Machine Learning Model Building and Evaluation AutoTrader Car Listings Dataset

Christian Jordan - 14061768

20 January 2023

1 Prior Works

Some of the early works in this project are built in the prior ”Data Exploration and Analysis of AutoTrader Car Listings Dataset” (C.Jordan, 2023). Improvements have been made during the data exploration phase for specific use cases of machine learning models that are built throughout this report.

2 Data Understanding and Exploration

Begin by loading the data into a Pandas Dataframe and removing any duplicate observations that exist. The dataset has 393,378 observations once duplicates have been removed, with 11 features. From the initial dataframe, notice missing values in ”reg” and ”year”, where ”year” is null when condition = ”NEW”.

```
# Read the data into dataframe, print first few rows.
cars = pd.read_csv('adverts.csv', index_col=['public_reference'])

# Drop any duplicate observations from the dataset
cars = cars.drop_duplicates()

# View the first five rows of data.
cars.head()
```

	mileage	reg	colour	make	model	condition	year	price	type	car_van	fuel
public_reference											
202006039777689	0.00000	NaN	Grey	Volvo	XC90	NEW	NaN	73970	SUV	False	Petrol Plug-in Hybrid
202007020778260	108230.00000	61	Blue	Jaguar	XF	USED	2011.00000	7000	Saloon	False	Diesel
202007020778474	7800.00000	17	Grey	SKODA	Yeti	USED	2017.00000	14000	SUV	False	Petrol
202007080986776	45000.00000	16	Brown	Vauxhall	Mokka	USED	2016.00000	7995	Hatchback	False	Diesel
202007161321269	64000.00000	64	Grey	Land Rover	Range Rover Sport	USED	2015.00000	26995	SUV	False	Diesel

Table 1: First five rows of car listings dataset.

2.1 Meaning and Type of Features

View the type of features available in the car listings dataset and the meaning of the features.

- Mileage: The mileage of the vehicle, continuous numeric feature.

- Reg: The registration code of the vehicle, categorical feature (perhaps with some order).
- Colour: The colour of the vehicle, categorical feature.
- Make: The maker of the vehicle, categorical feature.
- Model: The model of the vehicle, categorical feature.
- Condition: The condition of the vehicle, binary categorical feature, ”NEW” or ”USED”.
- Year: The year of registration, floating point number (could be converted to int), numeric feature.
- Type: The body type of the vehicle, categorical feature.
- Car_van: Whether the vehicle is a car or van, boolean feature, True = ”Van”, False = ”Car”.
- Price: The target variable, continuous numeric feature.

```
# Print the number of rows and columns in the dataset
print('The shape of the dataset is: {}'.format(cars.shape))
print('')

# Change feature names so they are easier to access
cars = cars.rename(columns={
    'reg_code':'reg',
    'standard_colour':'colour',
    'standard_make':'make',
    'standard_model':'model',
    'vehicle_condition':'condition',
    'year_of_registration':'year',
    'body_type':'type',
    'crossover_car_and_van':'car_van',
    'fuel_type':'fuel'})

# Get feature information and data types
cars.info();
```

```
Output:
The shape of the dataset is: (393378, 11)

<class 'pandas.core.frame.DataFrame'>
Int64Index: 393378 entries, 202006039777689 to 201512149444029
Data columns (total 11 columns):
 #   Column                 Non-Null Count   Dtype
---  ------                 --------------   -----
 0   mileage                393254 non-null  float64
 1   reg_code               368286 non-null  object
 2   standard_colour        388242 non-null  object
 3   standard_make          393378 non-null  object
 4   standard_model         393378 non-null  object
 5   vehicle_condition      393378 non-null  object
 6   year_of_registration   366851 non-null  float64
 7   price                  393378 non-null  int64
 8   body_type              392560 non-null  object
 9   crossover_car_and_van  393378 non-null  bool
 10  fuel_type              392820 non-null  object
dtypes: bool(1), float64(2), int64(1), object(7)
memory usage: 33.4+ MB
```

2.1.1 Numeric Distributions

Table 2 shows the descriptive statistics for the numeric features in the dataset. From this, it can be found that:

- Mileage has a range of 0 - 999,999, with mean 38,514.83862, median 29,435.00 and standard deviation 34,758.488.
- Year has a range of 999 - 2020, with mean 2014.98779, median 2016 and standard deviation 7.97477.
- Price has a range of 120 - 9,999,999, with mean 17,177.78633, median 12,495.00 and standard deviation 46,827.96995.

	mileage	year	price
count	393254.00000	366851.00000	393378.00000
mean	38514.83862	2014.98779	17177.78633
std	34758.48800	7.97477	46827.96995
min	0.00000	999.00000	120.00000
25%	11510.00000	2013.00000	7450.00000
50%	29435.50000	2016.00000	12495.00000
75%	57630.00000	2018.00000	19990.00000
max	999999.00000	2020.00000	9999999.00000

Table 2: Descriptive statistics of numeric features.

For mileage and price, the standard deviation in relation to the mean is very large, suggesting that the data is skewed. This can be visualised in Figures 1 and 3, where the tails appear very long for both features, indicating a positive skew. Similarly, year also appears skewed however it a negative skew. Check this using skew and kurtosis metrics for the distributions.

```
# Define function for calculating skewness and kurtosis
def get_skew(df, col):
    print('{} Skewness :{} '.format(col, df[col].skew()))
    print('{} Kurtosis :{} '.format(col, df[col].kurt()))

# Get skewness and kurtosis of mileage feature
get_skew(cars, 'mileage')

# Get skewness and kurtosis of price feature
get_skew(cars, 'price')

# Get skewness and kurtosis of year feature
get_skew(cars, 'year')
```

```
Output:
mileage Skewness :1.4499462144066446
mileage Kurtosis :7.4389873055707625

price Skewness :154.09492719087976
price Kurtosis :31805.275390658902

year Skewness :-87.94270712440368
year Kurtosis :10975.331329677867
```

As anticipated, the data for mileage and price are positively skewed, with skewness > 1 indicating large skews. Also, for year, the negative skew is large. This could indicate issues with outliers, erroneous values etc.

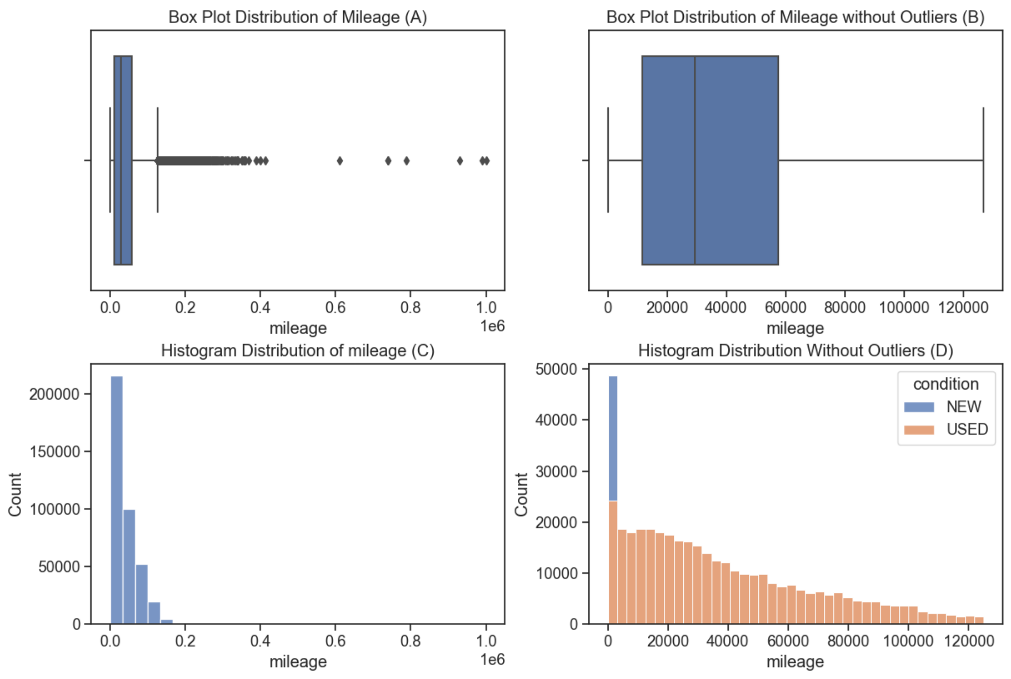

Figure 1: Distribution of mileage feature.

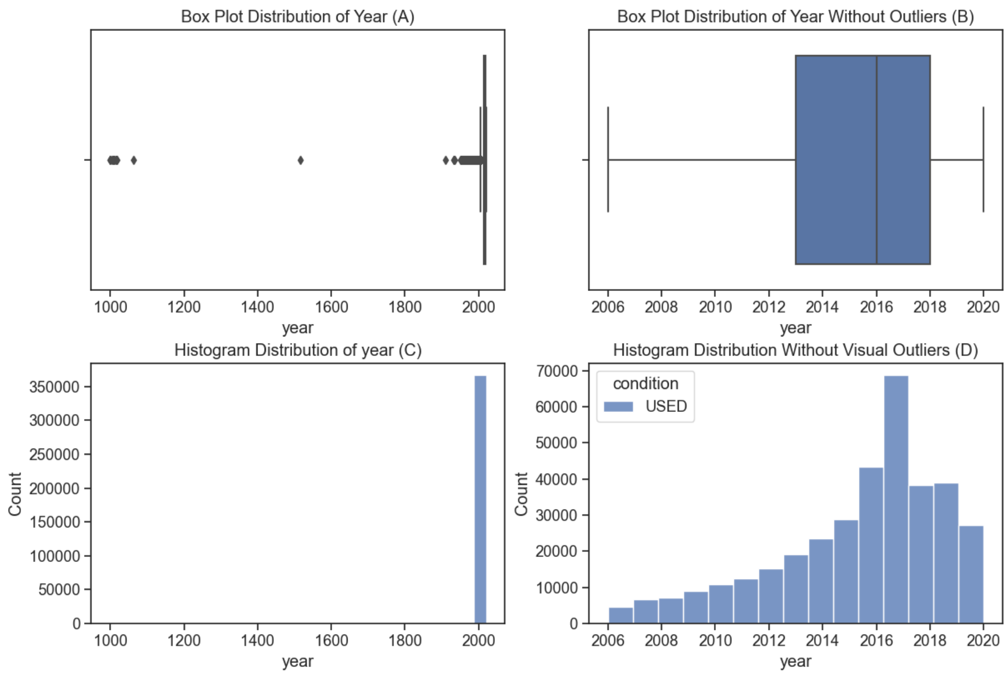

Figure 2: Distribution of year feature.

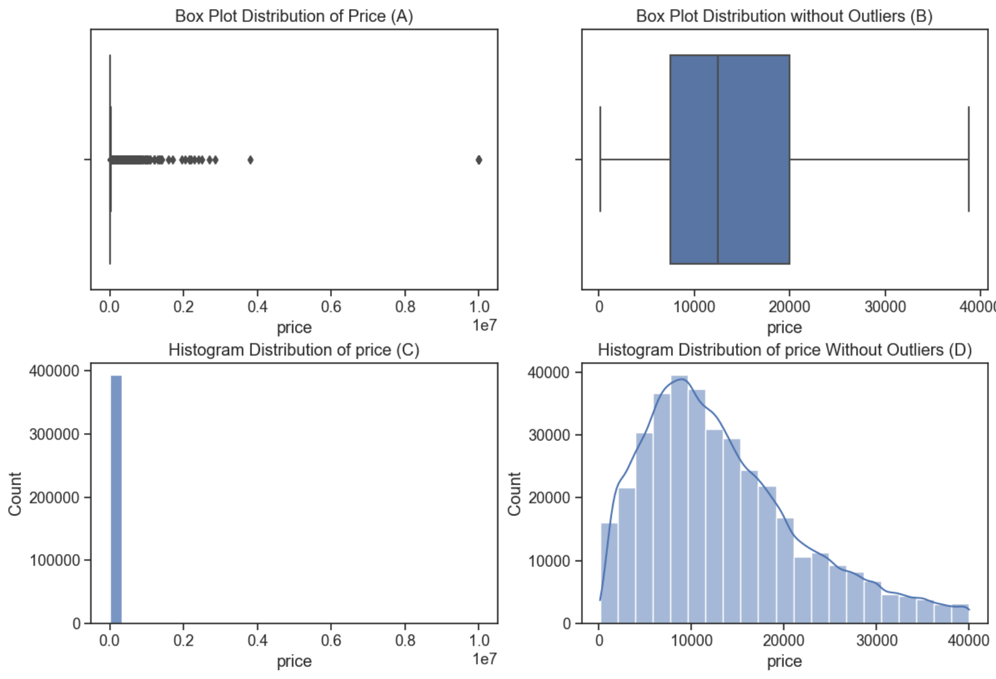

Figure 3: Distribution of price feature.

2.1.2 Categorical Distributions

The distributions of categorical data can be analysed using count plots as shown in figure 4. Things to note:

- In-balanced observations for a lot of the categorical variables including colour, condition, car_van, fuel.
- Petrol and Diesel vehicles make up approximately 350,000 observations from 393,000.
- Most observations are made up of the first six colours in Figure 4.
- There is less than 0.5% of vehicles that are vans.

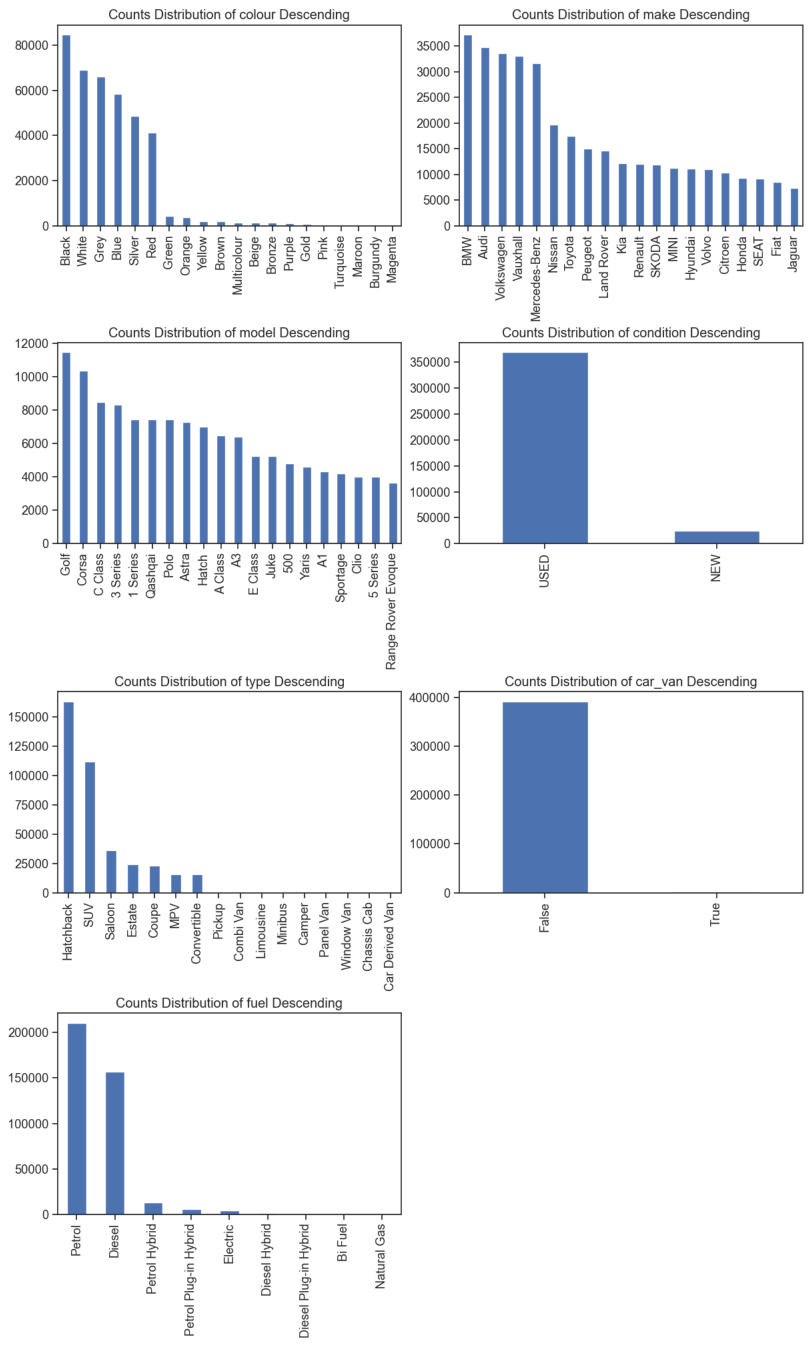

Figure 4: Distribution of categorical features.

2.2 Analysis of Predictive Power of Features

2.2.1 Correlation

Correlation can be an important metric in deciding which features can predict others, however should not be reliant upon in situations where there is a lot of categorical data. Correlation coefficients imply linear relationships between variables, whereas a linear model may not be best suited to this particular problem.

That doesn't mean it cannot provide insight to the model and help decide which features to include alongside some other feature prediction tools such as groupby aggregation, visualisations and the predictive power score in section 2.2.2.

```python
# Create annotated correlation heatmap
sns.heatmap(cars.drop(columns=['new']).corr(), annot=True, cmap="Blues", fmt='.2f');
```

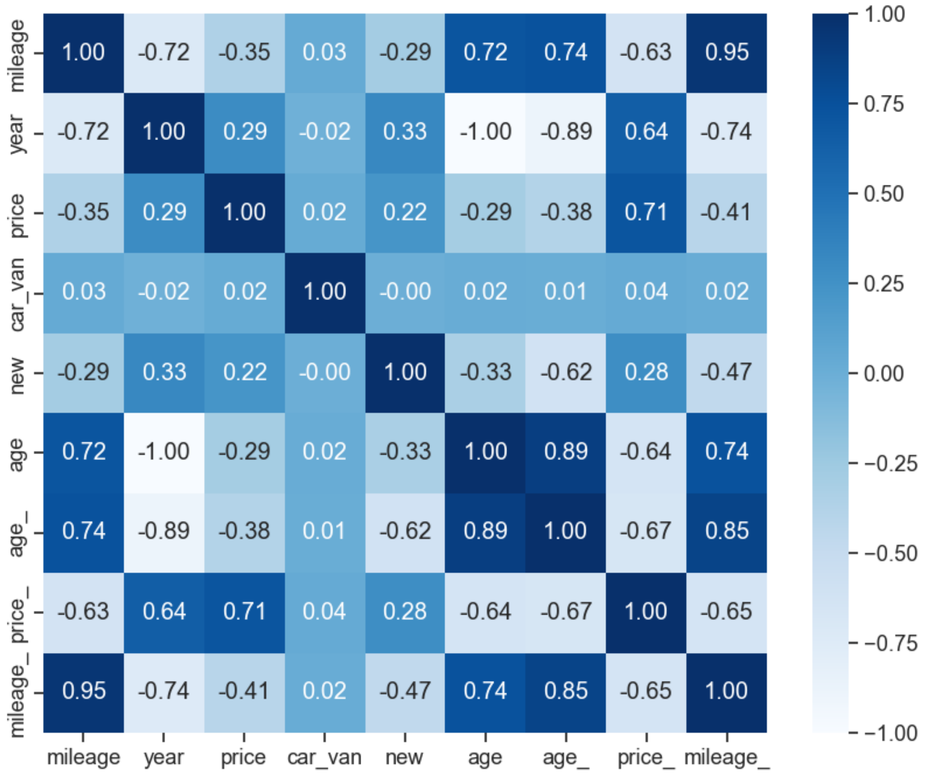

Figure 5: Correlation matrix of features.

From the correlation matrix in Figure 5, some of the best linear relationships exist with the log transformed price ”price_”. The correlations with mileage, year, new can all be improved with the transformation.

2.2.2 Predictive Power Score

The predictive power score (or ppscore) for a numeric target builds a regression model using each individual feature in the dataset to predict the target variable (price). It uses the DecisionTreeRegressor() from Sklearn to evaluate the mean absolute error from each of these models.

This provides a enhancement to the correlation matrix as it also provides predictive power of categorical features against the target.

```
# Build predictive power score table
pred_power = pps.predictors(cars.drop(columns=['price_']), 'price').sort_values(by='
                                                         model_score')
pred_power
```

	x	y	ppscore	case	is_valid_score	metric	baseline_score	model_score	model
0	age_	price_	0.31827	regression	True	mean absolute error	0.28164	0.19201	DecisionTreeRegressor()
1	year	price_	0.31827	regression	True	mean absolute error	0.28164	0.19201	DecisionTreeRegressor()
2	age	price_	0.31795	regression	True	mean absolute error	0.28164	0.19209	DecisionTreeRegressor()
3	year_bins	price_	0.27394	regression	True	mean absolute error	0.28164	0.20449	DecisionTreeRegressor()
4	model	price_	0.26867	regression	True	mean absolute error	0.28164	0.20597	DecisionTreeRegressor()
5	make	price_	0.14643	regression	True	mean absolute error	0.28164	0.24040	DecisionTreeRegressor()
6	type	price_	0.09151	regression	True	mean absolute error	0.28164	0.25587	DecisionTreeRegressor()
7	condition	price_	0.03949	regression	True	mean absolute error	0.28164	0.27052	DecisionTreeRegressor()
8	new	price_	0.03949	regression	True	mean absolute error	0.28164	0.27052	DecisionTreeRegressor()
9	mileage_	price_	0.02646	regression	True	mean absolute error	0.28164	0.27419	DecisionTreeRegressor()
10	fuel	price_	0.02568	regression	True	mean absolute error	0.28164	0.27441	DecisionTreeRegressor()
11	mileage	price_	0.02482	regression	True	mean absolute error	0.28164	0.27465	DecisionTreeRegressor()
12	colour	price_	0.01192	regression	True	mean absolute error	0.28164	0.27829	DecisionTreeRegressor()
13	car_van	price_	0.00000	regression	True	mean absolute error	0.28164	0.28204	DecisionTreeRegressor()

Table 3: Predictive power of features.

Table 3 shows the results of the PPScore for log transformed price. It gives similar scores to features that have been derived from each other, such as ”age_” and ”year”. Removing the feature with lower ”ppscore” if there is co-linearity from transformations such as the log transformations in section 3.

Building a new table of values and visualising in a bar-plot is Figure 6 shows the predicted feature importance, and what should be used in models moving forwards.

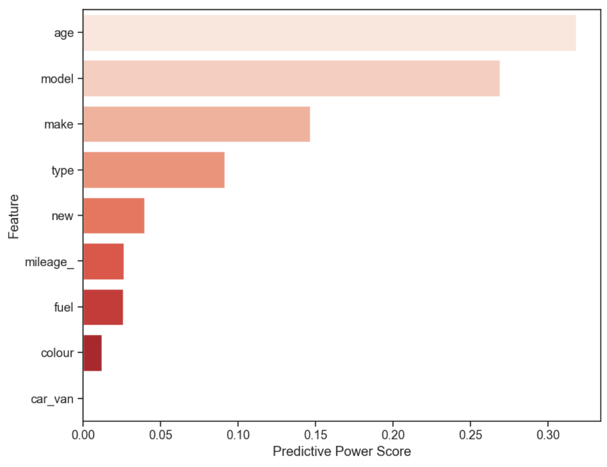

Figure 6: Barplot of predictive power score.

2.3 Data Processing for Exploration and Visualisation

With a large number of observations in the dataset, it can be difficult to visualise numeric features using scatter-plots. One technique that can be utilised to correct this is binning.

For example, binning year and mileage and comparing the overall summary statistics is one way to achieve visualisation of the predictive power of these variables.

```
# Split mileage data into 5 equally spaced bins.
mileage_labels = ['Very Low', 'Low', 'Medium', 'High', 'Very High']
cars['mileage_bins'] = pd.qcut(cars['mileage'], q=[0, 0.2, 0.4, 0.6, 0.8, 1], labels=
                                          mileage_labels)

# Create subplots for visualisation
fig, axs = plt.subplots(1, 2, figsize=(12,6), constrained_layout=True)

# Create box-plots of year_bins and mileage_bins against price
sns.boxplot(x='year_bins', y='price', data=cars, ax=axs[0], showfliers=False);
sns.boxplot(x='mileage_bins', y='price', data=cars, ax=axs[1], showfliers=False);
```

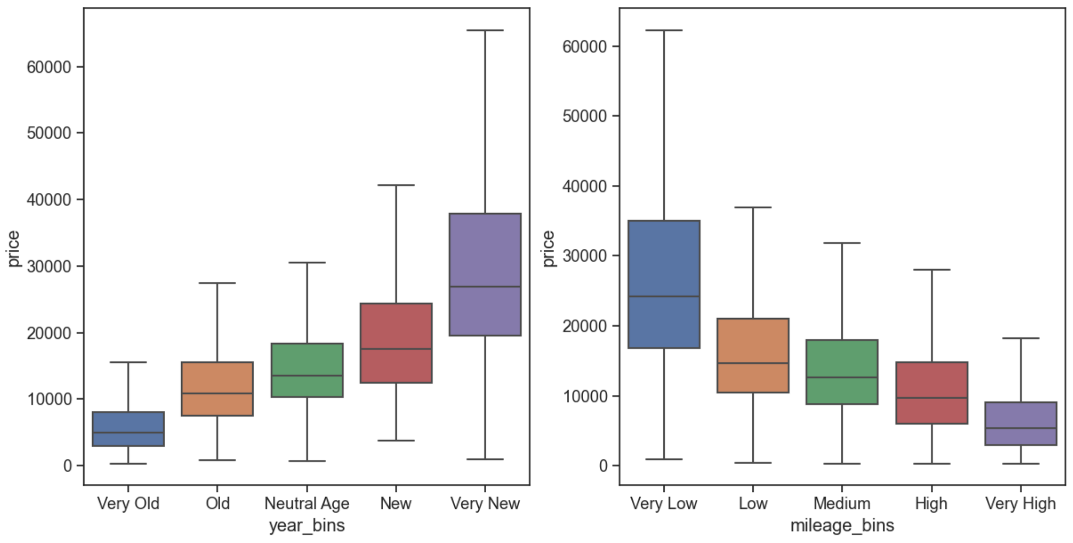

Figure 7: Box-plots of year_bins and mileage_bins by price

3 Data Processing for Machine Learning

3.1 Dealing with Missing Values, Outliers and Noise

3.1.1 Missing Values

There are many ways to impute missing values in the data, such as with the mean, median, mode, predicting the value using groupby means to transform the data or even using machine learning methods themselves to predict values. Table 4 shows the missing values in the dataset.

	0
mileage	124
reg	25092
colour	5136
make	0
model	0
condition	0
year	26527
price	0
type	818
car_van	0
fuel	558

Table 4: Missing values in dataset.

Missing Values in Mileage

To deal with the missing values in mileage, notice from the correlation matrix in 2.2.1 that mileage is highly (negatively) correlated with year.

Dividing year up into equal bins to then use these "year_bins" to fill the missing values of mileage will provide a more accurate representation than just filling with the mean or median themselves. Table 5 shows how the mean and median mileage varies by the age of the vehicle divided into bins for year.

```
# Group years into bins
year_labels = ['Very Old', 'Old', 'Neutral Age', 'New', 'Very New']
cars['year_bins'] = pd.qcut(cars['year'], q=[0, 0.2, 0.4, 0.6, 0.8, 1], labels=
                                            year_labels)

# Check mean and median values of mileage for each year_bin.
cars.groupby('year_bins')['mileage'].agg(['mean', 'median'])
```

	mean	median
year_bins		
Very Old	77481.00230	74653.50000
Old	45422.33098	41000.00000
Neutral Age	28831.97599	25149.00000
New	14487.38203	12162.50000
Very New	1571.48584	10.00000

Table 5: Groupby function of mileage mean and median by year bin.

It is clear that this would be a better predictor of mileage, therefore transform the null values to match.

```
# Replace null values with the grouped mileage median of 'year_bins'
cars['mileage'] = cars.groupby('year_bins')['mileage'].transform(lambda x: x.fillna(x
                                                     .median()))
```

3.1.2 Removing Extreme Outliers

An earlier iteration of machine learning models in this project yielded negative results for extreme outliers within the dataset. Viewing the vehicles that exist as outliers in the 0.1% of data for year and price shows vehicles that are unusual and would therefore models struggle to predict accurately. Dealing with this can be problematic, as the outliers do carry some significant information, however as it is such a small sum of observations, they can safely be removed to improve the model for the majority of other observations.

It could be likely that there is some information that isn't in the dataset, specifically regarding the price outliers, such as custom alterations or needed repairs that cause the price of vehicles to be so far from the mean of the data.

```
def remove_outliers(df,columns):
    # Loop through specified columns
    for col in columns:
        print('Working on column: {}'.format(col))

        # Set bounds of lower and upper 0.1%
        lower, upper = np.percentile(df[col], [0.01, 99.99])

        print('Upper bound: {}, Lower Bound: {}'.format(upper, lower))

        # Print the number of outliers
        print('Number of values labelled: ', len(df.loc[(df[col] > upper) | (df[col]
                                                  < lower)]))

        # Set outlier column to true if outlier
        df = df.loc[(df[col] <= upper) & (df[col] >= lower)]

    return df

# Run function to label outliers and assign to cars dataframe
cars = remove_outliers(cars, ['year', 'price'])
```

```
Output:
Working on column: year
Upper bound: 2021.0, Lower Bound: 1958.0
Number of values labelled:  39
Working on column: price
Upper bound: 932942.8899988707, Lower Bound: 300.0
Number of values labelled:  72
```

3.2 Feature Engineering, Data Transformations, Feature Selection

3.2.1 Transformations

One of the main transformations of the dataset has been applied to the price. Application of log base 10 transforms the distribution of the target variable to approximate a normal distribution as shown below.

```
# Transform price using log base 10
cars['price_'] = np.log10(cars['price'])
```

3.2.2 Train/Test Split

For this project two train/test splits have been created, one on the whole dataset and the other on a sample of 40,000 observations. Building the models and searching for the best parameters can be sped up using this method, however it is important to validate the results on the entire data.

```
# Create split for selected features
features = ['age','model','make','type','new','fuel','mileage_','colour','car_van']
X = cars[features]
y = cars['price_']

# Create a train/test split for model building and validation
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.25, random_state=0
)

print('Shape of training data: {}'.format(X_train.shape))
print('Shape of test data: {}'.format(X_test.shape))
```

```
Output:
Shape of training data: (294949, 9)
Shape of test data: (98317, 9)
```

And for the sample data:

```
# Take sample of the dataset for faster search times
sample = cars.sample(40000, random_state=0)
X_sample = sample[features]
y_sample = sample['price_']

# Create a sample data train/test split for model building and evaluation
SX_train, SX_test, Sy_train, Sy_test = train_test_split(
    X_sample, y_sample,
    test_size = 0.25, random_state=0
)

print('Shape of sample training data: {}'.format(SX_train.shape))
print('Shape of sample test data: {}'.format(SX_test.shape))
```

```
Output:
Shape of sample training data: (30000, 9)
Shape of sample test data: (10000, 9)
```

3.3 Encoding Using Pipelines and Transformers

3.3.1 One Hot Encoding

One hot encoding works well on all categorical variables, however if the feature has a large number of unique values within it, the dimensions of the data can become very large and models consequently take much longer to train. Therefore it is best to limit the encoded variables to those with less unique values, or reduce the number of unique values before encoding with this method.

```
# Show number of unique values in each feature
for feature in cars.columns:
    print(feature, len(cars[feature].unique()))
```

```
Output:
age 64
model 1146
make 108
type 16
new 2
fuel 8
mileage_ 80627
colour 23
car_van 2
```

```
# Define features for one hot encoding
onehot_features = ['type','fuel']

# Build transformer for one hot encoding
onehot_transformer = Pipeline(
    steps=[
        ("ohe", OneHotEncoder(handle_unknown="ignore"))
    ]
)

# Fit the sample training data
onehot_transformer.fit(SX_train[onehot_features])
```

The below code will trial the transformer to check how many variables have been created from the sample data, this may vary with the full training data. 22 features

```
# Trial the transformer
onehot = onehot_transformer.fit_transform(SX_train[onehot_features])
print('The shape of the array: ',onehot.todense().shape)
onehot.todense()
```

```
Output:
The shape of the array:  (30000, 22)
[155]:
matrix([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 1., 0., 0.],
        [0., 0., 0., ..., 1., 0., 0.],
        ...,
        [0., 0., 0., ..., 1., 0., 0.],
        [0., 0., 0., ..., 0., 0., 1.],
        [0., 0., 0., ..., 1., 0., 0.]])
```

3.3.2 Target Encoding

Target encoding by Category-Encoders provides a framework for replacing the values categorical feature with the mean of the target column. It is important that the encoder is fit on the training data and only this is used to transform the testing data. The Pipeline feature by Sk-learn performs this fitting and transformation for us.

```
# Define features for target encoding
target_features = ['make','model','colour']

# Build transformer for target encoding
target_transformer = Pipeline(
    steps=[
        ("tar", TargetEncoder()),
        ("scaler", StandardScaler())
    ]
)

# Fit the sample training data
target_transformer.fit(SX_train[target_features], Sy_train)
```

Again, check the transformer works on the sample training data and see the shape of the returned array.

```
# Check all is working correctly
target = target_transformer.fit_transform(SX_train[target_features], Sy_train)
print('The shape of the array: ', target.shape)
target
```

```
Output:
The shape of the array:  (30000, 3)
[902]:
array([[ 0.5924425 ,  1.40730523,  0.48929593],
       [-0.97413165, -0.61588901,  0.48929593],
       [-0.91810609, -0.88072704,  0.48929593],
       ...,
       [-0.54433077, -0.41965244,  0.63935539],
       [ 1.83736733,  1.68083496,  1.09900129],
       [-1.28215743, -0.86381407, -0.33177613]])
```

3.3.3 Numeric Scaling

The numeric data should be scaled for our models so that the weights are not disproportionate causing algorithms to give higher feature importance to such features.

```
# Define numeric features for scaling
num_features = ['mileage_','age']

# Building transformer for numeric features
num_transformer = Pipeline(
    steps=[
        ("scaler", StandardScaler())
    ]
)

# Fit the sample training data
num_transformer.fit(SX_train[num_features])
```

Once more, check the transformer is working as it should and get the shape of the array.

```
# Check all is working correctly
num = num_transformer.fit_transform(SX_train[num_features])
print('The shape of the array: ', num.shape)
num
```

```
Output:
The shape of the array:  (30000, 2)
[904]:
array([[-0.31307623, -0.34823817],
       [-0.46527848, -0.34823817],
       [ 0.96359991,  1.0225084 ],
       ...,
       [-0.62303836, -0.57669593],
       [-0.94074363, -0.80515369],
       [-0.60655484, -0.34823817]])
```

3.3.4 Finalising Pipeline

The Sk-Learn framework for building pipelines can be utilised to bring all of these encoders and transformers together in a single transformation that can be applied before the use of any algorithm. This can make it easier to make amendments to transformations for all models at once.

```
# Build a preprocessor to run numeric, one hot and target transformers
preprocessor = ColumnTransformer(
    transformers=[
        ("num", num_transformer, num_features),
        ("onehot", onehot_transformer, onehot_features),
        ("target", target_transformer, target_features),
    ],
    remainder='passthrough'
)

# Define function pipeline to make building Pipelines easier
def pipeline(reg):
    pipeline = Pipeline(steps=[("preprocessor", preprocessor), ("regressor", reg)])
    return pipeline
```

Trial the pipeline to ensure all variables are transformed like they should be and get resulting shape of returned array.

```
# Trial the preprocessor
pre = preprocessor.fit_transform(SX_train, Sy_train)
print('Shape of the array: ', pre.shape)
pre
```

```
Output:
Shape of the array:  (30000, 29)
[912]:
array([[-0.31307623, -0.34823817,  0.        , ...,  0.48929593,
         0.        ,  0.        ],
       [-0.46527848, -0.34823817,  0.        , ...,  0.48929593,
         0.        ,  0.        ],
       [ 0.96359991,  1.0225084 ,  0.        , ...,  0.48929593,
         0.        ,  0.        ],
       ...,
       [-0.62303836, -0.57669593,  0.        , ...,  0.63935539,
         0.        ,  0.        ],
       [-0.94074363, -0.80515369,  0.        , ...,  1.09900129,
         0.        ,  0.        ],
       [-0.60655484, -0.34823817,  0.        , ..., -0.33177613,
         0.        ,  0.        ]])
```

4 Model Building

4.1 Baseline Models

For reference, build models using the DummyRegressor() and a basic linear regression model so that we can compare the evaluation metrics of models with this. Section 5 will talk about these evaluation metrics more and compare the models based on these.

PrettyTable is a Python library to save the results of these models. The function "add_transformed_row" converts the predictions of log transformed price back to normal price for evaluation.

```
# Import library for recording evaluation metrics
from prettytable import PrettyTable
table = PrettyTable()
table.field_names = ['Model', 'MAE', 'MAPE', 'RMSE', 'R2-Score']
table.sortby = "MAE"

# Define function for transforming price_ back to price
def add_transformed_row(reg, X, y, label, table):
    y_pred = 10 ** reg.predict(X)
    y_true = 10 ** y

    mae = np.round(mean_absolute_error(y_pred, y_true),3)
    mape = np.round(mean_absolute_percentage_error(y_pred, y_true),3)
    rmse = np.round(mean_squared_error(y_pred, y_true, squared=False),3)
    r2 = np.round(r2_score(y_pred, y_true),3)

    table.add_row([label, mae, mape, rmse, r2])

    return y_pred, y_true
```

```
# Dummy regressor to predict mean only
dummy = DummyRegressor(strategy='mean')
dummy.fit(SX_train, Sy_train)
y_pred, y_true = add_transformed_row(dummy, SX_test, Sy_test, 'Dummy Mean', table)

# Dummy regressor to predict median only
dummy_med = DummyRegressor(strategy='median')
dummy_med.fit(SX_train, Sy_train)

add_transformed_row(dummy_med, SX_test, Sy_test, 'Dummy Median', table)

# Simple linear regression model using mileage and age
reg = LinearRegression()

reg.fit(SX_train[['mileage_','age']], Sy_train)
y_pred, y_true = add_transformed_row(reg, SX_test[['mileage_','age']], Sy_test, '
                                          Linear Regression Baseline', table)
```

Table 6 provides the output of evaluation metrics calculated using the baseline models. When the models have been built in the next section it will become clearer how much these models lag behind the others.

Model	MAE	MAPE	RMSE	R2-Score
Linear Regression Baseline	7811.103	0.776	19372.033	-5.744
Dummy Median	9985.309	0.793	21437.137	-1.3889071518350144e+32
Dummy Mean	9998.253	0.835	21577.216	0.0

Table 6: Evaluation table of baseline models.

4.2 Algorithm Selection, Model Instantiation and Configuration

There are a number of regression algorithms that can be used for this problem. Noted in section 1, the suitors for predicting price can be categoric and therefore may benefit from ensemble methods based on decision trees rather than linear models.

The list of regression models used in this project are:

- Linear Regression
- Ridge Regression
- SVM Regressor
- Decision Tree
- Random Forest
- Light GBM Regressor
- XGBoost Regressor

4.2.1 Building Models on Sample of Dataset

```
# Build linear regression model
reg = pipeline(LinearRegression())
reg.fit(SX_train, Sy_train)

# Add linear regression model to evaluation table
y_pred, y_true = add_transformed_row(reg, SX_test, Sy_test, 'Linear Regression (1)',
                                     table)
```

Repeating this function for the remaining models in the list outlined in 4.2, we get the following table of evaluation metrics.

Model	MAE	MAPE	RMSE	R2-Score
XGBoost	2278.218	0.143	8595.854	0.772
Random Forest	2454.277	0.159	9450.009	0.698
LGB Regressor	2760.451	0.164	9767.943	0.667
Decision Tree	3177.808	0.216	13115.642	0.564
SVM Regression	3265.92	0.193	10432.647	0.589
Ridge Regression	3944.588	0.286	11477.589	0.569
Linear Regression	3945.225	0.286	11478.149	0.569
Linear Regression Baseline	7811.103	0.776	19372.033	-5.744
Dummy Median	9985.309	0.793	21437.137	-1.3889071518350144e+32
Dummy Mean	9998.253	0.835	21577.216	0.0

Table 7: Evaluation of models without parameter tuning.

Table 7 shows how much even the linear regression model, which is one of the most basic algorithms, has improved with the new data it has access too.

So far, with mean absolute error (MAE) being the main metric for consideration, XGBoost seems to be the best at representing the dataset and predicting price. However, this is only on the sample data and should be treated carefully.

4.3 Overfit/Underfit Trade-Off

One way in which to determine correct parameters of models such as Decision Trees is to increase the depth in which it determines the fit of the training data. As it increases, compare with the error evaluation metrics on the test data. Once the error of test data stagnates, training error will continue to decrease, hence the overfitting. This can be visualised below in Figure 8.

```
# Set variables
depth = range(2, 50, 2)
train_mae = []
test_mae = []

# Loop through max_depth and record train/test MAE
for dep in depth:
    dtree = pipeline(DecisionTreeRegressor(max_depth=dep))
    dtree.fit(SX_train, Sy_train)

    train_mae.append(mean_absolute_error(10**dtree.predict(SX_train), 10**Sy_train))
    test_mae.append(mean_absolute_error(10**dtree.predict(SX_test), 10**Sy_test))

# Create dataframe from values
fit_data = pd.DataFrame([list(depth), train_mae, test_mae]).transpose()
fit_data.columns = ['depth','train_mae','test_mae']

# Visualise the train/test evaluation metric MAE for overfitting
fig = plt.figure(figsize=(8,6))
sns.lineplot(y=train_mae, x=depth);
sns.lineplot(x=depth, y=test_mae);
plt.xticks(range(0,48,2))
```

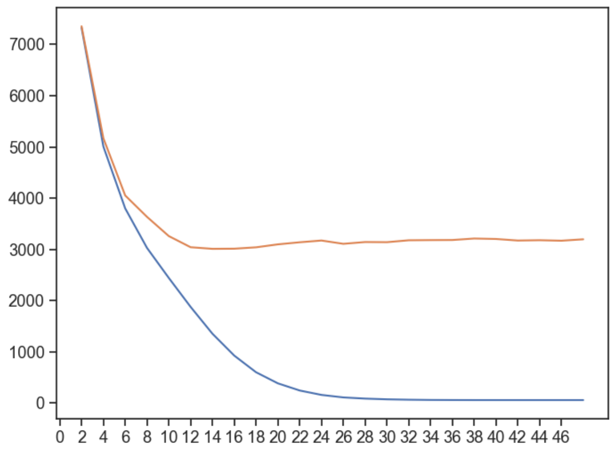

Figure 8: Train and Test MAE for changes in Decision Tree max_depth.

At approximately max_depth = 14, the curves diverge greatly and the test MAE is at it's lowest. Beyond this it begins to overfit the data.

4.4 Grid Search, Model Ranking and Selection

4.5 Grid Search

Grid search provides a solution to the overfit/underfit problem outlined in section 4. It searches through lists/ranges of different parameters (unique to each model) and provides the best score using cross validation techniques. That is, it splits the data into 5 folds (default number, can be altered) and performs the training and testing phase for a single fold using the other four and repeat this until complete. This can make grid search very computationally intensive using large datasets, therefore apply this to a sample of this dataset and use the findings of best parameters to apply to the entire dataset.

4.5.1 Random Forest

Random forest takes a large number of parameters that can be tuned finely to reveal the best possible model for the data. Given infinite time and computational resources, many more parameters would be tested on the whole training data during grid search, however this should be enough to evaluate these models.

```
# Set parameter values for search
grid_values = {'n_estimators':[10,50,100,200], 'max_depth':range(5,30,5), '
                                            min_samples_leaf': range(3,9)}

# Set evaluation metrics for results
scorers = ['neg_mean_absolute_error','neg_mean_absolute_percentage_error',
           'neg_root_mean_squared_error','r2']

# Initiate pipeline for grid search
```

```
forest = pipeline(GridSearchCV(RandomForestRegressor(), param_grid = grid_values,
                                          scoring=scorers, refit='
                                          neg_mean_absolute_error'))

# Fit the sample data
forest.fit(SX_train, Sy_train)

# Set columns for retrieval of results
cols_grid = ['rank_test_neg_mean_absolute_error','param_max_depth',
'param_min_samples_leaf', 'param_n_estimators',
             'mean_test_neg_mean_absolute_error',
             'mean_test_neg_mean_absolute_percentage_error',
             'mean_test_neg_root_mean_squared_error',
             'mean_test_r2']

# Print the resulting dataframe with best result first
results = pd.DataFrame(forest[1].cv_results_)[cols_grid]
results.sort_values(by='rank_test_neg_mean_absolute_error').head(5)
```

	rank_mean_absolute_error	max_depth	min_samples_leaf	n_estimators	mean_mean_absolute_error	mean_absolute_percentage_error	mean_root_mean_squared_error	mean_r2
99	1	25	3	200	-0.06338	-0.01603	-0.09536	0.93404
75	2	20	3	200	-0.06342	-0.01604	-0.09538	0.93400
74	3	20	3	100	-0.06349	-0.01605	-0.09539	0.93399
98	4	25	3	100	-0.06350	-0.01606	-0.09546	0.93390
73	5	20	3	50	-0.06365	-0.01609	-0.09574	0.93350

Table 8: Results from random forest grid search.

Table 8 shows the rankings as revealed by grid search, with the best parameters acknowledge as max_depth=25, min_samples_leaf = 3 and n_estimators=200.

Applying these parameters to the whole dataset to build a better model.

```
# Apply best parameters to whole training data
params = forest[1].best_params_
forest = pipeline(RandomForestRegressor(max_depth=params['max_depth'],
                                          min_samples_leaf=params['min_samples_leaf'
                                          ], n_estimators=params['n_estimators'],
                                          n_jobs=-1))
forest.fit(X_train, y_train)

# Add evaluation metrics to table
y_pred, y_true = add_transformed_row(forest, X_test, y_test, 'Random Forest
                                          GridSearch', best)
```

4.5.2 XGBoost GridSearch

Similar to the Random Forest grid search, follow the same principles to find the best parameters for XGBoost.

```
# Set grid values for search
grid_values = {
    'max_depth': range (4, 20, 2),
    'n_estimators': range(60, 220, 40),
    'learning_rate': [0.1, 0.01, 0.05]
}
scorers = ['neg_mean_absolute_error','neg_mean_absolute_percentage_error',
           'neg_root_mean_squared_error','r2']
```

```
# Build pipeline for grid search of XGB regressor
xgbR = pipeline(GridSearchCV(xgb.XGBRegressor(), param_grid = grid_values, scoring=
                                             scorers, refit='neg_mean_absolute_error',
                                             n_jobs=-1))

# Fit the grid search using sample data
xgbR.fit(SX_train, Sy_train)
```

	rank_mean_absolute_error	max_depth	min_samples_leaf	n_estimators	mean_mean_absolute_error	mean_absolute_percentage_error	mean_root_mean_squared_error	mean_r2
87	1	14	180	0.05000	-0.05804	-0.01469	-0.08790	0.94395
19	2	12	180	0.10000	-0.05819	-0.01476	-0.08876	0.94285
18	3	12	140	0.10000	-0.05826	-0.01477	-0.08873	0.94289
15	4	10	180	0.10000	-0.05832	-0.01478	-0.08866	0.94296
91	5	16	180	0.05000	-0.05834	-0.01477	-0.08842	0.94328

Table 9: Results from random XGB regressor grid search.

From this table, see that XGB regressor out-performs Random Forest through the grid search, with MAE of 0.05804 compared to Forest 0.06338, however again this is only on the sample data and needs to be validated.

5 Model Evaluation and Analysis

There are many evaluation metrics that can be interpreted for regression models. Some of the most common have already been included in this project, to be talked about in this section.

Mean Absolute Error (MAE)
MAE tells us the mean error between the predictions and true values, with lower representing a better model. Often, it is the most important evaluation metric in regression models.

Mean Absolute Percentage Error (MAPE)
Similar to MAE, MAPE identifies the percentage loss per observation, for example, a MAPE of 0.5 would indicate that the predictions are out on average 50%. Of course the goal is to minimise this evaluation metric.

Root Mean Squared Error (RMSE)
When dealing with large outliers, the RMSE is a lot more sensitive than MAE, therefore it can be useful in identifying how the model deals with these. Very high RMSE can show that outliers are not being dealt with apropriately by the model.

R^2 Score
The R^2 score tells us about how much of the variance in the dataset can be explained by its variables. An R^2 score of 1 would indicate that 100% of the variance can be explained, therefore higher the better.

5.1 Coarse-Grained Evaluation

Following the grid search and concluding that XGBoost Regressor seems to be the best algorithm to build the model, find the table evaluation metrics. From this, XGBoost best model has approximate (hasn't yet been validated) MAE of 1889.388, MAPE of 0.123%, RMSE of 7066.598 and R^2 score of 0.864.

Model	MAE	MAPE	RMSE	R2-Score
XGBoost Grid Best	1889.388	0.123	7066.598	0.864
Random Forest GridSearch	1961.937	0.13	7601.961	0.844
Decision Tree GridSearch	2223.784	0.149	7904.892	0.835
LGBM Regressor	2327.956	0.143	8648.101	0.771
Ridge	3784.52	0.27	11369.292	0.629

Table 10: Evaluation metrics of best parameters following grid search.

5.1.1 Validate Results

Using Sk-learn's "cross_validate", find the mean MAE on the test data to confirm that XGBoost is the best model during validation. Repeating for Random Forest, Decision Tree and Light GBM, gives the results in Table.

```
validate = cross_validate(xgbR, X, y, scoring='neg_mean_absolute_error')
validate['test_score'].mean()
```

Model	Validation Mean Test MAE
XGBoost	0.05131743057954076
Random Forest	0.053506972713155354
Light GBM	0.05948763168978832
Decision Tree	0.061131862167344606

Table 11: Results of cross validation with MAE evaluation metric.

This confirms XGBoost as the best model for this problem.

5.2 Feature Importance

Explainability is an important aspect in business when using machine learning models. A lot of the time it can be difficult to explain complex models in advanced machine learning techniques. Most ML models have built in ways to retrieve the importance of the features on predicting the target variable.

5.2.1 Naming Features

Save feature names by fitting the data using transformers in the preprocessor and retrieving the column names.

```
# Fit the training data
onehot_transformer.fit(X_train[onehot_features])
target_transformer.fit(X_train[target_features], y_train)
num_transformer.fit(X_train[num_features])

# Get feature names of numeric transformer
num_feat = list(reg['preprocessor'].transformers[0][1].get_feature_names_out(
                                          num_features))

# Get feature names of one hot transformer
one_hot_feat = list(reg['preprocessor'].transformers[1][1].get_feature_names_out(
                                          onehot_features))
```

```
# Build full feature names to replace columns
encoded_features = num_feat + one_hot_feat + target_features + ['new', 'car_van']
```

Feature names are now saved in a variable "encoded_features".

5.2.2 XGBoost

Visualising the feature importances retrieved from the built model from XGBoost, see, as expected from initial analysis that "Age" and "Model" are the two largest predictors of the price in the model, with 37% and 35% of the model being decided by the two respectively. That's over 70% of the predicted value coming from only 2 features. XGBoost assigns importance to 29 out of 31 features.

Also of note is the feature "New", which has 0 effect on the outcome and serves to slow down computation time of the model. In hindsight, this is probably a consequence of the imputation of "New" vehicles into "Year" earlier in the project. Fuel types and Vehicle Types appear to have quite a large impact on the model also, and would be interesting to investigate the effect that one hot encoding has on the influence within the model.

```
# See how model ranks the features.
feat_imp = pd.DataFrame(xgbR[1].feature_importances_)
feat_imp['name'] = encoded_features
feat_imp = feat_imp.sort_values(by=0, ascending=False)

# Build figure
fig = plt.figure(figsize=(10,8))

# Plot feature importance coefficients
sns.barplot(x=0, y='name', data=feat_imp);
plt.xlabel('Feature Importance')
plt.ylabel('Feature')

plt.show()
```

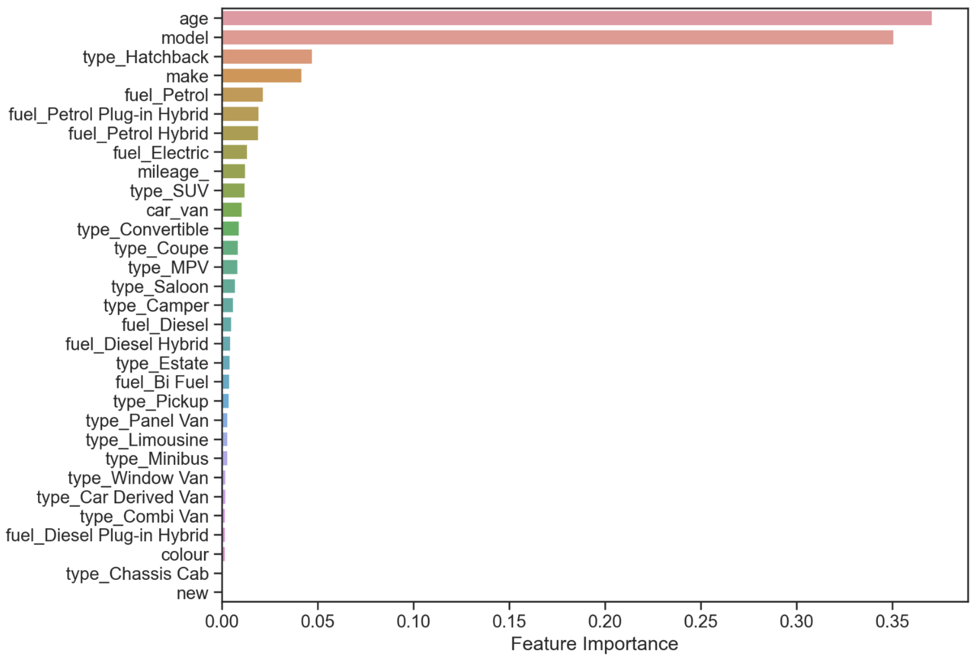

Figure 9: Feature importance of XGBoost regressor.

5.2.3 Random Forest

Similarly in Random Forest, even more importance is assigned to the two features "Age" and "Model", combining to make up 88% of the predicted value of price! In contrast to XGBoost, random forest only assigns importance to 23 of its 31 features, with even more features having almost no impact on the predicted value of price.

```
# See how model ranks the features.
feat_imp = pd.DataFrame(forest[1].feature_importances_)
feat_imp['name'] = encoded_features
feat_imp = feat_imp.sort_values(by=0, ascending=False)

# Build figure
fig = plt.figure(figsize=(10,8))

# Plot feature importance coefficients
sns.barplot(x=0, y='name', data=feat_imp);
plt.xlabel('Feature Importance')
plt.ylabel('Feature')
```

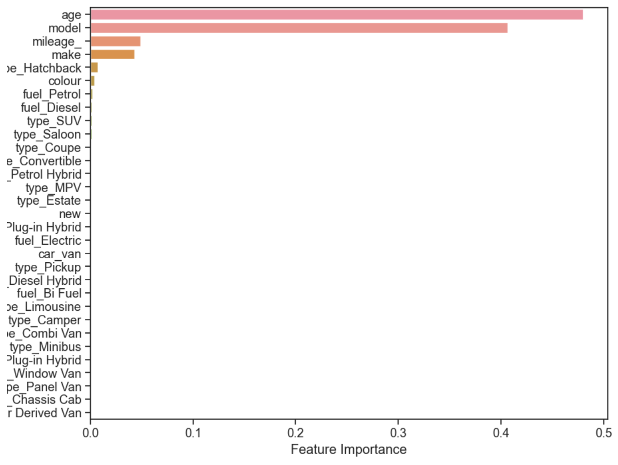

Figure 10: Feature importance of Random Forest Regressor.

5.2.4 Ridge

Both of the previous models are based on Decision Trees, therefore it might be interesting to visualise how a regression model values the features.

Interestingly, Ridge value ”mileage” as the second most important feature after ”model” when making predictions as shown in Figure 11, however as a under-performing model, that could be a cause rather than a solution to better model building.

```
# See how model ranks the features.
feat_imp = pd.DataFrame(ridge[1].coef_)
feat_imp['name'] = encoded_features
feat_imp = feat_imp.sort_values(by=0, ascending=False)
feat_imp['abs'] = np.abs(feat_imp[0])

# Build figure
fig, axs = plt.subplots(2, 1, figsize=(10,18))

# Plot feature importance coefficients
sns.barplot(x=0, y='name', data=feat_imp, ax=axs[0]);
axs[0].set_xlabel('Feature Importance Coefficient')
axs[0].set_ylabel('Feature')

feat_imp = feat_imp.sort_values(by='abs', ascending=False)

sns.barplot(x='abs', y='name', data=feat_imp, ax=axs[1]);
axs[1].set_xlabel('Absolute Feature Importance Coefficient')
axs[1].set_ylabel('Feature')
```

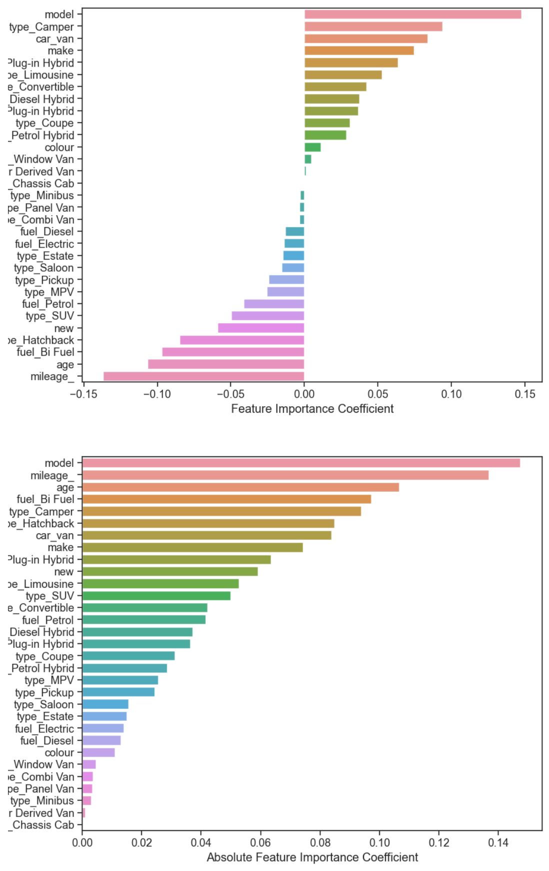

Figure 11: Feature importance of Ridge Regression.

5.2.5 SHAP

SHAP provides a framework for visualising the impact of independent features on the predictions of individual observations.

Figure 12 shows how the features impact the prediction of price, for example the "Model"s with high Feature Value (in red), have high mean model price (from target encoding) in the training data, and will therefore have an increasingly positive effect on the prediction of price. Contesting this is the "Age" of the vehicle, which when high (in red), will pull the prediction price back down. Thinking about the price of a vehicle from a common sense angle, this makes perfect sense.

```
# Assign XGBoost to model variable and create dataframe of 1000 samples
model = xgbR[1]
```

```
X = pd.DataFrame(xgbR[0].transform(X_test[0:1000]).todense())

# Define column names
X.columns = encoded_features

# Build SHAP explainer using model
explainer = shap.TreeExplainer(model)

# Get SHAP values for tree explainer
shap_values = explainer.shap_values(X);

# Visualise SHAP summary plot
shap.initjs()
shap.summary_plot(shap_values, X);
```

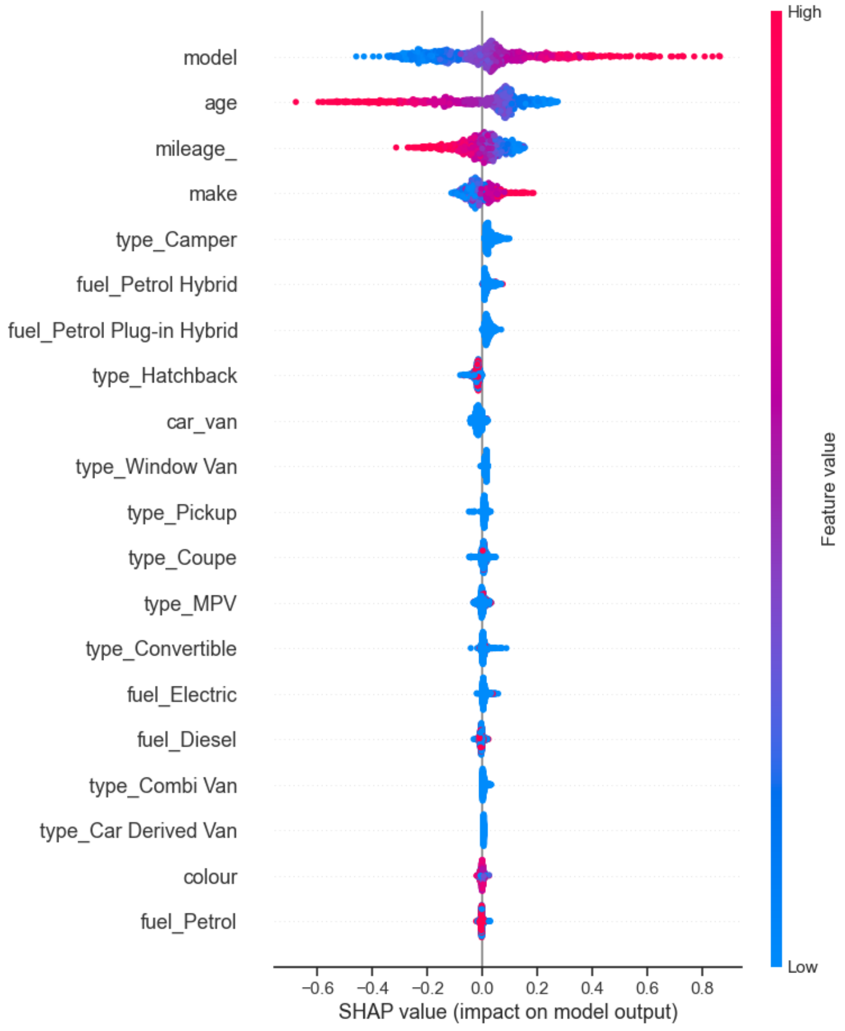

Figure 12: SHAP summary plot of features.

SHAP also provides a force plot feature which will make a prediction on a given point and output the effects each feature has on the prediction. In figure 13, the base value is 4.071 (log transformed) and increases by 1 because the vehicle is hatchback, then decreases because of its age etc. until a final prediction is reached. This can be very helpful with explainability of model predictions, especially in complex models.

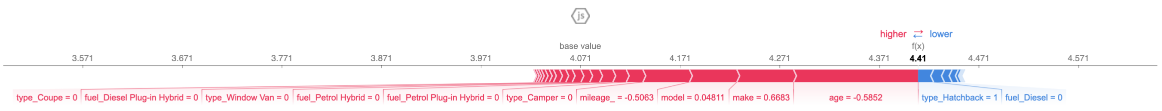

Figure 13: Caption

5.3 Fine-Grained Analysis

Analysing the predicted values can help modify future models to incorporate findings, firstly, analyse the true vs predicted value plot. Figure 14 shows the predictions in red that have an absolute error of more than 50,000.

```
# Transform values back from log base 10
y_pred, y_true = 10**xgbR.predict(X_test), 10**y_test

# Add residuals back to original test dataframe
test_data['price'], test_data['pred'] = y_true, y_pred
test_data['diff'] = test_data['price'] - test_data['pred']
test_data['diff_abs'] = np.abs(test_data['diff'])

# Initiate figure
plt.figure(figsize=(12,8))

# Plot true vs predicted values
ax = sns.scatterplot(x='price', y='pred', data=test_data.loc[test_data['diff_abs'] <
                                            50000], alpha=0.2);
ax = sns.scatterplot(x='price', y='pred', data=test_data.loc[test_data['diff_abs'] >=
                                             50000], marker='o');
ax.set_xlabel('Actual Target Value')
ax.set_ylabel('Predicted Target Value')
ax.set_xlim(0, 400000)
ax.set_ylim(0, 400000)
ax.plot((0, 400000), (0, 400000), ':k', alpha=1, lw=1);
```

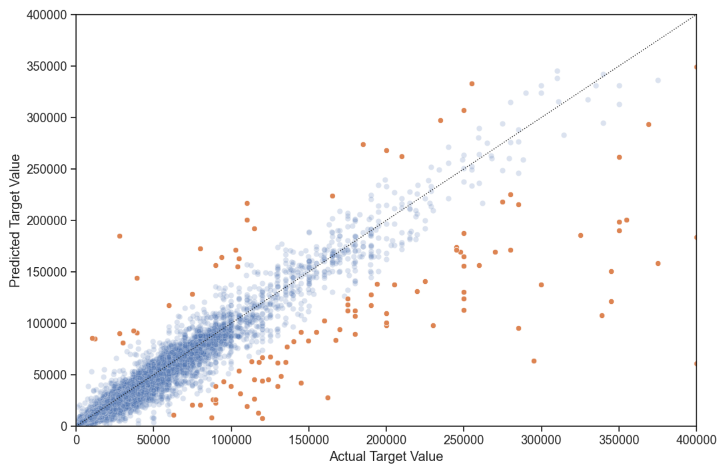

Figure 14: Caption

Considering these large errors separately, find that a lot of the errors are for vehicles that may be considered high end.

```
# Create dataframe of errors > 50,000
big_diff = test_data.loc[test_data['diff_abs'] > 50000]

# Create countplot of makes with large number of errors > 50,000
big_diff['make'].value_counts().sort_values(ascending=False).plot(kind='barh');
```

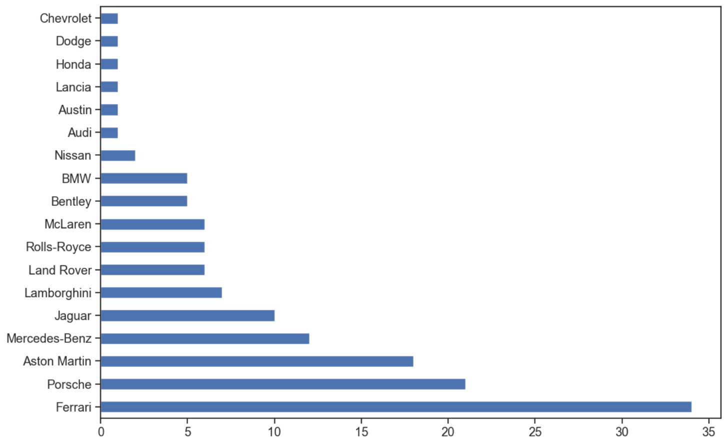

Figure 15: Count-plot of makes with error > 50,000

Isolating make = ”Ferrari” for instance, then viewing the mean and standard deviation of specific models within the training data, provides an insight as to why the models might be struggling to predict accurately. The high prices coinciding with high variance and minimal observations can all negatively impact a given ML models ability to predict accurately.

```
# Create dataframe of ferrari's from training data
ferrari_train = X_train.loc[X_train['make'] == 'Ferrari']
ferrari_train['price_'] = 10**y_train

# View size, mean and standard deviation of most variant ferrari's
ferrari_train.groupby('model')[['age','price_']].agg(['size','mean','std']).
                                                  sort_values(by=('price_','std'), ascending
                                                  =False).head()
```

	age			price_		
	size	mean	std	size	mean	std
model						
365	4	28.25000	26.87471	4	509975.00000	209150.66539
599	30	11.63333	3.59581	30	149916.40000	123508.16470
F12berlinetta	22	7.13636	1.35560	22	193117.95455	106641.54087
458	77	9.07792	1.33541	77	164857.36364	83058.09461
512	5	24.60000	10.96814	5	186967.00000	62095.62611

Table 12: Groupby function of Ferrari from training data.

This can also be visualised by taking the top 5 most commonly occurring Ferrari models, and creating a box-plot of price. Showing the high number of outliers in the data for certain models. To fix this issue, Ferrari (and others) may need to be isolated and predicted using a seperate model, or more observations can be collected to help predict this particular price accurately.

```
# Box-plot of most commonly occuring Ferrari's in training data
sns.boxplot(x='price_', y='model', data=ferrari_train.loc[ferrari_train['model'].isin
                                                  (ferrari_box.index)]);
```

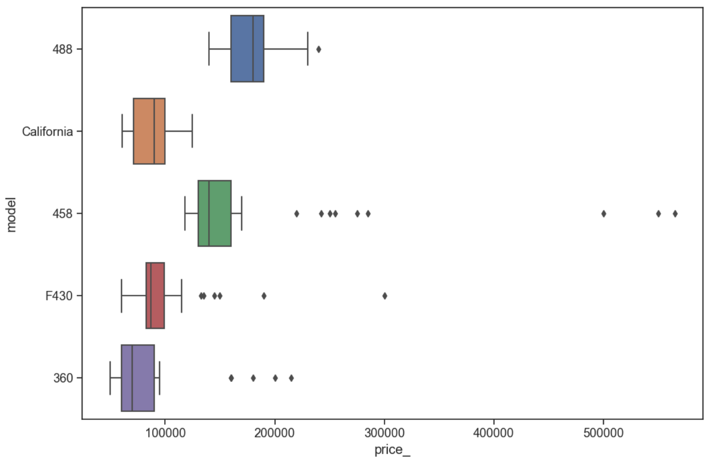

Figure 16: Box-plot of price by most commonly occuring Ferrari's.

5.4 References

Jordan, C. (2023) *CHRISTIANMCB/carListingsMLProject: A project to analyse autotrader car listings dataset., GitHub.* Available at: https://github.com/christianmcb/carListingsMLProject