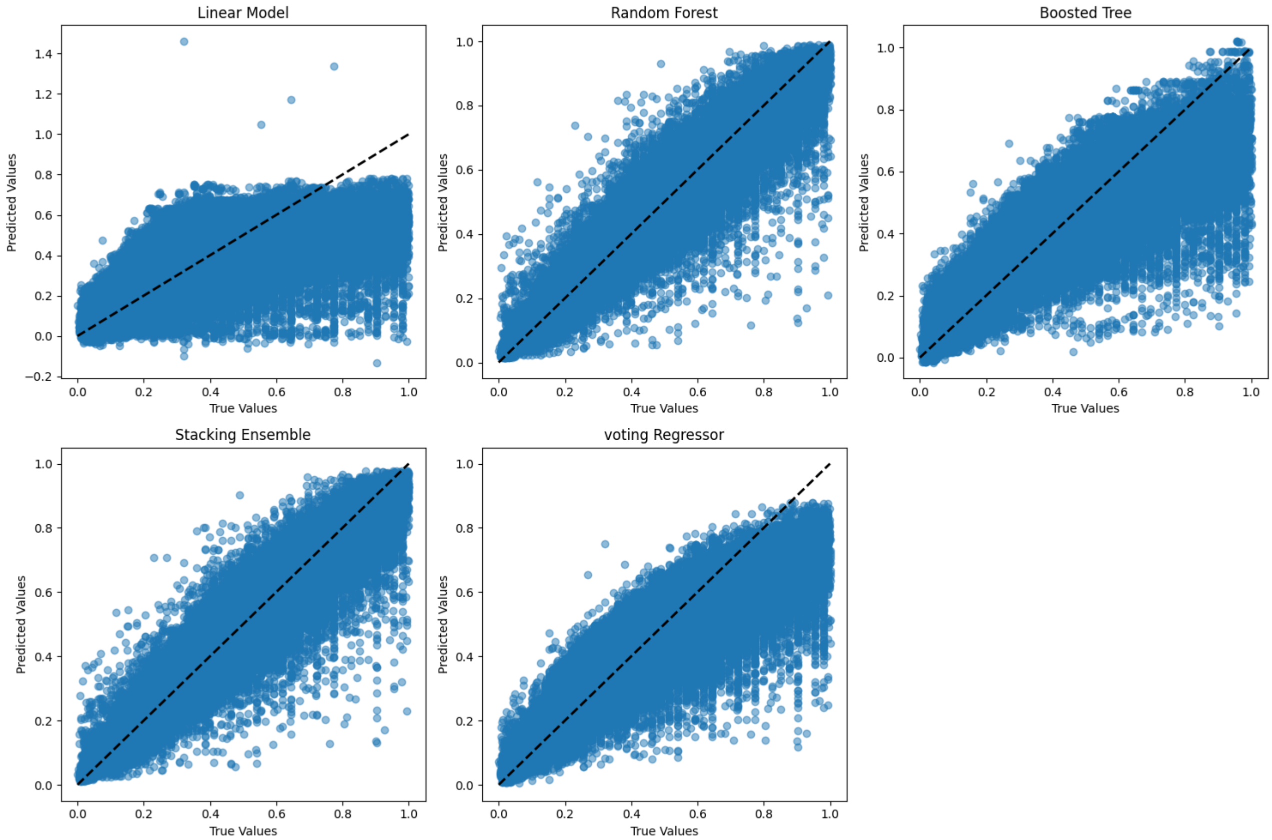


True vs Predicted Analysis

```python
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10))
axes = axes.flatten()

for ax, (name, model), model_pred in zip(axes[:-1], models, scores):
    ax.scatter(y_test, model_pred, label=name, alpha=0.5)
    ax.set_xlabel('True Values')
    ax.set_ylabel('Predicted Values')
    ax.set_title(name)
    ax.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--', lw=2)

fig.delaxes(axes[-1])
plt.tight_layout()
plt.show()
```



For the Linear Model, the plot shows a tendency to underestimate vehicle prices, particularly for higher-priced vehicles. The points are relatively close to the diagonal line for lower-priced vehicles but deviate significantly as the price increases. This suggests that the linear model may not capture the complexity of relationships between the features and the target variable.

In contrast, the Random Forest and Boosted Tree models demonstrate better overall fits compared to the linear model. The Random Forest model's points are closer to the diagonal line, with only some underestimation visible for higher-priced vehicles. The Boosted Tree model exhibits a slight curve in the plot, indicating that it overestimates lower-priced vehicles and underestimates higher-priced vehicles.

The Stacking Regressor model showcases a strong performance, with most points lying close to the diagonal line. However, it still displays some underestimation for higher-priced vehicles. This suggests that the combination of models in the ensemble technique improves the predictions but may still require further refinement.

Finally, the Voting Regressor model performs similarly to the Boosted Tree model. It displays a slight curve in the plot, tending to overestimate lower-priced vehicles and underestimate higher-priced vehicles.

Global and Local Explanations with SHAP

Standard SHAP values

```
shap.initjs()
```

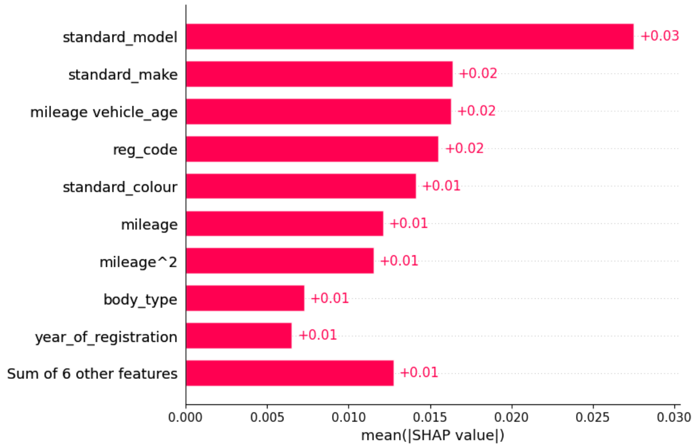
Standard SHAP values

```
[94] explainer = shap.Explainer(randomforest)
[95] shap_values = explainer(X[:1000])
[96] np.shape(shap_values.values)
     (1000, 10)
```

Global Explanation

Absolute Mean SHAP

```
shap.plots.bar(shap_values)
```



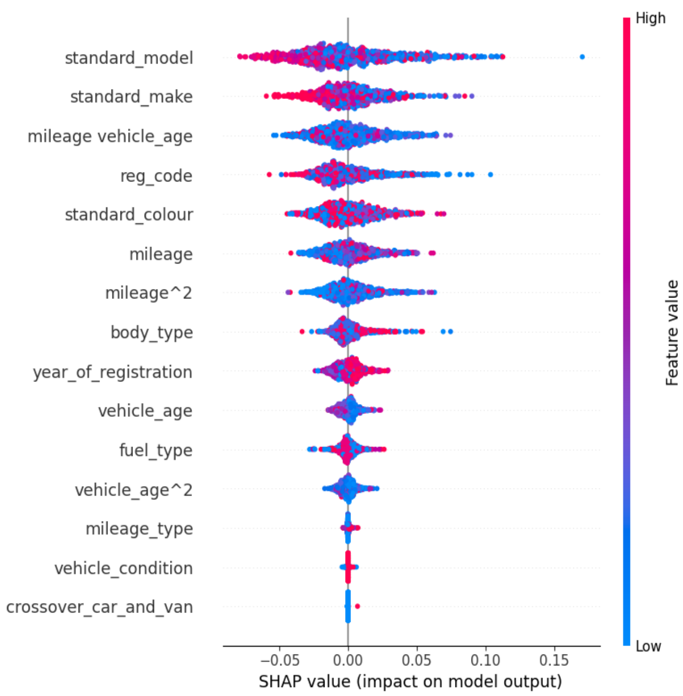
The mean absolute SHAP values provide a metric for feature importance, as they measure the significant contributions each feature makes to the model's predictions. By taking the absolute values, we prevent negative SHAP values from offsetting positive ones, ensuring that features with large positive or negative contributions are accurately represented.

Our analysis reveals that the standard model, standard make, and mileage vehicle_age are the most crucial features for predicting vehicle prices, with mean absolute SHAP values of +0.02 to +0.03. This indicates that the specific model and make of a vehicle, along with its mileage and age, have the strongest influence on its price. Other features, such as reg_code, standard_colour, and body_type, also contribute to the prediction, but to a lesser extent. Interestingly, the vehicle_condition and crossover_car_and_van features have the lowest mean absolute SHAP values, suggesting that they have minimal impact on the price prediction.

Summary plot

```
[112] shap.summary_plot(shap_values, X_transformed_copy[:1000])
```

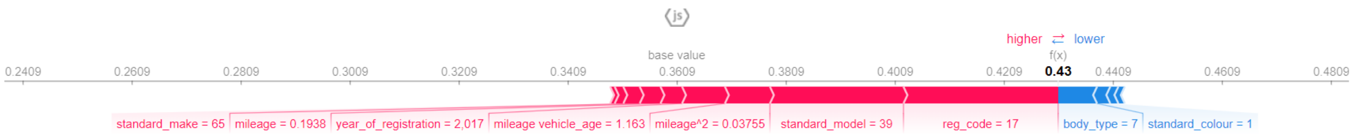
The plot demonstrates the importance of various features in determining vehicle prices, with standard_model, standard_make, and mileage vehicle_age being the most crucial factors. In the plot, red dots represent higher feature values, while blue dots represent lower feature values. Higher standard_model and standard_make values (red dots) generally lead to increased vehicle prices, whereas higher mileage vehicle_age values (red dots) result in lower prices. Features such as reg_code, standard_colour, and mileage have a moderate impact on price predictions, with red and blue dots indicating that their influence varies based on specific values. Meanwhile, body_type, year_of_registration, and vehicle_age have a smaller impact, as demonstrated by the less pronounced red and blue dots in the plot. Lastly, fuel_type, mileage_type, vehicle_condition, and crossover_car_and_van hold minimal influence on the model's predictions, as evidenced by the scarce red and blue dots in their respective rows.



Local Explanation

```
shap.initjs()
#Local explanations: SHAP values for a single instance
instance_index = 9  # Change this index to explain a different instance
shap_values_instance = shap_values[instance_index]

#SHAP force plot for the single instance
shap.plots.force(shap_values_instance)
```

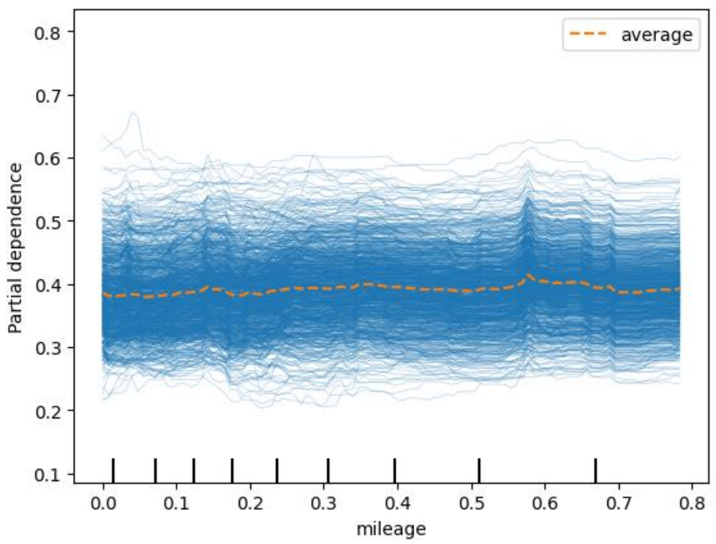


The features that are pushing the prediction higher (to the right) are shown in red color which are reg_code, standard_model, mileage^2, mileage vehicle_age, year_of_registration, mileage, and standard_make. The length of these arrows indicates the magnitude of the feature's contribution, and as its shown in the plot it is ordered, reg_code having the highest impact on it. The features that are pushing the prediction lower (to the left) are body_type and standard_make, the arrows length indicate the magnitude of it impact.

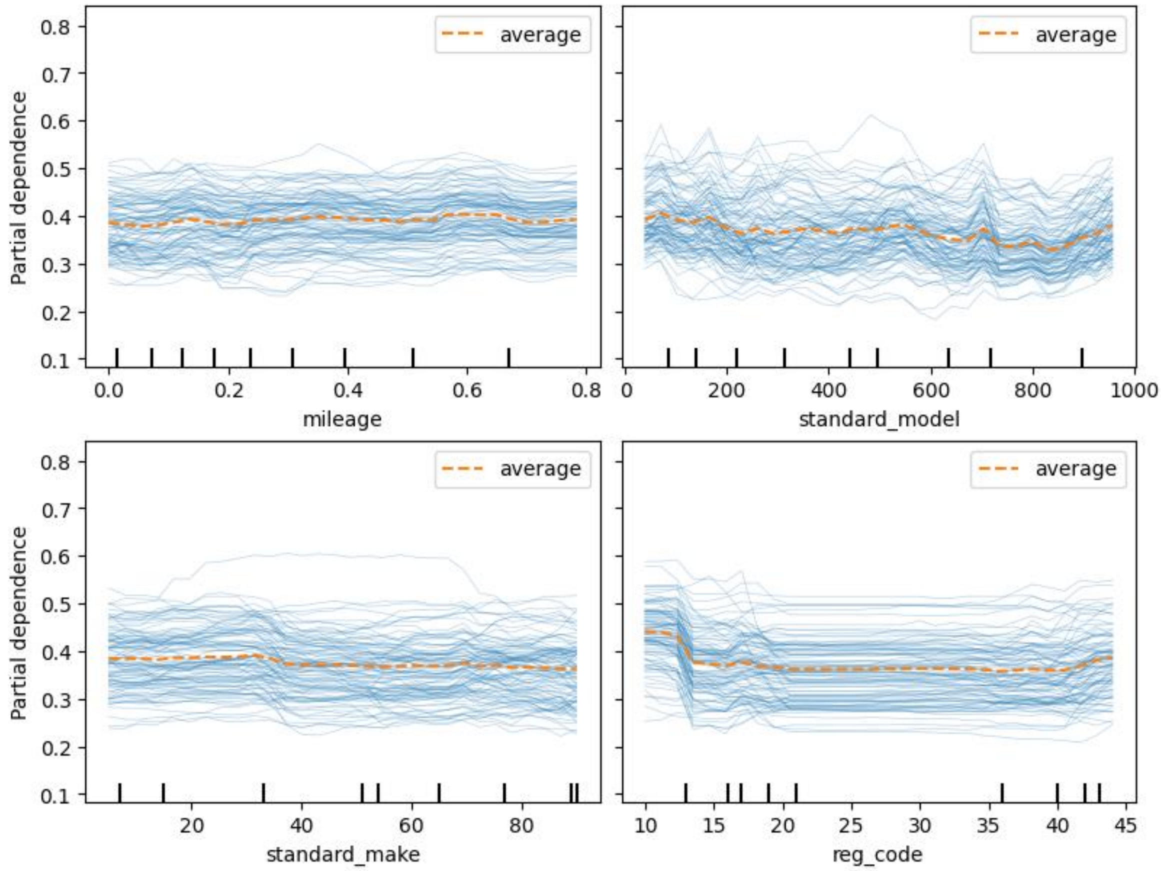
Partial Dependency Plots

```
# features we want to create the partial dependency plots for
features = ['mileage']

# Create the partial dependency plots
display = PartialDependenceDisplay.from_estimator(
    randomforest, X_transformed_copy, features, line_kw={"linewidth": 2}, n_jobs=-1
)
display.plot()
```

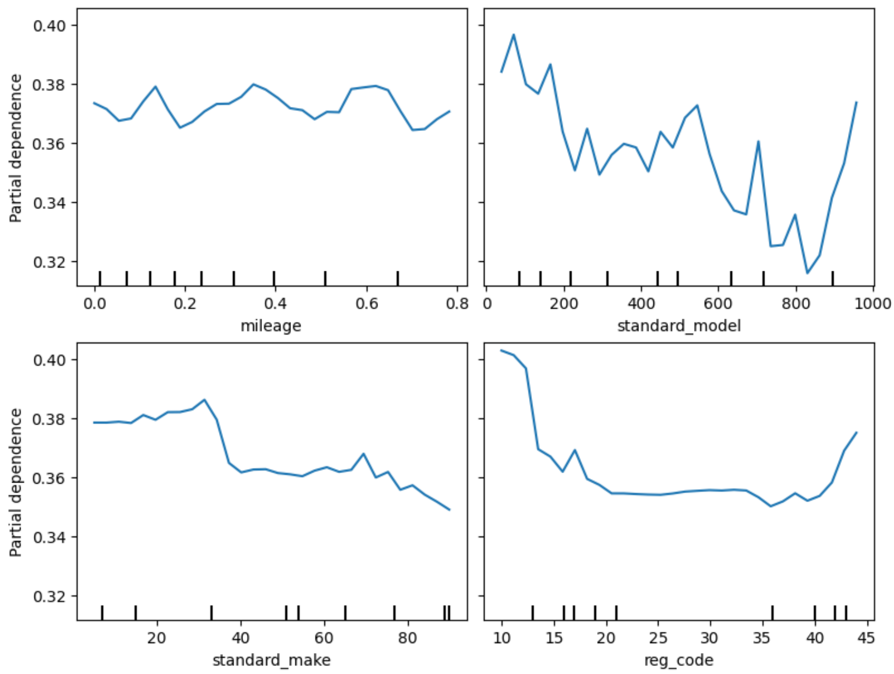



```
fig, ax = plt.subplots(figsize=(8,6), constrained_layout=True)
PartialDependenceDisplay.from_estimator(
    randomforest, X_transformed_copy, features=['mileage','standard_model', 'standard_make', 'reg_code'],
    kind='both',
    subsample=100, grid_resolution=30, n_jobs=2, random_state=0,
    ax=ax, n_cols=2
);
```




```
fig, ax = plt.subplots(figsize=(8,6), constrained_layout=True)
PartialDependenceDisplay.from_estimator(
    randomforest, X_transformed_copy, features=['mileage','standard_model', 'standard_make', 'reg_code'],
    subsample=100, grid_resolution=30, n_jobs=2, random_state=0,
    ax=ax, n_cols=2
);
```

For the standard model we can observe a non-linear and non-monotonic relationship, suggesting that different vehicle models have varying impacts on the predicted price. The plot shows multiple peaks and troughs, indicating that certain vehicle models are associated with higher predicted prices while others are linked to lower prices. This is consistent with the expectation that different vehicle models have different market values, depending on factors such as brand reputation, performance, and luxury features.



```
fig, ax = plt.subplots(figsize=(8,6), constrained_layout=True)
PartialDependenceDisplay.from_estimator(
    randomforest, X_transformed_copy, features=['standard_colour','body_type', 'fuel_type'], grid_resolution=30, n_jobs=-1, random_state=0,
    ax=ax, n_cols=2
);
```

The plot for 'standard_colour', shows a spike in the upward direction which could be the reason that some color cars are the most expensive, while the downward spike shows that some are the least expensive and we can clearly see the impact of it in the plot. 'Body_type' shows a downward curve which might indicates that some body type has lower prices. However, that the differences between the fuel types are relatively small.