On Geodesic Distance Computation: An Experimental Study

David Bautista-Villavicencio and Raúl Cruz-Barbosa

Universidad Tecnológica de la Mixteca, 69000, Huajuapan, Oaxaca, México
{dbautista,rcruz}@mixteco.utm.mx
(Paper received on November 28, 2010, accepted on January 28, 2011)

Abstract. The most common distance function used in the machine learning field is the Euclidean distance. It is due to its easily intuitive understanding and interpretation in the real world. However, it has been verified that for datasets representation presenting convoluted geometric properties (many foldings), the Euclidean distance is not the proper choice to measure the (dis)similarity between two elements of the data manifold. An alternative distance function that alleviates, in part, the previously mentioned problem is the geodesic distance, since it measures similarity along the embedded manifold, instead of doing it through the embedding space. Some problems of computing geodesic distances are related to computational time and storage restrictions about the graph representation and the shortest path algorithm to be used. Thus, the main objective of this paper is to show some characteristics about computational time and storage performance for computing the geodesic distance by using the basic Dijkstra algorithm and a full data matrix representation against some alternatives in order that researchers can select the suitable one depending on the available computational resources.

1 Introduction

Automation of the human being learning process is a complex task. When dividing the existing reality into different categories, we are seamlessly performing a classification task that can be improved over time through learning.

In the machine learning field, the task of unraveling the relationship between the observed data and their corresponding class labels can be seen as the modeling of the mapping between a set of data inputs and a set of discrete data targets. This is understood as supervised learning.

Unfortunately, in many real applications class labels are either completely or partially unavailable. The first case scenario is that of unsupervised learning, where the most common task to be performed is that of data clustering. The second case, semi-supervised learning, is less frequently considered but far more common than what one might expect: quite often, only a reduced number of class labels is readily available and even that can be difficult and/or expensive to obtain.

The distance functions have an important role in the machine learning field, mainly in unsupervised and semi-supervised learning. For clustering tasks, the

(C) C. Zepeda, R. Marcial, A. Sánchez
J. L. Zechinelli and M. Osorio (Eds)
Advances in Computer Science and Applications
Research in Computing Science 53, 2011, pp. 115-124

used distance can help to discover the involved groups by measuring how close are the data points to the groups prototypes. In the case of dimensionality reduction, specially manifold learning methods use a distance for finding the shortest path between two data points in the data manifold. For semi-supervised classification tasks, a distance function is used, for instance, for sharing or propagating the class label of labeled elements in the dataset to the closer unlabeled elements.

The most commonly used distance function in machine learning is the Euclidean distance. This has widely been used due to its easily intuitive understanding and interpretation in the real world. Besides, the simplicity of its computation makes it very suitable when we are in the process of selecting a distance function. However, it has been verified that for datasets representation presenting convoluted geometric properties (many foldings), the Euclidean distance is not the proper choice to measure the (dis)similarity between two elements of the data manifold [1–3]. In these cases, the Euclidean distance can be an inexact measure of the proximity between data. This problem becomes more complicated when we work with sample data that resides in a high dimensionality space and we have not additional information about their intrinsic geometry. This situation is the main motivation of this paper, since many datasets involving the previously mentioned properties are presented in real world applications as biomedicine, bioinformatics or web mining. Thus, the previous scenario encourages researchers for modeling this kind of data with an alternative distance function.

An alternative distance function that alleviates, in part, the previously mentioned problem is the geodesic distance, since it measures similarity along the embedded manifold, instead of doing it through the embedding space. Unlike Euclidean distance, geodesic distance follows the geometry of the manifold where data reside. In this way, it may help to avoid some of the distortions (such as breaches of topology preservation) that the use of a standard metric such as the Euclidean distance may introduce when learning the manifold, due to its excessive folding (that is, undesired manifold curvature effects).

Machine learning methods using geodesic distances can be categorized, according to their main task, as unsupervised and semi-supervised learning methods. Within the unsupervised learning methods are found the pioneering works of [1] y [4], where the main task to perform is non-linear dimensionality reduction. Other methods used for this task are [3] and [5]. Also, it can be found some variants of these as in [2]. For clustering and visualization tasks, the geodesic distance has been used in [6] and [7]. On the other hand, the first semi-supervised methods used for classification task were reported in [8] and [9]. These methods, as well as many others that involve the geodesic distance, are known as graph-based methods. Some methods of this type but from different nature are, for example, those based on Support Vector Machines [10], Self-Organizing Maps [11] and Generative models [12].

Most of the graph-based methods, previously mentioned, compute the data point pairwise distance of a graph using the basic Dijkstra algorithm and a full data matrix representation for finding the shortest path between them. The main

problems of this solution involve computational restrictions related to computational time and storage. Thus, the objective of this paper is to show some characteristics about computational time and storage performance for computing the geodesic distance by using the basic Dijkstra algorithm and a full data matrix representation against some alternatives in order that researchers can select the suitable one depending on the available computational resources.

The rest of the paper is organized as follows. In section II, the geodesic distance procedure and some alternatives of the involved modules in it are presented. The experimental results using several UCI datasets are shown in section III. Finally, the conclusions and future work are outlined in section IV.

2 Geodesic distances

At the present time, in some fields of computer science, such as machine learning, the use of alternative metrics like geodesic distance are common and widely used [1, 4, 5]. These methods use the geodesic distance as a basis for generating the data manifold. This metric favours similarity along the manifold, which may help to avoid some of the distortions that the use of a standard metric such as the Euclidean distance may introduce when learning the manifold. In doing so, it can avoid the breaches of topology preservation that may occur due to excessive folding.

The otherwise computationally intractable geodesic metric can be approximated by graph distances [13], so that instead of finding the minimum arc-length between two data points lying on a manifold, we would set to find the shortest path between them, where such path is built by connecting the closest successive data points. In this paper, this is done using the K-rule, which allows connecting the K-nearest neighbours (another alternative is the ϵ-rule, which allows connecting data points $\mathbf{x}$ and $\mathbf{y}$ whenever $\|\mathbf{x} - \mathbf{y}\| < \epsilon$, for some $\epsilon > 0$). A weighted graph is then constructed by using the data and the set of allowed connections. The data are the vertices, the allowed connections are the edges, and the edge labels are the Euclidean distances between the corresponding vertices. If the resulting graph is disconnected, some edges are added using a minimum spanning tree procedure in order to connect it. Finally, the distance matrix of the weighted undirected graph is obtained by repeatedly applying Dijkstra's algorithm [14], which computes the shortest path between all data samples. For illustration, this process is shown in Fig. 1.

2.1 Alternatives modules for graph distance computation

From Fig. 1, it can be shown that there are different alternatives for some of the involved modules in the geodesic distance computation. Some crucial characteristics, about computational time and storage constraints, for computing this distance are the graph representation of the dataset and the shortest path algorithm to be used. On the one hand, two alternatives for graph representation are: adjacency matrix and adjacency list. The former consists in a n by n matrix

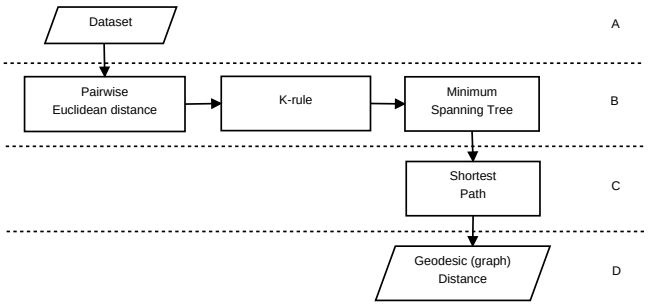

Fig. 1. Graph distance procedure scheme. Stage (A) represents the input data. Stage (B) is for building the weighted, undirected, connected graph. Stage (C) is for computing the geodesic (graph) distance, which is returned in stage (D).

structure, where n is the number of vertices in the graph. If there is an edge from a vertex i to a vertex j, then the element a_{ij} is 1, otherwise it is 0. This kind of structure provide faster access for some applications but can consume huge amounts of memory. The latter considers that each vertex has a list of which vertices it is adjacent to. This structure is often preferred for sparse graphs as it has smaller memory requirements.

On the other hand, three options (of several) for the shortest path algorithm are: (basic) Dijkstra, Dijkstra using a Fibonacci heap and Floyd-Warshall. All of them assumes the graph is a weighted, connected graph. The Dijkstra's algorithm is as follows.

Require: A source node x and a weighted, connected graph $G = (V, E)$, where V is a finite set of vertices (nodes), and E is a collection of edges that connect pairs of vertices.

Ensure: The shortest path between a source node x and every other node in V.

1. $V_{new} = x$, where $x \in V$ is a source node
2. $E_{new} = \emptyset$

repeat

(3a) Choose edge (u, v) from E with minimal weight such that $u \in V_{new}$ and $v \in V \setminus V_{new}$

(3b) Update path's length and add v to V_{new} and (u, v) to E_{new}.

until $V_{new} = V$

It its widely known that the time complexity of the simplest implementation, using the Big-O notation, for this algorithm is $O(|V|^2)$.

For some applications where the obtained graph is a sparse graph, Dijkstra's algorithm can save memory resources by storing the graph in the form of adjacency list and using a Fibonacci heap (F-heap) as a priority queue to implement extracting minimum efficiently. In this way, it can improve the running time of the algorithm 2.1 to $O(|E| + |V|log|V|)$, where the main improvement is made in the step (3a).

A Fibonacci heap is a binary tree which has the property that for every subtree, the root is the minimum item. This data structure is widely used as priority queue [15]. The priority queues are used to keep a dynamic list of different priorities jobs. A F-heap allows several operations as, for instance, ***Insert()*** adds a new job to the queue and ***ExtractMin()*** extracts the highest priority task.

Another approach for computing the shortest path is given by the Floyd-Warshall algorithm, which is an example of dynamic programming. Here, it finds the lengths of the shortest paths between all pairs of vertices. This algorithm is stated as follows.

Require: A weighted, connected graph $G = (V, E)$, where V is a finite set of n vertices (nodes), and E is a collection of edges that connect pairs of vertices.
Ensure: The shortest paths among the nodes in V.
Initialize the $path$ matrix, where $path[i][j] = weight(i, j)$, where $weight(i, j)$ returns the weight of the edge from i to j
for $k = 1...n$ **do**
  for $i = 1...n$ **do**
    for $j = 1...n$ **do**
      $path[i][j] = \min(path[i][j],$
        $path[i][k] + path[k][j])$
    end for
  end for
end for

Unlike Dijkstra's algorithm which assumes all weights are positive, this algorithm can deal with positive or negative edge weights. The complexity for this algorithm is $O(|V|^3)$.

3 Experiments

The goal of the experiments is twofold. Firstly, we aim to experimentally assess which combination of graph representations and shortest path algorithms produce the best computational time and storage performance for computing the geodesic distance of datasets with increasing number of items. Secondly, we aim to evaluate and compare the performance of the best combination (for graph representation and shortest path) found in the previous experiment using the C++ language (gcc compiler) and the Matlab®(R2009a) software. Since Matlab is widely used in the machine learning field and the computer science community, we select it for the comparison.

The process for computing geodesic distances is shown in Fig. 1 and explained in section 2. The experiments are carried out setting the K parameter equal to 10, in order to get a connected graph after the K-rule is applied. After that, the K parameter is set to 1 for showing the time performance of geodesic distance computation with an unconnected and sparse graph. The experiments are performed on a dual-processor 2.3 Ghz BE-2400 desk PC with 2.7Gb RAM.

3.1 Results and discussion

Five datasets, taken from the UCI machine learning repository [16], of increasing number of items were used for the experiments that follow. The first one is *Ecoli*, consisting on 336 7-dimensional points belonging to 8 classes representing protein location sites. The second dataset, German-credit-data (numerical version) herein called *German*, consists of 1000 24-dimensional data points belonging to good or bad credit risks. The third dataset is called *Segmentation*, which is formed by 2310 19-dimensional items representing several measurements of image characteristics belonging to seven different classes. The fourth dataset, *Pageblocks*, involves block measurements of distinct documents corresponding to five classes. It consists of 5473 items described by 10 attributes. The fifth set, herein called *Pendigits*, consists of 10992 16-dimensional items corresponding to (x, y) tablet coordinate information measurements, which belong to ten digits (classes).

The time performance results for computing geodesic (graph) distances, using $K = 10$, are shown in Table 1. Here, a combination of Adjacency matrix for graph representation and basic Dijkstra for shortest path algorithm outperformed the other combinations, except for *Pageblocks*, whereas the memory is not exceeded. This is due to the faster access to elements in an adjacency matrix when basic Dijkstra's algorithm required them. It is interesting to notice how the time performance for the adjacency list representation and Dijkstra is better for large datasets. Using Dijkstra, the time proportion between the adjacency matrix and list is decreased when the number of items is increased. It means that the graph storage by means of an adjacency list is crucial for large datasets. This effect is pronounced for the large *Pendigits* set, where the matrix representation can not deal with it due to operating system (it dedicates approximately 700 Mb for each process) memory restrictions. For large datasets, as *Pendigits*, it can be observed the best combination is for adjacency list and Dijkstra using Fibonacci heaps. Moreover, using the list representation, if time results are compared for Dijkstra and Dijkstra using F-heaps the time proportion is decreased when number of items is increased and this difference becomes better for Dijkstra implemented with F-heaps. This tendency is similar for the matrix representation. Thus, it can be inferred that for large and very large datasets the best time performance for computing geodesic distances should be using an adjacency list (or matrix, when storage restrictions are discarded) representation and Dijkstra using F-heaps. The opposite occurs for Floyd-Warshall algorithm independently from the graph representation. Its performance is noticeable only for small sets.

Now, the K parameter for the K-rule is set to 1, in order to show the computational time and storage performance when the procedure is dealing with an unconnected and sparse graph. The corresponding results are shown in Table 2. In general, it is clear how the modified minimum spanning tree procedure to connect the graph influences the time results. However, the results tendency observed from Table 1 are kept. In addition, it can be inferred from Tables 1 and

Table 1. Computational time performance results for graph distances computation (assuming a connected graph by setting $K = 10$) using several UCI datasets and different settings. '–' symbol means exceeded memory.

Dataset (# items)	**Shortest path**	**Representation**	**Time (s)**
Ecoli (336)	Dijkstra	Adjacency Matrix	0.43
	Dijkstra+F-heaps	Adjacency Matrix	1.19
	Floyd-Warshall	Adjacency Matrix	0.53
	Dijkstra	Adjacency List	0.67
	Dijkstra+F-heaps	Adjacency List	1.59
	Floyd-Warshall	Adjacency List	0.42
German (1000)	Dijkstra	Adjacency Matrix	12.43
	Dijkstra+F-heaps	Adjacency Matrix	25.03
	Floyd-Warshall	Adjacency Matrix	23.67
	Dijkstra	Adjacency List	16.18
	Dijkstra+F-heaps	Adjacency List	38.39
	Floyd-Warshall	Adjacency List	18.71
Segmentation (2310)	Dijkstra	Adjacency Matrix	185.57
	Dijkstra+F-heaps	Adjacency Matrix	297.31
	Floyd-Warshall	Adjacency Matrix	347.16
	Dijkstra	Adjacency List	229.83
	Dijkstra+F-heaps	Adjacency List	511.59
	Floyd-Warshall	Adjacency List	292.89
Pageblocks (5473)	Dijkstra	Adjacency Matrix	3621.90
	Dijkstra+F-heaps	Adjacency Matrix	4031.93
	Floyd-Warshall	Adjacency Matrix	18369.84
	Dijkstra	Adjacency List	3585.92
	Dijkstra+F-heaps	Adjacency List	8039.92
	Floyd-Warshall	Adjacency List	10409.90
Pendigits (10992)	Dijkstra	Adjacency Matrix	– –
	Dijkstra+F-heaps	Adjacency Matrix	– –
	Floyd-Warshall	Adjacency Matrix	– –
	Dijkstra	Adjacency List	124363.18
	Dijkstra+F-heaps	Adjacency List	66105.34
	Floyd-Warshall	Adjacency List	204604.99

2 that the larger the dataset, the less affected the Dijkstra+F-heaps connection algorithm is.

Finally, the time performance of our previous implementations, in C++, for computing geodesic distances using a matrix representation and basic Dijkstra algorithm is compared with Matlab software using the same settings. The K parameter is set to 10. The results are shown in Table 3. Overall, the time performance results for the C++ implementation are better than for Matlab. In fact, any result from Tables 1 and 2 is better than the respectively obtained by Matlab. Besides, the time performance is exponentially increased when the number of items are increased using the Matlab implementation. Also, it is noticeable that Matlab can not deal with medium size sets as *Pageblocks*. Since an adjacency matrix representation is used, the memory restriction tendency is the same as in Tables 1 and 2 but is more restrictive for Matlab.

4 Conclusion

In this paper, a procedure for geodesic distance computation was carried out. Different alternatives for graph representation and shortest path algorithm, in-

Table 2. Computational time performance results for graph distances computation (assuming an unconnected, sparse graph by setting $K = 1$) using several UCI datasets and different settings. '–' symbol means exceeded memory.

Dataset (# items)	**Shortest path**	**Representation**	**Time (s)**
Ecoli (336)	Dijkstra	Adjacency Matrix	0.47
	Dijkstra+F-heaps	Adjacency Matrix	1.21
	Floyd-Warshall	Adjacency Matrix	0.6
	Dijkstra	Adjacency List	0.67
	Dijkstra+F-heaps	Adjacency List	1.57
	Floyd-Warshall	Adjacency List	0.44
German (1000)	Dijkstra	Adjacency Matrix	12.85
	Dijkstra+F-heaps	Adjacency Matrix	25.72
	Floyd-Warshall	Adjacency Matrix	23.32
	Dijkstra	Adjacency List	16.18
	Dijkstra+F-heaps	Adjacency List	37.89
	Floyd-Warshall	Adjacency List	19.27
Segmentation (2310)	Dijkstra	Adjacency Matrix	186.55
	Dijkstra+F-heaps	Adjacency Matrix	294.22
	Floyd-Warshall	Adjacency Matrix	345.38
	Dijkstra	Adjacency List	228.47
	Dijkstra+F-heaps	Adjacency List	507.53
	Floyd-Warshall	Adjacency List	192.38
Pageblocks (5473)	Dijkstra	Adjacency Matrix	3483.08
	Dijkstra+F-heaps	Adjacency Matrix	3955.05
	Floyd-Warshall	Adjacency Matrix	10867.04
	Dijkstra	Adjacency List	5549.91
	Dijkstra+F-heaps	Adjacency List	7678.91
	Floyd-Warshall	Adjacency List	10179.90
Pendigits (10992)	Dijkstra	Adjacency Matrix	––
	Dijkstra+F-heaps	Adjacency Matrix	––
	Floyd-Warshall	Adjacency Matrix	––
	Dijkstra	Adjacency List	131085.17
	Dijkstra+F-heaps	Adjacency List	67312.69
	Floyd-Warshall	Adjacency List	193720.78

Table 3. C++ vs. Matlab time performance results for graph distances computation using Dijkstra's algorithm and an adjacency matrix

Dataset (# items)	**Language**	**Time (s)**
Ecoli (336)	C++	0.43
	Matlab	8.60
German (1000)	C++	12.43
	Matlab	220.98
Segmentation (2310)	C++	185.57
	Matlab	2479.50
Pageblocks (5473)	C++	3621.90
	Matlab	––
Pendigits (10992)	C++	––
	Matlab	––

volved in this procedure, were assessed using different UCI datasets with increasing number of items. Experimental results have shown that the use an adjacency matrix for storing the corresponding graph and Dijkstra's algorithm is recommendable for computing geodesic distances of small and medium datasets. When the number of items are increased forward larger datasets the use of an adjacency list for graph representation/storage becomes crucial in this computation. Furthermore, it is shown that for large sets the use of list representation and Dijkstra using Fibonacci heaps produce better time performance than any other of the analyzed graph representation and shortest path algorithms.

Also, a comparison of a C++ and Matlab implementation for geodesic distances computation was evaluated. The experimental results have shown that the C++ implementation for this procedure is much faster than the corresponding one in Matlab. Moreover, the computational storage (memory) constraint is more restrictive for Matlab than for the C++ implementation.

As future work, it is envisaged the insertion of the obtained geodesic distance computation module into manifold learning methods for dimensionality reduction.

Acknowledgments. Authors gratefully acknowledge funding from the Mexican SEP (PROMEP program) research project PROMEP/103.5/10/5058.

References

1. Tenenbaum, J.B., de Silva, V., Langford, J.C.: A global geometric framework for nonlinear dimensionality reduction. Science **290** (2000) 2319–2323
2. de Silva, V., Tenenbaum, J.: Global versus local methods in nonlinear dimensionality reduction. In Becker, S., Thrun, S., Obermayer, K., eds.: Advances in Neural Information Processing Systems. Volume 15., The MIT Press (2003)
3. Belkin, M., Niyogi, P.: Laplacian eigenmaps for dimensionality reduction and data representation. Neural Computation **15**(6) (2003) 1373–1396
4. Roweis, S.T., Lawrence, K.S.: Nonlinear dimensionality reduction by locally linear embedding. Science (290) (2000) 2323–2326
5. Lee, J.A., Lendasse, A., Verleysen, M.: Curvilinear distance analysis versus isomap. In: Proceedings of European Symposium on Artificial Neural Networks (ESANN). (2002) 185–192
6. Archambeau, C., Verleysen, M.: Manifold constrained finite gaussian mixtures. In Cabestany, J., Prieto, A., Sandoval, D.F., eds.: Proceedings of IWANN. Volume LNCS 3512., Springer-Verlag (2005) 820–828
7. Cruz-Barbosa, R., Vellido, A.: Geodesic Generative Topographic Mapping. In Geffner, H., Prada, R., Alexandre, I., David, N., eds.: Proceedings of the 11th Ibero-American Conference on Artificial Intelligence (IBERAMIA 2008). Volume 5290 of LNAI., Springer (2008) 113–122
8. Zhu, X., Ghahramani, Z.: Learning from labeled and unlabeled data with label propagation. Technical report, CMU-CALD-02-107, Carnegie Mellon University (2002)
9. Belkin, M., Niyogi, P.: Using manifold structure for partially labelled classification. In: Advances in Neural Information Processing Systems (NIPS). Volume 15., MIT Press (2003)

10. Wu, Z., Li, C.H., Zhu, J., Huang, J.: A semi-supervised SVM for manifold learning. In: Proceedings of the 18th International Conference on Pattern Recognition, IEEE Computer Society (2006)
11. Herrmann, L., Ultsch, A.: Label propagation for semi-supervised learning in self-organizing maps. In: Proceedings of the 6th WSOM 2007. (2007)
12. Cruz-Barbosa, R., Vellido, A.: Semi-supervised geodesic generative topographic mapping. Pattern Recognition Letters **31**(3) (2010) 202–209
13. Bernstein, M., de Silva, V., Langford, J.C., Tenenbaum, J.B.: Graph approximations to geodesics on embedded manifolds. Technical report, Stanford University, CA, U.S.A. (2000)
14. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
15. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3) (1987) 596–615
16. Asuncion, A., Newman, D.: UCI machine learning repository [http://www.ics.uci.edu/∼mlearn/MLRepository.html]. University of California, Irvine, School of Information and Computer Sciences (2007)