Ozone Analysis

Leon Zha
ITP449
James Faghmous

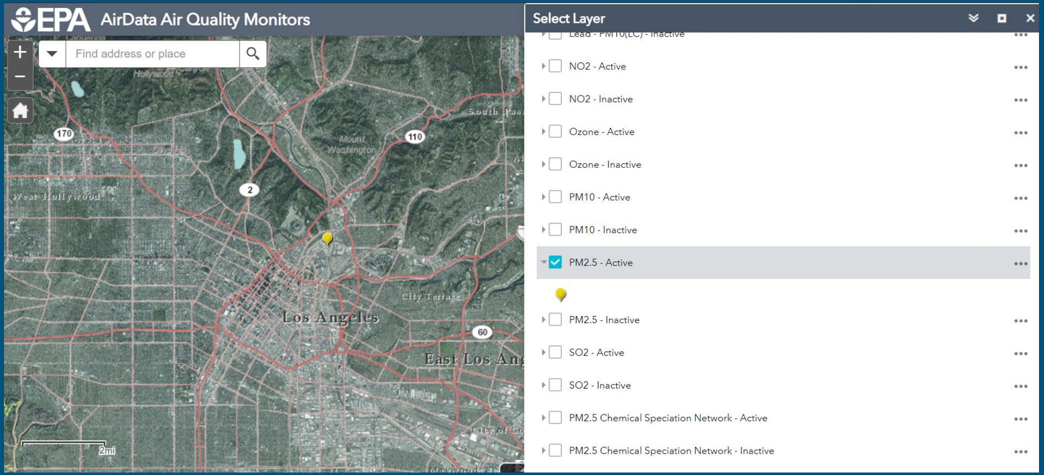
EPA Air Quality Monitor Dataset

- Somewhat of a continuation of my Exploratory Data Analysis project
- Same original dataset, different subset
  - This time, I’m looking at ozone

What is ozone?

- Ozone is an unstable gas comprised of three oxygen atoms
- Vital to our survival in the upper atmosphere (stratosphere) because it absorbs UV radiation from the Sun
- However, in the lower atmosphere where we live (troposphere) ozone is a pollutant that can harm us
    - Damages our cells, causing asthma attacks, eye and lung irritation, heart disease, etc.
    - Also damages materials such as rubber, paints, fibers, etc.
- Tropospheric ozone formation relies on UV light from the sun, nitrogen oxides, and hydrocarbons
- Source: https://scied.ucar.edu/learning-zone/air-quality/ozone-troposphere

Data Source



Select Layer

- [ ] Lead - PM10(LC) - Inactive
- [ ] NO2 - Active
- [ ] NO2 - Inactive
- [ ] Ozone - Active
- [ ] Ozone - Inactive
- [ ] PM10 - Active
- [ ] PM10 - Inactive
- [✓] PM2.5 - Active
- [ ] PM2.5 - Inactive
- [ ] SO2 - Active
- [ ] SO2 - Inactive
- [ ] PM2.5 Chemical Speciation Network - Active
- [ ] PM2.5 Chemical Speciation Network - Inactive

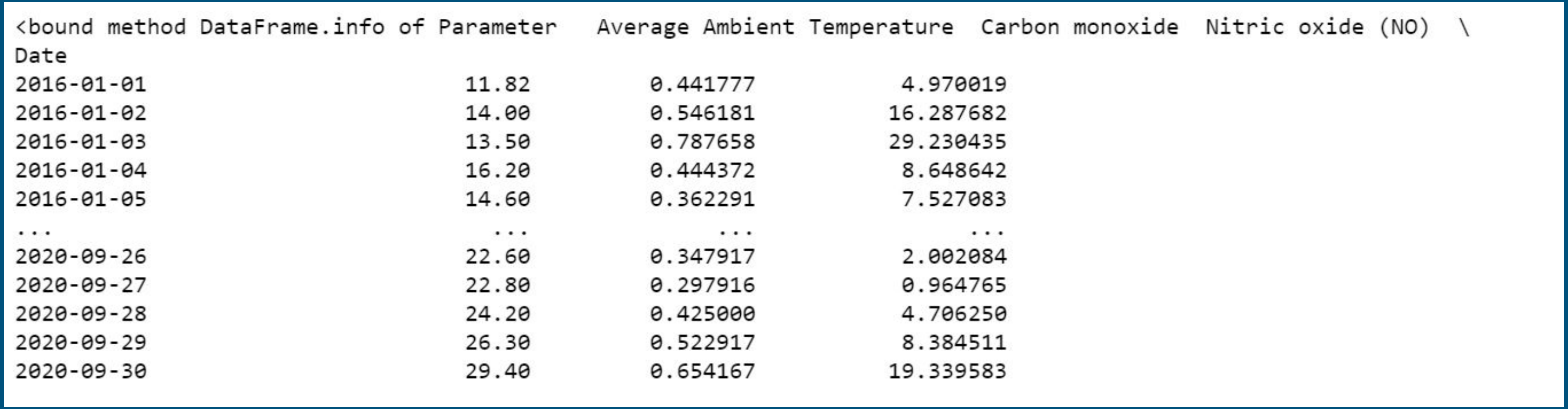
- https://epa.maps.arcgis.com/apps/webappviewer/index.html?id=5f239fd3e72f424f98ef3d5def547eb5&extent=-146.2334,13.1913,-46.3896,56.5319
- Los Angeles-North Main Street Station Monitor
- Looking at PM2.5 Chemical Speciation Network - Active
- Using daily data from 2016-2020

What am I doing in this notebook?

- Goal: predict ozone levels around LA based on other variables such as temperature, nitrogen dioxide, nitrogen oxide, etc.
- I will use linear regression and ensemble methods

After reading in and cleaning the data...

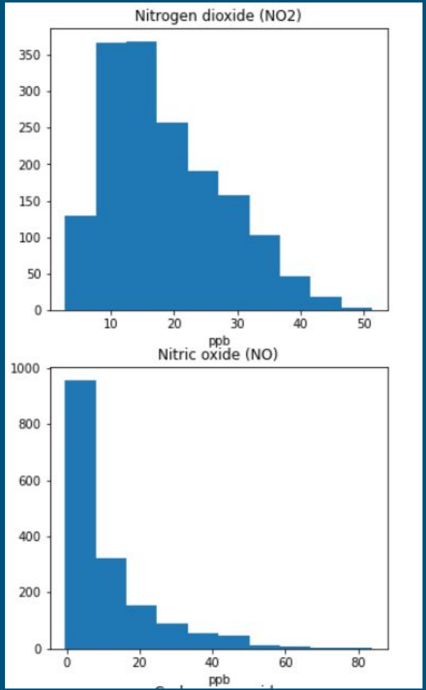
- We see there are 1640 rows and 6 columns
- No null values (dropped)
- A look at part of the data

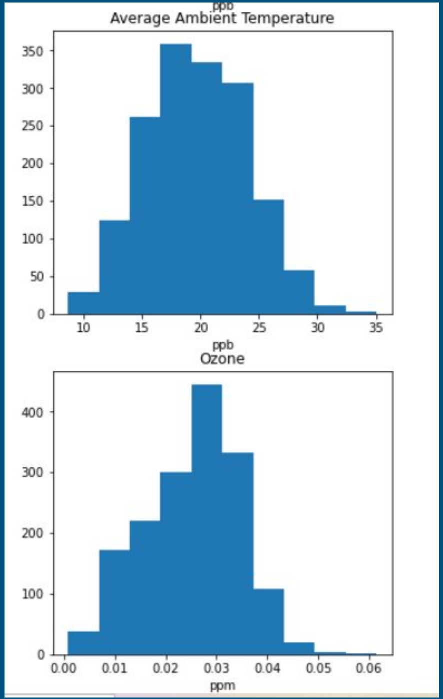


```
<bound method DataFrame.info of Parameter   Average Ambient Temperature  Carbon monoxide  Nitric oxide (NO)  \
Date
2016-01-01                         11.82         0.441777          4.970019
2016-01-02                         14.00         0.546181         16.287682
2016-01-03                         13.50         0.787658         29.230435
2016-01-04                         16.20         0.444372          8.648642
2016-01-05                         14.60         0.362291          7.527083
...                                  ...              ...               ...
2020-09-26                         22.60         0.347917          2.002084
2020-09-27                         22.80         0.297916          0.964765
2020-09-28                         24.20         0.425000          4.706250
2020-09-29                         26.30         0.522917          8.384511
2020-09-30                         29.40         0.654167         19.339583
```

Looking at distributions

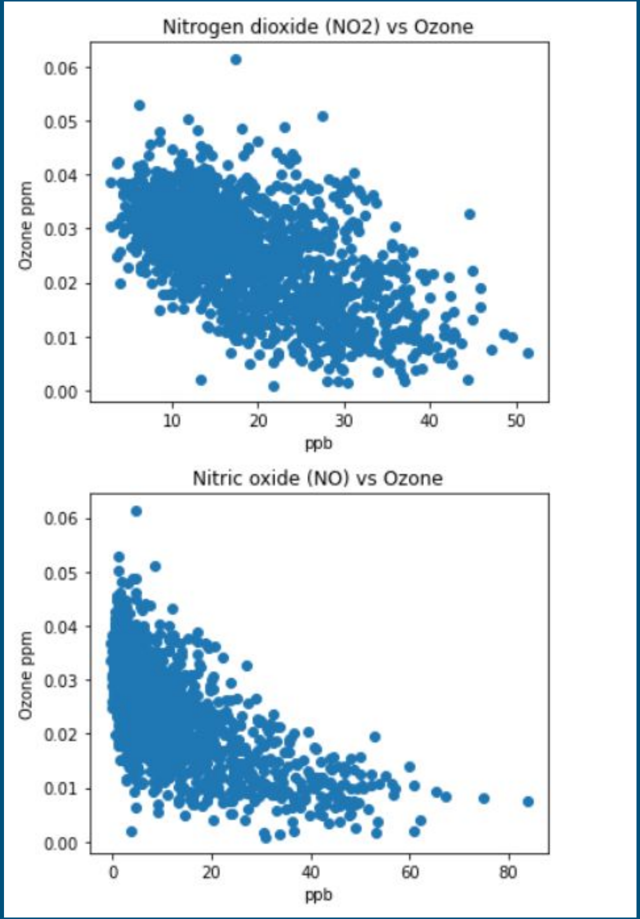
- Next, I created histograms of each variable to look at their distributions
- Most of them had tails to the right, but others were distributed more evenly
- Only four of these are shown to the right





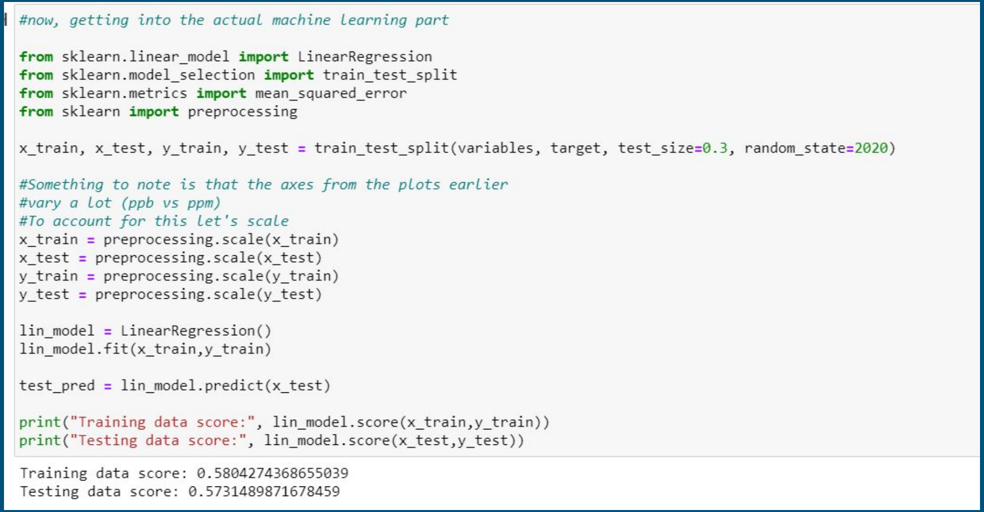
More exploratory analysis

- Next, I graphed ozone levels against each of the other variables
- Some, like nitrogen dioxide, exhibit negative relationships
  - Probably because the nitrogen dioxide is being converted into ozone
- Temperature, on the other hand, exhibits a positive relationship
  - More temperature probably means it's a sunnier day, meaning more UV light to power ozone formation



Attempt #1

- Since ozone levels are continuous, I went for a simple linear regression model



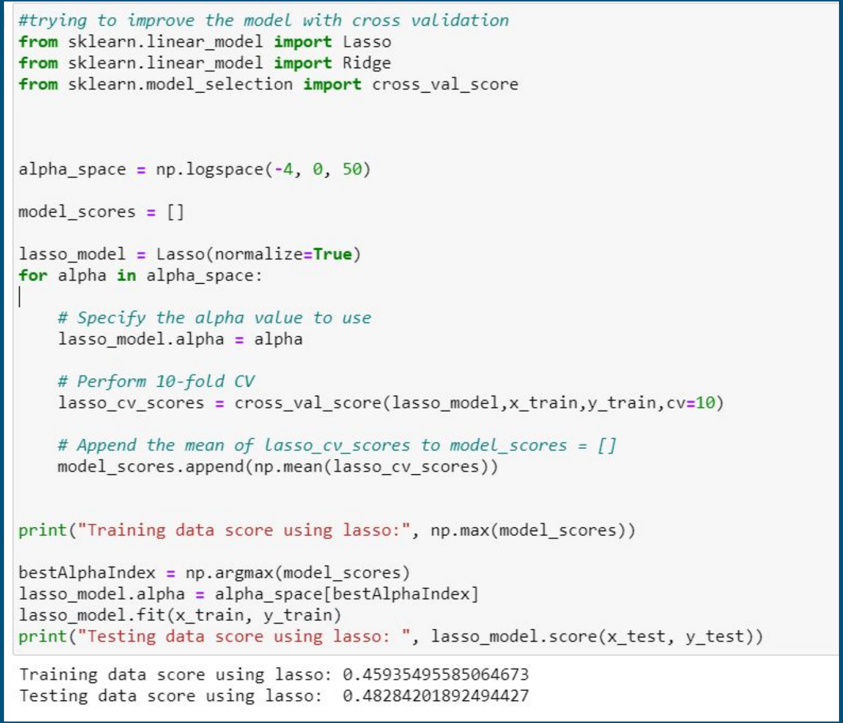
```
#now, getting into the actual machine learning part

from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn import preprocessing

x_train, x_test, y_train, y_test = train_test_split(variables, target, test_size=0.3, random_state=2020)

#Something to note is that the axes from the plots earlier
#vary a lot (ppb vs ppm)
#To account for this let's scale
x_train = preprocessing.scale(x_train)
x_test = preprocessing.scale(x_test)
y_train = preprocessing.scale(y_train)
y_test = preprocessing.scale(y_test)

lin_model = LinearRegression()
lin_model.fit(x_train,y_train)

test_pred = lin_model.predict(x_test)

print("Training data score:", lin_model.score(x_train,y_train))
print("Testing data score:", lin_model.score(x_test,y_test))
```

```
Training data score: 0.5804274368655039
Testing data score: 0.5731489871678459
```

- I split the training and testing sets 70/30, using random_state=2020 to ensure consistency between trials
- The base model, when tested against the testing data, resulted in a score of 0.5731489871678459
- The score is derived from scikit-learn's score() method, which calculates the difference between the model's predicted values and the actual predicted values (in other words, accuracy)

Making a better model

- To increase the performance of the base model I tried simplifying it through cross validation
- I chose selected the optimal alpha by generating a list of potentials and finding the one with the best performance
- First, I used lasso, which resulted in a slightly better score of 0.574463661133027

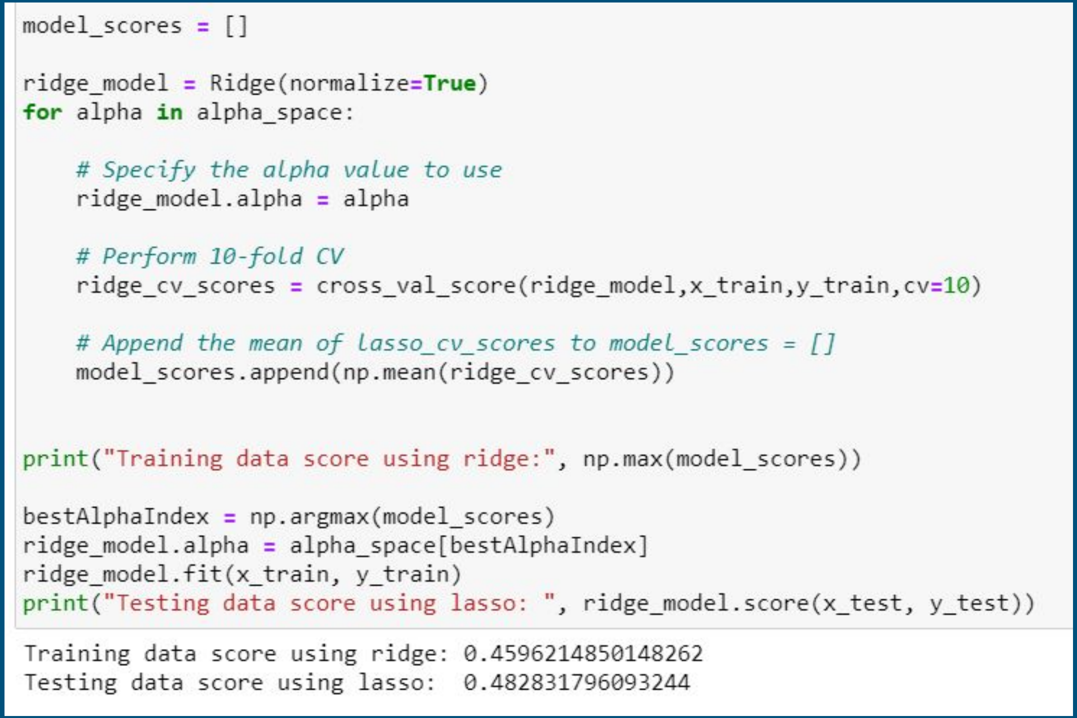


```
#trying to improve the model with cross validation
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score



alpha_space = np.logspace(-4, 0, 50)

model_scores = []

lasso_model = Lasso(normalize=True)
for alpha in alpha_space:

    # Specify the alpha value to use
    lasso_model.alpha = alpha

    # Perform 10-fold CV
    lasso_cv_scores = cross_val_score(lasso_model,x_train,y_train,cv=10)

    # Append the mean of lasso_cv_scores to model_scores = []
    model_scores.append(np.mean(lasso_cv_scores))


print("Training data score using lasso:", np.max(model_scores))

bestAlphaIndex = np.argmax(model_scores)
lasso_model.alpha = alpha_space[bestAlphaIndex]
lasso_model.fit(x_train, y_train)
print("Testing data score using lasso: ", lasso_model.score(x_test, y_test))
```

```
Training data score using lasso: 0.45935495585064673
Testing data score using lasso:  0.48284201892494427
```

Making a better model cont.

- Next, I tried using ridge
- Similarly to lasso, I generated a range of alpha values to find the optimal one
- Ridge performed slightly worse than lasso on the testing set, with a score of 0.5737411667807468

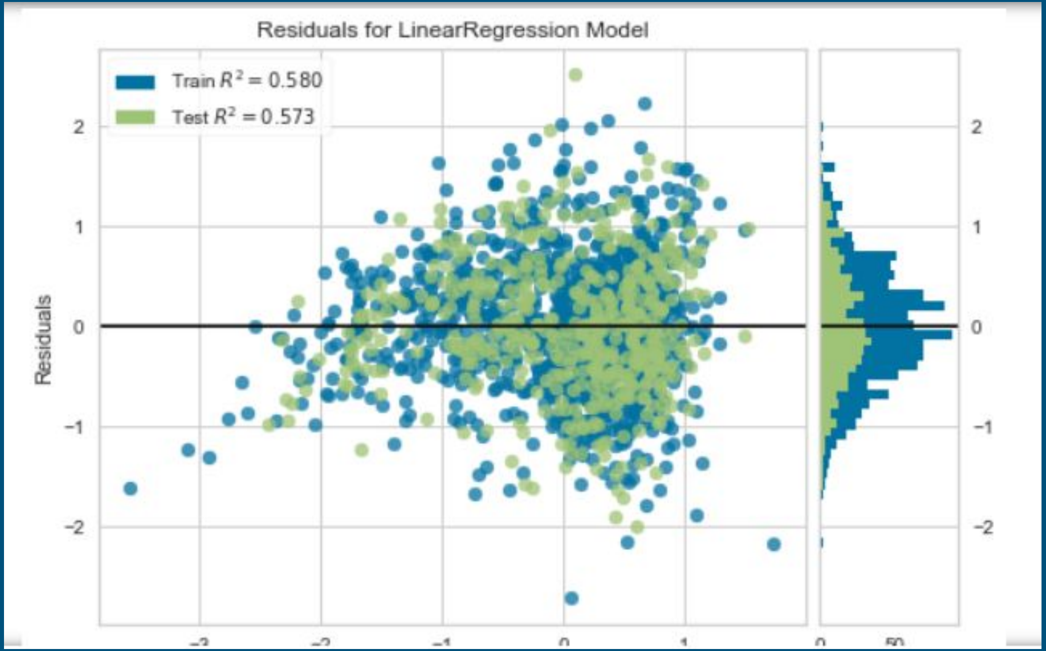


```
model_scores = []

ridge_model = Ridge(normalize=True)
for alpha in alpha_space:

    # Specify the alpha value to use
    ridge_model.alpha = alpha

    # Perform 10-fold CV
    ridge_cv_scores = cross_val_score(ridge_model,x_train,y_train,cv=10)

    # Append the mean of lasso_cv_scores to model_scores = []
    model_scores.append(np.mean(ridge_cv_scores))


print("Training data score using ridge:", np.max(model_scores))

bestAlphaIndex = np.argmax(model_scores)
ridge_model.alpha = alpha_space[bestAlphaIndex]
ridge_model.fit(x_train, y_train)
print("Testing data score using lasso: ", ridge_model.score(x_test, y_test))
```

```
Training data score using ridge: 0.4596214850148262
Testing data score using lasso:  0.482831796093244
```

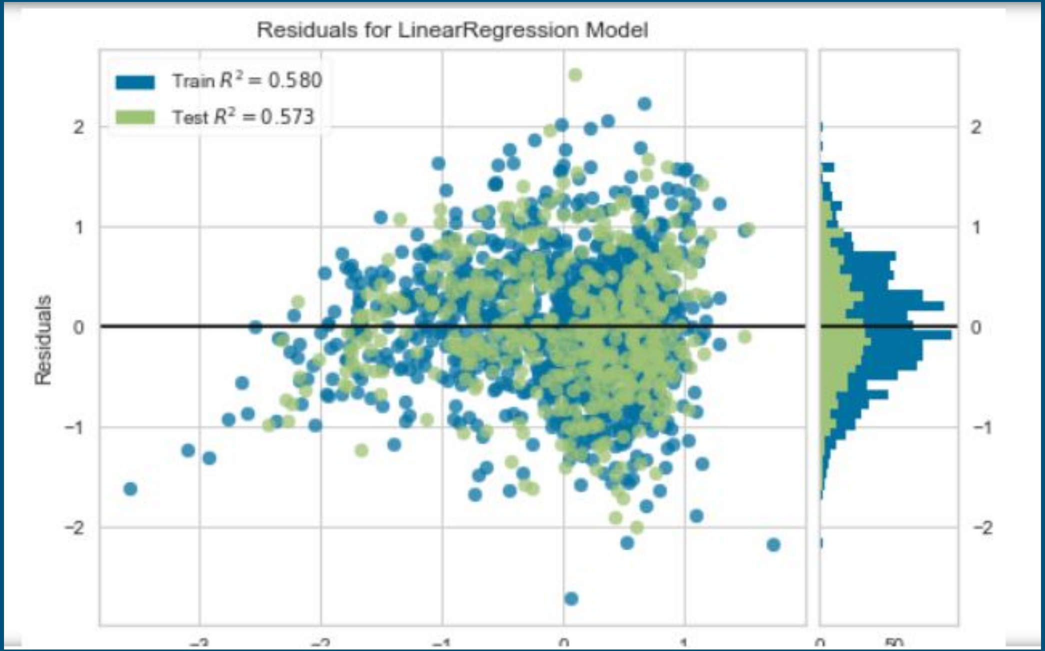
Making some checks

- I plotted the residuals to make sure everything looks okay
- There is a slight curve to the residuals
- Additionally, the data points show a trumpet like pattern
- However, points do appear to be normally distributed



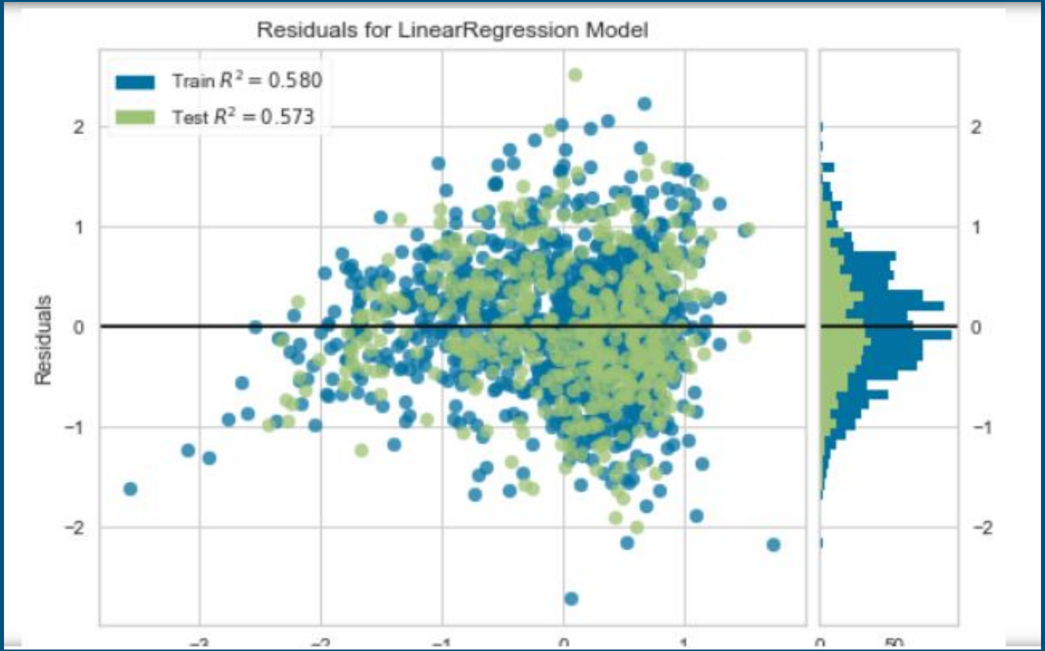
Making some checks cont.

- The trumpet shape indicates that the model might need additional predictor variables to explain the levels of ozone
- These additional variables probably include the hydrocarbons mentioned earlier, which aren't in the EPA dataset



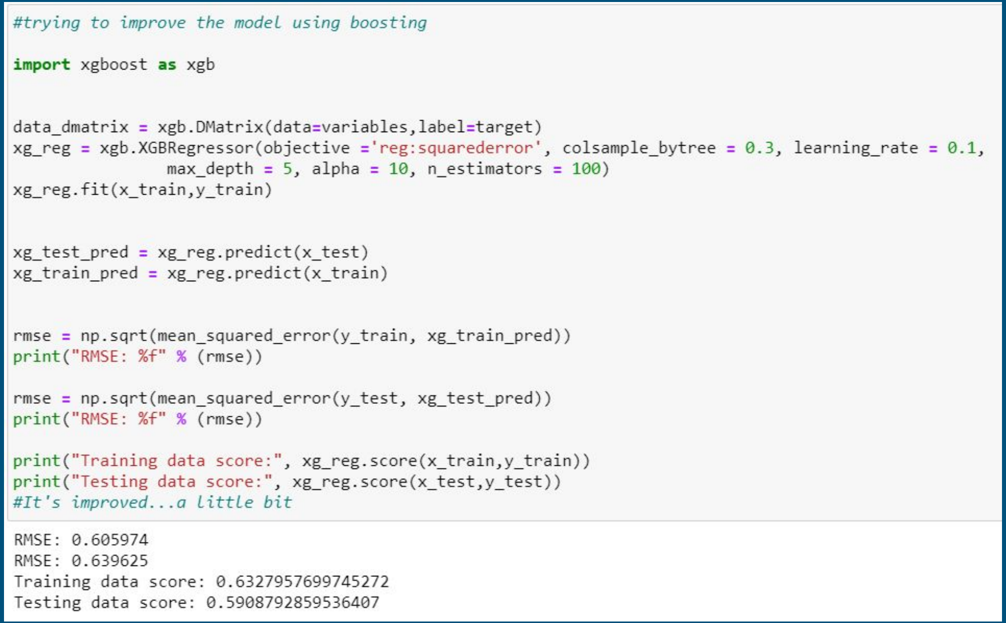
Making some check cont.

- Unfortunately, high quality, consistent air monitoring data is difficult to come by, so I'm going to keep using this data with the acknowledgement that this provides an incomplete picture



Increasing the models' performance

- Next, I tried XGBoost, an implementation of gradient boosted decision trees
- Did considerably better with a score of 0.5908792859536407

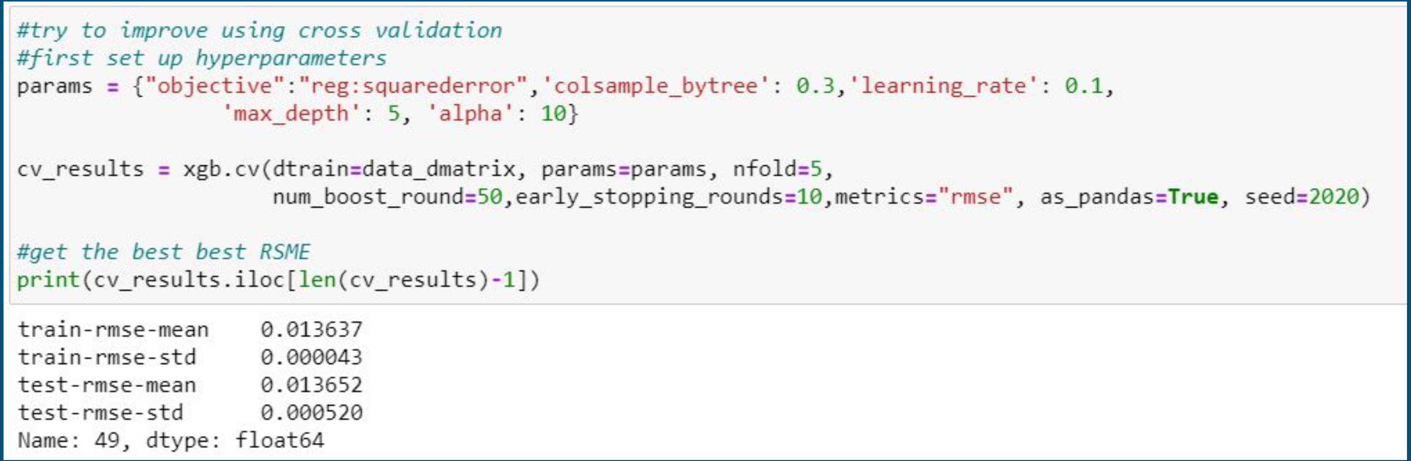


```
#trying to improve the model using boosting

import xgboost as xgb


data_dmatrix = xgb.DMatrix(data=variables,label=target)
xg_reg = xgb.XGBRegressor(objective ='reg:squarederror', colsample_bytree = 0.3, learning_rate = 0.1,
                max_depth = 5, alpha = 10, n_estimators = 100)
xg_reg.fit(x_train,y_train)


xg_test_pred = xg_reg.predict(x_test)
xg_train_pred = xg_reg.predict(x_train)


rmse = np.sqrt(mean_squared_error(y_train, xg_train_pred))
print("RMSE: %f" % (rmse))

rmse = np.sqrt(mean_squared_error(y_test, xg_test_pred))
print("RMSE: %f" % (rmse))

print("Training data score:", xg_reg.score(x_train,y_train))
print("Testing data score:", xg_reg.score(x_test,y_test))
#It's improved...a little bit
```

```
RMSE: 0.605974
RMSE: 0.639625
Training data score: 0.6327957699745272
Testing data score: 0.5908792859536407
```

XGBoost Cross Validation

- For cross validation, I set the metric to RMSE in the interim as a stand in to score(), which I was using previously
- RMSE of base XGBoost model on the testing data set was 0.639625
- Cross validation significantly improved this to 0.013637

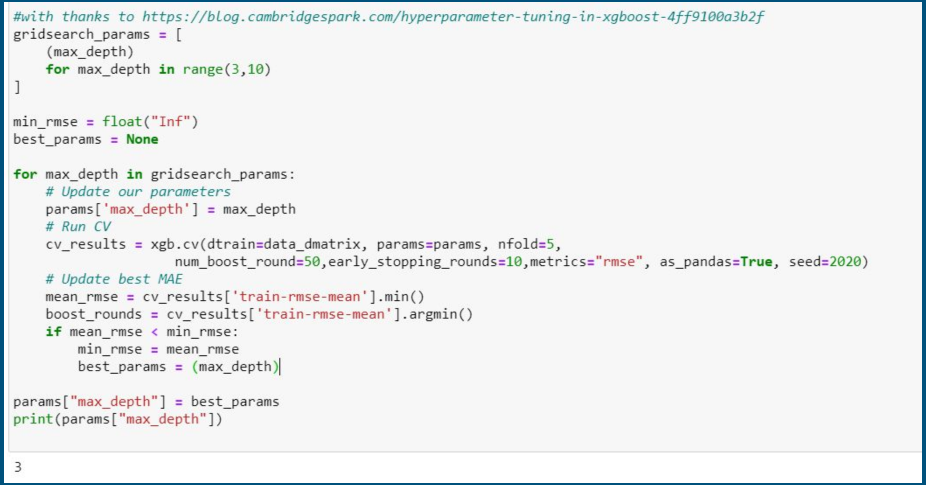


```
#try to improve using cross validation
#first set up hyperparameters
params = {"objective":"reg:squarederror",'colsample_bytree': 0.3,'learning_rate': 0.1,
                'max_depth': 5, 'alpha': 10}

cv_results = xgb.cv(dtrain=data_dmatrix, params=params, nfold=5,
                    num_boost_round=50,early_stopping_rounds=10,metrics="rmse", as_pandas=True, seed=2020)

#get the best best RSME
print(cv_results.iloc[len(cv_results)-1])
```

```
train-rmse-mean    0.013637
train-rmse-std     0.000043
test-rmse-mean     0.013652
test-rmse-std      0.000520
Name: 49, dtype: float64
```

XGBoost Hyperparameter Tuning

- To make the XGBoost cross validation model better, I try to find the optimal hyperparameters
- Modified code from https://blog.cambridgespark.com/hyperparameter-tuning-in-xgboost-4ff9100a3b2f
- Started off with selecting an optimal max_depth from the range of 3 to 9, inclusive

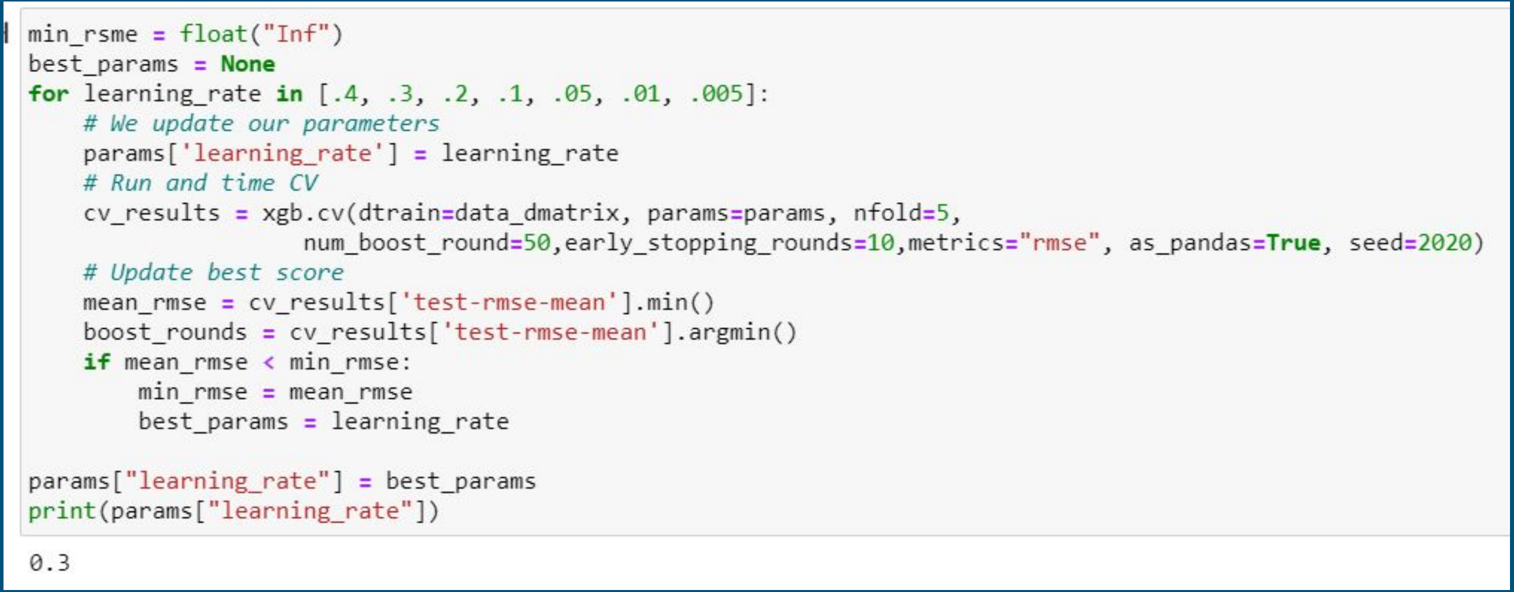


```
#with thanks to https://blog.cambridgespark.com/hyperparameter-tuning-in-xgboost-4ff9100a3b2f
gridsearch_params = [
    (max_depth)
    for max_depth in range(3,10)
]

min_rmse = float("Inf")
best_params = None

for max_depth in gridsearch_params:
    # Update our parameters
    params['max_depth'] = max_depth
    # Run CV
    cv_results = xgb.cv(dtrain=data_dmatrix, params=params, nfold=5,
                num_boost_round=50,early_stopping_rounds=10,metrics="rmse", as_pandas=True, seed=2020)
    # Update best MAE
    mean_rmse = cv_results['train-rmse-mean'].min()
    boost_rounds = cv_results['train-rmse-mean'].argmin()
    if mean_rmse < min_rmse:
        min_rmse = mean_rmse
        best_params = (max_depth)

params["max_depth"] = best_params
print(params["max_depth"])
```

```
3
```

Hyperparameter tuning cont.

- Next was learning_rate

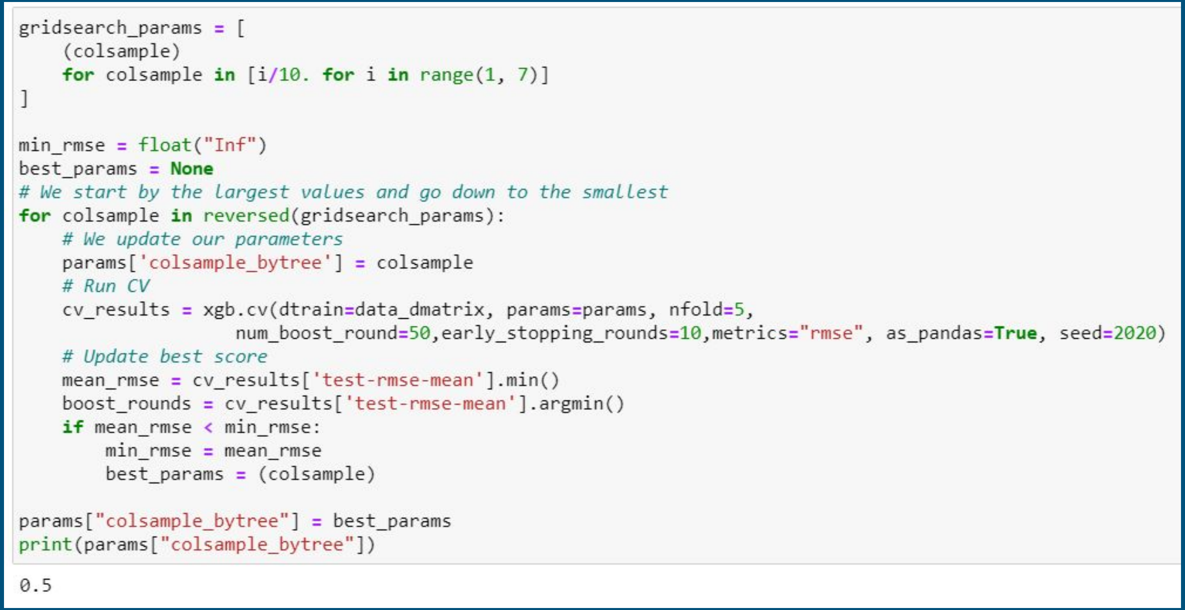


```
min_rsme = float("Inf")
best_params = None
for learning_rate in [.4, .3, .2, .1, .05, .01, .005]:
    # We update our parameters
    params['learning_rate'] = learning_rate
    # Run and time CV
    cv_results = xgb.cv(dtrain=data_dmatrix, params=params, nfold=5,
                        num_boost_round=50,early_stopping_rounds=10,metrics="rmse", as_pandas=True, seed=2020)
    # Update best score
    mean_rmse = cv_results['test-rmse-mean'].min()
    boost_rounds = cv_results['test-rmse-mean'].argmin()
    if mean_rmse < min_rmse:
        min_rmse = mean_rmse
        best_params = learning_rate

params["learning_rate"] = best_params
print(params["learning_rate"])
```

```
0.3
```

Hyperparameter tuning cont.

- Third was colsample_bytree from a range of 0.1 to 0.6, inclusive, with steps of 0.1

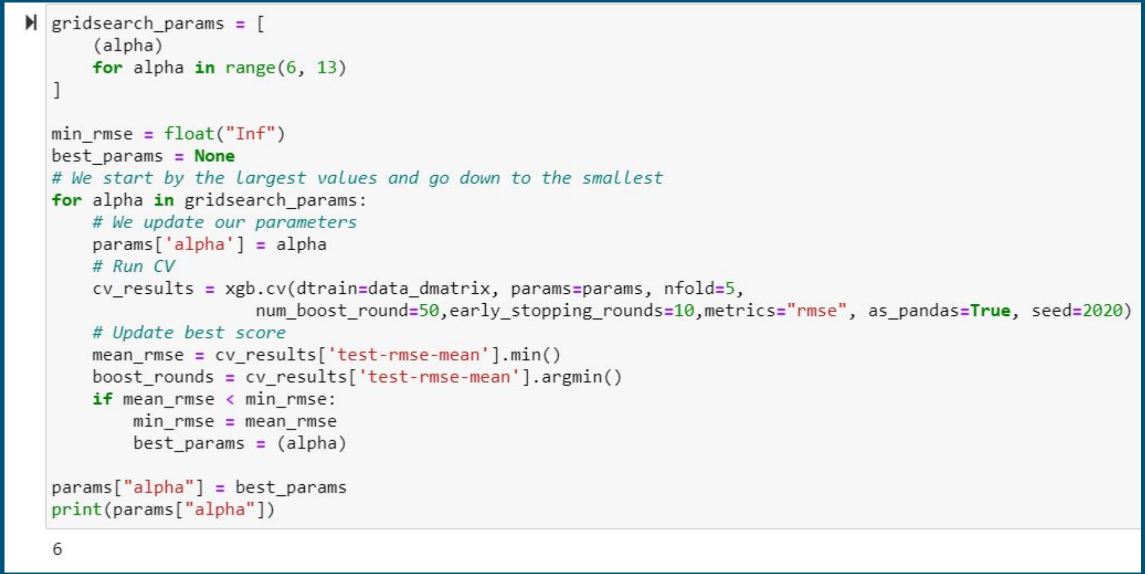


```
gridsearch_params = [
    (colsample)
    for colsample in [i/10. for i in range(1, 7)]
]

min_rmse = float("Inf")
best_params = None
# We start by the largest values and go down to the smallest
for colsample in reversed(gridsearch_params):
    # We update our parameters
    params['colsample_bytree'] = colsample
    # Run CV
    cv_results = xgb.cv(dtrain=data_dmatrix, params=params, nfold=5,
                num_boost_round=50,early_stopping_rounds=10,metrics="rmse", as_pandas=True, seed=2020)
    # Update best score
    mean_rmse = cv_results['test-rmse-mean'].min()
    boost_rounds = cv_results['test-rmse-mean'].argmin()
    if mean_rmse < min_rmse:
        min_rmse = mean_rmse
        best_params = (colsample)

params["colsample_bytree"] = best_params
print(params["colsample_bytree"])
```

```
0.5
```

Hyperparameter tuning cont.

- Finally was adjusting alpha, selecting an integer between 7 and 13, inclusive

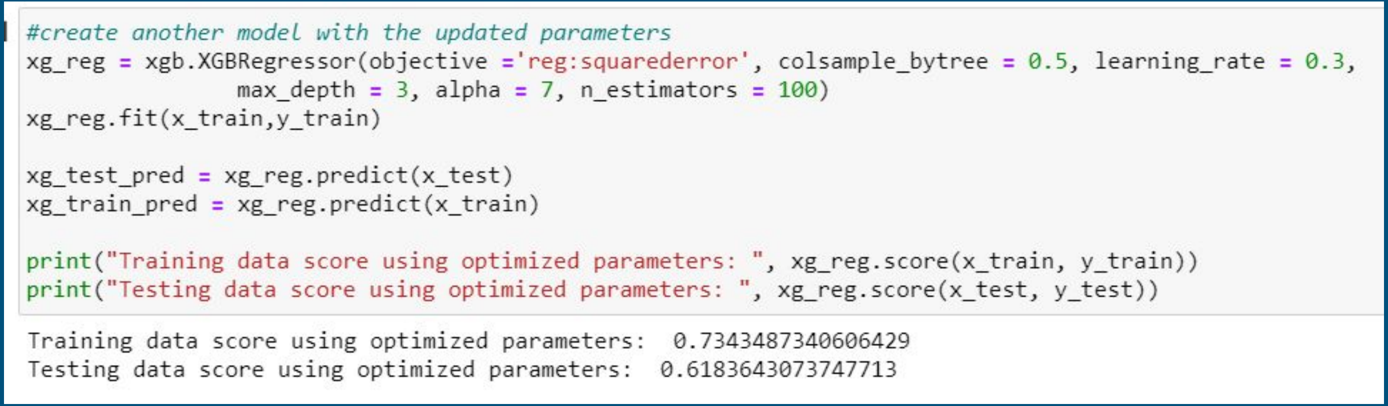


```
gridsearch_params = [
    (alpha)
    for alpha in range(6, 13)
]

min_rmse = float("Inf")
best_params = None
# We start by the largest values and go down to the smallest
for alpha in gridsearch_params:
    # We update our parameters
    params['alpha'] = alpha
    # Run CV
    cv_results = xgb.cv(dtrain=data_dmatrix, params=params, nfold=5,
                        num_boost_round=50,early_stopping_rounds=10,metrics="rmse", as_pandas=True, seed=2020)
    # Update best score
    mean_rmse = cv_results['test-rmse-mean'].min()
    boost_rounds = cv_results['test-rmse-mean'].argmin()
    if mean_rmse < min_rmse:
        min_rmse = mean_rmse
        best_params = (alpha)

params["alpha"] = best_params
print(params["alpha"])
```

```
6
```

Hyperparameter tuning cont.

- The optimal hyperparameters based on my ranges are:
  - Colsample_bytree = 0.5
  - Learning_rate = 0.3
  - Max_depth = 3
  - Alpha = 6

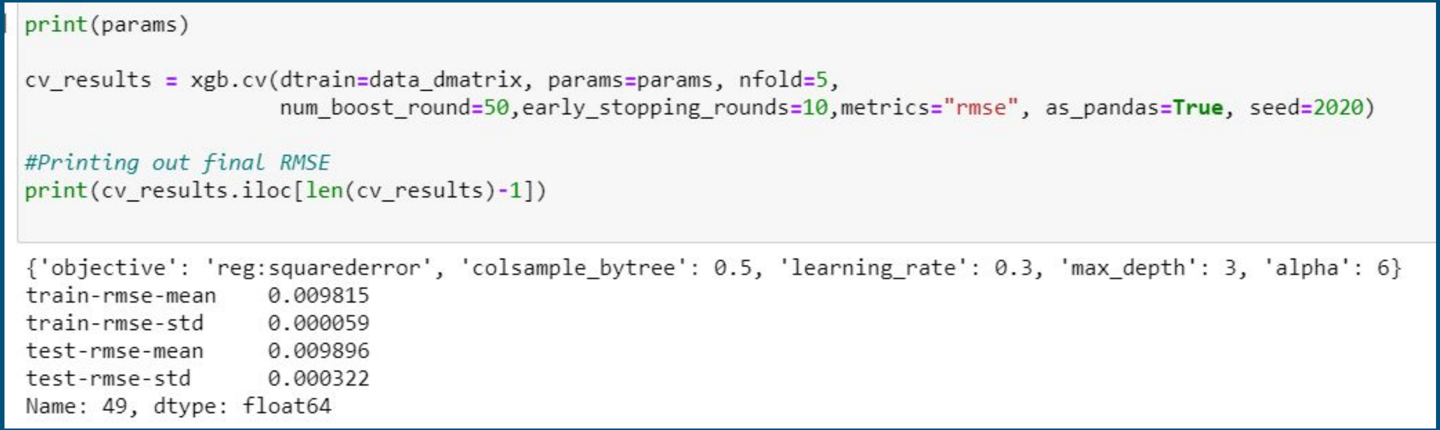


```
#create another model with the updated parameters
xg_reg = xgb.XGBRegressor(objective ='reg:squarederror', colsample_bytree = 0.5, learning_rate = 0.3,
                max_depth = 3, alpha = 7, n_estimators = 100)
xg_reg.fit(x_train,y_train)

xg_test_pred = xg_reg.predict(x_test)
xg_train_pred = xg_reg.predict(x_train)

print("Training data score using optimized parameters: ", xg_reg.score(x_train, y_train))
print("Testing data score using optimized parameters: ", xg_reg.score(x_test, y_test))
```

```
Training data score using optimized parameters:  0.7343487340606429
Testing data score using optimized parameters:  0.6183643073747713
```

Hyperparameter tuning results

- This yields a RMSE of 0.009815, which is much lower than what was previously achieved

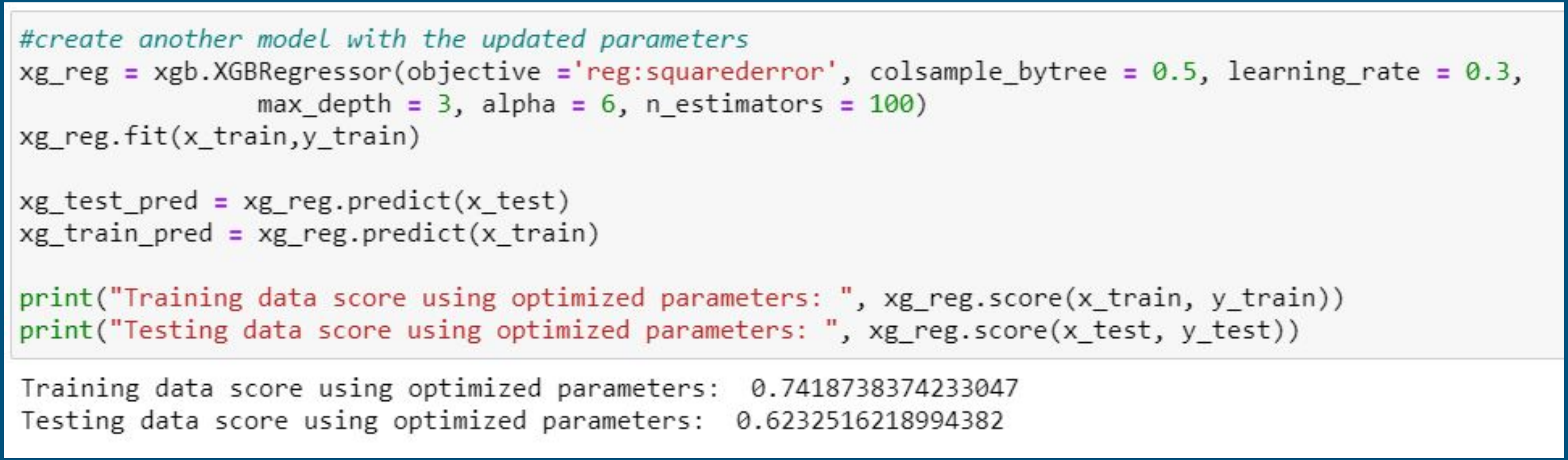


```
print(params)

cv_results = xgb.cv(dtrain=data_dmatrix, params=params, nfold=5,
                    num_boost_round=50,early_stopping_rounds=10,metrics="rmse", as_pandas=True, seed=2020)

#Printing out final RMSE
print(cv_results.iloc[len(cv_results)-1])
```

```
{'objective': 'reg:squarederror', 'colsample_bytree': 0.5, 'learning_rate': 0.3, 'max_depth': 3, 'alpha': 6}
train-rmse-mean    0.009815
train-rmse-std     0.000059
test-rmse-mean     0.009896
test-rmse-std      0.000322
Name: 49, dtype: float64
```

Hyperparameter tuning results cont.

- But what does this translate to in terms of score()?
  - A much improved 0.6232516218994382



```
#create another model with the updated parameters
xg_reg = xgb.XGBRegressor(objective ='reg:squarederror', colsample_bytree = 0.5, learning_rate = 0.3,
                max_depth = 3, alpha = 6, n_estimators = 100)
xg_reg.fit(x_train,y_train)

xg_test_pred = xg_reg.predict(x_test)
xg_train_pred = xg_reg.predict(x_train)

print("Training data score using optimized parameters: ", xg_reg.score(x_train, y_train))
print("Testing data score using optimized parameters: ", xg_reg.score(x_test, y_test))
```

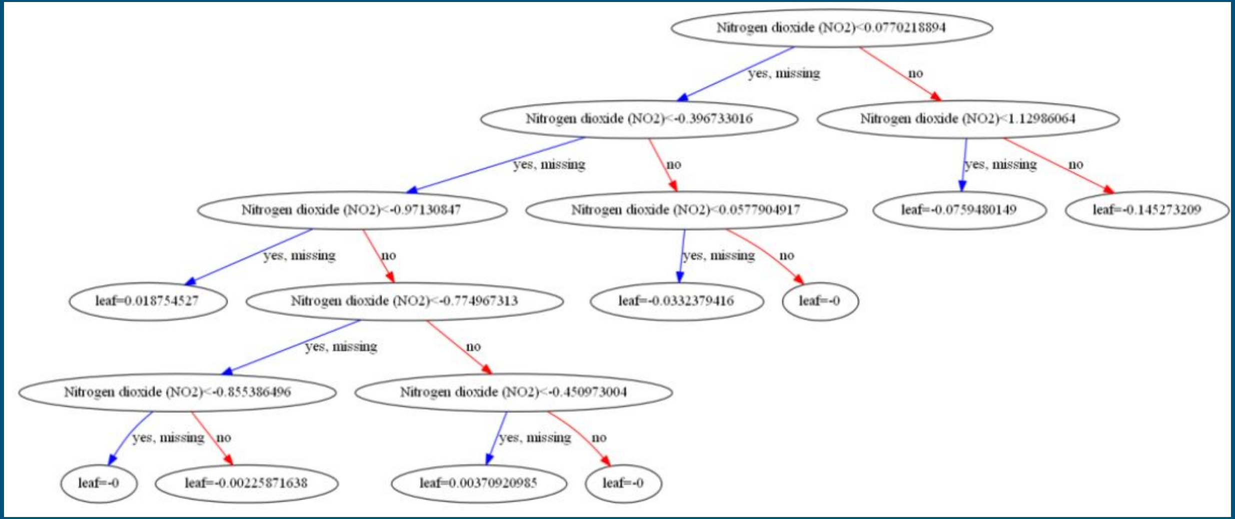
```
Training data score using optimized parameters:  0.7418738374233047
Testing data score using optimized parameters:  0.6232516218994382
```

Which model is the best?

- The XGBoost model with optimized hyperparameters
- Still only ~60% accuracy, but that’s okay considering the lack of inclusion of other relevant predictors
    - An accuracy of 60% is still a marked improvement over the base linear regression model

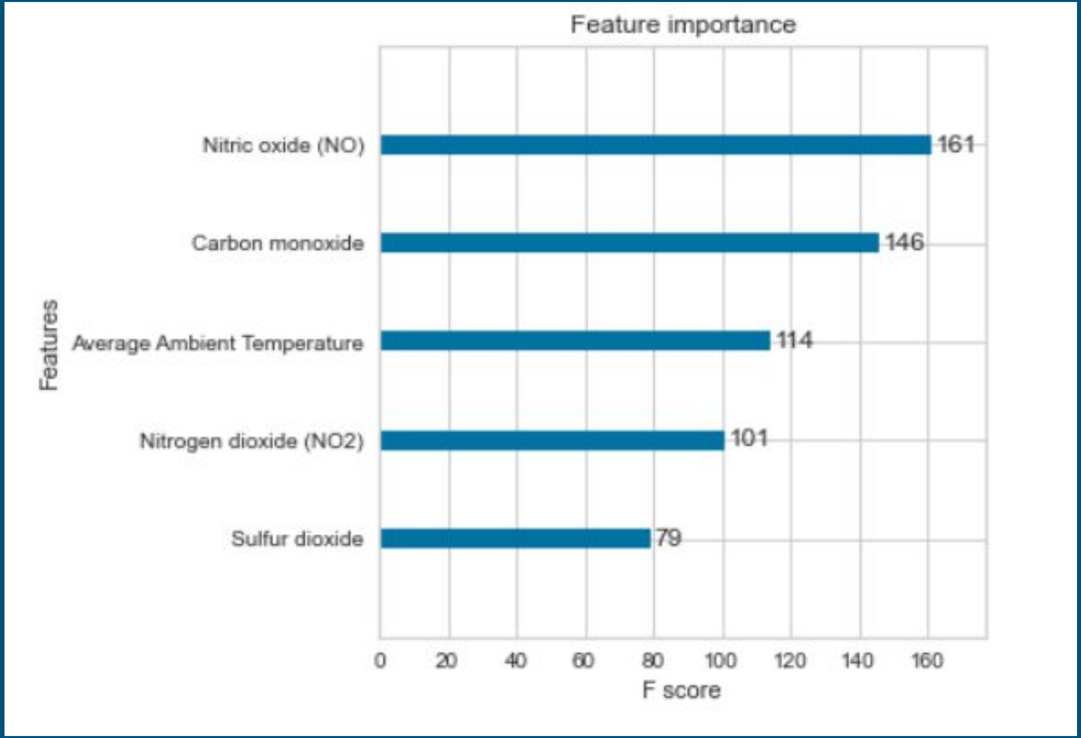
Visualizing the best model’s tree

- Here we can see the decision tree the final XGBoost model uses


Nitrogen dioxide (NO2)<0.0770218894
yes, missing
no
Nitrogen dioxide (NO2)<-0.396733016
Nitrogen dioxide (NO2)<1.12986064
yes, missing
no
yes, missing
no
Nitrogen dioxide (NO2)<-0.97130847
Nitrogen dioxide (NO2)<0.0577904917
leaf=-0.0759480149
leaf=-0.145273209
yes, missing
no
yes, missing
no
leaf=0.018754527
Nitrogen dioxide (NO2)<-0.774967313
leaf=-0.0332379416
leaf=-0
yes, missing
no
Nitrogen dioxide (NO2)<-0.855386496
Nitrogen dioxide (NO2)<-0.450973004
yes, missing
no
yes, missing
no
leaf=-0
leaf=-0.00225871638
leaf=0.00370920985
leaf=-0

Ranking the importance of predictors

- And here's the importance of each predictor


Feature importance
Features
Nitric oxide (NO)
Carbon monoxide
Average Ambient Temperature
Nitrogen dioxide (NO2)
Sulfur dioxide
F score