Machine Learning Concepts

Hal Kolb HAL@KOLB.CO.UK

Contents

1 Domain Understanding and Exploration

1.1 Meaning and Type of Features; Analysis of Distributions

The provided dataset consists of 402005 entries with the following 12 features: public_reference, mileage, reg_code, standard_colour, standard_make, standard_model, vehicle_condition, year_of_registration, price, body_type, crossover_car_and_van and fuel_type.

1.1.1 Price

Price is the feature that the models will aim to predict and is therefore very important to analyse, as with incorrect labels it is impossible for the model to produce useable results. Sampling some prices shows that the data is given as integers ranging from 120 to 9999999, indicating the value is in pounds. An initial look at the distribution shows extremely large positive skew as illustrated in figure 1, making it a good candidate for applying a transformation to later on. Examining the highest price entries in more detail reveals that many of the outliers belong to a few luxury makes and models. This can be seen in more detail in figure 2. To ensure the best results when fitting the model it is important to analyse and handle these outliers.

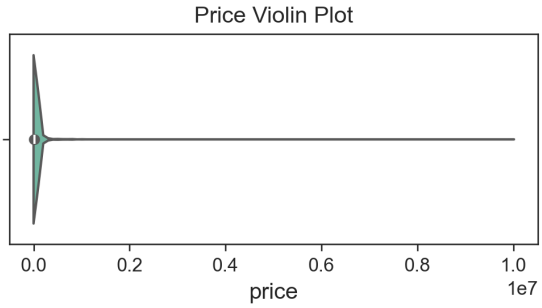


Figure 1: Violin plot of prices. For code see A.1.

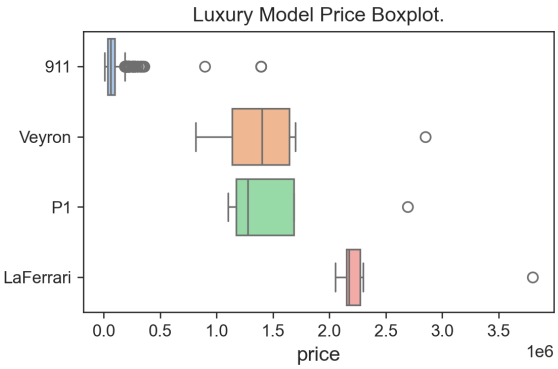


Figure 2: Box plots of prices for a sample of luxury models with extremely large prices. For code see A.2

1.1.2 Mileage

Mileage is a feature that intuitively should have a lot of predictive power as, along with the make and model, it is a major consideration for prospective customers. In the provided dataset, mileage is given as a floating point value which ranges from 0.0 to 999999.0. Additional, there are 127 null values for mileage which will need imputing later. The distribution of mileages also shows significant positive skew with a strong peak in the number of entries with 0 miles. Notably, this peak in entries with 0 miles is also present when just examining cars marked as used. This indicates a potential error with either the condition label or the mileage as it is unusual for so many used cars to have little-to-no miles driven.

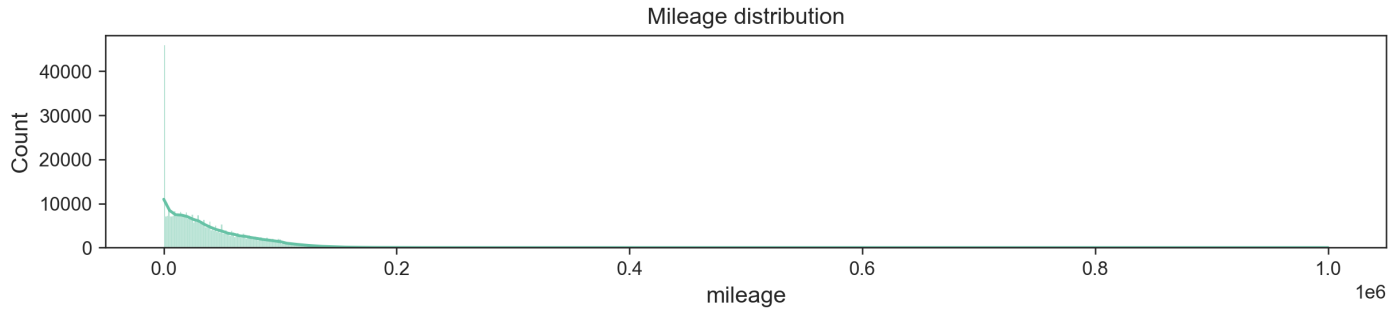


Figure 3: Histogram of mileages including a kernel density estimate showing the large positive skew and significant spike around 0 miles.

1.1.3 Make and Model

There are 1168 unique car models which belong to different 110 makes. Both make and model are strings and there are no null entries. The make largely encodes the same information as the model, as if the model is "911" that can only refer to the Porsche 911. However, some models are shared between multiple makes. For example, both Ferrari and Volkswagen have a "California" model. To ensure duplicate information is not included, and to reduce overlap, the make and model can be combined into a single feature with the line:

```
auto["make_model"] = auto["standard_make"] + " " + auto["standard_model"]
```

This a new feature, "make_model" containing 1217 unique entries, with the most popular being the Volkswagen Golf at 11,583. Each make_model has a median number of identical models of 18 and 217 "make_models" which are unique in the set. As this is still a categorical feature, with string data, it will need encoding before fitting to a model.

1.2 Analysis of Predictive Power of Features

To identify features which are likely to be predictive of the price some individual plots of each feature against price can be used. However, it is important to note that the interactions between features can have predictive value beyond what they each offer individually. As such, these 2D plots alone are not sufficient to fully assess the utility of each feature. A quick feature ranking with *shap* indentifies "model" as the most predictive feature, followed by "mileage" and "body type".

1.2.1 Mileage

Mileage is usually correlated with both the age of the vehicle and the amount of wear-and-tear it has received. As such, it usually one of the key considerations when valuing a used car. This is supported by results in figure 4 and is evaluated further by plotting price against mileage, show in figure 5. This plot demonstrates the approximately linear correlation between mileage and price on a log-linear plot, implying that price decays exponentially as mileage grows. The linear fit shown has an R^2 value of 0.368 indicating there is a a moderate linear correlation between the two features. Given this approximately exponential relationship, a log transformation of price may be effective for improving predictions.

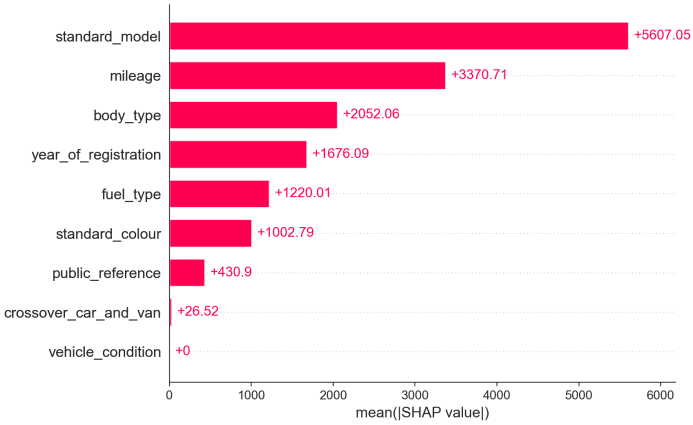


Figure 4: Mean absolute SHAP value for each feature showing which features have the largest impact on a fit model. For code see A.3.

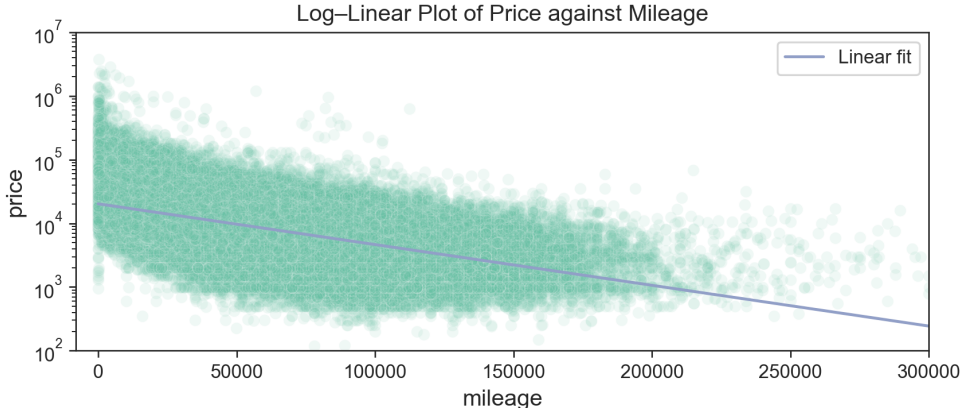


Figure 5: Log-Linear plot of price against mileage, with a linear (in log-space) fit and an R^2 value of 0.368. For code see A.4.

1.2.2 Year

Year of registration also shows a correlation on a log-linear plot but with an inflection point around the year 2002, before which the cars can become vintage and have increased price. With an R_2 value of 0.496 there is a clear linear (in log space) correlation between year and mileage suggesting it is a significantly predictive feature.

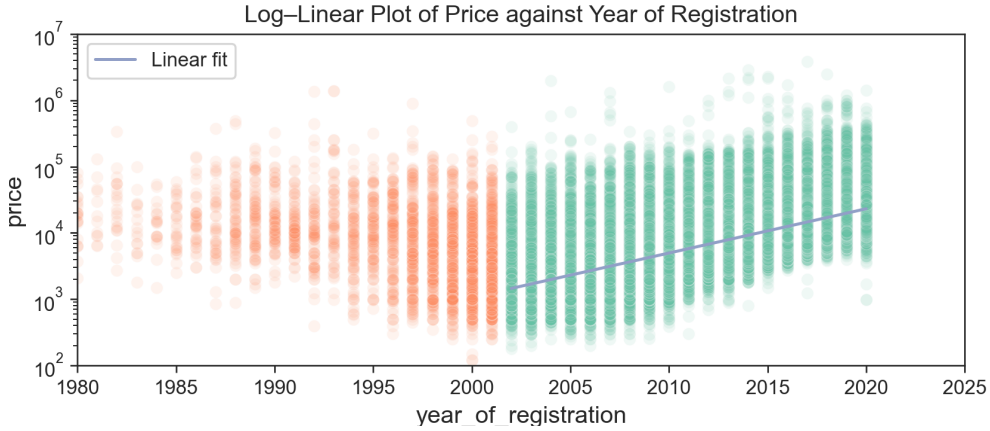
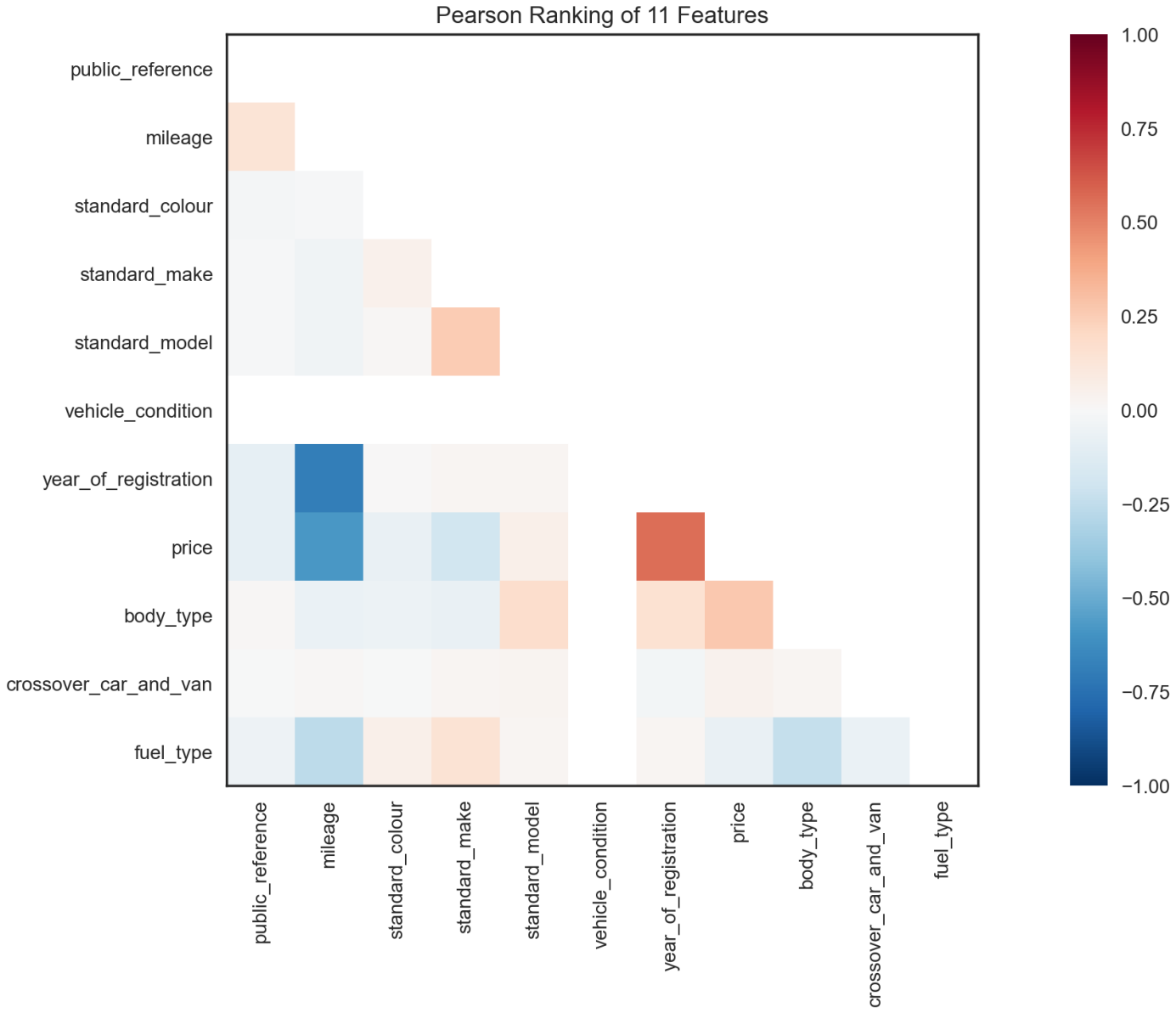


Figure 6: Log-Linear plot of price against year of registration, with a linear (in log-space) fit on data from 2002 onwards which has an R^2 value of 0.496.


Pearson Ranking of 11 Features
public_reference
mileage
standard_colour
standard_make
standard_model
vehicle_condition
year_of_registration
price
body_type
crossover_car_and_van
fuel_type
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

1.3 Data Processing for Data Exploration and Visualisation

1.3.1 Colour

The data set contains 22 unique "standard_colours" which differ greatly in their frequency in the data, this is illustrated by the bar graph shown in figure 7. The thee least common colours: Indigo, Navy and Magenta have only 1, 7 and 15 entries respectively, while there are are 86,287 black cars. With only seven cars labeled as navy it is likely that other navy cars have been included in the blue category, as navy would be expected to be a much more popular category. As such, navy can be replaced with blue to improve accuracy and reduce the number of unique entries to help will reduce dimensionality once one-hot encoded. To examine if any other colours are appropriate for combining I created figure 8 as a swatch of each of the car colours. The exact shade of each colour was decided by using matplotlib's default colours for each colours name, with the exception of bronze and burgundy, as these are not valid matplotlib colours.

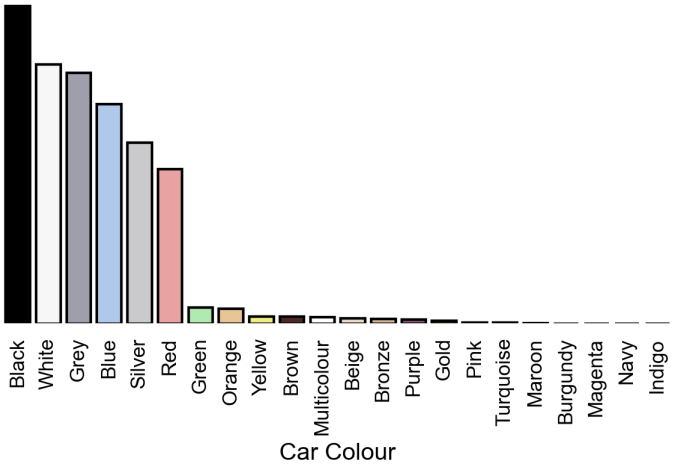


Figure 7: Bar graph showing the number of entires in the dataset for each colour.

Figure 8: Swatch of all unique "standard_colours". For code see A.6

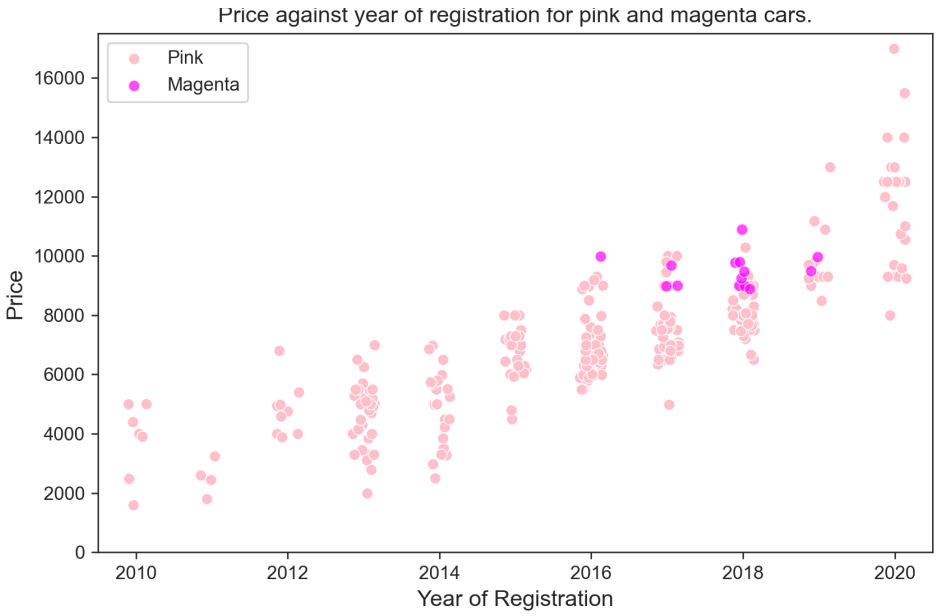


Figure 9: Price against year of registration for pink and magenta cars since 2010, with some x jittering for visibility. For code see A.7

Based on this swatch, as well as adding the 7 navy cars to blue, burgundy and maroon can be merged and the one indigo entry can be changed to purple. Other similarities include bronze and brown and magenta and pink. However, there are 1,330 bronze cars and 2,014 brown which suggests an intentional differentiation. Magenta on the other hand has only 15 entries so it may be reasonable to amend those to pink. Figure ?? compares the prices of pink and magenta cars in more detail which shows that (other than the 2016 Magenta Audi A1 which is more expensive than that year's pink cars) the prices for a given year are similar between the two.

2 Data Processing for Machine Learning

2.1 Dealing with Missing Values, Outliers, and Noise)

2.1.1 Missing Values

column	null_count
year_of_registration	33311
reg_code	31857
standard_colour	5378
body_type	837
fuel_type	601
mileage	127
public_reference	0
standard_make	0
standard_model	0
vehicle_condition	0
price	0
crossover_car_and_van	0

Figure 10: Table of the number of null values for each feature in the data set.

Missing values in the dataset are easy to identify and the table to the left shows the number of null values for each feature. Registration code and year of registration have the most missing data but fortunately there are several fixes for this.

Year of Registration

First, many missing years are for new cars which have not yet been registered. To avoid needing to encode unregistered cars separately and to allow comparison between registered and unregistered cars 2020 (the year the data is from) can be filled in for these missing years. This reduces the number of missing years to only 2062. Of these 2062, 1741 have data in the "reg_code" column which can be used to fill in the year. Over the decades, UK registration codes have encoded the registration year in 3 main ways: with an alphabetic suffix (1963-1983), an alphabetic prefix (1983-2001) and numerically (2001-present). This makes it trivial to infer the year for post-2001 vehicles but unfortunately the dataset does not indicate whether an alphabetic code is a prefix or suffix, introducing ambiguity. To resolve this, the entry is compared to other vehicles of the same model and the year is chosen which is closest to the mean registration year of the similar cars. If there are no other cars of the same model in the data set then the most recent year is chosen as there are far more modern cars in the data than old ones.

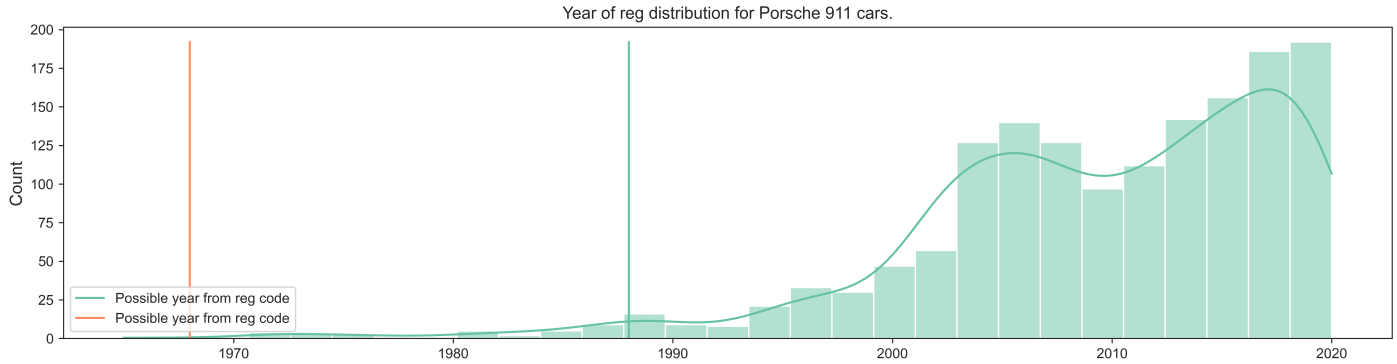


Figure 11: Histogram of possible years of registration for Porsche 911, showing how, for a reg code of **F**, a year of 1988 would be preferred over a year of 1967.

The final method for mapping the registration code to a year is made up of two parts. First the following code maps the code to either one year or a tuple of two possible years (or none if the reg code is not valid):

```
def reg_to_year(reg_code):
    try:
        reg_code = int(reg_code)
        if reg_code > 71 or (50 > reg_code > 20): return np.nan
        return 2000 + reg_code % 50
    except (ValueError, TypeError):
        if not isinstance(reg_code, str): return np.nan
        letters = "ABCDEFGHJKLMNPRSTXY"
        if reg_code == "V": return (1979, 1999)
        if reg_code == "W": return (1980, 2000)
        if reg_code not in letters: return np.nan
        return (1983 + letters.find(reg_code), 1963 + letters.find(reg_code))
```

Then a second function (viewable at A.5) selects the best year for entries with ambiguity. Once all useful information from the reg code has been converted to a year it can be used to fill in the missing years of registration and then dropped. Year of registration also has significant outliers with some years even predating the invention of the car such as: 999, 1006 and 1018. Comparing with the reg code reveals that these are all most likely off by exactly 1000, as the reg codes map to: 1999, 2006 and 2018 respectively. Applying the techniques listed above leaves 323 missing values which will be imputed with a KNN imputer once the data has been processed and scaled.

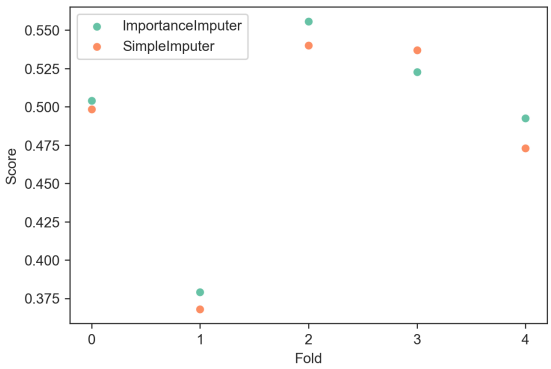


Figure 12: Cross validation scores for two identical KNN fits on a sample of 100,000 entries from the set. The plot compares between a SimpleImputer using the "most_frequent" strategy and the ImportanceImputer discussed here. The mean score for SimpleImputer was 0.483 and the mean score for ImportanceImputer was 0.491. For code see A.9

Colour

With reg code and year handled, standard_colour has the next largest number of missing values. These are more challenging to handle as no other column encodes the information and as they are string data they cannot be imputed with a KNN imputer. Without an obvious fix, I decided to write a categorical imputer I called an "importance imputer". This works by trying to match catagorical data from entries with missing data to entries with data present, according to a ranking of how "important" the category is. For example, if a used electric 2015 BMW i3 Hatchback has a missing colour entry it will look for entries in the dataset which match all of those categorical feature (i.e. other used electric 2015 BMW i3 Hatchbacks) and copy the colour from there. If there is no match found it will iteratively drop features based on how important they are. So if fuel type was deemed least important it would try to find matches for used 2015 BMW i3 Hatchbacks with any fuel type, and so on until a match is made. If multiple matches are found it uses the modal value and if no matches are found it uses the mode from the entire dataset. Critical to this process is ranking the features by importance. This is done by computing Cramér's V between the target feature and all other categorical features. Cramér's V is given as a value between 0 and 1 which quantifies the association between two categorical variables. By sorting the list of features by this value (high to low) it is possible to simply keep popping features from the end of the list until either a match is made, or the list is empty. The full code for this can be viewed in A.8. To measure the accuracy of the imputation, a sample of 5000 entries with known colours had their colours removed, imputed and compared with the true colour. In this test, predicting the mode gave an accuracy of 0.217 whereas the importance imputer had an accuracy of 0.284 (see A.10 for code). Furthermore, figure 12 compares two KNN pipelines which differ only in how categorical data is imputed. This plot shows that the importance imputer was slightly more effective than a simple mode imputation, and was able to improve the performance of the network, if ever so slightly. Between the importance imputer for catagorical data and a KNN imputer for numeric, all missing values can be filled in.

2.1.2 Outliers

As well as dealing with missing values, outliers also need to be identified for numerical data.

Mileage:

Mileage shows several unusual values at both the high and low end. At the high end, some fairly new cars have mileages which indicate they have been driving almost non-stop since the day they were purchased. A good estimate of the maximum mileage a vehicle can do can be made using the taxis. In the data set the taxis travel a mean 16,071 miles per year while some of the extreme entries in the data travel upwards of 250,000 miles per year. A common approach to determining outliers is to use a multiple of the interquartile range (IQR), beyond which values are considered to be outliers. This multiple (the z value) was calculated for the miles per year and a threshold of 3 was set. There were 4170 entries whose miles per year were more than 3 times the IQR from the mean. These values can be deleted and then imputed along with the other missing values. At the low end there are some suspiciously low mileages for older used cars which also can also be removed and replaced. To test the value of removing these outliers, several cross validation fits were performed on the KNN pipeline both with outliers and with them removed. Notably, removing these outliers seemed to only have negative effects on the model's performance, with a more pronounced reduction in score as the z threshold was lowered (from 7 to 3).

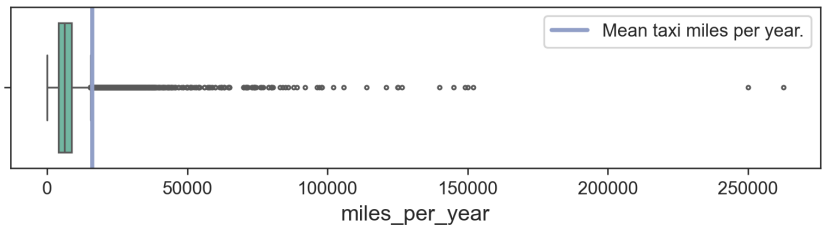


Figure 13: Box plot showing the distribution of miles per year.

2.2 Feature Engineering, Data Transformations, Feature Selection

2.2.1 Transforming

Transforming numerical data to better follow a Gaussian distribution is often useful to improve the accuracy of modelling and prediction. To perform this transformation, I used scikit-learn's PowerTransformer class to apply a Yeo-Johnson transform to the mileage, year and date listed (extracted from the public reference). Figure 14 shows the distribution before and after the transformation (MinMax scaled for comparison).

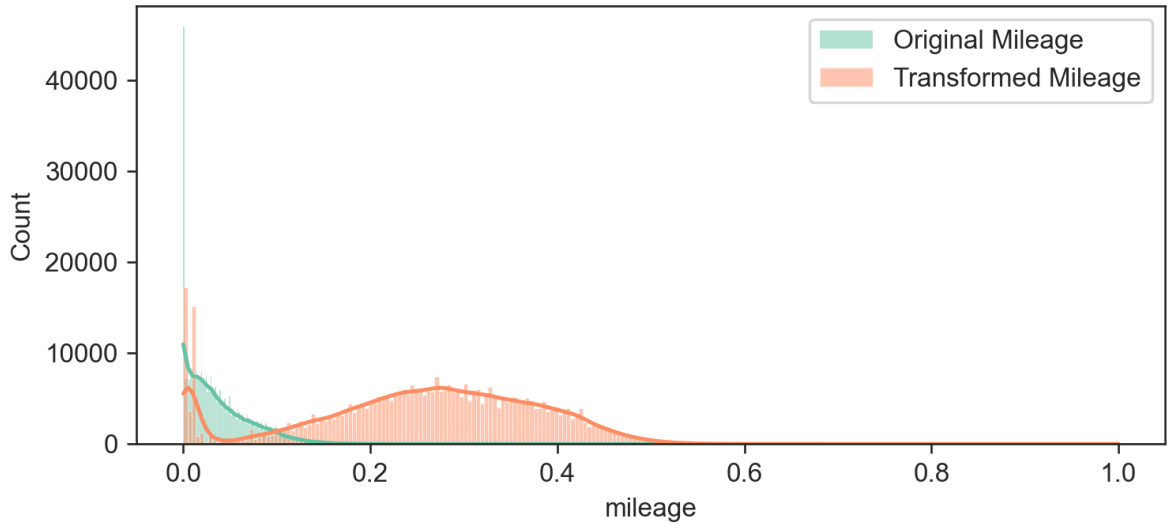


Figure 14: Histogram comparing mileage distribution with and without Yeo-Johnson transformation, both MinMax scaled to better compare the shape of the distribution.

2.2.2 Scaling

Some models are sensitive to the scaling of data. For example, when a KNN regressor finds the nearest neighbours if one feature is much larger than others it will be ranked as further away. To resolve this all features can be scaled to the range 0 to 1 with a MinMax scaler. This scaler is very sensitive to outlier and so the data was transformed before being scaled.

2.2.3 Encoding

Any features with string data need to be encoded to numerical values for fitting. In the dataset (once processed) there are 6 such string type categorical features: standard_colour, make_model, vehicle_condition, body_type, crossover_car_and_van, fuel_type.
Data with only two mutually exclusive values such as condition ("NEW" or "USED") and crossover_car_and_van ("True", "False"), can be encoded with a 1 or 0 to indicate which of the two values is correct. This does not increase the dimensionality of the data nor does it introduce unwanted ordinality as it is equivalent to a one-hot-encoding in this case. Unfortunately, the other features have many more unique values. Once processed there are 19 colours, 16 body types, 9 fuel types and 1217 combined makes and models. These were all encoded with a one-hot encoder to avoid ordinality at the cost of increasing the dimensionality.

```
class AutoEncodeBinary(BaseEstimator, TransformerMixin):
    ''' Find non numeric features with only two values and replace them with 0 and 1'''
    def __init__(self):
        pass

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

    def transform(self, X):
        X = X.copy()

        for col in X:
            if isinstance(col, (str, bool))  and len(X[col].unique()) == 2:
                X[col] = X[col].replace({X[col].unique()[0]:0, X[col].unique()[1]:1})

        return X
```

3 Model Building

3.1 Algorithm Selection, Model Instantiation and Configuration

Each of the preprocessing steps discussed above were added to a preprocessing pipeline which was used for each of the models. Each model's pipeline differs only in the final step based on which regressor is used.

The following code is used to setup the pipelines:

```
cat_features = ["year_of_registration", "fuel_type",
"standard_colour", "standard_make", "standard_model",
"vehicle_condition", "body_type", "crossover_car_van"]

num_features = ["reference", "mileage",
"year_of_registration"]

general_process = Pipeline([
    ("r2y", RegCodeToYear()),
    ("ryd", RegYearDisambiguator()),
    ("fyr", FillYearWithReg()),
    ("aeb", AutoEncodeBinary()),
    ("par", ParseReference()),
    ("col", MergeColours()),
])
cat_pipe = Pipeline([
    ("imp", ImportanceImputer(impute=cat_features)),
    ("cmm", CombineMakeModel()),
    ("drg", DropRegYear()),
    ("ohenc", OneHotEncoder(sparse_output=False,
                handle_unknown='ignore', dtype=int))
])
num_pipe = Pipeline([
    ("imp", KNNImputer()),
    ("scl", RobustScaler()),
    ("tsf", PowerTransformer()),
    ("scl2", MinMaxScaler())
])
col_transformer = ColumnTransformer(transformers=[("num", num_pipe, num_features)
                                                   ("cat", cat_pipe, cat_features)])
preprocessor = Pipeline([("gen", general_process),
                         ("pre", col_transformer)]).set_output(transform="pandas")

knn_pipeline = Pipeline([("preprocess", preprocessor), ("knn", KNeighborsRegressor())])
dtr_pipeline = Pipeline([("preprocess", preprocessor), ("dtr", DecisionTreeRegressor())])
lnr_pipeline = Pipeline([("preprocess", preprocessor), ("lnr", LinearRegression())])
```

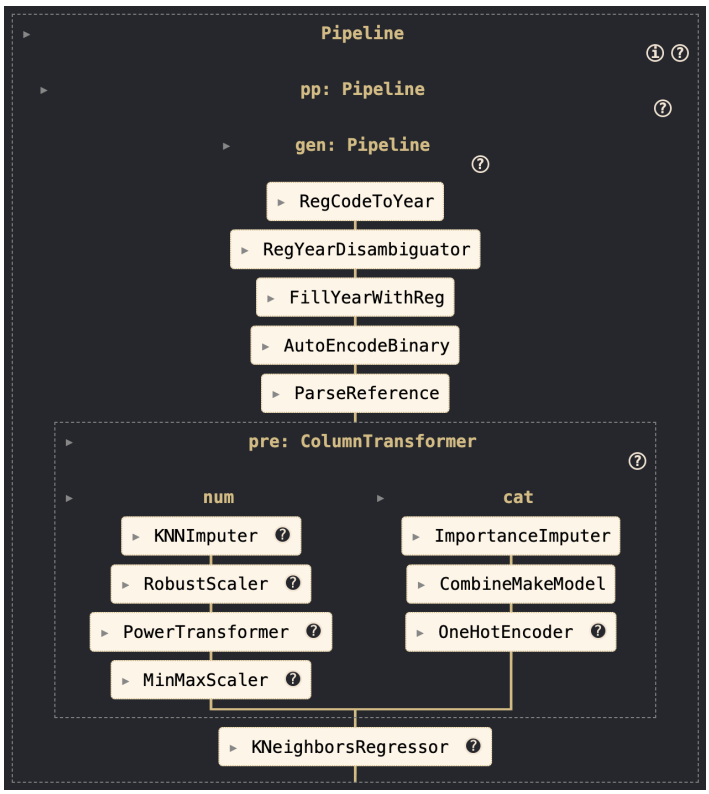


Figure 15: Visualisation of steps in the KNN pipeline.

Each regressor is initially set up with its default parameters and a grid search is performed afterwards to find the optimal choices. As this pipeline is only applied to the input data and not the labels an additional processing step is applied to the price by replacing it with the log of the price. This is a simple transformation that can easily be undone after prediction to return the prices to a value in GBP. Additionally, some records with extremely unrealistic prices are removed from the data set.

```
# Drop rows with unrealistic prices
auto = auto.loc[auto["price"] != 9999999]
# Log scale price
auto["price"] = np.log(auto["price"])
```

3.2 Grid Search, Model Ranking and Selection

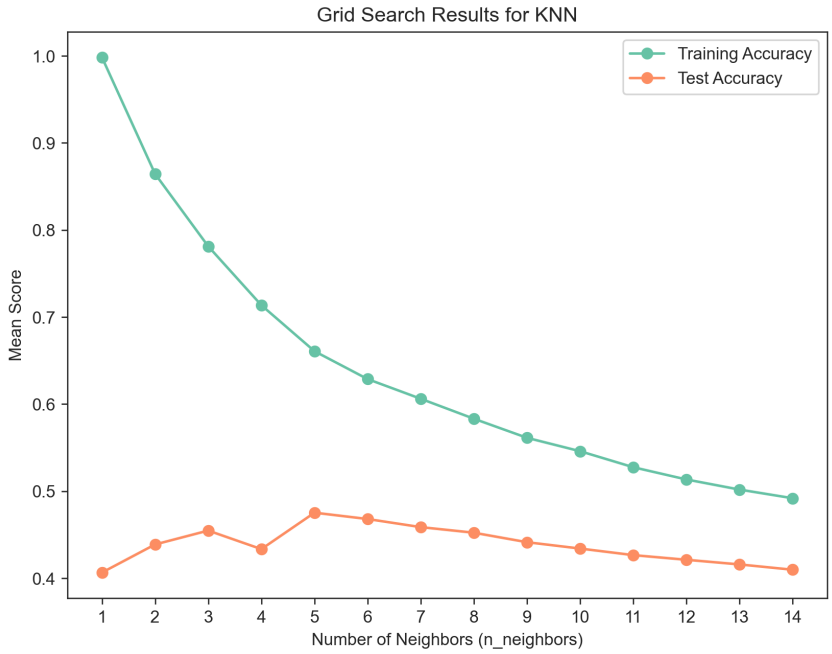


Figure 16: Scatter plot of training and test scores against the number of neighbours used in KNN model, with a maximum test score of 0.475 at 5 neighbours. Performance was evaluated on a sample of 100,000 records with 3 fold cross validation.

3.2.1 KNN

The only hyperparameter available for tuning a KNN regressor is the number of neighbours to use to determine the predicted output. Figure 16 shows how changes in this parameter affect the model's scores on the training and test data. Low values overfit the data, scoring well on the training set but failing to generalise enough to score well in testing whereas higher values are under-fitting and performing poorly in both training and testing. The grid search identified 5 as the optimal number of neighbours, scoring 0.475 on the test data.

3.2.2 DTR

The decision tree regressor has several parameters that can be adjusted to tune the model. By restricting attributes such as the maximum depth or maximum number of features to use the model can be simplified and made easily interpreted and human-usable. However, this comes at the cost of limiting the model's ability to fit the data and will result in under-fitting. It is also possible to overfit the data if there are few restrictions which will affect the model's accuracy on the training data. To analyse the effect of maximum depth and minimum samples per split a grid search was performed to produce the data shown in figure 17.

Figure 17: Comparison of decision tree training and test scores between different maximum depth and minimum samples per split. Scoring a maximum of 0.624 on the test data with no max depth and a minimum of 16 samples per split. Performance was evaluated on a sample of 100,000 records with 5 fold cross validation.

3.2.3 Linear Regressor

The linear regressor does not have any parameters for tuning and scored 0.636 on the testing data when 5 fold cross validated.

```
scores = cross_val_score(lnr_pipeline, X, y, n_jobs=12)
print(f"Mean Score: {round(scores.mean(), 3)}\nScore STD: {round(scores.std(),3)}\nAll Scores: {scores}")
```

4 Model Evaluation and Analysis

4.1 Coarse-Grained Evaluation/Analysis

To improve the model's predictions the prices were first scaled using a log function. The following scores measure each model's ability to predict the log of the price from a given set of inputs.

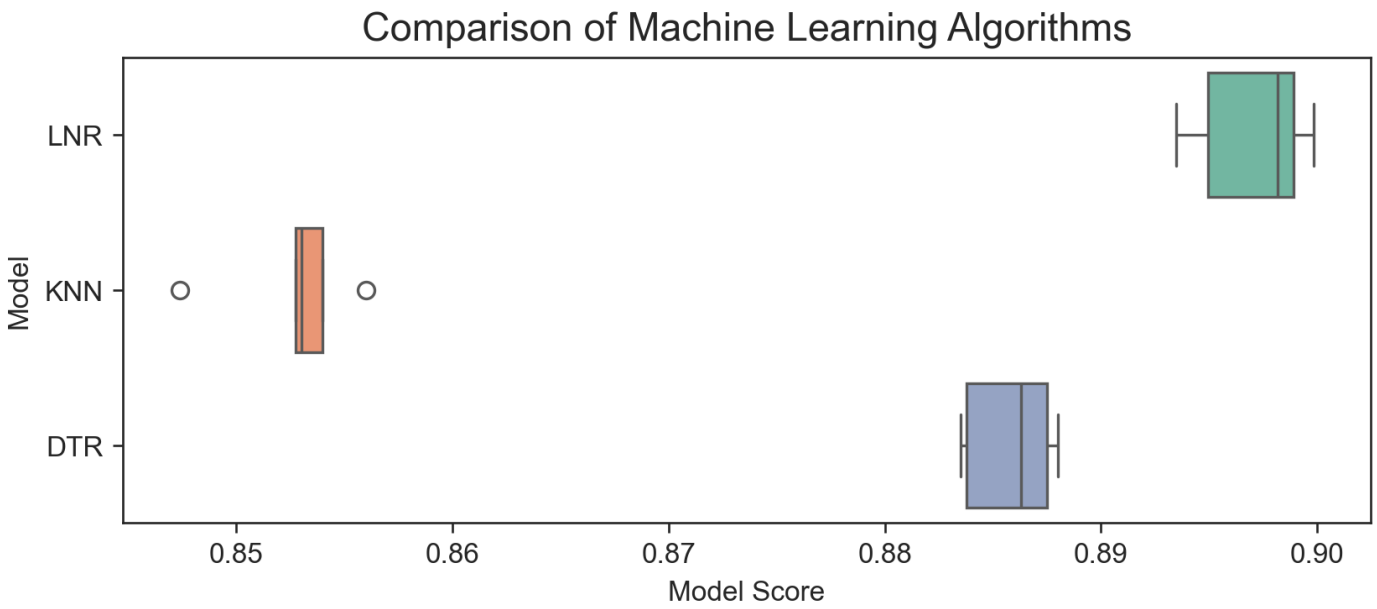


Figure 18: Box plot of scores from 5 fold cross validation for each model.

Scores

The linear regressor scored a mean of 0.897 with a standard deviation of 0.002, the K-nearest neighbour regressor scored a mean of 0.853 with a standard deviation of 0.003 and the decision tree regressor scored a mean of 0.886 with a standard deviation of 0.002. The optimal parameters discussed above were used for each model and the code for the evaluation is shown below.

```
dtr_pipeline.set_params(dtr__max_depth=None, dtr__min_samples_split=22)
knn_pipeline.set_params(knn__n_neighbors=5)

models=[];
models.append(('LNR', lnr_pipeline))
models.append(("KNN", knn_pipeline))
models.append(('DTR', dtr_pipeline))

names=[];
result=[];
for name,model in models:
    k_fold=KFold(n_splits=5, shuffle=True, random_state=seed)
    score=cross_val_score(model, X, y, n_jobs=12, cv=k_fold);
    result.append(score)
    names.append(name)
    print(name, score.mean().round(3), score.std().round(3))
```

The initial results indicate that the linear model produces the most accurate results and a more detailed discussion of the performance is discussed below.

4.2 Feature Importance

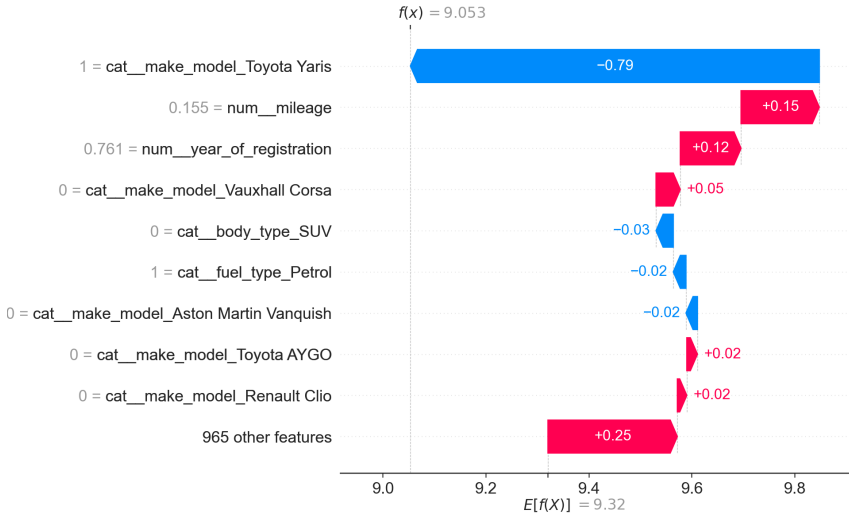


Figure 19: Waterfall plot of how each feature influences the result of the linear model's prediction for a Used Grey 2017 Toyota Yaris with a mileage of 16,092. For code see A.11.

4.2.1 Modelling Predictions

The waterfall plot in figure 19 shows how each feature influences the predicted price of this Toyota Yaris. This matches well with how a human might value a car, first evaluating the model, then the mileage, then the age with other features having a much less pronounced impact on the valuation. Using plots such as this the more accurate linear model can be used while nearly matching the interpretability of a decision tree.

4.2.2 Residuals

The plot in figure 20 shows the distribution of the residuals for each model. While evaluating all three on one plot is ideal in readability, it shows clearly that the linear fit is slightly more likely to overestimate the price relative to the others. However, the linear model has much more of its "mass" in the -0.3 to 0.3 range, meaning when its predictions are off, they are usually off by less than the other models. This is one reason why it scored highly on the cross validation.

The decision tree model has a sharper central peak, showing that it produces extremely accurate results more often than the other models. However, it scored slightly worse than linear overall, due to a larger number of very poor predictions.
The tree above used an unlimited depth and the final model had a depth of 151. While this improves the accuracy, it somewhat negates the interpretability of a decisions tree. I therefore fit a second decision tree regressor with a depth limit of 4 which scored 0.641 on the testing data and the tree is (just about) small enough to be shown below.

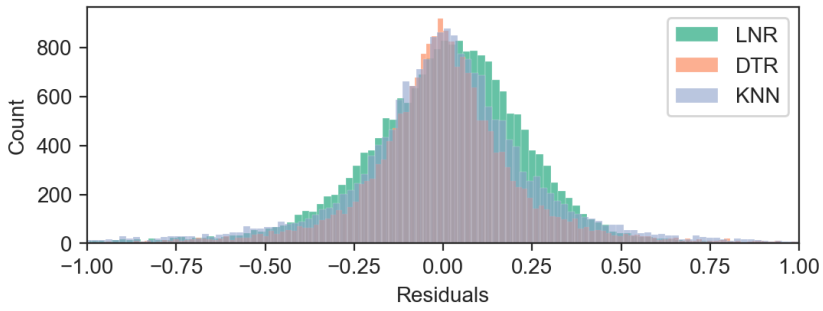


Figure 20: Distribution of the residuals for each model based on a single fit on all training data. The testing scores were 0.899, 0.880 and 0.844 for LNR, DRT and KNN respectively.

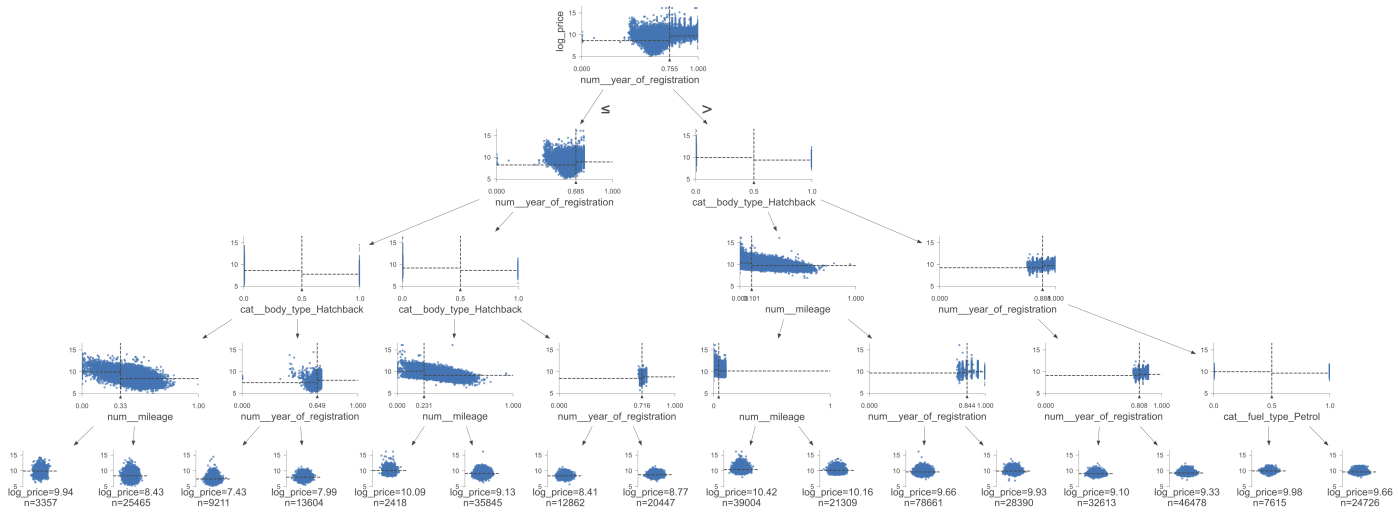


Figure 21: Tree produced with a depth limit of 4 using a sample of 50,000 records, scoring 0.641 on the test data.

Similar to the shap plot above this model splits primarily using year of registration and mileage.

4.3 Fine-Grained Evaluation

For a more detailed comparison of the predictions the plot in figure 22 shows 10 specific cars and the prices each model estimated for them.

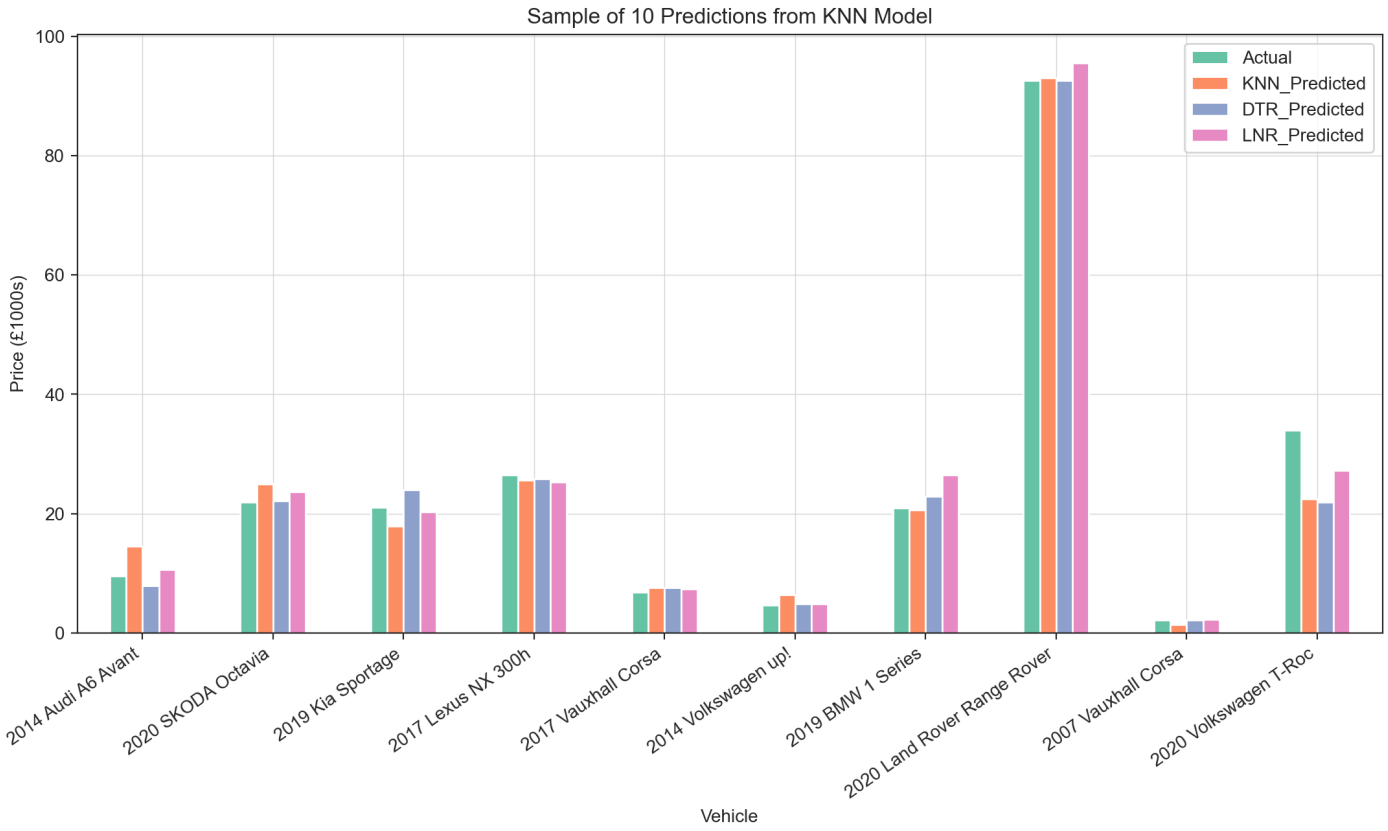


Figure 22: Sample of 10 price predictions, in 1000s of pounds, from each model, fit with optimal parameters.

The specific values predicted and shown in figure 22 are shown in the table below.

Name	Actual Price	KNN Predicted Price	DTR Predicted Price	LNR Predicted Price
2014 Audi A6 Avant	£9 500	£14 567	£7 908	£10 608
2020 SKODA Octavia	£21 900	£24 949	£22 159	£23 686
2019 Kia Sportage	£22 000	£17 860	£23 942	£20 291
2017 Lexus NX 300h	£26 450	£25 615	£25 838	£25 279
2017 Vauxhall Corsa	£6 795	£7 613	£7 545	£7 309
2014 Volkswagen up	£4 641	£6 410	£4 896	£4 805
2019 BMW 1 Series	£20 900	£20 654	£22 920	£26 496
2020 Land Rover Range Rover	£92 660	£93 010	£92 625	£95 580
2007 Vauxhall Corsa	£2 175	£1 368	£2 127	£2 208
2020 Volkswagen TRoc	£33 990	£22 396	£21 951	£27 203

Figure 23: Prices and predictions for the 10 cars in the sample above.

Based on all the data analysed it appears that the models, particularly the linear regressor, are able to give estimates that are fairly useable. A selection of the least accurate predictions are shown in the table below.

name	Off By	Percentage Error
2001 Audi A3	£5 912	1.689
2002 BMW X5	£7 209	1.4418
2003 Fiat Multipla	£5 017	1.0054
2002 Audi A6 Avant	£4 202	0.8489

Figure 24: The 4 least accurate predictions from the linear model.

Appendix A. Code

A.1 Prices Violin Plot - Return to text

```
# Visualise spread of price
plt.title("Price Violin Plot");
sns.violinplot(auto, x="price");
```

A.2 Luxury Car Model Price Box-Plot - Return to text

```
combined_prices = auto.loc[auto["standard_model"].isin(["Veyron", "LaFerrari", "P1", "911"]),
                                                                    ["price", "standard_model"]]
plt.title("Luxury Model Price Boxplot.");
sns.boxplot(data=combined_prices, y="standard_model", x="price", palette="pastel");
```

A.3 Bar Plot of Mean Absolute SHAP Value of Each Feature - Return to text

```
cat_features = ['standard_colour', 'standard_model', 'vehicle_condition',
                'body_type', 'crossover_car_and_van', 'fuel_type']

model = CatBoostRegressor()

model.fit(X, y, cat_features=cat_features)

explainer = shap.Explainer(model)
shap_values = explainer(X)
shap.plots.bar(shap_values)
```

A.4 Log-Linear Plot of Price vs Mileage, With Fit - Return to text

```
import statsmodels.api as sm

auto["log_price"] = np.log(auto["price"])

X = sm.add_constant(auto['mileage'])
model = sm.OLS(auto['log_price'], X).fit()

intercept, slope = model.params['const'], model.params['mileage']

print("R-squared (log-space):", model.rsquared)
print(f"mileage = {intercept:.3f} + {slope:.3f} * log(price)")

x_plot = np.linspace(auto['mileage'].min(), auto['mileage'].max(), 200)
y_hat = np.exp(intercept) * x_plot**(slope)
X_pred = sm.add_constant(x_plot)

pred = model.get_prediction(X_pred)
log_y_hat = pred.summary_frame(alpha=0.01)['mean']

# Transform back to original space
y_hat = np.exp(log_y_hat)

plt.figure(figsize=(8, 3), dpi=120)

sns.scatterplot(data=auto, y='price', x='mileage', alpha=0.1)
plt.plot(x_plot, y_hat, color="#919FC7", label='Linear fit')
plt.title("Log{Linear Plot of Price against Mileage")
plt.xlabel('mileage')
plt.ylabel('price')
plt.yscale("log")
plt.ylim(100, 1e7)
plt.xlim(-8e3, 0.3e6)
```

```
plt.legend()
plt.show()
```

A.5 Selecting Optimal Year for Letter Reg Codes - Return to text

```
class RegYearDisambiguator(BaseEstimator, TransformerMixin):
    '''
    Some letter reg codes can match to two different years e.g. "A" maps to both 1963 and 1983.
    This transformer finds the mean year of other cars of the same model and picks whichever of
    the two is closer to that mean. If there are no matching models of car the most recent year
    is taken as newer cars are more common in the dataset.
    '''
    def __init__(self):
        pass

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

    def transform(self, X):
        X = X.copy()

        if not("reg_code" in X or "year_of_registration" in X or "reg_code_year" in X):
            print("Skipping reg year disambiguation as year is not present")
            return X

        def is_alpha(val):
            try: return not val.isnumeric()
            except: return False

        missing = X[(X["reg_code"].apply(is_alpha)) & (X["year_of_registration"].isna())]

        # For each ambiguous year select the year closest to the mean for that make.
        missing = missing.loc[missing["reg_code_year"].notna()]
        full_missing = X.loc[missing.index]

        for i in range(len(full_missing)):
            missing_model = full_missing.iloc[i]["standard_model"]
            years = full_missing.iloc[i]["reg_code_year"]
            used = X.loc[(X["standard_model"] == missing_model) & (X["year_of_registration"].notna())]

            mean_year = np.array(used["year_of_registration"]).mean()
            closest_year = years[0] if abs(mean_year - years[0]) < abs(mean_year - years[1]) else years[1]

            X.at[missing.index[i], "reg_code_year"] = closest_year

        return X
```

A.6 Colour Swatch Plot - Return to text

```
import matplotlib.patches as patches

# Define a 5x5 grid of colours (hex codes or named colours)
colors = [
    ["black", "grey", "white", "blue", "silver", "red", "green"],
    ["orange", "yellow", "brown", "beige", "#cd7f32", "purple", "gold"],
    ["pink", "turquoise", "maroon", "#800020", "magenta", "navy", "indigo"]
]

# Create figure and axes
fig, ax = plt.subplots(figsize=(7, 3))
```

```
# Plot the squares
for i, row in enumerate(colors):
    for j, color in enumerate(row):
        rect = patches.Rectangle(
            (j, i), 1, 1, linewidth=0, edgecolor=None, facecolor=color
        )
        ax.add_patch(rect)
        # Add the colour name in the centre of the square
        if color != "#FFFFFF":
            label = color
            label_map = {"#800020": "burgundy", "#cd7f32": "bronze"}
            if color in label_map.keys():
                label = label_map[color]
            ax.text(
                j + 0.5,       # x-coordinate (centre of square)
                i + 0.5,       # y-coordinate (centre of square)
                label,         # Text to display
                ha="center",   # Horizontal alignment
                va="center",   # Vertical alignment
                fontsize=12,    # Font size
                color="black" if color not in ["black", "navy", "indigo", "blue"] else "white",
            )

# Set axis limits and turn off axes
ax.set_xlim(0, 7)
ax.set_ylim(3, 0)
ax.axis("off")
```

A.7 Price vs Year of Registration for Pink and Magenta since 2010 - Return to text

```
# Filter the data for Pink and Magenta
pink_data = auto.loc[auto["standard_colour"] == "Pink"]
magenta_data = auto.loc[auto["standard_colour"] == "Magenta"]

jitter_amount = 0.15
# Add jitter to x-values
pink_data["year_of_registration_jittered"] = pink_data["year_of_registration"] +
                                np.random.uniform(-jitter_amount, jitter_amount, size=len(pink_data))
magenta_data["year_of_registration_jittered"] = magenta_data["year_of_registration"] +
                                np.random.uniform(-jitter_amount, jitter_amount, size=len(magenta_data))

plt.figure(figsize=(8, 5), dpi=120)

# Create scatter plots with jittered x-values
sns.scatterplot(data=pink_data, x="year_of_registration_jittered", y="price",
                                                color="pink", label="Pink", alpha=1)
sns.scatterplot(data=magenta_data, x="year_of_registration_jittered", y="price",
                                                color="magenta", label="Magenta", alpha=0.7)

# Set limits and labels
plt.xlim(2009.5, 2020.5)
plt.ylim(0, 1.75e4)
plt.xlabel("Year of Registration")
plt.ylabel("Price")
plt.title("Price against year of registration for pink and magenta cars.")
plt.legend()
```

A.8 Importance Imputer - Return to text

```
class ImportanceImputer(BaseEstimator, TransformerMixin):
    '''
```

```
    Imputes categorical data by taking the mode of data which match other
    categorical features. The other categorical features are sorted based
    on their correlation with the target feature. To find the correlation
    Cramer's V is used. The imputer first attempts to locate entries that
    match every feature e.g. finding cars with the same make, model, year
    and condition. If no match is found, then the least important feature
    is ignored and the process continues until a minimum matching entries
    are found. Once found, the modes of the matches are filled in the gap.

    '''
    def __init__(self, impute_features, similar_threshold=1):
        self.impute_features = impute_features
        self.similar_threshold = similar_threshold
        self.ordered_features = {}

    def fit(self, X, y=None):
        X = X.copy()

        # Identify numeric and categorical features
        numerical_features = list(set(X.select_dtypes(include=[np.number]).columns))
        self.categorical_features_ = list(set(X.columns) - set(numerical_features))

        for feature in self.impute_features:
            # Compute Cramér"s V for categorical features
            self.cramer_dict_ = {}
            for feat in self.categorical_features_:
                if feat == feature: next

                contingency_table = pd.crosstab(X[feat], X[feature])
                chi2, _, _, _ = chi2_contingency(contingency_table)
                n = contingency_table.sum().sum()
                r, k = contingency_table.shape
                cramer_v = np.sqrt((chi2 / n) / (min(r, k) - 1))
                self.cramer_dict_[feat] = cramer_v

            self.cramer_dict_ = dict(sorted(self.cramer_dict_.items(), key=lambda x: -x[1]))

            ordered_features = list(self.cramer_dict_.keys())
            if feature in ordered_features: ordered_features.remove(feature)
            self.ordered_features[feature] = ordered_features

        return self

    def transform(self, X):
        X = X.copy()

        for feature in self.impute_features:
            # Impute missing values
            missing_indices = X[X[feature].isna()].index

            def impute_value(row):
                features_to_match = self.ordered_features[feature].copy()
                while features_to_match:
                    condition = True
                    for feat in features_to_match:
                        condition &= X[feat] == row[feat]
                    similar_entries = X.loc[condition & X[feature].notna()]
                    if len(similar_entries) >= self.similar_threshold:
                        return similar_entries[feature].mode()[0]
                    features_to_match.pop()
```

```
                return X[feature].mode()[0]

            # Apply imputation
            X.loc[missing_indices, feature] = X.loc[missing_indices].apply(impute_value, axis=1)

        return X
```

A.9 KNN Fit Imputer Comparison - Return to text

```
print(cross_val_score(knn_pipeline_simple, X, y, cv=5, n_jobs=12, error_score='raise'))
print(cross_val_score(knn_pipeline_importance, X, y, cv=5, n_jobs=12, error_score='raise'))

ax = sns.scatterplot(np.array([0.5343527 , 0.52080418, 0.42629521, 0.47077498, 0.6117846]), label="Importa
sns.scatterplot(np.array([0.53490801, 0.5054836 , 0.427131  , 0.48785469, 0.56022955]), label="SimpleImput

ax.xaxis.set_major_locator(MaxNLocator(integer=True))
```

A.10 Imputer Comparison Test – Return to text

```
X, y = auto.drop(columns="price"), auto[["price"]]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=seed)

importance_imp = ImportanceImputer(impute_features=["standard_colour"])
simp_imp = SimpleImputer(strategy="most_frequent")

sample_size = 5000
auto = auto.dropna()

test_set = auto.sample(sample_size, random_state=42)
answers = test_set.copy()

# Simple Imputer test
auto.loc[test_set.index, "standard_colour"] = np.NaN

auto[["standard_colour"]] = simp_imp.fit_transform(auto[["standard_colour"]])
results = (auto.loc[test_set.index, "standard_colour"] == answers.loc[test_set.index, "standard_colour"]).
print("SimpleImputer success rate:", results[1]/sample_size)

# Importance Imputer test
auto.loc[test_set.index, "standard_colour"] = np.NaN

auto["standard_colour"] = importance_imp.fit_transform(auto.drop(columns=["price"]))["standard_colour"]
results = (auto.loc[test_set.index, "standard_colour"] == answers.loc[test_set.index, "standard_colour"]).
print("Importance imputer success rate:", results[1]/sample_size)
```

A.11 Waterfall plot of shap values for a Toyota Yaris - Return to text

```
lr = LinearRegression()
lr.fit(X_train, y_train)

# Fit the explainer
explainer = shap.Explainer(lr.predict, X_test)
# Calculate the SHAP values
shap_values = explainer(X_test)

shap.plots.waterfall(shap_values[5])
# 2017 Used Grey Toyota Yaris, Petrol, 16092.0 miles
```

References