CMPE 482 - Assignment 4

Spring 2021

Due 02.06.2021, 20.00

Correction on 01.06.2021

Warning: Note that we finished *IALA* and are now using the book *Mathematics for Machine Learning* (MML). Any pages cited or the phrase "our book" will refer to MML from now on, unless otherwise specified.

1. **(3 points)** Let $A = \begin{bmatrix} 1 & 6 \\ 0 & -1 \\ -1 & -2 \end{bmatrix}$.
   - **a.** Compute the SVD of $A = U\Sigma V^T$, without using a previous implementation of it (e.g. do not use `numpy.linalg.svd`). However, you can use a previous implementation of eigendecomposition to obtain your solution (e.g. `numpy.linalg.eig`). Make sure to show your solution clearly - you do not have to write out whole matrices in latex, but your path to the solution should be clear mathematically and programatically. Print U, Σ, and V in their full dimensions. After this, compare your solution to that obtained by using `numpy.linalg.svd()`. **Hint**: Read the documentation for `numpy.linalg.svd()` carefully to understand the objects it is returning and how they correspond to your results, including but not limited to whether it is returning V or V^T.
   - **b.** For an A with real elements, we can be sure that there must exist an orthogonal U and V. Why should this be true? (**Hint**: The relevant book chapter might include a theorem that helps with this.)
   - **c.** Using your SVD results and using only the largest singular value, create a rank-one approximation of A called $\hat{A}$. Show that your approximation error conforms to that predicted by the Eckhart-Young Theorem (Pg. 131, Theorem 4.25).
2. **(4 points)** In this question, we will examine the transformation done by our matrix in a more detailed fashion, similar to Figure 4.8 (Pg. 120) in our book. We will use the fact that $Ax = U\Sigma V^T x$. While doing the plotting tasks below it might be a good idea to think of coloring your vectors, fixing your axes etc. to make the plots more interpretable.
   - **a.** Create 12 two-dimensional unit vectors (vectors of lengh 1). They have to start from $x_1 = (1, 0)$ and they should be 30 degrees apart each, and numbering should go counterclockwise. Stack them horizontally inside a 2×12 matrix called X. Print X.
   - **b.** Plot all 2-dimensional vectors in X on the 2-dimensional plane.
   - **c.** Multiply each vector with V^T. Plot the resulting vectors on the 2-dimensional plane.
   - **d.** Take the results of (**2c**) and multiply each vector with Σ, and plot each vector on the 3-dimensional plane. (Plotting 3-dimensional vectors are slightly more involved but still very much doable with `matplotlib` package in Python. A quick web search will turn out numerous open source results.)
   - **e.** Take the results of (**2d**) and multiply each vector with U, and plot each vector on the 3-dimensional plane.

f. Recall $\hat{A}$ from (**1c**). Let $\hat{A} = U\hat{\Sigma}V^T$, where U and V^T are same as above and $\hat{\Sigma}$ is the version of Σ where all elements are 0 besides the first singular value. Plot all 4 stages you plotted above for $\hat{A}$ as well.

g. Compare the last images you produced with A and $\hat{A}$. In what sense vectors produced by multiplying $\hat{A}$ can be said to produce a "summary" of those produced by A? Comment.

3. (5 points) In this question, we will use SVD for image compression. There is an example of this in our book as well. However, we will use a slightly different method than our book. We will be using three examples from the Olivetti faces dataset included in `sklearn` package. This data set includes face images stored in the form of 64×64 arrays with elements ranging from 0 to 1, with each element corresponding to a pixel. We will ignore the last row and column of pixels for practical reasons, so our images will be arrays of size 63×63.

a. Use the **Code Snippet 1** below to download, and select the three faces from the data set. We will call them A_1, A_2, and A_3, similar to the snippet. As seen in the code, `matplotlib.pyplot.imshow` is a good function to visualize arrays that represent images.

b. We will now extract 7×7 patches from the images in row-major order (i.e. take a patch, slide right, take another patch, when at the right end of the image, go below and start again). The patches should not overlap. Partitioned this way, each image comprises 81 patches. We will flatten each patch and stack them vertically to create a transformed version of these images, respectively named B_1, B_2, and B_3, each with dimensions 81×49.

Hint 1: `numpy` arrays have a built-in function called `flatten()` that can be used to create a 1-d vector from a matrix. You can reverse it with `np.reshape(your_array, (dimensions))`. Both assumes row-major order unless otherwise specified.

Hint 2: It would be wise to first create functions that create B_1 when given A_1, and create A_1 when given B_1. See **Code Snippet 2** for an example.

c. Conduct an SVD to obtain the factorization $B_1 = U_1\Sigma_1V_1^T$. Print the first 10 singular values. **Hint**: You may use `numpy.linalg.svd` here, but make sure you interpret the output correctly. You may or may not want to use the parameter `full_matrices` depending on your approach.

d. In this setting, what does the columns of V_1 correspond to? Comment. Show the first three columns of V_1 as 7×7 images. Are they interpretable? Interpret at least some of them. **Hint**: The colors are meaningful in a relative sense since these values can be scaled or negated when the full matrix is being reconstructed.

e. Using your decomposition, create a rank-1, rank-5, rank-10, and rank-20 approximation of B_1. Convert each approximation back to its face-shape and display each of them as face images.

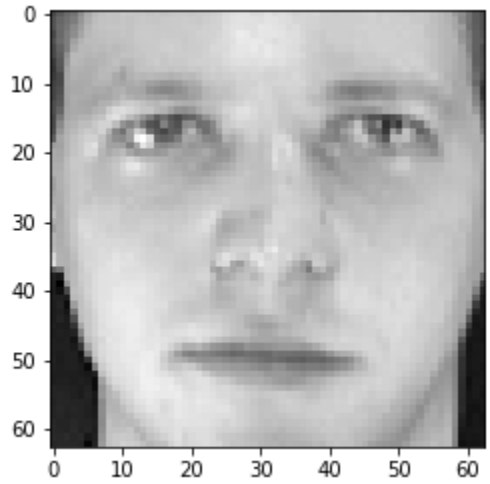
f. Notice that when we obtain, for example, a rank-5 approximation of B_1, we actually do not have to store all elements of U and V but only the first five columns. This means a reasonably faithful approximation could actually save us a lot of space. Imagine we are using a sparse representation (we only have to store non-zero elements). Given that we

are using $63 \times 63 = 3969$ parameters to store the original image, what fraction of parameters do we have to store for each of the approximations above compared to the original images parameters? Are we saving space with the approximations above?

g. We will now use the results of the SVD of A_1 to decompose the other images. Take only the first 20 columns of V_1 and call this matrix $V_{1,(20)}$. Observe the linear systems $B_2 = Y_2 V_{1,(20)}^T$ and $B_3 = Y_3 V_{1,(20)}^T$, where $V_{1,(20)}$ stands in for V_2 and V_3, and Y_2 and Y_3 stand in for $U_2\Sigma_2$ and $U_3\Sigma_3$. Find the best values of Y_2 and Y_3, and call them $\hat{Y}_2$ and $\hat{Y}_3$. Finally, compute the approximations to B_2 and B_3 by computing $\hat{Y}_2 V_{1,(20)}^T$ and $\hat{Y}_3 V_{1,(20)}^T$, convert them to original shape and show these approximations as images. Did these approximations work? Why? Which one worked better? Why?

Code Snippet 1

```
import matplotlib.pyplot as plt
from sklearn import datasets
faces = datasets.fetch_olivetti_faces()
A_1 = faces["images"][0][:63, :63]
A_2 = faces["images"][110][:63, :63]
A_3 = faces["images"][365][:63, :63]
plt.imshow(A_1, cmap="gray");
```



Code Snippet 2

```
plt.imshow(A_1, cmap="gray");
plt.figure()
plt.imshow(create_patches(A_1), cmap="gray");
plt.figure()
plt.imshow(create_face(create_patches(A_1)), cmap="gray");
```