Penalize Regression with R Package

A penalized regression approach that has been implemented by separating it according to each algorithm

JuHyun Kang

June 6, 2025

The Three Sisters of Newton
School of Mathematics, Statistics and Data Science
Sungshin Women's University

Outline

Introduction with penalize regression

- Penalized regression is a statistical technique that adds a regularization term to the loss function to prevent overfitting and enable variable selection by shrinking regression coefficients.

$$\min_{\beta} \left\{ \frac{1}{2n} \sum_{i=1}^{n} \left(y_i - x_i^\top \beta \right)^2 + \sum_{j=1}^{p} P_\lambda\left(|\beta_j|\right) \right\}$$

- y_i represents the response variable for observation i
- x_i is the vector of predictor variables for observation i
- β is the vector of regression coefficients
- $P_\lambda(\cdot)$ is a penalty function parameterized by a regularization parameter λ

Penalize Regression Method

- Ridge

  Uses $\ell 2$ penalty to shrink coefficients and reduce multicollinearity without variable selection.

- Lasso

  Employs $\ell 1$ penalty to induce sparsity, enabling automatic variable selection.

- Elastic Net

  Combines $\ell 1$ and $\ell 2$ penalties to balance sparsity and grouping effects, effective with correlated predictors.

- MCP (Minimax Concave Penalty)

  Non-convex penalty designed to reduce bias in large coefficients while performing variable selection.

- SCAD (Smoothly Clipped Absolute Deviation)

  Non-convex penalty that smoothly reduces shrinkage on large coefficients to alleviate bias and encourage sparsity.

Ridge Regression

- The ridge penalty is an L2 norm penalty that shrinks all coefficients towards zero proportionally, helping to reduce multicollinearity and overfitting.

$$P_\lambda(\beta_j) = \frac{\lambda}{2}\beta_j^2$$

- The objective function for ridge regression is strictly convex, ensuring a unique global minimum and making it suitable for convex optimization
- Ridge regression has a closed-form solution given by:

$$\widehat{\beta}_{\text{ridge}} = (X^\top X + \lambda I)^{-1} X^\top Y$$

- It effectively reduces multicollinearity by shrinking the regression coefficients but does not perform variable selection

Lasso Regression

- The lasso penalty is an L1 norm penalty that encourages sparsity by driving some coefficients exactly to zero, effectively performing variable selection.

$$P_\lambda(\beta_j) = \lambda|\beta_j|$$

- The objective function of lasso is convex but not strictly convex, so it may have multiple solutions when predictors are highly correlated.
- The penalty term is not differentiable at $\beta_j = 0$, which requires specialized optimization algorithms such as coordinate descent or subgradient methods.

Elastic Net Regression

- The elastic net penalty is a combination of the ridge and lasso penalty terms.

$$P_\lambda(\beta_j) = \lambda(\alpha|\beta_j| + \frac{1-\alpha}{2}\beta_j^2)$$

- The objective function of elastic net is convex, ensuring a unique global minimum.
- Elastic net is particularly effective when predictors are correlated, as it encourages a grouping effect while maintaining sparsity.
- $\alpha = 1 \rightarrow$ lasso
- $\alpha = 0 \rightarrow$ ridge
- $0 < \alpha < 1 \rightarrow$ elastic net

Minimax Concave Penalty Regression (MCP)

- The MCP is a non-convex penalty designed to reduce the bias of large coefficients while maintaining sparsity.

$$P_\lambda(\beta_j) = \begin{cases} \lambda|\beta_j| - \frac{\beta_j^2}{2\gamma}, & \text{if } |\beta_j| \leq \gamma\lambda, \\ \frac{\gamma\lambda^2}{2}, & \text{if } |\beta_j| > \gamma\lambda. \end{cases}$$

- The parameter γ controls the degree of concavity and non-linearity: larger values make the penalty closer to the lasso penalty.
- MCP enables variable selection and has desirable oracle properties under certain conditions.

Smoothing Clipped Absolute Deviation Regression (SCAD)

- The SCAD penalty is a non-convex penalty designed to overcome the bias problem of lasso for large coefficients while maintaining sparsity.

$$P_\lambda(\beta_j) = \begin{cases} \lambda|\beta_j|, & \text{if } |\beta_j| \le \lambda, \\ -\frac{|\beta_j|^2 - 2a\lambda|\beta_j| + \lambda^2}{2(a-1)}, & \text{if } \lambda < |\beta_j| \le a\lambda, \\ \frac{(a+1)\lambda^2}{2}, & \text{if } |\beta_j| > a\lambda. \end{cases}$$

- The parameter a controls the concavity of the penalty: commonly, $a = 3.7$ is recommended.
- SCAD encourages sparsity and achieves the oracle property under suitable condition.

Outline

Coordinate Descent Algorithm (CDA)

- Coordinate Descent is an iterative optimization algorithm that updates one parameter at a time while keeping the others fixed.
- It is especially efficient for problems where each coordinate update has a closed-form solution, such as in Lasso and Elastic Net regression.
- The algorithm is simple to implement and well-suited for high-dimensional problems.
- For the following objective,

$$\min_{\beta}\left\{\frac{1}{2n}\sum_{i=1}^{n}(y_i - x_i^\top\beta)^2 + \sum_{j=1}^{p}P_\lambda(|\beta_j|)\right\}$$

the coordinate-wise update step is given by:

$$\beta_j^{(k+1)} \leftarrow \text{argmin}_\beta\left\{\frac{1}{2n}\sum_{i=1}^{n}\left(y_i - x_i^\top\beta_{-j}^{(k+1)} - x_{ij}\beta_j\right)^2 + \sum_{j=1}^{p}P_\lambda(|\beta_j|)\right\}$$

where $\beta_{-j}^{(k+1)}$ denotes the current estimates for all parameters except β_j

Fast Iterative Soft-Thresholding Algorithm (FISTA)

- FISTA is an accelerated version of the proximal gradient descent method designed to solve optimization problems with non-smooth penalties, such as the Lasso.
- It achieves faster convergence compared to the standard iterative soft-thresholding algorithm.
- FISTA is widely used in sparse regression models, including Lasso and Elastic Net, due to its efficiency and simplicity.

FISTA Algorithm with Objective function

- Consider the following objective function:

$$\min_{\beta}\{f(\beta) + P_{\lambda}(\beta)\}$$

  where $f(\beta)$ is convex and differentiable, and $P_{\lambda}(\beta)$ s convex but possibly non-smooth.
- FISTA updates proceed as follows:

$$\begin{aligned}
\beta_{k+1} &= \mathrm{prox}_{\eta P_{\lambda}}\Big(y^k - \eta \nabla f(y^k)\Big), \\
t_{k+1} &= \frac{1 + \sqrt{1 + 4t_k^2}}{2}, \\
y^{k+1} &= \beta_{k+1} + \frac{t_k - 1}{t_{k+1}}(\beta_{k+1} - \beta_k).
\end{aligned}$$

  where $\mathrm{prox}_{\eta P_{\lambda}}$ is the proximal operator (often implemented as a soft-thresholding function for the Lasso).

Local Linear Approximation Algorithm (LLA)

- The Local Linear Approximation (LLA) algorithm is used to handle non-convex penalties, such as MCP and SCAD, by approximating the penalty locally with a linear function.
- This transforms the original non-convex optimization problem into a series of convex problems that are easier to solve.
- LLA helps to achieve desirable statistical properties, such as the oracle property.

- For example, the SCAD penalty can be locally approximated at the k-th iteration as follows:

$$P_\lambda(|\beta_j|) \quad P_\lambda(|\beta_j^{(k)}|) + P'(|\beta_j^{(k)}|)(|\beta_j| - |\beta_j^{(k)}|)$$

- Hence, the optimization problem at iteration $k+1$ becomes:

$$\min_\beta \left\{ \frac{1}{2n} \sum_{i=1}^{n} \left(y_i - x_i^\top \beta\right)^2 \; + \; \sum_{j=1}^{p} w_j^{(k)} |\beta_j| \right\}$$

where

$$w_j^{(k)} = P'_\lambda(|\beta_j^{(k)}|).$$

Outline

Form of Penalize Regression Function

```r
penalized_regression <- function(X, y,
                                 method = c("lasso", "ridge",
                                            "scad", "mcp", "elasticnet"),
                                 algorithm = c("cda", "fista", "lla"),
                                 lambda = 1, learning_rate = 0.01,
                                 max_iter = 1000, alpha = 0.5, gamma = 3.7) {
  method <- match.arg(method)
  algorithm <- match.arg(algorithm)

  check_algorithm_compatibility(method, algorithm)
```

```
  if (algorithm == "cda") {
    return(perform_CDA(X, y, method, lambda, learning_rate,
                       max_iter, alpha, gamma))
  } else if (algorithm == "fista") {
    return(perform_FISTA(X, y, method, lambda, learning_rate,
                         max_iter, alpha, gamma))
  } else if (algorithm == "lla") {
    return(perform_LLA(X, y, method, lambda, learning_rate,
                       max_iter, gamma))
  } else {
    stop("Unknown algorithm selected.")
  }
}
```

Check Algorithm Compatability Function

```
# check algorithm
check_algorithm_compatibility <- function(method, algorithm) {
  method <- tolower(method)
  algorithm <- tolower(algorithm)

  if (method == "ridge" && algorithm == "cda") {
    warning("CDA may not be the most appropriate choice for Ridge penalty.")
  }
  if (method %in% c("scad", "mcp") && algorithm == "fista") {
    stop("FISTA is not recommended for non-convex penalties
         like SCAD or MCP.")
  }
  if (!method %in% c("scad", "mcp") && algorithm == "lla") {
    warning("LLA is primarily designed for SCAD or MCP penalties and
            may not be optimal for Ridge, Lasso, or Elastic Net.")
  }
}
```

Coordinate Descent Algorithm (CDA) Function

```
perform_CDA <- function(X, y, method, lambda, learning_rate = 0.01,
                        max_iter = 1000, alpha = 0.5, gamma = 3.7) {
  n <- nrow(X)
  p <- ncol(X)
  beta <- rep(0, p)
  Xy <- t(X) %*% y
  XX <- colSums(X^2)

  soft_threshold <- function(z, t) {
    sign(z) * pmax(0, abs(z) - t)
  }

  for (iter in 1:max_iter) {
    for (j in 1:p) {
      r_j <- y - X %*% beta + X[, j] * beta[j]
      rho_j <- sum(X[, j] * r_j)
```

Penalty Updates in CDA (Part 1)

```
    if (method == "lasso") {
      beta[j] <- soft_threshold(rho_j / XX[j], lambda / XX[j])

    } else if (method == "ridge") {
      beta[j] <- rho_j / (XX[j] + 2 * lambda)

    } else if (method == "elasticnet") {
      z <- rho_j / XX[j]
      beta[j] <- soft_threshold(z, lambda * alpha / XX[j]) /
        (1 + lambda * (1 - alpha) / XX[j])

    } else if (method == "scad") {
      z <- rho_j / XX[j]
      if (abs(z) <= lambda) {
        beta[j] <- soft_threshold(z, lambda / XX[j])
      } else if (abs(z) <= gamma * lambda) {
        beta[j] <- soft_threshold(z, gamma * lambda / (gamma - 1) / XX[j])
      } else {
        beta[j] <- z}
```

Penalty Updates in CDA (Part 2)

```
    } else if (method == "mcp") {
      z <- rho_j / XX[j]
      abj <- abs(z)

      # MCP 업데이트
      if (abj <= gamma * lambda) {
        beta[j] <- soft_threshold(z, lambda / XX[j]) / (1 - 1 / gamma)
      } else {
        beta[j] <- z
      }

    } else {
      stop("Unsupported method in CDA.")
    }
  }
}

return(beta)
}
```

Fast Iterative Shrinkage-Thresholding Algorithm (FISTA) Function

```r
perform_FISTA <- function(X, y, method, lambda, learning_rate = 1e-3,
                          max_iter = 1000, alpha = 0.5, gamma = 3.7) {
  n <- nrow(X)
  p <- ncol(X)

  beta <- rep(0, p)
  beta_old <- beta
  t <- 1

  soft_threshold <- function(z, t) {
    sign(z) * pmax(0, abs(z) - t)
  }

  grad <- function(beta) {
    -t(X) %*% (y - X %*% beta)
  }
  penalty_grad <- function(beta_j) {
    if (method == "lasso") {
      return(lambda * sign(beta_j))
```

Penalty Gradient Definitions in FISTA (Part 1)

```r
  } else if (method == "ridge") {
    return(2 * lambda * beta_j)

  } else if (method == "elasticnet") {
    return(lambda * (alpha * sign(beta_j) + 2 * (1 - alpha) * beta_j))

  } else if (method == "scad") {

    abj <- abs(beta_j)
    if (abj <= lambda) {
      return(lambda * sign(beta_j))

    } else if (abj <= gamma * lambda) {
      return(((gamma * lambda - abj) / (gamma - 1)) * sign(beta_j))

    } else {
      return(0)
    }
```

Penalty Gradient Definitions in FISTA (Part 2)

```
    } else if (method == "mcp") {
      abj <- abs(beta_j)
      if (abj <= gamma * lambda) {
        return(lambda * (1 - abj / (gamma * lambda)) * sign(beta_j))
      } else {
        return(0)
      }
    } else {
      stop("Unsupported method.")
    }
  }

  for (k in 1:max_iter) {
    z <- beta + ((t - 1) / (t + 2)) * (beta - beta_old)
    grad_z <- grad(z)
        beta_new <- numeric(p)
    for (j in 1:p) {
      if (method == "ridge") {
        beta_new[j] <- z[j] - learning_rate * (grad_z[j] + 2 * lambda*z[j])
```

Iterative Updates in FISTA: Gradient and Thresholding Steps

```
    } else if (method %in% c("lasso", "elasticnet")) {
      pen_grad <- lambda * (if (method == "lasso") sign(z[j])
                            else alpha * sign(z[j]) + 2 * (1 - alpha)
                            * z[j])
      beta_new[j] <- soft_threshold(z[j] - learning_rate * grad_z[j],
                                    learning_rate * lambda * alpha)
    } else if (method %in% c("scad", "mcp")) {
      beta_new[j] <- soft_threshold(z[j] - learning_rate * grad_z[j],
                                    learning_rate * abs(penalty_grad(z[j])))
    } else {
      stop("Unsupported method in FISTA.")
    }
  }
      beta_old <- beta
  beta <- beta_new
  t <- t + 1
}return(beta)}
```

Local Linear Approximation (LLA) Function

```r
perform_LLA <- function(X, y, method, lambda, learning_rate = 0.01,
                        max_iter = 100, alpha = 0.5, gamma = 3.7) {
  n <- nrow(X)
  p <- ncol(X)
  beta <- rep(0, p)
  tol <- 1e-4

  for (iter in 1:max_iter) {
    weights <- rep(1, p)

    for (j in 1:p) {
      bj <- beta[j]

      if (method == "lasso") {
        weights[j] <- 1
      } else if (method == "ridge") {
        weights[j] <- 2 * abs(bj)
      } else if (method == "elasticnet") {
        weights[j] <- alpha + 2 * (1 - alpha) * abs(bj)
```

Weight Computation in LLA: Penalty-Specific Weights

```r
    } else if (method == "scad") {
      abj <- abs(bj)
      if (abj <= lambda) {
        weights[j] <- 1
      } else if (abj <= gamma * lambda) {
        weights[j] <- (gamma * lambda - abj) / ((gamma - 1) * lambda)
      } else {
        weights[j] <- 0
      }
    } else if (method == "mcp") {
      abj <- abs(bj)
      if (abj <= gamma * lambda) {
        weights[j] <- 1 - abj / (gamma * lambda)
      } else {
        weights[j] <- 0
      }
    } else {
      stop("Unsupported method in LLA.")
```

```
      }
    }

    for (j in 1:p) {
      r_j <- y - X %*% beta + X[, j] * beta[j]
      rho_j <- sum(X[, j] * r_j)
      XX_j <- sum(X[, j]^2)
      beta[j] <- sign(rho_j) * max(0, abs(rho_j) - lambda * weights[j]) /
        XX_j
    }
  }

  return(beta)
}
```

Q & A

Thank you :)