Lecture 8

Penalized Regression with optimization

We’ve looked at various penalty functions, and various optimization algorithms. Now, we are going to combine these elements to fully customize our penalized regression models. We are also going to time the elements so that we can assess which algorithms or settings converge the quickest.

We’ll build some specific examples that you can adapt using elements we’ve looked at before. Initially, we’ll consider four penalty situations:

- Lasso (L1): Encourages sparsity.
- Ridge (L2): Encourages shrinkage of coefficients.
- Elastic Net: Combines L1 and L2 penalties.
- Custom Penalties: You can define your own penalty function, for example, one that encourages group sparsity or specific types of shrinkage. Our example will involve $\lambda \sum_{i=1}^{n} |\beta_i|^p$, where we’ve set $p = 1.5$.

We’ll apply three different optimization algorithms:
- Gradient Descent: A basic and widely-used optimization algorithm.
- Coordinate Descent: Often used for Lasso because it efficiently handles the non-differentiability at zero.
- Stochastic Gradient Descent: Useful for large datasets.

We'll use the system.time() function in R to measure the efficiency of the optimization algorithms.

```
# Load necessary libraries
library(MASS)

# Define the dataset
data <- mtcars
x <- as.matrix(data[, c("disp", "hp", "drat", "wt", "qsec")])
y <- data$mpg

# Standardize the variables
x <- scale(x)
y <- scale(y)
```

The example selects 5 of the variables to use in the example, but we could run this on all 10 other variables for variable selection purposes. Also, note that the example here is using the built-in scale function. We could replace these with a custom scale function if we chose.

```
# Lasso penalty (L1)
lasso_penalty <- function(beta, lambda) {
  lambda * sum(abs(beta))
}
```

```
# Ridge penalty (L2)
ridge_penalty <- function(beta, lambda) {
  lambda * sum(beta^2)
}

# Elastic Net penalty
elastic_net_penalty <- function(beta, lambda, alpha = 0.5) {
  alpha * lasso_penalty(beta, lambda) + (1 - alpha) * ridge_penalty(beta, lambda)
}

# Custom penalty (for demonstration)
custom_penalty <- function(beta, lambda) {
  lambda * sum(abs(beta)^1.5)  # Custom non-linear penalty
}
```

We next need to define the loss function that will need to be optimized. Note that the standard loss function here uses the residual sum of squares (consistent with standard regression methods). This piece of the loss function can also be customized.

```
# Generalized loss function
loss_function <- function(beta, x, y, penalty_func, lambda, ...) {
  rss <- sum((y - x %*% beta)^2)
  penalty <- penalty_func(beta, lambda, ...)
  return(rss + penalty)
}
```

We'll start with gradient descent.

```
gradient_descent <- function(x, y, penalty_func, lambda, learning_rate = 0.01, max_iter = 1000) {
  n <- nrow(x)
  p <- ncol(x)
  beta <- rep(0, p)

  for (i in 1:max_iter) {
    # Calculate the gradient
    residuals <- y - x %*% beta
    gradient <- -2 * t(x) %*% residuals / n

    # Apply penalty gradient
    penalty_gradient <- grad_penalty(beta, penalty_func, lambda)

    # Update beta
    beta <- beta - learning_rate * (gradient + penalty_gradient)
  }

  return(beta)
}
```

```
grad_lasso <- function(beta, lambda) {
  lambda * sign(beta)
}

grad_ridge <- function(beta, lambda) {
  2 * lambda * beta
}

grad_elastic_net <- function(beta, lambda, alpha = 0.5) {
  alpha * lambda * sign(beta) + (1 - alpha) * 2 * lambda * beta
}

grad_custom <- function(beta, lambda) {
  lambda * 1.5 * abs(beta)^(0.5) * sign(beta)
}

# Then in the optimization function, call the appropriate gradient function based on the penalty
type.
gradient_descent <- function(x, y, penalty_func, lambda, learning_rate = 0.01, max_iter = 1000,
alpha = 0.5) {
  n <- nrow(x)
  p <- ncol(x)
  beta <- rep(0, p)

  for (i in 1:max_iter) {
    residuals <- y - x %*% beta
    gradient <- -2 * t(x) %*% residuals / n

    # Determine the penalty gradient
    penalty_gradient <- switch(
      penalty_func,
      "lasso" = grad_lasso(beta, lambda),
      "ridge" = grad_ridge(beta, lambda),
      "elastic_net" = grad_elastic_net(beta, lambda, alpha),
      "custom" = grad_custom(beta, lambda),
      stop("Unknown penalty function")
    )

    beta <- beta - learning_rate * (gradient + penalty_gradient)
  }

  return(beta)
}
```

Note that the derivative of the grad penalty for the custom function will have to be updated if you change the power from $p = 1.5$. Use the power rule. The coefficient will be p, and the new power will be $p - 1$ (recall from calculus 1).

Now, we can run the models and check the timing.

```
# Example Usage
system.time({
  beta_lasso <- gradient_descent(x, y, penalty_func = "lasso", lambda = 0.1, learning_rate = 0.01)
})

system.time({
  beta_ridge <- gradient_descent(x, y, penalty_func = "ridge", lambda = 0.1, learning_rate = 0.01)
})

system.time({
  beta_elastic_net <- gradient_descent(x, y, penalty_func = "elastic_net", lambda = 0.1,
learning_rate = 0.01)
})

system.time({
  beta_custom <- gradient_descent(x, y, penalty_func = "custom", lambda = 0.1, learning_rate =
0.01)
})
```

Finally, we can compare the standard penalty functions with functions built into packages.

```
# Lasso using glmnet
library(glmnet)
system.time({
  lasso_model <- glmnet(x, y, alpha = 1)
})

# Ridge using glmnet
system.time({
  ridge_model <- glmnet(x, y, alpha = 0)
})

# Elastic Net using glmnet
system.time({
  elastic_net_model <- glmnet(x, y, alpha = 0.5)
})
```

These package functions have been optimized, so we would expect a faster convergence than our custom functions. You may need a larger dataset to do a proper timing comparison. The convergence may be too quick to produce noticeably different results.

Interpretation of Results:
Compare the coefficients obtained from the custom implementations and the glmnet package. Discuss differences and analyze the accuracy and efficiency of the custom implementation.

Timing Analysis:

Evaluate the time taken for each method and discuss the efficiency of different optimization algorithms. Consider the impact of varying the learning rate, maximum iterations, and penalty parameters on both the solution and the time taken.

Customization:
Discuss how penalty functions and optimization algorithms can be customized further for specific applications, such as using adaptive learning rates, early stopping criteria, or combining different types of penalties.

Coordinate Descent example:
In the case of Lasso, apply a soft-thresholding operator to update the coefficients. This helps manage the non-differentiability at zero.

```
# Coordinate Descent for Lasso
coordinate_descent <- function(x, y, penalty_func = "lasso", lambda = 0.1, max_iter = 1000,
alpha = 0.5) {
  n <- nrow(x)
  p <- ncol(x)
  beta <- rep(0, p)
  X_transpose <- t(x)

  for (iter in 1:max_iter) {
    for (j in 1:p) {
      # Predict current residuals excluding the j-th predictor
      residuals <- y - x %*% beta + x[, j] * beta[j]

      # Unregularized update for beta[j]
      rho <- X_transpose[j, ] %*% residuals / n

      # Update rule depends on the penalty type
      if (penalty_func == "lasso") {
        beta[j] <- soft_thresholding(rho, lambda)
      } else if (penalty_func == "ridge") {
        beta[j] <- rho / (1 + 2 * lambda)
      } else if (penalty_func == "elastic_net") {
        beta[j] <- soft_thresholding(rho, alpha * lambda) / (1 + (1 - alpha) * 2 * lambda)
      } else if (penalty_func == "custom") {
        beta[j] <- custom_update(rho, lambda)  # Define your custom update rule
      } else {
        stop("Unknown penalty function")
      }
    }
  }

  return(beta)
}

# Soft Thresholding Operator for Lasso
soft_thresholding <- function(rho, lambda) {
  sign(rho) * max(0, abs(rho) - lambda)
}

# Custom update for custom penalty (example)
custom_update <- function(rho, lambda) {
  # Example custom update (you can adjust based on your penalty)
  return(rho / (1 + lambda * abs(rho)^0.5))
}

# Example usage
system.time({
  beta_lasso_cd <- coordinate_descent(x, y, penalty_func = "lasso", lambda = 0.1)
})
```

Coordinate Descent is often more efficient than full gradient descent for sparse models like Lasso because it can zero out coefficients quickly. Depending on the penalty and the structure of the data, the algorithm might converge faster or slower. It's good to experiment with different max_iter values to ensure convergence.

Stochastic Gradient Descent:

```
sgd_penalized <- function(x, y, penalty_func = "lasso", lambda = 0.1, learning_rate = 0.01,
max_epochs = 1000, alpha = 0.5) {
  n <- nrow(x)
  p <- ncol(x)
  beta <- rep(0, p)

  for (epoch in 1:max_epochs) {
    for (i in sample(1:n, n, replace = FALSE)) {
      xi <- x[i, , drop = FALSE]  # Get i-th data point as a row vector
      yi <- y[i]

      # Compute the prediction and residual
      prediction <- xi %*% beta
      residual <- yi - prediction

      # Gradient of the loss (for a single data point)
      gradient <- -2 * t(xi) %*% residual

      # Penalty gradient
      penalty_gradient <- switch(
        penalty_func,
        "lasso" = grad_lasso(beta, lambda),
        "ridge" = grad_ridge(beta, lambda),
        "elastic_net" = grad_elastic_net(beta, lambda, alpha),
        "custom" = grad_custom(beta, lambda),
        stop("Unknown penalty function")
      )
```

```
   # Update the coefficients
   beta <- beta - learning_rate * (gradient + penalty_gradient)
  }
 }

 return(beta)
}

# Reusing the gradient functions from the previous example
grad_lasso <- function(beta, lambda) {
 lambda * sign(beta)
}

grad_ridge <- function(beta, lambda) {
 2 * lambda * beta
}

grad_elastic_net <- function(beta, lambda, alpha = 0.5) {
 alpha * lambda * sign(beta) + (1 - alpha) * 2 * lambda * beta
}

grad_custom <- function(beta, lambda) {
 # Example custom gradient
 lambda * 1.5 * abs(beta)^(0.5) * sign(beta)
}

# Example usage with the mtcars dataset
# Preparing the data
data <- mtcars
y <- data$mpg
x <- as.matrix(data[, c("cyl", "disp", "hp", "wt", "qsec")])

# Standardize the data
x <- scale(x)
y <- scale(y)

# Running the SGD
set.seed(123)  # Set a seed for reproducibility
system.time({
 beta_sgd <- sgd_penalized(x, y, penalty_func = "lasso", lambda = 0.1, learning_rate = 0.01,
max_epochs = 500)
})

# Display the coefficients
beta_sgd
```

Experiment with setting different seeds to see how the convergence could change. One way of dealing with the randomization is to run the model several times and then average the results.

Random Shuffling: In each epoch, the data is randomly shuffled to ensure that the model parameters are updated with data points in a different order each time.

Gradient Calculation: For each data point, the gradient of the loss function and the penalty is computed. This is then used to update the model parameters.

Penalty Gradient: Similar to the earlier coordinate descent example, different penalty gradients are used depending on whether Lasso, Ridge, or Elastic Net is selected.

Learning Rate: Controls the size of the steps taken towards the minimum. It's crucial to tune this parameter, as too large a value can cause overshooting, while too small a value can result in slow convergence.

Benefits and Drawbacks of SGD
Pros:

- Efficiency: Much faster than full-batch gradient descent, especially on large datasets.
- Scalability: Suitable for very large datasets where full-batch gradient descent is impractical.

Cons:

- Convergence Issues: The noisy updates can sometimes make it difficult to converge to the exact minimum, leading to fluctuations around the minimum.
- Learning Rate Sensitivity: The choice of learning rate is critical and often requires careful tuning.

Customization Options

- Learning Rate Schedules: Implement learning rate decay strategies where the learning rate decreases as the number of epochs increases, helping with convergence.
- Mini-Batch SGD: Instead of using a single data point or the entire dataset, you can use small mini-batches of data points to update the model parameters. This strikes a balance between the efficiency of SGD and the stability of full-batch gradient descent.

Implement Penalized Splines with Custom Optimization
Spline Basis Creation:

- Construct a spline basis for your model. This can be done using B-splines or other types of splines. For simplicity, let's use B-splines.

Penalty Function:

- Define a penalty function, typically on the second derivative of the spline to control smoothness. You can customize this penalty.

Optimization Algorithm:

- Use a custom optimization algorithm (like SGD, Coordinate Descent, etc.) to fit the spline coefficients, incorporating the penalty.

We'll use a B-spline basis for the example.

```
library(splines)

# Example dataset
data <- mtcars
```

```
y <- data$mpg
x <- data$wt

# Create B-spline basis for 'x'
k <- 4  # Number of knots
degree <- 3  # Degree of spline
bs_basis <- bs(x, df = k, degree = degree)

# Design matrix (including intercept)
X <- cbind(1, bs_basis)
```

We define a custom penalty function.

```
# Penalty matrix for second derivative (assuming B-splines)
penalty_matrix <- diff(diag(k + degree), differences = 2) %*% t(diff(diag(k + degree), differences =
2))

# Regularization parameter
lambda <- 0.1
```

For our example, we'll use coordinate descent, but we can use other methods.

```
# Coordinate Descent for Penalized Spline Regression
coordinate_descent_spline <- function(X, y, lambda = 0.1, max_iter = 1000, tol = 1e-6) {
  p <- ncol(X)
  beta <- rep(0, p)

  for (iter in 1:max_iter) {
    beta_old <- beta

    for (j in 1:p) {
      residuals <- y - X %*% beta + X[, j] * beta[j]

      # Ridge penalty for smoothness (second derivative penalty)
      penalty_term <- lambda * penalty_matrix[j, ] %*% beta

      # Update rule (assuming squared loss + ridge penalty)
      beta[j] <- sum(residuals * X[, j]) / sum(X[, j]^2 + lambda * penalty_matrix[j, j])
    }

    # Check convergence
    if (sqrt(sum((beta - beta_old)^2)) < tol) {
      break
    }
  }

  return(beta)
}
```

```
# Example usage
beta_spline <- coordinate_descent_spline(X, y, lambda = 0.1)

# Predictions
y_pred <- X %*% beta_spline

# Plotting
plot(x, y, pch = 19, main = "Penalized Spline Fit")
lines(sort(x), y_pred[order(x)], col = "blue", lwd = 2)
```

Custom Penalty Functions: You can modify the penalty function to enforce different smoothness criteria. For example, you could penalize higher derivatives for smoother fits or modify the structure of the penalty matrix.

Choice of Optimization Algorithm: While coordinate descent is relatively simple, stochastic gradient descent (SGD) or even more complex methods like proximal gradient methods can be applied. Each has trade-offs between speed, accuracy, and ease of implementation.

Handling Multiple Variables: If you have multiple predictors, you would create a spline basis for each and possibly use an additive model structure. The optimization will then update coefficients for each basis function for each predictor.

Scaling: For large datasets or many basis functions, scaling becomes critical. You may need to implement techniques like mini-batching (in SGD) or more efficient matrix operations.

Splines have a number of hyperparameters that need to be set to achieve the best results. We can also create an optimization algorithm to select the best number of knots, and the best λ smoothing parameter for better results. Typically, this is done through cross validation or other methods.

Step-by-Step Process

Grid Search:

- Define a grid of potential values for the number of knots or the smoothing parameter λ.

Cross-Validation Setup:

- For each combination of knots and λ values, perform cross-validation to estimate the model's performance. Common metrics include mean squared error (MSE), root mean squared error (RMSE), or mean absolute error (MAE).

Model Training and Evaluation:

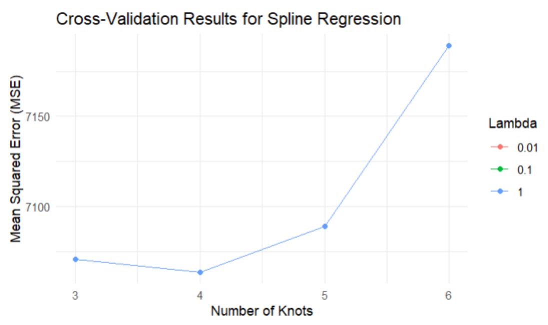
- For each fold of cross-validation, fit the model using the training data and evaluate it using the validation data. Record the performance metrics.

Optimal Parameter Selection:

- After completing the cross-validation process, select the combination of knots and λ that yields the best cross-validation performance.

Algorithm in R
Here's a basic implementation of such an algorithm in R, using a grid search over the number of knots and λ values, with cross-validation to find the optimal parameters. Here, we'll use built-in spline functions and penalty functions, and focus just on selecting the two hyperparameters.

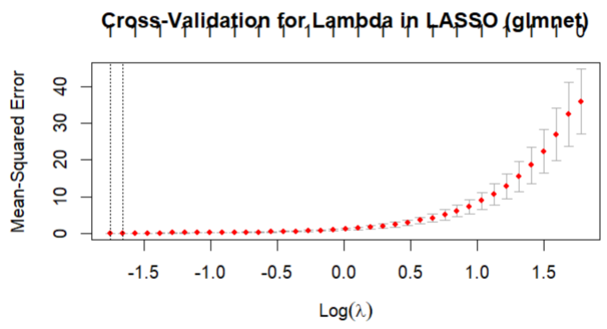
```
# Load necessary library
library(splines)

# Load the dataset
data(mtcars)

# Define the number of knots to test
knots_values <- c(3, 4, 5, 6)

# Define lambda values for regularization (for LASSO/Ridge, which we'll not implement here
directly)
lambda_values <- c(0.01, 0.1, 1)

# Placeholder for results
results <- data.frame(n_knots = integer(), lambda = numeric(), mse = numeric())

# Perform cross-validation to find the optimal number of knots
set.seed(123)
n <- nrow(mtcars)
fold_indices <- sample(rep(1:5, length = n))

for (n_knots in knots_values) {
  num_basis <- n_knots + 2
  for (lambda in lambda_values) {
    fold_mses <- numeric(5)

    for (fold in 1:5) {
      train_indices <- which(fold_indices != fold)
      test_indices <- which(fold_indices == fold)

      x_train <- mtcars$mpg[train_indices]
      y_train <- mtcars$hp[train_indices]
      x_test <- mtcars$mpg[test_indices]
      y_test <- mtcars$hp[test_indices]

      # Create B-spline basis for training data
      bs_basis_train <- bs(x_train, df = num_basis, degree = 3)
      X_train <- cbind(1, bs_basis_train)

      # Fit the model using linear regression (not penalized for this example)
      model <- lm(y_train ~ X_train - 1)

      # Create B-spline basis for test data
      bs_basis_test <- bs(x_test, df = num_basis, degree = 3)
      X_test <- cbind(1, bs_basis_test)

      y_pred <- predict(model, newdata = data.frame(X_test))

      # Calculate mean squared error on the test set
      fold_mses[fold] <- mean((y_test - y_pred)^2)
    }

    # Average MSE across all folds
    mean_mse <- mean(fold_mses)
    results <- rbind(results, data.frame(n_knots = n_knots, lambda = lambda, mse = mean_mse))
  }
}

# Find the best number of knots and lambda
best_params <- results[which.min(results$mse), ]
print(best_params)

# Plotting the cross-validation results for spline regression
library(ggplot2)

# Assuming 'results' is the data frame with your cross-validation results
ggplot(results, aes(x = n_knots, y = mse, color = factor(lambda))) +
  geom_line() +
  geom_point() +
  labs(title = "Cross-Validation Results for Spline Regression",
       x = "Number of Knots",
       y = "Mean Squared Error (MSE)",
       color = "Lambda") +
  theme_minimal()
```



We can likewise use cross validation to check for the best λ for penalized regression.

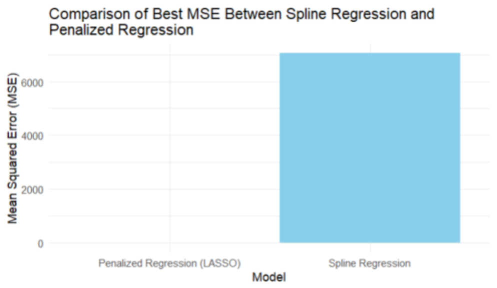
```
# Load necessary library
library(glmnet)

# Prepare the data
```

```
x <- as.matrix(mtcars[, c("mpg", "cyl", "disp", "hp", "wt")])
y <- mtcars$mpg

# Perform cross-validation to find the optimal lambda for LASSO
cv_model <- cv.glmnet(x, y, alpha = 1)  # alpha = 1 for LASSO, 0 for Ridge

# Get the best lambda
best_lambda <- cv_model$lambda.min
print(best_lambda)

# Fit the final model with the best lambda
final_model <- glmnet(x, y, alpha = 1, lambda = best_lambda)

# Print model coefficients
print(coef(final_model))

# Load necessary library
library(glmnet)

# Assuming you've run the cross-validation for `glmnet` as shown previously
plot(cv_model)
title("Cross-Validation for Lambda in LASSO (glmnet)")
```



We can create a comparison of the two model types.

```
# Suppose you have the best MSEs from spline regression and glmnet
best_spline_mse <- min(results$mse)
best_glmnet_mse <- min(cv_model$cvm)

# Data frame for comparison
comparison_df <- data.frame(
  Model = c("Spline Regression", "Penalized Regression (LASSO)"),
  MSE = c(best_spline_mse, best_glmnet_mse)
)

# Plotting the comparison
ggplot(comparison_df, aes(x = Model, y = MSE)) +
  geom_bar(stat = "identity", fill = "skyblue") +
  labs(title = "Comparison of Best MSE Between Spline Regression and Penalized Regression",
      x = "Model",
      y = "Mean Squared Error (MSE)") +
  theme_minimal()
```



The difference here makes me wonder if LASSO is overfit, or if the spline model is really that bad. This is a big difference in MSEs.

Resources:

1. https://medium.com/@HalderNilimesh/understanding-and-implementing-penalized-regression-in-r-a-comprehensive-guide-with-code-examples-c73347138159
2. https://cran.r-project.org/web/packages/penalized/vignettes/penalized.pdf