Distributed/ Divide & Conquer

2023-04-03

(1)———————— Using ADMM for KRR ——

Approach

Distributed ADMM (Alternating Direction Method of Multipliers) can be used to solve Kernel Ridge Regression for large scale datasets that cannot be processed on a single machine. The basic idea of distributed ADMM is to split the dataset across multiple machines, perform local updates on each machine, and then communicate the updates across the machines to obtain a global solution. Here's an implementation of distributed ADMM for Kernel Ridge Regression in R.

Math

```
library(parallel)
library(kernlab)

# Generate some sample data
set.seed(123)
n <- 1000
p <- 100
X <- matrix(rnorm(n*p), n, p)
y <- rnorm(n)


# Define the kernel function
kern   <- rbfdot(sigma=1/p)
k      <- kernelMatrix(kern, X)

# Define the distributed ADMM algorithm
distributed_admm <- function(X, y, k, lambda, rho, num_iterations, num_workers) {
  n <- nrow(X)
  p <- ncol(X)

  # Split the dataset into equal-sized chunks
  chunk_size <- ceiling(n / num_workers)
  chunks <- split(1:n, rep(1:num_workers, each = chunk_size)[1:n])

  # Initialize variables
  alpha <- matrix(0, n, 1)
  u <- matrix(0, n, num_workers)
  z <- matrix(0, n, num_workers)
```

```
  # Compute the inverse of the local kernel matrices
  inv_k <- lapply(chunks, function(idx) solve(k[idx,idx] + (rho/2) * diag(chunk_size)))

  # Define the local updates function
  local_update <- function(chunk_idx, chunk, alpha, u, z) {
    # Compute the local residual and update u
    r <- y[chunk_idx] - k[chunk_idx,] %*% alpha - z[chunk_idx,] + u[chunk_idx,]
    u[chunk_idx,] <- u[chunk_idx,] + r

    # Update alpha
    alpha[chunk_idx,] <- inv_k[[chunk_idx]] %*% (k[chunk_idx,] %*% alpha + r - z[chunk_idx,] + u[chunk_

    # Update z
    z[chunk_idx,] <- (rho / (rho + 1)) * (k[chunk_idx,] %*% alpha + u[chunk_idx,])

    return(list(alpha = alpha, u = u, z = z))
  }

  # Run the distributed ADMM algorithm
  for (iter in 1:num_iterations) {
    # Parallelize the local updates across workers
    local_results <- mclapply(seq_along(chunks), function(i) local_update(chunks[[i]], X[chunks[[i]],],

    # Combine the local updates
    alpha <- Reduce(`+`, lapply(local_results, function(result) result$alpha))
    u <- Reduce(`+`, lapply(local_results, function(result) result$u))
    z <- Reduce(`+`, lapply(local_results, function(result) result$z))

    # Apply the global consensus step
    alpha <- alpha / num_workers
    u <- u / num_workers
    z <- z / num_workers
  }

  return(alpha)
}

# Run the distributed ADMM algorithm with 4 workers, lambda = 0.01, rho = 1, and 10 iterations
num_workers <- 4
lambda <- 0.01
rho <- 1
num_iterations <- 10
alpha <- distributed_admm(X, y, k, lambda, rho, num_iterations, num_workers)

# Compute the predictions and mean squared error
y_pred <- k %*% alpha
mse <- mean((y - y_pred)^2)
print(paste("MSE:", mse))
```

In this implementation, we split the dataset into equal-sized chunks and perform local updates on each chunk using the kernel matrix and its inverse. We then combine the local updates by averaging them, and apply a global consensus step to obtain the next iteration. The process is repeated for a fixed number of iterations. Finally, we compute the predictions and mean squared

error using the obtained alpha values.

You can experiment with different values of num_workers, lambda, rho, and num_iterations to optimize the performance of the algorithm for your dataset. Note that the optimal values may depend on the size and characteristics of your dataset, as well as the computing resources available.