A Brain-Friendly Guide to PCA: Math, Visuals, Code

Figure 1: PCA at a glance.

Introduction

Principal Component Analysis (PCA) is a way to find the most informative directions in your data. Unlike ordinary feature selection, PCA can combine the original features into new ones that capture the main structure of the dataset.

As such, it is a very useful tool for dimensionality reduction, feature extraction, and data visualization. Many programming libraries offer PCA as a convenient out-of-the-box method, but if you want to understand what is happening behind the scenes, the mathematics can feel intimidating.

To make that easier, this post looks at the mathematical derivation of PCA and explains why it works. I also show how PCA relates to singular value decomposition (SVD) and include a short Python example to demonstrate how it can be used in practice.

I follow the excellent explanation in Pattern Recognition and Machine Learning by Bishop et al. [1], but I add more detail to make the ideas easier to follow. I also include visuals to make the explanation more intuitive.

If you are in a hurry and only want the practical steps, jump directly to Summary Compute PCA using SVD

What is PCA?

The idea of PCA is the following. You have a set of N measurements, and each measurement consists of D features which you collect in a matrix $\mathbf{X}$ see Figure 2).

Figure 2: Example dataset for PCA with N rows and D columns. The superscript indicates the observation and f_i denotes the feature. There are D features.

You put a lot of work into obtaining these measurements and you are sure that they contain valuable information.

But then you realize it's not easy at all to extract actual information from the dataset. You want to find a way to extract the most important information, and you want to do it computationally efficiently, thus you need math to the rescue, precisely PCA.

In fact, there are two ideas/approaches that both lead to PCA. Both project the data onto a lower-dimensional subspace, but they achieve this by focusing on different objectives. The **variance-maximization approach** projects onto a lower-dimensional subspace such that

Figure 3: You think your dataset is going to rule them all.

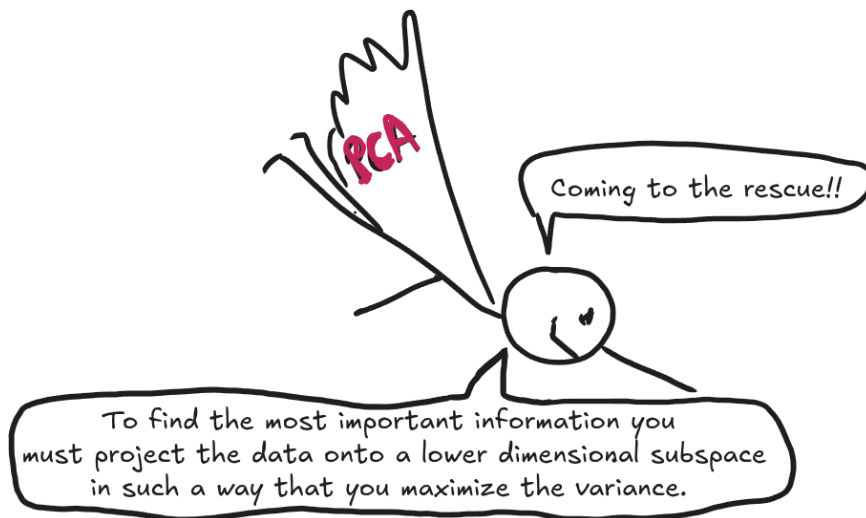

Figure 4: PCA tells us that to find the most important information, the data must be projected onto a lower-dimensional subspace in such a way that the variance of the projected data is maximized.

the variance of the projected data is maximized. The **minimum-error formulation** (I'm following Bishop's naming here) projects onto a lower-dimensional subspace such that the mean squared error between the original data and the projected data is minimized. Here, we will look at the variance-maximization approach.

Why does the variance-maximization approach work?

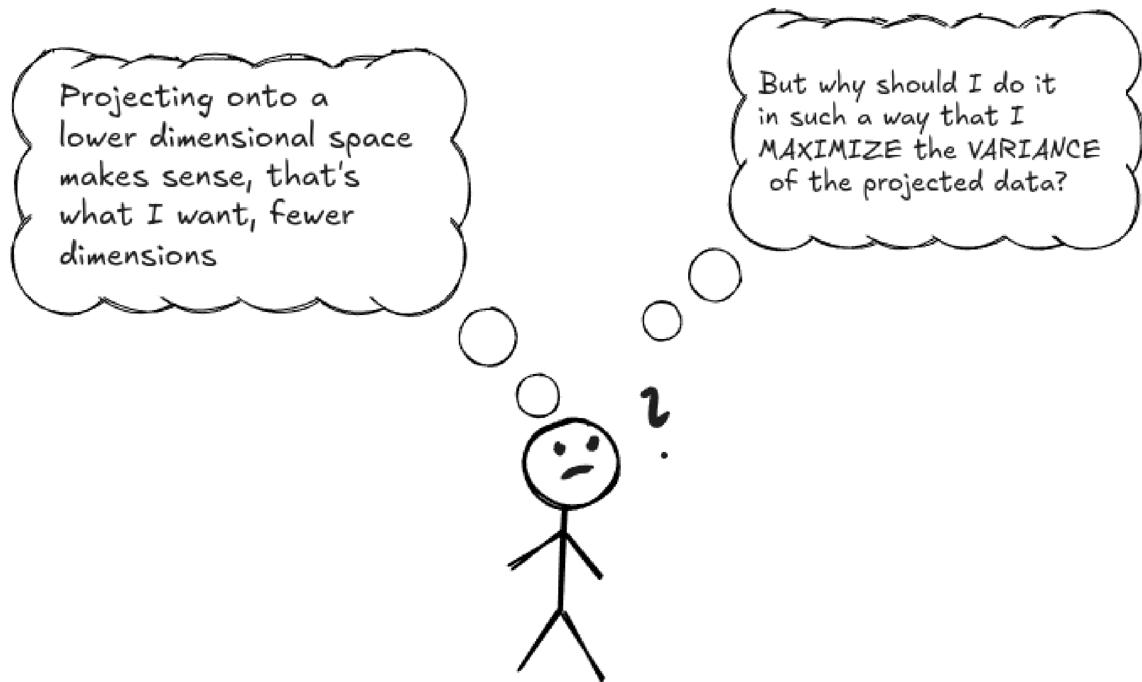

Figure 5: Why does the variance-maximization approach extract information? (image by author)

To understand why this approach works we can look at a simple example.

A Quick Refresher on Variance

Variance is a way to measure the spread of the data and is defined as the expected error of the squared distances of the data points from the mean. It's defined as:

$$Var(X) = \mathbb{E}[(X - \mu)^2] \tag{1}$$

In Figure 6 we see an example of measurements in one dimension with a low and a high variance. Imagine the data to be from repeated temperature measurements with two different thermometers. The first couple of measurements with thermometer 1 lead to the plot with the low variance and the first couple of measurements with thermometer 2 lead to the plot with the high variance. (I personally would throw that second thermometer away.)

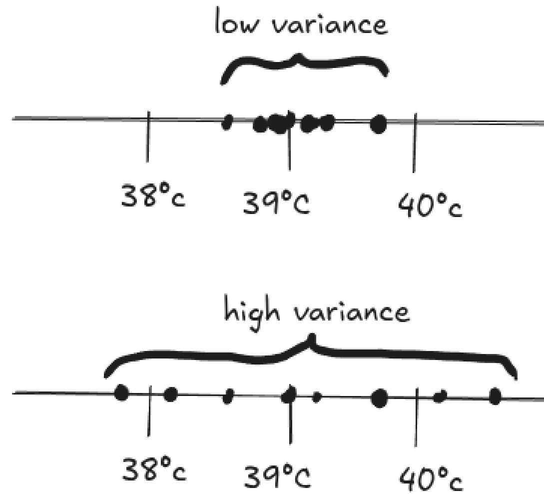

Figure 6: Data with low and high variance. (image by author)

But now to the promised example:

If our data has two features, i.e. $D=2$, then each data point can be visualized as a point in a 2D plane. If we project the data onto a lower-dimensional subspace, with $M=1<D$, we project onto a line. We could project onto *any* line, but PCA says to choose that one where our data will have the most variance. And here is visualized why this makes sense.

In Figure 7 we see 4 data points that are projected onto two different lines. The first projection (top row) is on a diagonal blue line and the second on a vertical purple line. On the far right we see the result of the projection onto both lines. If you throw away your actual data points and only keep their positions on the projected lines then the first projection (blue line) would allow you to identify each original data point. But if you only had the projections on line 2 (the purple projection line) you might not be able to distinguish between the first and second and the third and fourth data point anymore as they map to almost identical positions on the projection line. You would have lost valuable information. Thus, the projection onto the blue line is better (more informative), by "de-cluttering" the data, in other words, by maximizing their ..tatataa - variance!

I know what you are thinking now - "Ok, that's nice - but how do I find this line?"

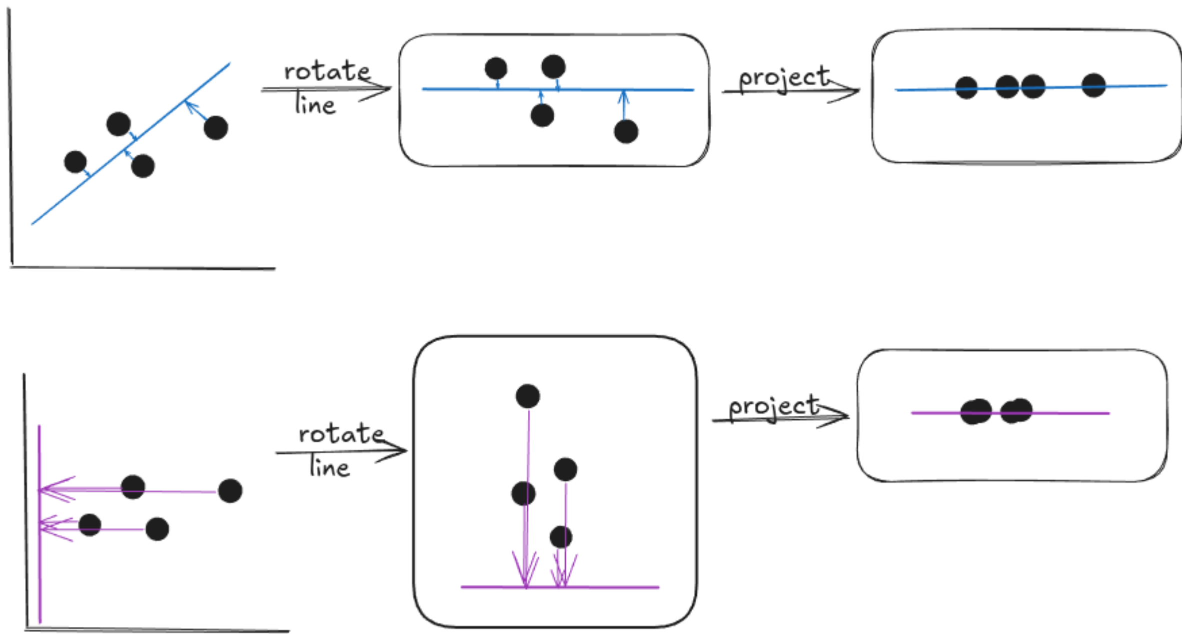

Figure 7: Projecting data onto two different lines. Each row shows one projection. Left: the data points are projected onto a line. Middle: the projection lines are rotated and displayed as horizontals. Right: only the projected coordinates on the projection lines are displayed. It can be seen that the projection on the blue line allows a clear separation of the data points whereas the projection on the purple line doesn't. Thus, the former preserves more information because it maintains more space between the data points, i.e. it has a larger variance.

How to find the projection line?

Here is an outline how we will go about it:

1) **What we want is** a line, let's call it $\mathbf{u}_1$ (a one dimensional vector in our case) and we want to project our data onto this line. The variance of the projected data onto $\mathbf{u}_1$ should be larger than it is for any other possible line.

2) **How we achieve this** is by expressing our requirements as an **optimization problem**. We must come up with an equation that gives us the variance of the projected data onto $\mathbf{u}_1$ and then we find those values for $\mathbf{u}_1$ that maximize this variance. This is the **objective function**. Here, we must construct a function of $\mathbf{u}_1$ that gives the variance: $f : \mathbf{u_1} \mapsto$ variance. Once we have this, we can use the machinery of optimization theory (thanks to all the mathematicians that developed this) to find the $\mathbf{u}_1$ that maximizes this function.

3) **Which technique exactly** are we going to use? Maximizing a function usually involves calculus. A standard approach is to take the derivative of the function with respect to the parameter we want to optimize (here, $\mathbf{u}_1$) and then set it to zero to find an optimum. Here, I (or rather Mr. Bishop) will refer to the method of **Lagrange multipliers**, which is a technique to find the local maxima and minima of a function subject to equality constraints. In our case, we will have the constraint that $\mathbf{u}_1$ must be a unit vector (i.e. it has length 1). I will **not** go into the details of the Lagrange multiplier method in this post.

A Quick refresher on dot products

According to our outline, the first task is to define a function that gives us the variance of the projected data onto $\mathbf{u}_1$. For that let's quickly recall that the dot product of two vectors $\mathbf{a}$ and $\mathbf{b}$ is the product of the length of both vectors and the cosine of the angle between them:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos(\theta)$$

Geometrically, the dot product of two vectors of length 1 tells us the component of one vector in the direction of the other. This is shown in Figure 8. On the left side of the figure, we see the definition of $cos()$ on the unit circle. On the right we see how this relates to the dot product of the unit-vectors (meaning they have length 1) $\mathbf{a}$ and $\mathbf{b}$ which take the role of the hypotenuse and the adjacent. The vector $\mathbf{a}$ is composed of the sum of the vectors $\mathbf{a}_1$ and $\mathbf{a}_2$. As you can see, $\mathbf{a}_1$ is the component (literally the component) of $\mathbf{a}$ into the direction of $\mathbf{b}$. It's also often referred to as "projection" for obvious reasons.

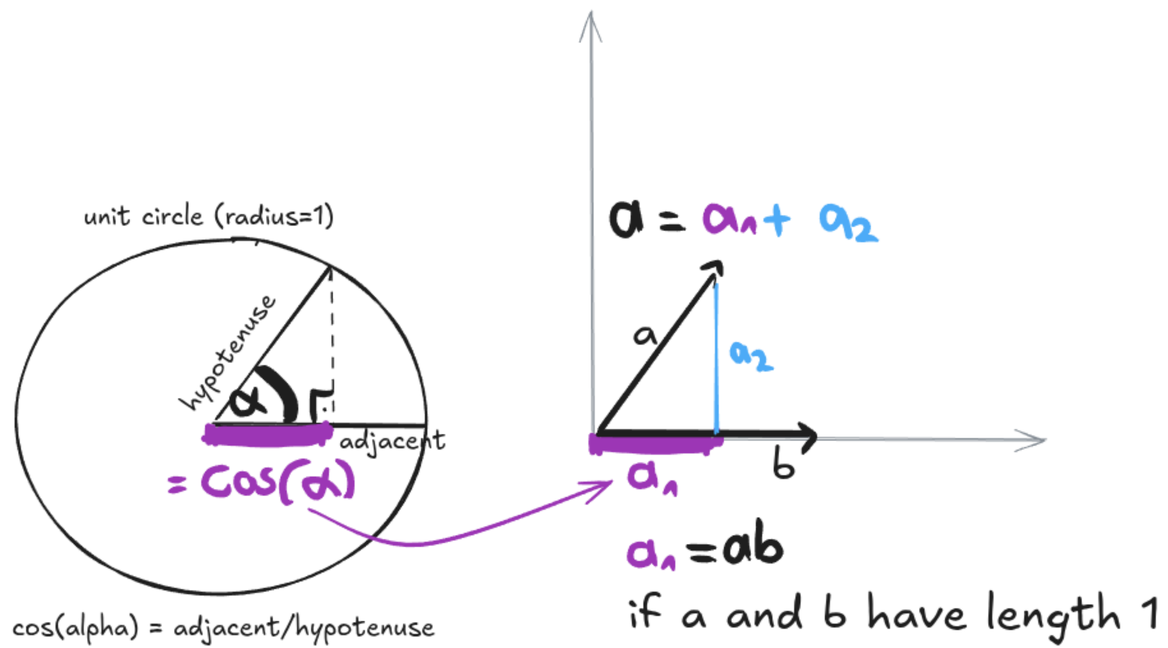

Figure 8: Showing how the definition of the dot product of two unit vectors leads to the conclusion that the dot product is the component of one vector into the direction of the other. (image by author)

The objective function

Now that we have the dot product out of the way we can start defining $\mathbf{u}_1$. Recall, that $\mathbf{u}_1$ is a unit vector. It indicates the direction of the line onto which we want to project our data.

We can take the dot product of a data point $\mathbf{x}^{(i)}$ and the direction vector which gives $\mathbf{u}_1^T\mathbf{x}^{(i)}$. The result is scalar (a single number), and it is the component of the data point into the direction of $\mathbf{u}_1$.

Ok, that's progress. Next, we realize that to get the variance of the projected data we need to plug in the projections into Equation 1. This leads us to the following equation for the variance of the projected data:

$$Var(\mathbf{u}_1^T\mathbf{x}) = \mathbb{E}[\{\mathbf{u}_1^T\mathbf{x} - \mathbf{u}_1^T\bar{\mathbf{x}}\}^2] \tag{2}$$

where $\bar{\mathbf{x}}$ is the sample mean of the original data points:

$$\bar{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}^{(i)}$$

An estimate of $Var(\mathbf{u}_1^T\mathbf{x})$ is s^2:

$$s^2 = \frac{1}{N}\sum_{i=1}^{N}\{\mathbf{u}_1^T\mathbf{x}^{(i)} - \mathbf{u}_1^T\bar{\mathbf{x}}\}^2 \tag{3}$$

(Note, to be pedantic we should actually divide by $N-1$, but we follow the approach sketched by Bishop and he uses the above expression. It doesn't change the result.)

Now, here's the magic (pay attention!): The covariance matrix of our data, let's call it $\mathbf{S}$, is by definition equal to:

$$\mathbf{S} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T \tag{4}$$

and it's possible to rewrite Equation 3 as follows:

$$s^2 = \frac{1}{N}\sum_{n=1}^{N}\{\mathbf{u}_1^T\mathbf{x}_n - \mathbf{u}_1^T\bar{\mathbf{x}}\}^2 = \mathbf{u}_1^T\mathbf{S}\mathbf{u}_1 \qquad \text{Objective Function} \tag{5}$$

Click to see the step-by-step calculations

1. **Start with the sample variance definition:**

$$s^2 = \frac{1}{N}\sum_{n=1}^{N}\{\mathbf{u}_1^T(\mathbf{x}_n - \bar{\mathbf{x}})\}^2$$

2. **Rewrite the square of a scalar:** Since $\mathbf{u}_1^T(\mathbf{x}_n - \bar{\mathbf{x}})$ is just a single number (scalar), we can use the identity $\mathbf{a}^2 = \mathbf{a} \cdot \mathbf{a}^T$. Because the transpose of a scalar is itself, we write:

$$s^2 = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{u}_1^T(\mathbf{x}_n - \bar{\mathbf{x}}))((\mathbf{x}_n - \bar{\mathbf{x}})^T\mathbf{u}_1)$$

3. **Regroup the terms:** Since matrix multiplication is associative, we can move the parentheses:

$$s^2 = \frac{1}{N}\sum_{n=1}^{N}\mathbf{u}_1^T\{(\mathbf{x}_n - \bar{\mathbf{x}})(\mathbf{x}_n - \bar{\mathbf{x}})^T\}\mathbf{u}_1$$

4. **Factor out the constant vectors:** Because $\mathbf{u}_1$ does not depend on the summation index n, we can pull $\mathbf{u}_1^T$ to the left and $\mathbf{u}_1$ to the right of the sum:

$$s^2 = \mathbf{u}_1^T\left[\frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n - \bar{\mathbf{x}})(\mathbf{x}_n - \bar{\mathbf{x}})^T\right]\mathbf{u}_1$$

5. **Substitute the Covariance Matrix:** The term inside the brackets is exactly the definition of the covariance matrix $\mathbf{S}$. Thus:

$$s^2 = \mathbf{u}_1^T\mathbf{S}\mathbf{u}_1 \tag{6}$$

(Note, that the most important step was writing the square as $a \cdot a^T$. We could have written it as $a^T \cdot a$. But putting the transpose second in the product paves the way to factoring out the covariance $\mathbf{S}$.)

This is the main "trick". With this we have established the objective function as described in step 2 in our outline above - a function that computes the variance in dependence of the vector $\mathbf{u}_1$. It involves the covariance matrix $\mathbf{S}$ of $\mathbf{X}$. We will need this later. We can now go ahead and optimize this function, i.e. find the $\mathbf{u}_1^T$ which maximizes it.

Optimizing the objective function

that we've just derived: $s^2 = \mathbf{u}_1^T\mathbf{S}\mathbf{u}_1$. As already stated, I will not go into the optimization - this would be a topic for another article - but I'll state the results and tell you where to find more information.

Because we know that $||\mathbf{u}_1|| = 1$ we have two functions:

- $f(\mathbf{u}_1) = \mathbf{u}_1^T\mathbf{S}\mathbf{u}_1 = s^2$ (objective function)
- $g(\mathbf{u}_1) = \mathbf{u}_1^T\mathbf{u}_1 = 1$ (constraint)

This allows us to use the method of **Lagrange multipliers** (see here or here for accessible introductions.) Calculus tells us that at an optimum, $\nabla f = \lambda_1 \nabla g$, i.e. the gradient of the objective and the gradient of the constraint are collinear although they may differ in magnitude by the factor λ_1. From this we can construct the **Lagrangian function**, which combines our objective and our constraint into a single expression:

$$\mathcal{L}(\mathbf{u}_1, \lambda_1) = \mathbf{u}_1^T \mathbf{S} \mathbf{u}_1 + \lambda_1 (1 - \mathbf{u}_1^T \mathbf{u}_1)$$

The procedure is then to take the gradient of the Lagrangian, set it to $\mathbf{0}$ and solve for $\mathbf{u}_1$:

$$\nabla_{\mathbf{u}_1} \left[\mathbf{u}_1^T \mathbf{S} \mathbf{u}_1 + \lambda_1 (1 - \mathbf{u}_1^T \mathbf{u}_1) \right] = 2\mathbf{S}\mathbf{u}_1 - 2\lambda_1 \mathbf{u}_1 = 0$$

It follows that:

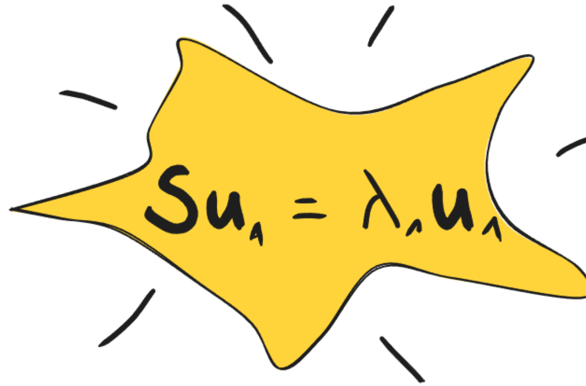

Figure 9: The optimization result equals the definition of an eigenvector!

and you see that the above formula {Figure 9} is the definition of an eigenvector! More precisely, any stationary point corresponds to an eigenvector, and the maximum variance is achieved by the eigenvector with the largest eigenvalue. OMG! This means, the projection line we are looking for, the vector $\mathbf{u}_1$, **is an eigenvector of the covariance matrix of our data**. This insight will help us to compute the PCA. It's the connection to techniques that allow us to compute eigenvectors for matrices, such as matrix diagonalization, spectral decomposition, and SVD.

Just out of curiosity, before we compute the eigenvectors, let's find the actual variance. For this we left-multiply both sides by $\mathbf{u}_1^T$:

$$\mathbf{u}_1^T \mathbf{S} \mathbf{u}_1 = \mathbf{u}_1^T \lambda_1 \mathbf{u}_1$$

Since λ_1 is a scalar, we can move it to the front:

$$\mathbf{u}_1^T \mathbf{S} \mathbf{u}_1 = \lambda_1 (\mathbf{u}_1^T \mathbf{u}_1)$$

Finally, we apply our constraint that $\mathbf{u}_1$ is a unit vector ($\mathbf{u}_1^T\mathbf{u}_1 = 1$), which gives us:

$$\mathbf{u}_1^T\mathbf{S}\mathbf{u}_1 = \lambda_1$$

This last equation tells us that the variance of the projected data onto $\mathbf{u}_1$ is the according eigenvalue λ_1. The direction vector $\mathbf{u}_1$, in this context, is called a **principal axis** or **principal component** (this is according to Bishop [1] p. 563, other authors might use the term differently).

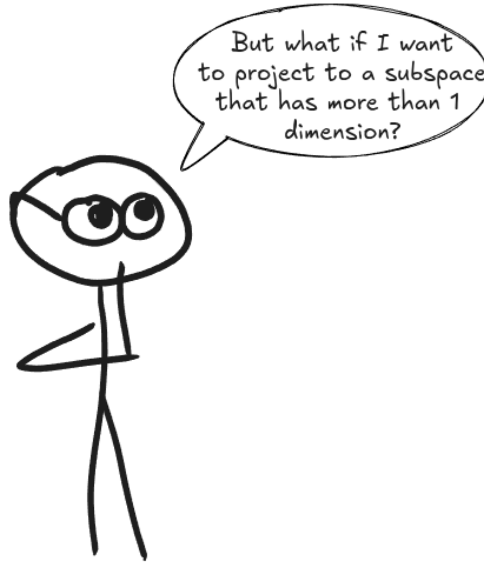

Figure 10: What about M>1?

To project onto a subspace with more than one dimensions we can find more axes and components by repeating the above process and making sure that each new found axis is orthogonal to the ones already found. **This can be shown to equal all the eigenvalues and eigenvectors of S.** (There was some handwaving here, as we didn't show the proof. In [1] this is left to show as an exercise.)

Right, the principal axes of $\mathbf{X}$ correspond to the eigenvectors of $\mathbf{S}$ and the variance along each axis is given by the corresponding eigenvalue. There will be at most D of those and we can choose $M < D$ as our M-dimensional subspace of D by choosing the M largest eigenvalues and their corresponding eigenvectors.

How to compute the PCA?

In the following we will see how to use SVD to compute PCA. This will be our bridge from math to code. However, in terms of coding effort there is a quicker way to compute PCA by

Figure 11: Insight.

using the
statsmodels library. I'll add a code snippet for how to use this at the end of Code section (Listing 2).

We know that the principal axes and component of $\mathbf{X}$ are just the eigenvectors and eigenvalues of $\mathbf{S}$, thus our task boils down to determining those. There is one more "trick" that will make this even easier. **Here's the key insight**: The covariance $\mathbf{S}$ of our data is according to Equation 4 But this expression involves a sum. If the mean of our data matrix is $\bar{\mathbf{x}} = 0$ 0 then we could easily **rewrite the sum as a matrix product**. :

$$\mathbf{S} = \frac{1}{N}\tilde{\mathbf{X}}^T\tilde{\mathbf{X}} \tag{7}$$

where

$$\tilde{\mathbf{X}} = \mathbf{X} - \bar{\mathbf{x}}$$

Take a moment to verify this! The covariance $\mathbf{S}$ remains the same, regardless if we compute it according to Equation 4, where we explicitly remove the mean, or if remove the mean first and then apply Equation 7. It might seem trivial, but the translation from sum to matrix product is key. Again: we can **express the covariance of a centered data matrix as a product of two matrices!**

Thus, when we say that we want the eigenvectors and eigenvalues of $\mathbf{S}$ we can now rephrase it as we want the eigenvectors of the centered data matrix $\frac{1}{N}\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$. The factor $\frac{1}{N}$ can be dropped because scalar multiplication does not change the eigenvectors - only the eigenvalues.

Since PCA uses unit-length eigenvectors by convention, we normalize the resulting directions separately.

So - what is an eigenbasis for $\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$? Ha! We know that $\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$ is a **symmetric matrix** by definition, therefore, we could use spectral decomposition to solve this. Alas, we would have to actually compute $\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$. This might be a massive matrix which can be expensive to compute. Luckily we can get the eigenvectors and values by using SVD without ever having to compute $\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$.

Computing the principal axes

SVD tells us that any matrix $\mathbf{K}$ can be decomposed into $\mathbf{K} = \mathbf{U}\mathbf{V}^T$, where $\mathbf{V}$'s columns are the eigenvectors of $\mathbf{K}^T\mathbf{K}$, and contains the squareroots of the eigenvalues on its diagonal. Since we are interested in the eigenvectors of $\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$ we can **simply compute SVD of $\tilde{\mathbf{X}}$!** The vectors in $\mathbf{V}$ are the eigenvectors we are looking for, the eigenvectors of $\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$.

Computing the principal components

Now to the eigenvalues aka principal components: First we need to square them to get the eigenvalues for $\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$. But we want the eigenvalues for $\mathbf{S}$ and $\mathbf{S} = \frac{1}{N}\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$. We have only computed the eigenvalues for $\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$ we still need to multiply by $\frac{1}{N}$. And that's it! Done.

Summary Compute PCA using SVD

In the future, you don't want to go through all the theory. You'll just need to know what to do. Follow these instructions along with the code below.

1) Center the data matrix $\mathbf{X} - \bar{\mathbf{x}} = \tilde{\mathbf{X}}$
2) Compute SVD of $\tilde{\mathbf{X}}$ to obtain $\mathbf{V}^T$ and . The rows of $\mathbf{V}^T$ (or columns of $\mathbf{V}$ are the principal axes we are looking for .
3) Square the diagonal values from to get the the eigenvalues (variances along each principal component) and multiply by $\frac{1}{N}$.
4) To actually project the data onto the M-dimensional subspace: multiply the centered data by the first M columns of $\mathbf{V}$: $\mathbf{X}_{new} = \tilde{\mathbf{X}}\mathbf{V}_M$.

See the code below for details.

Code

The result from above code is show in Figure 12.

Listing 1 Manual PCA computation using SVD

```
import numpy as np
import matplotlib.pyplot as plt

# 1. Create fictive data X
np.random.seed(42)
x1 = np.random.normal(4, 1.5, 100)
x2 = x1 + np.random.normal(0, 0.8, 100)
X = np.vstack((x1, x2)).T

# 2. Center the data
X_mean = np.mean(X, axis=0)
X_centered = X - X_mean

# 3. Compute SVD
U, Sigma, Vt = np.linalg.svd(X_centered, full_matrices=False)

# The columns of V (rows of Vt) are the Principal Components
eigenvectors = Vt.T

# Convert Singular Values to Eigenvalues: lambda = sigma^2 / (N)
N = X.shape[0]
eigenvalues = (Sigma**2) / (N)

# Project onto first M components:
M=1
X_new = X_centered @ eigenvectors[:,:M]
```

Code for statsmodels

As promised here is the code for how to use the *statsmodels*-library: Listing 2.

Summary

PCA touches many topics which all come together by the main idea: "Find the projection line (subspace) on which the projected data has the largest variance." The word "projection" refers us to linear algebra, "largest" refers to optimization theory and calculus and "variance" to statistics. We first had to construct an objective function which we could optimize. The solution to this was the eigenvalue definition which then lead us to SVD.

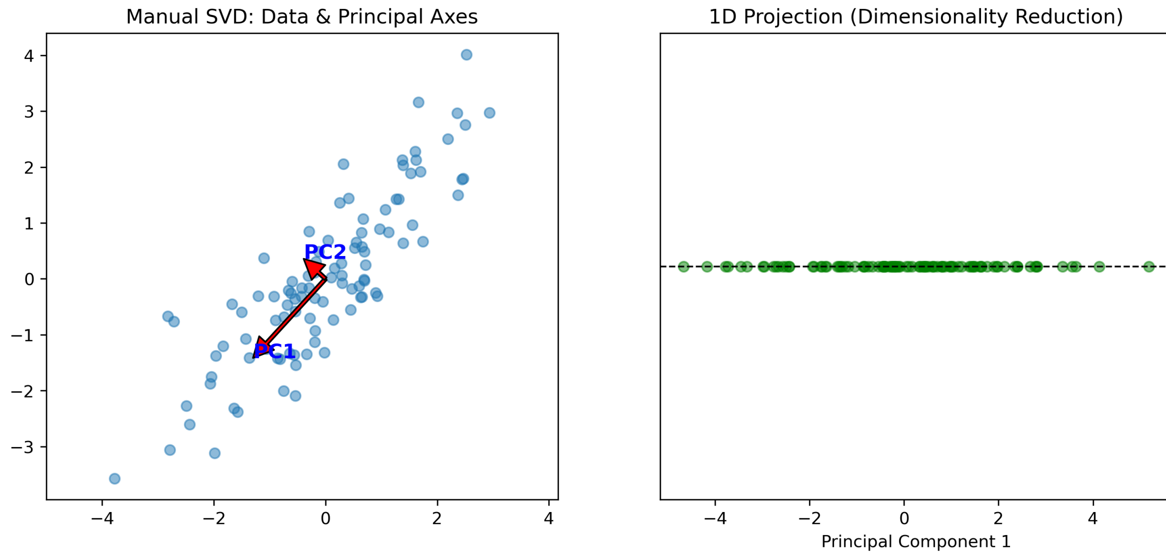

Figure 12: We project the 2D data from the left panel onto the first principal component (the line) shown on the right.

In practical terms: center the data, compute $\mathrm{SVD}(\tilde{\mathbf{X}})$ to get the **eigenvectors of S** $=$ **principal axes of X**. The **variances** are then $\lambda_i = \frac{\sigma_i^2}{N}$ from the SVD diagonal.

References

[1] C. M. Bishop, *Pattern recognition and machine learning.* in Information science and statistics. New York, NY: Springer Science+Business Media, LLC, 2019.

Listing 2 PCA computation using statsmodels library

```
from statsmodels.multivariate.pca import PCA
import pandas as pd
import numpy as np

# 1. create fictive data X
np.random.seed(42)
x1 = np.random.normal(4, 1.5, 100)
x2 = x1 + np.random.normal(0, 0.8, 100)
X = np.vstack((x1, x2)).T

# 2. Center the data
# Assuming X is the data matrix
# statsmodels handles centering and scaling automatically
pc = PCA(X, ncomp=2, standardize=True)

# 1. The Principal Axes (the V matrix from SVD)
eigenvectors = pc.loadings

# 2. The Eigenvalues (the variance)
eigenvalues = pc.eigenvals

# 3. The Projected Data (the new coordinates)
X_projected = pc.factors
```