Machine Learning for OR & FE

Dimension Reduction Techniques

Martin Haugh

Department of Industrial Engineering and Operations Research
Columbia University
Email: `martin.b.haugh@gmail.com`

Additional References: Christopher Bishop's *PRML*, David Barber's *BRML*, Hastie, Tibshirani and Friedman's *ESL*, Hastie, Tibshirani and Wainwright's *SLS* and Chiang's *Networked Life*

Principal Components Analysis

Let $\mathbf{x} = (x_1, \ldots, x_d)^\top$ denote a d-dimensional random vector with variance-covariance matrix, $\boldsymbol{\Sigma}$.

The goal of PCA is to construct linear combinations

$$p_i \; = \; \sum_{j=1}^{d} w_{ij} \; x_j \,, \quad \text{for } i = 1, \ldots, d$$

in such a way that:

(1) The p_i's are orthogonal so that $\mathsf{E}[p_i \, p_j] = 0$ for $i \neq j$

(2) The p_i's are ordered in such a way that:

(i) p_1 explains the largest percentage of the total variance

(ii) each p_i explains the largest percentage of the total variance that has not already been explained by $p_1, \ldots, p_{i-1}$.

The Eigen Decomposition of a Matrix

In practice it is common to apply PCA to the **normalized** random variables so that $\mathsf{E}[x_i] = 0$ and $\mathsf{Var}(x_i) = 1$

– achieved by subtracting the means from the original random variables and dividing by their standard deviations.

– done to ensure that no one component of **x** can influence the analysis by virtue of that component's measurement units.

Key tool of PCA is the eigen decomposition of a square matrix.

The eigen decomposition implies that any symmetric matrix, $\mathbf{A} \in \mathbb{R}^{d\times d}$ can be written as

$$\mathbf{A} \;=\; \mathbf{\Gamma}\,\mathbf{\Delta}\,\mathbf{\Gamma}^\top \tag{1}$$

where:

(i) $\mathbf{\Delta}$ is a diagonal matrix, diag$(\lambda_1, \ldots, \lambda_d)$, of the eigen values of $\mathbf{A}$

- without loss of generality ordered so that $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$.

(ii) $\mathbf{\Gamma}$ is an orthogonal matrix with i^{th} column of $\mathbf{\Gamma}$ containing i^{th} standardized eigen-vector, $\boldsymbol{\gamma}_i$, of $\mathbf{A}$.

- "standardized" means $\boldsymbol{\gamma}_i^\top \boldsymbol{\gamma}_i = 1$
- orthogonality of $\mathbf{\Gamma}$ implies $\mathbf{\Gamma}\,\mathbf{\Gamma}^\top \;=\; \mathbf{\Gamma}^\top\,\mathbf{\Gamma} \;=\; \mathbf{I}_d$.

PCA and the Factor Loadings

Since $\mathbf{\Sigma}$ is symmetric can take $\mathbf{A} = \mathbf{\Sigma}$ in (1). The positive semi-definiteness of $\mathbf{\Sigma}$ implies $\lambda_i \geq 0$ for all $i = 1, \ldots, d$.

The principal components of $\mathbf{x}$ then given by $\mathbf{p} = (p_1, \ldots, p_d)$ satisfying

$$\mathbf{p} \;=\; \mathbf{\Gamma}^\top \mathbf{x}. \tag{2}$$

Note that:

(a) $\mathrm{E}[\mathbf{p}] \;=\; \mathbf{0}$ since $\mathrm{E}[\mathbf{x}] \;=\; \mathbf{0}$

(b) $\mathrm{Cov}(\mathbf{p}) \;=\; \mathbf{\Gamma}^\top \mathbf{\Sigma} \mathbf{\Gamma} \;=\; \mathbf{\Gamma}^\top (\mathbf{\Gamma} \mathbf{\Delta} \mathbf{\Gamma}^\top) \mathbf{\Gamma} \;=\; \mathbf{\Delta}$

- so components of $\mathbf{p}$ are uncorrelated as desired
- and $\mathrm{Var}(p_i) = \lambda_i$.

The matrix $\mathbf{\Gamma}^\top$ is called the matrix of factor loadings. We can invert (2) to obtain

$$\mathbf{x} \;=\; \mathbf{\Gamma}\, \mathbf{p} \tag{3}$$

- so easy to go back and forth between $\mathbf{x}$ and $\mathbf{p}$.

Explaining The Total Variance

We can measure the ability of the first few principal components to explain the total variance:

$$\sum_{i=1}^{d} \mathsf{Var}(p_i) \;=\; \sum_{i=1}^{d} \lambda_i \;=\; \mathsf{trace}(\Sigma) \;=\; \sum_{i=1}^{d} \mathsf{Var}(x_i). \tag{4}$$

If we take $\sum_{i=1}^{d} \mathsf{Var}(p_i) = \sum_{i=1}^{d} \mathsf{Var}(x_i)$ to measure the total variability then by (4) can interpret

$$\frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{d} \lambda_i}$$

as the percentage of total variability explained by first k principal components.

The Maximum Variance Formulation

Let $p_1 = \boldsymbol{\gamma}_1^\top \mathbf{x}$ be the 1^{st} principal component. Can easily show that $\boldsymbol{\gamma}_1$ solves

$$\begin{aligned} \max_{\mathbf{a}} \quad & \mathsf{Var}(\mathbf{a}^\top \mathbf{x}) \\ \text{subject to} \quad & \mathbf{a}^\top \mathbf{a} = 1. \end{aligned}$$

More generally, let $p_i = \boldsymbol{\gamma}_i^\top \mathbf{x}$ be the i^{th} principal component for $i = 1, \ldots, d$.

Then $\boldsymbol{\gamma}_i$ solves

$$\begin{aligned} \max_{\mathbf{a}} \quad & \mathsf{Var}(\mathbf{a}^\top \mathbf{x}) \\ \text{subject to} \quad & \mathbf{a}^\top \mathbf{a} = 1 \qquad (5) \\ & \mathbf{a}^\top \boldsymbol{\gamma}_j = 0, \quad j = 1, \ldots, i-1. \qquad (6) \end{aligned}$$

So each successive principal component finds the linear combination of the components of $\mathbf{x}$ that has maximal variance subject to the normalization constraint (5) and the orthogonal constraints (6).

The Minimum Reconstruction Formulation

We can also obtain the principal components of $\mathbf{x}$ by posing the problem as one of minimizing the sum of squared differences between each sample vector, $\mathbf{x}_i$ and it's representation, $\tilde{\mathbf{x}}_i$, in some lower M-dimensional sub-space

– here we let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ denote (centered) sample observations of $\mathbf{x}$.

We have the following problem:

$$\begin{aligned} & \min_{\mathbf{B},\mathbf{Z}} \sum_{i=1}^{n} \sum_{j=1}^{d} \left[x_{j,i} - \sum_{k=1}^{M} b_{j,k} z_{k,i} \right]^2 \\ \equiv \quad & \min_{\mathbf{B},\mathbf{Z}} \|\mathbf{X} - \mathbf{B}\mathbf{Z}\|_F^2 \end{aligned} \tag{7}$$

where $\mathbf{B}_{j,k} := b_{j,k}$ is the basis in the M-dimensional subspace and $\mathbf{Z}_{k,i} := z_{k,i}$ is the weighting of the i^{th} sample on the k^{th} element in the basis.

The Minimum Reconstruction Formulation

Problem (7) is sufficient to identify a **unique reconstruction point** for each $\mathbf{x}_i$.

But it is not sufficient to identify a unique optimal $\mathbf{B}$ and $\mathbf{Z}$. Why?

- hence we impose the constraint that $\mathbf{B}^\top \mathbf{B} = \mathbf{I}_M$.

But rotations still (why?) a problem

- recall $\mathbf{U}$ a rotation matrix if $\mathbf{U}^\top \mathbf{U} = \mathbf{I}$

Choosing directions of maximal variance resolves this issue

- and then end up with solution where $\mathbf{B} =$ first M columns of $\mathbf{\Gamma}$.

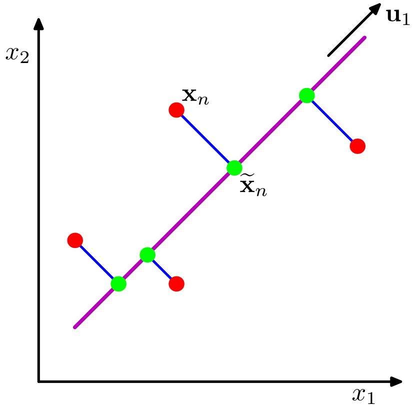


Figure 12.2 from Bishop: Principal component analysis seeks a space of lower dimensionality, known as the principal subspace and denoted by the magenta line, such that the orthogonal projection of the data points (red dots) onto this subspace maximizes the variance of the projected points (green dots). An alternative definition of PCA is based on minimizing the sum-of-squares of the projection errors, indicated by the blue lines.

The Singular Value Decomposition (SVD)

The **singular value decomposition** (SVD) of a $d \times n$ matrix $\mathbf{X}$ is given by

$$\mathbf{X} \;=\; \mathbf{U}\,\mathbf{D}\,\mathbf{V}^\top \tag{8}$$

where:

1. $\mathbf{U}$ is $d \times d$ and $\mathbf{U}^\top\mathbf{U} = \mathbf{I}_d$
2. $\mathbf{V}$ is $n \times n$ and $\mathbf{V}^\top\mathbf{V} = \mathbf{I}_n$
3. $\mathbf{D}$ is an $d \times n$ diagonal matrix of the positive singular values

We assume the decomposition has ordered the singular values so that the upper left diagonal element of $\mathbf{D}$ has the largest singular value etc.

Now note that

$$\begin{aligned}\mathbf{X}\mathbf{X}^\top &= \mathbf{U}\,\mathbf{D}\,\mathbf{V}^\top\,\mathbf{V}\,\mathbf{D}^\top\,\mathbf{U}^\top \\ &= \mathbf{U}\,\tilde{\mathbf{D}}\,\mathbf{U}^\top\end{aligned}$$

where $\tilde{\mathbf{D}} := \mathbf{D}\,\mathbf{D}^\top$ is a $d \times d$ diagonal matrix with the n squared singular values on the diagonal.

The Singular Value Decomposition (SVD)

We can also use the singular value decomposition of a matrix to determine the principal components. Why?

In practice we don't know the true covariance matrix and have to make do with the sample covariance matrix based on (centered) observations, $\mathbf{x}_1, \ldots, \mathbf{x}_n$.

Let $\boldsymbol{\Sigma}$ be the sample variance-covariance matrix so that

$$\boldsymbol{\Sigma} \; = \; \frac{1}{n-1} \, \mathbf{X} \, \mathbf{X}^\top$$

where $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]$.

The SVD is more general than the eigen decomposition and so PCA is often performed via an SVD decomposition:

1. The eigen vectors are given by the $\mathbf{U}$ matrix of the SVD
2. The eigen values are the squares of the singular values from the SVD

Note also that we can use the SVD to approximate $\mathbf{X}$ using only the first M singular values

$$\mathbf{X} \; = \; \mathbf{U} \, \mathbf{D} \, \mathbf{V}^\top \; \approx \; \mathbf{U}_M \, \mathbf{D}_M \, \mathbf{V}_M^\top .$$

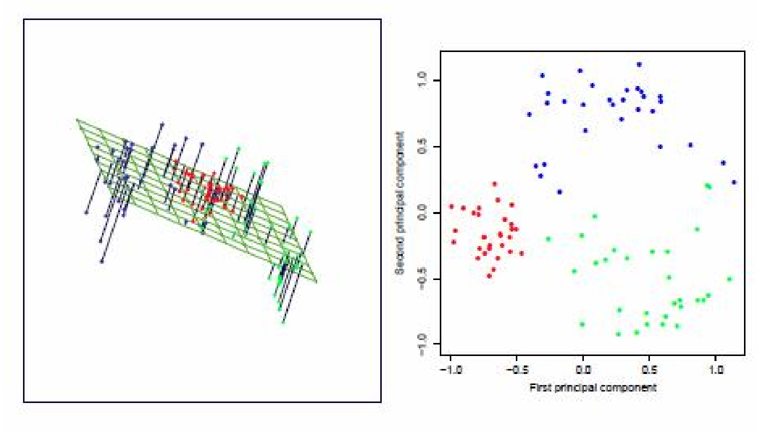


Figure 14.21 from HTF: The best rank-two linear approximation to the half-sphere data. The right panel shows the projected points with coordinates given by $\mathbf{U}_2\mathbf{D}_2$, the first two principal components of the data.

High-Dimensional Data

In practice it is often the case that $n << d$

- e.g. have $n = 500$ images of $d = 1000 \times 1000 = 10^6$ pixels
- can cause computational difficulties since computing eigen-decomposition takes $O(d^3)$ operations.

But number of non-zero eigen values **cannot exceed** n. Why?

Can exploit this fact as follows:

1. Let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be the n samples of the d-dimensional $\mathbf{x}$
2. Recall we can write $\boldsymbol{\Sigma} = \frac{1}{n-1}\mathbf{X}\mathbf{X}^\top$ where $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]$.
3. Let $\mathbf{E}$ be the $d \times n$ matrix of eigen-vectors of $\boldsymbol{\Sigma}$ with corresponding eigen-values in the $n \times n$ diagonal matrix $\boldsymbol{\Delta}$.

   This implies

$$\frac{1}{n-1}\mathbf{X}\mathbf{X}^\top\mathbf{E} \;=\; \mathbf{E}\boldsymbol{\Delta}$$

High-Dimensional Data

Therefore have

$$\frac{1}{n-1}\mathbf{X}^\top\mathbf{X}\mathbf{X}^\top\mathbf{E} \;=\; \mathbf{X}^\top\mathbf{E}\boldsymbol{\Delta}. \tag{9}$$

If we set $\tilde{\mathbf{E}} := \mathbf{X}^\top\mathbf{E}$ then (9) can be written as

$$\frac{1}{n-1}\mathbf{X}^\top\mathbf{X}\tilde{\mathbf{E}} \;=\; \tilde{\mathbf{E}}\boldsymbol{\Delta}$$

so that $\tilde{\mathbf{E}}$ is the matrix of eigen vectors for the $n \times n$ matrix $\frac{1}{n-1}\mathbf{X}^\top\mathbf{X}$

- can be calculated in $O(n^3)$ operations
- so much easier to calculate than eigen-vectors of the $d \times d$ matrix $\frac{1}{n-1}\mathbf{X}\mathbf{X}^\top$.

So we simply solve for $\mathbf{E}$ by finding $\tilde{\mathbf{E}}$ and then setting (why?)

$$\mathbf{E} = \frac{1}{n-1}\mathbf{X}\tilde{\mathbf{E}}\boldsymbol{\Delta}^{-1}$$

– eigen values are unchanged.

But note $O(n^2d)$ work required to compute $\mathbf{X}^\top\mathbf{X}$ so could instead use "economical" SVD.

Some PCA Applications in Machine Learning & OR/FE

Will consider several application domains for PCA:

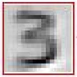
- Data compression
  - hand-writing
  - eigen faces
- Reduced-rank LDA
- Nearest neighbor classification
- Financial modeling
- Latent semantic analysis
- Missing data and collaborative filtering / recommender systems.

There are many, many more application domains.

Data Compression: Digit Recognition Part I

- Figure 14.22 from HTF displays 130 handwritten 3's from a total of 658
  - digitized as 16×16 grayscale images so can view them as points in $\mathbb{R}^{256}$.
- Figure 14.23 from HTF plots the first 2 principal components along with images that are close to the vertices of the grid
  - vertices are placed at 5%, 25%, 50%, 75% and 95% quantile points.
- First component accounts for "lengthening of the lower tail of the three".
- Second component accounts for "character thickness".
- Equation (14.55) from HTF yields the 2-component model

$$\hat{f}(\lambda) = \bar{x} + \lambda_1 v_1 + \lambda_2 v_2$$

$=$ $+ \lambda_1 \cdot$ $+ \lambda_2 \cdot$  .

- Figure 14.24 from HTF displays the singular values (from the SVD) compared with a randomized version of the data.

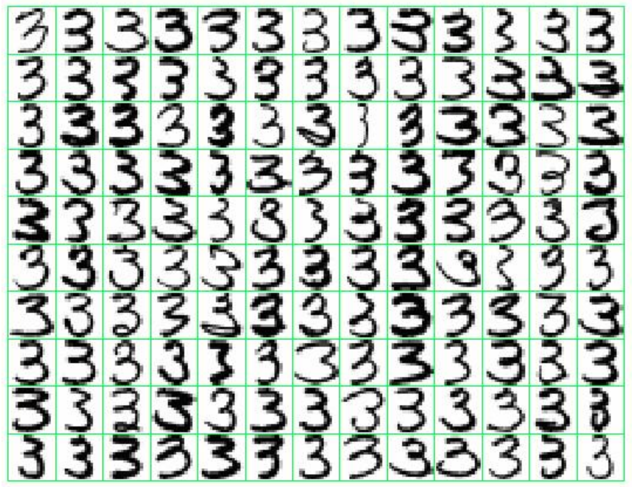


Figure 14.22 from HTF: A sample of 130 handwritten 3's shows a variety of writing styles.

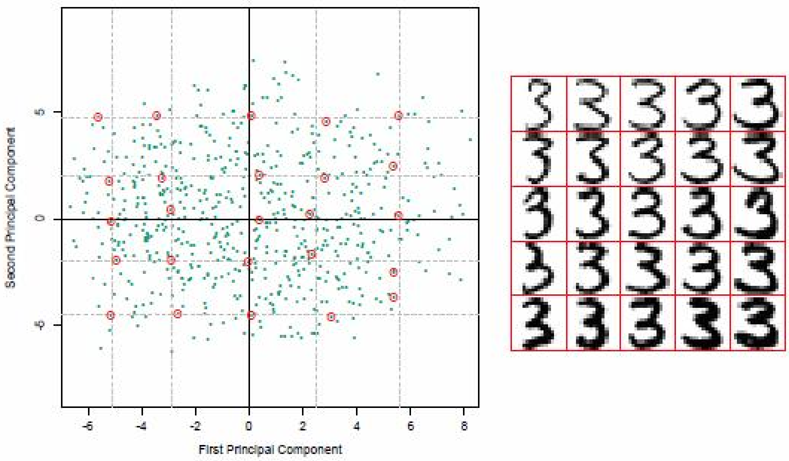


Figure 14.23 from HTF: Left panel: the first two principal components of the hand- written threes. The circled points are the closest projected images to the vertices of a grid, defined by the marginal quantiles of the principal components. Right panel: The images corresponding to the circled points. These show the nature of the first two principal components.

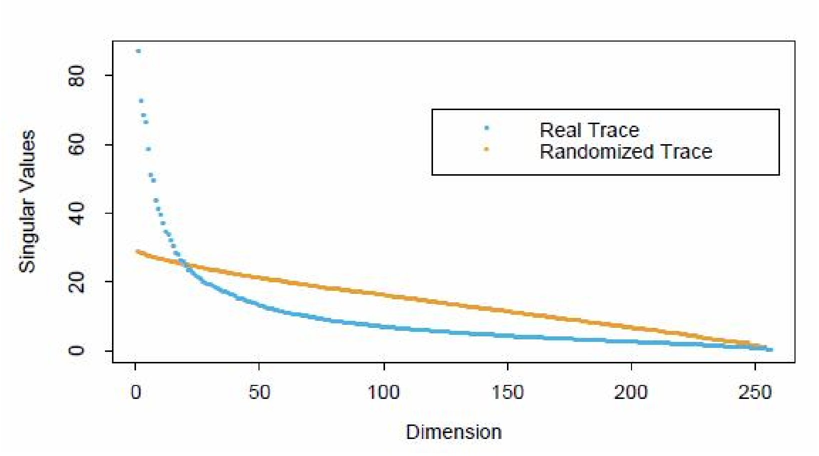


Figure 14.24 from HTF: The 256 singular values for the digitized threes, compared to those for a randomized version of the data (each column of **X** was scrambled).

Data Compression: Eigen Faces

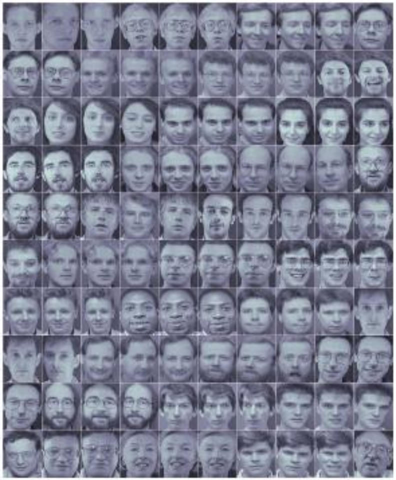
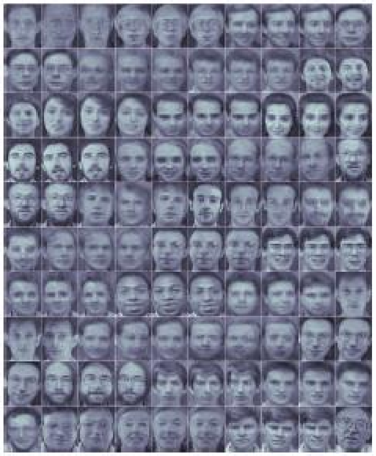


Figure 15.5 from Barber: 100 training images. Each image consists of $92 \times 112 = 10304$ greyscale pixels. The train data is scaled so that, represented as an image, the components of each image sum to 1. The average value of each pixel across all images is 9.70×10^{-5}. This is a subset of the 400 images in the full Olivetti Research Face Database.


(a)

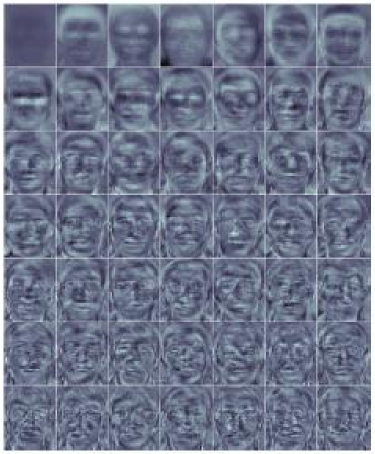

(b)

Figure 15.6 from Barber: (a): SVD reconstruction of the images in fig(15.5) using a combination of the 49 eigen-images. (b): The eigen-images are found using SVD of the images in fig(15.5) and taking the mean and 48 eigenvectors with largest corresponding eigenvalue. The images corresponding to the largest eigenvalues are contained in the first row, and the next 7 in the row below, etc. The root mean square reconstruction error is 1.121×10^{-5}, a small improvement over PLSA (see fig(15.16)).

Reduced-Rank LDA

Recall the log-ratio in LDA was given by

$$\log \frac{\mathsf{P}(G=k|\mathbf{X}=\mathbf{x})}{\mathsf{P}(G=l|\mathbf{X}=\mathbf{x})} = \log \frac{\hat{\pi}_k}{\hat{\pi}_l} - \frac{1}{2}\left(\mathbf{x}-\hat{\boldsymbol{\mu}}_k\right)^\top \hat{\boldsymbol{\Sigma}}^{-1}\left(\mathbf{x}-\hat{\boldsymbol{\mu}}_k\right) + \frac{1}{2}\left(\mathbf{x}-\hat{\boldsymbol{\mu}}_l\right)^\top \hat{\boldsymbol{\Sigma}}^{-1}\left(\mathbf{x}-\hat{\boldsymbol{\mu}}_l\right) \tag{10}$$

We let $\hat{\boldsymbol{\Sigma}} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{U}^\top$ be the eigen decomposition of $\hat{\boldsymbol{\Sigma}}$ where $\boldsymbol{U}$ is $m \times m$ orthornormal and $\boldsymbol{D}$ is a diagonal matrix of the positive eigen values of $\hat{\boldsymbol{\Sigma}}$.

Then used a change of variable with $\mathbf{x}^* := \boldsymbol{D}^{-1/2}\boldsymbol{U}^\top\mathbf{x}$ so that $\mathsf{Cov}\left(X^*\right) = \mathbf{I}$.

Could then rewrite (10) as

$$\log \frac{\mathsf{P}(G=k|\mathbf{X}=\mathbf{x})}{\mathsf{P}(G=l|\mathbf{X}=\mathbf{x})} = \log \frac{\hat{\pi}_k}{\hat{\pi}_l} - \frac{1}{2}\left(\mathbf{x}^*-\hat{\boldsymbol{\mu}}_k^*\right)^\top \left(\mathbf{x}^*-\hat{\boldsymbol{\mu}}_k^*\right) + \frac{1}{2}\left(\mathbf{x}^*-\hat{\boldsymbol{\mu}}_l^*\right)^\top \left(\mathbf{x}^*-\hat{\boldsymbol{\mu}}_l^*\right) \tag{11}$$

where $\hat{\boldsymbol{\mu}}_k^*$ and $\hat{\boldsymbol{\mu}}_l^*$ are the centroid estimates for class k and l respectively under the new coordinate system.

Reduced-Rank LDA

The K centroids lie in a $K-1$-dimensional subspace, H_{K-1} say, of $\mathbb{R}^m$ so we can project points to H_{K-1} and then classify the projected points according to the nearest centroid, after adjusting for $\log \hat{\pi}_k$'s.

Then looked for an L-dimensional subspace $H_L \subseteq H_{K-1}$ for $L < K-1$ where H_L is chosen to maximize **between-class** variance relative to **within-class** variance.

Particular procedure is:

1. First perform a change of variables $\mathbf{x}_i^* := \boldsymbol{D}^{-1/2}\boldsymbol{U}^\top \mathbf{x}_i$ for $i = 1, \ldots, N$.
2. Let M^* be the $m \times K$ matrix of class centroids under these new coordinates.
3. Treat the K centroids as "observations", compute their cov. matrix, $\boldsymbol{B}^*$.
4. Let $\boldsymbol{B}^* = \boldsymbol{V}\boldsymbol{D}_{\boldsymbol{B}}\boldsymbol{V}^\top$ be its eigen decomposition and let $\boldsymbol{v}_l$ be the l^{th} column of $\boldsymbol{V}$.
5. Then $Z_l := \boldsymbol{v}_l^\top \boldsymbol{D}^{-1/2}\boldsymbol{U}^\top \mathbf{x} = \boldsymbol{v}_l^\top \mathbf{x}^*$ is the l^{th} discriminant variable.

So central computational tool for reduced-rank LDA is simply **PCA** (applied to the centroids).

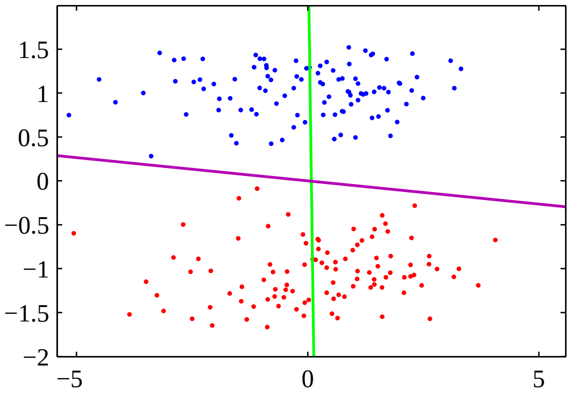


Figure 12.7 from Bishop: A comparison of principal component analysis with Fisher's linear discriminant for linear dimensionality reduction. Here the data in two dimensions, belonging to two classes shown in red and blue, is to be projected onto a single dimension. PCA chooses the direction of maximum variance, shown by the magenta curve, which leads to strong class overlap, whereas the Fisher linear discriminant takes account of the class labels and leads to a projection onto the green curve giving much better class separation.

PCA and Financial Modeling

There are many applications of PCA in finance:

1. Representing movements in term-structure of interest-rates
   - useful for interpretation
   - building factor models
   - and hedging.
2. Representing movements in futures strips
   - useful for interpretation
   - building factor models
   - and hedging.
3. Building factor models for equities.
4. Scenario generation in risk-management.
5. Estimation of risk measures such as VaR and CVaR.
6. And others ...

Nearest-Neighbor Classification

In nearest-neighbor classification it can be expensive to compute the distance between data-points when dimensionality is high.

Can overcome this problem by using PCA to approximate distances.

Let $\mathbf{\Gamma}_1$ contain the first k principal components with k large enough so that

$$\mathbf{x} = \mathbf{\Gamma}\mathbf{p} \approx \mathbf{\Gamma}_1\mathbf{p}_1$$

provides a good approximation. If $\mathbf{x}^a$ and $\mathbf{x}^b$ are any two points in $\mathbb{R}^d$ then

$$\begin{aligned} \left(\mathbf{x}^a - \mathbf{x}^b\right)^\top \left(\mathbf{x}^a - \mathbf{x}^b\right) &= (\mathbf{\Gamma}\mathbf{p}^a - \mathbf{\Gamma}\mathbf{p}^b)^\top(\mathbf{\Gamma}\mathbf{p}^a - \mathbf{\Gamma}\mathbf{p}^b) \\ &\approx (\mathbf{\Gamma}_1\mathbf{p}_1^a - \mathbf{\Gamma}_1\mathbf{p}_1^b)^\top(\mathbf{\Gamma}_1\mathbf{p}_1^a - \mathbf{\Gamma}_1\mathbf{p}_1^b) \\ &= (\mathbf{p}_1^a - \mathbf{p}_1^b)^\top\mathbf{\Gamma}_1^\top\mathbf{\Gamma}_1(\mathbf{p}_1^a - \mathbf{p}_1^b) \\ &= (\mathbf{p}_1^a - \mathbf{p}_1^b)^\top(\mathbf{p}_1^a - \mathbf{p}_1^b). \qquad (12) \end{aligned}$$

– can be much cheaper to compute (12).

And data is also often noisy so projecting onto a lower dimensional sub-space can produce superior classification: see Figure 15.7 from Barber.

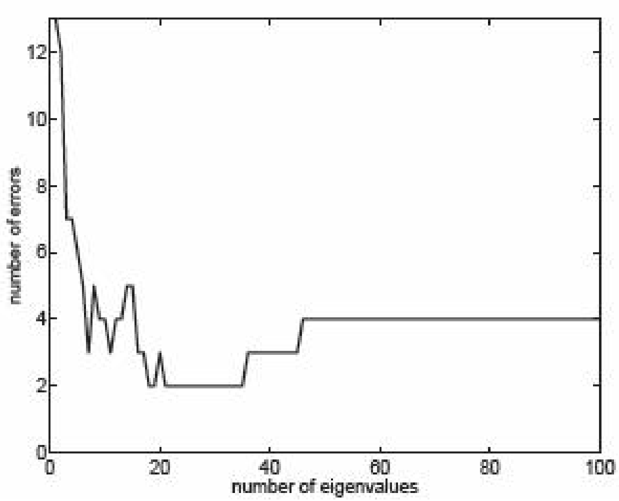


Figure 15.7 from Barber: Finding the optimal PCA dimension to use for classifying hand-written digits using nearest neighbours. 400 training examples are used, and the validation error plotted on 200 further examples. Based on the validation error, we see that a dimension of 19 is reasonable.

Latent Semantic Analysis (LSA)

In the document analysis literature, PCA is called latent semantic analysis (LSA).

We assume:

1. There are d words in our dictionary, $\mathcal{D}$
2. And n documents in the corpus.

The j^{th} document could then be represented by $\mathbf{x}^j = (x_1^j, \ldots, x_d^j)^\top$

- the so-called bag-of-words representation
- where x_i^j refers to the $\#$ occurrences of the i^{th} word in the j^{th} document.

But more common to normalize this number:

Definition. The term-frequency, tf_i^j, is the number of times that the word i appears in document j divided by the number of words in document j, i.e.

$$\text{tf}_i^j := \frac{\#_{i,j}}{\sum_i \#_{i,j}}$$

where $\#_{i,j}$ is the number of times that word i appears in document j.

The TF-IDF Representation

Definition. The inverse-document-frequency, idf_i, is defined as

$$\text{idf}_i \;:=\; \log\left(\frac{n}{\#\text{ of documents that contain word } i}\right).$$

It is possible to use different definitions of idf_i

- as long as rarely occurring words are given more weight when they do occur.

Can now define the TF-IDF representation:

$$x_i^j \;:=\; \text{tf}_i^j \times \text{idf}_i$$

A corpus of documents is then represented by

$$\mathbf{X} \;=\; [\mathbf{x}^1 \cdots \mathbf{x}^n]$$

– a $d \times n$ matrix which is typically very large!

Latent Semantic Analysis (LSA)

LSA is simply a matter of applying PCA to (the variance-covariance matrix of) $\mathbf{X}$

- with the interpretation that the principal directions, i.e. eigen vectors, now define **topics**.

One weakness of LSA is that eigen vectors can have negative components

- but how can a word contribute negatively to a "topic"?

Probabilistic latent semantic analysis (PLSA) overcomes this problem

- based on non-negative matrix factorization.

Topic modeling has been a "hot topic" in the machine learning community.

Examples 15.4 from Barber

The dictionary, $\mathcal{D}$, contains 10 words:

`influenza, flu, headache, nose, temperature,`
`bed, cat, dog, rabbit, pet`

The document corpus contains 2000 documents:

1. some articles discuss ailments, and some of these discuss influenza
2. some articles related to pets
3. some articles are background articles unrelated to ailments

Some of the ailment articles are informal and use the word "`flu`" whereas others use the more formal "`influenza`".

Each document is therefore represented by a 10-dimensional vector
- $x_i^j = 1$ if i^{th} word occurs in j^{th} document
- $x_i^j = 0$ otherwise.

See Figures 15.8 and 15.9 from Barber.

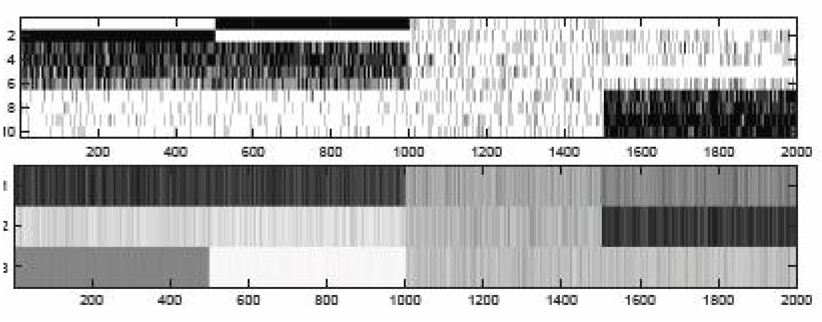


Figure 15.8 from Barber: (Top) Document data for a dictionary containing 10 words and 2000 documents. Black indicates that a word was present in a document. The data consists of two distinct topics and a random background topic. The first topic contains two sub-topics which differ only in their usage of the first two words, ‘influenza’ and ‘flu’. (Bottom) The projections of each datapoint onto the three principal components.

Information Retrieval and Example 15.5 from Barber

Consider a large collection of documents from which a dictionary, $\mathcal{D}$, is created.

Given a document, $\mathbf{x}^f$, how do we find the "closest" document to it in the collection?

First need a measure of dissimilarity between documents

- could use $d(\mathbf{x}^f, \mathbf{x}^i) := (\mathbf{x}^f - \mathbf{x}^i)^\top (\mathbf{x}^f - \mathbf{x}^i)$.

And then select the document that solves

$$\min_i \, d(\mathbf{x}^f, \mathbf{x}^i).$$

In this case it's a good idea (why?) to scale each vector so that it has unit length

- leads to the equivalent cosine similarity

$$s(\mathbf{x}^f, \mathbf{x}^i) \;:=\; \cos(\theta)$$

where θ is the angle between $\mathbf{x}^f$ and $\mathbf{x}^i$.

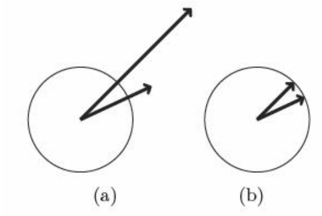


Figure 15.10 from Barber: (a): Two bag-of-word vectors. The Euclidean distance between the two is large. (b): Normalised vectors. The Euclidean distance is now related directly to the angle between the vectors. In this case two documents which have the same relative frequency of words will both have the same dissimilarity, even though the number of occurrences of the words is different.

Problem with bag-of-words representation is that TD matrix will have mainly zeros

- so differences may be due to noise.

LSA can help (why?) solve this problem

- consider Example 15.4 from Barber.

Kernel PCA

Given $\mathbf{x} \in \mathbb{R}^d$, can define $\tilde{\mathbf{x}} := \phi(\mathbf{x}) \in \mathbb{R}^D$ where $D > d$ and possibly $D = \infty$. Letting $\tilde{\mathbf{X}} := [\tilde{\mathbf{x}}_1 \cdots \tilde{\mathbf{x}}_n]$, the corresponding variance-covariance matrix is

$$\mathbf{\Sigma}_\phi \; := \; \frac{1}{n-1}\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\top.$$

Let $\mathbf{e}$ be an eigen vector of $\frac{1}{n-1}\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\top$ with Lagrange multiplier λ. Then have

$$\frac{1}{n-1}\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\top\mathbf{e} \; = \; \lambda\mathbf{e}.$$

After pre-multiplying by $\tilde{\mathbf{X}}^\top$ obtain

$$\frac{1}{n-1}\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}}\mathbf{f} \; = \; \lambda\mathbf{f} \qquad (13)$$

where $\mathbf{f} := \tilde{\mathbf{X}}^\top\mathbf{e}$

- an eigen equation that we have seen before!

Kernel PCA

Now note that $[\tilde{\mathbf{X}}^\top \tilde{\mathbf{X}}]_{i,j} = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j) := k(\mathbf{x}_i, \mathbf{x}_j) := \mathbf{K}_{i,j}$.

The kernel trick therefore applies and we can write (13) as

$$\frac{1}{n-1}\mathbf{K}\mathbf{f} = \lambda \mathbf{f}. \tag{14}$$

Having found $\mathbf{f}$ by solving (14) we can then recover $\mathbf{e} = \frac{1}{\lambda(n-1)}\tilde{\mathbf{X}}\mathbf{f}$.

Corresponding (kernel) principal component then given by

$$\begin{aligned} p &= \tilde{\mathbf{x}}^\top \mathbf{e} \\ &= \frac{1}{\lambda(n-1)}\tilde{\mathbf{x}}^\top \tilde{\mathbf{X}}\mathbf{f} \\ &= \frac{1}{\lambda(n-1)}\sum_{l=1}^{n} k(\mathbf{x}, \mathbf{x}_l) f_l \end{aligned}$$

where f_l is the l^{th} component of $\mathbf{f}$.

Centering the Data

Have implicitly assumed (where?) that the data, i.e. the $\phi(\mathbf{x}_i)$'s, have already been centered.

But even if the $\mathbf{x}_i$'s have been centered in general the $\phi(\mathbf{x}_i)$'s will not. Why?

Straightforward to check that we can resolve this problem, i.e. center the $\phi(\mathbf{x}_i)$'s, by replacing $\mathbf{K}$ in (14) with $\hat{\mathbf{K}}$ where

$$\begin{aligned} \hat{\mathbf{K}}_{i,j} \quad := \quad & \mathbf{K}_{i,j} \ - \ \frac{1}{n}\sum_{l=1}^{n} k(\mathbf{x}_l, \mathbf{x}_j) \ - \ \frac{1}{n}\sum_{d=1}^{n} k(\mathbf{x}_i, \mathbf{x}_d) \\ & + \ \frac{1}{n^2}\sum_{l,k=1}^{n} k(\mathbf{x}_l, \mathbf{x}_k). \end{aligned} \tag{15}$$

Problems with Kernel PCA

In regular PCA we can use the principal directions, i.e. eigen vectors corresponding to the largest eigen values, to construct good approximations to $\mathbf{x}$:

$$\mathbf{x} = \mathbf{\Gamma}\mathbf{p} \approx \mathbf{\Gamma}_1\mathbf{p}_1$$

Can also do this in kernel PCA but the approximation will lie in the higher dimensional space $\mathbb{R}^D$

- hard to visualize or interpret this approximation
- would prefer to project it back to the original space, $\mathbb{R}^d$
- but this is generally hard since the mapping $\mathbf{x} \rightarrow \phi(\mathbf{x})$ is **non-linear**.

One possible solution is to solve

$$\min_{\mathbf{x}'} \left(\phi(\mathbf{x}') - \mathbf{z}\right)^\top \left(\phi(\mathbf{x}') - \mathbf{z}\right)$$

where $\mathbf{z} \in \mathbb{R}^D$ is the approximation to $\phi(\mathbf{x})$ given by the kernel PCA

- generally a computationally expensive, non-convex optimization problem.

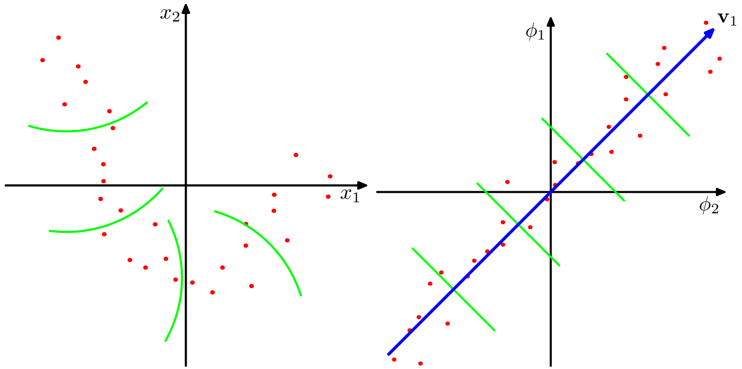


Figure 12.16 from Bishop: Schematic illustration of kernel PCA. A data set in the original data space (left-hand plot) is projected by a nonlinear transformation $\phi(\mathbf{x})$ into a feature space (right-hand plot). By performing PCA in the feature space, we obtain the principal components, of which the first is shown in blue and is denoted by the vector $\mathbf{v}_1$. The green lines in feature space indicate the linear projections onto the first principal component, which correspond to nonlinear projections in the original data space. Note that in general it is not possible to represent the nonlinear principal component by a vector in x space.

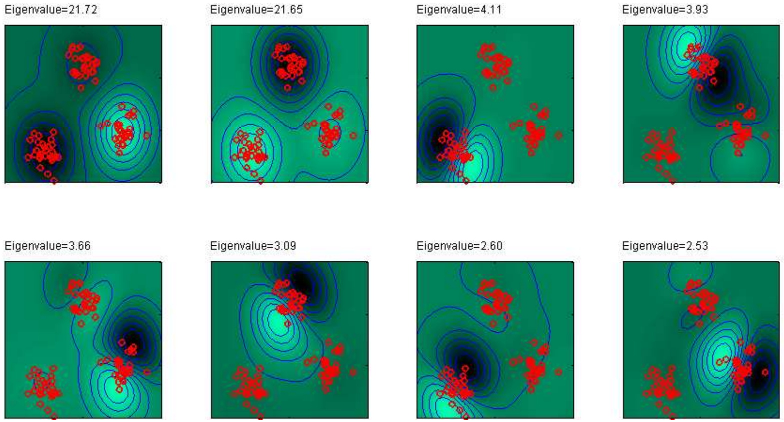
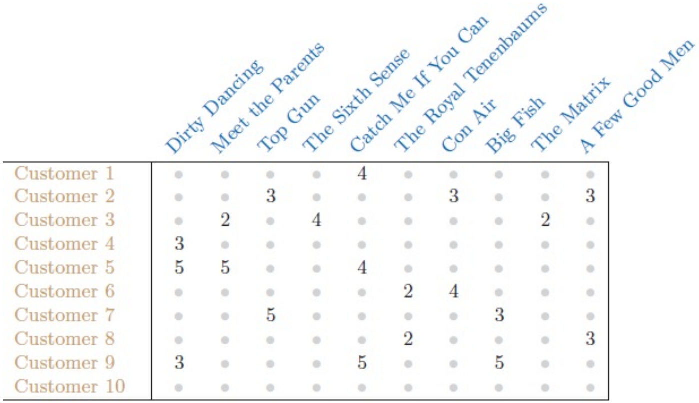


Figure 12.17 from Bishop: Example of kernel PCA, with a Gaussian kernel applied to a synthetic data set in two dimensions, showing the first eight eigenfunctions along with their eigenvalues. The contours are lines along which the projection onto the corresponding principal component is constant. Note how the first two eigenvectors separate the three clusters, the next three eigenvectors split each of the cluster into halves, and the following three eigenvectors again split the clusters into halves along directions orthogonal to the previous splits.

Matrix Completion and Collaborative Filtering



	Dirty Dancing	Meet the Parents	Top Gun	The Sixth Sense	Catch Me If You Can	The Royal Tenenbaums	Con Air	Big Fish	The Matrix	A Few Good Men
Customer 1	●	●	●	●	4	●	●	●	●	●
Customer 2	●	●	3	●	●	●	3	●	●	3
Customer 3	●	2	●	4	●	●	●	●	2	●
Customer 4	3	●	●	●	●	●	●	●	●	●
Customer 5	5	5	●	●	4	●	●	●	●	●
Customer 6	●	●	●	●	●	2	4	●	●	●
Customer 7	●	●	5	●	●	●	●	3	●	●
Customer 8	●	●	●	●	●	2	●	●	●	3
Customer 9	3	●	●	●	5	●	●	5	●	●
Customer 10	●	●	●	●	●	●	●	●	●	●

Table 7.2 from Hastie, Tibshirani and Wainwright's *Statistical Learning with Sparsity* (2015): Excerpt of the `Netflix` movie rating data. The movies are rated from 1 (worst) to 5 (best). A symbol shaded grey circles represents a missing value: a movie that was not rated by the corresponding customer.

Matrix Completion and Collaborative Filtering

In many applications we are missing many / most of the elements of a matrix **X**.

e.g. Consider a $d \times n$ movies-ratings matrix with d movies and n users

- then x_{ui} represents the rating (on some scale) of user u for movie i
- **X** will then be very sparse in that very few of the x_{ui}'s will be known.

Goal then is to somehow "fill in" or impute the missing values

- has obvious applications in collaborative filtering and recommender systems
- e.g. the Netflix prize.

Two main approaches to solving this problem:

1. **Neighborhood** methods
2. **Matrix factorization** methods

Will discuss each of them but first we establish some **baseline** estimators.

Baseline Estimators

A couple of obvious baseline estimators:

1. Set $\hat{x}_{ui} = \bar{\mathbf{x}}$, average of all ratings in the (training) data-set.
2. Introduce biases and set
$$\hat{x}_{ui} = \bar{\mathbf{x}} + b_u + b_i \tag{16}$$
where
$$b_i := \frac{\sum_u x_{ui}}{M_i} - \bar{x} \tag{17}$$
$$b_u := \frac{\sum_i x_{ui}}{M_u} - \bar{x} \tag{18}$$
$M_i = \#$ users that rated movie i and $M_u = \#$ movies rated by user u.
3. Use (16) but choose b_i's, b_u's via
$$\min_{b_i, b_u} \sum_{(u,i)} (x_{ui} - \hat{x}_{ui})^2 + \lambda \left(\sum_i b_i^2 + \sum_u b_u^2 \right).$$

(16) seems sensible (why?) so will henceforth work with the **residual matrix** $\tilde{\mathbf{X}}$
$$\tilde{\mathbf{X}}_{u,i} := \tilde{x}_{ui} := x_{ui} - \hat{x}_{ui}.$$

Neighborhood Methods

Will continue to use the movie-ratings matrix application to develop ideas.

Neighborhood methods rely on calculating **nearest neighbors**:

1. Two **rows** are near neighbors if corresponding users have similar taste in movies.
2. Two **columns** are near neighbors if the corresponding movies obtained similar ratings from users.

To compute nearest neighbors we need a distance / similarity metric.

Can use cosine similarity so

$$\begin{aligned} d_{ij} := \cos(\theta_{ij}) &= \frac{\tilde{\mathbf{x}}_i^\top \tilde{\mathbf{x}}_j}{\sqrt{||\tilde{\mathbf{x}}_i||_2 ||\tilde{\mathbf{x}}_j||_2}} \\ &= \frac{\sum_u \tilde{x}_{ui}\tilde{x}_{uj}}{\sqrt{\sum_u \tilde{x}_{ui}^2 \sum_u \tilde{x}_{uj}^2}} \end{aligned} \qquad (19)$$

where summation is only over users u that rated both movies i and j.

Can define d_{uv} for users u and v similarly.

Neighborhood Methods

Given a fixed movie i can rank all other movies in descending order of $|d_{ij}|$.

Then top L movies in the ranking are the L nearest neighbors of movie i.

Yields new estimator

$$\hat{x}^N_{ui} = \bar{\mathbf{x}} + b_u + b_i + \underbrace{\frac{\sum_{j \in \mathcal{L}_i} d_{ij}\, \tilde{x}_{uj}}{\sum_{j \in \mathcal{L}_i} |d_{ij}|}}_{\text{collaborative filtering term}} \tag{20}$$

where $\mathcal{L}_i$ denotes the neighborhood of movie i.

Question: Why rank movies according to $|d_{ij}|$ rather than d_{ij} when defining $\mathcal{L}_i$?

Remarks:

1. Could just as easily use d_{uv}'s rather than d_{ij}'s and adjust (20) appropriately.
2. Still need to choose value of L and other obvious tweaks can be made.
3. Can use (20) to define new residual matrix $\tilde{\mathbf{X}}_{u,i}$
   - and then start working on predicting missing elements of $\tilde{\mathbf{X}}_{u,i}$.

Matrix Factorization Methods

General matrix factorization problem formulated as

$$\hat{\mathbf{X}} = \arg \min_{\mathbf{M} \in \mathbb{R}^{d \times n}} ||\mathbf{X} - \mathbf{M}||_F^2 \tag{21}$$

subject to

$$\Phi(\mathbf{M}) \leq r \tag{22}$$

where $|| \cdot ||_F$ denotes **Frobenius norm** and $\Phi(\cdot)$ a constraint function used to encourage sparsity or regularization etc.

- many interesting problems obtained by varying Φ.

PCA Revisited

e.g. Take $\Phi(\mathbf{M}) = \text{rank}(\mathbf{M})$. Then solution to (21) is

$$\hat{\mathbf{X}} = \mathbf{U}\mathbf{D}_r\mathbf{V}^\top \tag{23}$$

where $\mathbf{U}\mathbf{D}\mathbf{V}^\top = \text{SVD}(\mathbf{X})$ and $\mathbf{D}_r$ is constructed from $\mathbf{D}$ by keeping just the first r singular values.

Really just an equivalent version of (7) once you know the following:

Fact: Any rank r matrix $\mathbf{M}$ can be factored as $\mathbf{M} = \mathbf{B}\mathbf{Z}$ where $\mathbf{B} \in \mathbb{R}^{d \times r}$ is orthogonal and $\mathbf{Z} \in \mathbb{R}^{r \times n}$.

Back to Matrix Completion

Let $\Omega \subseteq \{1, \ldots, d\} \times \{1, \ldots, n\}$ denote observed entries of **X**. Then one version of matrix completion problem is

$$\begin{aligned} & \min_{\text{rank}(\mathbf{M}) \leq r} \sum_{(u,i) \in \Omega} (x_{ui} - m_{ui})^2 \\ \equiv \quad & \min_{\mathbf{B},\mathbf{Z}} \sum_{(u,i) \in \Omega} \left(x_{ui} - \sum_{k=1}^{r} b_{uk} z_{ki} \right)^2 \end{aligned} \tag{24}$$

where we used fact from previous slide.

Unfortunately a non-convex problem and solution not known

- but many algorithms available for finding local minima.

Algorithm #1: Find optimal **Z** in terms of **B** and then try to minimize over **B**.

Matrix Completion: Alternating Least Squares

Algorithm #2 (ALTERNATING LEAST SQUARES):

Select an initial $\hat{\mathbf{B}}$ and then iterate the following two steps until convergence:

(i) Optimize over $\mathbf{Z}$ for given $\hat{\mathbf{B}}$. Let $\hat{\mathbf{Z}}$ be the optimal solution.

(ii) Optimize over $\mathbf{B}$ for given $\hat{\mathbf{Z}}$. Let $\hat{\mathbf{B}}$ be the optimal solution.

- Guaranteed to converge to a **local min** but may do so slowly in practice.
- Can find principal directions or eigen vectors by performing SVD on final $\mathbf{X} \approx \hat{\mathbf{B}}\hat{\mathbf{Z}}$.
- Should use cross-validation or validation / test set to choose r.

Figure 15.11 from Barber based on this method.

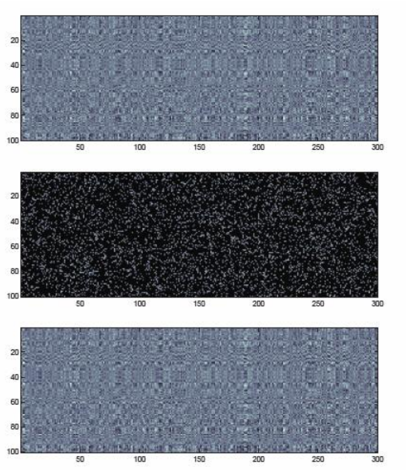


Figure 15.11 from Barber: Top: original data matrix **X**. Black is missing, white present. The data is constructed from a set of only 5 basis vectors. Middle : **X** with missing data (80% sparsity). Bottom : reconstruction found using `svdm.m`, SVD for missing data. This problem is essentially easy since, despite there being many missing elements, the data is indeed constructed from a model for which SVD is appropriate. Such techniques have application in collaborative filtering and recommender systems where one wishes to 'fill in' missing values in a matrix.

Matrix Completion: Hard-Impute

Algorithm #3 (HARD-IMPUTE):

1. Use initial guess of missing values to obtain initial complete matrix $\mathbf{X}_{\text{comp}}$.
2. Compute a rank-r SVD of $\mathbf{X}_{\text{comp}}$, i.e. solve (23) to obtain $\hat{\mathbf{X}}_{\text{comp}}$.
3. Use $\hat{\mathbf{X}}_{\text{comp}}$ to obtain new guess, $\mathbf{X}_{\text{comp}}$, of complete matrix.
4. Iterate steps 2 and 3 until convergence (which usually occurs but is not guaranteed).

As before, should use cross-validation or validation / test set to choose r.

A Convex Relaxation to Matrix Completion

Alternative formulation is based on the following convex relaxation of (24):

$$\min_{\mathbf{M}} \sum_{(u,i)\in\Omega} (x_{ui} - m_{ui})^2 + \lambda||\mathbf{M}||_* \tag{25}$$

where $||\cdot||_*$ is the nuclear norm or trace norm with

$$||\mathbf{M}||_* := \text{sum of singular values of } \mathbf{M}.$$

Algorithm #4 (to follow) easy to implement and solves (25).

But first need some definitions ...

A Convex Relaxation to Matrix Completion

Definition. Let $\mathbf{W} = \mathbf{U}\mathbf{D}\mathbf{V}^\top$ be the SVD of a rank-r matrix $\mathbf{W}$. Then the soft-threshold operator $S_\lambda(\cdot)$ is given by

$$S_\lambda(\mathbf{W}) := \mathbf{U}\mathbf{D}_\lambda\mathbf{V}^\top$$

where

$$\mathbf{D}_\lambda := \text{diag}\left[(s_1 - \lambda)^+, \ldots, (s_r - \lambda)^+\right].$$

where $s_1 \geq s_2 \cdots \geq s_r$ are the singular values on the diagonal matrix $\mathbf{D}$.

Definition. The projection operators, $P_\Omega : \mathbb{R}^{d \times n} \to \mathbb{R}^{d \times n}$ and $P_\Omega^\perp : \mathbb{R}^{d \times n} \to \mathbb{R}^{d \times n}$ are defined as

$$[P_\Omega(\mathbf{X})]_{ui} := \begin{cases} x_{ui}, & \text{if } (u,i) \in \Omega \\ 0, & \text{otherwise.} \end{cases}$$

and

$$\left[P_\Omega^\perp(\mathbf{X})\right]_{ui} := \begin{cases} x_{ui}, & \text{if } (u,i) \notin \Omega \\ 0, & \text{otherwise.} \end{cases}$$

Matrix Completion: Soft-Impute

Algorithm #4 (SOFT-IMPUTE):

Initialize $\mathbf{X}_{\text{old}} = 0$, set $\mathbf{X}$ to be any completion of the matrix and iterate until convergence:

1. Compute
$$\hat{\mathbf{X}}_\lambda := S_\lambda\left(P_\Omega(\mathbf{X}) + P_\Omega^\perp(\mathbf{X}_{\text{old}})\right)$$
2. Update $\mathbf{X}_{\text{old}} := \hat{\mathbf{X}}_\lambda$.

λ chosen via cross-validation or validation / test set approach.

SOFT-IMPUTE requires full SVD of a potentially large dense matrix in each iteration. Such a matrix may be too large to even store but note

$$P_\Omega(\mathbf{X}) + P_\Omega^\perp(\mathbf{X}_{\text{old}}) = \underbrace{P_\Omega(\mathbf{X}) - P_\Omega(\mathbf{X}_{\text{old}})}_{\text{sparse}} + \underbrace{\mathbf{X}_{\text{old}}}_{\text{low rank}} \tag{26}$$

Each component on r.h.s of (26) easy to store (why?) for sufficiently large λ.

Implemented in R in `softImpute` package.

Other Results and Formulations

Considerable theoretical work on determining how large $|\Omega|$ must be in order to recover **X** with **"high probability"**.

- results can depend on solution method, rank$(\mathbf{X})$, distribution of **X** in matrix population, "coherence" / "incoherence" of **X** etc.

Many other interesting instances (and generalizations) of (21) and (22) including sparse PCA, robust PCA.

e.g. Decompose **X** into sum of low-rank and sparse components by solving

$$\min_{\substack{\mathbf{L} \in \mathbb{R}^{d \times n} \\ \mathbf{S} \in \mathbb{R}^{d \times n}}} \frac{1}{2}||\mathbf{X} - (\mathbf{L} + \mathbf{S})||_F^2 + \lambda_1 ||\mathbf{L}||_* + \lambda_2 ||\mathbf{S}||_1$$

Chapters 7 and 8 of *Statistical Learning with Sparsity* by Hastie, Tibshirani and Wainwright a good (but quite technical) source.

The Netflix Prize

Define the **root mean-squared error** (RMSE) to be

$$\text{RMSE} := \sqrt{\sum_{(u,i)} \frac{(x_{ui} - \hat{x}_{ui})^2}{C}}.$$

In 2006 Netflix launched a competition with a $1m prize – the **Netflix Prize**.

Goal of competition was to develop a recommender system that improved *Cinematch* (Netflix's proprietary system) by at least 10%.

If goal not achieved in 3 years, consolation prize of $50k awarded to best team.

Progress prizes of $50k also awarded each year.

Competition sparked huge interest — over 5k teams and 44k submissions!

Data from period 1999 to 2005 on 480k users and 17,770 movies.

- Each movie rated by more than 5,000 users on average.
- Each user rated more than 200 movies on average.

The Netflix Data-Set

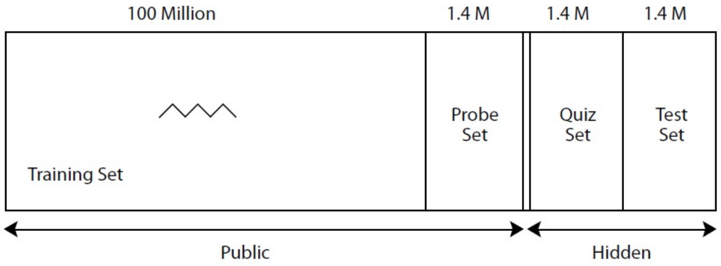


Figure 4.1 from Chiang's *Networked Life: 20 Questions and Answers* (2012): The Netflix Prize's four data sets. The training set and probe set were publicly released, whereas the quiz set and test set were hidden from the public and known only to Netflix. The probe, quiz, and test sets had similar statistical properties, but the probe set could be used by each competing team as often as they want, and the quiz set at most once a day. The final decision was based on comparison of the RMSE on the test set.

Improving *Cinematch* by 10% meant achieving an RMSE of

- 0.8563 on the quiz set
- 0.8572 on the test set.

Winner determined by performance on test set.

Matrix Factorization Results on the `Netflix` Data-set

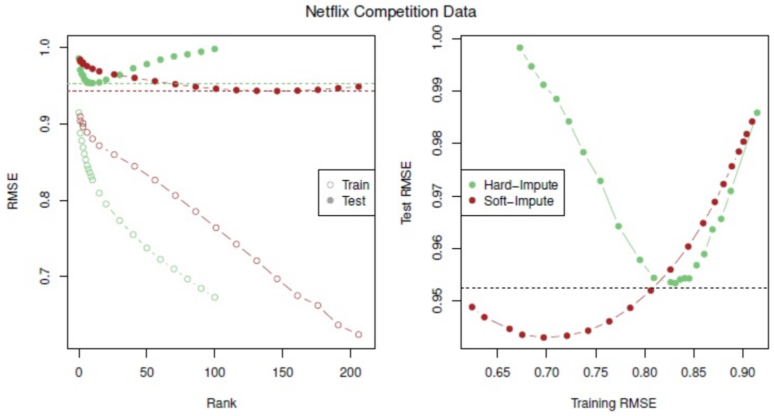


Figure 7.2 from Hastie, Tibshirani and Wainwright's *Statistical Learning with Sparsity* (2015): Left: Root-mean-squared error for the `Netflix` training and test data for the iterated-SVD (HARD-IMPUTE) and the convex spectral-regularization algorithm (SOFT-IMPUTE). Each is plotted against the rank of the solution, an imperfect calibrator for the regularized solution. Right: Test error only, plotted against training error, for the two methods. The training error captures the amount of fitting that each method performs. The dotted line represents the baseline "Cinematch" score.

And the Winner Is ...

Turns out that 10% was an excellent choice by `Netflix`:

- Relatively easy to improve on *Cinematch* by approx 8%
- But much(!) harder to improve by 10%.

Most successful submissions used combinations or **ensembles** of nearest neighbors and matrix factorization

- enough to get approx 8% improvement.

But many “tricks” required to get additional 2%.

Winning team’s tricks included **temporal effects** and **implicit feedback**

- hard to develop without domain knowledge!

Final leader-board available at at

`http://www.netflixprize.com/leaderboard.html`

- $1m prize determined by 20 minute time differential in submission of top two entries!

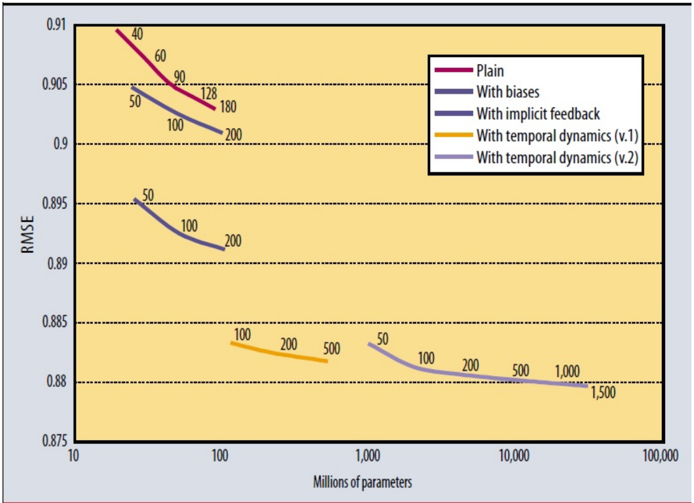


Figure 4. Matrix factorization models' accuracy. The plots show the root-mean-square error of each of four individual factor models (lower is better). Accuracy improves when the factor model's dimensionality (denoted by numbers on the charts) increases. In addition, the more refined factor models, whose descriptions involve more distinct sets of parameters, are more accurate. For comparison, the Netflix system achieves RMSE = 0.9514 on the same dataset, while the grand prize's required accuracy is RMSE = 0.8563.

Source: Matrix Factorization Techniques for Recommender Systems by Koren, Bell and Volinsky, *Computer* (2009).

Reputation Systems

The ever increasing amount of online activity and data has been accompanied by the **decentralization** of control.

For example:

1. `IMDB` and `Rotten Tomatoes` help us choose what movies to watch.
2. How do we decide what sellers to trust on `Ebay`?
3. Who should we follow on `Twitter`?
4. How does `Google` figure out the importance of web-pages?
5. Why is `Yelp` so popular?

Reputation Systems

An important problem is in figuring out who or what pages are the "best"

- clearly this is also a dimension reduction problem!

Here we will consider `Google`'s PageRank system:

- clearly of enormous importance
- has applications to networks beyond evaluating the importance of web-pages
- can use it to highlight the ongoing challenges with reputation systems.

Question: How do we define reputation? How about this definition?

"You're great if other great people think you're great."

Any problems with this? Yes ... but how do reputations in the real world work?

Google's Naive PageRank Algorithm

Consider the web and suppose there are a total of d pages.

Say $i \rightarrow j$ if page i links to page j and define $c(i) := \#$ of out-links from page i.

Let $X_t \in \{1, \ldots, d\}$ denote the page that a web-surfer visits at time t.

Initial model assumes X_t a Markov chain with transition matrix $\mathbf{Q}$ where

$$\begin{aligned} Q_{i,j} &:= \mathsf{P}\left(X_{t+1} = j \,|\, X_t = i\right) \\ &= \begin{cases} \frac{1}{c(i)}, & \text{if } i \rightarrow j \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

Would like to find a stationary distribution, $\boldsymbol{\mu}$, of the Markov chain so that

$$\boldsymbol{\mu} = \boldsymbol{\mu}\mathbf{Q}.$$

Could then use $\boldsymbol{\mu}(i)$ to measure the importance of the i^{th} page. Why?

But $\boldsymbol{\mu}$ will not be **unique** unless X is irreducible

- and X will not be irreducible in general because (for example) many web-pages will not have out-links.

Naive PageRank

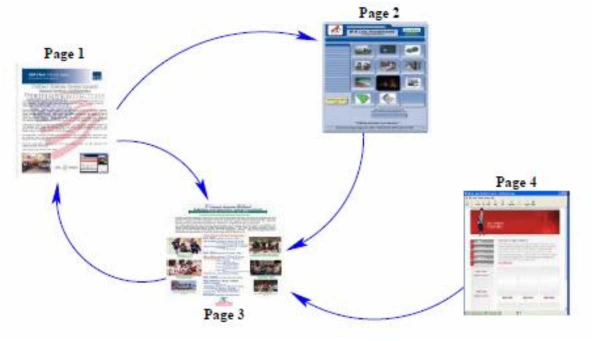


Figure 14.46 from HTF: Page-Rank algorithm: example of a small network

Another (less important) issue with defining web-page reputations in this way is that it is easily "gamed" by just a couple of web-pages that **collude**.

See for example "Making Eigenvector-Based Reputation Systems Robust to Collusion" (2004) by Zhang, Goel, Govindan, Mason and Van Roy.

Google's PageRank Algorithm

We resolve the uniqueness problem (and make collusion harder!) by instead assuming a transition matrix $\bar{\mathbf{Q}}$ where

$$\bar{Q}_{i,j} \quad := \quad (1-\epsilon)Q_{i,j} \; + \; \frac{\epsilon}{d} > 0 \tag{27}$$

for $0 < \epsilon < 1$. Can write (27) equivalently as

$$\bar{\mathbf{Q}} \quad := \quad (1-\epsilon)\mathbf{Q} \; + \; \frac{\epsilon}{d}\mathbf{1}\mathbf{1}^\top \tag{28}$$

where $\mathbf{1}$ is a $d \times 1$ vector of 1's.

Note that (28) implies that $\bar{\mathbf{Q}}$ is irreducible and therefore has a **unique stationary distribution**, $\boldsymbol{\mu}$, satisfying

$$\boldsymbol{\mu} \; = \; \boldsymbol{\mu}\bar{\mathbf{Q}}. \tag{29}$$

Can interpret the resulting Markov chain as one where with probability ϵ we choose a new web-page randomly from the entire universe of web-pages and with probability $1-\epsilon$ we click randomly on a link on the current page

- the **random walk** interpretation.

Google's PageRank Algorithm

$\boldsymbol{\mu}$ gives the page-ranks of all the web-pages.

If we run the Markov chain for a sufficiently long time then proportion of time we spend on page i is $\boldsymbol{\mu}(i)$.

Page-rank or "importance" of a page is therefore captured by importance of the pages that link to it

- so still a circular definition of reputation!

PageRank has been generalized, manipulated, defended, etc. and reputation systems in general have been the cause of much litigation

- should give you an idea of just how important they are!

Computing PageRank is computationally intensive. Rather than solving (29) directly there are two popular approaches:

1. Power iteration – the most commonly used approach.
2. Monte-Carlo which exploits our earlier **random walk** interpretation
    - an approach that has been proposed for the fast **updating** of PageRank.

Using Power Iteration to Estimate PageRank

Power iteration is very simple:

1. Choose an initial vector, $\boldsymbol{\mu}_0$. (Easy choice is $\boldsymbol{\mu}_0 = \mathbf{1}/d$.)
2. Then iterate $\boldsymbol{\mu}_{t+1} \; = \; \boldsymbol{\mu}_t \bar{\mathbf{Q}}$ for $t = 1, \ldots$ until convergence.

Question: Is convergence guaranteed?

Answer: Yes!

Proof: Let $\Delta_t := \sum_{i=1}^{d} |\boldsymbol{\mu}_t(i) - \boldsymbol{\mu}(i)|$ where $\boldsymbol{\mu}$ is the solution to (29).
Using the triangle inequality it is easy to see that

$$|\boldsymbol{\mu}_{t+1}(i) - \boldsymbol{\mu}(i)| \leq (1-\epsilon) \sum_{j:\, j \text{ links to } i} \frac{|\boldsymbol{\mu}_t(j) - \boldsymbol{\mu}(j)|}{c(j)} \tag{30}$$

Summing (30) over i yields $\Delta_{t+1} \leq (1-\epsilon)\Delta_t$.
Therefore (why?) $\Delta_t \to 0$ as $t \to \infty$ and so $\boldsymbol{\mu}_t(i)$ converges to $\boldsymbol{\mu}(i)$.$\square$

Question: Is the convergence fast?

PageRank

`Google` originally used PageRank to index the web and to provide an ordering of web-pages according to their reputation / importance.

This ordering was vital to search and advertising

- but also provided an obvious motive for companies to try and increase the PageRank of their web-pages.

Over the years the PageRank system has adapted and heuristics added

- in part to counter attempts at collusion and manipulation.

While the specific details are private it is very likely that PageRank still plays the key role in `Google`'s search algorithms.

PageRank and Search

Question: How might you use PageRank as part of a search system?

Here's how ...

1. First compute the PageRank of all web-pages and store it in descending order of importance
   - just like a book with the 1^{st} page having the highest PageRank, the 55^{th} page having the 55^{th} highest PageRank etc.
2. A reverse index for all commonly searched terms is also computed and stored
   - this is exactly like an index at the back of a book telling you what pages important terms appear on.

Steps 1 and 2 are updated periodically.

3. When a user searches for "dog" say, the reverse index tells you exactly what web-pages the term "dog" appears on and returns the first n of them.
   - This is easy and "correct". Why?

Question: How do you handle a multiple term search, e.g. "dog" *and* "pony"?

Other Applications / Extensions of PageRank

The ordering returned by PageRank can easily be adapted to reflect a preference for a specific subset, $\mathcal{C}$ say, of web-pages.

Can do this by changing the algorithm so that w.p. ϵ we choose a new web-page randomly **from $\mathcal{C}$** and w.p. $1 - \epsilon$ we click randomly on a link on the current page

- is known as personalized PageRank.

PageRank can be applied in most network settings where links between nodes indicate some sort of preference. Other examples include:

1. **Recommendation systems:** form a bipartite graph with movies on one side and users on the other. Add a link between a movie and a user if the user liked that movie. Now PageRank gives an ordering of movies *and* users.

   Question: How would you recommend movies to a specific user?

2. Networks can also be created out of users and tweets on `Twitter` with links added to identify creators of tweets and followers. Can use such a system to identify influential people and make recommendations to users.

Appendix: Non-Negative Matrix Factorization

Non-negative matrix factorization is an alternative to PCA where the matrix components are required to be non-negative

- useful for modeling non-negative data, e.g. images, text, trading volume etc.

Given a $d \times n$ matrix, $\mathbf{X} \geq 0$, the goal is to find $\mathbf{W} \geq 0$ and $\mathbf{H} \geq 0$ such that

$$\mathbf{X} \approx \mathbf{WH}$$

where $\mathbf{W}$ is $d \times r$, $\mathbf{H}$ is $r \times n$ and $r \leq \max(d, n)$.

One approach is to maximize

$$L(\mathbf{W}, \mathbf{H}) := \sum_{i=1}^{d} \sum_{j=1}^{n} \left[x_{ij} \log (\mathbf{WH})_{ij} - (\mathbf{WH})_{ij} \right] \tag{31}$$

- the log-likelihood from a model in which $x_{ij} \sim \text{Poisson}\left((\mathbf{WH})_{ij}\right)$

Appendix: Non-Negative Matrix Factorization

$L(\mathbf{W}, \mathbf{H})$ is not convex and therefore does not have a unique maximum

- instead must search for a good local maximum.

Can be shown the following updates converge to a local maximum:

$$w_{ik} \quad \leftarrow \quad w_{ik} \frac{\sum_{j=1}^{n} h_{kj} x_{ij} / (\mathbf{WH})_{ij}}{\sum_{j=1}^{p} h_{kj}} \tag{32}$$

$$h_{kj} \quad \leftarrow \quad h_{kj} \frac{\sum_{i=1}^{d} w_{ik} x_{ij} / (\mathbf{WH})_{ij}}{\sum_{i=1}^{d} w_{ik}} \tag{33}$$

Figure 14.33 from HTF shows NNMF, vector quantization and PCA applied to a database of $n = 2,429$ images

- each image represented by a 19×19 matrix of pixels so $d = 381$
- positive values are represented by black pixels
- negative values are represented by red pixels.

Note how NNMF represents faces with a set of basis images each of which resembles a face part.

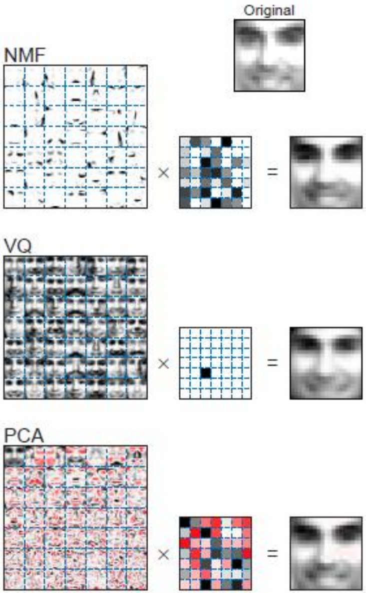


Figure 14.33 from HTF: Non-negative matrix factorization (NMF), vector quantization (VQ, equivalent to k-means clustering) and principal components analysis (PCA) applied to a database of facial images. Details are given in the text. Unlike VQ and PCA, NMF learns to represent faces with a set of basis images resembling parts of faces.

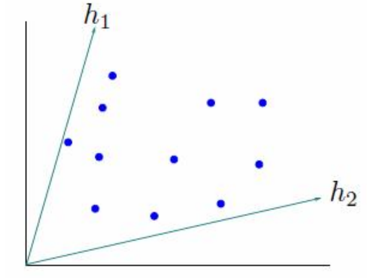


Figure 14.34 from HTF: Non-uniqueness of the non-negative matrix factorization. There are 11 data points in two dimensions. Any choice of the basis vectors h_1 and h_2 in the open space between the coordinate axes and data, gives an exact reconstruction of the data.

Often claimed that the basis in NNMF will be much easier to interpret in problem applications where it makes sense for basis elements to be non-negative

- but the non-uniqueness of the basis as demonstrated by Figure 14.34 makes this argument harder to make
- also implies solution to the NNMF problem depends on the **starting values**.

Appendix: Probabilistic Latent Semantic Analysis (PLSA)

PLSA is closely related to non-negative matrix factorization

- again the matrix elements must be non-negative
- but there is now a probabilistic interpretation of the matrix components which yields additional constraints on the factorization.

Again given a $d \times n$ matrix, $\mathbf{X} \geq 0$. Goal is to find $\mathbf{W} \geq 0$ and $\mathbf{H} \geq 0$ such that

$$\mathbf{X} \approx \mathbf{WH}$$

where $\mathbf{W}$ is $d \times r$, $\mathbf{H}$ is $r \times n$ and $r \leq \max(d, n)$.

Suppose we have a **count** matrix $\mathbf{C}$ where $C_{ij} = \#$ of times $U = i$ and $V = j$ where U and V are random variables. Then

$$X_{ij} := \frac{C_{ij}}{\sum_{i=1}^{d} \sum_{j=1}^{n} C_{ij}}$$

may be interpreted as $\mathsf{P}(U = i, V = j)$.

Appendix: Probabilistic Latent Semantic Analysis (PLSA)

May then seek a decomposition where

$$\begin{aligned} \mathsf{P}(U = i, V = j) &\approx \sum_k \underbrace{\tilde{\mathsf{P}}(U = i \,|\, Z = k)}_{\mathbf{W}_{ik}} \underbrace{\tilde{\mathsf{P}}(V = j \,|\, Z = k)\tilde{\mathsf{P}}(Z = k)}_{\mathbf{H}_{kj}} \\ &\equiv \tilde{\mathsf{P}}(U = i, V = j) \end{aligned} \quad (34)$$

where Z is a hidden or latent random variable and we're assuming U and V are conditionally independent given Z.

Note each column of **W** now represents a possible value of Z.

There are EM-style algorithms for finding **W** and **H** but only convergence to a local minimum is assured

- see Section 15.6 of Barber for details.

Note also that PLSA does not yield a **sequential** basis

- i.e. the $k = 2$ solution, $\mathbf{W}_2$ say, will not coincide with the first 2 columns of the $k = 3$ solution, $\mathbf{W}_3$
- not true of PCA.

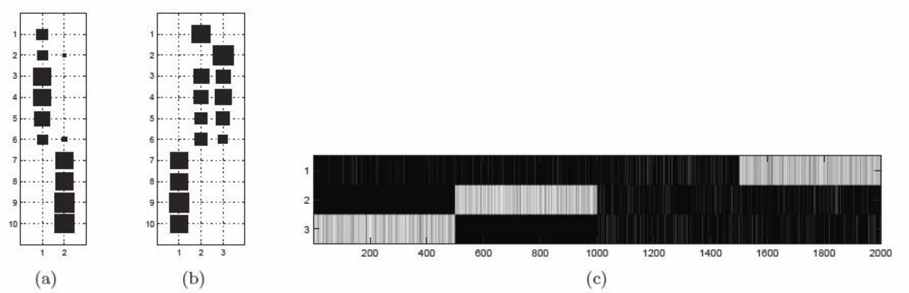


Figure 15.14 from Barber: PLSA for the document data in fig(15.8). (a): Hinton diagram for two basis vectors. (b): Hinton diagram for three basis vectors. (c): The projections for the three basis vectors case. The solution is quite satisfactory since the first 1000 documents are clearly considered to be from the similar 'ailments' topics, the next 500 from some other non-specific 'background' topic, and the last 500 from a separate 'pet' topic.

Results of applying PLSA to our example term-document matrix with $k = 3$: (b) is the **W** matrix and (c) is the **H** matrix.

Appendix: Conditional PLSA

Sometimes it is more natural to model X_{ij} as $\mathsf{P}(U = i \,|\, V = j)$.

May then seek a decomposition of the form

$$\begin{aligned} \mathsf{P}(U = i \,|\, V = j) &\approx \sum_k \underbrace{\tilde{\mathsf{P}}(U = i \,|\, Z = k)}_{\mathbf{W}_{ik}} \underbrace{\tilde{\mathsf{P}}(Z = k \,|\, V = j)}_{\mathbf{H}_{kj}} \\ &\equiv \tilde{\mathsf{P}}(U = i, \,|\, V = j) \end{aligned} \tag{35}$$

Conditional PLSA assumes that U is independent of V conditional on Z.

Again there are EM-style algorithms for performing conditional PLSA.

Appendix: Discovering the Basis (E.G. 15.7 in Barber)

Images displayed in Fig. 15.15(a) created from 4 base images in Fig. 15.15(b).

Each base image is positive and scaled so that the sum of the pixels is 1; i.e. for $k = 1, \ldots, 4$

$$\sum_i \mathsf{P}(U = i \mid Z = k) \; = \; 1$$

where U indexes the pixels and Z indexes the 4 base images.

Training images in (a) then constructed as **random** convex combinations of the 4 base images.

Each training image has elements $\mathsf{P}(U = i \mid V = j)$

- U indexes the pixels
- V indexes the training images.

Goal: given only the training images, reconstruct the 4 basis images.

Results are shown in Figure 15.15(c) and are very good

- but number of elements in basis, i.e. 4, was assumed to be known!

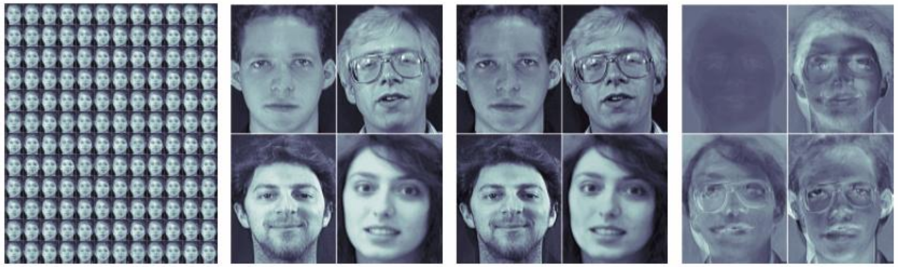


Figure 15.15 from Barber: (a) Training data, consisting of a positive (convex) combination of the base images. (b): The chosen base images from which the training data is derived. (c): Basis learned using conditional PLSA on the training data. This is virtually indistinguishable from the true basis. (d): Eigenbasis (sometimes called 'eigenfaces').

Eigenbasis from the first 4 principal components is given in Figure 15.15(d)

- not nearly as interpretable as the conditional PLSA basis
- but of course we "cheated".

Figure 15.16 provides a fairer comparison.

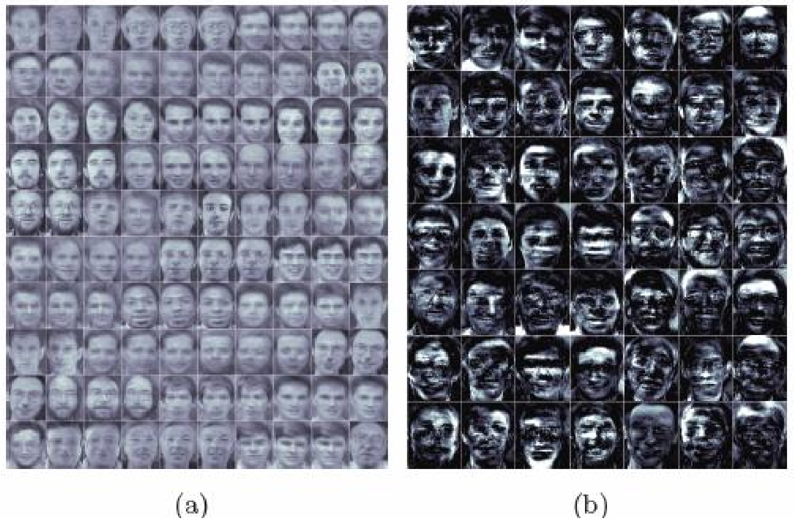


Figure 15.16 from Barber: (a): Conditional PLSA reconstruction of the images in Fig 15.5 using a positive convex combination of the 49 positive base images in (b). The root mean square reconstruction error is 1.391×10^{-5}. The base images tend to be more 'localised' than the corresponding eigen-images in Fig 15.6b. Here one sees local structure such as foreheads, chins, etc.

Appendix: Other Apps of NNMF & (Conditional) PLSA

Two further applications of NNMF and (conditional) PLSA include:

1. Modeling Citations
   - Let $d \in \{1, \ldots, D\}$ index a corpus of research articles and let $c \in \{1, \ldots, C\}$ index citations. Then d and c have the same domain, the index of a research article.
   - If document $d = i$ cites article $c = j$ then set $C_{ij} = 1$. Otherwise set $C_{ij} = 0$.
   - We can form a distribution over research articles with

$$\mathsf{P}(d = i, c = j) \;=\; \frac{C_{ij}}{\sum_{ij} C_{ij}}$$

   - Can now perform PLSA to discover citation-topics. See e.g. 15.9 in Barber.
2. Modeling the Web
   - Have a collection of web-sites indexed by i.
   - If web-site j points to web-site i then set $C_{ij} = 1$. Otherwise set $C_{ij} = 0$
     - yields a directed graph of web-site to web-site links.
   - A typical web-site will only "discuss" a small number of topics so a PLSA decomposition might be useful for determining latent topics and authoritative web-sites.

See Section 15.6.3 of Barber for further details and references.