Statistics, Data Analysis, and Machine Learning for Physicists
Timothy Brandt
Spring 2020

# Lectures 14 and 15

In this lecture we'll shift gears to machine learning. This is a fuzzy, ill-defined term that encompasses a huge amount of material in statistical modeling and data analysis. I'll distinguish the topic from the model fitting we did earlier in the following way. Earlier in this course, we could use a physically motivated model plus an accurate understanding of noise to fit data using a likelihood. When you can model your data in this way, you usually should. For much of the rest of the course we will assume that you are not in this enviable position.

For this class and next, we'll talk about dimensionality reduction. This is a common task: you have some data set with a huge number of dimensions (an image or a spectrum, for example), and many "points" (i.e. images or spectra). For example, 10,000 spectra each with 3,000 wavelengths represents 10,000 points in a 3,000-dimensional parameter space. Images are worse.

The task of dimensionality reduction is to represent some of the important features in that data using less information. Unfortunately, we might not be able to write down physical models of these data. Think of a data set as being a matrix of, say, $n_{\rm pixels} \times n_{\rm images}$. A linear dimensionality reduction is a *low-rank approximation*. The data matrix will generally have a rank equal to the lesser of $n_{\rm images}$ and $n_{\rm pixels}$. This may be expensive to store on disk, or perhaps some of the information in these images is just noise and I would like to extract the most significant parts of the images (which I will identify as the signal). A low-rank approximation to a data matrix $\mathbf{D}$ may be written

$$\mathbf{D} \approx \mathbf{B} \cdot \mathbf{W}, \tag{1}$$

where $\mathbf{B}$ is a $n_{\rm pixels} \times k$ matrix consisting of the basis images, $\mathbf{W}$ is a $k \times n_{\rm images}$ weight/coefficient matrix, and $k$ is the rank of the approximation. We'll discuss three low-rank approximations:

- Principal component analysis (PCA)
- Weighted low-rank approximation
- Non-negative matrix factorization

The differences between these methods come in the definition of "$\approx$" and in the conditions, if any, on the matrices $\mathbf{B}$ and $\mathbf{W}$.

Principal component analysis (commonly just called PCA) is the first and most famous of the low-rank approximations. For PCA, "$\approx$" in Equation (1) means that the Frobenius norm of the residual is minimized, and there are no additional constraints placed on the matrices $\mathbf{B}$ and $\mathbf{W}$. This turns out to be closely related to the singular value decomposition (SVD).

PCA is a tool to explain the variation of my data set, which implies that I am not worried about its average value. For PCA, I don't want my templates to change if, for example, I add a constant offset to all of my spectra. I'll then start by subtracting the average data/image/etc. from each column of my data matrix.

My data matrix $\mathbf{D}$ becomes $\mathbf{D}'$, and has its rank reduced by 1. We can then write down the covariance matrix of the data,

$$\mathbf{C} = \frac{1}{n_{\rm images}} \mathbf{D}' \mathbf{D}'^T. \tag{2}$$

To see how this is the covariance matrix, let's look at an element:

$$C_{ij} = \frac{1}{n_{\text{images}}} \sum_k D'_{ki} D'_{kj} = \frac{1}{n_{\text{images}}} \sum_k \left(D_{ki} - \langle D_i \rangle\right) \left(D_{kj} - \langle D_j \rangle\right). \tag{3}$$

I subtracted the mean image from each individual image $\mathbf{D}$ to produce the images in $\mathbf{D'}$, which is why the two sums are equivalent. This is another way of seeing why subtracting the mean is important–it enables me to relate $\mathbf{D'}$ with the covariance matrix of the data. These sums are over the images $k$ in my data set. So, $C_{ij}$ is the sum of the product of the pixel residuals in the images, which is just the covariance matrix of the pixel values.

The total variance of my data set is the sum of the squared values of $\mathbf{D'}$,

$$\sigma^2 = \sum_i \sum_k D'^2_{ki} = n_{\text{images}} \sum_i C_{ii}. \tag{4}$$

In other words, the total variance (up to a normalization factor) is the trace of the covariance matrix. This fact will come up again later.

Now, let's take the SVD of the matrix $\mathbf{D'}$. Recall that this means we write down

$$\mathbf{D'} = \mathbf{U}\mathbf{W}\mathbf{V}^T, \tag{5}$$

where $\mathbf{U}$ and $\mathbf{V}$ are unitary matrices and $\mathbf{W}$ is diagonal. What about the SVD of the covariance matrix? Well, I can write down the SVD of $\mathbf{D'}$ as above, and then $\mathbf{D'}^T$ is given by

$$\mathbf{D'}^T = \mathbf{V}\mathbf{W}^T\mathbf{U}^T. \tag{6}$$

I multiply these together to get

$$\begin{aligned} \mathbf{C} &= \frac{1}{n_{\text{images}}} \left(\mathbf{U}\mathbf{W}\mathbf{V}^T\right) \left(\mathbf{V}\mathbf{W}^T\mathbf{U}^T\right) \\ &= \frac{1}{n_{\text{images}}} \mathbf{U}\mathbf{W}^2\mathbf{U}^T. \end{aligned} \tag{7}$$

This is the eigendecomposition of $\mathbf{C}$ (a special case of the SVD), and it contains basically the same information as the SVD of $\mathbf{D'}$. So, now we have the SVD of both $\mathbf{C}$ and $\mathbf{D'}$ and the eigendecomposition of $\mathbf{C}$, and there are a few facts we will use:

- The columns of $\mathbf{U}$ form an orthonormal basis for the range of $\mathbf{C}$ (and for $\mathbf{D'}$, since the same $\mathbf{U}$ also appears in the SVD of $\mathbf{D'}$).
- The columns of $\mathbf{U}$ are the (orthonormal) eigenvectors of $\mathbf{C}$ and the diagonal elements of $\mathbf{W}^2$ are the corresponding eigenvalues.
- The trace of $\mathbf{C}$, the sum of the variances in all parameters, is equal to the sum of the eigenvalues, and also (up to a constant factor) equal to the variance in $\mathbf{D'}$.

The covariance between the basis images defined by two different eigenvectors (columns of $\mathbf{U}$) is zero since $\mathbf{U}^T\mathbf{C}\mathbf{U}$ is diagonal–we have diagonalized the covariance matrix. The covariance of a single column of $\mathbf{U}$ is the corresponding eigenvalue (element of $\mathbf{W}^2$). If I want the basis image that accounts for the maximum possible variance of my image, I just take the column of $\mathbf{U}$ corresponding to the largest eigenvalue. If I want $k$ basis images that together account for as much variance as possible, I take the $k$ columns of $\mathbf{U}$ corresponding to the $k$ largest eigenvalues. The fraction of the covariance accounted for by any given basis image is just its eigenvalue (component of $\mathbf{W}^2$) divided by the sum of all the eigenvalues (trace of $\mathbf{W}^2$). This follows from the last item above.

The original problem was to construct a low-rank approximation of D. As we showed earlier, the best rank-one approximation is just the average of the images. What about the best rank-two approximation? Well, $\chi^2$ (equivalently, the trace of the covariance matrix, or the sum of the $W_{ii}^2$ returned by the SVD of $\mathbf{D}'$) is the variance of the residual of the approximation. So, the best rank-two approximation will come from using the mean image plus the basis image with the largest variance, removing this variance from the residual. My best rank-two approximation is the mean image plus the SVD of $\mathbf{D}'$, keeping only the first eigenvalue:

$$\mathbf{D}_{k=2} = \langle \mathbf{D} \rangle + U_1 W_{11} V_1^T \tag{8}$$

where the subscript 1 on $U$ and $V$ means to take the first column only of those matrices. To look at it another way, if I use this approximation of $\mathbf{D}$, the residual is the same as $\mathbf{D}'$ but with one eigenvalue in its SVD (the largest one) set to zero:

$$\mathbf{D}'' = \mathbf{U}\mathbf{W}\mathbf{V}^T \text{ with } W_{11} = 0. \tag{9}$$

The variance of the residual is reduced by

$$\Delta\sigma^2 = -W_{11}^2. \tag{10}$$

The fractional variance being reduced by

$$\frac{\Delta\sigma^2}{\sigma^2} = -\frac{W_{11}^2}{\sum_i W_{ii}^2}, \tag{11}$$

where $W_{ii}$ are the singular values of the SVD of $\mathbf{D}'$. The exact same argument carries over to the best rank-3, rank-4, and rank-$k$ approximations to $\mathbf{D}$. You just take the mean image/spectrum/whatever, and then take the first $k-1$ principal components. Another way of thinking of this is that your low-rank approximation is to model the images using a mean image $I_0$ plus $I_1$, ..., $I_{k-1}$ (the first $k-1$ columns of $\mathbf{U}$), where $k$ is the rank of the approximation. The low-rank approximation of an image $J$ is then

$$J \approx I_0 + \sum_{i=1}^{k-1} \alpha_i I_i, \tag{12}$$

a linear combination of these basis images. You can then solve for the $\alpha_i$ using least-squares fitting. If you don't apply any weights to your least-squares fit (i.e. a uniform $\sigma^2$ in $\chi^2$), then the coefficients $\alpha$ will also be given by the rows of $\mathbf{V}^T$ from the SVD as Equation (8) suggests.

**Important Note:** you can either multiply out the data matrix to get the covariance matrix and then take the SVD, or you can just take the SVD of the data matrix. **It is almost always better to take the SVD of the data matrix**. Multiplying out to get the covariance matrix wastes a lot of computational effort. Also, the SVD in python includes a keyword `full_matrices` that is `True` by default. Set this equal to `False` when you use the SVD for PCA. If the data set has dimensions $10^5$ pixels by $10^3$ images, for example, the full, square unitary matrix $\mathbf{U}$ will be $10^5 \times 10^5$ and may not fit in RAM. The matrix $\mathbf{U}$ with `full_matrices=False` will be $10^5 \times 10^3$: very big, but no longer enough to cause a memory error. Those extra columns of $\mathbf{U}$ are not in the range of the data matrix $\mathbf{D}$ anyway, so they are not needed for PCA.

Here, then, is how you would perform PCA on a mean-subtracted data matrix $\mathbf{D}$ to get an approximation of rank $k$:

```
U, W, VT = np.linalg.svd(D, full_matrices=False)
basis = U[:, :k]
lowrank = np.linalg.multi_dot([U[:, :k], np.diag(W[:k]), VT[:k]])
```

Now, suppose you have an array of images stored as a numpy ndarray with an arbitrary number of axes ($\geq$2). For example, suppose that you have 100 images, each $200 \times 200$ pixels, for an array of shape `(100, 200, 200)` called `images`, and that you have not done any mean subtraction yet. Now, you could implement PCA using the following code:

```
meanimage = np.mean(images, axis=0)
resid = images - meanimage
D = np.reshape(resid, (images.shape[0], -1)).T        # -> n_pix x n_im 2D matrix
U, W, VT = np.linalg.svd(D, full_matrices=False)
lowrank = np.linalg.multi_dot([U[:, :k], np.diag(W[:k]), VT[:k]])
imapprox = np.reshape(lowrankapprox.T, images.shape)  # back to original shape
imapprox += meanimage                                 # the final low-rank approximation
```

Actually this approximation is of rank $k+1$, since I used the mean as one component plus another $k$ basis images from the SVD. There is a little bit of extra work to get things into the right shape and order, but it's not so bad. If you want to save the basis images in their own right, the first basis image (after the mean), for example, would be the following:

```
basis_1 = np.reshape(U[:, 0], images[0].shape)
```

Again, we need to go back and forth between flattened arrays and arrays shaped like detectors, but it's still just a few lines of code. You could also compute your low-rank approximation to a new image by using least-squares fitting on the basis images, just as we did when fitting models in a previous class. If you use least-squares to fit these basis images to your actual data, you would get the same result that you do when multiplying out $\mathbf{U}$, $\mathbf{W}$, and $\mathbf{V}^T$ after discarding all but the first few $W_{ii}$.

In the preceding discussion, we said nothing about how to choose the rank of the approximation. This is as much art as science, and the arbitrary nature of this choice limits (at least somewhat) the usefulness of PCA. Once you call the SVD, you do get the singular values of the data matrix. You can then plot the fraction of the variance explained by the first $k$ principal components:

```
U, W, VT = np.linalg.svd(D, full_matrices=False)
varfrac = np.cumsum(W**2)/np.sum(W**2)
```

Using `cumsum`, we get everything at once: `varfrac[0]` is the fraction of the variance explained by the first principal component, `varfrac[1]` is the fraction explained by the first two, etc. Once you have performed PCA, you can then make a plot of the fraction of variance explained against the number of principal components. Maybe that plot will have a kink in it, where (e.g.) the first five PCs each explain a lot of variance, but the sixth and further PCs explain much less. In that case, it would be natural to use a rank-5 approximation. However, the choice will be specific to your problem, and often there isn't such a nice kink in a plot of residual variance fraction against $k$. In that case, use your judgment and *make sure that your results are robust to your (arbitrary) choice of $k$.*

It is common to apply PCA to some very large system. For example, I might have 1000 images, each $1000 \times 1000$ pixels. The matrix $\mathbf{U}$, even with `full_matrices=False`, will be $10^6 \times 10^3$, and could be extremely expensive to compute. SVD of an $m \times n$ matrix with $m > n$ scales as $mn^2$, so for a $10^6 \times 10^3$ matrix, we need several $10^{12}$ floating point operations. That's a lot. This SVD would take around an hour on a computer that can perform $10^9$ floating point operations per second (flops). And after all of that, we probably won't pay any attention to anything beyond the first few principal components. There are other ways to get something close to the SVD but using a random starting point and iterative improvement. These can be much, much faster when only computing the first few principal components. When you only compute the first $k$ columns of $\mathbf{U}$ in the SVD, the algorithm is known as *truncated SVD*. The down side is that you have to decide the rank of your approximation in advance. If you want to change the rank of the approximation, you have to redo the whole iterative calculation. In that case, you might choose to compute a rank 10 approximation and see what the residuals look like, then try a rank 5 approximation, then a rank 15 approximation, and then decide what to use for your science. Even though you have to redo the calculation each time, the savings from not computing the full matrix $\mathbf{U}$ can be so large that this is still worthwhile. The package Skikit Learn (`sklearn`) includes an implementation of the truncated SVD.

PCA solves the low-rank approximation where my goodness-of-fit metric is the sum of the squares of the residuals. In $\chi^2$, I was also free to weight each point in the data. *I cannot do that in PCA.* To see why, remember that the SVD of the (mean-subtracted) data matrix immediately gave us the SVD of the covariance matrix. Now let's apply weights to the mean-subtracted data matrix $\mathbf{D}'$ to get a new matrix $\mathbf{M}$:

$$M_{ij} = \frac{D'_{ij}}{\sigma_{ij}}. \tag{13}$$

Remember that this worked fine when using SVD to solve a linear system of equations. But look what happens here. The units of $\mathbf{D}'$ were the units of my data, and the eigenimages had those same units. The matrix $\mathbf{M}$ is unitless and its eigenimages will also be unitless–they will be measured in units of standard deviations from the mean. If a data point has no valid measurement ($\sigma_{ij} = \infty$), then $M_{ij} = 0$. PCA will try to approximate this point with a value of zero, when we really wanted PCA to ignore it entirely. Also, PCA will give me an approximation of the weighted residual matrix where the rank of the approximation (the number of these strange basis images with units of standard deviations) is $k$. The rank of the data when I remove the weights and transform back into regular image units will probably be full–the original number of images. That is *not* what we wanted. Finally, if I want to fit a new image with these weird basis images, and the noise structure is a bit different (even if the data look the same), the fit will probably be poor. The weights/typical errors will be hiding in the basis images, when we really want them to be completely separate and used only to define the closeness of our low-rank approximation.

So, PCA solves a particular problem, and this has significant limitations. PCA gives the low-rank approximation that minimizes the sum of the unweighted squares of the residuals, without any additional constraints on the approximating matrices. Just like $\chi^2$ is not robust to outliers, neither is PCA, and I cannot fix this in PCA by just downweighting the data. I can do PCA iteratively and replace outliers with the predictions for that image and pixel from PCA and then re-run PCA; you might find some success with this approach. You can also just set outliers equal to zero in the mean-subtracted data. This isn't exactly what you want, but it's better than leaving the outliers in there.

There are two methods I will briefly discuss in addition to PCA before leaving this topic behind. The first is the weighted low-rank approximation, and the second is non-negative matrix factorization. The weighted low-rank approximation generalizes PCA in the sense that it finds a rank-$k$ approximation to minimize

$$\chi^2 = \sum_{i,j} w_{ij} \left(D_{ij} - D'_{ij}\right)^2 \tag{14}$$

where $i$ runs over images (or spectra, or whatever) and $j$ runs over the pixels/wavelengths/etc. in those data. Now I am free to apply a separate weight to each pixel of each image, and to throw out bad or missing data by applying a weight of zero. I might identify $w_{ij}$ with the inverse variance, $1/\sigma_{ij}^2$. The weighted low-rank approximation will require you to decide the rank of your approximation in advance. It will then start with a random guess and iteratively improve the approximation. There are a few disadvantages relative to PCA: you don't get all of the basis images and weights at once, and the basis images are not orthogonal. However, once you include weights, orthogonality wouldn't be useful anyway, and for large systems computing all of the basis images using SVD can be extremely expensive. The weighted low-rank approximation has a great advantage: it solves a problem that is much more relevant to most data sets. I'm not sure why it isn't more popular. If you want to apply the weighted low-rank approximation to your research, let me know and I will share python code with you.

The other low-rank approximation is nonnegative matrix factorization (NMF or NNMF). This is an approximate factorization of a data matrix $\mathbf{D}$ (not mean subtracted) of the form

$$\mathbf{D} \approx \mathbf{B} \cdot \mathbf{W}, \tag{15}$$

where the first matrix $\mathbf{B}$ is the matrix of basis images and has dimensions $n_{\mathrm{pixels}} \times k$ ($k$ being the rank of the approximation), and the second matrix $\mathbf{W}$ is a matrix of weights, dimensionality $k \times n_{\mathrm{images}}$. The "$\approx$" sign still means approximately equal in the sense of minimizing the (unweighted) squares of the residuals. However, NMF imposes the constraint on $\mathbf{B}$ and $\mathbf{W}$ that none of their elements can be negative. This might be physically meaningful in the case of, e.g., astronomical images. No astrophysical source will produce negative intensity, and I should be able to represent an image by a sum of nonnegative coefficients describing the contributions of various sources. Again, the solution to this problem isn't a straightforward application of linear algebra; we need to call some specialized, iterative solver.

It is something of a miracle that the solution to the unweighted low-rank approximation is just given by SVD, the same technique we used to solve the linear least-squares problem. We have gotten a lot of mileage out of SVD. The fact that PCA is so simple and deterministic, that we can get the answer just by a factorization of our data matrix, probably has a lot to do with its popularity. In many cases, however, I suspect that you will be better served by some other low-rank approximation. I urge you to think carefully about your particular problem, why you need to approximate your data with a smaller basis (do you want to look at the basis images and use them elsewhere?), and whether you want to impose constraints on your approximation. Finally, remember that all of the dimensionality reduction techniques we have discussed here are linear. If your data live in some low-dimensional parameter space, but the relationship between these parameters and, e.g., the images is highly nonlinear, you might find that none of these techniques are particularly useful.