Part A Implementation of Ridge Regression Using Gradient Descent

lossFunction()

This function computes the empirical risk (loss) and gradient for ridge regression. It takes as paramaters, the weights (theta), a feature matrix (X), label vector (y), and regularization paramater (Lambda). The function calculates the prediction error (resid) by subtracting the observed values from the predictions made using the current coefficients. The Ridge Regression loss is computed by adding the L2 norm (squared sum) of the coefficients scaled by the regularization parameter, Lambda, to the mean squared error. The gradient of the loss with respect to the coefficients is calculated from the empirical risk:

$$\begin{aligned}\hat{R}_{\text{ridge}}(\mathbf{w}) &= \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \mathbf{w}^\top \mathbf{x}_i\right)^2 + \lambda \mathbf{w}^\top \mathbf{w} \\ &= \frac{1}{n}(y - \mathbf{w}^\top X)^2 + \lambda \mathbf{w}^\top \mathbf{w} \\ \nabla_{\mathbf{w}} \hat{R}_{\text{ridge}}(\mathbf{w}) &= \frac{1}{n}\nabla_{\mathbf{w}}[(y - \mathbf{w}^\top X)^2] + \lambda \nabla_{\mathbf{w}} \mathbf{w}^\top \mathbf{w} \\ &= \frac{1}{n} - 2X(y - \mathbf{w}^\top X) + 2\lambda \mathbf{w} \\ &= -\frac{2}{n} X(y - \hat{y}) + 2\lambda \mathbf{w}\end{aligned}$$

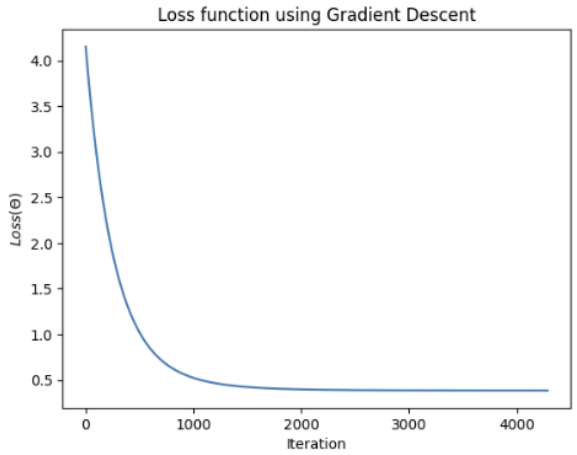
This is reflected by the folowing lines in lossFunction():

```
    regLoss = (1/n) * (np.sum(resid ** 2) + Lambda * np.sum(theta ** 2))
    grad = (-2/n) * (X.T @ resid) + 2 * Lambda * theta
```

gradientDescent()

This function optimizes theta by iteratively adjusting it in the direction of the steepest decrease in loss. It takes a feature matrix (X), label vector (y), and the initial weights (theta), regularization paramater (Lambda), a learning rate (eta), and a defined threshold indicatative of convergence (tolerance). A maximum of 5000 iterations is set to avoid an infinite loop.

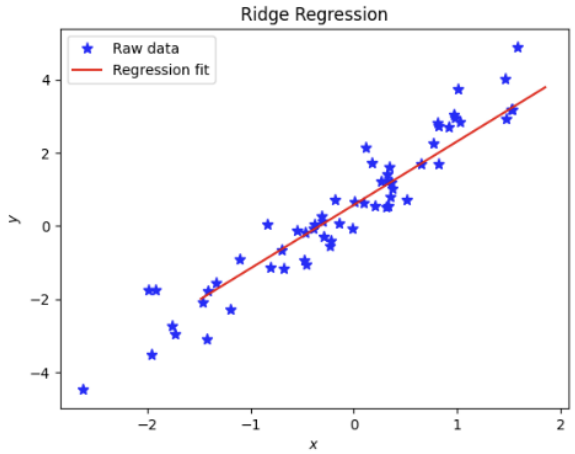
The function repeatedly calls lossFunction to compute the current loss and gradient. It updates theta by taking steps proportional to the negative of the gradient, scaled by eta. The process continues for a fixed number of iterations or until the change in theta is less than the specified tolerance, indicating convergence. It stores the loss of each iteration in Loss_history for graphing purposes. The graph representing the decrese in loss as the weights are updated is as follows:




```
def gradientDescent(X,y,theta,eta,Lambda,tolerance):
    Loss_history =[]
    n = len(y)
    for i in range(5000):
      loss, grad = lossFunction(theta, X, y, Lambda)
      Loss_history.append(loss)
      theta_new = theta - eta * grad
      if np.linalg.norm(theta_new - theta, ord=1) < tolerance:
          break
      theta = theta_new
    return theta, Loss_history
```

Test module output

The output from the "Test Module" section is included below. It shows the data generated by the generate_polynomial_data() and the regression fit line derived from the weights optimized by the gradient descent.

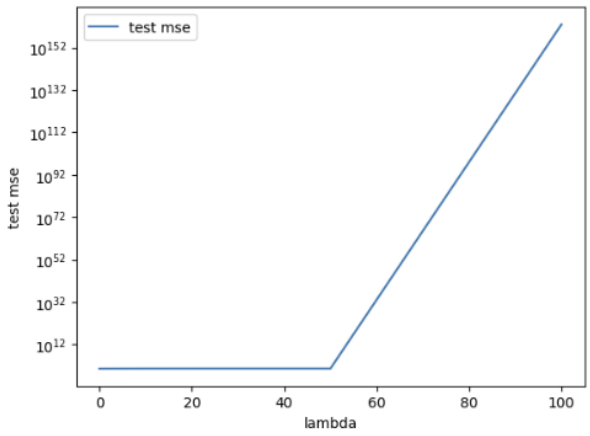


Part B Model Analysis and Optimization

train_and_eval()

This function trains a ridge regression model using gradient descent and evaluates its performance by computing the Mean Squared Error (MSE) on an evaluation (holdout) and outputs the MSE of the model on the evaluation set. It first standardizes the input data if it's in a pandas DataFrame or Series. Then, it initializes the model coefficients (theta) and calls gradientDescent() to optimize these coefficients. After training, it uses the optimized coefficients to make predictions on the evaluation set and computes the MSE by comparing these predictions to the actual response values.

This function is then used to illustrate the effect of lamda on a model's MSE. The plot below demonstrates how too much regularization can lead to underfitting, and extremely high mse.




```
def train_and_eval( X_train , y_train , X_eval , y_eval , lambda_ ):
  #...
  theta, loss_history = gradientDescent(X_train, y_train.reshape(-1, 1), theta, eta, lambda_, tolerance)
  predictions = X_eval@theta

  mse = ((predictions - y_eval) ** 2).mean()
  return mse
```

cross_validation()

Performs k-fold cross-validation to estimate the performance of the ridge regression model for a given value of λ, helping in the selection of the best regularization parameter. The function parameters are a training feature matrix (X_train), training response vector (y_train), and regularization parameter (lambda_), and it outputs the average MSE across all k hold-out sets.

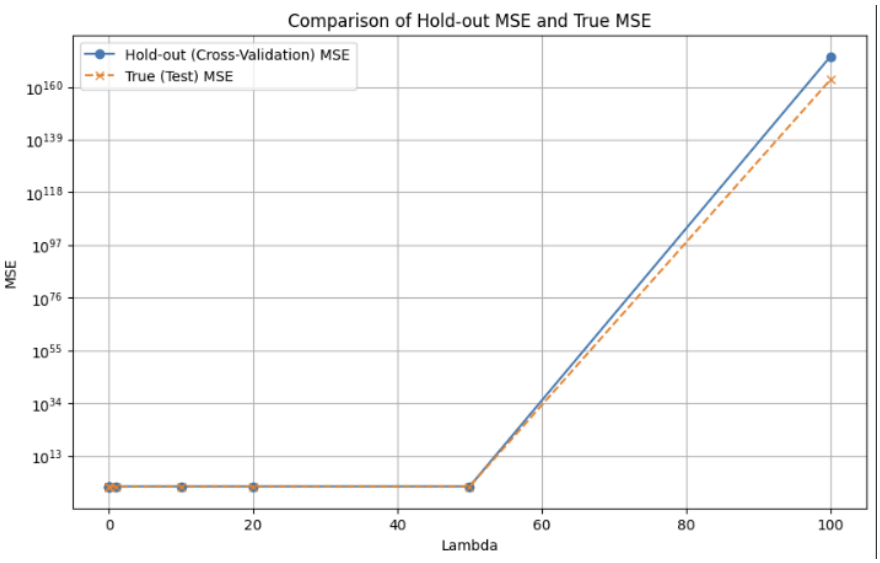
The function splits the training data into k folds. For each fold, it designates the fold as the hold-out set and the rest as the training set, then calls train_and_eval to train the model and compute the MSE on the hold-out set. Finally, it averages these MSE values to estimate the model's performance.

```
def cross_validation(X_train, y_train, lambda_, k=10):
    X_folds = np.array_split(X_train, k)
    y_folds = np.array_split(y_train, k)

    mse_list = []
    for i in range(k):
      X_holdout, y_holdout = np.vstack(X_folds[i]),np.hstack( y_folds[i])
      if i == 0:
        X_train_cv =np.vstack(X_folds[i + 1:])
        y_train_cv =  np.hstack(y_folds[i + 1:])
      else:
        X_train_cv = np.vstack(X_folds[:i] + X_folds[i + 1:])
        y_train_cv = np.hstack(y_folds[:i] + y_folds[i + 1:])
      mse = train_and_eval( X_train_cv , y_train_cv , X_holdout , y_holdout , lambda_ )
      mse_list.append(mse)
    avg_mse = np.mean(mse_list)
    return avg_mse
```

Model Selection

The code iterates over a predefined list of λ values, applying cross_validation to find the average MSE for each λ and train_and_eval to compute the true MSE on the set exculded from CV. The lowest average MSE from cross-validation is used to select the best λ, while the true MSE gives an unbiased estimate of the model's performance on unseen data. The Lamda with the lowest average MSE on CV holdout sets was determined to be 1. The plot comparing MSE for each lamda is included below



Bonus: LASSO Regresson

lossFunctionLASSO()

The function calculates the loss as the sum of the squared prediction errors and the L1 penalty term (the sum of the absolute values of the coefficients). The sub-gradient of the L1 term is computed as the sign of the coefficients, which is then used to update the coefficients during gradient descent.

Lasso regression risk function is defined as

$$\hat{R}_{\text{lasso}}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \mathbf{w}^\top \mathbf{x}_i\right)^2 + \lambda||\mathbf{w}||_1$$

Its gradient is the same as Ridge regression, with the exception of the regularization term.

$$\lambda||\mathbf{w}||_1 = \lambda\sum_{i=1}^{d}|w_i|$$

Due to the absolute value function, the gradient is not defined for $w_i = 0$

$$\nabla_{\mathbf{w}}(\lambda||\mathbf{w}||_1) = \lambda \begin{bmatrix} \text{sign}(w_1) \\ \text{sign}(w_2) \\ \vdots \\ \text{sign}(w_d) \end{bmatrix}$$

Where $\text{sign}(w_i)$ is the sign function, which is defined as:

- if $w_i > 0$ then $\text{sign}(w_i) = 1$
- if $w_i < 0$ then $\text{sign}(w_i) = -1$
- if $w_i = 0$ then $\text{sign}(w_i)$ is any value in $[-1, 1]$ (this is the subdifferential set for the non-differentiable point).

However, assigning all w_i = 0 t0 the set [-1,1] is impractical for computational purposes, so within the code, the choice was made to assign these values to 0. This choice is reflected in the line:

```
grad = (2/n) * X.T @ resid + Lambda * np.sign(theta)
```

Apart from defining the loss function and its gradient, the implementation is functionally the same as ridge regression.