NNSE 784
Advanced Analytics Methods

Instructor: F Doyle (CESTM L210)

MW 4:30 – 5:50, NFN 203

Slide Set #15
Polynomial Regression

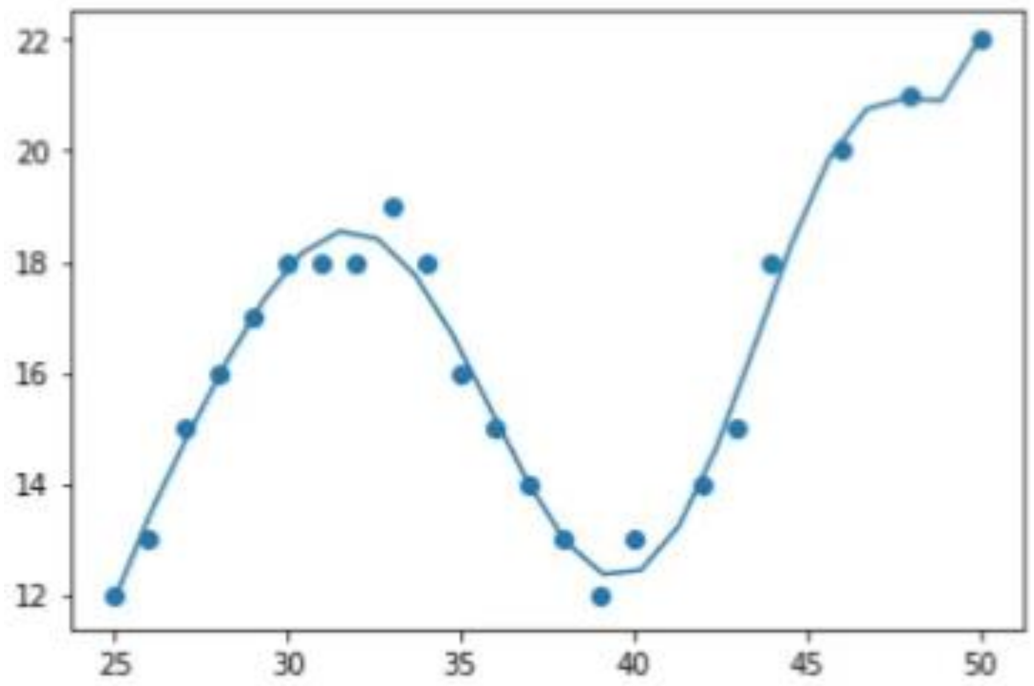
Lecture Outline

- Polynomial Regression
- Pipelines
- Overfitting vs Underfitting
- Training Error vs Testing Error

Polynomial Regression

- Special case of the general linear regression model
- Used to describe curvilinear relationships

Curvilinear relationship:
Exists when the ratio of change between predictor and target variables is not constant and cannot be fit to a straight line. Generally indicates that the predictor variable(s) will have squared or higher order terms associated.



Polynomial Regression

- Quadratic – 2nd order

$\hat{Y} = b_0 + b_1x_1 + b_2(x_1)^2$

- Cubic – 3rd order

$\hat{Y} = b_0 + b_1x_1 + b_2(x_1)^2 + b_3(x_1)^3$

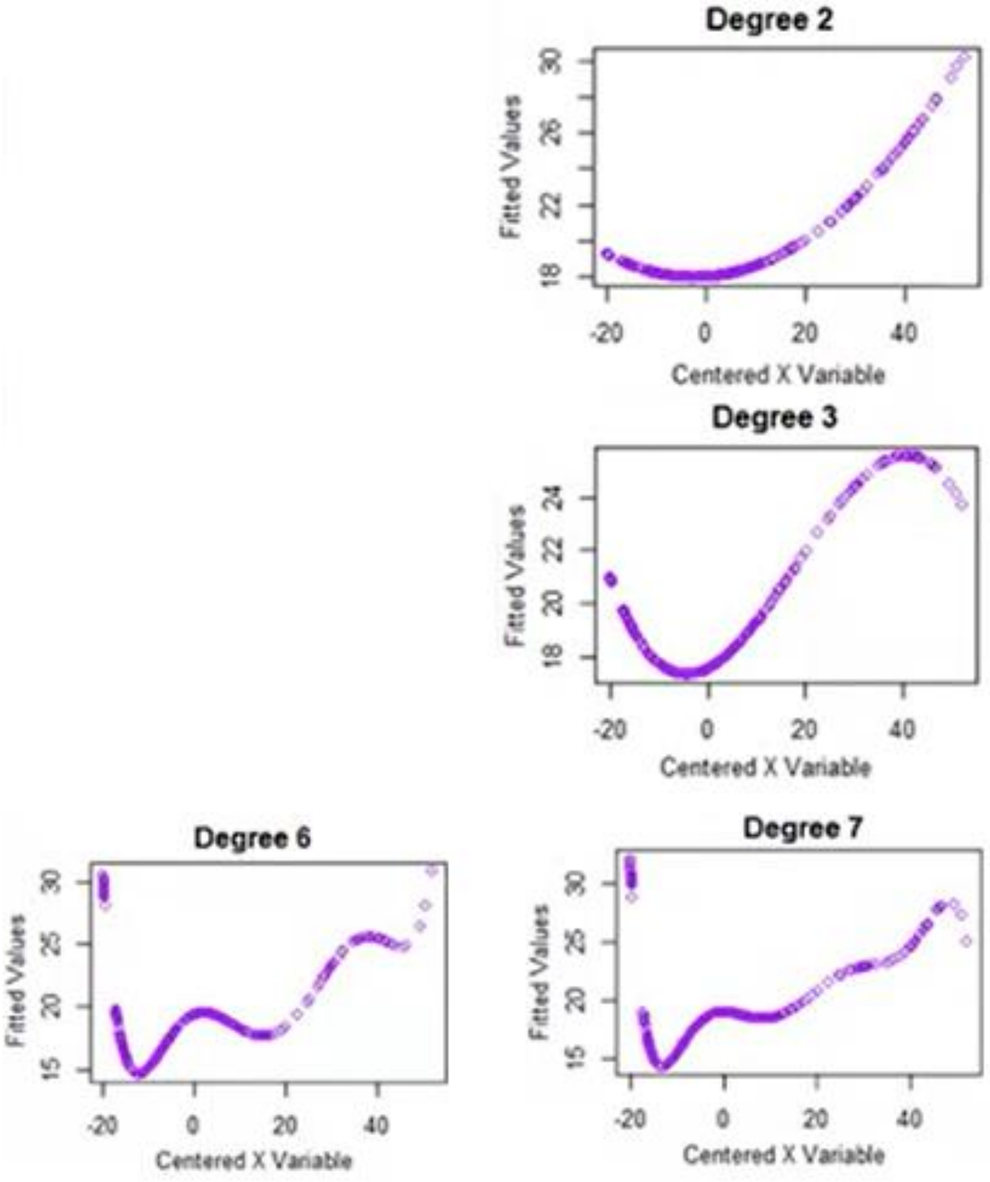
- Higher order

$\hat{Y} = b_0 + b_1x_1 + b_2(x_1)^2 + b_3(x_1)^3 + \ldots$

You might also see this written as:

$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 (x_1)^2 + \ldots$

(or some variation)



Polynomial Regression

- Calculate a polynomial of 3rd order:

```
f = np.polyfit(x, y, 3)
```

numpy

- Print the model:

```
p = np.poly1d(f)
print (p)
```

This is purely an illustrative example, we haven't defined x or y, so do not attempt to execute this code "as is"

$$-1.67(x_1)^3 + 200.3(x_1)^2 + 7856\, x_1 + 2.4532 * 10^3$$

Polynomial Regression with More than One Dimension (Multiple Predictor Variables)

- We can have multi-dimensional polynomial linear regression:

$$\hat{Y} = b_0 + b_1 x_1 + b_2 x_2 + b_3 x_1 x_2 + b_4 (x_1)^2 + b_5 (x_2)^2 + \ldots$$

numpy's polyfit() will not handle this

Polynomial Regression with More than One Dimension (continued)

- We can use the “preprocessing” library in scikit-learn:

```
from sklearn.preprocessing import PolynomialFeatures
pr = PolynomialFeatures(degree=2, include_bias=False)

x_poly = pr.fit_transform(x[[‘horsepower’, ‘curb-weight’]])
```

Generates a feature matrix of all polynomial combinations of the features with degree lees than or equal to specified degree. For example, if an input sample is two dimensional and of the form [a, b], the degree-2 polynomial features are: [1, a, b, a^2, ab, b^2]

Polynomial Regression with More than One Dimension (continued)

x_1	x_2
1	2

```
pr = PolynomialFeatures(degree=2, include_bias=False)
pr.fit_transform([[1, 2]])
```

x_1	x_2	x_1x_2	x_1^2	x_2^2
1	2	1(2)	1^2	2^2
=1	=2	=2	=1	=4

A Note on Standardization

- It is common to apply some form of standardization/normalization to data prior to using it to build a predictive model (particularly as the dimension of the data increases)
- As stated in the sklearn StandardScaler documentation:

> Standardization of a dataset is a common requirement for many machine learning estimators… they might behave badly if the individual features do not more or less look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance)

- StandardScaler offers a commonly used normalization technique
- This is the same Z transformation we discussed early in the normal distribution lecture:
    - $z = (x - \bar{x})/s$

Pre-processing Data

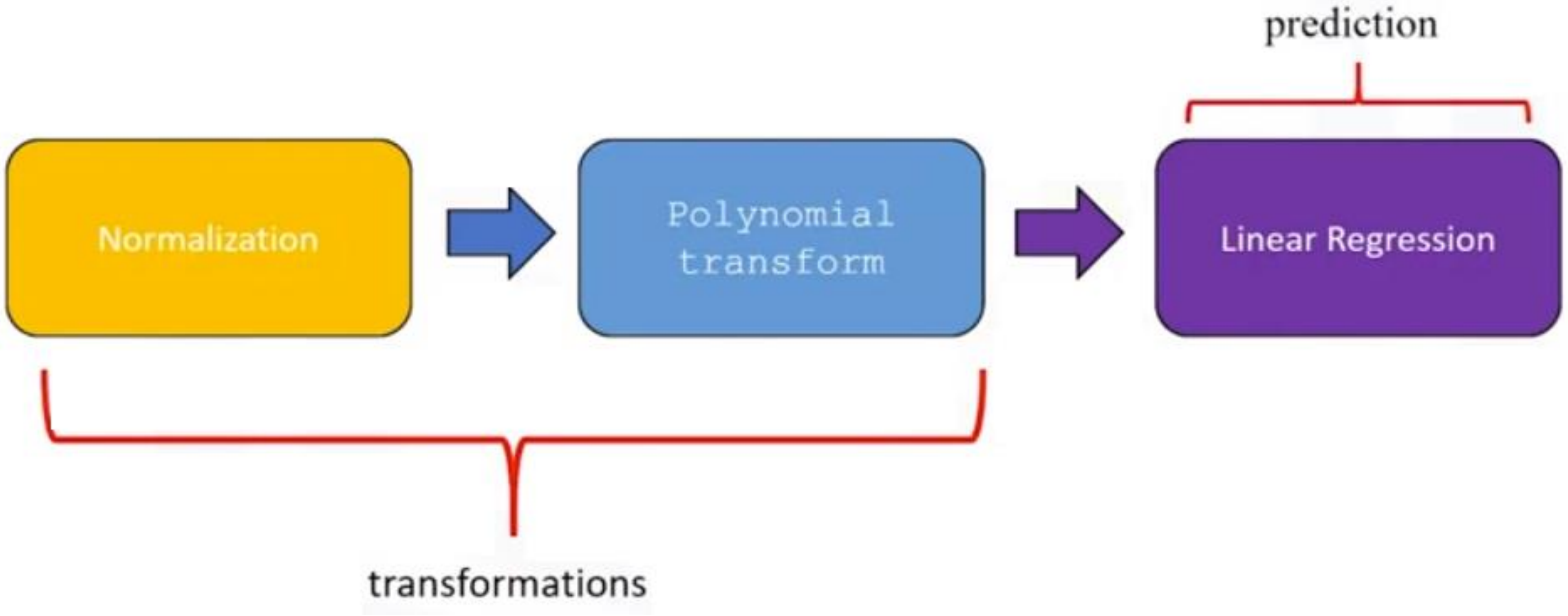
- We can normalize each feature of our data simultaneously using the preprocessing module:

```
#assume x_data is a dataframe of predictors
from sklearn.preprocessing import StandardScaler
SCALE = StandardScaler()
SCALE.fit(x_data[['horsepower', 'highway-mpg']])

x_scale = SCALE.transform(x_data[['horsepower', 'highway-mpg']])
```

Pipelines

- It is common to use “pipelines” to reduce code when building and using models that require multiple steps, such as:



Pipelines - continued

from sklearn.preprocessing import PolynomialFeatures

from sklearn.preprocessing import StandardScaler

from sklearn.linear_model import LinearRegression

from sklearn.pipeline import Pipeline

#create a set of tuples that define the name of the operation and the constructor of the object performing it in the pipeline

Input=[('scale',StandardScaler()),('polynomial',PolynomialFeatures(degree=2),…

('model',LinearRegression())]

#create the pipeline object

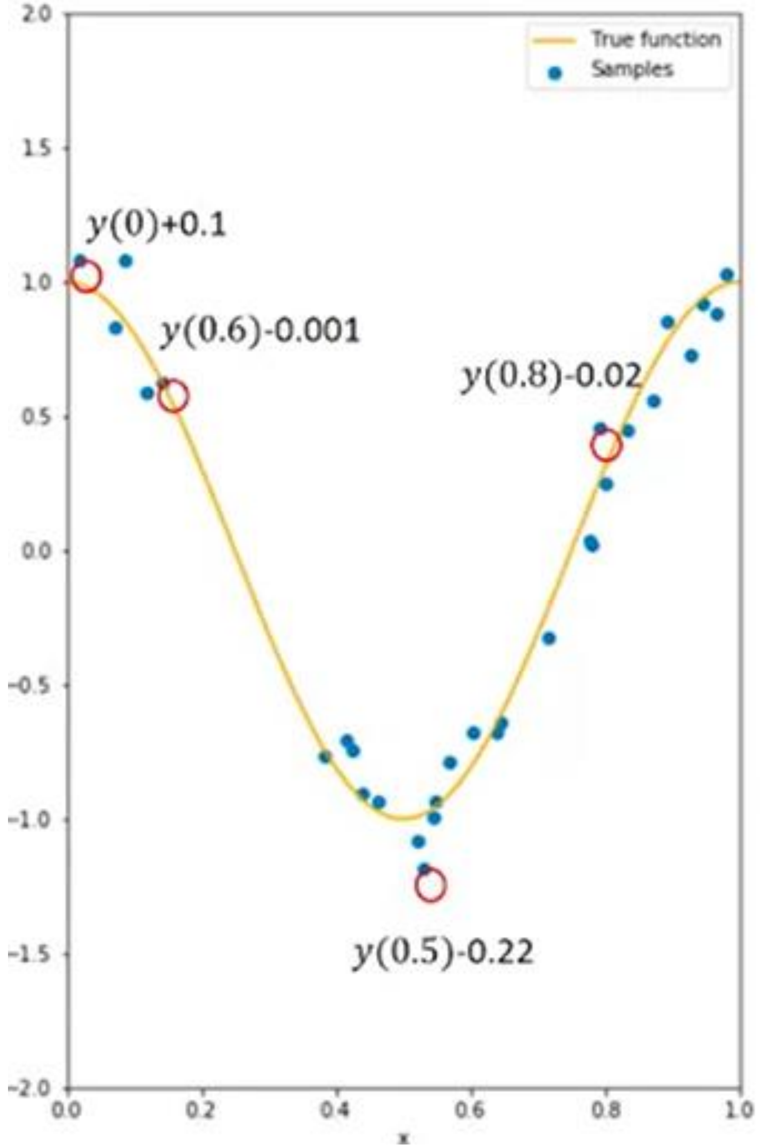
Pipe=Pipeline(Input)

Pipe.fit(df[['horsepower','curb-weight','engine-size','highway-mpg']],y)

yhat = Pipe.predict(X[['horsepower','curb-weight','engine-size','highway-mpg']])

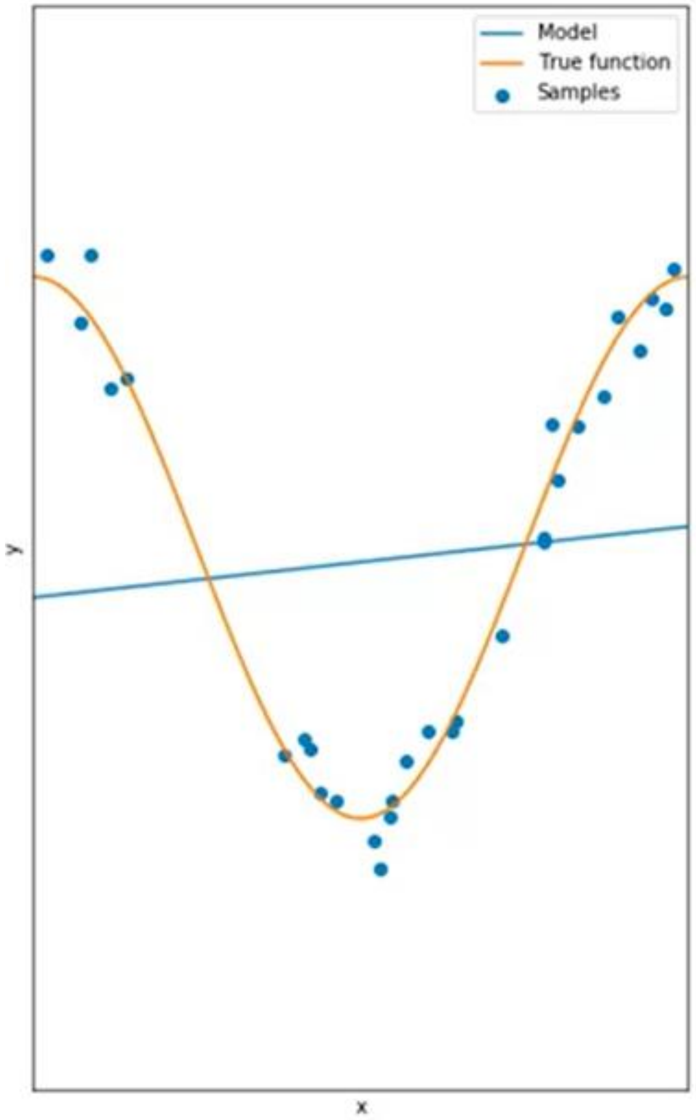
Underfitting vs Overfitting

y(x)+noise



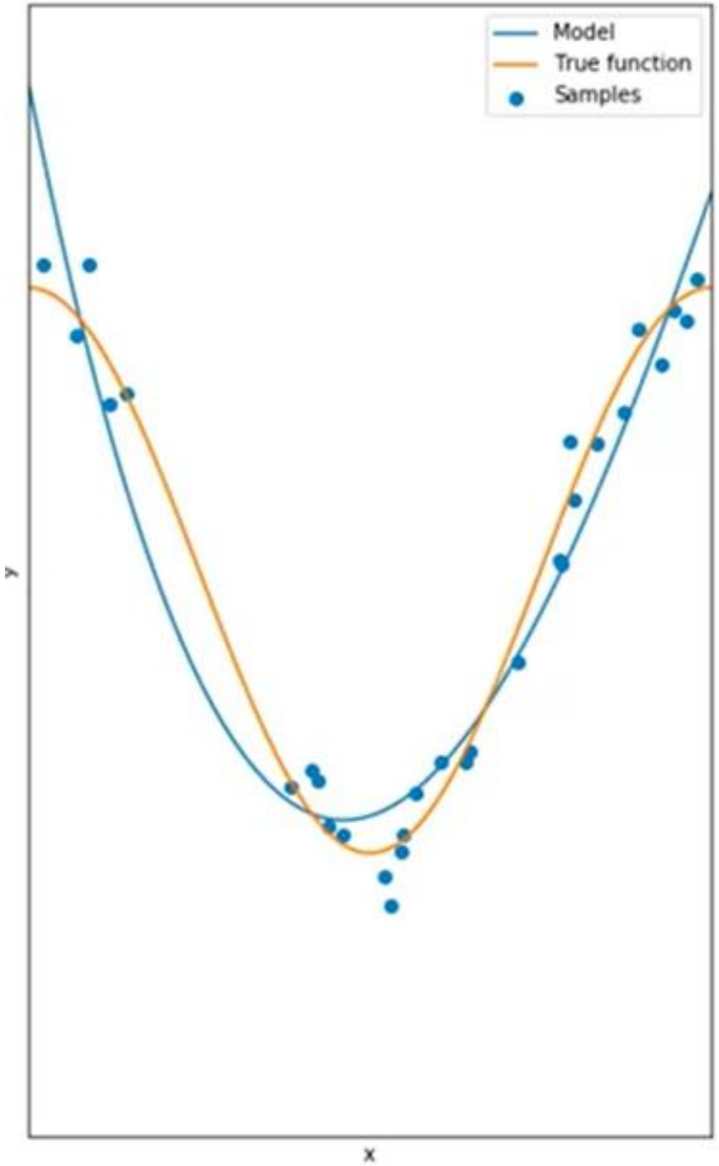
Underfitting vs Overfitting

$y = b_0 + b_1 x$



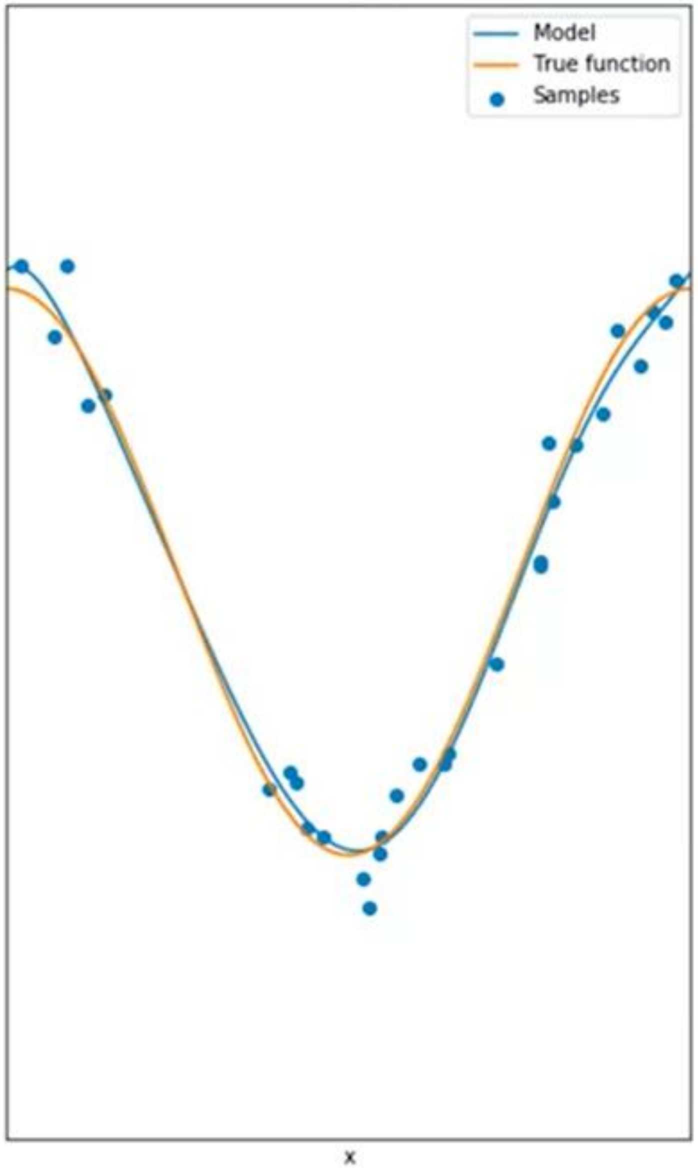
Underfitting vs Overfitting

$y = b_0 + b_1x + b_2x^2$



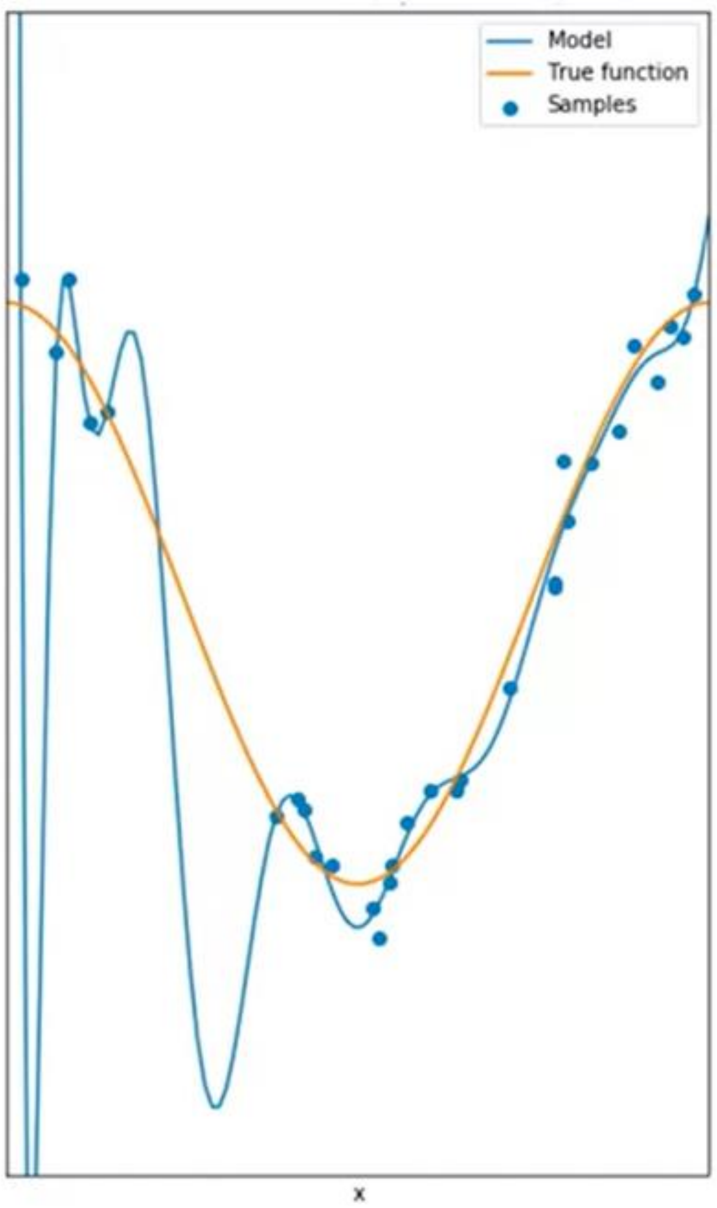
Underfitting vs Overfitting

$$y = b_0 + b_1x + b_2x^2 + b_3x^3 + b_4x^4 + b_5x^5 + b_6x^6 + b_7x^7 + b_8x^8$$



Underfitting vs Overfitting

$$y = b_0 + b_1x + b_2x^2 + b_3x^3 + b_4x^4 + b_5x^5 + b_6x^6 + b_7x^7 + b_8x^8 \ldots$$
$$+ b_9x^9 + b_{10}x^{10} + b_{11}x^{11} + b_{12}x^{12} + b_{13}x^{13} + b_{14}x^{14} + b_{15}x^{15} + b_{16}x^{16}$$



Training Error vs Testing Error