42

Zhuodiao Kuang

2024-03-17

Report

Path-wise coordinate descendent

Path-wise coordinate descent is a popular optimization technique used for fitting Lasso models. Lasso (Least Absolute Shrinkage and Selection Operator) is a regression analysis method that performs both variable selection and regularization in order to enhance the prediction accuracy and interpretability of the statistical model it produces.

How Path-wise Coordinate Descent Works

Path-wise coordinate descent optimizes the Lasso problem by iteratively updating one parameter at a time while keeping the others fixed. This approach simplifies the optimization problem into a series of one-dimensional problems that are much easier to solve.

Algorithm Steps

1. **Initialization**: Start with an initial estimate for β, often $\beta = 0$.
2. **Cyclically Update Each Coordinate**:
    - For each coefficient β_j in the vector β, update the value of β_j by solving the one-dimensional optimization problem, which involves minimizing the objective function with respect to β_j, keeping all other coefficients fixed.
    - This step often uses the soft-thresholding function due to the L1 penalty in the Lasso objective. The updated value for β_j is given by:

    $$\beta_j = \mathrm{S}(\frac{1}{n} X_j^T (y - X_{-j}\beta_{-j}), \lambda)$$

    where X_j is the j^{th} column of X, X_{-j} is the matrix X excluding X_j, β_{-j} is the coefficient vector excluding β_j, and S is the soft-thresholding operator.
3. **Convergence Check**: Repeat the cyclic update until the changes in β are below a certain threshold, indicating convergence.

Key Features

- **Efficiency**: The path-wise coordinate descent method is computationally efficient, especially for high-dimensional datasets, because it breaks down the complex optimization problem into simpler one-dimensional updates.
- **Sparsity**: By leveraging the L1 penalty, it effectively produces sparse models, where irrelevant variables are given coefficients of zero, thus performing variable selection.
- **Path Solution**: It can efficiently compute the entire path of solutions for a sequence of λ values, which is useful for selecting the best model through cross-validation.

Path-wise coordinate descent is favored for its simplicity, efficiency, and effectiveness in handling large, sparse datasets common in machine learning and statistical applications.

```r
# Define the sigmoid function
sigmoid <- function(z) {
  1 / (1 + exp(-z))
}


Pathwise = function(x, y, lambda = 0.001, beta = rep(0,ncol(x)), tol = 0.01){
  step = 1
  # n = nrow(x)
  # Calculate the initial loglike loss
  iniLL = -sum((y * (x %*% beta) - log(1 + exp(x %*% beta)))) + lambda *sum(beta[2:length(beta)])

  # Optimize log likelihood loss
  while (1) {
    # Notations for difference in beta now and before
    Changed = rep(TRUE, ncol(x))
    # restore the initial value
    beta_temp = beta


    # coordinate descendent
    for (j in 1:ncol(x)) {
      yhat = 1 / (1 + exp(-(x %*% beta)))
      e = y - yhat
      w = yhat * (1 - yhat)

      diff = x %*% beta - x[, -j] %*% beta[-j]
      beta_new = sum(x[, j] * ( w * diff + e ))

      m <- nrow(x)
      predictions <- sigmoid(x %*% beta)
      gradient <- t(x) %*% (predictions - y) / m

       if (j == 1) { # Skip regularization for intercept
      beta_new <- beta_new -  gradient[j]
    } else {
      ##use soft threshold for covariates##
      beta_new = sign(beta_new) * pmax(abs(beta_new) - lambda, 0)

    }
       beta_new = beta_new / (t(w) %*% (x[, j]^2))
```

```
      # Label if not changed too much
      if (abs(beta_new - beta[j]) < tol) {
        Changed[j] = FALSE
      }

      # save the new beta
      beta[j] = beta_new
    }

    ### coordinate descendant ends at this turn ###


    # calculate loglike loss after all the coeffient have been updated at this turn
    loglike_loss = -sum(w * (y * (x %*% beta) - log(1 + exp(x %*% beta)))) + lambda *sum(beta[2:length(

    if (iniLL < loglike_loss ||sum(Changed) == 0){

      return(list(beta_temp = beta_temp, loglike_loss = loglike_loss))
    }

    # default setting
    iniLL = loglike_loss
    step = step + 1
  }
}
```

training model

The `LASSO` function provided appears to implement logistic regression with LASSO regularization, allowing for variable selection and regularization of coefficients to prevent overfitting. Let's break down the code into its components and describe the mathematical notations and operations corresponding to each significant part.

1. **Including Zero Lambda:**
   - If `include_zero_lambda` is TRUE and zero is not already in the lambda sequence, zero is added to the sequence.
     - $\lambda = \lambda \cup \{0\}$ if $\lambda \not\ni 0$ and `include_zero_lambda` is TRUE.
2. **Standardizing Features:**
   - Calculate column means and scale of `x` (features matrix), then standardize `x` by subtracting the mean and dividing by its scale.
     - $\text{colmean}_i = \frac{1}{n}\sum_{j=1}^{n} x_{ji}$ for each column i of x.
     - Standardization: $x_{ji} = \frac{x_{ji} - \text{colmean}_i}{\sqrt{\sum_{j=1}^{n}(x_{ji} - \text{colmean}_i)^2}}$.
3. **Augmenting Feature Matrix:**
   - Augment `x` with a column of ones for the intercept.
     - $x = [1, x]$, where 1 represents a column vector of ones.

4. **Initialization of Coefficients:**
   - Initialize a coefficients vector `beta` with zeros.
     - $\beta = \mathbf{0}$, where $\mathbf{0}$ is a zero vector of length equal to the number of features plus one (for the intercept).
5. **Handling Single vs. Multiple Lambda Values:**
   - If only one lambda value is provided, compute `beta` using the Pathwise Coordinate Descent method for that lambda.
   - For multiple lambda values, iterate over them in decreasing order, updating `beta` for each, optionally using warm starts (reusing the `beta` from the previous lambda as the starting point for the next).
6. **Pathwise Coordinate Descent Call:**
   - For each lambda value:
     - If `warm_start` is TRUE, update `beta` using the Pathwise Coordinate Descent method, starting from the current `beta`.
     - If `warm_start` is FALSE, reinitialize `beta` to zeros and then update `beta` using the Pathwise Coordinate Descent method.
     - $\beta^{(\lambda)} = \text{Pathwise}(x, y, \lambda, \beta, \text{tol})$, where $\beta^{(\lambda)}$ denotes the coefficients vector obtained for a given λ.
7. **Returning Results:**
   - Return a list containing the sequence of lambda values, the list of `beta` coefficients for each lambda, and the column means and scales used for standardization.

In mathematical terms, this function configures and executes a series of optimization problems, adjusting the model's complexity by varying λ, the regularization strength. The function standardizes the input features to ensure consistent regularization effects across them and utilizes the Pathwise Coordinate Descent method to efficiently find sparse solutions in the context of LASSO regularization.

```r
LASSO = function(x, y, lambda, tol = 0.01, warm_start = FALSE, include_zero_lambda = TRUE){
  # If include_zero_lambda is TRUE and lambda does not already include 0, add 0 to the lambda sequence
  if (include_zero_lambda && sum(lambda == 0) == 0) {
    lambda = c(lambda, 0)
  }

  # Calculate column means of X (feature matrix) and standardize features
  colmean = colMeans(x) #  _j for each feature j
  colscale = c() # Initialize vector to store scaling factors ( _j for each feature j)
  for (i in 1:ncol(x)) {
    x[, i] = x[, i] - colmean[i] # Standardize: X_ij = X_ij -  _j
    colscale = c(colscale, sqrt(sum(x[, i] * x[, i]))) # Calculate  _j for feature j
    x[, i] = x[, i] / sqrt(sum(x[, i] * x[, i])) # Scale: X_ij = X_ij /  _j
  }

  # Add intercept term to X
  x = cbind(rep(1, nrow(x)), x) # Augment X with a column of 1s for intercept  _0
  beta = matrix(rep(0, ncol(x))) # Initialize  vector (model coefficients) with zeros

  beta_list = list() # Initialize list to store  vectors for different  values

  # Single lambda case: perform pathwise coordinate descent and store the result
  if (length(lambda) == 1) {
```

```
    beta <- Pathwise(x, y, lambda[1], beta, tol)$beta_temp
    beta_list[[paste("beta <- lambda:", lambda[k])]] = beta
    return(list(lambda = lambda, beta = beta_list, colmean = colmean, colscale = colscale))
  }
  # Multiple lambda case: sort lambda in decreasing order and iterate over it
  else{
    lambda = sort(lambda, decreasing = TRUE)
    for (k in 1:length(lambda)) {
      if (warm_start)
        beta = Pathwise(x, y, lambda[k], beta, tol)$beta_temp # Use previous  as starting point if warm
      else{
        beta = matrix(rep(0, ncol(x))) # Reset  to zeros if not using warm_start
        beta = Pathwise(x, y, lambda[k], beta, tol)$beta_temp
      }
      beta_list[[paste("beta <- lambda:", lambda[k])]] = beta # Store  vector for current
    }
    return(list(lambda = lambda, beta = beta_list, colmean = colmean, colscale = colscale)) # Return  s
  }
}
```

Criterion to select lambda

This code defines two primary functions and an example score function related to evaluating the performance of predictive models. To describe this code using notations, let's break down each function:

1. **RMSE (Root Mean Square Error) Score Function**

   This function calculates the root mean square error, a common measure for the differences between values predicted by a model and the values actually observed. The function is defined as follows:

$$\text{rmse\_score}(\mathbf{y}, \hat{\mathbf{y}}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

   - Where:
     - $\mathbf{y}$ represents the vector of true values ($true\_y$),
     - $\hat{\mathbf{y}}$ represents the vector of predicted values ($predict\_y$),
     - n is the number of observations ($length(true\_y)$).

2. **Criterion Function**

   This function iterates over a list of predictions, calculates a score for each using the `rmse_score` function, and returns a list of scores. The function can be expressed in a more general form as:

$$\text{criterion}(\mathbf{y}, \{\hat{\mathbf{y}}_\mathbf{1}, \hat{\mathbf{y}}_\mathbf{2}, ..., \hat{\mathbf{y}}_\mathbf{k}\}, \text{score}) = \{\text{score}(\mathbf{y}, \hat{\mathbf{y}}_\mathbf{1}), \text{score}(\mathbf{y}, \hat{\mathbf{y}}_\mathbf{2}), ..., \text{score}(\mathbf{y}, \hat{\mathbf{y}}_\mathbf{k})\}$$

   - Where:
     - $\mathbf{y}$ is the vector of true values,
     - $\{\hat{\mathbf{y}}_\mathbf{1}, \hat{\mathbf{y}}_\mathbf{2}, ..., \hat{\mathbf{y}}_\mathbf{k}\}$ is the list of prediction vectors,
     - score is the scoring function used to evaluate predictions, which in this example is `rmse_score`.

3. **Accuracy Score Function**

   This function calculates the accuracy, the proportion of true results (both true positives and true negatives) among the total number of cases examined. It is defined as follows:

$$\text{accuracy}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}(y_i = \hat{y}_i)$$

- Where:
  - $\mathbf{y}$ represents the vector of true values,
  - $\hat{\mathbf{y}}$ represents the vector of predicted values,
  - $\mathbf{1}(\cdot)$ is the indicator function, equal to 1 when $y_i = \hat{y}_i$ and 0 otherwise,
  - n is the number of observations.

These notations provide a mathematical representation of the R functions described in the code, translating programming constructs into mathematical formulas that describe the operations performed by the code.

```
rmse_score = function(true_y, predict_y){
  return(sqrt(sum((true_y - predict_y) ^ 2) / length(true_y)))
}

criterion = function(true_y, predict_y_list, score){
  result = list()
  for (i in names(predict_y_list)) {
    predict_y = predict_y_list[[i]]
    result[[i]] = rmse_score(true_y, predict_y)
  }
  return(result)
}

#### a score function example --- Accuracy:

accuracy = function(true_y, predict_y){
  return(sum(true_y == predict_y) / length(true_y))
}


predict = function(model, x){

  # use the scale factors for training data to standarize predicting data
  beta_list = model$beta
  colmean = model$colmean
  colscale = model$colscale
  predict_y_list = list()
  for (i in 1:ncol(x)) {
    x[, i] = x[, i] - colmean[i]
    x[, i] = x[, i] / colscale[i]
  }

  # add intercept
  x = cbind(rep(1, nrow(x)), x)

  # calculate the predicting reponse variable for each beta (or lambda)
  for (i in names(beta_list)) {
    # "i" is the value of lambda
    predict_y = 1 / (1 + exp(-x %*% beta_list[[i]]))
    predict_y[predict_y < 0.5] = 0
    predict_y[predict_y >= 0.5] = 1
```

```
    predict_y_list[[i]] = predict_y
  }

  return(predict_y_list)
}
```

Cross Validation

1. **Cross-Validation Loop**

   For each fold i from 1 to $k-1$:

   - Split D into training set $D_{\text{train}}^{(i)}$ and test set $D_{\text{test}}^{(i)}$.
   - Prepare feature matrices $X_{\text{train}}^{(i)}$ and $X_{\text{test}}^{(i)}$, and target vectors $y_{\text{train}}^{(i)}$ and $y_{\text{test}}^{(i)}$.

2. **Model Training and Prediction**

   Train a logistic regression model with LASSO regularization on $D_{\text{train}}^{(i)}$, using a specified sequence of λ values for regularization strength. Predict outcomes for $D_{\text{test}}^{(i)}$.

3. **Evaluation Metrics Calculation**

   - **RMSE** for fold i:

   $$\text{RMSE}^{(i)} = \sqrt{\frac{1}{n_{\text{test}}} \sum_{j=1}^{n_{\text{test}}} (\hat{y}_j - y_j)^2}$$

   - **Accuracy** for fold i:

   $$\text{Accuracy}^{(i)} = \frac{TP + TN}{n_{\text{test}}}$$

   where TP and TN are the counts of true positives and true negatives, respectively, and n_{test} is the size of the test set.
   - **AUC** for fold i is calculated using the ROCR package, comparing the predicted probabilities against the true binary outcomes to evaluate the model's discriminative ability.

4. **Return Results**

   Return a list containing vectors of RMSE, accuracy, and AUC values for the $k-1$ folds.

Note: The provided code has a logical discrepancy as it omits the last fold from evaluation and initializes metrics arrays with length $k-1$ instead of k. This may not fully utilize the provided data for cross-validation and could be an area for correction.

```
CrossValidation = function(data, k = 10) {
  # Create k folds from the data, stratified by the 'diagnosis' variable
  folds = createFolds(data$diagnosis, k = k, list = TRUE)

  # Initialize vectors to store the Root Mean Square Error (RMSE)
  #, accuracy, and Area Under the Curve (AUC) for each fold
  rmse = rep(NA, k - 1)
  accuracy = rep(NA, k - 1)
  auc = rep(NA, k - 1)

  # Iterate over each fold, leaving one out as test set in each iteration
  for (fold in 1:(length(folds) - 1)) {
```

```
    # Define the test set for the current fold and the training set as the rest
    test = data[folds[[fold]],]
    train = data[-folds[[fold]],]

    # Prepare the training data: select features
    # (excluding 'id' and 'diagnosis') and convert to matrix
    x_train = train |> select(-id, -diagnosis) |> as.matrix()
    # Prepare the training labels: binary encoding of 'diagnosis' (M = 1, B = 0)
    y_train <- matrix(rep(0, nrow(train)))
    y_train[train$diagnosis == "M"] = 1

    # Prepare the test data similarly to training data
    x_test = test |> select(-id, -diagnosis) |> as.matrix()
    y_test <- matrix(rep(0, nrow(test)))
    y_test[test$diagnosis == "M"] = 1

    # Train the LASSO model on the training data
    train_model = LASSO(x_train, y_train,
                        lambda = seq(exp(-8), exp(-7), exp(-9)),
                        tol = 0.01, warm_start = TRUE,
                        include_zero_lambda = FALSE)

    # Make predictions on the test set
    predict_result = predict(train_model, x_test)

    # Calculate and store the RMSE for the current fold
    error = c(predict_result[[1]]) - y_test
    rmse[fold] = sqrt(mean(error^2))

    # Calculate and store the accuracy for the current fold
    tp = sum(ifelse(c(predict_result[[1]]) == 1 & y_test == 1, 1, 0))
    # True Positives
    tn = sum(ifelse(c(predict_result[[1]]) == 0 & y_test == 0, 1, 0))
    # True Negatives
    fp = sum(ifelse(c(predict_result[[1]]) == 1 & y_test == 0, 1, 0))
    # False Positives
    fn = sum(ifelse(c(predict_result[[1]]) == 0 & y_test == 1, 1, 0))
    # False Negatives

    accuracy[fold] = (tp + tn) / length(y_test)

    # Calculate and store the AUC for the current fold
    pred = ROCR::prediction(c(predict_result[[1]]), y_test)
    auc_perf = ROCR::performance(pred, "measure" = "auc")
    auc[fold] = auc_perf@y.values[[1]]
  }

  # Return a list containing the computed RMSE, accuracy, and AUC for all folds
  return(list("RMSE" = rmse, "Accuracy" = accuracy, "AUC" = auc))
}
```

```
# load data
cancer = read.csv("breast-cancer.csv")
```

```r
x = cancer |> select(-id, -diagnosis) |> as.matrix()

y <-matrix(rep(0,nrow(cancer)))
y[cancer$diagnosis=="M"] = 1
```

```r
# calculate beta_hat
beta = Pathwise(x, y, tol = 0.01)$beta_temp
beta
```

```
##  [1]   -2.54365344    0.36908757   -0.07913149    0.04479890   71.27163517
##  [6]    4.23884754    7.84833165   70.08539219   14.31575292 -105.64397826
```

```r
# check the data in glm
model = glm( y~x, family = "binomial")
model$coefficients[-1]
```

```
##            xradius_mean           xtexture_mean         xperimeter_mean
##              -2.04930490               0.38473434              -0.07151042
##              xarea_mean        xsmoothness_mean       xcompactness_mean
##               0.03979620              76.43227376              -1.46242225
##         xconcavity_mean    xconcave.points_mean          xsymmetry_mean
##               8.46869976              66.82175685              16.27824232
## xfractal_dimension_mean
##             -68.33702689
```

The beta's are fairly close!

```r
set.seed(314)
fit_stats = CrossValidation(cancer, k = 10)
rmse = mean(fit_stats$RMSE)
rmse
```

```
## [1] 0.253325
```

```r
acc = mean(fit_stats$Accuracy)
acc
```

```
## [1] 0.9334075
```

```r
auc = mean(fit_stats$AUC)
auc
```

```
## [1] 0.9256373
```

High AUC(92.56%) models are particularly valuable in medical diagnostics for prioritizing follow-up tests or interventions for higher-risk individuals.