Fdakmapp: Optimization of the K-mean Alignment algorithm

APSC Project

Alessandro Zito, 817456

Academic Year 2016-2017
Abstract

This project is developed in the field of functional data analysis. In particular, it consists in the efficient implementation of an algorithm to perform clustering with alignment on functional dataset. This method was developed in (Sangalli, Secchi, Vantini and Vitelli) [1]. A previous R implementation of the algorithm exists but it’s not efficient enough to be comfortably used with large dataset. The fulfilled aim of the project is create an R package that performs simultaneously clustering and alignment and make it available to the research community. The source code is available at https://github.com/zitale/fdakmapp in the branch pacs. Further instructions on how to install the package and run some examples follow in the report. Furthermore a web application for live testing of the package is available at https://zitale.shinyapps.io/fdakmapp_webapp/.
Contents

Abstract I

1 Introduction 1
  1.1 Clustering for Functional Data .................. 1
  1.2 Report structure ............................ 2

2 The Algorithm 3
  2.1 A Mathematical Framework .................... 3
  2.2 The K-mean Alignment ........................ 5
     2.2.1 The optimization problem .................. 5
     2.2.2 The algorithm ............................ 6
  2.3 Template identification ....................... 7

3 Implementation 8
  3.1 R / C++ Interface ............................ 8
     3.1.1 Package Structure ....................... 9
     3.1.2 Public Functions ......................... 10
  3.2 Input parameters ............................ 10
  3.3 Output ....................................... 13
  3.4 Data Structure ................................ 14
  3.5 Algorithm workflow ........................... 15
     3.5.1 Loading Problem ......................... 16
     3.5.2 Execution ................................. 16
  3.6 Parallel version ............................. 17
     3.6.1 Other parallelization’s issues ............. 18
     3.6.2 Parallel performance ...................... 19
  3.7 Particular programming technique used ......... 19
  3.8 Profiling ..................................... 21

4 A live Web Application 22

5 Results and Conclusions 23
  5.1 Comparison with previous package ............... 25
  5.2 Conclusions .................................. 25
5.3 Future development ........................................ 26
5.4 Documentation ............................................ 26

6 FAQ ................................................................ 27
6.1 How to install the package? ......................... 27
6.2 Other examples .............................................. 28
6.3 How to add a new feature? ......................... 28

Bibliography ...................................................... 31
Chapter 1

Introduction

Clustering or unsupervised classification methods are an important topic in statistics, with many application in various fields. The aim of such techniques is to classify a sample of data into homogeneous groups, without having any a priori knowledge about the true underlying clustering structure. In particular, k-mean method for multivariate data is widely used in statistical classification problem for its flexibility. The algorithm divides a set of n data in k subgroups, iteratively alternating the search for the templates of the k clusters, that minimize the within-cluster variability, and the assignment of each of the n data to the cluster whose template is the nearest. K-medoid is a less computationally intensive modification of the k-mean algorithm. The main difference involves the way each cluster’s template is computed, with the k templates simply selected among the n data.

1.1 Clustering for Functional Data

In this project we considered the problem of clustering of functional data, especially curves. When working with clustering of curves, we could face a problem that is peculiar to functional data, known as data misalignment. If we use a standard algorithm the misalignment of the data acts as a confounding factor when trying to localize homogeneous groups and, for example, the above cited algorithms fail to give efficient results. This highlights the need for an algorithm that jointly manages clustering and misalignment. As shown in Figure 1.1 total variability can be decoupled in:

- **Phase Variability** due to the misalignment;
- **Amplitude Variability** due to the curve’s shape.
Figure 1.1: The left panel shows three curves varying only in amplitude. The right panel shows three curves varying only in phase.

Curve registration (or curve alignment) problem has been approached with many methods in literature. In this project we follow the line of research proposed by (Sangalli, Secchi, Vantini, Veneziani)[1] that define suitable dissimilarity indexes between curves and thus align the curves minimizing their dissimilarity (or their distance). The algorithm implemented in this work is able to jointly cluster and align a set of, unidimensional or multidimensional, functional data. The package is developed giving attention to generality and extensibility to help future researchers to adapt it to their own purposes.

1.2 Report structure

The report is organized as follows: In the Chapter 2 the algorithm is proposed after the introduction of a proper framework for clustering and alignment. In Chapter 3 the focus moves to the program work-flow and any relevant implementation details. In Chapter 4 a live web application is proposed for the live testing of the package functionalities. In Chapter 5 there are examples, conclusions and possible extensions. Finally a chapter with tutorials on how to install, extend and use the package is provided.
Chapter 2

The Algorithm

2.1 A Mathematical Framework

The goal of the alignment procedure is to eliminate or to minimize the phase variability such that the curves can be clustered according their amplitude variability. Let consider a set $C$ of curves $c(s) : \mathbb{R} \rightarrow \mathbb{R}^d$. Aligning $c_1 \in C$ and $c_2 \in C$ means find a warping function $h(s) : \mathbb{R} \rightarrow \mathbb{R}$ of the abscissa parameter $s$, such that the two curves $c_1 \circ h$ and $c_2$ are the most similar. It is thus necessary define:

- a dissimilarity index $\mathcal{E}(\cdot, \cdot) : C \times C \to \mathbb{R}$ that measures dissimilarity between 2 curves (it could be also a similarity or a distance);
- a class $W$ of warping functions $h$, such that $c \circ h \in C, \forall c \in C$ and $h \in W$, indicating the allowed transformation for the abscissa.

Aligning $c_1$ and $c_2$ means finding $h^* \in W$ that minimize (maximize in the case of similarity) $\mathcal{E}(c_1 \circ h, c_2)$. The proposed algorithm decouples phase and amplitude variability without loss of information because the first is captured by the warping functions $h$ while the latter remains between the curves.

The meaning of phase and amplitude variability is defined by the choice of the couple $(\mathcal{E}, W)$. A complete theoretical approach can be found in (Vantini)[3]. We only list the minimal requirements that the couple $(\mathcal{E}, W)$ must satisfy:

1. $\mathcal{E}$ is bounded from the below, with minimum value 0. Moreover $\mathcal{E}$ must be reflexive, symmetric and transitive.

2. The class of warping functions $W$ is a convex vector space and has a group structure with respect to function composition $\circ$.

3. The couple $(\mathcal{E}, W)$ functions is consistent in the sense that, if two functions $f_1$ and $f_2$ are simultaneously warped along the same warping
Figure 2.1: Example of decoupling without loss of information.

function \( h \in \mathcal{H} \), their dissimilarity does not change

\[
\mathcal{E}(f_1 \circ h, f_2 \circ h) = \mathcal{E}(f_1, f_2), \forall h \in \mathcal{H}
\]

This guarantees that it is not possible to obtain a fictitious decrement of dissimilarity between two curves \( f_1 \) and \( f_2 \) by simply warping them simultaneously to \( f_1 \circ h \) and \( f_2 \circ h \). This property is called \textit{W-invariance}.

From the previous follows that for all \( h_1 \) and \( h_2 \in \mathcal{H} \),

\[
\mathcal{E}(f_1 \circ h_1, f_2 \circ h_2) = \mathcal{E}(f_1 \circ h_1 \circ h_2^{-1}, f_2) = \mathcal{E}(f_1, f_2 \circ h_2 \circ h_1^{-1})
\]

This means that a change in dissimilarity between \( f_1 \) and \( f_2 \) obtained by warping simultaneously \( f_1 \) and \( f_2 \) can be obtained by warping only \( f_1 \) or \( f_2 \).

For a complete mathematical introduction to the choice of a consistent couple \((\mathcal{E}, W)\) see (Ferraty, Vieu)[4]. Table 2.2 reports some possible choices.
2.2 The K-mean Alignment

Given the problem of clustering and aligning a set of N curves \( \{c_1, ..., c_N\} \) with respect to a set of k template curves \( \varphi = \{\varphi_1, ..., \varphi_k\} \) (with \( \{c_1, ..., c_N\} \subset C \) and \( \varphi \subset C' \)), we should define for each template curve the domain of attraction

\[
\Delta_j(\varphi) = \left\{ c \in C : \inf_{h \in W} \mathcal{E}(\varphi_j, c \circ h) \leq \inf_{h \in W} \mathcal{E}(\varphi_r, c \circ h), \forall r \neq j \right\}
\]  

Furthermore, we define the labeling function:

\[
\lambda(\varphi, c) = \min \left\{ r : c \in \Delta_r(\varphi) \right\}
\]

\( \lambda(\varphi, c) = j \) means that the dissimilarity index obtained by aligning \( c \) to \( \varphi_j \) is at least as small as the index obtained by aligning \( c \) to any other template \( \varphi_r \), with \( r \neq j \). Therefore the curve \( c \) should be aligned to the template \( \varphi_{\lambda(\varphi, c)} \) and assigned to cluster \( \lambda(\varphi, c) \). The templates could be fixed but we are interested in the more complex case of unknown templates.

2.2.1 The optimization problem

To cluster and align the set of N curves \( c_1, ..., c_N \) with respect to K unknown templates we should first solve the following optimization problem.

Find \( \varphi = \{\varphi_1, ..., \varphi_k\} \subset C \) and \( h = \{h_1, ..., h_k\} \subset W \) such that
\[
\frac{1}{N} \sum_{i=1}^{N} \mathcal{E}(\varphi_{\lambda(c_i)}, c_i \circ h_i) \leq \frac{1}{N} \sum_{i=1}^{N} \mathcal{E}(\psi_{\lambda(c_i)}, c_i \circ g_i), \tag{3}
\]

\[
\forall \, \varphi = \{\varphi_1, ..., \varphi_k\} \subset C, \ g = \{g_1, ..., g_k\} \subset W
\]

then assign \(c_i\) to the cluster \(\lambda(\varphi, c_i)\) and warp \(c_i\) along \(h_i\).

The problem (3) describes a search for both the set of optimal \(k\) templates, and for the set of optimal \(N\) warping functions.

### 2.2.2 The algorithm

The non-linear optimization problem (3) is not analytically solvable then (Sangalli, Secchi, Vantini, Vitelli)\cite{1} propose an iterative procedure to find an approximate solution. Let \(\varphi^{[q-1]} = \{\varphi_1^{[q-1]}, ..., \varphi_k^{[q-1]}\}\) the set of templates after iteration \(q-1\), and \(\{c_1^{[q-1]}, ..., c_N^{[q-1]}\}\) the \(N\) curves aligned and clustered to \(\varphi^{[q-1]}\). At the \(q\)th iteration the algorithm performs the following steps:

- **Template identification step.** For \(j = 1, ..., k\), the template of the \(j\)th cluster \(\varphi_j^{[q]}\) is estimated using all curves assigned to cluster \(j\) at the previous iteration. The template \(\varphi_j^{[q]}\) should be the curve \(\varphi \in C\) that minimizes the within-cluster total dissimilarity

  \[
  \sum_{i: \lambda(\varphi^{[q-1]}, c_i^{[q-1]}) = j} \mathcal{E}(\varphi, c_i^{[q-1]}) \tag{4}
  \]

- **Assignment and alignment step.** The curves \(c_1^{[q-1]}, ..., c_N^{[q-1]}\) are clustered and aligned to the templates \(\varphi^{[q]} = \{\varphi_1^{[q]}, ..., \varphi_k^{[q]}\}\). In detail, once for \(i = 1, ..., N\) the \(i\)-th curve \(c_i^{[q-1]}\) is aligned to \(\varphi_{\lambda(\varphi^{[q]}, c_i^{[q-1]})}\) then curve \(\tilde{c}_i^{[q]} = c_i^{[q-1]} \circ h_i^{[q]}\) is assigned to cluster \(\lambda(\varphi^{[q]}, \tilde{c}_i^{[q]})\).

- **Normalization step.** At the end of each iteration we perform a normalization step, namely for \(j = 1, ..., k\) all the \(N_j^{[q]}\) curves \(\tilde{c}_i^{[q]}\) assigned to cluster \(j\) are warped by the warping function \((\tilde{h}_j^{[q]})^{-1}\), where

  \[
  \tilde{h}_j^{[q]} = \frac{1}{N_j^{[q]}} \sum_{i: \lambda(\varphi^{[q]}, c_i^{[q]}) = j} h_i^{[q]}, \tag{5}
  \]

obtaining \(c_i^{[q]} = c_i^{[q-1]} \circ h_i^{[q]} \circ (\tilde{h}_j^{[q]})^{-1}\). In this way the average warping applied to curves assigned to any cluster \(j\) is the identity transformation \(h(s) = s\). The normalization is thus used to select, among all candidates solution to the optimization problem, the one that leaves the
average location of the cluster unchanged, thus avoiding drift apart of clusters. This step preserves the clustering structure previously computed because $\lambda(\varphi_{[q]}, \tilde{c}_{i}^{[q]}) = \lambda(\varphi_{[q]}, c_{i}^{[q]})$.

The algorithm is initialized with a set of initial templates $\varphi_{[0]} = \{ \varphi_{1}^{[0]}, ..., \varphi_{k}^{[0]} \} \subset C$ and stopped when the decrements of the dissimilarity indexes are all lower than a fixed threshold or there are no change between the clusters.

2.3 Template identification

From a theoretical and computational point of view the template identification step is more difficult to be performed compared with the others. Practically, the identification of the template $\varphi_{j}^{[q]}$ as the curve $\varphi \in C$ that minimizes the total dissimilarity (4) cannot be easily computed. In the project two main methods are proposed:

- **Mean identification** The algorithm obtained using this center method is named *K-Mean Alignment*. In this case the representatives functions are searched in the whole functional space $C$ and are called Frechet templates.

- **Medoid identification** This method constitutes a direct approximation of the maximization problem (4) but the set over the optimization is carried out is restricted. The template $\varphi_{j}^{[q]}$ is the curve that optimize the total dissimilarity (4) among all curves assigned to cluster $j$ at the iteration $[q-1]$ and the representatives functions are called Karcher templates.

$$\varphi_{j}^{[q]} = \arg \min_{c_{i}^{[q-1]}, \lambda(\varphi_{[q-1]}, c_{i}^{[q-1]}), k: \lambda(\varphi_{[q-1]}, c_{k}^{[q-1]}) = j} \sum_{k: \lambda(\varphi_{[q-1]}, c_{k}^{[q-1]}) = j} \mathcal{E}(c_{i}^{[q-1]}, c_{k}^{[q-1]})$$

The algorithm obtained using this specification is called *k-medoid alignment*. 
Chapter 3

Implementation

In the implementation of the R package able to cluster and align multidimensional functional data, we can identify four different core issues that we need to solve:

- how to interface R with C++;
- which data structures to use;
- how to solve the optimization problem;
- where to parallelize the code;

In this section the implementation solutions chosen are explained. Before entering in the details of the code, we specify that the leading idea of the work is to find a trade-off between extensibility and performance maintaining the clarity such that the package can be furtherly extended and customized by an R user. In Chapter 6 there are instructions and examples on how add new features.

3.1 R / C++ Interface

The interface used to let R communicate with the C++ source code, is built using the Rcpp package (through .Call function) described in (Dirk, Romain)[5] and (Dirk)[6].

It is a powerful, yet quite simple, way to use pre-compiled code in R. It provides a consistent C++ class hierarchy that maps various types of R objects (vectors, matrices, functions, environments, . . . ) to dedicated C++ classes. Object interchange between R and C++ is managed by simple, flexible and extensible concepts which include broad support for C++ Standard Template Library idioms and for other useful libraries as Armadillo. C++ code can both be compiled, linked and loaded on the fly, or added via packages. Rcpp substantially lowers the barrier for programmers wanting to
combine C++ code with R. Once the R user interface is ready, the following step is the writing of the source code that will be exported imported in R.

3.1.1 Package Structure

The package is organized as follow:

- **R directory** contains R function that if exported will be usable by the final R user.

- **src directory** contains the source code (organized in .h and .cpp). The c++ exported functions can be used by the R functions in the R folder.

- **data directory** contains an .RData file that provides 2 dataset (simulated30 and aneurisk65) used in examples and tests.

- **man directory** contains documentation files (.Rd). These files are automatically generated using Roxygen2.

- **tests directory** contains tests for the R functions useful during the phase of development.

- The DESCRIPTION file contains basic information about the package such as "Author", "Maintainer" and "License". It also provides useful informations for the installation phase such as "dependencies" and other package that should be imported before the installation "Import".

- The NAMESPACE file contains namespace directives to manage the functions export to the R environment.

This projects (with multiple source C++ files) includes in the src directory a file named Makevars, where all the flags needed for compiling and linking are specified. Since the ultimate goal is to distribute the package
to other users, to make the package portable an analogous file for Windows users, called Makevars.win is included, as suggested in (CRAN)[7].

### 3.1.2 Public Functions

The functions available for R users are:

- **kmap** the main function of the package that allows to perform jointly clustering and alignment.

- **kmap_show_results** a function that prints the result of the previous function.

The `kmap` function stored in R directory and exported through the NAMESPACE directives calls directly the source code through the `.Call()` interface.

```r
kmap <- function(x, y, seeds, n_clust, warping_method, center_method, similarity_method, optim_method, warping_opt, center_opt, out_opt, fence, check_total_similarity, show_iter, comp_original_center, par_opt) {
  ...
  out <- .Call('_fdakmapp_kmap', PACKAGE = 'fdakmapp', x, y, nseeds, n_clust, warping_method, center_method, similarity_method, optim_method, warping_opt, center_opt, out_opt, fence, check_total_similarity, show_iter, comp_original_center, par_opt)
  ...
  out
}
```

### 3.2 Input parameters

In this section the inputs parameters are explained. Further material can be found in the R documentation.

- **x** and **y** are the input datasets. x is the abscissa and y are the values of the unidimensional or multidimensional functions.

- **seeds** are the initial centers that need to be specified by the users. If NULL random templates are chosen.

- **n_clust** is the number cluster that the algorithm will try to find.
• **warping method** is a string that should define which kind of warping transformation $h$ has to be applied. The warping can be chosen between: ‘affine’, ‘dilation’, ‘shift’, ‘noalign’. The **warping opt** is a numeric vector that must be chosen coherently according to Table 3.4.

![WarpingFunction inheritance diagram.](image)

- **center method** is a string that defines how to find the centers. The possible choice are: ‘mean’, ‘medoid’ and ‘pseudomedoid’. The **center opt** is a numeric vector that must be chosen coherently according to Table 3.4.

![CenterMethod inheritance diagram.](image)

<table>
<thead>
<tr>
<th>warping</th>
<th>option</th>
<th>center</th>
<th>option</th>
</tr>
</thead>
<tbody>
<tr>
<td>affine</td>
<td>(max_dilation, max_shift)</td>
<td>mean</td>
<td>(span, delta)</td>
</tr>
<tr>
<td>dilation</td>
<td>(max_dilation)</td>
<td>medoid</td>
<td>()</td>
</tr>
<tr>
<td>shift</td>
<td>(max_shift)</td>
<td>pseudomedoid</td>
<td>()</td>
</tr>
<tr>
<td>noalign</td>
<td>()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.4: The tables indicate the options to specify once a warping functions (on the left) and a center method (on the right) are chosen.
• **similarity_method** is a string that should define which kind of comparison between function has to be used. In reality only dissimilarity and distances are implemented because the algorithm solve a minimization problem. The available methods are 'pearson' and 'l2'.

![Dissimilarity inheritance diagram](image)

Figure 3.5: Dissimilarity inheritance diagram.

• **optim_method** is a string that allows to choose the optimizer to use. Actually only one is provided from Dlib library [8]: 'bobyqa'.

![OptimizerMethod inheritance diagram](image)

Figure 3.6: OptimizerMethod inheritance diagram.

• **out_opt** is a numeric vector with 3 options to set:
  - **n_out** is the size of the output’s grids.
  - **tolerance** is a stop condition. The algorithm stop if the dissimilarity has decreased less than tolerance for each function.
  - **max_iteration** maximum number of iterations allowed.
• **fence** is a boolean that if true activates an additional iterative algorithm. After each iteration the fence algorithm check if there are outliers warping parameters. Then, the functions with outliers parameters are re-warped with more stringent constrains. The algorithm is iterative so it is repeated until when the outliers disappear.

• **check_total_similarity** is a boolean that activates a check on the total similarity after each iteration. The algorithm stop if the total similarity doesn’t improve.

• **show_iter** is a boolean that activates print to screen.

• **comp_original_center** is an option that if true compute the original center between the whole dataset. It can be useful to have a benchmark to compare results after the clustering with alignment but can be very computationally expensive.

• **par_opt** is a numeric vector with 2 options to set:
  
  – **n_threads** is the number of threads that can be used during the execution.

  – **parallel_type** is an option that can be 0 or 1. It indicate which kind of parallelization has to be used during the computation of new centers. 0 is the trivial parallelization in which each tread compute the center of a cluster. It is not efficient especially if the number of cluster is lower than the number of available threads or if one cluster is bigger than others. 1 is the more efficient option that use all the available threads to compute the new centers.

### 3.3 Output

The output of **kmap** is a list with 22 elements that in part repeat the inputs. The most important elements are:

• **iterations** The number of iterations done by the algorithm.

• **x.center.orig** If **comp_original_cent** is true is the vector abscissa of the center. Otherwise is NULL.

• **y.center.orig** If **comp_original_cent** is true is the matrix with the values of the center. Otherwise is NULL.

• **similarity.origin** If **comp_original_cent** is true is a vector with the similarities between the original center and all the other functions. Otherwise is NULL.
• **x.final** A matrix with the abscissa of each function after the alignment procedure.

• **n.clust.final** The number of clusters after the execution. Can be different from the input n.clust because the case in which a cluster is found empty is handled reducing the the groups from the next iteration.

• **x.centers.final** A matrix with the abscissas of the final centers.

• **y.centers.final** The values of the final centers.

• **templates_vec** The values of the centers at each iteration.

• **x.out_vec** The abscissas of the centers at each iteration.

• **labels** The labels that indicate at which cluster an observation belongs.

• **similarity.final** The dissimilarity or distance of each curve with its own center.

• **parameters.list** The list of the warping parameters at each iteration.

• **parameters** The total warping parameters.

• **timer** The timer that reports execution time divided by step.

### 3.4 Data Structure

A first version of the algorithm was implemented with the native Rcpp structures, as *NumericVector*, *NumericMatrix* and *ListOf<NumericMatrix>*. The built-in structures gave two problems:

- They are not **multi-threads safe**.
- There was not an intuitive structure to handle **3D array** passed from R.

These problems have been faced using a high quality linear algebra library for the C++, called **Armadillo**. It is available under a permissive license, useful for both open-source and proprietary software so the library is integrated using RcppArmadillo [9]. The Armadillo’s documentation is available at [http://arma.sourceforge.net/](http://arma.sourceforge.net/). This library fits perfectly the needs of the project aiming towards a good balance between speed and ease of use. For example, to handle the 3D array from R an efficient class is provided, called **cube**. Our set of data is functional so each observation
(each row i) is composed by several point (each column j) and could have many dimensions (each slice k).

![Cube data structure](image)

**Figure 3.7: Cube data structure.**

### 3.5 Algorithm workflow

In the c++ code a class `kma_model` handles:

- check-in of the inputs.
- loading of the problem.
- algorithm execution.

![Function's workflow](image)

**Figure 3.8: Function’s workflow.**
3.5.1 Loading Problem

In kma_model constructor, four factories are declared respectively for dissimilarity, warping, center method and optimizer. After each declaration all the available options are registered. Once the check-in is done, from the input strings using the factories we obtain the pointers to the objects that will be used in execute.cpp.

3.5.2 Execution

After the problem loading the algorithm proposed in the previous chapter can be executed. The code in execute.cpp doesn’t depend on the inputs so it doesn’t need to be modified if a new feature is implemented. It can be divided in the three following steps executed iteratively.

New template computation

To compute a new template for each cluster two main options are available:

- **mean** In this case the template is estimated by a loess, with Gaussian kernel and appropriate smoothness parameter, of the curves assigned to a cluster at the previous iteration. However the fact that the template is no longer the function that maximizes (4) but is estimated by loess of the curves assigned to the cluster could cause a distortion of the algorithm. Moreover, the loess estimation is sensitive to the presence of anomalous data.

- **medoid** In this case the template is chosen as the function in the cluster that minimizes the within variability. It is less sensitive to the presence of anomalous data and computationally simpler.

Thanks to these alternative specification we will have the possibility of highlight the robustness of our alignment and clustering procedure when different algorithm specification are considered. Moreover, the comparison of the clustering results obtained with the two alternative algorithm versions, *k-mean* and *k-medoid*, could help to detect accidental anomalous data.

Optimization problem

In this step each functions has to be compared with each one of the templates computed at the previous step optimizing the warping parameters. The optimization is carried out using an external library called Dlib [8]. It is worth noting that most of Dlib is a small “header-only” library so it is directly included in the package to not complicate the installation process.
Normalization

It’s an easy task to be performed that uses the information computed at the previous steps.

3.6 Parallel version

Apparently the clustering with alignment algorithm is trivial to parallelize because each thread can work with a cluster but the trivial parallelization is not efficient so a more complex parallel version is provided (as saw par_opt allows to choose the parallel version).

The multi-threads version of the algorithm is implemented using OpenMP. Each omp directive is protected against compilers that don’t support it by guarding header, as in the case of the header inclusion.

```c
#define _OPENMP
#include <omp.h>
#endif
```

Given the above code, we have effectively provided protection from the compiler throwing an error due to OpenMP not being available on the system. In the event that the system does not have OpenMP, the process will be executed serially. The algorithm’s steps runnable in parallel (because the most computationally expensive) are optimization and computation of new center.

Parallel optimization

During the optimization step each function has to be compared with all the templates and being the comparisons mutually independent the observation are equally distributed among the threads. The obtained parallelization is efficient because the threads have homogeneous groups to work on.

Parallel new centers computation

During the computation of new center is possible parallelize at 2 different levels:

- **parrallel_type = 0** is the trivial parallelization in which each tread compute the center of a cluster. It is not efficient especially if the number of cluster is lower than the number of available threads or if one cluster is bigger than others.

- **parrallel_type = 1** is the more efficient option that use all the available threads to compute each new center. For the moment this option is only available with `medoid` center method.
3.6.1 Other parallelization’s issues

To make the code compatible with the openMP parallelization have been introduced:

- In dissimilarity.h the struct `grid` has been introduced to make possible the parallelization without modify the code. This element is strictly linked with the dissimilarity object used during the algorithm execution. It represents the approximation of the input functions on a common grid where is possible to compute the dissimilarity. During the parallel execution the dissimilarity object is used simultaneously by many threads so the grid attributes can’t be inside the dissimilarity object but has to be an external element different in each threads.

- In warping.h the struct `wset` has been introduced for analogous reasons. In particular during the optimization step the object warping is used by many threads on different functions that have to be specified externally.

In warping.h:

```cpp
1   /// Warping setting
2   struct warping_set
3   {
4       rowvec xf;
5       rowvec xg;
6       mat yf;
7       mat yg;
8   };
```
3.6.2 Parallel performance

In this section the performances of the package in parallel are analyzed. The algorithm is run on the MOX clusters for parallel applications (HPC), in particular on 1 node with 20 cpu Intel Xeon E5-4610v2 @2.30GHz of Gigat where obviously OpenMP is supported. Fixed the seeds, for each number of threads the execution is repeated 10 times to reduce the impact of uncontrollable processes. The center method used is medoid and the number of cluster is 5. In Figure 3.10 can be appreciated the difference in performance of parallel versions 0 (left) and 1 (right). As expected the parallel version 1 is more efficient than 0, and the speed-up is shown in Table 3.11.

![Figure 3.10: Parallel performances k-medoids parallel version 0 (left). Parallel performances k-medoids parallel version 1 (right).](image)

<table>
<thead>
<tr>
<th>num. of threads</th>
<th>speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.63</td>
</tr>
<tr>
<td>5</td>
<td>3.58</td>
</tr>
<tr>
<td>10</td>
<td>5.74</td>
</tr>
<tr>
<td>15</td>
<td>8.13</td>
</tr>
<tr>
<td>20</td>
<td>9.30</td>
</tr>
<tr>
<td>30</td>
<td>10.88</td>
</tr>
</tbody>
</table>

Figure 3.11: Speed-up version 1.

3.7 Particular programming technique used

In this section there are some programming technique that were very useful during the development of the package.
Factories

The use of factories helps to have a clear and easily extensible code. First of all they allow to have a file execution.cpp independent by input arguments so we can add new features without change 1 line of the file. To do so warping_method, center_method, optim_method and similarity_method are handled in execution.cpp file using pointer to the base class. It makes the code also easy to extend because if a new feature is added it needs only to be integrated in the registration class process done in the kma_model constructor without modify any other part of the package.

The SharedFactory class used follows:

```cpp
// Factory class
template<typename D>
class SharedFactory
{
public:

typedef std::unordered_map<std::string, 
    std::function<std::shared_ptr<D>()>> registry_map;

registry_map map;

// use this to instantiate the proper Derived class
std::shared_ptr<D> instantiate(const std::string& name)
{
    auto it = map.find(name);
    return it == map.end() ? nullptr : (it->second)();
}

// register the Derived class
template<typename T>
void FactoryRegister(std::string name)
{
    map[name] = []()
    {
        return std::make_shared<T>();
    };
}

};
```

The factory used in the package does not return an unique pointer as usual but a shared pointer because the it needs to be passed to other functions and used simultaneously.
**Lambda function**

The lambda functions are useful to deal with the optimization task. The optimizer has as input a functor parameter (an object with the operator()) which is the function that needs to be optimized. In our case the functor is a lambda function that capture the warping object and the object w_set; that has as argument the warping parameters that need to be optimized and return the minimal dissimilarity computed. The struct w_set is external to the class WarpingMethod and is captured by the lambda function because the object warping being used by different threads at the same time cannot save internally the function on which it is working.

Another issue is that the warping method should be written in a general way (without using data structures of the optimizer that can change) so once the general (with parameters an arma::colvec) lambda function is passed to the optimizer is defined another internal lambda function that adapts the input to the optimizer method chosen.

```cpp
// LAMBDA FUNCTION IN execute.cpp
auto fun = [this,&wset] (const colvec & arg)
{
    return this->warping->warp(wset,arg);
};
```

```cpp
// the lambda function is passed to the optimizer
index_temp(t) = optimizer->optimize( arg, warping, fun);
```

```cpp
// LAMBDA FUNCTION IN optimizer.cpp
auto fun2 = [&fun] (const argument & argt)
{
    // here I have to convert the argument input to an input for fun
    colvec argt_fun(argt.nr());
    for(uword i=0;i<argt.nr();i++)
        argt_fun(i)=argt(i);
    return fun(argt_fun);
};
```

### 3.8 Profiling

To profile the package the google CPU profiler *gperftools* has been used. The function with the biggest impact on the execution time was `approx()` ("the first function written"). Then `approx()` was reimplemented in a more efficient way reducing the execution time occupied from 24% to 6%. 

21
Chapter 4

A live Web Application

To test the package interactively a web application is available at https://zitale.shinyapps.io/fdakmapp/

The application is developed using the R shiny package. It is basically divided in three files:

- **ui.R** is the file that organize the front-end. The user interface is divided in a sider panel where is possible to choice the favorite inputs and a main panel where the output are returned.

- **server.R** is the back-end file that manages what happens once the execution is lunched through the RUN button in the user interface.

- **script.R** is the container file for all the functions used in the server file.

The App is subjected to restrictions because deployed using a free service.
Chapter 5

Results and Conclusions

In this chapter we can find applications of the package to some functional datasets example. The following datasets are available in the package and have been used during the development for testing. The examples are:

**Simulated30**

The dataset is composed by 30 mono-dimensional curves. The curves \( c_i(t): (0, 2\pi) \rightarrow \mathbb{R} \) such that curves from 11 to 20 are obtained substituting each abscissa \( t \), of the curves from 1 to 10, with the modified \(-\frac{1}{3} + \frac{3}{4}t\) such that the 2 groups differ only in the phase; end the curves from 21 to 30 are obtained modifying the amplitude.

![Simulated30 Example](image)

Thus, among the three clusters of curves, visible in Fig.5.1, only 2 amplitude clusters are present, and one of them has associated a further
cluster in the phase. In figure 5.1 we can see if we look for 2 clusters the phase variability disappears because of the alignment. If we look for 3 clusters all the groups are detected and aligned.

**A tutored example: Aneurisk65**

This dataset has been collected within the AneuRisk65 project and curves represent the reconstruction of the main cerebral vessel, the inner carotid artery, described in terms of the vessel centerlines.

A detailed description of the project’s aim can be found at the AneuRisk65 web page http://mox.polimi.it/it/progetti/aneurisk/. In this particular example the algorithm is applied to the first derivative functions. Once installed the package as shown in chapter 6 execute the following commands to reproduce the results.

```r
library(fdkmapp)
res <- kmap(x= aneurisk65$x, y= aneurisk65$y, 
n_cluster=2, seeds=c(32,64),
            comp_original_center = TRUE)

kmap_show_results(res,TRUE,TRUE)
```

Figure 5.2: Aneurisk65 Example.

The results are clear and stable (they don’t depend on the seeds chosen). Looking at the first dimension we observe that two groups are well defined:
the blue cluster with only one peak and the red cluster with 2 peaks in the first dimension.

Figure 5.3: Aneurisk65 Example: In left figure are compared the dissimilarities: between the original functions and the original center in the left boxplot; between the warped functions and the center of the cluster which they belong in the right boxplot. In the right figure the warping functions applied are printed by cluster.

As expected in figure 5.3 we can see the variation of the dissimilarity of the function with their own center. The warping functions have different domains because they depend on the domains of the original functions that are different.

5.1 Comparison with previous package

In this section are commented the results of the test to compare the new algorithm with the previous state of the art implementation in the package *fdakma* published on CRAN. The dataset used is simulated30, both the algorithms are set with the same specification: "medoid" as center method, same seeds (1,11,21), and same stopping conditions (toll=0.01), same dissimilarity method (pearson). The simulations have been repeated 10 times to obtain mean values. The serial version of the new package *fdakmapp* finds the same clusters in 0.083 seconds against 16.51 seconds needed by *fdakma*. Therefore, we can state that the execution time is reduced by a factor 200.

5.2 Conclusions

The *fdakmapp* package gives the expected outputs on known datasets furthermore being available the parallel versions gives us the possibility to apply
the clustering with alignment algorithm to very large dataset. It is actually being used to find functional representatives in a dataset with about 230000 curves representing the brain’s fibers. The computational performances are good enough to move on with the statistical research stalled by the inefficient previous implementation.

![Brain's fibers dataset](image1.png)

**Figure 5.4: Brain’s fibers dataset.**

5.3 Future development

The package is distributed on github so each user can download and customize it as explained in chapter 6. Further development could solve limitations found in the package:

- The package could handle the derivatives of input curves to support distances that involve them.
- A better `kma_show_results` can be developed (maybe integrating `rgl`).
- An `arma::vec` can’t contain more then $2^{31} - 1$ element. This can be a problem being the matrices vector with the dimensions attribute.

5.4 Documentation

The documentation is created using `Roxygen2` as usual for R packages (`fdakmapp.pdf` in the package directory). For the C++ code, additional documentation is created by `Doxygen` and available at: https://zitale.github.io/fdakmapp/.
Chapter 6

FAQ

6.1 How to install the package?

The package is linked against OpenMP, the BLAS and LAPACK libraries in Makevars. The package can be installed directly from github. In order to do this devtools library needs to be installed and loaded in the R environment. The dependencies (Rcpp and RcppArmadillo) will be automatically installed.

In R:

```r
1 install.packages("devtools")
2 library(devtools)
3
4 install_github("zitale/fdakmapp@pacs")
```

To install the development version simply remove ”@pacs”. Note that the installation in Windows require a proper version of Rtools.

How to use the package on MOX machines?

In the file .bashrc the following modules needs to be loaded:

```bash
1 module load gcc-glibc
2 module load R
3 module load lapack
```

If we directly try to install the package with install_github() we have an error during the Rcpp or RcppArmadillo installation:

```sh
1 error: "usr/bin/gtar: No such a file or directory"
```

So we need to install the dependencies separately. In the R environment execute:

```r
1 install.packages("Rcpp")
2 install.packages("RcppArmadillo")
```
And then proceed with installation of the package:

```r
install.packages("devtools")
library(devtools)
install_github("zitale/fdakmapp@pacs")
```

### 6.2 Other examples

After the installation try to execute the following examples.

```r
library(fdakmapp)

# ############ EXE ########################
res <- kmap(x = aneurisk65$x, y = aneurisk65$y, n_clust = 2,
seeds = c(32, 64))
# ############ OUTPUT ################
kmap_show_results(res, FALSE, FALSE)

# ############ EXE ########################
res <- kmap(x = simulated30$x, y = simulated30$y,
n_clust = 3, seeds = c(1, 11, 21))
# ############ OUTPUT ################
kmap_show_results(res, FALSE, FALSE)
```

### 6.3 How to add a new feature?

In this section we can find simple instructions to add new features.

#### Add new dissimilarity methods

To add a new dissimilarity the step to follow are:

- **Declaration** In dissimilarity.h create a class derived from Dissimilarity. For example, to add the \( L^2 \) distance the following code was included in the file.

```c
/// L2 Distance
class L2 final: public Dissimilarity
{
  public:
  L2():Dissimilarity() {};
  virtual double compute(const rowvec& xf,
  const rowvec& xg, const mat& yf, const mat& yg);
};
```
As we can see from the declaration this class has a virtual member function that has to override the compute method of the base class.

**Definition** In dissimilarity.cpp define the compute function.

```cpp
double L2::compute(const rowvec& xf, const rowvec& xg,
const mat& yf, const mat& yg)
{
    grid gr = setGrid(xf, xg, yf, yg);
    if(gr.yf_sim.is_empty())
        return 10000000;

    uword dim = gr.yf_sim.n_rows;
    uword len = gr.x_sim.size();
    double res(0);
    rowvec d = gr.x_sim.cols(1, len-1) -
                gr.x_sim.cols(0, len-2);
    double D = gr.x_sim(len-1) - gr.x_sim(0);

    for(uword k=0; k < dim; k++)
    {
        rowvec diff = sqrt(d) %
                      ( gr.yf_sim.row(k).cols(1, len-1) -
                        gr.yg_sim.row(k).cols(1, len-1) );
        res += dot(diff, diff)/(D*dim);
    }
    return sqrt(res);
}
```

In this case we are adding a distance so it’s ok for the minimizer. In the case of a similarity index this function must return the opposite that is a shifted dissimilarity.

**Registration** To use the dissimilarity method added it needs to be registered in the dissimilarity factory that can be found in the kma_model.cpp.

```cpp
// dissimilarity factory
SharedFactory<Dissimilarity> disfac;
disfac.FactoryRegister<Pearson>("pearson");
disfac.FactoryRegister<L2>("l2");
```

In the last line we can see how the method is registered

```cpp
Factory.FactoryRegister<DerivedClass>("InputString");
```
Add new optimization methods

To add new optimization methods the steps are the same.

- **Declaration** In optimizer.h create a derived class from Optimizer-Method that override the optimize virtual member function.

- **Definition** In optimizer.cpp can be included the library that performs the optimization and the optimize function is defined. If necessary the arma::colvec input can be adapted to the data structure used by the optimizer through a lambda function. Upper and lower bounds need to be defined using the warping object.

- **Registration** Don’t forget to register the new method in kma_model.cpp.

Add new center methods

- **Declaration** In center_method.h create a derived class from Center-Method that override all the member functions if needed.

- **Definition** In center_method.cpp define the member functions. If the parallel_type = 0 the method computeCenter will be used. If the parallel_type = 1 the method computeParallelCenter will be used. As in the case of Mean also one of them can be defined but remember to protect the execution updating checkin.cpp file.

- **Registration** Don’t forget to register the new method in kma_model.cpp.

Add a new warping method

The warping method is the most complex feature that can be added but the steps to follow are the same:

- **Declaration** In warping.h create a derived class from WarpingFunction. This class must have as attribute lower, upper buonds and a pointer to the class Dissimilarity.

- **Definition** All the virtual member function must to be defined. In particular the warp method must return the dissimilarity between the function in w.set with the abscissa xf warped. An example of warp function in the case affine transformation in warping.h:

  ```cpp
  virtual double warp(const warping_set& w_set,
                      const colvec& arg) const
  {
    return diss->compute(arg(0)*w_set.xf+arg(1),
                          w_set.xg,w_set.yf,
                          w_set.yg);
  }
  ```

- **Registration** Don’t forget to register the new method in kma_model.cpp.
Bibliography


