Mercedes-Benz Greener Manufacturing

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

[2]:
```
train = pd.read_csv('train.csv', index_col = 'ID')
test = pd.read_csv('test.csv', index_col = 'ID')
```

[3]:
```
train.head()
```

[3]:
```
          y  X0 X1  X2 X3 X4 X5 X6 X8  X10  …  X375  X376  X377  X378  X379  \
ID                                          …
0   130.81   k  v  at  a  d  u  j  o    0  …     0     0     1     0     0
6    88.53   k  t  av  e  d  y  l  o    0  …     1     0     0     0     0
7    76.26  az  w   n  c  d  x  j  x    0  …     0     0     0     0     0
9    80.62  az  t   n  f  d  x  l  e    0  …     0     0     0     0     0
13   78.02  az  v   n  f  d  h  d  n    0  …     0     0     0     0     0

    X380  X382  X383  X384  X385
ID
0      0     0     0     0     0
6      0     0     0     0     0
7      0     1     0     0     0
9      0     0     0     0     0
13     0     0     0     0     0

[5 rows x 377 columns]
```

Replacing strings with numbers in train and test dataframes. Note that a combined list of all unique strings is prepared for each feature (containing string) for both train and test data before replacing it with numbers. This is done to ensure that each strings gets mapped to same number for both train and test data.

[4]:
```
for col in train.columns:
    if(train[col].dtype != np.float64 and train[col].dtype != np.int64):

        # making a list of unique strings in train and test feature
```

```
        unique_train = train[col].unique().tolist()
        unique_test = test[col].unique().tolist()

        # making a combined list
        for member in unique_test:
            if member not in unique_train:
                unique_train.append(member)

        # mapping with numbers
        map_dict = dict(zip(unique_train, range(len(unique_train))))
        train[col] = train[col].replace(to_replace = map_dict)
        test[col] = test[col].replace(to_replace = map_dict)
```

[5]:
```
train.head()
```

[5]:
```
         y  X0  X1  X2  X3  X4  X5  X6  X8  X10  …  X375  X376  X377  X378  \
ID                                               …
0   130.81   0   0   0   0   0   0   0   0    0  …     0     0     1     0
6    88.53   0   1   1   1   0   1   1   0    0  …     1     0     0     0
7    76.26   1   2   2   2   0   2   0   1    0  …     0     0     0     0
9    80.62   1   1   2   3   0   2   1   2    0  …     0     0     0     0
13   78.02   1   0   2   3   0   3   2   3    0  …     0     0     0     0

    X379  X380  X382  X383  X384  X385
ID
0      0     0     0     0     0     0
6      0     0     0     0     0     0
7      0     0     1     0     0     0
9      0     0     0     0     0     0
13     0     0     0     0     0     0

[5 rows x 377 columns]
```

Checking if train or test data has any NaN value. Also, getting summary of data

[6]:
```
print(train.isnull().values.any())
print(test.isnull().values.any())
train.describe()
```

```
False
False
```

[6]:
```
                 y           X0           X1           X2           X3  \
count  4209.000000  4209.000000  4209.000000  4209.000000  4209.000000
mean    100.669318    12.110715     6.467569     7.851509     2.415301
std      12.679381     8.315637     4.789927     5.644031     1.361654
min      72.110000     0.000000     0.000000     0.000000     0.000000
```

```
25%     90.820000     6.000000     3.000000     4.000000     2.000000
50%     99.150000    11.000000     6.000000     7.000000     2.000000
75%    109.010000    15.000000     7.000000    10.000000     3.000000
max    265.320000    46.000000    26.000000    43.000000     6.000000

                X4           X5           X6           X8          X10  …  \
count  4209.000000  4209.000000  4209.000000  4209.000000  4209.000000  …
mean      0.002138    16.839629     3.031124    11.457591     0.013305  …
std       0.073900     6.357474     2.554581     7.040194     0.114590  …
min       0.000000     0.000000     0.000000     0.000000     0.000000  …
25%       0.000000    11.000000     1.000000     5.000000     0.000000  …
50%       0.000000    17.000000     2.000000    12.000000     0.000000  …
75%       0.000000    22.000000     6.000000    17.000000     0.000000  …
max       3.000000    28.000000    11.000000    24.000000     1.000000  …

              X375         X376         X377         X378         X379  \
count  4209.000000  4209.000000  4209.000000  4209.000000  4209.000000
mean      0.318841     0.057258     0.314802     0.020670     0.009503
std       0.466082     0.232363     0.464492     0.142294     0.097033
min       0.000000     0.000000     0.000000     0.000000     0.000000
25%       0.000000     0.000000     0.000000     0.000000     0.000000
50%       0.000000     0.000000     0.000000     0.000000     0.000000
75%       1.000000     0.000000     1.000000     0.000000     0.000000
max       1.000000     1.000000     1.000000     1.000000     1.000000

              X380         X382         X383         X384         X385
count  4209.000000  4209.000000  4209.000000  4209.000000  4209.000000
mean      0.008078     0.007603     0.001663     0.000475     0.001426
std       0.089524     0.086872     0.040752     0.021796     0.037734
min       0.000000     0.000000     0.000000     0.000000     0.000000
25%       0.000000     0.000000     0.000000     0.000000     0.000000
50%       0.000000     0.000000     0.000000     0.000000     0.000000
75%       0.000000     0.000000     0.000000     0.000000     0.000000
max       1.000000     1.000000     1.000000     1.000000     1.000000

[8 rows x 377 columns]
```

Splitting features and labels. Also, performing min_max scalling on features

[7]:
```
X_train = train.iloc[:,1:]
y_train = train.iloc[:,0]

from sklearn.preprocessing import MinMaxScaler
scalling = MinMaxScaler().fit(X_train)
X_train_scalled = scalling.transform(X_train)
test_scalled = scalling.transform(test)
```

Regression with linear model. Cross-validation score shows that the linear model

performs very poorly.

[8]:
```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

reg = LinearRegression()
print(cross_val_score(reg, X_train_scalled, y_train, cv=10))
```

```
[-1.33042147e+23 -1.91338315e+22 -4.94938240e+22 -1.33714229e+24
 -9.64726715e+22 -2.99456830e+21 -4.23383313e+23 -1.81628722e+23
 -6.40148587e+22 -7.38634788e+23]
```

Regression with Lasso model (L1 regularization). As the number of features are very large, Lasso regularization would assign lesser weights to non-important features and in-turn reduce their contributation in the final regression model.

[9]:
```
from sklearn.linear_model import Lasso
from sklearn.model_selection import GridSearchCV

grid_values = {'alpha': [0.0235, 0.024, 0.0245]}
grid_lasso = GridSearchCV(Lasso(), param_grid = grid_values, cv=10, scoring =␣
 ↪'r2')
grid_lasso.fit(X_train_scalled, y_train)
predict_lasso = grid_lasso.predict(test_scalled)

print('Mean score matrix: ', grid_lasso.cv_results_['mean_test_score'])
print('Grid best parameter (max. accuracy): ', grid_lasso.best_params_)
print('Grid best score (accuracy): ', grid_lasso.best_score_)
```

```
Mean score matrix:  [0.56300014 0.56300502 0.56299962]
Grid best parameter (max. accuracy):  {'alpha': 0.024}
Grid best score (accuracy):  0.5630050235597193
```

Let's also try Ridge regression (L2 regularization)

[15]:
```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

grid_values = {'alpha': [40, 40.5, 41]}
grid_ridge = GridSearchCV(Ridge(), param_grid = grid_values, cv=10, scoring =␣
 ↪'r2')
grid_ridge.fit(X_train_scalled, y_train)
predict_ridge = grid_ridge.predict(test_scalled)

print('Mean score matrix: ', grid_ridge.cv_results_['mean_test_score'])
print('Grid best parameter (max. accuracy): ', grid_ridge.best_params_)
print('Grid best score (accuracy): ', grid_ridge.best_score_)
```

```
Mean score matrix:  [0.55378705 0.55378761 0.55378743]
```

```
Grid best parameter (max. accuracy):  {'alpha': 40.5}
Grid best score (accuracy):  0.5537876100313096
```

Regression with Xgboost. It shows the best R2 score. We will use this model to do final predictions.

[16]:
```
pip install xgboost
```

```
Requirement already satisfied: xgboost in c:\users\91805\anaconda1\lib\site-
packages (1.7.5)
Requirement already satisfied: scipy in c:\users\91805\anaconda1\lib\site-
packages (from xgboost) (1.9.1)
Requirement already satisfied: numpy in c:\users\91805\anaconda1\lib\site-
packages (from xgboost) (1.21.5)
Note: you may need to restart the kernel to use updated packages.
```

[17]:
```
import xgboost as xgb
```

[18]:
```
conda install -c conda-forge xgboost
```

```
Collecting package metadata (current_repodata.json): …working… done
Solving environment: …working… done

# All requested packages already installed.


Note: you may need to restart the kernel to use updated packages.
```

[ ]:
```
grid_values = {'n_estimators': [100,105,106], 'learning_rate': [0.13,0.135,0.
 ↪14], 'max_depth': [1,2,3]}
grid_xgb = GridSearchCV(xgb.XGBRegressor(), param_grid = grid_values, cv=10,␣
 ↪scoring = 'r2')
grid_xgb.fit(X_train_scalled, y_train)
predict_xgb = grid_xgb.predict(test_scalled)

print('Mean score matrix: ', grid_xgb.cv_results_['mean_test_score'])
print('Grid best parameter (max. accuracy): ', grid_xgb.best_params_)
print('Grid best score (accuracy): ', grid_xgb.best_score_)
```

Mean score matrix: [0.55597401 0.55620147 0.55675942 0.58165304 0.5818088 0.58176478 0.57938243 0.57941404 0.57938819 0.55721329 0.55745941 0.55774666 0.58204994 0.58214103 0.58202328 0.57727201 0.57720783 0.57713946 0.55807455 0.55816391 0.55836724 0.58153668 0.58149164 0.58166668 0.57805699 0.57808365 0.57817218]

Grid best parameter (max. accuracy): {'learning_rate': 0.135, 'max_depth': 2, 'n_estimators': 75}

Grid best score (accuracy): 0.582141029072

[ ]:
```python
final_predictions = pd.DataFrame()
final_predictions['id'] = test.index
final_predictions['y'] = pd.Series(predict_xgb)
final_predictions.to_csv('predictions.csv', index=False)
```