Problem Decsription

In this problem, we need to develop predictive model to predict "Price" for a Toyota corolla based on its features. The features provided in the dataset all characteristics of the car. The feature names are self-descriptives:

1. Id
2. Model
3. Price
4. Age_08_04
5. Mfg_Month
6. Mfg_Year
7. KM
8. Fuel_Type
9. HP
10. Met_Color
11. Color
12. Automatic
13. CC
14. Doors
15. Cylinders
16. Gears
17. Quarterly_Tax
18. Weight
19. Mfr_Guarantee
20. BOVAG_Guarantee
21. Guarantee_Period
22. ABS
23. Airbag_1
24. Airbag_2
25. Airco
26. Automatic_airco
27. Boardcomputer
28. CD_Player
29. Central_Lock
30. Powered_Windows
31. Power_Steering
32. Radio
33. Mistlamps
34. Sport_Model
35. Backseat_Divider
36. Metallic_Rim

37. Radio_cassette
38. Parking_Assistant
39. Tow_Bar

Install upgraded libraries and Import other libraries

[ ] ↳ 3 cells hidden

Data Reading and Inspection

[ ] ↳ 6 cells hidden

Data Report

1. All the requirements for Data Preprocessing
2. Visualizations to quickly explore data

AutoViz helps summarize important dataset metrics in one line.

```
from autoviz.AutoViz_Class import AutoViz_Class
```

```
Imported v0.1.58. After importing, execute '%matplotlib inline' to display char
    AV = AutoViz_Class()
    dfte = AV.AutoViz(filename, sep=',', depVar='', dfte=None, header=0, verbos
                chart_format='svg',max_rows_analyzed=150000,max_cols_analyzed=30
Update: verbose=0 displays charts in your local Jupyter notebook.
        verbose=1 additionally provides EDA data cleaning suggestions. It also
        verbose=2 does not display charts but saves them in AutoViz_Plots folde
        chart_format='bokeh' displays charts in your local Jupyter notebook.
        chart_format='server' displays charts in your browser: one tab for each
        chart_format='html' silently saves interactive HTML files in your local
```

```
AV = AutoViz_Class() # Initialize Autoviz_Class
```

```
AV.AutoViz("", depVar= "Price",dfte = df, chart_format='bokeh')  # use method AutoViz.

# In dfte we specify the dataframe
# chart_format could be "html" or "server" or other options.
```

Guarantee_Period	9	int64	0	0.000000	0.626741	0
Mfg_Year	7	int64	0	0.000000	0.487465	0
Doors	4	int64	0	0.000000	0.278552	0
Gears	4	int64	0	0.000000	0.278552	0
Fuel_Type	3	object	0	0.000000	0.208914	17
Radio	2	int64	0	0.000000	0.139276	0
Automatic_airco	2	int64	0	0.000000	0.139276	0
Power_Steering	2	int64	0	0.000000	0.139276	0
Powered_Windows	2	int64	0	0.000000	0.139276	0
Central_Lock	2	int64	0	0.000000	0.139276	0
CD_Player	2	int64	0	0.000000	0.139276	0
Boardcomputer	2	int64	0	0.000000	0.139276	0
Automatic	2	int64	0	0.000000	0.139276	0
Airco	2	int64	0	0.000000	0.139276	0
Airbag_2	2	int64	0	0.000000	0.139276	0
Airbag_1	2	int64	0	0.000000	0.139276	0
ABS	2	int64	0	0.000000	0.139276	0
BOVAG_Guarantee	2	int64	0	0.000000	0.139276	0
Mfr_Guarantee	2	int64	0	0.000000	0.139276	0
Met_Color	2	int64	0	0.000000	0.139276	0
Mistlamps	2	int64	0	0.000000	0.139276	0

```
    30 Predictors classified...
    1 variables removed since they were ID or low-information variables
    List of variables removed: ['Model']
   Total columns > 30, too numerous to print.
Time to run AutoViz (in seconds) = 4
```

	Age_08_04	Mfg_Month	KM	HP	CC	Doors	Gears	Quarterly_Tax	Weight
0	23	10	46986	90	2000	3	5	210	1165
1	23	10	72937	90	2000	3	5	210	1165
2	24	9	41711	90	2000	3	5	210	1165
3	26	7	48000	90	2000	3	5	210	1165
4	30	3	38500	90	2000	3	5	210	1170
...	...	...	...	...	...	...	...	...	...
1431	69	12	20544	86	1300	3	5	69	1025

Data Report: It gives suggestions on the preprocessing steps. Now, based on the report, we shall plan preprocessing.

Now, we prepare a list of tasks we need to perform for each variable that requires data preprocessing.

Identify Categorical columns to see the cardinality of each of them.

```
for col in df.columns:
  if df[col].dtypes == "O":
    print(col)
    print(df[col].nunique())
```

```
Model
372
Fuel_Type
3
Color
10
```

Below are some custom transformers coded. We need to pass BaseEstimator and TransformerMixin into our custom class to inherit some functionality

Import libraries for Data Preprocessing.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
```

```
!pip install -U feature-engine # feature engine is a new library developed for feature engineering. This is fully compatible with sklearn pipeline and column transformer
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting feature-engine
  Downloading feature_engine-1.5.2-py2.py3-none-any.whl (290 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 290.0/290.0 KB 5.8 MB/s eta 0:00:00
Requirement already satisfied: statsmodels>=0.11.1 in /usr/local/lib/python3.8/dist-packages (from feature-engine) (0.13.5)
Requirement already satisfied: pandas>=1.0.3 in /usr/local/lib/python3.8/dist-packages (from feature-engine) (1.3.5)
Requirement already satisfied: scikit-learn>=1.0.0 in /usr/local/lib/python3.8/dist-packages (from feature-engine) (1.2.1)
Requirement already satisfied: numpy>=1.18.2 in /usr/local/lib/python3.8/dist-packages (from feature-engine) (1.22.4)
Requirement already satisfied: scipy>=1.4.1 in /usr/local/lib/python3.8/dist-packages (from feature-engine) (1.10.1)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.8/dist-packages (from pandas>=1.0.3->feature-engine) (2.8.2)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.8/dist-packages (from pandas>=1.0.3->feature-engine) (2022.7.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.8/dist-packages (from scikit-learn>=1.0.0->feature-engine) (3.1.0)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.8/dist-packages (from scikit-learn>=1.0.0->feature-engine) (1.2.0)
Requirement already satisfied: patsy>=0.5.2 in /usr/local/lib/python3.8/dist-packages (from statsmodels>=0.11.1->feature-engine) (0.5.3)
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.8/dist-packages (from statsmodels>=0.11.1->feature-engine) (23.0)
Requirement already satisfied: six in /usr/local/lib/python3.8/dist-packages (from patsy>=0.5.2->statsmodels>=0.11.1->feature-engine) (1.15.0)
Installing collected packages: feature-engine
Successfully installed feature-engine-1.5.2
```

Feature Engine is another library which works with Scikit-learn pipelines. It has many additional transformers to use readily . Learn more about the library here: https://feature-engine.trainindata.com/en/latest/

```
from feature_engine.encoding import RareLabelEncoder
```

Create Pipelines for different feature types.

```
# pipeline for nominal categorical columns
nom_cat_pipe = Pipeline(steps = [("imp", SimpleImputer(strategy= "constant", fill_value = "missing")),
                                 ("ohe", OneHotEncoder(sparse_output=False, handle_unknown='ignore')),])
```

```
df.columns
```

```
Index(['Id', 'Model', 'Price', 'Age_08_04', 'Mfg_Month', 'Mfg_Year', 'KM',
       'Fuel_Type', 'HP', 'Met_Color', 'Color', 'Automatic', 'CC', 'Doors',
       'Cylinders', 'Gears', 'Quarterly_Tax', 'Weight', 'Mfr_Guarantee',
       'BOVAG_Guarantee', 'Guarantee_Period', 'ABS', 'Airbag_1', 'Airbag_2',
       'Airco', 'Automatic_airco', 'Boardcomputer', 'CD_Player',
       'Central_Lock', 'Powered_Windows', 'Power_Steering', 'Radio',
       'Mistlamps', 'Sport_Model', 'Backseat_Divider', 'Metallic_Rim',
       'Radio_cassette', 'Parking_Assistant', 'Tow_Bar'],
      dtype='object')
```

Specify the column names in these lists.

```
nom_cat_vars = ['Model','Color', 'Fuel_Type'] # For Decision Tree perse, we don't need encoding of categorical columns
```

Put all preprocessing in a Column Transformer

```
preprocessor = ColumnTransformer(transformers = [("nom", nom_cat_pipe, nom_cat_vars)
                                                 #("ord", ord_cat_pipe, ord_cat_vars),
                                                 #("rare", rare_cat_pipe, rare_cat_vars),
                                                 #("norm", norm_num_pipe, norm_num_vars),
                                                 #("skew", skewed_num_pipe, skewed_num_vars),
                                                 #("disc", disc_pipe, disc_num_vars)
                                                 ],
                                 remainder = "passthrough")


preprocessor.set_output(transform = "pandas")
```

```
▸ ColumnTransformer
▸ nom    ▸ remainder
```

The above figure summarizes the preprocessing pipeline which makes our data ready to be fed into the model/estimator.

```
▸ OneHotEncoder
```

Import model libraries

```
from sklearn.tree import DecisionTreeRegressor

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_absolute_error, mean_squared_error

from sklearn.model_selection import train_test_split
```

Train-test split

```
X = df.drop(columns = ["Id", "Price"])
y = df["Price"]
```

We split the data into training data and testing data, y has the dependent variable and X has the independent variables/features/predictors

```
train_X, test_X, train_y, test_y = train_test_split(X, y, test_size = 0.3, random_state = 42)
```

```
train_X.shape
```

```
(1005, 37)
```

```
train_y.shape
```

```
(1005,)
```

```
DT_pipe = Pipeline(steps = [("pre", preprocessor),  ("rgr", DecisionTreeRegressor())])
```

```
DT_pipe.fit(train_X, train_y)
```

```
Pipeline
pre: ColumnTransformer
```

The pipeline has the estimator attached below the preprocessing steps, and we fit the pipeline using the training data

making predictions on the train and test sets

```
train_pred = DT_pipe.predict(train_X)

test_pred = DT_pipe.predict(test_X)
```

```
!pip install dmba # This library provides nice function for reporting model performance
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: dmba in /usr/local/lib/python3.8/dist-packages (0.1.0)
```

```
from dmba import regressionSummary # import a function for evaluating model performance
```

dmba has regressionSummary which summarizes important metrics in one line

```
# Model performance on Train data
print("DT Model Performance on Train data")
regressionSummary(train_y, train_pred)

print("****" * 15)
print("****" * 15)

# Model performance on Test data
print("DT Model Performance on Test data")
regressionSummary(test_y, test_pred)
```

```
DT Model Performance on Train data

Regression statistics

                      Mean Error (ME) : 0.0000
       Root Mean Squared Error (RMSE) : 0.0000
            Mean Absolute Error (MAE) : 0.0000
          Mean Percentage Error (MPE) : 0.0000
Mean Absolute Percentage Error (MAPE) : 0.0000
************************************************************
************************************************************
DT Model Performance on Test data

Regression statistics

                      Mean Error (ME) : -156.9466
       Root Mean Squared Error (RMSE) : 1376.0517
            Mean Absolute Error (MAE) : 1041.2854
          Mean Percentage Error (MPE) : -2.7997
Mean Absolute Percentage Error (MAPE) : 10.6835
```

Interpretation

On the test data, our predictions of "Price"

1. ME = -156.9466 --> On average, our predictions are greater by 157 dollars
2. MAE = 1041.2854 --> On average, our predictions are off by 1041 dollars.
3. MPE = -2.7 --> On average, our predictions are higher by 2.7 percentage.
4. MAPE = 10.6835 --> On average, our predictions are off by off by 10.6 percentage.

calculate these values ourselves

```
test_e = test_y - test_pred

abs_test_e = abs(test_e)

test_err_df = pd.DataFrame({"e": test_e, "y": test_y, "pred": test_pred, "abs_e": abs_test_e})

test_err_df.head()
```

	e	y	pred	abs_e
594	850.0	10800	9950.0	850.0
754	-1000.0	9950	10950.0	1000.0
630	-1450.0	7500	8950.0	1450.0
1259	250.0	9250	9000.0	250.0
903	-1200.0	9750	10950.0	1200.0

```
test_err_df["err_prec"] = (test_err_df["e"]/ test_err_df["y"]) *100

test_err_df["abs_e_prec"] = (test_err_df["abs_e"]/ test_err_df["y"]) *100

test_err_df.describe()
```

Random Forest

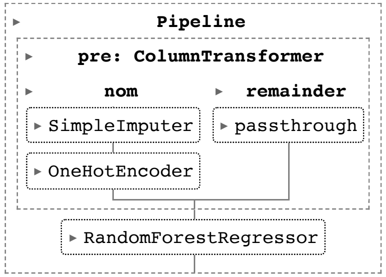
mean -156.946636 10734.143852 10891.090487 1041.285383 -2.799749 10.683506

```
from sklearn.ensemble import RandomForestRegressor
```

```
RF_pipe = Pipeline(steps = [("pre", preprocessor),  ("rf", RandomForestRegressor())])
```

25% -1000.000000 8425.000000 8750.000000 500.000000 -10.050251 4.422222

```
RF_pipe.fit(X = train_X, y = train_y)
```



Same as above pipeline, just the estimator is changed

```
train_pred = RF_pipe.predict(train_X)
```

```
test_pred = RF_pipe.predict(test_X)
```

```


# Model performance on Train data
print("RF Model Performance on Train data")
regressionSummary(train_y, train_pred)

print("****" * 15)
print("****" * 15)

# Model performance on Test data
print("RF Model Performance on Test data")
regressionSummary(test_y, test_pred)
```

```
RF Model Performance on Train data

Regression statistics

                      Mean Error (ME) : 5.1048
       Root Mean Squared Error (RMSE) : 408.4073
            Mean Absolute Error (MAE) : 293.0603
          Mean Percentage Error (MPE) : -0.3891
Mean Absolute Percentage Error (MAPE) : 2.9241
************************************************************
************************************************************
RF Model Performance on Test data
```

```
Regression statistics

                      Mean Error (ME) : -91.7326
       Root Mean Squared Error (RMSE) : 1062.7935
            Mean Absolute Error (MAE) : 775.4063
          Mean Percentage Error (MPE) : -2.2523
Mean Absolute Percentage Error (MAPE) : 7.9317
```

Fill the interpretations

On the test data, our predictions of "Sales"

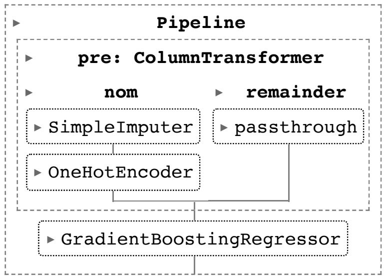
1. ME = -91.7326
2. MAE = 775.4063
3. MPE = -2.2523
4. MAPE = 7.9317

Gradient Boosting

```
from sklearn.ensemble import GradientBoostingRegressor
```

```
GB_pipe = Pipeline(steps = [("pre", preprocessor),  ("gb", GradientBoostingRegressor())])
```

```
GB_pipe.fit(train_X, train_y)
```




```
train_pred = GB_pipe.predict(train_X)

test_pred =GB_pipe.predict(test_X)
```

```
# Model performance on Train data
print("GB Model Performance on Train data")
regressionSummary(train_y, train_pred)

print("****" * 15)
print("****" * 15)

# Model performance on Test data
```

```
print("GB Model Performance on Test data")
regressionSummary(test_y, test_pred)
```

```
GB Model Performance on Train data

Regression statistics

                      Mean Error (ME) : -0.0000
       Root Mean Squared Error (RMSE) : 46.1503
            Mean Absolute Error (MAE) : 29.3304
          Mean Percentage Error (MPE) : -1.1901
Mean Absolute Percentage Error (MAPE) : 6.7039
************************************************************
************************************************************
GB Model Performance on Test data

Regression statistics

                      Mean Error (ME) : 1.3462
       Root Mean Squared Error (RMSE) : 47.5195
            Mean Absolute Error (MAE) : 30.2238
          Mean Percentage Error (MPE) : -0.9441
Mean Absolute Percentage Error (MAPE) : 6.8075
```

Fill the interpretations

On the test data, our predictions of "Sales"

1. ME = 1.7326
2. MAE = 30.4063
3. MPE = -0.9441
4. MAPE = 6.8075

This model gives us the least RMSE compared to Decision tree and Random forest. However, more steps like hyperparameter tuning might give us more promising results.