Given feature vectors $x_i$, $i = 1, \ldots, N$:

1. For each feature vector $x_i$, identify its **k** nearest neighbors $x_{I(1)}, x_{I(2)}, \ldots, x_{I(k)}$;
2. For each feature vector $x_i$ compute weights $w_1, w_2, \ldots, w_k$ minimizing reconstruction error

$$\|x_i - \sum_{j=1}^{k} w_j x_{I(j)}\|, \quad \text{w.r.t} \quad \sum_{j=1}^{k} w_j = 1.$$

This can be done with solving following system of linear equations

$$\underbrace{\begin{pmatrix} (x_i - x_{I(1)})(x_i - x_{I(1)}) & (x_i - x_{I(1)})(x_i - x_{I(2)}) & \ldots & (x_i - x_{I(1)})(x_i - x_{I(k)}) \\ (x_i - x_{I(2)})(x_i - x_{I(1)}) & (x_i - x_{I(2)})(x_i - x_{I(2)}) & \ldots & (x_i - x_{I(2)})(x_i - x_{I(k)}) \\ \vdots & \vdots & \ddots & \vdots \\ (x_i - x_{I(k)})(x_i - x_{I(1)}) & (x_i - x_{I(k)})(x_i - x_{I(2)}) & \ldots & (x_i - x_{I(k)})(x_i - x_{I(k)}) \end{pmatrix}}_{G_i} \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_k \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

In case $k$ is greater than dimension of feature vectors $d$ stated problem is ill-posed. Conventional regularization for this problem is

$$G_i \leftarrow G_i + \epsilon_1 \operatorname{tr}(G_i).$$

Store computed weights into alignment matrix $L$ with

$$L(I_i, I_i) = L(I_i, I_i) + W_i,$$

where $I_i = (i, I(1), \ldots, I(k))$ and $W_I = \begin{bmatrix} 1 & -w \\ -w^T & w^T w \end{bmatrix}$ with $L$ elements initially set to zero.

3. Compute $t$ eigenvectors of $L$ with smallest according eigenvalues (optionally regularize matrix before with $L = L - \epsilon_2 I$). Form embedding feature matrix with computed eigenvectors row-wise:

$$\begin{pmatrix} y_1^1 & y_1^2 & \ldots & y_1^N \\ \vdots & \vdots & \ldots & \vdots \\ y_t^1 & y_t^2 & \ldots & y_t^N \end{pmatrix} \to \underbrace{\left[ \begin{pmatrix} y_1^1 \\ \vdots \\ y_t^1 \end{pmatrix} \begin{pmatrix} y_1^2 \\ \vdots \\ y_t^2 \end{pmatrix} \ldots \begin{pmatrix} y_1^N \\ \vdots \\ y_t^N \end{pmatrix} \right]}_{\text{embedded feature vectors}}$$

Parameters:

- **k** : number of neighbors (typically greater than 10-20)
- **target dim** : embedding dimensionality $t$
- **reconstruction shift** : regularization parameter $\epsilon_1$ for ill-posed weight computation problem (is not used when computing embedding of data with dimensionality greater than k)
- **nullspace shift** : regularization parameter $\epsilon_2$ for the eigenproblem (should be negative and typically has order of magnitude in range $10^{-1} - 10^{-9}$)
- **auto k** : indicates whether automatic k parameter calculation in range (k, max k) should be used
- **max k** : maximal possible value of k parameter

src/lle.py

```
import modshogun as sg
import data

# load data
feature_matrix = data.swissroll()
# create features instance
features = sg.RealFeatures(feature_matrix)

# create Locally Linear Embedding converter instance
converter = sg.LocallyLinearEmbedding()

# set target dimensionality
converter.set_target_dim(2)
# set number of neighbors
converter.set_k(10)
# set number of threads
converter.parallel.set_num_threads(2)
# set reconstruction shift (optional)
converter.set_reconstruction_shift(1e-3)
# set nullspace shift (optional)
converter.set_nullspace_shift(-1e-6)
# check whether arpack is used
if converter.get_use_arpack():
        print 'ARPACK is used'
else:
        print 'LAPACK is used'

# compute embedding with Locally Linear Embedding method
embedding_first = converter.embed(features)

# enable auto k search in range of (10,100)
# based on reconstruction error
converter.set_k(50)
converter.set_max_k(100)
converter.set_auto_k(True)

# compute embedding with Locally Linear Embedding method
embedding_second = converter.embed(features)
```

Given feature vectors $x_i$, $i = 1, \dots, N$:

1. For each feature vector $x_i$, identify its **k** nearest neighbors $x_{I(1)}, x_{I(2)}, \dots, x_{I(k)}$;
2. For each feature vector $x_i$,
   - Form local feature matrix
   $$\begin{pmatrix} x_{I(1)} & x_{I(2)} & \dots & X_{I(k)} \end{pmatrix}$$
   and subtract mean feature vector $\overline{x} = \frac{1}{k}\sum_{i=1}^{k} x_{I(i)}$ from its columns.
   - Compute $t$ right singular vectors of modified local feature matrix
   $$\begin{pmatrix} x_{I(1)} - \overline{x} & x_{I(2)} - \overline{x} & \dots & X_{I(k)} - \overline{x} \end{pmatrix}$$
   and store into matrix V. Form matrix
   $$Y = \begin{pmatrix} 1 & v_1 & v_2 & \dots & v_t & v_1 \cdot v_1 & \dots \end{pmatrix}$$
   and QR factorize it: $Y = QR$.
   - Normalize columns of the matrix $Q$ (starting from $1 + t$) with dividing each column by $w$, $w_i = \sum_j Q_{j,1+t+i}$.
   - Compute $PP^T$, where $P = Q(:, 1 + t :)$ and store result into $L(I_i, I_i)$ with L initially set to zero.
3. Compute $t$ eigenvectors of $L$ with smallest according eigenvalues (optionally regularize matrix before with $L = L - \epsilon I$). Form embedding feature matrix with computed eigenvectors row-wise:
$$\begin{pmatrix} y_1^1 & y_1^2 & \dots & y_1^N \\ \vdots & \vdots & \dots & \vdots \\ y_t^1 & y_t^2 & \dots & y_t^N \end{pmatrix} \to \underbrace{\left[ \begin{pmatrix} y_1^1 \\ \vdots \\ y_t^1 \end{pmatrix} \begin{pmatrix} y_1^2 \\ \vdots \\ y_t^2 \end{pmatrix} \dots \begin{pmatrix} y_1^N \\ \vdots \\ y_t^N \end{pmatrix} \right]}_{\text{embedded feature vectors}}$$

Parameters:

- **k** : number of neighbors $k$ (typically greater than 10-20)
- **target dim** : embedding dimensionality $t$
- **nullspace shift** : regularization parameter $\epsilon$ for eigenproblem (should be negative and typically has order of magnitude in range $10^{-1} - 10^{-9}$)

src/hlle.py

```python
import modshogun as sg
import data

# load data
feature_matrix = data.swissroll()
# create features instance
features = sg.RealFeatures(feature_matrix)

# create Hessian Locally Linear Embedding converter instance
converter = sg.HessianLocallyLinearEmbedding()

# set target dimensionality
converter.set_target_dim(2)
# set number of neighbors
converter.set_k(10)
# set number of threads
converter.parallel.set_num_threads(2)
# set nullspace shift (optional)
converter.set_nullspace_shift(-1e-6)

# compute embedding with Hessian Locally Linear Embedding method
embedding = converter.embed(features)
```

Given feature vectors $x_i$, $i = 1, \dots, N$ and kernel function $k(x,y) = \langle \phi(x), \phi(y) \rangle$:

1. For each feature vector $x_i$, $i = 1, \dots, N$, identify its **k** nearest neighbors $x_{I(1)}, x_{I(2)}, \dots, x_{I(k)}$ with kernel distance:
$$d(x,y) = k(x,x) + k(y,y) - 2k(x,y).$$

2. For each feature vector $x_i$ compute weights $w_1, w_2, \dots, w_k$ minimizing reconstruction error of mapped feature vectors:
$$\|\phi(x_i) - \sum_{j=1}^{k} w_j \phi(x_{I(j)})\|, \quad \text{w.r.t} \quad \sum_{j=1}^{k} w_j = 1.$$
This can be done with solving following system of linear equations
$$\underbrace{\begin{pmatrix} g_{1,1} & g_{1,2} & \cdots & g_{1,k} \\ g_{2,1} & g_{2,2} & \cdots & g_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k,1} & g_{k,2} & \cdots & g_{k,k} \end{pmatrix}}_{(g_{m,n})} \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_k \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix},$$
where $g_{m,n} = k(x_i, x_i) - k(x_i, x_{I(n)}) - k(x_{I(m)}, x_i) + k(x_{I(m)}, x_{I(n)})$. Store computed weights into the alignment matrix $L$ with
$$L(I_i, I_i) = L(I_i, I_i) + W_i,$$
where $I_i = (i, I(1), \dots, I(k))$ and $W_I = \begin{bmatrix} 1 & -w \\ -w^T & w^T w \end{bmatrix}$ with $L$ elements initially set to zero.

3. Compute $t$ eigenvectors $y_1, \dots, y_t$ of $L$ with smallest according eigenvalues (optionally regularize matrix with $L - \epsilon_2 I$). Form embedding feature matrix with computed eigenvectors row-wise:
$$\begin{pmatrix} y_1^1 & y_1^2 & \cdots & y_1^N \\ \vdots & \vdots & \cdots & \vdots \\ y_t^1 & y_t^2 & \cdots & y_t^N \end{pmatrix} \to \underbrace{\left[ \begin{pmatrix} y_1^1 \\ \vdots \\ y_t^1 \end{pmatrix} \begin{pmatrix} y_1^2 \\ \vdots \\ y_t^2 \end{pmatrix} \cdots \begin{pmatrix} y_1^N \\ \vdots \\ y_t^N \end{pmatrix} \right]}_{\text{embedded feature vectors}}$$

Parameters:

- **k** : number of neighbors (typically greater than 10-20)
- **target dim** : embedding dimensionality $t$
- **reconstruction shift** : regularization parameter $\epsilon_1$ for ill-posed weight computation problem (is not used when computing embedding of data with dimensionality greater than k)
- **nullspace shift** : regularization parameter $\epsilon_2$ for eigenproblem (should be negative and typically has order of magnitude in range $10^{-1} - 10^{-9}$)
- **auto k** : indicates whether automatic k parameter calculation in range (k, max k) should be used
- **max k** : maximal possible value of k parameter
- **kernel** : kernel instance to be used

src/klle.py

```python
import modshogun as sg
import data
import numpy as np

# load data
feature_matrix = data.swissroll()
# create features instance
features = sg.RealFeatures(feature_matrix)

# create Kernel Locally Linear Embedding converter instance
converter = sg.KernelLocallyLinearEmbedding()

# set target dimensionality
converter.set_target_dim(2)
# set number of neighbors
converter.set_k(10)
# set number of threads
converter.parallel.set_num_threads(2)
# set nullspace shift (optional)
converter.set_nullspace_shift(-1e-6)

# create Gaussian kernel instance
kernel = sg.GaussianKernel(100,10.0)
# enable converter instance to use created kernel instance
converter.set_kernel(kernel)

# compute embedding with Kernel Locally Linear Embedding method
embedding = converter.embed(features)

# compute linear kernel matrix
kernel_matrix = np.dot(feature_matrix.T,feature_matrix)
# create Custom Kernel instance
custom_kernel = sg.CustomKernel(kernel_matrix)
# construct embedding based on created kernel
kernel_embedding = converter.embed_kernel(custom_kernel)
```

Given feature vectors $x_i$, $i = 1, \ldots, N$:

1. For each feature vector $x_i$, identify its **k** nearest neighbors $x_{I(1)}, x_{I(2)}, \ldots, x_{I(k)}$;
2. For each feature vector $x_i$ estimate local tangent coordinates with following steps:
   - Form local feature matrix
   $$\begin{bmatrix} x_{I(1)} & x_{I(2)} & \ldots & x_{I(k)} \end{bmatrix}$$
   and subtract mean feature vector $\overline{x} = \frac{1}{k} \sum_{i=1}^{k} x_{I(i)}$ from its columns.
   - Compute right singular vectors of modified local feature matrix
   $$\begin{bmatrix} x_{I(1)} - \overline{x} & x_{I(2)} - \overline{x} & \ldots & x_{I(k)} - \overline{x} \end{bmatrix}$$
   and store into matrix $V$. Form matrix $G = \left[\frac{1}{\sqrt{k}}; V\right]$.
   - Form part of the alignment matrix $L$ with
   $$L(I_i, I_i) = L(I_i, I_i) + E - GG',$$
   where indexes $I_i = (i, I(1), I(2), \ldots, I(k))$ and E is an identity matrix.
3. Compute $t$ eigenvectors of $L$ with smallest according eigenvalues. Optionally regularize matrix with
   $$L - \epsilon I.$$
   Form embedding feature matrix with computed eigenvectors row-wise:
   $$\begin{pmatrix} y_1^1 & y_1^2 & \ldots & y_1^N \\ \vdots & \vdots & \ldots & \vdots \\ y_t^1 & y_t^2 & \ldots & y_t^N \end{pmatrix} \rightarrow \underbrace{\left[ \begin{pmatrix} y_1^1 \\ \vdots \\ y_t^1 \end{pmatrix} \begin{pmatrix} y_1^2 \\ \vdots \\ y_t^2 \end{pmatrix} \ldots \begin{pmatrix} y_1^N \\ \vdots \\ y_t^N \end{pmatrix} \right]}_{\text{embedded feature vectors}}$$

Parameters:

- **k** : number of neighbors (typically greater than 10-20)
- **target dim** : embedding dimensionality
- **nullspace shift** : regularization parameter $\epsilon$ of the eigenproblem (should be negative and typically has order of magnitude in range $10^{-1} - 10^{-9}$)

src/ltsa.py

```python
import modshogun as sg
import data

# load data
feature_matrix = data.swissroll()
# create features instance
features = sg.RealFeatures(feature_matrix)

# create Local Tangent Space Alignment converter instance
converter = sg.LocalTangentSpaceAlignment()

# set target dimensionality
converter.set_target_dim(2)
# set number of neighbors
converter.set_k(10)
# set number of threads
converter.parallel.set_num_threads(2)
# set nullspace shift (optional)
converter.set_nullspace_shift(-1e-6)

# compute embedding with Local Tangent Space Alignment method
embedding = converter.embed(features)
```

Given feature vectors $x_i$, $i = 1, \dots, N$ and kernel function $k(x, y) = \langle \phi(x), \phi(y) \rangle$:

1. For each feature vector $x_i$, identify its **k** nearest neighbors $x_{I(1)}, x_{I(2)}, \dots, x_{I(k)}$ with kernel distance
$$d(x, y) = k(x, x) + k(y, y) - 2k(x, y).$$

2. For each feature vector $x_i$ estimate local tangent coordinates with following steps:

   - Form local Gram matrix
$$(g_{m,n}) = \begin{pmatrix} k(x_{I(1)}, x_{I(1)}) & k(x_{I(1)}, x_{I(2)}) & \dots & k(x_{I(1)}, x_{I(k)}) \\ k(x_{I(2)}, x_{I(1)}) & k(x_{I(2)}, x_{I(2)}) & \dots & k(x_{I(2)}, x_{I(k)}) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_{I(k)}, x_{I(1)}) & k(x_{I(k)}, x_{I(2)}) & \dots & k(x_{I(k)}, x_{I(k)}) \end{pmatrix}$$
   and center it with subtracting column means, row means and adding grand mean:
$$g_{m,n} = g_{m,n} - \frac{1}{k}\sum_{p=1}^{k} g_{m,p} - \frac{1}{k}\sum_{q=1}^{k} g_{q,n} + \frac{1}{k^2}\sum_{q=1}^{k}\sum_{p=1}^{k} g_{q,p}.$$

   - Compute $t$ eigenvectors with largest eigenvalues of the matrix $(g_{m,n})$, store these eigenvectors into the matrix $V$ and form matrix $G = [\frac{1}{\sqrt{k}}; V]$.

   - Form part of the alignment matrix $L$ with
$$L(I_i, I_i) = L(I_i, I_i) + E - GG',$$
   where $I_i = (i, I(1), I(2), \dots, I(k))$.

3. Compute $t$ eigenvectors of $L$ with smallest according eigenvalues. Optionally regularize matrix with
$$L - \epsilon I.$$
   Form embedding feature matrix with computed eigenvectors row-wise:
$$\begin{pmatrix} y_1^1 & y_1^2 & \dots & y_1^N \\ \vdots & \vdots & \dots & \vdots \\ y_t^1 & y_t^2 & \dots & y_t^N \end{pmatrix} \to \underbrace{\left[ \begin{pmatrix} y_1^1 \\ \vdots \\ y_t^1 \end{pmatrix} \begin{pmatrix} y_1^2 \\ \vdots \\ y_t^2 \end{pmatrix} \dots \begin{pmatrix} y_1^N \\ \vdots \\ y_t^N \end{pmatrix} \right]}_{\text{embedded feature vectors}}$$

Parameters:

- **k** : number of neighbors (typically greater than 10-20)
- **target dim** : embedding dimensionality
- **nullspace shift** : regularization parameter for eigenproblem (should be negative and typically has order of magnitude in range $10^{-1} - 10^{-9}$)
- **kernel** : Kernel instance to be used

src/kltsa.py

```python
import modshogun as sg
import data
import numpy as np

# load data
feature_matrix = data.swissroll()
# create features instance
features = sg.RealFeatures(feature_matrix)

# create Kernel Local Tangent Space Alignment converter instance
converter = sg.KernelLocalTangentSpaceAlignment()

# set target dimensionality
converter.set_target_dim(2)
# set number of neighbors
converter.set_k(10)
# set number of threads
converter.parallel.set_num_threads(2)
# set nullspace shift (optional)
converter.set_nullspace_shift(-1e-6)

# create Sigmoid kernel instance
kernel = sg.SigmoidKernel(100,10.0,1.0)
# enable converter instance to use created kernel instance
converter.set_kernel(kernel)

# compute embedding with Kernel Local Tangent Space Alignment method
embedding = converter.embed(features)

# compute linear kernel matrix
kernel_matrix = np.dot(feature_matrix.T,feature_matrix)
# create Custom Kernel instance
custom_kernel = sg.CustomKernel(kernel_matrix)
# construct embedding based on created kernel
kernel_embedding = converter.embed_kernel(custom_kernel)
```

Given feature vectors $x_i$, $i = 1, \ldots, N$ and feature matrix

$$X = [x_1, x_2, \ldots, x_N] :$$

1. For each feature vector $x_i$, identify its **k** nearest neighbors $x_{I(1)}, x_{I(2)}, \ldots, x_{I(k)}$;
2. For each feature vector $x_i$ compute weights $w_1, w_2, \ldots, w_k$ minimizing reconstruction error

$$\|x_i - \sum_{j=1}^{k} w_j x_{I(j)}\|, \quad \text{w.r.t} \quad \sum_{j=1}^{k} w_j = 1.$$

This can be done with solving following system of linear equations

$$\underbrace{\begin{pmatrix} (x_i - x_{I(1)})(x_i - x_{I(1)}) & (x_i - x_{I(1)})(x_i - x_{I(2)}) & \ldots & (x_i - x_{I(1)})(x_i - x_{I(k)}) \\ (x_i - x_{I(2)})(x_i - x_{I(1)}) & (x_i - x_{I(2)})(x_i - x_{I(2)}) & \ldots & (x_i - x_{I(2)})(x_i - x_{I(k)}) \\ \vdots & \vdots & \ddots & \vdots \\ (x_i - x_{I(k)})(x_i - x_{I(1)}) & (x_i - x_{I(k)})(x_i - x_{I(2)}) & \ldots & (x_i - x_{I(k)})(x_i - x_{I(k)}) \end{pmatrix}}_{G_i} \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_k \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

In case $k$ is greater than dimension of feature vectors $d$ stated problem is ill-posed. Conventional regularization for this problem is

$$G_i \leftarrow G_i + \epsilon_1 \operatorname{tr}(G_i).$$

Store computed weights into alignment matrix $L$ with

$$L(I_i, I_i) = L(I_i, I_i) + W_i,$$

where $I_i = (i, I(1), \ldots, I(k))$ and $W_I = \begin{bmatrix} 1 & -w \\ -w^T & w^T w \end{bmatrix}$ with $L$ elements initially set to zero.

3. Consider generalized eigenproblem

$$XLX^T v = \lambda XX^T v.$$

Compute eigenvectors $v_1, \ldots, v_t$ with smallest according eigenvalues and form transformation matrix

$$V = (v_1, \ldots, v_t).$$

Form embedding feature matrix with

$$V^T X = \begin{pmatrix} y_1^1 & y_1^2 & \ldots & y_1^N \\ \vdots & \vdots & \ldots & \vdots \\ y_t^1 & y_t^2 & \ldots & y_t^N \end{pmatrix} \rightarrow \underbrace{\left[ \begin{pmatrix} y_1^1 \\ \vdots \\ y_t^1 \end{pmatrix} \begin{pmatrix} y_1^2 \\ \vdots \\ y_t^2 \end{pmatrix} \ldots \begin{pmatrix} y_1^N \\ \vdots \\ y_t^N \end{pmatrix} \right]}_{\text{embedded feature vectors}}$$

Parameters:

- **k** : number of neighbors (typically greater than 10-20)
- **target dim** : embedding dimensionality

src/npe.py

```python
import modshogun as sg
import data

# load data
feature_matrix = data.swissroll()
# create features instance
features = sg.RealFeatures(feature_matrix)

# create Neighborhood Preserving Embedding converter instance
converter = sg.NeighborhoodPreservingEmbedding()

# set target dimensionality
converter.set_target_dim(2)
# set number of neighbors
converter.set_k(10)
# set number of threads
converter.parallel.set_num_threads(2)
# set nullspace shift (optional)
converter.set_nullspace_shift(-1e-6)

# compute embedding with Neighborhood Preserving Projections method
embedding = converter.embed(features)
```

Given feature vectors $x_i$, $i = 1, \dots, N$ and feature matrix

$$X = [x_1, x_2, \dots, x_N]:$$

1. For each feature vector $x_i$, identify its **k** nearest neighbors $x_{I(1)}, x_{I(2)}, \dots, x_{I(k)}$;
2. For each feature vector $x_i$ estimate local tangent coordinates with following steps:
   - Form local feature matrix
     $$\begin{bmatrix} x_{I(1)} & x_{I(2)} & \dots & x_{I(k)} \end{bmatrix}$$
     and subtract mean feature vector $\overline{x} = \frac{1}{k}\sum_{i=1}^{k} x_{I(i)}$ from its columns.
   - Compute right singular vectors of modified local feature matrix
     $$\begin{bmatrix} x_{I(1)} - \overline{x} & x_{I(2)} - \overline{x} & \dots & x_{I(k)} - \overline{x} \end{bmatrix}$$
     and store into matrix $V$. Form matrix $G = \left[\frac{1}{\sqrt{k}}; V\right]$.
   - Form part of the alignment matrix $L$ with
     $$L(I_i, I_i) = L(I_i, I_i) + E - GG',$$
     where indexes $I_i = (i, I(1), I(2), \dots, I(k))$ and E is an identity matrix.
3. Center matrix $L$ with
   $$L_{m,n} = L_{m,n} - \frac{1}{N}\sum_{q=1}^{N} L_{q,n} - \frac{1}{N}\sum_{p=1}^{N} L_{m,p} + \frac{1}{N^2}\sum_{q=1}^{N}\sum_{p=1}^{N} L_{q,p}$$
   and compute
   $$Q = XLX^T.$$
   Subtract mean feature vector $\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$ from each row of the feature matrix $X$:
   $$X = \begin{pmatrix} x_1 - \overline{x} & \dots & x_N - \overline{x} \end{pmatrix}.$$
4. Consider generalized eigenproblem
   $$Qv = \lambda XX^T v.$$
   Compute eigenvectors $v_1, \dots, v_t$ with smallest according eigenvalues and form transformation matrix
   $$V = (v_1, \dots, v_t).$$
   Form embedding feature matrix with
   $$V^T X = \begin{pmatrix} y_1^1 & y_1^2 & \dots & y_1^N \\ \vdots & \vdots & \dots & \vdots \\ y_t^1 & y_t^2 & \dots & y_t^N \end{pmatrix} \rightarrow \underbrace{\left[\begin{pmatrix} y_1^1 \\ \vdots \\ y_t^1 \end{pmatrix} \begin{pmatrix} y_1^2 \\ \vdots \\ y_t^2 \end{pmatrix} \dots \begin{pmatrix} y_1^N \\ \vdots \\ y_t^N \end{pmatrix}\right]}_{\text{embedded feature vectors}}$$

Parameters:

- **k** : number of neighbors (typically greater than 10-20)
- **target dim** : embedding dimensionality

src/lltsa.py

```python
import modshogun as sg
import dataloader as loader

# load data
feature_matrix = loader.load('mnist.mat')
# create features instance
features = sg.RealFeatures(feature_matrix)

# create Linear Local Tangent Space Alignment converter instance
converter = sg.LinearLocalTangentSpaceAlignment()

# set target dimensionality
converter.set_target_dim(2)
# set number of neighbors
converter.set_k(10)
# set number of threads
converter.parallel.set_num_threads(2)
# set nullspace shift (optional)
converter.set_nullspace_shift(-1e-6)

# compute embedding with Linear Local Tangent Space Alignment method
embedding = converter.embed(features)
```

Given feature vectors $x_i$, $i = 1, \dots, N$:

1. Construct the adjacency graph: put an edge between $i$th and $j$th feature vectors if $i$th vector is among $k$ nearest neighbors of $j$th vector or vice versa. Construct weight matrix $W$ with

$$W_{i,j} = \begin{cases} \exp\left\{-\frac{d^2(x_i,x_j)}{\tau}\right\}, & \text{if there is an edge between } i\text{th and } j\text{th vectors,} \\ 0, & \text{else.} \end{cases}$$

2. Compute matrices $D$ ($D_{i,i} = \sum_j W_{j,i}$) and $L = D - W$. Consider generalized eigenproblem

$$Lv = \lambda Dv,$$

compute eigenvectors $v_1, \dots, v_t$ with smallest according eigenvalues. Form embedding feature matrix with computed eigenvectors row-wise:

$$\begin{pmatrix} v_1^1 & v_1^2 & \dots & v_1^N \\ \vdots & \vdots & \dots & \vdots \\ v_t^1 & v_t^2 & \dots & v_t^N \end{pmatrix} \to \underbrace{\left[ \begin{pmatrix} v_1^1 \\ \vdots \\ v_t^1 \end{pmatrix} \begin{pmatrix} v_1^2 \\ \vdots \\ v_t^2 \end{pmatrix} \dots \begin{pmatrix} v_1^N \\ \vdots \\ v_t^N \end{pmatrix} \right]}_{\text{embedded feature vectors}}$$

Parameters:

- **k** : number of neighbors $k$ (typically greater than 10-20)
- **tau** : heat distribution multiplier $\tau$
- **target dim** : embedding dimensionality $t$

src/la.py

```python
import modshogun as sg
import data
import numpy as np

# load data
feature_matrix = data.swissroll()
# create features instance
features = sg.RealFeatures(feature_matrix)

# create Laplacian Eigenmaps converter instance
converter = sg.LaplacianEigenmaps()

# set target dimensionality
converter.set_target_dim(2)
# set number of neighbors
converter.set_k(20)
# set tau multiplier
converter.set_tau(1.0)

# compute embedding with Laplacian Eigenmaps method
embedding = converter.embed(features)

# compute cosine distance matrix 'manually'
N = features.get_num_vectors()
distance_matrix = np.zeros((N,N))
for i in range(N):
        for j in range(N):
                distance_matrix[i,j] = \
                  np.linalg.norm(feature_matrix[:,i]-feature_matrix[:,j],2)
# create custom distance instance
distance = sg.CustomDistance(distance_matrix)
# construct embedding based on created distance
converter.embed_distance(distance)
```

Given feature vectors $x_i$, $i = 1, \dots, N$ and feature matrix

$$X = [x_1, x_2, \dots, x_N]:$$

1. Construct the adjacency graph: put an edge between $i$th and $j$th feature vectors if $i$th vector is among $k$ nearest neighbors of $j$th vector or vice versa. Construct weight matrix $W$ with

$$W_{i,j} = \begin{cases} \exp\left\{-\frac{d^2(x_i,x_j)}{\tau}\right\}, & \text{if there is an edge between } i\text{th and } j\text{th vectors,} \\ 0, & \text{else.} \end{cases}$$

2. Compute matrices $D$ ($D_{i,i} = \sum_j W_{j,i}$) and $L = D - W$. Consider generalized eigenproblem

$$XLX^T v = \lambda XDX^T v,$$

   compute eigenvectors $v_1, \dots, v_t$ with smallest according eigenvalues. Form embedding feature matrix with computed eigenvectors row-wise:

$$\begin{pmatrix} v_1^1 & v_1^2 & \dots & v_1^N \\ \vdots & \vdots & \dots & \vdots \\ v_t^1 & v_t^2 & \dots & v_t^N \end{pmatrix} \rightarrow \underbrace{\left[ \begin{pmatrix} v_1^1 \\ \vdots \\ v_t^1 \end{pmatrix} \begin{pmatrix} v_1^2 \\ \vdots \\ v_t^2 \end{pmatrix} \dots \begin{pmatrix} v_1^N \\ \vdots \\ v_t^N \end{pmatrix} \right]}_{\text{embedded feature vectors}}$$

Parameters:

- **k** : number of neighbors $k$ (typically greater than 10-20)
- **tau** : heat distribution multiplier $\tau$
- **target dim** : embedding dimensionality $t$

src/lpp.py

```
import modshogun as sg
import data

# load data
feature_matrix = data.swissroll()
# create features instance
features = sg.RealFeatures(feature_matrix)

# create Locality Preserving Projections converter instance
converter = sg.LocalityPreservingProjections()

# set target dimensionality
converter.set_target_dim(2)
# set number of neighbors
converter.set_k(10)
# set number of threads
converter.parallel.set_num_threads(2)

# compute embedding with Locality Preserving Projections method
embedding = converter.embed(features)
```

Given feature vectors $x_i$, $i = 1, \dots, N$ and kernel function $k(x_i, x_j)$:

1. Construct kernel matrix

$$K = \begin{pmatrix} k(x_1, x_1) & k(x_1, x_2) & \dots & k(x_1, x_N) \\ k(x_2, x_1) & k(x_2, x_2) & \dots & k(x_2, x_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_N, x_1) & k(x_2, x_N) & \dots & k(xNn, x_N) \end{pmatrix}$$

   and compute column-sum vector $p$, $p_i = \sum_{j=1}^{N} k_{j,i}$.

2. Modify kernel matrix with following operations:

$$k_{i,j} = \frac{k_{i,j}}{(p_i p_j)^q},$$

   recompute $p_i = \sum j = 1^N k_{j,i}$ and

$$k_{i,j} = \frac{k_{i,j}}{\sqrt{p_i p_j}}.$$

3. Compute $t$ eigenvectors $y_1, \dots, y_t$ of the matrix $K^T K$ with largest according eigenvalues $\lambda_1, \dots, \lambda_t$. Form embedding feature matrix with computed eigenvectors row-wise:

$$\begin{pmatrix} y_1^1 & y_1^2 & \dots & y_1^N \\ \vdots & \vdots & \dots & \vdots \\ y_t^1 & y_t^2 & \dots & y_t^N \end{pmatrix} \rightarrow \underbrace{\left[ \begin{pmatrix} y_1^1 \\ \vdots \\ y_t^1 \end{pmatrix} \begin{pmatrix} y_1^2 \\ \vdots \\ y_t^2 \end{pmatrix} \dots \begin{pmatrix} y_1^N \\ \vdots \\ y_t^N \end{pmatrix} \right]}_{\text{embedded feature vectors}}$$

Parameters:

- **t** : number of time-steps $q$
- **target dim** : embedding dimensionality $t$

src/dm.py

```
import modshogun as sg
import data
import numpy as np

# load data
feature_matrix = data.swissroll()
# create features instance
features = sg.RealFeatures(feature_matrix)

# create Diffusion Maps converter instance
converter = sg.DiffusionMaps()

# set target dimensionality
converter.set_target_dim(2)
# set number of time-steps
converter.set_t(2)

# create Gaussian kernel instance
kernel = sg.GaussianKernel(100,10.0)
# enable converter instance to use created kernel instance
converter.set_kernel(kernel)

# compute embedding with Diffusion Maps method
embedding = converter.embed(features)

# compute linear kernel matrix
kernel_matrix = np.dot(feature_matrix.T,feature_matrix)
# create Custom Kernel instance
custom_kernel = sg.CustomKernel(kernel_matrix)
# construct embedding based on created kernel
kernel_embedding = converter.embed_kernel(custom_kernel)
```

Given feature vectors $x_i$, $i = 1, \ldots, N$ and distance function $d(x_i, x_j)$:

1. Compute distance matrix

$$(d_{i,j}) = \begin{pmatrix} d(x_1,x_1) & d(x_1,x_2) & \ldots & d(x_1,x_N) \\ d(x_2,x_1) & d(x_2,x_2) & \ldots & d(x_2,x_N) \\ \vdots & \vdots & \ddots & \vdots \\ d(x_N,x_1) & d(x_N,x_2) & \ldots & d(x_N,x_N) \end{pmatrix}$$

2. Square distances and center matrix with

$$c(x_i,x_j) = d^2(x_i,x_j) - \frac{1}{N}\sum_{q=1}^{N} d^2(x_q,x_j) - \frac{1}{N}\sum_{p=1}^{N} d^2(x_i,x_p) + \frac{1}{N^2}\sum_{q=1}^{N}\sum_{p=1}^{N} d^2(x_q,x_p).$$

3. Compute $t$ eigenvectors $y_1, \ldots, y_t$ of the matrix $C = \big(c(x_i,x_j)\big)_{i,j}$ with largest according eigenvalues $\lambda_1, \ldots, \lambda_t$. Form embedding feature matrix with computed eigenvectors row-wise:

$$\begin{pmatrix} y_1^1 & y_1^2 & \cdots & y_1^N \\ \vdots & \vdots & \ldots & \vdots \\ y_t^1 & y_t^2 & \cdots & y_t^N \end{pmatrix} \to \underbrace{\left[\begin{pmatrix} y_1^1 \\ \vdots \\ y_t^1 \end{pmatrix} \begin{pmatrix} y_1^2 \\ \vdots \\ y_t^2 \end{pmatrix} \cdots \begin{pmatrix} y_1^N \\ \vdots \\ y_t^N \end{pmatrix}\right]}_{\text{embedded feature vectors}}$$

Parameters:

- **target dim** : embedding dimensionality
- **landmark** : indicates whether landmark approximation should be used or not
- **num landmarks** : number of landmark vectors $l$ (typically 20-50% of total number of vectors)

src/mds.py

```python
import modshogun as sg
import data
import numpy as np

# load data
feature_matrix = data.swissroll()
# create features instance
features = sg.RealFeatures(feature_matrix)

# create Multidimensional Scaling converter instance
converter = sg.MultidimensionalScaling()

# set target dimensionality
converter.set_target_dim(2)

# compute embedding with Multidimensional Scaling method
embedding = converter.embed(features)

# enable landmark approximation
converter.set_landmark(True)
# set number of landmarks
converter.set_landmark_number(100)
# set number of threads
converter.parallel.set_num_threads(2)
# compute approximate embedding
approx_embedding = converter.embed(features)
# disable landmark approximation
converter.set_landmark(False)

# compute cosine distance matrix 'manually'
N = features.get_num_vectors()
distance_matrix = np.zeros((N,N))
for i in range(N):
	for j in range(N):
		distance_matrix[i,j] = \
		  np.cos(np.linalg.norm(feature_matrix[:,i]-feature_matrix[:,j],2))
# create custom distance instance
distance = sg.CustomDistance(distance_matrix)
# construct embedding based on created distance
converter.embed_distance(distance)
```

Given feature vectors $x_i$, $i = 1, \dots, N$ and distance function $d(x_i, x_j)$:

1. Compute distance matrix

$$(d_{i,j}) = \begin{pmatrix} d(x_1, x_1) & d(x_1, x_2) & \dots & d(x_1, x_N) \\ d(x_2, x_1) & d(x_2, x_2) & \dots & d(x_2, x_N) \\ \vdots & \vdots & \ddots & \vdots \\ d(x_N, x_1) & d(x_N, x_2) & \dots & d(x_N, x_N) \end{pmatrix}$$

2. For each feature vector $x_i$ identify its $k$ nearest neighbors and set distances to non-neighbors with $+\infty$. Compute shortest paths based on computed distance matrix with Dijkstra algorithm.

3. Square distances and center matrix with

$$c(x_i, x_j) = d^2(x_i, x_j) - \frac{1}{N}\sum_{q=1}^{N} d^2(x_q, x_j) - \frac{1}{N}\sum_{p=1}^{N} d^2(x_i, x_p) + \frac{1}{N^2}\sum_{q=1}^{N}\sum_{p=1}^{N} d^2(x_q, x_p).$$

4. Compute $t$ eigenvectors $y_1, \dots, y_t$ of the matrix $C = \big(c(x_i, x_j)\big)_{i,j}$ with largest according eigenvalues $\lambda_1, \dots, \lambda_t$. Form embedding feature matrix with computed eigenvectors row-wise:

$$\begin{pmatrix} y_1^1 & y_1^2 & \dots & y_1^N \\ \vdots & \vdots & \dots & \vdots \\ y_t^1 & y_t^2 & \dots & y_t^N \end{pmatrix} \to \underbrace{\left[ \begin{pmatrix} y_1^1 \\ \vdots \\ y_t^1 \end{pmatrix} \begin{pmatrix} y_1^2 \\ \vdots \\ y_t^2 \end{pmatrix} \dots \begin{pmatrix} y_1^N \\ \vdots \\ y_t^N \end{pmatrix} \right]}_{\text{embedded feature vectors}}$$

Parameters:

- **k** : number of neighbors (typically greater than 10-20, sometimes small values lead to unconnected neighborhood graph)
- **target dim** : embedding dimensionality
- **landmark** : indicates whether landmark approximation should be used or not
- **num landmarks** : number of landmark vectors $l$ (typically 20-50% of total number of vectors)

src/isomap.py

```
import modshogun as sg
import data
import numpy as np

# load data
feature_matrix = data.swissroll()
# create features instance
features = sg.RealFeatures(feature_matrix)

# create Isomap converter instance
converter = sg.Isomap()

# set target dimensionality
converter.set_target_dim(2)

# compute embedding with Isomap method
embedding = converter.embed(features)

# enable landmark approximation
converter.set_landmark(True)
# set number of landmarks
converter.set_landmark_number(100)
# set number of threads
converter.parallel.set_num_threads(2)
# compute approximate embedding
approx_embedding = converter.embed(features)
# disable landmark approximation
converter.set_landmark(False)

# compute cosine distance matrix 'manually'
N = features.get_num_vectors()
distance_matrix = np.zeros((N,N))
for i in range(N):
        for j in range(N):
                distance_matrix[i,j] = \
                  np.cos(np.linalg.norm(feature_matrix[:,i]-feature_matrix[:,j],2))
# create custom distance instance
distance = sg.CustomDistance(distance_matrix)
# construct embedding based on created distance
converter.embed_distance(distance)
```