Linear & Non-linear Approach of GAM

Research for GAM/ERG Model Team
By Member - Shashvat Joshi

Redback Operations

Introduction

In real world all the linear models are not perfectly linear. A generalized additive model (GAM) is a generalized linear model in which the linear response variable depends linearly on unknown smooth functions of some predictor variables, and interest focuses on inference about these smooth functions.

The key difference of GAM & Generalized Linear Models such as Linear Regression is GAM is allowed to learn non-linear features. In GAM output can be modelled by a sum of arbitrary functions of each feature.

Hence if any model is showing nonlinear relationship between predictors and target, we can apply GAM to build a model which will utilize non linearity and try to predict the best fit model

Dataset

We will use the kaggle Cycling VO2 data from :

https://www.kaggle.com/stpeteishii/cycling-vo2-with-autoviz/data

The data has been collected from 7 cyclists (called subject). Each cyclist has gone through two different protocol tests – protocol-1 & protocol-2 before going through Wingate Test. The incremental test results are also captured in a separate dataset.

The data includes power output (P), respiratory frequency (Rf), heart rate (HR),cadence (ω). Features are RF, HR, w.

Import required libraries

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Load data and read data into pandas data frame

Load the data from Cyclist-1 or Subject-1's exercise trials

In [22]:
```
# reading data files

df1 = pd.read_csv('sbj_1_I.csv')
df2 = pd.read_csv('sbj_1_II.csv')
df3 = pd.read_csv('sbj_1_Wingate.csv')
df4 = pd.read_csv('sbj_1_incremental.csv')
```

4 datasets of subject-1 are converted to 4 dataframes – df1, df2, df3, df4.

- We started the EDA and explored the dataset as below.

Data understanding

In [3]:
```
# Look at first few records of the dataset
print(df1.head())
```

```
   time  Power      Oxygen  Cadence         HR         RF
0     1    0.0  318.400000        0  75.600000  20.100000
1     2    0.0  356.166667        0  75.666667  19.750000
2     3    0.0  403.285714        0  75.714286  19.428571
3     4    0.0  456.250000        0  75.750000  19.125000
4     5    0.0  478.925926        0  75.740741  18.962963
```

In [4]:
```
print(df2.head())
```

```
   time  Power      Oxygen  Cadence         HR         RF
0     2    0.0  454.500000      0.0  69.600000  26.300000
1     3    0.0  501.583333      0.0  69.500000  25.083333
2     4    0.0  524.261905      0.0  69.523810  24.166667
3     5    0.0  531.687500      0.0  69.625000  23.437500
4     6    0.0  528.944444      0.0  69.777778  22.833333
```

In [5]:
```
print(df3.head())
```

```
   time  Power      Oxygen  Cadence         HR         RF
0     2    0.0  329.533333      0.0  82.333333  18.400000
1     3    0.0  345.333333      0.0  82.000000  18.666667
2     4    0.0  361.000000      0.0  81.571429  18.857143
3     5    0.0  387.312500      0.0  81.187500  19.312500
4     6    0.0  420.722222      0.0  80.833333  19.944444
```

In [6]:
```
print(df4.head())
```

```
   time  Power    Oxygen  Cadence    HR    RF
0     3    0.0  602.0000      0.0  86.0  16.0
1     4    0.0  578.1250      0.0  86.0  16.0
2     5    0.0  558.7500      0.0  86.0  16.0
3     6    0.0  542.1875      0.0  86.0  16.0
4     7    0.0  527.5000      0.0  86.0  16.0
```

In [7]:
```
# inspect the structure etc.
print(df1.info(), "\n")
print(df1.shape)
print("***********\n")

print(df2.info(), "\n")
print(df2.shape)
print("***********\n")

print(df3.info(), "\n")
print(df3.shape)
print("***********\n")

print(df4.info(), "\n")
print(df4.shape)
print("***********\n")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2819 entries, 0 to 2818
Data columns (total 6 columns):
```

```
Data columns (total 6 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   time     2819 non-null   int64  
 1   Power    2819 non-null   float64
 2   Oxygen   2819 non-null   float64
 3   Cadence  2819 non-null   int64  
 4   HR       2819 non-null   float64
 5   RF       2819 non-null   float64
dtypes: float64(4), int64(2)
memory usage: 132.3 KB
None

(2819, 6)
***********
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2579 entries, 0 to 2578
Data columns (total 6 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   time     2579 non-null   int64  
 1   Power    2579 non-null   float64
 2   Oxygen   2579 non-null   float64
 3   Cadence  2579 non-null   float64
 4   HR       2579 non-null   float64
 5   RF       2579 non-null   float64
dtypes: float64(5), int64(1)
memory usage: 121.0 KB
None

(2579, 6)
***********
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 751 entries, 0 to 750
Data columns (total 6 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   time     751 non-null    int64
 1   Power    751 non-null    float64
 2   Oxygen   751 non-null    float64
 3   Cadence  751 non-null    float64
 4   HR       751 non-null    float64
 5   RF       751 non-null    float64
dtypes: float64(5), int64(1)
memory usage: 35.3 KB
None

(751, 6)
***********
```

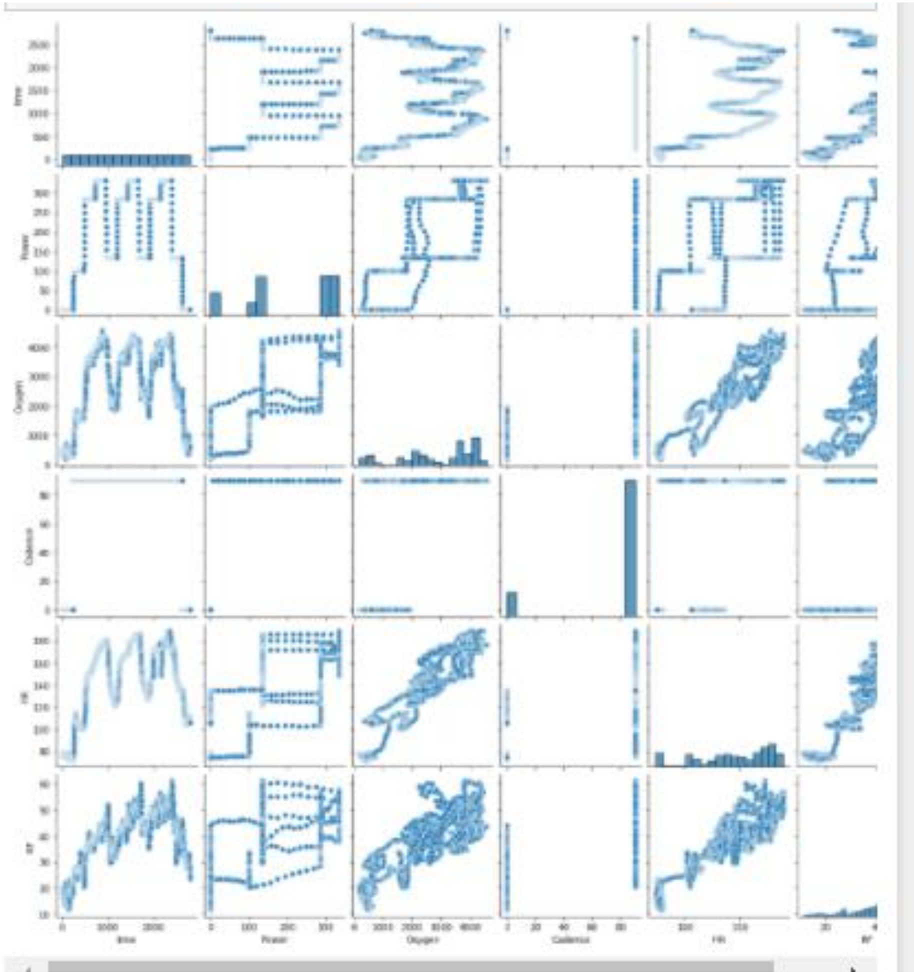
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2831 entries, 0 to 2830
Data columns (total 6 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   time     2831 non-null   int64
 1   Power    2831 non-null   float64
 2   Oxygen   2831 non-null   float64
 3   Cadence  2831 non-null   float64
 4   HR       2831 non-null   float64
 5   RF       2831 non-null   float64
dtypes: float64(5), int64(1)
memory usage: 132.8 KB
None

(2831, 6)
***********
```

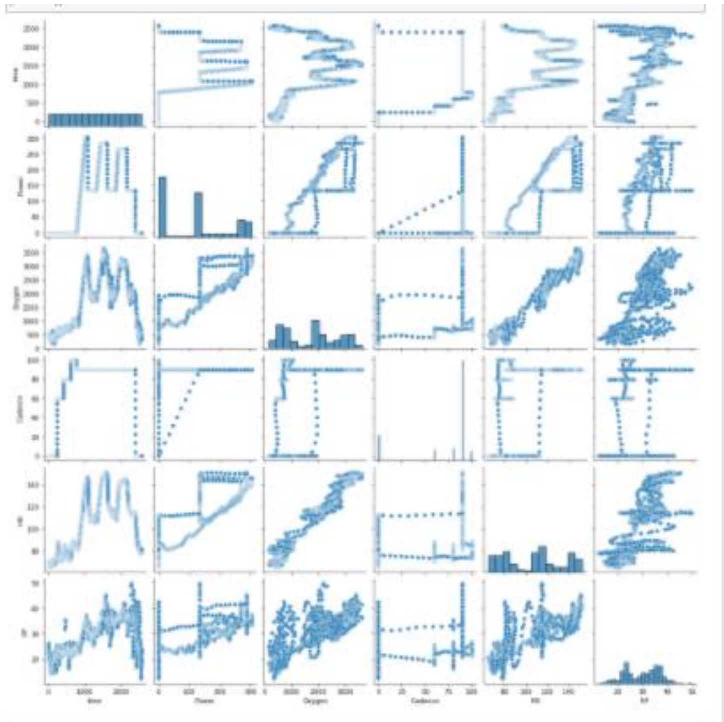
Visualization with Pairplot

Pairplot to visualize the feature correlations in subject-1 dataset for trtal-1, trial-2, wingate & incremental training datasets.

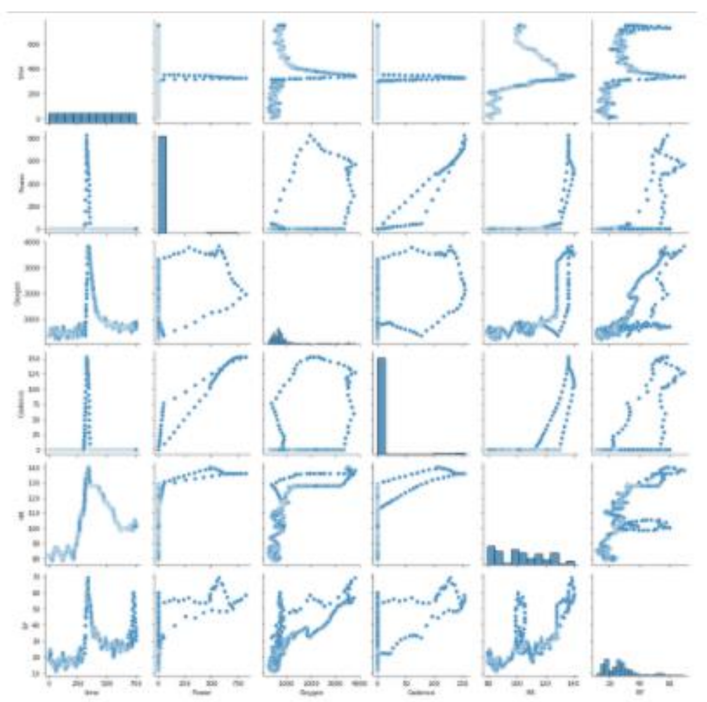
```python
# Pairwise scatter plot between any two features
sns.pairplot(df1)
plt.show()
```



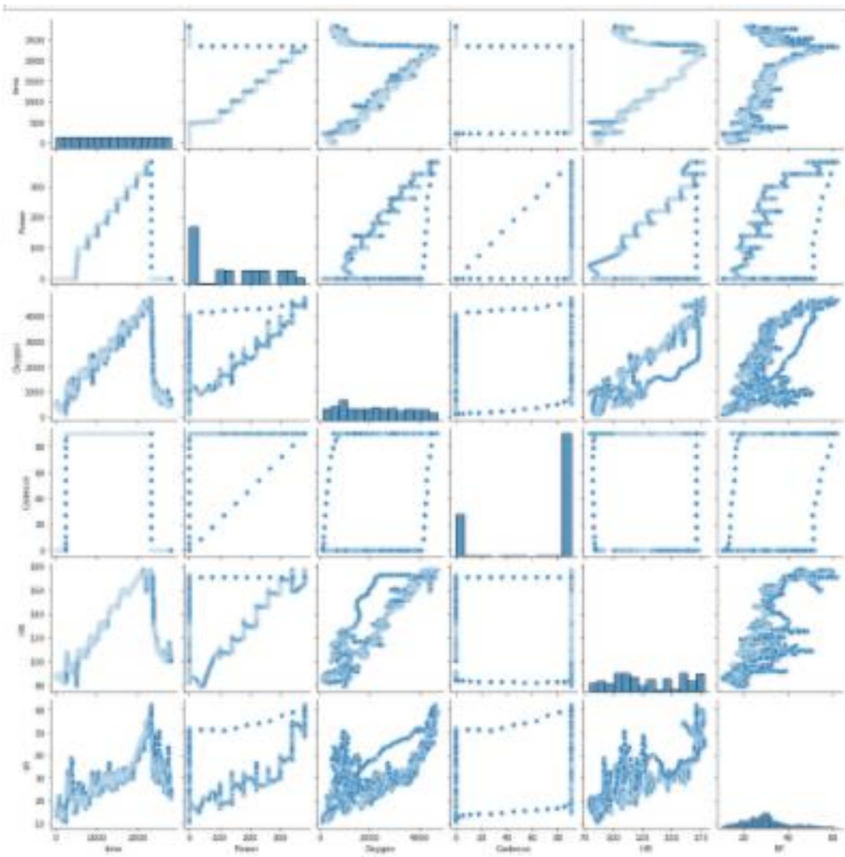
```
# Pairwise scatter plot between any two features
sns.pairplot(df2)
plt.show()
```



```
# Pairwise scatter plot between any two features
sns.pairplot(df3)
plt.show()
```



```
# Pairwise scatter plot between any two features
sns.pairplot(df4)
plt.show()
```



Clearly the above plots between Target variable “Oxygen” and other predictors are not linear.

In Linear Regression the relationship between target & predictors must be a simple weighted sum. GAMs avoids this limitations. In GAM we simply replace beta coefficients with a flexible function where we can use higher order coefficients which allows nonlinear relationships.

This flexible function is called a spline. Splines are complex functions that allow us to model non-linear relationships for each feature. The sum of many splines forms a GAM. The result is a highly flexible model which still has some of the Explanability of a linear regression.

We will implement some mathematical higher order nonlinear algorithms next week.

Exploring GAM

This week we will use the same oxygen uptake dataset used last week and explore GAM with python code.

Linear Regression

We are building a Linear Regression Model for Oxygen as Output and Power as input.

Imports

```
!pip install plotly
```

```
Requirement already satisfied: plotly in c:\users\admin\anaconda3\lib\site-packages (5.7.0)
Requirement already satisfied: six in c:\users\admin\anaconda3\lib\site-packages (from plotly) (1.16.
0)
Requirement already satisfied: tenacity>=6.2.0 in c:\users\admin\anaconda3\lib\site-packages (from pl
otly) (8.0.1)
```

```
#Load the libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import datapane as dp
import pygam

from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

path = 'sbj_1_I.csv' # enter your path to the dataset here!
```

Load Data

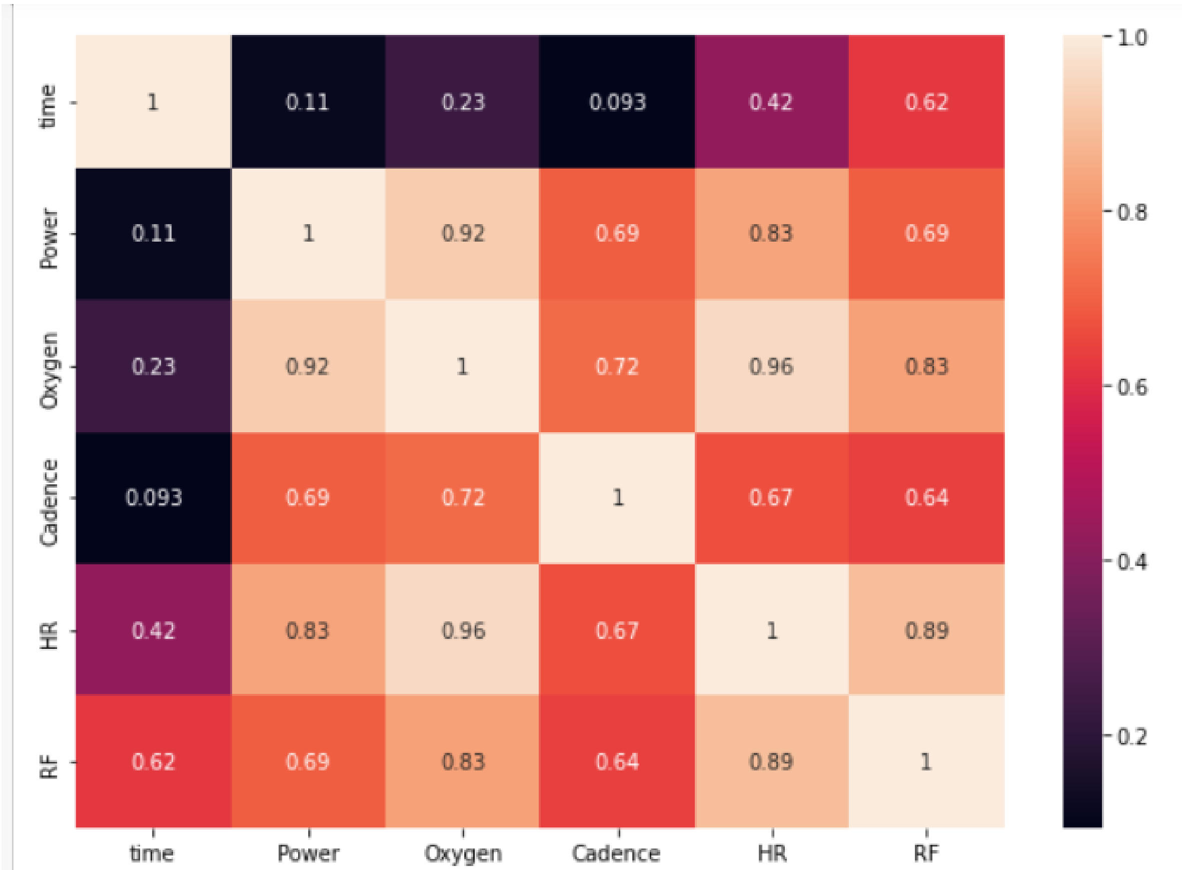
```
df = pd.read_csv(path)

df.head(3)
```

	time	Power	Oxygen	Cadence	HR	RF
0	1	0.0	318.400000	0	75.600000	20.100000
1	2	0.0	356.166667	0	75.666667	19.750000
2	3	0.0	403.285714	0	75.714286	19.428571

Display the heatmap of correlation of the features.

```
fig,ax = plt.subplots(figsize=(10,7))
sns.heatmap(df.corr(),annot=True);
```

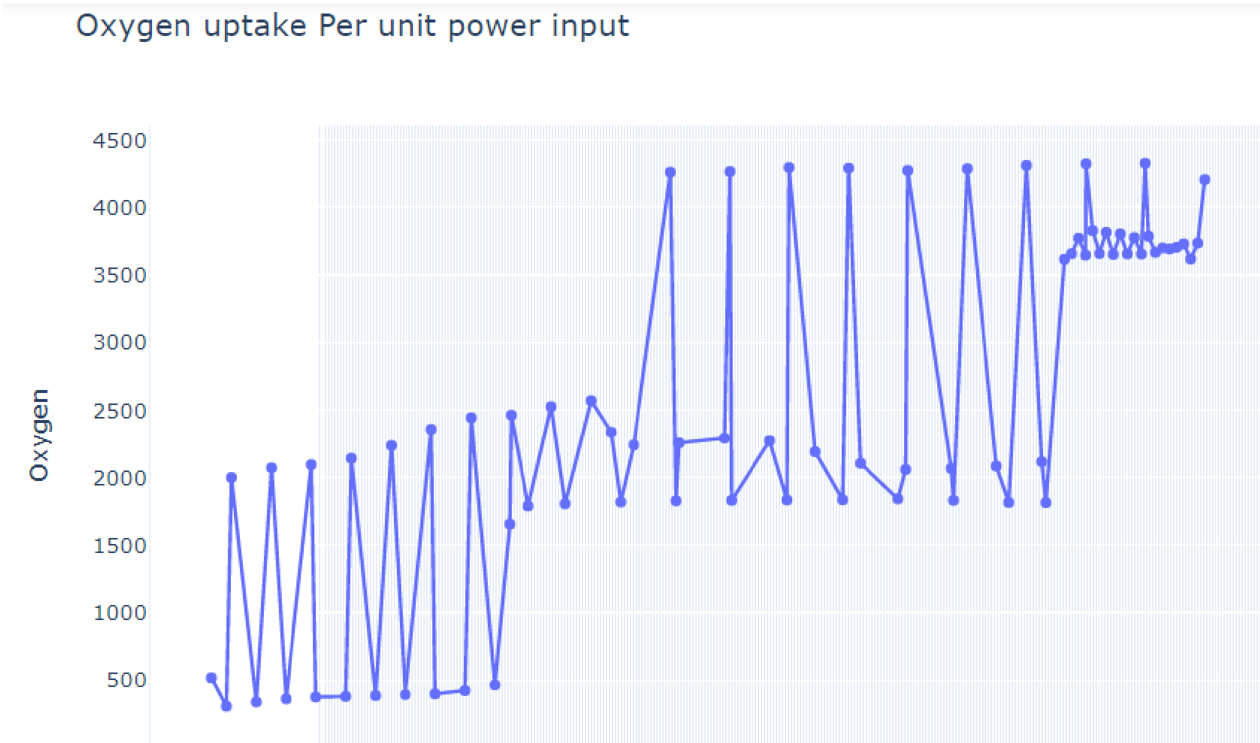


Calculate the median Oxygen value for unit power intake and build a dataframe.

Then plot the relationship between median Oxygen vs Power as shown below ->

```
median_df = df[['Power','Oxygen']].groupby('Power').median().reset_index()
```

```
plot = px.scatter(data_frame=median_df,
                  x='Power',
                  y='Oxygen',
                  title='Oxygen uptake Per unit power input')
plot.data[0].update(mode='markers+lines')
plot.update_layout(xaxis={'dtick':1},showlegend=True)
plot
```



As you can see the relationship is not linear.

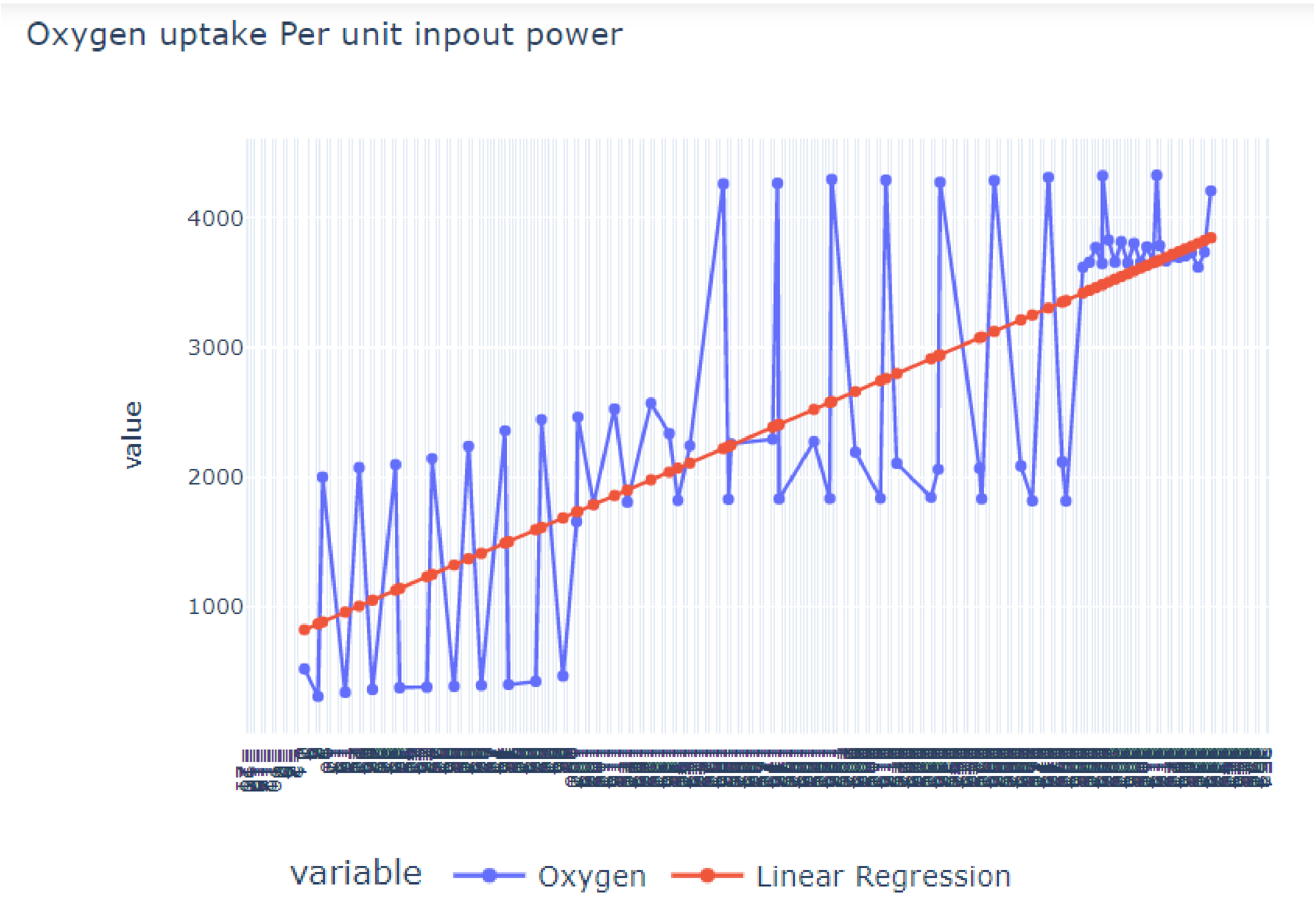
So, what will happen if we fit a simple linear regression? Lets' see ->

Linear Regression

```
]: model=LinearRegression()
   model.fit(median_df[['Power']],median_df['Oxygen'])
   median_df['Linear Regression'] = model.predict(median_df[['Power']])

   plot = px.scatter(data_frame=median_df,
                     x='Power',
                     y=['Oxygen','Linear Regression'],title='Oxygen uptake Per unit inpout power')

   plot.data[0].update(mode='markers+lines')
   plot.data[1].update(mode='markers+lines')
   plot.update_layout(xaxis={'dtick':1},showlegend=True,
                      legend=dict(
                                   orientation="h",
                                   yanchor="bottom",
                                   xanchor="right",x=.3,y=-.3))
   plot
```



As expected, this line is not a good fit solution. It doesn't capture the relationship between Oxygen and Power and cannot be used in this model.

Let's build a nonlinear model which will represent the nonlinear relationship. To do this, we create Polynomial features using the hour variable, e.g. Power^2, Power ^{3}, etc. As the order of polynomials gets higher, we use more variables, so for an order of 5 we use x, x^2, x^3,x^4 and x^5.

```
poly = PolynomialFeatures(30)
poly_df = pd.DataFrame(poly.fit_transform(median_df[['Power']]),columns = poly.get_feature_names())
poly_df.head(2)
```

	1	x0	x0^2	x0^3	x0^4	x0^5	x0^6	x0^7	x0^8	x0^9	...	x0^21	x0^22	x0^23	
0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000e+00	0.000000e+00	0.000000e+00	0
1	1.0	5.0	25.0	125.0	625.0	3125.0	15625.0	78125.0	390625.0	1953125.0	...	4.768372e+14	2.384186e+15	1.192093e+16	5

2 rows × 31 columns

```
vals = [1,3,5,10]
vals_col = []
```

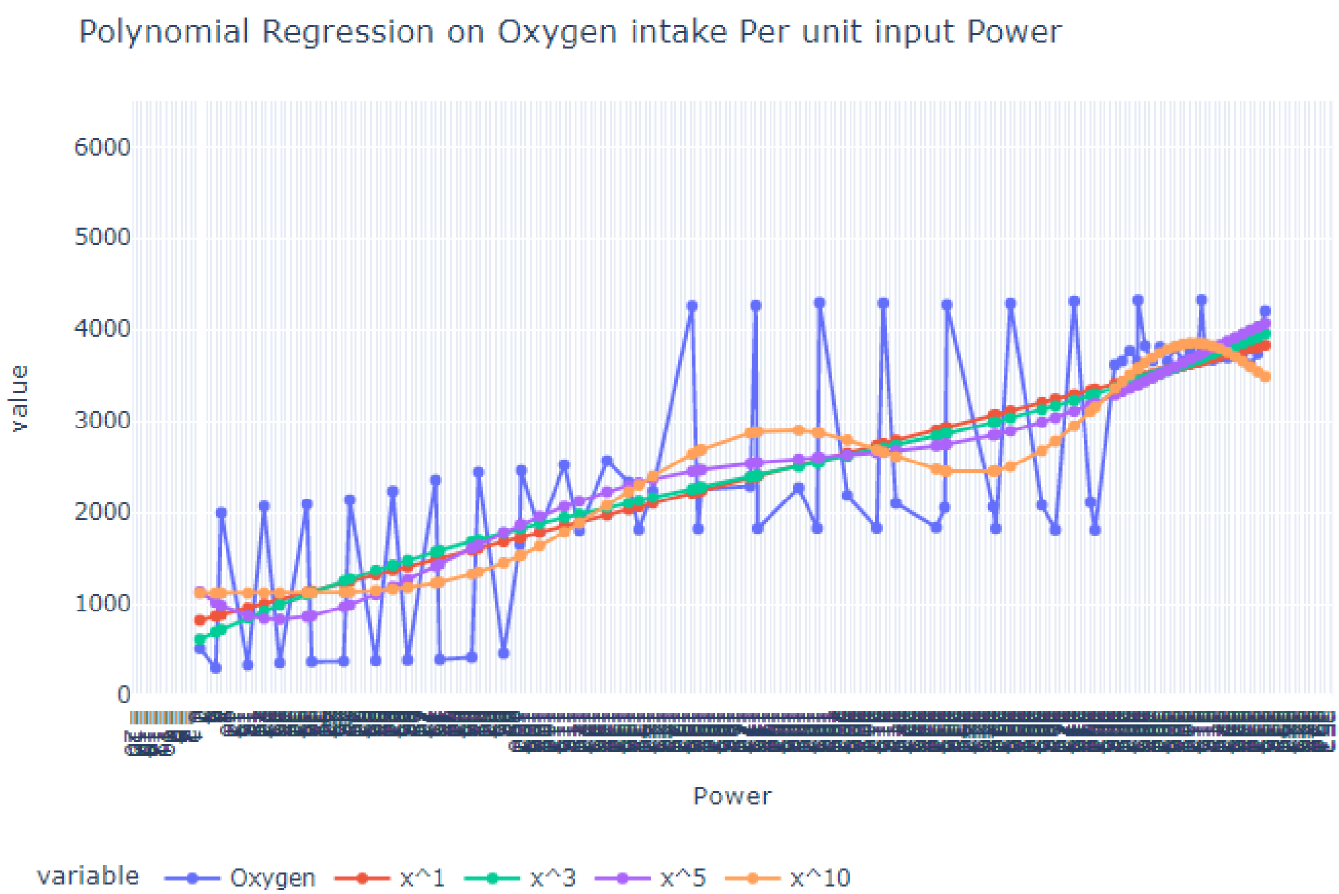
```python
for val in vals:
    n = val
    end = median_df.shape[0]-2

    model=LinearRegression()
    model.fit(poly_df.iloc[:end,:n+1],median_df['Oxygen'][:end])
    median_df[f'x^{n}'] = model.predict(poly.transform(median_df[['Power']])[:,:n+1])
    vals_col.append(f'x^{n}')
```

```python
plot = px.scatter(data_frame=median_df,
                  x='Power',
                  y=['Oxygen']+vals_col,
                 title='Polynomial Regression on Oxygen intake Per unit input Power')

for i in plot.data:
    i.update(mode='markers+lines')

plot.update_layout(xaxis={'dtick':1},showlegend=True,
                   legend=dict(
                               orientation="h",
                               yanchor="bottom",
                               xanchor="right",x=.3,y=-.3))
plot.update_yaxes(range=[0,6500])
plot
```



The above plot is showing the higher order relations using polynomial functions. These are polynomial features and will be used in GAM implementation.

We can see that the x10 model does a better job than the linear and the higher order models are able to start to simulate our relationship quite well, picking out the peaks of power.

GAM Implementation

GAM

```
median_df.head()
```

	Power	Oxygen	Linear Regression	x^1	x^3	x^5	x^10
0	0.0	517.375	820.339260	826.129591	620.068376	1136.972761	1122.264539
1	5.0	305.150	865.772067	871.307896	699.685903	1019.320467	1122.264570
2	6.7	1999.300	881.219221	886.668520	726.190111	986.859614	1122.264716
3	15.0	336.950	956.637681	961.664507	851.591257	877.443049	1122.284316
4	20.1	2071.500	1002.979143	1007.746379	925.450275	846.032224	1122.370894

```
from pygam import GAM, LinearGAM, s, f, te
```

```
lams = np.logspace(-5,5,20)
```

```
end = median_df.shape[0]-2
splines = 12
```

```
gam = LinearGAM(s(0,n_splines=splines)).gridsearch(median_df[['Power']].iloc[:end].values,
                                                    median_df['Oxygen'][:end].values,
                                                    lam=lams)
```

```
100% (20 of 20) |########################| Elapsed Time: 0:00:00 Time:  0:00:00
```

Here is the GAM Summary –

```
gam.summary()
```

```
LinearGAM
=============================================== ==========================================================
Distribution:                        NormalDist Effective DoF:                                      2.0013
Link Function:                     IdentityLink Log Likelihood:                                 -1036.0691
Number of Samples:                           72 AIC:                                             2078.1408
                                                AICc:                                            2078.4941
                                                GCV:                                           745717.9844
                                                Scale:                                         708503.1252
                                                Pseudo R-Squared:                                   0.5626
==========================================================================================================
Feature Function                  Lambda               Rank         EDoF         P > x        Sig. Code
================================= ==================== ============ ============ ============ ============
s(0)                              [100000.]            12           2.0          1.20e-09     ***
intercept                                              1            0.0          1.11e-16     ***
==========================================================================================================
Significance codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

WARNING: Fitting splines and a linear function to a feature introduces a model identifiability problem
         which can cause p-values to appear significant when they are not.

WARNING: p-values calculated in this manner behave correctly for un-penalized models or models with
         known smoothing parameters, but when smoothing parameters have been estimated, the p-values
         are typically lower than they should be, meaning that the tests reject the null too readily.
```

In GAMs, we drop the assumption that our target can be calculated using a linear combination of variables by simply saying we can use a non-linear combination of variables, denoted by s, for 'smooth function'.

$$Z = s_0 x_0 + s_1 x_1 + \cdots + s_n x_n$$

We define it with the equation below, here we see β coming back and it represents the same thing; a weight. Our other term, b is a basis expansion. A basis expansion is what we did earlier with polynomials, taking x^0, x^1, x^2, etc. There are other basis functions and they can be multi-dimensional.

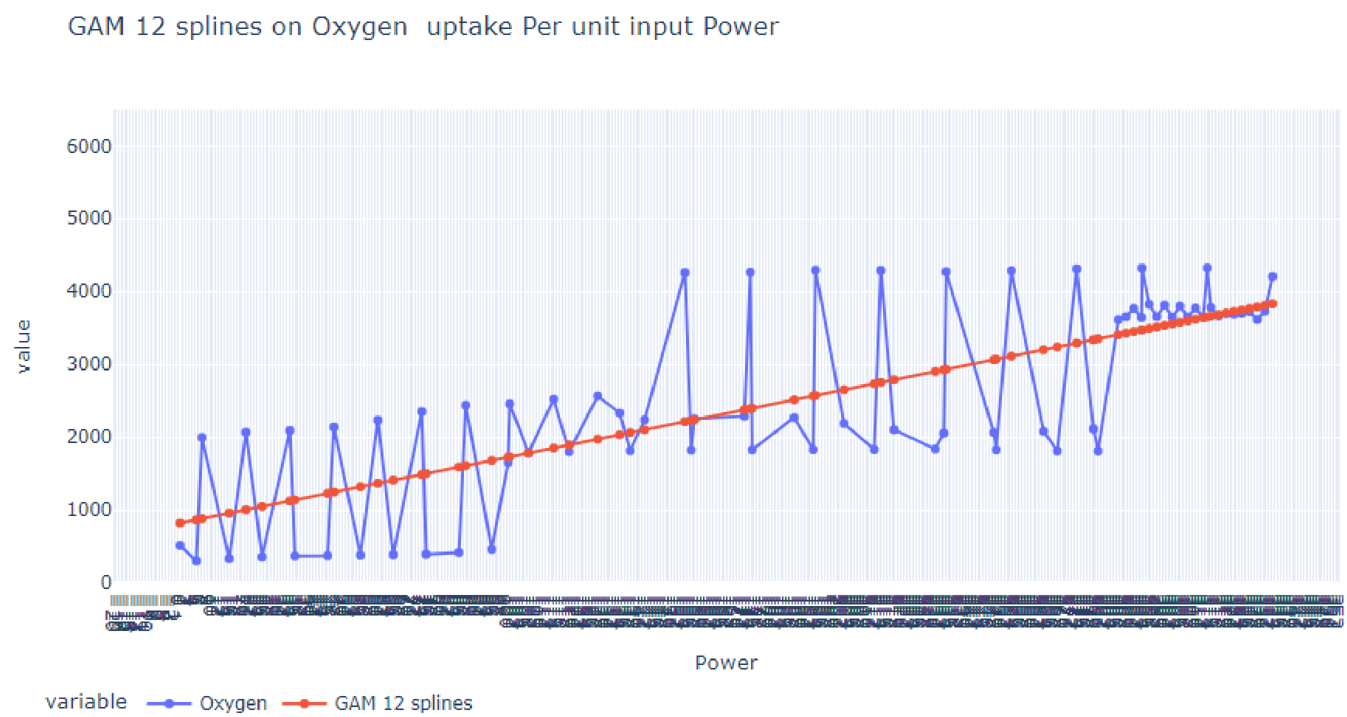
$$s(x) = \sum_{k=1}^{k} \beta_k b_k(x)$$

```
[ ] median_df[f'GAM {splines} splines'] = gam.predict(median_df[['Power']])
```

```
[ ] plot = px.scatter(data_frame=median_df,
                      x='Power',
                      y=['Oxygen',f'GAM {splines} splines'],
                     title=f'GAM {splines} splines on Oxygen  uptake Per unit input Power')

    for i in plot.data:
        i.update(mode='markers+lines')

    plot.update_layout(xaxis={'dtick':1},showlegend=True,
                       legend=dict(
                                   orientation="h",
                                   yanchor="bottom",
                                   xanchor="right",x=.3,y=-.3))
    plot.update_yaxes(range=[0,6500])
    plot
```



The great thing about this is that we can have k weights and functions per variable in our equation. This is much more flexible and much less linear than our linear regression.

This smooth function is also known as a spline. Unfortunately, splines are really hard to define, they are essentially polynomial functions that cover a small range. Splines are easier to understand if we visualize them. Here's an example of 4 splines, from the GAM we will fit shortly!

Exploring Splines

```
]: coefs = gam.coef_
```

```
]: matrix = pd.DataFrame(gam._modelmat(median_df[['Power']].iloc[:end].values).toarray())
```

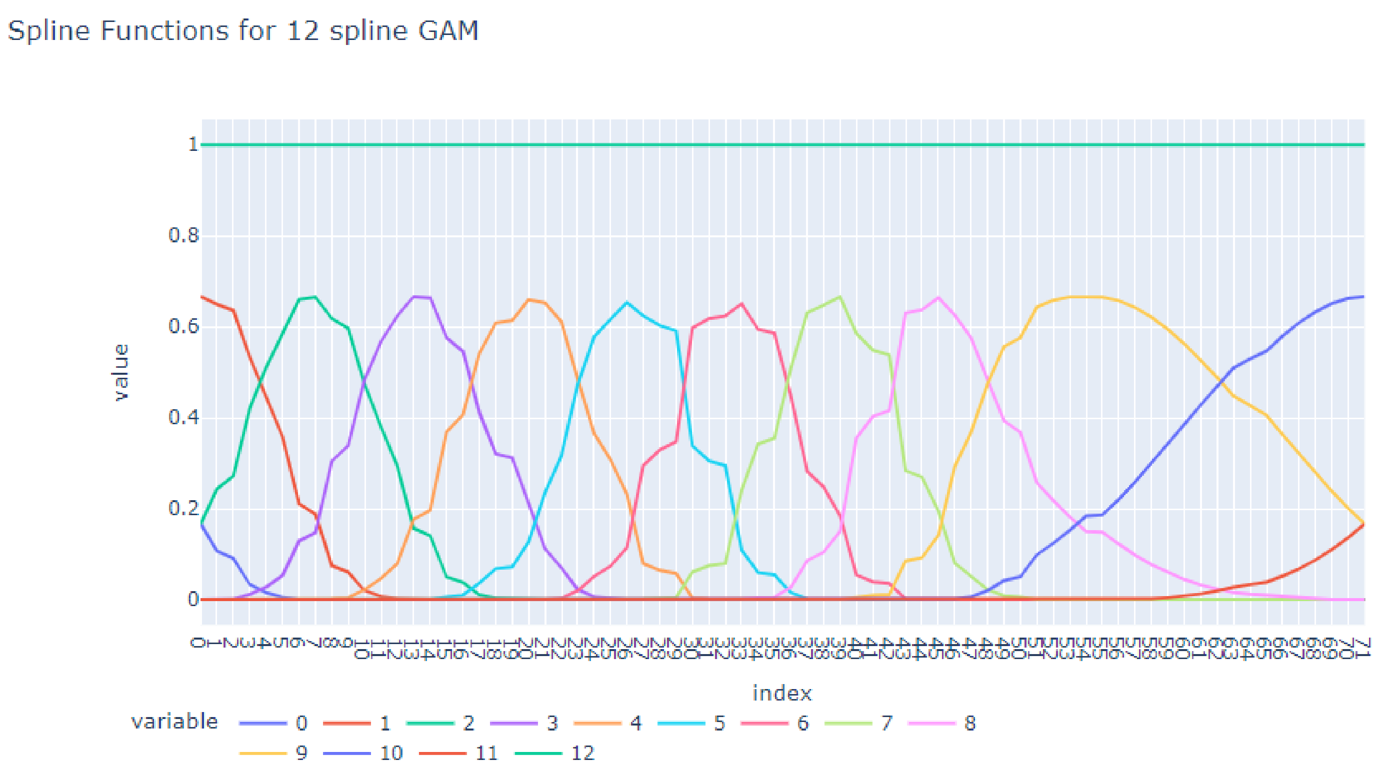
```
]: matrix.shape
]: (72, 13)
```

```
plot = px.scatter(data_frame=matrix,
                  x= matrix.index,
                  y= matrix.columns,
                 title=f'Spline Functions for {splines} spline GAM')

for i in plot.data:
    i.update(mode='lines')

plot.update_layout(xaxis={'dtick':1},showlegend=True,
                   legend=dict(
                                orientation="h",
                                yanchor="bottom",
                                xanchor="right",x=.52,y=-.3))
plot
```

The spine functions as calculated as 12 and the plots are shown below -



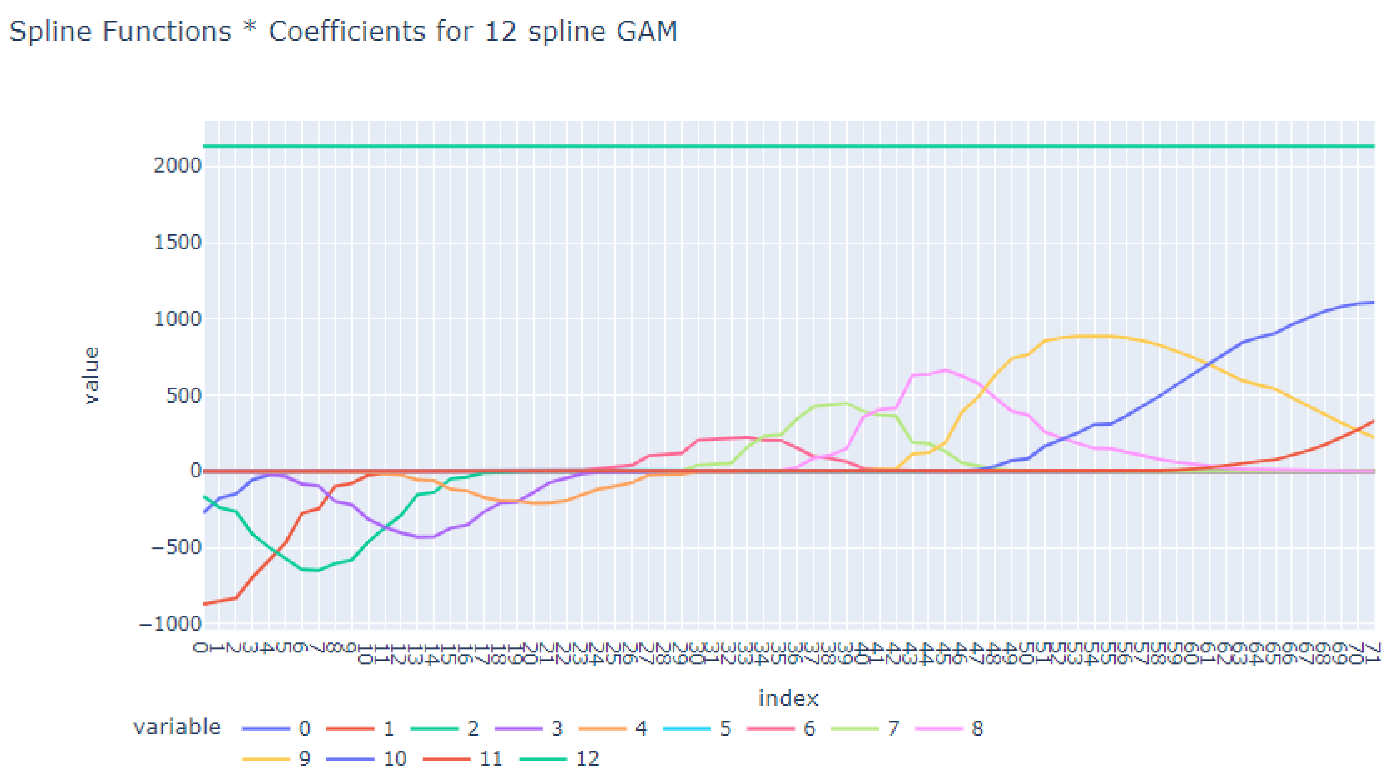
This doesn't look like it relates to our curve, this is because each spline function **also has a weight**. We can multiply the function output by the coefficients to understand what the model is doing.

```python
scaled_matrix = matrix*coefs
```

```python
plot = px.scatter(data_frame=scaled_matrix,
                  x= scaled_matrix.index,
                  y= scaled_matrix.columns,
                 title=f'Spline Functions * Coefficients for {splines} spline GAM')

for i in plot.data:
    i.update(mode='lines')

plot.update_layout(xaxis={'dtick':1},showlegend=True,
                   legend=dict(
                                orientation="h",
                                yanchor="bottom",
                                xanchor="right",x=.52,y=-.3))
plots
```

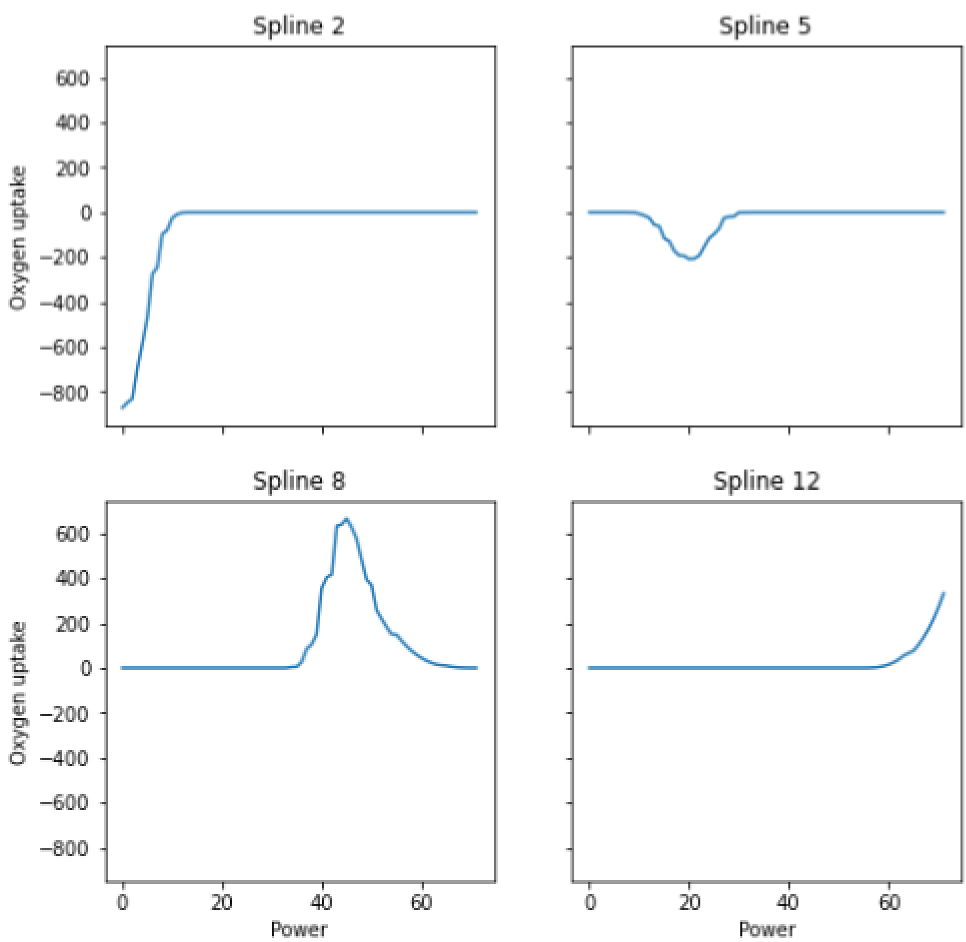


Now this looks more like our curve! Hopefully, it is intuitive as to why. **The curve is just the sum of our individual splines!** Our intercept term in green along the top.

This is a problem with a single variable, but GAM's can easily be applied to multiple variables. We can even select the number of splines **per variable** — we don't have to have the same number for everyone. We can also program interactions between variables manually.

The below plots are showing the graphs with different spline values.

```
fig, ax = plt.subplots(ncols=2,nrows=2, figsize = (8,8),sharey=True, sharex=True)
(matrix*coefs)[1].plot(title = 'Spline 2', ax=ax[0,0])
(matrix*coefs)[4].plot(title = 'Spline 5', ax=ax[0,1])
(matrix*coefs)[8].plot(title = 'Spline 8', ax=ax[1,0])
(matrix*coefs)[11].plot(title = 'Spline 12', ax=ax[1,1])
plt.setp(ax[-1, :], xlabel='Power')
plt.setp(ax[:, 0], ylabel='Oxygen uptake')
plt.show()
```



Conclusion

Here we have seen how the **spline** function is used as important factor in GAM implementation. GAM helps to model **non-linear data**. Because we can understand how our model will react to unseen data and we have to include interactions explicitly, GAMs are considered relatively **interpretable**. GAMs are best when we need an interpretable model for non-linear data.

During this research and conclusion, I learned about various jargons related to GAM and how are they going to impact the GAM model.

Spline

General rule is to use a high number of splines and use cross-validation of lambda (λ) values to find the model that generalizes best. We can have different splines and lambda values for every variable in our model.

Wiggliness

Wiggliness is literally how wiggly our line is. As number of splines increases, the line gets more wiggly with respect to the feature. The issue with this is that it will start to overfit to our data. We need to find the right number of splines so the model can learn the problem but generalize well.

Preventing Overfitting

There is another parameter called lambda, λ, this penalizes the splines. The higher lambda is, the less wiggly the line will be, until it reaches a straight line.

Link Functions

This is like regular Generalized Linear Models, link functions can be used for different distributions; the Logit function for classification problems or Log for a log transformation.

Distributions

We can also select different distributions such as Poisson, Binomial, and Normal.

Tensor Products

We can program interactions into our GAM. This is known as a tensor product. This way we can model how variables interact with each other, rather than just considering each variable in isolation.

Reference

https://environmentalcomputing.net/statistics/gams/