Tutorial on the R package TDA

Jisu Kim
Brittany T. Fasy, Jisu Kim, Fabrizio Lecci, Clément Maria, Vincent Rouvreau

Abstract

I present a short tutorial and introduction to using the R package **TDA**, which provides tools for Topological Data Analysis. Given data, the sailent topological features of underlying space can be quantified with persistent homology. **TDA** package provides a function for the persistent homology of the Rips filtration, and a function for the persistent homology of sublevel sets (or superlevel sets) of arbitrary functions evaluated over a grid of points. Some common choice of functions for the latter case, such as the distance function, the distance to a measure, the kNN density estimator, the kernel density estimator, and the kernel distance, are implemented in the TDA package. The R package **TDA** also provides a function for computing the confidence band that determines significance of the features in the resulting persistence diagrams.

Keywords: Topological Data Analysis, Persistent Homology.

1. Introduction

R(http://cran.r-project.org/) is a programming language for statistical computing and graphics.

R has several good properties: R has many packages for statistical computing. Also, R is easy to make (interactive) plots. R is a script language, and it is easy to use. But, R is slow. C or C++ stands on the opposite end: C or C++ also has many packages(or libraries). But, C or C++ is difficult to make plots. C or C++ is a compiler language, and is difficult to use. But, C or C++ is fast. In short, R has short development time but long execution time, and C or C++ has long development time but short execution time.

Several libraries are developed for Topological Data Analysis: for example, **GUDHI**(Maria 2014)(https://project.inria.fr/gudhi/software/), **Dionysus**(Morozov 2007)(http://www.mrzv.org/software/dionysus/), and **PHAT**(Bauer, Kerber, and Reininghaus 2012)(https://code.google.com/p/phat/). They are all written in C++, since Topological Data Analysis is computationally heavy and R is not fast enough.

R package **TDA**(http://cran.r-project.org/web/packages/TDA/index.html) bridges between C++ libraries(**GUDHI**, **Dionysus**, **PHAT**) and R. TDA package provides an R interface for the efficient algorithms of the C++ libraries **GUDHI**, **Dionysus** and **PHAT**. So by using **TDA** package, short development time and short execution time can be both achieved.

R package **TDA** provides tools for Topological Data Analysis. You can compute several different things with **TDA** package: you can compute common distance functions and density estimators, the persistent homology of the Rips filtration, the persistent homology of sublevel sets of a function over a grid, the confidence band for the persistence diagram, and the cluster density trees for density clustering.

2. Setting up

Obviously, you should download R first. R of version at least 3.1.0 is recommended:

http://cran.r-project.org/bin/windows/base/ (for Windows)

http://cran.r-project.org/bin/macosx/ (for (Mac) OS X)

R is part of many Linux distributions, so you should check with your Linux package management system.

You can use whatever IDE that you would like to use(Rstudio, Eclipse, Emacs, Vim...). R itself also provides basic GUI or CUI. I personally use Rstudio:

http://www.rstudio.com/products/rstudio/download/

Before installing R package **TDA**, Four packages are needed to be installed: **parallel**, **FNN**, **igraph**, and **scales**. **parallel** is included when you install R, so you need to install **FNN**, **igraph**, and **scales** by yourself. You can install them by following code (or pushing 'Install R packages' button if you use Rstudio).

```r
##########################################################################
# installing required packages
##########################################################################
if (!require(package = "FNN")) {
  install.packages(pkgs = "FNN")
}
if (!require(package = "igraph")) {
  install.packages(pkgs = "igraph")
}
if (!require(package = "scales")) {
  install.packages(pkgs = "scales")
}
```

After that, you can install R package **TDA** as in the following code (or pushing 'Install R packages' button if you use Rstudio).

```r
##########################################################################
# installing R package TDA
##########################################################################
if (!require(package = "TDA")) {
  install.packages(pkgs = "TDA")
}
```

Once installation is done, R package **TDA** should be loaded as in the following code, before using the package functions.

```r
##########################################################################
# loading R package TDA
##########################################################################
library(package = "TDA")
```

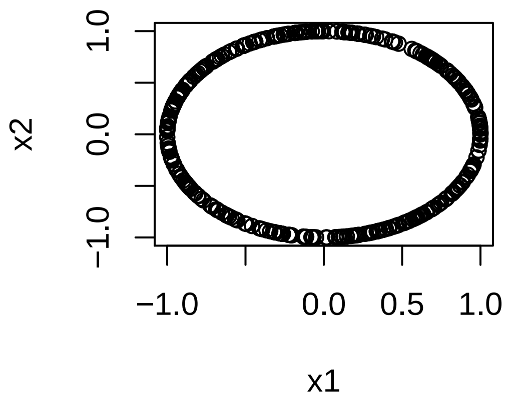
3. Sample on manifolds, Distance Functions, and Density Estimators

3.1. Uniform Sample on manifolds

A set of n points $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$ has been sampled from some distribution P.

- n sample from the uniform distribution on the circle in $\mathbb{R}^2$ with radius r.

```
###########################################################################
# uniform sample on the circle
###########################################################################
circleSample <- circleUnif(n = 400, r = 1)
plot(circleSample)
```



- n sample from the uniform distribution on the sphere S^d in $\mathbb{R}^{d+1}$ with radius r.

```
###########################################################################
# uniform sample on the sphere
###########################################################################
sphereSample <- sphereUnif(n = 10000, d = 2, r = 1)
if (!require(package = "rgl")) {
  install.packages(pkgs = "rgl")
}
library(rgl)
plot3d(sphereSample)
```

- n sample from the uniform distribution on the torus in $\mathbb{R}^3$ with small radius a and large radius b.

```
###########################################################################
# uniform sample on the torus
###########################################################################
```

```r
torusSample <- torusUnif(n = 10000, a = 1.8, c = 5)
if (!require(package = "rgl")) {
  install.packages(pkgs = "rgl")
}
library(rgl)
plot3d(torusSample)
```

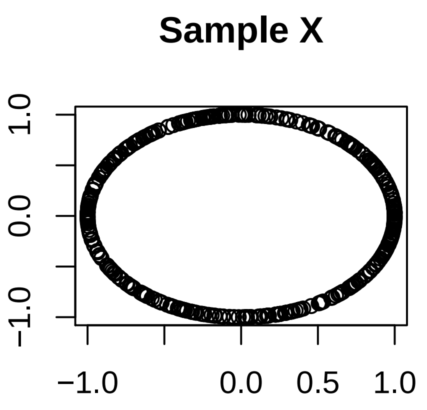
3.2. Distance Functions, and Density Estimators

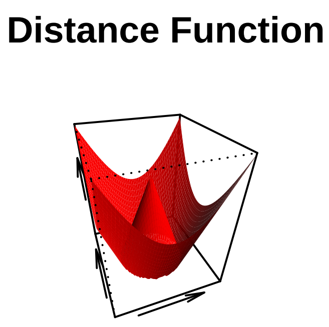
We compute distance functions and density estimators over a grid of points. Suppose a set of points $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$ has been sampled from some distribution P. The following code generates a sample of 400 points from the unit circle and constructs a grid of points over which we will evaluate the functions.

```r
###########################################################################
# uniform sample on the circle, and grid of points
###########################################################################
X <- circleUnif(n = 400, r = 1)

Xlim <- c(-1.6, 1.6)
Ylim <- c(-1.7, 1.7)
by <- 0.065
Xseq <- seq(from = Xlim[1], to = Xlim[2], by = by)
Yseq <- seq(from = Ylim[1], to = Ylim[2], by = by)
Grid <- expand.grid(Xseq, Yseq)
```

- The distance function is defined for each $y \in \mathbb{R}^d$ as $\Delta(y) = \inf_{x \in X} \|x - y\|_2$.

```r
###########################################################################
# distance function
###########################################################################
distance <- distFct(X = X, Grid = Grid)

par(mfrow = c(1,2))
plot(X, xlab = "", ylab = "", main = "Sample X")
persp(x = Xseq, y = Yseq,
      z = matrix(distance, nrow = length(Xseq), ncol = length(Yseq)),
      xlab = "", ylab = "", zlab = "", theta = -20, phi = 35, scale = FALSE,
      expand = 3, col = "red", border = NA, ltheta = 50, shade = 0.5,
      main = "Distance Function")
```





- Given a probability measure P, the distance to measure (DTM) is defined for each $y \in \mathbb{R}^d$ as

$$d_{m_0}(y) = \sqrt{\frac{1}{m_0}\int_0^{m_0} (G_y^{-1}(u))^2 du},$$

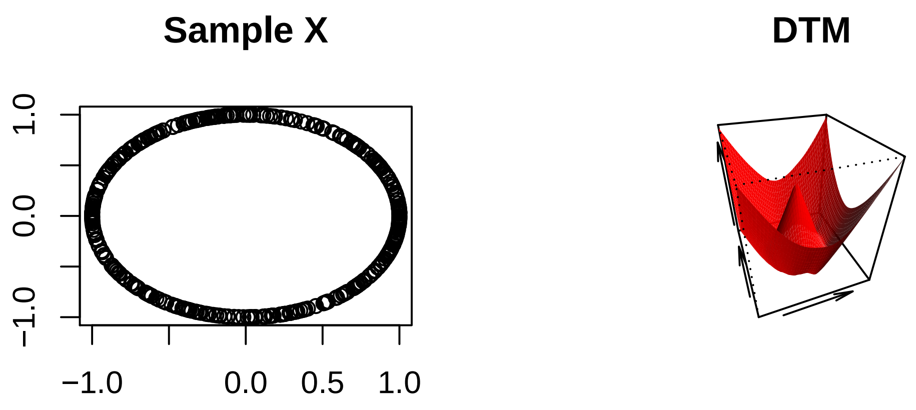
where $G_y(t) = P(\|X - y\| \le t)$ and $0 < m_0 < 1$ is a smoothing parameter. The DTM can be seen as a smoothed version of the distance function. For more details see Chazal, Cohen-Steiner, and Mérigot (2011).

Given $X = \{x_1, \ldots, x_n\}$, the empirical version of the DTM is

$$\hat{d}_{m_0}(y) = \sqrt{\frac{1}{k}\sum_{x_i \in N_k(y)} \|x_i - y\|^2},$$

where $k = \lceil m_0 n \rceil$ and $N_k(y)$ is the set containing the k nearest neighbors of y among $x_1, \ldots, x_n$.

```r
##########################################################################
# distance to measure
##########################################################################
m0 <- 0.1
DTM <- dtm(X = X, Grid = Grid, m0 = m0)

par(mfrow = c(1,2))
plot(X, xlab = "", ylab = "", main = "Sample X")
persp(x = Xseq, y = Yseq,
      z = matrix(DTM, nrow = length(Xseq), ncol = length(Yseq)),
      xlab = "", ylab = "", zlab = "", theta = -20, phi = 35, scale = FALSE,
      expand = 3, col = "red", border = NA, ltheta = 50, shade = 0.5,
      main = "DTM")
```

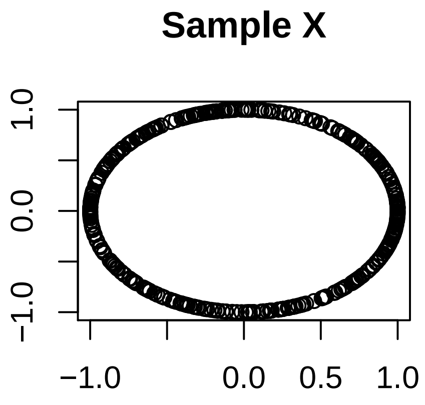


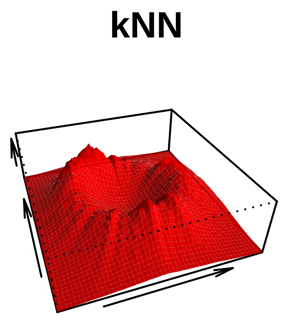
- The k Nearest Neighbor density estimator, for each $y \in \mathbb{R}^d$, is defined as

$$\hat{\delta}_k(y) = \frac{k}{n\ v_d\ r_k^d(y)},$$

where v_n is the volume of the Euclidean d dimensional unit ball and $r_k^d(x)$ is the Euclidean distance form point x to its kth closest neighbor among the points of X.

```
###########################################################################
# k nearest neighbor density estimator
###########################################################################
k <- 60
kNN <- knnDE(X = X, Grid = Grid, k = k)

par(mfrow = c(1,2))
plot(X, xlab = "", ylab = "", main = "Sample X")
persp(x = Xseq, y = Yseq,
      z = matrix(kNN, nrow = length(Xseq), ncol = length(Yseq)),
      xlab = "", ylab = "", zlab = "", theta = -20, phi = 35, scale = FALSE,
      expand = 3, col = "red", border = NA, ltheta = 50, shade = 0.5,
      main = "kNN")
```



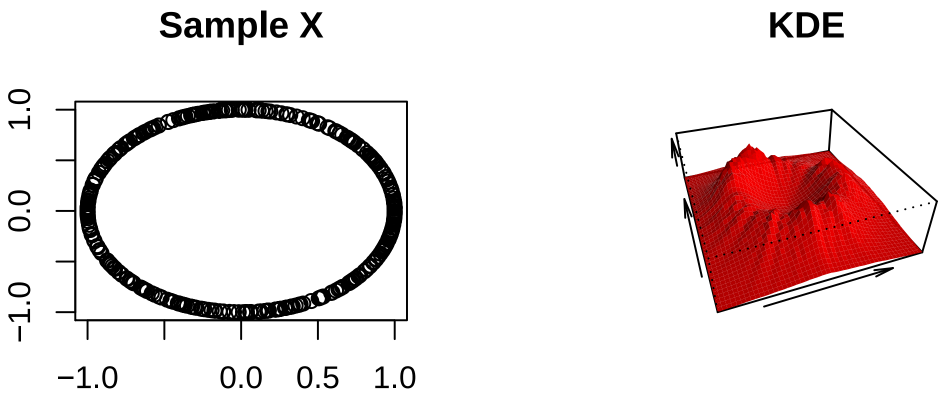


- The Gaussian Kernel Density Estimator (KDE), for each $y \in \mathbb{R}^d$, is defined as

$$\hat{p}_h(y) = \frac{1}{n(\sqrt{2\pi}h)^d} \sum_{i=1}^{n} \exp\left(\frac{-\|y - x_i\|_2^2}{2h^2}\right).$$

where h is a smoothing parameter.

```
###########################################################################
# kernel density estimator
###########################################################################
h <- 0.3
KDE <- kde(X = X, Grid = Grid, h = h)

par(mfrow = c(1,2))
plot(X, xlab = "", ylab = "", main = "Sample X")
persp(x = Xseq, y = Yseq,
      z = matrix(kNN, nrow = length(Xseq), ncol = length(Yseq)),
      xlab = "", ylab = "", zlab = "", theta = -20, phi = 35, scale = FALSE,
      expand = 3, col = "red", border = NA, ltheta = 50, shade = 0.5,
      main = "KDE")
```

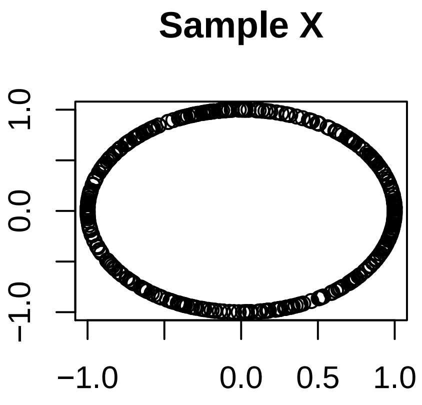


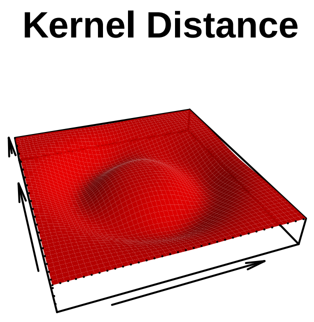
- The Kernel distance estimator, for each $y \in \mathbb{R}^d$, is defined as

$$\hat{\kappa}_h(y) = \sqrt{\frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} K_h(x_i, x_j) + K_h(y, y) - 2\frac{1}{n}\sum_{i=1}^{n} K_h(y, x_i)},$$

where $K_h(x, y) = \exp\left(\frac{-\|x-y\|_2^2}{2h^2}\right)$ is the Gaussian Kernel with smoothing parameter h.

```
###########################################################################
# kernel distance
###########################################################################
h <- 0.3
Kdist <- kernelDist(X = X, Grid = Grid, h = h)

par(mfrow = c(1,2))
plot(X, xlab = "", ylab = "", main = "Sample X")
persp(x = Xseq, y = Yseq,
      z = matrix(Kdist, nrow = length(Xseq), ncol = length(Yseq)),
      xlab = "", ylab = "", zlab = "", theta = -20, phi = 35, scale = FALSE,
      expand = 3, col = "red", border = NA, ltheta = 50, shade = 0.5,
      main = "Kernel Distance")
```





3.3. Bootstrap Confidence Bands

$(1-\alpha)$ confidence band can be computed for a function using the bootstrap algorithm, which we briefly describe using the kernel density estimator:

1. Given a sample $X = \{x_1, \ldots, x_n\}$, compute the kernel density estimator $\hat{p}_h$;
2. Draw $X^* = \{x_1^*, \ldots, x_n^*\}$ from $X = \{x_1, \ldots, x_n\}$ (with replacement), and compute $\theta^* = \sqrt{n}\|\hat{p}_h^*(x) - \hat{p}_h(x)\|_\infty$, where $\hat{p}_h^*$ is the density estimator computed using X^*;
3. Repeat the previous step B times to obtain $\theta_1^*, \ldots, \theta_B^*$;
4. Compute $q_\alpha = \inf\left\{q : \frac{1}{B}\sum_{j=1}^{B} I(\theta_j^* \geq q) \leq \alpha\right\}$;
5. The $(1-\alpha)$ confidence band for $\mathbb{E}[\hat{p}_h]$ is $\left[\hat{p}_h - \frac{q_\alpha}{\sqrt{n}}, \hat{p}_h + \frac{q_\alpha}{\sqrt{n}}\right]$.

Fasy, Lecci, Rinaldo, Wasserman, Balakrishnan, and Singh (2014) and Chazal, Fasy, Lecci, Michel, Rinaldo, and Wasserman (2014a) prove the validity of the bootstrap algorithm for kernel density estimators, distance to measure, and kernel distance, and use it in the framework of persistent homology.

`bootstrapBand` computes $(1-\alpha)$ bootstrap confidence band, with the option of parallelizing the algorithm (`parallel=TRUE`). The following code computes a 90% confidence band for $\mathbb{E}[\hat{p}_h]$.

```
###########################################################################
# bootstrap confidence band
###########################################################################
band <- bootstrapBand(X = X, FUN = kde, Grid = Grid, B = 100,
                      parallel = FALSE, alpha = 0.1, h = h)
print(band[["width"]])

##        90%
## 0.06426617
```

4. Persistent Homology

4.1. Persistent Homology Over a Grid

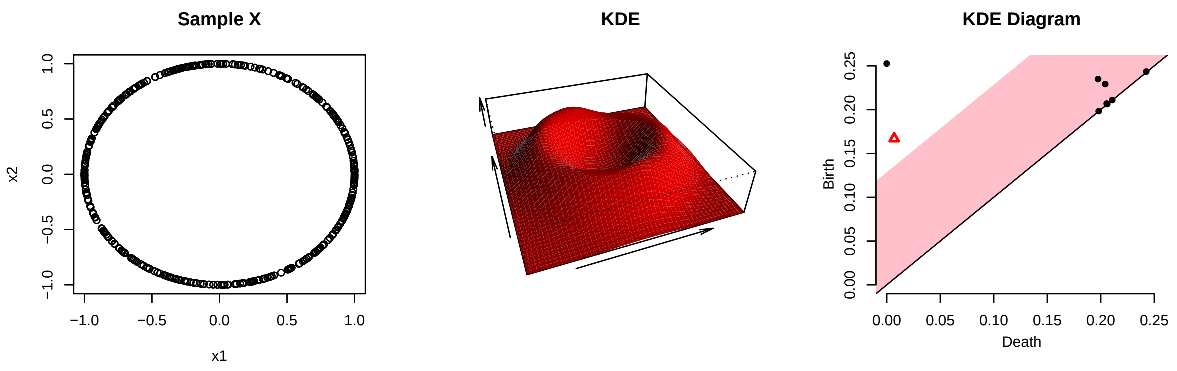
`gridDiag` function computes the persistent homology of sublevel (and superlevel) sets of the functions. The function `gridDiag` evaluates a given real valued function over a triangulated grid (in arbitrary dimension), constructs a filtration of simplices using the values of the function, and computes the persistent homology of the filtration. The user can choose to compute persistence diagrams using either the **Dionysus** library (`library = "Dionysus"`) or the **PHAT** library (`library = "PHAT"`) .

The following code computes the persistent homology of the superlevel sets (`sublevel = FALSE`) of the kernel density estimator (`FUN = kde`, `h = 0.3`) using the point cloud stored in the matrix `X` from the previous example. The other inputs are the features of the grid over which the `kde` is evaluated (`lim` and `by`), and a logical variable that indicates whether a progress bar should be printed (`printProgress`).

```
##########################################################################
# persistent homology of a function over a grid
##########################################################################
Diag <- gridDiag(X = X, FUN = kde, lim = cbind(Xlim, Ylim), by = by,
    sublevel = FALSE, library = "Dionysus", printProgress = FALSE, h = 0.3)
```

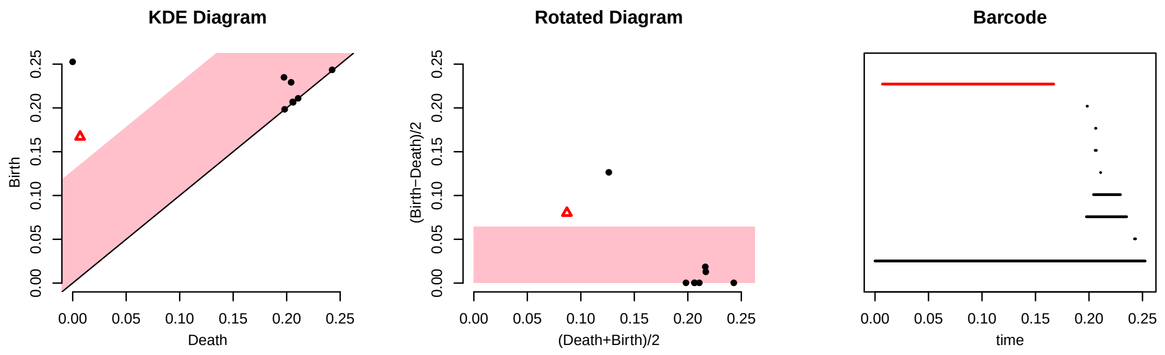
The function `plot` plots persistence diagram for objects of the class `"diagram"`. 8th line of the following command produces the third of the following plot. The option `band = 2 * band[["width"]]` produces a pink confidence band for the persistence diagram, using the confidence band constructed for the corresponding kernel density estimator in the previous section.

```
##########################################################################
# plotting persistence diagram
##########################################################################
par(mfrow = c(1,3))
plot(X, main = "Sample X")
persp(x = Xseq, y = Yseq,
      z = matrix(KDE, nrow = length(Xseq), ncol = length(Yseq)),
      xlab = "", ylab = "", zlab = "", theta = -20, phi = 35, scale = FALSE,
      expand = 3, col = "red", border = NA, ltheta = 50, shade = 0.9,
      main = "KDE")
plot(x = Diag[["diagram"]], band = 2 * band[["width"]], main = "KDE Diagram")
```



The function `plot` for the class `"diagram"` provide the options of rotating the diagram (`rotated = TRUE`), drawing the barcode in place of the diagram (`barcode = TRUE`).

```
##########################################################################
# other options for plotting persistence diagram
##########################################################################
par(mfrow = c(1,3))
plot(Diag[["diagram"]], band = 2 * band[["width"]], main = "KDE Diagram")
plot(Diag[["diagram"]], rotated = TRUE, band = band[["width"]],
     main = "Rotated Diagram")
plot(Diag[["diagram"]], barcode = TRUE, main = "Barcode")
```

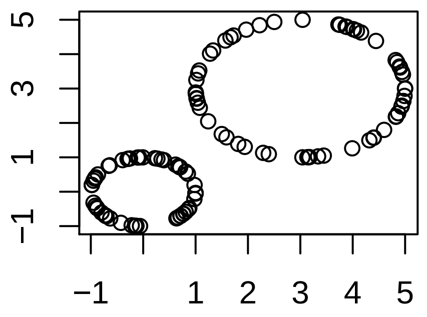


4.2. Rips Diagrams

The *Vietoris-Rips complex* $R(X, \varepsilon)$ consists of simplices with vertices in $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$ and diameter at most ε. The `ripsDiag` function computes the persistence diagram of the Rips filtration built on top of a point cloud. The user can choose to compute the Rips persistence diagram using either the C++ library **GUDHI** (`library = "GUDHI"`) , or **Dionysus** (`library = "Dionysus"`) .

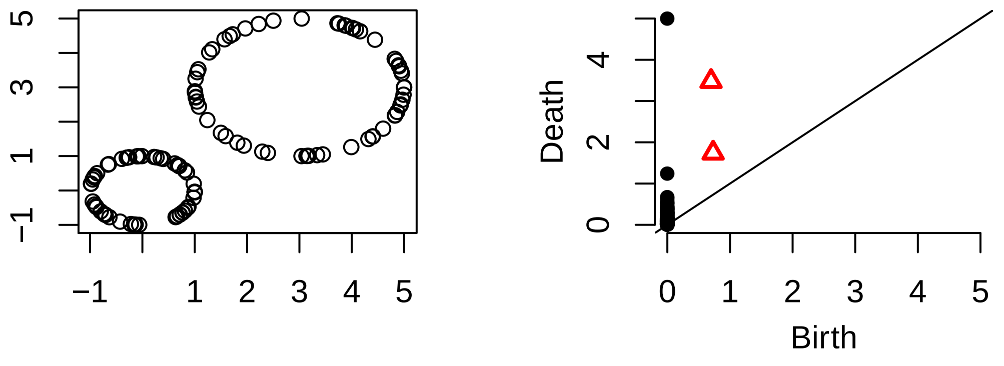
The following code generates 60 points from two circles, as in the following figure:

```
##########################################################################
# generating samples from two circles
##########################################################################
Circle1 <- circleUnif(n = 60)
Circle2 <- circleUnif(n = 60, r = 2) + 3
Circles <- rbind(Circle1, Circle2)
par(mfrow = c(1,1))
plot(Circles, xlab="", ylab="")
```



We specify the limit of the Rips filtration(`maxscale` = 5) and the max dimension(`maxdimension` = 1) of the homological features we are interested in (0 for components, 1 for loops, 2 for voids, etc.). Then we plot the data and the diagram.

```
##########################################################################
# Rips persistence diagram
##########################################################################
Diag <- ripsDiag(X = Circles, maxdimension = 1, maxscale = 5,
    library = "GUDHI", printProgress = FALSE)
par(mfrow=c(1,2))
plot(Circles, xlab="", ylab="")
plot(Diag[["diagram"]])
```



4.3. Bottleneck and Wasserstein Distances

Standard metrics for measuring the distance between two persistence diagrams are the bottleneck distance and the pth Wasserstein distance (Edelsbrunner and Harer 2010). The **TDA** package includes the functions `bottleneck` and `wasserstein`, which are R wrappers of the functions “bottleneck_distance” and “wasserstein_distance” of the C++ library **Dionysus**.

We generate two persistence diagrams of the Rips filtrations built on top of the two (separate) circles of the previous example, and we compute the bottleneck distance and the 2nd Wasserstein distance between the two diagrams. The option `dimension = 1` specifies that the distances between diagrams are computed using only one dimensional features (loops).

```
Diag1 <- ripsDiag(X = Circle1, maxdimension = 1, maxscale = 5)
Diag2 <- ripsDiag(X = Circle2, maxdimension = 1, maxscale = 5)
print(bottleneck(Diag1 = Diag1[["diagram"]], Diag2 = Diag2[["diagram"]],
    dimension = 1))

## [1] 1.389126

print(wasserstein(Diag1 = Diag1[["diagram"]], Diag2 = Diag2[["diagram"]],
    p = 2, dimension = 1))

## [1] 2.184218
```

4.4. Landscapes and Silhouettes

Persistence landscapes and silhouettes are real-valued functions that further summarize the information contained in a persistence diagram. They have been introduced and studied in Bubenik (2012), Chazal, Fasy, Lecci, Rinaldo, and Wasserman (2014c), and Chazal, Fasy, Lecci, Michel, Rinaldo, and Wasserman (2014b).

Landscape. The persistence landscape is a collection of continuous, piecewise linear functions $\lambda\colon \mathbb{Z}^+ \times \mathbb{R} \to \mathbb{R}$ that summarizes a persistence diagram. Consider the set of functions created by tenting each point $p = (x, y) = \left(\frac{b+d}{2}, \frac{d-b}{2}\right)$ representing a birth-death pair (b, d) in the persistence diagram D as follows:

$$\Lambda_p(t) = \begin{cases} t - x + y & t \in [x - y, x] \\ x + y - t & t \in (x, x + y] \\ 0 & \text{otherwise} \end{cases} = \begin{cases} t - b & t \in [b, \frac{b+d}{2}] \\ d - t & t \in (\frac{b+d}{2}, d] \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

We obtain an arrangement of piecewise linear curves by overlaying the graphs of the functions $\{\Lambda_p\}_p$; see Figure 1 (left). The persistence landscape of D is the collection of functions

$$\lambda(k, t) = \underset{p}{k\max}\, \Lambda_p(t), \quad t \in [0, T], k \in \mathbb{N}, \tag{2}$$

where $k$max is the kth largest value in the set. see Figure 1 (middle).

Silhouette. Consider a persistence diagram with N off diagonal points $\{(b_j, d_j)\}_{j=1}^N$. For every $0 < p < \infty$ we define the power-weighted silhouette

$$\phi^{(p)}(t) = \frac{\sum_{j=1}^N |d_j - b_j|^p \Lambda_j(t)}{\sum_{j=1}^N |d_j - b_j|^p}.$$

see Figure 1 (right).

`landscape` evaluates the landscape function over a one-dimensional grid of points `tseq`. In the following code, we use the rips persistence diagram in previous example to construct the corresponding landscape for one-dimensional features (`dimension = 1`). The option `KK = 1` specifies that we are interested in the 1st landscape function. `landscape` return a real valued vector, which can be simply plotted with `plot(tseq, Land, type = "l")`.

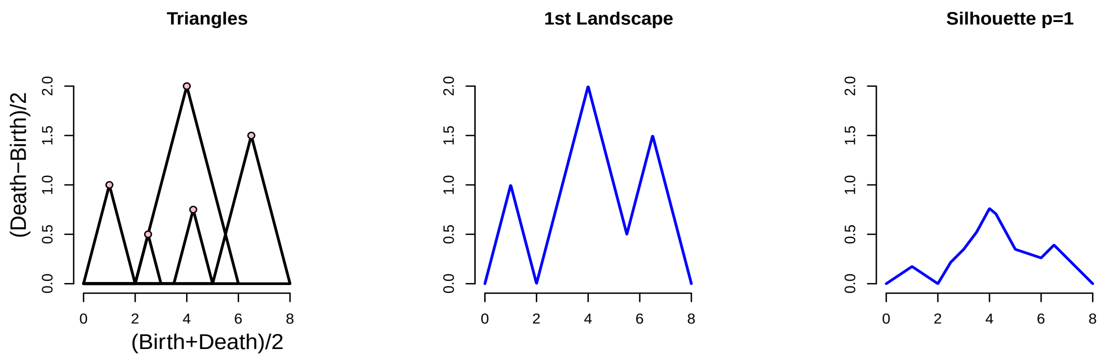
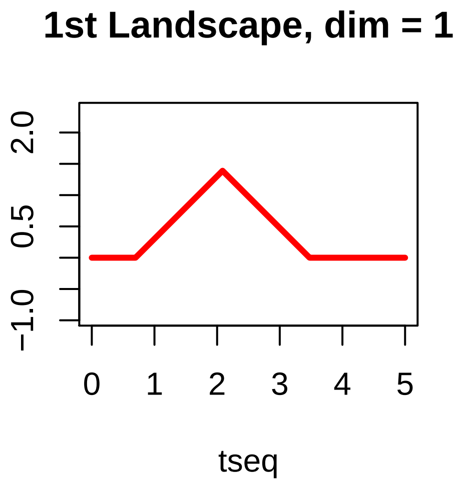


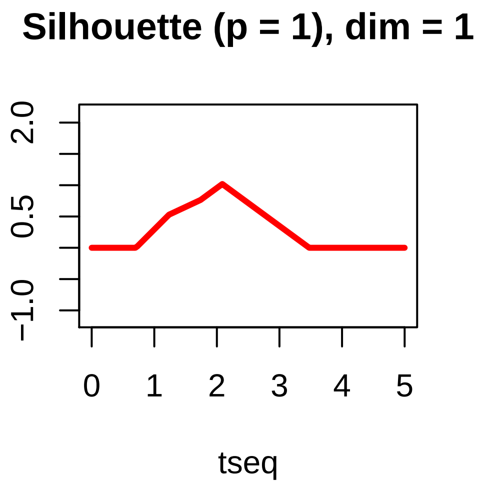
Figure 1: Left: we use the rotated axes to represent a persistence diagram D. A feature $(b, d) \in D$ is represented by the point $(\frac{b+d}{2}, \frac{d-b}{2})$ (pink). Middle: the blue curve is the landscape $\lambda(1, \cdot)$. Right: the blue curve is the silhouette $\phi^{(1)}(\cdot)$.

```
tseq <- seq(from = 0, to = 5, length = 1000)    #domain
Land <- landscape(Diag = Diag[["diagram"]], dimension = 1, KK = 1, tseq = tseq)
par(mfrow=c(1,1))
plot(tseq, Land, type = "l", main = "1st Landscape, dim = 1", ylab = "",
     asp = 1, col = "red", lwd = 3)
```



`silhouette` evaluates the silhouette function over a one-dimensional grid of points `tseq`. The silhoutte is constructed on the same rips persistence diagram for one-dimensional features (`dimension = 1`). The option `p=1` is the power of the weights in thesilhouette function. Evaluated silhoutte function can be simply plotted with `plot(tseq, Sil, type = "l")`.

```
tseq <- seq(from = 0, to = 5, length = 1000)    #domain
Sil <- silhouette(Diag = Diag[["diagram"]], p = 1,  dimension = 1, tseq = tseq)
par(mfrow=c(1,1))
plot(tseq, Sil, type = "l", main="Silhouette (p = 1), dim = 1", ylab = "",
     asp = 1, col = "red", lwd = 3)
```



References

Bauer U, Kerber M, Reininghaus J (2012). "PHAT, a software library for persistent homology." https://code.google.com/p/phat/.

Bubenik P (2012). "Statistical topological data analysis using persistence landscapes." *arXiv preprint arXiv:1207.6437*.

Chazal F, Cohen-Steiner D, Mérigot Q (2011). "Geometric inference for probability measures." *Foundations of Computational Mathematics*, **11**(6), 733–751.

Chazal F, Fasy BT, Lecci F, Michel B, Rinaldo A, Wasserman L (2014a). "Robust Topological Inference: Distance-To-a-Measure and Kernel Distance." *Technical Report*.

Chazal F, Fasy BT, Lecci F, Michel B, Rinaldo A, Wasserman L (2014b). "Subsampling Methods for Persistent Homology." *arXiv preprint arXiv:1406.1901*.

Chazal F, Fasy BT, Lecci F, Rinaldo A, Wasserman L (2014c). "Stochastic Convergence of Persistence Landscapes and Silhouettes." In *Annual Symposium on Computational Geometry*, pp. 474–483. ACM.

Edelsbrunner H, Harer J (2010). *Computational topology: an introduction*. American Mathematical Society.

Fasy BT, Lecci F, Rinaldo A, Wasserman L, Balakrishnan S, Singh A (2014). "Confidence Sets For Persistence Diagrams." *To appear in The Annals of Statistics*.

Maria C (2014). "GUDHI, Simplicial Complexes and Persistent Homology Packages." https://project.inria.fr/gudhi/software/.

Morozov D (2007). "Dionysus, a C++ library for computing persistent homology." http://www.mrzv.org/software/dionysus/.

Affiliation:
Firstname Lastname
Affiliation
Address, Country
E-mail: `name@address`
URL: `http://link/to/webpage/`