Due: Oct 28, 2020 **Homework 4** CPSC 475/BENG 475

For this assignment, parts 1, 2, 3, 4, and 5 are worth 20%, 20%, 25%, 30%, and 5%, respectively (as usual, only the starred questions will be graded). Parts 5b,c are optional but recommended if you are interested in implementing a more efficient version of PCA that uses SVD.

Clustering images in image space

1 Load the file `tinypics.mat`[1], which contains a 20×3 matrix where each row is a tiny horizontal image consisting of 3 pixels only.

– a You can visualize all the images at once calling `imagesc/plt.imshow` on the data matrix using a grayscale colormap, followed by `axis equal tight/plt.axis('image')`. Each row is a 1-by-3 "image". Can you see any patterns? In particular, compare the first 9 images with the last 11.

– b Make a 3-D scatter plot (`scatter3` in MATLAB, `plt.scatter` using a 3-D axis in Python) of this dataset by representing each image as a vector in $\mathbb{R}^3$, that is, each pixel will contain the value of one of the three coordinates. Label your axes as "pixel1," "pixel2," and "pixel3." Now, using the Rotate3D tool in the figure window that is opened when you plot it (in Python, run this outside of a Notebook for best results), move it around to see if you can *visually* separate the points into isolated clusters. How many clusters would you say are there? Are there any points that seem harder to assign to a specific cluster?

– c* Flesh out the code in `kmeans.m/py` to implement the k-means algorithm, which takes as inputs an $n \times p$ data matrix, `X` (where n is the number of points in your dataset and p is the dimensionality of each data point), as well as a number of clusters, `k`, and outputs a vector `clusters` assigning a cluster index to each data point and a `centroids` matrix. See the docstring included in the `.m/py` file for more information. Your job will be to implement

[1] in MATLAB, execute `load tinypics`; in Python, import `loadmat` from `scipy.io` and do `loadmat('tinypics.mat')['data']`

helper functions at the bottom of the `.m/py` file that are called from inside of `kmeans`.

– d Run your implementation of k-means on the `tinypics` dataset, using `k` = 2 and random initialization. Then create a new 3-D scatter plot, this time using the cluster assignments returned by your function to color each data point. In the same plot, also include the cluster centroids using a different color or marker. Are these the clusters you expected based on your visual inspection of the images in part a? Repeat this process by re-running k-means a few times—do you observe any changes? Why?

2 Now you will work with slightly more complex images. Load the dataset `logos.mat`[2], which is a 3-D array containing ten 48×100 images (each image is a slice). You can access each one of them by indexing the third dimension, e.g., `data(:,:,1)` returns the first image, `data(:,:,2)` the second, and so on.

– a* Use subplots to display the ten images simultaneously in a 2-by-5 grid, plotting each one of them with `imagesc/plt.imshow` using a grayscale colormap. Can you separate these images into two classes, using your own brain's ability for pattern recognition?

Suppose we have a set of data points $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ in $\mathbb{R}^p$—in other words, each $\mathbf{x}_i$ is a p-element vector, and each element of the vector is a measurement of some dimension or variable in the data. For instance, an 11×11 image patch can be considered as a 121 element vector, where each element corresponds to the value of a pixel in the patch. A year's worth of closing stock prices for some company can be thought of as a 365 element vector where each element corresponds to the closing price of the stock on a given day. Let $X = \left(\mathbf{x}_1^\mathsf{T}; \ldots; \mathbf{x}_n^\mathsf{T}\right)$, so X is a matrix with the $\mathbf{x}_i$s as **rows**. We'll call X our *data matrix* and the $\mathbf{x}_i$ *data points* (here, in contrast with the lecture notes, we will follow the common practice of defining data matrices with the data points as rows instead of columns, even though vectors are traditionally represented as columns in linear algebra!).

[2] in Python, load it with `data = scipy.io.loadmat('logos.mat')['data']`.

b* Create a new 10×4800 data matrix `X`, where the i^{th} row is a *vectorized* version (1×4800) of the i^{th} image in the `data` array (to vectorize each image, use `img(:)` in MATLAB or `img.flatten()` in Python). What is the dimensionality of each data point (each row in your matrix)? Is it possible to make a scatter plot analogous to the one from 1-b, where each image is visualized as a point having its pixel values as coordinates?

c* Now run your `kmeans` function to partition the data into 2 clusters. Is the resulting assignment what you expected based on what the images look like? Also display (with `imagesc`/`plt.imshow`!) the resulting centroids side by side (use `reshape` to convert each row vector into the original image dimensions, 48×100). **Repeat this process by re-running k-means a few times** and comment on the results.

Dimensionality reduction

In this part we'll explore PCA (principal components analysis), a technique for finding low-dimensional representations of high-dimensional data.

When p (the dimension of our data vectors) is large, it can be difficult to analyze the data and understand how it might be structured. Intuitively, we'd like to represent our data in a low dimensional space where analysis is easier without distorting it too much or throwing away any important information it contains. How might we do this?

3 a* In Lectures 8-9 you learned that, according to the statistical view of PCA, given the covariance matrix of a dataset, the eigenvector with largest eigenvalue will point in the direction of greatest variance. Using this view, implement PCA. It should be a function with the signature

```
[components, variances] = pca(X, m)
```

The `X` variable will be a data matrix as described above, and `m` will be a parameter indicating how many principal components/associated variances to return. The `components` output should be a matrix containing the principal components of `X` as its columns, and

the `variances` output should be a vector, not a diagonal matrix. Here is a list of the steps involved:

1. Center `X`, i.e., subtract the mean data point from each data point (in other words, subtract, from each row, the mean row of your data matrix). You can use the handy `mean` function to compute the mean along the 1st axis (0th in Python!).
2. Construct the covariance matrix $C = \frac{1}{(n-1)} X^\mathsf{T} X$.
3. Compute its eigenvectors and eigenvalues, i.e., the principal components and principal values of the original matrix `X`. The `eigs/scipy.sparse.linalg.eigs` function allows you to specify how many eigenvectors/eigenvalues you want.
4. Sort the returned e-values (and their corresponding e-vectors) in decreasing order, if not already so. These are the variances of each corresponding principal component. MATLAB only: Convert the matrix of principal values returned in step 3 into a vector. To do this, extract the diagonal with `diag`.

b* To check whether your implementation is working correctly, load the `gaussian.mat` file[3], which contains a data matrix called `gaussian` containing 1000 normally distributed points in two dimensions. You can plot the points with `scatter`, and your two principal components with `quiver`. You should scale the principal components by their corresponding variances to get vectors you can see easily, and also set `axis equal`. If everything is working, the data points will look like a fuzzy ellipse, and the first principal component should lie along the major axis of the ellipse (although its sign is arbitrary), with the second component orthogonal to the first. *Hint*: when using `quiver`, you can position the vectors at (0,0). The third argument to this function should be an array containing the values of the first coordinates of each principal component (scaled by their variances), and the fourth argument another array with the values of the second coordinates. This results in the following call: `quiver([0,0],[0,0],[`λ_1`PC1`$_x$`,`λ_2`PC2`$_x$`],[`λ_1`PC1`$_y$`,`λ_2`PC2`$_y$`])`, where `PC` is a principal component and λ is a principal value (remember: `PC1` and `PC2` are the first and second *columns* of `components`, respectively).

[3] in Python, load it with `scipy.io.loadmat('gaussian.mat')['gaussian']`.

Important: make sure you get the expected result here; that's how you will know you have a correct implementation of PCA to be used in parts 4 and 5.

— c What is an eigenvector? What is the relationship between an eigenvector and its corresponding eigenvalue?

— d Qualitatively, what does the covariance between two vectors measure? What is a covariance matrix? Why is it important that each component of the vectors in the data matrix have zero mean?

— e We can think of the set of principal components as a different coordinate system for the data. How does each principal component relate to the original data vectors?

4 a* Apply your `pca` function to the `logos` dataset. Using subplots, display the first four principal components in a 2-by-2 grid by `reshape`ing them into the appropriate dimensions (remember the image size is 48×100). In a separate figure, plot the variances of the first 4 principal components. What do you observe about the principal components and their corresponding variances? How many dimensions do you think should suffice to represent this dataset? Is this result what you expected?

— b We know the first principal component points in the direction of largest variance. How can the image you plotted in a be seen as a direction in image space?

– c* Project the data in `X` onto the first principal component obtained above. What is the dimensionality of your data now? How does the `components` matrix relate to the concept of basis, or coordinate system?

– d* Run `kmeans` on this reduced data using $k = 2$ and create a 2-D scatter plot using

the assigned clusters as the colors of the points. (Since you have a single component, use a vector of `0`s as the second argument (the y-values) of `scatter`.) Is the resulting assignment what you expected based on what the images look like? Compute the "centroid images" by taking the average between the images in each cluster (**use the full-dimension images in your data matrix for that**). Finally, display these two centroids side by side with `imagesc/plt.imshow` using a grayscale colormap. Repeat this process by running k-means a few more times and comment on the results.

e* Compare your results using dimensionality reduction with the ones you got in part 2, when you used the full data. Remember that k-means uses the Euclidean distance between two points to judge how similar they are (and, consequently, whether they should belong in the same cluster). Why did the same approach work better in one case, and worse in the other?

Thinking of your data points as vectors, which bases, or coordinate systems, did you use in each case? Which coordinate system was more appropriate for separating the images into "reasonable" clusters?

5 a* Apply your `pca` function to the dataset contained in the `faces.mat` file.[4] The size of the data matrix is 1000×9216, where each row represents the image of a face (96×96 pixels) under various poses and lighting conditions. This time you will need to do a little preprocessing: the data matrix is in the format `int32`, so before using it you need to convert it into double-precision floating-point format (use the function `double`) and properly normalize the range of values of each image (make the maximum value of each row be equal to 1). Explore the dataset a little bit by displaying some of its data points (you might need to transpose the images before plotting them). Finally, run PCA on the entire data matrix and show the first 9 principal components, or "eigenfaces" . Use subplots to display them in a 3-by-3 grid with `imagesc/plt.imshow` using a grayscale colormap. What do the eigenfaces tell you? What do you think are relevant features when comparing face images?

[4] in Python, load it with `scipy.io.loadmat('faces.mat')['Data']`.

— b Let $X = USV^\intercal$ be the singular-value decomposition of the data matrix `X`. Compute it in Matlab/Python for the faces dataset (de-mean the data first!) doing `[U,S,V] = svd(X)`/`[U,s,Vt] = np.linalg.svd(X)`, (in MATLAB, convert the diagonal matrix `S` into a vector `s` using `diag`). Then create a vector `v` composed of the squares of the singular values in `s` divided by (n-1), i.e. $v = \frac{s^2}{n-1}$. Now compare the `components` and `variances` you obtained in a with the first 9 columns in `V` and `v`, respectively. What do you notice? Explain why or prove that they are the same (except for possibly a change in sign).

— c Read the documentation for the `svd/np.linalg.svd` function signature that includes the argument `'econ'/full_matrices=False`. Explain how that option could be useful when computing PCA. Finally, re-implement PCA (in a function `pcasvd`) using SVD with this economical decomposition, and compare its *average* running time (over 10 runs on the faces dataset) with that of your implementation from part 3 (use Matlab's `tic` and `toc`, or Python's `time.time()` for that).