Building a machine learning model to estimate the value of motorbikes

Artificial Intelligence and Machine Learning – CMP5367

Birmingham City University

Contents

Introduction

1.1 Dataset Identification

The dataset is sourced from Kaggle:

https://www.kaggle.com/datasets/mexwell/motorbike-marketplace/data

The data used in the dataset is originally sourced from Zenrows:

www.zenrows.com/datasets/europe-motorbikes

The purpose of creating an AI model, based on this dataset, is to create an effective way of evaluating a motorbike based on simple parameters / features of the vehicle. This AI model could potentially be used by businesses that deal with mass sales of motorbikes to help with evaluating motorbikes, or even regular people who plan to sell their vehicle but are unsure if their vehicle is correctly priced or not.

The dataset used to train this predictive model is a collection of sales / listings of different motorbikes around Europe. The dataset initially consists of the following columns:

- Price
  - The value of the motorbike (in euros)
- Mileage
  - The mileage of the motorbike
- Power
  - The cubic capacity (cc) of the motorbike's engine
- Make Model
  - The make and model of the motorbike
- Date
  - The date of manufacture
- Fuel
  - The type of fuel the motorbike uses
- Gear
  - The transmission type
- Offer Type
  - The state / listing of the motorbike
- Version
  - An alternative version of the model
- Link
  - Part of the URL where the offer / listing is found

1.2 Supervised Learning Task Identification

The approach used to train this model is the use of regression rather than classification as the dataset contains mostly statistical (numerical) data, such as price or mileage. Additionally, the

purpose of this AI model is to predict a specific value of a vehicle rather than “predict discrete class labels” [1]. A comparison between the two approaches, in this scenario of motorbikes, would be that with a regression model, you are able to predict a price of a motorbike based on values given by the user, whereas a classification model would help predict if the motorbike is a high emission or low emission motorbike based on specific thresholds created by the data from the dataset. And for this experiment, training a model to predict the value of a motorbike is much more valuable to the user as vehicle prices are affected by multiple factors as vehicle prices are not set in store and fluctuate from even the slightest alterations.

Exploratory Data Analysis

2.1 Question Identification

The main purpose of this model is to predict value of motorbikes based on specific features, however, with this model could you also tell which feature affects the value of the motorbike the most? Which feature affects the value of the motorbike the least?

2.2 Splitting the dataset

The main factors from the dataset that are going to be used for the different regression models are shown below in *Figure 1*. All the columns that are to be used are placed into a new variable called **dsLearn** which will be referenced later when splitting / allocating data to the independent and dependent variables.

```
# selecting the columns to use for training the model
dsLearn = adaptedSet[['Mileage',
                      'Power',
                      'Make',
                      'Fuel Type',
                      'Transmission Type',
                      'State of Vehicle',
                      'Year',
                      'Value']]
```

Figure 1: Selecting columns from the dataset to use for training

The main purpose of the AI model is to predict the price of a motorbike based on different parameters of the vehicle. As such, in Figure 2.1 you can see that the independent variables (stored as the **x** variable") are the following columns: **Mileage**, **Power**, **Make**, **Fuel Type**, **Transmission Type**, **State of Vehicle** and **Year**. The values from these columns will be used to predict the **dependant variable** (stored as the **y** variable), which the model will be using the Value column (shown in Figure 2.2) to predict a value of the motorbike based on the **independent variables**.

```
# allocating the columns from Mileage to Year as x - the data used to predict
x = dsLearn.iloc[:, 0 : -1]
```

Figure 2.1: Allocating the independent variables

```
# allocating the column Value as y - what i am trying to predict
y = dsLearn.iloc[:, -1:, ]
```

Figure 2.2: Allocating the dependent variable

In the snippet of code below (Figure 3), the before mentioned **x** and **y** variables will be used to split all the data from the dataset into test data and training data. The way this data will be split is into an 80/20 split – 80% of the data from the dataset will be used as training data, and 20% of the data will be used as testing data. This is done by setting the parameter **test_size** to **0.2** in the method. This is a fair split of data as the whole dataset contains around 35,000 individual rows which leaves around 28,000 rows of data for the model to analyse and train on to be able to predict a more accurate price, and the remaining 7,000 rows to be used as testing data which will help calculate the effectiveness of the model later. Additionally, the method also uses a parameter called **random_state [2]** which allows for the selection of random rows from the dataset, rather than the same rows to be chosen repeatedly – but for testing and consistency purposes, the property has been set to a static number of 42 to prevent a random generation of data every time the program is ran.

```
# separating the data into test and train data
xTrain, xTest, yTrain, yTest = train_test_split(x, y, random_state=42, test_size=0.2)
```

*Figure 3: Splitting the **x** and **y** values into training and testing data*

2.3 Exploratory Data Analysis Process and Results

One of the major issues that was apparent in the dataset, was that there was a lot of missing values from some of the columns. Although its not a lot of columns that are affected by missing values, the gear column has 22,070 missing values, which equates to ~63% of values missing which is an issue that will affect the accuracy of the model. Although there are multiple factors that affect the accuracy of prediction models, having this many missing values will greatly affect the accuracy of as these missing values will have to be replaced rather than dropped as this would equate to the model losing a large quantity of data.

```
#showing null values in the dataset
ds.isnull().sum()
```

	0
price	0
mileage	0
power	6319
make_model	0
date	0
fuel	3190
gear	22070
offer_type	0

Figure 4: Function showing the count of missing values per column

Another issue, although a minor issue, was that majority of the data types used within the dataset were objects (string values / non-numeric values) which would pose an issue when using regression models as these models strictly work with numeric values, such as integer and float values, therefore the values from these columns later on would need to be converted into a numerical data type for the regression models to be able to utilise this information.

```
[ ] # checking which columns need to be converted from object datatype to int datatype
    ds.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 34917 entries, 0 to 34916
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   price       34917 non-null  int64
 1   mileage     34917 non-null  int64
 2   power       34917 non-null  float64
 3   make_model  34917 non-null  object
 4   date        34917 non-null  object
 5   fuel        34917 non-null  object
 6   gear        34917 non-null  object
 7   offer_type  34917 non-null  object
dtypes: float64(1), int64(2), object(5)
memory usage: 2.1+ MB
```

Figure 5: Function showing a simple breakdown of the dataset

Another major issue that was found when exploring the dataset, was that there is a lot of outliers / noise within the data, and this was shown using the **describe()** method shown in *Figure 6* below. This is shown in *Figure 6* when comparing the mean, max and standard deviation values of each column, as there is too much disparity between each of the values which shows that

there is too many outliers and within each of the columns and these outliers will need to be removed to provide more accurate results

```
# describing the dataset pre-encoding
ds.describe()
```

	price	mileage	power
count	3.491700e+04	3.491700e+04	34917.000000
mean	4.568532e+04	2.183175e+04	191.470344
std	4.850120e+06	2.059421e+05	8481.187105
min	1.000000e+00	0.000000e+00	1.000000
25%	6.999000e+03	2.932000e+03	60.000000
50%	9.920000e+03	1.100000e+04	107.000000
75%	1.259000e+04	2.500000e+04	125.000000
max	8.888889e+08	9.999999e+06	913595.000000

Figure 6: Function showing the count, mean, standard deviation, min, inner and outer quartile, midpoint and max values of the dataset (only shows 3 columns as this is pre-encoding)

These outliers can be visualised by plotting each of the columns into a boxplot graph. *Figure 7.1, 7.2 & 7.3* it is shown that there are outliers within each column (visualised using the small dots on the graph) and the disparity of these outliers is quite large compared to the box plot diagram as the diagram itself is only a small line at the bottom of the graph. These outliers will need to be removed when processing the data.

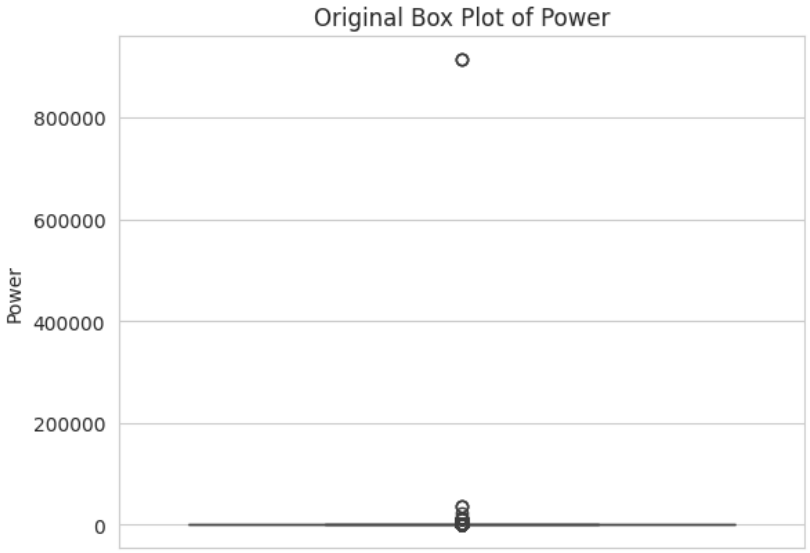

Figure 7.1: Box plot of the power column showing outliers

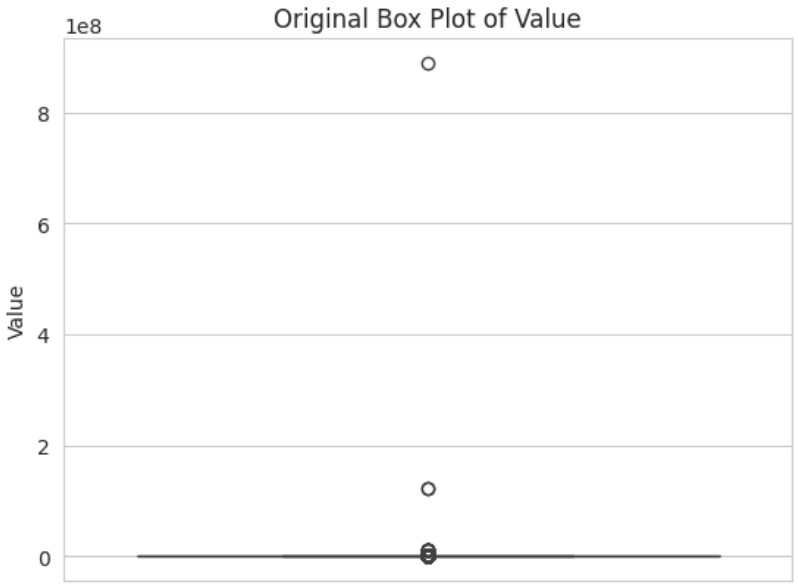

Figure 7.2: Box plot of the value column showing outliers

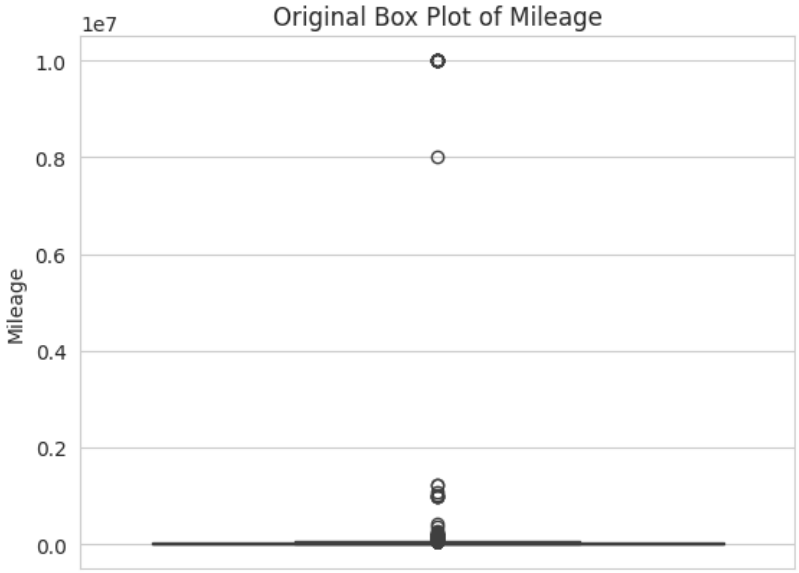

Figure 7.3: Box plot of the mileage column showing outliers

2.4 EDA Conclusions

After exploring the dataset, it is clear that there is a lot of work necessary – like replacing null values with a reasonable value in the columns that contain null values, changing some of the object datatype columns into numerical datatypes, encoding non-numeric values into numerical values, and general dataset clean up to make the dataset easier to work with and give it more meaning and clarity.

Experimental Design

3.1 Identification of chosen supervised learning algorithms

As this AI model is of a regression type, the following machine learning algorithms have been utilised to create a predictive AI model:

- Linear Regression
  - Very simple model
  - The model predicts the value of unknown data by using related data [3]
- Decision Tree Regression
  - Uses a flowchart-like tree structure [4]
  - Models all possible results [4]
- Random Forest Regression
  - Utilises the prediction of multiple models (decision trees) – also known as ensemble learning [5]
  - This method can be used for both regression and classification
- Lasso Regression
  - Prevents overfitting [6]
  - LASSO stands for Least Absolute Shrinkage and Selection Operator [6]

These algorithms have been chosen as the dataset has been converted to only numerical values, therefore regression type algorithms would be the more suitable options for this prediction model.

3.2 Identification of Appropriate Evaluation Techniques

After implementing the 4 different regression algorithms each of the algorithms have varying scoring metrics relating to how accurate each model is for this specific dataset, and how well it can predict the value of a motorbike. All the regression algorithms are working with the exact same data from the dataset. Each model has been measured with the following metrics:

- Mean Squared Error (MSE)
  - The average squared difference between the training and predicted values [8]
- Root Mean Squared Error (RMSE)
  - Square root of MSE
  - Easier to understand
- Mean Absolute Error (MAE)
  - The absolute difference between the predicted and actual values [9]
- R2 Score
  - How much of an impact the independent variables are on the dependent variable
- Training and Test Data Score
  - How accurate the predictions are based on the data used

3.3 Data Cleaning and Pre-processing Transformations

The very first step of the data cleaning process was to remove any columns that would not be necessary to the training of the model. There were 2 columns that needed to be removed, the **version** column and the **link** column. The **version** column contained ~50% missing values (as you can see in *Figure 8* below), alongside this, the data that it did contain wasn't too crucial to the prediction model as it only stored alternative versions of specific models. The **link** column contained URL links to the listing of the specific motorbike, which would also have no use for predicting the price of the motorbike.

```
# showing the dataset information
ds.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 34917 entries, 0 to 34916
Data columns (total 10 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   price       34917 non-null  int64  
 1   mileage     34917 non-null  int64  
 2   power       28598 non-null  float64
 3   make_model  34917 non-null  object 
 4   date        34917 non-null  object 
 5   fuel        31727 non-null  object 
 6   gear        12847 non-null  object 
 7   offer_type  34917 non-null  object 
 8   version     17413 non-null  object 
 9   link        34917 non-null  object 
dtypes: float64(1), int64(2), object(7)
memory usage: 2.7+ MB
```

Figure 8: Function showing a breakdown of the dataset

The next step in preparing the dataset for training was checking for null values in all the columns. Although the function used in *Figure 8* shows the number of rows that contain a value, the function shown in *Figure 9* gives the user a much more straightforward and easier to read alternative

```
#showing null values in the dataset
ds.isnull().sum()
```

	0
price	0
mileage	0
power	6319
make_model	0
date	0
fuel	3190
gear	22070
offer_type	0

Figure 9: Function showing the count of null values in the dataset

There are only 3 columns that contain missing values: **power**, **fuel** and **gear**. One possibility of dealing with null data would be to drop all the rows that contain a null value. However, if all the null values were to be dropped, then the dataset would lose around a minimum of 63% of data, which would significantly reduce the amount of data that each model has to train on, potentially further reducing the accuracy of each model. Therefore, the more suitable option would be to replace the null values with existing data.

First, the most common value (mode) must be checked for the column – for this example, this is done for the **power** column. This is shown by the function **value_counts()** used in *Figure 10.1* and the output provided.

```
# finding the most common value in the column
ds['power'].value_counts()
```

power	count
125.0	1594
110.0	1068
95.0	862
48.0	720
160.0	684
...	...
880.0	1
409.0	1
468.0	1
430.0	1
419.0	1

287 rows × 1 columns

Figure 10.1: Function showing the most common values to the least common values

Then all the null values must be replaced within the column to the most common value found before – in this case it is 125. This is done via the function **fillna()** shown in *Figure 10.2*. The function contains the parameter **value**, and this parameter is set to **125**. Also, for the changes to be applied to the current dataset, the **new power column overrides** the current power column.

```
# will be using 125 as the value to replace the null values as it is the most common one
ds['power'] = ds['power'].fillna(value=125)
```

Figure 10.2: Function to replace the null values

Finally, the same function from *Figure 10.1* is used after replacing the null values to check if the changes have been made to the current dataset. As you can see in *Figure 10.3*, the output is different compared to *Figure 10.1*, as the power column shows 0 null values. The same method was applied to the remaining two columns, where the most common (mode) value was chosen to replace the null values.

```
# checking if the fillna function applied properly to the column
ds.isnull().sum()
```

	0
price	0
mileage	0
power	0
make_model	0
date	0
fuel	3190
gear	22070
offer_type	0

Figure 10.3: Function to check if the new values have been applied to the dataset

The next step in preparing the dataset for training was converting the columns with non-numeric values into column with numeric values (integer or float data types). This is done by applying an encoding technique to each column and converting each unique entry of the column into its numerical counterpart. All the columns with an **object** datatype (**make_model**, **fuel**, **gear** and **offer_type**), will need to be converted into an **int** (integer) datatype. However, the column **make_model** will have a more steps than just encoding. Whereas the **date** column will only have to have its data split.

```
# checking which columns need to be converted from object datatype to int datatype
ds.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 34917 entries, 0 to 34916
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   price       34917 non-null  int64
 1   mileage     34917 non-null  int64
 2   power       34917 non-null  float64
 3   make_model  34917 non-null  object
 4   date        34917 non-null  object
 5   fuel        34917 non-null  object
 6   gear        34917 non-null  object
 7   offer_type  34917 non-null  object
dtypes: float64(1), int64(2), object(5)
memory usage: 2.1+ MB
```

Figure 11: Function showing all columns and their datatypes

First, the encoder needs to be initialised – for this example, the **fuel** column will be used – as shown in *Figure 12.1* by calling the class of **LabelEncoder()**. Then all the unique values from column are fitted to the encoder using the **fit()** function. Then a new column is made in the dataset to apply the transformed values to all the rows accordingly. Then in *Figure 12.2* using the **value_counts()** function on both the **fuel** and **fuelEncoded** columns, it shows which number is allocated to what fuel type. After ensuring that the changes have been applied properly to the dataset, then the old **fuel** column can be removed from the dataset, as shown in *Figure 12.3*. This same process is repeated for the **gear**, **offer_type**, and the **make** column once all the steps for the **make_model** column have been completed.

```
# creating a variable for the encoder
lblFuel = LabelEncoder()

# grab each unique variable from the column and turn it into its numerical version
lblFuel.fit(ds['fuel'].drop_duplicates())

# create a new column and apply the encoding values
ds['fuelEncoded'] = lblFuel.transform(ds['fuel'])
```

*Figure 12.1: Initializing the encoder and converting the **fuel** values into numerical values*

```
# checking which numerical value corresponds to the non-numerical value
ds[['fuel', 'fuelEncoded']].value_counts()
```

		count
fuel	fuelEncoded	
Gasoline	3	33684
Two Stroke Gasoline	6	497
Electric	1	411
Others	5	171
Diesel	0	144
Electric/Gasoline	2	8
LPG	4	2

Figure 12.2: Comparing the non-numerical values to their numerical counterparts

```
# dropping the old column from the dataset as it is unnecessary
ds = ds.drop('fuel', axis=1)
```

*Figure 12.3: Dropping the redundant **fuel** column*

The **make_model** column has a long string value. And the values follow a similar pattern, where it is the make followed by the model – for example, “Harley-Davidson Softail”, where “Harley-

Davidson" is the make, and "Softail" is the model. As all the values in the column follow the same rule set, the **make** of the motorbike, can be derived from the first word of the cell, and the **model** of the motorbike can be derived from the remaining words in the cell. Using the function **split()** with the parameters **" "** (setting a space as the delimiter to let the program know where to split the data) and **n = 1** (to allow for the data to be only split into **only** 2 different values) will place the split data into the variable **splitMakeModel**. From this variable only the make of the motorbike will be grabbed and placed into a new column in the dataset, called **make**. Then the **make_model** column is removed as it is redundant, using the **drop()** function. Then the new **make** column will need to be encoded into its numerical counterpart, following the same steps shown in *Figures 12.1 – 12.3*.

```
# splitting the make_model column into make and model by using ' ' as the delimiter
# limiting the amount of slices to 2 by using n = 1
splitMakeModel = ds['make_model'].str.split(' ', n = 1, expand=True)

# creating a new column for make
ds['make'] = splitMakeModel[0]

# dropping the old column from the dataset as it is unnecessary
ds = ds.drop('make_model', axis=1)
```

*Figure 13: Splitting the values of the **make_model** column into just the **make** and creating a **new column** for it in the dataset, then dropping the redundant **make_model** column*

The **date** column contains values in the following format "MM/YYYY", and due to there being a non-numerical character in the cell ("/"), the datatype is changed to an object (string data type). The column also contains the value "– (First Registration)" which is a non-numerical value and will need to be replaced. Using the **replace()** (shown in *Figure 14.*1) function the previous value is changed to a set value of "12/2024" to give the motorbike a valid registration date, as the most reasonable date to allocate it would be to the current month and year. After replacing the values, the column can then have the same technique that is used on the **make_model** column (refer to *Figure 13*) is used here, where each of the values must be split into 2 different (**new**) columns – the **month** column, and the **year** column. The month column will contain all the characters before the "/" character, and the year column will contain all the characters after the "/" character (the "/" acts as the delimiter). The new columns will need to be declared as an int datatype as they only contain numbers (shown in *Figure 14.2*). Once the **date** column has been split into the **month** and **year** column, the **date** column is dropped as it is redundant and will only duplicate the data that is already in the dataset.

```
# replacing the non-date value "- (First Registration)" in the column to a chosen date
ds['date'] = ds['date'].replace("- (First Registration)", "12/2024")
```

*Figure 14.1: Function to replace values to fit the date **format** of the column*

```
# converting the date and month column into int datatype from object datatype
ds[['month', 'year']] = ds[['month', 'year']].astype(str).astype(int)
```

*Figure 14.2: Converting the datatype of the column to **int** rather than string*

3.4 Limitations and Options

One of the major limitations posed by the dataset is the limited features given to work with to train the model on. After all the processing has been applied to the features that the model had to work with were **Mileage**, **Power**, **Make**, **Fuel Type**, **Transmission Type**, **State of Vehicle** and **Year**. However, when it comes to the sale of vehicles, there are countless other factors that will greatly affect the price of it, such as number of previous owners, vehicle condition, location, colour, accessories and service history **[10]**. If these factors were implemented, the predictive model could have more built out data to provide a more accurate method of evaluating a motorbike.

Additionally, the large quantities of missing data proved to be an issue as the data couldn't be dropped, therefore the data had to be entered to fill in the blank values, which will affect how accurate the predictions will be as the data entered is from within the dataset, rather than actual factual data.

Model Development

4.1 The predictive modelling process

4.1.1 Linear Regression

Initially the linear regression model had the worst R2, MSE and MAE scores but after clearing the outliers and making the dataset more usable, the metrics improved greatly as the R2 score went up from -0.7 to now a 0.3 (as you can see in *Figure 9.1.3*) – although a R2 score of 0.3 is a poor score **[7]** when it comes to predicting values, it is still a drastic improvement from the previous score.

```
# initalise and train the linear regression model
lrModel = LinearRegression()
lrModel.fit(xTrain, yTrain)

# make predictions
yPrediction = lrModel.predict(xTest)
```

Figure 15.1.1: Initializing the linear regression model and

```
# evaluating the model
mse = mean_squared_error(yPrediction, yTest)
sqrt_mse = np.sqrt(mse)
mae = mean_absolute_error(yPrediction, yTest)
r2 = r2_score(yTest, yPrediction)
```

Figure 15.1.2: Evaluating the linear regression model

```
MSE : 23448961.984
MSE_SQRT : 4842.413
MAE : 3530.499
R2 Score : 0.295
Coefficient : 0.309
```

Figure 15.1.3: Output of the metrics

```
# showing statistics of the model
print("Intercept:", lrModel.intercept_,
      "\nCoeficcient:", lrModel.coef_,
      "\nScore of the training data:", lrModel.score(xTrain, yTrain),
      "\nScore of the test data:", lrModel.score(xTest, yTest))
```

Figure 15.1.4: Evaluating the training and test data for the model

```
Intercept: [177548.00332845]
Coeficcient: [[-6.73622580e-02  5.41959372e+01 -1.52514278e+01 -6.46223166e+02
   9.74048691e+02 -9.59503505e+02 -8.20402174e+01]]
Score of the training data: 0.309435760550296
Score of the test data: 0.295066872506091
```

Figure 15.1.5: Output of the training and test data

4.1.2 Decision Tree Regression

The decision tree regression model initially had one of the better R2, MSE and MAE scores and after clearing the outliers and making the dataset more usable, the metrics improved somewhat as the R2 score went up from 0.2 to now a 0.5 (as you can see in *Figure 9.2.3*) – although a R2 score of 0.5 is considered a fair score **[7]** when it comes to predicting values, it is still a minor improvement from the previous score. However, this is still not the best model.

```
# initialise and train the Decision Tree Regressor model
dtModel = DecisionTreeRegressor(random_state=42)
dtModel.fit(xTrain, yTrain)

# make predictions
y_pred_dt = dtModel.predict(xTest)
```

Figure 15.2.1: Initializing the decision tree model

```
# evaluating the model
mse_dt = mean_squared_error(yTest, y_pred_dt)
rmse_dt = np.sqrt(mse_dt)
mae = mean_absolute_error(y_pred_dt, yTest)
r2_dt = r2_score(yTest, y_pred_dt)
```

Figure 15.2.2: Evaluating the decision tree model

```
Decision Tree Regressor Evaluation Metrics:
Mean Squared Error (MSE): 15346950.89
Root Mean Squared Error (RMSE): 3917.52
Mean Absolute Error (MAE): 2110.60
R-squared (R2 Score): 0.54
```

Figure 15.2.3: Output of the metrics

```
# showing statistics of the model
print("Score of the train data:",
      dtModel.score(xTrain, yTrain),
      "\nScore of the test data:",
      dtModel.score(xTest, yTest))
```

Figure 15.2.4: Evaluating the training and test data for the model

```
Score of the train data: 0.9623127313254756
Score of the test data: 0.538633134606625
```

Figure 15.2.5: Output of the training and test data

4.1.3 Random Forest Regression

The random forest regression model initially had the best R2, MSE and MAE scores and after clearing the outliers and making the dataset more usable, the metrics improved as the R2 score went up from 0.34 to now a 0.7 (as you can see in *Figure 9.3.3*) – although a R2 score of 0.7 is considered a good score **[7]** in some areas, it could still be improved upon by altering the dataset to suit this model more. This model is by far the best model out of all 4 models as it has provided the best accuracy metrics.

```
# initialise and train the Random Forest Regressor model
rfModel = RandomForestRegressor(random_state=42, n_estimators=100)
rfModel.fit(xTrain, yTrain)

# make predictions
y_pred_rf = rfModel.predict(xTest)
```

Figure 15.3.1: Initializing the random forest model

```
# evaluating the model
mse_rf = mean_squared_error(yTest, y_pred_rf)
rmse_rf = np.sqrt(mse_rf)
mae = mean_absolute_error(y_pred_rf, yTest)
r2_rf = r2_score(yTest, y_pred_rf)
```

Figure 15.3.2: Evaluating the random forest model

```
Random Forest Regressor Evaluation Metrics:
Mean Squared Error (MSE): 10104375.75
Root Mean Squared Error (RMSE): 3178.74
Mean Absolute Error (MAE): 1900.77
R-squared (R2 Score): 0.70
```

Figure 15.3.3: Output of the metrics

```
# showing statistics of the model
print("Score of the train data:", rfModel.score(xTrain, yTrain),
      "\nScore of the test data:", rfModel.score(xTest, yTest))
```

Figure 15.3.4: Evaluating the training and test data for the model

```
Score of the train data: 0.9262014748290451
Score of the test data: 0.6962377607520622
```

Figure 15.3.5: Output of the training and test data

4.1.4 Lasso Regression

The random forest regression model initially had the worst R2, MSE and MAE scores and after clearing the outliers and making the dataset more usable, the metrics have somewhat improved as the R2 score went up from -10.2 to now a -1.2 (as you can see in *Figure 9.4.3*), but this model is unusable as the R2 score is negative (when it should be between 0 – 1 to be classified as usable) yet it contains a similar MSE, RMSE and MAE to the linear regression model (shown in *Figure 9.1.3)*. This model would require a lot of work to be done on the dataset for it to function better and will need stricter processing techniques. This model is by far the worst model out of all 4 models as it has provided the worst accuracy metrics.

```
# initialising and fitting the training data onto the model
laModel = Lasso(alpha= 0.4)
laModel.fit(xTrain, yTrain)

# make predictions
yPred = laModel.predict(xTest)
```

Figure 15.4.1: Initializing the lasso model

```
# evaluating the model
mse = round(mean_squared_error(yPred, yTest, squared= False),  2 )
rmse = np.sqrt(mse)
mae = mean_absolute_error(yPred, yTest)
r2 = r2_score(yPred, yTest)
```

Figure 15.4.2: Evaluating the lasso model

```
Lasso Regression Evaluation Metrics
Mean Squared Error :  4842.37
Root Mean Squared Error 69.58713961645499
Mean Absolute Error :  3530.468294790894
R2_Squared   -1.263307574257551
```

Figure 15.4.3: Output of the metrics

```
# showing statistics of the model
print("Score of the train data:", laModel.score(xTrain, yTrain),
      "\nScore of the test data:", laModel.score(xTest, yTest))
```

Figure 15.4.4: Evaluating the training and test data for the model

```
Score of the train data: 0.3094356516971968
Score of the test data: 0.2950790620331286
```

Figure 15.4.5: Output of the training and test data

Conclusion

5.1 Summary of results

Each of the models were presented with varied data to get used to how each of the different models work and how they differentiate. But for testing purposes, to get a proper comparison between all the models, each of the models were given the exact same parameters to compare how different the outputs would be – as all the models had different R2 scores.

The liner and lasso regression models (shown in *Figures 16.1* and *16.2*) by far had the least accurate outputs, although their outputs only differed by around €10, it is still a drastic difference compared to the outputs of the random forest model (shown in *Figure 16.3*) or the decision tree model (shown in *Figure 16.4*). The difference in output between the random forest model and the decision tree model is around €1200, however, the random forest model had the highest accuracy metrics compared to all of the models, as such the most believable value predicted by all of the models would be the random forest model.

```
# using linear regression to predict the value of a motorbike based on given parameters
val1 = lrModel.predict([[2500, 125, 111, 1, 1, 1, 2019]]) # change variables here to have different values on output
value1 = val1[0][0]
print(f'Predicted Value of Vehicle = €{value1:.2f}')
```

```
Predicted Value of Vehicle = €16190.30
```

Figure 16.1: Output of the linear regression model

```
# using lasso regression to predict the value of a motorbike based on given parameters
val4 = laModel.predict([[2500, 125, 111, 1, 1, 1, 2019]]) # change variables here to have different values on output
value4 = val4[0]
print(f'Predicted Value of Vehicle = €{value4:.2f}')
```

```
Predicted Value of Vehicle = €16183.76
```

Figure 16.2: Output of the lasso regression model

```
# using random forest regression to predict the value of a motorbike based on given parameters
val3 = rfModel.predict([[2500, 125, 111, 1, 1, 1, 2019]]) # change variables here to have different values on output
value3 = val3[0]
print(f'Predicted Value of Vehicle = €{value3:.2f}')
```

```
Predicted Value of Vehicle = €7304.23
```

Figure 16.3: Output of the random forest regression model

```
# using decision tree regression to predict the value of a motorbike based on given parameters
val2 = dtModel.predict([[2500, 125, 111, 1, 1, 1, 2019]]) # change variables here to have different values on output
value2 = val2[0]
print(f'Predicted Value of Vehicle = €{value2:.2f}')
```

```
Predicted Value of Vehicle = €6150.00
```

Figure 16.4: Output of the decision tree regression model

5.2 Reflection on Individual Learning

The one of the main issues I faced when developing this predictive AI model, was finding the correct regression algorithms to use on this dataset, as prior to this project, my knowledge of regression models was limited to only linear regression.

Another main issue that I was faced with was dealing with outliers and invalid data in my dataset and learning how drastically they affect the outcome of a model. Initially, during the process of cleaning the dataset, the outliers were not removed, which greatly affected the accuracy of each of the regression models, as the MSE score was in the hundreds of millions and the R2 score was negative. Multiple methods (like scaling) were used to try to improve the accuracy of the models, but it was due to the outliers being present that the models were so inaccurate. But eventually, after trial and error the outliers were found and removed, greatly increasing the accuracy of the model.

References

[1] - Gupta, S. (2021). *Regression vs. Classification in Machine Learning: What's the Difference?* [online] Springboard Blog. Available at: https://www.springboard.com/blog/data-science/regression-vs-classification/.

[2] - pandas.pydata.org. (n.d.). *pandas.DataFrame.iloc — pandas 1.3.4 documentation*. [online] Available at: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iloc.html.

[3] - Amazon Web Services, Inc. (n.d.). *What is Linear Regression? - Linear Regression - AWS*. [online] Available at: https://aws.amazon.com/what-is/linear-regression/.

[4] - GeeksForGeeks (2018). *Python | Decision Tree Regression using sklearn*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/python-decision-tree-regression-using-sklearn/.

[5] - Dutta, A. (2019). *Random Forest Regression in Python - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/random-forest-regression-in-python/.

[6] - IBM (2024). *What is lasso regression? | IBM*. [online] www.ibm.com. Available at: https://www.ibm.com/topics/lasso-regression.

[7] - Fernando, J. (2024). *R-Squared: Definition, Calculation Formula, Uses, and Limitations*. [online] Investopedia. Available at: https://www.investopedia.com/terms/r/r-squared.asp.

[8] - Frost, J. (2021). *Mean Squared Error (MSE)*. [online] Statistics By Jim. Available at: https://statisticsbyjim.com/regression/mean-squared-error-mse/.

[9] - Ahmed, M.W. (2023). *Understanding Mean Absolute Error (MAE) in Regression: A Practical Guide*. [online] Medium. Available at: https://medium.com/@m.waqar.ahmed/understanding-mean-absolute-error-mae-in-regression-a-practical-guide-26e80ebb97df.

[10] - Bennetts BikeSocial Membership. (2020). *Motorcycle Price Guide | What impacts the price of your bike?* [online] Available at: https://www.bennetts.co.uk/bikesocial/news-and-views/advice/buying-and-selling-a-bike/motorcycle-price-guide [Accessed 9 Dec. 2024].