1. Data Processing for Machine Learning

Dealing with missing values: - I queried the dataset to know the latest year which is 2020, so I filled the missing values for **Reg code** and **year of registration** with 2021. I converted the values on the reg code column to numeric, and for every row where vehicle condition equals new, I changed it to 21 and also for every row where reg code equals 21, I set year as 2021.

```
#Converting the values on the reg_code column to numeric
new_ads['reg_code'] = pd.to_numeric(new_ads['reg_code'], errors= 'coerce')
#Getting all the rows where vehicle condition column = NEW
new_ads.loc[(new_ads['vehicle_condition'] == 'NEW' ) &
            (new_ads['reg_code'].isnull()), 'reg_code'] = 21
new_ads['year_of_registration'] = pd.to_numeric(new_ads['year_of_registration'], errors= 'coerce')
new_ads.loc[(new_ads['vehicle_condition'] == 'NEW' ) &
            (new_ads['year_of_registration'].isnull()), 'year_of_registration'] = 2021.0
```

After reducing missing values for reg code and year of registration to the minimum by filling up manually, I dropped all missing values, using the **df.dropna** command to drop all missing values.

```
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
fuel_type               601
```

```
mileage                 127
reg_code               3523
standard_colour        5378
standard_make             0
standard_model            0
vehicle_condition         0
year_of_registration   2062
price                     0
body_type               837
crossover_car_and_van     0
fuel_type               601
```

```
new_ads1.isnull().sum()
```

```
mileage                 0
standard_make           0
vehicle_condition       0
year_of_registration    0
price                   0
body_type               0
dtype: int64
```

Dealing with outliers: - I calculated the first quartile (Mil1) and third quartile (Mil2) of the column 'mileage', then I got the Interquartile range (IQR). Also, calculated the first quartile (Price1) and the third quartile (Price2) of the column 'price', then I got the Interquartile range(IQR). I used this method to detect and potentially address outliers based on the distribution of data within the interquartile range.

```
[22] #FOR PRICE WE HAVE
     Price1 = new_ads0['price'].quantile(0.25)
     Price2 = new_ads0['price'].quantile(0.75)
     #interquantile range
     IQR = Price1 - Price2
     low_brac = Price1 - 1.5 * IQR
     high_brac = Price2 + 1.5 * IQR
     outliers = new_ads0[(new_ads0['price']<low_brac) | (new_ads0['price']>high_brac)]
     outliers_mask = (new_ads0['price'] < low_brac) | (new_ads0['price'] > high_brac)
     no_outliers = new_ads0[~outliers_mask]
```

Data Splitting: - For my X and Y split, I used 'price' as my target variable on Y axis and other features as predictors on X-axis, also splitting my data into training, testing and validation parts, I used 20% of my data as testing set and 80% of my dataset for training set. Also using my random state as 42 for reproducibility.

```
[36] X = df_encoded.drop('price', axis=1)
     y = df_encoded['price']
```

```
[37] X_train, X_val_test, y_train, y_val_test = train_test_split(X, y, test_size=0.2, random_state=42)

     X_val, X_test, y_val, y_test = train_test_split(X_val_test, y_val_test, test_size=0.2, random_state
```

Categorical Encoding using Target Encoder: - I used target encoding to combine and encode categorical features like "standard_make" and "body_type" also. I fitted the encoder on the training data and then transform both the training, validation, and test sets using the fitted encoder. This ensures that the encoding is consistent across all sets.

```
encoder = TargetEncoder()
encoder.fit(X_train, y_train)
X_train_encoded = encoder.transform(X_train)
X_val_encoded = encoder.transform(X_val)
X_test_encoded = encoder.transform(X_test)
X_test_encoded.head(10)
```

	mileage	standard_make	vehicle_condition	year_of_registration	body_type	car_year
108498	26853.0	21563.477980	15562.761679	2017.0	16110.815962	4.0
242685	34996.0	20002.986108	15562.761679	2015.0	31892.102379	6.0
327037	77000.0	20002.986108	15562.761679	2015.0	19157.034703	6.0

2. Feature Engineering: -

New Feature: - I created a new feature called the 'car year', using domain knowledge, I used 2021 as the current year for recent cars then I deducted the 'year of registration' from recent year (2021).

```
[33] Recent_yr = 2021
     new_ads1['car_year'] = Recent_yr - new_ads1['year_of_registration']
     new_ads1['car_year']
     new_ads1.head(10)
```

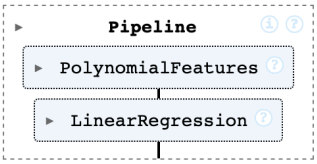
	mileage	standard_make	vehicle_condition	year_of_registration	price	body_type	car_year
0	0.0	Volvo	NEW	2021.0	73970	SUV	0.0
1	108230.0	Jaguar	USED	2011.0	7000	Saloon	10.0
2	7800.0	SKODA	USED	2017.0	14000	SUV	4.0

Polynomial Features: - I created a pipeline that performs polynomial regression using a degree of 2, that is simply squaring existing features by 2 and multiplying to create new features. After which the linear regression model will be trained on this new features, this allows the model to capture non-linear relationships between the original features and target variable.

```
[95] poly_reg = Pipeline(
         steps=[
             ('poly', PolynomialFeatures(2, include_bias=False)),
             ('est', LinearRegression())
         ]
     )
```

After that I fitted the polynomial regression pipeline on my training data

```
[105] poly_reg.fit(X_train_c, y_c_train)
```



I proceeded to perform cross-validation to evaluate the performance of the pipeline using negative mean absolute error (MAE).

```
eval_results = cross_validate(
    poly_reg, X_train_c, y_c_train, cv=5,
    scoring='neg_mean_absolute_error',
    return_train_score=True
)
```

These are the results I got before applying the polynomial regression

```
-eval_results['test_score'].mean(), eval_results['test_score'].std()
```

```
(4531.002101291481, 32.58775836875194)
```

```
[ ] -eval_results['train_score'].mean(), eval_results['train_score'].std()
```

```
(4530.415885369286, 11.862779914653286)
```

After applying the polynomial regression I got these results

```
[ ] -eval_results['test_score'].mean(), eval_results['test_score'].std()
```

```
(4392.145779317179, 95.16163448721969)
```

```
[ ] -eval_results['train_score'].mean(), eval_results['train_score'].std()
```

```
(4393.421028554289, 88.42341896131245)
```

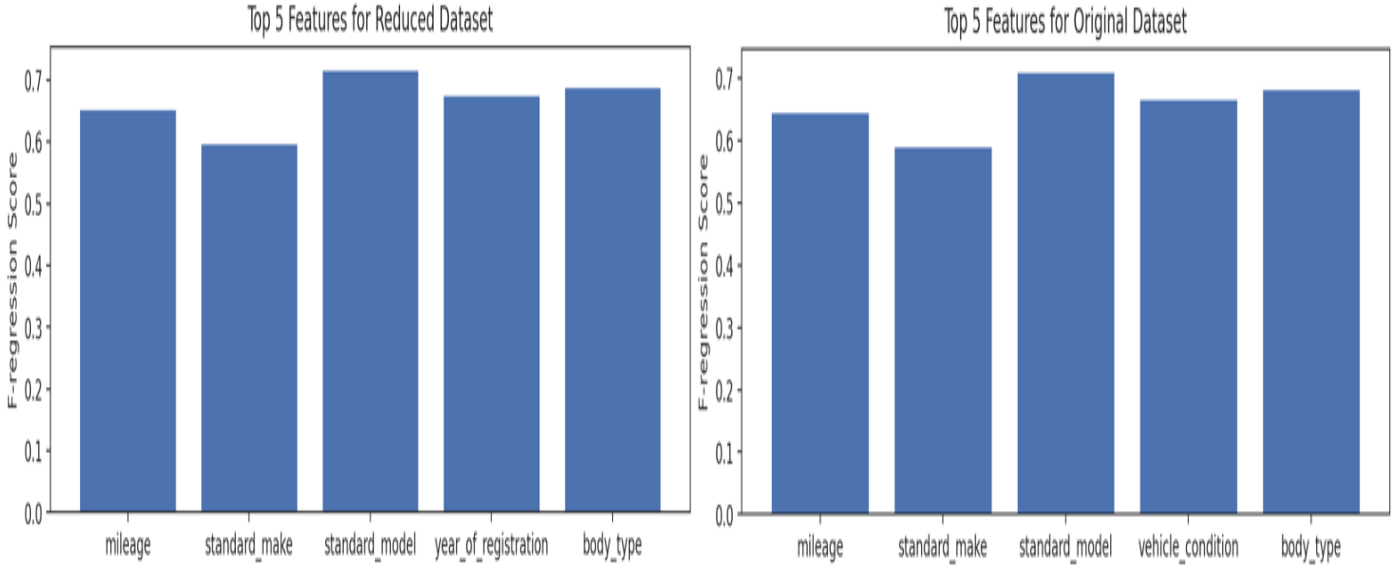
The mean test score with polynomial regression is 4392.15 is lower than the mean test score without it 4531.00, also the standard deviation with polynomial regression 95.16 is significantly higher than without polynomial regression 32.59. These results show that applying polynomial regression might have slightly led to slightly worse performance and possibly overfitting. so I did not apply polynomial regression to my dataset.

3.1. Feature Selection

Automated feature Selection: - Using domain knowledge I dropped features like **Public_reference**, **Reg_code**, **Standard_colour**, **crossover_car_and_van** because they don't have a direct correlation with price. After that I named it my "Reduced Dataset". I performed Automated feature selection on both the Reduced dataset and original dataset. I used the **selectKBest** method and **F_regression** method to select the top 5 features based on their F-values which is to measure the importance of the features.

```
[ ]  selector = SelectKBest(f_regression, k=5).fit(X_train_c, y_c_train)
     X_c_ext = selector.transform(X_train_c)
     selector_v = SelectKBest(f_regression, k=5).fit(X_train_v, y_v_train)
     X_v_orig = selector_v.transform(X_train_v)
```

The original dataset refers to the dataset with all features while the reduced dataset refers to the augmented dataset



From the chart we can see the difference in features selected by the SelectKbest. After that I created an Instance of a Linear regression model and fitted it to both the reduced and original data sets then calculated the mean and standard deviation of the cross validated scores to give an estimate of the model's performance. I got a mean score of 0.66 and a standard deviation of 0.04 for the reduced dataset, this is a lower score compared to the original dataset of mean 0.67 and standard deviation 0.04. I proceeded with all features since I got a better score than reduced features.

```
[40] model_v = LinearRegression().fit(X_c_orig, y_c_train)
     scores_v = cross_val_score(model_v, X_c_orig, y_c_train)
     scores_v.mean(), scores_v.std()
```

```
     (0.6673278447950329, 0.03998942265849398)
```

```
[41] model = LinearRegression().fit(X_v_ext, y_v_train)
     scores = cross_val_score(model, X_v_ext, y_v_train)
     scores.mean(), scores.std()
```

```
     (0.6600859470028586, 0.03988550965330926)
```

3.2. Dimensionality Reduction: -

I used a standard scaler to fit the mean and standard deviation for each feature to standardize it and make it more suitable for machine learning after that I created a Principal Component Analysis (PCA) instance at random state of 42 for reproducibility, this is to reduce the dimensionality of my standardized dataset.

```
[57] scaler = StandardScaler()
     X_scaled = scaler.fit_transform(X_train_c)
```

```
[58] pca = PCA(random_state=42)
     pca.fit(X_scaled)
```

```
PCA
PCA(random_state=42)
```

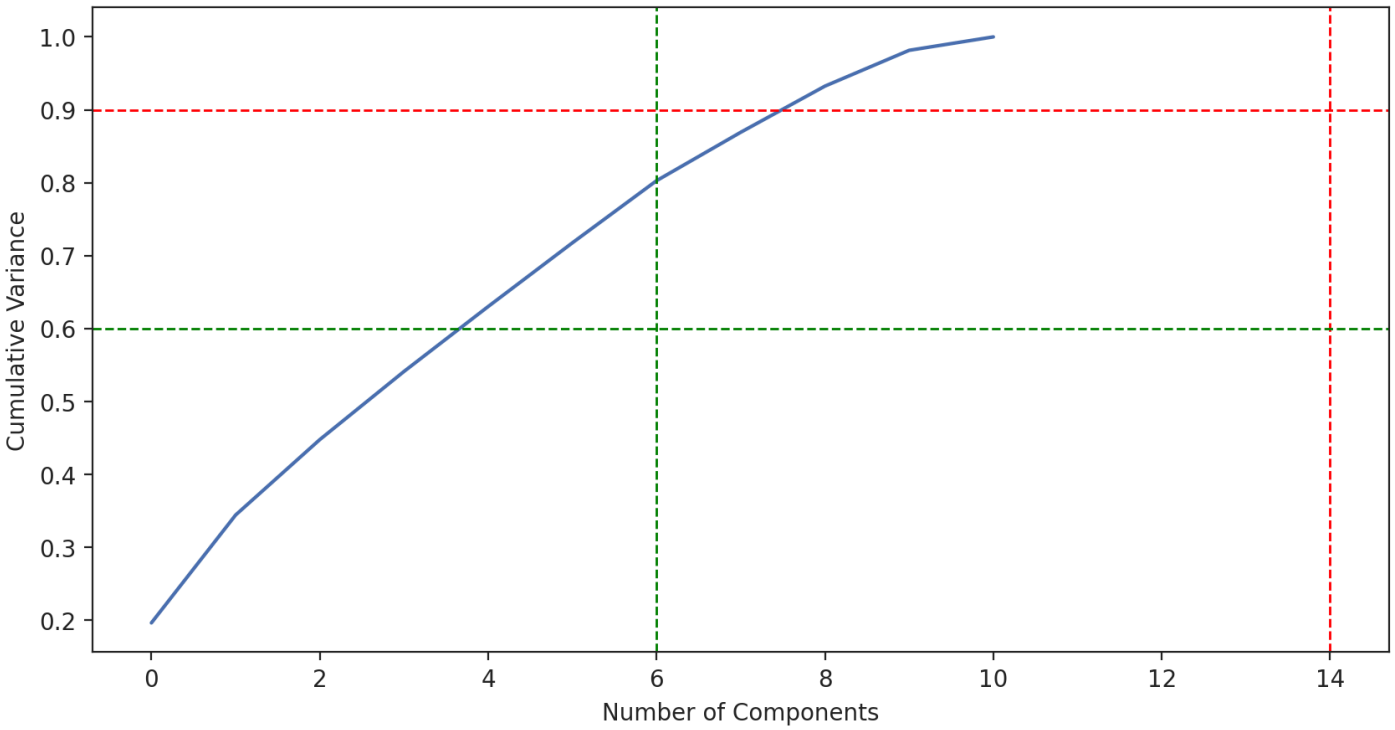
After that I accessed the fitted PCA object (**pca.explained_variance_ratio**) , calculated and displayed the cumulative variance ratio **(cvr)** and explained variance ratio **(evr)** for the principal components. The **cumsum()** calculates the cumulative sum of the evr across all principal components. The cvr contains array where elements represents the total variance explained by the corresponding number of principal components.

```
cvr = np.cumsum(pca.explained_variance_ratio_)
evr = pca.explained_variance_ratio_

pca_df = pd.DataFrame()
pca_df['Cumulative Variance Ratio'] = cvr
pca_df['Explained Variance Ratio'] = evr
display(pca_df.head(25))
```

	Cumulative Variance Ratio	Explained Variance Ratio
0	0.196481	0.196481
1	0.344381	0.147899

After that I plotted the number of components against the cumulative variance, with red and green lines illustrating vertical and horizontal limits.



6 Principal components explain 80% variance in the data

- This implies that the 6 components alone are capturing a significant portion of the overall variability present in the original dataset. capturing 80% of the variance with 6 components is considered to be a good result.
- My goal is to significantly reduce the dimensionality of the data while retaining many information as possible, I captured 80% of the variance with only six components. This shows a successful reduction in dimensionality and still preserved a significant amount of the original information.

I applied PCA on both original and reduced datasets, my original dataset gave me better PCA results and CVR results compared to my reduced dataset, so I proceeded with the original dataset to complete my machine learning processes with the complete features.

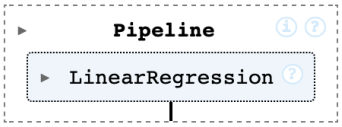
4. **Model Building**

A Linear Model: -

I created an instance of Linear regression model and fitted it to the train set.

```
lr1d = Pipeline(
    steps=[
        ('est', LinearRegression())
    ]
)
```

```
[89] lr1d.fit(X_train_c, y_c_train)
```



I used the predict method of the linear regression pipeline to generate predictions for the test data, I calculated the mean absolute error, mean squared error and R2 score.

```
[92] lr1d_pred = lr1d.predict(X_test_c)
     lr1d_mae = mean_absolute_error(y_c_test,lr1d_pred)
     lr1d_mse = mean_squared_error(y_c_test, lr1d_pred)
     lr1d_r2 = r2_score(y_c_test,lr1d_pred)

     print("Mean Absolute Error (MAE):", lr1d_mae)
     print("Mean Squared Error (MSE):", lr1d_mse)
     print("R-squared (R2) Score:", lr1d_r2)
```

```
Mean Absolute Error (MAE): 4679.6214447578095
Mean Squared Error (MSE): 488668947.84149885
R-squared (R2) Score: 0.4285219353549029
```

I also did the same for the train to compare both results

```
[91] lr1d_pred1 = lr1d.predict(X_train_c)
     lr1d_mae1 = mean_absolute_error(y_c_train, lr1d_pred1)
     lr1d_mse1 = mean_squared_error(y_c_train, lr1d_pred1)
     lr1d_r21 = r2_score(y_c_train, lr1d_pred1)

     print("Mean Absolute Error (MAE):", lr1d_mae1)
     print("Mean Squared Error (MSE):", lr1d_mse1)
     print("R-squared (R2) Score:", lr1d_r21)
```

```
Mean Absolute Error (MAE): 4530.641177334369
Mean Squared Error (MSE): 140111542.69541645
R-squared (R2) Score: 0.6681014637970937
```

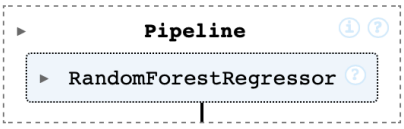
These scores show that the Linear Regression model has performed poorly having a poor R2 score of 0.67 on train and 0.42 on test. This indicates that the model's performance on the test set is slightly worse compared to the training set. Increase in MAE and MSE score and decrease in R2 score implies that the model is overfitting to the training data and not generalizing well to the test data.

4.2 A Random Forest

I created an instance of Random Forest Regressor model and fitted it to the train set. I also did manual grid search by simply tweaking the number of estimators **(n_estimators)**, I tweaked from 80 to 200 and settled with 100 as I got better scores and my model performed better with it.

```
rf = Pipeline(
    steps=[
        ('est',RandomForestRegressor(n_estimators=100, random_state=42))
    ]
)

rf.fit(X_train_c, y_c_train)
```



I used the predict method of the Random Forest Regressor pipeline to generate predictions for the test data, I calculated the mean absolute error, mean squared error and R2 score.

```
[116] rf_pred = rf.predict(X_test_c)
      rf_mae = mean_absolute_error(y_c_test,rf_pred)
      rf_mse = mean_squared_error(y_c_test, rf_pred)
      rf_r2 = r2_score(y_c_test,rf_pred)
      print("Mean Absolute Error (MAE):", rf_mae)
      print("Mean Squared Error (MSE):", rf_mse)
      print("R-squared (R2) Score:", rf_r2)
```

```
Mean Absolute Error (MAE): 1880.7278840479787
Mean Squared Error (MSE): 126504158.22136645
R-squared (R2) Score: 0.8520586343183145
```

I also did the same for the train to compare both results.

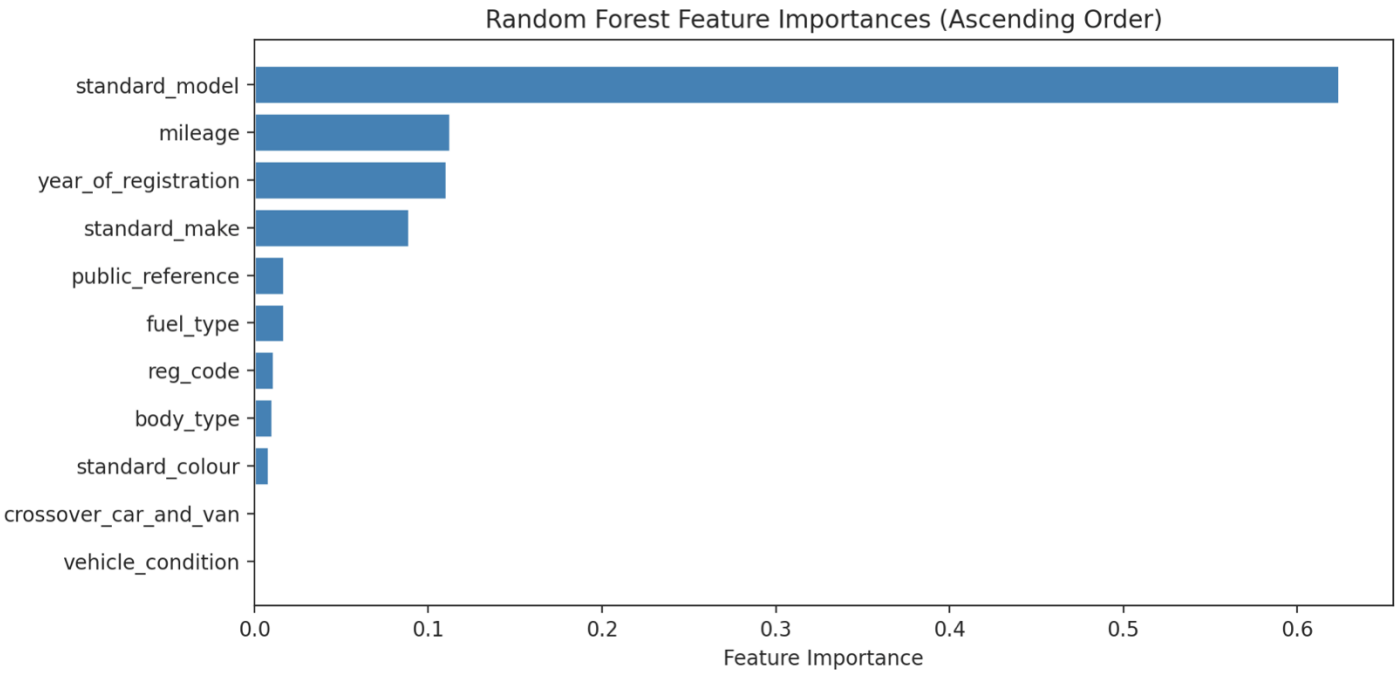
```
[136] rf_pred0 = rf.predict(X_train_c)
      rf_mae0 = mean_absolute_error(y_c_train,rf_pred0)
      rf_mse0 = mean_squared_error(y_c_train, rf_pred0)
      rf_r20 = r2_score(y_c_train,rf_pred0)

      print("Mean Absolute Error (MAE):", rf_mae0)
      print("Mean Squared Error (MSE):", rf_mse0)
      print("R-squared (R2) Score:", rf_r20)
```

```
Mean Absolute Error (MAE): 695.3267756778973
Mean Squared Error (MSE): 6404902.203278354
R-squared (R2) Score: 0.9848279618873942
```

These scores (higher R2 score and Lower values for MAE and MSE) indicate better performance compared to the linear model. Although there is some overfitting on the model considering the lower R-squared score of 0.85 and higher values of MAE and MSE for test set as compared to 0.98 for train set. These figures show that the model performed good on the training set but performs poorly on unseen dataset (test set), this simply means the model is overfitting

After that I got the feature importance using the **rf.feature_importance** and plotted a bar chart of Random Forest feature importance ranking them with how important they are.
Fr



From the chart we can see that Standard model is the most important feature in determining car prices, followed by mileage, year of registration and standard make.

4.3. A Boosted Tree

I created an instance of Gradient Boosting Regressor model and fitted it to the train set. I also manually tweak the number of trees "**n_estimators**" from 200, 150, 100 etc, to find the best number to give best results. 200 gave me a better result so I used 200 as number of trees.

```
[90] from sklearn.ensemble import GradientBoostingRegressor
     params = {
         'learning_rate': 0.1,
         'n_estimators': 200,
         'max_depth': 3,
         'min_samples_split': 2,
         'min_samples_leaf': 1
     }
     gbr = GradientBoostingRegressor(**params)
     gbr.fit(X_train_c, y_c_train)
     gbr_pred = gbr.predict(X_test_c)
```

I used the predict method of the Gradient Boosting Regressor to generate predictions for the test data, I calculated the mean absolute error, mean squared error and R2 score.

```
[91] gbr_mae = mean_absolute_error(y_c_test,gbr_pred)
     gbr_mse = mean_squared_error(y_c_test, gbr_pred)
     gbr_r2 = r2_score(y_c_test,gbr_pred)
     print("Mean Absolute Error (MAE):", gbr_mae)
     print("Mean Squared Error (MSE):", gbr_mse)
     print("R-squared (R2) Score:", gbr_r2)

     Mean Absolute Error (MAE): 2756.8962348107602
     Mean Squared Error (MSE): 106982618.068178
     R-squared (R2) Score: 0.8748882657792729
```

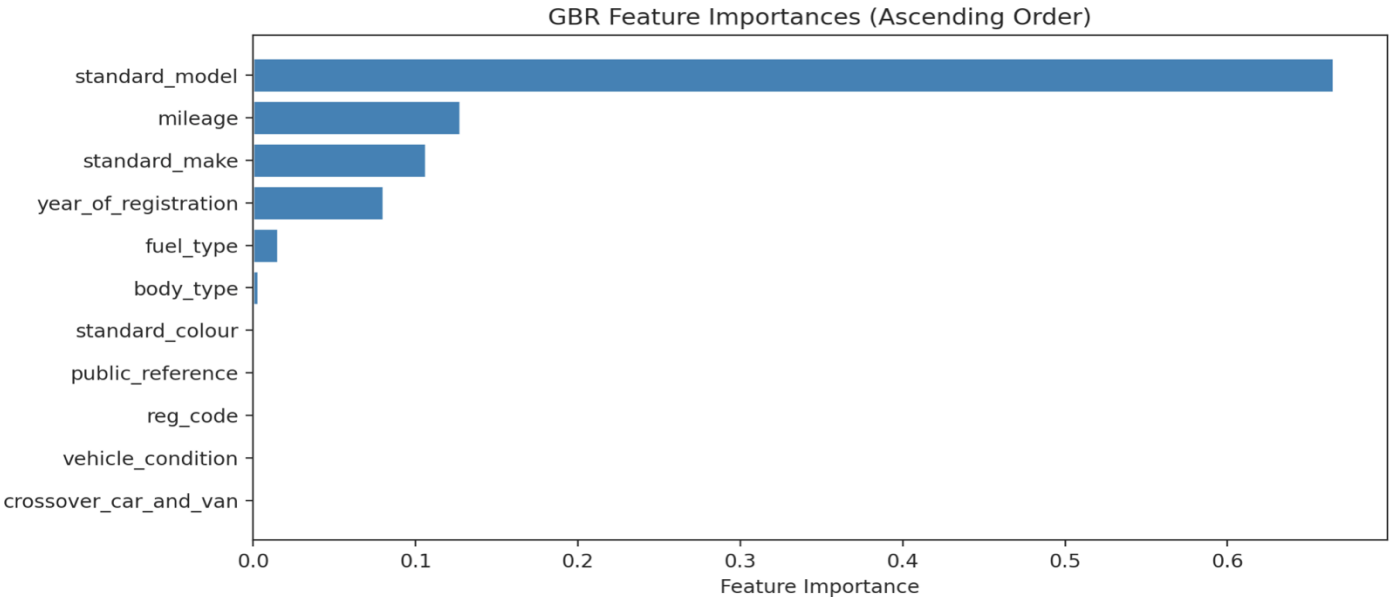
I also did the same for the train to compare both results.

```
gbr_pred1 = gbr.predict(X_train_c)
gbr_mae1 = mean_absolute_error(y_c_train,gbr_pred1)
gbr_mse1 = mean_squared_error(y_c_train, gbr_pred1)
gbr_r21 = r2_score(y_c_train,gbr_pred1)
print("Mean Absolute Error (MAE):", gbr_mae1)
print("Mean Squared Error (MSE):", gbr_mse1)
print("R-squared (R2) Score:", gbr_r21)

Mean Absolute Error (MAE): 2697.9545917136397
Mean Squared Error (MSE): 39392172.78809253
R-squared (R2) Score: 0.9066871705592349
```

These scores (higher R2 score and Lower values for MAE and MSE) indicate better performance compared to the other models. Although there is some overfitting on the model considering the lower R-squared score of 0.87 and higher values of MAE and MSE for test set as compared to 0.90 for train set. This increase in MAE and Decrease in MSE on test set signifies overfitting.

After that I got the feature importance using the **gbr.feature_importance** and plotted a bar chart of Gradient Boosting Regressor feature importance ranking them with how important they are.



4.4 A Voter

I created a Voting Regressor ensemble with my two best performing models which are Gradient Boosting Regressor model and Random Forest model as its base model, the ensemble combines the predictions of these two models to make final predictions on new data.

```
[126] from sklearn.ensemble import VotingRegressor
      gbr_model = gbr
      rf_model = rf
      ensemble = VotingRegressor(
          [
              ("gbr", gbr_model),
              ("rf", rf_model)
          ]
      )
      ensemble.fit(X_train_c, y_c_train)
      ensemble
```

After creating a voter regression pipeline, I proceeded to check the metric scores and r2 scores and compare with other models. For train scores I got an R2 score of 0.96 and Mae of 1598, as compared to test score of 0.87 and Mae of 2117, this scores further shows overfitting by the Voter regression model.

```
Test Mean Absolute Error (MAE): 2117.141102090374
Test Mean Squared Error (MSE): 110928883.16823645
Test R-squared (R2) Score: 0.8702732724347615

Train Mean Absolute Error (MAE): 1598.1851741813782
Train Mean Squared Error (MSE): 16473896.806491105
Train R-squared (R2) Score: 0.9609763611873285
```

Stacker Ensemble

I used a Stack Regressor to combine predictions from my best performing models i.e., Random Forest Regressor and Gradient Boosting Regressor to form a base model, after that I fitted the stacking model and made predictions.

```
[105] from sklearn.ensemble import StackingRegressor
      base_models = [('model1', rf), ('model2', gbr)]
      meta_model = GradientBoostingRegressor()
      stacking_model = StackingRegressor(estimators=base_models, final_estimator=meta_model)
      stacking_model.fit(X_train_c, y_c_train)
      predictionstack = stacking_model.predict(X_test_c)
```

After creating a voter regression pipeline, I proceeded to check the metric scores and r2 scores and compare with other models. For train scores I got an R2 score of 0.96 and Mae of 1598, as compared to test score of 0.87 and Mae of 2117, this scores further shows overfitting by the Voter regression model.

The scores for Test set: -

```
Test Mean Absolute Error (MAE): 1878.960312722596
Test Mean Squared Error (MSE): 108339027.00375606
Test R-squared (R2) Score: 0.8733020017925894
```

The scores for Train set: -

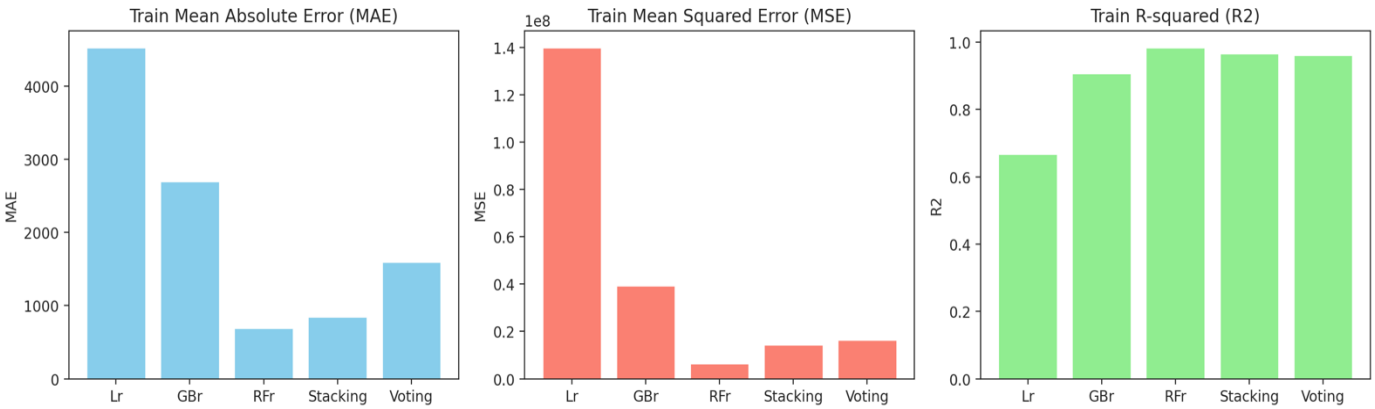
```
Train Mean Absolute Error (MAE): 849.3097063674082
Train Mean Squared Error (MSE): 14930657.207931742
Train R-squared (R2) Score: 0.9646320126341594
```

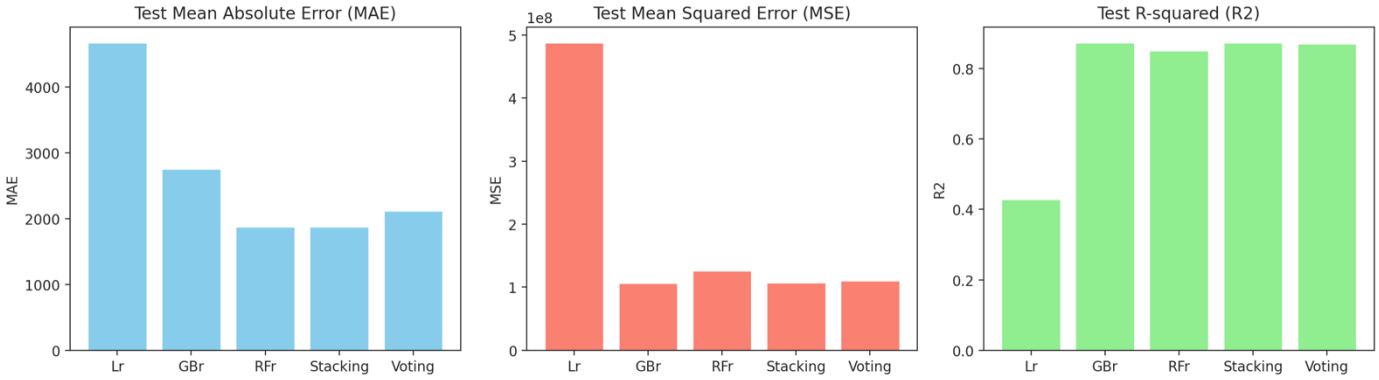
5.1 Overall Performance with Cross-Validation

	Model	TRAIN MAE	TRAIN MSE	TRAIN R2
0	Lr	4530.641177	1.401115e+08	0.668101
1	GBr	2697.954592	3.939217e+07	0.906687
2	RFr	695.326776	6.404902e+06	0.984828
3	Stacking_Model	849.838880	1.440121e+07	0.965886
4	Voting R	1598.185174	1.647390e+07	0.960976

	Model	TEST MAE	TEST MSE	TEST R2
0	Lr	4679.621445	4.886689e+08	0.428522
1	GBr	2756.645093	1.068432e+08	0.875051
2	RFr	1880.727884	1.265042e+08	0.852059
3	Stacking_Model	1874.395229	1.073197e+08	0.874494
4	Voting R	2116.432889	1.103720e+08	0.870925

These are my results from all 5 models, The Gradient Boosting Regressor has a better performance with less over fitting compared to the other models while the Linear Regression Model performed the worse on all 5 models.





I performed Cross validation for my best models, Gradient boosting and Random Forest

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import cross_val_predict
from sklearn import datasets
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error


models = {
    'Random Forest': RandomForestRegressor(),
    'Gradient Boosting': GradientBoostingRegressor()
}


predicted_values = {}
for name, model in models.items():
    predicted_values[name] = cross_val_predict(model, X_train_c, y_c_train, cv=10)
```

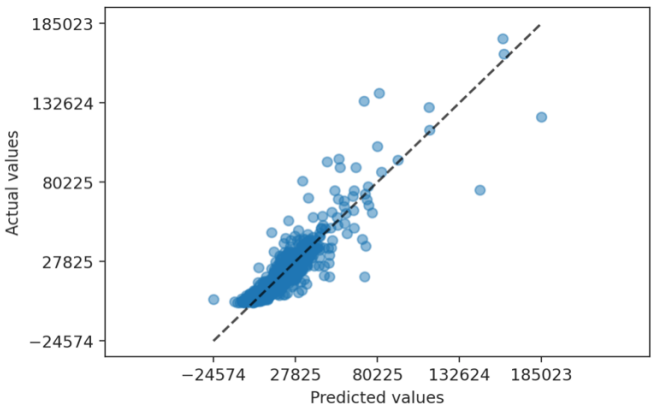
These are the results I got from performing cross validation

```
Evaluation metrics for Random Forest:
  Mean Squared Error: 42585900.99210761
  R2 Score: 0.8991218144468779
  Mean Absolute Error: 1816.8883225110837
Evaluation metrics for Gradient Boosting:
  Mean Squared Error: 59590926.44016923
  R2 Score: 0.8588400293367532
  Mean Absolute Error: 2939.333049299479
```

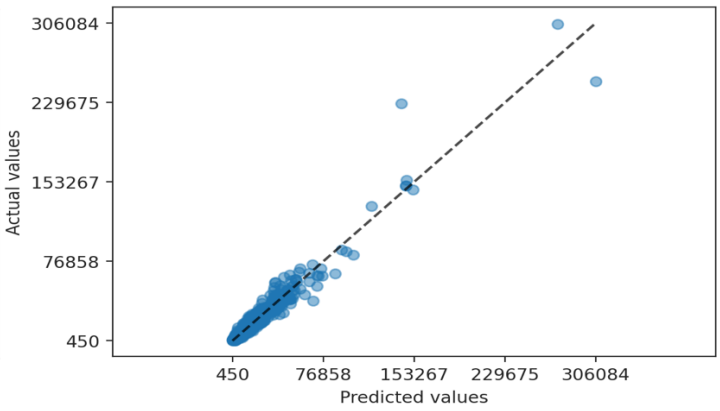
5.2 True vs Predicted Analysis for Combined Models.

This plot shows how well the model predictions align with the actual values.

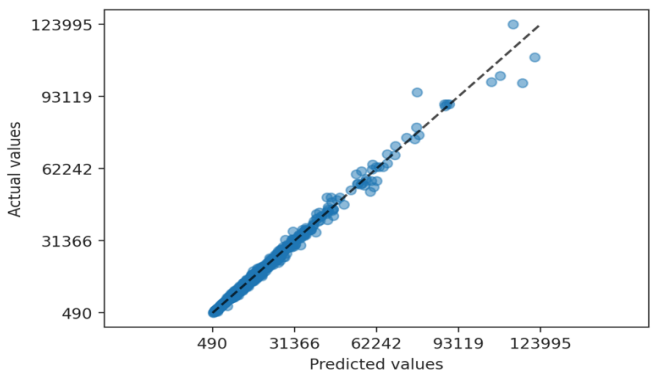
Actual vs Predicted for Linear Model.



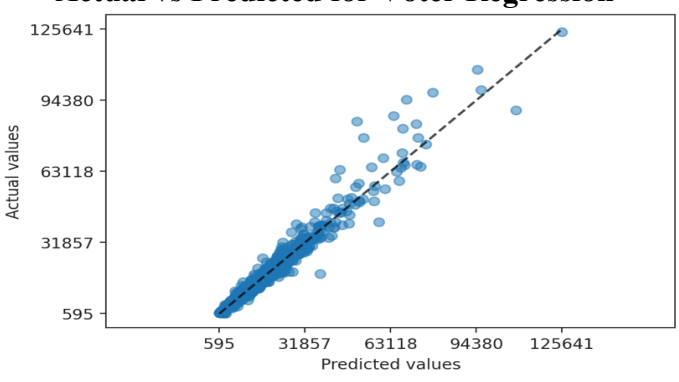
Actual vs Predicted for Gradient Boosting



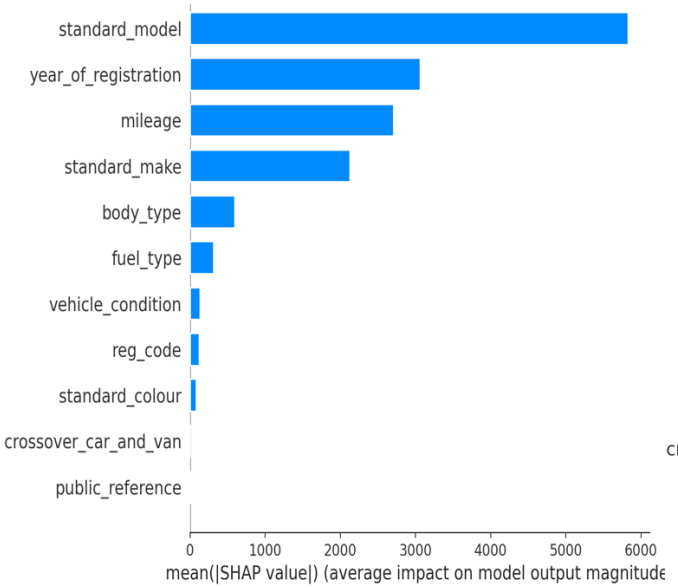
Actual vs Predicted for Random Forest Model

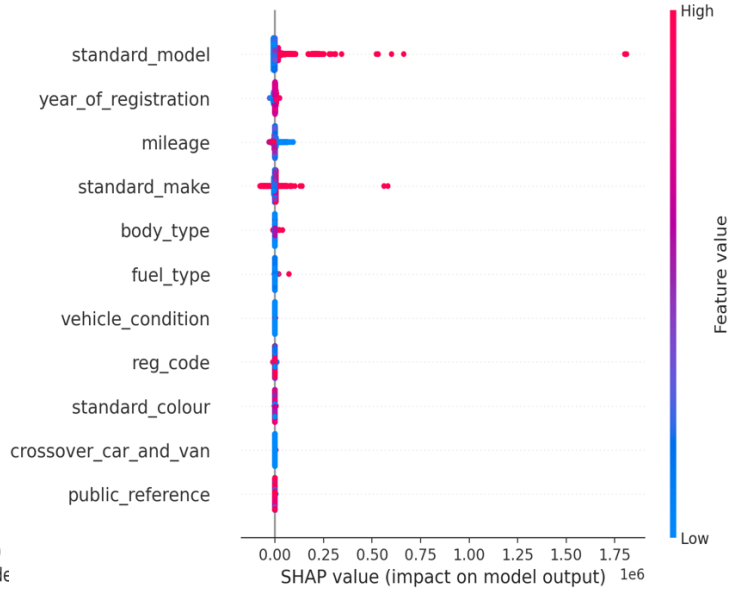


Actual vs Predicted for Voter Regression

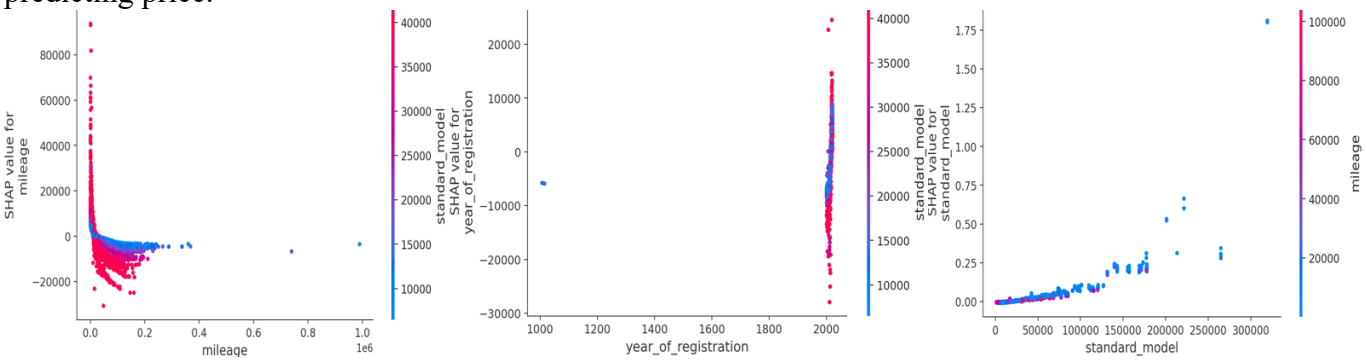


GLOBAL SHAP EXPLANATIONS

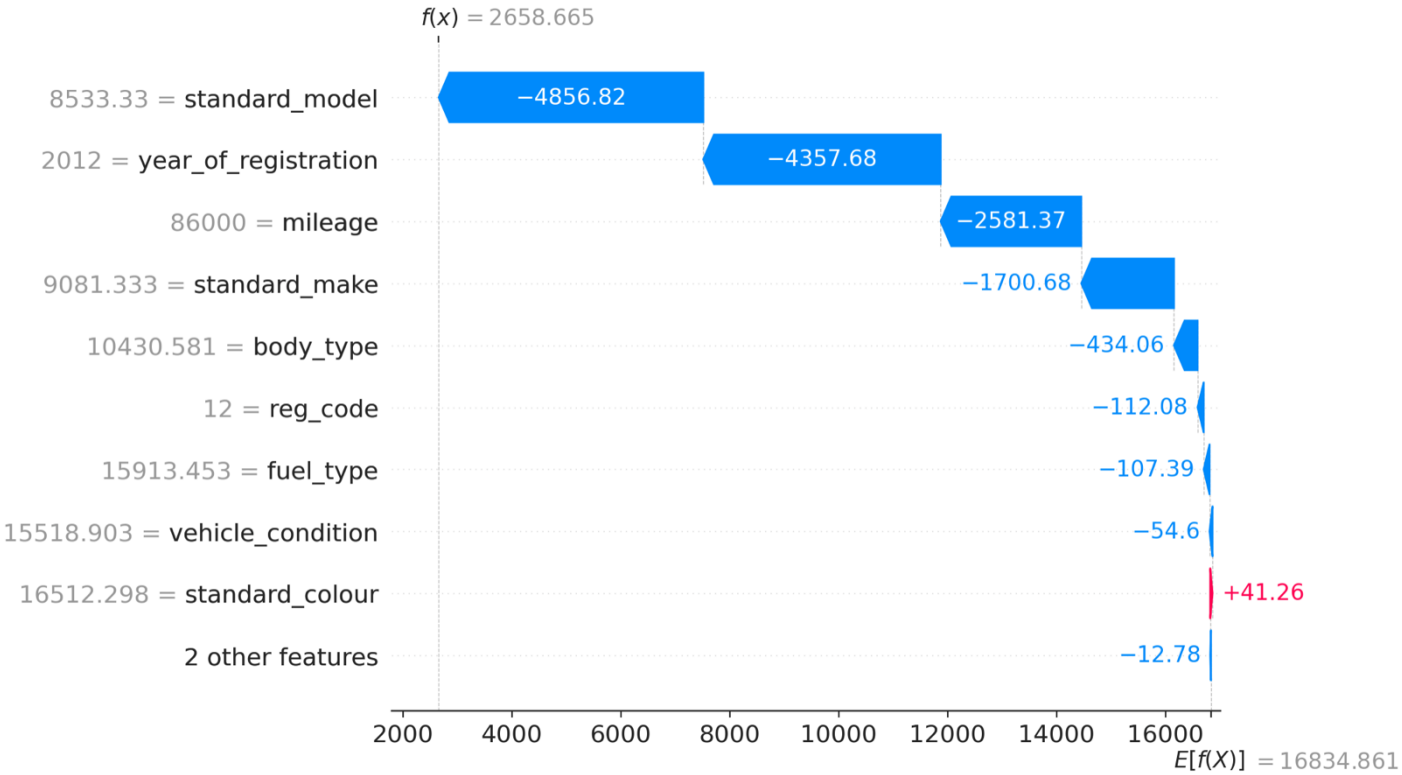




These plots show how individual features contribute to the model's prediction in a data point from the test data. The rightward bars indicates that the feature (e.g., standard model) increased the model's prediction from the base value. While the leftward bars (negative values) indicate that the feature decreased the model prediction from base value. From the plot we can see that Standard model has the strongest influence in predicting price.



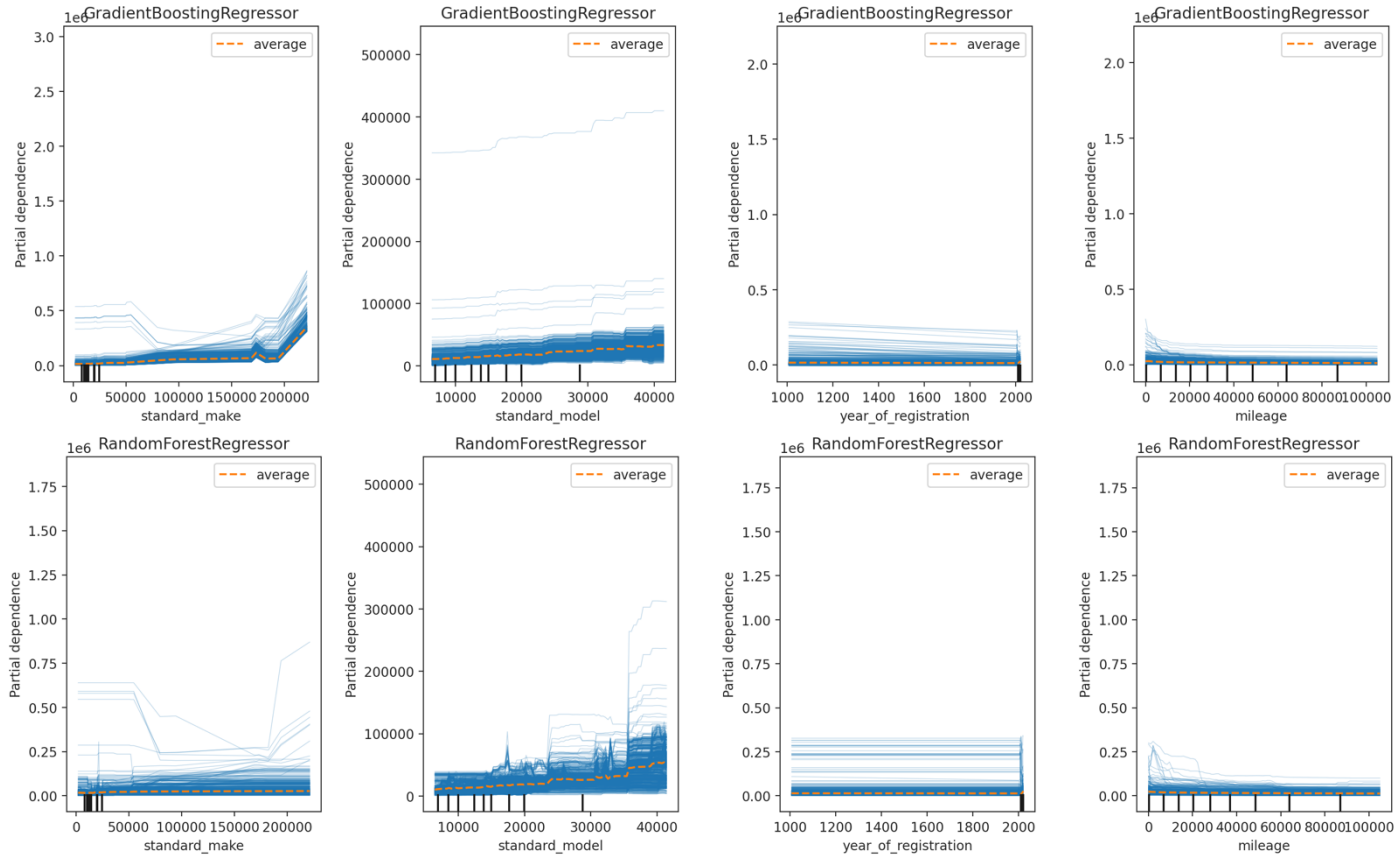
Shap dependence plots



This is a random Shap waterfall plot for instance index of 2.

PARTIAL DEPENDENCY PLOT

These are partial dependency plots for Gradient Boosting Regressor and Random Forest Regressor.



CONCLUSION: - These plot and results show that Gradient Boosting Regressor Model is the best performing models of all including the stack and ensemble, this is because it performs well on the train dataset and also on unseen dataset (test). It has better MAE, MSE and R2 scores on both test and train and also has less overfitting, Also we get to see the main 4 features that determine the price of cars.

PARTIAL DEPENDENCY PLOT FOR RANDOM FOREST REGRESSOR