FROM SCRATCH

Implementing PCA From Scratch

Brushing up your linear algebra skills with just Python and NumPy

Marvin Lanhenke Dec 27, 2021 · 5 min read

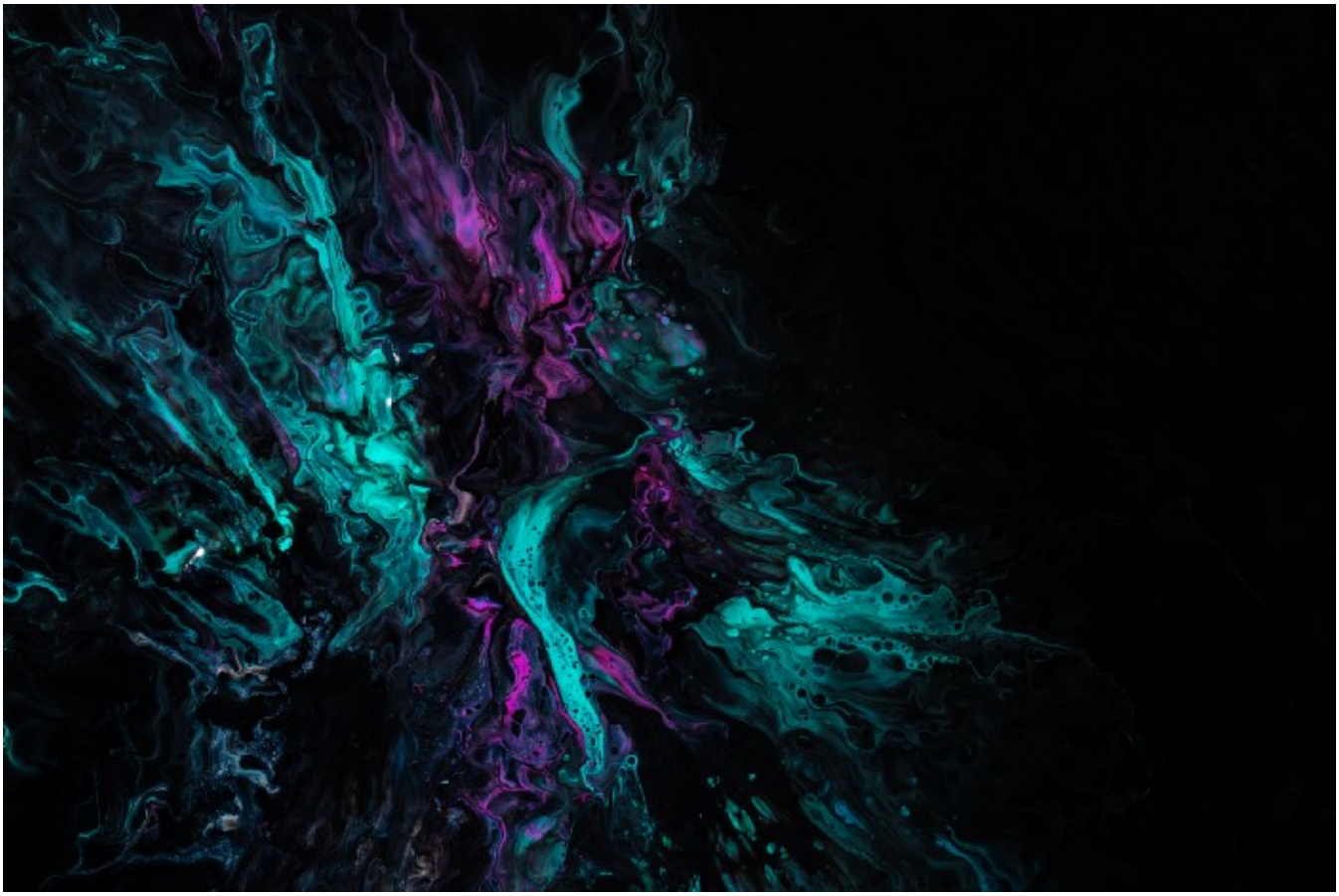


Photo by Pawel Czerwinski on Unsplash

I apply an algorithm and it somehow does the job.

Principal Component Analysis or PCA is one of those algorithms. It's an unsupervised learning algorithm with the purpose of dimensionality-reduction by transforming a large set of features into a smaller one, while preserving as much information as possible.

In this article, we are going to demystify some of the voodoo-magic, by implementing PCA from scratch. We will do this in a step-by-step fashion using just Python and NumPy.

General Overview

Before diving straight into the implementation details, let's get a high-level overview of how the algorithm actually works.

Principal Component Analysis, consists mainly of three steps:

1. First of all, we need to compute the *covariance matrix*.
2. Once we obtain this matrix, we need to decompose it, using *eigendecomposition*.
3. Next, we can select the most important eigenvectors based on the eigenvalues, to finally *project* the original matrix into its reduced dimension.

In the following section, we will take a closer look at each step, implementing PCA in just a single class. Below we can already take a look at the *skeleton class*, which provides us with some kind of blueprint.

```
class PCA:
    def fit_transform(self, X, n_components=2):
        pass

    def standardize_data(self, X):
        pass

    def get_covariance_matrix(self):
        pass
```

```
13
14      def project_matrix(self, eigenvectors):
15          pass
```

pca_skeleton.py hosted with ❤ by GitHub view raw

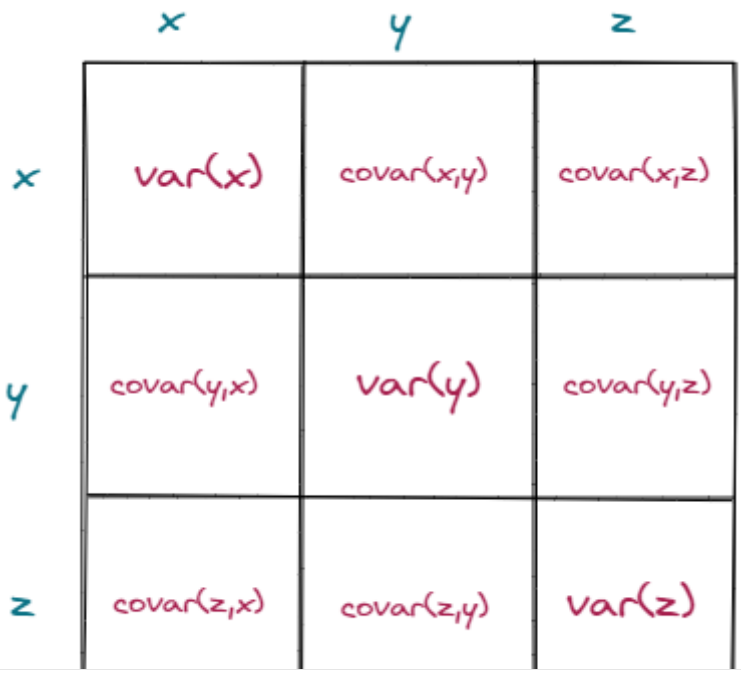
Implementation From Scratch

Compute the covariance matrix

As described in the general overview, our first step mainly involves the calculation of the covariance matrix. So why do we need to do that?

The covariance matrix, loosely speaking, tells us how much two random variables vary together. If the covariance is positive, then the two variables are correlated, hence moving (*increasing or decreasing*) in the same direction. If the covariance is negative, then the variables are inversely correlated, meaning they're moving in the opposite direction *(e.g. one is increasing while the other is decreasing)*

Let's assume we have a dataset with three features. If we calculate the covariance matrix, we will get a 3x3 matrix, containing the covariance of each column with all the other columns and itself.



The covariance matrix will be our centerpiece, when applying eigendecomposition, allowing us to choose the main vectors or the main directions, which explain most of the variance within our dataset.

Now that we know what a covariance matrix is, we still need to know how to compute it. The formula for calculating the matrix can be stated as the following:

$$\sigma(x,y) = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{n-1}$$

If we mean-center our data beforehand, we can omit the subtraction of *'x-bar'* and *'y-bar'* respectively. Simplifying our equation and expressing it in terms of linear algebra notation we get the following:

$$C = \frac{A^{\top}A}{n-1}$$

Which is simply the dot product of the mean-centered data matrix with itself divided by the number of samples.

Equipped with this new knowledge, we can implement our first two class methods.

```python
class PCA:
    #...

    def standardize_data(self, X):
        # subtract mean and divide by standard deviation columnwise
        numerator = X - np.mean(X, axis=0)
        denominator = np.std(X, axis=0)
        return numerator / denominator

    def get_covariance_matrix(self, ddof=0):
        # calculate covariance matrix with standardized matrix A
        C = np.dot(self.A.T, self.A) / (self.n_samples-ddof)
        return C
```

pca_covar.py hosted with ❤ by GitHub view raw

> *to the analysis. The covariance matrix can be also computed using the NumPy function numpy.cov(x).*

Decomposing the covariance matrix

The next step in our implementation mainly concerns the decomposition of the covariance matrix — or more specifically, the *eigendecomposition*.

Eigendecomposition describes the factorization of a matrix into *eigenvectors* and *eigenvalues*. The eigenvectors provide us with information about the direction of the data, whereas the eigenvalues can be interpreted as coefficients, telling us how much variance is carried in each eigenvector.

So long story short, if we decompose our covariance matrix, we obtain the eigenvectors, which explain the most variance within our dataset. We can use those vectors to project our original matrix into a lower dimension.

So how do we decompose our covariance matrix?

Luckily, we can simply rely on the built-in NumPy function, since the *'manual'* computation of the eigenvalues and eigenvectors is quite tedious. The only thing we have to do, is to sort the eigenvalues and the eigenvectors accordingly, allowing us to select the most important eigenvectors based on the number of components we specify.

```
class PCA:
    #...
def get_eigenvectors(self, C):
    # calculate eigenvalues & eigenvectors of covariance matrix 'C'
    eigenvalues, eigenvectors = np.linalg.eig(C)

    # sort eigenvalues descending and select columns based on n_components
    n_cols = np.argsort(eigenvalues)[::-1][:self.n_components]
    selected_vectors = eigenvectors[:, n_cols]
    return selected_vectors
```

pca_eigendecomposition.py hosted with ❤ by **GitHub** **view raw**

Projecting and putting it all together

We have already completed most of the work by now.

the matrix and the eigenvectors, which also can be interpreted as a simple linear transformation.

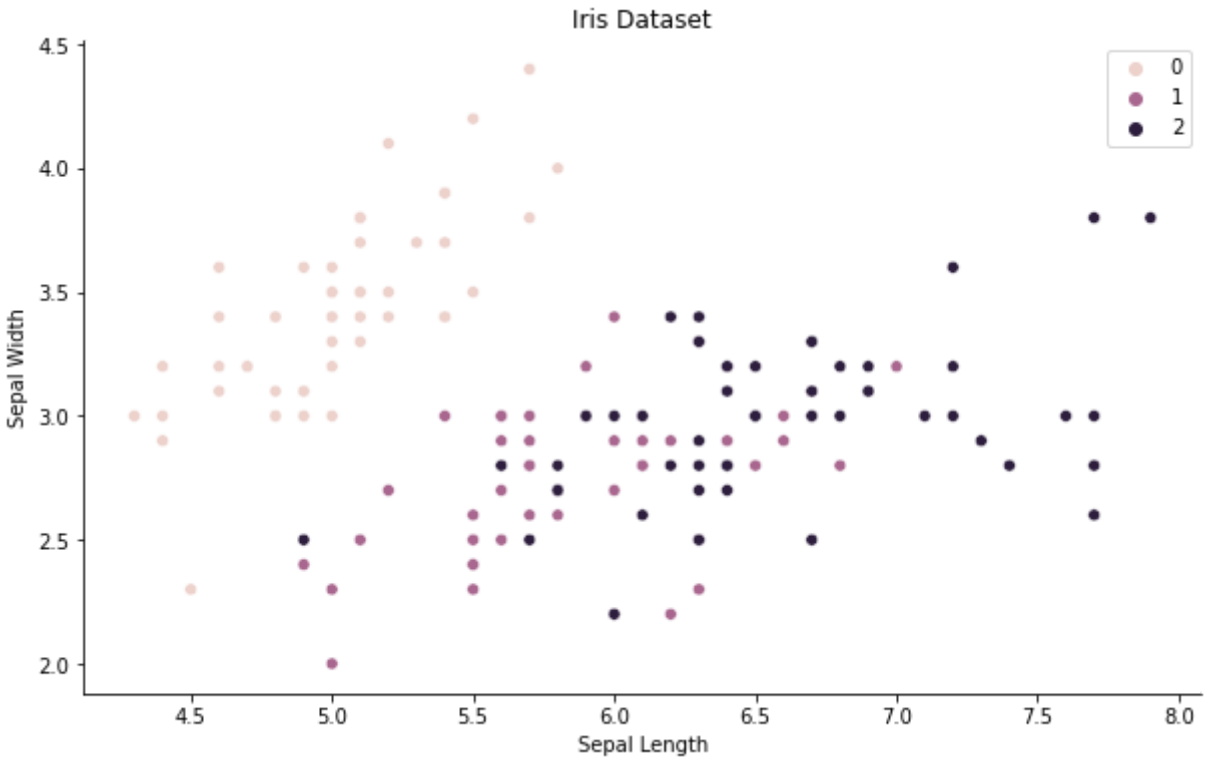
After implementing the two remaining methods, our final class looks like the following:

```python
class PCA:
    def fit_transform(self, X, n_components=2):
        # get number of samples and components
        self.n_samples = X.shape[0]
        self.n_components = n_components
        # standardize data
        self.A = self.standardize_data(X)
        # calculate covariance matrix
        covariance_matrix = self.get_covariance_matrix()
        # retrieve selected eigenvectors
        eigenvectors = self.get_eigenvectors(covariance_matrix)
        # project into lower dimension
        projected_matrix = self.project_matrix(eigenvectors)
        return projected_matrix

    def standardize_data(self, X):
        # subtract mean and divide by standard deviation columnwise
        numerator = X - np.mean(X, axis=0)
        denominator = np.std(X, axis=0)
        return numerator / denominator

    def get_covariance_matrix(self, ddof=0):
        # calculate covariance matrix with standardized matrix A
        C = np.dot(self.A.T, self.A) / (self.n_samples-ddof)
        return C

    def get_eigenvectors(self, C):
        # calculate eigenvalues & eigenvectors of covariance matrix 'C'
        eigenvalues, eigenvectors = np.linalg.eig(C)
        # sort eigenvalues descending and select columns based on n_components
        n_cols = np.argsort(eigenvalues)[::-1][:self.n_components]
        selected_vectors = eigenvectors[:, n_cols]
        return selected_vectors

    def project_matrix(self, eigenvectors):
        P = np.dot(self.A, eigenvectors)
        return P
```

Testing our Implementation

Now that we finished our implementation, there is just one thing left to do — we need to test it.

For testing purposes, we will use the iris dataset, which consists of 150 samples with 4 different features *(Sepal Length, Sepal Width, Petal Length, Petal Width)*. Let's take a look at the original data by plotting the first two features.



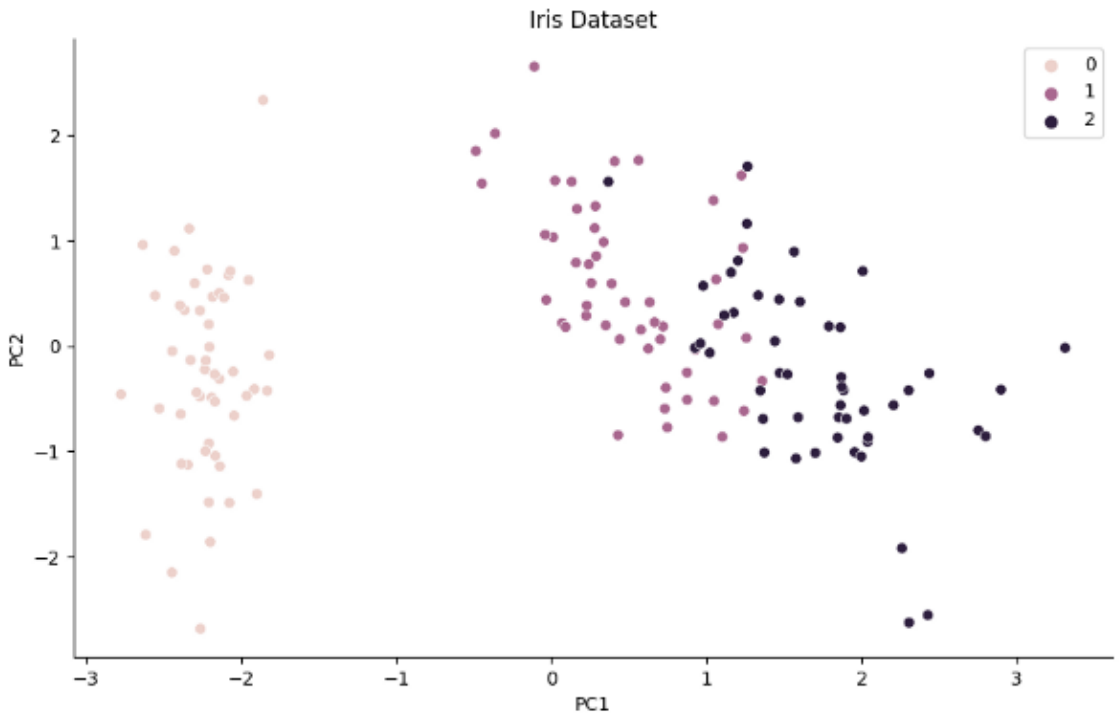
An Overview of the iris dataset by plotting the first two features [Image by Author]

We can now instantiate our own PCA class and fit it on the dataset. When running the code below, we get the following result.

```
from sklearn import datasets

# load iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# instantiate and fit_transform PCA
```

```
# plot results
fig, ax = plt.subplots(1, 1, figsize=(10,6))

sns.scatterplot(
    x = X_pca[:,0],
    y = X_pca[:,1],
    hue=y
)

ax.set_title('Iris Dataset')
ax.set_xlabel('PC1')
ax.set_ylabel('PC2')

sns.despine()
```

pca_iris_test.py hosted with ❤ by GitHub view raw



Iris dataset projected onto the first two principal components [Image by Author]

By applying PCA, we clearly untangled some of the class relations and separated the data more clearly. This lower-dimensional data structure should make a classification task a lot easier.

Conclusion

some very important linear algebra concepts.

PCA is a simple yet effective way to reduce, compress and untangle high-dimensional data. Understanding and implementing the algorithm from scratch provides a great way to build strong fundamentals and revise our linear algebra knowledge, since it ties many important concepts together.

You can find the full code here on my GitHub.

> *Thank you for reading! Make sure to stay connected & follow me here on Medium, Kaggle, or just say 'Hi' on LinkedIn*