Combined Cycle Power Plant

Ângelo Ferreira, Naomi Torres

December 12, 2022

1 Data Set Information

From the website provided: The dataset contains 9568 data points collected from a Combined Cycle Power Plant over 6 years (2006-2011), when the power plant was set to work with full load. Features consist of hourly average ambient variables Temperature (T), Ambient Pressure (AP), Relative Humidity (RH) and Exhaust Vacuum (V) to predict the net hourly electrical energy output (EP) of the plant.

A combined cycle power plant (CCPP) is composed of gas turbines (GT), steam turbines (ST) and heat recovery steam generators. In a CCPP, the electricity is generated by gas and steam turbines, which are combined in one cycle, and is transferred from one turbine to another. While the Vacuum is colected from and has effect on the Steam Turbine, the other three of the ambient variables effect the GT performance.

For comparability with our baseline studies, and to allow 5x2 fold statistical tests be carried out, we provide the data shuffled five times. For each shuffling 2-fold CV is carried out and the resulting 10 measurements are used for statistical testing. We provide the data both in .ods and in .xlsx formats.

2 Approach taken

The last part of the information was found irrelevant, thus, although the data was distributed through five different sheets, it was all joined in one single sheet (since the row entries were shuffled, it was found to be no problem to join all the entries/registers together). The goal with that is to be able to handle the dataset more easily.

In this project, it will be studied how the hourly average ambient variables T, AP, RH and V correlate with the EP of the power plant. The dataset will thus be divided in training and test sets. Different regression algorithms will be tried out in the training datasets and trained algorithms will be tested in the test dataset using accuracy as the metric.

3 Dataset import and feature correlation analysis

[1]:
```
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.linear_model import LinearRegression, Ridge, Lasso
```

```
from sklearn import preprocessing
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import KBinsDiscretizer
```

[2]:
```
df=pd.read_csv('data.csv')
```

[3]:
```
print(df.head())
print('\nDataframe shape (rows, columns): ', df.shape)
```

```
      AT      V       AP     RH      PE
0  14.96  41.76  1024.07  73.17  463.26
1  25.18  62.96  1020.04  59.08  444.37
2   5.11  39.40  1012.16  92.14  488.56
3  20.86  57.32  1010.24  76.64  446.48
4  10.82  37.50  1009.23  96.62  473.90

Dataframe shape (rows, columns):  (47840, 5)
```

[4]:
```
corrMatrix=df.corr().abs()
print(corrMatrix)
```

```
          AT         V        AP        RH        PE
AT  1.000000  0.844107  0.507549  0.542535  0.948128
V   0.844107  1.000000  0.413502  0.312187  0.869780
AP  0.507549  0.413502  1.000000  0.099574  0.518429
RH  0.542535  0.312187  0.099574  1.000000  0.389794
PE  0.948128  0.869780  0.518429  0.389794  1.000000
```

From the correlation matrix it is seen that the variable most correlated to the electrical output is the AT closely followed by V. Therefore, we can regard both RH and PE as less important to determine the value of PE, shifting the focus to AT and V.

Moreover it is also seen that V is very correlated to AT. We could say as a rough estimate that training a regression on AT values only to determine PE could be enough to predict PE well enough, since V is very correlated to AT and AT to PE.

4 Creating different arrays for different feature combinations

Here the data will be separated into 5 numpy arrays containing the values of our variables of interest (AT, V, AT+V, AT+V+RH+PE and PE). AT will correspond to xt, V to xv, AT+V to xtv, AT+V+AP+RH (ALL) to x and PE to y.

[5]:
```
xt=df[['AT']].to_numpy()
xv=df[['V']].to_numpy()
xtv=df[['AT','V']].to_numpy()
x=df[['AT','V','AP','RH']].to_numpy()
```

```
y=df['PE'].values
```

Then, 4 different training and testing sets are created, one for AT/PE (xt/y), for V/PE (xv/y), AT+V/PE (xtv/y) and one using all features and PE (x/y)

[6]:
```
xt_train, xt_test, yt_train, yt_test = train_test_split(xt, y, random_state=42)␣
  ↪#0.75/0.25 (test/train) by default
xv_train, xv_test, yv_train, yv_test = train_test_split(xv, y, random_state=43)
xtv_train, xtv_test, ytv_train, ytv_test = train_test_split(xtv, y,␣
  ↪random_state=44)
x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=45)
```

5 Linear regression

[7]:
```
lrt = LinearRegression().fit(xt_train, yt_train)
lrv = LinearRegression().fit(xv_train, yv_train)
lrtv= LinearRegression().fit(xtv_train, ytv_train)
lr= LinearRegression().fit(x_train, y_train)
```

[8]:
```
#print("Linear regression coefficient (AT/PE): {}".format(lrt.coef_))
#print("Linear regression intercept (AT/PE): {}".format(lrt.intercept_))
print("Training set score (AT/PE): {:.2f}".format(lrt.score(xt_train,␣
  ↪yt_train)))
print("Test set score (AT/PE): {:.2f}".format(lrt.score(xt_test, yt_test)))
#print("Linear regression coefficient (V/PE): {}".format(lrv.coef_))
#print("Linear regression intercept (V/PE): {}".format(lrv.intercept_))
print("Training set score (V/PE): {:.2f}".format(lrv.score(xv_train, yv_train)))
print("Test set score (V/PE): {:.2f}".format(lrv.score(xv_test, yv_test)))
#print("Linear regression coefficient (AT+V/PE): {}".format(lrtv.coef_))
#print("Linear regression intercept (AT+V/PE): {}".format(lrtv.intercept_))
print("Training set score (AT+V/PE): {:.2f}".format(lrtv.score(xtv_train,␣
  ↪ytv_train)))
print("Test set score (AT+V/PE): {:.2f}".format(lrtv.score(xtv_test, ytv_test)))
#print("Linear regression coefficient (ALL/PE): {}".format(lr.coef_))
#print("Linear regression intercept (ALL/PE): {}".format(lr.intercept_))
print("Training set score (ALL/PE): {:.2f}".format(lr.score(x_train, y_train)))
print("Test set score (ALL/PE): {:.2f}".format(lr.score(x_test, y_test)))
```

```
Training set score (AT/PE): 0.90
Test set score (AT/PE): 0.90
Training set score (V/PE): 0.76
Test set score (V/PE): 0.75
Training set score (AT+V/PE): 0.92
Test set score (AT+V/PE): 0.92
Training set score (ALL/PE): 0.93
```

```
Test set score (ALL/PE): 0.93
```

Reasonable performance for AT, but the same cannot be said for V. Using the linear regression using both features at the same time improves the result but the best result is obtained when all four features are used to train the algorithm.

A few other tests will be used.

5.1 Ridge

[9]:
```
rt = Ridge(alpha=0).fit(xt_train, yt_train) #alpha=1 by default
rv = Ridge(alpha=0).fit(xv_train, yv_train)
rtv = Ridge(alpha=0).fit(xtv_train, ytv_train)
r = Ridge(alpha=0).fit(x_train, y_train)
```

[10]:
```
print("Training set score (AT/PE): {:.2f}".format(rt.score(xt_train, yt_train)))
print("Test set score (AT/PE): {:.2f}".format(rt.score(xt_test, yt_test)))
print("Training set score (V/PE): {:.2f}".format(rv.score(xv_train, yv_train)))
print("Test set score (V/PE): {:.2f}".format(rv.score(xv_test, yv_test)))
print("Training set score (AT+V/PE): {:.2f}".format(rtv.score(xtv_train,␣
 ↪ytv_train)))
print("Test set score (AT+V/PE): {:.2f}".format(rtv.score(xtv_test, ytv_test)))
print("Training set score (ALL/PE): {:.2f}".format(r.score(x_train, y_train)))
print("Test set score (ALL/PE): {:.2f}".format(r.score(x_test, y_test)))
```

```
Training set score (AT/PE): 0.90
Test set score (AT/PE): 0.90
Training set score (V/PE): 0.76
Test set score (V/PE): 0.75
Training set score (AT+V/PE): 0.92
Test set score (AT+V/PE): 0.92
Training set score (ALL/PE): 0.93
Test set score (ALL/PE): 0.93
```

Regularization doesn't make any difference here since the amount of data provided to the model is sufficiently high (different alphas were tested showing no effect). For the same reason (having more than enough data), both ridge and linear regression have the same performance. This also adds that, given a high amount of data, it becomes harder for a linear model to overfit.

5.2 Lasso

[11]:
```
lt = Lasso().fit(xt_train, yt_train)
lv = Lasso().fit(xv_train, yv_train)
ltv = Lasso().fit(xtv_train, ytv_train)
l = Lasso().fit(x_train, y_train)
```

[12]:
```
print("Training set score (AT/PE): {:.2f}".format(lt.score(xt_train, yt_train)))
print("Test set score (AT/PE): {:.2f}".format(lt.score(xt_test, yt_test)))
print("Number of features used (AT/PE): {}".format(np.sum(lt.coef_ != 0)))
```

```
print("\nTraining set score (V/PE): {:.2f}".format(lv.score(xv_train,␣
 ↪yv_train)))
print("Test set score (V/PE): {:.2f}".format(lv.score(xv_test, yv_test)))
print("Number of features used (V/PE): {}".format(np.sum(lv.coef_ != 0)))
print("\nTraining set score (AT+V/PE): {:.2f}".format(ltv.score(xtv_train,␣
 ↪ytv_train)))
print("Test set score (AT+V/PE): {:.2f}".format(ltv.score(xtv_test, ytv_test)))
print("Number of features used (AT+V/PE): {}".format(np.sum(ltv.coef_ != 0)))
print("\nTraining set score (ALL/PE): {:.2f}".format(l.score(x_train, y_train)))
print("Test set score (ALL/PE): {:.2f}".format(l.score(x_test, y_test)))
print("Number of features used (ALL/PE): {}".format(np.sum(l.coef_ != 0)))
```

```
Training set score (AT/PE): 0.90
Test set score (AT/PE): 0.90
Number of features used (AT/PE): 1

Training set score (V/PE): 0.76
Test set score (V/PE): 0.75
Number of features used (V/PE): 1

Training set score (AT+V/PE): 0.92
Test set score (AT+V/PE): 0.92
Number of features used (AT+V/PE): 2

Training set score (ALL/PE): 0.93
Test set score (ALL/PE): 0.93
Number of features used (ALL/PE): 4
```

Lasso gives the same results therefore reafirming the last affirmation.

6 K-neighbours regression

[13]:
```
reg=KNeighborsRegressor(n_neighbors=1)

reg.fit(xt_train, yt_train)
print("Test set R^2 (AT/PE): {:.3f}".format(reg.score(xt_test, yt_test)))
reg.fit(xv_train, yv_train)
print("Test set R^2 (V/PE): {:.3f}".format(reg.score(xv_test, yv_test)))
reg.fit(xtv_train, ytv_train)
print("Test set R^2 (AT+V/PE): {:.3f}".format(reg.score(xtv_test, ytv_test)))
reg.fit(x_train, y_train)
print("Test set R^2 (ALL/PE): {:.3f}".format(reg.score(x_test, y_test)))
```

```
Test set R^2 (AT/PE): 0.873
Test set R^2 (V/PE): 0.847
Test set R^2 (AT+V/PE): 0.999
Test set R^2 (ALL/PE): 1.000
```

6.1 Results

For n_neighbors=1:

Test set R^2 (AT/PE): 0.873 Test set R^2 (V/PE): 0.847 Test set R^2 (AT+V/PE): 0.999 Test set R^2 (ALL/PE): 1.000

For n_neighbors=2:

Test set R^2 (AT/PE): 0.900 Test set R^2 (V/PE): 0.887 Test set R^2 (AT+V/PE): 0.999 Test set R^2 (ALL/PE): 0.999

For n_neighbors=3:

Test set R^2 (AT/PE): 0.912 Test set R^2 (V/PE): 0.897 Test set R^2 (AT+V/PE): 0.996 Test set R^2 (ALL/PE): 0.997

For n_neighbors=4:

Test set R^2 (AT/PE): 0.917 Test set R^2 (V/PE): 0.901 Test set R^2 (AT+V/PE): 0.991 Test set R^2 (ALL/PE): 0.993

For n_neighbors=5:

Test set R^2 (AT/PE): 0.920 Test set R^2 (V/PE): 0.904 Test set R^2 (AT+V/PE): 0.984 Test set R^2 (ALL/PE): 0.988

For n_neighbors=6:

Test set R^2 (AT/PE): 0.92 Test set R^2 (V/PE): 0.90 Test set R^2 (AT+V/PE): 0.98 Test set R^2 (ALL/PE): 0.98

For n_neighbors=7:

Test set R^2 (AT/PE): 0.92 Test set R^2 (V/PE): 0.91 Test set R^2 (AT+V/PE): 0.97 Test set R^2 (ALL/PE): 0.98

For n_neighbors=8:

Test set R^2 (AT/PE): 0.92 Test set R^2 (V/PE): 0.91 Test set R^2 (AT+V/PE): 0.97 Test set R^2 (ALL/PE): 0.98

For n_neighbors=9:

Test set R^2 (AT/PE): 0.92 Test set R^2 (V/PE): 0.91 Test set R^2 (AT+V/PE): 0.97 Test set R^2 (ALL/PE): 0.98

For n_neighbors=10:

Test set R^2 (AT/PE): 0.92 Test set R^2 (V/PE): 0.91 Test set R^2 (AT+V/PE): 0.96 Test set R^2 (ALL/PE): 0.97

We now see that a perfect prediction is obtained when using 1 neighbour for ALL, although AT+V performs almost perfectly as well. However, the R^2 for these feature combinations (AT+V and ALL) is reduced when the number of neighbours used to make the regression is increased. On the otherside, when the features AT and V are used individually to predict PE, the higher the number of neighours used, the higher the R^2 score (and thus the predicting accuracy).

7 Decision Tree

In order to make a decision tree regression work, we first need to group the values for PE in different bins, as it is followed:

[14]:
```
bins=KBinsDiscretizer(n_bins=2,encode='ordinal',strategy='uniform') #uniform:␣
 ↪bins have the same width

yt_train=yt_train.reshape(-1, 1)
yt_test=yt_test.reshape(-1, 1)
yv_train=yv_train.reshape(-1, 1)
yv_test=yv_test.reshape(-1, 1)
ytv_train=ytv_train.reshape(-1, 1)
ytv_test=ytv_test.reshape(-1, 1)
y_train=y_train.reshape(-1, 1)
y_test=y_test.reshape(-1, 1)

yt_train = bins.fit_transform(yt_train)
yt_test = bins.fit_transform(yt_test)
yv_train = bins.fit_transform(yv_train)
yv_test = bins.fit_transform(yv_test)
ytv_train = bins.fit_transform(ytv_train)
ytv_test = bins.fit_transform(ytv_test)
y_train = bins.fit_transform(y_train)
y_test = bins.fit_transform(y_test)

yt_train=yt_train.reshape(1, -1)[0]
yt_test=yt_test.reshape(1, -1)[0]
yv_train=yv_train.reshape(1, -1)[0]
yv_test=yv_test.reshape(1, -1)[0]
ytv_train=ytv_train.reshape(1, -1)[0]
ytv_test=ytv_test.reshape(1, -1)[0]
y_train=y_train.reshape(1, -1)[0]
y_test=y_test.reshape(1, -1)[0]
```

Now the decision tree regression will be applied to the different feature combination and the target array that now instead of being a continuous value is a discrete (binned) one.

[15]:
```
tree = DecisionTreeClassifier(random_state=50,max_depth=1000)

tree.fit(xt_train, yt_train)
print("Accuracy on training set (AT/PE): {:.3f}".format(tree.score(xt_train,␣
 ↪yt_train)))
print("Accuracy on test set (AT/PE): {:.3f}".format(tree.score(xt_test,␣
 ↪yt_test)))
tree.fit(xv_train, yv_train)
print("Accuracy on training set (V/PE): {:.3f}".format(tree.score(xv_train,␣
 ↪yv_train)))
```

```
print("Accuracy on test set (V/PE): {:.3f}".format(tree.score(xv_test,␣
 ↪yv_test)))
tree.fit(xtv_train, ytv_train)
print("Accuracy on training set (AT+V/PE): {:.3f}".format(tree.score(xtv_train,␣
 ↪ytv_train)))
print("Accuracy on test set (AT+V/PE): {:.3f}".format(tree.score(xtv_test,␣
 ↪ytv_test)))
tree.fit(x_train, y_train)
print("Accuracy on training set (ALL/PE): {:.3f}".format(tree.score(x_train,␣
 ↪y_train)))
print("Accuracy on test set (ALL/PE): {:.3f}".format(tree.score(x_test,␣
 ↪y_test)))
```

```
Accuracy on training set (AT/PE): 0.957
Accuracy on test set (AT/PE): 0.954
Accuracy on training set (V/PE): 0.960
Accuracy on test set (V/PE): 0.958
Accuracy on training set (AT+V/PE): 0.999
Accuracy on test set (AT+V/PE): 0.992
Accuracy on training set (ALL/PE): 1.000
Accuracy on test set (ALL/PE): 0.991
```

7.1 Results

For n_bins=2:

Accuracy on training set (AT/PE): 0.957 Accuracy on test set (AT/PE): 0.954 Accuracy on training set (V/PE): 0.960 Accuracy on test set (V/PE): 0.958 Accuracy on training set (AT+V/PE): 0.999 Accuracy on test set (AT+V/PE): 0.992 Accuracy on training set (ALL/PE): 1.000 Accuracy on test set (ALL/PE): 0.991

For n_bins=3:

Accuracy on training set (AT/PE): 0.888 Accuracy on test set (AT/PE): 0.869 Accuracy on training set (V/PE): 0.854 Accuracy on test set (V/PE): 0.851 Accuracy on training set (AT+V/PE): 0.999 Accuracy on test set (AT+V/PE): 0.971 Accuracy on training set (ALL/PE): 1.000 Accuracy on test set (ALL/PE): 0.975

Using a simple decision tree the results obtained are very satisfactory for all the different feature combinations, having the best results when only 2 bins are used in the decision tree (all the value of PE are diveded into 2 bins). This method gives by far the best results when AT and V are used individually to predict PE. However, the k-nearest neighbours algorithm performs yet better so far for both AT+V and ALL from 1 to 3 neighbours. Another interesting observation is that this is the first algorithm getting a better accuracy in the test set for AT+V than for ALL.

Since the best result obtained for the binned discretization of the PE array was for n_bins=2, this value will further be used for a random forest algorithm (since random forests predictive results are based in those of individual decision trees).

8 Random Forest

[16]:
```
forest = RandomForestClassifier(n_estimators=5, random_state=20)

forest.fit(xt_train, yt_train)
print("Accuracy on training set (AT/PE): {:.3f}".format(forest.score(xt_train,␣
 ↪yt_train)))
print("Accuracy on test set (AT/PE): {:.3f}".format(forest.score(xt_test,␣
 ↪yt_test)))
forest.fit(xv_train, yv_train)
print("Accuracy on training set (V/PE): {:.3f}".format(forest.score(xv_train,␣
 ↪yv_train)))
print("Accuracy on test set (V/PE): {:.3f}".format(forest.score(xv_test,␣
 ↪yv_test)))
forest.fit(xtv_train, ytv_train)
print("Accuracy on training set (AT+V/PE): {:.3f}".format(forest.
 ↪score(xtv_train, ytv_train)))
print("Accuracy on test set (AT+V/PE): {:.3f}".format(forest.score(xtv_test,␣
 ↪ytv_test)))
forest.fit(x_train, y_train)
print("Accuracy on training set (ALL/PE): {:.3f}".format(forest.score(x_train,␣
 ↪y_train)))
print("Accuracy on test set (ALL/PE): {:.3f}".format(forest.score(x_test,␣
 ↪y_test)))
```

```
Accuracy on training set (AT/PE): 0.956
Accuracy on test set (AT/PE): 0.953
Accuracy on training set (V/PE): 0.960
Accuracy on test set (V/PE): 0.958
Accuracy on training set (AT+V/PE): 0.999
Accuracy on test set (AT+V/PE): 0.992
Accuracy on training set (ALL/PE): 1.000
Accuracy on test set (ALL/PE): 0.991
```

8.1 Results

For n_estimators=5:

Accuracy on training set (AT/PE): 0.956 Accuracy on test set (AT/PE): 0.953 Accuracy on training set (V/PE): 0.960 Accuracy on test set (V/PE): 0.958 Accuracy on training set (AT+V/PE): 0.999 Accuracy on test set (AT+V/PE): 0.992 Accuracy on training set (ALL/PE): 1.000 Accuracy on test set (ALL/PE): 0.991

For n_estimators=10:

Accuracy on training set (AT/PE): 0.957 Accuracy on test set (AT/PE): 0.954 Accuracy on training set (V/PE): 0.960 Accuracy on test set (V/PE): 0.958 Accuracy on training set (AT+V/PE): 0.999 Accuracy on test set (AT+V/PE): 0.992 Accuracy on training set (ALL/PE): 1.000 Accuracy on test set (ALL/PE): 0.991

For n_estimators=100:

Accuracy on training set (AT/PE): 0.957 Accuracy on test set (AT/PE): 0.954 Accuracy on training set (V/PE): 0.960 Accuracy on test set (V/PE): 0.958 Accuracy on training set (AT+V/PE): 0.999 Accuracy on test set (AT+V/PE): 0.992 Accuracy on training set (ALL/PE): 1.000 Accuracy on test set (ALL/PE): 0.991 **(exactly the same as for n_estimators=10)**

The results for the random forest predictions are exactly the same as those obtained when simple decision trees were used (for n_bins=2). The random forest ends up being a redundant extension of combining all the previous decision trees in its predictive process.

9 Gradient boosted regression trees

[17]:
```
gbrt = GradientBoostingClassifier(random_state=21)

gbrt.fit(xt_train, yt_train)
print("Accuracy on training set (AT/PE): {:.3f}".format(gbrt.score(xt_train,␣
 ↪yt_train)))
print("Accuracy on test set (AT/PE): {:.3f}".format(gbrt.score(xt_test,␣
 ↪yt_test)))
gbrt.fit(xv_train, yv_train)
print("Accuracy on training set (V/PE): {:.3f}".format(gbrt.score(xv_train,␣
 ↪yv_train)))
print("Accuracy on test set (V/PE): {:.3f}".format(gbrt.score(xv_test,␣
 ↪yv_test)))
gbrt.fit(xtv_train, ytv_train)
print("Accuracy on training set (AT+V/PE): {:.3f}".format(gbrt.score(xtv_train,␣
 ↪ytv_train)))
print("Accuracy on test set (AT+V/PE): {:.3f}".format(gbrt.score(xtv_test,␣
 ↪ytv_test)))
gbrt.fit(x_train, y_train)
print("Accuracy on training set (ALL/PE): {:.3f}".format(gbrt.score(x_train,␣
 ↪y_train)))
print("Accuracy on test set (ALL/PE): {:.3f}".format(gbrt.score(x_test,␣
 ↪y_test)))
```

```
Accuracy on training set (AT/PE): 0.947
Accuracy on test set (AT/PE): 0.951
Accuracy on training set (V/PE): 0.949
Accuracy on test set (V/PE): 0.949
Accuracy on training set (AT+V/PE): 0.969
Accuracy on test set (AT+V/PE): 0.967
Accuracy on training set (ALL/PE): 0.977
Accuracy on test set (ALL/PE): 0.973
```

Results obtained are slightly worse than those for the decision trees' and random forests' regressions.