Tutorial on the R package TDA

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

Abstract

This tutorial gives an introduction to the R package TDA, which provides some tools for Topological Data Analysis. The salient topological features of data can be quantified with persistent homology. The R package TDA provide an R interface for the efficient algorithms of the C++ libraries GUDHI, Dionysus, and PHAT. Specifically, The R package TDA includes functions for computing the persistent homology of the Rips complex, alpha complex, and alpha shape complex, and a function for the persistent homology of sublevel sets (or superlevel sets) of arbitrary functions evaluated over a grid of points. The R package TDA also provides a function for computing the confidence band that determines the 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(https://project.inria.fr/gudhi/software/), Dionysus(http://www.mrzv.org/software/dionysus/), and PHAT(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. Installation

First, you should download R. R of version at least 3.1.0 is required:
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/

For Windows and Mac, 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")
}

## Loading required package: TDA
```

If you are using Linux, you should install R package TDA from the source. To do this, you need to install two libraries in advance: gmp (https://gmplib.org/) and mpfr (http://www.mpfr.org/). Installation of these packages may differ by your Linux distributions. Once those libraries are installed, you need to install four R packages: 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")
}

## Loading required package: FNN

if (!require(package = "igraph")) {
  install.packages(pkgs = "igraph")
}

## Loading required package: igraph
```

```r
## Attaching package: 'igraph'
## The following object is masked from 'package:FNN':
## knn
## The following objects are masked from 'package:stats':
## decompose, spectrum
## The following object is masked from 'package:base':
## union
```
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 \( r \).
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)
lim <- c(-1.7, 1.7)
by <- 0.05
margin <- seq(from = lim[1], to = lim[2], by = by)
Grid <- expand.grid(margin, margin)
```

- 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)
```

```r
par(mfrow = c(1,2))
plot(X, xlab = "", ylab = "", main = "Sample X", pch = 20)
persp(x = margin, y = margin,
     z = matrix(distance, nrow = length(margin), ncol = length(margin)),
     xlab = "", ylab = "", zlab = "", theta = -20, phi = 35, scale = FALSE,
     expand = 3, col = "red", border = NA, ltheta = 50, shade = 0.5,
     main = "Distance Function")```
The Gaussian Kernel Density Estimator (KDE), for each $y \in \mathbb{R}^d$, is defined as

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

where $h$ is a smoothing parameter.

```R
# 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", pch = 20)
persp(x = margin, y = margin,
     z = matrix(KDE, nrow = length(margin), ncol = length(margin)),
     xlab = "", ylab = "", zlab = "", theta = -20, phi = 35, scale = FALSE,
     expand = 3, col = "red", border = NA, ltheta = 50, shade = 0.5,
     main = "KDE")
```
Tutorial on the \textit{R} package \textit{TDA}

4. Persistent Homology

4.1. Persistent Homology Over a Grid

\texttt{gridDiag} function computes the persistent homology of sublevel (and superlevel) sets of the functions. The function \texttt{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 C++ library \texttt{GUDHI} (\texttt{library = "GUDHI"}), \texttt{Dionysus} (\texttt{library = "Dionysus"}), or \texttt{PHAT} (\texttt{library = "PHAT"}).

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

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

The function \texttt{plot} plots persistence diagram for objects of the class "\texttt{diagram}".

```r
#########################################################################
# plotting persistence diagram
#########################################################################
par(mfrow = c(1,3))
plot(X, main = "Sample X", pch = 20)
persp(x = margin, y = margin,
      z = matrix(KDE, nrow = length(margin), ncol = length(margin)),
      #...
4.2. Rips Persistent Homology

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 $\varepsilon$. In other words, a simplex $\sigma$ is included in the complex if each pair of vertices in $\sigma$ is at most $\varepsilon$ apart. The sequence of Rips complexes obtained by gradually increasing the radius $\varepsilon$ creates a filtration.

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 filtration using either the C++ library GUDHI or Dionysus. Then for computing the persistence diagram from the Rips filtration, the user can use either the C++ library GUDHI, Dionysus, or PHAT.

The following code computes the persistent homology of the Rips filtration using the point cloud stored in the matrix $X$ from the previous example, and the plot the data and the diagram.

```r
DiagRips <- ripsDiag(X = X, maxdimension = 1, maxscale = 0.5,
                     library = c("GUDHI", "Dionysus"), location = TRUE)
```

```r
par(mfrow = c(1,2))
plot(X, xlab = "", ylab = "", main = "Sample X", pch = 20)
plot(x = DiagRips[["diagram"]], main = "Rips Diagram")
```
4.3. Persistent Homology from filtration

Rather than computing persistence diagrams from built-in function, it is also possible to compute persistence diagrams from a user-defined filtration. A filtration consists of simplicial complex and the filtration values on each simplex. The functions `ripsDiag` has their counterparts for computing corresponding filtrations instead of persistence diagrams: namely, `ripsFiltration` corresponds to the Rips filtration built on top of a point cloud.

After specifying the limit of the Rips filtration and the max dimension of the homological features, the following code compute the Rips filtration using the point cloud $X$.

```r
FltRips <- ripsFiltration(X = X, maxdimension = 1, maxscale = 0.5, library = "GUDHI")
```

One way of defining a user-defined filtration is to build a filtration from a simplicial complex and function values on the vertices. The function `funFiltration` takes function values ($\text{FUNvalues}$) and simplicial complex ($\text{cmplx}$) as input, and build a filtration, where a filtration value on a simplex is defined as the maximum of function values on the vertices of the simplex.

In the following example, the function `funFiltration` construct a filtration from a Rips complex and the kernel density estimates on data points.

```r
h <- 0.3
KDEx <- kde(X = X, Grid = X, h = h)
FltFun <- funFiltration(FUNvalues = KDEx, cmplx = FltRips["cmplx"], sublevel = FALSE)
```

Once the filtration is computed, the function `filtrationDiag` computes the persistence diagram from the filtration. The user can choose to compute the persistence diagram using either the C++ library `GUDHI` or `Dionysus`.

```r
DiagFltFun <- filtrationDiag(filtration = FltFun, maxdimension = 1, library = "Dionysus", location = TRUE, printProgress = FALSE)
```

```r
par(mfrow = c(1,3))
plot(X, xlab = "", ylab = "", main = "Sample X", pch = 20)
plot(x = DiagGrid["diagram"], main = "KDE Diagram over grid")
plot(x = DiagFltFun["diagram"], diagLim = c(0, 0.27), main = "KDE Diagram over Rips filtration")
```
5. Statistical Inference on Persistent Homology

$(1 - \alpha)$ 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 $E[\hat{p}_h]$ is $\left[\hat{p}_h - \frac{q_{\alpha}}{\sqrt{n}}, \hat{p}_h + \frac{q_{\alpha}}{\sqrt{n}}\right]$.

`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 $E[\hat{p}_h]$.

```r
##########################################################################
# bootstrap confidence band for kde function
##########################################################################
bandKDE <- bootstrapBand(X = X, FUN = kde, Grid = Grid, B = 100,
                           parallel = FALSE, alpha = 0.1, h = h)
print(bandKDE[["width"]])
## 90%
## 0.06387487
```

Then such confidence band for $E[\hat{p}_h]$ can be used as the confidence band for the persistent homology.
Affiliation:
Firstname Lastname
Affiliation
Address, Country
E-mail: name@address
URL: http://link/to/webpage/