Department of Informatics

Technische Universität München

Bachelor's Thesis in Informatics

Scalable Manifold Learning through Landmark Diffusion

Veselina Vazova

Department of Informatics

Technische Universität München

Bachelor´s Thesis in Informatics

Scalable Manifold Learning through Landmark Diffusion

Skalierbares Lernen von Mannigfaltigkeiten durch Diffusion auf Untermengen

Author:	Veselina Vazova
Examiner:	Prof. Dr. Christian Mendl
Assistant advisor:	Dr. Felix Dietrich
Submission Date:	October 15th, 2021

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

October 15th, 2021 Veselina Vazova

Abstract

Manifold learning by spectral embedding is a technique that can be used for non-linear dimensionality reduction and clustering. By extracting the spectral properties of high dimensional data, the intrinsic manifold where data is presumably located on, can be embedded into a lower dimension.
A newly proposed algorithm in the field of spectral embedding that has the goal of providing a scalable and robust approach to dimensionality reduction is Roseland by Chao Shen and Hau-Tieng Wu. The algorithm requires two sets: the data set and the landmark set and returns an embedding of the data set in a desired lower dimension. To achieve this, a landmark-set affinity matrix is computed that represents the affinities between the points in the data set and the points in the landmark set. This matrix is then normalized, and the singular value decomposition of the normalized matrix is evaluated. Using the singular vectors and the singular values, the Roseland embedding for a given diffusion time is finally computed.
At its core, the algorithm is similar to the Diffusion Maps algorithm, whereas the main differences lie in the affinity matrix and the algorithm for spectral decomposition. In Diffusion Maps, the affinities between the data points themselves are calculated, without first ”detouring” through a landmark set. Instead of the singular value decomposition, the eigendecomposition is performed. If the number of landmarks is significantly smaller than the data set, Roseland fits the data set faster than Diffusion Maps.
In this thesis, we describe an efficient implementation of the Roseland algorithm in the datafold package. We consider different approaches to constructing the landmark set when it is not provided, and we compare the results. Finally, we evaluate the efficiency of the novel algorithm by comparing it to the performance of Diffusion Maps.

Contents

1 Introduction

Big data poses a challenge not only from the perspective of the large number of examples there are in a data set, but also in that each data point contained in the set may have a big number of features. In most situations, however, not all of those features are relevant when performing a standard machine learning algorithm. In some cases, they might even lead to misleading results. The number of features corresponds to the dimensionality of the data. Naturally, the more dimensions there are, the more inefficient it is to process the data sets in terms of runtime speed. Consequently, there are efforts in reducing the dimensionality. Some of them, e.g., the Principal Component Analysis take a linear approach. However, linear approaches assume that data lies on a linear subspace, which is a major drawback for the embedding of some data sets. In contrast, non-linear approaches can correctly reduce the number of dimensions of data that has more complex intrinsic geometry [23].
Manifold learning lands in the latter category, as it assumes that data is sampled from a manifold with a low intrinsic dimensionality that is embedded into a higher dimensional space. A d-dimensional manifold is a set that is locally homeomorphic to a d-dimensional Euclidean space. Manifold algorithms often additionally assume that the manifolds in question have a certain degree of smoothness [5], or metric structures like Riemannian metric tensors. Diffusion Maps is an algorithm based on manifold learning. The idea behind it is to compute the similarities (also called affinities) between data points and by using those similarities to diffuse over the data set in an effort to unravel the connectivity of the whole manifold. To do so, it needs to compute a Markov matrix using the affinities and to extract the spectral properties, namely eigenvalues and eigenvectors, of this transition matrix. By using a subset of these properties, we can then compute the coordinates of the manifold in a lower dimension. With a proper normalization of the similarity matrix, Diffusion Maps can successfully approximate the Laplace-Beltrami operator independently from the density of the data points and hence, provide a geometrically accurate unfolding of the manifold.
However, there are several issues associated with the algorithm, one of them being scalability. In its standard formulation, Diffusion Maps has a cubic complexity. Therefore, ordinary computers struggle with running the algorithm when the size of the data set grows. A newly proposed algorithm called Roseland [21, 20, 17] aims to improve the scalability by using a landmark set. The algorithm is closely related to Diffusion Maps: it provides a spectral embedding of the data by using a diffusion process [20]. The scalability is improved because the similarity matrix contains the affinities between the data points and the landmarks. Since per definition the number of landmarks is smaller than the size of the data set, the similarity matrix in Roseland has much fewer elements than the one in

Diffusion Maps. It is computationally more efficient to extract the spectral properties of this thin matrix, and since this is the step that dominates the overall complexity in both algorithms, the runtime of Roseland is lower. There are some drawbacks associated with Roseland. Diffusion Maps yields more accurate approximations of both the eigenvector and eigenvalues than Roseland for all landmark set sizes [21], even if the landmark set is the same as the data set. Another area where Diffusion Maps performs better than Roseland is when handling non-uniformly distributed data sets. In Diffusion Maps, the impacts of the non-uniform data sampling can be removed by an additional normalization with the term α, and approximations of the Laplace-Beltrami operator can be obtained [7]. In contrast, Roseland can work around non-uniformly distributed data by designing the landmark set, such that $p_Y(x) \propto \frac{1}{p_X^2(x)}$ [20]. Here, p stands for the probability density function, X the data set and Y the landmark set. This is not as straightforward as the normalization with α, and additionally, requires knowledge about the density of the data set.
The goal of this thesis is to implement Roseland in the datafold package [16] and evaluate its performance in terms of scalability and robustness in comparison to Diffusion Maps. The Python package datafold is a machine learning package that already contains an implementation of Diffusion Maps.
For the implementation, we consider how to acquire the landmark set when it is not provided by the user. Our approach is to subsample $\gamma \cdot n$ landmarks from the data set, where n represents the number of data points and $\gamma \in (0, 1]$ is a factor controlled by the user. We choose 0.25 as the default value since we aim at a number of landmarks that is relatively small but still big enough to ensure good convergence of the kernel matrix to the Laplace-Beltrami operator. We observe that when the data set size grows, the landmark set can have a smaller relative size.
Our implementation of Roseland has an implicit cubic complexity because it designs the landmark set as a subset of the data set (an exception is when the landmarks are provided when creating the Roseland instance). In this sense, the complexity of Diffusion Maps is not improved, unless the size of the landmark set is fixed. In that case, Roseland has a linear computational complexity with respect to the number of data points. Even if we do not fix the size of the landmark set, Roseland still has a lower runtime than Diffusion Maps because of the smaller affinity matrix. For this to hold, however, the landmark set size has to be below a certain threshold in relation to the data set size. We observe that when the amount of landmarks is equal to or more than 70% of the number of data points, the runtime of Roseland approaches and surpasses the one of Diffusion Maps. In such cases, there is no advantage to use Roseland.
We additionally compare Roseland and Diffusion Maps in terms of robustness to noise. When we use a Gaussian kernel to compute the affinities in Diffusion Maps, Roseland yields less noisy results than Diffusion Maps in a visual comparison. If we switch the Gaussian kernel with a Continous k-Nearest Neighbour kernel, Roseland handles noise better if the landmark set is not noisy. Otherwise, the results are comparable in terms of how noisy they are.

2 State of the art

In this chapter, we give an overview of popular dimensionality reduction approaches. We further discuss the manifold learning algorithm, Diffusion Maps, that is closely related to Roseland. In addition, we provide an overall description of datafold, where we mostly focus on the parts that have the most relevance in the context of dimensionality reduction.

2.1 Dimensionality reduction

Dimensionality reduction algorithms aim at reducing the number of dimensions while attempting to keep the intrinsic dimensionality of the data. That is, the minimum number of features needed to represent the properties of the set [23]. More formally, we have a data set $\mathcal{X}$ with n data vectors $x \in \mathbb{R}^q$, where q stands for the number of features or *dimensions* of the data. The goal is to find a function that maps x to $x' \in \mathbb{R}^{q'}$, where $q' \ll q$.
In this section, we present two established algorithms: a linear one and a non-linear one, that propose a solution to the problem. We highlight their differences and discuss several advantages and shortcomings of both of them.

2.1.1 PCA

Principal Component Analysis (PCA) [26] is one of the most well-known dimensionality reduction algorithms. It provides a linear approach at solving the problem by embedding the data into a linear subspace. The main idea of PCA is to identify a set of orthogonal vectors along which data varies the most [5]. These vectors are called principal components, and they are formed as a linear combination of the original features. To embed the data we use the first q' principal components that have the biggest variance. Sometimes, when data does not vary along more than q' principal components or if all $q' = q$ principal components are used, the embeddings will be exact. Otherwise, there will be some error. Because the principal components are orthogonal to each other, we compute a new orthogonal *basis* for our data and we recalculate the coordinates of the data points in this new basis.
More specifically, the data set is represented as a matrix X with $n \times q$ entries, where n is the number of data points, and q the number of features. Usually, the first step in the algorithm is a pre-processing step that makes sure that the mean of each column in this table is equal to 0 [1]. This step results in the matrix X'. This way, the points are centered

at the origin. Depending on the type of PCA (covariance or correlation) we want to calculate, there are additional pre-processing steps we need to compute [1]. For example, the covariance matrix of the matrix with centered columns X' is given [14] by

$$X_c = \frac{1}{n-1}(X')^\top X'.$$

To acquire the principal components, we need to evaluate the singular value decomposition of the centered matrix. Then, a matrix F is defined as the product of the matrix, containing the left singular vectors U, and the matrix containing the singular values Σ. F holds the factor scores of the data. Each factor score represents the recalculated new coordinate of a data sample for a feature in the new basis [1]. Note that if we multiply the original matrix and the matrix containing the right singular vectors V, we obtain the matrix F [1]:

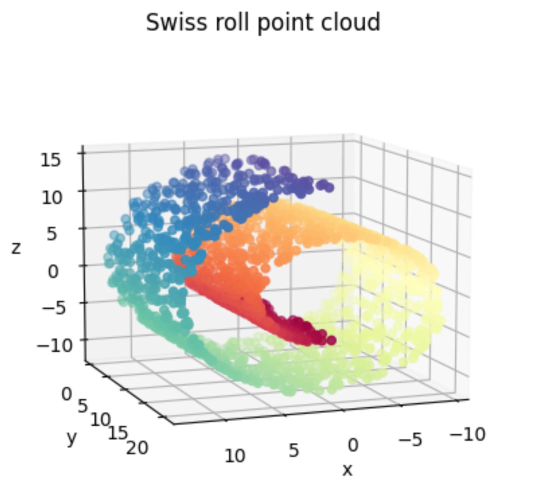
$$X'V = FV = U\Sigma.$$

Consequently, the matrix V can be interpreted as the mapping matrix, and thus the right singular vectors represent the principal components. The square of their corresponding singular values, divided by $(n-1)$ signifies their variance [14].
If we want to acquire the coordinates of the data in a lower dimension, we can accordingly only use a subset of the singular vectors. We simply choose the principal components with the largest variance to obtain the matrix V'. After that, we use the selected vectors to compute the matrix F', such that

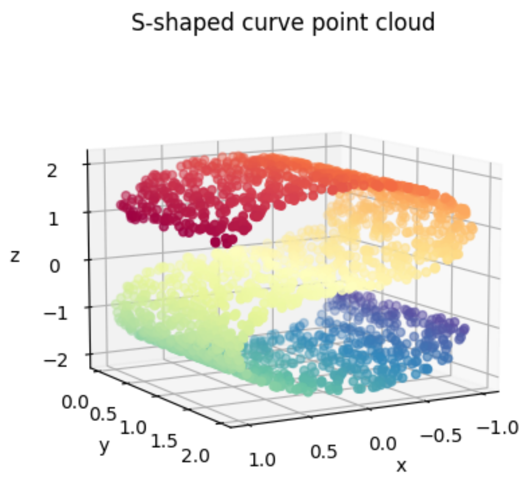
$$F' := X'V'.$$

Note that if we use the covariance matrix X_c, we can perform the eigendecomposition of X_c instead of the singular value decomposition (SVD) of X', since the two are linked [14]. In that case, the eigenvectors are equal to the right singular vectors, hence, they represent the principal components. Their corresponding eigenvalues stand for their variances.
The computational complexity of PCA is in $\mathcal{O}(q^3)$ [5]. This means that the runtime grows with the number of dimensions. Advantageously, PCA does not need any parameters other than the data set itself to infer the new coordinates of the points. In addition, when we order the principal components by their variance, we can easily choose which ones to use in the lower dimension, as the principal components with larger variance bring in more information about the data. Moreover, each principal component contributes a new direction of variation, unless it has a singular value (eigenvalue for covariance PCA) of 0. Hence, the true dimensionality of the data is equal to the number of principal components with non-trivial variances. Furthermore, out-of-sample extension is a straightforward task: we simply use the principal components (as a mapping matrix) to acquire the projections of a different (accordingly pre-processed) data set [1].
A drawback of PCA stems from it being a linear algorithm, as it provides a new, truncated basis using a subset of the singular vectors. Consequently, if the data has a non-linear intrinsic geometry, PCA will often not reduce the dimensions in a meaningful way. The majority of manifolds have in principle, a non-linear intrinsic geometry, as they are "folded"

into space. A classical example that highlights this problem of PCA is the swiss roll data set, see Figure 2.1a. There are points that appear close when measuring their Euclidean distance, which are in reality far apart when measuring their distance along the manifold. This phenomenon can also be observed in the s-shaped curve data set pictured in Figure 2.1b.



(a) The point cloud of the swiss roll data set



(b) The point cloud of the s-shaped curve data set

Figure 2.1: Examples of manifolds

2.1.2 Isomap

This drawback of PCA of projecting data into a linear subspace is addressed by non-linear algorithms. Manifold learning algorithms provide non-linear approaches at dimensionality reduction by assuming that data lies on a low-dimensional manifold.
One such algorithm is Isomap [22]. The idea of Isomap is to measure the geodesic distances between the points so the geometry of the underlying manifold can be preserved. Since manifolds locally resemble a Euclidean space, for a small neighbourhood of points, the Euclidean distance is a sufficient approximation of the geodesic distance. However, globally the Euclidean distance does not represent the geodesic distance (as it is evident in the swiss roll example given in the previous subsection). There are two ways of defining the ”small neighbourhood” around the point: by k that represents the number of nearest neighbours of the data point or by ϵ that depicts the neighbourhood radius around the data point. The estimation of the geodesic distance between points that are not close is equivalent to the shortest-path problem in a weighted graph. The vertices of the said graph are accordingly the data points themselves. The edges of the graph are connecting neighbouring points,

and their weights are the Euclidean distance between the points. Consequently, we can use Dijkstra's or Floyd's algorithm to find the shortest path between all pairs of vertices in the graph [5].
The last step of Isomap is to embed the data into a lower-dimensional space. The goal of the algorithm is to find points in the low-dimensional parameter space, that have Euclidean distances that are close to the geodesic distances in the ambient space [5]. This is done by applying classical Multidimensional Scaling (MDS) to the graph distances. Note that MDS is a dimensionality reduction algorithm that can be used on its own. The classical version of MDS is equivalent to PCA when the distances, that we wish to preserve in the new coordinate system, are Euclidean distances [1]. Since this is not the case in the Isomap algorithm, we do not compute the PCA. First, we need to double center the matrix containing the geodesic distances between all pairs of points. Next, we compute the eigendecomposition of the resulting matrix. All negative eigenvalues are then set to 0, resulting in the diagonal matrix Λ_+. We finally calculate the matrix $X := U\Lambda_+^{1/2}$, where U contains the eigenvectors. It is assumed that eigenpairs are ordered in a descending manner with respect to the eigenvalues. To obtain the coordinates in the lower dimension q', we simply keep the first q' columns in the matrix X [5].
The computational complexity of the algorithm is $\mathcal{O}(n^3)$, where n is the number of data points. Isomap is shown to perform better than PCA on some non-linear data sets by either retrieving the true structure of the underlying manifold or by identifying a non-linear structure of an unknown manifold [22]. Another advantage of Isomap is that by using MDS, it can detect the underlying dimensionality of the data that corresponds to the number of non-zero eigenvalues [5]. With the increase of the number of data points, Isomaps yields results that resemble the true structure of the manifold more accurately. The intuition behind this is that the geodesic distance between neighbouring points can be more accurately represented by the Euclidean distance when the points are geodesically very close. This happens when there is an abundance of points. With the number of points approaching infinity, the true geometry of the manifold can be captured almost perfectly.
As with most algorithms, there are some setbacks associated with Isomap. It has an optimality guarantee only under certain assumptions [5], e.g, the manifold has to be compact. A major drawback of Isomap is that it is topologically unstable [2]. The step that approximates the distances between neighbouring points via the Euclidean distances can result in "short-circuits". This means, that if the parameter k or ϵ has not been chosen properly, the same problem that Isomap tries to solve might occur: a point that is far away along the manifold might be interpreted as a neighbour. An error at such an early stage of the algorithm has drastic effects in later steps and accordingly leads to misleading results. Because of the same reasoning, Isomap is very sensitive to noise, as noise can make folds appear "closer" than they are in reality. Setting ϵ to a very small value might not be of an advantage, as it can fragment the manifold [2]. Consequently, the parameter k or ϵ needs to be chosen carefully, such that density and noise are taken into account. Furthermore, out-of-sample embedding is not a straightforward task, and there is no direct approach

that tackles it. A proposed solution [3] is using the Nyström extension.

2.2 Diffusion Maps

Diffusion Maps [7, 9] is another manifold learning technique. When compared to the methods discussed above, it does not directly rely on Euclidean or geodesic distance. Instead, it requires a connectivity (or affinity) score between pairs of data points. This connectivity has to be stored in an affinity matrix, that when normalized represents a transition matrix between all pairs of points. In many cases, the affinity is computed through a Gaussian kernel, with the local Euclidean distances between pairs of data points, as discussed below. We can define a Markov random walk on the transition matrix. By diffusing over this matrix, meaning we perform the walk for a number of time steps, we can identify important properties about the geometry of the underlying manifold.
To be able to compute the proximity between two points, we need to first choose a kernel function that is symmetric and that yields positive values for all inputs. Oftentimes, the chosen kernel is the Gaussian kernel $K(x_i, x_j) = e^{-\frac{\|x_i - y_j\|^2}{2\epsilon}}$. The parameter ϵ is the square of the bandwidth and intuitively represents the radius of the relative neighbourhood around a point, similarly to ϵ in Isomap. It is also referred to as the kernel scale. Using the kernel, we can capture the *local* affinities and thus, acquire a notion of the local geometry of the data. The resulting affinities are stored in a square matrix, that can be interpreted as the matrix containing the edge weights of a graph. We refer to this matrix as the *affinity matrix* or *kernel matrix*.
The normalization in Diffusion Maps consists of two steps. In many cases, the density of the data is not uniform on the underlying geometry. Therefore, using a data set with densely distributed regions and other areas, that have a sparse distribution, might result in misleading approximations of the geometry. Diffusion Maps provides an elegant way to remove the influences of the density of the data by normalizing the matrix with α. The normalized affinities then have the form

$$W_{ij}^{(\alpha)} := \frac{W_{ij}}{D_{ii}^{\alpha} D_{jj}^{\alpha}}.$$

In the formula above, W_{ij} signifies the affinity between the points i and j that are contained in the affinity matrix W. The elements in the diagonal matrix D correspond to the degree of each point (vertex), as they are defined as the sum of all affinities related to the point with index i: $\sum_{j=1}^{n} W_{ij}$. We need to accordingly recalculate the degrees so they match the new affinity matrix. To do so, we replace the non-normalized affinities in the sum with the ones normalized by α. We denote the new degree matrix by $D^{(\alpha)}$. The value of α is a number between 0 and 1 with three notable cases [7]. When $\alpha = 0$ the impact of the density is maximal, no normalization is performed. We are then reduced to the graph Laplacian case. The value 0.5 results in the Fokker-Planck equation. If $\alpha = 1$, then we can approximate the Laplace-Beltrami operator, and (ideally) the effect of the data density is removed. The third

case is arguably the most important one under the manifold assumption, as we obtain the underlying geometry of the manifold.
After we make sure that the density of the data does not influence the results (or influences them as much as we want to) we need to proceed with the random walk. To be able to define a random walk on our data, we need to transform our matrix in the *transition* matrix

$$P_{ij} := \frac{W_{ij}^{(\alpha)}}{D_{ii}^{(\alpha)}},$$

which contains the probability of transition from node i to node j. The resulting transition matrix is row-stochastic and in contrast to the affinity matrix, not symmetric. The probability of transition in this matrix is for a single time step. As a side note, if we want to obtain the transition probability between all pairs of points for t time steps, we need to calculate P^t. This is the intuition behind the ”connectivity” of the data: how probable it is to reach a data point from another point in a given number of time steps by taking all possible paths into account. Since the transition matrix is derived from the kernel matrix, it implicitly contains some information about the local geometry of the data set. Because we are considering a certain number of time steps we are *diffusing* over the data points. By running the random walk forward, we can uncover the geometry of the data on different scales [7].
The next step of the algorithm is to perform the eigendecomposition of the transition matrix. Using the obtained eigenvectors $\psi_k(i)$ and eigenvalues λ_k we can embed the data into a lower dimension q':

$$x_i \mapsto [\lambda_2^t \psi_2(i),\ \lambda_3^t \psi_3(i),\ \dots,\ \lambda_{q'+1}^t \psi_{q'+1}(i)].$$

Here, we assume that the eigenvalues are in descending order, and the lower index signifies the place of each eigenvalue in the ordered sequence. The eigenvectors correspond to the eigenvalues with the same index. The eigenvalue with the largest value (and subscript 1) is trivial, and as such, the eigenpair to which it belongs is not considered for the embedding. The Euclidean distance between two points in the space where they are embedded is close to the diffusion distance in the ambient space that is defined [7] as

$$D(x, y) = \left(\sum_{l>1} \lambda_l^{2t} (\psi_l(x) - \psi_l(y))^2 \right)^{\frac{1}{2}}.$$

Because the sum is specified over all paths between node i and node j with the length t, Diffusion Maps is more robust against short circuits, which are one of the major issues related to Isomap. Note that in the sum above we exclude the first trivial eigenvalue and its eigenvector.
Similarly to Isomap, the computational complexity of Diffusion Maps (with no optimizations) is $\mathcal{O}(n^3)$ with n being the number of data points [23]. The most computationally

expensive process is again the eigendecomposition. Diffusion Maps requires in principle two hyper-parameters: the kernel scale ϵ and the diffusion time t. There are kernel functions that do not require a scale, in which case the algorithm needs one parameter at best. Note that some of these kernel functions might require other parameters, e.g. the Continous k-Nearest Neighbour kernel [4]. Furthermore, an out-of-sample extension cannot be computed directly, instead, methods such as the Nyström extension can be used [9].
The Nyström extension is a technique originally proposed to speed up spectral segmentation methods that employ an affinity matrix [10]. The extension follows the idea that the number of output clusters of the provided data is much smaller than the data points received as input. As such, it takes a small random sample from the input data set and runs the clustering algorithm on it, after that it extends it to the rest of the data. Using this extension, the affinity matrix can be approximated.
In the context of Diffusion Maps, to extend the existing embedding to an unseen data set, the eigenvectors of this unseen data set have to be calculated by taking into account the similarities between the already embedded points and the new points [9]. Then, the new eigenvectors have the form

$$\Psi_{new} = K(\mathcal{X}, \mathcal{Z})\Psi_{old}\Lambda^{-1}.$$

Here, Ψ denotes the eigenvectors, Λ the eigenvalues. The "old" eigenvectors represent the eigenvectors we acquire after the fitting step of the Diffusion Maps algorithm, the "new" eigenvectors belong to the unseen set $\mathcal{Z}$. $K(\mathcal{X}, \mathcal{Z})$ is the component-wisely calculated and normalized kernel matrix, representing the (normalized) affinities between the embedded data set $\mathcal{X}$ and the new data set $\mathcal{Z}$. Each row j in this matrix contains the proximity of all embedded points to the j-th new point in $\mathcal{Z}$.
A further issue, associated with algorithms based on the analysis of the Laplace operator, is the selection of the eigenpairs used for the embeddings [8]. Additional eigenvectors might not provide new directions in the geometrical representation of the data. Thus, selecting the first q' eigenpairs does not guarantee a meaningful unfolding in the dimension q'. A proposed solution to address this problem, (apart from visually comparing possible embeddings) is to use local linear regression [8]. The overall idea is to fit a function to each eigenvector. The arguments of this function are the previous eigenvectors. If the function approximates the eigenvector well, it is assumed that this eigenvector does not contribute a new direction to the embedding and can be consequently omitted. The function is a local linear function and we estimate its arguments α and β for each point in the data set by solving an optimization problem [8]. In this problem a kernel function is used that has a kernel scale which is usually governed by the median of the distances between each pair of eigenvectors up to the one we are trying to approximate. After that, we calculate the residual by computing the normalized cross-validation error [8]. The residual represents how well we can approximate the eigenvector with the estimated local linear function. Since it is an error term, a small value indicates that the function approximates the eigenvector well. Conversely, a large value means that the eigenvector cannot be approximated by the previous ones. Therefore, when the residual is big, the eigenvector contributes a

new direction, and we can include it when embedding the data. The residual of the first eigenvector is set to be 1. When we have knowledge of the intrinsic dimension q' of the data, we can simply use the q' eigenvectors with the largest residual values. Otherwise, we can specify a threshold value. If an eigenvector has a residual larger than the threshold, we select it for the embedding.

An alternative to the standard Gaussian kernel and other kernel functions that compute the proximity between data points is provided by the Continuous k-Nearest Neighbors (CkNN) graph construction [4]. The idea of the algorithm is to switch the weighted graph represented by the affinity matrix with an *unweighted* graph aimed to depict the topology of the manifold. The matrix in the CkNN scheme is therefore an adjacency matrix. Data points are again interpreted as vertices, and then it should be decided which points have to be connected by an edge. Some standard schemes connect a point to its k nearest neighbours, others all points within a region with the radius ϵ. Depending on the density of the data, said strategies might fail [4]. The proposed approach of CkNN is instead to connect all points for which

$$d(x, y) < \delta\sqrt{d(x, x_k)d(y, y_k)}$$

holds, where d represents the distance between two points. The parameters k and δ are specified by the user. However, k can be fixed to a constant, and thus only the unitless continuous parameter δ should be accordingly adjusted, typically in the range (0,1]. The (unnormalized) graph Laplacian constructed by the CkNN scheme converges to the Laplace-Beltrami opearator when $n \to \infty$ and $\delta \to 0$. This consistency to the Laplace-Beltrami operator holds not only for compact manifolds but also for non-compact ones. Recall that compactness is a requirement for the optimality guarantee of Isomap [5]. This makes CkNN suitable for the task of manifold learning. Naturally, an advantage of this approach is the bigger set of potential manifolds on which it can be applied. The choice for δ is however not obvious. A potential solution is proposed in [4].

2.3 Datafold

Datafold [16] is an open-source Python package that provides data-driven models in the context of manifolds and dynamical systems. The package has a dedicated webpage that includes the needed information to get familiar with the package, take a deeper dive into the specification, and contribute to it, as pictured in Figure 2.2. Datafold has a three-layered structure, where each layer represents a level of abstraction. The structure forms a hierarchy, such that lower levels provide services and can be used in higher levels. Additionally, the modules in each layer can be used individually in an application that includes datafold as a dependency.

The lowest level `datafold.pcfold` lays the basis of datafold, as it includes mainly data structures and algorithms that can be used in more than one models. The two provided data structures are representative of the goals of the machine learning package. The class `PCManifold` signifies a point cloud. This definition closely corresponds to the concept

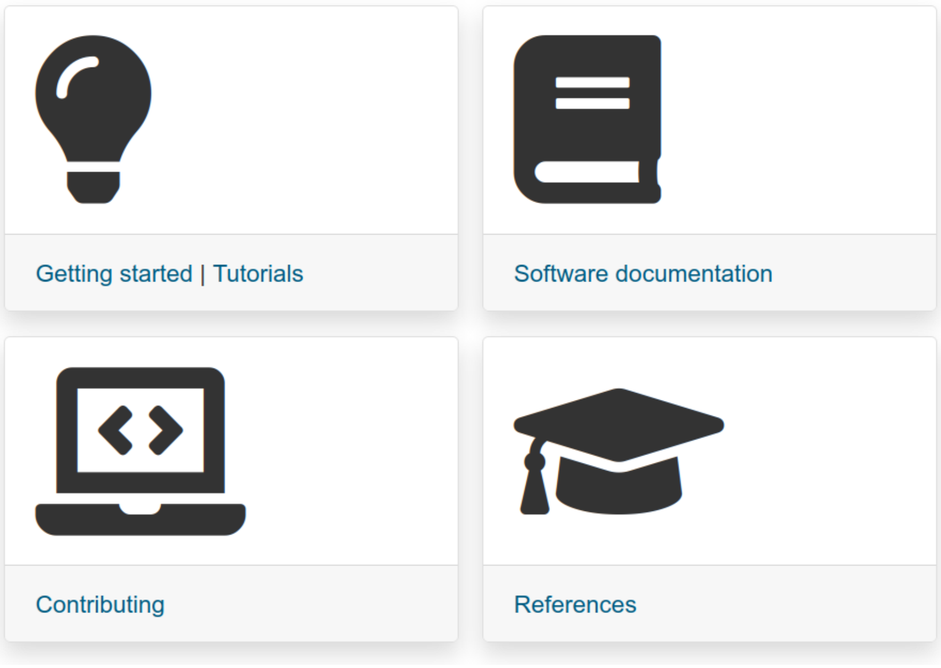


Figure 2.2: The home page of the website of datafold [16]

of a manifold, since it entails a geometry and a proximity measure between the points. `PCManifold` inherits from `ndarray` provided by NumPy [12]. This corresponds to the intuitive representation of a point with q features as a q-dimensional array. This array is contained in the point cloud under the `data` attribute. In addition to the data itself, a kernel associated with this data is attached to the point cloud. The kernel specifies the proximity measure that is used on the specific point cloud. A notable property that is part of the keyword arguments `dist_kwargs` of the class is the `cut_off`. It represents the value over which the proximity is considered to be irrelevant, and is treated as zero. Setting a non-infinite value for the `cut_off` makes the execution of a manifold learning algorithm less computationally expensive because it can disregard a big portion of the entries in the kernel matrix. The point cloud class contains methods that can be used for the calculation of the distance or the kernel (affinity) matrix. The method `optimize_parameters`, which is also a part of this class, returns estimations for the kernel scale and the `cut_off`. That is an important aspect, since we can use the approximated value for implementations of manifold learning techniques with a Gaussian kernel that also require knowledge of the bandwidth. Thereafter, we would only need to provide the time exponent. Another method that can prove useful is `pcm_subsample` that returns a uniformly distributed subsample of the given point cloud. This function is of particular importance for algorithms that cannot remove the impact of the density of the data.

The other main data type specified in datafold is a collection of time series. Time series allow the representation and analysis of dynamical data. It is assumed that the phase space of the time series data lies on a manifold [16]. Datafold provides the class `TSCDataFrame` that inherits from `DataFrame`, which is implemented by pandas [25]. `TSCDataFrame` represents a *collection* of time series data. Therefore, it has an index that contains two records: one that indicates the ID of the time series and one that represents the time. Accordingly, the first value must be an integer, the second one a non-negative number. The attribute `tsc_feature_col_name` contains the name of the features. A further requirement is that the feature names and the time series ID-s are unique. A number of methods are provided that handle time series or create a `TSCDataFrame` from a different data representation. Similarly to point clouds, time series data frames also have a kernel and allow the computation of the distance and the kernel matrix.
There are several different kernels implemented in datafold. All kernels are part of the lowest layer and derive from the base class `BaseManifoldKernel` that inherits from `gaussian_process.kernels.Kernel` class provided by Scikit-learn [19]. The kernels acting on point clouds inherit from `PCManifoldKernel`. These kernels provide functionality that computes the kernel matrix which contains the interpoint proximities. Some examples of kernels inheriting from `PCManifoldKernel` are the `GaussianKernel`, the `ContinuousNNKernel` [4], and the `DmapKernelFixed` that wraps another kernel and provides means of normalizing it, as described in section 2.2. Time series data can work with both a `PCManifoldKernel` and a `TSCManifoldKernel`. In the former case, time information is not taken into consideration. A class that derives from `TSCManifoldKernel` is the class `ConeKernel` [11].
The package `dynfold` builds on top of `pcfold`. The models implemented there can either be used in the top layer of datafold, or on their own in a different application. There are two main classes that extend the `BaseMixin` class and build the basis for the models. The first one, `TSCTransformerMixin` extends additionally the Scikit-learn class `TransformerMixin` [19]. In doing so, it supports not only array-like data sets, as the superclass it extends, but also time series data representations. When a class inherits from `TransformerMixin` it has to usually override the method `fit_transform`. This method fits the data and then transforms it in some way, specified by the subclass. The second base class is the `TSCPredictMixin`. It provides two methods that have to be overridden: `predict`, and `reconstruct`. Transformer classes contained in the package include `DiffusionMaps` that transforms points from one coordinate system to another one, `TSCPrincipalComponent` that extends the implementation of PCA in Scikit-learn to time series data representations, and `TSCApplyLambdas` that transforms the data set by applying a Lambda function to each element. As it is evident in these examples, the implemented transformation may vary, above all, in its goal.
The package that represents the highest layer in the architecture of datafold is the `appfold` package. Here, the separate modules contain implementations of specific machine learning algorithms. These implementations can use the data structures and base algorithms from `pcfold`, as well as the models from `dynfold`. Note, that since all transformer

classes implement the method `fit_transform` they can be chained in a pipeline, and this pipeline can be used in the implementation of some machine learning algorithms. Diagrams depicting the inheritance structure are provided for all three packages in the documentation of datafold [16].
The package `utils` is also worth mentioning, as it provides utility functions that target a specific task that does not fit in the layered structure, and is mostly thought to be for internal use. The functions range from methods that are used for plotting, to functions that verify the presence of a certain property of a given value. For example, if the given argument is a floating number. Additionally, there are functions that perform operations on matrices, such as multiplication with a diagonal matrix. The functions that provide plotting capabilities are contained in a separate module `plot.py`. Each package (including `utils`) accommodates a directory dedicated to tests. Each module is tested with the provided in this directory test modules.
The Diffusion Maps algorithm is implemented in the middle layer, the `dynfold` package. In addition to dimensionality reduction, the model can be used to approximate the eigenfunction of the Laplace-Beltrami, Fokker-Plank, and graph Laplacian operators. This is achieved internally by returning an instance of `DiffusionMaps` with the appropriate value for the attribute `alpha`. As mentioned above, the class derives from the class `TSCTransformerMixin`, and overrides the `fit_transform` method. Additionally, it inherits from `BaseEstimator` from Scikit-learn, and it provides its own implementation of the `fit` method. Another method that is specified by `TransformerMixin`, and overridden by `DiffusionMaps` is the `transform` function. After creating a `DiffusionMaps` instance, we can call all those three functions on it. The `fit`-method estimates the eigenpairs of the normalized kernel matrix. We can access them as attributes associated with the instance. After calling `transform`, we obtain the new coordinates of the data set provided as input, meaning we can additionally use it for out-of-sample extensions. Naturally, a data set has to have been fitted first. The algorithm used for the out-of-sample extension is the Nyström extension [9]. The method `fit_transform` performs the estimation of the eigenpairs and the embedding of the fitted data set one after the other. Furthermore, the implementation of Diffusion Maps in datafold lets the user perform an inverse transformation from the parameter space into the ambient space. Since it implements all those methods, `DiffusionMaps` can be used in a pipeline, and thus, it can be easily integrated into the layer above (or into a different application). Among others, a useful property of the implementation is the provided `symmetrize_kernel` attribute. Recall that the normalized kernel matrix (transition matrix) is not symmetrical. This can lead to numerical instabilities when performing the eigendecomposition [16]. Consequently, when the value of `symmetrize_kernel` is set to `True`, a symmetric conjugate of the matrix is instead computed and used.
If we wish to utilize the CkNN scheme described in section 2.2, we need to set the kernel to `ContinuousNNKernel` when creating the `DiffusionMaps` instance. Note that if we additionally set the `cut_off` to a non-infinite value, all points must have at least `k_neighbor` number of neighbours, where the attribute is specified at the time of cre-

ation of the instance. Since this fact poses a challenge, we can work around it by calling the function `pcm_remove_outlier` from the `pcfold` package on the point cloud. This method removes all points from the set that do not have the specified number of neighbours. The required number of neighbours is passed as the argument `kmin`.
The class `LocalRegressionSelection` is provided in the same module as the implementation of Diffusion Maps and it aims to solve the issue related to the selection of eigenpairs for the embedding. The implemented approach is the one discussed in section 2.2 using local linear regression [8]. It is implemented as an estimator and a transformer. In the fitting step, it infers the indices of the eigenpairs that best unfold the manifold by passing the eigenvectors approximated by Diffusion Maps as an argument. This is done, as discussed in section 2.2, by estimating the residual values for each eigenvector. The indices of the selected eigenvectors are stored in the class attribute `evec_indices_`. The strategy for the selection of the eigenpairs can be specified by the user. There are two potential strategies: either `dim` or `threshold`. The former is associated with an additional argument `intrinsic_dim` that specifies the underlying dimension of the manifold. Then, the number of eigenvectors selected is equal to the value of `intrinsic_dim`. This way, the indices of the eigenvectors with the largest residuals are chosen. Conversely, if the strategy `threshold` is used, then all eigenvectors with a residual bigger than the specified value of the attribute `regress_threshold` are included in the selection. When `transform` is called and all eigenvectors are passed as an argument, the eigenvectors whose indices were selected in the fitting step are returned. We can then use them to perform the embedding.

2.4 Roseland algorithm

Shen, Chao and Wu, Hau-Tieng proposed a new method of dimensionality reduction based on probability-driven spectral embedding in 2020 [21]. In their paper, they introduce the Roseland algorithm (RObust and Scalable Embedding via LANdmark Diffusion) as a less accurate, but computationally faster alternative to Diffusion Maps. They examine its behaviour and show that the algorithm fits into the manifold setup. Additionally, they prove the spectral and pointwise convergence to the Laplace-Beltrami operator which leads to Roseland being a suitable candidate for manifold learning.
In this thesis, we build on their work to implement Roseland in the datafold package. We aim to reproduce their observations about scalability and robustness by testing both Diffusion Maps and Roseland under the same setups and comparing the results.

3 Scalable manifold learning through landmark diffusion

In this chapter, we give an overview of the Roseland algorithm. We discuss different methods of constructing one of the main aspects of the novel algorithm, namely the landmark set. Taking all the theoretical considerations into account, we implement the algorithm in the datafold package, and we describe our approach. Finally, we evaluate our implementation by testing the results under different entry conditions. We evaluate its performance by comparing the results to the results obtained after running the Diffusion Maps algorithm under the same settings.

3.1 The Roseland algorithm

Roseland adopts a dimensionality reduction method similar to that of Diffusion Maps. Both algorithms assume data lies on a manifold that is embedded in a higher dimension. Their approach to dimensionality reduction is to perform a spectral embedding based on random walks by using the affinity matrix. However, Roseland does not consider the proximities between all point pairs in the data set. Instead, by using a landmark set, a landmark-set affinity matrix is computed, which contains the affinities between the data points and the landmark points. The matrix is then normalized by the degree matrix of the data set. The resulting matrix is thin, and as such, eigendecomposition cannot be performed on it. Thus, the more general SVD is calculated and used for the final embedding. In the following subsections, we describe the Roseland algorithm [21, 17, 20].

3.1.1 Input and output

Similarly to Diffusion maps, the key input is the *data set*. In Roseland, the data set $\mathcal{X}$ consists of n data points. Each one of these points has q features. Thus, each point has the dimension q.
Unlike in Diffusion Maps, however, there is one more set that is taken as an input in Roseland, namely the *landmark set* $\mathcal{Y}$. There is no requirement on the landmark set that it should be a subset of the given data set, but it should be smaller than the data set. The size makes it possible to reduce the runtime of the algorithm. Like the data set, each point in the landmark set has the dimension q. The landmark set is not a strict requirement as an input because it can be subsampled from $\mathcal{X}$. We discuss this variation in a later section dedicated to the landmarks.

The *kernel* function can also be provided as an input. By default, the Gaussian kernel is employed. The kernel function is used to compute the affinity between two points. A requirement for the kernel is to be positive-definite.
Lastly, the (lower) *dimension* q' for the embedding and the *diffusion time* t are plugged in. The output of Roseland is the embedding of $\mathcal{X}$ in the dimension q' computed by diffusing for a time t between the data points by first detouring through $\mathcal{Y}$.

3.1.2 Normalization

The first step of Roseland is to calculate the landmark-set affinity matrix

$$W_{ik}^{(r)} = K(x_i, y_k) \in \mathbb{R}^{n \times m},$$

which contains the similarity between each point x_i in $\mathcal{X}$ and each point y_k in $\mathcal{Y}$. Here, K indicates the kernel function, e.g., the Gaussian kernel $K(x_i, y_k) = e^{-\frac{\|x_i - y_k\|^2}{2\epsilon}}$ with a kernel scale ϵ and $\| \bullet \|$ a distance metric, e.g., the $\mathbb{R}^q$ Euclidean norm. Since $m < n$ the resulting matrix has fewer columns than rows, that is, a thin matrix. Recall that there are n samples in the data set, $\mathcal{X}$, and m samples in the landmark set $\mathcal{Y}$.
For the matrix to define a Markov process it needs to be normalized by its degree matrix. The degree matrix is a diagonal matrix defined as

$$D_{ii}^{(R)} := e_i^T W^{(r)} (W^{(r)})^\top \mathbf{1} \in \mathbb{R}^{n \times n}.$$

Here, e_i is the unit vector, containing a 1 as the i-th element, and zeroes as the remaining elements. The vector $\mathbf{1}$ is the all-ones vector, containing 1 in all entries. The degree is accordingly calculated for each data point in $\mathcal{X}$. It is important to note that the degrees are evaluated from the so-called *landmark-affinity matrix*

$$W^{(R)} := W^{(r)} (W^{(r)})^\top \in \mathbb{R}^{n \times n}.$$

This matrix is similar to the one in Diffusion maps as it represents the similarities between all data points in $\mathcal{X}$. The difference is that the affinities are not the direct affinities, rather the proximity from x_i to x_j by first detouring through the landmark set. If this matrix is multiplied by the reciprocal of the degree matrix it would result in a row-stochastic transition matrix. Therefore, this method conforms to the idea of a diffusion process [17].
The normalized matrix then has the form $(D_{ii}^{(R)})^{-1/2} W^{(r)} \in \mathbb{R}^{n \times m}$. That is, each entry w_{ik}, that represents the affinity between x_i and y_k, is divided by the square root of the degree of the data point x_i.

3.1.3 Fitting

The purpose of the fitting process is to obtain the spectral properties of the normalized landmark-set affinity matrix. Because this matrix is not square, we cannot compute the eigendecomposition. Thus, the singular vector decomposition is used instead:

$$(D_{ii}^{(R)})^{-1/2}W^{(r)} = U\Lambda V^\top.$$

Conventionally, $U \in \mathbb{R}^{n\times n}$ is an orthogonal matrix the columns of which are the left singular vectors. Analogously, $V^\top \in \mathbb{R}^{m\times m}$ is orthogonal and contains the right singular vectors in its rows. $\Lambda \in \mathbb{R}^{n\times m}$ is a diagonal matrix with the singular values as the entries on the diagonal, denoted by $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_m \geq 0$.

3.1.4 Embedding

To embed the manifold in the dimension q', we consider $\sigma_2, ..., \sigma_{q'+1}$ and their corresponding left singular vectors [21]. The *Roseland embedding* of a point x_i from the data sample is defined by

$$\Phi_t^{(R)} : x_i \mapsto e_i^\top \bar{U}_{q'}(L_{q'})^t.$$

This embedding is for the diffusion time t end essentially returns the new coordinates of x_i in the lower dimension q'. $\bar{U}$ is set to be $\bar{U} := (D^{(R)})^{1/2}U \in \mathbb{R}^{n\times n}$, which translates to the left singular vectors scaled by the square root of the degree of their corresponding data point. To acquire $\bar{U}_{q'} \in \mathbb{R}^{n\times q'}$, we take the second through $(q' + 1)$ column vectors in $\bar{U}$. $(L_{q'})^t \in \mathbb{R}^{q'\times q'}$ is a diagonal matrix, containing σ_2^{2t} through $\sigma_{q'+1}^{2t}$ as its diagonal entries. The unit vector is denoted by e_i and again has a single 1 as the i-th entry. In short,

$$x_i \mapsto [\,\bar{u}_{2,i}\,\sigma_2^{2t},\ \bar{u}_{3,i}\,\sigma_3^{2t}, ...,\ \bar{u}_{(q'+1),i}\,\sigma_{(q'+1)}^{2t}\,] \in \mathbb{R}^{q'},$$

where $\bar{u}_{j,i} = \frac{u_{j,i}}{\sqrt{d_{j,j}}}$. In other words, the i-th entry of the j-th left singular vector that corresponds to the j-th biggest singular value, scaled by $\frac{1}{\sqrt{d_{j,j}}}$.
This embedding is similar to the embedding of Diffusion Maps, where the second through $q' + 1$ eigenvectors and eigenvalues of the normalized affinity matrix are used.

3.1.5 Nyström extension

Similarly to Diffusion Maps [7, 9], out-of-sample embedding in Roseland is not a straightforward task, because there is no explicit way to acquire the embedding for unseen points from the already computed SVD of the fitted data set. Again, the approach we choose is the Nyström extension, since it can be applied to SVD-approximation as well [18]. In that case, the extended left singular vectors have the form

$$U_{new} = K(\mathcal{Y}, \mathcal{Z})V\Lambda_{m\times m}^{-1}.$$

As a reminder, V contains the right singular row vectors, and U the left ones. $\Lambda_{m\times m}^{-1}$ represents the $m \times m$ diagonal submatrix that contains all singular values. Note that originally Λ is a thin matrix since we perform the decomposition on a thin matrix. A difference to the formula applied in Diffusion Maps is that we calculate the kernel matrix using the landmarks instead of the data points. The reason for this lies in the dimensions. The right singular vectors have a dimension of $m \times m$, with m being the number of landmarks. Accordingly, the matrix $K(\mathcal{Y}, \mathcal{Z})$ has to have a dimension of $k \times m$, where k is the number of samples in the new unseen set. We will therefore get a result for the new left singular vectors of shape $k \times m$. Thereafter, to align the coordinates of them to each other, we have to multiply this matrix by a newly constructed diagonal matrix that corresponds to $K(\mathcal{Y}, \mathcal{Z})$ that has the same dimensions. This step is equivalent to deriving $\bar{U}$ during the embedding, as described in subsection 3.1.4.

3.1.6 Computational complexity

In [21] the size of the landmark set is defined to be $m := n^\beta$ with $\beta \in (0, 1]$, and n representing the size of the data set. Then, the overall complexity of Roseland is $\mathcal{O}(n^{1+2\beta})$. To understand how the complexity of Roseland is calculated we consider the complexity of each of the algorithm steps.
The calculation of the landmark-set affinity matrix is essentially computing the pairwise kernel function between the points in the two sets. Assuming the computation of the affinity between two points takes only a constant amount of time, this results in a complexity of $\mathcal{O}(nm) = \mathcal{O}(n^{1+\beta})$. The calculation of the diagonal matrix can be divided into three steps. First, computing the sum of each column of the landmark affinity matrix. Second, multiplying each of the m entries $w_{i,j}$ in each of the n rows of the landmark-set affinity matrix by the j-th sum computed in the previous step. This results in the landmark matrix $W^{(R)}$we described in 3.1.2. The third step is to sum each row of $W^{(R)}$ to finally acquire the diagonal entries. Each of the three steps is $\mathcal{O}(nm) = \mathcal{O}(n^{1+\beta})$. Since they are executed sequentially, the total complexity of the computation of the diagonal matrix is also $\mathcal{O}(nm) = \mathcal{O}(n^{1+\beta})$. To normalize the landmark-set affinity matrix, we multiply it by $(D^{(R)})^{-1/2}$. The complexity of this calculation is $\mathcal{O}(nm) = \mathcal{O}(n^{1+\beta})$ as well. When it comes to the complexity of the SVD, assuming an efficient method is used [6] , a complexity of $\mathcal{O}(nm^2) = \mathcal{O}(n^{1+2\beta})$ for a thin $n \times m$ matrix can be achieved. The actual in-sample embedding is $\mathcal{O}(n)$ because it calculates the new coordinates of the n data samples. The term that dominates the rest is the computation of the SVD. Therefore, the complexity of Roseland in the in-sample case is in total $\mathcal{O}(n^{1+2\beta})$.
In comparison, the classic Diffusion Maps algorithm has an overall cubic complexity $\mathcal{O}(n^3)$ due to the eigendecomposition. If a sparse matrix is computed, this term can be improved to $\mathcal{O}(n^{2+\eta})$, where $\eta > 0$ [21]. We can see that Roseland provides a better complexity than both of these cases, assuming the size of the landmark set is much smaller than the size of the data set.
Out-of-sample embedding with the Nyström extension for Diffusion Maps has a complexity of $\mathcal{O}(nd)$ [9], where $d = |\mathcal{Z}|$. Roseland extends the singular vectors of the embedded set to the ones of the new set $\mathcal{Z}$ in a similar manner to Diffusion Maps as we saw in 3.1.5. We still need to compute the new kernel matrix, that contains the pairwise similarities be-

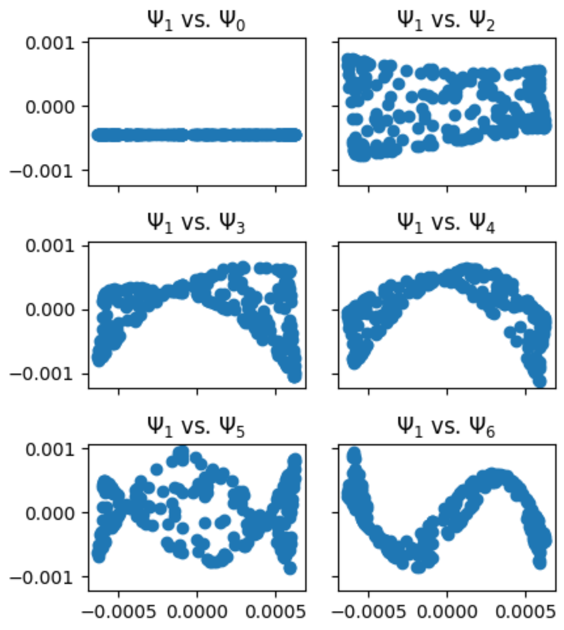
tween $y_i \in \mathcal{Y}$ and $z_j \in \mathcal{Z}$. Thus, the complexity is $\mathcal{O}(md)$. Therefore, we expect Roseland to compute the out-of-sample embedding faster than Diffusion Maps, when $m < n$.
An improvement that can be applied to both algorithms stems from the consideration that we only need the top q' eigenpairs [13], respectively SVD-pairs. Additionally, an SVD for sparse matrices can be computed to further improve the computational costs [15].

3.2 The landmark set

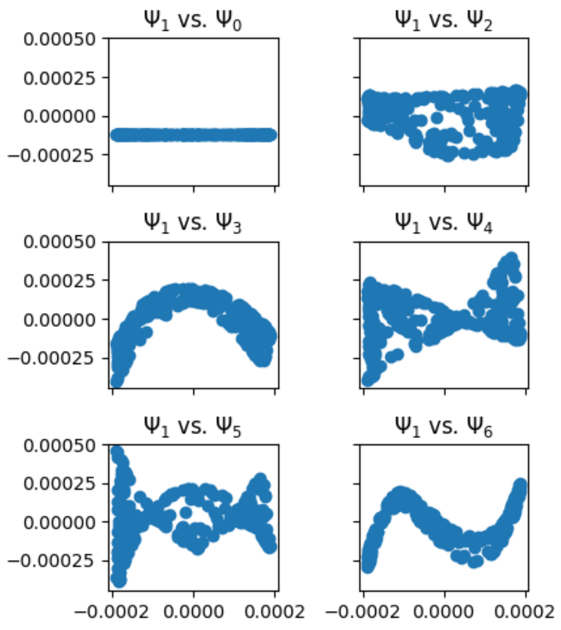
The landmark set is a decisive aspect of the Roseland algorithm. Because the landmark set is assumed to have a much smaller amount of points than the original data set, the resulting affinity matrix is smaller than the kernel matrix of Diffusion Maps, which contributes to the efficiency of the spectral embedding goal. However, some challenges arise because of the landmark set, that need further investigation. In this section, we look into approaches to designing the landmark set.

3.2.1 Acquiring the landmark set

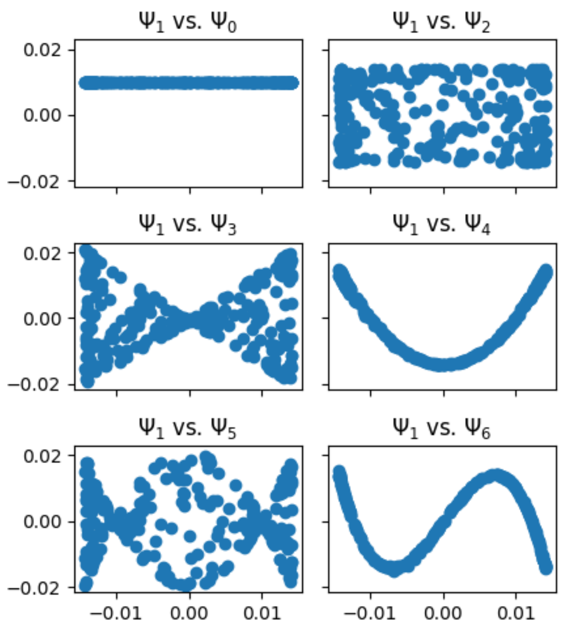
A side of Roseland that might be perceived as disadvantageous is that the landmark set is required as an additional input This leads to the idea that the landmark set can be subsampled from the data set. To still allow the user some control over the landmarks, an additional parameter $\gamma \in (0, 1]$ can be provided. We set the size of the landmark set $m := \gamma n$. The default value of γ is 0.25.
This brings us to the consideration of what the best way would be to subsample the landmarks. One approach is to use a random subsample. Another choice would be to subsample a uniformly distributed subset. To acquire a meaningful comparison, we test the two approaches by using the same data set and the same number of points to be subsampled. Additionally, the manifold we use is a uniformly sampled rectangle with 10 000 data points. To sample the points for the data set we use the function `random.uniform` from NumPy. In Figure 3.1 are pictured the results after fitting the data set. We plot pairings of the singular vectors with the first singular vector with a non-trivial singular value. We additionally plot the results from Diffusion Maps that has already been implemented in datafold. For plotting the embeddings we have adapted the code provided in the tutorials of the webpage of datafold.
In Figure 3.1a the embedding after randomly subsampling $m := \gamma n$ samples from the data set is portrayed. Figure 3.1b shows the uniform variant. To subsample a randomly distributed landmark set from the data set, we use the utility function `random_subsample` provided in datafold. To acquire a subsample with a uniform density, we rely on the function `pcm_subsample` [16]. The function iterates over blocks of points and computes the distance matrix between all the samples and the samples in the current block. Then, it constructs a new point cloud by choosing points from the blocks that satisfy one of two conditions: the point has either no neighbours or the smallest distance of the point to an-



(a) with random subsampling of 2500 landmark points



(b) with uniform subsampling of 2500 landmark points



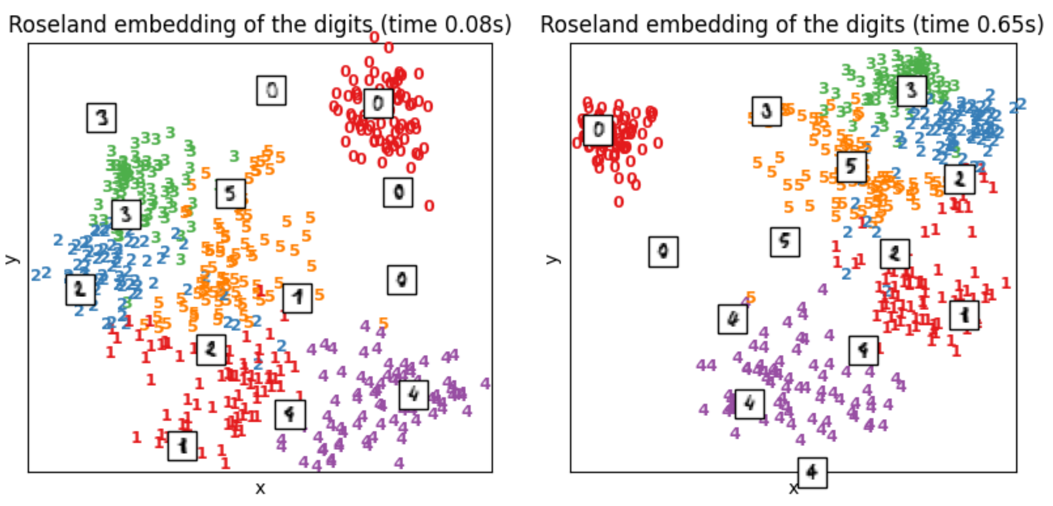
(c) using Diffusion Maps

Figure 3.1: Potential embeddings of a uniformly sampled (3x5) rectangle with 10 000 points

other one is larger than `min_distance`. This way, points from scarcely distributed areas are selected, and densely distributed neighbourhoods bring fewer points into the resulting set. Therefore, the function does not guarantee a size for the returned point cloud. One can tweak the argument `min_distance`. By making it smaller, more points are taken into the new set. Here, we set the argument to two times the `cut_off` of the data set that we acquire after running `optimize_parameters` on it. We can see that there is no improvement in the embedding, but this may be because the original data set is already almost uniformly distributed. Furthermore, depending on the size of the data set, the uniform subsampling can result in significant overhead. Since Roseland aims at efficient embedding, we prefer the random subsampling approach. Random subsampling results in an additional complexity of $\mathcal{O}(m) = \mathcal{O}(n^\beta)$, which does not influence the overall computational complexity of the algorithm.

3.2.2 Choosing the size

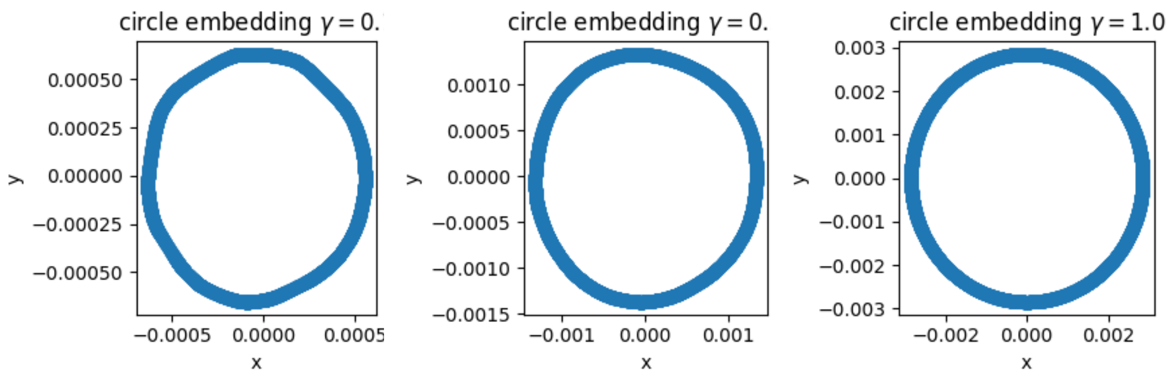
As we discussed in subsection 3.1.6, the size of the landmark set directly affects the computational complexity of Roseland. Since the original idea of Roseland is to provide a scalable spectral embedding technique for large datasets, e.g., $n \geq 10^6$ [21], a small landmark set is ideal to achieve this goal. However, the size of the landmark set is directly related to the convergence rate of Roseland to the Laplace-Beltrami operator: the smaller the set, the slower the convergence. This is in fact one aspect, where Roseland cannot achieve the same results as Diffusion Maps. Even when $n = m$, i.e., the sizes of both sets are equal, Roseland still has a slower convergence rate than Diffusion Maps [21].
Generally, for the goal of clustering the size of the landmark set is not of the highest importance in terms of accuracy. As we can see in Figure 3.2 there is no gradual improvement of the embedding for $\gamma = 1.0$. The most noticeable differences are the rotation and the flipping, which in the context of spectral clustering do not carry high importance. Therefore, one would rather prefer a smaller γ. In the case depicted in Figure 3.2, we see that a γ of 0.1 achieves a much faster embedding. In the two examples, we use the hand-written digits dataset from Scikit-learn. We run all examples on a Linux machine with a 4-core 1.8Ghz i7 CPU and 8GB memory.
However, in the case when we want to acquire an approximation of the Laplace-Beltrami operator and properly display the geometric structure of the manifold in the lower dimension we see differences, as evident in Figure 3.3. For these examples we extract 5000 points from the border of the unit circle as our data set and we run the Roseland algorithm with different entry conditions for γ. In the end, we plot the first singular vector with a non-trivial singular value against the second one using. In all three cases, we subsample the landmark set from the data set and we optimize the parameters for the kernel of the landmark point cloud using the function `optimize_parameters` with its default settings provided in datafold. It is easy to recognize that a larger landmark set improves the approximation.
This comes at the cost of efficiency: Roseland loses all advantages of a smaller kernel ma-



(a) $\gamma := 0.1$, time of fitting and embedding $= 0.08s$

(b) $\gamma := 1.0$, time of fitting and embedding $= 0.65s$

Figure 3.2: Embeddings of the hand-written digits 0-5 with different landmark set sizes by using the first and second left singular vectors with non-trivial singular values



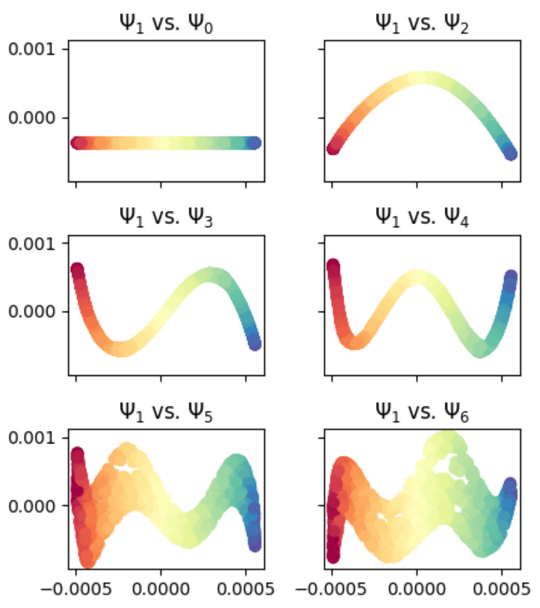
(a) $\gamma := 0.1$

(b) $\gamma := 0.5$

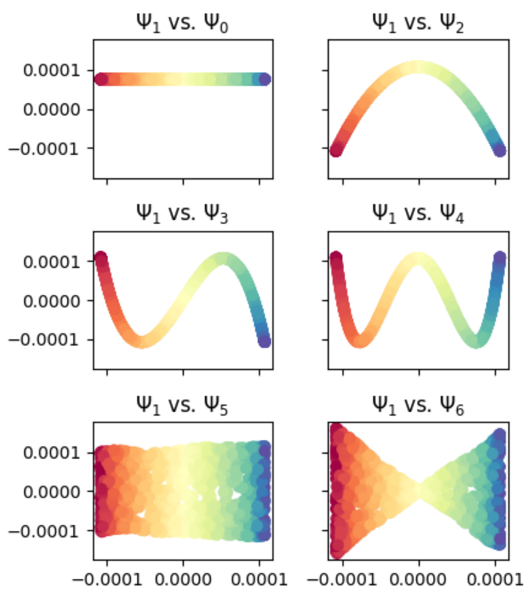
(c) $\gamma := 1.0$

Figure 3.3: Embeddings of the unit circle with 5000 data points and different entry set-ups for γ by using the first and second left singular vectors with non-trivial singular values

trix. As mentioned above, Diffusion Maps outperforms Roseland in terms of the rate of convergence to the Laplace-Beltrami operator. With all these considerations we can conclude that when one aims at a highly accurate Laplace-Beltrami operator approximation, one would still prefer Diffusion Maps over Roseland. By sacrificing some accuracy we achieve lower runtimes. In 3.3a the fitting takes on average 0.25s, in 3.3b 2.48s, and in 3.3c 18.12s.
An additional consideration to make is the size of the data set itself. We work under the assumption that there are enough points in the data set that capture the underlying manifold. The observation that we make is that the more data points there are, the smaller the relative size of the landmark set needs to be to achieve a more accurate unfolding, which in turn yields results faster than both Diffusion Maps and Roseland with a bigger number chosen as γ. As it is evident in Figure 3.4 the larger the data set, the more precisely Roseland unfolds the s-shaped curve provided by Scikit-learn (pictured in Figure 2.1b) with the same fraction chosen for γ. We plot the first non-trivial singular vector on



(a) 10 000 data points, $\gamma = 0.25$



(b) 1 000 000 data points, $\gamma = 0.25$

Figure 3.4: Potential embeddings of the S-shaped curve using a different data set size and the same fraction for γ.

the x-axis against the other singular vectors (0 through 6) on the y-axis by using the utility function `plot_pairwise_eigenvectors` provided in datafold. Because Roseland is aimed at larger data sets, it is sensible to choose a smaller number for the default value of γ. However, we still would want a relatively accurate embedding. These considerations bring us to the decision that γ has a default value of 0.25. The value can be adapted to the size of the data set.

3.3 Implementation

In this section, we describe our implementation of Roseland. We implement Roseland in Python 3.7 in the datafold [16] package. To set up the project it is best to follow the instructions provided in the datafold website under the tab ”Getting started”. This way, all required libraries for Roseland will also be installed and set up. In our implementation, we utilize the libraries NumPy [12], SciPy [24] and Scikit-learn [19], and we use packages, classes and functions provided by datafold. The module, `roseland.py`, resides in the `dynfold` package: the location of all models implemented in datafold.
In Figure 3.5 we illustrate the overall structure of Roseland via a class diagram. We omit

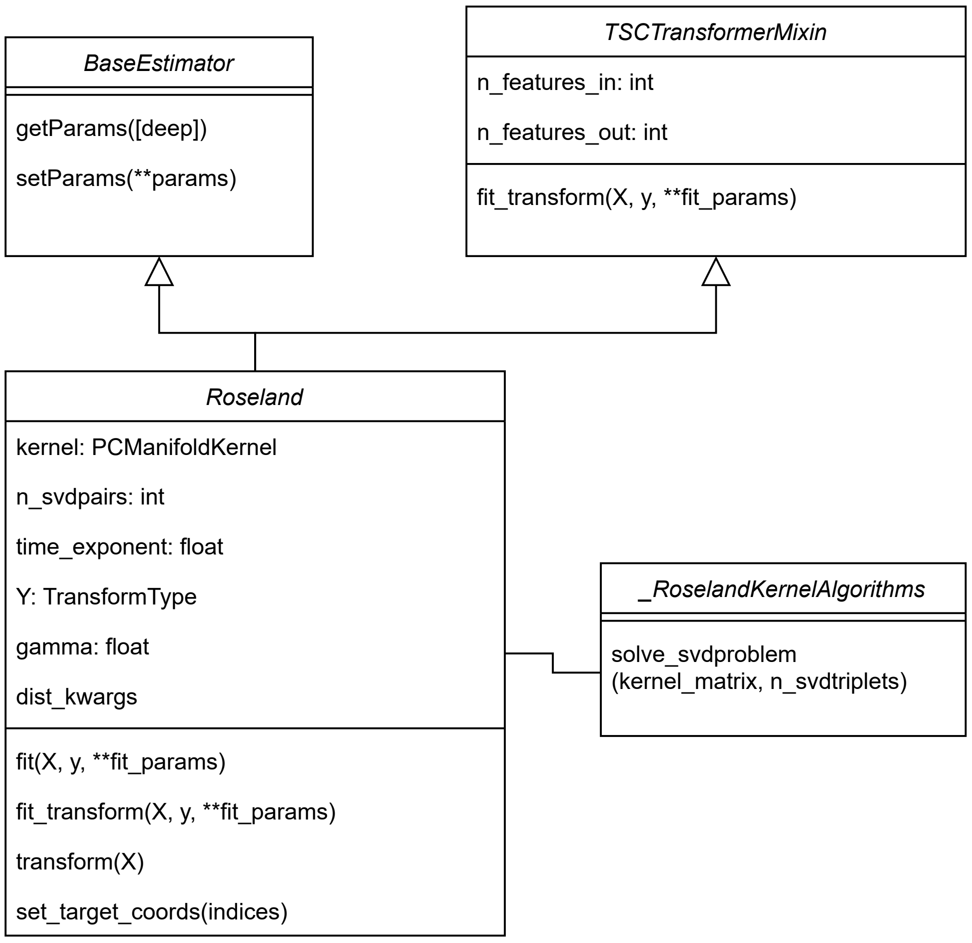


Figure 3.5: Class Diagram of Roseland

some information (methods and class attributes) to keep the diagram readable. In the module, we provide two classes: `Roseland` and `_RoselandKernelAlgorithms`. The latter is an internal class, the methods of which are used in the main class of our implementation, `Roseland`. The current implementation of the internal class contains one method that is responsible for the computation of the SVD, and it accordingly returns the singular values, the left and the right singular vectors.
The main class derives from the classes `BaseEstimator` and `TSCTransformerMixin`. The former is defined by Scikit-Learn, and thus we conform to the guidelines they provide. In summary, an estimator class must implement a method called `fit` that is responsible for fitting the model according to the algorithm. This way, some information about the data input can be inferred. Mapped to our specific case, we infer the singular vectors and singular values of the normalized affinity matrix. By doing that, we are able to use a subset of these spectral properties to reduce the dimensionality of the data. Additionally, we set all the class attributes that do not depend on the data in the `__init__` method, and as required, we define default values for them. The `kernel` is per default set to `None`, and later, during the fitting, set to be a Gaussian kernel with a kernel scale of 1.0, given a different kernel and scale are not provided. Similarly, during the initialization, `dist_kwargs` has the default value `None` and during the fitting, some default values are set, e.g., the `cut_off` is set to infinity. The default value for the number of singular vectors and values to be computed `n_svdpairs` is 10, and the time exponent is with a default value of 0.0. The other two attributes relate to the landmark set. The attribute `Y` is the set itself and it is of type `TransformType` which datafold defines as a union of the two allowed transformation types: `TSCDataFrame` and `np.ndarray`. The current implementation of Roseland supports only `np.ndarray`, but we use the more general type in case of a future extension. The default value for `Y` is `None`. The attribute `gamma` governs the size of the landmark set that needs to be subsampled from the data set, given no landmarks have been provided when creating the Roseland instance. The idea is described in subsecion 3.2.1. Its default value is 0.25. When both of these attributes receive a value upon creating the instance, `gamma` is ignored, and the landmark set is not subsampled. Instead, the provided one is used.
`TSCTransformerMixin` is defined in datafold and it is a subclass of the Scikit-Learn class `TransformerMixin`. As a mixin, `TSCTransformerMixin` provides a specific feature for `Roseland` to inherit, namely the setting of a number of parameters during the fitting. The parameters in question are `n_features_in_`, `n_features_out_`, `feature_names_in_`, and `feature_names_out_`. Because the current implementation of Roseland is limited to only point clouds and does not include functionality for time series, during the fitting, we only set the first two. Since we implicitly derive from `TransformerMixin`, we provide a `fit_transform` method that first fits the data set (finds the singular vectors and values), and then performs the embedding of the input data in the lower dimension. The method `transform` can be used for out-of-sample embedding using the Nyström method as described in subsection 3.1.5. This way, in terms of Scikit-learn terminology, `Roseland` is not only an estimator, but also a trans-

former. Inheriting from these classes and implementing the methods `fit`, `transform`, and `fit_transform` has one crucial advantage, namely pipelining. By utilizing pipelining, we can chain Roseland and other transformers in a pipeline. When the `fit` method is executed on the pipeline, the `fit_transform` method of each transformer is called and finally the `fit` method of the final estimator is executed [19]. This way, Roseland can be performed as a preprocessing step to reduce the dimensionality of the data prior to fitting it by any machine learning model that conforms to the Scikit-learn guidelines. For example, in the code snippet 3.1, we chain Roseland and a logistic regression estimator. We then

```
pipe = Pipeline(steps=[('roseland', rose), ('logistic', logistic)])
pipe.fit(X_train, y_train)
```

Source Code 3.1: Pipeline example

call the `fit`-method on the pipeline. This in turn will call the `fit_transform` method of Roseland and then the `fit`-method of the logistic regression estimator. One can then make predictions on the pipeline itself. In the example `rose` is a Roseland instance, and `logistic`: a logistic regression instance. Both have been initialized prior to defining the pipeline.
The simplest way to create a Roseland instance is by executing `Roseland()`, in which case the default settings for Roseland will be used. When a landmark set is provided, we recommend however running the method `optimize_parameters` from the module `pointcloud.py` in the package `pcfold` on the landmark set first. Then, we can plug in the calculated values for `epsilon` and `cut_off` when creating the instance, as shown in the code snippet 3.2. This way, a kernel scale can be inferred and by having a value for the

```
Y.optimize_parameters()

rose = dynfold.Roseland(
    kernel=pcfold.GaussianKernel(epsilon=Y.kernel.epsilon),
    Y=Y,
    dist_kwargs=dict(cut_off=np.inf)
)
```

Source Code 3.2: Instantiating Roseland

`cut_off` the algorithm is sped up, because the resulting kernel matrix is sparse. In [17] the kernel scale is required as an input, so by doing this we adhere to the requirement by first approximating its value.
In the UML diagram 3.5 are given the three most important methods, when using Rose-

land: they are responsible for fitting and/or transforming the data. We will now follow their execution with the help of a few flow charts.
We start with a flow chart of the fitting process. The steps of the algorithm are displayed in Figure 3.6. The first thing that we check in our implementation is to see if the landmark set

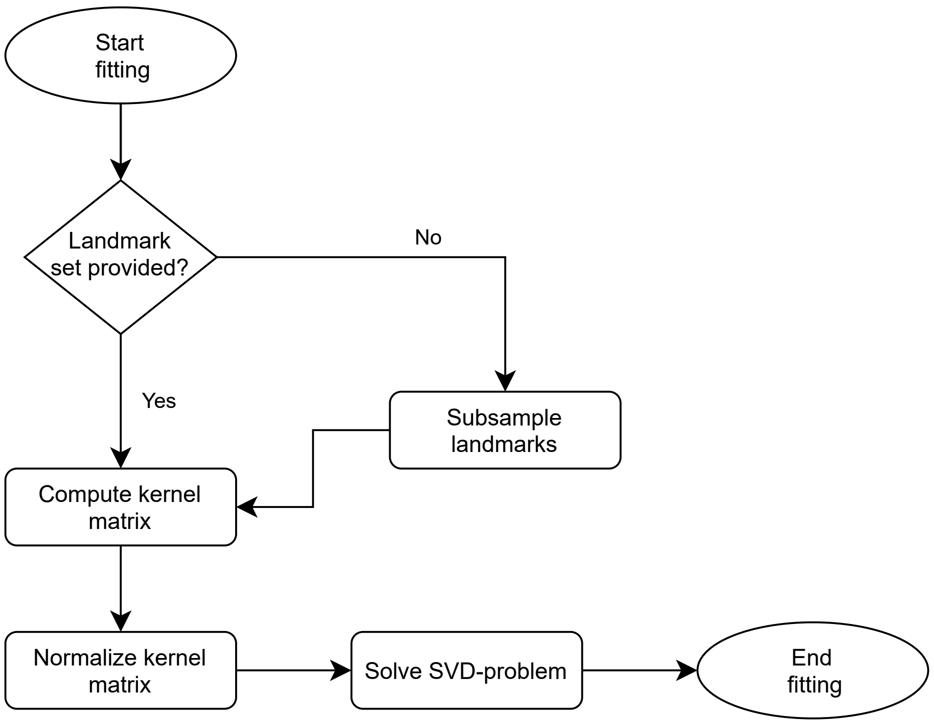


Figure 3.6: Roseland fitting process

has been provided as input, as it is needed to compute the landmark-set affinity matrix. If not, we subsample the landmarks as described in the subsection 3.2.1. After that, we calculate the landmark-set affinity matrix (the *kernel matrix*). To achieve this, we use the function `compute_kernel_matrix` provided in the module `pcfold.pointcloud`. In contrast to Diffusion Maps, we want to compute it component-wisely by using two matrices. Therefore, we also provide a value for the parameter `Y`. The resulting matrix will then be of shape $p \times r$, where p is the number of samples in the parameter `Y`, and r the number of samples in `self`. Since our goal is to compute the required thin matrix we call the specified function on the landmark set point cloud, and pass the data set point cloud as the parameter.
The next step in the fitting process is to normalize the kernel matrix, as described in 3.1.2. To achieve this, we first compute the degree matrix. We mentioned above that when we optimize the parameters of the landmark set before creating the `Roseland` instance, as shown in the code snippet 3.2, the resulting kernel matrix will be sparse. The type that of

this sparse matrix is then `scipy.sparse.spmatrix`. Since there are usage differences between this type and `numpy.ndarray`, which we obtain in the case of a dense matrix, the steps to compute the diagonal matrix slightly differ. The idea is however the same: we calculate the sum of each column, then we multiply the kernel matrix component-wisely by the `column_sums`. After that, we calculate the sum of each row. Squaring each element in the resulting array results in the diagonal matrix. As a side note: since a diagonal matrix has only one non-zero element per row, we do not need a whole matrix (`numpy.ndarray`) to represent it. The final step in this process is to transform the elements into their reciprocal. Because some values in the diagonal can be zero or approaching zero, the inverting of the digonal might result in an infinite number or in overflow. Therefore, we first suppress the warnings, with which NumPy provides us, by using `with np.errstate()`. Then, we set the values that resulted in `inf` or overflowed (meaning they are now negative) to 0. In the code snippet 3.3, we provide the implementation in the case of a dense matrix.

```
column_sums = np.sum(kernel_matrix, axis=0)
new_kernel_matrix = np.multiply(kernel_matrix, column_sums)
normalize_diagonal = np.sqrt(np.sum(new_kernel_matrix, axis = 1))

with np.errstate(divide="ignore", over="ignore"):
normalize_diagonal = np.reciprocal(
        normalize_diagonal, out=normalize_diagonal)

bool_invalid = np.logical_or(
    p.isinf(normalize_diagonal), normalize_diagonal < 0)
normalize_diagonal[bool_invalid] = 0
```

Source Code 3.3: Computing the degree matrix: dense matrix case

The final step of the fitting is to solve the SVD problem for the normalized kernel matrix. To do this, we use the function provided by SciPy `svds` that returns the singular values and vectors of a given matrix. As required in Roseland, we set the parameter `which` to `"LM"`, which means we want to compute the largest singular values and their corresponding vectors. The number of singular values and vectors that are computed is governed by the argument `n_svdpairs` that is set when instantiating Roseland. According to the documentation of SciPy [24], the order in which the singular values are returned is not guaranteed. Thus, we sort them thereafter in descending order. After that, we rearrange the singular vectors accordingly. In our implementation, we compute both the right and left singular vectors. The reason for this is the potential later usage of the Roseland instance for out-of-sample embedding, for which we need the left singular vectors as stated in subsection 3.1.5. We ignore the imaginary parts of the singular values and vectors, given they are small enough and, therefore, negligible. To finish with the calculation of the

singular vectors, we change their coordinates. This is similar to the normalization of the eigenvectors in Diffusion Maps, where the vectors are normalized by their norm. Here, we rather multiply them by the degree matrix used to normalize the kernel matrix. This step concludes the fitting process. The result is that the singular vectors and values of the kernel matrix associated with the specific instance have been inferred. This means that we have the needed properties to embed the data into a lower dimension.
We now continue with the other main step of Roseland: the transformation process, which in our case is equivalent to out-of-sample embedding. As a reminder, we want to approximate the left singular vectors of an unseen set $\mathcal{Z}$ by using the inferred singular vectors and the landmark points. The transformation process is illustrated in Figure 3.7.

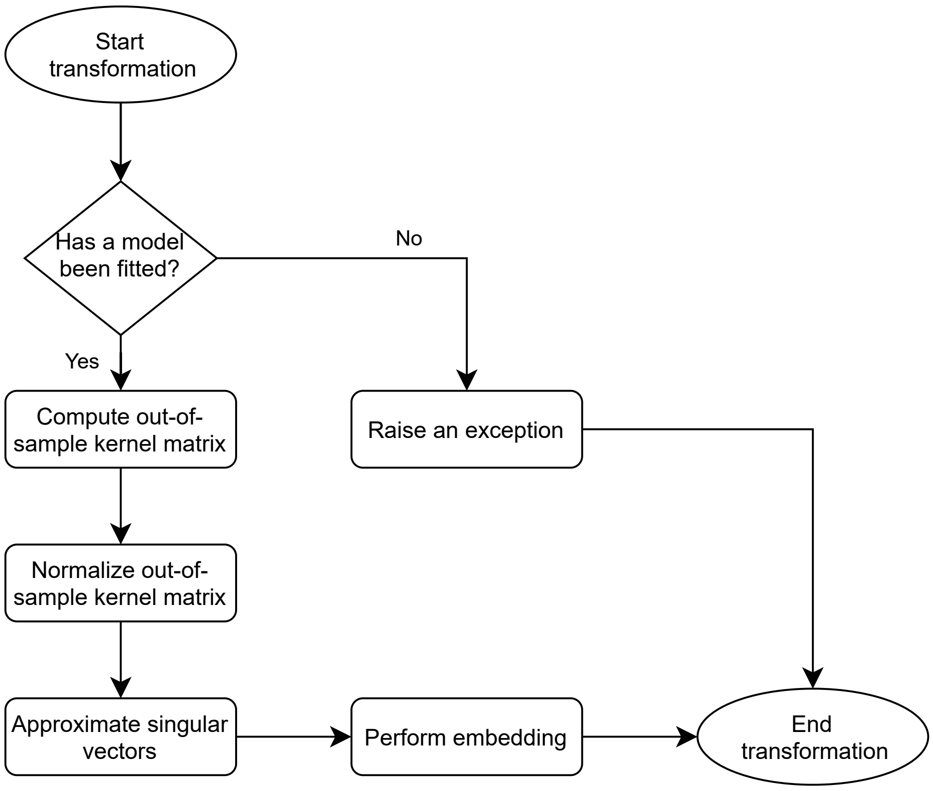


Figure 3.7: Roseland transformation process

A prerequisite for the execution of this function is that the data set has already been fitted. To assure this, we take advantage of the method `check_is_fitted` from the Scikit-learn package. If the model has not been fitted a `exceptions.NotFittedError` is raised, which is one of the requirements for the transformation method. In this case, the transformation ceases. If that is not the case, then the process continues. We compute the out-of-sample matrix by calculating the affinities between the points in the landmark set and the unseen set $\mathcal{Z}$. As stated in the formula in subsection 3.1.5, we need this matrix to be

of shape $k \times m$ with k being the number of samples in the to-be-transformed data set and m the number of landmarks. Accordingly, we call the method `compute_kernel_matrix` on the landmark set and pass the new set as a parameter. We then normalize the resulting matrix. This step is equivalent to the normalization in the fitting case. To get an approximation of the new left singular vectors we perform the Nyström extension, which is essentially calculating the result of the formula given in subsection 3.1.5. As described there, we acquire an approximation of the left singular vectors. We then change their coordinates by multiplying them by the diagonal matrix associated with the out-of-sample kernel matrix. The last step is to perform the embedding. We calculate the new coordinates in the lower dimension for each point in the unknown set by using the formula presented in subsection 3.1.4. For that, we simply plug in the approximated left singular vectors with the changed coordinates in the formula. By multiplying them by the singular values that have been raised to the power of `2*time_exponent`.
With this, the transformation ends and it results in an embedding in a lower dimension in the out-of-sample case. The selection of the dimension of this embedding differs slightly from the proposed approach in [17]. There, the first q' singular values and their corresponding vectors are selected. In our implementation, we have an attribute `target_coords_` associated with the Roseland instance that is an array of the indices of the singular vectors and values that should be returned. This attribute is however used only after fitting and before approximating the singular vectors with Nyström. Consequently, the singular values and vectors remain with the same size, determined by `n_svdpairs`. To be able to adjust the `target_coords_`, we provide a function named `set_target_coords` that takes in the array of the desired indices. If no indices are provided by using this function, then the first `n_svdpairs` with the largest singular values are used. By changing this property of Roseland, we not only make the implementation more similar to the one of Diffusion Maps in datafold, but we also make the embedding of Roseland more flexible. This way we can find a set of q' singular pairs that unfold the manifold into the dimension q' in a more meaningful way than by simply taking the q' pairs with the largest singular values. An option to infer this set is provided by the already discussed in section 2.3 `LocalRegressionSelection` class.
To embed the data set `X` into a lower dimension, we can first call `rose.fit(X)`, and then `rose.transform(X)`. Alternatively, we can directly call `rose.fit_transform(X)`. This method executes `fit`, after which it directly performs the embedding by using the inferred singular vectors and values.

3.4 Evaluation and Test

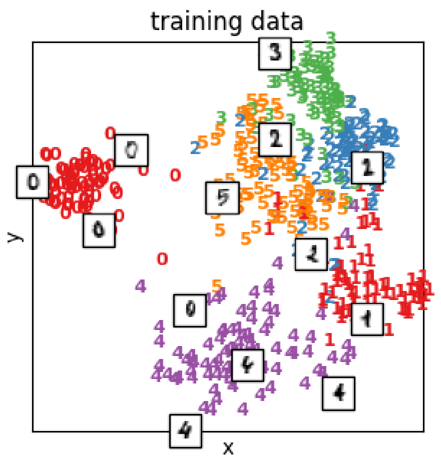
In this section, we present the results from the unit tests of our implementation. Additionally, we evaluate the scalability and robustness of the algorithm, and we compare the results to the ones we acquire when using Diffusion Maps.

3.4.1 Unit testing

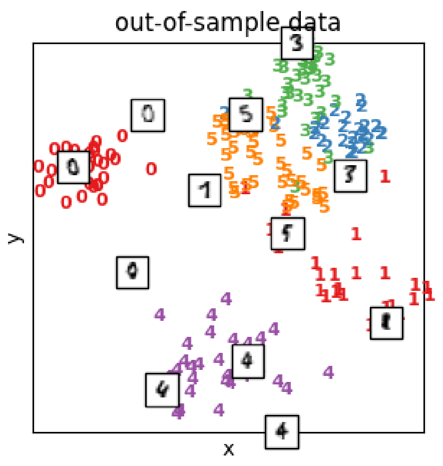
We use the Unit testing framework for Python to test the independent methods of Roseland, as well as how the separate parts of the algorithm work with each other. The tests and the acquired results after running them are described in Table 3.1. The following tests cover 94% of the Roseland module, which contains all the implementation code that has been contributed in the scope of this thesis. We aim to test Roseland for similar test cases to the ones Diffusion Maps has been tested for. In each test case, we use the same data and landmark sets to acquire meaningful comparisons. Analogously, the used kernel function, `epsilon` and `cut_off` values are also the same. The column "Pass?" represents if the given test case passes.

3.4.2 Out-of-sample embedding

In subsection 3.1.5, we discussed how the model fitted by Roseland can be used to determine the new coordinates of a different set without evaluating its SVD. In comparison to Diffusion Maps, we do not need the data set to perform the Nyström extension, but only the landmarks. As we can see in Figure 3.8, the regions of the computed coordinates for each digit match the regions from the original embedding. We can therefore conclude that



(a) In-sample embedding with $\gamma := 0.25$



(b) Out-of-sample embedding using the Nyström extension

Figure 3.8: In-sample and out-of-sample embeddings of the handwritten digits (0-5) by using the first and second singular vectors

the Nyström extension can successfully be used in the context of Roseland for the purpose of out-of-sample embedding.

Table 3.1: Unit tests

Test name	Description	Pass?
`test_dense_sparse`	The computed kernel matrix when no value for the `cut_off` is set (the matrix is dense), and the kernel matrix with a non-infinite `cut_off` should have values that are close (`rtol=1e-13`, `atol=1e-14`). The computed eigenvalues should be equal (compared using the method `assert_equal_eigenvectors` by datafold.	✓
`test_time_exponent`	The absolute result of the embedding after a very small time step (`time_exponent=1e-14`) should be almost identical (`rtol=1e-9`, `atol=1e-13`) to the case when there is no time step at all.	✓
`test_nystrom_out_of_sample_swiss_roll`	After setting the target coordinates, we should still be able to access all singular vectors, the number of which is specified by `n_svdpairs`. Furthermore, the test provides an argument `plot`. When it is set to `True`, the test case plots the first and fifth singular vectors of the swiss roll data set after the fitting, and the new coordinates of the said vectors after the transformation, as well as the absolute difference between the two plots. It is labelled ”out-of-sample” because the original set is treated as the unseen set, since the method `fit_transform` is not being used directly.	✓
`test_set_target_coords1`	The absolute values of the new coordinates of the selected `target_coords` should be close (`rtol=1e-10`, `atol=1e-14`) to the new coordinates of the said indices when no `target_coords` were set. This should hold no matter if `set_target_coords` was called before or after fitting the data. Additionally, the computed singular vectors of the examples with set target coordinates should be close (`rtol=1e-10`, `atol=1e-14`) to the ones where no indices for the embedding were provided. The final condition that must hold is that the attribute `n_features_out_` should be 2, so it conforms to the two dimensions we are outputting.	✓

test_set_target_coords2	When passing invalid values to set_target_coords as arguments, an exception should be raised. In the case of floating-point indices, a TypeError should be raised. Negative indices and indices bigger than the attribute n_svdpairs should result in a ValueError.	✓
test_dist_kwargs	The value of cut_off should be set correctly when passing it through dist_kwargs and it should result in the kernel matrix being sparse.	✓
test_speed	Prints the number of seconds the computation of the kernel matrix and svds takes, as well as their combined time, and the total runtime of the fitting.	✓

3.4.3 Accuracy

As we have briefly mentioned, Diffusion Maps always provides more accurate estimations of the eigenvectors and eigenvalues than Roseland. Shen and Wu provide a numerical comparison of the results from the fitting step of both algorithms in their work [21], illustrated (and slightly modified) in Figure 3.9. In Roseland, they use the same number of landmarks as data points. It is evident that the relative error of both the eigenvectors and eigenvalues of Roseland is (almost) always bigger. This makes Roseland less suitable than Diffusion Maps for applications where accuracy is vital.

3.4.4 Scalability

An area where we expect Roseland to outperform Diffusion Maps in the general case is scalability [20]. The first aspect related to scalability that we look into is the maximal number of data points we can process with Roseland. We test this by fitting the s-shaped curve data set on a Linux machine with a 6-core 3.7Ghz Ryzen 5 CPU and 16GB memory. The maximum amount of data points that can be processed depends as expected on the size of the landmark set. Another factor that contributes to the speed is the `cut_off` attribute, because when it has a non-infinite value the kernel matrix is sparse and, in turn, more efficient to process than a dense matrix. Since calling the function `optimize_parameters` on the landmark set is needed to get an approximation for the kernel scale, it makes using this function almost a requirement before creating the `Roseland` instance. Another advantage that results from calling `optimize_parameters` is that it always returns a value for the `cut_off`. These two arguments are the reasons why in almost all our test cases we set a value for `cut_off` and work with a sparse matrix. As evident in the unit tests above, the differences between the sparse and dense kernel matrix are negligible.

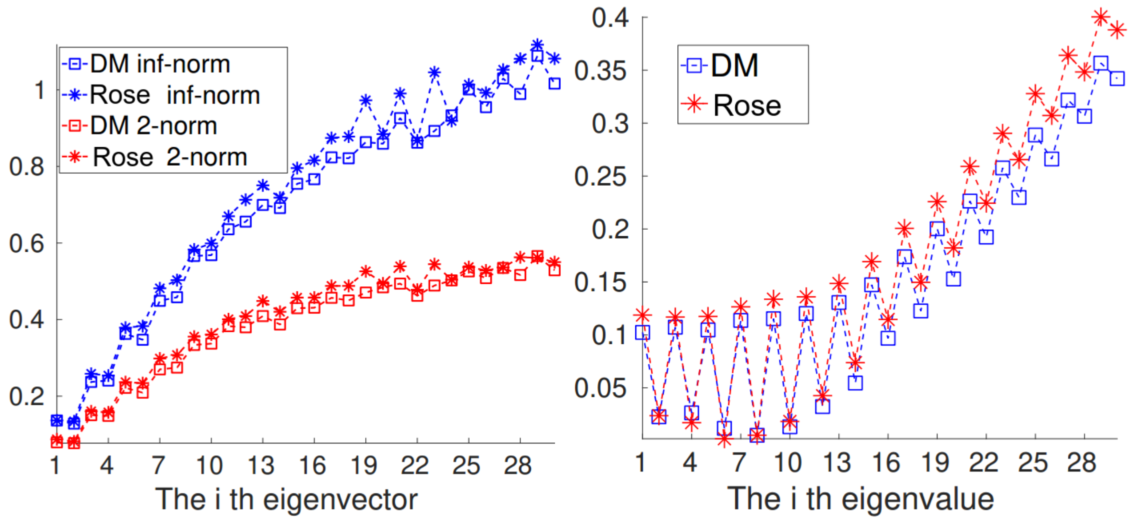
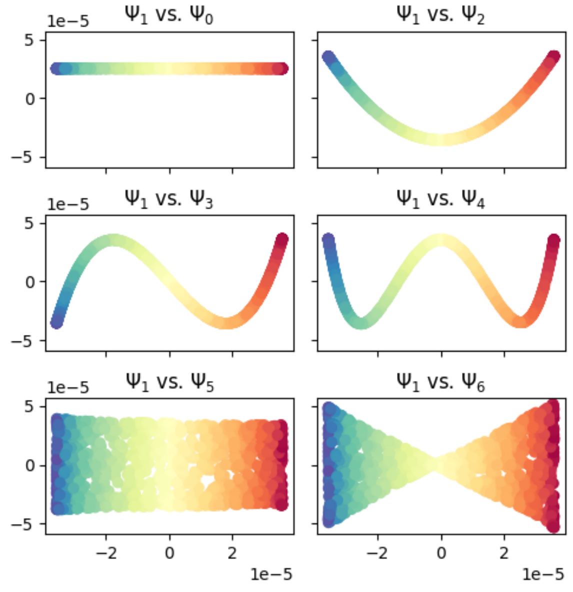
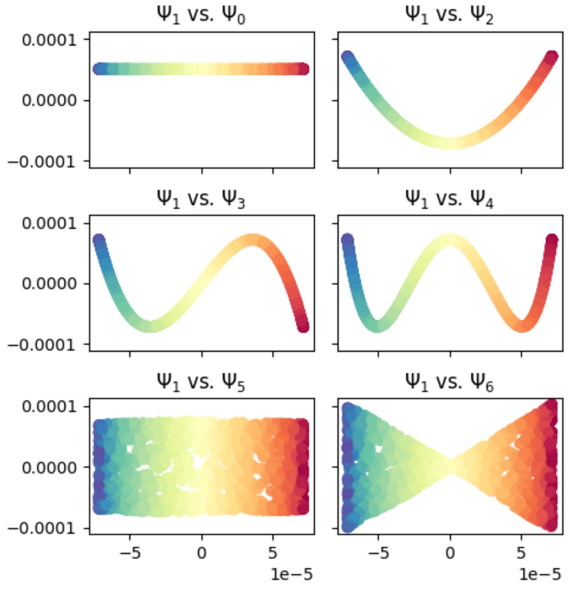


Figure 3.9: Comparison of the relative errors of the eigenvectors and eigenvalues after fitting 2500 uniformly sampled points with Diffusion Maps (DM) and with Roseland (Rose) with 2500 landmarks, modified from [21]

With these settings, we reach a limit for `gamma=0.1` of 4 000 000 data points with a runtime of 15575 seconds, which correspond to roughly 4,3 hours. Beyond that point, the process is either killed or not enough memory can be allocated for it to continue running. By increasing or decreasing the size of the landmark set, the data set number limit changes. For example, for `gamma=0.25` we can fit 2 500 000 data points in 14021 seconds or roughly 3,9 hours. The results of both runs are displayed in Figure 3.10. In comparison, on the same machine, we are able to run Diffusion Maps for up to 550 000 data points from the same manifold. The fitting in this run takes 3761 seconds, or around one hour.
All the following examples (except in Figure 3.12) are run on a Linux machine with a 4-core 1.8Ghz i7 CPU and 8GB memory. To evaluate experimentally if the actual complexity matches the calculated complexity in section 3.1.6 of $\mathcal{O}(nm^2)$, we first plot the fitting time against the number of data points from 10 000 to 100 000 with a time step of 10 000. We use the function `timeit` to measure the time and we perform the `fit`-function 10 times to lower the effects of the scheduler. It is sufficient to call only `fit` and not `fit_transform` because the fitting dominates the transformation in terms of computational complexity since it performs the SVD. The landmark set contains 1000 points and its size is constant for all data sizes. This way, we can assume that m is a constant in the complexity term. Therefore, we expect the plotted runtime of Roseland against the data set size with a fixed landmark size to resemble a linear function. It is easy to recognize that this is the case when comparing it with a first degree polynomial with respect to n, as shown in in Figure 3.11a. Analogously, we fix the size of the data set and only change the values for the land-



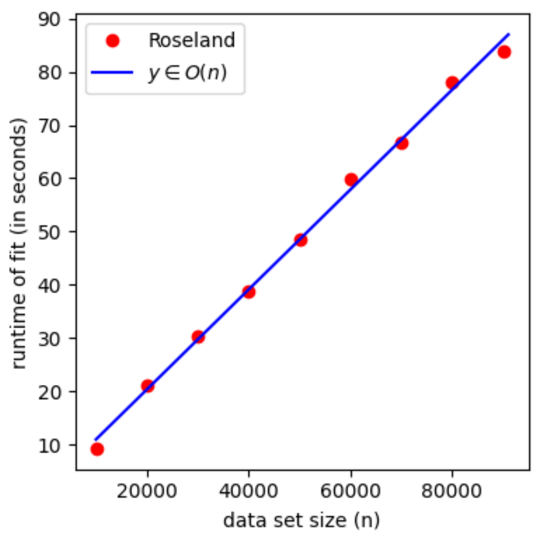
(a) 4 million data points, $\gamma := 0.1$, fitting time $= 15575s$



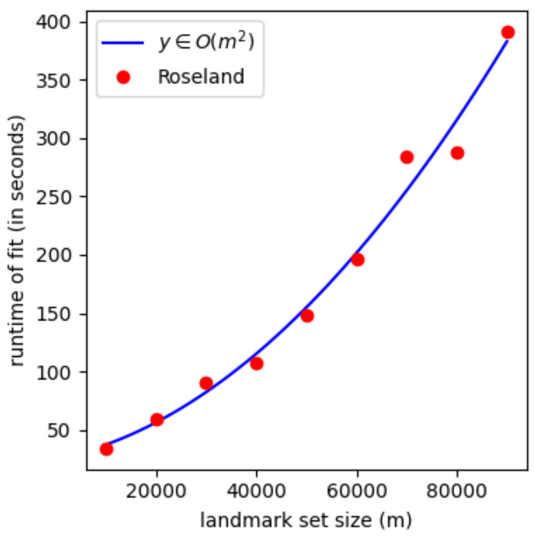
(b) 2.5 million data points, $\gamma := 0.25$, fitting time $= 14021s$

Figure 3.10: Potential embeddings of the s-shaped curve with different data and landmark set sizes

mark set. In that case, the plot is expected to resemble a quadratic function. When fixing the data set and using varying landmark set sizes we obtain the result illustrated in Figure 3.11b. To be able to compare the plotted runtimes with the expected quadratic function, we additionally plot the polynomial $c_0 m^2 + c_1 m + c_2$. To estimate the values for c_i we utilize the function `polyfit` from NumPy [12]. We then use `poly1d` from the same package to encapsulate the polynomial and then plot it. As we can see, the runtimes are close to the plotted function. In both cases, the runtime is improved by working with a sparse matrix. We additionally plot the runtime when both the data and the landmark set change with the same factor, meaning we use Roseland for different data sizes but with the same value for `gamma`. Because $m := 0.1n$, we expect the exhibited complexity to be cubic. Therefore, we plot a function $c_0 n^3 + c_1 n^2 + c_2 n + c_3$ by using `polyfit` and `poly1d` to visually assess if the plotted runtimes correspond to the overall complexity. We can see in Figure 3.12 that the plotted runtimes roughly correspond to the function. Considering all these arguments, we can conclude that the implementation exhibits the expected complexity.
Because in our implementation $m := \gamma n$, the complexity is implicitly in $\mathcal{O}(n^3)$, which corresponds to the complexity of Diffusion Maps. However, fixing the landmark set improves said complexity gradually to a linear one, as evident in 3.11a. When the landmark set size is not fixed, Roseland still has a better runtime than Diffusion Maps. The reason for this is



(a) Runtimes of fitting 10 times a data set with the size n by using 1000 landmarks



(b) Runtimes of fitting a data set with 100 000 points using a landmark with the size m

Figure 3.11: Evaluating the complexity by comparing the runtimes to a polynomial of first (left) and second (right) degree when fixing one variable and varying the values of the other one.

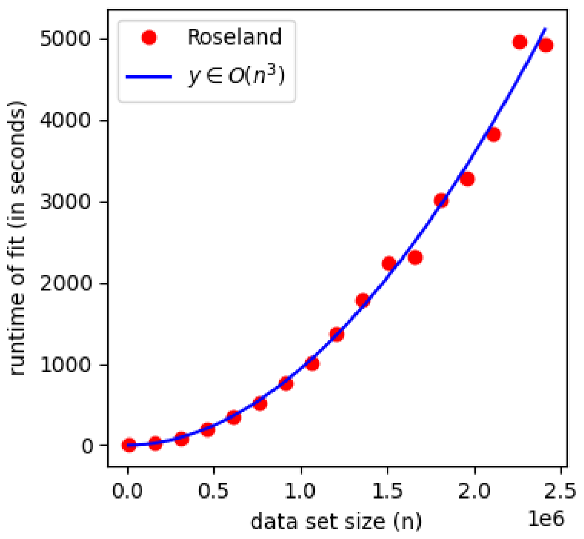
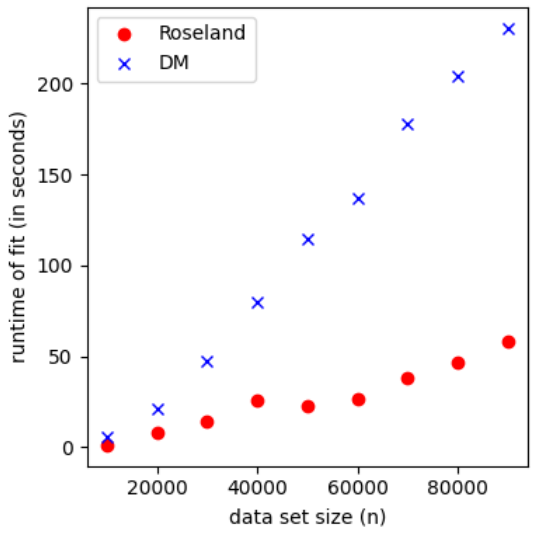
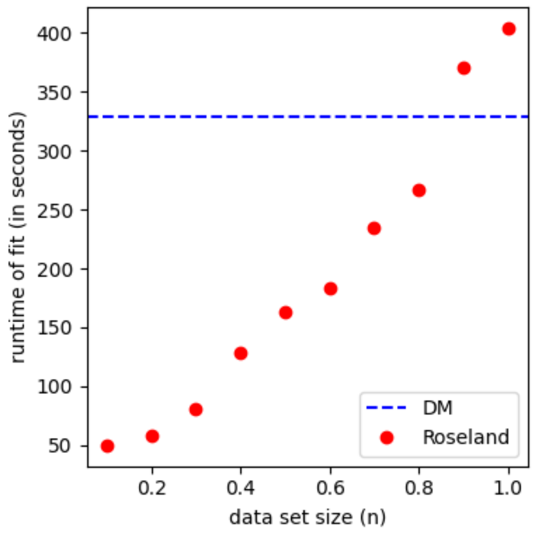


Figure 3.12: Evaluating the complexity by comparing the runtimes to a third-degree polynomial. The relative size of the landmark set is 0.1

the smaller kernel matrix. The runtime comparison between the two algorithms is shown in Figure 3.13. It is important to note that in both algorithms we use a sparse matrix, meaning they both have improved runtimes as opposed to the dense matrix cases. As expected, if the landmark set size increases too much, Roseland loses its scalability advantage, and in some cases results in a *worse* runtime than Diffusion Maps.



(a) Varying data set sizes, `gamma=0.25` (default setting)

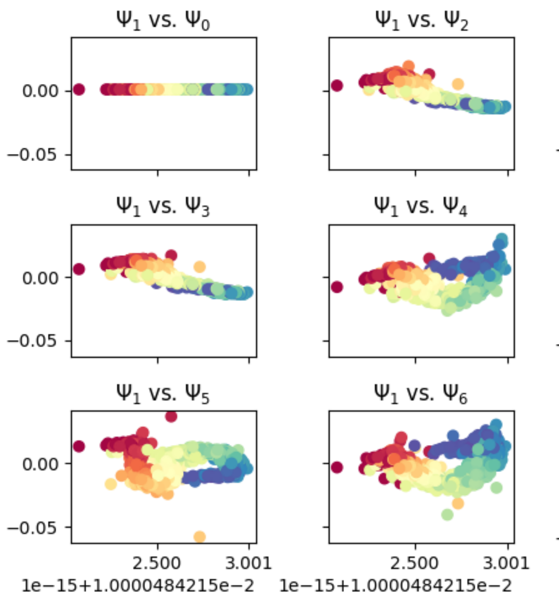


(b) 100 000 data samples and varying relative landmark set sizes

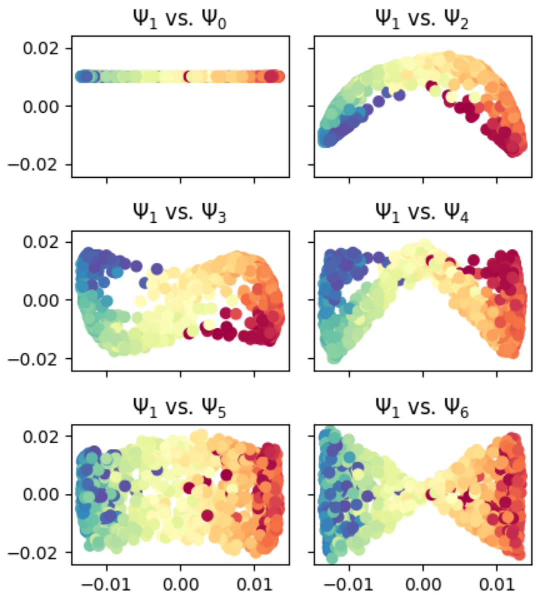
Figure 3.13: Runtime comparisons to Diffusion Maps

3.4.5 Robustness to Noise

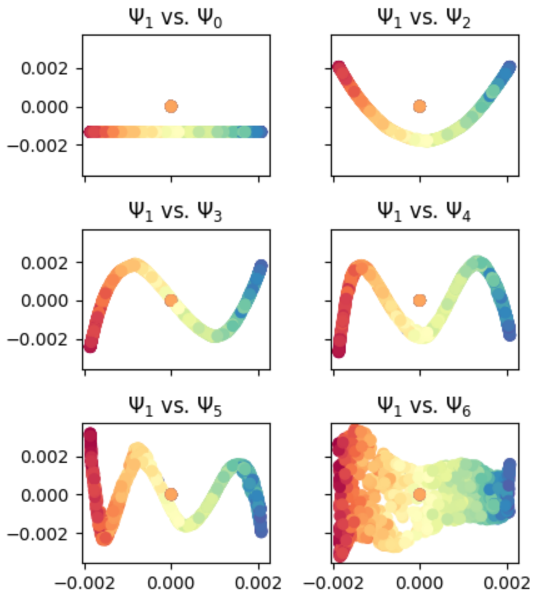
Another issue which spectral embedding approaches are facing is noisy data. According to [20], Roseland can tolerate rather low signal-to-noise ratios. In this subsection, we investigate the robustness of Roseland to noise by visually comparing the potential embeddings of the s-shaped curve to the results acquired by Diffusion Maps. For Diffusion Maps, we use the Gaussian kernel and the CkNN kernel [4]. For Roseland we consider two cases: noisy data set and noise-free landmark set, and noisy landmark set and noisy data set. We do not take the case of noise-free data and noisy landmarks into consideration since we aim to see the behaviour of both algorithms when the data set itself is noisy to be able to make a meaningful comparison. We use 10 000 data points from the data set, and the size of the landmark set has a value of 2 500. The noise we add is Gaussian with a standard deviation of 0.3. We specify the standard deviation when sampling the data set using the funcion `make_s_curve` from Scikit-learn. As evident in Figure 3.14, Roseland with a noise-free landmark set handles noise better than Diffusion Maps with a Gaussian kernel.



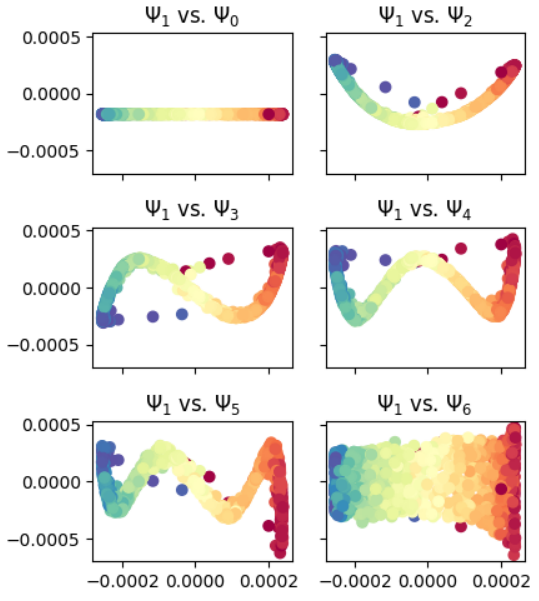
(a) Diffusion Maps with a Gaussian kernel



(b) Diffusion Maps with a Continous NN kernel wiith 10 neighbours, and $\delta = 6.8$



(c) Roseland with a noise-free landmark set with 2500 points



(d) Roseland with a landmark set (2500 points) with added Gaussian noise with the same σ

Figure 3.14: Robustness comparisons of Roseland and Diffusion Maps with additive Gausssian noise with a $\sigma = 0.3$. The potential embeddings of the s-curve with 10 000 data points are given.

Note that the approximation of the Laplace-Beltrami operator with the same number of data and landmark points for the s-shaped curve is not accurate even for noise-free sets, as pictured in section 3.2.2. As expected, the resulting potential embeddings are noisier when using a noisy landmark set, as pictured in Figure 3.14d. However, they are still more accurate than those acquired after running Diffusion Maps with a Gaussian kernel. It is a feasible assumption that in some application areas, collecting a small landmark set that is not noisy is relatively easy and not as computationally demanding as collecting a large data set that is not noisy [17]. When comparing the results of Roseland with a landmark set that is not noisy to the ones of Diffusion Maps with a CkNN kernel that is illustrated in Figure 3.14b, Roseland yields less noisy results. The results of Diffusion Maps, however, approximate the Laplace-Beltrami operator better. This is in part due to a well-chosen value for δ. When compared to Roseland with noisy landmarks, Diffusion Maps with a CkNN kernel looks similar in terms of how noisy the results are.
Another way of evaluating the robustness to noise could be to compare how close the eigenvalues are to the ground truth. By doing so, we can achieve a numerical comparison of the error terms for all four described approaches.

4 Conclusions

In this thesis, we implement the novel algorithm Roseland in the datafold [16] package. We evaluate the implementation by comparing the results of Roseland to these of Diffusion Maps in terms of scalability and robustness to noise.

4.1 Summary

In chapter 2 we discuss the state of the art by describing three approaches to dimensionality reduction: PCA (linear), Isomap (manifold learning), and Diffusion Maps (manifold learning). In addition, we present the Python package datafold and some of its functionalities. We then continue in chapter 3 by discussing the idea of the Roseland algorithm, our approach to choosing the landmarks and the size of the landmark set. Furthermore, we specify how we implement the algorithm in datafold by describing its architecture and the flow of the main functions. We finally evaluate our results by comparing them to the ones we acquire from the implementation of Diffusion Maps in the same package. For the comparisons, we consider the scalability and robustness to noise.

4.2 Discussion

The problem of high dimensionality emerges from the big and complex data sets, with which machine learning algorithms have to work. There are different approaches, both linear and non-linear that address the given problem. Here, we describe a manifold learning algorithm called Roseland that bears similarity to Diffusion Maps. Both algorithms aim at spectral embedding by a diffusion process. What sets Roseland apart from Diffusion Maps is the landmark set that is used when calculating the affinities between the data points. In effect, Roseland works with a landmark-set affinity matrix that contains the proximities between the data points and the landmarks. However, there is no equivalent to the normalization with α performed in Diffusion Maps that can remove the effects associated with the density of the data.
Our implementation in datafold provides means of inferring the singular vectors and values of the normalized kernel matrix (by fitting the data set), in-sample, and out-of-sample embeddings. The user can specify which singular vectors and values are used for the embedding. The out-of-sample extension is performed similarly to Diffusion Maps, with the Nyström extension. In contrast to Diffusion Maps, we do not need the entire data set to perform it, rather only the landmarks.

The landmark set is the central part of Roseland. In our implementation, the user can provide a landmark set or let a landmark set be subsampled from the data set by defining its relative size `gamma`. The bigger the landmark set (or `gamma`) is, the faster the convergence rate to the Laplace-Beltrami operator is. However, even when the landmark set has the same size as the data set (`gamma:=1.0`), Roseland has a worse variance than Diffusion Maps, making it less accurate [21]. This brings us to the conclusion that using a large landmark set is not beneficial. Since Roseland is rather aimed at scalability it is best to keep the landmark set small. This can positively affect the runtime performance. When the landmark set size is fixed even the computational complexity is improved. Roseland exhibits better runtimes than Diffusion Maps in the majority of cases. Additionally, it is able to process data sets with more points than Diffusion Maps. In our implementation, Roseland can handle point cloud data sets. Depending on the hardware, the number of points used in the landmark set, and if the `cut_off` attribute is set to some non-infinite value, a large number of points can be processed, e.g., 4 000 000 data points with 400 000 landmarks. On the same machine, Diffusion Maps runs for up to 550 000 data when using a sparse matrix. An exception to the better scalability is when the landmark set has a large number of points. If the relative size of the landmark set surpasses 0.7, the runtime of the algorithm worsens significantly. In some particularly unfavourable cases, it is higher than the runtime of Diffusion Maps.
Another area where Roseland performs better than classical Diffusion Maps is the robustness to noise. The landmarks implicitly act as the ”true” neighbours in the kNN scheme [17], and thus Roseland yields less noisy results. When visually compared to Diffusion Maps with CkNN kernel, the results of Roseland exhibit similar (or better in the case of not noisy landmark set) signal-to-noise ratios.

4.3 Outlook

The implementation needs further testing on different data sets, in particular, with real-world data. To be able to be fully integrated in the datafold package, the implementation of Roseland needs to provide support for time series. Additionally, to be able to be used in the same contexts as Diffusion Maps in all cases, the inverse transformation for Roseland has to be researched and implemented. The inverse transformation has the goal of mapping the coordinates from the (lower-dimensional) parameter space of the manifold to the (higher-dimensional) feature space of the original data set. A further improvement of the implementation is associated with a better design of the landmark set. To be able to remove the impact of a non-uniformly distributed data set, the landmark set should be sampled such that $p_Y(x) \propto \frac{1}{p_X^2(x)}$ [20], where p signifies the probability density function. There is currently ongoing research into how to best design the landmark set.

Bibliography

[1] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

[2] Mukund Balasubramanian, Eric L Schwartz, Joshua B Tenenbaum, Vin de Silva, and John C Langford. The isomap algorithm and topological stability. *Science*, 295(5552):7–7, 2002.

[3] Yoshua Bengio, Jean-françcois Paiement, Pascal Vincent, Olivier Delalleau, Nicolas Roux, and Marie Ouimet. Out-of-sample extensions for lle, isomap, mds, eigenmaps, and spectral clustering. *Advances in neural information processing systems*, 16:177–184, 2003.

[4] Tyrus Berry and Timothy Sauer. Consistent manifold representation for topological data analysis. *arXiv preprint arXiv:1606.02353*, 2016.

[5] Lawrence Cayton. Algorithms for manifold learning. *Univ. of California at San Diego Tech. Rep*, 12(1-17):1, 2005.

[6] Alan Kaylor Cline and Inderjit S Dhillon. Computation of the singular value decomposition. 2006.

[7] Ronald R Coifman and Stéphane Lafon. Diffusion maps. *Applied and computational harmonic analysis*, 21(1):5–30, 2006.

[8] Carmeline J Dsilva, Ronen Talmon, Ronald R Coifman, and Ioannis G Kevrekidis. Parsimonious representation of nonlinear dynamical systems through manifold learning: A chemotaxis case study. *Applied and Computational Harmonic Analysis*, 44(3):759–773, 2018.

[9] Ángela Fernández, Ana M González, Julia Díaz, and José R Dorronsoro. Diffusion maps for dimensionality reduction and visualization of meteorological data. *Neurocomputing*, 163:25–37, 2015.

[10] Charless Fowlkes, Serge Belongie, Fan Chung, and Jitendra Malik. Spectral grouping using the nystrom method. *IEEE transactions on pattern analysis and machine intelligence*, 26(2):214–225, 2004.

[11] Dimitrios Giannakis. Dynamics-adapted cone kernels. *SIAM Journal on Applied Dynamical Systems*, 14(2):556–608, 2015.

[12] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[13] Jingtian Hu and Andrew L Ferguson. Global graph matching using diffusion maps. *Intelligent Data Analysis*, 20(3):637–654, 2016.

[14] Ian T Jolliffe and Jorge Cadima. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150202, 2016.

[15] Rasmus Munk Larsen. Lanczos bidiagonalization with partial reorthogonalization. *DAIMI Report Series*, (537), 1998.

[16] Daniel Lehmberg, Felix Dietrich, Gerta Köster, and Hans-Joachim Bungartz. datafold: data-driven models for point clouds and time series on manifolds. *Journal of Open Source Software*, 5(51):2283, 2020.

[17] Yu-Ting Lin, Hau-Tieng Wu, and Chao Shen. Robust and scalable manifold learning via landmark diffusion for long-term medical signal processing. *bioRxiv*, 2020.

[18] Arik Nemtsov, Amir Averbuch, and Alon Schclar. Matrix compression using the nyström method. *Intelligent Data Analysis*, 20(5):997–1019, 2016.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[20] Chao Shen. *Robust and Scalable Unsupervised Learning via Landmark Diffusion, From Theory to Medical Application*. PhD thesis, Duke University, 2021.

[21] Chao Shen and Hau-Tieng Wu. Scalability and robustness of spectral embedding: landmark diffusion is all you need. *arXiv preprint arXiv:2001.00801*, 2020.

[22] Joshua B Tenenbaum, Vin De Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000.

[23] Laurens Van Der Maaten, Eric Postma, Jaap Van den Herik, et al. Dimensionality reduction: a comparative. *J Mach Learn Res*, 10(66-71):13, 2009.

[24] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[25] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[26] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.