Car Price Prediction using Regression Machine Learning Models

By

Chris Shi, Erika Grandy, and Sumedha

A project submitted in partial fulfilment of the

requirements for the course

EECS 3401

Introduction to Artificial Intelligence
and Logic Programming

York University

Professor: Dr. Ruba Alomari

November 2023

Student Numbers:

Chris Shi: 218869305

Erika Grandy: 217300948

Sumedha: 218561985

Acknowledgement

We are immensely grateful to Dr. Ruba Alomari, for providing us with the opportunity to work on this project and apply the knowledge we acquired from our coursework in EECS 3401, to a practical implementation in the form of a Machine Learning model. Furthermore, her continued support and guidance carried us through all stages of making our project and made it possible for us to bring our ideas to life.

We'd also like to thank each other, for everyone's continuous participation through every stage of this project, for being an amazing team, valuing everyone's ideas, and supporting each other through every hurdle faced during the project.

Lastly, we'd like to extend a huge thank-you to each of our families for their continued love and support. We would not have been able to complete this work without your encouragement.

Chris Shi, Erika Grandy, and Sumedha

Table of Contents

Looking at the Big Picture

Framing the Problem:

1. Supervised Learning: Since, the training dataset is labelled, we will be using supervised learning models.
2. Regression: We will predict, the price of a used car using its features using a model.
3. Batch Learning: Batch Learning will be used to train the dataset because there is no continuous flow of data coming into the system, and there is no need to adjust data rapidly.

Predicting the price of a used car can be quite challenging. From a customer's point of view, it may be confusing to determine what budget they should set based on their needs. It can be difficult to gauge the price of newer cars if they have not bought a car in a few years, and it might be even harder to gauge the current price of older cars. Although difficult to determine by oneself, we know there are features that affect the price of vehicles, such as larger vehicles being more expensive, or specific manufacturers (Toyota, BMW, Lexus, Mercedes) being more expensive, etc.

Developing a Machine Learning Model to predict the price of a car will help us in minimizing this issue. A model that can predict car prices would know how each feature is weighed within the decision-making process, and this would allow us to determine not only the price, but also which features of a car are more valued by customers.

Looking at a bigger picture, such a model predicting used car prices can be used by dealerships looking to determine which cars would be worth the investment, based on estimated costs through which they can maximise profits. Similarly for people wishing to sell their used cars, the model would help them in accurately pricing their car based on its features thus increasing their chances of selling it. On the other hand, this model would help buyers best estimate the price that they would have to pay for a car based on the features they value most. Hence, the model would assist both parties in a transaction, minimizing many misconceptions and frictions that might have occurred otherwise.

Description of Data Set

This data set contains the data for used car sales. It includes standard information about the vehicle, such as Manufacturer, Model, Production Year, and Mileage. It also includes additional information about the engine volume, fuel type, number of cylinders, transmission type, drive wheels, number of doors, color, number of airbags, and levy. Based on the dataset description, the levy of the vehicle is defined as the tax required if the vehicle were to be imported or exported.

The dataset initially has the following columns as strings/objects: Levy, Manufacturer, Model, Category, Fuel type, Engine volume, Mileage, Gearbox type, Drive wheels, Doors, Wheel, and Color. The dataset initially has the following columns as numerical: ID, Price, Prod. year, Cylinders, and Airbags.

General EDA Conclusions

Exploratory Data Analysis provided the following conclusions about the dataset. Most vehicles are Front-wheel drive. Additionally, most vehicles are sedans, followed by SUVs and then hatchbacks. Most used cars fall around the years 2012-2015. The distribution of production year is a normal distribution around this peak point. Most vehicles are 4 cylinders, followed by 6 cylinders, then 8. Very few vehicles are 3 or 5-cylinder. Most vehicles have an engine displacement of 2.0L, followed by 2.5L, 1.8L, 1.6L. Other common ones are 3.0L, 3.5L, and 1.5L. The average price of a used car is around $18,000.

1. **Correlation Matrix - Heatmap**

In viewing the heatmap, important correlation weights can be found between the features, and between the features and target. It can be seen that the production year and price have a correlation of 0.4, indicating that the production year is the strongest feature. This also indicates that increasing production years will have a positive correlation with price. It can also be seen that the next highest magnitude correlation is between mileage and price, at -0.21. This indicates that the price will be less as the mileage increases.

Between features, the highest correlation is between cylinders and engine volume at 0.78. This is reasonable because more cylinders allow for greater displacement. There is also a correlation of 0.23 between the production year and airbags and 0.23 between the production year and engine volume. This indicates that newer cars have more airbags and are increasing in engine volume.

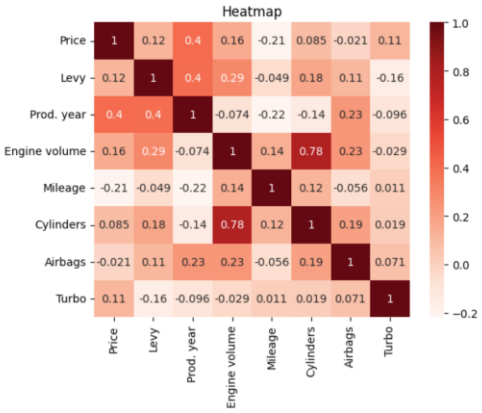

Figure 2 Heatmap

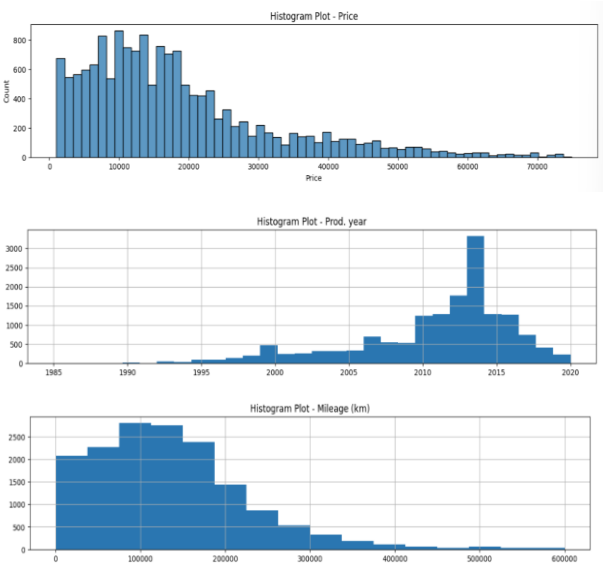

Figure 3 Histograms

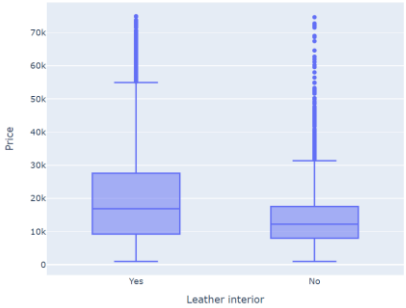

Figure 1 Boxplot - Leather Interior

1. **Histograms - Price, Production Year, Mileage**

Viewing histograms provide information about how common a specific value is within numerical data. Price, production year, and mileage are all distributed normally. The most common price of used cars falls within the $9000-$15,000 range. However, there are quite a few options at lower prices. In terms of production year, the most common production year for used cars for sale is between 2013-2015. In terms of mileage, the most common mileage is around 100,000 km.

2. **Boxplot - Leather Interior**

Boxplots allow for gaining insight into median values as well as quartiles. In Figure 3, the boxplot for leather interior vs price indicates that the median price for a car with leather interior is greater than that of a car without. The median price for a car with a leather interior is $16.9k, compared to $12.23k without. An important note is that a leather interior may indicate additional features in a higher-end car, also increasing the price.

Data Cleaning and Preprocessing

Data Cleaning

In terms of data cleaning, the first consideration was to remove any outliers that may skew the model.

Price Outliers - This dataset included around 2000 cars that were listed under $1000. These are considered as outliers, whether as a typo or as an extreme circumstance, such as a car that does not run, missing engine/transmission, extremely damaged, etc. Additionally, outliers on the higher end were considered using the IQR metric. These higher-end outliers were removed so as to not consider special circumstances about why a car may be extremely expensive, such as collection / show cars. This resulted in 56 car samples being removed.

Year Outliers - This dataset includes cars produced from the years 1939 to 2020. Older vehicles are difficult to predict price on, as older cars are often cheaper unless they reach the age where they become antiques. As such, older cars could be extremely cheap, or extremely expensive. This is difficult to gauge, as a specific model could be expensive if it was featured within media, such as a Dodge Charger as featured in The Dukes of Hazzard. However, another Dodge from the same year could be cheap. Additionally, the dataset does not provide information as to whether these older cars are original or restored, again making it difficult to gauge. As such, any car produced prior to 1985 was removed.

Data cleaning also included handling missing values. This only included the Levy column, as otherwise there were no missing values.

Levy - Missing Values - In the case of this dataset, about 30% of columns had '-' for the Levy value. This was considered to mean there was no listed value. This was replaced with 0, and Levy was converted into a numerical column.

Data cleaning was also performed to clean up some values, to either fix their meaning or to use North American terms.

Doors - The doors column contained 3 different values: '02-Mar', '04-May', and '>5'. These were considered to need cleaning, as this dataset was likely saved within a spreadsheet that assumed the range to be a date, converting 2-3 to '02-Mar'. These values were replaced with the correct corresponding values: '2-3', '4-5', '>5'.

Mileage - The mileage column was of type 'object', as it included 'km' at the end. This was removed and the column was converted to integer values. Additionally, any columns with mileage over 600,000 km were removed as outliers.

Wheel - The wheel column had inconsistent wording between the two options: 'Left wheel', and 'Right-hand drive'. The 'Left wheel' option was replaced to be consistent with the 'Right-hand drive' option.

Category - The category column refers to the type of vehicle, such as sedan, coupe, pickup truck, etc. However, this dataset uses British-English terminology. A few categories were renamed to improve readability when presenting the results. The 'Jeep' option was replaced with 'SUV', as Jeep is a Manufacturer here, and is contradictory to say another manufacturer produces a Jeep-type vehicle. The 'Cabriolet' option was replaced with 'Convertable' to improve understanding.

Gearbox - The gearbox column refers to the transmission type. One of the options is 'Variator'. This option was replaced with 'CVT' to align with North American terminology.

Engine Volume - The engine volume column refers to the displacement of the engine. However, this column lists the displacement along with if the vehicle has a turbo, such as '2.3 Turbo'. It was concluded that keeping the information about the Turbo is essential, as it results in higher horsepower and would affect the price. To account for this, an additional binary 'Turbo' column was created, where 1 indicates it has a turbo. The engine volume was then converted into a float numerical value.

The final step in data cleaning was to remove any duplicate values. It was found that there were 256 duplicate rows.

Preprocessing

For preprocessing, two pipelines were created, one each for numerical and categorical data. The categorical data is passed through a pipeline that will replace any missing values with the most frequent and then passed through an Ordinary Encoder to convert the values to numerical.

The numerical data is passed through a pipeline that replaces any missing values with the mean and then scales the values. Only the features are passed through this preprocessing (X) to produce the prepared data (X_prepared). The target column is not passed through any preprocessing.

Training and Evaluation of ML Models

Performance Metrics

The following metrics were used to compare the performance of regression models:

MAE: Mean Absolute Error is calculated by determining the absolute error between test and predicted values, and then taking the mean of these from all test samples. MAE is a useful metric as its units match the unit of the target, which is dollars. Additionally, it is robust to outliers.

R-Squared: R2, unlike the previous metric, describes how well the model performs rather than the error. R2 is also known as the Goodness of Fit. An R2 of 0 indicates the model has the worst possible performance. An R2 of 1 indicates that the model is perfect, which is unrealistic. An R2 score can be described as "the model is capable of explaining x percent of variance."

Model 1 - Polynomial

Polynomial Regression models the relationship between input features and output values, similarly to Linear Regression, but using a polynomial equation to a specified degree. This model is particularly effective with non-linear data.

To fit this model to the dataset, the *degree* hyperparameter was first to be tuned. Higher degrees risk overfitting of the model to the training dataset, while lower degrees may result in underfitting. The tested degrees were from 1 to 4 (inclusive), with the highest performing degree found to be 3. Using this degree, the minimum MAE was $5895, with a training score of 71.3%, and testing score of approximately 55.8%.

The result of a much higher training score compared to the testing score is likely due to overfitting of the data. Despite this, the degree of 3 outperformed a degree of 2 and 4, landing 3 as the optimal value.

Model 2 - Gradient Boosting

Gradient Boosting Regression is an ensemble method in which various weaker models are joined together to result in an overall good model. This regression model begins with a weak guess of a model and examines the errors of the model when compared to expected values. The model then makes iterations where it modifies the estimator line, attempting to reduce error in each section of the data. The model does so by minimizing the loss function, using the gradient of the function.

In terms of hyperparameters, GridSearch was used to find the best parameters. n_estimators describe the number of sequential decisions to be made and was found to be 3, so as to not overfit the data. learning_rate describes the magnitude of changes at each iteration. 0.3 was found to be ideal, as it's large enough to be quick, but small enough to not cause bouncing.

This model resulted in a training r-squared score of 0.873 and a test r-squared score of 0.751. The MAE was $4342.

Model 3 - Random Forest

The random forest model uses multiple decision trees, creating multiple randomly generated trees and averaging the results to create a strong model. GridSearch was used to find the best parameters for this model. n_estimators describe the number of decision trees, and the best value

was found to be 300. Min_samples_split_ describes the minimum number of samples required to split nodes and was found to be best at 3. Additionally, max_depth was set to none.

This model resulted in a training r-squared score of 0.959 and a test r-squared score of 0.779. The MAE was found to be $3611.

Comparing Results between the 3 Best Performing Algorithms based on Performance Metrics

The comparison of the results of the 3 Best Performing Algorithms can be seen in Table 1. In comparing r-squared value, the best performing algorithms in order are Random Forest, followed by Gradient Boosting and then Polynomial Regression. The same ranking holds when comparing MAE of the models.

Model	MAE	R-SQUARED
Polynomial	5929	0.488
Random Forest	3611	0.779
Gradient Boosting	4342	0.751

Table 1. Comparison of Performance Metrics among the 3 Best Performing Algorithms

Best Performing Algorithm

Comparing the results using r-squared (Figure 4) and MAE, the best-performing algorithm was Random Forest. It resulted in an r-squared of 0.779, and on average the prediction is only $3600 off.

The performance of different models varies depending on characteristics of the dataset. One advantage Random Forest has over Polynomial, specifically, is its consideration to the interaction between features. Often, features affect the final output in a combined fashion, which is not represented well in a Polynomial model.

Random Forest is also less sensitive to noise in the data compared to Polynomial due to its averaging of predictions by multiple decision trees (Sellahewa, 2023).

In this way, Random Forest and Gradient Boosting are similar by incorporating decision trees. The difference lies in their creation and grouping (Lok, 2022). Parameters in Gradient Boosting, such as learning_rate and n_estimators can be more involved than with Random Forest, which performs well even with default parameters.

Graphs for the Best Performing algorithm (Random Forest)

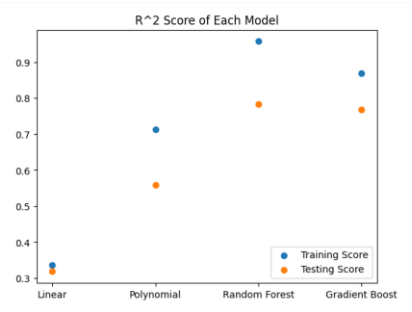

Figure 4. R^2 Score of Each Model

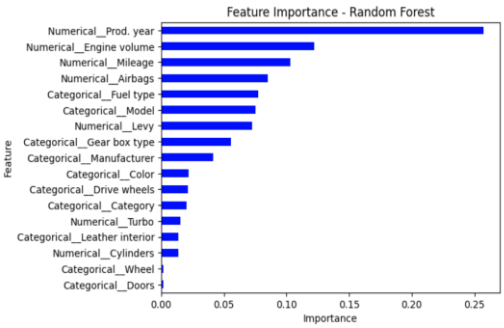

Figure 5. Feature Importance from Best Performing Algorithm

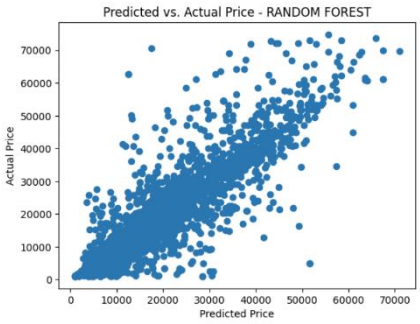

Figure 6. Performance of Best Performing Algorithm

Following the results described in terms of feature weight (See Figure 5), the production year had the greatest impact, followed by the engine volume. This is likely due to consumers buying bigger cars. Additionally, mileage was the third most important factor. Airbags were another prime factor, and although a consumer may not know the number of airbags, this correlates to safer models and bigger vehicles.

Additionally, Figure 4 demonstrates that Random Forest had the highest testing score when compared to other models, However, this Figure also demonstrates that Random Forest was overfitting the dataset, given the difference in training score and testing score. Regardless, Random Forest had the best testing score. The performance of Random Forest is demonstrated in Figure 6, where the models price prediction is plotted against the actual price. With a perfect model, this graph would form a linear line, with slope of 1.

Limitations

Some of the limitations we encountered during this project are as follows:

1. In the dataset, a lot of features that are otherwise considered very important in determining the price of a used car are found missing. For example, information like condition of car (any rust), if AC is working, power windows, keyless entry, remote start, other comfort features (heated seats, steering wheel, etc), is missing which could have drastic impact on the price of car being considered.
2. A huge number of outliers were initially found in the dataset. For instance, it included instances of some cars that despite being old and without many favourable features, were valued at high prices because of their ‘antique’, and ‘collectable’ nature, while on the other hand, some cars were priced exceptionally low despite having many redeemable features, hinting to them being in an un-usable state. Thus extensive, and careful data cleaning had to be performed to obtain a dataset still that had numerous instances without including an incredibly large number of features.
3. The use of OneHotEncoding appeared as an initial limitation, as it made the models incredibly slow. This led to a lot of time committed just to end up with a model with low performance. This was solved by switching to OrdinaryEncoding, as it refrained from increasing the number of features by a huge amount, which was achieved by converting different classification to categories to numerical values, unlike OneHotEncoding which created enormous amounts of new columns.
4. The small number of features in the data showed itself to be a huge limitation in the later stages, especially because many features proved to be unimpactful to the target, resulting in the relatively low performance metric values for many models.

Conclusion

In conclusion, the regression models that performed best on this dataset were Polynomial, Gradient Boosting, and Random Forest, with Random Forest being the best overall model, with a R-squared of 0.779, and a MAE of $3600. The most important features of the car impacting the price were found to be production year, followed by engine volume, then mileage, and airbags, by the Random Forest model. Though many important features that usually affect the price of a car drastically were found missing from the dataset, the best performing model can still predict the price of the used car quite accurately based on provided features which will allow our model to aid both sellers and buyers of used cars in estimating better prices for them, and carrying out better, and more informed sales, especially in cases where information about the car might be limited.

Appendix 1

Source Code

Note: The following is only the Python code and does not include all written steps and citations that are within the notebook. Please see the GitHub link in Appendix 2 for the full code including titles and comments.

```
# importing libraries
import sklearn
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px

# Load the dataset
url = "https://raw.githubusercontent.com/sumedha-git/EECS3401-
Car_Price_AI_Model/main/car_price_prediction.csv"
cars = pd.read_csv(url, sep=",")

# Backup dataset
data_backup = cars
#printing the dataset
cars
```

```
#prnting the first 5 rows(instances) of the dataset
cars.head()
```

```
#printing the describing info about all numerical columns in our dataset
cars.describe()
```

```
#printing the shape (number of rows and columns) of our dataset
cars.shape
```

```
#printing some defining info about the values in each column of our
dataset
cars.info()
```

```
# Check for Missing values that are input as '?' in the dataset
cars.isin(['?']).sum()
```

```
# Check for misiing values that are input as NaN in the dataset
cars.isin([np.NaN]).sum()
```

```
#printing the defining info about the numerical column 'Price' in our
dataset
cars.Price.describe()
```

```
# SOURCE https://saturncloud.io/blog/how-to-detect-and-exclude-outliers-in-a-pandas-dataframe/

# calculate IQR for column Height
Q1 = cars['Price'].quantile(0.05)
Q3 = cars['Price'].quantile(0.95)
IQR = Q3 - Q1

# identify outliers in Price column
threshold = 1.5
outliers = cars[(cars['Price'] < Q1 - threshold * IQR) | (cars['Price'] > Q3 + threshold * IQR)]
outliers.shape[0]

# Drop outliers in Price column
cars = cars.drop(outliers.index)
cars.shape

#printing the number of cars with a Price under 1000, and then dropping them
print("Cars under $1000: {}".format(cars[cars.Price < 1000].shape[0]))
cars.drop(cars[cars.Price < 1000].index, inplace=True)

#printing the number of cars with a Price over 75000, and then dropping them
print("Cars under $1000: {}".format(cars[cars.Price > 75000].shape[0]))
cars.drop(cars[cars.Price > 75000].index, inplace=True)

#printing the new shape(number of rows and columns) of the dataset
print("Shape after dropping: {}".format(cars.shape))

#plotting a histogram for column Prod. year
sns.histplot(cars["Prod. year"])
plt.show()

#dropping all cars with a prod year under 1985, due to the potential anomalous prices for cars considered 'antique' or 'really old'
cars.drop(cars[cars["Prod. year"] < 1985].index, inplace=True)

#plotting a new histogram for column Prod year after dropping some values
sns.histplot(cars["Prod. year"])
plt.show()

#printing the number of duplicates in the dataset
print("Number of duplicate rows: {}".format(cars.duplicated().sum()))

#dropping all duplicates
cars.drop_duplicates(inplace=True)
```

```
#printing the defining info of the numerical column levy
cars["Levy"].describe()

# Replace '-' in Levy with $0, as '-' are the missing values in Levy
cars['Levy'] = cars['Levy'].replace('-', 0)
cars['Levy'].isin(['-']).sum()

# Cast Levy with to an int
cars['Levy'] = cars['Levy'].astype(int)

#printing all values of Mileage column in the dataset
cars["Mileage"]

#dropping the 'km' suffix from all mileage values
cars["Mileage"] = cars["Mileage"].str.split(' ').str[0]

#converting mileage values to int
cars['Mileage'] = cars['Mileage'].astype(int)

#droppinh all instances of cars with mileage value more than 600,000
cars.drop(cars[cars.Mileage > 600000].index, inplace=True)

#printing major info about all columns in th
cars.info()

#printing the number of times different classes appear in the column
'Category'
cars['Category'].value_counts()

#Replacing the name of all instances called 'Jeep' with 'SUV', and
'Cabriolet' with 'Convertable'
cars['Category'] = cars['Category'].replace('Jeep',
'SUV').replace('Cabriolet','Convertable')

#printing the number of times different classes appear in the column
'Category'
cars['Category'].value_counts()

#printing the number of times different classes appear in the column
'Doors'
cars['Doors'].value_counts()

#Replacing the name of all instances called '04-May' with '4-5', and '02-
Mar' with '2-3'
cars['Doors'] = cars['Doors'].replace('04-May', '4-5').replace('02-Mar',
'2-3')

#printing the number of times different classes appear in the column
'Doors'
```

```
cars['Doors'].value_counts()

#Replacing the name of all instances called 'Variator' with 'CVT'
cars['Gear box type'] = cars['Gear box type'].replace('Variator', 'CVT')

#printing the number of times different classes appear in the column
'Gear box type'
cars['Gear box type'].value_counts()

#printing the number of times different classes appear in the column
'Wheel'
cars['Wheel'].value_counts()

#Replacing the name of all instances called 'Left wheel' with 'Left-hand
drive'
cars['Wheel'] = cars['Wheel'].replace('Left wheel', 'Left-hand drive')

#printing the number of times different classes appear in the column
'Wheel'
cars['Wheel'].value_counts()

#printing the number of times different classes appear in the column
'Engine volume'
cars['Engine volume'].value_counts()

#creating a new column Turbo and then for every instance in column Engine
volume, if it contains the word 'Turbo', set value in column Turbo as 1,
and 0 otherwise
#https://stackoverflow.com/questions/66622656/np-where-for-string-
containing-specific-word
cars["Turbo"] = np.where(cars["Engine volume"].str.contains('Turbo'), 1,
0)

#for every value in column Engine volume, if it contains the word Turbo,
replace the word 'Turbo' with ' '.
cars["Engine volume"] = cars["Engine volume"].str.replace("Turbo", "")

#convert the values of column Engine volume to type float
cars["Engine volume"] = cars["Engine volume"].astype(float)

#print the most defining info of the now numerical column Engine volume
in our dataset
cars["Engine volume"].describe()

#drop all instances from dataset where the value of Engine volume column
is more than 8
cars.drop(cars[cars["Engine volume"] > 8].index, inplace=True)

#drop the feature ID from the dataset
```

```
cars.drop(labels=['ID'], axis=1, inplace=True)

#print the first 5 instances of the dataset
cars.head()
```

```
#plotting a histogram of Price column
sns.histplot(cars.Price)
plt.title("Histogram Plot - Price")
plt.show()
```

```
#printing the defining info about the numerical column 'Price' in our
dataset
cars.Price.describe()
```

```
#plotting a histogram for column Cylinders in the dataset
print(cars['Cylinders'].hist(figsize=(7,7), range=[0, 10], align='mid'))
plt.title("Number of Cylinders")
plt.xlabel("Cylinders")
```

```
#plotting a histogram for column Prod. year in the dataset
cars['Prod. year'].hist(figsize=(15,7), align='mid', bins=30)
plt.title("Histogram Plot - Prod. year")
plt.xlabel("Year")
```

```
#plotting a histogram for column Airbags in the dataset
cars['Airbags'].hist(figsize=(15,7), align='mid', bins=16)
plt.title("Airbags")
plt.xlabel("Airbags")
plt.ylabel("Count")
```

```
#plotting a histogram for column Mileage in the dataset
cars['Mileage'].hist(figsize=(15,3), align='mid', bins=16)
plt.title("Histogram Plot - Mileage (km)")
plt.xlabel("Kilometers")
```

```
#https://www.geeksforgeeks.org/seaborn-barplot-method-in-python/

#storing the number of times different classes appear in column
'Category', to variable count
count=cars['Category'].value_counts()

#plotting a barplot using the count variable, that stores the number of
times evry class appears in the column 'Category'
plt.figure().set_figwidth(15)
sns.barplot(x=count.index, y=count.values)
plt.title('Feature - Category')
plt.ylabel('Count')
plt.xlabel('Category')
```

```
#storing the number of times different classes appear in column 'Drive
wheels', to variable count
count=cars['Drive wheels'].value_counts()

#plotting a barplot using the count variable, that stores the number of
times every class appears in the column 'Drive Wheels'
plt.figure().set_figwidth(15)
sns.barplot(x=count.index, y=count.values)
plt.title('Feature - Drive Wheels')
plt.ylabel('Count')
plt.xlabel('Drive Wheels')
```

```
#storing the number of times different classes appear in column 'Engine
volume', to variable count
count=cars['Engine volume'].value_counts()

#plotting a barplot using the count variable, that stores the number of
times every class appears in the column 'Engine volume'
plt.figure().set_figwidth(20)
sns.barplot(x=count.index, y=count.values)
plt.title('Feature - Engine volume')
plt.ylabel('Count')
plt.xlabel('Engine Volume', fontsize=2)
```

```
#storing the number of times different classes appear in column 'Turbo',
to variable count
count=cars['Turbo'].value_counts()

#plotting a barplot using the count variable, that stores the number of
times every class appears in the column 'Turbo'
plt.figure().set_figwidth(15)
sns.barplot(x=count.index, y=count.values)
plt.title('Feature - Turbo')
plt.ylabel('Count')
plt.xlabel('Turbo')
```

```
#storing the number of times different classes appear in column
'Manufacturer', to variable count
count=cars['Manufacturer'].value_counts()

#plotting a barplot using the count variable, that stores the number of
times every class appears in the column 'Manufacturer'
plt.figure().set_figwidth(20)
sns.barplot(x=count.index, y=count.values)
plt.title('Feature - Manfacturer')
plt.ylabel('Count')
plt.xticks(rotation=70)
plt.xlabel('Manufacturer', fontsize=2)
```

```
#storing the number of times different classes appear in column 'Model',
to variable count
count=cars['Model'].value_counts()

#dropping all classes where the count value is less than 20, in variable
count
count.drop(count[count < 100].index, inplace=True)

#plotting a barplot using the count variable, that stores the number of
times every class appears in the column 'Model'
plt.figure().set_figwidth(20)
sns.barplot(x=count.index, y=count.values)
plt.title('Feature - Model')
plt.ylabel('Count')
plt.xticks(rotation=70)
plt.xlabel('Model')
```

```
# Source: https://plotly.com/python/treemaps/

#importing library
import plotly.express as px

#plotting a tree map using columns Manufacturer and Model from the
dataset
fig = px.treemap(data_frame=cars, path=["Manufacturer","Model"])
fig.show()
```

```
#storing the number of times different classes appear in column 'Gear box
type', to variable count
count=cars['Gear box type'].value_counts()

#plotting a barplot using the count variable, that stores the number of
times every class appears in the column 'Gear box type'
plt.figure().set_figwidth(20)
sns.barplot(x=count.index, y=count.values)
plt.title('Feature - Gearbox Type')
plt.ylabel('Count')
plt.xlabel('Gearbox')
```

```
#storing the number of times different classes appear in column 'Doors',
to variable count
count=cars['Doors'].value_counts()

#plotting a barplot using the count variable, that stores the number of
times every class appears in the column 'Doors'
plt.figure().set_figwidth(20)
sns.barplot(x=count.index, y=count.values)
plt.title('Feature - Doors')
plt.ylabel('Count')
```

```
plt.xticks(rotation=70)
plt.xlabel('Doors')

#storing the number of times different classes appear in column 'Color',
to variable count
count=cars['Color'].value_counts()

plt.figure().set_figwidth(20)
sns.barplot(x=count.index, y=count.values)
plt.title('Feature - Color')
plt.ylabel('Count')
plt.xticks(rotation=70)
plt.xlabel('Color')

# Create the correlation matrix for numerical features and Price (target)
corr_matrix = cars.corr(numeric_only = True)
corr_matrix['Price'].sort_values(ascending=False)

# Use Seaborn to create a heatmap, to view correlation between features
sns.heatmap(cars.corr(numeric_only=True), annot=True, cmap='Reds')
plt.title("Heatmap")

# Create plot to view relationshio between prod year and price
sns.lmplot(x='Prod. year',y='Price', data=cars, fit_reg=False)
plt.show()

# Create plot to view relationship between Mileage and Price
sns.lmplot(x='Mileage',y='Price', data=cars, fit_reg=False)
plt.show()

#Create boc plots for Categorical Features: Manufacturer, Category,
Leather interior
bp = px.box(data_frame=cars,x='Manufacturer',y='Price')
bp.show()

bp = px.box(data_frame=cars,x='Category',y='Price')
bp.show()

bp = px.box(data_frame=cars,x='Leather interior',y='Price')
bp.show()

# Prepare for training, Split into X (features) and y (target)
X = cars.drop(['Price'], axis=1)
y = cars['Price']
print(X.shape, y.shape)

#Imports for pipeline
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import make_pipeline
```

```
from sklearn.preprocessing import OrdinalEncoder
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

# Code based on snippet fom Class Sides - EECS3401 with Dr. Ruba Al Omari
# Modifications made to pipelines, encoder used.

# Split into numerical columns and categorical columns
numerical_columns = X.select_dtypes(include='number').columns.to_list()
categorical_columns = X.select_dtypes(exclude='number').columns.to_list()

# Numerical pipeline - Use SimpleInputer to replace missing w/ mean, then
scale.
numerical_pipeline = make_pipeline(SimpleImputer(strategy='mean'),
StandardScaler())

# Categorical pipeline - Use SimpleImputer to replace missing w/ most
frequent,
# then OneHot encode to turn into numerical values.
categorical_pipeline =
make_pipeline(SimpleImputer(strategy='most_frequent'), OrdinalEncoder(),
StandardScaler())

preprocess = ColumnTransformer([
                                ('Numerical', numerical_pipeline,
numerical_columns),
                                ('Categorical', categorical_pipeline,
categorical_columns)],
                               )
preprocess
```

```
# Code snippet fom Class Sides - EECS3401 with Dr. Ruba Al Omari

X_prepared = preprocess.fit_transform(X) #Prepare X by feeding through
preprocess

# Feeding through preprocess converts dataframe into numpy array +
removes headers.
# The below code adds headers back and then casts back to a pandas
DataFrame
feature_names=preprocess.get_feature_names_out()
X_prepared = pd.DataFrame(data=X_prepared, columns=feature_names)

X_prepared.shape # View shape after preprocess.
```

```
# Code snippet fom Class Sides - EECS3401 with Dr. Ruba Al Omari

from sklearn.model_selection import train_test_split
```

```
# Split dataset into 80% training and 20% testing, then view shape
X_train, X_test, y_train, y_test = train_test_split(X_prepared, y,
test_size=0.2)
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)

# List to store results in
results = []

# import statements for models
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import mean_squared_error, mean_absolute_error,
r2_score
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
import matplotlib.pyplot as plt
from math import sqrt

# Fit model
lin_reg = LinearRegression().fit(X_train, y_train.values.ravel())
print("Training Score: ", lin_reg.score(X_train, y_train))
print("Testing score: ", lin_reg.score(X_test, y_test))

# Test model
y_predict = lin_reg.predict(X_test)

# https://scikit-learn.org/stable/modules/model_evaluation.html

# Compare Prediction to Test Targets
decimal_places = 3
mae = round(mean_absolute_error(y_test, y_predict), decimal_places)
mse = round(mean_squared_error(y_test, y_predict), decimal_places)
rmse = round(sqrt(mse), decimal_places)
r2 = round(r2_score(y_test, y_predict), decimal_places)

print("Model 1: LINEAR REGRESSION")
print("Mean Absolute Error: ", mae)
print("Mean Squared Error: ", mse)
print("Rooted Mean Squared Error: ", rmse)
print("R-squared Value: ", r2)

# Add results to list for future comparison
results.append(["Linear", lin_reg.score(X_train,y_train), v, rmse, mse,
mae])

# https://inria.github.io/scikit-learn-
mooc/python_scripts/dev_features_importance.html
```

```
# Plot the feature weights as determined by linear regression

coefs = pd.DataFrame(
    lin_reg.coef_, columns=["Coefficients"], index=X_train.columns
)
coefs.plot(kind="barh")
```

```
# find best degree
poly_params = np.arange(1, 5)
rmse_list = []
min_rmse, min_deg = 1e10, 0

# Fit model for each degree
for degree in poly_params:
  # train features
  poly_train = PolynomialFeatures(degree=degree).fit_transform(X_train)
  # regression
  poly_reg = LinearRegression().fit(poly_train, y_train)
  # test
  poly_test = PolynomialFeatures(degree=degree).fit_transform(X_test)
  y_predict = poly_reg.predict(poly_test)
  poly_mse = mean_squared_error(y_test, y_predict)
  poly_rmse = np.sqrt(poly_mse)
  rmse_list.append(poly_rmse)
  # comparison
  if min_rmse > poly_rmse:
    min_rmse = poly_rmse
    min_deg = degree

print("Best degree: ", min_deg, ", minimum error: ", min_rmse)
```

```
print("Training score: ", poly_reg.score(poly_train, y_train))
print("Testing score: ", poly_reg.score(poly_test, y_test))
```

```
# Compare Prediction to Test Targets
decimal_places = 3
mae = round(mean_absolute_error(y_test, y_predict), decimal_places)
mse = round(mean_squared_error(y_test, y_predict), decimal_places)
rmse = round(sqrt(mse), decimal_places)
r2 = round(r2_score(y_test, y_predict), decimal_places)

print("Model 2: POLYNOMIAL REGRESSION")
print("Mean Absolute Error: ", mae)
print("Mean Squared Error: ", mse)
print("Rooted Mean Squared Error: ", rmse)
print("R-squared Value: ", r2)

# Add results to list for future comparison
```

```
results.append(["Polynomial", poly_reg.score(poly_train,y_train), r2,
rmse, mse, mae])

# TRAIN MODEL USING BEST HYPERPARAMETERS
forest = RandomForestRegressor(n_estimators=200,
                        max_depth=None,
                        min_samples_split=3)

forest.fit(X_train, y_train.values.ravel())

# Compare training and test score
print("Training score: ", forest.score(X_train, y_train))
print("Testing score: ", forest.score(X_test, y_test))

# Test the model
y_predict = forest.predict(X_test)

# Compare Prediction to Test Targets
decimal_places = 3
mae = round(mean_absolute_error(y_test, y_predict), decimal_places)
mse = round(mean_squared_error(y_test, y_predict), decimal_places)
rmse = round(sqrt(mse), decimal_places)
r2 = round(r2_score(y_test, y_predict), decimal_places)

print("Model 3: RANDOM FOREST")
print("Mean Absolute Error: ", mae)
print("Mean Squared Error: ", mse)
print("Rooted Mean Squared Error: ", rmse)
print("R-squared Value: ", r2)

# Add results to list for future comparison
results.append(["Random Forest", forest.score(X_train,y_train), r2, rmse,
mse, mae])

# https://mljar.com/blog/feature-importance-in-random-forest/

# Calculate weights based on what Random Forest determined feature
importance to be
weights = pd.Series(forest.feature_importances_, index=X_train.columns)
weights.sort_values(ascending=True, inplace=True)

# Plot in Barplot horizontal
weights.plot.barh(color='blue')
plt.xlabel("Importance")
plt.ylabel("Feature")
plt.title("Feature Importance - Random Forest")

# Create scatter plot for Random Forest, showing accuracy in terms of
predicted vs actual target
```

```
plt.scatter(y_predict, y_test)
plt.xlabel("Predicted Price")
plt.ylabel("Actual Price")
plt.title("Predicted vs. Actual Price - RANDOM FOREST")
plt.show()
```

```
from sklearn.ensemble import GradientBoostingRegressor
# https://towardsdatascience.com/all-you-need-to-know-about-gradient-
boosting-algorithm-part-1-regression-2520a34a502

# GridSearched returned the below parameters as best
gbr = GradientBoostingRegressor(n_estimators=500,learning_rate=0.3)
gbr.fit(X_train, y_train)
```

```
# Compare training and test score
print(gbr.score(X_train,y_train))
print(gbr.score(X_test,y_test))
```

```
# Test the model
y_pred = gbr.predict(X_test)

# Compare Prediction to Test Targets
decimal_places = 3
mae = round(mean_absolute_error(y_test, y_pred ), decimal_places)
mse = round(mean_squared_error(y_test, y_pred), decimal_places)
rmse = round(sqrt(mse), decimal_places)
r2 = round(r2_score(y_test, y_pred), decimal_places)

print("Model 4: Gradient Boost")
print("Mean Absolute Error: ", mae)
print("Mean Squared Error: ", mse)
print("Rooted Mean Squared Error: ", rmse)
print("R-squared Value: ", r2)

results.append(["Gradient Boost", gbr.score(X_train,y_train), r2, rmse,
mse, mae])
```

```
# https://mljar.com/blog/feature-importance-in-random-forest/

# Plot important features as concluded by Gradient Boost model
weights = pd.Series(gbr.feature_importances_, index=X_train.columns)
weights.sort_values(ascending=True, inplace=True)
weights.plot.barh(color='blue')
plt.xlabel("Importance")
plt.ylabel("Feature")
plt.title("Feature Importance - Gradient Boost")
```

```
# Create scatter plot for Gradient Boost, showing accuracy in terms of
predicted vs actual target
```

```
plt.scatter(y_pred, y_test)
plt.xlabel("Predicted Price")
plt.ylabel("Actual Price")
plt.title("Predicted vs. Actual Price - GRADIENT BOOST")
plt.show()
```

```
# Create error plot for all 4 models.

# Convert results list into DataFrame
results_df = pd.DataFrame(results, columns=['Model', 'Training Score',
'Testing Score', 'RMSE', 'MSE', 'MAE'])

# Scatter plot of each models Training Score + Testing Score
plt.scatter(results_df['Model'], results_df['Training Score'])
plt.scatter(results_df['Model'], results_df['Testing Score'])

# Plot labels + legend
plt.title("R^2 Score of Each Model")
plt.legend(["Training Score", "Testing Score"], loc ="lower right")
plt.show()
```

```
# View final results in table form.
results_df
```

Appendix 2

Link to Video Presentation

https://youtu.be/qHMNtpEcoIM

Link to Dataset

The dataset used for this project can be found on Kaggle, at the following link:
https://www.kaggle.com/datasets/deepcontractor/car-price-prediction-challenge/data

Link to Notebook on GitHub

https://github.com/sumedha-git/EECS3401-Car_Price_AI_Model/blob/main/EECS3401_Final_Project_Group_1.ipynb

Works Cited

Agrawal, R. (2023, October 9). *Know the best evaluation metrics for your regression model !*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/05/know-the-best-evaluation-metrics-for-your-regression-model/

Alomari, R. (2018, April). A study of password recall, perceived memorability, and strength using bcis. https://ir.library.ontariotechu.ca/bitstream/handle/10155/1025/Alomari_Ruba.pdf?sequence=3

Create a pandas Dataframe by appending one row at a time. Stack Overflow. (1958, July 1). https://stackoverflow.com/questions/10715965/create-a-pandas-dataframe-by-appending-one-row-at-a-time

Decision trees, random forests and gradient boosting: What's the difference? Leon Lok. (2022, January 6). https://leonlok.co.uk/blog/decision-trees-random-forests-gradient-boosting-whats-the-difference/

GeeksforGeeks. (2020, April 12). *Matplotlib.pyplot.legend() in Python*. GeeksforGeeks. https://www.geeksforgeeks.org/matplotlib-pyplot-legend-in-python/

Jain, A. (2022, June 15). *Complete machine learning guide to parameter tuning in gradient boosting (GBM) in Python*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/

Masui, T. (2022, February 12). *All you need to know about gradient boosting algorithm − Part 1. regression*. Medium. https://towardsdatascience.com/all-you-need-to-know-about-gradient-boosting-algorithm-part-1-regression-2520a34a502

Plapinger, T. (2022, September 26). *Visualizing your exploratory data analysis*. Medium. https://towardsdatascience.com/visualizing-your-exploratory-data-analysis-d2d6c2e3b30e

Saini, A. (2023, August 2). *Gradient boosting algorithm: A complete guide for beginners*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/09/gradient-boosting-algorithm-a-complete-guide-for-beginners/

Sellahewa, K. (2023, May 14). *Random Forest explained! (regression and classification tasks)*. LinkedIn. https://www.linkedin.com/pulse/random-forest-explained-regression-classification-tasks-sellahewa