Dili Maduabum, Josh Bailey

Problem 1

Part 1

Consider a typical neural network setup, where each layer's output, h^l, is computed using a weight matrix W^l, a bias b^l, and an activation function f^l.

Given that the output of layer l is defined by:

$$h^l = f^l(W^l h^{l-1} + b^l)$$

where f^l is the activation function applied element-wise.

To show that:

$$\frac{\partial h^l}{\partial b^l} = \left(f^l\right)'(W^l h^{l-1} + b^l)$$

we can use the chain rule:

1. Start by noting the function composition in h^l:

   $$h^l = f^l(z^l)$$

   where $z^l = W^l h^{l-1} + b^l$.

2. The derivative of h^l with respect to b^l is:

   $$\frac{\partial h^l}{\partial b^l} = \frac{\partial f^l}{\partial z^l} \cdot \frac{\partial z^l}{\partial b^l}$$

   Applying the chain rule, we have:

   $$\frac{\partial h^l}{\partial b^l} = \left(f^l\right)' \left(W^l h^{l-1} + b^l\right) \cdot \frac{\partial (W^l h^{l-1} + b^l)}{\partial b^l}$$

3. Since z^l is linear in b^l, the derivative $\frac{\partial z^l}{\partial b^l}$ is straightforwardly 1 (keeping in mind that b^l is added element-wise to $W^l h^{l-1}$).

4. Therefore, $\frac{\partial h^l}{\partial b^l}$ simplifies to:

   $$\frac{\partial h^l}{\partial b^l} = \left(f^l\right)'(z^l)$$

   which is exactly:

$$(f^l)'(W^l h^{l-1} + b^l)$$

Part 2

Compute the gradient of the loss function L with respect to the biases b^l using the chain rule in the context of backpropagation.

1. Recognize that the gradient of the loss L with respect to the biases b^l can be propagated from the output backwards using the chain rule:

$$\frac{\partial L}{\partial b^l} = \frac{\partial L}{\partial h^l} \cdot \frac{\partial h^l}{\partial b^l}$$

2. From Part 1, we know:

$$\frac{\partial h^l}{\partial b^l} = (f^l)'(W^l h^{l-1} + b^l)$$

3. The derivative $\frac{\partial L}{\partial h^l}$ is calculated during backpropagation as the "upstream gradient" from later layers, often denoted as δ^l. It represents how the change in h^l impacts the change in loss.

4. Combining these, the formula for $\frac{\partial L}{\partial b^l}$ in terms of the gradients from later stages in the network is:

$$\frac{\partial L}{\partial b^l} = \delta^l \odot (f^l)'(W^l h^{l-1} + b^l)$$

   where $\odot$ denotes the element-wise product.

By backpropagating the gradient from the output layer back to the inputs using these steps, we ensure that the biases b^l are updated in a way that minimizes the loss L, thereby improving the model's performance with each training iteration.

Problem 2

Back ground functions

In [367…

```python
#implement the activation function
def act_func(x, type):
    """
    Compute the activation function for a given type.

    Parameters:
    - x (numpy.ndarray): Input data.
    - type (str): Type of activation function ('sigmoid' or 'ReLU').

    Returns:
```

```
    - numpy.ndarray: Output of the activation function.
    """
    if type == "sigmoid":
        # Compute the sigmoid activation function: 1 / (1 + exp(-x))
        return 1 / (1 + np.exp(-x))
    if type == "ReLU":
        # Compute the ReLU activation function: max(0, x)
        return np.maximum(0, x)


#get H's [sigma(WX + b)] and the Z's [WX + b]
def feed_forward(X, nl, act:list, parameters: dict):
    """
    Perform the feedforward pass through the neural network.

    Parameters:
    - X (numpy.ndarray): Input data.
    - nl (int): Number of layers in the neural network.
    - act (list): List of activation functions for each layer.
    - parameters (dict): Dictionary containing the parameters of the neural ne

    Returns:
    - forward: Dictionary containing the forward pass computations (ZL and HL)
    """
    p = parameters
    forward = {}
    forward["H0"] = X.T  # Input layer is the initial value of H0
    L = nl
    for l in range(1, L + 1):
        # Calculate the linear transformation Zl = Wl * Hl-1 + Bl
        forward["Z" + str(l)] = np.dot(p["W" + str(l)], forward["H" + str(l - 1

        # Apply the activation function to compute Hl
        forward["H" + str(l)] = act_func(forward["Z" + str(l)], act[l-1])
    return forward
```

Part 1a

In [368…

```
def MSE(y, y_pred, lambd: float, parameters: list):
    """
    Calculate the Mean Squared Error (MSE) loss function
    with L2 penalty (Ridge Regression).

    Parameters:
    - y (numpy.ndarray): The true target values.
    - y_pred (numpy.ndarray): The predicted values.
    - lambd (float): The regularization parameter for the L2 penalty.
    - parameters (dict): A dictionary containing the parameters of the neural n

    Returns:
    - MSE (float): The MSE loss with L2 penalty.
    """
    # Calculate L2 penalty (sum of squares of all parameters)
    # For each layer, square the parameters and then sum their sums
    # parameters.values() returns the value for each key in the dict
    L2_penalty = np.sum([np.sum(param**2) for param in parameters.values()])
    # Compute MSE with L2 penalty
    MSE = (1/2) * np.linalg.norm(y - y_pred)**2 + lambd * L2_penalty
```

```
    return MSE

def Cross_entropy(y, y_pred, lambd: float, parameters: list):
    """
    TO DO: Compute the cross entropy loss
    """
    Cross_entropy = -np.mean(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred)
    L2_penalty=np.sum([np.sum(param**2) for param in parameters.values()])
    return Cross_entropy +  L2_penalty
```

Part 1b

The MSE loss function is given by:

$$\text{MSE} = \frac{1}{2}\sum(y - \hat{y})^2$$

To find the derivative with respect to the network outputs $\hat{y}$, we have:

$$\frac{\partial \text{MSE}}{\partial \hat{y}} = \hat{y} - y$$

The derivative of x^2 is $2x$, and the $\frac{1}{2}$ factor cancels out the 2 from the derivative.

The cross-entropy loss function for two classes is given by:

$$\text{Cross-Entropy} = -\left(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right)$$

The derivative with respect to the network outputs $\hat{y}$ is:

$$\frac{\partial \text{Cross-Entropy}}{\partial \hat{y}} = -\left(\frac{y}{\hat{y}} - \frac{1 - y}{1 - \hat{y}}\right)$$

This can be simplified to:

$$\frac{\partial \text{Cross-Entropy}}{\partial \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})}$$

For a network using a softmax output layer (multiclass classification), the derivative simplifies to $\hat{y} - y$ because the softmax output assumes the entire output vector sums to 1.

In [444…

```
def MSE_grad_wrt_y_pred(y,y_pred):
    """
    Calculate the gradient of the Mean Squared Error (MSE) loss function
    with respect to the predicted value (last output of neural network)

    Parameters:
    - y (numpy.ndarray): The true target values.
    - y_pred (numpy.ndarray): The predicted values.

    Returns:
    - numpy.ndarray: The gradient of the MSE loss with respect to the predicte
    """
    mse_grad = (y_pred - y)
```

```
    return mse_grad


def CE_grad_wrt_y_pred(y,y_pred):
    """
    Calculate the gradient of the cross entropy loss with L2 penalty
    """
    cross_entrpy_grad = (y_pred - y) / (y_pred * (1 - y_pred))
    return cross_entrpy_grad
```

Part 2a

In [445…

```
def act_derivative(type, Zl):
    """
    Compute the derivative of the activation function with respect to its inpu

    Parameters:
    - activation (str): Type of activation function ('sigmoid' or 'ReLU').
    - Zl (numpy.ndarray): Input array to the activation function (pre-activati

    Returns:
    - numpy.ndarray: Derivative of the activation function evaluated at Zl.
    """
    if type == "sigmoid":
        # Compute the sigmoid of Zl
        sigmoid = 1 / (1 + np.exp(-Zl))
        # Compute the derivative of the sigmoid function
        derivative = sigmoid * (1 - sigmoid)
        return derivative

    if type == "ReLU":
        # Compute derivative: 1 when input > 0, 0 otherwise
        derivative = np.where(Zl > 0, 1, 0)
        return derivative
```

Part 3a

In [446…

```
#mine updated with doc string and comments
def backward_propagation(loss, nl, nh, Y, lambd, parameters, forward, act:list
    """
    Perform backward propagation for a neural network to compute gradients
    for all parameters.

    Args:
    nl (int): Number of layers in the neural network.
    nh (list): an int vec of length nl - that has the number of neurons in eacl
    Y (array): Observed values we try to predict
    lambd (float): Regularization parameter.
    parameters (dict): Dict containing parameters 'W' (weights) and 'B' (biases
    forward (dict): Dictionary containing the forward pass computations (ZL an
    act (list):a str vec of length nl - the activation function used in each la

    Returns:
    g (dict): A dictionary containing gradients of loss with respect to each pa
    """


    p = parameters
```

```
    f = forward
    L = nl
    g = {}

    HL = f["H"+str(L)] #Final prediction sigma(W^{L}H^{L-1} + B^{L})
    ZL = f["Z"+str(L)] #Last layer's Z (W^{L}H^{L-1} + B^{L})

    if loss == "MSE":
        # Compute gradient of loss wrt last layer Z (dL_dHL*dHL_dZL)
        g["dLoss_dZ"+str(L)] = MSE_grad_wrt_y_pred(Y, HL) * act_derivative(act
    if loss == "CE":
         g["dLoss_dZ"+str(L)] = CE_grad_wrt_y_pred(Y, HL) * act_derivative(act
    # Deriative of last layer Z wrt its weights & biases (dZL_dWL, dZL_dBL)
    g["dZ"+str(L)+"_dW"+str(L)] = f["H"+str(L-1)] #just H^{L-1}
    g["dZ"+str(L)+"_dB"+str(L)] = np.ones((nh[-1], 1))#vec of 1's row's of las

    # Calculate derivative with respect to weights and biases for the last laye
    # dLoss_dWL = dL_dHL*dHL_dZL*dZL_dWL, dLoss_dBL = dL_dHL*dHL_dZL*dZL_dBL
    g["dLoss_dW"+str(L)] = np.dot(g["dLoss_dZ"+str(L)],
                                  g["dZ"+str(L)+"_dW"+str(L)].T) \
                           + lambd * p["W"+str(L)]  # Include regularization t
    g["dLoss_dB"+str(L)] = np.dot(g["dLoss_dZ"+str(L)].T,
                                  g["dZ"+str(L)+"_dB"+str(L)]) \
                           + lambd * p["B"+str(L)]  # Include regularization t

    #from L-1 to first layer, which is 1
    for l in reversed(range(1, L)):
        # Calculate gradient of Z in layer l+1 wrt Z in layer l
        # dZl+1_dZl = dZl+1_dHl*dHl_dZl
        g["dZ_"+str(l+1)+"_dZ"+str(l)] = np.dot(p["W"+str(l+1)].T,
                                                g["dLoss_dZ"+str(l+1)])

        # Propagate the loss gradient back from layer l+1 to layer l
        #dLoss_dZl = dLoss_dZl+1*dZl+1_dHl*dHl_dZl
        g["dLoss_dZ"+str(l)] = g["dZ_"+str(l+1)+"_dZ"+str(l)] * \
                               act_derivative(act[l], f["Z"+str(l)])

        # Deriative of Z wrt its weights & biases (for each layer)
        g["dZ"+str(l)+"_dW"+str(l)] = f["H" + str(l-1)]
        g["dZ"+str(l)+"_dB"+str(l)] = np.ones((nh[l], 1))

        # Calculate derivatives with respect to weights and biases in layer l
        g["dLoss_dW"+str(l)] = np.dot(g["dLoss_dZ"+str(l)],
                                      g["dZ"+str(l)+ "_dW"+str(l)].T) \
                               + lambd * p["W" + str(l)]
        g["dLoss_dB"+str(l)] = np.dot(g["dLoss_dZ"+str(l)].T,
                                      g["dZ"+str(l)+ "_dB"+str(l)]) \
                               + lambd * p["B" + str(l)]




    return g
```

Part 3b

loading the neccesities from pset 8

In [447…
```
nl = 5
X = np.array([0.1, -0.2, 0.3, -0.4, 0.5]).reshape(1, -1)
nh = [X.shape[1], 5, 4, 3, 5, 1]
act = ["ReLU", "sigmoid", "ReLU", "sigmoid", "ReLU"]
Y = 3

def parameterss(nl):
    """
    Parameters:
    - nl (int): Number of layers in the neural network.

    Returns:
    - parameters (dict): Dictionary containing all the weights and biases
    """
    parameters = {}
    for l in range(1, nl + 1): #1 to 5
        # Select the values all but the last one (which is the bias)
        parameters["W" + str(l)] = pd.read_csv(f'data/layer{l}.csv').values[:,
        # Select the values the last one only (the bias), and make it a column
        parameters["B" + str(l)] = pd.read_csv(f'data/layer{l}.csv').values[:,
    return parameters
```

computing the back prop

In [448…
```
parameters = parameterss(nl)
forward = feed_forward(X, nl, act, parameters)
lambd = 1 #random since now was given

grads_mse = backward_propagation("MSE", nl, nh, Y, lambd, parameters, forward,

print("Gradient (MSE) of the weights in respect to Weights of the first hidden
display(pd.DataFrame(grads_mse['dLoss_dW1']))
print("Gradient (MSE)of the bias in respect to Bias of the first hidden layer:'
display(pd.DataFrame(grads_mse['dLoss_dB1']))

grads_ce = backward_propagation("CE", nl, nh, Y, lambd, parameters, forward, a

print("Gradient (Cross Entropy) of the weights in respect to Weights of the fi
display(pd.DataFrame(grads_ce['dLoss_dW1']))
print("Gradient (Cross Entropy) of the weights in respect to Bias of the first
display(pd.DataFrame(grads_ce['dLoss_dB1']))
```

```
Gradient (MSE) of the weights in respect to Weights of the first hidden layer:
```

	0	1	2	3	4
0	-0.062636	-0.082066	0.151207	-0.004532	0.091946
1	0.018354	0.048763	0.038955	-0.001580	0.078164
2	-0.083558	0.073823	-0.062109	0.094364	0.007481
3	0.159528	0.057579	-0.221471	0.082124	-0.198937
4	0.032952	-0.030541	0.112497	0.059385	0.061989

```
Gradient (MSE)of the bias in respect to Bias of the first hidden layer:
```

	0
0	-0.005556
1	-0.015523
2	-0.147018
3	-0.047758
4	0.041851

Gradient (Cross Entropy) of the weights in respect to Weights of the first hid
den layer:

	0	1	2	3	4
0	-0.062599	-0.082139	0.151316	-0.004677	0.092128
1	0.018318	0.048836	0.038844	-0.001432	0.077980
2	-0.083539	0.073786	-0.062054	0.094290	0.007573
3	0.159526	0.057582	-0.221475	0.082129	-0.198944
4	0.032957	-0.030551	0.112512	0.059365	0.062013

Gradient (Cross Entropy) of the weights in respect to Bias of the first hidden
layer:

	0
0	-0.005342
1	-0.015309
2	-0.146804
3	-0.047544
4	0.042065

Appendix, Finding the dimensions of each

In [449…

```
for i in reversed(range(1, nl  + 1)):
    print("shape of dLoss_dZ" +str(i) + ":", grads_mse["dLoss_dZ"+str(i)].shape
    print("shape of dLoss_dW" +str(i) + ":", grads_mse["dLoss_dW"+str(i)].shape
    print("shape of dLoss_dB" +str(i) + ":", grads_mse["dLoss_dB"+str(i)].shape
```

```
shape of dLoss_dZ5: (1, 1)
shape of dLoss_dW5: (1, 5)
shape of dLoss_dB5: (1, 1)

shape of dLoss_dZ4: (5, 1)
shape of dLoss_dW4: (5, 3)
shape of dLoss_dB4: (5, 1)

shape of dLoss_dZ3: (3, 1)
shape of dLoss_dW3: (3, 4)
shape of dLoss_dB3: (3, 1)

shape of dLoss_dZ2: (4, 1)
shape of dLoss_dW2: (4, 5)
shape of dLoss_dB2: (4, 1)

shape of dLoss_dZ1: (5, 1)
shape of dLoss_dW1: (5, 5)
shape of dLoss_dB1: (5, 1)
```

Part 4a

Early stopping helps avoid overfitting when training a machine learning model, particularly with iterative methods like stochastic gradient descent. The concept involves monitoring the model's performance on a validation set during training and stopping the training process if the performance on the validation set ceases to improve or begins to degrade, despite continued improvements on the training set.

Early stopping effectively adds a regularization effect without the need for explicit regularization techniques like L^1 or L^2 penalties. It prevents the model from learning noise and complex patterns in the training data that do not generalize to new data, which is essentially what overfitting is. It can also reduce computational costs by stopping training before all allocated resources (like epochs) are used, especially if no further learning improvement is detected.

While helpful, calling it a "free lunch" should be taken with caution. It doesn't avoid the need to deal with needing to tune of the patience parameter and set up the validation process to avoid premature stopping or too late stopping, which can still lead to underfitting or overfitting, respectively. Moreover, it relies heavily on having a well-separated validation set that accurately represents unseen data, which isn't always available or easy to construct, especially in cases with limited data.

Part 4b

New parameter funcction

In [ ]:
```
import numpy as np
from sklearn.model_selection import train_test_split
```

```
data = pd.read_csv('data/ps9.csv')
Y = data.iloc[:, 0].values.reshape(-1, 1)  # Assuming the first column is the
X = data.iloc[:, 1:].values  # Remaining columns are features

# Define neural network structure
nl = 5  # Number of layers
nh = [10, 20, 15, 10, 5]  # Example: Number of neurons in each layer
act = ["ReLU", "sigmoid", "ReLU", "sigmoid", "ReLU"]  # Activation functions

parameters = parameter(nl, nh, X) # Initialize parameters


# Set training parameters
batch_size = 100
patience = 10
lambd = 1  # Regularization strength
learning_rate = 0.01  # Learning rate for SGD updates


def stoch_grad_des_w_early_stop(loss, nh, X, Y, parameters, act, batch_size, pa
    """
    Implements stochastic gradient descent with early stopping, adjusting the

    Args:
        X (numpy.ndarray): Input feature matrix.
        Y (numpy.ndarray): Output target vector.
        parameters (dict): Initial neural network parameters.
        feed_forward (function): Function to perform the forward pass.
        backward_propagation (function): Function to perform the backward pass
        act (list): List of activation functions for each layer.
        batch_size (int): Size of each batch for gradient descent updates.
        patience (int): Number of epochs to continue without improvement before
        lambd (float): Regularization strength.
        initial_k (int): Initial value for the learning rate denominator.

    Returns:
        dict: Dictionary containing the best parameters and the corresponding v
    """
    # Split data into training, validation, and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25,
    X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_s


    best_params = parameters.copy()
    best_val_error = float('inf') #infinity, the highest value
    no_improvement = 0
    k = initial_k  # Initialize k for adaptive learning rate

    n_train = len(y_train)  # Number of rows in training data
    indices = np.arange(n_train)

    while no_improvement < patience:
        np.random.shuffle(indices)
        for start in range(0, n_train, batch_size):
            end = start + batch_size
            batch_indices = indices[start:end]
            batch_X, batch_Y = X_train[batch_indices], y_train[batch_indices]

            # Perform a forward pass
            forward = feed_forward(batch_X, len(act), act, parameters)
```

```
            print(forward["H5"].shape)
            print(batch_X.shape)
            # Compute gradients
            grads = backward_propagation(loss, len(act), nh, batch_Y, lambd, pa

            # Update parameters using a dynamic step size 1/k
            step_size = 1 / k
            for l in range(1, len(parameters) // 2 + 1):
                parameters["W" + str(l)] -= step_size * grads["dW" + str(l)]
                parameters["B" + str(l)] -= step_size * grads["dB" + str(l)]

        # After each epoch, evaluate on the validation set
        val_forward = feed_forward(X_val, len(act), act, parameters)
        val_error = np.linalg.norm(y_val - val_forward["H" + str(len(act))].T)

        # Update the best parameters if the current model is better
        if val_error < best_val_error:
            best_val_error = val_error
            best_params = {k: v.copy() for k, v in parameters.items()}
            no_improvement = 0
        else:
            no_improvement += 1

        # Increment k after each full pass through the data
        k += 1

    return {'best_params': best_params, 'best_val_error': best_val_error}

stoch_grad_des_w_early_stop("MSE", nh, X, Y, parameters, act, batch_size, patie
```

Part 5a

When initializing neural networks, the choice of starting values for the weights and biases can significantly impact the performance of the model, particularly in terms of convergence speed and the likelihood of reaching a good local minimum. I am choosing random values as my initial weights and bias

In [405…

```python
def parameter(nl, nh, X):
    nh.insert(0, X.shape[1]) # add the row of the X's to the first one
    nh[-1] = X.shape[0] #changing the "1" to be the layer of neur
    """
    Initialize the parameters (weights and biases) of a neural network.

    Parameters:
    - nl (int): Number of layers in the neural network.
    - nh (list): List containing the number of neurons in each layer.

    Returns:
    - dict: Dictionary containing the initialized (random parameters).
    """
    parameters = {}
    for n in range(1, nl + 1):
        # Initialize weights randomly from a uniform distribution
        parameters["W" + str(n)] = np.random.rand(nh[n], nh[n-1])
        # Initialize biases as zeros
        parameters["B" + str(n)] = np.zeros((nh[n], 1))
    return parameters
```

Part 6

Normalization is highly recommended, primarily because it helps in speeding up the training process and achieving better convergence. Here are the main reasons for normalizing data when training neural networks. Neural networks often perform better when input features are on a similar scale. This uniformity prevents certain features from disproportionately influencing the model's learning, which is particularly important for weights to converge more quickly during gradient descent. Moreover, when data are not normalized, the gradients during training can become very small (vanish) or very large (explode), especially in deep networks. This can lead to unstable training processes, where the network either learns too slowly or diverges. Relatedly, normalization ensures that the error surface is more spherical. Without normalization, the surface can be elongated, meaning gradient descent takes longer paths towards minima, potentially oscillating inefficiently down steep slopes.

The decision on whether to penalize weights, biases, or both in a neural network generally hinges on the goals of regularization and the model:

1. **Penalizing Weights**:
    - Regularization techniques like L^2 (Ridge) and L^1 (Lasso) are typically applied to weights. Penalizing weights helps in controlling the model's complexity by keeping the weights small, which encourages the model to find simpler patterns that may generalize better on unseen data.
    - By keeping weights small, it reduces the risk of the model fitting too closely to the noise in the training data.
2. **Penalizing Biases**:
    - It is generally less common to penalize biases because they do not control the complexity of the model in the same way that weights do. Biases are meant to

provide flexibility to the model by allowing shifts to the decision boundary.
- Penalizing biases can lead to underfitting since it restricts the model's ability to fit even the correct general trend of the data.

It's usually best to penalize just the weights because this directly addresses the model's complexity without unduly limiting its capacity to fit the data appropriately. Since biases help in fitting the model more flexibly to the data without significantly increasing the complexity, they are typically left unpenalized to maintain the model's adaptive performance across varying data dynamics.

Part 7

In [ ]:
```
patience = 5
lambd = [1, 10, 100, 1000, 10000]
loss = ["MSE", "CE"]

for lam in lambd:
    for los in loss:
        stoch_grad_des_w_early_stop(los, nh, X, Y, parameters,
                                    feed_forward, backward_propagation,
                                    act, batch_size, patience, lam)
```

Part 7(d)

Adding 50% more data to an existing dataset and retraining the model inherently changes the dynamics of how the model interacts with the data. If you randomly split the combined dataset into training, validation, and test sets, the key to ensuring that the test error estimate is unbiased relies on the independence of the test set. If the test data contains examples that are not representative of the overall dataset or are similar to the training set, the test error might not accurately reflect how the model will perform on truly unseen data. Moreover, there's a risk of data leakage if the test set includes data that's too similar to or influenced by the training data. This could happen if the new data shares specific characteristics or patterns with the old data that the model has already learned.

To ensure the test error estimate remains unbiased, we could consider splitting the data based on time (if it's time-series data) or some logical separation that ensures the test set is likely to encounter scenarios the model hasn't explicitly trained on. This helps mimic real-world application where future data or different scenarios are encountered. If the dataset has classes or categories, use stratified sampling to ensure that the training, validation, and test sets each contain a representative mix of all classes. This prevents class imbalance in any of the sets, which could bias the model's performance and thus the error estimate. From the combined dataset, allocate a portion as a hold-out test set before training begins. This set should not be used in any way during the model training or tuning processes, including not influencing the selection of λ. It serves as a final evaluator to test the model's performance. Finally, we would want to consider using k-fold cross-validation, especially if data is limited, to ensure that every data point gets used for both training and testing. This

helps in understanding the model's stability and robustness across different subsets of data.

In [ ]: