# Applications of the Singular Value Decomposition

John Burkardt (Substituting for Professor Dahmen)
Math 728-D
Machine Learning and Data Science
Department of Mathematics
University of South Carolina

...

LeConte 121
1:15-2:30, 12+14 February 2019

# SVD in Machine Learning

Machine learning extracts information from massive sets of data.

The singular value decomposition (SVD) starts with “data” which is a matrix $A$, and produces “information” which is a factorization $A = U * S * V'$ that explains how the matrix transforms vectors to a new space;

In many machine learning problems, the massive sets of data can be regarded as a collection of $m$-vectors, which can be arranged into an $m \times n$ matrix.

SVD application examples:

1. least squares line, *data: points in 2D*;
2. curve-fitting, *data: polynomial coefficients*;
3. matrix approximation, *data: entries in a matrix*;
4. image compression, *data: columns of an image*;
5. facial recognition, *data: pictures of a face* .

# Vectors

Any numeric list of M real-valued entries will be a vector $v$.

In mathematics, we primarily think of vectors as *column vectors*, that is, a vertical stack of numbers. There is a complementary concept of row vectors, which are not important for us right now.

Interesting MATLAB commands:

```
v = [ 1; 2; 3 ];           <-- 3x1 column vector
w = [ 4, 5, 6, 7 ];        <-- 1x4 row vector
A = [ 11, 12, 13;
      21, 22, 23 ].        <-- 2x3 matrix
u = v';                    <-- transpose column to row;
```

# Vector Length

By vector "length" we won't mean the number of entries in the vector, but rather a geometric length, known as the (Euclidean) length, or l2-norm, and represented by $||v||$ or $||v||_2$:

$$||v|| = \sqrt{\sum_{i=1}^{M} v_i^2}$$

A unit vector has length 1; unit vectors are often denoted by $u$.

In MATLAB, we can write:

```
    vnorm = norm ( v );
```

or, to use the *l1* norm instead of the default *l2* norm:

```
    vnorm = norm ( v, 1 );
```

# Angle Between Vectors

Any two vectors $v_1$ and $v_2$ lie in a common plane, defining an angle $\alpha$.

The dot product reveals the cosine of this angle:

$$v_1 \cdot v_2 = \sum_{i=1}^{n} v_1(i)\, v_2(i) = ||v_1||\, ||v_2||\, cos(\alpha)$$

We can solve for the angle:

$$\alpha = \arccos\left(\frac{v_1 \cdot v_2}{||v_1||\, ||v_2||}\right)$$

If a unit vector $u$ is involved, then $v \cdot u$ is the *projection* of $v$ onto $u$.

If $u_1$ and $u_2$ are unit vectors, then the formula is simply:

$$\alpha = \arccos(u_1 \cdot u_2)$$

In MATLAB, transpose first vector before multiplying:

```
v1dotv2 = v1' * v2;
```

# Angle Measures Similarity

Looking just at $|\cos(\alpha)|$ for vectors $v_1$ and $v_2$, we can say:

$$|\cos(\alpha)| = \begin{cases} 1 & \text{vectors have same direction} \\ 0 & \text{vectors are perpendicular, orthogonal} \end{cases}$$

Moreover, values near 1 indicate that the vectors are very similar in direction, while values near 0 indicate the vectors have little in common.

Because we are working in a linear algebra setting, it's the direction of the vectors that's important, not the length.

If we have $N$ vectors, and we want to select a subset that contains the most information, then we are looking for a set of dissimilar vectors. The vector dot product gives us that information in $|\cos(\alpha)|$.

# Norms, Dot Products, and Angles in MATLAB

Given vectors $v_1$, $v_2$, $v_3$, $v_4$:

```
v1 = [   3; 4     ];   norm(v1) = 5
v2 = [ - 4; 3     ];   norm(v2) = 5
v3 = [   4; 4     ];   norm(v3) = 5.6569
v4 = [ 0.5; 0.866 ];   norm(v4) = 1
```

we can measure pairwise cosines:

```
ca11 = (v1' * v1) / norm (v1) / norm (v1) =  1
ca12 = (v1' * v2) / norm (v1) / norm (v2) =  0
ca13 = (v1' * v3) / norm (v1) / norm (v3) =  0.9899
ca14 = (v1' * v4) / norm (v1) / norm (v4) =  0.9928
ca23 = (v2' * v3) / norm (v2) / norm (v3) = -0.1414
ca24 = (v2' * v4) / norm (v2) / norm (v4) =  0.1196
ca34 = (v3' * v4) / norm (v3) / norm (v4) =  0.9659
```

# Matrix Norms

Suppose a vector is the result of a matrix-vector multiplication:

$$w = A * v$$

The l2 norm for a matrix $A$ is defined as the square root of the maximum eigenvalue of this matrix $A' * A$. We use the matrix norm to write:

$$||w|| \leq ||A|| \, ||v||$$

The norms of $A$ and $v$ limit how big $w$ can be.

If $A$ is an $M \times M$ diagonal matrix, then $||A|| = \max_{i=1}^{M} |A_{i,i}|$

If $A$ is an $M \times M$ matrix whose columns have unit l2 norm, then $A$ is an orthogonal matrix, and $||A|| = 1$ (and inverse(A)=A').

If $A$ is an orthogonal matrix, $v$ any $M$ vector, $B$ any $M \times N$ matrix:

$$||A * x|| = ||x||$$
$$||A * B|| = ||B||$$

# Matrix Norms

Commonly-used matrix norms:

- l1: maximum sum of entries in a column of $A$;
- l2: maximum eigenvalue of $A'A$ or maximum singular value of $A$;
- l$\infty$: maximum sum of entries in a row of $A$;
- Frobenius: square root of sum of squares of all entries of $A$;

MATLAB evaluation:

- l1: norm(A,1)
- l2: norm(A) or norm(A,2)
- l$\infty$: norm(A,Inf)
- Frobenius: norm(A,'fro')

# Column Space

Any matrix $A$ can be looked at as a collection of column vectors.

The *column space* of $A$ is the linear space formed by all possible linear combinations of the columns. If $A$ is $m \times n$, then the column space is the set of all $m$-vectors $y = A * x$ where $x$ is any $n$-vector.

Given a matrix $A$, we can always use Gram-Schmidt or other orthogonalization schemes to construct an $m \times k$ matrix $U$ whose columns are of unit l2 norm and which are pairwise orthogonal. This matrix has the property that

$$U' * U = I_k, \text{ the } k \times k \text{ identity matrix.}$$

Unless $m = k$, $U$ is not, strictly speaking, an orthogonal matrix, because it is not square.

# The Projection Matrix

Let's assume that $k < m$, so that $U$ does not have full rank.

We know that $U' * U = I_k$, but what about the product $U * U'$?

This cannot be $I_m$, since $U$ has rank $k < m$.

But in fact, $U * U'$ is a very useful matrix; it's the projection matrix that maps $m$-vectors into the column space of $U$.

A projection matrix is a linear operator $P$ such that $P * P = P$. It is easy to see that $U * U'$ must be a projection matrix:

$$(U * U') * (U * U) = U * (U' * U) * U' = U * I_k * U' = U * U'$$

Let's look in detail at how a projection matrix is used.

# Projection

If $x$ is any vector, we can map it, or project it, into the column space of the orthonormal basis matrix $U$ by a simple operation:

$$c = U' * x$$

Here, $c$ describes a combination of the columns of $U$ which is the nearest vector to $x$, the least-squares approximation to $x$.

Notice: left-multiplying $x$ by the transpose of the basis, $U'$ reveals a recipe $c$ for creating the vector $x$ as a multiple of the columns of $U$.

```
x = [1]  U = [-0.169  0.897]  c=U’*x=[-7.944]
    [2]      [-0.507  0.276]         [-1.311]
    [8]      [-0.842 -0.345]
```

Notice: the column space of U is 2-dimensional, because there are just 2 columns. So $c$ is a vector of length 2, because it is in that column space. That means it's hard for us to see how $x$ and $c$ are related.

# Representation

The vector $c$ is a representation of the vector $x$ in the column space of U.

To see what $c$ really means, we just have to do what is says, that is, add the appropriate amounts of each column of $U$. And we do that with multiplication:

$$x2 = U * c$$

Notice: left-multiplying the coefficients $c$ by the orthonormal basis $U$ recovers the original form of the vector $x$.

```
x = [1]  U = [-0.169  0.897]  c=U'*x=[-7.944]  x2=U*c= [0.167]
    [2]      [-0.507  0.276]         [-1.311]          [3.667]
    [8]      [-0.842 -0.345]                           [7.166]

|| x - x2 || = 2.041 <-- lowest possible approximation error
                       = distance from x to column space of U
```

So, given a vector $x$, we get the coefficients by $c = U' * x$ and we see the resulting approximate vector by $x2 = U * c = U * U' * x$.

# Building a Basis

From $m$-vectors $v_1, v_2, ...v_N$ we seek basis vectors $u_1, u_2, ...u_K$.

The $u$ vectors should have unit norm, and be pairwise orthogonal:

$$u_i \cdot u_j = \begin{cases} 1 & i = j \\ 0 & otherwise \end{cases}$$

We could pack them as columns into an $M \times K$ basis matrix $U$:

$$U = [u_1|u_2|...|u_K]$$

This makes it easy to project any vector $v$ onto the set of $u$ vectors:

$$c = U' * v$$

The $c$ coefficients approximate $v$ using only the $u$ vectors:

$$v \approx \hat{v} = \sum_{i=1}^{K} u_i * c(i) = U * c$$

With enough $u$ vectors, the approximation is exact. But if we economize, using a small value $K << N$, we seek $U$ to minimize approximation error.

# Singular Value Decomposition

$A = U * D * V'$ where

- $U$ is orthogonal, and $m \times m$;
- $D$ is diagonal, and $m \times n$, with nonnegative diagonal entries $\sigma_i$;
- $V$ is orthogonal, and $n \times n$;

The $min(m, n)$ diagonal elements of $D$, written $\sigma_i$, are nonnegative, and in decreasing order. The value $\sigma_1$ is the l2 norm of A.

In MATLAB, get the factors by writing:

```
[ U, D, V ] = svd ( A );
```

# Singular Value Decomposition

- Define $n \times m$ diagonal matrix $D^+$ with entries $\begin{cases} 1/\sigma_i & \text{if } \sigma_i \neq 0 \\ 0 & \text{otherwise;} \end{cases}$
- Inverse $A^{-1}$ or pseudoinverse $A^+$ is $V * D^+ * U'$;
- Solve A*x=b by $x = V * D^+ * U' * b$;
- Same procedure for singular/nonsingular, square/rectangular $A$;
- Singular and underdetermined systems: minimal L2 norm of solution;
- Overdetermined systems: minimal L2 norm of residual;
- $|\det(A)| = \prod_{j=1}^{\min(m,n)} \sigma_j$;
- Condition(A) $= \frac{\max(|\sigma_j|)}{\min(|\sigma_j|)}$;
- Rank(A) = k = number of nonzero $\sigma_i$;
- Economy SVD: A=U(:,1:k)*D(1:k,1:k)*V'(:,1:k);
- Range(A) = U(:,1:k), Null space(A) = V'(:,k+1:n);
- Factors can produce a sequence of low rank approximants to $A$:

# SVD Case 1, Square Nonsingular Matrix:

```
A = [ 5, -1, 1; -4, -1, 2; 1, 1, 3];

[ U, D, V ] = svd ( A );
A2 = U * D * V'
UtU = U' * U

x = [1;2;10];
b = A * x
Dp = D'; Dp(1,1) = 1/Dp(1,1); Dp(2,2) = 1/Dp(2,2);
Ap = V * Dp * U'
inv ( A )
x2 = A \ b
x3 = inv ( A ) * b
x4 = V * Dp * U' * b

A1 =      U(:,1) * D(1,1) * V(:,1)'
A2 = A1 + U(:,2) * D(2,2) * V(:,2)'
A3 = A2 + U(:,3) * D(3,3) * V(:,3)'
```

# SVD Case 2, Square Singular Matrix:

```
A = [ 1, 2, 3; 4, 5, 6; 7, 8, 9];

[ U, D, V ] = svd ( A ); <-- Zero singular value on diagonal
A2 = U * D * V'
UtU = U' * U

x = [1;2;10];
b = A * x
Dp = D'; Dp(1,1) = 1/Dp(1,1); Dp(2,2) = 1/Dp(2,2);
Ap = V * Dp * U'
pinv ( A )                <-- Use pseudoinverse!
x2 = A \ b
x3 = pinv ( A ) * b
x4 = V * Dp * U' * b

A1 =      U(:,1) * D(1,1) * V(:,1)'
A2 = A1 + U(:,2) * D(2,2) * V(:,2)'
A2 = U(:,1:2) * D(1:2,1:2) * V(:,1:2)'
```

# SVD Case 3, Rectangular (wide) Underdetermined System:

```
A = [ 1, 1, 1; 1, 2, 3];

[ U, D, V ] = svd ( A );
A2 = U * D * V'
UtU = U' * U

x = [1;2;10];
b = A * x
Dp = D'; Dp(1,1) = 1/Dp(1,1); Dp(2,2) = 1/Dp(2,2);
Ap = V * Dp * U'
pinv ( A )                <-- Use pseudoinverse!
x2 = A \ b                <-- bigger solution norm than x!
x3 = pinv ( A ) * b       <-- minimum solution norm
x4 = V * Dp * U' * b.     <-- same as x3

A1 =      U(:,1) * D(1,1) * V(:,1)'
A2 = A1 + U(:,2) * D(2,2) * V(:,2)'
```

# SVD Case 4, Rectangular (tall) Overdetermined System:

```
A = [ 1, 1; 1, 2; 1, 3];

[ U, D, V ] = svd ( A );
A2 = U * D * V’
UtU = U’ * U

x = [1;2];
b = [3;5,10];             <-- b is NOT in range of A!
Dp = D’; Dp(1,1) = 1/Dp(1,1); Dp(2,2) = 1/Dp(2,2);
Ap = V * Dp * U’
pinv ( A )                <-- Use pseudoinverse!
x2 = A \ b                <-- Look at nonzero residual norm
x3 = pinv ( A ) * b
x4 = V * Dp * U’ * b.

A1 =      D(1,1) * U(:,1) * V(:,1)’
A2 = A1 + D(2,2) * U(:,2) * V(:,2)’
```

# SVD Case 5, Rank 1 Matrix:

```
u = [1;2;3]
v = [1;1;1]
A = u * v' = [ 1 1 1]
             [ 2 2 2]
             [ 3 3 3]

[ U, D, V ] = svd ( A );

[-0.267  0.948  0.172]   [6.481 0 0]   [-0.577 -0.817   0.000]'
[-0.535  0.003 -0.845] * [0     0 0] * [-0.577  0.408   0.707]
[-0.802 -0.318  0.506]   [0     0 0]   [-0.577  0.4082 -0.707]
```

Similarly, a rank 2 matrix will have 2 nonzero singular values, and so on.

# Example #1: The Least Squares Line

A common problem involves approximating data with a formula; often we plot a vector $x$ against $y$, and believe there is roughly a linear relationship.

Although, in this example, $x$ and $y$ will be 1-dimensional vectors, keep in mind that this same thing happens in higher dimensions; we seek a linear relationship between two sets of possibly multidimensional data.

Finding a linear relationship allows us to

- hypothesize a “law” explaining the data;
- predict new values $y$ from $x$ values;
- estimate the error in our data;
- replace all our data by simple formula;

# Linear versus Affine

The classical *slope/intercept* formula

$$y = a * x + b \quad \text{(affine relationship!)}$$

is not, strictly speaking, a linear relationship!

Strictly speaking, a linear relationship must produce a zero when given zero input, and so must have the form:

$$y = a * x \quad \text{(linear relationship)}$$

However, it is easy to go back and forth from an affine relationship to a linear relationship using a simple transformation of variables.

This fact will matter in a moment.

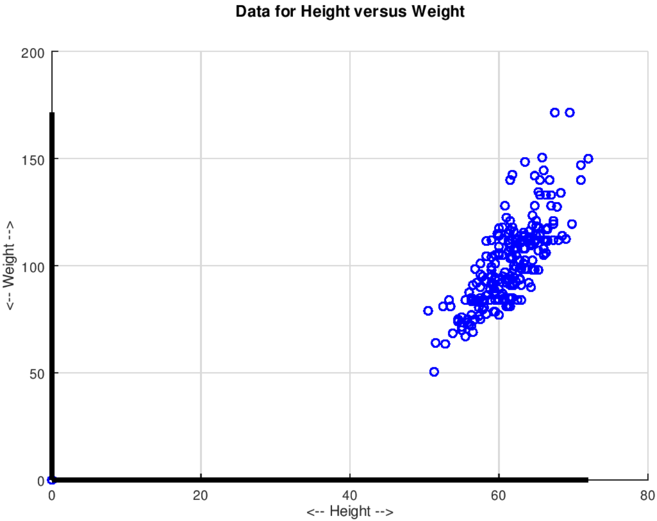
# Height and Weight Data

Suppose we measure the height and weight of 237 students. We can expect almost any combination of values within a certain range, but there ought to be an underlying pattern: *tall people are heavier*.

Is it possible that, at least over this age range, the relationship between height and weight is linear? We can't explain all the results this way, but if we make a plot, our visual sense can often convince us, if there is a pattern.

```
data = load ( 'sex_age_height_weight.txt' );
h = data(:,3);
w = data(:,4);
plot ( [ 0; h], [0;w], 'bo', 'linewidth', 1 );
```

# Plot Height and Weight Data



# Finding a Linear Relationship

We want use the SVD to find a linear relationship.

The plot shows that a line through the data won't go near the origin, so we have some work to do. We start by "centering" the data, that is, by subtracting the average.

```
h2 = h - mean ( h );
w2 = w - mean ( w );
```

Now we create a 2 × 237 matrix $A$. Row 1 is the (transposed) centered height vector, row 2 the (transposed) centered weight vector.

```
A = [ h2'; w2' ];
```

and now we ask for the SVD:

```
[ U, S, V ] = svd ( A );
```

# Identify the direction

For our data, the $2 \times 237$ matrix $S$ has diagonal entries 302.3 and 37.8; in other words, 90 % of the information is in the first dimension.

The columns of the $2 \times 2$ vector $U$ give the dominant and secondary directions:

```
U = [ -0.1578    -0.9875
      -0.9875     0.1578 ]
```

In other words, the line:

```
- 0.1578 y = - 0.9875 x
```

or

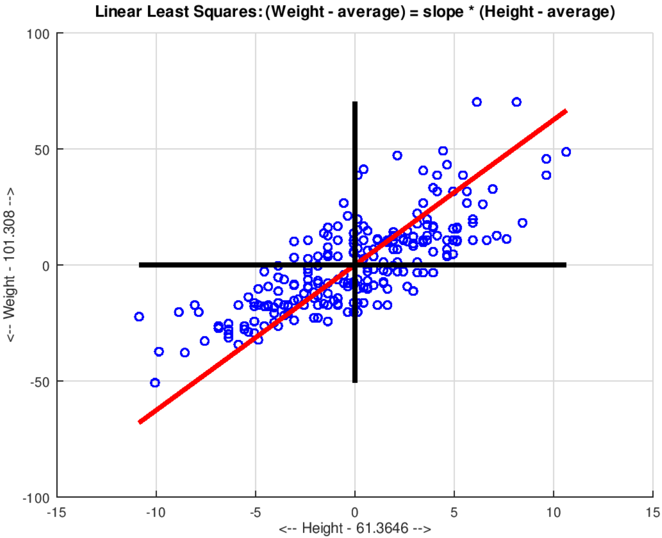
```
y = 6.256 * x
```

or, using our variable names

```
w2 = 6.256 * h2
```

Let's plot the approximate linear relationship that the SVD has discovered in our centered data.

# Linear Relationship



# Recovering the Affine Relationship

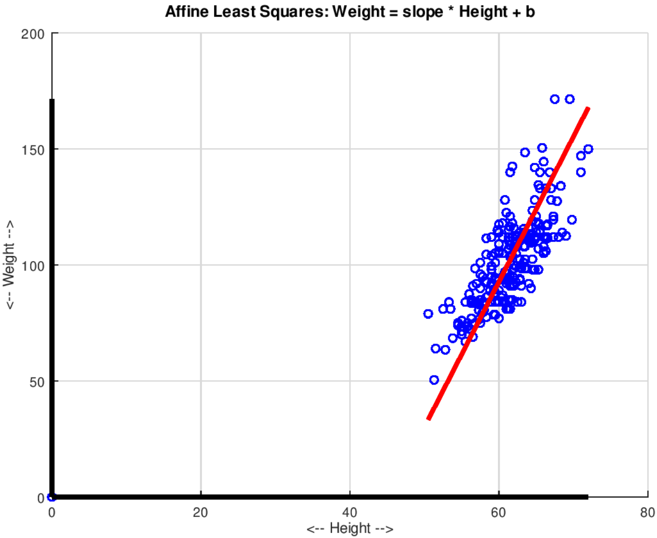
The SVD has discovered the linear relationship for us:

$$w2 = 6.256 * h2$$

but of course we want to work with the actual heights and weights, which means we need to work out the affine relationship:

$$\begin{aligned} w2 =& 6.256 * h2 \\ w - \text{mean}(w) =& 6.256 * (h - \text{mean}(h)) \\ w - 101.308 =& 6.256 * (h - 61.365) \\ w =& 6.256 * h - 282.644 \end{aligned}$$

# Affine Relationship



# What if We Skipped the Centering?

If we didn't use centered variables $h2$ and $w2$, but filled up the matrix A with the raw $h$ and $w$ values, the algorithm will still "work"... that is, it will produce a result.

Unfortunately, the result must be a linear function $y = a * x$, so it will be a line that goes through (0,0) and roughly the center of the cloud of data. This is not at all a good approximation to the (affine) behavior we can see.

But that's because the SVD approach expects to analyze linear behavior, not affine behavior. Centering the data beforehand is one way to fix this issue. Another option, which you will see when you look at various regression problems, is to add an extra variable that has the fixed value of 1.

# Example #2: Curve Fitting

Suppose we have $n$-vectors of $x$ and $y$ data but we find that a straight line does not seem to capture the behavior of the data.

Instead of a linear model, we might be tempted to look at a quadratic model, involving unknown coefficients $c$:

$$y = a + b * x + c * x^2$$

Ideally, we would hope that the values of $a, b, c$ would allow us to match our data perfectly. The equations that express this hope are:

$$\begin{aligned} a + b * x_1 + c * x_1^2 =& y_1 \\ a + b * x_2 + c * x_2^2 =& y_2 \\ ... =& ... \\ a + b * x_n + c * x_n^2 =& y_n \end{aligned}$$

# An Overdetermined Linear System

Let's replace the values $a, b, c$ by a 3-vector $c$. Then we can write

$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \dots & \dots & \dots \\ 1 & x_n & x_n^2 \end{bmatrix} * \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix}$$

which has the form:

$$A2 * c = y$$

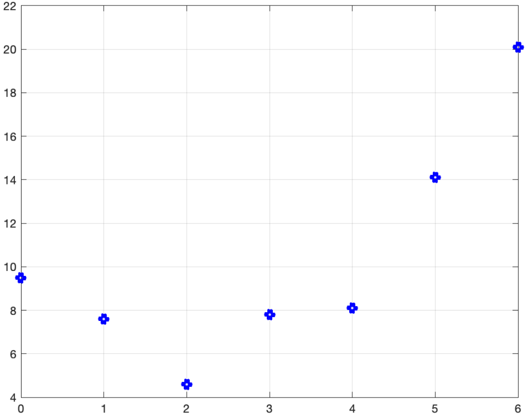
where $A2$ is an $n \times 3$ matrix, and hence overdetermined.
From the SVD we can derive the pseudoinverse, sometimes written $A^+$, which can be used to find the best approximate solution $c^+$ to this system, in the least squares sense.

$$\begin{aligned} A2 * c =& y && \text{(Equation we want to solve)} \\ A2^+ * A2 * c =& A2^+ * y && \text{(Imagine multiplying by } A2^+\text{)} \\ c^+ =& A2^+ * y && \text{(Best approximate solution)} \end{aligned}$$

We are **not** saying that $A2^+ * A2 = I$!

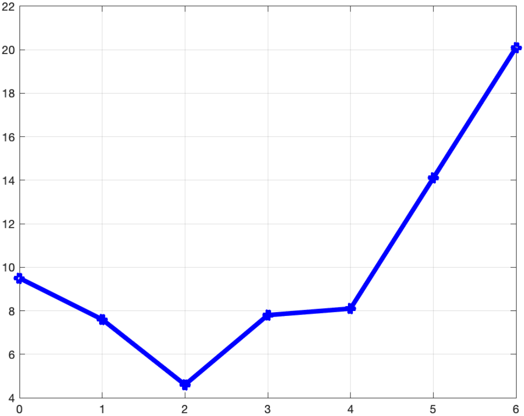
# Sort of Quadratic Data:

```
xd = [0;1;2;3;4;5;6]
yd = [9.5,7.6,4.6,7.8,8.1,14.1,20.1]
plot ( xd, yd, 'bo' ) )
```



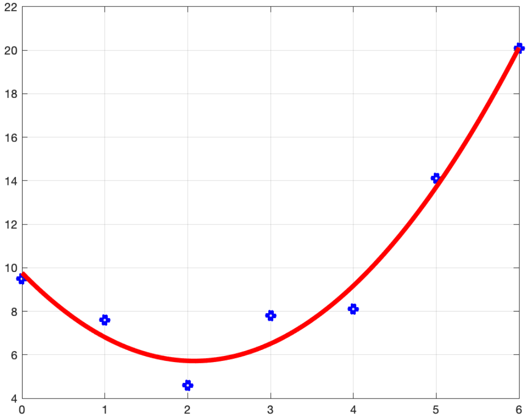
# Piecewise Linear "Fit":

```
plot ( xd, yd, 'b-' )
```



# SVD Approximation, Degree 2 Fit:

```
A2 = [ xd.^0, xd.^1, xd.^2]
c2 = pinv(A2) * yd        % c2 = [ 9.76, -3.89, 0.93]
% Define xp, plot yp = 9.76-3.89*xp+0.93*xp^2
```



# An Overdetermined Linear System

In Matlab, we determine the coefficients $c$ from the data $yd$ in this way:

```
c = pinv(A2) * yd
```

Then, if we wish create a set of plot points $(xp, yp)$, we can evaluate the quadratic formula with coefficients $c$ by:

```
yp = c(1) + c(2) * xp + c(3) * xp.^2;
```

Our matrix $A2$ went up to quadratic powers in $xd$. If we wished, we could have looked for a higher degree formula, say up to 6th degree, simply by defining the matrix

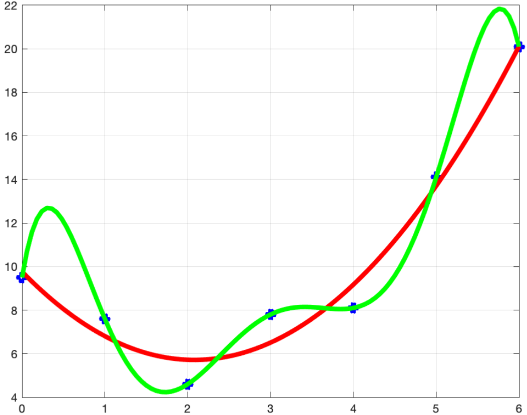
```
A6 = [ xd.^0, xd.^1, xd.^2, xd.^3, xd.^4, xd.^5, xd.^6];
```

which will give us a coefficient vector $c6$ of length 7.

What does our greater effort buy us?

# SVD Approximation, Degree 2 (red) and 6 (green) Fits:

```
A6 = [ xd.^0, xd.^1, xd.^2, xd.^3, xd.^4, xd.^5, xd.^6]
c6 = pinv(A6) * yd % c6 = polynomial coefficients
% Set plot points xp, evaluate formula for yp
```



# Example #2 Comments: Overfitting

If our goal was to match the data, then the 6th degree polynomial is great. But if our goal is to make a plausible formula that generates the data, then the wild wiggles in the curve seem wrong, and the 2nd degree polynomial might be more believable.

Wne we try too hard to fit our data exactly, we may be forcing the mathematical model to fit meaningless errors as well as our data.

The notion of "trying too hard" to model data is known as overfitting.

This is a common problem in machine learning; a model might work perfectly on your data because it has simply "memorized" all the answers.

When a model too tightly matches a set of data, it may not be able to be generalized to other sets of related data.

# Example #2 Comments: Our quadratic model

The SVD can be used to solve overdetermined linear systems, such as this curve fitting example.

In this case, we can think of the SVD as starting with the data, and extracting the simplifying information, that is, the approximate quadratic formula

$$y = 9.76 - 3.89x + 0.93x^2$$

Discovering this formula, we might feel free to discard all our data, or make some guesses about the process that generated the data, or test the formula with some extra data that we had held in reserve.

If we believe the formula is a good model for the process that generated the data, we can plot the formula, differentiate it or integrate, and use it in other ways.

# Example #3: Matrix Approximation

A idealized data set:

- a single item of data consists of $M$ values;
- we have $N$ data observations;
- $M$ and $N$ are large (thousands, or millions);
- the data is stored in an $M \times N$ array $A$;
- The data has patterns, and could be replaced by a simpler model;

Suppose we form the SVD of $A$. Then

- $A1 = D(1,1) * U(:,1) * V(:,1)'$ models the data using $1 + M + N$ numbers, and is the best such rank 1 approximation (in L2 norm);
- $A2 = A1 + D(2,2) * U(:,2) * V(:,2)'$ is best rank 2 approximation.
- We can construct a sequence of approximations;
- If data does have patterns, we may be able to get an acceptable result with a rank $K$ approximation, where $K * (1 + M + N) \ll M * N$;

# Best Rank-K Approximation

Recall that, if the $m \times n$ matrix $A$ has SVD factors $U, D, V$, then the rank $k$ approximation to $A$ is

$$A_k = U(1:m, 1:k) * D(1:k, 1:k) * V'(1:n, 1:k).$$

## Theorem (Best Rank k Approximation)

*Let $A$ be an $m \times n$ matrix, and $k$ a rank satisfying $1 \leq k \leq \min(m, n)$. Let $A_k$ be the rank $k$ approximation to $A$ derived from the SVD. Then, $A$ is closer to $A_k$ than to any other rank $k$ matrix $B$, when error is measured in the Frobenius norm:*

$$||A - A_k||_F \leq ||A - B||_F$$

So, if we want to approximate $A$ by a rank k matrix, $A_k$ is our best choice. But this will only be a good idea if A has some underlying structure or linear behavior.

# Random Matrix Example

Given this table $A$ of 66 random numbers, is it possible to approximate it using fewer numbers?

| | | | | | |
|---|---|---|---|---|---|
| 1 | -5 | -2 | -3 | -1 | 9 |
| -7 | 2 | 1 | -7 | -8 | -10 |
| -7 | -1 | -8 | 6 | -5 | 5 |
| -5 | -3 | -9 | -4 | 8 | 6 |
| 7 | 7 | 1 | 1 | -7 | 7 |
| -5 | 2 | 6 | -7 | 7 | -8 |
| 6 | 1 | 9 | 2 | 1 | -2 |
| -5 | 8 | -7 | -5 | 10 | -5 |
| 9 | -4 | 1 | 3 | -8 | 6 |
| -3 | 5 | -1 | 4 | -1 | -1 |
| -6 | 5 | -10 | 5 | -8 | 8 |

norm(A,'fro') $= 47.25$

# Random Matrix Example

If we compute the SVD for this matrix, the singular values tell us whether information is concentrated in the first few dimensions:

```
D =

   29.6176         0         0         0         0         0
         0   25.8722         0         0         0         0
         0         0   18.8619         0         0         0
         0         0         0   13.6690         0         0
         0         0         0         0   10.4910         0
         0         0         0         0         0    5.8097
         0         0         0         0         0         0
         0         0         0         0         0         0
         0         0         0         0         0         0
         0         0         0         0         0         0
         0         0         0         0         0         0
```

Unfortunately, while the singular values go down, it seems a fairly gradual process.

# Compare A and $A_2$ in Columns 1 and 4

| Column 1 | | Column 4 | |
|---|---|---|---|
| A | A2 | A | A2 |
| 1 | 0.62 | -3 | 2.49 |
| -7 | -2.49 | -7 | -2.90 |
| -7 | -3.95 | 6 | 2.96 |
| -5 | -7.58 | -4 | -0.76 |
| 7 | 5.03 | 1 | 3.91 |
| -5 | -3.16 | -7 | -5.71 |
| 6 | 6.04 | 2 | -0.27 |
| -5 | -9.22 | -5 | -4.76 |
| 9 | 7.66 | 3 | 4.95 |
| -3 | -1.63 | 4 | 0.04 |
| -6 | -4.17 | 5 | 4.18 |

Most of the signs are right, and some of the values are...not too bad, but overall, it's not much of a match.

# Random Matrix Example

We can watch the Frobenius approximation error for $k$ from 0 to 6 using commands like:

```
norm(A–U(:,1:3)*D(1:3,1:3)*V(:,1:3)','fro')
```

```
||A-A0|| = 47.2546
||A-A1|| = 36.8212
||A-A2|| = 26.1998      <-- At A2, half the error persists.
||A-A3|| = 18.1840
||A-A4|| = 11.9923
||A-A5|| =  5.8097
||A-A6|| =  0.0000
```

# Tiny Matrix Example

This is also a table of 66 numbers, but it comes from a physical example, and that means there may be some patterns in the values.

| | | | | | |
|---|---|---|---|---|---|
| 10.0000 | 8.0400 | 9.1400 | 7.4600 | 8.0000 | 6.5800 |
| 8.0000 | 6.9500 | 8.1400 | 6.7700 | 8.0000 | 5.7600 |
| 13.0000 | 7.5800 | 8.7400 | 12.7400 | 8.0000 | 7.7100 |
| 9.0000 | 8.8100 | 8.7700 | 7.1100 | 8.0000 | 8.8400 |
| 11.0000 | 8.3300 | 9.2600 | 7.8100 | 8.0000 | 8.4700 |
| 14.0000 | 9.9600 | 8.1000 | 8.8400 | 8.0000 | 7.0400 |
| 6.0000 | 7.2400 | 6.1300 | 6.0800 | 8.0000 | 5.2500 |
| 4.0000 | 4.2600 | 3.1000 | 5.3900 | 19.0000 | 12.5000 |
| 12.0000 | 10.8400 | 9.1300 | 8.1500 | 8.0000 | 5.5600 |
| 7.0000 | 4.8200 | 7.2600 | 6.4200 | 8.0000 | 7.9100 |
| 5.0000 | 5.6800 | 4.7400 | 5.7300 | 8.0000 | 6.8900 |

norm(A,'fro') $=$ 68.13

# The Singular Values

Looking at the singular values, we suspect that a very large proportion of the information is in the first two components of the singular value decomposition.

```
65.9197         0         0         0         0         0
      0   15.6834         0         0         0         0
      0         0    4.8767         0         0         0
      0         0         0    3.9058         0         0
      0         0         0         0    2.8167         0
      0         0         0         0         0    2.0286
      0         0         0         0         0         0
      0         0         0         0         0         0
      0         0         0         0         0         0
      0         0         0         0         0         0
      0         0         0         0         0         0
```

Let's see if a 1 or 2 term approximation does well.

# Approximant $A_1$

```
A1=U(:,1)*S(1,1)*V(:,1)'
```

```
10.2141     8.1324     8.1899     8.0390     7.7625     6.8756
 8.6104     6.9678     6.9782     6.9562     7.5758     6.4772
12.5272     9.8381     9.9548     9.6424     8.2723     7.6088
 9.8335     7.9801     7.9842     7.9802     8.8580     7.5333
10.8390     8.6607     8.7113     8.5800     8.5208     7.4833
12.4329     9.7068     9.8420     9.4783     7.6832     7.2039
 7.0947     5.8846     5.8444     5.9605     7.5581     6.2056
 3.3005     4.3875     3.8085     5.4085    18.6737    12.8913
11.9884     9.3333     9.4727     9.0971     7.1656     6.7859
 7.2596     6.1177     6.0439     6.2530     8.6192     6.9342
 5.7494     5.0153     4.8990     5.2242     8.3904     6.5241
```

norm(A-A1,'fro') = 17.23

# Approximant $A_2$

```
A2=U(:,1:2)*S(1:2,1:2)*V(:,1:2)'
```

```
10.2141    8.1324    8.1899    8.0390    7.7625    6.8756
 8.6104    6.9678    6.9782    6.9562    7.5758    6.4772
12.5272    9.8381    9.9548    9.6424    8.2723    7.6088
 9.8335    7.9801    7.9842    7.9802    8.8580    7.5333
10.8390    8.6607    8.7113    8.5800    8.5208    7.4833
12.4329    9.7068    9.8420    9.4783    7.6832    7.2039
 7.0947    5.8846    5.8444    5.9605    7.5581    6.2056
 3.3005    4.3875    3.8085    5.4085   18.6737   12.8913
11.9884    9.3333    9.4727    9.0971    7.1656    6.7859
 7.2596    6.1177    6.0439    6.2530    8.6192    6.9342
 5.7494    5.0153    4.8990    5.2242    8.3904    6.5241
```

norm(A-A2,'fro') = 7.14

# Approximant $A_3$

```
A3=U(:,1:3)*S(1:3,1:3)*V(:,1:3)'
```

```
 10.0525    8.5868    8.3993    7.5761    7.8396    6.7830
  8.4444    7.4347    7.1933    6.4806    7.6550    6.3822
 13.4295    7.3008    8.7857   12.2273    7.8420    8.1255
  9.5482    8.7824    8.3539    7.1628    8.9940    7.3699
 10.7867    8.8079    8.7791    8.4301    8.5458    7.4533
 12.4866    9.5558    9.7725    9.6321    7.6576    7.2346
  6.8006    6.7118    6.2256    5.1178    7.6984    6.0371
  3.3802    4.1632    3.7051    5.6370   18.6357   12.9369
 11.5959   10.4370    9.9812    7.9727    7.3527    6.5611
  7.4222    5.6604    5.8332    6.7189    8.5417    7.0273
  5.7201    5.0978    4.9371    5.1401    8.4044    6.5073
```

norm(A-A3,'fro') = 5.22

# Compare A and $A_2$ in Columns 1 and 4

```
   Column 1                Column 4
   A       A2              A       A2

10.00  10.21             7.46   8.03
 8.00   8.61             6.77   6.95
13.00  12.52            12.74   9.64
 9.00   9.83             7.11   7.98
11.00  10.83             7.81   8.58
14.00  12.43             8.84   9.47
 6.00   7.09             6.08   5.96
 4.00   3.30             5.39   5.40
12.00  11.98             8.15   9.09
 7.00   7.25             6.42   6.25
 5.00   5.74             5.73   5.22
```

# Matrix Compression

In this small example, we could approximate the $11 \times 6$ entries of $A$ by a rank $k = 2$ approximation using 2*11 entries in $U$, 2 entries in $S$, and 2*6 entries of $V$, dropping from an exact representation using 66 numbers to an approximate representation that uses 36 numbers.

The singular values of the SVD can indicate when a matrix is strongly patterned. If the first few singular values are much larger than later ones, then the information in the array can be greatly compressed.

In real applications, the data sizes are in the thousands and millions. Using a typical approximation of rank $k = 10$ or $k = 20$ can result in enormous reductions in storage and efficiency.

# Example #4: Compression of an Image using SVD

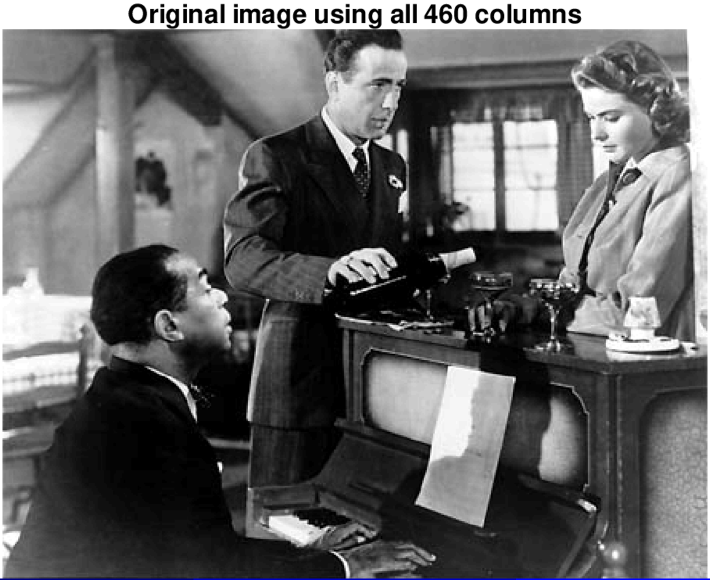
In a black and white image, a single number records the brightness of each pixel. Such an image is an $M \times N$ numeric array $A$. $M$ and $N$ are typically of the order of 1,000

We can think of each pixel of the image as a data item. An image will have many regions of the same color, and so it's likely that neighboring pixels in $A$ will have roughly the same values. In other words, we are not dealing with random data. And that means it is likely we can compress the data.

For any value $K$, we can ask the SVD to find $A_k$, the best rank $K$ approximation of $A$. The storage of $A_k$ can be much less than $A$ requires, while $A_k$ preserves most of the information in $A$.

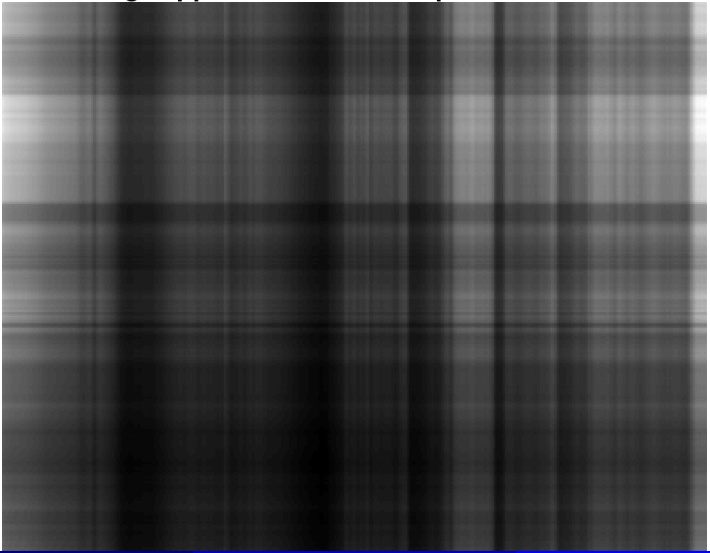
Let's take a sample image, compute its SVD, and then look at some typical approximating images.

# Image Compression: Original



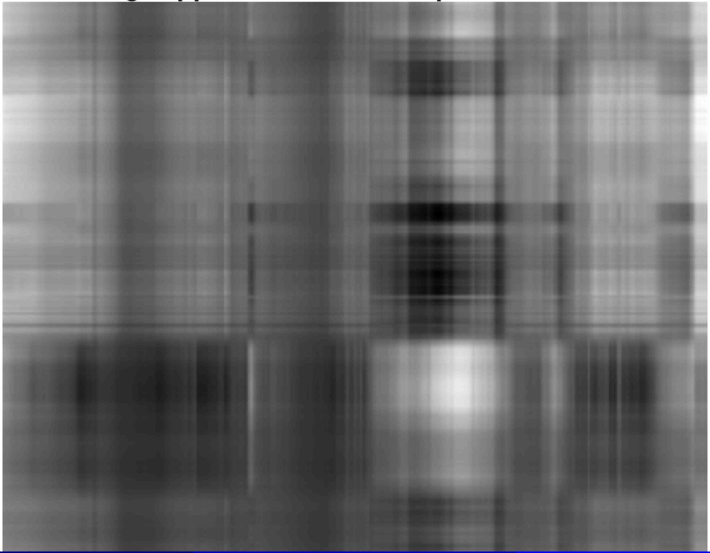
# Image Compression: Approximant A1

Image approximated with 1 optimal columns.



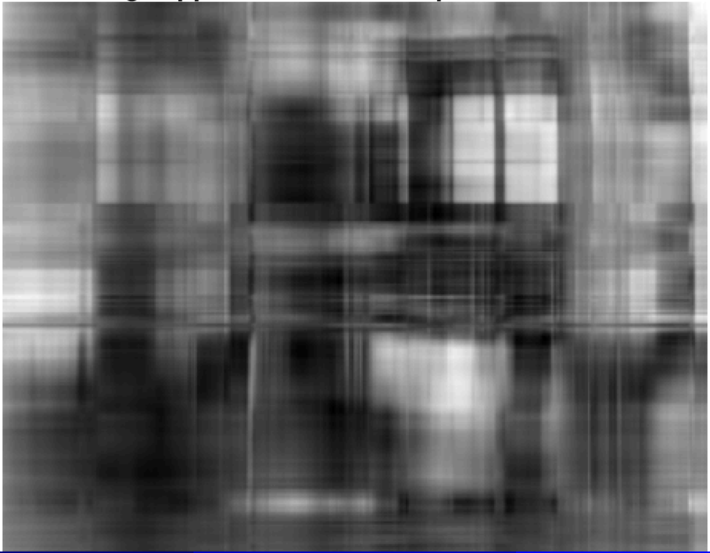
# Image Compression: Approximant A2

Image approximated with 2 optimal columns.



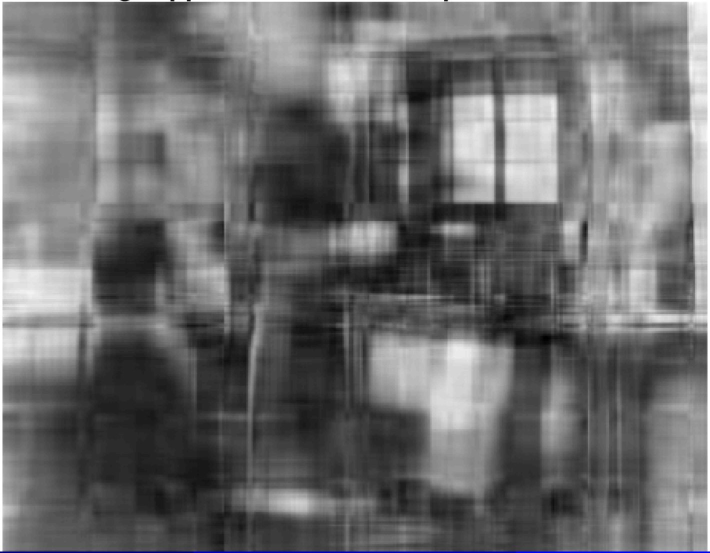
# Image Compression: Approximant A5

Image approximated with 5 optimal columns.



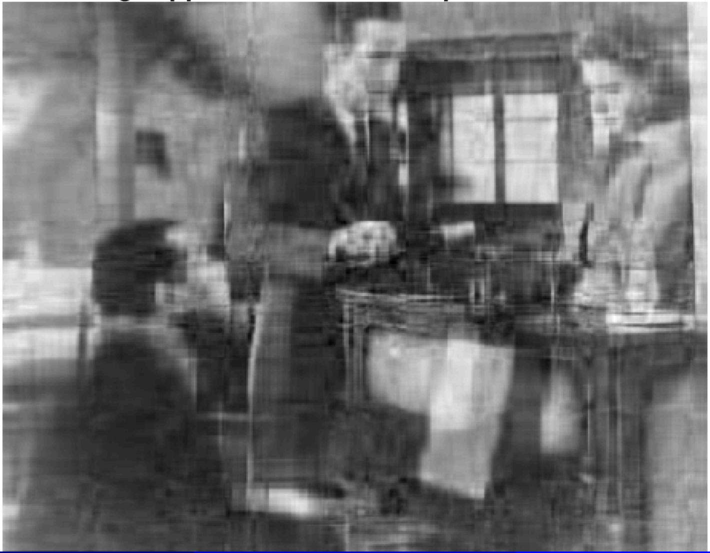
# Image Compression: Approximant A10

Image approximated with 10 optimal columns.



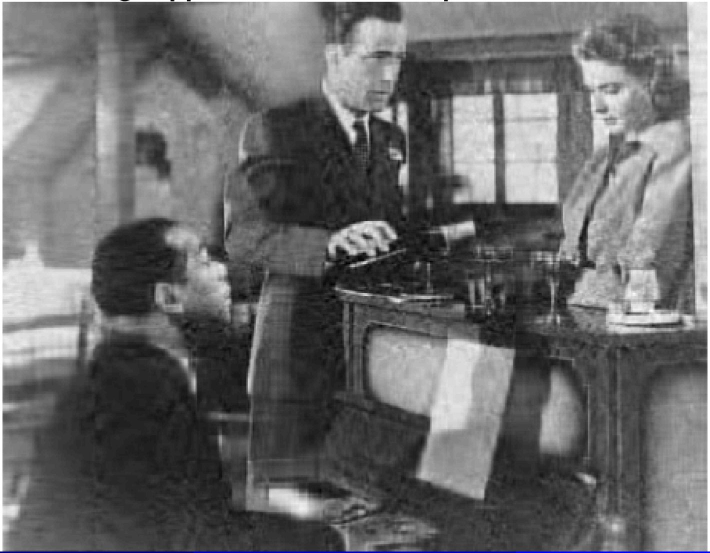
# Image Compression: Approximant A20

Image approximated with 20 optimal columns.



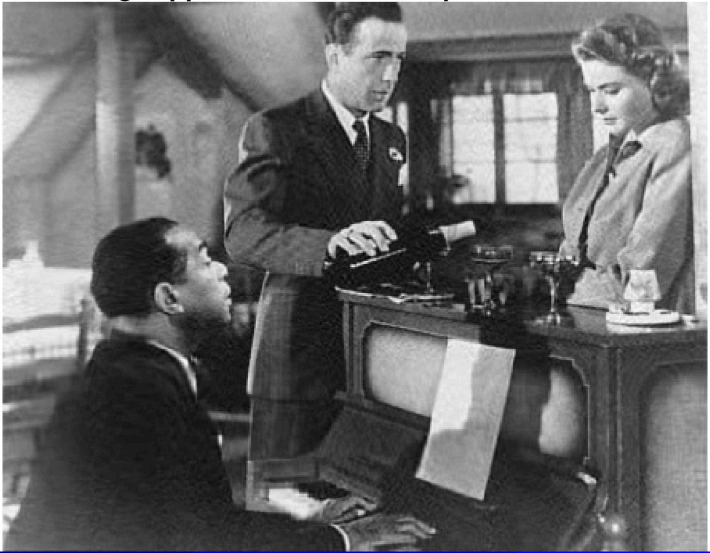
# Image Compression: Approximant A40

Image approximated with 40 optimal columns.



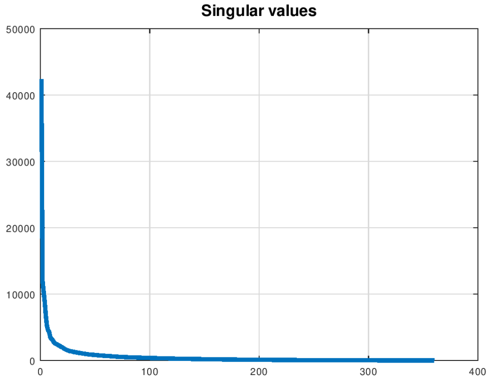
# Image Compression: Approximant A80

Image approximated with 80 optimal columns.



# Image Compression: The 360 Singular Values

Each singular value represents the size of the next addition to the approximation. For the photographic image, these values drop rapidly, suggesting that a low rank approximation gets most of the information.



# Image Compression: Comments

Assumptions:

- Choose the pixels of the image as our data items.
- Neighboring pixel values are usually related, not random.
- Groups of similar pixels have low information, and can be approximated.

Approximations:

- Rank 5 approximation shows light / dark patterns in the room;
- Rank 10 approxmation made $2\frac{1}{2}$ people visible;
- Rank 40 approximation shows all important features;

Conclusions:

- The photograph is a combination of information and minor details;
- SVD approximations expose important information first;
- Importance decreases with singular value size;
- A low rank SVD approximation may store all information we need;

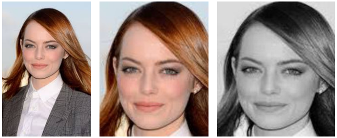
# Example #5: Facial Data: Preprocessing

Our image approximation treated each column of the image as a data item, and used the SVD to look for patterns and structure.

Can we use the SVD to process a collection of facial images?

In order to compare images of the same face, we will:

- seek images in which the subject faces forward;
- simplify the image from color to black and white (actually gray);
- crop each image to ear-to-ear and chin-to-top-of-head;
- resample all images to use $M \times N$ pixels;



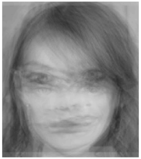
# Facial Data: Averaging

If the preprocessed images of a single subject are reasonably aligned, then we can take what amounts to an average of all the images, which we can use as our basic representation of the subject.

In MATLAB, a gray scale images stores only integer values between 0 and 255. To average an $M \times N \times K$ array $A$ of $K$ images, we convert to a numeric array, average, and convert back.

```
A_double = double ( A );
A_double_average = mean ( A_double, 3 );
A_average = uint8 ( A_double_average );
```

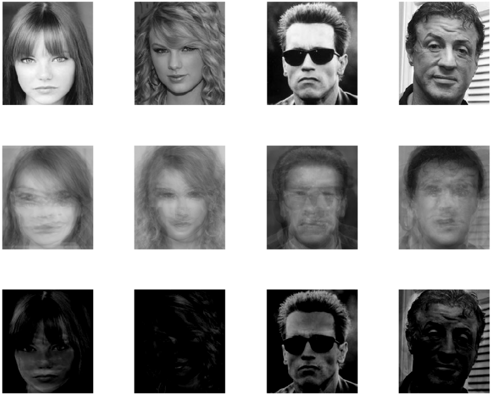
Average Emma Stone



# Facial Data: Subtract Average

Once we have the average face, we simplify our image collection by subtracting this average, so we have the basic face, plus a set of relatively small changes to the basic face.

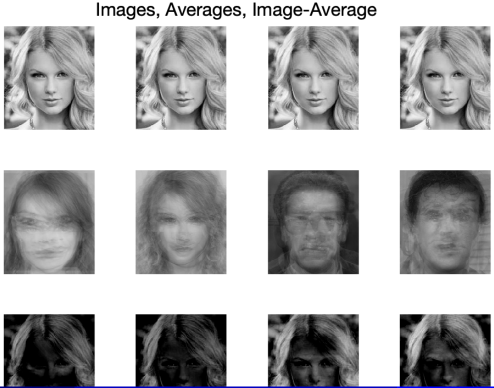
Images, Averages, Image-Average



# Facial Data: Classification by Distance from Average

Given a new face, measure distance to each average:

```
Image - Emma average      = 10769.2
Image - Taylor average    =  9864.3
Image - Arnold average    = 17007.8
Image - Sylvester average = 14557.9
```



Images, Averages, Image-Average

# Facial Data: A Basis for Face Space

Let $A$ be an $(M \times N) \times K$ array of images of a person after subtracting the average $A0$.

Apply the SVD: $A = U * D * V'$

The columns of $U$ indicate the facial variations needed to approximate all the images, with the most important ones coming first.

Thus, we can rebuild a face $F$ from the set $A$ by starting with the average, and adding small portions of each column of $U$.

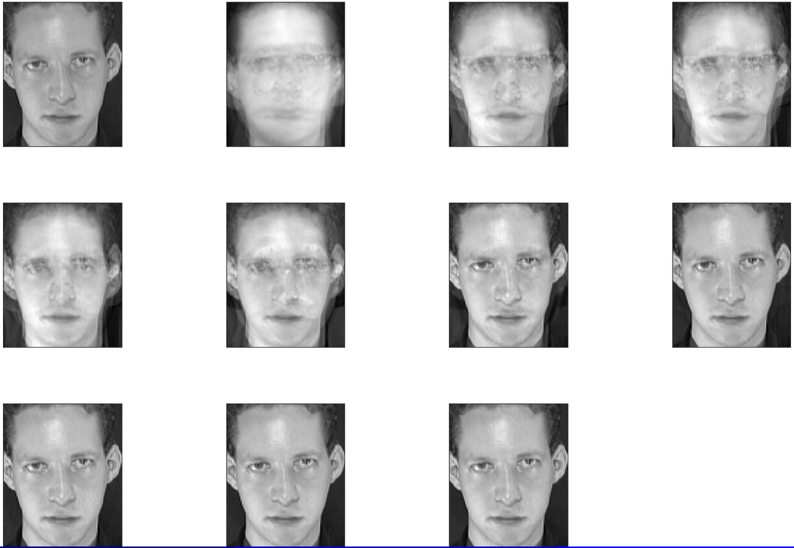
These amounts are the projections of the face onto each component. The first one is $c(1) = U(:,1)' \cdot F$, and we can compute them all by

$$c = U' * F$$

so that

$$F = A0 + U * c$$

# Facial Data: Reconstruction Example



Face | Average | Approx 1 : 10

# Facial Data: A Reduced Basis for Face Space

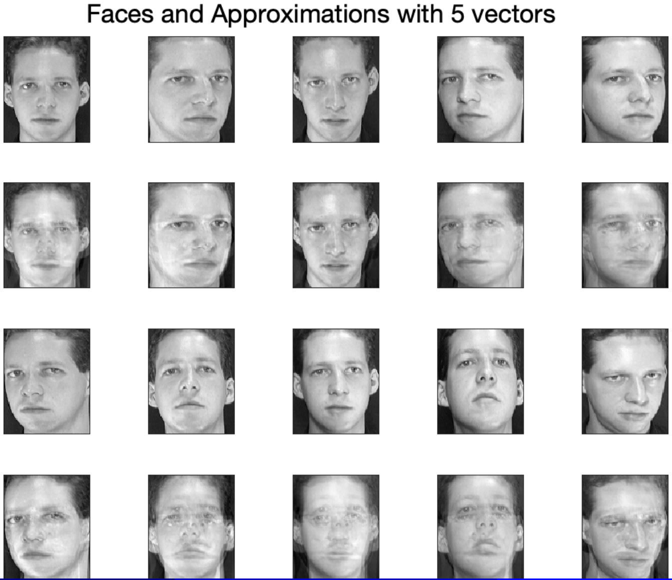
Using the SVD basis, each face can be **exactly** reconstructed from a small set of coefficients, and all the columns of the new basis $U$.

Suppose we are willing to accept approximations of our faces.

The columns of $U$ are ordered by importance. If we keep just the average face $A0$ and the first 5 columns of $U$, then we can throw away all our face data, replacing each face by just 5 coefficients.

If our original set was large, then this will be a huge saving.

# Facial Data: Reduced Basis Example



Faces and Approximations with 5 vectors

# Facial Data: Reduced Basis Example

We can use the average face and a few singular vectors to approximate each set of faces in our database

Each snapshot of a person can then be approximated by starting with the average face, and adding the appropriate amount of the singular vectors, as recorded in a coefficient vector $c$.

Given an unknown face, we can try to find the closest match in our database. Our first guess is to find the nearest averaged face. If that's not definitive, we can select several averaged faces that seem close, and look at the linear spaces defined by their singular vectors. Projection will give us the distance between the unknown face and each linear space, from which we can seek a match.

Obviously, much more is going on in this process, since the photo of the unknown face may be blurred, at an unusual angle, in bad light, and so on.

*These details keep facial recognition technicians happily employed!*

# Related MATLAB Labs

If you are interested in:

- the singular value decomposition, you can work though the example as part of Lab #2, *"Linear Algebra"*;
- the image compression application, you can work though this example as part of Lab #11, *"Principal Component Analysis"*;
- the facial data problem in Lab #14, *"Facial Recognition"*;

These should be available on

- the Blackboard site for this course;
- the website http://people.math.sc.edu/burkardt/classes/ml_2019/ml_2019.html
- the website http://people.sc.fsu.edu/∼jburkardt/classes/ml_2019/ml_2019.html

# Remarks

The SVD is not the only tool that can be used for facial recognition. Other methods are based on wavelets or the local cosine transform, and can provide superior performance.

In this simplified discussion, we have concentrated on the fact that the SVD can be used to compress the existing facial database, by identifying the most important features in each person's set of images.

We spent less time on the other vital task, of determining whether a new image corresponds to some subject in our database.

# Conclusion

Some machine learning applications involve an $M \times N$ array $A$ of data, were $N$ counts the number of data instances, and $M$ measures the size of each data item.

The array $A$ can be exactly represented by the full SVD, $A = U * D * V'$

Often, the data items have a great deal of similarity and correlation. Then the entries of the $D$ matrix will vary greatly in size.

Then the information in $A$ can be well approximated by a low rank approximation using only the first $K$ columns of the SVD factors.