Priniciple Component Analysis (and various Eigen-things)

pspieker@cs.washington.edu

Links + places that helped with this notebook:

- Fox/Guestrin's Slides (17WI used them) (https://courses.cs.washington.edu/courses/cse446/17wi/slides/pca-annotated.pdf)
- Noah Smith's Slides (17AU) (https://courses.cs.washington.edu/courses/cse446/17au/unsup2.pdf)
- Sham's Slides (from this quarter) (https://courses.cs.washington.edu/courses/cse446/18wi/slides/unsup2.pdf)
- Google Paper on PCA (https://arxiv.org/pdf/1404.1100.pdf)
- SVD Blog post (https://blog.statsbot.co/singular-value-decomposition-tutorial-52c695315254)

So we have some dataset with n examples and d dimensions. For some data, d could be large (~10,000 for example), meaning that the data matrix $n\ x\ d$ becomes quite cumbersome to deal with.

Having smaller dimensionality has several more benefits, including:

- making visualization easier (hard to visualize 3D/4D/RD)
- discovering 'intrinsic' dimensionality of the data
  - data could actually have many features that are irrelevant

Our goal then could be to reduce the "dimensionality" of the data by significantly reducing the size of d. Essentially we want to learn a mapping:

$$f : R^{nxd} \rightarrow R^{nxk}$$

with $k << d$.

We want this mapping to also be a "pretty good" representation of the data, so we need to introduce a notion of "goodness" that lets us measure how good a given mapping is.

Loosely, we'll think about our "pretty good" representation as one that is able re-create the original data without much loss of information about it.

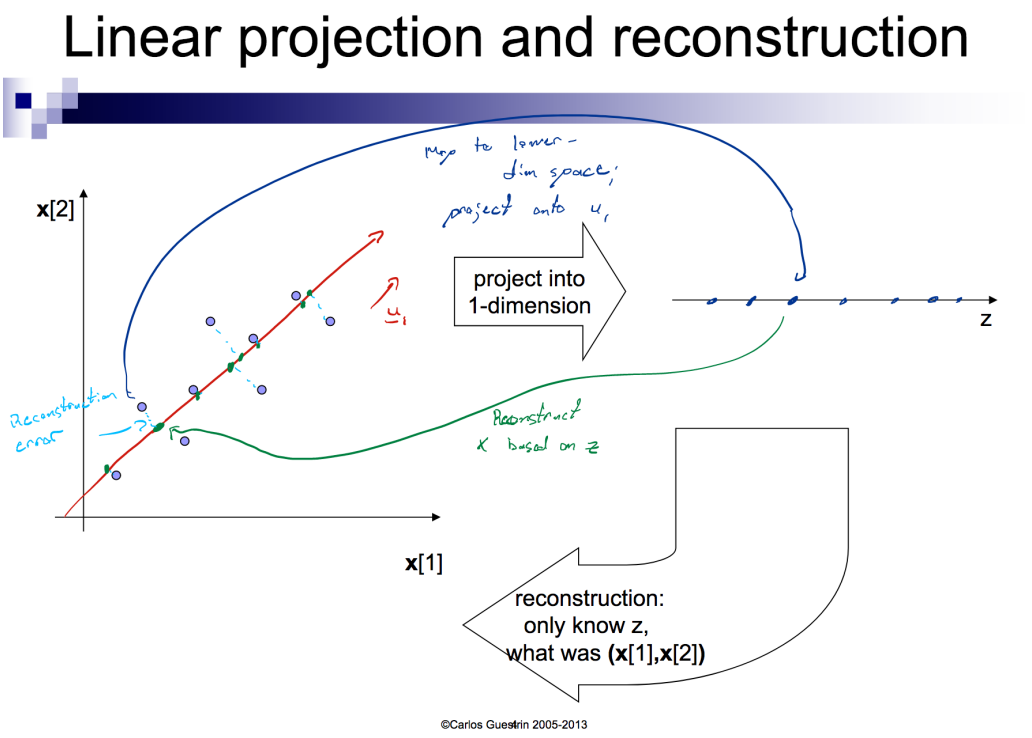
Making this concrete

We're also going to do this in the unsupervised setting (we don't have the associated y's).

For simplicity, let's assume I just have a 2 dimensional vector $x = (x_1, x_2)$ that I want to reduce to dimensionality 1.

In [4]:
```
from IPython.core.display import Image
Image(filename='reconstruction.png', width = 600)
```

Out[4]:



Okay, so we have this idea of projection and then reconstuction:

1. we project that data in the 2 dimensions down into the 1 dimension
2. we reconstruct the data in the 2 dimensions using ONLY the vector we've discovered

Given n data points: $x_1 = (x_i[1], x_i[2], \ldots, x_i[d])$ for $i = 1, \ldots, n$.

We'll represent each point as a projection:

$$\hat{x}_i = \bar{x} + \sum_{j=1}^{k} [z_i[j] * u_j]$$

Note that we can write our original point as

$$x_i = \bar{x} + \sum_{j=1}^{d} [z_i[j] * u_j]$$

Note here that each point has its own $z_i[j]$, and that the $u_{[j]}$ are shared for all points. You'll note that we have to keep track of one of these u_j for all $j = 1, \ldots, k$. Also:

$$z_i[j] = (x_i - \bar{x}) \cdot u_j$$

PCA

Given k<d, find $(u_1, \ldots, u_k)$ that minimize the reconstuction error:

$$\text{error}_k = \sum_{i=1}^{n} (x_i - \hat{x}_i)^2$$

The x_i here is the truth, and the $\hat{x}_i$ is the reconstructed version.

In linear algebra terms: identify the most meaningful basis to re-express a data set, in the hopes of filtering out noise and *reveal hidden structure*.

Understanding the reconstruction error

We'll transform the PCA problem into one you've probably seen before. We'll do this by re-writing the error:

$$\text{error}_k = \sum_{i=1}^{n}(x_i - [\bar{x} + \sum_{j=1}^{k} z_i[j] * u_j])^2$$

$$= \sum_{i=1}^{n}\left[\left[\bar{x} + \sum_{j=1}^{d} z_i[j] * u_j\right] - \left[\bar{x} + \sum_{j=1}^{k} z_i[j] * u_j\right]\right]^2$$

In [6]:
```
from IPython.core.display import Image
Image(filename='reconstruction-math.png', width = 800)
```

Out[6]:

$$= \sum_{i=1}^{N}\left[\sum_{j=K+1}^{D} \mathbf{z}_i[j]\mathbf{u}_j\right]^2 = \sum_{i=1}^{N}\left[\sum_{j=K+1}^{D} \mathbf{z}_i[j]\mathbf{u}_j \cdot \mathbf{u}_j\mathbf{z}_i[j] + 2\sum_{j=K+1}^{D}\sum_{\ell>j}^{D} \mathbf{z}_i[j]\mathbf{u}_j \cdot \mathbf{u}_\ell\mathbf{z}_i[\ell]\right]$$

$$= \sum_{i=1}^{N}\sum_{j=K+1}^{D}(\mathbf{z}_i[j])^2$$

Minimize the projection into the dimensions that are ignored

(orthonormal)

In [5]:
```
from IPython.core.display import Image
Image(filename='covar.png', width = 800)
```

Out[5]:

Plugging in the definition for $z_i[j]$

$$\boldsymbol{\Sigma} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T$$

$$\sigma_{m\ell} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i[m] - \bar{\mathbf{x}}[m])(\mathbf{x}_i[\ell] - \bar{\mathbf{x}}[\ell])$$

$$\text{error}_K = \sum_{i=1}^{N}\sum_{j=K+1}^{D}[\mathbf{u}_j \cdot (\mathbf{x}_i - \bar{\mathbf{x}})]^2$$

For vectors a, b: $(a \cdot b)^2 = (a^T b)^2$

$= (a^T b)^T (a^T b)$

$= b^T a a^T b$

$$= \sum_{i=1}^{N}\sum_{j=K+1}^{D}\mathbf{u}_j^T(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T\mathbf{u}_j$$

$$= \sum_{j=K+1}^{D}\mathbf{u}_j^T\left[\sum_{i=1}^{n}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T\right]\mathbf{u}_j \quad (= N\boldsymbol{\Sigma})$$

$$= N\sum_{j=K+1}^{D}\mathbf{u}_j^T\boldsymbol{\Sigma}\mathbf{u}_j$$

So we've completely rewritten our error such that we are: **picking an ordered, orthonormal basis $(u_1, \ldots, u_d)$ to minimize:**

$$error_k = N \sum_{j=k+1}^{D} u_j^T \Sigma u_j$$

Eigen-things

So before we go full Math 308 on you, brief eigen-things review:

Def: **eigenvector**: A vector u is an eigenvector of some matrix A iff there exists some scalar λ such that $Au = \lambda u$.

Def: **eigenvalue**: the λ that corresponds to u, so every eigenvector is associated w/an eigenvalue and vice versa.

Def: **eigenspace**: the set of all eigenvectors corresponding to a given λ

Quickly doing some algebra:

$$Au = \lambda u$$
$$Au - \lambda u = 0$$
$$(A - \lambda I)u = 0$$

So essentially we are looking for non-zero solutions to the above problem. Using the Big Theorem, the above equation has a non-zero solution iff $det(A - \lambda I) = 0$.

Example: Find the eigenvalues for:

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & 4 & 3 \end{bmatrix}$$

So we solve and recover:

$$det(A - \lambda I) = (1 - \lambda)(\lambda - 5)(\lambda + 1)$$

We have 3 eigenvalues here: $\lambda_1 = 5$, $\lambda_2 = 1$, $\lambda_3 = -1$.

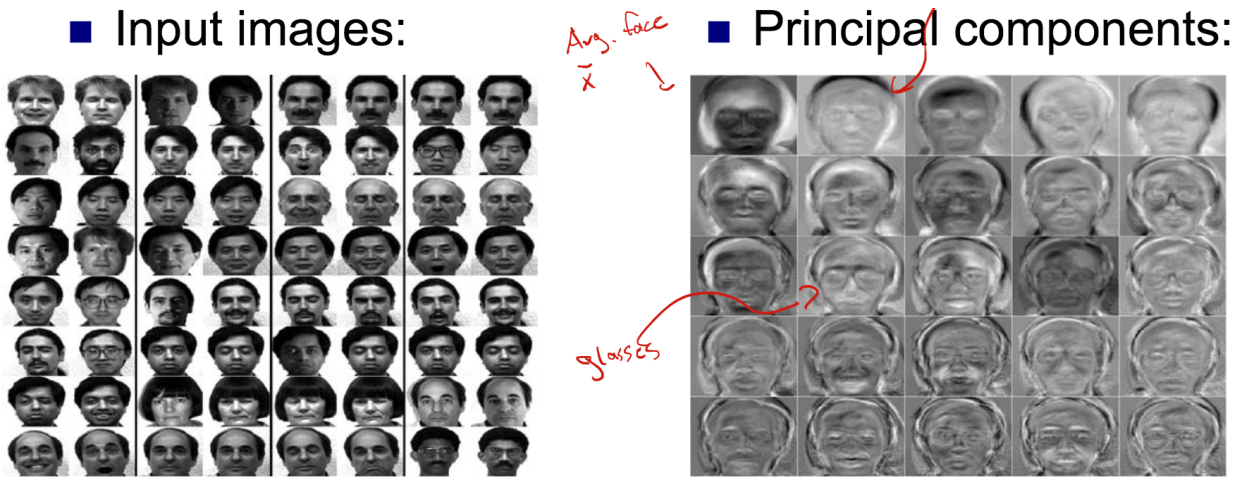
Another way to think about this is minimizing the **sum of the $d - k$ eigenvalues of Σ**, or **keeping the top k eigenvalues of Σ**.

The PCA Algorithm

- Start with n by d data matrix X
- Recenter: subtract mean from each row of X
- Compute the covariance matrix: $\Sigma := \frac{1}{N} X_c^T X_c$
- Find the eigenvalues and eigenvectors of Σ (usually via `numpy.linalg.eig`)
- The principal components: the k eigenvectors with the highest values

In [4]:
```
from IPython.core.display import Image
Image(filename='eigen-ex.png', width = 1000)
```

Out[4]:



Brief Aside - Singular Value Decomposition

Motivation: really hard to find all eigenvectors normally, SVD much better at finding to k eigenvectors (`scipy.linalg.svd`)

We write:

$$X = WSV^T$$

where:

- X is the n by d data matrix, with 1 row per data point
- W is a n by d weight matrix, with 1 row per data point (the coordinate of x_i in the eigenspace)
- S is the d by d singular value matrix, which is diagnol with each entry along the diagnol is eigenvalue λ_j
- V^T is the d by d singular vector matrix , which has each row as an eigenvector v_j

What is the big advantage?

A: **practical concerns**. Modeling the data matrix as the SVD makes finding the eigenvectors easier (see this blog post (https://intoli.com/blog/pca-and-svd/) for more on the SVD vs. PCA stuff). Performing the eigen-decomposition directly on $X^T X$ is difficult.