CMPE 482 - Assignment 4
Spring 2021
Due 02.06.2021, 20.00

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. <u>Compute</u> the SVD of $A = U\Sigma V^T$, <u>without</u> using a previous implementation of it (e.g. do not use `numpy.linalg.svd`). However, you <u>can</u> use a previous implementation of eigendecomposition to obtain your solution (e.g. `numpy.linalg.eig`). Make sure to <u>show</u> your solution clearly - you do <u>not</u> have to write out whole matrices in latex, but your path to the solution should be <u>clear</u> mathematically and programatically. <u>Print</u> U, Σ, and V in their full dimensions. After this, <u>compare</u> 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. <u>Why</u> 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, <u>create</u> a rank-one approximation of A called $\hat{A}$. <u>Show</u> 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. <u>Create</u> 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. <u>Stack</u> them horizontally inside a 2×12 matrix called X. <u>Print</u> X.

    b. <u>Plot</u> all 2-dimensional vectors in X on the 2-dimensional plane.

    c. <u>Multiply</u> each vector with V^T. <u>Plot</u> the resulting vectors on the 2-dimensional plane.

    d. Take the results of (**2c**) and <u>multiply</u> each vector with Σ, and <u>plot</u> 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 <u>multiply</u> each vector with U, and <u>plot</u> 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. <u>Plot</u> all 4 stages you plotted above for $\hat{A}$ as well.

g. <u>Compare</u> the <u>last</u> 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? <u>Comment</u>.

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 <u>Olivetti faces dataset</u> 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 <u>download</u>, and <u>select</u> 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 <u>extract</u> 7×7 patches from the images in <u>row-major order</u> (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 <u>not</u> overlap. Partitioned this way, each image comprises 81 patches. We will <u>flatten</u> each patch and <u>stack</u> them vertically to <u>create</u> 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. <u>Conduct an SVD</u> to obtain the factorization $B_1 = U_1\Sigma_1V_1^T$. <u>Print</u> 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? <u>Comment</u>. <u>Show</u> the first three columns of V_1 as 7×7 images. Are they interpretable? <u>Interpret</u> 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, <u>create</u> a rank-1, rank-5, rank-10, and rank-20 approximation of B_1. <u>Convert</u> each approximation back to its face-shape and <u>display</u> 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, <u>what fraction of parameters</u> do we have to store for each of the approximations above compared to the original images parameters? <u>Are we saving space</u> with the approximations above?

g. We will now use the results of the SVD of A_1 to decompose the other images. <u>Take</u> 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 $U_3 \Sigma_3$. <u>Find</u> the best values of Y_2 and Y_3, and call them $\hat{Y}_2$ and $\hat{Y}_3$.

Finally, <u>compute</u> the approximations to B_2 and B_3 by computing $\hat{Y}_2 V_{1,(20)}^T$ and $\hat{Y}_3 V_{1,(20)}^T$, <u>convert</u> them to original shape and <u>show</u> these approximations as images. Did these approximations work? <u>Why</u>? Which one worked better? <u>Why</u>?

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");
```
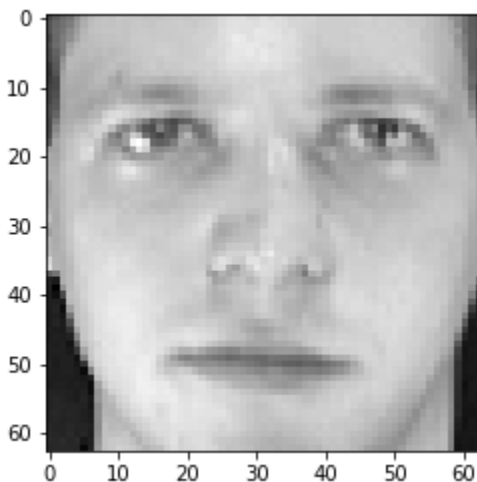
Code Snippet 2

```python
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");
```