Introduction to Statistical Learning Toolbox

Dahua Lin

MMLab,

The Chinese University of Hong Kong

Outline

- What is sltoolbox?
  - A brief introduction
- Get started
  - A Tutorial on examples
- A step further
  - Some advanced usage
- Behind the scene
  - Discussions on the design and implementations

What is sltoolbox?

- A comprehensive set of matlab codes for solving the problems in
  - Statistical learning (core)
  - Pattern recognition
  - Computer vision

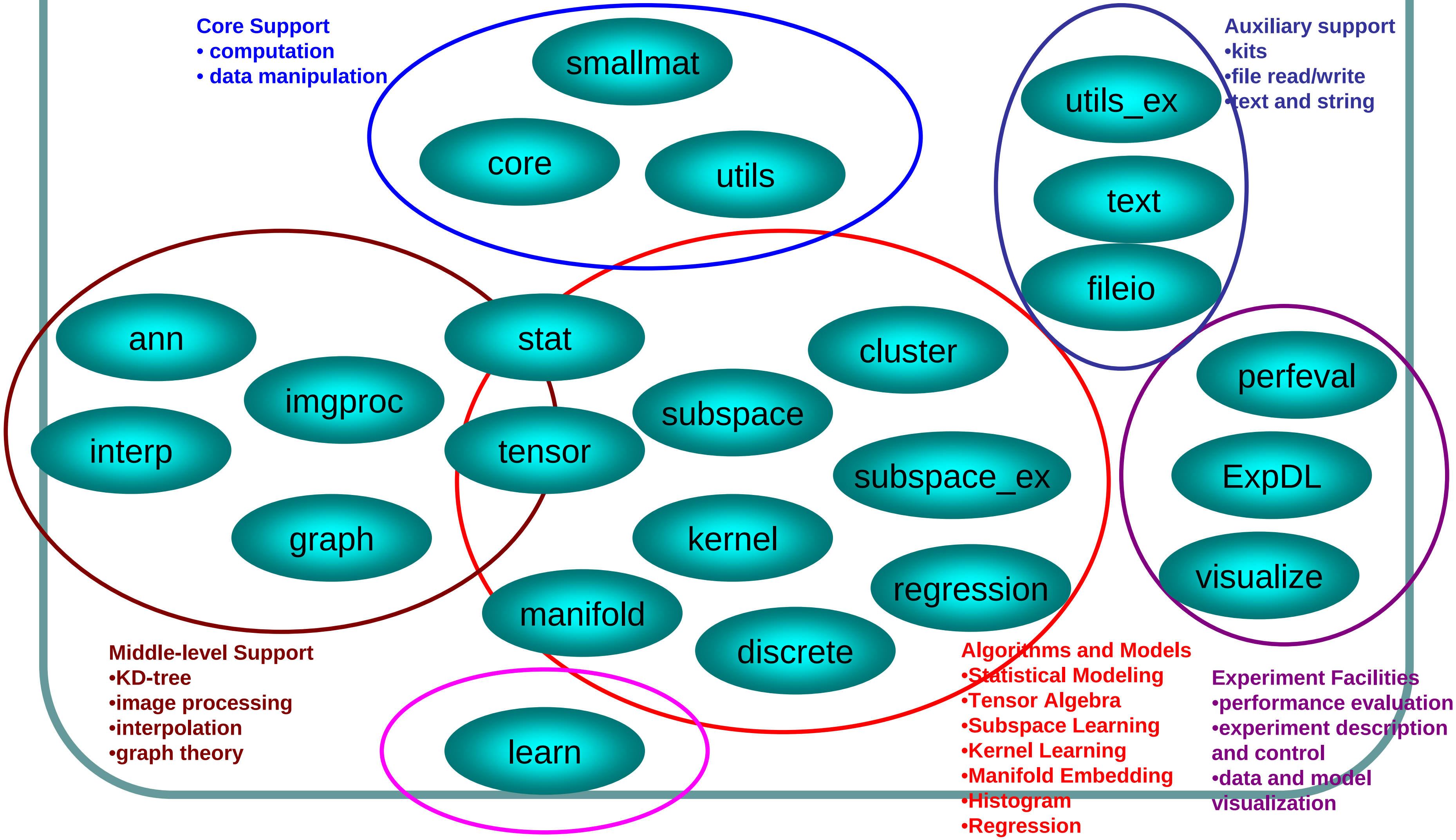
- Functions in different levels are offered
  - Routines for basic computation and data manipulation
  - Algorithms and Models
  - Framework (Architecture)

- Publicly available
  - Download page: (MATLAB Central Exchange: Mathematics.General) http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=12333&objectType=file
  - Manual Page: http://www.mydahua.net/sltdoc/index.html

Organization

- SItoolbox uses “category” to manage the codes.
  - The m-files are divided into different categories, and the files in the same category are grouped in a folder.
  - 256 m-files in 24 categories.

Map of the contents



Core Support
- computation
- data manipulation

Auxiliary support
- kits
- file read/write
- text and string

Middle-level Support
- KD-tree
- image processing
- interpolation
- graph theory

Learning architecture

Algorithms and Models
- Statistical Modeling
- Tensor Algebra
- Subspace Learning
- Kernel Learning
- Manifold Embedding
- Histogram
- Regression
- Clustering
- Spectral Learning

Experiment Facilities
- performance evaluation
- experiment description and control
- data and model visualization

Main Features

- Covers many active research topics
- Offers useful facilities and kits
- Highly optimized
  - Deduce equivalent math forms to reduce complexity
  - Group into matrix-based operations
  - Cpp-mex for some core codes
- Flexible and Extensible
  - Maximally tunable
  - mechanisms for customization
- Well organized
  - reusability and consistency
- Easy to use
  - Elaborate help in each m-file
  - Friendly and scalable function interface
- Robust
  - Numerically stable
  - Error-checking

How to install

- Download from Matlab Central Exchange
  - http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=12333&objectType=file
- Unzip the package to any folder
- Launch Matlab
  - switch to the root folder of sltoolbox
    - In matlab, the command “CD” can be used to switch folder
  - type the command: install_paths
    - it will add the sltoolbox directories to the search path of Matlab

Other issues

- cpp-mex
  - Some core codes are written by C++ for efficiency
  - The win32 and win64 versions of mex files for these codes are in the package
    - They are compiled with Matlab R2006a and Microsoft Visual Studio 2005
    - In normal cases, they are workable without any action.
  - The cpp sources are also shipped, if necessary, you can re-compile them.

- Third-party support
  - The slisomap (for ISOMAP) relies on Matlab BGL toolbox to compute the shortest path length.
  - The annsearch (KD-tree approximate Nearest neighbors) is based on a C++ library ANN version 1.1. However, with the mex files offered, you can run it without the existence of ANN. You need the ANN source codes when you need to recompile the annsearch_wrapper.cpp.
  - You are welcomed to use sltoolbox with all built-in or third-party toolboxes to accomplish your task. The sltoolbox can cooperate with other codes well.

Get Started

- sltoolbox is in itself easy to use.
- sltoolbox contains elaborate documentation.
  - You may use “help <function name>” to obtain the help on using a function.
- Read the introduction to get a basic idea on what the sltoolbox can give you.
  - introduction.txt has a brief list of all functions by categories.
- The following slides will take some examples to show you how to use Matlab.

Some conventions in using sltoolbox

- To differ from other toolboxes, most functions in sltoolbox use “sl” as prefix.
- Sample set representation
- Property list
- Grouping and labeling samples: labels, nums, bounds and partitions
- Function handle and functor

How to input samples

- In sltoolbox, the samples are stored in matrix by columns. (This is also the convention in literatures)
- In many algorithms, you can input weighted samples, If you have n samples, you can input the weights either of the two forms:
  - empty array []: means all weights are 1.
  - 1 x n row vector of all weights.
- If you input [ ] as weights, the function will select to use faster unweighted implementation. So when you do not need to assign different weights to different samples, just use [ ], which will be typically more efficient than ones(1, n).

To represent a sample set having 3 samples, the length of the sample vector is 2 (sample dimension is 2).
Suppose the samples are (0, 1), (3, 5) and (6, 9). Then you can represent them by a 2 x 3 matrix A as follows:

```
A =

   0   3   6
   1   5   9
```

If you want to assign different weights, such as 0.5, 0.3, and 0.2 to the samples, you can represent the weight by

```
w =

  0.5   0.3   0.2
```

slmean is a function that can accept weights.
m0 = slmean(A) or m0 = slmean(A, [ ]) computes normal mean of samples in A. m1 = slmean(A, w) computes weighted mean of the samples. They result in

```
m0 =     m1 =

   3         2.1
   5         3.8
```

Specify properties

- Many programs have a series of tunable parameters.
  - If list all parameters as normal input arguments, the function will be difficult to remember and use, and prone to error.
    - f(param1, param2, param3, param4, …)
  - In most cases, we only need to tune some of these parameters, and leave others in default values.
- In sltoolbox, the property list is used to solve the problem
  - property list is given in the end of the input arguments as
    - f(…, ‘name1’, value1, ‘name2’, value2, …)
  - You can specify none or some of the properties and leave others in default values.
  - You can use help to see the properties and their default values.

K-means is a well-known algorithm that you can use some parameters to control its process. By default, you can write

[means, labels] = slkmeans(X);

It will cluster the samples in X into 3 clusters.

Typically, you would like to control the number of cluters, say k, you can write

[means, labels] = slkmeans(X, ‘K’, k);

Furthermore, if you would like to set the maximum number of iterations to 100 and tell it not to display iteration information, you can write

[means, labels] = slkmeans(X, ‘K’, k, ‘maxiter’, 100, ‘verbose’, false);

For the parameters not specified, they will be in default values.

Group and label samples

- To represent samples in different classes, we have two ways
  - group the samples from the same class together and give the numbers of samples in the classes
    - In some functions, it is more convenient to convert nums to index bounds to represent the grouping.
  - Tell which samples are from which classes by assigning each sample a label
  - Functions are offered for conversion
    - slexpand
    - slcount
    - slnums2bounds

For example, we have 8 samples from 3 classes, and the samples from the same classes are grouped together. The first class has 2 samples, and the other two have 3 samples respectively.

```
nums =
   2   3   3
```

The index bounds are given by start indices and end indices.

```
si =
   1   3   6
ei =
   2   5   8
```

Then the labels of the samples are given by

```
labels =
   1   1   2   2   2   3   3   3
```

You can use the following functions for representation conversion:

```
[si, ei] = slnums2bounds(nums);
nums  = slcount(labels);
labels = slexpand(nums);
```

Function handle and functors

- It is a good way to increase the flexibility by allowing the user to define their own behavior.
- Matlab inherently offers three ways:
  - function name
  - inline object
  - function handle
- You can use **feval** to invoke a given function on some arguments.
- In sltoolbox, it offers one more option: **functor**. The aforementioned three types can all be considered as a functor, in addition, a parametric function can be given in the form as a cell array as
  - {f, param1, param2, …}
  - f can be either of a function name, inline object or function handle.
  - The sltoolbox offers **slevalfunctor** to evaluate a functor on arguments.

Use inline object

```
f = inline('x+y', 'x', 'y');
f(1, 2) => ans = 3
```

Use function handle

```
f = @floor
f(1.3) => ans = 1
```

Use anonymous function handle

```
f = @(x,y) x + y
f(1,2) => ans = 3
```

Use feval

```
feval('floor', 1.3) => ans = 1
feval(@floor, 1.3) => ans = 1
```

Use sltoolbox's functor and slevalfunctor

```
f = @(x,y,a) (x + y) * a
functor = {f, 2};
r = slevalfunctor(functor, 1, 2) => r = 6
```

Basic Example Categories List

- [core], [smallmat] and [utils]
  - efficient implementation of basic computational routines and data manipulation
- [fileio] and [text]
  - convenient kits to deal with files and text strings
- [imgproc] and [interp]
  - adapt the images to learning models
- [stat]
  - statistical computation and modeling
  - finite mixture model (such as GMM)
- [subspace] and [subspace_ex]
  - subspace learning algorithms
- [kernel]
  - kernel learning and kernelization
- [regression]
  - linear and logistic regression
- [tensor algebra]
  - tensor algebra for multilinear analysis
- [cluster]
  - K-Means Clustering
- [perfeval]
  - classification and verification performance evaluation
- [discrete]
  - build and compare histograms

Basic computation and manipulation routines

- add vectors to matrix
- sum up samples in different classes
- compute pairwise distances and their mean
- solve eigenvalue problem and generalized eigenvalue problem of symmetric matrix
- get Cartesian products
- partition an array into blocks
- process a large number of small matrices

add vectors to matrix

Data preparation

```
>> A = reshape(1:6, 2, 3)

A =

     1     3     5
     2     4     6

>> vc = [10; 20]

vc =

    10
    20

>> vr = [100 200 300]

vr =

   100   200   300
```

Add vectors to every columns / rows of matrix with the place dimension specified (1-columns, 2-rows)

```
>> A1 = sladdvec(A, vc, 1)

A1 =

    11    13    15
    22    24    26


>> A2 = sladdvec(A, vr, 2)

A2 =

   101   203   305
   102   204   306
```

less overhead

Add vectors to every columns / rows of matrix with the place dimension automatically determined

```
>> A1 = sladdvec(A, vc)

A1 =

    11    13    15
    22    24    26


>> A2 = sladdvec(A, vr)

A2 =

   101   203   305
   102   204   306
```

more friendly

Simultaneously add to rows and columns
B(i, j) = A(i, j) + vc(i) + vr(j)

```
>> B = sladdrowcols(A, vr, vc)

B =

   111   213   315
   122   224   326
```

Much faster than invoke sladdvec for twice.

See also

slmulvec multiple with vectors

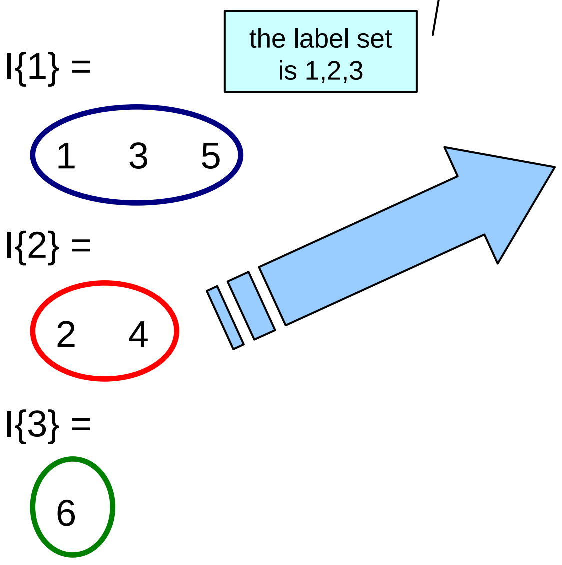
sladd, slmul
an extension: add sub-arrays to every sub-array of an array of any dimension

slmin, slmax, slsum
calculate minimum/maximum/sum for every sub array

slpwcalc
make a 2D result table by computing pairwisely on a row and a column.

Sum up samples in different classes

```
>> I = sllabelinds(labels, 1:3);
```



```
>> labels = [1 2 1 2 1 3]
```

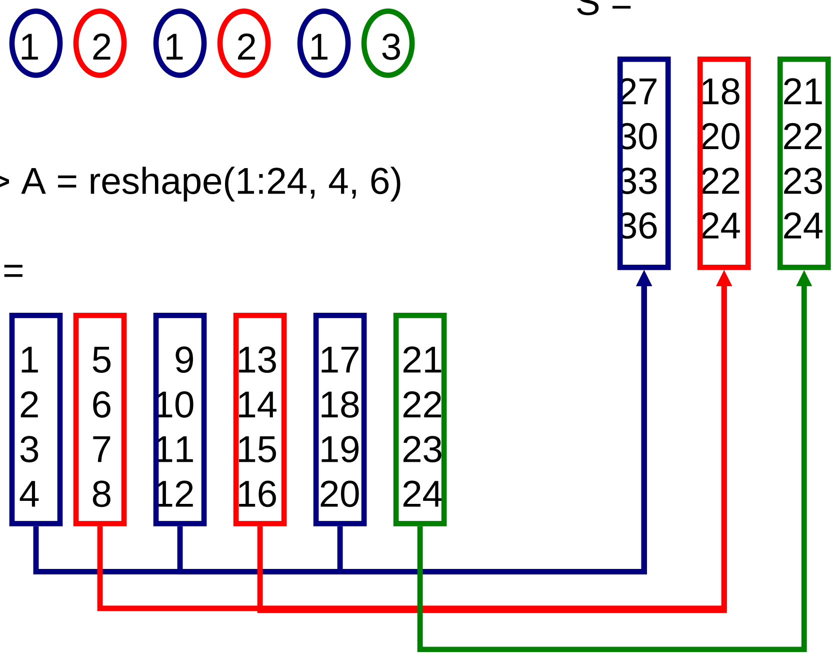
labels =

```
>> A = reshape(1:24, 4, 6)
```

A =

```
>> S = sllabeledsum(A, labels, 1:3)
```

S =



compute pairwise distances

```
>> X = rand(3, 2)

X =

    0.3529    0.1389
    0.8132    0.2028
    0.0099    0.1987

>> Y = rand(3, 3)

Y =

    0.6038    0.0153    0.9318
    0.2722    0.7468    0.4660
    0.1988    0.4451    0.4186
```

D(i, j) = d(X(:,i), Y(:,j))

```
>> D = slmetric_pw(X, Y, 'eucdist')

D =

    0.6256    0.5548    0.7892
    0.4701    0.6099    0.8639
```

Implement in a fast way by grouping the calculations into matrix computation.

See also

slmetric_cp

distances between corresponding pairs

sldiff_pw

pairwise differences

Support metric types:

- •Euclidean distance (L2)
- •Square of Euclidean distance
- •Dot product
- •Normalized Correlation
- •Angle between vectors
- •Quadratic form: x' Q y
- •Quadratic form of difference: (x-y)'* Q * (x-y)
- •City-block distance (L1)
- •Maximum absolute difference
- •Minimum absolute difference
- •Weighted square distance

Compute the mean of pairwise distances

- m = sldistmean(X, Y, ‘sqdist’);

Functionality

equivalent to

m = mean2(slmetric_pw(X, Y, ‘sqdist’));

Supported forms

square distances

weighted square distances

quadratic form of differences

Implementation

Based on an equivalent form

The complexity is reduced from O(n^2) to O(n * log(n))

Much faster

Solve (generalized) eigenvalue problem of symmetric matrix

solving eigenvalue problem of symmetric matrix

solve all eigenvalues and eigenvectors

[evals, evecs] = slsymeig(A)

solve the k largest eigenvalues and eigenvectors (sorted in descending order)
[evals, evecs] = slsymeig(A, k)

solve the k smallest eigenvalues and eigenvectors (sorted in ascending order)
[evals, evecs] = slsymeig(A, k, ‘ascend’)

solve the generalized eigenvalues and eigenvectors

[evals, evecs] = slsymgeig(A, B)

solve the generalized eigenvalues and eigenvectors with specified method to deal with the case when B is nearly singular
[evals, evecs] = slsymgeig(A, B, method, r)

Main features

Efficient: select efficient way specially for symmetric matrix and and for the cases when only a small number of eigenvectors are required.

Guarantee that the eigenvalues and eigenvectors are real (not complex)

Deal with the singular cases for generalized eigenvalue problem in a robust way.

The output is ordered.

Get Cartesian product

S1 = {'a', 'b', 'c'};

S2 = {1,2};

S3 = {‘X’, ‘Y’, ‘Z’}

C = slcartprod(S1, S2, S3)

size(C) = 3 x 3 x 2 x 3

#S3 #S2 #S1

The tuple length:
the number of
involved sets

The first set changes slow
The last set changes fast

```
C(:,:,1,1) =
    'a'    'a'    'a'
    [1]    [1]    [1]
    'X'    'Y'    'Z'

C(:,:,2,1) =
    'a'    'a'    'a'
    [2]    [2]    [2]
    'X'    'Y'    'Z'

C(:,:,1,2) =
    'b'    'b'    'b'
    [1]    [1]    [1]
    'X'    'Y'    'Z'

C(:,:,2,2) =
    'b'    'b'    'b'
    [2]    [2]    [2]
    'X'    'Y'    'Z'

C(:,:,1,3) =
    'c'    'c'    'c'
    [1]    [1]    [1]
    'X'    'Y'    'Z'

C(:,:,2,3) =
    'c'    'c'    'c'
    [2]    [2]    [2]
    'X'    'Y'    'Z'
```

partition an array into blocks

```
>> siz = size(A)

siz =

     4     6

>> PS = slpartition(siz, 'blksizes', [2 2], [2 3 1])

PS =

2x1 struct array with fields:
    sinds
    einds

>> PS(1)

ans =

   sinds: [1 3]
   einds: [2 4]

>> PS(2)

ans =

   sinds: [1 3 6]
   einds: [2 5 6]
```

other specification types on partitioning

-The number of blocks

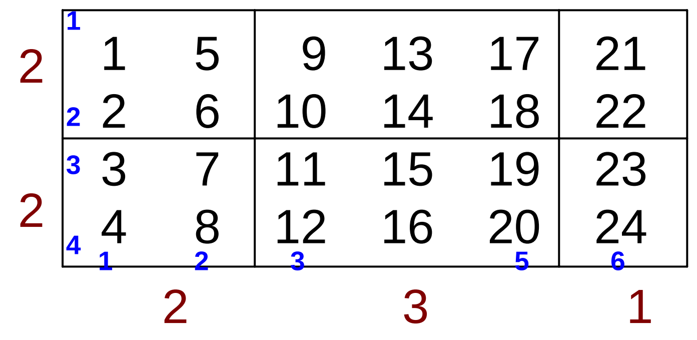
-Maximum block size

-The sizes of all blocks

-The bounds of indices

The index bounds

A =



process a large number of small matrices

```
>> A = reshape(1:12, 2, 2, 3)

A(:,:,1) =

     1     3          trace = 5
     2     4


A(:,:,2) =

     5     7          trace = 13
     6     8


A(:,:,3) =

     9    11          trace = 21
    10    12

>> sltrace2x2(A)

ans =

     5
    13
    21
```

The family of functions on batch of small matrices

sldet2x2:	compute determinant
slinv2x2:	compute inverse
sltrace2x2:	compute trace
slrotmat:	compute rotation matrices from angle

They are not to process the matrices one by one, but to process all matrices in a vectorized manner using the deduced formulas.

Kits for files and text

- Read and write single array efficiently
- Read and write text
- Parse and process filename
- Process strings

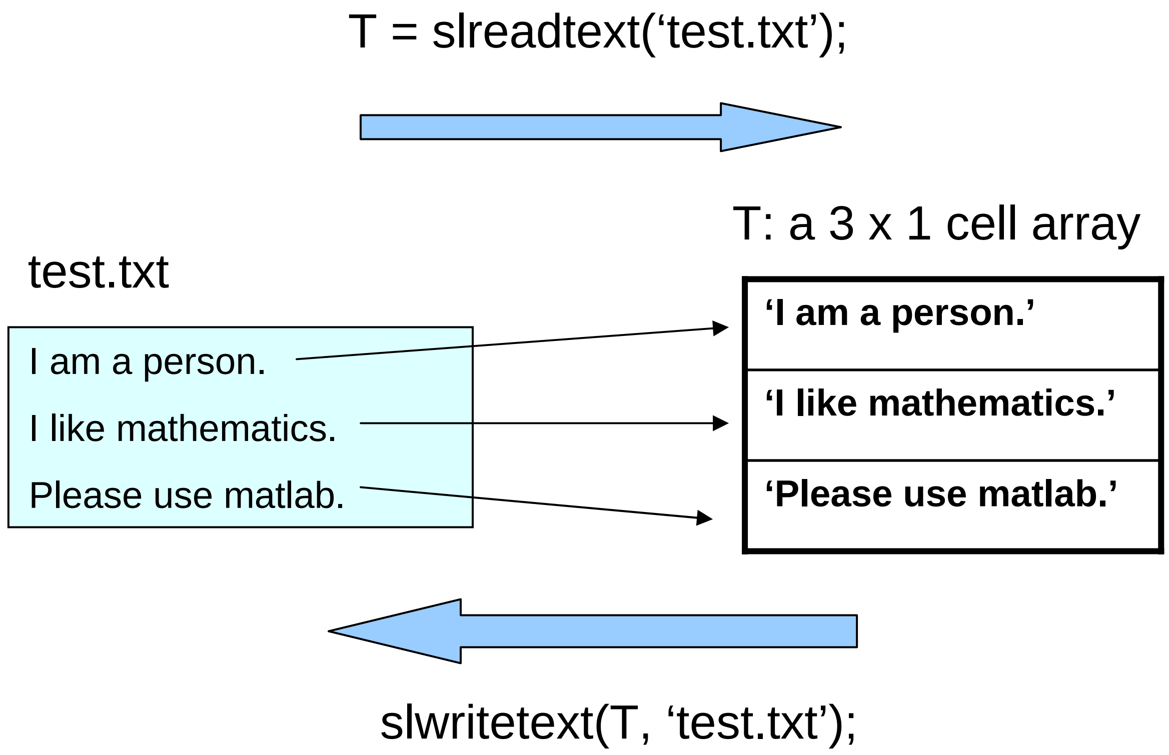
Read and write single array efficiently

- Store one array in a file
  - format specification: fileio/array_fileformat.txt
  - More efficient to load and save without processing much overhead information
  - Don’t need to set a variable name as in .mat file.

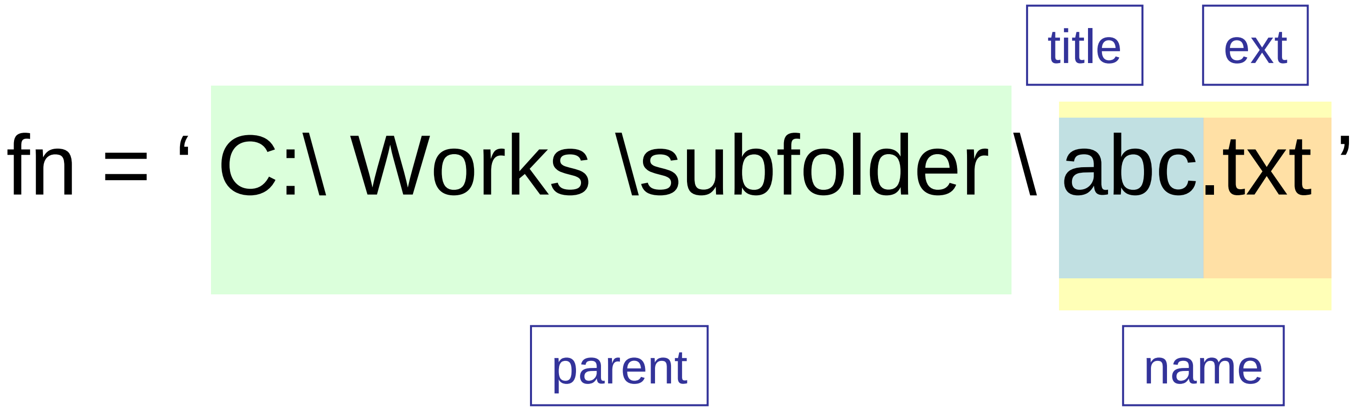
- Functions
  - slreadarray
  - slwritearray

- Syntax
  - Read from file
    - A = slreadarray(filename);
  - Write to file
    - slwritearray(A, filename);

Read and write text



Parse and process filenames



Parse

```
slfilepart(fn, ‘parent’) => ‘C:\Works\subfolder’
slfilepart(fn, ‘name’)  => ‘abc.txt’;
slfilepart(fn, ‘title’)     => ‘abc’
slfilepart(fn, ‘ext’)      => ‘.txt’
```

Change

```
slchangefilepart(fn, 'ext', '.bmp')
=> ‘C:\Works\subfolder\abc.bmp’
slchangefilepart(fn, ‘parent', ‘D:\MyWorks')
=> ‘D:\MyWorks\abc.bmp’
```

Batch Add Paths

```
>> sladdpath({'a01.jpg'; 'a02.jpg';
'a03.jpg'}, 'C:\imgs')

ans =

    'C:\imgs\a01.jpg'
    'C:\imgs\a02.jpg'
    'C:\imgs\a03.jpg'
```

Process strings

Parse assignment statement

[name, value] = slparse_assignment('a=1')

-name = 'a'

-value = 1

[name, value] = slparse_assignment('a=student')

-name = 'a'

-value ='student'

[name, value] = slparse_assignment('a="1"')

-name = 'a'

-value = '1'

Split a string

```
>> strs  = slstrsplit('abc efg', ' ')

strs =

    'abc'
    'efg'

>> strs  = slstrsplit('abc+efg-cd', '+-')

strs =

    'abc'
    'efg'
    'cd'
```

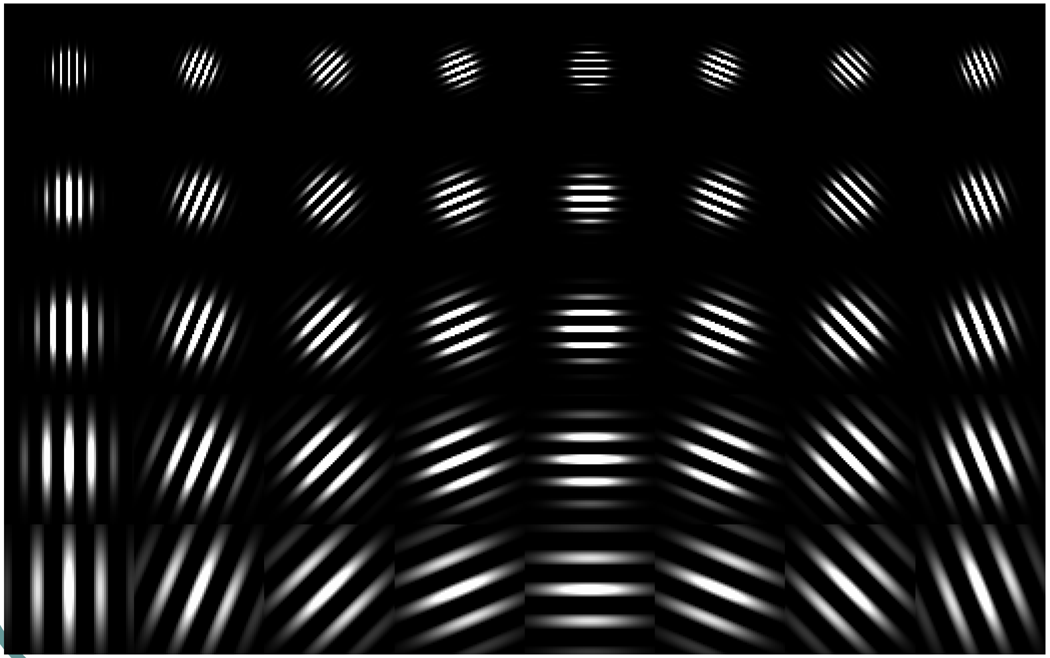
Image Processing and Interpolation

- Batch Filtering
- Pixel Value Normalization
- Pixel Neighbor Extraction
- Image Interpolation

Batch Filtering

FB = slgaborbands(33, 0:4, 0:7)
generates typical gabor filter bands in 5 scales, 8 orientations
FB is an array of size 33 x 33 x 5 x 8

The real part of the produced filter kernels



Apply the 40 filters to a set of images

slapplyfilterband(imgs, FB, 33);

If you would like to produce in vectorized form (more convenient for learning algorithms)

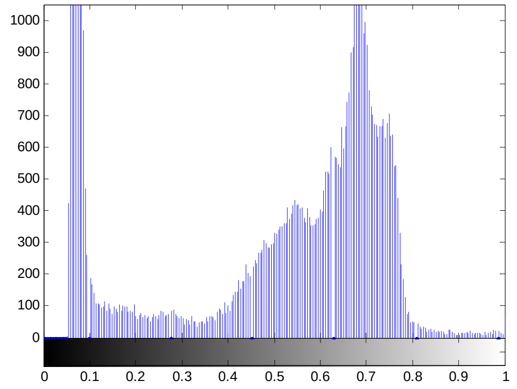
slapplyfilterband(imgs, FB, 33, ‘fbform’, ‘vec’);

Pixel Value Normalization

Normalize the pixel values to
mean = 0, var = 1

dstimg = slpixlinnorm(srcimg);

Hist of (0.25 * dstimg + 0.5)

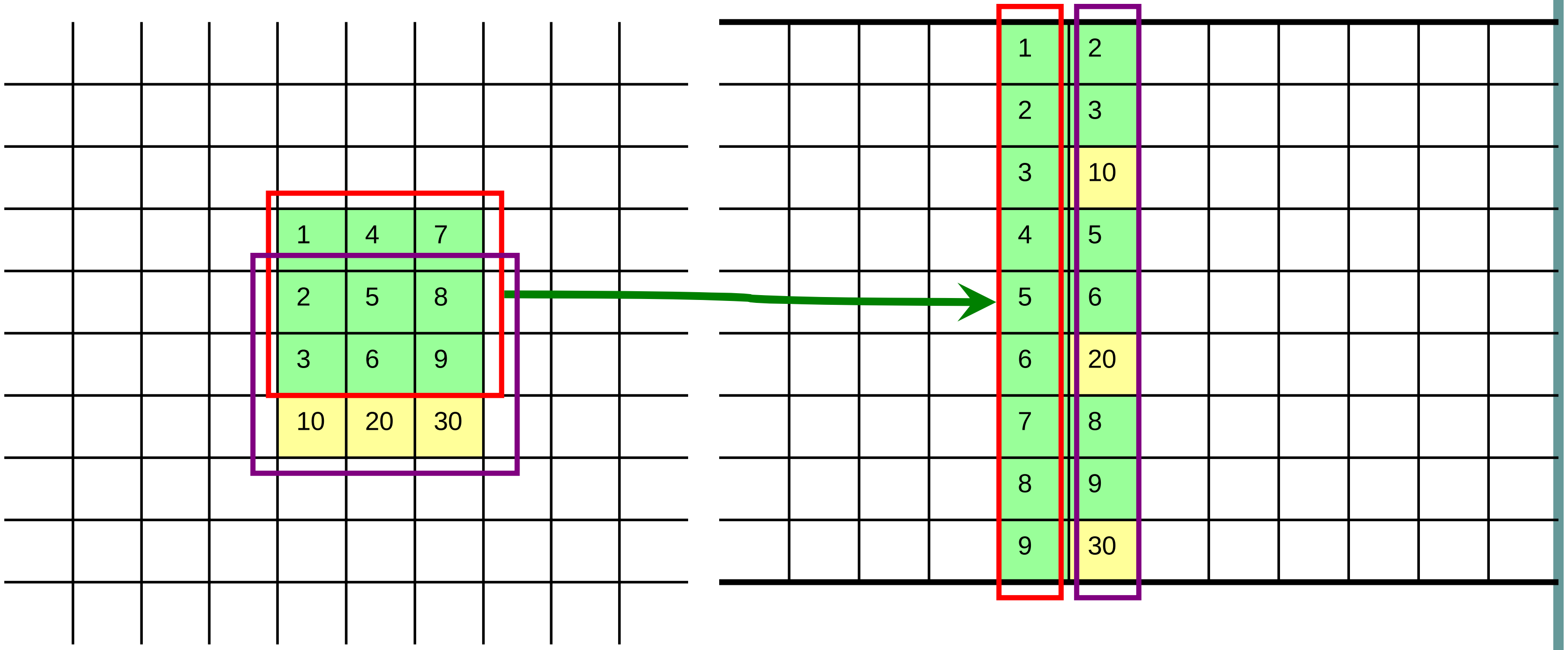


This is typically taken before vectorizing the images, and often achieves better performance than histogram equalization in learning algorithms.

Pixel Neighbor Extraction

R = slpixneighbors(img, [3, 3]);

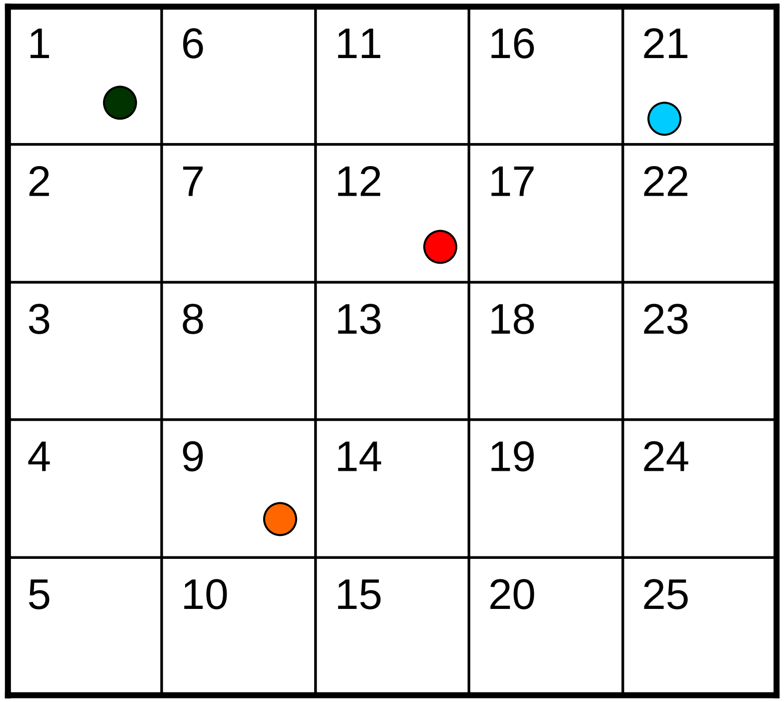
You can specify ROI, mask and sample step to control the extraction. It also supports multi-channel images.



h x w image

(3x3) x (hxw) sample matrix

Image Interpolation



```
>> A = reshape(1:25, 5, 5)

A =

     1     6    11    16    21
     2     7    12    17    22
     3     8    13    18    23
     4     9    14    19    24
     5    10    15    20    25

>> x = [1.2, 2.3, 3.4, 4.8];
>> y = [1.1, 4.3, 2.2, 1.4];

>> slimginterp(A, y, x)

ans =

   2.1000   10.8000   14.2000   20.4000
     ●         ●         ●         ●
```

You can select an interpolation scheme by specify the interpolation kernel.

V = slimginterp(A, I, J, interpker)

Three built in kernels are implemented: nearest, (bi)linear, (bi)cubic. By default, it uses linear kernel. You can also define your own kernel by function handle.

Statistical Modeling and Finite Mixture Models

- Basic Statistical Computation
  - mean, covariance and inverse of cov
  - pool and regularize covariance
  - whiten transform
- Gaussian models
  - model specification
  - random sampling and model estimation
  - probability density function
- Gaussian Mixture Models

Basic Statistical Computation

- Compute means
  - typical mean vector: **vmean = slmean(M);**
  - weighted mean: **vmean = slmean(M, w);**
  - means of grouped samples: **means = slmeans(M, [ ], nums);**
  - means of weighted grouped samples: **means = slmeans(M, w, nums);**

- Compute covariance
  - typical covariance: **C = slcov(X);**
  - covariance of weighted samples: **C = slcov(X, w);**
  - covariance with pre-computed mean: **C = slcov(X, w, vmean);**
  - covarfiance on centralized samples: **C = slcov(X, w, 0);**

- Compute inverse of covariance matrix: (C^{-1}) (can deal with singularity)
  - pseudo inverse: **R = slinvcov(C);**
  - regularized inverse: **R = slinvcov(C, 'reg', 1e-6);**
  - bounding inverse: **R = slinvcov(C, 'reg', 1e-6);**

- Pool covariance
  - **slpoolcov(Cs)** or **slpoolcov(Cs, w);** Cs is an d x d x n array of n classes

- Regularize covariance
  - **Cs_reg = slregcov(Cs, 'lambda', 0.1, 'gamma', 0.2);**

Gaussian models

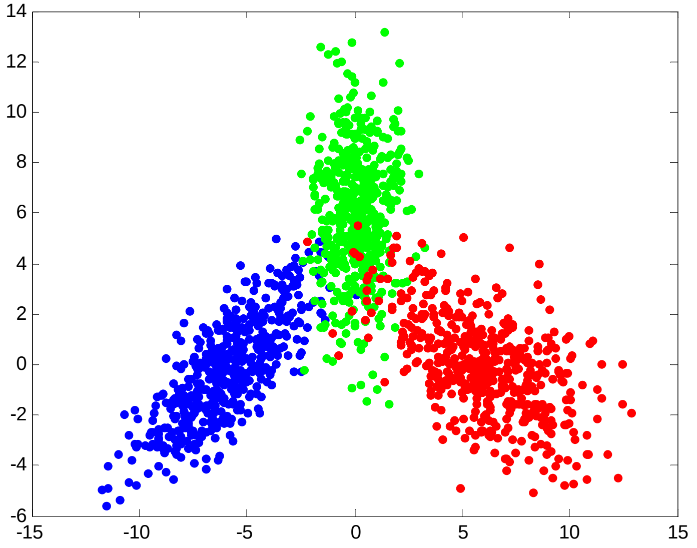
To specify a model with 3 means: (-6, 0), (0, 6), (6, 0), and covariance being [5 4; 4 5], [1 0; 0 6], [6 -3; -3 4]; you can write

```
>> Mm = [-6 0; 0 6; 6 0]';

>> Cs = cat(3, [5 4; 4 5], [1 0; 0 6], [6 -3, -3 4]);

>> GS = slgausscomb('means', Mm, 'covs', Cs)

GS =

        dim: 2
    nmodels: 3
      means: [2x3 double]
       covs: [2x2x3 double]
    invcovs: [2x2x3 double]
```

Please refer to stat/gauss_struct.txt for detailed information of gaussian model structure.

Generate random samples (500 for each model)

```
>> nums = [500 500 500];
>> X = slgaussrnd(GS, nums);
```



specification types

all models share a isometric variance:	**slgausscomb('means', Mm, 'vars', s);**
all models use different isometric variance:	**slgausscomb('means', Mm, 'vars', [s1, s2, …]);**
all models share a diagonal variances:	**slgausscomb('means', Mm, 'vars', d);**
all models use different diagonal variances:	**slgausscomb('means', Mm, 'vars, [d1, d2, …]);**
all models share a covariance matrix:	**slgausscomb('means', Mm, 'covs', C);**
all models use different covariance matrices:	**slgausscomb('means', Mm, 'covs', Cs);**

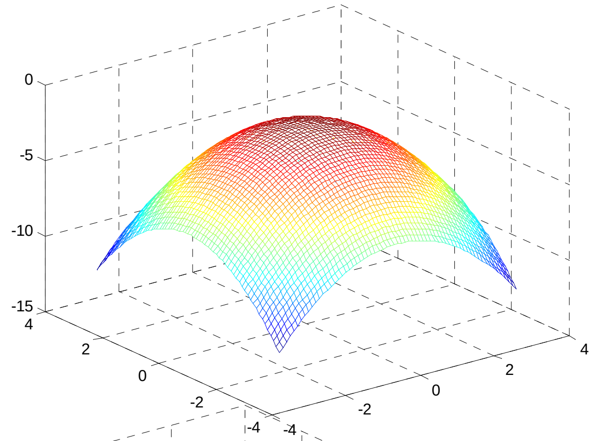
Gaussian model pdf

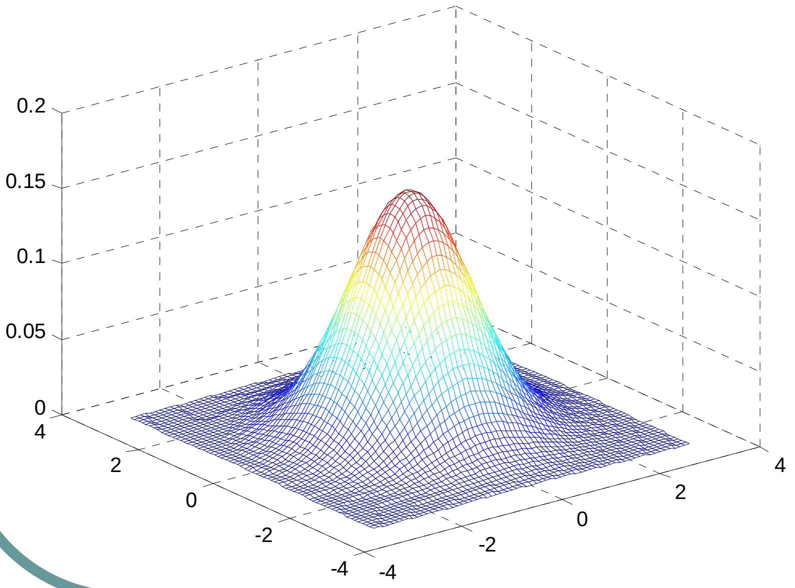
- probability density function (pdf)

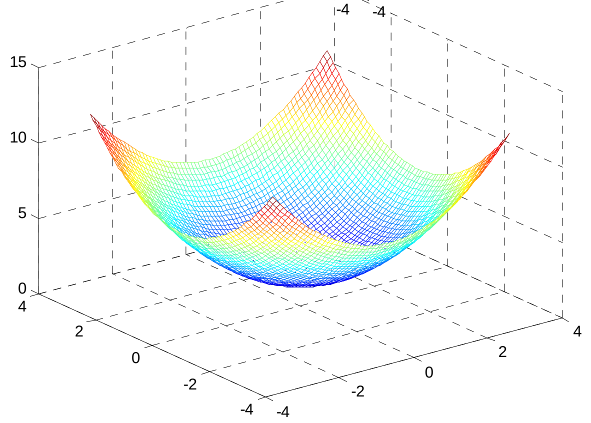
$$p(\mathbf{x};\boldsymbol{\mu},\mathbf{C}) = \frac{1}{\sqrt{(2\pi)^d\,|\mathbf{C}|}}\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\mathbf{C}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$$

$$\log p = -\frac{1}{2}\left(d\log(2\pi)+\log|\mathbf{C}|+(\mathbf{x}-\boldsymbol{\mu})^T\mathbf{C}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$$

slgausspdf(G, X, ‘output’, ‘log’); => log p





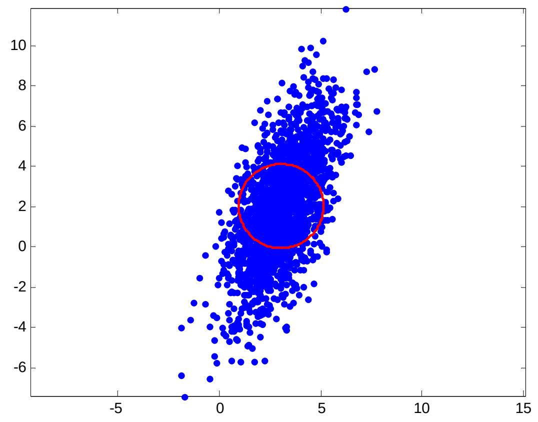


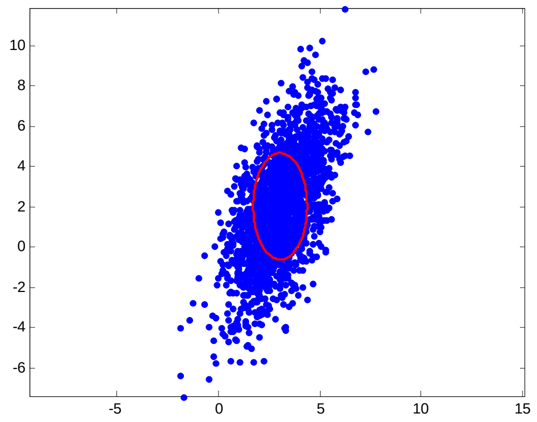
slgausspdf(G, X); => p

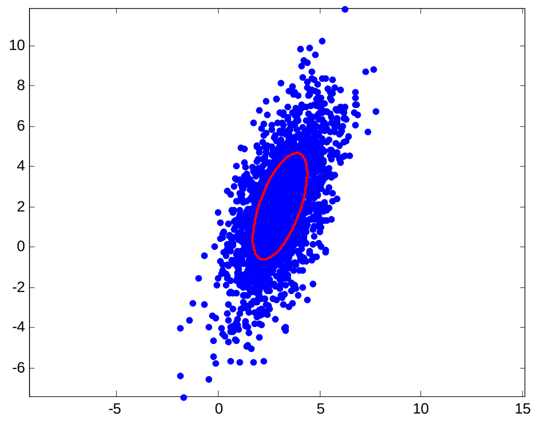
slgausspdf(G, X, ‘output’, ‘neglog’); => -log p

Gaussian model estimation

- Estimate single model from data







isotropic variance
g1 = slgaussest(X, 'varform', 'univar');

diagonal variance
g2 = slgaussest(X, 'varform', 'diagvar');

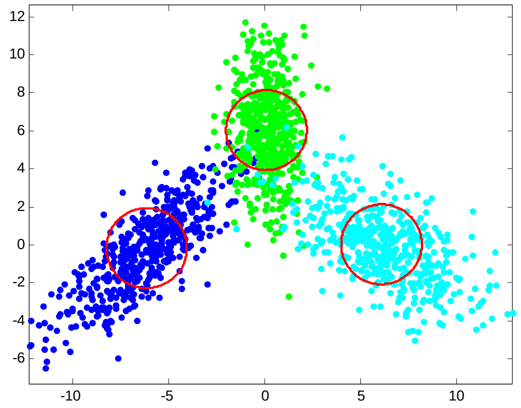
general covariance
g3 = slgaussest(X, 'varform', 'covar');

Support high-dimensional data.
You can specify more properties, such as sample weights, whether to compute the inverse of the covariances and so on.

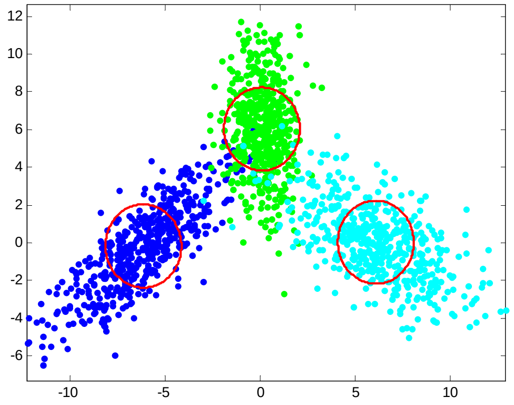
Gaussian model estimation

- Estimate multiple models from grouped samples (each from one group)

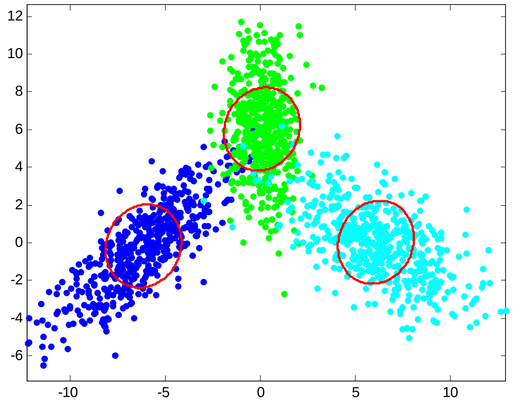
g = slgaussest(X, 'nums', nums, 'varform', 'univar', 'sharevar', true);

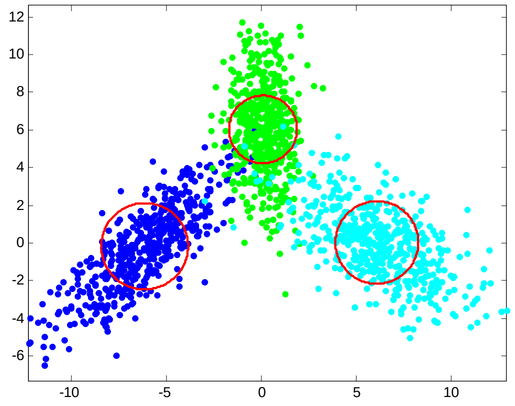


g = slgaussest(X, 'nums', nums, 'varform', ‘diagvar', 'sharevar', true);

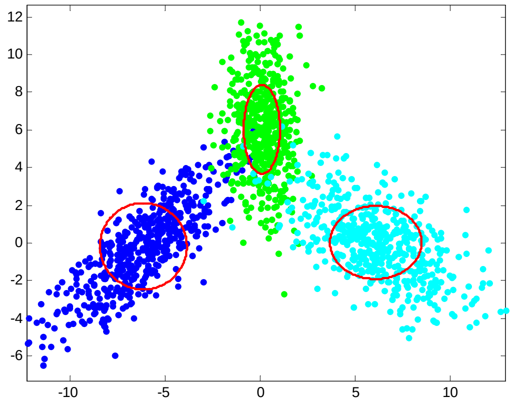


g1 = slgaussest(X, 'nums', nums, 'varform', ‘covar', 'sharevar', true);

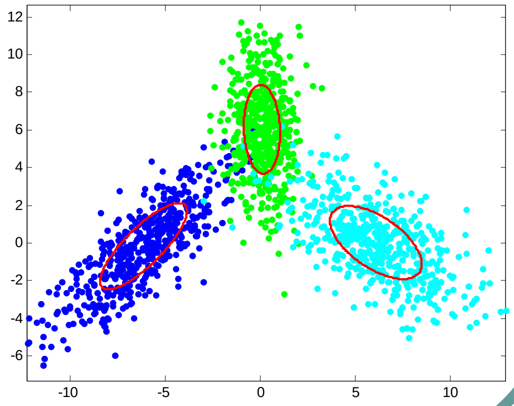




g = slgaussest(X, 'nums', nums, 'varform', 'univar', 'sharevar', false);



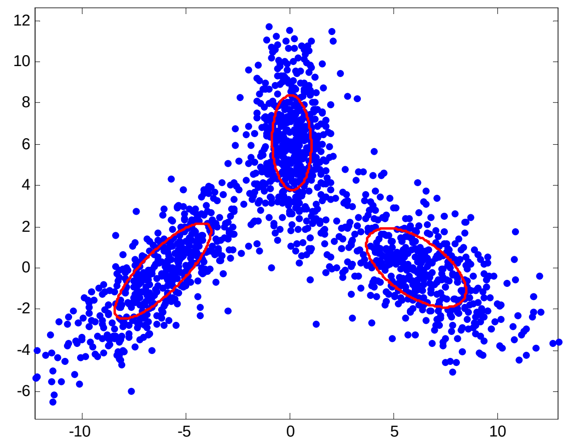
g = slgaussest(X, 'nums', nums, 'varform', ‘diagvar', 'sharevar', false);

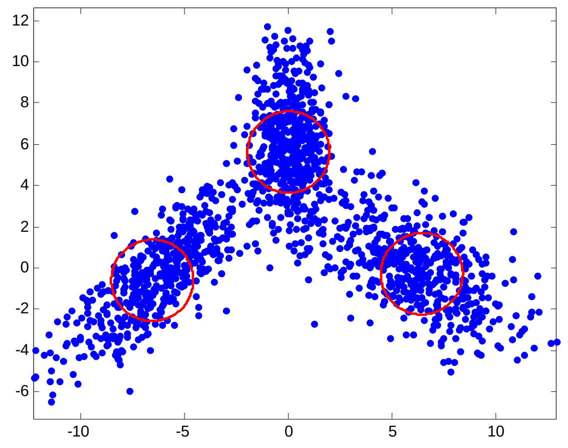


g1 = slgaussest(X, 'nums', nums, 'varform', ‘covar', 'sharevar', false);

Estimate Gaussian Mixture Model

- Estimate the following model $p(\mathbf{x}) = \sum_{k=1}^{m} \pi_k \, p(\mathbf{x}; \boldsymbol{\mu}, \mathbf{C})$
  - Use EM-based method
  - Have full control on the form of variance and whether the variance is shared by all component models



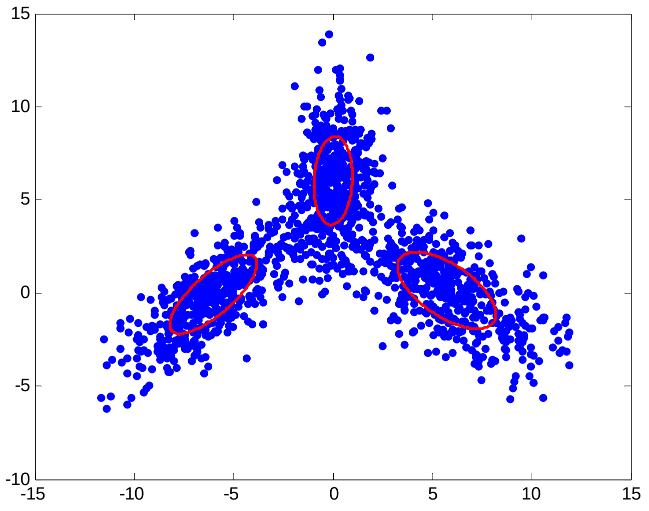


g = slgmm(X, 'K', 3);
be default, varform = ‘covar’, and sharevar = false

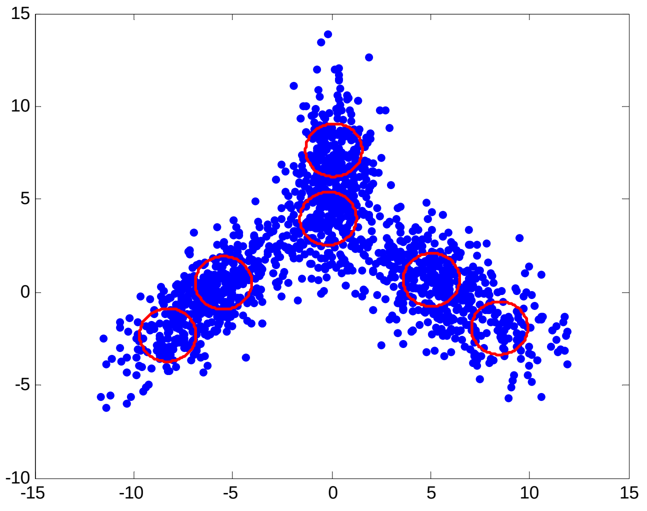
g = slgmm(X, 'K', 3, ‘varform’, ‘univar’, ‘sharevar’, true);

GMM with Component Annealing

- Sometimes it is difficult to determine K in advance, a better choice is to perform dynamic model selection during the learning process.
- A typical way is called component annealing. The basic idea is to set a relatively large K initially and then discard the trivial components during learning. It can be proved that this procedure actually optimizes a MDL-like criteria in a greedy manner.
- If slgmm, you can set a postive annthres in the property list. Then in run-time, the components with weights lower than annthres * (1/K) will be automatically discarded.



g = slgmm(X, 'K', 10, 'annthres', 0.5, 'maxiter', 300)



g = slgmm(X, 'K', 10, 'varform', 'univar', 'sharevar', true, 'annthres', 0.5, 'maxiter', 300);

using different variance forms, the suitable numbers of components are different.

Subspace Learning

- The sltoolbox offers the following representative algorithms and their variants
  - Principal Component Analysis (PCA)
  - Linear Discriminant Analysis (LDA)
  - 2D PCA
  - Partition-based PCA
  - Coupled PCA

Principal Component Analysis (PCA)

S=slpca(X);

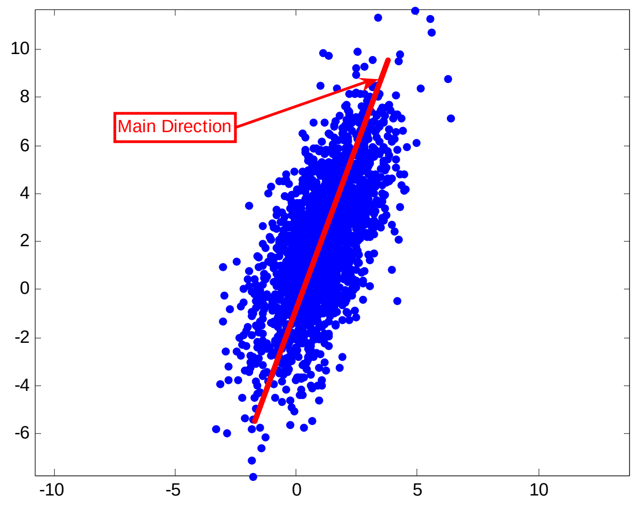
S is a struct of PCA model

sampledim:	sample dimension
feadim:	feature dimension
support:	the number of features in training
vmean:	the mean vector
P:	the principal basis
eigvals:	the eigen-spectrum
residue:	the discarded energy
energyratio:	the ratio of preserved energy

You can control the training by specifying more properties:

method:	the method used in training, auto: auto selection std: standard method svd: use SVD trans: based on transposed matrix
preserve:	the scheme to decide how many components are preserved
weights:	support weighted samples

Illustration of the main direction found by PCA on a set of 2D data



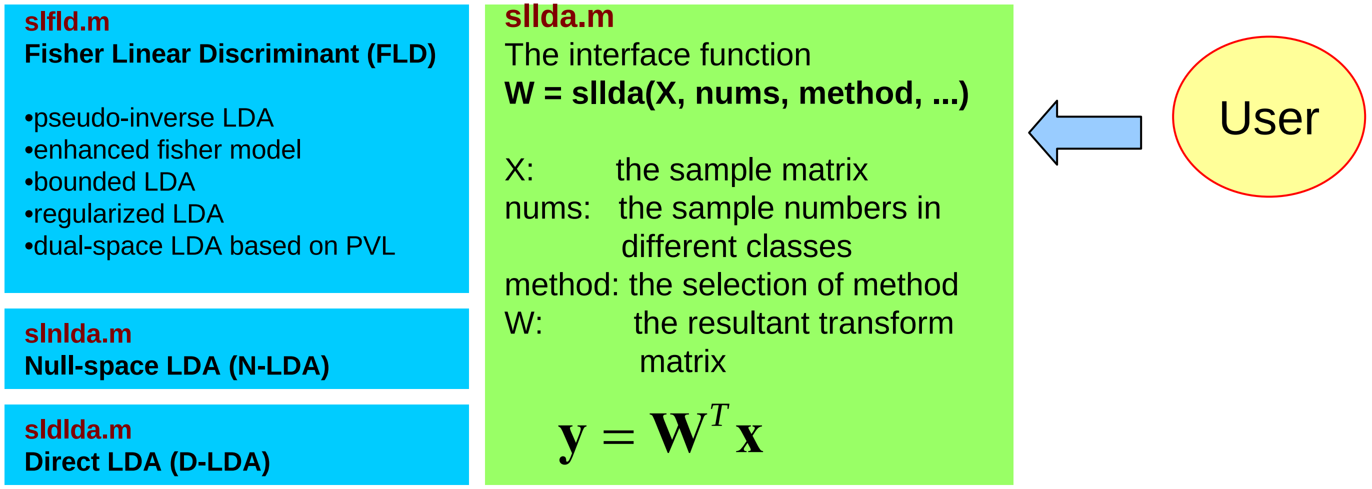
Feature extraction
Y = slapplypca(S, X);

Sample reconstruction
Xr = slpcarecon(S, Y);

Model truncation
Sr = slpcareduce(S, …);

Linear Discriminant Analysis

- A series of LDA-based algorithms are implemented. The usage is quite flexible.



slfld.m
Fisher Linear Discriminant (FLD)

•pseudo-inverse LDA
•enhanced fisher model
•bounded LDA
•regularized LDA
•dual-space LDA based on PVL

slnlda.m
Null-space LDA (N-LDA)

sldlda.m
Direct LDA (D-LDA)

sllda.m
The interface function
W = sllda(X, nums, method, ...)

X:	the sample matrix
nums:	the sample numbers in different classes
method:	the selection of method
W:	the resultant transform matrix

$$\mathbf{y} = \mathbf{W}^T \mathbf{x}$$

The user only need to know how to use sllda.m, the other three functions are internal implementations.

Linear Discriminant Analysis

- The properties of sllda
  - prepca: whether to perform a preceding PCA
  - rvalue: the ratio below which the eigenvalues are not considered as principal ones of Sw
  - dimset: the scheme to determine the output feature dimension
  - weights: the sample weights
  - Sw, Sb: the pre-computed scatter matrices or the scheme to compute them.
  - etc. For detailed info, please type “help sllda”.
- In the training sample matrix X, the samples from the same classes should be grouped together, and you should use nums to tell how many samples are from each class.

- Data preparation: 1000 400-dimensional samples from 200 classes, each one has 5 samples. Then
  - X should be a matrix of size 400 x 1000
  - The samples from the same classes should be grouped together
  - nums should be 5 * ones(1, 200)
- Train LDA transform using regularized LDA method and default settings
  - W = sllda(X, nums, ‘regdual’);
- Train LDA using dual-space method on weighted samples, and set the threshold at which the principal space and null space are delimited to be 1e-3 * maximum eigenvalue of Sw.
  - W = sllda(X, nums, ‘pvldual’, ‘rvalue’, 1e-3, ‘weights’, w);
  - w should be a 1 x n row vector.
- Train LDA using pseudo-inverse method and a preceding PCA reduction.
  - W = sllda(X, nums, ‘pinv’, ‘prepca’, true);
- Train LDA using null-space method, and set the dimensions with eigenvalues less than 1e-7 * max eigenvalue to be null space dimensions.
  - W = sllda(X, nums, ‘nlda’, ‘pdimset’, 1e-7);
- Train LDA with pre-computed scatter matrices using enhanced fisher model’s solver
  - W = sllda([ ], [ ], ‘ efm’, ‘Sb’, Sb, ‘Sw’, Sw);

Special construction of scatter matrices

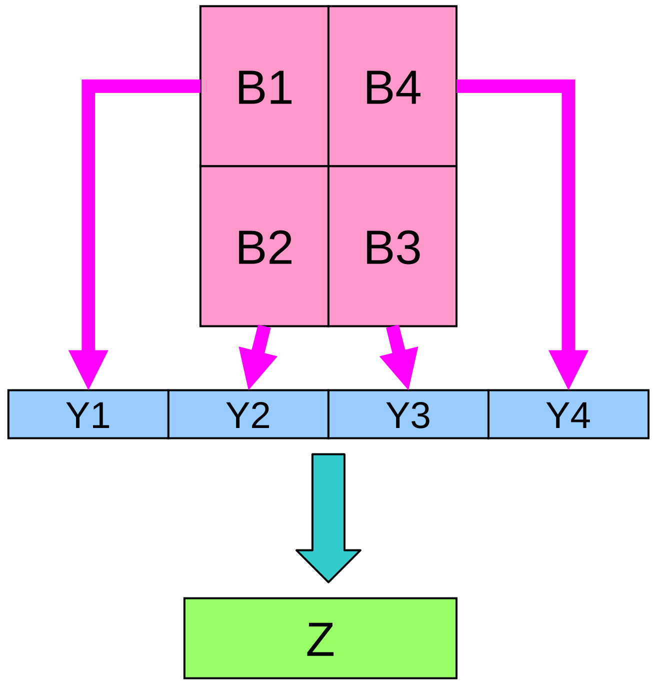
- By default, sllda will compute the scatter matrices internally using standard formulation.
- However, if you have your special formulations on scatter matrices, you can use them.
- LDA functions in sltoolbox use slscatter to compute scatter matrices. Please type help to see how to use slscatter.
- There are three ways
  - encapsulate the parameters to slscatter in a cell array and input to sllda
  - compute Sw and Sb using your slscatter and then input Sw and Sb to sllda
  - compute Sw and Sb using your functions and then input Sw and Sb to sllda to solve the transform.

Two-Dimensional PCA (2D PCA)

- 2D PCA is mainly to solve the problem of reducing the dimension of large images.
- It is formulated as $\mathbf{Y} = \mathbf{P}_L^T(\mathbf{X} - \mathbf{M})\mathbf{P}_R$
  - If input image size is h x w, and target feature size is fh x fw, then PL is of size h x fh, and PR is of size w x fw
  - PL is to reduce the row space, while PR is to reduce the column space
  - The two reduction processes are coupled during learning.
- Two learning methods are implemented in **sl2dpcaex.m**
  - ‘dimfix’: the feature size is decided before learning and fixed during learning
  - ‘grareduce’: the feature size is gradually reduced during learning by analyzing the eigen-spectrum.
- The training function supports weighted samples.
- You can use **sl2dpca_apply** and **sl2dpca_construct** for feature extraction and image reconstruction respectively.

Partition-based PCA

- It divides the arrays into small partitions, and apply PCA to the partitions respectively. After that a higher-level PCA is further applied to the principal components from all partitions.



Suppose, X is 100 x 100 x n array of n images of size 100 x 100. Considering the memory limit, you would like to divide them into blocks with edge length no more than 50, and then apply partition-based PCA. You can write

```
% generate the partition structure
ps = slpartition([100, 100], 'maxblksize', [50, 50]);
% train partition PCA based on ps
slpartitionpca(X, [100, 100], n, ps,
'D:\models\parpca.mat');
% the resulting model will be saved to parpca.mat
```

please “help slpartitionpca” for detailed information of the model file and the properties that can be specified to control the training process.

```
% In test stage, you can load the model by
M = load('parpca.mat');
% For a new set of images X2, you can obtain the
features by
Y = slpartitionpca_apply(M, 'D:\models', X2, n2);
```

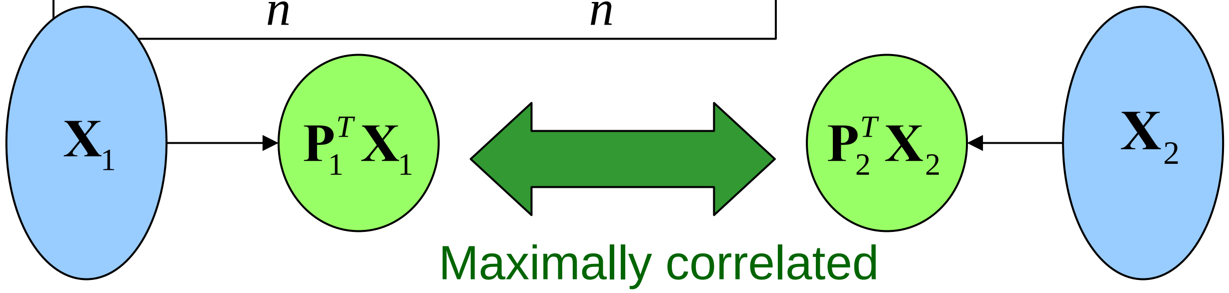
Coupled PCA

- Coupled PCA is pursue two subspaces on two sample sets that are maximally correlated with each other. It is useful to discover the connection between two sample sets.
- It is formulated as

$$\mathbf{P}_1, \mathbf{P}_2 = \arg\max_{\mathbf{P}_1, \mathbf{P}_2} \mathrm{tr}(\mathbf{P}_1^T \mathbf{C}_{12} \mathbf{P}_2 \mathbf{P}_2^T \mathbf{C}_{21} \mathbf{P}_1)$$
$$= \arg\max_{\mathbf{P}_1, \mathbf{P}_2} \mathrm{tr}(\mathbf{P}_2^T \mathbf{C}_{21} \mathbf{P}_1 \mathbf{P}_1^T \mathbf{C}_{12} \mathbf{P}_2)$$
$$\text{s.t. } \mathbf{P}_1^T \mathbf{P}_1 = \mathbf{I}, \quad \mathbf{P}_2^T \mathbf{P}_2 = \mathbf{I}$$
$$\mathbf{C}_{12} = \frac{1}{n} \mathbf{X}_1 \mathbf{X}_2^T \qquad \mathbf{C}_2 = \frac{1}{n} \mathbf{X}_2 \mathbf{X}_1^T$$

sltoolbox usage
[P1, P2] = slcopca(X1, X2, d)

•d is the target dimension
•X1 and X2 can be of different dimensions



Maximally correlated

Kernel Learning

- Kernel Computation and Centralization
- Kernelized Algorithms
  - KPCA
  - GDA and KDA

Kernel Computation and Centralization

- Kernel is to exploit the nonlinearity of samples by mapping them to a higher dimensional Hilbert space.

$$\varphi : \mathbf{x}_1 \rightarrow \varphi(\mathbf{x}_1) \in H$$
$$\varphi : \mathbf{x}_2 \rightarrow \varphi(\mathbf{x}_2) \in H$$
$$k(\mathbf{x}_1, \mathbf{x}_2) = \left\langle \varphi(\mathbf{x}_1), \varphi(\mathbf{x}_2) \right\rangle$$

- Kernel matrix

$$X_1 : n_1 \text{ samples, } \mathbf{x}_{11}, \mathbf{x}_{12}, \cdots, \mathbf{x}_{1n_1}$$
$$X_2 : n_2 \text{ samples, } \mathbf{x}_{21}, \mathbf{x}_{22}, \cdots, \mathbf{x}_{2n_2}$$
$$K(X_1, X_2) \Rightarrow \mathbf{K}(i, j) = k(\mathbf{x}_{1i}, \mathbf{x}_{2j})$$

- Kernel Centralization

$$\mathbf{K} \rightarrow \mathbf{K}_c$$
$$\mathbf{K}(i, j) = < \varphi_{1i}, \varphi_{2j} >$$
$$\mathbf{K}_c(i, j) = < \varphi_{1i} - \overline{\varphi}_1, \varphi_{2i} - \overline{\varphi}_2 >$$

- Built-in Supported kernels
  - linear kernel
  - polynomial kernel
  - gauss kernel
  - sigmoid kernel
  - inverse quadratic kernel

- Compute gram matrix of using gauss kernel with sigma = 2
  - K = slkernel(X, ‘gauss’, 2)
- Compute kernel matrix between X1 and X2 using polynomial kernel with a = 1, and degree = 2
  - K = slkernel(X1, X2, ‘poly’, 2, 1);
- Centralize gram matrix K
  - Kc = slcenkernel(K);
- Centralize kernel matrix on X2 w.r.t X0, named K, and the gram matrix of X0, named K0.
  - Kc = slcenkernel(K0, K);
- kernel centralization support weighted samples.

Kernelized Algorithms

- A set of representative kernelized subspace learning algorithms are implemented
  - Kernel PCA (KPCA): **slkpca**
  - Baudat's Generalized Discriminant Analysis (GDA): **slgda**
  - Kernel Fisher Discriminant Analysis (KFD): **slkfd**

- Notes
  - The usage of these functions are similar to that of PCA and LDA.
  - Instead of the sample matrix, they require the gram matrix of the training samples.
  - All of these functions support weighted samples.
  - **slkfd** supports user-supplied kernel scatter matrices. You can also use **slkernelscatter** to construct the kernel scatter matrix according to your need.
  - After the coefficient matrix is learned, you can use **slkernelfea** to extract features.

Regression

- What is regression?

  $Given\ sample\ pairs\ \{(x_i, y_i) : i = 1, \cdots, n\}$

  $Estimate\ \theta :\ \ f(x; \theta) \to y$

- Linear regression
- Logistic regression

Linear regression

- Homogeneous linear regression: $\mathbf{Ax} \rightarrow \mathbf{y}$

$$\hat{\mathbf{A}} = \arg\min_{\mathbf{A}} \sum_{i=1}^{n} \| \mathbf{Ax}_i - \mathbf{y}_i \|^2 = \arg\min_{\mathbf{A}} \| \mathbf{AX} - \mathbf{Y} \|^2$$

$$\hat{\mathbf{A}} = \mathbf{YX}^T (\mathbf{XX}^T)^{\dagger}$$

A = sllinreg(X, Y);

- Ridge regression

$$\hat{\mathbf{A}} = \arg\min_{\mathbf{A}} \sum_{i=1}^{n} \| \mathbf{Ax}_i - \mathbf{y}_i \|^2 + \lambda^2 \| \mathbf{A} \|^2 = \arg\min_{\mathbf{A}} \| \mathbf{A}(\mathbf{X}, \delta\mathbf{I}) - (\mathbf{Y}, \mathbf{0}) \|^2$$

$$\hat{\mathbf{A}} = \mathbf{YX}^T (\mathbf{XX}^T + \lambda^2 \mathbf{I})$$

A = sllinreg(X, Y, ‘lambdas’, lambda_value);

- Augmented linear regression:

$$\mathbf{Ax} + \mathbf{b} \rightarrow \mathbf{y} \Leftrightarrow [\mathbf{A} \quad \mathbf{b}] \begin{bmatrix} \mathbf{x} \\ \mathbf{1} \end{bmatrix} \rightarrow \mathbf{y}$$

A = sllinrega(X, Y, …);

- All functions above support weighted samples.

Multi-class Logistic Regression

- Logistic regression is a statistical regression for discrete values.

$$p(k \mid \mathbf{x}; \theta) = \frac{P_k \exp(\mathbf{a}_k^T \mathbf{x} + b_k)}{\sum_{l=1}^{C} P_l \exp(\mathbf{a}_l^T \mathbf{x} + b_l)}$$

$$\theta = \{\mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_C, b_1, b_2, \cdots, b_C\}$$

- Typically estimated by Maximum Likelihood

$$\theta = \arg\max_{\theta} \sum_{i=1}^{n} \log p(c_i \mid \mathbf{x}_i; \theta)$$

- Usage

[A, b] = sllogistreg(X, nums, ...)

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_C \end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix} b_1 & b_2 & \cdots & b_C \end{bmatrix}$$

•The samples from the same classes should be grouped together, and the nums is used to specify the numbers of samples in all classes in order.
•The function supports weighted samples and different priors on different classes.
•You can specify more options to control the learning, please type “help sllogistreg”.

Tensor Algebra

- Use multi-dimensional array to represent tensor.
- Computational routines
  - **sltensor_dot(T1, T2)** computes the dot product between T1 and T2;
  - **T = sltensor_fold(M, dims, k)** folds a matrix along a specified dimension to form a tensor.
  - **M = sltensor_unfold(T, k)** unfolds a tensor along a specified dimension
  - **sltensor_norm(T)** computes the Frobenius norm of the tensor T.
  - **sltensor_multiply(T, M, k)** multiplies a tensor and a matrix along a specified dimension.
  - **[C, U1, U2, …] = sltensor_svd(T)** performs Higher-Order SVD on the tensor T.
- For the definition of these calculations, please refer to
  - De Lathauwer L., De Moor B., Vandewalle J., ``A multilinear singular value decomposition", SIAM J. Matrix Anal. Appl., vol. 21, no. 4, Apr. 2000, pp. 1253-1278.

K-Means Clustering

- K-means is a widely-used tools in data clustering. It can be used separately or incorporated into a large algorithm such as spectral clustering.

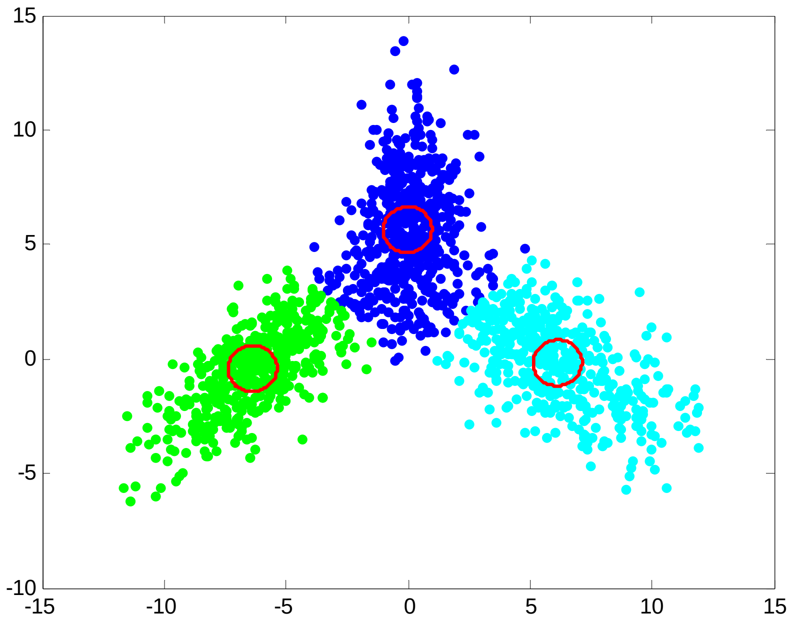
[means, labels] = slkmeans(X, 'K', 3);

When K is large, **KD-tree** may significantly enhanced the efficiency, you can enable KD-tree by
[means, labels] = slkmeans(X, 'K', 3, 'clsfunc', 'ann');

When K is not easy to be determined, it supports **component annealing** strategy. You can set annthres.

It supports **weighted samples**.

By default it will randomly selects initial means, it also supports **user-input intial means.**



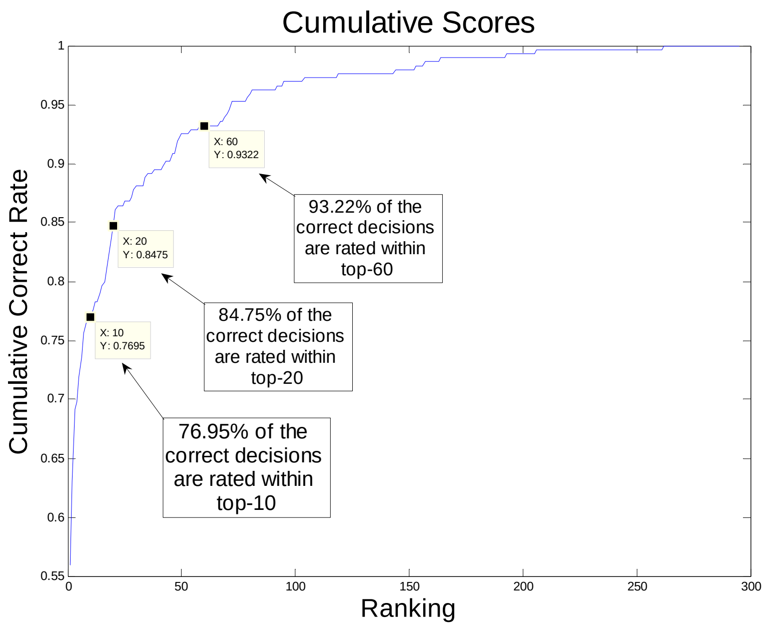
Classification Performance Evaluation

- The classification performance evaluation is based on pairwise score (distance) matrices. Typically, they can be computed using slmetric_pw.
  - If there are m targets (clients, galleries) and n query samples, then a score matrix should be of size m x n, with each column representing the distances(or similarities) from the corresponding query sample to all target samples (classes).
  - These functions require that you specify the attribute of the scores by the argument named op. If you set op be “low”, you mean that the lower the score, the closer the two samples, it is suitable for distances. If you set op be “high”, you mean that the higher the score, the closer the two samples, it is suitable for similarities.

- Performance Evaluation function
  - Recognition (Identification) Performance
  - Verification (Authentication) performance

Recognition(Identification) Performance

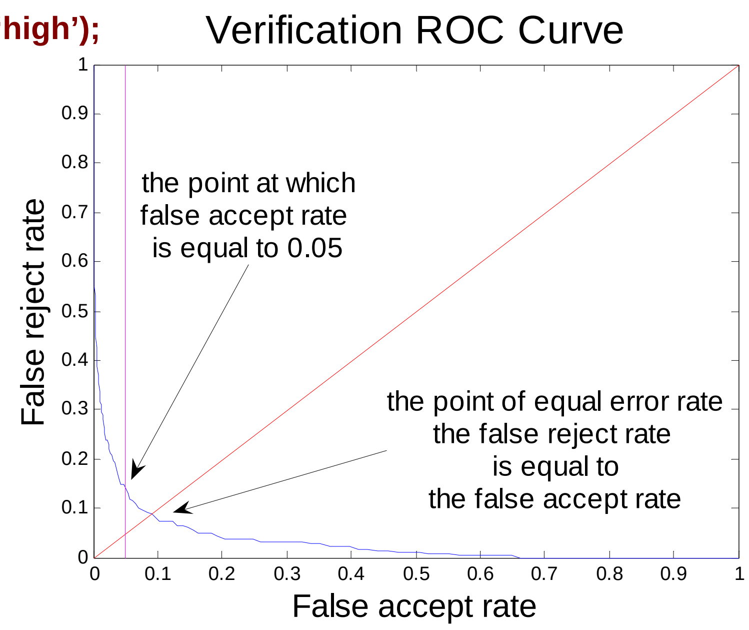
- Given a pairwise Euclidean distance matrix S, to compute the performance, you can write as
  - (Get correct rate) **cr = slcorrectrate(S, gallerylabels, querylabels, ‘low’);**
  - (Get cumulative scores) **cs = slcumuscore(S, gallerylabels, querylabels, ‘low’);**



Verification (Authentication) Performance

- Given a pairwise Normalized Correlation similarity matrix S, to compute the performance, you can write as

```
% get the ROC curve and corresponding threshold values
[thrs, fars, frrs] = slverifyroc(S, gallerylabels, querylabels, 'high');

% get the equal-error-rate
[thr1, fa1, fr1] = slgetroc(thrs, fars, frrs, 'ratio', 1);

% get the point at which fa = 0.05
[thr2, fa2, fr2] = slgetroc(thrs, fars, frrs, 'fixfa', 0.05);
```



Histogram

- Build histogram by voting
- Compute the metrics between histograms

Build vector histogram by voting

- In prototype-based systems, a simple yet effective method is to use k-means to select prototypes first and then build histogram by assigning the samples to the prototypes.
  - Each sample votes for the prototype it is assigned to. So it can be considered as a voting process.
- sltoolbox offers slvote for this task. In addition to the simplest voting, it supports a variety of voting scheme, including that each sample can vote for multiple prototypes with different weights. Such a strategy is sometimes used to build soft histograms.

- The simplest voting: just count how many samples fall in each bin
  - **slvechist(X0, X);**
- Each sample vote for the closest prototype with a weight exponentially attenuating with the increasing of distance.
  - f = @(x) exp(-x);
  - **slvechist(X0, X, ‘countrule’, ‘nmx’, ‘cfunc’, f);**
- Each sample vote for multiple prototypes select by K nearest prototypes, with a weight exponentially attenutating with the increasing of distance. The contribution of each sample is normlized to 1, and the contribution of the final histogram is normalized to 1.
  - f = @(x) exp(-x)
  - **slvechist(X0, X, ‘countrule’, ‘mmns’, ‘cfunc’, f, ‘normalized’, true);**

Compute pairwise distances between histograms

- Storage
  - All the histograms are stored in a matrix with each column being a histogram.
- Usage (similar to slmetric_pw)
  - **D = slhistmetric_pw(H1, H2, metric_type, …)**
- Supported metric types
  - L1 distance
  - L2 distance
  - Quadratic distance
  - hamming distance
  - histogram intersection
  - Chi-square distance
  - Kolmolgorov-Smirkov distance
  - Kramer/Von Mises
  - Kullback-Leibler divergence
  - Jeffrey divergence
- See also
  - **slhistmetric_cp** computes metrics between corresponding pair of histograms

A step further

- Some advanced topics
  - Graph Construction
  - Learning Manifold Embedding
  - Generic Learning Frameworks

How to represent a graph?

- In sltoolbox, it accepts the following graph representations. The detailed specification of the graph representation is given in graph/graph_represent.txt

	Single graph	Bi-graph
Edge set	Struct with fields: **n**: node number **edges**: n x 2 matrix or n x 3 matrix 1st col: source node index 2nd col: target node index 3rd col: the edge value	Struct with fields: **n**: source node number **nt**: target node number **edges**: edge set
Adjacency list	Struct with fields: **n**: node number **targets**: length-n cell array in each cell is column vector or two-column matrix. 1st col: target node index 2nd col: the edge value	Struct with fields: **n**: source node number **nt**: target node number **targets**: cell array to represent adjacency list
Adjacency matrix	n x n matrix (can be either full or sparse). A(i, j) represent the edge from i-th node to the j-th node.	n x nt matrix (can be either full or sparse), A(i,j) represent the edge from i-th source node to the j-th target node.

Notations:
n: the number of nodes
in the source node set
nt: the number of nodes
in the target node set

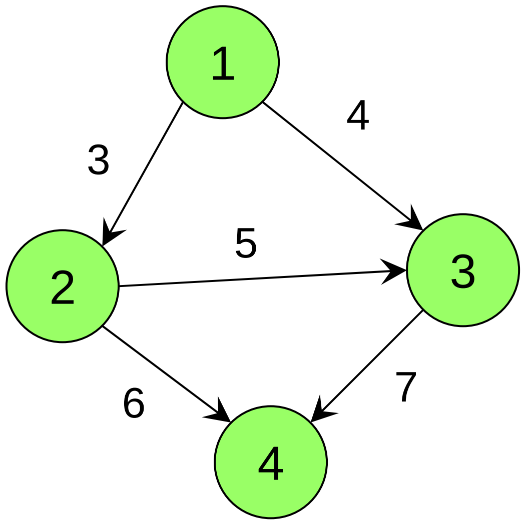
for single graph
n = nt = the number of all nodes

for bigraph
n and nt maybe not equal.

Represent graph (single)

Representations without edge values

Valued representations



Edge set

n = 4

edges =

```
1  2
1  3
2  3
2  4
3  4
```

Adjacency list

n = 4

targets = {

[2; 3];

[3; 4];

4;

[ ] };

Adjacency matrix

```
0  1  1  0
0  0  1  1
0  0  0  1
0  0  0  0
```

Edge set

n = 4

edges =

```
1  2  3
1  3  4
2  3  5
2  4  6
3  4  7
```

Adjacency list

n = 4

targets = {

[2 3; 3 4];

[3 5; 4 6];

[4 7];

[ ] };

Adjacency matrix

```
0  3  4  0
0  0  5  6
0  0  0  7
0  0  0  0
```

Represent bi-graph

Representations without edge values

Edge set

n = 2

nt = 3

edges =

```
  1  1
  1  2
  2  2
  2  3
```

Adjacency list

n = 2

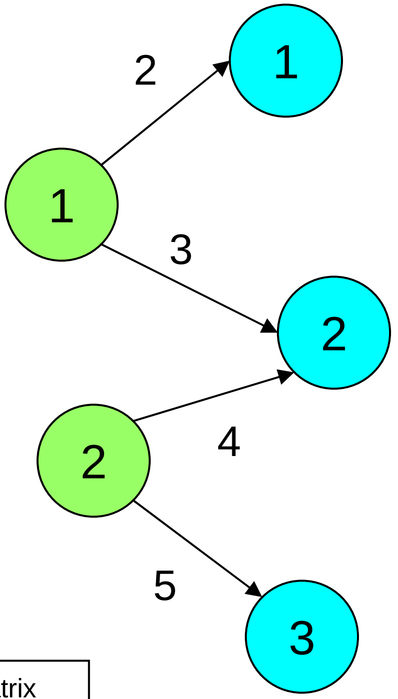
targets = {

[1; 2];

[2; 3] };

Adjacency matrix

```
1  1  0
0  1  1
```



Valued representations

Edge set

n = 2

nt = 3

edges =

```
  1  1  2
  1  2  3
  2  2  4
  2  3  5
```

Adjacency list

n = 2

nt = 3

targets = {

[1 2; 2 3];

[2 4; 3 5] };

Adjacency matrix

```
2  3  0
0  4  5
```

Graph Manipulation

gi = slgraphinfo(G)

•check the correctness of representation
•get the information of the graph
- type: 'ge' (general graph) | 'bi' (bi-graph)
-form: 'edgeset' | 'adjlist' | 'adjmat'
-n: the number of (source) nodes
-nt: the number of (target) nodes
-valued: whether contain edge values

conversion

sladjmat: convert to adjacency matrix
sladjlist: convert to adjacency list
sledgeset: convert to edge set

As = slsymgraph(A);
As = slsymgraph(A, rule);

symmetrize a graph adjacency matrix

Implement different rules:

avgor	if aij != 0 and aji != 0: d = (aij + aji) / 2
	if aij != 0 and aji == 0: d = aij
	if aij == aji == 0, d = 0

avgand	if aij != 0 and aji != 0: d = (aij + aji) / 2 otherwise d = 0
or	d = aij | aji
and	d = aij & aji
simavg	d = (aij + aji) / 2

Neighborhood Graph Construction

- Neighborhood Graph plays an important role in graph-based learning.
- **slfindnn** (the function to find neighborhood)
  - **[nnidx, dists] = slfindnn(X0, X, ‘knn’, ‘K’, 3);**
    - Find three nearest neighbors from X0 for every query sample in X.
    - Using exhaustive comparison to determine the nearest neighbors
    - nnidx is adjacency list, dists is corresponding distance values.
  - **[nnidx, dists] = slfindnn(X0, X, ‘ann’, ‘K’, 3);**
    - Use more efficient KD-tree to search approximate nearest neighbors
  - **[nnidx, dists] = slfindnn(X0, X, ‘knn’, ‘K’, 4, ‘metric’, ‘cityblk’);**
    - Change to use L1 distance to search neighbors
  - **[nnidx, dists] = slfindnn(H0, H, ‘knn’, ‘K’, 5, ‘metric’, @(x1, x2) slhistmetric(x1, x2, ‘kldiv’));**
    - Change to use slhistmetric function to compute KL-divergence as the base of neighborhood search.
  - **[nnidx, dists] = slfindnn(X0, X, ‘eps’, ‘e’, 1)**
    - Find the neighbors with distance smaller than 1
- **slnngraph** constructs neighborhood graph
  - find neighbors and also construct the graph
- **G = slnngraph(X0, X, {‘ann’, ‘K’, 3})**
  - construct a neighborhood graph by finding the 3 nearest neighbors for each sample.
  - The edge values are set to the distances
- **G = slnngraph(X0, X, {‘ann’, ‘K’, 3}, ‘tfunctor’, @(x) exp(-x))**
  - construct a neighborhood graph with the edge value computed by exp(-x) on distance values.
- **G = slnngraph(X0, [ ], {‘knn’, ‘K’, 3}, ‘sym’, true)**
  - construct a neighborhood graph and force it to be symmetric
  - Set X to [ ] is to tell that the graph is built on X0 and the edges connecting self are removed.
- **G = slnngraph(X0, X0, {‘knn’, ‘K’, 3, ‘metric’, {‘quaddiff’, Q}}, ‘sparse’, true)**
  - construct a neighborhood graph and make it represented by sparse matrix
  - Set X0 = X is to tell that the graph is built on X0, and the edges connecting the same nodes will be preserved.
  - use quadratic form distance instead of Euclidean distance with the quadratic matrix given by Q.

More convenient function

- **slaffinitymat**
  - directly constructs typical affinity matrix
- **Usage**
  - **A = slaffinitymat(X, [ ], {'ann', 'K', 5}, 'kernel', 'heat', 'diffusion', v)**
    - Use the following formula to construct affinity values

$$a = \exp\left(-\frac{d^2}{2\sigma^2}\right), \quad \sigma^2 = v^2 \text{average}(d^2)$$

    - Use KD-tree to search 5 neighbors for each sample
  - **A = slaffinitymat(X, [ ], {'knn', 'K', 5}, 'kernel', 'simple')**
    - Simply sets the values of all edges connecting neighboring nodes to 1
    - Use exhaustive search to find 5 nearest neighbors for each sample.
  - **A = slaffinitymat(X, [ ], {'knn', 'K', 5}, 'kernel', 'simple', 'excludeself', true)**
    - remove the edges connecting the same nodes.

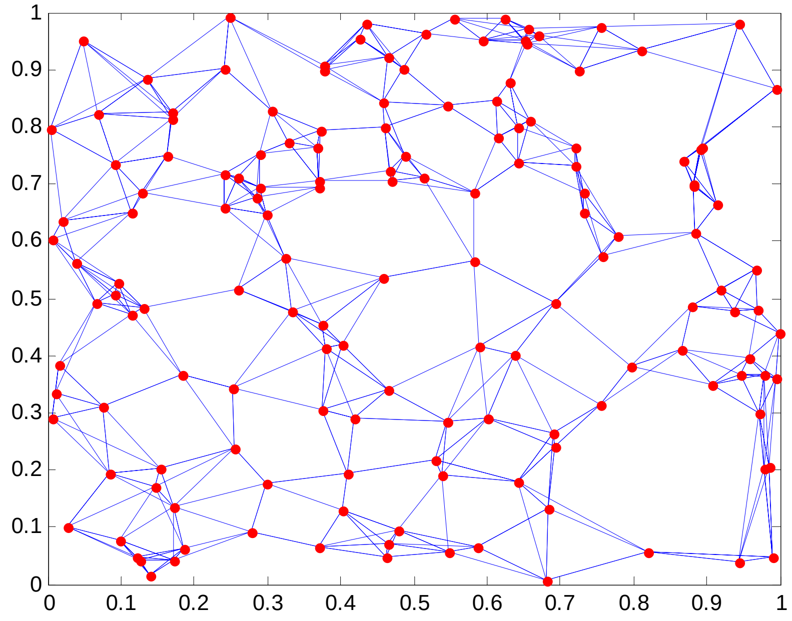


Illustration: the graph constructed by Knn, with K = 5.

Learning Manifold Embedding

- sltoolbox implements a set of manifold embedding learning algorithms
  - Multidimensional Scaling (MDS)
  - ISOMAP
  - Locally Linear Embedding (LLE)
  - Laplacian Eigenmap
  - Local Tangent Space Alignment (LTSA)
- Mainly based on graph instead of samples
  - Decoupling the embedding learning algorithm with graph construction algorithm will give you more flexibility to design your own algorithms.
  - Some core components that may be reusable in other algorithms are separated as single function

Design roadmap

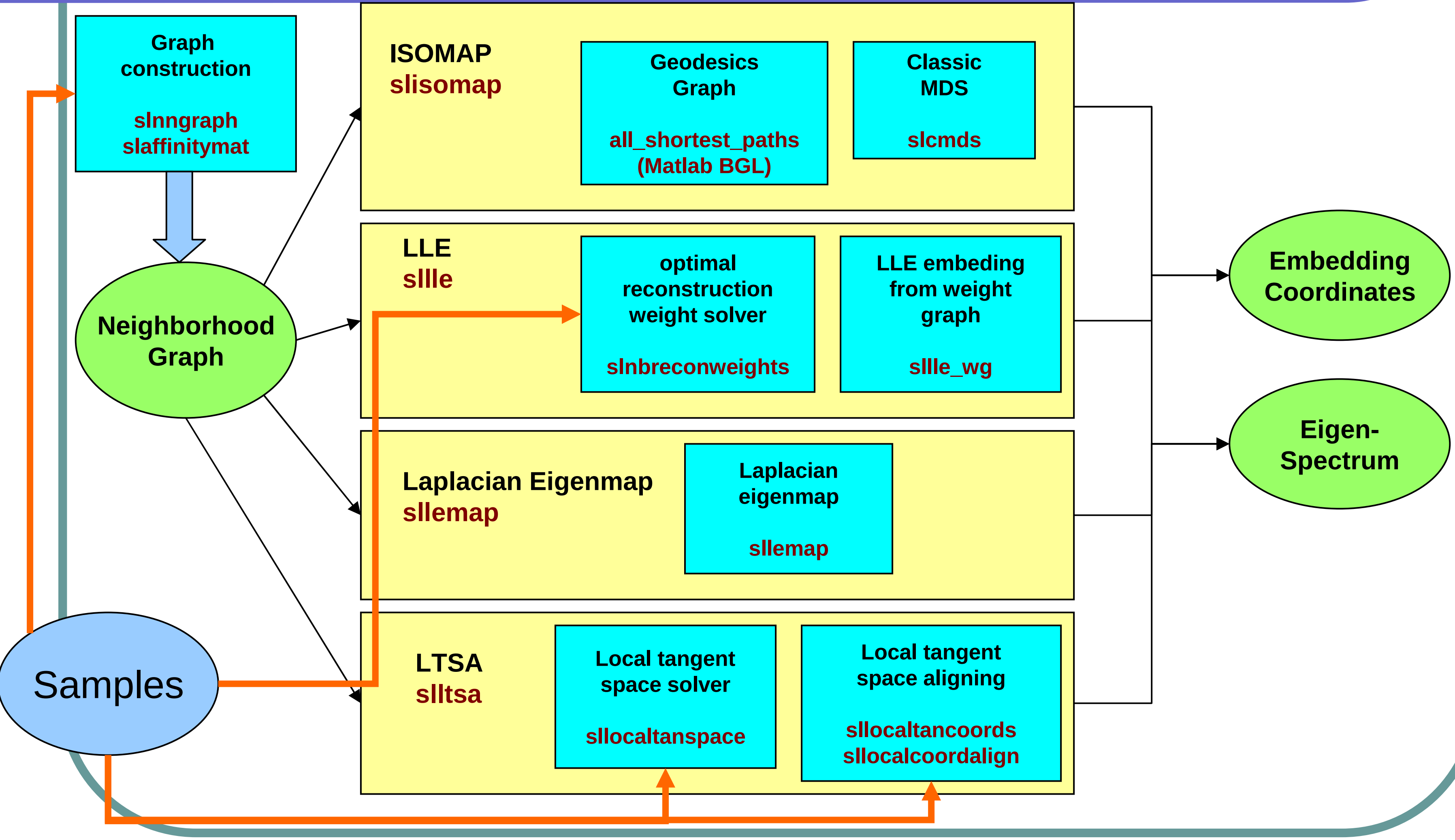
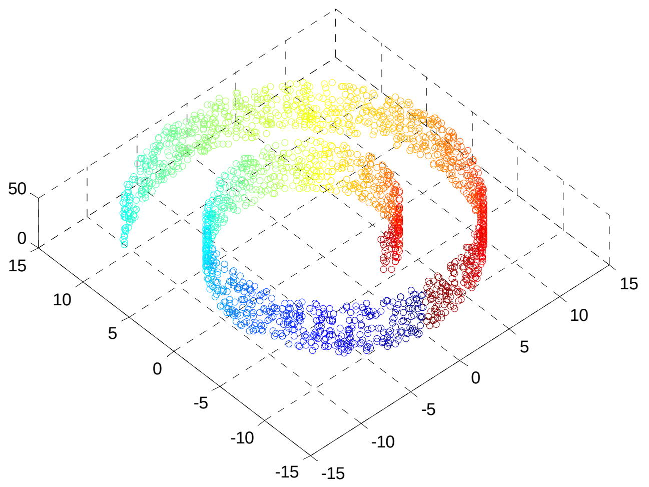
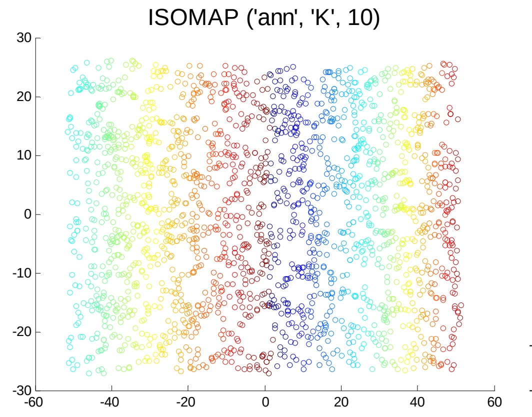
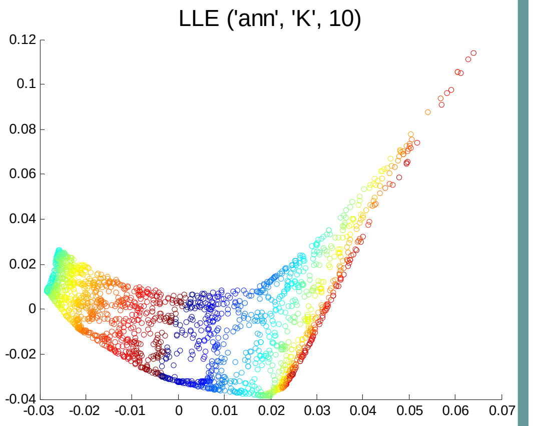
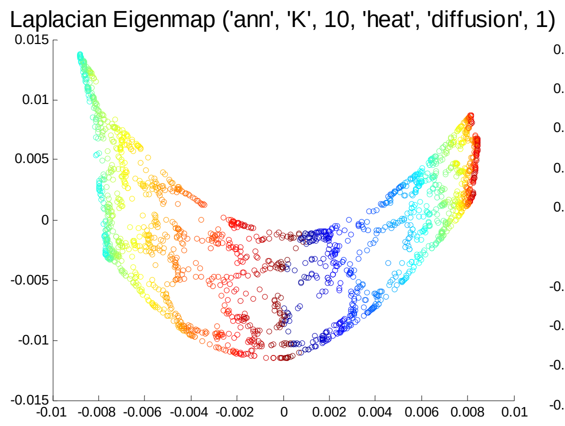


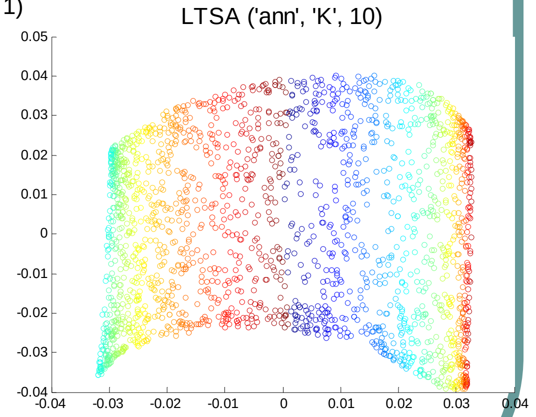
Illustration of some results









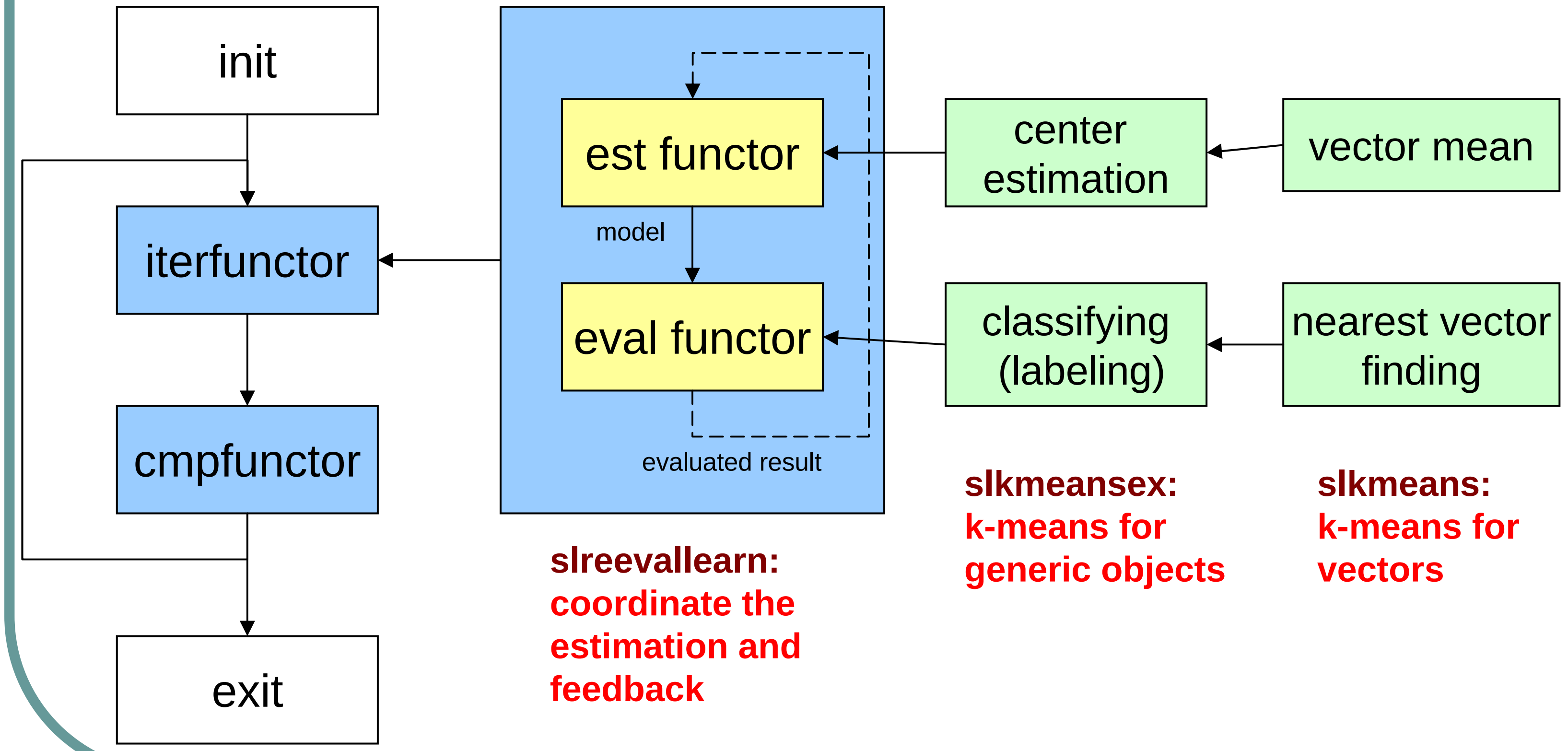


Generic Learning Framework

- The sltoolbox also tries to implement generic learning framework. It is an explorative work.
  - The idea is that the framework function controls the main procedure, and invoke user-input callback functors at proper places.
  - The interfaces of the functors are explicitly specified and clearly defined.
  - It is somewhat like the delegate or function pointer in high-level languages.
- Examples
  - Finite mixture model (slfmm)
  - Generalized K-means (slkmeansex)
  - General iterative learning (sliterproc)
  - Re-evaluation learning (slreevallearn)

The construction of K-means

- I would like to use K-means as an example to explain how I design the learning architecture.



slreevallearn:
coordinate the estimation and feedback

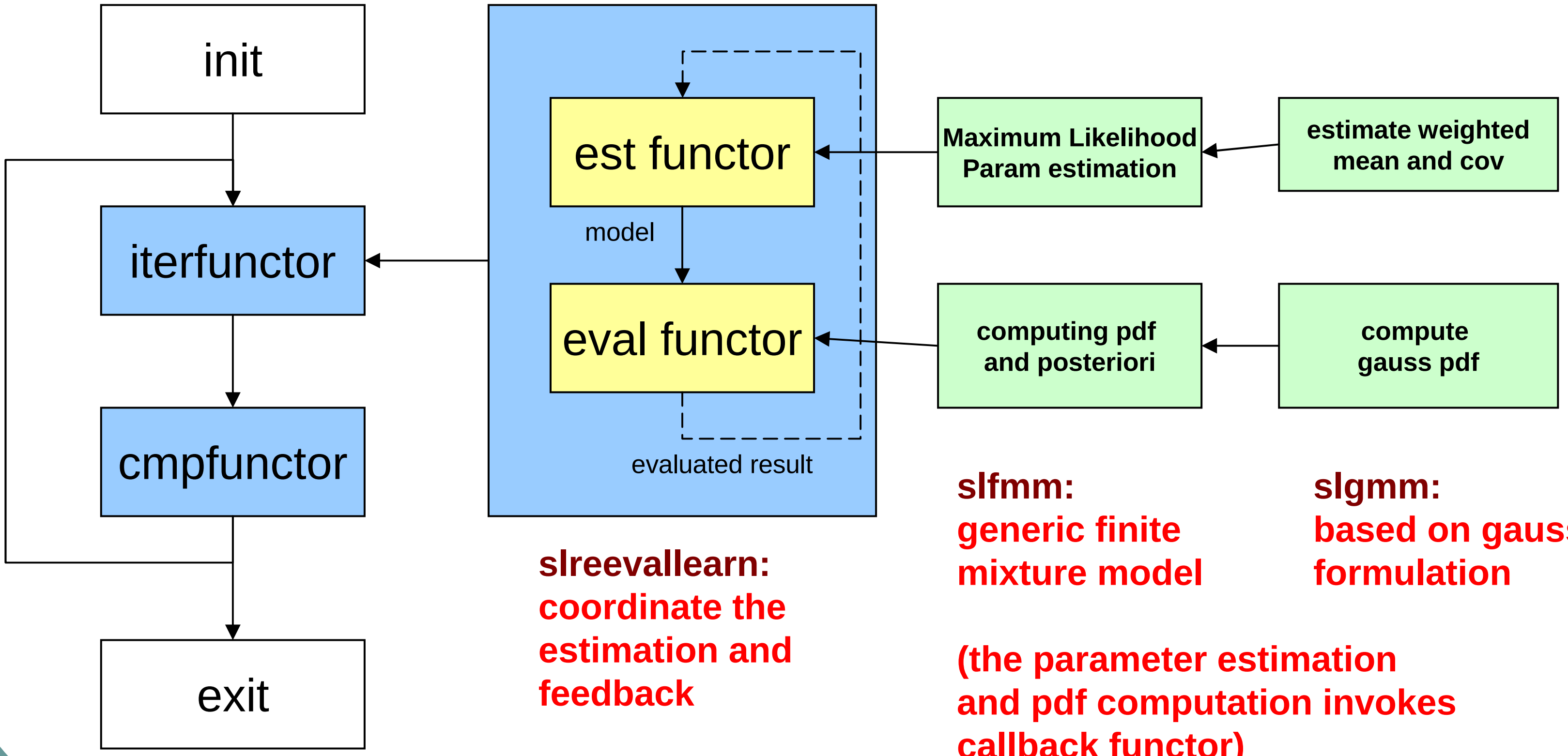
slkmeansex:
k-means for generic objects

slkmeans:
k-means for vectors

sliterproc:
control iteration process

The construction of FMM/GMM

- Another typical example is finite mixture model


init
iterfunctor
cmpfunctor
exit
est functor
model
eval functor
evaluated result
Maximum Likelihood Param estimation
estimate weighted mean and cov
computing pdf and posteriori
compute gauss pdf

slreevallearn:
coordinate the estimation and feedback

slfmm:
generic finite mixture model

slgmm:
based on gaussian formulation

(the parameter estimation and pdf computation invokes callback functor)

sliterproc:
control iteration process

Discussions on Architect design

- Core, Framework, Algorithm
  - decouple
  - inheritance
- Extensible design and Parameter Passing
  - slfindnn and slnngraph
  - sledges2adjmat
- About LDA
  - The balance between easy-to-use and flexible

Implementation: Efficiency and Robustness

- slmetric_pw and sldistmean
  - The power of mathematics
- sllemap
  - stability of eigen-problem
- sllabeledsum
  - considerations on algorithm design

End

Thank you