PFDA Package

Package Introduction

The main purpose of PFDA package is generating the private RKHS smoothing mean based on Mirshani et. al. [1]. the initial data records can be imported or simulated.

Existing dataset

For a existing data set which are put in columns of a matrix, first we need to choose a covariance operator and the penalty parameter. In the package, there are 5 different default kernels, "Exponential", "Matern $\nu = 3/2$", "Matern $\nu = 5/2$", "Gaussian" and "Sobolev" with range parameter ρwhich are defined in the following.

$$K_{Gau}(t,s) = \exp\left\{\frac{-|t-s|^2}{\rho}\right\}$$
$$K_{M5/2}(t,s) = (1 + \frac{\sqrt{5}|t-s|}{\rho} + \frac{5(t-s)^2}{3\rho^2})\exp\left\{\frac{-\sqrt{5}|t-s|}{\rho}\right\}$$
$$K_{M3/2}(t,s) = (1 + \frac{\sqrt{3}|t-s|}{\rho})\exp\left\{\frac{-\sqrt{3}|t-s|}{\rho}\right\}$$
$$K_{Exp}(t,s) = \exp\left\{\frac{-|t-s|}{\rho}\right\}$$
$$K_{Sob}(t,s) = \frac{\rho}{\sinh(\rho)}\cosh(1-\max(t,s))\cosh(\min(t,s))$$

So choosing a kernel and two range and penalty parameters are needed. In PFDA package, there exists a function which derives the optimum ϕ and ρ based on two different methods of Cross validation, *Regular* and "Irregualar". In *Regular Cross Validation* we fixed ϕ and then find out the ρ which gives the minimum 10-fold cross validation. We do not select optimum ϕ and ρ based on cross validation simultaneously because based on our observations, the minimum Cross Validation always be gotten from minimum ϕ. In *Irregular Cross Validation*, first we divide the data to 10 folds. For each fold, say $fold = i$, we calculate the pointwise mean $\hat{f}_{(fold=i)}$ and then with other folds we derive the private RKHS mean for 1000 replications which is called $\left\{\tilde{f}_{(j)(fold=i)}\right\}$ for $j = 1, \ldots, 1000$. If the mean of $\|\hat{f}_{(fold=i)} - \tilde{f}_{(j)(fold=i)}\|_{\mathbb{H}}$ for the whole replications is called $M_{(fold=i)}$, the Irregular C.V will be the mean of M_{fold} for $fold = 1, \ldots 10$. Changing ϕ and ρ on some different grids and finding out the optimum parameters which give the minimum Irregular C.V is the last part. To sum up, In *Regular C.V* just the original mean affects on but in *Irregular C.V* both the original and private mean influences on selecting the optimum parameters.

Simulating dataset

The first part is selecting a covariance function $K(t,s)$ and its corresponding range parameters ρ. We consider (λ_j, v_j) as eigenvalues and eigenfunctions of the kernels. The sample size

N and constant mean of the functions $\mu(t)$ should be selected to generate the samples based on *KL-expansion* (1) where $\{U_{ij}\}$ is iid uniformly distributed on $(-\tau, \tau)$, m is the number of positive eigenvalues, $u_j = v_j$ and $\{\gamma_j = j^{-p}\}$ which $p > 1$ is called *smoothing parameter* of $X_i(t)$ for $i = 1, \ldots, N$. The (γ_j, u_j) are like eigenvalues and eigenfunctions of kernels to generate $\{X_i\}$.

$$X_i(t) = \mu + \sum_{j=1}^{m} \sqrt{\gamma_j} U_{ij} v_j(t) \qquad t \in [0, 1] \tag{1}$$

Our aim is to release $\hat{\mu}$, a private RKHS smoothing mean function μ based on 2.

$$\hat{\mu} = \underset{M \in \mathbb{K}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \|X_i - M\|_{\mathbb{H}}^2 + \phi \|M\|_{\mathbb{K}}^2 \tag{2}$$

such that $\mathbb{K} = \left\{ h \in \mathbb{H} : \sum_{j=1}^{m} \frac{\langle h, v_j \rangle_{\mathbb{H}}^2}{\lambda_j} < \infty \right\}$ and $\langle M_1, M_2 \rangle_{\mathbb{K}} = \sum_{j=1}^{m} \frac{\langle M_1, v_j \rangle_{\mathbb{H}} \langle M_2, v_j \rangle_{\mathbb{H}}}{\lambda_j}$.
One can see $\|M\|_{\mathbb{K}}^2 = \sum_{j=1}^{m} \frac{\langle M, v_j \rangle_{\mathbb{H}}^2}{\lambda_j}$ and also generate $X_i(t) = \sum_{j=1}^{m} = \langle X_i, v_i(t) \rangle v_j(t) = \sum_{j=1}^{m} x_{ij} v_j(t)$ and $M(t) = \sum_{j=1}^{m} \langle M(t), v_j(t) \rangle v_j(t) = \sum_{j=1}^{m} M_j v_j(t)$. The $\hat{\mu}$ is actually a sample mean added by a penalty part.

PFDA package: details

In this part, we provide the details of PFDA package along with some examples.

To install the package, type the following commands in R after installing "devtools":

```
require(devtools)
install_github("ardeeshany/PFDA")
```

The PFDA package includes four functions:

- `MDP.RKHS`: generates Differential Private (DP) RKHS smoothing mean of a dataset.
- `PlotMDP`: plots RKHS smoothing mean and its DP version simultaneously.
- `CV`: finds the best values of penalty parameter and range parameter of RKHS smoothing mean based on cross validation.
- `MDP.Sim`: simulates a dataset and generates Differential Private (DP) RKHS smoothing mean of it.

In the following sections, we explain the details of each of these functions.

MDP.RKHS function

The `MDP.RKHS` function generates Differential Private (DP) RKHS smoothing mean of a dataset. The arguments of the function are:

- `grid`: provides a sequence of points associated with the set of data points on each curve. For example, if our data represents the height of different people at different ages, then the grid will be a sequence of ages. The default is to divide the interval $[0, 1]$ by the number of data points on each curve.

- `Data`: provides the dataset to the function and should be specified by the user. The provided data should be a matrix where each column is associated with a curve that we want to analyze and the elements of each column represent a sample of data points from the curve.

- `alpha`: provides a vector that contains the α parameter(s) of privacy. The default is `alpha` $= 1$.

- `beta`: provides a vector that contains the β parameter(s) of privacy. The default is `beta` $= 0.1$.

- `kernel`: provides the kernel to use for RKHS smoothing mean computation. If the sizes of `alpha` and `beta` are greater than 1, then we can provide a vector of kernels corresponding to each values of `alpha` and `beta`. There are 5 different options: "`Gau`" which represents Gaussian kernel, "`Exp`" which represents Exponential kernel, "`M3/2`" which represents Matern kernel with $\nu = \frac{3}{2}$, "`M5/2`" which represents Matern kernel with $\nu = \frac{5}{2}$, and "`Sob`" which represents Sobolev kernel. The default is "`Gau`".

- `phi`: provides the penalty parameter ϕ for RKHS smoothing mean computation. The default is `phi` $= 0.01$.

- `ro`: provides the range parameter ρ for the kernel. The default is `ro` $= 0.2$.

- `col.drop`: drops columns (or rows) related to missing data points. If there are some missing data in the data matrix, then setting `col.drop` $=$ `TRUE` removes the columns associated with the missing points from the data. If `col.drop` $=$ `FALSE`, then the rows associated with the missing points will be removed. The default is `col.drop` $=$ `TRUE`.

The `MDP.RKHS` function returns a list of 5 objects: `f.tilda` which includes the DP RKHS smoothing mean of the dataset, `delta` which includes the calculated noise coefficient δ for DP RKHS smoothing mean, `f` which includes the RKHS smoothing mean of the dataset, `grid` which includes the grid used by the function, and `X` which includes the data matrix after dropping the columns (or rows) associated with the missing data.

We illustrate how to use the `MDP.RKHS` function in the following example. In this example, we use the `DTI` data from the "refund" package. We set the arguments of the function to their default values but in general, we can only provide the data matrix and the rest of the arguments will be automatically set to their default values.

```
Data=t(DTI$cca)
set.seed(2016)
D=MDP.RKHS(grid= seq(0,1,length=dim(Data)[1]),Data = Data,
alpha = 1,beta = 0.1,kernel = "Gau",phi = 0.01,ro = 0.2,
col.drop = TRUE)
```

The output of the function is of the following form:

```
## List of 1
##  $ DP1:List of 5
##   ..$ f.tilda: num [1:93, 1] 0.461 0.466 0.471 0.476 0.48 ...
##   ..$ delta  : num 0.0637
##   ..$ f      : num [1:93] 0.466 0.472 0.477 0.482 0.487 ...
##   ..$ grid   : num [1:93] 0 0.0109 0.0217 0.0326 0.0435 ...
##   ..$ X      : num [1:93, 1:376] 0.491 0.517 0.536 0.555 0.593 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:93] "cca_1" "cca_2" "cca_3" "cca_4" ...
##   .. .. ..$ : chr [1:376] "1001_1" "1002_1" "1003_1" "1004_1" ...
```

As we have mentioned, `alpha` and `beta` are vectors. This means instead of one value of α parameter and one value of β parameter, we can provide several values for α parameter and the same number of values for β parameter and ask the function to find the DP RKHS smoothing means associated with these parameters. This is useful when we want to find the DP RKHS smoothing mean of different settings for α and β. For each of these settings, we can also provide different kernels which means we can provide a vector of kernels which has the same size as `alpha`. The output of the function includes a list associated with different settings and each list contains a list of objects `f.tilda`, `delta`, `f`, `grid`, and `X` as explained before.

The following is an example in which we use vectors of size 3 instead of single values for `alpha`, `beta`, and `kernel`:

```
set.seed(2016)
D=MDP.RKHS(Data = Data,alpha = c(0.5,1,5),beta = rep(0.1,3),
           kernel =c("Exp","M3/2","Gau"))
```

The output includes 3 lists where each list is associated with one of the 3 different settings:

```
## [1] "DP1" "DP2" "DP3"
```

In this example, each of "`DP1`", "`DP2`", and "`DP3`" includes a list of `f.tilda`, `delta`, `f`, `grid`, and `X` similar to the previous example.

plotMDP function

The `plotMDP` function plots RKHS smoothing mean and its DP version simultaneously. The arguments of the function are:

- `MDP`:provides an `MDP` object which is the output of `MDP.RKHS` function.
- `text.main`: provides the title of the plot.
- `legend.cex`: provides the relative size of the legend for each plot.
- `legend.loc`: provides the location of the legend in the plot.
- `seg.lin`: provides the length of the line to be used in the legend.
- `text.font`: provides the font size of the legend.
- `text.width`: provides the width of the legend.
- `legend.size`: provides the relative size of the text among all the plots.
- `xlab`: provides the title of x axis.
- `ylab`: provides the title of y axis.
- `extra_range`: provides extra range for y axis.
- `lty.p`: provides the type of line to be used for the PD RKHS smoothing mean curve.
- `lwd.p`: provides the width of line to be used for the PD RKHS smoothing mean curve.
- `par`: provides the arrangement of the plots if we have multiple plots.

The following example illustrates how we can use `plotMDP` function. The `MDP` object in this example is object that we have obtained from the previous example. In the previous example, we found the DP RKHS smoothing mean of 3 different vales of α and β. Thus, the result of `plotMDP` is 3 plots related to these 3 different settings. Figure (1) represents the plots.

```
plotMDP(MDP = D,text.main = rep("Diffusion Tensor Imaginf",3),
        legend.cex = rep(1.8,3),legend.loc = rep("topleft",3),
seg.lin = 0.75, text.font=2,text.width=0.15,legend.size=0.3,
xlab = "t",ylab = "cca",lty.p=3,lwd.p=3)
```

In these plots, the grey curves represent the data, the green curve represents the RKHS smoothing mean, and the red curve represents the DP RKHS smoothing mean.

CV function

The `CV` function finds the best values of penalty parameter and range parameter of RKHS smoothing mean based on cross validation. The arguments of the function are:

- `grid`: provides a grid. The details are similar to what was explained in `MDP.RKHS`
- `Data`: provides the dataset. The details are the same as what was explained in `MDP.RKHS`.

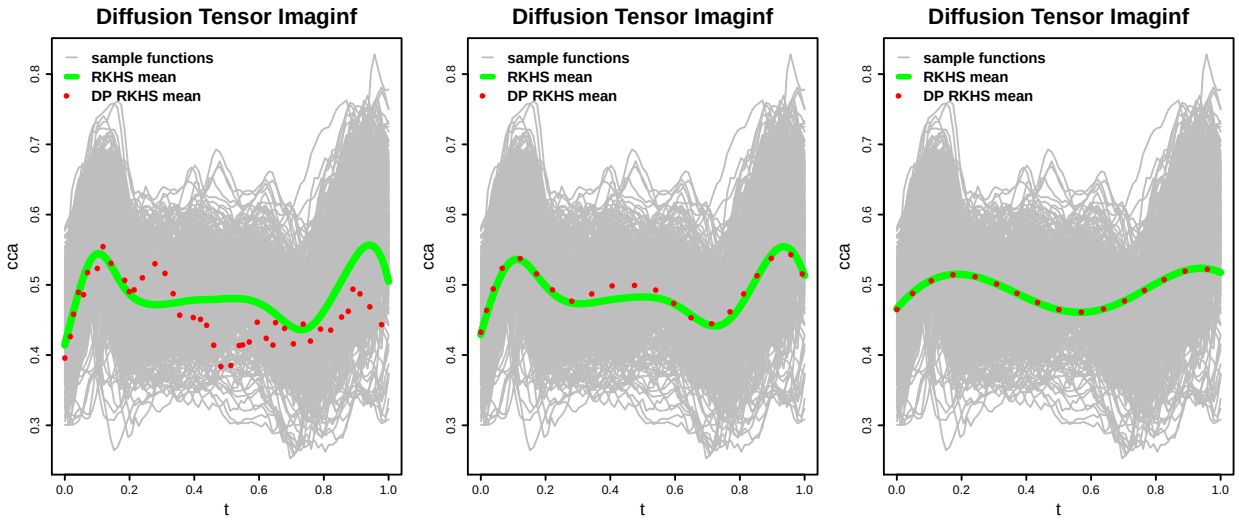

Figure 1: results of plotMDP function for 3 different settings

- `alpha`: provides the α parameter. The details are the same as what was explained in `MDP.RKHS`.
- `beta`: provides the β parameter. The details are the same as what was explained in `MDP.RKHS`.
- `kernel`: provides the kernel. The details are the same as what was explained in `MDP.RKHS`.
- `phi`: provides a vector of penalty parameters ϕ of RKHS smoothing mean to perform the cross validation.
- `ro`: provides a vector of range parameters ρ of the kernel to perform the cross validation.
- `fold`: provides number of folds for the cross validation. The default is `fold` = 10.
- `cv.penalty`: provides the method to perform the cross validation. There are two different methods: "`Regular`" and "`Irregular`". The default is "`Regular`".
- `Rep`: provides the number of iterations if `Rep` = "`Irregular`". The default is `Rep` = 1000.
- `print.cv`: prints the value of cross validation for all combinations and the best values of ϕ and ρ. The default is `print.cv` = "`TRUE`".
- `print.estimated.time`: prints the estimated time to perform the cross validation. The default is `print.estimated.time` = "`TRUE`".
- `col.drop`: drops columns (or rows) related to missing data points. The details are the same as what was explained in `MDP.RKHS`.

The following example illustrates how we can use `CV` function. In this example we provide a vector of size 3 for `phi` and a vector of size 2 for `ro`. The function compares the results of all possible combinations and returns the best value of ϕ and ρ parameters.

```
set.seed(2016)
D=CV(Data = Data,alpha = 1,beta = 0.1,kernel = "Gau",
phi = c(0.001,0.01,0.1),ro=c(0.02,0.2),fold = 10,
cv.penalty = "Regular",print.cv = "TRUE",
print.estimated.time = "TRUE",col.drop = TRUE)
```

The result of `CV` includes the best value of ϕ and ρ parameters. However, if we set `print.cv` = "`TRUE`", then the function prints a matrix such that the number of rows is equal to the size of `phi` and the number of columns is equal to the size of `ro`. Each element of this matrix represents the value of cross validation function corresponding to the associated ϕ and ρ parameters. The print also provides the best value of cross validation function among all the combinations and its corresponding values of ϕ and ρ. The following illustrates the printing result of the previous example:

```
## Estimation of remaining time is 20 seconds
##
##
##  CV matrix for each phi and ro is
##
##             [,1]       [,2]
## [1,] 0.06381979 0.06927894
## [2,] 0.06856146 0.07505608
## [3,] 0.16823003 0.11722081
## CV is gotten from row =  1 , col =  1
##
## CV is 0.06381979
##
## phi.cv = 0.001 , ro.cv = 0.02
##
```

MDP.Sim function

The `MDP.Sim` function simulates a dataset and generates DP RKHS smoothing mean of it. The arguments of the function are:

- `alpha`: provides a vector of the α parameters. The details are the same as what was explained in `MDP.RKHS`.
- `beta`: provides a vector of the β parameters. The details are the same as what was explained in `MDP.RKHS`.
- `kernel`: provides a vector of kernels.The details are the same as what was explained in `MDP.RKHS`.

- `phi`: provides the penalty parameter ϕ of RKHS smoothing mean.The details are the same as what was explained in `MDP.RKHS`.

- `ro`: provides the range parameter ρ of the kernel. The details are the same as what was explained in `MDP.RKHS`.

- `n`: provides the number of data points that should be generated for each curve. The generated data using this function has `n` rows. If `alpha` and `beta` are vectors, then `n` should also be a vector of the same size of `alpha` and `beta`. If `n` is a vector, then the function generates multiple datasets where the number of datasets is equal to the size of `n`. The default is `n` $= 100$.

- `N`: provides the number of curves to be generated for the data set. The generated data matrix from the function has `N` columns. If `alpha` and `beta` are vectors, then `N` should also be a vector of the same size of `alpha` and `beta`. The default is `N` $= 25$.

- `tau`: provides the τ parameter of the uniform distribution which is used in the generating process. If `alpha` and `beta` are vectors, then `tau` should also be a vector of the same size of `alpha` and `beta`. The default is `tau` $= 0.4$.

- `case`: provides the positive eighenvalues of the generated data. There are 3 different options: if `case` $= 0$, then the positive eigenvalues of the kernel (λ_j) are going to be used. If `case` $= 1$, then the positive values of $\lambda_j\gamma_j$ for $j = 1, ..., \texttt{n}$ are going to be used where $\gamma_j = j^{-p}$. If `case` $= 2$, then γ_j for $j = 1, ..., \texttt{n}$ are going to be used. The default is `case` $= 2$.

- `pow`: provides the p parameter of γ_j if `case` $= 1$ or `case` $= 2$. If `alpha` and `beta` are vectors, then `pow` should also be a vector of the same size of `alpha` and `beta`. The default is `pow` $= 4$.

- `mu`: provides the μ parameter used in the generating process. In the function setting, `mu` should be defined as a matrix. If `alpha` and `beta` are vectors, then `mu` should be a matrix such that the number of rows is equal to `n` and the number of columns is equal to the size of `alpha` and `beta`. The default is a matrix where each element is equal to $0.1\sin(2\pi t)$ where t represents the grid.

The following example illustrates how we can use the `MDP.Sim` function.

```
set.seed(2016)
D=MDP.Sim(alpha = 1,beta = 0.1,kernel = "Gau",phi = 0.01,
          ro = 0.2,n = 100,N =25,tau=0.4,pow=4)
```

The `MDP.Sim` function returns a `MDP` object which is a list of 5 objects: `f.tilda` which includes the DP RKHS smoothing mean of the dataset, `delta` which includes the calculated noise coefficient δ for DP RKHS smoothing mean, `f` which includes the RKHS smoothing mean of the dataset, `grid` which includes the grid used by the function, and `X` which includes the data matrix.

The result of the previous example is:

```
## List of 1
##  $ DP1:List of 5
##   ..$ f.tilda: num [1:100, 1] 0.0215 0.0245 0.0275 0.0305 0.0336 ...
##   ..$ delta  : num 0.112
##   ..$ f      : num [1:100] 0.0305 0.0341 0.0377 0.0413 0.0448 ...
##   ..$ grid   : num [1:100] 0 0.0101 0.0202 0.0303 0.0404 ...
##   ..$ X      : num [1:100, 1:25] -0.02758 -0.02143 -0.0159 -0.00975 -0.00333 ...
```

The following is an example where the `alpha` and `beta` are vectors of size 4.

```
set.seed(2016)
D=MDP.Sim(alpha = rep(1,4),beta = rep(0.1,4),
          kernel = c("Exp","M3/2","M5/2","Gau"),
n = rep(100,4),N =c(5,10,25,100),tau=rep(0.4,4),pow=rep(4,4))
```

In this example, `DP1` includes a 100×5 data matrix, `DP2` includes a 100×10 data matrix, `DP3` includes a 100×25 data matrix, `DP4` includes a 100×100 data matrix.

The output of the function includes a list corresponding to different settings and different datasets. Each list contains a list of objects `f.tilda`, `delta`, `f`, `grid`, and `X` as explained before.

```
## [1] "DP1" "DP2" "DP3" "DP4"
```

Since the result of `MDP.Sim` is in the same format as `MDP.RKHS`, we can use the function `plotMDP`. Figure (2) represents the results.

```
plotMDP(MDP = D,
   legend.loc = rep('topright',4),
   seg.lin=3,text.width = 0.4,legend.size = 0.5)
```

References

[1] A. Mirshani, M.Reimherr and S.Slavkovic (2016), Privacy for Functional Data (Has not been submitted)

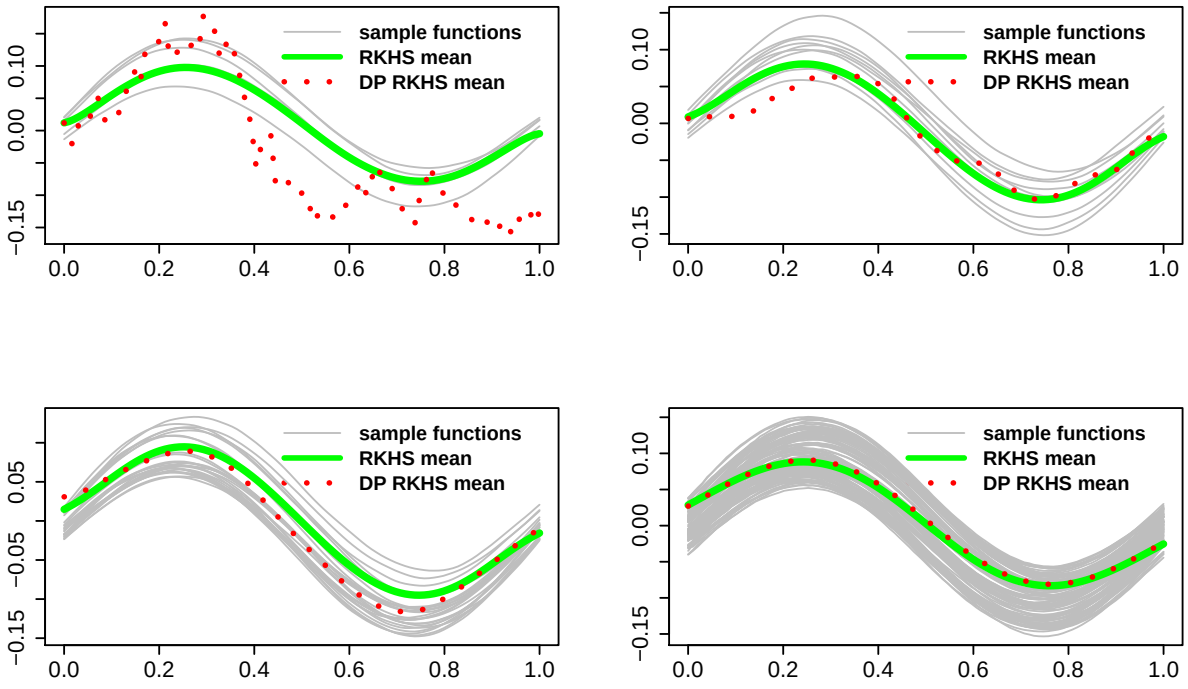

Figure 2: results of plotMDP function for 4 different simulated data