Adding Many Sparse Factors to a Flash Object

Jason Willwerscheid

2/17/2018

If you want to add a lot of sparse factors to a flash object, a few tricks are recommended.

As an example, I consider the data in the PLIER package. In their vignette, a set of 628 known genesets are considered. Here, I show how to efficiently obtain a flash object with these genesets added as fixed factors. (Here, "fixed factor" means that the sparsity pattern of the factor is fixed. The nonnull values of the factor are allowed to vary.)

If you just use `flash_add_fixed_f` and then backfit with default parameter settings, you can expect to wait for a very long time. I ran `flash_backfit` for over 48 hours on a MacBook Pro and came nowhere near convergence. (105 iterations were completed. In the final iteration, the objective decreased by 273. By way of comparison, the objective decreased by 438 in iteration 80, 363 in iteration 90, and 315 in iteration 100.)

To speed the process up, I recommend:

1. Using `ebnm_fn = ebnm_pn`, which uses the point-normal prior in the `ebnm` package rather than the more flexible unimodal prior in `ashr`.
2. Optimizing small flash objects, then combining pairs of small objects and optimizing these medium objects, then combining pairs of medium objects and optimizing, and so on. The idea is to provide good initializations at each stage. Factors that are similar and thus difficult to jointly optimize should be combined early on, before a huge flash object needs to be optimized. A good ordering can be obtained using hierarchical clustering. At these intermediate stages, the optimization does not need to be perfect, so `tol` can be set relatively high (I used `tol = 0.3`).
3. Even with the cleverest of initializations, a final optimization of the full flash object cannot be avoided. To accelerate this process, I've rewritten the backfitting algorithm to eliminate some inefficiencies when fitting large objects. Using the new backfitting function speeds up the final fitting process at least tenfold. (Note that this should not be used for smaller flash objects. In tests I ran, the new function only outperformed standard backfitting for flash objects with at least 32 factors.)

Using the following code, I was able to fit all 628 factors with final `tol = 0.01` overnight (the total running time was approximately ten hours).

The setup code is taken from the PLIER vignette.

```r
library(PLIER)
data("bloodCellMarkersIRISDMAP")
data("svmMarkers")
data("canonicalPathways")
data("vacData")

allPaths = combinePaths(bloodCellMarkersIRISDMAP, svmMarkers, canonicalPathways)
cm.genes = commonRows(allPaths, vacData)
vacDataN=rowNorm(vacData)

vacDataCM = t(vacDataN[cm.genes,])
```

Next I set up the data for flash.

```r
library(flashr)
data = flash_set_data(vacDataCM)
```

```
fixed_f = allPaths[cm.genes,]
fixed_f[fixed_f != 0] = NA
```

Finally I fit a flash object using the above recommendations.

```
# Use hierarchical clustering to find similar factors:
hc <- hclust(dist(t(allPaths[cm.genes,]), method="binary"), method="single")

var_type <- "constant"
n_factors <- ncol(fixed_f)
time <- vector("list", ceiling(log2(n_factors)))

# Start with flash objects with 4 factors each (careful -- this will create a very large list):
chunk_size <- 4
chunks <- vector("list", ceiling(n_factors / chunk_size))
begin_time <- Sys.time()
for (i in 1:length(chunks)) {
  message("CHUNK: ", i)
  # Use the clustering results to decide which factors to add next:
  if (4*i > n_factors) {
    next_idx <- hc$order[(4*i-3):n_factors]
  } else {
    next_idx <- hc$order[(4*i-3):(4*i)]
  }
  chunks[[i]] <- flash_add_fixed_f(data, fixed_f[, next_idx])
  chunks[[i]] <- flash_backfit(data, chunks[[i]], var_type=var_type, tol=0.1,
                               ebnm_fn=ebnm_pn, verbose=T, nullcheck=F)
}
end_time <- Sys.time()
t <- 1
time[[t]] <- end_time - begin_time
# backup progress:
saveRDS(chunks, "/Users/willwerscheid/Desktop/chunks.rds")

# Successively combine pairs of flash objects until only one remains:
while(length(chunks) > 1) {
  chunk_size <- min(chunk_size * 2, n_factors)
  message("Forming chunks of size ", chunk_size, "...")
  n_chunks <- ceiling(length(chunks) / 2)
  begin_time <- Sys.time()
  for (i in 1:n_chunks) {
    message("CHUNK:",i)
    if ((2*i) > length(chunks)) { # Odd chunk left over
      chunks[[i]] <- chunks[[2*i - 1]]
    } else {
      chunks[[i]] <- flash_combine(chunks[[2*i-1]], chunks[[2*i]])
      if (chunk_size > 16) {
        chunks[[i]] <- flash_optimize_multiple_fl(data, chunks[[i]], var_type=var_type,
                                                  tol=0.3, ebnm_fn=ebnm_pn, verbose=T)
      } else {
        chunks[[i]] <- flash_backfit(data, chunks[[i]], var_type=var_type, tol=0.3,
                                     ebnm_fn=ebnm_pn, verbose=T, nullcheck=F)
      }
    }
```

```r
  }
  chunks <- chunks[1:n_chunks]
  end_time <- Sys.time()
  t <- t + 1
  time[[t]] <- end_time - begin_time
  saveRDS(chunks, "/Users/willwerscheid/Desktop/chunks.rds")
}

# Do a final refinement of the full flash object:
final_fit <- chunks[[1]]
begin_time <- Sys.time()
final_fit <- flash_optimize_multiple_fl(data, final_fit, var_type=var_type, tol=0.01,
                                         ebnm_fn=ebnm_pn, verbose=T)
end_time <- Sys.time()
t <- t + 1
time[[t]] <- end_time - begin_time
saveRDS(final_fit, "/Users/willwerscheid/Desktop/final_fit.rds")
```

The code for `flash_optimize_multiple_fl` is as follows:

```r
#' @title flash_optimize_multiple_fl
#' @description Optimize multiple loadings and factors
#' @details This function iteratively optimizes a set of loadings and factors.
#' @param data an n by p matrix or a flash data object created using \code{flash_set_data}
#' @param f a fitted flash object to be refined
#' @param kset the indices of factors to be optimized (NULL indicates all factors)
#' @param var_type type of variance structure to assume for residuals
#' @param tol specifies how much objective must change to be considered not converged
#' @param ebnm_fn function to solve the Empirical Bayes Normal Means problem
#' @param ebnm_param parameters to be passed to ebnm_fn when optimizing; defaults set by
#' flash_default_ebnm_param()
#' @param verbose if TRUE various output progress updates will be printed
#' @param msg_after if verbose = TRUE, an update will be printed after every msg_after
#' factor/loadings have been updated
#' @param maxiter maximum number of iterations to perform (in the outer loop)
#' @param maxiter_single_fl if sequential = FALSE, this sets the maximum number of
#' iterations that will be performed on a single factor/loading before moving to the next
#' @param sequential if TRUE, each factor/loading will only be updated once before moving
#' to the next
#' @return an updated flash object
flash_optimize_multiple_fl = function(data, f, kset = NULL,
                                      var_type = c("by_column", "constant"),
                                      tol = 1e-2,
                                      ebnm_fn = ebnm_ash,
                                      ebnm_param = flash_default_ebnm_param(ebnm_fn),
                                      verbose = FALSE,
                                      msg_after = 100,
                                      maxiter = 100,
                                      maxiter_single_fl = 100,
                                      sequential = FALSE) {

  if (is.matrix(data)) {data = flash_set_data(data)}
  if (is.null(kset)) {kset = 1:get_k(f)}
  var_type = match.arg(var_type)
```

```r
Rk = get_Rk(data, f, kset[1])
R2 = get_R2(data, f)
KLobj = sum(unlist(f$KL_l)) + sum(unlist(f$KL_f))
missing = data$missing

F_obj = -Inf
diff = Inf
iteration_diff = Inf

last_k = NULL

outer_iter = 0
fl_changed = TRUE
# Continue while: 1. at least one factor/loading has changed significantly; 2. the
# overall objective has decreased significantly; 3. the maximum number of outer
# iterations has not been reached.
while (fl_changed && (iteration_diff > tol) && (outer_iter < maxiter)) {
  fl_changed = FALSE
  outer_iter = outer_iter + 1
  ks_updated = 0

  if (verbose) {message("Iteration: ", outer_iter)}
  F_obj_old = F_obj

  for (k in kset) {
    inner_iter = 0
    obj_decreased = TRUE

    # Update Rk:
    if (length(kset) > 1 && !is.null(last_k)) {
      Rk = Rk - outer(f$EL[, last_k], f$EF[, last_k]) + outer(f$EL[, k], f$EF[, k])
    }

    # Continue updating a single factor while: 1. sequential = FALSE; 2. the objective
    # has decreased significantly; 3. the maximum number of inner iterations has not
    # been reached.
    while (!sequential && obj_decreased && (inner_iter < maxiter_single_fl)) {
      obj_decreased = FALSE
      inner_iter = inner_iter + 1

      l_old = f$EL[, k]
      f_old = f$EF[, k]
      l2_old = f$EL2[, k]
      f2_old = f$EF2[, k]
      KL_l_old = f$KL_l[[k]]
      KL_f_old = f$KL_f[[k]]

      l_subset = which(!f$fixl[, k])
      f_subset = which(!f$fixf[, k])

      # Update tau:
      tau = compute_precision(R2, missing, var_type)
      # Update f:
      if (length(f_subset) > 0) {
        s2 = 1/(t(f$EL2[, k]) %*% tau[, f_subset, drop=F])
        if (any(is.finite(s2))) { # check that some values are finite before proceeding
          x = (t(f$EL[, k]) %*% (Rk[, f_subset, drop=F] * tau[, f_subset, drop=F])) * s2
          ebnm_f = ebnm_fn(x, sqrt(s2), ebnm_param)
          f$EF[f_subset, k] = ebnm_f$postmean
          f$EF2[f_subset, k] = ebnm_f$postmean2
          f$gf[[k]] = ebnm_f$fitted_g
          f$penloglik_f[[k]] = ebnm_f$penloglik
          f$KL_f[[k]] = (ebnm_f$penloglik - NM_posterior_e_loglik(x, sqrt(s2),
                                              ebnm_f$postmean, ebnm_f$postmean2))
        }
      }

      # Update l:
      if (length(l_subset) > 0) {
        s2 = 1/(tau[l_subset, , drop=F] %*% f$EF2[, k])
        if (any(is.finite(s2))) { # check that some values are finite before proceeding
          x = ((Rk[l_subset, , drop=F] * tau[l_subset, , drop=F]) %*% f$EF[, k]) * s2
          ebnm_l = ebnm_fn(x, sqrt(s2), ebnm_param)
          f$EL[l_subset, k] = ebnm_l$postmean
          f$EL2[l_subset, k] = ebnm_l$postmean2
          f$gl[[k]] = ebnm_l$fitted_g
          f$penloglik_l[[k]] = ebnm_l$penloglik
          f$KL_l[[k]] = (ebnm_l$penloglik - NM_posterior_e_loglik(x, sqrt(s2),
                                              ebnm_l$postmean, ebnm_l$postmean2))
        }
      }

      # Update R2:
      R2 = R2 + 2 * outer(l_old, f_old) * Rk - outer(l2_old, f2_old)
      R2 = R2 - 2 * outer(f$EL[, k], f$EF[, k]) * Rk + outer(f$EL2[, k], f$EF2[, k])

      # Update objective:
      KLobj = KLobj - KL_l_old - KL_f_old + f$KL_l[[k]] + f$KL_f[[k]]
      Fnew = KLobj + e_loglik_from_R2_and_tau(R2, tau, missing)
      diff = Fnew - F_obj
      F_obj = Fnew
      if (diff > tol) {
        obj_decreased = TRUE
        # The objective isn't computed until the first update of the first factor/
        # loading, so diff = Inf on the first update. But this first update shouldn't
        # trigger a continuation of the entire outer loop.
        if (diff < Inf) {
          fl_changed = TRUE
        }
      }
    } # end of inner loop

    last_k = k
    ks_updated = ks_updated + 1
    if (verbose && (ks_updated %% msg_after == 0)) {
          message("  ", ks_updated, " factor/loadings updated")
        }
      }

      if (verbose) {message("Objective: ", F_obj)}
      iteration_diff = F_obj - F_obj_old
    } # end of outer loop

    # Set remaining variables in f:
    f$tau = tau

    for (k in kset) {
      f$ebnm_param_f[[k]] = ebnm_param
      f$ebnm_param_l[[k]] = ebnm_param
    }

    # TODO: write wrapper function with nullcheck

    return(f)
}

# Compute the expected log-likelihood (at non-missing locations) based on expected squared
# residuals and tau:
e_loglik_from_R2_and_tau = function(R2, tau, missing){
  -0.5 * sum(log((2*pi) / tau[!missing]) + tau[!missing] * R2[!missing])
}
```