The Math Behind PCA

David T. Harvey* Bryan A. Hanson†

2022-02-01

Contents

This vignette is based upon `LearnPCA` version 0.1.1.

`LearnPCA` provides the following vignettes:

- Start Here
- A Conceptual Introduction to PCA
- Step By Step PCA
- Understanding Scores & Loadings
- Visualizing PCA in 3D
- The Math Behind PCA
- PCA Functions
- Alternatively, if you are offline or prefer accessing the vignettes with `R`, simply type `browseVignettes("LearnPCA")` to get a clickable list in a browser window.

*Professor of Chemistry & Biochemistry, DePauw University, Greencastle IN USA., harvey@depauw.edu
†Professor Emeritus of Chemistry & Biochemistry, DePauw University, Greencastle IN USA., hanson@depauw.edu

Vignettes are available in both pdf and html formats. However, the pdf versions lack certain interactive diagrams, so the html versions are recommended.

In this vignette we'll look closely at how the data reduction step in PCA is actually done. This vignette is intended for those who really want to dig deep. It's helpful if you know something about matrix manipulations, but we work hard to keep the level of the material accessible to those who are just learning.

1 Introduction

If you have read the Step By Step PCA vignette, you know that the first steps in PCA are:

1. Center the data by subtracting the column means from the columns.
2. Optionally, scale the data column-wise.
3. Carry out the reduction step, typically using `prcomp`.

For a data matrix with n rows of observations/samples and p variables/features, the results are the

- scores matrix with n rows and p columns, where each column corresponds to a principal component and the values are the scores, namely the positions of the samples in the new coordinate system.
- loadings matrix with p rows and p columns, which represent the contributions of each variable to each principal component.

In the Step By Step PCA vignette we also showed how to reconstruct or approximate the original data set by multiplying the scores by the transpose of the loadings.

In Figure 1 we show one way to represent the relationships between the original data matrix (**X**), the loadings matrix (**L**) and the scores matrix (**S**).

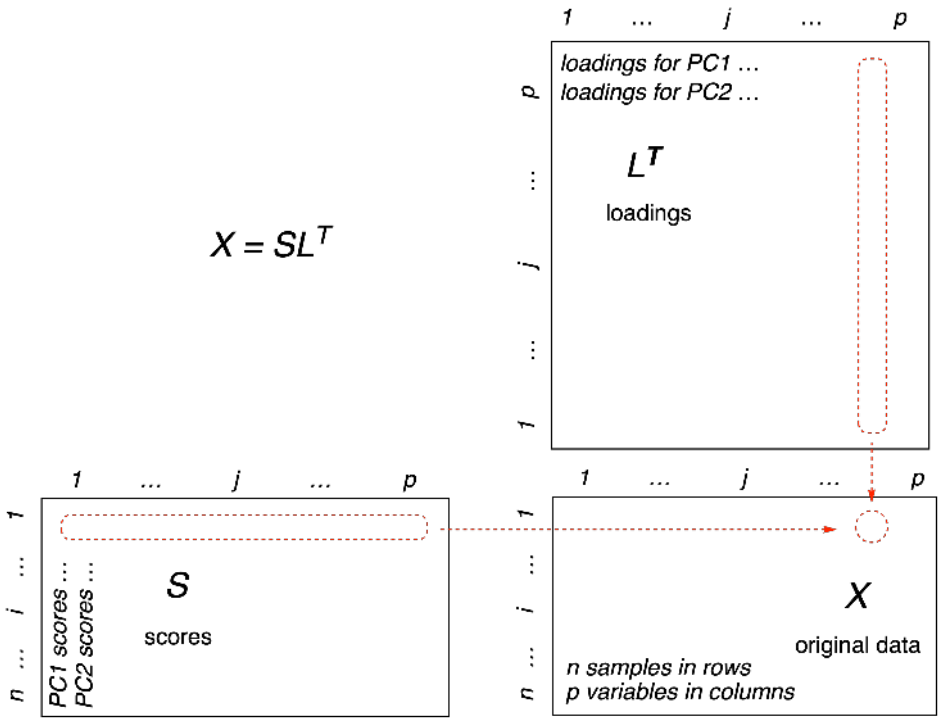

Figure 1: One way to look at the matrix algebra behind PCA. Reconstruction of the data matrix **X** is achieved by multiplying the score matrix (**S**) by the transpose of the loadings matrix (**L**). The method of matrix multiplication is symbolized in the red-dotted outlines: Each element of row i of the scores matrix is multiplied by the corresponding element of column j of the transposed loadings. These results are summed to give a single entry in the original data matrix $\mathbf{X}_{ij}$.

Figure 1 demonstrates that if we multiply the scores matrix ($\mathbf{S}$) by the transpose of the loadings matrix ($\mathbf{L^T}$), we get back the original data matrix ($\mathbf{X}$). When we start however, all we have is the original data matrix, how do we get the other two matrices? If we think of this as an algebra problem, we seem to be missing some variables; computing the loadings and scores matrices seems like it would be impossible, as there is not enough information. However, this is not a algebra problem, it is a *linear* algebra problem (linear algebra being the study of matrices). It *is* possible to determine the answer, even though we seem to be missing information, as we shall see shortly. The key is in something called matrix decompositions.[1]

2 Matrix Decompositions

In a moment we are going to look at two matrix decompositions in detail, the singular value decomposition (SVD) and the eigenvalue decomposition. These decompositions are representative of roughly a dozen matrix decompositions. A matrix decomposition or factorization breaks a matrix into pieces in such a way as to extract information and solve problems. The SVD is probably the most powerful decomposition there is – we will make extensive use of this insightful Twitter thread by Dr. Daniela Witten of the University of Washington. We also use this excellent Cross Validated answer by amoeba.

A short note however, before we dig deeper. Since we are using a computer to solve this problem, we need to keep in mind that using a computer is not quite the same as solving a problem using pencil and paper. On the computer, we usually have choices of algorithms to solve problems. Some algorithms are more robust than others. Algorithms need to take into account edge cases where the computation can become unstable. For instance, computers can only store numbers to a certain level of accuracy: when is "very small" actually zero in practice? We need to know this so we don't try to divide by zero. This is only one example of problems that can arise when using a computer to calculate values.

Finally, the two decompositions we are going to look at have something in common. Our approach will be to explain each without reference to the other, as this facilitates digestion and understanding of each (it doesn't seem fair to require you to understand the 2nd one that you haven't read while trying to understand the first one). Then, if you are still with us, we'll look at what they have in common.

3 The SVD Decomposition

We'll start from the original data matrix $\mathbf{X}$ which has samples in rows and measured variables in columns. Let's assume that we have column-centered the matrix. The SVD decomposition breaks this matrix $\mathbf{X}$ into three matrices (dimensions in parentheses):[2]

$$\mathbf{X}_{(n\ \times\ p)} = \mathbf{U}_{(n\ \times\ p)}\mathbf{D}_{(p\ \times\ p)}\mathbf{V}^T_{(p\ \times\ p)} \tag{1}$$

And here's the equation without matrix dimensions. Remember that $\mathbf{V}^T$ means "take the transpose" of $\mathbf{V}$, interchanging rows and columns.

$$\mathbf{X} = \mathbf{UDV}^T \tag{2}$$

The equation above is where we'd like to end up. How can we get there? Let's start with *What are these matrices*?

- $\mathbf{X}$ contains the original data
- The columns of $\mathbf{U}$ are vectors giving the principal axes. These define the new coordinate system.
- The scores can be obtained by $\mathbf{XV}$; scores are the projections of the data on the principal axes.

[1] If you need an introduction to linear algebra, there are many good books, but we particularly recommend Singh (2014) or Savov (2020).

[2] This treatment is the "compact SVD" case.

- **D** is a diagonal matrix, which means all non-diagonal elements are zero. The diagonal contains positive values sorted from largest to smallest. These are called the singular values.[3]
- The columns of **V** are the PCA loadings

In addition, **U** and **V** are semi-orthogonal matrices,[4] which means that when pre-multiplied by their transpose one gets the identity matrix:[5]

$$\mathbf{U}^T\mathbf{U} = \mathbf{I} \tag{3}$$

$$\mathbf{V}^T\mathbf{V} = \mathbf{I} \tag{4}$$

We'll use this fact to great advantage when we implement a simple version of SVD in a moment.

4 A Simple Implementation of SVD

Now that we know what these matrices are, we can look into how to compute them. As mentioned earlier, the algorithm is everything here. As a simple example, we'll look at an approach called "power iteration." This is by no means the best approach, but it is simple enough that we can understand the idea. First, let's generate some random data. In this simple example we are only going to compute the first principal component, so `V` is a vector, not a matrix (one can still do matrix multiplication with a vector, we just treat it as a "row vector" or a "column vector").[6] Note that the dimensions of the variables are chosen so that we can matrix multiply them (they are *conformable* matrices).

```
set.seed(30)
X <- matrix(rnorm(100*50), ncol = 50)
V <- rnorm(50)
```

Next, we'll use the built-in function `svd` to compute the "official" answer for comparison to our results.

```
X_svd <- svd(X)
```

We'll do a simple iterative calculation that computes the values of `U` and `V`. The loop runs for 50 iterations, and as it does the values of `U` and `V` are continuously updated and get closer to the actual answer.

```
for (iter in 1:50) {
  U <- X %*% V # Step 1
  U <- U/sqrt(sum(U^2)) # Step 2
  V <- t(X) %*% U # Step 3
  V <- V/sqrt(sum(V^2)) # Step 4
 if ((iter %% 10) == 0L) { # report every 10 steps; print the correlation between
   cat("\nIteration", iter, "\n") # the current U or V and the actual values from SVD
   cat("\tcor with V:", sprintf("%f", cor(X_svd$v[,1], V)), "\n")
   cat("\tcor with U:", sprintf("%f", cor(X_svd$u[,1], U)), "\n")
   }
}
```

[3] The singular values are used to calculate the variance explained by each principal component. We'll have more to say about that later.

[4] For semi-orthogonal matrices (or orthogonal matrices for that matter), $A^TA = AA^T = I$. Semi-orthogonal matrices are rectangular. For non-rectangular orthogonal matrices, if $n > p$, the dot product of any column with itself is 1, and the dot product of any column with a different column is zero. If $n < p$, then it is the rows rather than the columns that are relevant. See the Wikipedia article.

[5] The identity matrix **I** is a square matrix with ones on the diagonal and zeros everywhere else. The identity matrix can pre- or post-multiply any other matrix and not affect that matrix.

[6] When refering to the mathematical equations, we'll use **X**, but when referencing the values we compute, we'll use `X`.

```
##
## Iteration 10
##  cor with V: -0.931153
##  cor with U: -0.917383
##
## Iteration 20
##  cor with V: -0.998758
##  cor with U: -0.998542
##
## Iteration 30
##  cor with V: -0.999969
##  cor with U: -0.999965
##
## Iteration 40
##  cor with V: -0.999999
##  cor with U: -0.999999
##
## Iteration 50
##  cor with V: -1.000000
##  cor with U: -1.000000
```

Notice that there is no **D** matrix in this calculation. This is because we are only calculating a single principal component, and therefore in this case **D** is a scalar constant. We can drop it from the calculation. With that simplification, we can look at each step.

4.1 Step 1

The first step is to multiply the data matrix `X` by the initial estimate for `V` (remember at each iteration the estimate gets better and better). How does this relate to Equation (2)? If we drop **D** from equation (2) we have:

$$\mathbf{X} = \mathbf{U}\mathbf{V}^T \tag{5}$$

If we right multiply both sides by **V** we have:

$$\mathbf{X}\mathbf{V} = \mathbf{U}\mathbf{V}^T\mathbf{V} \tag{6}$$

which evaluates to:

$$\mathbf{X}\mathbf{V} = \mathbf{U}\mathbf{I} = \mathbf{U} \tag{7}$$

because **V** is a semi-orthogonal matrix. This is the line in the code.

4.2 Step 2

In this step we normalize (regularize, or scale) the estimate of `U`, by dividing by the square root of the sum of the squared values in `U`.[7] This has the practical effect of keeping the values in `U` from becoming incredibly large and possibly overflowing memory.[8]

[7] This is the L^2 or Euclidean norm, generally interpreted as a length.

[8] As the values grow larger and larger, first we lose precision and eventually the numbers become too big to store.

4.3 Step 3

Here we update our estimate of `V`. Similar to Step 1, we can rearrange Equation (2), this time by dropping $\mathbf{D}$, pre-multiplying both sides by $\mathbf{U}^T$ to give an identity matrix which drops out, and then transposing both sides:

$$\mathbf{U}^T\mathbf{X} = \mathbf{U}^T\mathbf{U}\mathbf{V}^T \tag{8}$$

$$\mathbf{U}^T\mathbf{X} = \mathbf{I}\mathbf{V}^T \tag{9}$$

$$\mathbf{U}^T\mathbf{X} = \mathbf{V}^T \tag{10}$$

$$(\mathbf{U}^T\mathbf{X})^T = (\mathbf{V}^T)^T \tag{11}$$

$$\mathbf{X}^T\mathbf{U} = \mathbf{V} \tag{12}$$

which is the operation we see in the code snippet.[9]

4.4 Step 4

Step 4 is the same operation as in Step 2, but on `V`.

4.5 Overall

Essentially what this algorithm is doing is alterating between the two calculations (for `U`, steps 1 & 2, then for `V` steps 3 & 4), with `X` constant. At each iteration these estimates improve, moving from the initial random value of `V` towards the best answer for both `V`and `U`.

4.6 Reporting

You'll notice that the code snippet above has a few lines to report the progress of the calculation. Every 10 steps the correlation between the current value of `V` with the official answer contained in `X_svd$v` is displayed (and the same for `U`). As you can see from the output the correlation is not bad after 10 iterations and only improves with more iterations. We report the correlation because the signs of the power iteration answers may vary from those computed by `svd`.[10]

4.7 Comparison to the Answer from `svd`

Let's compare the absolute values of the two different answers:

```
mean(abs(V) - abs(X_svd$v[,1]))
```

```
## [1] 4.735975e-06
```

```
mean(abs(U) - abs(X_svd$u[,1]))
```

```
## [1] 2.752052e-06
```

As you can see, the values are essentially the same except for sign.

[9] The operation from equation (11) to (12) is based upon the following property in linear algebra: $(AB)^T = B^T A^T$.

[10] From `?prcomp` "The signs of the columns of the rotation matrix are arbitrary, and so may differ between different programs for PCA, and even between different builds of R." From `?princomp` "The signs of the columns of the loadings and scores are arbitrary, and so may differ between different programs for PCA, and even between different builds of R: fix_sign = TRUE alleviates that." We discuss the origin of the different signs in more detail in PCA Functions vignette.

4.8 Comparison to the Answer from `prcomp`

We have seen that our estimate of `U` and `V` compared well to the results from the function `svd`. In practice users would probably use `prcomp` for PCA. So let's compare to the results from `prcomp`.

```
PCA <- prcomp(X)
mean(abs(X %*% V) - abs(PCA$x[,1])) # compare the scores
```

```
## [1] 0.004591257
```

```
mean(abs(V)- abs(PCA$rotation[,1])) # compare the loadings
```

```
## [1] 0.0001339306
```

These mean differences are very small but not quite as good as using `svd` directly.

Notice that we have worked through the SVD without mentioning eigen-anything. That was one of our goals. Now let's take a different point of view.

5 The Eigen Decomposition

The eigen decomposition is another way to decompose a data matrix. This decomposition breaks the data matrix $\mathbf{X}$ into two matrices.[11] Again, let's assume $\mathbf{X}$ has been centered.

$$\mathbf{X}_{(n \times n)} = \mathbf{Q}_{(n \times n)} \mathbf{\Lambda}_{(n \times n)} \mathbf{Q}^T_{(n \times n)} \tag{13}$$

And here's the equation without matrix dimensions.[12]

$$\mathbf{X} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T \tag{14}$$

The equation above is where we'd like to end up; it looks like a variation on the equation for SVD. How can we get there? Once again, let's take inventory of the matrices in the equation.

- $\mathbf{X}$ is the original data matrix. Notice the dimensions are $n \times n$, unlike in SVD. In other words, it is a square matrix. Your data is not square you say? Eigen decomposition requires an square matrix. Fortunately, there's a simple fix for this. We can work instead with the covariance matrix, which is square: $\mathbf{X}^T\mathbf{X}/(n-1)$. This retains all the structure of the original data.[13]
- $\mathbf{Q}$ is square matrix that whose columns will contain the *eigenvectors*. $\mathbf{Q}$ is also an orthogonal matrix (discussed earlier in the SVD section). Notice that $\mathbf{Q}$ appears twice in (14), the second time as its transpose.
- $\mathbf{\Lambda}$ (upper case Greek letter Lambda) is a diagonal matrix, very similar in function to $\mathbf{D}$ in SVD. All non-diagonal elements are zero. The diagonal contains values sorted from largest to smallest. These are called the *eigenvalues*.

So what are eigenvectors and eigenvalues? The eigenvalues are related to the amount of variance explained for each principal component. The eigenvectors are the principal axes, which as we have seen constitute a new coordinate system for looking at the data (see the Visualizing PCA in 3D vignette for details). If we post-multiply the original data by the eigenvectors in $\mathbf{Q}$ we get the scores:

$$\mathbf{XQ} = scores \tag{15}$$

And the loadings are simply the eigenvectors in $\mathbf{Q}$.

[11] Eigenvalues and eigenvectors are extremely important in linear algebra. The concept is usually introduced in the simpler form $Au = \lambda u$.

[12] One also sees this written $\mathbf{X} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1}$. This is equivalent because for orthogonal matrices $\mathbf{A}^T = \mathbf{A}^{-1}$.

[13] We can also use the correlation matrix, which is evaluated via the same formula. For covariance, one centers the raw data columns. For correlation, one centers the raw data and then scales them by their standard deviation.

5.1 A Simple Implementation of the Eigen Decomposition

As we did for `SVD`, we can use a power iteration to compute estimates for the first eigenvector.

```
set.seed(30)
X <- matrix(rnorm(100*50), ncol = 50)
X <- cor(X)
Q <- rnorm(50)
```

Next, we'll use the built-in function `eigen` to compute the "official" answer for comparison to our results.

```
X_eig <- eigen(X)
```

A simple iterative process as we did for SVD will continuously update the value of `Q`.

```
for (iter in 1:50) {
  Q <- X %*% Q # Step 1
  Q <- Q/sqrt(sum(Q^2)) # Step 2

 if ((iter %% 5) == 0L) { # report every 5 steps; print the correlation between
   cat("\nIteration", iter, "\n") # the current Q and the actual values from SVD
   cat("\tcor with Q:", sprintf("%f", cor(X_eig$vectors[,1], Q)), "\n")
   }
}
```

```
##
## Iteration 5
##  cor with Q: 0.669676
##
## Iteration 10
##  cor with Q: 0.933282
##
## Iteration 15
##  cor with Q: 0.989643
##
## Iteration 20
##  cor with Q: 0.998310
##
## Iteration 25
##  cor with Q: 0.999698
##
## Iteration 30
##  cor with Q: 0.999942
##
## Iteration 35
##  cor with Q: 0.999988
##
## Iteration 40
##  cor with Q: 0.999997
##
## Iteration 45
##  cor with Q: 0.999999
##
## Iteration 50
##  cor with Q: 1.000000
```

How does the final estimate for `Q` compare to the official answer? We can check our result as before:

```
mean(abs(Q) - abs(X_eig$vectors[,1])) # check the loadings
```

```
## [1] -8.0837e-06
```

Good work by the power iteration!

6 The Relationship Between SVD and Eigen Decomposition

It's apparent that SVD and the eigen decomposition have a lot in common. The `R` function `prcomp` uses the `svd` function "under the hood", and the function `princomp` uses `eigen` under the hood. The vignette PCA Functions goes into greater detail about the similarities and differences between these two decompositions as implemented in `R`.

6.1 Singular Values vs Eigenvalues

We've talked about "values" in the context of each decomposition. Is this terminology accidental, or is there a relationship? If you square the singular values from SVD and divide by $n-1$, you get the eigenvalues. Here "diagonal" means take the diagonal elements of the matrix (which would be a vector of values):

$$(diagonal(\mathbf{D}))^2/(n-1) = diagonal(\mathbf{\Lambda}) \tag{16}$$

Either of these "values" can be used to compute the amount of variance explained by each principal component. Details are in the PCA Functions vignette.

6.2 Pros and Cons

- `svd` can handle rectangular matrices $n \times p$ where $n \neq p$. Either $n > p$ or $n < p$ is acceptable. On the other hand, `eigen` must have $n = p$. `prcomp` wraps `eigen` and helps the user convert their raw data matrix into a square matrix.

7 Works Consulted

In addition to references and links in this document, please see the Works Consulted section of the *Start Here* vignette for general background.

References

Savov, Ivan. 2020. *No Bullshit Guide to Linear Algebra.* 2nd ed. minireference.com.
Singh, Kuldeep. 2014. *Linear Algebra Step by Step.* Oxford University Press.