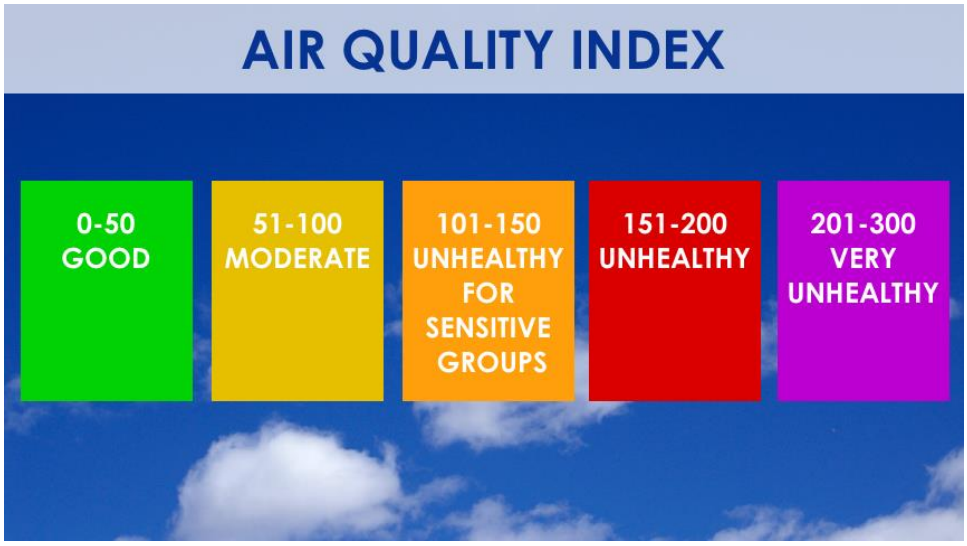
Air Quality in Italy
Predicting the PPM of Indium Oxide
Machine Learning Techniques for Regression

By: Denis O'Byrne

Date: 4/4/2022

IBM Supervised Machine Learning: Regression Final Project



Purpose and Associated Jupyter Notebook

This report is part of my completion of the IBM Machine Learning Professional Certificate. All work related to this Certificate Can be found on my GitHub Linked here:

< https://github.com/DenisOByrne/IBM-Machine-Learning-Professional-Certificate >

This project is specifically for the IBM Supervised Machine Learning Course. The folder containing the dataset, this report and the Jupyter notebook used to complete this assignment can be found at the link below:

< https://github.com/DenisOByrne/IBM-Machine-Learning-Professional-Certificate/tree/main/Supervised-Machine-Learning-Regression>

Introduction

Air pollution is a problem in all industrialized nations, negatively impacting the planet and environment which has become a major concern for global governing bodies around the world. The majority of issues revolve around the by-products of combustion from fossil fuels, which release many harmful chemicals into the air. In 2004 the Italian government funded a 1-year study by the ENEA (National Agency for New Technologies, Energy and Sustainable Economic Development) to record data on pollutants and other components of the air and weather conditions of a city (the name of the city is not publicly available). Data were recorded from March 2004 to February 2005 representing the longest freely available recordings of on field deployed air quality chemical sensor devices responses. With this data scientists were able to study seasonal trends in pollution which had yet to be properly documented. Thanks to this study, the scale of the consequences of fossil fuel emissions was documented for the world to see leading to many similar studies in densely populated areas in other countries.

With this dataset it would be ideal to be able to build a model to predict the concentration of pollutants (in ppm) of the air to see if we can measure the air quality and predict dates which may have poor air quality. With such information the government could help inform the public on which days and times may be best to stay indoors for people with respiratory problems. For the purposes of this study, I will be using regression to build a model which predicts the concentration of Indium oxide in the air.

The Data Set

Data for this study comes from the UCI Machine Learning Repository, specifically we are working with the Air Quality Data Set linked here. As previously stated, the source of the data is the study in 2004-2005 by the ENEA provided to the UCI by Saverio De Vito (saverio.devito@enea.it). The dataset contains 9358 instances of hourly averaged responses from an array of 5 metal oxide chemical sensors embedded in an Air Quality Chemical Multisensor Device. The device was located on the field in a significantly polluted area, at road level, within

an Italian city. Data were recorded from March 2004 to February 2005 (one year) representing the longest freely available recordings of on field deployed air quality chemical sensor devices responses. Ground Truth hourly averaged concentrations for CO, Non Metanic Hydrocarbons, Benzene, Total Nitrogen Oxides (NOx) and Nitrogen Dioxide (NO2) and were provided by a co-located reference certified analyzer. Evidences of cross-sensitivities as well as both concept and sensor drifts are present as described in De Vito et al., Sens. And Act. B, Vol. 129,2,2008 (citation required) eventually affecting sensors concentration estimation capabilities. Missing values are tagged with -200 value.

Target:

PT08.S5 - continuous - (indium oxide) hourly averaged sensor response (nominally O3 targeted)

Features:

1. Date (DD/MM/YYYY)
2. Time (HH.MM.SS)
3. CO(GT) - Continuous - True hourly averaged concentration CO in mg/m^3 (reference analyzer)
4. PT08.S1 - continuous - (tin oxide) hourly averaged sensor response (nominally CO targeted)
5. NMHC(GT) - continuous - True hourly averaged overall Non Metanic HydroCarbons concentration in microg/m^3 (reference analyzer)
6. C6H6(GT) - continuous - True hourly averaged Benzene concentration in microg/m^3 (reference analyzer)
7. PT08.S2 - continuous - (titania) hourly averaged sensor response (nominally NMHC targeted)
8. NOx(GT) - continuous - True hourly averaged NOx concentration in ppb (reference analyzer)
9. PT08.S3 - continuous - (tungsten oxide) hourly averaged sensor response (nominally NOx targeted)
10. NO2(GT) - continuous - True hourly averaged NO2 concentration in microg/m^3 (reference analyzer)
11. PT08.S4 - continuous - (tungsten oxide) hourly averaged sensor response (nominally NO2 targeted)
12. PT08.S5 - continuous - (indium oxide) hourly averaged sensor response (nominally O3 targeted)
13. T - continuous - Temperature in °C
14. RH - continuous - Relative Humidity (%)
15. AH - continuous - Absolute Humidity

Handling Missing Data and Feature Engineering:

For convenience I first converted all the –200 variables to NA values so that Python na methods would work without issues. We can observe the total number of missing values and the percentage of each variable that is missing in the table below.

:

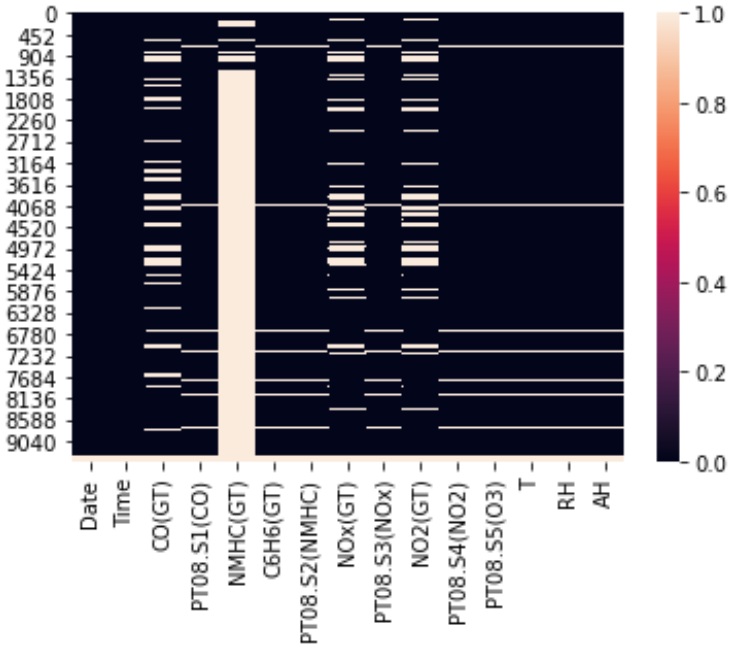
	Observations missing	Percent of Total
Date	114.0	1.203674
Time	114.0	1.203674
CO(GT)	1797.0	18.973709
PT08.S1(CO)	480.0	5.068103
NMHC(GT)	8557.0	90.349488
C6H6(GT)	480.0	5.068103
PT08.S2(NMHC)	480.0	5.068103
NOx(GT)	1753.0	18.509133
PT08.S3(NOx)	480.0	5.068103
NO2(GT)	1756.0	18.540809
PT08.S4(NO2)	480.0	5.068103
PT08.S5(O3)	480.0	5.068103
T	480.0	5.068103
RH	480.0	5.068103
AH	480.0	5.068103

From this tabular view we can spot some important information, first, we note that the NMHC(GT) variable is missing data for more than 90% of the total observations. At first, I had considered that it is possible we are just missing measurements recorded as 0s but upon further snooping of the source for the data online it turns out the device used to measure this variable was broken after about 1 month into the study, and was already faulty for the month while it was being used, therefore we need to remove this variable.

Next, we can see that many of the features and the response variable are missing data for 480 observations and the time and date variables are missing for 114 each, this is indication that certain observation times were accidentally skipped in the data. We can view this in the data by looking at a heatmap for missing values below.

```
]: sns.heatmap(data.isnull())

]: <matplotlib.axes._subplots.AxesSubplot at 0x290c9fc9ca
```



We see here that for all 480 missing values of our response variable PT08.S5(O3), we are also missing values for the other variables which are missing 480 observations. The exception to this is that for the last 114 observations, data is missing for all columns. The reason for this is that the data was stored with 114 blank extra rows, so we can remove all of these rows entirely. I chose to remove all 480 rows where the majority of the data were missing as these observations were too sparse to be useful. After removing these rows, we are left with 8991 observations. We can also see that after around the first 1000 observations all observations for NMHC(GT) are missing indicating the point where the recording device for this variable broke thus as suggested previously, we will remove this variable from the data entirely. In the end we are left with the following amount of missing data.

```
]: data.isna().sum()

]: Date                 0
   Time                 0
   CO(GT)            1647
   PT08.S1(CO)          0
   NMHC(GT)          8104
   C6H6(GT)             0
   PT08.S2(NMHC)        0
   NOx(GT)           1595
   PT08.S3(NOx)         0
   NO2(GT)           1598
   PT08.S4(NO2)         0
   PT08.S5(O3)          0
   T                    0
   RH                   0
   AH                   0
   dtype: int64
```

We are left to impute the missing data of around 1600 observations on the variables CO(GT), NOx(GT), and NO2(GT). To impute these variables, I would like to impute the mean observation for the data grouped by the month and hour of the day at which these missing observations should have occurred. We will see in a moment that these three variables are highly correlated with the time of day and month of the year.

Feature Engineering:

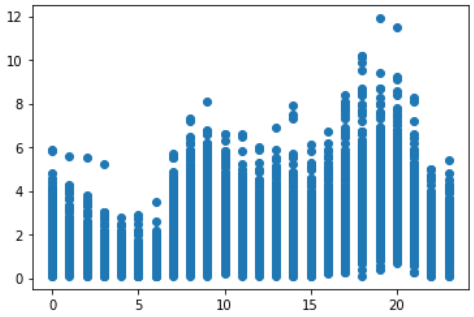
Using Python's DateTime package we can convert the separate Date and Time variables into multiple more useful variables for a machine learning model to use. I decided to add variables to indicate the day of the week, month, hour, day of the month, year, and number of days since the experiment had begun. With these new variables I retained all information from the original Date and Time variables as well as some new data. I removed the Date and Time Variables after this.

The date time package is able to pull all of these variables directly from the date times and output them in a new variable in our data frame. You can view my Jupyter notebook for this project for exact details.

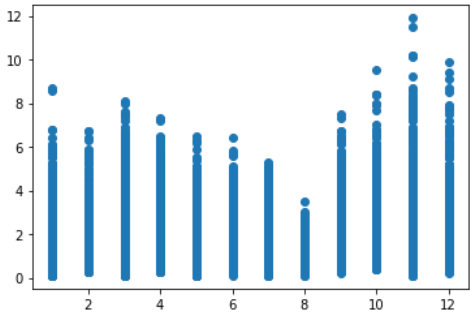
For imputing data, I first plotted the three variables with missing data based on the month and hour of the day to find these the variables are periodically related to the month and hour as seen in the plots below.

CO(GT)

```
In [45]: plt.scatter(data['Hour'], data['CO(GT)'])
Out[45]: <matplotlib.collections.PathCollection at 0x290cbc373a0>
```

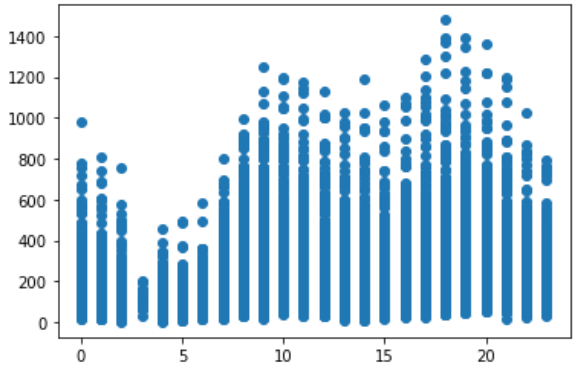



```
In [46]: plt.scatter(data['Month'], data['CO(GT)'])
Out[46]: <matplotlib.collections.PathCollection at 0x290cbcdbcd0>
```

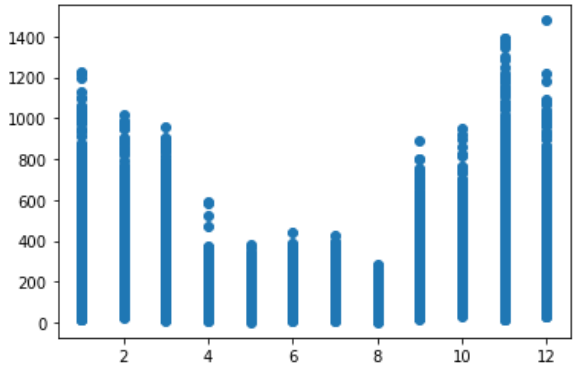


NOx(GT)

```
In [47]: plt.scatter(data['Hour'], data['NOx(GT)'])
Out[47]: <matplotlib.collections.PathCollection at 0x290cbd347f0>
```




```
In [48]: plt.scatter(data['Month'], data['NOx(GT)'])
Out[48]: <matplotlib.collections.PathCollection at 0x290cbd46bb0>
```

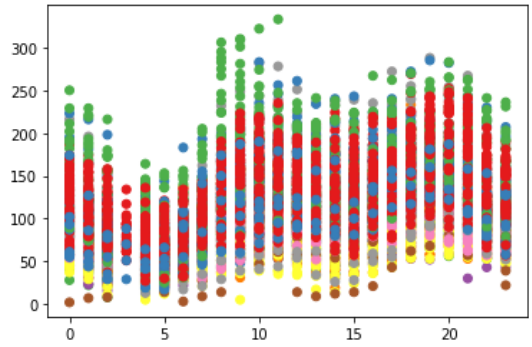


NO2(GT)

(I tried to color code the hour plot by month but it is hard to see which month is which so the colors can be ignored)

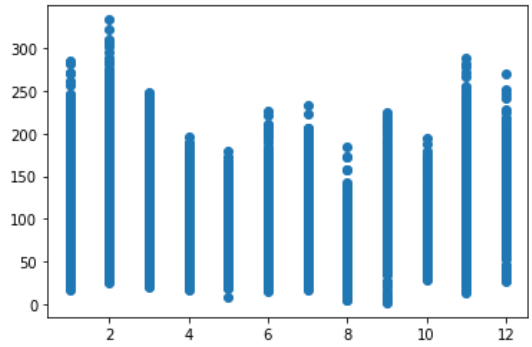
```
plt.scatter(data['Hour'], data['NO2(GT)'], c = data['Month'].map(color_map), label=color_map)
```

```
<matplotlib.collections.PathCollection at 0x290cab1eee0>
```




```
plt.scatter(data['Month'], data['NO2(GT)'])
```

```
<matplotlib.collections.PathCollection at 0x290cbe40280>
```



We can see that the month and hour impact these three variables enough to get a general trend in the data suggesting they are affected by the season and hour of the day. For this reason, I chose to impute the missing values in these columns with the average of the data grouped by month then hour. This is accomplished with the following lines of code (replace the appropriate variable name):

```
means = data.groupby(['Month','Hour'])['NOx(GT)'].mean()
rep = []
for i in data['NOx(GT)'].index:
    if data['NOx(GT)'].isna()[i]:
        rep.append(means[data['Month'][i],data['Hour'][i]])
    else:
        rep.append(data['NOx(GT)'][i])
data['NOx(GT)'] = rep
data['NOx(GT)'].isna().sum()
```

```
261
```

The final 2 lines replace the missing values with the appropriate group means then output the number of missing values remaining. For all three variables I found that there were some months where every observation for a specific hour of the day was missing. These regular missing values could be attributed to a researcher regularly checking the monitors at a specific time of day everyday of that month. Specifically, there are missing values for the variables at the following months and times:

CO(GT) is missing data for every observation at 5 am during May

NOx(GT) and NO2(GT) are missing data for every observation at 4 am in the months of January, February, May, June, July, August, September, October, and December.

To account for these missing values, I replaced the missing value with the average of the group means for the same month but 1 hour above and below. This is accomplished with the following code:

```
]: means = data.groupby(['Month','Hour'])['NOx(GT)'].mean()
   rep = []
   for i in data['NOx(GT)'].index:
       if data['NOx(GT)'].isna()[i]:
           new_val = (means[data['Month'][i],data['Hour'][i-1]]+means[data['Month'][i],data['Hour'][i+1]])/2
           rep.append(new_val)
       else:
           rep.append(data['NOx(GT)'][i])
   data['NOx(GT)'] = rep
   data['NOx(GT)'].isna().sum()
]: 0
```

In the end we see that all the missing data has been successfully imputed below and we are left with no missing values.

```
In [66]: data = data.drop(columns = ['NMHC(GT)'])

In [67]: data.isna().sum()

Out[67]: CO(GT)           0
         PT08.S1(CO)      0
         C6H6(GT)         0
         PT08.S2(NMHC)    0
         NOx(GT)          0
         PT08.S3(NOx)     0
         NO2(GT)          0
         PT08.S4(NO2)     0
         PT08.S5(O3)      0
         T                0
         RH               0
         AH               0
         DateTime         0
         Day_Of_Week      0
         Hour             0
         Month            0
         Day              0
         Year             0
         dtype: int64
```

Splitting Data into Training and Testing Sets:

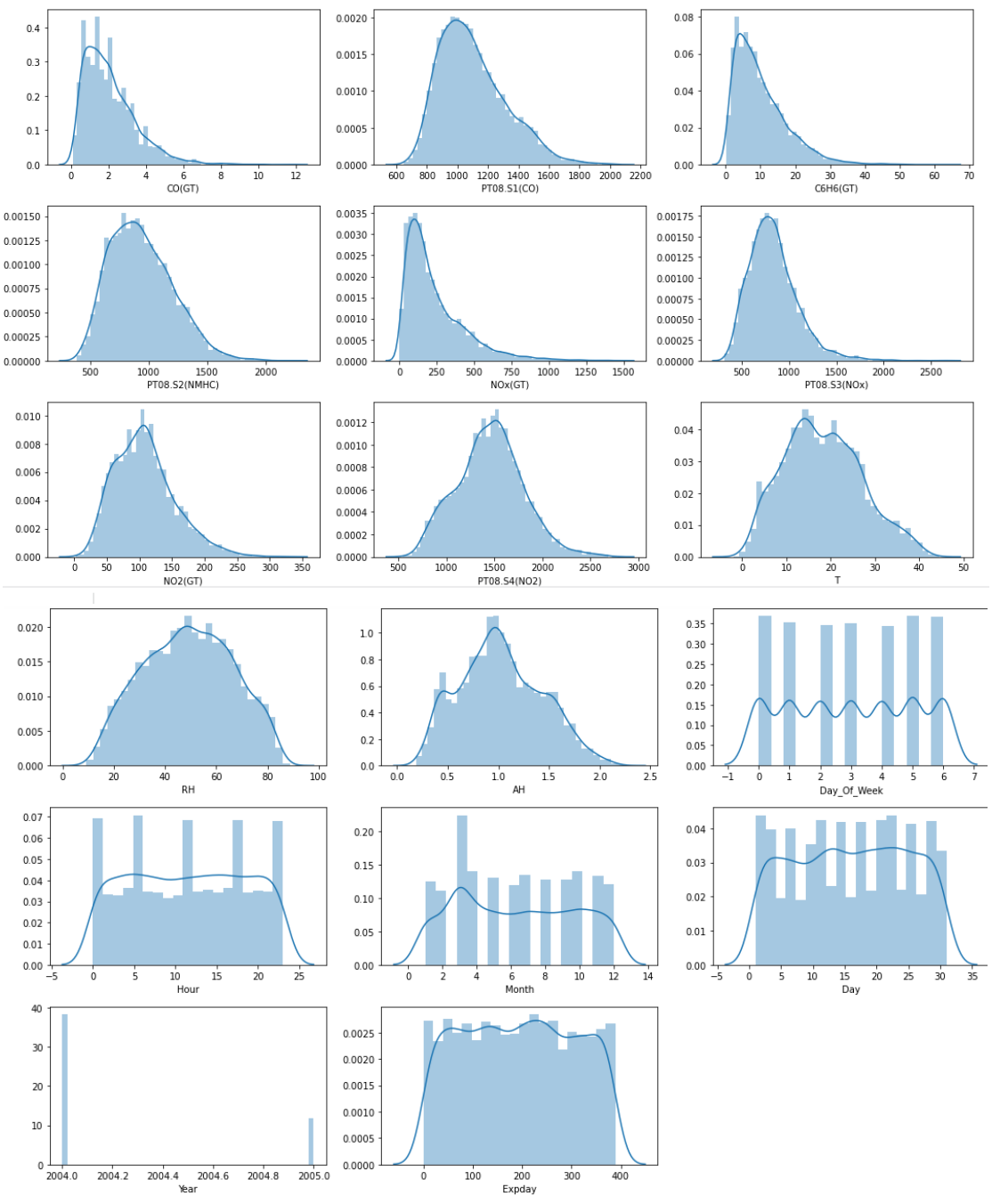
Using Sklearn's test_train_split function, I split the data set into an 80-20 split, using 80% of the data for training our models and 20% for testing the model on unseen data. Recall after removing some of the rows we are left with a total of 8991 observations, so after splitting we have 7192 samples for training models, and 1799 samples for testing our models. Recall that our response y variable is PT08.S5(O3) and we using a total of 17 predictor variables seen below.

```
|: X_train.dtypes

|: CO(GT)           float64
   PT08.S1(CO)      float64
   C6H6(GT)         float64
   PT08.S2(NMHC)    float64
   NOx(GT)          float64
   PT08.S3(NOx)     float64
   NO2(GT)          float64
   PT08.S4(NO2)     float64
   T                float64
   RH               float64
   AH               float64
   Day_Of_Week        int64
   Hour               int64
   Month              int64
   Day                int64
   Year               int64
   Expday             int64
   dtype: object
```

Exploratory Data Analysis:

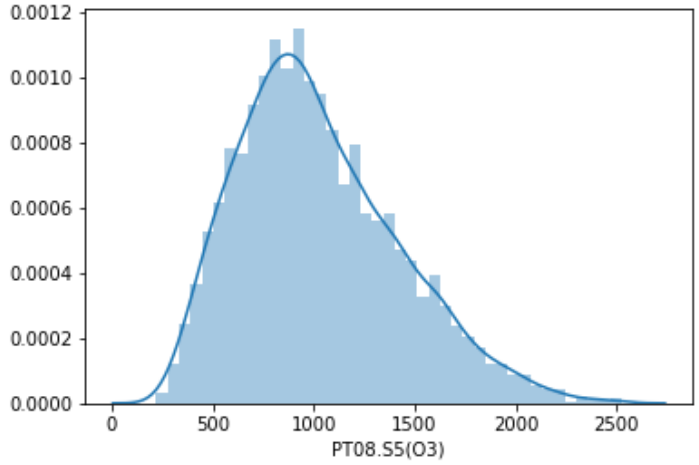
Below I have plotted the distributions of the feature variables in the training data.



Next, I have plotted the distribution of the response variable in the training data.

```
: sns.distplot(y_train)

: <matplotlib.axes._subplots.AxesSubplot at 0x290cce4dee0
```




```
]: from scipy.stats.mstats import normaltest
   normaltest(y_train)

]: NormaltestResult(statistic=413.17009971210285, pvalue=1.9109653121381873e-90)
```

We see that the data for the response variable is slightly skewed to the right. Running a test for normality we find that using D'Agostino's Normality Test the data is statistically different from a normal distribution with p-value = 1*10E(-90). We could use a Log transformation or Box Cox transformation to fix this, however having a normally distributed response variable is not necessary for OLS regression, so before doing any variable transformation I would like to see how a linear model would perform without any transformation.

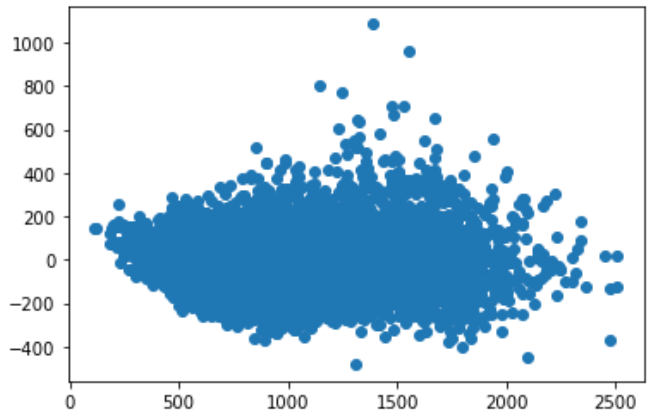
Linear Regression:

Using Sklearn's LinearRegression on the data we find that a normal OLS Regression using the 17 features achieves an R-squared value of 0.8938426195574631 on the training data which is pretty strong. The RMSE for this model is 130.19846869039853 ppm.

Checking the residuals vs fitted plot below we find that the model is struggling the most on intermediary values of PT08.S5(O3).

```
plt.scatter(prediction, residual)
```

```
<matplotlib.collections.PathCollection at 0x290cdfa
```

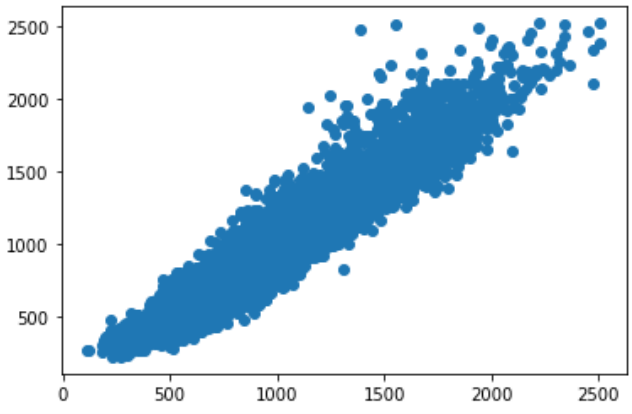


From this plot we find that the majority of predictions are within 400 ppm of the true value, but the model likely to underpredict large values to be closer to 1500 leading to multiple predictions of around 1500 when the true value is closer to 2500.

We can see the predicted vs true value plot below with the predictions on the X axis and the True value on the Y axis.

```
plt.scatter(prediction, y_train)
```

```
<matplotlib.collections.PathCollection at 0x290cc0664f0>
```



We see that the variance in the predictions increases as the predicted value gets larger, but in general the predictions are close to the true value. This suggests it might be suitable to replace the response variable with the boxcox transformation.

Applying a boxcox transform however, I found that the results did not improve as can be seen below.

```
y_train_log = np.log(y_train)
normaltest(y_train_log)
```

```
NormaltestResult(statistic=136.07686286114617, pvalue=2.8267322407588265e-30)
```

```
from scipy.stats import boxcox
bc_res = boxcox(y_train)
boxcox_y_train = bc_res[0]
lam = bc_res[1]
```

```
normaltest(boxcox_y_train)
```

```
NormaltestResult(statistic=62.81187750807082, pvalue=2.2938979861936103e-14)
```

```
lr_log = LinearRegression()
lr_log.fit(X_train, y_train_log)
lr_log.score(X_train, y_train_log)
```

```
0.8887572201678513
```

```
lr_bc = LinearRegression()
lr_bc.fit(X_train, boxcox_y_train)
lr_bc.score(X_train, boxcox_y_train)
```

```
0.8943922418962704
```

```
prediction_bc = lr_bc.predict(X_train)
residual_bc = (boxcox_y_train - prediction_bc)
normaltest(residual_bc)
```

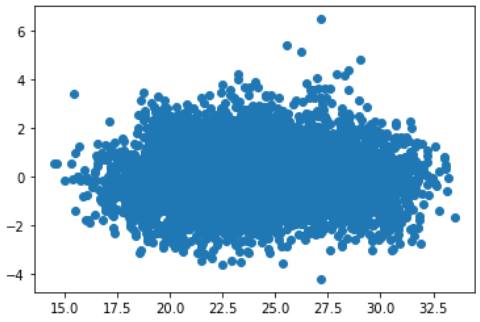
```
NormaltestResult(statistic=103.51394495834833, pvalue=3.328376540337062e-23)
```

We see above that normality of residuals has not been restored, nor is the target variable normally distributed.

Below we see that the prediction errors are still similarly distributed, but now there is less variance in the predictions on larger values of the target. This is only true in the transformed view, and using the inverse transform we find that our total model accuracy has gone down slightly, with equally poor performance on large values of the target.

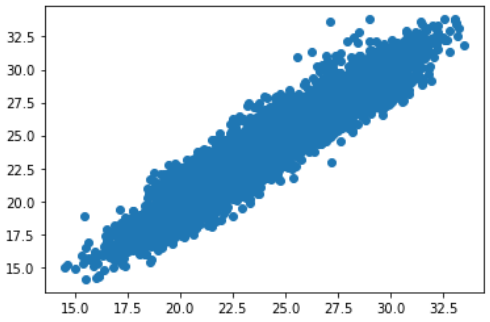
```
: plt.scatter(prediction_bc, residual_bc)
```

```
: <matplotlib.collections.PathCollection at 0x290d0afda60>
```



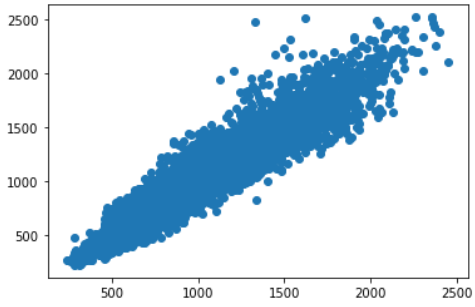

```
: plt.scatter(prediction_bc, boxcox_y_train)
```

```
: <matplotlib.collections.PathCollection at 0x290ce51d100>
```




```
]: from scipy.special import inv_boxcox
   truey = inv_boxcox(boxcox_y_train,lam)
   pred_true = inv_boxcox(prediction_bc,lam)
   plt.scatter(pred_true, truey)
```

```
]: <matplotlib.collections.PathCollection at 0x290d0baf5e0>
```




```
]: r2_score(pred_true, truey)
```

```
]: 0.8796590885242365
```

```
]: rmse_true = mean_squared_error(truey, pred_true, squared=False)
   print(rmse_true)
```

```
131.9627951053701
```

Overall, transforming our target variable has a negligibly negative effect on model performance reducing the total R-squared on the true values to 0.8796590885242365 and increasing the RMSE to 131.9627951053701 ppm.

Looking at the accuracy on the testing data we find that the model on the regular data achieves an R-squared of 0.8956480930504273 with an RMSE of 127.21610227473454.

Meanwhile the Boxcox model achieves an R-squared of 0.891781086244455 without inverting the transformation and an R-squared of 0.8834031045054416 after inverting the transformation, with an RMSE of 129.17828992369897 ppm. Again, the Boxcox model slightly under performs the normal model. Thus, we conclude that the Boxcox transformation is not needed.

Polynomial Features, Cross Validation, and Feature Scaling in a Pipeline:

Using the package of sklearn GridSearchCV along with a Pipeline estimator we can use cross validation to find the best hyper parameters to determine what is the highest order of interaction terms we should include in our model. We can achieve this using the following block of code.

```
]: from sklearn.preprocessing import StandardScaler, PolynomialFeatures
   from sklearn.model_selection import KFold, cross_val_predict
   from sklearn.model_selection import GridSearchCV
   from sklearn.metrics import r2_score
   from sklearn.pipeline import Pipeline

   kf = KFold(shuffle=True, random_state=7, n_splits=10)
   estimator = Pipeline([("scaler", StandardScaler()),
           ("polynomial_features", PolynomialFeatures()),
           ("linear_regression", LinearRegression())])
   params = {
       'polynomial_features__degree': [1, 2, 3]
   }

   grid = GridSearchCV(estimator, params, cv=kf)

]: grid.fit(X_train, y_train)

]: GridSearchCV(cv=KFold(n_splits=10, random_state=7, shuffle=True),
                estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                          ('polynomial_features',
                                           PolynomialFeatures()),
                                          ('linear_regression',
                                           LinearRegression())]),
                param_grid={'polynomial_features__degree': [1, 2, 3]})

]: grid.best_score_, grid.best_params_

]: (0.9321767547104851, {'polynomial_features__degree': 2})
```

For the polynomial linear model, we do not need to scale the data, but we will be using this in future models for LASSO and Ridge, so I just include it here to make the code easy to reuse for the future models. That being said here is the over view of how the pipeline and grid search works.

First, we start off by defining our cross-validation method. I chose to use KFold CV with k=10, which randomly partitions the training data into 10 chunks. KFold operates by building the model k times using k-1 of the chunks and scoring the model on the left-out partition. Each model leaves out a different chunk for the testing resulting in 10 models. The score on the out of sample data is averaged over each model to get an overall score for the model. The scoring method in this case is R-squared.

Next up the pipeline estimator is defined using a scaler, polynomial feature builder, and a modeling method. In this case I chose to use the Standard Scaler which takes each column of the training data and subtracts all values in the column by the column average and divides the difference by the column standard deviation, thus producing the standard Z-score for that observation in the column. For the model, we are still using a linear regression.

After defining the Pipeline, we define a dictionary of lists of hyperparameters for the model that we would like to test for the elements of the Pipeline. In the case of this model, we are only concerned with finding the appropriate highest order interaction term, so we set the list of possible values for the polynomial_features degree variable to be [1, 2, 3] so that we will only compare models of degree 1, 2, or 3. I chose not to go higher than degree 3 as this often leads to over fitting, and as we can see later in the picture the model found that degree 2 was better than degree 3 anyway, meaning higher order interactions are unnecessary.

Lastly, we define the grid object which will fit the defined estimator on a specified set of data using every possible combination of the parameters in the params dictionary of lists. Each model will be fit using the 10-fold cross validation method. In the end we can choose the appropriate parameters for the model as the model with the highest average score during cross validation.

In the end we find that the linear model achieves the highest average out of sample R-squared value using 2nd order interactions between the feature variables as predictors, with a cross validation R-squared value of 0.9321767547104851

GridSearchCV has a final step behind the scene, by default a grid model then refits the model to the entire dataset using the best parameters, meaning no data is left out in the final model.

After refitting we can find that the R-squared value on the entire training data is 0.9406166666316249 and the model has an RMSE of 97.37849672221881 ppm. Thus, we find that on the training data the model using 2nd order polynomial has significantly improved over the original 1st order model from the previous section.

On the testing data we retain our accuracy quite well thanks to our cross-validation methods and find we achieve a R-Squared of 0.9345212195661008 with an RMSE of 100.77253444915712 ppm.

Ridge Penalized Regression Model:

With Ridge Regression, we penalize the model based on the size of the coefficients by adding the L2 Norm of the coefficients to our OLS estimate that we wish to minimize as follows:

$$\hat{\beta}^{\text{ridge}} = \underset{\beta}{\operatorname{argmin}} \left\{ \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij} \beta_j \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}.$$

Here $\beta^{ridge} = < \beta_0, \beta_1, \ldots, \beta_p >$ is the vector of coefficients for the p columns of X plus the intercept. In normal OLS, we find our estimate for β^{OLS} using the same formula without the final summation term. In ridge regression we penalize the model for using excessive numbers of features by adding on the sum of the square of each coefficient (excluding the intercept) to the formula which we wish to minimize. This encourages the model to find the best model using the fewest number of features possible, as removing features from the model means setting $\beta_k = 0$ for the feature k you wish to remove form the model, which reduces the penalty term. The value λ in ridge regression is a hyper parameter set by the data scientist to determine the level of penalty desired. Larger values of λ ensure stricter penalties for features. In our case, we will be using GridSearchCV to find the appropriate value for λ. In Python, this value is called alpha instead of lambda.

Note that Ridge regression requires that the scale of the feature variables and the target to be similar so that the sizes of the coefficients have equivalent effects on the model output. If we allow for unequally scaled columns, then then we may have a column which is very important on a small scale requires a large coefficient to reach the scale of the target, while a feature with a larger scale requires a smaller coefficient to produce the same result. As a result, Ridge regression favors features with larger scales by default so to remove this bias it is necessary to scale the data. As previously mentioned, I chose to use the Standard Scaler to achieve this.

It should be mentioned that Ridge Regression is notable for rarely opting to remove features in practice as using the squares of each coefficient makes them sufficiently small to be within tolerance levels for convergence using default library tolerances for Iteratively Weighted Least Squares Regression methods implemented to find the minimizing coefficients. We will find that LASSO Regression is much more likely to remove variables while Ridge will make many of the coefficients quite small for weak predictors.

Below we see the code used to build a Ridge Regression Model on the data.

L2 Norm Penalized (Ridge) Regression

```
from sklearn.linear_model import Ridge
kf_rid = KFold(shuffle=True, random_state=7, n_splits=10)
estimator_rid = Pipeline([("scaler", StandardScaler()),
        ("polynomial_features", PolynomialFeatures()),
        ("ridge_regression", Ridge())])
params_rid = {
    'polynomial_features__degree': [1, 2, 3],
    'ridge_regression__alpha': np.geomspace(0.01, 100, 100)
}

grid_rid = GridSearchCV(estimator_rid, params_rid, cv=kf_rid)
```

```
grid_rid.fit(X_train, y_train)
```

```
GridSearchCV(cv=KFold(n_splits=10, random_state=7, shuffle=True),
             estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                       ('polynomial_features',
                                        PolynomialFeatures()),
                                       ('ridge_regression', Ridge())]),
             param_grid={'polynomial_features__degree': [1, 2, 3],
                         'ridge_regression__alpha': array([1.00000000e-02, 1.09749877e-02, 1.20450354e-02, 1.32194115e-02,
       1.45082878e-02...
       1.17681195e+01, 1.29154967e+01, 1.41747416e+01, 1.55567614e+01,
       1.70735265e+01, 1.87381742e+01, 2.05651231e+01, 2.25701972e+01,
       2.47707636e+01, 2.71858824e+01, 2.98364724e+01, 3.27454916e+01,
       3.59381366e+01, 3.94420606e+01, 4.32876128e+01, 4.75081016e+01,
       5.21400829e+01, 5.72236766e+01, 6.28029144e+01, 6.89261210e+01,
       7.56463328e+01, 8.30217568e+01, 9.11162756e+01, 1.00000000e+02])})
```

```
grid_rid.best_score_, grid_rid.best_params_
```

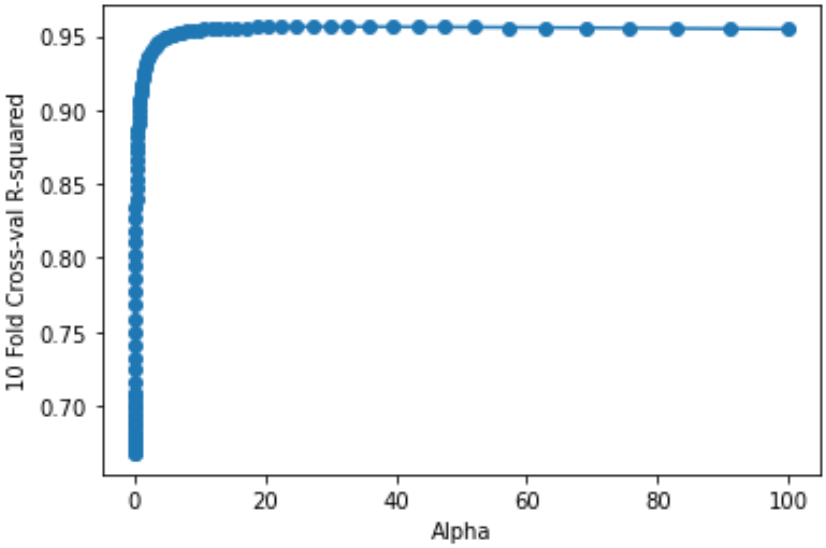
```
(0.9562651773109637,
 {'polynomial_features__degree': 3,
  'ridge_regression__alpha': 29.836472402833405})
```

For this model I used the same 10Fold cross validation method as used for the polynomial model with the same random state meaning the scores of the models are less random but more comparable to the scores obtained by the polynomial model. Overall, the code is very similar to before however we now include a list of alpha values to test for our model. Using the suggestions of this course, I tested alpha values using an evenly spaced set of alpha values over a logarithmic scale using NumPy's geomspace function. This function produces an array of values evenly spaced between and including Log (0.01) and Log (100) (my first two arguments). The length of the array is the 3rd argument, which I chose to be 100. These values are then exponentiated to produce a list between 0.01 and 100 inclusive. Lastly, we are scaling using a standard scaler and testing using polynomial features with degrees 1, 2, and 3. In total our models will be fit via cross validation 10 times for each pair of alpha and degrees, meaning we are fitting a total of 10*3*100 = 3000 models for cross validation comparison, and then the best model is refit to the entire training data set, meaning we are training total of 3001 models. This can take a long time, but can be sped up considerably on a multicore processor, by fitting each set of 10 Coss validated models on a single core, however my laptop would not run this notebook in parallel, meaning I ran the entire calculation on a single core which took about 5-10 minutes.

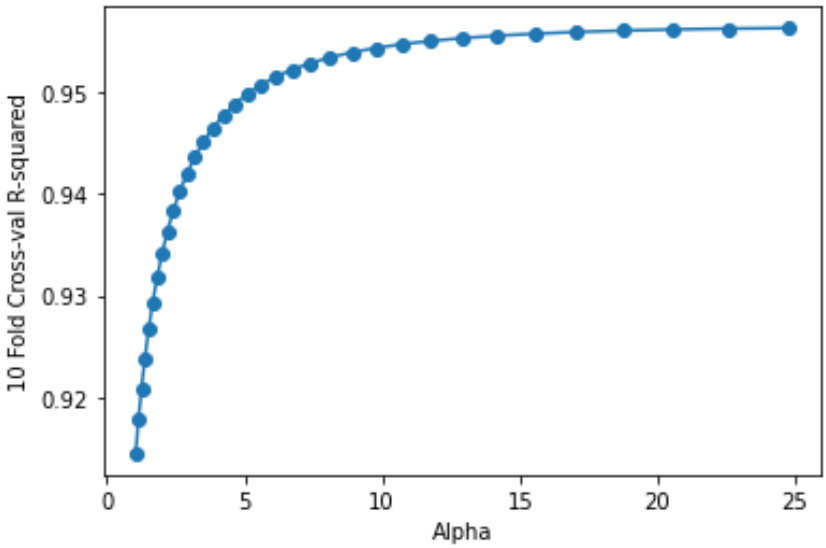
The model finds via cross validation that the best hyper parameters out of those tested are using 3rd order interaction terms and an alpha value of 29.836472402833405. Looking at our list of alphas tested we see that our model also tested alpha = 27.1858824 and alpha = 32.7454916

giving us an upper and lower bound for possible choices of alpha, and our model happened to test 29.836472402833405 in that range. For this project I do not have a specific tolerance for the range on alpha which I would like to define but we could refit a new grid search model now using only degree 3 polynomial inputs and alpha values between the new upper and lower bounds to improve the model hyper parameters further. I will however plot the average cross validation score as a function of the value of alpha to show that this value peaks between 27.1858824 and 32.7454916.

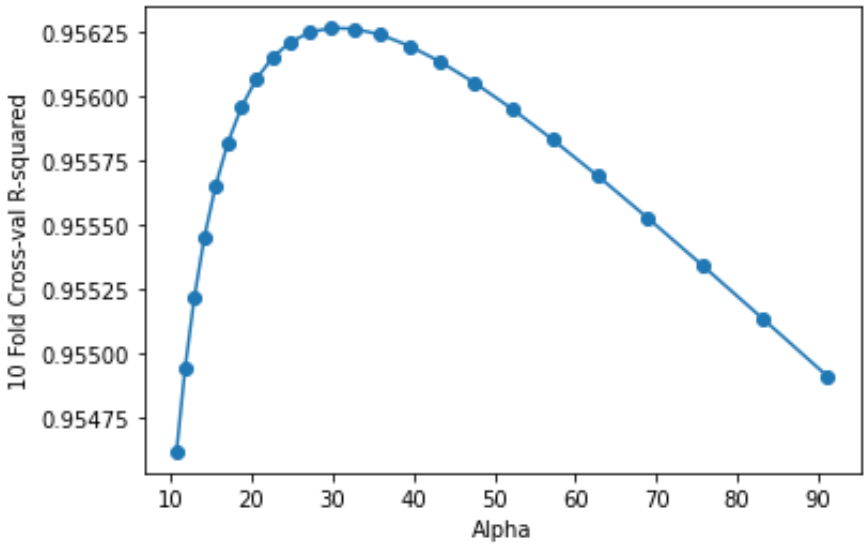
Refitting the data now only using degree 3 polynomials we find that the plot of the average R-squared cross validation score as a function of alpha quickly reaches 0.95 and very little difference in the average score is seen beyond an alpha around 5 in the plot below.



Zooming in further we find the slope is not exactly zero and is still growing between 5 and 25 but only by less than 0.01



Lastly, we zoom in on the peak to find that an alpha of 29.836472402833405 is only 0.001 better than its two nearest neighbors and less than 0.002 better than any alpha above 10.



We conclude that our choice of alpha is not necessary for any further optimization and we could choose any value between 10 and 90 for alpha with negligible differences in the overall model score.

By default, we use the best model tested with degree 3 polynomial interactions and an alpha of 29.836472402833405. Refitted to the entire training dataset we get a model with an R-squared = 0.9649823601721527 on the training data and an RMSE = 74.77806372186178 ppm on the training data.

Moving over to the testing data we retain or strong model scores with an R-squared = 0.9579539599205492 and an RMSE = 80.75221437161132 indicating this model is an improvement over the polynomial degree 2 model from earlier.

Lastly, we look at the number of features used in the model. Recall that the total features in a polynomial model is found using the formula $\sum_{k=0}^{d}\left(\frac{f!}{k!(f-k)!}\cdot\frac{d!}{k!(d-k)!}\right)$-1 where f is the original number of features in the model and d is the degree of the polynomial. We subtract 1 to remove the intercept in this equation which is not a feature variable. We started with 17 explanatory variables for our model meaning that a degree 2 model uses 171-1= 170 features plus an intercept while a degree 3 model uses 1140-1 = 1139 features plus an intercept. This can be a massive tax on computation time if we intend to use all interactions in a model, and if we use the model on a large dataset, we may not be able to store the data for the model efficiently with 1139 columns. Lastly the whole point of using a Ridge Regression was to hopefully remove some less important features, so we should see if that was accomplished.

We can use python to check how many coefficients in the model became zero for our ridge model in the code below.

```
: grid_rid.best_estimator_.named_steps['ridge_regression'].coef_

: array([  0.        , -12.42998285,  70.68100359, ..., -16.39984561,
         -7.0214349 ,  -2.82279967])

: zeros = 0
  non_zeros = 0
  for i in grid_rid.best_estimator_.named_steps['ridge_regression'].coef_:
      if i == 0:
          zeros+=1
      else:
          non_zeros+=1
  print('Total predictors = ' + str(zeros+non_zeros))
  print('Total Zeroed out coefs = ' + str(zeros))
  print('Predictors used = ' + str(non_zeros))

  Total predictors = 1140
  Total Zeroed out coefs = 1
  Predictors used = 1139
```

WOW! We removed 1 coefficient, and even better, we can see that the only zero coefficient was the intercept term (the first term in the coefficients array is the intercept term and this value is 0). I check this for a Ridge model using alpha = 5 since that was the minimum value to achieve a cross validation score of 0.95, but even at a lower alpha value I found that the model still used 1139 predictors. In the end we find that although penalizing the coefficients did help improve the model on 3rd order interactions (as we saw that the OLS model on polynomial degree 3 was weaker than a degree 2 OLS model), our penalty was unable to remove any features from our model. Thus, if we wish to use this model for any sort of interpretation to advise law makers on ways to improve air quality, we could choose the polynomial degree 2 Ordinary Least Squares Regression model, but if we only wish to accurately predict the presence of PT08.S5 (indium oxide) in the air to advise vulnerable respiratory patients to stay indoors, then we would probably prefer the Ridge Regression Model.

LASSO Penalized Regression:

Again, we are seeking to penalize the use of an excessive number of variables in our model. This time we are using the Least Absolute Shrinkage and Selection Operator, which as the name implies, uses the Absolute value of the coefficients as the penalty for variable selection. Using the absolute value of the coefficients avoids the issue mentioned earlier where squaring the coefficients for sufficiently small values makes them small enough to avoid removal entirely. The absolute value of each coefficient is summed together and multiplied by a hyper parameter λ once again. The L1 norm of the vector β gives the sum of the absolute value of each element in beta, so LASSO can be considered the L1 Normalization Penalty while Ridge was the L2 Normalization Penalty. The equation below gives the cost function we wish to minimize for LASSO Regression. Again, we get the OLS Mean Squared Error as the first term (note this function considers X to include a column of all 1s for the intercept term so it is not pulled out as it was before.) and the penalty is the sum of the absolute values of the coefficients multiplied by λ.

$$L_{lasso}(\hat{\beta}) = \sum_{i=1}^{n}(y_i - x_i^T\hat{\beta})^2 + \lambda\sum_{j=1}^{m}|\hat{\beta}_j|$$

In python again λ is called α but the math is entirely the same, however we solve the equation using iteratively reweighted least squares, meaning that a tolerance and maximum number of iterations must be set for convergence. Since LASSO has a much higher tendency to remove variables the function takes much longer to converge than Ridge as it will need to determine new coefficients entirely once the variables are removed. This causes the computation for LASSO Regression to take much longer, however if alpha is large enough, we remove so many variables near the beginning of the calculation that the time for computation actually decreases. In any case LASSO on average takes much longer to converge than Ridge, so for my machine I was not able to check nearly as many values for alpha nor was it feasible to use k=10 for KFold Cross Validation. Instead, I used k=3 and tested using 10 alpha values geometrically evenly spaced between 0.1 and 1, along with every 0.5 value for alpha from 1.0 up to 20.0. Using LASSO, I also ran the calculations on the polynomial interactions of order degree 3, so that the solution would be comparable to that found using Ridge.

For the code in this section, I decided to use sklearn's cross_val_predict method to score the models on individual levels of alpha and so that I could view the cross-validation score as a function of alpha, and add more values of alpha as needed. Below we see the general structure for the code.

```
: from sklearn.model_selection import cross_val_predict
  from sklearn.linear_model import Lasso
```

```
: s = StandardScaler()
  kf_las = KFold(shuffle=True, random_state=7, n_splits=3)
  pf = PolynomialFeatures(degree=3)
  scores = []
  alphas = np.geomspace(0.1, 1, 10)
  for alpha in alphas:
      las = Lasso(alpha=alpha, max_iter=10000)

      estimator = Pipeline([
          ("scaler", s),
          ("make_higher_degree", pf),
          ("lasso_regression", las)])

      predictions = cross_val_predict(estimator, X_train, y_train, cv = kf_las)


      score = r2_score(y_train, predictions)

      scores.append(score)
      print(alpha)
```

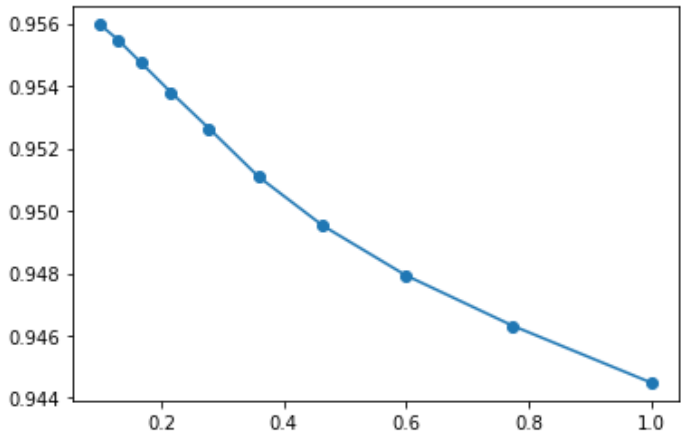
```
C:\Users\denis\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:529: ConvergenceWarning: Objective did not converge. You might want to increase the number of ite
ations. Duality gap: 1796070.8269230947, tolerance: 74758.13761061743
  model = cd_fast.enet_coordinate_descent(
C:\Users\denis\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:529: ConvergenceWarning: Objective did not converge. You might want to increase the number of ite
ations. Duality gap: 1875458.9200012758, tolerance: 77419.9048997706
  model = cd_fast.enet_coordinate_descent(
C:\Users\denis\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:529: ConvergenceWarning: Objective did not converge. You might want to increase the number of ite
ations. Duality gap: 2137178.5212712735, tolerance: 77500.09082494266
  model = cd_fast.enet_coordinate_descent(

0.1

C:\Users\denis\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:529: ConvergenceWarning: Objective did not converge. You might want to increase the number of ite
ations. Duality gap: 934774.2344775908, tolerance: 74758.13761061743
  model = cd_fast.enet_coordinate_descent(
```

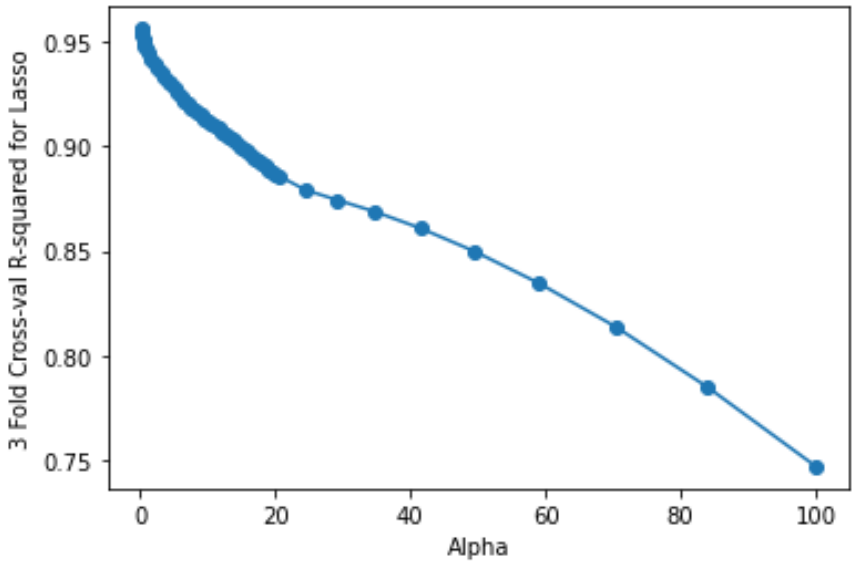
The code is almost entirely the same until the estimator is defined. Instead of fitting a grid search object however we instead make a prediction on the training data using cross validation. We then use these predictions to calculate the average R-squared score for each value of alpha and store these values in a list named scores. For my own benefit, I also printed out the alpha value at the end of the loop to see how far the calculations had progressed. A side benefit of this can be seen in the output, where we note the red error blocks denoting that the calculation did not converge for alpha = 0.1 in the specified number of maximum iterations for calculations. Using a maximum number of iterations set to 10,000 the calculation was unable to converge for only the smallest values of alpha, specifically 0.1, 0.1291549665014884, 0.16681005372000587, and 0.21544346900318834. These values of alpha can be ignored for future consideration as the time needed to calculate the model makes these values inefficient and makes using a lasso regression model a poor choice for these values anyways.

From these models we can look at the average cross validation r-squared value as a function of alpha below.

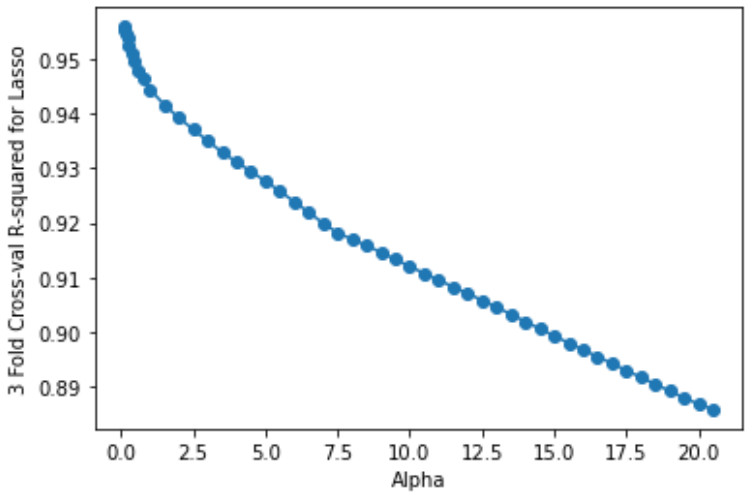


We find that for alpha values between 0.22 and 1, the average cross validated R-squared on the data is between 0.953 and 0.944 meaning that all of these models better than the OLS Polynomial Degree 2 model, and are within 0.01 of the R-squared for the best model using Ridge Regression.

I decided to extend the method used to calculate the cross validated R-Squared score using alphas between 0.2 and 100 which can be seen below.



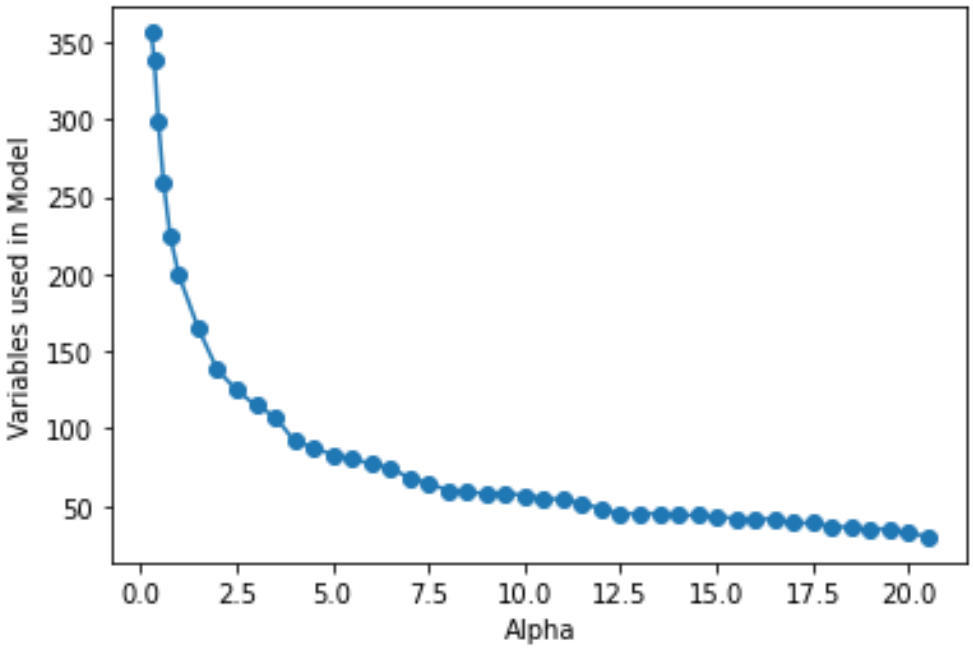
In the graph we can see that I more densely studied models in the range from alpha=0.2 to alpha = 20 and we see that for values above 20 the R-squared is worse than for our Linear Model using no interaction terms, meaning these models are not useful to us so I will zoom in on the values less than 20.



In this range we find that for alpha values between 0.2 to 1 the R-squared value drops fast but at a decreasing rate, then from alpha =1 up to alpha = 7.5 the R-squared seems to decrease linearly with alpha at a slope of about -0.02/6.5 = -0.00307692. After alpha = 7.5, the R-squared still decreases linearly but with a smaller slope of –0.03/12.5 = -0.0024

There is no suggestion on the best alpha to pick in this case however it is often useful to pick values of alpha where the slope seems to develop a kink or elbow (points where the slope changes). In this case the best choices for alpha seem to be 0.6, 1.0, 2.0, and 7.5.

Before we study these models further to select the best choice for our purposes, we should look at the number of estimators as a function of alpha which I have lotted below.



We see that as alpha increases the number of estimators removed from the model drops significantly at first but slows down considerably at alpha = 2, then at alpha = 4 the slop of the graph minimizes suggesting that we retain only the strongest predictors for our model at this point. Checking exactly, we find that at alpha = 2 the model uses 138 predictors and at alpha = 4 the model uses 92 predictors. In both cases this is well below the number used by the Polynomial Degree 2 model under OLS Regression and over 1000 fewer than the Ridge Model uses.

The alphas of particular interest to compare now are 0.6, 1.0, 2.0, 4.0, and 7.5. I will produce a table of the important scores related to these models.

Alpha	R-squared on Training Data	RMSE on Training Data (PPM)	Predictors in Model	R-squared on Testing Data	RMSE on Testing Data (PPM)
0.6	0.952	87.21	259	0.946	88.88
1.0	0.948	90.98	199	0.942	91.87
2.0	0.942	96.18	138	0.935	96.15
4.0	0.933	103.13	92	0.924	102.33
7.5	0.920	113.29	65	0.905	112.39

We find that for alphas between 4 and 0.6 the Lasso model is at least as strong as the polynomial degree 2 linear model suggesting that we should only choose between these models. Second, we find that the 0.6 model uses 120 more predictors than the 2.0 model (in fact this is almost double the number of predictors) for an increase of 0.01 in R-squared on both testing and training, an only 8 ppm for RMSE meaning the average information gain from these predictors is less than 0.0001 in R-squared and less than 0.08 in RMSE. It therefore seems more appropriate to choose the 2.0 model over the 0.6 model in terms of the average strength of the predictors. By a similar argument we would find that the 1.0 model is less useful than the 4.0 model as it adds 100 more predictors (more than double the predictors) for a 0.015 increase in R-Squared and a 12 ppm increase in RMSE. If accuracy is our sole priority, then of course we would choose the 0.6 model, but if reducing the number of predictors is important than we would choose either the 4.0 or 2.0 model. For deciding between the 4.0 and 2.0 models, I refer back to the analysis of the plot of predictors as a function of alpha, where we noted that it seems the model has removed all weak predictors at alpha = 4.0 at which point the slope of the graph decreased by a large amount while between 3.5 and 4.0 the graph has the largest drop in predictors for any 0.5 increase in alpha above 1.5. For this reason, I would suggest using alpha = 0.6 if accuracy is important for the model, while 4.0 would be the best choice if understanding the model is important.

Overall Model Selection:

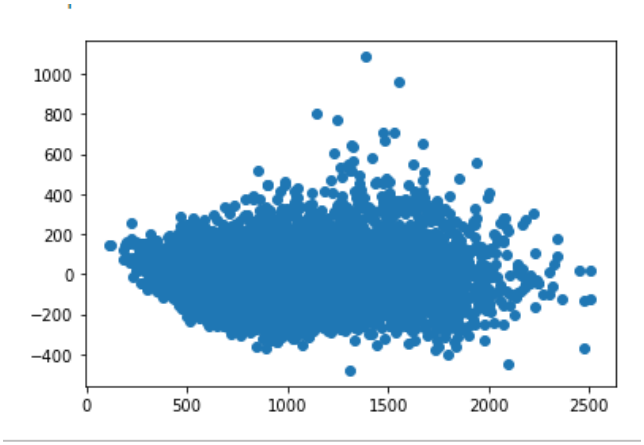
I will place in a table the list of the best models similar to the table in the previous section, for the 4 regression models that we used, OLS with no interaction terms, OLS with polynomial interactions, Ridge (L2 Norm Penalized) Regression, and LASSO (L1 Norm Penalized) Regression

Model	R-squared on Training Data	RMSE on Training Data (PPM)	Predictors in Model	R-squared on Testing Data	RMSE on Testing Data (PPM)
Linear	0.894	130.20	17	0.896	127.22
Poly Degree = 2	0.941	97.38	171	0.935	100.77
Ridge (alpha = 29.836, degree 3)	0.965	74.78	1139	0.958	80.75
Lasso (alpha =0.6, degree = 3)	0.952	87.21	259	0.946	88.88
Lasso (alpha =4.0, degree = 3)	0.933	103.13	92	0.924	102.33

If accuracy is our main concern, then it is clear that the Ridge Regression model is the winner, but this uses 1139 predictor variables, so it would not be useful in understanding the factors our model is based on. We also can see that with the Lasso model using alpha = 0.6 we can achieve almost equivalent results with nearly 1/5 the total predictors indicating that there are many unnecessary features in our Ridge model which may lead to confounding results on future data. We see a similar relation between the Polynomial degree 2 model and the Lasso model with alpha = 4.0, in that we achieve very similar results with almost half the predictors.

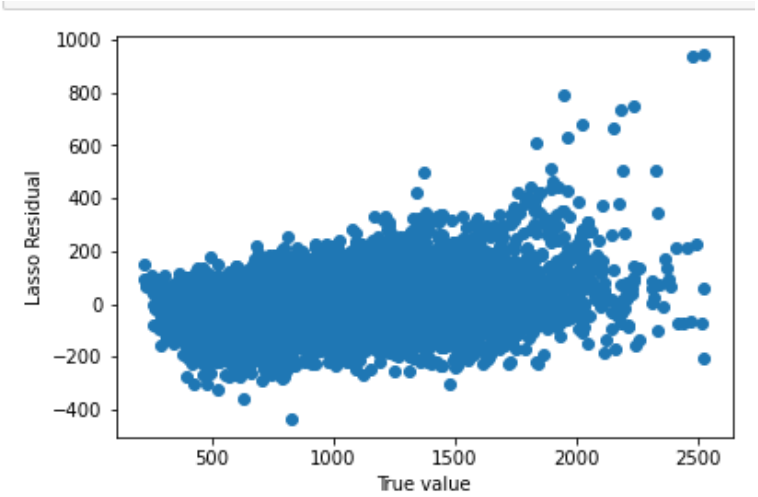
For Model Selection, I would suggest using the Lasso model with penalization parameter alpha = 4.0 if an interpretable model is desired, as it contains 3rd order interaction polynomial terms but uses fewer total variables than a degree 2 polynomial (almost half) with similar predictive power on both the training data and unseen testing data. If we seek a highly accurate model, I would suggest using the Ridge Regression model using degree 3 polynomial interactions and penalization parameter alpha = 29.836472402833405. However, it is likely that this model is highly over fit to the dataset and is also computationally inefficient so it may be weaker than the LASSO model with penalization parameter alpha = 0.6 using degree 3 polynomial interactions.

I also wanted to check the distribution of the residuals for the two models chosen. Recall that for the OLS linear model our predictions for the training data were distributed as seen below.



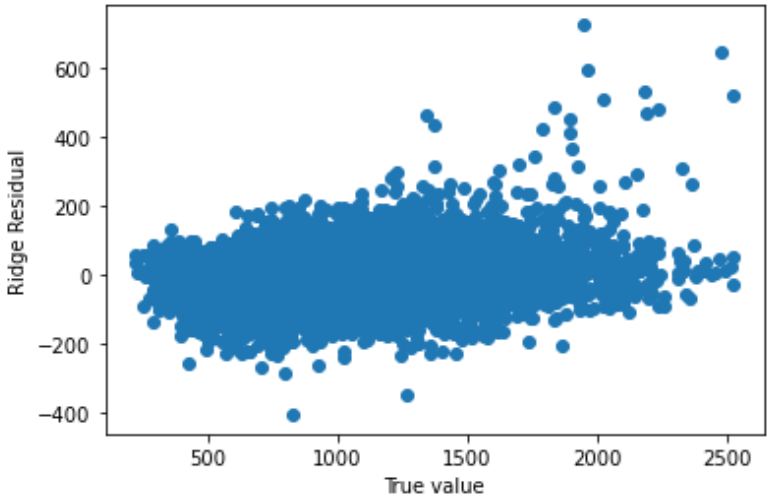
From this plot we could see that the model was tending to under predict by an average of 200 less than the true value for low values of the target, but tended to over predict with an average 200 above the true value of for large values of the target. In the middle of the distribution the model tended to under predict by an average of 200 below the target but could be as extreme as 1000 below.

Next, we observe the residuals vs actual plot for the Lasso Model using alpha = 4.0



Here we find that the residuals are distributed with a mean of 0 for small and medium values of the target variable, but for very large values of the target variable we tend to under predict with an average prediction 400 ppm below the true value, with the most extreme residuals being around 980 ppm below the true value when the Indium Oxide is actually around 2500 ppm. Aside from these two extremes, the residuals for this model have a much smaller variance and are better distributed around 0 than the linear model indicating our Lasso model is a better predictor. We should be aware of the propensity to under predict when the PPM is above 2000

Last up is the Ridge Model.



This model has an average residual of 0 for small and medium values of the target variable. Like the Lasso Model we tend to under predict for large observed PPM levels, however the largest residual is now 700 below the true value when the true observed actual ppm was around 2000 ppm. In general, the residuals for this model have a much smaller variance and the range is smaller than that of the Lasso Model, thus the Ridge model is a better predictor of the true ppm.

Important Features:

Lastly, I have reproduced below the list of features selected by the Lasso Model using alpha = 4.0 and their associated coefficient in the model to show the most influential features. The coefficients are ordered by the number of interactions between variables. There is no implied order of variables with an equivalent number of interactions.

```
'PT08.S1(CO)': 145.12965537767937,
      'PT08.S2(NMHC)': 134.70803656758034,
      'PT08.S3(NOx)': -48.743577415793844,
      'NO2(GT)': 1.8101782292533517,
      'Hour': -56.02909822708514,
      'PT08.S1(CO) NO2(GT)': 8.416095818468781,
      'C6H6(GT) RH': -10.943937288158494,
      'NOx(GT) RH': -15.931127196655261,
      'NOx(GT) Hour': 1.885990978754237,
      'NO2(GT) RH': -8.708706206366676,
      'PT08.S4(NO2) T': 12.003189523015514,
      'T Day_Of_Week': -1.778727911124095,
      'T Hour': -7.658399784905559,
      'RH^2': -16.42681230753713,
      'AH Day_Of_Week': -1.1402070713685937,
      'Day_Of_Week^2': -7.793825397712833,
```

```
'Day_Of_Week Day': 0.1645409773638183,
'Hour^2': 22.82372608038908,
'Expday^2': -11.696231720951609,
'CO(GT)^3': 0.040649984504274136,
'CO(GT)^2 Hour': -3.205861376911894,
'CO(GT)^2 Month': 0.3068577329301263,
'CO(GT)^2 Day': 0.27989160608080416,
'CO(GT)^2 Year': -2.18591255493694,
'CO(GT) C6H6(GT)^2': -0.396003545272386,
'PT08.S1(CO)^2 NO2(GT)': 2.890093960810776,
'PT08.S1(CO)^2 PT08.S4(NO2)': -2.733980533304987,
'PT08.S1(CO)^2 Day': 1.5391785521992143,
'PT08.S1(CO) NOx(GT)^2': 1.3247441184176756,
'PT08.S1(CO) NOx(GT) AH': -2.622438956761014,
'PT08.S1(CO) PT08.S4(NO2) T': 3.5662278790757953,
'PT08.S1(CO) T Expday': -4.473340573980969,
'PT08.S1(CO) AH^2': 1.5735374512146907,
'PT08.S1(CO) Month^2': 34.20375985587053,
'PT08.S1(CO) Month Day': 0.19799777858041734,
'PT08.S1(CO) Year^2': 7.23224228909134,
'C6H6(GT)^3': -0.06445937914186958,
'PT08.S2(NMHC) T Day': -1.6763556091349512,
'PT08.S2(NMHC) Day_Of_Week^2': 4.853593566925198,
'PT08.S2(NMHC) Day_Of_Week Year': 0.46679237363317916,
'NOx(GT)^3': 0.05280629020811531,
'NOx(GT)^2 PT08.S4(NO2)': -2.393117743793517,
'NOx(GT)^2 Day_Of_Week': -0.23081580952341885,
'NOx(GT)^2 Month': 1.7055465290817209,
'NOx(GT)^2 Day': 0.5553904074401304,
'NOx(GT) PT08.S3(NOx) Day': -1.6020734008520414,
'NOx(GT) Day_Of_Week^2': 3.962704663440932,
'NOx(GT) Day^2': 2.5085069848041504,
'NOx(GT) Year^2': 6.32348842065331,
'NOx(GT) Expday^2': 10.525972025165622,
'PT08.S3(NOx)^3': 0.491760836770277,
'PT08.S3(NOx)^2 Month': 2.7436391316281385,
'PT08.S3(NOx) PT08.S4(NO2) T': -4.031379182399431,
'PT08.S3(NOx) Hour^2': -8.6741238560604,
'PT08.S3(NOx) Month Day': -6.574090378898675,
'PT08.S3(NOx) Expday^2': -3.927341575788943,
'NO2(GT)^2 Month': 0.38984523951746747,
'NO2(GT) PT08.S4(NO2) T': 5.7943108500932015,
'NO2(GT) T Hour': -1.8825349857075322,
'NO2(GT) RH Day_Of_Week': 2.2680776736035195,
'NO2(GT) Hour^2': 3.2208893311102975,
'NO2(GT) Day^2': 1.0032492913564648,
'NO2(GT) Day Year': -0.6768975937028107,
'PT08.S4(NO2)^2 Year': -5.811548681067905,
```

```
'PT08.S4(NO2) Month^2': -4.173906725914063,
'T^3': -1.9042324839667395,
'T^2 AH': -9.411459760448997,
'T^2 Hour': -9.92832942720318,
'T^2 Day': -0.010690313086473092,
'T RH Hour': 1.1208821062381698,
'T Day_Of_Week^2': -3.2625630414107394,
'T Month^2': -16.695394431806104,
'T Day Year': 4.539549814549717,
'RH^3': -4.8304303144362315,
'RH^2 Hour': -0.2909654980839629,
'RH^2 Month': 1.45383175813316,
'RH AH Month': 1.7309979422420634,
'RH Day_Of_Week Month': -1.8145475256946546,
'RH Day^2': 4.33057349034303,
'RH Expday^2': 2.888145723915533,
'AH^3': -3.1463823246151112,
'AH Day_Of_Week Month': -1.3891581000260957,
'AH Hour^2': -6.883641747932717,
'AH Month Day': -2.144087691552822,
'Day_Of_Week^2 Month': 5.672163486290576,
'Day_Of_Week Hour^2': -3.6157159111353,
'Day_Of_Week Hour Month': 0.22625708852172374,
'Day_Of_Week Month Day': 1.8079285306298254,
'Hour^3': 23.019808310175552,
'Hour Day^2': -2.8101326555997055,
'Hour Expday^2': -7.111376600564624,
'Month^2 Day': -2.808029984430886
```

Next Steps and Future Suggestions for Data Collection:

Our model was used to help predict the presence of Indium Oxide, a byproduct of combustion in car engines, in the air based on data from the year 2004 in an Italian city. Since then, the European Union has incentivized consumers to purchase electric cars which should help reduce emissions from motorists. If these trends have made an impact, then we should find that our model would tend to over predict the ppm detected on new data as our model would have been fit to data following a different distribution with a higher average pollution measurement. Therefore, it would be ideal for a new study to be conducted collecting the data in a similar manner to the data used for the original study and see how our models perform on the new data. The new study would not necessarily need to record every hour for an entire year, but we could collect data from a random sample of hours during the year. If we find that the model is regularly over estimating the pollution level, we would have evidence that the efforts to reduce carbon emissions have improved the air quality in this Italian city.

Secondly, as stated in the introduction it might be useful for the city to know when the air quality is poor, so that we can properly advise citizens with respiratory illnesses to stay indoors. To make such a recommendation we could use the predictions output by our model and have a medical professional advise us on a sufficient threshold at which we should inform the public. However, in an alternative approach we could train a classifier model to predict when the observed value is above the recommended threshold. As seen in our residual plots, our models tended to under predict the amount of Indium Oxide in the air when the true amount was above 2000 ppm, so it would help to have a classifier which can inform us if it is possible that our regression model has made a mistake.