Documentation: Unit Test of Diffusion Maps with Python

Ketson R. M. dos Santos
email: ketson.santos@epfl.ch

1 Introduction

1.1 Objective and methods

Although `TestDiffusionMaps.py` is self-explanatory, this supplementary document contains a detailed description of the Diffusion Maps framework. Further, the classes in `TestDiffusionMaps.py` can be used to solve problems using Diffusion Maps and to perform unit tests in the method `fit`. If you have interest, see the following GitHub repository https://tinyurl.com/3mxe9mzt, where Jupyter notebooks for two distinct problems and for the unit tests are provided.

The main objective of `TestDiffusionMaps.py` is to show how the implementation of Diffusion Maps can be simple and powerful. Moreover, two simple examples of unit tests are implemented to verify the code reliability. The classes in `TestDiffusionMaps.py` were implemented in Python 3.9 using the oriented-object programming (OOP) paradigm, and the examples were run on a computer with macOS. Further, the code requires the following Python toolboxes numpy, scipy, and scikit-learn.

1.2 Theory of Diffusion Maps

Nonlinear dimensionality reduction techniques consider that high-dimensional data can lie on a low-dimensional manifold. To reveal this embedded low-dimensional structure, one can resort to kernel-based techniques such as Diffusion Maps [1]; where the spectral decomposition of the transition matrix of a random walk performed on the data is used to determine a new set of coordinates, also known as diffusion coordinates, embedding this manifold into a space of reduced dimension. For example, data observed in a 3-D space can be constrained to a 2-D structure that can be revealed by the diffusion coordinates. Next, the Diffusion Maps technique is described in details.

Given a dataset $S_{\mathbf{X}} = \{\mathbf{X}_i, \ldots, \mathbf{X}_N\}$ with $\mathbf{X}_i \in \mathbb{R}^n$, and a positive semi-definite kernel $k : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ such as the Gaussian kernel presented in

Eq. (1), we can construct the kernel matrix $\mathbf{K} = [k_{ij}] = [k(\mathbf{X}_i, \mathbf{X}_j)] \in \mathbb{R}^{N \times N}$ encoding the pairwise similarity of data points in $S_{\mathbf{X}}$. In this regard, the kernel attains its maximum value when $\mathbf{X}_i = \mathbf{X}_j$.

$$k(\mathbf{X}_i, \mathbf{X}_j) = \exp\left(-\frac{||\mathbf{X}_i - \mathbf{X}_j||_2^2}{4\epsilon}\right), \tag{1}$$

where ϵ is the lenght-scale parameter controlling the level of correlation of a point with its neighbors. Next, to obtain the coordinates embedding the high-dimensional dataset $S_{\mathbf{X}}$ into a low-dimensional space, we first build the following diagonal matrix $\mathbf{D} = [D_{ii}] \in \mathbb{R}^{N \times N}$ as

$$D_{ii} = \sum_{j=1}^{N} k_{ij}. \tag{2}$$

Next, a normalized version of the kernel matrix k_{ij} is obtained as

$$\kappa_{ij} = \frac{k_{ij}}{\sqrt{D_{ii} D_{jj}}}, \tag{3}$$

and the transition matrix $\mathbf{P}$ is obtained from the following normalization

$$P_{ij} = \frac{\kappa_{ij}}{\sum_{k=1}^{N} \kappa_{ik}}. \tag{4}$$

The eigendecomposition of $\mathbf{P} = [P_{ij}]$ yields a set of eigenvectors $\mathbf{\Phi} = [\phi_0, \ldots, \phi_N]$ and their respective eigenvalues $\mathbf{\Lambda} = \text{diag}(\lambda_0, \ldots, \lambda_N)$. Thus, every element $\mathbf{X}_i$ of $S_{\mathbf{X}}$ has a representation on a low-dimensional space defined by the **diffusion coordinates** $\boldsymbol{\psi}_i = [\lambda_0 \Phi_{i0}, \ldots, \lambda_r \Phi_{ir}]^T$, where $r \leq N$. For a more detailed description of the Diffusion Maps framework see [1].

2 Class *DiffusionMaps*

The Diffusion Maps framework introduced in Section 1 is implemented as a class in `TestDiffusionMaps.py`. Next, some elements of the class `DiffusionMaps` are presented. First, the attributes of `DiffusionMaps` are the following:

- Kernel matrix $\mathbf{K}$ (`kernel_matrix`)
- Transition matrix $\mathbf{P}$ (`transition_matrix`)
- Input data $S_{\mathbf{X}}$ (`X`)
- Diffusion coordinates $\boldsymbol{\psi}_i$ (`diffusion_coordinates`)

as presented in the following piece of code.

```python
class DiffusionMaps:
    """
    Diffusion maps is a nonlinear dimensionality reduction technique
    for embedding high-dimensional data into a
    low-dimensional Euclidean space revealing their intrinsic geometric
     structure.
    """

    def __init__(self):
        # Attributes of ``DiffusionMaps``.
        self.kernel_matrix = None  # Kernel matrix.
        self.transition_matrix = None  # Kernel matrix.
        self.X = None  # Input data.
        self.diffusion_coordinates = None  # Diffusion Coordinates.
```

Listing 1: class DiffusionMaps.

These attributes are instantiated by using the method `fit`; where `X` is the input data (2-D numpy array) equivalent to $S_{\mathbf{X}}$, and `epsilon` is the length-scale parameter ϵ in Eq. 1 (it must be larger than zero).

```python
    def fit(self, X=None, epsilon=None):
```

Listing 2: Instantiating the attributes of DiffusionMaps

Once the class `DiffusionMaps` is instantiated, the attributes and methods are accessible from the object using OOP paradigm with Python.

3 Class MyTestCase

The file `TestDiffusionMaps.py` also contain the class `MyTestCase` for performing the unit tests in two distinct cases. The first test (Listing 3) is used to compare the number of points in `diffusion_coordinates` and the number of points in the the input dataset `X`. Therefore, the diffusion coordinates must be consistent with the input dataset.

```python
# Test 1: test the if the number of diffusion coordinates is equal to
    the number of data points in `X`.
def test_length_coordinates(self):
    X, _ = make_swiss_roll(n_samples=1000)  # Sample `n_samples` points
     from the Swiss Roll manifold.
    dfm = DiffusionMaps()  # Object of `DiffusionMaps`.
    dfm.fit(X=X, epsilon=1.0)  # Instantiate the attributes of `
    DiffusionMaps` with `epsilon` = 1.

    # Test if the number of data points in `X` is equal to the length
    of diffusion_coordinates.
    self.assertEqual(np.shape(X)[0], np.shape(dfm.diffusion_coordinates
    )[0])
```

Listing 3: Unit test 1.

The second unit test (Listing 4) check the raise of exceptions in the code. In particular, it verifies if `raise ValueError` is called when the shape of `X` is not acceptable. In this implementation of Diffusion Maps `X` must be an array

with two dimensions (row and columns), otherwise the code shows the following error message: `Not acceptable shape for 'X'`. Therefore, the code will pass this test only if it raises an exception for this particular condition.

```
# Test 2: test if the code correctly raise an exception for the shape of the input data 'X'.
    def test_input_shape_exception(self):
        # Get a random array with shape (100, 3, 1). DiffusionMaps only accepts len(np.shape('X')) = 2.
        X = np.random.rand(100, 3, 1)
        dfm = DiffusionMaps()  # Object of 'DiffusionMaps'.

        # Get the exception for raise ValueError.
        with self.assertRaises(ValueError) as exception_context:
            dfm.fit(X=X, epsilon=1.0)  # Instantiate the attributes of 'DiffusionMaps' with 'epsilon' = 1.

        # The code will pass the test when it will raise the following exception.
        self.assertEqual(str(exception_context.exception), 'Not acceptable shape for 'X'.')
```

Listing 4: Unit test 2.

4 Running DiffusionMaps

This section shows how to run one example using `DiffusionMaps`. The problem consist of unwrapping the Swiss Roll manifold, which is defined in a 3-D space, and it is implicitly included in the unit test presented in Listing 3. Herein, the Swiss Roll manifold is sampled, and a point cloud of points in the 3-D space represent this surface, as observed in Fig. 1 when 2,000 samples are generated using the following scikit-learn command:

```
from sklearn.datasets import make_swiss_roll
X, color = make_swiss_roll(n_samples=2000, random_state=1)
```

Listing 5: Sampling the Swiss Roll manifold.

One can easily see that the Swiss Roll manifold is defined in 3-D, but it has an intrinsic 2-D structure. Thus, one can use Diffusion Maps to unwrap this 3-D structure. Using the code presented herein, one can obtain the representation of the Swiss Roll manifold in a 2-D space. To this aim, one can use the following commands to instantiate the `DiffusionMaps` class.

```
dfm = DiffusionMaps()
dfm.fit(X=X, epsilon=1.0)
```

Listing 6: Instantiating the class DiffusionMaps.

One can observe that the an object of `DiffusionMaps` (`dfm`) is created without input arguments. To instantiate the attributes one can use the method `fit`, which receives the input dataset `X` and the value of `epsilon` equal to 1.0 (which is selected by the user).

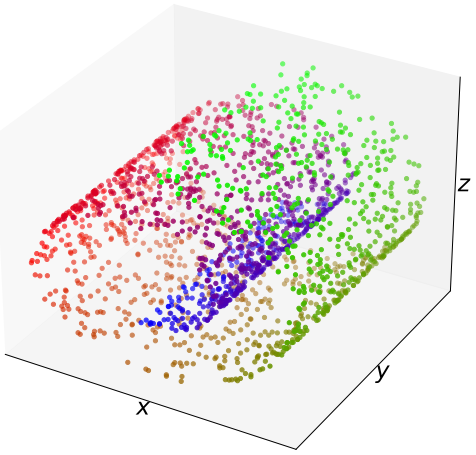

Figure 1: Example 1: Grassmannian diffusion manifold: a) training set for GH, and b) predicted Grassmannian diffusion manifold for 3,000 additional samples.

One of the attributes of `DiffusionMaps` is `diffusion_coordinates` (presented as $\boldsymbol{\psi}_i$ in Section 1.2), which stores the diffusion coordinates used to embed the Swiss Roll manifold into a 2-D space. This embedding is presented in Fig. 2, where the diffusion coordinates ψ_2 (`diffusion_coordinates`[:, 2]), ψ_3 (`diffusion_coordinates`[:, 3]), ψ_4 (`diffusion_coordinates`[:, 4]), and ψ_5 (`diffusion_coordinates`[:, 5]) are plotted with respect to ψ_1 (`diffusion_coordinates`[:, 1]). **It is important mentioning that the 0th diffusion coordinates (ψ_0) are not used in this kind of embedding because they represent the trivial eigendirection. Therefore, when plotting the diffusion coordinates there is no need to show `diffusion_coordinates`[:, 0]**. Based on Fig. 2 one can observe that ψ_1 and ψ_5 are the directions that unwrap the Swiss Roll manifold.

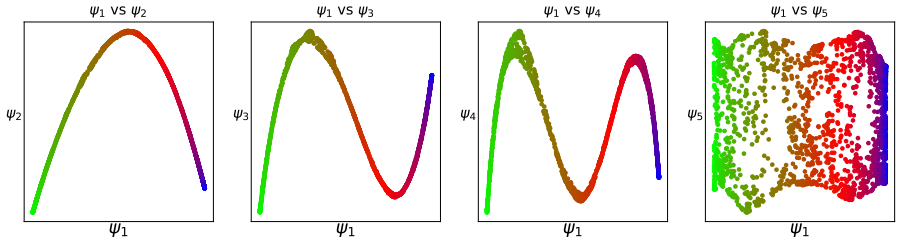

Figure 2: Example 1: Grassmannian diffusion manifold: a) training set for GH, and b) predicted Grassmannian diffusion manifold for 3,000 additional samples.

5 Running MyTestCase

To run the unit tests, one can use the following command in the directory containing `TestDiffusionMaps.py`:

```
$ python -m unittest TestDiffusionMaps
```

Listing 7: Running the unit test.

Therefore, one can expect the following outcome:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.514s

OK
```

Listing 8: Outcome of the unit test.

References

[1] R. R. Coifman and S. Lafon. Diffusion maps. *Applied and Computational Harmonic Analysis*, 21(1):5 – 30, 2006. Special Issue: Diffusion Maps and Wavelets.