Manchester Metropolitan University

Department of Computing and Mathematics

M.Sc. in Artificial Intelligence

Assessment:

AUTOTRADER CARPRICE PREDICTION

Module:

ADVANCED **MACHINE LEARNING**

6G7V0017_2223_1F

Semester 2: 2022/2023

Name: ADERIYE ABIOLA MIRACLE

Student ID: @22545807

Tutor: LUCIANO GERBER

ROCHELLE TAYLOR

Chapter 1: Data Processing for Machine Learning

1.0. I used this to generates a dataset summary to make sense of my data e.g correlations

```
[1] !pip install ydata-profiling
```

1.1. I mount directly from my Google Drive so I don't have to upload every time.

```
from google.colab import drive
drive.mount("/content/drive/", force_remount=True)
```

```
Mounted at /content/drive/
```

1.2 Here I imported my libraries for visualization, my analysis, sets plotting format, random seed (for reproducing my result), and imports 'ProfileReport' and 'StandardScaler'.

```
import random
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns
from ydata_profiling import ProfileReport
from sklearn.preprocessing import StandardScaler


%config InlineBackend.figure_format = 'retina'
sns.set(
    style='ticks',
    context='talk',
    font_scale=0.8,
    rc={'figure.figsize': (8,6)}
)

seed = 60
random.seed(seed)
np.random.seed(seed)
```

1.3. Here, I read my file from the Google Drive and displays the first few rows of the DataFrame.

```
df = pd.read_csv("https://drive.google.com/uc?export=download&id=124sfwX5V1RhsApypyDOwHYurGbifhzqW")
df.head()
```

	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
0	202006039777689	0.0	NaN	Grey	Volvo	XC90	NEW	NaN	73970	SUV	False	Petrol Plug-in Hybrid
1	202007020778260	108230.0	61	Blue	Jaguar	XF	USED	2011.0	7000	Saloon	False	Diesel
2	202007020778474	7800.0	17	Grey	SKODA	Yeti	USED	2017.0	14000	SUV	False	Petrol
3	202007080986776	45000.0	16	Brown	Vauxhall	Mokka	USED	2016.0	7995	Hatchback	False	Diesel
4	202007161321269	64000.0	64	Grey	Land Rover	Range Rover Sport	USED	2015.0	26995	SUV	False	Diesel

1.4. I checked the price of over 2 million to really know if they were real great cars or mere mistakes.

```
[ ] over_2m = df.query('price > 2e6')
    print(over_2m.shape)
    over_2m.head()
```

```
(16, 12)
```

	public_reference	mileage	reg_code	standard_colour	standard_make	standard_model	vehicle_condition	year_of_registration	price	body_type	crossov
51741	202002257718775	4400.0	14	Black	Bugatti	Veyron	USED	2014.0	2850000	Coupe	
64910	202006039766650	189.0	NaN	Black	McLaren	P1	USED	NaN	2695000	Coupe	
72681	202007010711087	475.0	15	Yellow	Ferrari	LaFerrari	USED	2015.0	2299950	Coupe	
94033	202007020778467	1900.0	18	White	Pagani	Huayra	USED	NaN	2400000	Convertible	
141833	202007050883898	87450.0	NaN	Red	Ferrari	250	USED	NaN	9999999	Coupe	

Chapter 2. Feature Engineering

2.0.I dropped public_reference, standard_model, and reg_code columns because they were too distinct, meaning unique/uncorrelated and represented already

2.1. I also dropped rows with a price and mileage of 9,999,999 because they were clearly placeholders/errors.

2.2.Moving further, I converted the mileage & crossover_car_and_van columns to float and string respectively. The reason I did this is to enable me work with the mileage more precisely and since ‘crossover’ is categorical, it makes more sense as string.

```
# 1. remove ppublic reference
clean_df = df.drop(['public_reference', 'standard_model', 'reg_code'], axis=1)

#2. remove price=9999999
clean_df = clean_df[clean_df['price'] != 9_999_999]

#3. remove mileage=9999999
clean_df = clean_df[clean_df['mileage'] != 999_999]
clean_df = clean_df[~clean_df['mileage'].isna()]

clean_df.loc[:, 'mileage'] = clean_df['mileage'].astype(float)
clean_df.loc[:, 'crossover_car_and_van'] = clean_df['crossover_car_and_van'].astype(str)

# Get the 'standard_colour' column from the given DataFrame
vehicle_colors = clean_df['standard_colour']
# Create a boolean mask to filter out missing values
na_mask = vehicle_colors.isna()
colors = vehicle_colors[~na_mask].tolist()

clean_df['standard_colour'] = clean_df['standard_colour'].map(lambda x: random.choice(colors) if pd.isna(x) else x)

clean_df['year_of_registration'][clean_df['vehicle_condition'] == "NEW"] = 2023

clean_df = clean_df[~clean_df['year_of_registration'].isna()]
clean_df = clean_df[~clean_df['year_of_registration'].isin(range(0, 1921))]
clean_df.loc[:, 'year_of_registration'] = clean_df['year_of_registration'].astype(int)

# clean_df['reg_code'][(clean_df['reg_code'].isna()) & (clean_df['year_of_registration'] == 2023)] = 23
# clean_df = clean_df[~clean_df['reg_code'].isna()]

clean_df = clean_df[~clean_df['body_type'].isna()]

clean_df = clean_df[~clean_df['fuel_type'].isna()]

print(clean_df.shape)
clean_df.head()
```

2.3.For standard_colour, instead of dropping missing rows ridiculously, I gathered all the colours and generated random colours from my list for the missing columns.

2.4.I did a tricky thing next, for all the new cars I changed it to 2023 for year registered (This doesn’t matter, I just thought they can be bought this year)

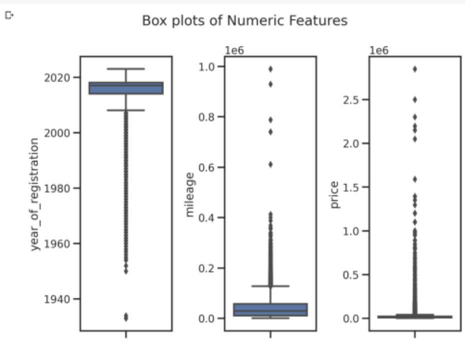
2.5.For invalid year of registration, I went further with my inquisitive nature, I saw some real cars registered between 1921 & ‘new’. I literarily googled the old car and saw them. So, instead of dropping unintelligently, I simply dropped values below 1921. Moreso, cars were invented in 1886 and they were weird cars at that, more like carriages. Doesn’t make sense to have cars registered earlier. Then, I converted to integers.

2.6.Body_type & fuel_type missing values are removed. No point keeping them, no impact

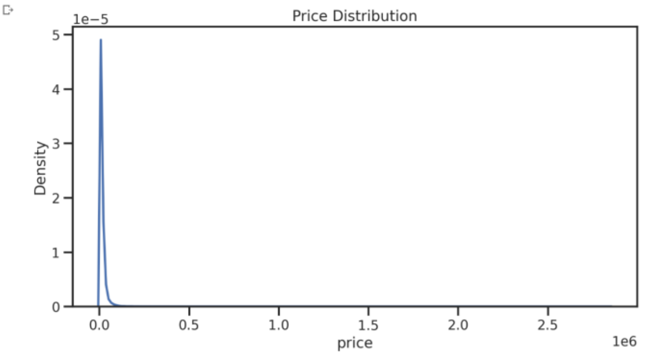
	mileage	standard_colour	standard_make	vehicle_condition	year_of_registration	price	body_type	crossover_car_and_van	fuel_type
0	0.0	Grey	Volvo	NEW	2023	73970	SUV	False	Petrol Plug-in Hybrid
1	108230.0	Blue	Jaguar	USED	2011	7000	Saloon	False	Diesel
2	7800.0	Grey	SKODA	USED	2017	14000	SUV	False	Petrol
3	45000.0	Brown	Vauxhall	USED	2016	7995	Hatchback	False	Diesel
4	64000.0	Grey	Land Rover	USED	2015	26995	SUV	False	Diesel

2.7.My cleaned dataset

```
numeric_cols = ['mileage', 'price']
fig, ax = plt.subplots(ncols=3)
sns.boxplot(clean_df, y='year_of_registration', ax=ax[0])
sns.boxplot(clean_df, y='mileage', ax=ax[1])
sns.boxplot(clean_df, y='price', ax=ax[2])
plt.suptitle("Box plots of Numeric Features")
plt.tight_layout()
```




```
plt.figure(figsize=(10, 5))
sns.kdeplot(clean_df[numeric_cols], x='price')
plt.title("Price Distribution")
plt.savefig("")
```



2.8.Here, I visualized my year of registration, mileage, price and the distribution. To see the outliers. I know there are but I want to make it robust.

Chapter 3. Feature Selection and Dimensionality Reduction

3.0. I did numerical scaling to my price, mileage, year of registration here.

3.1. To prepare the new dataset for training, I dropped price and parsed the rest into X

3.2. I assigned the price, my target to Y

3.3. When I use the get_dummies(), it auto creates a new column for each unique value in the variable. For each row, the corresponding column is set to 1, while all other columns are set to 0. The reason why I did this is to make my Machine learning easy. This step is known as one-hot encoding.

```
[11] scaler = StandardScaler()
     numeric_cols = ['mileage', 'year_of_registration', 'price']
     scaled_num = scaler.fit_transform(clean_df[numeric_cols])

     X = clean_df.drop('price', axis=1)
     X['mileage'] = scaled_num[:, 0]
     X['year_of_registration'] = scaled_num[:, 1]

     y = scaled_num[:, 2]
```

```
[12] X.head()
```

	mileage	standard_colour	standard_make	vehicle_condition	year_of_registration	body_type	crossover_car_and_van	fuel_type
0	-1.086376	Grey	Volvo	NEW	1.569499	SUV	False	Petrol Plug-in Hybrid
1	2.024252	Blue	Jaguar	USED	-1.002011	Saloon	False	Diesel
2	-0.862197	Grey	SKODA	USED	0.283744	SUV	False	Petrol
3	0.206965	Brown	Vauxhall	USED	0.069452	Hatchback	False	Diesel
4	0.753042	Grey	Land Rover	USED	-0.144841	SUV	False	Diesel

```
[13] X = pd.get_dummies(X)
     X.head()
```

	mileage	year_of_registration	standard_colour_Beige	standard_colour_Black	standard_colour_Blue	standard_colour_Bronze	standard_colour_Brown	st
0	-1.086376	1.569499	0	0	0	0	0	
1	2.024252	-1.002011	0	0	1	0	0	
2	-0.862197	0.283744	0	0	0	0	0	
3	0.206965	0.069452	0	0	0	0	1	
4	0.753042	-0.144841	0	0	0	0	0	

5 rows × 158 columns

Chapter 4. Model Building

4.1 **A Linear Model**: After importing my necessary libraries, I used the X_train and y_train data to train the linear regression model, the target variable values are then predicted using the X_test data. The values in the array are my y_pred.

```
[15] from sklearn.linear_model import LinearRegression
     from sklearn.metrics import precision_recall_fscore_support

     # We will create a Linear Regression model object
     lr_model = LinearRegression()

     # Then here we train the model
     lr_model.fit(X_train, y_train)

     # Then, we predict the target variable model we trained above
     y_pred = lr_model.predict(X_test)
```

```
[16] y_pred

     array([-0.14553737, -0.56337729,  0.11060696, ...,  0.14093772,
            -0.1159312 , -0.17476165])
```

```
[17] y_test

     array([-0.24141814, -0.46871322,  0.06301404, ..., -0.16886895,
            -0.21590587, -0.13207982])
```

```
[18] from sklearn.metrics import mean_squared_error, r2_score
     # To get my mean squared error
     mse = mean_squared_error(y_test, y_pred)
     print("Mean squared error: %.2f" % mse)

     # For the root mean squared error
     rmse = np.sqrt(mse)
     print("Root mean squared error: %.2f" % rmse)

     # coefficient of determination (R-squared)
     r2 = r2_score(y_test, y_pred)
     print("Coefficient of determination (R-squared): %.2f" % r2)

     Mean squared error: 0.30
     Root mean squared error: 0.55
     Coefficient of determination (R-squared): 0.59
```

To know how well my the test fits the data, I checked for; Mean squared error (MSE) measuring the average squared difference of the predicted values and the actual values. Root mean squared error (RMSE) is the square root of the MSE above. Coefficient of determination explains the proportion of the variance in the target variable that is explained by the linear regression model.

Result

With a mean squared error of 0.3 and a root mean squared error of 0.55, my linear regression model's overall accuracy was 59%.

4.2 **A Random Forest Model**: Using the training set of data, I built my Random Forest regressor object with 40 trees. Finally, fit & predicted the target variable for the test using the trained model.

```
from sklearn.ensemble import RandomForestRegressor

# Creating the Random Forest regressor object with 40 trees
rf_model = RandomForestRegressor(n_estimators=40, random_state=60)

# Let's Train the model on the training data
rf_model.fit(X_train, y_train)

# Then we Predict the target variable using the trained model
y_pred = rf_model.predict(X_test)
```

```
[20] from sklearn.metrics import mean_squared_error, r2_score

     # Predicting the target variable using the trained Random Forest model
     y_pred = rf_model.predict(X_test)

     # Calculating my mean squared error
     mse = mean_squared_error(y_test, y_pred)
     print("Mean squared error: %.2f" % mse)

     # To calculate the root mean squared error
     rmse = np.sqrt(mse)
     print("Root mean squared error: %.2f" % rmse)

     # Coefficient of determination (R-squared)
     r2 = r2_score(y_test, y_pred)
     print("Coefficient of determination (R-squared): %.2f" % r2)

     Mean squared error: 0.23
     Root mean squared error: 0.48
     Coefficient of determination (R-squared): 0.69
```

Result

The Random Forest model is better than Linear regression above because it reduced the mean squared error to 0.23, RSME to 0.48 and improved the R-squared value to 69%.

4.3 **A Boosted Tree**: I used a learning rate of 0.1 and a random state of 60, this trains the

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error, r2_score

# To create the Gradient Boosting regressor object
gb_model = GradientBoostingRegressor(n_estimators=40, learning_rate=0.1, random_state=60)

# Training the model with the training data
gb_model.fit(X_train, y_train)

# Here we predict the target variable using the trained model
y_pred = gb_model.predict(X_test)

# We evaluate the performance
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)
print("Mean squared error: %.2f" % mse)
print("Root mean squared error: %.2f" % rmse)
print("Coefficient of determination (R-squared): %.2f" % r2)

Mean squared error: 0.30
Root mean squared error: 0.54
Coefficient of determination (R-squared): 0.60
```

Gradient Boosting Regression model with 40 estimators. The target variable is then predicted by the model like the models above where accuracy of the prediction is assessed using R-squared, MSE, and RSME.

Result

The MSE being the average squared difference between pred and actual values is 0.30, RMSE i.e square root as said earlier is 0.54. The coefficient of determination (R-squared), which is 60% in this case. So, Random Forest is better in explaining my data.

4.4. **LightGBM**: I found this model type while doing my assessment and tried it. I installed

```
[22] !pip install lightgbm

Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: lightgbm in /usr/local/lib/python3.10/dist-packages (3.3.5)
Requirement already satisfied: wheel in /usr/local/lib/python3.10/dist-packages (from lightgbm) (0.40.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from lightgbm) (1.22.4)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from lightgbm) (1.9.3)
Requirement already satisfied: scikit-learn!=0.22.0 in /usr/local/lib/python3.10/dist-packages (from lightgbm) (1.2.2)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn!=0.22.0->lightgbm) (1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn!=0.22.0->lightgbm) (3.1.0)

[23] import lightgbm as lgb

# This is a new model, really cool.
# Create a LightGBM dataset
train_data = lgb.Dataset(X_train, label=y_train)

# Set the hyperparameters for the model
params = {
    "objective": "regression",
    "metric": "rmse",
    "num_leaves": 31,
    "learning_rate": 0.05,
}

# Training the model on with training data
lgb_model = lgb.train(params, train_data, num_boost_round=100)

# To predict the target variable with the trained model
y_pred = lgb_model.predict(X_test)

[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Warning] Auto-choosing col-wise multi-threading, the overhead of testing was 0.075066 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 523
[LightGBM] [Info] Number of data points in the train set: 319096, number of used features: 105
[LightGBM] [Info] Start training from score 0.000162

[24] # Here, we predict the target variable with the trained model
y_pred = lgb_model.predict(X_test)

# Evaluate the performance
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)
print("Mean squared error: %.2f" % mse)
print("Root mean squared error: %.2f" % rmse)
print("Coefficient of determination (R-squared): %.2f" % r2)

Mean squared error: 0.24
Root mean squared error: 0.49
Coefficient of determination (R-squared): 0.68
```

the LightGBM library, created a LightGBM dataset, then went on to set my model's hyperparameters. Lastly, trained the model using the training data, and the target variable is predicted as the initial model I did above.

Result

The trained LightGBM model is used to predict the target variable, and the performance of the model is assessed using the mean squared error, root mean squared error, and coefficient of determination (R-squared). When completed, coefficient of determination (R-squared), target variable's variance is explained to a degree of 68%, the MSE is 0.24 shows low level error and my RMSE of 0.49 meaning predictions are close to true values but not totally at all.

4.5. **An Averager/Voter/Stacker Ensemble**

- I created list of all models used above except LightGBM to use in the ensemble
- Then, I trained each model on the training data and tried the predictions on the test data
- Next, I computed the aevrage of the predictions above and the root mean squared error (RMSE) from the same average predictions
- Went on to computes the majority vote of the predictions and the RMSE of the majority vote's predictions
- I did a new dataset from the predictions and used it to train a meta-model (optional)
- Let's see the stacked model's predictions and the RMSE.

Result

- The Ensemble RMSE is 0.46: root mean squared error of the predictions made by average the predictions of the three models
- Majority Vote RMSE: 0.55: root mean squared error of the predictions made by taking the majority vote of the three models.
- Stacked Model RMSE: 0.44: root mean squared error of the predictions made by training a meta-model on the predictions of the three models. I used linear regression for my meta-model.

```
[25]
# Creating a list of models to use in the ensemble
models = [rf_model,
          gb_model,
          lr_model]

# Train each model on the training data and make predictions on the test data
predictions = []
for model in models:
    y_pred = model.predict(X_test)
    predictions.append(y_pred)

# Computing the average of the predictions
average_predictions = np.mean(predictions, axis=0)

# We compute the root mean squared error based on the averager predictions
rmse = np.sqrt(mean_squared_error(y_test, average_predictions))
print("Ensemble RMSE: {:.2f}".format(rmse))

# We compute the majority vote of the predictions
vote_predictions = np.round(np.mean(predictions, axis=0))

# We Compute the root mean squared error of the majority vote's predictions
rmse = np.sqrt(mean_squared_error(y_test, vote_predictions))
print("Majority Vote RMSE: {:.2f}".format(rmse))

# Here we will create a new dataset from the predictions and use it to train a meta-model (advanced)
# I really don't need this step, But let's see
stacked_dataset = np.column_stack(predictions)
meta_model = LinearRegression()
meta_model.fit(stacked_dataset, y_test)

# We make predictions with the stacked model
stacked_predictions = meta_model.predict(stacked_dataset)

# Finally. Computing the root mean squared error of the stacked model's predictions
rmse = np.sqrt(mean_squared_error(y_test, stacked_predictions))
print("Stacked Model RMSE: {:.2f}".format(rmse))

Ensemble RMSE: 0.46
Majority Vote RMSE: 0.55
Stacked Model RMSE: 0.44
```

```
[26]
from sklearn.ensemble import VotingRegressor
from sklearn import metrics
enble_model = VotingRegressor([gb_model,lr_model])
```

```
[27]
estimators = [('gb', gb_model), ('lr', lr_model)]
enble_model = VotingRegressor(estimators)
```

Observation

I noticed the scores are somewhat on par with my other models which is bizarre. I expected a better result since it averages all the model. Apparently, the linear regression just takes in the new dataset as just figures.

5.0 Model Evaluation and Analysis

5.1 Overall Performance with Cross-Validation

a. I went on to Perform the Cross-Validation utilizing the training data (X_train and Y_train) and a linear regression model (lr_model), the code runs 5-fold cross-validation with 'r2' as my score metric. The observed result of the five scores array, with the negative scores denoting a bad match and positive values denoting a better fit, shows the goodness of fit of the model for each fold. My first score's is severe negativity (-3.07834745e+15) and the other scores' proximity to zero, the first score stands out as an anomaly.

```
[ ] #LInear model
    from sklearn.model_selection import cross_val_score
    from sklearn import metrics
    result = cross_val_score( lr_model, X_train, y_train, cv=5, scoring='r2')
    result

    array([-3.07834745e+15,  5.69733780e-01, -5.47851404e+14,  4.89956444e-01,
            5.55501405e-01])
```

```
[ ] #The Boosted Tree model
    from sklearn import metrics
    result = cross_val_score( gb_model, X_train, y_train, cv=5, scoring='r2')
    result

    array([0.47280835, 0.58396908, 0.58407736, 0.63075761, 0.62191164])
```

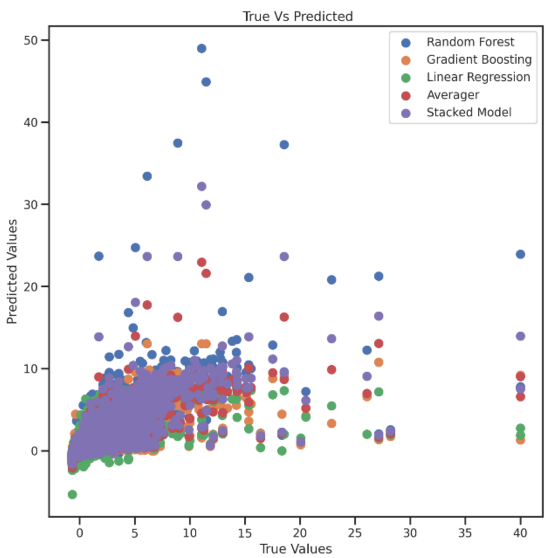
```
[ ] #my Random Forest model
    from sklearn import metrics
    from sklearn.model_selection import cross_val_score
    result = cross_val_score(rf_model, X_train, y_train, cv=5, scoring='r2')
    result

    array([0.64259594, 0.713573  , 0.75093412, 0.77949902, 0.74459367])
```

b. For Boosted Trees, the scores range from 0.47 to 0.63, the array of 5 scores shows that the model has a moderate to good fit for all folds. Overall, the findings of the cross-validation indicate that the Boosted Tree model would be a superior option over the linear regression model.

c. For Random Forest, the scores range from 0.64 to 0.78. Now, the cross-validation results shows that the Random Forest model could be a strong candidate for further analysis.

5.2 True vs Predicted Analysis

```
# Plot true vs predicted values for each model and the ensemble
plt.figure(figsize=(10, 10))
plt.scatter(y_test, predictions[0], label="Random Forest")
plt.scatter(y_test, predictions[1], label="Gradient Boosting")
plt.scatter(y_test, predictions[2], label="Linear Regression")
plt.scatter(y_test, average_predictions, label="Averager")
plt.scatter(y_test, stacked_predictions, label="Stacked Model")
plt.xlabel("True Values")
plt.ylabel("Predicted Values")
plt.title(" True Vs Predicted")
plt.legend()
plt.show()
```



The scatter plot here shows my 5 models (Random Forest, Gradient Boosting, Linear Regression, an Averager model, and a Stacked model). This contrasts the true values of the target variable (y_test) with their anticipated values. The true values are on the x-axis and predicted on y_axis. The predicted values are plotted as a scatter of points on the graph for each model.

Result and Observation:

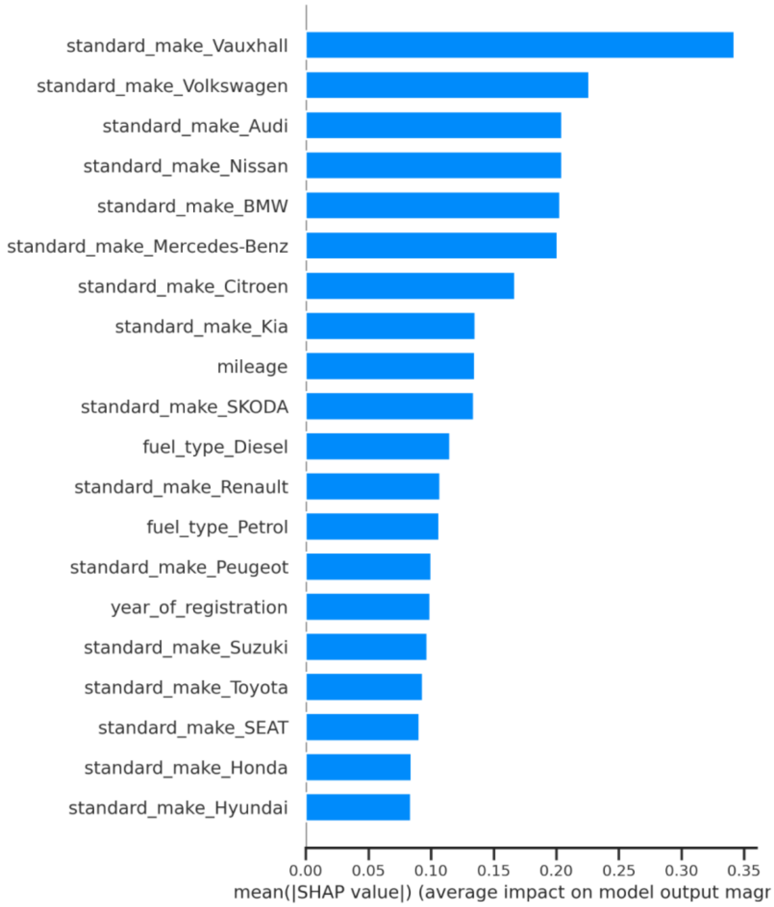
From what we can see comparing the performance of the models visually, we have almost all the models towards 0 and at one side. This means that the models are not making too much large errors in their prediction. They are similar, my actual and predicted. Each colour on the graph corresponds to a particular model, as shown by the legend.

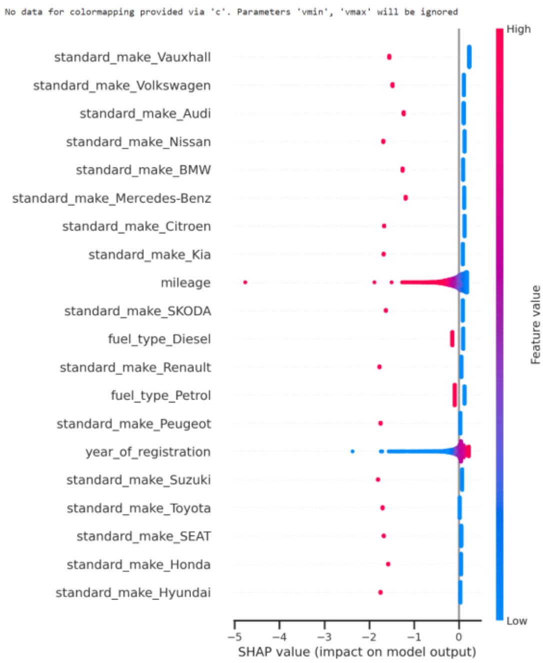
5.3 Global and Local Explanations with SHAP

```
[ ] !pip install -q shap
    import shap
    shap.initjs()

        ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 572.6/572.6 kB 27.0 MB/s eta 0:00:00
```

```
[ ] import shap
    # model is your desired model that you want to show its shap values, X_train is the data that your model has been trained on
    shap_values = shap.LinearExplainer(lr_model, X_train).shap_values(X_test)

    # this line will plot the shap graph bar for the training data
    shap.summary_plot(shap_values, X_test, plot_type="bar")

    # plotting the usual shap graph
    shap.summary_plot(shap_values, X_test)
```

I imported the library for SHAP and used linear model which creates the SHAP values using a LinearExplainer object and uses two plots namely; a summary bar plot and a scatter plot to show the feature importance of each variable in the model's predictions. The scatter plot shows a more in-depth look at the link between the feature values and the model's predictions, while the summary bar plot displays the average absolute SHAP value for each feature.



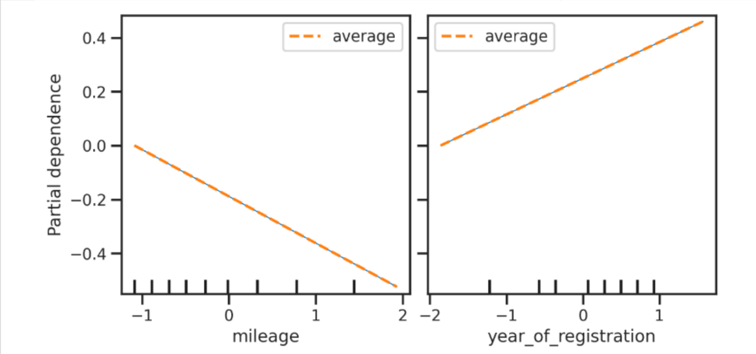


Result and Observation

The plots above shows the features that are perfect. Meaning they are the best for my model. You will notice that some of them are not originally column features from the adverts dataset e.g Standard_make_vauxhall and Volkswagen's. They were created from when I did the one hot encoding. My future work will be to test this features in building another model to see if they perform better.

5.4 Partial Dependency Plots

```
[ ] from sklearn.inspection import PartialDependenceDisplay

fig, ax = plt.subplots(figsize=(8,4), constrained_layout=True)
PartialDependenceDisplay.from_estimator(
    lr_model,X_test, features=['mileage', 'year_of_registration'], kind='both', centered=True,
    subsample=100, grid_resolution=30, n_jobs=2, random_state=0,
    ax=ax, n_cols=2
);
```



After importing my libraries, I generated my partial dependence plot for the mileage and year_of_registration features. The plot shows the anticipated target variable shifts as each feature is altered while maintaining those of the other features. Another thing is, so as to display individual and two-way interaction graphs, the 'kind' option is set to 'both'. Just as I did with the other above, the computation and presentation of the plot are done by the 'subsample', 'grid_resolution', and 'n_jobs' parameters. The 'axe' parameter defines the plot's axes, while the 'n_cols' option establishes the plot's number of columns.

Result & Observation: The main reason why all my result and also PDP is showing -1 to 2, -2 to 1, -0.4 to 0.4 is because of my scaler. I had scaled it earlier. The plot as mentioned above shows my spread and average of the feature I selected to understand it more.