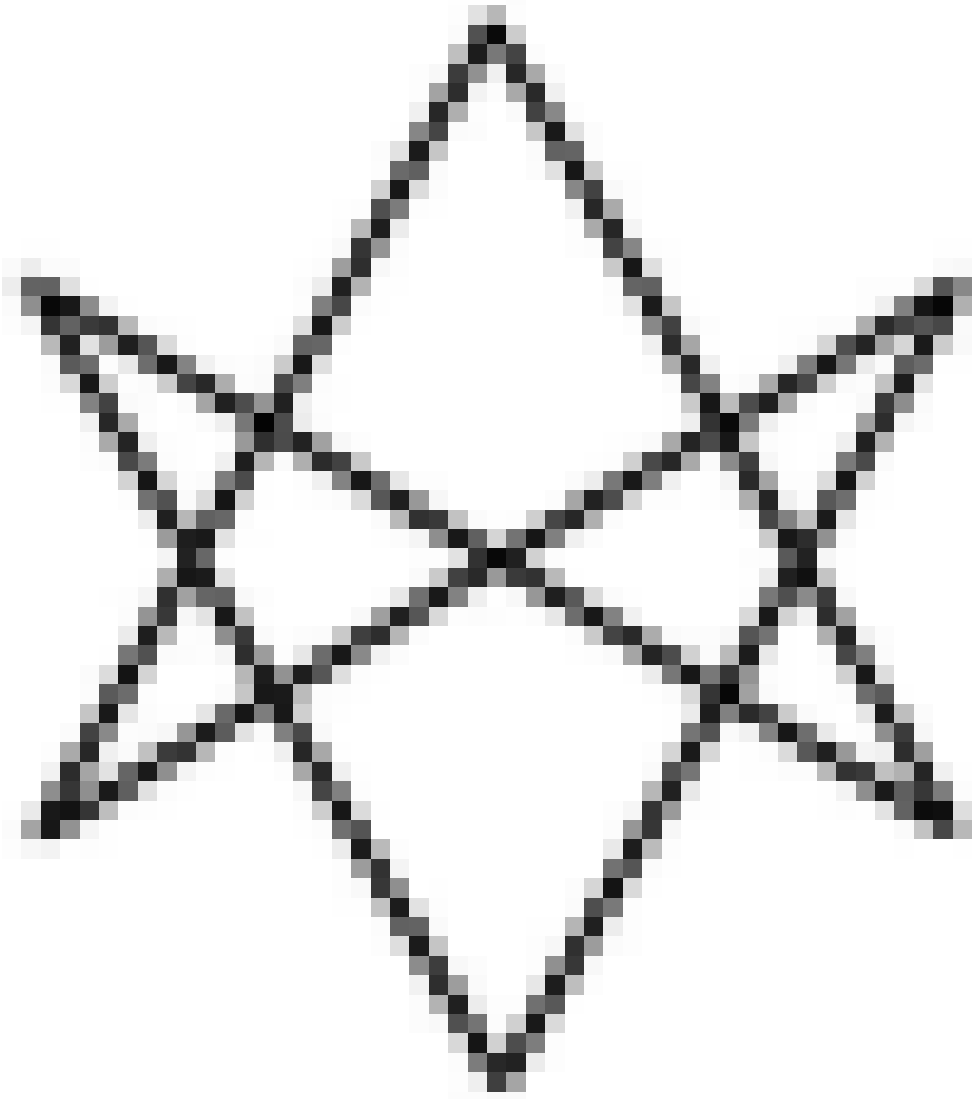
Algorithms and Benchmarks for the coop Package

September 6, 2024

Drew Schmidt
wrathematics@gmail.com



Version 0.6-2

Disclaimer

Any opinions, findings, and conclusions or recommendations expressed in this material are those only of the authors. The findings and conclusions in this article should not be construed to represent any determination or policy of University, Agency, Administration and National Laboratory.

This manual may be incorrect or out-of-date. The author(s) assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This publication was typeset using LaTeX.

Contents

1 Introduction

In this document, we will introduce the algorithms underlying the **coop** package [3], and offer some benchmarks. In order to recreate the benchmarks here, one needs a compiler that supports **OpenMP** [2] and a high-performance **BLAS** library [1]. See the other **coop** package vignette *Introducing coop: Fast Covariance, Correlation, and Cosine Operations* [4] for details.

We do not bother to go into details for the covariance and correlation algorithms, because they are obvious and uninteresting.

1.1 A Note on Sparse Operations

Of the three operations, only cosine similarity currently has a sparse implementation. The short reason why is that the other two operations require centering and/or scaling.

To better understand the problem, consider the 5×20 matrix whose first row is a row of ones, and all other rows consist entirely of zeros:

```
x <- matrix(0, 20, 5)
x[1, ] <- 1
```

The original matrix is, obviously, 95% sparse:

```
coop::sparsity(x)
```

But if we center, the data, it becomes 100% dense:

```
coop::sparsity(scale(x, T, F))
```

2 The Algorithms with Notes on Implementation

For dense implementations, the performance should scale well, and the non-BLAS components will use multiple threads (if your compiler supports OpenMP) when the matrix has more than 1000 columns. Additionally, we try to use vector operations (using OpenMP's `simd` construct) for additional performance; but you need a compiler that supports a relatively modern OpenMP standard for this.

2.1 Dense Matrix Input

Given an $m \times n$ matrix A (input) and an $n \times n$ matrix C (preallocated output):

1. Compute the upper triangle of the crossproduct `C = t(A) %*% X` using a symmetric rank-k update (the `_syrk` BLAS function).
2. Iterate over the upper triangle of C:
   (a) Divide its off-diagonal values by the square root of the product of its i'th and j'th diagonal entries.
   (b) Replace its diagonal values with 1.
3. Copy the upper triangle of C onto its lower triangle.

The total number of floating point operations is:

1. $mn(n+1)$ for the symmetric rank-k update.

2. $\frac{3n(n+1)}{2}$ for the rescaling operation.

The algorithmic complexity is $O(mn^2)$, and is dominated by the symmetric rank-k update. The storage complexity, ignoring the required allocation of outputs (namely the C matrix), is $O(1)$.

2.2 Dense Vector-Vector Input

Given two n-length vectors x and y (inputs):

1. Compute `crossprod = t(x) %*% y` (using the `_gemm` BLAS function).
2. Compute the square of the Euclidean norms of x and y (using the `_syrk` BLAS function).
3. Divide `crossprod` from 1 by the square root of the product of the norms from 2.

The total number of floating point operations is:

1. $2n - 1$ for the crossproduct.
2. $4 * n - 2$ for the two (square) norms.
3. 3 for the division and square root/product.

The algorithmic complexity is $O(n)$. The storage complexity is $O(1)$.

2.3 Sparse Matrix Input

Given an $m \times n$ sparse matrix A stored as a COO with row/column indices i and j **where they are sorted by columns first, then rows**, and corresponding data vector a (inputs), and given a preallocated $n \times n$ dense matrix C (output):

1. Initialize C to 0.
2. For each non-zero column j of the conceptually dense matrix A (call it x), find its first and final position in the COO storage.
   (a) If x is missing (its entries are all 0), set the j'th row and column of the lower triangle of C to `NaN` (for compatibility with dense routines). Go to 2.
   (b) Otherwise, for each column `i>j` of `a` (call it `y`), find its first and final position in the COO storage.
   (c) Compute the dot product of x and y, $x \cdot y$.
   (d) If $x\dot{y} > \epsilon$ (`epsilon=1e-10` for us):
   - Compute the dot products of x with itself $x \cdot x$ and y with itself $y \cdot y$.
   - Set the (i, j)'th entry of C to $\frac{x \cdot y}{\sqrt{x \cdot x}\sqrt{y \cdot y}}$.
3. Copy the lower triangle to the upper and set the diagonal to 1.

The worst case runtime complexity occurs when the matrix is dense but stored as a sparse matrix, and is $O(mn^2)$, the same as in the dense case. However, this will cause serious cache thrashing, and the performance will be abysmal.

The function stores the j'th column data and its row indices in temporary storage for better cache access patterns. Best case, this requires 12 KiB of additional storage, with 8 for the data and 4 for the indices. Worse case (an all-dense column), this balloons up to $12m$. The storage complexity is best case $O(1)$,

and worst case $O(m)$.

3 Benchmarks

The source code for all benchmarks presented here can be found in the source tree of this package under `inst/benchmarks/`, or in the binary installation under `benchmarks/`.

All benchmarks were performed using:

- R 3.2.2
- OpenBLAS
- gcc 5.2.1
- 4 cores of a Core i5-2500K CPU @ 3.30GHz

Throughout the benchmarks, we will use the following packages and data:

```
library(rbenchmark)
reps <- 100
cols <- c("test", "replications", "elapsed", "relative")
```

3.1 Dense Matrix Input

Compared to the version in the lsa package (as of 27-Oct-2015), this implementation performs quite well:

```
m <- 2000
n <- 200
x <- matrix(rnorm(m*n), m, n)

benchmark(coop::cosine(x), lsa::cosine(x), columns=cols, replications=
    reps)
##                test replications elapsed relative
## 1 coop::cosine(x)          100   0.177    1.000
## 2    lsa::cosine(x)          100 113.543  641.486
```

3.2 Dense Vector-Vector Input

Here the two perform identically:

```
n <- 1000000
x <- rnorm(n)
y <- rnorm(n)

benchmark(coop::cosine(x, y), lsa::cosine(x, y), columns=cols,
    replications=reps)
##                   test replications elapsed relative
## 1 coop::cosine(x, y)          100   0.757    1.000
## 2    lsa::cosine(x, y)          100   0.768    1.015
```

3.3 Sparse Matrix Input

Benchmarking sparse matrix methods can be more challenging than with dense for a variety of reasons, chief among them being that the level of sparsity can make an enormous impact in performance.

We present two cases here of varying levels of sparsity. First, we will generate a 0.1% dense / 99.9% sparse matrix:

```
m <- 6000
n <- 250
dense <- coop:::dense_stored_sparse_mat(m, n, .001)
sparse <- slam::as.simple_triplet_matrix(dense)
```

This gives us a fairly dramatic difference in storage:

```
memuse::memuse(dense)
## 11.444 MiB
memuse::memuse(sparse)
## 24.445 KiB
```

So the dense matrix needs roughly 479 times as much storage for the exact same data. In such very sparse cases, the sparse implementation will perform quite nicely:

```
benchmark(dense=coop::cosine(dense), coop::cosine(sparse), columns=cols,
    replications=reps)
##      test replications elapsed relative
## 1   dense          100   0.712    3.082
## 2 sparse          100   0.231    1.000
```

Note that this is a 3-fold speedup over our already highly optimized implementation. This is quite nice, especially considering the sparse implementation uses only one thread and limited vectorization, while the dense one uses 4 threads and vectorization. However, as the matrix becomes more dense (and it doesn't take much), dense methods begin to perform better:

```
dense <- coop:::dense_stored_sparse_mat(m, n, .01)
sparse <- slam::as.simple_triplet_matrix(dense)

memuse::memuse(dense)
## 11.444 MiB
memuse::memuse(sparse)
## 235.383 KiB

benchmark(coop::cosine(dense), coop::cosine(sparse), as.matrix(sparse),
   columns=cols, replications=reps)
benchmark(cosine(dense), cosine(sparse), as.matrix(sparse), columns=cols
   , replications=reps)
##      test replications elapsed relative
## 1   dense          100   0.707    1.000
## 2 sparse          100   2.076    2.936
```

While the sparse implementation performs significantly worse than the dense one for this level of sparsity and data size, note that the memory usage for the dense case is greater than that of the sparse by a factor of 50.

It is hard to give perfect advice for when to use a dense or sparse method, but a general rule of thumb is that if you have more than 5% non-zero data, definitely use dense methods. For 1-5%, there is a memory/runtime tradeoff worth considering; if you can comfortably store the matrix densely, then by all means use dense methods. For data <1% dense, sparse methods will generally have better runtime performance than dense methods.

References

[1] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[2] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, July 2013.

[3] Drew Schmidt. *Co-Operation: Fast Correlation, Covariance, and Cosine Similarity*, 2016. R package version 0.6-0.

[4] Drew Schmidt. *Introducing coop: Fast Covariance, Correlation, and Cosine Operations*, 2016. R Vignette.