Machine Learning Concepts

1. Data/Domain Understanding and Exploration:

The dataset provided for ML assignment shows the information about Vehicle registration plates of the United Kingdom. It has rows and columns, showing various car attributes to predict vehicle selling prices.

Figure 1:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 402005 entries, 0 to 402004
Data columns (total 12 columns):
 #   Column                Non-Null Count   Dtype
---  ------                --------------   -----
 0   public_reference      402005 non-null  int64
 1   mileage               401878 non-null  float64
 2   reg_code              370148 non-null  object
 3   standard_colour       396627 non-null  object
 4   standard_make         402005 non-null  object
 5   standard_model        402005 non-null  object
 6   vehicle_condition     402005 non-null  object
 7   year_of_registration  368694 non-null  float64
 8   price                 402005 non-null  int64
 9   body_type             401168 non-null  object
 10  crossover_car_and_van 402005 non-null  bool
 11  fuel_type             401404 non-null  object
dtypes: bool(1), float64(2), int64(2), object(7)
memory usage: 34.1+ MB
```

Figure 2:

	0
public_reference	402005
mileage	80634
reg_code	72
standard_colour	22
standard_make	110
standard_model	1168
vehicle_condition	2
year_of_registration	84
price	30578
body_type	16
crossover_car_and_van	2
fuel_type	9

dtype: int64

Figure 3:

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

dtype: int64

Fig 1 provides the information about the dataset:

- **Size:** The dataset has 40,205 entries.
- **Columns:** It has 12 columns.
- **Non-Null Count:** It shows the number of non-missing values for each column.
- **Data Type:** It states the data types for each column.

Fig 2 is a output of df.isnull().sum() query which indicates the missing values of car features.

- **No Missing Values:** The features public_reference, standard_make, standard_model, vehicle_condition, price, crossover_car_and_van have 0 missing values and do not require any data processing.
- **Moderate Missing Values:** The features like mileage, fuel_type, and body_type have missing values such as 127, 601, and 837 respectively.
- **Significant Missing Values:** The features like reg_code (31,857), year_of_registration (33,311), and standard_colour (5378)

Fig 3 shows the uniqueness of features of a given dataset and can be categorizes as follows:

- **Most Unique Features:** These features consist of highest number of unique values such as standard_model(1186), standard_make(110) and year_of_registration(84).
- **Moderately Unique Features:** These features have a moderate number of unique values such as price (30,578), mileage (80634), body_type (16) and standard_colour(22).
- **Less Unique Features:** These features contain less unique values followed by vehicle_condition(2), and fuel_type(9).

Figure 1:

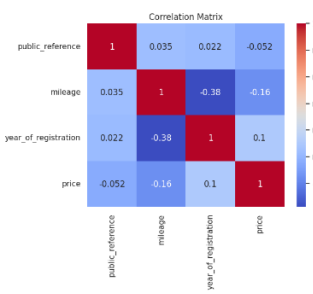
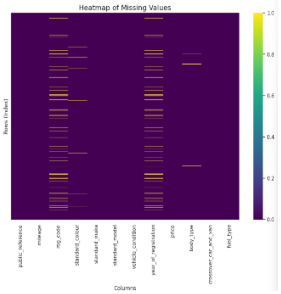


Figure 2:



Above **Fig 1** and **Fig 2** are Correlation Matrix and Heatmap of Missing Values, respectively,

Correlation Matrix: Correlation is a statistical measure that describes the relationship between two variables.

1. Mileage and Price: There is a negative correlation of -0.16 between mileage and price which means they are inversely proportional to each other. If mileage increases, the price tends to go down, often means that the car has been driven more.

2. Year of Registration and Price: There is a moderate positive correlation (0.1) between year of registration and price. This indicates that newer cars tend to be more expensive.

Heatmap of Missing Values: The heatmap visualizes missing values in a dataset.

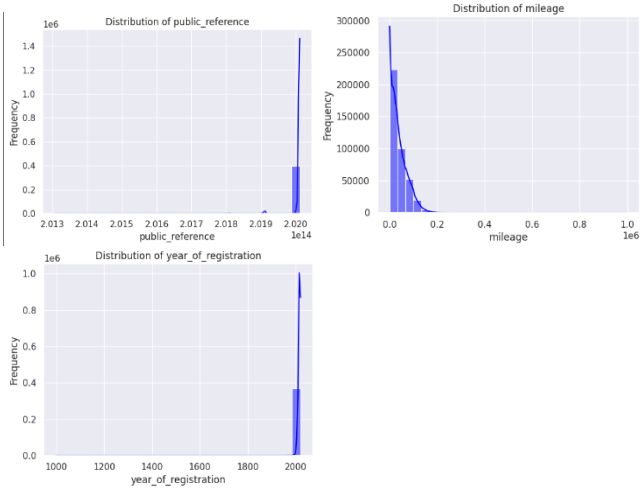
- The columns mileage, standard_colour, standard_make, standard_model, vehicle_condition, year_of_registration, and price have multiple missing values.
- The columns reg_code, body_type, crossover_car_and_van, and fuel_type have a few missing values.
- This column has no missing values, indicated by the absence of yellow lines.

```
[ ] numeric_data = df.select_dtypes(include=['int64', 'float64']).columns
    discrete_features = df.select_dtypes(include=['object', 'bool']).columns

    print(f"Numeric data: {numeric_data}", "\n" f"Discrete Features: {discrete_features}")

Numeric data: Index(['public_reference', 'mileage', 'year_of_registration', 'price'], dtype='object')
Discrete Features: Index(['reg_code', 'standard_colour', 'standard_make', 'standard_model',
       'vehicle_condition', 'body_type', 'crossover_car_and_van', 'fuel_type'],
      dtype='object')
```

Later, divided the dataset into 2 categories, numeric_data (features with numerical values like integers and floats) and discrete_features (categorical or binary values).

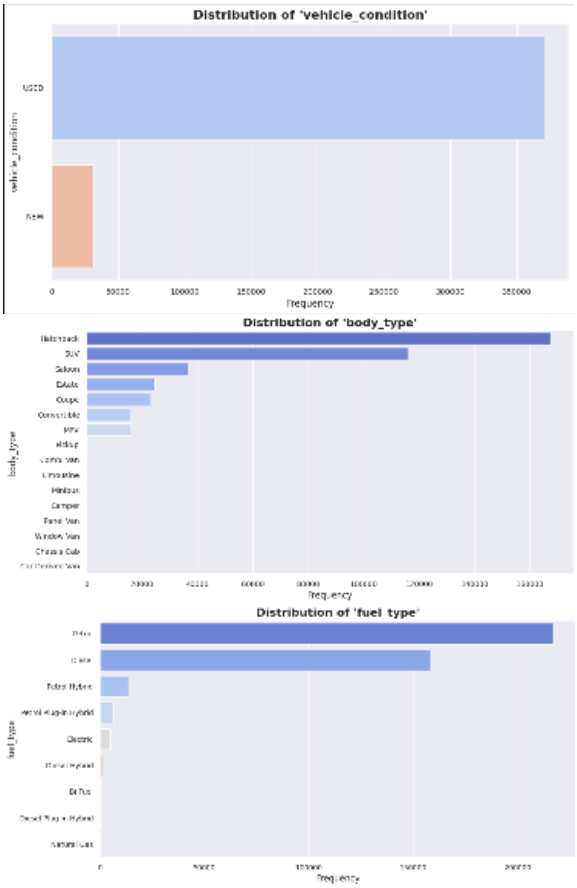


These are histograms which we use to visualize the distribution of numeric features.

1. Distribution of public_reference: It plots a unique identifier distribution. The concentration at the end shows most values are concentrated around a narrow range and hence there is variation in this feature.

2. Distribution of mileage: It is very skewed, most of the cars have low mileage and a few have high mileage. It is an important to understand the range and possible transformation needs.

3. Distribution of year_of_registration: As expected from vehicle data, most of the values are concentrated in recent years.



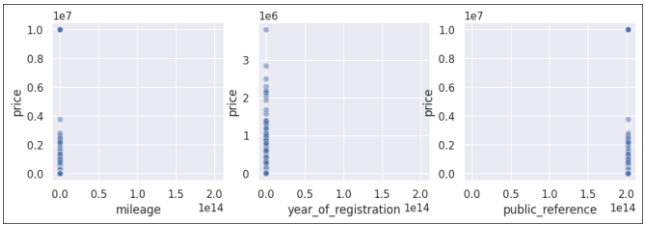
1. Distribution of 'vehicle_condition': This indicates that the dataset consists more used vehicles as compared to new vehicles.

2. Distribution of 'body_type': According to the data, hatchbacks are the most popular. SUVs and saloons have consumer preferences for larger vehicles.

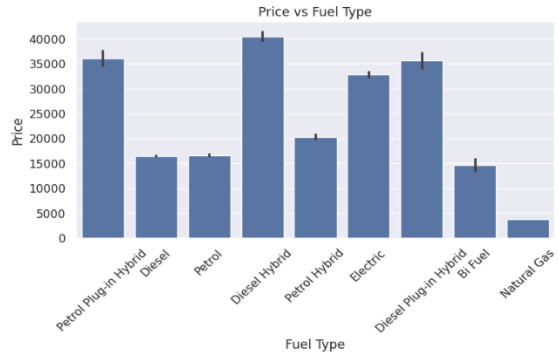
3. Distribution of 'fuel_type': The power of petrol and diesel cars over other reflects traditional trends in automotive marketplace. Also, hybrid and electric vehicles may consider as eco-friendly alternatives.

For crossover_car_and_van, the table is not that much useful hence dropped later.

The colours on x-axis are standard_colour printed for reference. Missing values are filled with unknown. The colours Yellow and Inidgo have the highest average prices while colour pink has the lowest average price.



This scatter plots showing relationship between price and 3 different features like mileage, year_of_registration and public_reference. The public_reference feature is not relevant for price predicting while other two features need to be scaled properly for analysis of price.



The x-axis represents different fuel types while y-axis represents the average price of vehicles for each fuel type. Diesel Hybrid has the highest average price, followed closely by Petrol Plug-in Hybrid and Diesel Plug-in Hybrid. Natural Gas has the lowest average price. This indicate that vehicles using natural gas are less in demand. Electric vehicles have high average prices while diesel and petrol have almost similar average prices.

2. Data Processing for Machine Learning:

```
df.drop(df.loc[(df['year_of_registration'].isnull()) & (df['vehicle_condition'] == 'NEW') & (df['reg_code'].isnull())].index, inplace=True)

modes = (
    df.groupby(['standard_make', 'standard_model'])['standard_colour']
    .agg(lambda x: x.mode().iloc[0] if not x.mode().empty else None)
    .reset_index()
)
modes.rename(columns={'standard_colour': 'mode_standard_colour'}, inplace=True)

modes.rename(columns={'standard_colour': 'mode_standard_colour'}, inplace=True)
df = df.merge(modes, on=['standard_make', 'standard_model'], how='left')

df['standard_colour'] = df['standard_colour'].fillna(df['mode_standard_colour'])

df.drop(columns=['mode_standard_colour'], inplace=True)
```

filler1-> df.groupby('standard_make', 'standard_model')[column to be filled].transform(lambda function for mode) Applying below query because, it's not possible to apply filter 1 fro remaining 24 columns

```
df['standard_colour'].fillna(df['standard_colour'].mode()[0], inplace=True)
```

```
#Columns standard_make and standard_model has relation with body_type.
modes = (
    df.groupby(['standard_make', 'standard_model'])['body_type']
    .agg(lambda x: x.mode().iloc[0] if not x.mode().empty else None)  # Calculate mode
    .reset_index()
)
modes.rename(columns={'body_type': 'mode_body_type'}, inplace=True)

modes.rename(columns={'body_type': 'mode_body_type'}, inplace=True)
df = df.merge(modes, on=['standard_make', 'standard_model'], how='left')

# fill missing values in 'body_type' with the computed mode
df['body_type'] = df['body_type'].fillna(df['mode_body_type'])

df.drop(columns=['mode_body_type'], inplace=True)
```

After above query there are still 70 data is missing, thus we are applying below query.

```
df['body_type'].fillna(df['body_type'].mode()[0], inplace=True)
```

```
#Columns standard_make and standard_model has relation with body_type.
modes = (
    df.groupby(['standard_make', 'standard_model'])['body_type']
    .agg(lambda x: x.mode().iloc[0] if not x.mode().empty else None)  # Calculate mode
    .reset_index()
)
modes.rename(columns={'body_type': 'mode_body_type'}, inplace=True)

modes.rename(columns={'body_type': 'mode_body_type'}, inplace=True)
df = df.merge(modes, on=['standard_make', 'standard_model'], how='left')

# fill missing values in 'body_type' with the computed mode
df['body_type'] = df['body_type'].fillna(df['mode_body_type'])

df.drop(columns=['mode_body_type'], inplace=True)
```

After above query there are still 70 data is missing, thus we are applying below query.

```
df['body_type'].fillna(df['body_type'].mode()[0], inplace=True)
```

```
#Columns standard_make, body_type and standard_model has relation with fuel_type.
modes = (
    df.groupby(['standard_make', 'standard_model', 'body_type'])['fuel_type']
    .agg(lambda x: x.mode().iloc[0] if not x.mode().empty else None)  # Calculate mode
    .reset_index()
)
modes.rename(columns={'fuel_type': 'mode_fuel_type'}, inplace=True)

modes.rename(columns={'fuel_type': 'mode_fuel_type'}, inplace=True)
df = df.merge(modes, on=['standard_make', 'standard_model', 'body_type'], how='left')

#fill missing values in 'fuel_type' with the computed mode
df['fuel_type'] = df['fuel_type'].fillna(df['mode_fuel_type'])

df.drop(columns=['mode_fuel_type'], inplace=True)
```

```
df['fuel_type'].fillna(df['fuel_type'].mode()[0], inplace=True)
```

Below Handling the reg_code and year_of_registration

```
def calculate_year(reg_code):
    if pd.isnull(reg_code) or not str(reg_code).isdigit():
        return None
    reg_code = int(reg_code)

    if reg_code <= 50:
        return 2000 + reg_code
    else:
        return 2000 + (reg_code - 50) + 1

df['calculated_year'] = df['reg_code'].apply(calculate_year)
df[['reg_code', 'calculated_year']]
```

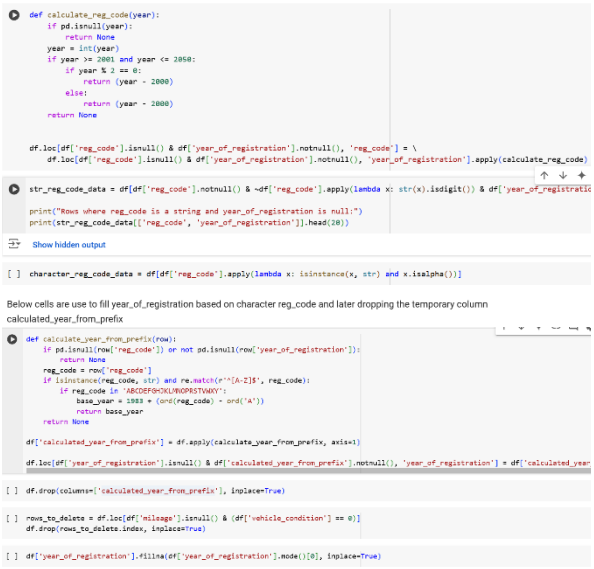
Show hidden output

```
df.loc[
    df['reg_code'].notnull() & df['year_of_registration'].isnull() & df['reg_code'].apply(lambda x: str(x).isdigit()),
    'year_of_registration'
] = df['calculated_year']

df.drop(columns=['calculated_year'], inplace=True)

df['vehicle_condition'] = df['vehicle_condition'].map({ 'USED': 0, 'NEW': 1, np.NaN: 0} )

df['crossover_car_and_van'] = df['crossover_car_and_van'].replace({False: 0, True: 1})
```



All above figures provide the step-by-step visualization of data processing for model building.

Below is the explanation of data processing code:

1. From dataset, whose year_of_registration & reg_code is null and vehicle_condition is new, these rows are irrelevant as new vehicles must have these details.

2. To fill the standard_colour and body_type, used the mode of two columns standard_make and standard_model.

3. If no mode is available and still there is missing data, used the overall mode of each column of standard_colour and body_type.

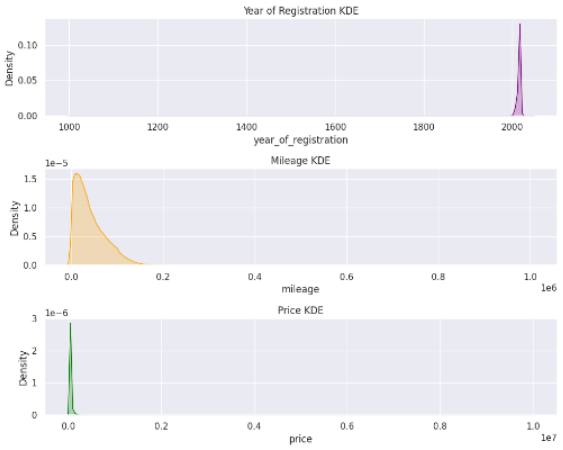
4. There is a relationship between standard_make, standard_model, and body_type with fuel_type. To ensure the accuracy, uses grouped mode method and overall mode to fill missing values.

5. As per the Wikipedia link there is a relation between year_of_registration and reg_code. For filling the reg_code, used year_of_registration and vice versa. calculate_year(reg_code) function is used to calculate a year based on registration code. calculate_year_from_prefix(row) function is used to calculate a year based on a prefix.

6. Convert the categorical columns (vehicle_condition & crossover_car_and_van) into numerical format for further modeling.

7. Removed the data whose mileage is null and vehicle_condition is used, as it not possible to have a used car with no mileage.

8. For any missing values remaining in year_of_registration is filled by using mode.



These KDE plots are easier to observe the overall potential outliers.

1. Year of Registration Plot: The most of the data is around the expected range (close to 2000). The values like 1000 or 1500 shows outliers.

2. Mileage Plot: For new vehicles, plot shows density curve near low mileage. The tail of curve extends towards higher mileage, which is indicating outliers.

3. High prices far from peak shows the outliers or rare vehicles.

Above image explains the outlier handling code. The public_reference column is excluded from outliers processing as it is not necessary. To handle outliers, it uses IQR (Interquartile range) method for rest of the columns such as 'mileage', 'year_of_registration', 'price'. IQR shows the values which fall outside the range Q1 − 1.5 × IQR to Q3 + 1.5 × IQR. These help to make sure the data should fit within the reasonable range and does not affect further analysis.

Further, the unwanted columns such as reg_code, crossover_car_and_van, and public_reference were dropped.

```
df_encoded = pd.get_dummies(df['fuel_type'], prefix='fuel_type').astype(int)  # Convert to 0/1 explicitly

df[df_encoded.columns] = df_encoded

df.drop(columns=['fuel_type'], inplace=True)
```

```
[52] df['price_category'] = np.random.choice([0, 1], size=len(df))

columns_to_encode = ['standard_colour', 'standard_make', 'standard_model', 'body_type']

# Sometime this doesn't work, so added alternative code for transformation
# encoder = TargetEncoder(cols=columns_to_encode)
# df[columns_to_encode] = encoder.fit_transform(df[columns_to_encode], df['price_category'])

for col in columns_to_encode:
    mapping = df.groupby(col)['price_category'].mean().to_dict()
    df[col] = df[col].map(mapping)

df.drop(columns=['price_category'], inplace=True)
```

```
[53] scaler = StandardScaler()

df[['mileage', 'price', 'year_of_registration']] = scaler.fit_transform(df[['mileage', 'price', 'year_of_registration']])
```

The pd.get_dummies() function used to perform one-hot encoding on fuel_type. It converts categorical data into a new binary column explicitly contains 1 or 0. After encoding the fuel_type column is no longer needed and dropped using df.drop(). OneHotEncoding is not useful to convert categorical columns (standard_make, standard_colour, standard_model, body_type) into numeric, thus Target Encoding were used. Target encoding replaces the values in categorical with mean of the target variable price_category. After encoding, the price_category column is no longer needed.

StandardScaler ensures that the features mileage, price, year_of_registration have the same scale and distribution, which can help many machine learning models perform better.

3. Model Building:

```
X = df.drop(columns="price")
y = df["price"]
```

```
[56] X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
     X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
```

```
[57] print("Shapes of datasets:")
     print("X_train:", X_train.shape, "y_train:", y_train.shape)
     print("X_val:", X_val.shape, "y_val:", y_val.shape)
     print("X_test:", X_test.shape, "y_test:", y_test.shape)

Shapes of datasets:
X_train: (259440, 16) y_train: (259440,)
X_val: (55594, 16) y_val: (55594,)
X_test: (55595, 16) y_test: (55595,)
```

```
[58] df.isna().sum().sum()

0
```

1. Feature and Target Separation:
    a. X = df.drop(columns="price"): This separates the feature variables into X. Except price, leaving all other columns as features.
    b. y = df["price"]: This selects price column as target variable y
2. Training Data: train_test_split(X, y, test_size=0.3, random_state=42) divides feature variable x and target variable y into training and temporary (validation + test) datasets. 30% of the data is allocated for the temporary, remaining 70% for the training.
3. Validation and Test Data: train_test_split(X_temp, y_temp, test_size=0.5, random_state=42) this divides the previously created temporary dataset into validation and test. 50% of X_temp and y_temp is allocated to test, and the remaining 50% allocates to the validation set.
4. The shapes are printed to give an overview of the dataset sizes for each split.

Sklearn is used to perform Linear Regression, KNN and Decision Tree Regressor.

Linear Regression

```
[59] lr_model = LinearRegression()
     lr_scores = cross_val_score(lr_model, X_train, y_train, cv=5, scoring='neg_mean_squared_error')
     lr_rmse = np.sqrt(-lr_scores)
     print(f"Cross-Validation RMSE: {lr_rmse}")
     print(f"Mean RMSE: {lr_rmse.mean()}")

Cross-Validation RMSE: [0.76952326 0.77145921 0.76781896 0.77459778 0.77230989]
Mean RMSE: 0.7711418197205486
```

```
[60] print("\n--- Validation Evaluation ---")
     # Linear Regression
     lr_model.fit(X_train, y_train)
     lr_val_preds = lr_model.predict(X_val)
     lr_val_rmse = np.sqrt(mean_squared_error(y_val, lr_val_preds))
     print(f"Linear Regression Validation RMSE: {lr_val_rmse}")

--- Validation Evaluation ---
Linear Regression Validation RMSE: 0.7738352876245543
```

This code is used to perform Linear Regression using cross validation and validation set evaluation.

- cross_val_score function performs k-fold cross-validation with cv=5, that means training data is divided into 5 equal-sized folds.
- The model is trained on 4 folds and estimated on the remaining 1-fold. Negative mean squared error performance metric is used.
- Then calculate the root mean squared error (RMSE) using np.sqrt. lr_rmse stores the output of RMSE. The mean RMSE (Mean RMSE) is calculated to explore model's overall performance.
- The entire training dataset is used to train Linear Regression model. lr_model.predict used on trained model to predict target variable.
- The output RMSE (0.7738) and mean RMSE (0.7711), shows that model generalizes reasonably well to the validation set.

KNN

```
knn_model = KNeighborsRegressor()

knn_params = {
    'n_neighbors': [7],
    'weights': ['distance'],
    'metric': ['euclidean']
}

knn_grid = GridSearchCV(knn_model, knn_params, scoring='neg_mean_squared_error', n_jobs=-1)

# Fit the model
knn_grid.fit(X_train, y_train)

# Print results
print(f"Best Parameters: {knn_grid.best_params_}")
print(f"Best RMSE: {np.sqrt(-knn_grid.best_score_)}")

Best Parameters: {'metric': 'euclidean', 'n_neighbors': 7, 'weights': 'distance'}
Best RMSE: 0.6692630157595624
```

```
[ ] knn_best = knn_grid.best_estimator_
    knn_val_preds = knn_best.predict(X_val)
    knn_val_rmse = np.sqrt(mean_squared_error(y_val, knn_val_preds))
    print(f"KNN Validation RMSE: {knn_val_rmse}")

KNN Validation RMSE: 0.655582503071795
```

The code evaluates a K-Nearest Neighbors (KNN) Regressor model using grid search.

- Parameters (n_neighbors: 7, weights: distance, metric: euclidean) are used for tuning. These parameters found best hyperparameters by grid search. Best RMSE: 0.6692 is found by using these parameters.
- Same as linear regression negative mean squared error is used as the scoring matric. The KNN validation RMSE = 0.6555 is calculated between knn_val_pred (predicted) and y_val (y_val).
- This process ensures that the model is improved for the given dataset.

Decision Tree Regressor

```
dt_model = DecisionTreeRegressor(random_state=42)
dt_params = {
    'max_depth': [5, 10, None],
    'min_samples_split': [5, 10],
    'min_samples_leaf': [1, 2]
}

dt_grid = GridSearchCV(dt_model, dt_params, cv=5, scoring='neg_mean_squared_error')
dt_grid.fit(X_train, y_train)
print(f"Best Parameters: {dt_grid.best_params_}")
print(f"Best RMSE: {np.sqrt(-dt_grid.best_score_)}")
```

```
Best Parameters: {'max_depth': None, 'min_samples_leaf': 2, 'min_samples_split': 10}
Best RMSE: 0.28513107978798363
```

```
# Decision Tree
dt_best = dt_grid.best_estimator_
dt_val_preds = dt_best.predict(X_val)
dt_val_rmse = np.sqrt(mean_squared_error(y_val, dt_val_preds))
print(f"Decision Tree Validation RMSE: {dt_val_rmse}")
```

```
Decision Tree Validation RMSE: 0.2794617237270336
```

- Parameters (max_depth: [5, 10, None], min_samples_split: [5, 10], min_samples_leaf: [1, 2]) are used for tuning. After tunning, it is found that max_depth: None, min_samples_leaf: 2, min_samples_split: 10 are the best parameters and generates the result as Best Cross-Validation RMSE: 0.2851 and Validation RMSE: 0.2795.
- This shows that chosen hyperparameters enhance the model's performance on the dataset.

```
--- Linear Regression Metrics ---
R² Score: 0.40117625517874667
Mean Squared Error: 0.5988210523729768
Mean Absolute Error: 0.5780942063327613

--- KNN Metrics ---
R² Score: 0.5702096492305758
Mean Squared Error: 0.42978841833388004
Mean Absolute Error: 0.4398668127480416

--- Decision Tree Metrics ---
R² Score: 0.9219007938196531
Mean Squared Error: 0.07809885502848486
Mean Absolute Error: 0.17033553143173966
```

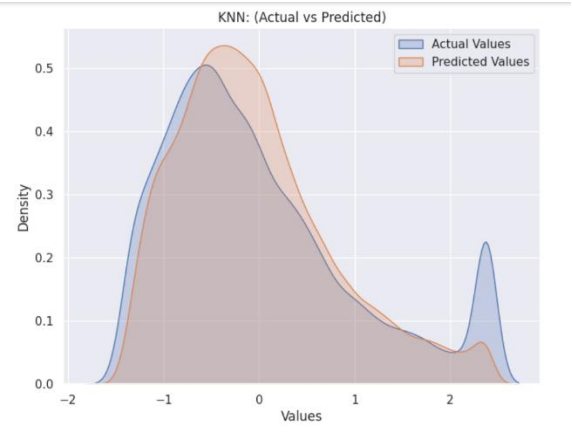
1. Linear Regression:
   a. R^2 Score: 0.4011
      i. The target variable shows 40.11% variance by Linear Regression model.
      ii. Wear fit suggests by low value.
   b. Mean Square Error (MSE): 0.5988
      i. MSE measures the average squared difference between predicted and actual values.
   c. Mean Absolute Error (MAE): 0.5781
      i. MAE measures the average absolute difference between predicted and actual values.

2. K-Nearest Neighbors (KNN):
   a. R^2 Score: 0.5702
      i. The target variable shows 57.02% variance by KNN.
      ii. It is better fir that Linear Regression but yet nit the perfect one.
   b. Mean Squared Error (MSE): 0.4298
      i. It is lower than Linear Regression, showing better predictive performance.
   c. Mean Absolute Error (MAE): 0.4399
      i. It is lower than Linear Regression, showing smaller prediction errors on average.

3. Decision Tree Metrics:
   a. R^2 Score: 0.9219
      i. The target variable shows 92.19% variance by Decision Tree Model.
      ii. It is a very strong fir, shows that the model captures the data well.
   b. Mean Squared Error (MSE): 0.0781
      i. As compared to other two models, it is much smaller which shows highly accurate predictions.
   c. Mean Absolute Error (MAE): 0.1703
      i. It is very smallest absolute error among other two models, stating the Decision Tree's superior performance.

4. Model Evaluation and Analysis:

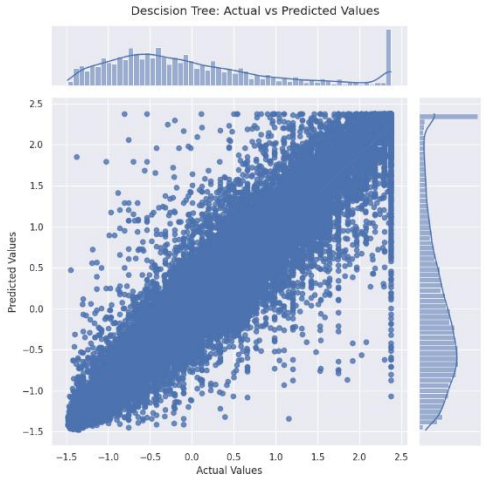
➢ KNN:



The above plot compares the distribution of actual values and predicted values for a KNN model to calculate its performance.

1. The blue curve shows the distribution of actual values whereas orange curve represents the distribution of predicted values.
2. The overlap between two curves specifies the model's prediction match the actual values.
3. If the curves are closely aligned with each other, it means model's prediction is similar to actual values.
4. Whereas gaps suggest the model might not be performing well.

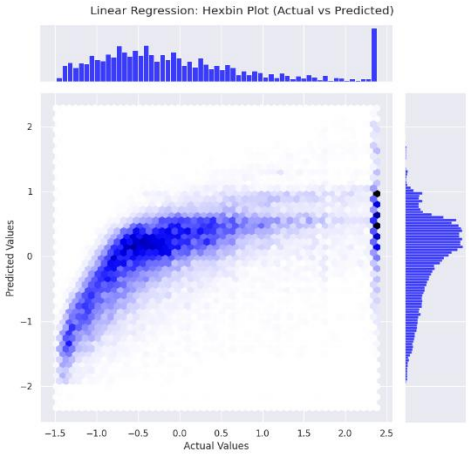
➢ Decision Tree:



This scatter plot with marginal histograms evaluates the performance of Decision Tree by comparing actual and predicted values.

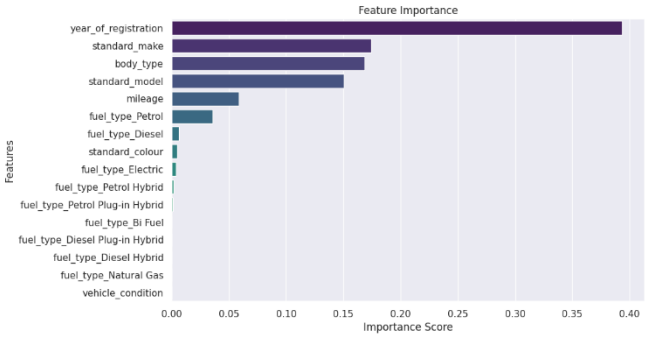
1. In the above plot, x-coordinate (top) is the actual values and y-coordinate (right) is the predicted values.
2. The closer points near to the diagonal line i.e., actual = predicted, the better performance of the model.
3. If the points are close to the diagonal, it means model's prediction is similar to actual values.
4. Spread points highlights where the model struggles to make accurate prediction.

➢ Linear Regression:



This hexbin plot compares the actual and predicted values of a Linear Regression model to evaluate its performance.

1. In the above plot, x-coordinate (top) is the actual values and y-coordinate (right) is the predicted values.
2. The hexagonal bins represent dark regions which suggest good performance.
3. If spread is observed, it indicates further model improvement is needed.



The above bar chart represents the **Feature Importance** of a machine learning model.

1. The x-axis shows the Importance score whereas y-axis shows the Features.
2. Higher score of a feature means the higher impact.
3. The features are sorted from most important to least important.
4. The year_of_registration and standard_make have the highest importance scores.
5. Features like body_type, standard_model, and mileage have medium importance scores.

6. Features like fuel_type_Electric or vehicle_condition have very low importance scores.

7. Feature importance can guide to focus on impactful features whereas unimportant or least important features might be removed to improve model's performance.

Residuals of KNN

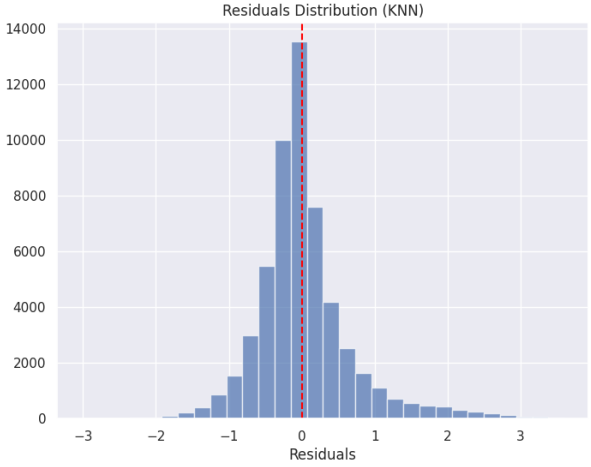
```
residuals = y_val - knn_val_preds
```

```
KNN_prediction_analysis = pd.DataFrame({
    'Actual': y_val,
    'Predicted': knn_val_preds,
    'Residual': residuals
}).reset_index(drop=True)

poor_predictions = KNN_prediction_analysis.assign(AbsResidual=np.abs(KNN_prediction_analysis['Residual'])) \
                                          .sort_values(by='AbsResidual', ascending=False) \
                                          .head(5)
print("Poor Predictions:\n", poor_predictions)
```

```
Poor Predictions:
         Actual  Predicted  Residual  AbsResidual
52420  2.377263  -1.237230  3.614494     3.614494
14334  2.377263  -1.152108  3.529371     3.529371
44897  2.377263  -1.149268  3.526531     3.526531
13893  2.377263  -1.143348  3.520611     3.520611
10203  2.377263  -1.116549  3.493812     3.493812
```

Residual represent the difference between actual values and predicted values. A histogram is used to represent the residual distribution of KNN Regression model.



A bell-shaped curve histogram shows that residuals are centered around 0. This indicates that model's predictions are generally accurate, with some small errors. The symmetric histogram suggesting no over-predictions. This residual distribution supports the validity of the KNN regression model.

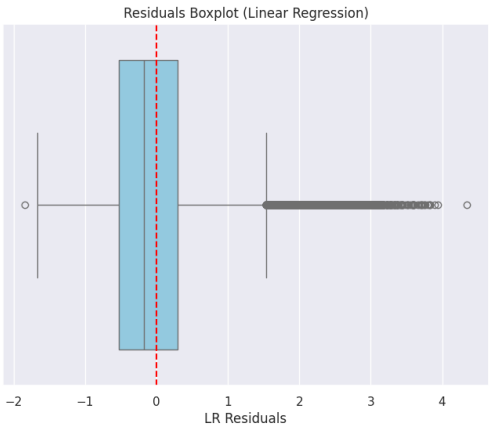
Residuals of Linear Regression

```
residuals_lr = y_val - lr_val_preds
```

```
lr_prediction_analysis = pd.DataFrame({'Actual': y_val,'Predicted': lr_val_preds, 'Residual': residuals_lr}).reset_index(drop=True)

poor_predictions = lr_prediction_analysis.assign(AbsResidual=np.abs(lr_prediction_analysis['Residual'])) \
                                         .sort_values(by='AbsResidual', ascending=False) \
                                         .head(5)
print("Poor Predictions:\n", poor_predictions)
```

```
Poor Predictions:
         Actual  Predicted  Residual  AbsResidual
29498  1.527784  -7.645312  9.173096     9.173096
16281  2.377263  -6.586878  8.964142     8.964142
22093  1.206568  -6.006141  7.212709     7.212709
29853  2.377263  -3.649118  6.026382     6.026382
16376  1.750457  -4.205099  5.955556     5.955556
```



The boxplot is indicating the residual of Linear Regression model. A variable is used to store the residual and their absolute residuals. A symmetric residual boxplot shows that residuals are properly distributed, indicating no over-predictions. Points beyond the whiskers shows examples where the model's predictions significantly differ from the actual values. This plot helps to find issues in the regression model, such as whether the residuals meet the assumptions.

Residuals of Decision Tree Regressor

```
residuals_dt = y_val - dt_val_preds
```

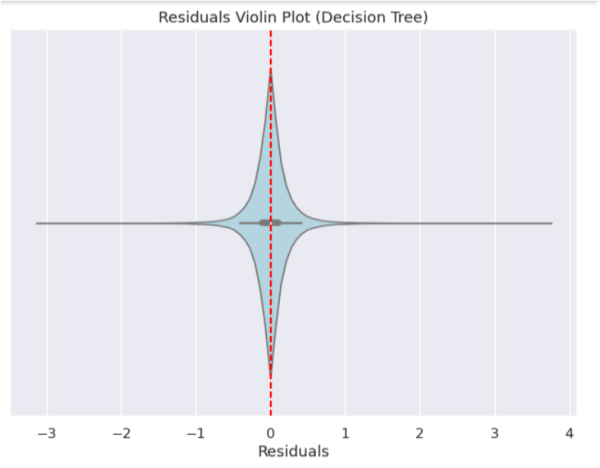
```
if isinstance(residuals_dt, pd.DataFrame):
    residuals_dt = residuals_dt.values.flatten()

residuals_dt_df = pd.DataFrame({
    'Actual': y_val,
    'Predicted': dt_val_preds,
    'Residual': residuals_dt
}).reset_index(drop=True)

poor_predictions = residuals_dt_df.assign(AbsResidual=np.abs(residuals_dt_df['Residual'])) \
                                  .sort_values(by='AbsResidual', ascending=False) \
                                  .head(5)

print("Poor Predictions:\n", poor_predictions)
```

```
Poor Predictions:
         Actual  Predicted  Residual  AbsResidual
33334  2.377263  -1.319481  3.696745     3.696745
8860   2.377263  -1.247829  3.625093     3.625093
1707   2.377263  -1.180181  3.557444     3.557444
34985  2.377263  -1.129324  3.506588     3.506588
15777  2.377263  -0.762608  3.139871     3.139871
```



Above plot is called Violin Plot and the code snippet for analyzing residuals. A violin plot combines two plots i.e., a boxplot and a density plot. It showing both distribution shape and summary statistics. In the above plot, the red line shows the ideal case where predictions perfectly match actual values. A balanced plot around the center indicates that the model's prediction errors are evenly distributed.