Explaining PCA with Cats

What is PCA and How to Apply It?

Principal Component Analysis (PCA) is a dimensionality reduction algorithm used to simplify complex datasets, while retaining the most information (variance) possible.

Data Source and Image Loading

The images used in this example are from the **Cat and dog face** dataset on Kaggle. Specifically, the portion corresponding to the cat images was used.

Link to the dataset on Kaggle

Loading the Images

To perform this analysis, the images must first be loaded.

```
import os
import scipy
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

def load_images_from_folder(folder):
    images = []
    for filename in os.listdir(folder):
        img_path = os.path.join(folder, filename)
        try:
            with Image.open(img_path) as img:
                # Convert to grayscale ('L') and resize to 64x64
                img_gray = img.convert('L').resize((64, 64))
                images.append(np.array(img_gray))
        except IOError:
            pass
    return images

# Path to the folder containing the cat images
data_folder = './data/'
imgs = load_images_from_folder(data_folder)
```

1. Flatten the Data

Each 64x64 pixel image (a matrix) is converted into a 4096-pixel vector (one row). This transforms the dataset into a large data matrix Y where each row is an image and each column is a variable (a pixel).

```
# 'imgs' is a list of image matrices (64x64)
imgs_flatten = np.array([im.reshape(-1) for im in imgs])
```

2. Center the Data

The mean of each column (each pixel) is calculated and subtracted from every value in that column. This centers the data around the origin, which is a requirement for correctly calculating the covariance.

General Calculation:
For each column j, the mean μ_j is calculated as:

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} y_{i,j}$$

The centered matrix X is obtained by subtracting the mean of each column from the original matrix Y:

$$X_{i,j} = Y_{i,j} - \mu_j$$

```
def center_data(Y):
    # Calculate the mean vector for each column (axis=0)
    mean_vector = np.mean(Y, axis=0)
    # Subtract the mean vector from the original data matrix
    X = Y - mean_vector
    return X, mean_vector

# Apply the function
X, mean_vector = center_data(imgs_flatten)
```

3. Calculate the Covariance Matrix

From the centered data X, the covariance matrix (Σ) is calculated. This matrix describes the relationship and variance among the different pixels across the dataset.

General Calculation:

$$\Sigma = \frac{1}{m-1}X^T X$$

Where X^T is the transpose of the centered matrix and m is the number of observations (images).

```
def get_cov_matrix(X):
    # Number of observations
    m = X.shape[0]
    # Calculate the covariance matrix
    cov_matrix = np.dot(X.T, X) / (m - 1)
    return cov_matrix

# Apply the function
cov_matrix = get_cov_matrix(X)
```

4. Calculate Eigenvalues and Eigenvectors

The eigenvalues (λ) and eigenvectors (v) of the covariance matrix are computed. The eigenvectors, sorted by their corresponding eigenvalues from highest to lowest, are the **principal components**.

General Calculation:
Solve the characteristic equation for the covariance matrix Σ:

$$\Sigma v = \lambda v$$

```
# 'k' is the number of largest eigenvalues/vectors to find
eigenvals, eigenvecs = scipy.sparse.linalg.eigsh(cov_matrix, k=55)
# More robust but less performing option:
# eigenvals, eigenvecs = np.linalg.eigh(cov_matrix)

# Sort them from highest to lowest
eigenvals = eigenvals[::-1]
eigenvecs = eigenvecs[:,::-1]
```

5. Reduce Dimensionality (Projection)

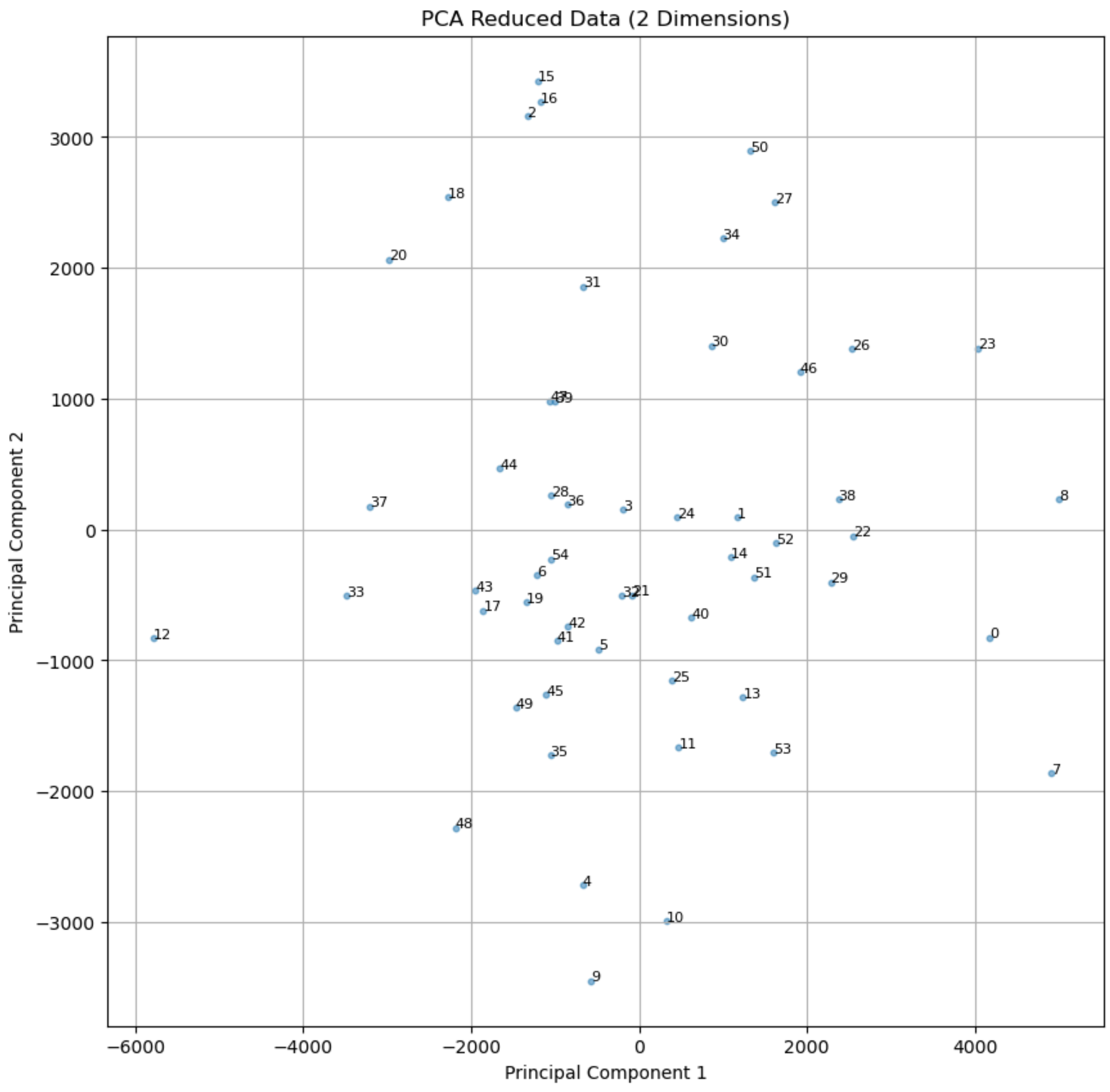
To reduce the dimensionality from 4096 variables to a smaller number *k* (e.g., to 2 dimensions for visualization), the centered data X is projected onto the first *k* principal components (eigenvectors).

General Calculation:

$$X_{red} = X \cdot V_k$$

Where V_k is the matrix formed by the first *k* eigenvectors.

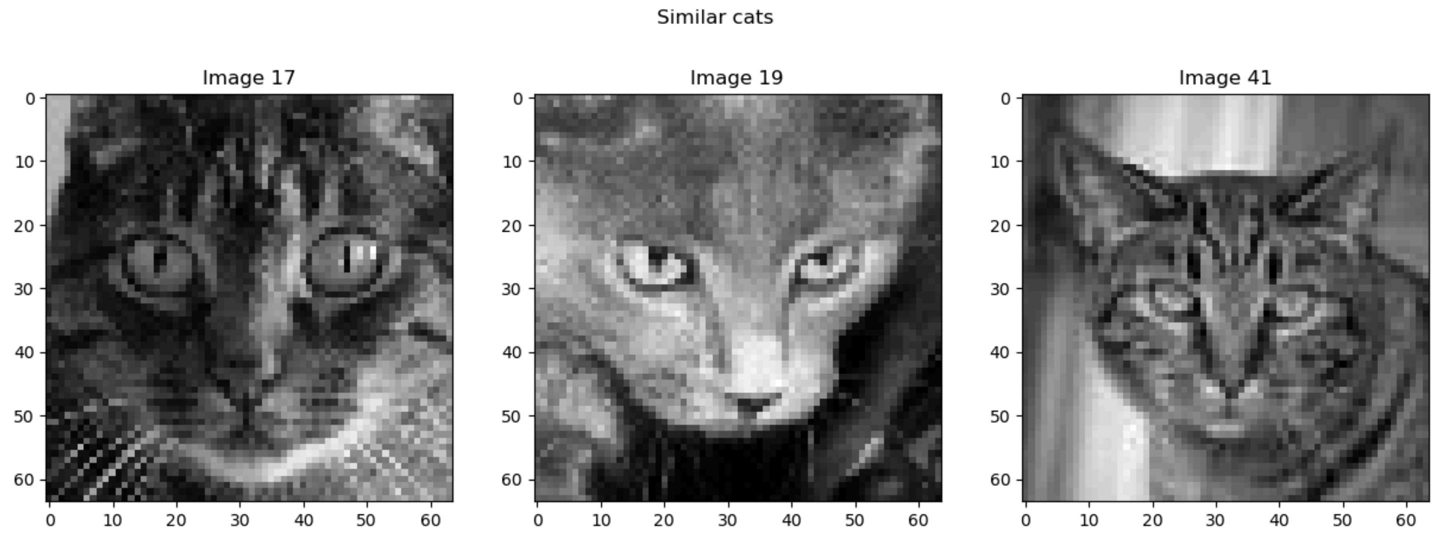
```
def perform_PCA(X, eigenvecs, k):
    # Select the first k eigenvectors
    V_k = eigenvecs[:, :k]
    # Project the data
    X_red = np.dot(X, V_k)
    return X_red

# Reduce to 2 dimensions
X_red_2 = perform_PCA(X, eigenvecs, 2)

# graphing the reduced data with number of image in each point
plt.figure(figsize=(10, 10))
plt.scatter(X_red_2[:, 0], X_red_2[:, 1], s=10, alpha=0.5)
for i in range(len(X_red_2)):
    plt.annotate(str(i), (X_red_2[i, 0], X_red_2[i, 1]), fontsize=8)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA Reduced Data (2 Dimensions)')
plt.grid()
plt.show()
```



In this plot, each dot represents a cat image, reduced to just 2 dimensions. Nearby points correspond to cat images with similar features.

To verify this, we can visualize images 19, 21, and 41, which appear close together on the plot. As you can see, all three cats have very similar features, such as a white muzzle and dark fur around their eyes.

```
fig, ax = plt.subplots(1,3, figsize=(15,5))
ax[0].imshow(imgs[17], cmap='gray')
ax[0].set_title('Image 17')
ax[1].imshow(imgs[19], cmap='gray')
ax[1].set_title('Image 19')
ax[2].imshow(imgs[41], cmap='gray')
ax[2].set_title('Image 41')
plt.suptitle('Similar cats')
```



Similarly, if we take points that are far apart on the graph, such as images 18, 41 and 51, we see that the cats are visually very different from each other.

```
fig, ax = plt.subplots(1,3, figsize=(15,5))
ax[0].imshow(imgs[12], cmap='gray')
ax[0].set_title('Image 12')
ax[1].imshow(imgs[8], cmap='gray')
ax[1].set_title('Image 8')
ax[2].imshow(imgs[15], cmap='gray')
ax[2].set_title('Image 15')
plt.suptitle('Different cats')
```

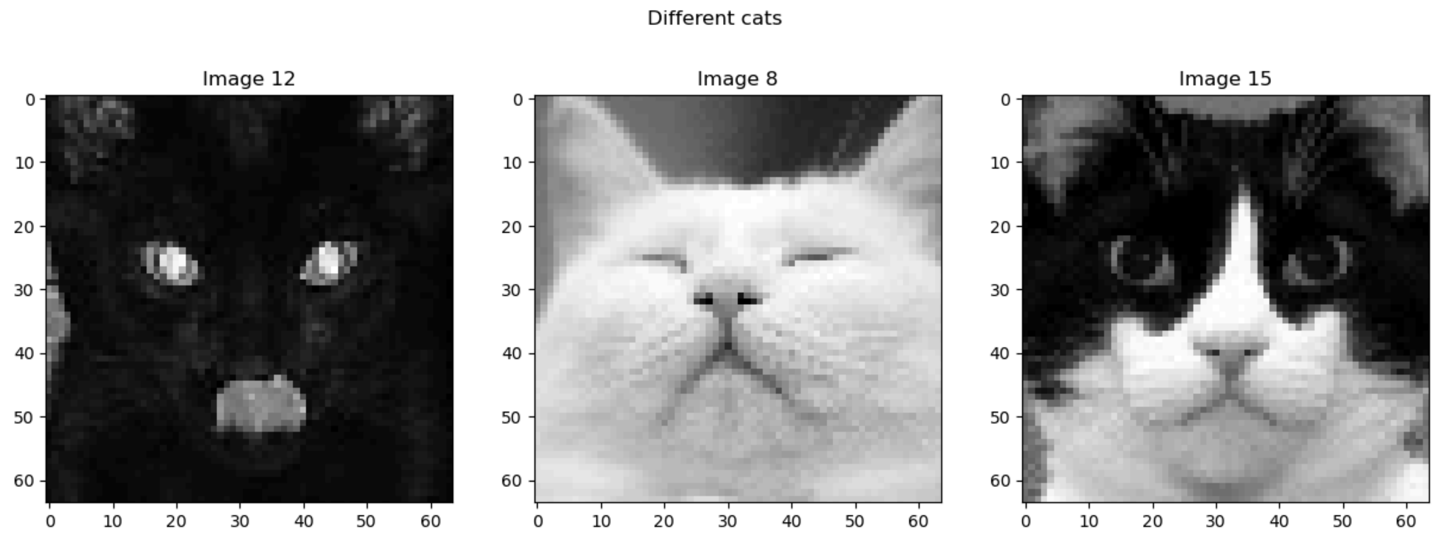


Image Reconstruction from the Components

One of the most visual applications of PCA is seeing how the principal components capture the essential features of the images.

Image Reconstruction Process

We can reverse the PCA projection to reconstruct the original image from the reduced data. This process allows us to visualize how much information has been preserved after compression. A crucial step is to add back the vector of averages that was initially subtracted, to return the image to its original color/brightness space.

General Calculation:

$$X_{rec\_centrado} = X_{red} \cdot V_k^T$$

$$Y_{rec} = X_{rec\_centered} + mean$$

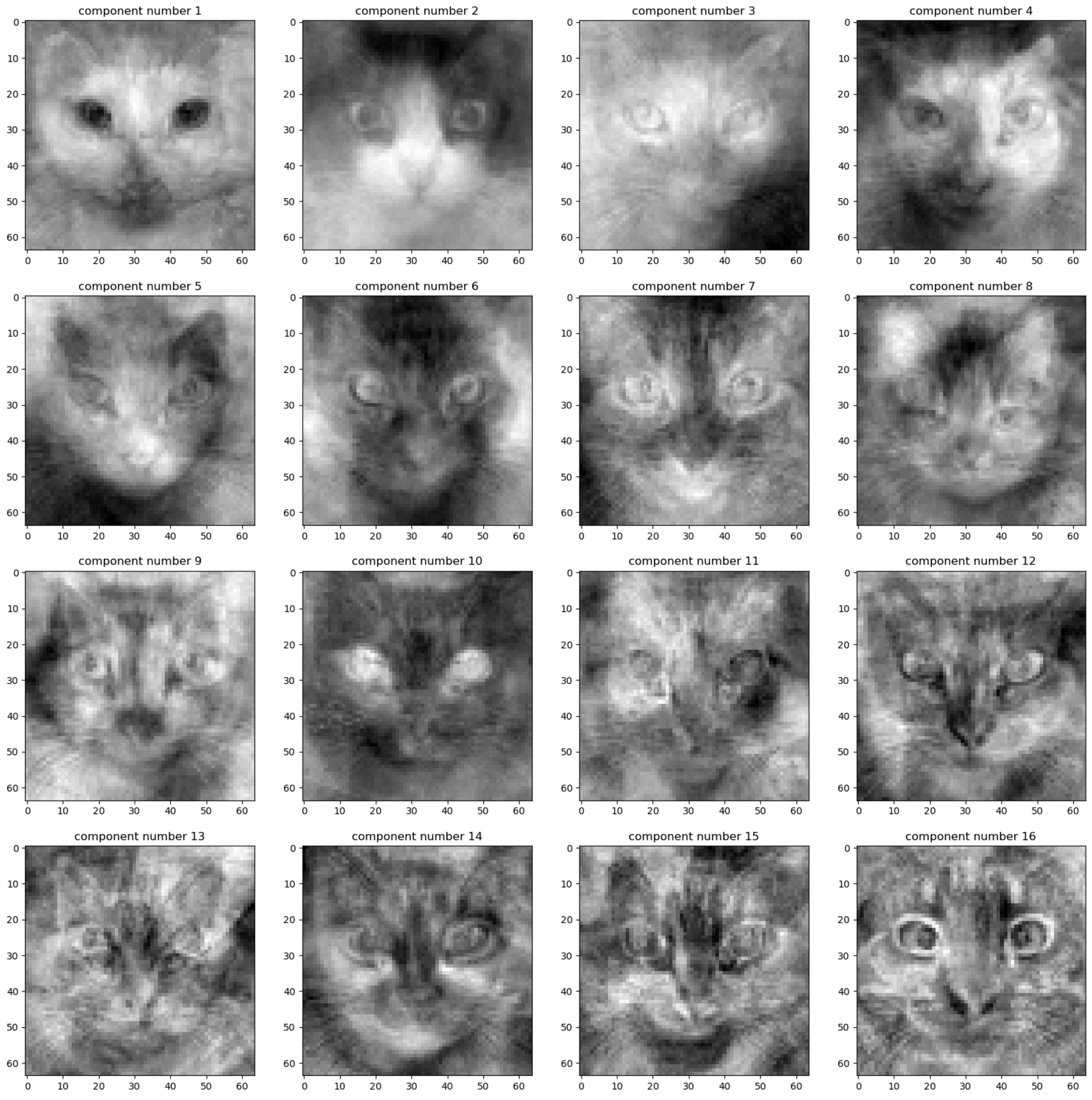
Where X_{red} is the data in the reduced space and V_k^T is the transpose of the matrix with the *k* eigenvectors used.

```python
def reconstruct_image(Xred, eigenvecs, mean_vector):
    # The number of components k is implied by Xred's shape
    k = Xred.shape[1]
    # Reconstruct the image by projecting back to the original space
    X_reconstructed = Xred.dot(eigenvecs[:,:k].T)
    # Add the average to return it to the original space.
    X_reconstructed = X_reconstructed + mean_vector
    return X_reconstructed
```

Visualization of Individual Components

Each principal component can be reshaped to 64x64 to be visualized as an image. These "eigencats" represent fundamental patterns found in the dataset.

```python
height, width = imgs[0].shape
fig, ax = plt.subplots(4,4, figsize=(20,20))
for n in range(4):
    for k in range(4):
        ax[n,k].imshow(eigenvecs[:,n*4+k].reshape(height,width), cmap='gray')
        ax[n,k].set_title(f'component number {n*4+k+1}')
```

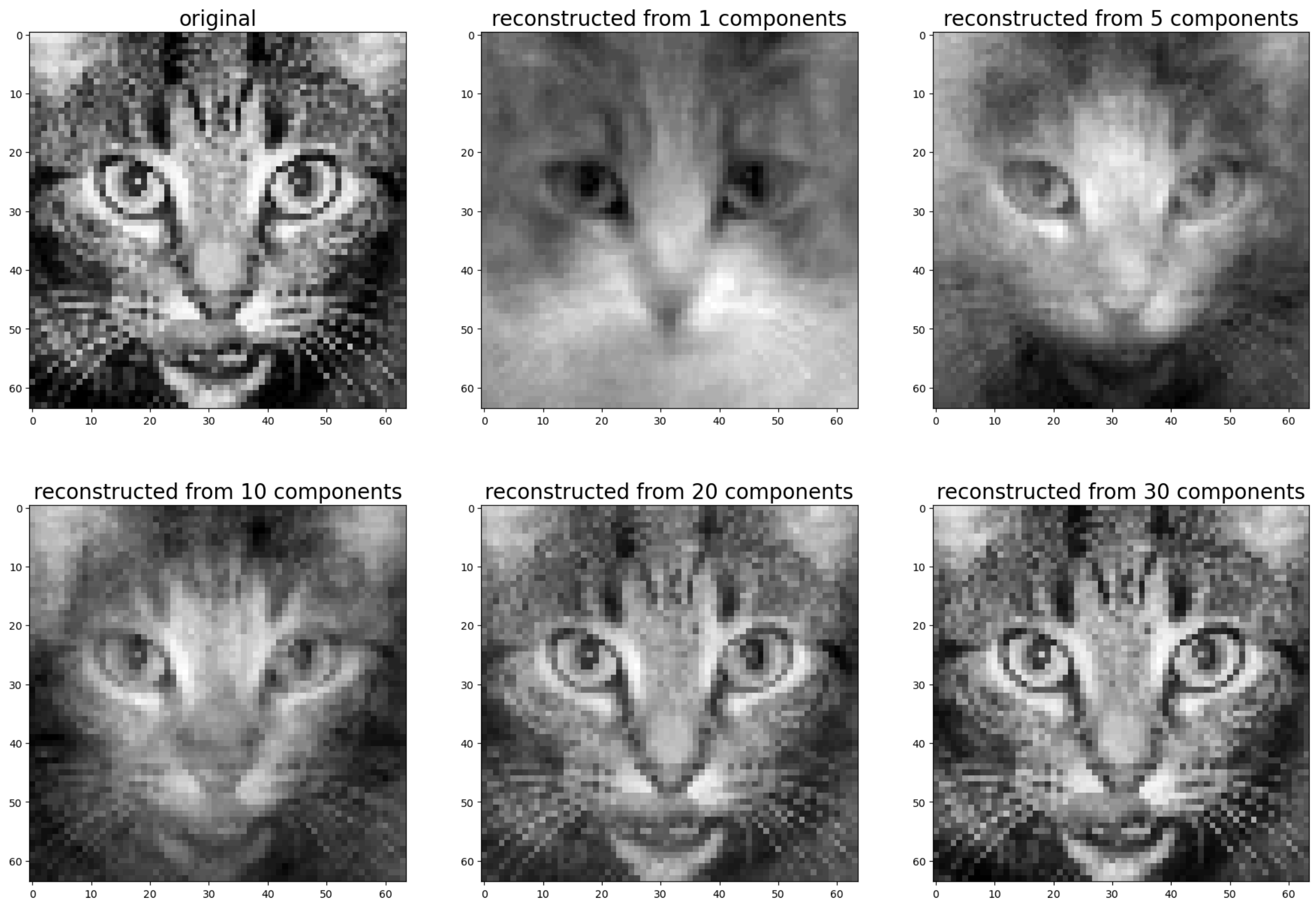


Visualization of the first 16 principal components. The first components capture the general shape of a cat's face, while subsequent ones capture finer details.

Reconstruction with an Increasing Number of Components

As we use more principal components for the reconstruction, the image recovers more detail and looks more like the original, demonstrating the trade-off between compression and fidelity.

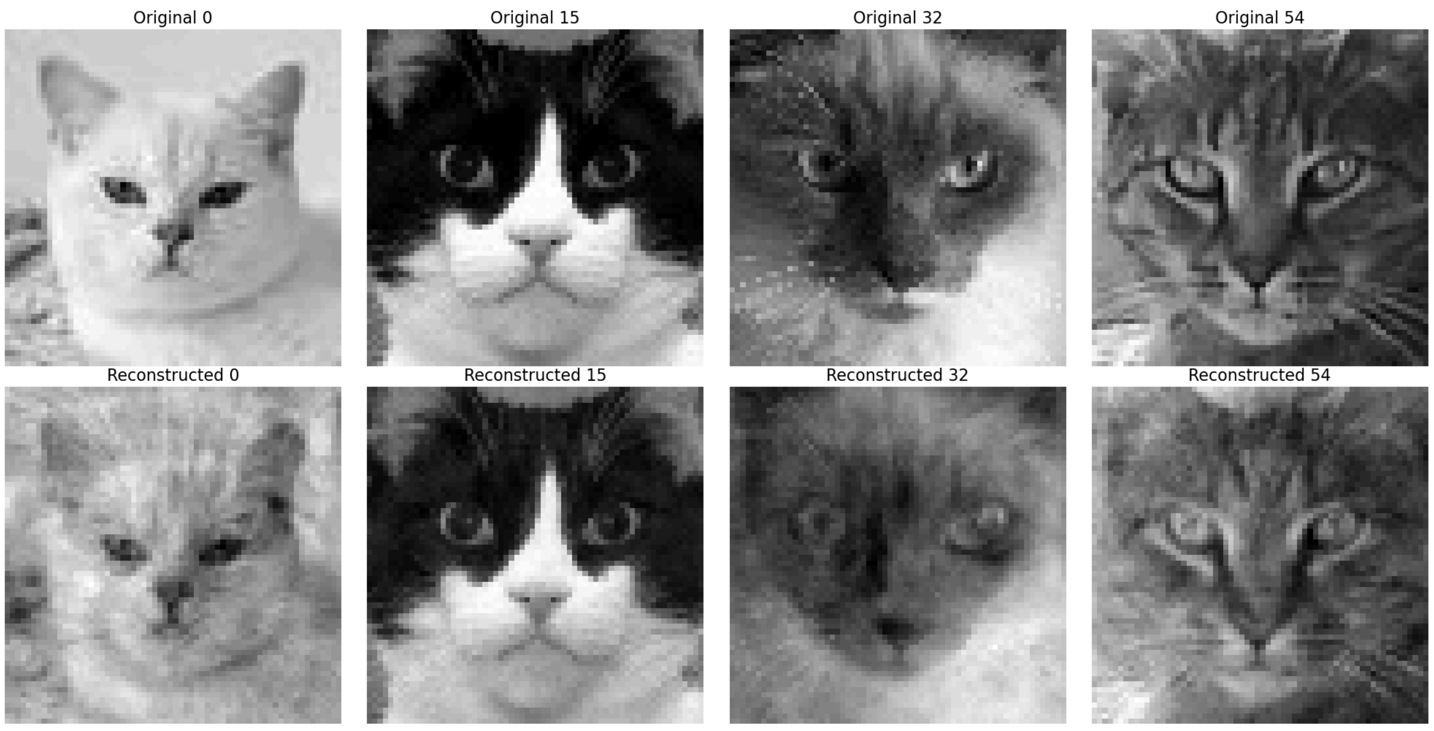
```
Xred1 = perform_PCA(X, eigenvecs,1) # reduce dimensions to 1 component
Xred5 = perform_PCA(X, eigenvecs, 5) # reduce dimensions to 5 components
Xred10 = perform_PCA(X, eigenvecs, 10) # reduce dimensions to 10 components
Xred20 = perform_PCA(X, eigenvecs, 20) # reduce dimensions to 20 components
Xred30 = perform_PCA(X, eigenvecs, 30) # reduce dimensions to 30 components
Xrec1 = reconstruct_image(Xred1, eigenvecs) # reconstruct image from 1 component
Xrec5 = reconstruct_image(Xred5, eigenvecs) # reconstruct image from 5 components
Xrec10 = reconstruct_image(Xred10, eigenvecs) # reconstruct image from 10 components
Xrec20 = reconstruct_image(Xred20, eigenvecs) # reconstruct image from 20 components
Xrec30 = reconstruct_image(Xred30, eigenvecs) # reconstruct image from 30 components

fig, ax = plt.subplots(2,3, figsize=(22,15))
ax[0,0].imshow(imgs[21], cmap='gray')
ax[0,0].set_title('original', size=20)
ax[0,1].imshow(Xrec1[21].reshape(height,width), cmap='gray')
ax[0,1].set_title('reconstructed from 1 components', size=20)
ax[0,2].imshow(Xrec5[21].reshape(height,width), cmap='gray')
ax[0,2].set_title('reconstructed from 5 components', size=20)
ax[1,0].imshow(Xrec10[21].reshape(height,width), cmap='gray')
ax[1,0].set_title('reconstructed from 10 components', size=20)
ax[1,1].imshow(Xrec20[21].reshape(height,width), cmap='gray')
ax[1,1].set_title('reconstructed from 20 components', size=20)
ax[1,2].imshow(Xrec30[21].reshape(height,width), cmap='gray')
ax[1,2].set_title('reconstructed from 30 components', size=20)
```


original
reconstructed from 1 components
reconstructed from 5 components
reconstructed from 10 components
reconstructed from 20 components
reconstructed from 30 components
0
10
20
30
40
50
60

Comparison of an original image with its reconstructions using an increasing number of components. With just 30-35 components, the reconstructed image is nearly identical to the original.

Below, several examples of original images are shown next to their reconstructed version using 35 principal components, demonstrating the high fidelity that can be achieved with a fraction of the original information.

```
Xred35 = perform_PCA(X, eigenvecs, 35) # reduce dimensions to 35 components
Xrec35 = reconstruct_image(Xred35, eigenvecs) # reconstruct image from 35 components

fig, ax = plt.subplots(2, 4, figsize=(20, 10))

indexs = [0, 15, 32, 54]
for i, idx in enumerate(indices):
    # Fila superior: originales
    ax[0, i].imshow(imgs[idx], cmap='gray')
    ax[0, i].set_title(f'Original {idx}', size=16)
    ax[0, i].axis('off')
    # Fila inferior: reconstruidas
    ax[1, i].imshow(Xrec35[idx].reshape(height, width), cmap='gray')
    ax[1, i].set_title(f'Reconstructed {idx}', size=16)
    ax[1, i].axis('off')

plt.tight_layout()
plt.show()
```



References:

- **Course (Intuitive & Practical)**: *Linear Algebra for Machine Learning and Data Science* (deeplearning.ai)
- **Course (Mathematical & Formal)**: *Mathematics for Machine Learning: PCA* (Imperial College London)
- **Book (Advanced):** *Mathematics for Machine Learning*, M. P. Deisenroth, A. A. Faisal, and C. S. Ong., Chapter 10, "Dimensionality Reduction with Principal Component Analysis". (Free Book)