LAB 13 - SINGULAR VALUE DECOMPOSITIONS AND IMAGE COMPRESSION

1. The singular value decomposition

Although we often think of $n \times m$ matrices as describing linear transformations from $\mathbb{R}^m$ to $\mathbb{R}^n$, there are many other uses for matrices and the techniques we study in linear algebra. In this lab we will study an application of the *singular value decomposition* (or SVD) of matrices.

Recall that any $n \times m$ matrix A can be written as

$$A = U\Sigma V^T,$$

where U is an $n \times n$ orthogonal matrix, V is an $m \times m$ orthogonal matrix, and Σ is an $n \times m$ matrix of the form

$$\Sigma = \left[\begin{array}{cccc:ccc} \sigma_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_r & 0 & \cdots & 0 \\ \hdashline 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \end{array}\right] = \begin{bmatrix} D & O_1 \\ O_2 & O_3 \end{bmatrix}. \tag{1}$$

Here D denotes an $r \times r$ diagonal matrix with positive entries on the diagonal, while O_1, O_2, and O_3 are zero matrices of the appropriate size. We order the σ_j in decreasing order so that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$. Here r is equal to the rank of the matrix A.

We call such a factorization of A a *singular value decomposition of* A, and the values σ_j are the nonzero *singular values* of A. The columns of U are called *left singular vectors* of A, while the columns of V (which are the rows of V^T) are called *right singular vectors* of A.

2. The outer product form of the SVD

One way to think of an SVD of a matrix A is that it provides us a way to approximate A as the sum of a collection of much simpler matrices. To make this more precise, let $\mathbf{u}_1, \cdots, \mathbf{u}_n$ be the columns of U, let $\mathbf{v}_1, \cdots, \mathbf{v}_m$ be the columns of V (so that $\mathbf{v}_1^T, \cdots, \mathbf{v}_m^T$ are the rows of V^T), and consider the following computation:

$$A = U\Sigma V^T = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_n \end{bmatrix} \left[\begin{array}{cccc:ccc} \sigma_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_r & 0 & \cdots & 0 \\ \hdashline 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \end{array}\right] \begin{bmatrix} \mathbf{v}_1^T \\ \mathbf{v}_2^T \\ \vdots \\ \mathbf{v}_m^T \end{bmatrix}$$

$$= \begin{bmatrix} \sigma_1\mathbf{u}_1 & \sigma_2\mathbf{u}_2 & \cdots & \sigma_r\mathbf{u}_r & \mathbf{0} & \cdots & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{v}_1^T \\ \mathbf{v}_2^T \\ \vdots \\ \mathbf{v}_m^T \end{bmatrix}$$

$$= \sigma_1\mathbf{u}_1\mathbf{v}_1^T + \sigma_2\mathbf{u}_2\mathbf{v}_2^T + \cdots + \sigma_r\mathbf{u}_r\mathbf{v}_r^T + \mathbf{0}\mathbf{v}_{r+1}^T + \cdots + \mathbf{0}\mathbf{v}_m^T$$

$$= \sigma_1\mathbf{u}_1\mathbf{v}_1^T + \sigma_2\mathbf{u}_2\mathbf{v}_2^T + \cdots + \sigma_r\mathbf{u}_r\mathbf{v}_r^T.$$

- Note that each term $\mathbf{u}_j\mathbf{v}_j^T$ in the above sum is an $n \times m$ matrix with rank equal to 1. Can you see why this is true?

We can thus write

$$A = \sigma_1\mathbf{u}_1\mathbf{v}_1^T + \sigma_2\mathbf{u}_2\mathbf{v}_2^T + \cdots + \sigma_r\mathbf{u}_r\mathbf{v}_r^T,$$

where each term $\sigma_j\mathbf{u}_j\mathbf{v}_j^T$ is a rank 1 matrix. We call this expression the *outer product form of the SVD*. We think of this as giving us a way to break up the matrix A (which may have very high rank) into the sum of a collection of simpler rank 1 matrices. Each piece $\mathbf{u}_j\mathbf{v}_j^T$ contains a small amount of information from the original matrix A, and the singular value σ_j tells us how large of a contribution $\mathbf{u}_j\mathbf{v}_j^T$ makes to the overall matrix.

To make this more concrete, consider the matrix

$$A = \begin{bmatrix} 100 & -0.05 & 100 & -0.05 \\ 0 & -90 & 0 & 90 \\ 100 & 0.05 & 100 & 0.05 \end{bmatrix},$$

which you can check has rank 3 by row reducing. A singular value decomposition of A is given by

$$A = U\Sigma V^T = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ 0 & 1 & 0 \\ \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 200 & 0 & 0 & 0 \\ 0 & 90\sqrt{2} & 0 & 0 \\ 0 & 0 & 0.1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} \\ 0 & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 & 0 & -\frac{1}{\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \end{bmatrix}^T.$$

We therefore have

$$\sigma_1\mathbf{u}_1\mathbf{v}_1^T = 200 \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \end{bmatrix} = 200 \begin{bmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \end{bmatrix}$$

$$\sigma_2\mathbf{u}_2\mathbf{v}_2^T = 90\sqrt{2} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & -\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{bmatrix} = 90\sqrt{2} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\sigma_3 \mathbf{u}_3 \mathbf{v}_3^T = 0.1 \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{bmatrix} = 0.1 \begin{bmatrix} 0 & -\frac{1}{2} & 0 & -\frac{1}{2} \\ 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix}.$$

Thus, we can write

$$A = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^T + \sigma_3 \mathbf{u}_3 \mathbf{v}_3^T$$

$$= 200 \begin{bmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \end{bmatrix} + 90\sqrt{2} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ 0 & 0 & 0 & 0 \end{bmatrix} + 0.1 \begin{bmatrix} 0 & -\frac{1}{2} & 0 & -\frac{1}{2} \\ 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix},$$

where each of the 3×3 matrices in the above sum has rank 1. Notice, however, that not all three terms contribute the same amount to the matrix A. Indeed, the coefficient on the third term (which is $\sigma_3 = 0.1$) is much smaller than the other two coefficients. This means that the third term doesn't contribute much to the values in the matrix A, relative to the first two terms (whose coefficients are $\sigma_1 = 200$ and $\sigma_2 = 90\sqrt{2} \approx 127.279$, respectively).

What happens if we ignore the third term altogether, and don't include it in the above sum? In this case we get a new matrix A_2, given by

$$A_2 = 200 \begin{bmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \end{bmatrix} + 90\sqrt{2} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 100 & 0 & 100 & 0 \\ 0 & -90 & 0 & 90 \\ 100 & 0 & 100 & 0 \end{bmatrix}.$$

- Compare the matrix A_2 to our original matrix A. How much does A_2 differ from A?

The matrix A_2 has rank 2, and closely approximates our original matrix A. In other words, we are able to approximate our rank 3 matrix A by a matrix of rank 2, which we obtain by adding only the first two terms in our outer product form of the SVD.

This procedure works in general. Indeed, if we want to approximate a rank r matrix A by a matrix with smaller rank (say rank $k < r$), then we add up the first k terms of the outer product form of the SVD of A, and omit the remaining terms with smaller coefficients.

Why might we want to find a smaller rank representation of our matrix? One reason may be if our matrix contains important data, and is particularly large. We may want to compress the data, while still retaining most of the important trends and features of the data. Indeed, one way of storing the data in the matrix

$$A = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^T + \cdots + \sigma_r \mathbf{u}_r \mathbf{v}_r^T$$

in a computer's memory would be to store the list of singular values $\sigma_1, \ldots, \sigma_r$, as well as the left and right singular vectors $\mathbf{u}_1, \ldots, \mathbf{u}_r$ and $\mathbf{v}_1, \ldots, \mathbf{v}_r$. If we are content with keeping a lower rank representation of the data

$$A_k = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^T + \cdots + \sigma_k \mathbf{u}_k \mathbf{v}_k^T$$

where $k < r$, we can store the data in our computer using fewer singular values and singular vectors. Not only does this allow us to save on memory requirements when storing the data, it also allows for faster rates when transmitting data. In Section 4 we will illustrate this procedure

by compressing image data. Finding lower rank approximations of data via the singular value decomposition can also be useful when trying to analyze other large data sets by allowing us to eliminate noise in the data and isolate trends, and can even be used to create facial recognition software! We will discuss applications of the SVD to data science at the end of this lab.

3. Computing the singular value decomposition in NumPy

While we could write our own code for computing the SVD of a matrix, we will instead use the function `np.linalg.svd`. The function `np.linalg.svd` accepts as input an $n \times m$ matrix A, formatted as a NumPy array `A`, and it returns three NumPy arrays, which we refer to as `u`, `s`, `v_t` respectively.

The first array `u` corresponds to the matrix U in a singular value decomposition $A = U\Sigma V^T$, while the third array `v_t` corresponds to V^T (notice that it corresponds *not* to the matrix V, but rather the *transpose* of V). The second array that is returned by `np.linalg.svd(A)` is *not* the matrix Σ, but rather a 1-dimensional NumPy array which contains the singular values of `A`. In order to reconstruct the matrix `A` from the output of the function `np.linalg.svd(A)`, we will need to first build the matrix Σ from the list `s` of singular values.

Problem 1. Define a function `sigma(m,n,s)` which accepts as input two positive integers `m` and `n`, along with a 1-dimensional NumPy array `s`, and returns an `m` $\times$ `n` matrix which is all zeros, except for the values in s which are placed along the diagonal as in equation (1). You may assume that the length of `s` is less than or equal to both `m` and `n`.

Hint: The function `np.zeros((a,b))` *might come in handy. Try it for a couple of different values of* `a` *and* `b` *to see if you can figure out how it works.*

For example, the function call `sigma(5,7,np.array([5,4,3]))` should return the array

```
array([[5., 0., 0., 0., 0., 0., 0.],
       [0., 4., 0., 0., 0., 0., 0.],
       [0., 0., 3., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0.]])
```

We are now ready to call the function `np.linalg.svd` and verify that it behaves as expected. As a small example, we will apply it to the matrix

$$M = \begin{bmatrix} 10 & 5 & -20 & 13 & -3 \\ 1 & 1 & -4 & 19 & 0 \\ 3 & 16 & -2 & 1 & 14 \end{bmatrix}.$$

To compute the SVD of M we do the following:

```
1 M=np.array([[10,5,-20,13,-3],[1,1,-4,19,0],[3,16,-2,1,14]])
2
3 u,s,v_t=np.linalg.svd(M)
```

The values of the matrices U and V^T are now stored in the arrays `u` and `v_t` respectively, while the list of singular values for M is saved as the 1-dimensional NumPy array `s`.

- Copy the code above to compute the singular value decomposition of the matrix M in your practice notebook, and examine the arrays `u` and `v_t`. Are they the sizes that you anticipated?
- Verify that this indeed gives an SVD for M (i.e., compute the product $U\Sigma V^T$ and verify that it equals the matrix M). *Hint: Use the function* `sigma` *that you defined above to construct the matrix* Σ.

Instead of repeating the procedure above every time we want to check an SVD, we will define a function that will make our lives easier going forward.

Problem 2. Define a function `reconstructed_array(u,s,v_t)` which takes as input arrays `u`, `s`, and `v_t`, where `u` and `v_t` are 2-dimensional NumPy arrays, and `s` is a 1-dimensional NumPy array. The output of the function `reconstructed_array(u,s,v_t)` should be a NumPy array representing the matrix product $U\Sigma V^T$, where Σ is the matrix constructed from the list `s` of singular values as above, and U and V^T are the matrices represented by `u` and `v_t` respectively.

You can test your function with many different matrices, of all different sizes. Just define your favorite NumPy array `A`, and then compute an SVD of `A` by

```
u,s,v_t=np.linalg.svd(A)
```

Then the function call `reconstructed_array(u,s,v_t)` should return the array `A`. Try this for several different arrays of different shapes to confirm that your function is working correctly.

Recall now that if A is a matrix with rank r, then the outer product form for an SVD of A is given by

$$A = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^T + \cdots + \sigma_r \mathbf{u}_r \mathbf{v}_r^T,$$

and that to find a rank k approximation to A we throw away all but the first k terms in the above sum

$$A_k = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^T + \cdots + \sigma_k \mathbf{u}_k \mathbf{v}_k^T.$$

How does finding a lower rank approximation look in terms of the matrix product $A = U\Sigma V^T$? If

$$A = U\Sigma V^T = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_n \end{bmatrix} \left[\begin{array}{cccc:ccc} \sigma_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_r & 0 & \cdots & 0 \\ \hdashline 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \end{array}\right] \begin{bmatrix} \mathbf{v}_1^T \\ \mathbf{v}_2^T \\ \vdots \\ \mathbf{v}_m^T \end{bmatrix}$$

then a rank k approximation for A (where $k \leq r$) is given by the product

$$A_k = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_k \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_k \end{bmatrix} \begin{bmatrix} \mathbf{v}_1^T \\ \mathbf{v}_2^T \\ \vdots \\ \mathbf{v}_k^T \end{bmatrix}.$$

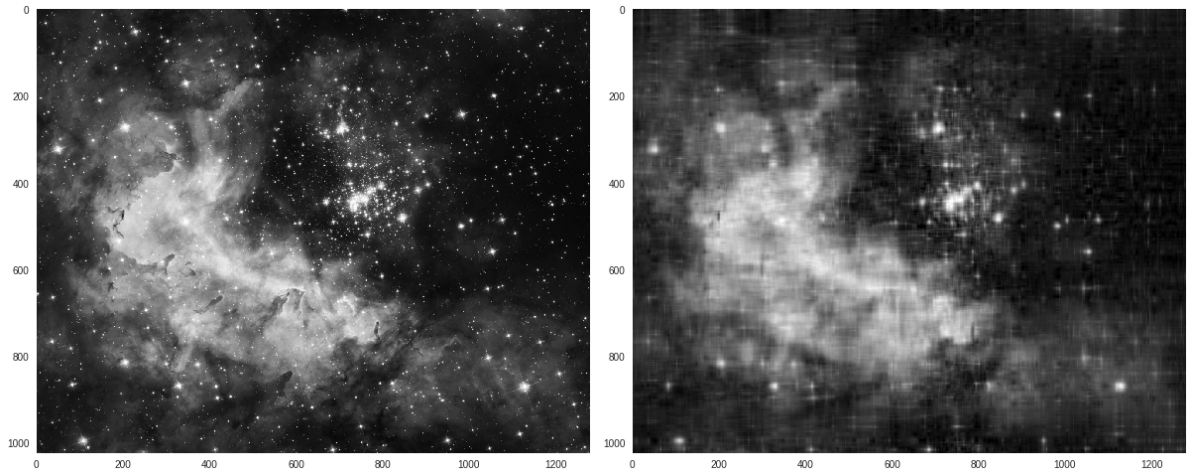
In other words we keep only the first k columns of U and rows of V^T, and we shrink the matrix Σ down to a $k \times k$ diagonal matrix.

Problem 3. Define a function `lower_rank(A,k)` which accepts as input a matrix `A`, which is formatted as a NumPy array, and an integer `k`, and returns a rank `k` approximation to the matrix `A`, using a singular value decomposition of `A` as described above. You may assume that `k` is less than or equal to the number of rows of `A` as well as the number of columns of `A`.

Hint: You may find taking slices of arrays, which we discussed in Lab 2, to be helpful here when you are dropping rows and columns from your matrices. Be careful though when slicing your arrays to make sure that you really are keeping the correct number of columns from U and rows from V^T. You can check that the output of your function has the correct rank using the function `np.linalg.matrix_rank`.

For example, if `M=np.array([[10,5,-20,13,-3],[1,1,-4,19,0],[3,16,-2,1,14]])`, then the output of `lower_rank(M,2)` should be

```
array([[ 7.62918243,  4.75650545, -15.97002054, 17.81802609,
        -1.41800606],
       [ 4.62356047,  1.37215737, -10.15942552, 11.63612305,
        -2.41792146],
       [ 3.39305278, 16.04036844, -2.66812169, 0.20122974,
        13.7377246 ]])
```



(A) The original rank 1024 image. (B) A rank 25 approximation to the image.

FIGURE 1. A Hubble Space Telescope image of the cluster Westerlund 2.

In the next section we will use the function `lower_rank(A,k)` which you just defined to find lower rank approximations to images that are stored as arrays.

4. Images as NumPy arrays

Matrices can be used to represent images in a fairly straightforward way. Suppose, for example, that we have a color image that is 1000 pixels tall and 500 pixels wide. Suppose that we can describe any possible pixel color by using a single number (i.e. red $= 1$, blue $= 2$, etc.). Then we can represent the image as a 1000×500 matrix, where the value of each entry in the matrix describes the color of the corresponding pixel. For example, the entry in the first row and first column of the matrix tells us the color of the top left pixel, the entry in the first row and second column tells us the color of the next pixel over, and so on. Such images are called bitmap (or raster) images, which are different than the vector graphics we discussed in Lab 8.

In practice the situation is a bit more complicated (at least for color images). Color images are typically represented by 3 separate matrices, one to describe how each of the colors red, green, and blue are distributed. Each matrix is called a *color channel*, and when they are combined these three colors mix to produce the many colors you may see in a digital photograph.

On the other hand, *grayscale* images (commonly called "black and white" images) can be represented by a single matrix. In our examples each entry in the matrix will be a number between 0 and 1, where 0 represents a black pixel, 1 represents a white pixel, and values in between give various shades of gray.

- Using the code provided in the notebook, import the image "image.png" and create two arrays. The first array `RGB_array` should represent the color image, while the other array `gray_array` should represent the grayscale image.
- Using the functions `show_color` and `show_gray` provided in the notebook, display the color image and the grayscale image.

While there are ways to apply the singular value decomposition to color images, for simplicity in this lab we will focus on working with the grayscale image contained in the `gray_array` matrix.

Problem 4. Determine the size of the matrix representation `gray_array` of the grayscale image. In other words, how many entries are there in the matrix `gray_array`? Save your answer as the variable `original_size`.

- Compute the rank of the matrix `gray_array`, and compare its value to the shape of the array `gray_array` (i.e., the number of rows and columns).

Notice that as the matrix `gray_array` has full rank, it implies that none of the rows of `gray_array` are linear combinations of the other rows. This makes sense, as we would expect there to be some minor variation in the coloring even between pixels that are right next to each other. On the other hand, throughout much of the image (away from the edges of objects, for example, or in places with large blocks of solid color), we might expect that this variation in the coloring of nearby pixels is so minor that our eyes can't even detect it. This suggests that we might be able to find a lower rank approximation to the image matrix, which still looks as sharp to our eyes as the original image. We will use the SVD to find such a lower rank approximation to `gray_array`.

- Compute an SVD for the array `gray_array`. Save the resulting output as `u_orig`, `s_orig`, and `v_t_orig` respectively.
- Plug the arrays `u_orig`, `s_orig`, and `v_t_orig` into the function `reconstructed_array`, and then display the resulting image using the function `show_gray` (this can be used as a further check to make sure that your functions are all working as intended).

To determine what rank we may want to use to approximate our image matrix, we will plot our singular values.

Problem 5. Plot the singular values of the array `gray_array`.

Hint: Recall that if `arr = [s_1,...,s_r]` *is a 1-dimensional array, then* `plt.plot(arr)` *will plot each of the values* `s_1`,...,`s_r` *as y-coordinates, with x-coordinates* `0`,`1`,...,`r-1`.

Based on the plot you created in Problem 5, what do you observe? You should see that although some of the initial singular values are very large (for example, $\sigma_1 = 553.38$), the singular values quickly taper down in size. The 100th singular value is $\sigma_{100} = 7.02$, while the 200th singular value is $\sigma_{200} = 3.00$, and from σ_{369} onwards, all of the remaining singular values are less than 1.

Problem 6. Using the functions `lower_rank` and `show_gray`, compute and display rank k approximations to our image, for various k values less than or equal to 1428 (which is the original rank of `gray_array`). What does the image look like when k takes the values $1, 10, 25, 50, 75, 100, 200, 300, 400$?

Find the smallest value of k which we can choose so that the resulting image looks as clear as the original image (i.e. without being noticeably pixelated or of lower quality). Save this value as the variable `min_rank`. *(There is no one correct answer to this question, and the auto-grader will accept answers within a range of values. We just want to make sure that you've tried plotting some different values to compare.)*

Since our whole goal is to compress the image (i.e. reduce the amount of data we need to store on our computer while still being able to recreate the image), we should think about whether we have actually been successful in doing this. For concreteness, let's consider the rank 100 approximation A_{100} of the array `gray_array` (here we will use A to denote the matrix represented by `gray_array`). Since A_{100} and A are matrices of the same size they have the same number of entries, and hence if we store the individual entries in A_{100} we won't actually save any space.

Recall however, that the rank 100 representation A_{100} of A can be reconstructed by knowing the first 100 columns of U and first 100 rows of V^T, along with the first 100 singular values of A. It turns out that by storing just these values (the first 100 columns of U, rows of V^T and singular values) we are able to compress the data significantly.

Problem 7. Compute the number of values needed to store the rank 100 approximation of A. Save your answer as the variable `rank_100_size`. Then compute the relative size of our compressed image, by computing `rank_100_size/original_size`, and storing the result as the variable `relative_size`.

The value of `rank_100_size` should count the combined number of entries in the first 100 columns of U, the first 100 rows of V^T, plus 100 (which is number of singular values needed for the rank 100 approximation). You don't need to store the entire matrix Σ, since all that is needed to reconstruct it is the singular values.

5. SVD and data science

Finally, we close out this lab with a few comments on how the SVD can be applied in data science. Often times data scientists are presented with large *noisy* data sets. Noise refers to error in the data, often times stemming from faulty or imprecise measurements, errors introduced when recording or transmitting the data, or even from rounding and natural limits to machine precision. Noise in a dataset makes it more difficult to both interpret the data and to draw conclusions from it. An important task in data science, therefore, is separating the important information contained in a dataset from the noise.

One way of doing this involves using the singular value decomposition, similar to what we did above. Indeed, by computing the outer product form SVD of a large data set that is expressed as a matrix A, we can represent our data as the sum of a collection of rank 1 matrices, each of which is scaled by a singular value σ_j of A. Terms which correspond to very small singular values often correspond to noise or error in the data, and by discarding these terms from the sum we are able to eliminate noise. By eliminating only the terms corresponding to small singular values we are able to preserve the broader trends and information contained in the data (which correspond to larger singular values).