计算机应用数学课程论文-Hessian LLE

李新锋(11224014)

May 24, 2013

1 引言

Hessian LLE [1] 描述了一种将高维欧式空间流形M上的离散数据点，降到低维空间的方法。该方法从一个局部等距的概念框架中导出。在局部等距中流形M 被看做是欧式空间的一个黎曼子流形，对低维欧式空间连通子集也是局部等距的。由于该方法并不要求连通子集是凸集，因而该框架能够处理的情况比ISOMAP 方法更广泛。理论框架主要是求解方程$\mathcal{H}(f) = \int_M ||H_f(m)||_F^2$ 其中定义f:$M \to \mathcal{R}$, H_f表示f的Hessian矩阵, $\mathcal{H}f$对流形M上的Hessian矩阵的Frobenius 范数求均值。为了定义Hessian，我们使用在流形M切平面上的正交坐标系。观察到，如果流形M确实是局部等距于低维欧式空间的一个连通子集，那么$\mathcal{H}f$有一个$(d+1)$维零空间，该零空间由一个常量函数和一个由原始等距坐标张成的d 维函数空间。因此等距坐标可以被恢复到线性等距。该方法可以看做是是LLE方法的修正，其理论框架可以看做是Laplacian eigenmaps框架的修正，其中将原来的Laplacian替换为基于Hessian的二次形式。

2 方法概述

LLE方法假设数据在流形M上，并且流形M被看做是周围欧式空间的黎曼子流形，与低维欧式空间的一个凸子集是全局等距的。然而存在不是全局等距的情况－－部分等距，因而凸子集的限制是不恰当的。HLLE主要解决部分等距的，非凸集的情况。

2.1 数学描述

假设有一个低维参数空间$\Theta \subset \mathcal{R}^d$ 和一个平滑映射$\psi : \Theta \to \mathcal{R}^n$,其中$d < n$. 那么称$M = \psi(\Theta)$为流形。向量$\theta$ 可以看做是一些控制参数，流形看做当参数改变时所有可能的测度信息$m = \psi(\theta)$. 关键工作是要从观测数据点m_i中恢复出潜在参数θ_i

给定流形上的数据点m_i 的集合，恢复出唯一的映射函数ψ和参数点θ_i，是不可能的。因为如果找到一个ψ是问题的解，那么总是可以找到另一个映射$\phi : \mathcal{R}^d \to \mathcal{R}^d$，那么将两个映射进行组合$\psi \circ \phi$就可以得到另外一个解。出于这种考虑我们需要做一些额外的假设从而使得取得的解唯一。

ISOMAP给出的两个假设是：

(ISO1)等距假设：映射函数ψ保留映射前后的两点之间的测地距不变，即

$$G(m, m') = |\theta - \theta'|, \forall m \leftrightarrow \theta, m' \leftrightarrow \theta'$$

(ISO2)凸集假设：参数空间Θ是$\mathcal{R}^d$空间的凸子集，即如果θ, θ' 是Θ空间的两个点，那么位于两点之间的线段$\{(1-t)\theta + t\theta' : t \in (0,1)\}$ 也位于Θ 内。

然而并非所有的问题中流形都是全局等距的，也并非所有的参数空间都是凸的，因而文章针对这些问题提出了更松的假设：

(LocISO1)等距假设：在每个点m足够小的范围内邻居，附近点m'到m的测地距等于其在参数空间内对应点θ和θ'之间的欧式距离。

(LocISO2)连通假设：参数空间Θ是$\mathcal{R}^d$中一个开放的连通子集。

假设$M \subset \mathcal{R}^n$是一个光滑流形，$T_m(M)$在点$m \in M$的切空间。考虑切空间是$\mathcal{R}^n$ 的一个子空间，我们可以讲每一个这样的切空间关联一个正交坐标系统通过使用内积的方式。

立即可以想到将$T_m(M)$看做是$\mathcal{R}^n$一个仿射子空间，与M相切于点m，原点$0 \in T_m(M)$等同于$m \in M$.m邻居中的每一个点$m' \in N_m$ 有唯一的最近邻点$v' \in T_m(M)$, 并且映射$m' \to v'$是光

滑的。在$T_m(M)$ 中的点的坐标由我们为$T_m(M)$选择的选择的正交坐标唯一确定。通过这种方式，我们可以获得$m \in M$邻居N_m的坐标，不妨设为$x_1^(tan, m), \cdots, x_d^(tan, m)$。

现在使用局部坐标来定义函数$f : M \to \mathcal{R}$的Hessian矩阵.假设$m' \in N_m$有局部坐标$x = x^(tan, m)$，那么规则$g(x) = f(m')$定义了一个函数$g : U \to \mathcal{R}$,其中U是$\mathcal{R}^d$中原点0的邻居。由于映射$m' \to x$是光滑的，因此函数g是二阶可导的。定义切平面坐标中m点出f 的Hessian矩阵作为g的Hessian矩阵。

$$H_f^{tan}(m)_{i,j} = \frac{\partial}{\partial x_i}\frac{\partial}{\partial x_j}g(x)|_{x=0}$$

简而言之，在每一个点m处，我们使用切平面坐标和f的微分。我们把这种方式的构建简称为切Hessian矩阵.

现在我们考虑定义在C^2上的二次形式

$$\mathcal{H}(f) = \int_M \parallel H_f^{(tan)} \parallel_F^2 dm$$

其中dm表示在M上的有严格正密度的概率测度.$\mathcal{H}(f)$测度了f 在流形M上的平均弯曲程度。

定理. *假设$M = \psi(\Theta)$,其中Θ是$\mathcal{R}^d$的一个开放连通子集，ψ 是将Θ局部等距嵌入$\mathcal{R}^n$的函数。那么$\mathcal{H}(f)$有一个$(d+1)$维零空间，包含一个常数函数空间和一个d维函数空间，由正交等距坐标张成。*

推论. *在上述定理的假设之下，原始的等距坐标θ能够被恢复,通过为$\mathcal{H}(f)$的零空间设定一组合适的基。*

2.2 算法步骤

考虑流形面M上的样本数据点m_i，我们希望恢复潜在的参数ψ和潜在的参数设置θ_i。上面的定理和推论告诉我们通过下列的算法可以解决这个问题。

HLLE algorithm:

输入: $\mathcal{R}^n$中N个点集合(m_i:$i = 1, ...N$)。
参数：d,参数空间的维数；k，邻居点的个数。
约束：$min(k, n) > d$
输出：$\mathcal{R}^d$中的对应的N点集合$(w_i : i = 1, \cdots, N)$

Procedure:

- 确定邻居：对于每一个数据点$m_i, i = 1, ..., n$,确定其欧式空间中最近k 近邻。记$\mathcal{N}_i$ 表示邻居节点的集合。对于每一个$\mathcal{N}_i, i = 1, ..., N$,组成一个$k * N$的矩阵$M^i$,每一行表示一个去中心化的邻居节点集合$m_j - \overline{m}_i, j \in \mathcal{N}_i$,其中$\overline{m}_i = Ave\{m_j : j \in \mathcal{N}_i\}$
- 获得切面坐标：对M_i做SVD分解，生成矩阵U,D,V; U是$k * min(k, n)$. U的前d列给出了点$\mathcal{N}_i$的切面坐标
- 计算Hessian估计：计算Hessian的最小平方估计。本质上，这是一个矩阵H^i，其特性是如果f是一个光滑函数$f : M \to \mathcal{R}$，$f_j = (f(m_i))$，通过从f中提出对应的邻居N_i中对应的点得到向量v^i，从而得到矩阵向量乘积$H^i v^i$给出$d(d+1)/2$个向量，每一项近似于Hessian矩阵的每一项$(\partial f/\partial U_i \partial U_j)$。
- 计算二次型：构建对称矩阵$\mathcal{H}_{ij}$满足

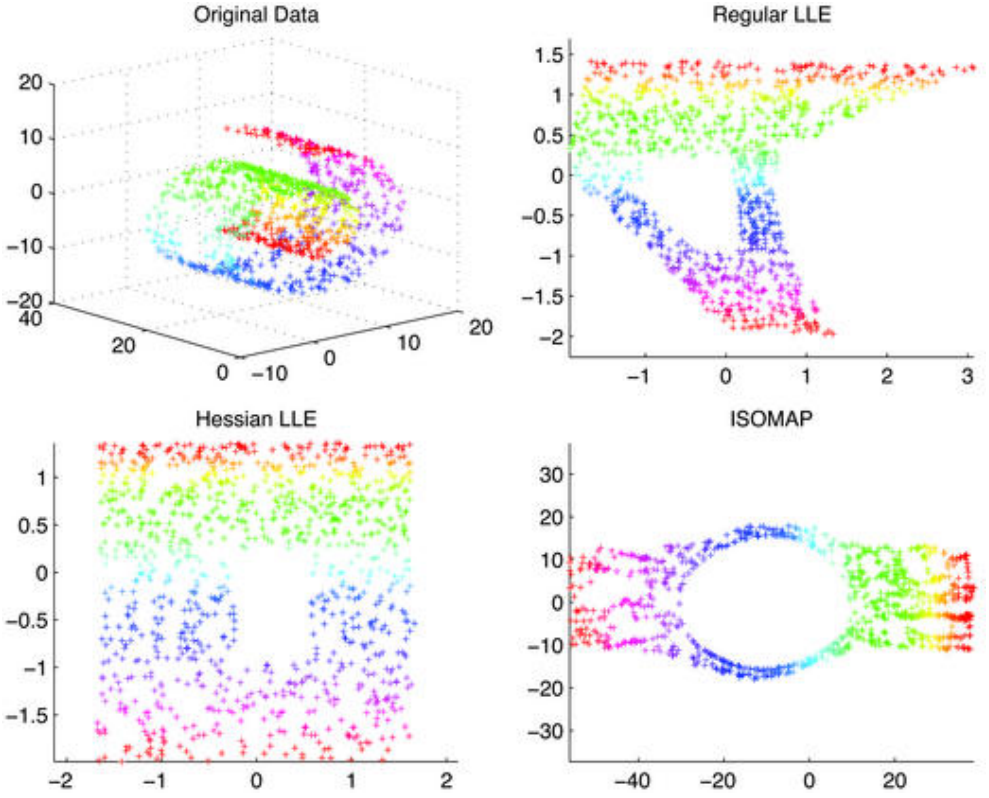
  $$\mathcal{H}_{i,j} = \sum_l \sum_r ((H^l)_{r,i}(H^l)_{r,j})$$

  其中H^l表示$d(d+1) \times k$矩阵，其中行r对应于Hessian矩阵中的项，列i对应邻居集合中的点。
- 寻找近似零空间：对$\mathcal{H}$进行特征分析，从其$d+1$个最小特征值中提出$d+1$维子空间。其中有一个特征值为0对应于常数函数的子空间，其余的d个特征值对应于有特征向量张成的d为空间$\hat{V}_d$。
- 寻找零空间的基：为$\hat{V}_d$选择一组基，其为特定的邻居几点N_0提供了一组标准正交基。给定的基有基向量$w^1, \cdots, w^d$

3 实验结果

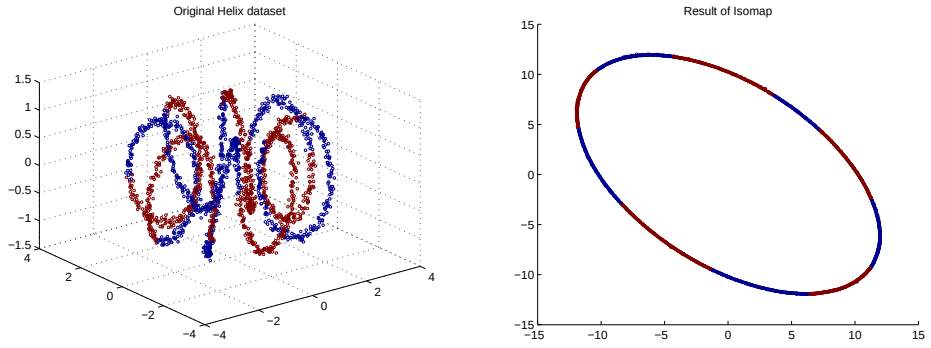
3.1 原论文结果

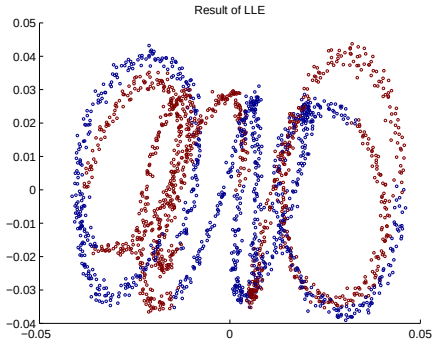
实验中将原来的矩形流形挖去了一个矩形条，流形在地位空间的展开不再是凸域，但是仍然是局部等距与欧式空间，缺失的矩形条状趋于使得结果函数不再对症，并且对于原始的参数不再线性。在这种情况下，非凸域使得ISOMAP引起缺失区域的膨胀，并且扭曲了剩余的区域。而HLLE 几乎能够完美的恢复到二维空间。

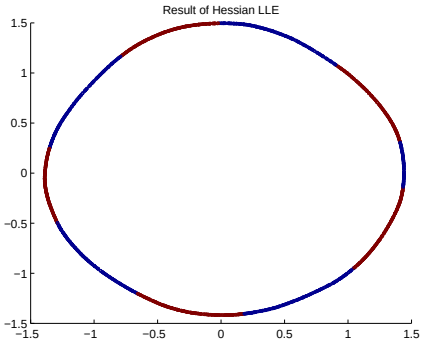


3.2 自己的结果

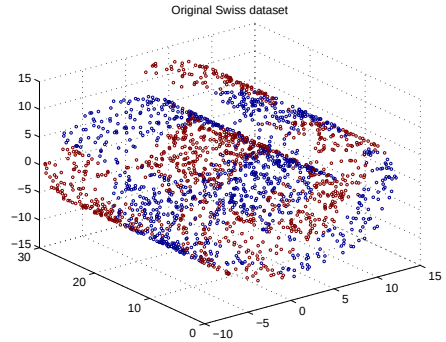
3.2.1 Helix数据集上的结果

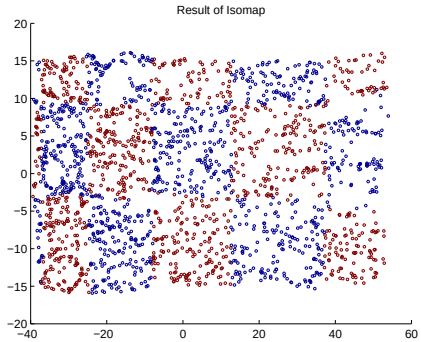


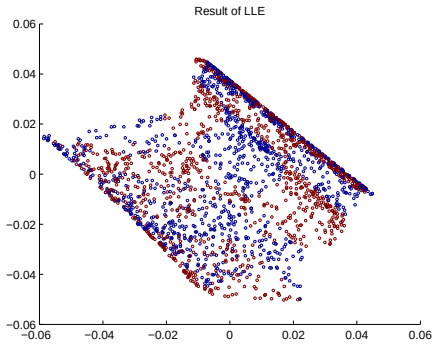


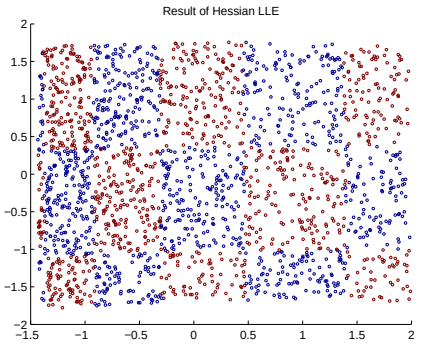


3.2.2 Swiss数据集上的结果

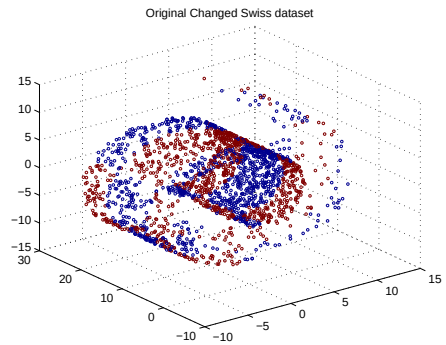


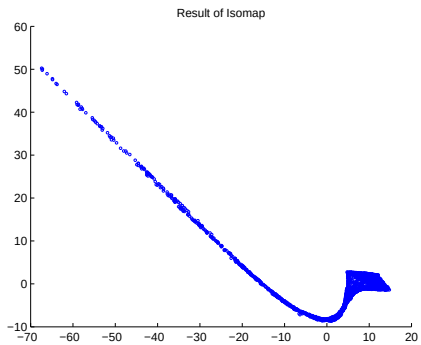


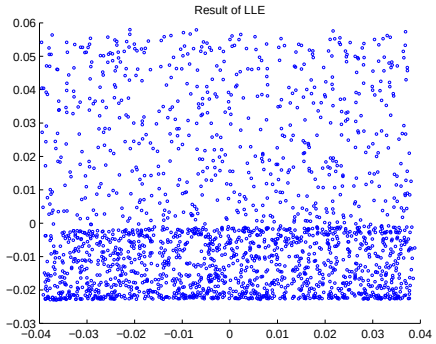


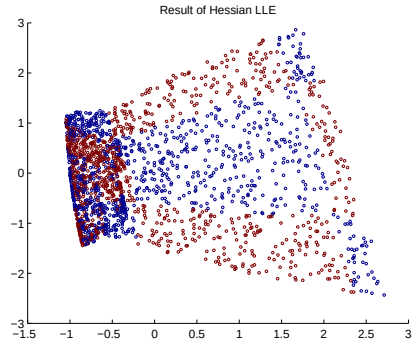


3.2.3 Changed Swiss数据集上的结果

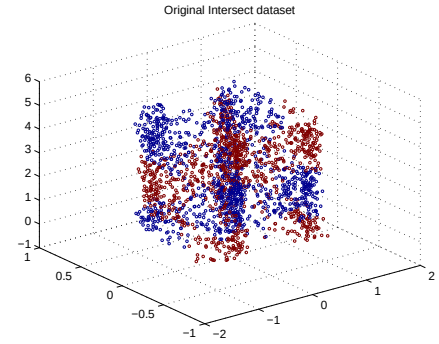


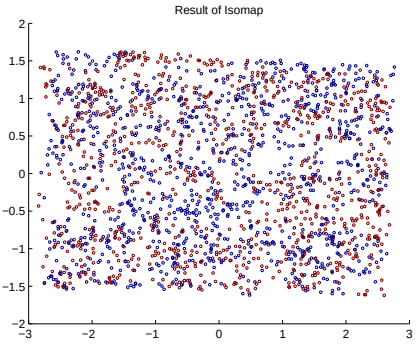


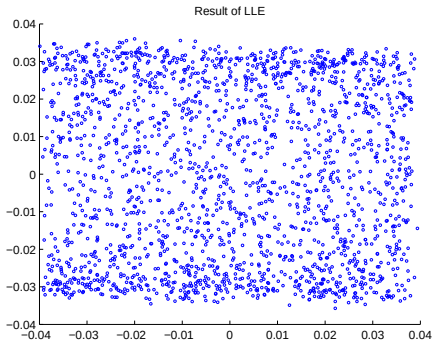


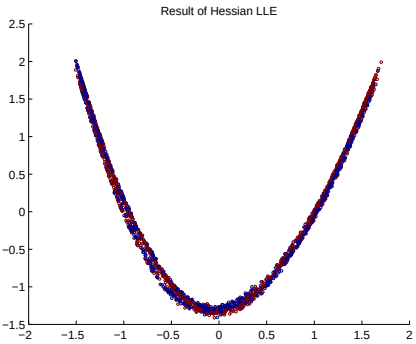


3.2.4 Intersect数据集上的结果









4 小结与讨论

本文主要对提出的Hessian LLE算法进行了解读，针对其提出的算法进行了相应的实现。Hessian LLE算法的提出一方面是对于ISOMAP算法所提出的两个严格要求的放松，ISOMAP要求流形是凸域，要求全局的等距；而HLLE针对实际情况中存在不是凸域的流形，提出不要求流形是严格的凸域，也不要求高维到低维空间的全局等距，只要求部分等距即可，这样就扩大了算法的适用范围。

与LLE算法相比，HLLE主要的差别在于引入了Hessian矩阵，但是为什么为引入Hessian 矩阵这个奇怪的东西还不是特别理解。但是从自己的实验结果来看，HLLE算法在流形为非凸集的情况下，确实好于ISOMAP和LLE算法。然而其也有不好的情况，比如对于测试的Intersect数据集，HLLE 算法的降维效果反而不如ISOMAP和LLE，笔者猜测的原因恰恰在于HLLE只是考虑了局部等距，而Intersect数据集的中间空隙恰恰使得只考虑局部等距特性不考虑全局距离是难以恢复的，这反而在某种程度上限制了HLLE的特性。

参考文献

[1] Donoho DL, Grimes C, Hessian eigenmaps:Locally linear embedding techniques for high-dimensional data, Proc Natl Acad Sci U S A. 2003 May 13: 100(10): 5591-5596

Appendix 核心算法代码

Hessian Algorithm

```
function mappedX = hlle(X, no_dims, k, eig_impl)
%HLLE Runs the Hessian LLE algorithm
%
%   mappedX = hlle(X, no_dims, k, eig_impl)
%
% Runs the Hessian LLE algorithm on dataset X to reduce its
    dimensionality
% to no_dims. The variable k specifies the number of nearest
    negihbors that
% is used.
%
%
    if ~exist('no_dims', 'var')
        no_dims = 2;
    end
    if ~exist('k', 'var')
        k = 12;
    end
    if ~exist('eig_impl', 'var')
        eig_impl = 'Matlab';
    end

    % Compute nearest neighbors
    if ischar(k)
        warning('Adaptive neighborhood selection often leads to
            problems in Hessian LLE.');
    end
    disp('Finding nearest neighbors...');
    [D, nind] = find_nn(X, k);
    max_k = size(nind, 2);

    % Size of original data
    n = size(X, 1);

    % Extra term count for quadratic form
    dp = no_dims * (no_dims + 1) / 2;
    W = sparse([], [], [], dp * n, n, dp * n * max_k);

    % For all datapoints
    disp('Building Hessian estimator for neighboring points...');
    for i=1:n
        % Center datapoints by substracting their mean
        tmp_ind = nind(i,:);
        tmp_ind = tmp_ind(tmp_ind ~= 0);
        kt = length(tmp_ind);
        thisx = X(tmp_ind,:);
        thisx = (thisx - repmat(mean(thisx, 1), kt, 1))';

        % Compute local coordinates (using SVD)
        [U, D, Vpr] = svd(thisx);
        if size(Vpr, 2) < no_dims
            no_dims = size(Vpr, 2);
            dp = no_dims * (no_dims + 1) / 2;
            warning(['Target dimensionality reduced to ' num2str(
                no_dims) '...']);
```

```
        end
        V = Vpr(:,1:no_dims);
            % Basically, the above is applying PCA to the
                neighborhood of Xi. The PCA mapping that is found
            % (and that is contained in V) is an approximation
                for the tangent space at Xi.

        % Build Hessian estimator
        clear Yi; clear Pii;
        ct = 0;
        for mm=1:no_dims
            startp = V(:,mm);
            for nn=1:length(mm:no_dims)
                indles = mm:no_dims;
                Yi(:,ct+nn) = startp .* (V(:,indles(nn)));
            end
            ct = ct + length(mm:no_dims);
        end
        Yi = [repmat(1, kt, 1) V Yi];

        % Gram-Schmidt orthogonalization (works different from Matlab
            QR function)
        [Yt, Orig] = mgs(Yi);
        Pii = Yt(:,no_dims + 2:end)';

        % Double check weights sum to 1
        for j=1:dp
            if sum(Pii(j,:)) > 0.0001
                tpp = Pii(j,:) ./ sum(Pii(j,:));
            else
                tpp = Pii(j,:);
            end

            % Fill weight matrix
            W((i - 1) * dp + j, tmp_ind) = tpp;
        end
    end

    % The weight matrix W is now entirely filled, perform
        eigenanalysis of W
    disp('Computing HLLE embedding (eigenanalysis)...');

    % Make sparse matrix that is inproduct of W
    G = W' * W;
        G(isnan(G)) = 0;
    G = sparse(G);

    % Clear some memory
    clear X thisx W D nind U D Vpr;

    % Perform eigendecomposition
    tol = 0;
    if strcmp(eig_impl, 'JDQR')
        options.Disp = 0;
        options.LSolver = 'bicgstab';
        [mappedX, eigenvals] = jdqr(G, no_dims + 1, tol, options);
    else
        options.disp = 0;
```

```
        options.issym = 1;
        options.isreal = 1;
        [mappedX, eigenvals] = eigs(G, no_dims + 1, tol, options);
    end

    % Sort eigenvalues and eigenvectors
    [eigenvals, ind] = sort(diag(eigenvals), 'ascend');
    if size(mappedX, 2) < no_dims + 1, no_dims = size(mappedX, 2) -
        1; end
    mappedX = mappedX(:,ind(2:no_dims + 1));

    % Extract nonzero coordinates
    mappedX = mappedX(:,1:no_dims)' * sqrt(n);
    mappedX = mappedX';
```

ISOMAP Algorithm

```
function [mappedX, mapping] = isomap(X, no_dims, k)
%ISOMAP Runs the Isomap algorithm
%
%   [mappedX, mapping] = isomap(X, no_dims, k);
%
% The functions runs the Isomap algorithm on dataset X to reduce the
% dimensionality of the dataset to no_dims. The number of neighbors
    used in
% the compuations is set by k (default = 12). This implementation
    does not
% use the Landmark-Isomap algorithm.
%
% If the neighborhood graph that is constructed is not completely
% connected, only the largest connected component is embedded. The
    indices
% of this component are returned in mapping.conn_comp.


    if ~exist('no_dims', 'var')
        no_dims = 2;
    end
    if ~exist('k', 'var')
        k = 12;
    end

    % Construct neighborhood graph
    disp('Constructing neighborhood graph...');
    D = real(find_nn(X, k));

    % Select largest connected component
    blocks = components(D)';
    count = zeros(1, max(blocks));
    for i=1:max(blocks)
        count(i) = length(find(blocks == i));
    end
    [count, block_no] = max(count);
    conn_comp = find(blocks == block_no);
    D = D(conn_comp, conn_comp);
    mapping.D = D;
    n = size(D, 1);

    % Compute shortest paths
```

```
    disp('Computing shortest paths...');
    D = dijkstra(D, 1:n);
    mapping.DD = D;

    % Performing MDS using eigenvector implementation
    disp('Constructing low-dimensional embedding...');
    D = D .^ 2;
    M = -.5 .* (bsxfun(@minus, bsxfun(@minus, D, sum(D, 1)' ./ n),
        sum(D, 1) ./ n) + sum(D(:)) ./ (n .^ 2));
    M(isnan(M)) = 0;
    M(isinf(M)) = 0;
    [vec, val] = eig(M);
        if size(vec, 2) < no_dims
                no_dims = size(vec, 2);
                warning(['Target dimensionality reduced to ' num2str(
                    no_dims) '...']);
        end

    % Computing final embedding
    [val, ind] = sort(real(diag(val)), 'descend');
    vec = vec(:,ind(1:no_dims));
    val = val(1:no_dims);
    mappedX = real(bsxfun(@times, vec, sqrt(val)'));

    % Store data for out-of-sample extension
    mapping.conn_comp = conn_comp;
    mapping.k = k;
    mapping.X = X(conn_comp,:);
    mapping.vec = vec;
    mapping.val = val;
    mapping.no_dims = no_dims;
```

LLE Algorithm

```
function [mappedX, mapping] = lle(X, no_dims, k, eig_impl)
%LLE Runs the locally linear embedding algorithm
%
%   mappedX = lle(X, no_dims, k, eig_impl)
%
% Runs the local linear embedding algorithm on dataset X to reduces
    its
% dimensionality to no_dims. In the LLE algorithm, the number of
    neighbors
% can be specified by k.
% The function returns the embedded coordinates in mappedX.

    if ~exist('no_dims', 'var')
        no_dims = 2;
    end
    if ~exist('k', 'var')
        k = 12;
    end
    if ~exist('eig_impl', 'var')
        eig_impl = 'Matlab';
    end

    % Get dimensionality and number of dimensions
    [n, d] = size(X);
```

```
% Compute pairwise distances and find nearest neighbors (
    vectorized implementation)
disp('Finding nearest neighbors...');
[distance, neighborhood] = find_nn(X, k);

% Identify largest connected component of the neighborhood graph
blocks = components(distance)';
count = zeros(1, max(blocks));
for i=1:max(blocks)
    count(i) = length(find(blocks == i));
end
[count, block_no] = max(count);
conn_comp = find(blocks == block_no);

% Update the neighborhood relations
tmp = 1:n;
tmp = tmp(conn_comp);
new_ind = zeros(n, 1);
for i=1:n
    ii = find(tmp == i);
    if ~isempty(ii), new_ind(i) = ii; end
end
neighborhood = neighborhood(conn_comp,:)';
for i=1:n
    neighborhood(neighborhood == i) = new_ind(i);
end
n = numel(conn_comp);
X = X(conn_comp,:)';
max_k = size(neighborhood, 1);

% Find reconstruction weights for all points by solving the MSE
    problem
% of reconstructing a point from each neighbours. A used
    constraint is
% that the sum of the reconstruction weights for a point should
    be 1.
disp('Compute reconstruction weights...');
if k > d
    tol = 1e-5;
else
    tol = 0;
end

% Construct reconstruction weight matrix
W = zeros(max_k, n);
for i=1:n
    nbhd = neighborhood(:,i);
    nbhd = nbhd(nbhd ~= 0);
    kt = numel(nbhd);
    z = bsxfun(@minus, X(:,nbhd), X(:,i));                    %
        Shift point to origin
    C = z' * z;
                                                              % Compute
         local covariance
    C = C + eye(kt, kt) * tol * trace(C);
                        % Regularization of covariance (if K > D)
    wi = C \ ones(kt, 1);                                     %
        Solve linear system
```

```
    wi = wi / sum(wi);                                          %
        Make sure that sum is 1
    W(:,i) = [wi; nan(max_k - kt, 1)];
end

% Now that we have the reconstruction weights matrix, we define
    the
% sparse cost matrix M = (I-W)'*(I-W).
M = sparse(1:n, 1:n, ones(1, n), n, n, 4 * max_k * n);
for i=1:n
   w = W(:,i);
   j = neighborhood(:,i);
   indices = find(j ~= 0 & ~isnan(w));
   j = j(indices);
   w = w(indices);
   M(i, j) = M(i, j) - w';
   M(j, i) = M(j, i) - w;
   M(j, j) = M(j, j) + w * w';
end

    % For sparse datasets, we might end up with NaNs or Infs in M
        . We just set them to zero for now...
    M(isnan(M)) = 0;
    M(isinf(M)) = 0;

% The embedding is computed from the bottom eigenvectors of this
    cost matrix
    disp('Compute embedding (solve eigenproblem)...');
tol = 0;
if strcmp(eig_impl, 'JDQR')
    options.Disp = 0;
    options.LSolver = 'bicgstab';
    [mappedX, eigenvals] = jdqr(M + eps * eye(n), no_dims + 1,
        tol, options);
else
    options.disp = 0;
    options.isreal = 1;
    options.issym = 1;
    [mappedX, eigenvals] = eigs(M + eps * eye(n), no_dims + 1,
        tol, options);          % only need bottom (no_dims + 1)
        eigenvectors
end
[eigenvals, ind] = sort(diag(eigenvals), 'ascend');
if size(mappedX, 2) < no_dims + 1
            no_dims = size(mappedX, 2) - 1;
            warning(['Target dimensionality reduced to ' num2str(
                no_dims) '...']);
end
eigenvals = eigenvals(2:no_dims + 1);
mappedX = mappedX(:,ind(2:no_dims + 1));
                                    % throw away zero eigenvector/
    value

% Save information on the mapping
mapping.k = k;
mapping.X = X';
mapping.vec = mappedX;
mapping.val = eigenvals;
```

```
mapping.conn_comp = conn_comp;
mapping.nbhd = distance;
```