Lecture 6: Principal component analysis

CS 3780/5780, Sp25

Tushaar Gangavarapu (TG352@cornell.edu)

In the previous lecture, we introduced the unsupervised regime and saw two *clustering* algorithms: k-means (non-probabilistic) and mixture of Gaussians (probabilistic). Let's recap the mixture of Gaussians before proceeding.

Recap (mixture of Gaussians). The goal was to make "soft" assignments (unlike in k-means where we said, each point belonged to a single cluster). To this end, we assumed that the data we observe was generated by k Gaussians and asked: "What is the probability that a point, $x^{(j)}$, was generated by Gaussian-i, given that we actually observed the point in our data?"

Our algorithm to recover these Gaussians was as follows (for $k = 2$):

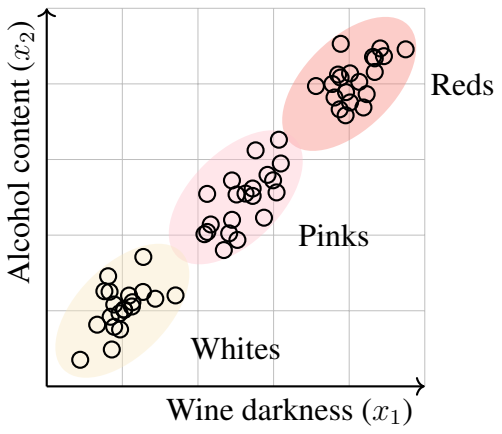
(a) Guess the parameters of the Gaussians: μs and σs, and the probability of picking each Gaussian, p.

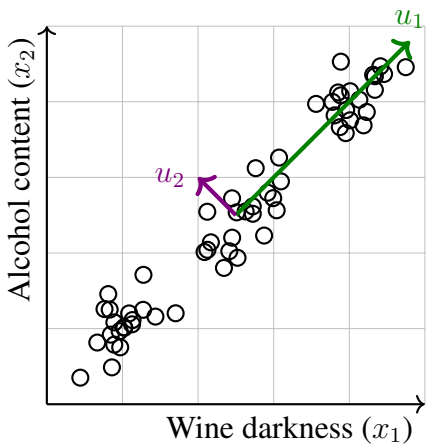
(b) Compute the soft assignments: $P(Z^j = i | x^j; \mu^{(i)}, \sigma^{(i)}, p^{(i)})$[1]

(c) Recompute the parameters μs and σs by taking a weighted average (of all points for μs and of deviations from μ for σs); estimate p as the average probability, $P(Z^j = i | x^j)$.

In this lecture, we will continue in the same regime but a slightly different setting, where we are interested in finding a subspace (and not clustering), if it exists, in which data approximately lies—specifically, we will discuss the principal component analysis (PCA).

As a motivating example, consider that we have a dataset of cars, and we collected d features for each car, $\{x_1, \ldots, x_d\}$[2]—max speed, turn radius, etc. Now, unbeknownst to us, say $x^{(10)}$ is the car's max speed in mph and $x^{(26)}$ is the max speed in kph. One can easily argue that our data really lies in a $d - 1$-dimensional subspace. The question we are going to ask is if there is an automated way of removing such redundancy in our data.





[1] One thing to note here is that when we say $P(x^{(j)} | Z^{(j)} = i)$, we don't mean the probability that Gaussian takes a specific value, $x^{(j)}$; what we mean is that it's the *probability density* or likelihood of $x^{(j)}$ under Gaussian-i, which can be computed using the "$1/\sqrt{2\pi}\sigma^{(i)} \cdots$," which tells us how "dense" probability mass is around $x^{(j)}$, but not the probability of any single point.

[2] We will use subscript to denote the feature index and superscript to denote the data index.

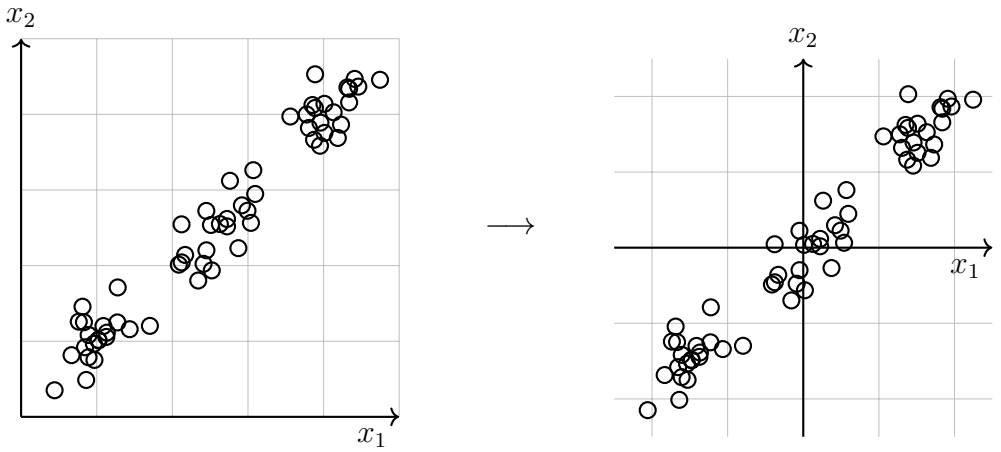
While the toy example drives the motivation, let's look at a less contrived example of wine tasting,[3] where x_1 is the darkness of the wine and x_2 is the alcohol content in the wine.

As it turns out, x_1 and x_2 are strongly correlated, and indeed, one might realize that data actually lives along the u_1 axis (shown above to the right). This is equivalent to saying what is the direction that gives the best possible reconstruction of my data? The question now is how do we compute this u_1 direction?

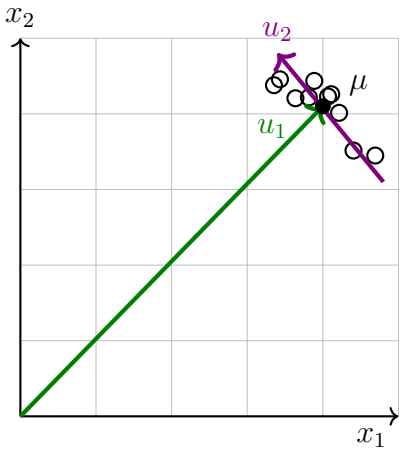
1 Preprocessing the data

The first thing we are going to do is to center our data by zeroing out the mean as follows:

$$\mu = \frac{1}{n}\sum_{j=1}^{n} x^{(j)}; \qquad x^{(j)} \leftarrow x^{(j)} - \mu.$$



The key problem in working with data with sufficiently large mean is the direction of μ becomes the most-interesting axis, i.e., the direction that gives the best possible reconstruction is the μ axis. In the figure to the right, the best approximation is by the mean vector, but we are more interested in understanding the relationship between x_1 and x_2. (We will make this more concrete once we pose PCA as an optimization problem.)



Next, we may need to normalize the data such that each feature has unit variance:

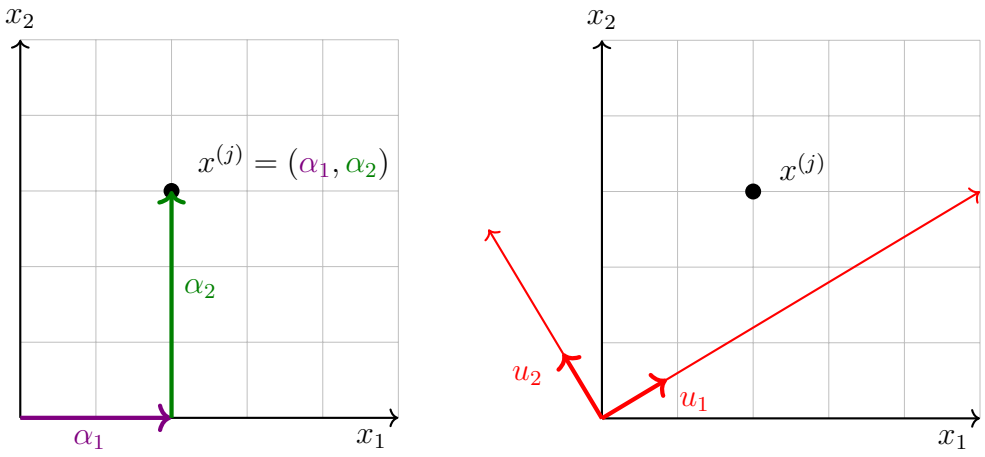
$$\sigma_i^2 = \frac{1}{n}\sum_{j=1}^{n}(x^{(j)} - \mu_i)^2; \quad x^{(j)} \leftarrow \frac{x^{(j)}}{\sigma_i},$$

which ensures that different features are treated on the same scale. For example, wine darkness is measured in nanometers (typically in hundreds), while alcohol content is measured in alcohol-by-volume (percentage)—variance normalization makes them more comparable.

[3] Inspired from https://stats.stackexchange.com/a/140579.

2 Principal component analysis

2.1 Distance to the chosen directions



A given point in our 2D coordinate system, $x^{(j)} = (\alpha_1, \alpha_2)$, indicates that we walk α_1 steps along the x_1 direction and α_2 steps along the x_2 direction, starting from $(0, 0)$.

Now, consider different axes shown in red, u_1 and u_2 are *unit* vectors in each direction, how do we compute the coordinates of $x^{(j)}$ in this tilted coordinate system? Now, α_1 would simply be the closest point to $x^{(j)}$ on u_1-axis and we can compute it as:[4]

$$\begin{aligned}
\alpha_1 &= \arg\min_{\alpha} \|x^{(j)} - \alpha u_1\|^2 \\
&= \arg\min_{\alpha} (x^{(j)} - \alpha u_1)^T (x^{(j)} - \alpha u_1) \\
&= \arg\min_{\alpha} {x^{(j)}}^T x^{(j)} - 2\alpha u_1^T x^{(j)} + \alpha^2 u_1^T u_1 \\
&= \arg\min_{\alpha} \|x^{(j)}\|^2 - 2\alpha u_1^T x^{(j)} + \alpha^2 \underbrace{\|u_1\|^2}_{=1}.
\end{aligned}$$

To compute the minimizer we simply compute the derivate of the above with respect to α and set it to zero:

$$\nabla_\alpha \|x^{(j)}\|^2 - 2\alpha u_1^T x^{(j)} + \alpha^2 = -2u_1^T x^{(j)} + 2\alpha \overset{\text{set}}{=} 0,$$

which gives us $\alpha = u_1^T x^{(j)}$—the closest point to $x^{(j)}$ on u_1-axis is $\langle x^{(j)}, u_1 \rangle$. This may seem obvious, but it's important to note nonetheless. Similarly, we can solve for $\alpha_2 = \langle x^{(j)}, u_2 \rangle$. Now, one final thing to note is that $x^{(j)} = \alpha_1 u_1 + \alpha_2 u_2$, i.e., $x^{(j)}$ can be expressed as a linear combination of u_1 and u_2, which are often referred to as the *basis* vectors.

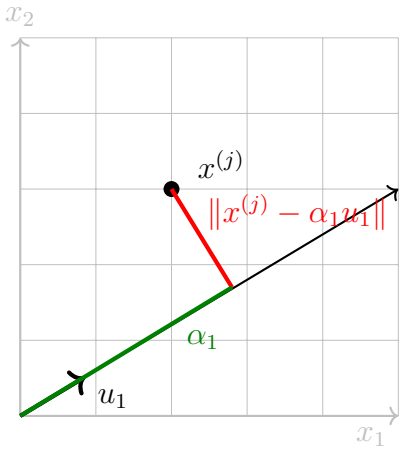
We can extend the above from 2D to n dimensions, and ask: "What is the closest point to $x^{(j)}$ on some k-dimensional hyperplane?"—this would simply involve finding $\alpha_1, \ldots, \alpha_k$ as

$$\arg\min_{\alpha_1,\ldots,\alpha_k} \|x^{(j)} - \sum_{i=1}^{k} \alpha_i u_i\|^2.$$

We leave it as a self exercise to solve this to see that $\alpha_i = \langle x^{(j)}, u_i \rangle$.

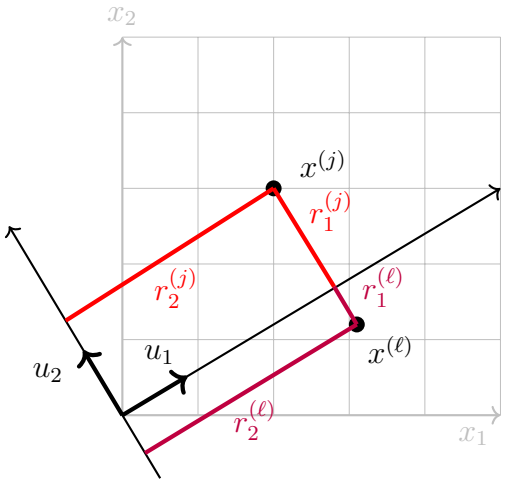
[4] When we say α_1 here, what we really mean is $\alpha_1^{(j)}$, since α_1 is dependent on $x^{(j)}$; we will make this more explicit in the following sections.

2.2 PCA as an optimization problem

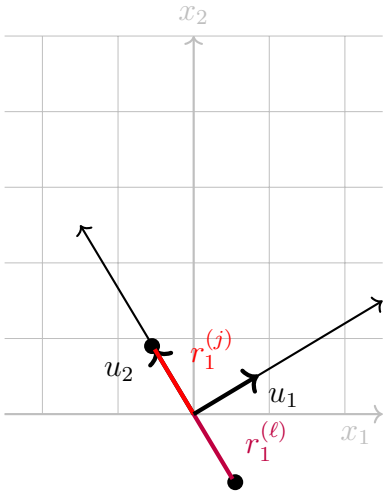


Recall that our motivation is to find a principal direction that explains our data, or equivalently, the direction that facilitates best possible reconstruction. More precisely, if we could pick any u_1, which would you choose? We can reason about this in terms of "how much remains unexplained in our data by choosing u_1." This way of thinking is important if our goal is dimensionality reduction, where we reduce hundreds and thousands of features into two or three, or even ten features.[5] In the example to the left, $x^{(j)} - \alpha_1 u_1$ is unexplained, and is often referred to as the *residual*, and we will denote it as $r^{(j)}$.

One way to think about this task of finding u_1 is that we want choose u_1 to minimize the residual. In case of a single instance dataset, this becomes trivial—we just choose u_1 as the line passing through origin and $x^{(1)}$. Let's consider a small dataset of two points, $x^{(1)}$ and $x^{(2)}$, and two directions, u_1 and u_2 as shown below:



Uncentered data



Centered data ($r_2^{(j)} = r_2^{(\ell)} = 0$)

As an aside, we show why mean centering is crucial. Looking at the right figure, we note that the residual using u_2 is zero, meaning that u_2 reconstructs the data perfectly. Intuitively, this makes sense; if we were to use u_1, both $x^{(j)}$ and $x^{(\ell)}$ would be represented the same, making them indistinguishable along u_1.

Before making this more rigorous, we make a simple observation that minimizing the residuals is equivalent to making α values as large as possible. While this may not seem as obvious at first, the (hand-wavy[6]) argument is as follows: the residual vector and $\alpha_i^{(j)} u_i$ form the non-

[5] An important distinction to make here is that we are not talking about *removing* features, we are instead talking about coming up with new directions, which are linear combinations of old directions, that offer best possible reconstruction of our data.

[6] See https://stats.stackexchange.com/a/136072 for a more formal proof.

hypotenuse sides of a right triangle, meaning, sum of squared residual and $\|\alpha_i^{(j)} u_i\|^2 = (\alpha_i^{(j)})^2$ is constant, since the hypotenuse (distance between origin and $x^{(j)}$) doesn't depend on the orientation of u_i.

In summary, we have two ways of finding the (first) principal direction or component:

(a) minimize the residuals,

$$\arg\min_{\substack{u_1 \in \mathbb{R}^d \\ \|u_1\|=1}} \frac{1}{n} \sum_{j=1}^{n} \|x^{(j)} - \alpha_1^{(j)} u_1\|^2,$$

OR equivalently,

(b) maximize the α values,

$$\arg\max_{\substack{u_1 \in \mathbb{R}^d \\ \|u_1\|=1}} \frac{1}{n} \sum_{j=1}^{n} (\alpha_1^{(j)})^2.$$

In this notes, we will proceed with (b), but optimizing (a) should yield the same result (refer to (Shalizi, 2008) for proof). Why?—maximizing α values has the same effect as saying that we want to retain as much "spread" (or, variance) as in our original data. Now, recall that $\alpha_1^{(j)} = u_1^T x^{(j)} = x^{(j)^T} u_1$:

$$\begin{aligned}
\sum_{j=1}^{n} \alpha_1^2 &= \frac{1}{n} \sum_{j=1}^{n} (u_1^T x^{(j)})^2 \\
&= \frac{1}{n} \sum_{j=1}^{n} (u_1^T x^{(j)})(u_1^T x^{(j)}) \\
&= \frac{1}{n} \sum_{j=1}^{n} (u_1^T x^{(j)})(x^{(j)^T} u_1) \\
&= \frac{1}{n} \sum_{j=1}^{n} u_1^T (x^{(j)} x^{(j)^T}) u_1 \\
&= u_1^T \underbrace{\left(\frac{1}{n} \sum_{j=1}^{n} x^{(j)} x^{(j)^T} \right)}_{\Sigma} u_1.
\end{aligned}$$

Now, our optimization is as follows:

$$\arg\max_{\substack{u_1 \in \mathbb{R}^d \\ \|u_1\|=1}} u_1^T \Sigma u_1,$$

where $\Sigma = (1/n) \sum_{j=1}^{n} x^{(j)} x^{(j)^T}$. Observe that for $x^{(j)} \in \mathbb{R}^d$, we have $\Sigma \in \mathbb{R}^{d \times d}$. We will come back to what the significance of Σ is later, for now, let us solve the optimization problem, subject to $\|u_1\| = 1$. This is often what is referred to as a *contrained* optimization problem and is solved using the method of Lagrange multipliers.[7] We will absorb the constraint into the

[7] https://en.wikipedia.org/wiki/Lagrange_multiplier.

optimization by introducing a new parameter, λ, as follows:

$$\mathcal{L}(u_1, \lambda) = u_1^T \Sigma u_1 - \lambda \underbrace{(u_1^T u_1 - 1)}_{\text{constraint}}.$$

Now our optimization is:

$$\arg\max_{u_i} \left[\arg\min_{\lambda} \mathcal{L}(u_i, \lambda) \right],$$

and we can solve for u_1 by taking the derivative of $\mathcal{L}$ with respect to u_1 and setting it to zero:

$$\nabla_{u_i} \mathcal{L}(u_i, \lambda) = \Sigma u_i - \lambda u_i \overset{\text{set}}{=} 0.$$

Hence, we have $\Sigma u_1 = \lambda u_1$, which is essentially the eigenequation, i.e., λ is an eigenvalue of Σ and u_1 is the corresponding eigenvector.

Okay, a $d \times d$ real symmetric matrix has d real eigenvalues (proof in Appendix B)—so which eigenvector?[8] As it turns out u_1 is the eigenvector corresponding to the *largest* eigenvalue. This is nontrivial (and should be surprising if you haven't seen it before!). If we substitute the solution, $\Sigma u_1 = \lambda u_1$, back in our objective function, we get

$$u_1^T \Sigma u_1 + \lambda \underbrace{(u_1^T u_1 - 1)}_{\text{constraint: } \|u_1\|=1} = u_1^T \underbrace{\Sigma u_1}_{=\lambda u_1} = \lambda u_1^T u_1 = \lambda.$$

Recall that the objective of the constrained optimization was to *maximize* the above, which equals λ. Hence, λ has to be the biggest lambda, and consequently, u_1 is the eigenvector corresponding to the largest eigenvalue.

Once we know u_1, we can represent $x^{(j)}$ in our new 1D system as $x^{(j)} = \alpha_1 = \langle u_1, x^{(j)} \rangle$. We can "bulk" process this through a matrix multiplication as:

$$\underbrace{\begin{bmatrix} -{x^{(1)}}^T - \\ -{x^{(2)}}^T - \\ \vdots \\ -{x^{(n)}}^T - \end{bmatrix}}_{\mathbb{R}^{n \times d}} \underbrace{u_1}_{\mathbb{R}^d} = \underbrace{\begin{bmatrix} {x^{(1)}}^T u_1 \\ {x^{(2)}}^T u_1 \\ \vdots \\ {x^{(n)}}^T u_1 \end{bmatrix}}_{\mathbb{R}^n} = \begin{bmatrix} \alpha_1^{(1)} \\ \alpha_1^{(2)} \\ \vdots \\ \alpha_1^{(n)} \end{bmatrix}$$

In summary, to find the first principal component of maximum variation, we (1) normalize the data to zero mean and unit variance, (2) form Σ, (3) compute its eigenvalues and eigenvectors, and choose u_1 as the eigenvector corresponding to the largest eigenvalue, and (4) compute $\alpha_1^{(j)}$ for each $x^{(j)}$.

Using `numpy.linalg.eig` to compute the eigenvectors, we can compute u_1 in the following *three* lines of code:

```
import numpy as np

# Generate random data with 100 samples and 2 features.
X = np.random.rand(100, 2)

# PCA to transform the data using the first principal component.
X -= X.mean(0) # mean=0, np.random.rand: variance=1
```

[8] Strictly speaking, the covariance matrix, Σ, is formed by adding n rank-one matrices, meaning $\text{rank}(\Sigma) \leq n$. Hence, the rank of $d \times d$ covariance matrix is $\min(n, d)$.

```
_, eigvecs = np.linalg.eig(X.T@X / X.shape[0]) # computes all eigenvectors
X_pca = X@eigvecs[:, 0] # "bulk" compute alpha values
```

Extension beyond finding the first component. In our simple case of 2D, we only needed the (first) principal component of variation. In practice, however, we have hundreds and thousands of features, and are looking to reduce it tens. To yield k principal components, we can optimize:

$$\underset{\substack{u_1,\dots,u_k\in\mathbb{R}^d\\ \|u_i\|=1, u_i^T u_\ell=0}}{\arg\min} \frac{1}{n}\sum_{j=1}^{n}\|x^{(j)} - \sum_{i=1}^{k}\alpha_i^{(j)}u_i\|^2,$$

or equivalently,

$$\underset{\substack{u_1,\dots,u_k\in\mathbb{R}^d\\ \|u_i\|=1, u_i^T u_\ell=0}}{\arg\max} \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{k}(u_i^T x^{(j)})^2.$$

This yields u_2 to be eigenvector of Σ with the second largest eigenvalue, u_3 to the eigenvector of Σ with the third largest eigenvalue, and so on. Hence, the top-k eigenvectors of Σ are the k principal components of maximum variation.

Aside: Covariance. Notice $\Sigma = (1/n)\sum_{j=1}^{n} x^{(j)}{x^{(j)}}^T$ to be the empirical covariance matrix of the data, assuming the data has zero mean (which we can, because we processed it as such). It essentially computes the covariance between every pair of dimensions, and is a $d \times d$ matrix for $x^{(j)} \in \mathbb{R}^d$.

We are interested in noting what happens to the covariance matrix after we apply PCA. With this in mind, let us look at the PCA-transformed data using k principal components:

$$\underbrace{\begin{bmatrix} -{x^{(1)}}^T- \\ -{x^{(2)}}^T- \\ \vdots \\ -{x^{(n)}}^T- \end{bmatrix}}_{\mathbb{R}^{n\times d}} \underbrace{\begin{bmatrix} | & | & & | \\ u_1 & u_2 & \dots & u_k \\ | & | & & | \end{bmatrix}}_{\mathbb{R}^{d\times k}} = \underbrace{\begin{bmatrix} {x^{(1)}}^T u_1 & {x^{(1)}}^T u_2 & \dots & {x^{(1)}}^T u_k \\ {x^{(2)}}^T u_1 & {x^{(2)}}^T u_2 & \dots & {x^{(2)}}^T u_k \\ \vdots & \vdots & \ddots & \vdots \\ {x^{(n)}}^T u_1 & {x^{(n)}}^T u_2 & \dots & {x^{(n)}}^T u_k \end{bmatrix}}_{\mathbb{R}^{n\times k}}.$$

Observe that what was previously $x^{(j)}$ is now $U^T x^{(j)} \in \mathbb{R}^k$, where $U = \begin{bmatrix} u_1 & \dots & u_k \end{bmatrix} \in \mathbb{R}^{d\times k}$. Now, the updated covariance matrix is

$$\frac{1}{n}\sum_{j=1}^{n} x^{(j)}(x^{(j)})^T = \frac{1}{n}\sum_{j=1}^{n}(U^T x^{(j)})(U^T x^{(j)})^T = U^T\left(\frac{1}{n}\sum_{j=1}^{n} x^{(j)}{x^{(j)}}^T\right)U = U^T\Sigma U.$$

Now, noting that $\Sigma U = \Lambda U$, where $\Lambda = \mathrm{diag}(\lambda_1, \dots, \lambda_k)$, we have the updated covariance matrix as $U^T\Lambda U$. This means that non-diagonal entries of the updated covariance matrix are zeros, while the diagonal entries are λ_is. This is quite interesting because it tells us that the principal components are *uncorrelated*!

3 Notable asides

3.1 Data reconstruction

In our 2D example, the PCA transformation using u_1 can simply be written as $\alpha_1^{(j)} = u_1^T(x^{(j)} - \mu)$. Now, we can reconstruct $x^{(j)}$ from $\alpha_1^{(j)}$ as $u_1\alpha_1^{(j)} + \mu$. This can be generalized to k principal

components, transforming d-dimensional data, where $x^{(j)}$ can be reconstructed from $z^{(j)} = U^T x^{(j)} \in \mathbb{R}^k$ as $Uz^{(j)} + \mu$.

What happens if $k = d$, i.e., we had as many principal components as the data dimensions? Then, the reconstruction $Uz^{(j)} + \mu = UU^T x^{(j)} + \mu$. Observe that U is now the orthonormal basis of $\mathbb{R}^d$, i.e., $UU^T = U^T U = \mathrm{I}$. Hence, we would have perfect data reconstruction.

3.2 On the choice of the number of components

The short answer: unfortunately, there is no right way of choosing k, the number of components, and are often determined by various other factors (e.g., how many new dimensions can we analyze for interpretability).

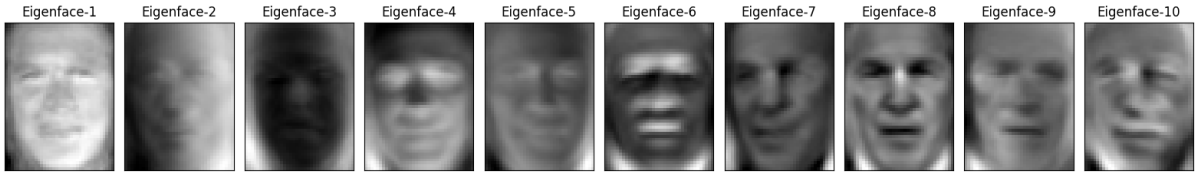
That said, we can always choose the number of clusters based on how much reconstruction loss we are willing to incur. Here, we could follow an approach similar to the elbow method discussed in k-means, where we choose k to be the value beyond which the gains in variance (or, reductions in residuals) is minimal.

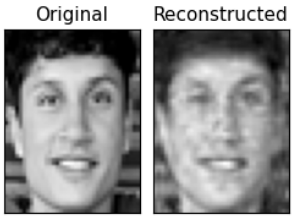
4 Conclusion

In this lecture, we saw PCA as a method of choosing the principal component of variation (which can be extended to k components).

PCA has several applications, and an obvious one (which we have used throughout this lecture notes as a base motivation) is dimensionality reduction. A non-obvious extension of this is visualization—for instance, if we reduce our cars data down to two dimensions, we could plot the data in 2D and see which cars are similar to each other, what groups emerge, etc. Another standard use of PCA (and dimensionality reduction in general) is to preprocess the data to reduce dimensionality before running a supervised learning algorithm.

One of the most famous applications of the PCA is face recognition using "eigenfaces" (Turk et al., 1991).[9] Each face (an image) is $h \times w$-dimensional (in the demo, we use 50×37 images, resulting in $1,850$-dimensional representations), with each coordinate indicating the pixel intensity value. Using PCA, we can represent each image in much lower dimensions. We show the top-10 principal components (or, eigenfaces) obtained from running PCA on the Labeled Faces in the Wild dataset with $k = 300$ (16% of the total features):[10]





In running PCA, we observe that the principal components or eigenfaces retain the interesting and systematic variations between faces that captures what a person really looks like, and not the noise in the images (e.g., lighting variations). We then evaluate this by reconstructing the images in this PCA-reduced subspace, using the steps outlined in §3.1; an example reconstruction is shown to the right.

[9] Interactive demo available here: https://colab.research.google.com/drive/1UJ2JSXaiXHIDzpmTMsbDwHI17SpmZGn4.

[10] https://scikit-learn.org/0.19/datasets/labeled_faces.html.

A Notation

$\mathcal{D}$	The training dataset of n samples
n	The number of training samples in the dataset $\mathcal{D}$
d	The number of feature dimensions
k	The number of principal components
$x^{(j)} \in \mathbb{R}^d$	The d-dimensional (feature) vector associated with the j-th training sample
$x_\ell^{(j)} \in \mathbb{R}$	The ℓ-th element of $x^{(j)}$
μ_i	The mean computed in the i-th dimension
σ_i	The standard deviation computed in the i-th dimension
Σ	The empirical covariance matrix
$u_i \in \mathbb{R}^d$	The i-th principal component
$\alpha_i^{(j)}$	The u_i-th coordinate of the closest point to $x^{(j)}$ on u_i
$Au = \lambda u$	Then, u is an eigenvector of A, with λ as the corresponding eigenvalue
$U \in \mathbb{R}^{d \times k}$	A matrix of k principal components, u_is
Λ	A diagonal matrix with diagonal entries being the sorted eigenvalues—the first diagonal entry is the largest eigenvalue

B Eigenvalues of a real symmetric matrix

Given a real symmetric matrix, $\Sigma \in \mathbb{R}^{d \times d}$, we wish to show that Σ has d real eigenvalues.[11]

Let us first show that every eigenvalue of Σ is real. Recall that $u^H u$ is real for any complex u. For some eigenvalue, λ of A, we have:

$$\lambda(u^H u) = u^H \lambda u = u^H \Sigma u.$$

Now, taking conjugate on both sides, and noting that Σ is real and symmetric, i.e., $\Sigma^H = \Sigma^T = \Sigma$, we have

$$\bar{\lambda}(u^H u) = u^H \Sigma^H u = u^H \Sigma u = \lambda(u^H u).$$

Since $u \neq 0$, we have $\bar{\lambda} = \lambda$, i.e., λ is real. Since we didn't show this for some specific λ, we can conclude that all eigenvalues of Σ are real.

As for why d eigenvalues?—recall that we obtain eigenvalues from solving the characteristic polynomial, $\det(\Sigma - \lambda \mathrm{I}) = 0$, which is a polynomial of degree d. Now, from the fundamental theorem of algebra, we realize that a d-degree polynomial will have d roots, including multiplicity.

□

[11] Note that the covariance matrix is a real symmetric matrix.

References

A. Ng. CS229 Lecture notes. *CS229 Lecture notes*, 1(1):155–160, 2000. URL https://cs229.stanford.edu/main_notes.pdf. Version: June 11, 2023.

C. Shalizi. Mathematics and Interpretation of Principal Components. *Statistics 36-350: Data Mining Lecture Notes*, 1(1):2–5, 2008. URL https://www.stat.cmu.edu/~cshalizi/350/lectures/10/lecture-10.pdf. Last compiled: 18 September 2009.

M. A. Turk, A. Pentland, et al. Face recognition using eigenfaces. In *CVPR*, volume 91, pages 586–591, 1991.

(Last compiled: 2/26/2025, 10.33am ET.)