Machine Learning Assignment 1 - Predicting Car Prices!

This Jupyter notebook is a template for solving the assignment problem, i.e., Chaky company makes some car but he has difficulty setting the price for the car. . Here, I will try to apply the skills I've learned over the past lectures. This notebook contains the following structure:

- **1. Setup**: Import block with all necessary imports (also provide some blocks with connection to drive, kaggle, and etc. for future use)
- **2. Loading the Data**: Loading, EDA, data cleaning, feature selection, and preprocess the dataset.
- **3. Models**: Starter code for basic models to kickstart your experimentation.
- **4. Evaluation Metrics**: Tools to evaluate your models using various metrics.
- **5. Inference and Conclusion**: Testing the best model and generating Report.

Let's start!

Some notes:

The typical workflow of data science project is following:

```
1. Problem Definition
- Objective: Clearly define the problem you're trying to solve.
Understand the business or research goals and translate them into a
data science problem.
Tasks:
- - Identify the key objectives and success metrics.
- - Understand the constraints and resources available.
- - Formulate hypotheses or research questions.
2. Data Collection
- Objective: Gather the necessary data from various sources, which
could be internal databases, APIs, web scraping, or external datasets.
Tasks:
- - Identify data sources and acquire the data.
- - Integrate data from multiple sources if needed.
- - Ensure data privacy and compliance with regulations (e.g., GDPR).
3. Data Exploration and Analysis (Exploratory Data Analysis - EDA)
- Objective: Understand the data, its patterns, and any potential
issues through visualization and basic statistical analysis.
Tasks:
- - Summarize the data using descriptive statistics.
- - Visualize distributions, correlations, and trends.
- - Identify patterns, outliers, and potential relationships between
features.
- - Formulate additional hypotheses based on the data.
```

4. Data Preprocessing
- Objective: Clean and prepare the data for modeling.
Tasks:
- - Handle missing values (imputation or removal).
- - Handle outliers.
- - Encode categorical variables.
- - Normalize or standardize numerical features.
- - Split the data into training, validation, and test sets.
5. Feature Engineering
- Objective: Create new features or modify existing ones to improve model performance.
Tasks:
- - Create new features from existing data (e.g., interaction terms, polynomial features).
- - Apply feature scaling (normalization or standardization).
- - Transform features to handle skewness (e.g., log transformations).
- - Reduce dimensionality if necessary (e.g., PCA).
6. Model Selection
- Objective: Choose the appropriate machine learning models for the problem.
Tasks:
- - Compare different algorithms (e.g., linear models, decision trees, ensemble methods, neural networks).
- - Consider baseline models for comparison.
- - Choose models based on the problem type (e.g., classification, regression).
7. Model Training
- Objective: Train the chosen models on the preprocessed data.
Tasks:
- - Train the models using the training dataset.
- - Perform hyperparameter tuning (e.g., using grid search or random search).
- - Use cross-validation to evaluate model performance.
8. Model Evaluation
- Objective: Assess the model’s performance using relevant metrics and ensure it meets the project goals.
Tasks:
- - Evaluate model performance on the validation dataset.
- - Use appropriate metrics (e.g., accuracy, precision, recall, F1-score, RMSE).
- - Analyze model errors and refine the model if necessary.
9. Model Deployment
- Objective: Integrate the model into a production environment where it can be used to make predictions.
Tasks:
- - Deploy the model as a service (e.g., REST API, microservice).
- - Ensure scalability and monitor the model’s performance in production.
- - Handle model retraining as needed (e.g., with new data).

```
10. Monitoring and Maintenance
- Objective: Continuously monitor the model’s performance and maintain
its accuracy over time.
Tasks:
- - Track model performance using key metrics.
- - Monitor for data drift and update the model if necessary.
- - Address any issues in production and ensure model reliability.
11. Documentation and Reporting
- Objective: Document the entire process and communicate the results
to stakeholders.
Tasks:
- - Prepare detailed reports and visualizations.
- - Document the data, model, and processes.
- - Share insights and actionable recommendations with stakeholders.
12. Iteration and Optimization
- Objective: Refine the project by iterating over the steps to improve
results.
Tasks:
- - Revisit earlier steps based on feedback and new insights.
- - Optimize the model and the workflow for better performance.
```

1. Setup

The following libraries are required to run this notebook. If you are running this on Colab it should be all smooth sailing. If you are running it locally please make sure you have all of these installed.

```
# Installing packages I would need later to plot and visualizations
!pip install pandas==1.5.3 # Needed older version of pandas
!pip install ppscore
!pip install shap
```

```
Requirement already satisfied: pandas==1.5.3 in
/usr/local/lib/python3.10/dist-packages (1.5.3)
Requirement already satisfied: python-dateutil>=2.8.1 in
/usr/local/lib/python3.10/dist-packages (from pandas==1.5.3) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in
/usr/local/lib/python3.10/dist-packages (from pandas==1.5.3) (2024.1)
Requirement already satisfied: numpy>=1.21.0 in
/usr/local/lib/python3.10/dist-packages (from pandas==1.5.3) (1.26.4)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1-
>pandas==1.5.3) (1.16.0)
Requirement already satisfied: ppscore in
/usr/local/lib/python3.10/dist-packages (1.3.0)
Requirement already satisfied: pandas<2.0.0,>=1.0.0 in
/usr/local/lib/python3.10/dist-packages (from ppscore) (1.5.3)
```

```
Requirement already satisfied: scikit-learn<2.0.0,>=0.20.2 in
/usr/local/lib/python3.10/dist-packages (from ppscore) (1.3.2)
Requirement already satisfied: python-dateutil>=2.8.1 in
/usr/local/lib/python3.10/dist-packages (from pandas<2.0.0,>=1.0.0-
>ppscore) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in
/usr/local/lib/python3.10/dist-packages (from pandas<2.0.0,>=1.0.0-
>ppscore) (2024.1)
Requirement already satisfied: numpy>=1.21.0 in
/usr/local/lib/python3.10/dist-packages (from pandas<2.0.0,>=1.0.0-
>ppscore) (1.26.4)
Requirement already satisfied: scipy>=1.5.0 in
/usr/local/lib/python3.10/dist-packages (from scikit-
learn<2.0.0,>=0.20.2->ppscore) (1.13.1)
Requirement already satisfied: joblib>=1.1.1 in
/usr/local/lib/python3.10/dist-packages (from scikit-
learn<2.0.0,>=0.20.2->ppscore) (1.4.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from scikit-
learn<2.0.0,>=0.20.2->ppscore) (3.5.0)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1-
>pandas<2.0.0,>=1.0.0->ppscore) (1.16.0)
Requirement already satisfied: shap in /usr/local/lib/python3.10/dist-
packages (0.46.0)
Requirement already satisfied: numpy in
/usr/local/lib/python3.10/dist-packages (from shap) (1.26.4)
Requirement already satisfied: scipy in
/usr/local/lib/python3.10/dist-packages (from shap) (1.13.1)
Requirement already satisfied: scikit-learn in
/usr/local/lib/python3.10/dist-packages (from shap) (1.3.2)
Requirement already satisfied: pandas in
/usr/local/lib/python3.10/dist-packages (from shap) (1.5.3)
Requirement already satisfied: tqdm>=4.27.0 in
/usr/local/lib/python3.10/dist-packages (from shap) (4.66.5)
Requirement already satisfied: packaging>20.9 in
/usr/local/lib/python3.10/dist-packages (from shap) (24.1)
Requirement already satisfied: slicer==0.0.8 in
/usr/local/lib/python3.10/dist-packages (from shap) (0.0.8)
Requirement already satisfied: numba in
/usr/local/lib/python3.10/dist-packages (from shap) (0.60.0)
Requirement already satisfied: cloudpickle in
/usr/local/lib/python3.10/dist-packages (from shap) (2.2.1)
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in
/usr/local/lib/python3.10/dist-packages (from numba->shap) (0.43.0)
Requirement already satisfied: python-dateutil>=2.8.1 in
/usr/local/lib/python3.10/dist-packages (from pandas->shap) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in
/usr/local/lib/python3.10/dist-packages (from pandas->shap) (2024.1)
```

```
Requirement already satisfied: joblib>=1.1.1 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn->shap)
(1.4.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn->shap)
(3.5.0)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1-
>pandas->shap) (1.16.0)
```

```
# Import section, basically importing everything what I need later +
default imports
import os
import random
import zipfile
from collections import defaultdict

%matplotlib inline
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# For sklearn imports I will import them in model sections for better
explanation purposes
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import GradientBoostingRegressor

from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import ShuffleSplit, KFold

from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
```

#2.1 DataLoading

First thing we need to do is load in the data. We will be looking at the cars dataset (shared for this assignment cars). This dataset is tabular and contains information regarding car details(year, brand, mileage, and etc.) and we need to predict the price of the car(regression).

Using Colab (skip if running locally)

The below is for use in a Colab notebook. Make a new folder called 'assignment1' in your own Google Drive account. Simply upload everything into this 'assignment1' folder. **NB** compress the images folder into a .zip file before uploading it otherwise it will take very long to upload. So you

can upload '*.zip*', '.csv', '*.pkl', and other files. After that you can run the blocks of code below. First let's create a link to this new folder you created for ease of use.

```
# Skip this block
# Commenting all since I am not using google drive in this assignment

# # mount drive and create symlink
# from google.colab import drive
# drive.mount('/content/drive')
# data_folder_name = 'assignment1'
# path_name = f"/content/drive/My Drive/{data_folder_name}"
# # make sure to check if this is the correct directory for your
Google account some people do not have a space in 'My Drive'
# # !ln -s "/content/drive/My Drive/assignment1"
"/content/assignment1" # NB when changing this
```

API for unzipping if .zip files exist, for this assignment we are skiping this part.

```
# only run this once to unzip your files
# Skip this block
# Commenting all since I don't have any zip files in this assignment

# def unzip_folder(folder_path, extract_to=None):
#     """
#     Unzip the given folder.

#     Parameters:
#     folder_path (str): Path to the zipped folder.
#     extract_to (str): Directory to extract files to. Defaults to the
same directory as the zipped folder.

#     Returns:
#     None
#     """
#     if extract_to is None:
#         extract_to = os.path.dirname(folder_path)

#     with zipfile.ZipFile(folder_path, 'r') as zip_ref:
#         zip_ref.extractall(extract_to)
#     print(f'Extracted all files to {extract_to}')

# Example usage:
# filename = 'images.zip'
# cwd = os.getcwd()
# unzip_folder(os.path.join(cwd,data_folder_name,filename), extract_to
= os.path.join(cwd,data_folder_name))

# if running locally, change the data_folder_name
# data_folder_name = '/path_to_the_assignment' # uncomment and change
this to the correct directory if running locally
```

```
# train_csv_path = os.path.join(data_folder_name,"train.csv")

# Proceeding with this variant, usually would need above code
train_csv_path = 'cars.csv'
df = pd.read_csv(train_csv_path)
```

Data Preprocessing and Label Encoding

We need to represent categorical data into numerical form via encoding. This step should be done before EDA

```
# Let's observe what are columns and their data types
df.dtypes
```

```
name              object
year               int64
selling_price      int64
km_driven          int64
fuel              object
seller_type       object
transmission      object
owner             object
mileage           object
engine            object
max_power         object
torque            object
seats            float64
dtype: object
```

The task is following (note: I will deal with nan valuese on the fly alongs tasks, I am thinking since its regression task our predictions is approximate (it is okay to change with mean/median based on distribution), but for classification tasks I think it is better to drop such rows. Therefore, I will keep them.):

```
1. Feature owner - map First owner to 1, ..., Test drive car to 5
2. Feature fuel - remove all rows with CNG and LPG because CNG and LPG use a different mileage system (km/kg) which is different from kmpl for Diesel and Petrol
3. Feature mileage - remove "kmpl" and convert to float
4. Feature engine - remove "CC" and convert to numerical
5. Feature max power - same as engine
6. Feature brand - take first word and remove other
7. Drop feature torque
8. Test Drive cars are expensive, so delete all samples

# task 1 - Feature owner - map First owner to 1, ..., Test drive car to 5
df_copy = df.copy()
```

```
# First Owner   5289
# Second Owner  2105
# Third Owner   555
# Fourth & Above Owner     174
# Test Drive Car 5

# Better to use one-hot encoding, but as per hw instructions doing
mapping.
owner_map = {
    'owner': {
        "First Owner": 1,
        "Second Owner": 2,
        "Third Owner": 3,
        "Fourth & Above Owner": 4,
        "Test Drive Car": 5,
    }
}

df_copy.replace(owner_map, inplace=True)

# task8 - Test Drive cars are expensive, so delete all samples
df_copy = df_copy[df_copy.owner != 5]
print(df_copy.owner.value_counts())

# doing in such a sandwich way for testing purposes on the fly
df = df_copy.copy()
```

```
1    5289
2    2105
3     555
4     174
Name: owner, dtype: int64
```

```
# Encoder for binary categorical values
from sklearn.preprocessing import LabelEncoder

# defining encoder
le = LabelEncoder()

# task2 Feature fuel - remove all rows with CNG and LPG
df_copy = df.copy()

# df_copy['fuel'].value_counts()
# CNG 57
# LPG 38

print(df_copy.shape)
df_copy = df_copy[~df_copy.fuel.isin(['CNG', 'LPG'])]

print(df_copy.shape)
df_copy.fuel.value_counts()
```

```
# And also let's encode it
df_copy.fuel = le.fit_transform(df_copy.fuel)
print(df_copy.fuel.value_counts())


df = df_copy.copy()
```

```
(8123, 13)
(8028, 13)
0    4401
1    3627
Name: fuel, dtype: int64
```

```
# task6 - Feature brand - take first word and remove other
# same approach

df_copy = df.copy()

# Changing name to brand
df_copy.rename(columns = {'name': 'brand'}, inplace=True)
df_copy.brand = df_copy.brand.str.split().str[0]

print(df_copy.brand.isna().sum())

# Doing mapping

# Bad choice, I will proceed with one-hot encoding (though too much
values)
# brand_name_map = {'brand': {v:k for k, v in zip(range(1, 33),
#         ['Maruti', 'Skoda', 'Honda', 'Hyundai', 'Toyota', 'Ford',
'Renault',
#        'Mahindra', 'Tata', 'Chevrolet', 'Fiat', 'Datsun', 'Jeep',
#        'Mercedes-Benz', 'Mitsubishi', 'Audi', 'Volkswagen', 'BMW',
#        'Nissan', 'Lexus', 'Jaguar', 'Land', 'MG', 'Volvo', 'Daewoo',
#        'Kia', 'Force', 'Ambassador', 'Ashok', 'Isuzu', 'Opel',
'Peugeot'])
#     }
# }

# I will proceed with groupped one-hot encoding
group_map = {
    'Economy': ['Maruti', 'Tata', 'Hyundai', 'Datsun', 'Renault',
'Ford', 'Chevrolet', 'Fiat'],
    'Midrange': ['Honda', 'Toyota', 'Mahindra', 'Nissan', 'Skoda',
'Mitsubishi', 'Kia', 'MG'],
    'Luxury': ['Audi', 'BMW', 'Mercedes-Benz', 'Volvo', 'Jaguar',
'Lexus', 'Jeep', 'Land'],
    'Others': ['Daewoo', 'Ambassador', 'Ashok', 'Isuzu', 'Opel',
'Peugeot', 'Force']
}
```

```
# local mapper - later maybe need to define this in backend code
brand_to_group = {brand: group for group, brands in group_map.items()
for brand in brands}

# mapping cars to its groups
df_copy.brand = df_copy.brand.map(brand_to_group)

# creating columns of brand groupping
df_encoded = pd.get_dummies(df_copy, columns=['brand'],
drop_first=True)

df_encoded.head()

df = df_encoded.copy()
```

```
0
```

```
# Transmission feature has 2 classes only, so use LabelEncoder
df_copy = df.copy()

df_copy.transmission = le.fit_transform(df_copy.transmission)
print(df_copy.transmission.value_counts())

df = df_copy.copy()
```

```
1    6982
0    1046
Name: transmission, dtype: int64
```

```
# seller_type feature has 3 classes: individual,  dealer, trustmark
dealer -> use one-hot encoding
df_copy = df.copy()

# one-hot encoding, drop_first=True to drop one not required column
df_copy = pd.get_dummies(df_copy, columns=['seller_type'],
drop_first=True)
df_copy.head()

df = df_copy.copy()

# task3 - Feature mileage - remove "kmpl" and convert to float
# Hint: use df_copy.mileage.str.split

df.mileage = df.mileage.str.split().str[0].astype(float)

# task4 - Feature engine - remove "CC" and convert to numerical
# Same as task3

df.engine = df.engine.str.split().str[0].astype(float)
```

```
# task5 - Feature max power - same as engine

df.max_power = df.max_power.str.split().str[0].astype(float)

# task7 - dropping torque column
# so that it would not have impact on EDA - even though its bad
practice

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

# Checking if everything is fine
# But probably, it would be better to keep torque and transfer for
numerical form for the EDA basis
# I will test it in next iteration
df.dtypes
```

```
year                              int64
selling_price                     int64
km_driven                         int64
fuel                              int64
transmission                      int64
owner                             int64
mileage                         float64
engine                          float64
max_power                       float64
seats                           float64
brand_Luxury                      uint8
brand_Midrange                    uint8
brand_Others                      uint8
seller_type_Individual            uint8
seller_type_Trustmark Dealer      uint8
dtype: object
```

Now we can proceed with EDA

2.2 Exploratory Data Analysis (EDA)

DataFrame columns:

```
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   name           8128 non-null   object
 1   year           8128 non-null   int64
 2   selling_price  8128 non-null   int64
 3   km_driven      8128 non-null   int64
 4   fuel           8128 non-null   object
 5   seller_type    8128 non-null   object
 6   transmission   8128 non-null   object
```

```
 7   owner          8128 non-null   object
 8   mileage        7907 non-null   object
 9   engine         7907 non-null   object
 10  max_power      7913 non-null   object
 11  torque         7906 non-null   object
 12  seats          7907 non-null   float64
```

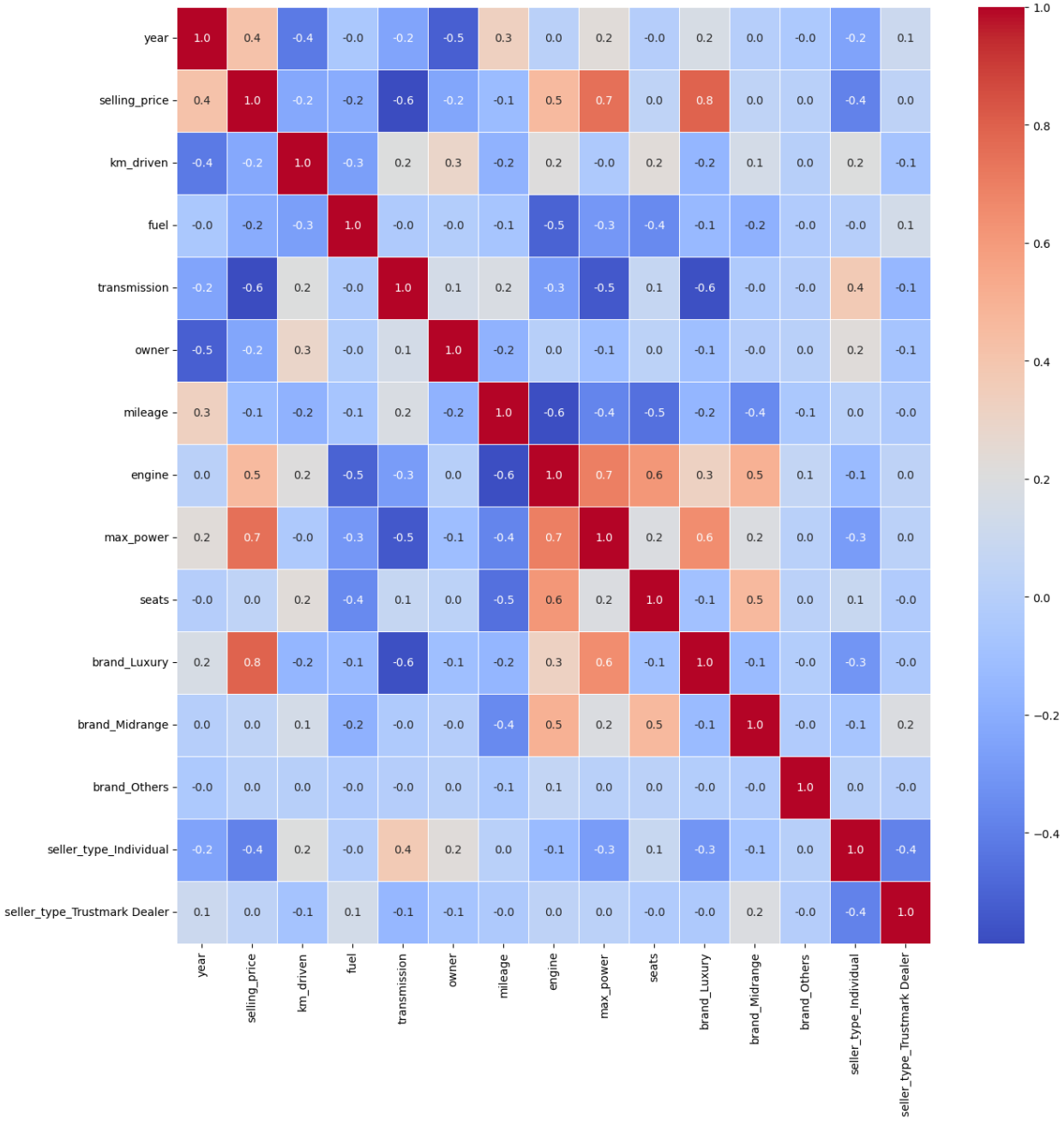
General Notes about EDA:

```
value_counts(): Frequency counts
outliers: the value that is considerably higher or lower from rest of
the data

Value at 75% is Q3 and value at 25% is Q1 -> Q stands for "quartile"
Outlier are smaller than Q1 - 1.5(Q3-Q1) and bigger than Q3 + 1.5(Q3-
Q1). (Q3-Q1) = IQR
IQR stands for "interquartile range"

We will use describe() method. Describe method includes:
count: number of entries
mean: average of entries
std: standart deviation
min: minimum entry
25%: first quantile
50%: median or second quantile
75%: third quantile
max: maximum entry

# Let's see all columns
df.columns

Index(['year', 'selling_price', 'km_driven', 'fuel', 'transmission',
'owner',
       'mileage', 'engine', 'max_power', 'seats', 'brand_Luxury',
       'brand_Midrange', 'brand_Others', 'seller_type_Individual',
       'seller_type_Trustmark Dealer'],
      dtype='object')

# Some basic info about each column
# We see there are null values
df.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 8028 entries, 0 to 8127
Data columns (total 15 columns):
 #   Column                         Non-Null Count  Dtype
---  ------                         --------------  -----
 0   year                           8028 non-null   int64
 1   selling_price                  8028 non-null   int64
 2   km_driven                      8028 non-null   int64
```

```
 3   fuel                          8028 non-null   int64
 4   transmission                  8028 non-null   int64
 5   owner                         8028 non-null   int64
 6   mileage                       7814 non-null   float64
 7   engine                        7814 non-null   float64
 8   max_power                     7820 non-null   float64
 9   seats                         7814 non-null   float64
 10  brand_Luxury                  8028 non-null   uint8
 11  brand_Midrange                8028 non-null   uint8
 12  brand_Others                  8028 non-null   uint8
 13  seller_type_Individual        8028 non-null   uint8
 14  seller_type_Trustmark Dealer  8028 non-null   uint8
dtypes: float64(4), int64(6), uint8(5)
memory usage: 729.1 KB
```

Plotting:

```
# # lets drop for now not number columns
# df_only_nums = df[['year', 'selling_price', 'km_driven', 'seats']]
# #only numbers

# previously was keeping categorical as categorical, but now we can
plot all features
# print(df.corr())

# #correlation map
f, ax = plt.subplots(figsize=(15, 15))
sns.heatmap(df.corr(), annot=True, linewidths=.5, fmt= '.1f',ax=ax,
cmap="coolwarm")

# So the more red color, the more correlation
plt.show()
```




```
# Observing first 5 data
df.head(5)

# Observing last 5 data
#df.tail()
```


```
{"summary":"{\n  \"name\": \"#df\",\n  \"rows\": 5,\n  \"fields\": [\n
{\n      \"column\": \"year\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 3,\n        \"min\": 2006,\n
\"max\": 2014,\n        \"num_unique_values\": 4,\n
```

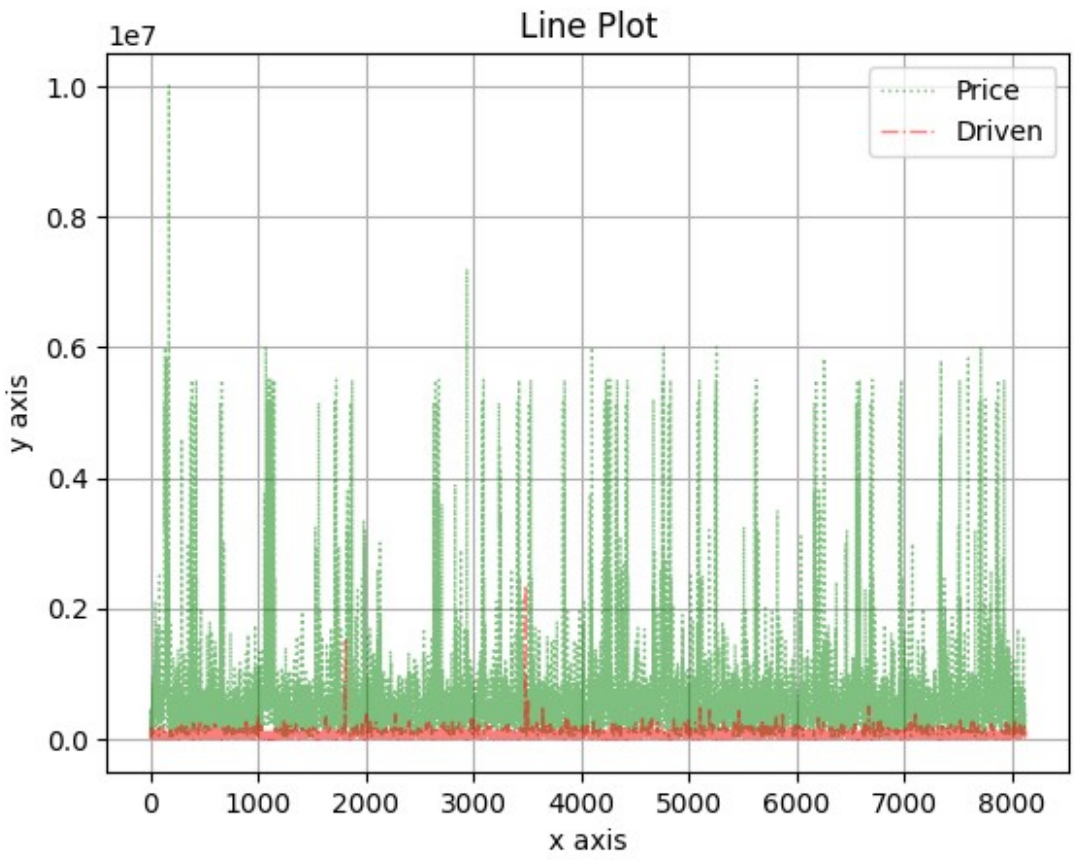
```
\"samples\": [\n          2006,\n          2007,\n          2014\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"selling_price\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
138303,\n        \"min\": 130000,\n        \"max\": 450000,\n
\"num_unique_values\": 5,\n        \"samples\": [\n          370000,\n
130000,\n          158000\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"km_driven\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 11704,\n        \"min\":
120000,\n        \"max\": 145500,\n        \"num_unique_values\": 4,\n
\"samples\": [\n          120000,\n          127000,\n
145500\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"fuel\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": 0,\n        \"min\": 0,\n        \"max\": 1,\n
\"num_unique_values\": 2,\n        \"samples\": [\n          1,\n
0\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"transmission\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 0,\n        \"min\": 1,\n
\"max\": 1,\n        \"num_unique_values\": 1,\n        \"samples\":
[\n          1\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"owner\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": 0,\n        \"min\": 1,\n        \"max\": 3,\n
\"num_unique_values\": 3,\n        \"samples\": [\n          1\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"mileage\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
3.240388865553021,\n        \"min\": 16.1,\n        \"max\": 23.4,\n
\"num_unique_values\": 5,\n        \"samples\": [\n          21.14\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"engine\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\":
113.7356584365695,\n        \"min\": 1248.0,\n        \"max\":
1498.0,\n        \"num_unique_values\": 5,\n        \"samples\": [\n
1498.0\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"max_power\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 11.543642406103888,\n        \"min\":
74.0,\n        \"max\": 103.52,\n        \"num_unique_values\": 5,\n
\"samples\": [\n          103.52\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"seats\",\n      \"properties\": {\
n        \"dtype\": \"number\",\n        \"std\": 0.0,\n
\"min\": 5.0,\n        \"max\": 5.0,\n        \"num_unique_values\":
1,\n        \"samples\": [\n          5.0\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
```

```
n    },\n    {\n      \"column\": \"brand_Luxury\",\n
\"properties\": {\n        \"dtype\": \"uint8\",\n
\"num_unique_values\": 1,\n        \"samples\": [\n          0\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"brand_Midrange\",\n
\"properties\": {\n        \"dtype\": \"uint8\",\n
\"num_unique_values\": 2,\n        \"samples\": [\n          1\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"brand_Others\",\n
\"properties\": {\n        \"dtype\": \"uint8\",\n
\"num_unique_values\": 1,\n        \"samples\": [\n          0\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"seller_type_Individual\",\n
\"properties\": {\n        \"dtype\": \"uint8\",\n
\"num_unique_values\": 1,\n        \"samples\": [\n          1\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"seller_type_Trustmark Dealer\",\
n      \"properties\": {\n        \"dtype\": \"uint8\",\n
\"num_unique_values\": 1,\n        \"samples\": [\n          0\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    }\n  ]\n}","type":"dataframe"}
```

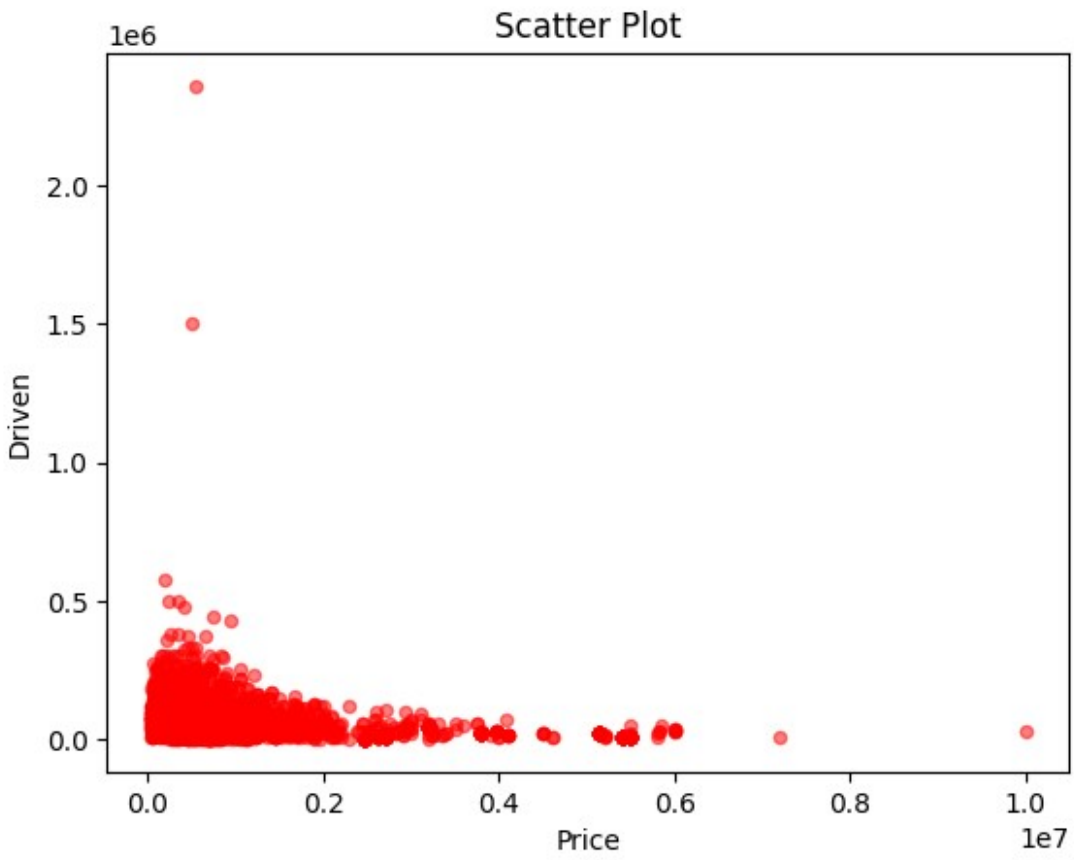

Let's try to plot some line, scatter and histogram plots. To choose between, there are some differences in plots:

- Line plot is better when x axis is time.
- Box plots: visualize basic statistics like outliers, min/max or quantiles
- Scatter is better when there is correlation between two variables
- Histogram is better when we need to see distribution of numerical data.
- Customization: Colors,labels,thickness of line, title, opacity, grid, figsize, ticks of axis and linestyle

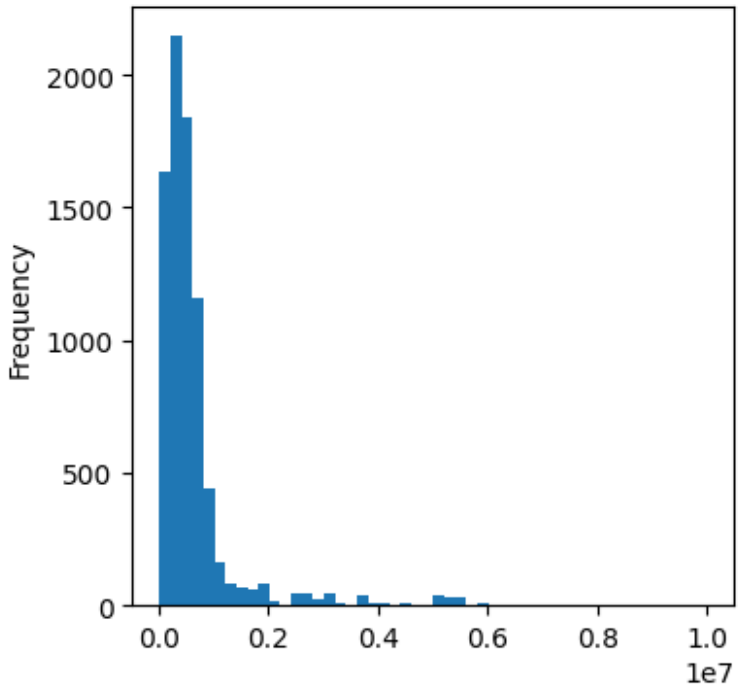
```
# Line plot
# It might be seen there is no correlation between features,
# but basically I am just exploring type of plots

# Line plot is better when x axis is time
df['selling_price'].plot(kind = 'line', color = 'g',label =
'Price',linewidth=1,alpha = 0.5,grid = True,linestyle = ':')
df['km_driven'].plot(color = 'r',label = 'Driven',linewidth=1, alpha =
0.5,grid = True,linestyle = '-.')
plt.legend(loc='upper right')     # legend = puts label into plot
plt.xlabel('x axis')              # label = name of label
plt.ylabel('y axis')
plt.title('Line Plot')            # title = title of plot
plt.show()
```




```
# Scatter plot
# Scatter is better when there is correlation between two variables

df.plot(kind='scatter', x='selling_price', y='km_driven',alpha =
0.5,color = 'red')
plt.xlabel('Price')              # label = name of label
plt.ylabel('Driven')
plt.title('Scatter Plot')

Text(0.5, 1.0, 'Scatter Plot')
```

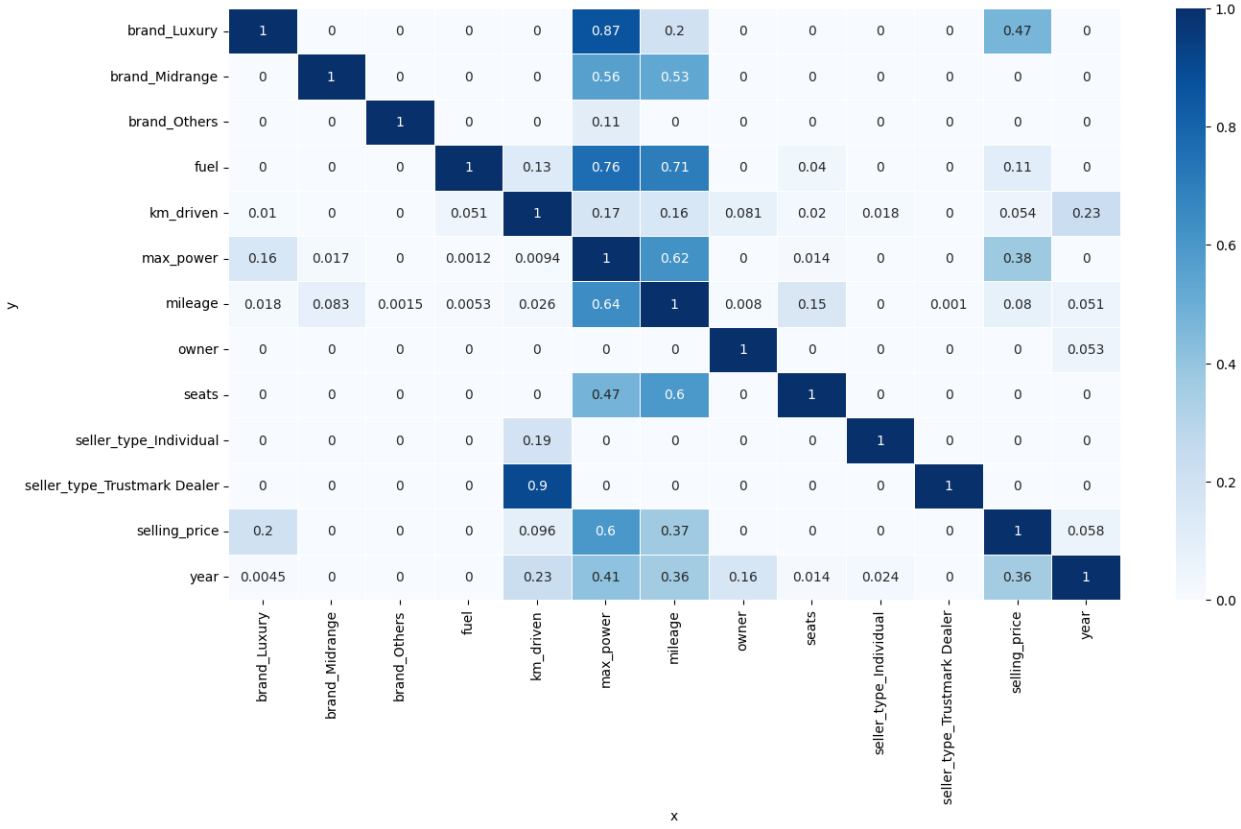



```
# Histogram
# bins = number of bar in figure
# Histogram is better when we need to see distribution of numerical
data.

df['selling_price'].plot(kind = 'hist',bins = 50,figsize = (4,4))
plt.show()
```



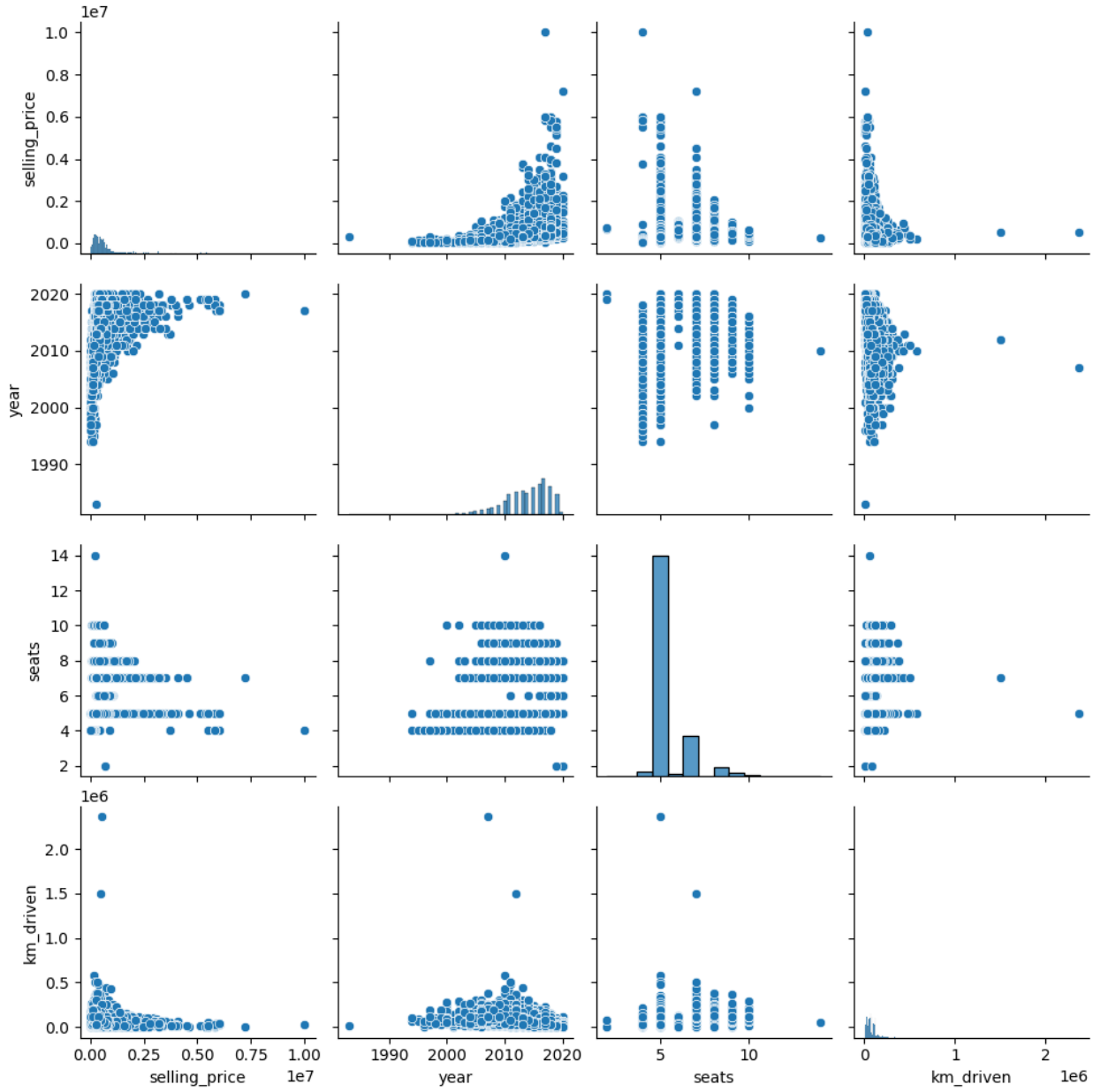

```
import ppscore as pps

# This is another way to check the predictive power of some feature.
# Unlike correlation, `pps` actually obtained from actual prediction.

# before using pps, let's drop engine and transmission
dfcopy = df.copy()
dfcopy.drop(['engine', 'transmission'], axis='columns', inplace=True)

#this needs some minor preprocessing because seaborn.heatmap
unfortunately does not accept tidy data
matrix_df = pps.matrix(dfcopy)[['x', 'y',
'ppscore']].pivot(columns='x', index='y', values='ppscore')

#plot
plt.figure(figsize = (15,8))
sns.heatmap(matrix_df, vmin=0, vmax=1, cmap="Blues", linewidths=0.5,
annot=True)
```

```
<Axes: xlabel='x', ylabel='y'>
```




```
sns.pairplot(df[['selling_price', 'year', 'seats', 'km_driven']])

# From the result we can see that we need to normalize features
```

```
<seaborn.axisgrid.PairGrid at 0x7d5730f6cfa0>
```



2.3 Feature Engineering

[08/18/2024] 1st attempt: I think there is no need to create new features, I will try with existing ones (created this section for future use - will require this)

2.4 Feature Selection

```
[08/18/2024] 1st attempt: I am thinking taking all features except the ones that we need to drop: torque
```

```
[08/20/2024] 2nd attempt: I will choose 5 features that are most
important: max_power, engine, km_driven, mileage, and year

# Just to remind what are columns
df.columns

Index(['year', 'selling_price', 'km_driven', 'fuel', 'transmission',
'owner',
       'mileage', 'engine', 'max_power', 'seats', 'brand_Luxury',
       'brand_Midrange', 'brand_Others', 'seller_type_Individual',
       'seller_type_Trustmark Dealer'],
      dtype='object')
```

Outliers

I want to handle them before proceeding to training ['year', 'km_driven', 'mileage', 'engine', 'max_power'] - will chose those only

[08/21/2024] 3rd attempt - trying with outliers before splitting the dataset

```
# To see all outliers

def outlier_count(col, data = df):
    # calculate your 25% quatile and 75% quatile
    q75, q25 = np.percentile(data[col], [75, 25])

    # calculate your inter quatile
    iqr = q75 - q25

    # min_val and max_val
    min_val = q25 - (iqr*1.5)
    max_val = q75 + (iqr*1.5)

    # count number of outliers, which are the data that are less than
min_val or more than max_val calculated above
    outlier_count = len(np.where((data[col] > max_val) | (data[col] <
min_val))[0])

    # calculate the percentage of the outliers
    outlier_percent = round(outlier_count/len(data[col])*100, 2)

    if(outlier_count > 0):
        print("\n"+15*'-' + col + 15*'-'+"\n")
        print('Number of outliers: {}'.format(outlier_count))
        print('Percent of data that is outlier: {}
%'.format(outlier_percent))

# Printing outliers per column
for col in ['year', 'km_driven', 'mileage', 'engine', 'max_power']:
    outlier_count(col)
```

```
---------------year---------------

Number of outliers: 78
Percent of data that is outlier: 0.97%

---------------km_driven---------------

Number of outliers: 168
Percent of data that is outlier: 2.09%
```

Let's not remove them, but cap them to a fixed value (5th or 95th percentile) - reduce impact of extreme values

```
# Capping outliers
def cap_outliers(df, column):
    lower_limit = df[column].quantile(0.05)
    upper_limit = df[column].quantile(0.95)
    df[column] = np.where(df[column] < lower_limit, lower_limit,
df[column])
    df[column] = np.where(df[column] > upper_limit, upper_limit,
df[column])
    return df

# applying for 'year' and 'km_driven' since there are only two
outliers
# for chosen set of features
df = cap_outliers(df, 'year')
df = cap_outliers(df, 'km_driven')

# Printing outliers per column
for col in ['year', 'km_driven', 'mileage', 'engine', 'max_power']:
    outlier_count(col)

# Same approach as in label encoding
df_copy = df.copy()

# shape (m,)
y = df_copy['selling_price']
# df_copy = df_copy.drop(columns=['selling_price'])
print(y.shape)
assert len(y.shape) == 1

# Taking shape (m, n)
X = df_copy[['year', 'km_driven', 'mileage', 'engine', 'max_power']]
print(X.shape)
assert len(X.shape) == 2

(8028,)
(8028, 5)
```

```
from sklearn.model_selection import train_test_split

# Splitting the dataset, will proceed with processing it
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.13, random_state=42)
```

2.5 Preprocessing

```
# Let's see what is the train dataset size
X_train.shape
```

```
(6984, 5)
```

```
# Same for test
X_test.shape
```

```
(1044, 5)
```

NUll values

```
# Let's observe all null values in training set (did not deal with
them - to avoid data leakage)
X_train.isna().sum()
```

```
year           0
km_driven      0
mileage      187
engine       187
max_power    181
dtype: int64
```

```
# Same for the testing dataset
X_test.isna().sum()
```

```
year          0
km_driven     0
mileage      27
engine       27
max_power    27
dtype: int64
```

```
# Removing null values for mileage
print(X_train.mileage.mean(), X_train.mileage.median())

# sns.distplot(X_train, x=X_train.mileage)

# Interchanging nan values with mean - the distribution is normal
X_train.mileage.fillna(X_train.mileage.mean(), inplace=True)
X_test.mileage.fillna(X_train.mileage.mean(), inplace=True)
```

```
19.38204354862439 19.3
```

```
# df_copy.engine.isna().sum() # 214
print(X_train.engine.mean(), X_train.engine.median())

# sns.distplot(X_train, x=X_train['engine'])

# Interchanging nan values with median - the distribution is skewed
X_train.engine.fillna(X_train.engine.mean(), inplace=True)
X_test.engine.fillna(X_train.engine.mean(), inplace=True)
```

```
1463.756068853906 1248.0
```

```
# df_copy.max_power.isna().sum() # 208
print(X_train.max_power.mean(), X_train.max_power.median())

# sns.distplot(X_train, x=X_train.max_power) # distribution is skewed
a little

# Interchanging nan values with median - the distribution is skewed
X_train.max_power.fillna(X_train.max_power.mean(), inplace=True)
X_test.max_power.fillna(X_train.max_power.mean(), inplace=True)
```

```
91.74543877701014 82.85
```

```
# # And we want to remove all null values from seats feature - 214
rows
# print(X_train.seats.mean(), X_train.seats.median())

# # sns.distplot(X_train, x=X_train.seats) # distribution is skewed a
little

# # Interchanging nan values with median - the distribution is skewed
# X_train.seats.fillna(X_train.seats.mean(), inplace=True)
# X_test.seats.fillna(X_train.seats.mean(), inplace=True)

# Now verify if everything is fine
X_train.isna().sum()
```

```
year          0
km_driven     0
mileage       0
engine        0
max_power     0
dtype: int64
```

```
# Same for test set
X_test.isna().sum()
```

```
year          0
km_driven     0
mileage       0
```

```
engine       0
max_power    0
dtype: int64
```

```
# Just to be sure
y_train.isna().sum()
```

```
0
```

```
# Now we can proceed
y_test.isna().sum()
```

```
0
```

Scaling

```
# Observing what need to be scaled
X_train.head()
```


```
{"summary":"{\n  \"name\": \"X_train\",\n  \"rows\": 6984,\n
\"fields\": [\n    {\n      \"column\": \"year\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
3.659587820815056,\n        \"min\": 2006.0,\n        \"max\":
2019.0,\n        \"num_unique_values\": 14,\n        \"samples\": [\n
2015.0,\n          2008.0,\n          2016.0\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"km_driven\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
39989.476920443805,\n        \"min\": 9000.0,\n        \"max\":
150000.0,\n        \"num_unique_values\": 665,\n        \"samples\":
[\n          36521.0,\n          37161.0,\n          61260.0\
n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"mileage\",\n      \"properties\": {\n        \"dtype\": \"number\",\
n        \"std\": 3.957601312730234,\n        \"min\": 0.0,\n
\"max\": 28.4,\n        \"num_unique_values\": 355,\n
\"samples\": [\n          13.73,\n          18.9,\n          24.04\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"engine\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\":
499.16548108959586,\n        \"min\": 624.0,\n        \"max\":
3604.0,\n        \"num_unique_values\": 121,\n        \"samples\": [\n
1495.0,\n          793.0,\n          1197.0\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"max_power\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
34.98868265709969,\n        \"min\": 0.0,\n        \"max\": 282.0,\n
\"num_unique_values\": 300,\n        \"samples\": [\n          187.4,\
n          194.0,\n          184.0\n        ],\n
```

\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   }\n]\n}","type":"dataframe","variable_name":"X_train"}

```
# We need to scale all numerics whose difference is large
from sklearn.preprocessing import StandardScaler

# After observing above, we can proceed with the following columns
col_names = ['km_driven', 'mileage', 'engine', 'max_power']

# Defining Scaler
sc = StandardScaler()

# Scaling is performed
X_train[col_names] = sc.fit_transform(X_train[col_names])
X_test[col_names] = sc.transform(X_test[col_names])

# Let's see if its fine
X_train.head()
```

{"summary":"{\n \"name\": \"X_train\",\n \"rows\": 6984,\n
\"fields\": [\n   {\n   \"column\": \"year\",\n
\"properties\": {\n   \"dtype\": \"number\",\n   \"std\":
3.659587820815056,\n   \"min\": 2006.0,\n   \"max\":
2019.0,\n   \"num_unique_values\": 14,\n   \"samples\": [\n
2015.0,\n   2008.0,\n   2016.0\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"km_driven\",\n
\"properties\": {\n   \"dtype\": \"number\",\n   \"std\":
1.000071599899852,\n   \"min\": -1.448991818002361,\n
\"max\": 2.07718822847358,\n   \"num_unique_values\": 665,\n
\"samples\": [\n   -0.76073649134233,\n   -
0.7447311351030066,\n   -0.14205444758510796\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"mileage\",\n   \"properties\":
{\n   \"dtype\": \"number\",\n   \"std\":
1.0000715998998517,\n   \"min\": -4.897772607526585,\n
\"max\": 2.2788051204512874,\n   \"num_unique_values\": 355,\n
\"samples\": [\n   -1.4282510510077053,\n   -
0.12181066883708584,\n   1.1770488213673604\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"engine\",\n   \"properties\":
{\n   \"dtype\": \"number\",\n   \"std\":
1.000071599899852,\n   \"min\": -1.6824404473465515,\n
\"max\": 4.287951097349699,\n   \"num_unique_values\": 121,\n
\"samples\": [\n   0.06259681286500324,\n   -
1.343851128201026,\n   -0.5344423416046217\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"max_power\",\n
\"properties\": {\n   \"dtype\": \"number\",\n   \"std\":
1.000071599899852,\n   \"min\": -2.622333874082587,\n

\"max\": 5.43799220151296,\n   \"num_unique_values\": 300,\n
\"samples\": [\n   2.734067213033035,\n
2.9227131424618666,\n   2.636885976660606\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   }\n]\n}","type":"dataframe","variable_name":"X_train"}

```
# Same for test set
X_test.head()
```

{"summary":"{\n \"name\": \"X_test\",\n \"rows\": 1044,\n
\"fields\": [\n   {\n   \"column\": \"year\",\n
\"properties\": {\n   \"dtype\": \"number\",\n   \"std\":
3.5763468117108608,\n   \"min\": 2006.0,\n   \"max\":
2019.0,\n   \"num_unique_values\": 14,\n   \"samples\": [\n
2008.0,\n   2017.0,\n   2011.0\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"km_driven\",\n
\"properties\": {\n   \"dtype\": \"number\",\n   \"std\":
0.9870045152499457,\n   \"min\": -1.448991818002361,\n
\"max\": 2.07718822847358,\n   \"num_unique_values\": 189,\n
\"samples\": [\n   -0.43457734122786745,\n
0.03400447104744967,\n   -1.448991818002361\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"mileage\",\n   \"properties\":
{\n   \"dtype\": \"number\",\n   \"std\":
0.9821024110837534,\n   \"min\": -2.4971568182382127,\n
\"max\": 5.715476145116748,\n   \"num_unique_values\": 239,\n
\"samples\": [\n   0.0601307383510862,\n   0.0,\n
-1.4611015828611253\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   },\n   {\n   \"column\":
\"engine\",\n   \"properties\": {\n   \"dtype\": \"number\",\n
\"std\": 0.9821540596402079,\n   \"min\": -1.343851128201026,\n
\"max\": 4.287951097349699,\n   \"num_unique_values\": 88,\n
\"samples\": [\n   -0.24594019984748752,\n   -
0.19184604826802484,\n   -1.3318302056278122\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"max_power\",\n
\"properties\": {\n   \"dtype\": \"number\",\n   \"std\":
1.0747809681788743,\n   \"min\": -1.6448049670422762,\n
\"max\": 8.810752757967833,\n   \"num_unique_values\": 187,\n
\"samples\": [\n   8.810752757967833,\n
0.06444148444926165,\n   0.6646785326319087\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   }\n]\n}","type":"dataframe","variable_name":"X_test"}

```
# Same for selling price, we want to do np.log transformation
y_train = np.log(y_train)
y_train
```

```
# We dont need to log the actual target set - so leaving it
# y_test = np.log(y_test)
```

```
4419    14.508658
6103    14.115615
7893    12.429216
7427    13.337475
1448    13.458836
          ...
5293    12.611538
5461    12.847927
865     13.527828
7701    15.454507
7366    13.560618
Name: selling_price, Length: 6984, dtype: float64
```

3. Modeling

[8/19/2024] first attempt - at first iteration, I proceeded with RandomForest, using cross_validation+gridsearch found the

```
best params: {'bootstrap': True, 'max_depth': None, 'n_estimators':
15}
mse error: -0.04769509278891849
(y_test is scaled into np.log - otherwise error is very huge - because
y_test and y_preds are huge numbers)
```

Also looked for grid.best_estimator_.feature_importances_:

```
0     year  0.464258
7     max_power  0.396566
6     engine     0.053404
1     km_driven  0.026052
5     mileage    0.025340
8     seats 0.008029
10    brand_Midrange   0.007669
4     owner 0.006844
9     brand_Luxury     0.006078
2     fuel  0.002882
12    seller_type_Individual      0.001563
3     transmission     0.001192
11    brand_Others     0.000072
13    seller_type_Trustmark Dealer     0.000052
```

Checked also with other metrics - seems fine, I will proceed to task2:

```
from sklearn.linear_model import LinearRegression  # Drawing a line
based on linear regression for continuous prediction
from sklearn.naive_bayes import GaussianNB  # Not commonly used for
regression

## Situational (but we don't use much)
from sklearn.neighbors import KNeighborsRegressor  # KNN for
regression

## Complex
from sklearn.ensemble import RandomForestRegressor  # Using trees to
predict continuous values
from sklearn.svm import SVR  # Drawing a line (or hyperplane) based on
maximum distance for regression
from sklearn.ensemble import GradientBoostingRegressor  #<<-------is
the MOST complex
from xgboost import XGBRegressor #<<------- well, now it is the MOST
complex
from sklearn.linear_model import Ridge #<<------- Fine

# Defining each models, where reproducing randomness
lr = LinearRegression()
rf = RandomForestRegressor(random_state=52)
sv = SVR()
knn = KNeighborsRegressor()
gbr = GradientBoostingRegressor(random_state=52)
rdg = Ridge(random_state=42)
xgb = XGBRegressor(random_state=42)

# Creating a list of models to run cross_validation
models = [lr, rf, sv, knn, gbr, rdg, xgb]

# For printing purposes
model_name = ['LinearRegression', 'RandomForestRegressor', 'SVR',
              'KNeighborsRegressor', 'GradientBoostingRegressor',
'Ridge',
              'XGBRegressor']

# Perform cross validation using KFold
from sklearn.model_selection import KFold, cross_val_score

kfold = KFold(n_splits=5, shuffle=True, random_state=999)

for i, model in enumerate(models):
    score = cross_val_score(model, X_train, y_train, cv=kfold,
scoring='neg_mean_squared_error')  # Common scoring metric for
regression
    print(model_name[i], " : ", " scores: ", score, "- Scores mean: ",
score.mean(), "- Scores std (lower better): ", score.std())  # The
scores here are negative MSE, closer to 0 is better
```

```
LinearRegression  :   scores:  [-0.10145655 -0.09800626 -0.09851201 -
0.10454421 -0.09745995] - Scores mean:  -0.09999579544523861 - Scores
std (lower better):  0.0026612889921397947
RandomForestRegressor  :   scores:  [-0.05371637 -0.05033086 -
0.05072447 -0.05320963 -0.05316482] - Scores mean:  -
0.05222923113712748 - Scores std (lower better):
0.0014082766321813378
SVR  :   scores:  [-0.67019568 -0.62142392 -0.65569914 -0.70181133 -
0.67099462] - Scores mean:  -0.6640249393607249 - Scores std (lower
better):  0.026071559678470017
KNeighborsRegressor  :   scores:  [-0.07950841 -0.0824752  -0.07430314
-0.08219958 -0.07834243] - Scores mean:  -0.07936575189927411 - Scores
std (lower better):  0.0029798249524733947
GradientBoostingRegressor  :   scores:  [-0.06424864 -0.05854013 -
0.0618793  -0.0629295  -0.06387734] - Scores mean:  -
0.062294982536248565 - Scores std (lower better):
0.0020491441965677392
Ridge  :   scores:  [-0.10145676 -0.09800482 -0.09851205 -0.10454348 -
0.09746151] - Scores mean:  -0.0999957259851429 - Scores std (lower
better):  0.0026609728448584217
XGBRegressor  :   scores:  [-0.05071456 -0.04721526 -0.04987028 -
0.05147016 -0.05258891] - Scores mean:  -0.050371835793018925 - Scores
std (lower better):  0.0018140948796450222
```

```python
# Choosing RandomForest because it has almost best performance
(slighlt lower performance than more complicated ones)
rf = RandomForestRegressor(random_state=52)

# Grid to search parameters
# param_grid = {
#     'n_estimators': [100, 200, 300],        # Number of trees in the
forest
#     'max_features': ['auto', 'sqrt'],       # Number of features to
consider for splits
#     'max_depth': [None, 10, 20, 30],        # Maximum depth of each
tree
#     'min_samples_split': [2, 5, 10],        # Minimum number of
samples required to split an internal node
#     'min_samples_leaf': [1, 2, 4]           # Minimum number of
samples required to be at a leaf node
# }

# Running above params taking so many time, will proceed with less
time-consuming one
param_grid = {
    'bootstrap': [True],
    'max_depth': [5, 10, None],
    'n_estimators': [5, 6, 7, 8, 9, 10, 11, 12, 13, 15]
}
```

```
# GridSearchCV with cross-validation
grid = GridSearchCV(rf, param_grid, scoring="neg_mean_squared_error",
cv=kfold, refit=True, return_train_score=True)

# Fit the grid, performing cross-validation across all combinations
grid.fit(X_train, y_train)

# Print the best parameters and the best score
print("Best parameters:", grid.best_params_)
print("Best score (negative MSE):", grid.best_score_)
print("Cross-validation results:", grid.cv_results_)
```

```
Best parameters: {'bootstrap': True, 'max_depth': None,
'n_estimators': 15}
Best score (negative MSE): -0.053924765728631106
Cross-validation results: {'mean_fit_time': array([0.03198304,
0.03612061, 0.04055409, 0.04476023, 0.05304184,
       0.06119442, 0.06265736, 0.06904616, 0.07236519, 0.08338952,
       0.05088358, 0.05983081, 0.07224836, 0.08015676, 0.08906469,
       0.10183458, 0.10773344, 0.15622807, 0.18265705, 0.2160686 ,
       0.09819832, 0.09151855, 0.10408058, 0.11998181, 0.13448792,
       0.14753876, 0.16400971, 0.17747679, 0.19289255, 0.22264953]),
'std_fit_time': array([0.00158698, 0.00188377, 0.00108663, 0.00028124,
0.00178767,
       0.00403017, 0.00149626, 0.0018681 , 0.00136285, 0.00131975,
       0.00188848, 0.00118317, 0.00764465, 0.00429461, 0.0016795 ,
       0.00401011, 0.00275984, 0.01982639, 0.0020795 , 0.00944404,
       0.02148572, 0.0030988 , 0.00098625, 0.00374794, 0.00458623,
       0.00116398, 0.00457438, 0.00309031, 0.0031295 , 0.00363427]),
'mean_score_time': array([0.00356603, 0.00359879, 0.00317793,
0.00338159, 0.00348887,
       0.00400834, 0.00387979, 0.00422955, 0.00387583, 0.00409188,
       0.004106  , 0.00422726, 0.00409417, 0.00423875, 0.00448742,
       0.00465574, 0.00488439, 0.00663347, 0.00717764, 0.00980606,
       0.00565448, 0.00530381, 0.00550814, 0.00636387, 0.00637755,
       0.00675607, 0.00691528, 0.00726538, 0.00796428, 0.0092978 ]),
'std_score_time': array([8.40430449e-04, 5.80335870e-04, 8.81801227e-
05, 2.07621360e-04,
       9.37889842e-05, 7.76381326e-04, 2.18407191e-04, 7.69205070e-04,
       9.02415550e-05, 1.61394734e-04, 6.30773404e-04, 8.88379153e-04,
       1.95318542e-04, 8.03940378e-05, 1.18795688e-04, 4.53926935e-05,
       1.02627786e-04, 8.83962190e-04, 5.78068320e-04, 1.30926742e-03,
       1.18308038e-03, 4.33484643e-04, 9.88053768e-05, 6.36618824e-04,
       2.46928206e-04, 2.59428645e-04, 1.16695936e-04, 1.06914016e-04,
       3.45865989e-04, 1.39343153e-03]), 'param_bootstrap':
masked_array(data=[True, True, True, True, True, True, True, True,
True,
                   True, True, True, True, True, True, True, True,
True,
                   True, True, True, True, True, True, True, True,
```

```
True,
                   True, True, True],
             mask=[False, False, False, False, False, False, False,
False,
                   False, False, False, False, False, False, False,
False,
                   False, False, False, False, False, False, False,
False,
                   False, False, False, False, False, False],
       fill_value='?',
            dtype=object), 'param_max_depth': masked_array(data=[5, 5,
5, 5, 5, 5, 5, 5, 5, 10, 10, 10, 10, 10, 10,
                   10, 10, 10, 10, None, None, None, None, None, None,
                   None, None, None, None],
             mask=[False, False, False, False, False, False, False,
False,
                   False, False, False, False, False, False, False,
False,
                   False, False, False, False, False, False, False,
False,
                   False, False, False, False, False, False],
       fill_value='?',
            dtype=object), 'param_n_estimators': masked_array(data=[5,
6, 7, 8, 9, 10, 11, 12, 13, 15, 5, 6, 7, 8, 9, 10,
                   11, 12, 13, 15, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15],
             mask=[False, False, False, False, False, False, False,
False,
                   False, False, False, False, False, False, False,
False,
                   False, False, False, False, False, False, False,
False,
                   False, False, False, False, False, False],
       fill_value='?',
            dtype=object), 'params': [{'bootstrap': True, 'max_depth':
5, 'n_estimators': 5}, {'bootstrap': True, 'max_depth': 5,
'n_estimators': 6}, {'bootstrap': True, 'max_depth': 5,
'n_estimators': 7}, {'bootstrap': True, 'max_depth': 5,
'n_estimators': 8}, {'bootstrap': True, 'max_depth': 5,
'n_estimators': 9}, {'bootstrap': True, 'max_depth': 5,
'n_estimators': 10}, {'bootstrap': True, 'max_depth': 5,
'n_estimators': 11}, {'bootstrap': True, 'max_depth': 5,
'n_estimators': 12}, {'bootstrap': True, 'max_depth': 5,
'n_estimators': 13}, {'bootstrap': True, 'max_depth': 5,
'n_estimators': 15}, {'bootstrap': True, 'max_depth': 10,
'n_estimators': 5}, {'bootstrap': True, 'max_depth': 10,
'n_estimators': 6}, {'bootstrap': True, 'max_depth': 10,
'n_estimators': 7}, {'bootstrap': True, 'max_depth': 10,
'n_estimators': 8}, {'bootstrap': True, 'max_depth': 10,
'n_estimators': 9}, {'bootstrap': True, 'max_depth': 10,
```

```
'n_estimators': 10}, {'bootstrap': True, 'max_depth': 10,
'n_estimators': 11}, {'bootstrap': True, 'max_depth': 10,
'n_estimators': 12}, {'bootstrap': True, 'max_depth': 10,
'n_estimators': 13}, {'bootstrap': True, 'max_depth': 10,
'n_estimators': 15}, {'bootstrap': True, 'max_depth': None,
'n_estimators': 5}, {'bootstrap': True, 'max_depth': None,
'n_estimators': 6}, {'bootstrap': True, 'max_depth': None,
'n_estimators': 7}, {'bootstrap': True, 'max_depth': None,
'n_estimators': 8}, {'bootstrap': True, 'max_depth': None,
'n_estimators': 9}, {'bootstrap': True, 'max_depth': None,
'n_estimators': 10}, {'bootstrap': True, 'max_depth': None,
'n_estimators': 11}, {'bootstrap': True, 'max_depth': None,
'n_estimators': 12}, {'bootstrap': True, 'max_depth': None,
'n_estimators': 13}, {'bootstrap': True, 'max_depth': None,
'n_estimators': 15}], 'split0_test_score': array([-0.09272368, -
0.09198444, -0.09176181, -0.09176233, -0.09061588,
       -0.09043593, -0.09079487, -0.09121512, -0.09129171, -
0.08991105,
       -0.06022753, -0.05937399, -0.05826126, -0.05820393, -
0.05723787,
       -0.05685195, -0.05673651, -0.05656256, -0.05627594, -
0.05575841,
       -0.05827262, -0.05774372, -0.05725626, -0.05647457, -
0.05595514,
       -0.05595954, -0.05555016, -0.05548488, -0.05528246, -
0.05480819]), 'split1_test_score': array([-0.08654646, -0.0874426 , -
0.08633015, -0.08645241, -0.08691339,
       -0.0853727 , -0.08522022, -0.08546226, -0.08491317, -
0.08480029,
       -0.05454878, -0.05433029, -0.05392994, -0.05356979, -
0.05303393,
       -0.05240404, -0.05212411, -0.05221685, -0.05244394, -
0.05228607,
       -0.05801765, -0.05662395, -0.05573289, -0.05461212, -
0.05390037,
       -0.05312775, -0.05285985, -0.05269918, -0.05237314, -
0.05190438]), 'split2_test_score': array([-0.09417045, -0.092424  , -
0.09001766, -0.09093398, -0.09147314,
       -0.09131226, -0.09051055, -0.09076096, -0.09012173, -
0.08981693,
       -0.05966751, -0.05836105, -0.05722426, -0.05725958, -
0.05621707,
       -0.05609017, -0.05564203, -0.05530906, -0.05453205, -
0.05447395,
       -0.05703823, -0.05515213, -0.05427   , -0.05444355, -
0.053495  ,
       -0.05381844, -0.0533317 , -0.05294175, -0.0519683 , -
0.05198122]), 'split3_test_score': array([-0.09441054, -0.09425752, -
0.09385849, -0.09415368, -0.09388375,
```

```
       -0.09477645, -0.09462158, -0.0934197 , -0.09345514, -
0.09197693,
       -0.06266005, -0.06108972, -0.06038621, -0.05967708, -
0.05918472,
       -0.05906362, -0.05825498, -0.05761448, -0.05777153, -
0.05714447,
       -0.06302735, -0.06045268, -0.05935988, -0.05867956, -
0.05780121,
       -0.05740021, -0.057086  , -0.05630382, -0.05604287, -
0.05546592]), 'split4_test_score': array([-0.09709069, -0.09767762, -
0.09456889, -0.09379663, -0.09329549,
       -0.094033  , -0.09471869, -0.09537149, -0.09596625, -
0.09528536,
       -0.06034872, -0.05974461, -0.05879388, -0.05788303, -
0.05703895,
       -0.05700111, -0.05700764, -0.0568069 , -0.05672253, -
0.05643809,
       -0.06136106, -0.05930652, -0.05876201, -0.05729795, -
0.05677151,
       -0.05622583, -0.05565321, -0.05542423, -0.0555018 , -
0.05546413]), 'mean_test_score': array([-0.09298837, -0.09275724, -
0.0913074 , -0.09141981, -0.09123633,
       -0.09118607, -0.09117318, -0.0912459 , -0.0911496 , -
0.09035811,
       -0.05949052, -0.05857993, -0.05771911, -0.05731868, -
0.05654251,
       -0.05628218, -0.05595305, -0.05570197, -0.0555492 , -
0.0552202 ,
       -0.05954338, -0.0578558 , -0.05707621, -0.05630155, -
0.05558464,
       -0.05530635, -0.05489619, -0.05457077, -0.05423372, -
0.05392477]), 'std_test_score': array([0.00351617, 0.0033291 ,
0.00295934, 0.00276231, 0.00246487,
       0.00332776, 0.00347799, 0.00332964, 0.00370096, 0.0034141 ,
       0.00267498, 0.00229754, 0.00215245, 0.00203578, 0.00200635,
       0.00217508, 0.00208722, 0.00189312, 0.00187194, 0.00171124,
       0.00226541, 0.00188086, 0.00188514, 0.00161146, 0.00165303,
       0.00158849, 0.00157419, 0.00146451, 0.00170733, 0.00163613]),
'rank_test_score': array([30, 29, 27, 28, 25, 24, 23, 26, 22, 21, 19,
18, 16, 15, 13, 11, 10,
        9,  7,  5, 20, 17, 14, 12,  8,  6,  4,  3,  2,  1],
dtype=int32), 'split0_train_score': array([-0.08304479, -0.08230476, -
0.08217854, -0.08237051, -0.08155251,
       -0.081307  , -0.08165739, -0.08219851, -0.08223493, -
0.08130404,
       -0.03388635, -0.03310244, -0.03268421, -0.03220602, -
0.03211036,
       -0.03180145, -0.03160879, -0.03141826, -0.03121756, -
0.03082376,
```

```
       -0.01640867, -0.01549689, -0.01500791, -0.01463749, -
0.01439206,
       -0.01422619, -0.01409784, -0.01398758, -0.01385201, -
0.01355323]), 'split1_train_score': array([-0.08944226, -0.09013158, -
0.08917861, -0.08920697, -0.08976656,
       -0.08852916, -0.08816067, -0.0882341 , -0.08800265, -
0.08799294,
       -0.0342897 , -0.03373711, -0.03338418, -0.03283797, -
0.03260542,
       -0.03231213, -0.03204847, -0.0320698 , -0.03208229, -
0.03185235,
       -0.01749189, -0.01653415, -0.015948  , -0.01535362, -
0.01496411,
       -0.01469613, -0.01452614, -0.01435298, -0.0141305 , -
0.01383067]), 'split2_train_score': array([-0.08935183, -0.08772222, -
0.08573142, -0.08611284, -0.08631376,
       -0.08605939, -0.08497687, -0.08542272, -0.08500597, -
0.08484178,
       -0.03505894, -0.03419949, -0.03355724, -0.03339151, -
0.03312522,
       -0.03307476, -0.03283073, -0.03250406, -0.03217336, -
0.03201049,
       -0.01694124, -0.01609974, -0.01560591, -0.01521575, -
0.01498722,
       -0.01465155, -0.01431561, -0.01404313, -0.01381367, -
0.01375567]), 'split3_train_score': array([-0.08606277, -0.08611396, -
0.08628251, -0.08639504, -0.08653794,
       -0.08687287, -0.08705625, -0.08647386, -0.08649771, -
0.08527242,
       -0.03470119, -0.03409366, -0.03358556, -0.03358749, -
0.03334057,
       -0.03297889, -0.03279741, -0.03284937, -0.03252394, -
0.03211157,
       -0.01660632, -0.01580937, -0.01509841, -0.01472863, -
0.01439395,
       -0.01411467, -0.01379915, -0.01366799, -0.01347727, -
0.01318033]), 'split4_train_score': array([-0.0846002 , -0.08478146, -
0.08268441, -0.08269368, -0.08206906,
       -0.08254294, -0.08288998, -0.08344475, -0.08381429, -
0.0835092 ,
       -0.03532752, -0.03478225, -0.03416539, -0.03381498, -
0.0335362 ,
       -0.03313796, -0.03283612, -0.03281581, -0.03262999, -
0.03215823,
       -0.01721069, -0.0161272 , -0.01566565, -0.01539895, -
0.01503696,
       -0.01464844, -0.01438296, -0.01416552, -0.01396699, -
0.01355586]), 'mean_train_score': array([-0.08650037, -0.0862108 , -
0.0852111 , -0.08535581, -0.08524796,
```

```
       -0.08506227, -0.08494823, -0.08515479, -0.08511111, -
0.08458408,
       -0.03465274, -0.03398299, -0.03347531, -0.03316759, -
0.03294355,
       -0.03266104, -0.0324243 , -0.03233146, -0.03212543, -
0.03179128,
       -0.01693176, -0.01601347, -0.01546517, -0.01506689, -
0.01475486,
       -0.0144674 , -0.01422434, -0.01404344, -0.01384809, -
0.01357515]), 'std_train_score': array([0.00255063, 0.00264448,
0.00255892, 0.00254886, 0.00306527,
       0.00271075, 0.00244256, 0.00214279, 0.0020124 , 0.00219429,
       0.00051804, 0.00055385, 0.00047523, 0.00057945, 0.00051969,
       0.00052159, 0.00050602, 0.00053573, 0.00049835, 0.000495  ,
       0.00039273, 0.00034641, 0.00035685, 0.00032045, 0.00029639,
       0.0002456 , 0.00025352, 0.00022574, 0.00021569, 0.00022566])}
```

```
# Let's see the best parameters found so far
grid.best_params_
```

```
{'bootstrap': True, 'max_depth': None, 'n_estimators': 15}
```

```
# And its best value, should ignore the minus because it's
neg_mean_squared_error
grid.best_score_
```

```
-0.053924765728631106
```

Testing

We should no longer do anything like improving the model. It's illegal! since X_test is the final final test set.

```
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.metrics import mean_absolute_error
import numpy as np

yhat = grid.predict(X_test)
print("PREDICTED DATA: ", np.exp(yhat)[:10])
print("ACTUAL DATA: ", list(y_test[:10]))

# bringing back yhat to the exponential
pred_y = np.exp(yhat)

# Doing just MSE is not okay, since our predictions are huge numbers
and there are outliers, the mse is not accurate
print("MSE: ", mean_squared_error(y_test, pred_y))

# RMSE performs better, but still not so good
print("RMSE: ", np.sqrt(mean_squared_error(y_test, pred_y)))
```

```
# So from this it is clear that squaring is really bad idea, since we
get huge numbers as per selling_price
# Therefore better to proceed with other metrics like below:

# MAE
print("MAE: ", mean_absolute_error(y_test, pred_y))
# Performance of MAE is good - 71856. It means we can use this model
for deploying.

# Percentage error seems fine, we got 15% error
percentage_error = np.abs((pred_y - y_test) / y_test) * 100
mean_percentage_error = np.mean(percentage_error)
print("Mean Percentage Error:", mean_percentage_error)

# We can also do cosine similarity to check whether the predictions is
fine or not
# Reshape the vectors to be 2D arrays for cosine_similarity function
y_test_reshaped = y_test.values.reshape(1, -1)
pred_y_reshaped = pred_y.reshape(1, -1)

# Cosine similarity predictions
cos_sim = cosine_similarity(y_test_reshaped, pred_y_reshaped)
print("Cosine Similarity:", cos_sim[0][0])
```

```
PREDICTED DATA:  [222821.30812028 927371.16481098 297129.76266185
509231.20187591
 680801.3687177  195757.8543135  300516.29437276 480859.86758875
 339796.91003707 394981.79997677]
ACTUAL DATA:  [225000, 900000, 320000, 650000, 520000, 170000, 280000,
500000, 170000, 335000]
MSE:  53802638404.5708
RMSE:  231953.95751004294
MAE:  71856.95767761378
Mean Percentage Error: 15.833384145330337
Cosine Similarity: 0.9795473733102381
```

Analysis: Feature Importance

Understanding why is **key** to every business, not how low MSE we got. Extracting which feature is important for prediction can help us interpret the results. There are several ways: algorithm, permutation, and shap. Note that these techniques can be mostly applied to most algorithms.

Most of the time, we just apply all, and check the consistency.

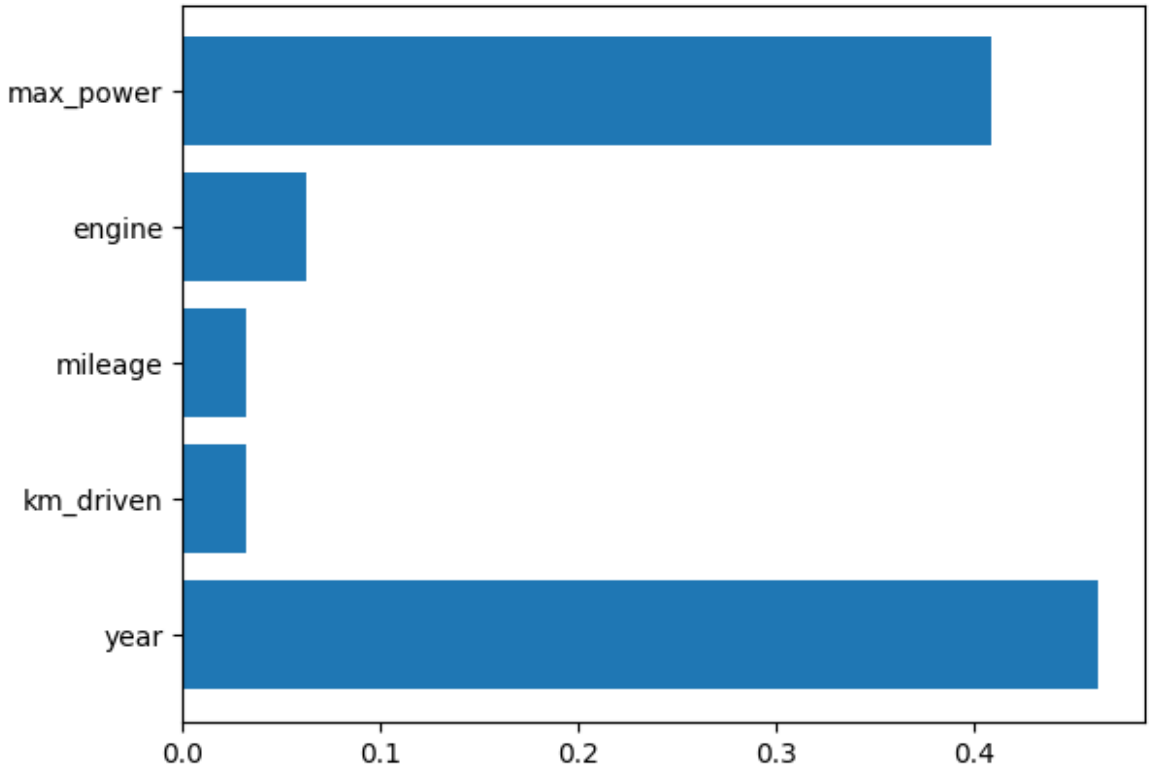
```
# taking best estimator so far
rf = grid.best_estimator_

# Random Forest has great attribute called feature_importances_ just
```

```
to see which feature has impact most
rf.feature_importances_
```

```
array([0.463033  , 0.03242367, 0.03242622, 0.06300381, 0.4091133 ])
```

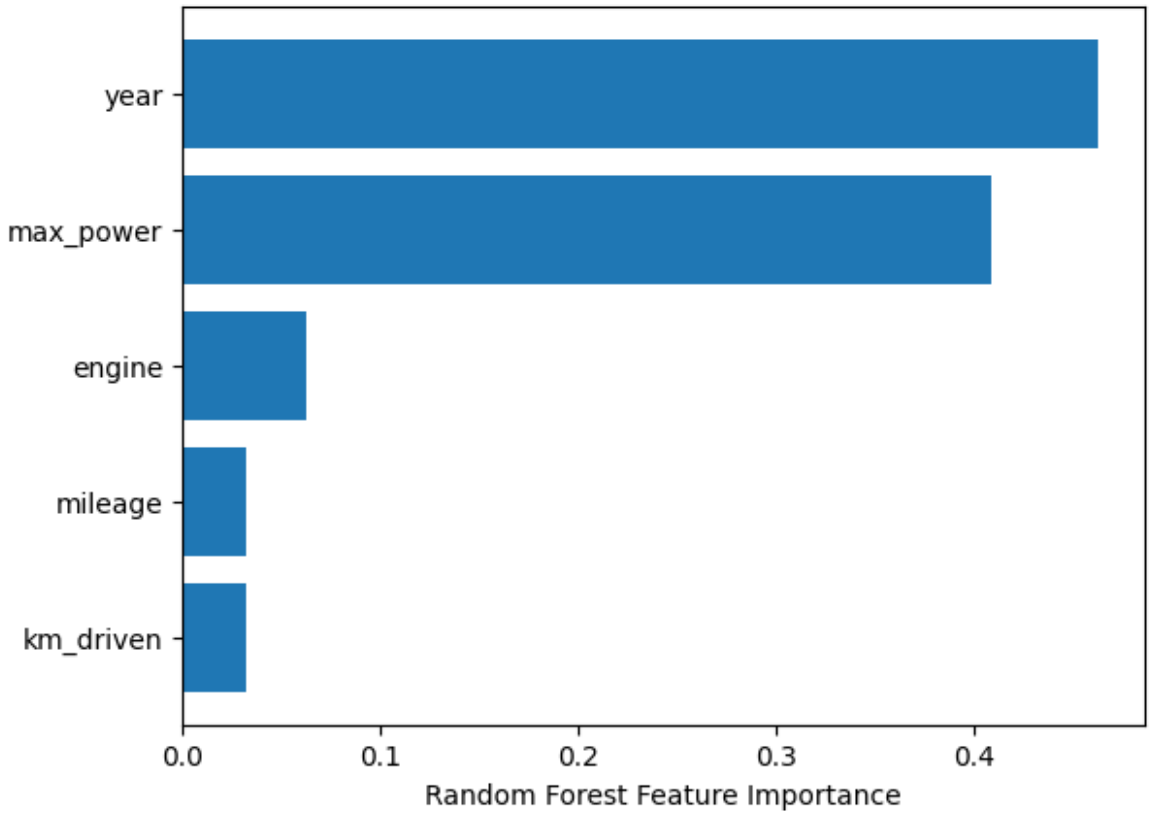
```
plt.barh(X_train.columns, rf.feature_importances_)
```

```
<BarContainer object of 5 artists>
```




```
sorted_idx = rf.feature_importances_.argsort()
plt.barh(X.columns[sorted_idx], rf.feature_importances_[sorted_idx])
plt.xlabel("Random Forest Feature Importance")
```

```
Text(0.5, 0, 'Random Forest Feature Importance')
```




```
# most important features in percentages as dataframe
importances = rf.feature_importances_
importances_df = pd.DataFrame({'Feature': X_train.columns,
'Importance': importances})
importances_df = importances_df.sort_values(by='Importance',
ascending=False)
importances_df
```


```
{"summary":"{\n  \"name\": \"importances_df\",\n  \"rows\": 5,\n
\"fields\": [\n    {\n      \"column\": \"Feature\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 5,\n        \"samples\": [\n
\"max_power\",\n          \"km_driven\",\n          \"engine\"\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"Importance\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.21670572349829922,\n        \"min\": 0.0324236733924615,\n
\"max\": 0.4630329953032097,\n        \"num_unique_values\": 5,\n
\"samples\": [\n          0.40911330120464673,\n
0.0324236733924615,\n          0.06300381008059737\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe","variable_name":"importances_df"}
```

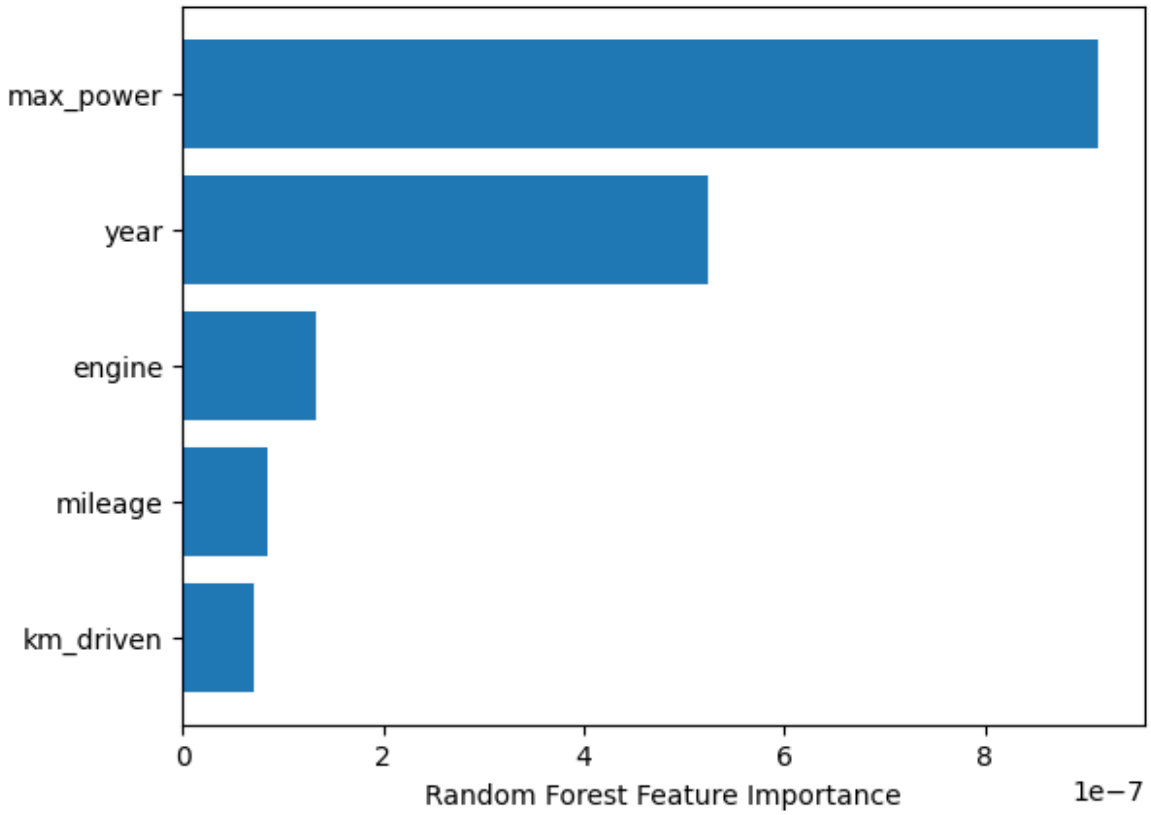
Permutation way

This method will randomly shuffle each feature and compute the change in the model's performance. The features which impact the performance the most are the most important one.

Note: The permutation based importance is computationally expensive. The permutation based method can have problem with highly-correlated features, it can report them as unimportant.

```
from sklearn.inspection import permutation_importance

# shuffling each feature and computing the change in the model's
performance
perm_importance = permutation_importance(rf, X_test, y_test)

sorted_idx = perm_importance.importances_mean.argsort()
plt.barh(X.columns[sorted_idx],
perm_importance.importances_mean[sorted_idx])
plt.xlabel("Random Forest Feature Importance")
```

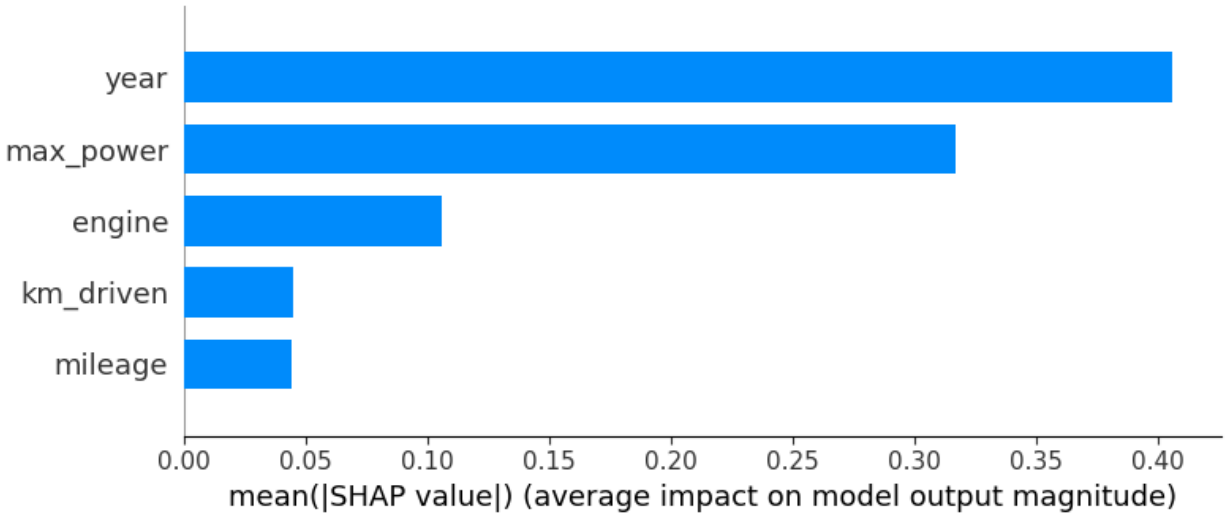
```
Text(0.5, 0, 'Random Forest Feature Importance')
```



SHAP

The SHAP (SHapley Additive exPlanations) interpretation can be used (it is model-agnostic) to compute the feature importances. It is using the Shapley values from game theory to estimate the how does each feature contribute to the prediction.

```
import shap

# Tree Explainer is to explain the output of tree-based machine
learning models
explainer = shap.TreeExplainer(rf)

# compute the feature importances
shap_values = explainer.shap_values(X_test)

#shap provides plot
shap.summary_plot(shap_values, X_test, plot_type="bar", feature_names
= X.columns)
```



Inference

To provide inference service or deploy, it's best to save the model for latter use.

```
import joblib

# save the model to disk
filename = 'rf_carPrice_5_feature_3rd_attempt.pkl'
joblib.dump(grid, filename)
```

```
['rf_carPrice_5_feature_3rd_attempt.pkl']
```

```
sample_df = X_test.reset_index(drop=True)
sample_df = sample_df.loc[17]
sample_df
```

```
year          2010.000000
km_driven       -0.423649
mileage         -2.191394
engine           0.970176
max_power       -0.049889
Name: 17, dtype: float64
```

```
sample_y = y_test
sample_y = sample_y.reset_index(drop=True)
sample_y = sample_y.loc[17]
sample_y
```

```
235000
```

```
fake_data = np.array([[2010, -0.42, -2.19, 0.97, -0.04]])
np.exp(rf.predict(fake_data))
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:465:
UserWarning: X does not have valid feature names, but
RandomForestRegressor was fitted with feature names
  warnings.warn(
```

```
array([223072.62939516])
```

The inference is kind a fine, I believe

Task 2: Report - Conclusion

1. Which features are important? Which are not? Why?

   Based on the analysis, the most crucial features for predicting car prices are (top-5): year, max_power, engine, mileage, and km_driven by 46%, 39%, 5%, 2%, and 2% respectively. Personally, I think the same way that those features are essential when buying a used-before car, but I also think the brand makes sense at all in price consideration. And as well, the not-so-important features which are seats, fuel, transmission, and owner are truly unimportant. This suggests that while these attributes contribute to overall pricing, their influence is less significant compared to more directly relevant features.

2. Which algorithm performs well? Which does not? Why?

   I have tried several algorithms namely LinearRegression, naive-bayes (GaussianNB), knn, RandomForest, SVR, GradientBoosting, XGBoost (the best so far), and Ridge, and noticed that the best top-3 algorithms (order matters): XGBoost, RandomForest, and GradientBoosting. Random Forest was chosen for further analysis due to its balance between performance and complexity, making it a practical choice for model development (it has almost the same performance as XGB). The worst performance was monitored by SVR (Support Vector Regression - type of SVM). I know that SVR is sensible for scalings - maybe this is the reason. Anyways its

performance was - mean: -0.6635002241153819 and std: 0.025864161834637186 - which is not as bad as it might seem.

3. What I liked about the dataset?

   It contains a lot of features that actually might matter to the price as the overall product is based on cultural dependence - bias. For example, in my country - Uzbekistan, people tend to use more petrol-based cars, and they are more expensive than diesel-based cars because of the rate of interest. This interest is based on the climate of Uzbekistan - Uzbekistan is located between two deserts, and the usage of diesel-based cars is a bit dangerous because of extremely hot weather. So, I believe the features themselves can't be generalized over all climates and cultures - in some places unimportant features would benefit.

[08/20/2024] 2nd attempt - I have taken 5 features, and seems like the results are not very different. I will try to deploy both models (let user choose which model to use).

[08/21/2024] 3rd attempt - just handler outliers, to check will it make sense