Manifold Learning*

Ke Chen

To carry out this assessed coursework, you will need the Python notebook file, `manifold.ipynb`, which contain code to get you started with each of the assignments and the data files: `bars.npz` for the bar images and `face_tenenbaum.npz` for the face images. Other datasets (e.g. `ten_city`) are generated by using built-in functions in the notebook. All above are available in a zipped file on BlackBoard alongside this document.
Caveat: You may not display some resultant figures correctly with `%matplotlib notebook` which appears in the first line of the given Jupyter notebook file, especially on MS Windows. To deal with this issue, you need to use the Firefox browser instead of Chrome, Microsoft IE and Edge.

Manifold learning is one of the central themes in representation learning. In this coursework, you are asked to use Python to implement several manifold learning algorithms learnt from this course and apply your own implementation to synthetic and real datasets for manifold learning.

Supportive Software

To do this coursework, you are provided with our own Python implementation of LLE and visualisation tools required by this coursework.

Our implementation in Python includes: optimization functions in `optimization.py`, locally linear embedding in `lle.py`, display functions in `helpers.py` and dataset functions in `dataset.py`.

`optimization.py` enables you to carry out the gradient descent method required for the assignments regarding the stress-based MDS algorithms.

To solve an optimization problem by using the gradient descent method, the signature of this function is as follows:

```
gradient_descent(D, x0, loss_f, grad_f, lr, tol, max_iter)
```

where `D` is a distance matrix between points in the original space and `x0` is an initial value we want to get. `loss_f` and `grad_f` are the loss function and its first derivative or gradient, respectively. `lr` and `tol` are learning rate and tolerance for early stopping. `max_iter` is the maximum number of iteration.

In `lle.py`, the signature of `lle` function is as follows:

```
lle(data, n_components=2, n_neighbors=None, epsilon=None, dist_func=None,
                                                    reg_func=None)
```

where `data` contains data points in the original space, `n_components` is the dimensionality of target embedded space, `n_neighbors` is the number of neighbours for KNN, `epsilon` is the value of

*__Assessed Coursework__: the deadline and requirements can be found at the end of this document.

fixed radius for ϵ-distance, `dist_func` is the function to find out neighbours and their distances, and `reg_func` implements a regularization term to avoid the singular case.

Below, you can see two examples on how to use the `lle` function with different neighbourhoods.

```
from Code.lle import lle
data = ... # data point in the original space

n_dim = 2
k = 7
Y = lle(data,n_components=n_dim, n_neighbors =k,
        dist_func=nearest_neighbor_distance, reg_func = reg_func)
```

```
from Code.lle import lle
data = ... # data point in the original space

n_dim = 2
e = 5.3
Y = lle(data, n_components=n_dim, epsilon =e,
        dist_func=fixed_radius_distance, reg_func = reg_func)
```

where KNN is used in the first case and ϵ-NN is used in the second case. For the `reg_func` argument, the `None` value can be set except for **Assignment 7** where the function `reg_func(C, K)` provided in `manifold.ipynb` must be used.

`helpers.py` provides the useful functions to enable you to visualise the results of each assignment. This Python module contains `VIS_Shortest_path_2d` and `ImageViewer`, `VIS_Bars` classes.

`VIS_Shortest_path_2d` is used to display not only the embedded points in latent space but also the shortest path between two specified data points. Its constructor is as follows:

`VIS_Shortest_path_2d(proj, dist, predecessors, fig_vis)`

where `proj` refers to the embedded points in the target space, `dist` and `predecessors` are distance matrix and the index of predecessors for the shortest path which are obtained from `isomap` function. `fig_vis` is figure object created by the Python built-in function `figure()` in the matplotlib package. The usage of this class is as follows:

```
import matplotlib.pyplot as plt
from Code.isomap import isomap, dist_nearest_neighbor
points = ... # data point in column vector
n_components = 2
n_neighbors = 6

Y, dist, predecessors = isomap(...)
fig = plt.figure()
VIS_Shortest_path_2d(Y, dist, predecessors, fig)
```

By using the `VIS_Shortest_path_2d`, the embedded points in latent space will be plotted. Also, you can see the shortest path between two selected points.

`ImageViewer` is used to display images on the shortest path and its constructor is as follows:

```
ImageViewer(data, index, image_size, fig_vis, max_row=5)
```

where `data` refers to data points in the source space, `index` refers to the indices of points on the shortest path, `image_size` is the size of images used in `data`, `fig_vis` is a figure object created by Python built-in function, `figure()`, in the matplotlib package and `max_row` is the maximum number of images in a row. The usage of this class is as follows:

```
import matplotlib.pyplot as plt

data = ... # data points in the original space
image_size = [64,64]

start_idx = 1
end_idx = 100
path = get_shortest_path(predecessors, start_idx, end_idx)

fig = plt.figure()
img_viewer = ImageViewer(data, path, image_size, fig, 5)
img_viewer.show()
```

where `get_shortest_path` is provided in `manifold.ipynb` for **Assignment 5** and `predecessors` is returned from the `isomap` function.

`VIS_Bars` allows you to display the bar images in the dataset, bars.npz, and its result yielded by the `lle` function. Its constructor is as follows:

```
VIS_Bars(data, proj, fig_vis, color='r', image_size=(40,40), both=True)
```

where `data` are points in the source space, `proj` are the embedded points yielded by `lle`, `fig_vis` is a figure object created by `figure()` in the matplotlib package, `color` is the value of colour to represent the embedded points in this colour (e.g. 'r' for red, 'b' for blue), `image_size` is the size of images specified in `data`, and `both` is the flag to indicate whether both horizontal bars and vertical bars appear in `data` or not. If the last parameter, `both`, is set as True, the colour parameter, `color`, will be ignored. The usage of this class is as follows:

Case 1: set `both=True`

```
from Code.dataset import bars

data_bar, centers = bars()
data_bar = data_bar.T
centers = centers.T
image_size = [40,40]
```

```
n_neighbors = 5
n_components = 2
Y_bar = lle(...)

fig_bar = plt.figure()
vis_both = VIS_Bars(data=data_bar, proj=Y_bar, fig_vis=fig_bar,
                    image_size=image_size, both=True)
```

Case 2: set `both=False`

```
from Code.dataset import bars

data_bar, centers = bars()
data_bar = data_bar.T
centers = centers.T
image_size = [40,40]

n_neighbors = 5
n_components = 2
Y_bar = lle(...)

fig_bar = plt.figure()
vis_v = VIS_Bars(data=data_bar[:,:500], proj=Y_bar[:,:500],
                 fig_vis=fig_v_bar, color='r',
                 image_size=image_size, both=False)
```

The `dataset.py` module contains functions for loading datasets given in the `Data/` directory or for being used as the built-in functions for data generation.

`ten_city` contains data regarding the distances between 10 U.S.A. cities. This returns a lower triangle part of the distance matrix. The full distance matrix for this dataset can be achieved by adding this matrix and its transpose as follows:

```
from Code.dataset import ten_city

flying_dist, city = ten_city()
flying_dist = flying_dist + flying_dist.T
```

`synthetic_spiral` generates 3-D spiral dataset. This is a 3×30 matrix, meaning that there are 30 points in 3-D space. This function can be used as follows:

```
from Code.dataset import synthetic_spiral

X_spiral = synthetic_spiral()
```

`bars` loads the bar image dataset from `Data/bars.npz` and returns bar images in row vector and its center coordinates, and the size of images is 40×40. The bar dataset can be loaded and used as follows:

```
from Code.dataset import bars

data_bar, centers = bars()
data_bar = data_bar.T
centers = centers.T
image_size = [40,40]
```

`face_tenenbaum` loads the face dataset from `Data/face_tenenbaum.npz`. The size of each image in this dataset is 64×64. The images are returned as column vectors, so this dataset can be used as follows:

```
from Code.dataset import face_tenenbaum

data_face = face_tenenbaum()
image_size = [64,64]
```

1 Multidimensional Scaling

Multidimensional scaling (MDS) is one of manifold learning techniques that learns to preserve the distance between observations in a high-dimensional source space into a low-dimensional target space. It has been one of the most commonly used techniques for visualisation.

Given a distance matrix for N points in d-dimensional source space, $\Delta = (\delta_{ij})_{N\times N}$, MDS is going to find out low-dimensional representations of N points in a p-dimensional target space ($p < d$), $Z_{p\times N} = \{\boldsymbol{z}_1, \boldsymbol{z}_2, \cdots, \boldsymbol{z}_N\}$, such that $d_{ij} \approx f(\delta_{ij})$, where d_{ij} is the distance between points i and j in the target space and $f(\cdot)$ is a parametric monotonic function, e.g., $f(\delta_{ij}) = \alpha + \beta\delta_{ij}$. To solve the MDS problem, there are several different algorithms, e.g., cMDS and stress-based MDS[1].

1.1 Classical MDS (cMDS)

For a centralised data matrix, X, its inner-product (Gram) matrix, $G_{N\times N} = X^T X$, is expressible by its distance matrix, $\Delta^2 = (\delta_{ij}^2)_{N\times N}$:

$$G = X^T X = -\frac{1}{2} H \Delta^2 H; H_{N\times N} = I_{N\times N} - \frac{1}{N}\boldsymbol{e}\boldsymbol{e}^T,$$

where $I_{N\times N}$ is the identity matrix and $\boldsymbol{e} = [11\cdots 1]^T$

In cMDS, p-dimensional optimal coordinates in the target space can be obtained by conducting eigen-decomposition on $G = X^T X$. Let Σ_p is a diagonal matrix consisting of top p eigenvalues of the Gram matrix, G, and V_p is the matrix of the corresponding N-dimensional eigenvectors,

[1] For details of the cMDS and stress-based algorithms, see the "Multi-dimensional Scaling (MDS)" lecture note.

$\boldsymbol{v}_1, \boldsymbol{v}_2, \cdots, \boldsymbol{v}_p$. The optimal p-dimensional coordinates of N data points in X are collected in $Z^*_{p\times N}$ as follows:

$$Z^*_{p\times N} = \Sigma_p^{\frac{1}{2}} V_p^T.$$

Assignment 1 [6 marks] In your answer notebook, complete the function `cmds`, which implements cMDS algorithm described above. In your implementation, you can use the built-in functions in Python libraries, e.g., `sklearn.metrics.pairwise.euclidean_distances`, to calculate Euclidean distance and `numpy.linalg.eigh` to conduct eigen decomposition. Apply your completed `cmds` code on the `ten_city` dataset. In your answer notebook, (a) report the first 2 largest eigenvalues and their corresponding eigenvectors achieved on the dataset, and (b) visualise 1-D and 2-D embedded points along with the name of 10 cities.

1.2 Stress-based MDS

In stress-based MDS, optimal embedded coordinates are obtained by minimising loss functions (a.k.a. stress). The Sammon mapping is a well-known stress-based MDS.

Assignment 2 [2 marks] In your answer notebook, implement two functions: `loss_sammon` and `grad_sammon`, which are used to calculate the stress and its gradient in the Sammon mapping algorithm, respectively. In your implementation, you can use the built-in function in scikit-learn, `sklearn.metrics.pairwise.euclidean_distances`, to calculate distances between data points.

Assignment 3 [4 marks] Run the provided `stress_based_mds` function using the two functions you implement in **Assignment 2**. In your answer notebook, (a) report the optimal hyperparameters: learning rate (`lr`) and tolerance (`tol`) by setting `max_iter=6000`, (b) explain how you obtain two optimal hyperparameters with justification, and (c) display 2-D embedding results with the optimal parameters.

2 Isometric Feature Mapping (ISOMAP)

In the ISOMAP[2], geodesic distance is used to overcome the weakness of cMDS where the Euclidean distance metric does not respect the geometry of a non-linear manifold.

The ISOMAP algorithm consists of two steps:

1. Find out **approximate geodesic distances** between any points in high-dimensional space.
2. Apply **cMDS** to the geodesic distance matrix for low-dimensional embedding.

Assignment 4 [4 marks] To gain a geodesic distance matrix, local distances need to be calculated via one of two ways: i) *K*-nearest neighbourhood (KNN) or ii) ϵ-radius neighbourhood. In your answer notebook, a) implement the `fixed_radius_distance`, the `nearest_neighbour_distance` and `isomap` functions (where you must re-use your own c-MDS implementation in **Assignment 1**), and (b) run your implemented `isomap` function on the Swiss roll dataset with the given parameters:

[2] For details of the ISOMAP, see the "Isometric Feature Mapping (ISOMAP)" lecture note.

`k`=6, `epsilon`=3.5, `target=2-D` and display your results. Note that you can use the built-in function in scikit-learn, `sklearn.metrics.pairwise.euclidean_distances`, to calculate distances between data points. However, the use of `sklearn.neighbors.NearestNeighbors` in your code is strictly forbidden.

Assignment 5 [2 marks] Run your own `isomap` code implemented in **Assignment 4** on the face dataset, `face_tenenbaum.npz`. The routine used for displaying the connectivity between the neighbouring embedded points in low-dimensional space is provided for this dataset in the notebook file. In the figure showing the connectivity, randomly pick two pairs of start and end points to specify two shortest paths that reflect meaningful manifold structures underlying this dataset. In your answer notebook, (a) display the images on the *two* specific paths you choose with the layout of 5 images in each row, and (b) report the indices of points you choose on each path and explain what you observe on those points in two paths in terms of manifold learning. To do this assignment, you may use the provided functions/classes, `VIS_Shortest_path_2d` for selecting a pair of start and end points to generate a shortest path, and `ImageViewer` to display all images on your specified paths. Nevertheless, you are allowed to show the images with your own display functions.

3 Locally Linear Embedding (LLE)

The LLE[3] is yet another method to solve the same problem as ISOMAP encounters. For a given dataset, $X_{p\times N}$, a point $\boldsymbol{x}_i$ is reconstructed by linear combination of its neighbours $\boldsymbol{x}_j$ and optimal weights, W^*, can be learned by minimising the following loss function:

$$\mathcal{L}(W;X)=\sum_{i=1}^{N}\|\boldsymbol{x}_j-\sum_{j=1}^{N}W_{ij}\boldsymbol{x}_j\|^2 \quad s.t.\sum_{j=1}^{N}W_{ij}=1.$$

Once the weights are obtained, the points ,$\boldsymbol{x}_i$, can be embedded into target space as p-dimensional coordinates ($p<d$), $Z_{p\times N}=\{\boldsymbol{z}_1^*,\boldsymbol{z}_2^*,\cdots,\boldsymbol{z}_N^*\}$, by minimising the loss function with respect to the embedded coordinates in Z:

$$\mathcal{L}(Z;W^*)=\sum_{i=1}^{N}\|\boldsymbol{z}_i-\sum_{j=1}^{N}\boldsymbol{z}_j\|^2 \quad s.t.\sum_{i=1}^{N}\boldsymbol{z}_i=0,\frac{1}{N-1}\sum_{i=1}^{N}\boldsymbol{z}_i\boldsymbol{z}_i^T=I_{N\times N}.$$

The p-dimensional optimal embedded coordinates can be obtained by conducting eigen-analysis on $D=(I-W^*)^T(I-W^*)$ where W^* is the optimal weight matrix for data reconstruction in high dimensional source space.

Assignment 6 [4 marks] In the notebook file, you are provided with our LLE implementation in Python (see **Supportive Software** for details):

```
lle(data, n_components=2, n_neighbors=None, epsilon=None, dist_func=None,
                                reg_func=None)
```

[3] For details of the LLE, see the "Locally Linear Embedding (LLE)" lecture note.

In this assignment, you are asked to apply this function to the S-Curve dataset provided in the scikit-learn library, `sklearn.datasets.make_s_curve`, which has been included in the notebook file. In your experiment, you need to use both KNN and the fixed ϵ radius methods to decide local neighbourhood of a point. Thus, you need to find proper values for `n_neighbors` and `epsilon`, respectively. As an exploratory exercise, you are asked to figure out a criterion that can guide you to find out the optimal hyperparameters, K and ϵ (Note that any trial-and-test criteria are unacceptable). In your experiment, apply your criterion to investigate the value of K taken in the range from 5 to 50 with the increment by 1 for KNN neighbourhood and the value of ϵ in the range from 0.1 to 0.8 with the increment by 0.1. In your answer notebook, (a) describe a criterion that works effectively in finding out the optimal hyperparameters, K and ϵ; (b) provide the computational evidence based on your criterion described in (a) to find out the optimal K and ϵ on the S-Curve dataset. (c) display 2-D embedded coordinates of the S-Curve dataset with the optimal parameters. You can reuse the distance functions: `fixed_radius_distance` and `nearest_neighbour_distance`, you implement in **Assignment 4**.

Assignment 7 [3 marks] In this assignment, you are asked to apply the provided LLE function with KNN neighbourhood to the bar dataset, `bars.npz`, where there are 1,000 images; 500 vertical and 500 horizontal bars. For this dataset, you must use the provided regularisation function `reg_func` and the `nearest_neighbour_distance`, you implement in **Assignment 4** in the `lle` function (Note that failing to use these two functions may result in incorrect results). To display 2-D embedding coordinates, you can use the provided `VIS_Bars` class. In your answer book, (a) applying the criterion you described in **Assignment 6**(a), report the optimal K in the range (40 < K < 60) with computational evidence, (b) display 2-D embedded coordinates of 1,000 bar images using the optimal K, and (c) describe what you observe from those embedded coordinates in terms of manifold learning.

Requirement: Before starting working on this assessed coursework, you need to

1. download all the files required by this coursework from Blackboard as specified at the beginning of this document;
2. unzip the file then you should be see a Jupyter notebook file named `manifold.ipynb` and two sub-directories named `Data` and `Code`, respectively (you must keep this directory/sub-directory structure and their names unchanged when you work on this coursework);
3. rename `manifold.ipynb` in the directory as `yourfullname_manifold.ipynb`. For instance, if your name is "John Smith", your filename should be `john_smith_manifold.ipynb`. This file will be your answer notebook to be submitted for marking, so you must include everything required by the coursework in this Jupyter notebook.

Deliverable: Only your answer notebook, `yourfullname_manifold.ipynb`, which should include all your code, output, answers and your interpretation/justification. In this Jupyter notebook, all assignments have been separated with the clear delimiters. You must put your stuff regarding an assignment in those cells related to this assignment and, if necessary, create new cells within the delimiters of this assignment.

Your answer notebook, `yourfullname_manifold.ipynb`, must be submitted via the Black-

board.

Marking: Marking is on the basis of (1) correctness of results and quality of comments on your code; (2) rigorous experimentation; (3) how informative and clear your description/answers presented in your answer book; and (4) your knowledge exhibited, interpretation and justification.

Late Submission Policy: The default university late submission policy (for details, see the official document: http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=29825) is applied to this coursework.

Extension Policy: The default departmental extension policy is applied to this coursework. That is, no extension is allowed unless you have a mitigating circumstance. If you have any mitigating circumstance and want to make an extension, you should submit the completed mitigating circumstance form to SSO (for details, see the departmental Mitigating Circumstances page: http://studentnet.cs.manchester.ac.uk/assessment/mitigatingcircumstances.php?view=pgt). Note: the decision will be made by the departmental mitigating circumstance panel rather than the lecturer.

Deadline: 18:00 GMT, 2nd December 2021 (Thursday)