CS540 Fall 2023 Homework 3

1 Assignment Goals

- Explore Principal Component Analysis (PCA) and the related Python packages (`numpy`, `scipy`, and `matplotlib`)
- Make pretty pictures :)

2 Summary

In this project, you'll be implementing a facial analysis program using PCA, using the skills from **the linear algebra + PCA lecture**. You'll also continue to build your Python skills. We'll walk you through the process step-by-step (at a high level).

3 Packages Needed for this Project

You'll use Python 3 with the libraries NumPy, SciPy, and matplotlib (installation instructions linked). You should use a SciPy version `>= 1.5.0` and the following import commands:

```
>>> from scipy.linalg import eigh
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

4 Dataset

You will be using part of the Yale face dataset (processed). The dataset is saved in the 'YaleB_32x32.npy' file. The '.npy' file format is used to store numpy arrays. We will test your code only using this provided dataset.

The dataset contains 2414 sample images, each of size 32×32. We will use n to refer to the number of images (so $n = 2414$) and d to refer to the number of features for each sample image (so $d = 1024 = 32 \times 32$). Note, we'll use x_i to refer to the ith sample image which is a d-dimensional feature vector.

5 Program Specification

Implement these **six** Python functions in `hw3.py` to perform PCA on the dataset:

1. `load_and_center_dataset(filename)`: load the dataset from the provided `.npy` file, center it around the origin, and **return** it as a `numpy` array of floats.
2. `get_covariance(dataset)`: calculate and **return** the covariance matrix of the dataset as a `numpy` matrix ($d \times d$ array).
3. `get_eig(S, m)`: perform eigendecomposition on the covariance matrix S and **return** a diagonal matrix (`numpy` array) with the largest m eigenvalues on the diagonal *in descending order*, and a matrix (`numpy` array) with the corresponding eigenvectors as columns.

4. `get_eig_prop(S, prop)`: similar to `get_eig`, but instead of returning the first m, **return** all eigenvalues and corresponding eigenvectors in a similar format that explain more than a `prop` proportion of the variance (specifically, please make sure the eigenvalues are returned in descending order).
5. `project_image(image, U)`: project each $d \times 1$ image into your m-dimensional subspace (spanned by m vectors of size $d \times 1$) and **return** the new representation as a $d \times 1$ `numpy` array.
6. `display_image(orig, proj)`: use `matplotlib` to display a visual representation of the original image and the projected image side-by-side.

5.1 Load and Center the Dataset ([20] points)

You'll want to use the the `numpy` function `load()` to load the `YaleB_32x32.npy` file into Python.

```
>>> x = np.load(filename)
```

This should give you a $n \times d$ dataset (recall that $n = 2414$ is the number of images in the dataset and $d = 1024$ is the number of dimensions of each image). In other words, each row represents an image feature vector.

Your next step is to center this dataset around the origin. Recall the purpose of this step from lecture — it is a technical condition that makes it easier to perform PCA, but it does not lose any important information.

To center the dataset is simply to subtract the mean μ_x from each data point x_i (image, in our case), i.e., $x_i^{\text{cent}} = x_i - u_x$, where

$$\mu_x = \frac{1}{n}\sum_{i=1}^{n} x_i.$$

You can take advantage of the fact that `x` (as defined above) is a `numpy` array and, as such, has this convenient behavior:

```
>>> x = np.array([[1,2,5],[3,4,7]])
>>> np.mean(x, axis=0)
array([2., 3., 6.])
>>> x - np.mean(x, axis=0)
array([[-1., -1., -1.],
       [ 1.,  1.,  1.]])
```

After you've implemented this function, it should work like this:

```
>>> x = load_and_center_dataset('YaleB_32x32.npy')
>>> len(x)
2414
>>> len(x[0])
1024
>>> np.average(x)
-8.315174931741023e-17
```

(Its center isn't *exactly* zero, but taking into account precision errors over 2414 arrays of 1024 floats, it's what we call "close enough.")

From now on, we will use x_i to refer to x_i^{cent}.

5.2 Find the Covariance Matrix ([15] points)

Recall, from lecture, that one of the interpretations of PCA is that it is the eigendecomposition of the sample covariance matrix. We will rely on this interpretation in this assignment, with all of the information you need below.

The covariance matrix is defined as

$$S = \frac{1}{n-1}\sum_{i=1}^{n} x_i x_i^\intercal.$$

Note that x_i is one of the n images in the (centered) dataset and is considered to be a column vector of size $d \times 1$.

To calculate S, you'll need a couple of tools from **numpy** again:

```
>>> x = np.array([[1,2,5],[3,4,7]])
>>> np.transpose(x)
array([[1, 3],
       [2, 4],
       [5, 7]])
>>> np.dot(x, np.transpose(x))
array([[30, 46],
       [46, 74]])
>>> np.dot(np.transpose(x), x)
array([[10, 14, 26],
       [14, 20, 38],
       [26, 38, 74]])
```

The result of this function for our sample dataset should be a $d \times d$ (or 1024×1024) matrix.

5.3 Get the m Largest Eigenvalues and their Eigenvectors ([17] points)

Again, recall from lecture that eigenvalues and eigenvectors are useful objects that characterize matrices. Better yet, PCA can be performed by doing an eigendecomposition and taking the eigenvectors corresponding to the largest eigenvalues. This replaces the recursive deflation step we discussed in class.

You may find `scipy.linalg.eigh` from the `scipy` library very helpful when writing this function. The optional parameter called `subset_by_index` might be of particular use.

Return the *largest m eigenvalues* of S as a diagonal matrix, in descending order, and the corresponding eigenvectors as columns in a matrix.

To return more than one thing from a function in Python, you can do this:

```
def multi_return():
    return "a string", 5
my_string, my_int = multi_return()
```

Make sure to return the diagonal matrix of eigenvalues *first*, then the eigenvectors in corresponding columns. You may have to rearrange the output of `eigh` to get the eigenvalues in decreasing order and *make sure to keep the eigenvectors in the corresponding columns* after that rearrangement.

```
>>> S = get_covariance(x)
>>> Lambda, U = get_eig(S, 2)
>>> print(Lambda)
[[1369142.41612494       0.        ]
 [      0.         1341168.50476773]]
>>> print(U)
[[-0.01304065 -0.0432441 ]
 [-0.01177219 -0.04342345]
 [-0.00905278 -0.04095089]
 ...
 [ 0.00148631 0.03622013]
 [ 0.00205216 0.0348093 ]
 [ 0.00305951 0.03330786]]
```

5.4 Get all Eigenvalues/Eigenvectors that Explain More than a Certain Proportion of the Variance ([8] points)

Instead of taking the top m eigenvalues/eigenvectors, we may want *all* the eigenvectors that explain more than a certain proportion of the variance. Let λ_i be an eigenvalue of the covariance matrix S. Then λ_i's corresponding eigenvector's proportion of variance is

$$\frac{\lambda_i}{\sum_{j=1}^{n} \lambda_j}.$$

Return the eigenvalues as a diagonal matrix, in descending order, and the corresponding eigenvectors as columns in a matrix. Hint: `subset_by_index` was useful for the previous function, so perhaps something similar could come in handy here. What is the trace of a matrix?

Again, make sure to return the diagonal matrix of eigenvalues *first*, then the eigenvectors in corresponding columns. You may have to rearrange the output of `eigh` to get the eigenvalues in decreasing order and *make sure to keep the eigenvectors in the corresponding columns* after that rearrangement.

```
>>> Lambda, U = get_eig_prop(S, 0.07)
>>> print(Lambda)
[[1369142.41612494       0.        ]
 [      0.         1341168.50476773]]
>>> print(U)
[[-0.01304065 -0.0432441 ]
 [-0.01177219 -0.04342345]
 [-0.00905278 -0.04095089]
 ...
 [ 0.00148631 0.03622013]
 [ 0.00205216 0.0348093 ]
 [ 0.00305951 0.03330786]]
```

5.5 Understanding PCA

Optional: A rigorous mathematical exploration of PCA.

Let's describe PCA a little more formally. Let $X = [x_1, x_2, ..., x_n] \in \mathbb{R}^{d\times n}$ represent our centered data matrix, where n is the number of data samples and d the data dimensionality or number of features. In this section, we will differentiate

1. The PCA Projection of our data $U^T X$, which is in $\mathbb{R}^{m\times n}$
2. The PCA Reconstruction of our data X^{pca}, which is in $\mathbb{R}^{d\times n}$

5.5.1 PCA Projection

Recall that the covariance matrix of our data

$$S = \frac{1}{n-1} X^T X$$

Recall that our goal is to **minimize projection error** (sums of squared distances) while reducing the data dimensionality d. We can perform an "eigendecomposition" of the covariance matrix S, where we represent it in terms of its eigenvectors and corresponding eigenvalues as:

$$S = V \Lambda V^T$$

where the columns of $V \in \mathbb{R}^{d\times d}$ is a $d \times d$ are eigenvectors or *principal components* of S, and Λ is a diagonal matrix of eigenvalues. To reduce the dimensionality of our data, we will consider only the *top* m eigenvalues of V, or its first m **principal components**, and call this matrix $U \in \mathbb{R}^{d\times m}$.

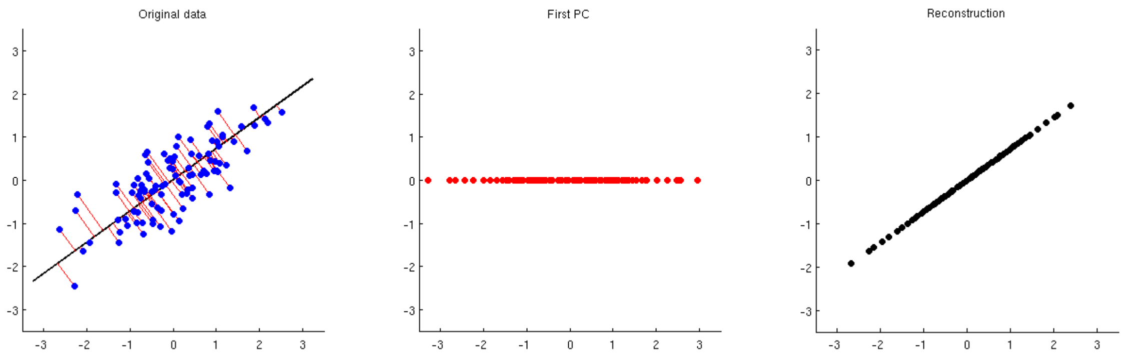


Figure 1: A visualization of PCA [1] with the blue dots representing some data points $X\in \mathbb{R}^2$. The black line is the axis of the first principal component, which maximizes the variance of the data (sums of squared distances). The second plot shows the value of the PCA projection as red dots, i.e $Z\in \mathbb{R}^1$, where we have reduced the dimensionality of our data from 2 to 1. Finally the last plot shows a reconstruction of our data $\mathbf{X^{pca}} \in \mathbb{R}^2$ in black dots, with visible reconstruction error when compared to the original data X in the first plot.

The PCA Projection or "scores" of our data are thus given by a simple matrix product:

$$Z = U^T X, Z \in \mathbb{R}^{m\times n}$$

Note that a data point $x_i \in X, i \leq n$ has been projected to $z_i \in \mathbb{R}^m$, determined by the m eigenvectors of S contained in U. What if we want to reconstruct our data in our original feature space, i.e. $x_i^{pca} \in \mathbb{R}^d$ from our PCA projection z_i?

5.5.2 PCA Reconstruction

We will now use the eigenvectors U to project our data Z back to our original feature space $\mathbb{R}^d$.

$$X^{pca} = UZ = UU^T X, X^{pca} \in \mathbb{R}^d$$

Hint: if we had not reduced the number of eigenvectors during PCA projection, i.e. $m = d$, then $U = V$. Since $VV^T = I$ (the identity matrix), $X^{pca} = VV^T X = IX = X$ and we can perfectly reconstruct our original data. We encourage to utilize this fact while debugging your code.

5.6 Project the Images ([15] points)

Given an image in the dataset and the eigenvectors from `get_eig` (or `get_eig_prop`), compute the PCA representation of the image.

Let u_j represent the jth column of `U`. Every u_j is an eigenvector of S with size $d \times 1$. If `U` has m eigenvectors, the image x_i is projected into an m dimensional subspace. The PCA projection represents images as a weighted sum of the eigenvectors. This projection only needs to store the weight for each eigenvector (m-dimensions) instead of the entire image (d-dimensions). The projection $\alpha_i \in \mathbb{R}^m$ is computed such that $\alpha_{ij} = u_j^T x_i$.

The full-size representation of α_i can be computed as $x_i^{\text{pca}} = \sum_{j=1}^m \alpha_{ij} u_j$. Notice that each eigenvector u_j is multiplied by its corresponding weight α_{ij}. The reconstructed image, x_i^{pca}, will not necessarily equal the original image because of the information lost projecting x_i to a smaller subspace. This information loss will increase as less eigenvectors are used. Implement `project_image` to compute x_i^{pca} for an input image.

```
>>> projection = project_image(x[0], U)
>>> print(projection)
[6.84122225 4.83901287 1.41736694 ... 8.75796534 7.45916035 5.4548656 ]
```

5.7 Visualize ([25] points)

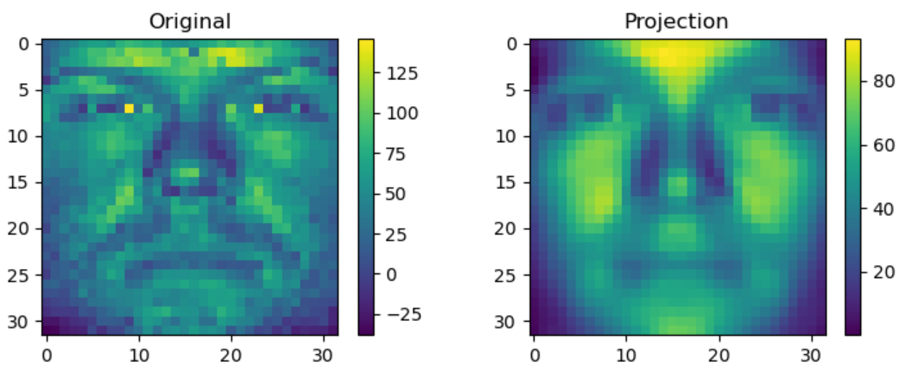
We'll be using `matplotlib`'s `imshow`.

Follow these steps to visualize your images:

1. Reshape the images to be 32×32 (before this, they were being thought of as 1 dimensional vectors in $\mathbb{R}^{1024}$).
2. Create a figure with one row of two subplots with `fig`, `ax1` and `ax2` objects. `ax1` should represent the original image, and `ax2` should represent the reconstructed image.
3. Title the first subplot (the one on the left) as "Original" (without the quotes) and the second (the one on the right) as "Projection" (also without the quotes).
4. Use `imshow` with the optional argument `aspect='equal'`
5. Use the return value of `imshow` to create a `colorbar` for each image.
6. Return the `fig`, `ax1` and `ax2` objects used in step 2 from `display_image()`.
7. Testing: Render your plots. DO NOT include this in your submission, this is only to test your implementation. We suggest calling `display_image()` from `main()`, and rendering images there.

Below is a simple snippet of code for you to test your functions. Do **not** include it in your submission!

```
>>> x = load_and_center_dataset('YaleB_32x32.npy')
>>> S = get_covariance(x)
>>> Lambda, U = get_eig(S, 2)
>>> projection = project_image(x[0], U)
>>> fig, (ax1, ax2) = display_image(x[0], projection)
```



6 Submission Notes

Please submit your files in a `.zip` archive named `hw3_<netid>.zip`, where you replace `<netid>` with your netID (i.e., your wisc.edu login). Inside your zip file, there should be **only** one file named `hw3.py`. Do *not* submit a Jupyter notebook `.ipynb` file.

Be sure to **remove all debugging output** before submission; failure to do so will be **penalized ([10] points)**:

- Your functions should run silently (except for the image rendering window in the last function).
- No code should be put outside the function definitions (except for import statements; helper functions are allowed).

ALL THE BEST!

References

[1] amoeba. Making sense of principal component analysis, eigenvectors & eigenvalues. Cross Validated. url: https://stats.stackexchange.com/q/140579 (version: 2022-08-31).