Condition number ¶

- (Wikipedia) In the field of numerical analysis, the condition number of a function with respect to an argument measures how much the output value of the function can change for a small change in the input argument. This is used to measure how sensitive a function is to changes or errors in the input, and how much error in the output results from an error in the input.
- There are many different condition numbers, although people usually refer to the condition number for inversion. In general, a condition number applies not only to a particular matrix, but also to the problem being solved. Are we inverting the matrix, finding its eigenvalues, or computing the exponential? A matrix can be poorly conditioned for inversion while the eigenvalue problem is well conditioned.
- The value of condition number for matrix inversion depends the specific norm used. If we use the L_2 norm, then the condition number C is the **ratio of the largest to smallest singular value in the singular value decomposition** of a matrix. The $\log_b C$ is an estimate of how many base b digits are lost in solving a linear system with that matrix. In other words, it estimates worst-case loss of precision.
- A system is said to be singular if the condition number is infinite, and ill-conditioned if it is too large, where "too large" means roughly log(C)>~ the precision of matrix entries.

scipy.linalg vs numpy.linalg

scipy.linalg contains all the functions in numpy.linalg. plus some other more advanced ones not contained in numpy.linalg. Another advantage of using scipy.linalg over numpy.linalg is that it is always compiled with BLAS/LAPACK support, while for numpy this is optional. Therefore, the scipy version might be faster depending on how numpy was installed. Therefore, unless you don't want to add scipy as a dependency to your numpy program, use scipy.linalg instead of numpy.linalg.

Time complexity of matrix operations

- Most square matrix operations such as multiplication, inversion, SVD, determinant have a standard $O(n^3)$ complexity and an optimal complexity of $O(n^{2.373})$. Is SVD has such an optimal complexity. Normally, SVD has a $O(mn^2)$ complexity where $m \geq n$.
- Most machine learning problems are over determined, meaning $m \geq n$, or $m \gg n$. Therefore, the typical SVD complexity of $O(mn^2)$ is not a big problem. If we use left inverse, or projection matrix, the obtained square matrix is with shape of $n \times n$, then the complexity is not big due to the small number of features n.
- See complexity of other mathematical operators from the following link. https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations (https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations)

Matrix multiplication with numpy array

- If both a and b are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both a and b are 2-D arrays, it is matrix multiplication, but using **matmul or a @ b** is preferred.
- If a is an N-D array and b is a 1-D array, it is a sum product over the last axis of a and b.
- If a is an N-D array and b is an M-D array (where M>=2), it is a sum product over the last axis of a and the second-to-last axis of b: `dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])` . Note this is generalized from the matrix multiplication. $C_{ij} = \sum_{k=1}^{n} a_{i,k} b_{kj}$.
- The standard operations `*, +, -, /` work **element-wise on arrays**. Instead, you could try using numpy.matrix, and * will be treated like matrix multiplication.

Singular value decomposition

In [ ]:
```
import numpy as np
from scipy import linalg
A = np.array([[1,2,3],[4,5,6]])
M,N = A.shape
print(A.shape)
U,s,Vh = linalg.svd(A)

Sig = linalg.diagsvd(s,M,N)
print(U)
print(Sig)
print(Vh)
U.dot(Sig.dot(Vh)) #check computation
```

Matrix inverse, pseudo inverse, generalized inverse

- Many same-name library functions are with different meaning. scipy.linalg.pinv shares the same algorithm as scipy.lstsq. Both of them calculate Moore-Penrose pseudo inverse. Also note the same Moore-Penrose pseudo-inverse might sometimes refers to the left-inverse of a matrix. scipy.linalgl.pinv2, however, calculate Moore-Penrose pseudo-inverse using SVD.
- In numpy, numpy.linalg.pinv calcuates pseudo-inverse with SVD, which is different from that of scipy.pinv.

Not defined issues

- Make sure the problem is really eligible to handling with the following approaches, but rather not due to errors in the equation.
- A normal way is to add or subtract, depending specific situations, a small number to, e.g., the following issues. We should tune the small number to see whether it affect the performance or results.
  - Divided by zero.
  - Negative number for square root.
  - Log (0).

Reshape and squeeze

Dimension basics

The different usage of the word "dimension" below.

- A multi-dimensional array is an array with more than one level or dimension. For example, a 2D array, or two-dimensional array, is an array of arrays, meaning it is a matrix of rows and columns.
- The shape of an array is a tuple of integers giving the size of the array **along each dimension.**
- A 4×2 matrix has 4 dimensional column vector, or $\in \mathbb{R}^4$, and 2 dimensional row vector. If the two column vectors are linearly independent, the column space spanned by them is 2, otherwise it is 1 dimension. The dimension of row space spanned by 4 rows must the same as the dimension of column space by 2 columns, which is the rank of this matrix.
- In numpy, an array with shape, e.g., $(4, 3, 3, 2)$ is a four-dimensional 'array'. There are 4 elements in this multi-dimensional array, each element is a $(3, 3, 2)$ array. For each of this four $(3, 3, 2)$ array, there are 3 elements it it, and each element is a $(3, 2)$ array. Note the shape of the same word 'array' is different for different context.
- When displaying numpy array with print, if we have n `[` in the left, then it is n dimensional array.
- See examples below. **Should be very familiar with the printed out format, and write out its shape from the format. Also, be familiar to write out a displayed form if a shape is given.**

In [2]:
```
import numpy as np
x = np.random.randn(4) # in numpy shape (4,) is different from the shape (4,1)
print(x,x.shape)
print('----------------------------')
x = np.random.randn(4,1) # 4 elements, each element is 1 dimensional vector.
print(x,x.shape)
print('----------------------------')
x = np.random.randn(4,3) # 4 elements, each element is a 3 dimensional vector.
print(x,x.shape)
print('----------------------------')
x = np.random.randn(4,3,3) # 4 elements, each element is a 3x3 'vector' or matrix,
print(x,x.shape)
print('----------------------------')
x = np.random.randn(4, 3, 3, 2)
print(x)
```

```
[-1.97123256 -0.72001376  0.45677402 -0.93323281] (4,)
----------------------------
[[ 0.00969311]
 [-0.30932668]
 [-1.17867124]
 [ 0.07923948]] (4, 1)
----------------------------
[[-1.45290522  2.29477604 -0.80650091]
 [-0.02632981  0.35321895 -0.91948831]
 [-1.40971185 -0.44581214 -1.9467669 ]
 [-0.59215511  0.01320084 -1.83264454]] (4, 3)
----------------------------
[[[ 1.30754871 -1.46091596  1.22704889]
  [ 0.78006592  2.49484965 -0.46486018]
  [ 0.58291458  1.0815692  -0.61865608]]

 [[ 1.65768983 -0.24345759 -0.52642296]
  [ 0.21688405 -1.83891901 -2.25274752]
  [-1.12630928  2.01549332  0.09671539]]

 [[ 2.34857179  0.61086973  0.49920229]
  [-2.15336297 -0.09491424  0.09217111]
  [ 0.37794753 -0.24225248  0.85499224]]

 [[ 0.68912114 -0.36782294  0.65732845]
  [ 0.28176934 -0.69667337  1.54064036]
```

```
  [-0.5510123  -0.89298179 -0.69312759]]] (4, 3, 3)
----------------------------
[[[[-0.25082334 -0.91692464]
   [-0.18481464  1.19576289]
   [-1.18519231  0.74784042]]

  [[ 0.2205176  -1.47586764]
   [ 1.65708882  0.82925786]
   [-0.55001603  0.63880321]]

  [[ 0.40321448  0.16556041]
   [-1.92394237 -1.19136447]
   [-0.91843099 -0.17295797]]]


 [[[-0.32970934  0.31034076]
   [-0.38047872  1.69525935]
   [-0.09511759 -0.25699424]]

  [[ 0.73950744  0.20981964]
   [-0.45850501 -0.3265745 ]
   [-1.28702917 -0.29286849]]

  [[ 0.72992666  0.14755271]
   [-0.38527136 -1.7200559 ]
   [-0.04857366  0.1145018 ]]]


 [[[ 0.31024243 -0.67704094]
   [ 1.50023395  0.65948793]
   [ 0.67568156 -0.89773566]]

  [[-1.45816519 -0.39373317]
   [ 0.10107887  0.89551625]
   [ 1.75678349 -1.3761228 ]]

  [[-0.1337154  -1.00732953]
   [-0.29903102 -0.88064432]
   [-0.23053089 -0.29815651]]]


 [[[-0.1018546  -1.60287637]
   [ 0.90284698 -0.0177537 ]
```

```
 [-1.16121619  2.11104648]]

[[ 0.38247417 -1.24232509]
 [-0.77742505 -0.01292019]
 [-0.17990679 -0.66749976]]

[[-0.06411546 -0.16845881]
 [ 0.06937835 -0.53512194]
 [-1.28499735 -0.23119401]]]
```

Re-shape with explicit dimensions to avoid confusion

- The best way to think about NumPy arrays is that they consist of two parts, a data buffer which is just a block of raw elements, and a view which describes how to interpret the data buffer. For example, if we create an array of 12 integers (0,1,...11). Here the shape (12,) means the array is indexed by a single index which runs from 0 to 11. We can also reshape it into (3,4), (4,3), or even (1, 2, 1, 6, 1). Check the link https://stackoverflow.com/questions/22053050/difference-between-numpy-array-shape-r-1-and-r (https://stackoverflow.com/questions/22053050/difference-between-numpy-array-shape-r-1-and-r) for details.
- A trick for reshaping is using -1 to denote the unknown dimension. For example, to flatten an image data set X.shape = (209, 64, 64, 3) to (209,unknown number), we can do `array.reshape(X.shape[0],-1)` .
- Explicitly reshape vector to column vector or row vector can avoid hidden bugs and be consistent with mathematical equations. In most cases, this will not affect performance. However, **it does affect in some special cases**, as shown in the following example. So it is **not always necessary** to reshape a dimension from (N,) to (N,1) etc.
- For different functions, the syntax to explicitly set dimensions are different. For example, we have a = np.zeros((1000000,1)) but a = np.random.rand(1000000,1). If we cannot specify dimension during data construction, we almost always can use reshape() to do it later.

Numpy squeeze

- numpy.squeeze(a, axis=None) removes single-dimensional entries from the shape of an array. 'axis' selects a subset of the single-dimensional entries in the shape. If an axis is selected with shape entry greater than one, an error is raised.
- Write out the shape of `x = np.array([[[0], [1], [2]]])` .

In [4]:
```
import numpy as np
x = np.array([[[0], [1], [2]]])
print(x)
print(x.shape)
y = np.squeeze(x)
print(y.shape)
print(np.squeeze(x, axis=(2,)).shape)
```

```
[[[0]
  [1]
  [2]]]
(1, 3, 1)
(3,)
(1, 3)
```

In [5]:
```
import numpy as np
x = np.arange(9).reshape(1,3,3) #better than x = np.array([0,1,2...8])

print('Array X:')
print (x)
print ('\n')
y = np.squeeze(x)

print ('Array Y:')
print (y)
print ('\n')

print ('The shapes of X and Y array:')
print (x.shape, y.shape)
```

```
Array X:
[[[0 1 2]
  [3 4 5]
  [6 7 8]]]


Array Y:
[[0 1 2]
 [3 4 5]
 [6 7 8]]


The shapes of X and Y array:
(1, 3, 3) (3, 3)
```

Dimension check and type check

- Perform dimension check during calculation is always a good way to check hidden bugs. Two quantities with different dimensions will be added and give a new dimension. This is very dangerous. So at least in the debugging period, use as many assert statements as possible.
- In numpy, (n,) is for 1D array and (n,1) is for column vector.
- Examples
  - assert(Y_prediction.shape == (1, m))
  - cost = np.squeeze(cost)

- isinstance(cost, float)

Sort eigenvectors w.r.t eigenvalues

In [ ]:
```
# C is a symmetric matrix and so it can be diagonalized:
l, principal_axes = la.eig(C)

# sort results wrt. eigenvalues. This indicates that the 'raw' results are not ordered? Always?
#Returns the indices that would sort this array.
idx = l.argsort()[::-1]

l, principal_axes = l[idx], principal_axes[:, idx]
```

Compare two array to within a relative or absolute error.

In [ ]:
```
np.allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)
# Returns True if two arrays are element-wise equal within a tolerance.
```

Timing the big difference between explicit loops and vectorization

See more about vectorization in https://stackoverflow.com/questions/35091979/why-is-vectorization-faster-in-general-than-loops (https://stackoverflow.com/questions/35091979/why-is-vectorization-faster-in-general-than-loops). Vectorization uses special hardware. Unlike a multicore CPU, for which each of the parallel processing units is a fully functional CPU core, vector processing units can perform only simple operations, and all the units perform the same operation at the same time, operating on a sequence of data values (a vector) simultaneously. This is not same as the traditional multi-threading.

- Avoid explicit loops as much as possible and favor vectorization.
- Avoid using list, DataFrames to calculate and favor numpy arrays.

Estimating covariance matrix

Why is preferable to do PCA with the SVD

It is well known that estimating the covariance matrix by the ML estimator (sample covariance) can be very numerically unstable (in high dimensions). An answer to address this is provided in the following link. https://stats.stackexchange.com/questions/245712/intuition-as-to-why-estimates-of-a-covariance-matrix-are-numerically-unstable (https://stats.stackexchange.com/questions/245712/intuition-as-to-why-estimates-of-a-covariance-matrix-are-numerically-unstable)

Catastrophic cancellation due to small correlation

$\operatorname{cov}(X, Y) = E[X, Y] - E[X]E[Y]$. When $E[XY] \approx E[X]E[Y]$, this might lead to catastrophic cancellation when computing with floating point arithmetic and thus should be avoided in computing when the data **has not been centered before**. Numerically stable algorithm should be preferred in this case. https://en.wikipedia.org/wiki/Covariance (https://en.wikipedia.org/wiki/Covariance).
If the above approximate equation is true, it indicates the two variables are almost independent. Therefore, the correlation (scaled covariance) would normally small. Thus we will have rank-deficient matrix. Note however, tiny covariance indicates linearly independent but not generally independent.

Effective rank

It is critical to use a reasonable threshold to determine the effective rank. If the threshold is too small, then the effective rank determined is meaningless.

In [2]:

```
#Calculate the effective rank of a matrix
import numpy as np
from numpy.linalg import matrix_rank

#Using numpy.linalg.matrix_rank
matrix_rank(np.eye(4)) # Full rank matrix. eye(4) is a 4x4 identity matrix
I=np.eye(4)
I[-1,-1] = 0. # rank deficient matrix. Note the syntax for accessing matrix element.
print(matrix_rank(I)) #rank = 3.

print(matrix_rank(np.ones((4,))))  # rank = 1
print(matrix_rank(np.zeros((4,)))) # rank = 0


#When figuring out singular value cutoff, using a matrix after normalization.
#Otherwise its range is normally very huge.
def rank(A, cutoff):
    u, s, vh = np.linalg.svd(A)
    if(cutoff <= 0):
        cutoff =  s.max() * max(A.shape) * np.finfo(float).eps
    return len([x for x in s if abs(x) > cutoff])

#Note double brackets in the syntax of creating a matrix with either np.array or np.matrix.
B = np.array( [[1,2,0,0],
               [2,4,0,0],
               [0,0,0,0]])
C = np.matrix([[1,3,2,4],[2,5,6,8],[2,5,6,8.00001]])
#Note the third row. Changing the 8 to 8.00001 will change the rank from 2 to 3.
#This suggest using an absolute small cutoff is sometime problematic.

print(rank(B,0.0001))
print(rank(C,0.0000001))
print(matrix_rank(B,tol = 0.0001))
print(matrix_rank(C,tol = 0.00000001))
```

```
3
1
0
1
3
1
3
```