Section 08: PCA and SVD

1. Singular Value Decomposition

(a) Let A be a symmetric $n \times n$ matrix. Take U to be a matrix whose columns are an orthonormal eigenbasis of A, and S to be a diagonal matrix of eigenvalues (ordered the same as U). Show that for any vector x, Ax is the same as $USU^T x$ (i.e. USU^T is the SVD of A)

(b) Let A have SVD USV^T. Show AA^T has the columns of U as eigenvectors with associated eigenvalues S^2.

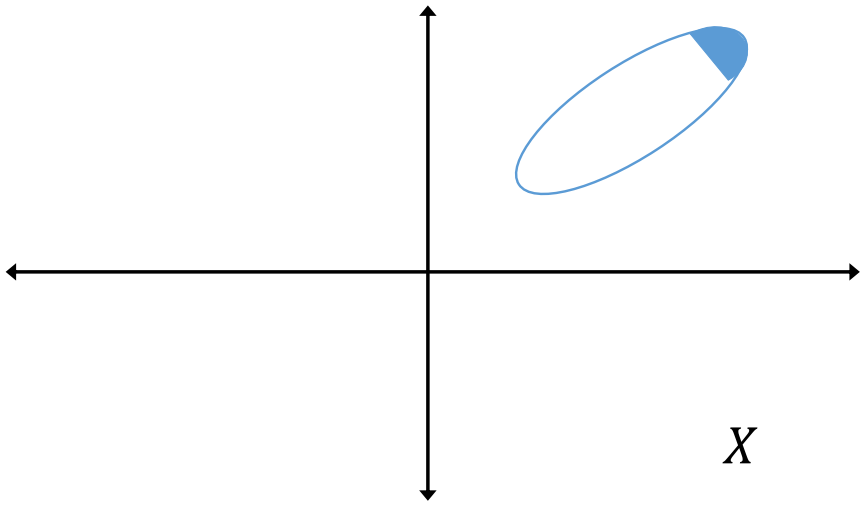
(c) Let A have SVD USV^T. Show $A^T A$ has the columns of V as eigenvectors with associated eigenvalues S^2.

2. SVD Pictures

We've seen before that demeaning our data makes it easier to work with. There's a more general operation called "whitening" where we also normalize the important directions of our data. In this problem, we'll do the operations corresponding to one version of whitening as a way to get better intuition on how SVD works.

Let $X \in \mathbb{R}^{n \times d}$ be a matrix of data points, and J be $\mathbf{I} - \mathbf{1}\mathbf{1}^T/n$. Let JX have a singular value decomposition of $JX = USV^T$.

Suppose we know that our points were drawn from a Gaussian distribution with covariance Σ. We would expect most of our points to lie in an ellipse, whose axes are the eigenvectors of Σ, scaled by the corresponding eigenvectors. We've drawn that ellipse below, with one area shaded so we can see its orientation.



For each of the following matrices:

- Verify that the resulting matrix is still $n \times d$, and therefore can be interpreted as modifying the datapoints of X
- Draw what the resulting data set would look like (i.e. how would the ellipse representing the covariance move?).

(a) JX

(b) JXV

(c) $JXVS^{-1}$, where S^{-1} is the $d \times d$ diagonal matrix, where $S^{-1}_{i,i} = 1/S_{i,i}$ (note that since S is not square, calling a matrix S^{-1} is an abuse of notation)

(d) $JXVS^{-1}V^T$

3. Extra Material: Kernel Principal Component Analysis

This material here is intended as a way to get more practice with both PCA and kernels – we won't expect you to have a deep understanding of kernel PCA as a process.

We've seen that kernels are a way of using complicated feature maps without actually calculating the feature maps, as long as:

- We only care about inner products in the feature space.
- The new feature space is chosen so that we can use the "kernel trick" to calculate dot products directly, without actually applying the feature map.

Since we know:

- PCA is useful in settings where we have linear relationships in our data.
- feature maps can let us use linear methods even when our data has non-linear relationships

it's a reasonable idea to think we might try to apply a feature map, and do PCA in the feature space. But we've seen that we can use very powerful feature maps if all we care about is inner products (via the kernel trick).

In this problem we'll see that this is possible. Let $\{x_i\}_{i=1}^n$ be a set of datapoints in d-dimensions, already demeaned. Define $\Sigma = \frac{1}{n}\sum_{i=1}^n x_i x_i^T$ to be the $(d \times d)$ emperical covariance matrix of the data points.

(a) Let $\phi : \mathbb{R}^d \to \mathbb{R}^k$ be a feature map. What is the formula for Ψ, the emperical covariance matrix of $\{\phi(x_i)\}_{i=1}^n$? What are its dimensions?

(b) Let v_j be an eigenvector of Ψ. Show that Ψv_j can be written as a linear combination of $\phi(x_i)$.

(c) Use the observation from the previous part to write the following equation:

$$\frac{1}{n}\sum_{i=1}^n \phi(x_i)\phi(x_i)^T \sum_{i=1}^n a_{ij}\phi(x_i) = \lambda_j \sum_{i=1}^n a_{ij}\phi(x_i)$$

where a_{ij} are scalars.

(d) Rewrite the equation above so that it uses the kernel matrix (Hint: left-multiply by $\phi(x_\ell)^T$ first).

(e) Use the previous equation to generate a matrix version of this equation (Hint: what would happen if you replaced $\phi(x_\ell)$ with each x_i in order and stacked the equations?)

(f) Argue that if we have a ϕ that allows for the "kernel trick" everything we need to do for PCA (i.e. finding X, a_j and calculating $\phi(x)^T v_j$ for any x and v_j can always be done without going into the kernel space (i.e. using only the kernel function).