Title: Machine Learning for fuel consumption prediction

Content of the document

1. Problem Statement

The objective of this project is **to predict the fuel efficiency of vehicles (MPG)** based on the other information about the vehicles. My company provided me with historical continuous data on MPG based on the fuel efficiency of each vehicle from the 70s to the 80s.

In order to accomplish this, I need to **create an end-to-end supervised machine learning pipeline** . Once the pipeline is designed and implemented, it will be submitted to the company's lead data scientist for prediction purposes.

Here are the steps I will take to build my pipeline:

```
1. Data Collection: I will use the  Auto MPG dataset obtained from the UCI ML Repository.
2. Data Exploration: This will be done to identify the most important features and combine them in new ways.
3. Data Preprocessing: Lay out a pipeline of tasks for transforming data for use in my machine learning model.
4. Model selection & Hyperparameter Tuning : Cross-validate a few models and fine-tune hyperparameters for
   models that showed promising predictions.
5. Model Assessment: Determine the performance of the final trained model.
6. A feature importance analysis
7. Conclusion & recommendations
```

2. Data Collection

In this step I will:

- Identify data sources
- Split the data into training and test sets

Before starting, as a first step, I will call some libraries I need in order to build my model.

In [1]:
```
import warnings
warnings.filterwarnings('ignore')
#install the necessary libraries
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.model_selection import StratifiedShuffleSplit
# import linear regression
from sklearn.linear_model import LinearRegression
# Import mean squared error
from sklearn.metrics import mean_squared_error
# Import Grid search CV
from sklearn.model_selection import GridSearchCV
# Import the SVR
from sklearn.svm import SVR
#import Random Forest Regressor
from sklearn.ensemble import RandomForestRegressor
```

Source of the data: (UCI Machine Learning Repository: Auto MPG Data Set, 2022)

In [2]:

```
# Load the data from UCI ML Repository

!wget "http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data"
```

```
--2023-03-13 10:20:36--  http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data
Resolving archive.ics.uci.edu (archive.ics.uci.edu)... 128.195.10.252
Connecting to archive.ics.uci.edu (archive.ics.uci.edu)|128.195.10.252|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 30286 (30K) [application/x-httpd-php]
Saving to: ‘auto-mpg.data.1’

auto-mpg.data.1     100%[===================>]  29.58K   187KB/s    in 0.2s

2023-03-13 10:20:37 (187 KB/s) - ‘auto-mpg.data.1’ saved [30286/30286]
```

In [3]:

```
# Using Pandas to read data from a file

attributes = ['mpg','cylinders','displacement','horsepower','weight','Speed', 'year model', 'origin']

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

In [4]:

```
# Create a copy of the original data
my_data = initial_data.copy()

# Examine my data structure and return the top 5 rows of the data frame.
my_data.head(5)
```

Out[4]:

	mpg	cylinders	displacement	horsepower	weight	Speed	year model	origin
0	18.0	8	307.0	130.0	3504.0	12.0	70	1
1	15.0	8	350.0	165.0	3693.0	11.5	70	1
2	18.0	8	318.0	150.0	3436.0	11.0	70	1
3	16.0	8	304.0	150.0	3433.0	12.0	70	1
4	17.0	8	302.0	140.0	3449.0	10.5	70	1

In [5]:

```
#Split my data into training and test sets
split = StratifiedShuffleSplit(n_splits=1, test_size=0.25, random_state=42)
for tr_ind, test_ind in split.split(my_data, my_data["cylinders"]):
```

```python
    tr_set = my_data.loc[tr_ind]
    test_set = my_data.loc[test_ind]
```

In [6]:
```python
# Segregating Target and Feature variables
data_set = tr_set.drop("mpg", axis=1)
target = tr_set["mpg"].copy()
```

In [7]:
```python
data_set.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 298 entries, 227 to 254
Data columns (total 7 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   cylinders     298 non-null    int64  
 1   displacement  298 non-null    float64
 2   horsepower    294 non-null    float64
 3   weight        298 non-null    float64
 4   Speed         298 non-null    float64
 5   year model    298 non-null    int64  
 6   origin        298 non-null    int64  
dtypes: float64(4), int64(3)
memory usage: 18.6 KB
```

3. Data Exploration

Check for Data type of columns

In [8]:
```python
# Check the info of my data
data_set.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 298 entries, 227 to 254
Data columns (total 7 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   cylinders     298 non-null    int64  
 1   displacement  298 non-null    float64
 2   horsepower    294 non-null    float64
 3   weight        298 non-null    float64
 4   Speed         298 non-null    float64
 5   year model    298 non-null    int64  
 6   origin        298 non-null    int64  
dtypes: float64(4), int64(3)
memory usage: 18.6 KB
```

4 values are missing from the variable "horsepower". As far as the formatting is concerned, nothing needs to be done.

Check for null values

In [9]:
```python
# Looking for all the null values
data_set.isnull().sum()
```

Out[9]:
```
cylinders       0
displacement    0
horsepower      4
weight          0
Speed           0
year model      0
origin          0
dtype: int64
```

It has been mentioned earlier that only **"horsepower" has four missing values** .

In [10]:
```
### Check summary statistics
data_set.describe()
```

Out[10]:

	cylinders	displacement	horsepower	weight	Speed	year model	origin
count	298.000000	298.000000	294.000000	298.000000	298.000000	298.000000	298.000000
mean	5.453020	192.489933	103.911565	2984.996644	15.671812	75.959732	1.567114
std	1.701497	101.224631	37.547953	827.999217	2.791729	3.691612	0.793827
min	3.000000	68.000000	46.000000	1755.000000	8.000000	70.000000	1.000000
25%	4.000000	105.000000	76.000000	2257.500000	13.925000	73.000000	1.000000
50%	4.000000	146.000000	93.500000	2866.500000	15.500000	76.000000	1.000000
75%	8.000000	261.500000	125.000000	3573.000000	17.300000	79.000000	2.000000
max	8.000000	455.000000	230.000000	5140.000000	24.800000	82.000000	3.000000

Look for the category distribution in categorical columns

Now I want to see the distribution to know the % of how many rows belong to a particulare class of value. To do that I will first count the number of rows for each class of value then I will devide it by the total number of rows. In my case I will do that for both "origin" & "cylinders"

In [11]:
```
## Origin distribution
data_set['origin'].value_counts()/ len(data_set)
```

Out[11]:
```
1    0.624161
3    0.191275
2    0.184564
Name: origin, dtype: float64
```

According to the results, more than 62% of the origin "1", 29% from "2" and 18% from "3".

In [12]:
```
## Cylinders distribution
data_set["cylinders"].value_counts() / len(data_set)
```

Out[12]:
```
4    0.513423
8    0.258389
6    0.211409
3    0.010067
5    0.006711
Name: cylinders, dtype: float64
```

According to the results, more than 50% of the engines are 4 cylinders, 25% are 8 cylinders, 21% are 6 cylinders, and the remaining are 3 cylinders and 5 cylinders.

My consideration of both distributions leads me to keep in mind that **while testing that most of the vehicles belong to 4 cylinders & are mostly from origin 1**

Checking correlation between different attributes

To do that I will use the function Corr of Pandas

In [13]:
```
data_set.corr().style.background_gradient(cmap="GnBu")
```

Out[13]:

	cylinders	displacement	horsepower	weight	Speed	year model	origin
cylinders	1.000000	0.951216	0.838601	0.891272	-0.514603	-0.342831	-0.562276
displacement	0.951216	1.000000	0.892009	0.932420	-0.554452	-0.373840	-0.624830
horsepower	0.838601	0.892009	1.000000	0.868459	-0.684387	-0.407914	-0.464907
weight	0.891272	0.932420	0.868459	1.000000	-0.420830	-0.302283	-0.597929
Speed	-0.514603	-0.554452	-0.684387	-0.420830	1.000000	0.272655	0.203228
year model	-0.342831	-0.373840	-0.407914	-0.302283	0.272655	1.000000	0.201992
origin	-0.562276	-0.624830	-0.464907	-0.597929	0.203228	0.201992	1.000000

This helps to understand witch are the most important features to look at when building my machine learning

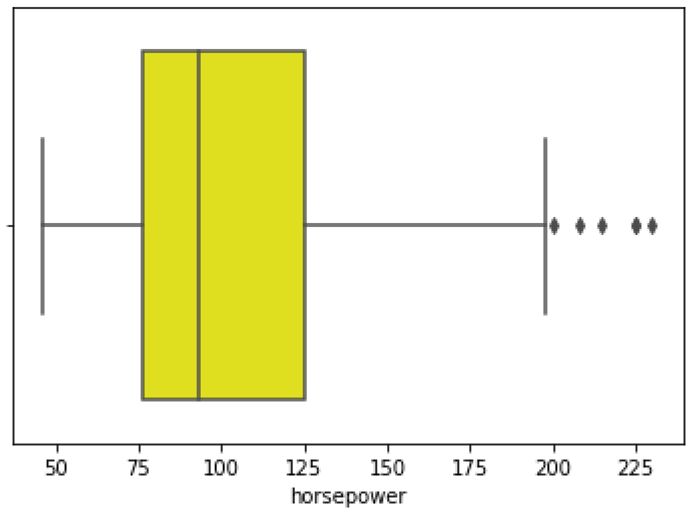
4. Data Preprocessing

Choosing the best imputation technique (mean, median or mode)is key to getting the best value from missing values. Using this value, missing values can be replaced appropriately by finding out which measures the central tendency best. **(python, 2022)**

A distribution plot or a box plot is extremely useful for determining which technique to use. For that we use the function sns.boxplot as follow

In [14]:
```
sns.boxplot(x=data_set['horsepower'], color='yellow')
```

Out[14]: <AxesSubplot:xlabel='horsepower'>



Considering there are only a few outliers, I opted to **impute null values based on the median**

In [15]:
```
# calculate the median
my_median = data_set['horsepower'].median()
```

Impute null values of "horsepower"

In [16]:
```
#impute my null values with median
data_set['horsepower'] = data_set['horsepower'].fillna(my_median)
```

In [17]:
```
# Check my new values
data_set.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 298 entries, 227 to 254
Data columns (total 7 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   cylinders     298 non-null    int64
 1   displacement  298 non-null    float64
 2   horsepower    298 non-null    float64
 3   weight        298 non-null    float64
 4   Speed         298 non-null    float64
 5   year model    298 non-null    int64
 6   origin        298 non-null    int64
dtypes: float64(4), int64(3)
memory usage: 18.6 KB
```

4. Selecting and Training Models

In this section I will train the 3 following models, train them and compare between them:

```
- Linear Regression
- Random Forest
- Support Vector Machine regressor
```

Linear Regression

In [18]:
```
linear_reg = LinearRegression()
linear_reg.fit(data_set, target)
```

Out[18]:
```
▾ LinearRegression
LinearRegression()
```

In [19]:
```
# Testing the predictions
sample_mydata = data_set.iloc[:10]
sample_target = target.iloc[:10]

print("Prediction of samples: ", linear_reg.predict(sample_mydata))
```

```
Prediction of samples:  [19.71572776 32.15336989 27.48068721 35.6183993  15.70886788 15.77215121
 13.71220266 10.75170415 14.87417604 31.1086198 ]
```

In [20]:
```
print("Actual Labels of samples: ", list(sample_target))
```

```
Actual Labels of samples:  [19.0, 39.4, 24.0, 38.0, 15.5, 11.0, 15.0, 14.0, 18.0, 34.3]
```

Calculate the Mean Squared Error

In [21]:
```
mpg_pred = linear_reg.predict(data_set)
mse_linear = mean_squared_error(target, mpg_pred)
rmse_linear = np.sqrt(mse_linear)
```

```
print('The mean squared error is for linear regression model is:')
rmse_linear
```

```
The mean squared error is for linear regression model is:
Out[21]: 3.3144856970961296
```

Cross validation for linear regression model

When Scikit-Learn performs a K-fold cross-validation, the training set is randomly split into K subsets called folds, and then the model is trained and evaluated K times, with each fold being evaluated at a different time, and each fold being trained on the following time.

The result is an array containing the scores for all K evaluations:

```
In [22]: from sklearn.model_selection import cross_val_score

#Pass linear regression model & prepare the data labels scoring method and then 10 quick k-fold cross validation
scor = cross_val_score(linear_reg, data_set, target, scoring="neg_mean_squared_error", cv = 10)
linear_reg_scor_rmse = np.sqrt(-scor)
print('The mean square error values of the 10 quick K-fold cross validations:')
linear_reg_scor_rmse
```

```
The mean square error values of the 10 quick K-fold cross validations:
Out[22]: array([3.02494035, 2.54628965, 4.30429811, 2.48921885, 3.28636331,
       2.92807517, 3.96718379, 3.74807318, 3.16317425, 3.96040375])
```

```
In [23]: # Find out the average
print('The average mean square error for Linear regression model: ')
linear_reg_scor_rmse.mean()
```

```
The average mean square error for Linear regression model:
Out[23]: 3.341802040394275
```

Random Forest model

```
In [24]: # Utilize the fit method to initiate training
regress_forst = RandomForestRegressor()
regress_forst.fit(data_set, target)

#Provide the cross value score
forest_reg_cv_scor= cross_val_score(regress_forst,
                                         data_set,
                                         target,
                                         scoring='neg_mean_squared_error',
                                         cv = 15)

# For all 10 values I have, calculate the square root of my negative values
forest_reg_rmse_scor = np.sqrt(-forest_reg_cv_scor)
```

```
In [25]: # Calculate the average
print('The average mean square error for Random Forest Regressor : ')
forest_reg_rmse_scor.mean()
```

```
The average mean square error for Random Forest Regressor :
Out[25]: 2.491128354424783
```

Random Forest performed better than the linear regression model

Support Vector Machine Regressor

In [26]:
```
# I have selected linear to map a lower dimensional data into a higher dimensional data
regr_svm = SVR(kernel='linear')
# fit the data with fit function
regr_svm.fit(data_set, target)
#cross validation
regr_svm_cv_scor = cross_val_score(regr_svm, data_set, target,
                                scoring='neg_mean_squared_error',
                                cv = 15)

rmse_scor_svm = np.sqrt(-regr_svm_cv_scor)
```

In [27]:
```
# Calculate the average
print('The average mean square error for SVMR : ')
rmse_scor_svm.mean()
```

```
The average mean square error for SVMR :
```

Out[27]:
```
4.424479835233397
```

So far we see Random Forest turns out to be the best model out of the 3. Now I will perform Hyperparameter tuning to find out which set of parameters of the random forest model works the best. So if we can improve the performane of random forest model from what we already have.

GridSearchCV for hyperparameter tuning

The hyperparameters of the random forest regressor must be fine-tuned here. In order to do so, I selected the grid search of the cyclic learns model selection module.

In [28]:
```
# define the parameter grid
prm_grid_ = [
    {'n_estimators': [2, 10, 15], 'max_features': [2, 4, 6,8]},
    {'bootstrap': [False], 'n_estimators': [4, 8], 'max_features': [2, 4, 5]},
  ]

frst_regres = RandomForestRegressor()

search_grid = GridSearchCV(frst_regres, prm_grid_,
                           scoring='neg_mean_squared_error',
                           return_train_score=True,
                           cv=10,
                          )
# Fit the data
search_grid.fit(data_set, target)
```

Out[28]:
```
▸          GridSearchCV
▸ estimator: RandomForestRegressor
      ▸ RandomForestRegressor
```

In [29]:
```
print("The best parameters we could have for Random Forest are:")
search_grid.best_params_
```

```
The best parameters we could have for Random Forest are:
```

Out[29]:
```
{'max_features': 8, 'n_estimators': 10}
```

Now we want to see which parameters had returned what scores

In [30]:
```
# Keeping track of all our scores
scor_cv = search_grid.cv_results_

# Print all the parameters along with their scores
for scor_mean, prms in zip(scor_cv['mean_test_score'], scor_cv["params"]):
    print(np.sqrt(-scor_mean), prms)
```

```
3.583679961850042 {'max_features': 2, 'n_estimators': 2}
2.744335225074773 {'max_features': 2, 'n_estimators': 10}
2.704608099267951 {'max_features': 2, 'n_estimators': 15}
3.0573984602644755 {'max_features': 4, 'n_estimators': 2}
2.7993798841229434 {'max_features': 4, 'n_estimators': 10}
2.6692158285174643 {'max_features': 4, 'n_estimators': 15}
3.2060766208639766 {'max_features': 6, 'n_estimators': 2}
2.815043302172002 {'max_features': 6, 'n_estimators': 10}
2.7587333962289127 {'max_features': 6, 'n_estimators': 15}
3.162983278234459 {'max_features': 8, 'n_estimators': 2}
2.6673712488722567 {'max_features': 8, 'n_estimators': 10}
2.7372290497890273 {'max_features': 8, 'n_estimators': 15}
2.9145130431575756 {'bootstrap': False, 'max_features': 2, 'n_estimators': 4}
2.810570030664618 {'bootstrap': False, 'max_features': 2, 'n_estimators': 8}
2.9243131868769683 {'bootstrap': False, 'max_features': 4, 'n_estimators': 4}
2.738782807737731 {'bootstrap': False, 'max_features': 4, 'n_estimators': 8}
3.0209392212912247 {'bootstrap': False, 'max_features': 5, 'n_estimators': 4}
2.8730333665961334 {'bootstrap': False, 'max_features': 5, 'n_estimators': 8}
```

I still have **my best model, the Random Forest Regressor, with a square error of 2.51.**

5. Model Assessement

In order to assess the model using the data I kept for testing. First , I must prepare it and ensure that there are no null values.

In [31]:
```
test_set.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100 entries, 364 to 69
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   mpg           100 non-null    float64
 1   cylinders     100 non-null    int64  
 2   displacement  100 non-null    float64
 3   horsepower    98 non-null     float64
 4   weight        100 non-null    float64
 5   Speed         100 non-null    float64
 6   year model    100 non-null    int64  
 7   origin        100 non-null    int64  
dtypes: float64(5), int64(3)
memory usage: 7.0 KB
```

Using the same approach I applied to the preprocessing data step, I will fill in the two missing values for the attribute horsepower.

In [32]:
```
# calculate the median
test_median = test_set['horsepower'].median()
#impute my null values with median
test_set['horsepower'] = test_set['horsepower'].fillna(test_median)
# Check my new values
test_set.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100 entries, 364 to 69
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   mpg           100 non-null    float64
 1   cylinders     100 non-null    int64  
 2   displacement  100 non-null    float64
 3   horsepower    100 non-null    float64
 4   weight        100 non-null    float64
 5   Speed         100 non-null    float64
 6   year model    100 non-null    int64  
 7   origin        100 non-null    int64  
dtypes: float64(5), int64(3)
memory usage: 7.0 KB
```

The time has come to test my model, and I have chosen the Random Forest Regressor as my model.

In [33]:
```
# capture my best model in selected model variable
selected_model = search_grid.best_estimator_


# drop the mpg from our test data
data_test = test_set.drop("mpg", axis=1)

#segregate my mpg from my testing data
target_test = test_set["mpg"].copy()
```

In [34]:
```
#Predict the result
selected_model_pr = selected_model.predict(data_test)

#calculate squared error
mse_last = mean_squared_error(target_test, selected_model_pr)
rmse_last=np.sqrt(mse_last)
```

In [35]:
```
#Print
rmse_last
```

Out[35]:
```
3.168060132005073
```

It is encouraging to see that the squared error has decreased from 2.81 to 1.27 compared to the training one.

In [36]:
```
# Testing the predictions using my test data
sample_testdata = data_test.iloc[:5]
sample_testtarget = target_test.iloc[:5]

print("Prediction of samples with the my selected model: ", selected_model.predict(sample_testdata))
```

```
Prediction of samples with the my selected model:  [17.28 41.14 13.75 27.1  15.55]
```

In [37]:
```
print("Actual Labels of samples: ", list(sample_testtarget))
```

```
Actual Labels of samples:  [26.6, 29.8, 16.0, 28.0, 13.0]
```

Based on my testing data, I consider the model chosen to be good

6. Feature importance Analysis

In [38]:
```
# calculate features importance
feature_import = search_grid.best_estimator_.feature_importances_
```

```
feature_import
```

Out[38]:
```
array([0.06128835, 0.51262105, 0.13036814, 0.12590964, 0.01868379,
       0.14588714, 0.00524188])
```

We cannot make sense of these numbers if we keep them in this manner without knowing which features they belong to. To do that, I'll combine their names with features' names

In [39]:
```
# With the reverse method, the most important feature will appear at the top and so on
print("Features importance:")
sorted(zip(attributes, feature_import), reverse=True)
```

```
Features importance:
```

Out[39]:
```
[('year model', 0.005241880421957686),
 ('weight', 0.01868379142496108),
 ('mpg', 0.061288347898734094),
 ('horsepower', 0.12590964370100546),
 ('displacement', 0.1303681447872625),
 ('cylinders', 0.5126210480252618),
 ('Speed', 0.14588714374081754)]
```

The year model appears to be the most important feature based on the results above. It is now time to evaluate our model with test data.

8. Conclusion

As a result, the machine created for the company can be an effective solution. It may not be 100% accurate, but it can be improved since the squared error was only 2.81 and on testing data we saw a significant improvement of 1.27. Thus, machine learning needs to be trained better to reduce errors. With this machine, the company can start working right away.

9.References

1. **python. [online] Available at: https://vitalflux.com/pandas-impute-missing-values-mean-median-mode/ #:~:text=When%20the%20data%20is%20skewed,be%20done%20with%20numerical%20data [Accessed 8 July 2022].**

Archive.ics.uci.edu. 2022. UCI Machine Learning Repository: Auto MPG Data Set. [online] Available at: http://archive.ics.uci.edu/ml/datasets/Auto+MPG [Accessed 5 July 2022].