EECS 16B Designing Information Devices and Systems II
Fall 2017 Miki Lustig and Michel Maharbiz **Homework 9**

This homework is due November 7, 2017, at noon.

1. The Moore-Penrose pseudoinverse for "fat" matrices

Say we have a set of linear equations described as $A\vec{x} = \vec{y}$. If A is invertible, we know that the solution is $\vec{x} = A^{-1}\vec{y}$. However, what if A is not a square matrix? In 16A, you saw how this problem could be approached for tall matrices A where it really wasn't possible to find a solution that exactly matches all the measurements. The Linear Least-Squares solution gives us a reasonable answer that asks for the "best" match in terms of reducing the norm of the error vector.

This problem deals with the other case — when the matrix A is short and fat. In this case, there are generally going to be lots of possible solutions — so which should we choose? Why? We will walk you through the **Moore-Penrose Pseudoinverse** that generalizes the idea of the matrix inverse and is derived from the singular value decomposition.

(a) Say you have the following matrix.

$$A = \begin{bmatrix} 1 & -1 & 1 \\ 1 & 1 & -1 \end{bmatrix}$$

Calculate the SVD decomposition of A. That is to say, calculate U, Σ, V such that,

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

What are the dimensions of U, Σ and V?
Note. Do NOT use a computer to calculate the SVD. You may be asked to solve similar questions on your own in the exam.

(b) Let us think about what the SVD does. Let us look at matrix A acting on some vector $\vec{x}$ to give the result $\vec{y}$. We have,

$$A\vec{x} = U\Sigma V^T\vec{x} = \vec{y}$$

Observe that $V^T\vec{x}$ rotates the vector, Σ scales it and U rotates it again. We will try to "reverse" these operations one at a time and then put them together.
If U "rotates" the vector $\left(\Sigma V^T\right)\vec{x}$, what operator can we derive that will undo the rotation?

(c) Derive a matrix that will "unscale", or undo the effect of Σ where it is possible to undo. Recall that Σ has the same dimensions as A. Ignore any division by zeros (that is to say, let it stay zero).

(d) Derive an operator that would "unrotate" by V^T.

(e) Try to use this idea of "unrotating" and "unscaling" to derive an "inverse" (which we will use $A^\dagger$ to denote). That is to say,

$$\vec{x} = A^\dagger\vec{y}$$

The reason why the word inverse is in quotes (or why this is called a pseudo-inverse) is because we're ignoring the "divisions" by zero.

(f) Use $A^\dagger$ to solve for $\vec{x}$ in the following systems of equations.

$$\begin{bmatrix} 1 & -1 & 1 \\ 1 & 1 & -1 \end{bmatrix} \vec{x} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

(g) (Optional) Now we will see why this matrix is a useful proxy for the matrix inverse in such circumstances. Show that the solution given by the Moore-Penrose Psuedoinverse satisfies the minimality property that if $\vec{\hat{x}}$ is the psuedo-inverse solution to $A\vec{x} = \vec{y}$, then $\|\vec{\hat{x}}\| \leq \|\vec{z}\|$ for all other vectors $\vec{z}$ satisfying $A\vec{z} = \vec{y}$.

(Hint: look at the vectors involved in the V basis. Think about the relevant nullspace and how it is connected to all this.)

This minimality property is useful in both control applications (as you will see in the next problem) and in communications applications.

2. Eigenfaces

In this problem, we will be be exploring the use of PCA to compress and visualize pictures of human faces. We use the images from the data set Labeled Faces in the Wild. Specifically, we use a set of 13,232 images aligned using deep funneling to ensure that the faces are centered in each photo. Each image is a 100x100 image with the face aligned in the center. To turn the image into a vector, we stack each column of pixels in the image on top of each other, and we normalize each pixel value to be between 0 and 1. Thus, a single image of a face is represented by a 10,000 dimensional vector. A vector this size is a bit challenging to work with directly. We combine the vectors from each image into a single matrix so that we can run PCA. For this problem, we will provide you with the first 1000 principal components, but you can explore how well the images are compressed with fewer components. Please refer to the IPython notebook to answer the following questions.

(a) We provide you with a randomly selected subset of 1000 faces from the training set, the first 1000 principle components, all 13,232 singular values, and the average of all of the faces. What do we need the average of the faces for?

(b) We provide you with a set of faces from the training set and compress them using the first 100 principal components. You can adjust the number of principal components used to do the compression between 1 and 1000. What changes do you see in the compressed images when you used a small number of components and what changes do you see when you use a large number?

(c) You can visualize each principal component to see what each dimension "adds" to the high-dimensional image. What visual differences do you see in the first few components compared to the last few components?

(d) By using PCA on the face images, we obtain orthogonal vectors that point in directions of high variance in the original images. We can use these vectors to transform the data into a lower dimensional space and plot the data points. In the notebook, we provide you with code to plot a subset of 1000 images using the first two principal comonents. Try plotting other components of the data, and see how the shape of the points change. What difference do you see in the plot when you use the first two principal components compared with the last two principal components? What do you think is the cause of this difference?

(e) We can use the principal components to generate new faces randomly. We accomplish this by picking a random point in the low-dimensional space and then multiplying it by the matrix of principal components. In the notebook, we provide you with code to generate faces using the first 1000 principal components. You can adjust the number of components used. How does this affect the resulting images?

3. Image Processing by Clustering In this homework problem, you will learn how to use the k-means algorithm to solve two image processing problems: (1) color quantization and (2) image segmentation.

Digital images are composed of pixels (you could think of pixels as small points shown on your screen.) Each pixel is a data point (sample) of an original image. The intensity of each pixel is its feature. If we use 8-bit integer Var_i to present the intensity of one pixel, $(Var_i = 0)$ means black, while $(Var_i = 255)$ means white. Images expressed only by pixel intensities are called *grayscale* images.

In color image systems, the color of a pixel is typically represented by three component intensities (features) such as Red, Green, and Blue. The features of one pixel are a list of three 8-bit integers:$[Var_r, Var_g, Var_b]$. Here $[0,0,0]$ means black, while $[255,255,255]$ presents white. You can find tools like this website to visualize RGB colors:
https://www.colorspire.com/rgb-color-wheel/.

Now think about your own experience: Have you needed to delete some photos on your cell phones because you ran out of the memory? Have you felt "why does it take forever to upload my photos"? We need ways to reduce the memory size of each photo by image compression.

In computer graphics, there are two types of image compression techniques: lossless and lossy image compression. The former technique is preferred for archival purposes, which aim to perfectly record an image but reduce the required memory to store it. The latter techniqe tries to remove some irrelevant and redundant features without destroying the main features of the image. In this problem, you will learn how to use the k-means algorithm to perform lossy image compression by color quantization.

Color quantization is one way to reduce the memory size of an image. In real schemes, it can be used in combination with other techniques, but here we apply it directly to pixels on its own for simplicity. The target of color quantization is to to reduce the number of colors used in an image, while trying to make the new image visually similar to the original image. Graphics Interchange Format (GIF) is one format that uses color quantization.

For example, consider an image of the size 800×600, where each pixel takes 24 bits (3 bytes - one each for red, green, and blue) to store its color intensities. This raw image takes $800 \times 600 \times 3$ bytes, about 1.4MB to store. If we use only 8 colors to represent all pixels in this image, we could include a color map, which stores the RBG values for these representation colors, and then use 3 bits for each pixel to indicate which one is its representation color. In that case, the compressed image will take $8 \times 24 + 800 \times 600 \times 3$ bits, about 0.2MB, to store it. This compression has saved a considerable amount of memory.

There are two main tasks in color quantization: (1) decide the representation colors, and (2) determine which representation color each pixel should be assigned. Both tasks can be done by the k-means algorithm: It groups data points (pixels) into k different clusters (representation colors), and the centroids of the clusters will be the representation colors for those pixels inside the cluster.

Here is another thing we can do with this strategy: image segmentation. Image segmentation partitions a digital image into multiple segments (sets of pixels). It is typically used to locate boundaries of objects in images. Once we isolate objects from images, we can perform object detection and recognization, which play essential roles in artificial intelligence. This can be done by clustering pixels with similar features and labeling them with an indicating color for each cluster (object).

(a) Please look at the ipython notebook file, where you will find a 4 by 4 grayscale image. Perform the k-means algorithm on the 16 data points with $k = 4$. What are the representation colors (centroids)? Show the image after color quantization.

(b) See the ipython notebook. Apply the k-means algorithm to the grayscale image with different values of k. Observe the distortion. Choose a good value for k, which should be the minimum value for keeping

the compressed image visually similar to the original image. Calculate the memory we need for the compressed image.

(c) See the ipython notebook. Apply the k-means algorithm to the color image with different values of k. Observe the distortion. Choose a good value for k, calculate the memory we need for the compressed image.

(d) See the ipython notebook. Here we combine three features of each pixel into one feature as the intensity in grayscale images. Use the k-means algorithm to locate the boundaries of objects. How many objects do you expect from this image? Adjust the value of k and describe your observations. If you are interested in this, you could learn more algorithms for image segmentation from EECS courses in image processing and computer vision.

4. Brain-machine interface

The iPython notebook pca_brain_machine_interface.ipynb will guide you through the process of analyzing brain machine interface data using principle component analysis (PCA). This will help you to prepare for the project, where you will need to use PCA as part of a classifier that will allow you to use voice or music inputs to control your car.

Please complete the notebook by following the instructions given.

Contributors:

- Siddharth Iyer.
- Justin Yim.
- Stephen Bailey.
- Yu-Yun Dai.