AMS 595

Group Project

Regression

(Linear, Ridge, LASSO)

By: Vivianne Huang, Yeji Kim, Justin Zhong

Our project revolves around regression and the impact of regularization techniques, particularly Ridge and LASSO regression and their effects. For our project, we took the Boston Housing dataset found in Python through the “mglearn” package and split the data into training and test sets. We performed linear, Ridge, and LASSO regression and compared our results, mainly by looking at the mean squared error (MSE) of our test set. For the project, we implemented the splitting of the data, Ridge regression, and LASSO regression from scratch to show our understanding of the models and Python.

To divide the workload, each of us took on a different task. Yeji worked on creating functions for split training and test datasets and mean squared error. Vivianne and Justin worked on implementing Ridge and LASSO regression, respectively. We came together to check up on each other and help each other out when we had difficulties with the code. For our slides and report, we focused mainly on our own sections when writing and presenting.

Before building a linear regression model, we have developed a function called 'train_test_split' to assess the model's effectiveness in predicting unseen data. This function divides our dataset into training and testing subsets, using four parameters. The first parameter is 'X,' representing a matrix of input features where each row in 'X' corresponds to a single sample from the dataset. The second parameter is 'y,' a target vector containing the output values our model aims to predict. The third parameter, 'test_size,' determines the size of the test dataset. If 'test_size' is a proportion, the number of test data is calculated as a product of 'test_size' and the total number of samples in the dataset. If it is an integer, it implies the exact number of samples. This approach provides flexibility in defining the portion of data used for testing versus training. The last parameter, 'random_state,' is an optional integer seed used for random number generation, similar to 'random.seed()'. This parameter is crucial as it ensures that the data splits are reproducible and consistent across different runs, which is essential for verifying and comparing our model's performance.

In this function, the initial step is to generate a sequence of indices corresponding to our dataset's samples using 'np.arange(n_samples),' where 'n_samples' is determined by the length of 'X,' our feature matrix. This creates an array of indices ranging from 0 to 'n_samples - 1,' with each index representing a unique sample in the dataset. After generating these indices, we use 'np.random.shuffle(indices)' to randomly shuffle them. This shuffling is critical to ensure that the selection of samples for the training and testing sets is unbiased, avoiding potential order-based biases that might exist in the original dataset. After shuffling the indices, we determine the test set's size. This is done by checking if 'test_size' is a float, in which case it's interpreted as a proportion of the dataset. We then calculate the number of test samples by multiplying 'test_size' by 'n_samples,' rounding the result to the nearest integer. Next, we use these calculated indices to divide the dataset into training and testing subsets. We extract the test indices by selecting the first 'test_size' elements from the shuffled indices array, while the remaining elements are used as

training indices. This method ensures a clear separation of the dataset into distinct sets for training and testing purposes. With these indices, we proceed to partition the data. 'X_train' and 'X_test' are formed by indexing the input feature matrix 'X' with 'train_indices' and 'test_indices,' respectively. Similarly, 'y_train' and 'y_test' are generated by applying the same indexing to the target vector 'y.' This results in corresponding training and testing sets for both the features and the target values. By returning these four datasets (X_train, X_test, y_train, y_test), our function completes its task, providing separated data subsets ready for model training and subsequent evaluation of its performance on unseen data.

Additionally, we have also implemented two functions for performing linear regression analysis. These two functions are from chatgpt. The 'fit_linear_regression(X, y)' function fits a linear regression model to our dataset. The first parameter, 'X,' represents a matrix of input features where each row in 'X' corresponds to a single sample from our dataset. The 'y' parameter is a target vector containing the output values that our model aims to predict. In this function, we begin by adding a column of ones to the input features to account for an intercept term in linear regression. We then calculate the coefficients of the linear regression model using the normal equation. These coefficients represent the slope and intercept of the linear regression model.

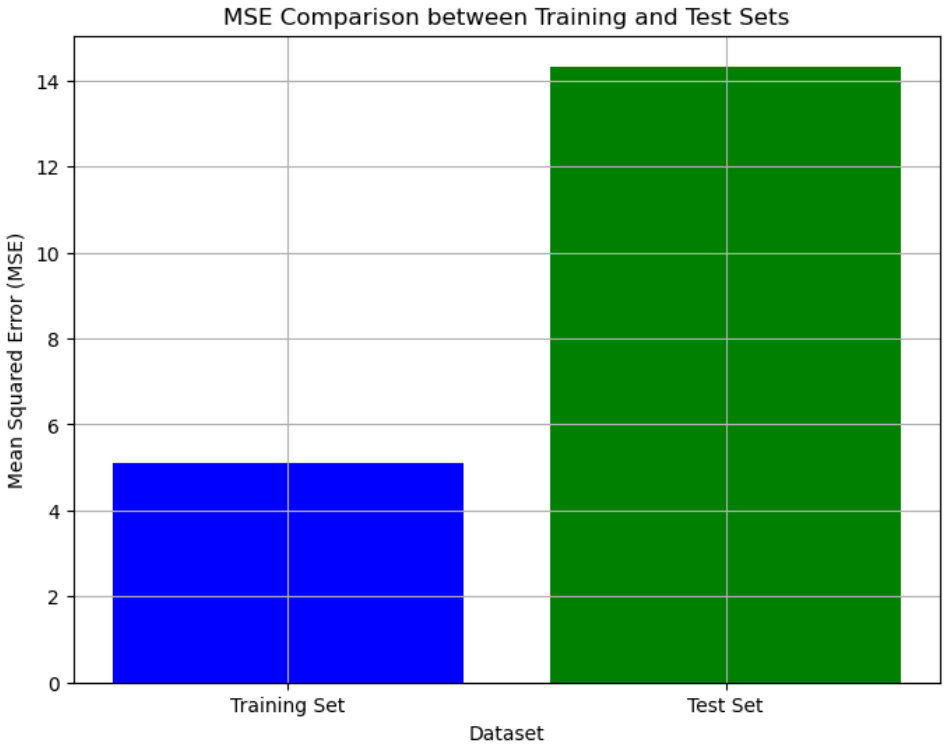
The 'predict(X, coefficients)' function is designed to make predictions using a previously fitted linear regression model. Similar to the 'fit_linear_regression' function, the 'X' parameter represents a matrix of input features, while the 'coefficients' parameter represents the slope and intercept of the linear regression model, obtained from the 'fit_linear_regression' function. In the prediction process, a column of ones is added to the input feature matrix 'X' to account for the intercept term. Using the dot product of the extended feature matrix and the previously calculated coefficients, the function calculates the predicted target values. These predictions represent the model's estimates for the target variable and are returned as the function's output.

MSE is a fundamental indicator of how well the model's predictions correspond to the true data values. It is evaluated by creating a linear regression model. The MSE quantifies this correspondence by calculating the average of the squared differences between the predicted values and the actual target values. We've created a function called mean_squared_error that calculates the Mean Squared Error (MSE) between two arrays of values: y_true (representing the true or actual values) and y_pred (representing the predicted values). This function begins by checking whether the lengths of y_true and y_pred are the same. If their lengths do not match, it raises a ValueError to indicate the requirement for both arrays to have the same length.

Before computing the MSE, we initialize a list named 'squared_diff' to store the squared differences between corresponding elements of y_true and y_pred. We calculate the squared difference (y_true[i] - y_pred[i]) ** 2 for each pair of corresponding elements by iterating through both arrays using a for loop. These squared differences are then appended to the

'squared_diff' list. Next, the function calculates the MSE by summing up all the squared differences in the 'squared_diff' list and dividing this sum by the total number of elements in the arrays. The number of elements is determined using len(y_true).

We perform a linear regression analysis using the extended Boston Housing dataset. First, we split the data into training and test sets and fit a linear regression model to the training data (X_train and y_train). Then, we calculate predicted values on the test data using this trained model. Subsequently, we compute the Mean Squared Error (MSE) on both the training and test sets to evaluate the model's performance.



This graph shows the MSE on the training set and test set. The MSE on the training set was around 5.12 and 14.33 for the test set. In this scenario, overfitting has occurred where there is low training MSE but high test MSE. This shows that the model is great with training data but struggles with new data. To address this problem of overfitting, we decided to implement Ridge and LASSO regression because of their regularization.

Ridge regression, also known as Tikhonov or L2 regularization, is a regularization technique typically used to address a model that is overfit. This is possible because it involves a penalty term that is added to the cost function to lower the mean squared error. That penalty term involves a regularization parameter, in our case α, where if $\alpha = 0$, we are just conducting the least squares linear regression. This added term prevents the model parameters from becoming very

large as it measures the magnitude of the coefficients. Ridge regression involves a tradeoff as it results in higher bias but lower variance.

For the project, we attempted two ways of implementing ridge regression, the closed-form approach and the gradient descent approach. The cost function in this case is: $J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} \theta_i^2$ which can be minimized with $\widehat{\theta} = (X^T X + \alpha I)^{-1} X^T y$. In this case, our closed-form approach allowed us to solve for θ as the vector of coefficients with the zero index being the bias and the rest being the weights. For the gradient descent approach, we took the partial derivatives with respect to the weights and bias and looped it to get the minimum values. Unfortunately, for this code, there were some issues with getting the gradient descent approach to align with the closed-form approach.

After creating a function from the closed-form approach, we tested our results against multiple test alphas to determine the best regularization parameter. We tested when α = [0.001, 0.01, 0.1, 1, 10, 100, 1000]. We output the respective weight, bias, and the test MSE for each α using the "Pandas" dataframe. When rounded to four decimal places, the test MSEs show the following: [13.5197, 12.2576, 11.0134, 12.7108, 20.1799, 33.0254, 53.9702]. From the table, it can be seen that the lowest MSE for this dataset is around 11.0134 when α = 0.1, which is our optimal regularization. We can note that this testing MSE is lower than that of our MSE in our linear regression test. To check our work, we used "sklearn.linear_model" to import "Ridge" which is Python's pre-built ridge regression function. We tested the test alphas to print the MSE using the "Ridge" function and found that the MSE of the different levels matched with our closed-form approach. We also obtain the same conclusion that ridge regression lowered the MSE of the testing set.

LASSO stands for Least Absolute Shrinkage and Selection Operator, and is a linear regression technique with added regularization. Regularization adds a penalty term used to adjust the loss function to prevent over or underfitting. In our case LASSO regression prevents overfitting by penalizing the larger coefficients that cause high losses to our lost function. The penalizing of our coefficients will shrink all our coefficients towards zero and some terms to exactly zero.

Loss function for LASSO: $J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$

Our goal in LASSO regression is to minimize our loss function which is the MSE plus our penalty term which is: $\alpha \sum|\Theta|$. α is our tuning parameter where $\alpha > 0$ and the larger the alpha the bigger the constraint and the more our coefficients are shrunk. The LASSO regression tries to minimize the MSE while increasing bias. If we use analytical methods to solve for the

minimum of our loss function we encounter an issue. The derivative of an absolute value of theta does not exist when theta = 0. This means that there is no closed form solution to our LASSO regression. We would need to use other methods to optimize our loss function. In our project we used gradient descent. Gradient descent is a way of taking a guess for the minimum value then iteratively adjusting that guess by subtracting the gradient at the guess with a step size. This method works for finding the minimum for convex functions. In our case we would have to use a subgradient descent. This is because we will assume that the derivative of absolute value of theta at 0 is between -1 and 1 from the subgradients. But we will just set it equal to 0 if theta is 0. Which means the subgradient of absolute value of theta is just the sign function. Where if theta < 0 it is -1, if theta = 0 it is 0 and theta > 0 is 1. So if we vectorize and use linear algebra to represent our loss function we get that the subgradient of our loss function is:

$$\frac{d}{dW} J(\hat{\theta}) = \frac{1}{m} X^T (Y - \hat{\theta} X) + \alpha sgn(W)$$

$$\frac{d}{db} J(\hat{\theta}) = \frac{1}{m} (Y - \hat{\theta} X)$$

For coding the function we will first need to consider the parameters it takes. First is X the column matrix of all our observations, y a column vector of our dependent variable, alpha the tuning parameter, num_iterations for the number of iterations of gradient descent and learning_rate which is the step size we will take. We will define our function lasso_regression(X, y, alpha, num_iterations, learning_rate). Then we will store the number of independent variables in n and observations in m using X.shape. We then initialize our vector of coefficients W and just set them all equal to 0 for now. Also, initialize the intercept term b as 0 for now. Then we will create a for loop that first finds the predicted value with our current coefficient guesses, then calculates the residuals then the subgradient for our coefficients and performs subgradient descent to find the coefficients that minimize our loss function. After it iterates a certain amount of time it returns us the coefficients W and intercept b.

We tested our function on the boston housing data set at different alpha values (0.001, 0.01, 0.1, 1, 10, 100, 1000, 100000). The MSE of our subgradient descent function for the respective alpha values, learning rate of 0.05 and number of iterations of 10,000 were: [11.357, 12.370, 22.463, 54.927, 67.103, 135.221, 518498.232]. While the MSE for the “sklearn” LASSO we compared it to is [11.413, 12.902, 22.457, 55.314, 75.760, 75.760, 75.760]. It seems like the MSE of our subgradient descent function is less than the "sklearn" LASSO MSE meaning we have found better estimates for our coefficients to minimize our loss function. But this is at a cost of 100000 iterations needed and at a learning rate of 0.05. We can also see that for alpha = 1000 and 10000, our subgradient descent starts to diverge to infinity while the "sklearn" model thresholds the increases so after alpha = 10 all MSE were set to 75.760.

Then we compared the "sklearn" model coefficients to our functions outputs for alpha value of 0.01. After adjusting all the absolute values of our coefficients less than 0.01 to 0 we can see that our gradient descent almost matches up to the "sklearn" model. But this is at the cost of using 200,000 iterations at a learning rate of 0.2. Our function seems to be an inefficient way of finding the coefficients due to subgradient descent not converging as fast as the "sklearn" method of coordinate descent.

Some errors we have is that if the learning rate is too high in our function some overflow error occurs. We may need to add a threshold to shrink coefficients to exactly 0 and find a more efficient converging descent method.

The result of LASSO regression with alpha of 0.01, iterations of 200,000 and learning rate of 0.2 got a training MSE of 10.515 and a test MSE of 12.891. So our test MSE was better than the OLS regression test MSE of 14.33 while having the training and test MSE to be closer for our function which means we are not overfitting as much.

From implementing ridge and LASSO regression, we can see how both regularization techniques address the problem of overfitting. We first determined that we should use ridge and LASSO regression after testing our imported Boston Housing dataset using linear regression. We saw that the mean squared error of the training and test sets had a huge difference. Since the training MSE was low and the test MSE was high, we concluded that overfitting occurred. To try to mitigate this, we used ridge and LASSO regression. They were able to lower the testing MSE to reduce overfitting because of the penalty term in both cost functions. In LASSO regression, it can be seen that the training MSE of 10.515 and test MSE of 12.891 are closer than the training MSE of 5.12 and test MSE of 14.33 in the linear regression model. For ridge regression, we were able to find the optimal regularization parameter for $\alpha = 0.1$ where the test MSE was at its lowest of 11.0134.

References

1. Giba, Lari. “Ridge Regression Explained, Step by Step.” *Machine Learning Compass*, 23 May 2021, machinelearningcompass.com/machine_learning_models/ridge_regression/.
2. Giba, Lari. “Subgradient Descent Explained, Step by Step.” *Machine Learning Compass*, 27 July 2021, machinelearningcompass.com/machine_learning_math/subgradient_descent/.
3. Guestrin, Carlos. “Ridge Regression - Stanford University.” *CS229: Machine Learning*, cs229.stanford.edu/notes2021fall/lecture10-ridge-regression.pdf. Accessed 15 Dec. 2023.
4. Zach. “Introduction to Ridge Regression.” *Statology*, 10 Feb. 2021, www.statology.org/ridge-regression/.