

ESE 417 Homework 2

Problem 1 (OLS, linear regression)

Given a data set $\{(x_i, y_i)\}_{i=1}^N$, where $x_i, y_i \in \mathbb{R}$, we want to find a "least squares line" $\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$ to fit the data set with minimum squared residuals $\sum_{i=1}^N (y_i - \hat{y}_i)^2$.

a. Show that:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n x_i y_i - \frac{1}{n} (\sum_{i=1}^n x_i) (\sum_{i=1}^n y_i)}{\sum_{i=1}^n x_i^2 - \frac{1}{n} (\sum_{i=1}^n x_i)^2} = \frac{\sum_{i=1}^n x_i y_i - n \bar{x} \bar{y}}{\sum_{i=1}^n x_i^2 - n \bar{x}^2}$$
$$\hat{\beta}_0 = \frac{1}{n} \sum_{i=1}^n y_i - \hat{\beta}_1 \frac{1}{n} \sum_{i=1}^n x_i = \bar{y} - \hat{\beta}_1 \bar{x}$$

Where,

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i; \quad \bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$$

This question is written at the beginning.

b. Use Python to generate and visualize a synthetic data set $\{x_i, y_i\}, i = 1, 2, \dots, n$, that satisfy:

$$y_i = -5x_i + 15 + \epsilon_i, \quad i = 1, 2, \dots, n$$

Where, x_i 's are values evenly spaced between 1 and 10, i.e., $1 \leq x_i \leq 10$; ϵ_i is a noise that follows a normal distribution with zero mean and standard deviation of 1.

```
In [57]: import numpy as np
import matplotlib.pyplot as plt

# Set the random seed
np.random.seed(0)

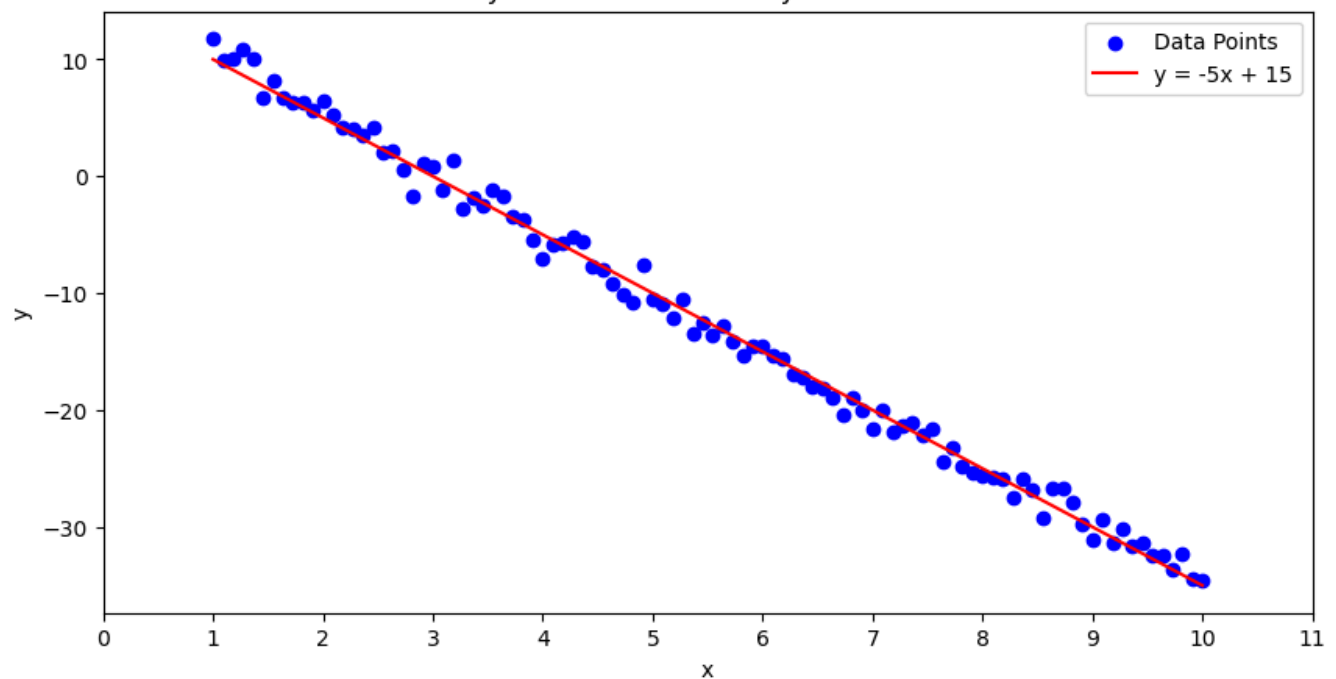
# Generate evenly x values between 1 and 10
n = 100
x = np.linspace(1, 10, n)

# Generate noise from a normal distribution with mean 0 and standard deviation 1
epsilon = np.random.normal(0, 1, n)

# Calculate y values using the given equation y = -5x + 15 + epsilon
y = -5 * x + 15 + epsilon

# Plot the synthetic dataset and the line y = -5x + 15
plt.figure(figsize=(10, 5))
plt.scatter(x, y, label='Data Points', color='blue')
plt.plot(x, -5*x + 15, label='y = -5x + 15', color='red')
plt.title('Synthetic Data Set and y = -5x + 15')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.xticks(np.arange(0, 12, 1))
plt.show()
```

Synthetic Data Set and $y = -5x + 15$



c. Use Python (do not use sklearn package) to find the "least squares line" $\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$ for the data set generated in 1.b and create a figure showing the data points in the data set and the calculated "least squares line". Calculate the sum of the squares of the residuals associated with the "least squares line".

```
In [58]: # Compute the least squares coefficients manually

# Calculate means of x and y
mean_x = np.mean(x)
mean_y = np.mean(y)

# # Calculate the slope (beta_1)
beta_1 = (np.sum(x*y) - n*np.mean(x)*np.mean(y)) / (np.sum(x**2) - n*np.mean(x)**2)

# Calculate the intercept (beta_0)
beta_0 = mean_y - beta_1 * mean_x

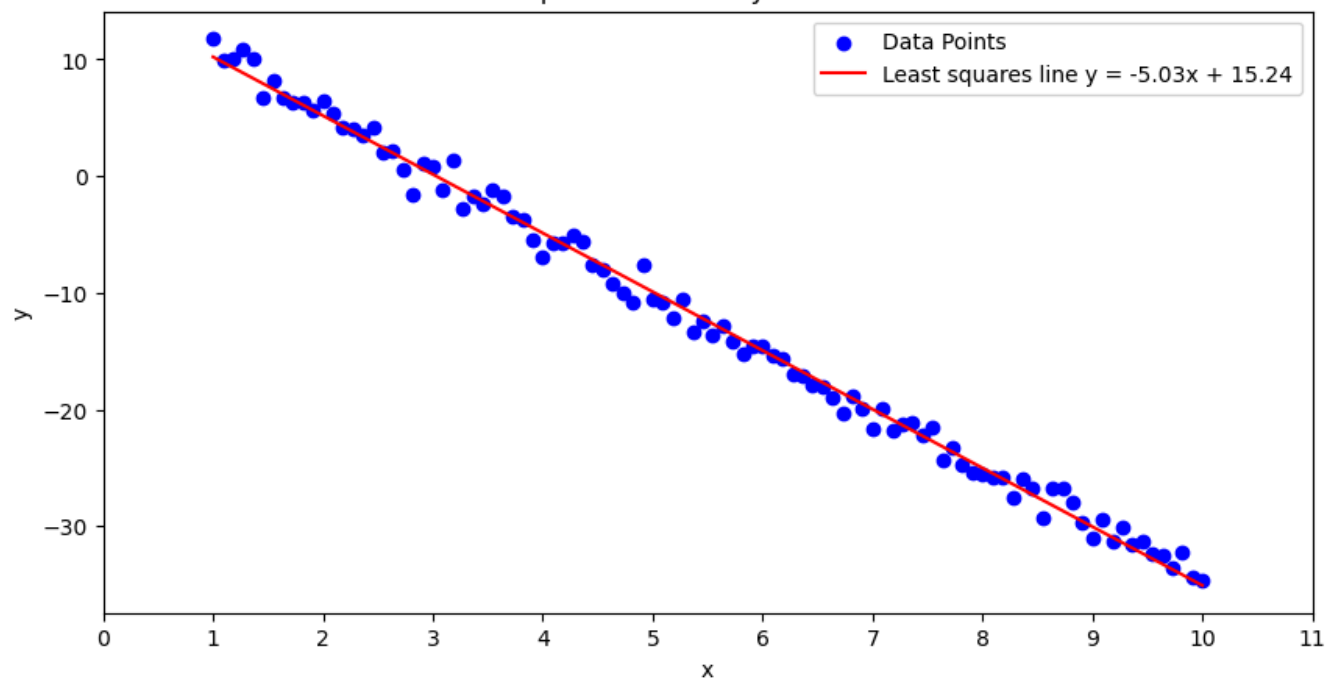
# Calculate the least squares line y_hat = beta_0 + beta_1 * x
y_hat = beta_0 + beta_1 * x

# Calculate the residuals (y - y_hat) and the sum of squared residuals
residuals = y - y_hat
sum_squared_residuals = np.sum(residuals**2)

# Plot the data points and the least squares line
plt.figure(figsize=(10, 5))
plt.scatter(x, y, label='Data Points', color='blue')
plt.plot(x, y_hat, label=f'Least squares line y = {beta_1:.2f}x + {beta_0:.2f}', color='red')
plt.title('Least Squares Line for Synthetic Data Set')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.xticks(np.arange(0, 12, 1))
plt.show()

# Print the sum of the squared of the residuals
print(f'Sum of squared residuals: {sum_squared_residuals:.2f}')
```

Least Squares Line for Synthetic Data Set



Sum of squared residuals: 100.83

d. Use sklearn package to verify your result in 1.c.

```
In [59]: from sklearn.linear_model import LinearRegression

# Reshape x from a 1D array to a 2D array of shape (n, 1)
x_resaped = x.reshape(-1, 1)

# Create a linear regression model
model = LinearRegression()

# Fit the linear regression model to the data
model.fit(x_resaped, y)

# Get the coefficients
beta_0_sklearn = model.intercept_
beta_1_sklearn = model.coef_[0]

# Predict y values using the model
y_hat_sklearn = model.predict(x_resaped)

# Calculate the sum of squared residuals
residuals_sklearn = y - y_hat_sklearn
sum_squared_residuals_sklearn = np.sum(residuals_sklearn**2)

# Print the line and the sum of squared residuals
print(f'Least squares line (sklearn): y = {beta_1_sklearn:.2f}x + {beta_0_sklearn:.2f}')
print(f'Sum of squared residuals (sklearn): {sum_squared_residuals_sklearn:.2f}')

# Verify that the coefficients and the sum of squared residuals are the same
print('The coefficients are the same:',
      np.allclose([beta_0, beta_1], [beta_0_sklearn, beta_1_sklearn]))
print('The sum of squared residuals is the same:',
      np.isclose(sum_squared_residuals, sum_squared_residuals_sklearn))

# Plot the data points and the least squares line
plt.figure(figsize=(10, 5))
plt.scatter(x, y, label='Data Points', color='blue')
plt.plot(x, y_hat_sklearn, label=f'Least squares line y = {beta_1_sklearn:.2f}x + {beta_0_sklearn:.2f}')
plt.title('Least Squares Line for Synthetic Data Set (sklearn)')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
```

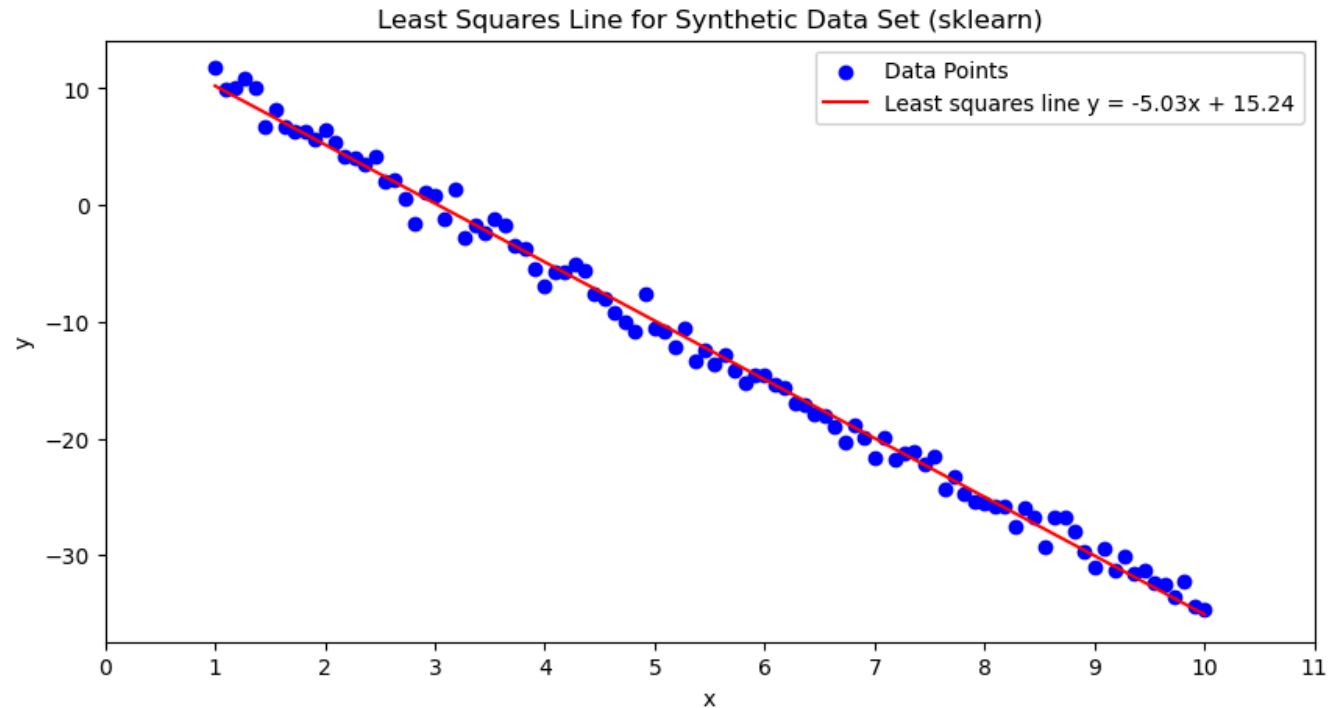
```
plt.xticks(np.arange(0, 12, 1))
plt.show()
```

Least squares line (sklearn): $y = -5.03x + 15.24$

Sum of squared residuals (sklearn): 100.83

The coefficients are the same: True

The sum of squared residuals is the same: True



Problem 2 (Polynomial Regression)

a. Generate a synthetic data set with 100 data points (x_i, y_i) , where the input $-5 \leq x_i \leq 5$ are evenly spaced and the output $y_i = 12 \sin(x_i) + 0.5x_i^2 + 2x_i + 5$ is polluted by Gaussian noise with zero mean and standard deviation of 2. Create a figure showing the scattering of the generated noisy data points and the true pattern of $y = f(x)$.

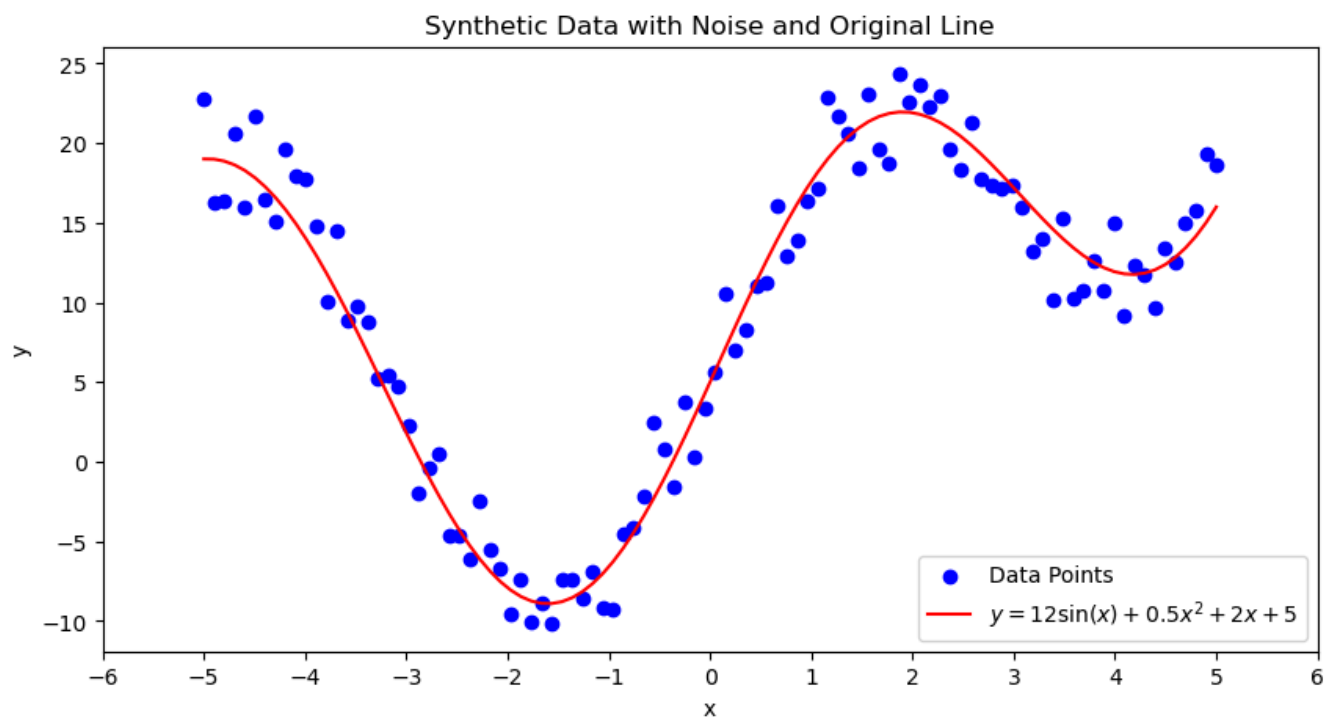
```
In [60]: # Generate a synthetic data set with 100 data points
n = 100
x = np.linspace(-5, 5, n)

# Generate noise from a normal distribution with mean 0 and standard deviation 2
noise = np.random.normal(0, 2, n)

# Calculate y values using the given equation: y = 12*sin(x) + 0.5*x^2 + 2*x + 5 + noise
y_noise = 12 * np.sin(x) + 0.5 * x**2 + 2 * x + 5 + noise

# True value without noise: y = 12 * sin(x) + 0.5 * x^2 + 2 * x + 5
y_true = 12 * np.sin(x) + 0.5 * x**2 + 2 * x + 5

# Plot the synthetic noisy data points and the original line
plt.figure(figsize=(10, 5))
plt.scatter(x, y_noise, label='Data Points', color='blue')
plt.plot(x, y_true, label='$y = 12 \sin(x) + 0.5x^2 + 2x + 5$', color='red')
plt.title('Synthetic Data with Noise and Original Line')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.xticks(np.arange(-6, 7, 1))
plt.show()
```



b. Create a sklearn pipeline model for polynomial regression by chaining PolynomialFeatures feature mapping with degree of 5 and a LinearRegression model.

```
In [61]: from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Create a pipeline for polynomial regression with degree 5
# containing PolynomialFeatures and LinearRegression
polynomial_regression = Pipeline([
    ('polynomial_features', PolynomialFeatures(degree=5)),
    ('linear_regression', LinearRegression())
])
```

C. Split the data set into training set and test set with test size = 20%. Train the polynomial regression model on the training set and make prediction on the test set. Calculate the mean squared test error.

```
In [62]: from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Split the data into training and test sets with test size 20%
x_train, x_test, y_train, y_test = train_test_split(x.reshape(-1, 1), y_noise,
                                                    test_size=0.2, random_state=0)

# Train the polynomial regression model on the training set
polynomial_regression.fit(x_train, y_train)

# Make predictions on the test set
y_pred = polynomial_regression.predict(x_test)

# Calculate the mean squared test error
mse_test_error = mean_squared_error(y_test, y_pred)

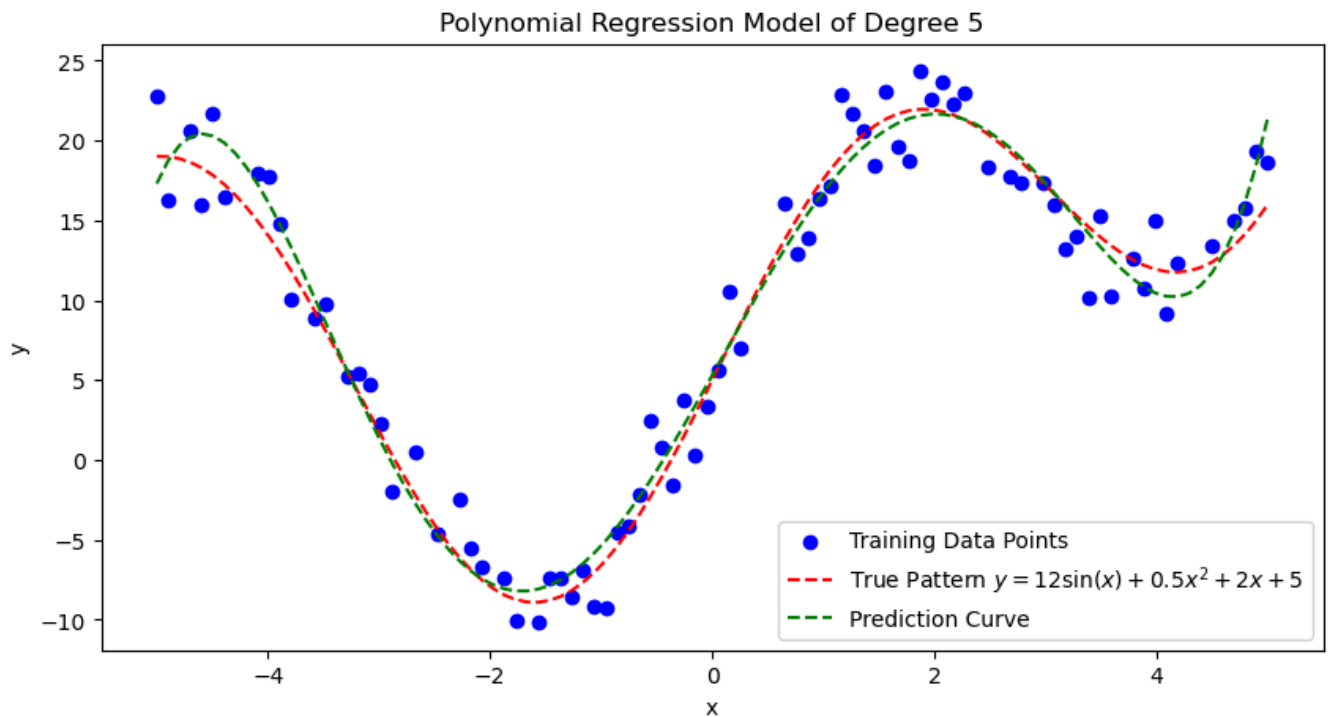
# Print the mean squared test error
print(f'Mean squared test error: {mse_test_error:.2f}')
```

Mean squared test error: 2.89

d. Create a figure to show the prediction curve of the trained model on top of the training data points and the true pattern as in a.

```
In [63]: # Generate predictions using the trained model on the entire range of x values
y_pred_full = polynomial_regression.predict(x.reshape(-1, 1))

# Plot the training data, true pattern, and prediction curve
plt.figure(figsize=(10, 5))
plt.scatter(x_train, y_train, label='Training Data Points', color='blue')
plt.plot(x, y_true, label='True Pattern  $y = 12 \sin(x) + 0.5x^2 + 2x + 5$ ', color='red', lines
plt.plot(x, y_pred_full, label='Prediction Curve', color='green', linestyle='--')
plt.title('Polynomial Regression Model of Degree 5')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



Problem 3 (Polynomial Regression, hyperparameter tuning using GridSearchCV)

a. Create a sklearn pipeline model for polynomial regression by using PolynomialFeatures feature mapping and a LinearRegression model.

```
In [64]: from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Create a pipeline for polynomial regression
polynomial_regression = Pipeline([
    ('polynomial_features', PolynomialFeatures()),
    ('linear_regression', LinearRegression())
])
```

b. Split the data set in Q2.a into training set and test set with test size = 20%. To find the best polynomial model on the given dataset, cross validation method is used to tune the degree of the polynomial feature transformation. Please use GridSearchCV method in sklearn to find the best degree and mean squared test error of the best model.

```
In [65]: import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
```

```

from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error

# Split the data set in Q2.a into training and test sets with test size 20%
x_train, x_test, y_train, y_test = train_test_split(x.reshape(-1, 1), y_noise,
                                                    test_size=0.2, random_state=0)

# Define the parameter for GridSearchCV to tune the degree of the polynomial
# Search from degree 1 to 10
parameter_grid = {
    'polynomial_features__degree': np.arange(1, 11)
}

# Use GridSearchCV to find the best degree of the polynomial with 5-fold cross-validation
grid_search = GridSearchCV(polynomial_regression,
                            parameter_grid,
                            cv=5,
                            scoring='neg_mean_squared_error')

# Train the polynomial regression model on the training set
grid_search.fit(x_train, y_train)

# Get the best model on the training set via GridSearchCV
best_model = grid_search.best_estimator_
best_degree = best_model.named_steps['polynomial_features'].degree
best_mse = -grid_search.best_score_

# Test the best model on the test set
y_pred = best_model.predict(x_test)
test_mes = mean_squared_error(y_test, y_pred)

# Print the best degree and the mean squared error on the test set
print(f'Best degree: {best_degree}')
print(f'Mean squared error of best model on the training set: {best_mse:.2f}')
print(f'Mean squared error of best model on the test set: {test_mes:.2f}')

```

Best degree: 7
Mean squared error of best model on the training set: 4.80
Mean squared error of best model on the test set: 3.35

c. Create a figure to show the prediction curve of the trained model on top of the training data points and the true pattern.

```

In [66]: import matplotlib.pyplot as plt
import numpy as np

# Assuming x_train, y_train, and the true function (y_true) are already defined
# You can replace y_true with the actual function for the true pattern, e.g., y_true = f(x)

# Generate a smooth curve for the true pattern
x_plot = np.linspace(x_train.min(), x_train.max(), 1000).reshape(-1, 1)
y_true_plot = 12 * np.sin(x_plot) + 0.5 * x_plot**2 + 2 * x_plot + 5

# Predict the curve using the best model from GridSearchCV
y_pred_plot = best_model.predict(x_plot)

# Plot the results
plt.figure(figsize=(10, 5))

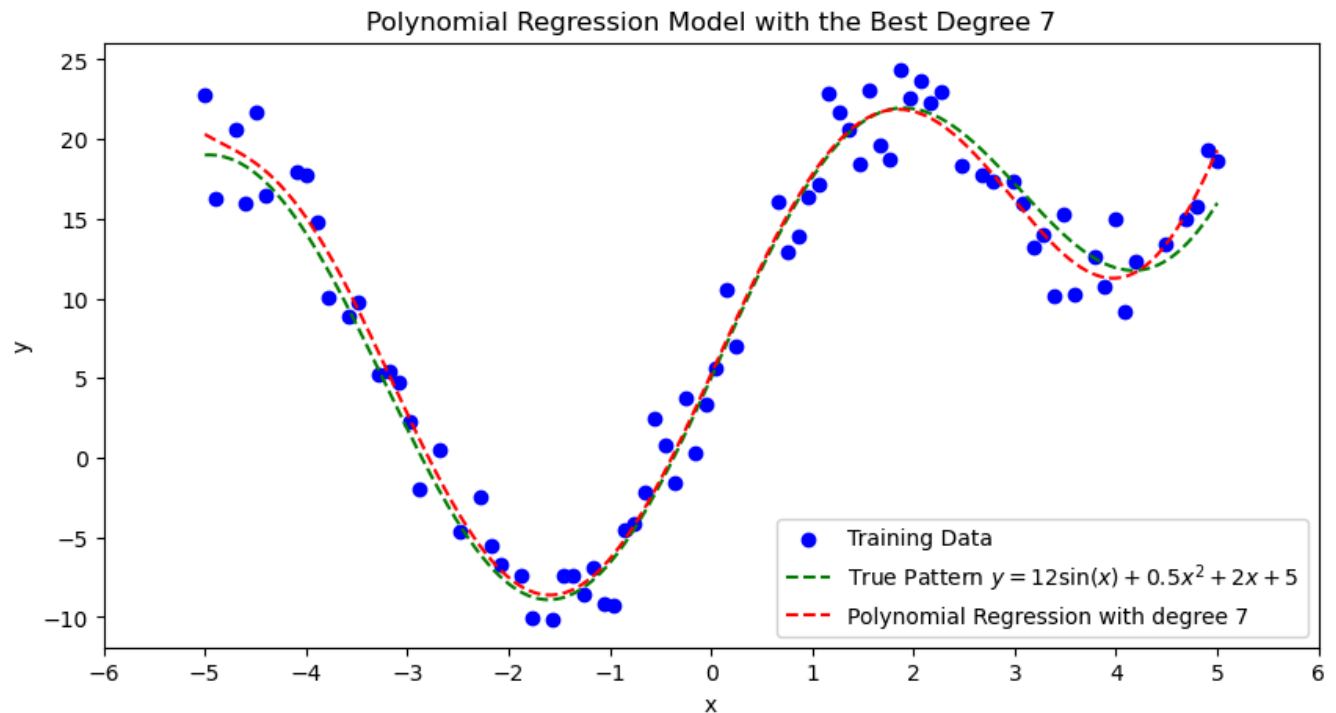
# Plot the training data points
plt.scatter(x_train, y_train, color='blue', label='Training Data')

# Plot the true pattern (without noise)
plt.plot(x_plot, y_true_plot, color='green', linestyle='--', label='True Pattern $y = 12 \sin(x) + 0.5x^2 + 2x + 5$')

# Plot the prediction curve from the best polynomial model
plt.plot(x_plot, y_pred_plot, color='red', label=f'Polynomial Regression with degree {best_degree}')
plt.xlabel('x')

```

```
plt.ylabel('y')
plt.title(f'Polynomial Regression Model with the Best Degree {best_degree}')
plt.legend()
plt.xticks(np.arange(-6, 7, 1))
plt.show()
```



Problem 4 (Underfitting/Overfitting, Polynomial Regression)

Use Python to carry out the following:

a. Generate a synthetic data set with 100 data points (x_i, y_i) , where the input $-5 \leq x_i \leq 5$ are evenly spaced and the output $y_i = 12 \sin(x_i) + 0.5x_i^2 + 2x_i + 5$ is polluted by Gaussian noise with zero mean and standard deviation of 2. Create a figure showing the scattering of the generated noisy data points and the true pattern of $y = f(x)$.

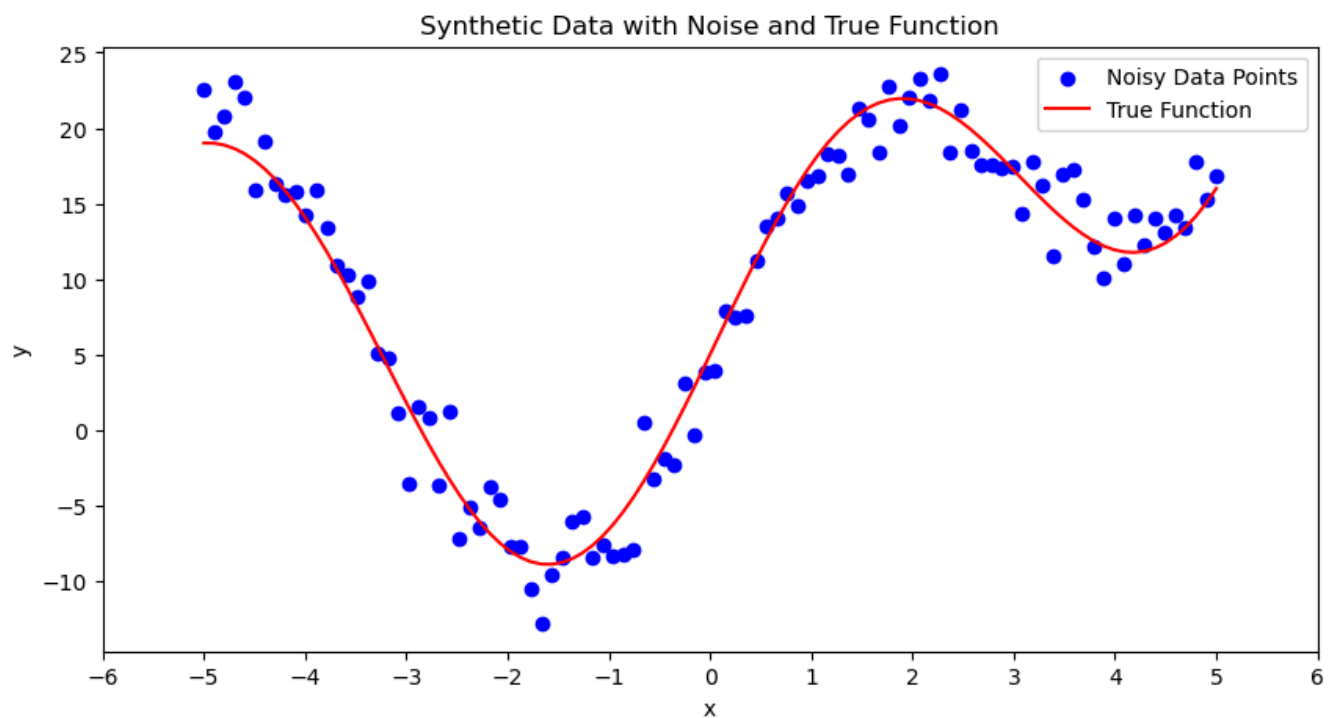
```
In [67]: import numpy as np
import matplotlib.pyplot as plt

# Generate 100 evenly spaced xi in [-5, 5]
x = np.linspace(-5, 5, 100)

# True function without noise
y_true = 12 * np.sin(x) + 0.5 * x**2 + 2 * x + 5

# Add Gaussian noise with mean 0 and standard deviation 2
np.random.seed(0) # For reproducibility
noise = np.random.normal(0, 2, size=x.shape)
y_noisy = y_true + noise

# Plot the data
plt.figure(figsize=(10, 5))
plt.scatter(x, y_noisy, label='Noisy Data Points', color='blue')
plt.plot(x, y_true, color='red', label='True Function')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Synthetic Data with Noise and True Function')
plt.legend()
plt.xticks(np.arange(-6, 7, 1))
plt.show()
```



b. Create a sklearn pipeline model for polynomial regression by using PolynomialFeatures feature mapping and LinearRegression model.

```
In [68]: from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Create a pipeline model for polynomial regression
def PolynomialRegression(degree):
    polynomial_regression = Pipeline([
        ('polynomial_features', PolynomialFeatures(degree)),
        ('linear_regression', LinearRegression())
    ])
    return polynomial_regression
```

```
In [69]: # from sklearn.pipeline import make_pipeline
# from sklearn.preprocessing import PolynomialFeatures
# from sklearn.linear_model import LinearRegression

# # Create a pipeline model for polynomial regression
# def PolynomialRegression(degree):
#     return make_pipeline(PolynomialFeatures(degree), LinearRegression())
```

c. Split the data set into training set and test set with test size = 20%.

```
In [70]: from sklearn.model_selection import train_test_split

# Reshape xi to be a 2D array for sklearn
X = x.reshape(-1, 1)
y = y_noisy

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

d. Train the polynomial regression model on the training set and then use the trained model to make prediction on the test set. Show prediction curves with different polynomial degrees on the test data scatter plot.

```
In [71]: degrees = np.arange(1, 21)

plt.figure(figsize=(12, 8))
```

```

plt.scatter(X_test, y_test, color='black', label='Test Data')
# train data
plt.scatter(X_train, y_train, color='gray', label='Train Data')

# Generate a fine grid for plotting
X_plot = np.linspace(X.min(), X.max(), 500).reshape(-1, 1)

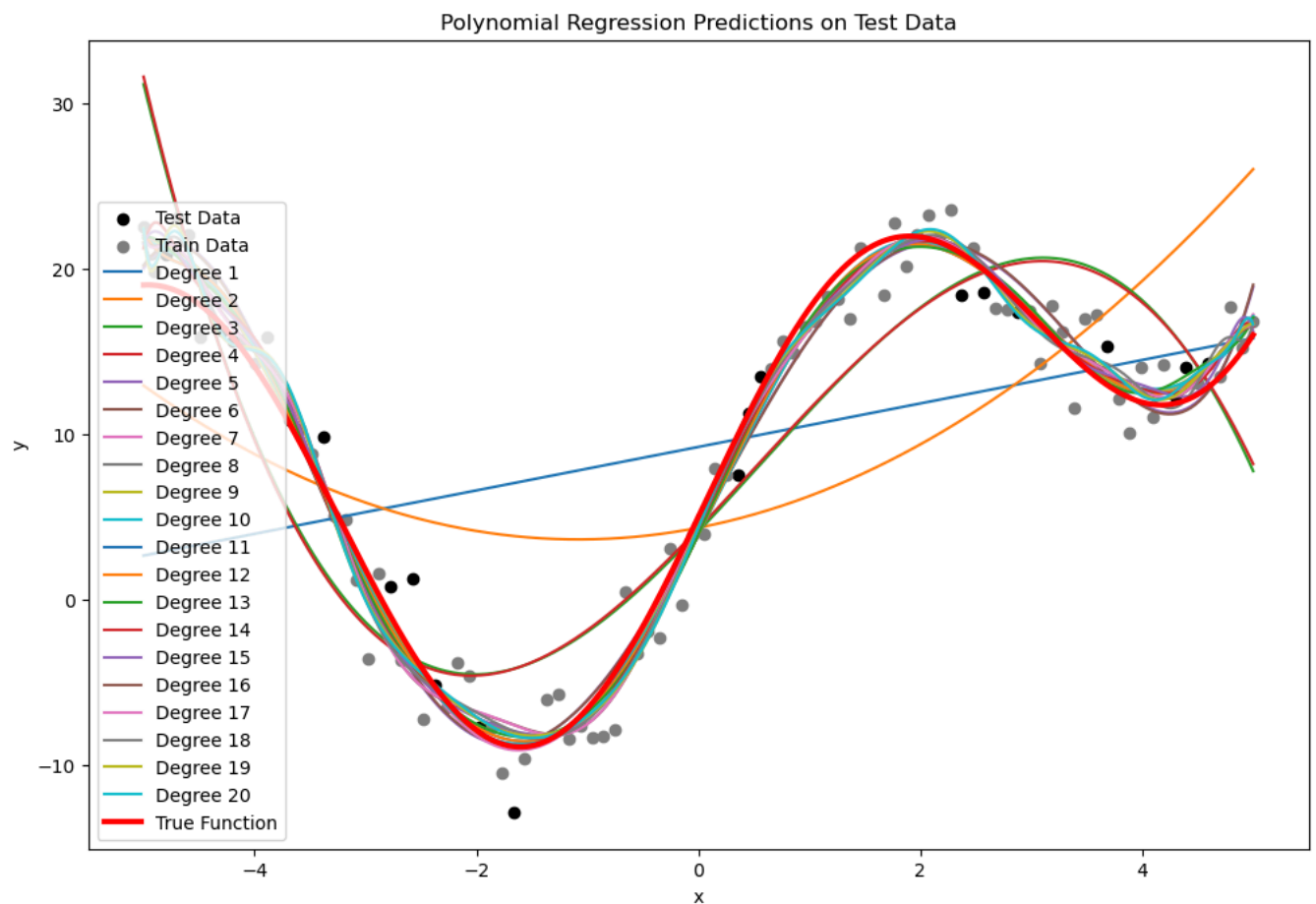
# Define the true function f(x)
def f(x):
    return 12 * np.sin(x) + 0.5 * x**2 + 2 * x + 5

for degree in degrees:
    model = PolynomialRegression(degree)
    model.fit(X_train, y_train)
    y_plot = model.predict(X_plot)

    plt.plot(X_plot, y_plot, label=f'Degree {degree}')

plt.plot(X_plot, f(X_plot), color='red', label='True Function', linewidth=3)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Polynomial Regression Predictions on Test Data')
plt.legend()
plt.show()

```



e. Repeat step d for the degree of polynomial taking value in [1, 2, 5, 8, 12, 14, 16, 18, 20].

```

In [72]: degrees = [1, 2, 5, 8, 12, 14, 16, 18, 20]

plt.figure(figsize=(12, 8))
plt.scatter(X_test, y_test, color='black', label='Test Data')

# Generate a fine grid for plotting
X_plot = np.linspace(X.min(), X.max(), 500).reshape(-1, 1)

# Define the true function f(x)
def f(x):
    return 12 * np.sin(x) + 0.5 * x**2 + 2 * x + 5

```

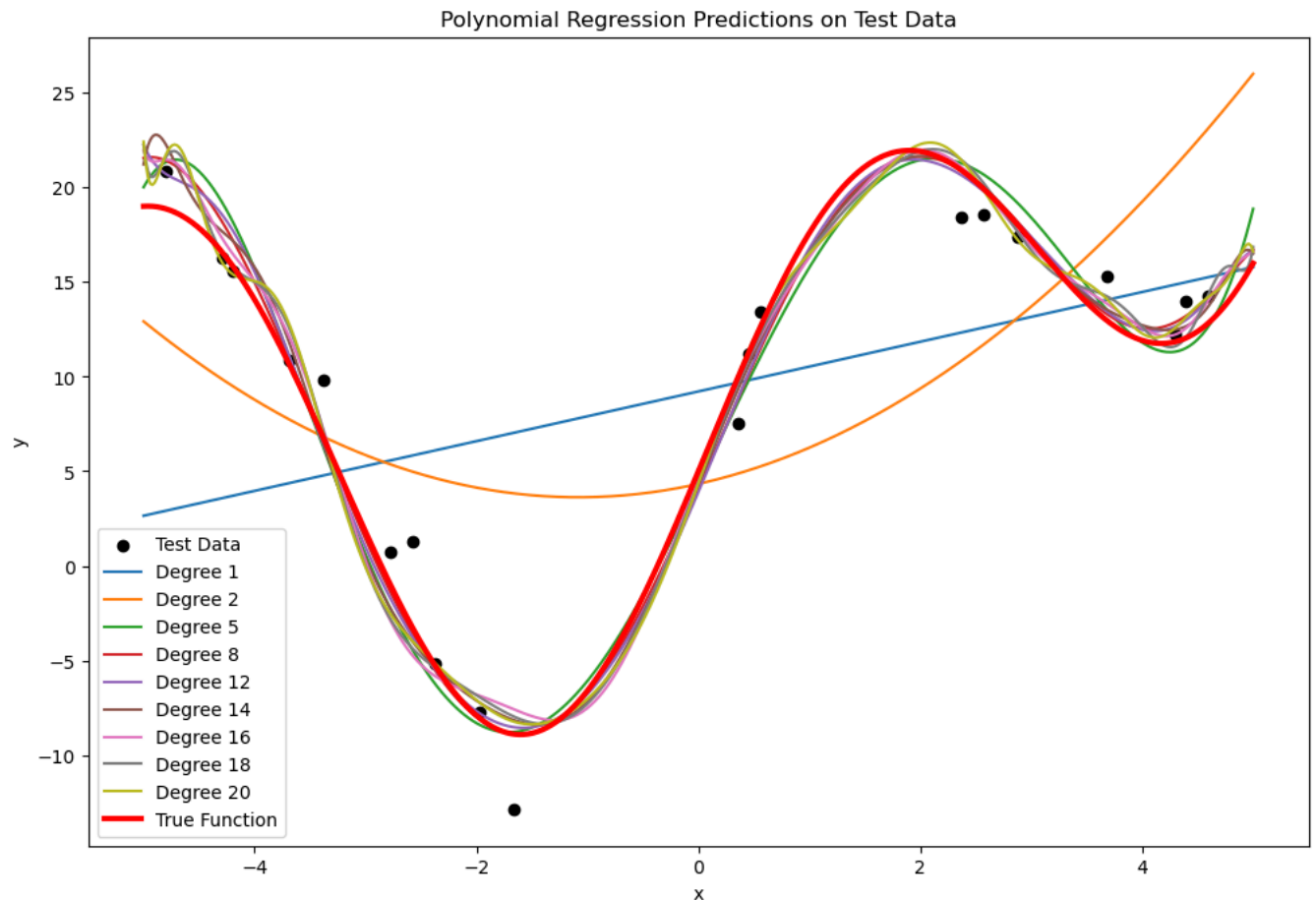
```

for degree in degrees:
    model = PolynomialRegression(degree)
    model.fit(X_train, y_train)
    y_plot = model.predict(X_plot)

    plt.plot(X_plot, y_plot, label=f'Degree {degree}')

plt.plot(X_plot, f(X_plot), color='red', label='True Function', linewidth=3)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Polynomial Regression Predictions on Test Data')
plt.legend()
plt.show()

```



f. Observe the plots, which polynomial degree gives the model the best fitting to the training data? Is that degree optimal for the model?

Based on Observation, the polynomial with degree 16 gives the model the best fitting to the training data. However, the degree that fits the training data best usually the high degree is not necessarily optimal for the model. Then we calculate the MSE for each degree. MSE of degree 8 is the smallest which means that polynomial with degree 8 are the best model on the test set.

```

In [73]: # Calculate the mean square error in test set for each degree
test_errors = []

# Train models for each degree and predict on test set
for degree in degrees:
    # Create a pipeline for the current degree
    model = Pipeline([
        ('polynomial_features', PolynomialFeatures(degree=degree)),
        ('linear_regression', LinearRegression())
    ])

    # Train on the training set
    model.fit(x_train, y_train)

    # Predict on the test set

```

```

y_test_pred = model.predict(x_test)

# Calculate the mean squared error
mse = mean_squared_error(y_test, y_test_pred)

# Append to the list of test errors
test_errors.append(mse)

# Print the test errors for each degree
for degree, mse in zip(degrees, test_errors):
    print(f'MSE of Degree {degree}: {mse:.2f}')

```

```

MSE of Degree 1: 79.27
MSE of Degree 2: 59.13
MSE of Degree 5: 6.26
MSE of Degree 8: 4.08
MSE of Degree 12: 4.46
MSE of Degree 14: 4.87
MSE of Degree 16: 5.40
MSE of Degree 18: 5.12
MSE of Degree 20: 4.60

```

g. Plot training errors and test errors curves (mean square error) with regard to polynomial degrees. Based on the plots, verify your observation in c. Describe what is underfitting and what is overfitting. Which degree of polynomial feature mapping gives the best generalization performance?

```

In [74]: from sklearn.metrics import mean_squared_error

# Define a range of polynomial degrees from 3 to 20
degrees_range = [1, 2, 5, 8, 12, 14, 16, 18, 20]

# Initialize lists to store training and test errors
train_errors = []
test_errors = []

# Loop through each degree and compute the MSE for both training and test sets
for degree in degrees_range:
    # Create a pipeline for the current degree
    model = Pipeline([
        ('poly_features', PolynomialFeatures(degree=degree)),
        ('linear_regression', LinearRegression())
    ])

    # Train on the training set
    model.fit(x_train, y_train)

    # Predict on the training set and calculate training error (MSE)
    y_train_pred = model.predict(x_train)
    train_errors.append(mean_squared_error(y_train, y_train_pred))

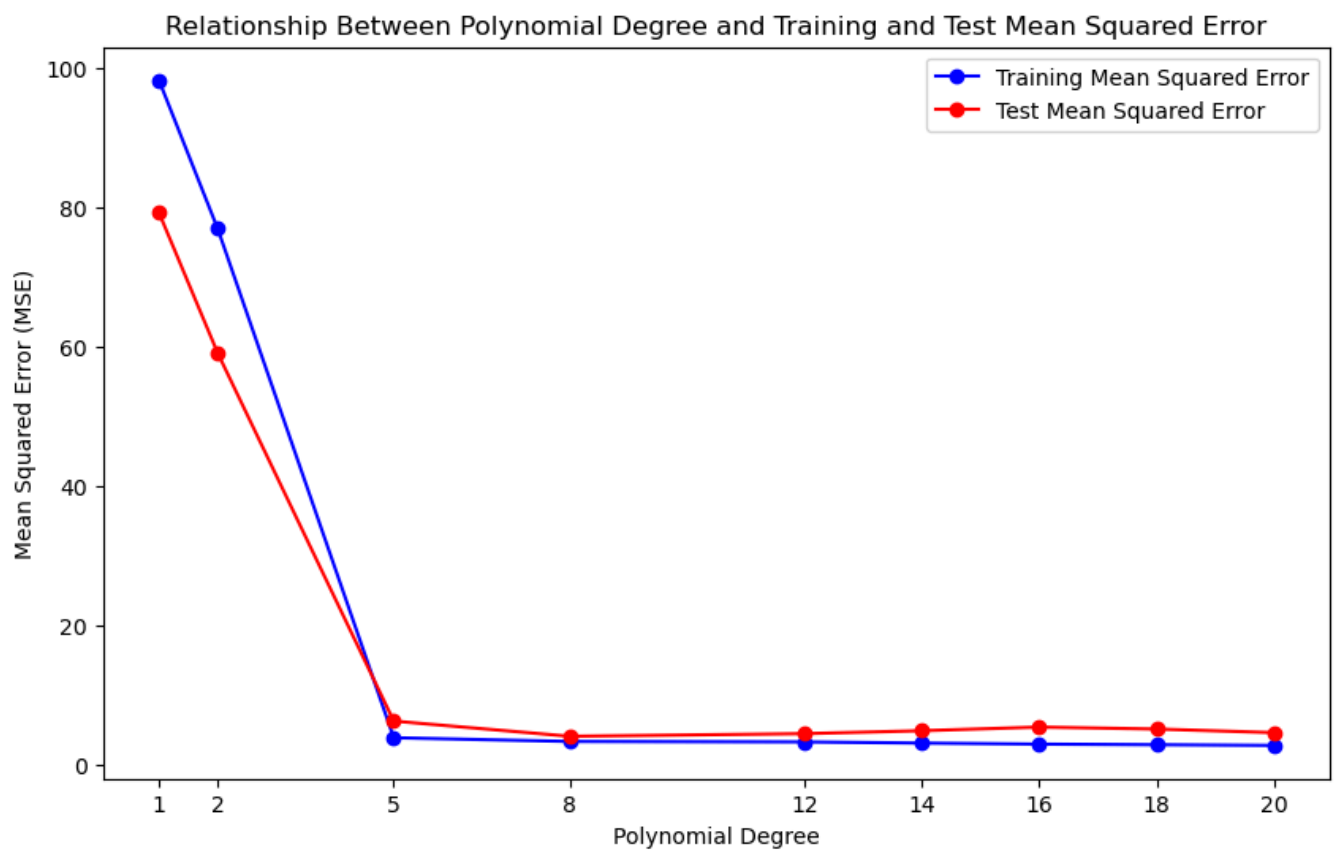
    # Predict on the test set and calculate test error (MSE)
    y_test_pred = model.predict(x_test)
    test_errors.append(mean_squared_error(y_test, y_test_pred))

# Plot training and test errors with respect to polynomial degree
plt.figure(figsize=(10, 6))

plt.plot(degrees_range, train_errors, label='Training Mean Squared Error', marker='o', color='blue')
plt.plot(degrees_range, test_errors, label='Test Mean Squared Error', marker='o', color='red')

plt.title("Relationship Between Polynomial Degree and Training and Test Mean Squared Error")
plt.xlabel("Polynomial Degree")
plt.ylabel("Mean Squared Error (MSE)")
plt.legend()
plt.xticks(degrees_range)
plt.show()

```



Underfitting occurs when a model is too simple to capture the underlying patterns in the data, resulting in high training and test errors. In this case, it happens when the polynomial degree is low, where the model cannot capture the complexity of the data.

Overfitting happens when a model is too complex and captures not only the true patterns but also the noise in the training data, resulting in low training error but high test error. This typically occurs with high polynomial degrees, although in this graph, after a certain degree, the model does not exhibit significant overfitting.

The polynomial degree that provides the best generalization performance, indicated by the lowest and most similar training and test errors, appears to be around degree 8. This range shows minimal error on both the training and test sets, suggesting the model accurately captures the data's patterns without overfitting.