ASSIGNMENT 2

NAME: ZAIN ZAHOOR

REG NO: FA21-BSE-166

SUBJECT: ARTIFICIAL INTELLIGENCE

CLASS: BSE 6B

SUBMITTED TO: SIR AHMAD SAEED KHATTAK

QUESTION 1:

Limitations of Using Average Mean Square Error (AMSE)

- **Bias-Variance Tradeoff:**
  AMSE can push us toward overly complicated models (like high-degree polynomials), which might result in overfitting.

- **Sensitivity to Outliers:**
  It disproportionately penalizes large deviations, making it vulnerable to outliers.

- **Interpretability Issue:**
  AMSE doesn't clearly show how well the model will perform on new data; it only measures how well it fits the given data**.**

- **Dependence on Scale:**
  AMSE values depend on the scale of the data, making comparisons across different datasets tricky

Suggested Alternative Measure

To address these issues, **Adjusted R^2** or **Cross-Validation Error** is recommended:

- **Adjusted R^2:**

  Considers model complexity by penalizing extra predictors.

- **Cross-Validation Error:**

  Measures how the model might perform on new data by splitting the dataset for training and testing.

We'll include Adjusted R^2 in the code implementation below for testing the goodness of fit.

Code:

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.metrics import mean_squared_error

from sklearn.model_selection import train_test_split

from numpy.polynomial.polynomial import Polynomial


def generate_data(n_points=100, noise_level=0.1):

    x = np.linspace(-10, 10, n_points)

    y = 3 * x**2 + 2 * x + 5 + noise_level * np.random.randn(n_points)

    return x, y
```

```python
def adjusted_r2(y_true, y_pred, n_params):
    n = len(y_true)
    r2 = 1 - (np.sum((y_true - y_pred) ** 2) / np.sum((y_true - np.mean(y_true)) ** 2))
    adj_r2 = 1 - (1 - r2) * (n - 1) / (n - n_params - 1)
    return adj_r2


def fit_polynomial(x, y, degree):
    coeffs = np.polyfit(x, y, degree)
    poly = np.poly1d(coeffs)
    y_pred = poly(x)
    amse = mean_squared_error(y, y_pred)
    adj_r2 = adjusted_r2(y, y_pred, degree)
    return poly, amse, adj_r2


# Part 1:
def print_amse():
    x, y = generate_data()
    _, amse, _ = fit_polynomial(x, y, degree=2)
    print(f"AMSE for degree 2 polynomial: {amse}")


# Part 2:
def goodness_of_fit():
    x, y = generate_data()
    poly1, amse1, adj_r2_1 = fit_polynomial(x, y, degree=1)
    poly2, amse2, adj_r2_2 = fit_polynomial(x, y, degree=2)
```

```python
    print("Goodness of Fit:")
    print(f"Degree 1 - AMSE: {amse1:.4f}, Adjusted R²: {adj_r2_1:.4f}")
    print(f"Degree 2 - AMSE: {amse2:.4f}, Adjusted R²: {adj_r2_2:.4f}")
```

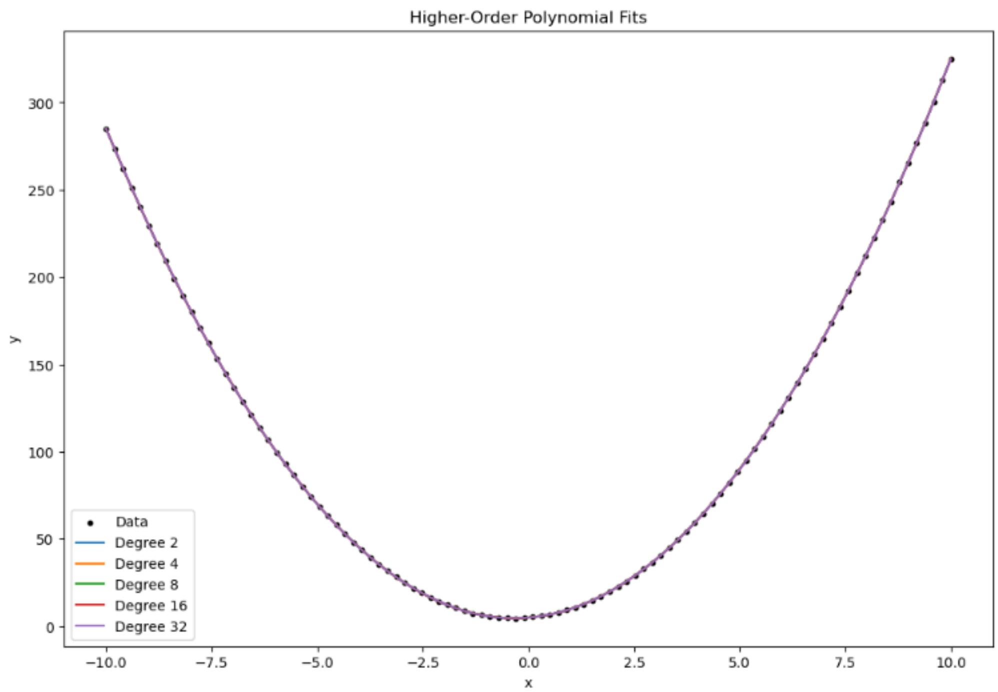
 # Part 3:

```python
def higher_order_comparison():
    x, y = generate_data()
    degrees = [2, 4, 8, 16, 32]
    amse_list = []
    adj_r2_list = []

    plt.figure(figsize=(12, 8))
    plt.scatter(x, y, label="Data", color="black", s=10)

    for degree in degrees:
        poly, amse, adj_r2 = fit_polynomial(x, y, degree)
        amse_list.append(amse)
        adj_r2_list.append(adj_r2)
        plt.plot(x, poly(x), label=f"Degree {degree}")

    plt.title("Higher-Order Polynomial Fits")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend()
    plt.show()
```

```
    print("Higher-Order Fit Comparison:")
    for i, degree in enumerate(degrees):
        print(f"Degree {degree} - AMSE: {amse_list[i]:.4f}, Adjusted R²: {adj_r2_list[i]:.4f}")


print_amse()
goodness_of_fit()
higher_order_comparison()
```

RUNNING IN ANACONDA NAVIGATOR INSIDE JUPYTERLAB:

```
AMSE for degree 2 polynomial: 0.008892962257391463
Goodness of Fit:
Degree 1 - AMSE: 8327.0220, Adjusted R²: 0.0060
Degree 2 - AMSE: 0.0099, Adjusted R²: 1.0000
```




```
Higher-Order Fit Comparison:
Degree 2 - AMSE: 0.0084, Adjusted R²: 1.0000
Degree 4 - AMSE: 0.0082, Adjusted R²: 1.0000
Degree 8 - AMSE: 0.0075, Adjusted R²: 1.0000
Degree 16 - AMSE: 0.0069, Adjusted R²: 1.0000
Degree 32 - AMSE: 0.0054, Adjusted R²: 1.0000
```

[ ]:

QUESTION 2:

Why Did the Higher Polynomial Fit Give the Best Result?

- Higher-degree polynomials fit the training data perfectly because they memorize all the details even the random noise.

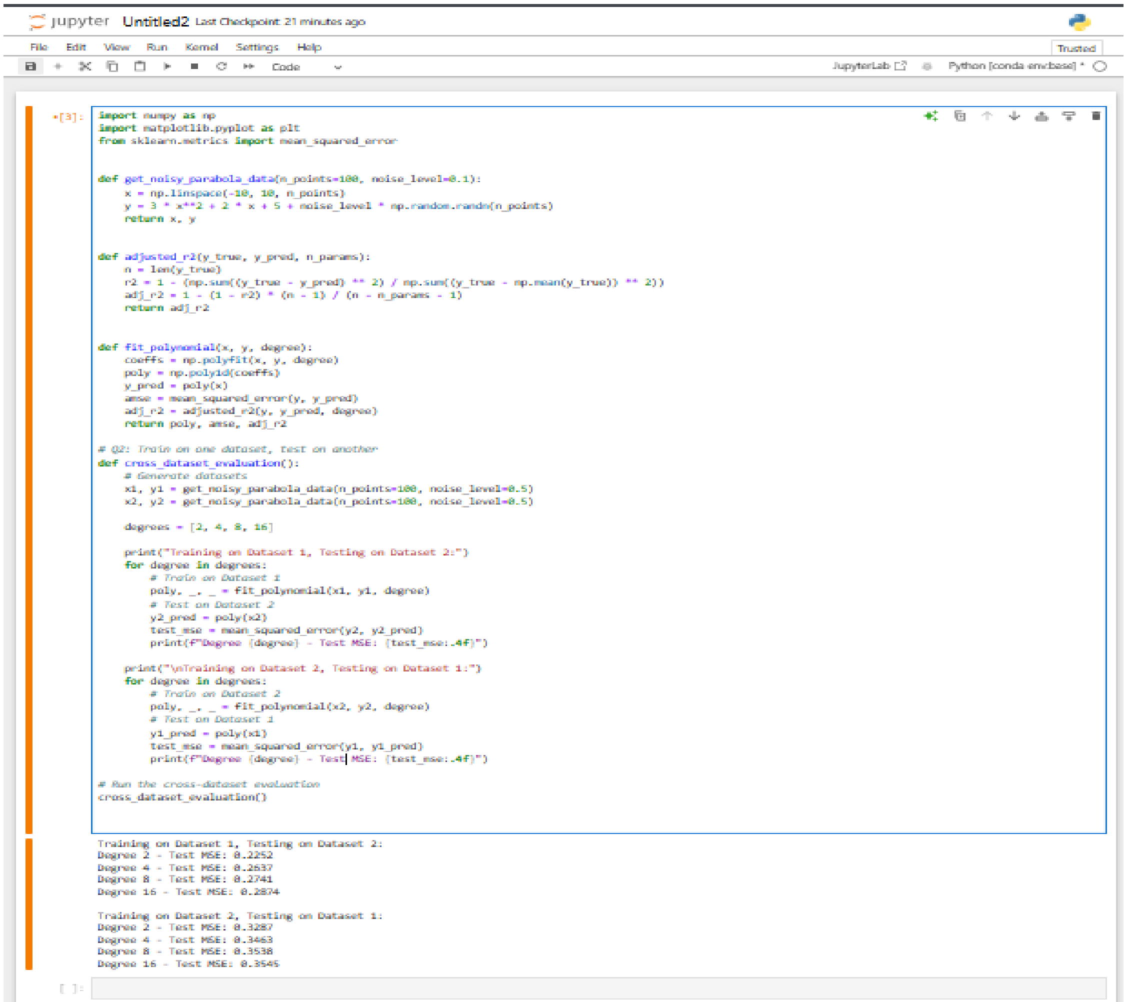
But this doesn't mean they're actually good models. They only work well on the training data and struggle with new data. So, while they might look like the best on the surface, they're not great for real-world use because they can't handle anything outside the training data.

CODE:

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.metrics import mean_squared_error


def get_noisy_parabola_data(n_points=100, noise_level=0.1):

    x = np.linspace(-10, 10, n_points)

    y = 3 * x**2 + 2 * x + 5 + noise_level * np.random.randn(n_points)

    return x, y


def adjusted_r2(y_true, y_pred, n_params):

    n = len(y_true)

    r2 = 1 - (np.sum((y_true - y_pred) ** 2) / np.sum((y_true - np.mean(y_true)) ** 2))

    adj_r2 = 1 - (1 - r2) * (n - 1) / (n - n_params - 1)

    return adj_r2


def fit_polynomial(x, y, degree):

    coeffs = np.polyfit(x, y, degree)

    poly = np.poly1d(coeffs)

    y_pred = poly(x)
```

```
    amse = mean_squared_error(y, y_pred)
    adj_r2 = adjusted_r2(y, y_pred, degree)
    return poly, amse, adj_r2

# Training on one dataset, test on another
def cross_dataset_evaluation():
    x1, y1 = get_noisy_parabola_data(n_points=100, noise_level=0.5)
    x2, y2 = get_noisy_parabola_data(n_points=100, noise_level=0.5)

    degrees = [2, 4, 8, 16]

    print("Training on Dataset 1, Testing on Dataset 2:")
    for degree in degrees
        poly, _, _ = fit_polynomial(x1, y1, degree)
        # Test on Dataset 2
        y2_pred = poly(x2)
        test_mse = mean_squared_error(y2, y2_pred)
        print(f"Degree {degree} - Test MSE: {test_mse:.4f}")

    print("\nTraining on Dataset 2, Testing on Dataset 1:")
    for degree in degrees:
        # Training on Dataset 2
        poly, _, _ = fit_polynomial(x2, y2, degree)
        # Testing on Dataset 1
        y1_pred = poly(x1)
        test_mse = mean_squared_error(y1, y1_pred)
```

```
    print(f"Degree {degree} - Test MSE: {test_mse:.4f}")

cross_dataset_evaluation()
```



QUESTION 3:

What Just Happened in Question 2?

When testing higher-degree polynomials on different datasets, we notice the following:

1. **Training on Dataset 1, Testing on Dataset 2:**

    The model learns the noise and details of Dataset 1 too closely, so it performs poorly on Dataset 2.

2. **Training on Dataset 2, Testing on Dataset 1:**

Similarly, the model memorizes Dataset 2's noise and struggles when tested on Dataset 1.

This phenomenon is called overfitting.

What Is Overfitting?

Overfitting happens when a model tries too hard to fit the training data, even capturing random noise and irrelevant details.

Explanation:

1. **Training Performance:**

   Higher-degree polynomials align perfectly with noisy training data.

2. **Testing Performance:**

   The model fails to generalize due to over-specialization.

Precautionary Measures in Real-Life Projects:

1. **Use Cross-Validation:**

   Divide your data into multiple parts and test the model on each part to check its consistency**.**

2. **Choose Simpler Models:**

   Avoid using overly complicated models unless when it's necessary.

3. **Regularization:**

   Use techniques like Lasso, Ridge, or ElasticNet to control model complexity.

4. **Test on Unseen Data:**

   Always test your model on a separate dataset to ensure it works on new data.

5. **Feature Engineering:**

   Create meaningful and relevant features instead of relying on a complex model to find patterns.