Understanding the Bias-Variance Error with Specific Python Examples

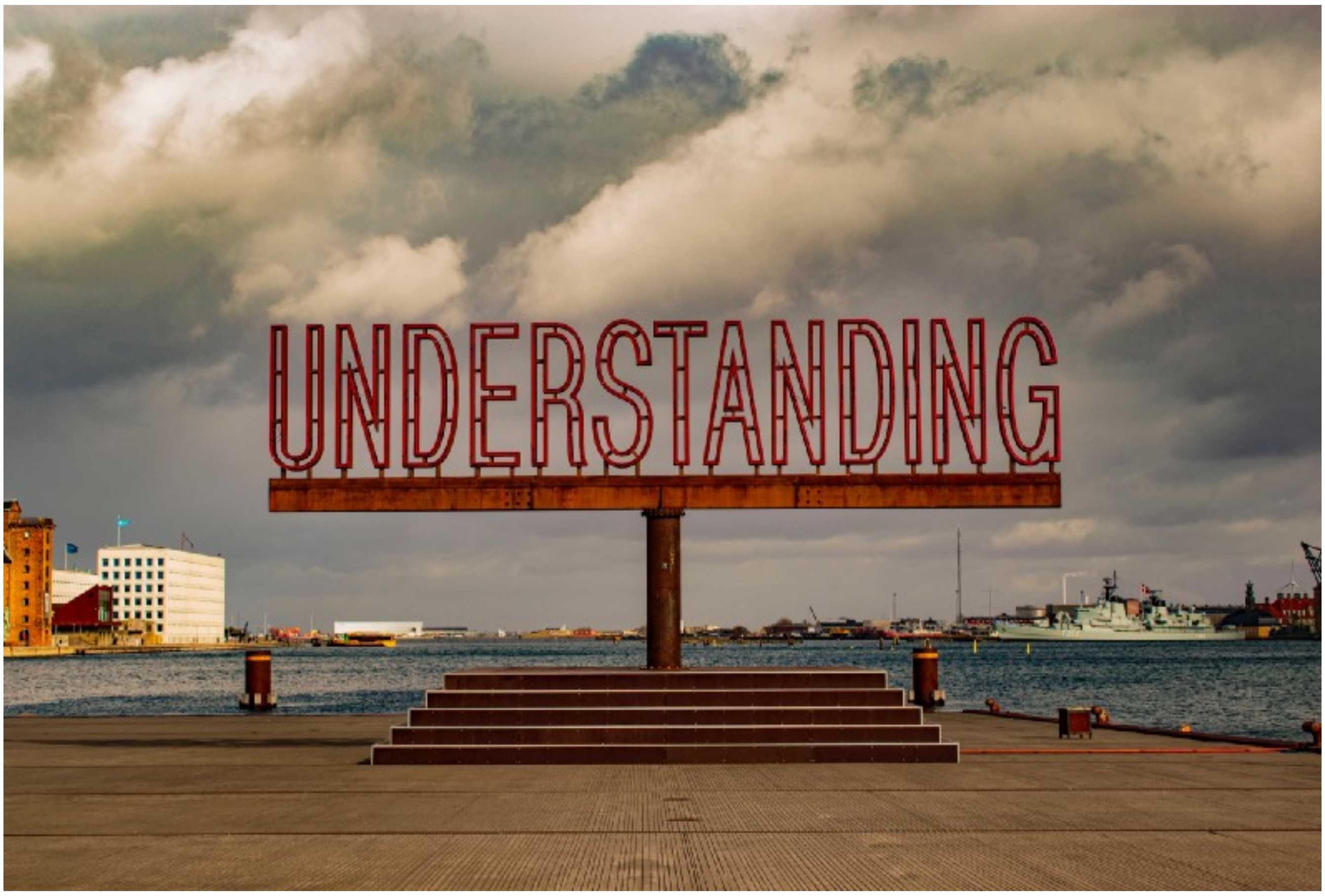


Photo by Zuzana Ruttkay on Unsplash

Introduction

In a previous **article**, I derived the equation that relates the total error in the *test data* to the Bias and Variance errors. That equation is given by

$$E_{D,\epsilon}(C_{\text{test}}) = \text{Var}(\epsilon)+\text{Bias}^2[f(\boldsymbol{x},\bar{\boldsymbol{\theta}}_D)]+\text{Var}(f(\boldsymbol{x},\bar{\boldsymbol{\theta}}_D)) \quad (1$$

where I calculated the expectation value of the test data cost function after the vector parameter $\boldsymbol{\theta}$ is "learned" in the training data**.** If you have not seen my ***article*** yet, I would recommend looking at it before reading this article**.** Said this, now is time to see the BV error in more detail and in particular by giving specific Python examples that help you better understand the concepts of Bias and Variance in statistical modelling.

As with the case of the derivation of the BV error in my previous ***article*****,** the explanation of bias and variance errors with specific examples is not very well taught in most learning materials and courses in data science and machine learning. However, since bias and variance are extremely important concepts, their understanding is imperative in statistical/machine learning. In this article, I will give specific examples of bias and variance generation in statistical/machine learning that will help you understand these concepts, hopefully once for all. Money-back guaranteed!

In this article I assume that the reader knows Python, and its most important libraries such as *Numpy, Matplotlib* and *Sklearn* that will use below.

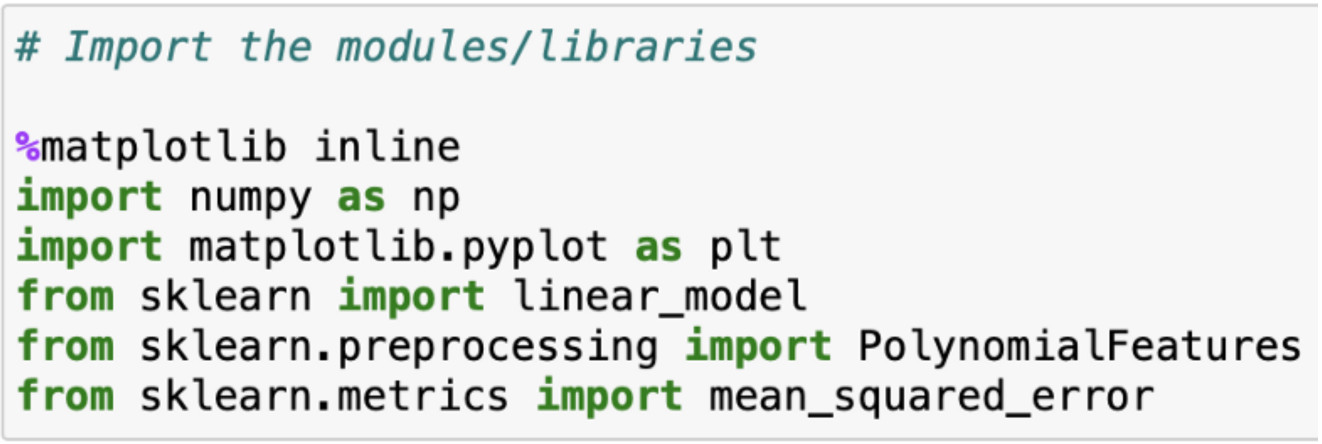
Simulating the data for the true function

Suppose that we have a situation where data are collected in the form of (*x_i, y_i*) where *x_i* are the predictor data and *y_i* are output or independent data. Suppose also that these data are generated by a probabilistic random process where we relate the output variable to the predictor variable through the equation

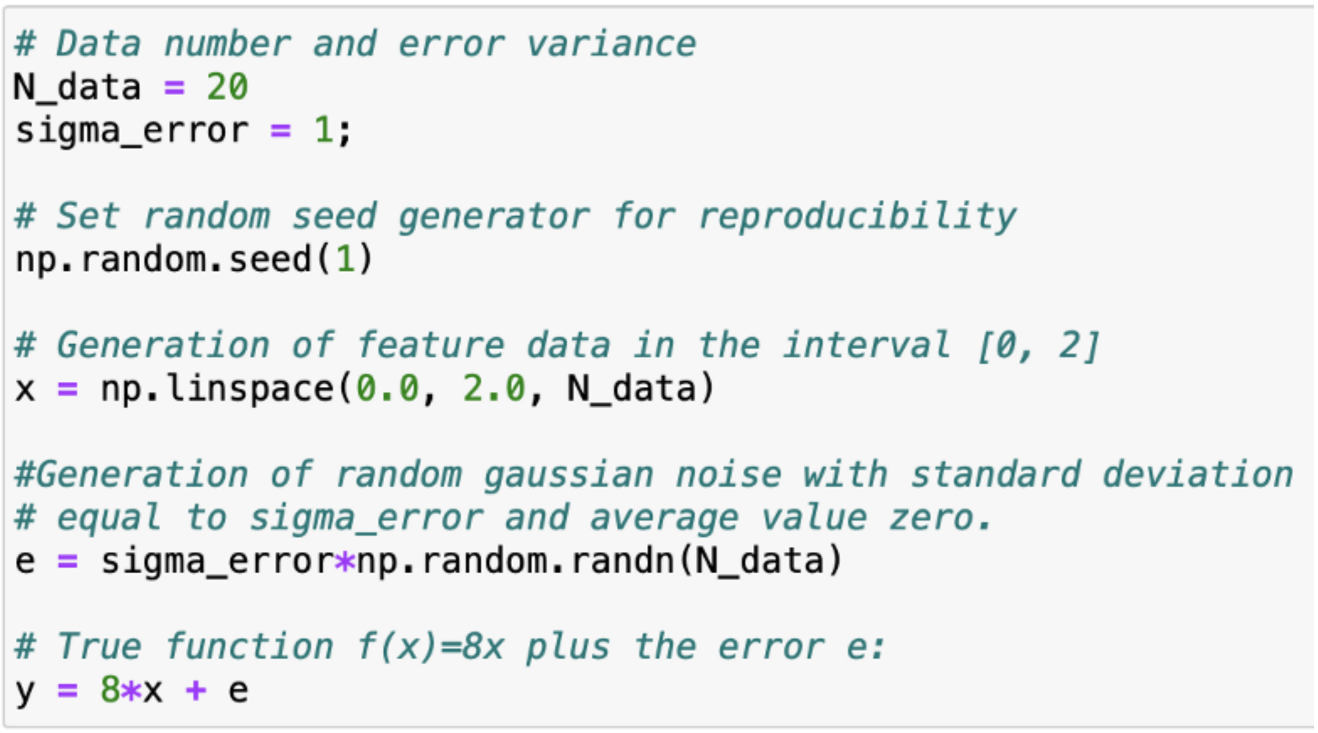
$$y(x_i) = f(x_i) + \epsilon_i \qquad (*)$$

where in equation (*) ϵ*_i* are the components of the error that is generated when we collect the data. For this article, I will consider a single independent(predictor) variable *x* instead of many independent variables. The intrinsic error ϵ is assumed to have zero mean and variance σ² and its components are assumed to be independent, uncorrelated, and follow a Gaussian distribution. Under this general hypothesis, equation (*) is a normal linear equation with a single variable where *f(x_i)* is the true function that generates the data and which, in general, is not known. Since the error is a random variable, by equation (*), *y_i,* is also a random variable. So in our analysis, we have the data in form of components, *(x_i, y_i),* that are related by equation (*) and we can sample them in many ways.

Now suppose that the data are generated by a simple linear function of the form *f(x)* = *8x*. This type of relationship might represent a physical or a biological or an economic or a social process that is modelled with a linear equation with no intercept term. Also assume that in addition exists an intrinsic error generated when the data is collected with variance, Var(ϵ) = σ^2. At this point, I will use Python to simulate the data with a fixed error standard deviation σ. I use the following Python libraries in my ***jupyter notebook***:


```
# Import the modules/libraries

%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
```
Python Code 1.

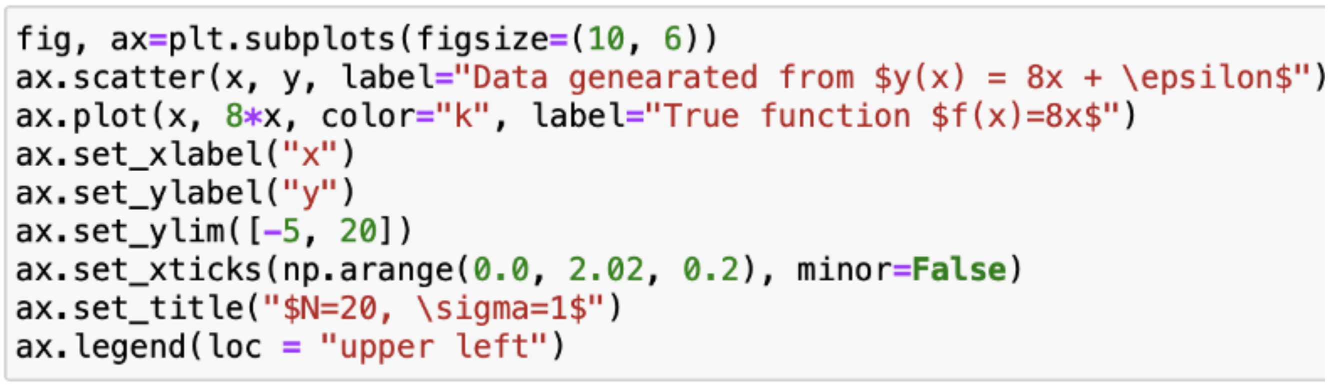
After importing the libraries of Python Code 1 that I use in what follows, I generate the data to simulate the function *y(x)* = *8x* + ϵ (where the true function that generates the data is *f(x)=8x)* with the following Python code:



```
# Data number and error variance
N_data = 20
sigma_error = 1;

# Set random seed generator for reproducibility
np.random.seed(1)

# Generation of feature data in the interval [0, 2]
x = np.linspace(0.0, 2.0, N_data)

#Generation of random gaussian noise with standard deviation
# equal to sigma_error and average value zero.
e = sigma_error*np.random.randn(N_data)

# True function f(x)=8x plus the error e:
y = 8*x + e
```

Python Code 2.

In Python Code 2, again I assume that the reader is familiar with the Python functions that are present in the code. In Python Code 2, I considered an error with a standard deviation equal to 1 as a matter of example. Now I plot the true function that generates the data, *f(x)=8x*, and the function that we find when the collect the data, *y(x)* = *8x* + ϵ



```
fig, ax=plt.subplots(figsize=(10, 6))
ax.scatter(x, y, label="Data genearated from $y(x) = 8x + \epsilon$")
ax.plot(x, 8*x, color="k", label="True function $f(x)=8x$")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_ylim([-5, 20])
ax.set_xticks(np.arange(0.0, 2.02, 0.2), minor=False)
ax.set_title("$N=20, \sigma=1$")
ax.legend(loc = "upper left")
```

Python Code 3.

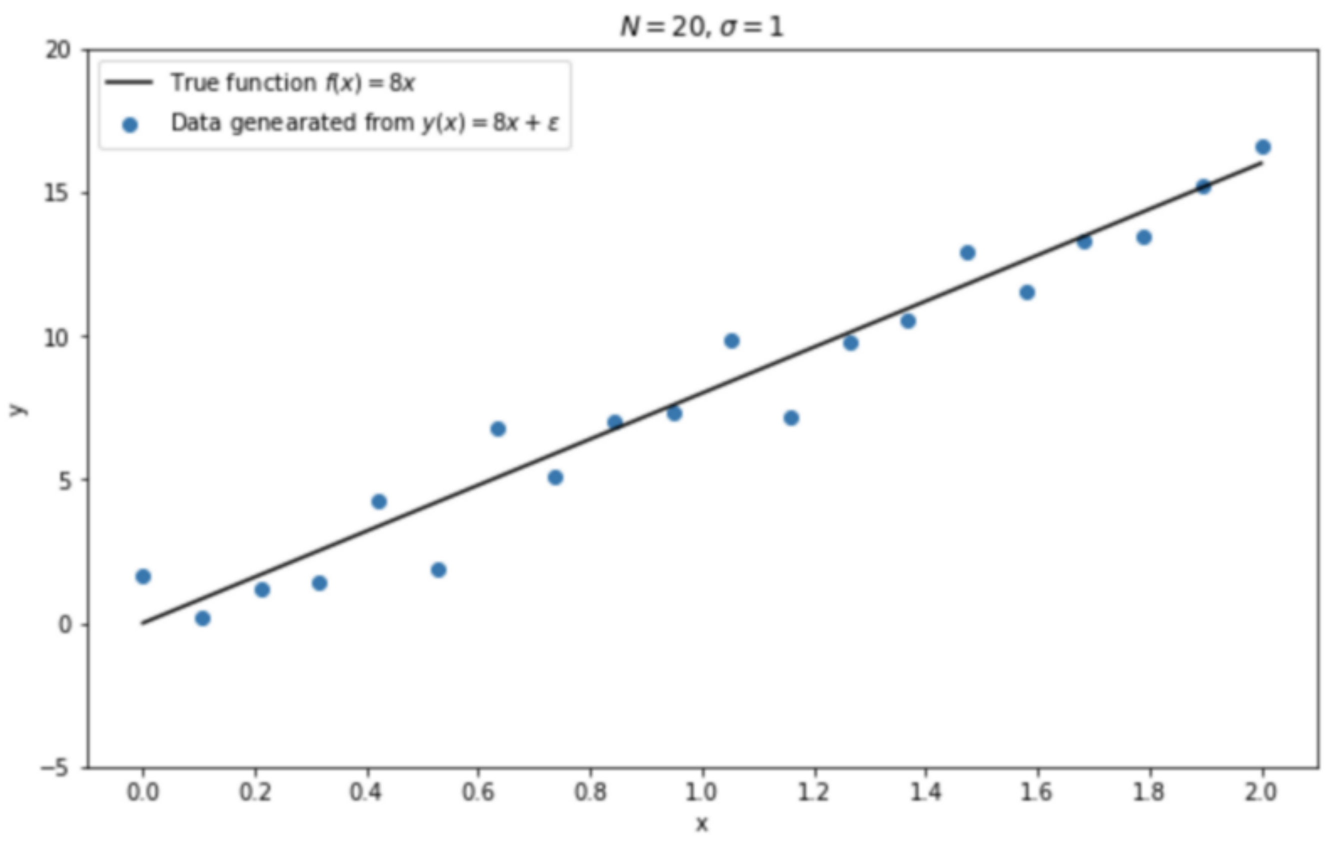


Fig. 1. Plot of the true function f(x) = 8x and the simulated function y(x) = f(x) + ϵ, for data number N=20 and error standard deviation equal to 1, generated with the Python Code 3 (Figure explicitly created by the author for educational purposes).

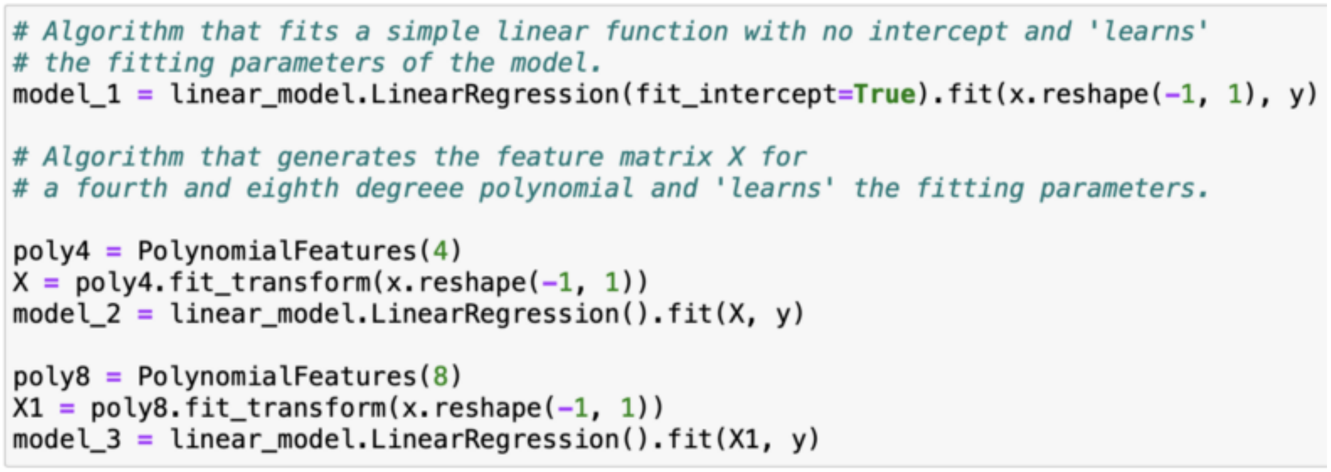
The key point here that want you to understand is that in general, the true function that generates the data f(x) is never observed. What is actually observed is the function *y(x)* that I simulated above. This function is given by the sum of the true function *f(x)* and the error ϵ that is generated when we collect the data. If you want some references how error is generated in data collection, see my **article**. The data points in Fig. 1 are the data that we would get in a specific model that we want to fit. These data intrinsically include the error ϵ that exists when data are collected. Clearly, here I am not considering possible error bars that might be associated with each data point in Fig. 1 because it is out of context for this article.

If you have managed to follow me so far, then congratulations to you! If you have something that did not understand, then I would suggest reading the information above again to understand the details.

Regression for training and test data

The total error in equation (1) is valid in the case of quantitative variables (discrete/continuous), namely for regression problems. Now, suppose that we collected the data and want to fit the data in Fig. 1 with three different models. The first fitting model is that of a simple linear function of the type $y\_1(x) = a + bx$, the second model is a fourth-order polynomial $y\_2(x) = y\_1(x)+cx^2 + dx^3 + gx^4$, and the third model is eighth-order polynomial $y\_3(x) = y\_2(x)+ hx^5 + jx^6 +kx^7 + lx^8$. The Latin letters in front of the polynomial variable *x* are just parameters and not variables. Clearly, since I am considering a single predictor variable *x*, there are not mixed terms in the above polynomial functions but just self-interactions of the variable *x*.

The first model function *y_1(x)* is a subclass of the model functions *y_2(x)* and *y_3(x)*, while the model function *y_2(x)* is a subclass of the function *y_3(x)*.Arrived at this point, I run the training-test method in order to fit and predict for each model function. Here I use the sklearn library and run the following Python code for the training data:



```
# Algorithm that fits a simple linear function with no intercept and 'learns'
# the fitting parameters of the model.
model_1 = linear_model.LinearRegression(fit_intercept=True).fit(x.reshape(-1, 1), y)

# Algorithm that generates the feature matrix X for
# a fourth and eighth degreee polynomial and 'learns' the fitting parameters.

poly4 = PolynomialFeatures(4)
X = poly4.fit_transform(x.reshape(-1, 1))
model_2 = linear_model.LinearRegression().fit(X, y)

poly8 = PolynomialFeatures(8)
X1 = poly8.fit_transform(x.reshape(-1, 1))
model_3 = linear_model.LinearRegression().fit(X1, y)
```

Python Code 4.

The Python Code 4 follows the Python Codes 1 and 2 for the training data, where I considered N_data=60 and sigma_error=1, instead of N_data=20, sigma_error=1. After the parameters are 'learned' for the training data, namely after we know the functions *f(x,**θ_D**)* from the training set, I run a similar code to the Python Code 3 (you can see the detailed code sequence in my GitHub page) to plot the learned functions for the training data as shown in Fig. 2 below

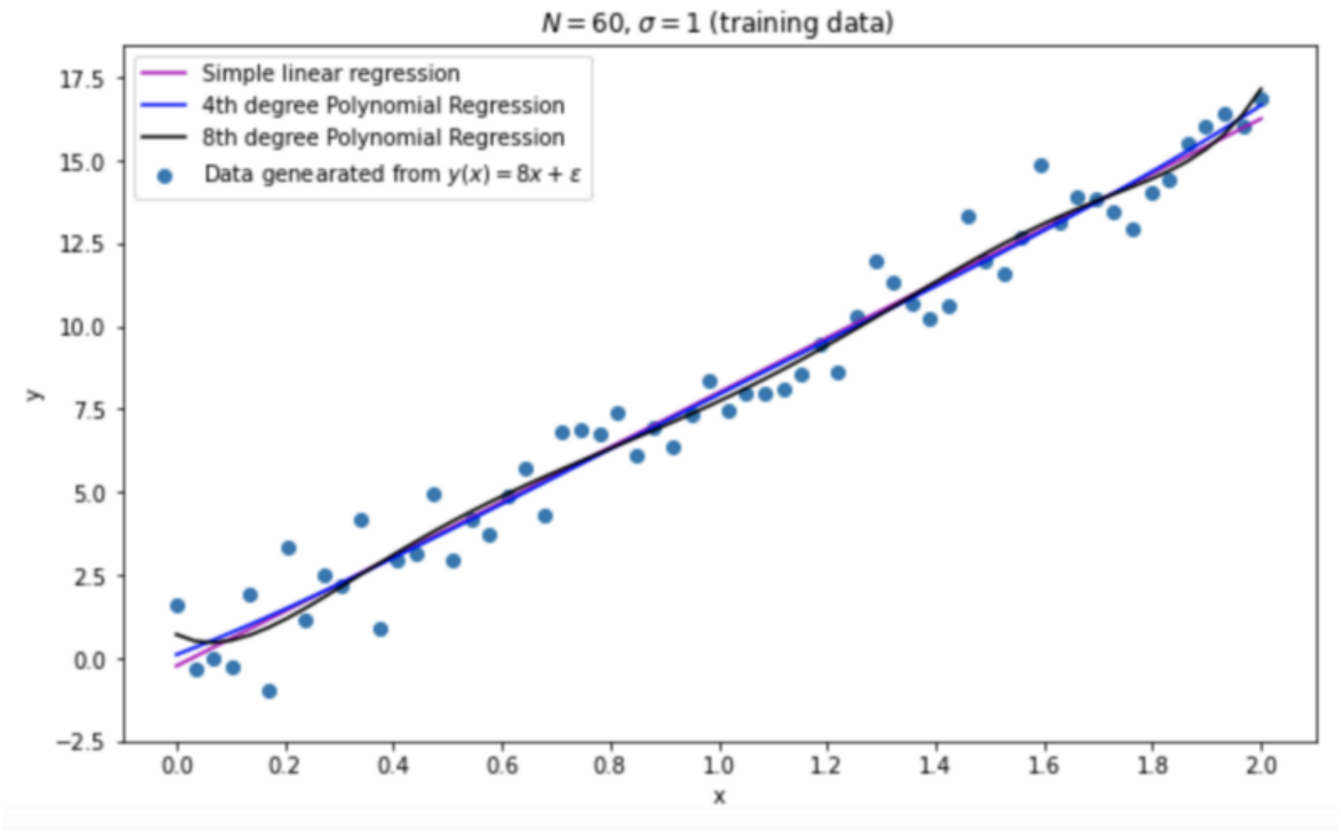
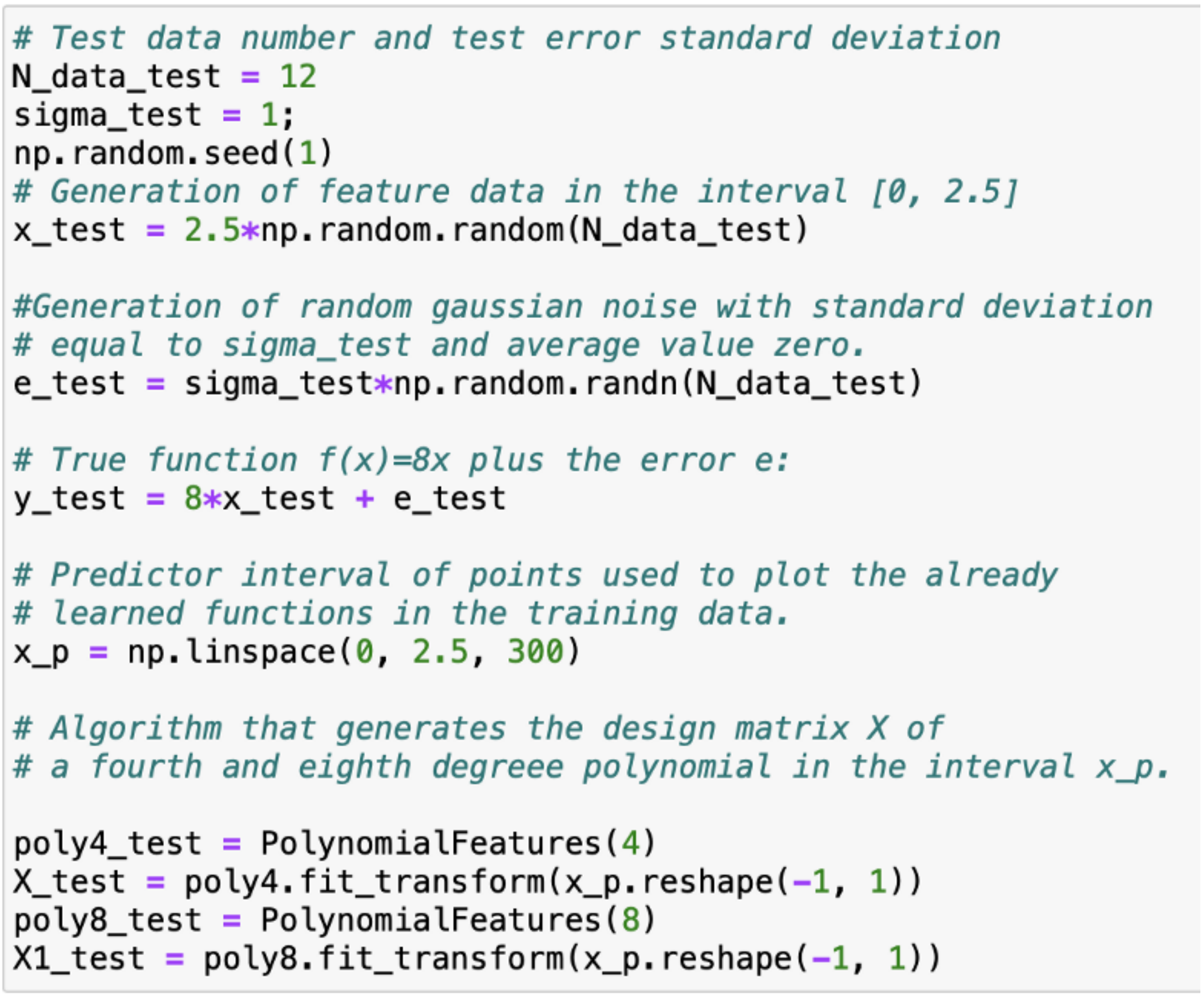


Fig.2 (Figure created by the author for educational purposes)

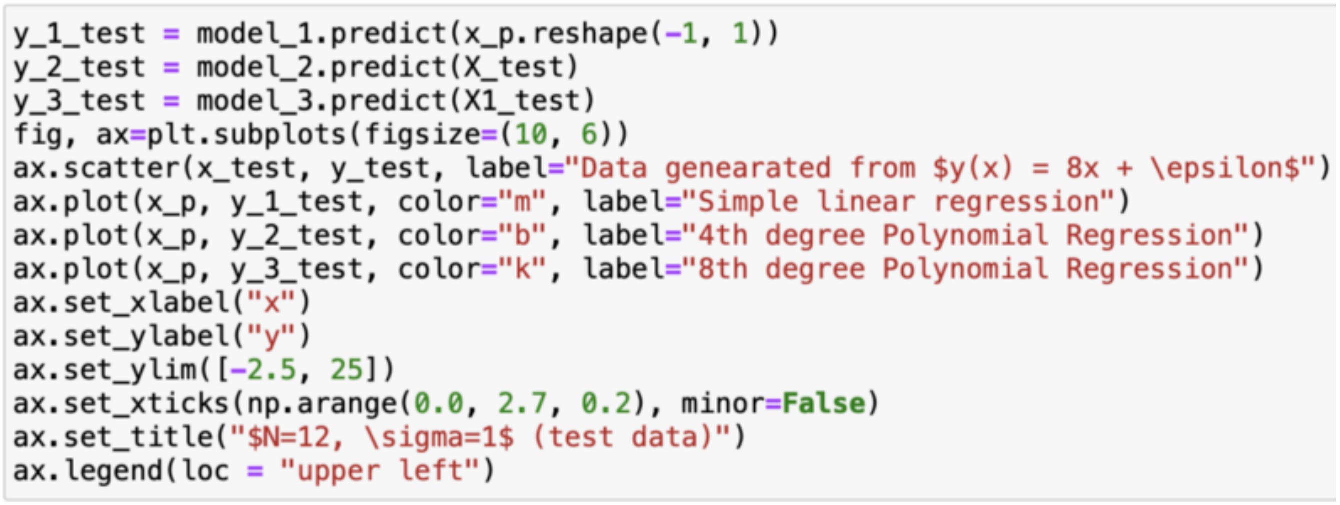
In Fig. 2, one can visually see how well the functions *y_1, y_2, y_3* fit the training data that have been randomly generated. At this point, I might explicitly check which of the functions *y_1, y_2, y_3* better fits the randomly generated data by calculating the *MSE (*Mean Square-Error*)* for the training data, but that is not the purpose of statistical/machine learning. The goal of statistical/machine learning is that of how well do the models used in the training data, namely *y_1, y_2, y_3*, predict for not yet seen data.

I use the "learned" functions *y_1, y_2, y_3* in new data, the test data, in order to see which function gives the best predictions. I run the Python Codes 5 and 6 below and after I show the plots obtained from the Python Code 6. The plots are shown in Fig. 2 for the test data.



```
# Test data number and test error standard deviation
N_data_test = 12
sigma_test = 1;
np.random.seed(1)
# Generation of feature data in the interval [0, 2.5]
x_test = 2.5*np.random.random(N_data_test)

#Generation of random gaussian noise with standard deviation
# equal to sigma_test and average value zero.
e_test = sigma_test*np.random.randn(N_data_test)

# True function f(x)=8x plus the error e:
y_test = 8*x_test + e_test

# Predictor interval of points used to plot the already
# learned functions in the training data.
x_p = np.linspace(0, 2.5, 300)

# Algorithm that generates the design matrix X of
# a fourth and eighth degreee polynomial in the interval x_p.

poly4_test = PolynomialFeatures(4)
X_test = poly4.fit_transform(x_p.reshape(-1, 1))
poly8_test = PolynomialFeatures(8)
X1_test = poly8.fit_transform(x_p.reshape(-1, 1))
```

Python Code 5.



```
y_1_test = model_1.predict(x_p.reshape(-1, 1))
y_2_test = model_2.predict(X_test)
y_3_test = model_3.predict(X1_test)
fig, ax=plt.subplots(figsize=(10, 6))
ax.scatter(x_test, y_test, label="Data genearated from $y(x) = 8x + \epsilon$")
ax.plot(x_p, y_1_test, color="m", label="Simple linear regression")
ax.plot(x_p, y_2_test, color="b", label="4th degree Polynomial Regression")
ax.plot(x_p, y_3_test, color="k", label="8th degree Polynomial Regression")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_ylim([-2.5, 25])
ax.set_xticks(np.arange(0.0, 2.7, 0.2), minor=False)
ax.set_title("$N=12, \sigma=1$ (test data)")
ax.legend(loc = "upper left")
```

Python Code 6.

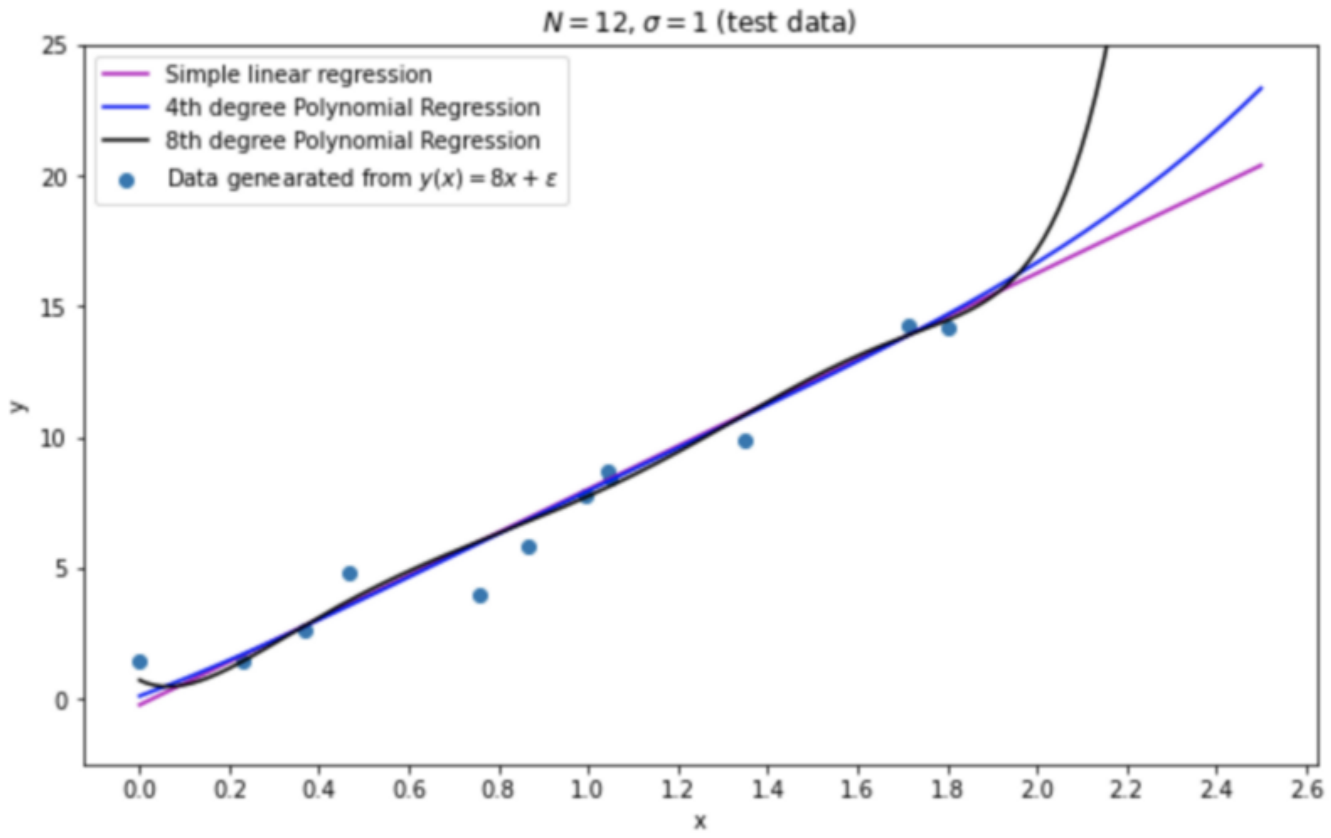


Fig. 3 (Figure created by the author for educational purposes)

Now it's time to pause a little and try to reflect on what has been done so far. After training the data and having "learned" the functions *y_1, y_2,* and *y_3,* I plotted the learned functions as shown in Fig. 2. Regarding the test data, let me explain the steps involved in the Python Codes 5 and 6 to better understand the whole procedure. In Python Code 5, I used only 12 data points for the test dataset, approximately 20% of the number of the training data. For the test data, I randomly generated 12 predictor data, *x_test*, in the interval [0, 2.5] by using the Python function np.random.random. After that, I also generated the random error, *e_test*, and formed the random variable *y_test(x)=8x_test+e_test*. Since the generated pair *(x_test, y_test)* is completely random, they are generally different from training data, so the test data are in general different from the training data.

In Python Code 5, I also generated the array, *x_p,* and used it to create the design matrix for the polynomial functions *y_2* and *y_3* in the interval [0, 2.5] and after I created the plots in Fig. 3 with the Python Code 6. One may note that I used *x_p* to plot the learned functions. Indeed, in Python Code 4, the model functions are already learned, namely, Python internally learned the coefficients of each function in the training dataset. By using the array *x_p,* Python has enough data points to plot the learned functions in a larger interval, so the output plots look smoother and not wiggly. I did not use *(x_test, y_test)* data to plot the "learned" functions. With the plots of the learned functions, *y_1, y_2,* and *y_3*, I also created a scatter plot of the randomly generated test data with the equation *y_test(x)=8x_test+e_test* in Fig. 3.

In Fig. 3, I show the learned functions *y_1, y_2* and *y_3* obtained from the training data and the test data *(x_test, y_test)*. Now we can ask the question: which learned function fits better the test data? Before I reply to this question, let me comment first on Fig. 2 that was created by using the *training data*. In that figure, you can see that the eighth-order polynomial function *y_3* better fits the training data and it gives the ***lowest training MSE*** in comparison to the other functions since it fits the training data much better. On the other hand, the simple linear function *y_1*, has the ***highest training MSE***. *This observation tells us that as the number of parameters that enter the model function increases (the function flexibility increases), its **training MSE** decreases.*

Now the returning to the question above of which function fits better the test *MSE*, you can see from Fig. 3, that the eighth-order polynomial *y_3* fits the test data the worts to the linear and fourth-order functions. This is a classical case of *overfitting* of the function *y_3*.

Both plots in Fig. 2 and Fig. 3 have been created by using a relatively low number of data points in the training and test datasets. What happens to these plots if we increase the number of data in both datasets like, N=200 (training dataset) and N=40 (test dataset)? To see the impact of larger datasets on the plots, I repeat the same steps that led to the creation of the plots in Fig. 2 and Fig. 3 and get the following plots:

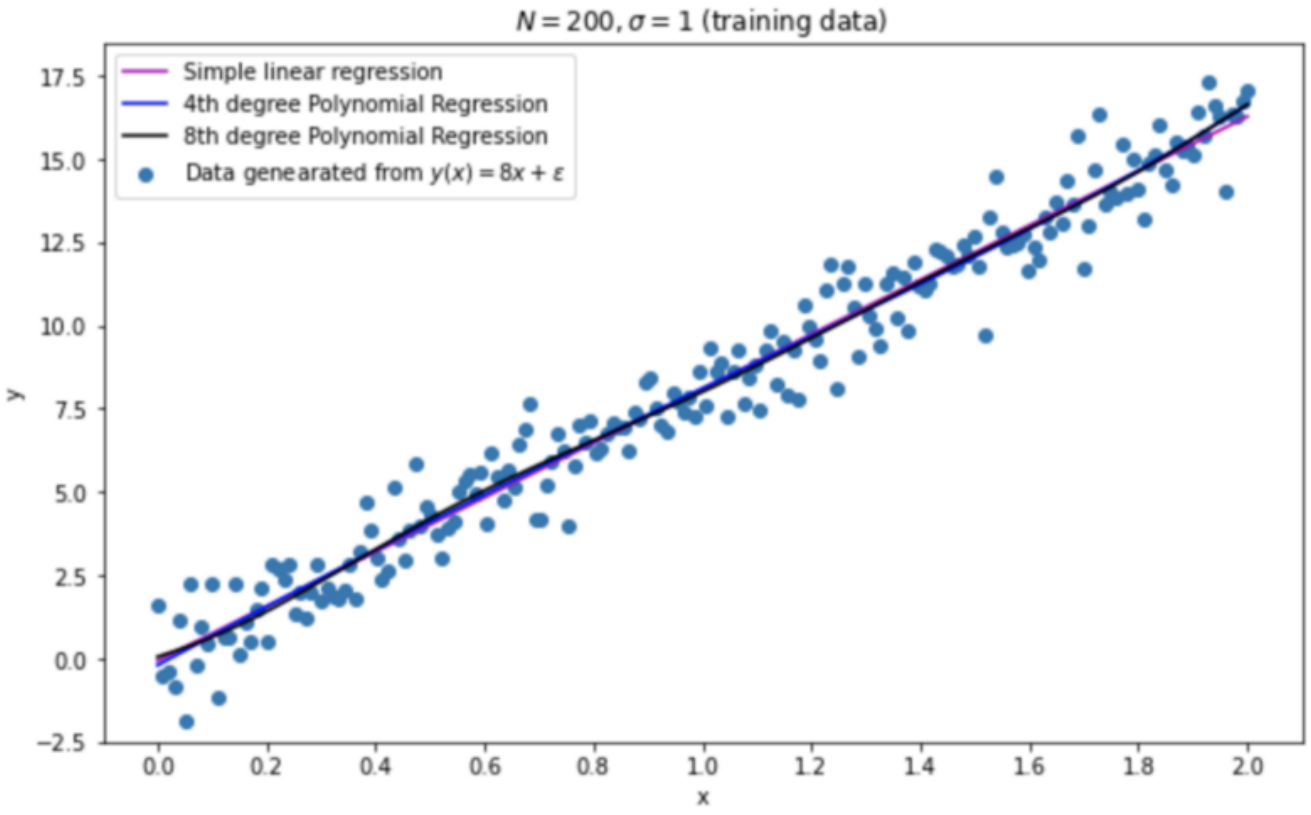


Fig. 4. Plots of the linear and polynomial functions for the training dataset for N = 200.

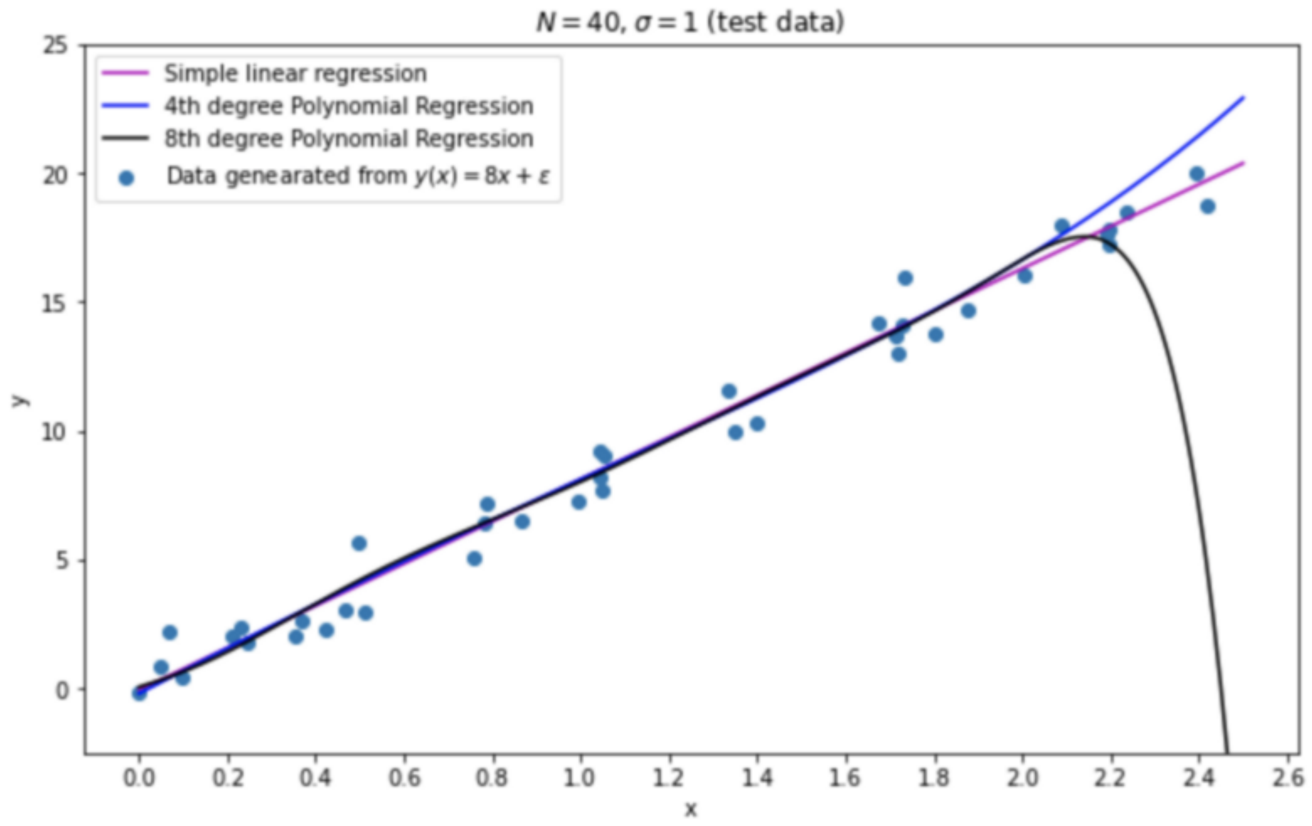


Fig. 5. Plots of the linear and polynomial functions for the test dataset for N = 40.

Interpretation of the of Bias-Variance error

Now it's time to interpret of the BV error in equation (1) with the help of the test data plots of Fig. 3 and Fig. 5. It is worth recalling that equation (1) has been derived for the test data, so it represents, on average, the test data *MSE,* and this is why I am focusing on the test data plots and not on training data plots. The BV equation (1) is valid for the test data. In my previous article, I defined the Bias squared and Variance as:

$$\text{Bias}^2[f(\boldsymbol{x};\bar{\boldsymbol{\theta}}_D)] = \sum_{i=1}^{N}\left[f(\boldsymbol{x}_i) - E_D\left(f(\boldsymbol{x}_i;\bar{\boldsymbol{\theta}}_D)\right)\right]^2, \qquad \text{Var}\left(f(\boldsymbol{x};\bar{\boldsymbol{\theta}}_D)\right) =$$

$$\sum_{i=1}^{N} E_D\left(\left[f(\boldsymbol{x}_i;\bar{\boldsymbol{\theta}}_D) - E_D\left(f(\boldsymbol{x}_i;\bar{\boldsymbol{\theta}}_D)\right)\right]^2\right) \qquad (2)$$

Let me focus on Fig. 3 and consider first the simple linear regression function *y_1(x)* and let me extract the intercept and slope with the following Python Code:

```
(slope, intercept) = (model_1.coef_[0], model_1.intercept_)
slope, intercept
```

Python Code 7

In the case when the training data number is N=60 and N=200 as in Fig. 2 and Fig. 3, I get the following equations for *y_1(x) = f(x ; **θ_D**):*

$$\begin{aligned} y_1(x_i) &\simeq -0.23 + 8.24 x_i \qquad (N_{\text{train}} = 60) \\ y_1(x_i) &\simeq -0.06 + 8.17 x_i \qquad (N_{\text{train}} = 200) \qquad (3) \end{aligned}$$

In equation (3), one can see how the simple linear regression line changes with the number of training data. *More training data imply that the fitted y_1(x) is more closely to the true function y(x) = f(x).* As the number of training data increases, the intercept term of equation (3) tends to zero, and the slope tends to 8. In a similar way that led me to equation (3), one can also extract with a similar code to Python Code 7 the coefficients of the fourth and eighth polynomial regression lines that do not show here for simplicity.

By changing the training dataset and its number, the learned function *y_1(x)* changed its form but not that much as one can see from equation (3) and Fig. 2 and Fig. 3. This is a case when the function *y_1(x)* is said to have a ***variance error or more precisely a low variance error***. As one can see from the definition of variance in equation (2), it measures how much a "learned" function changes from its average value trained over many datasets. In the case of the simple linear regression, the learned function *y_1(x)* changed very little in its form (see equation (3)) when trained in different sized datasets. On the other hand, the function *y_1(x)* is not very close in form to the true function *f(x) = 8x* as one can see from equation (3). The learned function *y_1(x)* has acquired an intercept during the training procedure and still is not close to the true function even though its intercept value gets smaller as the training dataset increases. This is a situation when the learned function has a ***bias error.*** As one can see from the definition of bias in equation (2), it measures the difference between the true function and the expectation value of the learned function over many training datasets.

Now let me focus on the eighth-order polynomial function, *y_3(x),* which plots for different test datasets are shown in Fig. 3 and Fig. 5. As one can see the function *y_3(x)* changes its shape significantly as the size of training datasets changes. In this case, the function *y_3(x)* is said to have a ***high variance error***. In Fig. 3, one can see that *y_3(x)* tends to increases in value for higher predictor values and in Fig. 5, conversely, it tends to decrease in value for higher predictor values. In addition to having a high variance, the function *y_3(x)* has also a ***high bias*** since its form is not close to the true function *f(x) = 8x*.

One important thing to note is that the performance of each learned function on the test dataset strongly depends on the training predictor interval x. In fact, one can see from Fig. 3 and Fig. 5 that both y_2(x) and y_3(x) perform badly on predicting new values once outside the ***predictor training interval*** *[0, 2].*

Which of the functions *y_1(x), y_2(x),* and *y_3(x)* gives better prediction on the test data? Well, if you look at the Fig. 3, it is easy to see that *y_2(x)* and *y_3(x)* better fit the test data for N=12. One can see this by running the following Python code on the test data that calculates the *MSE* for each learned function:

```
MSE_y1 = mean_squared_error(y_test, model_1.predict(x_test.reshape(-1, 1)))
MSE_y2 = mean_squared_error(y_test, model_2.predict(poly4.fit_transform(x_test.reshape(-1, 1))))
MSE_y3 = mean_squared_error(y_test, model_3.predict(poly8.fit_transform(x_test.reshape(-1, 1))))
```

```
MSE_y1, MSE_y2, MSE_y3
```

Python Code 7.

I use Python Code 7 and get the following results for the test data *MSE* for the learned functions *y_1(x), y_2(x),* and *y_3(x)* for different training and test dataset sizes:

$$\left[MSE(y_1) \simeq 0.92, \quad MSE(y_2) \simeq 0.81, \quad MSE(y_3) \simeq 0.7\right], (N_{\text{test}} = 12)$$
$$\left[MSE(y_1) \simeq 0.64, \quad MSE(y_2) \simeq 1.01, \quad MSE(y_3) \simeq 9.67\right], (N_{\text{test}} = 40)$$
$$\left[MSE(y_1) \simeq 1.05, \quad MSE(y_2) \simeq 1.18, \quad MSE(y_3) \simeq 9.74\right], (N_{\text{test}} = 200),$$

(4)

Calculations done by the author with Python Code 7.

You can see in equation (4) that as the number of the training, and consequently, test dataset increases, the simple linear regression gives the lowest MSE for test data number N=40 and N=200, while the eight-order polynomial regression function gives the largest MSE. *Why does y_1(x) gives the lowest MSE and thus the best prediction on test data as the test data number increases?* The answer to this question has to do with the variance and bias errors that I described above. The learned function *y_1(x),* indeed, has low variance and also a moderate bias because its form is linear as that of the true function. On the other hand, the functions *y_2(x)* and *y_3(x)* have high variance and low bias and thus higher test *MSE*.

Few words on model complexity

As I mentioned above, there is a relation between the model complexity and the total prediction error, bias error, and variance error. You can see this relation from Fig. 6 below, which I took from my previous article and re-post here again.

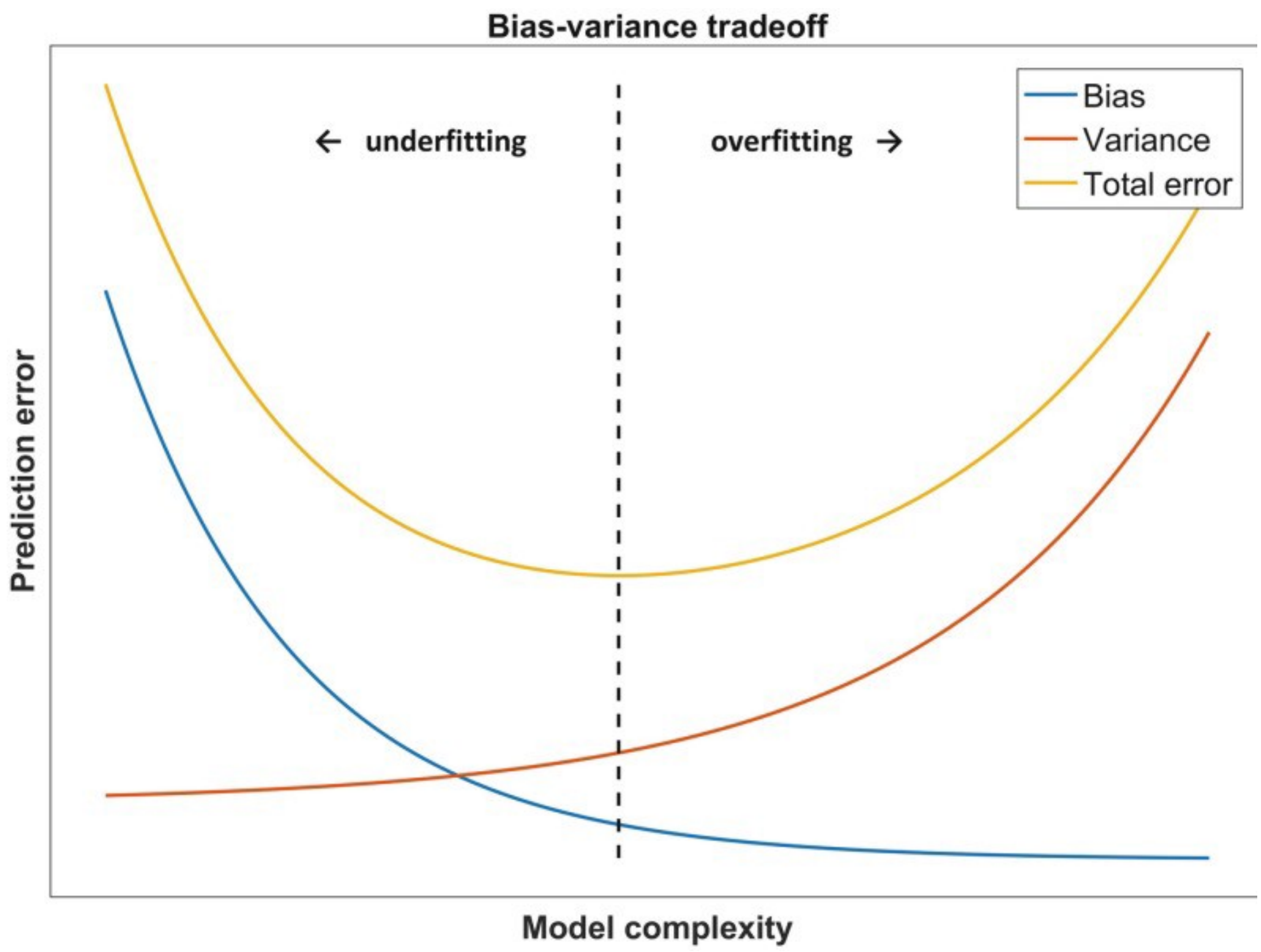


Fig. 6. A simple intuitive figure that represents the prediction (test data error) error as a function of the model complexity. The bias error decreases with model complexity while the variance error increases (Image credit from the book “Fundamentals of Clinical Data Science” under the terms of Creative Commons Attribution 4.0 International License.)

One might ask what does model complexity mean? It essentially means how complex is the fitted function on the training data of **fixed size**. The more parameters a fitted function has, the more complexity is said to have. For example, the function *y_1(x)* has a model complexity of 2, while *y_2(x)* has a model complexity of 5, and *y_3(x)* has a model complexity of 9. In general, a polynomial fitted function of degree *n* has a model complexity of *n+1*.

As you can see from Fig. 6, the bias decreases with the model complexity while the variance increases. In fact, as I showed above, the eighth degree fitted polynomial function *y_3(x)* has a high variance. The reason why complex functions (high in complexity) have high variance is due to the fact that during the training process, these functions tend to fit very well the training data *including the noise*. Complex models with many parameters, such as the eighth-order polynomial, can capture both the global trends and noise-generated patterns at the same time. On the other hand, the simple linear regression function *y_1(x)* ignores noise fluctuations and captures global trends and thus tends to change less in form with changing (randomly generated) training datasets.

Complex model functions, on the other hand, have less bias as the model complexity increases. The reason is due to the fact that these functions having more parameters, tend to adjust these parameters to the training data very well to match the underlying true function that generates the data. For example, the eighth-order polynomial *y_3(x)* considered above has less bias than the simple linear function *y_1(x)*. I recall that the bias measures the difference between the true function and the averaged learned function over many datasets, so, the better a function fits the training data, the closer is to the true function and the less bias it has.

Calculation of bias and variance errors in practice

If you want to calculate the errors associated with the bias and variance, here I outline the procedure to follow. I do not show this type of calculation in this article because it is lengthy, and the Python code would be as well. Said this, in order to calculate the bias and variance errors, you should focus on equation (2) and try to understand what it actually means. If you understand by yourself, then congratulations, if not then keep following.

The first thing to note is that in equation (2) does appear everywhere the expectation value of the learned function *f(x ;* ***θ_D****)* over the dataset *D*. What does this mean? It essentially means that to calculate the *E_D(f(x ;* ***θ_D****)),* one needs a large number of **training datasets**, theoretically an infinite number. Suppose we have a large number of training datasets, *D_M,* where M is a large positive number. Let each dataset have a very large number of labelled data *D_1 ={(x_1, y_1), …, (x_k, y_k)}_1, D_2 ={(x_1, y_1), …, (x_k, y_k)}_2, …, D_M ={(x_1, y_1), …, (x_k, y_k)}_M, where all datasets have equal (large) number of paired data points k.*

After partitioning the datasets, one chooses a model function to fit the training data on each dataset. After the function is learned on each of these datasets, we get in theory, different learned functions for each training dataset, namely we get a collection of learned functions, { *f(x ;* ***θ_1****), f(x ;* ***θ_2****),…, f(x ;* ***θ_M****) }*. Then to calculate the expectation value *E_D(f(x_i ;* ***θ_D****))* at some point *x_i,* one needs to take the *average value* of all learned functions *f(x ;* ***θ_M****)* at the point *x_i*. If each dataset is randomly generated and has an equal probability distribution function, then the average value of all learned functions is just the *arithmetical mean* of these functions at the point *x_i*. Once *E_D(f(x_i ;* ***θ_D****))* is calculated, then to find the bias and variance errors, one needs to use equation (2) and sum for all **test dataset points** *x_i*.

Conclusion

The bias and variance error equation (1) is valid for regression problems, for large numbers of data and large number of datasets. It is valid for the test data. The bias and variance errors depend on the model complexity and training predictor data interval. The model that better fits the training data does not necessarily give better predictions for the test data because of the variance issue.

If you liked my article, please share it with your friends that might be interested in this topic and cite/refer to my article in your research studies. Do not forget to follow me and subscribe for other related topics that will post in the future.

By Damian Ejlli on August 18, 2021.

Canonical link

Exported from Medium on October 8, 2021.