Newton Extensions

Minghe

Quasi–Newton Methods (Surrogate Hessian Approximations)

SR1 (Symmetric Rank–1 Update)

```r
quasiNewton_SR1 <- function(x0, grad, tol=1e-10, max_iter=500) {
  # x0: initial parameter vector (of length p)
  p <- length(x0)
  x <- x0
  H_inv <- diag(p)  # initial inverse Hessian approximation (identity)
  g <- grad(x)
  i <- 0
  res <- list(x = x, iter = i, grad = g)
  while(i < max_iter && sqrt(sum(g^2)) > tol) {
    s <- - H_inv %*% g  # proposed step
    x_new <- x + s
    g_new <- grad(x_new)
    y_vec <- g_new - g
    d <- s - H_inv %*% y_vec
    denom <- as.numeric(t(d) %*% y_vec)
    if(abs(denom) > 1e-8) {
      H_inv <- H_inv + (d %*% t(d)) / denom
    }
    x <- x_new
    g <- g_new
    i <- i + 1
    res[[i+1]] <- list(x = x, iter = i, grad = g)
  }
  return(res)
}
```

DFP (Davidon–Fletcher–Powell)

```r
quasiNewton_DFP <- function(x0, grad, tol=1e-10, max_iter=500) {
  p <- length(x0)
  x <- x0
  H_inv <- diag(p)
  g <- grad(x)
  i <- 0
  res <- list(x = x, iter = i, grad = g)
  while(i < max_iter && sqrt(sum(g^2)) > tol) {
    s <- - H_inv %*% g
    x_new <- x + s
    g_new <- grad(x_new)
```

```
    y_vec <- g_new - g
    s <- matrix(s, ncol=1)
    y_vec <- matrix(y_vec, ncol=1)
    H_inv <- H_inv + (s %*% t(s))/(t(s) %*% y_vec) - (H_inv %*% y_vec %*% t(y_vec) %*% H_inv)/(t(y_vec)
    x <- x_new
    g <- g_new
    i <- i + 1
    res[[i+1]] <- list(x = x, iter = i, grad = g)
  }
  return(res)
}
```

BFGS (Broyden–Fletcher–Goldfarb–Shanno)

```
quasiNewton_BFGS <- function(x0, grad, tol=1e-10, max_iter=500) {
  p <- length(x0)
  x <- x0
  H_inv <- diag(p)
  g <- grad(x)
  i <- 0
  res <- list(x = x, iter = i, grad = g)
  while(i < max_iter && sqrt(sum(g^2)) > tol) {
    s <- - H_inv %*% g
    x_new <- x + s
    g_new <- grad(x_new)
    y_vec <- g_new - g
    rho <- as.numeric(1 / (t(y_vec) %*% s))
    if(rho <= 0) {
      # Reset if update is not positive-definite
      H_inv <- diag(p)
    } else {
      I <- diag(p)
      H_inv <- (I - rho * s %*% t(y_vec)) %*% H_inv %*% (I - rho * y_vec %*% t(s)) + rho * s %*% t(s)
    }
    x <- x_new
    g <- g_new
    i <- i + 1
    res[[i+1]] <- list(x = x, iter = i, grad = g)
  }
  return(res)
}
```

Coordinate–Wise Optimization

```
coordinateDescent <- function(x0, f, grad, tol=1e-10, max_iter=500) {
  x <- x0
  p <- length(x)
  i <- 0
  while(i < max_iter) {
    x_old <- x
    for(j in 1:p) {
      # Here, we update the jth coordinate while holding others fixed.
```

```r
      # One approach is to compute the partial derivative w.r.t. x[j] and take a small step.
      g <- grad(x)
      step <- 1e-3  # A fixed step size (or use a line search)
      x[j] <- x[j] - step * g[j]
    }
    i <- i + 1
    if(sqrt(sum((x - x_old)^2)) < tol) break
  }
  return(list(x = x, iter = i))
}
```

path-wise coordinate–descent algorithm for the logistic–lasso

```r
# Soft-thresholding operator (for L1 penalty)
soft_threshold <- function(z, gamma) {
  sign(z) * pmax(abs(z) - gamma, 0)
}

# The main function to compute the logistic-lasso path using coordinate descent
logistic_lasso_path <- function(X, y, lambda_min_ratio = 0.001, nlambda = 100,
                                tol = 1e-4, max_iter = 1000) {
  # X is an n x p matrix (predictors) and y is a binary vector (0/1)
  n <- nrow(X)
  p <- ncol(X)

  # Step 1: Compute lambda_max. For logistic regression with intercept,
  # when beta = 0 we have p_i = 1/2, so the subgradient condition yields:
  #   |sum_i x_ij (y_i - 1/2)| <= lambda,  for all j.
  lambda_max <- max(abs(colSums(X * (y - 0.5))))

  # Step 2: Define a decreasing sequence of lambda values from lambda_max to lambda_min.
  lambda_min <- lambda_max * lambda_min_ratio
  lambda_seq <- exp(seq(log(lambda_max), log(lambda_min), length.out = nlambda))

  # Initialize coefficient estimates.
  # For the intercept, a common starting value is the logit of the mean of y.
  beta0 <- log(mean(y) / (1 - mean(y)))
  beta <- rep(0, p)

  # To store the path: one row per lambda.
  intercept_path <- numeric(nlambda)
  beta_path <- matrix(0, nrow = nlambda, ncol = p)

  # Loop over lambda values
  for (l in 1:nlambda) {
    lambda <- lambda_seq[l]

    # Coordinate descent for current lambda.
    converged <- FALSE
    iter <- 0
    while (!converged && iter < max_iter) {
      iter <- iter + 1
```

```r
      # Compute current linear predictor and probabilities.
      eta <- beta0 + X %*% beta
      p_i <- 1 / (1 + exp(-eta))

      # Compute weights and working response.
      w <- p_i * (1 - p_i)
      # To avoid division by very small w, one might add a small constant if needed.
      z <- eta + (y - p_i) / w

      # Update the intercept (unpenalized):
      beta0_new <- sum(w * (z - X %*% beta)) / sum(w)

      # Update each coordinate for beta:
      beta_new <- beta
      for (j in 1:p) {
        # Compute partial residual excluding feature j:
        r_j <- z - beta0_new - X[ , -j, drop = FALSE] %*% beta_new[-j]
        # Compute numerator and denominator for the j-th coordinate.
        num <- sum(w * X[, j] * r_j)
        den <- sum(w * X[, j]^2)
        # Apply soft-thresholding update:
        beta_new[j] <- soft_threshold(num, lambda) / den
      }

      # Check for convergence (e.g. maximum change in coefficients is below tol)
      if (max(abs(beta0_new - beta0), abs(beta_new - beta)) < tol) {
        converged <- TRUE
      }

      beta0 <- beta0_new
      beta <- beta_new
    } # end of coordinate descent for current lambda

    intercept_path[l] <- beta0
    beta_path[l, ] <- beta
  }

  return(list(lambda_seq = lambda_seq, intercept_path = intercept_path,
              beta_path = beta_path))
}

# Suppose X is an n x p matrix and y is an n-vector (with values 0 or 1)
#fit_path <- logistic_lasso_path(X, y, lambda_min_ratio = 0.001, nlambda = 100)
# You can inspect fit_path$lambda_seq, fit_path$intercept_path, and fit_path$beta_path
```