Mohammad Amirian - MMU ID: 24821023

1. Data / Domain Understanding and Exploration

1.1. Meaning and Type of Features; Analysis of Distributions

The purpose of this section is to conduct a preliminary analysis of the data set to understand what each feature means and what type it is, and look at their distributions. This initial analysis helps in identifying some data quality issues and informs data preparation to ensure effective modeling.

This dataset has near 402,000 rows and entries about car sale adverts from AutoTrade. It contains some details and features which related to different cars such as mileage, fuel type, body type, color, price and etc.

At first we load the dataset and by using info function (data.info()), we display the details and information and type of each features.

```
<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 1.1.1

As you can see this dataset has 12 features which we can divide to 2 type of features, Categorical and Numerical features. Categorical contains features that is of type bool and object, and Numerical contains features that is of type int and float. You can see other details such as the number of non-null data or the number of data types which this dataset has, and memory usage that shows 34.1 MB.

There is a significant right-skewedness in the mileage distribution. The majority of the cars in this dataset have minimal mileage, and as mileage rises, their frequency significantly declines. Because the dataset has enormous entries, we modify the range in plot by using bins.

```
sns.histplot(data['mileage'], kde=True, bins=50, color='green')
```

Recent years, after 2000, have seen a significant increase in registrations, which are distinguished by a clear peak in the present era. There are some questionable or inaccurate data points on the x-axis, such as the years 1000 and 1800, which indicate errors in the dataset.

The most prevalent automakers in the dataset are displayed in the bar chart. BMW is the most common, although it is clear that Audi, Volkswagen, and BMW rule the dataset. Mercedes-Benz, Toyota, and Vauxhall are in the middle of the chart. Despite being among the top 10, Renault has a lower count compared to others.

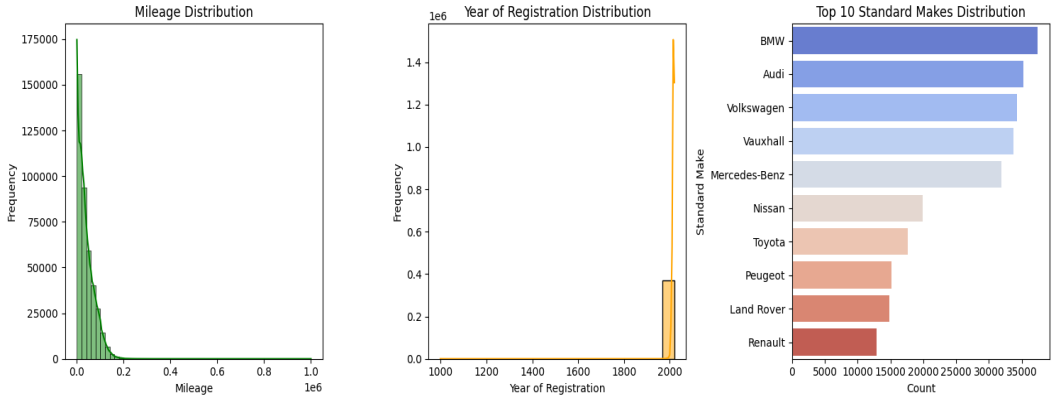

Figure 1.1.2

As I said, we have some inaccurate data in year of registration which make our distribution complicated and we cannot get any information from its plot. To solve this problem, we filter the dataset and we will make it only show the data after 2000 in year of registration and the data before 200000 in mileage.

I fixed the problem in Figure 1.1.2 with the following code and you can see the modified plots in Figure 1.1.3.

```
data = data[(data['year_of_registration'] >= 2000)]
data = data[data['mileage'] <= 200000]
```

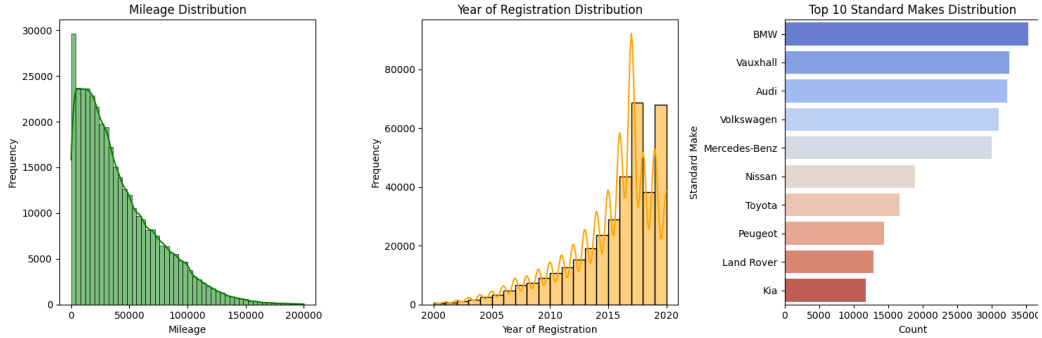

Figure 1.1.3

Now we can see a left-skewedness in year of registration plot and there is also a peak in registrations around the year 2015 and mileage distribution is now much more understandable.

1.2. Analysis of Predictive Power of Features

For this section I have used 2 method of analyzing, the Pearson correlation for numerical features and ANOVA for categorical features. We calculate the Pearson correlation between Price, Mileage and Year of registration.

```
correlations = data[['price', 'mileage',
'year_of_registration']].corr()
print("Correlation with Price:\n", correlations['price'])
```

```
Correlation with Price:
 price                 1.000000
mileage               -0.273294
year_of_registration   0.257982
Name: price, dtype: float64
```

Figure 1.2.1

Mileage has a fairly strong negative association with price which is indicated by the coefficient of -0.273294. It means that as the mileage of a car increases, its price decreases. By looking at the coefficient between year of registration and price, which is 0.257982 we can conclude that newer cars have higher price than an older version. I illustrate their relationships with heatmap to visualize it and for better understanding which you can see in figure 1.2.2.

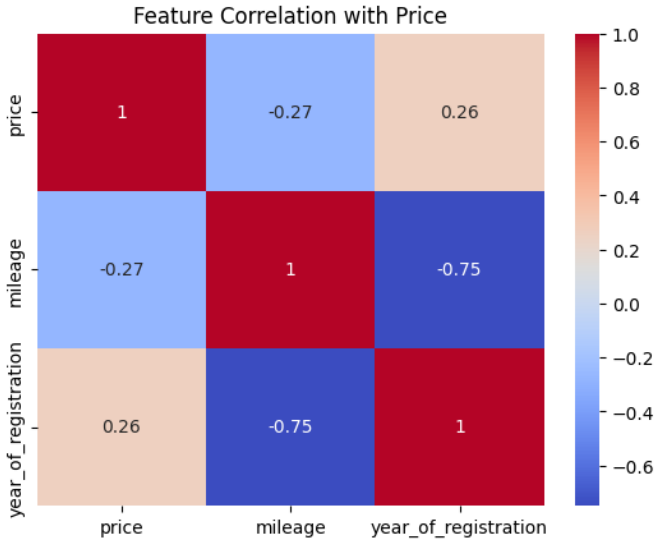

Figure 1.2.2

The feature standard-make had a significant F-value of 2825.1387820933032 and a P-value of 0.0 from the ANOVA results. This test suggests that the mean prices for the different automobile brands differ significantly. The extremely low P-value strongly suggests that these differences are statistically significant, demonstrating once more how strongly a car's brand predicts its price. (figure 1.2.3)

```
categories = data['standard_make'].unique()
grouped_data = {cat: data['price'][data['standard_make'] == cat]
for cat in categories}
f_val, p_val = stats.f_oneway(*grouped_data.values())
print(f"ANOVA result for 'standard_make': F-value = {f_val}, P-
value = {p_val}")
```

```
ANOVA result for 'standard_make': F-value = 2825.1387820933032, P-value = 0.0
```

Figure 1.2.3

The analysis shows that both mileage and year of registration have predictive powers with regard to automobile prices, but mileage is a stronger predictor based on its higher absolute correlation coefficient. The second notable outcome of the ANOVA analysis on standard-make is its high impact on pricing differences between different brands, hence proving that this variable should be definitely included in any model for the prediction of automobile prices.

1.3. Data Processing for Data Exploration and Visualisation

In this section, I compared 3 main features with price. These features are actually the features which we concluded in section 1.2. They are powerful to detecting the target which is price.

The scatter plot Mileage Vs Price explains a visible trend in the decrease of prices with increasing mileage. This negative correlation is very clear and in agreement with the common view that cars lose value when high mileage results from wear and tear. Moreover, the significant grouping of data points at lower mileage and higher price levels suggests that newer or less-used cars tend to have higher market values.

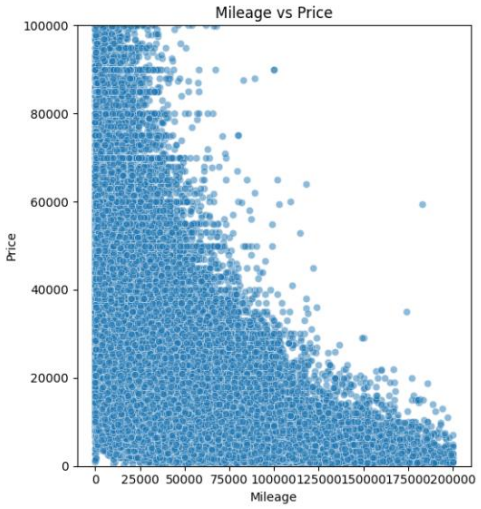

Figure 1.3.1

For comparing Year of registration and Price, I calculated the average price per year.

```
yearly_avg_price = data.groupby('year_of_registration')['price'].mean().reset_index()
```

After a period of general price stability in the first few years, prices started rising in 2010. As you can see we can conclude that the newer the car, the more expensive it is.

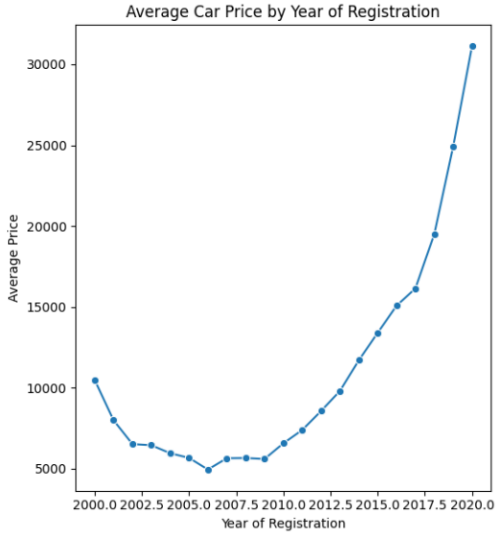

Figure 1.3.2

The box plot Price distribution by Standard make describes the way car prices are distributed among different brands. The higher the brand in terms of price range, such as Mercedes-Benz, Audi, and BMW, the greater the possibility that their product could be very expensive or even relatively cheap. Otherwise, brands like Kia and Peugeot have a lower price range, which leads to the fact that prices are pretty consistent within those brands. This picture helps explain which brands are in the premium area and which ones are for the low-budget customers, which affects the general price distribution. (figure 1.3.3)

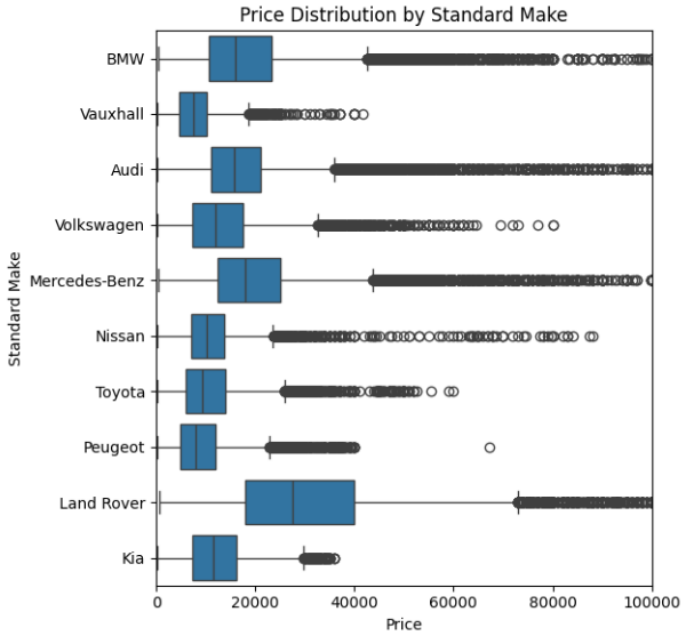

Figure 1.3.3

2. Data Processing for Machine Learning

2.1. Dealing with Missing Values, Outliers, and Noise

At first, we drop two features from our dataset which do not help us in target prediction. These two features are public reference and crossover car and van which are unnecessary.

```
data.drop(['public_reference', 'crossover_car_and_van'], axis=1, inplace=True)
```

The dataset was grouped into categorical variables and numerical variables. It created the opportunity of applying different preprocessing techniques for each of them.

Categorical Features: Handling missing values in categorical features was done with SimpleImputer more frequent value being imputed. Such strategy is helpful in maintaining the structure of categorical data.

Year of Registration: An additional custom function, extract_year_from_code, was introduced to predict year_of_registration from reg_code column. Thus, missing values were replaced by attributes, which were approximated during the calculation process, and the created interim column was deleted.

```
def extract_year_from_code(reg_code):
    if pd.isna(reg_code):
        return None
    try:
        # extract the age identifier from the registration code
        age_identifier = int(reg_code[:2])
        # determine the year based on the age identifier
        if age_identifier >= 50:
            year = 2000 + (age_identifier - 50)
        else:
            year = 2000 + age_identifier
    except ValueError:
        return None
    return year
# apply the function to the reg_code column to create a new year column
data['calculated_year'] = data['reg_code'].apply(extract_year_from_code)

# fill in missing values in year_of_registration with the values from calculated_year
data['year_of_registration'].fillna(data['calculated_year'], inplace=True)

# drop the temporary calculated_year column
data.drop('calculated_year', axis=1, inplace=True)
```

Numerical Features: Numerical features which contained missing values in their features, filled with the mean of each features using SimpleImputer.

For categorical features, features that make up less than 1% of the data were clustered under what was termed ‘Other’. It removes noise and improves the interpretability of categorical features on this step.

To handle outliers in a set of numerical features, the Interquartile Range (IQR) technique was used. For each feature, outliers were identified as values outside the range of [Q1 - 1.5*IQR, Q3 + 1.5*IQR]. These outliers have been removed from dataset to maintain more accuracy of the results to be obtained.

```python
# removing outliers using IQR for numerical features
for feature in numerical_features:
    Q1 = data[feature].quantile(0.25)
    Q3 = data[feature].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    data = data[(data[feature] >= lower_bound) & (data[feature] <= upper_bound)]
```

At the end, I removed the target (Price) from the list of numerical features to prevent unintended preprocessing that could impact model predictions. To check and confirm that we fill all the missing values, we use this function:

```python
data.isnull().sum()
```

which you can see the result in figure 2.1.1 that illustrates we do not have any missing values.

```
mileage                0
reg_code               0
standard_colour        0
standard_make          0
standard_model         0
vehicle_condition      0
year_of_registration   0
price                  0
body_type              0
fuel_type              0
dtype: int64
```

Figure 2.1.1

2.2. Feature Engineering, Data Transformations, Feature Selection

To decrease the skewness and make the variance in the numerical features more stable the Power Transformer was used. This conversion assists in making the amount of data sets more standard for utilization in the models that deal with learning.

```python
# applying Power Transformation to reduce skewness and handle noise
pt = PowerTransformer()
data[numerical_features] = pt.fit_transform(data[numerical_features])
```

The categorical features were processed using two encoding techniques to ensure they are represented numerically:

- Target Encoding: Used on feature, for which the difference between the feature values and the target variable (price) made sense. The following features were encoded using this method: all categories excluding fuel_type and vehicle_condition.
- One-Hot Encoding: Used as the formula to convert fuel_type and vehicle_condition, because these features both have the least number of unique data in comparison with other categorical features. This encoding produces binary columns on the categories so that the model can interpret them well.

```python
# define which columns should be encoded using which method
target_encoded_cols = [col for col in categorical_features if col not in ['fuel_type', 'vehicle_condition']]
one_hot_encoded_cols = ['fuel_type', 'vehicle_condition']
```

After that, I created the preprocessing pipeline to combine both encoding strategies. This structure allows for efficient and organized transformation of categorical features, ensuring consistent processing during model training and evaluation.

```python
# create the preprocessing pipeline
preprocessor = ColumnTransformer(
    transformers=[
        ('target', TargetEncoder(), target_encoded_cols),
        ('one_hot', OneHotEncoder(sparse_output=False), one_hot_encoded_cols)
    ])
```

At the end, the dataset was split into training and testing data which 80 percent was used for training while the rest 20 percent was used for testing. In this model our target (Price) is y and X is defined as the combination of numerical and categorical features.

```
# split the data
y = data[target]
X = data[categorical_features + numerical_features]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

3. Model Building

I combined these two sections (3.1. Algorithm Selection, Model Instantiation and Configuration and 3.2. Grid Search, and Model Ranking and Selection) together, and in this part I want to explain each model one by one.

3.1. KNN

To make these steps more manageable, a Pipeline was developed to contain all the preprocessing and modeling steps. It consists of:

- Preprocessor: Performed transformations on categorical predictor variables, which includes target encoding, one-hot encoding.
- Scaler: Rescales the numerical features to the same domain using the MinMaxScaler to make all the value equal.
- KNN Regressor: implements the K-Nearest Neighbors regression model.

```
# create a full pipeline with KNN
pipeline_knn = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('scaler', MinMaxScaler()),
    ('KNN', KNeighborsRegressor())
])
```

A grid search was performed to identify the best combination of hyperparameters:

- n_neighbors: Defines the amount of neighbors to involve. Tested values: [15, 25, 35, 50]. I have tried big numbers for neighbors such as 100, 200, 300, and etc. But I did not get a good result and the training took too much time to process.
- weights: Describes how the influence of neighbors is weighted (uniform or distance)

```
# gridSearchCV for KNN parameter tuning within the pipeline
param_grid_knn = {
    'KNN__n_neighbors': [15, 25, 35, 50],
    'KNN__weights': ['uniform', 'distance']
}
```

after that, we train the model with GridSearchCV using 5-fold cross-validation to ensure robust hyperparameter evaluation. The final model of the forward selection of hyperparameters is chosen based on the highest mean of the test R^2.

The model was then evaluated on both the training and test sets:

- R^2 Score: shows the proportion of variance explained by the model.
- Mean Squared Error (MSE): An average of the square of the differences between actual and predicted observations.

```
# ensure the correct scorer is used
r2_scorer = make_scorer(r2_score)
grid_search_knn = GridSearchCV(pipeline_knn, param_grid_knn, cv=5, scoring=r2_scorer, return_train_score=True, n_jobs=-1)
grid_search_knn.fit(X_train, y_train)

# best parameters
print("Best Parameters:", grid_search_knn.best_params_)

# evaluate the model
best_model_knn = grid_search_knn.best_estimator_
y_pred_train_knn = best_model_knn.predict(X_train)
y_pred_test_knn = best_model_knn.predict(X_test)

print("R2 Score on Train Set:", r2_score(y_train, y_pred_train_knn))
print("MSE on Train Set:", mean_squared_error(y_train, y_pred_train_knn))
print("R2 Score on Test Set:", r2_score(y_test, y_pred_test_knn))
print("MSE on Test Set:", mean_squared_error(y_test, y_pred_test_knn))
```

Looking at figure 3.1.1, we can conclude that the best hyperparameters are 35 for the number of neighbors and distance for weights. These hyperparameters achieved the highest mean R^2 score on the test set, illustrating strong predictive performance. The R^2 score on training set is 0.8788 and on testing set is 0.8388 which shows our model works well in predicting the target with these parameters. And then we can see the results of MSE which is

6649521.26 on training and 8842503.15 on testing the data. The amount of MSE, shows that we have minimal error on training data. On test data, MSE increased slightly but our model still works well. At the end, the table provides some information about all the different outputs of the possible model parameters. The table was sorted and ranked by the mean of R^2 test score, the higher is the better.

```
Best Parameters: {'KNN__n_neighbors': 35, 'KNN__weights': 'distance'}
R2 Score on Train Set: 0.8788123468207789
MSE on Train Set: 6649521.260886619
R2 Score on Test Set: 0.8388395164581763
MSE on Test Set: 8842503.157493507
```

	rank_test_score	params	mean_r2_test_score	mean_r2_train_score	std_r2_test_score	std_r2_train_score
5	1	{'KNN__n_neighbors': 35, 'KNN__weights': 'dist...	0.834809	0.881809	0.002245	0.000322
7	2	{'KNN__n_neighbors': 50, 'KNN__weights': 'dist...	0.834731	0.882017	0.002208	0.000324
3	3	{'KNN__n_neighbors': 25, 'KNN__weights': 'dist...	0.834425	0.881302	0.002244	0.000285
1	4	{'KNN__n_neighbors': 15, 'KNN__weights': 'dist...	0.832952	0.879653	0.002030	0.000294
0	5	{'KNN__n_neighbors': 15, 'KNN__weights': 'unif...	0.826640	0.844456	0.000766	0.000318
2	6	{'KNN__n_neighbors': 25, 'KNN__weights': 'unif...	0.821778	0.834647	0.001296	0.000257
4	7	{'KNN__n_neighbors': 35, 'KNN__weights': 'unif...	0.817144	0.827180	0.001244	0.000335
6	8	{'KNN__n_neighbors': 50, 'KNN__weights': 'unif...	0.811503	0.818974	0.001375	0.000519

Figure 3.1.1

Figure 3.1.2 shows the comparison of actual and predicted price on KNN model.

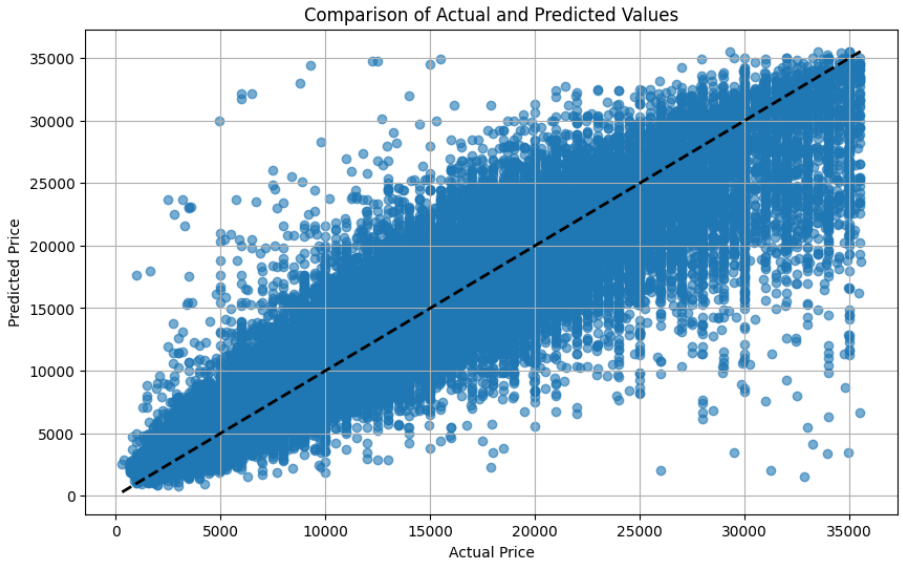

Figure 3.1.2

3.2. Linear Regression

For this model, I have used the same approach like the KNN model and in this section I will describe the differences. In pipeline the only difference is the regressor which implements the Linear Regression model.

```
pipeline_lg = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('scaler', MinMaxScaler()),
    ('regressor', LinearRegression())
])
```

In this model the only hyperparameter is fit_intercept which has two value; True or False. True means that the model specified requires inclusion of an intercept term to the model for purposes of fitting the data well.

```
param_grid_lg = {
    'regressor__fit_intercept': [True, False]
}
```

The process of evaluation, again is the same like the KNN model.

```
r2_scorer = make_scorer(r2_score)
grid_search_lg = GridSearchCV(pipeline_lg, param_grid_lg, cv=5, scoring=r2_scorer, n_jobs=-1, return_train_score=True)
grid_search_lg.fit(X_train, y_train)

print("Best Parameters:", grid_search_lg.best_params_)

best_model_lg = grid_search_lg.best_estimator_
y_pred_train_lg = best_model_lg.predict(X_train)
y_pred_test_lg = best_model_lg.predict(X_test)

print("R2 Score on Train Set:", r2_score(y_train, y_pred_train_lg))
print("MSE on Train Set:", mean_squared_error(y_train, y_pred_train_lg))
print("R2 Score on Test Set:", r2_score(y_test, y_pred_test_lg))
print("MSE on Test Set:", mean_squared_error(y_test, y_pred_test_lg))
```

The results at figure 3.2.1 indicates that there is no big difference between two values of fit_intercept. However, choosing fit_intercept=False is slightly more efficient because it makes the model simpler. As you can see the R^2 score on test and train set is 71%, and MSE on train set is 15514887.19 and on the test set is 15445037.77.

```
Best Parameters: {'regressor__fit_intercept': False}
R2 Score on Train Set: 0.7172408817632235
MSE on Train Set: 15514887.194361322
R2 Score on Test Set: 0.7185039448460848
MSE on Test Set: 15445037.77735621
```

	rank_test_score	mean_r2_test_score	mean_r2_train_score	param_regressor__fit_intercept
1	1	0.717109	0.717240	False
0	2	0.717109	0.717239	True

Figure 3.2.1

Figure 3.2.2 shows the comparison of actual and predicted price on Linear Regression model.

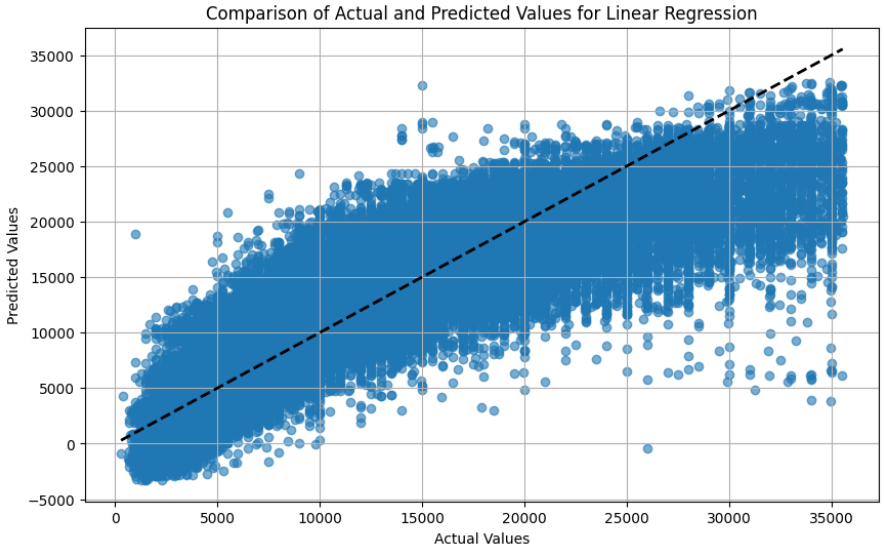

Figure 3.2.2

3.3. Decision Tree

The approach that has used in this model is the same like previous models, so again I just explain the differences. In pipeline the only difference is the regressor which implements the Decision Tree Regressor model.

```
pipeline_dt = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('scaler', MinMaxScaler()),
    ('regressor', DecisionTreeRegressor(random_state=0))
])
```

The hyperparameters in this model are:

- max_depth: Defines the maximum depth which the decision tree can reach ([10, 20, 30]). Reducing the depth in a tree helps in minimizing overfitting.
- min_samples_split: The minimum samples required to split a node is defined by values of [2, 10, 20].
- min_samples_leaf: It represent the minimum number of samples that must be in a leaf node ([1, 5, 10]).

```
param_grid_dt = {
    'regressor__max_depth': [10, 20, 30],
    'regressor__min_samples_split': [2, 10, 20],
    'regressor__min_samples_leaf': [1, 5, 10]
}
```

The process of evaluation, again is the same like the previous models.

```
r2_scorer = make_scorer(r2_score)
grid_search_dt = GridSearchCV(pipeline_dt, param_grid_dt, cv=5, scoring=r2_scorer, n_jobs=-1, return_train_score=True)
grid_search_dt.fit(X_train, y_train)

print("Best Parameters:", grid_search_dt.best_params_)

best_model_dt = grid_search_dt.best_estimator_
y_pred_train_dt = best_model_dt.predict(X_train)
y_pred_test_dt = best_model_dt.predict(X_test)
print("R2 Score on Train Set:", r2_score(y_train, y_pred_train_dt))
print("MSE on Train Set:", mean_squared_error(y_train, y_pred_train_dt))
print("R2 Score on Test Set:", r2_score(y_test, y_pred_test_dt))
print("MSE on Test Set:", mean_squared_error(y_test, y_pred_test_dt))
```

Figure 3.3.1 provides information about best hyperparameters for this model which are:

- max_depth: 20
- min_samples_leaf: 5
- min_samples_split: 20

with these parameters we can reach the highest mean R^2 score on the test set in this model. As you can see the R^2 score on train set is 86% and on the test set is 84% which shows our model works good, and MSE on train set is 7285078.36 and on the test set is 8384219.78. The table illustrates different possibilities performances with their rank and parameters. It also shows, the similarity among different parameters performance, so it is better to choose the hyperparameters with less complexity.

In all training model, the Overfitting has been controlled effectively.

```
Best Parameters: {'regressor__max_depth': 20, 'regressor__min_samples_leaf': 5, 'regressor__min_samples_split': 20}
R2 Score on Train Set: 0.8672293064087859
MSE on Train Set: 7285078.361504378
R2 Score on Test Set: 0.8471920348962114
MSE on Test Set: 8384219.780339323
```

	rank_test_score	mean_r2_test_score	mean_r2_train_score	param_regressor__max_depth	param_regressor__min_samples_split	param_regressor__min_samples_leaf
14	1	0.844958	0.867437	20	20	5
23	2	0.844866	0.867556	30	20	5
11	3	0.844558	0.870648	20	20	1
20	4	0.844386	0.870854	30	20	1
17	5	0.844213	0.862809	20	20	10
16	5	0.844213	0.862809	20	10	10
15	5	0.844213	0.862809	20	2	10
26	8	0.844167	0.862863	30	20	10
25	8	0.844167	0.862863	30	10	10
24	8	0.844167	0.862863	30	2	10
12	11	0.844071	0.869736	20	2	5
13	11	0.844071	0.869736	20	10	5
21	13	0.843922	0.869901	30	2	5
22	13	0.843922	0.869901	30	10	5
10	15	0.842051	0.875763	20	10	1
19	16	0.841640	0.876150	30	10	1
9	17	0.835864	0.881455	20	2	1

Figure 3.3.1

Figure 3.3.2 shows the comparison of actual and predicted price on Decision Tree model.

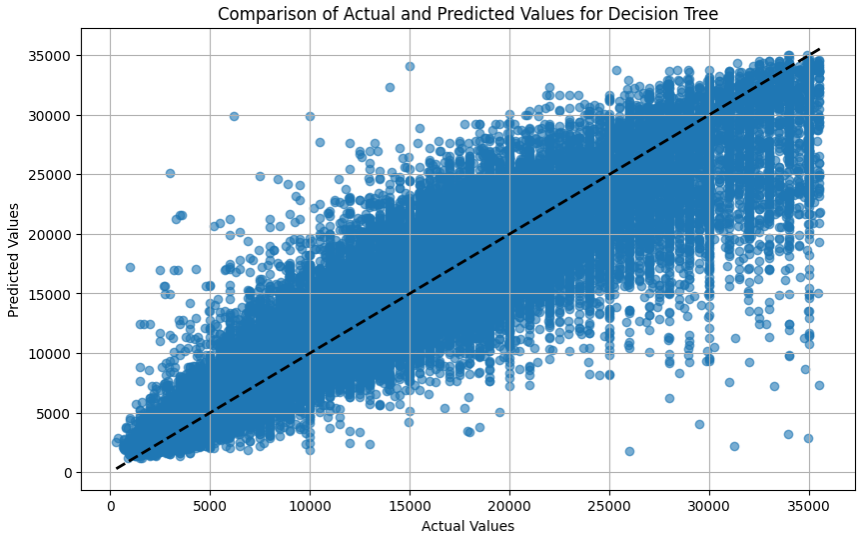

Figure 3.3.2

4. Model Evaluation and Analysis

4.1. Coarse-Grained Evaluation/Analysis

This code is developed to visualize the performance of the three models on the bases of R^2 scores. It is marked which of the models works best according to the criterion of determination R^2. The use of the plot allows you to compare the models easily. First, I separate the R^2 scores of each model by different variables, and then I use functions to visualize the results.

```
r2_knn = r2_score(y_test, y_pred_test_knn)
r2_lg  = r2_score(y_test, y_pred_test_lg)
r2_dt  = r2_score(y_test, y_pred_test_dt)

# dataframe to compare
df = pd.DataFrame({
    'Model': ['KNN', 'Linear Regression', 'Decision Tree'],
    'R2 Score': [r2_knn, r2_lg, r2_dt]
})

# barplot
plt.figure(figsize=(7, 5))
sns.barplot(data=df, x='Model', y='R2 Score', palette='viridis')

plt.title('Comparison of Best R2 Score Across Models')
plt.xlabel('Model')
plt.ylabel('R2 Score')
plt.grid(axis='y')
plt.ylim(0, 1)

plt.show()
```

Figure 4.1.1 illustrates that the Decision Tree model has the best R^2 score on test set. After that, the KNN model comes in second place by a very small margin. Finally, the Linear Regression model has the smallest value of R^2 score of all three models. This means that the process is non-linear and results in low performance of features with regard to the target variables. Although we can see that the R^2 score of the KNN model and the Decision Tree are almost equal, the Decision Tree is more ideal model due to the time complexity of running the model and training the data. In addition, the Decision Tree model has the lowest MSE value on test set among these tree models. In conclusion, the Decision Tree is the most ideal model for this dataset.

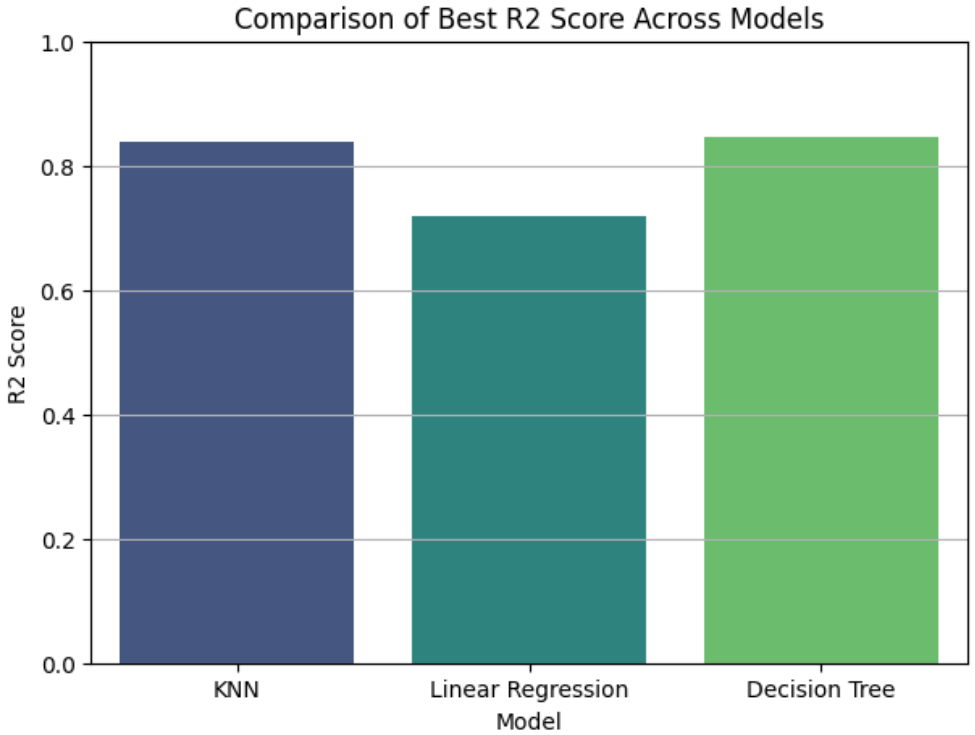

Figure 4.1.1

4.2. Feature Importance

For this section, I used permutation_importance to measure the feature importance. permutation_importance shows the significance of the feature by permuting it and comparing the results of the model afterwards. When shuffling a feature greatly decrease the accuracy of the model it is important.

For each model, the bar chart is produced after using a plot(kind='barh') to depict their 4 most significant features. There are features associated with each chart that helps to understand which of them has a high influence on the model's predictions.

For KNN model we have:

```
from sklearn.inspection import permutation_importance

# KNN feature importance
result_knn = permutation_importance(best_model_knn, X_test, y_test, n_repeats=10, random_state=42)
perm_importances_knn = pd.Series(result_knn.importances_mean, index=X_train.columns)
perm_importances_knn.nlargest(4).plot(kind='barh')
plt.title('KNN Permutation Feature Importance')
plt.show()
```

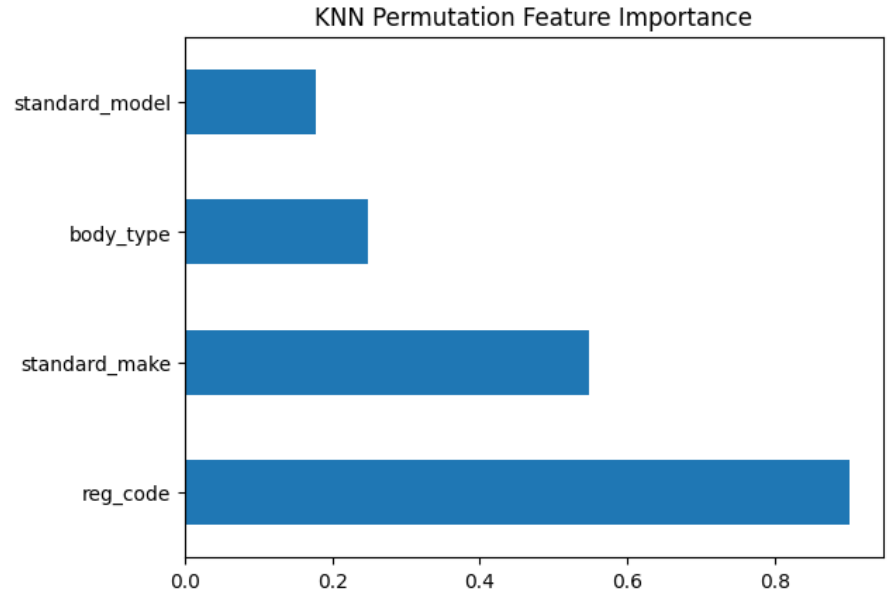

Figure 4.2.1

For Linear Regression model we have:

```
# LinearRegression feature importance
result_lr = permutation_importance(best_model_lg, X_test, y_test, n_repeats=10, random_state=42)
perm_importances_lr = pd.Series(result_lr.importances_mean, index=X_train.columns)
perm_importances_lr.nlargest(4).plot(kind='barh')
plt.title('Linear Regression Permutation Feature Importance')
plt.show()
```

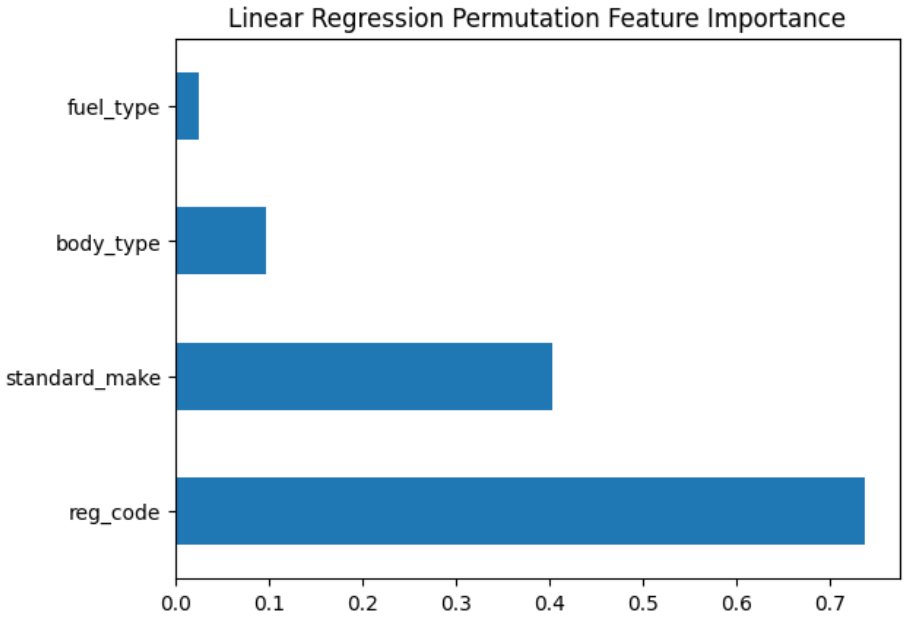

Figure 4.2.2

For Decision Tree model we have:

```
# DecisionTreeRegressor feature importance
result_dt= permutation_importance(best_model_dt, X_test, y_test, n_repeats=10, random_state=42)
perm_importances_dt = pd.Series(result_dt.importances_mean, index=X_train.columns)
perm_importances_dt.nlargest(4).plot(kind='barh')
plt.title('Decision Tree Permutation Feature Importance')
plt.show()
```

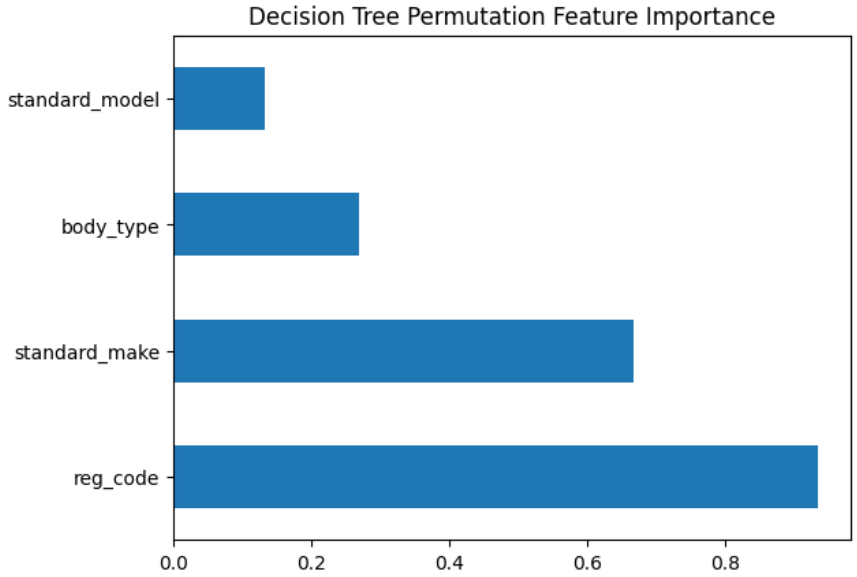

Figure 4.2.3

Looking to figures 4.2.1, 4.2.2 and 4.2.3, we can conclude that in these three models reg-code and standard-make features have the most influence on the prediction of target (Price). Body-type and standard-model placed third and fourth respectively on Decision Tree and KNN models. On Linear Regression we can see that the only difference is on the fourth place, which is fuel-type. The reason that reg-code has the most effect on these three models is that it shows and represents the year of registration in different way.

4.3. Fine-Grained Evaluation

in this section, the code which is developed shows a comparison of three machine learning models namely KNN, Linear Regression and a Decision Tree via their absolute errors. It evaluates instance-level performance differences using two types of visualizations:

- Line Plot: Illustrates individual sample errors.
- Box Plot: provides information about the error distribution.

For the purpose of simulation, the y_test which represents the true target values are considered as random numbers in the range of 0 to 1000. New variables (y_pred_test_knn, y_pred_test_lg, and y_pred_test_dt) are obtained by adding random noise to y_test to introduce predictions by KNN, Linear Regression, and Decision Tree models.

```
# assuming the predicted test values and actual test values are as follows:
# for simulation purposes, using random data here, replace it with actual predictions and y_test values.

np.random.seed(0)
y_test = np.random.rand(100) * 1000  # simulated true values
y_pred_test_knn = y_test + np.random.normal(0, 50, 100)  # simulated KNN predictions
y_pred_test_lg = y_test + np.random.normal(0, 70, 100)  # simulated LinearRegression predictions
y_pred_test_dt = y_test + np.random.normal(0, 30, 100)  # simulated DecisionTree predictions

# calculating instance-level errors
errors_knn = np.abs(y_test - y_pred_test_knn)  # absolute errors for KNN
errors_lr = np.abs(y_test - y_pred_test_lg)  # absolute errors for LinearRegression
errors_dt = np.abs(y_test - y_pred_test_dt)  # absolute errors for DecisionTree

# create a range of indices for x-axis
indices = np.arange(len(y_test))
```

To show the absolute errors for each sample, a line plot is constructed. Separate markers and color coding for each model are chosen to increase readability. Finally, box plots are used to compare the distribution of the absolute errors of the three models. The boxes are also colored for clearer differentiation between them.

```
# plotting line plots for error
plt.figure(figsize=(14, 7))
plt.plot(indices, errors_knn, label='KNN Errors', marker='o', linestyle='-', markersize=5)
plt.plot(indices, errors_lr, label='Linear Regression Errors', marker='x', linestyle='-', markersize=5)
plt.plot(indices, errors_dt, label='Decision Tree Errors', marker='^', linestyle='-', markersize=5)

plt.title('Line Plot of Absolute Errors Across Models')
plt.xlabel('Sample Index')
plt.ylabel('Absolute Error')
plt.legend()
plt.grid(True)
plt.show()


# plotting box plots for error distribution
plt.figure(figsize=(10, 6))
box = plt.boxplot([errors_knn, errors_lr, errors_dt], labels=['KNN Errors', 'Linear Regression Errors', 'Decision Tree Errors'], patch_artist=True)

colors = ['blue', 'yellow', 'green']
for patch, color in zip(box['boxes'], colors):
    patch.set_facecolor(color)

plt.title('Comparison of Absolute Errors Across Models')
plt.ylabel('Absolute Error')
plt.grid(True)
plt.show()
```

Figures 4.3.1 and 4.3.2 give us this information that among these models, the Decision Tree model has the lowest errors and the errors in the model are almost stable. Since Linear Regression has the highest Variability and most of the outliers appear in this model, this model has least accuracy. According to average error rates, the Decision Tree model appears to outcompete its rivals as the most accurate predictions approach is, followed by KNN and then the Linear Regression approach.

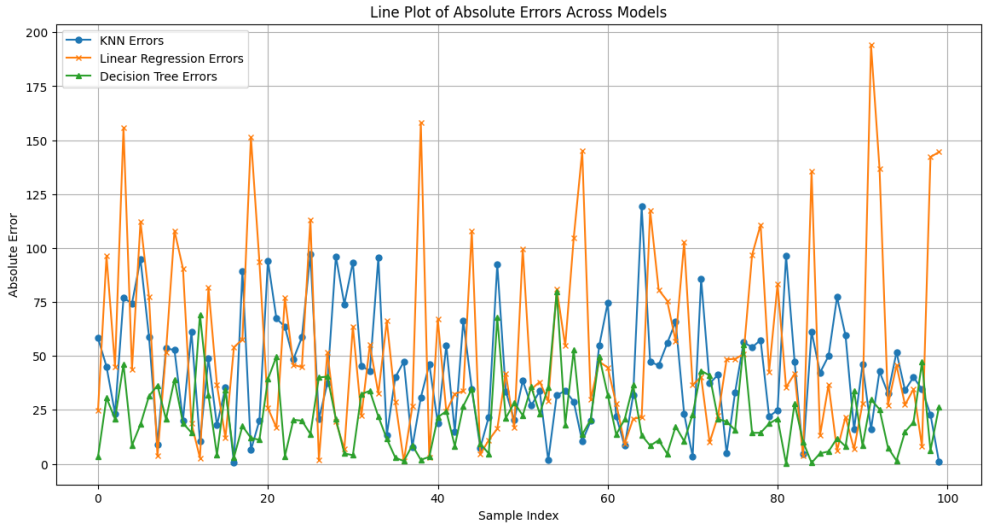

Figure 4.3.1

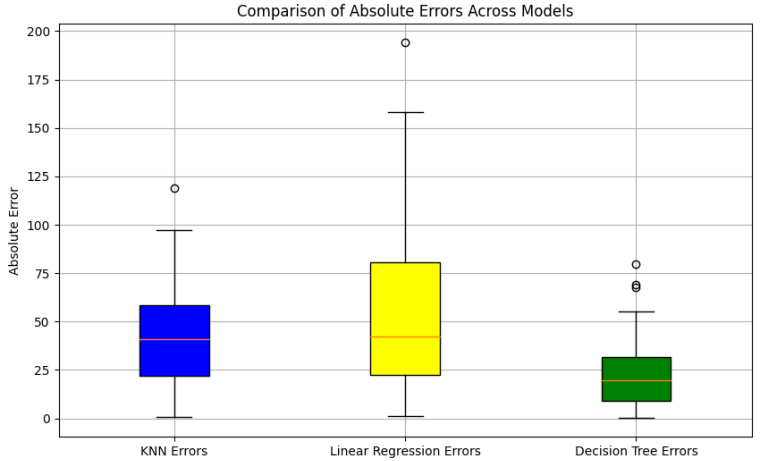

Figure 4.3.2