# Appendix A

# Derivatives

## A.1 Derivatives in TKE

We rewrite the objective function in matrix form as

$$L = -\mathrm{tr}[\mathbf{KS}] + \lambda_k \mathrm{tr}[\mathbf{KK}] + \lambda_x \mathrm{tr}[\mathbf{XX}^\top] \tag{A.1}$$

where we denote $\mathbf{K}_\mathbf{x}$ by $\mathbf{K}$ and $\mathbf{K}_\mathbf{y}$ by $\mathbf{S}$ to simplify the notation. Chain rule is applied to solve for $\frac{\partial L}{\partial \mathbf{K}}$. Since $\mathbf{K}$ is a function of $\mathbf{X}$, we first compute the partial derivatives of $\mathbf{K}$ from (A.1)

$$\frac{\partial L}{\partial \mathbf{K}} = 2\lambda_k \mathbf{K} - \mathbf{S}. \tag{A.2}$$

Basically, $\kappa_x(\cdot,\cdot)$ is the RBF kernel. Therefore, $\mathbf{K}_{ij} = \gamma \exp\{-\sigma\|\mathbf{x}_i - \mathbf{x}_j\|^2\}$ and $\mathbf{K}$ is

$$\mathbf{K} = \begin{bmatrix} 0 & \gamma e^{\{-\sigma\|\mathbf{x}_1 - \mathbf{x}_N)\|^2\}} & \dots & \gamma e^{\{-\sigma\|\mathbf{x}_1 - \mathbf{x}_N)\|^2\}} \\ \gamma e^{\{-\sigma\|\mathbf{x}_2 - \mathbf{x}_1)\|^2\}} & 0 & \dots & \gamma e^{\{-\sigma\|\mathbf{x}_2 - \mathbf{x}_N)\|^2\}} \\ \vdots & \vdots & & \vdots \\ \gamma e^{\{-\sigma\|\mathbf{x}_N - \mathbf{x}_1)\|^2\}} & \gamma e^{\{-\sigma\|\mathbf{x}_N - \mathbf{x}_2)\|^2\}} & \dots & 0 \end{bmatrix}.$$

According to the chain rule which reads $\frac{\partial g(U)}{\partial \mathbf{X}_{ij}} = \sum_k \sum_l \frac{\partial g(U)}{\partial u_{kl}} \frac{\partial u_{kl}}{\partial \mathbf{X}_{ij}}$, we have

$$\frac{\partial L}{\partial \mathbf{X}_{kl}} = \sum_{i=1}^{N} \sum_{i=1}^{N} \frac{\partial L}{\partial \mathbf{K}_{ij}} \frac{\partial \mathbf{K}_{ij}}{\partial \mathbf{X}_{kl}} + 2\lambda_x \mathbf{X}_{kl} \tag{A.3}$$

where the second term is obtained from the last term in (A.1). In the following discussion, we only consider the first term in (A.3) and focus on $\frac{\partial \mathbf{K}_{ij}}{\partial \mathbf{X}_{kl}}$.

$$\frac{\partial \mathbf{K}_{ij}}{\partial \mathbf{X}_{kl}} = \begin{cases} 0, & i, j \neq k; \\ 0, & i = j = k; \\ \frac{\partial \mathbf{K}_{ij}}{\partial \mathbf{X}_{kl}}, & i = k \text{ or } j = k, \end{cases} \tag{A.4}$$

then

$$\begin{cases} \frac{\partial \mathbf{K}_{kj}}{\partial \mathbf{X}_{kl}} = -2\sigma\gamma(\mathbf{X}_{kl} - \mathbf{X}_{jl}) \exp\{-\sigma(\mathbf{x}_i - \mathbf{x}_j)^\top(\mathbf{x}_i - \mathbf{x}_j)\}, & i = k; \\ \frac{\partial \mathbf{K}_{ik}}{\partial \mathbf{X}_{kl}} = -2\sigma\gamma(\mathbf{X}_{kl} - \mathbf{X}_{jl}) \exp\{-\sigma(\mathbf{x}_i - \mathbf{x}_j)^\top(\mathbf{x}_i - \mathbf{x}_j)\}, & j = k. \end{cases}$$

From the symmetry of the kernel Gram matrix $\mathbf{K}$ we have

$$\frac{\partial L}{\partial \mathbf{X}_{kl}} = \sum_{j=1}^{N} \frac{\partial L}{\partial \mathbf{K}_{kj}} \frac{\partial \mathbf{K}_{kj}}{\partial \mathbf{X}_{kl}} + \sum_{i=1}^{N} \frac{\partial L}{\partial \mathbf{K}_{ik}} \frac{\partial \mathbf{K}_{ik}}{\partial \mathbf{X}_{kl}} = 2\sum_{j=1}^{N} \frac{\partial L}{\partial \mathbf{K}_{kj}} \frac{\partial \mathbf{K}_{kj}}{\partial \mathbf{X}_{kl}}.$$

From (A.4) we see that only when $i = k$, the derivatives of $L$ w.r.t $\mathbf{X}_{kl}$ are not zeros, so we just need to differentiate the $k$ row of $\mathbf{K}$ w.r.t $\mathbf{X}_{kl}$ in order to obtain $\frac{\partial L}{\partial \mathbf{X}_{kl}}$.

Because the $\frac{\partial \mathbf{K}_{ij}}{\partial \mathbf{X}_{kl}} = 0$ if $i \neq k$, so the derivatives of $\mathbf{K}$ w.r.t $\mathbf{X}$ can be reorganized as Figure A.1 in a 3-dimensional array by discarding the zeros. The elements in the array is indexed in the order of column, row, and slide. Denote the $\frac{\partial L}{\partial \mathbf{K}}$ by $\mathbf{H}$ and the 3-dimensional array in Figure A.1 by $\mathbf{G}$, then we can express the $\frac{\partial L}{\partial \mathbf{X}}$ in this form

$$\frac{\partial L}{\partial \mathbf{X}} = \begin{bmatrix} \mathbf{H}_{1,:} \cdot \mathbf{G}_{:,1,1} & \dots & \mathbf{H}_{1,:} \cdot \mathbf{G}_{:,l,1} & \dots & \mathbf{H}_{1,:} \cdot \mathbf{G}_{:,d,1} \\ \vdots & & \vdots & & \vdots \\ \mathbf{H}_{k,:} \cdot \mathbf{G}_{:,1,k} & \dots & \mathbf{H}_{k,:} \cdot \mathbf{G}_{:,l,k} & \dots & \mathbf{H}_{k,:} \cdot \mathbf{G}_{:,d,k} \\ \vdots & & \vdots & & \vdots \\ \mathbf{H}_{N,:} \cdot \mathbf{G}_{:,1,N} & \dots & \mathbf{H}_{N,:} \cdot \mathbf{G}_{:,l,N} & \dots & \mathbf{H}_{N,:} \cdot \mathbf{G}_{:,d,N} \end{bmatrix}$$

and the $k$th row of $\frac{\partial L}{\partial \mathbf{X}}$ can be computed efficiently by $\mathbf{H}_{k,:} \cdot \mathbf{G}_{:,:,k}$, i.e. multiplying the $k$th row of $\mathbf{H}$ by $k$th slide of $\mathbf{G}$ where the colon in the matrix notation means to take all the elements in that dimension. Finally, we can combine the above result with $2\lambda_x \mathbf{X}_{kl}$ to have the derivative of $L$ with respect to $\mathbf{X}$.

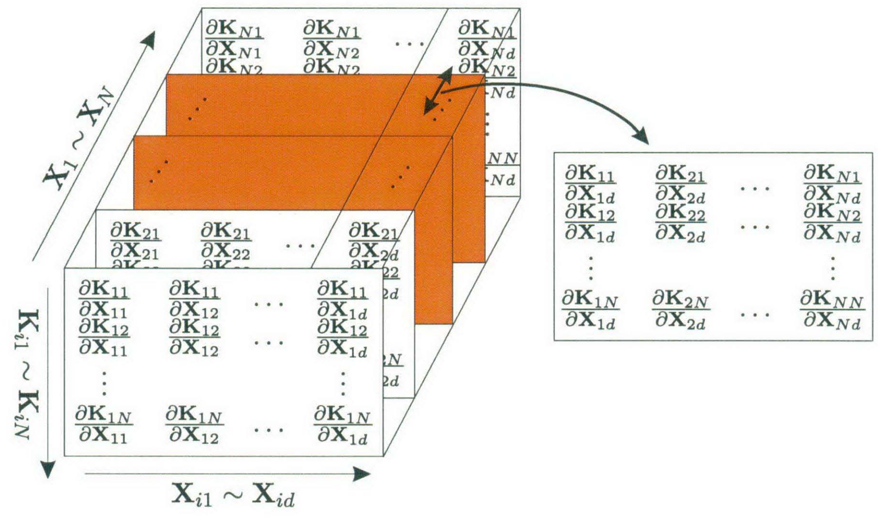
$\mathbf{X}_1 \sim \mathbf{X}_N$
$\mathbf{K}_{i1} \sim \mathbf{K}_{iN}$
$\mathbf{X}_{i1} \sim \mathbf{X}_{id}$
$\frac{\partial \mathbf{K}_{11}}{\partial \mathbf{X}_{11}}$ $\frac{\partial \mathbf{K}_{11}}{\partial \mathbf{X}_{12}}$ $\cdots$ $\frac{\partial \mathbf{K}_{11}}{\partial \mathbf{X}_{1d}}$
$\frac{\partial \mathbf{K}_{12}}{\partial \mathbf{X}_{11}}$ $\frac{\partial \mathbf{K}_{12}}{\partial \mathbf{X}_{12}}$ $\cdots$ $\frac{\partial \mathbf{K}_{12}}{\partial \mathbf{X}_{1d}}$
$\frac{\partial \mathbf{K}_{1N}}{\partial \mathbf{X}_{11}}$ $\frac{\partial \mathbf{K}_{1N}}{\partial \mathbf{X}_{12}}$ $\cdots$ $\frac{\partial \mathbf{K}_{1N}}{\partial \mathbf{X}_{1d}}$
$\frac{\partial \mathbf{K}_{21}}{\partial \mathbf{X}_{21}}$ $\frac{\partial \mathbf{K}_{21}}{\partial \mathbf{X}_{22}}$ $\cdots$ $\frac{\partial \mathbf{K}_{21}}{\partial \mathbf{X}_{2d}}$
$\frac{\partial \mathbf{K}_{N1}}{\partial \mathbf{X}_{N1}}$ $\frac{\partial \mathbf{K}_{N1}}{\partial \mathbf{X}_{N2}}$ $\cdots$ $\frac{\partial \mathbf{K}_{N1}}{\partial \mathbf{X}_{Nd}}$
$\frac{\partial \mathbf{K}_{11}}{\partial \mathbf{X}_{1d}}$ $\frac{\partial \mathbf{K}_{21}}{\partial \mathbf{X}_{2d}}$ $\cdots$ $\frac{\partial \mathbf{K}_{N1}}{\partial \mathbf{X}_{Nd}}$
$\frac{\partial \mathbf{K}_{12}}{\partial \mathbf{X}_{1d}}$ $\frac{\partial \mathbf{K}_{22}}{\partial \mathbf{X}_{2d}}$ $\cdots$ $\frac{\partial \mathbf{K}_{N2}}{\partial \mathbf{X}_{Nd}}$
$\frac{\partial \mathbf{K}_{1N}}{\partial \mathbf{X}_{1d}}$ $\frac{\partial \mathbf{K}_{2N}}{\partial \mathbf{X}_{2d}}$ $\cdots$ $\frac{\partial \mathbf{K}_{NN}}{\partial \mathbf{X}_{Nd}}$

Figure A.1: The organization of the partial derivatives in $\mathbf{K}$ wrt. $\mathbf{X}$.

## A.2 Derivatives in BCTKE

In BCTKE, we just substitute the kernel mapping function into the objective function of TKE to replace every $\mathbf{x}_i$ by the backward mapping. So here we still use chain rule to get the derivatives of $L$ with respect to the coefficients in the mapping function. We first give the mapping function in matrix form

$$\mathbf{X} = \mathbf{K_y A} = \mathbf{SA}$$

which can be further expanded to

$$\mathbf{X} = \begin{bmatrix} S_{11} & \dots & S_{1N} \\ \vdots & & \vdots \\ S_{k1} & \dots & S_{kN} \\ \vdots & & \vdots \\ S_{N1} & \dots & S_{NN} \end{bmatrix} \begin{bmatrix} \alpha_{11} & \dots & \alpha_{1d} \\ \vdots & & \vdots \\ & & \\ \vdots & & \vdots \\ \alpha_{N1} & \dots & \alpha_{Nd} \end{bmatrix}$$

$$= \begin{bmatrix} \sum_l S_{1m}\alpha_{m1} & \dots & \sum_m S_{1m}\alpha_{md} \\ \vdots & & \vdots \\ \sum_m S_{km}\alpha_{m1} & \dots & \sum_m S_{km}\alpha_{md} \\ \vdots & & \vdots \\ \sum_m S_{Nm}\alpha_{m1} & \dots & \sum_m S_{Nm}\alpha_{md} \end{bmatrix} = \begin{bmatrix} \mathbf{X}_{11} & \dots & \mathbf{X}_{1d} \\ \vdots & & \vdots \\ \mathbf{X}_{k1} & \dots & \mathbf{X}_{kd} \\ \vdots & & \vdots \\ \mathbf{X}_{N1} & \dots & \mathbf{X}_{Nd} \end{bmatrix}$$

showing the relation between elements clearly. So

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{A}_{ij}} = \frac{\partial L}{\partial \alpha_{ij}} &= \sum_k \sum_l \frac{\partial L}{\partial \mathbf{X}_{kl}} \frac{\partial \mathbf{X}_{kl}}{\partial \alpha_{ij}} \qquad (\mathbf{X}_{kl} = \sum_m S_{km}\alpha_{ml}) \\ &= \sum_k \frac{\partial L}{\partial \mathbf{X}_{kj}} \frac{\partial \mathbf{X}_{kj}}{\partial \alpha_{ij}} \qquad (\mathbf{X}_{kj} = \sum_m S_{km}\alpha_{mj}) \\ &= \sum_k \frac{\partial L}{\partial \mathbf{X}_{kj}} \cdot S_{ki} \end{aligned}$$

where we made use of the fact that

$$\frac{\partial \mathbf{X}_{kl}}{\partial \alpha_{ij}} = \begin{cases} 0, & l \neq j \Rightarrow \text{discard}; \\ S_{ki}, & l = j. \end{cases}$$

Use the $\frac{\partial L}{\partial \mathbf{X}}$ we have already in last section and denote $\frac{\partial L}{\partial \mathbf{X}}$ by $\mathbf{M}$,

$$\frac{\partial L}{\partial \mathbf{A}} = \mathbf{SM}. \tag{A.5}$$

It is also possible to define the mapping as

$$\mathbf{X}_{ij} = \sum_{m=1}^{N} \alpha_{mj}\kappa_y(\mathbf{y}_i, \mathbf{y}_m) + \theta_j.$$

Let $\Theta = [\theta_1, \dots, \theta_d]$, and $\mathbf{1} = [1 \dots 1]^\top$ all 1 vector with size $N \times 1$,

$$\mathbf{X} = \mathbf{K_y A} + \mathbf{1}\Theta = \mathbf{SA} + \mathbf{1}\Theta$$

According to the chain rule $\frac{\partial L}{\partial \theta_j} = \sum_k \sum_l \frac{\partial L}{\partial \mathbf{X}_{kl}} \frac{\partial \mathbf{X}_{kl}}{\partial \theta_j}$, and

$$\frac{\partial \mathbf{X}_{kl}}{\partial \theta_j} = \begin{cases} 0, & l \neq j \Rightarrow \text{discard} \\ 1, & l = j \end{cases},$$

$\frac{\partial L}{\partial \theta_j} = \sum_k \frac{\partial L}{\partial \mathbf{X}_{kj}}$, therefore $\frac{\partial L}{\partial \theta_j} = \mathbf{1}^\top \cdot \mathbf{M}_{:,j}$ which leads to

$$\frac{\partial L}{\partial \Theta} = \mathbf{1}^\top \cdot \mathbf{M}.$$

The initial values for $\Theta$ can be all zeros. The optimum can be eventually reached by gradient based algorithm.

There are two hyperparameters in the kernel $\kappa_x$ which can also be optimized in the minimization. So we also need to work out the derivatives of $L$ w.r.t those hyperparameters collected in $\Xi = [\gamma, \sigma]^\top$. The same procedure involves the chain rule $\frac{\partial L}{\partial \Xi_i} = \sum_k \sum_l \frac{\partial L}{\partial \mathbf{K}_{kl}} \frac{\partial \mathbf{K}_{kl}}{\partial \Xi_i}$. We put the partial derivatives of $\mathbf{K}_{ij}$ with respect to $\gamma$ and $\sigma$ into matrices $\mathbf{J}^\gamma$ and $\mathbf{J}^\sigma$ respectively and hence their elements are defined as

$$\mathbf{J}^\gamma_{ij} = \frac{\partial \mathbf{K}_{kl}}{\partial \gamma} = \exp\{-\sigma \|\mathbf{x}_k - \mathbf{x}_l\|^2\},$$

$$\mathbf{J}^\sigma_{ij} = \frac{\partial \mathbf{K}_{kl}}{\partial \sigma} = -\gamma \|\mathbf{x}_k - \mathbf{x}_l\|^2 \exp\{-\sigma \|\mathbf{x}_k - \mathbf{x}_l\|^2\}$$

and we denote $\mathbf{H} = \frac{\partial L}{\partial \mathbf{K}}$, finally

$$\frac{\partial L}{\partial \gamma} = \mathrm{tr}[(\mathbf{H} \circ \mathbf{J}^\gamma)(\mathbf{H} \circ \mathbf{J}^\gamma)^\top], \tag{A.6}$$

$$\frac{\partial L}{\partial \sigma} = \mathrm{tr}[(\mathbf{H} \circ \mathbf{J}^\sigma)(\mathbf{H} \circ \mathbf{J}^\sigma)^\top], \tag{A.7}$$

where $\circ$ is Hadamard (elementwise) product.

# Appendix B

# Matlab Code for TKE and BCTKE

## B.1 Notes for the code

The code is for TKE and BCTKE. It must run with NETLAB and GPLVM toolboxes. NETLAB is available at http://www.ncrg.aston.ac.uk/netlab/ and GPLVM can be downloaded freely from http://www.cs.man.ac.uk/ neill/fgplvm/.

## B.2 tke.m

```
%{
tkeback creates a TKE model according to the given data.

Usage:  [model,options]=tke(op)

The input argument op should at least include the following
values:
(The optimal parameters are given as well)
op.n=13; %The size of neighborhood in n nearest neighbor
          filtering for input data kernel.
op.iters=400; %The iterations of the optimazation process.
op.nc=10; %The # of classes of the input data.
op.ns={50}; %The # of samples in each class. If the numbers
             of samples are equal, set it as the given
     sample. Otherwise, each cell is the number for
     each class.
op.latentDim=2; %The target latent space dimension.
op.kernelReg=0.005; %Kernel regularization parameter.
op.variableReg=0.001; %Variable regularization parameter.
op.kgram %Kernel Gram matrix of input data used by TKE.

op.initalg.alg='KLE'; %The name of the initialization method.
op.initalg.kern='SCIGV'; %The name of the kernel used in
                          initialization.
op.kern='SCIGV'; %The name of the kernel used in TKE.
op.initX: %The initialization of X initialized by KLE or
           KPCA etc.

op.optimiser = 'scg'; %The optimiser, could be 'conjgrad'
                       or 'scg'.

If constraints are included, should specify this parameter:
op.back='km'; %Setup constraint type which can be kernel
               mapping now.

%}

function [model,options]=tke(op)

%% Record the start time
t = clock;

%% Check kernel gram matrix defined on input data
if ~isfield(op,'kgram')
    error('Should assign kernel gram matrix of input data!');
end

%% Filter the kernel Gram matrix using n nearest neighboring
W = FilteringKern(op.kgram, 'nn', op.n);

%% Setup options for TKE model
options = TKEOptions;

options.kernelReg = op.kernelReg;
options.variableReg = op.variableReg;

if isfield(op,'optimiser')
    options.optimiser = op.optimiser;
else
    options.optimiser = 'scg';
end;
```

```
latentDim = op.latentDim; options.X = op.initX;

%% setup constraint type and init constraint model
% If constraints are wanted, setup parameters for it.
% Use switch structure to embrace other backward mapping
% function in future.
if isfield(op,'back')
    switch op.back
        case 'km' % use km type constraints
            options.back = 'km';
            options.backOptions.X = NaN;
            options.backOptions.kern = 'custom';
            options.backOptions.K = op.kgram;
            options.optimiseInitBack = 1;
        otherwise
            error('Have not got such type of constraints!');
    end
end

%% Create TKE model
model = TKECreate(latentDim, W, options);

%% start the optimization process
if ~isfield(op, 'iters')
    iters = 400;
else
    iters = op.iters;
end;

model = TKEOptimise(model, 1, iters);

%% Get 1NN classification error of the embedding result
% 1NN classification error and error rate
labels=GenLabels(op.nc,op.ns);
if isfield(op, 'nc') & isfield(op,'ns')
    model.err = knnrelabel(model.X,labels,1);
    model.errate = model.err/length(labels);
    options.initXerrate = ...
       knnrelabel(options.X,labels,1)/length(labels);
end;

% Record the time consumed in TKE modeling.
model.comptime = etime(clock,t);
```

## B.3 FilteringKern.m

```
%{
FilteringKern will filter the kernel Gram matrix using
n nearest neighbor.
Usage: Kf = FilteringKern(K,n)

Notes for arguments:

INPUTS:
K: the kernel gram matrix of data set. This can be
   calculated independently using any kernel for
   any type of data.
n: size of neighborhood.

OUTPUTS:
Kf: sparse symmetric N by N matrix after filtering.

FilteringKern.m version 1.0
%}

function Kf = FilteringKern(K,n)
% calculate the adjacency matrix for DATA
A = adjgraph(K, TYPE, n);
Kf = diag(diag(K));

% disassemble the sparse matrix
[A_i, A_j, A_v] = find(A);

for i = 1: size(A_i)
    Kf(A_i(i), A_j(i)) = K(A_i(i), A_j(i));
end;
```

## B.4 adjgraph.m

```
%{
adjacency will Compute the adjacency graph of the data
set provided the similarity or dissimlarity matrix of it.

Usage: A = adjgraph(M,n,type);

Example:
```

```
A = adjacency(M,6,1)
A contains the adjacency matrix for the data set. For
each point, the distances to 6 adjacent points are stored.
M is a similarity matrix.

Notes for arguments:

INPUTS:
M: the Euclidean distance matrix of the data set which
   will be used to construct the graph. it must be a
   N by N square matrix.

n: size of neighborhood.
type: if M is similairty matrix, type is 1; otherwise 0.
      Default is 1.

OUTPUTS:
A: sparse symmetric N by N matrix of distances between the
   adjacent points.

adjgraph.m version 1.0
%}

function A = adjgraphbydist(M, TYPE, n, type);

if (nargin < 2)
    disp('ERROR: Arguments WRONG!');
    return;
end

m = size(M,1);
l = size(M,2);

if (l ~= m)
    disp('Error! The distance matrix must be square!');
    return;
end

if (nargin < 3)
    type = 1;
end;
```

```
disp ([num2str(m),' points.']);

A = sparse(m,m);
step = 100;

% Construct the graph by nearest neighbor
% step by step to process large scale data
for i1=1:step:m
    i2 = i1 + step - 1;
    if (i2 > m)
        i2 = m;
    end;
    dt = M(i1:i2,:);
    if (type == 1)
        [Z,I] = sort(dt,2,'descend');
    else
        [Z,I] = sort(dt,2);
    end;

    for i=i1:i2
        if ( mod(i, 500) == 0)
            disp(sprintf('%d points processed.', i));
        end;
        for j=2:n+1
            A(i,I(i-i1+1,j)) = Z(i-i1+1,j);
            A(I(i-i1+1,j),i) = Z(i-i1+1,j);
        end;
    end
end;
```

## B.5 TKEOptions.m

```
%{
TKEOptions setup the parameters of the TKE model.

Usage: options = TKEOptions(varargin)

TKEOptions.m version 1.5
%}

function options = TKEOptions(varargin)
% varargin is not used here.
```

```
options.learnScales = 0;
options.isSpherical = 1;

options.kern = 'rbf';
options.beta = [];

options.kernelReg = 1;
options.variableReg = 1;

options.prior = 'gaussian';
```

## B.6 TKECreate.m

```
%{
TKECreate Create a TKE model.

Usage: model = TKECreate(latentdim, Kgram, options);

TKECreate.m version 1.5
%}

function model = TKECreate(latentdim, Kgram, options)

if size(Kgram, 2) ~= size(Kgram, 1)
    error(['Input K Gram matrix is not a quare matrix']);
end

model.type = 'TKE';

if isstruct(options.prior)
  model.prior = options.prior;
else
  if ~isempty(options.prior)
    model.prior = priorCreate(options.prior);
  end
end

if isfield(options, 'X')
    model.X = options.X;
else
   [U, T] = schur(Kgram);
   mn = min(min(T));
   Y = U*sqrt(T + (mn<0)*abs(mn)*eye(size(T)));
   X = ppcaEmbed(Y, latentdim);
   model.X = X;
end

model.d = latentdim;
model.dataKgram = Kgram;
model.N = size(Kgram, 1);

model.optimiser = options.optimiser;
if isstruct(options.kern)
    model.kern = options.kern;
else
    model.kern = kernCreate(model.X, options.kern);
end

model.kernelReg = options.kernelReg;
model.variableReg = options.variableReg;

% For back constraints
if isfield(options, 'back') & ~isempty(options.back)
  if isstruct(options.back)
    model.back = options.back;
  else
    if ~isempty(options.back)
        if options.back == 'km'
            model.d = inf;
            model.y = NaN;
        else
            model.d = size(options.backOptions.X,2);
            model.y = options.backOptions.X;
        end;
      model.back = modelCreate(options.back, model.d, ...
          model.d, options.backOptions);
    end
  end
  if options.optimiseInitBack
    model.back = mappingOptimise(model.back, model.y,  ...
        model.X);
  end
end
% For back constraints
```

```
initParams = TKEExtractParam(model);
% This forces kernel computation.
model = TKEExpandParam(model, initParams);
```

## B.7 TKEOptimise.m

```
%{
TKEOptimise optimise the TKE model.

model = TKEOptimise(params, model)

TKEOptimise.m version 1.5
%}

function model = TKEOptimise(model, display, iters);

if nargin < 3
  iters = 2000;
  if nargin < 2
    display = 1;
  end
end

params = TKEExtractParam(model);

options = optOptions;
options(2) = 0.0000001;
options(3) = 0.0000001;
if display
  options(1) = 1;
  if length(params) <= 100
    options(9) = 1;
  end
end options(14) = iters;

if isfield(model, 'optimiser')
  optim = str2func(model.optimiser);
else
  optim = str2func('conjgrad');
end
```

```
params = optim('TKEObjective', params,  options, ...
               'TKEGradient', model);

model = TKEExpandParam(model, params);
```

## B.8 TKEExtractParam.m

```
%{
TKEExtractParam Extract a parameter vector from a TKE model.

params = TKEExtractParam(model)

TKEExtractParam.m version 1.5
%}

function params = TKEExtractParam(model)

params = kernExtractParam(model.kern);
% For back constraints
if isfield(model, 'back')
  params = [modelExtractParam(model.back) params];
else
  params = [model.X(:)' params];
end
% For back constraints
```

## B.9 TKEExpandParam.m

```
%{
TKEExpandParam expand parameters for TKE model.

model = TKEExpandParam(model, params)

TKEExpandParam.m version 1.5
%}

function model = TKEExpandParam(model, params)

startVal = 1;
```

```
% For back constraints
if isfield(model, 'back')
  endVal = model.back.numParams;
  model.back = modelExpandParam(model.back, ...
    params(startVal:endVal));
  model.X = modelOut(model.back, model.y);
else
  endVal = model.N * model.q;
  model.X = reshape(params(startVal:endVal), ...
    model.N, model.q);
end;
startVal = endVal + 1;
endVal = endVal + model.kern.nParams;
% For back constraints

model.kern = kernExpandParam(model.kern, ...
    params(startVal:end));
```

## B.10 GenLabels.m

```
%{
Generate labels for KNN relabeling.

Notes for the arguments:

nc: a number, the number of the classes.
ns: a cell, indicates the number in each class. If all the
    classes have the same number of samples, just use one
    cell for that number.
    For example: ns={137,391,208}; or ns={50}; In former
    case, there are 137 samples for 1st class, 391 for 2nd,
    208 for 3rd. In latter case, there are 50 sample in
    each class.

GenLabels.m version 1.0
%}

function labels = GenLabels(nc,ns)

if length(ns) == 1
    loops = nc;
    ls = repmat(ns,1,nc);
```

```
else
    if length(ns) ~= nc
        error('The number of classes is wrong!');
    else
        loops = length(ns);
        ls = ns;
    end;
end start = 1;
for i=1:loops
    labels(start:start+ls{i}-1) = i - 1;
    start = start + ls{i};
end;
labels = labels';
```

## B.11 knnrelabel.m

```
%{
knnrelabel will relabel the data by n nearest neighbour.
The error will be the number of mislabeled objects.

Notes for the arguments:

data: input data vectors; data in rows.
labels: the lables for of data which is a number.
n: size of neighborhood.

knnrelabel.m version 1.0
%}

function err=knnrelabel(data,labels,n)

m = size(data, 1);
ml = size(labels, 1);

if (ml ~= m)
    disp('wrong parameters!');
    return;
end;

s = unique(labels);
newlabels = zeros(size(labels));
```

```
dm = l2dist(data);
for i=1:m
    c = zeros(length(s),1);
    v = dm(i,:);
    [d idx] = sort(v);
    tmp = labels(idx);
    for j=2:m
        [d idx] = max(c);
        if (d >= n)
            newlabels(i) = s(idx);
            break;
        end;
        idx = find(s == tmp(j));
        c(idx) = c(idx) + 1;
    end;
end;
err = length(find(labels ~= newlabels));
```