Universitá
Ca'Foscari
Venezia

Master course

in Computer Science

Artificial Intelligence: Knowledge Representation and Planning

-

Assignment 3

Studente
Francesco Bruno
Matricola 875812

Anno Accademico
2021 / 2022

Contents

Chapter 1

Introduction

1.1 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
- Shock

The zip files contain csv files representing the adjacecy matrices of the graphs and of the lavels. the files graphxxx.csv contain the adjaccency matrices, one per file, while the file labels.csv contains all the labels

1.2 Graph approach

1.2.1 Graph

Until now *feature vectors* were used in order to represent an object with its attributes, another way to deal with this problem is to represent objects as graphs.
A **graph** $G(V, E)$ is a tuple composed of two sets V and E where the first set contains all graph *vertices* $V = \{v_1, v_2, ..., v_n\}$ and $n \in N$, E instead contains all the edges $E \subset V \times V$ of the graph.
Graphs can be *weighted* or *unweighted* depending on if edges have or not have weights on them, also graphs can be *directed* or *undirected* depending on if edges have a specific direction from a vertex to the other or if there is a symmetry where both vertices can be the starting point to reach the other one, thus in order to solve the task undirected graph are used.

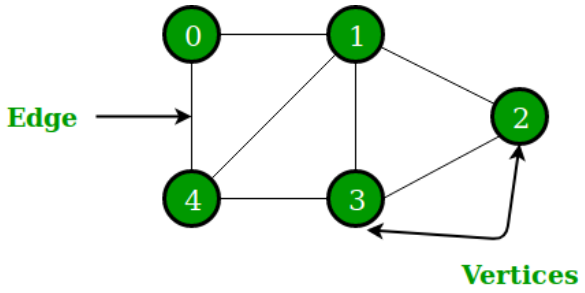

Figure 1.1: Example of undirected and unweighted graph taken from `https://media.geeksforgeeks.org/wp-content/cdn-uploads/undirectedgraph.png`

A Graph can be represented by using an **adjacency matrix** M with dimension $|V| \times |V|$ where $M_{ij} = 1$ if graph is unweighted and exists an edge between node i and j, 0 if it doesn't exist, $M_{ij} = w_{ij}$ if graph is weighted and obviously if exists and edge between node i and j, 0 otherwise.

Table 1.1: Adjacency matrix of the unweighted graph above

0	1	0	0	1
1	0	1	1	1
0	1	0	1	0
0	1	1	0	1
1	1	0	1	0

1.2.2 Graph similarity & isomorpishm

Having defined what graphs are, how can we compare two existing graphs? Given G_1 and G_2 two graphs, the *similarity* between them can be expressed as a function:

$$s : G \times G \rightarrow R$$

where $s(G_1, G_2)$ returns the similarity between G_1 and G_2.
This measure can be found found thanks to graph **isomorphism**, this is a problem since it's an NP-Hard problem with a exponential time computation.
Graph comparison is a very important problem since in many fields is assumed that similar objects have a relation so when they are transformed to graphs this comparison can't be ignored and should be processed as fast as possible, thus in polynomial time.

Chapter 2

Graph Kernel

2.1 Definition

Since computing similarity through graph isomorphism requires exponential time, a new solution is found. **Graph kernels** are *kernel functions* used to compute similarity between two graphs in polynomial-time through extrapolating graph patterns and comparing them.
So, graph kernels allows SVM algorithms to work directly on graphs without transforming them into feature vectors, fortunatelly, since it could be a difficult problem because graph nodes aren't ordered and also there isn't an unique structure for graphs and vectors wouldn't have a fixed length.
There are different graph kernels, the ones proposed by the professor are:

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

Each one has its own characteristics on taking graph patterns and comparing them so only the first one is now analyzed as the requirements ask.

2.2 Shortest-Path Kernel

The kernel analyzed is the **Shortest-Path** one, its simple idea is to compute from $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ the respective shortest paths graphs $S_1 = (V_{s1}, E_{s1})$ and $S_2 = (V_{s2}, E_{s2})$, then these are compared according to a similarity measure defined as:

$$K_{shortestPath}(S_1, S_2) = \sum_{e_1 \in E_{s1}} \sum_{e_2 \in E_{s2}} k(e_1, e_2)$$

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.
An example of similarity could be the cardinality of nodes inside the shortest path or the number of common node labels for each path.
Shortest paths only are used to determine the similarity between graphs since determining all paths or the longest ones are NP-Hard problems and this problem is instead solvable in polynomial time.

2.3 Floyd-Warshall Algorithm

In order to compute the shortest path of a graph, a great algorithm able to do it in polynomial time must be defined.
One of the most famous algorithm built that is able to find the shortest path between the nodes of a graph is the **Floyd-Warshall** algorithm.

```
1   FloydWarshall(G):
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|
8       for i from 1 to |V|
9         for j from 1 to |V|
10          if dist[i][j] > dist[i][k] + dist[k][j]
11            dist[i][j] ← dist[i][k] + dist[k][j]
12          end if
13
```

Code 2.1: Floyd-Warshall algorithm taken from `https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm`

This algorithm is good since it can compute shortest paths beetween all pairs of vertices in a graph with complexity $O(|V|^3)$ when graph is both dense or sparse. An alternative algorithm to this one is the so called *Dijkstra* one, another famous algorithm with complexity:

	Array	Heap
Sparse	$O(\|V\|^2)$	$O(\|V\|log(\|V\|))$
Dense	$O(\|V\|^2)$	$O(\|V\|^2log(\|V\|))$

In order to find all shortest paths between all pairs of vertices in a graph using Dijkstra, the algorithm must be iterated over each possible vertex, increasing the already existing complexity of the table above by multypling it with the number of nodes $|V|$.
As we can see Floyd-Warshall one is more versatile since it doesn't require the knowledge of the type of the graph, so this algorithm has been choosen.

2.4 Similarity measure between paths

There are many ways that can be taken in order to measure the similarity between two shortest path graphs, so to implement the similarity function previously defined then **delta kernel** is introduced.

2.4.1 Delta kernel

Key idea of delta kernel is to measure the similarity between two shortest path graphs by considering the frequency of the length of their paths. So, given δ constant value, we create for each graph a vector of dimension δ that contains the shortest path occurences when their weights are lower than δ. In our case, δ will be equal to the maximum weight of an existing path situated inside one of the two shortest path graphs, so:

$$\delta = \max\{\max\{shortestPathGraph_1\}, \max\{shortestPathGraph_2\}\}$$

So, given a vector V of dimension δ associated to a shortest path graph SP, V_i contains the occurances of paths which weights are equal to i inside SP. After these vectors are computed, they are normalized and is made a dot product between them so to have a similarity measure with value between 0 and 1.
An example, given two shortest path matrices:

Table 2.1: Shortest path matrices of two different graphs

0	1	1	0	1	2
1	0	1	1	0	1
1	1	0	2	1	0

$\delta = \max\{\max\{shortestPath_1\}, \max\{shortestPath_2\}\} = 2$
thus, the relatives frequency vectors are:

Table 2.2: Frequency vectors

3	6	0	3	4	2

The respective normalized vectors are:

Table 2.3: Normalized Frequency vectors

$\frac{1}{\sqrt{5}}$	$\frac{2}{\sqrt{5}}$	0	$\frac{3}{\sqrt{29}}$	$\frac{4}{\sqrt{29}}$	$\frac{2}{\sqrt{29}}$

Dot product is then computed and result is equal to 0.9135.

Chapter 3

Manifold Learning

Manifold learning is a model used to reduce dimensionality of data since some problems can occur if large set of features are considered, for example k-Nearest Neighbours algorithms work well with low-dimensional data and in contrast work poorly on high dimensional data since data becomes more sparse, this problem is called *curse of dimensionality*.
Key idea of manifold learning algorithms is that data points may consist of many features but they can also be described as a function of few underlying parameters, so these algorithms try to get a good low dimensional representation of high dimensional data.
Some manifold learning algorithms are:

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

Only two of these algorithms are choosen, so let's analyze them.

3.1 Isomap

Isomap is one efficient algorithm, generalization of **MultiDimensional Scaling** (MDS), that finds a lower dimensional representation of data which maintains *geodesic*[1] distances between all points.
Geodesic distances can be approximated by Euclidean distance if two points are close (they can be found using kNN or *Ball Tree* algorithms), otherwise it can be estimated as equal to their shortest path length (it can be found using algorithms like Floyd-Warshall or Dijkstra).

[1] Shortest distance between two point on a manifold

Despite this, wrong neighbors distances can cause bad general results, especially on long range distances or when manifold contains holes, thus Isomap is quite sensitive to noise.

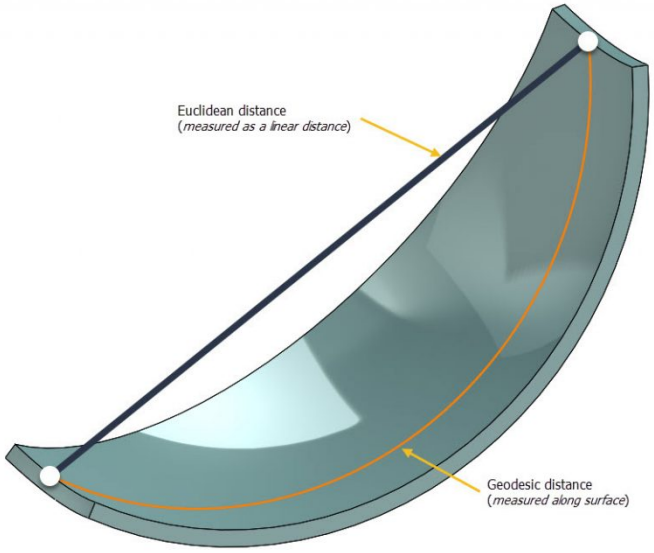

Figure 3.1: Differences from Geodesic and Euclidean distance

3.2 Local Linear Embedding

Local Linear Embedding is a manifold learning algorithm which reduces an N-dimensional feature space into a smaller $n - dimensional$ one where $n << N$.
The key idea of this technique is that the algorithm defines data points in terms of their local linear relationship, so as a linear combination of its neighbors, so thanks to neighbors information we can map a point to another point with lower dimensionality and at the same time this new representation preserves the local linear relationship between neighbors.
A simple way to formulate this algorithm is to find k nearest neighbors for each data point (for example using Euclidean distance), after having found the neighbors of our points we define the data points by expressing their relationship with the neighbors points, thus the following cost function is computed:

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

where W_{ij} is the contribution of each data point to the reconstruction and has the constraint:

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

The function illustrated above represents the total reconstruction error of the point we are interested to reconstruct, so the lower is its value the better is the approximation of the k points, thus the aim of this part is to minimize the value of that cost function over the points by finding the optimal weights. Last step, after having computed each point $x_i \in R^N$ as a relationship with its neighbors, is to map each x_i to a lower dimensional representation, thus to find its representation $y_i \in R^n$.
This is done by considering another time a new cost function, similar to the previous one:

$$\theta(\mathrm{Y}) = \sum_i |y_i - \sum_j W_{ij} y_j|^2$$

We have to choose n coordinates y_i that minimize the above cost function having already fixed the weights W_{ij}.

Chapter 4

Comparison

Now are presented some results obtained by using both PPI and SHOCK datasets on short path kernel with and without manifold learning algorithms applied.
For each test is performed a 10-way cross validation and dataset is also shuffled, then in order to sum up the information gained, only best and worst performances are shown when manifold learning algorithms are used.

4.1 PPI dataset

Without Manifold Learning

Kernel	Min Score	Max Score	Avg Score
Shortest-Path Kernel	0.5	0.8888	0.7638

With Manifold Learning - Local Linear Embedding

Tests made have been done with the number of neighbors $\in \{2, 3, ..., 30\}$ and the number of components $\in \{2, 3, ..., 20\}$.
Best performances:

Shortest-Path Kernel				
N.Neighbors	**N.Components**	**Min Score**	**Max Score**	**Avg Score**
6	20	0.625	1.0	0.8027

Worst performances:

Shortest-Path Kernel				
N.Neighbors	**N.Components**	**Min Score**	**Max Score**	**Avg Score**
21	5	0.375	0.5555	0.5097

With Manifold Learning - Isomap

Tests made have been done with the number of neighbors $\in \{2, 3, ..., 30\}$ and the number of components $\in \{2, 3, ..., 20\}$.
Best performances:

Shortest-Path Kernel				
N.Neighbors	**N.Components**	**Min Score**	**Max Score**	**Avg Score**
30	18	0.5555	1.0	0.8152

Worst performances:

Shortest-Path Kernel				
N.Neighbors	**N.Components**	**Min Score**	**Max Score**	**Avg Score**
10	3	0.125	0.5555	0.3958

4.2 Shock dataset

Without Manifold Learning

Kernel	Min Score	Max Score	Avg Score
Shortest-Path Kernel	0.1333	0.6666	0.4466

With Manifold Learning - Local Linear Embedding

Tests made have been done with the number of neighbors $\in \{2, 3, ..., 30\}$ and the number of components $\in \{2, 3, ..., 20\}$.
Best performances:

Shortest-Path Kernel				
N.Neighbors	**N.Components**	**Min Score**	**Max Score**	**Avg Score**
29	19	0.1333	0.4666	0.3

Worst performances:

Shortest-Path Kernel				
N.Neighbors	**N.Components**	**Min Score**	**Max Score**	**Avg Score**
5	2	0	0.0666	0.06

With Manifold Learning - Isomap

Tests made have been done with the number of neighbors $\in \{2, 3, ..., 30\}$ and the number of components $\in \{2, 3, ..., 20\}$.
Best performances:

Shortest-Path Kernel				
N.Neighbors	**N.Components**	**Min Score**	**Max Score**	**Avg Score**
21	17	0.3333	0.7333	0.5

Worst performances:

Shortest-Path Kernel				
N.Neighbors	**N.Components**	**Min Score**	**Max Score**	**Avg Score**
7	3	0.1333	0.4	0.2533

Chapter 5

Conclusion

After having seen the meaning of graph and graph kernels, thus the existance of other methods to represent data other than feature vectors and kernels that accept this new representation of data as input, the document has analyzed one specific graph kernel, the Shortest-Path one, showing its properties and performances. Then a theoretical introduction about dimensionality reduction and why in some cases is important is done, thus manifold learning topic is introduced and two manifold learning algorithms called Local Linear Embedding and Isomap have been choosen from a set of 4 choices. After having analyzed them showing their properties and how they work, these algorithms have been applied with the previous kernel and results are derived again.
After results have been gained, a conclusion has been made, we can notice how both Local Linear Embedding and Isomap have slightly improved average accuracy over PPI dataset if parameters are properly set, in our case when the number of neighbors is 6 and the number of components is 20 for the first algorithm and 30 neighbors and 18 components for the second one, meaning that more in general it can be possible to reduce the dimensionality of data and at the same time is still possible to correctly distinguish classes and also to have a more readable data when manifold learning is used.
This seems good but other results obtained have shown how the tuning of parameters is important in order to have a slightly improvement, otherwise performances are equal or worse than before. Considering PPI dataset the worst performances obtained have values about $\frac{1}{4}$ and $\frac{1}{2}$ respectively lower than the accuracy gained without manifold learning algorithm applied, instead Shock dataset has experienced a poor increase on the average accuracy when Isomap is applied, on the other hand when Local Linear Embedding is used the performances are really worse than before, so choosing the right parameters matter in order to have a correct distinction between classes at lower dimensions.
In conclusion manifold learning is a powerful technique used to improve the

classification accuracy with a lower dimensionality dataset, but on the other hand in order to obtain good results some parameters must be found, such as the correct number of neighbors and components, otherwise accuracy can be equal or lower than what we have experienced without manifold learning, in particular in the PPI dataset and LLE algorithm best results have been obtained with a low number of neighbors, best case has 6 neighbors, with Isomap on the contrary the number of neighbors for the best case is higher, on our case is equal to 30. Shock dataset instead has experienced better results with higher number of neighbors for both LLE and Isomap, 29 and 21 respectively, so parameters can depend on the dataset used and find the right values can be expensive.

List of Figures