SBF vignette

Amal Thomas

Contents

1 Background

Joint matrix factorization facilitates the comparison of expression profiles from different species without using gene mapping. Transforming gene expression profiles into reduced eigengene space using singular value decomposition (SVD) has been shown to capture meaningful biological information (Alter, Brown, and Botstein 2000). Tamayo et al. (2007) used a non-negative matrix factorization approach to learn a low-dimensional approximation of the microarray expression datasets and used the reduced space for comparisons. Matrix factorization-based methods are commonly used for gene expression analysis (Alter, Brown, and Botstein 2000; Tamayo et al. 2007). An orthology independent matrix factorization framework based on generalized singular value decomposition [GSVD; Van Loan (1976)] was used by Alter, Brown, and Botstein (2003) to compare gene-expression profiles from two species. This framework was later extended to develop higher-order generalized singular value decomposition (HO GSVD) to analyze data from more than two species (Ponnapalli et al. 2011).

This study developed a joint diagonalization approach called approximate shared basis factorization (A-SBF) for cross-species expression comparisons. This approach extends the exact factorization approach we developed called shared basis factorization (SBF). We discuss the details of the two methods in the following sections.

2 Shared basis factorization

Consider a set of real matrices $D_i \in \mathbb{R}^{m_i \times k}$ $(i = 1, \ldots, N)$ with full column rank. We define shared basis factorization (SBF) as

$$\begin{aligned} D_1 &= U_1 \Delta_1 V^T, \\ D_2 &= U_2 \Delta_2 V^T, \\ &\vdots \\ D_N &= U_N \Delta_N V^T. \end{aligned}$$

Here each $U_i \in \mathbb{R}^{m_i \times k}$ is a dataset-specific left basis matrix, each $\Delta_i \in \mathbb{R}^{k \times k}$ is a diagonal matrix with positive values δ_{ik}, and V is a square invertible matrix.

2.1 Estimating the shared right basis matrix

Let M be the scaled sum of the $D_i^T D_i$. We define M is defined as

$$M = \frac{\sum_{i=1}^{N} D_i^T D_i / w_i}{\alpha}.$$

The scaling factor w_i is the total variance explained by the column vectors of D_i, and α is the inverse sum of the total variance of D_i, for $i = 1 \cdots N$. The weights w_i and α are defined as

$$\begin{aligned} w_i &= \sum_{j=1}^{k} \sigma_{jj}^{2\ (i)} \text{ and} \\ \alpha &= \sum_{i=1}^{N} \frac{1}{\sum_{j=1}^{k} \sigma_{jj}^{2\ (i)}}. \end{aligned}$$

Here $\sum_{j=1}^{k} \sigma_{jj}^{2\ (i)} = \operatorname{tr}(D_i^T D_i) = \operatorname{tr}(A_i)$. Using the w_i and α, individual $D_i^T D_i$ are standardized. If all the variances are equal, M becomes the arithmetic mean of the sum of $D_i^T D_i$. The shared right basis matrix V is then determined from the eigenvalue decomposition of M, where $M = V \Theta V^T$. The shared right basis matrix V is an orthogonal matrix as M is symmetric. Given V, we compute U_i and Δ_i by solving the linear system $D_i V = U_i \Delta_i = L_i$. By normalizing the columns of L_i, we have $\delta_{ik} = \|l_{ik}\|$ and $\Delta_i = \operatorname{diag}(\delta_{i1}, \ldots, \delta_{ik})$.

3 Approximate shared basis factorization

Consider a set of matrices $D_i \in \mathbb{R}^{m_i \times k}$ $(i = 1, \ldots, N)$, each with full column rank. We define approximate shared basis factorization (A-SBF) as

$$\begin{aligned} D_1 &= U_1 \Delta_1 V^T + \epsilon_1, \\ D_2 &= U_2 \Delta_2 V^T + \epsilon_2, \\ &\vdots \\ D_N &= U_N \Delta_N V^T + \epsilon_N. \end{aligned}$$

Each $U_i \in \mathbb{R}^{m_i \times k}$ is a species-specific left basis matrix with **orthonormal** columns (eigengenes), $\Delta_i \in \mathbb{R}^{k \times k}$ is a diagonal matrix with positive values Δ_{ik} and V is a non singular square matrix. The right basis matrix V is identical in all the N matrix factorizations and defines the common space shared by all species. We estimate the factorization such that the estimated V is closest to that in the exact decomposition and by minimizing the total decomposition error $\sum_{i=1}^{N} \epsilon_i = \sum_{i=1}^{N} \|D_i - U_i \Delta_i V^T\|^2{}_F$.

4 Usage cases

4.1 SBF examples

```
# load SBF package
library(SBF)
```

Let us create some random matrices using the `createRandomMatrices` function from the SBF package. We will create four matrices, each with three columns with rows varying from 4 to 6.

```
set.seed(1231)
mymat <- createRandomMatrices(n = 4, ncols = 3, nrows = 4:6)
sapply(mymat, dim)
#>      mat1 mat2 mat3 mat4
#> [1,]    5    6    4    5
#> [2,]    3    3    3    3
```

Rank of each of this matrices

```
sapply(mymat, function(x) {
  qr(x)$rank
  })
#> mat1 mat2 mat3 mat4
#>    3    3    3    3
```

Let us compute SBF using different approaches.

- Estimate V using sum of $D_i^T D_i / N$
- Estimate V using sum of $D_i^T D_i / N$ with inverse variance weighting
- Estimate V using inter-sample correlation

```
sbf <- SBF(matrix_list = mymat)
sbf_inv <- SBF(matrix_list = mymat, weighted = TRUE)
sbf_cor <- SBF(matrix_list = mymat, transform_matrix = TRUE)
```

When D_i's are transformed to compute inter-sample correlation, we do not need to scale it using inverse-variance weighting anymore. We recommend using inverse variance weights, giving a more robust estimate of V when noisy datasets are present. We estimate V using inter-sample correlation when dealing with gene expression data sets.

`?SBF` help function shows all arguments for the SBF function. Let us inspect the output of the `SBF` call.

```
names(sbf)
#> [1] "v"      "lambda" "u"      "delta"  "m"
```

`sbf$u`, `sbf$v`, and `sbf$delta` correspond to the estimated left basis matrix, shared right basis matrix, and diagonal matrices.

The estimated $V \in R^{k \times k}$ has a dimension of $k \times k$, where k is the number of columns in D_i.

```
sbf$v
#>           [,1]       [,2]       [,3]
#> [1,] 0.4793022  0.8669998  0.1363110
#> [2,] 0.7027353 -0.2860780 -0.6514004
#> [3,] 0.5257684 -0.4080082  0.7463892
```

The delta values for each matrix for the three cases are shown below.

```
printDelta <- function(l) {
  for (eachmat in names(l$delta)) {
  cat(eachmat, ":", l$delta[[eachmat]], "\n")
  }
}
cat("sbf\n");printDelta(sbf)
#> sbf
#> mat1 : 205.4915 29.6746 71.43295
#> mat2 : 206.5816 71.72548 55.682
#> mat3 : 189.9136 52.6758 42.36825
#> mat4 : 192.6911 80.22868 58.57913
cat("sbf_inv\n");printDelta(sbf_inv)
#> sbf_inv
#> mat1 : 205.5109 22.4888 73.95623
#> mat2 : 206.5963 77.00352 48.05638
#> mat3 : 189.8719 58.60394 33.92988
#> mat4 : 192.6942 72.41955 67.98802
cat("sbf_cor\n");printDelta(sbf_cor)
#> sbf_cor
#> mat1 : 200.4197 44.25134 77.99852
#> mat2 : 199.8621 80.34494 67.23723
#> mat3 : 176.1738 76.95449 60.64485
#> mat4 : 185.4579 67.51833 89.69184
```

The $V \in R^{k \times k}$ estimated in SBF is orthogonal. So $V^T V = V V^T = I$.

```
zapsmall(t(sbf$v) %*% sbf$v)
#>      [,1] [,2] [,3]
#> [1,]    1    0    0
#> [2,]    0    1    0
#> [3,]    0    0    1
```

The estimated V is an invertible matrix.

```
qr(sbf$v)$rank
#> [1] 3
```

The U_i matrices estimated in the SBF do not have orthonormal columns. Let us explore that.

```
sapply(sbf$u, dim)
#>      mat1 mat2 mat3 mat4
#> [1,]    5    6    4    5
#> [2,]    3    3    3    3
```

Let us take the first matrix $U_i \in R^{m_i \times k}$ to check this. For this matrix, $U_i^T U_i$ will be $k \times k$ matrix where $k = 3$.

```
t(sbf$u[[names(sbf$u)[1]]]) %*% sbf$u[[names(sbf$u)[1]]]
#>             [,1]        [,2]       [,3]
#> [1,]  1.00000000 -0.07071457  0.1405487
#> [2,] -0.07071457  1.00000000 -0.6201468
#> [3,]  0.14054867 -0.62014676  1.0000000
```

The estimated M matrix is stored `sbf$m` and `sbf$lambda` gives the eigenvalues in the eigenvalue decomposition ($M = V\Theta V^T$).

```
sbf$lambda
#> [1] 39524.940  3809.127  3357.434
```

SBF is an exact factorization. Let compute the factorization error for the three cases using `calcDecompError` function.

```
calcDecompError(mymat, sbf$u, sbf$delta, sbf$v)
#> [1] 1.851693e-26
calcDecompError(mymat, sbf_inv$u, sbf_inv$delta, sbf_inv$v)
#> [1] 1.894292e-26
calcDecompError(mymat, sbf_cor$u, sbf_cor$delta, sbf_cor$v)
#> [1] 2.835482e-26
```

The errors are close to zero in all three cases.

4.1.1 Adding new dataset

The total column variance of matrix 1-4 in `mymat` is approximately in the same range.

```
sapply(mymat, function(x) sum(diag(cov(x))))
#>    mat1    mat2    mat3    mat4
#> 2076.80 2273.50 2375.25 2860.40
```

Now, let us create two new matrix lists containing the `mymat`. We will add a dataset with a similar variance to the first list and a high variance to the second.

```
mat5 <- matrix(c(130, 183, 62, 97, 147, 94, 102, 192, 19), byrow = T,
                    nrow = 3, ncol = 3)
mat5_highvar <- matrix(c(406, 319, 388, 292, 473, 287, 390, 533, 452),
                       byrow = T, nrow = 3, ncol = 3)

mymat_new <- mymat
mymat_new[["mat5"]] <- mat5
sapply(mymat_new, function(x) sum(diag(cov(x))))
#>     mat1     mat2     mat3     mat4     mat5
#> 2076.800 2273.500 2375.250 2860.400 2299.667
mymat_new_noisy <- mymat
mymat_new_noisy[["mat5"]] <- mat5_highvar
sapply(mymat_new_noisy, function(x) sum(diag(cov(x))))
#>     mat1     mat2     mat3     mat4     mat5
#>  2076.80  2273.50  2375.25  2860.40 22915.00
```

Let us compute SBF with the new datasets.

```
sbf_new <- SBF(matrix_list = mymat_new)
sbf_inv_new <- SBF(matrix_list = mymat_new, weighted = TRUE)


sbf_new_noisy <- SBF(matrix_list = mymat_new_noisy)
sbf_inv_new_noisy <- SBF(matrix_list = mymat_new_noisy, weighted = TRUE)
```

Let us take the newly estimated values U_i, Δ_i, and V for the four initial matrices in `mymat`. We will then compare the decomposition error for the two cases with and without inverse variance weighting.

```
e1 <- calcDecompError(mymat, sbf_new$u[1:4], sbf_new$delta[1:4], sbf_new$v)
e2 <- calcDecompError(mymat, sbf_new_noisy$u[1:4], sbf_new_noisy$delta[1:4],
                      sbf_new_noisy$v)
e2 / e1
#> [1] 3.887268
```

```
e3 <- calcDecompError(mymat, sbf_inv_new$u[1:4], sbf_inv_new$delta[1:4],
                      sbf_inv_new$v)
e4 <- calcDecompError(mymat, sbf_inv_new_noisy$u[1:4],
                      sbf_inv_new_noisy$delta[1:4], sbf_inv_new_noisy$v)
e4 / e3
#> [1] 1.657059
```

With inverse variance weighting, the deviation is smaller.

4.2 A-SBF examples

```
set.seed(1231)
mymat <- createRandomMatrices(n = 4, ncols = 3, nrows = 4:6)
sapply(mymat, dim)
#>      mat1 mat2 mat3 mat4
#> [1,]    5    6    4    5
#> [2,]    3    3    3    3
```

Now let us compute Approximate-SBF for the same datasets.

- A-SBF
- A-SBF with inverse variance weighting
- A-SBF with inter-sample correlation

```
asbf <- SBF(matrix_list = mymat, approximate = TRUE)
asbf_inv <- SBF(matrix_list = mymat, weighted = TRUE, approximate = TRUE)
asbf_cor <- SBF(matrix_list = mymat, approximate = TRUE, transform_matrix = TRUE)
```

```
names(asbf)
#> [1] "v"       "lambda"  "u"       "u_ortho" "delta"   "m"       "error"
```

A-SBF is not an exact factorization. A-SBF output has two additional values. `asbf$u_ortho` is the left basis matrix with orthonormal columns and `asbf$error` gives the decomposition error.

```
asbf$error
#> [1] 2329.73
asbf_inv$error
#> [1] 1651.901
asbf_cor$error
#> [1] 14045.99
```

The same error can also be computed using the `calcDecompError` function.

```
calcDecompError(mymat, asbf$u_ortho, asbf$delta, asbf$v)
#> [1] 2329.73
calcDecompError(mymat, asbf_inv$u_ortho, asbf_inv$delta, asbf_inv$v)
#> [1] 1651.901
calcDecompError(mymat, asbf_cor$u_ortho, asbf_cor$delta, asbf_cor$v)
#> [1] 14045.99
```

In A-SBF factorization, U_i has orthonormal columns, and V is orthogonal.

```
zapsmall(t(asbf$u_ortho[[names(asbf$u_ortho)[1]]]) %*%
           asbf$u_ortho[[names(asbf$u_ortho)[1]]])
#>      [,1] [,2] [,3]
#> [1,]    1    0    0
#> [2,]    0    1    0
#> [3,]    0    0    1
```

```
zapsmall(t(asbf$v) %*% asbf$v)
#>      [,1] [,2] [,3]
#> [1,]    1    0    0
#> [2,]    0    1    0
#> [3,]    0    0    1
```

5 Reduce A-SBF factorization error

We will use the following propositions to develop an iterative approach to minimize the factorization error.

5.1 Proposition I

Let $A \in \mathbb{R}^{m \times n}$ with rank n and SVD of $AQ^T = X\Sigma Y^T$. Among all matrices $B \in \mathbb{R}^{m \times n}$ with orthonormal columns, the Forbenius norm $\|A - BQ\|_F^2$ is minimized when $B = XY^T$.

$$\begin{aligned} \min \|A - BQ\|_F^2 &= \operatorname{tr}\left(A - BQ\right)^T \left(A - BQ\right) \\ &= \operatorname{tr}(A^T A) + \operatorname{tr}(Q^T Q) - 2\operatorname{tr}(AQ^T B^T). \end{aligned}$$

This is equivalent to maximizing $\operatorname{tr}(AQ^T B^T)$. Let SVD of $AQ^T = X\Sigma Y^T$, where $X \in R^{m \times m}$ is left singular matrix, $\Sigma \in R^{m \times n} = \begin{bmatrix} \Sigma_n \\ 0 \end{bmatrix}$ with $\Sigma_n = \operatorname{diag}(\sigma_1, \ldots, \sigma_n)$, and $V \in R^{m \times n}$ is the right singular matrix. Let $Z = Y^T B^T X$. Now Z is a rectangular matrix with orthonormal rows ($ZZ^T = I$) as $B^T B = I$. Now we have,

$$\begin{aligned} \operatorname{tr}(AQ^T B^T) &= \operatorname{tr}(X\Sigma Y^T B^T) = \operatorname{tr}(Y^T B^T X \Sigma) = \operatorname{tr}(Z\Sigma) \\ &= \sum_{i=1}^{n} z_{ii}\sigma_i \leq \sum_{i=1}^{n} \sigma_i. \end{aligned}$$

The upper bound is obtained when $Z = I$ and thus $B = XY^T$.

5.2 Proposition II

Let $A \in \mathbb{R}^{m \times n}$ with rank n and SVD of $A^T B = X\Sigma Y^T$. Among all orthogonal matrices Q, the Forbenius norm $\|A - BQ^T\|_F^2$ is minimized when $Q = XY^T$.

$$\begin{aligned} \min \|A - BQ^T\|_F^2 &= \operatorname{tr}\left(A - BQ^T\right)^T \left(A - BQ^T\right) \\ &= \operatorname{tr}(A^T A) + \operatorname{tr}(B^T B) - 2\operatorname{tr}(A^T BQ^T). \end{aligned}$$

This is equivalent to maximizing $\operatorname{tr}(A^T BQ^T)$. Let SVD of $A^T B = X\Sigma Y^T$, where $\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_n)$. We define an orthogonal matrix \$Z = Yˆ T Q X \$. Now we have,

$$\begin{aligned} \operatorname{tr}(A^T BQ^T) &= \operatorname{tr}(X\Sigma Y^T Q^T) = \operatorname{tr}(Y^T Q^T X \Sigma) = \operatorname{tr}(Z\Sigma) \\ &= \sum_{i=1}^{n} z_{ii}\sigma_i \leq \sum_{i=1}^{p} \sigma_i. \end{aligned}$$

The upper bound is obtained when $Z = I$ and thus $Q = XY^T$.

5.3 Proposition III

For a set of n matrices $A_1, A_2, \ldots, A_n$, where $A_i \in \mathbb{R}^{k\times k}$, the orthogonal matrix $Q \in \mathbb{R}^{k\times k}$ minimizing the Forbenius norm $\sum_{i=1}^{n} \|A_i - Q\|_F^2$ is given by $Q = XY^T$, where SVD of $\sum_{i=1}^{n} A_i = X\Sigma Y^T$.

Let us first consider $i = 1$ case. We want to minimize the objective function:

$$\mathcal{F}(Q) = \|A_1 - Q\|_F^2 \text{ subject to } Q^T Q = I.$$

The objective function can be reformulated as

$$\mathcal{F}(Q) = \|A_1 - Q\|_F^2 = \operatorname{tr}\left(A_1^T A_1 - 2A_1^T Q + Q^T Q\right).$$

Let Φ be a matrix with Lagrange multipliers for constraint $Q^T Q - I = 0$. The Lagrange $\mathcal{L}$ is

$$\mathcal{L}(Q) = \operatorname{tr}\left(A_1^T A_1 - 2A_1^T Q + Q^T Q\right) + \operatorname{tr}\left(\Phi(Q^T Q - I)\right).$$

The partial derivative of $\mathcal{L}$ with respect to Q gives

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial Q} &= -2A_1 + Q(\Phi^T + \Phi) \\ A_1 &= Q\left(\frac{\Phi^T + \Phi}{2}\right) \\ Q &= A_1\left(\frac{\Phi^T + \Phi}{2}\right)^{-1}. \end{aligned}$$

Given that the Q is orthogonal, we have

$$\begin{aligned} A_1^T A_1 &= \left(\frac{\Phi^T + \Phi}{2}\right)^2 \\ \left(\frac{\Phi^T + \Phi}{2}\right) &= (A_1^T A_1)^{1/2}. \end{aligned}$$

Thus for $i = 1$, $Q = A_1(A_1^T A_1)^{-1/2}$. Now for $i = 2$, we have

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial Q} &= -2A_1 - 2A_2 + Q(\Phi^T + \Phi) \\ A_1 + A_2 &= Q\left(\frac{\Phi^T + \Phi}{2}\right) \\ Q &= (A_1 + A_2)\left(\frac{\Phi^T + \Phi}{2}\right)^{-1} \\ &= (A_1 + A_2)\left((A_1 + A_2)^T(A_1 + A_2)\right)^{-1/2}. \end{aligned}$$

Now for $i = n$, we have

$$\begin{aligned} Q &= M(M^T M)^{-1/2} \text{ where } M = \sum_{i=1}^{n} A_i = X\Sigma Y^T \\ &= X\Sigma Y^T (Y\Sigma^2 Y^T)^{-1/2} \\ &= XY^T. \end{aligned}$$

5.4 Iterative update

Using these three propositions, we iteratively update U_i, Δ_i, and V. The steps of the iterative algorithm are shown below.

```
1. Input: $D_i, i = 1 \dots N$
2. Output: $U_i, \Delta_i$, and $V$, where $U_i^T U_i = I$ and $V^T V = V V^T = I$
3. Initialize $U_i^k$, $\Delta_i^k$, and $V^k$. Compute $\epsilon^{k,k,k} = \sum_{i=1}^N \|D_i - U_i^k \Delta_i^k V^{k^T}\|^2{}_F$
4. Update $\mathbf{U_i^{k+1}}$:
     $U_i^{k+1} = ZY^T$, where SVD of $D_i V^k \Delta_i^k = Z \Sigma Y^T$. Compute $\epsilon^{k+1,k,k}$
     If $\epsilon^{k+1,k,k} < \epsilon^{k,k,k}$:
       $U_i \leftarrow U_i^{k+1}$.
5. Update $\mathbf{\Delta_i^{k+1}}$:
     $\mathbf{\Delta_i^{k+1}} = \text{diag}((U_i^{k+1})^T D_i V^k)$. Compute $\epsilon^{k+1,k+1,k}$
     If $\epsilon^{k+1,k+1,k} < \epsilon^{k+1,k,k}$:
       $\Delta_i \leftarrow \Delta_i^{k+1}$
6. Update $\mathbf{V^{k+1}}$:
     $V^{k+1} = MQ^T$, where SVD of $\sum_i^N D_i^T U_i^{k+1} \Delta_i^{k+1} = M \Phi Q^T$. Compute $\epsilon^{k+1,k+1,k+1}$
     If $\epsilon^{k+1,k+1,k+1} < \epsilon^{k+1,k+1,k}$:
       $V \leftarrow V^{k+1}$
7. Repeat steps 4-6 until convergence.
```

Our iterative approach is a block-coordinate descent algorithm. In gradient descent, we have the general update:

$$\theta_{t+1} = \theta_t - \eta \nabla \mathcal{F}(\theta_t),$$

where t is the iteration counter, η is the learning rate and $\nabla \mathcal{F}(\theta)$ is the gradient of the cost function. In A-SBF, the gradient of the cost function with respect to U is $-2DV\Delta$. So we have

$$U_{t+1} = U_t + \eta D V \Delta.$$

By setting $\eta = I - U_t(DV\Delta)^{-1}$, we have $U_{t+1} = DV\Delta$. Since we require orthonormal columns, we find the closest orthogonal matrix to $DV\Delta$, and set the final value of $U_{t+1} = ZY^T$, where SVD of $DV\Delta = Z\Sigma Y^T$. This approach employs orthogonal Procrustes solution along with the gradient descent algorithm. We can also achieve this by directly setting $\eta = \frac{ZY^T - U_t}{DV\Delta}$, so that U_{t+1} has all the properties we need. The learning rate determines the size of the steps. If the value of U_t is very different from ZY^T, η will be high, and we take larger steps. As it becomes closer, the learning rate also decreases. We have a similar case for updating V and Δ.

5.5 Examples

5.5.1 Optimizing A-SBF error

Let us optimize the factorization error using the `optimizeFactorization` function for the three cases of A-SBF computation.

```r
set.seed(1231)
mymat <- createRandomMatrices(n = 4, ncols = 3, nrows = 4:6)
asbf <- SBF(matrix_list = mymat, approximate = TRUE)
asbf_inv <- SBF(matrix_list = mymat, weighted = TRUE, approximate = TRUE)
asbf_cor <- SBF(matrix_list = mymat, approximate = TRUE, transform_matrix = TRUE)
```

Depending upon the data matrix and initial values of U_i, Δ_i, and V, optimization could take some time.

```r
myopt <- optimizeFactorization(mymat, asbf$u_ortho, asbf$delta, asbf$v)
myopt_inv <- optimizeFactorization(mymat, asbf_inv$u_ortho, asbf_inv$delta,
                                   asbf_inv$v)
```

```
myopt_cor <- optimizeFactorization(mymat, asbf_cor$u_ortho, asbf_cor$delta,
                                   asbf_cor$v)
```

The number of iteration taken for optimizing and new factorization error:

```
cat("For asbf, # iteration =", myopt$error_pos, "final error =", myopt$error)
#> For asbf, # iteration = 220 final error = 1411.555
cat("\nFor asbf inv, # iteration =", myopt_inv$error_pos, "final error =",
    myopt_inv$error)
#>
#> For asbf inv, # iteration = 202 final error = 1411.555
cat("\nFor asbf cor, # iteration =", myopt_cor$error_pos, "final error =",
    myopt_cor$error)
#>
#> For asbf cor, # iteration = 196 final error = 1411.555
```

```
#> After optimization, for all three A-SBF factorizations, the final error
#> is the same (up to 2 decimals).
#> The final error is 1411.56
```

5.5.2 Using different initial values

```
set.seed(1231)
mymat <- createRandomMatrices(n = 4, ncols = 3, nrows = 4:6)
```

1. Let us initialize the `optimizeFactorization` function with a random orthogonal matrix and check the final optimization error. The V matrix estimated from the `mymat` matrix has a dimension of 3×3. First we will create a random 3×3 matrix and obtain an orthogonal matrix based on this.

```
set.seed(111)
rand_mat <- createRandomMatrices(n = 1, ncols = 3, nrows = 3)
cat("\nRank is:", qr(rand_mat[[1]])$rank,"\n")
#>
#> Rank is: 3
dim(rand_mat[[1]])
#> [1] 3 3
```

Get an orthogonal V matrix using SVD. We will set V as the right basis matrix from the SVD.

```
mysvd <- svd(rand_mat[[1]])
randV <- mysvd$v
```

Now for this V, we will first compute U_i's and Δ_i for different D_i matrices in the `mymat`. We achieve this by solving the linear equations: $D_i = U_i \Delta_i V^T$ for $i = 1, \ldots, 4$. We then orthonormalize the columns of U_i using Proposition I.

```
# get Ui and Delta for this newV
out <- computeUDelta(mymat, randV)
names(out)
#> [1] "u"        "u_ortho" "d"        "d_ortho" "error"
```

The initial decomposition error is :

```
calcDecompError(mymat, out$u_ortho, out$d, randV)
#> [1] 22879.08
```

Now we will try to optimize using the new random V and corresponding U_i's and Δ_i's.

```r
newopt <- optimizeFactorization(mymat, out$u_ortho, out$d, randV)
# Number of updates taken
newopt$error_pos
#> [1] 220
# New error
newopt$error
#> [1] 1411.555
```

We achieve the same factorization error (`1411.5550218`) after the `optimizeFactorization` function call.

2. Now instead of the right basis matrix from the SVD, we will set V as the left basis matrix.

```r
mysvd <- svd(rand_mat[[1]])
randV <- mysvd$u
dim(randV)
#> [1] 3 3
```

```r
# get Ui and Delta for this newV
out <- computeUDelta(mymat, randV)
calcDecompError(mymat, out$u_ortho, out$d, randV)
#> [1] 13903.45
```

Now we will try to optimize with these matrices as our initial values.

```r
newopt <- optimizeFactorization(mymat, out$u_ortho, out$d, randV)
# Number of updates taken
newopt$error_pos
#> [1] 283
# New error
newopt$error
#> [1] 1411.555
```

Again we get the same decomposition error after optimizing.

3. Instead of initial value being an orthogonal matrix, we will initialize U_i's, Δ_i, and V with random matrices such that it does not guarantee

- orthogonal property for V and
- orthonormal columns for U_i's.

```r
set.seed(111)
# new random v
newv <- createRandomMatrices(n = 1, ncols = 3, nrows = 3)[[1]]
# seed value
k <- 2392
newu <- newd <- list()
for(i in names(mymat)){
  myrow <- nrow(mymat[[i]])
  mycol <- ncol(mymat[[i]])
  set.seed(k)
  # new random u_i
  newu[[i]] <- createRandomMatrices(n = 1, ncols = mycol, nrows = myrow)[[1]]
  set.seed(k*2)
  # new random d_i
  newd[[i]] <- sample(1:1000, size = mycol)
  newmat <- newu[[i]] %*% diag(newd[[i]]) %*% t(newv)
  if (!qr(newmat)$rank == mycol)
```

```
    cat("\nNew matrix does not have full column rank")
  k = k+1
}
error <- calcDecompError(mymat, newu, newd, newv)
cat("\nInitial error = ", error,"\n")
#> 
#> Initial error =  2.062531e+15
```

We see a very high factorization error because of the random initialization.

```
newopt <- optimizeFactorization(mymat, newu, newd, newv)
newopt$error_pos
#> [1] 142
newopt$error
#> [1] 1411.555
```

Again, we get the same factorization error after optimizing. Try changing the seed value and compare the results.

This shows that the iterative update procedure converges and achieves the same decomposition error regardless of the initial values.

5.5.3 Estimating SVD

We will further demonstrate the case for $N = 1$ when we have just one matrix. The `optimizeFactorization` function gives U_i's with orthonormal column, Δ_i a diagonal matrix, and an orthogonal V. If the function converges, the results should be identical to a standard SVD, except for the sign changes corresponding to U and V columns. So we will compare the results from the `optimizeFactorization` function with the standard SVD output. Let us generate one example matrix say `newmat`.

```
set.seed(171)
newmat <- createRandomMatrices(n = 1, ncols = 3, nrows = 3)
newmat
#> $mat1
#>      [,1] [,2] [,3]
#> [1,]   41   10    6
#> [2,]   64   85    8
#> [3,]   82   87   57
```

We will estimate the SVD of `newmat` using our iterative update function from another random matrix with the same dimension.

```
set.seed(253)
randmat_new <- createRandomMatrices(n = 1, ncols = 3, nrows = 3)
randmat_new
#> $mat1
#>      [,1] [,2] [,3]
#> [1,]   94   30   77
#> [2,]   60   35  100
#> [3,]   67   84   58
newsvd <- svd(randmat_new[[1]])
```

Let us create a list for the u and δ matrices we just obtained from the SVD of the random matrix. This allows us to use these matrices as the initial values for the `optimizeFactorization` function.

```
newu <- newd <- list()
newu[[names(randmat_new)]] <- newsvd$u
```

```
newd[[names(randmat_new)]] <- newsvd$d
```

The factorization error

```
calcDecompError(newmat, newu, newd, newsvd$v)
#> [1] 19465
```

Let us optimize.

```
opt_new <- optimizeFactorization(newmat, newu, newd, newsvd$v)
cat("\n # of updates:", opt_new$error_pos,"\n")
#> 
#>  # of updates: 235
opt_new$error
#> [1] 0.0006461125
```

Error is close to zero. Let us compare the original matrix with the reconstructed matrix based on the estimated u, d and v using the `optimizeFactorization` function.

```
newmat
#> $mat1
#>      [,1] [,2] [,3]
#> [1,]   41   10    6
#> [2,]   64   85    8
#> [3,]   82   87   57
opt_new$u[[1]] %*% diag(opt_new$d[[1]]) %*% t(opt_new$v)
#>          [,1]     [,2]      [,3]
#> [1,] 40.99550 10.00886  5.988544
#> [2,] 64.01211 84.99162  7.993461
#> [3,] 81.99163 87.00424 57.008260
```

The estimated value is very close.

We can further improve our estimate by decreasing the tolerance parameter (`tol`) in the optimization function.

```
opt_new1 <- optimizeFactorization(newmat, newu, newd, newsvd$v, tol = 1e-21)
cat("\n # of updates:", opt_new1$error_pos,"\n")
#> 
#>  # of updates: 673
opt_new1$error
#> [1] 9.156226e-14
opt_new1$u[[1]] %*% diag(opt_new1$d[[1]]) %*% t(opt_new1$v)
#>      [,1] [,2] [,3]
#> [1,]   41   10    6
#> [2,]   64   85    8
#> [3,]   82   87   57
```

The reconstructed matrix is the same as the original matrix. Let us compare the U and V with that from the standard SVD.

```
newmat_svd <- svd(newmat[[1]])
```

```
newmat_svd$u
#>            [,1]       [,2]        [,3]
#> [1,] -0.2067092  0.1766241 -0.96232802
#> [2,] -0.6071027 -0.7944740 -0.01541011
#> [3,] -0.7672664  0.5810465  0.27145407
opt_new1$u[[1]]
```

```
#>            [,1]       [,2]        [,3]
#> [1,] -0.2067092 -0.1766241  0.96232803
#> [2,] -0.6071027  0.7944740  0.01541009
#> [3,] -0.7672664 -0.5810465 -0.27145405
```

```
newmat_svd$v
#>            [,1]       [,2]       [,3]
#> [1,] -0.6458388  0.1264120 -0.7529358
#> [2,] -0.7054605 -0.4758901  0.5252181
#> [3,] -0.2919209  0.8703727  0.3965270
opt_new1$v
#>            [,1]       [,2]       [,3]
#> [1,] -0.6458388 -0.1264120  0.7529358
#> [2,] -0.7054605  0.4758901 -0.5252181
#> [3,] -0.2919209 -0.8703727 -0.3965269
```

The results agree!

6 Cross-species gene expression analysis using A-SBF

For cross-species gene expression datasets, we estimate the common space V based on correlation (R_i) between column phenotypes (such as tissues, cell types, etc.) within a species. In our study, we have shown that the inter-tissue gene expression correlation is similar across species. Let $X_i \in \mathbb{R}^{m_i \times k}$ be a standardized gene expression matrix where $X_i = C_i D_i {S_i}^{-1}$. Here $C_i = I_{m_i} - {m_i}^{-1} 1_{m_i} {1_{m_i}}^T$ is a centering matrix and $S_i = \text{diag}(s_1, \ldots, s_k)$ is a diagonal scaling matrix, where s_p is the standard deviation of p-th column of D_i. The matrix X_i is a matrix with columns of D_i mean-centered and scaled by the standard deviation. The correlation between expression profiles of k tissue types in species i is given by $R_i = X_i^T X_i / m_i$. We then define an expected correlation matrix $(\mathbb{E}(R_i))$ across N species as M, where M is defined as

$$M = \frac{\sum_{i=1}^{N} R_i}{N}.$$

The shared right basis matrix V capturing the inter-tissue gene expression correlation is determined from the eigenvalue decomposition of M, where $M = V \Theta V^T$. Once the V and Δ_i are estimated using the SBF factorization, we compute U_i with orthonormal columns using proposition I. The estimated V space captures inter-tissue gene expression correlation relationship. For gene expression analysis, if we want the shared space to represent inter-sample correlation relationship, we do not update/change V while optimizing the factorization error. In such cases, while reducing the factorization error we set `optimizeV = FALSE` in the `optimizeFactorization` function.

6.1 Usage examples

Let us load the SBF package's in-built gene expression dataset. The dataset contains the average gene expression profile of five similar tissues in three species.

```
# load dataset
avg_counts <- SBF::TissueExprSpecies
# check the names of species
names(avg_counts)
#> [1] "Homo_sapiens"   "Macaca_mulatta" "Mus_musculus"
```

```
# head for first species
avg_counts[[names(avg_counts)[1]]][1:3, 1:3]
#>                 hsapiens_brain hsapiens_heart hsapiens_kidney
#> ENSG00000000003         2.3109         1.9414          5.2321
#> ENSG00000000005         0.0254         0.2227          0.5317
#> ENSG00000000419         5.2374         5.3901          5.5659
```

The number of genes annotated in different species is different. As a result, the number of rows (genes) in the expression data will be different for different species.

```
sapply(avg_counts, dim)
#>      Homo_sapiens Macaca_mulatta Mus_musculus
#> [1,]        58676          30807        54446
#> [2,]            5              5            5
```

Let us compute A-SBF with inter-tissue correlation.

```
# A-SBF call using correlation matrix
asbf_cor <- SBF(matrix_list = avg_counts, check_col_matching = TRUE,
                col_index = 2, approximate = TRUE, transform_matrix = TRUE)
# decomposition error
asbf_cor$error
#> [1] 65865.92
```

Optimize factorization to reduce decomposition error but by not updating V.

```
myopt_gef <- optimizeFactorization(avg_counts, asbf_cor$u_ortho, asbf_cor$delta,
                                   asbf_cor$v, optimizeV = FALSE)
names(myopt_gef)
#> [1] "u"         "v"         "d"         "error"     "error_pos" "error_vec"
```

```
# new error
myopt_gef$error
#> [1] 63540.08
# number of iterations
myopt_gef$error_pos
#> [1] 10
```

The number of iterations taken to optimize = 10.

```
identical(asbf_cor$v, myopt_gef$v)
#> [1] TRUE
```

Check whether estimated U_i's have orthonormal columns.

```
zapsmall(t(as.matrix(myopt_gef$u[[names(myopt_gef$u)[1]]])) %*%
           as.matrix(myopt_gef$u[[names(myopt_gef$u)[1]]]))
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    0    0    0    0
#> [2,]    0    1    0    0    0
#> [3,]    0    0    1    0    0
#> [4,]    0    0    0    1    0
#> [5,]    0    0    0    0    1
```

References

Alter, Orly, Patrick O Brown, and David Botstein. 2000. "Singular Value Decomposition for Genome-Wide Expression Data Processing and Modeling." *Proceedings of the National Academy of Sciences* 97 (18): 10101–6.

———. 2003. "Generalized singular value decomposition for comparative analysis of genome-scale expression data sets of two different organisms." *Proceedings of the National Academy of Sciences* 100 (6): 3351–56.

Ponnapalli, Sri Priya, Michael A Saunders, Charles F Van Loan, and Orly Alter. 2011. "A higher-order generalized singular value decomposition for comparison of global mRNA expression from multiple organisms." *PloS One* 6 (12): e28072.

Tamayo, Pablo, Daniel Scanfeld, Benjamin L Ebert, Michael A Gillette, Charles WM Roberts, and Jill P Mesirov. 2007. "Metagene projection for cross-platform, cross-species characterization of global transcriptional states." *Proceedings of the National Academy of Sciences* 104 (14): 5959–64.

Van Loan, Charles F. 1976. "Generalizing the Singular Value Decomposition." *SIAM Journal on Numerical Analysis* 13 (1): 76–83.