Project 1: Mercedes-Benz Greener Manufacturing Report

By: Kureishi Shivanand

Course: Machine Learning

Date: February 19, 2020

Background

You are required to reduce the time that cars spend on the test bench. Others will work with a dataset representing different permutations of features in a Mercedes-Benz car to predict the time it takes to pass testing. Optimal algorithms will contribute to faster testing, resulting in lower carbon dioxide emissions without reducing Mercedes-Benz's standards.

Problem Statement

Reduce the time a Mercedes-Benz spends on the test bench

Tasks

Original Datasets

```
import pandas as pd

d_train = pd.read_csv('train.csv')
d_test = pd.read_csv('test.csv')

# Note: train and test sets contain different records (different ID). Hence, infer that data is already split
print(d_train.shape) # TRAIN SET
print(d_test.shape) # TEST SET
```

```
(4209, 378)
(4209, 377)
```

Figure 1.

```
d_train.head()
```

	ID	y	X0	X1	X2	X3	X4	X5	X6	X8	...	X375	X376	X377	X378	X379	X380	X382	X383	X384	X385
0	0	130.81	k	v	at	a	d	u	j	o	...	0	0	1	0	0	0	0	0	0	0
1	6	88.53	k	t	av	e	d	y	l	o	...	1	0	0	0	0	0	0	0	0	0
2	7	76.26	az	w	n	c	d	x	j	x	...	0	0	0	0	0	0	1	0	0	0
3	9	80.62	az	t	n	f	d	x	l	e	...	0	0	0	0	0	0	0	0	0	0
4	13	78.02	az	v	n	f	d	h	d	n	...	0	0	0	0	0	0	0	0	0	0

5 rows × 378 columns

```
d_test.head()
```

	ID	X0	X1	X2	X3	X4	X5	X6	X8	X10	...	X375	X376	X377	X378	X379	X380	X382	X383	X384	X385
0	1	az	v	n	f	d	t	a	w	0	...	0	0	0	1	0	0	0	0	0	0
1	2	t	b	ai	a	d	b	g	y	0	...	0	0	1	0	0	0	0	0	0	0
2	3	az	v	as	f	d	a	j	j	0	...	0	0	0	1	0	0	0	0	0	0
3	4	az	l	n	f	d	z	l	n	0	...	0	0	0	1	0	0	0	0	0	0
4	5	w	s	as	c	d	y	i	m	0	...	1	0	0	0	0	0	0	0	0	0

5 rows × 377 columns

Figure 2.

Figures 1 displays how train.csv and test.csv that was provided are imported to d_train and d_test, respectively. Figure 2 gives an indication of how both data frames initially look. The d_train set has 378 variables (including output/target variable: y) and d_test set has 377 variables (since does not include

target variable). Note it is inferred that the sets were pre-split since there are different IDs listed between d_train and d_test.

Question 1: Remove Zero Variance Variables

```
# QUESTION 1
# Remove Zero Variance columns (constant value columns)
# for each column, check if the values match the first one (true: no, false: yes) -> returns dataframe with boolean values
# only if all values in column are false (constant value column) -> return false (based on any() function) -> column not include
d_train = d_train.loc[:, (d_train != d_train.iloc[0]).any()]
d_test = d_test.loc[:, (d_test != d_test.iloc[0]).any()]
```

```
print(d_train.shape) # after removing 0 variance columns -> 366 features (12 variables removed)
print(d_test.shape) # after removing 0 variance columns -> 372 features (5 variables removed)
```

```
(4209, 366)
(4209, 372)
```

Figure 3.

Figure 3 answer Task 1 of removing all zero variance columns. Zero variance essentially means constant column in which all values in the column are the same [1]. An explanation of the code is given in the comments. It can be seen that by removing the zero variance variables, d_train now has 12 less variables and d_test now has 5 less variables.

Question 2: Check for null and unique values for test and train sets

```
# QUESTION 2
d_train.isnull().sum() # no null values
```

```
ID      0
y       0
X0      0
X1      0
X2      0
       ..
X380    0
X382    0
X383    0
X384    0
X385    0
Length: 366, dtype: int64
```

```
d_train.nunique() # Binary values for features (X10-X385) to indicate whether have it (1) or not (0)
```

```
ID      4209
y       2545
X0        47
X1        27
X2        44
        ...
X380       2
X382       2
X383       2
X384       2
X385       2
Length: 366, dtype: int64
```

Figure 4.

```
d_test.isnull().sum() # no null values
```

```
ID      0
X0      0
X1      0
X2      0
X3      0
       ..
X380    0
X382    0
X383    0
X384    0
X385    0
Length: 372, dtype: int64
```

```
d_test.nunique() # Binary values for features (X10-X385) to indicate whether have it (1) or not (0)
```

```
ID      4209
X0        49
X1        27
X2        45
X3         7
        ...
X380       2
X382       2
X383       2
X384       2
X385       2
Length: 372, dtype: int64
```

Figure 5.

Figure 4 shows that there are no null/missing values in the training set (d_train). It also shows the number of unique variables in the set. Note that since there is a massive number of unique values for the target value, it is inferred to be continuous. Also note that from X0 – X8, the alphabet is used to represent categorical values. From X10 – X385, binary values are used to indicate whether the vehicle passes with the features or not. Similar results are achieved for the testing set (d_test) in Figure 5, excluding the target variable.

Question 3: Apply label encoder (on both training and testing sets)

```
# QUESTION 3
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder() # LabelEncoder would label each unique value of a variable a specific number

# use only for features which contain letters since X10 - X385 are binary values already
d_train['X0'] = le.fit_transform(d_train['X0'])
d_train['X1'] = le.fit_transform(d_train['X1'])
d_train['X2'] = le.fit_transform(d_train['X2'])
d_train['X3'] = le.fit_transform(d_train['X3'])
d_train['X4'] = le.fit_transform(d_train['X4'])
d_train['X5'] = le.fit_transform(d_train['X5'])
d_train['X6'] = le.fit_transform(d_train['X6'])
d_train['X8'] = le.fit_transform(d_train['X8'])

d_test['X0'] = le.fit_transform(d_test['X0'])
d_test['X1'] = le.fit_transform(d_test['X1'])
d_test['X2'] = le.fit_transform(d_test['X2'])
d_test['X3'] = le.fit_transform(d_test['X3'])
d_test['X4'] = le.fit_transform(d_test['X4'])
d_test['X5'] = le.fit_transform(d_test['X5'])
d_test['X6'] = le.fit_transform(d_test['X6'])
d_test['X8'] = le.fit_transform(d_test['X8'])
```

Figure 6.

```
d_train.head()
```

	ID	y	X0	X1	X2	X3	X4	X5	X6	X8	...	X375	X376	X377	X378	X379	X380	X382	X383	X384	X385
0	0	130.81	32	23	17	0	3	24	9	14	...	0	0	1	0	0	0	0	0	0	0
1	6	88.53	32	21	19	4	3	28	11	14	...	1	0	0	0	0	0	0	0	0	0
2	7	76.26	20	24	34	2	3	27	9	23	...	0	0	0	0	0	0	1	0	0	0
3	9	80.62	20	21	34	5	3	27	11	4	...	0	0	0	0	0	0	0	0	0	0
4	13	78.02	20	23	34	5	3	12	3	13	...	0	0	0	0	0	0	0	0	0	0

5 rows × 366 columns

```
d_test.head()
```

	ID	X0	X1	X2	X3	X4	X5	X6	X8	X10	...	X375	X376	X377	X378	X379	X380	X382	X383	X384	X385
0	1	21	23	34	5	3	26	0	22	0	...	0	0	0	1	0	0	0	0	0	0
1	2	42	3	8	0	3	9	6	24	0	...	0	0	1	0	0	0	0	0	0	0
2	3	21	23	17	5	3	0	9	9	0	...	0	0	0	1	0	0	0	0	0	0
3	4	21	13	34	5	3	31	11	13	0	...	0	0	0	1	0	0	0	0	0	0
4	5	45	20	17	2	3	30	8	12	0	...	1	0	0	0	0	0	0	0	0	0

5 rows × 372 columns

Figure 7.

The label encoder is applied on only non-binary categorical features (X0-X8). This is done since X10-X385 are already stated in binary number (0/1). The letters used prior to indicate values are replaced with a corresponding number. This usually makes the model run more efficient since most algorithms used numbers instead of letters when generating outputs, is required in PCA. Figure 6 shows the label encoder was applied on both d_train and d_test so the values can be standardized in the model while fitting and predicting. Figure 7 shows the new values as numbers instead of letters. Note that LabelEncoder was used instead of OneHotEncoder with dummy variables, since the point is to reduce the current-massive number of features. Hence a single number representing each unique categorical value is a better choice than a new variable for each unique value. This is what I thought made the most logical sense.

Question 4: Perform dimensionality reduction (PCA on both training and testing sets)

```
# QUESTION 4
d_train_Y = d_train.loc[:, 'y'] # not include ID since doesn't give important information
d_train_X = d_train.loc[:, 'X0':'X385'] # not include ID since doesn't give important information
```

```
# use PCA (Principal Component Analysis) as dimension reduction technique to reduce features
from sklearn.decomposition import PCA
sklearn_pca = PCA(n_components=0.95) # feature reduction such that 95% variance is explained
sklearn_pca.fit(d_train_X)
d_train_X_transformed = sklearn_pca.transform(d_train_X)
```

```
d_train_X_transformed.shape # reduced down to 6 features
```

```
(4209, 6)
```

```
d_test_X = d_test.loc[:, 'X0':'X385']
sklearn_pca.fit(d_test_X)
d_test_X_transformed = sklearn_pca.transform(d_test_X)
```

```
d_test_X_transformed.shape # reduced down to 6 features
```

```
(4209, 6)
```

Figure 8.

Figure 8 demonstrates dimensionality reduction using the PCA technique on the training and testing sets. Firstly, the features of d_train were assigned to variable d_train_X and the target variable to d_train_Y. Then PCA was used to reduce the dimensionality of the features of d_train in a way that 95% of the variance would still be retained. This was similarly done for d_test as well. The dimensions for both training and testing features were reduced to 6 (drastic reduction compared to before). This would help immensely in training the models.

Question 5: Predict your test_df (from test.csv) values using XGBoost

```
# QUESTION 5
from sklearn import model_selection
from xgboost import XGBRegressor # use Regressor since y (target variable) is continuous
import xgboost as xgb

seed = 7
num_trees = 30 # ensemble of decision trees used to achieve more accurate and efficient model
reg = XGBRegressor(objective ='reg:linear', n_estimators=num_trees, random_state=seed)

reg.fit(d_train_X_transformed, d_train_Y) # XGBRegressor trained on train.csv
```

```
D:\ProgramData\Anaconda3\lib\site-packages\xgboost\core.py:587: FutureWarning: Series.base is deprecated and will be removed in
a future version
  if getattr(data, 'base', None) is not None and \
D:\ProgramData\Anaconda3\lib\site-packages\xgboost\core.py:588: FutureWarning: Series.base is deprecated and will be removed in
a future version
  data.base is not None and isinstance(data, np.ndarray) \
```

```
[12:05:48] WARNING: C:/Jenkins/workspace/xgboost-win64_release_0.90/src/objective/regression_obj.cu:152: reg:linear is now depr
ecated in favor of reg:squarederror.

XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=1, gamma=0,
             importance_type='gain', learning_rate=0.1, max_delta_step=0,
             max_depth=3, min_child_weight=1, missing=None, n_estimators=30,
             n_jobs=1, nthread=None, objective='reg:linear', random_state=7,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
             silent=None, subsample=1, verbosity=1)
```

Figure 9.

```
d_pred_Y = reg.predict(d_test_X_transformed) # predicted target/output value for test.csv
```

```
d_pred_Y # predicted output value for test set
```

```
array([ 79.23512 ,  93.708176,  96.23384 , ...,  99.30209 , 103.07663 ,
        93.89882 ], dtype=float32)
```

```
# since no y (target) values in test.csv, can't calculate model accuracy. However, the central tendencies of d_train_Y
# and d_pred_Y are similar. Hence, it can be inferred that the model is working well to some degree
```

Figure 10.

Figure 9 displays the code that was used to program the model to predict test values. The XGBRegressor was used since the target variable: y is continuous. Seed is the random state which is used to get reproducible results. There are 30 estimators (num_trees) used in the ensemble to create a more efficient model. A linear regression is used as the objective since the model is used to predict a continuous variable not a categorical or binary one, as mentioned before [2]. Since the training and testing set were already separated (provided in the assignment), the entire training set is used to train the model, with the transformed X (d_train_X_transformed) as features and d_train_Y as target. Figure 10 shows the transformed features from the testing set (d_test_X_transformed) is used to predict output values. The second output in Figure 10 shows an array of predicted outcomes corresponding to the test features. Note:

the task called for prediction to be done of the test.csv dataset, however since this did not contain a target variable, the accuracy of the model cannot definitively be determined.

This completes all tasks that were required of the project; however, I feel the accuracy and efficiency of the model generated still needs to be measured. To do this, I will run some comparative tests in the next section, 'Further Insight', to have a more complete perspective of the generated results.

Further Insight

```
# FURTHER INSIGHT
# to truly test the accuracy of the model, the train.csv dataset will be used (since contains target variable as well)
# Dmatrix (supported by XGBoost) increases efficiency of training. assign X and Y
data_matrix = xgb.DMatrix(data=d_train_X_transformed, label=d_train_Y)
```

```
# cv() is k-fold method supported by XGBoost
params={'objective':'reg:linear'}
cv_results = xgb.cv(dtrain=data_matrix, params=params, nfold=4, num_boost_round=num_trees, metrics='rmse'
                    ,as_pandas=True, seed=seed)
```

```
[12:06:01] WARNING: C:/Jenkins/workspace/xgboost-win64_release_0.90/src/objective/regression_obj.cu:152: reg:linear is now depr
ecated in favor of reg:squarederror.
[12:06:01] WARNING: C:/Jenkins/workspace/xgboost-win64_release_0.90/src/objective/regression_obj.cu:152: reg:linear is now depr
ecated in favor of reg:squarederror.
[12:06:01] WARNING: C:/Jenkins/workspace/xgboost-win64_release_0.90/src/objective/regression_obj.cu:152: reg:linear is now depr
ecated in favor of reg:squarederror.
[12:06:01] WARNING: C:/Jenkins/workspace/xgboost-win64_release_0.90/src/objective/regression_obj.cu:152: reg:linear is now depr
ecated in favor of reg:squarederror.
```

```
cv_results.head() # shows training and testing mean/std after each fold
```

	train-rmse-mean	train-rmse-std	test-rmse-mean	test-rmse-std
0	71.185606	0.138558	71.189764	0.508571
1	50.477693	0.122056	50.509084	0.539331
2	36.170735	0.125883	36.246786	0.601516
3	26.393880	0.150877	26.529490	0.700408
4	19.818660	0.188345	20.066635	0.777701

Figure 11.

```
# extract final boosting round (Round 30) metric
print((cv_results['test-rmse-mean']).tail(1)) # can get lower depending on parameters chosen,
```

```
29    10.225727
Name: test-rmse-mean, dtype: float64
```

```
# testing accuracy of original model (XGBRegressor object: reg) on train set (since includes features and target)
# using same dataset (train.csv) to determine accuracy of XGBRegressor model by splitting into train/test (80:20)
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(d_train_X_transformed, d_train_Y, test_size=0.2, random_state=seed)

reg.fit(X_train, Y_train)

pred = reg.predict(X_test)
```

```
[12:06:06] WARNING: C:/Jenkins/workspace/xgboost-win64_release_0.90/src/objective/regression_obj.cu:152: reg:linear is now depr
ecated in favor of reg:squarederror.
```

```
import numpy as np
from sklearn.metrics import mean_squared_error # use mean-squared-error since target variable is continuous, not categorical

rmse = np.sqrt(mean_squared_error(Y_test, pred))
print(rmse)
```

```
12.768666721302612
```

Figure 12.

```
# Note: A better (lower) RMSE is achieved when using kfold cross validation to train and test than the base XGBRegressor

# to show that the preprocessing of the dataset (test bench) has decreased, track how long training takes
import time
start_time = time.time()
reg.fit(d_train_X_transformed, d_train_Y) # transformed training set
print('Process time:',time.time() - start_time)
```

```
[12:06:10] WARNING: C:/Jenkins/workspace/xgboost-win64_release_0.90/src/objective/regression_obj.cu:152: reg:linear is now depr
ecated in favor of reg:squarederror.
Process time: 0.0848233699798584
```

```
D:\ProgramData\Anaconda3\lib\site-packages\xgboost\core.py:587: FutureWarning: Series.base is deprecated and will be removed in
a future version
  if getattr(data, 'base', None) is not None and \
D:\ProgramData\Anaconda3\lib\site-packages\xgboost\core.py:588: FutureWarning: Series.base is deprecated and will be removed in
a future version
  data.base is not None and isinstance(data, np.ndarray) \
```

```
start_time = time.time()
reg.fit(d_train_X, d_train_Y) # un-transformed training set (however, still with zero variance variables removed)
print('Process time:',time.time() - start_time) # 15 times longer with using non-transformed features
```

```
[12:06:14] WARNING: C:/Jenkins/workspace/xgboost-win64_release_0.90/src/objective/regression_obj.cu:152: reg:linear is now depr
ecated in favor of reg:squarederror.
Process time: 1.3115425109863281
```

Figure 13.

In order to measure the model accuracy, the target variable needs to be present. The only dataset that was provided which contains the target variable was train.csv. Hence, this will be used to conduct both training and testing. The features will be d_train_X_transformed and the target will be d_train_Y. In Figure 11, cross-validation (K-Fold) is used to training and test on the dataset using 4 folds. The cv() method is the cross-validation method supported by XGBoost [2]. The DMatrix data structure (data_matrix) is supported by XGBoost and increases the model efficiency during training [2]. The data_matrix is assigned to the dtrain parameter. The parameters (params) are given as a dictionary indicating that linear regression will be used. Number of estimators is same as before (30). To measure the model accuracy, the RMSE value will be used since the predicted value is continuous and numeric. The output of the model (cv_result) displays the means of the train and test sets of each fold for 30 rounds. The final mean after boosting of the test set was 10.226. This is not a bad result, however a lower RMSE may be achieved with manipulating the parameters.

The next test conducted was splitting the d_train set into a train and test set (using train_test_split module) since this set also contained the target variable. The training contained 80% of the dataset, while the testing contained 20%. After fitting the original regressor model (Figure 9) with this new set of training data and predicting on the test features, the RMSE turned out to be 12.769, which is higher than when using K-Fold cross validation. Hence, it is concluded that to attain a more accurate and efficient model, cross validation should be used (with XGBoost) rather than the base regressor provided by XGBoost.

The whole point of the problem statement was trying to find a solution to decrease the time a Mercedes-Benz vehicle spends on the test bench. To see that the solution achieves this. The system time was recorded before and after the model was trained with the transformed dataset and the untransformed one (although this set still had the zero variance columns removed, so the actual time may be a bit higher). The time to train the transformed data was approximately 0.085 seconds, while the time to trained the untransformed data was 1.312 seconds. This is a drastic difference and the times recorded may have something to do with the system. However, the point is there is a clear decrease in the time took (according to the recorded data, 1/15 of the time). Hence, the overall objective has been definitely achieved.

Final Remarks and Conclusion

Throughout this project, the original datasets were manipulated multiple times in order to achieve faster training of the predictor model as well as testing new vehicles for required features. Dimensionality Reduction provided an immense decrease in features, while still maintain majority of the variance within the model (95%). The XGBoost ensemble library was used to create a more efficient and accurate model by using multiple estimators then taking the average of the output, instead of just one. This is proven to provide a more accurate result as seen above.

In conclusion, the model generated achieves a faster training (as well as testing) of the transformed data, when compared to the original dataset. Hence, the objective for the project has been achieved and Mercedes-Benz will be able to spend less time on the test bench.

References:

[1] pandas dataframe remove constant column (February 19, 2020). *Stack Overflow*. Retrieved from https://stackoverflow.com/questions/20209600/pandas-dataframe-remove-constant-column

[2] Using XGBoost in Python (February 19, 2020). *DataCamp*. Retrieved from https://www.datacamp.com/community/tutorials/xgboost-in-python#apply