COMS 4721: Machine Learning for Data Science
Lecture 19, 4/6/2017

Prof. John Paisley

Department of Electrical Engineering
& Data Science Institute

Columbia University

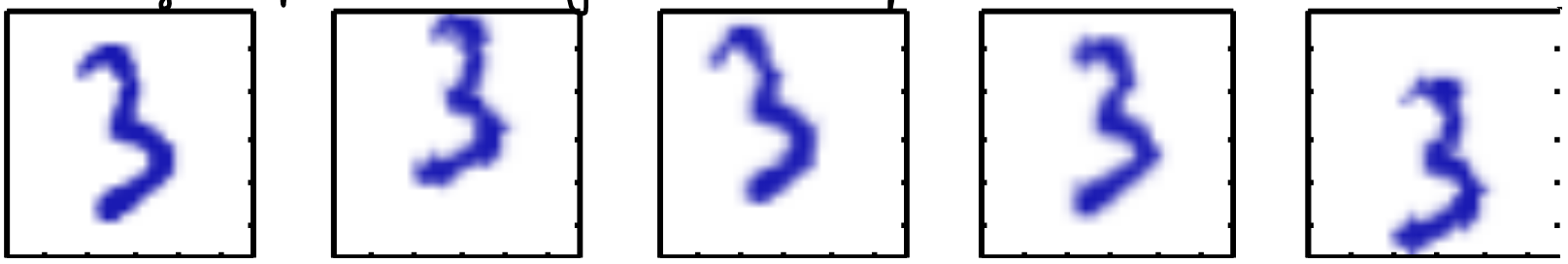
3rd and final matrix factorization technique from a few different perspectives, but they all relate to something called PCA.

Principal Component Analysis

often used for dimensionality reduction.

DIMENSIONALITY REDUCTION

We're given data $x_1, \ldots, x_n$, where $x \in \mathbb{R}^d$. This data is often high-dimensional, but the "information" doesn't use the full d dimensions.



For example, we could represent the above images with three numbers since they have three degrees of freedom. Two for shifts and a third for rotation.

Principal component analysis can be thought of as a way of automatically mapping data x_i into some new low-dimensional coordinate system.

- It capture most of the information in the data in a few dimensions
- Extensions allow us to handle missing data, and "unwrap" the data.

* Meaning that vectors in the lower dimensional space all kind of have the same relationship to each other that they had in the higher dimensional space.

And so learning an algorithm on data from the lower dimensional space shouldn't give worse results than what we would get from learning on a higher dimensional space.

↓

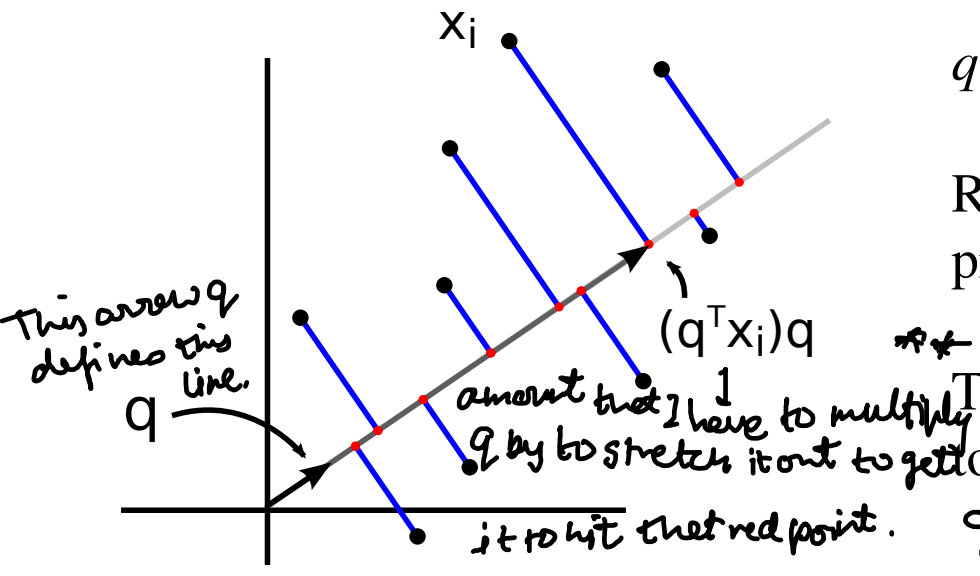
how is this possible?

** All we need to learn is a shift and a rotation;
, i.e., 2 shifts and one rotation for these 3 images.
So where is the center, that's a 2-dim vector for the center and then the rotation is a 3rd dimension.

And so we really 3 no.s for this dataset to capture everything going into it.
And also the base pattern of 3 that's going to be reappearing in each of these images

PRINCIPAL COMPONENT ANALYSIS

Example: How can we approximate this data using a unit-length vector q?



q is a unit-length vector, $q^T q = 1$.

Red dot: The length, $q^T x_i$, to the axis after projecting x onto the line defined by q.

The vector $(q^T x_i)q$ takes q and stretches it to the corresponding red dot.

So what's a good q? How about minimizing the squared approximation error,

$$q = \arg\min_q \sum_{i=1}^{n} \|x_i - qq^T x_i\|^2 \quad \text{subject to} \quad q^T q = 1$$

$qq^T x_i = (q^T x_i)q$: The approximation of x_i by stretching q to the "red dot."

* PCA essentially does this problem.

Is take this vector Q, make it unit length, and now represent each of these data points by dropping a line perpendicular to the line constructed by the line constructed by the direction of Q.

** ↳ • Now we wanna take each of these points and represent it by the amount that we have to stretch Q to get to the red dot by dropping it that line.

• What does this dot product of this unit vector q and x_i represent? That represents the length from this origin to the red dot here.

Dimensionality reduction:

So I've done very trivial and not interesting example of dimensionality reduction by taking a two dimensional vector and reducing it now to one dimension.
And so for each of these points, I need to keep only one number.
I need to keep the value of $q^T x_i$.

And then also somewhere, I need to keep the vector Q, potentially if I wanna construct this space.

linear regression
minimizing the sum of squared errors
dropping along the o/p dimension.
PCA → dropping 1 to q

*** (x_i)

We're taking this two dimensional vector and approximating it with this two dimensional vector. $(qq^T x_i)$ ← same dimensional space as x_i
But all of those approximations have to fall along the same one dimensional subspace.

PCA : THE FIRST PRINCIPAL COMPONENT

This is related to the problem of finding the largest eigenvalue,

$$q = \arg\min_q \sum_{i=1}^{n} \|x_i - qq^T x_i\|^2 \quad \text{s.t.} \quad q^T q = 1$$

throw away doesn't include q.

vector transpose multiplied by itself.

how?

$$= \arg\min_q \sum_{i=1}^{n} x_i^T x_i - q^T \underbrace{\left(\sum_{i=1}^{n} x_i x_i^T\right)}_{= XX^T} q$$

Constructed by taking each of the our data points x_i and putting it along a column.

$d \times n$ (each column is a data point).

We've defined $X = [x_1, \ldots, x_n]$. Since the first term doesn't depend on q and we have a negative sign in front of the second term, equivalently we solve

$$q = \arg\max_q \; q^T (XX^T) q \qquad \text{subject to} \quad q^T q = 1$$

we want to find the vector q that maximises this product

This is the eigendecomposition problem:

- q is the first eigenvector of XX^T → $d \times d$ and symmetric
- $\lambda = q^T (XX^T) q$ is the first eigenvalue [the value of this product evaluated at the optimal q.]

PCA: General

That's how we would represent every data point with 1 vector. Meaning we take that original data and map it down to 1d space. So we can represent every data point now with 1 no. *

The general form of PCA considers K eigenvectors, ** find k vectors in order to this (approximate x_i by linear combination of k different vectors.)

$$q = \arg\min_{q} \sum_{i=1}^{n} \| x_i - \underbrace{\sum_{k=1}^{K} (x_i^T q_k) q_k}_{\text{approximates } x} \|^2 \quad *** \quad \text{s.t. } q_k^T q_{k'} = \begin{cases} 1, & k = k' \\ 0, & k \neq k' \end{cases}$$

(unit length)

(orthogonal to each other.)

K different vectors for q

2 constraints

how just adding them

$$= \arg\min_{q} \sum_{i=1}^{n} x_i^T x_i - \sum_{k=1}^{K} q_k^T \underbrace{\left(\sum_{i=1}^{n} x_i x_i^T \right)}_{= XX^T} q_k$$

same as before

same as before but now we have a sum over k different vectors

The vectors in $Q = [q_1, \ldots, q_K]$ give us a K dimensional subspace with which to represent the data:

more. this over k vectors q subject to 2 constraints

Reduced the no of dimensions to k.

$$x_{\text{proj}} = \begin{bmatrix} q_1^T x \\ \vdots \\ q_K^T x \end{bmatrix}, \qquad x \approx \sum_{k=1}^{K} (q_k^T x) q_k = Q x_{\text{proj}}$$

how projection relates to original value

amount we have to stretch to approximate it.

sum over all vectors.

The eigenvectors of (XX^T) can be learned using built-in software.

OG

* It's the amount that we have to multiply q by to get it as close to the original date as possible.

**
However, PCA in general doesn't just consider one eigenvector and one eigenvalue.
Meaning that it doesn't map the data to one-dimensional space.
It often will consider K eigenvectors where k is arbitrary

*** We take each eigen vector q_K and we stretch it by the dot product of the point x_i with the eigen vector. So this dot product $(x_i^T q_K)$ is the amount that we have to stretch q_K by to get that individual vector as close to x_i as possible.

And now because each of these qs are orthogonal to each other,
we treated as K different approximations and then some each of those up.
So we're trying to approximate the data with a K dimensional vector,
which is the k dimensions that we get from the K different dot products here.

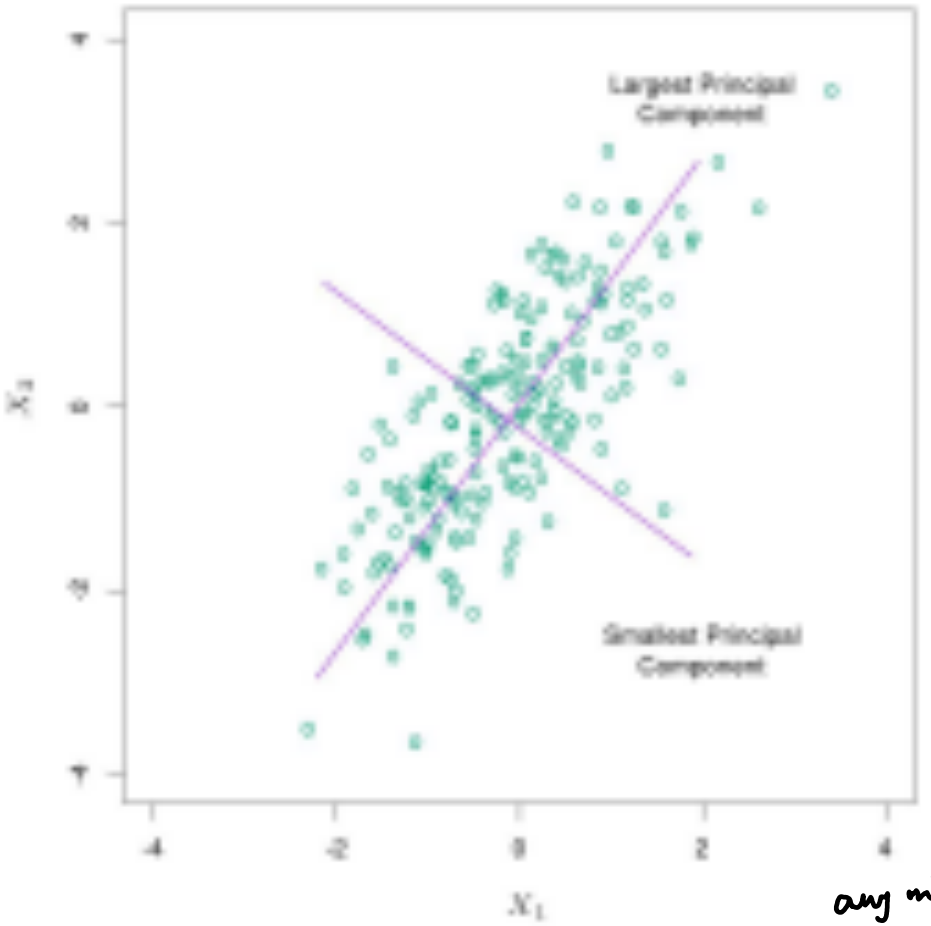
oo what I gain from moving in 1 dimension has nothing to do with other dimensions.

- So now, we've taken the original data and reduce the number of dimensions to K where the projection that we get of the data from the higher dimension is simply represented by taking the dot products of the data with each of these vectors q.

- In practice, when we do this, what we'll often do is we'll take this matrix X,
  Take a dot product.
  And then call some function to find the K largest eigenvectors,
  as well as their associated eigenvalues.
  The kth eigenvalue will be the value of this product for q_K.

↳ $q_K^T \left(\sum_{i=1}^{n} x_i x_i^T\right) q_K$

EIGENVALUES, EIGENVECTORS AND THE SVD



An equivalent formulation of the problem is to find (λ, q) such that

$$(XX^T)q = \lambda q$$ *

Since (XX^T) is a PSD matrix, **
there are $r \leq \min\{d, n\}$ pairs,

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_r > 0,$$

$$q_k^T q_k = 1, \quad q_k^T q_{k'} = 0$$

Why is (XX^T) PSD? Using the SVD, $X = USV^T$, we have that

any matrix → columns of U are orthonormal; rows of V^T are also " → why?

non-negative diagonal values

$$(XX^T) = US^2U^T \quad \Rightarrow \quad Q = U, \quad \lambda_i = (S^2)_{ii} \geq 0$$

↳ i^{th} eigenvalue [squared, can get to > than 0.]

why? → Preprocessing: Usually we first subtract off the mean of each dimension of x. ***

eigenvectors along its columns

* This matrix (XX^T) here multiplied by q, it doesn't change it's direction. It only stretches it by value λ.

**

These eigenvalues being positive is a result of this being a positive semidefinite matrix and where we can get are these different eigenvectors where each one is unit length, and are orthogonal to each other.a

° We can find the eigenvectors and the eigenvalues of this matrix product (XX^T) by just taking SVD of the original matrix.

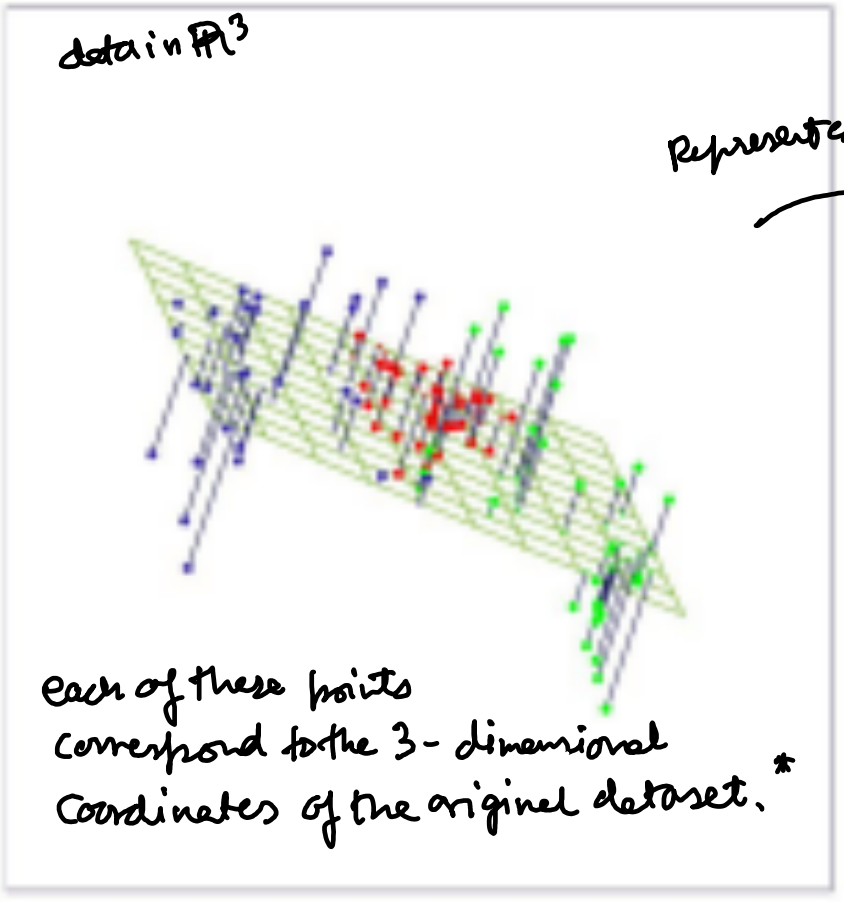
*** Rows of X have their mean subtracted off. So we center the data, so we center the data so that all of the dimensions in our dataset have mean 0.
↳ Because we want the data to be centered along the original axes. So that we can represent the data in its Ist principal component and its IInd principal component. So that comes from the fact of these PCs having to pass through the origin

If you imagine this dataset shifted very far away from the origin, it would be much harder to find such a nice representation, if this were rotated a little bit.

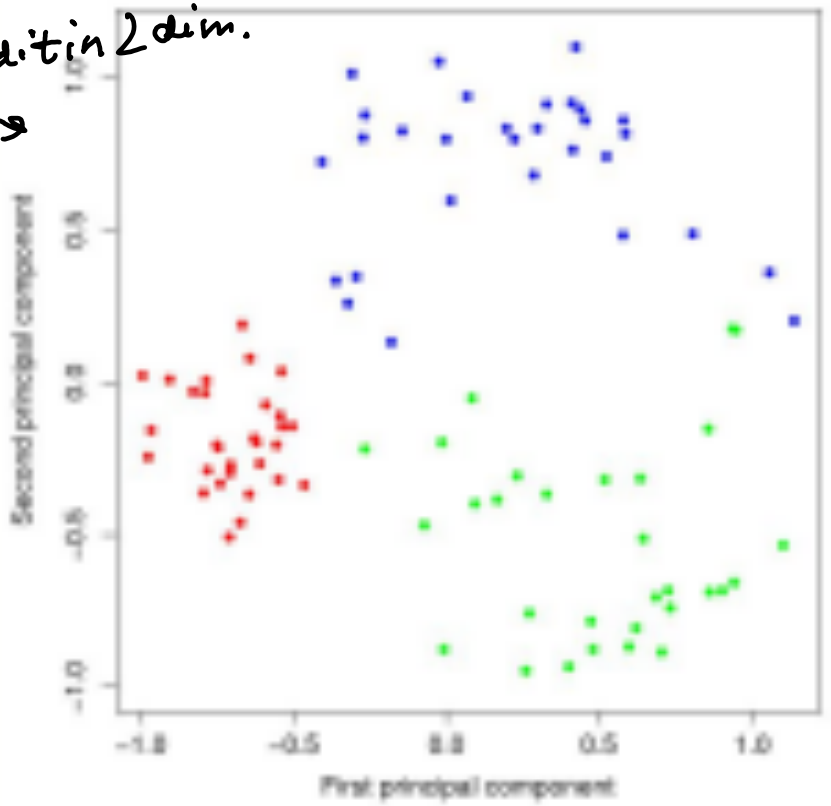
why we do this?

[why?

PCA: EXAMPLE OF PROJECTING FROM $\mathbb{R}^3$ TO $\mathbb{R}^2$ using eigen decomposition



Represented in 2 dim.



For this data, most information (structure in the data) can be captured in $\mathbb{R}^2$.**

(left) The original data in $\mathbb{R}^3$. The hyperplane is defined by q_1 and q_2.

(right) The new coordinates for the data: $x_i \rightarrow x_i^{proj} = \begin{bmatrix} x_i^T q_1 \\ x_i^T q_2 \end{bmatrix}$.

*

When we try to find an eigen decomposition to project it into two dimensions, it's like finding a plane that slices through the data with a little error as possible.

So, we take a plane.We drop a line to the hyperplane perpendicular to it.
That's our error and we sum that squared error.That's the thing that we're trying to minimize.

**

Notice that if we had just simply taken two of the original dimensions and thrown away the third, we couldn't have gotten as nice of a representation of the dataset. So it's like trying to shift and rotate the data around, so that we can just throw away some of the dimensions but keep the remaining, but where we align the information as much as possible along the dimensions that we're going to keep.

why does ← align the information mean?

why are we doing it?

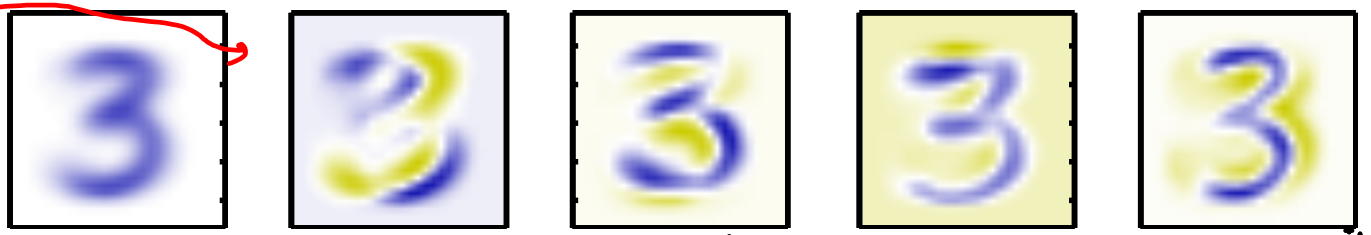
EXAMPLE: DIGITS

we don't think 256 no. save necessarily required to keep all of the information about each of these 3s. So we project them to a lower dimension using eigen value decomposition, PCA. ← why?

Data: 16×16 images of handwritten 3's (as vectors in $\mathbb{R}^{256}$)

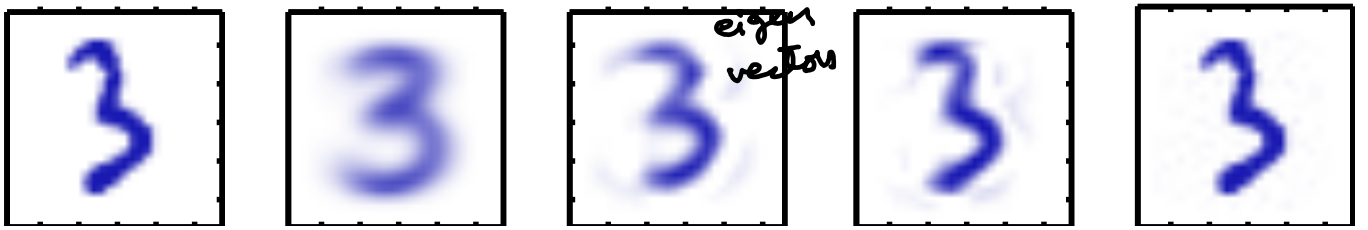
what's yellow? what's blue?

2nd eigenvector — IInd and IIIrd capture decreasing amounts of whatever information is left, not by previous eigenvectors.



Each eigen vector is 256 dimensions. We fold back into the 16x16 matrix and show it as an image.

Above: The first four eigenvectors q and their eigenvalues λ.



0 mean vector & 9 eigen vectors

We're almost identical and the remaining ones will pick up on this noise here. ?

Above: Reconstructing a 3 using the first $M - 1$ eigenvectors plus the mean, and approximation

$$x \approx \text{mean} + \sum_{k=1}^{M-1} (x^T q_k) q_k$$

So this image is a 10 dimensional or really a 9 dimensional representation of the 3. ← how? what is the 9d vector?

*

(1) So if we took this image and dot multiply that with any other image here and summed it up, it would equal 0.

(2) And if we took each of these images and dot multiplied it by itself and summed it up, it would equal 1.

Now, we have a 16x16 image, and want to project it into a few dimensions:

1. First, we subtract off the mean from it ← doesn't that change the intensity & result image be not seen as original.
2. We take the remaining subtracted image and dot multiply it by each of these and sum up the results. (4 eigenvectors)

** We can get now 4 no.s which is the result of that corresponding dot products of each of these images with the image that just came in.

And now if we want to reconstruct the image, we take the dot product of our means subtracted image, we multiply it by the eigen vector and now that's the approximation like we have discussed previously. of the original image in that dimension only and we sum them up!

o If we wanna represent that only in one vector, we would represent it in the mean ← why? use 1st eigenvector?

oo We are representing this image X as being approximately equal to the mean plus the dot prodcut of X with each eigenvector and we scale the eigenvector and sum it up. This is the approximation we are making.

PROBABILISTIC PCA

We take PCA, and view it as a probability model. And learn the parameters using a probabilistic representation.

PCA AND THE SVD (connection recall)

used the original matrix rather than the dot product.

We've discussed how any matrix X has a singular value decomposition,

$$X = USV^T, \quad U^TU = I, \quad V^TV = I$$

and S is a diagonal matrix with non-negative entries.

[Assuming X has more columns than rows. If it had more rows than columns then some of these transposes would be switched.]

Therefore, (which we want to learn eigendecomposition of)

$$XX^T = US^2U^T \quad \Leftrightarrow \quad (XX^T)U = US^2$$

$(XX^T)q = \lambda q$ (Representation from prev. slide)

U is a matrix of eigenvectors, and S^2 is a diagonal matrix of eigenvalues.

[eigenvectors along the columns of U.]

[λ along the diagonal elements of S^2]

The left singular vectors (of X) (U) correspond to the principal components of this matrix.
The square of the singular values correspond to the eigenvalues of this matrix.

A MODELING APPROACH TO PCA

A model that uses the EM algorithm, so we have to define a generative process and then do EM to run a point estimate of it.

Using the SVD perspective of PCA, we can also derive a probabilistic model for the problem and use the EM algorithm to learn it.

This model will have the advantages of: (that PCA can't handle easily, directly).

- Handling the problem of missing data
- Allowing us to learn additional parameters such as noise
- Provide a framework that could be extended to more complex models
- Gives distributions used to characterize uncertainty in predictions
- etc. ↑ Reasons why we might want to choose a probabilistic model.

PROBABILISTIC PCA (can also view as doing probabilistic SVD because of the connection.)

In effect, this is a new matrix factorization model.

- With the SVD, we had $X = USV^T$. (eigenvectors, eigenvalues) [What is the significance of V here?] [So we can directly learn this sort of factorization without taking the product of X with itself.]
- We now approximate $X \approx WZ$, where (what's we're going to do)
  - W is a $d \times K$ matrix. In different settings this is called a "factor loadings" matrix, or a "dictionary." It's like the eigenvectors, but no orthonormality.
  - The ith column of Z is called $z_i \in \mathbb{R}^K$. Think of it as a low-dimensional representation of x_i. [If the i^{th} column in X corresponds to the i^{th} data point, the i^{th} column of Z corresponds to the i^{th} data point except projected into a lower K dimensional space.]

Remove all constraints. Depending upon the problem it is being applied to. corresponding value of

The generative process of Probabilistic PCA is

$$x_i \sim N(Wz_i, \sigma^2 I), \qquad z_i \sim N(0, I).$$

(W: $D \times K$; noise; z_i: K dim vector) [Multi-variate Gaussian]

↓ the vector z_i is now also a random variable from a Gaussian with zero mean and identity covariance matrix.

? why this assumption?

In this case, we don't know W or any of the z_i.

Now, we want to learn both of these things.

(We are going to do this with MLE.)

* d - dimensionality of data
  k - no. of eigenvalues / eigenvectors we want to learn

** We're also going to relax that constraint that w has to be orthonormal.
So that's a crucial difference between prob. PCA and regular PCA.
The columns don't have to unit length and they don't have to be orthogonal to each other.

*** Difference from matrix factorization neither of these have to be non-negative
We want to learn a low rank factorization.

$$X = W Z$$

1000 dimensions × 10,000 observations = 10,000 × 20 × 20 × 10,000

Don't watch!

- The error in the low-rank approximation is Gaussian which makes sense since PCA did a squared also (which is like Gaussian.)

what makes it probabilistic)

Expectation of $x_i \sim N(W z_i, \sigma^2 I)$ $\quad z_i \sim N(0, I)$ as a prior

$[\ ]$ tall & skinny

Maximum likelihood

Our goal is to find the maximum likelihood solution of the matrix W under the marginal distribution, i.e., with the z_i vectors integrated out, ← why?

(integrate out: impact of z_is)

Then do a point estimate that maximizes the marginal likelihood with respect to matrix W.

$$W_{\text{ML}} = \arg\max_W \ln p(x_1, \ldots, x_n | W) = \arg\max_W \sum_{i=1}^{n} \ln p(x_i | W). \quad \text{①}$$

all of the data vectors. product → sum independence assumption ?

This is intractable because $p(x_i|W) = N(x_i|0, \sigma^2 I + WW^T)$, ← because z_i has a 0 mean

covariance matrix

density:

$$N(x_i|0, \sigma^2 I + WW^T) = \frac{1}{(2\pi)^{\frac{d}{2}} |\sigma^2 I + WW^T|^{\frac{1}{2}}} e^{-\frac{1}{2} x^T (\sigma^2 I + WW^T)^{-1} x}$$

We can set up an EM algorithm that uses the vectors $z_1, \ldots, z_n$.

even though we couldn't take the derivative of this with respect to W & optimize it. We can still evaluate it at each value of W. We can use this eqn. ① where we take the log of this ① over each.

reintroduce these vectors as missing part of our model. and then do EM on that.

if we take the log of it, we can't do this and solve for W analytically.

EM FOR PROBABILISTIC PCA

Setup set up the EM equality

The marginal log likelihood can be expressed using EM as

marginal likelihood which is hard to optimize over W

introduced a q distribution over each of these vectors z_i

joint likelihood of x_i & z_i

$$\sum_{i=1}^{n} \ln \int p(x_i, z_i|W)\, dz_i = \sum_{i=1}^{n} \int q(z_i) \ln \frac{p(x_i, z_i|W)}{q(z_i)}\, dz_i \quad \leftarrow \quad \mathcal{L}$$

Marginal likelihood we want to maximize over W as a integral z..z is integrated out.

$$+ \sum_{i=1}^{n} \int q(z_i) \ln \frac{q(z_i)}{p(z_i|x_i, W)}\, dz_i \quad \leftarrow \quad \mathrm{KL}$$

divergence of q distribution on z_i and conditional posterior distribution of z_i given the matrix W.

[Now if we want to use the RHS in order to maximize the LHS over W.]

EM Algorithm: Remember that EM has two iterated steps

E-step

1. Set $q(z_i) = p(z_i|x_i, W)$ for each i (making $\mathrm{KL} = 0$) and calculate $\mathcal{L}$

M-step

2. Maximize $\mathcal{L}$ with respect to W

Again, for this to work well we need that

in closed form.

- we can calculate the conditional posterior distribution $p(z_i|x_i, W)$, and
- maximizing $\mathcal{L}$ is easy, i.e., we update W using a simple equation

THE ALGORITHM

EM for Probabilistic PCA

Given: Data $x_{1:n}$, $x_i \in \mathbb{R}^d$ and model $x_i \sim N(Wz_i, \sigma^2 I)$, $z_i \sim N(0, I)$, $z \in \mathbb{R}^K$

Output: Point estimate of W and posterior distribution on each z_i

E-Step: Set each $q(z_i) = p(z_i|x_i, W) = N(z_i|\mu_i, \Sigma_i)$ where

$$\Sigma_i = (I + W^T W/\sigma^2)^{-1}, \quad \mu_i = \Sigma_i W^T x_i/\sigma^2$$

M-Step: Update W by maximizing the objective $\mathcal{L}$ from the E-step

$$W = \left[\sum_{i=1}^{n} x_i \mu_i^T\right] \left[\sigma^2 I + \sum_{i=1}^{n} (\mu_i \mu_i^T + \Sigma_i)\right]^{-1}$$

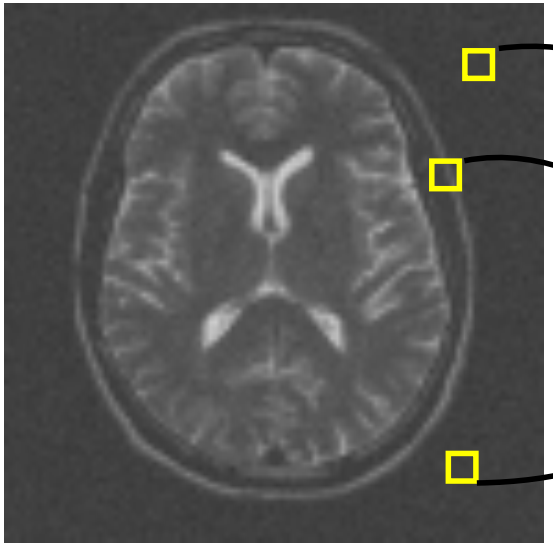
Iterate E and M steps until increase in $\sum_{i=1}^{n} \ln p(x_i|W)$ is "small."

Comment:

- The probabilistic framework gives a way to learn K and σ^2 as well.

* So we don't just get a low embedding for each point x_i in the q of z_i, we also get some distribution on that embedding.

EXAMPLE: IMAGE PROCESSING problem of denoising an image.



■ = 8 x 8 patch is arbitrary. It could be bigger/smaller

X

data matrix, e.g., 64 x 262,144 columns

every

For image problems such as denoising or inpainting (missing data)

- Extract overlapping patches (e.g., 8×8) and vectorize to construct X
- Model with a factor model such as Probabilistic PCA
- Approximate $x_i \approx W\mu_i$, where μ_i is the posterior mean of z_i
- Reconstruct the image by replacing x_i with $W\mu_i$ (and averaging)

Factorise using

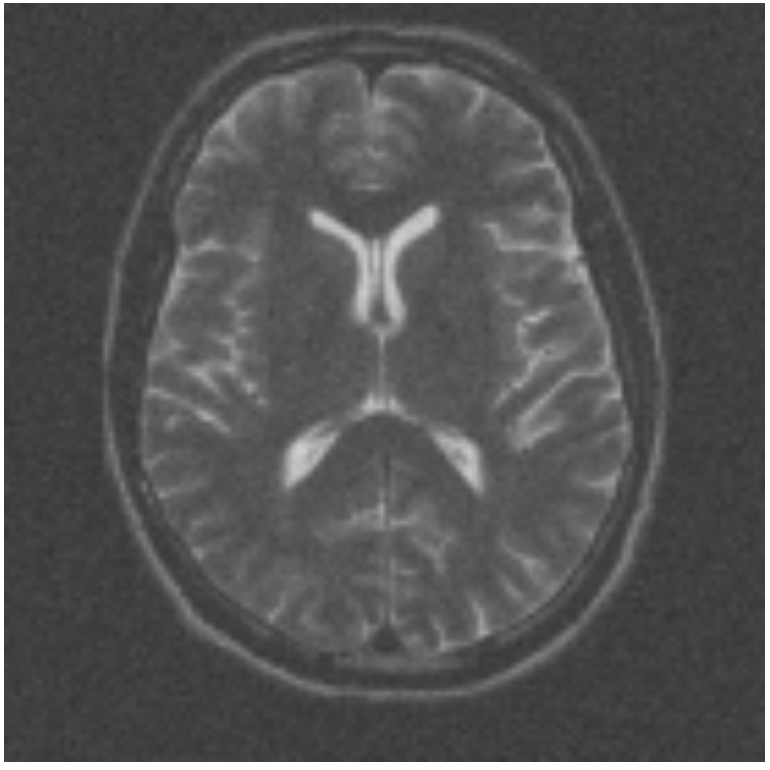
conditional posterior mean

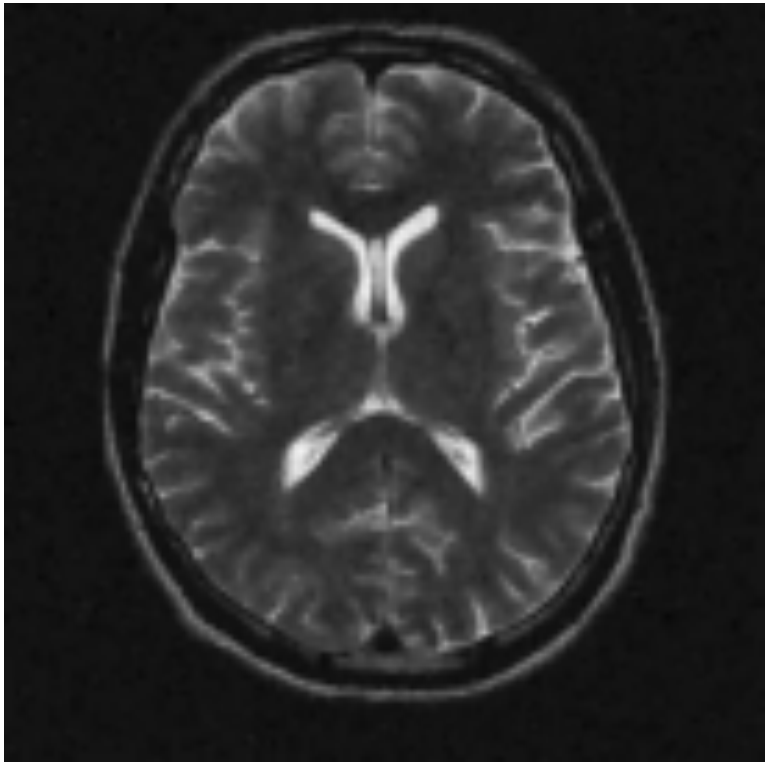
how?

To be lazy, just plug in the mean & forget all the uncertainty we had.

Noise is gone is in noise term of the generative model.

EXAMPLE: DENOISING





Noisy image on left, denoised image on right. The noise variance parameter σ^2 was learned for this example.

EXAMPLE: MISSING DATA

by throwing away 80%. i.e., you didn't throw away any information because all information is in a low-dimensional image.



Extract a 8x8x3 cube and we filled in the missing data using EM.

Another somewhat extreme example:

- Image is 480×320×3 (RGB dimension)
- Throw away 80% of value at random
- (left) Missing data, (middle) reconstruction, (right) original image

KERNEL PCA

KERNEL PCA

We've seen how we can take an algorithm that uses dot products, $x^T x$, and generalize with a nonlinear kernel. This generalization can be made to PCA.

replace dot products with ↑

Recall: With PCA we find the eigenvectors of the matrix $\sum_{i=1}^{n} x_i x_i^T = XX^T$

X = each x_i along its column & transpose

find the eigenvectors of this matrix.

outer product

to higher dimensions

- Let $\phi(x)$ be a feature mapping from $\mathbb{R}^d$ to $\mathbb{R}^D$, where $D \gg d$

Projecting to much higher dimensions before projecting to much lower dimension can do even more for us.

- We want to solve the eigendecomposition

$$\left[\sum_{i=1}^{n} \phi(x_i)\phi(x_i)^T\right] q_k = \lambda_k q_k$$

$(XX^T)q_k = \lambda_k q_k$

Find the vector q and λ for matrix from this higher dimensional projection.

without having to work in the higher dimensional space.

(Even in Gaussian kernels, this is an ∞-dim space.)

- That is, how can we do PCA without explicitly using $\phi(\cdot)$ and q?

KERNEL PCA

Notice that we can reorganize the operations of the eigendecomposition

i → indexing data point
k → eigenvalue, eigenvector pair.

$$\sum_{i=1}^{n} \phi(x_i) \underbrace{\left(\phi(x_i)^T q_k\right)/\lambda_k}_{= a_{ki}} = q_k$$

Multiply higher dimensional projection of x_i with k^{th} eigenvector, divided by k^{th} eigenvalue.

λ_k on the left side by dividing it.

That is, the eigenvector $q_k = \sum_{i=1}^{n} a_{ki}\phi(x_i)$ for some vector $a_k \in \mathbb{R}^n$.

Represent each eigenvector as a linear combination of data projected to the higher dimensional space.

The trick is that instead of learning q_k, we'll learn a_k.

k^{th} eigen vector
↳ vector in $\mathbb{R}^n$ n → no. of data points
Trick: Instead of learning this potentially infinite dimensional vector, we're going to learn an n-dimensional vector corresponding to that k^{th} eigenvector.

Plug this equation for q_k back into the first equation:

(Why are you plugging it back into the same equation?)

dots products! (λ_k back)

$$\underline{\Phi}^T \sum_{i=1}^{N} \phi(x_i) \sum_{j=1}^{n} a_{kj} \underbrace{\phi(x_i)^T \phi(x_j)}_{= K(x_i, x_j)} = \underline{\Phi}^T \lambda_k \sum_{i=1}^{n} a_{ki}\phi(x_i)$$

replace kernel fn!

and multiply both sides by $\phi(x_l)^T$ for each $l \in \{1, \ldots, n\}$.

$\underline{\Phi} = [\phi(x_1)\ \phi(x_2) \ldots \phi(x_N)]$

projection of the l^{th} data point, for each of the n data points.

KERNEL PCA

(will simplify our problem)

When we multiply the following by $\phi(x_l)^T$ for $l = 1 \ldots, n$:

$$\Phi^T \sum_{i=1}^{N} \phi(x_i) \sum_{j=1}^{n} a_{kj} \underbrace{\phi(x_i)^T \phi(x_j)}_{= K(x_i, x_j)} = \Phi^T \lambda_k \sum_{i=1}^{n} a_{ki} \phi(x_i)$$

$\Phi = [\phi(x_1) \ldots \phi(x_N)]$

$\Phi = [\phi(x_1) \ldots \phi(x_n)]$

we get a new set of linear equations

multiply both sides by K^{-1} (PSD matrix)

how?

$$K^2 a_k = \lambda_k K a_k \iff K a_k = \lambda_k a_k$$

Kernel

K_{ij} → data point x_i & x_j

where K is the $n \times n$ *kernel matrix* constructed on the data.

a_k - eigenvector associated with the kernel matrix constructed among data

Because K is guaranteed to be PSD because it is a matrix of dot-products, the LHS and RHS above share a solution for (λ_k, a_k).

Now perform "regular" PCA, but on the kernel matrix K instead of the data matrix XX^T. We summarize the algorithm on the following slide.

Originally, I could have mapped the data into such a high dimensional space can infinite dimensional space

KERNEL PCA ALGORITHM

Kernel PCA

define b/w all points.

Given: Data $x_1, \ldots, x_n$, $x \in \mathbb{R}^d$, and a kernel function $K(x_i, x_j)$.

Construct: The kernel matrix on the data, e.g., $K_{ij} = b \exp\left\{-\frac{\|x_i - x_j\|^2}{c}\right\}$.

Solve: The eigendecomposition

$$K\boldsymbol{a}_k = \lambda_k \boldsymbol{a}_k$$

dimensionality we want to project the data into.

for the first $r \ll n$ eigenvector/eigenvalue pairs $(\lambda_1, \boldsymbol{a}_1), \ldots, (\lambda_r, \boldsymbol{a}_r)$. of kernel K

.

Output: A new coordinate system for x_i by (implicitly) mapping $\phi(x_i)$ and then projecting $q_k^T \phi(x_i)$

back to the lower dimensional space. (r-dimensional space.)

x_i $\xrightarrow{\text{projection}}$ $\begin{bmatrix} \lambda_1 a_{1i} \\ \vdots \\ \lambda_r a_{ri} \end{bmatrix}$

& learnt r different eigen vectors these

For the k^{th} dimension of our projection, we learnt the k^{th} eigenvector or eigenvalue.

For the i^{th} datapoint we look at the i^{th} dimension of k^{th} eigenvector and multiply it.

where a_{ki} is the ith dimension of the kth eigenvector $\boldsymbol{a}_k$.

KERNEL PCA AND NEW DATA

Q: How do we handle new data, x_0? Before, we could take the eigenvectors q_k and project $x_0^T q_k$, but $\boldsymbol{a}_k$ is different here.

A: Recall the relationship of $\boldsymbol{a}_k$ to q_k in kernel PCA is

$$q_k = \sum_{i=1}^{n} a_{ki} \phi(x_i).$$

We used the "kernel trick" to avoid working with or even defining $\phi(x_i)$.

As with regular PCA, after mapping x_0 we want to project onto eigenvectors

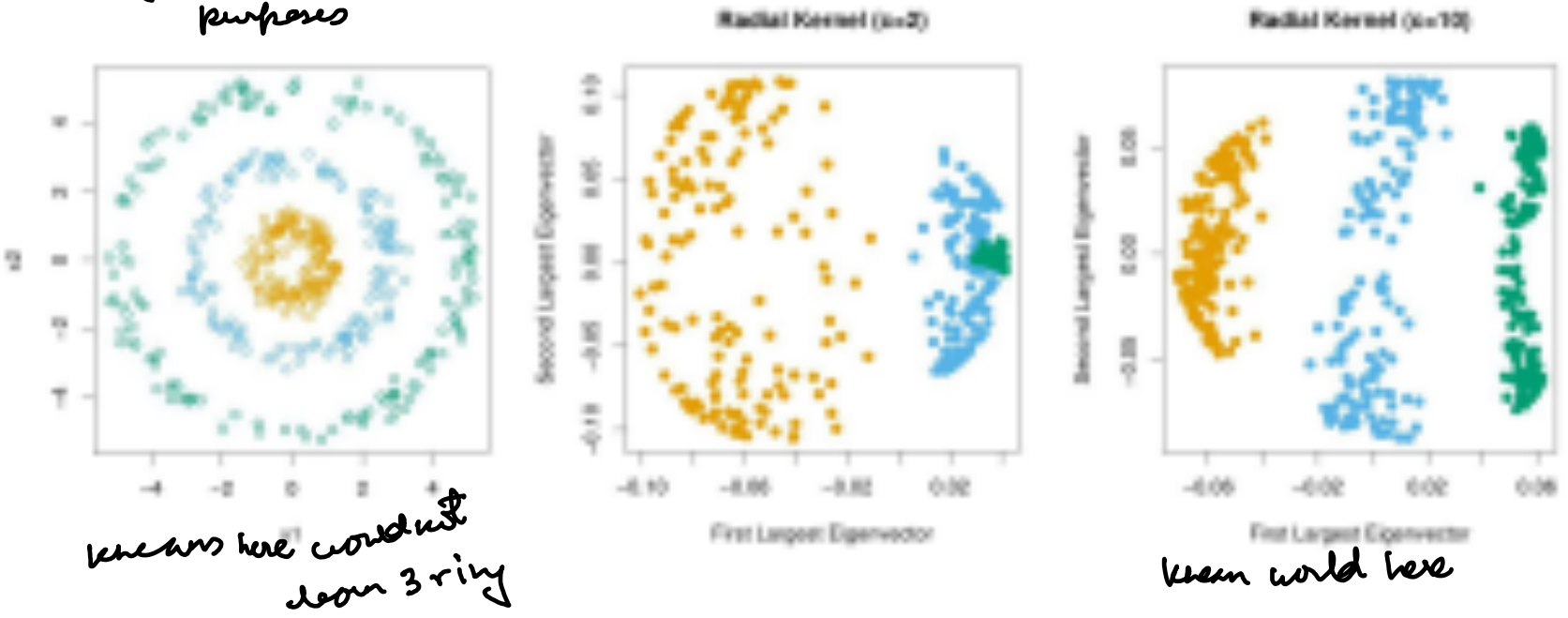
$$x_0 \xrightarrow{\text{projection}} \begin{bmatrix} \phi(x_0)^T q_1 \\ \vdots \\ \phi(x_0)^T q_r \end{bmatrix}$$

Plugging in for q_k: $\phi(x_0)^T q_k = \sum_{i=1}^{n} a_{ki} \phi(x_0)^T \phi(x_i) = \sum_{i=1}^{n} a_{ki} K(x_0, x_i)$.

* So I evaluate the kernel between a new incoming point with all the points in my dataset. I multiply the kernel by the eigenvector I got from the matrix, (the original matrix K), sum it up.

That's my projection.

EXAMPLE RESULTS



An example of kernel PCA using the Gaussian kernel.

(left) Original data, colored for reference (but may be classes)

(middle) New coordinates using kernel width $c = 2$

(right) New coordinates using kernel width $c = 10$

Terminology: What we are doing is closely related to "spectral clustering" and can be considered an instance of "manifold learning."

Take each of these data points and calculating the kernel matrix. The pairwise kernel matrix using a Gaussian kernel and then using that $n \times n$ kernel matrix, projecting onto its I^{st} 2 eigenvectors and eigenvalues.