Understanding Overfitting and Underfitting

When building machine learning models, one of the primary challenges is to ensure that the model generalizes well to new, unseen data. We can measure how well the model generalises using independent Test set. Some of the evaluation metrics that can be used to measure performance on test set are :

- prediction accuracy
- mis-classification error

Simply we say that a good model has -

- high generalization accuracy
- low generalization error

Now, overfitting and underfitting are two terms that we can use to diagnose a machine learning model based on the training and test set performance

Overfitting

Overfitting occurs when a model learns the training data too well, capturing noise and details that do not generalize to unseen data. This usually happens when the model is too complex, such as having too many parameters relative to the number of observations.

If we say in very simple and concise language : Overfitting occurs when model starts fitting the noise. It thinks noise also to be an important structure of data that needs to get modelled. This happens because the model is too complex(more parameters than required) for data.

Indicators of Overfitting

- **Low Training Error, High Test Error**
- **High Training Accuracy, Low Test Accuracy**
- **The model performs exceptionally well on training data but poorly on validation/test data**
- **Complex Model**: The model may have large number of parameters. For eg: Higher order polynomial fitting simple data.

Visual Example

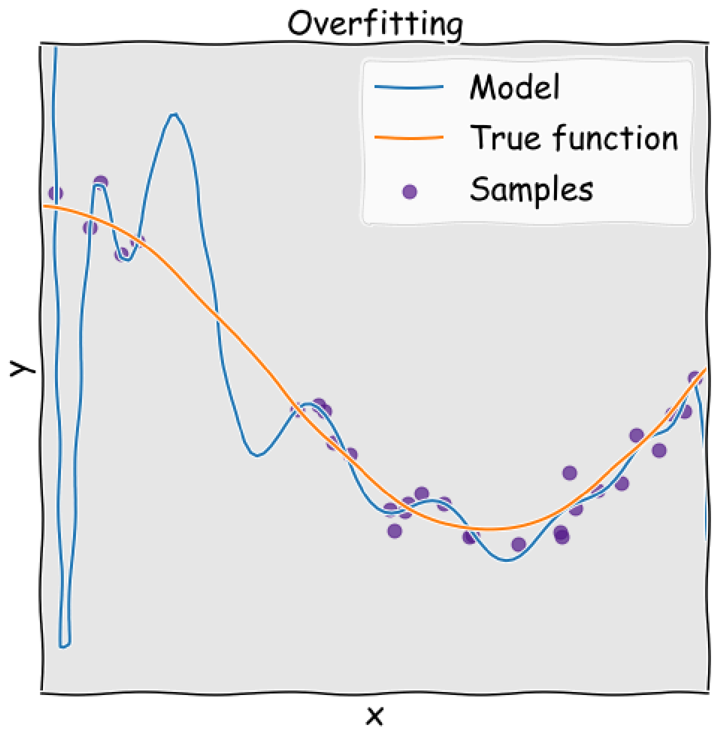


Image Source : keeto.github.io

How to Address Overfitting

1. **Simplify the Model**: Reduce the complexity of the model by decreasing the number of features or parameters.
2. **Regularization**: Techniques like L1 or L2 regularization can penalize large coefficients, helping to prevent overfitting.
3. **More Data**: Increasing the size of the training dataset can help the model to generalize better.

Underfitting

What is Underfitting?

Underfitting occurs when a model is too simple to capture the underlying structure of the data. This usually happens when the model has too few parameters, making it unable to learn the patterns in the data.

If we say in very simple and concise language : Underfitting occurs when model starts almost everything as noise. It does not fit the actual structure for given data leave about noise. This happens when the model has less number of parameters so that it is not powerful enough to model given data's structure.

Indicators of Underfitting

- **High Training Error, High Validation/Test Error**
- **low Training Accuracy, Low Test Accuracy**
- **The model performs poorly on both training and validation/test data.**
- **Simple Model**: The model may have too few features or parameters, making it incapable of capturing complexities in the data.

Visual Example

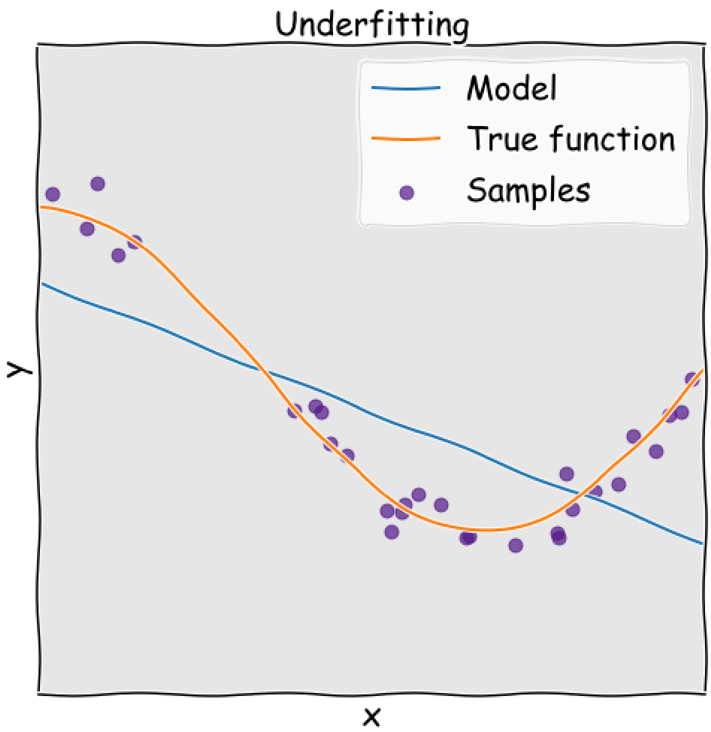


Image Source : keeto.github.io

How to Address Underfitting

1. **Increase Model Complexity**: Add more features or parameters to the model.
2. **Feature Engineering**: Create new features that can help capture the underlying patterns in the data.
3. **Reduce Bias**: Use techniques to reduce the bias in the model.

Finding the Right Balance

The goal is to find a balance between overfitting and underfitting, which is often referred to as the bias-variance tradeoff. **Bias** refers to errors due to overly simplistic models, while **variance** refers to errors due to overly complex models.

In [ ]:
```
# Numpy and pandas as usual
import numpy as np
import pandas as pd
```

```
# Scikit-Learn for fitting models
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import make_pipeline

# For plotting in the notebook
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

# Default parameters for plots
matplotlib.rcParams['font.size'] = 12
matplotlib.rcParams['figure.titlesize'] = 16
matplotlib.rcParams['figure.figsize'] = [8, 6]

import warnings
warnings.filterwarnings("ignore")
```

Data Generation for Polynomial Regression Example

we generate the dataset that will be used to demonstrate the concepts of overfitting and underfitting in polynomial regression. We set the random seed to 42 using `np.random.seed(42)` to ensure reproducible results. This allows us to generate the same random numbers each time the code is run.

Underlying distribution or Generating Function:

- In our case, the underlying function from which data comes is defined as a sine function: (y = \sin(2\pi x)).
- We generate 120 `x` values uniformly distributed between 0 and 1, sorted in ascending order.
- The corresponding `y` values are calculated using the true generating function and then a small amount of Gaussian noise is added.

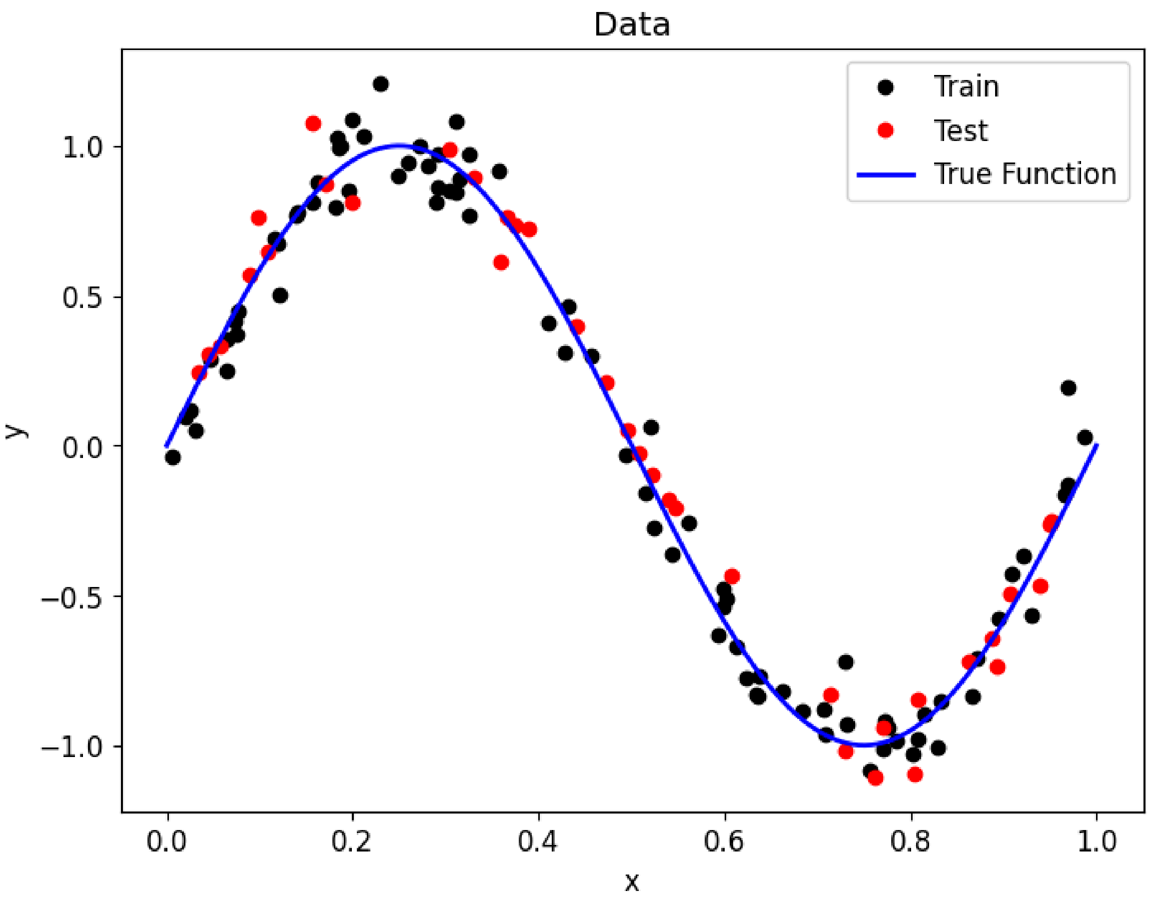
In [ ]:
```
np.random.seed(42)

# "True" generating function representing a process in real life
# def true_gen(x):
#     y = np.sin(1.2 * x * np.pi)
#     return(y)

def true_gen(x):
    y = np.sin(2 * np.pi * x)
    return y

x = np.sort(np.random.rand(120))
y = true_gen(x) + 0.1 * np.random.randn(len(x))
```

In [ ]:
```
# Random indices for creating training and testing sets
random_ind = np.random.choice(list(range(120)), size = 120, replace=False)
```

```python
xt = x[random_ind]
yt = y[random_ind]

# Training and testing observations
train = xt[:int(0.7 * len(x))]
test = xt[int(0.7 * len(x)):]

y_train = yt[:int(0.7 * len(y))]
y_test = yt[int(0.7 * len(y)):]

# Model the true curve
x_linspace = np.linspace(0, 1, 1000)
y_true = true_gen(x_linspace)
```

In [ ]:

```python
# Visualize observations and true curve
plt.plot(train, y_train, 'ko', label = 'Train');
plt.plot(test, y_test, 'ro', label = 'Test')
plt.plot(x_linspace, y_true, 'b-', linewidth = 2, label = 'True Function')
plt.legend()
plt.xlabel('x'); plt.ylabel('y'); plt.title('Data');
```



Polynomial Regression Model Function

The `fit_poly` function fits a polynomial regression model to the training data and evaluates its performance on both the training and testing datasets. The function also provides options for plotting the results and returning key metrics.

Parameters

- `train` : Array-like, shape `(n_samples,)` : Training data features.
- `y_train` : Array-like, shape `(n_samples,)` : Training data target values.
- `test` : Array-like, shape `(n_samples,)` : Testing data features.
- `y_test` : Array-like, shape `(n_samples,)` : Testing data target values.
- `degrees` : int : The degree of the polynomial to fit.
- `plot` : str, optional, default='train' : If 'train', plots the model fitted on training data; if 'test', plots the model predictions on the test data.
- `return_scores` : bool, optional, default=False : If True, returns the training error, testing error, cross-validation score, and model coefficients.

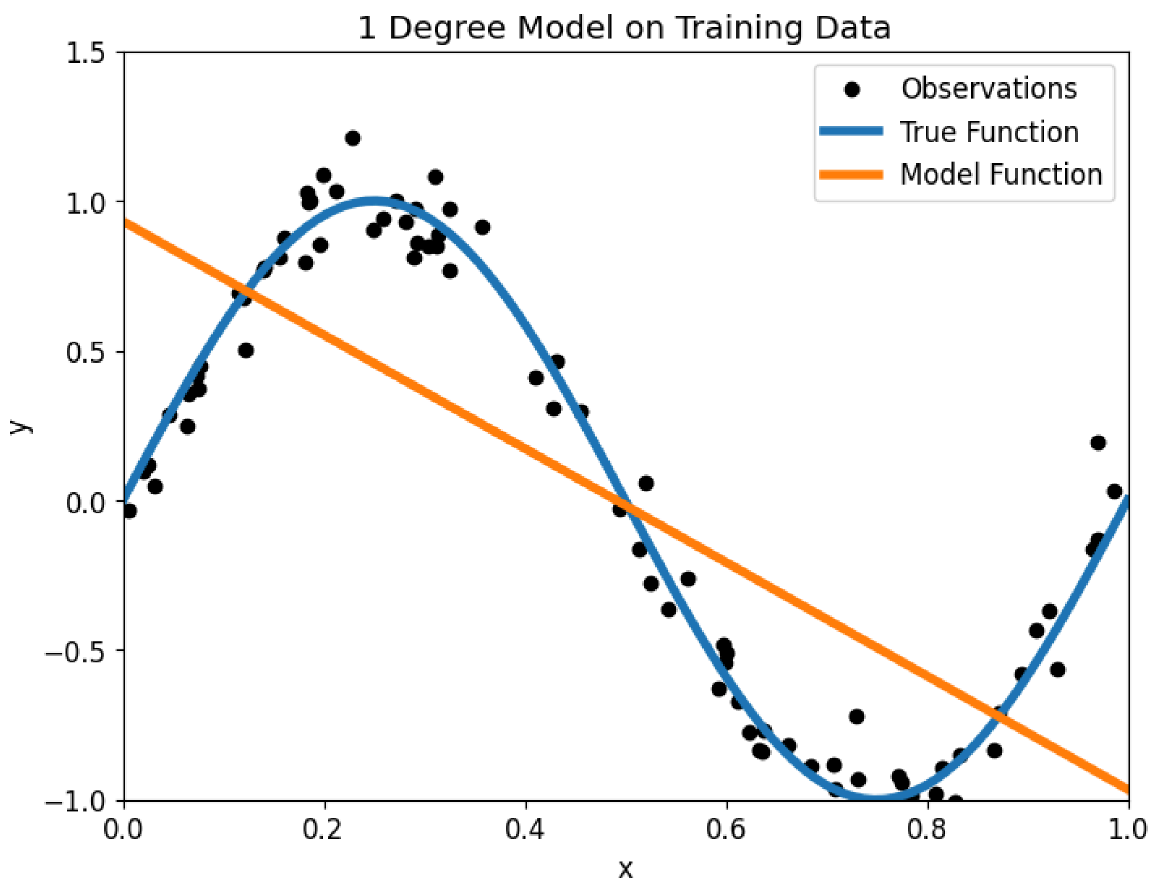
Returns

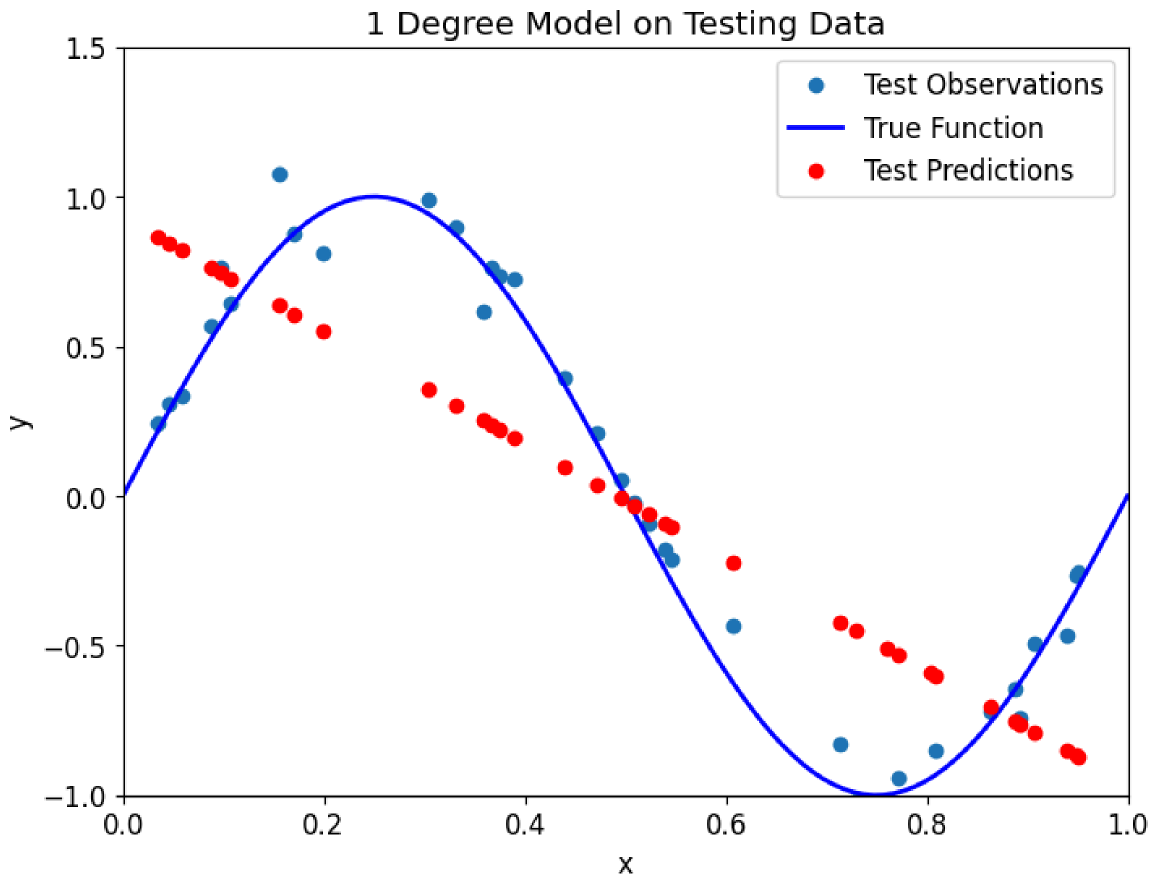
- `training_error` : float : Mean squared error on the training data.
- `testing_error` : float : Mean squared error on the testing data.
- `model.coef_` : array : Coefficients of the polynomial regression model.

In [ ]:
```python
def fit_poly(train, y_train, test, y_test, degrees, plot='train', return_scores=
    # Create a polynomial transformation of features
    features = PolynomialFeatures(degree=degrees, include_bias=False)

    # Reshape training features for use in scikit-learn and transform features
    train = train.reshape((-1, 1))
    train_trans = features.fit_transform(train)

    # Create the linear regression model and train
    model = LinearRegression()
    model.fit(train_trans, y_train)

    # Training predictions and error
    train_predictions = model.predict(train_trans)
    training_error = mean_squared_error(y_train, train_predictions)

    # Format test features
    test = test.reshape((-1, 1))
    test_trans = features.fit_transform(test)

    # Test set predictions and error
    test_predictions = model.predict(test_trans)
    testing_error = mean_squared_error(y_test, test_predictions)

    # Find the model curve and the true curve
    x_curve = np.linspace(0, 1, 100)
    x_curve = x_curve.reshape((-1, 1))
    x_curve_trans = features.fit_transform(x_curve)

    # Model curve
    model_curve = model.predict(x_curve_trans)
```

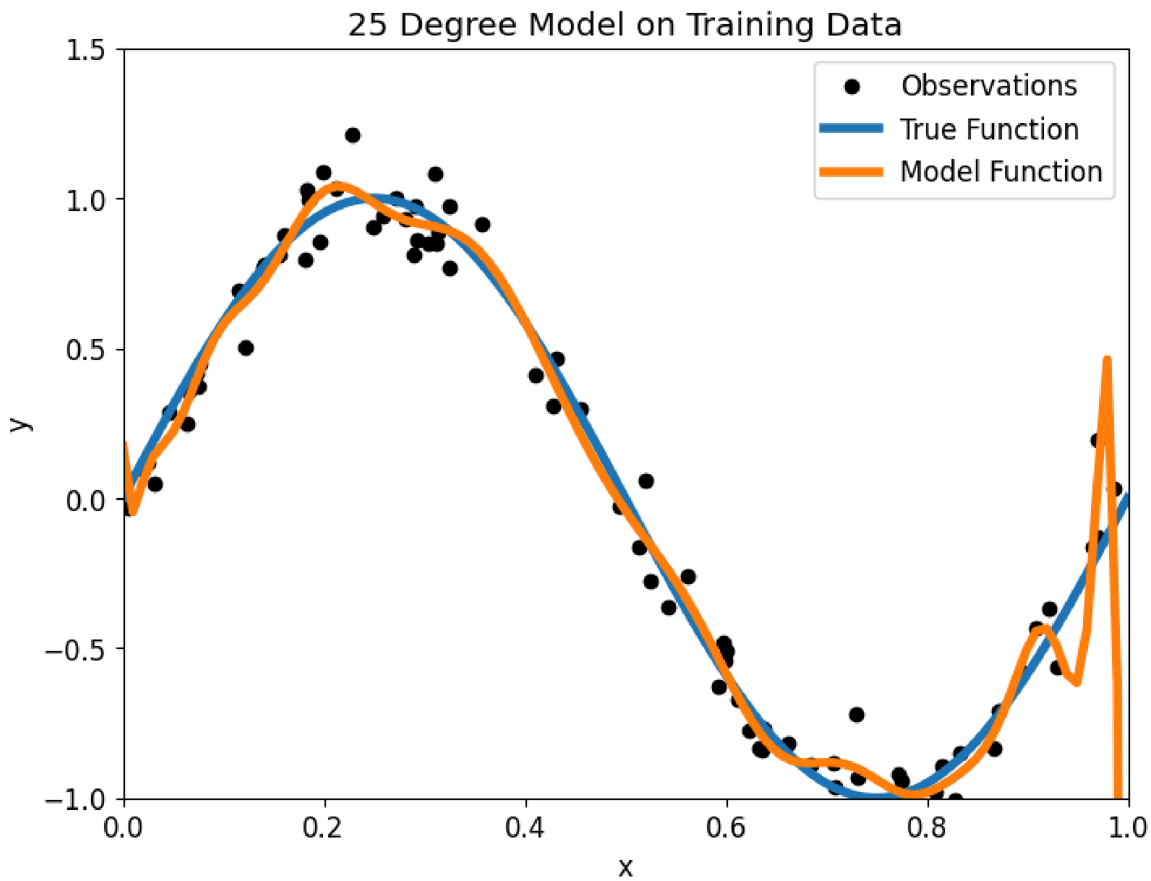
```
    # True curve
    y_true_curve = true_gen(x_curve[:, 0])

    # Plot observations, true function, and model predicted function
    if plot == 'train':
        plt.plot(train[:, 0], y_train, 'ko', label = 'Observations')
        plt.plot(x_curve[:, 0], y_true_curve, linewidth = 4, label = 'True Funct
        plt.plot(x_curve[:, 0], model_curve, linewidth = 4, label = 'Model Funct
        plt.xlabel('x'); plt.ylabel('y')
        plt.legend()
        plt.ylim(-1, 1.5); plt.xlim(0, 1)
        plt.title('{} Degree Model on Training Data'.format(degrees))
        plt.show()
    elif plot == 'test':
        # Plot the test observations and test predictions
        plt.plot(test, y_test, 'o', label = 'Test Observations')
        plt.plot(x_curve[:, 0], y_true_curve, 'b-', linewidth = 2, label = 'True
        plt.plot(test, test_predictions, 'ro', label = 'Test Predictions')
        plt.ylim(-1, 1.5); plt.xlim(0, 1)
        plt.legend(), plt.xlabel('x'), plt.ylabel('y'); plt.title('{} Degree Mod

    # Return the metrics and coefficients
    if return_scores:
        return training_error, testing_error, model.coef_
```

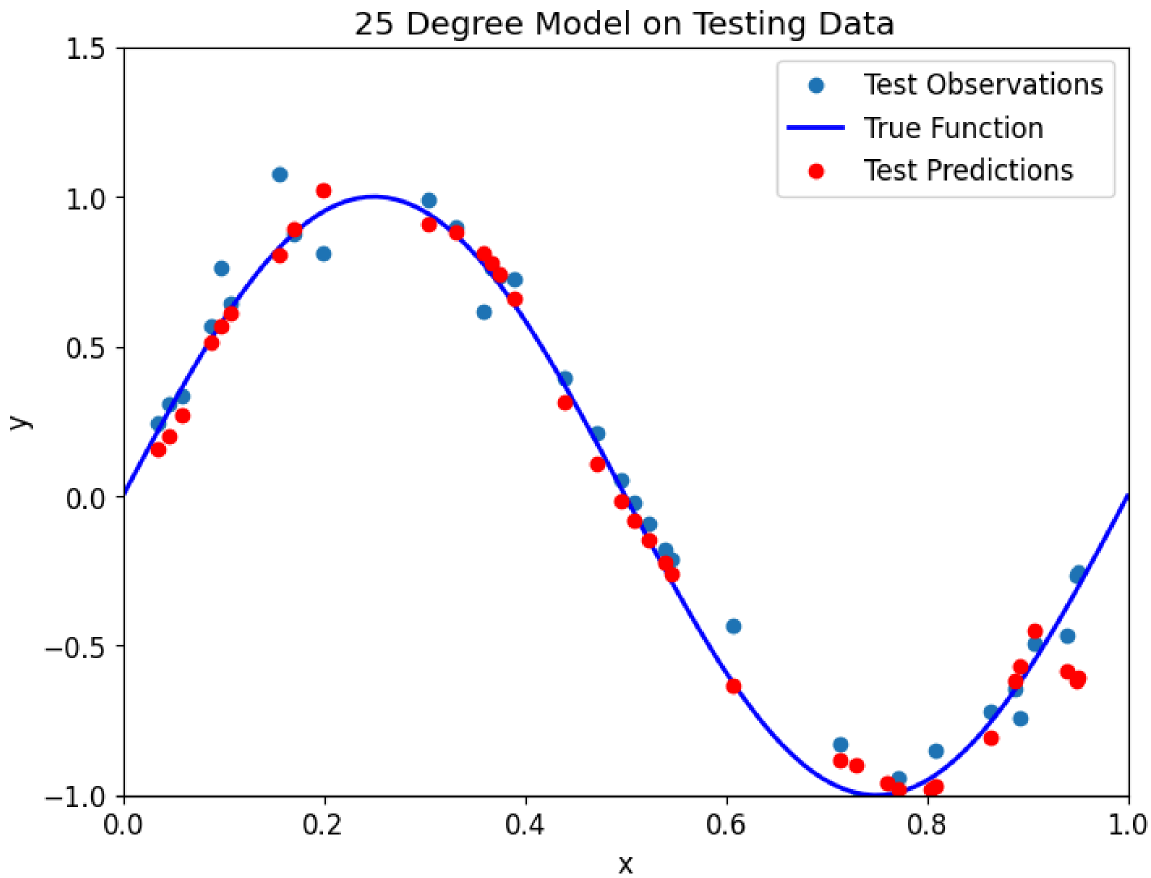
In [ ]:
```
fit_poly(train, y_train, test, y_test, degrees = 1, plot='train')

fit_poly(train, y_train, test, y_test, degrees = 1, plot='test')
```






```
In [ ]: fit_poly(train, y_train, test, y_test, plot='train', degrees = 25)
        fit_poly(train, y_train, test, y_test, degrees=25, plot='test')
```





In [ ]:
```python
import pandas as pd

# degrees = [1, 2, 3, 7, 10, 15, 20, 25, 30, 35]
degrees = [int(x) for x in np.linspace(1, 40, 40)]
results = pd.DataFrame(columns=['train_error', 'test_error'], index=degrees)
coefficients = []

# Try each value of degrees for the model and record results
for degree in degrees:
    degree_results = fit_poly(train, y_train, test, y_test, degree, plot=False,
    results.loc[degree, 'train_error'] = degree_results[0]
    results.loc[degree, 'test_error'] = degree_results[1]

    # Store coefficients in a dictionary with the degree
    coefs = degree_results[2]
    coef_dict = {'degree': degree}
    coef_dict.update({f'coef_{i}': coef for i, coef in enumerate(coefs)})
    coefficients.append(coef_dict)

# Convert the list of dictionaries to a DataFrame
coefficients_df = pd.DataFrame(coefficients)
coefficients_df.fillna(value=0, inplace=True)
coefficients_df.set_index('degree', inplace=True)
coefficients_df = coefficients_df.T
coefficients_df = coefficients_df.applymap(lambda x: f"{x:.2f}")
```

In [ ]:
```python
results
```

Out[ ]:

	train_error	test_error
1	0.228933	0.153
2	0.228641	0.151413
3	0.010658	0.009581
4	0.010634	0.009351
5	0.008646	0.009666
6	0.008407	0.010392
7	0.00839	0.010618
8	0.008389	0.010609
9	0.008304	0.010754
10	0.00817	0.011321
11	0.008104	0.011342
12	0.008095	0.011405
13	0.008084	0.011387
14	0.008077	0.011216
15	0.00755	0.011449
16	0.00722	0.012302
17	0.007169	0.012137
18	0.007118	0.012722
19	0.00685	0.015722
20	0.006787	0.017718
21	0.006773	0.018121
22	0.006703	0.019116
23	0.006703	0.019432
24	0.006689	0.018867
25	0.006675	0.018804
26	0.00663	0.018868
27	0.006429	0.022477
28	0.006422	0.02275
29	0.006465	0.022796
30	0.006488	0.02277
31	0.006386	0.018857
32	0.006394	0.01837
33	0.00643	0.018358

	train_error	test_error
34	0.006468	0.018529
35	0.006186	0.039825
36	0.006126	0.04663
37	0.006757	0.109419
38	0.006749	0.125783
39	0.006267	0.038482
40	0.006201	0.051216

In [ ]:

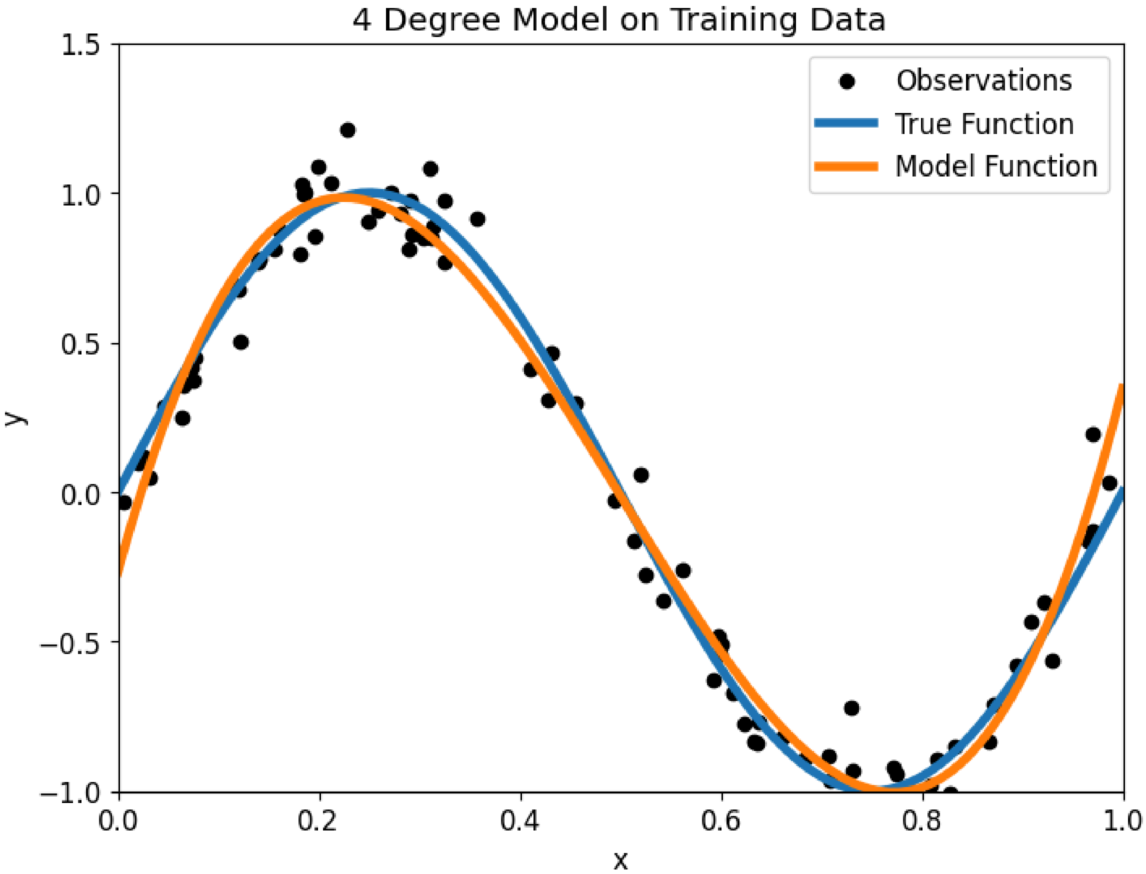
```
desired_degrees = [1, 2, 3, 5, 10, 15, 20, 25, 30, 35, 40]
coefficients_df[desired_degrees]
```

Out[ ]:

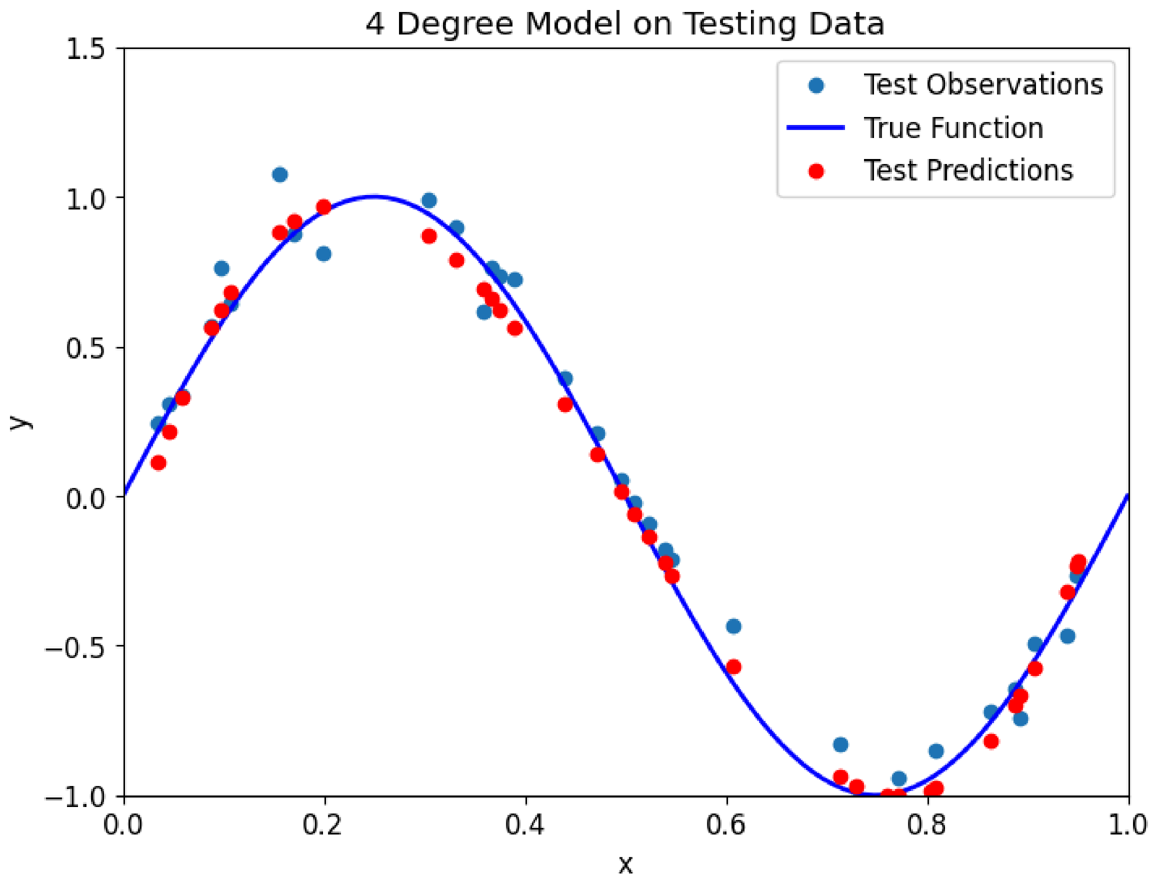
degree	1	2	3	5	10	15	20	
coef_0	-1.90	-1.66	12.54	7.76	7.38	-27.09	35.08	
coef_1	0.00	-0.24	-36.13	-2.96	-58.59	2053.19	-2134.43	
coef_2	0.00	0.00	24.20	-63.51	987.76	-55189.98	74868.69	
coef_3	0.00	0.00	0.00	97.94	-7887.12	805653.13	-1562392.36	
coef_4	0.00	0.00	0.00	-38.95	32664.70	-7216752.08	22051943.97	
coef_5	0.00	0.00	0.00	0.00	-79709.50	42671574.53	-227542401.60	
coef_6	0.00	0.00	0.00	0.00	119093.72	-174414973.65	1788921469.32	-
coef_7	0.00	0.00	0.00	0.00	-107004.53	506810202.44	-10900877690.64	
coef_8	0.00	0.00	0.00	0.00	53059.59	-1062236139.44	51737042471.97	-3
coef_9	0.00	0.00	0.00	0.00	-11153.32	1610332000.97	-191432481791.24	9
coef_10	0.00	0.00	0.00	0.00	0.00	-1748915955.78	552270090833.62	-23
coef_11	0.00	0.00	0.00	0.00	0.00	1326226674.53	-1241391069484.67	39
coef_12	0.00	0.00	0.00	0.00	0.00	-666609338.36	2168347659489.14	-44
coef_13	0.00	0.00	0.00	0.00	0.00	199504891.74	-2925021936127.82	22
coef_14	0.00	0.00	0.00	0.00	0.00	-26904674.77	3012095171077.38	16
coef_15	0.00	0.00	0.00	0.00	0.00	0.00	-2321188165065.61	-35
coef_16	0.00	0.00	0.00	0.00	0.00	0.00	1294749610800.10	11
coef_17	0.00	0.00	0.00	0.00	0.00	0.00	-493401797489.27	26
coef_18	0.00	0.00	0.00	0.00	0.00	0.00	114877788679.39	-29
coef_19	0.00	0.00	0.00	0.00	0.00	0.00	-12322977096.96	-5
coef_20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	35
coef_21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-35
coef_22	0.00	0.00	0.00	0.00	0.00	0.00	0.00	17
coef_23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-4
coef_24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
coef_25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
coef_26	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
coef_27	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
coef_28	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
coef_29	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
coef_30	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
coef_31	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
coef_32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	

degree	1	2	3	5	10	15	20
coef_33	0.00	0.00	0.00	0.00	0.00	0.00	0.00
coef_34	0.00	0.00	0.00	0.00	0.00	0.00	0.00
coef_35	0.00	0.00	0.00	0.00	0.00	0.00	0.00
coef_36	0.00	0.00	0.00	0.00	0.00	0.00	0.00
coef_37	0.00	0.00	0.00	0.00	0.00	0.00	0.00
coef_38	0.00	0.00	0.00	0.00	0.00	0.00	0.00
coef_39	0.00	0.00	0.00	0.00	0.00	0.00	0.00

In [ ]:
```
fit_poly(train, y_train, test, y_test, degrees=4, plot='train')
```



In [ ]:
```
fit_poly(train, y_train, test, y_test, degrees=4, plot='test')
```



In [ ]:
```
print('10 Lowest Training Errors\n')
train_eval = results.sort_values('train_error').reset_index(level=0).rename(colu
train_eval.loc[:,['degrees', 'train_error']] .head(10)
```

```
10 Lowest Training Errors
```

Out[ ]:

	degrees	train_error
0	36	0.006126
1	35	0.006186
2	40	0.006201
3	39	0.006267
4	31	0.006386
5	32	0.006394
6	28	0.006422
7	27	0.006429
8	33	0.00643
9	29	0.006465

In [ ]:
```
print('10 Lowest Testing Errors\n')
train_eval = results.sort_values('test_error').reset_index(level=0).rename(colum
train_eval.loc[:,['degrees', 'test_error']] .head(10)
```

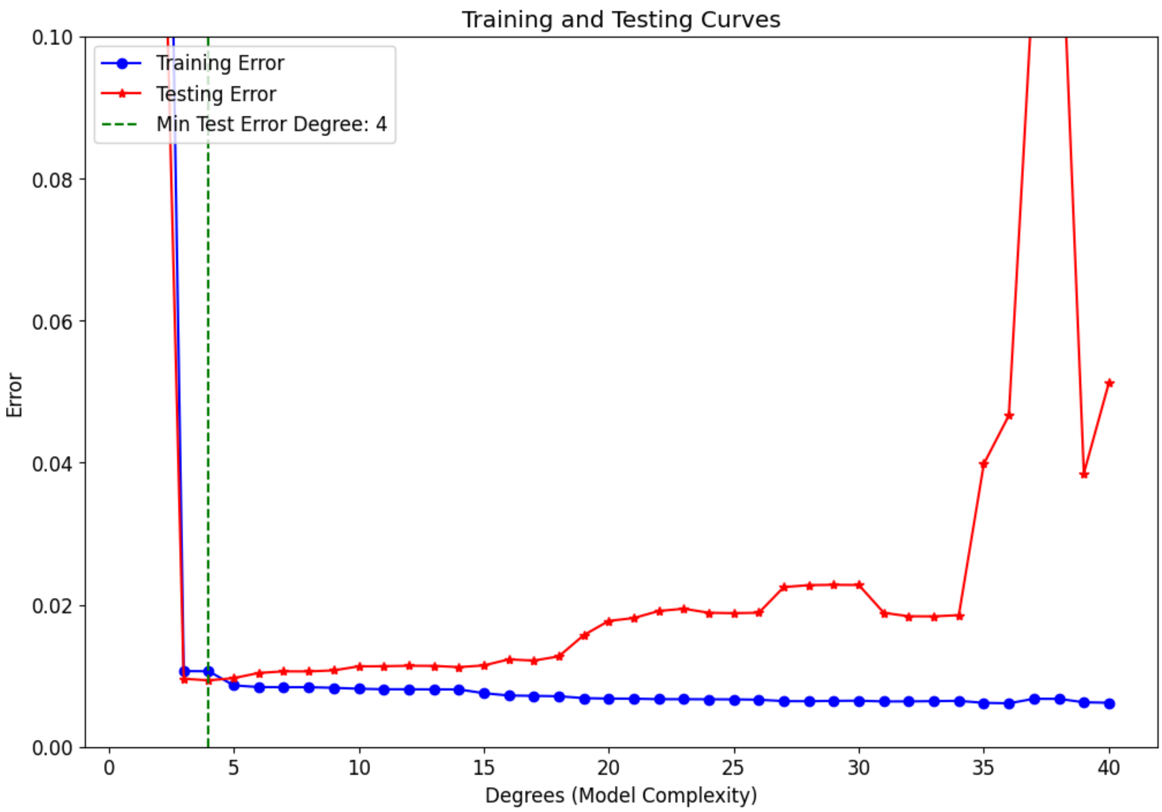
```
10 Lowest Testing Errors
```

Out[ ]:

	degrees	test_error
0	4	0.009351
1	3	0.009581
2	5	0.009666
3	6	0.010392
4	8	0.010609
5	7	0.010618
6	9	0.010754
7	14	0.011216
8	10	0.011321
9	11	0.011342

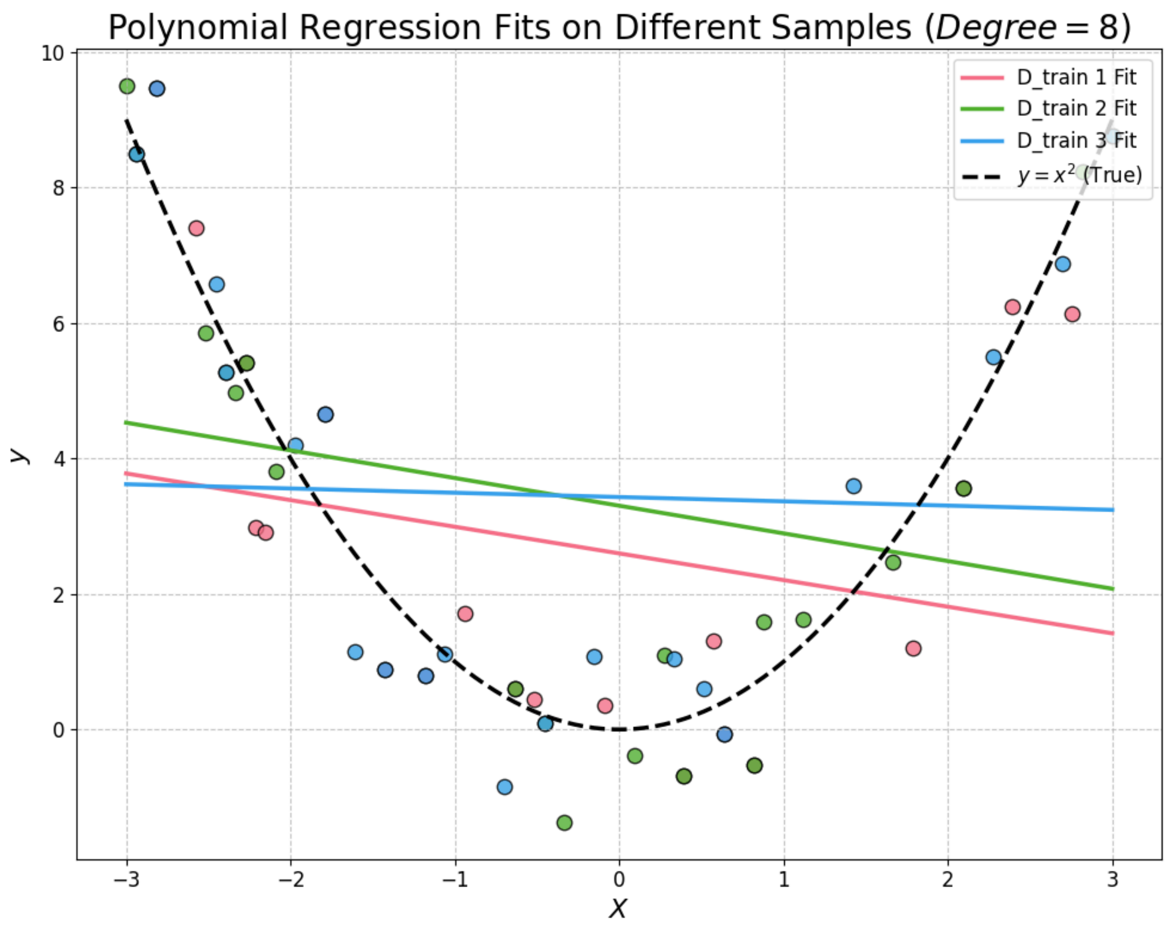
In [ ]:

```
plt.figure(figsize=(12, 8))
plt.plot(results.index, results['train_error'], 'b-o', ms=6, label = 'Training E
plt.plot(results.index, results['test_error'], 'r-*', ms=6, label = 'Testing Err
min_test_error_deg = results['test_error'].idxmin()
plt.axvline(min_test_error_deg, color='green', linestyle='--', label=f'Min Test 
plt.legend(loc=2); plt.xlabel('Degrees (Model Complexity)'); plt.ylabel('Error')
plt.ylim(0, 0.10); plt.show()

print('\nMinimum Training Error occurs at {} degrees.'.format(int(np.argmin(resu
print('Minimum Testing Error occurs at {} degrees.\n'.format(results['test_error
```




```
Minimum Training Error occurs at 35 degrees.
Minimum Testing Error occurs at 4 degrees.
```

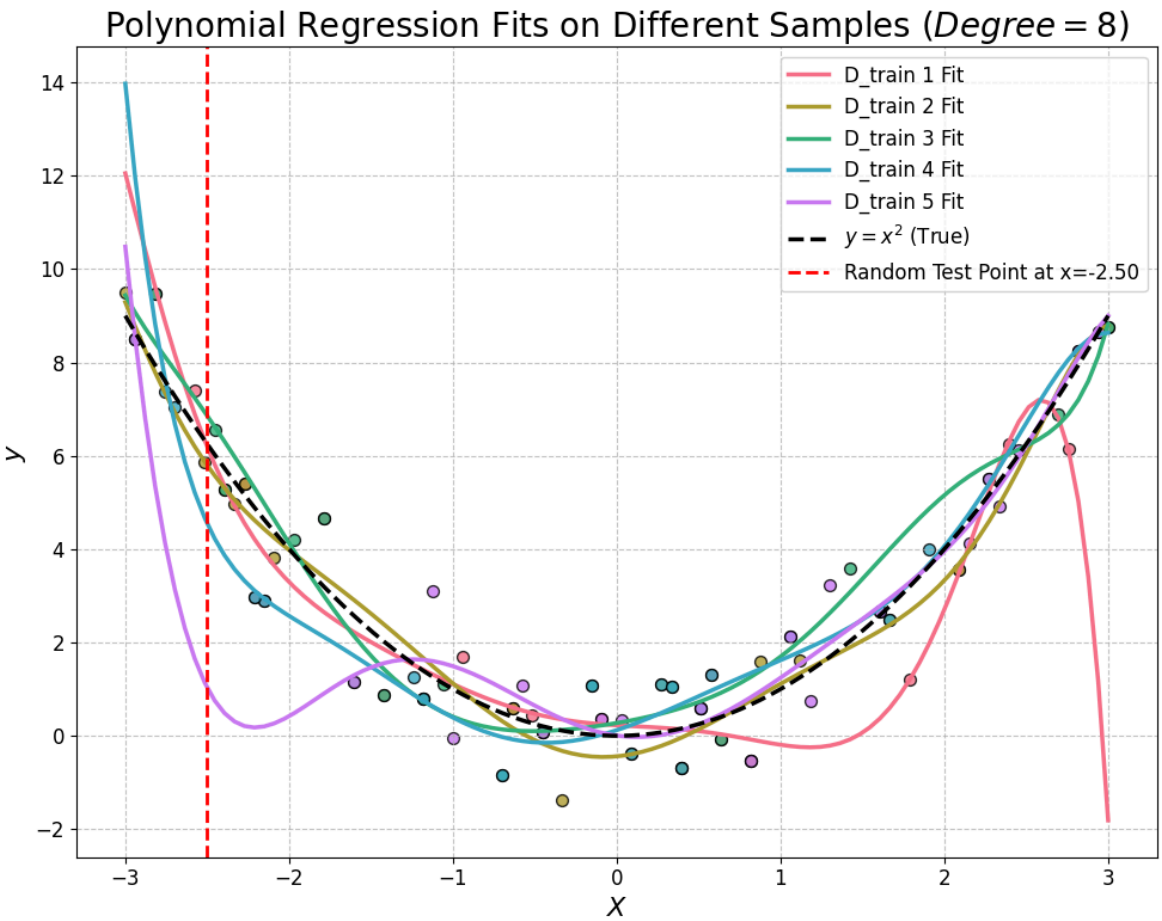
In [ ]:
```python
# Generate the dataset following y = x^2
np.random.seed(42)
X = np.linspace(-3, 3, 100).reshape(-1, 1)
y = X**2 + np.random.normal(scale=1.0, size=X.shape)  # Adding some noise

# Define the number of samples to draw from the dataset
num_samples = 3
sample_size = 20

# Colors for different samples
colors = sns.color_palette("husl", num_samples)
plt.figure(figsize=(10, 8))

# Train a polynomial regression model (degree 8) on different samples and plot t
for i in range(num_samples):
    # Randomly select a sample from the dataset
    indices = np.random.choice(range(len(X)), size=sample_size, replace=False)
    X_sample = X[indices]
    y_sample = y[indices]

    # Create a polynomial regression model of degree 8
    model = LinearRegression()
    model.fit(X_sample, y_sample)

    # Predict y values across the entire range for plotting the fitted curve
    y_pred = model.predict(X)

    # Plot the sample data points
    plt.scatter(X_sample, y_sample, color=colors[i], s=80, alpha=0.8, edgecolor=

    # Plot the fitted polynomial curve
    plt.plot(X, y_pred, color=colors[i], linestyle='-', linewidth=2.5, label=f'D

# Plot the original function for reference
plt.plot(X, X**2, color='black', linestyle='--', linewidth=2.5, label=r'$y = x^2

# Enhance the plot with labels, title, and legend
plt.xlabel(r'$X$', fontsize=16)
plt.ylabel(r'$y$', fontsize=16)
plt.title(r'Polynomial Regression Fits on Different Samples ($Degree = 8$)', fon
plt.legend(loc='upper right', fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()

# Display the plot
plt.show()
```



In [ ]:

```
# Generate the dataset following y = x^2
np.random.seed(42)
X = np.linspace(-3, 3, 100).reshape(-1, 1)
y = X**2 + np.random.normal(scale=1.0, size=X.shape)  # Adding some noise

# Define the number of samples to draw from the dataset
num_samples = 5
sample_size = 20

# Colors for different samples
colors = sns.color_palette("husl", num_samples)

plt.figure(figsize=(10, 8))

# Train a polynomial regression model (degree 8) on different samples and plot t
for i in range(num_samples):
    # Randomly select a sample from the dataset
    indices = np.random.choice(range(len(X)), size=sample_size, replace=False)
    X_sample = X[indices]
    y_sample = y[indices]

    # Create a polynomial regression model of degree 8
    model = make_pipeline(PolynomialFeatures(degree=8), LinearRegression())
    model.fit(X_sample, y_sample)

    # Predict y values across the entire range for plotting the fitted curve
    y_pred = model.predict(X)

    # Plot the sample data points
    plt.scatter(X_sample, y_sample, color=colors[i], s=50, alpha=0.8, edgecolor=
```

```
    # Plot the fitted polynomial curve
    plt.plot(X, y_pred, color=colors[i], linestyle='-', linewidth=2.5, label=f'D


# Plot the original function for reference
plt.plot(X, X**2, color='black', linestyle='--', linewidth=2.5, label=r'$y = x^2


# Plot a vertical line at the random test point
test_point = -2.5
plt.axvline(x=test_point, color='red', linestyle='--', linewidth=2, label=f'Rand

# Enhance the plot with labels, title, and legend
plt.xlabel(r'$X$', fontsize=16)
plt.ylabel(r'$y$', fontsize=16)
plt.title(r'Polynomial Regression Fits on Different Samples ($Degree = 8$)', fon
plt.legend(loc='upper right', fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()

# Display the plot
plt.show()
```



In [ ]:

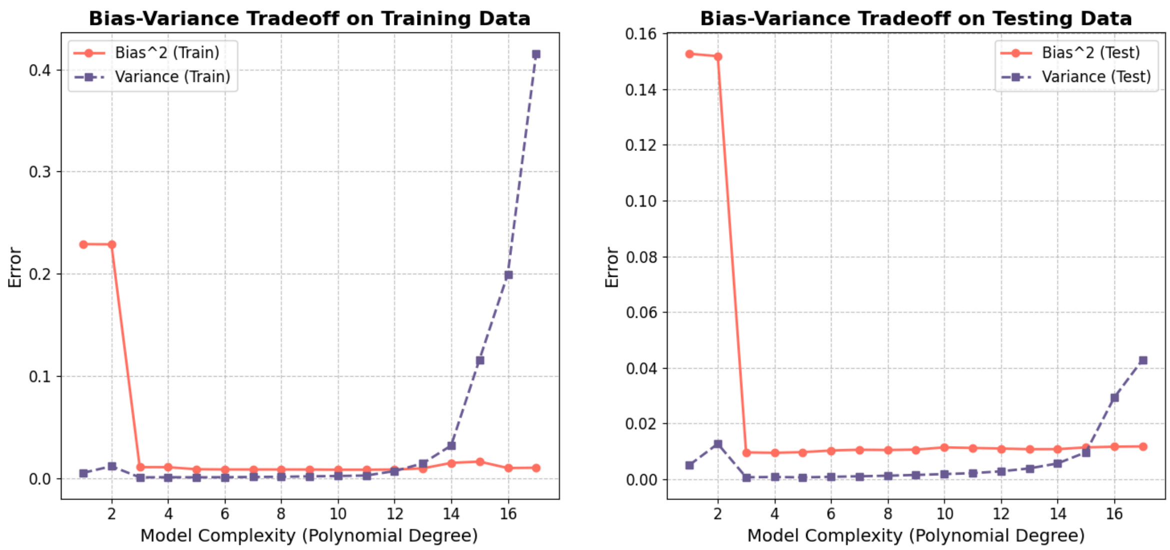
In [ ]:

```
def calculate_bias_variance(train, y_train, test, y_test, degrees, n_iterations=
    if random_seed is not None:
        np.random.seed(random_seed)  # Set the random seed

    features = PolynomialFeatures(degree=degrees, include_bias=False)
    train = train.reshape((-1, 1))
```

```python
    test = test.reshape((-1, 1))

    # Transform train and test data once, outside the loop
    train_transformed = features.fit_transform(train)
    test_transformed = features.fit_transform(test)

    all_predictions_train = []
    all_predictions_test = []

    for _ in range(n_iterations):
        # Resample the training data
        indices = np.random.choice(range(len(train)), len(train), replace=True)
        train_resample = train[indices]
        y_train_resample = y_train[indices]

        # Transform the resampled train data  X--- phi(X)
        train_resample_trans = features.fit_transform(train_resample)

        # Fit the model on the resampled training data
        model = LinearRegression().fit(train_resample_trans, y_train_resample)

        # Predict on the original train and test sets
        all_predictions_train.append(model.predict(train_transformed))
        all_predictions_test.append(model.predict(test_transformed))

    # Convert lists to arrays
    all_predictions_train = np.array(all_predictions_train)
    all_predictions_test = np.array(all_predictions_test)

    # Bias Calculation
    bias_train = np.mean((np.mean(all_predictions_train, axis=0) - y_train) ** 2
    bias_test = np.mean((np.mean(all_predictions_test, axis=0) - y_test) ** 2)

    # Variance Calculation
    variance_train = np.mean(np.var(all_predictions_train, axis=0))
    variance_test = np.mean(np.var(all_predictions_test, axis=0))

    # MSE Calculation
    mse_train = bias_train + variance_train
    mse_test = bias_test + variance_test

    return {
        'bias_train': bias_train,
        'variance_train': variance_train,
        'mse_train': mse_train,
        'bias_test': bias_test,
        'variance_test': variance_test,
        'mse_test': mse_test
    }
```

In [ ]:
```python
def plot_bias_variance(train, y_train, test, y_test, max_degree=15, n_iterations
    biases_train, variances_train, biases_test, variances_test = [], [], [], []
    degrees = range(1, max_degree + 1)

    for degree in degrees:
        results = calculate_bias_variance(train, y_train, test, y_test, degree,
        biases_train.append(results['bias_train'])
        variances_train.append(results['variance_train'])
        biases_test.append(results['bias_test'])
        variances_test.append(results['variance_test'])
```

```python
    plt.figure(figsize=(14, 7))

    # Custom color palette
    color_bias = '#FF6F61'
    color_variance = '#6B5B95'

    # Plot for Training Data
    plt.subplot(1, 2, 1)
    plt.plot(degrees, biases_train, label='Bias^2 (Train)', color=color_bias, ma
    plt.plot(degrees, variances_train, label='Variance (Train)', color=color_var
    plt.xlabel('Model Complexity (Polynomial Degree)', fontsize=14)
    plt.ylabel('Error', fontsize=14)
    plt.title('Bias-Variance Tradeoff on Training Data', fontsize=16, weight='bc
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.legend(fontsize=12)

    # Plot for Testing Data
    plt.subplot(1, 2, 2)
    plt.plot(degrees, biases_test, label='Bias^2 (Test)', color=color_bias, mark
    plt.plot(degrees, variances_test, label='Variance (Test)', color=color_varia
    plt.xlabel('Model Complexity (Polynomial Degree)', fontsize=14)
    plt.ylabel('Error', fontsize=14)
    plt.title('Bias-Variance Tradeoff on Testing Data', fontsize=16, weight='bol
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.legend(fontsize=12)

    # Adjust layout for better spacing
    plt.tight_layout(pad=3.0)
    plt.show()

# Example usage:
plot_bias_variance(train, y_train, test, y_test, max_degree=17, n_iterations=100
```



In [ ]: