STATS 306B: Unsupervised Learning | Spring 2014

Lecture 8 — April 23

Lecturer: Lester Mackey | *Scribe: Kexin Nie, Nan Bi*

8.1 Principal Component Analysis

Last time we introduced the mathematical framework underlying Principal Component Analysis (PCA); next we will consider some of its applications. Please refer to the accompanying slides.

8.1.1 Examples

Example 1. Digit data (Slide 2:) Here is an example taken from the textbook. This set of hand written digital images contains 130 threes, and each three is a 16×16 greyscale image. Hence we may represent each datapoint as a vector of 256 greyscale pixels. (Slide 3:) The figure on the left shows the first two principal components of these images. The rectangular grid is computed by selected quantiles of the two principal components. Based on the projected coordinates on the two directions, the circled points refer to the images that are closest to these vertices of the grid. The figure on the right displays the threes corresponding to the circled points. The vertical component appears to capture changes in line thickness / darkness, while the horizontal component appears to capture changes in the length of the bottom of the three. (Slide 4:) This is a visual representation of the learned two-component PCA model. The first term is the mean of all images, and the following v_1 and v_2 are two visualized principal directions (the loadings), which can also be called "eigen" threes.

Example 2. Eigen-faces (Slide 5:) PCA is widely used in face recognition. Suppose $X_{d\times n}$ is the pixel-image matrix, where each column is a face image. d is the number of pixels and x_{ji} is the intensity of j-th pixel in image i. The loadings returned by PCA are linear combinations of faces, which can be called "eigen-faces." The working assumption is that the PC scores z_i, gotten by projecting the original image onto the eigen-face space, represent a more meaningful and compact representation of the i-th face than the raw pixel representation. Then z_i can be used in place of x_i for nearest-neighbor classification. Since the dimension of face-space has decreased from d to k, the computational complexity becomes $\mathcal{O}(dk + nk)$ instead of $\mathcal{O}(dn)$. This is of great efficiency when $n, d \gg k$.

Example 3. Latent semantic analysis (Slide 6:) Another application of PCA is in text analysis. Let d to be the total number of words in the vocabulary; then each document $x_i \in \mathbb{R}^d$ is a vector of word counts, and x_{ji} is the frequency of word j in document i. After we apply PCA here, the similarity between two documents is now $z_i^T z_j$, which is often more informative than the raw measure $x_i^T x_j$. Notice that there may not be significant computational savings, since the original word-document matrix was sparse, while the reduced representation is typically dense.

Example 4. Anomaly detection (Slide 7:) PCA can be used in network anomaly detection. In the time-link matrix X, x_{ji} represents the amount of traffic on link j in the network during time-interval i. In the two pictures on the left, traffic appears periodic and reasonably deterministic on the selected principal component, which asserts that these two are normal behaviors. In the contrast, traffic "spikes" in the pictures on the right, which indicates anomalous behavior in this flow.

Example 5. Part-of-speech tagging (Slide 8:) Unsupervised part-of-speech tagging is a common task in natural language processing, as manually tagging a large corpus is expensive and time-consuming. Here it is common to model each word in a vocabulary by its context distribution, i.e., x_{ji} is the number of times that word i appears in context j. The key idea of unsupervised POS tagging is that words appearing in similar contexts tend to have same POS tags. Hence, a typical tagging technique is to cluster words according to their contexts. However, in any given corpus, any given context may occur rather infrequently (the vectors x_i are too sparse), so PCA has been used to find a more suitable, comparable representation for each word before clustering is applied.

Example 6. Multi-task learning (Slide 9:) In multi-task learning, one is attempting to solve n related learning tasks simultaneously, e.g., classifying documents as relevant or not for n users. Often task i reduces to learning a weight vector x_i which produces for example the classification rule. Our goal is to exploit the similarities amongst these tasks to do more effective learning overall. One way to accomplish this is to use PCA is to identify a small set of eigen-classifiers among the learned rules $x_1, \ldots, x_n$. Then, the classifiers can be retrained with an added regularization term encouraging each x_i to lie near the subspace spanned by the principal directions. These two steps of PCA and retraining are iterated until convergence. In this way, low-dimensional representation of classifiers can help to detect the shared structures between independent tasks.

8.1.2 Choosing a number of components

(Slide 10:) As in the clustering setting, we face a model selection question: how do we choose the number of principal components? While there is no agreed-upon solution to this problem, here are some guidelines.

- The number of principal components might be constrained by the problem goal, your computational or storage resources, or by the minimum fraction of variance to be explained. For example, it is common to choose 3 or fewer principal components when doing visualization problems.

- Recall that eigenvalue magnitudes determine the explained variance. In the accompanying figure, the first 5 principal components already explain nearly all of the variance, so a small number of principal components may be sufficient (although one must use care in drawing this conclusion, since small differences in reconstruction error may still be semantically significant; consider face recognition for example). Furthermore, we may look for elbow criterion or compare explained variance with that obtained under a reference distribution.

8.1.3 PCA limitations and extensions

While PCA has a great number of applications, it has its limitations as well:

- Squared Euclidean reconstruction error is not appropriate for all data types. Various extensions, such as **exponential family PCA**, have been developed for binary, categorical, count, and nonnegative data.
- PCA can only find **linear** compressions of data. **Kernel PCA** is an important generalization designed for non-linear dimensionality reduction.

8.2 Non-linear dimensionality reduction with kernel PCA

8.2.1 Intuition

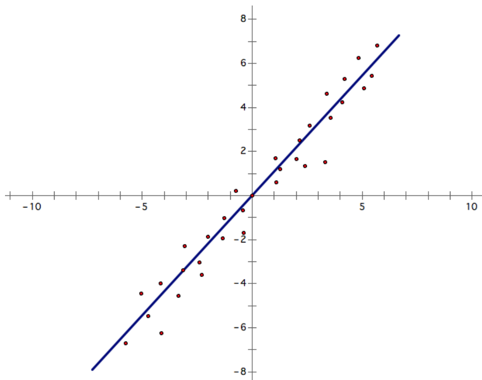

Figure 8.1. Data lying near a linear subspace

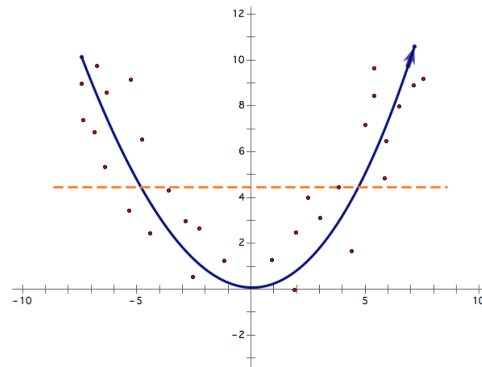

Figure 8.2. Data lying near a parabola

Figure 8.1 displays a 2D example in which PCA is effective because data lie near a linear subspace. However, in Figure 8.2 PCA is ineffective, because data the data lie near a parabola. In this case, the PCA compression of the data might project all points onto the orange line, which is far from ideal. Let us consider the differences between these two settings mathematically.

- **Linear subspace (Figure 8.1)**: In this example we have ambient dimension $p = 2$ and component dimension $k = 1$. Since the blue line is a k-dimensional linear subspace of $\mathbb{R}^p$, we know that there is some matrix $U \in \mathbb{R}^{p\times k}$ such that the subspace S takes the form

$$\begin{aligned} S &= \{x \in \mathbb{R}^p : x = Uz, z \in \mathbb{R}^k\} \\ &= \{(x_1, x_2) : x_1 = u_1 z, x_2 = u_2 z\} \\ &= \{(x_1, x_2) : x_2 = \frac{u_2}{u_1} x_1\}, \end{aligned}$$

where $U = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$, since $(p, k) = (2, 1)$ in our example.

- **Parabola (Figure 8.2)**: In this example we again have ambient dimension $p = 2$ and component dimension $k = 1$. Moreover, there is some fixed matrix $U \in \mathbb{R}^{p\times k}$ such that the underlying blue parabola takes the form

$$S' = \{(x_1, x_2) : x_2 = \frac{u_2}{u_1} x_1^2\}$$

which is similar to the representation derived in the linear model. Indeed, if we introduce an auxiliary variable z, we get,

$$\begin{aligned} S' &= \{(x_1, x_2) : x_1^2 = u_1 z, x_2^2 = u_2 z, \text{for } z \in \mathbb{R}\} \\ &= \{x \in \mathbb{R}^p : \phi(x) = Uz, z \in \mathbb{R}^k, \} \end{aligned}$$

where $\phi(x) = \begin{bmatrix} x_1^2 \\ x_2 \end{bmatrix}$ is a non-linear function of x. In this final representation, U is still a linear mapping of the latent components z, but the representation being reconstructed linearly is no longer x itself but rather a potentially non-linear mapping ϕ of x.

8.2.2 Take-away

We should be able to capture non-linear dimensionality reduction in x space by performing linear dimensionality reduction in $\phi(x)$ space (we often call $\phi(x)$ the **feature space**). Of course we still need to find the right feature space to perform dimensionality reduction in. One option is to hand-design the feature mapping ϕ explicitly coordinate by coordinate, e.g., $\phi(x) = (x_1, x_2^2, x_1 x_2, \sin(x_1), \ldots)$. However, this process quickly becomes tedious and has to be *ad hoc*. Moreover, working in feature space becomes expensive if $\phi(x)$ is very large. For example, consider the number of all quadratic terms $x_i x_j = \mathrm{O}(p^2)$.

An alternative, which we will explore next, is to encode ϕ implicitly via its inner products using the **kernel trick**.

8.2.3 The Kernel Trick

Our path to the kernel trick begins with an interesting claim: the PCA solution depends on the data matrix

$$X = \begin{bmatrix} - & x_1 & - \\ - & x_2 & - \\ & \cdots & \\ - & x_n & - \end{bmatrix}$$

only through the **Gram matrix** (a.k.a. the **Kernel matrix**), $K = XX^T \in \mathbb{R}^{n*n}$. The kernel matrix is the matrix of inner products $K_{ij} =< x_i, x_j >$.

Proof. Each Principal Component loading u_j is an eigenvector of $\frac{X^T X}{n}$
$\Rightarrow \frac{X^T X}{n} u_j = \lambda_j u_j$ for some λ_j
$\Rightarrow \boxed{u_j = X^T \alpha_j = \sum_{i=1}^n \alpha_{ji} x_i}$ for some weights α_j. That is, u_j is a linear combination of the datapoints. This is called a **representer theorem** for the PCA solution. It is analogous

to representer theorems you may have seen for Support Vector Machines or ridge regression. Therefore one can restrict attention to candidate loadings u_j with this form.

Now consider the PCA objective

$$\max_{u_j} \; u_j^T \frac{X^T X}{n} u_j \quad \text{s.t.} \; \| u_j \|_2 = 1, \; u_j^T \frac{X^T X}{n} u_l = 0, \; \forall l < j$$

$$\Leftrightarrow \max_{\alpha_j} \; \alpha_j^T X \frac{(X^T X)}{n} X^T \alpha_j \quad \text{s.t.} \; \alpha_j^T X X^T \alpha_j = 1, \quad \alpha_j^T X \frac{(X^T X)}{n} X^T \alpha_l = 0, \; \forall l < j$$

$$\Leftrightarrow \max_{\alpha_j} \; \alpha_j^T \frac{K^2}{n} \alpha_j, \quad \text{s.t.} \; \alpha_j^T K \alpha_j = 1, \; \alpha_j^T \frac{K^2}{n} \alpha_l = 0, \; \forall l < j \tag{8.1}$$

which only depends on the data through K! □

The final representation of PCA in kernel form (8.1) is an example of a **generalized eigenvalue problem**, so we know how to compute its solution. However, we will give a more explicit derivation of its solution by converting this problem into an equivalent eigenvalue problem. Hereafter we will assume K is non-singular.

Let $\beta_j = K^{\frac{1}{2}} \alpha_j$ so that $\alpha_j = K^{-\frac{1}{2}} \beta_j$. Now the problem becomes (8.1)

$$\max_{u_j} \; \frac{\beta_j^T K \beta_j}{n}, \quad \text{s.t.} \; \beta_j^T \beta_j = 1, \; \beta_j^T \frac{K}{n} \beta_j = 0, \; \forall l < j$$

This is a eigenvalue problem with solution given by

$$\beta_j^* = \text{ the j-th leading eigenvector of K} \quad \text{and hence} \quad \alpha_j^* = K^{-\frac{1}{2}} \beta_j^* = \frac{\beta_j^*}{\sqrt{\lambda_j(K)}}.$$

Furthermore, we can recover the principal component scores from this representation by

$$z = u^{*T} X^T = [\alpha_1^*, \ldots, \alpha_k^*]^T X X^T = [\alpha_1^*, \ldots, \alpha_k^*]^T K.$$

The punchline is that we can solve PCA by finding the eigenvectors and eigenvalues of K; this is **kernel PCA**, the kernelized form of the PCA algorithm (note that the solution is equivalent to the original PCA solution if $K = XX^T$). Hence, the inner products of X are sufficient, and we do not need additional access to explicit datapoints.

Why is this relevant? Suppose we want to run kernel PCA on a non-linear mapping of data

$$\Phi = \begin{bmatrix} - & \phi(x_1) & - \\ - & \phi(x_2) & - \\ & \cdots & \\ - & \phi(x_n) & - \end{bmatrix}.$$

Then we do not need to compute or store Φ explicitly; $K^\phi = \Phi \Phi^T$ suffices to run kernel PCA. Moreover, we can often compute entries of $K_{ij}^\phi = < \phi(x_i), \phi(x_j) >$ via a kernel function $K(x_i, x_j)$ without forming $\phi(x_i)$ explicitly. This is the kernel trick. Here are a few common examples:

Kernel trick examples

Kernel	$K(x_i, x_j)$	$\phi(x)$
Linear	$< x_i, x_j >$	x
Quadratic	$(1+ < x_i, x_j >)^2$	$(1, x_1, \ldots, x_p, x_1^2 \ldots, x_p^2, x_1x_2, \ldots, x_{p-1}x_p)$
Polynomial	$(1+ < x_i, x_j >)^d$	all monomials of order d or less
Gaussian/ Radial basis function	$\exp\left(\frac{-\|x_i - x_j\|_2^2}{\sigma^2}\right)$	infinite dimensional feature vector

A principal advantage of the kernel trick is that one can carry out non-linear dimension reduction with little dependence on the dimension of the non-linear feature space. However, one has to form and operate on a $n \times n$ matrix (which can be quite expensive). It is common to approximate the kernel when n is large using random (e.g., the Nystrom method of Williams & Seeger, 2000) or deterministic (e.g., the incomplete Cholesky decomposition of Fine & Scheinberg, 2001) low-rank approximations.