Introduction to Geometric Learning in Python with Geomstats


To cite this version:


HAL Id: hal-02908006
https://inria.hal.science/hal-02908006
Submitted on 28 Jul 2020
Introduction to Geometric Learning in Python with Geomstats

Nina Miolane‡*, Nicolas Guigui§, Hadi Zaaititi, Christian Shewmake, Hatem Hajri, Daniel Brooks, Alice Le Brigant, Johan Mathe, Benjamin Hou, Yann Thanwerdas, Stefan Heyder, Olivier Peltre, Niklas Koep, Yann Cabanes, Thomas Gerald, Paul Chauchat, Bernhard Kainz, Claire Donnat, Susan Holmes, Xavier Pennec

https://youtu.be/Ju-Wsd84uG0

Abstract—There is a growing interest in leveraging differential geometry in the machine learning community. Yet, the adoption of the associated geometric computations has been inhibited by the lack of a reference implementation. Such an implementation should typically allow its users: (i) to get intuition on concepts from differential geometry through a hands-on approach, often not provided by traditional textbooks; and (ii) to run geometric machine learning algorithms seamlessly, without delving into the mathematical details. To address this gap, we present the open-source Python package geomstats and introduce hands-on tutorials for differential geometry and geometric machine learning algorithms - Geometric Learning - that rely on it. Code and documentation: github.com/geomstats/geomstats and geomstats.ai.

Index Terms—differential geometry, statistics, manifold, machine learning

Introduction

Data on manifolds arise naturally in different fields. Hyperspheres model directional data in molecular and protein biology [KH05] and some aspects of 3D shapes [JDM12], [HVS16]. Density estimation on hyperbolic spaces arises to model electrical impedances [HKKM10], networks [AS14], or reflection coefficients extracted from a radar signal [CBA15]. Symmetric Positive Definite (SPD) matrices are used to characterize data from Diffusion Tensor Imaging (DTI) [PFA06], [YZLM12] and functional Magnetic Resonance Imaging (fMRI) [STK05]. These manifolds are curved, differentiable generalizations of vector spaces. Learning from data on manifolds thus requires techniques from the mathematical discipline of differential geometry. As a result, there is a growing interest in leveraging differential geometry in the machine learning community, supported by the fields of Geometric Learning and Geometric Deep Learning [BBL17].

Despite this need, the adoption of differential geometric computations has been inhibited by the lack of a reference implementation. Projects implementing code for geometric tools are often custom-built for specific problems and are not easily reused. Some Python packages do exist, but they mainly focus on optimization (Pymanopt [TKW16], Geoopt [BG18], [Koc19], McTorch [MJK18]), are dedicated to a single manifold (PyRiemann [Bar15], PyQuaternion [Wyn14], PyGeometry [Cen12]), or lack unit-tests and continuous integration (TheanoGeometry [KS17]). An open-source, low-level implementation of differential geometry and associated learning algorithms for manifold-valued data is thus thoroughly welcome.

Geomstats is an open-source Python package built for machine learning with data on non-linear manifolds [MGLB17]: a field called Geometric Learning. The library provides object-oriented and extensively unit-tested implementations of essential manifolds, operations, and learning methods with support for different execution backends - namely NumPy, PyTorch, and TensorFlow. This paper illustrates the use of geomstats through hands-on introductory tutorials of Geometric Learning. These tutorials enable users: (i) to build intuition for differential geometry through a hands-on approach, often not provided by traditional textbooks; and (ii) to run geometric machine learning algorithms seamlessly without delving into the lower-level computational or mathematical details. We emphasize that the tutorials are not meant to replace theoretical expositions of differential geometry and geometric learning [Pos01], [PSF19]. Rather, they will complement them with an intuitive, didactic, and engineering-oriented approach.

Presentation of Geomstats

The package geomstats is organized into two main modules: geometry and learning. The module geometry implements low-level differential geometry with an object-oriented paradigm and two main parent classes: Manifold and RiemannianMetric. Standard manifolds like the Hypersphere or the Hyperbolic space are classes that inherit from Manifold. At the time of writing, there are over 15 manifolds implemented in geomstats. The class RiemannianMetric provides computations related to Riemannian geometry on such manifolds such as the inner product of two tangent vectors at a base point, the geodesic distance between two points, the Exponential and Logarithm maps at a base point, and many others.

The module learning implements statistics and machine learning algorithms for data on manifolds. The code is object-oriented and classes inherit from scikit-learn base classes and mixins such as BaseEstimator, ClassifierMixin, or RegressorMixin. This module provides implementations...
of Fréchet mean estimators, K-means, and principal component analysis (PCA) designed for manifold data. The algorithms can be applied seamlessly to the different manifolds implemented in the library.

The code follows international standards for readability and ease of collaboration, is vectorized for batch computations, undergoes unit-testing with continuous integration, and incorporates both TensorFlow and PyTorch backends to allow for GPU acceleration. The package comes with a visualization module that enables users to visualize and further develop an intuition for differential geometry. In addition, the datasets module provides instructive toy datasets on manifolds. The repositories examples and notebooks provide convenient starting points to get familiar with geomstats.

First Steps
To begin, we need to install geomstats. We follow the installation procedure described in the first steps of the online documentation. Next, in the command line, we choose the backend of interest: NumPy, PyTorch or TensorFlow. Then, we open the iPython notebook and import the backend together with the visualization module. In the command line:

```python
export GEOMSTATS_BACKEND=numpy
```
then, in the notebook:

```python
import geomstats.backend as gs
import geomstats.visualization as visualization
visualization.tutorial_matplotlib()
```

INFO: Using numpy backend

Modules related to matplotlib and logging should be imported during setup too. More details on setup can be found on the documentation website: geomstats.ai. All standard NumPy functions should be called using the gs. prefix - e.g. gs.exp, gs.log - in order to automatically use the backend of interest.

Tutorial: Statistics and Geometric Statistics
This tutorial illustrates how Geometric Statistics and Learning differ from traditional Statistics. Statistical theory is usually defined for data belonging to vector spaces, which are linear spaces. For example, we know how to compute the mean of a set of numbers or of multidimensional arrays.

Now consider a non-linear space: a manifold. A manifold $M$ of dimension $m$ is a space that is possibly curved but that looks like an $m$-dimensional vector space in a small neighborhood of every point. A sphere, like the earth, is a good example of a manifold. What happens when we apply statistical theory defined for linear vector spaces to data that does not naturally belong to a linear space? For example, what happens if we want to perform statistics on the coordinates of world cities lying on the earth’s surface: a sphere? Let us compute the mean of two data points on the sphere using the traditional definition of the mean.

```python
from geomstats.geometry.hypersphere import \n    Hypersphere

n_samples = 2
sphere = Hypersphere(dim=2)
points_in_manifold = sphere.random_uniform{
    n_samples=n_samples}
```

The result is shown in Figure 1 (left). What happened? The mean of two points on a manifold (the sphere) is not on the manifold. In our example, the mean of these cities is not on the earth’s surface. This leads to errors in statistical computations. The line `sphere.belongs(linear_mean)` returns False. For this reason, researchers aim to build a theory of statistics that is - by construction - compatible with any structure with which we equip the manifold. This theory is called Geometric Statistics, and the associated learning algorithms: Geometric Learning.

In this specific example of mean computation, Geometric Statistics provides a generalization of the definition of “mean” to manifolds: the Fréchet mean.

```python
from geomstats.learning.frechet_mean import \n    FrechetMean

estimator = FrechetMean(metric=sphere.metric)
estimator.fit(points_in_manifold)
frechet_mean = estimator.estimate_
```

Notice in this code snippet that geomstats provides classes and methods whose API will be instantly familiar to users of the widely-adopted scikit-learn. We plot the result in Figure 1 (right). Observe that the Fréchet mean now belongs to the surface of the sphere!

Beyond the computation of the mean, geomstats provides statistics and learning algorithms on manifolds that leverage their specific geometric structure. Such algorithms rely on elementary operations that are introduced in the next tutorial.

Tutorial: Elementary Operations for Data on Manifolds
The previous tutorial showed why we need to generalize traditional statistics for data on manifolds. This tutorial shows how to perform the elementary operations that allow us to “translate” learning algorithms from linear spaces to manifolds.

We import data that lie on a manifold: the world cities dataset, that contains coordinates of cities on the earth’s surface. We visualize it in Figure 2.

```python
from geomstats.geometry.hypersphere import \n    Hypersphere

n_samples = 2
sphere = Hypersphere(dim=2)
points_in_manifold = sphere.random_uniform{
    n_samples=n_samples}
```

```python
from geomstats.datasets.utils as data_utils
data, names = data_utils.load_cities()
```

```python
linear_mean = gs.sum(\n    points_in_manifold, axis=0) / n_samples
```

```python
frechet_mean = estimator.estimate_
```

```python
visualize.tutorial_matplotlib()
```
How can we compute with data that lie on such a manifold? The elementary operations on a vector space are addition and subtraction. In a vector space (in fact seen as an affine space), we can add a vector to a point and subtract two points to get a vector. Can we generalize these operations in order to compute on manifolds?

For points on a manifold, such as the sphere, the same operations are not permitted. Indeed, adding a vector to a point will not give a point that belongs to the manifold: in Figure 3, adding the black tangent vector to the blue point gives a point that is outside the surface of the sphere. So, we need to generalize the operations of addition and subtraction.

On manifolds, the exponential map is the operation that generalizes the addition of a vector to a point. The exponential map takes the following inputs: a point and a tangent vector to the manifold at that point. These are shown in Figure 3 using the blue point and its tangent vector, respectively. The exponential map returns the point on the manifold that is reached by “shooting” with the tangent vector from the point. “Shooting” means following a “geodesic” on the manifold, which is the dotted path in Figure 3. A geodesic, roughly, is the analog of a straight line for general manifolds - the path whose length, or energy, is minimal between two points, where the notions of length and energy are defined by the Riemannian metric. This code snippet shows how to compute the exponential map and the geodesic with geomstats.

```python
from geomstats.geometry.hypersphere import \n    Hypersphere

sphere = Hypersphere(dim=2)
initial_point = paris = data[19]
vector = gs.array([1, 0, 0.8])
tangent_vector = sphere.to_tangent(
    vector, base_point=initial_point)
end_point = sphere.metric.to_tangent(
    tangent_vector, base_point=initial_point)

geodesic = sphere.metric.geodesic(
    initial_point=initial_point,
    initial_tangent_vec=tangent_vector)
```

Similarly, on manifolds, the logarithm map is the operation that generalizes the subtraction of two points on vector spaces. The logarithm map takes two points on the manifold as inputs and returns the tangent vector required to “shoot” from one point to the other. At any point, it is the inverse of the exponential map. In Figure 3, the logarithm of the orange point at the blue point returns the tangent vector in black. This code snippet shows how to compute the logarithm map with geomstats.

```python
log = sphere.metric.log(
    point=end_point, base_point=initial_point)
```

We emphasize that the exponential and logarithm maps depend on the “Riemannian metric” chosen for a given manifold: observe in the code snippets that they are not methods of the sphere object, but rather of its metric attribute. The Riemannian metric defines the notion of exponential, logarithm, geodesic and distance between points on the manifold. We could have chosen a different metric on the sphere that would have changed the distance between the points: with a different metric, the “sphere” could, for example, look like an ellipsoid.

Using the exponential and logarithm maps instead of linear addition and subtraction, many learning algorithms can be generalized to manifolds. We illustrated the use of the exponential and logarithm maps on the sphere only; yet, geomstats provides their implementation for over 15 different manifolds in its geometry module with support for a variety of Riemannian metrics. Consequently, geomstats also implements learning algorithms on manifolds, taking into account their specific geometric structure by relying on the operations we just introduced. The next tutorials show more involved examples of such geometric learning algorithms.

**Tutorial: Classification of SPD Matrices**

**Tutorial context and description**

We demonstrate that any standard machine learning algorithm can be applied to data on manifolds while respecting their geometry. In the previous tutorials, we saw that linear operations (mean, linear weighting, addition and subtraction) are not defined on manifolds. However, each point on a manifold has an associated tangent space which is a vector space. As such, in the tangent space, these operations are well defined! Therefore, we can use the logarithm map (see Figure 3 from the previous tutorial) to go from points on
manifolds to vectors in the tangent space at a reference point. This first strategy enables the use of traditional learning algorithms on manifolds.

A second strategy can be designed for learning algorithms, such as K-Nearest Neighbors classification, that rely only on distances or dissimilarity metrics. In this case, we can compute the pairwise distances between the data points on the manifold, using the method `metric.dist`, and feed them to the chosen algorithm.

Both strategies can be applied to any manifold-valued data. In this tutorial, we consider symmetric positive definite (SPD) matrices from brain connectomics data and perform logistic regression and K-Nearest Neighbors classification.

**SPD matrices in the literature**

Before diving into the tutorial, let us recall a few applications of SPD matrices in the machine learning literature. SPD matrices are ubiquitous across many fields [CS16], either as input of or output to a given problem. In DTI for instance, voxels are represented by "diffusion tensors" which are 3x3 SPD matrices representing ellipsoids in their structure. These ellipsoids spatially characterize the diffusion of water molecules in various tissues. Each DTI thus consists of a field of SPD matrices, where each point in space corresponds to an SPD matrix. These matrices then serve as inputs to regression models. In [YZLM12] for example, the authors use an intrinsic local polynomial regression to compare fiber tracts to regression models. In [YZLM12] for example, the authors use an intrinsic local polynomial regression to compare fiber tracts between HIV subjects and a control group. Similarly, in fMRI, it is possible to extract connectivity graphs from time series of patients’ resting-state images [WZD+13]. The regularized graph Laplacians of these graphs form a dataset of SPD matrices. This provides a compact summary of brain connectivity patterns which is useful for assessing neurological responses to a variety of stimuli, such as drugs or patient’s activities.

More generally speaking, covariance matrices are also SPD matrices which appear in many settings. Covariance clustering can be used for various applications such as sound compression in acoustic models of automatic speech recognition (ASR) systems [SMA10] or for material classification [FHP15], among others. Covariance descriptors are also popular image or video descriptors [HHLS16].

Lastly, SPD matrices have found applications in deep learning. The authors of [GWB+19] show that an aggregation of learned deep convolutional features into an SPD matrix creates a robust representation of images which outperforms state-of-the-art methods for visual classification.

**Manifold of SPD matrices**

Let us recall the mathematical definition of the manifold of SPD matrices. The manifold of SPD matrices in $n$ dimensions is embedded in the General Linear group of invertible matrices and defined as:

$$ SPD = \{ S \in \mathbb{R}_{n \times n} : S^T = S, \forall z \in \mathbb{R}^n, z \neq 0, z^T S z > 0 \} $$

The class `SPDMatricesSpace` inherits from the class `EmbeddedManifold` and has an `embedding_manifold` attribute which stores an object of the class `GeneralLinear`. SPD matrices in 2 dimensions can be visualized as ellipses with principal axes given by the eigenvectors of the SPD matrix, and the length of each axis proportional to the square-root of the corresponding eigenvalue. This is implemented in the visualization module of geomstats. We generate a toy data-set and plot it in Figure 4 with the following code snippet.

```python
import geomstats.datasets.sample_sdp_2d as sampler
n_samples = 100
dataset_generator = sampler.DatasetSPD2D(
    n_samples, n_features=2, n_classes=3)
ellipses = visualization.Ellipsis2D()
for i, x in enumerate(data):
    y = sampler.get_label_at_index(i, labels)
ellipses.draw(
    x, color=eellipses.colors[y], alpha=.1)
```

Figure 4 shows a dataset of SPD matrices in 2 dimensions organized into 3 classes. This visualization helps in developing an intuition on the connectomes dataset that is used in the upcoming tutorial, where we will classify SPD matrices in 28 dimensions into 2 classes.

**Classifying brain connectomes in Geomstats**

We now delve into the tutorial in order to illustrate the use of traditional learning algorithms on the tangent spaces of manifolds implemented in geomstats. We use brain connectome data from the MSLP 2014 Schizophrenia Challenge. The connectomes are correlation matrices extracted from the time-series of resting-state fMRIs of 86 patients at 28 brain regions of interest; they are points on the manifold of SPD matrices in $n = 28$ dimensions. Our goal is to use the connectomes to classify patients into two classes: schizophrenic and control. First we load the connectomes and display two of them as heatmaps in Figure 5.

```python
import geomstats.datasets.utils as data_utils
data, patient_ids, labels = \
data_utils.load_connectomes()
```

Multiple metrics can be used to compute on the manifold of SPD matrices [DKZ09]. As mentioned in the previous tutorial, different metrics define different geodesics, exponential and logarithm maps and therefore different algorithms on a given manifold. Here, we import two of the most commonly used metrics on the SPD matrices, the log-Euclidean metric and the affine-invariant metric [PFA06], but we highlight that geomstats contains many more. We also check that our connectome data indeed belongs to the manifold of SPD matrices:
Fig. 5: Subset of the connectomes dataset, available in geomstats with the function load_connectomes from the module datasets.utils. Connectomes are correlation matrices of 28 time-series extracted from fMRI data: they are elements of the manifold of SPD matrices in 28 dimensions. Left: connectome of a schizophrenic subject. Right: connectome of a healthy control.

import geomstats.geometry spd_matrices as spd

manifold = spd.SPDMatrices(n=28)
le_metric = spd.SPDMetricLogEuclidean(n=28)
ai_metric = spd.SPDMetricAffine(n=28)
logging.info(gs.all(manifold.belongs(data)))

INFO: True

Great! Now, although the sum of two SPD matrices is an SPD matrix, their difference or their linear combination with non-positive weights are not necessarily. Therefore we need to work in a tangent space of the SPD manifold to perform simple machine learning that relies on linear operations. The preprocessing module with its ToTangentSpace class allows to do exactly this.

from geomstats.learning.preprocessing import \ 
   ToTangentSpace

ToTangentSpace has a simple purpose: it computes the Fréchet Mean of the data set, and takes the logarithm map of each data point from the mean. This results in a data set of tangent vectors at the mean. In the case of the SPD manifold, these are simply symmetric matrices. ToTangentSpace then squeezes each symmetric matrix into a 1d-vector of size dim = 28 * (28 + 1) / 2, and outputs an array of shape [n_connectomes, dim], which can be fed to your favorite scikit-learn algorithm.

We emphasize that ToTangentSpace computes the mean of the input data, and thus should be used in a pipeline (as e.g. scikit-learn’s StandardScaler) to avoid leaking information from the test set at train time.

from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_validate

pipeline = make_pipeline(
   ToTangentSpace(le_metric), LogisticRegression(C=2))

We use a logistic regression on the tangent space at the Fréchet mean to classify connectomes, and evaluate the model with cross-validation. With the log-Euclidean metric we obtain:

result = cross_validate(pipeline, data, labels)
logging.info(result['test_score'].mean())

INFO: 0.67

And with the affine-invariant metric, replacing le_metric by ai_metric in the above snippet:

INFO: 0.71

We observe that the result depends on the metric. The Riemannian metric indeed defines the notion of the logarithm map, which is used to compute the Fréchet Mean and the tangent vectors corresponding to the input data points. Thus, changing the metric changes the result. Furthermore, some metrics may be more suitable than others for different applications. Indeed, we find published results that show how useful geometry can be with data on the SPD manifold (e.g. [WAZF18], [NDV+14]).

We saw how to use the representation of points on the manifold as tangent vectors at a reference point to fit any machine learning algorithm, and we compared the effect of different metrics on the manifold of SPD matrices. Another class of machine learning algorithms can be used very easily on manifolds with geomstats: those relying on dissimilarity matrices. We can compute the matrix of pairwise Riemannian distances, using the dist method of the Riemannian metric object. In the following code-snippet, we use ai_metric.dist and pass the corresponding matrix pairwise_dist of pairwise distances to scikit-learn’s K-Nearest-Neighbors (KNN) classification algorithm:

from sklearn.neighbors import KNeighborsClassifier

classifier = KNeighborsClassifier(
   metric='precomputed')

result = cross_validate(
   classifier, pairwise_dist, labels)
logging.info(result['test_score'].mean())

INFO: 0.72

This tutorial showed how to leverage geomstats to use standard learning algorithms for data on a manifold. In the next tutorial, we see a more complicated situation: the data points are not provided by default as elements of a manifold. We will need to use the low-level geomstats operations to design a method that embeds the dataset in the manifold of interest. Only then, we can use a learning algorithm.

Tutorial: Learning Graph Representations with Hyperbolic Spaces

Tutorial context and description

This tutorial demonstrates how to make use of the low-level geometric operations in geomstats to implement a method that embeds graph data into the hyperbolic space. Thanks to the discovery of hyperbolic embeddings, learning on Graph-Structured Data (GSD) has seen major achievements in recent years. It had been speculated for years that hyperbolic spaces may better represent GSD than Euclidean spaces [Gro87] [KPK+10] [BPK10] [ASM13]. These speculations have recently been shown effective through concrete studies and applications [NK17] [CCD17] [SDS18] [GZH+19]. As outlined by [NK17], Euclidean embeddings require large dimensions to capture certain complex relations such as the Wordnet noun hierarchy. On the other hand, this complexity can be captured by a lower-dimensional model of hyperbolic geometry such as the hyperbolic space of two dimensions [SDS18], also called the hyperbolic plane. Additionally, hyperbolic embeddings provide better visualizations of clusters on graphs than their Euclidean counterparts [CCD17].
This tutorial illustrates how to learn hyperbolic embeddings in `geomstats`. Specifically, we will embed the Karate Club graph dataset, representing the social interactions of the members of a university Karate club, into the Poincaré ball. Note that we will omit implementation details but an unabridged example and detailed notebook can be found on GitHub in the examples and notebooks directories of `geomstats`.

**Hyperbolic spaces and machine learning applications**

Before going into this tutorial, we review a few applications of hyperbolic spaces in the machine learning literature. First, Hyperbolic spaces arise in information and learning theory. Indeed, the space of univariate Gaussians endowed with the Fisher metric densities is a hyperbolic space [CSS05]. This characterization is used in various fields, for example in image processing, where each image pixel can be represented by a Gaussian distribution [AVF14], or in radar signal processing where the corresponding echo is represented by a stationary Gaussian process [ABY13]. Hyperbolic spaces can also be seen as continuous versions of trees and are therefore interesting when learning representations of hierarchical data [NK17]. Hyperbolic Geometric Graphs (HGG) have also been suggested as a promising model for social networks - where the hyperbolicity appears through a competition between similarity and popularity of an individual [PKS12] and in learning communities on large graphs [GZH19].

**Hyperbolic space**

Let us recall the mathematical definition of the hyperbolic space. The \( n \)-dimensional hyperbolic space \( H_n \) is defined by its embedding in the \((n+1)\)-dimensional Minkowski space as:

\[
H_n = \{ x \in \mathbb{R}^{n+1} : -x_1^2 + \ldots + x_{n+1}^2 = -1 \}.
\]

In `geomstats`, the hyperbolic space is implemented in the class Hyperboloid and PoincareBall, which use different coordinate systems to represent points. These classes inherit from the class EmbeddedManifold and have an embedding_manifold attribute which stores an object of the Hyperboloid class in `geomstats`. The 2-dimensional hyperbolic space is called the hyperbolic plane or Poincaré disk.

**Learning graph representations with hyperbolic spaces in `geomstats`**

**Parameters and Initialization:** We now proceed with the tutorial embedding the Karate club graph in a hyperbolic space. In the Karate club graph, each node represents a member of the club, and each edge represents an undirected relation between two members. We first load the Karate club dataset, display it in Figure 6 and print information regarding its nodes and vertices to provide insights into the graph’s complexity.

```python
karate_graph = data_utils.load_karate_graph()
```

```python
nb_vertices_by_edges = {
    (len(e_2) for _, e_2 in karate_graph.edges.items())
}
```

```python
logging.info('Number of vertices: %s', len(karate_graph.edges))
```

```python
logging.info('Mean edge-vertex ratio: %s',
             (sum(nb_vertices_by_edges, 0) / len(karate_graph.edges)))
```

INFO: Number of vertices: 34
INFO: Mean edge-vertex ratio: 4.588235294117647

Table 1 defines the parameters needed to embed this graph into a hyperbolic space. The number of hyperbolic dimensions should be high \((n > 10)\) only for graph datasets with a large number of nodes and edges. In this tutorial we consider a dataset with only 34 nodes, which are the 34 members of the Karate club. The Poincaré ball of two dimensions is therefore sufficient to capture the complexity of the graph. We instantiate an object of the class PoincareBall in `geomstats`.

```python
from geomstats.geometry.poincare_ball
import PoincareBall
```

```python
hyperbolic_manifold = PoincareBall(dim=2)
```

Other parameters such as `max_epochs` and `lr` will be tuned specifically for each dataset, either manually leveraging visualization functions or through a grid/random search that looks for parameter values maximizing some performance function (a measure for cluster separability, normalized mutual information (NMI), or others). Similarly, the number of negative samples and context size are hyperparameters and will be further discussed below.

**Learning the embedding by optimizing a loss function:** Denote \( V \) as the set of nodes and \( E \subset V \times V \) the set of edges of the graph. The goal of hyperbolic embedding is to provide a faithful and exploitable representation of the graph. This goal is mainly achieved by preserving first-order proximity that encourages nodes sharing edges to be close to each other. We can additionally pre-

![Fig. 6: Karate club dataset, available in `geomstats` with the function `load_karate_graph` from the module `datasets.utils`. This dataset is a graph, where each node represents a member of the club and each edge represents a tie between two members of the club.](attachment:karate_graph.png)
Gradient direction for context samples  
Gradient direction for negative samples

**Fig. 7:** Embedding of the graph’s nodes \( \{v_i\} \) as points \( \{\phi_i\} \) of the hyperbolic plane \( H_2 \), also called the Poincaré ball of 2 dimensions. The blue and red arrows represent the direction of the gradient of the loss function \( \mathcal{L} \) from Equation 2. This brings context samples closer and separates negative samples.

serve second-order proximity by encouraging two nodes sharing the “same context”, i.e. not necessarily directly connected but sharing a neighbor, to be close. We define a context size (here equal to 1) and call two nodes “context samples” if they share a neighbor, and “negative samples” otherwise. To preserve first and second-order proximities, we adopt the following loss function similar to [NK17] and consider the “negative sampling” approach from [MSC+13]:

\[
\mathcal{L} = - \sum_{v_i \in V} \sum_{j \in C_i} \log(\sigma(-d^2(\phi_i, \phi_j))) + \sum_{v_k \sim \mathcal{P}_n} \log(\sigma(d^2(\phi_i, \phi_k))) \tag{2}
\]

where \( \sigma(x) = (1 + e^{-x})^{-1} \) is the sigmoid function and \( \phi \in H_2 \) is the embedding of the \( i \)-th node of \( V \). \( C_i \) the nodes in the context of the \( i \)-th node, \( \phi_j' \in H_2 \) the embedding of \( v_j \in C_i \). Negatively sampled nodes \( v_k \) are chosen according to the distribution \( \mathcal{P}_n \) such that \( \mathcal{P}_n(v) = (\text{deg}(v)^{3/4} \sum_{v_i \in V} \text{deg}(v_i)^{3/4})^{-1} \).

Intuitively one can see in Figure 7 that minimizing \( \mathcal{L} \) makes the distance between \( \phi_i \) and \( \phi_j \) smaller, and the distance between \( \phi_i \) and \( \phi_k \) larger. Therefore by minimizing \( \mathcal{L} \), one obtains representative embeddings.

**Riemannian optimization:** Following the literature on optimization on manifolds [GBH18], we use the following gradient updates to optimize \( \mathcal{L} \):

\[
\phi^{t+1} = \text{Exp}_\phi\left(-lr \frac{\partial \mathcal{L}}{\partial \phi}\right)
\]

where \( \phi \) is a parameter of \( \mathcal{L} \), \( t \in \{1,2,\cdots\} \) is the iteration number, and \( lr \) is the learning rate. The formula consists of first computing the usual gradient of the loss function for the direction in which the parameter should move. The Riemannian exponential map \( \text{Exp} \) is the operation introduced in the second tutorial: it takes a base point \( \phi' \) and a tangent vector \( T \) and returns the point \( \phi^{t+1} \). The Riemannian exponential map is a method of the PoincareBallMetric class in the geometry module of geomstats. It allows us to implement a straightforward generalization of standard gradient update in the Euclidean case. To compute the gradient of \( \mathcal{L} \), we need to compute the gradients of: (i) the squared distance \( d^2(x,y) \) on the hyperbolic space, (ii) the log sigmoid \( \log(\sigma(x)) \), and (iii) the composition of (i) with (ii).

For (i), we use the formula proposed by [ABY13] which uses the Riemannian logarithmic map. Like the exponential \( \text{Exp} \), the logarithmic map is implemented under the PoincareBallMetric.

```python
def grad_squared_distance(point_a, point_b, manifold):
    log = manifold.metric.log(point_b, point_a)
    return -2 * log
```

For (ii), we compute the well-known gradient of the logarithm of the sigmoid function as: \( \log(\sigma(x)) = (1 + \exp(x))^{-1} \). For (iii), we apply the composition rule to obtain the gradient of \( \mathcal{L} \). The following function computes \( \mathcal{L} \) and its gradient on the context samples, while ignoring the part dealing with the negative samples for simplicity of exposition. The code implementing the whole loss function is available on GitHub.

```python
def loss(example, context_embedding, manifold):
    context_distance = manifold.metric.squared_dist(example, context_embedding)
    context_loss = log_sigmoid(-context_distance)
    context_log_sigmoid_grad = -grad_log_sigmoid(-context_distance)
    context_distance_grad = grad_squared_distance(example, context_embedding)
    context_grad = (context_log_sigmoid_grad + context_distance_grad)
    return context_loss, -context_grad
```

**Capturing the graph structure:** We perform initialization computations that capture the graph structure. We compute random walks initialized from each \( v_i \) up to some length (five by default). The context nodes \( v_j \) will be later picked from the random walk of \( v_i \).

```python
random_walks = karate_graph.random_walk()
```

Negatively sampled nodes \( v_k \) are chosen according to the previously defined probability distribution function \( \mathcal{P}_n(v) \) implemented as

```python
negative_table_parameter = 5
negative_sampling_table = []
for i, nb_v in enumerate(nb_vertices_by_edges):
    negative_sampling_table += ([i] * int((nb_v * (3 / 4)))
        + negative_table_parameter)
```

**Numerically optimizing the loss function:** We can now embed the Karate club graph into the Poincaré disk. The details of the initialization are provided on GitHub. The array embeddings contains the embeddings \( \phi \)'s of the nodes \( v_i \)'s of the current iteration. At each iteration, we compute the gradient of \( \mathcal{L} \). The graph nodes are then moved in the direction pointed by the gradient. The movement of the nodes is performed by following geodesics in the Poincaré disk in the gradient direction. In practice, the key to obtaining a representative embedding is to carefully tune the learning rate so that all of the nodes make small movements at each iteration.

A first level loop iterates over the epochs while the table total_loss records the value of \( \mathcal{L} \) at each iteration. A second
level nested loop iterates over each path in the previously computed random walks. Observing these walks, note that nodes having many edges appear more often. Such nodes can be considered as important crossroads and will therefore be subject to a greater number of embedding updates. This is one of the main reasons why random walks have proven to be effective in capturing the structure of graphs. The context of each $v_j$ will be the set of nodes $v_i$ belonging to the random walk from $v_j$. The context size specified earlier will limit the length of the walk to be considered. Similarly, we use the same context size to limit the number of negative samples. We find $\phi_i$ from the embeddings array.

A third and fourth level nested loops will iterate on each $v_j$ and $v_k$. From within, we find $\phi'_j$ and $\phi'_k$ and call the loss function to compute the gradient. Then the Riemannian exponential map is applied to find the new value of $\phi_i$ as we mentioned before.

```python
for epoch in range(max_epochs):
    total_loss = []
    for path in random_walks:
        for example_index, one_path in enumerate(path):
            context_index = path[max(0, example_index - context_size):
                                 min(example_index + context_size, len(path))]
            negative_index = gs.random.randint(negative_sampling_table.shape[0],
                                               size=(len(context_index), n_negative))
            negative_index = (negative_sampling_table[negative_index])
            example_embedding = embeddings[one_path]
            context_embedding = (embeddings[one_context_i])
            negative_embedding = (embeddings[one_negative_i])
            l, g_ex = loss(example_embedding, context_embedding, negative_embedding, hyperbolic_manifold)
            total_loss.append(l)
        example_to_update = (embeddings[one_path])
        embeddings[one_path] = (hyperbolic_metric.exp(-lr * g_ex, example_to_update))
    logging.info('Iteration %d loss_value %f', epoch, sum(total_loss, 0) / len(total_loss))
```

INFO: iteration 0 loss_value 1.819844
INFO: iteration 14 loss_value 1.363593

Figure 8 shows the graph embedding at different iterations with the true labels of each node represented with color. Notice how the embedding at convergence separates well the two clusters. Thus, we use the same context size to limit the number of negative samples. We find $\phi_i$ from the embeddings array.

```python
for epoch in range(max_epochs):
    total_loss = []
    for path in random_walks:
        for example_index, one_path in enumerate(path):
            context_index = path[max(0, example_index - context_size):
                                 min(example_index + context_size, len(path))]
            negative_index = gs.random.randint(negative_sampling_table.shape[0],
                                               size=(len(context_index), n_negative))
            negative_index = (negative_sampling_table[negative_index])
            example_embedding = embeddings[one_path]
            context_embedding = (embeddings[one_context_i])
            negative_embedding = (embeddings[one_negative_i])
            l, g_ex = loss(example_embedding, context_embedding, negative_embedding, hyperbolic_manifold)
            total_loss.append(l)
        example_to_update = (embeddings[one_path])
        embeddings[one_path] = (hyperbolic_metric.exp(-lr * g_ex, example_to_update))
    logging.info('Iteration %d loss_value %f', epoch, sum(total_loss, 0) / len(total_loss))
```

INFO: iteration 0 loss_value 1.819844
INFO: iteration 14 loss_value 1.363593

Figure 8 shows the graph embedding at different iterations with the true labels of each node represented with color. Notice how the embedding at convergence separates well the two clusters. Thus, it seems that we have found a useful representation of the graph.

To demonstrate the usefulness of the embedding learned, we show how to apply a $K$-means algorithm in the hyperbolic plane to predict the label of each node in an unsupervised approach. We use the learning module of geomstats and instantiate an object of the class RiemannianKMeans. Observe again how geomstats classes follow scikit-learn’s API. We set the number of clusters and plot the results.

```python
from geomstats.learning.kmeans import RiemannianKMeans
kmeans = RiemannianKMeans(
    hyperbolic_manifold.metric, n_clusters=2,
    mean_method='frechet-poincare-ball')
centroids = kmeans.fit(X=embeddings, max_iter=100)
labels = kmeans.predict(X=embeddings)
```

Figure 9 shows the true labels versus the predicted ones: the two groups of the karate club members have been well separated!

**Conclusion**

This paper demonstrates the use of geomstats in performing geometric learning on data belonging to manifolds. These tutorials, as well as many other learning examples on a variety of manifolds, can be found at geomstats.ai. We hope that this hands-on presentation of Geometric Learning will help to further democratize the use of differential geometry in the machine learning community.

**Acknowledgements**

This work is partially supported by the National Science Foundation, grant NSF DMS RTG 1501767, the Inria-Stanford associated team GeomStats, and the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement G-Statistics No. 786854).


