Regularized Linear Regression

Asm Nurussafa
Electronics Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
asm.nurussafa@stud.hshl.de

***Abstract*—It is of utmost importance to teach our machine learning model to reduce over-fitting, as it leads to lower accuracy. Overfitting may cause from bias- generalization error due to wrong assumptions, or variance- sensitivity to small changes in training data. Our goal is reduce this over-fitting error by implementing Regularised Linear Regression. As the name suggests, we regularize the model by constraining it, i.e. in simple terms, giving it less degrees of freedom and discourage it to learn from flexible models. In this paper, we will see how different linear regression models like Ridge, Lasso, Elastic and Early stopping to improve our accuracy for our machine learning model.**

***Index Terms*—Regularized Linear Regression, Ridge, Lasso, Elastic, Early Stopping.**

I. Introduction

Managerial decision making, organizational efficiency, and revenue generation are all areas that can be improved through the utilization of data-based insights. Currently, these insights are being more readily sought out as technological accessibility stretches further and competitive advantages in the market are harder to acquire. One field that seeks to realize value within collected data samples is predictive analytics. By leveraging mathematical/statistical techniques and programming, practitioners are able to identify patterns within data allowing for the generation of valuable insights.

Regression is one technique within predictive analytics that is used to predict the value of a continuous response variable given one or many related feature variables. Algorithms of this class accomplish this task by learning the relationships between the input (feature) variables and the output (response) variable through training on a sample dataset. How these relationships are learned, and furthermore used for prediction varies from algorithm to algorithm. The practitioner is faced with options for regression modeling algorithms, however, linear regression models tend to be explored early on in the process due to their ease of application and high explainability.

II. Linear Regression Modeling

A linear regression model learns the input-output relationships by fitting a linear function to the sample data. This can be mathematically formalized as:

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \ldots + \beta_p x_{i,p} + \epsilon_i \forall i \in \{1, \ldots, n\} \tag{1}$$

where:

- p is the number of features.
- n is the number of samples.
- ϵ is an error term with mean of zero and finite variance.

Thus, the response is modeled as a weighted sum of the input variables multiplied by linear coefficients with an error term included. It will prove useful in future steps involving optimization to use vector notation. The linear modeling equation can be expressed this way as:

$$\mathbf{y} = \mathbf{X}\beta + \epsilon \tag{2}$$

where:

- y is a response vector $[y_1, y_2, \ldots, y_n]^{\mathbf{T}}$ of length n.
- X is a $n \times (p+1)$ design matrix of features $[\mathbf{1}, \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_p}]$.
- β is a length $(p+1)$ coefficient vector $[\beta_0, \beta_1, \beta_2, \ldots, \beta_p]$.

An important aspect of the above equation to note is that there is a column of 1's appended to the design matrix. This is such that the first coefficient of the coefficient vector can serve as an intercept term. In cases where an intercept is not sought after this column can be omitted.
Thus the goal of model training is to find an estimate of the coefficient vector, $\boldsymbol{\beta}^{\wedge}$, which can then be utilized with the above equations to make predictions of the response given new feature data. This can be accomplished by applying optimization theory to the model equations above to derive an equation for the model coefficient estimator that minimizes a notion of model error found by training on the sample data.

A. Minimizing a Notion of Model Error

To consider how model error can be minimized, a consideration of model error must first be made. Prediction error for a single prediction can be expressed as:

$$\hat{y}_i = \sum_{j=1}^{p} x_{i,j} \hat{\beta}_j \tag{3}$$

which is a model prediction of sample i, has an error defined as: $y_i - \hat{y}_i$. Thus, in vector notation, total model error across all predictions can be found as:

$$y_i - \hat{y}_i \tag{4}$$

where all sample model predictions, has an associated total model error: $\|\mathbf{y} - \hat{\mathbf{y}}\|_2$.
However, for the uses of finding a minimal overall model error, the $\mathbf{L2}$, norm above is not a good objective function. This is

due to the fact that negative errors and positive errors will cancel out, thus a minimization will find an objective value of zero even though in reality the model error is much higher.

This signed error cancellation issue can be solved by squaring the model's prediction error producing the sum of squared error (SSE) term:

$$\sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{p} x_{i,j} \hat{\beta}_j \right)^2 \tag{5}$$

This same term can be expressed in vector notation as:

$$\|\mathbf{y} - \hat{\mathbf{y}}\|_2^2 = \|\mathbf{y} - \mathbf{X}\hat{\beta}\|_2^2 \tag{6}$$

As will be seen in future optimization applications, this function is much better suited to serve as a loss function, a function minimized that aptly models the error for a given technique. Many different models other than regularized linear models use the SSE error term as a term in their respective loss functions.

III. Ordinary Least Squares

Now that linear modeling and error has been covered, we can move on to the most simple linear regression model, Ordinary Least Squares (OLS). In this case, the simple SSE error term is the model's loss function and can be expressed as:

$$L(\beta) = \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2 = \|\mathbf{y} - \mathbf{X}\beta\|_2^2$$

Equation #7

Using this loss function, the problem can now be formalized as a least-squares optimization problem. This problem serves to derive estimates for the model parameters, , that minimize the SSE between the actual and predicted values of the outcome and is formalized as:

$$\hat{\beta} = \arg\min_{\beta} L(\beta) = \arg\min_{\beta} \frac{1}{2n} \|\mathbf{y} - \mathbf{X}\beta\|_2^2$$

Equation #8

The $1/(2n)$ term is added in order to simplify solving the gradient and allow the objective function to converge to the expected value of the model error by the Law of Large Numbers.

Aided by the problem's unconstrained nature, a closed-form solution for the OLS estimator can be obtained by setting the gradient of the loss function (objective) equal to zero and solving the resultant equation for the coefficient vector, $\boldsymbol{\beta}\wedge$. This produces the following estimator:

$$\hat{\beta} = \left(\mathbf{X}^\mathbf{T}\mathbf{X}\right)^{-1} \left(\mathbf{X}^\mathbf{T}\mathbf{y}\right)$$

Equation #9

However, this may not be the only optimal estimator, thus its uniqueness should be proven. To do this, it will suffice to show that the loss function (Eq. #8) is convex since any local optimality of a convex function is also global optimality and therefore unique.

One possible way to show this is through the second-order convexity conditions, which state that a function is convex if it is continuous, twice differentiable, and has an associated Hessian matrix that is positive semi-definite. Due to its quadratic nature, the OLS loss function (Eq. #8) is both continuous and twice differentiable, satisfying the first two conditions. Furthermore, this Hessian can be shown to be positive semi-definite as:

$$\beta^\mathbf{T} \left(2\mathbf{X}^\mathbf{T}\mathbf{X}\right) \beta = 2(\mathbf{X}\beta)^\mathbf{T}\mathbf{X}\beta = 2\|\mathbf{X}\beta\|_2^2 \succeq 0 \quad \forall \quad \beta$$

Equation #11

Thus, by the second-order conditions for convexity, the OLS loss function (Eq. #8) is convex, thus the estimator found above (Eq #9) is the unique global minimizer to the OLS problem.

A. Implementing the Estimator Using Python and NumPy

Solving for the OLS estimator using the matrix inverse does not scale well, thus the NumPy function solve, which employs the LAPACK_gesv routine, is used to find the least-squares solution. This function solves the equation in the case where A is square and full-rank (linearly independent columns). However, in the case that A is not fullrank, then the function 1stsq should be used, which utilizes the xGELSD routine and thus finds the singular value decomposition of A.

One possible implementation in Python of OLS with an optional intercept term is:

```python
def ols(X, y, fit_intercept=True):
    """Ordinary Least Squares (OLS) Regression model with intercept term.
    Fits an OLS regression model using the closed-form OLS estimator equation.
    Intercept term is included via design matrix augmentation.

    Params:
        X - NumPy matrix, size (N, p), of numerical predictors
        y - NumPy array, length N, of numerical response
        fit_intercept - Boolean indicating whether to include an intercept term

    Returns:
        NumPy array, length p + 1, of fitted model coefficients
    """
    m, n = np.shape(X)
    if fit_intercept:
        X = np.hstack((np.ones((m, 1)), X))
```

B. Drawbacks of OLS

As is with most things, there are tradeoffs that have to made when modeling. One such major tradeoff is that of the bias-variance tradeoff. Any model's error can be broken down into two components: bias and variance. Bias can be considered the error implicit in the modeling algorithm, whereas variance can be considered the error derived from differences in idiosyncrasies across training datasets. A good model is one that should have the overall error minimized, thus both bias and variance should be minimized. However, there is a tradeoff to consider as increasing bias will often decrease variance.

For the OLS model, a high variance is a concern. Since the SSE is being optimized, the model tends to fit outlier data

points since they will produce higher error values due to the squared term within the loss function. By fitting these outlier points, the OLS model can subsequently base predictions off modeled patterns that are only present in the training data — idiosyncratic outlying points — and not representative of the entire population. This phenomenon is called overfitting and can lead to predictive models with low accuracy when generalizing to new predictions.

Since OLS is a low bias model, it is well-suited to have its variance lowered through bias addition, which may result in higher overall predictive ability. One way to add bias is through shrinkage, biasing model coefficient estimates toward zero. This shrinkage can be achieved through the addition of a regularization penalty to the loss function which applies a unique form of shrinkage to the overall coefficient estimates.

IV. Ridge Regression

This form of regression is also known as Tikhonov Regularization and modifies the OLS loss function (Eq. 7) with the addition of an L penalty with an associated tuning parameter, . This loss function can be described using vector notation as:

$$L(\beta) = \|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_2^2 \quad \text{with tuning parameter } \lambda \geq 0$$

Equation #1

Similarly to the OLS case, this loss function can then be formulated as a least-squares optimization problem to find estimates for the model coefficients that minimize the loss function as:

$$\hat{\beta} = \arg\min_{\beta} L(\beta) = \arg\min_{\beta} \frac{1}{2n}\|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_2^2$$

Equation #2

Just like the OLS case, a 1/(2n) term is added in order to simply solving the gradient and allow the objective function to converge to the expected value of the model error by the Law of Large Numbers.

This problem is also unconstrained and a closed-form solution for the Ridge estimator can be found by setting the gradient of the loss function (objective) equal to zero and solving the resultant equation. This produces an estimator result of:

$$\hat{\beta} = (\mathbf{X}^\mathsf{T}\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}$$

Equation #3

This estimator should also be shown to be unique. In this case, the associated Hessian matrix is:

$$\mathbf{H} = 2\mathbf{X}^\mathbf{T}\mathbf{X} + 2\lambda\mathbf{I}$$

Equation #4

It turns out that this matrix can be shown to be positive definite by:

$$\beta^\mathbf{T}(\mathbf{X}^\mathbf{T}\mathbf{X} + \lambda\mathbf{I})\beta = (\mathbf{X}\beta)\mathbf{X}\beta + \lambda\beta^\mathbf{T}\beta = \|\mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_2^2 \succ 0 \;\; \forall \;\; \beta \neq \mathbf{0}$$

Equation #5

Thus, since the associated Hessian matrix of the Ridge loss function is positive definite, the function is strongly convex, which implies that the Ridge estimator (Eq. 3) is the unique global minimizer to the Ridge Regression problem.

Implementations of the estimator can be simplified by noticing that the problem can be reformulated as an OLS problem through data augmentation. This would take the form of:

$$\begin{bmatrix}\hat{\beta}_0 \\ \hat{\beta}\end{bmatrix} = \arg\min_{\beta_0,\beta} \left\| \begin{bmatrix}\mathbf{Y} \\ 0\end{bmatrix} - \begin{bmatrix}\mathbf{1_n} & \mathbf{X} \\ 0 & \lambda\cdot\mathbf{I}\end{bmatrix} \begin{bmatrix}\beta_0 \\ \beta\end{bmatrix} \right\|_2^2$$

Equation #6

Thus, by utilizing the above data augmentation the same result as Equation 9 from the last part can be used to solve for the coefficient estimates. That result is reproduced here:

$$\hat{\beta} = (\mathbf{X}^\mathbf{T}\mathbf{X})^{-1}(\mathbf{X}^\mathbf{T}\mathbf{y})$$

Equation #7

A. Implementing the Estimator Using Python and NumPy

Similar to the OLS case, the matrix inverse does not scale well, thus the NumPy function solve, which employs the LAPACK $_gesv$ routine, is used to find the least-squares solution. This function solves the equation in the case where A is square and full-rank (linearly independent columns). However, in the case that A is not full-rank, then the function lstsq should be used, which utilizes the xGELSD routine and thus finds the singular value decomposition of A.

One possible Python implementation of Ridge Regression with an optional intercept term is:

```
def ridge(X, y, l2):
    """Ridge Regression model with intercept term.
    L2 penalty and intercept term included via design matrix augmentation.
    This augmentation allows for the OLS estimator to be used for fitting.

    Params:
        X - NumPy matrix, size (N, p), of numerical predictors
        y - NumPy array, length N, of numerical response
        l2 - L2 penalty tuning parameter (positive scalar)

    Returns:
        NumPy array, length p + 1, of fitted model coefficients
    """
    m, n = np.shape(X)
    upper_half = np.hstack((np.ones((m, 1)), X))
    lower = np.zeros((n, n))
    np.fill_diagonal(lower, np.sqrt(l2))
    lower_half = np.hstack((np.zeros((n, 1)), lower))
    X = np.vstack((upper_half, lower_half))
    y = np.append(y, np.zeros(n))
    return np.linalg.solve(np.dot(X.T, X), np.dot(X.T, y))
```

V. The Lasso for Regression

The Lasso, or Least Absolute Shrinkage and Selection Operator, includes the addition of an L penalty to the OLS loss function (Part One: Eq. 7), bringing selective model parameters to zero for a large enough value of an associated tuning parameter, . In other words, the Lasso performs automated feature selection producing a vector of model coefficients with sparsity (amount of elements that are zero) varying on the magnitude of a tuning parameter.

The Lasso loss function can be formalized as:

$$L(\beta) = \|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_1 \quad \text{with tuning parameter } \lambda \geq 0$$

Equation #1

Similar to previous cases, an intercept term can be included through data augmentation of the design matrix with a column of 1s. Furthermore, formulating the problem as a least-squares optimization problem produces:

$$\hat{\beta} = \arg\min_{\beta} L(\beta) = \arg\min_{\beta} \frac{1}{2n}\|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_1$$

Equation #2

However, unlike previous cases, no closed-form solution exists for this problem. This is due to the fact that the addition of the L penalty makes the function no longer continuously differentiable because of the non-smooth absolute component. To remedy this issue, a discrete optimization technique needs to be applied to search for an optimal solution.

Numerous algorithms exist to this end, such as LARS (Least Angle Regression) and Forward Stepwise Regression, however, the Pathwise Coordinate Descent algorithm is leveraged within this work. In short, this algorithm optimizes a parameter at a time holding all other parameters constant.

A. Pathwise Coordinate Descent

Before beginning the algorithm, all features should be standardized to have zero mean and variance of one. From there, a p+1 length coefficient vector is initialized to zero. Cycles are then run across all coefficients until convergence — where values of the coefficients stabilize and do not change more than a certain tolerance — is reached. Within each cycle, for every coefficient, an update is calculated and subsequently has the soft-thresholding operator applied to it.

The simplest form of Coordinate Descent updates calculates — for each coefficient — the simple (single variable as opposed to multiple regression) least-squares coefficient value using the partial residuals across all other features in the design matrix. Partial residuals, in this case, are found as:

$$r_{i,j} = y_i - \sum_{k \neq j} x_{i,k}\beta_k$$

Equation #3

Therefore, the estimate for a particular coefficient value can be found as:

$$\beta_j^* = \frac{1}{n}\sum_{i=1}^{n} x_{i,j} r_{i,j}$$

Equation #4

Now, the penalty, as dictated by the tuning parameter, is included in the model through the soft-thresholding operator. This is expressed as:

$$\beta_j = \mathbf{S}(\beta_j^*, \lambda) = sign(\beta_j^*)(|\beta_j^*| - \lambda)_+ = \begin{cases} \beta_j^* - \lambda & \beta_j^* > 0 \text{ and } \lambda < |\beta_j^*| \\ \beta_j^* + \lambda & \beta_j^* > 0 \text{ and } \lambda < |\beta_j^*| \\ 0 & \lambda \geq |\beta_j^*| \end{cases}$$

Equation #5

Naive updates can be utilized for improved efficiency. These updates are found via:

$$\beta_j^* = \frac{1}{n}\sum_{i=1}^{n} x_{i,j} r_i + \beta_j$$

Equation #6

where r is the current model residual for all samples, n.

When the number of samples is much greater than the number of features (n ¿¿ p), further efficiency improvements can be derived by using covariance updates. For these updates, the first term of the Naive update equation above (Eq. 6) is replaced as shown by:

$$\sum_{i=1}^{n} x_{i,j} r_i = \langle x_j, y \rangle - \sum_{k:|\beta_k|>0} \langle x_j, x_k \rangle \beta_k$$

Equation #7

Utilizing warm starts can bring efficiency boosts as well. Using warm starts, a sequence of models are fitted — with tuning parameter values from a max tuning parameter value down to a minimum tuning parameter value that is some small factor (thousandth) of the max value — initializing the coefficients of each iteration to the solution of the last iteration. In many cases, it is actually faster to fit this path of models than a single model for some small . Thus the path can be expressed as:

$$\lambda_{\max} \rightarrow \lambda_{\min} = \epsilon \cdot \lambda_{\max}$$

Equation #8

where $epsilon$ iss typically 0.001 and there are 100 values spaced on a log scale.

Furthermore, the max value of the tuning parameter to begin the path at can be found by finding the minimum value that will bring the estimates for all model coefficients to zero. This is since any values above this value will result in total sparsity of the coefficient vector. The max value of the path (starting point) can be found as:

$$\lambda_{\max} = \frac{\max_l |\langle x_l, y \rangle|}{n}$$

Equation #9

By providing a starting place to begin searching for optimality, warm starting can many times speed up convergence and also puts the pathwise in the Pathwise Coordinate Descent algorithm.

B. Implementation of the Lasso In Python Using NumPy

One possible way to implement pathwise coordinate descent for the Lasso (with options for tuning the convergence tolerance, path length, and returning the path) is:

```python
def lasso(X, y, l1, tol=1e-6, path_length=100, return_path=False):
    """The Lasso Regression model with intercept term.
    Intercept term included via design matrix augmentation.
    Pathwise coordinate descent with co-variance updates is applied.
    Path from max value of the L1 tuning parameter to input tuning parameter value.
    Features must be standardized (centered and scaled to unit variance)

    Params:
        X - NumPy matrix, size (N, p), of standardized numerical predictors
        y - NumPy array, length N, of numerical response
        l1 - L1 penalty tuning parameter (positive scalar)
        tol - Coordinate Descent convergence tolerance (exited if change < tol)
        path_length - Number of tuning parameter values to include in path (positive integer)
        return_path - Boolean indicating whether model coefficients along path should be returned

    Returns:
        if return_path == False:
            NumPy array, length p + 1, of fitted model coefficients
        if return_path == True:
            List, length 3, of last fitted model coefficients, tuning parameter path and coefficie
    """
    X = np.hstack((np.ones((len(X), 1)), X))
    m, n = np.shape(X)
    B_star = np.zeros((n))
    l_max = max(list(abs(np.dot(np.transpose(X[:, 1:]), y)))) / m
    # At or above l_max, all coefficients (except intercept) will be brought to 0
    if l1 >= l_max:
        return np.append(np.mean(y), np.zeros((n - 1)))
    l_path = np.geomspace(l_max, l1, path_length)
    coeffiecients = np.zeros((len(l_path), n))
    for i in range(len(l_path)):
        while True:
            B_s = B_star
            for j in range(n):
                k = np.where(B_s != 0)[0]
                update = (1/m)*((np.dot(X[:,j], y)- \
                                 np.dot(np.dot(X[:,j], X[:,k]), B_s[k])) + \
                                 B_s[j]
                B_star[j] = (np.sign(update) * max(abs(update) - l_path[i], 0))
            if np.all(abs(B_s - B_star) < tol):
                coeffiecients[i, :] = B_star
                break
    if return_path:
        return [B_star, l_path, coeffiecients]
    else:
        return B_star
```

22

VI. The Elastic Net

In this form of regularized linear regression, the OLS loss function is changed by the addition of both an L and L penalty to the loss function with tuning parameters controlling the intensity of regularization and the balance between the different penalties. This loss function can be formalized as:

$$L(\beta) = \|\mathbf{y}-\mathbf{X}\beta\|_2^2+\lambda[(1-\alpha)\frac{1}{2}\|\beta\|_2^2+\alpha\|\beta\|] \text{ with tuning parameters } \lambda \geq 0, 0 \leq \alpha \leq 1$$

Equation #10

Having both L and L penalties, the Elastic Net serves to deliver a compromise between Ridge regression and the Lasso, bringing coefficients towards zero and selectively to zero. Here, can be considered as the parameter determining the ratio of L penalty to add, whereas can be thought of as the intensity of regularization to apply.

The Elastic Net loss function is also used to formalize a least-squares optimization problem:

$$\hat{\beta} = \arg\min_{\beta} L(\beta) = \arg\min_{\beta} \frac{1}{2n}\|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda[(1-\alpha)\frac{1}{2}\|\beta\|_2^2 + \alpha\|\beta\|]$$

Equation #11

Similarly to the Lasso, an intercept is included through design matrix augmentation, a 1/(2n) term is added for mathematical completeness, and pathwise coordinate descent is implemented to solve since a closed-form solution does not exist due to the L penalty term.

Just as in the Lasso, pathwise coordinate descent is used to solve for model coefficient estimates however changes need to be made to the update equations to account for the dual penalties. In this case, the jth coefficient value obtained after soft-thresholding is now found as:

$$\beta_j = \frac{\mathbf{S}(\beta_j^*, \lambda\alpha)}{1+\lambda(1-\alpha)}$$

Equation #12

The soft-thresholding operator is the same operator applied in the Lasso update (Eq. 5):

$$sign(\beta_j^*)(|\beta_j^*| - \lambda\alpha)_+$$

Equation #13

Furthermore, naive updates or covariance updates can be used along with warm starts. For warm starts, the max value can be calculated as:

$$\lambda_{\max} = \frac{\max_l |\langle x_l, y\rangle|}{n\alpha}$$

Equation #14

A. Implementation in Python Using NumPy

One possible way to implement pathwise coordinate descent to solve the Elastic Net (with options for tuning the convergence tolerance, path length, and returning the path) is:

```python
def elastic_net(X, y, l, alpha, tol=1e-4, path_length=100, return_path=False):
    """The Elastic Net Regression model with intercept term.
    Intercept term included via design matrix augmentation.
    Pathwise coordinate descent with co-variance updates is applied.
    Path from max value of the L1 tuning parameter to input tuning parameter value.
    Features must be standardized (centered and scaled to unit variance)

    Params:
        X - NumPy matrix, size (N, p), of standardized numerical predictors
        y - NumPy array, length N, of numerical response
        l - l penalty tuning parameter (positive scalar)
        alpha - alpha penalty tuning parameter (positive scalar between 0 and 1)
        tol - Coordinate Descent convergence tolerance (exited if change < tol)
        path_length - Number of tuning parameter values to include in path (positive integer)

    Returns:
        NumPy array, length p + 1, of fitted model coefficients
    """
    X = np.hstack((np.ones((len(X), 1)), X))
    m, n = np.shape(X)
    B_star = np.zeros((n))
    if alpha == 0:
        l2 = 1e-15
    l_max = max(list(abs(np.dot(np.transpose(X), y)))) / m / alpha
    if l >= l_max:
        return np.append(np.mean(y), np.zeros((n - 1)))
    l_path = np.geomspace(l_max, l, path_length)
    for i in range(path_length):
        while True:
            B_s = B_star
            for j in range(n):
                k = np.where(B_s != 0)[0]
                update = (1/m)*((np.dot(X[:,j], y)- \
                                 np.dot(np.dot(X[:,j], X[:,k]), B_s[k])) + \
                                 B_s[j]
                B_star[j] = (np.sign(update) * max(
                    abs(update) - l_path[i] * alpha, 0)) / (1 + (l_path[i] * (1 - alpha)))
            if np.all(abs(B_s - B_star) < tol):
                break
    return B_star
```

VII. Summary

Throughout this series, different regularized forms of linear regression have been examined as tools to overcome the tendency to overfit training data of the Ordinary Least Squares model. The Elastic Net — due to its balance between regularization varieties — tends to be the most robust to aid the overfitting issue, however, the Lasso certainly can prove helpful in many situations due to its automated feature selection. Ridge Regression is also a good tool to use to ensure

a reduction in possible model overfitting as it shrinks model coefficients towards zero, reducing model variance.

The header image of this series well demonstrates the difference between Ridge Regression and the Lasso, as it can be seen that the model coefficients all shrink towards zero on the left for the Ridge Regression case, and, on the right, coefficients are being brought to zero in a selective order for the Lasso case.

REFERENCES

[1] Géron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. " O'Reilly Media, Inc.".

[2] Theobald, O. (2017). Machine learning for absolute beginners: a plain English introduction (Vol. 157). Scatterplot press.

[3] Bonaccorso, G. (2017). Machine learning algorithms. Packt Publishing Ltd.

[4] Walsh, Wyatt. "Regularized Linear Regression Models." Medium, Towards Data Science, 18 Jan. 2021, towardsdatascience.com/regularized-linear-regression-models-44572e79a1b5.

[5] Friedman, J., Hastie, T., and Tibshirani, R. (2010). Regularization paths for generalized linear models via coordinate descent. Journal of statistical software, 33(1), 1.