Artificial Intelligence: knowledge representation and planning

Year: 2017/2018

Assignment 3

Manifold Learning and Graph Kernels

Author: **Antonio Emanuele Cinà**

13th February 2019

Index

1 Introduction

The first goal of this assignment is to compare the performance obtained from a learning algorithm on raw data and the one obtained following an approach of non-linear dimensionality reduction, called **manifold learning**, on data. The main idea of the second approach is that probably an *high-dimensional space* can introduce some noise, instead a *low-dimensional space* can represent better the data points.
The second interesting topic that is covered by this assignment is that data are not expressed with the vector data model, but they are expressed using the graph model and for that reason we use the **graph kernels** to deal with them. To better understand the theory behind this assignment it's suggested to read the article *Unfolding Kernel Embeddings of Graphs: Enhancing Class Separation through Manifold Learning*, which present a way to improve the discriminative power of graph kernels.

Compare the performance of an SVM trained on the given kernel, with or without the manifold learning step, on the following datasets:

- **PPI**, consists of protein-protein interaction (PPIs) networks related to histidine kinase, which is a key protein in the development of signal transduction. If two proteins have direct (physical) or indirect (functional) association, they are connected by an edge. Here we consider the PPIs from two different groups: 40 PPIs from `Acidovorax` and 46 PPIs from `Acidobacteria`.

- **Shock**, consists of graphs from a database of 2D shapes. Each graphs is a skeletal-based representation of the differential structure of the boundary of a 2D shape. There are 150 graphs divided into 10 classes, each containing 15 graphs.

Datasets	**#graphs**	**#classes**	**min #vertices**	**avg #vertices**	**max #vertices**
PPI	86	2	3	106.60	232
Shock	150	10	4	13.16	33

Table 1: Datasets summary.

We will see that there are different graphs kernels and different manifold learning techniques, but in particular this document will provide, analyze and discuss the results obtained using the *Shortest-path* and *Weisfeiler-Lehman* kernels, with the application of the two manifold learning techniques *Isomap* and *Local Linear Embedding*. The two approaches will be analyzed in theory and also on the obtained results.

This document is organized as follow: on the first part we will focus our attention to graph kernel theory and the implementation of the chosen methods, secondly we will discuss about the theory of manifold learning, and finally we will analyze the results provided by the different procedures and we will compare them.

2 Feature Vector

A **vector** is a sequence of numbers that can be projected in a particular space. In machine learning, we call **feature vector** a vector of multiple characteristics of entities taken from feature space. One of the first step managed by learning algorithms is the **feature detection**, which is intrinsically depended on the entities that we want to model and the problem that we want to address. Examples of feature vectors can be:

- **Image processing**: color, gray-scale intensity, edges, areas, gradient magnitude, etc.
- **Speech recognition**: sound lengths, noise level, noise ratios, etc.

Machine learning algorithms typically require a numerical representation of objects in order for the algorithms to do processing and statistical analysis. Feature extraction is not always so directly, and can compromise a very deep analysis or introduction of different techniques to extract them. A simple case is represented by text, from a mathematical point of view it is not directly and easy to transform words in numbers. A common way to extract features from text is to adopt the bag of words model, discussed on the previous assignment, in which we associate to each word a numeric ID and we increase position ID of the feature vector as much time as the word appear into the document.

2.1 Graph model

Another way to represent entities is to structure them in form of graphs:

Definition A graph G consists of an ordered set of n vertices $V = \{v_1, v_2, \ldots, v_n\}$, and a set of directed edges $E \subset V \times V$. When G is unweighted, it means no weights on the edges, it could be represented by a $n \times n$ matrix A, with $A_{ij} = 1 \quad \text{if} \quad (i,j) \in E$. For weighted graphs the matrix A is composed by $A_{ij} = w_{ij}$.

In our case we are dealing with unweighted and undirected graphs, which means that the matrix A is symmetric and this observation will be interesting later when we will talk about shortest-path kernel.
Graph-based representations is largely used in field like: biology, social science, link analysis, etc due their ability to characterize in a natural way a large number of complex relations. Graph model has the advantage of improving the expressiveness and versatility of our models, but it introduce a new problem: how can we use learning algorithms over them? We have seen that common machine learning algorithms are based on the usage of vectorial form, and there is not a perfect definition on how we can convert a graph into a vector. Graphs have an arbitrary structure: they are collections of nodes and edges without a location in space, or with an arbitrary location. They have no proper start-node and end-node, and two nodes connected to each other are not necessarily close, since there are different ways to represent the same graph structure.
Due to the difficulty of converting a graph into a vector, it is necessary to overcome the problem giving a method that allows to represent graphs into a space for learning purpose. The solution adopted in this assignment is based on the usage of the **kernel trick**, which allows us to represent graphs not in their vectorial representation but using their similarity representation. Informally, a kernel is a function of two objects that measures their similarity, instead from a mathematical point of view it corresponds to an inner product in a reproducing kernel *Hilbert space*.

$$K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$$

Thanks to kernel trick it is not necessary to perform feature extraction from graphs to transform them into feature vectors, which are complex or don't have a precise definition, and we can apply standard learning techniques (like SVM) on data using a defined similarity measure.
In our case, since we are dealing with graphs, it is not a simple issue since there are no well-define similarity measures between them. This similarity measure can be expressed as a function $s : G \times G \to \mathbb{R}$ of two graphs $G1$ and $G2$ that measures their similarity or dissimilarity. Common algorithm for deciding the similarity between two graphs are based on the concept of graph isomorphism, giving a simple binary similarity measure that returns 1 if they are equal and 0 otherwise.
Leaving for a moment the fact that isomorphism problem is an NP-Hard problem, and its computation has exponential time, we can analyze another problem of this approach. A binary similarity function has the drawback that it does not give any idea of how the two graphs are similar or dissimilar. For our purpose we are looking for a more complex measure that gives us more valuable results. A simple example is provided as follow:

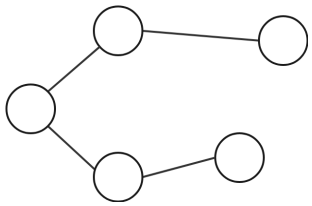

Figure 1: Graph G1.

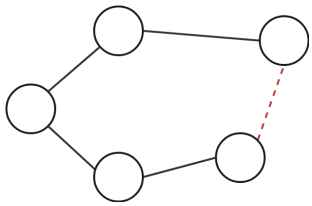

Figure 2: Graph G2.

Here we can see that the two graphs are not the same, since G2 has an extra edge, but we can see that they are more or less similar. Considering only isomorphism as similarity we have 0 as result, instead we would like to define a new positive and simmetric similarity measure such that $K(G1, G2) = K(G2, G1) \geq 0$.

3 Graph Kernel

In section 2.1 we have seen how data can be expressed in graphs and thanks to a technique called **kernel trick** we can use standard learning algorithm over them. **Graph kernels** are used for computing the similarity between pairs of graphs, based on common substructures they share and that can be computed in polynomial-time. Graph kernels are based on the idea of extrapolating patterns from the graphs and compare them, as follow it is presented an example of patterns that could be considered by a kernel.

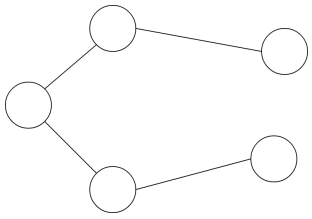

Figure 3: Graph G1.

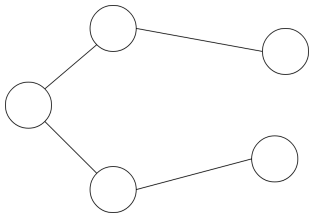

Figure 4: Graph G2.

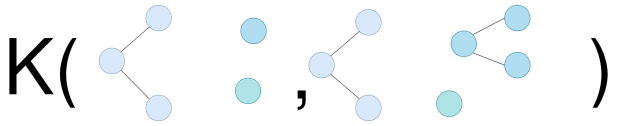

Figure 5: Kernel function between graph patterns

Since there isn't a unique and right way to describe similarity between graph, in literature are proposed different kernels that stand out by the way in which they localize patterns and compare them. An essential requirement for defining a kernel is that it must be a *positive definite kernel*. The most used ones are so proposed:

- **Shortest-path Kernel**
- **Random Walk Kernel**
- **Graphlet Kernel**
- **Weisfeiler-Lehman Kernel**

Selecting the right kernel for a given application is a practical hurdle when applying kernel methods in practice. For this reason and to give a better idea of the various techniques, we will deal with two different graph kernels, shortest-path and Weisfeiler-Lehman, we will analyze their results on two datasets and their computational costs.

3.1 Shortest-Path Kernel

In literature we can find different kernels which base their similarity measure on the structure of the graph, like number of triangles, paths or walks. The idea of these approaches is to get and summarize the structure information of a given graph in something that can possibly represent the graph structure. A very well know approach consists on enumerating all the paths of the two graphs and compare the similarity between the paths. The drawback of this solution is given by the time complexity required by the kernel, since until now the problem of enumerating all the paths in a graph is NP-hard and no polynomial time algorithm exists for it.
A similar approach consists not on computing all the paths, but only the shortest ones in the graph. Computing shortest paths in a graph, however, is a problem solvable in polynomial time. Very well known algorithms such as Dijkstra, for shortest paths from one source node, or Floyd-Warshall, for all pairs of nodes, allow to determine shortest distances in polynomial time. From this idea of enumerating all the shortest paths and compare them between the two graphs comes the *Shortest-Path Kernel*.

Definition Let $G = (V, E)$ be an undirected graph and let S be its shortest-path graph,i.e., a graph defined over the same set of nodes V where there exist an edge between two nodes if these are connected by a walk in G. The shortest-path kernel counts the numbers of common shortest-path in two graphs G_1 and G_2, that first are transformed into their corresponding shortest-path graphs $S_1 = (V_1, E_1)$ and $S_2 = (V_2, E_2)$ and then the kernel similarity is defined as:

$$K_{sp}(S_1, S_2) = \sum_{e_1 \in E_1} k(e_1, e_2)$$

where k is an arbitrary positive definite kernel that measures the similarity between the shortest paths corresponding to the edges e_1 and e_2.

The shortest-path kernel algorithm is very simple and can be summarized in few steps:

- Compute from the adjacency matrix of G_1 and G_2 the corresponding shortest paths (S_1 S_2), using a polynomial-time algorithm such as Floyd-Warshall.
- Define a similarity measure between paths and compare the set of paths previously generated. An example of measure could be the number of nodes in the shortest-path, or the number of nodes labels that are in common for each path.

3.2 Floyd-Warshall algorithm

We have seen that the first step useful for computing the shortest path kernel between two graphs consists on computing all their shortest paths. For this problem, a number of algorithms is known. Here it is provided a simple table that can summarize the time complexity required for each algorithm.

G Structure \ Algorithm	**Dijkstra**	**Bellman Ford**	**Floyd Warshall**
Sparse graph	$\mathcal{O}(\|V\|^2 \log \|V\|)$	$\mathcal{O}(\|V\|^3)$	$\mathcal{O}(\|V\|^3)$
Dense Graph	$\mathcal{O}(\|V\|^3 \log \|V\|)$	$\mathcal{O}(\|V\|^4)$	$\mathcal{O}(\|V\|^3)$

Table 2: Shortest path algorithms complexities.

We can see that the best case is given by Dijkstra's algorithm over sparse graphs, instead Floyd-Warshall algorithm maintains the same complexity in both of the two cases. In principle we can analyze the graphs dataset and decide what algorithm could give better performance, but in order to give good performance in general the Floyd-Warshall's algorithm could be an optimal compromise.

Algorithm 1 Floyd-Warshall algorithm.

```
1:  procedure FW(G)                                              ▷ G = (V, E)
2:      Let dist be a |V| × |V| array of minimum distances initialized to ∞.
3:      for each edge (u, v) do
4:          dist[u, v] ← w(u, v)                    ▷ w(u, v) weight of the edge (u, v)
5:      for each vertex v do
6:          dist[v, v] ← 0
7:      for k from 1 to |V| do
8:          for i from 1 to |V| do
9:              for j from 1 to |V| do
10:                 if dist[i, j] > dist[i, k] + dist[k, j] then
11:                     dist[i, j] = dist[i, k] + dist[k, j]
```

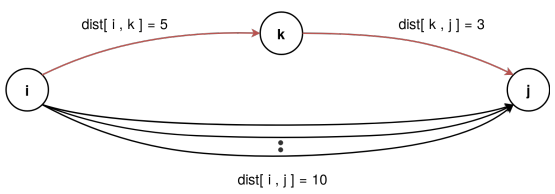

Figure 6: Floyd-Warshall path decision.

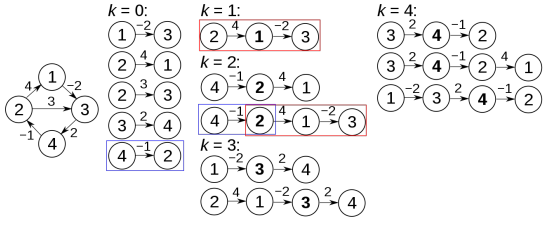

Figure 7: Floyd-Warshall iteration.

The major characteristic of the FW's algorithm is that at each iteration it discovers new paths, that pass from node k, and picks update the shortest paths considering the min between the current shortest path and the new one. So at the end of the procedure each node is connected to the others with a shortest path algorithm.

An important aspect of the Floyd-Warshall's algorithm is that it can be optimized if the adjacency matrix is symmetric. In our case the all the graphs are undirected graphs with weights 0 or 1, this means that we have a symmetric adjacency matrix and when we compute the path $dist[i, j]$ we can immediately conclude that $dist[j, i] = dist[i, j]$. The improved version so tries to address this property reducing the number of iterations required by the algorithm.

Algorithm 2 Floyd-Warshall algorithm for symmetric graphs.

```
1:  procedure symmetric-FW(G)                                   ▷ G = (V, E)
2:      Let dist be a |V| × |V| array of minimum distances initialized to ∞.
3:      for each edge (u, v) do
4:          dist[u, v] ← w(u, v)                    ▷ w(u, v) weight of the edge (u, v)
5:      for each vertex v do
6:          dist[v, v] ← 0
7:      for k from 1 to |V| do
8:          for i from 1 to |V| do
9:              for j from i to |V| do
10:                 dist[i, j] = dist[j, i] = min(dist[i, j], dist[i, k] + dist[k, j])
```

3.3 Similarity between paths

Once the shortest paths matrices are computed it is necessary to implement the similarity function k between them:

$$k : S_1 \times S_2 \to \mathbb{R} \qquad S_1 = |V_1| \times |V_1| \qquad S_2 = |V_2| \times |V_2|$$

In theory there is not a similarity measure between paths, but what can be done is to preserve structure/topological information about them in vectors and compute their similarity. In particular I decided to analyze two different approaches:

Steps Kernel The step kernel k_{steps} measures the similarity between two shortest paths graphs considering for each node the sum of all shortest paths always starting from the source node. Here it is presented an example that considers two distinct graphs(2,6) taken from the PPI dataset and belonging into the acidovorax class.

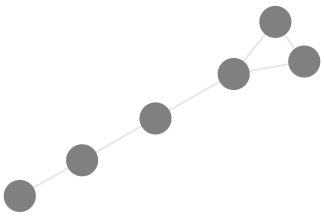

Figure 8: Graph_1 adjacency matrix.

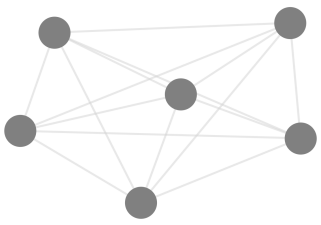

Figure 9: Shortest paths matrix.

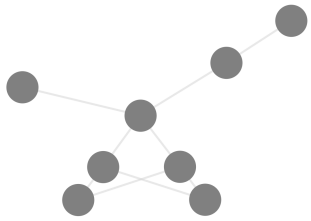

Figure 10: Graph_2 adjacency matrix.

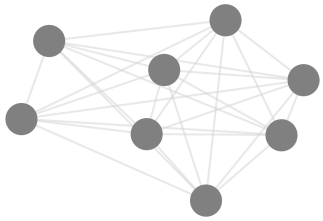

Figure 11: Shortest paths matrix.

Below are given the shortest paths matrices computed from G_1 and G_2, which are graphically represented on in Figure 21 and Figure 11:

$$SP_1 = \begin{bmatrix} 0 & 1 & 1 & 3 & 4 & 2 \\ 1 & 0 & 1 & 2 & 3 & 1 \\ 1 & 1 & 0 & 3 & 4 & 2 \\ 3 & 2 & 3 & 0 & 1 & 1 \\ 4 & 3 & 4 & 1 & 0 & 2 \\ 2 & 1 & 2 & 1 & 2 & 0 \end{bmatrix} \qquad SP_2 = \begin{bmatrix} 0 & 1 & 1 & 2 & 2 & 3 & 3 & 2 \\ 1 & 0 & 2 & 3 & 3 & 4 & 4 & 3 \\ 1 & 2 & 0 & 1 & 1 & 2 & 2 & 1 \\ 2 & 3 & 1 & 0 & 2 & 3 & 3 & 2 \\ 2 & 3 & 1 & 2 & 0 & 1 & 1 & 2 \\ 3 & 4 & 2 & 3 & 1 & 0 & 2 & 1 \\ 3 & 4 & 2 & 3 & 1 & 2 & 0 & 1 \\ 2 & 3 & 1 & 2 & 2 & 1 & 1 & 0 \end{bmatrix}$$

Then the two shortest path matrices are reduced to the following vectors:

SP1	11	8	11	10	14	8	0	0
SP2	14	20	10	16	12	16	16	12

Figure 12: Vectorial reduction

Note that each position i of these two vectors contains the sum of all the shortest paths that start from node i. Since graphs can have different number of nodes, the smaller one is padded with zeros (like SP1). Then, in order to compute their similarity, it is computed the dot product between them and normalization is performed so that the resulting similarity belongs to $[0, 1]$. Since we are using dot product between positive vectors we have the guarantee that the kernel is still positive definite.

Delta Kernel The delta kernel k_δ measures the similarity between all the paths of two graphs considering the occurrences of each path's length. In other words, given a threshold δ we create a vector with dimension δ that represents the occurrences of paths with weights lower that δ. The choice of δ can be useful to decide the level of approximation that we want to use and also to reduce the computational costs. For these experiences δ is chosen equal to the maximum path's weight between the two graphs.
Below is proposed an example of how this kernel works considering the two previous graphs:

$$\delta = max\Big\{max\{SP_1\}, max\{SP_2\}\Big\} = 4$$

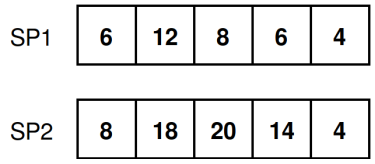

Figure 13: Delta vectors for SP_1 and SP_2

The $i-th$ cell of a frequency vectors contains the occurrences of path with weight equal to i. For example G_1 has exactly 6 path with weight equal to 4, which are:

$$\Big\{0 \to 4, 2 \to 4, 4 \to 0, 4 \to 2, 0 \to 4, 0 \to 4\Big\}$$

Once the delta vectors are computed they are normalized and then the dot product between them is computed, so that at the end we have a similarity measure between $[0, 1]$. Since we are using dot product between positive vectors we have the guarantee that the kernel is still positive definite.

3.4 Weisfeiler-Lehman Kernel

The Weisfeiler-Lehman (WL) kernel is based on the application of the WL test of isomorphism, more specifically its 1-dimensional variant, also known as *naive vertex refinement*. Initially the WL method was considered as a way to solve the graph isomorphism problem in polynomial time, but then it was discovered that there are families of non-isomorphic pairs of graphs which cannot be solved correctly in polynomial time by this method.
Assume we are given two graphs G and G' and we would like to test whether they are isomorphic, then the WL test is nothing else that an iterative procedure between the two graphs that compresses and propagates information in the nodes. In other words the key idea of the algorithm is to consider the node labels by the sorted set of node labels of neighbouring nodes, and compresses 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.
Respect to the shortest-path kernel the definition of a graph is a little bit different since now a graph G is a triplet $G = (V, E, l)$, where now l is a function returning the label of a node. In each iteration i of the Weisfeiler-Lehman algorithm 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. In our datasets, nodes of graphs are not associated to a specific label, for that reason a first step of my algorithm was to define the function l as the node degree function.

The developed instance of WL kernel is called **Weisfeiler-Lehman subtree kernel**.

Definition Let G and G' be graphs. Define $\Sigma_i \subset \Sigma$ as the set of letters that occur as node labels at least once in G or G' at the end of 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 $\Sigma_i = \{\sigma_{i1}, \ldots, \sigma_{i|\Sigma_i|}, \}$ is ordered. Define a map $c_i : \{G, G'\} \times \Sigma_i \to \mathbb{N}$ such that $c_i(G, \sigma_{ij})$ 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:

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

where

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

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

The key idea of the Weisfeiler-Lehman subtree kernel is to count the number of common original and compressed labels in two graphs. The greater is that number the more similar are the two graphs.
The developed algorithm is composed by 4 essential steps:

1. **Multiset-label determination**, for each of the two graph a label is determined. Each node label is composed by its original label and all the labels of the adjacent nodes, creating a multiset of labels.
2. **Sorting**, for each node the labels inside the multiset is sorted.
3. **Label compression**, maps each string of labels to a compressed label.
4. **Relabeling**, assigns to each node its compressed label.

Algorithm 3 One iteration of the WL subtree kernel computation on N graphs.

1: Multiset-label determination

- Assign a multiset-label $M_i(v)$ to each node v in G which consists of the multiset $\{l_{i-1}(u) | u \in \mathcal{N}(v)\}$ ▷ $\mathcal{N}(v)$ list of 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 compressed label using a hash function $f : \Sigma^* \to \Sigma$ such that $f(s_i(v)) = f(s_i(w))$ if and only if $s_i(v) = s_i(w)$.

4: Relabeling

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

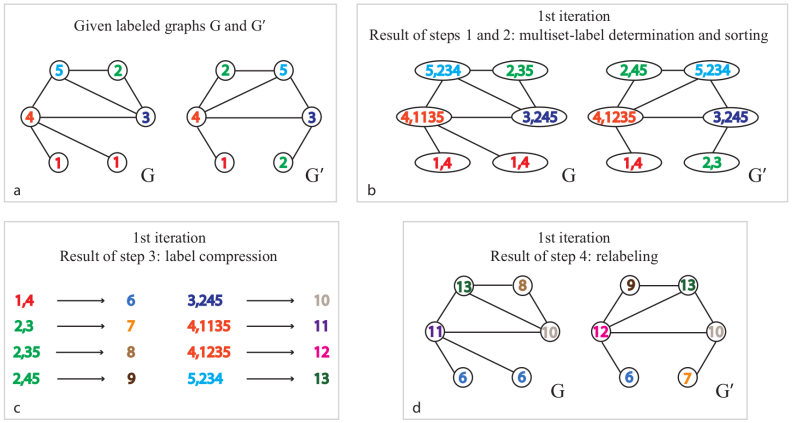

Figure 14: WL subtree kernel algorithm steps

$$\varphi^{(1)}_{WLsubtree}(G) = (2, 1, 1, 1, 1, 2, 0, 1, 0, 1, 1, 0, 1)$$

$$\varphi^{(1)}_{WLsubtree}(G') = (\underbrace{1, 2, 1, 1, 1}_{\text{Counts of original node labels}}, \underbrace{1, 1, 0, 1, 1, 0, 1, 1}_{\text{Counts of compressed node labels}})$$

$$k^{(1)}_{WLsubtree}(G,G') = \langle \varphi^{(1)}_{WLsubtree}(G), \varphi^{(1)}_{WLsubtree}(G') \rangle = 11.$$

Figure 15: End of the 1st iteration Feature vector representations of G and G'

In theory is also proved that 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 complexity here comes from the fact that computing $\phi^{(h)}_{\text{WLsubtree}}$ has a complexity equal to $O(Nhm)$, assuming that $m > n$. The number of iteration h defines how much the information between nodes are propagated, so the greater it is the more information about the graph we have. In case h is limited by an upper bound beyond which no extra information is extracted and the algorithm reaches a stationary point.

4 Manifold Learning

Real-world data, such as speech signals, images, ecological data or data on health status, usually has a high dimensionality, which means that they are composed by a large set of features. Intuitively we could think that the greater is the level of information that we have the best are the results, but this statement is not always good. When analyzing huge sets of numerical data, problems often occur when the raw data are high-dimensional. This problem was firstly discussed by Richard E. Bellman who gives to it the name of **curse of dimensionality**, which refers to the problem of finding structure in data embedded in a highly dimensional space. In other words, when the dimensionality of data increases the volume of the space increases too and available data become sparse, and the result of this situation is that all data points appear to be sparse and dissimilar in many ways. Many algorithms, like KNN in which we need to find the nearest neighbor, tend to perform very well with low-dimensional data, but their complexity increase when we increase dimensionality of data. Note also that for humans it is not possible to perceive data in a very **high-dimensional space**, so reducing the dimensionality of data to two or three and visualizing embedded data has become increasingly useful for multivariate analysis.
In literature we can find different dimensionality reduction techniques widely used for the analyzing and visualize complex sets of data. These techniques can be classified in linear or non-linear ones. One of the first and most common method of dimension reduction is **principle component analysis** (PCA), which is basically a linear technique based on preserving the maximum explained variability. PCA finds the directions along which the data has maximum variance in addition to the relative importance of these directions. The main assumption of PCA is that the principle components are a linear combination of the original features. If this is not true, PCA could not provide accurate results.
However, nonlinear structures are rather common in real data and it is necessary to introduce more sophisticated models like **manifold learning**. Manifold learning algorithms are based on the idea that the dimensionality of many data sets is only artificially high. Although the data

points may consist of thousands of features, they may be described as a function of only a few underlying parameters. That is, the data points are actually samples from a low-dimensional manifold that is embedded in a high-dimensional space. Manifold learning algorithms attempt to extract a low-dimensional representation that represent as better as possible high-dimensional data.

The most famous manifold learning technique are:

- **Isomap**
- **Diffusion Map**
- **Laplacian Eigenmaps**
- **Local Linear Embedding**

The goal of this assignment is to use different manifold learning techniques to the distance matrix in order to increase if possible the class separation. The distance matrix $D = (d_{ij})$ is computed starting from the kernel matrix $K = (k_{ij})$ obtained from the previous explained graph kernels over a set of n graphs. In particular D is computed as follow:

$$d_{ij} = \sqrt{k_{ii} + k_{jj} - 2k_{ij}}$$

Then a linear SVM is used for learning data and class separation. The rest of the document will provide some theoretical knowledges about two manifold learning techniques, isomap and local linear embedding, and performance over the two dataset.

4.1 Isomap

Isomap, short for Isometric Feature Mapping, has been one of the first algorithms introduced for manifold learning and has gained significant popularity due to its conceptual simplicity and efficient implementation. Isomap can be seen as a generalization of **Multidimensional Scaling** (MDS) where the distance metric is the **geodesic distances**. Geodesics are lines of shortest length between points on a manifold, which act like straight lines on a plane.
The key idea of geodesic distance is that it preserves global isometries. Multi-dimensional data space is often curved rather than Euclidean and the simplest example is given by data points placed along a sphere, like Figure 16.

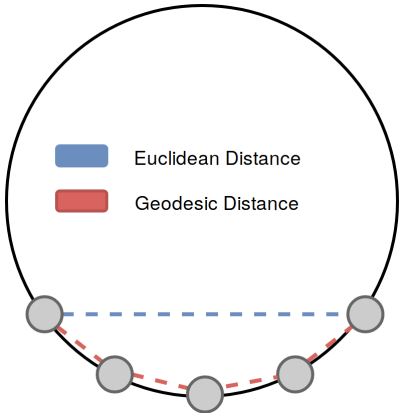

Figure 16: Distances on a sphere manifold.

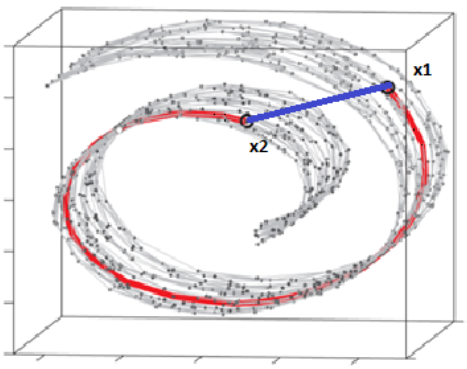

Figure 17: Distances on a real data manifold.

From Figure 16 we can see that distances among the points are best measured by the geodesic distance since it preserves distances along the surface of the sphere rather than the direct measure

through the sphere. A simple approximation of the geodesic distances assumes that the distance between close points can be well approximated by the Eucledian distance and the distance between points which are farther is estimated considering the length of the shortest path between them. In other words the curvature of the space can be approximated through a network, then:

- if nodes i and j are close then the effect of curvature is minimal and the Eucledian distance is a good estimator for the geodesic distance.
- if nodes i and j are farther then we can have a strong curvature along the manifold so the geodesic distance is estimated as the length of the minimal path between i and j on a neighborhood graph.

Given a collection of n objects on which a distance function between objects $\delta_{i,j}$ is defined, we compute the dissimilarity matrix Δ, which is so defined:

$$\Delta = \begin{pmatrix} \delta_{1,1} & \delta_{1,2} & \cdots & \delta_{1,n} \\ \delta_{2,1} & \delta_{2,2} & \cdots & \delta_{2,n} \\ \vdots & \vdots & & \vdots \\ \delta_{n,1} & \delta_{n,2} & \cdots & \delta_{n,n} \end{pmatrix}$$

where $\delta_{i,j}$ = distance between i-th and j-th objects.
Then the goal of MDS is to find n vectors $x_1, \ldots, x_n \in \mathbb{R}^N$ starting from Δ, such that:

$$||x_i - x_j|| \sim \delta_{i,j} \qquad \forall i, j \in 1, \ldots, n$$

The vectors $x_1, \ldots, x_n$ represents the embedding from the objects into a new space $\mathbb{R}^N$ in which distances are preserved.
Respect to the MDS procedure Isomap uses the geodesic distance as distance function, with the goal of preserving global isometries and distances between points.

Isomap algorithm can be summarized in the following steps:

- Step 1: Given data set $x_i \in R^D$, form the weighted **k**-nearest neighbor graph G. That is put edge between vertices x_i and x_j if x_j is one of the **k** nearest neighbors of x_i or vice versa in the Euclidean distance. It assigns each edge (x_i, x_j) weight $kx_i - x_jk$.
- Step 2: Compute the shortest path distances between all pair of vertices x_i and x_j store it in S_{ij}. This can be done using Floyd-Warshall's or by Dijkstra's algorithm and storing the square of the shortest path into distance matrix $D_{ij} = S_{ij}$.
- Step 3: Apply MDS algorithm with dissimilarity matrix D from previous step.

Isomap seems to not perform well when manifold is not well sampled and contains holes, for instance when we have disjoint parts. From a computation point of view neighborhood graph creation is tricky, expensive and slightly wrong parameters can produce bad results since the use of geodesic distances makes Isomap strongly dependent to the topology of the neighborhood graph.

4.2 Local Linear Embedding

Local Linear Embedding (LLE) is another strategy that can be adopted for nonlinear dimensionality reduction problems. The idea of this manifold technique is to reconstruct each data point on the manifold from its neighbors, provided that there is a sufficient amount of data. The expected behavior of a manifold is that each data point and its neighbors to lie on or close to

a locally linear patch of the manifold. In the simplest formulation of LLE identifies K nearest neighbors per data point, using metrics like Euclidean distance, and then the following cost function is evaluated:

$$\epsilon(W) = \sum_i |X_i - \sum_j W_{ij} X_j|^2$$

where the weights W_{ij} indicate the contribution of each data point to the reconstruction. There is a unique constraint on the weights:

$$\sum_j W_{ij} = 1$$

The proposed cost function adds up the squared distances between all the data points and their reconstructions, so the lower it is the better is the approximation ability of the K points. The goal of LLE is to solve a constraint problem that minimize the cost function $\epsilon(W)$. The desirable behavior is that the local geometry in the original data space is maintained equally valid for local patches on the manifold. In particular, the same weights W_{ij} that reconstruct the i-th data point in dimensions D should also reconstruct its embedded manifold coordinates in d-dimensions, with $d \ll D$.

An important theoretical result is that the optimal weights W are invariant to some transformation like rotation, re-scaling and translations to all the data points in the manifold.

In summary the LLE algorithm can be shortly summarized in the following steps:

- Compute the neighbors of each data point x_i
- Compute the weights W_{ij} that best reconstruct each data point x_i from its neighbors, minimizing the cost $\epsilon(W)$.
- Each high dimensional observation x_i is mapped to a low dimensional vector y_i representing global internal coordinates on the manifold. This is done by choosing d dimensional coordinates y_i to minimize the embedding cost function:

  $$\phi(y) = \sum_i |y_i - \sum_j W_{ij} y_j|^2$$

  which indicates the locally linear reconstruction error fixing the weights W_{ij}. The focus of the optimization problem is to find the coordinated y_i that minimize the cost function $\phi(y)$.

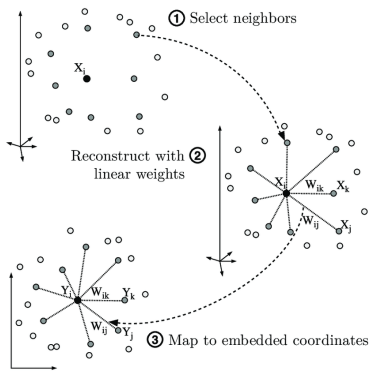

Figure 18: Distances on a sphere manifold.

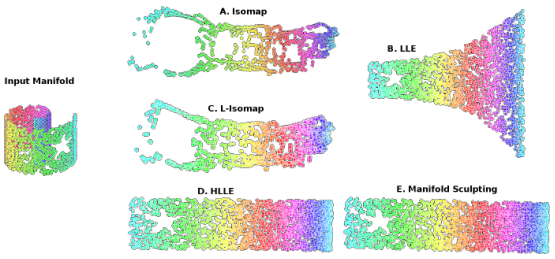

Figure 19: Distances on a real data manifold.

As we can see from Figure 19 the presence of an hole in the manifold produce not well results, instead LLE seems to be more robust for the manifold embedding.

5 Experimental Results and Comparison

In this section it is presented and analyzed the different results provided by the combination of the various kernels, parameters and manifold learning techniques on the two datasets of graphs (PPI and Shock). First of all are proposed the results obtained without the application of any manifold learning technique, so the resulting scores will be retrieved from an SVM over the graph kernel distances. Secondly are proposed the results with the two techniques of manifold learning (Isomap, LLE) will be provided. At the end of the section it is analyzed the general behavior of the applied techniques and possible patterns or dependencies.

5.1 PPI dataset

First are proposed the results obtained from the PPI dataset.

Kernel	**min score**	**avg score**	**max score**	**stand dev**
SP kernel - step	0.5	0.692	0.877	0.133
SP kernel - delta	0.625	0.779	0.888	0.098
WL_1 *Kernel*	0.666	**0.8138***	1	0.132
WL_2 *Kernel*	0.555	0.775	1	0.113
WL_3 *Kernel*	0.5	0.752	0.888	0.189

Table 3: PPI classification accuracy and standard error without manifold learning.

		Isomap		**LLE**	
#neighbors	*#dimensions*	*avg*	*std*	*avg*	*std*
1	6	0.511	0.042	0.567	0.069
1	17	0.547	0.128	0.719	0.062
1	18	0.524	0.116	**0.779***	**0.108***
4	11	0.664	0.181	0.607	0.109
4	18	**0.781***	**0.128***	0.735	0.108
4	24	0.699	0.181	0.732	0.113
10	23	0.662	0.151	0.757	0.159
16	17	0.733	0.184	0.71	0.123
19	4	0.701	0.13	0.556	0.096
19	16	0.721	0.149	0.708	0.083

Table 4: PPI classification accuracy and std with m.l. and shortest path kernel.

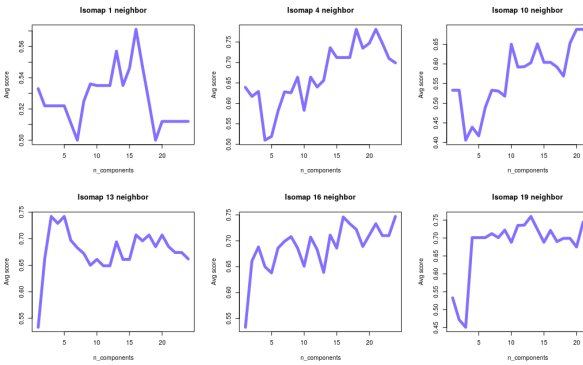

Figure 20: Isomap performance.

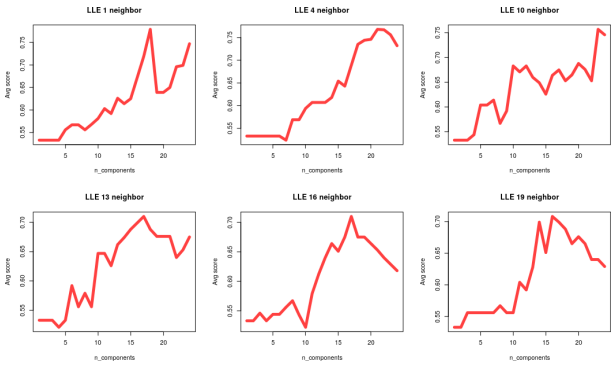

Figure 21: LLE performance.

		Isomap		**LLE**	
#neighbors	*#dimensions*	*avg*	*std*	*avg*	*std*
1	17	0.476	0.108	0.593	0.077
4	11	0.724	0.134	0.636	0.093
4	18	0.7	0.145	0.672	0.112
4	24	0.606	0.177	0.721	0.096
10	8	0.708	0.127	0.719	0.08
10	17	**0.815***	**0.112***	0.718	0.102
16	9	0.749	0.162	0.662	0.108
16	17	0.719	0.112	**0.79***	**0.069***
19	4	0.674	0.099	0.661	0.071
19	16	0.676	0.152	0.71	0.088

Table 5: PPI classification accuracy and std with m.l. and Weisfeiler-Lehman kernel.

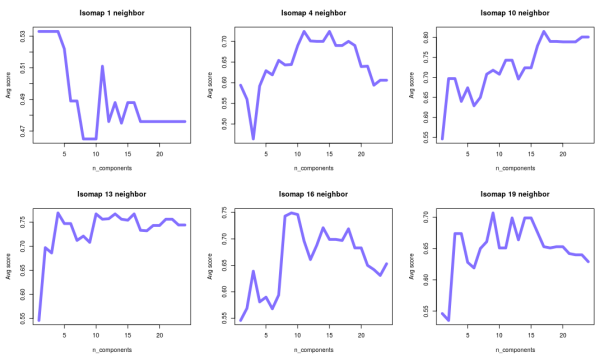

Figure 22: Isomap performance.

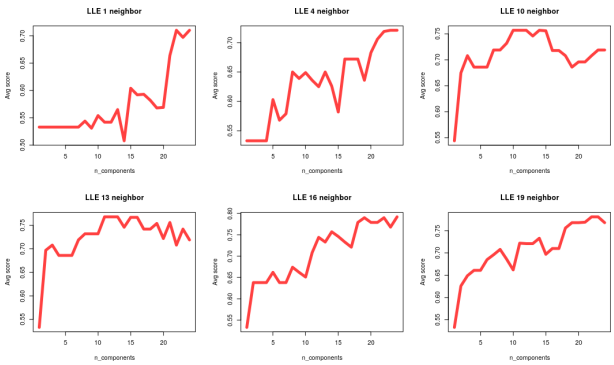

Figure 23: LLE performance.

For each table the minimum, average, maximum scores and standard deviation are obtained performing a 10-Way cross validation and the accuracy scores noted with the symbol $*$ are the best ones. From Table 3 we can see that, considering the shortest path kernel, the best scores are provided by the delta kernel, that as we have seen it keeps information about the number of nodes in the graphs and number of shortest path per weight. The results obtained by the SVM without manifold learning give us the idea that the WL kernel tends to better describe the similarity between the graphs in the manifold, since we can see that in general the best maximum results are obtained from it. An interesting observation on WL kernel is that we can see that increasing the depth h (number of iterations) the scores do not improve, meaning that the kernel tries to consider new subtrees, reducing the locality information, that are not useful. Table 4 is obtained considering the shortest path kernel and the two manifold learning techniques, then different results are proposed in order to better understand how the results change with different numbers of neighbors and dimensions. Combining the results obtained from Table 4, Figure 22 and Figure 23 we can see that the best results are obtained increasing the number of components, that result is given by the fact that we are giving more degrees of freedom to the embedding algorithms. The choice of the number of neighbors seems to be strongly dependent on the number of components, since when we increase the number of neighbors the best results are provided with an high number of components. The unique difference is given by LLE, since we can see from Figure 23 that when we increase the number of neighbors, having too high number of components seems to be not efficient and we have a decrease on the accuracy scores. Considering instead the comparison between the two manifold techniques Isomap seems to produce relative better results respect to LLE.
Table 5, Figure 22 and Figure 23 are obtained considering the WL kernel with depth equals to 2 ($h = 2$). Here we can see some differences respect to the previous analysis. Isomap provides again the best scores, but now when the number of neighbors increases the better results are provided when the number of components remains restricted and does not increase to much, otherwise we can see a negative trend of the scores. LLE instead works very well when we increase the number of components and with high number of neighbors. From a certain point of view with LLE the most significant parameter here seems to be the number of neighbors that the algorithm use.

5.2 Shock dataset

In this section are proposed the scores obtained considering the shock dataset, which consists on graphs representing 2D shapes. The particularity of this dataset is that it is composed not just by 2 classes, like the previous one, but it contains 10 different ones. In this case the classifier can apply two different strategies:

- **one vs all**
- **one vs one**

The difference between the two strategies is the number of classifiers that it is necessary to learn, which strongly correlates with the decision boundary they create. Considering a dataset composed by N different classes with the **One vs all** classifier will train one classifier per class, so that at the end we have N classifiers. For class i it will assume i-labels as positive and the rest as negative. In **One vs one** it will learn a separate classifier for each different pair of labels, leading to $\frac{N(N-1)}{2}$ classifiers.
The problem of the first solution is that it is sensible to possible imbalanced datasets, but efficient from a computational point of view. The One vs one instead has a greater complexity but deals better with imbalanced datasets.

Kernel	**min score**	**avg score**	**max score**	**stand dev**
SP kernel - step	0.35	0.455	0.65	0.11
SP kernel - delta	0.3	0.425	0.65	0.129
WL_1 *kernel*	0.25	0.415	0.6	0.15
WL_2 *kernel*	0.3	0.41	0.6	0.15
WL_3 *kernel*	0.35	**0.459***	0.7	0.117

Table 6: Shock classification accuracy and std without m.l.

		Isomap		**LLE**	
#neighbors	*#dimensions*	*avg*	*std*	*avg*	*std*
4	1	0.265	0.092	0.19	0.086
4	6	0.435	0.087	0.27	0.051
10	6	0.36	0.099	0.44	0.087
10	18	0.35	0.155	0.375	0.163
13	2	0.315	0.084	0.23	0.075
13	5	**0.44***	**0.1***	0.44	0.089
13	13	0.4	0.147	0.4	0.128
16	6	0.35	0.105	0.4	0.089
16	16	0.395	0.117	0.45	0.13
19	11	0.4	0.112	0.38	0.117
19	17	0.385	0.129	**0.46***	**0.122***

Table 7: Shock classification accuracy and std with m.l. and shortest path kernel.

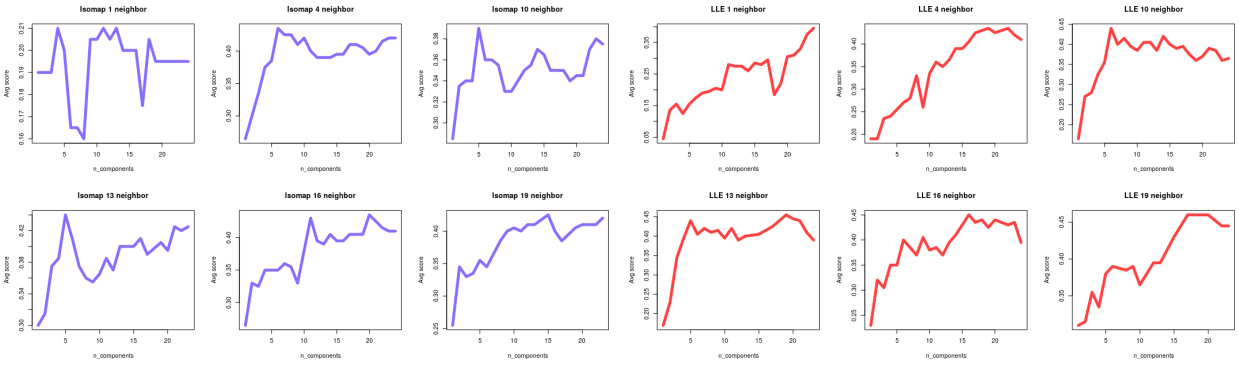

Figure 24: Isomap performance.

Figure 25: LLE performance.

In table 6 we can see that the performance applying any kind of graph kernel and then a linear SVM the scores are not good. This means that the dataset is not simple and probably using a linear classifier is not the right way to proceed and more complex techniques should be adopted. In our case we just want to evaluate the performance of applying or not a manifold learning technique independently on the type of SVM. The results suggest that the Weisfeiler-Lehman kernel produces better results increasing the depth h used by the algorithm, meaning that the considering more information coming from other subtrees of the shapes can be useful for classification and mitigate different shapes.
Table 7, which is obtained considering the shortest path kernel and the two manifold learning techniques, shows better results in terms of average score, meaning that Isomap and LLE improve class separability over the manifold. Regarding the parameters, Figure 24 and Figure 25 suggest that it is better to pick large number of components and neighbors, giving more degrees of freedom. Comparing the two results, what we can see is that in general LLE works better than Isomap, enhancing the class separability.

		Isomap		**LLE**	
#neighbors	*#dimensions*	*avg*	*std*	*avg*	*std*
4	1	0.14	0.044	0.155	0.079
4	13	0.38	0.1	0.4	0.097
4	17	0.38	0.078	0.465	0.103
10	6	0.375	0.115	0.215	0.063
10	18	0.365	0.132	0.38	0.119
13	8	0.345	0.162	0.295	0.151
13	15	0.33	0.15	0.385	0.123
13	21	0.39	0.109	0.39	0.139
16	10	0.41	0.136	0.455	0.079
19	10	0.37	0.114	**0.485***	**0.063***
21	6	0.395	0.14	0.315	0.078
21	12	0.37	0.114	0.465	0.059
26	9	0.425	0.145	0.36	0.077
26	19	**0.465***	**0.116***	0.435	0.095

Table 8: Shock classification accuracy and std with m.l. and Weisfeiler-Lehman kernel.

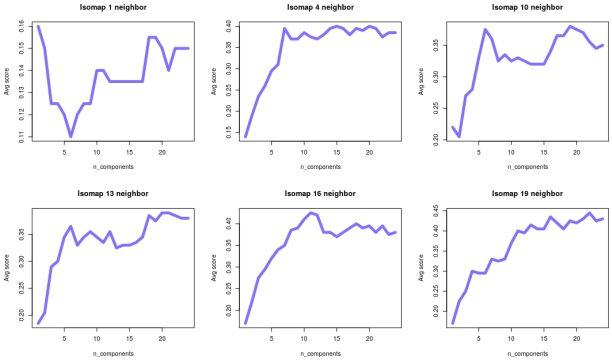

Figure 26: Isomap performance.

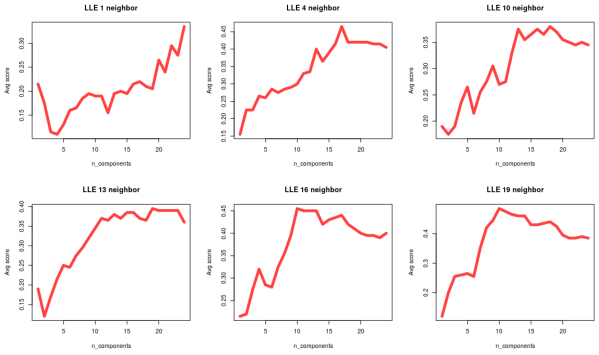

Figure 27: LLE performance.

Considering table 8 we can see that the effect of manifold learning seems gives a slight enhancing on the class separability. It is also possible to observe that the major improvement is given by LLE, instead Isomap starts to give better results when we increase a lot the number of dimensions and neighbors. Figure 26 and Figure 27 present the same pattern that we have previously analyzed, that is the manifold learning technique works well with a discrete number of degree of freedom and a tolerant number of neighbors.

6 Conclusion

In this document, we have faced the problem of converting directly graph structures into exact vectorial form with the purpose of using the classical machine learning techniques over them. We have seen that there is not a precise definition useful to convert a graph into a vector, but it is possible to stem the problem taking advantage on the kernel trick strategy, which basically consists on computing the similarity between the given graphs in a proper space. We have seen graph kernels that can be applied and each of them considers different types of information (labels, topology, etc.). In particular we have analyzed the theoretical aspects, properties and complexities of two well known graph kernels, Shortest-Path and Weisfeiler-Lehman kernel.
Another aspect of this assignment was to give also some theoretical knowledges and properties about manifold learning algorithms, commonly useful for dimensionality reduction. Two important manifold learning algorithms, Isomap and Local Linear Embedding, were analyzed. The main goal was to evaluate if the two algorithms could enhance the classes separability projecting the original manifold in a low-dimensional embedding. Then, in order to give a correct and unbiased analysis, the followed strategy have proposed different linear SVM classifiers, based or not on the application of a manifold techniques, learned over two datasets of graphs (PPI and Shock). Analyzing the obtained results with the two datasets, we have seen that the application of both the manifold learning algorithms brought to a slight improvement of the obtained scores, meaning that they have enhanced the class separability a little bit. Of course another important aspect is that with any manifold learning technique we are embedding the original data-space into a new one which is composed by less components, meaning that it is possible to reduce the computational cost time and space required to analyze and store data.
Manifold learning techniques so can be useful to reduce data with a non-linear dimensionality reduction strategies and this also improving the accuracy scores, obtained from a general classifier. It is important to notice that the proposed manifold learning techniques work considering two fundamental hyper-parameters, number of neighbors and number of components. During the analysis step we have also seen that the choice of the two in fundamental, since a small number of components could not give enough degrees of freedom and the number of neighbors decide how many local information should be considered.
More in general, we have observed that in most of the cases the separation of the data could be increased after applying manifold learning on the original kernels by picking the right hyper-parameters.