Artificial Intelligence - Knowledge Representation and Planning - Assignment 3

Lorenzo Soligo - 875566

Academic Year 2018-2019

1 Requirements

Read this article presenting a way to improve the disciminative power of graph kernels. Choose one graph kernel among

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

Choose one manifold learning technique among

- Isomap
- Diffusion Maps
- Laplacian Eigenmaps
- Local Linear Embedding

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

- PPI: this is a Protein-Protein Interaction dataset. Here proteins (nodes) are connected by an edge in the graph if they have a physical or functional association.
- Shock: representing 2D shapes. Each graph is a skeletal-based representation of the differential structure of the boundary of a 2D shape.

Note: the datasets are contained in Matlab files. The variable `G` contains a vector of cells, one per graph. The entry `am` of each cell is the adjacency matrix of the graph. The variable `labels`, contains the class-labels of each graph.

2 Background

2.1 Kernel functions

A positive-definite kernel is a generalization of a positive-definite function or a positive-definite matrix.

Let $\mathcal{X}$ be a nonempty set. A *symmetric* function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a positive-definite kernel on $\mathcal{X}$ if

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

Notice that positive definite kernels require $c_i = 0 \ \forall i$, while positive semi-definite kernels do not impose this condition. This relates to the spectrum of a finite matrix constructed by pairwise evaluation $\mathbf{K}_{ij} = K(x_i, x_j)$: in the former case we have only positive eigenvalues; in the latter we have non-negative eigenvalues.

2.1.1 The kernel trick

Kernel methods (namely SVMs and many more) exploit kernel functions to work on high-dimensional, implicit feature spaces without having to compute the coordinates of the data in that space. This is achieved by performing inner products between the images of all pairs of data in the feature space. This operation is called *kernel trick*. It is extremely useful in the case the dataset is not linearly separable, but can be easily separated by an hyperplane in a higher-dimensional space.

Formally, a kernel maps two objects x and x' via a mapping ϕ into the feature space $\mathcal{H}$, measuring their similarity in $\mathcal{H}$ as $\langle \phi(x), \phi(x') \rangle$. The kernel trick is nothing but computing the inner product in $\mathcal{H}$ as kernel in the input space: $k(x, x') = \langle \phi(x), \phi(x') \rangle$.

2.2 Graph comparison problem

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 (or dissimilarity) of G and G'.

2.3 Graph isomorphism

Given two graphs G_1 and G_2, find a mapping f of the vertices of G_1 to the vertices of G_2 such that G_1 and G_2 are identical, i.e. (x, y) is an edge of G_1 iff $(f(x), f(y))$ is an edge of G_2. Then f is an isomorphism, and G_1 and G_2 are said to be isomorphic.

At the moment 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.

2.3.1 Graph edit distances

The idea is to count the number of operations that is necessary to transform G_1 into G_2, assigning different costs to the different types of operations (e.g. edge/node insertion/deletion, modification of labels, . . .). This allows us to capture (partial) similarities between graphs, but contains a check for subgraph isomorphism (which is NP-complete) as an intermediate step.

2.3.2 Topological descriptors

The idea here is to map each graph to a feature vector and then using distances and metrics on vectors for learning on graphs. In this case the clear advantage is that known, efficient tools for feature vectors can be reused, but the feature vector transformation either leads to a loss of topological information or still includes subgraph isomorphism as one step.

3 Graph kernels

3.1 Introduction

From the background, we have understood that computing whether two graphs are isomorphic is usually expensive, often becoming infeasible for “big” graphs. Therefore it would be great to have a polynomial time similarity measure for graphs. Graph kernels allow us to compare substructures of graphs that are computable in polynomial time. We want a graph kernel to be expressive, efficient to compute, positive definite and applicable to a wide range of graphs.

3.2 Representation of graphs

Graphs are usually represented using adjacency lists/matrices. However, standard pattern recognition techniques require data to be represented in vectorial form. This is quite a tough operation for graphs. First of all, the nodes in a graph are not ordered, therefore a reference structure must be established as a prerequisite. Second, even though the vectors could be encoded as vectors, their length would be variable and they would therefore belong to different spaces.

3.2.1 The kernel trick, again

The kernel trick has the advantage of shifting the problem from a vectorial representation -now implicit- to a similarity representation, allowing standard learning techniques to be applied to data for which a vectorial representation is hard to achieve.

3.3 Definition and problems

3.3.1 What is a 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 SVMs to work directly on graphs, without having to do feature extraction to transform them to fixed-length, real-valued feature vectors.

Computing graph similarities is fundamental in many research areas: for example, a common assumption when working with previously unseen molecules is that molecules with similar structures will have similar functional properties. This is a typical case in which measuring the similarity between graphs is a fundamental aspect of the research work.

To better explain graph kernels, let us introduce R-convolution kernels, a family graph kernels are instances of. These kernels compare decompositions of two discrete, structured, compound objects. Most R-convolution kernels simply count the number of isomorphic substructures in the two compared graphs and differ mainly by the type of substructures used in the deconvolution and the algorithms used to count them efficiently.

$$k_{convolution}(x, x') = \sum_{(x_d, x) \in R} \sum_{(x'_d, x') \in R} k_{parts}(x_d, x'_d)$$

Graph kernels are nothing but convolution kernels on pairs of graphs. A new decomposition relation R results in a new graph kernel. A graph kernel makes the whole family of kernel methods applicable to graphs.

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

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

3.3.2 Link to graph isomorphism (hardness result)

One of the main problems with the aforementioned approach is that given the high degree of information that graphs express, the task of defining complete kernels (i.e. ϕ is injective) is proved to be as hard as solving the graph isomorphism problem.

In particular, 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.

In fact, since ϕ is injective, we have

$$\begin{aligned}
&\sqrt{k(G,G) - 2k(G,G') + k(G',G')} \\
&= \sqrt{\langle \phi(G) - \phi(G'), \phi(G) - \phi(G') \rangle} \\
&= ||\phi(G) - \phi(G')|| = 0
\end{aligned}$$

iff G is isomorphic to G'.

3.3.3 Complexity and horseshoe effect

As we have said, computing kernels for injective mappings is as hard as deciding graph isomorphism. Instead, what we are looking for is a polynomial-time, non-injective kernel which combines expressivity with efficiency.

Many graph kernels are very effective in generating implicit embeddings, but there is no guarantee that the data in the Hilbert space will show a better class separation. This happens because of the complexity of the structural embedding problem and the limits for efficient kernel computation. For example, data tends to cluster tightly along a curve that wraps around the embedding space due to the consistent underestimation of the geodesic distances on the manifold, placing data onto a highly non-linear manifold in the Hilbert space. As a matter of fact, this *horseshoe* is the intersection between the manifold and the plane used to visualise the data. It might be caused by kernel normalisation, that projects data points from the Hilbert space to the unit sphere possibly creating an artificial curvature of the space that either generates or exaggerates the horseshoe effect.

3.3.4 Locality

Generally the non-linearity of the mapping is used to improve local class separability, while a large curvature might fold the manifold reducing long range separability. The impact of the locality of distance information on the performance of the kernel thus becomes a key point to be studied: we will use some manifold learning techniques to embed the graphs onto a low-dimensional vectorial space, trying to unfold the embedding manifold and increase class separation.

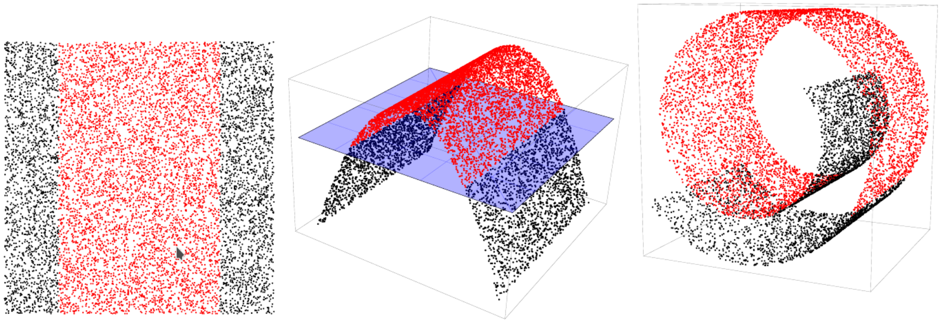

Figure 1: Example of reduced linear separability due to high curvature of the embedding. Introducing a non-linear mapping to a low-curvature manifold makes the data linearly separable. Mapping to high global curvature manifold results in low linear separability of the data. The higher the curvature the less separable the data is.

Also, many kernels proposed in the literature neglet locational information for the substructures in a graph, and cannot therefore establish reliable structural correspondences between the substructures in a pair of graphs, lowering the precision of the similarity measure.

In this assignment, I will test the Wiesfeiler-Lehman kernel.

3.4 Weisfeiler-Lehman Kernel

3.4.1 Description

The Weisfeiler-Lehman kernel is a state-of-the-art graph kernel that enumerates the common subtrees between two graphs by using the Weisfeiler-Lehman test of graph isomorphism.

Given two attributed graphs $G_1 = (V_1, E_1, l_1)$ and $G_2 = (V_2, E_2, l_2)$ where the l_i's denote the set of labels of V_i, the idea is that of associating with each vertex v a multiset label based on the labels of the neighbors of v. This is iterated a fixed number of times and at each step the resulting multisets are sorted and compressed to generate new vertex labels.

Let G_1^i and G_2^i denote the graphs G_1 and G_2 after i iterations of vertices relabeling procedure. The Weisfeiler-Lehman is then defined as

$$k_{wl}^h(G_1, G_2) = \sum_{i=0}^{h} k_\delta(G_1^i, G_2^i)$$

where $k_\delta(G_1^i, G_2^i)$ is a positive definite kernel which enumerates the pairs of vertices in G_1^i and G_2^i that share the same label. The computation of the Weisfeiler-Lehman with h iterations has complexity $\mathcal{O}(hm)$, where m is the number of edges of the graph.

3.4.2 Algorithm

Perform the following three steps h times:

1. sorting: represent each node v as a sorted list L_v of its neighbors ($\mathcal{O}(m)$)
2. compression: compress this list into a hash value $h(L_v)$ ($\mathcal{O}(m)$)
3. relabeling: relabel v with $h(L_v)$ as its new node label ($\mathcal{O}(n)$)

Complexity:

- $\mathcal{O}(m \cdot h)$ per pair of graphs
- $\mathcal{O}(N \cdot m \cdot h + N \cdot n \cdot h)$ for N graphs.

4 Manifold Learning

4.1 Introduction and motivations

Manifold learning is an approach to non-linear dimensionality reduction. Algorithms for this task are based on the idea that the dimensionality of many data sets is only artificially high and the data actually resides in a low-dimensional manifold embedded in the high-dimensional feature space. Also, the manifold may fold or wrap in the feature space so much that the natural feature-space parametrization does not capture the underlying structure of the problem. Manifold learning algorithms attempt to uncover a non-linear parametrization for the data manifold in order to find a low-dimensional representation of the data that effectively unfolds the manifold and reveals the underlying data structure.

High-dimensional datasets can be very difficult to visualize. In order to visualize the structure of a dataset, the dimension must be reduced in some way. The simplest way to accomplish this dimensionality reduction is by taking a random projection of the data. Though this allows some degree of visualization of the data structure, in the random projection it is likely that the more interesting structure within the data will be lost.

To address this concern, a number of supervised and unsupervised linear dimensionality reduction frameworks have been designed, such as Principal Component Analysis (PCA) and many others. These algorithms define specific rubrics to choose an “interesting” linear projection of the data. These methods can be powerful, but often miss important non-linear structure in the data.

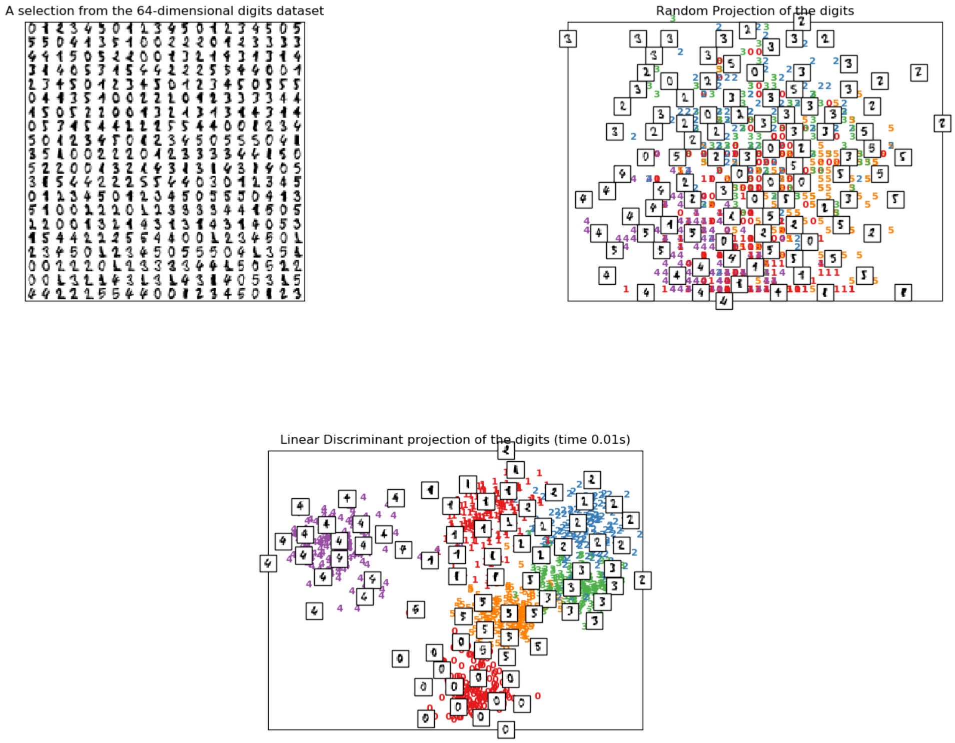

Figure 2: The representation drastically improves using dimensionality reduction techniques

4.2 Defining manifold learning

Manifold Learning can be thought of as an attempt to generalize linear frameworks like PCA to be sensitive to non-linear structure in data. Though supervised variants exist, the typical manifold learning problem is unsupervised: it learns the high-dimensional structure of the data from the data itself, without the use of predetermined classifications. Intuitively, the “curvier” is the considered manifold, the denser the data must be.

Now we will define the two manifold learning algorithms used in this assignment: we will see a global approach (Isomap) and a local one (LLE).

4.3 Isomap

4.3.1 Description

Isomap (short for **iso**metric feature **map**ping) seeks a low-dimensional representation of the data which maintains geodesic (namely, the shortest path between two points on a surface/manifold) distances between all points. In this sense, it is a direct generalization of Multidimensional Scaling (MDS). Isomap assumes that only the pairwise distances between neighboring points are known. The geodesic distances are approximated as the length of the minimal path on a neighborhood graph, i.e. each distance is estimated as the shortest path distance between the corresponding nodes in the graph.

In Isomap, the long-range distances become more important than the local structure, and this makes it quite sensitive to noise: depending on the topology of the neighborhood graph, Isomap suffers shortcutting and other distortions.

Isomap mainly relies on a k-nearest neighbor search, done for each point in the dataset, that leads to the construction of a k-neighbors graph, in which each point is connected with its k nearest neighbors.

4.4 Locally Linear Embedding (LLE)

Locally linear embedding (LLE) seeks a lower-dimensional projection of the data which preserves distances within local neighborhoods. The underlying assumption is that a point and its neighbors in the original space should line on (or close to) a locally linear patch of the manifold; this also makes t possible to reconstruct each point as a linear combination of its neighbors. In particular, the optimal coefficients w_{ij} are found by minimising the reconstruction error

$$\mathcal{E}(W) = \sum_{i=1}^{n} \big(\mathbf{x}_i - \sum_{j \in \mathcal{N}(i)} w_{ij} \mathbf{x}_j\big)$$

where $\mathcal{N}(i)$ denotes the neighborhood of $\mathbf{x}_i$.

LLE can be thought of as a series of local Principal Component Analyses which are globally compared to find the best non-linear embedding. Here the manifold is seen as a collection of overlapping coordinate patches: if the neighborhoods are small enough and the manifold is smooth enough, the local geometry of the patches can be considered approximately linear.

Since LLE focuses on preserving distances locally, LLE can distort the global structure of the data. The idea, in fact, is to characterize the local geometry of each neighborhood as a linear function and to find a mapping to a lower dimensional Euclidean space that preserves the linear relationship between a point and its neighbors.

4.4.1 Modified Locally Linear Embeddng

One well-known issue with LLE is the regularization problem. When the number of neighbors is greater than the number of input dimensions, the matrix defining each local neighborhood is rank-deficient. Standard LLE addresses this problem by applying an arbitrary regularization parameter r, which is chosen relative to the trace of the local weight matrix. It can be proved that for $r \to 0$ the solution converges to the desired embedding, but there is no guarantee that the optimal solution will be found for $r > 0$. This results in a distortion of the underlying geometry of the manifold.

One method to address the regularization problem is to use multiple weight vectors in each neighborhood. This is the essence of modified locally linear embedding (MLLE).

The steps taken are the same as standard LLE, but the weight matrix construction takes more time because we need to construct the weight matrix from multiple weights. In practice, however, this increase in the cost is negligible.

5 Graph Kernels and Manifold Learning

5.1 Challenges

As previously said, applying multidimensional scaling to the distances in the implicit Hilbert space obtained from R-convolution graph kernels often results in the horseshoe effect: data is distributed tightly on a highly curved line or manifold. This comes from a consistent underestimation of the long-range distances consistent with the properties of these kernels.

R-convolution kernels typically count the number of isomorphic substructures in the decomposition of the two graphs, not considering locational information for the substructures in a graph. The similarity of the substructures are not related to the relative position in the graphs.

When graphs are very dissimilar, many similar small substructures can appear simply because of the statistics of random graphs, and the smaller and simpler the substructures are in the decomposition, the more likely it is to find them in many locations of the two structures. In other words, the smaller is the considered sample, the higher is the probability of finding similarities because of random fluctuations. Notice that what decreases as the size increases is the proportion of correct matches with respect to the total possible correspondences of the same size.

The lack of a locality condition and the consequent summation over the entire structure amplifies the effects of these random similarities, resulting in a lower bound on the kernel value that is a function only of the random graph statistics. This leads to a consistent reduction in the estimated distances for dissimilar graphs, adding a strong curvature to the embedding manifold -which can fold on itself- and increasing the effective dimensionality of the embedding.

5.2 Improving graph kernels with manifold learning

This assignment requires us to compare an SVM trained on a kernel with and without the manifold learning step. The goal is to try and see whether applying an optimal manifold learning process to the distance matrix leads to an increase in the class separation.

Given a set of n graphs $\mathcal{G} = \{G_1, ..., G_n\}$ and their kernel matrix $K = (k_{ij})$ we can compute the distance matrix $D = (d_{ij})$ with $d_{ij} = \sqrt{k_{ii} + k_{jj} - 2k_{ij}}$. Then we can apply the selected manifold learning algorithm and train an SVM classifier (C-SVM) with a linear kernel. Finally we can select the optimal set of parameters using cross validation for the ν-tuple of parameters which maximises

$$\text{argmax}_{p_1,...,p_{\nu-1}} \max_C \alpha$$

where α is the 10-fold cross validation accuracy of the C-SVM, C is the regularizer constant, and $p_1, ..., p_{\nu-1}$ are the parameters of the chosen manifold learning technique

6 Experiments

6.1 Code

The code has been implemented in Python, mainly relying on the Numpy and Scikit-Learn libraries.

Here is a high-level description of what it does, taken from the original paper:

1. **Multiset label determination**
    - assign a multiset label $M_i(v)$ to each node $v \in G$ which consists of the multiset $\{l_{i-1}(u) \mid \text{u is a neighbor of v}\}$
        - done in \texttt{determine_labels}
        - as per the paper, since our graphs are unlabelled, we use the node-degrees as starting labels for the node
2. **Sorting each multiset**
    - Sort elements in $M_i(v)$ in ascending order and concatenate them into a string $s_i(v)$
        - sorted and merged in \texttt{get_labels}
    - Add $l_{i-1}(v)$ as a prefix to $s_i(v)$
        - done in \texttt{extend_labels}. Returns the string formatted as requested
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 $s_i(v) = s_i(w)$
        - done in \texttt{compress_label} and `relabel`
    - As the first "hash", I use the highest degree of a node in all graphs, plus one (hence I'm sure that one is a hash instead of an original label
4. **Relabeling**
    - Set $l_i(v) = f(s_i(v))$ for all nodes in G
        - done in `relabel`

After having done all of the above, the similarity matrix for the N graphs is computed. The `run()` method returns the similarity matrix containing the normalized values for all the graphs.

6.2 Results

The reported results are obtained from a 10-fold Cross Validation with shuffled dataset. For the manifold learning algorithms, I set the number of neighbors to 15 and the number of components to 2.

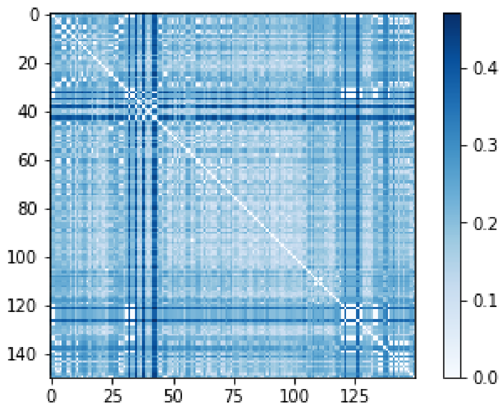

Figure 3: Pairwise distances for the SHOCK dataset

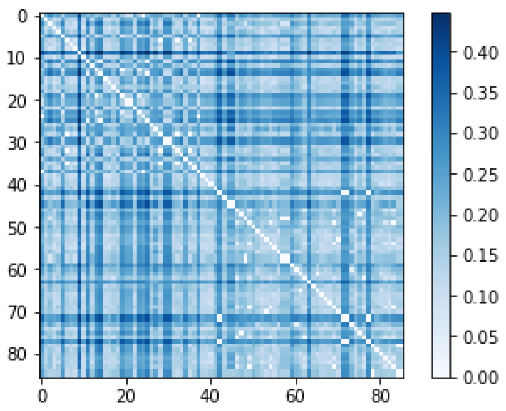

Figure 4: Pairwise distances for the PPI dataset

6.2.1 SVMs without manifold learning step

6.2.1.1 SHOCK dataset

Value	Accuracy
Minimum	0.1
Mean	0.34
Max	0.6
Standard deviation	0.14

6.2.1.2 PPI dataset

Value	Accuracy
Minimum	0.555
Mean	0.71
Max	0.78
Standard deviation	0.07

6.2.2 SVMs with Isomap manifold learning

6.2.2.1 SHOCK dataset

Value	Accuracy
Minimum	0.1
Mean	0.275
Max	0.4
Standard deviation	0.10

6.2.2.2 PPI dataset

Value	Accuracy
Minimum	0.5
Mean	0.70
Max	0.88
Standard deviation	0.13

6.2.3 SVMs with LLE manifold learning

6.2.3.1 SHOCK dataset

Value	Accuracy
Minimum	0.1
Mean	0.245
Max	0.4
Standard deviation	0.10

6.2.3.2 PPI dataset

Value	Accuracy
Minimum	0.5
Mean	0.61
Max	0.66
Standard deviation	0.07

7 Conclusions

8 Resources

- http://www.dsi.unive.it/~atorsell/AI/graph/Unfolding.pdf
- http://www.dsi.unive.it/~atorsell/AI/graph/kernels.pdf
- https://en.wikipedia.org/wiki/Kernel_method
- https://en.wikipedia.org/wiki/Positive-definite_kernel
- https://www.ethz.ch/content/dam/ethz/special-interest/bsse/borgwardt-lab/documents/slides/
- https://ethz.ch/content/dam/ethz/special-interest/bsse/borgwardt-lab/documents/slides/CA10_WeisfeilerLehman.pdf
- https://scikit-learn.org/stable/modules/manifold.html
- https://scikit-learn.org/stable/auto_examples/manifold/plot_lle_digits.html
- https://en.wikipedia.org/wiki/Graph_kernel