Introduction to Matrix Decomposition

A brief introduction to matrix decomposition and some R syntax for carrying out matrix decomposition.

AUTHOR

Andrew Zieffler

PUBLISHED

Sept. 8, 2020

Matrix Decomposition

Matrix decomposition is a method of reducing or factoring a matrix into a set of product matrices. Another name for this is *matrix factorization*. Working with these product matrices often makes it easier to carry out more complex matrix operations (e.g., computing an inverse).

In many ways, matrix decomposition is similar to factoring scalars. For example, we can factor the scalar 36 as:

$$36 = 12 \times 3$$

We could also have used the following factorizations:

$$36 = 9 \times 4$$

$$36 = 18 \times 2$$

$$36 = 6 \times 3 \times 2$$

$$36 = 2 \times 2 \times 3 \times 3$$

The last factorization is referred to as the *prime factorization* as the factors of 36 are all prime numbers.
There are potentially multiple ways to factor a scalar, and in different applications, some of these factorizations may prove more useful than others. The same is true of matrix decompoosition.

- There are many methods of matrix decomposition.

- Depending on the application, some of these methods are more useful than others.

Below we will explore a few of the more common decomposition methods, including LU decomposition, QR decomposition, singular value decomposition, and Cholsky decomposition.

LU Decomposition

One common method of matrix decomposition is LU decomposition. A square matrix, **A**, can be written as the product of two square matrices, **L** and **U**, of the same order.

$$\underset{n\times n}{\mathbf{A}} = \underset{n\times n}{\mathbf{L}}\ \underset{n\times n}{\mathbf{U}}$$

Matrix **L** is a lower-triangular matrix (all elements above the main diagonal are 0) and **U** is an upper-triangular matrix (all elements below the main diagonal are 0). For example, LU decomposition of a 2x2 matrix **A** would be:

$$\underset{2\times 2}{\mathbf{A}} = \underset{2\times 2}{\mathbf{L}}\ \underset{2\times 2}{\mathbf{U}}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ l_{22} & l_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix}$$

The goal would be finding the values for each of the non-zero elements in **L** and **U**. For example consider the following tangible example where

$$\underset{2\times 2}{\mathbf{A}} = \begin{bmatrix} 5 & 1 \\ -4 & 2 \end{bmatrix}$$

Then LU decomposition

$$\underset{2\times 2}{\mathbf{A}} = \underset{2\times 2}{\mathbf{L}}\ \underset{2\times 2}{\mathbf{U}}$$

$$\begin{bmatrix} 5 & 1 \\ -4 & 2 \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix}$$

And, writing out the system of equations based on the matrix algebra, we get:

$$\begin{aligned} 5 &= l_{11}(u_{11}) + 0(0) \\ 1 &= l_{11}(u_{12}) + 0(u_{22}) \\ -4 &= l_{21}(u_{11}) + l_{22}(0) \\ 2 &= l_{21}(u_{12}) + l_{22}(u_{22}) \end{aligned}$$

Unfortunately there are more unknowns (6) than equations (4) which means the system is underdetermined. To find a unique solution, we need to add some additional constraints on the system. One way to do this is to For example, require **L** to be a *unit triangular matrix* (i.e. elements on

system. One way to do this is to For example, require **L** to be a *unit triangular matrix* (i.e. elements on the main diagonal are ones). In our example, $l_{11} = l_{22} = 1$, and our system of equations becomes:

$$
\begin{aligned}
5 &= 1(u_{11}) + 0(0) \\
1 &= 1(u_{12}) + 0(u_{22}) \\
-4 &= l_{21}(u_{11}) + 1(0) \\
2 &= l_{21}(u_{12}) + 1(u_{22})
\end{aligned}
$$

Which is now uniquely solvable (i.e., four equations; four unknowns). In our example,

$$
\begin{aligned}
l_{21} &= -0.8 \\
u_{11} &= 5 \\
u_{12} &= 1 \\
u_{22} &= 2.8
\end{aligned}
$$

Thus, the LU decomposition of **A** is,

$$
\begin{bmatrix} 5 & 1 \\ -4 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -0.8 & 1 \end{bmatrix} \begin{bmatrix} 5 & 1 \\ 0 & 2.8 \end{bmatrix}
$$

Checking our work in R, we find the solution holds.

```
# Create L and U
L = matrix(c(1, -0.8, 0, 1), nrow = 2)
U = matrix(c(5, 0, 1, 2.8), nrow = 2)

# Product
L %*% U
```

```
     [,1] [,2]
[1,]    5    1
[2,]   -4    2
```

LUP Decomposition

Unfortunately if $a_{11} = 0$ we run into problems. Since $a_{11} = l_{11}(u_{11})$ This implies that either l_{11} or u_{11} is zero. This would immediately make **L** or **U** singular (determinant is zero). This is a paradox if **A** itself is nonsingular. This problem goes away if we simply re-order/swap the rows of **A** so that the (1,1) element is non-zero. Swapping (or permuting) the rows or columns in a matrix is related to the process of *pivoting* in matrix algebra.

If we swap only the rows of **A** called, *partial pivoting*, the LU decomposition is now expressed as

$$
PA = LU
$$

$$\mathbf{PA} = \mathbf{LU}$$

where **P** is a permutation matrix which after pre-multiplying by **A** permutes the rows of **A**. This is called LUP decomposition. Since all square matrices can be decomposed in this form, LUP decomposition a useful technique in practice.

We can carry out LUP decomposition using the `lu()` function from the **Matrix** package. This function outputs a list with the **L**, **U**, and **P** matrices.

```
# Create A
A = matrix(c(5, -4, 1, 2), nrow = 2)

# Load Matrix library
library(Matrix)

# PLU decomposition
plu_decomp = lu(A)

# View results
expand(plu_decomp)
```

```
$L
2 x 2 Matrix of class "dtrMatrix" (unitriangular)
     [,1] [,2]
[1,]  1.0    .
[2,] -0.8  1.0

$U
2 x 2 Matrix of class "dtrMatrix"
     [,1] [,2]
[1,]  5.0  1.0
[2,]    .  2.8

$P
2 x 2 sparse Matrix of class "pMatrix"

[1,] | .
[2,] . |
```

Here the results are,

$$\mathbf{L} = \begin{bmatrix} 1 & 0 \\ -0.8 & 1 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 5 & 1 \\ 0 & 2.8 \end{bmatrix}$$

The notation in the permutation matrix, **P**, indicates that this is essentially an identity matrix. To double-check that the decomposition worked, we compute **PLU** and see if we re-obtain **A**. We use the

`expand()` function along with the `$` notation to access each element of the output.

```
# Compute PLU
expand(plu_decomp)$P %*% expand(plu_decomp)$L  %*% expand(plu_decomp)$U
```

```
2 x 2 Matrix of class "dgeMatrix"
     [,1] [,2]
[1,]    5    1
[2,]   -4    2
```

There are other methods of LU decomposition, including **full pivoting**, in which case we permute both rows and columns in **A**. There is also LDU decomposition in which **A** is decomposed into unit triangular matrices **L** and **U**, and diagonal matrix **D**.

Solving Systems of Equations with the LU Decomposition

Imagine if we wanted to solve a set of simultaneous equations to compute unknown values of **B** using our matrix **A**, and known values for **Y**, say,

$$\mathbf{AB} = \mathbf{Y}$$

To find the elements of **B** we would need the inverse of our matrix **A**. However, we know have an alternative solution from the LU decomposition. Since $\mathbf{A} = \mathbf{LU}$, we can re-write our simultaneuous equations as:

$$\mathbf{LUB} = \mathbf{Y}$$

Pre-multiplying this by **L** inverse, we get

$$\mathbf{L}^{-1}\mathbf{LUB} = \mathbf{L}^{-1}\mathbf{Y}$$
$$\mathbf{UB} = \mathbf{L}^{-1}\mathbf{Y}$$

Let us call the right-hand side of this equation **Z**. This gives us two sets of equations related to **Z**:

$$\mathbf{UB} = \mathbf{Z}$$
$$\mathbf{L}^{-1}\mathbf{Y} = \mathbf{Z}$$

The second equation, we can also express (after pre-multiplying by **L**) as $\mathbf{LZ} = \mathbf{Y}$. Thus we now have the two equations that are now based on the matrices **L** and **U** from our decomposition.

$$\mathbf{UB} = \mathbf{Z}$$
$$\mathbf{LZ} = \mathbf{Y}$$

$$\mathbf{LZ} = \mathbf{Y}$$

Remember, the goal was to solve for **B**, so to do this, we first solve the second equation, $\mathbf{LZ} = \mathbf{Y}$, for **Z** (which is unknown), and then use that to solve the first equation for **B**. Let's see it in action using our example. To do so, lets solve this set of simultaneous equations:

$$\underset{\mathbf{A}}{\begin{bmatrix} 5 & 1 \\ -4 & 2 \end{bmatrix}} \underset{\mathbf{B}}{\begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix}} = \underset{\mathbf{Y}}{\begin{bmatrix} 1 \\ -3 \end{bmatrix}}$$

Step 1: Solve for Z

$$\mathbf{LZ} = \mathbf{Y}$$

$$\begin{bmatrix} 1 & 0 \\ -0.8 & 1 \end{bmatrix} \begin{bmatrix} z_{11} \\ z_{21} \end{bmatrix} = \begin{bmatrix} 1 \\ -3 \end{bmatrix}$$

The two equations from this multiplication are:

$$z_{11} = 1$$
$$-0.8z_{11} + z_{21} = -3$$

The triangular matrix makes this really easy to solve these equations for the two elements of **Z**. Here, $z_{11} = 1$ and $z_{21} = -2.2$. Next, we substitute these into **Z** in the second equation.

Step 2: Solve for B

$$\mathbf{UB} = \mathbf{Z}$$

$$\begin{bmatrix} 5 & 1 \\ 0 & 2.8 \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} = \begin{bmatrix} 1 \\ -2.2 \end{bmatrix}$$

The two equations from this multiplication are:

$$5(b_{11}) + b_{21} = 1$$
$$2.8(b_{21}) = -2.2$$

Again, the triangular matrix makes this really easy to solve these equations for the two elements of **B**. After carrying out the algebra,

$$\mathbf{B} \sim \begin{bmatrix} 0.36 \end{bmatrix}$$

$$\mathbf{B} \sim \begin{bmatrix} -0.79 \end{bmatrix}$$

The exciting thing here is that we have solved for **B** without ever finding the inverse of the **A** matrix! It turns out that this is also a much more computationally efficient method to solve systems of equations. (Even though it maybe didn't feel like it when we used a 2x2 matrix.)

In OLS regression our goal is to estimate regression coefficients from a data matrix and vector of outcomes. It is exactly this problem, taking a system of equations,

$$\mathbf{X}\boldsymbol{\beta} = \mathbf{Y}$$

and solving for $\boldsymbol{\beta}$. The $\mathbf{X}$ matrix can always be made square by multiplying by the transpose. And once this is square, we can simply decompose $\mathbf{X}^\intercal\mathbf{X}$ and then use the same methodology to compute the elements of $\boldsymbol{\beta}$.

Why Use Decomposition Rather than Compute an Inverse?

Unfortunately computational functions that are used to compute inverses of matrices are typically numerically unstable. To demonstrate what this means, we will construct a simulated data set of $n = 50$ cases:

```
# Number of cases
n = 50

# Create 50 x-values evenly spread b/w 1 and 500
x = seq(from = 1, to = 500, len = n)

# Create X matrix
X = cbind(1, x, x^2, x^3)
colnames(X) <- c("Intercept", "x", "x2", "x3")

# Create beta matrix
beta <- matrix(c(1, 1, 1, 1), nrow = 4)

# Create vector of y-values
set.seed(1)
y = X %*% beta + rnorm(n, mean = 0, sd = 1)
```

What happens if we try to use `solve()` to find the inverse of the $\mathbf{X}^\intercal\mathbf{X}$ matrix to obtain the regression coefficient estimates?

```
solve(crossprod(X)) %*% crossprod(X,y)
```

```
Error in solve.default(crossprod(X)): system is computationally singular: reciprocal conditi
```

The standard R function for inverse gives an error. To see why this happens, we can take a closer look at the $\mathbf{X}^\intercal\mathbf{X}$ matrix. Here we will examine these values:

```
# Set the number of digits
options(digits = 4)

# Compute X^TX matrix
crossprod(X)
```

```
          Intercept         x        x2        x3
Intercept 5.000e+01 1.253e+04 4.217e+06 1.597e+09
x         1.253e+04 4.217e+06 1.597e+09 6.454e+11
x2        4.217e+06 1.597e+09 6.454e+11 2.716e+14
x3        1.597e+09 6.454e+11 2.716e+14 1.176e+17
```

Note the difference of several orders of magnitude. On a computer, we have a limited range of numbers. This makes some numbers behave like 0, when we also have to consider very large numbers. This in turn leads to what is essentially division by 0, which produces errors. Solving systems via decomposition fixes this problem!

QR Decomposition

Another decomposition method is QR decomposition. QR decomposition does not require that the decomposition be carried out on a square nor symmetric matrix . Similar to LU decomposition, QR decomposition results in factoring matrix **A** into the product of two matrices, namely **Q** and **R**.

$$\underset{m\times n}{\mathbf{A}} = \underset{m\times m}{\mathbf{Q}}\ \underset{m\times n}{\mathbf{R}}$$

where **Q** is an *orthogonal matrix* (its columns are orthogonal unit vectors) which implies that $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$ or that $\mathbf{Q}^\intercal = \mathbf{Q}^{-1}$. **R** is an upper-triangular matrix. Although there is a way to hand-calculate the matrices **Q** and **R** (e.g., using the Gram-Schmidt process), we will rely on computation.

To carry out a QR decomposition we will use the `qr()` function which returns a list of output related to the QR decomposition. To extract the actual **Q** and **R** matrices from this output, we will use the `qr.Q()` and `qr.R()` functions, respectively.

```
# Create matrix A
A = matrix(c(5, -4, 1, 2), nrow = 2)


# Carry out QR decomposition
qr_decomp = qr(A)

# View Q matrix
qr.Q(qr_decomp)
```

```
        [,1]   [,2]
[1,] -0.7809 0.6247
[2,]  0.6247 0.7809
```

```
# View R matrix
qr.R(qr_decomp)
```

```
      [,1]   [,2]
[1,] -6.403 0.4685
[2,]  0.000 2.1864
```

You can see that **R** is an upper-triangular matrix, and we can check that **Q** is orthonormal by testing whether $\mathbf{Q}^{\mathsf{T}} = \mathbf{Q}^{-1}$.

```
# Q^T
t(qr.Q(qr_decomp))
```

```
        [,1]   [,2]
[1,] -0.7809 0.6247
[2,]  0.6247 0.7809
```

```
# Q^-1
solve(qr.Q(qr_decomp))
```

```
        [,1]   [,2]
[1,] -0.7809 0.6247
[2,]  0.6247 0.7809
```

We can also check to see that the product of the decomposed matrices is **A**.

```
# A = QR?
qr.Q(qr_decomp) %*% qr.R(qr_decomp)
```

```
     [,1] [,2]
[1,]    5    1
[2,]   -4    2
```

We could use the two decomposed matrices to again compute the elements of **B** in the same way we did for the LU decomposition.

QR decomposition is how the `lm()` function computes the regression coefficients.

Singular Value Decomposition (SVD)

A third decomposition method that is worth knowing about is singular value decomposition. This decomposition method is commonly used for data compression or variable reduction, and plays a

large role in machine learning applications. SVD decomposes a matrix into the product of three matrices:

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times n}{\mathbf{U}} \; \underset{n \times n}{\mathbf{S}} \; \underset{n \times n}{\mathbf{V}^\intercal}$$

where, **U** and **V** are an orthogonal matrices and **S** is a diagonal matrix. From the properties of the transpose, we know that,

$$\mathbf{A}^\intercal = \mathbf{V}\mathbf{S}^\intercal\mathbf{U}^\intercal$$

And for mathematical convenience, we can take advantage that **U** and **V** are an orthogonal matrices ($\mathbf{U}^\intercal\mathbf{U} = \mathbf{V}^\intercal\mathbf{V} = \mathbf{I}$) by expressing two equations:

$$\begin{aligned}\mathbf{A}\mathbf{A}^\intercal\mathbf{U} &= \mathbf{U}\mathbf{S}\mathbf{V}^\intercal\mathbf{V}\mathbf{S}^\intercal\mathbf{U}^\intercal\mathbf{U} \\ &= \mathbf{U}\mathbf{S}^2\end{aligned}$$

And,

$$\begin{aligned}\mathbf{A}^\intercal\mathbf{A}\mathbf{V} &= \mathbf{V}\mathbf{S}^\intercal\mathbf{U}^\intercal\mathbf{U}\mathbf{S}\mathbf{V}^\intercal\mathbf{V} \\ &= \mathbf{V}\mathbf{S}^2\end{aligned}$$

These two equations are called *eigenvalue* equations, which show up all over the place in statistical work. These are easy to solve by computation. For example, we would solve the second eigenvalue equation to find **V** and **S** and then find **U** by

$$\mathbf{U} = \mathbf{A}\mathbf{V}\mathbf{S}^{-1}$$

In practice, we will use the `svd()` function.

```
# Create A
A = matrix(c(5, -4, 1, 2), nrow = 2)

# Singular value decomposition
sv_decomp = svd(A)

# View results
sv_decomp
```

```
$d
[1] 6.422 2.180

$u
        [,1]   [,2]
[1,] -0.7630 0.6464
[2,] -0.6464 0.7630
```

```
[2,]  0.6464 0.7630

$v
         [,1]    [,2]
[1,] -0.99659 0.08248
[2,]  0.08248 0.99659
```

The resulting decomposition is:

$$\mathbf{U} = \begin{bmatrix} -0.76 & 0.65 \\ 0.65 & 0.76 \end{bmatrix}$$

$$\mathbf{D} = \begin{bmatrix} 6.42 & 0 \\ 0 & 2.18 \end{bmatrix}$$

$$\mathbf{V} = \begin{bmatrix} -1.00 & 0.08 \\ 0.08 & 1.00 \end{bmatrix}$$

Optional: Cholsky Decomposition

Another commonly used decomposition method is Cholsky decomposition. This method decomposes a symmetric matrix **A** into the product of two matrices, $\mathbf{L}$ (a lower-triangular matrix) and $\mathbf{L}^*$ (the *conjugate transpose* of $\mathbf{L}$). The conjugate transpose is computed by taking the transpose of a matrix and then finding the *complex conjugate* of each element in the matrix.

To understand what a complex conjugate is, we first remind you of the idea of a complex number. Remember that all real and imaginary numbers are complex numbers which can be expressed as $a + bi$, where *a* is the real part of the number and *b* is the imaginary part of the number, and *i* is the square root of -1. For example the number 2 can be expressed as $2 + 0i$.

The complex conjugate of a number is itself a complex number that has the exact same real part and an imaginary part equal in magnitude but opposite in sign. For example the complex conjugate for the number $3 + 2i$ is $3 - 2i$.

Note that the complex conjugate of a real number is just the real number, since $a + 0i = a - 0i = a$.
As an example, say that matrix **A** was

$$\mathbf{A} = \begin{bmatrix} 1 & 3+i & -2+3i \\ 0 & 4 & 0-i \end{bmatrix}$$

The conjugate transpose of **A**, symbolized as $\mathbf{A}^*$, can be found by first computing the transpose of **A**, and then finding the complex conjugate of each element in the transpose.

$$\mathbf{A}^\intercal = \begin{bmatrix} 1 & 0 \\ 3+i0 & 4 \\ -2+3i & 0-i \end{bmatrix}$$

And,

$$\mathbf{A}^* = \begin{bmatrix} 1 & 0 \\ 3 - i0 & 4 \\ -2 - 3i & 0 + i \end{bmatrix}$$

Say we wanted to compute a Cholsky decomposition on a 2x2 symmetric matrix:

$$\underset{2\times2}{\mathbf{A}} = \begin{bmatrix} 5 & -4 \\ -4 & 5 \end{bmatrix}$$

$$\underset{2\times2}{\mathbf{A}} = \underset{2\times2}{\mathbf{L}}\ \underset{2\times2}{\mathbf{L}^*}$$

$$\begin{bmatrix} 5 & -4 \\ -4 & 5 \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} \\ 0 & l_{22} \end{bmatrix}$$

Carrying out the matrix algebra we have the following three unique equations:

$$\begin{aligned} 5 &= l_{11}(l_{11}) \\ -4 &= l_{11}(l_{21}) \\ 5 &= l_{21}(l_{21}) + l_{22}(l_{22}) \end{aligned}$$

Since we have three equations with three unknowns we can solve for each element in **L**.

$$\begin{aligned} l_{11} &= \sqrt{5} \approx 2.24 \\ l_{21} &= \frac{-4}{\sqrt{5}} \approx -1.79 \\ l_{22} &= \sqrt{1.8} \approx 1.34 \end{aligned}$$

So the Cholsky decomposition is,

$$\begin{bmatrix} 5 & -4 \\ -4 & 5 \end{bmatrix} = \begin{bmatrix} \sqrt{5} & 0 \\ \dfrac{-4}{\sqrt{5}} & \sqrt{1.8} \end{bmatrix} \begin{bmatrix} \sqrt{5} & \dfrac{-4}{\sqrt{5}} \\ 0 & \sqrt{1.8} \end{bmatrix}$$

Or using the approximations,

$$\begin{bmatrix} 5 & -4 \\ -4 & 5 \end{bmatrix} \approx \begin{bmatrix} 2.24 & 0 \\ -1.79 & 1.34 \end{bmatrix} \begin{bmatrix} 2.24 & -1.79 \\ 0 & 1.34 \end{bmatrix}$$

We can use the `chol()` function to compute the Cholsky decomposition.

```
# Create A
A = matrix(c(5, -4, -4, 5), nrow = 2)

# Cholsky decomposition
cholsky_decomp = chol(A)

# View results
cholsky_decomp
```

```
     [,1]   [,2]
[1,] 2.236 -1.789
[2,] 0.000  1.342
```

Note that the output from `chol()` is actually $\mathbf{L}^*$. To obtain the **L**, we need to take the transpose of this output. We can also check that the decomposition worked.

```
# Check results LL*
t(cholsky_decomp) %*% cholsky_decomp
```

```
     [,1] [,2]
[1,]    5   -4
[2,]   -4    5
```

🚧 CAUTION: The `chol()` function does not check for symmetry. If you use the function to decompose a nonsymmetric matrix, the results will likely be meaningless. For example, if we compute the decomposition with our original example matrix **A**, we get the following.

```
# Create A
A = matrix(c(5, -4, 1, 2), nrow = 2)

# Cholsky decomposition
chol(A)
```

```
     [,1]   [,2]
[1,] 2.236 0.4472
[2,] 0.000 1.3416
```

Checking this we do not get back **A**

```
# Check results
t(chol(A)) %*% chol(A)
```

```
     [,1] [,2]
[1,]    5    1
[2,]    1    2
```

[2,] 1 2

For symmetric matrices, the Cholsky decomposition method is much more computationally efficient than the LU method. Similar to the LU method, there are variations on the Cholsky decomposition method. Two popular variants include LDL and LDLT decomposition.