Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

PREDICTIVE ANALYSIS OF DUST LEVELS IN VIT

IoT Domain Analyst
ECE3502
(B1 Slot)

Submitted by:

Indu Bhushan (20BEC0636)
Josh G. Jacob (20BEC0304)

Under the guidance of
Dr. Biswajit Dwivedy

TABLE OF CONTENT

Acknowledgements:

We would like to express our sincere gratitude to our esteemed faculty Dr. Biswajit Dwivedy for his invaluable guidance and support throughout our project on "Predictive Analysis of Dust Levels in VIT". His expertise in the field and willingness to share his knowledge have been instrumental in the success of our project. He taught us many of the concepts that were used in this project as well as offered timely suggestions when we required them. Thank you for your time, encouragement, and valuable insights that helped us navigate through the project with confidence. We would also like to thank VIT University for facilitating this project.

Abstract:

Dust levels can have a significant impact on human health, especially for those who work in environments with high levels of dust such as construction sites, mines, and factories. Predictive analysis of dust levels using machine learning algorithms such as Polynomial Regression or Random Forest Regression can be a powerful tool for real-time monitoring and prediction of dust levels. By collecting data using a dust sensor and storing it in Excel sheets, we can pre-process the data, split it into training and testing datasets, and use polynomial regression to build a model that can predict future dust levels based on the model developed. In this way, we can reduce the risk of respiratory problems and other health hazards caused by high levels of dust in the environment by notifying users about how high dust levels may be on a particular day. The same methodology may be employed over a larger period to enable other similar studies to be performed.

Introduction:

In today's world, where automobiles, factories, industries, and the like are prevalent, there has been an increase in the amount of dust in the air as compared to the past. This is of huge concern to many people, especially those with lung and breathing related ailments such as asthma, bronchitis etc., where exposure to dust can trigger symptoms of the same ranging from those mild in nature to others that are more serious. As such, it is very useful to be able to predict the concentration of dust of different types (such as PM1.0, PM2.5 and PM10) may be present on a given day, so as to be able to provide warning to such individuals if required. By examining the concentration of different kinds of particulate matter over an extended period, machine learning algorithms may be employed in order to perform predictive analytics such as Polynomial Regression and Random Forest Regression.

Theory:

K-means clustering is a machine-learning approach that is frequently used to cluster data points according to how similar they are. K-means clustering may be used to find patterns in dust concentration data in the context of dust monitoring, which can help identify high-risk regions and create efficient monitoring and mitigation plans.

Data points are iteratively divided into k groups according to how similar they are in the K-means clustering technique. The k centroids, which stand for the centers of each cluster, are first chosen at random by the algorithm. After that, each data point is allocated to the closest centroid, which is then updated using the mean of all the data points assigned to it.

The data points are clustered into k groups based on their closeness to the centroids and this procedure is repeated until convergence is attained.

K-means clustering can be utilized in the context of dust monitoring to spot trends in the data on dust concentration over time or between various sites. K-means clustering can help in the identification of high-risk regions and the creation of efficient monitoring and mitigation techniques by grouping data points into groups based on their similarity.

In order to pinpoint regions with persistently high dust concentrations for additional monitoring or mitigation measures, K-means clustering might be utilized, for instance.

As an alternative, K-means clustering might be used to find temporal trends in data on dust concentration, which could assist forecast future dust levels and allow for the adoption of preventative measures to lessen the risk of respiratory issues and other health risks brought on by high levels of dust.

Generally, K-means clustering is a potent machine learning algorithm that can help with the identification of patterns in data on dust concentration, which can help to improve dust monitoring and mitigation practices and lessen the risk of respiratory issues and other health risks brought on by high levels of dust in the environment.

Polynomial Regression:

Polynomial Regression is a regression algorithm that models the relationship between a dependent variable (y) and independent variable (x) as an nth degree polynomial. The Polynomial Regression equation is given below:

$$y= b_0+b_1x_1+ b_2x_1^2+ b_2x_1^3+. \quad b_n x_1^n$$

It is also called the special case of Multiple Linear Regression in ML because we add some polynomial terms to the Multiple Linear regression equation to convert it into Polynomial Regression. It is a linear model with some modification in order to increase the accuracy. The dataset used in Polynomial regression for training is of non-linear nature. It makes use of a linear regression model to fit the complicated and non-linear functions and datasets. In polynomial regression, the relationship between the dependent variable and the independent variable is modelled as an nth-degree polynomial function. When the polynomial is of degree 2, it is called a quadratic model; when the degree of a polynomial is 3, it is called a cubic model, and so on. The degree of order which to use is a Hyperparameter, and we need to choose it wisely. But using a high degree of polynomial tries to overfit the data, and for smaller values of degree, the model tries to underfit, so we need to find the optimum value of a degree. Polynomial Regression models are usually fitted with the method of least squares. The least square method minimizes the variance of the coefficients under the Gauss-Markov Theorem.

Random Forest Regression:

Random Forest Regression is a supervised machine learning algorithm that uses ensemble learning (combines predictions from multiple ML algorithms) method for regression. In the Random Forest Regression model, k data points are chosen, and a decision tree is built associated with these data points. N such trees are made, and the dependent variable y can be predicted by providing the data-point to eachtree. The final output y is the average of the results from all trees.

Particulate Matter:

PM2.5 and PM10 refer to particulate matter with particle diameter up to 2.5 microns and 10 microns respectively and are among the most dangerous air pollutants. Due to their small size, PM2.5 particles can travel deep into the human lung and cause a variety of health issues; for instance, by triggering asthma attacks or contributing to cardiovascular disease.

High concentrations of dust or PM is a serious health concern. PM2.5 is less than 2.5 microns in diameter, and PM10 is less than 10 microns in diameter. This means a PM10 report includes PM2.5 as well. Both these particles are much smaller than a human hair, which is about 70 microns in width.

PM10: Operations such as stone crushing, coal grinding, rotary kilning in the cement industry, and dust on road stirred by moving vehicles can increase PM10 levels. PM10 limit for 24-hour average is 150µg/m3.

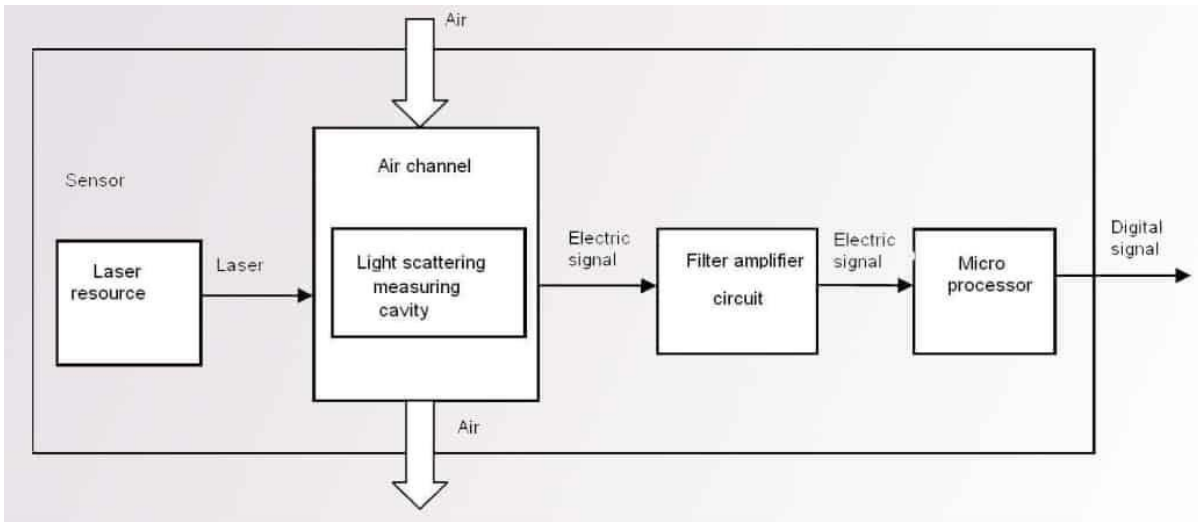
PM2.5: This is a result of fine particles produced from all types of combustion, including motor vehicles, thermal power plants, residential wood burning, forest fires, agricultural burning, and other industrial processes. PM2.5 limit for 24-hour average is 35µg/m3.

So, for measuring the Particulate Matter size of PM1.0, PM2.5 & PM10 we are using Plant power PMS7003 Dust Sensor.

PMS7003 Dust Sensor Working:

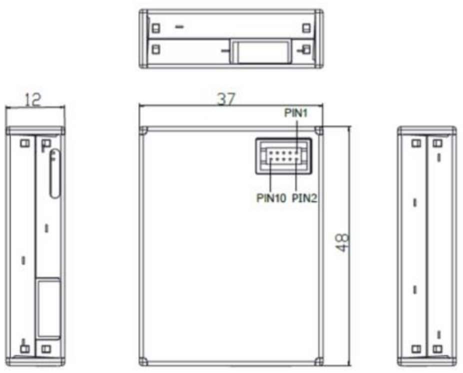
PMS7003 is a kind of digital and universal particle concentration sensor, which can be used to obtain the number of suspended particles in the air, i.e., the concentration of particles, and output them in the form of a digital interface. This sensor can be inserted into variable instruments related to the concentration of suspended particles in the air or other environmental improvement equipment to provide correct concentration data in time.

Laser scattering principle is used for such sensor, i.e., produce scattering by using a laser to radiate suspending particles in the air, then collect scattering light in a certain degree, and finally obtain the curve of scattering light change with time. In the end, equivalent particle diameter and the number of particles with different diameters per unit volume can be calculated by microprocessor-based on MIE theory.



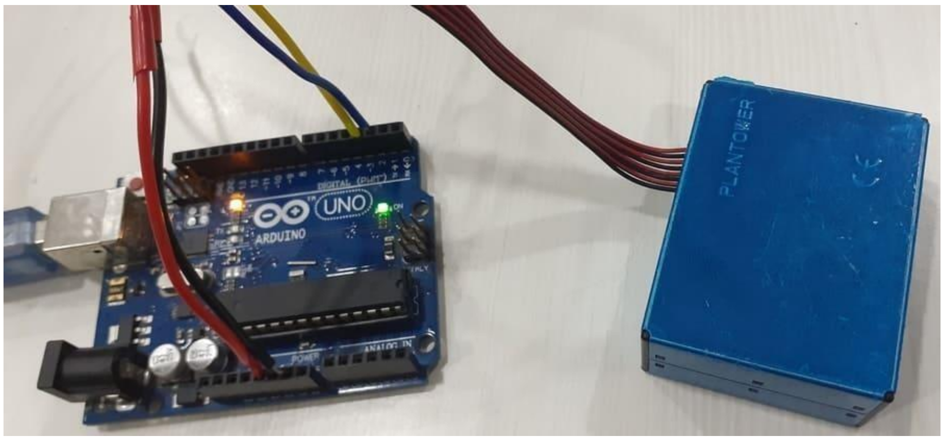
Minimal resolution particle size 0. 3um.Zero error alarm rate. Real-time response and support for the continuous acquisition. Inlet direction is optional, the use of a wide range of users without the need for duct design. Ultra-thin design, only 12mm, for portable devices to TTL serial download cable, compatible with win XP / VISTA / 7/8 / 8.1 system, can emulate existing applications on the COM port based on most operating systems and can easily adapt to any connector and handheld device. Power supply voltage: 5V.

Pin Configuration:



引脚定义：

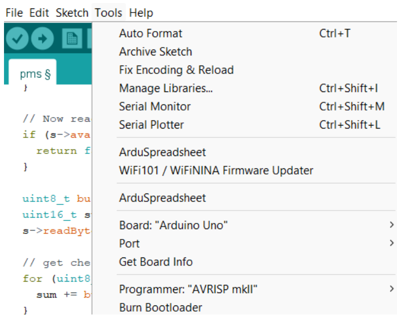
Pin1----------	VCC
Pin2----------	VCC
Pin3----------	GND
Pin4----------	GND
Pin5----------	Reset
Pin6----------	N/C
Pin7----------	RX
Pin8----------	N/C
Pin9----------	TX
Pin10---------	Set



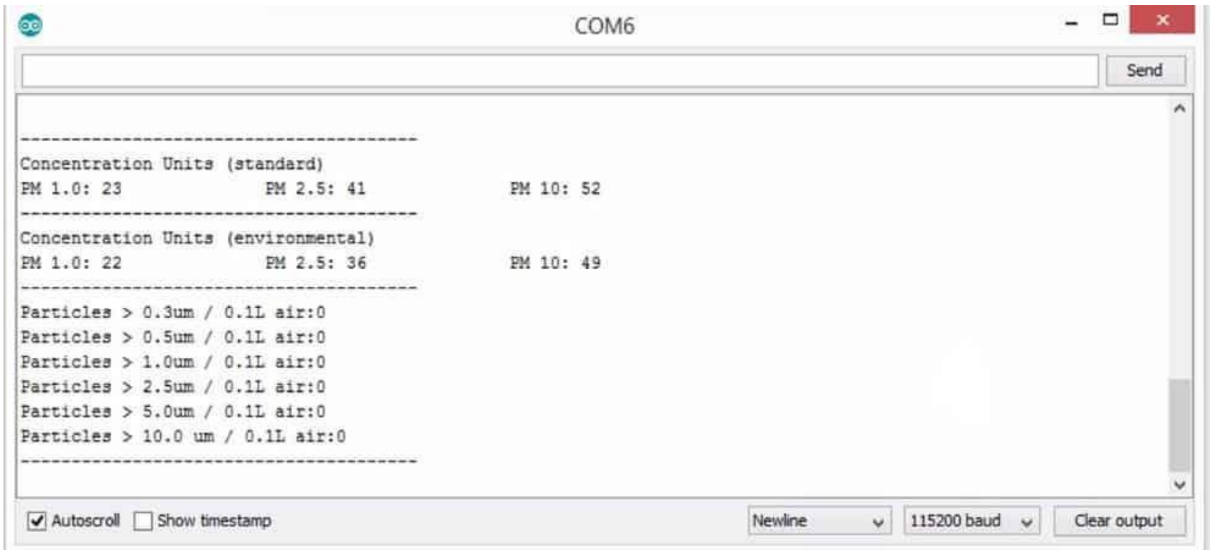
Methodology:

Converting from Serial Monitor to Excel (CSV file):

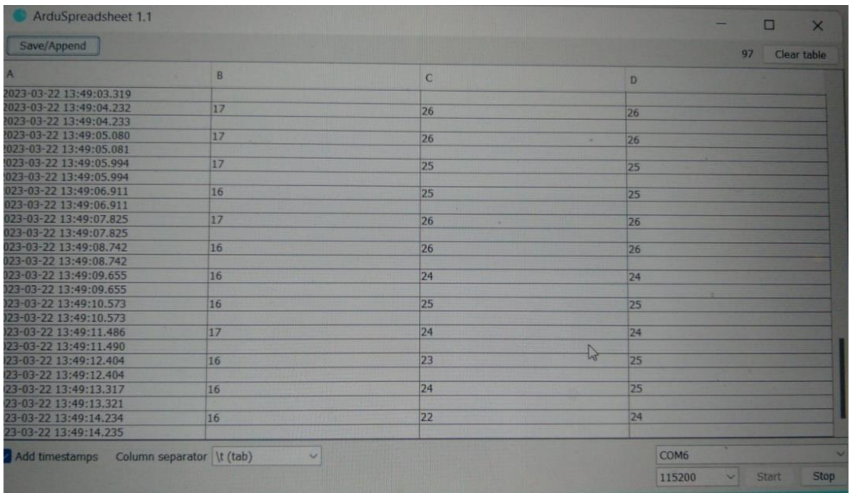
We have used “ArduSpreadsheet” to make the conversion.


File Edit Sketch Tools Help
pms §
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
ArduSpreadsheet
WiFi101 / WiFiNINA Firmware Updater
ArduSpreadsheet
Board: "Arduino Uno"
Port
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader

```
}

// Now rea
if (s->ava
  return f
}

uint8_t bu
uint16_t s
s->readByt

// get che
for (uint8
  sum += b
}
```



```
---------------------------------------
Concentration Units (standard)
PM 1.0: 23              PM 2.5: 41              PM 10: 52
---------------------------------------
Concentration Units (environmental)
PM 1.0: 22              PM 2.5: 36              PM 10: 49
---------------------------------------
Particles > 0.3um / 0.1L air:0
Particles > 0.5um / 0.1L air:0
Particles > 1.0um / 0.1L air:0
Particles > 2.5um / 0.1L air:0
Particles > 5.0um / 0.1L air:0
Particles > 10.0 um / 0.1L air:0
---------------------------------------
```



A	B	C	D
2023-03-22 13:49:03.319			
2023-03-22 13:49:04.232	17	26	26
2023-03-22 13:49:04.233			
2023-03-22 13:49:05.080	17	26	26
2023-03-22 13:49:05.081			
2023-03-22 13:49:05.994	17	25	25
2023-03-22 13:49:05.994			
023-03-22 13:49:06.911	16	25	25
023-03-22 13:49:06.911			
023-03-22 13:49:07.825	17	26	26
023-03-22 13:49:07.825			
023-03-22 13:49:08.742	16	26	26
023-03-22 13:49:08.742			
023-03-22 13:49:09.655	16	24	24
023-03-22 13:49:09.655			
023-03-22 13:49:10.573	16	25	25
023-03-22 13:49:10.573			
023-03-22 13:49:11.486	17	24	24
023-03-22 13:49:11.490			
23-03-22 13:49:12.404	16	23	25
23-03-22 13:49:12.404			
23-03-22 13:49:13.317	16	24	25
23-03-22 13:49:13.321			
23-03-22 13:49:14.234	16	22	24
23-03-22 13:49:14.235			

Add timestamps ✓ Column separator \t (tab)
COM6
115200

Arduino Code:

```
#include <SoftwareSerial.h>

SoftwareSerial pmsSerial(2, 3);

 void setup() {

  // our debugging output

  Serial.begin(115200);



  // sensor baud rate is 9600

  pmsSerial.begin(9600);

}

 struct pms5003data {

  uint16_t framelen;

  uint16_t pm10_standard, pm25_standard, pm100_standard;

  uint16_t pm10_env, pm25_env, pm100_env;

  uint16_t particles_03um, particles_05um, particles_10um, particles_25um, particles_50um, particles_100um;

  uint16_t unused;

  uint16_t checksum;
```

```
};
 struct pms5003data data;
    void loop() {
  if (readPMSdata(&pmsSerial)) {
    Serial.print(data.particles_03um);
    Serial.print("\t");
    Serial.print(data.particles_05um);
    Serial.print("\t");
    Serial.print(data.particles_10um);
    Serial.print("\t");
    Serial.print(data.particles_25um);
    Serial.print("\t");
    Serial.print(data.particles_50um);
    Serial.print("\t");
    Serial.print(data.particles_100um);
    Serial.print("\t");
    Serial.print(data.pm10_env);
    Serial.print("\t");
    Serial.print(data.pm25_env);
    Serial.print("\t");
    Serial.println(data.pm100_env);
  }
}
 boolean readPMSdata(Stream *s) {
  if (! s->available()) {
```

```
  return false;
 }
 // Read a byte at a time until we get to the special '0x42' start-byte
 if (s->peek() != 0x42) {
  s->read();
  return false;
 }
 // Now read all 32 bytes
 if (s->available() < 32) {
 return false;
 }
  uint8_t buffer[32];
 uint16_t sum = 0;
 s->readBytes(buffer, 32);
 // get checksum ready
 for (uint8_t i=0; i<30; i++) {
  sum += buffer[i];
 }
// The data comes in endian'd, this solves it so it works on all platforms
 uint16_t buffer_u16[15];
 for (uint8_t i=0; i<15; i++) {
  buffer_u16[i] = buffer[2 + i*2 + 1];
  buffer_u16[i] += (buffer[2 + i*2] << 8);
 }
```

```
  // put it into a nice struct :)

  memcpy((void *)&data, (void *)buffer_u16, 30);



  if (sum != data.checksum) {

    Serial.println("Checksum failure");

    return false;

  }

  // success!

  return true;

}
```

In the above code **“data.pm10_env”** refers to PM1.0 concentration unit μg/m3（under atmospheric environment). Similarly, for **“data.pm25_env”** and **“data.pm100_env”.**

“data.particles_03um” indicates the number of particles with diameter beyond 0.3 um in 0.1 L of air. Similarly for the rest and **“data.particles_100um”** indicates the number of particles with diameter beyond 10um in 0.1 L of air.

<u>Polynomial Regression:</u>

```
import pandas as pd

import numpy as np

df = pd.read_csv("C:/Users/ Indu

Bhushan/OneDrive/Desktop/IoT/Project/test2.csv")

df.head()

#Dropping NaN values

df=df.dropna()

df.head()

#Dropping outliers rows of each column

micro1_mean=df["0.3um"].mean()

micro1_std=df["0.3um"].std()

micro2_mean=df["0.5um"].mean()

micro2_std=df["0.5um"].std()

micro3_mean=df["1um"].mean()
```

```
micro3_std=df["1um"].std()
```

```
micro4_mean=df["2.5um"].mean()

micro4_std=df["2.5um"].std()

micro5_mean=df["5um"].mean()

micro5_std=df["5um"].std()

micro6_mean=df["10um"].mean()

micro6_std=df["10um"].std()

pm1_mean=df["PM1.0"].mean()

pm1_std=df["PM1.0"].std()

pm2_mean=df["PM2.5"].mean()

pm2_std=df["PM2.5"].std()

pm3_mean=df["PM10"].mean()

pm3_std=df["PM10"].std()

df_cleaned=df[(df["0.3um"]>=(micro1_mean-
3*micro1_std))&(df["0.3um"]<=(micro1_mean+3*micro1_std))]

df_cleaned=df_cleaned[(df_cleaned["0.5um"]>=(micro2_mean-
3*micro2_std))&(df_cleaned["0.5um"]<=(micro2_mean+3*micro2_std))]

df_cleaned=df_cleaned[(df_cleaned["1um"]>=(micro3_mean-
3*micro3_std))&(df_cleaned["1um"]<=(micro3_mean+3*micro3_std))]

df_cleaned=df_cleaned[(df_cleaned["2.5um"]>=(micro4_mean-
3*micro4_std))&(df_cleaned["2.5um"]<=(micro4_mean+3*micro4_std))]

df_cleaned=df_cleaned[(df_cleaned["5um"]>=(micro5_mean-
3*micro5_std))&(df_cleaned["5um"]<=(micro5_mean+3*micro5_std))]

df_cleaned=df_cleaned[(df_cleaned["10um"]>=(micro6_mean-
3*micro6_std))&(df_cleaned["10um"]<=(micro6_mean+3*micro6_std))]

df_cleaned=df_cleaned[(df_cleaned["PM1.0"]>=(pm1_mean-
3*pm1_std))&(df_cleaned["PM1.0"]<=(pm1_mean+3*pm1_std))]

df_cleaned=df_cleaned[(df_cleaned["PM2.5"]>=(pm2_mean-
3*pm2_std))&(df_cleaned["PM2.5"]<=(pm2_mean+3*pm2_std))]

df_cleaned=df_cleaned[(df_cleaned["PM10"]>=(pm3_mean-
3*pm3_std))&(df_cleaned["PM10"]<=(pm3_mean+3*pm3_std))]

df_cleaned.head()

df1=df_cleaned.copy()

df1['Date Index']=np.arange(len(df1.index))

df1.head()

# Training data
```

```
X = df1.loc[:, ['Date Index']] # for x-axis
y1 = df1.loc[:, '0.3um'] #y-axis =0.3um
y2 = df1.loc[:, '0.5um']
y3 = df1.loc[:, '1um']
y4 = df1.loc[:, '2.5um']
y5 = df1.loc[:, '5um']
y6 = df1.loc[:, '10um']
y7 = df1.loc[:, 'PM1.0']
y8 = df1.loc[:, 'PM2.5']
y9 = df1.loc[:, 'PM10']
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=4, include_bias=False)
poly_features = poly.fit_transform(X)
poly_features
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
X_train, X_test, y1_train, y1_test = train_test_split(poly_features, y1, test_size=0.3, random_state=3)
poly_reg_model1 = LinearRegression()
poly_reg_model1.fit(X_train, y1_train)
poly_reg_y1_predicted = poly_reg_model1.predict(X_test)
poly_reg_y1_predicted
poly_reg_model1.score(X_test,y1_test)
X_train, X_test, y2_train, y2_test = train_test_split(poly_features, y2, test_size=0.3, random_state=3)
poly_reg_model2 = LinearRegression()
poly_reg_model2.fit(X_train, y2_train)
poly_reg_y2_predicted = poly_reg_model2.predict(X_test)
poly_reg_y2_predicted
poly_reg_model2.score(X_test,y2_test)
X_train, X_test, y3_train, y3_test = train_test_split(poly_features, y3, test_size=0.3, random_state=3)
poly_reg_model3 = LinearRegression()
poly_reg_model3.fit(X_train, y3_train)
```

```
poly_reg_y3_predicted = poly_reg_model3.predict(X_test)
poly_reg_y3_predicted
poly_reg_model3.score(X_test,y3_test)
X_train, X_test, y4_train, y4_test = train_test_split(poly_features, y4, test_size=0.3, random_state=3)
poly_reg_model4 = LinearRegression()
poly_reg_model4.fit(X_train, y4_train)
poly_reg_y4_predicted = poly_reg_model4.predict(X_test)
poly_reg_y4_predicted
poly_reg_model4.score(X_test,y4_test)
X_train, X_test, y5_train, y5_test = train_test_split(poly_features, y5, test_size=0.3, random_state=3)
poly_reg_model5 = LinearRegression()
poly_reg_model5.fit(X_train, y5_train)
poly_reg_y5_predicted = poly_reg_model5.predict(X_test)
poly_reg_y5_predicted
poly_reg_model5.score(X_test,y5_test)
X_train, X_test, y6_train, y6_test = train_test_split(poly_features, y6, test_size=0.3, random_state=3)
poly_reg_model6 = LinearRegression()
poly_reg_model6.fit(X_train, y6_train)
poly_reg_y6_predicted = poly_reg_model6.predict(X_test)
poly_reg_y6_predicted
poly_reg_model6.score(X_test,y6_test)
X_train, X_test, y7_train, y7_test = train_test_split(poly_features, y7, test_size=0.3, random_state=3)
poly_reg_model7 = LinearRegression()
poly_reg_model7.fit(X_train, y7_train)
poly_reg_y7_predicted = poly_reg_model7.predict(X_test)
poly_reg_y7_predicted
poly_reg_model7.score(X_test,y7_test)
X_train, X_test, y8_train, y8_test = train_test_split(poly_features, y8, test_size=0.3, random_state=3)
poly_reg_model8 = LinearRegression()
poly_reg_model8.fit(X_train, y8_train)
poly_reg_y8_predicted = poly_reg_model8.predict(X_test)
poly_reg_y8_predicted
```

```
poly_reg_model8.score(X_test,y8_test)
X_train, X_test, y9_train, y9_test = train_test_split(poly_features, y9, test_size=0.3, random_state=3)
poly_reg_model9 = LinearRegression()
poly_reg_model9.fit(X_train, y9_train)
poly_reg_y9_predicted = poly_reg_model9.predict(X_test)
poly_reg_y9_predicted
poly_reg_model9.score(X_test,y9_test)
plt.title('0.3um')
plt.scatter(X_test[:,0],y1_test)
plt.scatter(X_test[:,0],poly_reg_y1_predicted,c="red")
plt.show()
plt.title('0.5um')
plt.scatter(X_test[:,0],y2_test)
plt.scatter(X_test[:,0],poly_reg_y2_predicted,c="red")
plt.show()
plt.title('1um')
plt.scatter(X_test[:,0],y3_test)
plt.scatter(X_test[:,0],poly_reg_y3_predicted,c="red")
plt.show()
plt.title('2.5um')
plt.scatter(X_test[:,0],y4_test)
plt.scatter(X_test[:,0],poly_reg_y4_predicted,c="red")
plt.show()
plt.title('5um')
plt.scatter(X_test[:,0],y5_test)
plt.scatter(X_test[:,0],poly_reg_y5_predicted,c="red")
plt.show()
plt.title('10um')
plt.scatter(X_test[:,0],y6_test)
plt.scatter(X_test[:,0],poly_reg_y6_predicted,c="red")
plt.show()
plt.title('PM1.0')
plt.scatter(X_test[:,0],y7_test)

plt.scatter(X_test[:,0],poly_reg_y7_predicted,c="red")

plt.show()

plt.title('PM2.5')

plt.scatter(X_test[:,0],y8_test)

plt.scatter(X_test[:,0],poly_reg_y8_predicted,c="red")

plt.show()

plt.title('PM10')

plt.scatter(X_test[:,0],y9_test)

plt.scatter(X_test[:,0],poly_reg_y9_predicted,c="red")

plt.show()

#Example prediction

input_index=17400

input_poly_reg=[[input_index,input_index**2,input_index**3,input_index**4]]

predicted_PM10 = poly_reg_model9.predict(input_poly_reg)

predicted_PM10
```

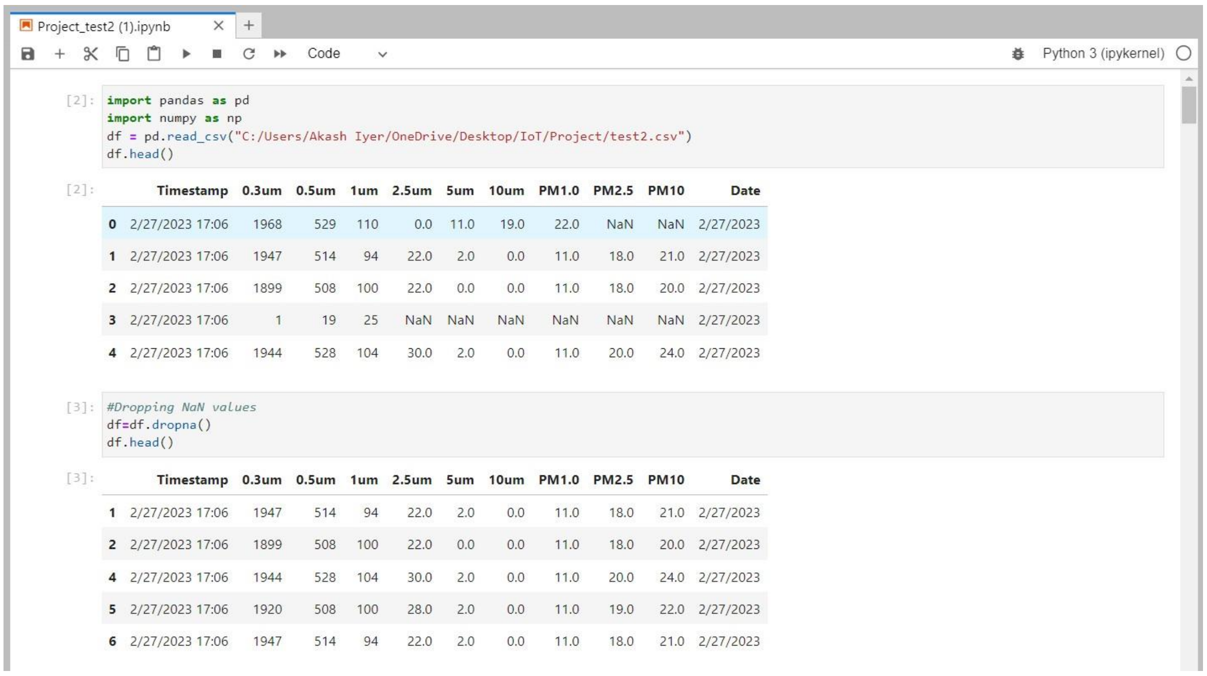
Code Explanation:

- In the above ML program, we first import the pandas and numpy libraries in order to form and then clean the dataframe.
- After creating the dataframe that contains all the obtained data, the rows that contain NaN (not a number) values are removed. NaN values are generally obtained when the sensor initially starts collecting data each day because it needs some time to stabilize.
- After this, our data is cleaned to remove outliers. Here, we consider data values that do not lie between (mean – 3*std_dev) to (mean + 3*std_dev) to be outliers and remove all rows that have such values.
- A copy of the cleaned data frame is made and every entry is indexed. This is done because all future values will be predicted based on the index value. In the data collected, we have around 500 indices per day.
- Now, each parameter value (column) is extracted separately and stored in a variable. The index (which is indicative of the date) is also stored in a separate variable and acts as the independent variable based on which predictions are made.
- PolynomialFeatures is imported from the sklearn.preprocessing library. In the above given program, we use a 4th degree polynomial to model the collected data and make predictions about it.
- LinearRegression is imported from the sklearn.linear_model library. Furthermore, train_test_split is imported from the sklearn.model_selection library and the matplotlib.pyplot library is also imported
- 70 percent of the data is used to train the polynomial regression model for each of the 9 dependent variables. Then, the model is fit to each of the variables and the accuracy of the model for each variable is found.

- Lastly, both the actual data obtained as well as the polynomial regression model are plotted against the index values to compare the two.
- The code also shows an example of how the model is used to predict future values of a parameter by providing an input index value.

<u>**Outputs Obtained:**</u>

Given below are screenshots of the code written as well as the corresponding outputs obtained when each block of code is run. An explanation is also provided if required



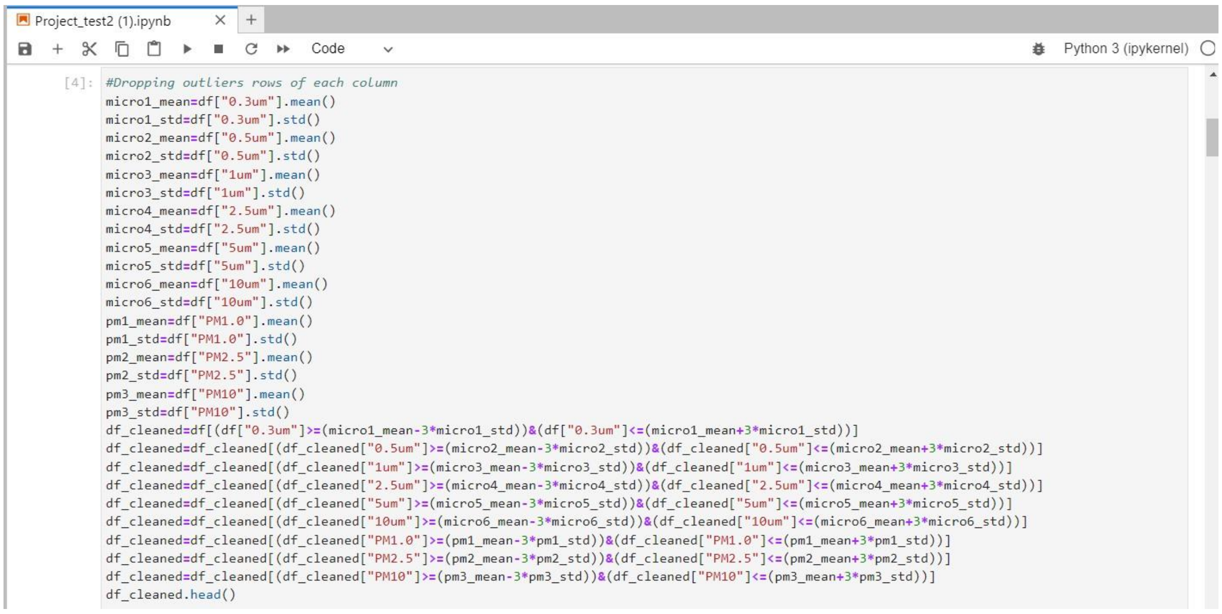
```
[2]: import pandas as pd
     import numpy as np
     df = pd.read_csv("C:/Users/Akash Iyer/OneDrive/Desktop/IoT/Project/test2.csv")
     df.head()
```

```
[2]:
         Timestamp  0.3um  0.5um  1um  2.5um   5um  10um  PM1.0  PM2.5  PM10       Date
0  2/27/2023 17:06   1968    529  110    0.0  11.0  19.0   22.0    NaN   NaN  2/27/2023
1  2/27/2023 17:06   1947    514   94   22.0   2.0   0.0   11.0   18.0  21.0  2/27/2023
2  2/27/2023 17:06   1899    508  100   22.0   0.0   0.0   11.0   18.0  20.0  2/27/2023
3  2/27/2023 17:06      1     19   25    NaN   NaN   NaN    NaN    NaN   NaN  2/27/2023
4  2/27/2023 17:06   1944    528  104   30.0   2.0   0.0   11.0   20.0  24.0  2/27/2023
```

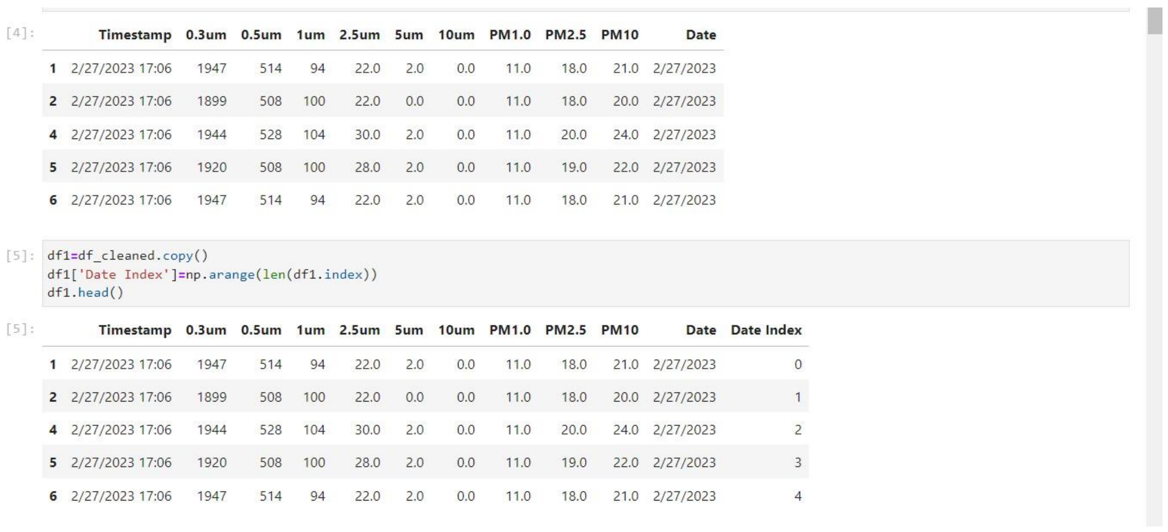
```
[3]: #Dropping NaN values
     df=df.dropna()
     df.head()
```

```
[3]:
         Timestamp  0.3um  0.5um  1um  2.5um   5um  10um  PM1.0  PM2.5  PM10       Date
1  2/27/2023 17:06   1947    514   94   22.0   2.0   0.0   11.0   18.0  21.0  2/27/2023
2  2/27/2023 17:06   1899    508  100   22.0   0.0   0.0   11.0   18.0  20.0  2/27/2023
4  2/27/2023 17:06   1944    528  104   30.0   2.0   0.0   11.0   20.0  24.0  2/27/2023
5  2/27/2023 17:06   1920    508  100   28.0   2.0   0.0   11.0   19.0  22.0  2/27/2023
6  2/27/2023 17:06   1947    514   94   22.0   2.0   0.0   11.0   18.0  21.0  2/27/2023
```

In the above, it is seen that rows with NaN values are dropped. This can be verified by noticing the internally provided index values to each row of the dataframe before and after dropping them.



```
[4]: #Dropping outliers rows of each column
     micro1_mean=df["0.3um"].mean()
     micro1_std=df["0.3um"].std()
     micro2_mean=df["0.5um"].mean()
     micro2_std=df["0.5um"].std()
     micro3_mean=df["1um"].mean()
     micro3_std=df["1um"].std()
     micro4_mean=df["2.5um"].mean()
     micro4_std=df["2.5um"].std()
     micro5_mean=df["5um"].mean()
     micro5_std=df["5um"].std()
     micro6_mean=df["10um"].mean()
     micro6_std=df["10um"].std()
     pm1_mean=df["PM1.0"].mean()
     pm1_std=df["PM1.0"].std()
     pm2_mean=df["PM2.5"].mean()
     pm2_std=df["PM2.5"].std()
     pm3_mean=df["PM10"].mean()
     pm3_std=df["PM10"].std()
     df_cleaned=df[(df["0.3um"]>=(micro1_mean-3*micro1_std))&(df["0.3um"]<=(micro1_mean+3*micro1_std))]
     df_cleaned=df_cleaned[(df_cleaned["0.5um"]>=(micro2_mean-3*micro2_std))&(df_cleaned["0.5um"]<=(micro2_mean+3*micro2_std))]
     df_cleaned=df_cleaned[(df_cleaned["1um"]>=(micro3_mean-3*micro3_std))&(df_cleaned["1um"]<=(micro3_mean+3*micro3_std))]
     df_cleaned=df_cleaned[(df_cleaned["2.5um"]>=(micro4_mean-3*micro4_std))&(df_cleaned["2.5um"]<=(micro4_mean+3*micro4_std))]
     df_cleaned=df_cleaned[(df_cleaned["5um"]>=(micro5_mean-3*micro5_std))&(df_cleaned["5um"]<=(micro5_mean+3*micro5_std))]
     df_cleaned=df_cleaned[(df_cleaned["10um"]>=(micro6_mean-3*micro6_std))&(df_cleaned["10um"]<=(micro6_mean+3*micro6_std))]
     df_cleaned=df_cleaned[(df_cleaned["PM1.0"]>=(pm1_mean-3*pm1_std))&(df_cleaned["PM1.0"]<=(pm1_mean+3*pm1_std))]
     df_cleaned=df_cleaned[(df_cleaned["PM2.5"]>=(pm2_mean-3*pm2_std))&(df_cleaned["PM2.5"]<=(pm2_mean+3*pm2_std))]
     df_cleaned=df_cleaned[(df_cleaned["PM10"]>=(pm3_mean-3*pm3_std))&(df_cleaned["PM10"]<=(pm3_mean+3*pm3_std))]
     df_cleaned.head()
```



[4]:

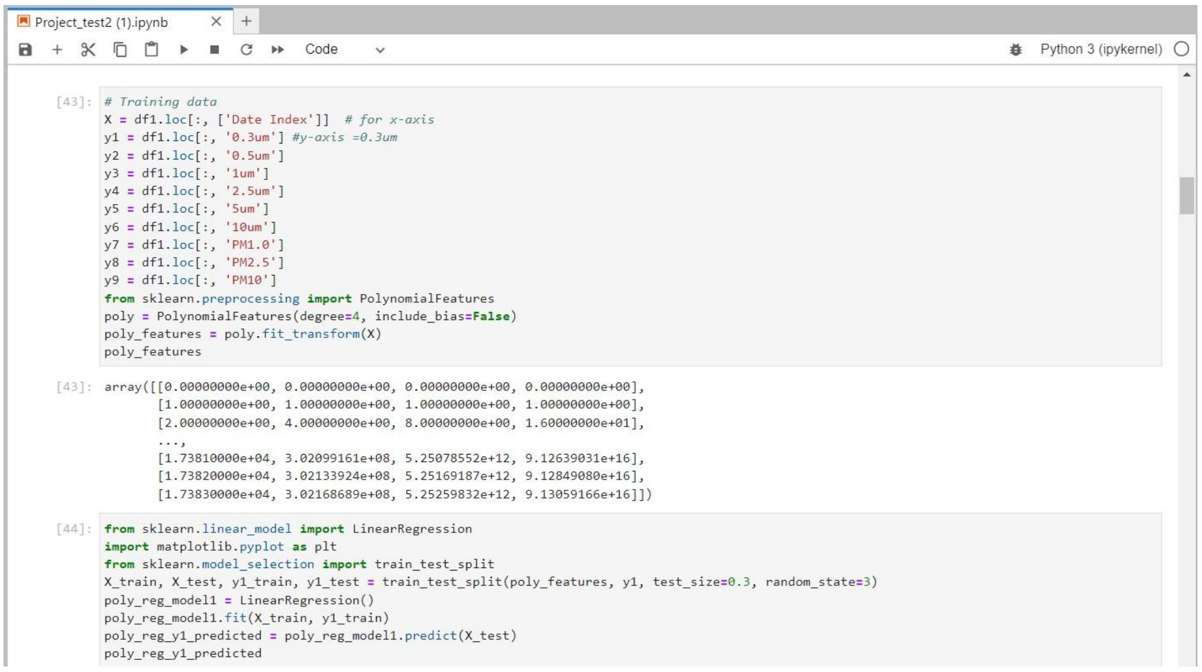
	Timestamp	0.3um	0.5um	1um	2.5um	5um	10um	PM1.0	PM2.5	PM10	Date
1	2/27/2023 17:06	1947	514	94	22.0	2.0	0.0	11.0	18.0	21.0	2/27/2023
2	2/27/2023 17:06	1899	508	100	22.0	0.0	0.0	11.0	18.0	20.0	2/27/2023
4	2/27/2023 17:06	1944	528	104	30.0	2.0	0.0	11.0	20.0	24.0	2/27/2023
5	2/27/2023 17:06	1920	508	100	28.0	2.0	0.0	11.0	19.0	22.0	2/27/2023
6	2/27/2023 17:06	1947	514	94	22.0	2.0	0.0	11.0	18.0	21.0	2/27/2023

```
[5]: df1=df_cleaned.copy()
     df1['Date Index']=np.arange(len(df1.index))
     df1.head()
```

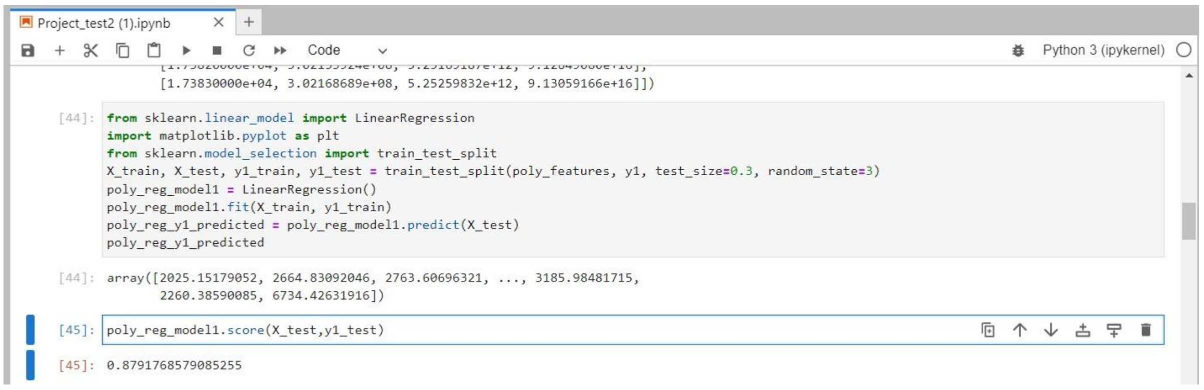
[5]:

	Timestamp	0.3um	0.5um	1um	2.5um	5um	10um	PM1.0	PM2.5	PM10	Date	Date Index
1	2/27/2023 17:06	1947	514	94	22.0	2.0	0.0	11.0	18.0	21.0	2/27/2023	0
2	2/27/2023 17:06	1899	508	100	22.0	0.0	0.0	11.0	18.0	20.0	2/27/2023	1
4	2/27/2023 17:06	1944	528	104	30.0	2.0	0.0	11.0	20.0	24.0	2/27/2023	2
5	2/27/2023 17:06	1920	508	100	28.0	2.0	0.0	11.0	19.0	22.0	2/27/2023	3
6	2/27/2023 17:06	1947	514	94	22.0	2.0	0.0	11.0	18.0	21.0	2/27/2023	4

The above two screenshots depict the cleaning of the dataframe by removing outliers as explained earlier. Following this, a copy of the cleaned dataframe is made and index values are assigned to each row based on which predictions are made.

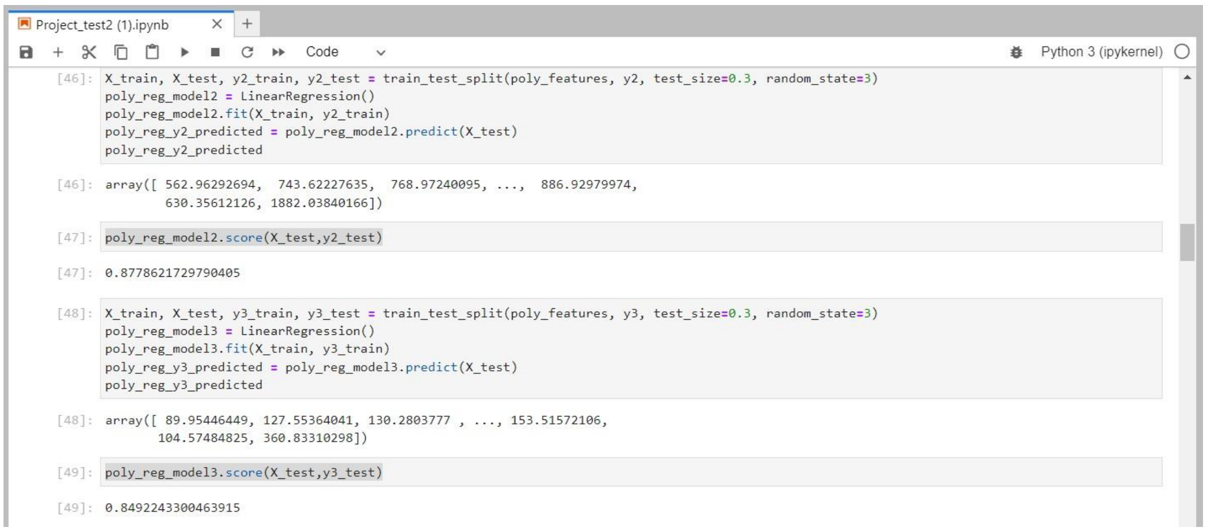


```
[43]: # Training data
      X = df1.loc[:, ['Date Index']]  # for x-axis
      y1 = df1.loc[:, '0.3um'] #y-axis =0.3um
      y2 = df1.loc[:, '0.5um']
      y3 = df1.loc[:, '1um']
      y4 = df1.loc[:, '2.5um']
      y5 = df1.loc[:, '5um']
      y6 = df1.loc[:, '10um']
      y7 = df1.loc[:, 'PM1.0']
      y8 = df1.loc[:, 'PM2.5']
      y9 = df1.loc[:, 'PM10']
      from sklearn.preprocessing import PolynomialFeatures
      poly = PolynomialFeatures(degree=4, include_bias=False)
      poly_features = poly.fit_transform(X)
      poly_features

[43]: array([[0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
             [1.00000000e+00, 1.00000000e+00, 1.00000000e+00, 1.00000000e+00],
             [2.00000000e+00, 4.00000000e+00, 8.00000000e+00, 1.60000000e+01],
             ...,
             [1.73810000e+04, 3.02099161e+08, 5.25078552e+12, 9.12639031e+16],
             [1.73820000e+04, 3.02133924e+08, 5.25169187e+12, 9.12849080e+16],
             [1.73830000e+04, 3.02168689e+08, 5.25259832e+12, 9.13059166e+16]])

[44]: from sklearn.linear_model import LinearRegression
      import matplotlib.pyplot as plt
      from sklearn.model_selection import train_test_split
      X_train, X_test, y1_train, y1_test = train_test_split(poly_features, y1, test_size=0.3, random_state=3)
      poly_reg_model1 = LinearRegression()
      poly_reg_model1.fit(X_train, y1_train)
      poly_reg_y1_predicted = poly_reg_model1.predict(X_test)
      poly_reg_y1_predicted
```



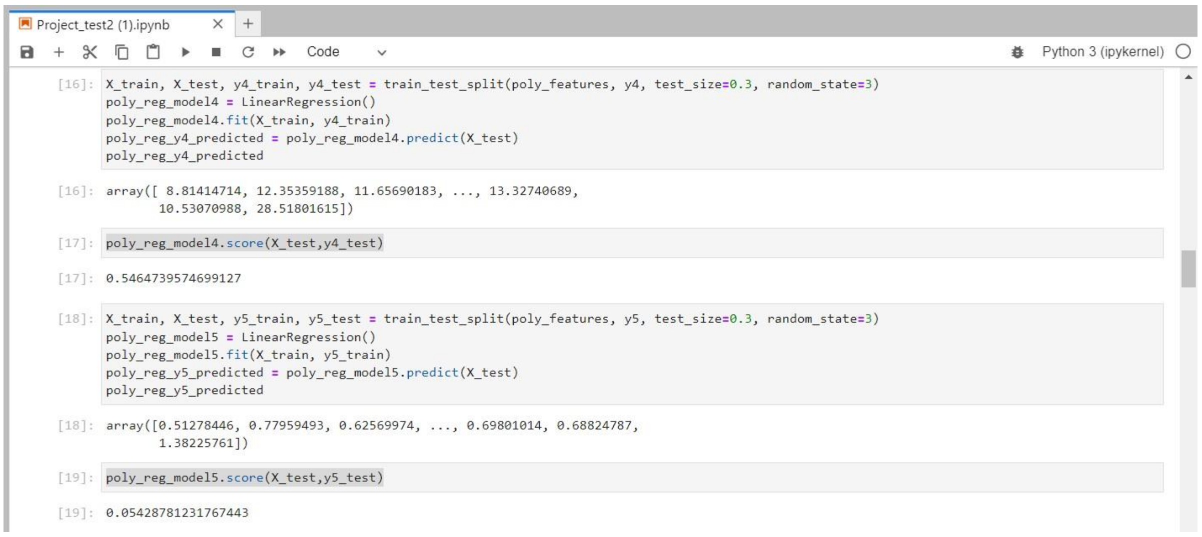
```
             [1.73830000e+04, 3.02168689e+08, 5.25259832e+12, 9.13059166e+16]])

[44]: from sklearn.linear_model import LinearRegression
      import matplotlib.pyplot as plt
      from sklearn.model_selection import train_test_split
      X_train, X_test, y1_train, y1_test = train_test_split(poly_features, y1, test_size=0.3, random_state=3)
      poly_reg_model1 = LinearRegression()
      poly_reg_model1.fit(X_train, y1_train)
      poly_reg_y1_predicted = poly_reg_model1.predict(X_test)
      poly_reg_y1_predicted

[44]: array([2025.15179052, 2664.83092046, 2763.60696321, ..., 3185.98481715,
             2260.38590085, 6734.42631916])

[45]: poly_reg_model1.score(X_test,y1_test)

[45]: 0.8791768579085255
```

The above screenshots depict the training and testing of the 4th degree polynomial regression model for the '0.3um' variable. As seen, the model possesses an accuracy of around 87.92%.


Project_test2 (1).ipynb

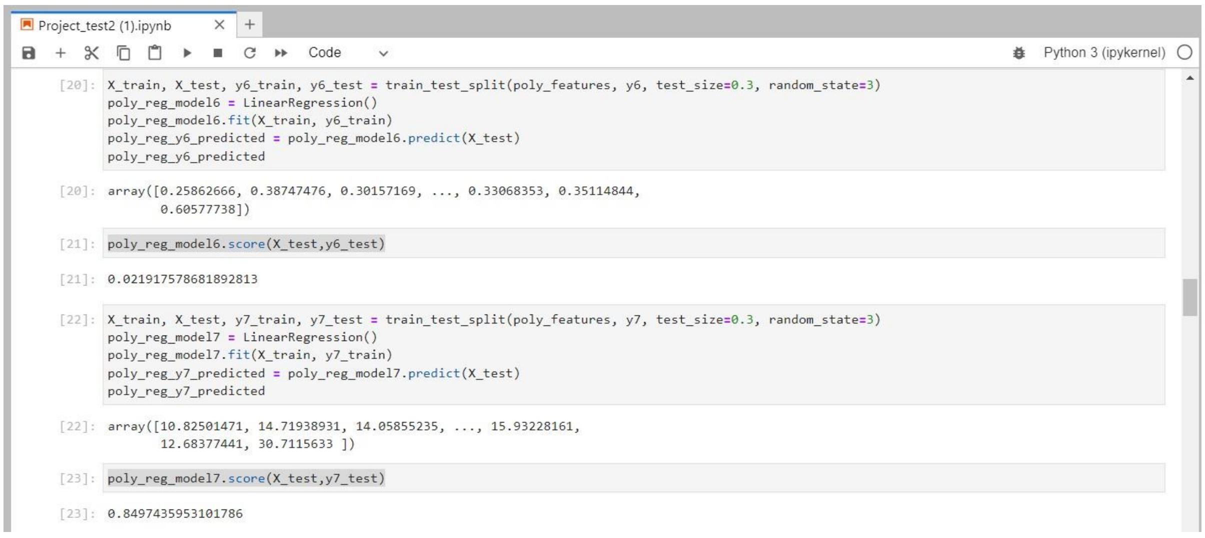
```
[46]: X_train, X_test, y2_train, y2_test = train_test_split(poly_features, y2, test_size=0.3, random_state=3)
      poly_reg_model2 = LinearRegression()
      poly_reg_model2.fit(X_train, y2_train)
      poly_reg_y2_predicted = poly_reg_model2.predict(X_test)
      poly_reg_y2_predicted

[46]: array([ 562.96292694,  743.62227635,  768.97240095, ...,  886.92979974,
              630.35612126, 1882.03840166])

[47]: poly_reg_model2.score(X_test,y2_test)

[47]: 0.8778621729790405

[48]: X_train, X_test, y3_train, y3_test = train_test_split(poly_features, y3, test_size=0.3, random_state=3)
      poly_reg_model3 = LinearRegression()
      poly_reg_model3.fit(X_train, y3_train)
      poly_reg_y3_predicted = poly_reg_model3.predict(X_test)
      poly_reg_y3_predicted

[48]: array([ 89.95446449, 127.55364041, 130.2803777 , ..., 153.51572106,
             104.57484825, 360.83310298])

[49]: poly_reg_model3.score(X_test,y3_test)

[49]: 0.8492243300463915
```

The accuracy of the model for the ‘0.5um’ variable is around 87.79% while that of the model for the ‘1um’ variable is 84.92%


Project_test2 (1).ipynb

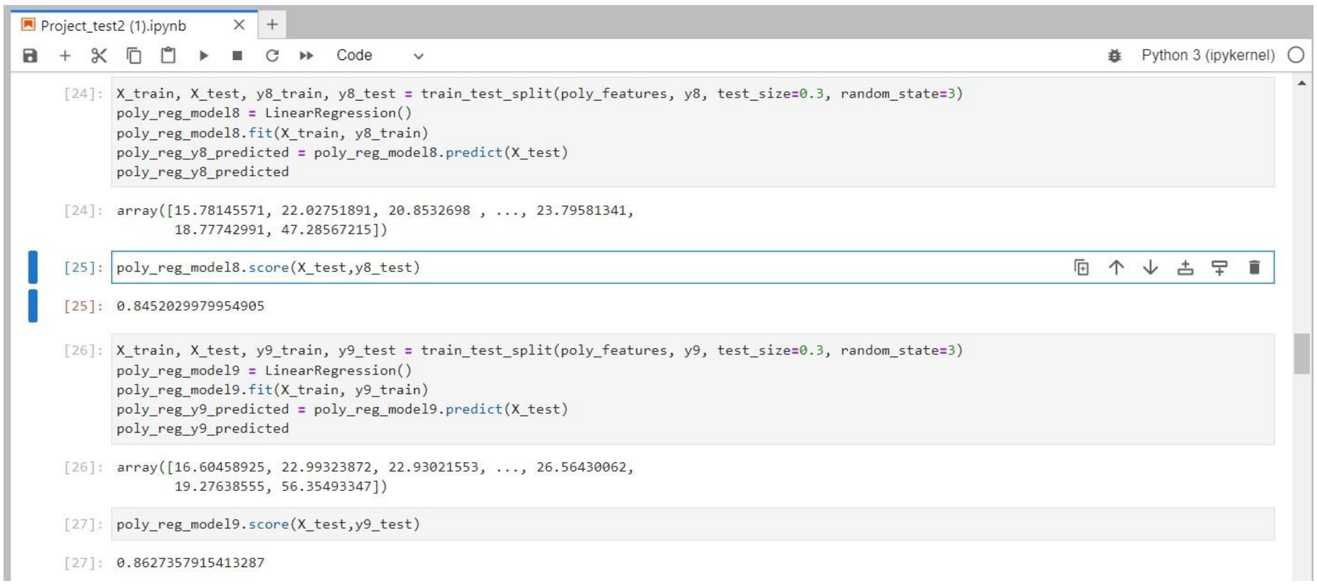
```
[16]: X_train, X_test, y4_train, y4_test = train_test_split(poly_features, y4, test_size=0.3, random_state=3)
      poly_reg_model4 = LinearRegression()
      poly_reg_model4.fit(X_train, y4_train)
      poly_reg_y4_predicted = poly_reg_model4.predict(X_test)
      poly_reg_y4_predicted

[16]: array([ 8.81414714, 12.35359188, 11.65690183, ..., 13.32740689,
             10.53070988, 28.51801615])

[17]: poly_reg_model4.score(X_test,y4_test)

[17]: 0.5464739574699127

[18]: X_train, X_test, y5_train, y5_test = train_test_split(poly_features, y5, test_size=0.3, random_state=3)
      poly_reg_model5 = LinearRegression()
      poly_reg_model5.fit(X_train, y5_train)
      poly_reg_y5_predicted = poly_reg_model5.predict(X_test)
      poly_reg_y5_predicted

[18]: array([0.51278446, 0.77959493, 0.62569974, ..., 0.69801014, 0.68824787,
             1.38225761])

[19]: poly_reg_model5.score(X_test,y5_test)

[19]: 0.05428781231767443
```

The accuracy of the model for the ‘2.5um’ variable is around 54.65% while that of the model for the ‘5um’ variable is just 5.43%. The low accuracies obtained for these are explained when the model and the scatterplots for the variables are shown further on.


Project_test2 (1).ipynb

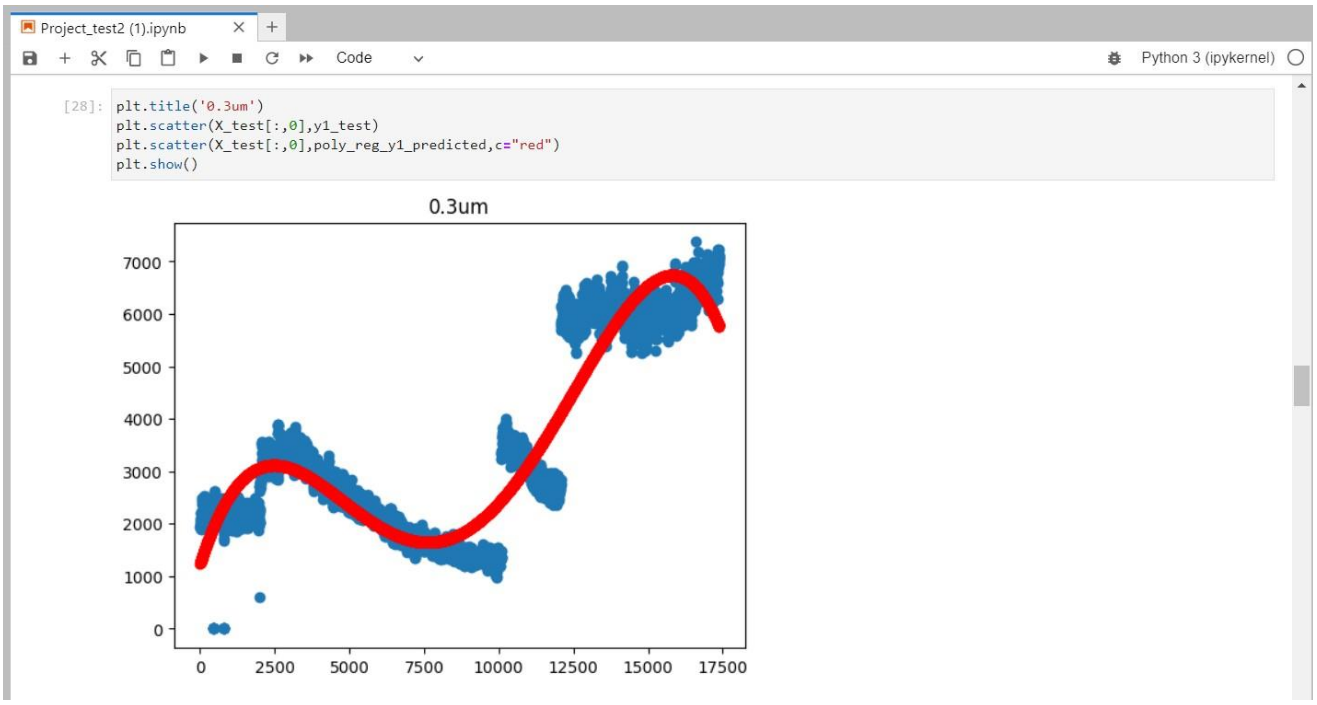
```
[20]: X_train, X_test, y6_train, y6_test = train_test_split(poly_features, y6, test_size=0.3, random_state=3)
      poly_reg_model6 = LinearRegression()
      poly_reg_model6.fit(X_train, y6_train)
      poly_reg_y6_predicted = poly_reg_model6.predict(X_test)
      poly_reg_y6_predicted

[20]: array([0.25862666, 0.38747476, 0.30157169, ..., 0.33068353, 0.35114844,
             0.60577738])

[21]: poly_reg_model6.score(X_test,y6_test)

[21]: 0.021917578681892813

[22]: X_train, X_test, y7_train, y7_test = train_test_split(poly_features, y7, test_size=0.3, random_state=3)
      poly_reg_model7 = LinearRegression()
      poly_reg_model7.fit(X_train, y7_train)
      poly_reg_y7_predicted = poly_reg_model7.predict(X_test)
      poly_reg_y7_predicted

[22]: array([10.82501471, 14.71938931, 14.05855235, ..., 15.93228161,
             12.68377441, 30.7115633 ])

[23]: poly_reg_model7.score(X_test,y7_test)

[23]: 0.8497435953101786
```

The accuracy of the model for the variable ‘10um’ is just 2.19%. Again, this is explained later on. The accuracy of the model for the ‘PM1.0’ variable is around 84.97%.



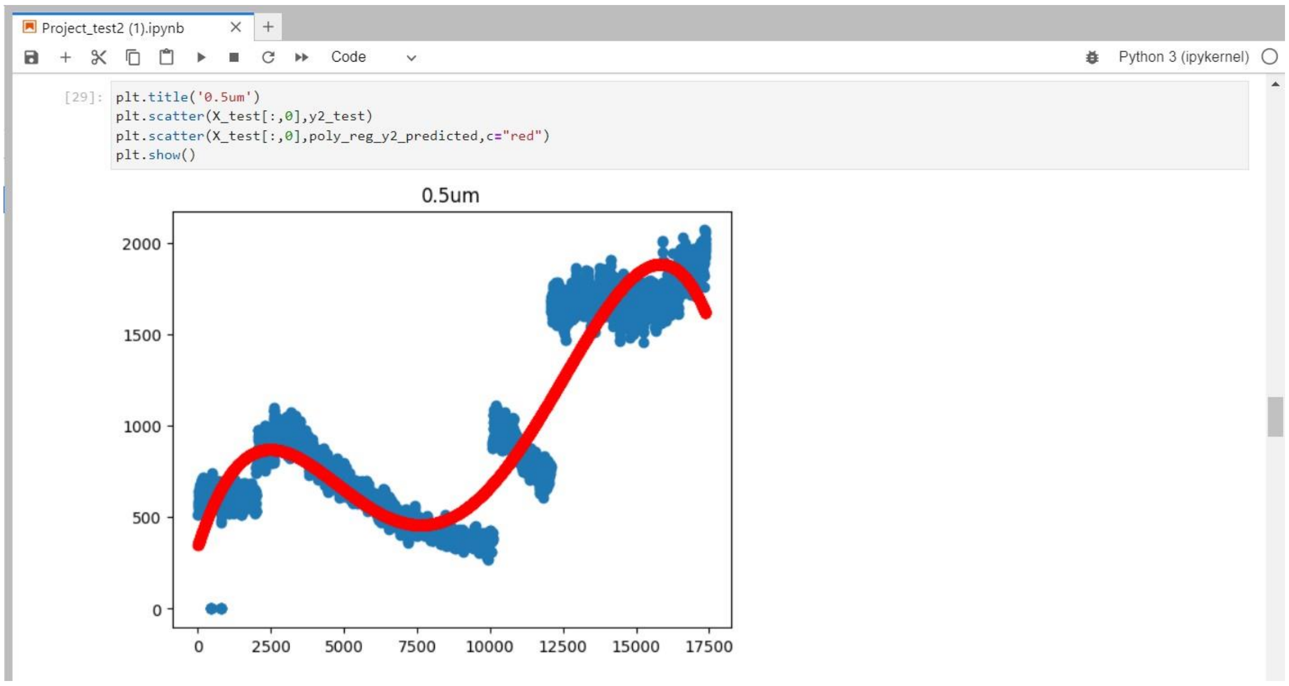
```
[24]: X_train, X_test, y8_train, y8_test = train_test_split(poly_features, y8, test_size=0.3, random_state=3)
      poly_reg_model8 = LinearRegression()
      poly_reg_model8.fit(X_train, y8_train)
      poly_reg_y8_predicted = poly_reg_model8.predict(X_test)
      poly_reg_y8_predicted

[24]: array([15.78145571, 22.02751891, 20.8532698 , ..., 23.79581341,
             18.77742991, 47.28567215])

[25]: poly_reg_model8.score(X_test,y8_test)

[25]: 0.8452029979954905

[26]: X_train, X_test, y9_train, y9_test = train_test_split(poly_features, y9, test_size=0.3, random_state=3)
      poly_reg_model9 = LinearRegression()
      poly_reg_model9.fit(X_train, y9_train)
      poly_reg_y9_predicted = poly_reg_model9.predict(X_test)
      poly_reg_y9_predicted

[26]: array([16.60458925, 22.99323872, 22.93021553, ..., 26.56430062,
             19.27638555, 56.35493347])

[27]: poly_reg_model9.score(X_test,y9_test)

[27]: 0.8627357915413287
```

The accuracy of the model for the ‘PM2.5’ variable is around 84.52% while the accuracy for the ‘PM10’ variable is 86.27%.



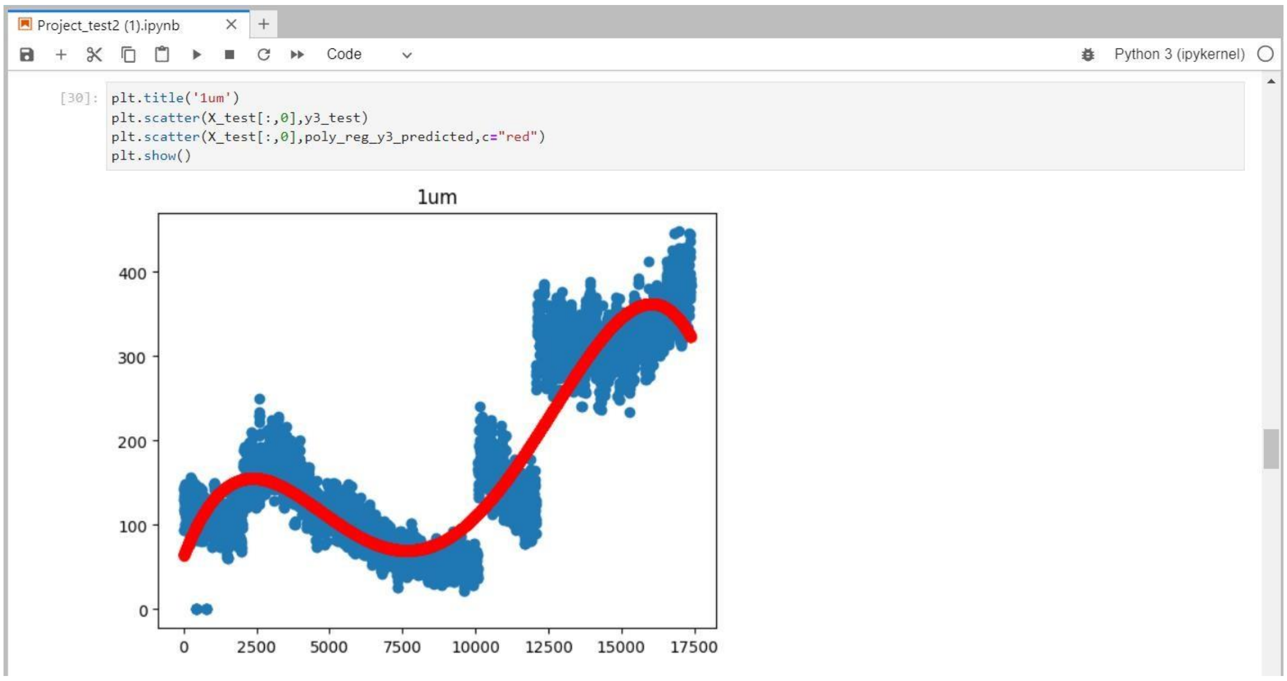
```
[28]: plt.title('0.3um')
      plt.scatter(X_test[:,0],y1_test)
      plt.scatter(X_test[:,0],poly_reg_y1_predicted,c="red")
      plt.show()
```

Given above is the scatterplot and regression curve for the ‘0.3um’ variable.



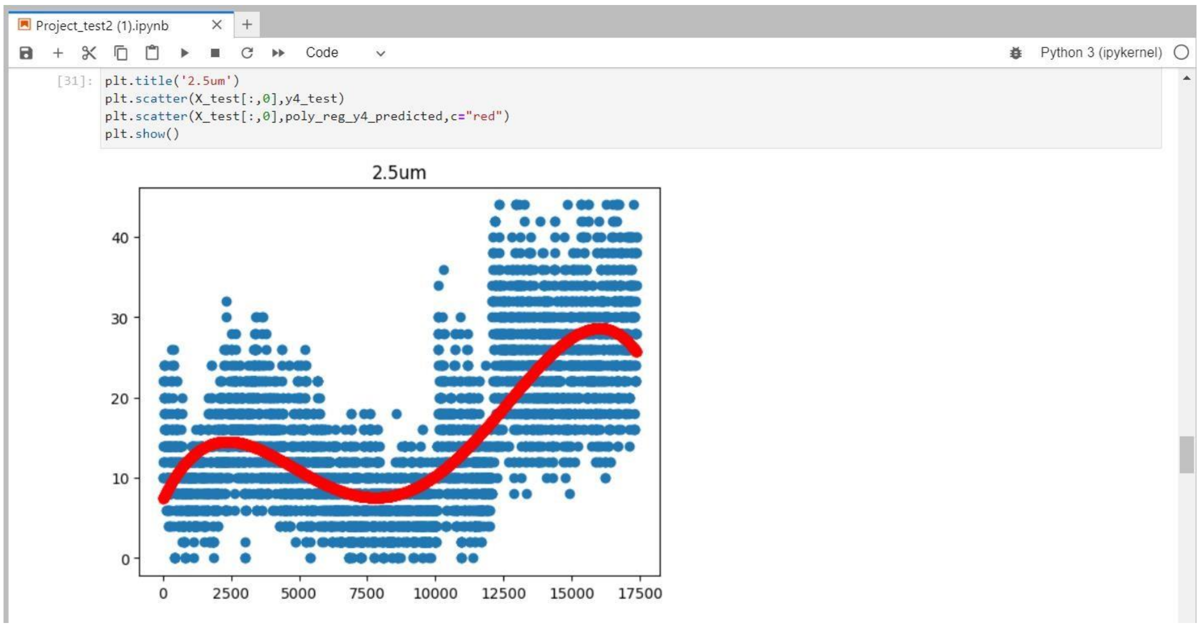
```
[29]: plt.title('0.5um')
      plt.scatter(X_test[:,0],y2_test)
      plt.scatter(X_test[:,0],poly_reg_y2_predicted,c="red")
      plt.show()
```

Given above is the scatterplot and regression curve for the ‘0.5um’ variable.



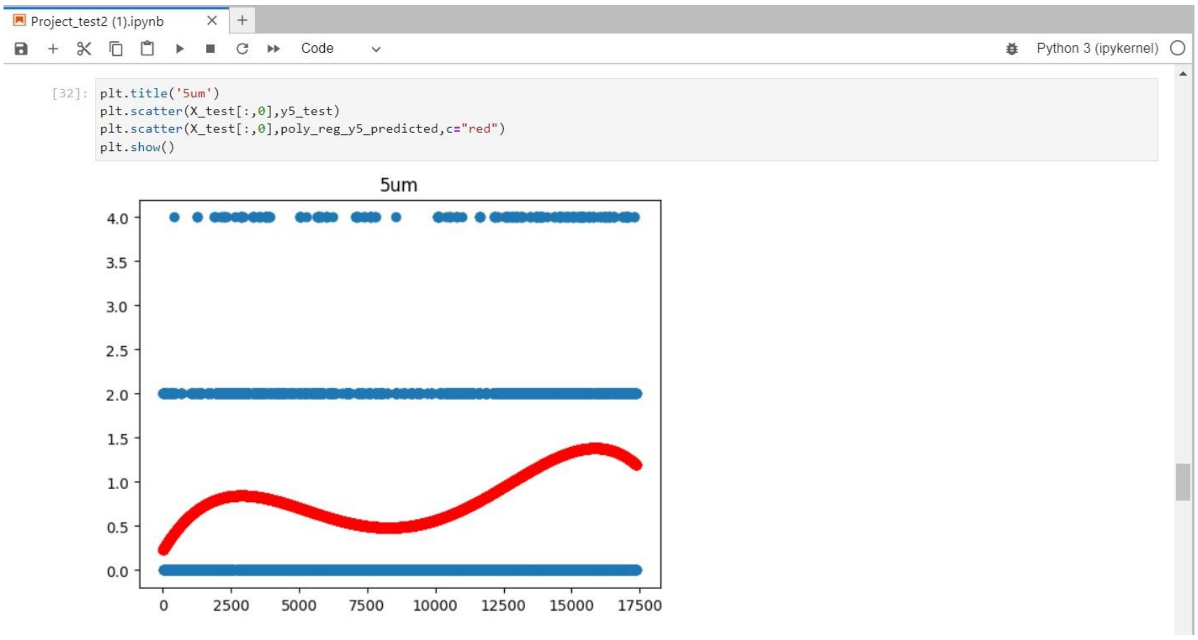
```
[30]: plt.title('1um')
      plt.scatter(X_test[:,0],y3_test)
      plt.scatter(X_test[:,0],poly_reg_y3_predicted,c="red")
      plt.show()
```

Given above is the scatterplot and regression curve for the ‘1um’ variable.



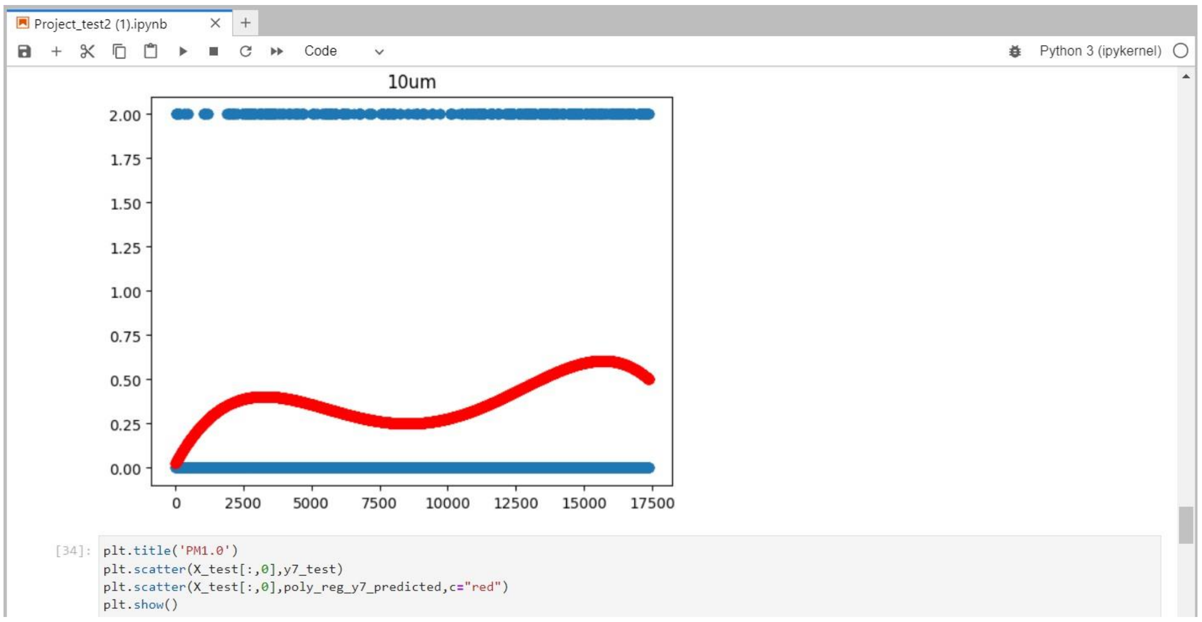
```
plt.title('2.5um')
plt.scatter(X_test[:,0],y4_test)
plt.scatter(X_test[:,0],poly_reg_y4_predicted,c="red")
plt.show()
```

Given above is the scatterplot and regression curve for the ‘2.5um’ variable. From the scatterplot, it is observed that there are far fewer distinct values at a much lower scale than those of the previous variables. Thus, a deviation from the regression curve has much more impact on the accuracy of the model as compared to before, thus leading to the lower accuracy of 54.65%.



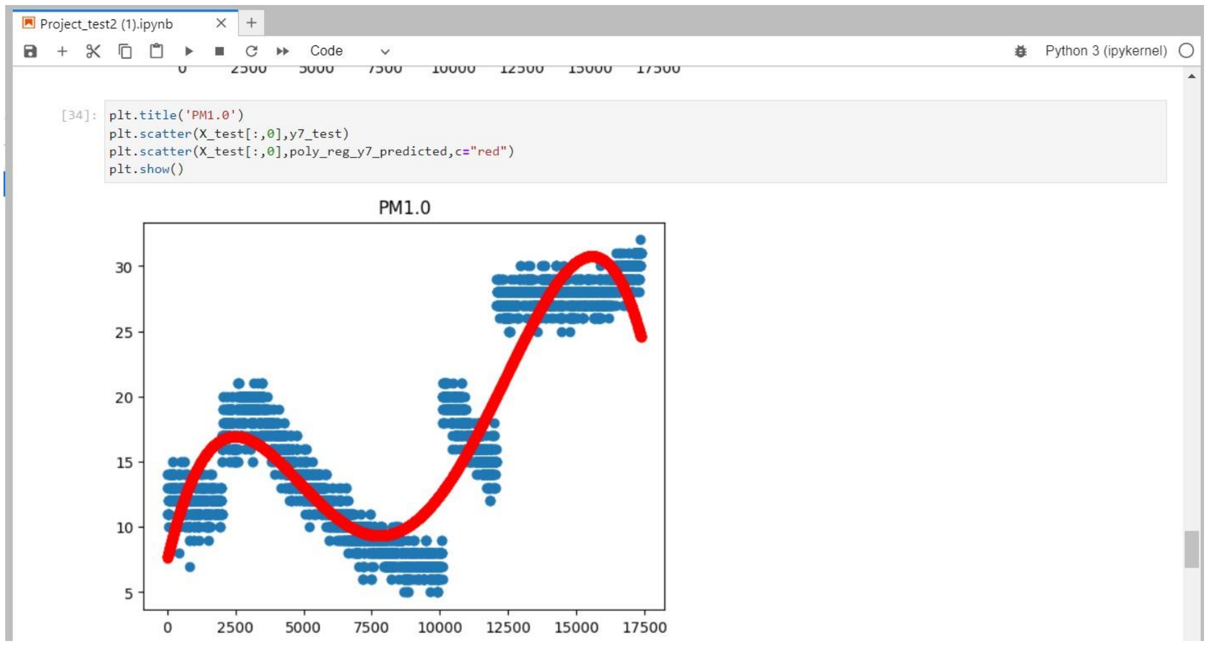
```
plt.title('5um')
plt.scatter(X_test[:,0],y5_test)
plt.scatter(X_test[:,0],poly_reg_y5_predicted,c="red")
plt.show()
```

From the above, even without the accuracy score, it is clear that the model for the ‘5um’ variable is very inaccurate. This can be explained by the fact that there are exactly 3 distinct values that the variable actually takes and at a scale very small deviations produce very high error. The regression curve, being continuous, cannot produce just 3 individual values and as such, cannot be accurate.



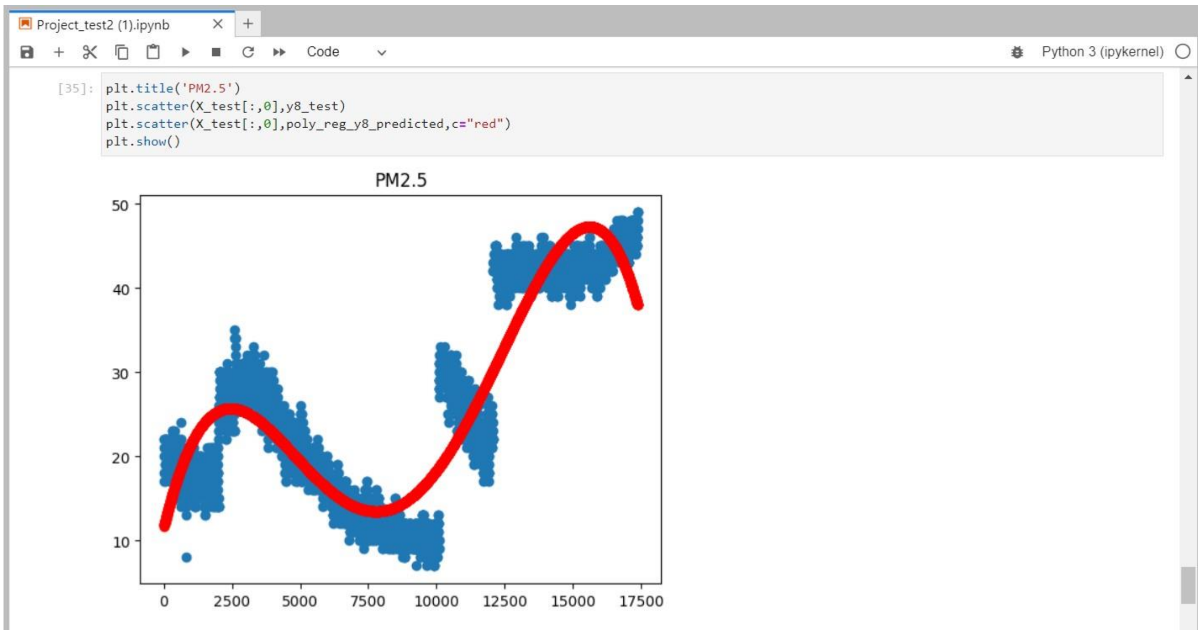
```
[34]: plt.title('PM1.0')
      plt.scatter(X_test[:,0],y7_test)
      plt.scatter(X_test[:,0],poly_reg_y7_predicted,c="red")
      plt.show()
```

The 10um parameter takes just two values. As such, the regression model is wildly inaccurate for the same reason that the 5um model is.



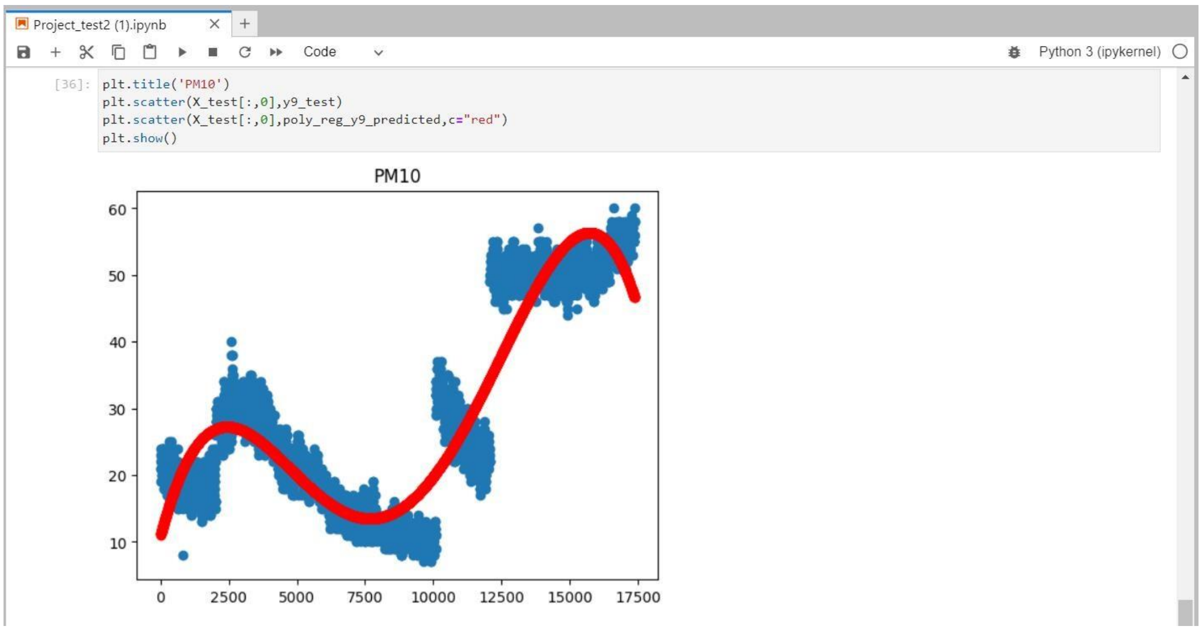
```
[34]: plt.title('PM1.0')
      plt.scatter(X_test[:,0],y7_test)
      plt.scatter(X_test[:,0],poly_reg_y7_predicted,c="red")
      plt.show()
```

The above depicts the regression model and the scatterplot of the PM1.0 parameter. It is clear that the model is fairly accurate, and this is seen when finding the accuracy score of the same to be around 84.97%.



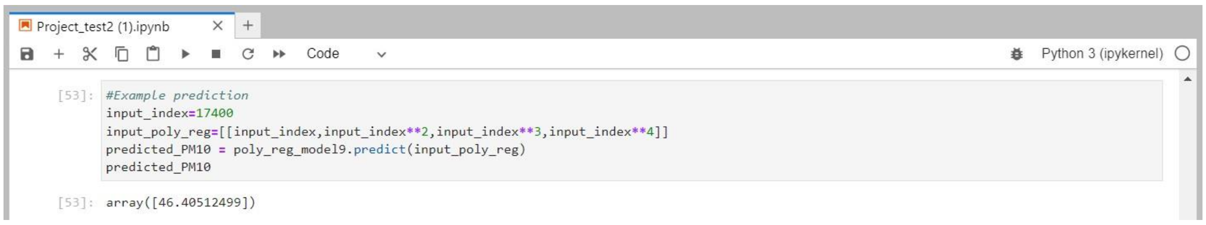
```
[35]: plt.title('PM2.5')
      plt.scatter(X_test[:,0],y8_test)
      plt.scatter(X_test[:,0],poly_reg_y8_predicted,c="red")
      plt.show()
```

The above depicts the regression model and the scatterplot of the PM2.5 parameter.



```
[36]: plt.title('PM10')
      plt.scatter(X_test[:,0],y9_test)
      plt.scatter(X_test[:,0],poly_reg_y9_predicted,c="red")
      plt.show()
```

The above shows the scatterplot and the regression curve of the PM10 parameter.



```
[53]: #Example prediction
      input_index=17400
      input_poly_reg=[[input_index,input_index**2,input_index**3,input_index**4]]
      predicted_PM10 = poly_reg_model9.predict(input_poly_reg)
      predicted_PM10

[53]: array([46.40512499])
```

The above shows an example prediction made using the obtained regression models. Data was collected upto a little above index 17000. Thus, the above shows that for an index of 17400 (which is beyond the index of the last data entry obtained from the hardware), the predicted PM10 value is around 46.405 µg/m^3 (standard units).

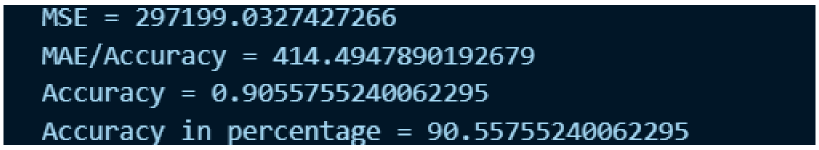
Limitations of using Polynomial Regression:

- The presence of one or two outliers in the data can seriously affect the results of the nonlinear analysis.
- There are fewer model validation tools for the detection of outliers in nonlinear regression than there are for linear regression.
- The fitted model is less reliable when it is built on a small sample size.
- Polynomial models have poor interpolator properties. High degree polynomials are notorious for oscillations between exact-fit values.
- Polynomial models have poor extrapolatory properties. Polynomials may provide good fits within the range of data, but they will frequently deteriorate rapidly outside the range of the data.
- Polynomial models have poor asymptotic properties. By their nature, polynomials have a finite response for finite x values and have an infinite response if and only if the x value is infinite. Thus, polynomials may not model asymptotic phenomena very well.
- Polynomial models have a shape/degree trade-off. In order to model data with a complicated structure, the degree of the model must be high, indicating and the associated number of parameters to be estimated will also be high. This can result in highly unstable models.

Random Forest Regression:

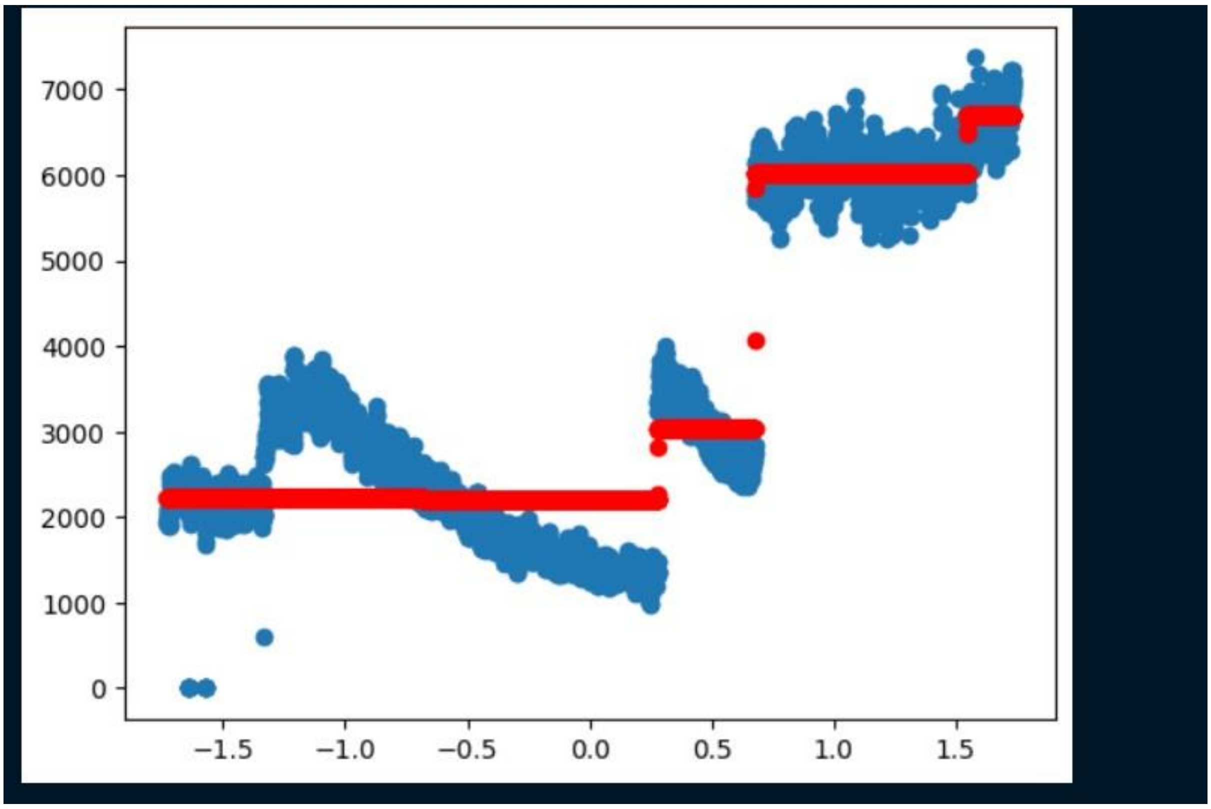
```
#Note that data cleaning and variable allocation was done just as before

import numpy as np

import matplotlib.pyplot as plt

import seaborn as seabornInstance

from sklearn.linear_model import LinearRegression

from sklearn import metrics

%matplotlib inline

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X_train = sc.fit_transform(X_train)

X_test = sc.transform(X_test)

from sklearn.ensemble import RandomForestRegressor

regr = RandomForestRegressor(max_depth=2, random_state=0, n_estimators=100)

regr.fit(X_train, y1_train)

y1_pred = regr.predict(X_test)


from sklearn.metrics import mean_squared_error as mse

from sklearn.metrics import mean_absolute_error as mae

from sklearn.metrics import r2_score
```

```
print('MSE =', mse(y1_pred, y1_test))

print('MAE/Accuracy =', mae(y1_pred, y1_test))

print('Accuracy =', r2_score(y1_pred, y1_test))

print('Accuracy in percentage =', r2_score(y1_pred, y1_test)*100)
```

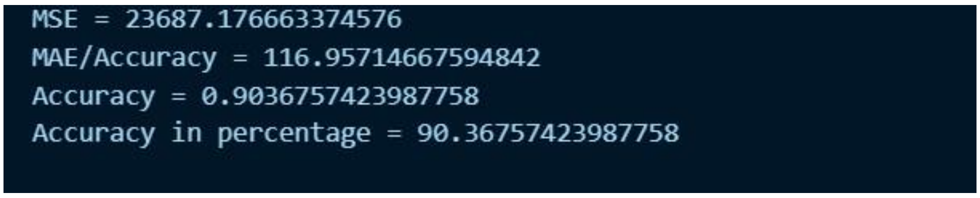


```
MSE = 297199.0327427266
MAE/Accuracy = 414.4947890192679
Accuracy = 0.9055755240062295
Accuracy in percentage = 90.55755240062295
```

```
plt.scatter(X_test[:,0],y1_test)

plt.scatter(X_test[:,0],y1_pred,c="red")

plt.show()
```

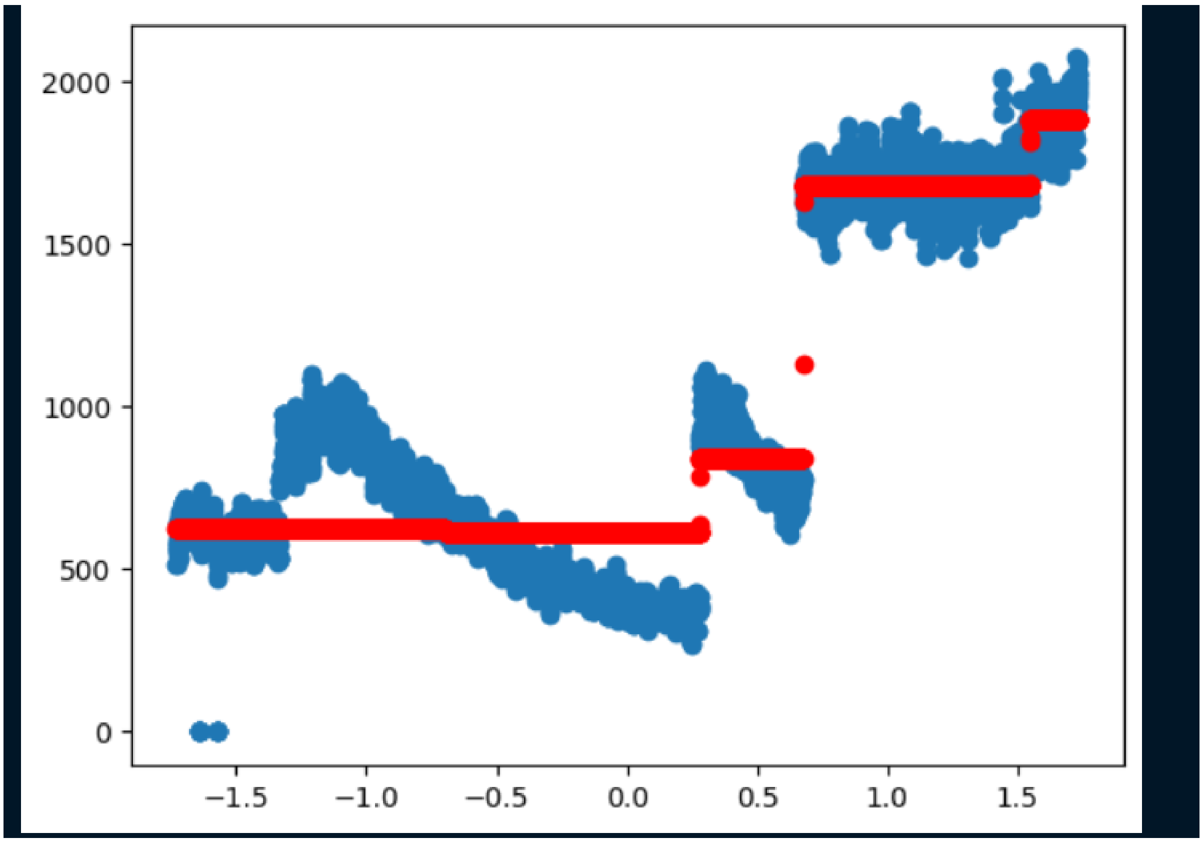


```
regr = RandomForestRegressor(max_depth=2, random_state=0, n_estimators=100)

regr.fit(X_train, y2_train)

y2_pred = regr.predict(X_test)


print('MSE =', mse(y2_pred, y2_test))

print('MAE/Accuracy =', mae(y2_pred, y2_test))

print('Accuracy =', r2_score(y2_pred, y2_test))

print('Accuracy in percentage =', r2_score(y2_pred, y2_test)*100)
```

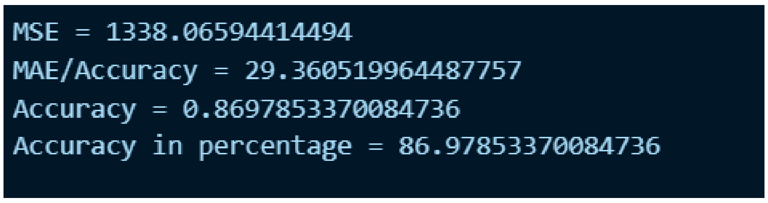


```
MSE = 23687.176663374576
MAE/Accuracy = 116.95714667594842
Accuracy = 0.9036757423987758
Accuracy in percentage = 90.36757423987758
```

```
plt.scatter(X_test[:,0],y2_test)
plt.scatter(X_test[:,0],y2_pred,c="red")
plt.show()
```

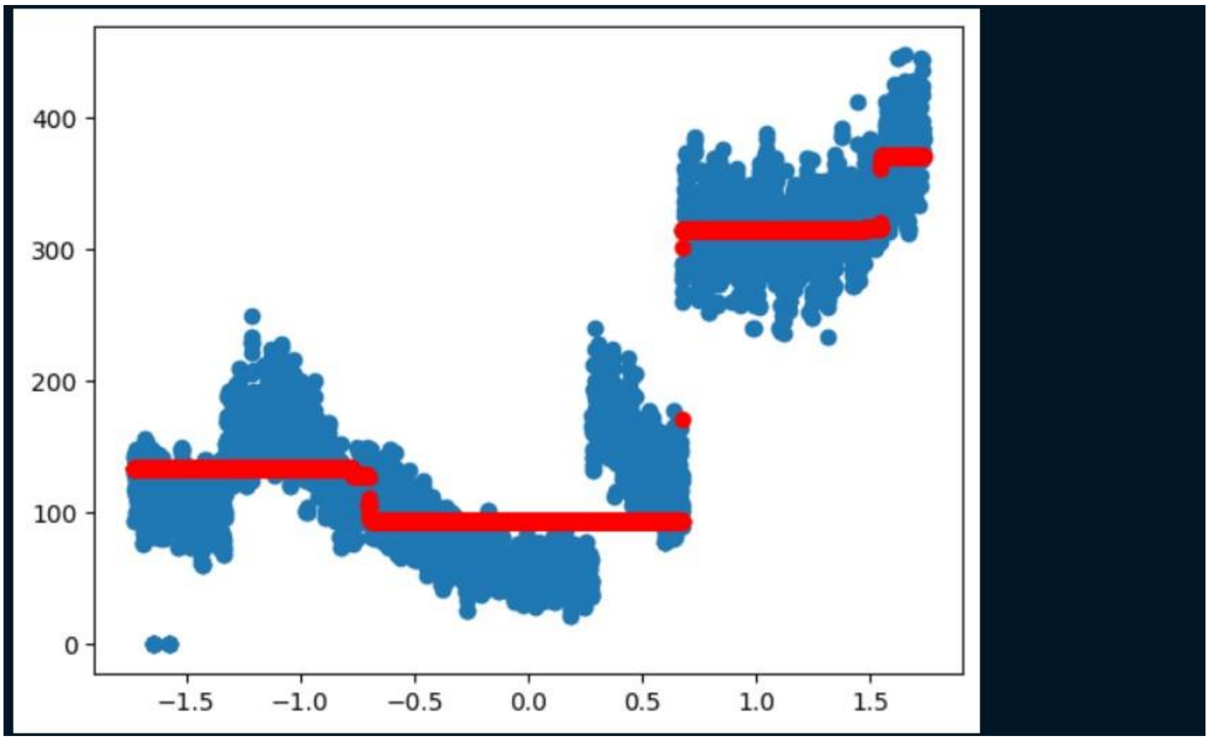


```
regr = RandomForestRegressor(max_depth=2, random_state=0, n_estimators=100)
regr.fit(X_train, y3_train)
y3_pred = regr.predict(X_test)
print('MSE =', mse(y3_pred, y3_test))
print('MAE/Accuracy =', mae(y3_pred, y3_test))
print('Accuracy =', r2_score(y3_pred, y3_test))
print('Accuracy in percentage =', r2_score(y3_pred, y3_test)*100)
```

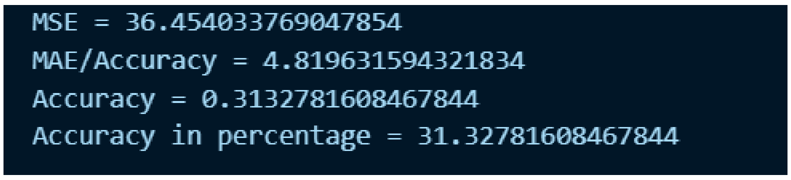


```
MSE = 1338.06594414494
MAE/Accuracy = 29.360519964487757
Accuracy = 0.8697853370084736
Accuracy in percentage = 86.97853370084736
```

```
plt.scatter(X_test[:,0],y3_test)

plt.scatter(X_test[:,0],y3_pred,c="red")

plt.show()
```

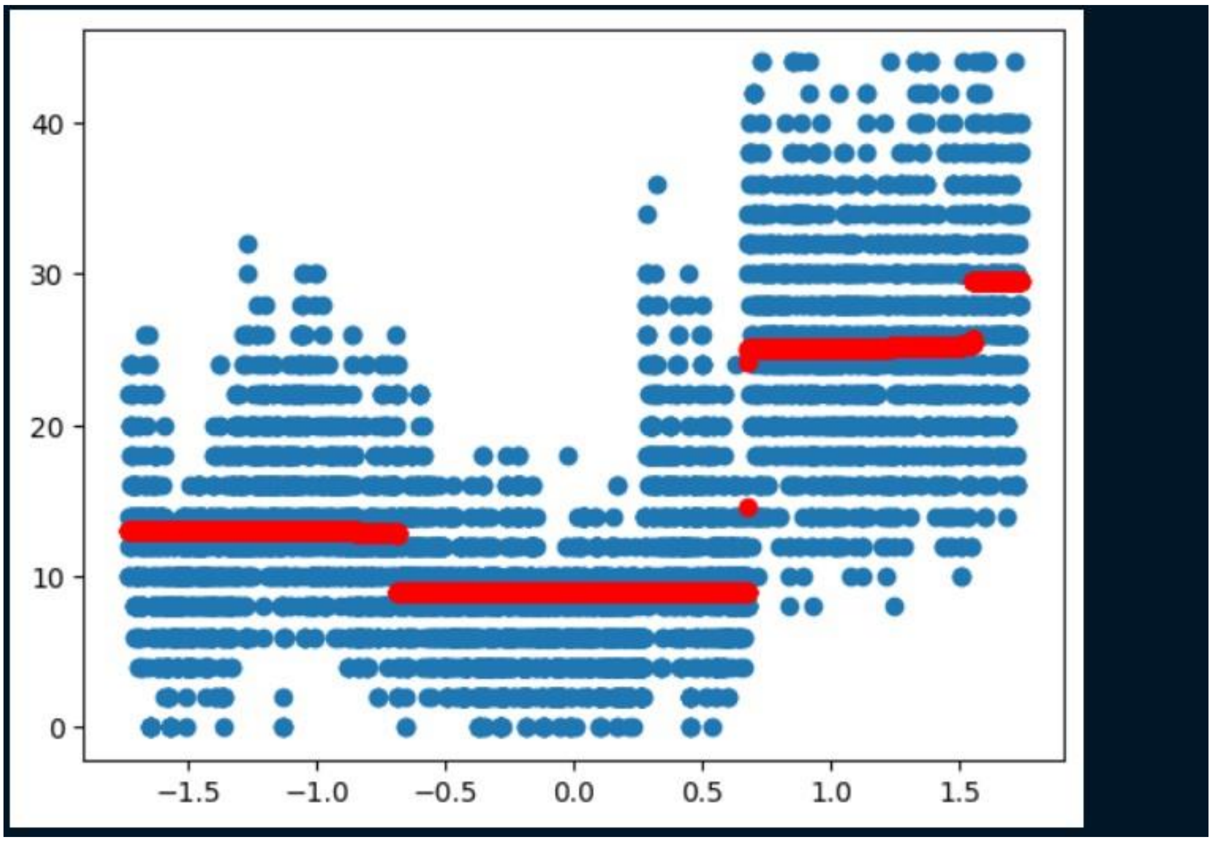


```
regr = RandomForestRegressor(max_depth=2, random_state=0, n_estimators=100)

regr.fit(X_train, y4_train)

y4_pred = regr.predict(X_test)

print('MSE =', mse(y4_pred, y4_test))

print('MAE/Accuracy =', mae(y4_pred, y4_test))

print('Accuracy =', r2_score(y4_pred, y4_test))

print('Accuracy in percentage =', r2_score(y4_pred, y4_test)*100)
```



```
MSE = 36.454033769047854
MAE/Accuracy = 4.819631594321834
Accuracy = 0.3132781608467844
Accuracy in percentage = 31.32781608467844
```

```
plt.scatter(X_test[:,0],y4_test)

plt.scatter(X_test[:,0],y4_pred,c="red")

plt.show()
```



regr = RandomForestRegressor(max_depth=2, random_state=0, n_estimators=100)

regr.fit(X_train, y5_train)

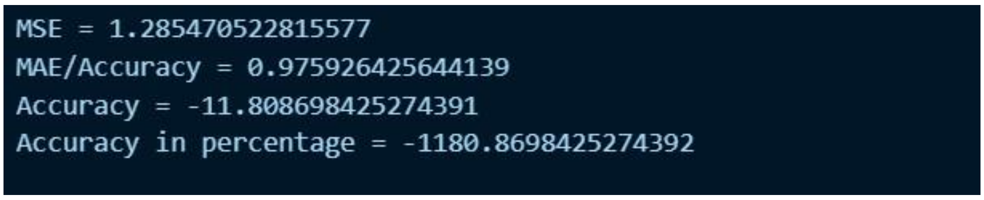
y5_pred = regr.predict(X_test)

print('MSE =', mse(y5_pred, y5_test))

print('MAE/Accuracy =', mae(y5_pred, y5_test))

print('Accuracy =', r2_score(y5_pred, y5_test))

print('Accuracy in percentage =', r2_score(y5_pred, y5_test)*100)

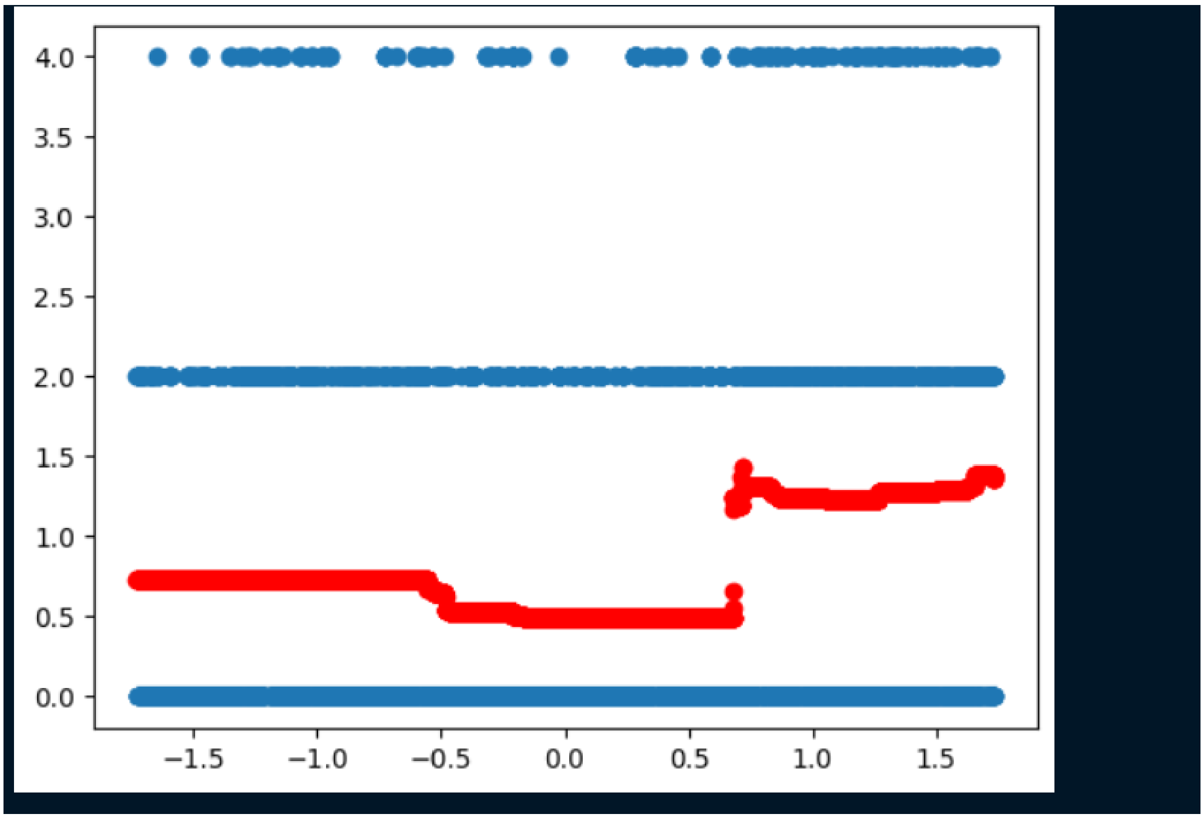


```
MSE = 1.285470522815577
MAE/Accuracy = 0.975926425644139
Accuracy = -11.808698425274391
Accuracy in percentage = -1180.8698425274392
```

plt.scatter(X_test[:,0],y5_test)

plt.scatter(X_test[:,0],y5_pred,c="red")

plt.show()



regr.fit(X_train, y6_train)

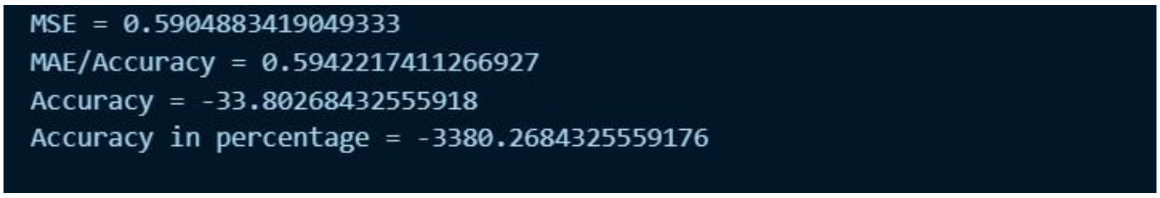
y6_pred = regr.predict(X_test)

print('MSE =', mse(y6_pred, y6_test))

print('MAE/Accuracy =', mae(y6_pred, y6_test))

print('Accuracy =', r2_score(y6_pred, y6_test))

print('Accuracy in percentage =', r2_score(y6_pred, y6_test)*100)

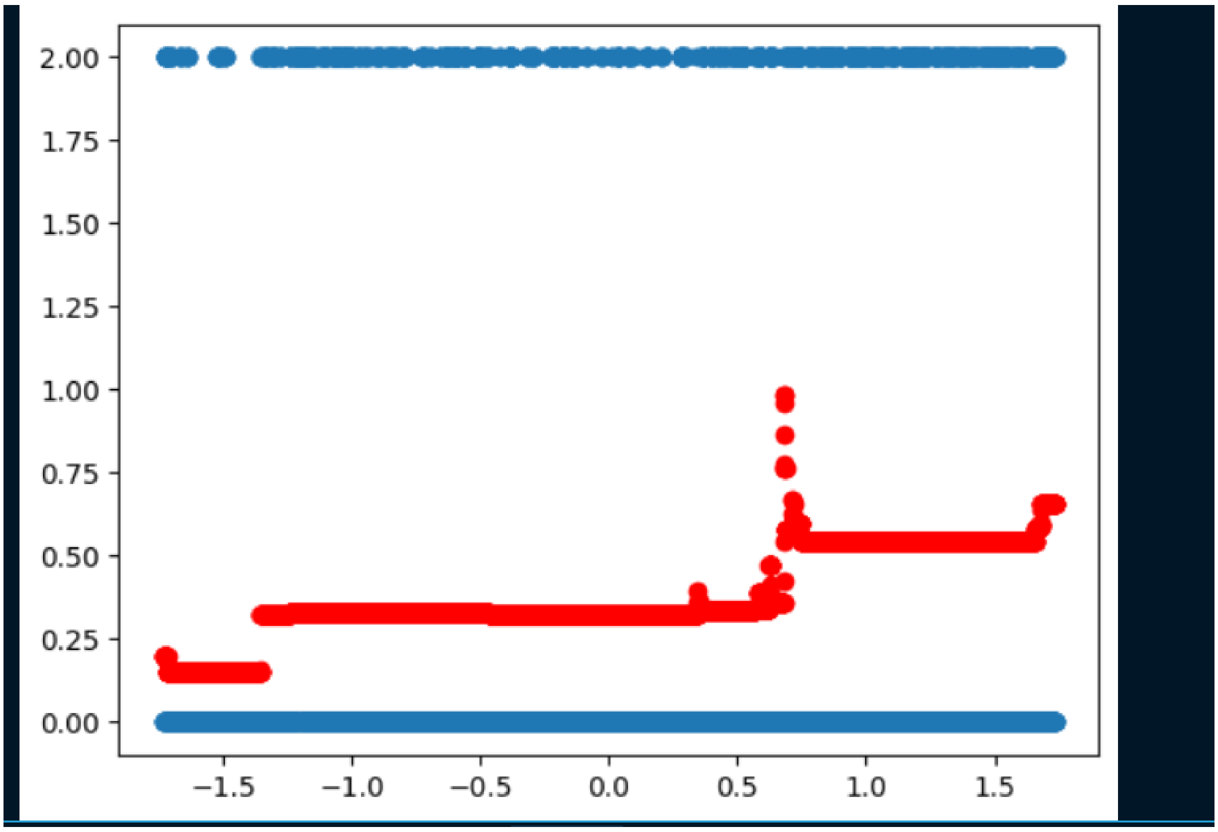


```
MSE = 0.5904883419049333
MAE/Accuracy = 0.5942217411266927
Accuracy = -33.80268432555918
Accuracy in percentage = -3380.2684325559176
```

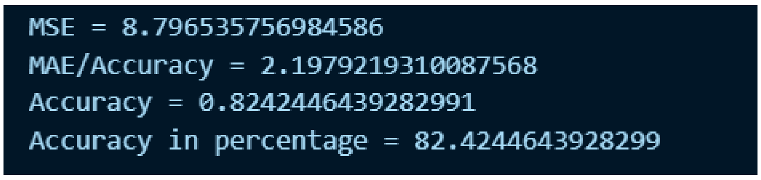
plt.scatter(X_test[:,0],y6_test)

plt.scatter(X_test[:,0],y6_pred,c="red")

plt.show()

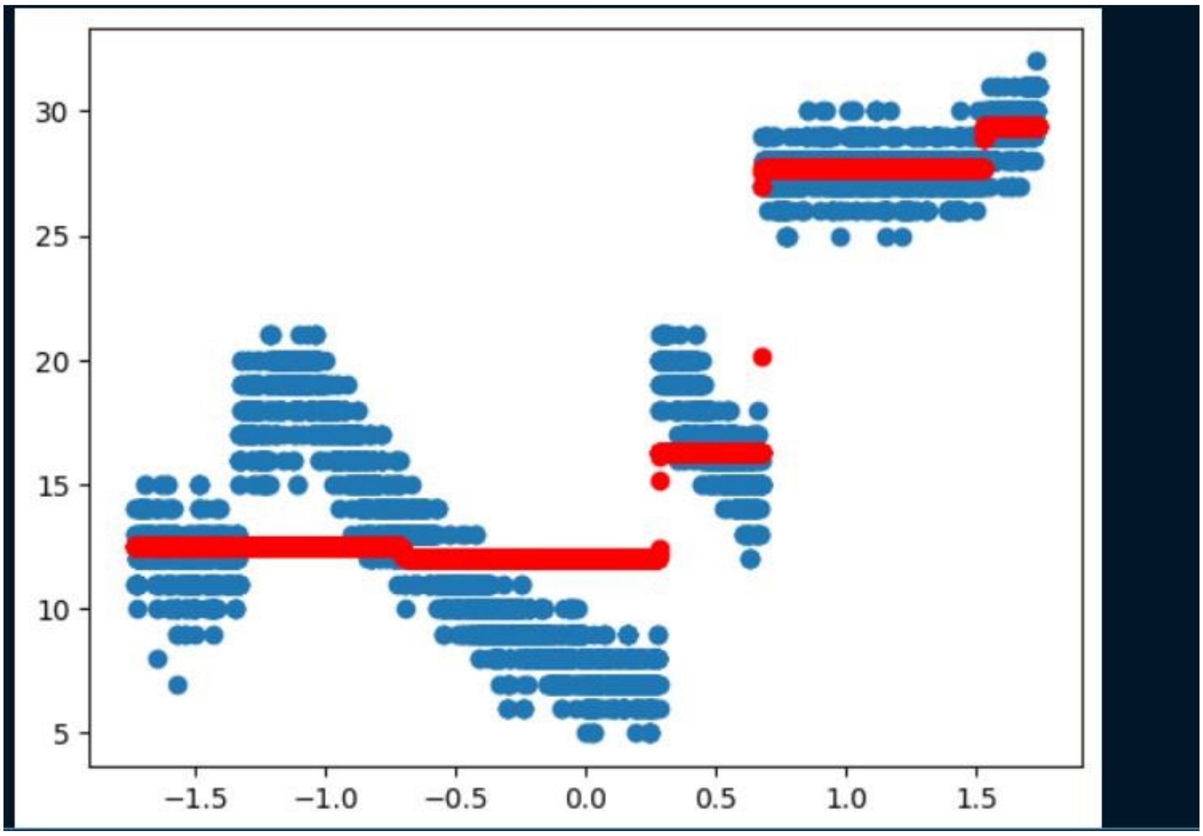


```
regr = RandomForestRegressor(max_depth=2, random_state=0, n_estimators=100)
regr.fit(X_train, y7_train)
y7_pred = regr.predict(X_test)
print('MSE =', mse(y7_pred, y7_test))
print('MAE/Accuracy =', mae(y7_pred, y7_test))
print('Accuracy =', r2_score(y7_pred, y7_test))
print('Accuracy in percentage =', r2_score(y7_pred, y7_test)*100)
```



```
MSE = 8.796535756984586
MAE/Accuracy = 2.1979219310087568
Accuracy = 0.8242446439282991
Accuracy in percentage = 82.4244643928299
```

```
plt.scatter(X_test[:,0],y7_test)
plt.scatter(X_test[:,0],y7_pred,c="red")
plt.show()
```



regr = RandomForestRegressor(max_depth=2, random_state=0, n_estimators=100)

regr.fit(X_train, y8_train)

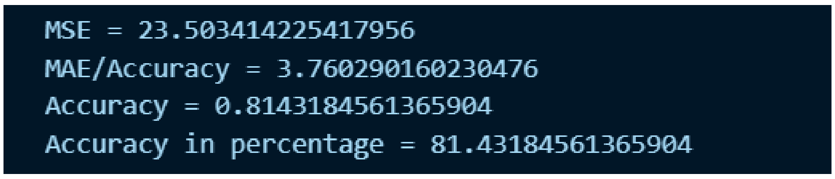
y8_pred = regr.predict(X_test)

print('MSE =', mse(y8_pred, y8_test))

print('MAE/Accuracy =', mae(y8_pred, y8_test))

print('Accuracy =', r2_score(y8_pred, y8_test))

print('Accuracy in percentage =', r2_score(y8_pred, y8_test)*100)

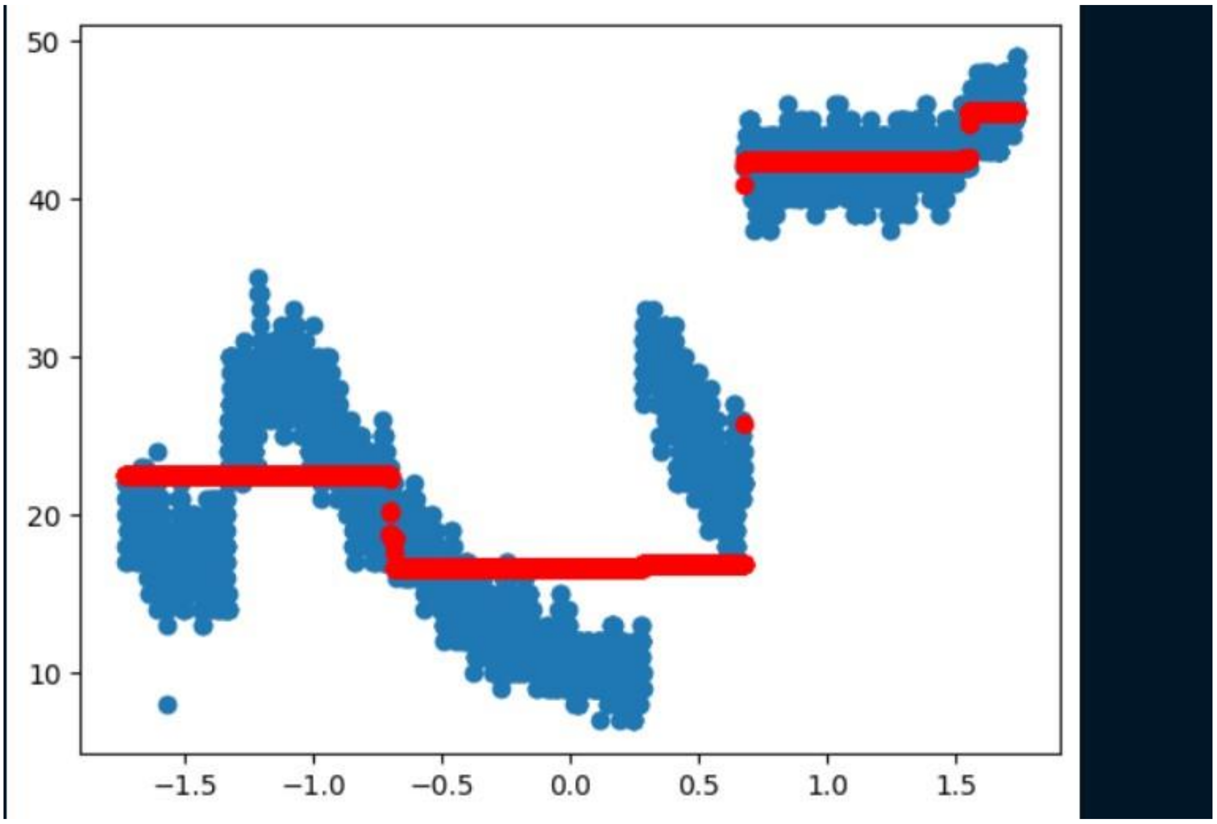


```
MSE = 23.503414225417956
MAE/Accuracy = 3.760290160230476
Accuracy = 0.8143184561365904
Accuracy in percentage = 81.43184561365904
```

plt.scatter(X_test[:,0],y8_test)

plt.scatter(X_test[:,0],y8_pred,c="red")

plt.show()



regr = RandomForestRegressor(max_depth=2, random_state=0, n_estimators=100)

regr.fit(X_train, y9_train)

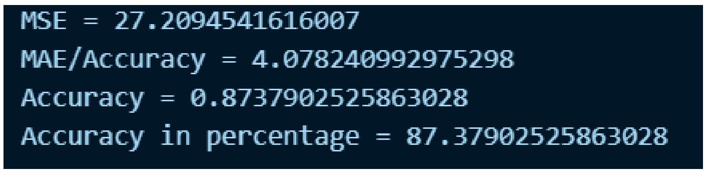
y9_pred = regr.predict(X_test)

print('MSE =', mse(y9_pred, y9_test))

print('MAE/Accuracy =', mae(y9_pred, y9_test))

print('Accuracy =', r2_score(y9_pred, y9_test))

print('Accuracy in percentage =', r2_score(y9_pred, y9_test)*100)

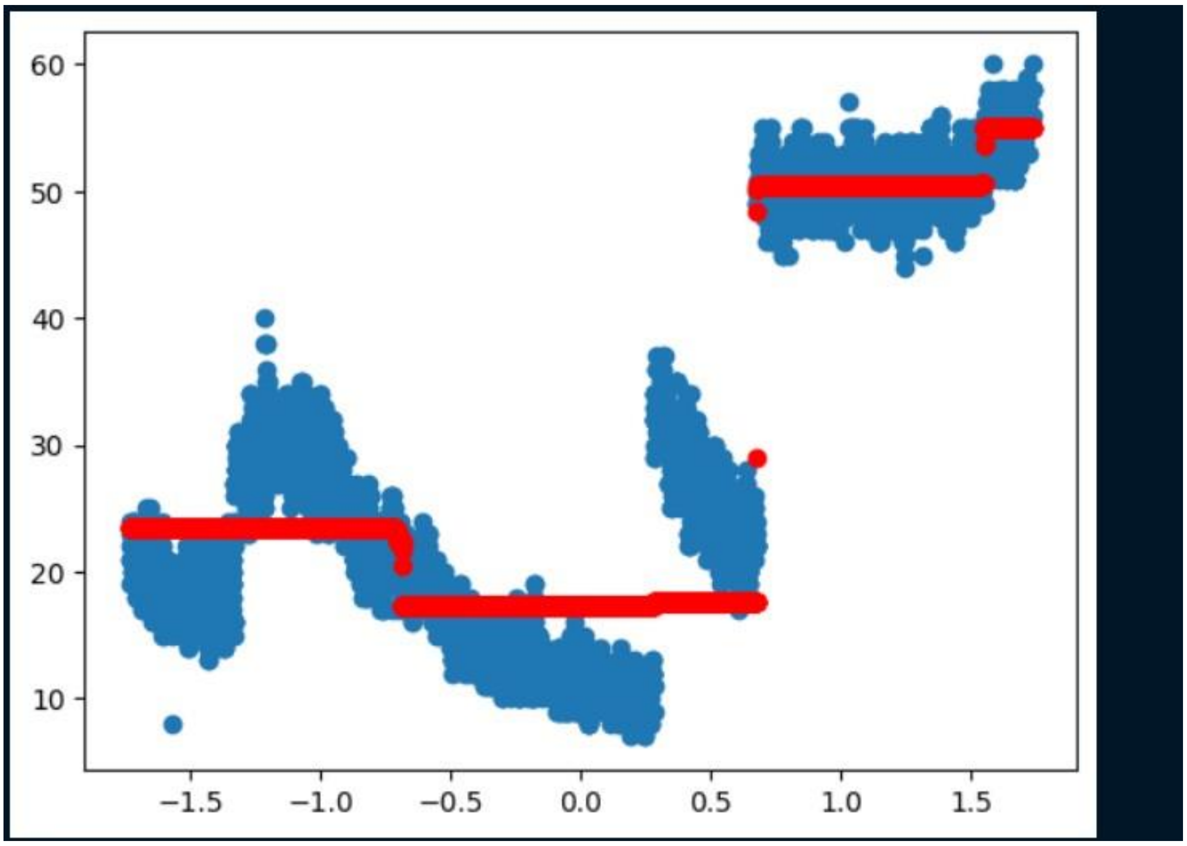


```
MSE = 27.2094541616007
MAE/Accuracy = 4.078240992975298
Accuracy = 0.8737902525863028
Accuracy in percentage = 87.37902525863028
```

plt.scatter(X_test[:,0],y9_test)

plt.scatter(X_test[:,0],y9_pred,c="red")

plt.show()



Limitations of Random Forest Regression:

- Many trees are required to produce accurate results.
- Using more trees slows down the model.
- Overfitting may occur if too many trees are used.
- Does not perform well if data is unbalanced (seen in 5 and 10um data)
- Not suitable for predictions outside the data range.

K mean Clustering

import pandas as pd

import numpy as np

df = pd.read_csv("C:/Users/Indu

Bhushan/OneDrive/Desktop/IoT/Project/test2.csv")df

#Dropping NaN values

df=df.dropna()

df.head()

#Dropping outliers rows of each column

micro1_mean=df["0.3um"].mean()

micro1_std=df["0.3um"].std()

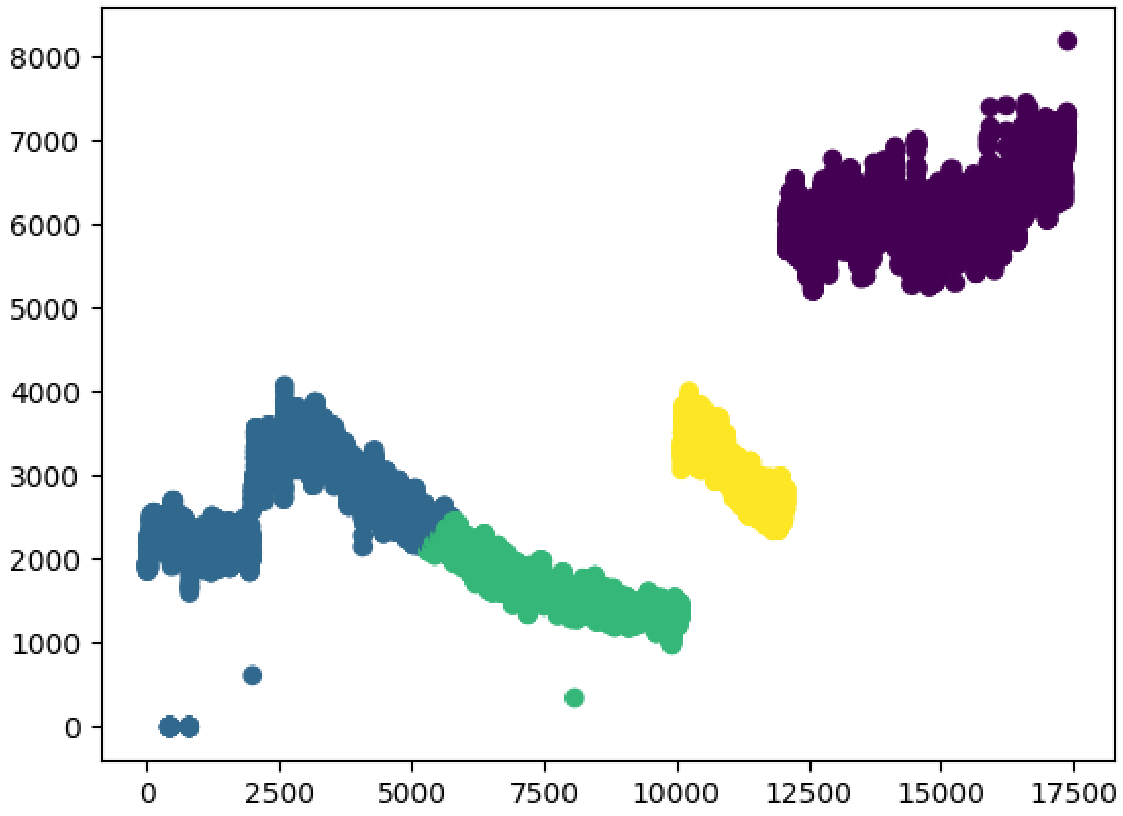
micro2_mean=df["0.5um"].mean()

```
micro2_std=df["0.5um"].std()

micro3_mean=df["1um"].mean()

micro3_std=df["1um"].std()

micro4_mean=df["2.5um"].mean()

micro4_std=df["2.5um"].std()

micro5_mean=df["5um"].mean()

micro5_std=df["5um"].std()

micro6_mean=df["10um"].mean()

micro6_std=df["10um"].std()

pm1_mean=df["PM1.0"].mean()

pm1_std=df["PM1.0"].std()

pm2_mean=df["PM2.5"].mean()

pm2_std=df["PM2.5"].std()

pm3_mean=df["PM10"].mean()

pm3_std=df["PM10"].std()

df_cleaned=df[(df["0.3um"]>=(micro1_mean-
3*micro1_std))&(df["0.3um"]<=(micro1_mean+3*micro1_std))]

df_cleaned=df_cleaned[(df_cleaned["0.5um"]>=(micro2_mean-
3*micro2_std))&(df_cleaned["0.5um"]<=(micro2_mean+3*micro2_std))]

df_cleaned=df_cleaned[(df_cleaned["1um"]>=(micro3_mean-
3*micro3_std))&(df_cleaned["1um"]<=(micro3_mean+3*micro3_std))]

df_cleaned=df_cleaned[(df_cleaned["2.5um"]>=(micro4_mean-
3*micro4_std))&(df_cleaned["2.5um"]<=(micro4_mean+3*micro4_std))]

df_cleaned=df_cleaned[(df_cleaned["5um"]>=(micro5_mean-
3*micro5_std))&(df_cleaned["5um"]<=(micro5_mean+3*micro5_std))]

df_cleaned=df_cleaned[(df_cleaned["10um"]>=(micro6_mean-
3*micro6_std))&(df_cleaned["10um"]<=(micro6_mean+3*micro6_std))]

df_cleaned=df_cleaned[(df_cleaned["PM1.0"]>=(pm1_mean-
3*pm1_std))&(df_cleaned["PM1.0"]<=(pm1_mean+3*pm1_std))]

df_cleaned=df_cleaned[(df_cleaned["PM2.5"]>=(pm2_mean-
3*pm2_std))&(df_cleaned["PM2.5"]<=(pm2_mean+3*pm2_std))]

df_cleaned=df_cleaned[(df_cleaned["PM10"]>=(pm3_mean-
3*pm3_std))&(df_cleaned["PM10"]<=(pm3_mean+3*pm3_std))]

df_cleaned.head()

df1=df_cleaned.copy()
```

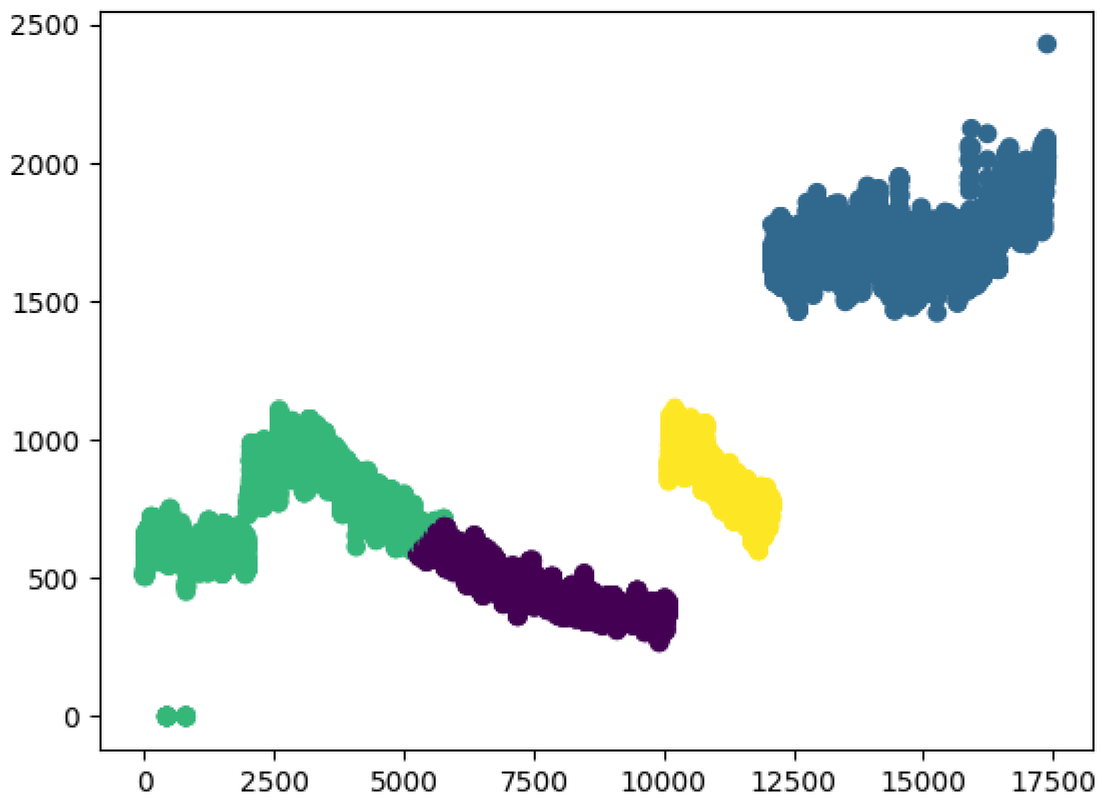
```
df1['Date Index']=np.arange(len(df1.index))
df1.head()
# Training data
X1 = df1[['Date Index','0.3um']]
X2 = df1[['Date Index','0.5um']]
X3 = df1[['Date Index','1um']]
X4 = df1[['Date Index','2.5um']]
X5 = df1[['Date Index','5um']]
X6 = df1[['Date Index','10um']]
X7 = df1[['Date Index','PM1.0']]
X8 = df1[['Date Index','PM2.5']]
X9 = df1[['Date Index','PM10']]
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X1)
scaled_data1 = scaler.transform(X1)
scaled_data
scaler.fit(X2)
scaled_data2 = scaler.transform(X2)
scaled_data2
scaler.fit(X3)
scaled_data3 = scaler.transform(X3)
scaled_data3
scaler.fit(X4)
scaled_data4 = scaler.transform(X4)
scaled_data4
scaler.fit(X5)
scaled_data5 = scaler.transform(X5)
scaled_data5
scaler.fit(X6)
scaled_data6 = scaler.transform(X6)
scaled_data6
```

```
scaler.fit(X7)
scaled_data7 = scaler.transform(X7)
scaled_data7
scaler.fit(X8)
scaled_data8 = scaler.transform(X8)
scaled_data8
scaler.fit(X9)
scaled_data9 = scaler.transform(X9)
scaled_data9
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
kmeans1 = KMeans(n_clusters=4, random_state=0,init='k-means++',n_init=10)
kmeans1.fit(scaled_data1)
df1["clusters_0.3"]=kmeans1.labels_
df1.head()
kmeans2 = KMeans(n_clusters=4, random_state=0,init='k-means++',n_init=10)
kmeans2.fit(scaled_data2)
df1["clusters_0.5"]=kmeans2.labels_
df1.head()
kmeans3 = KMeans(n_clusters=4, random_state=0,init='k-means++',n_init=10)
kmeans3.fit(scaled_data3)
df1["clusters_1"]=kmeans3.labels_
df1.head()
kmeans4 = KMeans(n_clusters=4, random_state=0,init='k-means++',n_init=10)
kmeans4.fit(scaled_data4)
df1["clusters_2.5"]=kmeans4.labels_
df1.head()
kmeans5 = KMeans(n_clusters=3, random_state=0,init='k-means++',n_init=10)
kmeans5.fit(scaled_data5)
df1["clusters_5"]=kmeans5.labels_
df1.head()
kmeans6 = KMeans(n_clusters=2, random_state=0,init='k-means++',n_init=10)
```

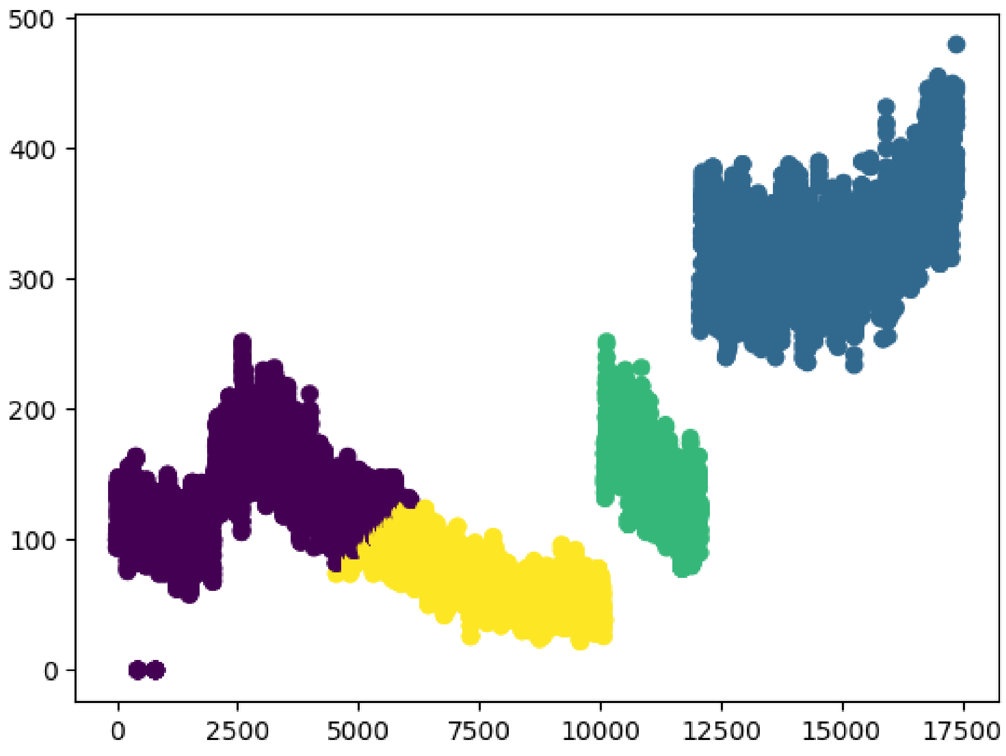
```
kmeans6.fit(scaled_data6)
df1["clusters_10"]=kmeans6.labels_
df1.head()
kmeans7 = KMeans(n_clusters=4, random_state=0,init='k-means++',n_init=10)
kmeans7.fit(scaled_data7)
df1["clusters_PM1.0"]=kmeans7.labels_
df1.head()
kmeans8 = KMeans(n_clusters=4, random_state=0,init='k-means++',n_init=10)
kmeans8.fit(scaled_data8)
df1["clusters_PM2.5"]=kmeans8.labels_
df1.head()
kmeans9 = KMeans(n_clusters=4, random_state=0,init='k-means++',n_init=10)
kmeans9.fit(scaled_data9)
df1["clusters_PM10"]=kmeans9.labels_
df1.head()
plt.scatter(df1['Date Index'],df1['0.3um'],c=df1["clusters_0.3"])
plt.show()
```



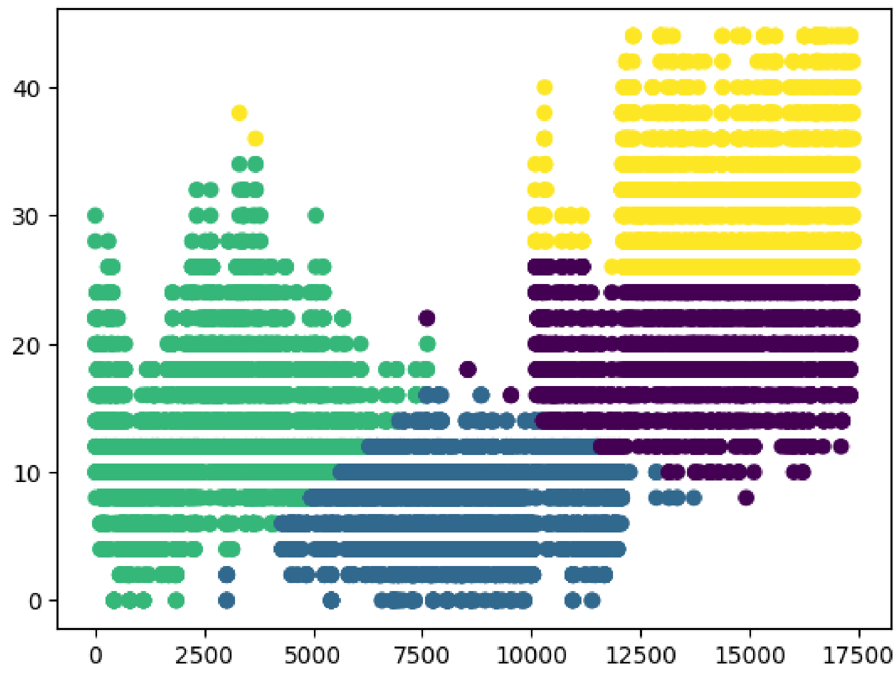
```
plt.scatter(df1['Date Index'],df1['0.5um'],c=df1["clusters_0.5"])
plt.show()
```



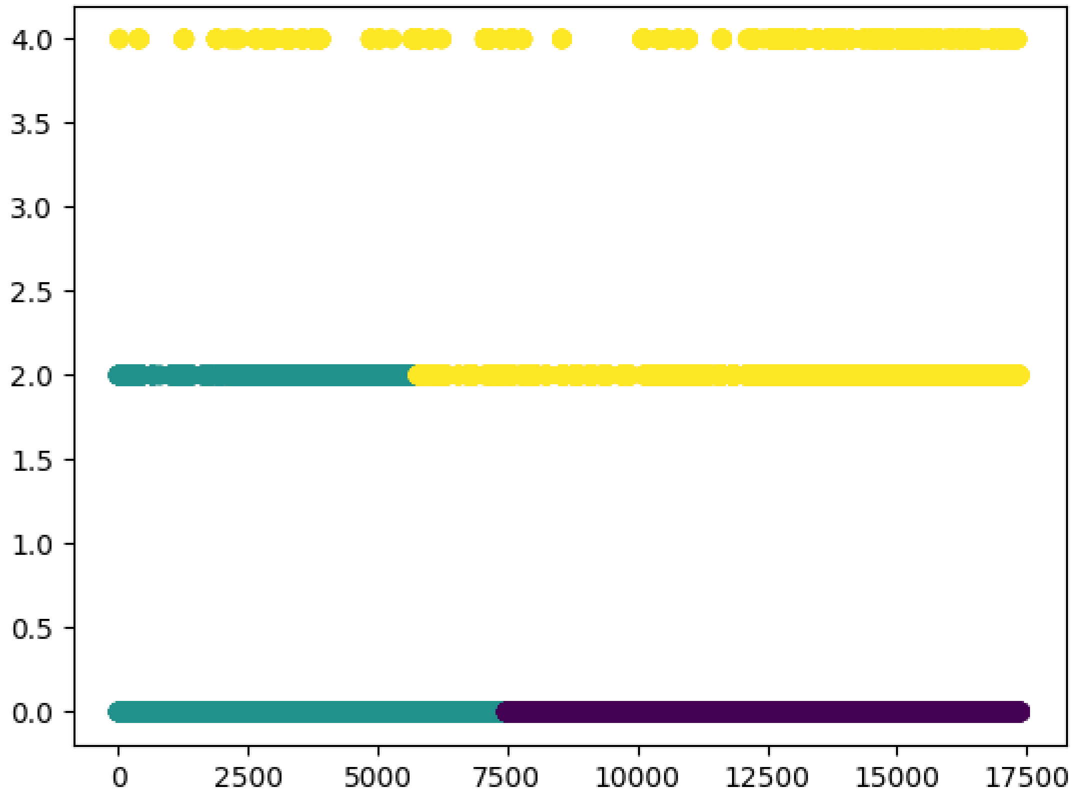
```
plt.scatter(df1['Date Index'],df1['1um'],c=df1["clusters_1"])
plt.show()
```



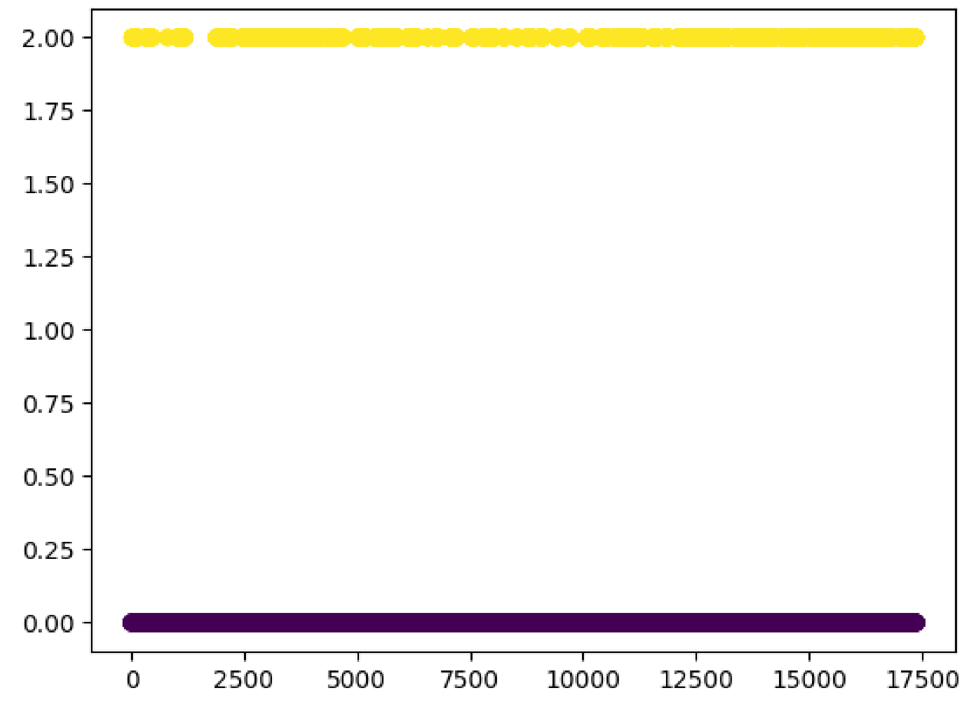
```
plt.scatter(df1['Date Index'],df1['2.5um'],c=df1["clusters_2.5"])
plt.show()
```



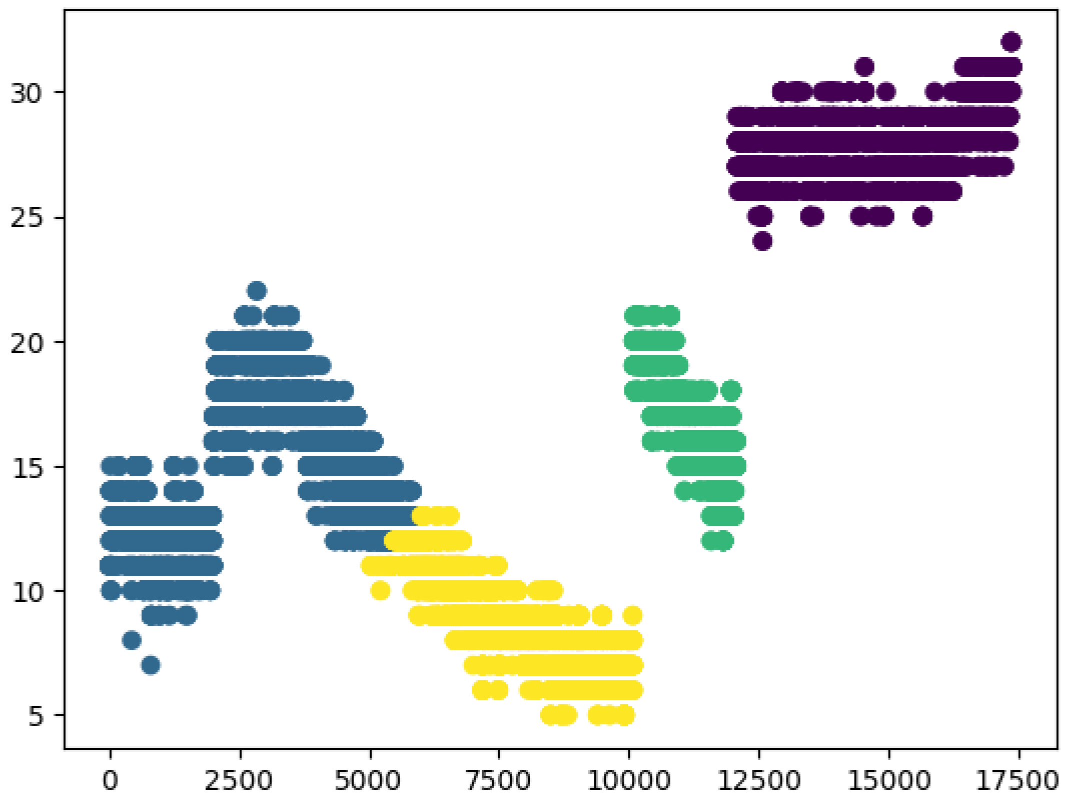
```
plt.scatter(df1['Date Index'],df1['5um'],c=df1["clusters_5"])
plt.show()
```



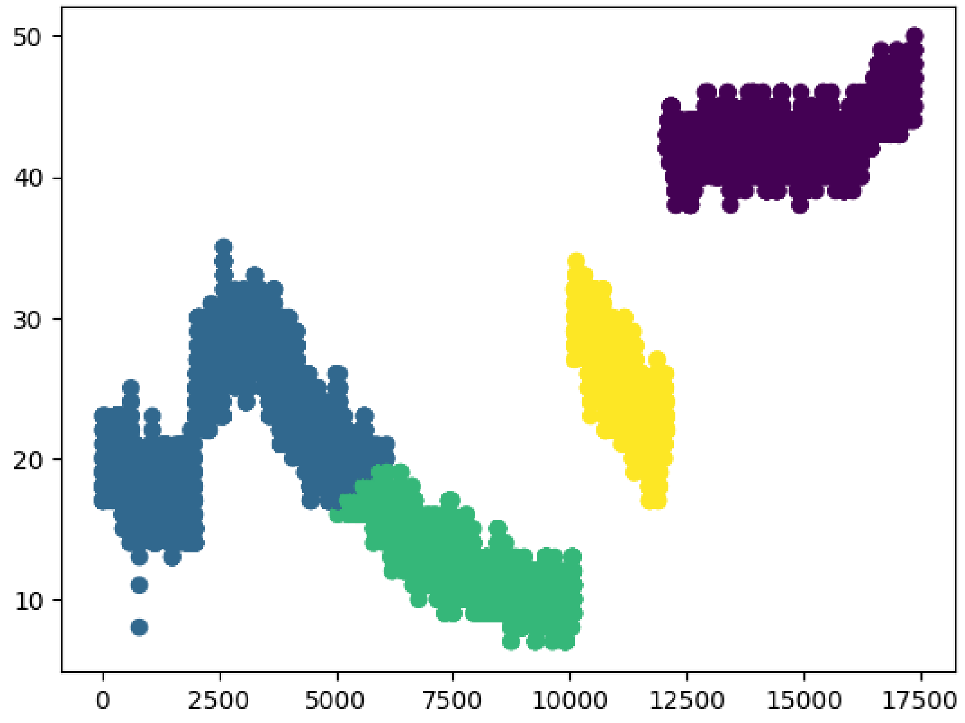
```
plt.scatter(df1['Date Index'],df1['10um'],c=df1["clusters_10"])
plt.show()
```



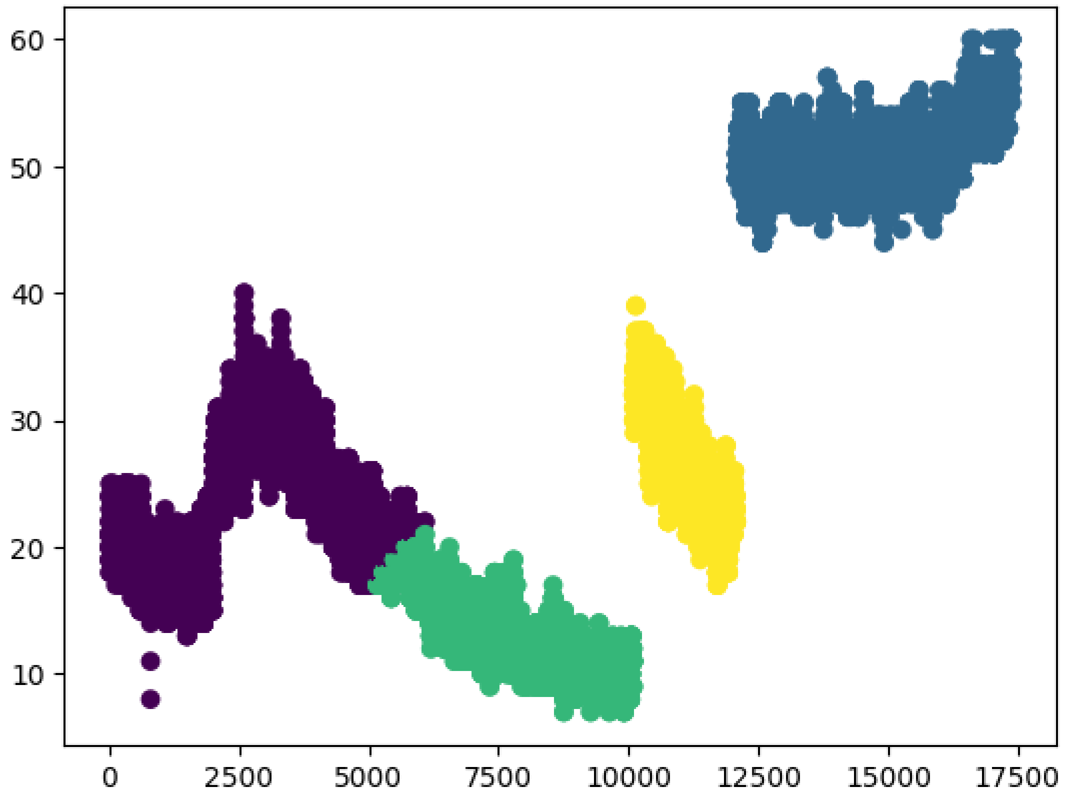
```
plt.scatter(df1['Date Index'],df1['PM1.0'],c=df1["clusters_PM1.0"])
plt.show()
```



```
plt.scatter(df1['Date Index'],df1['PM2.5'],c=df1["clusters_PM2.5"])
plt.show()
```



```
plt.scatter(df1['Date Index'],df1['PM10'],c=df1["clusters_PM10"])
plt.show()
```



Limitations of the Project:

- Data was collected only for a short while (around 30 days). Thus, it is not indicative of the trends in dust levels over the whole year or even during a particular season.
- The dependent variable used is an index value. Thus, it is not possible to enter a specific date to predict a parameter level for that day but rather, a range of indices (around 500) must be found for that day.
- The highest accuracy obtained is around 86%. This is again because not enough data was collected.
- The data collected is only indicative of a particular area in the VIT Vellore campus. Dust levels can vary elsewhere.
- While the models can predict a single value, the algorithm used cannot classify the different parameters into high, low, or medium levels. Thus, it is difficult to effectively utilize for general uses such as to suggest whether one should wear a mask on a particular day etc.

Conclusion:

The suggested method of applying machine learning algorithms for predictive analysis of dust levels offers a substantial advancement over conventional techniques for monitoring and forecasting dust levels. Traditional approaches sometimes rely on manual monitoring, which is frequently expensive and time-consuming and may not give consumers real-time warnings about excessive dust levels in the area.

In contrast, real-time monitoring and prediction of dust levels may be done more cheaply and effectively by using machine learning techniques. The data may be pre-processed, separated into training and testing datasets, and utilised to create a model that can forecast future dust levels by using data collected using a dust sensor and stored in Excel sheets. This method may be used to notify users in real-time when there are excessive amounts of dust in the air, allowing them to take preventative action to lower the risk of respiratory issues and other health risks brought on by high dust levels.

Using machine learning techniques to anticipate dust levels has the potential to completely change the way dust monitoring is done. This strategy can enable more frequent and thorough monitoring of dust levels in locations with high dust concentrations, such as construction sites, mines, and factories, by offering a cost-effective and efficient method of monitoring and forecasting dust levels. This can aid in lowering the likelihood of respiratory issues and other health risks brought on by excessive environmental dust levels.

Furthermore, this method's ramifications go beyond only monitoring and forecasting dust. This method can open the door for the development of comparable applications in other sectors including air pollution monitoring, water quality monitoring, and climate prediction by showcasing the potential of machine learning algorithms for predictive analysis of environmental parameters.

Overall, the proposed method of using machine learning algorithms for predictive analysis of dust levels represents a significant advancement in dust monitoring techniques, with implications for the health and safety of people working in high-dust environments as well as for the creation of related applications in other fields.

References:

[1] Xiong, Ruoxin & Tang, Pingbo. (2021). Machine learning using synthetic images for detecting dust emissions on construction sites. Smart and Sustainable Built Environment. ahead-of-print. 10.1108/SASBE-04-2021-0066.

[2] Machine learning holography for measuring 3D particle distribution by Siyao Shao

[3] Machine learning shadowgraph for particle size and shape characterization by Jiaqi Li

[4] A Survey paper on Vehicles Emitting Air Quality and Prevention of Air Pollution by using IoT Along with Machine Learning Approaches by M.Dhanalakshmi

[5] Méndez, M., Merayo, M.G. & Núñez, M. Machine learning algorithms to forecast air quality: a survey. Artif Intell Rev (2023). https://doi.org/10.1007/s10462-023-10424-4