1.

함수 u(x)의 변화율을 나타내는 미분벡터

$$u' = \frac{g^T \Delta x}{\|\Delta x\|}$$

벡터

함수의 변화율을 나타내는 선형결합식

Transpose 연산

$(a+b)^T = a^T + b^T$

$(AB)^T = B^T A^T$

함수 u(x)의 모든 방향에서의 변화율을 나타내는 벡터

$$g^T \Delta x = \lim_{h \to 0} \frac{u(x + h\Delta x) - u(x)}{h}$$

벡터 Δx 방향에서의 함수의 변화율 $g'(x)$

$\Delta y = u(x + \Delta x) - u(x)$

$g^T \Delta x \approx \Delta y$

(a) $u(x) = b^T x$, $b \in R^n$

$\Delta y = b^T (x + \Delta x) - b^T x = b^T \Delta x$

$\|\Delta x \to 0\|$ 일때 $g^T \Delta x \approx \Delta y$ 이므로

$g^T \Delta x \approx b^T \Delta x$

$\therefore$ 미분벡터 g는 $\boxed{g = b}$

(b) $u(x) = x^T A x$, $A \in R^{n \times n}$ (A가 정사각행렬인 경우)

$x : \underset{n \times 1}{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}}$ $\quad x^T : \underset{1 \times n}{[\, x_1 \, x_2 \cdots x_n \,]}$ $\quad A : n \times n$

$\Delta y = (x + \Delta x)^T A (x + \Delta x) - x^T A x$

$= (x^T + \Delta x^T)(Ax + A\Delta x) - x^T A x$

$= \cancel{x^T A x} + x^T A \Delta x + \Delta x^T A x + \Delta x^T A \Delta x - \cancel{x^T A x}$

$= x^T A \Delta x + \Delta x^T A x + \Delta x^T A \Delta x$

$\|\Delta x\| \to 0$ 일때 $g^T \Delta x \approx \Delta y$ 이므로

$\Delta y = x^T A \Delta x + \Delta x^T A x + \cancel{\Delta x^T A \Delta x}$

작은변화 Δx에 대해 $\Delta x^T A \Delta x$ 항은 무시할 수 있다.

$\Delta x^T A x$ 는 1×1 스칼라이므로, 이를 전치하여 $x^T A^T \Delta x$ 로 바꿀 수 있다.

$\Delta y = x^T A \Delta x + x^T A^T \Delta x$

$= x^T (A + A^T) \Delta x$

$g^T \Delta x \approx \Delta y$ 이므로

$g^T \Delta x \approx x^T (A + A^T) \Delta x$

$\boxed{\therefore \; g = (A^T + A)x}$ $\cdots$ ㉠

(c) ㉠ 에서 $A^T = A$ 이므로

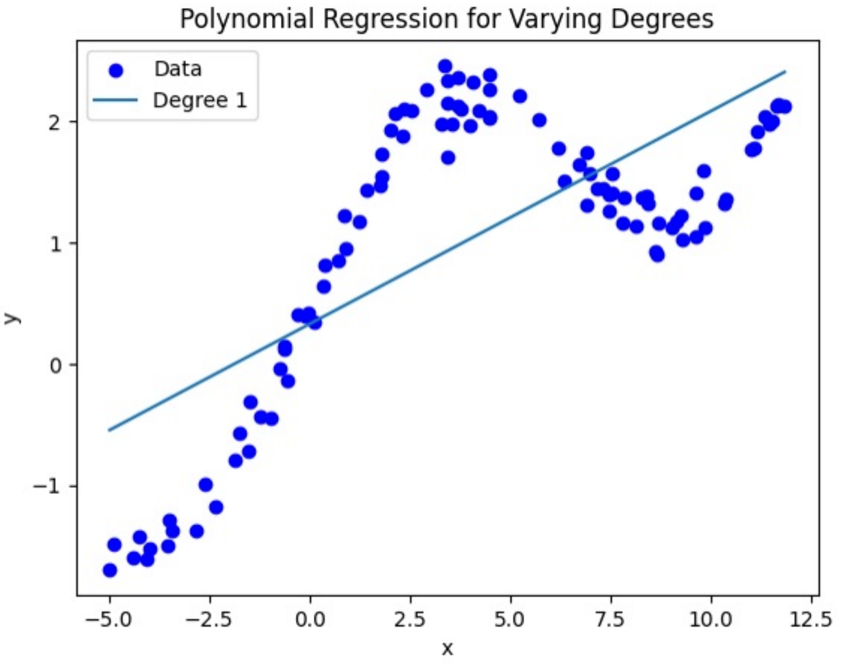
$g = (A^T + A)x = (A + A)x = 2Ax$

$\boxed{\therefore \; g = 2Ax}$

2.

a)

```python
import numpy as np
import matplotlib.pyplot as plt

# Load the data from the files
x_data = np.loadtxt('hw1x.dat')
y_data = np.loadtxt('hw1y.dat')

# 호환성을 위해 데이터 차원을 재조정
x_data = x_data.reshape(-1, 1)
y_data = y_data.reshape(-1, 1)


# 주어진 차수(degree)에 대해 다항 회귀를 수행하고, 학습된 가중치와 예측값 반환
def polynomial_regression(degree):
    # polynomial features 생성
    x_poly = np.hstack([x_data**i for i in range(0, degree +
1)])
    # Closed-form solution (정규 방정식) 을 통한 계수값 계산
    X = np.array(x_poly)
    Y = np.array(y_data)

    # X.T @ X는 (X^T X), np.linalg.inv()는 역행렬을 계산
    beta = np.linalg.inv(X.T @ X) @ X.T @ Y
    print(f'Closed-form solution for beta: {beta}')

    y_pred = X @ beta

    return y_pred

plt.scatter(x_data, y_data, color='blue', label='Data')

# x_data를 정렬한 후 정렬된 인덱스 기반으로 정렬
sorted_idx = np.argsort(x_data[:, 0])
sorted_x_data = x_data[sorted_idx]


# degree에 따른 다항 회귀
y_pred = polynomial_regression(1)

# y_pred도 정렬된 순서에 맞추기
sorted_y_pred = y_pred[sorted_idx]

# 각 차수에 대한 결과를 플롯에 추가
plt.plot(sorted_x_data, sorted_y_pred, label=f'Degree 1')


# 그래프 레이블 설정
plt.title('Linear Regression Using Closed-form solution')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

# 결과 플롯 출력
plt.show()
```



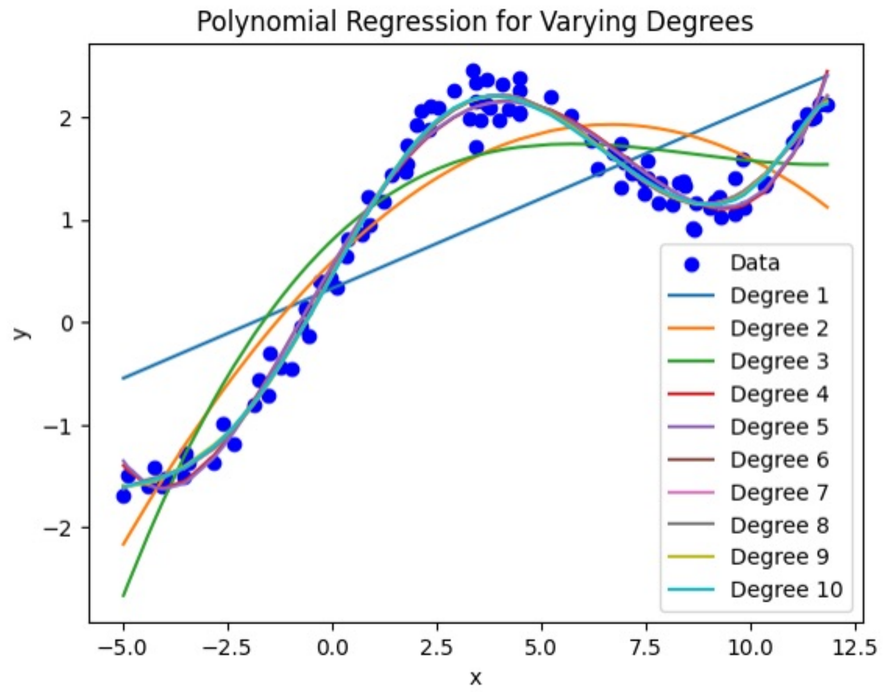
Closed-form solution for beta:
[[0.32767539]
 [0.17531122]]

정규방정식을 통해 θ를 계산한 결과

$$\boxed{\theta_0 = 0.32767539 \qquad \theta_1 = 0.17531122}$$

b)

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Load the data from the files
x_data = np.loadtxt('hw1x.dat')
y_data = np.loadtxt('hw1y.dat')

# 호환성을 위해 데이터 차원을 재조정
x_data = x_data.reshape(-1, 1)
y_data = y_data.reshape(-1, 1)


# 주어진 차수(degree)에 대해 다항 회귀를 수행하고, 학습된 가중치와 예측값 반환
def polynomial_regression(degree):

    # polynomial features 생성
    x_poly = np.hstack([x_data**i for i in range(0, degree + 1)])

    # Closed-form solution (정규 방정식) 을 통한 계수값 계산
    X = np.array(x_poly)
    Y = np.array(y_data)

    # 1. 정규방정식을 통해 beta를 구하고 y_pred 예측하기

    # X.T @ X는 (X^T X), np.linalg.inv()는 역행렬을 계산
    beta = np.linalg.inv(X.T @ X) @ X.T @ Y

    print(f'Closed-form solution for beta: {beta}')


    y_pred = X @ beta

    # 2. LinearRegression 모델을 활용해서 y_pred 예측하기
    #lr = LinearRegression()
    #lr.fit(x_poly, y_data)
    #y_pred = lr.predict(x_poly)

    return y_pred

# 오버핏 문제를 관찰하기 위해 n을 반복적으로 증가시킴
def observe_overfitting(max_degree):
    plt.scatter(x_data, y_data, color='blue', label='Data')

    # x_data를 정렬한 후 정렬된 인덱스 기반으로 정렬
    sorted_idx = np.argsort(x_data[:, 0])
    sorted_x_data = x_data[sorted_idx]

    for degree in range(1, max_degree + 1):
        # degree에 따른 다항 회귀
        y_pred = polynomial_regression(degree)

        # y_pred도 정렬된 순서에 맞추기
        sorted_y_pred = y_pred[sorted_idx]

        # 각 차수에 대한 결과를 플롯에 추가
        plt.plot(sorted_x_data, sorted_y_pred, label=f'Degree
{degree}')

    # 그래프 레이블 설정
    plt.title('Polynomial Regression for Varying Degrees')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()

    # 결과 플롯 출력
    plt.show()

# 차수를 증가시키면서 오버피팅 관찰
observe_overfitting(max_degree=10)
```



처음에 n=1 로 시작하면 매우 단순한 선형 모델이 만들어진다. 복잡도가 낮기 때문에 데이터의 전반적인 경향만을 파악하고 모든 데이터 포인트에 완벽히 맞추지 못할 수 있다. 그러나, n을 증가시킴에 따라 모델의 복잡도가 증가하여 훈련 데이터에 점점 더 잘 맞게 되지만 어느 시점에서는 모델이 훈련 데이터의 작은 변동이나 노이즈에 민감하게 반응하게 되어 훈련 데이터에 매우 잘 맞는 반면, 새로운 데이터(테스트 데이터)에서는 성능이 떨어지는 과적합이 발생한다. N=10 차의 그래프에서는 데이터의 모든 작은 변화까지 모델이 과도하게 따라가는 것을 관찰할 수 있다.

3.

a)

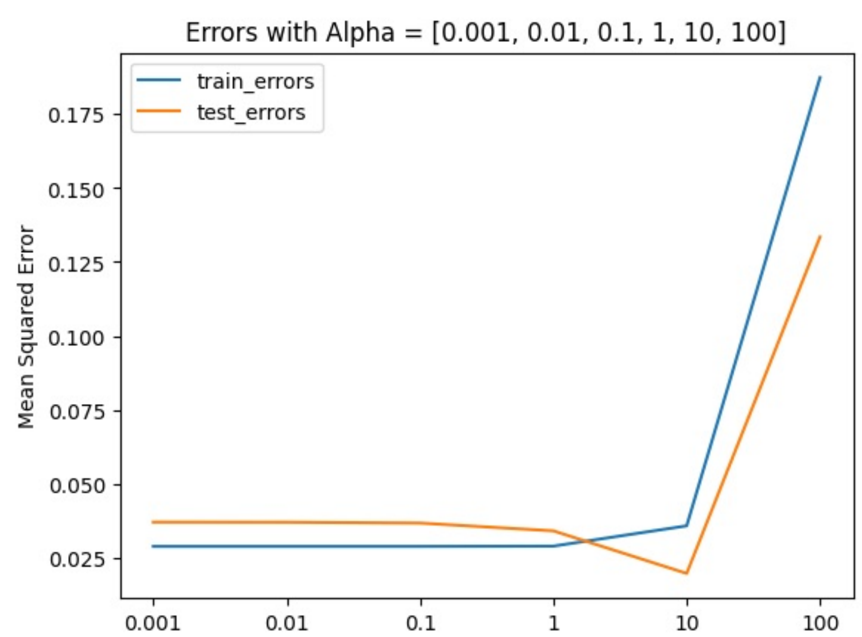
본 코드는 Ridge 회귀를 사용하여 같은 n = 5차 다항 회귀에 대하여 다양한α값에서 모델의 과적합(overfitting)을 관찰하는 실험을 수행한다.

Ridge 회귀는 L2 정규화를 사용하여 모델의 과적합(overfitting)을 줄이는 회귀 기법이다. 그 원리는 모델의 가중치(파라미터) 크기에 패널티를 부과하여, 너무 큰 가중치가 생기는 것을 방지하는 데 있다. Ridge 회귀는 손실 함수에 L2 정규화 항을 추가하며, 이 추가 항은 가중치(계수)의 제곱합에 비례하며, 다음과 같은 형태로 나타난다.

$$J(\theta) = \sum_{i=1}^{m} \left(y^{(i)} - \theta^T x^{(i)}\right)^2 + \alpha \sum_{j=1}^{n} \theta_j^2$$

이 때, α 값은 Ridge 회귀에서 정규화 정도를 제어하는 하이퍼파라미터로서, 0.001 ~ 100 까지 값을 변화시키면서 손실 함수MSE 의 변화율을 관찰할 것이다.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

# 로컬 파일에서 데이터 로드
x_data = np.loadtxt('hw1x.dat')
y_data = np.loadtxt('hw1y.dat')

# 호환성을 위해 모양 변경
x_data = x_data.reshape(-1, 1)
y_data = y_data.reshape(-1, 1)

# 데이터를 훈련 세트와 테스트 세트로 분할
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.3,
random_state=42)
# 다양한 alpha 값에 대해 Ridge 회귀 수행
def ridge_regression(degree, alpha):
    # 다항식 특징 생성
    x_poly_train = np.hstack([x_train**i for i in range(0, degree + 1)])
    x_poly_test = np.hstack([x_test**i for i in range(0, degree + 1)])

    # Ridge 회귀 모델
    ridge = Ridge(alpha=alpha)
    ridge.fit(x_poly_train, y_train)

    # 값 예측
    y_train_pred = ridge.predict(x_poly_train)
    y_test_pred = ridge.predict(x_poly_test)

    # 오차 계산
    train_error = mean_squared_error(y_train, y_train_pred)
    test_error = mean_squared_error(y_test, y_test_pred)

    return y_train_pred, y_test_pred, train_error, test_error

# Ridge 회귀에서 다양한 alpha 값으로 과적합 관찰
def observe_ridge_regularization(max_degree, alphas):
    train_errors = []
    test_errors = []

    sorted_idx_train = np.argsort(x_train[:, 0])
    sorted_x_train = x_train[sorted_idx_train]

    sorted_idx_test = np.argsort(x_test[:, 0])
    sorted_x_test = x_test[sorted_idx_test]

    for alpha in alphas:
        y_train_pred, y_test_pred, train_error, test_error = ridge_regression(5, alpha)
        train_errors.append(train_error)
        test_errors.append(test_error)

    # 오차 그래프
    plt.plot(np.log10(alphas), train_errors, label=f'train_errors')
    plt.plot(np.log10(alphas), test_errors, label=f'test_errors')
    plt.xticks(np.log10(alphas), alphas)
    plt.ylabel('평균 제곱 오차 (Mean Squared Error)')
    plt.legend()
    plt.title(f'Alpha 값에 따른 오차 (Alphas = {alphas})')
    plt.show()

# 다양한 alpha 값으로 테스트
observe_ridge_regularization(max_degree=3, alphas=[0.001, 0.01, 0.1, 1, 10, 100])
```



[결과 분석]

• α값이 작을 때에는 Ridge 정규화가 거의 적용되지 않아 여전히 모델이 테스트 세트보다 훈련 세트에 더 잘 적응 하는 과적합 상태이다.
• 그러나, alpha 값이 커질수록 강한 정규화가 적용되어, 모델이 과적합이 줄어들고 테스트 오차가 줄어드는 경향을 보일 수 있다
o L2 정규화 항은 가중치 벡터의 크기에 패널티를 부과하므로, 모델이 최적화를 통해 큰 가중치를 갖지 못하도록 한다.
o 과적합된 모델은 보통 훈련 데이터에 지나치게 잘 맞추기 위해 가중치가 커지는 경향이 있는데 Ridge 회귀는 이를 억제하여 모델이 더 단순해지도록 한다.
• 적절한 α값인 α= 10 을 선택하였을 때, 모델이 과적합을 피하면서도 충분한 복잡성을 유지하도록 된다.

4. locally weighted linear regression

서로 다른 training sample에 대하여
각각 다른 가중치 적용

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m} w^{(i)} \left(\theta^T x^{(i)} - y^{(i)}\right)^2$$

a) 가중치 손실 함수 $J(\theta)$는 다음과 같이 정의된다.

data 수

i번째 데이터 포인트의 특성 벡터

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m} w^{(i)} \left(\theta^T x^{(i)} - y^{(i)}\right)^2$$

i번째 데이터 포인트의 실제 타겟값

모델 가중치 벡터

i 번째 데이터 포인트의
실제 타겟값

⇒ 즉, 개별 데이터 포인트 $(x^{(i)}, y^{(i)})$에 대한 오차 $(\theta^T x^{(i)} - y^{(i)})$에 대해
가중치 $w^{(i)}$를 적용한 후 제곱합을 계산한것 임을 알 수 있다.

① X : $x^{(i)}$를 한데모은 특징 행렬

n개의 특성

$$\text{특징 행렬 } X = \overbrace{\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & & & \vdots \\ x_1^{(m)} & & \cdots & x_n^{m} \end{bmatrix}}^{n\text{개의 특성}} \Bigg\} m\text{개의 데이터}$$

X는 모든 데이터 포인트의 특징을 포함하는 n×m 크기의 행렬이다.

② $\vec{y}$: 실제 타겟값 $y^{(i)}$를 한데 모은 타겟 벡터

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

③ W : 가중치 $w^{(i)}$를 포함하는 대각 행렬

$$W = \mathrm{diag}(w^{(1)}, w^{(2)}, \dots, w^{(m)}) \qquad \begin{bmatrix} w^{(1)} & 0 & 0 \\ 0 & w^{(2)} & 0 \\ 0 & 0 \ddots & w^{(m)} \end{bmatrix}$$

앞서 정의한 X, y, W를 사용하여 주어진 손실 함수를 벡터 및 행렬 형태로 변환할 수 있다.

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m} w^{(i)} \left(\theta^T x^{(i)} - y^{(i)}\right)^2$$

$w^{(i)}, x^{(i)}, y^{(i)}$는 모두 앞서 정의한 행렬 W, X, Y의 i번째 데이터(행)을 의미한다.

Σ i=1 부터 m까지 반복하여 i번째 data point에 대하여 가중치× 오차^{2}을 더하고 있으므로

이 부분을 행렬로 바꾸어 전체 데이터 포인트에 대한 연산을 나타내도록 할 수 있다.

따라서 $\sum_{i=1}^{m}$을 지우고 $w^{(i)}, x^{(i)}, y^{(i)}$를 각각 열방향으로 m개의 data로 확장하여 행렬을 만들면

$$J(\theta) = \frac{1}{2} W (X\theta^T - \vec{y})^2$$ 이 유도된다.

$$J(\theta) = \frac{1}{2} \underbrace{\overset{W}{\begin{bmatrix} w_0 & & \\ & \ddots & \\ & & w_m \end{bmatrix}}}_{m\times m} \left(\underbrace{\overset{X}{\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & & & \vdots \\ x_1^{(m)} & & \cdots & x_n^{m} \end{bmatrix}}}_{m\times n} \underbrace{\overset{\theta^T}{\begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}}}_{n\times 1} - \underbrace{\overset{\vec{y}}{\begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}}}_{m\times 1} \right)^2$$

$A^2 = A^T A$ 이므로 이를 적용하면 최종적으로 아래와 같은 식을 얻을 수 있다.

$$J(\theta) = \frac{1}{2} W (X\theta^T - \vec{y})(X\theta^T - \vec{y})^T$$
$$= \frac{1}{2} (X\theta^T - \vec{y}) W (X\theta^T - \vec{y})^T$$

b) $$\nabla_\theta J(\theta) = \nabla_\theta \left(\frac{1}{2} W (X\theta - \vec{y})^2 \right)$$
$$= X^T W (X\theta - \vec{y})$$

주어진 손실함수를 최소화하는 값을 찾기위하여
그라디언트를 0으로 설정한다.

$X^T W (X\theta - \vec{y}) = 0$, 이를 풀면

$$X^T W X\theta - X^T W \vec{y} = 0$$
$$\theta = (X^T W X)^{-1} X^T W \vec{y}$$

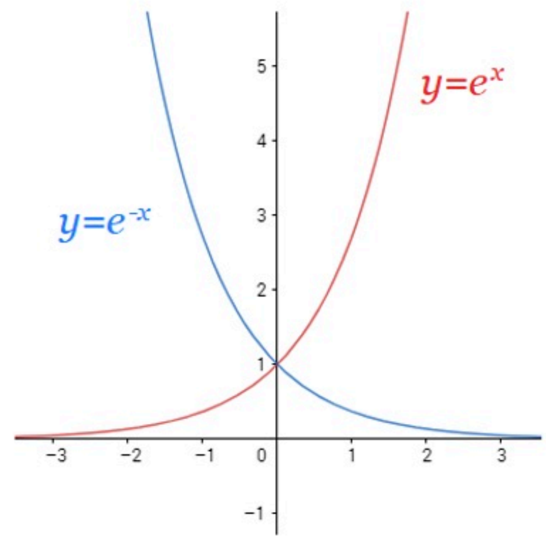
그리고 이것이 가중치가 포함된 설정에서의 θ 값이다.

c)

Locally Weighted Linear Regression(LWLR)은 **특정 데이터 포인트 주변의 데이터에 더 높은 가중치를 부여**하여 예측하는 **회귀 기법이다.** LWLR 에서는 쿼리 포인트를 기준으로, 그 주변에 있는 데이터 포인트들을 활용해 모델을 만들어 예측을 수행한다. 각 예측을 위한 모델이 해당 쿼리 포인트에 특화되기 때문에, 쿼리 포인트마다 각각 weight parameter 가 다른 모델이 형성된다.

주어진 가중치 함수:

$$w^{(i)} = \exp\left(-\frac{(x - x^{(i)})^2}{2\tau^2}\right)$$

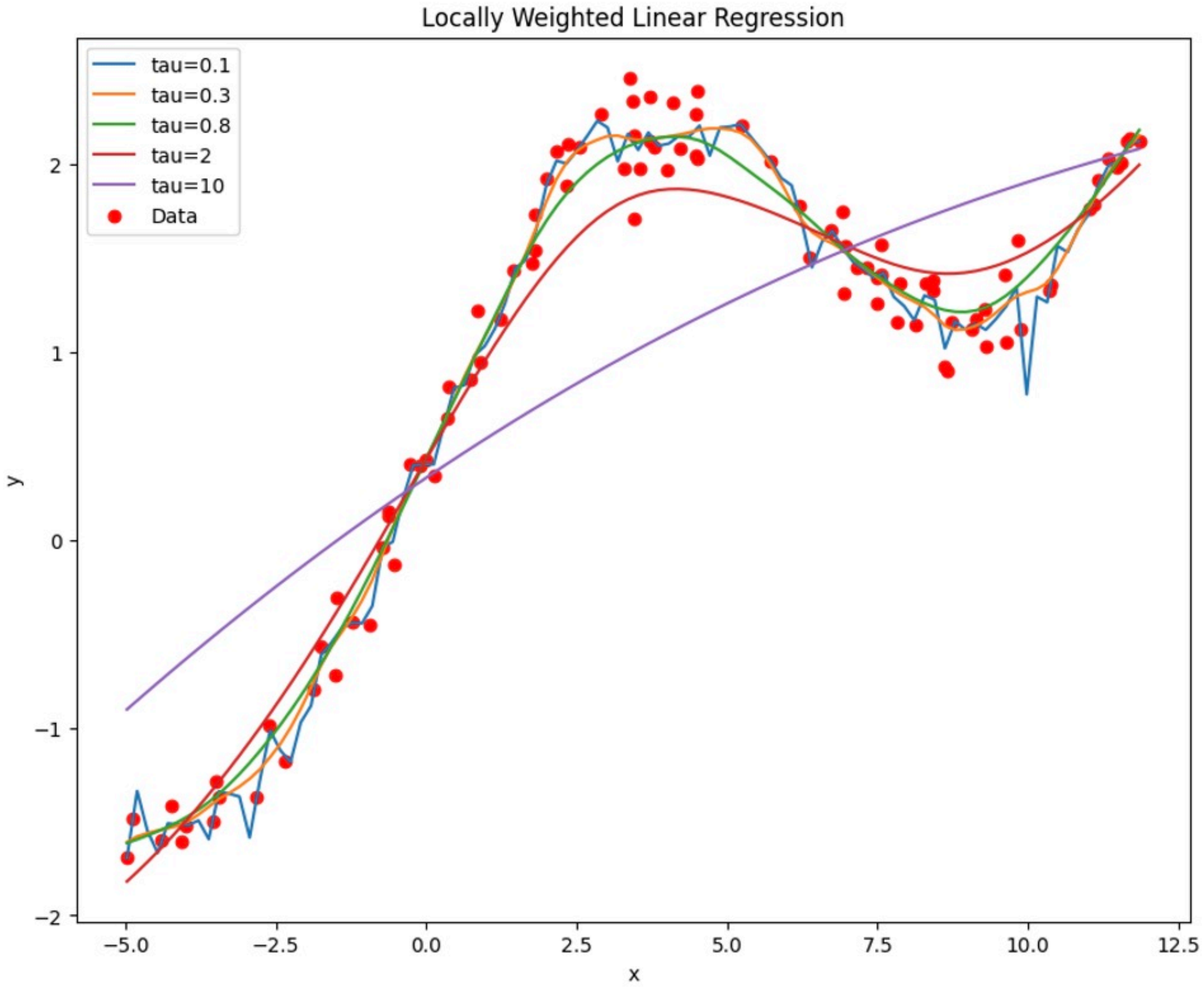


- 이 함수는 기준 데이터 x 와 **쿼리 데이터 포인트 x(i)** 사이의 거리에 따라 가중치를 할당한다.
    - x 와 $x^{(i)}$ 사이의 거리가 가까울수록 $(x - x^{(i)})^2$ 값이 작아지므로, 가중치 $w^{(i)}$는 1 에 가까워진다.
    - 반대로, 거리가 멀수록 $(x - x^{(i)})^2$ 값이 커지므로, 가중치 $w^{(i)}$는 0 에 가까워진다.

- τ 는 거리에 대한 가중치 감소의 **완만함**을 결정하는 파라미터이다.
    - τ 값이 **작으면:**
        - 가중치 함수의 분모에 작은 수가 들어가므로 x와 $x^{(i)}$ 사이의 거리가 조금만 멀어져도 $w^{(i)}$ 는 급격히 감소한다.
        - 이는 모델이 **매우 국소적으로** 데이터를 고려하도록 만들어, **쿼리 포인트 주변의 몇 개의 데이터 포인트에만 큰 가중치를 부여**하게 된다.
    - τ 값이 **크면**:
        - 가중치가 거리에 덜 민감하게 변화하여, 더 멀리 있는 데이터 포인트에도 높은 가중치를 부여할 수 있다.
        - 결과적으로 모델은 더 **전역적인 시야**를 가지고 데이터를 고려하며, **쿼리 포인트로부터 먼 포인트들의 영향**을 받게 된다.

```python
import numpy as np
import matplotlib.pyplot as plt

# 데이터 로드
x_data = np.loadtxt('hw1x.dat')
y_data = np.loadtxt('hw1y.dat')

# Reshape for compatibility
x_data = x_data.reshape(-1, 1)
y_data = y_data.reshape(-1, 1)

# Locally Weighted Linear Regression (LWLR) 구현
def locally_weighted_regression(x_query, tau):
    m = len(x_data)
    W = np.exp(-((x_data - x_query)**2) / (2 * tau**2))

    #print((x_data - x_query).shape)
    W = np.diag(W.flatten())  # 대각 행렬로 변환

    # Normal equation: theta = (X^T W X)^{-1} X^T W y
    X = np.hstack((np.ones_like(x_data), x_data))  # 상수항 추가
    theta = np.linalg.inv(X.T @ W @ X) @ X.T @ W @ y_data
    return theta

# 예측 함수
def predict(x_query, tau):
    theta = locally_weighted_regression(x_query, tau)
    return np.hstack([1, x_query]) @ theta

# 다양한 tau 값에 대해 결과 시각화
taus = [0.1,0.3,2,0.8,10]
x_range = np.linspace(min(x_data), max(x_data), 100)

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

for tau in taus:
    y_pred = [predict(x, tau) for x in x_range]
    plt.plot(x_range, y_pred, label=f'tau={tau}')

# 실제 데이터 플로팅
plt.scatter(x_data, y_data, color='red', label='Data')

plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('Locally Weighted Linear Regression')
plt.show()
    plt.title(f'Alpha 값에 따른 오차 (Alphas = {alphas})')
    plt.show()

# 다양한 alpha 값으로 테스트
observe_ridge_regularization(max_degree=3, alphas=[0.001, 0.01, 0.1, 1, 10,
100])
```

결국, τ 값은 쿼리 포인트 주변의 데이터를 얼마나 국소적으로 반영할지를 결정하는 중요한 파라미터로, 작은 τ 값은 모델이 쿼리 포인트 주변의 데이터에 매우 민감하게 반응하도록 하고, 큰 τ 값은 멀리 있는 데이터에도 영향을 받을 수 있도록 한다.



[결과 분석]

- **τ 값이 작으면** 모델이 매우 좁은 영역의 데이터 포인트에만 크게 가중치를 부여한다. τ 값이 너무 작은 경우, τ=0.1 인 곡선은 데이터를 정확히 따라가지만 지나치게 세세한 변동까지 반영하고 있어 전체적인 패턴을 포착하지 못한다.
- **τ 값이 크면** 넓은 영역의 데이터 포인트가 고려되므로, 모델이 더 많은 포인트에 균등한 가중치를 부여한다. 그래프에서 τ=10 인 곡선은 데이터의 세부적인 패턴을 놓치고 전반적으로 단순해진다.