

## Clustering images in image space

**1** Load the file `tinypics.mat` (in MATLAB, move into your homework directory and execute `load tinypics`), which contains a  $20 \times 3$  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` on the data matrix, followed by `axis equal tight`. Can you see any patterns? In particular, compare the first 9 images with the last 11.

— **b** Make a 3-D scatter plot (`scatter3`) 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, move it around to see if you can visually separate the points into isolated clusters. How many clusters would you say are there?

— **c** Flesh out the code in `mykmeans.m` 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` file for more information. Your job will be to implement helper functions at the bottom of the `.m` file that are called from inside of `mykmeans`.

— **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?

MATLAB hint: to easily select the points belonging to one particular cluster, you can use logical indexing, e.g., `data(clusters == 1,:)` for cluster 1. For specifying colors, you will need to use the signature `scatter3(x,y,z,s,c)`, where `s` and `c` stand for the sizes and colors of your points (run `doc scatter3` for more details). Don't forget to use `hold on` before plotting the second cluster.

2 Now you will work with slightly more complex images. Load the dataset `logos.mat`, which is a 3-D array containing ten  $48 \times 100$  images. 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 `subplot` to display the ten images simultaneously in a 2-by-5 grid, plotting each one of them with `imshow` (maximize the figure window for better visualization). 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  $\{\mathbf{x}_1, \dots, \mathbf{x}_n\}$  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  $11 \times 11$  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  $X = (\mathbf{x}_1^T; \dots; \mathbf{x}_n^T)$ , so  $X$  is a matrix with the  $\mathbf{x}_i$  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!).

— b Create a new  $10 \times 4800$  data matrix  $X$ , where the  $i^{\text{th}}$  row is a vectorized version of the  $i^{\text{th}}$  image in the `data` array. What is the dimensionality of each data point (each row in your matrix)? Can you 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 `mykmeans.m`<sup>1</sup> to partition the data into 2 clusters. Is the resulting assignment what you expected based on what the images look like? Also plot the resulting centroids side by side (use `reshape` to convert each row vector into the original image dimensions,  $48 \times 100$ ). 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 Lecture 8 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 using the `mypca.m` file. This should be a MATLAB function with the signature

$$[\text{components} \text{ variances}] = \text{mypca}(\mathbf{X}, \mathbf{m})$$

The  $\mathbf{X}$  variable will be a data matrix as described above, and  $\mathbf{m}$  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  $\mathbf{X}$  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  $\mathbf{X}$ , i.e., subtract the mean data point from each data point (in other words, subtract from each row the mean row in your data matrix). Check MATLAB's handy `mean` function.

---

<sup>1</sup>If you're not confident that your implementation is correct, for this part and the next you can use an equivalent MATLAB implementation by running `kmeans(X,2,'Start','uniform')`.

2. Construct the covariance matrix  $C = \frac{1}{(n-1)} X^T X$ .
  3. Compute its eigenvectors and eigenvalues, i.e., the principal components and principal values of the original matrix  $X$ . MATLAB's `eigs` function allows you to specify how many eigenvectors/eigenvalues you want.
  4. Convert the matrix of principal values returned in 3 into a vector. To do this, extract the diagonal with `diag`. These are the variances of each corresponding principal component.
- **b** To check whether your implementation is working, load the `gaussian.mat` file, which contains a data matrix called `gaussian` containing normally distributed points in two dimensions. You can plot the points with `scatter`, and your two principal components with the `quiver` function (remember to `hold on`). 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.
- MATLAB 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.
- **c** Qualitatively, what does the covariance between two vectors measure? What information does the covariance matrix you computed contain? Why is it important that each component of the vectors in the data matrix have zero mean?
- **d** Let  $X = USV^T$  be the singular-value decomposition of the data matrix  $X$ . Compute it in Matlab for the gaussian dataset doing `[U,S,V] = svd(X)`, and 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 **b** with  $V$  and  $v$ , respectively. What do you notice? How can

you explain that?

**4 a** Apply `mypca.m` to the `logos` dataset. Using `subplot`, 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  $48 \times 100$ ). 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** 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?

— **c** Run k-means 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 0s as the second argument 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, plot these two centroids side by side with `imshow`. Repeat this process by running k-means a few more times and comment on the results.

— **d** 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?

**5 a** Apply `mypca.m` to the dataset contained in the `faces.mat` file. The size of the data matrix is  $1000 \times 9216$ , where each row represents the image of a face ( $96 \times 96$  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 will need to transpose the images before plotting them). Finally, run your PCA function on the entire data matrix and show the first 9 principal components, or “eigenfaces” . Use `subplot` to display them in a 3-by-3 grid with `imshow(..., [])`—notice the brackets this time, for rescaling. What do the eigenfaces tell you? What do you think are relevant features when comparing face images?

— **b** *Optional (5% extra credit)*: Read the Matlab documentation for the `svd` function signature that includes the argument ‘`econ`’. Explain how that could be useful when computing PCA. Re-implement PCA (in a new file named `pcasvd.m`) using SVD with economical decomposition, and compare its *average* running time with that of your previous implementation (use Matlab’s `tic` and `toc`). Comment on the results.