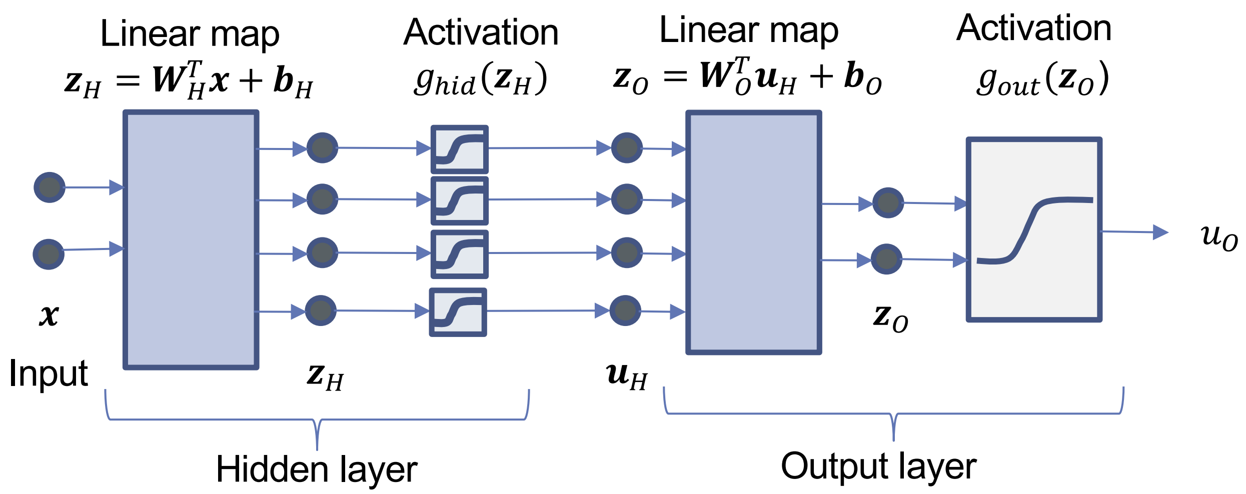
LECTURE 10: PRINCIPAL COMPONENT ANALYSIS (PCA)

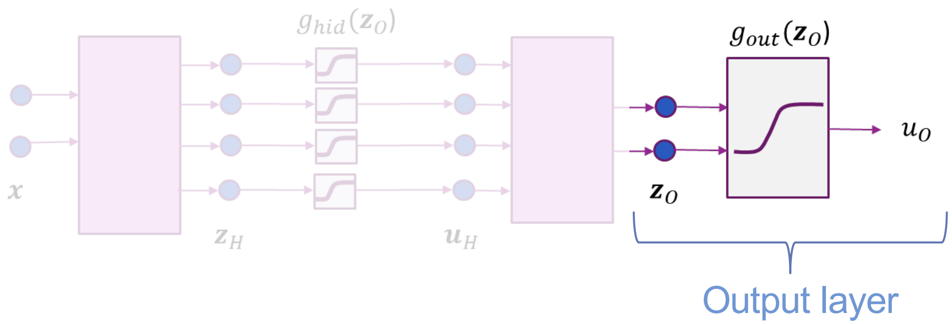
Ehsan Aryafar
earyafar@pdx.edu
http://web.cecs.pdx.edu/~aryafare/ML.html

Recall: General Neural Net Block Diagram

- Hidden layer: $\boldsymbol{z}_H = \boldsymbol{W}_H^T \boldsymbol{x} + \boldsymbol{b}_H, \quad \boldsymbol{u}_H = g_{hid}(\boldsymbol{z}_H)$
- Output layer: $\boldsymbol{z}_O = \boldsymbol{W}_O^T \boldsymbol{u}_H + \boldsymbol{b}_O, \ u_O = g_{out}(\boldsymbol{z}_O)$

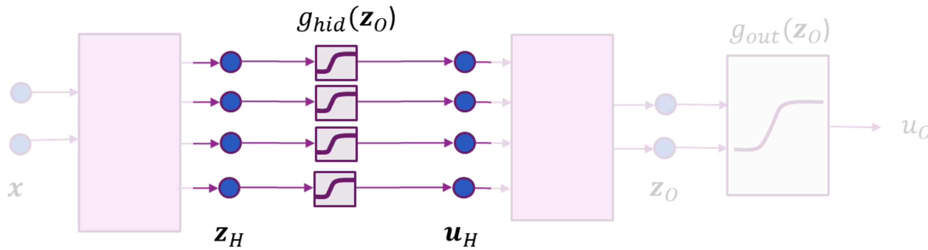


Recall: Selecting the Output Activation



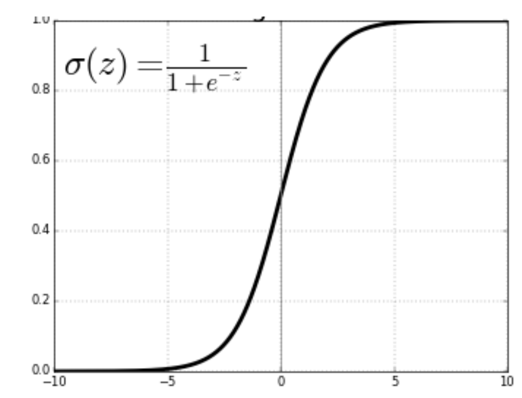
Target	**Num output units** =dim$(\boldsymbol{u}_o)$ = dim$(\boldsymbol{z}_o)$	**Output activation** $\boldsymbol{u}_O = g_{out}(\boldsymbol{z}_O)$	**Interpretation**
Binary classification	1	$u_O = \mathrm{sigmoid}(z_O)$	$u_O = P(y = 1 \mid x)$
K-class classification	K	$\boldsymbol{u}_O = \mathrm{softmax}(\boldsymbol{z}_O)$	$u_{O,k} = P(y = k \mid x)$
Regression with K outputs	K	$\boldsymbol{u}_O = \boldsymbol{z}_O$	$u_{O,k} = \hat{y}_k$

Recall: Selecting the Hidden Activation

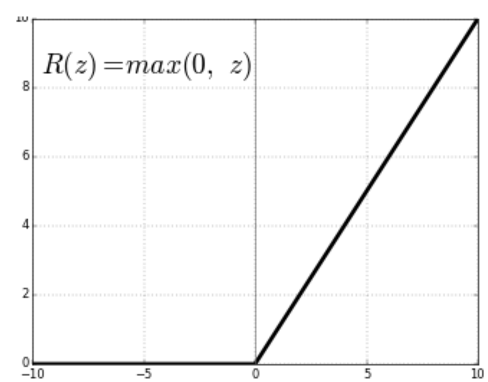


- Two common choices
- Sigmoid:
  - $u_{H,k} = 1/(1 + \exp(-z_{H,k}))$
- ReLU (Rectified linear unit):
  - $u_{H,k} = \max\{0, z_{H,k}\}$

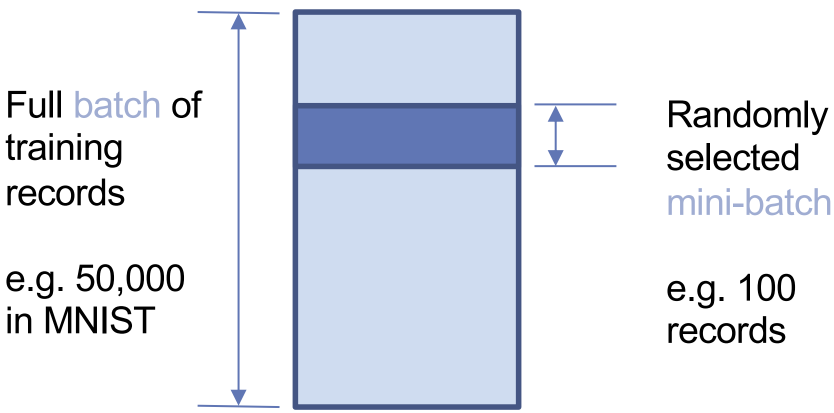
Sigmoid



ReLU



Recall: Stochastic Gradient Descent



- In each step:
  - Select random small "mini-batch"
  - Evaluate gradient on mini-batch

- For $t = 1$ to N_{steps}
  - Select random mini-batch $I \subset \{1, \ldots, N\}$
  - Compute gradient approximation (only over mini-batch samples):
    $$g^t = \frac{1}{|I|} \sum_{i \in I} \nabla L(x_i, y_i, \theta)$$
  - Update parameters:
    $$\theta^{t+1} = \theta^t - \alpha^t g^t$$

Recall: Simple MNIST Neural Network

- 784 inputs, 100 hidden units, 10 outputs

```
nin = Xtr.shape[1]  # dimension of input data
nh = 100      # number of hidden units
nout = int(np.max(ytr)+1)    # number of outputs = 10 since there are 10 classes
model = Sequential()
model.add(Dense(units=nh, input_shape=(nin,), activation='sigmoid', name='hidden'))
model.add(Dense(units=nout, activation='softmax', name='output'))
```

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
hidden (Dense)               (None, 100)               78500     
_________________________________________________________________
output (Dense)               (None, 10)                1010      
=================================================================
Total params: 79,510
Trainable params: 79,510
Non-trainable params: 0
```

Recall: Fitting the Model

- Run for 20 epochs, ADAM optimizer, batch size = 100
- Final accuracy = 0.972
- Not great, but much faster than SVM. Also, CNNs do better.

```
opt = optimizers.Adam(lr=0.001) # beta_1=0.9, beta_2=0.
model.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

```
model.fit(Xtr, ytr, epochs=10, batch_size=100, validation_data=(Xts,yts))
```

```
Epoch 7/10
50000/50000 [==============================] - 3s - loss: 0.0474 - acc: 0.9868 - val_loss: 0.0886 - val_ac
c: 0.9717
Epoch 8/10
50000/50000 [==============================] - 3s - loss: 0.0440 - acc: 0.9884 - val_loss: 0.0875 - val_ac
c: 0.9718
Epoch 9/10
50000/50000 [==============================] - 2s - loss: 0.0393 - acc: 0.9903 - val_loss: 0.0872 - val_ac
c: 0.9732
Epoch 10/10
50000/50000 [==============================] - 3s - loss: 0.0381 - acc: 0.9901 - val_loss: 0.0875 - val_ac
c: 0.9718
```

Recall: Initialization and Data Normalization

- Solution by gradient descent algorithm depends on the initial weights
- Typically, weights are set to random values near zero.
- Small weights make the network behave like linear classifier.
  - Hence model starts out nearly linearly
  - Becomes nonlinear as weights increase during the training process.
- Starting with large weights often lead to poor results.
- **Normalizing data to zero mean and unit variance**
  - **Allows all input dimensions be treated equally and facilitate better convergence.**
- With normalized data, it is typical to initialize the weights to be uniform in [-0.7, 0.7] [ESL]

Recall: Regularization

- To avoid the weights get too large, can add a penalty term explicitly, with regularization level λ
- Ridge penalty

$$R(\theta) = \sum_{d,m} w_{H,d,m}^2 + \sum_{m,k} w_{O,m,k}^2 = \|w_H\|^2 + \|w_O\|^2$$

- Total loss

$$L_{reg}(\theta) = L(\theta) + \lambda R(\theta)$$

- Change in gradient calculation
- Typically used regularization
  - L2 = Ridge: Shrink the sizes of weights
  - L1: Prefer sparse set of weights
  - L1-L2: use a combination of both

Recall: Regularization in Keras

- `kernel_regularizer` : instance of `keras.regularizers.Regularizer`
- `bias_regularizer` : instance of `keras.regularizers.Regularizer`
- `activity_regularizer` : instance of `keras.regularizers.Regularizer`

Example

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01),
                activity_regularizer=regularizers.l1(0.01)))
```

Available penalties

```
keras.regularizers.l1(0.)
keras.regularizers.l2(0.)
keras.regularizers.l1_l2(0.)
```

Activity regularization tries to make the output at each layer small or sparse.

Learning Objectives

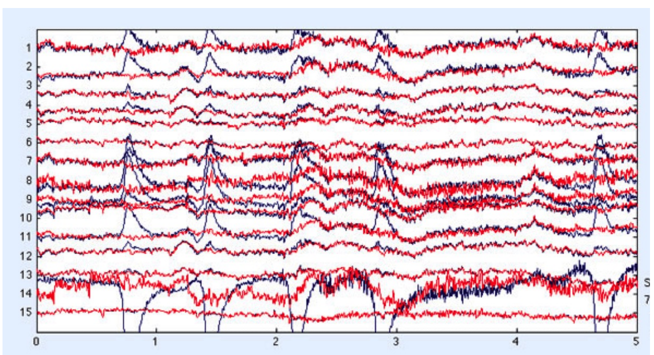
- Identify cases to use dimensionality reduction
- Mathematically describe principal components representations of data
- Compute principal components via SVDs
- Compute PC components in python
- Add PCA transforms as a pre-processing step to classification and regression
- Implement low-rank transforms for recommender systems

Outline

- Why dimensionality reduction?
- Principal components and directions of variance
- Approximation with PCs
- Computing PCs via the SVD
- Face example in python
- Training models from PCs
- Low rank approximations and recommender systems

High-Dimensional Data

- Many data sets have very high dimension
- Training can be difficult
  - Especially when number of samples is small
  - Classifier needs many parameters

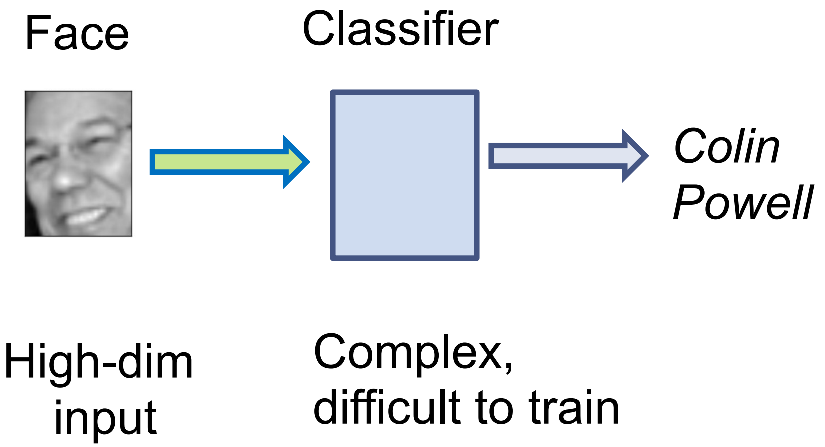


EEG
Ex: 32 channels x 1 kHz x 10s

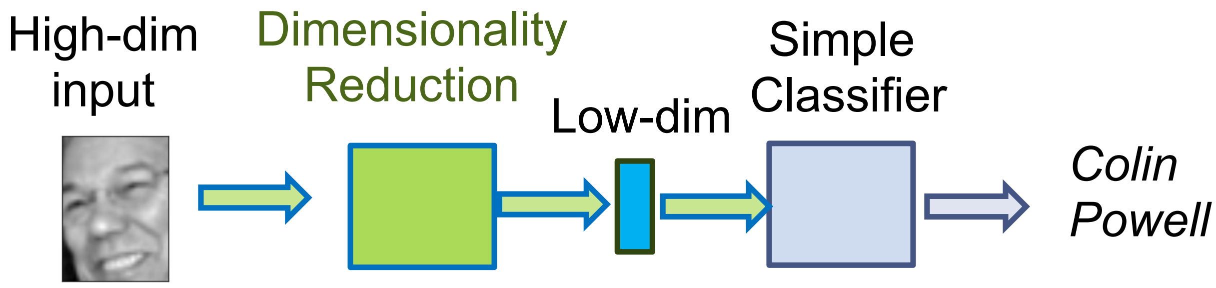
Face recognition with high-resolution images

Problems with High-Dimensions

- Consider face recognition
- Input is high-dimensional
  - Esp. for high resolution image
- Resulting classifier:
  - Requires many parameters
  - Difficult to train
  - Needs many samples
  - Computationally complex



Dimensionality Reduction



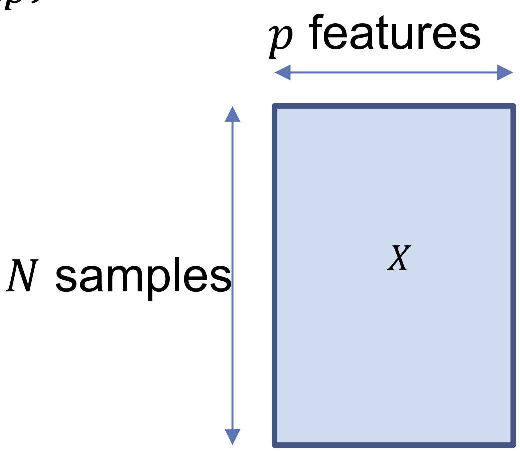
- Dimensionality reduction:
  - Reduce the input dimension to lower dimensional representation
- Can build simpler classifier
- Low-dimensional representational also good for:
  - Visualizing data
  - Clustering and other unsupervised tasks
  - Finding underlying structure of the data

Outline

- Dimensionality reduction
- Principal components and directions of variance
- Approximation with PCs
- Computing PCs via the SVD
- Face example in python
- Training models from PCs
- Low rank approximations and recommender systems

Data Definitions

- Given data: $\boldsymbol{x}_i, i = 1, \ldots, N$
  - Each sample has p features: $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{ip})$
  - Represent as an $N \times p$ matrix



- Unsupervised learning
  - Samples do not have a label
  - Or we choose to ignore the label for now

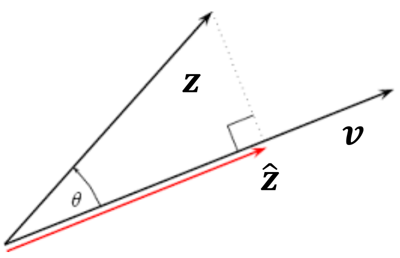
- Dimension p is large
- How do we reduce the dimension?

Projections

- PCA reduces dimensionality by “projecting” data to a lower dim subspace
- Projection: Given vectors $\boldsymbol{z}$ and $\boldsymbol{v}$, the projection of $\boldsymbol{z}$ onto $\boldsymbol{v}$ is:

$$\hat{\boldsymbol{z}} = \mathrm{Proj}_{\boldsymbol{v}}(\boldsymbol{z}) = \alpha \boldsymbol{v}, \qquad \alpha = \frac{\boldsymbol{v}^T \boldsymbol{z}}{\boldsymbol{v}^T \boldsymbol{v}} = \frac{||\boldsymbol{z}||}{||\boldsymbol{v}||} \cos\theta$$

  - $\alpha =$ coefficient of the projection



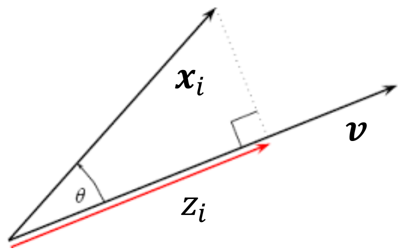
- Theorem: $\mathrm{Proj}_{\boldsymbol{v}}(\boldsymbol{z})$ is closest point in V to $\boldsymbol{z}$:

$$\hat{\boldsymbol{z}} = \arg\min_{\boldsymbol{w} \in V} ||\boldsymbol{z} - \boldsymbol{w}||^2$$

  - $V = \{\alpha \boldsymbol{v} | \alpha \in R\}$ = vectors on the line spanned by $\boldsymbol{v}$

Maximal Directional Variance

- Given data: $\boldsymbol{x}_i, i = 1, \ldots, N$ and direction $\boldsymbol{v}$ with $||\boldsymbol{v}|| = 1$
- Let $z_i = \boldsymbol{v}^T \boldsymbol{x}_i$ = coefficient of the projection of $\boldsymbol{x}_i$ onto $\boldsymbol{v}$
- Sample mean and variance in direction $\boldsymbol{v}$ is :
  - Sample mean $\bar{z} = \frac{1}{N} \sum_{i=1}^{N} z_i$
  - Sample variance $s_z^2 = \frac{1}{N} \sum_{i=1}^{N} (z_i - \bar{z})^2$



Problem: Find the direction $\boldsymbol{v}$ that maximizes the variance s_z^2

- Why?
  - Captures the most variation of the data
  - Provides the best vector for dimensionality reduction

Sample Covariance Matrix

- Sample mean of the data: $\bar{\boldsymbol{x}} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_i$
- Sample covariance matrix: Matrix $\boldsymbol{Q}$ with components:

$$Q_{k\ell} = \frac{1}{N}\sum_{i=1}^{N}(x_{ik} - \bar{x}_k)(x_{i\ell} - \bar{x}_\ell)$$

  - Covariance between feature k and ℓ in the dataset
  - Matrix is $p \times p$
- Sample covariance is given by

$$\boldsymbol{Q} = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{x}_i - \bar{\boldsymbol{x}})(\boldsymbol{x}_i - \bar{\boldsymbol{x}})^T = \frac{1}{N}\widetilde{\boldsymbol{X}}^T\widetilde{\boldsymbol{X}}$$

  - $\widetilde{\boldsymbol{X}}$ = data matrix with sample mean removed (rows: $\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i - \bar{\boldsymbol{x}}$)
  - Compute sample covariance via a matrix product

Sample Covariance and Directional Variance

- Let $z_i = \boldsymbol{v}^T \boldsymbol{x}_i$ = coefficient of the projection of $\boldsymbol{x}_i$ onto $\boldsymbol{v}$
- Can compute the sample mean and variance of z_i from $\overline{\boldsymbol{x}}$ and $\boldsymbol{Q}$
- Sample mean of the coefficients:

$$\bar{z} = \frac{1}{N}\sum_{i=1}^{N} z_i = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{v}^T \boldsymbol{x}_i = \boldsymbol{v}^T \left[\frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_i\right] = \boldsymbol{v}^T \overline{\boldsymbol{x}}$$

- Sample variance of the coefficients:

$$\begin{aligned} s_z^2 &= \frac{1}{N}\sum_{i=1}^{N} (z_i - \bar{z})^2 = \frac{1}{N}\sum_{i=1}^{N} (\boldsymbol{v}^T (\boldsymbol{x}_i - \overline{\boldsymbol{x}}))^2 \\ &= \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{v}^T (\boldsymbol{x}_i - \overline{\boldsymbol{x}})(\boldsymbol{x}_i - \overline{\boldsymbol{x}})^T \boldsymbol{v} \\ &= \boldsymbol{v}^T \boldsymbol{Q} \boldsymbol{v} \end{aligned}$$

Maximizing Directional Variance

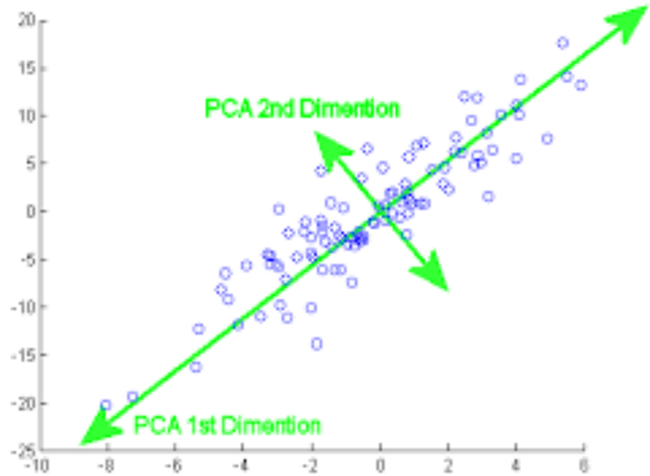
- From previous slide: Directional variance $s_z^2 = \frac{1}{N}\sum_{i=1}^{N}(z_i - \bar{z})^2 = \boldsymbol{v}^T\boldsymbol{Q}\boldsymbol{v}$
- Maximizing directional variance can be formulated as an optimization problem:

$$\max_{\boldsymbol{v}} \boldsymbol{v}^T\boldsymbol{Q}\boldsymbol{v} \quad \text{s.t.} \quad \|\boldsymbol{v}\| = 1$$

- Let $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p$ be the eigenvectors of $\boldsymbol{Q}$: $\boldsymbol{Q}\boldsymbol{v}_j = \lambda_j \boldsymbol{v}_j$
- Sort eigenvalues in descending order: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_p$
  - Can show that eigenvalues are real and non-negative
- Theorem: Any local maxima of the variance directional is an eigenvector
  - $\boldsymbol{v} = \boldsymbol{v}_j$ for some j and $\boldsymbol{v}^T\boldsymbol{Q}\boldsymbol{v} = \lambda_j$
  - Proof below

Visualizing Principal Components

- Principal components: The eigenvectors of $\boldsymbol{Q}$, $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p$
  - Always normalized $\|\boldsymbol{v}_j\| = 1$
  - Sorted by eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_p$
  - Each vector is of dimension p
- Key property: Vectors are orthogonal
  - $\boldsymbol{v}_j^T \boldsymbol{v}_k = 0$ if $j \neq k$



- Represents directions of decreasing variance:
  - $\boldsymbol{v}_1$: PC 1 = Direction of max variance
  - $\boldsymbol{v}_2$: PC 2 = Direction of second most variance
  - $\boldsymbol{v}_3$: PC 3 = Direction of third most variance
  - …

Proof PCs = Eigenvectors of $\boldsymbol{Q}$ (Advance Concept)

- PC constrained optimization problem:

$$\max_{\boldsymbol{v}} \boldsymbol{v}^{\boldsymbol{T}} \boldsymbol{Q} \boldsymbol{v} \quad \text{s.t. } \|\boldsymbol{v}\| = 1$$

- Define Lagrangian: $L(\boldsymbol{v}, \lambda) = \boldsymbol{v}^{\boldsymbol{T}} \boldsymbol{Q} \boldsymbol{v} - \lambda[\|\boldsymbol{v}\|^2 - 1\ ]$
- At any local maxima:

$$\frac{\partial L}{\partial v} = 0 \Rightarrow \boldsymbol{Q}\boldsymbol{v} - \lambda \boldsymbol{v} = \boldsymbol{0}$$

- This shows that $\boldsymbol{v}$ is an eigenvector of $\boldsymbol{Q}$

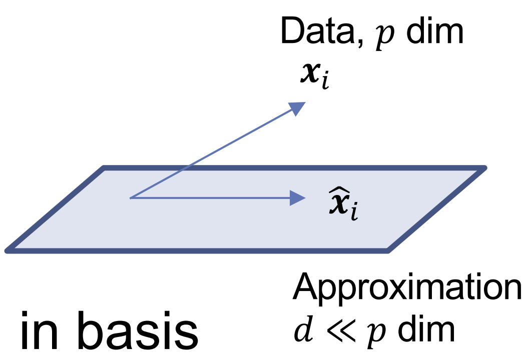
Outline

- Dimensionality reduction
- Principal components and directions of variance
- Approximation with PCs
- Computing PCs via the SVD
- Face example in python
- Training models from PCs
- Low rank approximations and recommender systems

Low-Dimensional Representations

- Given data $\boldsymbol{x}_i, i = 1, \dots, N$. Each $\boldsymbol{x}_i \in \mathbb{R}^p$
- Problem: Find basis vectors $\boldsymbol{v}_j, j = 1, \dots, d$ such that:

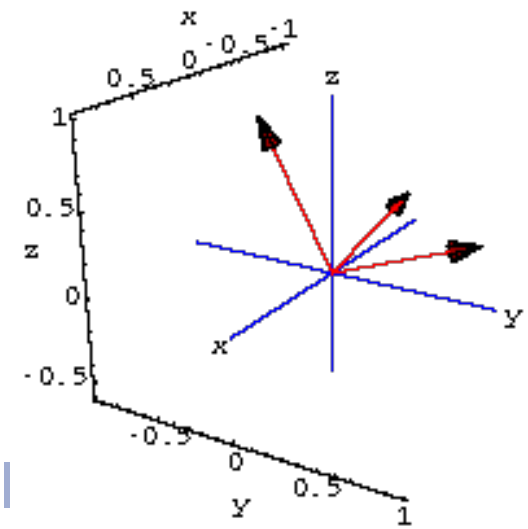
$$\boldsymbol{x}_i \approx \widehat{\boldsymbol{x}}_i = \overline{\boldsymbol{x}} + \sum_{j=1}^{d} \alpha_{ij} \boldsymbol{v}_j$$



  - Sample mean + linear combination of basis vectors
  - $\alpha_i = (\alpha_{i1}, \dots, \alpha_{id})$ is an approximate coordinates of $\boldsymbol{x}_i$ in basis $(\boldsymbol{v}_1, \dots, \boldsymbol{v}_d)$
- Dimensionality reduction:
  - If $d \ll p$ we have represented $\boldsymbol{x}_i$ with a smaller number of coefficients.

Orthonormal Sets and Basis

- Definition: A set of vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d$ are an orthonormal set if:
  - $\|\boldsymbol{v}_j\| = 1$ for all j (unit length)
  - $\boldsymbol{v}_j^T \boldsymbol{v}_k = 0$ if $j \neq k$ (perpendicular to one another)



- Matrix form: If $\boldsymbol{V} = [\boldsymbol{v}_1 \ \ldots \boldsymbol{v}_d]$, then $\boldsymbol{V}^T\boldsymbol{V} = I_d$
- If $d = p$ then $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p$ is called an orthonormal basis
  - $\boldsymbol{V}$ is an orthonormal matrix
- Key property: the PCs form an orthonormal basis

Coefficients in an Orthonormal Basis

- Suppose $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p$ is an orthonormal basis
- Given a vector $\boldsymbol{z}$, can write

$$\boldsymbol{z} = \sum_{j=1}^{p} \alpha_j \boldsymbol{v}_j \,, \qquad \alpha_j = \boldsymbol{v}_j^T \boldsymbol{z}$$

  - Simple expression for computing coefficients in an orthonormal basis
- Matrix form:

$$\boldsymbol{\alpha} = \boldsymbol{V}^T \boldsymbol{z}, \qquad \boldsymbol{z} = \boldsymbol{V} \alpha$$

Approximating the Data Matrix

- Given data $\boldsymbol{x}_i, i = 1, \dots, N$
- Let $\boldsymbol{v}_1, \dots, \boldsymbol{v}_p$ be the PCs
- Find coefficient expansion of each data sample:

$$\boldsymbol{x}_i = \overline{\boldsymbol{x}} + \sum_{j=1}^{p} \alpha_{ij} \boldsymbol{v}_j \,, \qquad \alpha_{ij} = \boldsymbol{v}_j^T (\boldsymbol{x}_i - \overline{\boldsymbol{x}})$$

- Approximation with d coefficients:

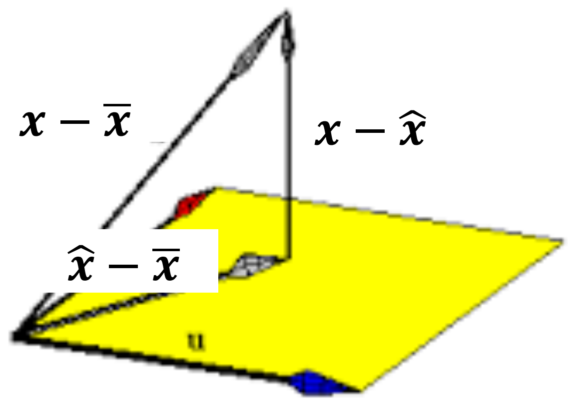
$$\widehat{\boldsymbol{x}}_i = \overline{\boldsymbol{x}} + \sum_{j=1}^{d} \alpha_{ij} \boldsymbol{v}_j$$

Geometry of Approximations

- Approximation can be interpreted geometrically
- Let V be set of all linear combinations

$$\sum_{j=1}^{d} \alpha_j \boldsymbol{v}_j$$

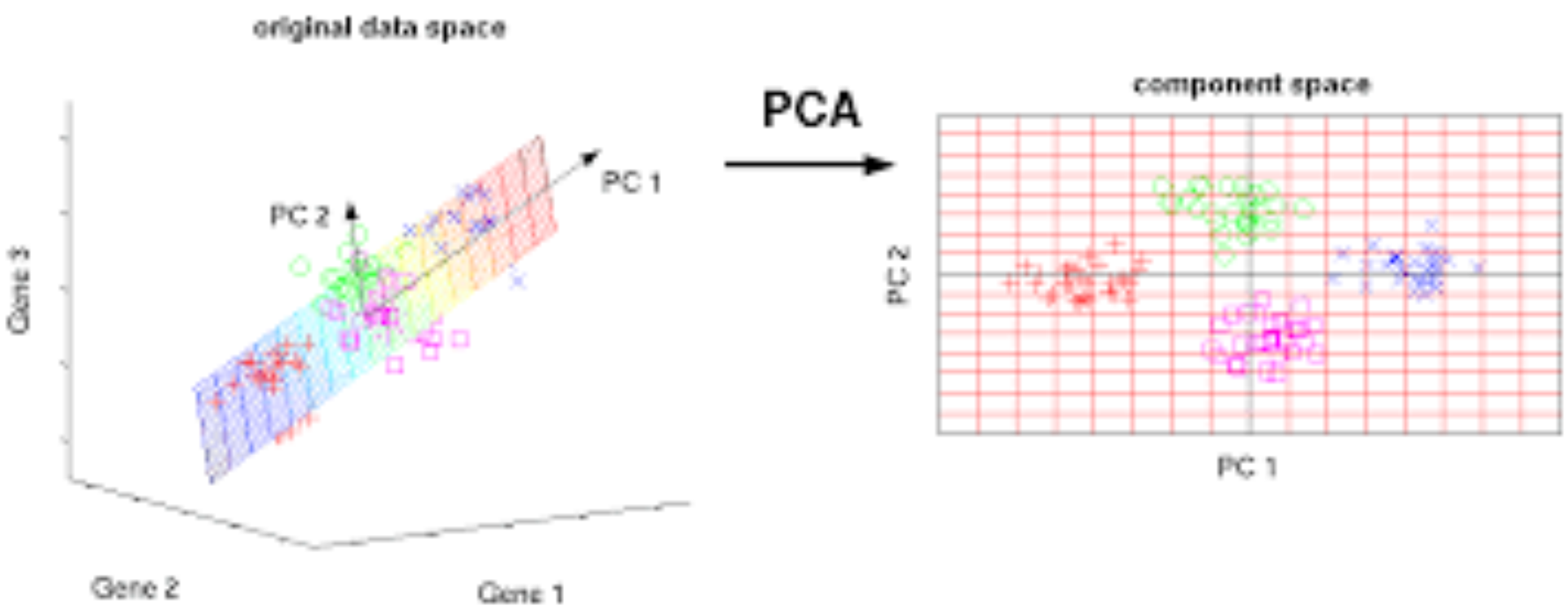
  - V is a vector space
  - Called the span of $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d$
- $\hat{\boldsymbol{x}} - \overline{\boldsymbol{x}}$ is the closest vector in V to $\boldsymbol{x} - \overline{\boldsymbol{x}}$
  - Note the subtraction of the mean



Space spanned by $v_1, \ldots, v_d$

Visualizing the Representation

- Finds a low-dimensional representation



Example Calculation

- Problem:
  - Let $\boldsymbol{v}_1 = \frac{1}{\sqrt{2}}[1,1,0]$, $\boldsymbol{v}_2 = \frac{1}{\sqrt{6}}[1,-1,2]$
  - Show v_1 and v_2 are orthogonal
- Solution:
  - $\boldsymbol{v}_1^T \boldsymbol{v}_1 = \frac{1}{2}(1^2 + 1^2 + 0^2) = 1$
  - $\boldsymbol{v}_2^T \boldsymbol{v}_2 = \frac{1}{6}(1^2 + (-1)^2 + 2^2) = 1$
  - $\boldsymbol{v}_1^T \boldsymbol{v}_2 = \frac{1}{\sqrt{2(3)}}(1(1) + 1(-1) + 0(2)) = 0$

Example Calculation Continued

- Problem:
  - Let $\boldsymbol{v}_1 = \frac{1}{\sqrt{2}}[1,1,0]$, $\boldsymbol{v}_2 = \frac{1}{\sqrt{6}}[1,-1,2]$ be two PCs
  - Let $\overline{\boldsymbol{x}} = [0,1,2]$ be the mean of the data
  - Find the approximation of data record $\boldsymbol{x} = [2,4,4]$ with the two PCs
- Solution:
  - Subtract mean: $\boldsymbol{x} - \overline{\boldsymbol{x}} = [2,3,2]$
  - Coeff on PC1: $\alpha_1 = \boldsymbol{v}_1^T(\boldsymbol{x} - \overline{\boldsymbol{x}}) = \frac{1}{\sqrt{2}}[2+3+0] = \frac{5}{\sqrt{2}}$
  - Coeff on PC2: $\alpha_2 = \boldsymbol{v}_2^T(\boldsymbol{x} - \overline{\boldsymbol{x}}) = \frac{1}{\sqrt{6}}[2-3+4] = \frac{3}{\sqrt{6}}$
  - Approximation: $\hat{\boldsymbol{x}} = \overline{\boldsymbol{x}} + \sum_{j=1}^{d} \alpha_j \boldsymbol{v}_j = [0,1,2] + \frac{5}{2}[1,1,0] + \frac{3}{6}[1,-1,2] \approx [3,3,3]$

Average Approximation Error

- Let $\hat{\boldsymbol{x}}_i$ = approximation with d PCs
- Error in sample i:

$$\boldsymbol{x}_i - \hat{\boldsymbol{x}}_i = \sum_{j=d+1}^{p} \alpha_{ij} \boldsymbol{v}_j$$

- Theorem: Average error with a d PC approximation is:

$$\frac{1}{\mathrm{N}} \sum_{i=1}^{N} \|\boldsymbol{x}_i - \hat{\boldsymbol{x}}_i\|^2 = \sum_{j=d+1}^{p} \lambda_j$$

  - Sum of the smallest $p - d$ eigenvalues

Proportion of Variance (PoV)

- Total variance of data set:

$$\frac{1}{\mathrm{N}}\sum_{i=1}^{N}\|\boldsymbol{x}_i - \overline{\boldsymbol{x}}\|^2 = \sum_{j=1}^{p}\lambda_j$$

- Average approximation error:

$$\frac{1}{\mathrm{N}}\sum_{i=1}^{N}\|\boldsymbol{x}_i - \widehat{\boldsymbol{x}}_i\|^2 = \sum_{j=d+1}^{p}\lambda_j$$

- The proportion of variance explained by d PCs is:

$$PoV(d) = \frac{\sum_{j=1}^{d}\lambda_j}{\sum_{j=1}^{p}\lambda_j}$$

  - Measure of approximation error in using d PCs

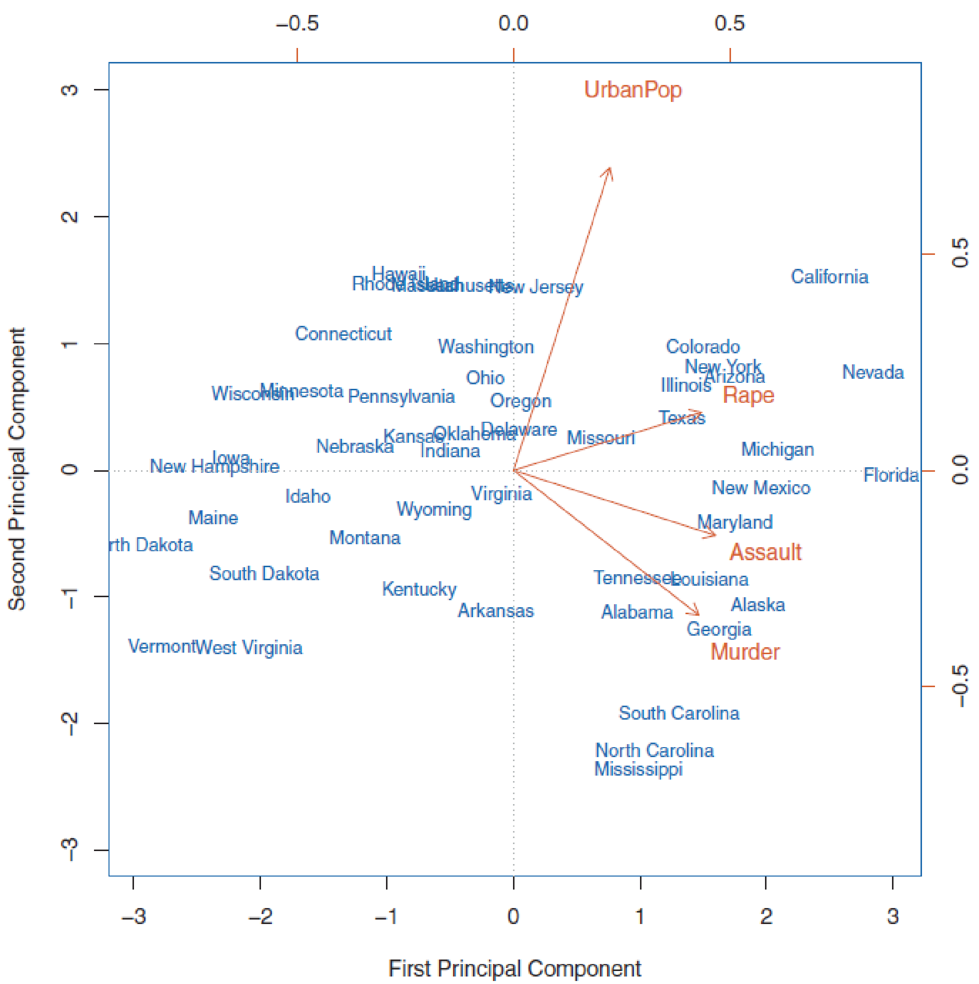
Example: suppose dataset with four dimensions

PC index	λ_i	POV(i)
1	10	10/14.3≈ 0.70
2	4	14/14.3≈ 0.98
3	0.2	14.2/14.3$\approx$ 0.99
4	0.1	14.3/14.3$= 1$

Latent Representations

- Each record is of the form: $\boldsymbol{x}_i \approx \overline{\boldsymbol{x}} + \sum_{j=1}^{d} \alpha_{ij} \boldsymbol{v}_j$
- Variance in $\boldsymbol{x}_i$ explained by small number of "latent components"
  - Coefficients α_{ij} are the latent representations of $\boldsymbol{x}_i$
- Example:
  - $\boldsymbol{x}_i$ = list of movie preferences for customer i
  - Movie preferences are highly correlated.
  - Could be explained by small number of components (action, romance, presence of stars, …)
  - PCA can be used to find these out

Example: USArrests

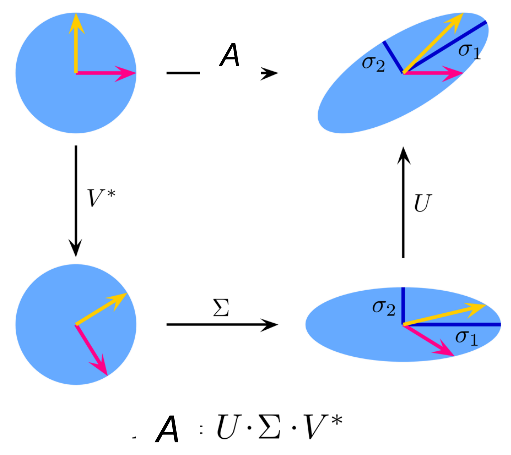


- Arrests per capita in four categories
  - One record per US state
- Visualize PCA in a biplot
  - See the scores (i.e. coefficients of each state)
  - Overlay loading plot (PC vectors)
- Fig from ESL 10.1

Outline

- Dimensionality reduction
- Principal components and directions of variance
- Approximation with PCs
- Computing PCs via the SVD
- Face example in python
- Training models from PCs
- Low rank approximations and recommender systems

Singular Value Decomposition



- SVD: Powerful method in linear algebra

- Given a matrix A:
  - Decomposes the matrix into a product: $A = USV^T$
  - Provides orthonormal bases of the input and output spaces
  - Multiplication of A is equivalent to scaling in that basis

- For PCA:
  - Identifies low rank subspaces for data
  - Computes coefficients in that subspace

Singular Value Decomposition Defined

- Given matrix $\boldsymbol{A} \in \mathbb{F}^{n \times d}$ (F is for field, just consider R)
  - For PCA, this will be a scaled version of the data matrix

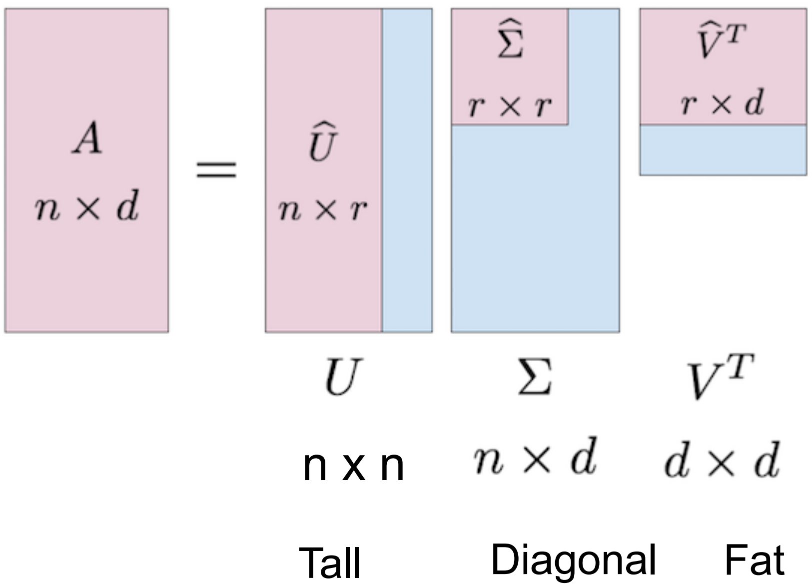
- SVD is $\boldsymbol{A} = \boldsymbol{U\Sigma V}^T$, where
  - $\boldsymbol{U} \in \mathbb{F}^{n \times r}$, columns are orthonormal
  - $\boldsymbol{V} \in \mathbb{F}^{d \times r}$, columns are orthonormal
  - $\boldsymbol{\Sigma} = \mathrm{diag}(s_1, \ldots, s_r)$, sorted $s_1 \geq s_2 \geq \cdots \geq s_r \geq 0$.
  - Called the singular values
- All matrices have an SVD
  - Matrices do not have to be square.
- Number of singular values $r \leq \min(n, d)$

Economy vs. Full SVD

- Suppose $\boldsymbol{A} \in \mathbb{R}^{n\times d}$ with rank $r \leq \min\{n, d\}$
- Two types of SVDs
- Economy SVD: $A = USV^*$
  - $U \in \mathbb{F}^{n\times r}$, columns are orthonormal
  - $V \in \mathbb{F}^{d\times r}$, columns are orthonormal
  - $\Sigma \in \mathbb{F}^{r\times r}$ diagonal $\Sigma = diag(s_1, \dots, s_r)$,
- Full SVD: $A = USV^*$
  - $U \in \mathbb{F}^{n\times n}$, columns are an orthonormal basis of $\mathbb{R}^n$
  - $V \in \mathbb{F}^{d\times d}$, columns are an orthonormal basis of $\mathbb{R}^d$
  - $\Sigma \in \mathbb{F}^{n\times d}$ with diagonal upper left $\Sigma = \begin{bmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{bmatrix}$

SVD Visualized

- Pink matrices represent “economy” SVD
- Blue represent “full SVD”



Example

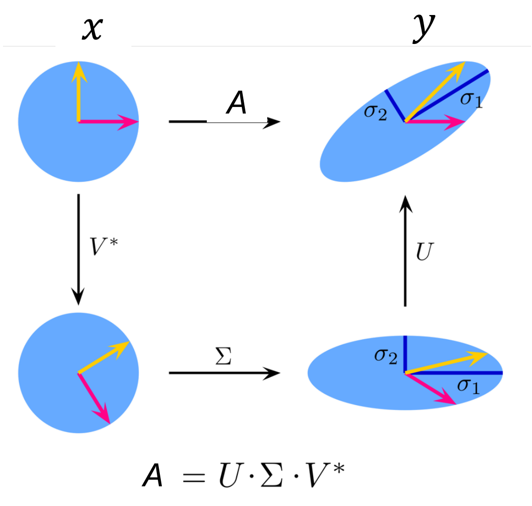
- Let $A = \begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \end{bmatrix}$
- Then can check that $A = U\Sigma V^*$

$$\mathbf{U} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \boldsymbol{\Sigma} = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{V}^* = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \sqrt{0.2} & 0 & 0 & 0 & \sqrt{0.8} \\ 0 & 0 & 0 & 1 & 0 \\ -\sqrt{0.8} & 0 & 0 & 0 & \sqrt{0.2} \end{bmatrix}$$

  - Also verify that $UU^* = I_5$ and $VV^* = I_5$
  - But, in general, use a computer to compute SVD

Geometric Interpretation (Advanced)

- Let $A = U\Sigma V^*$ and $y = Ax$
- Consider a transformed space
  - $\boldsymbol{w} = \boldsymbol{V}^*\boldsymbol{x} = [w_1, \dots, w_N]$ coefficients in input basis $V = [v_1, \dots, v_N]$
  - $\boldsymbol{z} = \boldsymbol{U}^*\boldsymbol{y} = [z_1, \dots, z_M]$: coefficients in output basis $U = [u_1, \dots, u_M]$
- Then: $\boldsymbol{z} = \boldsymbol{\Sigma w}$ so $z_i = \sigma_i w_i$
- Each input direction $\boldsymbol{v}_i$ is mapped to $\sigma_i \boldsymbol{u}_i$
- Consequence:
  - SVD finds orthonormal bases U, V such that matrix A is a linear scaling in each basis vector



Example Problem

- Suppose that $A = U\Sigma V^* \in \mathbb{R}^{3\times 4}$ with $\Sigma = diag(3,0.2,0,0)$
- If $\boldsymbol{x} = 2\boldsymbol{v}_1 + 3\boldsymbol{v}_2 + 4\boldsymbol{v}_3 + 5\boldsymbol{v}_4$ find $y = Ax$ in terms of basis u_1, u_2, u_3
- Solution:
  - $\boldsymbol{Av}_i = \sigma_i \boldsymbol{u}_i$ for all i
  - Therefore,

$$\begin{aligned} \boldsymbol{y} = \boldsymbol{Ax} &= 2\boldsymbol{Av}_1 + 3\boldsymbol{Av}_2 + 4\boldsymbol{Av}_3 + 5\boldsymbol{Av}_4 \\ &= 2(3)\boldsymbol{u}_1 + 3(0.2)\boldsymbol{u}_2 + 4(0)\boldsymbol{u}_3 \\ &= 6\boldsymbol{u}_1 + 0.6\,\boldsymbol{u}_2 \end{aligned}$$

Computing the SVD in Python

- Random matrix

```
# Create some random matrix
A = np.random.normal(0,1,(100,10))
```

- Full SVD

```
# Full SVD
U,s,Vtr = np.linalg.svd(A)
```

```
A.shape   = (100, 10)
U.shape   = (100, 100)
s.shape   = (10,)
Vtr.shape = (10, 10)
```

- Economy SVD

```
# Economy SVD
U,s,Vtr = np.linalg.svd(A, full_matrices=False)
```

```
U.shape   = (100, 10)
s.shape   = (10,)
Vtr.shape = (10, 10)
```

- Reconstruction:

```
# Recovers back A
Ahat = (U*s[None,:]).dot(Vtr)
```

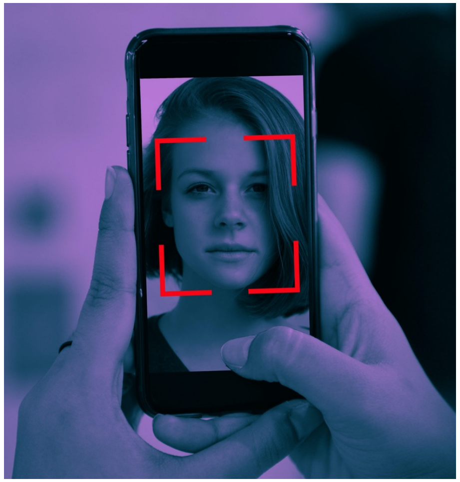
Computing the PCA via SVD

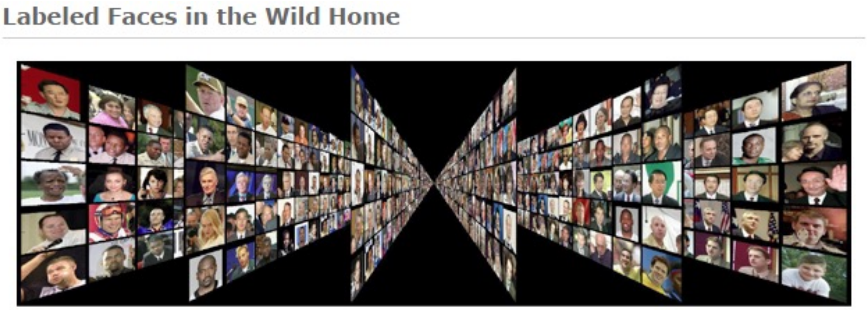
- Let $\boldsymbol{A} = \frac{1}{\sqrt{N}}\widetilde{\boldsymbol{X}}$ = scaled data matrix with sample mean removed.
- Take SVD: $\boldsymbol{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$
- Properties:
  - Sample covariance matrix is $\boldsymbol{Q} = \frac{1}{N}\widetilde{\boldsymbol{X}}^{\boldsymbol{T}}\widetilde{\boldsymbol{X}} = \boldsymbol{A}^{\boldsymbol{T}}\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Sigma}\mathbf{U}^T\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T = \boldsymbol{V}\boldsymbol{\Sigma}^2\boldsymbol{V}^T$
  - Eigenvalues of $\mathbf{Q}$ = squared singular values of $\boldsymbol{A}$
  - PCs are $\boldsymbol{v}_j$, columns of $\boldsymbol{V}$
  - Coefficients are $\boldsymbol{Z} = \widetilde{\boldsymbol{X}}\boldsymbol{V} = \sqrt{N}\boldsymbol{A}\boldsymbol{V} = \sqrt{N}\boldsymbol{U}\boldsymbol{\Sigma}$
- Hence, SVD provides PCs, eigenvalues coefficients, $\boldsymbol{Z}$ in the PCA representation.

Outline

- Dimensionality reduction
- Principal components and directions of variance
- Approximation with PCs
- Computing PCs via the SVD
- Face recognition using PCA in python
- Training models from PCs
- Low rank approximations and recommender systems

Example: Face Recognition





- Face recognition challenges:
  - Face images can be high-dimensional
  - We will use 50 x 37 = 1850 pixels
- Applying PCA:
  - Should be few degrees of freedom
  - Can transform to lower dimensional representations
- Data Labelled Faces in the Wild project
  - http://vis-www.cs.umass.edu/lfw
  - Large collection of faces (13000 images)
  - Taken from web articles about 20 years ago

Loading the Data

- Built-in routines to load data from sk-learn package
- Can take several minutes the first time (Be patient)

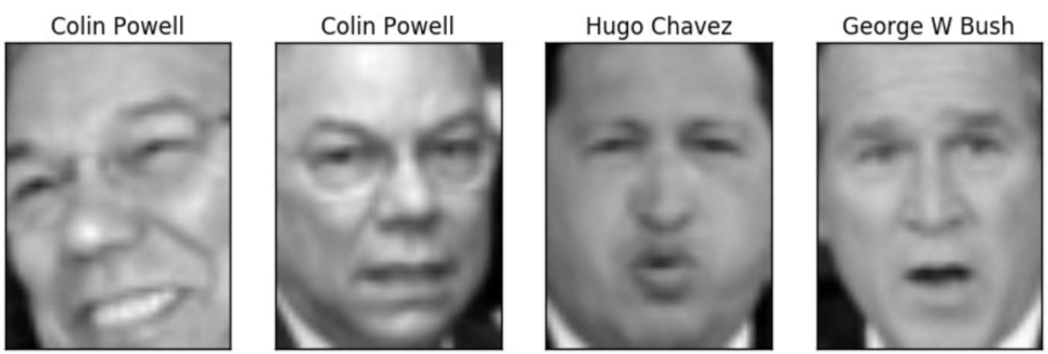
```
from sklearn.datasets import fetch_lfw_people
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)
```

```
2016-11-14 14:15:30,862 Downloading LFW metadata: http://vis-www.cs.umass.edu/lfw/pairsDevTrain.txt
2016-11-14 14:15:30,958 Downloading LFW metadata: http://vis-www.cs.umass.edu/lfw/pairsDevTest.txt
2016-11-14 14:15:31,028 Downloading LFW metadata: http://vis-www.cs.umass.edu/lfw/pairs.txt
2016-11-14 14:15:31,294 Downloading LFW data (~200MB): http://vis-www.cs.umass.edu/lfw/lfw-funneled.tgz
```

```
Image size     = 50 x 37 = 1850 pixels
Number faces   = 1288
Number classes = 7
```

Plotting the Data

- Some example faces
- You may be too young to remember them all




```
def plt_face(x):
    h = 50
    w = 37
    plt.imshow(x.reshape((h, w)), cmap=plt.cm.gray)
    plt.xticks([])
    plt.yticks([])

I = np.random.permutation(n_samples)
plt.figure(figsize=(10,20))
nplt = 4;
for i in range(nplt):
    ind = I[i]
    plt.subplot(1,nplt,i+1)
    plt_face(X[ind])
    plt.title(target_names[y[ind]])
```

Computing the PCA

```
npix = h*w
Xmean = np.mean(X,0)
Xs = X - Xmean[None,:]
```

```
U,S,Vtr = np.linalg.svd(Xs, full_matrices=False)
```

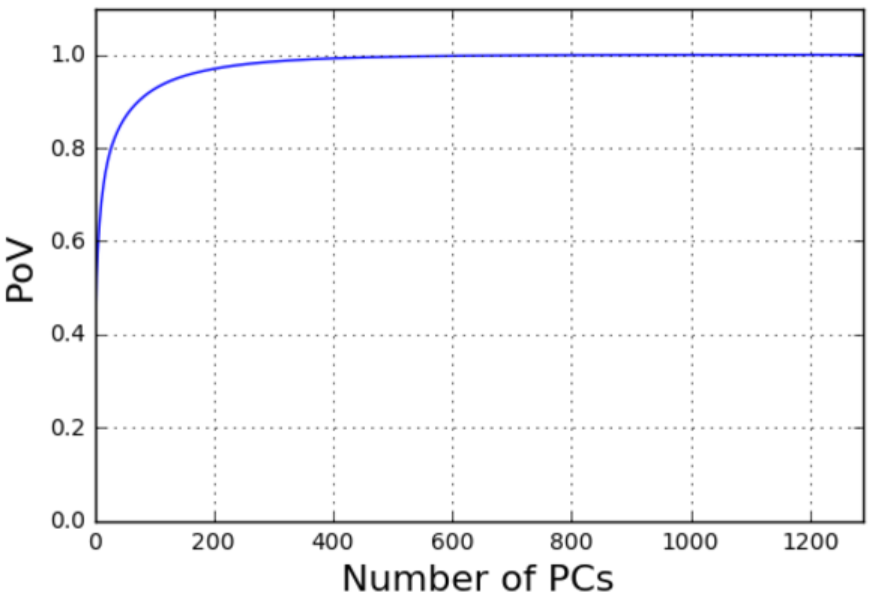
```
from sklearn.decomposition import PCA

# Construct the PCA object
pca = PCA(n_components=ncomp,
          svd_solver='randomized', whiten=True)

# Fit the PCA components on the entire dataset
pca.fit(X)
```

- Manually compute the PCs with SVD
  - Remove the mean
  - Use broadcasting
  - Compute the SVD
- Use sklearn builtin PCA function
  - Construct a PCA object
  - Call fit: Computes mean and PC components
  - Stores values internally in the pca class

Finding the PoV



- Most variance explained in about 400 components
- Some reduction

```
lam = S**2
PoV = np.cumsum(lam)/np.sum(lam)

plt.plot(PoV)
plt.grid()
plt.axis([1,n_samples,0, 1.1])
plt.xlabel('Number of PCs', fontsize=16)
plt.ylabel('PoV', fontsize=16)
```

Plotting Approximations

```
nplt = 2                  # number of faces to plot
ds = [0,5,10,20,100]      # number of SVD approximations
use_pca = True            # True=Use sklearn reconstruction, else use SVD
```

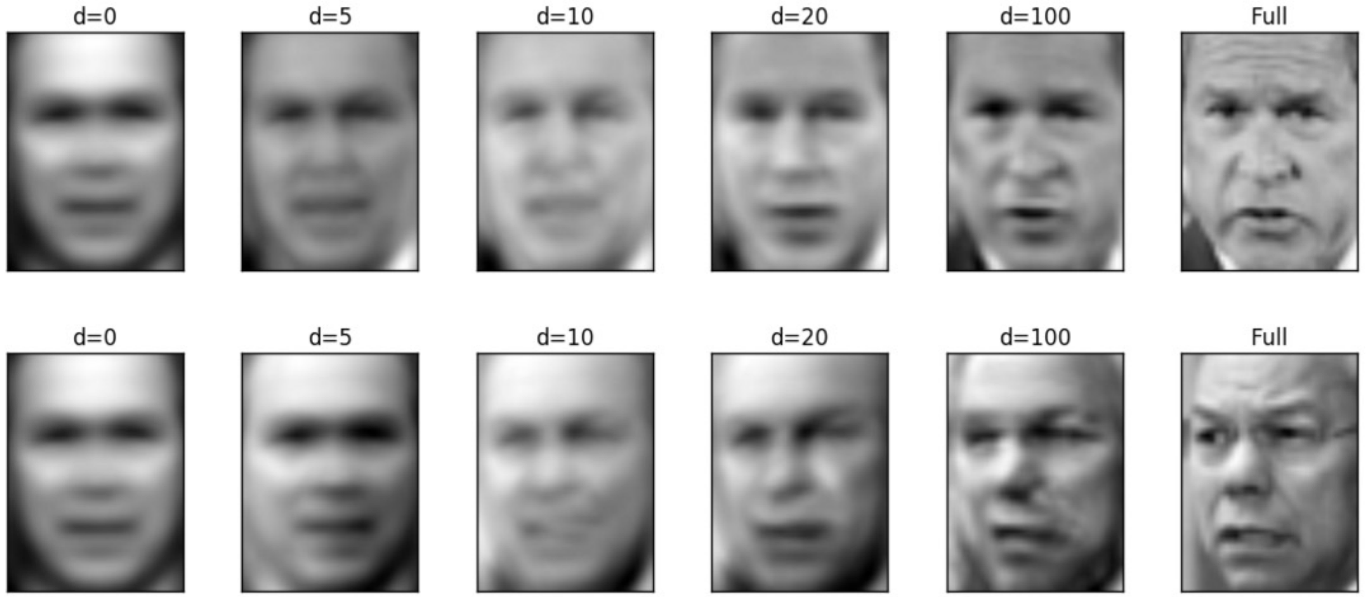
```
# Loop over figures
iplt = 0
for ind in inds:
    for d in ds:
        plt.subplot(nplt,nd+1,iplt+1)
        if use_pca:
            # Zero out coefficients after d.
            # Note, we need to copy to not overwrite the coefficients
            Zd = np.copy(Z[ind,:])
            Zd[d:] = 0
            Xhati = pca.inverse_transform(Zd)
        else:
            # Reconstruct with SVD
            Xhati = (U[ind,:d]*S[None,:d]).dot(Vtr[:d,:]) + Xmean

        plt_face(Xhati)
        plt.title('d={0:d}'.format(d))
        iplt += 1

    # Plot the true face
    plt.subplot(nplt,nd+1,iplt+1)
    plt_face(X[ind,:])
    plt.title('Full')
    iplt += 1
```

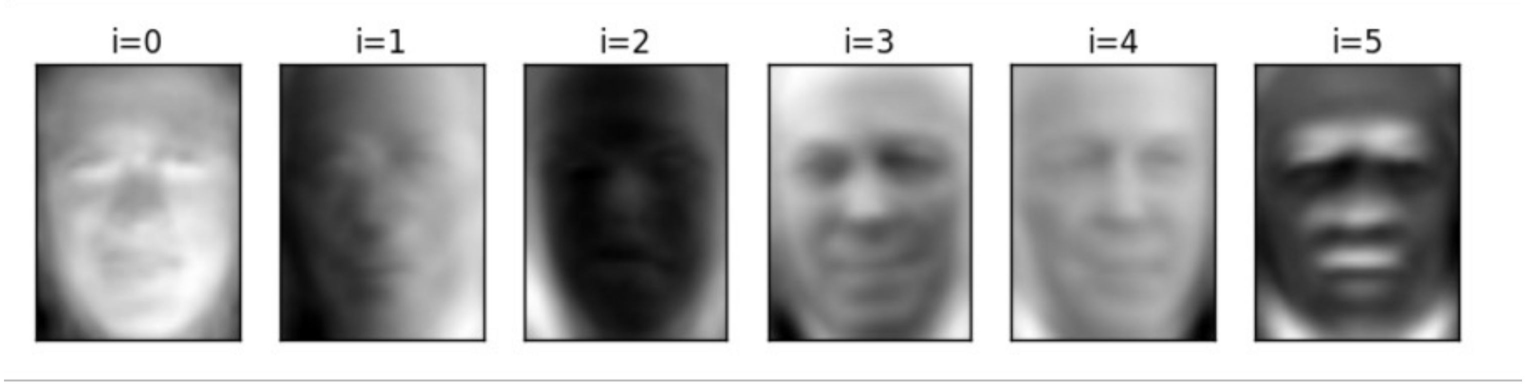
- Reconstruction using sklearn method
  - Uses the inverse_transform method to get back values
- Reconstruction using SVD
  - Note use of broadcasting

Plotting the Approximations



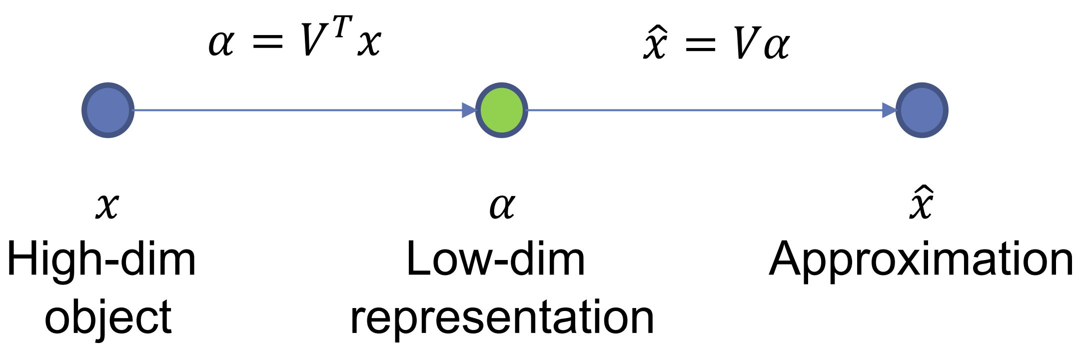
Plotting the PCs

- The PCs can be plotted as well



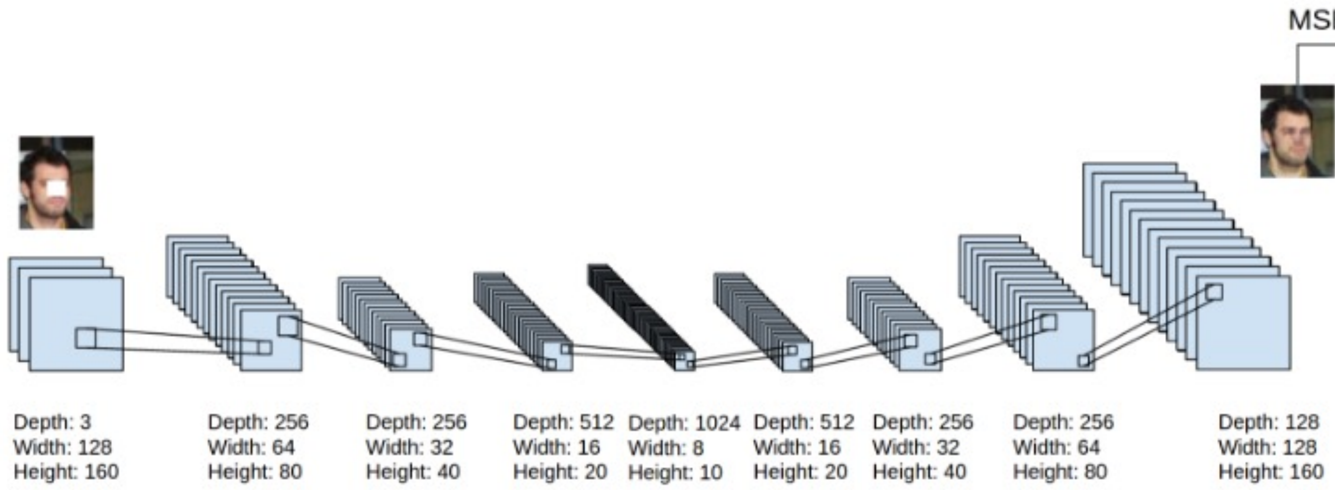
State-of-the-Art: Auto-Encoders

- PCA is a simple example of an autoencoder
- Tries to find low-dim representation
- Restricted to linear transforms
- Not very good for images and complex data



Deep Auto-Encoders

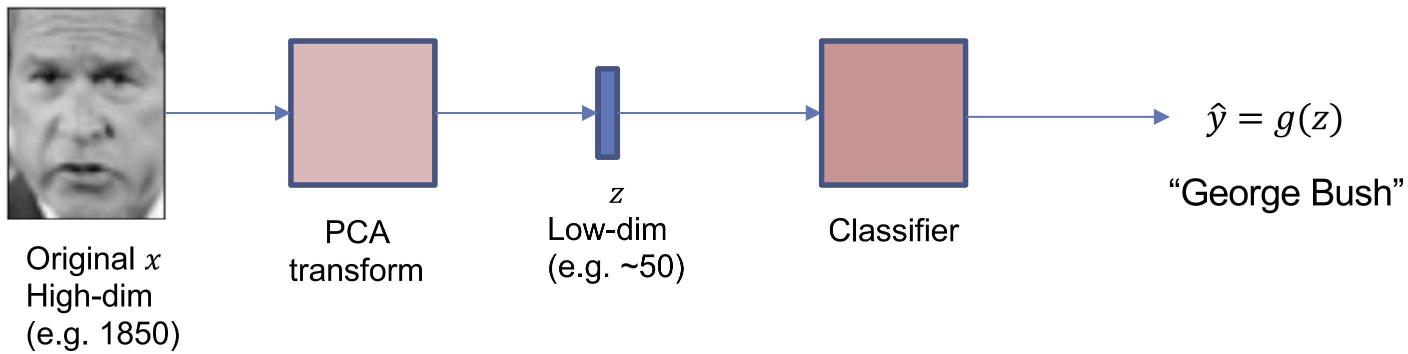
- Can use deep networks for learning complex latent representations and their inverses
  - http://www.cc.gatech.edu/~hays/7476/projects/Avery_Wenchen/
  - https://swarbrickjones.wordpress.com/2016/01/13/enhancing-images-using-deep-convolutional-generative-adversarial-networks-dcgans/ (Code in Theano not tensorflow)



Outline

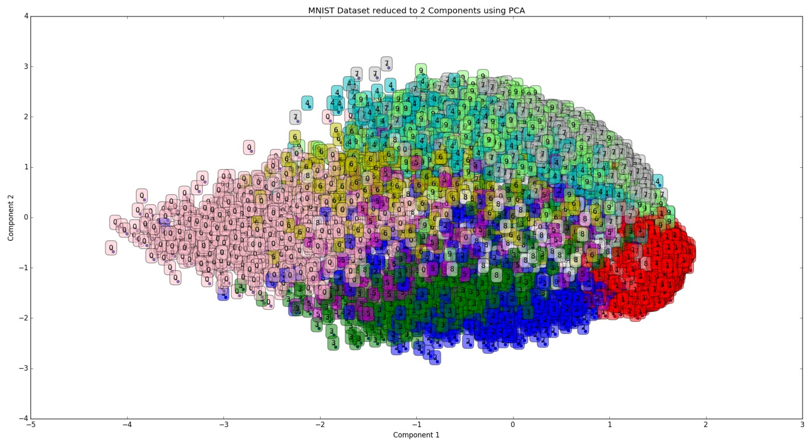
- Dimensionality reduction
- Principal components and directions of variance
- Approximation with PCs
- Computing PCs via the SVD
- Face example in python
- Training models from PCs
- Low rank approximations and recommender systems

Classification Using PCs



- Many problems: Dimensionality of data x is too large
  - Classifier in original space will have too many parameters
- Key idea:
  - Learn a dimension reducing transform via PCA: $z = f(x)$
  - Train classifier on low-dim transform $\hat{y} = g(z)$

Why This Would Work?



- PCA works if:
  classes are separable in transformed domain
- Example to left:
  - MNIST digits plotted in two PCs
  - Can mostly separate the classses

Training and Testing

- Split data in training and test: $X_{tr}, y_{tr}, X_{ts}, y_{ts}$
- Fit PCA transform on $Z = g(X)$ on training data X_{tr}
  - Do not include test data in PCA fit!
  - Many students make this mistake
- Transform training and test:
  - $Z_{tr} = g(X_{tr}),\ Z_{ts} = g(X_{ts})$
- Fit classifier $\hat{y} = f(z)$ on transformed training data (Z_{tr}, y_{tr})
- Predict classifier on transformed test data: $\hat{y}_{ts} = f(Z_{ts})$
- Score error rate / MSE on test data: $\epsilon = \frac{1}{N}\#\{\hat{y}_{ts}^{i} \neq y_{ts}^{i}\}$

How low of a dimension should I choose?

Cross-Validation

- To find number of PCs and other parameters use cross-validation (you can also use k-fold validation)
- Split data in training and test: $X_{tr}, y_{tr}, X_{ts}, y_{ts}$
- For each set of parameters:
  - Fit PCA transform on $Z = g(X, \mathrm{numPCs})$ on training data X_{tr}
  - Transform training and test: $Z_{tr} = g(X_{tr}),\ Z_{ts} = g(X_{ts})$
  - Fit classifier $\hat{y} = f(z)$ on transformed training data (Z_{tr}, y_{tr})
  - Predict classifier on transformed test data: $\hat{y}_{ts} = f(Z_{ts})$
  - Score (e.g. error rate / MSE) on test data: $\epsilon = \frac{1}{N}\#\{\hat{y}_{ts}^{i} \neq y_{ts}^{i}\}$
- Select the parameters with lowest score
  - Number of PCs to use
  - Classifier may have some parameters too, e.g., gamma in RBF for SVM

Example: SVM classification with PCAs

```
npc_test = [25,50,75,100,200]
gam_test = [1e-3,4e-3,1e-2,1e-1]
C = 100
n0 = len(npc_test)
n1 = len(gam_test)
acc = np.zeros((n0,n1))
acc_max = 0

for i0, npc in enumerate(npc_test):

    # Fit PCA on the training data
    pca = PCA(n_components=npc, svd_solver='randomized', whiten=True)
    pca.fit(Xtr)

    # Transform the training and test
    Ztr = pca.transform(Xtr)
    Zts = pca.transform(Xts)

    for i1, gam in enumerate(gam_test):

        # Fiting on the transformed training data
        svc = SVC(C=c, kernel='rbf', gamma = gam)
        svc.fit(Ztr, ytr)

        # Predict on the test data
        yhat = svc.predict(Zts)

        # Compute the accuracy
        acc[i0,i1] = np.mean(yhat == yts)
        print('npc=%d gam=%12.4e acc=%12.4e' % (npc,gam,acc[i0,i1]))
```

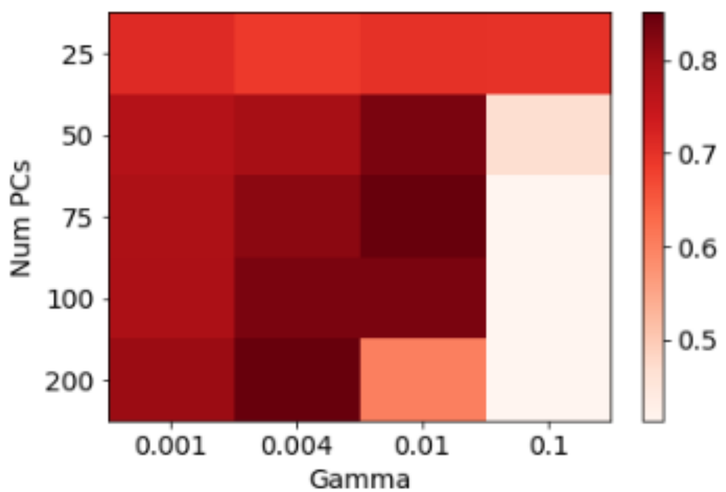
- Parameters to search
  - Number of PCs and gamma for RBF in SVM
  - C param in SVM fixed to 100
- Fit on the training data.
  - This is in the loop!
- Transform the data
- Fit classifier on transformed training data
- Test on the transformed test data
- Score on test data

Example: Parameter Search

- Search over:
  - Number of PCs $\in \{25,50,75,100,200\}$
  - $\gamma \in \{0.001, 0.004, 0.01, 0.1\}$

- Plotted is the test accuracy
- Best test accuracy $\approx 85\%$
- Original data has 1850 dimension, but 75 PCs is optimal! Large reduction!
- More PCs reduces accuracy

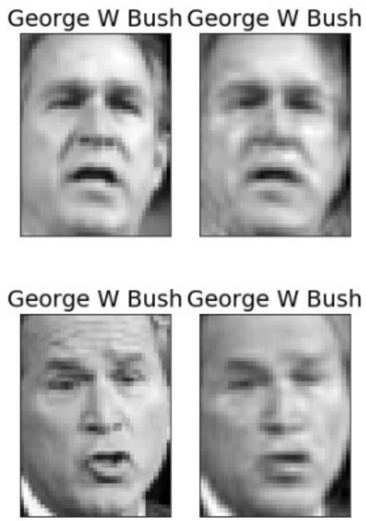
Test Accuracy




```
Optimal num PCs = 75
Optimal gamma   = 0.010000
```

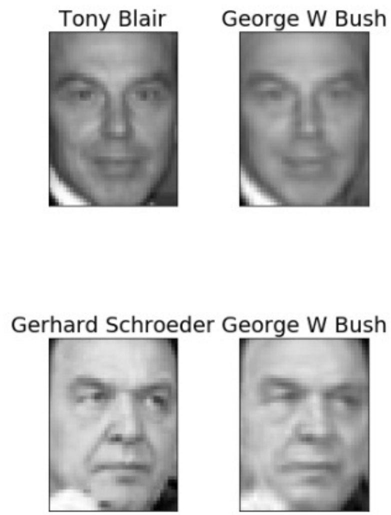
Examples

- Correct images



Original Reduced

- Error images



Original Reduced

Outline

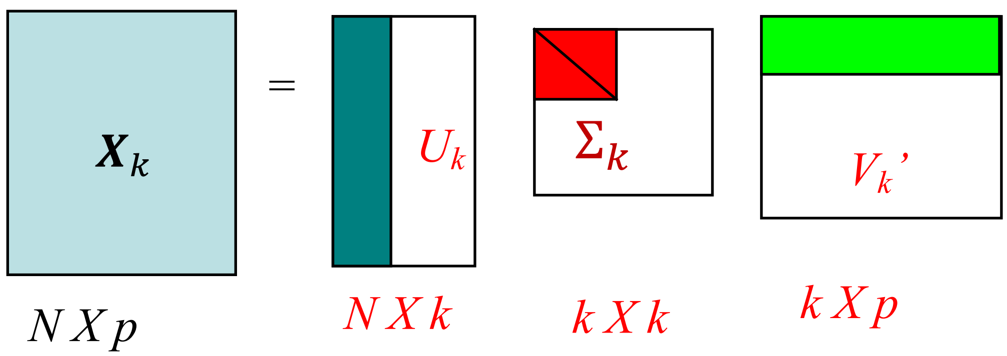
- Dimensionality reduction
- Principal components and directions of variance
- Approximation with PCs
- Computing PCs via the SVD
- Face example in python
- Training models from PCs
- Low rank approximations and recommender systems

Low-Rank Approximations

- SVD can be used for a low-rank approximation
- SVD can be written: $\boldsymbol{X} = \mathbf{U\Sigma V}^T = \sum_{j=1}^{r} \alpha_j \boldsymbol{u}_j \boldsymbol{v}_j^T$
- Consider k −term approximation: $\boldsymbol{X_k} = \sum_{j=1}^{k} \alpha_j \boldsymbol{u}_j \boldsymbol{v}_j^T$
- Properties:
  - $\boldsymbol{X}_k$ is rank k
  - $\boldsymbol{X}_k = \boldsymbol{U_k \Sigma_k V_k^T}$
  - Error is $\|X - X_k\|_F^2 = \sum_i \sum_j \left(X_{ij} - X_{k,ij}\right)^2 = \sum_{j=k+1}^{r} \alpha_j^2$
  - If $s_{k+1,} \dots, s_r$ is small, then matrix is well approximated by rank k matrix

The next slides just give some high level ideas!

Low-Rank Approximation Visualized



- Can show: Reconstructed matrix X_k is optimal rank k approximation

Recommender Systems

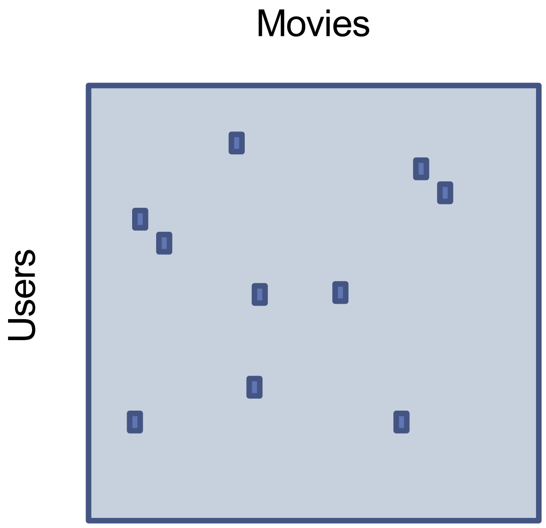
- How do you recommend a movie to a user?
- MovieLens dataset:
  - Get past ratings from users
  - Make recommendations for future

t[3]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

Ratings Matrix

- Data can be represented as ratings matrix
  - Users x movies
- Problem: Most users have only rated a small fraction
- Need to estimate unseen entries
  - Very sparse
- How can we complete this matrix



Name	Dates	Users	Movies	Ratings	Density
ML Latest	'95 – '16	247,753	34,208	22,884,377	0.003%
ML Latest Small	'96 – '16	668	10,329	105,339	0.015%

Latent Factor Model for Ratings

- Idea: Ratings for movies dependent on small number of latent factors
  - E.g. Action, famous actors, genre, …
- Mathematically model as:

$$R_{ij} \approx \hat{R}_{ij} = b_i^u + b_j^m + \sum_{k=1}^{K} A_{ik} B_{jk}$$

  - R_{ij} =Rating of movie j by user i
  - b_i^u =Bias of user i
  - b_j^m =Bias of movie j
  - K = number of latent factors. Typically small $K \ll N_{user}, N_{movies}$
  - A_{ik} = Preference of user i to factor k
  - B_{jk} = Component of factor k in movie j