Manifold Learning and Graph Kernels

Ca' Foscari University of Venice
Department of Environmental Sciences, Informatics and Statistics

Artificial Intelligence: Knowledge Representation and Planning [CM0472-1]
Academic Year 2021 - 2022

Students
Zuliani Riccardo 875532
Fortin Luca 858986

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This report introduce two main problem that affects the field of machine learning:

1. Compare performance obtained from the run of SVM on raw data and the one computed by **manifold algorithm**. The last cited method makes the assumption that move the problem space into an higher dimension (like it is normally done with SVM Kernel) will introduce some noise that negatively affect our model. Instead of this, it bring the problem space into a lower dimension.

2. Describe the data not using the classic **vector base model** but instead use **graph model**. Thus we have to take into account the topic **graph base kernel** in order to compute the similarity between data points.

As previous mention we have to compare the performance of SVM trained on respective kernel with and without the manifold learning step, on the following datasets:

- **PPI:** which stands for Protein-Protein Interactions (PPI) Introduced by Hamilton et al. in Inductive Representation Learning on Large Graphs[1]. Protein roles—in terms of their cellular functions from gene ontology—in various protein-protein interaction (PPI) graphs, with each graph corresponding to a different human tissue. positional gene sets are used, motif gene sets and immunological signatures as features and gene ontology sets as labels (121 in total), collected from the Molecular Signatures Database. The average graph contains 2373 nodes, with an average degree of 28.8 [1].

- **SHOCK:** which consists on a dataset of 2D shapes. If we visualize the graph represent a scheletrical representation of ad 2D figure. Here we find 150 graph subdivided in 10 class, so each contains 10 graphs example. Similar work had been done in [11].

In this report we have make the choice to analyse and discuss the result obtained using *Weisferler-Lehman* kernel with the application of two manifold learning algorithm *Isomap* and *Local Linear Embedding*.

[1] https://paperswithcode.com/paper/inductive-representation-learning-on-large

Chapter 2

Background

Up to now all the observation from a given classification problem that we have analysed were in **Feature Base Model**, the two dataset that we have introduced before open a new representation for us, the **Graph Base Model** representation. In this section we have a look on the classical feature representation and the new one previously cited.

Before to describe the two model we re-introduce the **Kernel Trick** since it is useful in both strategy.

2.1 Kernel Trick

Kernel Methods exploit kernel functions to work on high dimensional. This is provided by performing inner products between the images of all pairs of the data in that space, if we are considering problems regarding images. The previous mentioned operation is called **Kernel Trick**, it is a key point in the case of non linear separable problem, because using this we are able to solve these type of problem by generating an hyperplane that separate the data points in a higher-dimensional space.

Formally, a kernel maps two object x and z via a mapping ϕ into a feature space $\mathcal{H}$, measuring their *similarity* in $\mathcal{H}$ as computing:

$$K(x, z) = \phi(x)^T \phi(z) \tag{2.1}$$

The Kernel Trick is nothing but computing the inner product in $\mathcal{H}$ as kernel in the input space as saw in equation 2.1. But we have to be careful because the Kernel Trick has to satisfy two rules in order to consider the **kernel a valid one:**

- $K(x, z) = \phi(x)^T \phi(z) \quad \forall x, z \in S$
- **Positivity of a kernel:** A symmetric function $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is a **positive define kernel** if:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j K(x_i, x_j) \geq 0 \quad \forall n \in \mathbb{N} \quad \forall x_1, ..., x_n \in \mathcal{X} \quad \forall c_1, ..., c_n \in \mathbb{R}$$

2.2 Feature Base Model

The first method is the classic one, the Feature Base Model. This representation act to model a single observation of a problem into a vector that describe all its *Feature*. It is very usable since the vector can be projected into a particular space, where the grade of that space is the number of features. Moreover as we saw SVM uses 2.1 to move the current feature space into an other with higher dimension in order to try to separate all the observation into two main groups.

2.3 Graph Base Model

An other way to represent the entities of a given problem is to structure them in form of **graphs**:

Definition A graph G consists of an ordered set of n vertices $V = v_1, v_2, ..., v_n$ and a set of directed edges $E \subset V \times V$. When G is unweighted, it means there are no weights on the edges, so the graph could be represented by a $n \times n$ matrix A, where

$$A_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

For weighted graphs the matrix is composed by $A_{i,j} = w_{i,j}$.

This type of representation is widely used since it can describe in a natural way a large number of complex relations and structures. Moreover graph base model has the advantage of improving the expressiveness and versatility of complex models. But we have the drawback that since the common machine learning algorithm works with the feature base model we have to obtain a conversion that it is not perfectly defined. To overcome this problem we adopt the already mentioned **Kernel Trick**, since as the definition stands say, it is a function of two object that measures their similarity.

2.3.1 Graph Comparison Problem

Thanks to the kernel trick it is not necessary to perform feature extraction from graph to transform them into feature vectors. The similarity measure can be expressed as the following definition.

Definition Given two graphs G and G' from the space of graphs $\mathcal{G}$ the problem of graph comparison is to find a mapping

$$s : \mathcal{G} \times \mathcal{G} \to \mathbb{R}$$

Such that $s(G, G')$ quantifies the *similarity (of dissimilarity)* of G and G'.

2.3.2 Graph Isomorphism

Common algorithm for deciding the similarity between two graphs are based on the concept of graph isomorphism.

Definition Given two graph G_1 and G_2 find a mapping f of the vertices of G_1 to the vertices of G_2 s.t. G_1 and G_2 are identical. Then f is an **isomorphism**, and G_1 and G_2 are said to be **isomorphic**.

Nowadays we do not know a polynomial time algorithm for graph isomorphism, but we also do not know whether the problem is NP-complete. On the other hand, we know that **subgraph isomorphism** is NP-complete. Subgraph isomorphism checks whether there is a subset of edges and vertices of G_1 that is isomorphic to a smaller graph G_2.

2.3.3 Positive Symmetric Similarity Measure

There is a similar problem respect to graph isomorphism, the binary similarity function. This gives no information of the amount of similarity or dissimilarity of the two graph since as the section noun suggest it us the binary language. Thus we want to generate a measure that gives us a more useful measure of the graphs similarity.

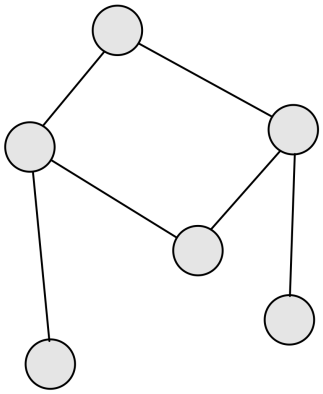


Figure 2.1: Graph G_1

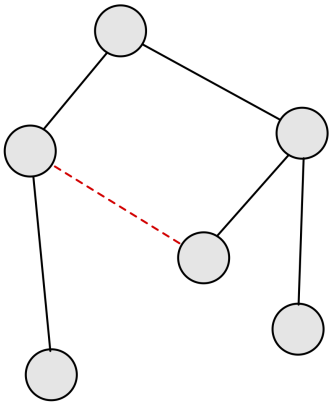


Figure 2.2: Graph G_2

Taken these two graph as example, all of us agree with the fact that they are not the same, because G_2 has not an edge that G_1 has. On the other hand we also know that except the edge taken into account previously these two are equal. If we consider *Isomorphism* as similarity measure we obtain 0 as result because they are no the same graph. However we would like to have a new positive and symmetric similarity measure that describe the similarity of two graphs via continuous value and not binary value, so indicating how much the two are similar, in the following form:

$$K(G_1, G_2) \geq 0$$

Chapter 3

Graph Kernel

A graph kernel is a kernel function that computes an inner product on graphs. Graph kernels can be intuitively understood as functions measuring the similarity of pairs of graphs. They allow kernelized learning algorithms such as support vector machines to work directly on graphs (as we would do soon), without having to do feature extraction to transform them to fixed-length, real-valued feature vectors [14]. Moreover graph kernel based their measure on common substructures which can be computed in polynomial-time. An important assumption for kernel definition it has to be made, as in the SVM's case also here a kernel it must be in the form of *positive definite kernel*.

3.1 Definition and Problems

In this section we analyse a deepening definition of Graph Kernel, with a focus on the problems.

3.1.1 R-convolution Kernels

To more explain graph kernel is better to introduce R-convolution kernels a family of graph kernel which is instance of.

- Let X be a set of composite objects (e.g., cars), and $\bar{X_1}, ..., \bar{X_D}$ be sets of parts (e.g., wheels, brakes, etc.). All sets are assumed countable.
- Let R denote the relation "being part of":

$$R(\bar{x_1}, ..., \bar{x_D}, x) = 1, \quad \iff \quad \bar{x_1}, ..., \bar{x_D} \text{ are parts of } x$$

- The inverse relation R^{-1} is defined as:

$$R^{-1}(x) = \bar{x} : R(\bar{x}, x) = 1$$

  In other words, for each object x, $R^{-1}(x)$ is a set of component subsets, that are part of x.

- We say that R is finite, if R^{-1} is finite for all $x \in X$. [7]

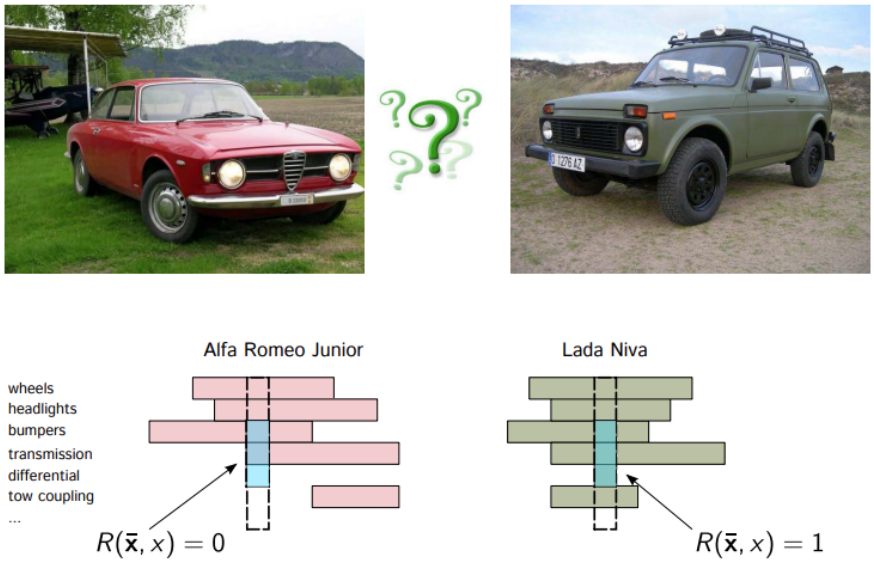


Figure 3.1: A Naive Example

- Let x be a D-tuple in $X = X_1 \times ... \times X_D$. Let each of the D components of $x \in X$ be a part of x. Then $R(\bar{x}, x) = 1 \iff \bar{x} = x$.
- Let $X_1 = X_2 = X$ be sets of all finite strings over a finite alphabet. Define $R(\bar{x_1}, \bar{x_2}, x) = 1 \iff x = x_1 \circ x_2$, i.e. concatenation of x_1 and x_2.
- Let $X_1 = ... = X_D = X$ be a set of D-degree ordered and rooted trees. Define $R(\bar{x}, x) = 1 \iff \bar{x_1}, ..., \bar{x_D}$ are D subtrees of the root of $x \in X$

Definiton. Let $x, y \in X$ and $\bar{x}$ and $\bar{y}$ be the corresponding sets of parts. Let $K_d(\bar{x_d}, \bar{y_d})$ be a kernel between the d-th parts of x and y $(1 \leq d \leq D)$. Then the **convolution kernel** between x and y is defined as:

$$K(x, y) = \sum_{\bar{x} \in R^{-1}(x)} \sum_{\bar{y} \in R^{-1}(y)} \prod_{d=1}^{D} K_d(x_d, y_d)$$

Subset Product Kernel - Theorem. Let K be a kernel on a set $U \times U$. The for all finite, non-empty subsets $A, B \subseteq U$

$$K'(A, B) = \sum_{x \in A} \sum_{y \in B} K(x, y)$$

is a *valid kernel*. [7]

In other word these kernel compare decompositions of a specific instance or object, most of them do so by count the number of isomorphic substructures. Further we can say that Graph Kernel are nothing but convolution kernel on pairs of graphs.

Moreover, once we have defined a *positive semi-definite kernel* $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ on a set X, there exists a map in the form of $\phi : X \to \mathcal{H}$ into a *Hilbert* space $\mathcal{H}$ such that $k(x, y) = \phi(x)^T \phi(y) \quad \forall x, y \in X$. The distance between $\phi(x)$ and $\phi(y)$ can be computed as:

$$||\phi(x), \phi(y)||^2 = \phi(x)^T \phi(x) + \phi(y)^T \phi(y) - 2\phi(x)^T \phi(y)$$

3.1.2 Hardness Result

The main problem in the previous method is that given the degree of data expressed by the graph, defining complete kernels, has been proved to be as hard as solve the graph isomorphism problem. [4]

Proposition. Let $k(G, G') = \langle \phi(G), \phi(G') \rangle$ be a graph kernel. Let ϕ be injective, then computing any complete graph kernel is at least as hard as deciding whether two graphs are isomorphic. Thus, since ϕ is injective, we have:

$$\sqrt{k(G, G) - 2k(G, G') + k(G', G')}$$
$$= \sqrt{\langle \phi(G) - \phi(G'), \phi(G) - \phi(G') \rangle}$$
$$= ||\phi(G) - \phi(G')|| = 0 \iff G \text{ is } \textbf{isomorphic} \text{ to } G'$$

3.1.3 Horseshoe Effect

Instead of computing kernel for injective mappings, we would like to have a polynomial-time, non-injective kernel that is able to explain correctly what is happened.

Furthermore, we have to mention that kernel are very effective in generating implicit embedding, but there is the problem that we are not completely sure that the new projected data into the Hilbert space will give a better result respect the previous space. This is a common problem. Bellow we gave an example in which the data tends to be clustered along a curve that wraps around the embedding space [2]

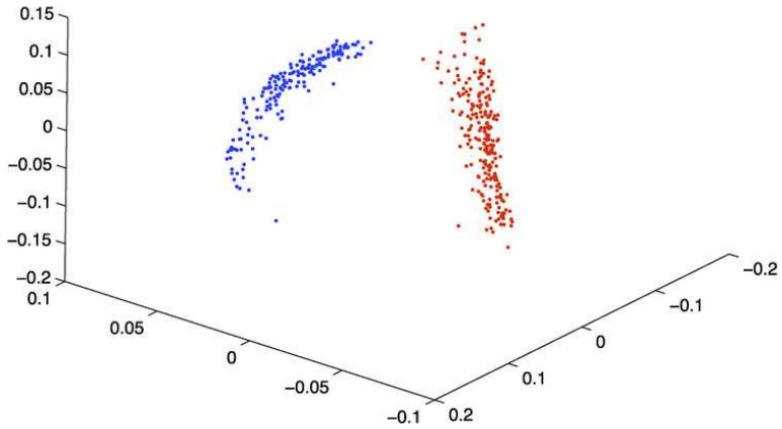


Figure 3.2: Example of Horseshoe Hffect. 3-Dimensional MDS output of legislators based on the 2005 U.S. House roll call votes

This *Horseshoe* is the intersection between the manifold and the plane. This effect could be caused by the normalization, that project all the data point from the Hilbert space into the unit sphere possibly creating an unwanted curvature of the data points. All of this is called the *Horseshoe effect*.

Usually the non-linearity of the mapping is used to improve the local class separability. In the following chapter we use some manifold technique in order to embed the graph into a lower dimensional vectorial space. But now we move our focus in describe the only used Graph Kernel, the Weisfeiler - Lehman Kernel.

3.2 Weisfeiler - Lehman Kernel

We would like to note that all of this section is take from the following scientific paper [6]. The Weisfeiler - Lehman (WL) kernel uses concepts from the Weisfeiler-Lehman test of isomorphism (Weisfeiler and Lehman, 1968), more specifically its 1-dimensional variant, also known as *"naive vertex refinement"*. Assume we are given two graphs G and G' and we would like to test whether they are isomorphic. This algorithm test proceeds in iterations, which are indicated by i as we can see in Algorithm 1.

The key idea of the algorithm is to augment the node labels by the sorted set of node labels of neighbouring nodes, and compress these augmented labels into new, short labels. These steps are then repeated until the node label sets of G and G' differ, or the number of iterations reaches n.

The Weisfeiler-Lehman algorithm terminates after step 4 of iteration i if:

$$\{l_i(v)|v \in V\} \neq \{l_i(v')|v' \in V'\}$$

That is, if the sets of newly created labels are not identical in G and G', the graphs are then not isomorphic. Otherwise if the sets are identical after n iterations, it means that either G and G' are isomorphic, or the algorithm has not been able to determine that they are not isomorphic. In this algorithm the graph is defines like so $G = (V, E, l)$, where l is a function returning the label of a node.

In each iteration i of the Weisfeiler-Lehman algorithm 1, we get a new labeling $l_i(v)$ for all nodes v. Recall that this labeling is concordant in G and G', meaning that if nodes in G and G' have identical multiset labels, and only in this case, they will get identical new labels.

Algorithm 1 One iteration of the 1-dim. Weisfeiler-Lehman test of graph isomorphism

1. Multiset-label determination
   - For $i = 0$, set $M_i(v) := l_0(v) = l(v)$
   - For $i > 0$, assign a multiset-label $M_i(v)$ to each node v in G and G' which consists of the multiset $\{l_{i-1}(u) | u \in \mathcal{N}(v)\}$
2. Sorting each multiset
   - Sort elements in $M_i(v)$ in ascending order and concatenate them into a string $s_i(v)$
   - Add $l_{i-1}(v)$ as a prefix to $s_i(v)$ and call the resulting string $s_i(v)$
3. Label compression
   - Sort all of the string $s_i(v)$ for all v from G and G' in ascending order
   - Map each string $s_i(v)$ to a new compressed label, using a function

$$f : \Sigma^* \to \Sigma \quad \text{s.t} \quad f(s_i(v)) = f(s_i(w)) \iff s_i(v) = s_i(w)$$

4. Relabeling
   - Set $l_i(v) := f(s_i(v))$ for all nodes in G and G'

Furthermore since in our datasets (PPI and SHOCK) the nodes are not associated to a specific label, we decide to compute a function in order to compute the degree of each nodes.

3.2.1 Weisfeiler - Lehman Subtree Kernel

The complete version of WL kernel is called **Weisfeiler - Lehman Subtree Kernel** in which we process all N graphs simultaneously and conduct the steps given in Algorithm 2 on each graph G in each of h iterations.

Definition Let G and G' be two graphs. Define $\Sigma_i \subset \Sigma$ as set of letters that occur as node labels at least once in G or G' at the end opf the $i-th$ iteration of the Weisfeiler - Lehman algorithm. Let Σ_0 be the set of original node labels of G and G'. Assume all Σ_i are pairwise disjoint. Without loss of generality, assume that every $\Sigma_i = \{\sigma_{i1}, ..., \sigma_{i|\Sigma_i|}\}$ is ordered, Define a map $c_i : \{G, G'\} \times \Sigma_i \to \mathbb{N}$ s.t. $c_i(G, \sigma_{i,j})$ is the number of occurrences of the latter σ_{ij} in the graph G. The Weisfeiler-Lehman subtree kernel on two graphs G and G' with h iterations is defined as:

$$k^{(h)}_{WLsubtree}(G, G') = \langle \phi^{(h)}_{WLsubtree}(G), \phi^{(h)}_{WLsubtree}(G') \rangle$$

where

$$\phi^{(h)}_{WLsubtree}(G) = (c_0(G, \sigma_{01}), ..., c_0(G, \sigma_{0|\Sigma_0|}), ..., c_h(G, \sigma_{h1}), ..., c_h(G, \sigma_{h|\Sigma_h|}))$$

and

$$\phi^{(h)}_{WLsubtree}(G') = (c_0(G', \sigma_{01}), ..., c_0(G', \sigma_{0|\Sigma_0|}), ..., c_h(G', \sigma_{h1}), ..., c_h(G', \sigma_{h|\Sigma_h|}))$$

In summary the Weisfeiler-Lehman subtree kernel counts the common *original* and *compressed* label in both graphs. **The greater this number is, the more similar are the two graphs.**

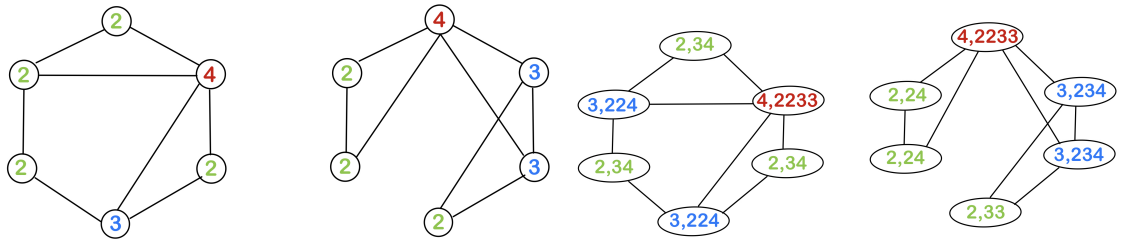
Algorithm 2 One iteration of the Weisfeiler-Lehman subtree kernel computation on N graphs

1. Multiset-label determination
   - Asign a multiset-label $M_i(v)$ to each node v in G which consists of the multiset $\{l_{i-1}(u) | u \in \mathcal{N}(v)\}$. Where $\mathcal{N}(v)$ is the *neighbors* of v
2. Sorting each multiset
   - Sort elements in $M_i(v)$ in ascending order and concatenate them into a string $s_i(v)$
   - Add $l_{i-1}(v)$ as a prefix to $s_i(v)$
3. Label compression
   - Map each string $s_i(v)$ to a new compressed label, using a function

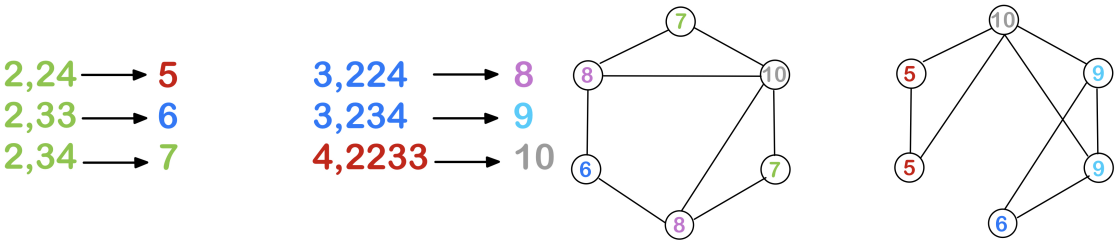
$$f : \Sigma^* \to \Sigma \quad \text{s.t} \quad f(s_i(v)) = f(s_i(w)) \iff s_i(v) = s_i(w)$$

4. Relabeling
   - Set $l_i(v) := f(s_i(v))$ for all nodes in G

Now we graphically see an application of WL subtree kernel algorithm.



(a) Graphs Label of G and G' (b) Mutiset label determination and sorting



(c) Label compression (d) Relabling

Figure 3.3: WL subtree kernel algorithm steps [12]

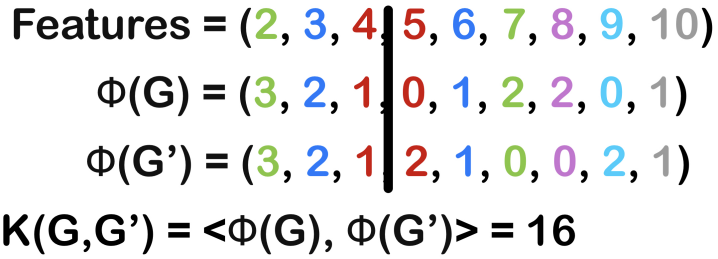


Figure 3.4: End of the first iteration, describing the feature vector representations of G and G' [12]

We have to note that in Figure 3.4 on the right part of the black bar that divide the two feature vectors, there are the counts of *compressed node labels* and, on the left hand side of the bar there are the counts of *original node labels.*

Finally we analyse the complexity for N graphs. The Weisfeiler-Lehman subtree kernel with h iterations on all pairs of these graphs can be computed in $O(Nhm + N^2hn)$. This effort comes from the fact that computing $\sigma^{(h)}_{WLsubtree}$ on all N graphs in h iterations is $O(Nhm)$, assuming that $m > n$. And to get all pairwise kernel values, we have to multiply all feature vectors, which requires a runtime of $O(N^2hn)$, as each graph G has at most hn non-zero entries in $\sigma^{(h)}_{WLsubtree}(G)$.

Chapter 4

Manifold Learning

Machine learning is being used extensively in fields like computer vision, natural language processing, and data mining. In many modern applications that are being built, we usually derive a classifier or a model from an extremely large data set. The accuracy of the training algorithms is directly proportional to the amount of data we have. So most modern data sets often consist of a large number of examples, each of which is made up of many features. Having access to a lot of examples is very useful in extracting a good model from the data, but managing a large number of features is usually a burden to our algorithm. The thing is that some of these features may be irrelevant, so it's important to make sure the final model doesn't get affected by this. If the feature sets are complex, then our algorithm will be slowed down and it will be very difficult to find the global optimum. Given this situation, a good way to approach it would be to reduce the number of features we have. But if we do that in a careless manner, we might end up losing information. We want to reduce the number of features while retaining the maximum amount of information. [3]

This let us to introduce the *course of dimensionality* which refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces that do not occur in low-dimensional settings. The common theme of this subject is that when the dimensionality increases, the volume of the space increases so fast that the available data become sparse. In high dimensional data, however, all objects appear to be sparse and dissimilar in many ways. [13]

There are many approaches to dimensionality reduction based on a variety of assumptions. But the most popular algorithm for this task is Principal Component Analysis (PCA), PCA finds the directions along which the data has maximum variance in addition to the relative importance of these directions. PCA is most useful in the case when data lies on or close to a linear sub-space of the data set. We could resume PCA in a *linear dimensionality reduction technique*. [3]

There is a variant of this technique, a *non-linear* one, in which **manifold learning** belongs to. This method can be viewed as the non-linear version of PCA. In PCA we project the data onto some low-dimensional surface, the problem is that it is restrictive in the sense that those surfaces are all linear. What if the the best representation lies in some weirdly shaped surface? PCA would not work very well in this situation because it will look for a planar surface to describe this data. But the point is that the planar surface does not exist. Manifold learning solves this problem very efficiently. [3]

There are many implementations of manifold learning such as:

- Isomap
- Diffusion Map
- Laplacian Eignemaps
- Local Linear Embedding

In this report we look only at **Isomap** and **Local Linear Embedding** methods to the distance matrix computed starting from the kernel matrix obtained from the previous graph kernel over a set of n graphs. Moreover we apply a classic linear SVM in order to learn data and class separation.

4.1 Isomap

Isomap, which states for *Isometric Feature Mapping* is a non-linear dimensionality reduction method. It is a different version of metric Multidimensional Scaling (MDS), and reduces the dimensionality while preserving geodesic distance. The most obvious difference between MDS and Isomap: **Euclidean distance is preserved in MDS, while geodesic distance is preserved in ISOMAP.** [5]

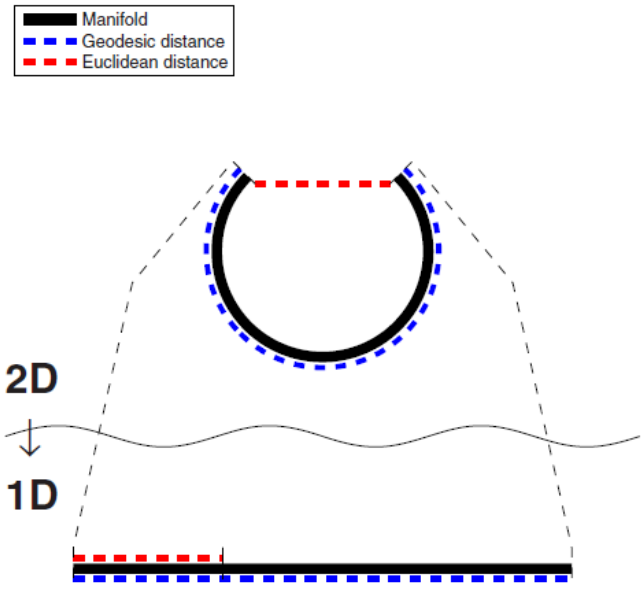


Figure 4.1: Euclidean Distance vs Geodesic Distance

While the Euclidean distance calculates only the distance by ignoring the shape of the dataset, the **geodesic distance** is calculated by passing the shortest path on the dataset. In this case, we can roughly say that the difference between the geodesic distance and the Euclidean distance; while the geodesic distance considers the data adjacent to these data, in the Euclidean distance it is only calculated the shortest linear path. Of course, since our goal is to reduce the dimensionality of the dataset with the least loss, more effective results can be obtained with geodesic distance according to the dataset. [5]

Moreover we can resume what we have sayed by:

- If node i and j are **close** the effect of *curvature* is minimal and the **Euclidean Distance** is a good estimator for geodesic distance
- Otherwise, if node i and j are **far away from each other** we can have a heavy curvature along the manifold, thus the geodesic distance is estimated as the **length of the minimal path** between i and j on a neighborhood graph.

So mathematically speaking given a data sample X of dimension n objects and a distance function between node i and j identified by $d_X(i,j)$, we compute the **dissimilarity matrix** Δ, in which each cell indicate the distance from the couple of nodes. The **goal** of MDS is to find the n vectors $x_1, ..., x_n \in \mathcal{R}^N$ from Δ, s.t.:

$$||x_i - x_j|| = d_X(x_i, x_j) \quad \forall i, j \in 1, ..., n$$

Where the vectors represent the embedding from the objects into a new space in which distances are preserved.

Isomap algorithm can be summarized in the following steps [5]:

- Use a KNN approach to find the k nearest neighbors of every data point.
- Once the neighbors are found, construct the neighborhood graph where points are connected to each other if they are each other's neighbors. Data points that are not neighbors remain unconnected.
- Compute the shortest path between each pair of data points (nodes). Typically, it is either Floyd-Warshall or Dijkstra's algorithm that is used for this task. Note, this step is also commonly described as finding a geodesic distance between points.
- Use multidimensional scaling (MDS) to compute lower-dimensional embedding. Given distances between each pair of points are known, MDS places each object into the N-dimensional space (N is specified as a hyperparameter) such that the between-point distances are preserved as well as possible.

4.2 LLE

Locally Linear Embedding or LLE is an unsupervised learning algorithm that computes low-dimensional, neighborhood-preserving embeddings of high-dimensional inputs. Unlike clustering methods for local dimensionality reduction, LLE maps its inputs into a single global coordinate system of lower dimensionality, and its optimizations do not involve local minima. [8]

To begin, suppose the data consist of N real-valued vectors $\vec{X}_i$ (or inputs), each of dimensionality D, sampled from an underlying manifold. Provided there is sufficient data (such that the manifold is well-sampled), we expect each data point and its neighbors to lie on or close to a locally linear patch of the manifold. [10]

In the simplest formulation of LLE, one identifies K nearest neighbors per data point, as measured by Euclidean distance. Moreover the loss function state as follow:

$$E(W) = \sum_i \left| \vec{X}_i - \sum_j W_{ij} \vec{X}_j \right|^2 \tag{4.1}$$

Where the weight W_{ij} indicate the contribution on the $j-$th data point to the $i-$th reconstruction. [10]

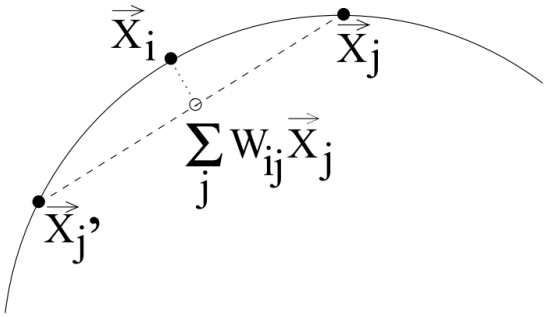


Figure 4.2: A data point $\vec{X}_i$, its neighbours $\vec{X}_j$ and its locally linear reconstruction $\sum_j W_{ij}\vec{X}_j$

To compute the weights, we minimize the cost function in Equation 4.1 subject to two constraints: [10]

1. **Sparseness:** each data point $\vec{X}_i$ is reconstructed only from its neighbors, enforcing $W_{ij} = 0$ if $\vec{X}_j$ does not belong to this set.

2. **Invariance:** the rows of the weight matrix sum to one: $\sum_j W_{ij} = 1$.

The goal of LLE is to minimize the $E(W)$ and what we want to obtain is the local geometry in the input space to be equally valid for local patches on the manifold. In particular, the same weights W_{ij} that reconstruct the input $\vec{X}_i$ in D dimensions should also reconstruct its embedded manifold coordinates in d dimensions, with $d \ll D$. [10]

And finally the main steps of the LLE algorithm are the following one [10]:

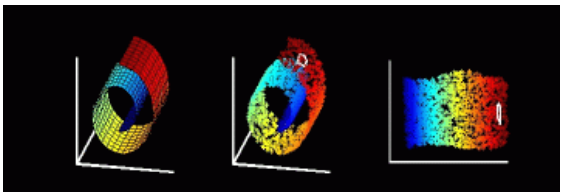
- Compute the neighbors of point, $\vec{X}_i$.

- Compute the weights W_{ij} that best reconstruct each data point $\vec{X}_i$ from its neighbors, minimizing the cost in Equation 4.1 by constrained linear fits.

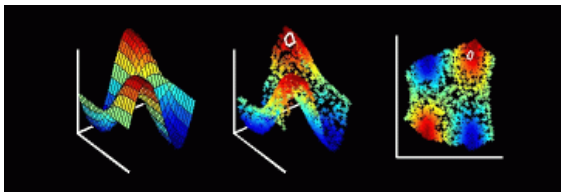
- Each high dimensional input $\vec{X}_i$ is mapped to a low dimensional output $\vec{Y}_i$ representing global internal coordinates on the manifold. This is done by choosing the d dimensional coordinates of each output $\vec{Y}_i$ to minimize the embedding cost function:

$$\phi(Y) = \sum_i \left| \vec{Y}_i - \sum_j W_{ij}\vec{Y}_j \right|^2 \tag{4.2}$$

This last cost function is based on locally linear reconstruction errors, but here we fix the weights W_{ij} while optimizing the outputs $\vec{Y}_i$.



(a) Swiss Role Example [9]



(b) Twin Peaks Example [9]

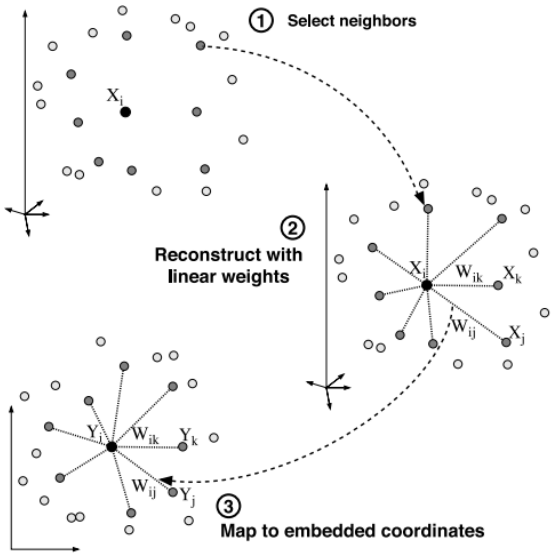
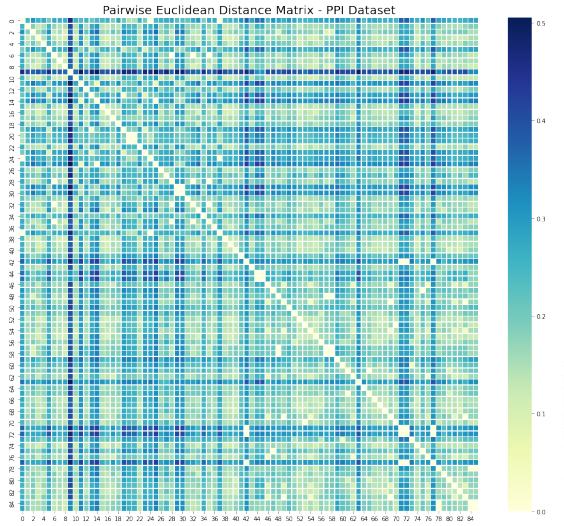


Figure 4.4: LLE Algorithm's steps [10]

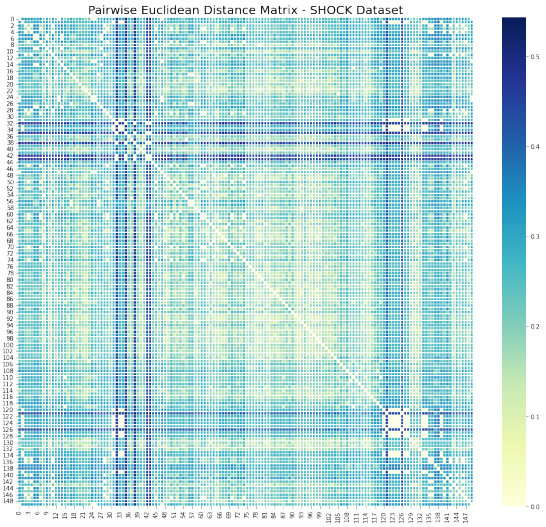
Chapter 5

Result & Comparison

The reported results are obtained from a 10-fold Cross Validation with shuffled dataset. The Weisfeiler-Lehman kernel is run with $h = 5$.



(a) PW distances for the PPI dataset



(b) PW distances for the SHOCK dataset

5.1 SVM without ML step

Measure	Value
Minimum	0.625
Mean	0.706
Maximum	1
Variance	0.013
Standard Deviation	0.113

Table 5.1: PPI - CV Results

Measure	Value
Minimum	0.267
Mean	0.313
Maximum	0.467
Variance	0.004
Standard Deviation	0.067

Table 5.2: SHOCK - CV Results

5.2 SVM with ML step

The performance with the manifold learning step heavily vary depending on two parameters: the number of **neighbors** and the number of **components** to be considered. We decided to test all the series of combination using number of **neighbors** $\in \{2, ..., 30\}$ and the number of components $\in \{2, ..., 20\}$. In the following tables we report all the useful measure for each datasets and manifold learning technique couple.

5.2.1 PPI dataset

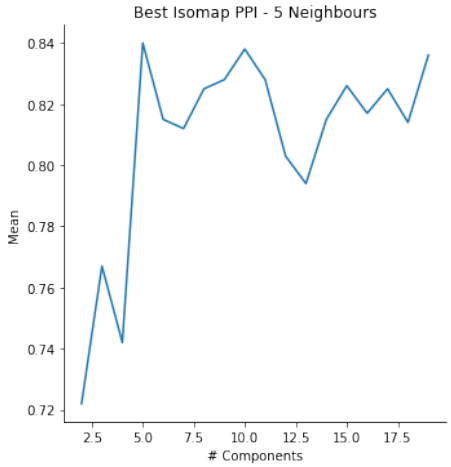
Isomap

Measure	Value
Minimum	0.556
Mean	0.84
Maximum	1.0
Variance	0.015
Standard Deviation	0.122
# Neighbors	5
# Components	5

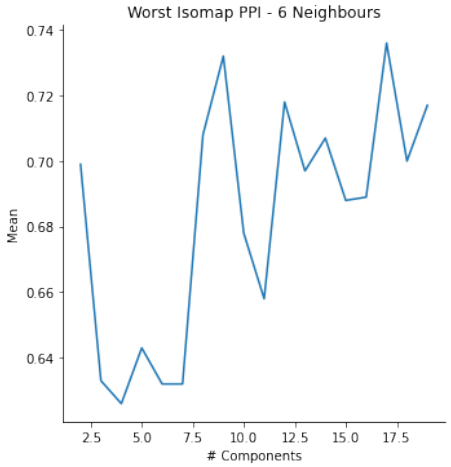
Table 5.3: PPI Isomap - Best Result

Measure	Value
Minimum	0.375
Mean	0.626
Maximum	0.778
Variance	0.014
Standard Deviation	0.119
# Neighbors	6
# Components	4

Table 5.4: PPI Isomap - Worst Result



(a) Components trend - Best Isomap PPI - 5 Neighbours



(b) Components trend - Worst Isomap PPI - 6 Neighbours

Figure 5.2: Components trend for Best and Worst Result PPI Isomap

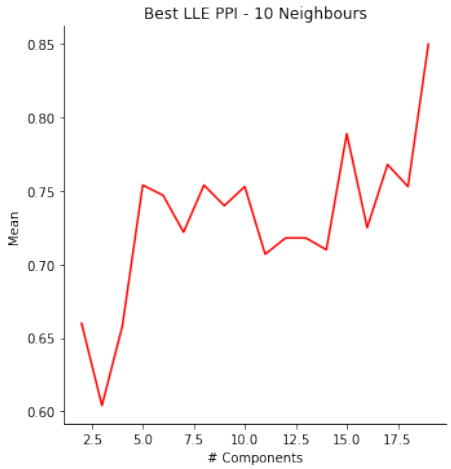
LLE

Measure	Value
Minimum	0.556
Mean	0.85
Maximum	1.0
Variance	0.013
Standard Deviation	0.113
# Neighbors	10
# Components	19

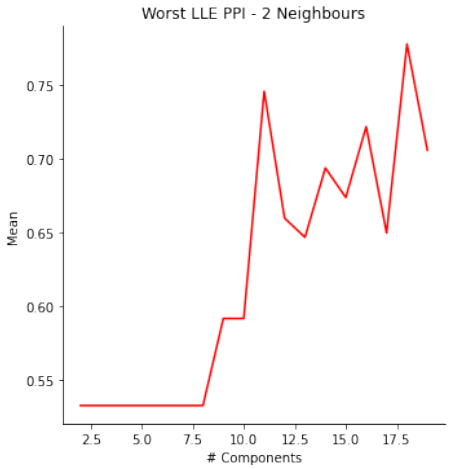
Table 5.5: PPI LLE - Best Result

Measure	Value
Minimum	0.5
Mean	0.533
Maximum	0.556
Variance	0.001
Standard Deviation	0.027
# Neighbors	2
# Components	2

Table 5.6: PPI LLE - Worst Result



(a) Components trend - Best LLE PPI - 10 Neighbours



(b) Components trend - Worst LLE PPI - 2 Neighbours

Figure 5.3: Components trend for Best and Worst Result PPI LLE

5.2.2 SHOCK dataset

Isomap

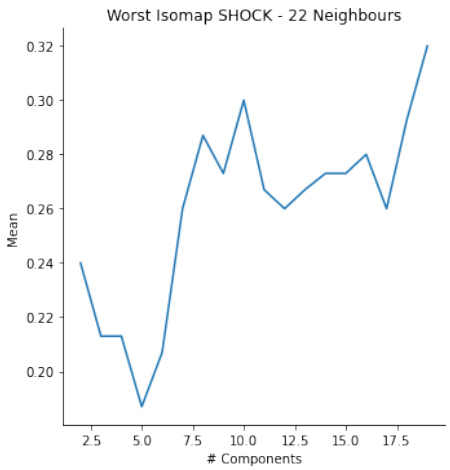
Measure	Value
Minimum	0.2
Mean	0.427
Maximum	0.667
Variance	0.014
Standard Deviation	0.12
# Neighbors	10
# Components	17

Table 5.7: SHOCK Isomap - Best Result

Measure	Value
Minimum	0.0
Mean	0.187
Maximum	0.333
Variance	0.01
Standard Deviation	0.102
# Neighbors	22
# Components	2

Table 5.8: SHOCK Isomap - Worst Result

(a) Components trend - Best Isomap SHOCK - 10 Neighbours



(b) Components trend - Worst Isomap SHOCK - 22 Neighbours

Figure 5.4: Components trend for Best and Worst Result SHOCK Isomap

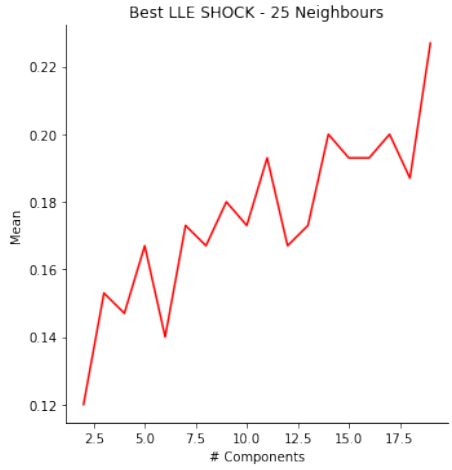
LLE

Measure	Value
Minimum	0.133
Mean	0.237
Maximum	0.333
Variance	0.006
Standard Deviation	0.074
# Neighbors	25
# Components	19

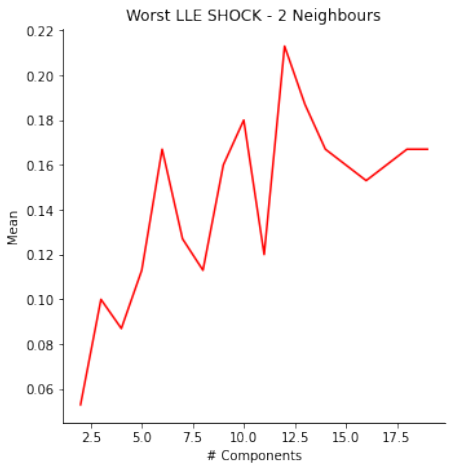
Table 5.9: SHOCK LLE - Best Result

Measure	Value
Minimum	0.0
Mean	0.053
Maximum	0.2
Variance	0.005
Standard Deviation	0.065
# Neighbors	2
# Components	2

Table 5.10: SHOCK LLE - Worst Result



(a) Components trend - Best LLE SHOCK - 25 Neighbours



(b) Components trend - Worst LLE SHOCK - 2 Neighbours

Figure 5.5: Components trend for Best and Worst Result SHOCK LLE

Chapter 6

Conclusion

As we can see from the best and worst performance in the previous chapter, the usage of a manifold learning technique it does not bring with it an high improvement of the final result, unless we are able to find the right couple of *(# neighbours, # components)*. In summary we can say that even if the set manifold learning techniques can be useful, we have to take into account also the tuning of the parameters, which lead us to an other main topic of machine learning.

But before drawing conclusions, we have a look to the two datasets separately. By first talking about PPI dataset we see that both the best result of Isomap and LLE increase the result of the linear SVM without the manifold learning step, respectively of 13.4% for Isomap (from 70.6% up to 84%) and of 14.4% for LLE (from 70.6% up to 85%). On the other hand both worst results had a decrease in performance respect to the linear SVM of the 8% for Isomap and of 17.3% for LLE. Further taking into account the SHOCK dataset we see that Isomap out perform LLE in the comparison of best result (42.7% against 18.7%), increasing the overall accuracy of the model of 11.4%. Whereas both worst result have bad overall performance.

The worst results of both datasets are obtained by the execution of manifold learning algorithms with more on less a very small number of neighbours, from 2 to 6 (except for the Shock dataset with Isomap that has 22 neighbours). This is a correct result since as expected consider a too few neighbours is not a great choice since they give little information on the structure of the graph, hence leads to a poor classification accuracy. This of course depends on the structure of the graph and also on the manifold learning technique. For example in the PPI dataset the best result for Isomap is from few neighbours whereas for LLE the best result come from an higher number. Regarding SHOCK dataset the best results of for both techniques are from a relative high number of neighbours. This makes sense because the first dataset has only two classes and the second one has 150 elements divided by 10 classes, so it has not a huge amount of data to describe the problem. Moreover we note that LLE makes a higher use of neighbours respect to Isomap since is more accurate in preserving local structure.

To conclude we remark the fact that the usage of manifold learning technique in order to improve the graph kernel can lead to a better classification accuracy, with the drawback of computing a sort of *grid search* to find the best hyper-parameters. In other word, through the usage of manifold learning we optimize our result, but at the same time we are not sure what parameters value could improve the overall result, unless we try a number of sufficient combinations.

Bibliography

[1] Paper With Code. *PPI (Protein-Protein Interactions (PPI))*. URL: https://paperswithcode.com/dataset/ppi.

[2] Persi Diaconis, Sharad Goel, and Susan Holmes. "Horseshoes in multidimensional scaling and local kernel methods". In: *The Annals of Applied Statistics* 2.3 (Sept. 2008). DOI: 10.1214/08-aoas165. URL: https://doi.org/10.1214%2F08-aoas165.

[3] PRATEEK JOSHI. *What Is Manifold Learning?* URL: https://prateekvjoshi.com/2014/06/21/what-is-manifold-learning/.

[4] Oliver Stegle Karsten Borgwardt. *An Introduction to Graph Kernels*. URL: https://ethz.ch/content/dam/ethz/special-interest/bsse/borgwardt-lab/documents/slides/CA10_GraphKernels_intro.pdf.

[5] Ibrahim Kovan. *Preserving Geodesic Distance for Non-Linear Datasets: ISOMAP*. URL: https://towardsdatascience.com/preserving-geodesic-distance-for-non-linear-datasets-isomap-d24a1a1908b2.

[6] Christopher Morris, Kristian Kersting, and Petra Mutzel. "Global Weisfeiler-Lehman Graph Kernels". In: *CoRR* abs/1703.02379 (2017). arXiv: 1703.02379. URL: http://arxiv.org/abs/1703.02379.

[7] Blaine Nelson Pavel Laskov. *Kernel Methods for Structured Inputs*. URL: http://www.ra.cs.uni-tuebingen.de/lehre/ss12/advanced_ml/lecture9.pdf.

[8] Sam T. Roweis and Lawrence K. Saul. "Nonlinear Dimensionality Reduction by Locally Linear Embedding". In: *Science* 290.5500 (2000), pp. 2323–2326. DOI: 10.1126/science.290.5500.2323. eprint: https://www.science.org/doi/pdf/10.1126/science.290.5500.2323. URL: https://www.science.org/doi/abs/10.1126/science.290.5500.2323.

[9] Lawrence K. Saul Sam T. Roweis. *Locally Linear Embedding*. URL: https://cs.nyu.edu/~roweis/lle/.

[10] Lawrence K. Saul and Sam T. Roweis. "Think Globally, Fit Locally: Unsupervised Learning of Low Dimensional Manifolds". In: 4.null (Dec. 2003), pp. 119–155. ISSN: 1532-4435. DOI: 10.1162/153244304322972667. URL: https://doi.org/10.1162/153244304322972667.

[11] Thomas Sebastian, Philip Klein, and Benjamin Kimia. "Recognition of shapes by editing shock graphs". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 26 (June 2004), pp. 550–571. DOI: 10.1109/TPAMI.2004.1273924.

[12] Andrea Torsello. *Graph Kernel*.

[13] Wikipedia. *Curse of dimensionality*. URL: `https://en.wikipedia.org/wiki/Curse_of_dimensionality`.

[14] Wikipedia. *Graph Kernels*. URL: `https://en.wikipedia.org/wiki/Graph_kernel`.