C S 487/519 Applied Machine Learning I
Project-Stage

Group members:

1. Amer Al-Radaideh (800641066) (in-class session)
2. Khandker Mushfiqul Islam (800701988) (online-class session)

Problem Formulation:

In Aerospace research in general, scientists need to perform computer simulations before conducting any real experimental tests. However, it's difficult to obtain an exact dynamics model of the systems to represent the actual one.

In such research, researchers need to know the dynamics model between the PWM input commands to the drone's motors and the collective thrust force that is generated, see **Figure 1**. The dataset plotted and shown in Figure 2

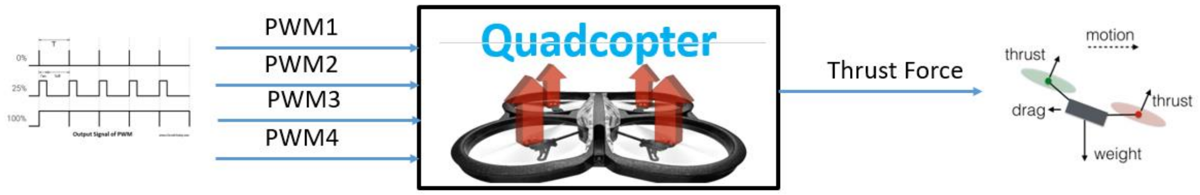

Figure 1Input-output diagram

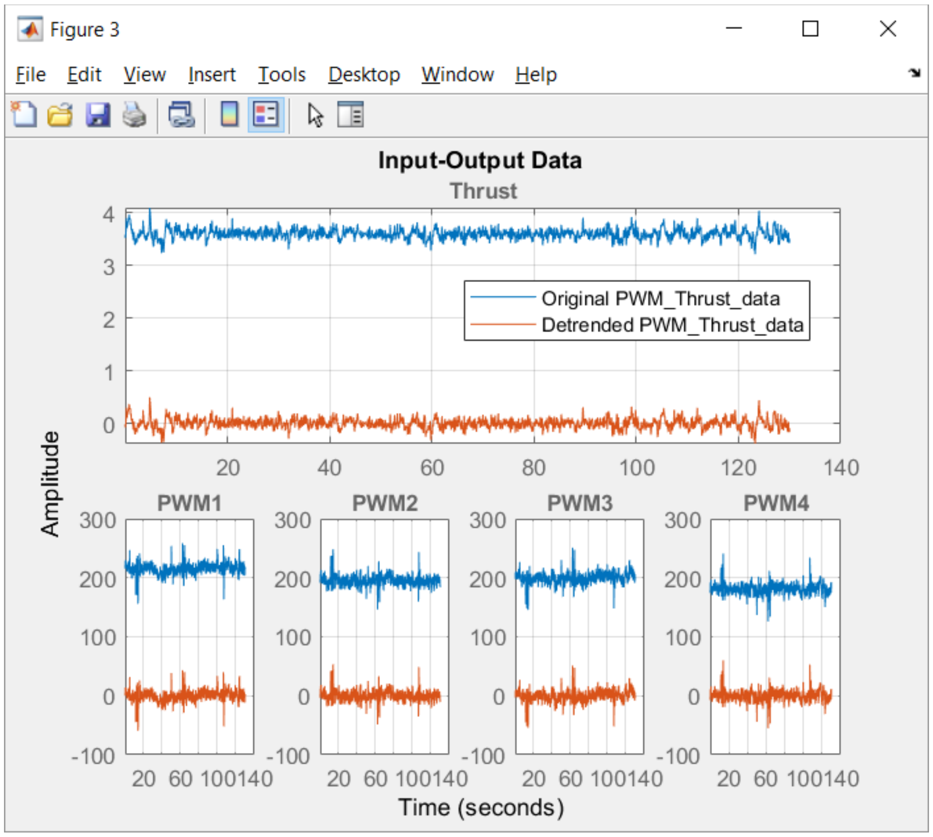

Figure 2 Input-Output Data plots

Motivations:

The motivation behind this project is to be able to get the most possible accurate model that represents the thrust force due to the input PWM commands.

Thrust force estimation for rotary-wing vertical take-off and landing (VTOL) UAVs is challenging due to the difficulty to mount the load cell sensors and get the readings while its flying in the air.

Unlike traditional aerodynamic modeling solutions, in this project, we are looking forward to utilize one of the machine learning-based method which does not require the details of the aerodynamic information to model the Thrust-PWM relation.

The proposed method includes two stages: an off-line training stage and an on-line thrust estimation stage. Only flight data is used for the on-line estimation stage. We use Parrot AR.Drone as the testing quadrotor.

Preliminary studies:

We have used the system Identification toolbox under Matlab environment to see how good are the results can be. The data were split into 25% for estimation and the remaining 75% for validation (see Figure 3).

Several models (available in the Matlab system Identification toolbox, see Figure 4) were selected to try them. As shown in Table 1, the state space gave the best results.

We are looking forward to try one of the Machine-Learning based techniques to find the best fit model and compare them with the models estimated using Matlab.

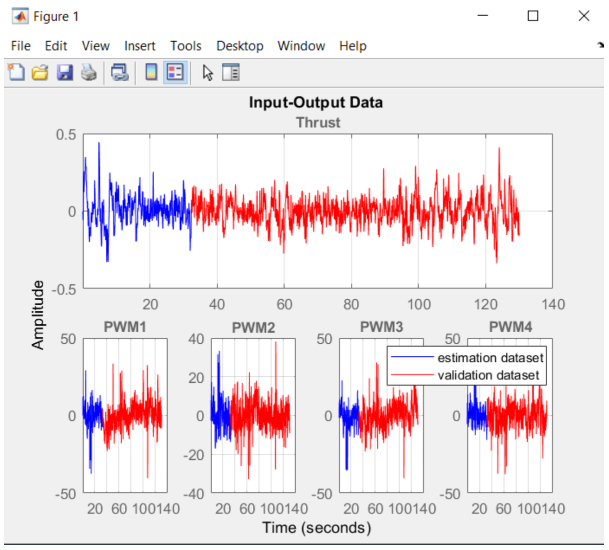

Figure 3 Data splitting for Estimation and Validation

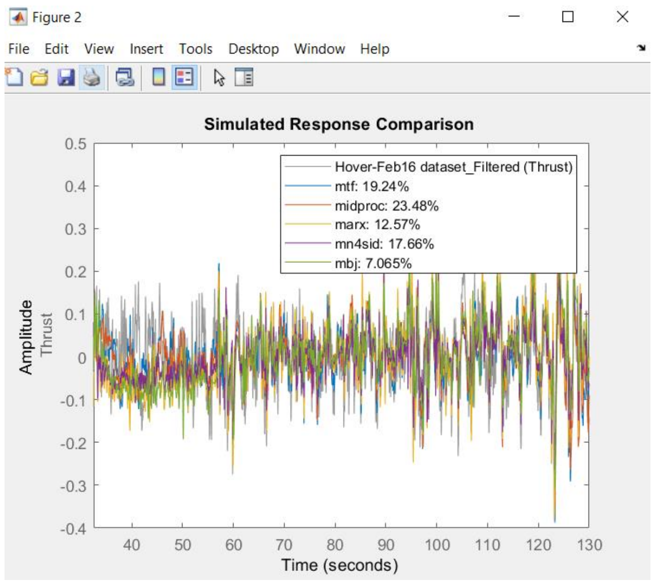

Figure 4 Several Data models and their best fit percentage

Table 1 Models fitting with percentage fit

Model name	Best Fit %
Transfer Function (mtf)	55.79%
Process Model (midproc0)	51.74%
Black-Box model-ARX Model (marx)	97.14%
State-Space Models Using (mn4sid)	99.45%
Box-Jenkins Model (bj)	95%

Proposed Solution:

The PWM-Thrust modeling problem is a highly nonlinear and has a mutually coupled terms. Since most of the methodologies that we have learned so far in the class are concerning the linear processes/functions we had to research the available tools to handle fitting multi-variable regression models to a data set with a highly nonlinear and mutually coupled terms as well as handling the noise presence issue. The noise (see Figure 5) makes it hard for the model to be bias-free and it also pushes the model towards overfitting because the model tries to make sense of the noisy data patterns and instead of discovering the real pattern, it fits itself to the noise.

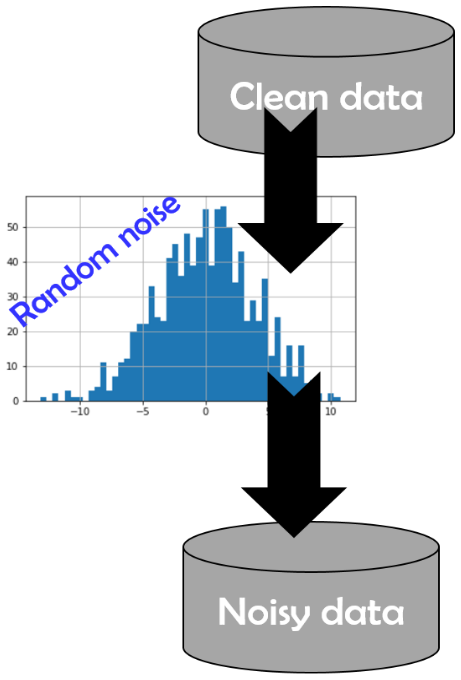

Figure 5 Dataset with noise

After a research on the Machine Learning methodologies that can be utilized to solve this problem, we came across few possible strategies for the nonlinear regressions:

1. Kernel Ridge Regression (KRR)
2. Support Vector Regression (SVR)
3. Random Forest Regression

For the Kernel ridge regression (KRR) (Murphy, 2012) combines Ridge regression and classification (linear least squares with l2-norm regularization) with the kernel trick. It thus learns a linear function in the space induced by the respective kernel and the data. For non-linear kernels, this corresponds to a non-linear function in the original space.

The form of the model learned by Kernel Ridge is identical to support vector regression (SVR). However, different loss functions are used: KRR uses squared error loss while support vector regression uses -insensitive loss, both combined with l2 regularization. In contrast to SVR, fitting Kernel Ridge can be done in closed-form and is typically faster for medium-sized datasets. On the other hand, the learned model is non-sparse and thus slower than SVR, which learns a sparse model for $\epsilon > 0$, at prediction-time.

For the last method to be used, a random forest (Raschka, 2015), to deal with the nonlinear relationships, it is an ensemble of multiple decision trees. It can be understood as the sum of piecewise linear functions in contrast to the global linear and polynomial regression models. In other words, via the decision tree algorithm, we are subdividing the input space into smaller regions that become more manageable.

With the research done, we found out that we can take advantage of Python machine learning library, scikit-learn to normalize the data, fit the model, keep the coefficients from becoming too large thereby maintaining bias-variance trade-off, and plot the regression score to judge the accuracy and robustness of each model.

Dataset:

The following figures show the dataset plots of both the input (Figure 8 PWM dataset from the 4 motorsFigure 8) and the output (Figure 9). The dataset is uploaded to the github folder.

```
def viz_plotting_dataset():
    plt.plot(df['PWM1'],'b',label="PWM1")
    plt.plot(df['PWM2'],'g',label="PWM2")
    plt.plot(df['PWM3'],'r',label="PWM3")
    plt.plot(df['PWM4'],'c',label="PWM4")

    plt.title('PWM inuput commands to the Motors of the ARDrone')
    plt.legend(loc="best")
    plt.xlabel('# of samples')
    plt.ylabel('PWM Percentage')
    plt.grid()
    plt.savefig('PWM_Inputs.png')
    plt.show()
    plt.plot(df['Thrust'],'m',label="Thrust")
    plt.title('Thrust produced by the drone motors')
    plt.xlabel('# of samples')
    plt.ylabel('Thrust Force [N]')
    plt.legend(loc='best')
    plt.grid()
    plt.savefig('Thrust.png')
    plt.show()
    return
```

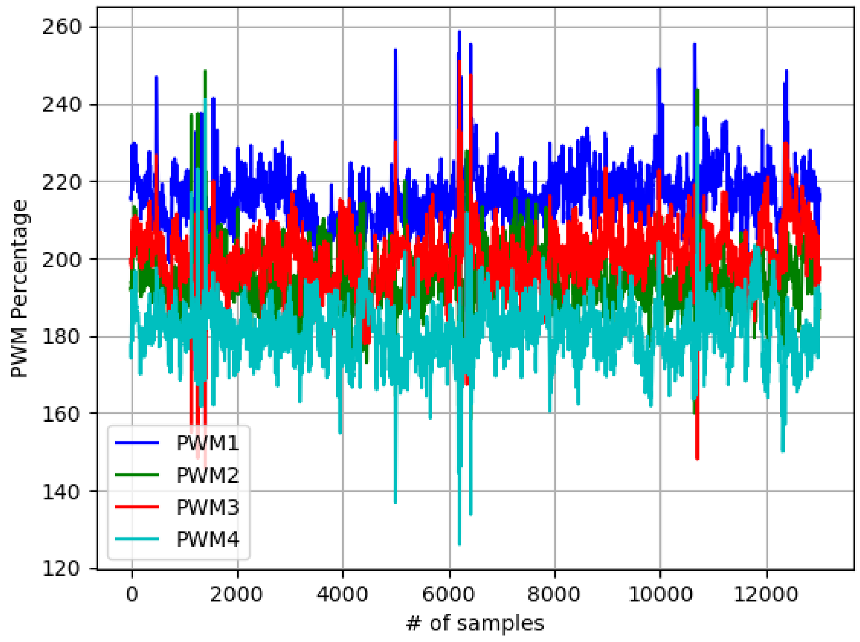

Figure 6 PWM dataset from the 4 motors

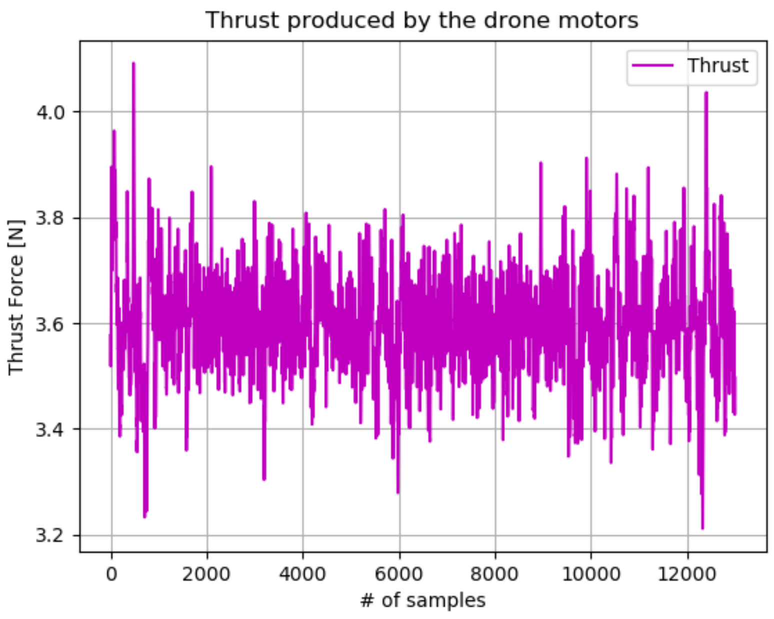

Figure 7 Thrust dataset estimated from the accelerometer

The following figures were generated using Matlab, just to make sure that we didn't mix any data.

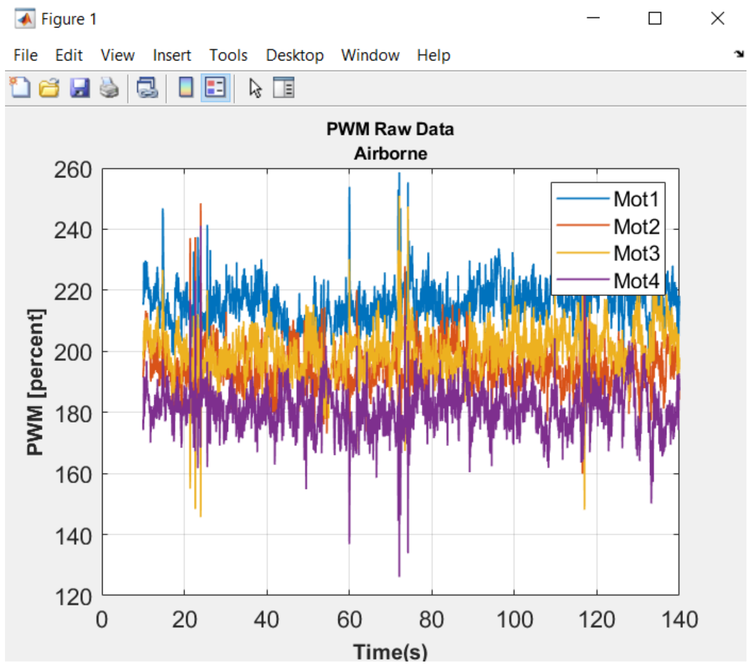

Figure 8 PWM dataset from the 4 motors

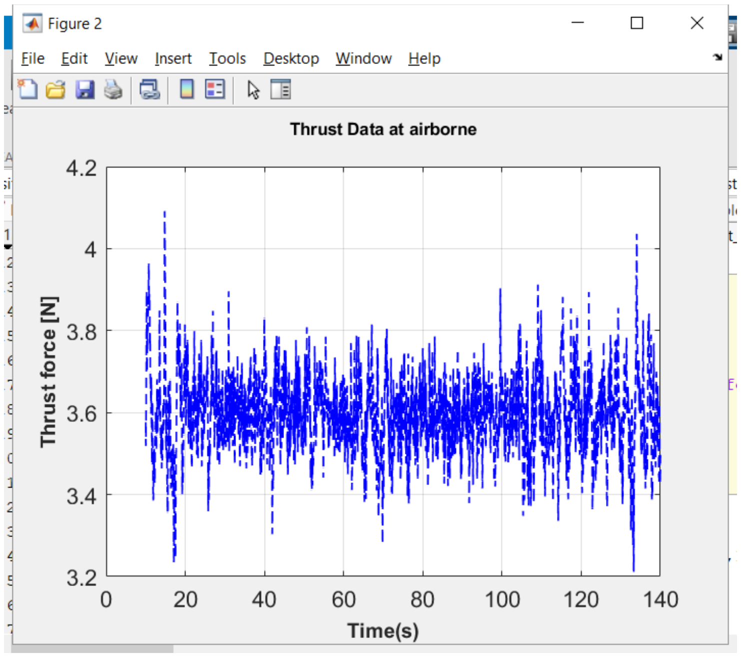

Figure 9 Thrust dataset estimated from the accelerometer

First, we will create a scatterplot matrix that allows us to visualize the pair-wise correlations between the different features in this dataset in one place. To plot the scatterplot matrix, we will use the pairplot function from the Seaborn library (http://stanford.edu/~mwaskom/software/seaborn/), which is a Python library for drawing statistical plots based on Matplotlib.

```
def viz_correlation_heatmap():
    cols = ['PWM1', 'PWM2', 'PWM3', 'PWM4', 'Thrust']
    sns.pairplot(df[cols],kind="reg", height=2.5)
    plt.tight_layout()
    plt.savefig('pairwise_correlation.png')
    plt.show()
    cm = np.corrcoef(df[cols].values.T)
    sns.set(font_scale=1.5)
    hm = sns.heatmap(cm,cbar=True,annot=True,square=True,fmt='.2f',annot_kws={'size':
15},yticklabels=cols,xticklabels=cols)
    plt.savefig('correlation_heatmap.png')
    plt.show()
    return
```

As we can see in the following figure, the scatterplot matrix provides us with a useful graphical summary of the relationships in a dataset:

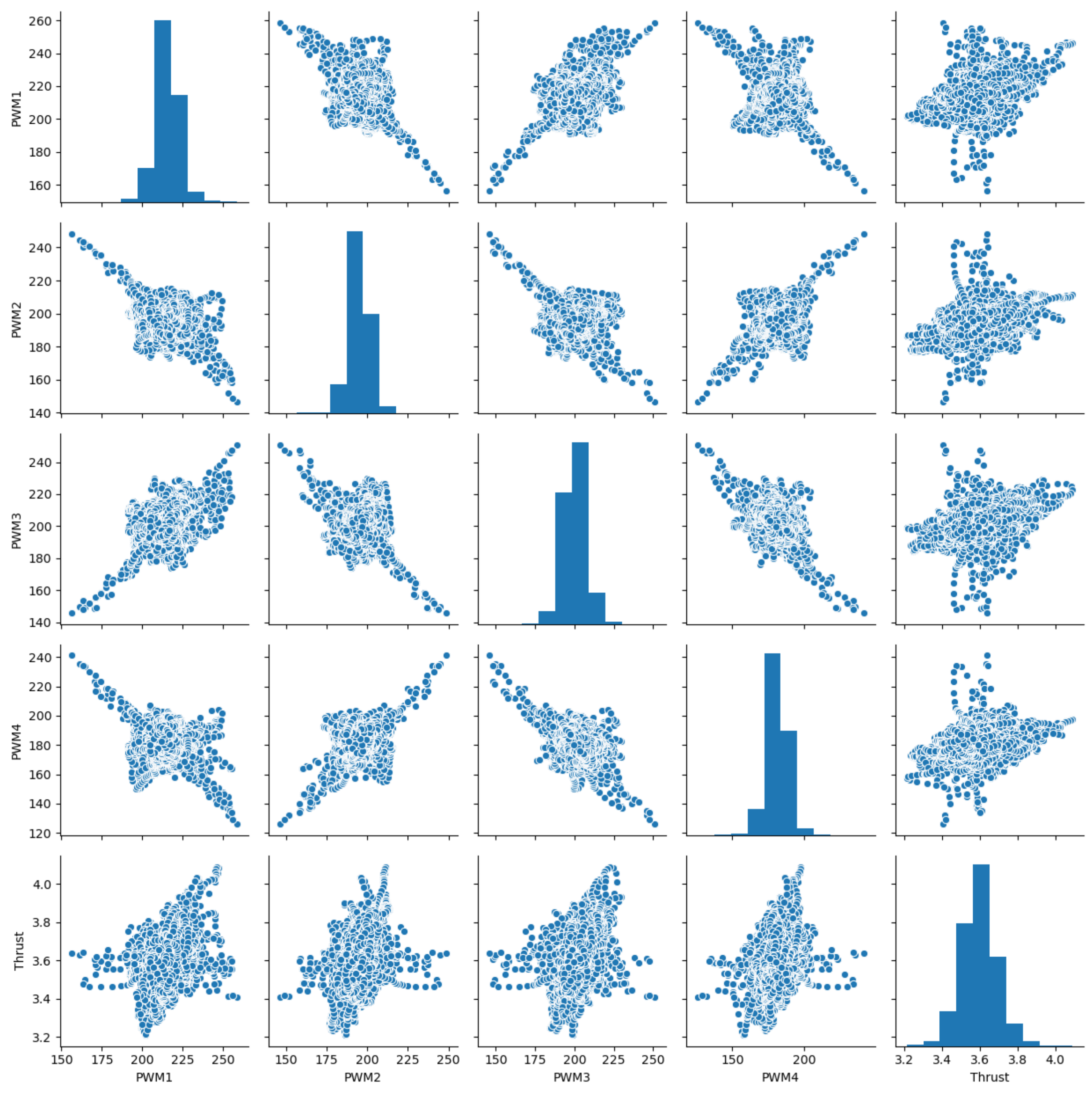

Figure 10 Pairwise correlation between the features of the dataset

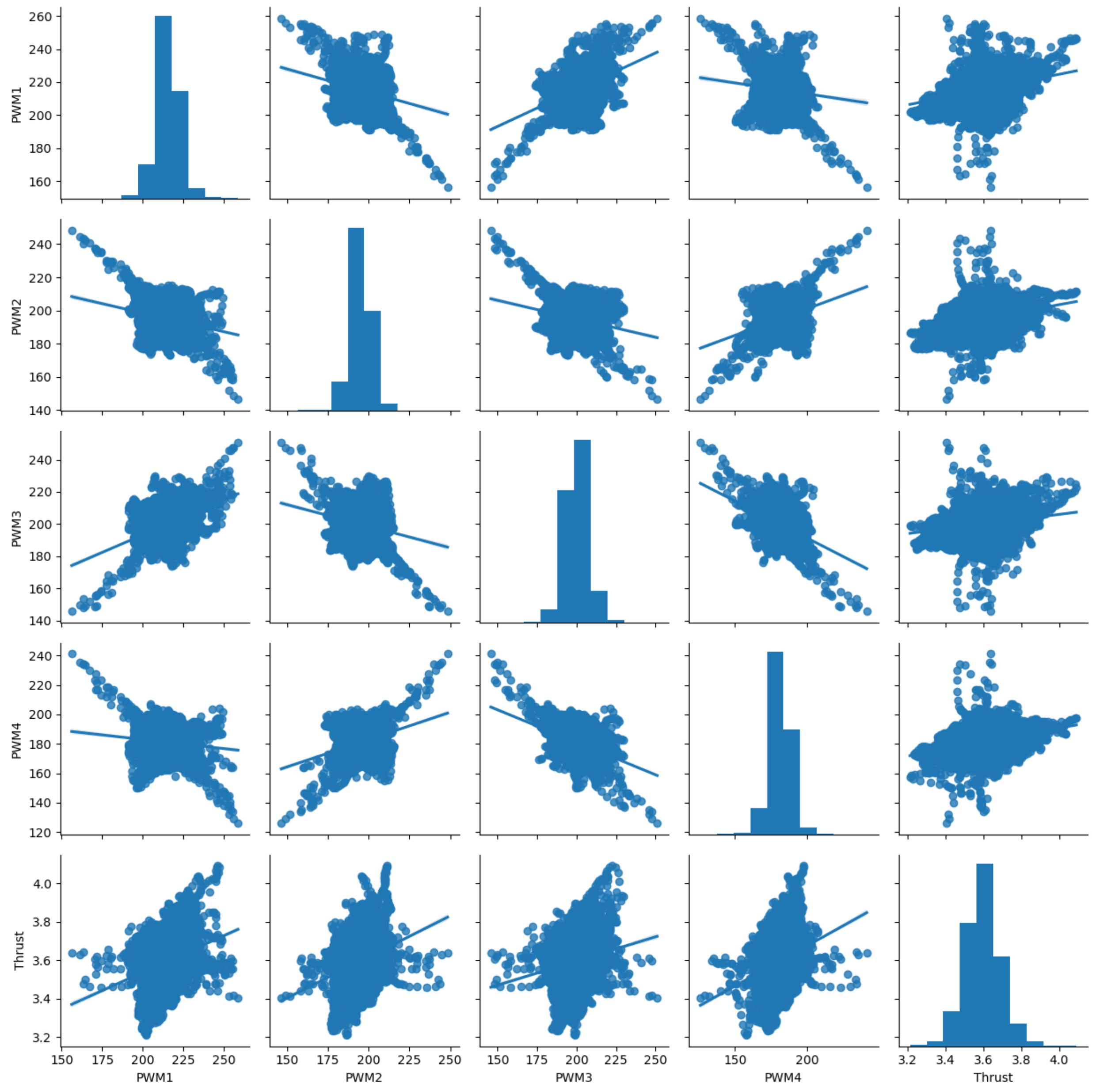

Figure 11 Pairwise correlation plot with linear regression fitting

Using this scatterplot matrix, we can now quickly eyeball how the data is distributed and whether it contains outliers. For example, we can see that there is a non-linear relationship between Thrust and all PWM signals (see the last row).
Furthermore, we can see in the histogram—the lower-right subplot in the scatter plot matrix—that the Thrust variable seems to be normally distributed but contains several outliers.

In the following figure, we have used the NumPy's corrcoef function on the five feature columns that we previously visualized in the scatterplot matrix, and we have used the Seaborn's heatmap function to plot the correlation matrix array as a heat map.

As we can see in the resulting figure, the correlation matrix provides us with another useful summary graphic that can help us to select features based on their respective correlations:

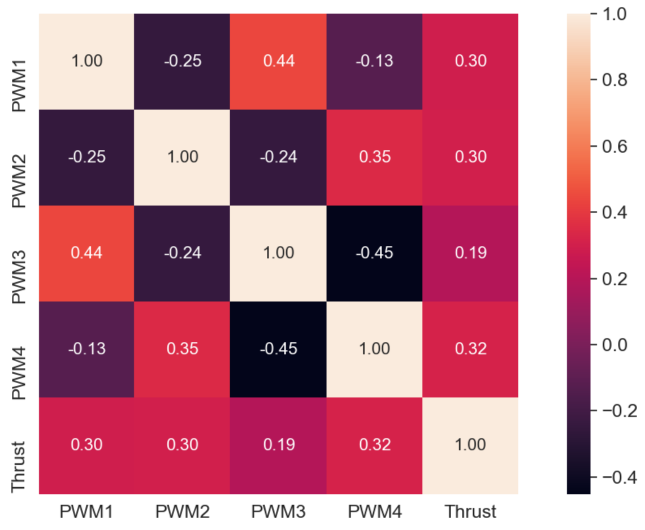

Figure 12 correlation matrix array as a heat map

Looking at the previous correlation matrix, we see that our target variable (Thrust) shows a low correlation with the PWM (input variables) and that's a clear nonlinear relationship between them. Based on the above we can introduce the concepts of non-linear regression models that we have proposed earlier.

Linear Regression first

Linear Regression is applied for the data set that their values are linear, although we know that our data is nonlinear, we want to start with the linear regression to compare it with the rest of regression models. The following code is used to do the linear regression fitting:

```
# Fitting Linear Regression to the dataset
#################################################################
def viz_linear_regressor():
    print('starting with linear Regressor:\n')
    lin_reg = LinearRegression()
    lin_reg.fit(X, y)
    y_train_pred = lin_reg.predict(X_train)
    y_test_pred = lin_reg.predict(X_test)
    print('MSE train: %.3f, test: %.3f\n' % (mean_squared_error(y_train,
y_train_pred),mean_squared_error(y_test, y_test_pred)))
    print('R^2 train: %.3f, test: %.3f\n' % (r2_score(y_train, y_train_pred), r2_score(y_test,
y_test_pred)))
    plt.plot(y_test, color='red')
    plt.plot(y_test_pred, color='blue')
    plt.title('PWM-Thrust (Linear Regression)')
    plt.xlabel('# of samples')
    plt.ylabel('Thrust Force[N]')
    plt.savefig('linear_regression.png')
    plt.show()
    print('Done with linear Regressor:\n')
    return
```

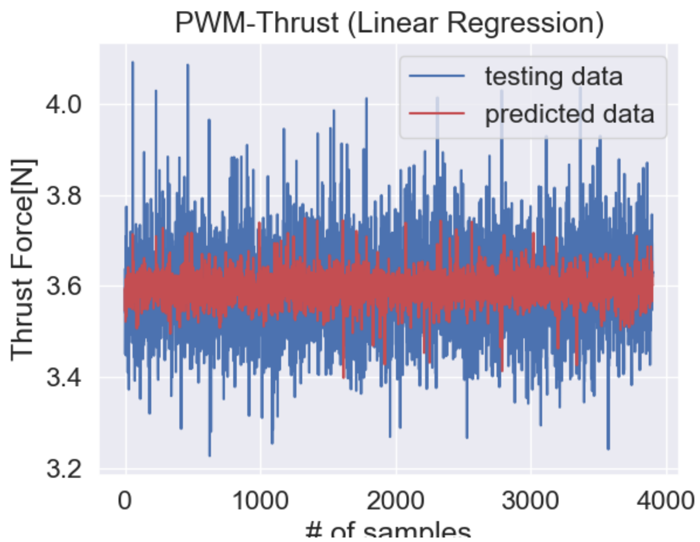

Figure 13 Linear Regression

MSE train: 0.008, test: 0.008

R^2 train: 0.090, test: 0.083

From the above results, it's clear that applying Linear Regression in this scenario is not giving us the right value. It's time to try the Polynomial Regression.

Polynomial Regression

```
##################################################################
# Fitting Polynomial Regression to the dataset
##################################################################
def viz_polynomial_regression():
    print('starting with Polynomial Regression (deg=4):\n')
    poly_reg = PolynomialFeatures(degree=4)
    X_poly = poly_reg.fit_transform(X)
    pol_reg = LinearRegression()
    pol_reg.fit(X_poly, y)
    y_train_pred=pol_reg.predict(poly_reg.fit_transform(X_train))
    y_test_pred=pol_reg.predict(poly_reg.fit_transform(X_test))
    print('MSE train: %.3f, test: %.3f\n' % (mean_squared_error(y_train,
y_train_pred),mean_squared_error(y_test, y_test_pred)))
    print('R^2 train: %.3f, test: %.3f\n' % (r2_score(y_train, y_train_pred), r2_score(y_test,
y_test_pred)))
    plt.plot(y_test, color='red')
    plt.plot(y_test_pred, color='blue')
    plt.title('PWM-Thrust (Linear Regression)')
    plt.xlabel('# of samples')
    plt.ylabel('Thrust Force[N]')
    plt.savefig('polynomial_regression_deg4.png')
    plt.show()
    print('Done with Polynomial Regression:\n')
    return
```

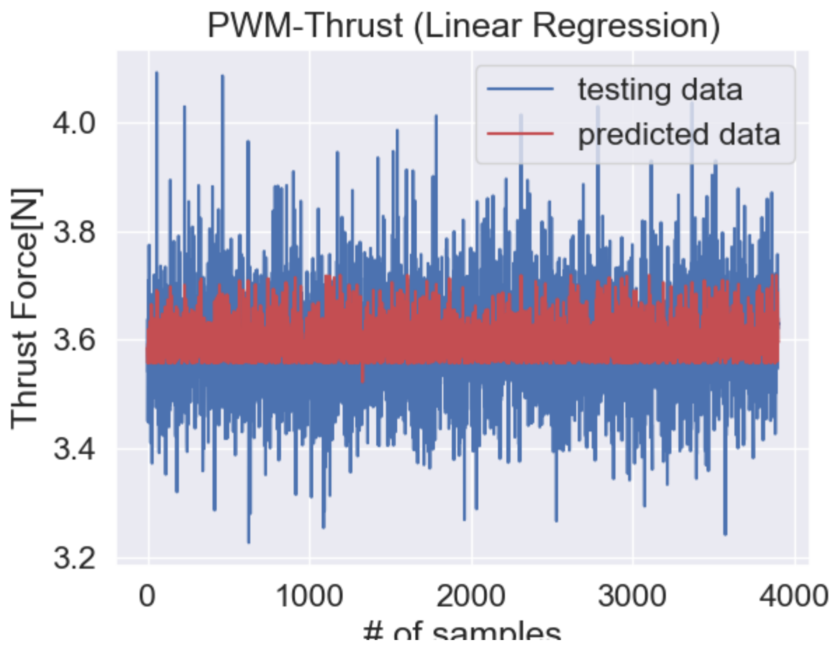

Figure 14 Polynomial Regression (4th degree)

MSE train: 0.008, test: 0.008

R^2 train: 0.117, test: 0.113

We have even tried the polynomial of 6th degree to see if it can help:

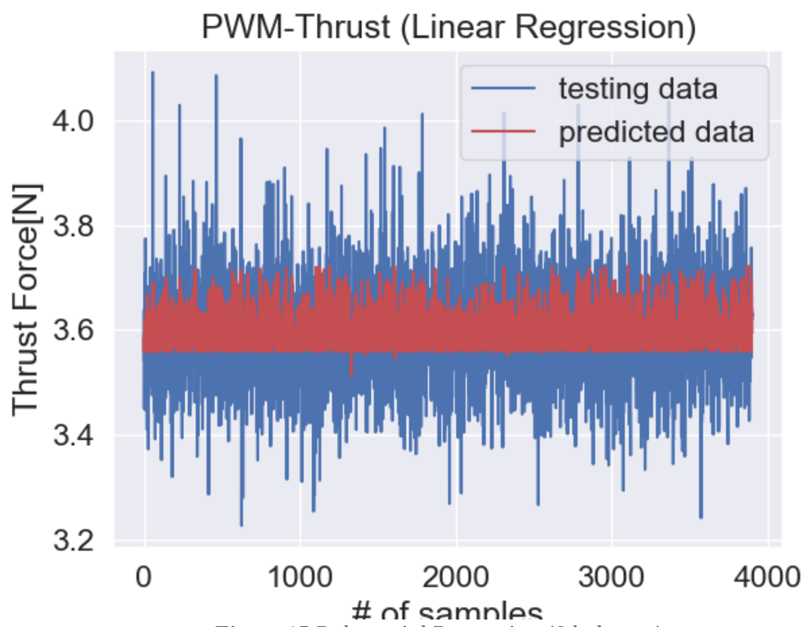

Figure 15 Polynomial Regression (6th degree)

MSE train: 0.008, test: 0.008

R^2 train: 0.118, test: 0.114

From the above results, the polynomial regression with high degree didn't help.

Kernel ridge regression:

Kernel ridge regression (KRR) combines Ridge regression and classification (linear least squares with l2-norm regularization) with the kernel trick. It thus learns a linear function in the space induced by the respective kernel and the data. For non-linear kernels, this corresponds to a non-linear function in the original space.

```
def viz_Kernal_Ridge():
    krr = KernelRidge(alpha=1.0)
    krr.fit(X_train, y_train)
    print ("Training accuracy for KRR")
    print (krr.score(X_train,y_train))
    y_train_pred= krr.predict(X_train)
    y_test_pred= krr.predict(X_test)
    r2_train = r2_score(y_train, y_train_pred)
    r2_test = r2_score(y_test, y_test_pred)
    error_train= mean_squared_error(y_train, y_train_pred)
    error_test= mean_squared_error(y_test, y_test_pred)
    print('MSE train: %.3f, test:%.3f' % (error_train, error_test))
    print('R^2 value train: %.3f, test:%.3f' % (r2_train, r2_test))
    plt.plot(y_test, color='red')
    plt.plot(y_test_pred, color='blue')
    plt.title('PWM-Thrust (Kernel Ridge)')
    plt.xlabel('# of samples')
    plt.ylabel('Thrust Force[N]')
    plt.savefig('Kernel_ridge.png')
    plt.show()
    print('Done with Kernel Ridge:\n')
    return
```

MSE train: 0.017, test:0.018
R^2 value train: -0.881, test:-1.010

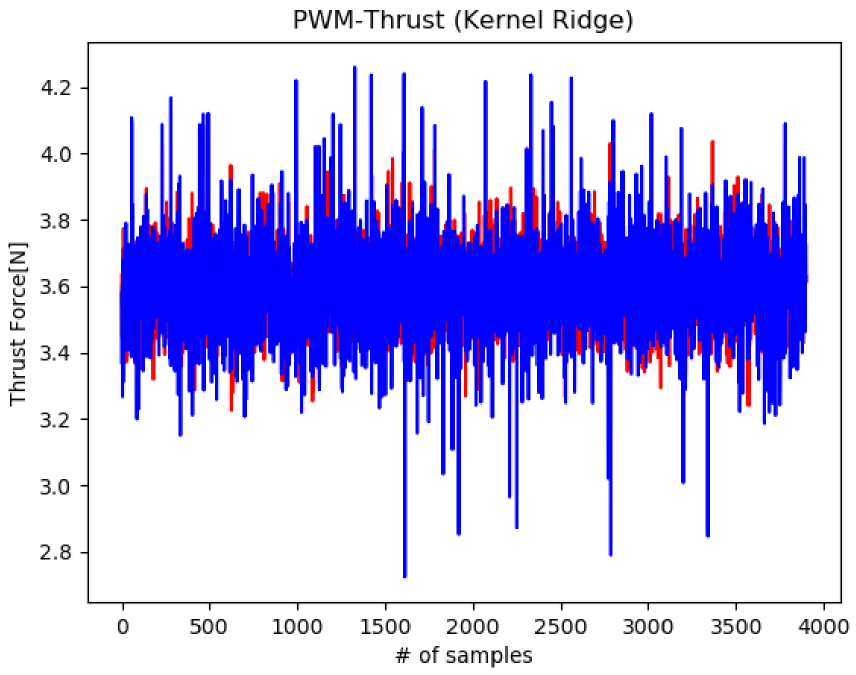

Figure 16 Ridge Kernel

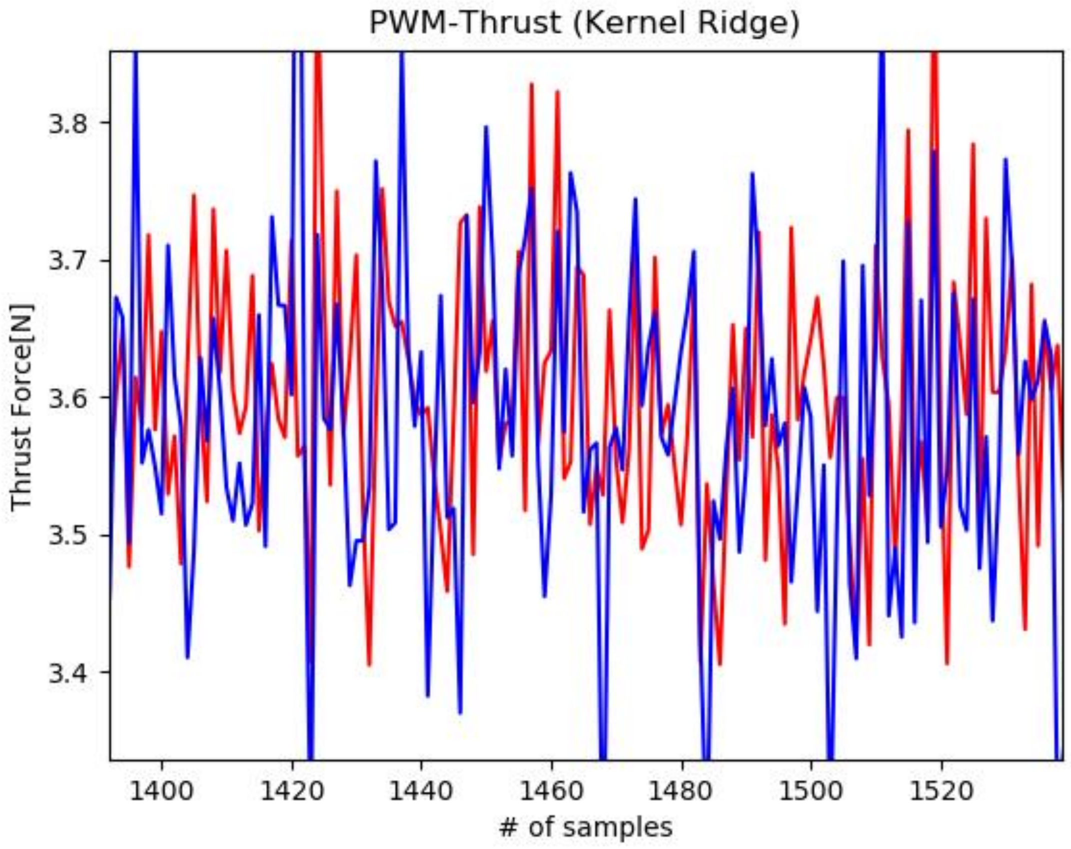

Figure 17 (Zoomed In) for Ridge Kernel

The ridge kernel from the above results didn't show a good results, so we had to move forward with the Random Forest.

Random Forest:

random forest, which is an ensemble of multiple decision trees, can be understood as the sum of piecewise linear functions in contrast to the global linear and polynomial regression models.
Also, random forest usually has a better generalization performance than an individual decision tree due to randomness, which helps to decrease the model's variance. Other advantages of random forests are that they are less sensitive to outliers in the dataset and don't require much parameter tuning. The only parameter in random forests that we typically need to experiment with is the number of trees in the ensemble.

Data pre-processing:

Before feeding the data to the random forest regression model, we need to do some pre-processing.
Here, we'll create the x and y variables by taking them from the dataset and using the train_test_split function of scikit-learn to split the data into training and test sets.
Note that the test size of 0.4 indicates we've used 40% of the data for testing. random_state ensures reproducibility. For the output of train_test_split, we get x_train, x_test, y_train, and y_test values.
The code is as follows:

```
X = df[['PWM1', 'PWM2', 'PWM3', 'PWM4']] .values
y = df['Thrust'].values
print('the x data are:\n',X)
print('The y data are:\n',y)

print('starting with Random Forest Regressor:\n')
X_train, X_test, y_train, y_test =train_test_split(X, y,test_size=0.4,random_state=1)
```

Train the model:

We're going to use x_train and y_train, obtained above, to train our random forest regression model. We're using the fit method and passing the parameters as shown in the code below.
Note that the output of this cell is describing a large number of parameters like criteria, max depth, etc. for the model. All these parameters are configurable, and we can tune them to match our requirements.

```
forest = RandomForestRegressor(n_estimators=1000,criterion='mse',random_state=10,n_jobs=-1)
forest.fit(X_train, y_train)
```

Prediction:
Once the model is trained, it's ready to make predictions. We can use the predict method on the model and pass x_test as a parameter to get the output as y_pred.
Notice that the prediction output is an array of real numbers corresponding to the input array.

```
y_train_pred = forest.predict(X_train)
y_test_pred = forest.predict(X_test)
```

Model Evaluation:

Another

Finally, we need to check to see how well our model is performing on the test data. For this, a useful quantitative measure of a model's performance is the so-called Mean Squared Error (MSE), which is simply the averaged value of the SSE cost that we minimized to fit the linear regression model. The MSE is useful to compare different regression models or for tuning their parameters via grid search and cross-validation, as it normalizes the SSE by the sample size:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 .$$

Sometimes it may be more useful to report the coefficient of determination (R2), which can be understood as a standardized version of the MSE, for better interpretability of the model's performance. Or in other words, R2 is the fraction of response variance that is captured by the model. The R2 value is defined as:

$$R^2 = 1 - \frac{SSE}{SST}$$

$$1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2}$$

$$1 - \frac{MSE}{Var(y)}$$

For the training dataset, the R_2 is bounded between 0 and 1, but it can become negative for the test set. If $R_2 = 1$, the model fits the data perfectly with a corresponding $MSE = 0$.

Now we will compute the MSE and R^2 of our training and test predictions:

```
print('MSE train: %.3f, test: %.3f\n' % (mean_squared_error(y_train,
y_train_pred),mean_squared_error(y_test, y_test_pred)))
print('R^2 train: %.3f, test: %.3f\n' % (r2_score(y_train, y_train_pred), r2_score(y_test, y_test_pred)))
```

<u>MSE train: 0.001, test: 0.004</u>

<u>R^2 train: 0.938, test: 0.543</u>

We see that the MSE on the training set is 0.001, and the MSE of the test set is also low, with a value of 0.004, which is an indicator that our data values are dispersed closely to its central moment (mean); which is usually great. It reflects on the distribution of our data values and it is centralized, and that it is not skewed, and most of all, that it has smaller errors (errors measured by the dispersion of the data values from its mean).

Evaluated on the training data, the R2 of our model is 0.938, which is a great fit. However, the R2 on the test dataset is only 0.543,

Now, let us also take a look at the residuals of the prediction where we simply subtract the true target variables from our predicted responses:

```
plt.scatter(y_test_pred,y_test_pred -
y_test,c='limegreen',edgecolor='white',marker='s',s=35,alpha=0.9,label='Test data')
plt.xlabel('Predicted values')
plt.ylabel('Residuals')
plt.legend(loc='upper left')
plt.hlines(y=0, xmin=3, xmax=5, lw=2, color='black')
plt.xlim([3.1, 4.2])
plt.grid()
plt.savefig('residuals_random_forest.png')
plt.show()
print('Done with Random Forest Regressor:\n')
```

After executing the code, we see a residual plot with a line passing through the *x*-axis origin as shown here:

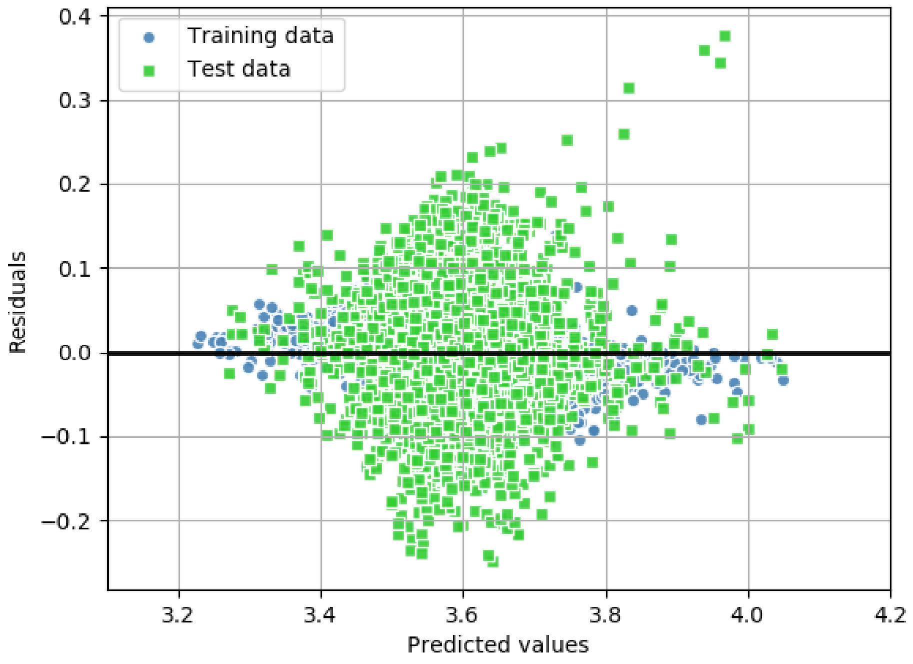

Figure 18 Residuals plot for the Random Forest

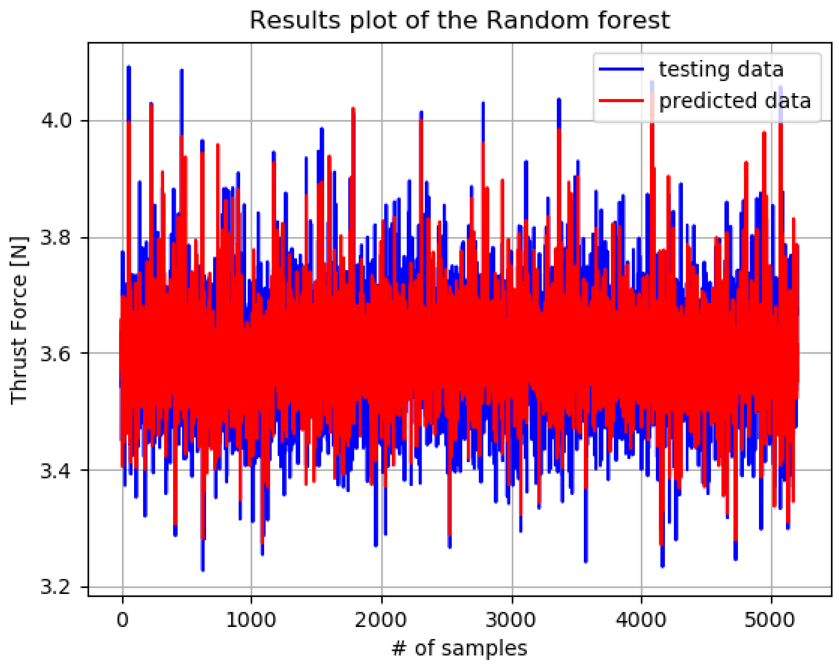

Figure 19 Predicted Vs Tested data

In case of a perfect prediction, the residuals would be exactly zero, which we will probably never encounter in realistic and practical applications. However, for a good regression model, we would expect that the errors are randomly distributed and the residuals should be randomly scattered around the centerline. If we see patterns in a residual plot, it means that our model is unable to capture some explanatory information, which has leaked into the residuals, as we can slightly see in our residual plot. Furthermore, we can also use residual plots to detect outliers, which are represented by the points with a large deviation from the centerline.

As it was already summarized by the R2 coefficient, we can see that the model fits the training data better than the test data, as indicated by the outliers in the y-axis direction. Also, the distribution of the residuals does not seem to be completely random around the zero center point, indicating that the model is not able to capture all the exploratory information.

If we observe patterns in the prediction errors, for example, by inspecting the residual plot, it means that the residual plots contain predictive information. Unfortunately, there is now a universal approach for dealing with non-randomness in residual plots, and it requires experimentation. Depending on the data that is available to us, we may be able to improve the model by transforming variables, tuning the hyper parameters of the learning algorithm, choosing simpler or more complex models, removing outliers, or including additional variables.

References

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective.* The MIT Press.
Raschka, S. (2015). *Python machine learning.* Packt Publishing Ltd.