	Model P(x)	Not probability
Subspace	Factor Analysis	PCA
Groups	Mixture of Gausian	k-mean

lecture 15: Singular value decomposition (SVD) -> compute pca very effectively
X = UDV' with:
X: R(mxn)
U: R(mxn)
D: R(nxn)
V: R(nxn)
Top k value of V is the top k value of eigenvector X'X

CS229 Lecture notes

Andrew Ng

Lecture 14

Part XI

Principal components analysis

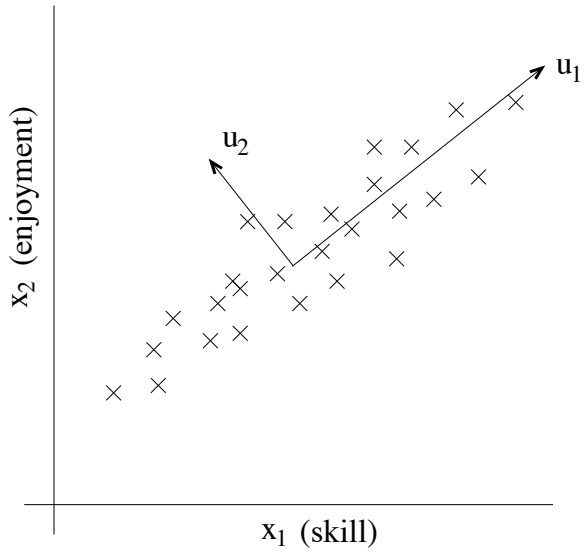
In our discussion of factor analysis, we gave a way to model data $x \in \mathbb{R}^n$ as "approximately" lying in some k-dimension subspace, where $k \ll n$. Specifically, we imagined that each point $x^{(i)}$ was created by first generating some $z^{(i)}$ lying in the k-dimension affine space $\{\Lambda z + \mu; z \in \mathbb{R}^k\}$, and then adding Ψ-covariance noise. Factor analysis is based on a probabilistic model, and parameter estimation used the iterative EM algorithm.

In this set of notes, we will develop a method, Principal Components Analysis (PCA), that also tries to identify the subspace in which the data approximately lies. However, PCA will do so more directly, and will require only an eigenvector calculation (easily done with the `eig` function in Matlab), and does not need to resort to EM.

Suppose we are given dataset $\{x^{(i)}; i = 1, \ldots, m\}$ of attributes of m different types of automobiles, such as their maximum speed, turn radius, and so on. Lets $x^{(i)} \in \mathbb{R}^n$ for each i ($n \ll m$). But unknown to us, two different attributes—some x_i and x_j—respectively give a car's maximum speed measured in miles per hour, and the maximum speed measured in kilometers per hour. These two attributes are therefore almost linearly dependent, up to only small differences introduced by rounding off to the nearest mph or kph. Thus, the data really lies approximately on an $n - 1$ dimensional subspace. How can we automatically detect, and perhaps remove, this redundancy?

For a less contrived example, consider a dataset resulting from a survey of pilots for radio-controlled helicopters, where $x_1^{(i)}$ is a measure of the piloting skill of pilot i, and $x_2^{(i)}$ captures how much he/she enjoys flying. Because RC helicopters are very difficult to fly, only the most committed students, ones that truly enjoy flying, become good pilots. So, the two attributes x_1 and x_2 are strongly correlated. Indeed, we might posit that that the

data actually likes along some diagonal axis (the u_1 direction) capturing the intrinsic piloting "karma" of a person, with only a small amount of noise lying off this axis. (See figure.) How can we automatically compute this u_1 direction?



We will shortly develop the PCA algorithm. But prior to running PCA per se, typically we first pre-process the data to normalize its mean and variance, as follows:

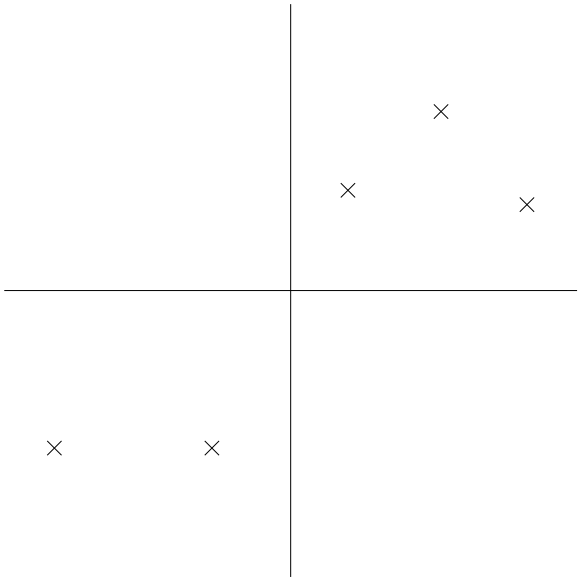
1. Let $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$.
2. Replace each $x^{(i)}$ with $x^{(i)} - \mu$. Zero out mean
3. Let $\sigma_j^2 = \frac{1}{m}\sum_i (x_j^{(i)})^2$ Normalize to unit variance
4. Replace each $x_j^{(i)}$ with $x_j^{(i)}/\sigma_j$.

Steps (1-2) zero out the mean of the data, and may be omitted for data known to have zero mean (for instance, time series corresponding to speech or other acoustic signals). Steps (3-4) rescale each coordinate to have unit variance, which ensures that different attributes are all treated on the same "scale." For instance, if x_1 was cars' maximum speed in mph (taking values in the high tens or low hundreds) and x_2 were the number of seats (taking values around 2-4), then this renormalization rescales the different attributes to make them more comparable. Steps (3-4) may be omitted if we had apriori knowledge that the different attributes are all on the same scale. One

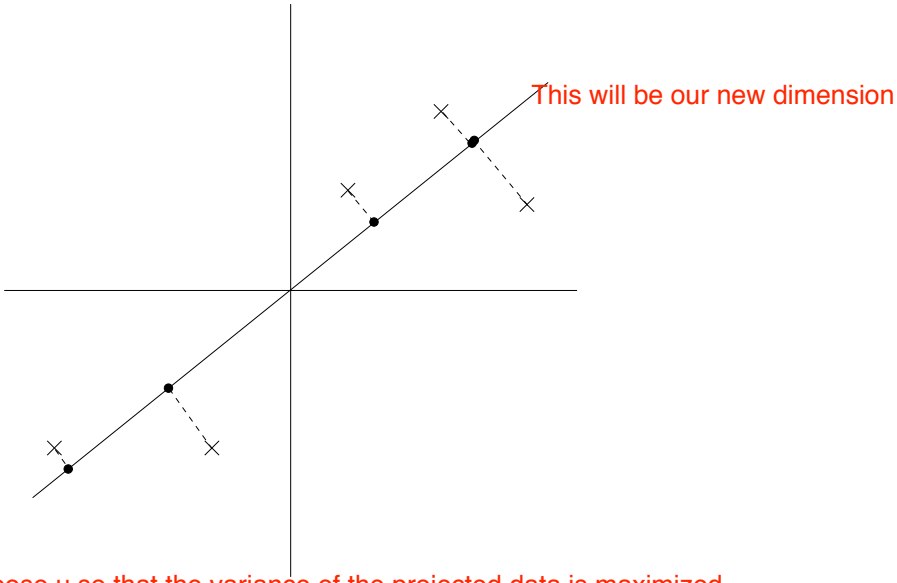
example of this is if each data point represented a grayscale image, and each $x_j^{(i)}$ took a value in $\{0, 1, \ldots, 255\}$ corresponding to the intensity value of pixel j in image i.

Now, having carried out the normalization, how do we compute the "major axis of variation" u—that is, the direction on which the data approximately lies? One way to pose this problem is as finding the unit vector u so that when the data is projected onto the direction corresponding to u, the variance of the projected data is maximized. Intuitively, the data starts off with some amount of variance/information in it. We would like to choose a direction u so that if we were to approximate the data as lying in the direction/subspace corresponding to u, as much as possible of this variance is still retained.

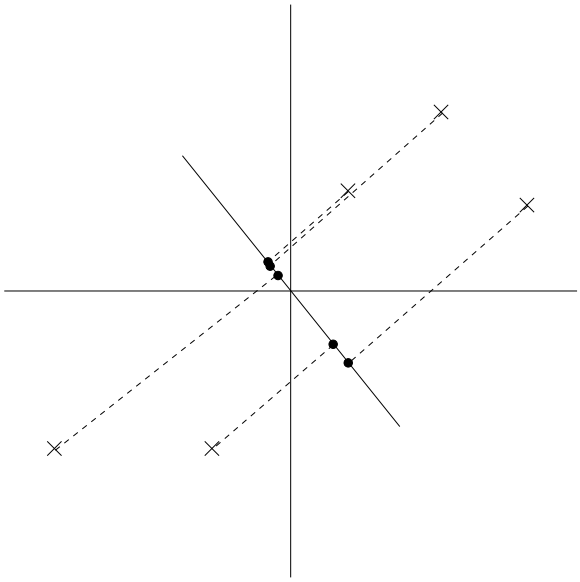
Consider the following dataset, on which we have already carried out the normalization steps:



Now, suppose we pick u to correspond the the direction shown in the figure below. The circles denote the projections of the original data onto this line.



We see that the projected data still has a fairly large variance, and the points tend to be far from zero. In contrast, suppose had instead picked the following direction:



Here, the projections have a significantly smaller variance, and are much closer to the origin.

We would like to automatically select the direction u corresponding to the first of the two figures shown above. To formalize this, note that given a

unit vector u and a point x, the length of the projection of x onto u is given by $x^T u$. I.e., if $x^{(i)}$ is a point in our dataset (one of the crosses in the plot), then its projection onto u (the corresponding circle in the figure) is distance $x^T u$ from the origin. Hence, to maximize the variance of the projections, we would like to choose a unit-length u so as to maximize:

max uT∑u
s.t uTu = 1
Lagrange:
L(u, λ) = uT∑u - λ(uTu - 1)
-> ΔuL(u, λ) = ∑u - λu = 0

$$\begin{aligned}\frac{1}{m}\sum_{i=1}^{m}(x^{(i)^T}u)^2 &= \frac{1}{m}\sum_{i=1}^{m}u^T x^{(i)}x^{(i)^T}u \\ &= u^T\left(\frac{1}{m}\sum_{i=1}^{m}x^{(i)}x^{(i)^T}\right)u.\end{aligned}$$

We easily recognize that the maximizing this subject to $||u||_2 = 1$ gives the principal eigenvector of $\Sigma = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}x^{(i)^T}$, which is just the empirical covariance matrix of the data (assuming it has zero mean).[1]

To summarize, we have found that if we wish to find a 1-dimensional subspace with with to approximate the data, we should choose u to be the principal eigenvector of Σ. More generally, if we wish to project our data into a k-dimensional subspace ($k < n$), we should choose $u_1, \ldots, u_k$ to be the top k eigenvectors of Σ. The u_i's now form a new, orthogonal basis for the data.[2]

Then, to represent $x^{(i)}$ in this basis, we need only compute the corresponding vector

$$y^{(i)} = \begin{bmatrix} u_1^T x^{(i)} \\ u_2^T x^{(i)} \\ \vdots \\ u_k^T x^{(i)} \end{bmatrix} \in \mathbb{R}^k.$$

Thus, whereas $x^{(i)} \in \mathbb{R}^n$, the vector $y^{(i)}$ now gives a lower, k-dimensional, approximation/representation for $x^{(i)}$. PCA is therefore also referred to as a **dimensionality reduction** algorithm. The vectors $u_1, \ldots, u_k$ are called the first k **principal components** of the data.

Remark. Although we have shown it formally only for the case of $k = 1$, using well-known properties of eigenvectors it is straightforward to show that

[1] If you haven't seen this before, try using the method of Lagrange multipliers to maximize $u^T\Sigma u$ subject to that $u^T u = 1$. You should be able to show that $\Sigma u = \lambda u$, for some λ, which implies u is an eigenvector of Σ, with eigenvalue λ.

[2] Because Σ is symmetric, the u_i's will (or always can be chosen to be) orthogonal to each other.

of all possible orthogonal bases $u_1, \ldots, u_k$, the one that we have chosen maximizes $\sum_i ||y^{(i)}||_2^2$. Thus, our choice of a basis preserves as much variability as possible in the original data.

In problem set 4, you will see that PCA can also be derived by picking the basis that minimizes the approximation error arising from projecting the data onto the k-dimensional subspace spanned by them.

PCA has many applications, our discussion with a small number of examples. First, compression—representing $x^{(i)}$'s with lower dimension $y^{(i)}$'s—is an obvious application. If we reduce high dimensional data to $k = 2$ or 3 dimensions, then we can also plot the $y^{(i)}$'s to visualize the data. For instance, if we were to reduce our automobiles data to 2 dimensions, then we can plot it (one point in our plot would correspond to one car type, say) to see what cars are similar to each other and what groups of cars may cluster together.

Another standard application is to preprocess a dataset to reduce its dimension before running a supervised learning learning algorithm with the $x^{(i)}$'s as inputs. Apart from computational benefits, reducing the data's dimension can also reduce the complexity of the hypothesis class considered and help avoid overfitting (e.g., linear classifiers over lower dimensional input spaces will have smaller VC dimension).

Lastly, as in our RC pilot example, we can also view PCA as a noise reduction algorithm. In our example it, estimates the intrinsic "piloting karma" from the noisy measures of piloting skill and enjoyment. In class, we also saw the application of this idea to face images, resulting in **eigenfaces** method. Here, each point $x^{(i)} \in \mathbb{R}^{100\times100}$ was a 10000 dimensional vector, with each coordinate corresponding to a pixel intensity value in a 100x100 image of a face. Using PCA, we represent each image $x^{(i)}$ with a much lower-dimensional $y^{(i)}$. In doing so, we hope that the principal components we found retain the interesting, systematic variations between faces that capture what a person really looks like, but not the "noise" in the images introduced by minor lighting variations, slightly different imaging conditions, and so on. We then measure distances between faces i and j by working in the reduced dimension, and computing $||y^{(i)} - y^{(j)}||_2$. This resulted in a surprisingly good face-matching and retrieval algorithm.

Visualization: High dimensional data to 3D (or even 2d) to visualize the structure of that data
Compression:
Learning: less dimension data -> faster but not sacrifice accuracy, less overfitting
Anomaly (outliner) detection algorithm
matching, distance calculation: example: find similar face (eigenface)