**Car\_Dheko Project Documentation**

# **Data\_extraction - Converting unstructured data into a structured data:**

Note - Consider opening this file in google doc for better visibility

import pandas as pd

import ast

# Load the CSV file

file\_path = r"C:\Users\sandy\Downloads\Hyderabad\_output\_file.csv" # Adjust this if necessary

df = pd.read\_csv(file\_path)

# Function to par'se the JSON-like strings into actual dictionaries

def parse\_json\_column(column):

return column.apply(lambda x: ast.literal\_eval(x))

# Parsing the columns

df['new\_car\_detail'] = parse\_json\_column(df['new\_car\_detail'])

df['new\_car\_overview'] = parse\_json\_column(df['new\_car\_overview'])

df['new\_car\_feature'] = parse\_json\_column(df['new\_car\_feature'])

df['new\_car\_specs'] = parse\_json\_column(df['new\_car\_specs'])

# Extracting details from 'new\_car\_detail'

df\_details = pd.json\_normalize(df['new\_car\_detail'])

# Extracting overview from 'new\_car\_overview'

df\_overview = pd.json\_normalize(df['new\_car\_overview'].apply(lambda x: x.get('top', {})))

# Extracting features from 'new\_car\_feature'

df\_features = pd.json\_normalize(df['new\_car\_feature'].apply(lambda x: x.get('top', {})))

# Extracting specifications from 'new\_car\_specs'

df\_specs = pd.json\_normalize(df['new\_car\_specs'].apply(lambda x: x.get('top', {})))

# Extracting additional data from 'new\_car\_overview', 'new\_car\_feature', and 'new\_car\_specs' if present

df\_overview\_data = pd.json\_normalize(df['new\_car\_overview'].apply(lambda x: x.get('data', {})))

df\_features\_data = pd.json\_normalize(df['new\_car\_feature'].apply(lambda x: x.get('data', {})))

df\_specs\_data = pd.json\_normalize(df['new\_car\_specs'].apply(lambda x: x.get('data', {})))

# Combine all extracted data into a single dataframe

df\_final = pd.concat([df\_details, df\_overview, df\_features, df\_specs,

df\_overview\_data, df\_features\_data, df\_specs\_data,

df['car\_links']], axis=1)

# Save the structured data to your specified path

output\_file = 'C:/Users/sandy/Desktop/Project\_realected\_practice/Car\_deako/Structed\_Hyderabad\_cars.csv'

df\_final.to\_csv(output\_file, index=False)

print(f"Structured data saved to {output\_file}")

## Step 1 - Extracting Column Wise Data:

### **Reading File and converting json-string into a Dictionary Format:**

In this the raw data is being read and the function is created to convert all the Json-like Strings into a Dictionary it is applied to all the columns that need to be converted.

import pandas as pd

import ast

# Load the CSV file

file\_path = r"C:\Users\sandy\Downloads\Hyderabad\_output\_file.csv" # Adjust this if necessary

df = pd.read\_csv(file\_path)

# Function to parse the JSON-like strings into actual dictionaries

def parse\_json\_column(column):

return column.apply(lambda x: ast.literal\_eval(x))

### **Extracting the data inside the dictionary and putting it as a separate columns:**

Since the column New\_car\_detail is do have any complication inside it each data was converted into a each column but for other three column the structure of the data present is so different so first I have extracted the New\_car\_detail only and in other three column I only extracted the Key - Top and Key Data if present and Put this two into a separate dictionary format for further extraction and making every extracted column will a separate data frame.

# Parsing the columns

df['new\_car\_detail'] = parse\_json\_column(df['new\_car\_detail'])

df['new\_car\_overview'] = parse\_json\_column(df['new\_car\_overview'])

df['new\_car\_feature'] = parse\_json\_column(df['new\_car\_feature'])

df['new\_car\_specs'] = parse\_json\_column(df['new\_car\_specs'])

# Extracting details from 'new\_car\_detail'

df\_details = pd.json\_normalize(df['new\_car\_detail'])

# Extracting overview from 'new\_car\_overview'

df\_overview = pd.json\_normalize(df['new\_car\_overview'].apply(lambda x: x.get('top', {})))

# Extracting features from 'new\_car\_feature'

df\_features = pd.json\_normalize(df['new\_car\_feature'].apply(lambda x: x.get('top', {})))

# Extracting specifications from 'new\_car\_specs'

df\_specs = pd.json\_normalize(df['new\_car\_specs'].apply(lambda x: x.get('top', {})))

# Extracting additional data from 'new\_car\_overview', 'new\_car\_feature', and 'new\_car\_specs' if present

df\_overview\_data = pd.json\_normalize(df['new\_car\_overview'].apply(lambda x: x.get('data', {})))

df\_features\_data = pd.json\_normalize(df['new\_car\_feature'].apply(lambda x: x.get('data', {})))

df\_specs\_data = pd.json\_normalize(df['new\_car\_specs'].apply(lambda x: x.get('data', {})))

### **Finally concatenating all the dataframe:**

Finally concatenate all the data frames together and save that dataframe as a csv file for further extraction of the data.

df\_final = pd.concat([df\_details, df\_overview, df\_features, df\_specs,

df\_overview\_data, df\_features\_data, df\_specs\_data,

df['car\_links']], axis=1)

# Save the structured data to your specified path

output\_file = 'C:/Users/sandy/Desktop/Project\_realected\_practice/Car\_deako/Structed\_Hyderabad\_cars.csv'

df\_final.to\_csv(output\_file, index=False)

print(f"Structured data saved to {output\_file}")

## Step 2 - Extracting the column 18 to 52 (Top and Data Dictionary):

import pandas as pd

import ast

def process\_cell(cell):

if pd.isna(cell):

return {}

try:

data = ast.literal\_eval(cell)

if isinstance(data, dict):

return data

except (ValueError, SyntaxError):

return {}

return {}

def transform\_data(input\_file, output\_file):

# Load the CSV file

data = pd.read\_csv(input\_file)

# Apply the function to relevant columns to convert JSON-like strings into dictionaries

json\_columns = data.columns[18:52]

for col in json\_columns:

data[col] = data[col].apply(process\_cell)

# Initialize a list to hold the structured data rows

structured\_data\_rows = []

# Process each row to extract and organize the structured data

for index, row in data.iterrows():

new\_row = {}

specification\_count = 1

for col in json\_columns:

cell\_data = row[col]

# Type 1 and Type 4: Key-Value pairs

if 'key' in cell\_data:

new\_row[cell\_data['key']] = cell\_data['value']

# Type 2: Only Value

elif 'value' in cell\_data and 'key' not in cell\_data:

new\_row[f'specification{specification\_count}'] = cell\_data['value']

specification\_count += 1

# Type 3: Heading with list of values

elif 'heading' in cell\_data and 'list' in cell\_data:

values\_list = [item['value'] for item in cell\_data['list']]

new\_row[cell\_data['heading']] = values\_list

# Append the new row dictionary to the list of structured data rows

structured\_data\_rows.append(new\_row)

# Convert the list of dictionaries to a DataFrame

structured\_data = pd.DataFrame(structured\_data\_rows)

# Merge the structured data back with the original data (excluding JSON columns)

final\_data = pd.concat([data.drop(columns=json\_columns), structured\_data], axis=1)

# Save the final structured data to a new CSV file

final\_data.to\_csv(output\_file, index=False)

# Usage

input\_file = r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\Structed\_Hyderabad\_cars.csv'

output\_file = r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\structured\_data\_Hyderabad.csv'

transform\_data(input\_file, output\_file)

### JSON-like strings into dictionaries

If the data is empty it will return the empty dictionary or else if there is a value it will first convert it into a Dictionary and in that process if it catches any error means it will return an empty dictionary in that place.

def process\_cell(cell):

if pd.isna(cell):

return {}

try:

data = ast.literal\_eval(cell)

if isinstance(data, dict):

return data

except (ValueError, SyntaxError):

return {}

return {}

### Converting the Json-string into a Dictionary Cell wise:

In this code it will take the particular column and convert it into a dictionary cell wise for easy handling and accessing with a Key

def transform\_data(input\_file, output\_file):

# Load the CSV file

data = pd.read\_csv(input\_file)

# Apply the function to relevant columns to convert JSON-like strings into dictionaries

json\_columns = data.columns[18:52]

for col in json\_columns:

data[col] = data[col].apply(process\_cell)

### Handling and extracting the data inside the dictionary based on its structure:

In the there are totally three different format are present first is Key - Value pair , second is only value and third one is the Key with a list of value What it will do means 1st - it will extract the Key content and the value and inserted it into the empty dictionary and 2nd the Specification column the key will be specification and based on how many specification the column name will represent as per that and last one is the heading will be the key and the all the value inside it will be inside the List and after collecting all this it will append into the List.

for index, row in data.iterrows():

new\_row = {}

specification\_count = 1

for col in json\_columns:

cell\_data = row[col]

# Type 1 and Type 4: Key-Value pairs

if 'key' in cell\_data:

new\_row[cell\_data['key']] = cell\_data['value']

# Type 2: Only Value

elif 'value' in cell\_data and 'key' not in cell\_data:

new\_row[f'specification{specification\_count}'] = cell\_data['value']

specification\_count += 1

# Type 3: Heading with list of values

elif 'heading' in cell\_data and 'list' in cell\_data:

values\_list = [item['value'] for item in cell\_data['list']]

new\_row[cell\_data['heading']] = values\_list

# Append the new row dictionary to the list of structured data rows

structured\_data\_rows.append(new\_row)

### Converting the Data into the Data Frame:

Finally all the extracted data will be converted into a data frame and saved as a new csv file which will be used for next level extraction.

# Convert the list of dictionaries to a DataFrame

structured\_data = pd.DataFrame(structured\_data\_rows)

# Merge the structured data back with the original data (excluding JSON columns)

final\_data = pd.concat([data.drop(columns=json\_columns), structured\_data], axis=1)

# Save the final structured data to a new CSV file

final\_data.to\_csv(output\_file, index=False)

# Usage

input\_file = r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\Structed\_Hyderabad\_cars.csv'

output\_file = r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\structured\_data\_Hyderabad.csv'

transform\_data(input\_file, output\_file)

## Step 3 - Extracting only the required data from the required column:

import pandas as pd

# Load the structured CSV file

file\_path = r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\structured\_data\_Hyderabad.csv'

data = pd.read\_csv(file\_path)

# List of columns that contain lists of values

list\_columns = [

'Interior',

'Safety',

'Miscellaneous',

]

# Function to split list values into separate columns

def split\_list\_into\_columns(row, col\_name):

values = row[col\_name]

if pd.isna(values):

return {}

values = values.strip('[]').split(',')

values = [v.strip().strip("'") for v in values]

split\_columns = {f'{col\_name}{i+1}': values[i] for i in range(len(values))}

return split\_columns

# Apply the function to each row and update the dataframe

for col in list\_columns:

new\_columns = data.apply(lambda row: pd.Series(split\_list\_into\_columns(row, col)), axis=1)

data = pd.concat([data, new\_columns], axis=1)

data.drop(columns=[col], inplace=True)

# Save the updated dataframe to a new CSV file

output\_file\_path = r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\Hyderabad.csv'

data.to\_csv(output\_file\_path, index=False)

output\_file\_path

### Load the file and mentioning the column name which we want to extract:

In this the updated csv file will be loaded and inside the new list i have mentioned the column I need to extract.

import pandas as pd

# Load the structured CSV file

file\_path = r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\structured\_data\_Hyderabad.csv'

data = pd.read\_csv(file\_path)

# List of columns that contain lists of values

list\_columns = [

'Interior',

'Safety',

'Miscellaneous',

]

### Function to Split List Values into Separate Columns:

It retrieves the value of the column for the row. If the value is NaN (missing), it returns an empty dictionary

And after that the Strips function strips square brackets ([]) and splits the string by commas, then further strips whitespace and single quotes from each list element.Creates a dictionary with new column names (e.g., Interior1, Interior2) and their corresponding values from the list.

def split\_list\_into\_columns(row, col\_name):

values = row[col\_name]

if pd.isna(values):

return {}

values = values.strip('[]').split(',')

values = [v.strip().strip("'") for v in values]

split\_columns = {f'{col\_name}{i+1}': values[i] for i in range(len(values))}

return split\_columns

### Applying the Function to Each Row for the Specified Columns:

For each row, the split\_list\_into\_columns function generates a new Series (a Pandas data structure) containing the split columns and the new columns are concatenated to the original DataFrame data

the original column containing the list is dropped from the DataFrame after that this data frame will be

Saved as a new csv file.

for col in list\_columns:

new\_columns = data.apply(lambda row: pd.Series(split\_list\_into\_columns(row, col)), axis=1)

data = pd.concat([data, new\_columns], axis=1)

data.drop(columns=[col], inplace=True)

# Save the updated dataframe to a new CSV file

output\_file\_path = r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\Hyderabad.csv'

data.to\_csv(output\_file\_path, index=False)

output\_file\_path

By doing all these steps the unstructured data is converted into a semi-structured format as csv format.

# Concatenating the Structured File ,Data Cleaning and Filtering only the Required Column:

## Step 1 - Concatenating the Structured File :

Firstly, in the files list all the data location for all six file has been present and in addition to that there is also the city name is also mentioned along with the file location.And the for loop initialize what it will do is first read the csv file and add a column city in that it give a respective city name and append it into the new empty list all the dataframe are concatenated together and saved it into a new file.

import pandas as pd

# List of file paths and corresponding city names

files = [

(r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\File\_to\_merge\Bangalore.csv', 'Bangalore'),

(r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\File\_to\_merge\Chennai.csv', 'Chennai'),

(r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\File\_to\_merge\Delhi.csv', 'Delhi'),

(r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\Hyderabad.csv', 'Hyderabad'),

(r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\File\_to\_merge\Jaipur.csv', 'Jaipur'),

(r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\File\_to\_merge\Kolkata.csv', 'Kolkata')

]

# Initialize an empty list to store DataFrames

dfs = []

# Iterate through files, read them, add city column, and append to list

for file, city in files:

df = pd.read\_csv(file)

df['City'] = city

dfs.append(df)

# Concatenate all DataFrames into one

combined\_df = pd.concat(dfs, ignore\_index=True)

# Save the combined DataFrame to a new CSV file

combined\_df.to\_csv(r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\File\_to\_merge\cars\_data.csv', index=False)

## Step 2 - Deciding what are the column is necessary:

### Dropping unnecessary columns based on the Domain Knowledge:

After concatenation there are about 8300 rows and nearly 129 columns cleaning all these column is unnecessary and based on the use case (What purpose I am going to use this file after cleaned) I have decided what are the required columns for EDA and for ML training for Price prediction I have narrowed a required column based on the Domain Knowledge and Use case. So finally I have decided to use 33 columns further.

New\_car\_data=Car\_data[['City','it', 'ft', 'bt', 'km', 'transmission', 'ownerNo', 'owner', 'oem', 'model', 'modelYear', 'centralVariantId', 'variantName', 'price', 'priceActual', 'priceSaving', 'priceFixedText', 'Registration Year', 'Insurance Validity', 'Fuel Type', 'Seats', 'Kms Driven', 'RTO', 'Ownership', 'Engine Displacement', 'Year of Manufacture', 'Mileage', 'Engine', 'Max Power', 'Torque', 'Interior2', 'Safety1', 'Miscellaneous1']]

New\_car\_data.head()

So after analyzing the File I have decided these are the columns needed for further process.

### Dropping the Duplicate columns:

After the inspection I have find that the same value is repeated in other columns but with different column name for example Km and Kms Driven have a same value in it so I dropped , similar to that some column don’t have a variety of value like ‘it’ column it have only one value 0 so it won’t be useful in any case so i dropped it.

New\_car\_data.drop(['centralVariantId','it','Registration Year','Kms Driven','owner','Ownership','priceActual','priceSaving','priceFixedText','RTO'],axis=1,inplace=True)

New\_car\_data.head()

## Step 3 - Renaming the Columns for better understanding :

This has around 23 columns and some columns are not abbreviated and some are not given a meaning so I have changed those kinds of columns in a meaningful way.

New\_car\_data.rename(columns = {'it':'Ignition\_type.','ft':'Fuel\_type','bt':'Body\_type','km':'kilometers\_Driven','transmission':'Transmission\_Type','ownerNo':'Previous\_owners','oem':'Manufactured\_By','model':'Model','modelYear':'Model\_Year','Interior2':'Interior\_system','Safety1':'Safety\_features','Miscellaneous1':'Gear\_Types'}, inplace =True)

New\_car\_data.head()

## Step 4 - Standardising Data Formats:

### Removing the Unnecessary units and other stuff in a required columns :

Few columns need to change the datatypes of it to use it for ML operation and also for gathering insights out of it but before doing that the data is not in correct format to convert so cleaning the unwanted things like removing CC from Engine Displacement column, (,) from the Kilometers Driven column, Nm and nm in Torque columns and ‘kmpl’ in the Mileage column and (₹) from the price column and there are several other things are need to be cleaned.

New\_car\_data['kilometers\_Driven']=New\_car\_data['kilometers\_Driven'].str.replace(',',"")

New\_car\_data['Engine']=New\_car\_data['Engine'].str.replace('CC',"")

New\_car\_data['Mileage']=New\_car\_data['Mileage'].str.replace('kmpl',"")

New\_car\_data['Torque']=New\_car\_data['Torque'].str.replace('Nm',"")

New\_car\_data['Price']=New\_car\_data['Price'].str.replace('₹',"")

New\_car\_data.head()

It will replace the given thing 1st(CC) with the 2nd(“”).

### Conversion or Replace in Price columns:

#### Understanding of the Price Column

In these columns there are Three types for a range of values are presented Thousands,Lakhs, and also Crore are present This needs to be addressed before simply removing the Lakh, Crore. So I have checked how much data is present in these three ranges. I found that most of the data is presented in between Lakh and few is presented in Thousand and Crore.

For Lakh

filtered\_data1 = New\_car\_data[New\_car\_data['price'].str.contains('Lakh')]

filtered\_data1.count()

For Crore

filtered\_data = New\_car\_data[New\_car\_data['price'].str.contains('Crore')]

filtered\_data.count()

#### Split the values and its unit for better handling

I have split each row value based on the space and it is converted into a list which has a value in the 1 index and Unit in the second index for thousand three is no unit mentioned so it will remain empty.

0 [, 4, Lakh]

1 [, 8.11, Lakh]

2 [, 5.85, Lakh]

3 [, 4.62, Lakh]

4 [, 7.90, Lakh]

...

8364 [, 5.10, Lakh]

8365 [, 1.80, Lakh]

8366 [, 5.50, Lakh]

8367 [, 1.40, Lakh]

8368 [, 5, Lakh]

#### Conversion of value into a correct range

Based on the unit present in the second index it will multiple the value to make a correct conversion.

import pandas as pd

# Function to convert value to integer

def convert\_to\_int(row):

value = row[1].replace(',', '')

if row[2] == 'Lakh':

return int(float(value) \* 1e5)

elif row[2] == 'Crore':

return int(float(value) \* 1e7)

elif row[2] == '': # For values like 75,000 without a unit

return int(value)

else:

return int(value) # Default conversion

# Apply the conversion to each row in 'price' column

New\_car\_data['Price'] = New\_car\_data['price'].apply(convert\_to\_int)

So after this all the values are converted properly.

|  | **Price** | **price** |
| --- | --- | --- |
| 0 | 400000 | [, 4, Lakh] |
| 1 | 811000 | [, 8.11, Lakh] |
| 2 | 585000 | [, 5.85, Lakh] |
| 3 | 462000 | [, 4.62, Lakh] |
| 4 | 790000 | [, 7.90, Lakh] |
| ... | ... | ... |
| 8364 | 509999 | [, 5.10, Lakh] |
| 8365 | 180000 | [, 1.80, Lakh] |
| 8366 | 550000 | [, 5.50, Lakh] |
| 8367 | 140000 | [, 1.40, Lakh] |
| 8368 | 500000 | [, 5, Lakh] |

After done and checked whether all the values are converted correctly or not I have droped the old column.

#droping the old price column

New\_car\_data.drop('price',axis=1,inplace=True)

New\_car\_data.head()

## Step 5 - Type Conversion:

Kilometers\_driven ---> convert it to int

Model\_Year ----> convert it to Year format

Seats ---> convert it to int format

Mileage ----> Float

Engine CC,Max\_Power,Torque ---> Converting all this to a Float

After structured out what are the column need to be convert I have started the conversion process

Float=['Mileage','Engine','Max\_Power','Torque','Seats']

for j in zip(Float):

New\_car\_data[j]=New\_car\_data[j].astype('float64')

Similarly I have done the same for int conversion as well.

## Step - 6 Handling the Missing value :

#### Handling categorical Missing value :

##### Technique 1:

Missing value containing columns ----->

----> Insurance Validity(String),Interior\_system(String),Safety\_features(String),Gear\_Types(String)

Checking how many missing values are there

New\_car\_data['Interior\_system'].isna().sum()

First I have grouped the Interior\_system based on the Model why have done this means the model have a same kind of interior system even though it will change based on the varient here it is the only option to do so I have went with this option and grouped the Interior\_system based on the Model and taken which model have which interior system.

interior\_grouped = New\_car\_data.groupby(['Interior\_system'])['Model']

After done this I have filled the NA values with this values based on which model have the Na based on that the particular value will be filled.

Similarly I have done the same for all othe column which have a large missing value and it was a categorical data.

##### Technique 2:

Missing only Few values:

For this kind of column what I have done is I have filtered the data based on the null value in missing and the model column needs to be present with some value based on that I have filtered the missing values.

filtered\_data = New\_car\_data[New\_car\_data['Model'].notna() & New\_car\_data['Gear\_Types'].isna()]

After that I have checked the Model and what the gear type that the particular model has based on the majority I have filled the value. If all the columns have a Null value I have filled the value with the original data that is present in the internet.

A=New\_car\_data[New\_car\_data['Model'] == "Ambassador"][['Gear\_Types']]

A.head()

The Values are:

| **Gear\_Types** |  |
| --- | --- |
| 7063 | 5 Speed |
| 7398 | 5 Speed |
| 7745 | NaN |

So I have filled the missing value with the 5 speed

For complete NAN values;

A=New\_car\_data[New\_car\_data['Model'] == "Maruti Estilo"][['Gear\_Types']]

A.head()

The Values are

| **Gear\_Types** |  |
| --- | --- |
| 1175 | NaN |
| 7608 | NaN |
| 7669 | NaN |

All the values are NAN so I have search In online for the particular model it was 5 speed so I have filled with that value

New\_car\_data.loc[(New\_car\_data['Gear\_Types'].isna()) & (New\_car\_data['Model'] == "Maruti Estilo"), 'Gear\_Types'] = '5 Speed'

New\_car\_data.loc[(New\_car\_data['Gear\_Types'].isna()) & (New\_car\_data['Model'] == "Ambassador"), 'Gear\_Types'] = '5 Speed'

New\_car\_data.loc[(New\_car\_data['Gear\_Types'].isna()) & (New\_car\_data['Model'] == "Ford Endeavour"), 'Gear\_Types'] = '6 Speed'

So the categorical missing value is filled with these two techniques.

#### Handling the Numerical Missing data:

Since there is a possibility of outliers if I take mean means it will give a wrong average values so what is done is I have taken a median for the column so that the value won’t be affected with the extreme values due to the outliers.But instead of taking the general median value I have taken the model specific median so while filling the NAN value it will fill the specific average to the specific Models so it will be able to match to the original value.

import pandas as pd

# Group by 'Model' and calculate the median for 'Torque\_Nm' and 'Max Power'

median\_values = New\_car\_data.groupby('Model')[['Torque\_Nm', 'Max Power']].median()

# Fill NaN values in 'Torque\_Nm' and 'Max Power' with the median values

New\_car\_data['Torque\_Nm'] = New\_car\_data.apply(

lambda row: median\_values.loc[row['Model'], 'Torque\_Nm'] if pd.isna(row['Torque\_Nm']) else row['Torque\_Nm'], axis=1

)

New\_car\_data['Max Power'] = New\_car\_data.apply(

lambda row: median\_values.loc[row['Model'], 'Max Power'] if pd.isna(row['Max Power']) else row['Max Power'], axis=1

)

# Display the updated DataFrame

New\_car\_data.head()

Similarly I have done the same for all the numerical columns which contain the missing value.

So finally the categorical missing value and the numerical missing value has been filled and lets move onto the next process.

## Step - 7 Handling Outliers:

By using the describe function we can get a rough overview of which column have a extreme values

Final\_cleaned.describe()

Identified what are the columns there are possibility for the Outliers:

Lets see which are the column there is a possibility for any outliers

kilometers\_Driven ,Mileage,Engine,Max\_Power,Torque\_Nm and Lets check Price for safer side.

For Max power There is only one value that has an extreme value so I dropped it.

Next - For mileage

For this I have filled with a mean after founded the upper and lower limit for the mileage.

In this I have noticed there is value below the lower limit and I have filled the average for that particular model.

percentile25 = Final\_cleaned['Mileage'].quantile(0.25)

percentile75 = Final\_cleaned['Mileage'].quantile(0.75)

iqr=percentile75 - percentile25

print(iqr)

print(percentile25)

print(percentile75)

upper\_limit = percentile75 + 1.5 \* iqr

lower\_limit = percentile25 - 1.5 \* iqr

print(upper\_limit)

print(lower\_limit)

average\_Mi = Final\_cleaned[Final\_cleaned['Mileage'] != 0]['Mileage'].mean()

# Filter for 'Mercedes-Benz S-Class' in the 'Model' column

mercedes\_s\_class = Final\_cleaned[Final\_cleaned['Model'] == 'Jaguar XJ']

# Calculate the mean for the filtered group

mercedes\_s\_class\_mean = mercedes\_s\_class['Mileage'].mean()

# Display the mean values

print(round(mercedes\_s\_class\_mean,2))

Final\_cleaned.loc[(Final\_cleaned['Mileage']<lower\_limit) & (Final\_cleaned['Model'] == "Jaguar XJ"), 'Mileage'] = 12.35

### Issue In Filling outlier:

Even though some Models have a value which is very much higher than the upper limit it was not a false value.

False Value in the sense those value are not an abnormal values they are original value for that particular model for example the model like Jaguar XJ 5.0L it have a engine power of 5000CC which is to be a extreme value but it was the actual value of that model Before filling with the mean need to check in all possible way and fill the outlier.

whereas in this case all the columns have a extreme values but it was not a false value it was a actual value of that particular model we are doing the cleaning process in this kind of dataset we not need to blindly trust the IQR and replace it actual value is ethically wrong and it will lead to the False insight and false prediction.

### Capping the outliers:

So what I have done is I have capped the outliers so it will show a less effect in the feature process.and that to I have considered the 5% to 95% of the data.

import pandas as pd

columns\_to\_cap = ['Price','Engine', 'Mileage','kilometers\_Driven']

# Using IQR to calculate lower and upper thresholds

for column in columns\_to\_cap:

Q1 = Price\_outlier[column].quantile(0.05)

Q3 = Price\_outlier[column].quantile(0.95)

IQR = Q3 - Q1

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

# Capping the values

Price\_outlier[column] = Price\_outlier[column].clip(lower\_bound, upper\_bound)

And finally I have saved the cleaned file into a csv.

## EDA - Exploratory data analysis.

### Visual Analysis

Since it was mainly focused on price prediction I have not explored all the possible combinations to visualize the data to get an in -depth analysis.But still I have explored a required analysis to get an intermediate knowledge of the data.

#### Let see which car the cardekho have the most Top 15 car Count wise Based on Model

import seaborn as sns

import matplotlib.pyplot as plt

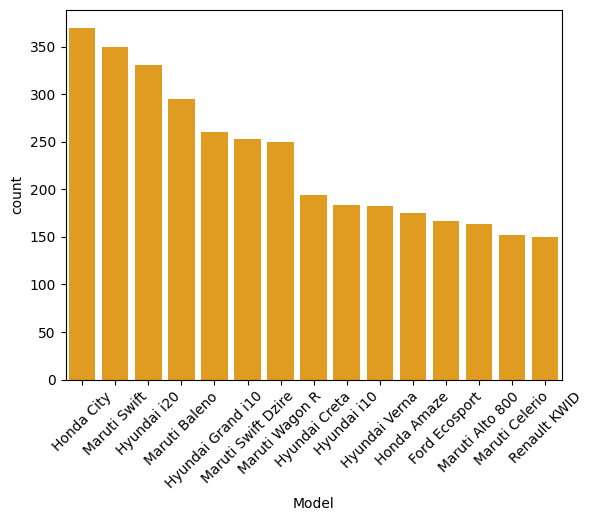
# Get the top 15 models based on their frequency

top\_15\_models = EDA\_df['Model'].value\_counts().head(15).index

# Create the countplot for the top 15 models

sns.countplot(x='Model', data=EDA\_df, color='orange', order=top\_15\_models)

plt.xticks(rotation=45)



#### Which variant does the cardekho have the most :

import seaborn as sns

import matplotlib.pyplot as plt

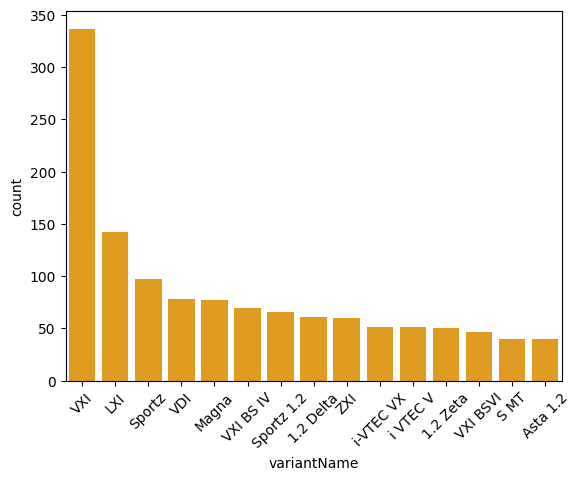
# Get the top 15 models based on their frequency

top\_15\_models = EDA\_df['variantName'].value\_counts().head(15).index

# Create the countplot for the top 15 models

sns.countplot(x='variantName', data=EDA\_df, color='orange', order=top\_15\_models)

plt.xticks(rotation=45)



#### Which city have the more number of cars in cardekho:

import seaborn as sns

import matplotlib.pyplot as plt

# Get the top 15 models based on their frequency across all cities

models = EDA\_df['variantName'].value\_counts().index

filtered\_df = EDA\_df[EDA\_df['variantName'].isin(models)]

# Create a countplot to visualize the distribution of cars across cities for the top 15 models

plt.figure(figsize=(12, 8))

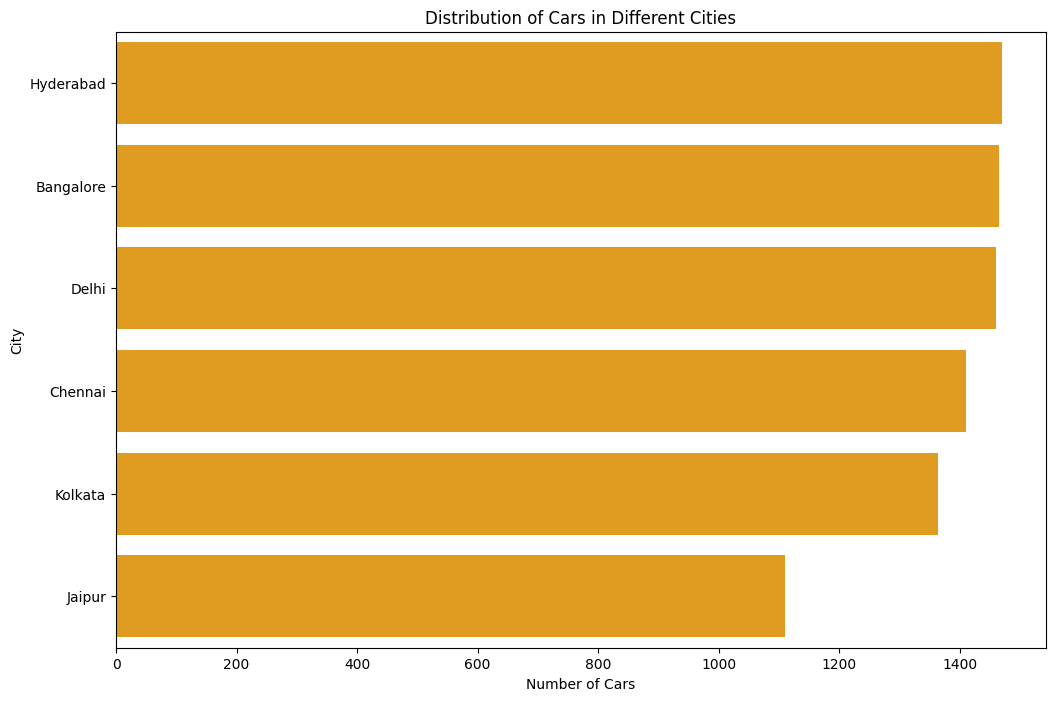
sns.countplot(y='City', data=filtered\_df, order=filtered\_df['City'].value\_counts().index, color='orange')

plt.title('Distribution of Cars in Different Cities')

plt.xlabel('Number of Cars')

plt.ylabel('City')

plt.show()



#### Which fuel types cars does cardekho have in different cities:

import seaborn as sns

import matplotlib.pyplot as plt

sns.countplot(x='Fuel\_type', data=EDA\_df, hue='City', order=EDA\_df['Fuel\_type'].value\_counts().index)

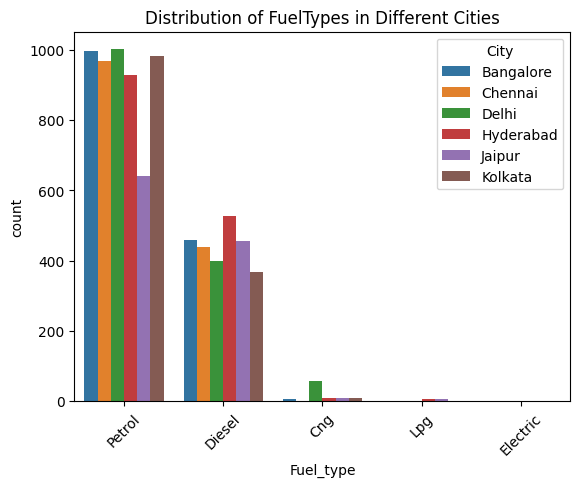
# Rotate the x-axis labels

plt.xticks(rotation=45)

plt.title('Distribution of FuelTypes in Different Cities')

# Display the plot

plt.show()



#### Lets compare the same for the Transmission Type:

import seaborn as sns

import matplotlib.pyplot as plt

sns.countplot(x='Transmission\_Type', data=EDA\_df, hue='City', order=EDA\_df['Transmission\_Type'].value\_counts().index)

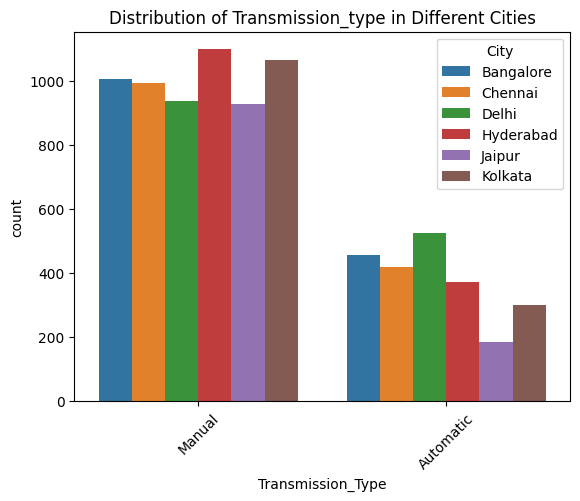
# Rotate the x-axis labels

plt.xticks(rotation=45)

plt.title('Distribution of Transmission\_type in Different Cities')

# Display the plot

plt.show()



#### Now lets see which year model cars do we have the most

import seaborn as sns

import matplotlib.pyplot as plt

# Group the data by 'Year' and count the number of models

yearly\_data = EDA\_df.groupby('Model\_Year')['Model'].count().reset\_index()

yearly\_data = yearly\_data.sort\_values(by='Model\_Year')

# Create a bar plot

plt.figure(figsize=(12, 8))

sns.barplot(x='Model\_Year', y='Model', data=yearly\_data, palette='viridis')

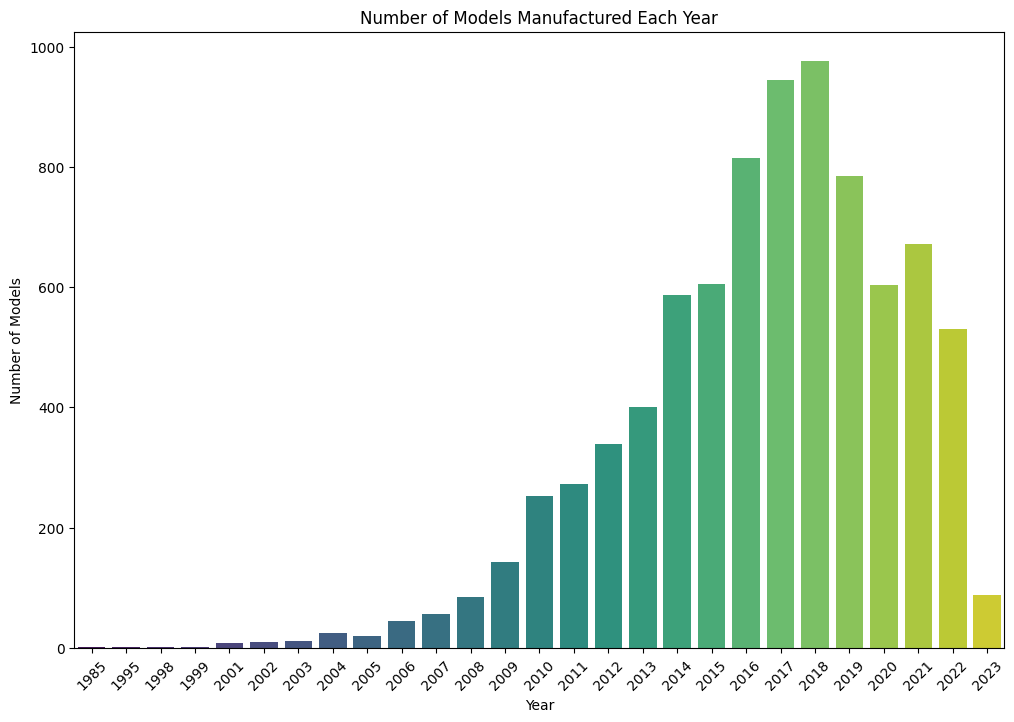
plt.title('Number of Models Manufactured Each Year')

plt.xlabel('Year')

plt.ylabel('Number of Models')

plt.xticks(rotation=45)

plt.show()



Now lets see which type of interior systems we have in our car the most:

Note : in this column there is no air conditioner in it.

import seaborn as sns

import matplotlib.pyplot as plt

sns.countplot(x='Interior\_system', data=EDA\_df, hue='City', order=EDA\_df['Interior\_system'].value\_counts().index)

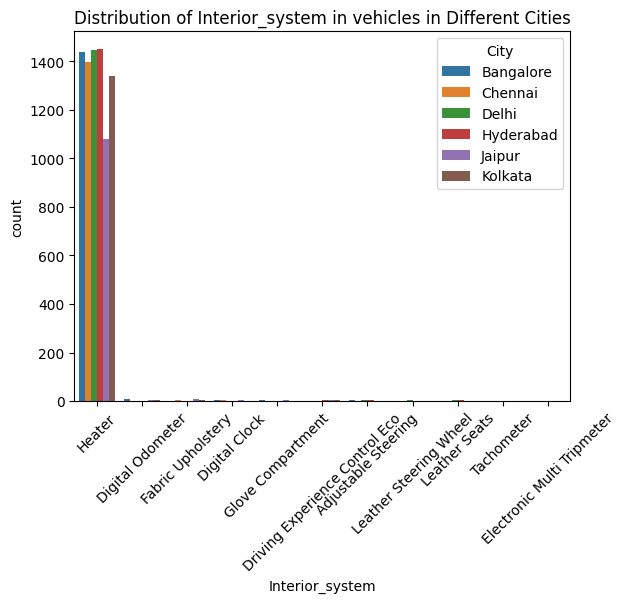
# Rotate the x-axis labels

plt.xticks(rotation=45)

plt.title('Distribution of Interior\_system in vehicles in Different Cities')

# Display the plot

plt.show()



#### Now lets see which type of safety systems we have in our car the most:

import seaborn as sns

import matplotlib.pyplot as plt

sns.countplot(x='Safety\_features', data=EDA\_df, hue='City', order=EDA\_df['Safety\_features'].value\_counts().index)

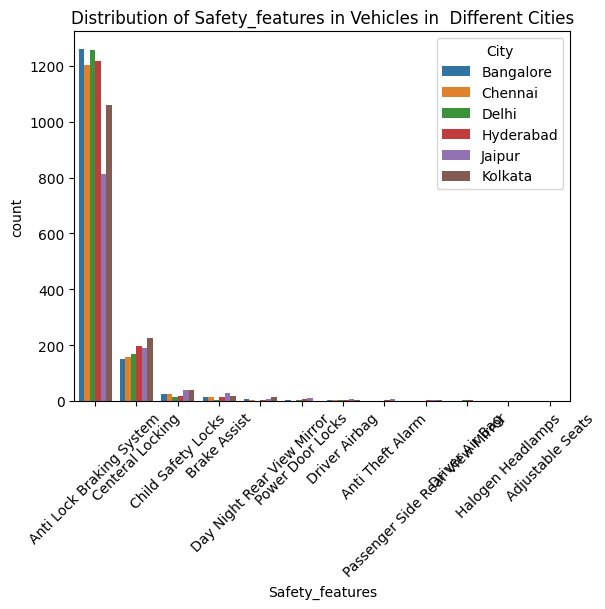
# Rotate the x-axis labels

plt.xticks(rotation=45)

plt.title('Distribution of Safety\_features in Vehicles in Different Cities')

# Display the plot

plt.show()



#### Now let see which Model perform exceptionally well:

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

current\_year = 2024

# Calculate the age of each car

EDA\_df['Car Age'] = current\_year - EDA\_df['Model\_Year']

# Calculate the average kilometers driven per year

EDA\_df['Km per Year'] = EDA\_df['kilometers\_Driven'] / EDA\_df['Car Age']

# Group by model and calculate the average kilometers driven per year for each model

model\_life = EDA\_df.groupby('Model')['Km per Year'].mean().reset\_index()

# Sort by average kilometers driven per year to identify models with longer life

model\_life\_sorted = model\_life.sort\_values(by='Km per Year', ascending=False)

Final=model\_life\_sorted.head(15)

# Visualization: Bar plot to show which models have the longest life

plt.figure(figsize=(12, 8))

sns.barplot(y='Model', x='Km per Year', data=Final, palette='magma')

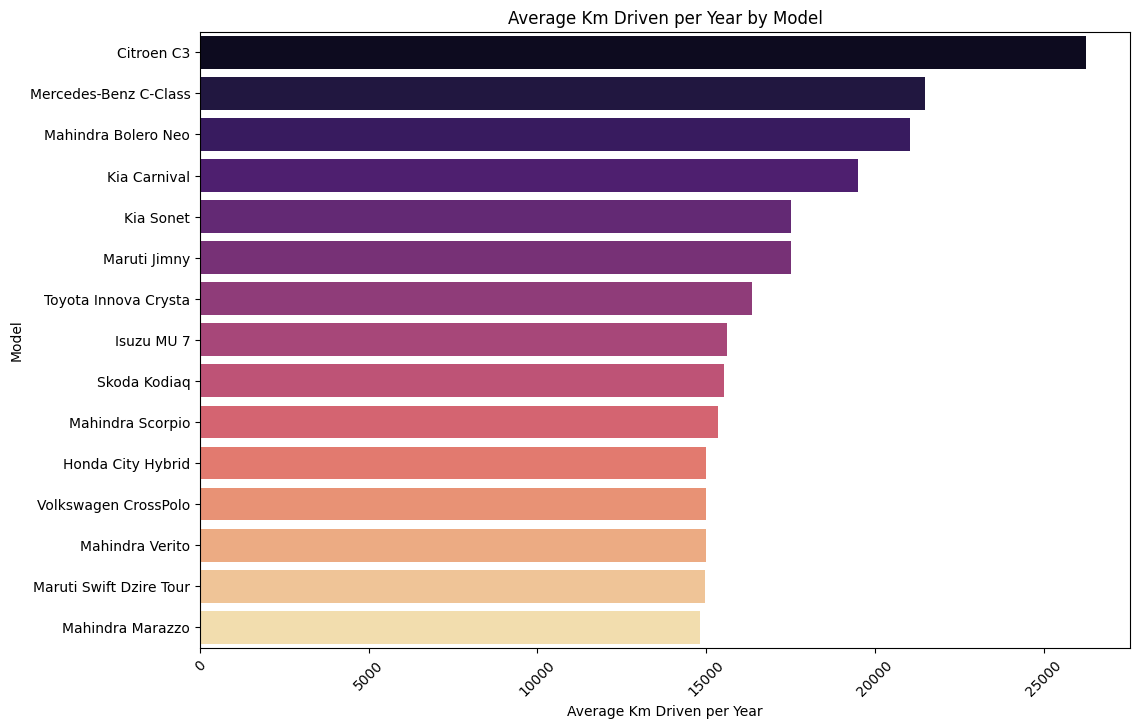
plt.title('Average Km Driven per Year by Model')

plt.xlabel('Average Km Driven per Year')

plt.ylabel('Model')

plt.xticks(rotation=45)

plt.show()



#### How car age is important for the Price:

plt.figure(figsize=(14, 10))

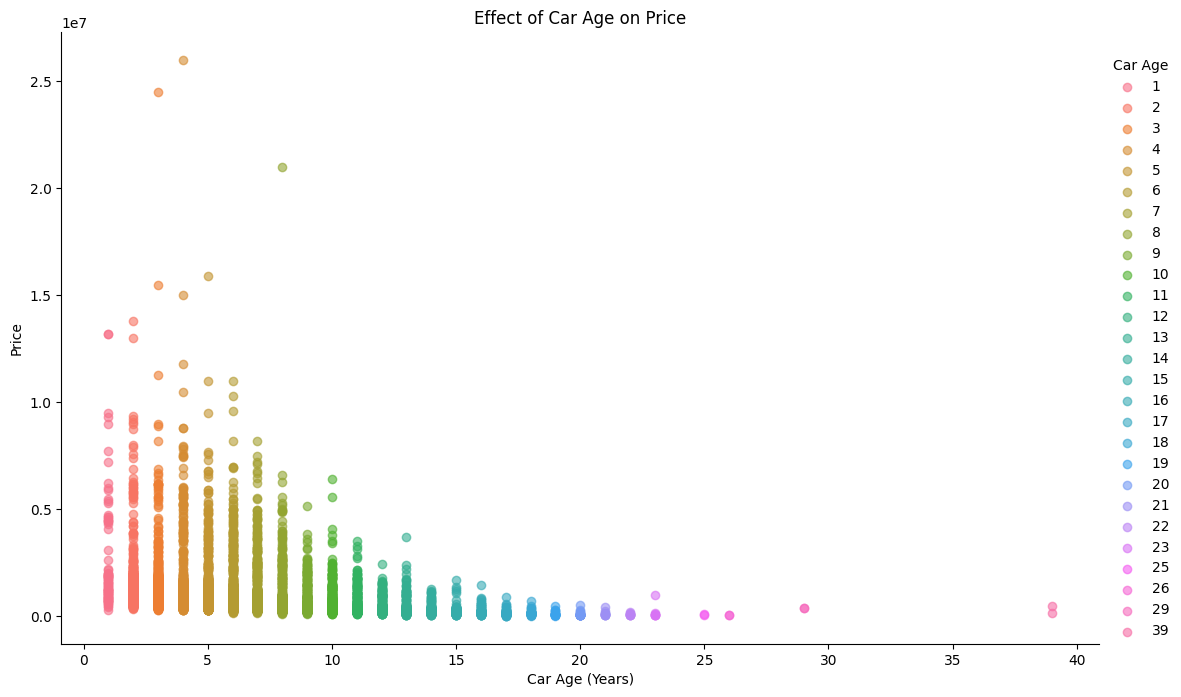
sns.lmplot(x='Car Age', y='Price', hue='Car Age', data=EDA\_df, height=7, aspect=1.6, scatter\_kws={'alpha':0.6}, ci=None)

plt.title('Effect of Car Age on Price')

plt.xlabel('Car Age (Years)')

plt.ylabel('Price')

plt.show()



#### How car Previous owner is important to the Price:

plt.figure(figsize=(14, 10))

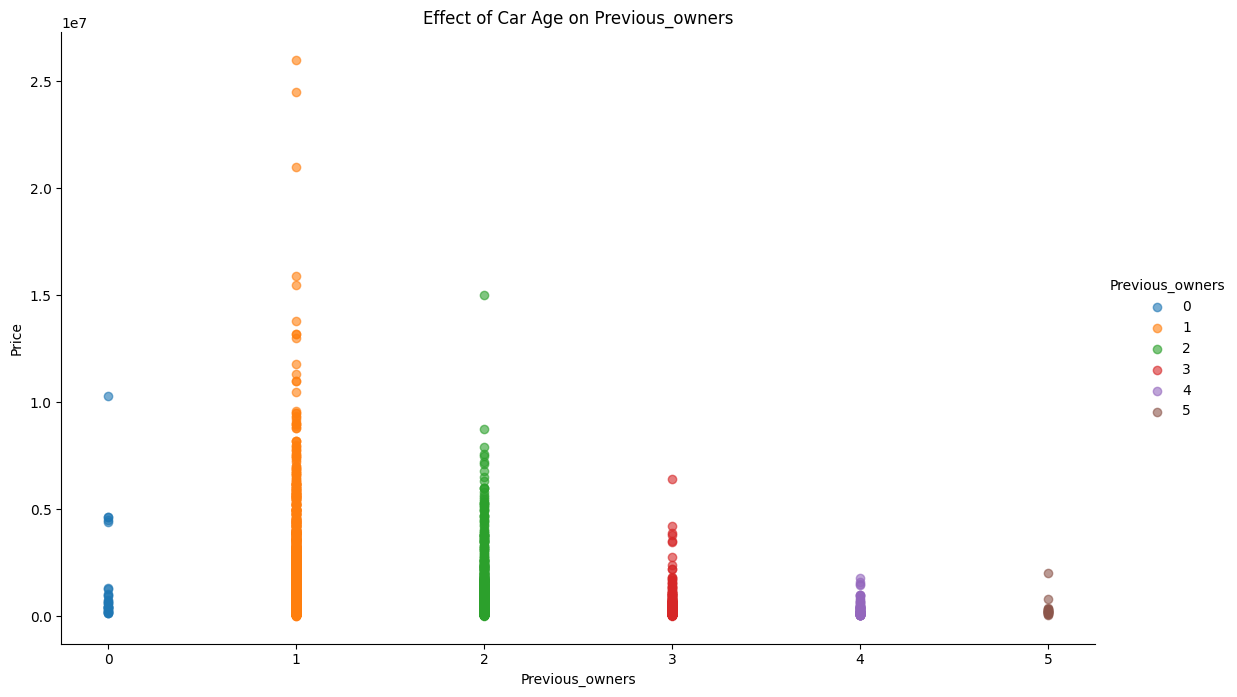
sns.lmplot(x='Previous\_owners', y='Price', hue='Previous\_owners', data=EDA\_df, height=7, aspect=1.6, scatter\_kws={'alpha':0.6}, ci=None)

plt.title('Effect of Car Age on Previous\_owners')

plt.xlabel('Previous\_owners')

plt.ylabel('Price')

plt.show()



#### Effect of Engine on price

plt.figure(figsize=(14, 10))

sns.lmplot(x='Engine', y='Price', data=EDA\_df, height=7, aspect=1.6, scatter\_kws={'alpha':0.6}, ci=None)

plt.title('Effect of Engine on Price')

plt.xlabel('Engine')

plt.ylabel('Price')

plt.show()



#### What type of Body type vehicle we do have the most:

import seaborn as sns

import matplotlib.pyplot as plt

# Get the top 15 models based on their frequency

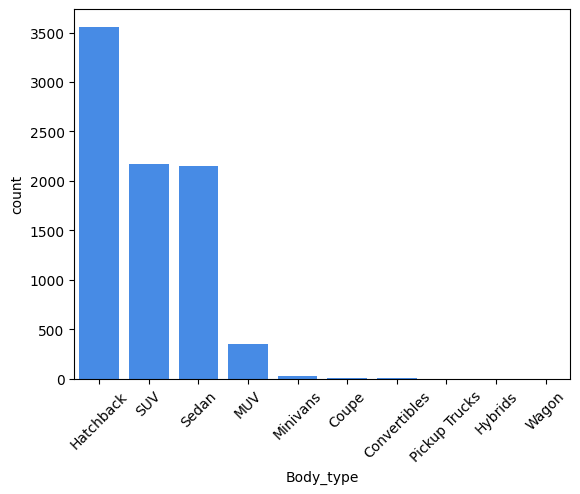
top\_15\_models = EDA\_df['Body\_type'].value\_counts().head(15).index

# Create the countplot for the top 15 models

sns.countplot(x='Body\_type', data=EDA\_df, color='#2D88FF', order=top\_15\_models)

plt.xticks(rotation=45)

plt.show()



#### Spread of Insurance Validity in cars in different cities:

import seaborn as sns

import matplotlib.pyplot as plt

sns.countplot(x='Insurance Validity', data=EDA\_df, hue='City', order=EDA\_df['Insurance Validity'].value\_counts().index)

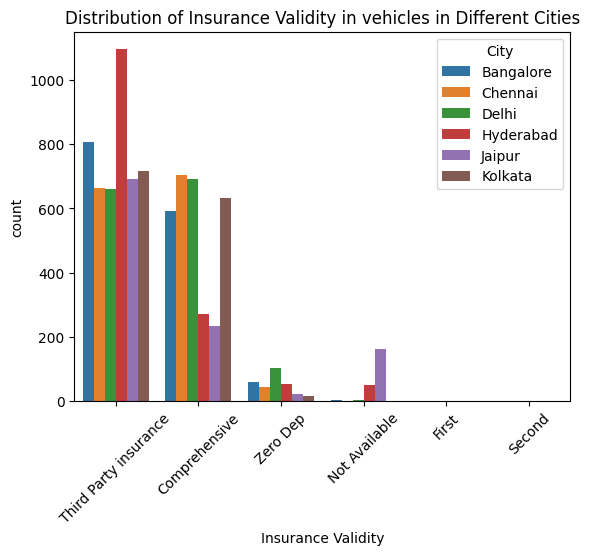
# Rotate the x-axis labels

plt.xticks(rotation=45)

plt.title('Distribution of Insurance Validity in vehicles in Different Cities')

# Display the plot

plt.show()



As I said earlier I have not done a detailed analysis, I have done an intermediate analysis to get to know about the feature importances.For doing detailed analysis there is a scope in this dataset.

### Hypothesis Testing for getting the idea for feature Importance

Before that I am going to take required Numerical columns using the domain knowledge and People's preference for Training ML algorithm Columns are ----> kilometers\_Driven,Previous\_owners,Seats,Mileage,Engine,Car Age for Numerical value

For Categorical value let do the Hypothesis testing and pick based on that.

Since I will going to predict the price I done a test only for price and the associated categorical column

The price column is not evenly distributed and I comparing it with categorical value I have chosen

**kruskal - Test**

#### City Vs price:

How the city is determining the price whether the is any dependency between the price to the car present in the city.

c0 =EDA\_df[EDA\_df['City'] == 'Bangalore']['Price']

c1 = EDA\_df[EDA\_df['City'] == 'Chennai']['Price']

c2 = EDA\_df[EDA\_df['City'] == 'Hyderabad']['Price']

c3 = EDA\_df[EDA\_df['City'] == 'Jaipur']['Price']

c4 = EDA\_df[EDA\_df['City'] == 'Kolkata']['Price']

c5 = EDA\_df[EDA\_df['City'] == 'Delhi']['Price']

h\_stat, p\_value = stats.kruskal(c0, c1, c2, c3, c4, c5)

print(f'H-statistic: {h\_stat}')

print(f'P-value: {p\_value}')

P-Value → 1.8441955252207967e-57

It shows the significant difference so there is a strong dependency between the Price and the City.

#### Fuel Type Vs price:

c0 =EDA\_df[EDA\_df['Fuel\_type'] == 'Petrol']['Price']

c1 = EDA\_df[EDA\_df['Fuel\_type'] == 'Diesel']['Price']

c2 = EDA\_df[EDA\_df['Fuel\_type'] == 'Cng']['Price']

c2 = EDA\_df[EDA\_df['Fuel\_type'] == 'Lpg']['Price']

c2 = EDA\_df[EDA\_df['Fuel\_type'] == 'Electric']['Price']

h\_stat, p\_value = stats.kruskal(c0, c1, c2, c3, c4, c5)

print(f'H-statistic: {h\_stat}')

print(f'P-value: {p\_value}')

P- Value → 7.589145851682627e-173

It shows the significant difference so there is a strong dependency between the Price and the Fuel\_Type.

#### Transmission Type Vs Price

Since there is only two grom and the numerical column is not normally distributed I have gone with **mann whitney u Test**

import scipy.stats as stats

c0 = EDA\_df[EDA\_df['Transmission\_Type'] == 'Manual']['Price']

c1 = EDA\_df[EDA\_df['Transmission\_Type'] == 'Automatic']['Price']

# Perform the Mann-Whitney U test

u\_stat, p\_value = stats.mannwhitneyu(c0, c1, alternative='two-sided')

print(f'U-statistic: {u\_stat}')

print(f'P-value: {p\_value}')

P-Value —> 0.0

The extremely low p-value suggests that the price distribution for vehicles with Manual transmissions is significantly different from that of vehicles with Automatic transmissions.

Manufactured By Vs The Price

import scipy.stats as stats

# Extract Price data for each category

categories = EDA\_df['Manufactured\_By'].unique()

price\_data = [EDA\_df[EDA\_df['Manufactured\_By'] == cat]['Price'] for cat in categories]

# Perform the Kruskal-Wallis H-test

h\_stat, p\_value = stats.kruskal(\*price\_data)

print(f'H-statistic: {h\_stat}')

print(f'P-value: {p\_value}')

P - Value —-> 0.0

The Kruskal-Wallis H-test results indicate that there is a statistically significant difference in vehicle prices across different manufacturers. The extremely low p-value (0.0) strongly suggests that the manufacturer is a key factor influencing vehicle prices.

So Based on this I am able to decide which feature is majorly influencing the price of the car.

Numerical Columns are ----> kilometers\_Driven,Previous\_owners,Seats,Mileage,Engine,Car Age

Categorical data are ----> City,Body\_type,Fuel\_type,Transmission\_Type ,Manufactured\_By

## 

## ML - Machine Learning Algorithm and Developing the prediction model.

First as usual loading the data and storing it in the variable.

import pandas as pd

ML\_Data=pd.read\_csv(r'C:\Users\sandy\Desktop\Project\_realected\_practice\Car\_deako\ML\_Dataset\Caped\_ML\_Dataset.csv')

ML\_Data.head()

### Encoding for the Categorical data containing columns.

First let's do a Frequency encoding for all the categorical columns and store the Mapped value since it will be used in the Deployment process, Since the unique categorical value are higher I am going with the Frequency Encoding it will capture the Relation without increasing the dimension of the DataFrame..

In addition to that I need to take the mapped encoded values because it will be useful in the deployment phase of my ML model so I have taken the encoded value as a key-value pair in dictionary format.

import pandas as pd

from sklearn.preprocessing import OneHotEncoder

import pickle

# Sample data loading (replace with your actual data loading)

# ML\_Data = pd.read\_csv('path\_to\_your\_data.csv')

# Define categorical columns

frequency\_encoding\_columns = ['Body\_type', 'Manufactured\_By','City', 'Fuel\_type', 'Transmission\_Type']

# Frequency Encoding for Body\_type and Manufactured\_By

category\_mappings = {}

for column in frequency\_encoding\_columns:

frequency\_map = ML\_Data[column].value\_counts(normalize=True) # Get frequency distribution

ML\_Data[column + '\_freq'] = ML\_Data[column].map(frequency\_map) # Map frequencies to a new column

# Store frequency mappings

category\_mappings[column] = frequency\_map.to\_dict()

# Drop original categorical columns if needed

ML\_Data.drop(columns=frequency\_encoding\_columns, inplace=True)

# Save the encoded DataFrame and category mappings to a pickle file

with open(r'Fre\_en.pkl', 'wb') as f:

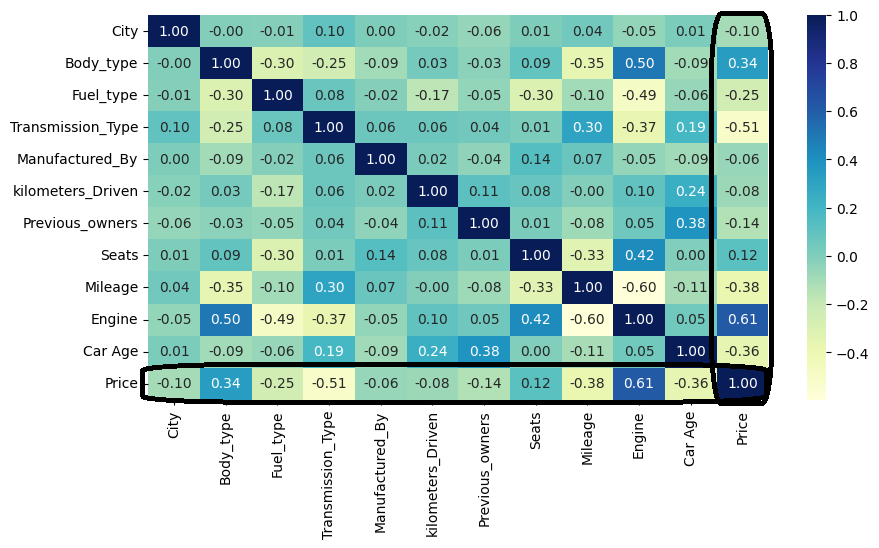
pickle.dump((category\_mappings), f)

print("Data and mappings saved to pickle file successfully!")

Now the encoding part is completed and lets check which feature have a higher weightage for the prediction of the price.

### Correlation of the Feature and target.

All the first 11 columns are the feature and the price column is the target. I have checked which feature is majorly impacting the price and whether it is impacting in positive or in negative way.



Strong >--------------< Weak

Positive correlation - Engine > Body\_type > Seats

Negative Correlation - Transmission\_Type > Mileage > Car Age > Fuel\_Type > Previous\_Owners > City > Kilometers\_Driven > Manufactured\_By

### Deciding and taking the Features and Targets:

By gathering information from EDA,Hypothesis Testing and from Heatmap(correlation) I have taken the right columns so no lets take the Features and Target.

It will leave the Price column and take all other column and assign it with a feature named variable

Features=ML\_Data.iloc[:,:-1]

And for Target I have taken only the Price column.

Target=ML\_Data['Price']

Target.head()

### Splitting the Features and targets for Training and Testing.

Basically the standard splitting ratio of Test and Train data is 70% for Training and 30% for Testing or 80% for Training and 20% for Testing. Since I got a little higher accuracy in 80:20 ratio I have gone with that itself.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(Features, Target, test\_size=0.2, random\_state=42)

And the random state is 42. This number can be anything, it is just a placeholder which can be useful in taking the same record i.e. every time we run this code the souffle and take the same data again.

### Scaling or Normalization of the features:

People majorly use this two scaling package one is Min max scaler and another one is standard scaler.

I have used a standard scaler here for scaling so I have applied it for only features.I have not applied for the target since there is no need for that.

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

### Train 7 Model with this dataset to compare which model is performing well:

For this I have taken the Linear regression,Decision Tree Regressor,Random Forest Regressor,Gradient Boosting Regressor,SVR , KNeighborsRegressor ,SGDRegressor for this dataset out of this let's find out which model performs well.

First all the needed packages are imported and Initialize the Models in different variables and that starts training the model.

from sklearn.tree import DecisionTreeRegressor

from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

from sklearn.svm import SVR

from sklearn.neighbors import KNeighborsRegressor

from sklearn.linear\_model import SGDRegressor

# Initialize the models

model\_Lr = LinearRegression()

model\_dt = DecisionTreeRegressor(random\_state=42)

model\_rf = RandomForestRegressor(random\_state=42, n\_estimators= 100)

model\_gb = GradientBoostingRegressor(n\_estimators = 100)

model\_svr = SVR()

model\_knn = KNeighborsRegressor(n\_neighbors=8)

model\_sgd = SGDRegressor()

# Fit each model

model\_Lr.fit(X\_train\_scaled, y\_train)

model\_dt.fit(X\_train\_scaled, y\_train)

model\_rf.fit(X\_train\_scaled, y\_train)

model\_gb.fit(X\_train\_scaled, y\_train)

model\_svr.fit(X\_train\_scaled, y\_train)

model\_knn.fit(X\_train\_scaled, y\_train)

model\_sgd.fit(X\_train\_scaled, y\_train)

### Checking the R2 score of all the Trained Model:

For this I have put the model name and its variable in the dictionary and I have separately given the code for Validating and evaluating the model performance Evaluation is the process of testing the model accuracy in a or with a Unseen or Unknown data simply the testing of accuracy in Test(Feature) data.Validation is the process of testing the model accuracy with the known data which is used for training (Trained Features Data)

So it will take the model one by one and check its R2 score and append the value one by one in the empty list which can be used for the Visualization purpose.

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.metrics import root\_mean\_squared\_error

# Calculate and print MSE and R^2 for each model

models = {

"LinearRegression": model\_Lr,

"Decision Tree": model\_dt,

"Random Forest": model\_rf,

"Gradient Boosting": model\_gb,

"Support Vector Regression": model\_svr,

"K-Nearest Neighbors": model\_knn,

"SGDRegressor": model\_sgd

}

Model=[]

R2\_Evalu=[]

R2\_validation=[]

# Iterate through the models to compute and display the metrics

for name, model in models.items():

y\_pred = model.predict(X\_test\_scaled)

rmse = root\_mean\_squared\_error(y\_test, y\_pred)

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

Model.append(name)

R2\_Evalu.append(round(r2,2))

print(f'Evaluation score')

print(f"{name} Model:")

print(f"Mean Squared Error: {mse:.2f}")

print(f"Root Mean Squared Error (RMSE):, {rmse:.2f}")

print(f"R^2 Score: {r2:.2f}")

print("-" \* 30)

y\_pred\_train = model.predict(X\_train\_scaled)

mse\_1 = mean\_squared\_error(y\_train,y\_pred\_train)

rmse\_1 = root\_mean\_squared\_error(y\_train,y\_pred\_train)

r2\_1 = r2\_score(y\_train,y\_pred\_train)

R2\_validation.append(round(r2\_1,2))

print(f'Validation score')

print(f"{name} Model:")

print(f"Mean Squared Error: {mse\_1:.2f}")

print(f"Root Mean Squared Error (RMSE): {rmse\_1:.2f}")

print(f"R^2 Score: {r2\_1:.2f}")

print("-" \* 30)

### 

### Models(Non- Optimized) comparison and accuracy check :

What this code basically does is that previously I have appended the model and its R2 score. It will create a graph for better understanding of each model's accuracy.

import matplotlib.pyplot as plt

import numpy as np

# Set up the figure and axes

plt.figure(figsize=(10, 6))

# Plotting the grouped bar chart

bar\_width = 0.35

index = np.arange(len(Model))

plt.bar(index, R2\_Evalu, width=bar\_width, color='royalblue', label='R2 Evaluation')

plt.bar(index + bar\_width, R2\_validation, width=bar\_width, color='coral', label='R2 Validation')

# Labeling the chart

plt.xlabel('Models', fontsize=13, color='darkblue')

plt.ylabel('R2 Score', fontsize=13, color='darkblue')

plt.title('Model Performance: Evaluation vs Validation', fontsize=15, color='darkred')

# Adding model names

plt.xticks(index + bar\_width / 2, Model, rotation=45, fontsize=11, color='darkgreen')

# Adding a legend and grid

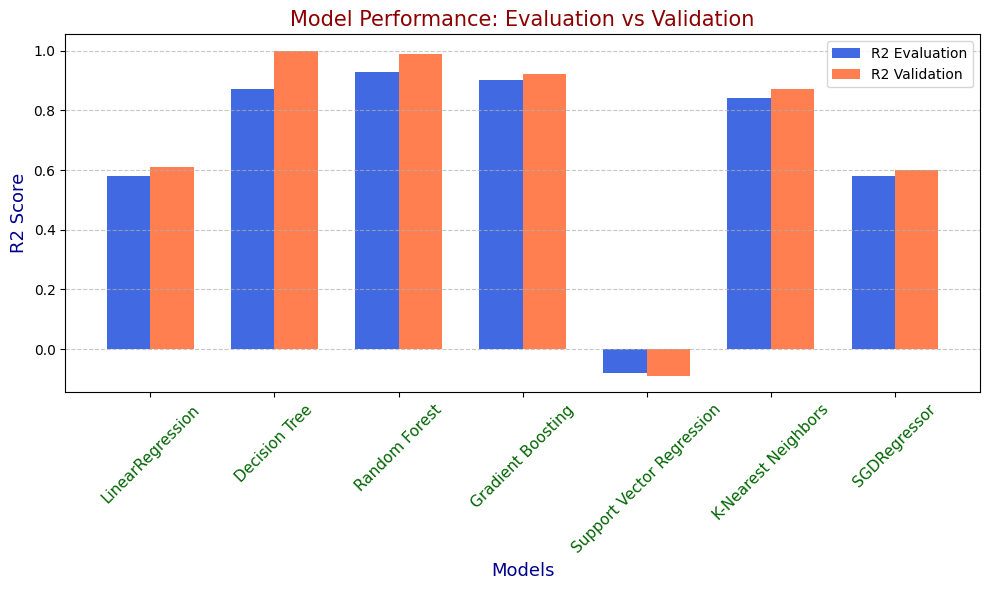
plt.legend()

plt.grid(axis='y', linestyle='--', alpha=0.7)

# Show the plot

plt.tight\_layout()

plt.show()



| **Characteristics** | **Linear Regression** | **Decision Tree** | **Random Forest** | **Gradient Boosting** | **Support Vector Regression (SVR)** | **K-Nearest Neighbors (KNN)** | **SGDRegressor** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| R2-Score Evaluation | 0.58 | 0.87 | 0.93 | 0.90 | -0.08 | 0.84 | 0.58 |
| R2 Score Validation | 0.61 | 1.00 | 0.99 | 0.92 | -0.09 | 0.87 | 0.60 |
| Mean Squared Error Evaluation | 416737029221.67 | 126191234133.15 | 69675931269.28 | 98919341046.31 | 1074393875411.62 | 161520072926.58 | 422159623381.42 |
| Mean Squared Error  Validation | 431284865766.95 | 141631729.67 | 11322133276.87 | 88454304104.67 | 1192343366112.46 | 140119676109.26 | 433009065611.74 |
| Root Mean Squared Error Evaluation | 645551.72 | 355234.06 | 263961.99 | 314514.45 | 1036529.73 | 401895.60 | 649738.12 |
| Root Mean Squared Error  Validation | 656722.82 | 11900.91 | 106405.51 | 297412.68 | 1091944.76 | 374325.63 | 658034.24 |

#### **Linear Regression:**

* **Performance:** The Linear Regression model shows a moderate performance with an R2

score of 0.58 on evaluation and 0.61 on validation. The RMSE values are relatively high, indicating significant errors in predictions.

* **Overfitting/Underfitting:** The model does not show significant overfitting or underfitting, as the validation scores are close to the evaluation scores. However, the overall performance suggests that it struggles to capture complex patterns in the data.

#### **Decision Tree:**

* **Performance:** The Decision Tree model shows Good performance on both evaluation (R2 =0.87) and validation (R2 =1.00) it is practically not possible. However, the validation RMSE is extremely low compared to the evaluation RMSE.
* **Overfitting/Underfitting:** This model overfits the data, as indicated by the perfect R2 score and very low RMSE on the validation set compared to the evaluation set.So it clearly shows it is overfitted.

#### **Random Forest :**

* **Performance:** Random Forest performs very well with an R2 score of 0.93 on evaluation and 0.99 on validation. The RMSE values are also relatively low.
* **Overfitting/Underfitting:** There is still an indication of overfitting, as the validation R2 is close to 1, but the gap between evaluation and validation RMSE is less pronounced compared to the Decision Tree. This suggests it was slightly good.

#### **Gradient Boosting**

* **Performance**: Gradient Boosting also performs well with an 𝑅 2 R 2 score of 0.90 on evaluation and 0.92 on validation. The RMSE values indicate lower prediction errors than some other models.

* **Overfitting/Underfitting**: The model shows balanced performance as the evaluation and validation scores are close.It looks better than other modes.

#### **Support Vector Regression (SVR)**

* **Performance:** The SVR model performs poorly, with a negative R2 score, indicating that the model is worse than a simple mean predictor.
* **Overfitting/Underfitting:** Both the evaluation and validation scores are negative, and RMSE is very high, suggesting that the model is underfitting and failing to capture the relationships in the data.

#### **K-Nearest Neighbors (KNN)**

* **Performance:** KNN shows a good performance with 𝑅 2 = 0.84 on evaluation and R 2 = 0.87 on validation. RMSE values are also reasonable.
* **Overfitting/Underfitting:** The KNN model shows a slight risk of overfitting but is relatively stable as the validation and evaluation scores are close.

#### **SGDRegressor**

* **Performance:** The SGDRegressor model has moderate performance, similar to Linear Regression, with an R2 score around 0.58-0.60 and high RMSE values.
* **Overfitting/Underfitting:** Like Linear Regression, it shows no significant overfitting or underfitting but has difficulty capturing complex relationships..

So By seeing this we are able to clearly say Gradient Boosting Regressor is performing well when compared to other Models.

### Model Optimization - Using Grid search CV

Since the Random Forest , Decision Tree and Gradient Boosting Regressor are given a Higher R2 score Lets Try to Optimize it with Grid Search CV.

**GridSearchCV** is a way to automatically find the best settings for a model by testing all combinations of parameters you provide.

#### Decision Tree:

from sklearn.tree import DecisionTreeRegressor

from sklearn.model\_selection import GridSearchCV

# Define the parameter grid for the DecisionTreeRegressor

params = {

'max\_depth': [2, 4, 6, 8, 10, 12, 15, 20], # Added deeper trees

'min\_samples\_split': [2, 3, 4, 5, 10], # Added more options for minimum samples required to split

'min\_samples\_leaf': [1, 2, 4, 6, 8], # Added more options for minimum samples at a leaf node

'max\_features': [None, 'auto', 'sqrt', 'log2'], # Number of features to consider for the best split

'criterion': ['mse', 'friedman\_mse', 'mae'], # Different criteria for splitting

'splitter': ['best', 'random'] # Strategy used to split at each node

}

# Initialize the DecisionTreeRegressor

regressor = DecisionTreeRegressor()

# Set up the GridSearchCV

gcv = GridSearchCV(estimator=regressor, param\_grid=params, cv=5, scoring='r2', n\_jobs=-1, verbose=2)

# Fit the grid search on the training data

gcv.fit(X\_train\_scaled, y\_train)

# Output the best parameters and the best score

print("Best Parameters: ", gcv.best\_params\_)

print("Best R² Score on Training Set: ", gcv.best\_score\_)

# Evaluate the best model on the test set

best\_regressor = gcv.best\_estimator\_

y\_pred = best\_regressor.predict(X\_test\_scaled)

r2\_test = r2\_score(y\_test, y\_pred)

print("R² Score on Test Set: ", r2\_test)

**Gridsearch Parameters:**

* **estimator=regressor:**

The machine learning model you want to tune. In this case, a regressor could be a model like GradientBoostingRegressor, RandomForestRegressor, etc.

* **param\_grid=params:**

A dictionary (params) where the keys are the hyperparameters of the estimator, and the values are lists of parameters to test. GridSearchCV will test all possible combinations of these parameters.

* **cv=5:**

The number of cross-validation folds. Here, 5-fold cross-validation is used, meaning the data will be split into 5 parts, and the model will be trained and validated 5 times, each time using a different part of the data for validation and the remaining parts for training.

* **scoring='r2':**

The scoring metric to evaluate the models. 'r2' refers to the R-squared metric, which indicates how well the model explains the variability of the data. A higher R-squared value means a better fit.

* **n\_jobs=-1:**

The number of jobs (cores) to run in parallel. -1 means using all available CPU cores, which speeds up the search.

* **verbose=2:**

Controls the verbosity of the output during the search. verbose=2 provides more detailed logs, showing the number of parameter combinations being processed.

**Params Parameter**

* **Max\_depth:**

The maximum depth of the tree. Controls how deep the tree can grow. Deeper trees can model more complex patterns but are more prone to overfitting.

* **min\_samples\_split:**

The minimum number of samples required to split an internal node. Higher values prevent the model from learning overly specific patterns (overfitting) by requiring more data at each split.

* **min\_samples\_leaf:**

The minimum number of samples required to be at a leaf node. Setting this to a higher value can smooth the model by limiting the number of splits.

* **max\_features:**

The number of features to consider when looking for the best split. None uses all features, 'auto' usually selects all features in decision trees, 'sqrt' uses the square root of the number of features, and 'log2' uses the logarithm base 2 of the number of features.

* **criterion:**

The function to measure the quality of a split. 'mse' is Mean Squared Error, 'friedman\_mse' is a variation of MSE optimized for boosting, and 'mae' is Mean Absolute Error. Different criteria can affect how the tree is split.

#### 

#### Random Forest Model Optimization:

**from sklearn.ensemble import RandomForestRegressor**

**from sklearn.model\_selection import GridSearchCV**

**# Initialize the RandomForestRegressor**

**rf = RandomForestRegressor(random\_state=42)**

**# Define an expanded parameter grid**

**param\_grid = {**

**'n\_estimators': [20,50,100], # Increased options for number of trees**

**'max\_depth': [10, 20, 40, None], # More depth values including deeper trees**

**'min\_samples\_split': [2, 5, 15], # More variations for splits**

**'min\_samples\_leaf': [1, 2, 6], # More leaf sizes**

**'max\_features': ['auto', 'sqrt', 'log2'], # All common feature selection methods**

**'bootstrap': [True, False] # Bootstrap on/off**

**}**

**# Setup the GridSearchCV**

**grid\_search = GridSearchCV(estimator=rf, param\_grid=param\_grid,**

**cv=5, n\_jobs=-1, verbose=2, scoring='r2')**

**# Fit the grid search to the data**

**grid\_search.fit(X\_train\_scaled, y\_train)**

**# Best parameters and best score**

**print(f"Best Parameters: {grid\_search.best\_params\_}")**

**print(f"Best R2 Score: {grid\_search.best\_score\_:.4f}")**

**# Get the best model from grid search**

**best\_rf = grid\_search.best\_estimator\_**

**# Make predictions using the best model**

**Best\_y\_pred = best\_rf.predict(X\_test\_scaled)**

* **n\_estimators**

The number of trees in the forest. More trees generally improve performance but increase computational cost. Testing different values helps balance accuracy and efficiency.

#### 

#### Finally The Gradient Boosting:

**from sklearn.ensemble import GradientBoostingRegressor**

**from sklearn.model\_selection import GridSearchCV, train\_test\_split**

**from sklearn.metrics import r2\_score**

**gbr = GradientBoostingRegressor(random\_state=42)**

**# Define the parameter grid**

**param\_grid = {**

**'n\_estimators': [100, 200, 300], # Number of boosting rounds**

**'learning\_rate': [0.01, 0.05, 0.1], # Step size shrinkage**

**'max\_depth': [3, 4, 5], # Maximum depth of each tree**

**'min\_samples\_split': [2, 5, 10], # Minimum samples to split an internal node**

**'min\_samples\_leaf': [1, 2, 4], # Minimum samples at a leaf node**

**'subsample': [0.8, 0.9, 1.0], # Fraction of samples used for fitting**

**'max\_features': ['auto', 'sqrt', 'log2'] # Number of features to consider**

**}**

**# Set up GridSearchCV**

**grid\_search = GridSearchCV(estimator=gbr,**

**param\_grid=param\_grid,**

**scoring='r2', # Optimizing for R² score**

**cv=5, # 5-fold cross-validation**

**n\_jobs=-1, # Use all cores**

**verbose=2)**

**# Fit the grid search**

**grid\_search.fit(X\_train\_scaled, y\_train)**

**# Best parameters and score**

**print("Best Parameters: ", grid\_search.best\_params\_)**

**print("Best R² Score on Training Set: ", grid\_search.best\_score\_)**

**# Evaluate on the test set**

**best\_gbr = grid\_search.best\_estimator\_**

**y\_pred\_gbr = best\_gbr.predict(X\_test\_scaled)**

**r2\_test\_gbr = r2\_score(y\_test, y\_pred\_gbr)**

**print("R² Score on Test Set: ", r2\_test\_gbr)**

* **N\_estimators:**

The number of boosting rounds (i.e., the number of trees in the ensemble). In gradient boosting, more estimators generally improve accuracy but can lead to overfitting if not regularized.

* **learning\_rate**

The step size shrinkage applied to each tree’s contribution. Lower values make the model more robust by requiring more trees to achieve the same effect. It's a critical parameter for controlling overfitting.

* **Subsample**

The fraction of samples used for fitting each individual tree. Using a subset (less than 1.0) can reduce overfitting and increase model robustness by introducing randomness.

### 

### Models R2 Scores after Optimization:

models\_Final=['DecisionTree\_before','DecisionTree\_After','RandomForest\_Before','RandomForest\_After','GradientBoosting\_Before','GradientBoosting\_After']

R2\_evalu\_Scores=[0.87,0.87,0.93,0.94,0.90,0.94]

R2\_Valu\_Scores=[1.0, 0.97,0.99,1.0,0.92,0.99]

import matplotlib.pyplot as plt

import numpy as np

# Set up the figure and axes

plt.figure(figsize=(10, 6))

# Plotting the grouped bar chart

bar\_width = 0.35

index = np.arange(len(models\_Final))

plt.bar(index, R2\_evalu\_Scores, width=bar\_width, color='royalblue', label='R2 Evaluation')

plt.bar(index + bar\_width, R2\_Valu\_Scores, width=bar\_width, color='coral', label='R2 Validation')

# Labeling the chart

plt.xlabel('Models', fontsize=13, color='darkblue')

plt.ylabel('R2 Score', fontsize=13, color='darkblue')

plt.title('Model Performance: Evaluation vs Validation', fontsize=15, color='darkred')

# Adding model names

plt.xticks(index + bar\_width / 2, models\_Final, rotation=45, fontsize=11, color='darkgreen')

# Adding a legend and grid

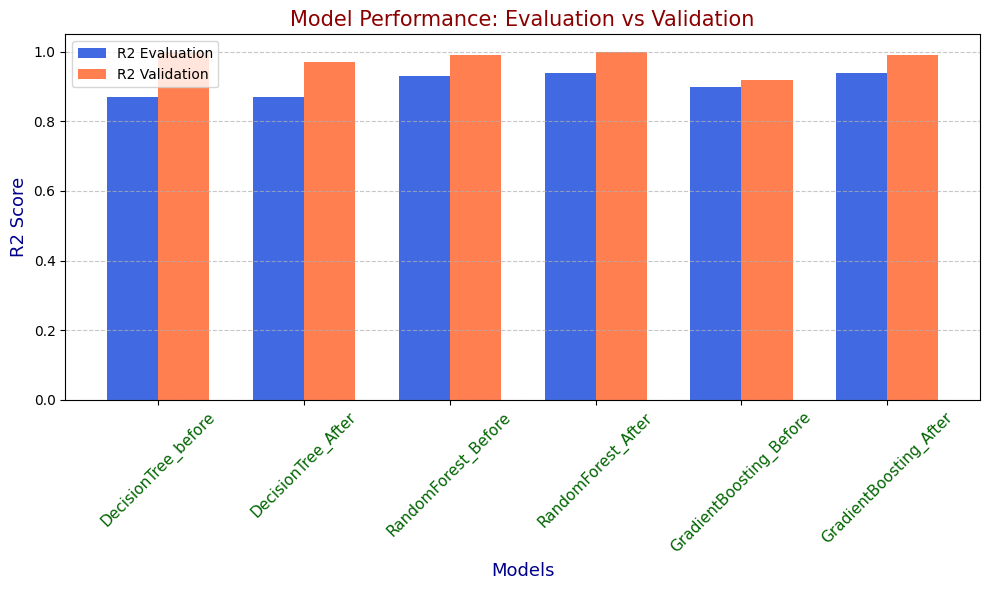
plt.legend()

plt.grid(axis='y', linestyle='--', alpha=0.7)

# Show the plot

plt.tight\_layout()

plt.show()



The Gradient Boosting Model’s R2 Score is increased Slightly even though it was increased this is a slight over Fitting is there so I have decided to go with the Gradient Boosting Previous model itself since it has less difference between The Evaluation and Validation scores so I will use that model for Deployment Process.

## Preserving the Trained Model:

I have used the Pickling technique for preserving my ML model and it can be used for the future deployment process.

\

import pickle

# Save the model to a file using Pickle

with open('Less\_Gradient\_boost\_model.pkl', 'wb') as file:

pickle.dump(model\_gb, file)

## Preserving the Scaler and Encoded File:

This is required while we deploying the app at conversion phase it needed.

import pickle

# Save the model to a file using Pickle

with open('Scale.pkl', 'wb') as file:

pickle.dump(scaler, file)

## 

## Deploying the App:

import streamlit as st

import pickle

import numpy as np

import time

# Load the pre-trained model, scaler, and frequency mappings

model\_path = r'C:\Users\sandy\Desktop\Project\_realected\_practice\Bank\Gradient\_boost\_model.pkl'

scale\_path = r'C:\Users\sandy\Desktop\Project\_realected\_practice\Bank\Scale.pkl'

frequency\_mappings\_path = r'C:\Users\sandy\Desktop\Project\_realected\_practice\Bank\Fre\_en.pkl'

with open(model\_path, 'rb') as file:

model = pickle.load(file)

with open(scale\_path, 'rb') as file:

scaler = pickle.load(file)

with open(frequency\_mappings\_path, 'rb') as file:

category\_mappings = pickle.load(file)

# Function to format price into thousands, lakhs, or crores

def format\_price(price):

if price >= 1e7:

return f"₹{price / 1e7:.2f} Cr"

elif price >= 1e5:

return f"₹{price / 1e5:.2f} Lakh"

else:

return f"₹{price / 1e3:.2f} Thousand"

# CSS for styling the app

css\_code = """

<style>

body {

background-image: url('file:///C:/Users/sandy/Downloads/vecteezy\_ai-generated-polished-shiny-beautiful-black-car-on-dark\_39617728.jpg'); /\* Replace with your background image \*/

background-size: cover;

background-repeat: no-repeat;

color: #ffffff; /\* Text color \*/

font-family: 'Arial', sans-serif; /\* Font family \*/

}

h1, h3 {

text-align: center;

color: #ffcc00; /\* Gold color for headings \*/

}

.sidebar .sidebar-content {

background-color: rgba(0, 0, 0, 0.7); /\* Semi-transparent sidebar \*/

color: #ffffff; /\* Sidebar text color \*/

}

.stButton>button {

background-color: #ff6600; /\* Button color \*/

color: white; /\* Button text color \*/

}

.stButton>button:hover {

background-color: #cc5200; /\* Button hover color \*/

}

.spinner {

color: #ffcc00; /\* Spinner color \*/

}

.price {

font-weight: bold;

font-size: 24px;

}

.car-animation img {

width: 300px;

display: block;

margin: auto;

}

</style>

"""

# Display the CSS in the Streamlit app

st.markdown(css\_code, unsafe\_allow\_html=True)

# Title of the app

st.title("Vehicle Price Predictor")

# Add a sidebar for navigation

st.sidebar.title("Navigation")

pages = ["Predict Price", "User Guide"]

selected\_page = st.sidebar.selectbox("Choose a page", pages)

if selected\_page == "Predict Price":

# Sidebar for categorical inputs

st.sidebar.header("Select Categorical Features")

# Dropdowns for categorical features

city = st.sidebar.selectbox("City", list(category\_mappings['City'].keys()))

body\_type = st.sidebar.selectbox("Body Type", list(category\_mappings['Body\_type'].keys()))

fuel\_type = st.sidebar.selectbox("Fuel Type", list(category\_mappings['Fuel\_type'].keys()))

transmission\_type = st.sidebar.selectbox("Transmission Type", list(category\_mappings['Transmission\_Type'].keys()))

manufactured\_by = st.sidebar.selectbox("Manufactured By", list(category\_mappings['Manufactured\_By'].keys()))

# Sliders for numerical features

kilometers\_driven = st.slider("Kilometers Driven", 0, 600000, 50000, step=500)

previous\_owners = st.slider("Previous Owners", 0, 8, 0)

seats = st.slider("Seats", 2, 10, 5)

mileage = st.slider("Mileage (kmpl)", 0.0, 50.0, 15.0)

engine = st.slider("Engine (cc)", 50.0, 5000.0, 500.0)

car\_age = st.slider("Car Age (years)", 0, 40, 5)

# Encoding categorical features based on the frequency mappings

encoded\_features = [

category\_mappings['Body\_type'].get(body\_type, 0),

category\_mappings['Manufactured\_By'].get(manufactured\_by, 0),

category\_mappings['City'].get(city, 0),

category\_mappings['Fuel\_type'].get(fuel\_type, 0),

category\_mappings['Transmission\_Type'].get(transmission\_type, 0),

]

# Combine all features into a single input array

features = np.array([

kilometers\_driven,

previous\_owners,

seats,

mileage,

engine,

car\_age,

\*encoded\_features

]).reshape(1, -1)

# Predict button with animation effect

if st.button("Predict Price"):

# Display animation while predicting

with st.spinner('Predicting...'):

time.sleep(0.5) # Simulate delay for prediction

# Predict and format the price

predicted\_price = model.predict(scaler.transform(features))[0]

formatted\_price = format\_price(predicted\_price)

# Display the estimated price

st.markdown(f"<h3 class='price'>Estimated Vehicle Price is: {formatted\_price}</h3>", unsafe\_allow\_html=True)

elif selected\_page == "User Guide":

st.header("User Guide")

st.write("""

### How to Use the Vehicle Price Predictor App:

\*\*Note:\*\* If the sidebar is not visible, click the (>) shaped icon to make it appear.

1. \*\*Select Categorical Features:\*\*

Use the dropdown menus in the sidebar to select the city, body type, fuel type, transmission type, and manufacturer of the vehicle.

- \*\*Step 1:\*\* Click the (>) shaped icon.

- \*\*Step 2:\*\* Select one option based on your preference or what you are looking for. For example:

- City → Chennai

- Body Type → Hatchback

- Fuel Type → Petrol

- Transmission Type → Automatic

- Manufactured By → Audi

2. \*\*Adjust Numerical Features:\*\*

Use the sliders to input the kilometers driven, number of previous owners, number of seats, mileage, engine size, and car age.

- \*\*Step 1:\*\* Click the (>) shaped icon.

- \*\*Step 2:\*\* Slide each slider based on your needs for each option.

After completing these steps, click the \*\*Predict Price\*\* button.

3. \*\*Predict Price:\*\*

Click on the \*\*'Predict Price'\*\* button to estimate the vehicle's price.

4. \*\*View Results:\*\*

After clicking the button, the predicted price will be displayed prominently in bold and dark formatting.

\*\*Note:\*\* The price will be displayed in an appropriate unit (thousands, lakhs, crores) depending on its value.

""")

### 

### Loading the Pickled Files:

# Load the pre-trained model

model\_path = 'Less\_Gradient\_boost\_model.pkl'

with open(model\_path, 'rb') as file:

model = pickle.load(file)

Scale\_path = 'Scale.pkl'

with open(Scale\_path, 'rb') as file:

Scale = pickle.load(file)

with open(frequency\_mappings\_path, 'rb') as file:

category\_mappings = pickle.load(file)

First I have loaded the Gradient boost model and after that I have loaded and stored the scale pickled file In a scale variable

Why the Scale file is important in the Gradient boost model will be stored. The knowledge will be that when the user is given an input first the input will be encoded and then it will be scaled so the output will be as we expected.

### Knowledge of Categorical Encoding:

While doing the label encoding itself I have created this dictionary and stored it as a json file

So the mapping will be done correctly and give the correct output as expected.

categorical\_mappings = {

"City": {"Bangalore": 0, "Chennai": 1, "Delhi": 2, "Hyderabad": 3, "Jaipur": 4, "Kolkata": 5},

"Body\_type": {"Convertibles": 0, "Coupe": 1, "Hatchback": 2, "Hybrids": 3, "MUV": 4, "Minivans": 5, "Pickup Trucks": 6, "SUV": 7, "Sedan": 8, "Wagon": 9},

"Fuel\_type": {"Cng": 0, "Diesel": 1, "Electric": 2, "Lpg": 3, "Petrol": 4},

"Transmission\_Type": {"Automatic": 0, "Manual": 1},

"Manufactured\_By": {"Audi": 0, "BMW": 1, "Chevrolet": 2, "Citroen": 3, "Datsun": 4, "Fiat": 5, "Ford": 6, "Hindustan Motors": 7, "Honda": 8, "Hyundai": 9, "Isuzu": 10, "Jaguar": 11, "Jeep": 12, "Kia": 13, "LandRover": 14, "Lexus": 15, "MG": 16, "Mahindra": 17, "Mahindra Renault": 18, "Mahindra Ssangyong": 19,

"Maruti": 20, "Mercedes-Benz": 21, "Mini": 22, "Mitsubishi": 23, "Nissan": 24, "Opel": 25, "Porsche": 26,

"Renault": 27, "Skoda": 28, "Tata": 29, "Toyota": 30, "Volkswagen": 31, "Volvo": 32

}}

### Converting the Price to a right format for correct understanding:

def format\_price(price):

if price >= 1e7:

return f"₹{price / 1e7:.2f} Cr"

elif price >= 1e5:

return f"₹{price / 1e5:.2f} Lakh"

else:

return f"₹{price / 1e3:.2f} Thousand"

The final result will be pronuced in cr or Lakh or in Thousand so there won’t be any misunderstanding.

### Front-End of my app:

Totally my app have a two pages one is Predict price page and other one is user guide page i predict price page the user need to give the 11 input 5 input as a selectbox where you can select the thing based on your preference and 6 as a slider.

st.sidebar.title("Navigation")

pages = ["Predict Price", "User Guide"]

selected\_page = st.sidebar.selectbox("Choose a page", pages)

if selected\_page == "Predict Price":

# Sidebar for categorical inputs

st.sidebar.header("Select Categorical Features")

# Dropdowns for categorical features

city = st.sidebar.selectbox("City", list(categorical\_mappings['City'].keys()))

body\_type = st.sidebar.selectbox("Body Type", list(categorical\_mappings['Body\_type'].keys()))

fuel\_type = st.sidebar.selectbox("Fuel Type", list(categorical\_mappings['Fuel\_type'].keys()))

transmission\_type = st.sidebar.selectbox("Transmission Type", list(categorical\_mappings['Transmission\_Type'].keys()))

manufactured\_by = st.sidebar.selectbox("Manufactured By", list(categorical\_mappings['Manufactured\_By'].keys()))

# Sliders for numerical features

kilometers\_driven = st.slider("Kilometers Driven", 0, 600000, 50000, step=500)

previous\_owners = st.slider("Previous Owners", 0, 6, 0)

seats = st.slider("Seats", 3, 10, 5)

mileage = st.slider("Mileage (kmpl)", 0.0, 50.0, 15.0)

engine = st.slider("Engine (cc)", 50.0, 5000.0, 500.0)

car\_age = st.slider("Car Age (years)", 0, 40, 5)

Predict Price funtionality:

First the input will be given by the user if it is categorical input means it will be convert into a respective numbers and all the categorical and numerical value are put into the array format and after that before predicting the price it will be scaled and the the prediction will be happens so thatr how the price is been calculated.

# Encoding categorical features based on the mapping

encoded\_features = [

categorical\_mappings['City'][city],

categorical\_mappings['Body\_type'][body\_type],

categorical\_mappings['Fuel\_type'][fuel\_type],

categorical\_mappings['Transmission\_Type'][transmission\_type],

categorical\_mappings['Manufactured\_By'][manufactured\_by],

]

# Combine all features into a single input array

features = np.array([

\*encoded\_features,

kilometers\_driven,

previous\_owners,

seats,

mileage,

engine,

car\_age

]).reshape(1, -1)

# Predict button

if st.button("Predict Price"):

# Display animation while predicting

with st.spinner('Predicting...'):

time.sleep(0.5) # Simulate delay for prediction

# Predict and format the price

predicted\_price = model.predict(Scale.transform(features))[0]

formatted\_price = format\_price(predicted\_price)

# Display the estimated price

st.markdown(f"<h3 style='color:darkyellow; font-weight:bold;'>Estimated Vehicle Price is: {formatted\_price}</h3>", unsafe\_allow\_html=True)

### User Guide:

Finally the user guide also provided for the better understanding of the user who is new to my app. In the user guide every step is clearly explained and the demo video also attached for the clear understanding.

That is how I have done the project right from the extraction to the app deployment. If there is any query or suggestion , please contact me through the git.

**Thank You**