Implementation of Principal Component Analysis (PCA) in Python

Author: Henry Ofoe Agbi-Kaiser

1 Introduction

Principal component analysis (PCA) is a statistical technique that is used to identify patterns in data and reduce the dimensionality of the data. It does this by finding a new set of variables, called principal components, that capture the most important patterns in the data. The first principal component is the direction in the data that captures the most variation. The second principal component is the direction in the data that captures the second-highest variation, and so on. Each subsequent principal component is orthogonal (perpendicular) to the previous ones, which means that they capture patterns in the data that are independent of the patterns captured by the previous components. PCA is often used to visualize high-dimensional data, reduce the complexity of the data, or identify patterns in the data that might not be apparent when looking at the raw data. It is also used in a variety of applications, such as image recognition, speech recognition, and natural language processing.

Note: The concept of PCA is difficult to understand without visuals, so I will suggest you watch this video before we continue reading.

2 Method

Before we get to Python, we need to know the steps for performing PCA. The steps are explained in the subsections below.

2.1 Standardize the data

It is generally a good idea to standardize the data before performing the principal component analysis (PCA). Standardization involves subtracting the mean from each feature and dividing by the standard deviation, which helps give all the features the same scale. This is important for PCA because it allows the algorithm to treat all the features equally rather than giving more weight to features with larger numerical values.

There are a few cases where standardization may not be necessary, such as when all the features have the same scale already or when the features have already been transformed in some way that gives them the same scale. However, in most cases, standardization is recommended as a preprocessing step before performing PCA.

Let X be an $(n \times m)$ matrix representation of our data. That is,

$$\begin{array}{c} \begin{matrix} X_1 & X_2 & X_3 & \cdots & X_m \end{matrix} \\ \begin{pmatrix} X_{11} & X_{12} & X_{13} & \cdots & X_{1m} \\ X_{21} & X_{22} & X_{23} & \cdots & X_{2m} \\ X_{31} & X_{32} & X_{33} & \cdots & X_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ X_{n1} & X_{n2} & X_{n3} & \cdots & X_{nm} \end{pmatrix} \end{array} \tag{1}$$

where $X_j \ for \ j = 0, 1, 2, \ldots, m$ represent each feature.

Then each value in X_j is standardized as follows. First calculate the mean with (Equ.2).

$$\overline{X_j} = \frac{\sum_{i=0}^{n} X_{ij}}{n} \tag{2}$$

Next calculate the variance with (Equ.3).

$$Var(X_j) = \frac{\sum_{i=0}^{n} (X_{ij} - \overline{X_j})^2}{n-1} \tag{3}$$

and replace the value using (Equ.4)

$$X_{ij} = \frac{X_{ij} - \overline{X_j}}{\sqrt{Var(X_j)}} \tag{4}$$

2.2 Compute the covariance matrix

The covariance matrix is a square matrix that contains the pairwise covariances between all the variables in the dataset. It is used to measure the strength of the linear relationship between pairs of variables. If Z is the standardized data matrix, then the covariance matrix $(COV(Z))$ is given by (Equ.5)

$$\begin{bmatrix} COV(X_1, X_1) & COV(X_1, X_2) & COV(X_1, X_3) & \cdots & COV(X_1, X_m) \\ COV(X_2, X_1) & COV(X_2, X_2) & COV(X_2, X_3) & \cdots & COV(X_2, X_m) \\ COV(X_3, X_1) & COV(X_3, X_2) & COV(X_3, X_3) & \cdots & COV(X_3, X_m) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ COV(X_m, X_1) & COV(X_m, X_2) & COV(X_m, X_3) & \cdots & COV(X_m, X_m) \end{bmatrix} \tag{5}$$

2.3 Compute the eigenvectors and eigenvalues of the covariance matrix

The eigenvectors and eigenvalues of the covariance matrix are used to determine the principal components of the dataset. The eigenvectors are the directions along which the data varies the most, and the eigenvalues are the magnitudes of these variations.

Let $A = COV(Z)$. To find the eigenvalues and eigenvectors of A, you can follow these steps:

1. Start by finding the characteristic polynomial of the matrix. This is a polynomial equation whose roots are the eigenvalues of the matrix. To find the characteristic polynomial, you can use the following formula:

$$det(A - \lambda I) = 0 \tag{6}$$

where A is the matrix, λ is a scalar value, and I is the identity matrix.

2. Solve the characteristic polynomial for the values of λ that make the equation equal to zero. These values are the eigenvalues of the matrix.

3. For each eigenvalue, find the corresponding eigenvector. To do this, you can use the following formula:

$$(A - \lambda I)v = 0 \tag{7}$$

where v is the eigenvector. This equation represents the condition that the matrix $(A - \lambda I)$ must be singular (i.e., its determinant must be zero) for the given eigenvalue λ.

1. Solve the equation for v. This will give you the eigenvector corresponding to the given eigenvalue.
2. Repeat the process for each eigenvalue to find all of the eigenvectors.

2.4 Select the principal components

The number of principal components to keep is typically determined by the amount of variance in the data that needs to be explained or the number of dimensions that are needed to represent the data. The principal components are the eigenvectors with the largest eigenvalues.

2.5 Transform the data

The original data can be transformed into a new set of principal components by multiplying the data by the matrix of eigenvectors. The transformed data is a new dataset with the same number of rows as the original data, but with fewer columns (the number of principal components).

2.6 Interpret the results

The principal components can be used to summarize the information in the original dataset in a more compact and interpretable form. The first principal component explains the most variance in the data, the second principal component explains the second most variance, and so on. The principal components can be plotted to visualize the patterns in the data and to identify clusters or trends.

3 PCA in Python

There are several ways of finding the principal components of a dataset in Python. In this analysis, we will use the "eig" function, the "svd" function, and the "pca" function.

The dataset used in this analysis is the "digits" dataset, which is a collection of images of handwritten digits (0–9) that is often used for machine learning and data analysis. It is included in the scikit-learn library in Python. The digits dataset consists of 1797 images, each of which is 8x8 pixels in size. Each image is represented by a 64-dimensional feature vector, with each feature representing the grayscale intensity of one pixel in the image. The dataset also includes a label for each image, indicating which digit it represents.

Note: The codes and data used in this discussion can be downloaded from my PCA-Python GitHub Repository.

Before we perform any analysis of our data we need a few python libraries. Let's import them.

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Let's start by having a look at our data.

In [2]:

```
#Read data from its location
digits = np.loadtxt('digits.csv', delimiter=',')
digits
```

Out[2]:

```
array([[ 0.,  0.,  5., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  1.],
       [ 0.,  0.,  0., ...,  9.,  0.,  2.],
       ...,
       [ 0.,  0.,  1., ...,  0.,  0.,  8.],
       [ 0.,  0.,  2., ...,  0.,  0.,  9.],
       [ 0.,  0., 10., ...,  1.,  0.,  8.]])
```

In [3]:

```
#Get the shape of the data
digits.shape
```

Out[3]:

```
(1797, 65)
```

We can see that the data set has 1797 rows (an image per row) and 65 columns (a pixel value per column). Each pixel in an image is represented by the first 64 columns of a selected "digit," and the last column is the label.

We then assign names to our variables.

In [4]:

```
# Assigning names to the measurements and target variable
# X represents the matrix of pixels of the images
X = digits[:, :-1]
# Y represents the vector of labels of the images
Y = digits[:, -1]
```

Let's visualize a few images from X.

In [5]:

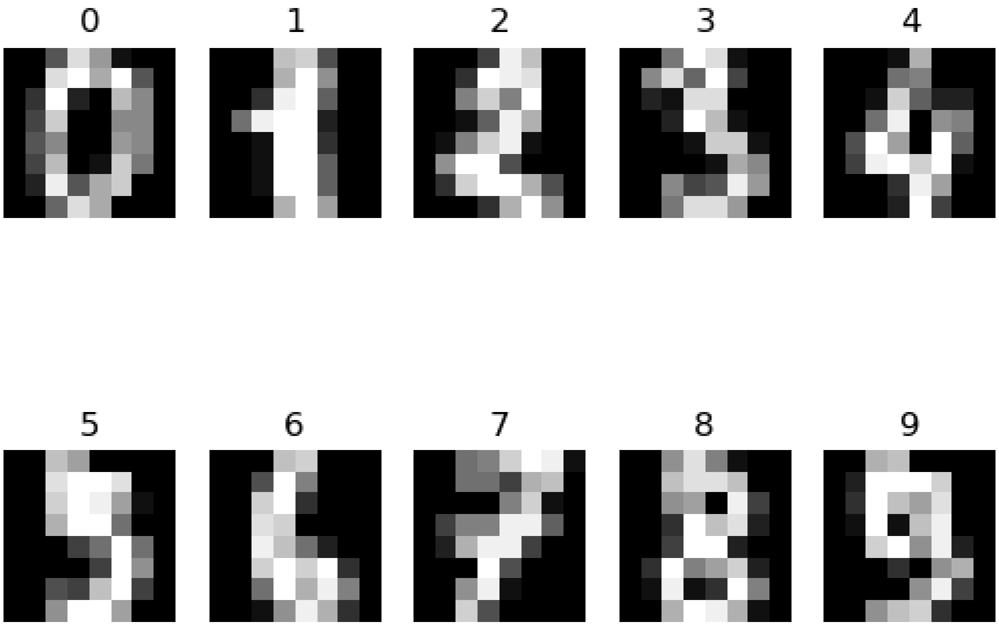
```
# Plot the first 10 digits
for i in range(1, 11):
    # Create a subplot
    plt.subplot(2, 5, i)

    # Reshape the image
    rs = np.reshape(X[i-1, :], (8, 8))

    # Rearrange the reshaped matrix
    plt.imshow(rs, cmap='gray')
    plt.title(str(int(Y[i-1])))
    plt.axis('off')

plt.suptitle('Figure 1: First Ten Numbers Plot')
# Show the plot
plt.show()
```

Figure 1: First Ten Numbers Plot



Let's generate one final pot before we move on to performing the analysis. We can plot the first 4 columns of pixels against each other to see the distribution of all 1797 rows of digits.

In [6]:

```
# Convert the first 4 columns of X into a Pandas DataFrame with specified column names
Xt = pd.DataFrame(X[:, :4], columns=['X1', 'X2', 'X3','X4'])

# Create a scatter plot using Seaborn
sns.pairplot(Xt, kind="scatter")

plt.suptitle('Figure 2: First Ten Numbers Plot')

# Show the plot
plt.show()
```

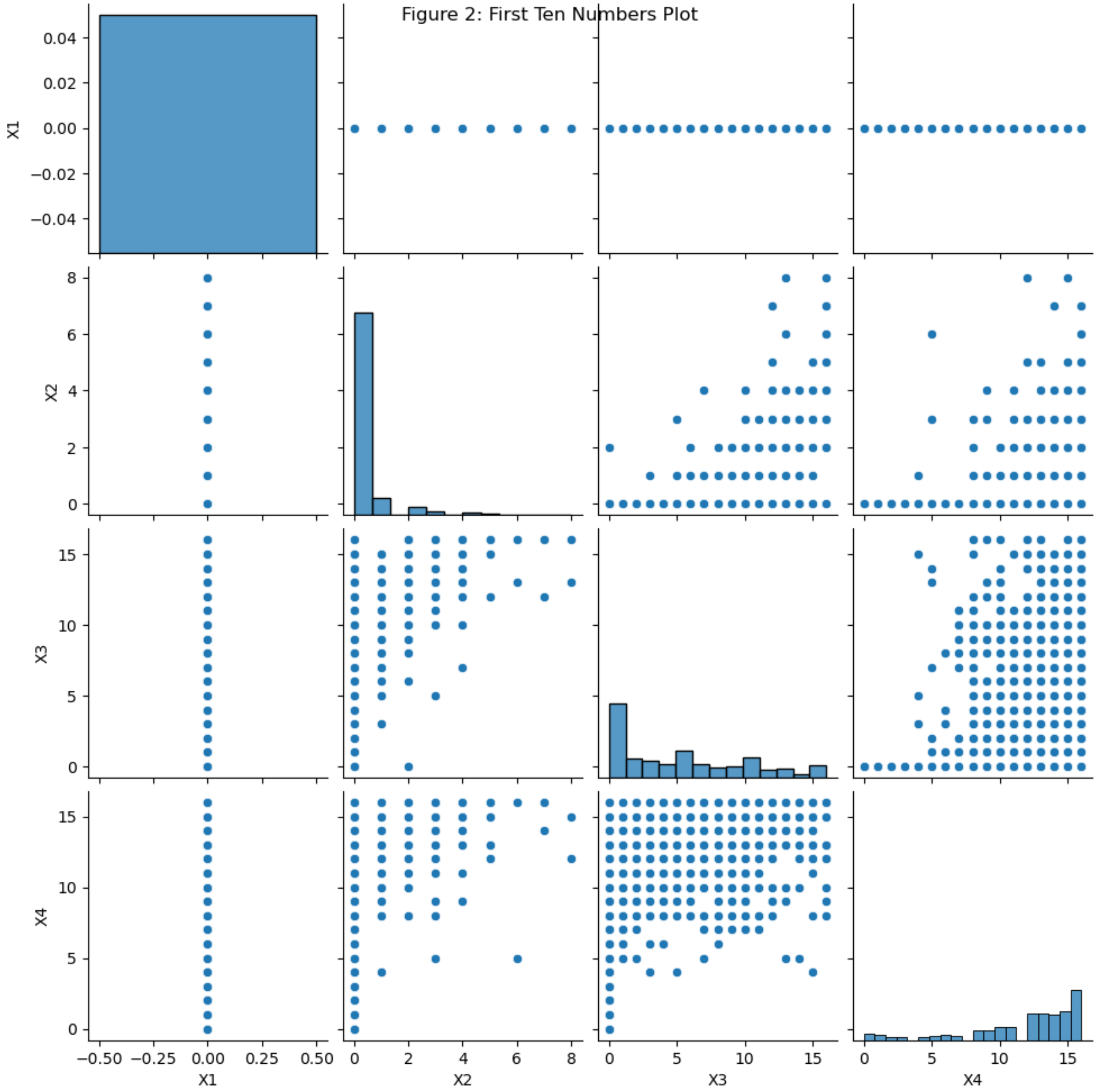


Figure 2: First Ten Numbers Plot

From Figure (2), there is no clear relationship between any two of the four selected pixels. To see a clear relationship, we need to include more pixels. However, if we want to plot the relationship between more than two pixels, it gets a little bit difficult. We can get away with three pixels by plotting in 3D. But an issue arises when we want to visualize the relationship between four or more pixels at once. The digits dataset is large enough to explain and illustrate the usefulness of a PCA.

To verify that there are no clear relations, let us plot only the fourth pixel against the third pixel since there are more data points in Figure (2) above.

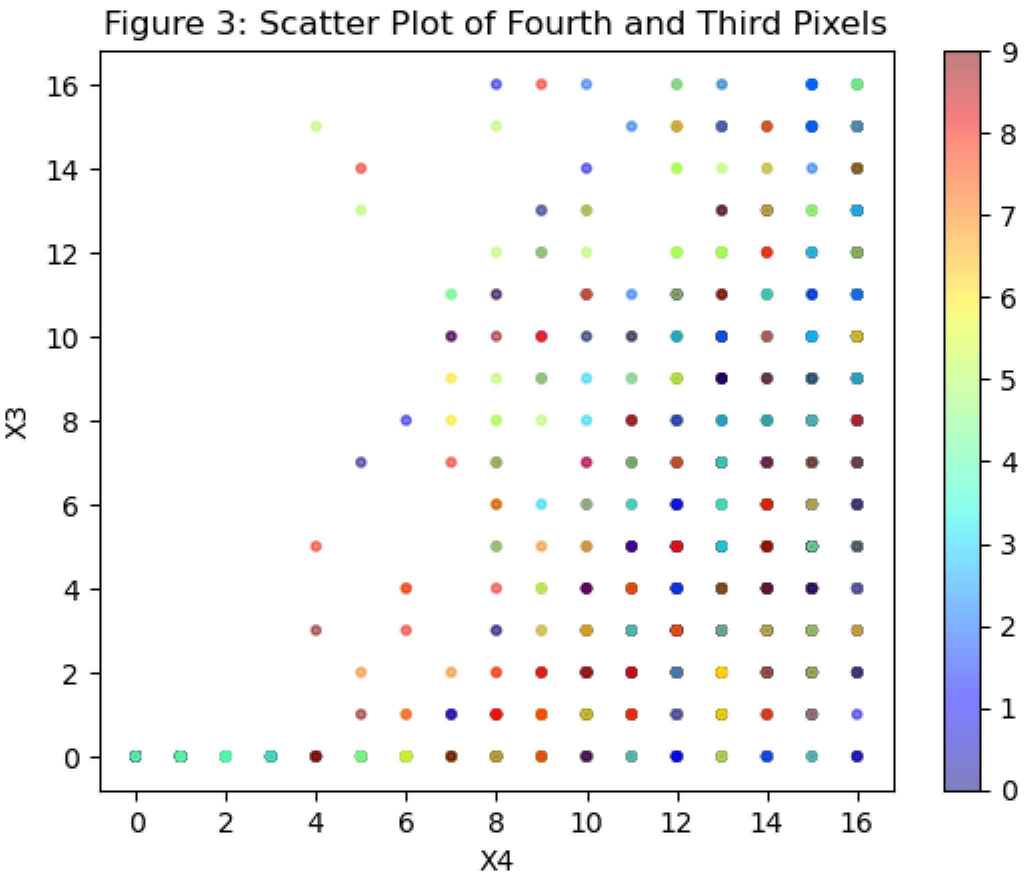
In [7]:

```
# Create a scatter plot
plt.scatter(X[:, 3], X[:, 2], c=Y, s=10, cmap='jet', alpha=0.5)

# Add a colorbar
plt.colorbar()

# Add labels to the plot
plt.xlabel('X4')
plt.ylabel('X3')
plt.title('Figure 3: Scatter Plot of Fourth and Third Pixels')

# Show the plot
plt.show()
```



From Figure (3), each data point represents a "digit", and there is no clear relationship that we can point out. Hence, we would use PCA to identify the pixels that would explain our data.

3.1 Performing PCA with the eig function

To demonstrate the concept of PCA, we will follow the steps we discussed earlier.

In [8]:

```
# Stardardize the measurements in X
# Since the features of the dataset is on the same scale there is no need to standardize
# but if there is a need to standardize, use the following commented code
    # from scipy.stats import zscore
    # X_standard = zscore(X, axis=0)

#we only need to center the data

# Mean center the data
X_centered = X - np.mean(X, axis=0)

# Calculate the covariance matrix
cov_matrix = np.cov(X_centered.T)

# Calculate the eigenvalues and eigenvectors of the covariance matrix
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

# Sort the eigenvalues and eigenvectors in descending order
idx = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:, idx]
```

```
# Select the top `num_components` eigenvectors
num_components = 2
eigenvectors = eigenvectors[:, :num_components]

# Project the data onto the principal components
X_projected = np.dot(X_centered, eigenvectors)
```

Let's make a scatter plot with the projected data.

In [9]:

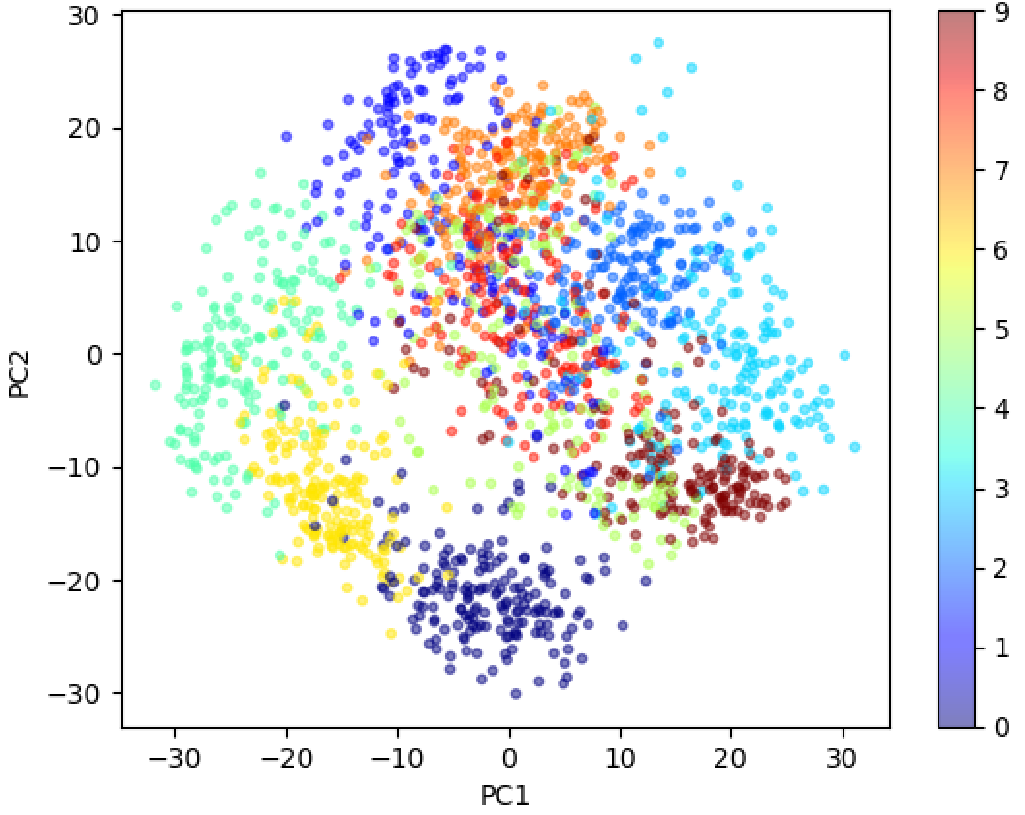
```
# Create a scatter plot
plt.scatter(X_projected[:, 0], X_projected[:, 1], c=Y, s=10, cmap='jet', alpha=0.5)

# Add a colorbar
plt.colorbar()

# Add labels to the plot
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('Figure 4: Scatter Plot of the First Two Principal Components  Using The EIG F

# Show the plot
plt.show()
```

Figure 4: Scatter Plot of the First Two Principal Components Using The EIG Function



From Figure (4) we can see that the numbers that are alike (6 and 0) are close to each other, and the numbers that are not alike (4 and 3) are far apart. This is a better representation of our data than Figure (3).

Here I pick out the first two principal components for visualization purposes. To get the number of principal components that will capture the most variance, we will need a plot of the percentage of explained variance.

To make a plot of the explained variance under the eig function, use the following code:

In [10]:

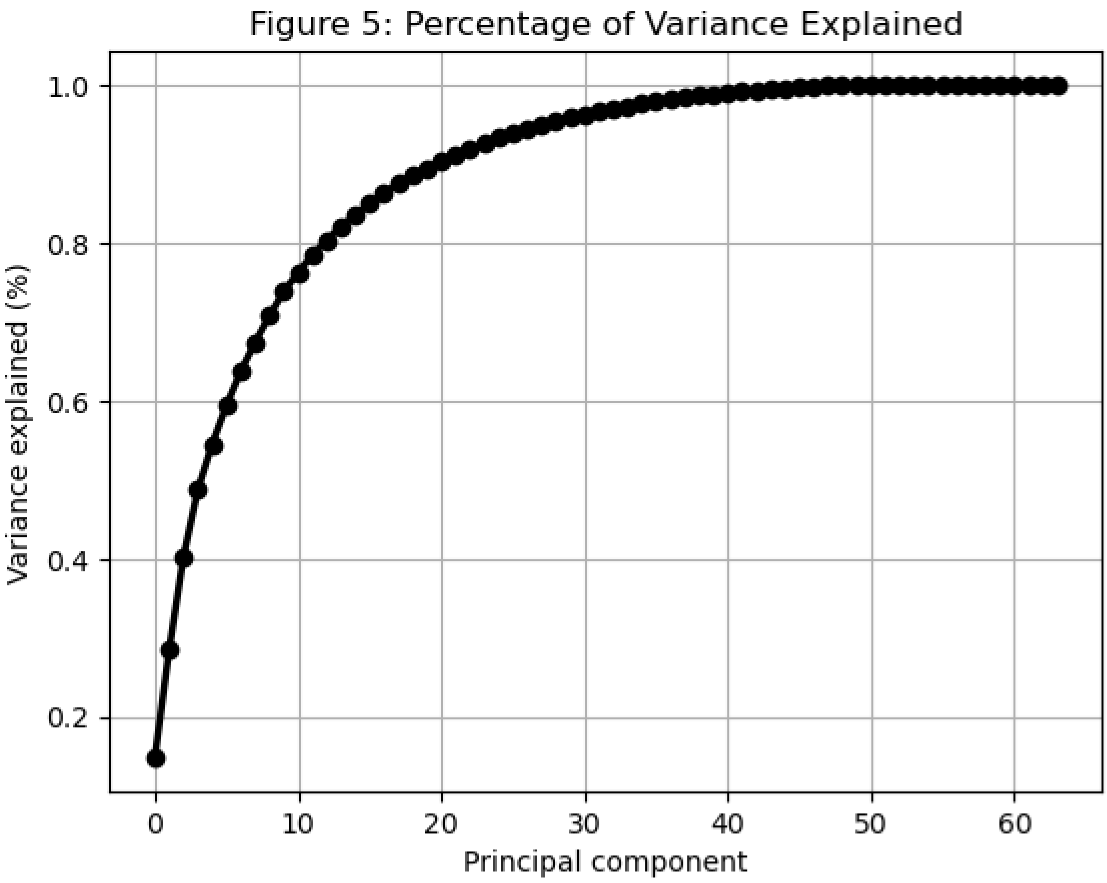
```
# Calculate a cumulative sum of the sorted eigenvalues
cumsum_ = np.cumsum(eigenvalues)

# Sum all the sorted eigenvalues
```

```
sum_ = np.sum(eigenvalues)

# Calculate the percentages of variance explained by the first n principal components
pvex = cumsum_ / sum_

# Plot the percentage
plt.figure()
plt.plot(pvex, 'k-o', linewidth=2.5)
plt.xlabel('Principal component')
plt.ylabel('Variance explained (%)')
plt.title('Figure 5: Percentage of Variance Explained')
plt.axis('tight')
plt.grid(True)
```



Let's assume we want to keep the first n components that explain at least 90% of the variance in the data. From Figure (5), the number of components to keep is 20.

Note: It is important to consider the trade-off between retaining as much of the variance in the data as possible and reducing the dimensionality of the data. In some situations, it may not be necessary to keep a high percentage of the variance. Instead, it may be more important to reduce the number of dimensions to make the data easier to understand or to make the calculations go faster.

3.2 Performing PCA with the svd function

In [11]:

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

# Compute the covariance matrix
cov_matrix = np.cov(X_centered, rowvar=False)

# Compute the eigenvectors and eigenvalues
U, Sigma, V = np.linalg.svd(cov_matrix)
```

```
# Select the number of components to keep
num_components = 2
eigenvectors = V[:num_components,:]

# Project the data onto the principal components
X_projected = np.dot(X_centered,eigenvectors.T)
```

Let’s make a scatter plot with the projected data.

In [12]:

```
# Create a scatter plot
plt.scatter(X_projected[:, 0], X_projected[:, 1], c=Y, s=10, cmap='jet', alpha=0.5)

# Add a colorbar
plt.colorbar()

# Add labels to the plot
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('Figure 6: Scatter Plot of the First Two Principal Components Using The SVD Fu

# Show the plot
plt.show()
```

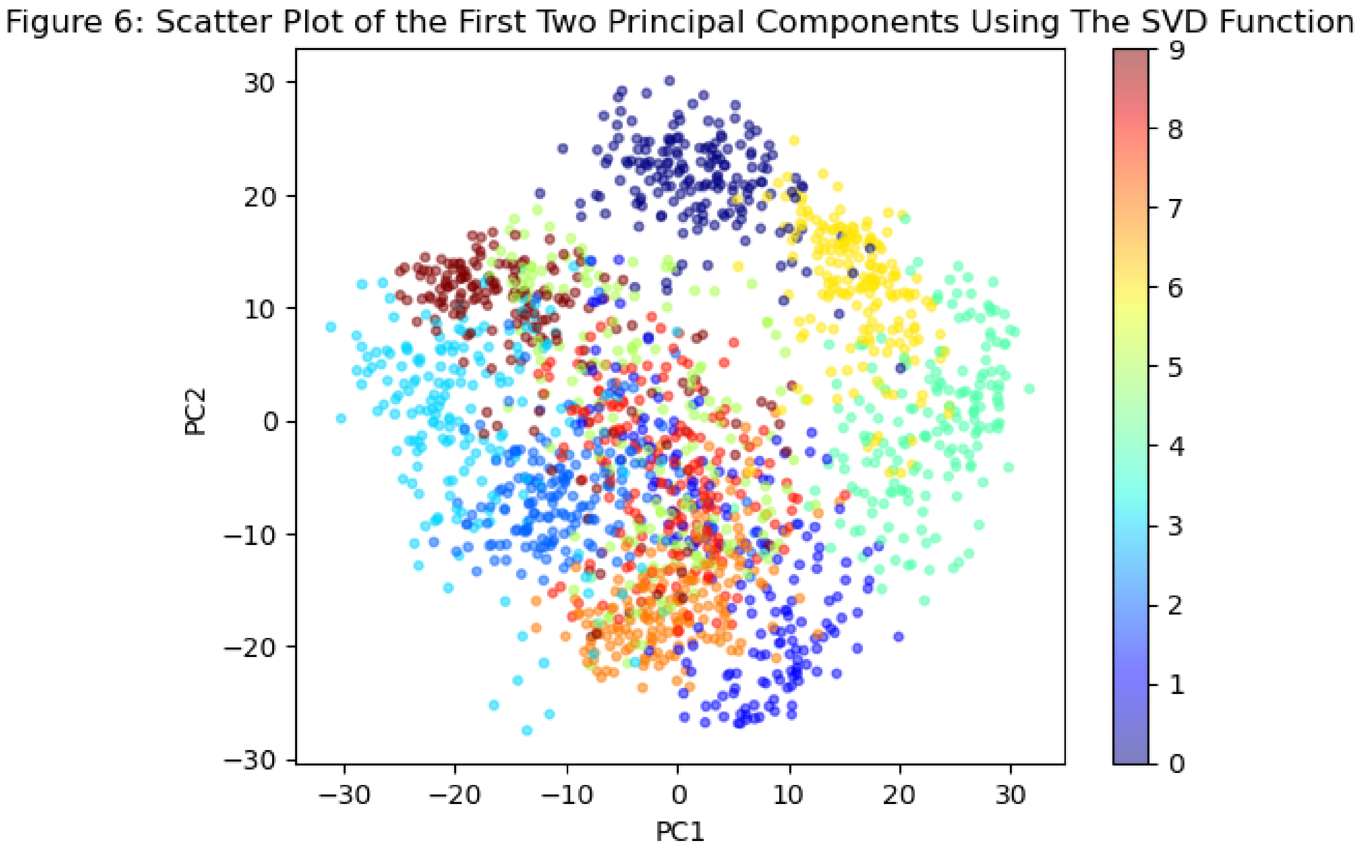


Figure (6) shows the same distribution of our data as Figure (4). The only difference is that both PC1 and PC2 of the svd function are the additive inverses of PC1 and PC2 of the eig function.

To make a plot of the explained variance under the svd function, use the following code:

In [13]:

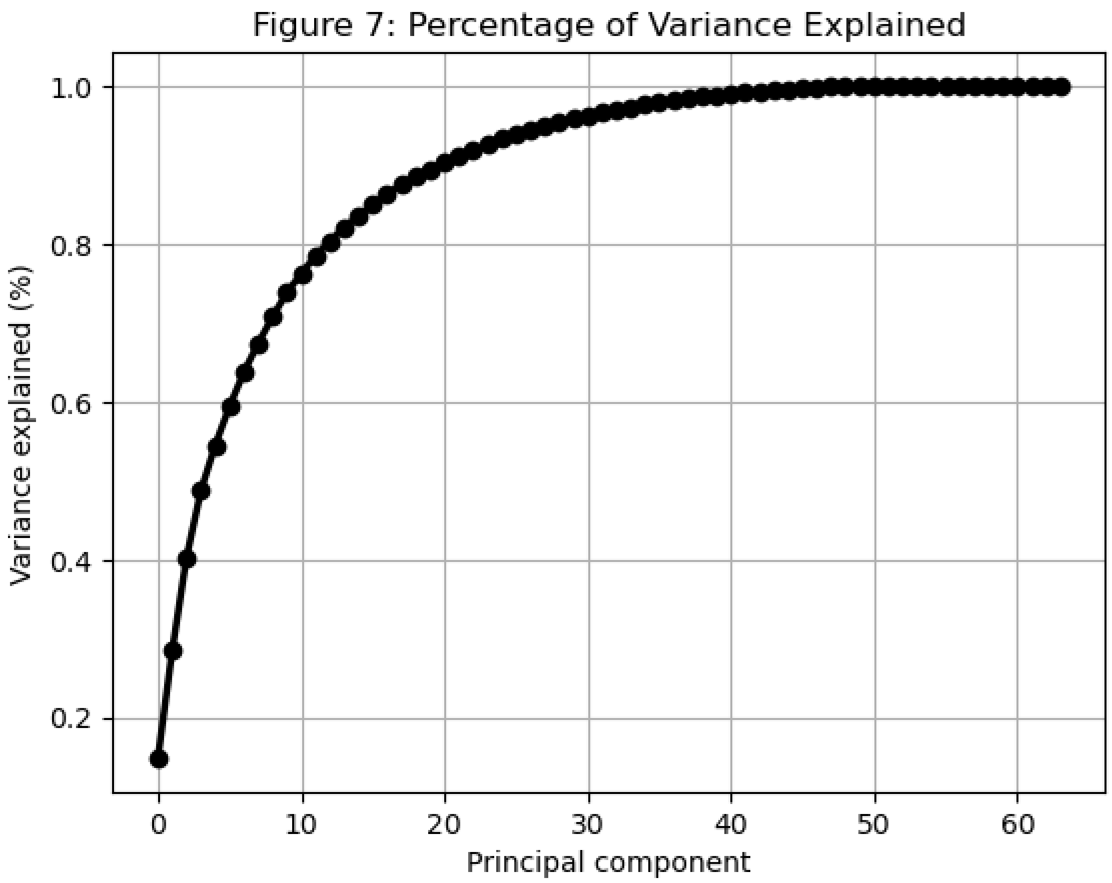
```
# Calculate a cumulative sum of the sorted eigenvalues
cumsum_ = np.cumsum(Sigma)

# Sum all the sorted eigenvalues
sum_ = np.sum(Sigma)

# Calculate the percentages of variance explained by the first n principal components
```

```
pvex = cumsum_ / sum_

# Plot the percentage
plt.figure()
plt.plot(pvex, 'k-o', linewidth=2.5)
plt.xlabel('Principal component')
plt.ylabel('Variance explained (%)')
plt.title('Figure 7: Percentage of Variance Explained')
plt.axis('tight')
plt.grid(True)
```



3.3 Performing PCA with the pca function

In [14]:

```
from sklearn.decomposition import PCA
```

In [15]:

```
# Perform PCA
pca = PCA()
X_pca = pca.fit_transform(X)

# Select the number of components to keep
num_components = 2

# Project the data onto the principal components
X_projected = X_pca[:, :num_components]
```

Let’s make a scatter plot with the projected data.

In [16]:

```
# Create a scatter plot
plt.scatter(X_projected[:, 0], X_projected[:, 1], c=Y, s=10, cmap='jet', alpha=0.5)

# Add a colorbar
plt.colorbar()

# Add labels to the plot
```

```
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('Figure 8: Scatter Plot of the First Two Principal Components Using The PCA Fu

# Show the plot
plt.show()
```

Figure 8: Scatter Plot of the First Two Principal Components Using The PCA Function

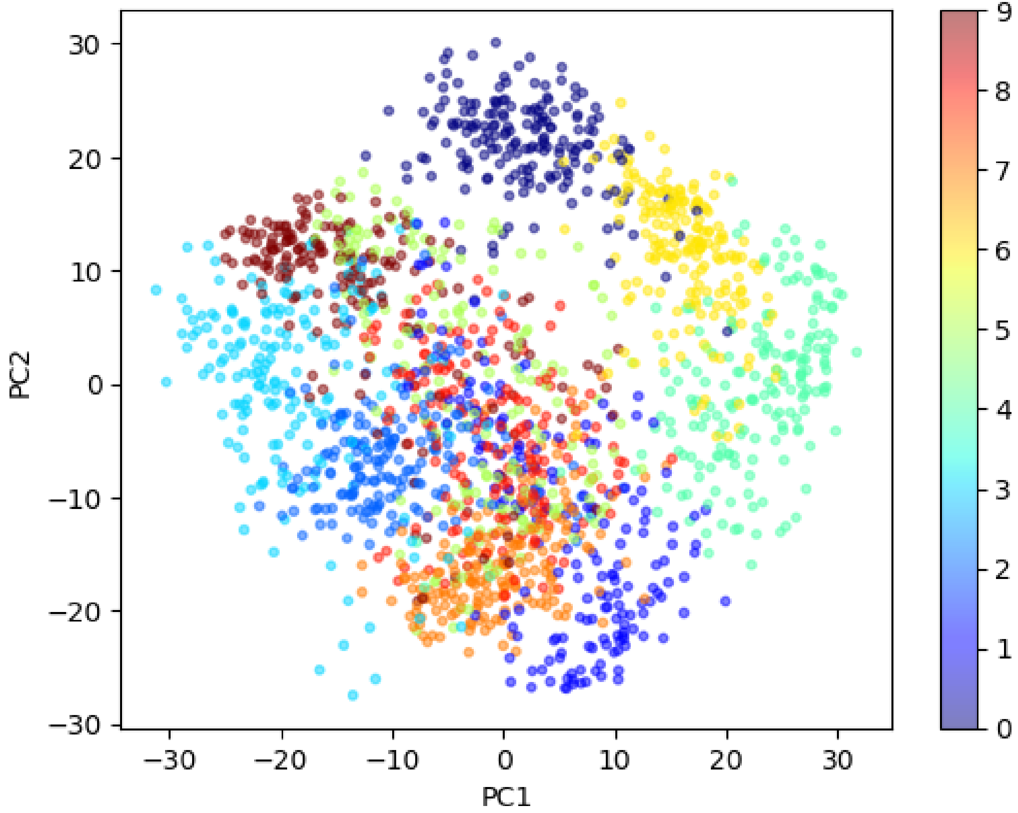


Figure (8) shows the same distribution as our data as Figure (6).

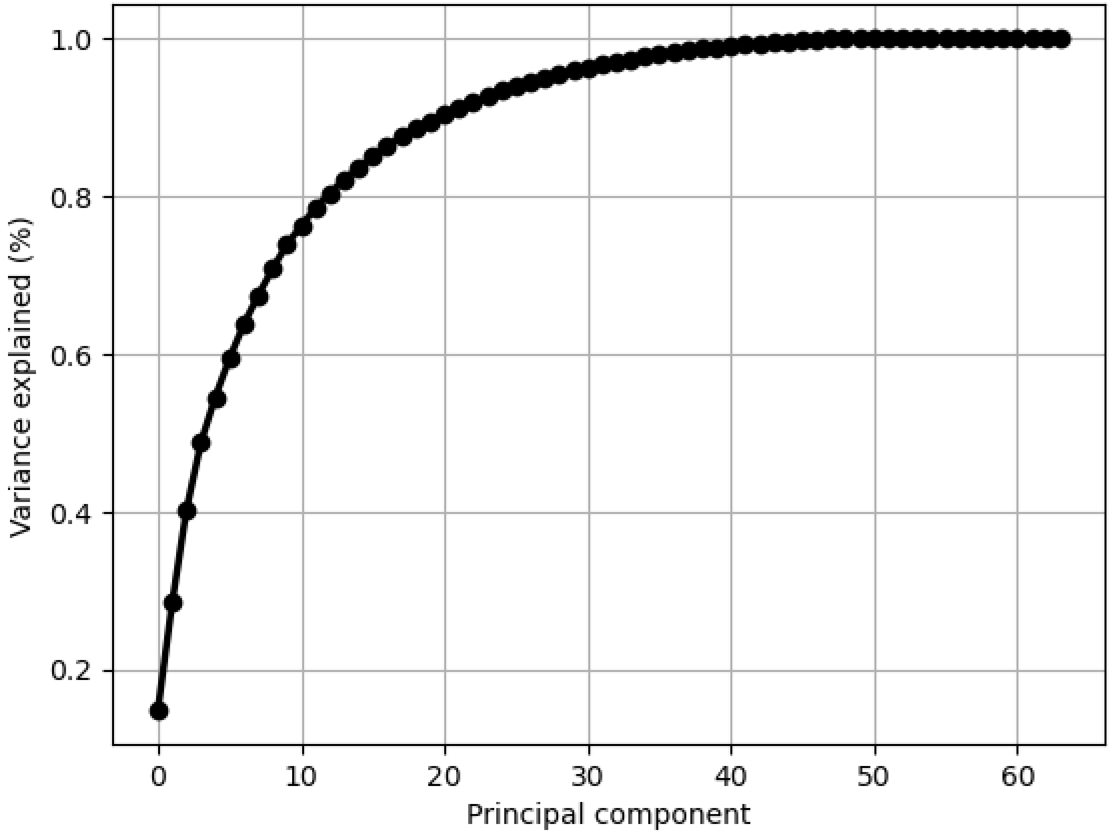
To make a plot of the explained variance under the pca function, use the following code:

```
In [17]: # Get the amount of variance explained by the PCA model
variance_explained = pca.explained_variance_ratio_

# Calculate a cumulative sum of the variance explained
cumsum_ = np.cumsum(variance_explained)

# Plot the percentage
plt.figure()
plt.plot(pvex, 'k-o', linewidth=2.5)
plt.xlabel('Principal component')
plt.ylabel('Variance explained (%)')
plt.title('Figure 9: Percentage of Variance Explained')
plt.axis('tight')
plt.grid(True)
```

Figure 9: Percentage of Variance Explained



4 Conclusion

PCA is a powerful technique that has many applications in data science and machine learning. Understanding how it works and how to implement it is crucial for anyone working in these fields. While it has its limitations, PCA is still a valuable tool for reducing the complexity of high-dimensional data and identifying patterns and trends in the data.

Good Luck!