computational details of low-rank matrix completion via adaptive-impute

Alex Hayes

2019-10-30

Abstract

TODO: motivate ugh this is a show

This paper is a tutorial covering the computational details associated with computing the low-rank rank adaptive imputations for matrices described in (Cho, Kim, and Rohe 2015, 2018). This work extends previous data adaptive matrix imputation strategies such as that of (Mazumder, Hastie, and Tibshirani 2010), and has better performance while eliminating tuning parameters.

The tutorial proceeds in three parts. First, we introduce the imputation algorithms, useful tidbits of linear algebra and the R package `Matrix`, which we will use to illustrate computations. After a naive initial implementation which eats up lots of memory, we demonstrate a memory-efficient implementation. Finally, we extend this memory-efficient implementation to the partially observed matrices that possibly have a large number of observed zeros.

(Bates 2005; Cho, Kim, and Rohe 2015, 2018; Maechler and Bates 2006; Mazumder, Hastie, and Tibshirani 2010; Bro, Acar, and Kolda 2007)

Notation & Algorithm

There are two steps to computing the low rank approximation of (Cho, Kim, and Rohe 2018). First we use compute an initial low rank estimate with the `AdaptiveInitialize` algorithm. This step is essentially a debiased SVD. We then use the initial solution as a seed for the `AdaptiveImpute` algorithm, a form of iterative SVD thresholding with a data adaptive thresholding parameter.

The input to these algorithms is a partially observed matrix M, and r, the desired rank of the low-rank approximation. We define Y to be an indicator matrix that tells us whether or not we observed a particular value of M.

TODO: flip the sign of v, not lambda, the singular values being positive is a good sanity check we don't want to lose

Here $\mathbf{u}_i, \boldsymbol{\lambda}_i, \mathbf{v}_i$ are functions that return the i^{th} left singular vector, singular value, and right singular value, respectively. $\tilde{\alpha}$ is the data adaptive thresholding parameter. Note that it is the average of the uncalculated singular values, and is thus positive.

In line 8, $\langle , \rangle$ denotes the Frobenius inner norm. That is, for vectors x, y and matrices A, B:

Algorithm 1: `AdapativeInitialize`

Input: M, Y and r

$\hat{p} \leftarrow \frac{1}{nd}\sum_{i=1}^{n}\sum_{j=1}^{d} Y_{ij}$

$\Sigma_{\hat{p}} \leftarrow M^T M - (1-\hat{p})\,\mathrm{diag}(M^T M)$

$\Sigma_{t\hat{p}} \leftarrow M M^T - (1-\hat{p})\,\mathrm{diag}(M M^T)$

$\hat{V}_i \leftarrow \mathbf{v}_i(\Sigma_{\hat{p}})$ for $i = 1, ..., r$

$\hat{U}_i \leftarrow \mathbf{u}_i(\Sigma_{t\hat{p}})$ for $i = 1, ..., r$

$\tilde{\alpha} \leftarrow \frac{1}{d-r}\sum_{i=r+1}^{d} \boldsymbol{\lambda}_i(\Sigma_{\hat{p}})$

$\hat{\lambda}_i \leftarrow \frac{1}{\hat{p}}\sqrt{\boldsymbol{\lambda}_i(\Sigma_{\hat{p}}) - \tilde{\alpha}}$ for $i = 1, ..., r$

$\hat{s}_i \leftarrow \mathrm{sign}(\langle \hat{V}_i, \mathbf{v}_i(M)\rangle)\,\mathrm{sign}(\langle \hat{U}_i, \mathbf{u}_i(M)\rangle)$ for $i = 1, ..., r$

$\hat{\lambda}_i \leftarrow \hat{s}_i \cdot \hat{\lambda}_i$

return $\hat{\lambda}_i, \hat{U}_i, \hat{V}_i$ *for* $i = 1, ..., r$

$$\langle x, y\rangle = x^T y = \sum_i x_i y_i \tag{1}$$

$$\langle A, B\rangle = \sum_{i,j} A_{ij} B_{ij} = \sum_{ij} A \odot B \tag{2}$$

$$\tag{3}$$

We use $A \odot B$ to mean the elementwise (Hadamard) product of matrices A and B.

We can use the output of `AdaptiveInitialize` to construct a low rank approximation to M via:

$$\hat{M} = \sum_{i=1}^{r} \hat{\lambda}_i \hat{U}_i \hat{V}_i^T \tag{4}$$

Next up is `AdaptiveImpute`:

Here we again have some new notation. First we define

$$P_\Omega(A) = A \odot Y \tag{5}$$

$$P_\Omega^\perp(A) = A \odot (1 - Y) \tag{6}$$

These are the projecttion of a matrix A onto the observed of M and the unobserved elements of M, respectively. Similar Ω is the set of all pairs (i, j) such that $M_{i,j}$ is observed. $\boldsymbol{\lambda}_i^2(A)$ is a function that returns the i^{th} *squared* singular value of A (i.e. $\boldsymbol{\lambda}_i^2(A) = (\boldsymbol{\lambda}_i(A))^2$). Finally $\|\cdot\|_F$ is the Frobenius norm of a matrix.

Algorithm 2: `AdaptiveImpute`

Input: M, Y, r and $\varepsilon > 0$

$Z^{(1)} \leftarrow \texttt{AdaptiveInitialize}(M, Y, r)$

repeat

  $\tilde{M}^{(t)} \leftarrow P_\Omega(M) + P_\Omega^\perp(Z_t)$

  $\hat{V}_i^{(t)} \leftarrow \mathbf{v}_i(\tilde{M}^{(t)})$ for $i = 1, ..., r$

  $\hat{U}_i^{(t)} \leftarrow \mathbf{u}_i(\tilde{M}^{(t)})$ for $i = 1, ..., r$

  $\tilde{\alpha}^{(t)} \leftarrow \frac{1}{d-r} \sum_{i=r+1}^{d} \boldsymbol{\lambda}_i^2(\tilde{M}^{(t)})$

  $\hat{\lambda}_i^{(t)} \leftarrow \sqrt{\boldsymbol{\lambda}_i^2(\tilde{M}^{(t)}) - \tilde{\alpha}^{(t)}}$ for $i = 1, ..., r$

  $Z^{(t+1)} \leftarrow \sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T}$

  $t \leftarrow t + 1$

until $\|Z_{t+1} - Z_t\|_F^2 / \|Z_{t+1}\|_F^2$

return $\hat{\lambda}_i^{(t)}, \hat{U}_i^{(t)}, \hat{V}_i^{(t)}$ *for* $i = 1, ..., r$

Pre-requisites

The Matrix package

If you haven't used the `Matrix` package before, we recommend reading this introduction as well as this 2nd introduction.

```
library(Matrix)

set.seed(27)

# create a random 8 x 12 sparse matrix with 30 nonzero entries
M <- rsparsematrix(8, 12, nnz = 30)
M
```

```
summary(M)
```

TODO: different storage formats for sparse matrices. CSC, triplet, symmetric. for the most part `Matrix` does the right thing for you. The triplet form will be most important for us later on when we right out some matrix multiplications by hand.

```
# note that Matrix objects are S4 classes so we access their
# slots using the @ symbol
class(M)

M@x  # vector of values in M
M@i  # corresponding row indices
```

coercing

```r
# we want a tripletMatrix, but *don't* specify the subclass
M2 <- as(M, "TsparseMatrix")

# i.e. the following is bad style and will possibly break
M2 <- as(M, "dgTMatrix")

# but in triplet format we can get the corresponding column indices
M2@j
```

We will repeatedly calculate squared Frobenious norms throughout this tutorial. It's important to know that there are *many, many* ways to calculate this norm. We will almost always calculate the squared Frobenius norm of a sparse Matrix `M` via `sum(M@x^2)`. That said, these are all equivalent:

```r
M^2    # square each element in M elementwise, return as sparse matrix
M@x^2  # square each element in M elementwise, return as vector of nonzeros

# the second version is much faster
bench::mark(
  sum(M@x^2),
  sum(M^2),
  sum(colSums(M^2)),
  norm(M, type = "F")^2,
  sum(M * M),
  iterations = 20
)
```

The projections $P_\Omega(A)$ and $P_\Omega^\perp(A)$ where Ω indicates the observed elements of a matrix M and A is another matrix with the same dimensions as M.

```r
y <- as(M, "lgCMatrix")  # indicator matrix only
all.equal(y * M, M)  # don't lose anything multiplying by indicators

A <- matrix(1:(8*12), 8, 12)

all.equal(dim(A), dim(M))  # appropriate to practice projections with

# Omega indicates whether an entry of M was observed

# P_Omega (A)
A * y

!y

# P_Omega^perp (A): NOTE: this results in a *dense* matrix
A * (1 - y)
```

```r
all(A * y + A * (1 - y) == A)  # can recover A from both projections together
```

matrix-matrix crossproducts

```r
bench::mark(
  crossprod(M),      # dsCMatrix -- most specialized class, want this
  crossprod(M, M),   # dgCMatrix
  t(M) %*% M,        # dgCMatrix
  check = FALSE
)
```

matrix-vector crossproducts

```r
# TODO
```

the `drop()` function helps us manage dimensions

```r
one_col <- matrix(1:4)
one_row <- matrix(5:8, nrow = 1)

drop(one_col)
drop(one_row)

c(one_col)  # same thing, less explicit. use drop to be explicit
```

```r
# TODO: drop vs drop0 vs drop1
```

diagonal of crossproduct

```r
v_sign == colSums(svd_M$v * v_hat)
diag(t(svd_M$v) %*% v_hat)
diag(crossprod(svd_M$v, v_hat))

bench::mark(
  colSums(svd_M$v * v_hat),
  crossprod(rep(1, d), svd_M$v * v_hat),
  iterations = 50,
  check = FALSE
)

# write a diag_crossprod helper
```

```r
rhos <- matrix(1:12, ncol = 4, byrow = TRUE)

bench::mark(
  diag(crossprod(rhos)),
  diag(t(rhos) %*% rhos),
```

```r
  colSums(rhos * rhos),
  crossprod(rep(1, nrow(rhos)), rhos^2),
  check = FALSE
)


rhos <- matrix(1:12, ncol = 4, byrow = TRUE)

bench::mark(
  diag(tcrossprod(rhos)),
  diag(crossprod(t(rhos))),
  diag(rhos %*% t(rhos)),
  rowSums(rhos * rhos),
  check = FALSE
)
```

what we get from `eigen()` and `svd()`: slightly different stuff: `u`, `d` and `v` versus `values` and `vectors`

NOTE: RSpectra *only* does truncated decompositions. if you want the full decomposition, you have to use base R stuff. different algos.

Brief aside in sign ambiguity

yada yada yada the signs of the left and right singular vectors are not identified in SVD

A more elegant solution as proposed in Bro, Acar, and Kolda (2007) and used in Karl's paper is to take inner products

NOTE TO SELF: identifying the signs of a single SVD is a much harder task than comparing two SVDs and seeing if they are the same up to sign differences. we only need to check if they are the same up to sign differences.

```r
set.seed(17)
M <- rsparsematrix(8, 12, nnz = 30) # small example, not very sparse

# number of singular vectors to compute
k <- 4

s <- svd(M, k, k)
s2 <- svds(M, k, k)

# irritating: svd() always gives you all the singular values even if you
# only request the first K singular vectors
s$u %*% diag(s$d[1:k]) %*% t(s$v)

# A and B are matrices
```

```r
equal_up_to_sign <- function(A, B) {
  isTRUE(all.equal(A, A * sign(A) * sign(B), check.attributes = FALSE))
}

# based on the flip_signs function of
# https://stats.stackexchange.com/questions/134282/relationship-between-svd-and-pca-how-to-use-svd-to-p
equal_svds <- function(s, s2) {

  # svd() always gives you all the singular values, but we only
  # want to compare the first k
  k <- ncol(s$u)

  # the term sign(s$u) * sign(s2$u) performs a sign correction

  # isTRUE because output of all.equal is not a boolean, it's something
  # weird when the inputs aren't equal. lol why

    u_ok <- equal_up_to_sign(s$u, s2$u)
  v_ok <- equal_up_to_sign(s$v, s2$v)
  d_ok <- isTRUE(all.equal(s$d[1:k], s2$d[1:k], check.attributes = FALSE))

  u_ok && v_ok && d_ok
}
```

Linear algebra facts

Throughout these computations, we will repeatedly use several key facts about eigendecompositions, singular value decompositions (SVD) and the relationship between the two.

https://en.wikipedia.org/wiki/Gramian_matrix X'X – positive semi-def, so the singular values of M'M are the same as the eigenvalues

question: if A, B positive, is sum(svd(A - B)$d) == sum(svd(A)$d) - sum(svd(B)$d)

answer: NO! can't split into two easy computations and then combine them possibly use this to get some sort of bound?

think about what happens as p_hat -> 0

A key observation here is that $M^T M$ and

Fact: sum of squared singular values is `trace(A^T A)` https://math.stackexchange.com/questions/2281721/sum-of-singular-values-of-a-matrix

Fact: for symmetric positive definite matrices the eigendecomp is equal to the singular value decomp

Fact: sum of eigenvalues of M is equal to trace(M)

Consequence: for pos def symmetric M the sum of the singular values is trace(M) as well

Again computing `alpha` deserves some explanation.

- reference: https://math.stackexchange.com/questions/1463269/how-to-obtain-sum-of-square-of-eigenvalues-without-fir
- Frobenius norm (A) = trace(crossprod(A))
- TODO: how we know this thing is strictly positive to prevent sqrt() from exploding

```
## STOPPED HERE: WHY ARE the following not the same?
  isSymmetric(sigma_p)
  eigen(sigma_p)$values
  sum(diag(sigma_p))
  sum(svd(sigma_p)$d)
  sum(eigen(sigma_p)$values)

  # let's think just about the first term for a moment
  sum(diag(MtM / p_hat^2))
  sum(svd(MtM / p_hat^2)$d)

  sum(colSums(M^2 / p_hat^2))

  # has some negative eigenvalues

  # Fact: for a symmetric matrix, the singular values are the *absolute* values
  # of the eigenvalues

  # https://www.mathworks.com/content/dam/mathworks/mathworks-dot-com/moler/eigs.pdf

  eigen(sigma_p)$values
  sum(eigen(sigma_p)$values)
  sum(abs(eigen(sigma_p)$values))
  sum(svd(sigma_p)$d)
  sum(abs(diag(sigma_p)))

  # those agree so what about the second term

  # note to self: alpha should be positive
  # issue karl ran into:
  # https://math.stackexchange.com/questions/381808/sum-of-eigenvalues-and-singular-values
  # how to get the sum of singular values itself (start):
  # https://math.stackexchange.com/questions/569989/sum-of-singular-values-of-ab

  # options when sigma_p is not positive definite:
  # - calculate the full SVD
  # - set alpha to zero (don't truncate the singular values)
```

```
  # - this lower bounds the average of the remaining singular values
  # -

  # positive semi-definite is enough since symmetric and eigen/singular values
  # of zero don't matter

  # this is only an issue in the initialization. in the iterative updates
  # we use the squared singular values, which we can more easily calculate
  # the sum of

  # ask Karl what he wants to do about this: computing a full SVD is gonna be really expensive.
```

FACT: sum(diag(crossprod(M))) == sum(M^2)

Reference implementation

```
library(RSpectra)
library(Matrix)
```

```
adaptive_initialize <- function(M, r) {

  # TODO: ignores observed zeros!
  p_hat <- nnzero(M) / prod(dim(M))  # line 1

  MtM <- crossprod(M)
  MMt <- tcrossprod(M)

  # need to divide by p^2 from Cho et al 2016 to get the "right"
  # singular values / singular values on a comparable scale

  # both of these matrices are symmetric, but not necessarily positive
  # this has important implications for the SVD / eigendecomp relationship

  sigma_p <- MtM / p_hat^2 - (1 - p_hat) * diag(diag(MtM))  # line 2
  sigma_t <- MMt / p_hat^2 - (1 - p_hat) * diag(diag(MMt))  # line 3

  # crossprod() and tcrossprod() return dsCMatrix objects,
  # sparse matrix objects that know they are symmetric

  # unfortunately, RSpectra doesn't support dsCMatrix objects,
  # but does support dgCMatrix objects, a class representing sparse
  # but not symmetric matrices
```

```
  # support for dsCMatrix objects in RSpectra is on the way,
  # which will eliminate the need for the following coercions.
  # see: https://github.com/yixuan/RSpectra/issues/15

  sigma_p <- as(sigma_p, "dgCMatrix")
  sigma_t <- as(sigma_t, "dgCMatrix")

  svd_p <- svds(sigma_p, r)  # TODO: is eigs_sym() faster?
  svd_t <- svds(sigma_t, r)

  v_hat <- svd_p$v  # line 4
  u_hat <- svd_t$u  # line 5

  n <- nrow(M)
  d <- ncol(M)

  # NOTE: alpha is incorrect due to singular values and eigenvalues
  # being different when sigma_p is not positive

  alpha <- (sum(diag(sigma_p)) - sum(svd_p$d)) / (d - r)  # line 6
  lambda_hat <- sqrt(svd_p$d - alpha) / p_hat             # line 7

  svd_M <- svds(M, r)

  v_sign <- crossprod(rep(1, d), svd_M$v * v_hat)
  u_sign <- crossprod(rep(1, n), svd_M$u * u_hat)
  s_hat <- drop(sign(v_sign * u_sign))

  lambda_hat <- lambda_hat * s_hat  # line 8

  list(u = u_hat, d = lambda_hat, v = v_hat)
}
```

It's worth commenting on computation of `alpha` and `s_hat`.

When we compute `alpha` in line 20 `adaptive_initialize()`, we don't want to do the full eigendecomposition of $\Sigma_{\hat{p}}$ since that could take a long time, so we use trick and recall that the trace of a matrix (the sum of it's diagonal elements) equals the sum of all the eigenvalues. Then we subtract off the first r eigenvalues, which we do compute, and are left with $\sum_{i=r+1}^{d} \lambda_i(\Sigma_{\hat{p}})$.

```
adaptive_impute <- function(M, r, epsilon = 1e-7) {

  s <- adaptive_initialize(M, r)
  Z <- s$u %*% diag(s$d) %*% t(s$v)  # line 1
```

```r
  delta <- Inf

  while (delta > epsilon) {

    y <- M != 0  # indicator if entry of M observed
    M_tilde <- M + Z * (1 - y)  # line 3

    svd_M <- svds(M_tilde, r)

    u_hat <- svd_M$u  # line 4
    v_hat <- svd_M$v  # line 5

    d <- ncol(M)

    alpha <- (sum(M_tilde^2) - sum(svd_M$d^2)) / (d - r)  # line 6

    lambda_hat <- sqrt(svd_M$d^2 - alpha)  # line 7

    Z_new <- u_hat %*% diag(lambda_hat) %*% t(v_hat)

    delta <- sum((Z_new - Z)^2) / sum(Z^2)
    Z <- Z_new

    print(glue::glue("delta: {round(delta, 8)}, alpha: {round(alpha, 3)}"))
  }

  Z
}
```

Finally we can do a minimal sanity check and see if this code even runs, and see if we are recovering something close-ish to implanted low-rank structure.

```r
n <- 500
d <- 100
r <- 5

A <- matrix(runif(n * r, -5, 5), n, r)
B <- matrix(runif(d * r, -5, 5), d, r)
M0 <- A %*% t(B)

err <- matrix(rnorm(n * d), n, d)
Mf <- M0 + err

p <- 0.3
```

```
y <- matrix(rbinom(n * d, 1, p), n, d)
dat <- Mf * y

init <- adaptive_initialize(dat, r)
filled <- adaptive_impute(dat, r)
```

Low-rank implementation

The reference implementation has some problems. As our data matrix M gets larger, we can no longer fit the dense representation of $\hat{M}$ and $Z^{(t)}$ into memory. Instead, we need to work with just the low rank components $\hat{\lambda}, \hat{U}$ and $\hat{V}$.

This leads us to following implementation:

```
sparse_adaptive_initialize <- function(M, r,
  additional = min(100, ncol(M) / 4)) {

  M <- as(M, "CsparseMatrix")

  n <- nrow(M)
  d <- ncol(M)

  p_hat <- nnzero(M) / (n * d)  # line 1

  # now that we have counted explicitly observed zeros
  # we can make the explicit zeros implicit for more
  # efficiency in the initializer
  M <- drop0(M)

  # NOTE: skip explicit computation of line 2
  # NOTE: skip explicit computation of line 3

  # since sigma_p and sigma_t are symmetric the eigenvectors
  # and left and right singular vectors are all the same
  # and it is slightly nicer to user the eigs_sym() interface
  # here than svds(). however, we need to be careful about
  # the singular values. the singular values will be the
  # absolute values of the eigenvalues

  args <- list(M = M, p = p_hat)

  additional <- min(d - r - 1, additional)

  # next we run into the issue that computing the entire
```

```r
  # SVD of sigma_p will almost always be computational
  # infeasible. instead of computing all the remaining
  # singular values after the first r singular values,
  # we only compute `additional` many more, and assume
  # the uncalculate singular values are zero. this leads
  # to slightly lower value of alpha and less truncation
  # than in the AdapativeInitialize algorithm itself.
  # users can specify the `additional` argument if they
  # better information about the rank of their matrix

  # need the minimum to keep from asking for more singular values
  # of sigma_p than exist. note that the eigenvalues are
  svd_p <- svds(Mx, r + additional, dim = c(d, d), Atrans = Mx,
                args = args)

  svd_t <- svds(Mtx, r, dim = c(n, n), Atrans = Mtx, args = args)

  v_hat <- svd_p$v[, 1:r]  # line 4
  u_hat <- svd_t$u         # line 5

  # approximate calculation of line 6
  alpha <- sum(svd_p$d[r + 1:additional]) / (d - r)

  lambda_hat <- sqrt(svd_p$d[1:r] - alpha) / p_hat  # line 7

  svd_M <- svds(M, r)

  # diag(crossprod()) patterns
  v_sign <- crossprod(rep(1, d), svd_M$v * v_hat)
  u_sign <- crossprod(rep(1, n), svd_M$u * u_hat)
  s_hat <- drop(sign(v_sign * u_sign))  # line 8

  # make the sign adjustment to v_hat so we don't have
  # to carry s_hat around with use. multiplies each
  # *row* of v_hat (i.e. column v_hat^T) by the corresponding
  # element of s_hat
  v_hat <- sweep(v_hat, 2, s_hat, "*")

  list(u = u_hat, d = lambda_hat, v = v_hat)
}

r <- 5
lr_init <- sparse_adaptive_initialize(M, r, additional = 7)
```

```r
s <- svd(M, r)

lr_init$d
s$d[1:r]

lr_init$d - s$d[1:r]

lr_init$u - s$u


# sanity check and make sure shit is on the same scale, etc
```

Now we need to make sure we've calculated the SVD correctly

(TODO: I suspect the reference code may be wrong)

(TODO: be sure the dimension notation is consistent throughout document)

Let $M \in \mathbb{R}^{n \times d}$. Observe that $\Sigma_{\hat{p}} \in \mathbb{R}^{d \times d}$ and $\Sigma_{t\hat{p}} \in \mathbb{R}^{n \times n}$. So let $x \in \mathbb{R}^d, y \in \mathbb{R}^n$. We need to be able to calculate

$$\Sigma_{\hat{p}}\, y \tag{7}$$

$$= M^T My - (1 - \hat{p}) \operatorname{diag}(M^T M)y \tag{8}$$

and

$$\Sigma_{t\hat{p}}^T\, x \tag{9}$$

$$= MM^T x - (1 - \hat{p}) \operatorname{diag}(MM^T)x \tag{10}$$

TODO: figure out all this divide by p^2 nonsense

```r
# args are `M` and `p`
# M is n x d, x is a vector in R^d. M^T M is then d x d.
# computes M^T M x / p^2 - (1 - p) diag(M^T M) x
# using tricks from before to calculate diagonals of cross-products
Mx <- function(x, args) {
  drop(
    crossprod(args$M, args$M %*% x) -
      (1 - args$p) * Diagonal(ncol(args$M), colSums(args$M^2)) %*% x
  )
}

# args are `M` and `p`
```

```r
# M is n x d, x is a vector in R^d. M M^T is then
# computes M^T M x / p^2 - (1 - p) diag(M^T M) x
# using tricks from before to calculate diagonals of cross-products
Mtx <- function(x, args) {
  drop(
    args$M %*% crossprod(args$M, x) -
      (1 - args$p) * Diagonal(nrow(args$M), rowSums(args$M^2)) %*% x
  )
}
```

Now we check that `Mx()` and `eigen_helper()` work

```r
library(RSpectra)
library(Matrix)
library(testthat)

n <- 8
d <- 12

M <- rsparsematrix(n, d, nnz = 20)

x <- rnorm(d)
y <- rnorm(n)

p_hat <- nnzero(M) / prod(dim(M))
r <- 5

MtM <- crossprod(M)
MMt <- tcrossprod(M)

sigma_p <- MtM / p_hat^2 - (1 - p_hat) * diag(diag(MtM))
sigma_t <- MMt / p_hat^2 - (1 - p_hat) * diag(diag(MMt))

sigma_p <- as(sigma_p, "dgCMatrix")
sigma_t <- as(sigma_t, "dgCMatrix")

args <- list(M = M, p = p_hat)

Mx_expected <- sigma_p %*% x
Mx_result <- Mx(x, args)

t(sigma_p) %*% y

Mtx_expected <- sigma_t %*% y
```

```r
Mtx_result <- Mtx(y, args)

expect_equal(
  drop(Mx_result),
  drop(Mx_expected)
)

expect_equal(
  drop(Mtx_result),
  drop(Mtx_expected)
)

expected_svd_p <- svds(sigma_p, r)
expected_svd_t <- svds(sigma_t, r)


# why use svds() instead of eigs_sym()? mostly to avoid
# screwing around with possibly negative eigenvalues
# TODO: is this slower?
svd_p_result <- svds(Mx, r, Atrans = Mx, dim = c(d, d), args = args)
svd_t_result <- svds(Mtx, r, Atrans = Mtx, dim = c(n, n), args = args)

expect_true(
  equal_svds(svd_p_result, expected_svd_p)
)

expect_true(
  equal_svds(svd_t_result, expected_svd_t)
)

eig_p <- eigs_sym(Mx, r, n = d, args = args)
eig_t <- eigs_sym(Mtx, r, n = n, args = args)

# left and right singular vectors are both just the eigenvectors
# since we're working with symmetric matrices

equal_up_to_sign(eig_p$vectors, svd_p_result$u)
equal_up_to_sign(eig_t$vectors, svd_t_result$u)

equal_up_to_sign(eig_p$vectors, svd_p_result$v)
equal_up_to_sign(eig_t$vectors, svd_t_result$v)
```

TODO: update alg description to divide by p^2 to get the right singular values

Quickly check that the components works before we try the code that integrates them all together

Finally, sanity check this by comparing to the reference implementation. These don't agree, which isn't great:

```
lr_init <- sparse_adaptive_initialize(dat, r)

# some weird stuff is happening with the singular values but I'm
# going to not worry about it for the time being

equal_svds(init, lr_init)
```

Space-efficient adaptive impute

TODO: figure out the actual space complexity

Recall the algorithm looks like

$$\hat{M} = \sum_{i=1}^{r} \hat{s}_i \hat{\lambda}_i \hat{U}_i \hat{V}_i^T \tag{11}$$

Algorithm 3: `AdapativeImpute`

Input: M, y, r and $\varepsilon > 0$

$Z^{(1)} \leftarrow \texttt{AdaptiveInitialize}(M, y, r)$

repeat

$\quad \tilde{M}^{(t)} \leftarrow P_\Omega(M) + P_\Omega^\perp(Z_t)$

$\quad \hat{V}_i^{(t)} \leftarrow \mathbf{v}_i(\tilde{M}^{(t)})$ for $i = 1, ..., r$

$\quad \hat{U}_i^{(t)} \leftarrow \mathbf{u}_i(\tilde{M}^{(t)})$ for $i = 1, ..., r$

$\quad \tilde{\alpha}^{(t)} \leftarrow \frac{1}{d-r} \sum_{i=r+1}^{d} \boldsymbol{\lambda}_i^2(\tilde{M}^{(t)})$

$\quad \hat{\lambda}_i^{(t)} \leftarrow \sqrt{\boldsymbol{\lambda}_i^2(\tilde{M}^{(t)}) - \tilde{\alpha}^{(t)}}$ for $i = 1, ..., r$

$\quad Z^{(t+1)} \leftarrow \sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T}$

$\quad t \leftarrow t + 1$

until $\|Z_{t+1} - Z_t\|_F^2 / \|Z_{t+1}\|_F$

return $\hat{\lambda}_i^{(t)}, \hat{U}_i^{(t)}, \hat{V}_i^{(t)}$ *for* $i = 1, ..., r$

Now we need two things:

1. The SVD of $\tilde{M}^{(t)}$
2. (Certain sums of) the squared singular values.

Sums of squared singular values

For a matrix A, the sum of squared singular values (denoted by λ_i) equals the squared frobenius norm:

$$\sum_{i=1}^{\min(n,d)} \lambda_i^2 = ||A||_F^2 = \mathrm{trace}(A^T A) \tag{12}$$

Also note that

$$||A+B||_F^2 = ||A||_F^2 + ||B||_F^2 + 2 \cdot \langle A, B \rangle_F \tag{13}$$

Now we consider $\tilde{M}^{(t)}$. Suppose that unobserved values of M are set to zero, as is the case for M stored in a sparse matrix representation

$$\begin{aligned} \tilde{M}^{(t)} &= P_\Omega(M) + P_\Omega^\perp(Z_t) && (14) \\ &= P_\Omega(M) + P_\Omega^\perp\left(\sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T}\right) && (15) \end{aligned}$$

Now we need

$$\begin{aligned} ||\tilde{M}^{(t)}||_F^2 &= \left\| P_\Omega(M) + P_\Omega^\perp\left(\sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T}\right)\right\|_F^2 && (16) \\ &= \|P_\Omega(M)\|_F^2 + \left\| P_\Omega^\perp\left(\sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T}\right)\right\|_F^2 + 2 \cdot \left\langle P_\Omega(M), P_\Omega^\perp\left(\sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T}\right)\right\rangle_F && (17) \\ &= \|P_\Omega(M)\|_F^2 + \left\| P_\Omega^\perp\left(\sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T}\right)\right\|_F^2 && (18) \end{aligned}$$

Where the cancellation in the final line follows because

$$\left\langle P_\Omega(M), P_\Omega^\perp\left(\sum_{i=1}^{r} \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T}\right)\right\rangle_F = \sum_{i,j} P_\Omega(M)_{ij} \cdot P_\Omega^\perp(Z_t)_{ij} = \sum_{i,j} 0 = 0 \tag{19}$$

Now we need one more trick, which is that

$$\|Z_t\|_F^2 = \left\|P_\Omega(Z_t) + P_\Omega^\perp(Z_t)\right\|_F^2 = \|P_\Omega(Z_t)\|_F^2 + \left\|P_\Omega^\perp(Z_t)\right\|_F^2 \tag{20}$$

and then

$$\begin{aligned} \left\|P_\Omega^\perp(Z_t)\right\|_F^2 &= \|Z_t\|_F^2 - \|P_\Omega(Z_t)\|_F^2 && (21) \\ &= \sum_{i=1}^{r} \lambda_i^2 - \|Z_t \odot Y\|_F^2 && (22) \end{aligned}$$

Putting it all together we see

$$||\tilde{M}^{(t)}||_F^2 = \|P_\Omega(M)\|_F^2 + \left\| P_\Omega^\perp \left(\sum_{i=1}^r \hat{\lambda}_i^{(t)} \hat{U}_i^{(t)} \hat{V}_i^{(t)^T} \right) \right\|_F^2 \tag{23}$$

$$= \|M\|_F^2 + \sum_{i=1}^r \lambda_i^2 - \|Z_t \odot Y\|_F^2 \tag{24}$$

In code we will have a sparse matrix `M` and a list `s` with elements of the SVD. The first Frobenious norm is quick to calculate, but I am not sure how to calculate the other two frobenius norms.

```
# s is a matrix defined in terms of it's svd
# G is a sparse matrix
# compute only elements of U %*% diag(d) %*% t(V) only on non-zero elements of G
# G and U %*% t(V) must have same dimensions

# maybe call this svd_perp?
svd_perp <- function(s, mask) {

  # note: must be dgTMatrix to get column indexes j larger
  # what if we used dlTMatrix here?
  m <- as(mask, "dgTMatrix")

  # the indices for which we want to compute the matrix multiplication
  # turn zero based indices into one based indices
  i <- m@i + 1
  j <- m@j + 1

  # gets rows and columns of U and V to multiply, then multiply
  ud <- s$u %*% diag(s$d)
  left <- ud[i, ]
  right <- s$v[j, ]

  # compute inner products to get elements of U %*% t(V)
  uv <- rowSums(left * right)

  # NOTE: specify dimensions just in case
  sparseMatrix(i = i, j = j, x = uv, dims = dim(mask))
}
```

Test it

```
set.seed(17)

M <- rsparsematrix(8, 12, nnz = 30)
```

```r
s <- svds(M, 5)

y <- as(M, "lgCMatrix")

Z <- s$u %*% diag(s$d) %*% t(s$v)

all.equal(
  svd_perp(s, M),
  Z * y
)
```

So, to take an eigendecomp you just need to be able to do Mx. To take an SVD, what do you need? matrix vector and matrix transpose vector multiplication

```r
set.seed(17)
r <- 5

M <- rsparsematrix(8, 12, nnz = 30)
y <- as(M, "lgCMatrix")

s <- svds(M, r)
Z <- s$u %*% diag(s$d) %*% t(s$v)

M_tilde <- M + Z * (1 - y)  # dense!

Z_perp <- svd_perp(s, M)
sum_singular_squared <- sum(M@x^2) + sum(s$d^2) - sum(Z_perp@x^2)

all.equal(
  sum(svd(M_tilde)$d^2),
  sum_singular_squared
)
```

SVD of M tilde

```r
set.seed(17)
r <- 5

M <- rsparsematrix(8, 12, nnz = 30)
y <- as(M, "lgCMatrix")

s <- svds(M, r)
Z <- s$u %*% diag(s$d) %*% t(s$v)
```

```r
M_tilde <- M + Z * (1 - y)  # dense!

svd_M_tilde <- svds(M_tilde, r)
svd_M_tilde
```

```r
Ax <- function(x, args) {
  drop(M_tilde %*% x)
}

Atx <- function(x, args) {
  drop(t(M_tilde) %*% x)
}

# is eigs_sym() with a two-sided multiply faster?
args <- list(u = s$u, d = s$d, v = s$v, m = M)
test1 <- svds(Ax, k = r, Atrans = Atx, dim = dim(M), args = args)

test1
svd_M_tilde

all.equal(
  svd_M_tilde,
  test1
)
```

So we're done our first sanity check of the function interface. Let x be a vector. Now we want to calculate

$$
\begin{aligned}
\tilde{M}^{(t)} x &= \left[P_\Omega(M) + P_\Omega^\perp(Z_t)\right] x && (25) \\
&= \left[P_\Omega(M) - P_\Omega(Z_t) + P_\Omega(Z_t) + P_\Omega^\perp(Z_t)\right] x && (26) \\
&= P_\Omega(M - Z_t) x + Z_t x && (27)
\end{aligned}
$$

where we can think of $R_t \equiv P_\Omega(M - Z_t)$ as "residuals" of sorts. Crucially, R_t is sparse, and

$$
\begin{aligned}
Z_t x &= (\hat{U} \operatorname{diag}(\hat{\lambda}_1, ..., \hat{\lambda}_r) \hat{V}^t) x && (28) \\
&= (\hat{U} (\operatorname{diag}(\hat{\lambda}_1, ..., \hat{\lambda}_r)(\hat{V}^t x))) && (29)
\end{aligned}
$$

So now the memory requirement of the computation has been reduced to that of two sparse matrix vector multiplications, rather than that of fitting the dense matrix $P_\Omega^\perp(Z_t)$ into memory.

Similarly, for the transpose, we have

$$\tilde{M}^{(t)^T} x = \left[P_\Omega(M) + P_\Omega^\perp(Z_t)\right]^T x \tag{30}$$

$$= \left[P_\Omega(M) - P_\Omega(Z_t) + P_\Omega(Z_t) + P_\Omega^\perp(Z_t)\right]^T x \tag{31}$$

$$= P_\Omega(M - Z_t)^T x + Z_t^T x \tag{32}$$

This leads us to a second, less memory intensive implementation of `Ax()` and `Atx()`:

```
# input: M, Z_t as a low-rank SVD list s

R <- M - svd_perp(s, M)  # residual matrix
args <- list(u = s$u, d = s$d, v = s$v, R = R)

Ax <- function(x, args) {
  drop(args$R %*% x + args$u %*% diag(args$d) %*% crossprod(args$v, x))
}

Atx <- function(x, args) {
  # TODO: can we use a crossprod() for the first multiplication here?
  drop(t(args$R) %*% x + args$v %*% diag(args$d) %*% crossprod(args$u, x))
}

# is eigs_sym() with a two-sided multiply faster?
test2 <- svds(Ax, k = r, Atrans = Atx, dim = dim(M), args = args)

all.equal(
  svd_M_tilde,
  test2
)
```

```
relative_f_norm_change <- function(s_new, s) {
  # TODO: don't do the dense calculation here

  Z_new <- s_new$u %*% diag(s_new$d) %*% t(s_new$v)
  Z_new <- s$u %*% diag(s$d) %*% t(s$v)

  sum((Z_new - Z)^2) / sum(Z^2)
}
```

```
low_rank_adaptive_impute <- function(M, r, epsilon = 1e-03) {

  # coerce M to sparse matrix such that we use sparse operations
  M <- as(M, "dgCMatrix")

  # low rank svd-like object, s ~ Z_1
```

```r
  s <- low_rank_adaptive_initialize(M, r)  # line 1
  delta <- Inf
  d <- ncol(M)
  norm_M <- sum(M@x^2)

  while (delta > epsilon) {

    # update s: lines 4 and 5
    # take the SVD of M-tilde

    R <- M - svd_perp(s, M)  # residual matrix
    args <- list(u = s$u, d = s$d, v = s$v, R = R)

    s_new <- svds(Ax, k = r, Atrans = Atx, dim = dim(M), args = args)

    MtM <- norm_M + sum(s_new$d^2) - sum(svd_perp(s_new, M)^2)
    alpha <- (sum(MtM) - sum(s_new$d^2)) / (d - r)  # line 6

    s_new$d <- sqrt(s_new$d^2 - alpha)  # line 7

    # NOTE: skip explicit computation of line 8
    delta <- relative_f_norm_change(s_new, s)

    s <- s_new

    print(glue::glue("delta: {round(delta, 8)}, alpha: {round(alpha, 3)}"))
  }

  s
}
```

```r
out <- low_rank_adaptive_impute(M, r)
out
```

$$\texttt{MtM} = {\tilde{M}^{(t)}}^T \tilde{M}^{(t)} \tag{33}$$

Extension to a mixture of observed and unobserved missingness

originally solving an optimization vaguely of the form

$$\left\| M - \hat{M} \right\|_F^2 \tag{34}$$

where M is a partially observed matrix. now, we let $M' = YM$ where Y is an indicator of whether or not M was observed. Typically we have some setup like

$$M = \begin{bmatrix} \cdot & \cdot & 3 & 1 & \cdot \\ 3 & \cdot & \cdot & 8 & \cdot \\ \cdot & -1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 2 & \cdot & \cdot & \cdot \\ 5 & \cdot & 7 & \cdot & 4 \end{bmatrix}, \qquad Y = \begin{bmatrix} \cdot & \cdot & 1 & 1 & \cdot \\ 1 & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot \\ 1 & \cdot & 1 & \cdot & 1 \end{bmatrix} \tag{35}$$

Here the symbol $\cdot$ means that an entry of the matrix was unobserved.

but what we do now is, continuing to represent M as a sparse matrix with no zero entries, is *observe* a bunch of zeros. So we might know that the upper triangule of M has structurally missing zeros that we have observed are missing. These zeros are primarily important because they affect the residuals in our calculations. In this particular case, the take multiplication by M, a sparse operation, and make it into a dense operation.

At this point it becomes useful to introduce some additional notation. Let $\tilde{\Omega}$ be the set of indicies (i, j) such that $M_{i,j}$ is non-zero. Observe that $\tilde{\Omega} \subset \Omega$. Then we have $P_{\tilde{\Omega}}(A) = P_{\Omega}(A)$.

$$M = \begin{bmatrix} 0 & 0 & 3 & 1 & 0 \\ 3 & 0 & 0 & 8 & 0 \\ \cdot & -1 & 0 & 0 & 0 \\ \cdot & \cdot & \cdot & 0 & 0 \\ \cdot & 2 & \cdot & \cdot & 0 \\ 5 & \cdot & 7 & \cdot & 4 \end{bmatrix}, \qquad Y = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ \cdot & 1 & 1 & 1 & 1 \\ \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & 1 & \cdot & \cdot & 1 \\ 1 & \cdot & 1 & \cdot & 1 \end{bmatrix}, \qquad M' = \begin{bmatrix} \cdot & \cdot & 3 & 1 & \cdot \\ 3 & \cdot & \cdot & 8 & \cdot \\ \cdot & -1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 2 & \cdot & \cdot & \cdot \\ 5 & \cdot & 7 & \cdot & 4 \end{bmatrix} \tag{36}$$

And now we need to figure out how to calculate

$$\begin{aligned} \tilde{M}^{(t)} x &= \left[P_{\Omega}(M) + P_{\Omega}^{\perp}(Z_t) \right] x && (37) \\ &= \left[P_{\Omega}(M) - P_{\Omega}(Z_t) + P_{\Omega}(Z_t) + P_{\Omega}^{\perp}(Z_t) \right] x && (38) \\ &= P_{\Omega}(M)x - P_{\Omega}(Z_t)x + Z_t x && (39) \\ &= P_{\tilde{\Omega}}(M)x - P_{\Omega}(Z_t)x + Z_t x && (40) \end{aligned}$$

We already know how to calculate $P_{\tilde{\Omega}}(M)$ and $Z_t x$ using sparse operations, so we're left with $P_{\Omega}(Z_t)x$. Note that $P_{\Omega}(Z_t)x \neq P_{\tilde{\Omega}}(Z_t)x$ since Z_t is not necessarily zero on $\tilde{\Omega}^{\perp}$. In other words $P_{\Omega}^{\perp}(Z_t) \neq Z_t$.

When Y is dense, there is no way to avoid paying the computational cost of the dense or near dense computation. Our primary concern is fitting Y into memory for large datasets. This is an issue when Y is dense but with no discernible structure that permits a more compact representation.

Similarly we need to be able to calculate

$$
\begin{aligned}
\tilde{M}^{(t)^T} x &= \left[P_\Omega(M) + P_\Omega^\perp(Z_t)\right]^T x && (41)\\
&= \left[P_\Omega(M)^T + P_\Omega^\perp(Z_t)^T\right] x && (42)\\
&= \left[P_\Omega(M)^T - P_\Omega(Z_t)^T + P_\Omega(Z_t)^T + P_\Omega^\perp(Z_t)^T\right] x && (43)\\
&= P_\Omega(M)^T x - P_\Omega(Z_t)^T x + Z_t^T x && (44)\\
&= P_{\tilde{\Omega}}(M)^T x - P_\Omega(Z_t)^T x + Z_t^T x && (45)
\end{aligned}
$$

If we can fit Y into memory, we can do a low-rank computation, only calculating elements $Z_{ij}^{(t)}$ when $Y_{ij} = 1$. When Y is stored as a vector of row indices together with a vector of column indices (plus some information about the dimension), we can write the computation out:

```
M <- Matrix(
  rbind(
    c(0, 0, 3, 1, 0),
    c(3, 0, 0, 8, 0),
    c(0, -1, 0, 0, 0),
    c(0, 0, 0, 0, 0),
    c(0, 2, 0, 0, 0),
    c(5, 0, 7, 0, 4)
  )
)

Y <- rbind(
  c(1, 1, 1, 1, 1),
  c(1, 1, 1, 1, 1),
  c(0, 1, 1, 1, 1),
  c(0, 0, 1, 1, 1),
  c(0, 1, 0, 1, 1),
  c(1, 0, 1, 0, 1)
)

s <- svds(M, 2)

Y <- as(Y, "TsparseMatrix")

# triplet form
# compressed column matrix form even better but don't
# understand the format
Y <- as(Y, "lgCMatrix")
Y <- as(Y, "lgTMatrix")

# ugh: RcppArmadillo only supports dgTMatrix rather than
# lgTMatrix which is somewhat unfortunate
```

```r
# link: https://cran.r-project.org/web/packages/RcppArmadillo/vignettes/RcppArmadillo-sparseMatrix.pdf

Y

x <- rnorm(5)

# want to calculate
Z <- s$u %*% diag(s$d) %*% t(s$v)
out <- drop((Z * Y) %*% x)
out
```

At this point it's worth writing out explicitly how calculate $Z_{ij}^{(t)}$.

$$Z_{ij}^{(t)} = \left(\sum_{\ell=1}^{r} \hat{U}_\ell \hat{d}_\ell \hat{V}_\ell^T \right)_{ij} \tag{46}$$

For a visual reminder, this looks like (using $\operatorname{diag}(\hat{d})$ and $\hat{\Sigma}$ somewhat interchangeably here)

$$Z^{(t)} = \hat{U} \operatorname{diag}(\hat{d}) \hat{V}^T = \begin{bmatrix} U_1 & U_2 & \dots & U_r \end{bmatrix} \begin{bmatrix} d_1 & 0 & \dots & 0 \\ 0 & d_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & d_r \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \\ \vdots \\ V_r^T \end{bmatrix} \tag{47}$$

$$\begin{bmatrix} U_{11} & U_{12} & \dots & U_{1r} \\ U_{21} & U_{22} & \dots & U_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ U_{n1} & U_{n2} & \dots & U_{nr} \end{bmatrix} \begin{bmatrix} d_1 & 0 & \dots & 0 \\ 0 & d_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & d_r \end{bmatrix} \begin{bmatrix} V_{11} & V_{21} & \dots & V_{d1} \\ V_{12} & V_{22} & \dots & V_{d2} \\ \vdots & \vdots & \ddots & \vdots \\ V_{1r} & V_{2r} & \dots & V_{dr} \end{bmatrix} \tag{48}$$

n x d = (n x r) x (r x r) x (r x d)

(r x d) is after the transpose

$$\begin{bmatrix} U_{11} & U_{12} & \dots & U_{1r} \\ U_{21} & U_{22} & \dots & U_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ U_{n1} & U_{n2} & \dots & U_{nr} \end{bmatrix} \begin{bmatrix} d_1 & 0 & \dots & 0 \\ 0 & d_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & d_r \end{bmatrix} \begin{bmatrix} V_{11} & V_{21} & \dots & V_{d1} \\ V_{12} & V_{22} & \dots & V_{d2} \\ \vdots & \vdots & \ddots & \vdots \\ V_{1r} & V_{2r} & \dots & V_{dr} \end{bmatrix} \tag{49}$$

Also because I never remember anything, recall that Z is $n \times d$, x is $d \times 1$ and Zx is $n \times 1$:

$$(Zx)_i = \sum_{j=1}^{d} Z_{ij} \cdot x_j \tag{50}$$

(recall that X_i always refers to the i^{th} column of X in this document, and $X_{i\cdot}$ refers to the i^{th} row of X)

Also note that

$$\begin{aligned} Z_{ij} &= \sum_{k=1}^{r} U_{ik} d_k V_{kj} && (51) \\ &= \langle U_{i\cdot}, (DV^T)_j \rangle && (52) \\ &= \langle U_{i\cdot}, (DV)_{j\cdot} \rangle && (53) \\ &= U_{i\cdot}^T (DV)_{j\cdot} && (54) \end{aligned}$$

Recall that $x \in \mathbb{R}^d$. Putting these together we find

$$\begin{aligned} (Zx)_i &= \sum_{j=1}^{d} Z_{ij} \cdot x_j && (55) \\ &= \sum_{j=1}^{d} U_{i\cdot}^T (DV)_{j\cdot} \cdot x_j && (56) \end{aligned}$$

```
# mask as a pair list
# L and Z / svd are both n x d matrices
# x is a d x 1 matrix / vector
masked_svd_times_x <- function(s, mask, x) {

  stopifnot(inherits(mask, "lgTMatrix"))

  u <- s$u
  d <- s$d
  v <- s$v

  zx <- numeric(nrow(u))

  # lgTMatrix uses zero based indexing, add one
  row <- mask@i + 1
  col <- mask@j + 1

  # need to loop over index of indexes
  # double looping over i and j here feels intuitive
  # but is incorrect
```

```r
  for (idx in seq_along(row)) {
    i <- row[idx]
    j <- col[idx]

    z_ij <- sum(u[i, ] * d * v[j, ])
    zx[i] <- zx[i] + x[j] * z_ij
  }

  zx
}

# how to calculate just one element of the reconstructed
# data using the SVD

i <- 6
j <- 4

sum(s$u[i, ] * s$d * s$v[j, ])
Z[i, j]

# the whole masked matrix multiply

Z <- s$u %*% diag(s$d) %*% t(s$v)
out <- drop((Z * Y) %*% x)

# check that we did this right
all.equal(
  masked_svd_times_x(s, Y, x),
  out
)
```

This is gonna be painfully slow in R so we rewrite in C++

```cpp
#include <RcppArmadillo.h>

using namespace arma;

// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]

vec masked_svd_times_x_impl(
  const mat& U,
  const rowvec& d,
  const mat& V,
```

```
  const vec& row,
  const vec& col,
  const vec& x) {

  int i, j;
  double z_ij;

  vec zx = zeros<vec>(U.n_rows);

  for (int idx = 0; idx < row.n_elem; idx++) {

    i = row(idx);
    j = col(idx);

    // % does elementwise multiplication in Armadillo
    // accu() gives the sum of elements of resulting vector
    z_ij = accu(U.row(i) % d % V.row(j));

    zx(i) += x(j) * z_ij;
  }

  return zx;
}
```

```
# wrap with slightly nicer interface
masked_svd_times_x_cpp <- function(s, mask, x) {
  drop(masked_svd_times_x_impl(s$u, s$d, s$v, mask@i, mask@j, x))
}
```

```
bench::mark(
  masked_svd_times_x_cpp(s, Y, x),
  masked_svd_times_x(s, Y, x)
)
```

Now we also need to work out $(Z_t^T x)_i$, which I am hella blanking on.

```
library(Matrix)
library(RSpectra)
library(testthat)

set.seed(27)

# create a random 8 x 12 sparse matrix with 30 nonzero entries
M <- rsparsematrix(8, 11, nnz = 20)
M
```

```r
s <- svds(M, 3)

Y <- M != 0
mask <- as(M, "TsparseMatrix")

u <- s$u
d <- s$d
v <- s$v


Z <- u %*% diag(d) %*% t(v)

Zt2 <- v %*% diag(d) %*% t(u)

expect_equal(Zt2, t(Z))

x <- rnorm(8)
Ztx <- t(Z * Y) %*% x

ztx <- numeric(nrow(v))

# lgTMatrix uses zero based indexing, add one
row <- mask@i + 1
col <- mask@j + 1

# need to loop over index of indexes
# double looping over i and j here feels intuitive
# but is incorrect
for (idx in seq_along(row)) {

  # here's where we do the row / column swap
  i <- row[idx]
  j <- col[idx]

  z_ij <- sum(u[i, ] * d * v[j, ])
  ztx[j] <- ztx[j] + x[i] * z_ij
}

Ztx
ztx

impl_result <- p_omega_ztx_impl(u, d, v, mask@i, mask@j, x)
```

```r
testthat::expect_equivalent(
  drop(ztx),
  drop(Ztx)
)
```

```r
# add the upper triangle back to Y since we're in the citation
# setting
Y <- Y | upper.tri(Y)

obs_Zt <- t(Z * Y)                   # project SVD onto observed matrix

Y <- as(Y, "TsparseMatrix")
impl_result <- p_omega_ztx_impl(s$u, s$d, s$v, Y@i, Y@j, x)

expect_equal(
  drop(impl_result),
  drop(obs_Zt %*% x)
)
```

START MISC

Z is an $n \times d$ matrix. Thus we must have $x \in \mathbb{R}^n$, and then $Z^T x \in \mathbb{R}^d$.

$$(Z^T x)_i = \sum_{j=1}^{n} Z_{ij}^T \cdot x_j \tag{57}$$

Also note that

$$\begin{aligned} Z_{ij}^T &= Z_{ji} && (58) \\ &= \sum_{k=1}^{r} d_k U_{jk} V_{ki}^T && (59) \end{aligned}$$

Putting these together we find

$$\begin{aligned} (Z^T x)_i &= \sum_{j=1}^{n} Z_{ij}^T \cdot x_j && (60) \\ &= \sum_{j=1}^{n} \sum_{k=1}^{r} d_k U_{jk} V_{ki} \cdot x_j && (61) \\ &= \sum_{j=1}^{n} U_{j\cdot}^T (DV)_{i\cdot} \cdot x_j && (62) \end{aligned}$$

END MISC

TODO:

- the tranpose multiplication
- the average singular value calculation

Recall that to calculate the average singular value we want

$$||\tilde{M}^{(t)}||_F^2 = \|M\|_F^2 + \sum_{i=1}^{r} \lambda_i^2 - \|Z_t \odot Y\|_F^2 \tag{63}$$

KEY NOTE: λ_i is the i^{th} singular value of the $Z^{(t-1)}$, *not* $Z^{(t)}$.

This is the other computationally intensive part so let's write it up in Armadillo as well

```
#include <RcppArmadillo.h>

using namespace arma;

// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]

double p_omega_f_norm_impl(
    const mat& U,
    const rowvec& d,
    const mat& V,
    const vec& row,
    const vec& col) {

  int i, j;
  double total = 0;

  for (int idx = 0; idx < row.n_elem; idx++) {

    i = row(idx);
    j = col(idx);

    total += accu(U.row(i) % d % V.row(j));
  }

  return total;
}
```

```
# wrap with slightly nicer interface
p_omega_f_norm_cpp <- function(s, mask) {
```

```
  p_omega_f_norm_impl(s$u, s$d, s$v, mask@i, mask@j)
}
```

```
all.equal(
  sum(Z * Y),
  p_omega_f_norm_cpp(s, Y)
)
```

Citation graphs

Now it turns out that if we have a generally sparse mask of additional zeros, what we've done so far works out pretty nicely. When we have lots and lots of additional zeros, then we end up redoing the same computations over and over when we expand $Z^{(t)}$ in the $\tilde{M}^{(t)}x$ multiplication steps.

We're particularly interested in the case where M is a sparse, but we know that all the elements of the upper triangle are observed. This is the case for the adjacency matrix of a citation network, for example.

The fundamental issue now is how to reduce the *time complexity* (recall that we previously reduced the space complexity) of the $P_\Omega(Z_t)x$ operation in

$$\tilde{M}^{(t)}x = P_{\tilde{\Omega}}(M)x - P_\Omega(Z_t)x + Z_t x \tag{64}$$

Let U be the set of pairs (i,j) such that $i > j$. Note that this inequality is strict – in citation matrices we will assume items do not cite themselves (*loud coughing*). Next let $\tilde{U} \subset \Omega$ be the set of pairs (i,j) where $i \leq j$. Observe that $U \cup \tilde{U} = \Omega$. So we can write

$$P_\Omega(Z_t)\,x = P_U(Z_t)\,x + P_{\tilde{U}}(Z_t)\,x \tag{65}$$

$P_{\tilde{U}}(Z_t)x$ is again easy so we focus on:

$$\begin{aligned}
(P_U(Z_t)\,x)_i &= \sum_{j=1}^{d} Z_{ij} \cdot \mathbf{1}(i < j) \cdot x_j && (66)\\
&= \sum_{j=1}^{d} \left(\sum_{k=1}^{r} U_{ik} (DV^T)_{kj} \right) \cdot x_j \cdot \mathbf{1}(i < j) && (67)\\
&= \sum_{j=1}^{d} \left(\sum_{k=1}^{r} U_{ik} (DV^T)_{kj} x_j \right) \cdot \mathbf{1}(i < j) && (68)
\end{aligned}$$

Now let W be an $r \times d$ matrix such that that j^{th} column of W is $(DV^T)_j \cdot x_j$. Then

$$\begin{aligned}
... &= \sum_{j=1}^{d} \sum_{k=1}^{r} U_{ik} (DV^T)_{kj} x_j \cdot \mathbf{1}(i < j) && (69) \\
&= \sum_{k=1}^{r} \sum_{j=1}^{d} U_{ik} (DV^T)_{kj} x_j \cdot \mathbf{1}(i < j) && (70) \\
&= \sum_{k=1}^{r} U_{ik} \sum_{j=1}^{d} (DV^T)_{kj} x_j \cdot \mathbf{1}(i < j) && (71) \\
&= \sum_{k=1}^{r} U_{ik} \sum_{j=1}^{d} W_{kj} \cdot \mathbf{1}(i < j) && (72) \\
&= \sum_{k=1}^{r} U_{ik} \sum_{j=i+1}^{d} W_{kj} && (73)
\end{aligned}$$

Now let $\tilde{W}_{ki} = \sum_{j=i+1}^{d} W_{kj}$. We can form $\tilde{W}$ by taking cumulative row sums of W from the right rather than from the left. Then

$$\begin{aligned}
... &= \sum_{k=1}^{r} U_{ik} \tilde{W}_{ki} && (74) \\
&= \langle U_{i\cdot}, \tilde{W}_i \rangle && (75)
\end{aligned}$$

Also because I never remember anything, recall that Z is $n \times d$, x is $d \times 1$ and Zx is $n \times 1$:

$$(Zx)_i = \sum_{j=1}^{d} Z_{ij} \cdot x_j \tag{76}$$

(recall that X_i always refers to the i^{th} column of X in this document, and $X_{i\cdot}$ refers to the i^{th} row of X)

Also note that

$$\begin{aligned}
Z_{ij} &= \sum_{k=1}^{r} U_{ik} d_k V_{kj} && (77) \\
&= \langle U_{i\cdot}, (DV^T)_j \rangle && (78) \\
&= \langle U_{i\cdot}, (VD)_{j\cdot} \rangle && (79) \\
&= U_{i\cdot}^T (VD)_{j\cdot} && (80)
\end{aligned}$$

Recall that $x \in \mathbb{R}^d$. Putting these together we find

$$\begin{aligned}
(Zx)_i &= \sum_{j=1}^{d} Z_{ij} \cdot x_j && (81) \\
&= \sum_{j=1}^{d} U_{i\cdot}^T (VD)_{j\cdot} \cdot x_j && (82)
\end{aligned}$$

```r
library(Matrix)
library(RSpectra)
library(testthat)

set.seed(27)

# NOTE: this currently works with square and wide matrices
# it could probably be extended to long matrices but that would
# take more work

M <- rsparsematrix(9, 11, nnz = 20)
M

n <- nrow(M)  # 8
d <- ncol(M)  # 11

x <- rnorm(d)

Y <- as(upper.tri(M), "CsparseMatrix")

s <- svds(M, 3)

U <- s$u
DVt <- diag(s$d) %*% t(s$v)

expected <- drop((U %*% DVt * Y) %*% x)
expected

# you may be tempting to do the following, which is incorrect:
# ARRRRGH VECTOR RECYCLING GROSS GROSS GROSS GROSS
# DVtx <- DVt * x
W <- sweep(DVt, 2, x, "*")

# sanity check that the second column is right
testthat::expect_equal(W[, 2], DVt[, 2] * x[2])
#
# # want cumulative row sums starting from the right rather than
# # the left
# W <- matrix(1:6, 2, 3)
# W
#
# ## approach 1
#
```

```r
# W <- cbind(W, 0)
#
# for (j in (ncol(W) - 2):1)
#   W[, j] <- W[, j] + W[, j + 1]
#
# W[, -1]
#
# ## approach 2
#
# CW <- matrixStats::rowCumsums(W)
# -sweep(CW, 1, rowSums(W))

## back to the regularly scheduled programming

CW <- matrixStats::rowCumsums(W)
W_tilde <- -sweep(CW, 1, rowSums(W))

out <- numeric(n)

for (j in 1:n) {
  out[j] <- sum(U[j, ] * W_tilde[, j])
}

expect_equal(
  drop(out),
  drop(expected)
)
```

Now it's time to make this fast in C++

```cpp
#include <RcppArmadillo.h>

using namespace arma;

// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]
arma::vec p_u_zx_impl(
    const arma::mat& U,
    const arma::vec& d,
    const arma::mat& V,
    const arma::vec& x) {

  // just DVt at this point
  arma::mat W = diagmat(d) * V.t();
```

```
  // multiply columns by x to obtain W
  for (int j = 0; j < W.n_cols; j++) {
    W.col(j) *= x(j);
  }

  // cumulative summation step

  // the farthest right column should should always be all zero
  // add a column of zeros to the right side of matrix
  W.insert_cols(W.n_cols, 1);

  // perform the cumulative summation from right to left
  // skip the rightmost two columns, which are already fine
  // and the leftmost column, which we will drop
  for (int j = W.n_cols - 3; j > 0; j--) {
    W.col(j) += W.col(j + 1);
  }

  // drop the leftmost column. now we have W_tilde in full
  W.shed_col(0);

  // do the matrix-vector multiplication
  arma::vec zx = zeros<vec>(U.n_rows);

  for (int j = 0; j < U.n_rows; j++) {
    zx(j) = dot(U.row(j), W.col(j));
  }

  return zx;
}
```

Test that the C++ does what we want it to

```
impl_result <- p_u_zx_impl(s$u, s$d, s$v, x)

expect_equal(
  drop(impl_result),
  drop(expected)
)
```

Now that we have this, we need to complete the calculation of $P_\Omega(Z_t)x$, which requires computing $P_{\tilde{U}}(Z_t)x$. We do this next:

```cpp
#include <RcppArmadillo.h>

using namespace arma;

// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]
arma::vec p_u_tilde_zx_impl(
    const arma::mat& U,
    const arma::rowvec& d,
    const arma::mat& V,
    const arma::vec& row,
    const arma::vec& col,
    const arma::vec& x) {

  // first add the observed elements on the lower triangle

  int i, j;
  double z_ij;

  arma::vec zx = zeros<vec>(U.n_rows);

  for (int idx = 0; idx < row.n_elem; idx++) {

    i = row(idx);
    j = col(idx);

    // only elements of the lower triangle
    if (i > j) {
      z_ij = arma::accu(U.row(i) % d % V.row(j));
      zx(i) += x(j) * z_ij;
    }
  }

  return zx;
}
```

Now we sanity check the calculations

```r
# lower triangular non-zero mask
L <- M & lower.tri(M)

# project
lower_expected <- (s$u %*% diag(s$d) %*% t(s$v) * L) %*% x
```

```r
mask <- as(L, "TsparseMatrix")
lower_impl <- p_u_tilde_zx_impl(s$u, s$d, s$v, mask@i, mask@j, x)

testthat::expect_equal(
  drop(lower_impl),
  drop(lower_expected)
)
```

Now we see if we can combine the two to achieve what was previously happening with `p_omega_zx_impl()`:

```r
mmask <- as(M, "TsparseMatrix")

no_cumsum <- p_omega_zx_impl(s$u, s$d, s$v, mmask@i, mmask@j, x)
yes_cumsum <- p_u_zx_impl(s$u, s$d, s$v, x) +
  p_u_tilde_zx_impl(s$u, s$d, s$v, mmask@i, mmask@j, x)

testthat::expect_equal(
  drop(no_cumsum),
  drop(yes_cumsum)
)
```

Okay now we need to do a little bit of benchmarking to make sure this has all been worth it

```r
# big matrix for more realistic stress test

set.seed(27)

n <- 150000
nnz <- 900000

M <- rsparsematrix(nrow = n, ncol = n, nnz = nnz)
x <- rnorm(n)

s <- svds(M, 10)
mask <- as(M, "TsparseMatrix")

no <- function(s, mask, x) {
  drop(p_omega_zx_impl(s$u, s$d, s$v, mask@i, mask@j, x))
}

yes <- function(s, mask, x) {
  drop(
    p_u_zx_impl(s$u, s$d, s$v, x) +
      p_u_tilde_zx_impl(s$u, s$d, s$v, mask@i, mask@j, x)
  )
```

```
}

bench::mark(
  yes(s, mask, x)
)
```

To make things fast we also need to speed up the transposed multiply. This looks like:

$$
\begin{aligned}
\tilde{M}^{(t)^T} x &= P_{\tilde{\Omega}}(M)^T x - P_{\Omega}(Z_t)^T x + Z_t^T x && (83) \\
&= P_{\tilde{\Omega}}(M)^T x - P_U(Z_t)^T x - P_{\tilde{U}}(Z_t)^T x + Z_t^T x && (84)
\end{aligned}
$$

Quick aside: the upper triangle of A contains all of the elements A_{ij} where $i < j$, and the lower triangule contains all of the elements A_{ij} such that $i > j$. These upper and lower triangule excluding the diagonal. To include the diagonal, use an inclusive inequality rather than a strict inequality.

Again we are primarily concerned with $P_U(Z_t)^T x$. We are now working with $x \in \mathbb{R}^n$. Thus (the first two lines are easiest to understand if you draw a picture rather than work symbolically):

$$
\begin{aligned}
\left(P_U(Z_t)^T x\right)_i &= \left\langle \left(P_U(Z_t)^T\right)_{i \cdot}, x \right\rangle && (85) \\
&= \langle P_U(Z_t)_i, x \rangle && (86) \\
&= \sum_{j=1}^{n} Z_{ji} \cdot \mathbf{1}(j < i) \cdot x_j && \text{watch indices carefully!} \quad (87) \\
&= \sum_{j=1}^{n} \sum_{k=1}^{r} x_j (UD)_{jk} V_{ki}^T \cdot \mathbf{1}(j < i) && (88) \\
&= \sum_{j=1}^{n} \sum_{k=1}^{r} S_{jk} V_{ki}^T \cdot \mathbf{1}(j < i) && (89) \\
&= \sum_{k=1}^{r} V_{ki}^T \sum_{j=1}^{n} S_{jk} \cdot \mathbf{1}(j < i) && (90) \\
&= \sum_{k=1}^{r} V_{ki}^T \sum_{j=1}^{i-1} S_{jk} && (91) \\
&= \sum_{k=1}^{r} \tilde{S}_{ik} V_{ki}^T && (92) \\
&= \langle \tilde{S}_{i \cdot}, V_i^T \rangle && (93)
\end{aligned}
$$

where S is an $n \times r$ matrix such that that j^{th} row of S is $(UD)_{j \cdot} \cdot x_j$ and $\tilde{S}_{ik} = \sum_{j=1}^{i-1} S_{jk}$.

Now we proceed to ~~HAMMERTIME~~ IMPLEMENTATION TIME.

```
library(Matrix)
library(RSpectra)
library(testthat)
```

```r
set.seed(27)

# NOTE: this currently works with square and wide matrices
# it could probably be extended to long matrices but that would
# take more work

M <- rsparsematrix(11, 11, nnz = 20)
x <- rnorm(11)

Y <- as(upper.tri(M), "CsparseMatrix")
s <- svds(M, 3)

expected <- drop(t(s$u %*% diag(s$d) %*% t(s$v) * Y) %*% x)

UD <- s$u %*% diag(s$d)
S <- sweep(UD, 1, x, "*")

# sanity check that the second row is right
expect_equal(S[2, ], UD[2, ] * x[2])

# get S_tilde
S_tilde <- matrixStats::colCumsums(S)
S_tilde <- rbind(0, S_tilde)
S_tilde <- S_tilde[-nrow(S_tilde), ]

out <- numeric(d)

for (i in 1:d) {
  out[i] <- sum(S_tilde[i, ] * s$v[i, ])
}

expect_equal(
  drop(out),
  drop(expected)
)
```

Now bump to Armadillo

```cpp
#include <RcppArmadillo.h>

using namespace arma;

// [[Rcpp::depends(RcppArmadillo)]]
```

```
// [[Rcpp::export]]
arma::vec p_u_ztx_impl(
    const arma::mat& U,
    const arma::vec& d,
    const arma::mat& V,
    const arma::vec& x) {

  // just UD at this point
  arma::mat S = U * diagmat(d);

  // multiply rows by x to obtain W
  for (int i = 0; i < S.n_rows; i++) {
    S.row(i) *= x(i);
  }

  // cumulative summation step

  S = cumsum(S); // column-wise by default
  S.insert_rows(0, 1); // add 1 row of zeros at zeroth row index
  S.shed_row(S.n_rows - 1);

  // do the matrix-vector multiplication
  arma::vec ztx = zeros<vec>(V.n_cols);

  for (int i = 0; i < V.n_cols; i++) {
    ztx(i) = dot(S.row(i), V.row(i));
  }

  return ztx;
}
```

```
#include <RcppArmadillo.h>

using namespace arma;

// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]
double p_u_f_norm_sq_impl(
    const arma::mat& U,
    const arma::rowvec& d,
    const arma::mat& V) {

  int n = U.n_rows;
```

```
  double total = 0;

  arma::mat DVt = diagmat(d) * V.t();

  // i^th row of Z = U D V^T truncated to the portion
  // in the upper triangle
  arma::rowvec Z_i_trunc;

  // for (int i = 0; i < n; i++) {
  //   Z_i_trunc = U.row(i) * DVt.cols(i, n - 1);
  //   total += dot(Z_i_trunc, Z_i_trunc);
  // }

  // norm of the upper triangle excluding diagonal
  for (int i = 0; i < n - 1; i++) {
    Z_i_trunc = U.row(i) * DVt.cols(i + 1, n - 1);
    total += dot(Z_i_trunc, Z_i_trunc);
  }

  return total;
}
```

```
library(Matrix)
library(RSpectra)
library(testthat)

set.seed(27)

# NOTE: this currently works with square and wide matrices
# it could probably be extended to long matrices but that would
# take more work

n <- 11

M <- rsparsematrix(n, n, nnz = 20)
s <- svds(M, 3)

Z <- s$u %*% diag(s$d) %*% t(s$v)
Z_ut <- as(Z * upper.tri(Z), "CsparseMatrix")
expected <- sum(Z_ut^2)

DVt <- diag(s$d) %*% t(s$v)
```

```r
# A <- matrix(0, nrow = nrow(M), ncol = ncol(M))
#
# for (i in 1:(n-1)) {
#   print(i:n)
#   print(s$u[i+1, ] %*% DVt[, (i+1):n])
#   A[i, i:n] <- s$u[i, ] %*% DVt[, i:n]
# }
#
# expect_equivalent(
#   as.matrix(A),
#   as.matrix(Z_ut)
# )

Z_ut_with_diag <- as(Z * upper.tri(Z, diag = TRUE), "CsparseMatrix")

expected <- sum(Z_ut_with_diag^2)

out <- p_u_f_norm_sq_impl(s$u, s$d, s$v)
expect_equal(out, expected)


Z_ut_wo_diag <- as(Z * upper.tri(Z, diag = FALSE), "CsparseMatrix")

expected_wo_diag <- sum(Z_ut_wo_diag^2)
expect_equal(out, expected_wo_diag)
```

Now we speed up the matrix norm calculation

$$
\begin{aligned}
||\tilde{M}^{(t)}||_F^2 &= \|P_\Omega(M)\|_F^2 + \left\|P_\Omega^\perp(Z_t)\right\|_F^2 & (94)\\
&= \|M\|_F^2 + \|Z_t\|_F^2 - \|P_\Omega(Z_t)\|_F^2 & (95)\\
&= \|M\|_F^2 + \|Z_t\|_F^2 - \|P_U(Z_t)\|_F^2 - \|P_{\tilde{U}}(Z_t)\|_F^2 & (96)
\end{aligned}
$$

Now the critical step is calculating $\|P_U(Z_t)\|_F^2$. If M is $n \times n$ and we take a rank r SVD, the naive matrix expansion is $\mathcal{O}(n^2 k)$. For large citation matrices, the n^2 term is prohibitive[1].

[1] And on the 8^{th} day the Lord gathered together a smattering of social eccentricities, tweed jackets with elbow patches, and deep mathematical wisdom, creating the first of what would later on become known as Ph.D advisors. Thus he spake: “That you might go out into the world and discover $\mathcal{O}(nk^2)$ algorithms wheretohence your students have only seen the $\mathcal{O}(n^2 k)$ implementation”. Thanks Karl!

$$\|P_U(Z_t)\|_F^2 = \sum_{i=1}^n \sum_{j=1}^n \langle U_{i\cdot}, DV_j^T \rangle^2 \, \mathbf{1}(i < j) \tag{97}$$

$$= \sum_{i=1}^n \sum_{j=1}^n \left(\sum_{\ell=1}^r U_{i\ell} DV_{\ell j}^T \right)^2 \mathbf{1}(i < j) \tag{98}$$

$$= \sum_{i=1}^n \sum_{j=1}^n \left(\sum_{\ell=1}^r \sum_{q=1}^r U_{i\ell} DV_{\ell j}^T U_{iq} DV_{qj}^T \right) \mathbf{1}(i < j) \tag{99}$$

$$= \sum_{\ell=1}^r \sum_{q=1}^r \left(\sum_{i=1}^n \sum_{j=1}^n U_{i\ell} DV_{\ell j}^T U_{iq} DV_{qj}^T \right) \mathbf{1}(i < j) \tag{100}$$

$$= \sum_{\ell=1}^r \sum_{q=1}^r \left(\sum_{i=1}^n U_{i\ell} U_{iq} \sum_{j=1}^n DV_{\ell j}^T DV_{qj}^T \right) \mathbf{1}(i < j) \tag{101}$$

$$= \sum_{\ell=1}^r \sum_{q=1}^r \left(\sum_{i=1}^n U_{i\ell} U_{iq} \sum_{j=i+1}^n DV_{\ell j}^T DV_{qj}^T \right) \tag{102}$$

We now want to express the inner two sums as an inner product of some U-term with a cumulative summation. Let $U^{\ell q} \in \mathbb{R}^n$, and $V^{\ell q \triangle} \in \mathbb{R}^n$. So we want something like:

$$U^{\ell q} = U_\ell \odot U_q \qquad \text{elementwise product of column vectors} \tag{103}$$

$$U_i^{\ell q} = U_{i\ell} U_{iq} \tag{104}$$

$$V_i^{\ell q \triangle} = \sum_{j=i+1}^n DV_{\ell j}^T DV_{qj}^T \tag{105}$$

and then we get

$$\ldots = \sum_{\ell=1}^r \sum_{q=1}^r \left(\sum_{i=1}^n U_{i\ell} U_{iq} \sum_{j=i+1}^n DV_{\ell j}^T DV_{qj}^T \right) \tag{106}$$

$$= \sum_{\ell=1}^r \sum_{q=1}^r \left(\sum_{i=1}^n U_i^{\ell q} V_i^{\ell q \triangle} \right) \tag{107}$$

$$= \sum_{\ell=1}^r \sum_{q=1}^r \langle U^{\ell q}, V^{\ell q \triangle} \rangle \tag{108}$$

Now we play the implementation game

```
library(Matrix)
library(RSpectra)
library(testthat)

set.seed(27)
```

```r
n <- 11
r <- 3

M <- rsparsematrix(n, n, nnz = 20)
s <- svds(M, r)

Z <- s$u %*% diag(s$d) %*% t(s$v)
Z_ut <- as(Z * upper.tri(Z), "CsparseMatrix")
expected <- sum(Z_ut^2)

U <- s$u
DVt <- diag(s$d) %*% t(s$v)

Z_ut_wo_diag <- as(Z * upper.tri(Z, diag = FALSE), "CsparseMatrix")

expected_wo_diag <- sum(Z_ut_wo_diag^2)
```

```r
total <- 0

for (l in 1:r) {
  for (q in 1:r) {
    U_lq <- U[, l] * U[, q]
    V_lq <- DVt[l, ] * DVt[q, ]

    # shifted inverse cumulative sum
    V_lq_tri <- sum(V_lq) - cumsum(V_lq)

    total <- total + sum(U_lq * V_lq_tri)
  }
}

expect_equal(total, expected_wo_diag)
```

Now let's do it in C++

```cpp
#include <RcppArmadillo.h>

using namespace arma;

// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]
double p_u_f_norm_sq_impl(
    const arma::mat& U,
    const arma::rowvec& d,
```

```
    const arma::mat& V) {

  int r = U.n_cols;

  double f_norm_sq = 0;

  arma::mat DVt = diagmat(d) * V.t();

  arma::vec U_lq;
  arma::rowvec V_lq, V_lq_tri;

  for (int l = 0; l < r; l++) {
    for (int q = 0; q < r; q++) {

      U_lq = U.col(l) % U.col(q);
      V_lq = DVt.row(l) % DVt.row(q);
      V_lq_tri = sum(V_lq) - cumsum(V_lq);

      f_norm_sq += dot(U_lq, V_lq_tri);
    }
  }

  return f_norm_sq;
}
```

Test that it works:

```
out <- p_u_f_norm_sq_impl(s$u, s$d, s$v)
expect_equal(out, expected_wo_diag)
```

Now there is one final thing to speed up (and I swear to god if things are still slow after this I will flip a table), which is the relative change in Frobenius norm

$$\|Z_{t+1} - Z_t\|_F^2 \,/\, \|Z_{t+1}\|_F^2 \tag{109}$$

Recall that we have taken the SVD of both these matrices, and have $Z_{t+1} = U_{t+1} D_{t+1} V_{t+1}^T$ and $Z_t = U_t D_t V_t^T$. So we already have $\|Z_{t+1}\|_F^2 = \sum_{i=1}^r D_{ii}$. Using a result about Frobenius norms from earlier we have

$$\|Z_{t+1} - Z_t\|_F^2 = \|Z_{t+1}\|_F^2 + \|Z_t\|_F^2 - 2\,\langle Z_{t+1}, Z_t \rangle \tag{110}$$

To calculate all of these quickly need to do the same thing, which is to find a term of the form $\langle A, B \rangle$ given the rank r partial SVD of both $A = UDV^T$ and $B = \tilde{U}\tilde{D}\tilde{V}^T$ (I'm switching notation here to a little bit more obvious).

$$
\begin{aligned}
\langle A, B\rangle &= \sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}B_{ij} && (111)\\
&= \sum_{i=1}^{n}\sum_{j=1}^{n}\left(\sum_{\ell=1}^{r}\sum_{q=1}^{r} U_{i\ell}DV^T_{\ell j}\tilde{U}_{iq}\tilde{D}\tilde{V}^T_{qj}\right) && (112)\\
&= \sum_{i=1}^{n}\sum_{j=1}^{n}\left(\sum_{\ell=1}^{r}\sum_{q=1}^{r} U_{i\ell}DV^T_{\ell j}\tilde{U}_{iq}\tilde{D}\tilde{V}^T_{qj}\right) && (113)\\
&= \sum_{\ell=1}^{r}\sum_{q=1}^{r}\left(\sum_{i=1}^{n} U_{i\ell}\tilde{U}_{iq}\right)\left(\sum_{j=1}^{n} DV^T_{\ell j}\tilde{D}\tilde{V}^T_{qj}\right) && (114)
\end{aligned}
$$

In R this looks like:

```r
library(Matrix)
library(RSpectra)
library(testthat)

set.seed(27)

n <- 11
r <- 3

A <- rsparsematrix(n, n, nnz = 20)
s_A <- svds(M, r)

B <- rsparsematrix(n, n, nnz = 20)
s_B <- svds(M, r)

U_A <- s_A$u
DVt_A <- diag(s_A$d) %*% t(s_A$v)

U_B <- s_B$u
DVt_B <- diag(s_B$d) %*% t(s_B$v)


f_inner_prod <- 0

for (l in 1:r) {
  for (q in 1:r) {
    U_lq <- U_A[, l] * U_B[, q]
    V_lq <- DVt_A[l, ] * DVt_B[q, ]

    f_inner_prod <- f_inner_prod + sum(U_lq) * sum(V_lq)
```

```r
  }
}

Z_A <- U_A %*% DVt_A
Z_B <- U_B %*% DVt_B

frob_expected <- sum(Z_A * Z_B)

expect_equal(f_inner_prod, frob_expected)
```

Then we translate to C++

```cpp
#include <RcppArmadillo.h>

using namespace arma;

// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]
double svd_frob_inner_prod(
    const arma::mat& new_U,
    const arma::rowvec& new_d,
    const arma::mat& new_V,
    const arma::mat& U,
    const arma::rowvec& d,
    const arma::mat& V) {

  // assumes dimensions of new and old SVD match up, both internally
  // and between the SVDs
  int r = new_U.n_cols;

  arma::mat new_DVt = diagmat(new_d) * new_V.t();
  arma::mat DVt = diagmat(d) * V.t();

  arma::vec U_lq;
  arma::rowvec V_lq;

  double frob_inner_prod;

  for (int l = 0; l < r; l++) {
    for (int q = 0; q < r; q++) {
      U_lq = new_U.col(l) % U.col(q);
      V_lq = new_DVt.row(l) % DVt.row(q);

      frob_inner_prod += sum(U_lq) * sum(V_lq);
```

```
    }
  }

  return frob_inner_prod;
}
```

Now we sanity check the C++ version

```
frob_cpp <- svd_frob_inner_prod(s_A$u, s_A$d, s_A$v, s_B$u, s_B$d, s_B$v)
expect_equal(frob_cpp, frob_expected)
```

Numerically stable Frobenius norms

issue: overflow and underflow. http://arma.sourceforge.net/docs.html#norm http://www.cs.utexas.edu/users/flame/Notes/NotesOnStability.pdf

references: https://timvieira.github.io/blog/post/2014/11/10/numerically-stable-p-norms/

TODO: we probably want to do a fast norm calculation in general if possible? not strictly necessary but would let us use relative change in frobenius norm to assess convergence instead of something else

A naive solution: the epsilon trick

issue: we've moved back into dense computation land

Armadillo resources

useful examples to crib from https://github.com/lsolanka/armadillo/blob/master/examples/example1.cpp

http://arma.sourceforge.net/docs.html#syntax

References

Bates, Douglas. 2005. "Introduction to the Matrix Package." https://cran.r-project.org/web/packages/Matrix/vignettes/Introduction.pdf.

Bro, Rasmus, Evrim Acar, and Tamara G. Kolda. 2007. "Resolving the Sign Ambiguity in the Singular Value Decomposition," 18.

Cho, Juhee, Donggyu Kim, and Karl Rohe. 2015. "Asymptotic Theory for Estimating the Singular Vectors and Values of a Partially-Observed Low Rank Matrix with Noise." *arXiv:1508.05431 [Stat]*, August. http://arxiv.org/abs/1508.05431.

———. 2018. "Intelligent Initialization and Adaptive Thresholding for Iterative Matrix Completion; Some Statistical and Algorithmic Theory for Adaptive-Impute." *Journal of Computational and Graphical Statistics*, September, 1–26. https://doi.org/10.1080/10618600.2018.1518238.

Maechler, Martin, and Douglas Bates. 2006. "2nd Introduction to the Matrix Package." https://cran.r-project.org/web/packages/Matrix/vignettes/Intro2Matrix.pdf.

Mazumder, Rahul, Trevor Hastie, and Robert Tibshirani. 2010. "Spectral Regularization Algorithms for Learning Large Incomplete Matrices." https://web.stanford.edu/~hastie/Papers/mazumder10a.pdf.