Machine Learning Concepts Project

Contents

Data/Domain Understanding and Exploration

Meaning and Type of Features; Analysis of Distributions

Creating and Previewing a Sample Dataset

The output table presents a preview of the sampled data with various columns like public_reference, mileage, reg_code, standard_colour, and others. Each row corresponds to a vehicle listing, showcasing details such as make, model, year of registration, price, and fuel type.

```
df_sample = df.sample(frac=0.2, random_state=42)
df_sample.head()
```
✓ 0.0s

	public_reference	mileage	reg_code	standard_colour	standard_make	standard_model	vehicle_condition	year_of_registration	price	body_type	crossover_car_and_van	fuel_type
332044	202010074692259	2826.0	69	Silver	Volkswagen	Sharan	USED	2019.0	23000	MPV	False	Diesel
173955	202009023198786	10601.0	19	Red	Nissan	Qashqai	USED	2019.0	16000	SUV	False	Petrol

Reviewing Data Types in the DataFrame

Columns such as 'public_reference', 'mileage', 'year_of_registration', and 'price', which involve quantitative values have types like 'int64' and 'float64'. Categorical data such as 'reg_code', 'standard_colour', 'standard_make', 'standard_model', 'vehicle_condition', 'body_type', and 'fuel_type' have the 'object' type, typically for strings. The 'crossover_car_and_van' column has 'bool' type, indicating boolean values.

```
df.dtypes
```
✓ 0.0s

```
public_reference          int64
mileage                 float64
reg_code                 object
standard_colour          object
standard_make            object
```

Statistical Summary of Vehicle Mileage

Count: 401,878 entries, without counting NANs. This shows the number of vehicles with recorded mileage.

Mean: The mean mileage is approximately 37,743 miles.

Standard Deviation: 34,831 miles. This indicates a wide range of vehicle mileages.

Minimum Mileage: The lowest recorded mileage is 0 miles, indicating new vehicles.

```
df['mileage'].describe()
```
✓ 0.2s

```
count    401878.000000
mean      37743.595656
std       34831.724018
min           0.000000
25%       10481.000000
50%       28629.500000
```

25% Percentile: 10,481 miles. Indicates the mileage below which 25% of vehicles' mileages are.

Median (50% Percentile): 28,629 miles. Half of the vehicles have less mileage, and half have more.

75% Percentile: 56,875 miles. 75% of the vehicles have mileage below this value.

Maximum Mileage: The maximum recorded mileage is 999,999 miles, which may suggest outliers.

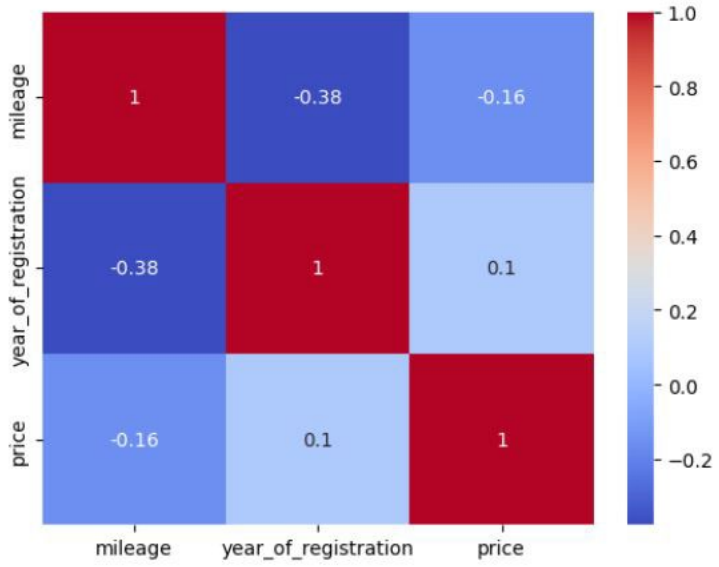
Analysis of Predictive Power of Features

Correlation Analysis Among Mileage, Year of Registration, and Price

The heatmap of the correlation matrix between 'mileage', 'year_of_registration', and 'price' provides insights into the possible linear correlation between features. (-0.16) and (0.1): The weak negative and positive correlation between 'mileage', 'year_of_registration', and 'price' suggests that 'mileage' and 'year_of_registration' has a limited linear predictive power for 'price'. However, there might be a non-linear relationship that Pearson correlation cannot detect.

```
df_numeric = df[["mileage", "year_of_registration", "price"]]
correlation_matrix = df_numeric.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.show()
```

✓ 0.4s



The grouped statistical data on vehicle prices by fuel type can provide insights into the predictive power of the ‘fuel_type’ feature for vehicle prices: The data shows significant differences in mean, median, first quartile (Q1), and third quartile (Q3) prices among various fuel types. This indicates that 'fuel_type' is a distinguishing factor in predicting vehicle prices.

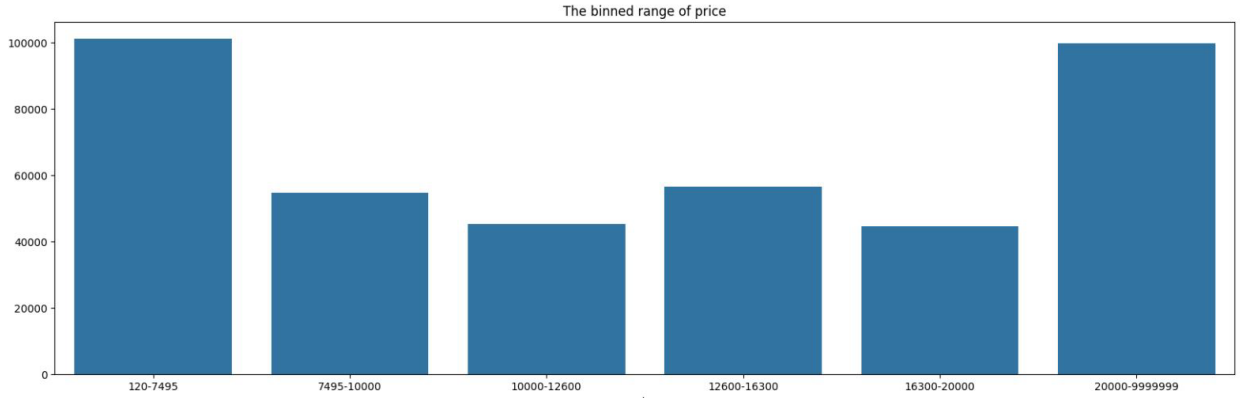
```
grouped = df.groupby('fuel_type')['price'].agg(
    mean='mean',
    median='median',
```

fuel_type	mean	median	Q1	Q3
Bi Fuel	14630.524887	14000.0	11945.00	15421.0
Diesel	16505.048387	13495.0	8250.00	20495.0

Data Processing for Data Exploration and Visualisation

To have more nuanced data visualization of vehicle prices, the dataset's 'price' column has been segmented into distinct bins. This graphical representation shows that a significant number of vehicles are priced between $120-$7495 and $20000-$999999. This shows a notable presence of high and low priced vehicles.

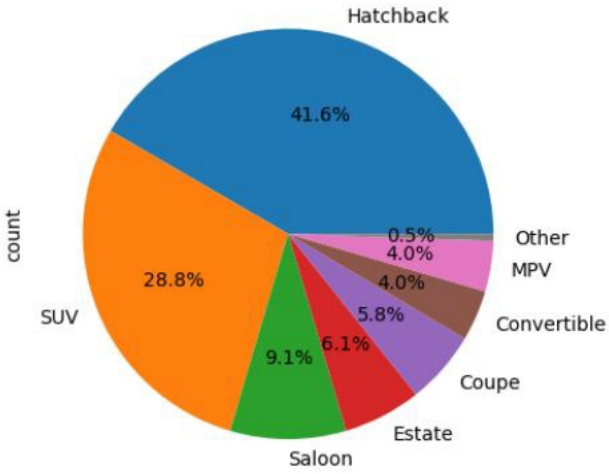
```
bins = [120, 7495, 10000, 12600, 16300, 20000, 9999999]
labels = ["120-7495","7495-10000", "10000-12600","12600-16300", "16300-20000", "20000-9999999"]
price_binned = pd.cut(df["price"], bins, right=True, labels=labels)
bins_on_x = price_binned.value_counts().sort_index()
```



Vehicle Body Type Distribution

To streamline the analysis of vehicle body types in the dataset, a threshold-based grouping approach has been applied. Based on a defined occurrence threshold, the distribution of vehicle body types are visualized by aggregating less common types into a single "Other" category. Threshold Setting: A count threshold of 16,000 has been set. Body types occurring fewer than 16,000 times in the dataset are classified as "Other".

```
body_type_counts = df["body_type"].value_counts(dropna=False)
threshold = 16000
df_other = df.copy()
df_other['body_type_grouped']  = df_other['body_type'].apply(lambda x: x if body_type_counts[x] > 16000 else "Other")
```



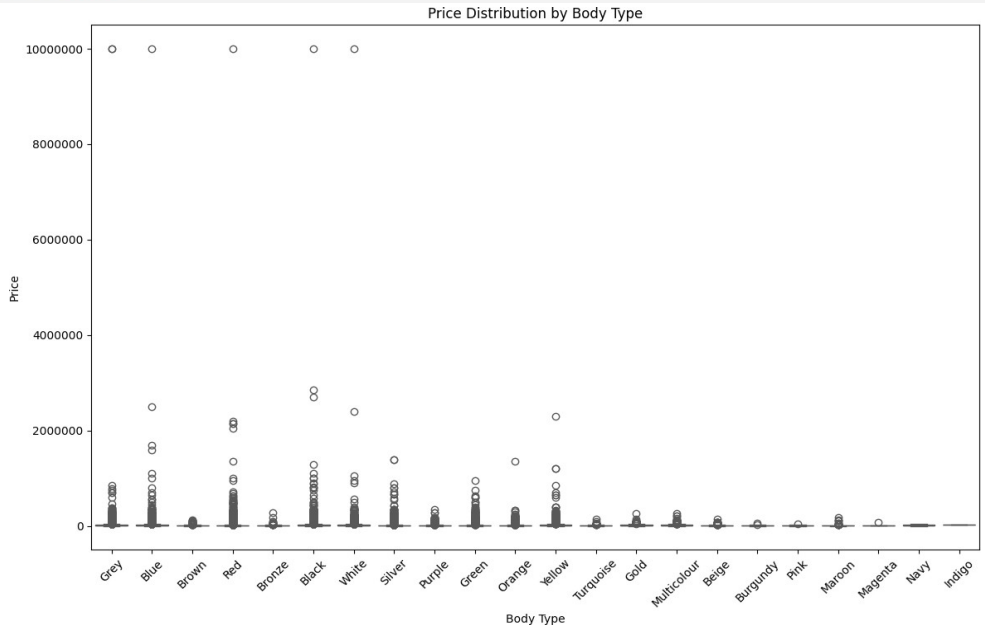
Data Processing for Machine Learning

Dealing with Missing Values, Outliers, and Noise

Dealing with Outliers in Vehicle Pricing

This step involves identifying and removing the outliers to ensure more accurate and representative statistical analysis.

```
plt.figure(figsize=(12, 8))
sns.boxplot(data=df, x='standard_colour', y='price', palette='Set2')

plt.title('Price Distribution by Body Type')
plt.xlabel('Body Type')
plt.ylabel('Price')
```



By removing these high-price outliers, the dataset's price distribution becomes more normalized.

```
indices_to_drop = df.loc[df['price'] == 9999999].index
df.drop(indices_to_drop, inplace= True)
```

Imputation of Missing 'Year of Registration' Values Using Mode

For the 'year_of_registration' column, missing values were addressed by imputing the mode of the column. The mode provides a statistically reasonable estimate to fill the missing values as it helps preserve the main characteristics of the data.

```
mode_value_of_year_of_registration = df["year_of_registration"].mode()[0]
df["year_of_registration"].fillna(mode_value_of_year_of_registration, inplace=True)
df["year_of_registration"].isna().value_counts()
```

✓ 0.0s

```
year_of_registration
False    401999
Name: count, dtype: int64
```

Feature Engineering, Data Transformations, Feature Selection

Feature Scaling Using MinMaxScaler

To normalize the numeric features and have them on a comparable scale, the “MinMaxScaler” was applied to the following columns: 'mileage', 'year_of_registration', and 'price'. This scaling transforms the data to a range of 0 to 1 to standardize numeric features.

```
from sklearn.preprocessing import MinMaxScaler
model = MinMaxScaler()
df.loc[: , ['mileage', 'year_of_registration','price']] = model.fit_transform(df.lo
df.head(3)
```

	mileage	standard_colour	standard_make	vehicle_condition	year_of_registration	price	body_type	crossover_car_and_van	fuel_type
0	0.000000	Grey	Volvo	NEW	0.842105	0.061485	SUV	False	Petrol Plug-in Hybrid
3	0.321429	Brown	Vauxhall	USED	0.789474	0.006497	Hatchback	False	Diesel
4	0.457143	Grey	Land Rover	USED	0.736842	0.022333	SUV	False	Diesel

One-Hot Encoding of Categorical Variables

The one-hot encoding is used to convert categorical features to numeric ones. This is crucial as machine learning models need numeric data. Each unique category in the 'vehicle_condition' and 'crossover_car_and_van' columns is transformed into a new feature column without adding to the number of columns because these columns have only two kind of values which removes the need of creating new columns.

```
X = df.drop(columns=["price"])
y = df[["price"]]
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder(sparse_output=False, drop='if_binary')
```

	mileage	standard_colour	standard_make	vehicle_condition	year_of_registration	body_type	crossover_car_and_van	fuel_type
0	0.000000	Grey	Volvo	0.0	0.842105	SUV	0.0	Petrol Plug-in Hybrid
3	0.321429	Brown	Vauxhall	1.0	0.789474	Hatchback	0.0	Diesel

Model Building

Algorithm Selection, Model Instantiation and Configuration

Model Training: k-Nearest Neighbors, Decision Tree, and Linear Regression

Three models are trained on the dataset.

Hyperparameter Configuration: Parameters for grid search included 'max_depth', 'min_samples_split', and 'min_samples_leaf' with specific ranges, such as 2, 5, and 10 for ‘min_samples_split’.

```
knn = KNeighborsRegressor()
knn.fit(X_train, y_train)
```

✓ 0.1s

```
KNeighborsRegressor
KNeighborsRegressor()
```

```
tree = DecisionTreeRegressor()
tree.fit(X_train, y_train)
tree_params = {'max_depth': [3, 5, 10, 20],
               'min_samples_split': [2, 5, 10],
               'min_samples_leaf': [1, 5, 10, 50]}
```

✓ 1.4s

```
linear = LinearRegression(fit_intercept=False)
linear.fit(X_train, y_train)
```

✓ 0.3s

```
LinearRegression
LinearRegression(fit_intercept=False)
```

Grid Search, and Model Ranking and Selection

Grid Search for Hyperparameter Optimization

GridSearchCV object is used to perform hyperparameter tuning for the DecisionTreeRegressor model. Best Parameters Identified are: 'max_depth': 20, 'min_samples_leaf': 10, 'min_samples_split': 5

```
clf = GridSearchCV(tree, tree_params, return_train_score=True)
gs_results = clf.fit(X_train, y_train)
gs_results.best_params_
```

```
{'max_depth': 20, 'min_samples_leaf': 10, 'min_samples_split': 10}
```

The columns of this dataframe provide a comprehensive view of how different combinations of hyperparameters performed during the grid search process. It helps in identifying the most optimal settings for the model.

```
gs_df_2 = pd.DataFrame(gs_results.cv_results_)
gs_df_2.columns
```

```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_max_depth', 'param_min_samples_leaf', 'param_min_samples_split',
       'params', 'split0_test_score', 'split1_test_score', 'split2_test_score',
       'split3_test_score', 'split4_test_score', 'mean_test_score',
       'std_test_score', 'rank_test_score', 'split0_train_score',
       'split1_train_score', 'split2_train_score', 'split3_train_score',
       'split4_train_score', 'mean_train_score', 'std_train_score'],
      dtype='object')
```

Top Grid Search Results

The DataFrame 'gs_df_2_sh' displays the top 4 results from the grid search ranked based on the best 'rank_test_score'. The columns provide insights into the performance of different parameter combinations. The 'mean_test_score' for the best parameter set is 0.7373, with a standard deviation of 0.047. The score shows the performance is almost reliable on the test data.

```
gs_df_2_sh = gs_df_2[ ['param_max_depth', 'param_min_samples_leaf', 'param_min_samples_split',
    'mean_train_score', 'std_train_score',
    'mean_test_score', 'std_test_score', 'rank_test_score'
] ].sort_values('rank_test_score')
```

	param_max_depth	param_min_samples_leaf	param_min_samples_split	mean_train_score	std_train_score	mean_test_score	std_test_score	rank_test_score
44	20	10	10	0.737338	0.012423	0.675535	0.047728	1
42	20	10	2	0.737338	0.012423	0.675526	0.047732	2

Model Evaluation and Analysis

Coarse-Grained Evaluation/Analysis

Model Evaluation: K-Nearest Neighbors Training Score

The 'score' method evaluates the performance of the K-Nearest Neighbors (KNN) model on the training dataset.

The score for the KNN model on the training data is 0.7432. It indicates that the model covers approximately 74.32% of the variance in the training data.

```
knn.score(X_train, y_train)
```

✓ 46.2s

```
0.743221027874299
```

Mean Absolute Error for Linear Regression

Purpose: The Mean Absolute Error (MAE) measures the average magnitude of errors between the predicted and actual values. It provides insight into the prediction accuracy of the linear regression model. The MAE was calculated by averaging the absolute differences between predicted and actual values in y_test. The MAE for the linear regression model is 0.0056. This indicates that the model's predictions deviate from the actual values by approximately 0.0056 units.

```
linear.fit(X_train, y_train)
mae = mean_absolute_error(y_test, linear.predict(X_test))
mae
```

✓ 0.2s

```
0.0056016045901811935
```

Cross-Validation Score for Decision Tree Regressor

The mean cross-validation score for the decision tree model is 0.6755. This score is relatively higher than that of simpler models like linear regression.

```
tree_2 = DecisionTreeRegressor(max_depth= 20, min_samples_leaf = 10, min_samples_split = 5)
scores = cross_val_score(tree_2, X_train, y_train, cv=5)
scores.mean()
```

✓ 2.5s

```
0.6755114538602205
```

Feature Importance

Extract Feature Importances

Tree-based models calculate feature importance based on the reduction in the impurity achieved by splitting on a particular feature.

The feature mileage has the highest importance score of 0.4722, indicating it is the most influential variable in predicting the target.

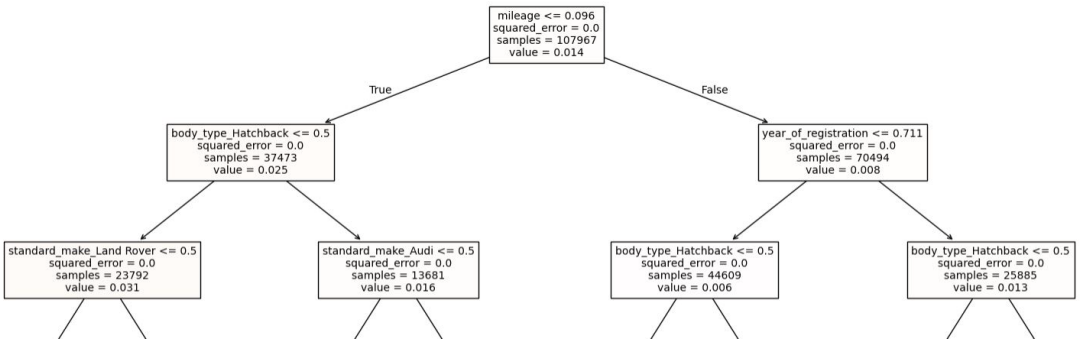
```
feature_importances = tree.feature_importances_
feature_names = X_train.columns

importance_df = pd.DataFrame({
    "Feature": feature_names,
    "Importance": feature_importances
```

	Feature	Importance
0	mileage	0.472463
30	body_type_Hatchback	0.115147
2	year_of_registration	0.087954
19	standard_make_Land Rover	0.073568

Structure of Decision Tree Regressor

Features that appear in the early layers of a decision tree tend to be more important because they have a larger impact on the resulting prediction. The visualized tree proves the previous achieved results as it represents again that 'mileage', 'year_of_registration' and generated 'body_type_Hatchback'feature are the most important features of the model.

```
from sklearn.tree import plot_tree
plt.figure(figsize=(20,10))
```



Fine-Grained Evaluation

Fine-grained Evaluation with Predictions vs. Actual Values

The first few predictions of the KNeighborsRegressor model (y_pred) with the actual target values from the test dataset (y_test) are compared. The predictions are relatively close to the actual values for the first five instances. For instance, the first predicted value (0.0029) is slightly higher than the actual value (0.0014). This suggests that the model performs well.

```
y_pred = knn.predict(X_test)
y_pred[:5]
```
✓ 11.6s

```
array([[0.00292165],
       [0.00735606],
       [0.02316619],
       [0.00389032],
       [0.01639073]])
```

```
y_test.head(5).to_numpy()
```
✓ 0.0s

```
array([[0.00141274],
       [0.00759627],
       [0.02317053],
       [0.00548425],
       [0.01962827]])
```

Confusion Matrix

The confusion matrix is represented, which measures predictive performance of the model. The result indicates that almost all of the actual values fall between 0-0.25, when all the model's predicted values are also in that class.

```
bins = [0, 0.25, 0.5, 0.75, 1]
labels = ['0-0.25', '0.25-0.5', '0.5-0.75', '0.75-1']

y_test_binned = pd.cut(y_test['price'], bins=bins, labels=labels, include_lowest=True)
y_pred = y_pred.ravel()
y_pred_binned = pd.cut(y_pred, bins=bins, labels=labels, include_lowest=True)

confusion_matrix = pd.crosstab(y_test_binned, y_pred_binned, rownames=['Actual'], colnames=['Predicted'])
plt.figure(figsize=(7, 6))
sns.heatmap(confusion_matrix, annot=True, fmt='d', cmap='Blues', linewidths=.5)
```