Train and test data

23 September 2022 13:20

Understanding Train and Test data

- The train-test split is a technique for evaluating the performance of a machine learning algorithm
- It can be used for any supervised learning algorithm (Classification or Regression)

The procedure involves taking a dataset and dividing it into **two subsets.**

1. First subset called **"Training dataset"**
2. Second subset called **"Test dataset"**

Train Dataset:

- Used to fit the machine learning model.

Test Dataset:

- Used to evaluate the fit machine learning model.
- i.e. the input element of the dataset is provided to the model, then predictions are made and compared to the expected values

Objective of Test Dataset:

- To estimate the performance of the machine learning model on new data(test data)
- This is how we expect to use the model in practice.
- Namely, to fit it on available data with known inputs and outputs,
- then make predictions on new examples in the future where we do not have the expected output or target values.

Underfitting and overfitting

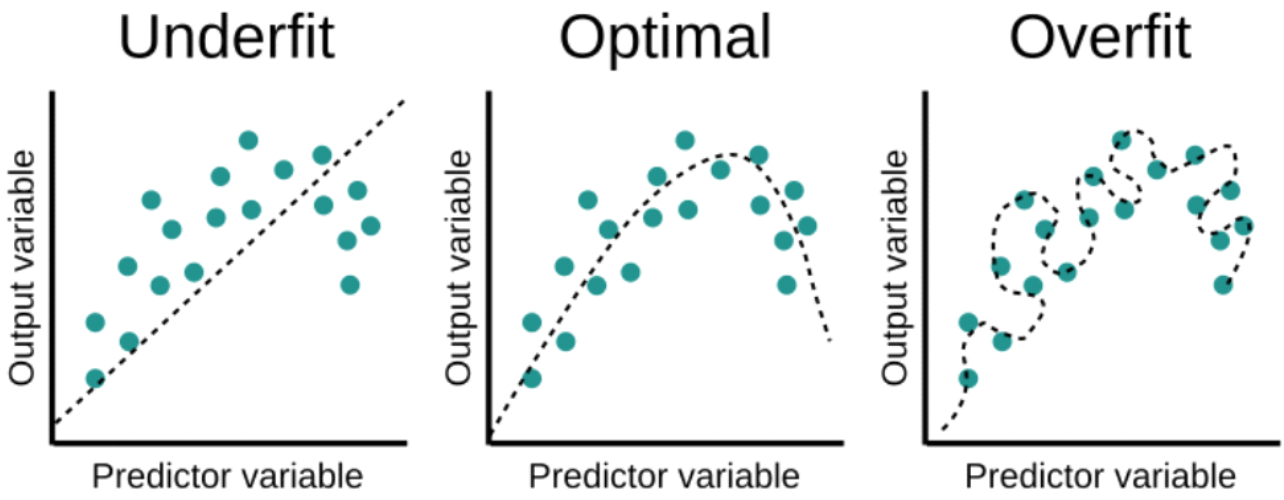
24 September 2022 14:28

Understanding underfitting and overfitting

- In Machine Learning and Pattern recognition, there are infinite ways to solve a problem.
- Thus it is important to have an objective criterion for assessing the **accuracy** of candidate approaches and for selecting the **right mode**l for a data set at hand.

Why we need to understand it?

- the concepts of under- and overfitting are related to the statistical quantities bias and variance.
- these concepts can be applied to select a model that will accurately generalize to data sets.



Note:

- Optimal fit or best fit
- Poor fir - which not actually depicting the pattern of the datapoints

Can a poor fit converted to best fit ?

- Yes by iterations
- That is the concept we learnt in "Gradient descent"

Why complex multiple (polynomial) regression is overfit?

- After getting the algorithm, it fails for the unknown dataset in future
- Where over-confidence always fails
- Also residuals will be increased as the polynomial increases

Note:

Instead of linear regression, polynomial regression mostly tends to give the overconfidence(over fitting)

Good balance:

Slim fitting:

Models for Regression

24 September 2022 14:36

In Regression analysis:

Q:	How would you characterize the change in dependent variable with changes in independent variable?
A:	We assume that there is some true **relationship function f(x)** that maps the independent variable values onto the dependent variable values

- Thus, we could like to determine the form of f(x) from the observation of independent and dependent variable.
- However, in the **real world**, we **don't** get to observe **f(x) directly**, but instead get **noisy** observations y , where

$$y=f(x)+\epsilon$$

ϵ - Assume it is random variable distributed according to a **zero**- mean **Gaussian** with standard deviation **σ2ϵ** or N(0,σ2ϵ)

ϵ is random variable hence,

y also a random variable (with a mean that is conditioned on both x and f(x) , and exhibiting a variance σ2ϵ .)

EXAMPLE:

The true function f(x) we want to determine has the following form (though we don't know it):

$$f(x)=\sin(\pi x)$$

Thus the observations y , we get to see have the following distribution,

$$y=\sin(\pi x)+N(0,\sigma 2\epsilon)$$

STEP 01: "Importing the Libraries"

```
# Importing the libraries
import numpy as np
import pandas as pd
from numpy import math

from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error

import matplotlib.pyplot as plt
```

STEP 02: "Define and Display the f(x)"

```
np.random.seed(123)
MARKER_SIZE = 100
DATA_COLOR = 'black'
ERROR_COLOR = 'darkred'
POLYNOMIAL_FIT_COLORS = ['orange', 'royalblue', 'darkgreen']
LEGEND_FONTSIZE = 14
TITLE_FONTISIZE = 16
N_OBSERVATIONS = 10
NOISE_STD = 1.

x = 2 * (np.random.rand(N_OBSERVATIONS) - .5)
x_grid = np.linspace(-1, 1, 100)


def f(x):
    """Base function"""
    return np.sin(x * np.pi)

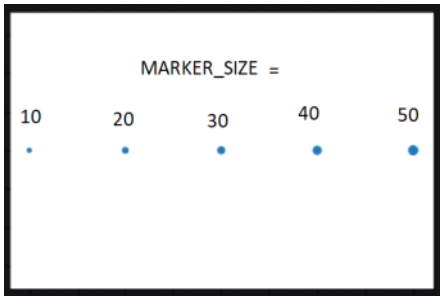

def sample_fx_data(shape, noise_std=NOISE_STD):
    return f(x) + np.random.randn(*shape) * noise_std

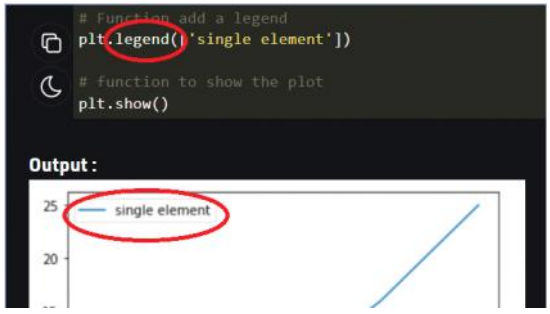

def plot_fx_data(y=None):
    """Plot f(x) and noisy samples"""
    y = y if y is not None else sample_fx_data(x.shape)
    fig, axs = plt.subplots(figsize=(6, 6))
    plt.plot(x_grid, f(x_grid), color=DATA_COLOR, label='f(x)')
    plt.scatter(x, y, s=MARKER_SIZE, edgecolor=DATA_COLOR, facecolors='none', label='y')
```

```
# Plot the data
y = sample_fx_data(x.shape)
plot_fx_data(y)
plt.legend(fontsize=14)
plt.title(f'$f(x) = sin(\pi x)$ and some observations, $y$', fontsize=16)
plt.xlim([-1, 1])
plt.ylim([-2, 2])
```

Explanation:

- `np.random.seed(123) #With the seed reset (every time), the same set of numbers will appear every time.`
- `MARKER_SIZE = 100`



- `DATA_COLOR = 'black'`
  `ERROR_COLOR = 'darkred'`
- `POLYNOMIAL_FIT_COLORS = ['orange', 'royalblue', 'darkgreen']`
  - As we planned to provide 3 types of polynomial to fit.
- `LEGEND_FONTSIZE = 14`
  - **Matplotlib.pyplot.legend()**
  - A legend is an area describing the elements of the graph
  - 

- `TITLE_FONTISIZE = 16`
- `N_OBSERVATIONS = 10`
  `NOISE_STD = 1.`
  - As we seek 10 observation
  - And providing noise value as "1"
- `x = 2 * (np.random.rand(N_OBSERVATIONS) - .5)`
  - Providing the value to "x"
  - From random observation values
  - We may get the value for x as,

```
[10] x
```

```
array([ 0.39293837, -0.42772133, -0.54629709,  0.10262954,  0.43893794,
       -0.15378708,  0.9615284 ,  0.36965948, -0.0381362 , -0.21576496])
```

- `x_grid = np.linspace(-1, 1, 100)`
  - Creating the x-axis grid with 100 values from -1 to 1
- ```
  def f(x):
      """Base function"""
      return np.sin(x * np.pi)
  ```
  - As we take f(x) as sin(pi*x)
  - True function
  - Output:

```
f(x)
```

```
array([ 0.94396758, -0.97433016, -0.98944128,  0.31686298,  0.98165657,
       -0.46455884,  0.12056807,  0.91732924, -0.11952198, -0.62711638])
```
```

```
def sample_fx_data(shape, noise_std=NOISE_STD):
    return f(x) + np.random.randn(*shape) * noise_std
```

- For the observation of y we have some distribution
- Hence **y=sin(πx)+N(0,σ2ϵ)**
- Here, shape is the number of dimension of the x (note: we provided with 100)

Output:

```
sample_fx_data(x.shape)

array([ 1.94802148, -0.58814376, -0.2520727 ,  1.80759501,  0.04582271,
        0.71127021, -1.1333126 ,  0.27957774,  0.78758322, -2.05579708])
```

```
x.shape

(10,)
```

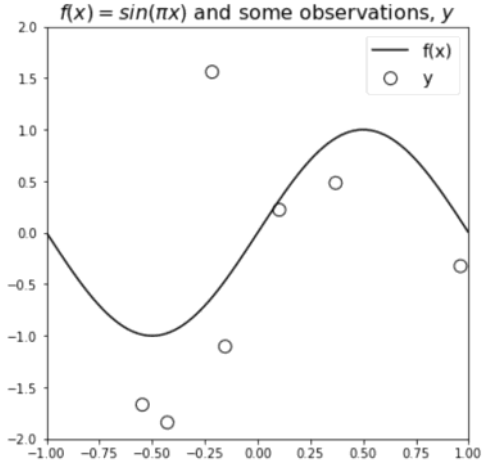
```
def plot_fx_data(y=None):
    """Plot f(x) and noisy samples"""
    y = y if y is not None else sample_fx_data(x.shape) # condition providing because in future we may have to tackle unknown dataset with "None value"
    fig, axs = plt.subplots(figsize=(6, 6)) #size of the graph
    plt.plot(x_grid, f(x_grid), color=DATA_COLOR, label='f(x)')  #Defining the curve (~)
    plt.scatter(x, y, s=MARKER_SIZE, edgecolor=DATA_COLOR, facecolors='none', label='y') #Defining the points (o)
```

- Just to plot the y
- We provide the value for y as "sample_fx_data(x.shape)

```
# Plot the data
y = sample_fx_data(x.shape)
plot_fx_data(y)

# To plot in good manner we need to add some input's to matlab
plt.legend(fontsize=14)
plt.title(f'$f(x) = sin(\pi x)$ and some observations, $y$', fontsize=16)
plt.xlim([-1, 1]) #x-limit
plt.ylim([-2, 2]) #y-limit
```

OUTPUT:



What is Our Goal?

- Our goal is to characterize the function f(x)

Did we know about the function form of f(x)?

- "No"

Solution to this **?**

- we will instead estimate some other function g(x) that we believe will provide an accurate approximation to f(x)
- In simple words:
  The function g(x) is called an estimator of f(x)

What is estimator ?

- an estimator is some parameterized model that can capture a wide range of functional forms.
- One such class of estimators is the weighted combination of ordered polynomials:
  - $g_D(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_D x^D$

Properties of polynomial:

- As the polynomial order D increases, the functions gD(x) are able to capture increasingly complex behaviour.
- For example,
  - g0(x) describes a horizontal line with an adjustable vertical offset θ0 ,
  - g1(x) describes a line with adjustable vertical offset θ0 and adjustable linear slope θ1 ,
  - g2(x) describes a function that also includes a weight on the quadratic term θ2 .

We thus try to fit the values of the parameters for a given estimator gD(x) to best account for observed data in the hopes that we will also accurately approximate f(x) .

Below we estimate the parameters of three polynomial model functions of increasing complexity (using NumPy's polyfit) to the sampled data displayed above. Specifically, we estimate the functions g1(x) , g3(x) and g10(x)

```
plot_fx_data(y)

polynomial_degrees = [1, 3, 10]
theta = {}
fit = {}
for ii, degree in enumerate(polynomial_degrees):
    # Note: we should get an overconditioned warning for degree 10 because of extreme overfitting
    theta[degree] = np.polyfit(x, y, degree)
    fit[degree] = np.polyval(theta[degree], x_grid)
    # plt.figure(figsize=(10,10))
    plt.plot(x_grid, fit[degree], POLYNOMIAL_FIT_COLORS[ii], label=f"$g_{degree}(x)$")

plt.legend(fontsize=LEGEND_FONTSIZE)
plt.xlim([-1, 1])
plt.ylim([-2, 2])
plt.title("Various Polynomial Functions Fit to Observations", fontsize=TITLE_FONTISIZE)
```

Explanation:

```
plot_fx_data(y)

polynomial_degrees = [1, 3, 10]
theta = {}
fit = {}
for ii, degree in enumerate(polynomial_degrees): #enumarate gives (0,1) (1,3) (2,10)
    # Note: we should get an overconditioned warning for degree 10 because of extreme overfitting
```

- Created a list of degree in variable name as "polynomial degree"
- Empty dictionary as theta and fit

```
theta[degree] = np.polyfit(x, y, degree)
```

What is **numpy.polyfit ?**

Fit a polynomial `p(x) = p[0] * x**deg + ... + p[deg]` of degree *deg* to points *(x, y)*. Returns a vector of coefficients *p* that minimises the squared error in the order *deg*, *deg-1*, ... *0*.

- Using "polyfit" function we will get array of theta for each iteration (each degree)
- Note: We can also use the "linear regression" instead of polyfit
- Example:

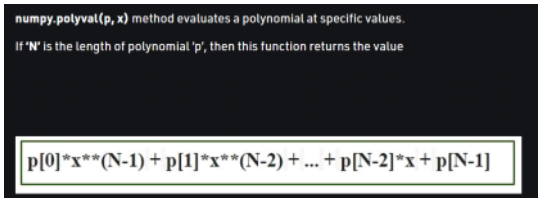
  Theta[degree] for last iteration, which means it will provided 10 values of theta

  ```
  theta[degree]
  ```

  ```
  array([ 1.00166969e+04, -3.94690600e+04,  5.75983608e+04, -2.03579919e+03,
         -3.78275489e+04,  6.90778623e+03,  6.83524551e+03, -9.91765997e+02,
         -3.61729333e+02,  1.25555384e+01,  3.02252451e+00])
  ```

```
fit[degree] = np.polyval(theta[degree], x_grid)
```

What is **numpy.polyval?**



Output: As x grid we provided 100

```
fit[degree]
```

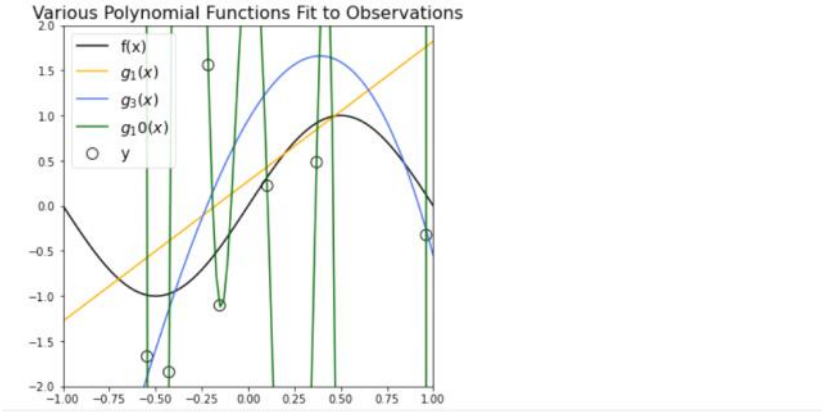
```
array([ 7.18403309e+04,  5.88785955e+04,  4.79724727e+04,  3.88572453e+04,
        3.12707116e+04,  2.49897224e+04,  1.98190008e+04,  1.55882276e+04,
        1.21493742e+04,  9.37426772e+03,  7.15237110e+03,  5.38876448e+03,
        4.00231332e+03,  2.92400978e+03,  2.09547482e+03,  1.46760880e+03,
        9.99379464e+02,  6.56736254e+02,  4.11641025e+02,  2.41205776e+02,
        1.26927841e+02,  5.40147662e+01,  1.07905380e+01, -1.18241202e+01,
       -2.07643288e+01, -2.12129954e+01, -1.69381522e+01, -1.05813027e+01,
       -3.98198709e+00,  2.01660727e+00,  6.59653178e+00,  9.62434892e+00,
        1.11370141e+01,  1.13890953e+01,  1.04908380e+01,  8.93618681e+00,
        6.98150286e+00,  4.90926965e+00,  2.95267751e+00,  1.28652207e+00,
        2.45419909e-02, -7.78291549e-01, -1.11983709e+00, -1.04232867e+00,
       -6.22377761e-01,  3.97439557e-02,  8.30852189e-01,  1.63506635e+00,
        2.34304244e+00,  2.85988601e+00,  3.11149046e+00,  3.04914238e+00,
        2.65230781e+00,  1.92958295e+00,  9.17853597e-01, -3.20239577e-01,
       -1.70038687e+00, -3.12206470e+00, -4.47514543e+00, -5.64742068e+00,
       -6.53277588e+00, -7.03973783e+00, -7.10010586e+00, -6.67737026e+00,
       -5.77461880e+00, -4.44163250e+00, -2.78087564e+00, -9.52091136e-01,
        8.24778928e-01,  2.26861747e+00,  3.03835922e+00,  2.73666526e+00,
        9.16276868e-01, -2.91073351e+00, -9.26060978e+00, -1.86622668e+01,
       -3.16381727e+01, -4.86815890e+01, -7.02300095e+01, -9.66346419e+01,
       -1.28125783e+02, -1.64773937e+02, -2.06446523e+02, -2.52760015e+02,
       -3.03027335e+02, -3.56200331e+02, -4.10807106e+02, -4.64883983e+02,
       -5.15981837e+02, -5.60686477e+02, -5.95332738e+02, -6.15111874e+02,
       -6.14371783e+02, -5.86429538e+02, -5.23455609e+02, -4.16349078e+02,
       -2.54603060e+02, -2.61594257e+01,  2.82748172e+02,  6.87764119e+02])
```

OUTPUT:

```
    # plt.figure(figsize=(10,10))
    plt.plot(x_grid, fit[degree], POLYNOMIAL_FIT_COLORS[ii], label=f"$g_{degree}(x)$")

plt.legend(fontsize=LEGEND_FONTSIZE)
plt.xlim([-1, 1])
plt.ylim([-2, 2])
plt.title("Various Polynomial Functions Fit to Observations", fontsize=TITLE_FONTISIZE)
```

```
Text(0.5, 1.0, 'Various Polynomial Functions Fit to Observations')
```



Conclusion of this output:

- Qualitatively, we see that the estimator g1(x) (**orange** line) provides a poor fit to the observed data, as well as a poor approximation to the function f(x) (black curve).
- We see that the estimator g10(x) (**green** curve) provides a very accurate fit to the data points, but varies wildly to do so, and therefore provides an inaccurate approximation of f(x) .
- Finally, we see that the estimator g3(x) (**blue** curve) provides a fairly good fit to the observed data, and a much better job at approximating f(x) .

- Our original goal was to approximate f(x) , not the data points per se.
- Therefore g3(x) , at least qualitatively, provides a more desirable estimate of f(x) than the other two estimators.
- The fits for g1(x) and g10(x) are examples of "**underfitting**" and "**overfitting**" to the observed data, respectively:
    - Underfitting occurs when an estimator g(x) is not flexible enough to capture the underlying trends in the observed data.
    - Overfitting occurs when an estimator is too flexible, allowing it to capture illusory trends in the data. These illusory trends are often the result of the noise in the observations y .

Bias and Variance of an Estimator

24 September 2022 17:31

- The model previously discussed fits for gD(x) discussed above were based on a single, randomly-sampled data set of observations y .
- However, because ϵ is a random variable, there are in principle a potentially infinite number of random data sets that can be observed.
- In order to determine a good model of f(x) , it would be helpful to have an idea of how an estimator will perform on any or all of these potential datasets.
- To get an idea of how each of the estimators discussed above performs in general we can repeat the model fitting procedure for many data sets.

Example:

Here we perform such an analyses, sampling ***50 independent data sets***, then fitting the parameters for the polynomial functions of model order **D=(1,3,10)** to each dataset.

Note: "defaultdict"
The functionality of both **dictionaries** and **defaultdict** are almost same except for the fact that defaultdict never raises a Key Error. It provides a default value for the key that does not exists.

```
from collections import defaultdict

n_simulations = 50  #Taking sample of 50 independent data set
simulation_fits = defaultdict(list)
for sim in range(n_simulations):
    # Start from same samples
    y_simulation = sample_fx_data(x.shape)    #y= f(x)
    for degree in polynomial_degrees[:-1]:
        # Note: we should get an overconditioned warning
        # for degree 10 because of extreme overfitting
        # hence we considering only degree [1,3]
        theta_tmp = np.polyfit(x, y_simulation, degree)
        simulation_fits[degree].append(np.polyval(theta_tmp, x_grid))

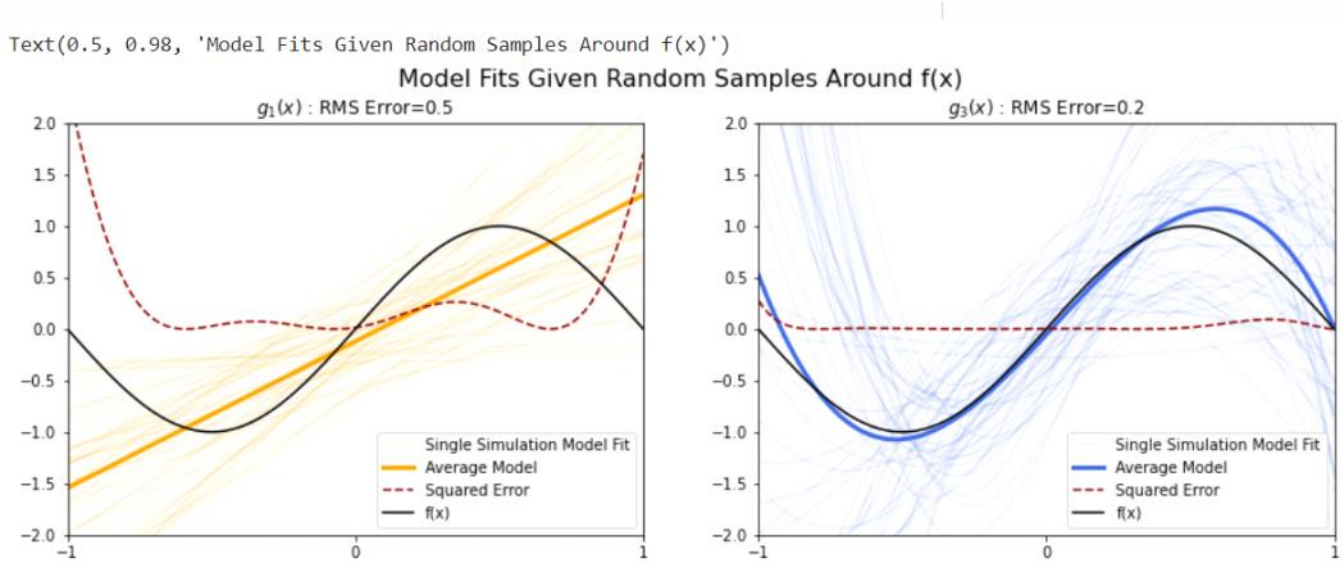

def error_function(pred, actual):
    return (pred - actual) ** 2
```

```
fig, axs = plt.subplots(1, 2, figsize=(15, 5))
for ii, degree in enumerate(polynomial_degrees[:-1]):
    plt.sca(axs[ii])
    for jj, fit in enumerate(simulation_fits[degree]):
        label = 'Single Simulation Model Fit' if jj == 0 else None
        plt.plot(x_grid, fit, color=POLYNOMIAL_FIT_COLORS[ii], alpha=.1, label=label)
    average_fit = np.array(simulation_fits[degree]).mean(0)
    squared_error = error_function(average_fit, f(x_grid))
    rms = np.sqrt(np.mean(squared_error))
    plt.plot(x_grid, average_fit, color=POLYNOMIAL_FIT_COLORS[ii], linewidth=3, label='Average Model')
    plt.plot(x_grid, squared_error, '--', color=ERROR_COLOR, label='Squared Error')
    plt.plot(x_grid, f(x_grid), color='black', label='f(x)')


    plt.xlim([-1, 1])
    plt.ylim([-2, 2])
    plt.xticks([-1, 0, 1])
    plt.title(f"$g_{degree}(x)$ : RMS Error={np.round(rms, 1)}")
    plt.legend(loc='lower right')
plt.suptitle('Model Fits Given Random Samples Around f(x)', fontsize=TITLE_FONTISIZE)
```

Output:

```
Text(0.5, 0.98, 'Model Fits Given Random Samples Around f(x)')
```

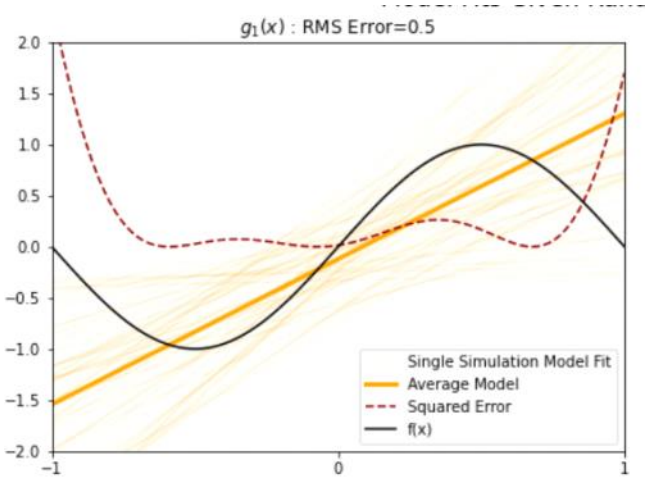


Observation:

- The lightly-coloured curves in each of the two plots (degree 1,3) above are an individual polynomial model fit to one of the 50 sampled data sets.
- The darkly-coloured curve in each plot is the average over the 50 individual fits.
- The dark curve is the true, underlying function f(x)

Estimator Bias

24 September 2022 19:48



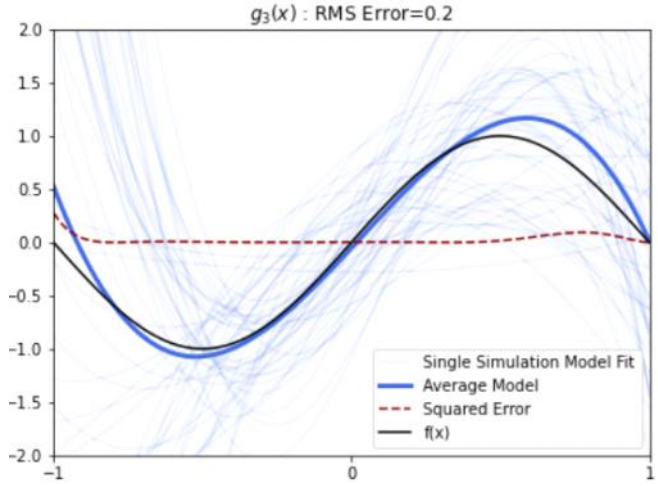
- We see that for the **estimator g1(x)** (light orange curves), model fits do not vary too dramatically from data set to data set.
- Thus the **averaged estimator fit** over all the data sets (dark orange curve), formally written as **E[g(x)]** , is similar (in terms of slope and vertical offset) to each of the individual fits.

- A commonly-used statistical metric that tries to assess the **average accuracy of an estimator g(x)** at approximating a **target function f(x)** is what is called the **bias of the estimator**. Formally defined as:

$$\text{bias} = E[g(x)] - f(x)$$

- The bias describes how much the average estimator fit over many datasets E[g(x)] deviates from the value of the actual underlying target function f(x) .

- We can see from the plot for g1(x) that E[g1(x)] deviates significantly from f(x) . Thus we can say that the estimator g1(x) exhibits **large bias** when approximating the function f(x) .



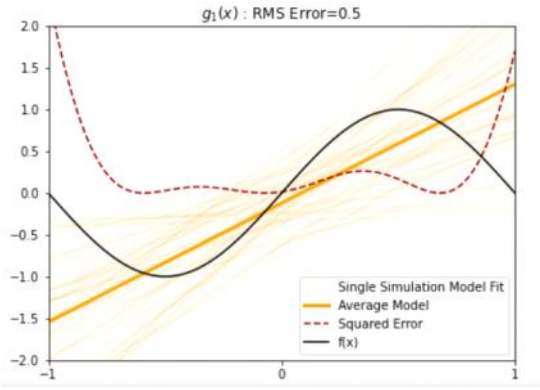
- When averaging over the individual fits for the **estimator g3(x)** (blue curves), we find that the average estimator E[g3(x) (dark blue curve) accurately approximates the true function f(x) , indicating that the estimator g3(x) has **low bias**.

Estimator Variance

24 September 2022 19:48

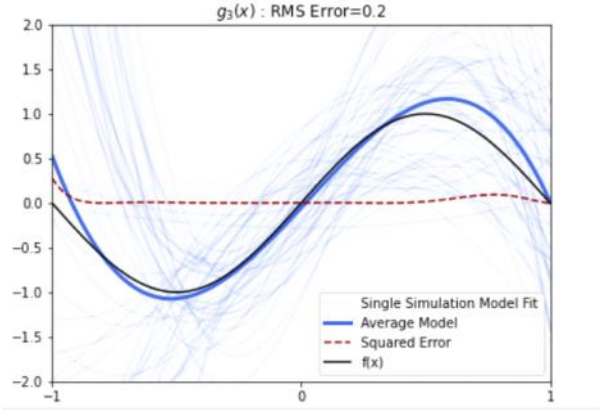
- Another common statistical metric attempts to capture the **average consistency** of an estimator when fit to multiple datasets.
- This metric, referred to as the variance of the estimator is formally defined as variance = **E[(g(x)—E[g(x)])2]**

- The variance is the expected (i.e. average) squared difference between any single dataset-dependent estimate of g(x) and the average value of g(x) estimated over all datasets, E[g(x)]



- According to the definition of variance, we can say that the **estimator g1(x)** exhibits **low variance** because the each individual g1(x) is fairly similar across datasets.

- Investigating the results for the **estimator g10(x)** (green curves), we see that each individual model fit varies dramatically from one data set to another. Thus we can say that this estimator exhibits **high variance.**



- We established earlier that the **estimator g3(x)** provided a qualitatively better fit to the function f(x) than the other two polynomial estimators for a single dataset.
- It appears that this is also the case over many datasets.
- We also find that estimator g3(x) exhibits **low bias and low variance**,

whereas the other two, less-desirable estimators, have either high bias or high variance.

- Thus it would appear that having both low bias and low variance is a **reasonable criterion for selecting an accurate model of f(x)** .

Expected Prediction Error and the Bias-variance Trade-off

24 September 2022 19:49

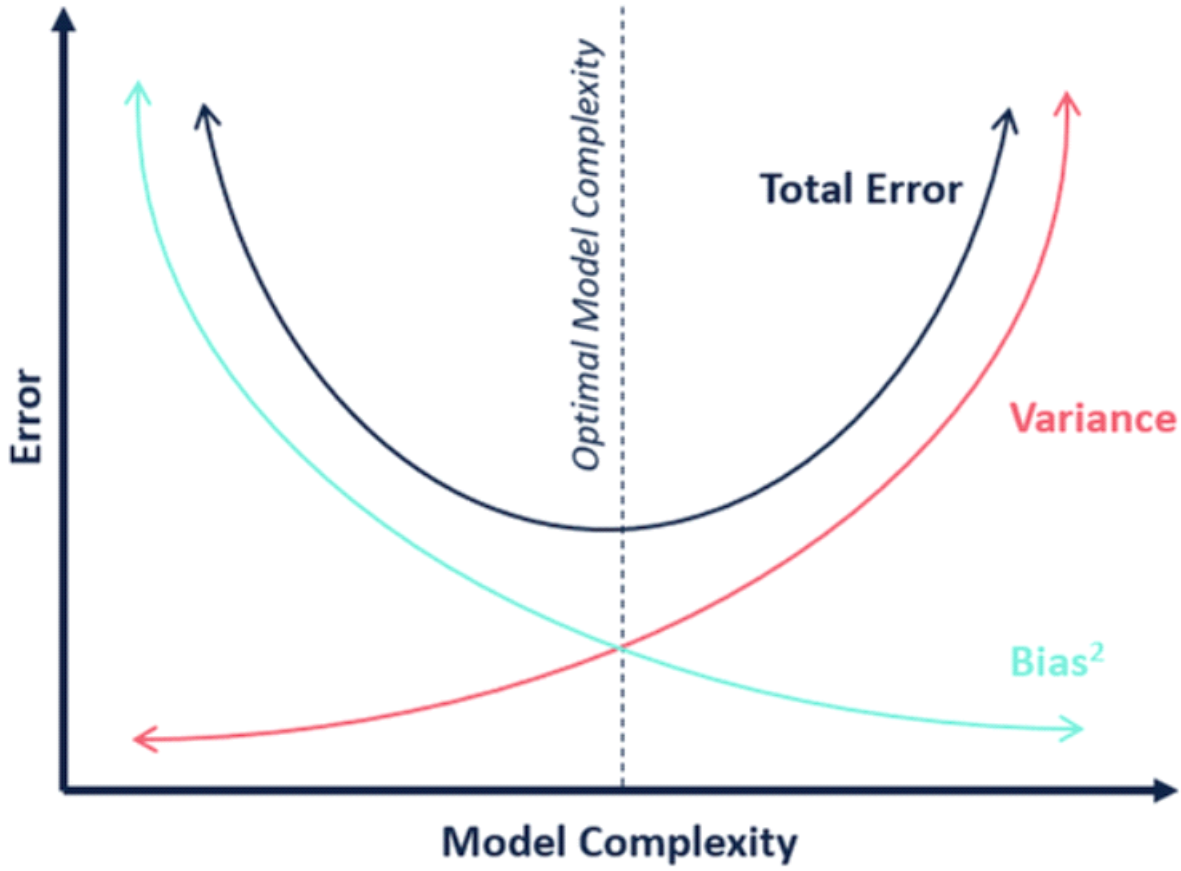
- For a given **estimator g(x)** fit to a data set of **x—y pairs** , we would like to know, given all the possible datasets out there,
- **what is the expected prediction error** we will observe for a new data point **$x*$, $y*=f(x)+\epsilon$** .
- If we define prediction error to be the squared difference in model prediction $g(x*)$ and observations $y*$, the expected prediction error is then:

$$E[(g(x*)-y*)2] = E[(g(x*)-E[g(x*))2] + (E[(g(x*)]-f(x*))2 + E[(y*-f(x*))2]$$

Error = Variance + Bias2 + Irreducible error

Where,

  - The first term is the **variance** of the estimator introduced above.
  - The second term is the **squared bias** of the estimator, also introduced above.
  - The third term is the **variance of the observation noise** and describes how much the observations y vary from the true function f(x) .
    - Notice that the noise term does not depend on the estimator g(x) .
    - This means that the **noise term** is a constant that places a lower bound on expected prediction error, and in particular is equal to the variance the noise term $\sigma 2\epsilon$



Commenting on above diagrammatic representation:

- When the model complexity increases,
  - It decreases the bias
  - It increases the variance and
  - Total error initially decreases with increase in model complexity until reaching the optimal model complexity, then tends to increase as the model complexity increases.

Demonstration of the Bias-variance Tradeoff

24 September 2022 20:04

Reference link:

We already discussed about the "Boston Housing dataset" in T-02 "Introduction to Linear regression". Let we further work on the same dataset for fitting the polynomial regression on this dataset with degree 1 to 10.

So far the dataset that pre-processed as,

```
new_df.head()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
1	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
2	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
3	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2
4	0.02985	0.0	2.18	0.0	0.458	6.430	58.7	6.0622	3.0	222.0	18.7	394.12	5.21	28.7

Now we define a new function to create the polynomial regression model.

```
def create_polynomial_regression_model():
  "creating the polynomial regression model for the given degree"
  fig, axs = plt.subplots(5,3, figsize = (15, 30))

  # j and k is for iterating through axis in subplot

  j=0
  k=-1

  for i in independent_variable:
    #setting title for the plot
    title = i

    #creating the independent variable data
    X = new_df[i].values
    X = np.array(X).reshape((len(X), 1))

    #creating the dependent variable data
    y = new_df[dependent_variable].values


    # Splitting the data into the train and test
    X_train, X_test, y_train, y_test = train_test_split( X, y, test_size = 0.2, random_state = 0)


    degree = 1
    train_error_list = [] #Storing training error for each degree
    test_error_list = []
    degree_list = [1,2,3,4,5,6,7,8,9,10] # Storing all the degree
```

```
  while degree <= 10:
    # Defining the degree for the polynomial feature
    poly_features = PolynomialFeatures( degree = degree)

    # Transforming the existing feature into the higher degree polynomial feature.
    X_train_poly = poly_features.fit_transform(X_train)

    y_test_poly = poly_features.fit_transform(X_test)

    # Fit the Transformer features into the Linear Regression
    poly_model = LinearRegression()
    poly_model.fit(X_train_poly, y_train)

    # Predicting the training data_set for both y_train and y_test
    y_train_predicted = poly_model.predict(X_train_poly)
    y_test_predicted = poly_model.predict(y_test_poly)

    # Evaluating the model on training the dataset
    rmse_train = np.sqrt(mean_squared_error(y_train, y_train_predicted))
    train_error_list.append(rmse_train)

    # Evaluating the model on test dataset
    rmse_test = np.sqrt(mean_squared_error(y_test, y_test_predicted))
    test_error_list.append(rmse_test)
```

```
    #Providing the values for the degree to iterate
    degree = degree + 1
  # Updating the value for the j and k for the subplot
  k = k + 1
  if k> 2:
    k = 0
    j = j + 1
  axs[j,k].plot(degree_list,train_error_list,color='green',label='Train_Error')
  axs[j,k].plot(degree_list,test_error_list,color='red',linestyle='--',linewidth=1.0,label='Test_Error')
  axs[j,k].set_title(title)
  axs[j,k].legend()
for ax in axs.flat:
  ax.set(xlabel='Model Complexity', ylabel='RMSE')
```

Detailed Step-by-Step Explanation:

1.

What is **fig, axs = plt.subplots()**

- plt.subplots() is a function that returns a tuple containing a figure and axes object(s). Thus when using fig, ax = plt.subplots() you unpack this tuple into the variables fig and ax.

```
fig, ax = plt.subplots()
```

- is more concise than this:

```
fig = plt.figure()
ax = fig.add_subplot(111)
```

```
fig, axs = plt.subplots(5,3, figsize = (15, 30))
```

- Which create 15 graphs in total with 5 rows and 3 columns as below,

2.

```
for i in independent_variable:
```

- Creating the for loop to iterate in every independent variables
- From that to create the graph **"Model complexity" vs "RMSE"** for each independent variables.

3.

```
#setting title for the plot
title = i
```

- As each graph with its own title of independent variable

4.

```
#creating the independent variable data
X = new_df[i].values
X = np.array(X).reshape((len(X), 1))

#creating the dependent variable data
y = new_df[dependent_variable].values
```

- Creating the variable x and y for the values in the dataset

5.

```
# Splitting the data into the train and test
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size = 0.2, random_state = 0)
```

6.

```
degree = 1
train_error_list = [] #Storing training error for each degree
test_error_list = []
degree_list = [1,2,3,4,5,6,7,8,9,10] # Storing all the degree

while degree <= 10:
```

- Before coding the while loop
- Ensuring the variables for the degree and empty list for train/test.

7.

```
while degree <= 10:
  # Defining the degree for the polynomial feature
  poly_features = PolynomialFeatures( degree = degree)
```

- Creating a new variable for polynomial feature
- This feature will tend to implement for each degree 1 to 10.

8.

```
# Transforming the existing feature into the higher degree polynomial feature.
X_train_poly = poly_features.fit_transform(X_train)

y_test_poly = poly_features.fit_transform(X_test)
```

- Transforming to polynomial

9.

```
# Fit the Transformer features into the Linear Regression
poly_model = LinearRegression()
poly_model.fit(X_train_poly, y_train)
```

- Fitting the polynomial mode to Linear regression
- Instead of the "poly fit"

10.

```
# Predicting the training data_set for both y_train and y_test
y_train_predicted = poly_model.predict(X_train_poly)
y_test_predicted = poly_model.predict(y_test_poly)
```

11.

```
# Evaluating the model on training the dataset
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_predicted))
train_error_list.append(rmse_train)

# Evaluating the model on test dataset
rmse_test = np.sqrt(mean_squared_error(y_test, y_test_predicted))
test_error_list.append(rmse_test)
```

12.

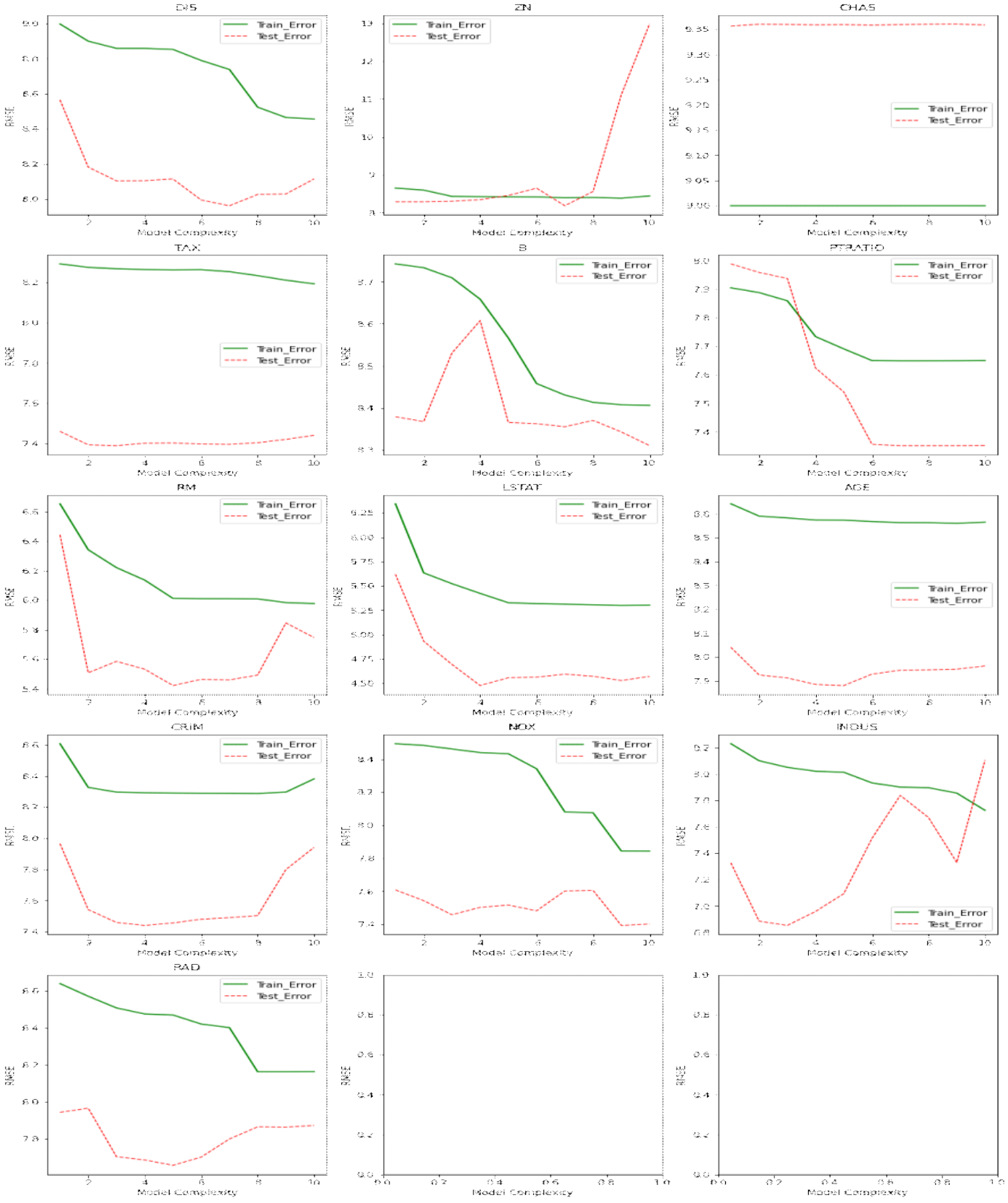
```
#Providing the values for the degree to iterate
degree = degree + 1
```

- At the end of the while loop, increase the value of degree for iteration to continue

13.

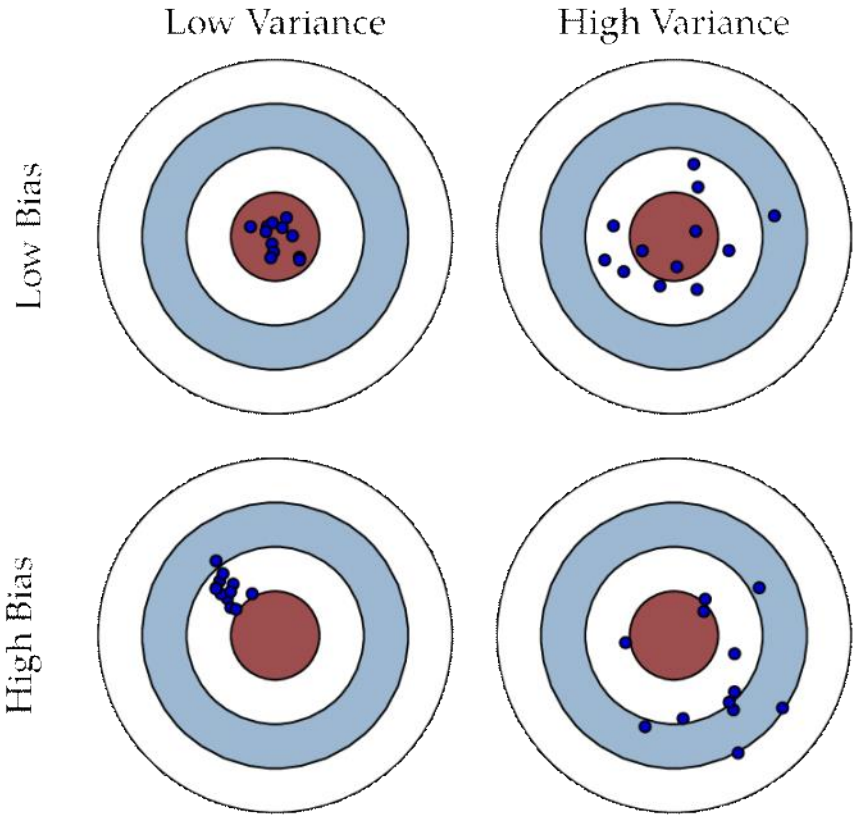
```
    # Updating the value for the j and k for the subplot
    k =  k + 1
    if k> 2:
      k = 0
      j = j + 1
    axs[j,k].plot(degree_list,train_error_list,color='green',label='Train_Error')
    axs[j,k].plot(degree_list,test_error_list,color='red',linestyle='--',linewidth=1.0,label='Test_Error')
    axs[j,k].set_title(title)
    axs[j,k].legend()
  for ax in axs.flat:
    ax.set(xlabel='Model Complexity', ylabel='RMSE')
```

Output:


DIS
ZN
CHAS
TAX
B
PTRATIO
RM
LSTAT
AGE
CRIM
NOX
INDUS
RAD
Train_Error
Test_Error
RMSE
Model Complexity

Conclusion

04 October 2022 14:43



What we discussed so far,

- Is how the bias and variance of an estimator are related to squared prediction error on the testing set.
- Though we focused on regression, these concepts can also be applied to classification problems.

Major findings,

- We found that an optimal estimator will have both low variance and low bias.
- We further found that information about squared bias and variance is contained in expected prediction error calculated on a testing set of data not used to fit a model's parameters.

Limitations,

- The concepts of estimator bias and variance are generally only clear in the context of an ensemble of datasets.
- However, in real-world applications, there is generally only a single observed dataset.
- In such cases the roles of bias and variance are less obvious (though, it is possible to calculate estimates of variance and bias using **resampling methods** such as **bootstrapping**).

However,

- the direct connection we made between bias, variance with the mean-squared error calculated on a testing set give us a direct means for assessing a group of candidate estimators in light of a single data set.
- We only need to partition the available data set into a Training Set used to fit model parameters and a Testing Set used to assess prediction accuracy.
- Comparing prediction accuracy across potential estimators is equivalent to assessing biases and variances of the estimators across many datasets.
- Note that resampling methods such as cross-validation can prove helpful here, particularly when the amount of observed data is small.