580 Homework 3

Yiming Mao

February 18, 2025

1 Maximum Likelihood Estimation

(a) Compute the Likelihood for a Single Sample

For a single sample (x_i, y_i), the likelihood function is given by:

$$P((x_i, y_i); \mu_0, \mu_1) = P(y_i)P(x_i \mid y_i)$$

Since y_i follows a uniform distribution:

$$P(y_i = 0) = \frac{1}{2}, \quad P(y_i = 1) = \frac{1}{2}$$

And given y_i, $x_i \sim \mathcal{N}(\mu_{y_i}, I)$ with the probability density function:

$$q(x_i; \mu_{y_i}) = \frac{1}{(2\pi)^{d/2}} \exp\left(-\frac{1}{2}\|x_i - \mu_{y_i}\|_2^2\right)$$

Thus, the likelihood function is:

$$P((x_i, y_i); \mu_0, \mu_1) = \frac{1}{2} \cdot \frac{1}{(2\pi)^{d/2}} \exp\left(-\frac{1}{2}\|x_i - \mu_{y_i}\|_2^2\right).$$

(b) Compute the Log-Likelihood of the Training Dataset

For the dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$, the total likelihood is:

$$P(\mathcal{D} \mid \mu_0, \mu_1) = \prod_{i=1}^{n} P((x_i, y_i) \mid \mu_0, \mu_1).$$

Taking the log:

$$\log P(\mathcal{D} \mid \mu_0, \mu_1) = \sum_{i=1}^{n} \log P((x_i, y_i) \mid \mu_0, \mu_1).$$

Substituting from part (a):

$$\log P(\mathcal{D} \mid \mu_0, \mu_1) = \sum_{i=1}^{n} \left[\log \frac{1}{2} + \log \frac{1}{(2\pi)^{d/2}} - \frac{1}{2}\|x_i - \mu_{y_i}\|_2^2\right].$$

Simplifying:

$$\log P(\mathcal{D} \mid \mu_0, \mu_1) = -n \log 2 - \frac{nd}{2} \log(2\pi) - \frac{1}{2} \sum_{i=1}^{n} \|x_i - \mu_{y_i}\|_2^2.$$

(c) Compute the MLE Estimates of μ_0 and μ_1

The Maximum Likelihood Estimation (MLE) aims to maximize the log-likelihood:

$$\arg\max_{\mu_0, \mu_1} \log P(\mathcal{D} \mid \mu_0, \mu_1).$$

Since only the last term depends on μ_0, μ_1, we need to minimize:

$$\sum_{i=1}^{n} \|x_i - \mu_{y_i}\|_2^2.$$

Taking the derivative with respect to μ_0 and μ_1 and setting it to zero gives:

$$\mu_0^{MLE} = \frac{1}{n_0} \sum_{i \in S_0} x_i, \quad \mu_1^{MLE} = \frac{1}{n_1} \sum_{i \in S_1} x_i.$$

where: - S_0 is the set of indices where $y_i = 0$, and $n_0 = |S_0|$. - S_1 is the set where $y_i = 1$, and $n_1 = |S_1|$.

(d) Compute $P(y = 1|x; \mu_0, \mu_1)$

Using Bayes' theorem:

$$P(y = 1|x; \mu_0, \mu_1) = \frac{P(x|y = 1)P(y = 1)}{P(x|y = 0)P(y = 0) + P(x|y = 1)P(y = 1)}.$$

Since $P(y = 0) = P(y = 1) = \frac{1}{2}$, we get:

$$P(y = 1|x; \mu_0, \mu_1) = \frac{q(x; \mu_1)}{q(x; \mu_0) + q(x; \mu_1)}.$$

Using the Gaussian density:

$$q(x; \mu) = \frac{1}{(2\pi)^{d/2}} \exp\left(-\frac{1}{2}\|x - \mu\|_2^2\right).$$

Substituting and simplifying:

$$P(y = 1|x; \mu_0, \mu_1) = \frac{\exp((\mu_1 - \mu_0)^T x)}{C + \exp((\mu_1 - \mu_0)^T x)}$$

where:

$$C = \exp\left(\frac{1}{2}(\|\mu_1\|_2^2 - \|\mu_0\|_2^2)\right).$$

(e) When Does $P(y = 1|x)$ Match Logistic Regression?

The logistic regression posterior probability is given by:

$$P_{\text{logistic}}(y = 1|x; \mu_0, \mu_1) = \frac{\exp(\mu_1^T x)}{\exp(\mu_0^T x) + \exp(\mu_1^T x)}.$$

For the two probability expressions to match:

$$\frac{\exp((\mu_1 - \mu_0)^T x)}{C + \exp((\mu_1 - \mu_0)^T x)} = \frac{\exp(\mu_1^T x)}{\exp(\mu_0^T x) + \exp(\mu_1^T x)}.$$

Rearranging:

$$\exp(\mu_1^T x) = \exp((\mu_1 - \mu_0)^T x).$$

Since this must hold for all x, we conclude:

$$\mu_1^T x = (\mu_1 - \mu_0)^T x.$$

For this to be true for all x, we must have:

$$\mu_0 = 0.$$

Thus, the two probability models match **if and only if $\mu_0 = 0$**.

2 Maximum Likelihood Estimation for Gaussian Mean and Covariance

Let $\Sigma \in \mathbb{R}^{d\times d}$ be a symmetric, positive-definite matrix, and let $w \in \mathbb{R}^d$. The data samples are given by:

$$x_i \sim \mathcal{N}(w, \Sigma), \quad i = 1, \ldots, n$$

where the dataset is $\mathcal{D} = (x_1, \ldots, x_n)$.

(a) Log-Likelihood Function

The probability density function of a multivariate normal distribution $\mathcal{N}(w, \Sigma)$ is:

$$p(x_i \mid w, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x_i - w)^T\Sigma^{-1}(x_i - w)\right).$$

For n independent samples, the likelihood function is:

$$P(\mathcal{D} \mid w, \Sigma) = \prod_{i=1}^{n} p(x_i \mid w, \Sigma).$$

Taking the logarithm:

$$\log P(\mathcal{D} \mid w, \Sigma) = -\frac{nd}{2}\log(2\pi) - \frac{n}{2}\log|\Sigma| - \frac{1}{2}\sum_{i=1}^{n}(x_i - w)^T\Sigma^{-1}(x_i - w).$$

(b) Maximum Likelihood Estimator of w

To find the MLE of w, we maximize $\log P(\mathcal{D} \mid w, \Sigma)$. Taking the derivative with respect to w and setting it to zero:

$$\frac{\partial}{\partial w}\left(-\frac{1}{2}\sum_{i=1}^{n}(x_i - w)^T\Sigma^{-1}(x_i - w)\right) = 0.$$

Solving for w, we obtain:

$$w^{MLE} = \frac{1}{n}\sum_{i=1}^{n} x_i.$$

(c) Log-Likelihood with Σ as a Diagonal Matrix

Assume $w = 0$ and that Σ is diagonal, i.e., $\Sigma_{ij} = 0$ for $i \neq j$ and $\Sigma_{ii} = \frac{1}{v_i}$. The determinant and inverse are:

$$|\Sigma| = \prod_{i=1}^{d}\frac{1}{v_i}, \quad \Sigma^{-1} = \mathrm{diag}(v_1, v_2, \ldots, v_d).$$

The log-likelihood simplifies to:

$$\log P(\mathcal{D} \mid v_1, \ldots, v_d) = -\frac{nd}{2}\log(2\pi) + \frac{n}{2}\sum_{i=1}^{d}\log v_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{d} v_j x_{ij}^2.$$

(d) Maximum Likelihood Estimator of $v_1, \ldots, v_d$

To find the MLE of v_j, we take the derivative of the log-likelihood with respect to v_j and set it to zero:

$$\frac{n}{2v_j} - \frac{1}{2}\sum_{i=1}^{n} x_{ij}^2 = 0.$$

Solving for v_j, we obtain:

$$v_j^{MLE} = \frac{1}{n}\sum_{i=1}^{n} x_{ij}^2, \quad j = 1, \ldots, d.$$

(e) Log-Likelihood in Terms of Eigenvalues

Now, let $\Sigma = U\Lambda U^T$ be the eigenvalue decomposition, where $U \in \mathbb{R}^{d\times d}$ is an orthogonal matrix, and Λ is diagonal with eigenvalues $\lambda_1, \ldots, \lambda_d$. Given $w = 0$, the log-likelihood becomes:

$$\log P(\mathcal{D} \mid \lambda_1, \ldots, \lambda_d) = -\frac{nd}{2}\log(2\pi) - \frac{n}{2}\sum_{i=1}^{d}\log\lambda_i - \frac{1}{2}\sum_{i=1}^{n} x_i^T U\Lambda^{-1}U^T x_i.$$

Using the hint $\det(\Sigma) = \prod_{i=1}^{d}\lambda_i$, the MLE of λ_i is found similarly as:

$$\hat{\lambda}_i^{MLE} = \frac{1}{n}\sum_{j=1}^{n}(U^T x_j)_i^2, \quad i = 1, \ldots, d.$$

3 Cross validation on image block

(a)

Input

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Lasso
from sklearn.metrics import mean_squared_error

# Set seed for reproducibility
np.random.seed(42)

# Parameters
K = 8  # Image block size (KxK)
S = 30  # Number of sensed pixels
m = S // 6  # Validation pixels per iteration
M = 20  # Number of random subsets
lambdas = np.logspace(-3, 2, 6)  # 6 log-spaced values over 6 decades

# Generate a synthetic corrupted image block (KxK)
original_block = np.random.rand(K, K)  # Simulated true image block

# Randomly select S sensed pixels
sensed_indices = np.random.choice(K*K, S, replace=False)
sensed_pixels = np.zeros((K, K))
sensed_pixels.flat[sensed_indices] = original_block.flat[sensed_indices]

# Pick the first random subset for validation
validation_indices = np.random.choice(sensed_indices, m, replace=False)
training_indices = np.setdiff1d(sensed_indices, validation_indices)

# Prepare training and validation sets
X_train = np.array([np.unravel_index(i, (K, K)) for i in training_indices])  # Pixel positions
y_train = original_block.flat[training_indices]  # Pixel values

X_val = np.array([np.unravel_index(i, (K, K)) for i in validation_indices])
y_val = original_block.flat[validation_indices]

# Store results
reconstructed_blocks = []
errors = []
model_weights = []

# Apply LASSO regression for each lambda
for lambda_val in lambdas:
    lasso = Lasso(alpha=lambda_val)
    lasso.fit(X_train, y_train)  # Train model
    # Predict on validation set
    y_pred = lasso.predict(X_val)
    error = mean_squared_error(y_val, y_pred)
    errors.append(error)
    # Reconstruct sensed image block
    reconstructed_block = np.zeros((K, K))
    reconstructed_block.flat[training_indices] = y_train  # Training pixels remain the same
    reconstructed_block.flat[validation_indices] = y_pred  # Predicted values for validation
        set
    reconstructed_blocks.append(reconstructed_block)
    # Store model weights
    model_weights.append(lasso.coef_)

# Visualization: Display results for all 6 values of lambda
fig, axes = plt.subplots(6, 3, figsize=(15, 20))

for i, lambda_val in enumerate(lambdas):
    # 1. Reconstructed sensed image block
    axes[i, 0].imshow(reconstructed_blocks[i], cmap='gray', interpolation='nearest')
    axes[i, 0].set_title(f"Reconstructed␣(  ={lambda_val:.3e})")
    # 2. Original corrupted image block
    axes[i, 1].imshow(sensed_pixels, cmap='gray', interpolation='nearest')
    axes[i, 1].set_title("Original␣Corrupted␣Image␣Block")
    # 3. Model Weights Stem Plot
    axes[i, 2].stem(model_weights[i])
    axes[i, 2].set_title(f"Model␣Weights␣(  ={lambda_val:.3e})")

plt.tight_layout()
plt.show()
```

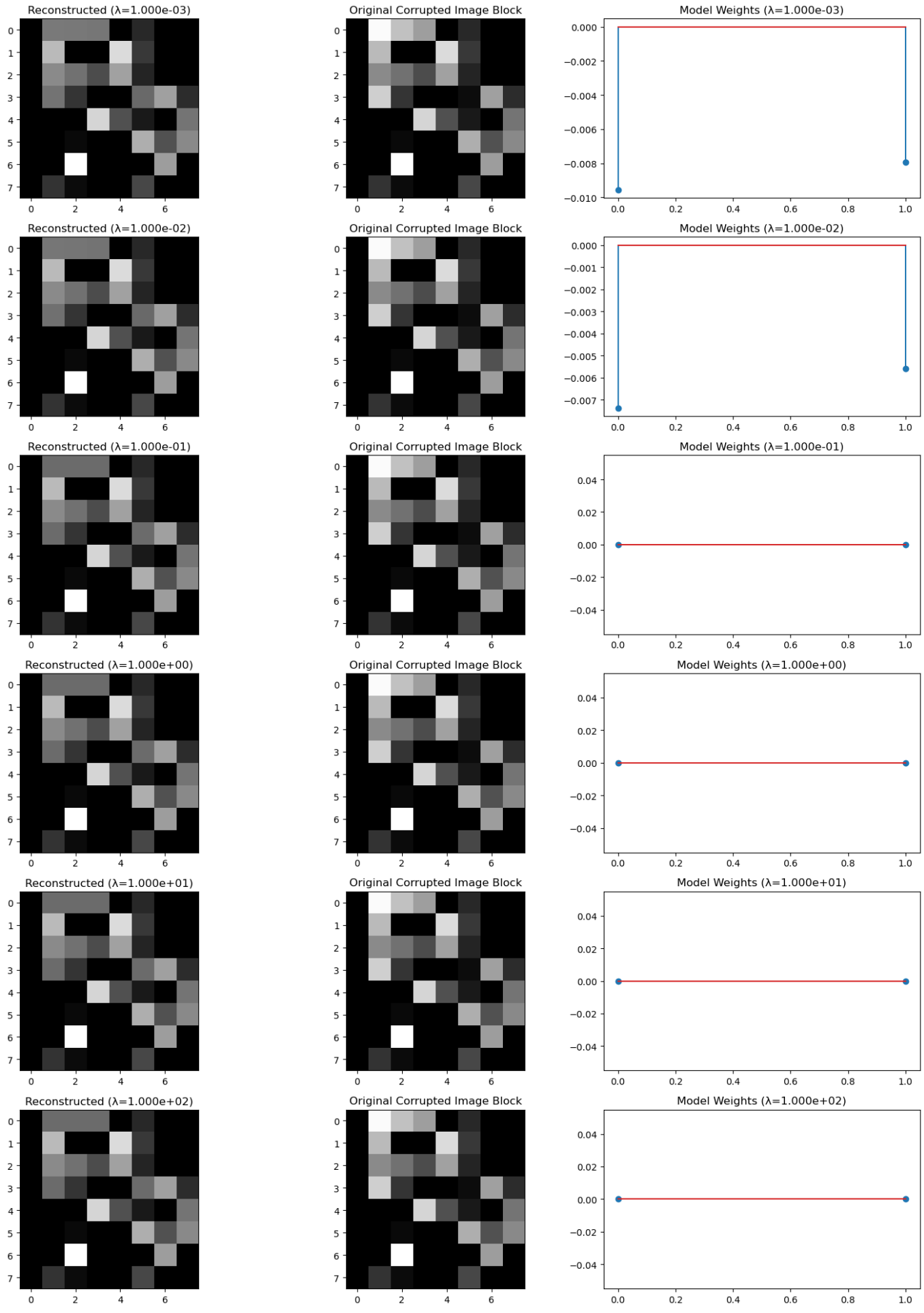

Figure 1: 3(a)Output

(b)

Input

```
# Store MSE values
mse_values = []

# Apply LASSO regression for each lambda
for lambda_val in lambdas:
    lasso = Lasso(alpha=lambda_val)
    lasso.fit(X_train, y_train)  # Train model

    # Predict on validation set
    y_pred = lasso.predict(X_val)
    error = mean_squared_error(y_val, y_pred)
    mse_values.append(error)

# Plot MSE vs. Lambda
plt.figure(figsize=(8, 5))
plt.plot(lambdas, mse_values, marker='o', linestyle='-')
plt.xscale("log")  # Log scale for lambda
plt.xlabel(r"Regularization Parameter $\lambda$")
plt.ylabel("Mean Squared Error (MSE)")
plt.title("MSE vs. Regularization Parameter $\lambda$")
plt.grid(True, which="both", linestyle="--", linewidth=0.5)

plt.show()
```

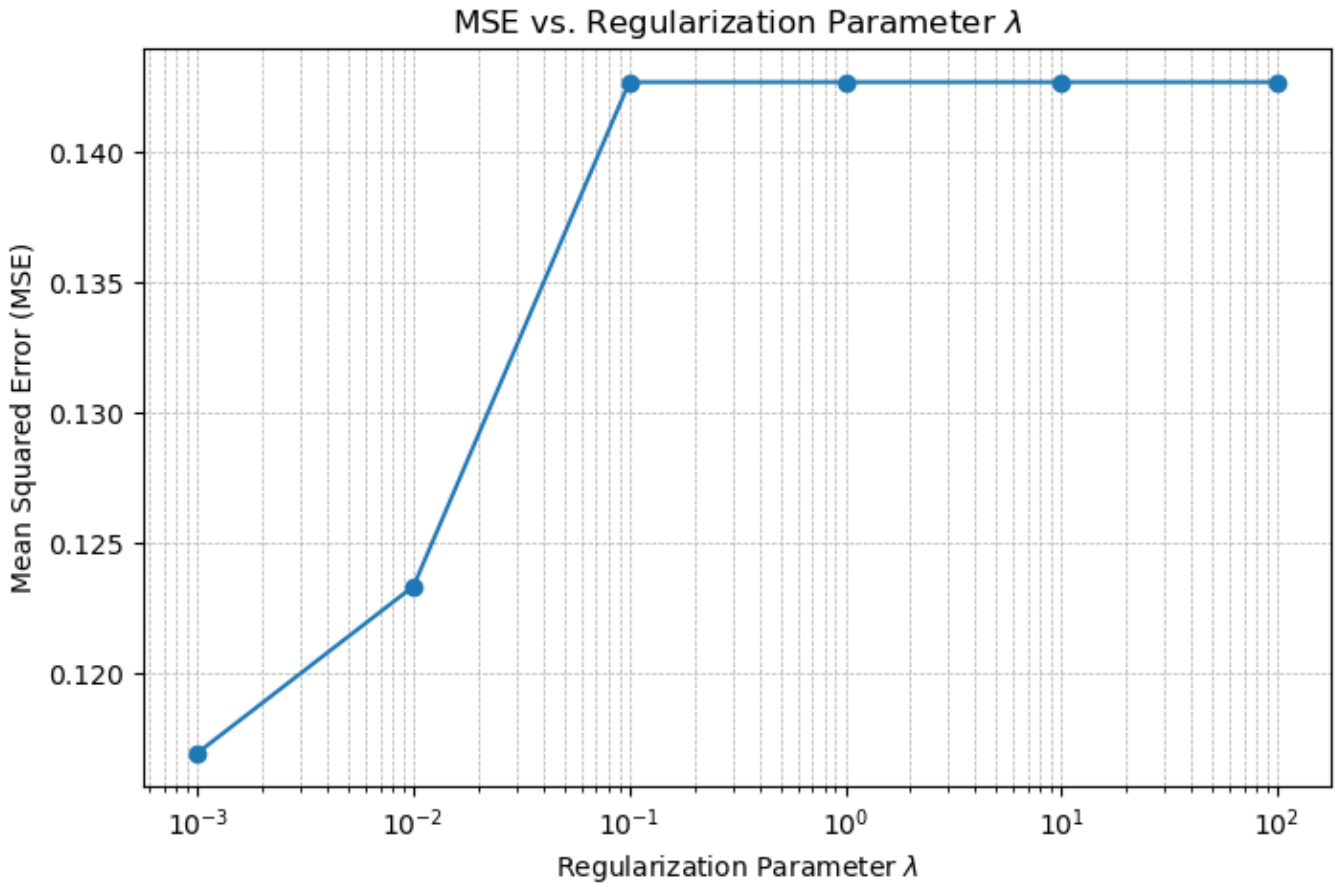

Figure 2: 3(b)Output

(c)

Input

```
# Store MSE results for all subsets
mse_matrix = np.zeros((M, len(lambdas)))
# Repeat process for M subsets
for subset in range(M):
    # Select a new random validation subset
    validation_indices = np.random.choice(sensed_indices, m, replace=False)
    training_indices = np.setdiff1d(sensed_indices, validation_indices)
    # Prepare training and validation sets
    X_train = np.array([np.unravel_index(i, (K, K)) for i in training_indices])  # Pixel
        positions
    y_train = original_block.flat[training_indices]  # Pixel values
    X_val = np.array([np.unravel_index(i, (K, K)) for i in validation_indices])
    y_val = original_block.flat[validation_indices]
    # Apply LASSO regression for each lambda
    for i, lambda_val in enumerate(lambdas):
        lasso = Lasso(alpha=lambda_val)
        lasso.fit(X_train, y_train)  # Train model
        # Predict on validation set
        y_pred = lasso.predict(X_val)
        error = mean_squared_error(y_val, y_pred)
        mse_matrix[subset, i] = error
# Compute the average MSE across M subsets
avg_mse = np.mean(mse_matrix, axis=0)
# Plot the average MSE vs. lambda
plt.figure(figsize=(8, 5))
plt.plot(lambdas, avg_mse, marker='o', linestyle='-', label="Average␣MSE")
plt.xscale("log")  # Log scale for lambda
plt.xlabel(r"Regularization␣Parameter␣$\lambda$")
plt.ylabel("Average␣Mean␣Squared␣Error␣(MSE)")
plt.title("Average␣MSE␣vs.␣Regularization␣Parameter␣$\lambda$")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.show()
```

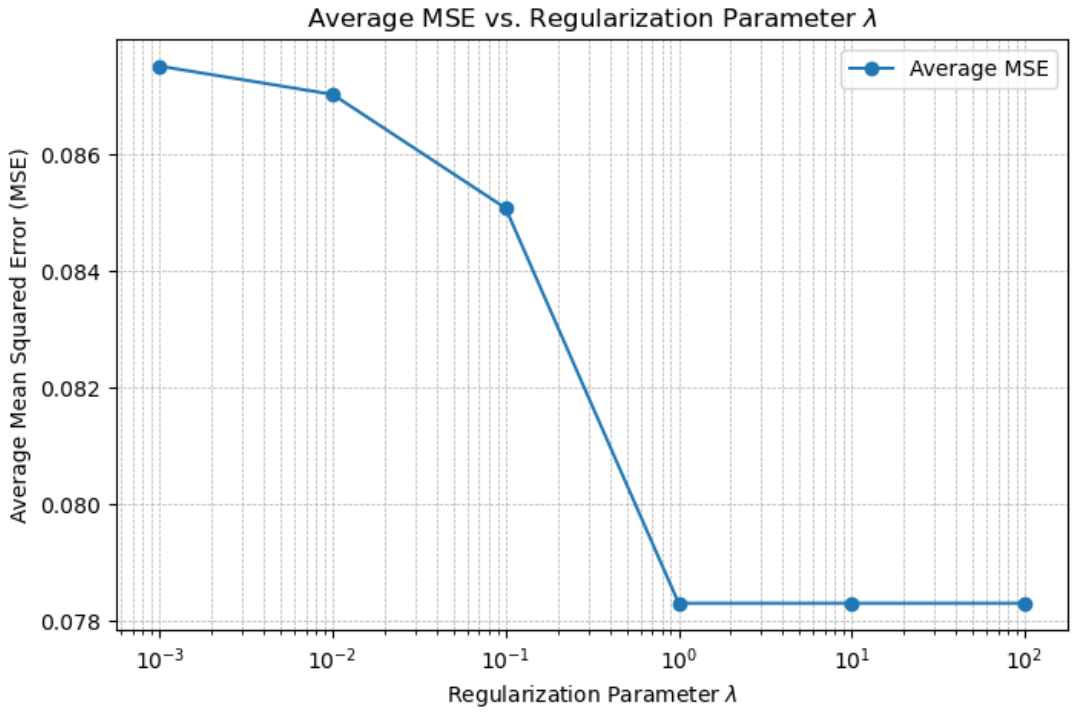

Figure 3: 3(c)Output

(c)

Input

```
# Use the best lambda from previous cross-validation
best_lambda = lambdas[np.argmin(np.mean(mse_matrix, axis=0))]

# Identify missing pixel indices (K^2 - S unknown pixels)
missing_indices = np.setdiff1d(np.arange(K*K), sensed_indices)

# Prepare training set (all sensed pixels)
X_train = np.array([np.unravel_index(i, (K, K)) for i in sensed_indices])  # Pixel positions
y_train = original_block.flat[sensed_indices]  # Pixel values

# Prepare test set (all missing pixels)
X_test = np.array([np.unravel_index(i, (K, K)) for i in missing_indices])

# Train LASSO with the best lambda
lasso = Lasso(alpha=best_lambda)
lasso.fit(X_train, y_train)

# Predict missing pixels
y_pred = lasso.predict(X_test)

# Reconstruct full image block
reconstructed_block = np.copy(sensed_pixels)
reconstructed_block.flat[missing_indices] = y_pred  # Fill in missing pixels

# Visualization
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# 1. Reconstructed image block
axes[0].imshow(reconstructed_block, cmap='gray', interpolation='nearest')
axes[0].set_title("Reconstructed Image Block")

# 2. Original (Corrupted) Image Block
axes[1].imshow(sensed_pixels, cmap='gray', interpolation='nearest')
axes[1].set_title("Original Corrupted Image Block")

# 3. Stem Plot of Model Weights
axes[2].stem(lasso.coef_)
axes[2].set_title("LASSO Model Weights")

plt.show()
```

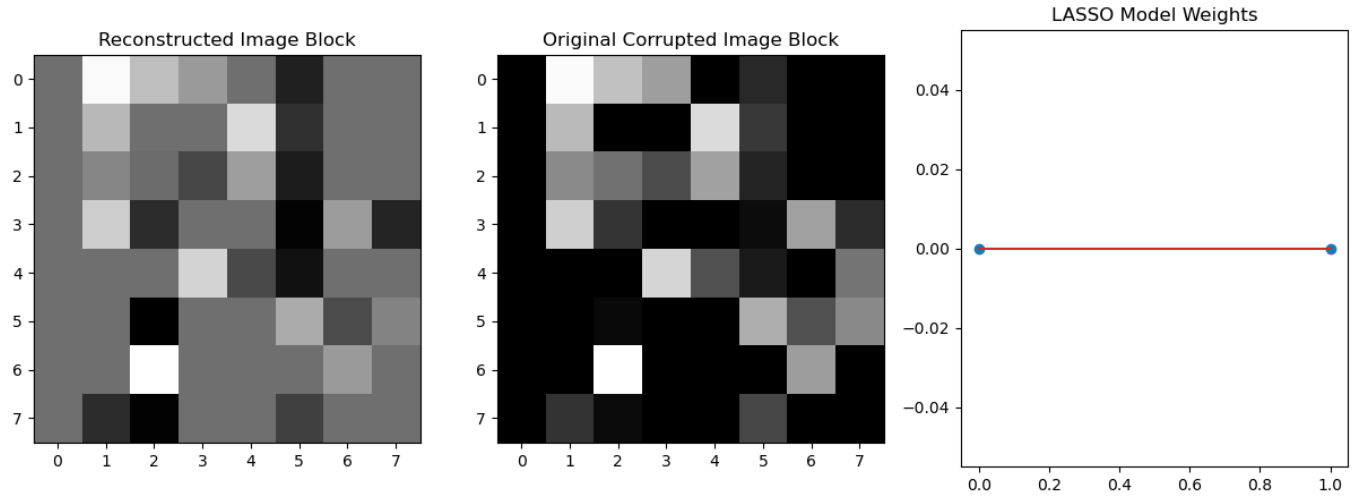

Figure 4: 3(d)Output

4 Recorver a while image

(a)

Input

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, color, util

# Load the "fishing_boat" image (convert to grayscale)
image = io.imread('fishing_boat.bmp', as_gray=True)
image = util.img_as_float(image)  # Normalize pixel values to [0,1]

# Get image dimensions
H, W = image.shape
block_size = 8  # 8x8 blocks
S = 30  # Number of sensed pixels per block

# Compute number of blocks in each dimension
num_blocks_H = H // block_size
num_blocks_W = W // block_size

# Initialize corrupted image
corrupted_image = np.zeros_like(image)

# Process each 8x8 block
for i in range(num_blocks_H):
    for j in range(num_blocks_W):
        # Extract the current 8x8 block
        block = image[i * block_size:(i + 1) * block_size, j * block_size:(j + 1) * block_size
            ]

        # Flatten block and randomly select S sensed pixels
        flat_block = block.flatten()
        sensed_indices = np.random.choice(block_size * block_size, S, replace=False)

        # Create a corrupted block (only keep sensed pixels)
        corrupted_block = np.zeros_like(flat_block)
        corrupted_block[sensed_indices] = flat_block[sensed_indices]
        corrupted_block = corrupted_block.reshape(block_size, block_size)

        # Insert the corrupted block into the full image
        corrupted_image[i * block_size:(i + 1) * block_size, j * block_size:(j + 1) *
            block_size] = corrupted_block

# Display the whole corrupted image
plt.figure(figsize=(10, 6))
plt.imshow(corrupted_image, cmap='gray')
plt.title("Whole Corrupted Image (8 8 Blocks, S=30)")
plt.axis("off")
plt.show()
```

Figure 5: 4(a)Output

(b)

Input

```
from sklearn.linear_model import Lasso
from sklearn.metrics import mean_squared_error

# Image and block parameters
H, W = image.shape
block_size = 8  # 8x8 blocks
S = 30  # Number of sensed pixels per block
m = S // 6  # Validation pixels per iteration
M = 20  # Number of cross-validation subsets
lambdas = np.logspace(-4, 3, 8)  # 8 log-spaced values for lambda

# Compute number of blocks in each dimension
num_blocks_H = H // block_size
num_blocks_W = W // block_size

# Initialize images
corrupted_image = np.zeros_like(image)
reconstructed_image = np.zeros_like(image)

# Process each 8x8 block independently
for i in range(num_blocks_H):
    for j in range(num_blocks_W):
        # Extract the current 8x8 block
        block = image[i * block_size:(i + 1) * block_size, j * block_size:(j + 1) * block_size
            ]

        # Flatten block and randomly select S sensed pixels
        flat_block = block.flatten()
        sensed_indices = np.random.choice(block_size * block_size, S, replace=False)

        # Create a corrupted block (only keep sensed pixels)
        corrupted_block = np.zeros_like(flat_block)
        corrupted_block[sensed_indices] = flat_block[sensed_indices]
        corrupted_block = corrupted_block.reshape(block_size, block_size)

        # Insert corrupted block into the full corrupted image
        corrupted_image[i * block_size:(i + 1) * block_size, j * block_size:(j + 1) *
            block_size] = corrupted_block

        # Perform Cross-Validation to find the best lambda
        mse_matrix = np.zeros((M, len(lambdas)))

        for subset in range(M):
            # Select a new random validation subset
            validation_indices = np.random.choice(sensed_indices, m, replace=False)
            training_indices = np.setdiff1d(sensed_indices, validation_indices)

            # Prepare training and validation sets
            X_train = np.array([np.unravel_index(i, (block_size, block_size)) for i in
                training_indices])
            y_train = flat_block[training_indices]
            X_val = np.array([np.unravel_index(i, (block_size, block_size)) for i in
                validation_indices])
            y_val = flat_block[validation_indices]

            # Apply LASSO regression for each lambda
            for idx, lambda_val in enumerate(lambdas):
                lasso = Lasso(alpha=lambda_val)
                lasso.fit(X_train, y_train)

                # Predict on validation set
                y_pred = lasso.predict(X_val)
                error = mean_squared_error(y_val, y_pred)
                mse_matrix[subset, idx] = error

        # Select best lambda based on lowest average MSE
        best_lambda = lambdas[np.argmin(np.mean(mse_matrix, axis=0))]

        # Identify missing pixel indices
        missing_indices = np.setdiff1d(np.arange(block_size * block_size), sensed_indices)

        # Train LASSO on all sensed pixels
        X_train_full = np.array([np.unravel_index(i, (block_size, block_size)) for i in
            sensed_indices])
        y_train_full = flat_block[sensed_indices]
```

Input

```
        lasso_final = Lasso(alpha=best_lambda)
        lasso_final.fit(X_train_full, y_train_full)

        # Predict missing pixels
        X_test = np.array([np.unravel_index(i, (block_size, block_size)) for i in
            missing_indices])
        y_pred = lasso_final.predict(X_test)

        # Reconstruct full block
        reconstructed_block = np.copy(corrupted_block.flatten())
        reconstructed_block[missing_indices] = y_pred
        reconstructed_block = reconstructed_block.reshape(block_size, block_size)

        # Insert reconstructed block into the full image
        reconstructed_image[i * block_size:(i + 1) * block_size, j * block_size:(j + 1) *
            block_size] = reconstructed_block

# Display the original, corrupted, and reconstructed images
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# 1. Original Image
axes[0].imshow(image, cmap='gray')
axes[0].set_title("Original␣Image")
axes[0].axis("off")

# 2. Corrupted Image
axes[1].imshow(corrupted_image, cmap='gray')
axes[1].set_title("Corrupted␣Image")
axes[1].axis("off")

# 3. Reconstructed Image
axes[2].imshow(reconstructed_image, cmap='gray')
axes[2].set_title("Reconstructed␣Image")
axes[2].axis("off")

plt.show()
```

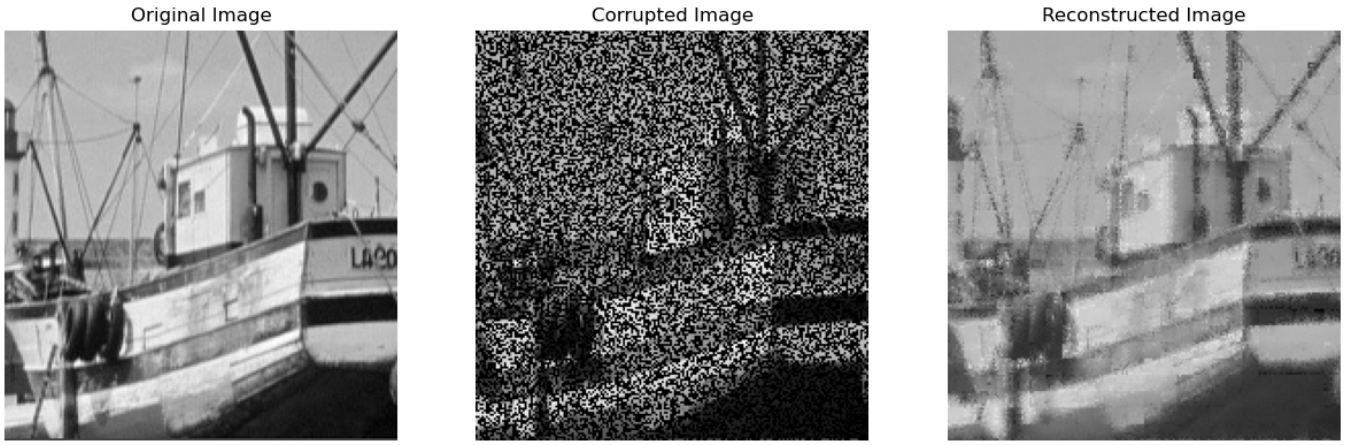

Figure 6: 4(b)Output

(c)

Input

```
from sklearn.metrics import mean_squared_error

def calculate_image_mse(original_image, reconstructed_image):
    mse = mean_squared_error(original_image.flatten(), reconstructed_image.flatten())
    return mse

# Compute MSE for the whole image
mse_whole_image = calculate_image_mse(image, reconstructed_image)
print(f"Mean Squared Error (MSE) for the whole reconstructed image: {mse_whole_image:.6f}")
```

Output

```
Mean Squared Error (MSE) for the whole reconstructed image: 0.005261
```