Predicting Fuel Efficiency of Vehicles - Part 3

Selecting and Training Models

1. Select and Train a few Algorithms(Linear Regression, Decision Tree, RandomForest)
2. Evaluation using Mean Squared Error
3. Model Evaluation using Cross Validation
4. Hyperparameter Tuning using GridSearchCV
5. Check Feature Importance
6. Evaluate the Final System on test data
7. Saving the Model

In [1]:

```
##importing a few general use case libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline

from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer



import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
pip install wget
```

```
Requirement already satisfied: wget in c:\users\lenovo\anaconda3\lib\site-packages (3.2)
Note: you may need to restart the kernel to use updated packages.
```

In [3]:
```
# reading the .data file using pandas
!python -m wget http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data
cols = ['MPG','Cylinders','Displacement','Horsepower','Weight',
                'Acceleration', 'Model Year', 'Origin']

df = pd.read_csv('./auto-mpg.data', names=cols, na_values = "?",
                comment = '\t',
                sep= " ",
                skipinitialspace=True)

data = df.copy()

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(data, data["Cylinders"]):
    strat_train_set = data.loc[train_index]
    strat_test_set = data.loc[test_index]
```

```
Saved under auto-mpg (1).data
```

In [4]:
```
##segregate the feature and target variable
data = strat_train_set.drop("MPG", axis=1)
data_labels = strat_train_set["MPG"].copy()
data
```

Out[4]:

	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	Origin
145	4	83.0	61.0	2003.0	19.0	74	3
151	4	79.0	67.0	2000.0	16.0	74	2
388	4	156.0	92.0	2585.0	14.5	82	1
48	6	250.0	88.0	3139.0	14.5	71	1
114	4	98.0	90.0	2265.0	15.5	73	2
...	...	...	...	...	...	...	...
147	4	90.0	75.0	2108.0	15.5	74	2
156	8	400.0	170.0	4668.0	11.5	75	1
395	4	135.0	84.0	2295.0	11.6	82	1
14	4	113.0	95.0	2372.0	15.0	70	3
362	6	146.0	120.0	2930.0	13.8	81	3

318 rows × 7 columns

In [5]:
```
##preprocess the Origin column in data
def preprocess_origin_cols(df):
    df["Origin"] = df["Origin"].map({1: "India", 2: "USA", 3: "Germany"})
    return df
```

In [6]:

```python
##creating custom attribute adder class
acc_ix, hpower_ix, cyl_ix = 4,2, 0

class CustomAttrAdder(BaseEstimator, TransformerMixin):
    def __init__(self, acc_on_power=True): # no *args or **kargs
        self.acc_on_power = acc_on_power
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X):
        acc_on_cyl = X[:, acc_ix] / X[:, cyl_ix]
        if self.acc_on_power:
            acc_on_power = X[:, acc_ix] / X[:, hpower_ix]
            return np.c_[X, acc_on_power, acc_on_cyl]

        return np.c_[X, acc_on_cyl]
```

In [7]:
```
def num_pipeline_transformer(data):
    '''
    Function to process numerical transformations
    Argument:
        data: original dataframe
    Returns:
        num_attrs: numerical dataframe
        num_pipeline: numerical pipeline object

    '''
    numerics = ['float64', 'int64']

    num_attrs = data.select_dtypes(include=numerics)

    num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attrs_adder', CustomAttrAdder()),
        ('std_scaler', StandardScaler()),
        ])
    return num_attrs, num_pipeline


def pipeline_transformer(data):
    '''
    Complete transformation pipeline for both
    nuerical and categorical data.

    Argument:
        data: original dataframe
    Returns:
        prepared_data: transformed data, ready to use
    '''
    cat_attrs = ["Origin"]
    num_attrs, num_pipeline = num_pipeline_transformer(data)
    full_pipeline = ColumnTransformer([
        ("num", num_pipeline, list(num_attrs)),
        ("cat", OneHotEncoder(), cat_attrs),
        ])
    prepared_data = full_pipeline.fit_transform(data)
    return prepared_data
```

From raw data to processed data in 2 steps

In [8]:
```
##from raw data to processed data in 2 steps
preprocessed_df = preprocess_origin_cols(data)
prepared_data = pipeline_transformer(preprocessed_df)
prepared_data
```

Out[8]:
```
array([[-0.85657842, -1.07804475, -1.15192977, ...,  1.        ,
         0.        ,  0.        ],
       [-0.85657842, -1.1174582 , -0.9900351 , ...,  0.        ,
         0.        ,  1.        ],
       [-0.85657842, -0.3587492 , -0.31547399, ...,  0.        ,
         1.        ,  0.        ],
       ...,
       [-0.85657842, -0.56566984, -0.53133355, ...,  0.        ,
         1.        ,  0.        ],
       [-0.85657842, -0.78244384, -0.23452666, ...,  1.        ,
         0.        ,  0.        ],
       [ 0.32260746, -0.45728283,  0.44003446, ...,  1.        ,
         0.        ,  0.        ]])
```

In [9]:
```
prepared_data[0]
```

Out[9]:
```
array([-0.85657842, -1.07804475, -1.15192977, -1.17220298,  1.21586943,
       -0.54436373,  1.70952741,  1.29565517,  1.        ,  0.        ,
        0.        ])
```

Selecting and Training Models

1. Linear Regression
2. Decision Tree
3. Random Forest
4. SVM regressor

In [10]:
```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(prepared_data, data_labels)
```

Out[10]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

In [11]:
```
##testing the predictions with the
sample_data = data.iloc[:5]
sample_labels = data_labels.iloc[:5]

sample_data_prepared = pipeline_transformer(sample_data)

print("Prediction of samples: ", lin_reg.predict(sample_data_prepared))
```

```
Prediction of samples:  [29.08069379 27.78336755 26.08031176 12.70419279 22.23454159]
```

In [12]:
```
print("Actual Labels of samples: ", list(sample_labels))
```

```
Actual Labels of samples:  [32.0, 31.0, 26.0, 18.0, 26.0]
```

Mean Squared Error

In [13]:
```
from sklearn.metrics import mean_squared_error

mpg_predictions = lin_reg.predict(prepared_data)
lin_mse = mean_squared_error(data_labels, mpg_predictions)
lin_rmse = np.sqrt(lin_mse)
lin_rmse
```

Out[13]:
```
2.9590402225760863
```

Decision Tree

In [14]:
```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(prepared_data, data_labels)
```

Out[14]:
```
DecisionTreeRegressor(ccp_alpha=0.0, criterion='mse', max_depth=None,
                      max_features=None, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, presort='deprecated',
                      random_state=None, splitter='best')
```

In [15]:
```
mpg_predictions = tree_reg.predict(prepared_data)
tree_mse = mean_squared_error(data_labels, mpg_predictions)
tree_rmse = np.sqrt(tree_mse)
tree_rmse
```

Out[15]:
```
0.0
```

But no model is perfect, this means that our model has overfit the data to a great extent.

We won't be touching out test data until we finalize our model. So, how do we check for what's happening?

Model Evaluation using Cross Validation

Scikit-Learn’s K-fold cross-validation feature randomly splits the training set into `K` distinct subsets called folds, then it trains and evaluates the model K times, picking a different fold for evaluation every time and training on the other K-1 folds.

The result is an array containing the K evaluation scores:

In [16]:
```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(tree_reg,
                         prepared_data,
                         data_labels,
                         scoring="neg_mean_squared_error",
                         cv = 10)
tree_reg_rmse_scores = np.sqrt(-scores)
```

In [17]:
```
tree_reg_rmse_scores
```

Out[17]:
```
array([2.91510077, 2.77302858, 3.03320169, 3.37374977, 2.19003425,
       2.9557148 , 2.70918576, 5.2122452 , 4.20560302, 2.6146609 ])
```

In [18]:
```
tree_reg_rmse_scores.mean()
```

Out[18]:
```
3.19825247393513
```

In [19]:
```
scores = cross_val_score(lin_reg, prepared_data, data_labels, scoring="neg_mean_squared_error", cv = 10)
lin_reg_rmse_scores = np.sqrt(-scores)
lin_reg_rmse_scores
```

Out[19]:
```
array([3.43254597, 3.45157629, 3.6621715 , 2.59652976, 2.48023405,
       2.74798115, 3.32524647, 2.42208917, 3.78133275, 2.8573747 ])
```

In [20]:
```
lin_reg_rmse_scores.mean()
```

Out[20]:
```
3.07570817937093244
```

Random Forest model

In [21]:
```
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor()
forest_reg.fit(prepared_data, data_labels)
forest_reg_cv_scores = cross_val_score(forest_reg,
                                       prepared_data,
                                       data_labels,
                                       scoring='neg_mean_squared_error',
                                       cv = 10)

forest_reg_rmse_scores = np.sqrt(-forest_reg_cv_scores)
forest_reg_rmse_scores.mean()
```

Out[21]: 2.5685941515759576

Support Vector Machine Regressor

In [22]:
```
from sklearn.svm import SVR

svm_reg = SVR(kernel='linear')
svm_reg.fit(prepared_data, data_labels)
svm_cv_scores = cross_val_score(svm_reg, prepared_data, data_labels,
                                scoring='neg_mean_squared_error',
                                cv = 10)
svm_rmse_scores = np.sqrt(-svm_cv_scores)
svm_rmse_scores.mean()
```

Out[22]: 3.08659162080283

Hyperparameter Tuning using GridSearchCV

In [23]:
```python
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid,
                           scoring='neg_mean_squared_error',
                           return_train_score=True,
                           cv=10,
                          )

grid_search.fit(prepared_data, data_labels)
```

Out[23]:
```
GridSearchCV(cv=10, error_score=nan,
             estimator=RandomForestRegressor(bootstrap=True, ccp_alpha=0.0,
                                             criterion='mse', max_depth=None,
                                             max_features='auto',
                                             max_leaf_nodes=None,
                                             max_samples=None,
                                             min_impurity_decrease=0.0,
                                             min_impurity_split=None,
                                             min_samples_leaf=1,
                                             min_samples_split=2,
                                             min_weight_fraction_leaf=0.0,
                                             n_estimators=100, n_jobs=None,
                                             oob_score=False, random_state=None,
                                             verbose=0, warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid=[{'max_features': [2, 4, 6, 8],
                          'n_estimators': [3, 10, 30]},
                         {'bootstrap': [False], 'max_features': [2, 3, 4],
                          'n_estimators': [3, 10]}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='neg_mean_squared_error', verbose=0)
```

In [24]:
```
grid_search.best_params_
```

Out[24]:
```
{'max_features': 6, 'n_estimators': 10}
```

In [25]:
```
cv_scores = grid_search.cv_results_

##printing all the parameters along with their scores
for mean_score, params in zip(cv_scores['mean_test_score'], cv_scores["params"]):
    print(np.sqrt(-mean_score), params)
```

```
3.5298036773871795 {'max_features': 2, 'n_estimators': 3}
3.1543065463436575 {'max_features': 2, 'n_estimators': 10}
2.9200301478012345 {'max_features': 2, 'n_estimators': 30}
3.4367720636873687 {'max_features': 4, 'n_estimators': 3}
2.86113945515678 {'max_features': 4, 'n_estimators': 10}
2.7573866919109475 {'max_features': 4, 'n_estimators': 30}
3.416158941425653 {'max_features': 6, 'n_estimators': 3}
2.674367098025250 3 {'max_features': 6, 'n_estimators': 10}
2.7323235595460096 {'max_features': 6, 'n_estimators': 30}
3.191919436072898 {'max_features': 8, 'n_estimators': 3}
2.6796452995595543 {'max_features': 8, 'n_estimators': 10}
2.68494337425188 {'max_features': 8, 'n_estimators': 30}
3.2998308648412786 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
2.9058143534045002 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
3.1910324793160885 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
2.8726444715767814 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
3.4152733193520746 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
2.693486477642302 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

Checking Feature importance

In [26]:
```python
# feature importances

feature_importances = grid_search.best_estimator_.feature_importances_
feature_importances
```

Out[26]:
```
array([0.28706325, 0.32659439, 0.09904163, 0.1015734 , 0.0182555 ,
       0.11371115, 0.03097858, 0.01759758, 0.00280047, 0.00114504,
       0.00123901])
```

In [27]:
```python
extra_attrs = ["acc_on_power", "acc_on_cyl"]
numerics = ['float64', 'int64']
num_attrs = list(data.select_dtypes(include=numerics))

attrs = num_attrs + extra_attrs
sorted(zip(attrs, feature_importances), reverse=True)
```

Out[27]:
```
[('acc_on_power', 0.030978576378998753),
 ('acc_on_cyl', 0.01759757612432403),
 ('Weight', 0.10157340340308983),
 ('Model Year', 0.11371115169237556),
 ('Horsepower', 0.09904163488978267),
 ('Displacement', 0.3265943896835737),
 ('Cylinders', 0.28706324723924487),
 ('Acceleration', 0.018255500770649784)]
```

Evaluating the entire system on Test Data

In [28]:
```python
final_model = grid_search.best_estimator_

X_test = strat_test_set.drop("MPG", axis=1)
y_test = strat_test_set["MPG"].copy()

X_test_preprocessed = preprocess_origin_cols(X_test)
X_test_prepared = pipeline_transformer(X_test_preprocessed)

final_predictions = final_model.predict(X_test_prepared)
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
```

In [29]:
```
final_rmse
```
Out[29]:
```
3.137180461815992
```

Creating a function to cover this entire flow

In [30]:
```
def predict_mpg(config, model):

    if type(config) == dict:
        df = pd.DataFrame(config)
    else:
        df = config

    preproc_df = preprocess_origin_cols(df)
    prepared_df = pipeline_transformer(preproc_df)
    y_pred = model.predict(prepared_df)
    return y_pred
```

In [31]:
```
##checking it on a random sample
vehicle_config = {
    'Cylinders': [4, 6, 8],
    'Displacement': [155.0, 160.0, 165.5],
    'Horsepower': [93.0, 130.0, 98.0],
    'Weight': [2500.0, 3150.0, 2600.0],
    'Acceleration': [15.0, 14.0, 16.0],
    'Model Year': [81, 80, 78],
    'Origin': [3, 2, 1]
}

predict_mpg(vehicle_config, final_model)
```
Out[31]:
```
array([29.22, 18.3 , 19.24])
```

Save the Model

In [32]:
```
import pickle
```

In [33]:
```
##saving the model
with open("model.bin", 'wb') as f_out:
    pickle.dump(final_model, f_out)
    f_out.close()
```

In [34]:
```
##loading the model from the saved file
with open('model.bin', 'rb') as f_in:
    model = pickle.load(f_in)

predict_mpg(vehicle_config, model)
```

Out[34]:
```
array([29.22, 18.3 , 19.24])
```

In [ ]: