Pattern Recognition

Lecture 5

PCA

Statistical Foundations

Thomas M. Breuel

Ilya... exercise this Wednesday

as before...

- please remember to hand in your exercise sheets
- you can now run Sage on tux1 etc.
- if worksheets are not available from home page, there may be PDF files at the bottom of the page

PCA (Principal Component Analysis)

input data

- let $\{x_1, \ldots, x_N\}$ be samples from some pattern recognition problem
- let Σ be the covariance matrix (empirical or true)
- think of the x_i as coming from a normal density (even if they don't)

recall that

- a normal density with cov Σ can be derived by rotation and scaling from a spherical normal density
- $x \to S \cdot R \cdot x$ then $\Sigma^{-1} = R^T \cdot S^T \cdot S \cdot R$ (S diag scale, R rotation)
- assume that S is ordered in decreasing size

PCA = ...

- undo the rotation matrix R
- truncate the diagonal matrix (perform a projection) somewhere

PCA

input

- list of vectors $x_1, \ldots, x_N$

algorithm

- compute covariance matrix Σ
- compute the eigenvalues and eigenvectors
- sort eigenvalues largest to smallest
- the eigenvectors determine rotation matrix R
- the eigenvalues determine S^2
- take the input vectors x and transform them into $R^{-1}x$
- compute the projection $y = P_k R^{-1} x$
- perform classification with the $y_1, \ldots, y_N$

PCA — Why?

- the y have lower dimensionality than x
- the y are the best lsq approximation to x of dimension k
- the components of y are uncorrelated
- the variance of component j of y is S_{jj}
- the covariance matrix of y is S (it is diagonal)
- y contains the “non-noise portions” of x
- for any x, x' we have $d_E(y, y') \approx d_x(x, x')$

PCA — How to Compute?

obvious algorithm

- compute μ for the input data
- transform the data by $x_i \rightarrow x_i - \mu$
- compute data matrix X with rows equal to x_i
- compute $\hat{\Sigma} = X^T X / N$
- compute eigenvectors & eigenvalues, or compute svd

note

- 30 x 30 input image = 900 x 900 = 810000 element cov matrix

PCA — How to Compute?

iterative algorithm

- start with any unit vector v
- compute $v' = \Sigma v$
- normalize $v'' = v'/|v'|$
- iterate until convergence

this will usually give you the eigenvector corresponding to the largest eigenvalue

why? ... think about it in diagonalized form (blackboard)

PCA — How to Compute?

we have an algorithm for the “largest eigenvector”

how do we find the others?

iterative algorithm

- find the “largest eigenvector”, call it v_1
- project onto $v_1^{\perp}$
- repeat

projection... one of

- replace $x \to x - (x \cdot v_1) v_1$
- multiply $\hat{\Sigma}$ by a projection matrix

“Neural PCA”

PCA by fixed point (above)

- computes one vector at a time
- requires less storage than the classical numerical algorithm
- is much less accurate
- numerical analysts hate it

“neural algorithm”

- computes the top *n* components at once
- will explain later

Summary

looked at invariants and canonicalization again

invariant feature extraction vs linear transforms

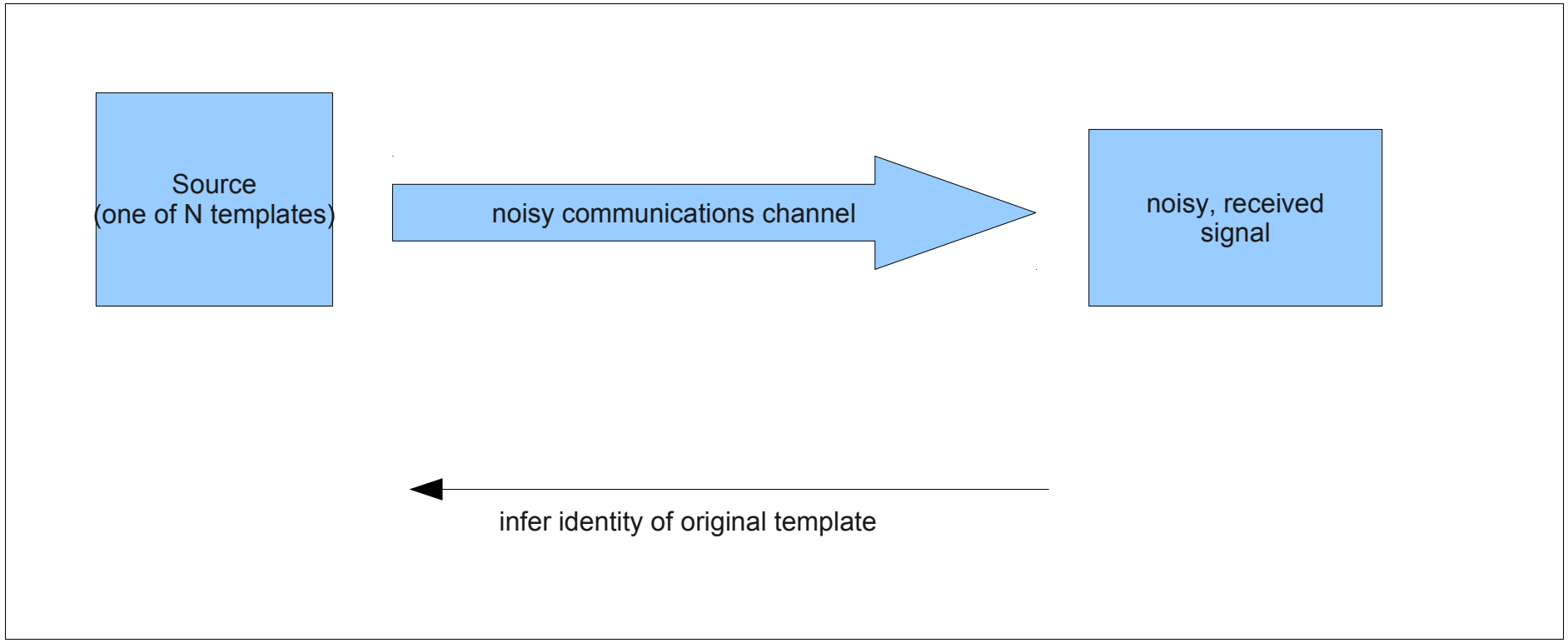
PCA — meaning and interpretation

PCA — “obvious” algorithm

PCA — iterative algorithm

“Nearest Neighbor” makes intuitive sense—is there a mathematical justification?

“Standard Model” — Communications Channel



Simple example:

- transmission of morse code
- seven segment example
- binary transmission of bitmapped character

consider binary images

template $T(x, y) \in \{0,1\}$

image $I(x, y) \in \{0,1\}$

where $(x, y) \in \{0,\ldots, n-1\} \times \{0,\ldots, n-1\}$

Bit Flip Model

assume that the observed image was derived from the template by flipping bits randomly with a low probability ϵ

what is the probability of observing I given T ?

Bit Flip Model

we have

$I(x,y)=T(x,y)$ with probability $1-\epsilon$

$I(x,y)\neq T(x,y)$ with probability ϵ

so

$$
\begin{aligned}
P(I|T) &= \prod_{x,y} (\lfloor T(x,y)=I(x,y) \rfloor \cdot (1-\epsilon) + \lfloor T(x,y)\neq I(x,y) \rfloor \cdot \epsilon) \\
&= \prod_{x,y} ((1-\delta_{x,y}) \cdot (1-\epsilon) + \delta_{x,y} \cdot \epsilon) \\
&= \prod_{x,y} (1 - 2\,\delta_{x,y}\epsilon + \epsilon^2 + \delta_{x,y}\epsilon) \\
&= \prod_{x,y} (1 + \epsilon^2 - \delta_{x,y}\epsilon^2)
\end{aligned}
$$

Bit Flip Model

taking logs

$$\log P(I|T) = \Sigma_{x,y} \log(1 + \epsilon^2 - \delta_{x,y}\epsilon)$$

the summand is just one of two values: $\log(1+\epsilon^2)$ or $\log(1+\epsilon^2-\epsilon)$

therefore

$$\log P(I|T) = n_{=}\, a + n_{\neq}\, b = n\, c - n_{\neq}\, d$$

for some constants a, b, c, d

Bit Flip Model

finding the most similar template = maximizing $\log P(I|T)$

this means using $-\log P(I|T)$ as a dissimilarity measure

$$-\log P(I|T) = n_{\neq}\, d - n\, c$$

$$arg\,min - \log P(I|T) = arg\,min(n_{\neq}\, d - nc) = arg\,min\, n_{\neq}$$

Bit Flip Model vs Hamming Distance

Classification under the Hamming distance is equivalent to finding the “most similar template” under a bitflip model.

Bitflip vs MNIST

you already tried the Hamming distance

was that statistically justified?

how can we do better?

Bitflip vs MNIST

What's wrong?

Digit image generation:

- start with one of several “mental” templates
- reproduce at different scales, translations, rotations, skews
- perform non-uniform distortions
- scanning ← *bitflip model models this noise quite well*

Bitflip vs MNIST

Observe:

- We can apply a bitflip model, but only if we somehow undo the effects of geometric transformations an multiple templates first.
- However, even if we ignore these, the model may yield some improvement.
- The bitflip model is somewhat better for printed characters.
- We'll address finding multiple templates later.

Non-Uniform Bitflip Model

bits of the template aren't equally likely to flip

e.g., pixels near the boundary are more likely to flip

so, actually ϵ depends on x, y

$$P(I|T) = \prod_{x,y} (\lfloor T(x,y) = I(x,y) \rfloor \cdot (1 - \epsilon_{x,y}) + \lfloor T(x,y) \neq I(x,y) \rfloor \cdot \epsilon_{x,y})$$

or in the log domain

$$\log P(I|T) = \Sigma_{x,y} \log((1 - \delta_{x,y})(1 - \epsilon_{x,y}) + \delta_{x,y}\epsilon_{x,y})$$

(this is the equivalent of a diagonal covariance matrix for normal densities; the uniform bitflip model is the equivalent of a spherical normal density)

Estimating Bitflip Probabilities

how do we estimate the bitflip probabilities?

assume

- we have one template T_ω per class ω
- we are given training samples (I_r, ω_r)
- the images I_r were all generated by non-uniform bit-flipping from their corresponding T_{ω_r}

estimating $\epsilon_{x,y}$ is easy now:

- for each x, y count $I(x, y) \neq T(x, y)$ and divide by the total # images
- repeat for each ω

Asymmetric Bitflip Probabilities

Above, we assumed that flipping a black pixel to white has the same probability as flipping a white pixel to black; however, this is often not the case.

We really need two probabilities, the probability of flipping a white pixel to black and the probability of flipping a black pixel to white.

$$P(I(x,y)=1|T(x,y)=0)=\epsilon^{(1)}_{x,y}$$

$$P(I(x,y)=0|T(x,y)=1)=\epsilon^{(0)}_{x,y}$$

These can be estimated from data in analogy to the symmetric bitflip probabilities (by counting).

(this is the equivalent of a diagonal covariance matrix and non-zero mean for normal densities)

Bitflip Dependencies

For the bitflip probabilities, we assumed that the bits were flipped independently.

In general, however, if a bit is flipped, we expect a nearby bit to flip as well.

$$P(I(x,y)=1, I(u,v)=1 | T(x,y)=0, T(u,v)=0)$$

This models small degrees of geometric distortions (why?).

For $d((x,y),(u,v))$ larger than a few pixels, we expect that these probabilities are independent

$$P(I(x,y)=1, I(u,v)=1 | T(x,y)=0, T(u,v)=0) =$$
$$P(I(x,y)=1 | T(x,y)=0) \cdot P(I(u,v)=1 | T(u,v)=0)$$

These kinds of probabilities are the equivalent of a covariance matrix for vectors.