A blue and white logo

Description automatically generated

*ASSIGNMENT 2*

## NAME: TOHEED AHMAD QURESHI

## REG NO: FA21-BSE-156

## 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 givendata**.**

* **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 R²** or **Cross-Validation Error** is recommended:

* **Adjusted R²:**

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 R² 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

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)

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:**

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):

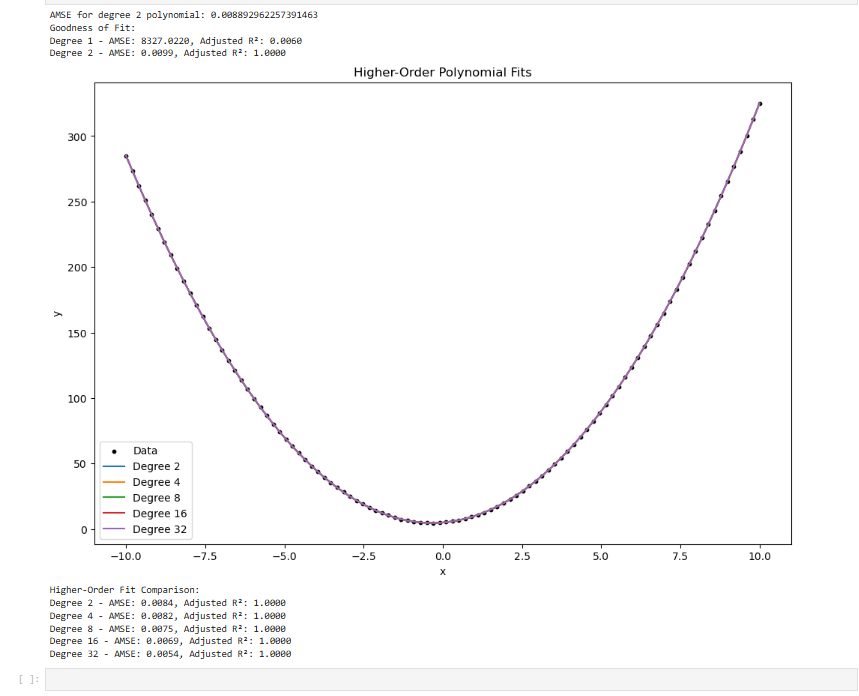
print(f"Degree {degree} - AMSE: {amse\_list[i]:.4f}, Adjusted R²: {adj\_r2\_list[i]:.4f}")

print\_amse()

goodness\_of\_fit()

higher\_order\_comparison()

**RUNNING IN ANACONDA NAVIGATOR INSIDE JUPYTERLAB:**



**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)

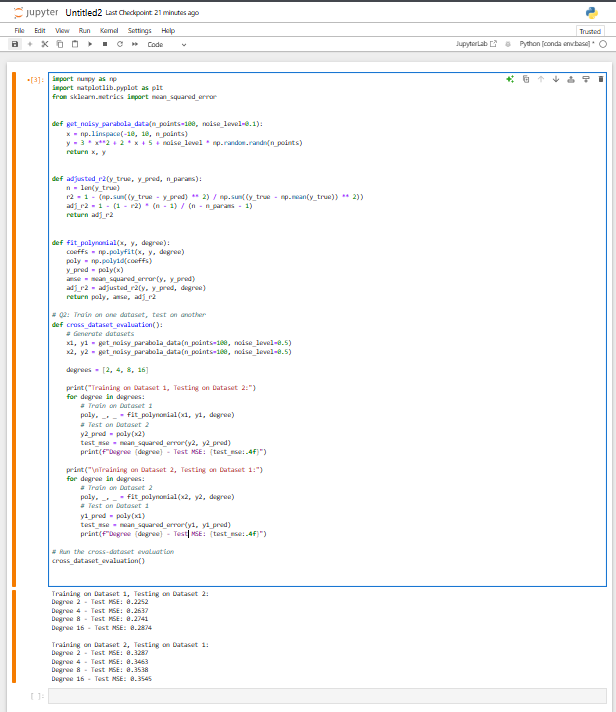
# Testing on Dataset 1

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.

1. **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.

1. **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**.**

1. **Choose Simpler Models:**

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

1. **Regularization:**

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

1. **Test on Unseen Data:**

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

1. **Feature Engineering:**

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