14_SVD

March 28, 2017

1 14 Linear Algebra: Singular Value Decomposition

One can always decompose a matrix A

$$\mathsf{A} = \mathsf{U}\,\mathrm{diag}(w_j)\,\mathsf{V}^T \tag{1}$$
$$\mathsf{U}^T\mathsf{U} = \mathsf{U}\mathsf{U}^T = 1 \tag{2}$$
$$\mathsf{V}^T\mathsf{V} = \mathsf{V}\mathsf{V}^T = 1 \tag{3}$$

where U and V are orthogonal matrices and the w_j are the *singular values* that are assembled into a diagonal matrix W.

$$\mathsf{W} = \mathrm{diag}(w_j)$$

The inverse (if it exists) can be directly calculated from the SVD:

$$\mathsf{A}^{-1} = \mathsf{V}\mathrm{diag}(1/w_j)\mathsf{U}^T$$

1.1 Solving ill-conditioned coupled linear equations

```
In [1]: import numpy as np
```

1.1.1 Non-singular matrix

Solve the linear system of equations

$$\mathsf{A}\mathbf{x} = \mathbf{b}$$

Using the standard linear solver in numpy:

```
In [2]: A = np.array([
                [1, 2, 3],
                [3, 2, 1],
                [-1, -2, -6],
            ])
        b = np.array([0, 1, -1])
```

```
In [3]: np.linalg.solve(A, b)
```

```
Out[3]: array([ 0.83333333, -0.91666667,  0.33333333])
```

Using the inverse from SVD:

$$\mathbf{x} = \mathsf{A}^{-1}\mathbf{b}$$

```
In [4]: U, w, VT = np.linalg.svd(A)
        print(w)

[ 7.74140616  2.96605874  0.52261473]
```

First check that the SVD really factors $\mathsf{A} = \mathsf{U}\,\mathrm{diag}(w_j)\,\mathsf{V}^T$:

```
In [5]: U.dot(np.diag(w).dot(VT))

Out[5]: array([[ 1.,  2.,  3.],
               [ 3.,  2.,  1.],
               [-1., -2., -6.]])

In [6]: np.allclose(A, U.dot(np.diag(w).dot(VT)))

Out[6]: True
```

Now calculate the matrix inverse $\mathsf{A}^{-1} = \mathsf{V}\mathrm{diag}(1/w_j)\mathsf{U}^T$:

```
In [7]: inv_w = 1/w
        print(inv_w)

[ 0.1291755   0.33714774  1.91345545]

In [8]: A_inv = VT.T.dot(np.diag(inv_w)).dot(U.T)
        print(A_inv)

[[ -8.33333333e-01   5.00000000e-01  -3.33333333e-01]
 [  1.41666667e+00  -2.50000000e-01   6.66666667e-01]
 [ -3.33333333e-01  -1.38777878e-17  -3.33333333e-01]]
```

Check that this is the same that we get from `numpy.linalg.inv()`:

```
In [9]: np.allclose(A_inv, np.linalg.inv(A))

Out[9]: True
```

Now, *finally* solve (and check against `numpy.linalg.solve()`):

```
In [10]: x = A_inv.dot(b)
         print(x)
         np.allclose(x, np.linalg.solve(A, b))
```

```
[ 0.83333333 -0.91666667  0.33333333]
```

```
Out[10]: True
```

```
In [11]: A.dot(x)
```

```
Out[11]: array([ -8.88178420e-16,   1.00000000e+00,  -1.00000000e+00])
```

```
In [12]: np.allclose(A.dot(x), b)
```

```
Out[12]: True
```

1.1.2 Singular matrix

If the matrix A is *singular* (i.e., its rank (linearly independent rows or columns) is less than its dimension and hence the linear system of equation does not have a unique solution):

For example, the following matrix has the same row twice:

```
In [78]: C = np.array([
             [ 0.87119148,  0.9330127,  -0.9330127],
             [ 1.1160254,   0.04736717, -0.04736717],
             [ 1.1160254,   0.04736717, -0.04736717],
            ])
         b1 = np.array([ 2.3674474,  -0.24813392, -0.24813392])
         b2 = np.array([0, 1, 1])
```

```
In [79]: np.linalg.solve(C, b1)
```

```
        ---------------------------------------------------------------------------

        LinAlgError                               Traceback (most recent call last)

        <ipython-input-79-0d740b22028e> in <module>()
    ----> 1 np.linalg.solve(C, b1)


        /Users/oliver/anaconda3/lib/python3.5/site-packages/numpy/linalg/linalg.py
        382     signature = 'DD->D' if isComplexType(t) else 'dd->d'
        383     extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
    --> 384     r = gufunc(a, b, signature=signature, extobj=extobj)
        385
        386     return wrap(r.astype(result_t, copy=False))


        /Users/oliver/anaconda3/lib/python3.5/site-packages/numpy/linalg/linalg.py
         88
         89 def _raise_linalgerror_singular(err, flag):
```

```
---> 90       raise LinAlgError("Singular matrix")
     91
     92 def _raise_linalgerror_nonposdef(err, flag):


   LinAlgError: Singular matrix
```

NOTE: failure is not always that obvious: numerically, a matrix can be *almost* singular

```
In [80]: D = C.copy()
         D[2, :] = C[0] - 3*C[1]
         D

Out[80]: array([[ 0.87119148,  0.9330127 , -0.9330127 ],
                [ 1.1160254 ,  0.04736717, -0.04736717],
                [-2.47688472,  0.79091119, -0.79091119]])

In [81]: np.linalg.solve(D, b1)

Out[81]: array([ -1.70189831e+00,   2.34823174e+16,   2.34823174e+16])
```

Note that some of the values are huge, and suspiciously like the inverse of machine precision? Sign of a nearly singular matrix.

Now back to the example with C:

SVD for singular matrices If a matrix is *singular* or *near singular* then one can *still* apply SVD.

One can then compute the *pseudo inverse*

$$\mathsf{A}^{-1} = \mathsf{V}\text{diag}(\alpha_j)\mathsf{U}^T \tag{4}$$

$$\alpha_j = \begin{cases} \frac{1}{w_j}, & \text{if } w_j \neq 0 \\ 0, & \text{if } w_j = 0 \end{cases} \tag{5}$$

i.e., any singular $w_j = 0$ is being "augmented" by setting

$$\frac{1}{w_j} \to 0 \quad \text{if} \quad w_j = 0$$

in diag$(1/w_j)$.

Perform the SVD for the singular matrix C:

```
In [82]: U, w, VT = np.linalg.svd(C)
         print(w)

[  1.99999999e+00   1.00000000e+00   1.06263691e-33]
```

Note the third value $w_2 \approx 0$: sign of a singular matrix.

Test that the SVD really decomposes $\mathsf{A} = \mathsf{U}\,\text{diag}(w_j)\,\mathsf{V}^T$:

```
In [83]: U.dot(np.diag(w).dot(VT))
```

```
Out[83]: array([[ 0.87119148,  0.9330127 , -0.9330127 ],
                [ 1.1160254 ,  0.04736717, -0.04736717],
                [ 1.1160254 ,  0.04736717, -0.04736717]])
```

```
In [84]: np.allclose(C, U.dot(np.diag(w).dot(VT)))
```

```
Out[84]: True
```

There are the **singular values**:

```
In [85]: singular_values = np.abs(w) < 1e-12
         print(singular_values)
```

```
[False False  True]
```

Pseudo-inverse Calculate the **pseudo-inverse** from the SVD

$$\mathsf{A}^{-1} = \mathsf{V}\mathrm{diag}(\alpha_j)\mathsf{U}^T \tag{6}$$

$$\alpha_j = \begin{cases} \frac{1}{w_j}, & \text{if } w_j \neq 0 \\ 0, & \text{if } w_j = 0 \end{cases} \tag{7}$$

Augment:

```
In [88]: inv_w = 1/w
         inv_w[singular_values] = 0
         print(inv_w)
```

```
[ 0.5  1.   0. ]
```

```
In [89]: C_inv = VT.T.dot(np.diag(inv_w)).dot(U.T)
         print(C_inv)
```

```
[[-0.04736717  0.46650635  0.46650635]
 [ 0.5580127  -0.21779787 -0.21779787]
 [-0.5580127   0.21779787  0.21779787]]
```

Now solve the linear problem with SVD:

```
In [90]: x1 = C_inv.dot(b1)
         print(x1)
```

```
[-0.34365138  1.4291518  -1.4291518 ]
```

```
In [91]: C.dot(x1)
```

```
Out[91]: array([ 2.3674474 , -0.24813392, -0.24813392])
```

```
In [92]: C.dot(x1) - b1
```

```
Out[92]: array([  8.88178420e-16,  -1.11022302e-16,  -1.11022302e-16])
```

Thus, using the pseudo-inverse C^{-1} we can obtain solutions to the equation

$$\mathsf{C}\mathbf{x}_1 = \mathbf{b}_1$$

However, $\mathbf{x}_1$ is not the only solution: there's a whole line of solutions that are formed the special solution and a combination of the basis vectors in the *null space* of the matrix:

The (right) *kernel* or *null space* contains all vectors $\mathbf{x^0}$ for which

$$\mathsf{C}\mathbf{x^0} = 0$$

(The dimension of the null space corresponds to the number of singular values.) You can find a basis that spans the null space. Any linear combination of null space basis vectors will also end up in the null space when $\mathbf{A}$ is applied to it.

Specifically, if $\mathbf{x}_1$ is a special solution and $\lambda_1 \mathbf{x}_1^0 + \lambda_2 \mathbf{x}_2^0 + \dots$ is a vector in the null space then

$$\mathbf{x} = \mathbf{x}_1 + (\lambda_1 \mathbf{x}_1^0 + \lambda_2 \mathbf{x}_2^0 + \dots)$$

is **also a solution** because

$$\mathsf{C}\mathbf{x} = \mathsf{C}\mathbf{x^0} + \mathsf{C}(\lambda_1 \mathbf{x}_1^0 + \lambda_2 \mathbf{x}_2^0 + \dots) = \mathsf{C}\mathbf{x^0} + 0 = \mathbf{b}_1 + 0 = \mathbf{b}_1$$

The λ_i are arbitrary real numbers and hence there is an infinite number of solutions.

In SVD:

- The columns $U_{\cdot,i}$ of U (i.e. `U.T[i]` or `U[:, i]`) corresponding to non-zero w_i, i.e. $\{i : w_i \neq 0\}$, form the basis for the *range* of the matrix A.
- The columns $V_{\cdot,i}$ of V (i.e. `V.T[i]` or `V[:, i]`) corresponding to zero w_i, i.e. $\{i : w_i = 0\}$, form the basis for the *null space* of the matrix A.

Note that `x1` can be written as a linear combination of `U.T[0]` and `U.T[1]`:

```
In [93]: x1
```

```
Out[93]: array([-0.34365138,  1.4291518 , -1.4291518 ])
```

```
In [94]: U.T
```

```
Out[94]: array([[ -7.07106782e-01,  -4.99999999e-01,  -4.99999999e-01],
               [  7.07106780e-01,  -5.00000001e-01,  -5.00000001e-01],
               [ -8.23369199e-17,  -7.07106781e-01,   7.07106781e-01]])
```

```
In [95]: VT
```

```
Out[95]: array([[-0.8660254 , -0.35355339,  0.35355339],
                [-0.5       ,  0.61237244, -0.61237244],
                [-0.        , -0.70710678, -0.70710678]])
```

```
In [96]: U.T[0].dot(x1), U.T[1].dot(x1)
```

```
Out[96]: (0.24299822382783764, -0.24299822305983237)
```

```
In [97]: VT[2].dot(x1)
```

```
Out[97]: 0.0
```

```
In [98]: U.T[0].dot(x1) * U.T[0] + U.T[1].dot(x1) * U.T[1] + 2 * VT[2]
```

```
Out[98]: array([-0.34365138, -1.41421356, -1.41421356])
```

Thus, **all** solutions are

```
x1 + lambda * VT[2]
```

The solution vector x_2 is in the null space:

```
In [99]: x2 = C_inv.dot(b2)
         print(x2)
         print(C.dot(x2))
         print(C.dot(x2) - b2)

[ 0.9330127  -0.43559574  0.43559574]
[ -5.55111512e-16   1.00000000e+00   1.00000000e+00]
[ -5.55111512e-16   2.22044605e-16   2.22044605e-16]
```

```
In [100]: C.dot(10*x2)
```

```
Out[100]: array([ -4.44089210e-15,   1.00000000e+01,   1.00000000e+01])
```

```
In [101]: C.dot(VT[2])
```

```
Out[101]: array([  0.00000000e+00,  -6.93889390e-18,  -6.93889390e-18])
```

```
In [34]: VT[2]
```

```
Out[34]: array([-0.        , -0.70710678, -0.70710678])
```

```
In [102]: null_basis = VT[singular_values]
```

```
In [103]: C.dot(null_basis.T)
```

```
Out[103]: array([[  0.00000000e+00],
                 [ -6.93889390e-18],
                 [ -6.93889390e-18]])
```

1.2 SVD for fewer equations than unknowns

N equations for M unknowns with $N < M$:

- no unique solutions (underdetermined)
- $M - N$ dimensional family of solutions
- SVD: at least $M - N$ zero or negligible w_j: columns of V corresponding to singular w_j span the solution space when added to a particular solution.

Same as the above **Solving ill-conditioned coupled linear equations**.

1.3 SVD for more equations than unknowns

N equations for M unknowns with $N > M$:

- no exact solutions in general (overdetermined)
- but: SVD can provide best solution in the least-square sense

$$\mathbf{x} = \mathsf{V}\,\mathrm{diag}(1/w_j)\,\mathsf{U}^T\,\mathbf{b}$$

where

- $\mathbf{x}$ is a M-dimensional vector of the unknowns,
- V is a $M \times N$ matrix
- the w_j form a square $M \times M$ matrix,
- U is a $M \times N$ matrix (and U^T is a $N \times M$ matrix), and
- $\mathbf{b}$ is the N-dimensional vector of the given values

It can be shown that $\mathbf{x}$ minimizes the residual

$$\mathbf{r} := |\mathsf{A}\mathbf{x} - \mathbf{b}|.$$

(For a $N \leq M$, one can find $\mathbf{x}$ so that $\mathbf{r} = 0$ – see above.)
(In the following, $\mathbf{x}$ will correspond to the N parameter values of the model and M is the number of observations.)

1.3.1 Linear least-squares fitting

This is the *liner least-squares fitting problem*: Given N data points (x_i, y_i) (where $1 \leq i \leq N$), fit to a linear model $y(x)$, which can be any linear combination of M functions of x.

For example, if we have N functions x^k with parameters a_k

$$y(x) = a_1 + a_2 x + a_3 x^2 + \cdots + a_M x^{M-1}$$

or in general

$$y(x) = \sum_{k=1}^{M} a_k X_k(x)$$

The goal is to determine the M coefficients a_k.

Define the **merit function**

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - \sum_{k=1}^{M} a_k X_k(x_i)}{\sigma_i} \right]^2$$

(sum of squared deviations, weighted with standard deviations σ_i on the y_i).

Best parameters a_k are the ones that *minimize* χ^2.

Design matrix A ($N \times M$, $N \geq M$), vector of measurements $\mathbf{b}$ (N-dim) and parameter vector $\mathbf{a}$ (M-dim):

$$A_{ij} = \frac{X_j(x_i)}{\sigma_i} \tag{8}$$

$$b_i = \frac{y_i}{\sigma_i} \tag{9}$$

$$\mathbf{a} = (a_1, a_2, \ldots, a_M) \tag{10}$$

Minimum occurs when the derivative vanishes:

$$0 = \frac{\partial \chi^2}{\partial a_k} = \sum_{i=1}^{N} {\sigma_i}^{-2} \left[y_i - \sum_{k=1}^{M} a_k X_k(x_i) \right] X_k(x_i), \quad 1 \leq k \leq M$$

(M coupled equations)

$$\sum_{j=1}^{M} \alpha_{kj} a_j = \beta_k \tag{11}$$

$$\alpha \mathbf{a} = \beta \tag{12}$$

with the $M \times M$ matrix

$$\alpha_{kj} = \sum_{i=1}^{N} \frac{X_j(x_i) X_k(x_i)}{\sigma_i^2} \tag{13}$$

$$\alpha = \mathsf{A}^T \mathsf{A} \tag{14}$$

and the vector of length M

$$\beta_k = \sum_{i=1}^{N} \frac{y_i X_k(x_i)}{\sigma_i^2} \tag{15}$$

$$\beta = \mathsf{A}^T \mathbf{b} \tag{16}$$

The inverse of α is related to the uncertainties in the parameters:

$$\mathsf{C} := \alpha^{-1}$$

in particular

$$\sigma(a_i) = C_i i$$

(and the C_{ij} are the co-variances).

Solution of the linear least-squares fitting problem with SVD We need to solve the overdetermined system of M coupled equations

$$\sum_{j=1}^{M} \alpha_{kj} a_j = \beta_k \tag{17}$$

$$\alpha \mathbf{a} = \beta \tag{18}$$

SVD finds $\mathbf{a}$ that minimizes

$$\chi^2 = |\mathsf{A}\mathbf{a} - \mathbf{b}|$$

The errors are

$$\sigma^2(a_j) = \sum_{i=1}^{M} \left(\frac{V_{ji}}{w_i}\right)^2$$

Example Synthetic data

$$y(x) = 3\sin x - 2\sin 3x + \sin 4x$$

with noise r added (uniform in range $-5 < r < 5$).

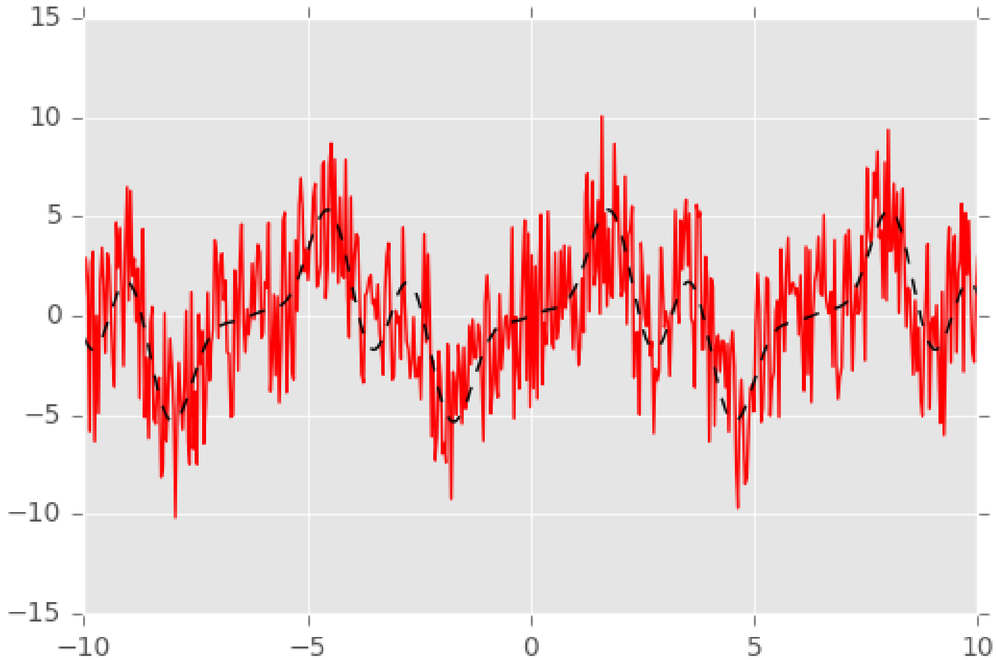
```
In [37]: import matplotlib
         import matplotlib.pyplot as plt
         %matplotlib inline
         matplotlib.style.use('ggplot')

         import numpy as np
```

```
In [38]: def signal(x, noise=0):
             r = np.random.uniform(-noise, noise, len(x))
             return 3*np.sin(x) - 2*np.sin(3*x) + np.sin(4*x) + r
```

```
In [59]: X = np.linspace(-10, 10, 500)
         Y = signal(X, noise=5)
```

```
In [60]: plt.plot(X, Y, 'r-', X, signal(X, noise=0), 'k--')
```

```
Out[60]: [<matplotlib.lines.Line2D at 0x10e84ceb8>,
          <matplotlib.lines.Line2D at 0x10e852198>]
```




```
In [61]: def fitfunc(x, a):
             return a[0]*np.cos(x) + a[1]*np.sin(x) + \
                    a[2]*np.cos(2*x) + a[3]*np.sin(2*x) + \
                    a[4]*np.cos(3*x) + a[5]*np.sin(3*x) + \
                    a[6]*np.cos(4*x) + a[7]*np.sin(4*x)

         def basisfuncs(x):
             return np.array([np.cos(x), np.sin(x),
                              np.cos(2*x), np.sin(2*x),
                              np.cos(3*x), np.sin(3*x),
                              np.cos(4*x), np.sin(4*x)])

In [62]: M = 8
         sigma = 1.
         alpha = np.zeros((M, M))
         beta = np.zeros(M)
         for x in X:
             Xk = basisfuncs(x)
             for k in range(M):
                 for j in range(M):
                     alpha[k, j] += Xk[k]*Xk[j]
         for x, y in zip(X, Y):
             beta += y * basisfuncs(x)/sigma

In [63]: U, w, VT = np.linalg.svd(alpha)
         V = VT.T
```

In this case, the singular values do not immediately show if any basis functions are superfluous (this would be the case for values close to 0).

```
In [64]: w
```

```
Out[64]: array([ 296.92809624,  282.94804954,  243.7895787 ,  235.7300808 ,
                 235.15938555,  235.14838812,  235.14821093,  235.14821013])
```

... nevertheless, remember to routinely mask any singular values or close to singular values:

```
In [65]: w_inv = 1/w
         w_inv[np.abs(w) < 1e-12] = 0
         alpha_inv = V.dot(np.diag(w_inv)).dot(U.T)
```

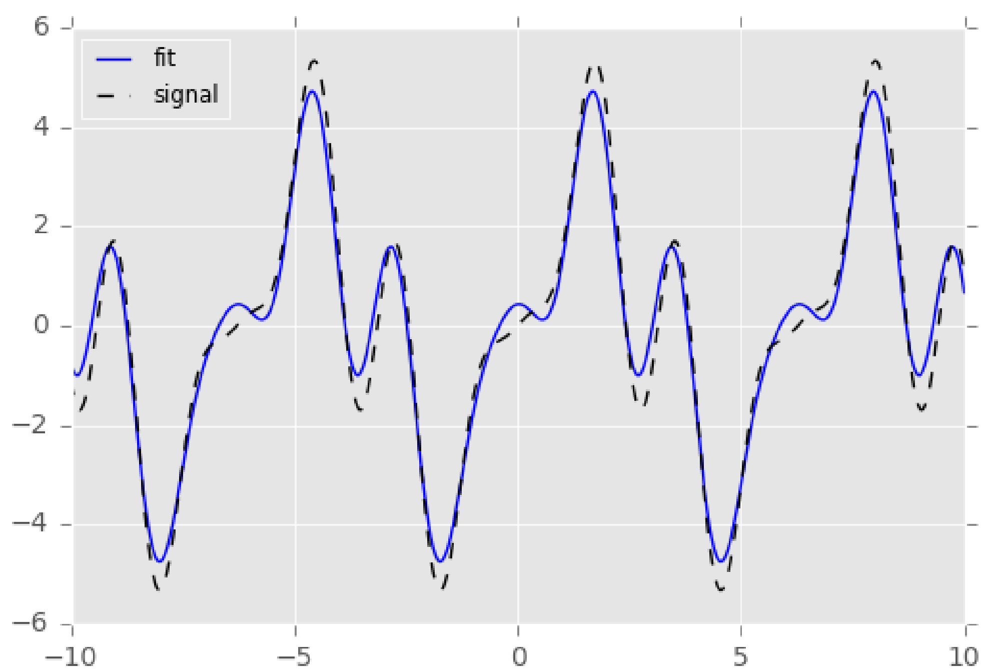
Compare the fitted values to the original parameters $a_j = 0, +3, 0, 0, 0, -2, 0, +1$.

```
In [66]: a_values = alpha_inv.dot(beta)
         print(a_values)
```

```
[-0.05602761  2.76553973  0.25531225 -0.03780974 -0.05668003 -1.76371356
  0.28272354  0.68902357]
```

```
In [67]: plt.plot(X, fitfunc(X, a_values), 'b-', label="fit")
         plt.plot(X, signal(X, noise=0), 'k--', label="signal")
         plt.legend(loc="best", fontsize="small")
```

```
Out[67]: <matplotlib.legend.Legend at 0x10e8bd400>
```




```
In [ ]:
```