StatisCLAS

Methods for Statistical Classification

David Casado de Lucas[1]

Last update: 29th May 2013

[1] Complutense University of Madrid (Spain), http://www.Casado-D.org, davidcasado@pdi.ucm.es

Key words and phrases: classification, time series, integrated periodogram, functional data, depth, robustness, software.

Contents

1 Prologue

This folder (where this file is) contains MATLAB code that implements—with some new features—the classification methods for time series and functional data proposed in Alonso et al. (2008, 2012): *DbC* and *DbC*-α both for classifying stationary and nonstationary time series, and *WI* and *WD* for classifying functional data. Parts of the code were initially written in collaboration with some of my coauthors, Andrés--M. Alonso and Sara López Pintado, to whom I am grateful. Because of the huge amount of time that I have spent on this code, well over that necessary to publish our works (e.g., preparing this helping file has taken several weeks), we decided to maintain only my name in the authorship.

The implementation has evolved from just the abovementioned methods to a more general implementation with which it is possible to optimize, modify, create, compare and select classification methods. For each type of datum (time series and functional data) there are three scripts: one to work with simulated data, other to work with real data and a third one to really classify new (unlabelled) data, the kind of problem most researchers need to solve usually. Some ideas implemented in this package are, to the best of my knowledge, new (I have named some concepts too); hence, I shall probably use them—alone or with collaborators—in publications. Now, however, making a living is more important than publishing and being lucky is more important than working hard. Or I think so.

I have spent some extra time preparing this code, so if someone detects any bug, I would be very grateful to know about it. In writing this helping file, the criteria of *Modern Language Association Style Manual and Guide to Scholarly Publishing* and *Practical English Usage* have been taken into account—see references Gibaldi (1998) and Swan (1995). Suggestions and information about misprints or linguistic matters would be welcomed too. I thank you very much for your help.

1.1 What Is New

In the present version of the package, some improvements and new features have been implemented:

1. Two previous packages, `codeDFMforFD` and `codeRFDforTS`, have been merged, and the code has been modified so as to make it easier to maintain and reuse—the six main scripts of the new package are different but share many lines.

2. The principal steps or tasks of the algorithms are identified with a number, which allows only these parts to be changed—e.g., functional distance or transformation—or new methods to be designed. Each method is determined by few numbers and a name. See section 2.4.

3. The value of a parameter—*number of blocks* for time series and *differentiation order* for functions—can be optimized in each run. In this case, for each value a measure of the minimizing-power is shown. See section 2.5.1.

4. When there is more than one method, in each run the most appropriate can be selected automatically. In this case, for each method a measure of the minimizing-power is shown. See section 2.5.2.

5. For given samples it is possible, in the learning scheme determined by the previous method selection, to select the most appropriate data transformation, distance, type of discriminant vector, multivariate classification submethod, et cetera. Some of these ideas were outlined in my doctoral thesis.

6. Theoretical explanations have been included so that to explain how methodologies and scripts work—e.g., effects on the quality of the estimations, or when a larger number of runs is necessary.

7. Some code has been written to reduce the computational effort (that described in the two previous items can be used for this purpose too) or to control and warn the user. See section 2.8.

8. A new kind of script has been added, to obtain labels instead of error rates. The user can place new data by applying different types of call. See section 3.

9. The preprogrammed stochastic models, for processes or functions, have been generalized by using coefficients.

10. Several simulation exercises have been included just as mere examples or to show some concepts.

A new version of this package—already partially written— with some new interesting capabilities and ideas will appear in some months.

2 Main Features

2.1 Design of the Code

The code of this package has been designed:

1. To choose possible different sizes for the four training and testing samples.
2. To consider other pairs of models easily: it is enough to create new cases in `dataExercisesFD` or `dataExercisesTS`.
3. For given data, to apply one classification method or to compare several ones. It is easy to consider other existing methods rather than our proposals, they can be integrated in our schemes by using the simple trick of calling the identity in one step and the method itself in the other.
4. When more than one method is applied, to select automatically the one that is expected to minimize the estimated overall misclassification error rate. As a particular case, it selects automatically between methods *WI* and *WD*, or betweeen methods *DbC* and *DbC*-α. A measure of the minimizing-power of the methods is given after the B runs.
5. To design new global methods of classification by selecting the intermediate steps, since each global method is determined by few numbers and a name.
6. For each method, to enter the initial vector of parameter values to be considered—*number of blocks* for time series and *differentiation order* for functional data.
7. For each method, to select the parameter value that is expected to minimize the mean estimated overall misclassification error rate. A measure of the minimizing-power of the values is given after the B runs.
8. To make decisions fastly and automatically (without human supervision) about the method, the distance, the transformation and the parameter value that are most convenient for the particular framework. This has many practical applications, for example in Engineering.
9. To use different functional distances: L_1 (default), L_2, L_∞, et cetera.
10. To use, as reference function of a set, the mean (default), the α-trimmed mean or others the user can implement.
11. To gain access to the estimated error rates or the estimated labels of each group separately or jointly. In some cases, the behaviour can be different for the two groups. Error rates are useful for methodological researches, while labels are useful for applied researches or even teachers.

12. To obtain graphical information during the first run: crude data, transformations of the data, discriminant variables, expected mean estimated error rates, expected mean computational time, expected minimizing-power measure of the parameter values or methods, et cetera. These figures, which also show the similarity between the training and the testing samples of each group, can be used as a descriptive and prospective tool.

13. To represent, after the runs, the estimated misclassification error rates of the algorithms graphically, with the values of the parameter as labels—*number of blocks* for time series and *differentiation order* for functional data.

14. When neither the parameter nor the method is optimized, to obtain an estimation of the results as if it had been done.

15. To gain access to the computational times of each run through two measures: time spent for each value of the parameter and time spent for the whole vector of values.

16. To record the results in a file, with enough information to analyse and represent the results later.

17. For time series, to consider the periodogram, the normalized (default) or nonnormalized version of the integrated periodogram, or the logarithm of the periodogram. The user can easily implement other functional data to be constructed from the series, as well as transformations of the previous ones.

18. For functional data:

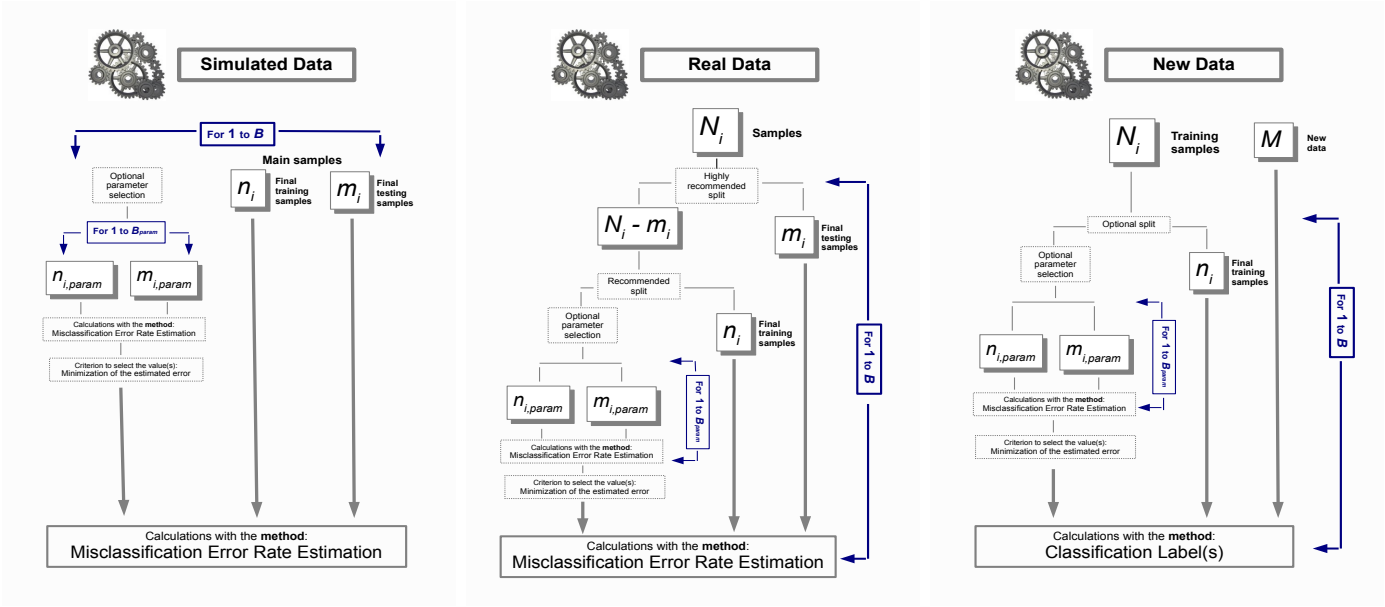
    - To add four types of smooth peak to any model.
    - To apply a smoothing method to the crude functions (others can easily be implemented).
    - To choose between the derivatives or the differentials of the functions.
    - To select the best differentiation order or, alternatively, combine all of them through a multivariate vector.
    - To consider only the crude functions or additional, possibly nonconsecutive derivatives or differentials up to any order.
    - To generalize the methodology to other types of discriminant variables, possibly not based on the derivatives or differentials.

19. An additional function is included to generate tables in the LaTeX sintaxis—the user can edit this function.

20. To make the code easy to maintain, reuse and generalize: all the scripts are different but share many lines of code, lots of comments have been included, variables have been given meaningful names, et cetera.

The code works with many quantities and values, for which management hypermatrixes (multidimensional arrays) have been intensively used. Thus, it seems that these structures must play a central role in modern programming languages.

Figure 1: Ways of using the samples



2.2 Ways of Using the Data

In figure 1 (expanded versions are given in appendix B) we represent how the available data are usually used to estimate the misclassification error rates or the labels. Our scripts, described in section 3, implement these three ways. The main idea is that each datum can be used several times (in different runs), but preferably for only one of the three main tasks at the same time.

- In the scheme on the left, each datum is used only once, which is usually something difficult to afford.
- In the scheme in the middle, data are exploited more efficiently, since each datum is used B times.
- Finally, in the scheme on the left, the training data can be used one or several times, as desired.

On the number of runs B, too small a value does not guarantee a minimum quality of the estimations involved in the process, while too big a value can cause overfitting (such a good fit to the data that the method has poor behaviour when it is extrapolated). See section A.1.2.

It is worth noticing that the error of (good) estimators usually varies nonlinearly with the sample size, so only with an endless quantity of data is the first scheme (on the left) applied to real data. In our implementation of the second and third schemes (in the middle and on the right), the highly recommended split is mandatory while the recommended split is controlled by the parameter `paramDataReuse` (nevertheless, not reusing the data may imply small samples in the inner loop, which is also undesirable—see section A.1.2. The user can try both options).

The objective of the scheme on the right is to classify—as good as possible—the elements, only once. The nested or inner loop of the schemes allows making decisions fastly and automatically (without human supervision) so that to use the method, the distance, the transformation and the parameter value that best fit the data. This has many practical applications.

When more than one classification method are to be applied, all of them are called with just the same data (or almost the same data when the depth-based robustifying approach described in section 2.6.2, which may remove some elements, is applied).

2.3 Descriptive and Prospective Run

For the first run, when `dataFigures = true` the scripts generate some figures if `paramOptimization = false`, as otherwise not all the values and methods are considered in the main loop, where figures are generated. However, the user can previously execute the script without parameter optimization nor method selection, that is, with `paramOptimization = false` and `methodSelection = false`.

The plots can provide useful descriptive and prospective information about the similarity of the groups, the representativity of the training and testing samples, the discriminant power of the variables (a bimodal character is desired), the parameter value or method that seems to minimize the error rate, et cetera.

The figures are based on one run, so perhaps executing this part of the code several times would be advisable. In the examples of section 6, the user can see information of the same kind than that provided during this run.

2.4 (Global) Classification Methods

Since our methodologies can be divided in two main steps (transformation of the crude data and classification of the new type of datum), we use the adjective *global* to make it clear whether we are talking about the whole classification process. In the code, each of these methods is determined by few numbers and a name—see `numbersTStoCLASSmethods` and `namesTStoCLASSmethods` for time series, and `numbersFDtoCLASSmethods` and `namesFDtoCLASSmethods` for functions. In the scripts, only our algorithms are implemented and called by default, although users can call only one of them or implement new methods. Classification methods are integrated in our schemes as follows.

2.4.1 Our Methods

From the meaning of the positions in `numbersTStoCLASSmethods`, the global methods that we proposed in Alonso et al. (2008), for classifying time series, are:

- Method *DbC*, determined by 1 1 1 1
- Method *DbC*-α, determined by 1 2 1 1

while from the positions in `numbersFDtoCLASSmethods` the global methods proposed in Alonso et al. (2012), for classifying functions, are:

- Method *WI*, determined by 1 1 1 1
- Method *WD*, determined by 1 2 1 1
- Method *dKNN*, determined by 1 3 1 1

2.4.2 How to Implement New Methods

For time series, a method not building functional data as an intermediate step can be implemented by calling the identity in `fromTStoFD` (already implemented under the number 0) and the method itself in `fromFDtoCLASS`. In general, new functional classification methods can easily be added directly or using `fromFDtoCLASS` as mapping function. After this, it is necessary to edit in the code the `switch`-structures that are just before and after the calls to the function `fromFDtoCLASS`:

1. In `SCRIPTsimulatedTS`, around lines 179 and 298.

2. In `SCRIPTrealTS`, around lines 378 and 491.

3. In `SCRIPTnewTS`, around lines 348, 468 and 476.

Finally, `numbersTStoCLASSmethods` and `namesTStoCLASSmethods` allow the call.

For functional data, a method not building multivariate data as an intermediate step can be implemented in `fromFDtoMV` while the identity is called in `fromMVtoCLASS` (already implemented under the number 0). In general, the user can directly implement new multivariate classification methods or use `fromMVtoCLASS` as mapping function. After this, it is necessary to edit in the code the `switch`-structures that are just before and after the calls to the function `fromMVtoCLASS`:

1. In `SCRIPTsimulatedFD`, around lines 439 and 453.

2. In `SCRIPTrealFD`, around lines 643 and 657.

3. In `SCRIPTnewFD`, around lines 595 and 607.

Finally, `numbersFDtoCLASSmethods` and `namesFDtoCLASSmethods` allow the call.

2.5 Optional Inner Loop

By running this optional code, some quick decisions can be made so that to reduce the calculations in the main loop. Since these decisions are based on the estimation of the misclassification error rates, it is important to take into account the possible poor estimation due to small sample sizes, regardless the training sample sizes. See section A.1.2. When `verbose = true`, the estimations of the error rates are shown also for this loop.

2.5.1 Parameter Optimization

The user enter the initial set of parameter values to be considered—*number of blocks* for time series or *differentiation order* for functional data. When all the parameter values are considered, the output does not provide direct information about how many times a value has minimized the error rate. By looking at either the textual or the graphical information, a value or a range of values can be chosen for a later execution of the code—see figure 8 in section 5.2.

Alternatively, by setting `paramOptimization = true` the user can enable the automatic optimization of the parameter value for each (global) method. When this optimization is applied, the relative minimizing--power of each value is measured as described in section 2.5.3. The mean minimizing-power is shown after the runs. On the other hand, when this loop is disabled, the minimum overall error of each run is registered and the user is given an estimation of the error that would have arisen if the parameter had been optimized. Since different values can be selected in different runs when the optimization is applied, the results of all methods are textually and graphically shown after the B runs; the user must carefully interpret these results. In our implementation, the parameter optimization is necessary to apply the method selection described in section 2.5.2. This optimization is implemented in the two following versions.

In Two Steps

This approach is available in the two first schemes of figure 1, in section 2.2, as they are thought for researchers to evaluate or compare classification methods. In this framework, it makes sense to select in the inner loop more than one parameter value—or even method—and consider them again, for a second chance, in the outer loop (it can be thought of as a way of creating pseudoties). Thus,

1. In the inner loop, after considering all the initial parameter values, the minimum of the run-averaged overall error rates is calculated. Then, the parameter values for which the method have provided a rate inside the interval `[minimum, minimum*(1+discretenessMargin)]` are enabled for the main loop:

```
methsXparams(i1,meanErrorRatesParam(i1,:)<=...
                    (methodMinError(i1)*(1+discretenessMargin))) = 1;
```

2. In the outer loop, all the parameter values previously selected are considered. Finally, the parameter value for which the method provides the minimum run-averaged overall error rate is chosen. Through the minimizing-power measure, our code registers possible, improbable ties.

The user can change the mentioned margin or disable it. This technique in two steps is a way of improving not the classification methods themselves but the parameter optimization and therefore the estimation of the error rates, as the final obtained estimation would correspond to a better estimation of the parameter in the inner loop. This improvement, though, increases the computational effort.

In One Step

For the third type of script of figure 1, in section 2.2, as it is thought to really label new data, the parameter value—and the method—must be selected as follows:

1. In the inner loop, after considering all the initial parameter values, the minimum of the run-averaged overall error rates is calculated. Then, the parameter value for which the method provides the minimum is chosen. Through the minimizing-power measure, our code registers possible, improbable ties.

2.5.2 Method Selection

The user can include more than one classification methods in the call. By looking at either the textual or the graphical information, the user can select a method for a later execution of the code. When all the methods are considered, the user does not have direct information about how many times a method has minimized the error rate.

Alternatively, by setting `methodSelection = true` the code selects the method—and the parameter value—that is expected to minimize the overall misclassification error rate. In this case, the code automatically enables the parameter optimization described in section 2.5.1. It is allowed to apply the method selection even for one parameter value only, but it is disabled if there is only one initial method. There are several possible criteria to order the methods totally so as to select one. When this selection is applied, the relative minimizing-power of each method is measured as described in section 2.5.3. The mean minimizing-power is shown after the runs. As a particular case, this approach can be used to select between

our methods. For example, when classifying time series with moderate training sample sizes, if there is no contamination *DbC* uses all the data and tends to provide smaller error rates than *DbC*-α, while this method leaves out the data with smallest depth and therefore tends to have better behaviour in the presence of contamination. This selection is implemented in the two following versions.

In Two Steps

Since the discreteness effect described in section A.1.2 may affect the selection, it seems reasonable to implement also a two-step approach in which more than one method can be selected in the inner loop so as to consider it again, for a second chance, in the outer loop (it can be thought of as a way of creating pseudoties). Such an implementation makes sense in the scripts whose aim is the evaluation or comparison of classification methods, that is, in the two first schemes of figure 1, in section 2.2. As we considered a margin in selecting the parameter values, now we use the mean of the estimations of the method for those selected values (other criteria could be applied). Thus,

1. In the inner loop, for each method the mean of the run-averaged error rates for the parameter values selected as described in section 2.5.1 is compute

   ```
   methodValueCutoffs(i1) = mean(meanErrors(methsXparams(i1,:)));
   ```

   where `methsXparams(i1,:)` is a vector with 0's and 1's, and their minimum is calculated (we can call it *minimum-in-mean*). Then, those methods that have provided a mean outside the interval `[minimum, minimum*(1+discretenessMargin)]` are disabled for the main loop:

   ```
   methsXparams(methodMinError>(min(methodMinError)*(1+discretenessMargin)),:)  =...
     zeros(sum(methodMinError>(min(methodMinError)*(1+discretenessMargin))),length(paramValues));
   ```

2. In the outer loop, all the methods previously selected are considered. After the runs, the minimizing--in-mean method and its best parameter value are chosen. Through the minimizing-power measure, our code registers possible, improbable ties.

Then, between two methods having the same behaviour in both the inner and the outer loop (this might happen, because of the discreteness effect, especially for small-sized testing samples), the output of the method allocated firstly in the call is shown. Notice that a method with good average behaviour for the selected parameter values is preferred to a method with the smallest error rate for a value but worse mean results for the selected parameter values (we suppose that the initial parameter values are not too sparse). The user can change the mentioned margin or disable this two-step approach.

Finally, this technique in two steps is a way of improving not the classification methods themselves but the selection procedure.

In One Step

For the third type of script of figure 1, in section 2.2, which is thought to label new data, both the parameter value and the classification method are selected as follows:

1. In the inner loop, for each method the mean of the run-averaged error rates for the parameter values selected as described in section 2.5.1 is computed

```
methodValueCutoffs(i1) = mean(meanErrors(methsXparams(i1,:)));
```

where `methsXparams(i1,:)` is a vector with 0's and 1's, and their minimum is calculated (we can call it *minimum-in-mean*). Then, the minimizing-in-mean method and its best parameter value are chosen. Through the minimizing-power measure, our code registers possible, improbable ties.

Then, between two methods having the same behaviour in both the inner and the outer loop (this might happen, because of the discreteness effect, for small-sized testing samples).

Learning Scheme

From a theoretical point of view, it is worth noticing that this method selection represents a way of combining—as they all are involved—different classification methods.

A global method is determined by a transformation, a classification submethod, a distance, et cetera, and the inner loop can be used to learn from the training data so that to classify the final testing samples as good as possible. The automatic selection among our classification methods has already been mentioned, and another specific application will be introduced in section 2.6.1. Now, we highlight some other interesting applications of this learning scheme. Let be a global method, say `[1 1 1 1]`; then,

Transformation-Selection

Since each data transformations is characterized by a number in the first column of a matrix, the best of three transformations can be automatically selected by considering the matrix

```
[1  1  1  1;   2  1  1  1;   3  1  1  1].
```

As particular cases, it can be considered several types of built function for time series and several types of discriminant vector for functions—in our papers we considered, respectively, the integrated periodogram and the discriminant vector based on derivatives or differentials.

Submethod-Selection

Since each classification submethod is characterized by a number in the second column of a matrix, the best of three submethods can be automatically selected by considering the matrix

```
[1  1  1  1;   1  2  1  1;   1  3  1  1].
```

Distance-Selection

Since each distance is characterized by a number in the third column of a matrix, the best of three distances can be automatically selected by considering the matrix

```
[1  1  1  1;   1  1  2  1;   1  1  3  1].
```

For example, in some situations we may be interested in possible peaks—representing an event—of the functional data, while others we may want to ignore them.

Reference-Selection

Since each type of set reference is characterized by a number in the forth column of a matrix, the best type of three can be automatically selected by considering the matrix

```
[1  1  1  1;   1  1  1  2;   1  1  1  3].
```

This can be used, for example, to select among the mean, the trimmed mean and the weighted mean.

2.5.3 Minimizing Power

As a way to measure the relative importance of the parameter values and methods in minimizing the error rates, we have described that:

- If the parameter optimization is enabled but the method selection is not, for each method a "unit of parameter minimizing-power" is distributed among the minimizing parameter values (usually one). If there are p parameter values, for method i-th, $\{u_{ij}^{(p)}\}$, for $j = 1, ..., p$, with $\sum_{j=1}^{p} u_{ij}^{(p)} = 1$.
- If both the parameter optimization and the method selection are enabled, a "unit of method minimizing-power" is distributed among the minimizing methods (usually one) and, at the same time, for each minimizing method a "unit of parameter minimizing-power" is distributed among the minimizing parameter values (usually one). If there are m global methods, $\{u_i^{(m)}\}$, for $i = 1, ..., m$, with $\sum_{i=1}^{m} u_i^{(m)} = 1$. In this case, the quantities $\{u_{ij}^{(p)}\}$ takes also into account the number of times method i-th has minimized the overall error rate.

For a parameter value or a method, the previous quantities $u_{ij}^{(p)}$ and $u_i^{(m)}$ are affected by—take into account—the other parameter values and other methods in the comparison, that is, it is a relative measure of the minimizing-power.

Registering the minimizing-power in this manner allows noticing ties (among values and among methods), although almost always only one parameter value and one method are selected and these measures can additionally be interpreted as the relative frequencies with which each value or method has minimized the overall error rate.

2.6 Robustifying Techniques

2.6.1 Selection-Based

When there are several (global) methods, the schemes of figure 1, in section 2.2, allow selecting the method with the smallest estimated overall error rate in the inner loop so that to use it in the outer loop—see the details in section 2.5.2. If one of the methods is robust, it is expected to be automatically selected when a noticeable amount of atypical data is present in the samples.This can be thought of as the robustification of the whole classification process.

This selection-based approach is a characteristic of the schemes, not a robust classification method in itself. The importance of the robust method could be noticed by looking at the minimizing-power measure.

2.6.2 Depth-Based

In this section, an approach—of which I am the author—to robustify any classification method is described. It consists in identifying the deepest elements at the beginning so that to remove them from the training data during the runs—although they are maintained in the final testing samples. In our methodology for time series, the depth is applied at the functional data level (after the functions are constructed from the series), while in our methodology for functional data the depth is applied at their level (crude data). In classifying time series, this robustifying process is an alternative—recommended for big sample sizes—to the robust algorithm *DbC*-α. For each global method and each parameter value:

1. The deepest functional elements are identified. They can be different for each method and even each parameter value.

2. In the inner loop, only these elements are maintained in both the training and testing samples (see why in section A.1.1). After removing the elements with the smallest depth, the run of the loop is registered as untrustworthy—for this method and parameter value—if the two testing samples are empty or so is any of the two training samples. This step can imply a slight reduction—usually negligible in practice—in the sample sizes or in the number or runs. The overall error rates are estimated by using only the trustworthy runs.

3. In the outer loop, the deepest elements are maintained in the training samples, while the testing samples remain unchanged so that to allocate all the initial data.

Another implementation is possible: after considering only the deepest elements of the training data of the outer split, the split of samples of the inner loop can be applied for each method and parameter value. In our implementation, the same splits of the samples are applied to each method and parameter value, and then each method decides which elements are trustworthy; this maximizes the similarity of the samples and also guarantees that just the same data are used by all the methods when no element is removed from the samples. On the other hand, this avoids the possible variability that could come from the splits and not from the methods themselves.

It is possible to think about our implementation as a way of working with the truncated probability distributions of the underlying stochastic models. Notice, though, that in the outer loop all data are taken into account for estimating the error rate.

On the other hand, this technique is related to some effects, namely: negative training- and testing--sample-size effects in the inner loop, a training-sample-size effect in the outer loop, training and testing strengthening and weakening effects, and, for some methods, a sample-size time effect. See section A.

We have implemented this approach only in the scripts for real or new data, since for simulation exercises the user can control the characteristics of the data (or the other scripts can be adapted). Some simulation exercises have shown that the method *DbC* with this robustifying approach can outperform the robust method *DbC*-α.

Finally, this depth-based robustifying approach can be applied outside the framework of this package, to other type of datum, to other classification methods or with other depth measure.

2.6.3 (Unweighted) Majority-Vote

In the scripts for new data, our code provides the labels of the classification methods plus, when $B > 1$, the unweighted majority-vote classifier (it provides useful information only when the parameter value has been optimized, as otherwise the same labels are obtained in all runs). Theoretical explanations on the robustifying effect of averaging several classifiers are given in section A.1.5.

It is worth noticing that this approach is a characteristic of the schemes, not a robust classification method in itself.

2.7 Computational Time

The scripts of this package show textual and graphical information about the computational time. From it some patterns of dependence may appear: for series computational time depends nonlinearly with the number of blocks while for functions it does not depend on the order of differentiation. Computational time is measured in two manners. In a hypermatrix the time that each global method spents for each value of the parameter is registered, while a matrix contains the joint time that each global method spents for all the parameter values.

When the parameter is optimized, the individual measure must be cautiously interpreted as it depends also on the number of times that each value is selected (absolute frequencies of the parameter values), not only on the global method itself—in this case, though, the user is usually interested in the joint measure. When the method selection is enabled, computational time depends also on the number of times that each method is selected (absolute frequencies of the methods).

Apart from the computer on which the code is run, there are some other facts that may affect the computational time registered—see section A.2.

2.8 Reducing Computational Effort

We have written some code to make algorithms and schemes faster. Most of these ideas can be applied outside the framework of this package.

1. The selection of any initial set of values for the parameter, possibly nonconsecutive, allows avoiding additional effort for some models—see figure 8 in section 5.2.

2. Some time-consuming calculations—for example, the functional transformations of the time series or the calculation of the depth—are made for each datum only once, before the main loop.

3. In the scripts for simulated data, the code tries to generate all the data, for all the runs of the main loop, at the beginning; if there is any problem with the memory of the computer, the samples for each run are subsequently generated inside the loop. The user can edit the code to force the generation of the data in the main loop, which would reduce the necessary memory but would increase the computational time. As regards the inner loop, the simulated data are not precomputed, although the user can implement this option.

4. The optimization of the parameter value in the inner loop reduces the calculations in the outer one.

5. The automatic method selection in the inner loop shortens the whole classification process.

6. When possible, matricial calculations are applied rather than `for`-loops—see, for example, the code of `distancesFD` or `funcARMApq`.

7. To reuse some calculations when several methods are subsequently applied, in considering all possible parameter values and methods the nesting order is not

```
for i1 = 1:size(numbersTStoCLASSmethods,1)
    for p = 1:length(paramValues)
      ⋮
```

   but rather we make it

```
for p = 1:length(paramValues)
    for i1 = 1:size(numbersTStoCLASSmethods,1)
      ⋮
```

   For a run, given the data and the parameter value p, this allows reusing the transformations applied by previous methods.

```
if isequal(i1,1) ||...
   (i1>1 && ∼isequal(numbersTStoCLASSmethods(i1,1),numbersTStoCLASSmethods(i1-1,1)))
    ⋮
else
    ⋮
end
```

   However, this cannot be applied when methods change data—e.g., with the technique of section 2.6.2. Finally, in implementing this idea for more than two steps, that all the previous steps are the same must be checked.

8. As a consequence of using fewer but better elements in the samples, computational time may decrease—see section A.2.1.

9. Some "unnecessary" parts of the code, written with controlling or monitoring purposes, can be disabled manually. For example:

   - In the scripts for simulated or real data, the two-step approaches of sections 2.5.1 and 2.5.2.
   - In the scripts for new data, the precaution of including the data also in the testing sample of group 2 to check whether, as expected, the same labels are obtained.
   - For time series, in the scripts for real or new data, the trick to solve the inherent effect described in A.2.4, if: (i) the user is not interested in the accuracy of the computational time; (ii) many runs will be considered; or (iii) or the samples have many elements (and must be removed in the latter case).

10. Other tricks can simplify the calculations, namely: overwriting variables—instead of creating others—saves memory (although, on the other hand, resizing variables lasts); not multiplying by a constant before comparing two quantities if they will not be used later; not creating unnecesary copies of some data but referring to their position (row or column); et cetera.

11. For time series, the length T is recommended to be a power of two—see section A.2.2.

12. In our algorithms for classifying functions, considering differentials rather than derivatives needs fewer calculations.

2.9 Remarks and Pieces of Advice

In this section, we highlight some details of the schemes, the methods and the output, on the one hand, and we advise to avoid wrong interpretations and computational problems, on the other hand.

2.9.1 Initial Values and Schemes

1. Some parameters have no default value assigned in the scripts, and the user must select it.

2. By looking at figure 1, in section 2.2, the user must choose valid, proper values for the sample sizes of the loops. When the depth-based robustifying technique (subsection 2.6.2) is applied, in most runs the sizes of the samples are slightly reduced, since only the deepest elements are maintained in the samples; for small sample sizes this may worsen the discreteness effect described in section A.1.2 and even cause a crash if all the elements are removed from a sample (the probability of this occurring increases with α and the number of runs). The code skips and registers—for the posterior estimation of the error rate—the runs in which the two testing samples are empty or so is any of the two training samples:

```
if isempty([positionsTestingGroup1paramTemp;positionsTestingGroup2paramTemp])||...
   isempty(positionsTrainingGroup1paramTemp) ||...
   isempty(positionsTrainingGroup2paramTemp)
     ⋮
   continue
end
```

3. The parameter T is related to the amount of information available in the data, so the estimations of the error rates strongly depend on it. For small values of T, it is not possible to consider as many blocks (for time series) or derivatives/differentials (for functions) as desired. On the other hand, to capture the details of some models, a large value for T is necessary. Our code covers the case of equispaced data (equispaced variable t, really).

4. Time series and functions are allocated as rows in the matrixes of data.

5. After initializing the statistical and setting parameters, the code sets some quantities and does some calculations, for example: some transformations are applied, some quantities are measured, the method selection is disabled when there is only one method, the parameter optimization is enabled if so was the method selection, et cetera.

6. In classifying new data, if by default the implemented rule allocates the frontier elements (those equally distanced to both groups) to one of the groups, as the rule given by expression 18 does, the estimations of the error rates or the labels will show spurious information. Our code counts the number of elements for which this occurs

```
equalDist = ([groupMV1test(:,i1,p); groupMV2test(:,i1,p)] == 0);
```

warns the user and allocates these elements at random

```
if any(equalDist)
  ⋮
   outclass(equalDist) = randi(2,sum(equalDist),1)-1;
end
```

7. Let the populations be $P^{(k)},\ k = 1, \ldots, K$, an element be $e \in \cup_k P^{(k)}$ and the classification rule be $C : \cup_{k=1}^{K} P^{(k)} \to \{1, 2, \ldots, K\}$. Let us define the successes $\mathcal{E}^{(k)} = \{C(e) \neq k\} \cap \{e \in P^{(k)}\}$, $k = 1, \ldots, K$; that is, $\mathcal{E}^{(k)}$ is the event *misclassifying an element of the k-th population*. For two populations the overall error can be written, for an elements chosen at random from any population, as

$$\begin{aligned} p(\mathcal{E}) &= p(\{C(e) = 2\} \mid \{e \in P^{(1)}\}) \cdot p(\{e \in P^{(1)}\}) \\ &\quad + p(\{C(e) = 1\} \mid \{e \in P^{(2)}\}) \cdot p(\{e \in P^{(2)}\}). \end{aligned} \tag{1}$$

This is the theoretical misclassification error rate. To estimate the overall error rate, the previous formula can be applied by using the two estimations and by estimating the quantities $p(\{e \in P^{(i)}\})$ through $m_i/(m_1 + m_2)$. This formula is applied in our algorithms at the end of the inner loop, since sometimes the size of the testing samples may be reduced slightly:

```
errorRatesParam = (m1param*errorRatesGroup1param +...
                   m2param*errorRatesGroup2param)/(m1param+m2param);
```

On the other hand, at the end of the main loop the overall error rate is calculated by dividing the number of all misclassified testing elements over the total number of them, that is, as

```
errorRates = (errorRatesGroup1 + errorRatesGroup2)/(m1 + m2);
```

before dividing `errorRatesGroup1` and `errorRatesGroup2` by `m1` and `m2`, respectively. This alternative formula needs fewer calculations than the first.

8. In general, if by default frontier elements are assigned to a group (see expression 18), labels could be different for these elements. The labels shown by our code are calculated after allocating the new data as final testing sample of group 1. Our code checks that the same classification is obtained when the data are allocated in the final testing samples of both groups

```
if ∼isequal(sum(pseudoMisclass1 ∼= pseudoMisclass2),M)
   disp(['Different classification, when (p,i1)=(' num2str(p) ',' num2str(i1) ')
     for new data ' num2str(find(pseudoMisclass1 == pseudoMisclass2)) ])
       ⋮
end
```

For some models or classification rules, frontier elements might be differently labelled merely for having been placed at random.

2.9.2 Classification Methods

9. For time series:

   (a) When contamination is introduced, 10% of the contaminating series are allocated at the beginning of the training sample of group 1, since the plotting functions use the first rows of the matrixes. If there is only one of such series, it is allocated at the beginning of the mentioned sample.

   (b) In simulation exercise 3, which involves autoregressive processes with time-varying coefficients, we have checked that values between $c_{k2} = -0.9$ and $c_{k2} = +0.9$ do not cause, for any value of t, the roots of the characteristic polynomial of the process to be inside the unit circle (the process would not be stationary for some values of t).

10. For functions:

    (a) In our framework, the data—the values of the variable t, really—are supposed to be equispaced. Hence, for functional data the derivatives and differentials are respectively calculated as follows

    ```
    ((T-1)^diffOrder)*diff(groupFD1,diffOrder,2);
    ```

    and

    ```
    diff(groupFD1,diffOrder,2);
    ```

    For unevenly spaced data, the user can write preprocessing code to interpolate the functions and then take new evenly spaced data, or, alternatively, apply specific techniques for this sort of datum—see, for example, Gerald and Wheatley (1999).

    (b) In our algorithms for classifying functions, considering differentials instead of derivatives does not need dividing by a tiny quantity, which may avoid too big values.

    (c) Algorithms *WI*, *WD* and *dKNN* work with the same discriminant variables, and hence their information—textual and graphical—about these variables is the same.

    (d) In our procedure *WI*, if a_1 of $\mathbf{a}_F$ is negative, this vector is multiplied by -1 (both vectors are eigenvectors of the same eigenvalue and determine the same projecting direction).

11. Let *eps* be *the positive distance from ABS(X) to the next larger in magnitude floating point number of the same precision as X* (usually $2.2204e-016$). In this package, the discriminant variables with absolute value smaller than $\sqrt[3]{eps}$ are multiplied by 0:

    ```
    groupsMV = groupsMV.*(abs(groupsMV) > (eps)^(1/3))
    groupsMVtest = groupsMVtest.*(abs(groupsMVtest) > (eps)^(1/3))
    ```

    A crash may occur if the models are such that all the discriminant variables take values extremely close to zero.

12. With respect to the within-group variability of the discriminant variables, only those with pooled unbiased estimated variance bigger than `variaCutoff` are considered in the multivariate vector of discriminant variables:

```
Sw = ((n1-1)*cov(X(1:n1,:))  + (n2-1)*cov(X((n1+1):(n1+n2),:)))/(n1+n2-2)
varsIncluded = find(diag(Sw) > variaCutoff)
```

13. The MATLAB's function `classifyMod` validates the pooled covariance matrix $\mathbf{S}_x$ of the multivariate data. For some models and many variables, we have obtained $\mathbf{S}_x$ being singular in some runs. The probability of this occurring increases with the number of discriminant variables and the number or runs.

14. MATLAB dependences:

    (a) To smooth random noise in some functional models implemented in `dataExercisesFD`, the function `csaps` of the Curve Fitting Toolbox of MATLAB is called.

    (b) As implementation of the multivariate k-nearest neighbours classification method, called in `fromMVtoCLASS`, the function `knnclassify` of the Bioinformatics Toolbox of MATLAB is used.

2.9.3 Output Information

15. When necessary, some important information is textually shown even if `verbose = false`, namely: the variables with no variability between samples, the number of elements in the frontier, the untrustworthy runs, and the different labelling of new data.

16. When plotting information with descriptive or prospective purposes, it is recommended to execute the script several times, since the normal statistical generation of the data or their split can lead to slightly different numerical and graphical results. A careful interpretation of the figures is always recommended in Statistics (for example, the range of values of the vertical axis).

17. When `dataFigures = true`, many plots are generated. The user can modify this behaviour.

18. Some figures include the testing elements against the training sample means.

19. Although only some series or functions are included in their figures, the values of the discriminant variables are included in the graphics for all the elements of the samples—also the variables not used by the methods because of their negligible variability.

20. In most figures, the axes are automatically adjusted. The user can edit the characteristics of the figures easily: axes, titles, colors, et cetera.

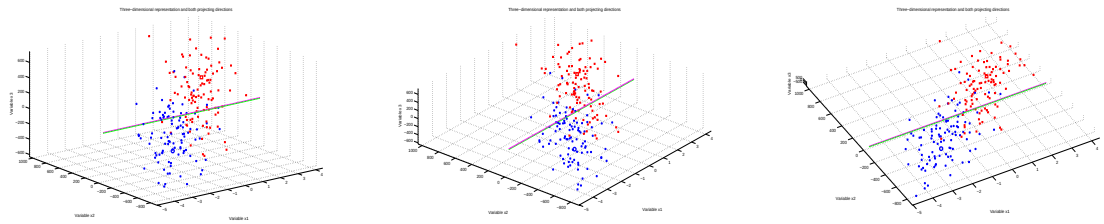
21. When $m1 = 1$ and $m2 = 1$, the figures are based on a split of the data with 33% for the testing samples; that is, the parameter `numElementsToPlot` is ignored.

22. Since for some types of call different parameter values and methods can be selected in different runs, after the main loop some results are textually and graphically shown for all methods, and the user must carefully interpret these results when either the parameter optimization or the method selection has been applied.

23. If, to integrate a new method, the user calls the identity transformation in any step of our methodologies (see section 2.4), different figures will show the same information.

24. When `leaveOneOut = true`, the final boxplots for the error rates are meaningless.

Figure 2: A two-dimensional representation from a three-dimensional one



25. For functional data, when `paramOptimization = false` the discriminant function

$$y(\mathbf{x}) = \mathbf{a}^t\mathbf{x} = \sum_{i=1}^{q} a_i x_i \tag{2}$$

is built from the discriminant variables; then, the vector $\mathbf{a} = (a_1, \ldots, a_q)^t$ determines a direction in $\mathbb{R}^q$ and the code draws it for the cases $q = 2$ and $q = 3$.

26. Software programs usually work with the labels 0 and 1 for the two groups; we show 1 and 2 in the textual information; yet they are registered as 0 and 1 in the matrixes. In the scripts for new data, when $B > 1$ the majority-vote classifier is shown with the label 0 for ties. It provides useful information only when the parameter value has been optimized—otherwise all runs have the same labels.

27. Because of some possible effects—see section A.2.4—the user is recommended to consult the computational time of a run other than the first.

28. Some crashes may not be caused by the code. For example, in our methodology to classify functions, the code can manage as many derivatives or differentials as desired, and the limit can come from the hardware (memory), the software (we have tested the code with the orders `diffOrders=1:30`) or the theory (with many discriminant variables, the sample pooled covariance matrix can be singular).

29. MATLAB allows the user to rotate the figures by using the mouse, even in frames with several figures. For example, in figure 2 an approximately two-dimensional representation is obtained from the three--dimensional version.

3 Scripts

For each type of datum—time series or functions—there are three different scripts. Two of them are thought for researches to compare the behaviour of classification methods by using simulated and real data, while the third allows applied researchers or even teachers to label new data. All the scripts are different but share many code lines.

`SCRIPTsimulatedTS` and `SCRIPTsimulatedFD`

They allow applying our procedures—or others' methods—to two-group classification problems with simulated stochastic processes or stochastic functions, respectively. The scheme on the left of figure 1, in section 2.2, is implemented in these scripts. Their main characteristics are:

- Cross-validation is not implemented, as the user generates the data.
- For parameter optimization, the two-step approach of section 2.5.1 is implemented.
- For method selection, the two-step approach of section 2.5.2 is implemented.
- The depth-based robustifying technique of section 2.6.2 is not implemented, as the user generates the data.

If desired, the user can adapt the scripts for real data so that to used them with simulated data.

`SCRIPTrealTS` and `SCRIPTrealFD`

These scripts allow the application of classification procedures to two-group classification problems with real time series or functions, respectively. The scheme in the middle of figure 1, in section 2.2, is implemented in them. Notice that in general it is not possible (it would be too lasting) to exhaustively consider all possible subgroups of $m1$ and $m2$ elements out of $n1$ and $n2$. When $m1 = 1$ and $m2 = 1$, if `leaveOneOut = true`, all the elements are subsequently left out once, and only once; if `leaveOneOut = false`, an element of each group, chosen at random, is left out in each run. The main characteristics of this type of script are:

- Cross-validation is implemented.
- For parameter optimization, the two-step approach of section 2.5.1 is implemented.
- For method selection, the two-step approach of section 2.5.2 is implemented.
- The depth-based robustifying technique of section 2.6.2 is implemented.

`SCRIPTnewTS` and `SCRIPTnewFD`

The final application of any classification method requires working with few—even one—new unlabelled data, which makes it impossible to know the number of misclassified elements. That is why all the tasks depending on the estimated error rates must be carried out just after the inner loop, e.g., optimizing the parameter and selecting the method. The scheme on the right of figure 1, in section 2.2, is implemented in these scripts. This third kind of script is an adaptation of that for real data. The final testing samples can have any number of data, while large final testing samples are desired in the previous scripts for simulated and real data so as to estimated the error rates properly. The main characteristics of this type of script are:

- The final testing data are always the data to be allocated.
- The labels, rather than the error rates, are obtained in the outer loop. In the textual information, the code shows the classification obtained when the data are allocated in the testing sample of group 1. Additionally, as a control measure the new data are also introduced in the testing sample of group 2 so that to warn the user if any element is classified differently. On the other hand, if several methods provide the same result, the results of the method firstly called are shown.
- Cross-validation is implemented.
- For parameter optimization, the one-step approach of section 2.5.1 is implemented.

- For method selection, the one-step approach of section 2.5.2 is implemented.
- The depth-based robustifying technique of section 2.6.2 is implemented.
- The scheme allows the user to choose between two approaches:
  - Using all the training data only once to obtain the classification.
  - Splitting the training data in the inner loop to obtain several classifications of each new datum. When `B>1` our code shows the labels of the runs and the majority-vote classifier.

  In the second approach, the classifiers of the runs can be considered "weaker" than the unique classifier of the first approach, in the sense that they are based on fewer data (sample-size effects—see section A.1.2). Nevertheless, the second approach takes advantage from several facts: (i) for some runs the subsamples do not contain the contaminants and therefore are more reliable; (ii) there is a "natural" reduction of variability due to the combination of several "weaker" classifiers (run-averaging effect—see section A.1.5). Finally, when neither the parameter optimization nor the method selection is enabled, only the case `B=1` makes sense, as the same labels would be obtained for other runs, and the majority-vote classifier would be useless.

4 Functions

In this section, the functions programmed in this package are introduced. Apart from the following information, in the file where each function is written there is information on the input and output arguments, as well as some pieces of advice.

`classifyMod`

This function is a version of the MATLAB's function `classify` that provides the linear classifier as output.

`dataExercisesFD`

This function generates pairs of stochastic functional models. The parameters are $[c_{11}\ c_{12}\cdots]$ and $[c_{21}\ c_{22}\cdots]$, entered through `paramVector1` and `paramVector2`, respectively. To smooth the random noise, the first method of `smoothingFD` is called; this method needs the function `csaps` of the Curve Fitting Toolbox of MATLAB. If desired, the user can introduce contaminating functions directly in the scripts, as we have done in some examples of section 6.
When `numberExercise = 1`: The models are

$$\begin{array}{l} \mathcal{X}_e^{(1)} = c_{11}\cdot t + c_{12}\cdot U_e \\ \mathcal{X}_e^{(2)} = c_{21}\cdot t + c_{22}\cdot V_e \end{array}, \tag{3}$$

where U_e is a uniform random variable on the interval $(0,1)$, and V_e is a uniform random variable on the interval $(1/2, 3/2)$.

When `numberExercise = 2`: The models are

$$\begin{aligned} \mathcal{X}_e^{(1)} &= (c_{11} \cdot t + c_{12} \cdot U_e)^2 \\ \mathcal{X}_e^{(2)} &= c_{21} \cdot t^2 + c_{22} \cdot V_e \end{aligned}, \tag{4}$$

where U_e is a uniform random variable on the interval $(0, 1)$, and V_e is a uniform random variable on the interval $(0, 1)$.

When `numberExercise = 3`: The models are

$$\begin{aligned} \mathcal{X}_e^{(1)} &= (c_{11} \cdot t + c_{12} \cdot U_e)^2 + c_{13} \\ \mathcal{X}_e^{(2)} &= (c_{21} \cdot t + c_{22} \cdot V_e)^2 \end{aligned}, \tag{5}$$

where U_e is a uniform random variable on the interval $(0, 1)$, and V_e is a uniform random variable on the interval $(1/2, 3/2)$.

When `numberExercise = 4`: The models are

$$\begin{aligned} \mathcal{X}_e^{(1)} &= c_{11} \cdot t + c_{12} \cdot \epsilon_e^{(1)}(t) \\ \mathcal{X}_e^{(2)} &= c_{21} \cdot t + c_{22} \cdot \epsilon_e^{(2)}(t) \end{aligned}, \tag{6}$$

where $\epsilon_e^{(k)}$, $k = 1, 2$, are (the spline smoothing of) Gaussian processes with zero mean and covariance function $\sigma(t, s) = 0.25 \cdot \exp(-|t - s|^2)$. Similar models were considered in López--Pintado and Romo (2006), although as slopes they considered 4 and 7 instead of 4 and 4.5.

When `numberExercise = 5`: The models are

$$\begin{aligned} \mathcal{X}_e^{(1)} &= U_e h_1(t) + (1 - U_e) h_2(t) + c_{11} \cdot \epsilon_e^{(1)}(t) \\ \mathcal{X}_e^{(2)} &= V_e h_1(t) + (1 - V_e) h_3(t) + c_{21} \cdot \epsilon_e^{(2)}(t) \end{aligned}, \tag{7}$$

where U_e and V_e are uniform random variable on the interval $(0, 1)$, the processes $\epsilon_e^{(k)}$, $k = 1, 2$ are (the spline smoothing of) white noise and $h_1(t) = max(6/20 - |t - 10/20|, 0)$, $h_2(t) = h_1(t - 4/20)$ and $h_3(t) = h_1(t + 4/20)$, with $t \in [0, 1]$. Similar models were considered in Ferraty and Vieu (2003).

When `numberExercise = 6`: The models are

$$\begin{aligned} \mathcal{X}_e^{(1)} &= c_{11} \cdot t + c_{12} \cdot \epsilon_e^{(1)}(t) + c_{13} \cdot f(t) \\ \mathcal{X}_e^{(2)} &= c_{21} \cdot t + c_{22} \cdot \epsilon_e^{(2)}(t) \end{aligned}, \tag{8}$$

where $\epsilon_e^{(k)}$, $k = 1, 2$, are (the spline smoothing of) Gaussian processes with zero mean and covariance function $\sigma(t, s) = 0.25 \cdot \exp(-|t - s|^2)$, and $f(t)$ is the probability density function of a normal random variable with mean 0 and standard deviation 0.001.

When `numberExercise = 7`: The models are

$$\begin{aligned} \mathcal{X}_e^{(1)} &= c_{11} \cdot D_{c_{13}}(t) + c_{12} \cdot \epsilon_e^{(1)}(t) \\ \mathcal{X}_e^{(2)} &= c_{21} \cdot F_{c_{23}}(t) + c_{22} \cdot \epsilon_e^{(2)}(t) \end{aligned}, \tag{9}$$

where $D_N(t) = \frac{sin((N+1/2)t)}{sin(t/2)}$ is the Dirichlet's kernel, $F_N(t) = \frac{1}{N}\left(\frac{sin(Nt/2)}{sin(t/2)}\right)^2$ is the Fejér's kernel, and $\epsilon_e^{(k)}$, $k = 1, 2$, are (the spline smoothing of) white noise.

When `numberExercise = 60`: The models are

$$\begin{array}{l} \mathcal{X}_e^{(1)} = c_{11} \cdot t + c_{12} \cdot \epsilon_e^{(1)}(t) + c_{13} \cdot U_e \cdot f(t) \\ \mathcal{X}_e^{(2)} = c_{21} \cdot t + c_{22} \cdot \epsilon_e^{(2)}(t) \end{array} , \tag{10}$$

where $\epsilon_e^{(k)}$, $k = 1, 2$, are (the spline smoothing of) Gaussian processes with zero mean and covariance function $\sigma(t, s) = 0.25 \cdot \exp(-|t - s|^2)$, the function $f(t)$ is the probability density function of a normal random variable with mean 0 and standard deviation 0.001, and U_e is a uniform random variable on the interval $(0.8, 1)$.

When `numberExercise = 70`: The models are

$$\begin{array}{l} \mathcal{X}_e^{(1)} = c_{11} \cdot U_e \cdot D_{c_{13}}(t) + c_{12} \cdot \epsilon_e^{(1)}(t) \\ \mathcal{X}_e^{(2)} = c_{21} \cdot V_e \cdot F_{c_{23}}(t) + c_{22} \cdot \epsilon_e^{(2)}(t) \end{array} , \tag{11}$$

where $D_N(t) = \frac{sin((N+1/2)t)}{sin(t/2)}$ is the Dirichlet's kernel, $F_N(t) = \frac{1}{N}\left(\frac{sin(Nt/2)}{sin(t/2)}\right)^2$ is the Fejér's kernel, $\epsilon_e^{(k)}$, $k = 1, 2$, are (the spline smoothing of) white noise, and U_e and V_e are uniform random variables on the interval $(0.8, 1)$.

dataExercisesTS

This function generates pairs of stochastic process models. The parameters are $[c_{11}\ c_{12} \cdots]$ and $[c_{21}\ c_{22} \cdots]$, entered through `paramVector1` and `paramVector2`, respectively. When called with the parameter `contamType` the contaminating series are generated; alternatively, the user can introduce contaminating series directly in the scripts, as we have done in some examples of section 6.
When `numberExercise = 1`: The models are

Two autoregressive moving average, ARMA(1,1), processes $\{X_t\}$ and Y_t:

$$\begin{array}{ll} X_t^{(i)} = c_{11} \cdot X_{t-1}^{(i)} - c_{12}\epsilon_{t-1}^{(i)} + \epsilon_t^{(i)} & t = 1, \ldots, T \text{ and } i = 1, \ldots, I \\ Y_t^{(j)} = c_{21} \cdot X_{t-1}^{(j)} - c_{22}\epsilon_{t-1}^{(j)} + \epsilon_t^{(j)} & t = 1, \ldots, T \text{ and } j = 1, \ldots, J \end{array} , \tag{12}$$

where $\epsilon_t^{(i)}$ and $\epsilon_t^{(j)}$ are independent and identically distributed random variables following the standard normal distribution. Series are stationary in this case. **Contamination A**. It consists in switching the values of the two parameters. **Contamination B**. This type of contamination corresponds to a parameter value of $c_{11} = -0.9$ instead of the correct value. **Contamination C**. Equal to contamination B, but using a value +0.9 instead of -0.9.

When `numberExercise = 2`: The models are

Two processes composed half by white noise and half by an autoregressive process of order one:

$$\begin{array}{l} X_t^{(i)} = \begin{cases} \epsilon_t^{(i)} & \text{if } t = 1, \ldots, T/2 \\ X_t^{(i)} = c_{11} \cdot X_{t-1}^{(i)} - c_{12}\epsilon_{t-1}^{(i)} + \epsilon_t^{(i)} & \text{if } t = T/2 + 1, \ldots, T \end{cases} \\ Y_t^{(j)} = \begin{cases} \epsilon_t^{(j)} & \text{if } t = 1, \ldots, T/2 \\ Y_t^{(j)} = c_{21} \cdot X_{t-1}^{(j)} - c_{22}\epsilon_{t-1}^{(j)} + \epsilon_t^{(j)} & \text{if } t = T/2 + 1, \ldots, T \end{cases} \end{array} , \tag{13}$$

with $i = 1, \ldots, I$ and $j = 1, \ldots, J$. In this case, the series are made up of stationary parts, but the whole series are not stationary. **Contamination A**. It consists in switching the values of the two parameters in the ARMA half of the processes. **Contamination B**. This type of contamination corresponds to a parameter value of $c_{11} = -0.9$. **Contamination C**. Equal to contamination B, but using a value +0.9 instead of -0.9.

When `numberExercise = 3`: The models are

The stochastic models in both classes are slowly time-varying second order autoregressive processes:

$$\begin{array}{ll} X_t^{(i)} = \phi_1(t, c_{11}, c_{12}) \cdot X_{t-1}^{(i)} - c_{13} \cdot X_{t-2}^{(i)} + \epsilon_t^{(i)} & t = 1, \ldots, T \\ Y_t^{(j)} = \phi_2(t, c_{21}, c_{22}) \cdot Y_{t-1}^{(j)} - c_{23} \cdot Y_{t-2}^{(j)} + \epsilon_t^{(j)} & t = 1, \ldots, T \end{array}, \tag{14}$$

with $i = 1, \ldots, I$, $j = 1, \ldots, J$ and $\phi_k(t, c_{k1}, c_{k2}) = c_{k1} \cdot [1 - c_{k2} \cos(\pi t/1024)]$ (see Huang et al. [2004]). Note that a coefficient of the autoregressive structure is not fixed and it changes with time, making the processes nowhere stationary. **Contamination A**. The parameter value $c_{12} = 0.5$ is substituted by the value 0.2. **Contamination B**. This type of contamination corresponds to a parameter value $c_{12} = -0.9$ instead of the correct value. **Contamination C**. Equal to contamination B, but using a value +0.9 instead of -0.9.

`distancesFD`

This function implements distances between two functions, $\chi_1(t)$ and $\chi_2(t)$.
When `numberFDdistance = 0`: The L_∞ distance is used

$$d(\chi_1, \chi_2) = max_t\{|\chi_1(t) - \chi_2(t)|\} \tag{15}$$

When `numberFDdistance = 1`: The L_1 distance (default) is used

$$d(\chi_1, \chi_2) = \int_{-1/2}^{+1/2} |\chi_1(t) - \chi_2(t)| dt \tag{16}$$

When `numberFDdistance = 2`: The L_2 distance is used

$$d(\chi_1, \chi_2) = \left(\int_{-1/2}^{+1/2} (\chi_1(t) - \chi_2(t))^2 dt \right)^{1/2}. \tag{17}$$

The user can easily add other distances. To exploit the speed of the matricial calculations, the argument `X` of our MATLAB function can be a matrix.

`fromFDtoCLASS`

It is a function to facilitate the application of any functional data classification method. Only those for our proposals are implemented by default, but the user can easily add others—see section 2.4.
When `numFDtoCLASSmethod = 1`: It is applied the rule

$$C(e) = \begin{cases} 1 & \text{if} \quad x < 0 \\ 2 & \text{if} \quad x \geq 0 \end{cases} \tag{18}$$

based on the the univariate discriminant variable

$$x = d(\chi_X, \mathcal{R}^{(1)}) - d(\chi_X, \mathcal{R}^{(2)}), \tag{19}$$

where, for our procedures, χ_X is the function constructed from the series X and $\mathcal{R}^{(k)}$ is the representative function—the mean function, for this method—of the training sample of the k-th group. This discriminant variable, used in our method *DbC*, were defined in Alonso et al. (2008).

When `numFDtoCLASSmethod = 2`: It is applied the rule based on the univariate variable of the form 19 but with the α-trimmed mean, instead of the mean, as representative function for each group. This variable, used in our method *DbC*-α, was also introduced in Alonso et al. (2008).

`fromFDtoMV`

For any function χ, this function calculates a vector of discriminant variables

$$\mathbf{x} = (x_1, x_2, \ldots, x_q)^t. \tag{20}$$

This reduction of the data dimension implies a loss of information, but also a better framework to learn in, since sample sizes are bigger than data dimension—the *curse of dimensionality* is avoided—and the risk of overfitting is smaller.

When `numberFDtoMVmethod = 1`: The discriminant variables are

$$x_i = d(D^{i-1}\chi, \overline{D^{i-1}\chi}^{(1)}) - d(D^{i-1}\chi, \overline{D^{i-1}\chi}^{(2)}), \tag{21}$$

for $i = 1, 2, \ldots, q$, where χ is the function and $\overline{D^{i-1}\chi}^{(k)} = n_k^{-1} \sum_{e=1}^{n_k} D^{i-1}\chi_e^{(k)}$, $k = 1, 2$. The parameter `diffsMode` allows choosing between $D^i\chi = \frac{d^i\chi}{dt^i}$ or $D^i\chi = d^i\chi$ (the differential is defined as $d^i\chi = \chi^{i)}dt^i = \frac{d^i\chi}{dt^i}dt^{i)}$. The latter case is a redimension of the former.

The term $d\chi(t)$ has the same unit of measure as $\chi(t)$, and the term dt the same as t. Therefore, supposing that a variable t and a function $\chi(t)$ are not dimensionless (scalars without unit of measure) and they do not have the same unit of measure, the derivative $D^1\chi(t) = d\chi(t)/dt$ has different dimension to its original function. As a consequence, all the discriminant variables are dimensionless only when t and $\chi(t)$ are dimensionless or have the same dimension. When the derivatives can take large values (for example, if narrow picks are considered in the models), the differentials can avoid some computational problems, since they are not divided by a tiny quantity. Besides, the derivatives imply more computes and memory. These discriminant variables, used in our methods *WI* and *WD*, were introduced in Alonso et al. (2012).

`fromMVtoCLASS`

It is a function to facilitate the application of any multivariate classification method to the vector of discriminant variables

$$\mathbf{x} = (x_1, x_2, \ldots, x_q)^t. \tag{22}$$

There are some auxiliary functions in the same file—see section 2.4. On the other hand, in Alonso et al. (2012) we defined *DFMi* as the method based on the i-th discriminant variable, that is, the rule

$$C(e) \;=\; \begin{cases} 1 & \text{if} \quad x_i < 0 \\ 2 & \text{if} \quad x_i \geq 0 \end{cases} \tag{23}$$

These classification methods are directly applied in the scripts, not through any mapping function like this.
When `numberMVtoCLASSmethod = 0`: The identity is applied—see section 2.4.2.
When `numberMVtoCLASSmethod = 1`: The Fisher's Linear Discriminant Analysis (LDA) is applied. This method is based on the objective function

$$[\mathbf{a}^t(\overline{\mathbf{x}}^{(1)} - \overline{\mathbf{x}}^{(2)})]^2, \tag{24}$$

implemented in our MATLAB function `objectiveFunction` (in fact, it implements $-[\mathbf{a}^t(\overline{\mathbf{x}}^{(1)} - \overline{\mathbf{x}}^{(2)})]^2$ so that to use the MATLAB's `fmincon`, which finds minima). This step is used by our method *WI*, proposed in Alonso et al. (2012).
When `numberMVtoCLASSmethod = 2`: It is applied the Fisher's Linear Discriminant Analysis with the usual restrictions to determine a unique eigenvalue (from the infinite number of them):

$$\begin{cases} \mathbf{a}^t\mathbf{W}\mathbf{a} = 1 \\ \mathbf{a} \geq \mathbf{0} \end{cases}, \tag{25}$$

implemented in our MATLAB function `constraintsFunction`. This step is used by our method *WD*, proposed also in Alonso et al. (2012).
When `numberMVtoCLASSmethod = 3`: The K-Nearest Neighbours (*KNN*) approach is applied. Our code calls the MATLAB's function `knnclassify` with k given by the parameter `numberNeighbours`; the default distance (Euclidean) and rule (majority rule with nearest tie-break) are considered. This function belongs to the Bioinformatics Toolbox of MATLAB.

`fromTStoFD`

For each time series X, the methods in this function calculate a function χ_X. Concretely, when time series are split into blocks the functions take the form $\chi_X = (F_X^{(1)} \dots F_X^{(k)})$, where $F_X^{(j)}$ is a function constructed from the j-th block of the series X (analogously for the samples of Y). Let $I_T(t_k) = \sum_{h=-(T-1)}^{(T-1)} \hat{\rho}_h \exp(-2\pi i h t_k)$ be the periodogram of a time series, where $\hat{\rho}_h$ is an estimator of the autocorrelation function of the series (see, for example, Priestley [1981]). The following transformations are implemented by default.
When `numberTStoFDmethod = 0`: The identity is implemented—see section 2.4.2.
When `numberTStoFDmethod = 1`: The normalized integrated periodogram (default option) is constructed

$$F_T(t_k) = \sum_{i=1}^{k} I_T(t_i) / \sum_{i=1}^{m} I_T(t_i), \tag{26}$$

This function is used by our methods *DbC* and *DbC*-α, introduced in Alonso et al. (2008).
When `numberTStoFDmethod = 2`: The nonnormalized integrated periodogram is constructed

$$F_T(t_k) = \sum_{i=1}^{k} I_T(t_i), \tag{27}$$

This function is also used by our methods *DbC* and *DbC*-α.
When `numberTStoFDmethod = 3`: The periodogram is constructed

$$I_T(t_k), \tag{28}$$

When `numberTStoFDmethod = 4`: The logperiodogram is constructed

$$L_T(t_k) = log(I_T). \tag{29}$$

Previous definitions have been given for a finite set of values, since this is the only case we can consider in practice; in fact, choosing the set of *Fourier frequencies* provides among other advantages some simplification in the algebra—see, for example, Priestley (1981). These definitions could have been given for $\lambda \in [-\pi, +\pi]$.

We include here some explanations taken from Casado (2010): *The normalized version of the cumulative periodogram takes into account the shape of the curves more than the nonnormalized version, which also considers the scale.*

In our case, we propose using the normalized version when the graphs of the functions of the different groups tend to intersect and there is no clear scale pattern, and using the nonnormalized one when the graphs do not tend to intersect.

Some of the advantages of using the integrated periodogram are: it is a nondecreasing and quite smooth curve; it has good asymptotic properties (for example, while the periodogram is an asymptotically unbiased but inconsistent estimator of spectral density, the integrated periodogram is a consistent estimator of spectral distribution); although, in practice, for stationary processes the integrated spectrum is usually estimated via the estimation of the spectrum, from a theoretical point of view, spectral distribution always exists, whereas spectral density exists only under absolutely continuous distributions; finally, from a theoretical point of view, the integrated spectrum completely determines the stochastic processes.

Since the periodogram is defined only for stationary stochastic processes, to be able to classify nonstationary time series, we shall consider locally stationary series. With this assumption we can split them into blocks, compute the integrated periodogram of each block and merge these periodograms into a final curve; hence, we approximate the locally stationary processes by piecewise stationary processes.

The user can easily implement other functional data, as well as functional transformations of the previous ones. There are some auxiliary functions in the same file: `toIntegration` applies the MATLAB's function `cumsum` to the rows of a matrix, which corresponds to integrate; `toPeriodogram` applies the MATLAB's function `periodogram` to the rows of a matrix, which corresponds to compute the periodogram; finally, `toNormalization` divides each function (row) by its maximum, which is the last value when the function is increasing (so are the integrated periodogram and `cumsum`).

`funcAR2tv`

This function generates the time-varying autoregressive stochastic processes of `dataExercisesTS`.

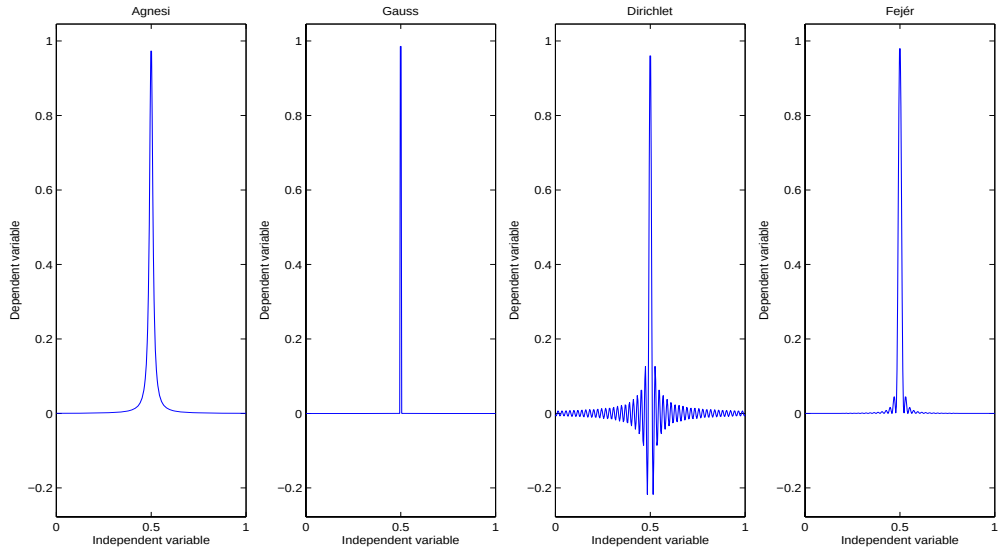
`funcARMApq`

This function generates the autoregressive moving average processes of `dataExercisesTS`. It can be used to generate ARMA(p,q) processes outside the framework of this package.

`functionRFC`

This function applies the robust functional classification. It can be used with any functional data, outside the framework of this package.

Figure 3: Plot with the four possible kinds of peak



`generatePeak`

This function adds to a function a deterministic peak centered in the middle value of the independent variable t. Four possible types of smooth (differentiable) peak are possible—see figure 3.
When `peakType = 'Agnesi'`: The Agnesi's function is added

$$a_d(t) = \frac{d^3}{d^2 + t^2} \tag{30}$$

When `peakType = 'Gauss'`: The Gauss's function is added

$$f_\sigma(t) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{t^2}{2\sigma^2}} \tag{31}$$

When `peakType = 'Dirichlet'`: The Dirichlet's function is added

$$D_N(t) = \frac{sin((N + 1/2)t)}{sin(t/2)} \tag{32}$$

When `peakType = 'Fejer'`: The Fejér's function is added

$$F_N(t) = \frac{1}{N} \left(\frac{sin(Nt/2)}{sin(t/2)} \right)^2 . \tag{33}$$

The parameter in the subindex is `peakParam`, and it allows the user to control the width of the peak. To control the height of the peak, this function divides the function by its maximum (value at 0) and multiplicates it by the parameter `peakHeightCoeff`. That is, if $c_{peakParam}(t)$ represents any of the previous expressions, this function generates

$$p_{peakParam}(t) = \frac{c_{peakParam}(t - T/2)}{c_{peakParam}(0)} \cdot peakHeightCoeff. \tag{34}$$

The final height of the peak is `peakHeightCoeff`. The user can obtain the "original curves," but centered in zero, by choosing $peakHeightCoeff = c_{peakParam}(0)$. This kinds of peak—and the random functions implemented in our models—may appear in areas like Physics or Engineering.

plotDiscrimFunctions

This function plots the values of a univariate discriminant function $y = \mathbf{a}^t\mathbf{x} = \sum_{i=1}^{p} a_i x_i$. The direction determined by the vector of coefficients $\mathbf{a} = (a_1, \ldots, a_q)^t$ is plotted, for the two- and three-variable cases, by the function `plotDiscrimVariables`.

plotDiscrimVariables

This function plots the values of the univariate discriminant variables in the vector $\mathbf{x} = (x_1, x_2, \ldots, x_q)^t$. For time series, the unique variable is defined as described in the function `fromFDtoCLASS`. For functional data, the variables are defined as described in the function `fromFDtoMV`. When a discriminant function

$$y(\mathbf{x}) = \mathbf{a}^t\mathbf{x} = \sum_{i=1}^{q} a_i x_i \tag{35}$$

is built from the discriminant variables, the vector $\mathbf{a} = (a_1, \ldots, a_q)^t$ of coefficients determines a direction in $\mathbb{R}^q$, and for two or three variables the code draws the version of this direction that contains the global centroid of the data.

plotFunctions

This function plots functional data. The user can choose the number of functions of each group to be plotted.

plotResults

For each (global) classification method, this function creates some figures with the results: error rates, computational times, minimizing-power measures, coefficients, et cetera.

plotTimeSeries

This function plots time series. The user can choose the number of series of each group to be plotted.

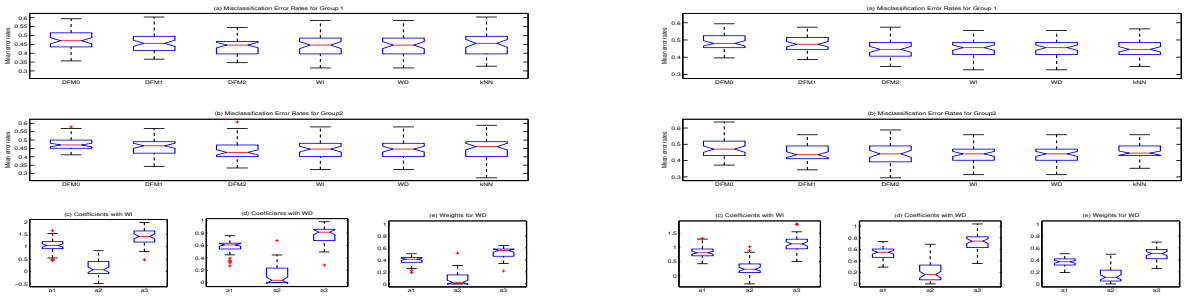
referenceElement

This function includes methods to construct a representative element of a group from a sample. By default, the usual mean is implemented (it can be applied to vectors, time series and functions). Additionally, if atypical functional data are expected the user can select the α-trimmed mean—see López-Pintado and Romo (2006)—although it is time-consuming. For example, in the simulation exercise 6, the use of the trimmed mean reduces the variability of the extreme values of both the error rates and the coefficients, see figure 4; with other data or models, the reference function can improve the results in a higher degree.

smoothingFD

This function includes methods to smooth functional data. A classical method based on cubic splines is implemented. This method needs the function `csaps` of the Curve Fitting Toolbox of MATLAB.

Figure 4: Results with the mean (left) and the trimmed mean (right)



`toLaTeXmatrix`

This function generates (in the screen, but the user can call it while a plain text is being created by the `diary` command of MATLAB) the LaTeX code of a generic table, so that to copy and paste it to a LaTeX file. The user can customize this function.

`trimFunction`

For a sample of functions and a value of α (`alpha`, in the code), this function provides the α-trimmed mean of the functions. It also provides the positions (rows) of the deepest functions. The user can edit this function for the values of the depth to be provided. Besides, this function can be used outside the framework of this package, for example to identify the deepest elements and to construct other robust quantities, different to the mean. There is one auxiliary function in the same file, `modBandDepth2`, which implements the modified band depth with $j = 2$—see López-Pintado and Romo (2006). Finally, the user can implement other functional depth measures.

5 Parameters

5.1 Setting Parameters

`dataFigures`

Logical parameter to indicate whether to plot, during the first run, the crude data, their transformation and the discriminant variables. See section 2.3. (Default value: `dataFigures = true`)

`modelsJointly`

Logical parameter to indicate whether to plot the series or the functions of the two groups separately or jointly. (Default value: `modelsJointly = false`)

`numElementsToPlot`

Number to indicate the number of elements—series or functions—of each sample that will be included in the figures; when plotting the discriminant variables, all the elements are considered. (Default value: `numElementsToPlot = ceil(m1*0.05)`, that is, 5% of `m1`)

`percentMargin`

Number to control, in the figures, the distance from the most external values to the axes, e.g., as in

```
set(gca, 'XLim', [xMin-xRange*percentMargin/100 xMax+xRange*percentMargin/100])
```

(Default value: `percentMargin = 5`, that is, 5% of the range of values)

`resultsFigures`

Logical parameter to indicate whether to represent a boxplot with the misclassification error rates and, when different parameter values have been tried, a figure with the evolution of the mean error rates as a function of these values. (Default value: `resultsFigures = true`)

`saveResults`

Logical parameter to indicate whether to save in a file the main output information of the runs: error rates, sample sizes, exercise, parameters, et cetera. The information in the file is sufficient for a future analysis—textual and graphical—of the results. (Default value: `saveResults = false`)

`verbose`

Logical parameter to indicate whether to show some textual information during the computes. Some important information is given even if this parameter takes the value `false`—see section 2.9. (Default value: `verbose = true`)

5.2 Statistical Parameters

`alpha`

Proportion of data that the functional depth leaves out to ignore the least representative elements. If few of such elements are expected, a smaller value should be considered for a better use of the available samples. (Default value: `alpha = 0.2`)

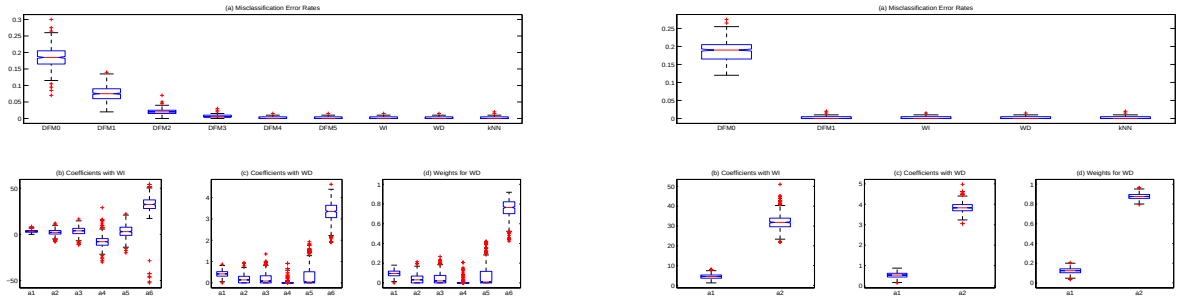
`B`

Number of runs in the outer or main loop. It must be neither too small nor too big—see section A.1.5. (Default value: `B = .` In the scripts for real or new data, when `leaveOneOut = true` the value `B = n1 + n2` is set)

`Bparam`

Number of runs in the inner or nested loop, where the parameter value is optimized or the method is selected. It must be neither too small nor too big—see section A.1.5. (Default value: `Bparam = .` In the scripts for real or new data, when `leaveOneOutParam = true` the value `Bparam = n1param + n2param` is set)

Figure 5: Error rates with `diffOrders = [0 1 2 3 4 5]` and `[0 5]`, respectively ($B = 500$)



`contamination`

Logical parameter to indicate whether to introduce contaminating elements during the generation of the simulated data (they can also be directly introduced in the scripts). The code introduces the simulated contaminants in the sample of group 1, but in real or new data they can belong to any sample. (Default value: `contamination = false`)

`contamType`

Letter to indicate the type of contamination, previously implemented in the models, to be introduced during the simulation of the data. In our case, this parameter can take the value 'A', 'B' or 'C'. (Default value: `contamType = ''`)

`depthBasedRobustness`

Logical parameter to indicate whether to apply the depth-based robustifying technique described in section 2.6.2. When activated, the robust method DbC-α is automatically disabledM; the user can modify this characteristic of the code, as we have done for some simulation exercises of section 6. (Default value: `depthBasedRobustness = false`)

`diffOrders`

Numerical vector to indicate the orders of the derivatives or differentials that are considered initially. The crude functions—value 0—can be excluded, although in such a case functions differing in a constant would be indistinguishable (this might be desirable sometimes). It holds that the number of discriminant variables q is `length(diffOrders)`. On the other hand, the highest possible order is technically determined by the number of points T, although before this occurring the quality of the derivatives or differentials decreases and the variables loose their discriminant power. Our methods are not affected by variables corresponding to unnecessary derivatives or differentials—we may be working with noise sometimes—as they will not have information useful to classify and the multivariate methods will discard them. This vector allows the user to consider only some derivatives or differentials possibly nonconsecutive, which can reduce the computes in some cases, e.g., for models with a tendency in the discriminant power of the successive derivatives or differentials—see figure 5. On the convenience of considering many derivatives or differentials, figure 6 shows the strong stability of our procedures while figure 7 gives support to the possible convenience. (Default value: `diffOrders = []`)

Figure 6: Boxplots when $p = 21$ (with $B = 500$)

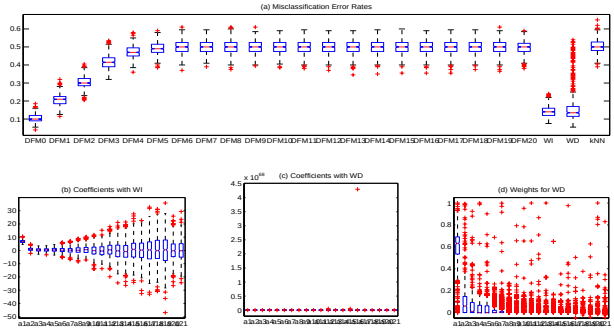
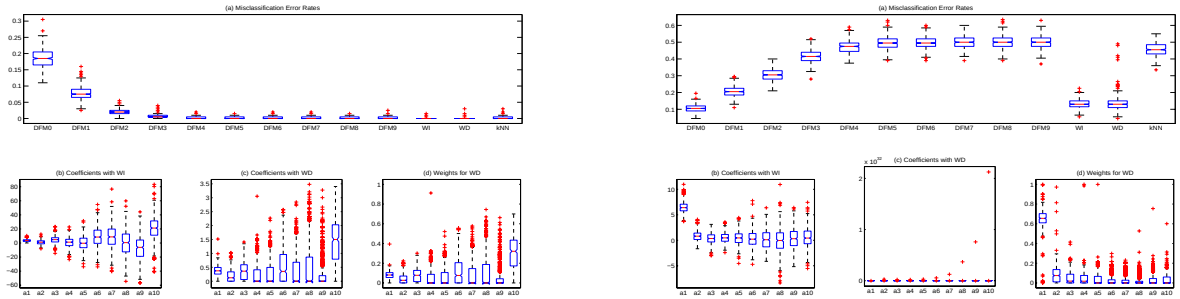


Figure 7: Two two-group classification problems with different performance when many derivatives or differentials are considered (with $B = 500$)



`diffsMode`

In our methods for classifying functional data, *WI* and *WD*, string parameter to choose between the derivatives or the differentials in the i-th discriminant variable—see the function `fromFDtoMV`. This parameter can take the value `'derivatives'` or `'differentials'`. (Default value: `diffsMode = 'derivatives'`)

`discretenessMargin`

Number to set the interval `[minimum, minimum*(1+discretenessMargin)]` in the two-step approaches of sections 2.5.1 and 2.5.2. For example, the value 0.25 corresponds to an interval whose length is 25% of the minimum. The user can change the margin or disable the two-step approach by setting `discretenessMargin = 0`. (Default value: `discretenessMargin = 0.25`)

`dyadicSplits`

Number of dyadic splits to set the default value of `numberBlocks`. For example, if `dyadicSplits = 3` then `numberBlocks = [1 2 4 8]`. If the user chooses the values of `numberBlocks`, this parameter is ignored. (Default value: `dyadicSplits = 0`)

`leaveOneOut`

Logical parameter to indicate, when $m1 = 1$ and $m2 = 1$, whether to apply exhaustive leave-one-out cross--validation; that is, whether each datum is subsequently left out instead of choosing a datum at random in each run. (Default value: `leaveOneOut = false`)

`leaveOneOutParam`

In the inner loop, logical parameter to indicate, when $m1param = 1$ and $m2param = 1$, whether to apply exhaustive leave-one-out cross-validation. (Default value: `leaveOneOutParam = false`)

`m1` and `m2`

Numbers with the testing sample sizes of the two groups in the outer loop. (Default values: `m1 =` and `m2 = m1`. In the scripts for real or new data, when `leaveOneOut = true` the values `m1=1` and `m2=1` are set.)

`m1param` and `m2param`

Numbers with the testing sample sizes of the two groups in the inner loop. (Default values: `m1param =` and `m2param = m1param`. In the scripts for real or new data, when `leaveOneOutParam = true` the values `m1param=1` and `m2param=1` are set.)

`methodSelection`

Logical parameter to indicate whether to select, in the inner loop, only the method and the parameter value that best fit the data (in the sense of minimizing the estimated overall misclassification error rates) and therefore is expected to minimize the rate in the outer loop too. The user is given the minimizing-power measure of the methods (see section 2.5.3 for details). When `paramOptimization = true` but `methodSelection = false`, the user is given an estimation of the error rates as if the method had been optimized. When `methodSelection = true`, the code sets `paramOptimization = true`. The method selection can be applied even if the parameter values vector has one value only. The code sets `methodSelection = false` if there is only one global method. The selection process has been implemented in two slightly different approaches—see section 2.5.2 for details. (Default value: `methodSelection = false`)

`n1` and `n2`

Numbers with the training sample sizes of the two groups in the outer loop. In the inner loop of the scripts for real or new data, if `paramDataReuse = false` and `n1 = -1 = n2` the code splits the training samples in two subsamples of the same size, with 50% of the data (if possible): one is the final testing sample and the other is used for optimizing the parameter or selecting the method. (Default values: `n1 =` and `n2 = n1`)

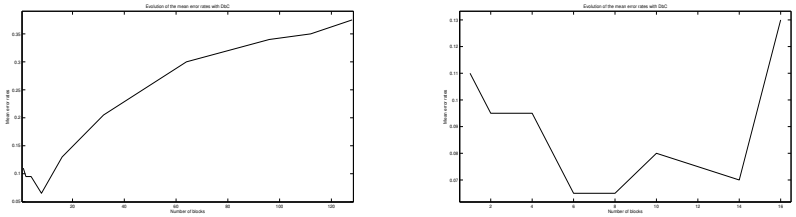
`n1c`

In the outer loop, number of contaminating series to be introduced in the final training sample of group 1 when simulating data. (Default value: `n1c = ceil(n1*0.07)`, that is, 7% of `n1`)

`n1cParam`

In the inner loop, number of contaminating series to be introduced in the training sample of group 1 when simulating data. (Default value: `n1cParam =`)

Figure 8: Fewer values of `numberBlocks` can be considered in a second execution of the code



`n1param` and `n2param`

In the inner loop, numbers with the training sample sizes of the two groups. (Default values: `n1param =` and `n2param = n1param`)

`namesFDtoCLASSmethods`

Array with the names of the global functional data classification methods to be used in the figures. Its length must be the number of rows of `numbersFDtoCLASSmethods`. (Default value: `namesFDtoCLASSmethods = {'WI', 'WD', 'dKNN'}`)

`namesTStoCLASSmethods`

Array with the names of the global time series classification methods to be used in the figures. Its length must be the number of rows of `numbersTStoCLASSmethods`. Method DbC-α is automatically disabled—from `numbersTStoCLASSmethods` and this array—when `depthBasedRobustness = true`. The user can modify this behaviour. (Default value: `namesTStoCLASSmethods = {'DbC' 'DbC-alpha'}`)

`numberBlocks`

Numerical vector with the number of blocks into which series will be subsequently split. The highest possible number of blocks is technically determined by the length T, although before this occurring the quality of the built functions decreases and the variables loose their discriminant power. The user can select the desired initial values, for example `numberBlocks = [4]` or `numberBlocks = [1 3 6]`. This vector allows the user to consider only some number of blocks possibly nonconsecutive, which can reduce the computes in some cases. For stationary series, the value of `numberBlocks` must be 1; only the nonstationary series (piecewise stationary series, for example) need splitting into blocks. If `resultsFigures = true`, the code plots the mean estimated misclassification error rates as a function of `numberBlocks`. This allows the user to choose visually the value that minimizes the error so as to execute the code again with a more appropriate range of values for `numberBlocks`—see figure 8 in section 5.2. This minimizing value may depend on T—see section A.1.4. (Default value: `numberBlocks = [1 2 ... 2.^dyadicSplits]`)

`numberExercise`

Number of the simulation exercise to be executed. Some default models are programmed, although the user can create new cases in `dataExercisesFD` and `dataExercisesTS`. (Default value: `numberExercise = `)

`numberNeighbours`

Number of neighbours in the k-nearest neighbours method. (Default value: `numberNeighbours = 3`)

`numbersFDtoCLASSmethods`

Matrix with the global functional data classification methods. In each row:

- The first number indicates the method of `fromFDtoMV` to be used
- The second number indicates the method of `fromMVtoCLASS` to be used
- The third number indicates the functional distance of `distancesFD` to be used
- The forth number indicates the functional reference of `referenceElement` to be used

Our procedures are implemented and called by default, but the users can call only one of them or other methods—see section 2.4. Method *WI* is determined by 1 1 1 1, *WD* by 1 2 1 1 and *dKNN* by 1 3 1 1. (Default value: `numbersFDtoCLASSmethods = [1 1 1 1; 1 2 1 1; 1 3 1 1]`)

`numbersTStoCLASSmethods`

Matrix with the global time series classification methods. In each row:

- The first number indicates the method of `fromTStoFD` to be used
- The second number indicates the method of `fromFDtoCLASS` to be used
- The third number indicates the functional distance of `distancesFD` to be used
- The forth number indicates the functional reference of `referenceElement` to be used

Our procedures are implemented and called by default, but the users can call only one of them or other methods—see section 2.4. Method *DbC*-α is automatically disabled—from this matrix and from the string `namesTStoCLASSmethods`—when `depthBasedRobustness = true`. The user can modify this behaviour. Method *DbC* is determined by 1 1 1 1 and *DbC*-α by 1 2 1 1. (Default value: `numbersTStoCLASSmethods = [1 1 1 1; 1 2 1 1]`)

`paramDataReuse`

Logical parameter to indicate whether to use the training data of the main loop for both optimizing the parameter value (in the inner loop) and estimating the final error rates (in the outer loop). See sections 2.2 and 3. (Default value: `paramDataReuse = false`)

`paramOptimization`

Logical parameter to indicate whether to optimize, in the inner loop, the parameter value of each method that minimizes the estimated overall error rate and therefore is expected to minimize the error rate in the outer loop too. The user is given the minimizing-power measure of the values (see section 2.5.3 for details). When disabled, the user is given an estimation of the error rates as if the parameter had been optimized. This option is automatically disabled when `length(numberBlocks)=1`. When `methodSelection = true`,

the code sets `paramOptimization = true`. The optimization process has been implemented in two slightly different approaches—see section 2.5.1 for details. (Default value: `paramOptimization = false`)

`paramVector1` **and** `paramVector2`

For simulated data, vectors with the parameters of the two models. (Default value: `paramVector1 = []` and `paramVector2 = []`)

`T`

For functional data, number of equispaced points in [0,1] at which the functions are evaluated; the points themselves are `t = 0:1/(T-1):1` and the relation between T and t is important to compute the derivatives and the differentials, since a technique for evenly spaced data is applied. For time series, length of the processes and series; it is recommended to be a power of two (see section 2.8). (Default values: `T = `)

`variaCutoff`

Number to select the cutoff for the variability, when approximately constant variables are ignored before applying the multivariate linear discriminant analysis in our algorithms. (Default value: `variaCutoff = sqrt(eps)`)

6 Examples

In Alonso et al. (2008, 2012) our proposals were applied to real data. In this section, several simulation exercises are run, while others are suggested. The calls we present here will include the parameters that are important or take a value different to that assigned by default. The output would be different for other calls; and even slightly different numerical quantities would be obtained—because of the randomness of the samples—for the calls we have used here. Besides, we have not included all the text and figures the code generates.

It is worth highlighting that with these exercises we want both to show the output of the code and to justify some concepts, while they are not a complete study. In some of them we are not directly interested in the estimated overall misclassification error rates, so we have considered a moderate number of runs B without caring whether the standard error of the estimation, measured as the sample standard deviation divided by $\sqrt{B}$, has the same order of magnitude than the mean of the estimations, that is, the uncertainty is similar to the estimation itself—see section A.1.5.

Since the functional classification methods *WI*, *WD* and *dKNN* are based on just the same values for the discriminant variables, the option `paramOptimization = true` is interesting to select the differentiation order that best classify the data but not to show some effects. Hence, for this task methods *DbC* and *DbC*-α are more appropriate—on the one hand, they do not work with just the same discriminant variables and, on the other hand, they are not so fast.

6.1 Times Series

6.1.1 Simulation Exercise E1ts: Output of the Code. Methodology Effect

In this section, the output of `SCRIPTsimulatedTS` is shown for the models given by expressions 14 (both are nowhere stationary processes): `numberExercise = 3`, `paramVector1 = [0.8 0.5 0.81]`, `paramVector2 = [0.8 0.3 0.81]`, `n1 = 50`, `n2 = n1`, `m1 = 75`, `m2 = m1`, `T = 512`, `contamination = true`, `contamType = 'B'`, `n1c = 2`, `alpha = 0.2`, `B = 100`, `numbersTStoCLASSmethods = [1 1 1 1; 1 2 1 1]`, `namesTStoCLASSmethods = {'DbC', 'DbC-alpha'}`, `numberBlocks = [1 2 4]`, `paramOptimization = true`, `n1param = 20`, `n2param = n1param`, `m1param = n1-n1param`, `m2param = n2-n2param`, `Bparam = 10`, `n1cParam = 1`, `methodSelection = true`.

Textual Information

When `verbose = true`, the following text is shown. During the runs

```
START

Generating the data...

Starting the iterations...

    ITERATION: 1
     ⋮
    ITERATION: 3
     Optimization loop...
     ...end of the optimization loop
     Information for method DbC-alpha
      Error rates for group 1:   0.013333
      Error rates for group 2:   0.013333
      Overall error rates:   0.013333
      Time spent for each value of the parameter values:   1.325  1.897  7.156
      Joint time spent for all values of the parameter values:   10.378
    ITERATION: 4
     ⋮

...ending the iterations
```

and, as a summary of the results, the code shows

```
MISCLASSIFICATION ERROR RATES (with both parameter optimization and method selection)
-> Mean of the estimated error rates for group 1:
    0.0248
  -> Standard deviation:
     0.018669
  -> Standard error:
```

```
     0.0019
-> Mean of the estimated error rates for group 2:
    0.0204
  -> Standard deviation:
     0.01605
  -> Standard error:
     0.0016
-> Mean of the estimated overall error rates:
    0.0226
  -> Standard deviation:
     0.011996
  -> Standard error:
     0.0012
METHOD: DbC
    MINIMIZING-POWER OF THE PARAMETER VALUES
    -> Mean:
       0  0  0.01
    COMPUTATIONAL TIMES
    -> Mean time spent for each value of the parameter values:
       0.58744  1.0898  2.1211
    -> Mean joint time spent for all values of the parameter values:
       3.7992
METHOD: DbC-alpha
    MINIMIZING-POWER OF THE PARAMETER VALUES
    -> Mean:
       0  0.02  0.98
    COMPUTATIONAL TIMES
    -> Mean time spent for each value of the parameter values:
       1.2999  1.922  6.0111
    -> Mean joint time spent for all values of the parameter values:
       9.2342
MINIMIZING-POWER OF THE METHODS
-> Mean:
    0.01  0.99

Result figures...

STOP    (And remember:  'Everything and nothing is possimpible', Barney Stinson)
```

From the previous information, we can see that 99 times out of the 100 the robust algorithm DbC-α has been selected, which means that the contamination in the samples is not negligible. The algorithm DbC-α has almost always provided the smallest error rates when series have been split in four blocks. To avoid

Figure 9: **E1ts**. Time series of the two populations

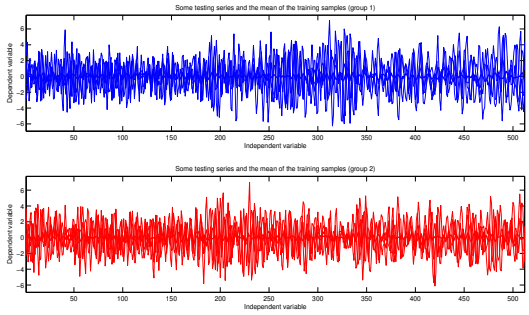
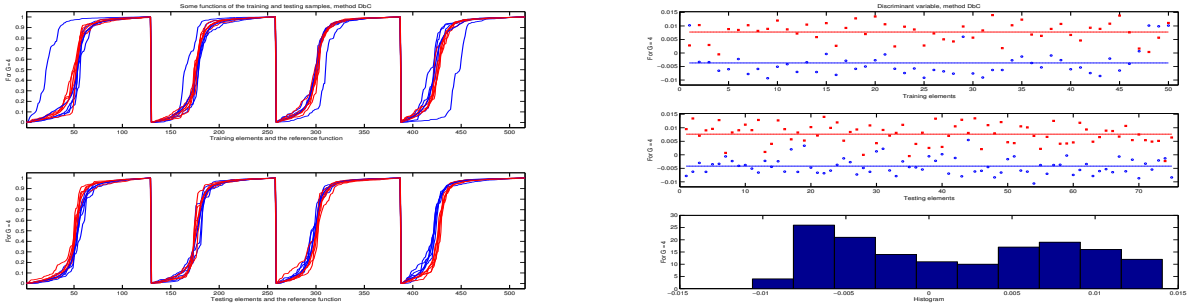


Figure 10: **E1ts**. For four blocks: functional data and discriminant variable



any possible beginner effect—see section A.2.4—we have provided the output of the third run rather than the first. Finally, since $B = 100$ the order of magnitude of the standard error is one unit smaller than the estimations of the mean error rates.

Graphical Information

When the inner loop is executed for optimizing the parameter or selecting the method, as different values and methods can be chosen in different runs, the descriptive or prospective figures are not generated even if `dataFigures = true`. Only the crude data are plotted (figure 9). Now we include as an example the functional data and the discriminant variable that would be generated for each method and each number of blocks (figure 10). The plots with the results (figures 11 and 12) are based on the B runs. During the first run, result figures are also shown (we do not include them here) if `dataFigures = true`, which can be used for prospective purpose before considering a big value for B. It can also be seen that the robust method is automatically selected almost always. As expected, for these nowhere stationary processes, the best results are obtained for four blocks. Besides, the histograms and the minimizing-power measure agree to identify the variable with the highest discriminant power. The computational time depend nonlinearly

Figure 11: **E1ts**. Estimated misclassification error rates and minimizing-power measure of the methods

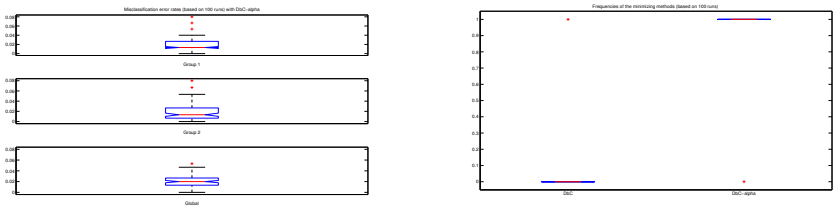
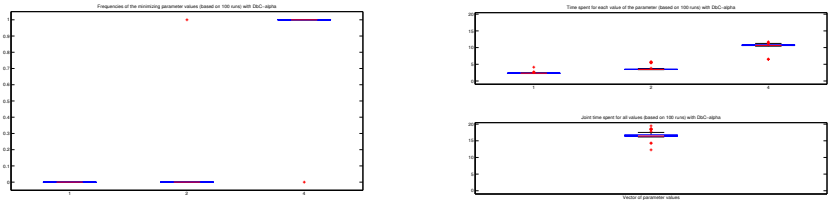


Figure 12: **E1ts**. For method *DbC*: minimizing-power measure of the values and computational times



on the parameter value (number of blocks, here).

Methodology Effect

As the models considered in this simulation exercise are nowhere stationary, on first thought it seems that the estimated error rates should endlessly decrease with the number of blocks because of a better local estimation of the everywhere-changing integrated periodogram; on second thought, however, we realize that also decreases the number of points in each block and therefore the quantity of information, the quality of the periodogram and the discriminant power of the variables (see section A.1.4). To notice this effect in practice, we can repeat the exercise with higher values in `numberBlocks` or with a smaller value for T. When `numberBlocks = [1 2 4 8 16]`, the robust algorithm *DbC*-α always outperforms the algorithm *DbC*, but splitting in eight or sixteen blocks does not provides better results, in general, as there are not enough points in each block for the variables to have higher discriminant power:

```
MINIMIZING-POWER OF THE METHODS
    -> Mean:
       0   1
```

and

```
METHOD: DbC-alpha
       MINIMIZING-POWER OF THE PARAMETER VALUES
       -> Mean:
          0   0.01   0.87   0.11   0.01
```

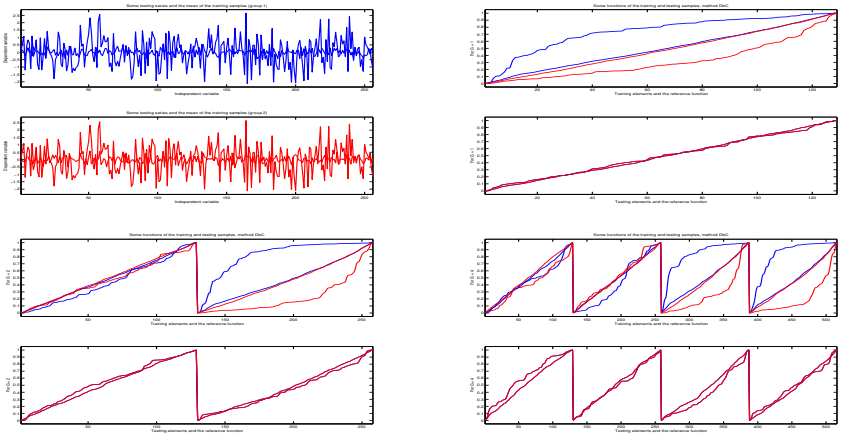
When the length, instead of the number of blocks, is changed, the results for `numberBlocks = [1 2 4]` and `T = 128` are

```
MINIMIZING-POWER OF THE METHODS
    -> Mean:
       0.08   0.92
```

and

```
METHOD: DbC-alpha
       MINIMIZING-POWER OF THE PARAMETER VALUES
       -> Mean:
          0   0.46   0.54
```

Figure 13: **E2ts**. Time series and functional data for exercise E2ts



Because of the loss of points—and therefore quality—in each block, we can see that some times *DbC* outperforms *DbC*-α even with the contamination, and almost half the times the latter algorithm chooses two blocks rather than four.

6.1.2 Simulation Exercise E2ts: Types of Call

In this section, the output of `SCRIPTnewTS` is shown for the models given by expressions 13 (both are two-piece piecewise stationary processes). With the code:

```
numberExercise = 2;
T = 256;
% Training samples
    paramVector1 = [-0.1 +0.1]; paramVector2 = [+0.1 +0.1]
    N1= 23; N2 = 25;
    allGroupTS1 = dataExercisesTS(numberExercise,1,N1,T,paramVector1);
    allGroupTS1(1:3,:)  = dataExercisesTS(numberExercise,1,3,T,[+0.7 -0.2]);
    allGroupTS2 = dataExercisesTS(numberExercise,2,N2,T,paramVector2);
    allGroupTS2(1:3,:)  = dataExercisesTS(numberExercise,2,3,T,[-0.7 +0.2]);
% New data (M elements of each group)
    M = 4;
    newDataTS = [dataExercisesTS(numberExercise,1,M,T,paramVector1);...
          dataExercisesTS(numberExercise,2,M,T,paramVector2)];
```

we generate two training and one testing samples of pseudo-real time series. Both training samples have three contaminating series. The time series and their functional data are shown in figure 13. In this situation, we know in advance that the true labels of the eight testing elements are:

```
1   1   1   1   2   2   2   2
```

In our implementation, we have run the previous code only once so that to compare, for just the same data, the labels obtained from different calls (and, as a particular case, for different splits of the samples). The calls are summarized in table 1.

Table 1: Types of call for simulation exercise E2ts

Call	B	`paramOptimization`	`methodSelection`	`depthBasedRobustness`
1	1	false	false	false
2	1	true	false	false
3	1	true	true	false
4	1	false	false	true
5	1	true	true	true
6	10	true	false	false
7	10	true	true	true

For these calls we can apply some reasoning, with the concepts of appendix A in mind, to deduce the effects involved in the classification process determined by the call. Firstly, the atypical contaminating elements introduce a (negative) training-sample weakening effect. Secondly, when the inner loop is executed, training- and testing-sample-size effects as well as a run-averaging effect appear; besides, since more than one testing element are being combined to estimate the misclassification error rates in the inner loop, an implicit sample-averaging effect appears too. Thirdly, in the outer loop each new datum is independently treated and, therefore, there is no sample-averaging effect, but there will be a run-averaging effect if several runs are executed in this loop.

The same parameter values and methods will be applied to just the same data in all calls (for some we have disabled temporarily the code that removes the robust algorithm when the depth-based robustifying technique is enabled). That is, `numberBlocks = [1 2 4]`, `numbersTStoCLASSmethods = [1 1 1 1; 1 2 1 1]`, `namesTStoCLASSmethods = {'DbC','DbC-alpha'}`.

Type of Call 1

One run, without paramater optimization nor method selection, and without the robustifying technique; that is, with `B = 1`, `paramOptimization = false`, `methodSelection = false`, `depthBasedRobustness = false`. Then, the ouput is:

```
METHOD: DbC
   CLASSIFICATION LABELS (without parameter optimization nor method selection)
   -> Labels for the new data:
      Parameter value p-th = 1
         Run b = 1:   1  2  2  2  2  1  1  1
      Parameter value p-th = 2
         Run b = 1:   2  2  2  1  2  2  1  2
      Parameter value p-th = 4
         Run b = 1:   2  2  2  2  1  2  1  1
   COMPUTATIONAL TIMES
   -> Mean time spent for each value of the parameter values:
```

```
        0.073  0.073  0.147
    -> Mean joint time spent for all values of the parameter values:
        0.293
METHOD: DbC-alpha
    CLASSIFICATION LABELS (without parameter optimization nor method selection)
    -> Labels for the new data:
      Parameter value p-th = 1
        Run b = 1:   2  1  1  2  2  2  2  2
      Parameter value p-th = 2
        Run b = 1:   1  1  1  2  2  2  2  2
      Parameter value p-th = 4
        Run b = 1:   2  1  1  2  2  2  2  1
    COMPUTATIONAL TIMES
    -> Mean time spent for each value of the parameter values:
      0.138  0.176  0.329
    -> Mean joint time spent for all values of the parameter values:
      0.643
```

With this call, *DbC* has misclassified from 4 to 7 elements (results as bad as those of classifying at random) while the robust *DbC*-α has misclassified 1 to 3 elements. We can see that *DbC* is affected by the contaminating elements while *DbC*-α is not. Concretely, the former method suffers from a training-sample weakening (switching, in fact) effect, as it tends to classify the first four elements in group 2 and the last four ones in group 1. As expected (see the formulas of the models), both methods obtain the best results when series are split in two blocks.

In comparing the labels of this call with those of the following ones, we expect: (i) the robust method *DbC*-α to be automatically selected in call 3; (ii) the results of method *DbC* to improve when the depth-based robustifying technique is applied in call 4.

Type of Call 2

One run, with parameter optimization only, and without the robustifying technique, that is, with: `B = 1`, `paramOptimization = true`, `paramDataReuse = false`, `m1param = 4`, `m2param = m1param`, `leaveOneOutParam = false`, `Bparam = 10`, `methodSelection = false`, `depthBasedRobustness = false`. Then, the ouput is:

```
METHOD: DbC
    CLASSIFICATION LABELS (the parameter value has been optimized)
    -> Labels for the new data:
      Run b = 1:   2  1  2  2  2  2  1  1
    MINIMIZING-POWER OF THE PARAMETER VALUES
    -> Mean:
      0  0  1
    COMPUTATIONAL TIMES
    -> Mean time spent for each value of the parameter values:
```

```
      0.094  0.114  0.193
   -> Mean joint time spent for all values of the parameter values:
      0.402
METHOD: DbC-alpha
   CLASSIFICATION LABELS (the parameter value has been optimized)
   -> Labels for the new data:
      Run b = 1:   2  1  1  2  2  2  2  1
   MINIMIZING-POWER OF THE PARAMETER VALUES
   -> Mean:
      0  0  1
   COMPUTATIONAL TIMES
   -> Mean time spent for each value of the parameter values:
      0.134  0.163  0.305
   -> Mean joint time spent for all values of the parameter values:
      0.603
```

For this call *DbC* has still misclassified 5 elements, more than the expected 4 of classfying at random—obviously, optimizing the parameter does not solve the training-sample weakening effect caused by the contamination. Now, method *DbC*-α has misclassified 3 elements while in call 1 only one element were misclassified for the best parameter value; on first thought this seems paradoxical, but on second thought we realize that in chosing the best result in call 1 we are applying a "visual optimization," while the automatic parameter optimization is based on the run-averaging of classifiers that use fewer data and, therefore, may provide worse results when in the inner loop the positive run-averaging effect does not compensate the negative training- and testing-sample-size effects due to the splits.

Type of Call 3

One run, with both parameter and method optimizations, and without the robustifying technique, that is, with: `B = 1`, `paramOptimization = true`, `paramDataReuse = false`, `m1param = 4`, `m2param = m1param`, `leaveOneOutParam = false`, `Bparam = 10`, `methodSelection = true`, `depthBasedRobustness = false`. Then, the ouput is:

```
CLASSIFICATION LABELS (both the parameter value and the method have been optimized)
   -> Labels for the new data:
   Run b = 1:   1  1  1  2  2  2  2  1
METHOD: DbC
   MINIMIZING-POWER OF THE PARAMETER VALUES
   -> Mean:
      0  0  1
   COMPUTATIONAL TIMES
   -> Mean time spent for each value of the parameter values:
      0.097  0.105  0.174
   -> Mean joint time spent for all values of the parameter values:
      0.376
```

```
METHOD: DbC-alpha
    MINIMIZING-POWER OF THE PARAMETER VALUES
    -> Mean:
       0  0  1
    COMPUTATIONAL TIMES
    -> Mean time spent for each value of the parameter values:
       0.113  0.147  0.307
    -> Mean joint time spent for all values of the parameter values:
       0.569
MINIMIZING-POWER OF THE METHODS
-> Mean:
    0  1
```

By looking at the minimizing-power measure of the methods, we know that DbC-α has been selected in the unique run. Then, 2 elements have been misclassified after splitting series into four blocks—take a look at the minimizing-power measure of the parameter values.

Type of Call 4

Although the code disables automatically the robust method DbC-α when `depthBasedRobustness = true`, now we have maintained it both to compare the two methods and to see how they behave when the depth--based robustifying technique is applied. Then, this call is characterized by: one run, without parameter optimization nor method selection, and with the robustifying technique, that is: `B = 1`, `paramOptimization = false`, `methodSelection = false`, `depthBasedRobustness = true`, `alpha = 0.2`. Then, the ouput is:

```
METHOD: DbC
    CLASSIFICATION LABELS (without parameter optimization nor method selection)
    -> Labels for the new data:
       Parameter value p-th = 1
          Run b = 1:   2  1  1  2  2  2  2  2
       Parameter value p-th = 2
          Run b = 1:   1  1  1  2  2  2  2  2
       Parameter value p-th = 4
          Run b = 1:   2  1  1  2  2  2  2  1
    COMPUTATIONAL TIMES
    -> Mean time spent for each value of the parameter values:
       0.161  0.249  0.353
    -> Mean joint time spent for all values of the parameter values:
       0.763
METHOD: DbC-alpha
    CLASSIFICATION LABELS (without parameter optimization nor method selection)
    -> Labels for the new data:
       Parameter value p-th = 1
          Run b = 1:   2  1  1  2  2  2  2  2
```

```
      Parameter value p-th = 2
         Run b = 1:   1  1  1  2  2  2  2  2
      Parameter value p-th = 4
         Run b = 1:   2  1  1  2  2  2  2  1
   COMPUTATIONAL TIMES
   -> Mean time spent for each value of the parameter values:
      0.183  0.347  0.45
   -> Mean joint time spent for all values of the parameter values:
      0.98
```

With the depth-based robustness, *DbC* has misclassified from 1 to 3 elements, and therefore now it is not only faster than *DbC*-α but also as good classifier as it. Method *DbC*-α has neither improved nor worsened its behaviour. Both methods obtain the best results when series are split in two blocks.

Type of Call 5

As we did for call 4, we have maintained *DbC*-α in the execution. Then, this call is characterized by: one run, with both parameter and method optimizations, and with the robustifying technique, that is: `B = 1`, `paramOptimization = true`, `paramDataReuse = false`, `m1param = 4`, `m2param = m1param`, `leaveOneOutParam = false`, `Bparam = 10`, `methodSelection = true`, `depthBasedRobustness = true`, `alpha = 0.2`. Then, the ouput is:

```
CLASSIFICATION LABELS (both the parameter value and the method have been optimized)
   -> Labels for the new data:
   Run b = 1:   2  1  1  1  2  2  2  2
METHOD: DbC
   MINIMIZING-POWER OF THE PARAMETER VALUES
   -> Mean:
      0  0  1
   COMPUTATIONAL TIMES
   -> Mean time spent for each value of the parameter values:
      0.177  0.21  0.385
   -> Mean joint time spent for all values of the parameter values:
      0.774
METHOD: DbC-alpha
   MINIMIZING-POWER OF THE PARAMETER VALUES
   -> Mean:
      0  0  1
   COMPUTATIONAL TIMES
   -> Mean time spent for each value of the parameter values:
      0.188  0.239  0.438
   -> Mean joint time spent for all values of the parameter values:
      0.865
MINIMIZING-POWER OF THE METHODS
```

```
-> Mean:      1  0
```

By looking at the minimizing-power measure of the methods, we can see that *DbC* has been selected in the unique run. We would realize that the two methods have misclassified the same number of elements—although perhaps different elements—by looking at the minimizing-power measure. In case of tie, the labels shown are those of the method called firstly. Since *DbC* has outperformed *DbC*-α (the minimizing-power measure has not registered a tie through the values 0.5 0.5), it should also be selected when it is allocated in the second position of the call. Indeed, we have repeated the call with the inverse order, that is, with

```
numbersTStoCLASSmethods = [1 2 1 1; 1 1 1 1];
namesTStoCLASSmethods = 'DbC-alpha', 'DbC';
```

and *DbC* has been selected again:

```
MINIMIZING-POWER OF THE METHODS
-> Mean:      0  1
```

Finally, the results of this call prove that in some situations *DbC* with the depth-based robustifying technique can outperform the robust method *DbC*-α.

Type of Call 6

Ten runs, without parameter optimization nor method selection, and without the robustifying technique, that is, with: `B = 10`, `paramOptimization = true`, `paramDataReuse = false`, `m1param = 4`, `m2param = m1param`, `leaveOneOutParam = false`, `Bparam = 10`, `methodSelection = false`, `depthBasedRobustness = false`. Then, the ouput is:

```
METHOD: DbC
    CLASSIFICATION LABELS (the parameter value has been optimized)
    -> Labels for the new data:
       Run b = 1:   1  1  1  2  2  2  2  1
       Run b = 2:   2  2  2  1  2  2  1  2
       Run b = 3:   2  2  2  1  2  2  2  2
       Run b = 4:   1  2  2  1  1  1  1  1
       Run b = 5:   2  1  2  2  2  2  2  1
       Run b = 6:   2  2  2  1  2  2  1  2
       Run b = 7:   2  2  2  1  1  1  1  1
       Run b = 8:   2  2  2  1  2  2  2  2
       Run b = 9:   1  2  2  2  2  2  2  1
       Run b = 10:   2  2  2  1  2  2  2  2
       Majority vote (0 for ties),  2  2  2  1  2  2  2  0
    MINIMIZING-POWER OF THE PARAMETER VALUES
    -> Mean:
       0.2  0.5  0.3
    COMPUTATIONAL TIMES
    -> Mean time spent for each value of the parameter values:
```

```
      0.0639  0.0916  0.167
   -> Mean joint time spent for all values of the parameter values:
      0.3231
METHOD: DbC-alpha
   CLASSIFICATION LABELS (the parameter value has been optimized)
   -> Labels for the new data:
      Run b = 1:   2  1  1  2  2  2  2  1
      Run b = 2:   1  1  1  2  2  2  2  1
      Run b = 3:   1  1  1  1  2  2  2  2
      Run b = 4:   2  1  2  2  2  2  2  2
      Run b = 5:   1  1  1  2  2  2  2  1
      Run b = 6:   2  1  1  2  2  2  2  2
      Run b = 7:   1  1  2  2  2  2  2  2
      Run b = 8:   1  1  1  2  2  2  2  1
      Run b = 9:   2  1  1  2  2  2  2  2
      Run b = 10:   1  1  1  2  2  2  2  2
      Majority vote (0 for ties),  1  1  1  2  2  2  2  2
   MINIMIZING-POWER OF THE PARAMETER VALUES
   -> Mean:
      0.2  0.3  0.5
   COMPUTATIONAL TIMES
   -> Mean time spent for each value of the parameter values:
      0.1144  0.1523  0.268
   -> Mean joint time spent for all values of the parameter values:
      0.5356
```

The training-sample weakening effect of the contaminats still affects *DbC*, and the run-averaging effect of the majority-vote classifier cannot improve the results (averaging acts through the variance, not the bias). Method *DbC*-α has misclassified 1 element, as it did in call 1 for the best parameter value. From all these labels it seems that, if the call is repeated several times, the variability of the majority-vote classifier will be smaller than that of the classification method itself, regardless the bias of the classification (see the simulation exercise in section 6.1.4).

Type of Call 7

As we did for call 4, we have maintained *DbC*-α in the execution. Now the call is characterized by: ten runs, with both parameter and method optimizations, and with the robustifying technique, that is: `B = 10`, `paramOptimization = true`, `paramDataReuse = false`, `m1param = 4`, `m2param = m1param`, `leaveOneOutParam = false`, `Bparam = 10`, `methodSelection = true`, `depthBasedRobustness = true`, `alpha = 0.2`. Then, the ouput is:

```
CLASSIFICATION LABELS (both the parameter value and the method have been optimized)
-> Labels for the new data:
   Run b = 1:   2  1  1  2  2  2  2  2
```

```
    Run b = 2:   2  1  1  2  2  2  2  2
    Run b = 3:   1  1  1  2  2  2  2  2
    Run b = 4:   1  1  1  2  2  2  2  1
    Run b = 5:   2  1  1  2  2  2  2  1
    Run b = 6:   1  1  1  2  2  1  1  1
    Run b = 7:   1  1  1  2  2  2  2  1
    Run b = 8:   2  1  1  2  2  2  2  2
    Run b = 9:   1  1  1  2  2  2  2  1
    Run b = 10:   1  1  1  2  2  2  2  1
    Majority vote (0 for ties),  1  1  1  2  2  2  2  1
METHOD: DbC
    MINIMIZING-POWER OF THE PARAMETER VALUES
    -> Mean:
       0.1  0.5  0.4
    COMPUTATIONAL TIMES
    -> Mean time spent for each value of the parameter values:
       0.1527  0.1973  0.3958
    -> Mean joint time spent for all values of the parameter values:
       0.7471
METHOD: DbC-alpha
    MINIMIZING-POWER OF THE PARAMETER VALUES
    -> Mean:
       0.3  0.5  0.2
    COMPUTATIONAL TIMES
    -> Mean time spent for each value of the parameter values:
       0.183  0.2337  0.4478
    -> Mean joint time spent for all values of the parameter values:
       0.8657
MINIMIZING-POWER OF THE METHODS
-> Mean:
    0.6  0.4
```

Now the depth-based robustifying technique has removed the training-sample weakening effect of the contaminating series, and, as a consequence, *DbC* has outperformed *DbC*-α sixty percent of times.

6.1.3 Simulation Exercise E3ts: Data-Quality Effects

In this section, the output of `SCRIPTnewTS` is shown for the models given by expressions 13 (both models are two-piece piecewise stationary). With this exercise, we want to show the effects—positive or negative—that some "contaminating" elements can introduce in the evaluation of classification methods. In section A.1.1 these effects are described and named *training-* and *testing-sample strengthening* and *weakening effects*. The presence of the training-sample weakening effect were also noticed in the exercise in section 6.1.2.

The call has these characteristics: methods *DbC* and *DbC*-α, one run, without parameter optimiza-

tion nor method selection, and without the robustifying technique: `numbersTStoCLASSmethods = [1 1 1 1; 1 2 1 1]`, `namesTStoCLASSmethods = {'DbC','DbC-alpha'}`, `numberBlocks = [1 2 4 8]`, `B = 1`, `paramOptimization = false`, `methodSelection = false`, `depthBasedRobustness = false`.

Without Atypical Training Data

By executing the code

```
numberExercise = 2;
paramVector1 = [-0.1 0]; paramVector2 = [+0.1 0];
T = 64;
% Training samples
    N1= 12; N2 = 13;
    allGroupTS1 = dataExercisesTS(numberExercise,1,N1,T,paramVector1);
    allGroupTS2 = dataExercisesTS(numberExercise,2,N2,T,paramVector2);
% New data (M elements of each group)
    M = 3;
    newDataTS = [dataExercisesTS(numberExercise,1,M,T,paramVector1);...
          dataExercisesTS(numberExercise,2,M,T,paramVector2)];
```

we generate two training and one testing samples of pseudo-real time series. In this situation, we know in advance that the true labels of the six testing elements are:

```
1  1  1  2  2  2.
```

As in example of section 6.1.2, for a better comparison of the results, we have run the previous code only once so that to use the same time series. Our algorithms provide the labels

```
METHOD: DbC
    CLASSIFICATION LABELS (without parameter optimization nor method selection)
    -> Labels for the new data:
       Parameter value p-th = 1
          For b = 1:   1  1  1  2  1  1
       Parameter value p-th = 2
          For b = 1:   1  1  1  2  1  2
       Parameter value p-th = 4
          For b = 1:   1  1  1  1  2  1
       Parameter value p-th = 8
          For b = 1:   1  1  1  2  2  1
    COMPUTATIONAL TIMES
    -> Mean time spent for each value of the parameter values:
       0.065  0.044  0.074  0.143
    -> Mean joint time spent for all values of the parameter values:
       0.326
METHOD: DbC-alpha
    CLASSIFICATION LABELS (without parameter optimization nor method selection)
```

```
   -> Labels for the new data:
      Parameter value p-th = 1
         For b = 1:   1  1  1  2  1  1
      Parameter value p-th = 2
         For b = 1:   1  1  1  1  1  1
      Parameter value p-th = 4
         For b = 1:   1  1  1  1  1  1
      Parameter value p-th = 8
         For b = 1:   1  1  1  2  2  1
   COMPUTATIONAL TIMES
   -> Mean time spent for each value of the parameter values:
      0.068  0.059  0.104  0.205
   -> Mean joint time spent for all values of the parameter values:
      0.436
```

If there are no atypical series, method *DbC* have slightly better behaviour than *DbC*-α—the former method uses all the available data while the latter does not.

With Strengthening Atypical Training Data

By substituting the "contaminating" series

```
allGroupTS1(1:3,:)  = dataExercisesTS(numberExercise,1,3,T,[-0.6 0]);
allGroupTS2(1:3,:)  = dataExercisesTS(numberExercise,2,3,T,[+0.6 0]);
```

the output labels are

```
METHOD: DbC
   CLASSIFICATION LABELS (without parameter optimization nor method selection)
   -> Labels for the new data:
      Parameter value p-th = 1
         For b = 1:   1  1  1  2  1  1
      Parameter value p-th = 2
         For b = 1:   2  1  1  2  2  2
      Parameter value p-th = 4
         For b = 1:   2  1  1  2  2  1
      Parameter value p-th = 8
         For b = 1:   2  1  1  2  2  1
   COMPUTATIONAL TIMES
   -> Mean time spent for each value of the parameter values:
      0.064  0.043  0.075  0.143
   -> Mean joint time spent for all values of the parameter values:
      0.325
METHOD: DbC-alpha
   CLASSIFICATION LABELS (without parameter optimization nor method selection)
```

```
-> Labels for the new data:
   Parameter value p-th = 1
      For b = 1:   1  1  1  1  1  1
   Parameter value p-th = 2
      For b = 1:   1  1  1  2  2  2
   Parameter value p-th = 4
      For b = 1:   2  1  1  2  2  1
   Parameter value p-th = 8
      For b = 1:   2  1  1  2  2  1
COMPUTATIONAL TIMES
-> Mean time spent for each value of the parameter values:
   0.068  0.059  0.107  0.199
-> Mean joint time spent for all values of the parameter values:
   0.433
```

Because of the training strengthening effect introduced by elements that separate the reference elements of the two groups, now *DbC* and *DbC*-α have misclassified fewer elements.

With Weakening Atypical Training Data

With the "contaminating" series

```
allGroupTS1(1:3,:)  = dataExercisesTS(numberExercise,1,3,T,[+0.6 0]);
allGroupTS2(1:3,:)  = dataExercisesTS(numberExercise,2,3,T,[-0.6 0]);
```

the output labels are

```
METHOD: DbC
   CLASSIFICATION LABELS (without parameter optimization nor method selection)
   -> Labels for the new data:
      Parameter value p-th = 1
         For b = 1:   2  2  2  1  2  2
      Parameter value p-th = 2
         For b = 1:   1  2  2  1  1  1
      Parameter value p-th = 4
         For b = 1:   2  2  2  1  1  1
      Parameter value p-th = 8
         For b = 1:   1  2  2  1  2  2
   COMPUTATIONAL TIMES
   -> Mean time spent for each value of the parameter values:
      0.066  0.044  0.074  0.145
   -> Mean joint time spent for all values of the parameter values:
      0.329
METHOD: DbC-alpha
   CLASSIFICATION LABELS (without parameter optimization nor method selection)
```

```
-> Labels for the new data:
   Parameter value p-th = 1
      For b = 1:   2  2  2  1  2  2
   Parameter value p-th = 2
      For b = 1:   1  2  2  1  1  1
   Parameter value p-th = 4
      For b = 1:   2  2  1  1  1  1
   Parameter value p-th = 8
      For b = 1:   2  2  2  1  2  1
COMPUTATIONAL TIMES
-> Mean time spent for each value of the parameter values:
   0.069  0.059  0.107  0.223
-> Mean joint time spent for all values of the parameter values:
   0.458
```

Now, the training weakening effect appears with data that make the reference elements of the two groups closer. Now *DbC* and *DbC*-α have misclassified almost all the elements—the criterion has even been switched.

With Strengthening Atypical Testing Data

If we generate new contamination-free training samples but we modify the new data so that to make them easier to classify, with

```
% New data (M elements of each group)
    M = 3;
    paramVector1 = [-0.5 0]; paramVector2 = [+0.5 0];
    newDataTS = [dataExercisesTS(numberExercise,1,M,T,paramVector1);...
         dataExercisesTS(numberExercise,2,M,T,paramVector2)];
```

both methods classify the six testing elements correctly for any parameter value. These elements, which may be legitimately generated by the underlying process, would introduce a perturbance in the estimation of the misclassification error rates.

With Weakening Atypical Testing Data

Finally, with contamination-free training samples and uncorrectly-labelled new data (an extreme of "difficult to classify"),

```
% New data (M elements of each group)
    M = 3;
    paramVector1 = [+0.7 0]; paramVector2 = [-0.7 0];
    newDataTS = [dataExercisesTS(numberExercise,1,M,T,paramVector1);...
         dataExercisesTS(numberExercise,2,M,T,paramVector2)];
```

both methods misclassify the six testing elements for any parameter value. Again, these legitimate, improbable elements would introduce a perturbance in the estimation of the misclassification error rates.

6.1.4 Simulation Exercise E4ts: Run-Averaging Effect

To show the run-averaging effect described in section 2.6.3, on a copy of `SCRIPTnewTS` we have run its code $H = 500$ times. After disabling the figures (too many would be generated), we have written few lines of code to register the labels of both the classification method DbC and the majority-vote classifier. For a fair comparison, in each repetition only DbC's labels of the first run ($b = 1$) are considered for the estimation. It is worth noticing that, in each repetition, the previous single instance of the classifier does not depend on B, while the majority-vote classifier does. Finally, the mean square error of the two classifiers have been calculated as the following formulas tell (there will be M testing data e_i)

$$\begin{aligned}
\hat{MSE}(c_1(e)) &= \frac{1}{M}\sum_{i=1}^{M}\hat{MSE}(c_1(e_i)) \\
&= \frac{1}{M}\sum_{i=1}^{M}\left(\left[\frac{1}{H}\sum_{h=1}^{H}c_{1,h}(e_i) - g_{e_i}\right]^2 + \frac{1}{H-1}\sum_{h=1}^{H}\left[c_{1,h}(e_i) - \frac{1}{H}\sum_{k=1}^{H}c_k^{(1)}(e_i)\right]^2\right) \\
&= \frac{1}{M}\sum_{i=1}^{M}\left[\frac{1}{H}\sum_{h=1}^{H}c_{1,h}(e_i) - g_{e_i}\right]^2 + \frac{1}{M}\frac{1}{H-1}\sum_{i=1}^{M}\sum_{h=1}^{H}\left[c_{1,h}(e_i) - \frac{1}{H}\sum_{k=1}^{H}c_k^{(1)}(e_i)\right]^2 \qquad (36)
\end{aligned}$$

the sample mean has mean square error

$$\begin{aligned}
\hat{MSE}(C(e)) &= \frac{1}{M}\sum_{j=1}^{M}\hat{MSE}(C(e_i)) \\
&= \frac{1}{M}\sum_{j=1}^{M}\left(\left[\frac{1}{H}\sum_{h=1}^{H}C_h(e_i) - g_{e_i}\right]^2 + \frac{1}{H-1}\sum_{h=1}^{H}\left[C_h(e_i) - \frac{1}{H}\sum_{k=1}^{H}C_k(e_i)\right]^2\right) \\
&= \frac{1}{M}\sum_{j=1}^{M}\left[\frac{1}{H}\sum_{h=1}^{H}C_h(e_i) - g_{e_i}\right]^2 + \frac{1}{M}\frac{1}{H-1}\sum_{j=1}^{M}\sum_{h=1}^{H}\left[C_h(e_i) - \frac{1}{H}\sum_{k=1}^{H}C_k(e_i)\right]^2 \qquad (37)
\end{aligned}$$

The training and new data have been generated with the code

```
numberExercise = 3;
    paramVector1 = [0.8 0.5 0.81]; paramVector2 = [0.8 0.3 0.81];
T = 64;
% Training samples
    N1= 28; N2 = 28;
    allGroupTS1 = dataExercisesTS(numberExercise,1,N1,T,paramVector1);
    allGroupTS2 = dataExercisesTS(numberExercise,2,N2,T,paramVector2);
% New data (M elements of each group)
    M = 3;
    newDataTS = [dataExercisesTS(numberExercise,1,M,T,paramVector1);...
                 dataExercisesTS(numberExercise,2,M,T,paramVector2)];
trueLabels = [1 1 1 2 2 2];
```

The call is determined by `numbersTStoCLASSmethods = [1 1 1 1]`, `namesTStoCLASSmethods = 'DbC'`, `numberBlocks = [1 2]`, `paramOptimization = true`, `paramDataReuse = false`, `n1 = -1`, `n2 = -1`, `m1param = 10`, `m2param = m1param`, `leaveOneOutParam = false`, `Bparam = 10`, `methodSelection = false`, and `depthBasedRobustness = false`. Three values have been considered for B.

With $B = 10$

For the first run of all repetitions, the estimated mean square error of $c_1 = DbC$ is

```
0.1954 + 0.0681 = 0.2635
```

while for the 100 repetitions, the estimated mean square error of the majority-vote classifier C (based on the B runs of DbC) is

```
0.1681 + 0.1031 = 0.2712
```

For these models and call, when $B = 10$ both have similar mean square error, although C has slightly smaller bias and salightly higher variance.

With $B = 25$

For this number of runs, the estimated mean square error of $c_1 = DbC$ is

```
0.2931 + 0.1331 = 0.4262
```

and the estimated mean square error of the majority-vote classifier C is

```
0.3337 + 0.0093 = 0.3430
```

Now, when $B = 25$ the classifier C has slightly higher bias but smaller mean square error and quite smaller variance.

With $B = 50$

In 50 runs the estimated mean square error of $c_1 = DbC$ is

```
0.2945 + 0.1454 = 0.4399
```

while for the majority-vote classifier C is

```
0.3842 + 0.0960 = 0.4802
```

When $B = 50$, both classifiers provide slightly worse results. The majority-vote classifier C has still smaller variance than c_1 but higher bias and mean square error.

With $B = 100$

Finally, in 100 runs the estimated mean square error of $c_1 = DbC$ is

```
0.2901 + 0.1659 = 0.4560
```

while for the majority-vote classifier C is

```
0.4537 + 0.0331 = 0.4868
```

For this value of B, both classifiers provide similar mean square error, though C has quite higher bias than c_1 but also quite smaller variance (due to the run-averaging effect).

6.1.5 Simulation Exercise E5ts: Discreteness Effect

On a copy of `SCRIPTrealTS`, we have modified the code so that to register also the minimizing-power of each classification method in the inner loop (by default it is measured only in the outer loop). In case of ties, all the minimizing methods are registered. Then, we will call two methods when the discreteness effect of section A.1.2 is notoriously present; in this situation, the two methods should be selected for the outer loop even if it is clear than one would be better than the other in a "reasonable framework." Finally, we will show that: (1) the effect is smaller for either higher testing sample sizes or higher number or runs, and (2) that the two-step approach is capable of creating ties to give a second chance to some methods, that is, that the approach would work in practice. By using the code

```
numberExercise = 2;
    paramVector1 = [-0.1 0]; paramVector2 = [+0.1 +0.1];
N1= 75; N2 = 75; T = 512;
allGroupTS1 = dataExercisesTS(numberExercise,1,N1,T,paramVector1);
allGroupTS2 = dataExercisesTS(numberExercise,2,N2,T,paramVector2);
```

we generate two samples of pseudo-real data (pure ARMA processes, here). Now we will consider the call

```
m1 = 50; m2 = m1;
B = 100;
numbersTStoCLASSmethods = [1 1 1 1; 1 2 1 1];
namesTStoCLASSmethods = 'DbC', 'DbC-alpha';
    numberBlocks = [1 2]
paramOptimization = true;
paramDataReuse = false;
    n1 = -1;
    n2 = -1;
methodSelection = true;
```

with which the final testing samples will have $m1 = 15 = m2$ series, the final training samples will have $n1 = 20/2 = n2$ series, and the 10 series of each population will be used to select the method.

The "natural importance" of the methods in minimizing the error rates can be evaluated when both methods are always considered for the other roop. By using the code

```
m1param = 2; m2param = 2;
Bparam = 2;
discretenessMargin = 0;
```

we cause a situation where ties are probable due to the discreteness effect and the two-step method-selection approach is disabled. Two possible contiguous values of the estimation are separated by a distance $1/(2(2+2)) = 0.125$. The minimizing-power measurements of the methods at the end of the inner and the outer loops are, respectively:

```
0.5000    0.5000
```

and

```
0.7000    0.3000
```

Thus, we can see that for these models and call, the first method outperforms the second

Statistical Weakening of the Effect

With the call

```
m1param = 8; m2param = 8;
Bparam = 2;
discretenessMargin = 0;
```

8 elements are used for training, instead of 2. Two possible contiguous values of the estimation are separated by a distance $1/(2(8+8)) = 0.03125$. Now the minimizing-power measurement is

```
0.6100    0.3900
```

We can see that the discreteness effect causes a smaller proportion of ties in the inner loop.

On the other hand, with

```
m1param = 2; m2param = 2;
Bparam = 10;
discretenessMargin = 0;
```

the number of runs is increased, instead of the testing-sample sizes. Now, two possible contiguous values of the estimation are separated by a distance $1/(8(2+2)) = 0.03125$. As a consequence,

```
0.5900    0.4100
```

A similar variation in the minimizing-power, measured in the inner loop, has been obtained—notice that the step between possible estimated values is, for these sample sizes and number or runs, equal to that of the previous call.

Methodological Creation of Pseudoties

Although it would be better to evaluate the two-step approach with a real problem, we used this simulation exercise to show that, once ties are little probable, the approach creates new pseudoties. Firstly, we increase both the testing sample sizes and the number of runs

```
m1param = 8; m2param = 8;
Bparam = 8;
discretenessMargin = 0;
```

and the minimizing-power measurement is

```
0.6900    0.3100
```

For this call, two possible contiguous values of the estimation are separated by a distance $1/(8(8+8)) = 0,0078125$, and we can see that the "natural" minimizing-power measurements of the outer loop are also obtained in the inner loop.

Finally, by enabling the two-step approach with

```
discretenessMargin = 2;
```

the minimizing-power measurement is

```
0.5000   0.5000
```

That means that the margin of error is capable of creating the necessary pseudoties so that to weaken the discreteness effect in the inner loop.

6.1.6 Simulation Exercise E6ts: Inherent Effect

In this section, the output of `SCRIPTapplicationTS` is shown for the models given by expressions 13. The aim of this simulation exercise is to show an example of the inherent effect defined in section A.2.4. An additional while is necessary to do some calculations the first time, with respect to the following ones. As a tricky solution, some calculations are done one more time than necessary:

```
computationalTimesTStoFD = zeros(size(numbersTStoCLASSmethods,1),length(paramValues));
for p = 1:length(paramValues)
  infoTStoFDmethod = paramValues(p);
  for i1 = 1:size(numbersTStoCLASSmethods,1)
   if isequal(i1,1) ||...
       (i1>1 && ~any(numbersTStoCLASSmethods(i1,1)==numbersTStoCLASSmethods(1:(i1-1),1)))
    % The following lines are a trick
    if isequal(p,1)
     eval(...  CodeHere ...)
    end
    startClock = clock;
     eval(...  CodeHere ...)
          etime(clock,startClock);
   else % To use previous calculations, since i1-1, i1-2,...  were also considered
     ⋮
   end
 end
end
```

As an example, with this code the computational time of a call were

```
METHOD: DbC
    COMPUTATIONAL TIMES
    -> Mean time spent for each value of the parameter values:
       0.053  0.036  0.062  0.119
    -> Mean joint time spent for all values of the parameter values:
       0.27
METHOD: DbC-alpha
    COMPUTATIONAL TIMES
    -> Mean time spent for each value of the parameter values:
```

```
    0.048  0.043  0.079  0.16
  -> Mean joint time spent for all values of the parameter values:
    0.33
```

while without the three lines of the solution, a call to the code provided

```
METHOD: DbC
  COMPUTATIONAL TIMES
  -> Mean time spent for each value of the parameter values:
    0.381  0.035  0.06  0.115
  -> Mean joint time spent for all values of the parameter values:
    0.591
METHOD: DbC-alpha
  COMPUTATIONAL TIMES
  -> Mean time spent for each value of the parameter values:
    0.375  0.042  0.075  0.141
  -> Mean joint time spent for all values of the parameter values:
    0.33
```

Watch the time of the first parameter value. Since our code allows some methods to use calculations previously done by another method and to register its computational time, the perturbation propagates to them.

I have verified this effect in different computers, operating systems and versions of MATLAB. A simple call like `eval(['litter=0; clear litter'])`, instead of the implemented trick, does not solve the problem. This effect is not related to either the MATLAB's function `eval` or the type or datum.

As the effect is negligible for many runs or big data, the user should remove the trick especially in the latter case.

6.1.7 Suggested Simulation Exercises

Simulation Exercise 1ts

`numberExercise = 1`, `paramVector1 = [0 +0.05]`, `paramVector2 = [0 -0.05]`, `n1 = 10`, `n2 = n1`, `m1 = 20`, `m2 = m1`, `T = 1024`, `dyadicSplits = 0`, `numberBlocks = 2.^(0:dyadicSplits)`, `contamination = false`, `B = 250`.

Simulation Exercise 2ts

`numberExercise = 1`, `paramVector1 = [0.5 +0.1]`, `paramVector2 = [0.5 -0.1]`, `n1 = 25`, `n2 = n1`, `m1 = 75`, `m2 = m1`, `T = 1024`, `dyadicSplits = 0`, `numberBlocks = 2.^(0:dyadicSplits)`, `contamination = true`, `contamType = 'B'`, `n1c = 4`, `alpha = 0.2`, `B = 50`.

Simulation Exercise 3ts

`numberExercise = 2`, `paramVector1 = [-0.1 0]`, `paramVector2 = [+0.1 0.3]`, `n1 = 14`, `n2 = 16`, `m1 = 39`, `m2 = 37`, `T = 2048`, `dyadicSplits = 2`, `numberBlocks = 2.^(0:dyadicSplits)`, `contamination = true`, `contamType = 'C'`, `n1c = 1`, `alpha = 0.2`, `B = 75`.

Simulation Exercise 4ts

`numberExercise = 3`, `paramVector1 = [0.8 0.5 0.81]`, `paramVector2 = [0.8 0.3 0.81]`, `n1 = 30`, `n2 = n1`, `m1 = 70`, `m2 = m1`, `T = 512`, `numberBlocks = [2 4 6 8]`, `contamination = true`, `contamType = 'B'`, `n1c = 5`, `alpha = 0.2`, `B = 50`.

6.2 Functional Data

6.2.1 Simulation Exercise E1fd: Output of the Code

In this section, the output of `SCRIPTsimulatedFD` is shown for: `numberExercise = 7`, `paramVector1 = [1/2 100 300]`, `paramVector2 = [1/300 100 300]`, `n1 = 50`, `n2 = n1`, `m1 = 75`, `m2 = m1`, `T = 300`, `B = 200`, `numbersFDtoCLASSmethods = [1 1 1 1; 1 2 1 1; 1 3 1 1]`, `diffOrders = [0 1 2]`, `diffsMode = 'differentials'`, `namesFDtoCLASSmethods = {'WI', 'WD', 'dKNN'}`, `paramOptimization = false`, `methodSelection = false`. The models are given by expressions 9.

Textual Information

When `verbose = true`, the following text is shown. During the runs

```
START


Generating the data...


Starting the iterations...


    ITERATION: 1
     ⋮
    ITERATION: 3
     Information for method WI
      Variables with null variability in the whole data set:   None
      Discriminant variables available:   x1  x2  x3
      Discriminant variables considered:   x1  x2  x3
     Information for method WD
      ⋮
     Information for method WI
      Error rates for group 1:   0.12  0.26667  0.37333
      Error rates for group 2:   0.13333  0.22667  0.33333
      Overall error rates:   0.12667  0.24667  0.35333
      Time spent for each value of the parameter values:   0  0  0
      Joint time spent for all values of the parameter values:   0
      Multivariate method
       Error rates for group 1:   0.14667
       Error rates for group 2:   0.13333
       Overall error rates:   0.14
```

```
          Coefficients of the parameter values:   9.0255  0.11274  1.9241
          Time spent with the vector of variables:   0
        Information for method WD
         ⋮
       ITERATION: 4
        ⋮

    ...ending the iterations
```

and, as a summary of the results, the code shows

```
METHOD: WI
   MISCLASSIFICATION ERROR RATES (without parameter optimization nor method selection)
   -> Mean of the estimated error rates for group 1:
      0.1364  0.24907  0.35067
    -> Standard deviation:
      0.045249  0.057639  0.063526
    -> Standard error:
      0.0032  0.0041  0.0045
   -> Mean of the estimated error rates for group 2:
      0.13007  0.23427  0.33933
    -> Standard deviation:
      0.045304  0.060524  0.06915
    -> Standard error:
      0.0032  0.0043  0.0049
   -> Mean of the estimated overall error rates:
      0.13323  0.24167  0.345
    -> Standard deviation:
      0.028366  0.035149  0.038544
    -> Standard error:
      0.0020  0.0025  0.0027
   MISCLASSIFICATION ERROR RATES (as if parameter optimization had been applied)
   -> Mean of the estimated error rates for group 1:
      0.1364
    -> Standard deviation:
      0.045249
    -> Standard error:
      0.0032
   -> Mean of the estimated error rates for group 2:
      0.13007
    -> Standard deviation:
      0.045304
    -> Standard error:
```

```
        0.0032
    -> Mean of the estimated overall error rates:
       0.13323
     -> Standard deviation:
        0.028366
     -> Standard error:
        0.0020
    COMPUTATIONAL TIMES
    -> Mean time spent for each value of the parameter values:
       0.00971  0.00635  0.00663
    -> Mean joint time spent for all values of the parameter values:
       0.02285
    MULTIVARIATE METHOD
    -> Mean of the estimated error rate for group 1:
       0.15773
     -> Standard deviation:
        0.056081
     -> Standard error:
        0.0040
    -> Mean of the estimated error rate for group 2:
       0.15273
     -> Standard deviation:
        0.051412
     -> Standard error:
        0.0036
    -> Mean of the estimated overall error rates:
       0.15523
     -> Standard deviation:
        0.033001
     -> Standard error:
        0.0023
    -> Mean coefficients of the parameter values:
       7.0638  0.44181  3.046
    -> Mean joint time spent for all values of the parameter values:
       0.03376
METHOD: WD
  ⋮
MISCLASSIFICATION ERROR RATES (as if both parameter optimization and method selection...)
-> Estimation of the mean of the estimated error rates for group 1:
    0.1364
  -> Standard deviation:
     0.045249
```

```
   -> Standard error:
      0.0032
-> Estimation of the mean of the estimated error rates for group 2:
    0.13007
   -> Standard deviation:
      0.045304
   -> Standard error:
      0.0032
-> Estimation of the mean of the estimated overall error rates:
    0.13323
   -> Standard deviation:
      0.028366
   -> Standard error:
      0.0020

Result figures...

STOP   (And remember:  'Everything and nothing is possimpible', Barney Stinson)
```

To avoid any possible beginner effect—see section A.2.4—we provide the output of the third run rather than the first. Since $B = 200$ the order of magnitude of the standard error is one unit smaller than the estimations of the mean error rates.

Graphical Information

When `dataFigures = true` and `paramOptimization = false`, for each global method the plots in figure 14 are generated from during the first run (we include here those of method *WI* only). The figures showing the results, 17 and 18, are based on the B runs. During the first run, result figures similar to these are also generated (we do not include them here). Figures can be used for prospective purpose before considering a big value for B. In this exercise, the histograms and the coefficients agree to identify the discriminant variable with the highest discriminant power. As expected, the computational time does not depend on the value of the parameter (order of the differentiation, here).

6.2.2 Suggested Simulation Exercises

Simulation Exercise 1fd

`numberExercise = 1`, `paramVector1 = [1 1]`, `paramVector2 = [1 1]`, `n1 = 100`, `n2 = n1`, `m1 = 100`, `m2 = m1`, `T = 30`, `B = 1000`, `diffOrders = [0:2]`, `diffsMode = 'derivatives'`.

Simulation Exercise 2fd

`numberExercise = 2`, `paramVector1 = [1 1]`, `paramVector2 = [1 1]`, `n1 = 100`, `n2 = n1`, `m1 = 100`, `m2 = m1`, `T = 30`, `B = 1000`, `diffOrders = [0:2]`, `diffsMode = 'derivatives'`.

Figure 14: **E1fd**. Some crude functions and their differential up to the second order (on the left), and the discriminant variable for all the functions (on the right)

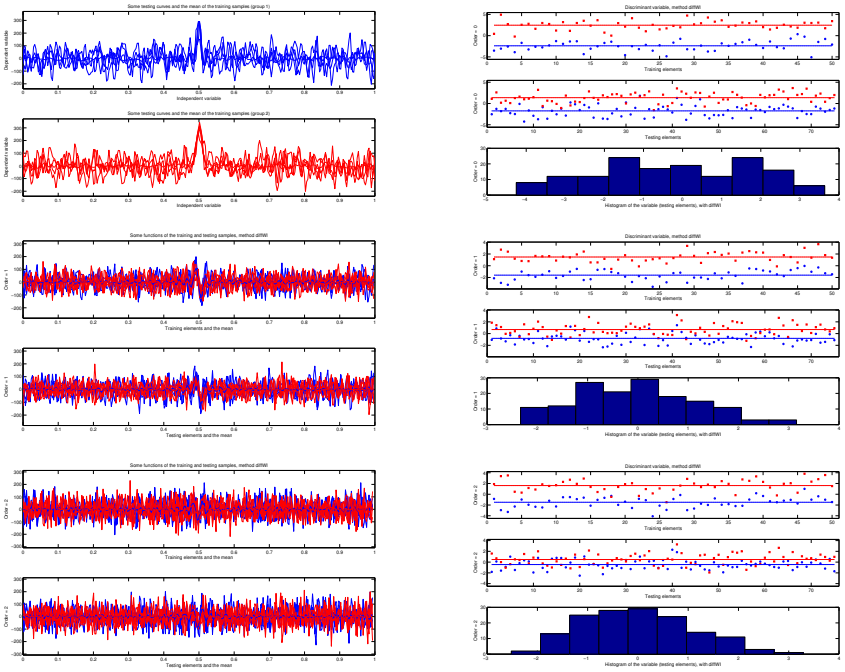


Figure 15: **E1fd**. Scatter plot of the three variables and three-dimensional representation

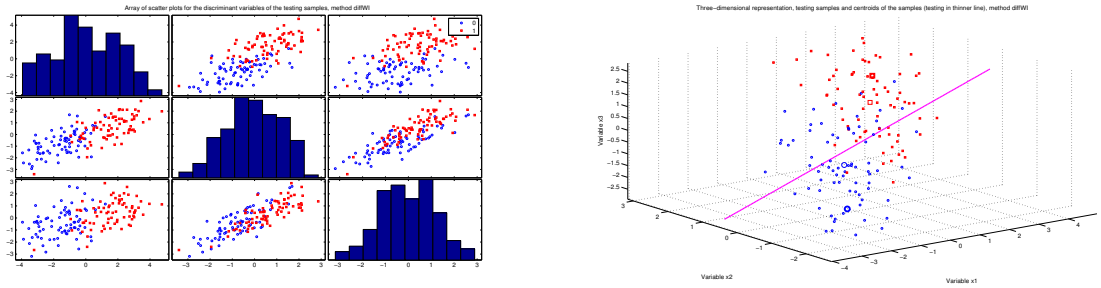


Figure 16: **E1fd**. Discriminant function (for all the functions)

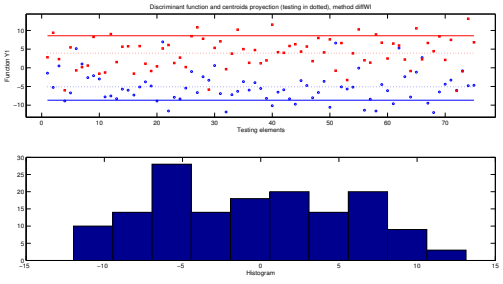


Figure 17: **E1fd**. Estimated misclassification error rates and evolution of the mean

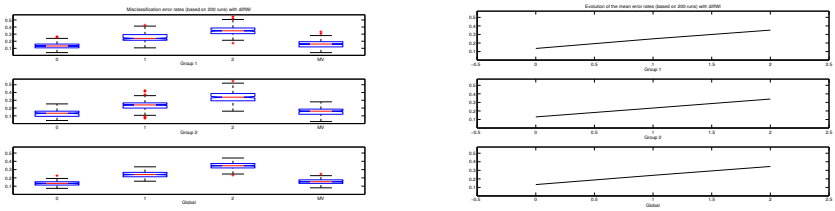
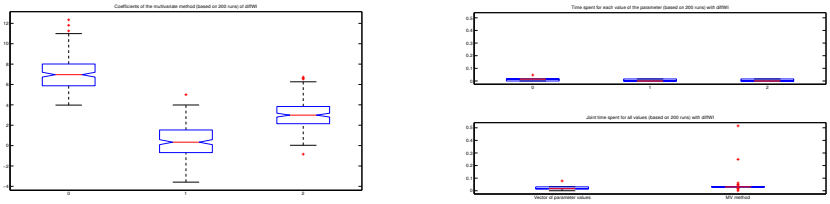


Figure 18: **E1fd**. Coefficients of the multivariate method and computational times



Simulation Exercise 3fd

`numberExercise = 3, paramVector1 = [1 1 5/4], paramVector2 = [1 1], n1 = 100, n2 = n1, m1 = 100, m2 = m1, T = 30, B = 1000, diffOrders = [0:2], diffsMode = 'derivatives'`.

Simulation Exercise 4fd

`numberExercise = 4, paramVector1 = [4 1], paramVector2 = [4.5 1], n1 = 100, n2 = n1, m1 = 100, m2 = m1, T = 30, B = 1000, diffOrders = [0:3], diffsMode = 'derivatives'`.

Simulation Exercise 5fd

`numberExercise = 5, paramVector1 = [1/20], paramVector2 = [1/20], n1 = 100, n2 = n1, m1 = 100, m2 = m1, T = 200, B = 1000, diffOrders = [0:3], diffsMode = 'derivatives'`.

Simulation Exercise 6fd

`numberExercise = 6, paramVector1 = [4 1 1/100], paramVector2 = [4 1], n1 = 100, n2 = n1, m1 = 100, m2 = m1, T = 300, B = 500, diffOrders = 0:11, diffsMode = 'differentials'`.

Simulation Exercise 7fd

`numberExercise = 7, paramVector1 = [1/2 100 300], paramVector2 = [1/300 100 300], n1 = 100, n2 = n1, m1 = 100, m2 = m1, T = 300, B = 500, diffOrders = [0 1 3 5], diffsMode = 'differentials'`.

A Effects

In this appendix we give theoretical explanations on some easy-to-identify perturbations that may affect the classification process and the computational effort. First, we devote a subsection to those due to the quality of the data, the quantity of data, the classification method itself, and the scheme where the whole process is implemented. In a second subsection, we include some time effects due to the quantity of data, the possible auxiliary techniques the method needs, the order in which the methods are called in the scheme, and the software and hardware of the computer on which the code is executed. I have named all the effects introduced here except the *overfitting effect*—the only one I knew.

A.1 On the Classification

A.1.1 Quality of the Data

Before considering the quantity of data in the samples, it is necessary to guarantee a "reasonable" quality. Some elements should perhaps be removed or weighted.

Training Samples: Strenghening and Weakening Effects

For correctly-labeled data, a classification method is useless if it provides results worse than those of allocating the elements at random (for two groups this happens when the error rate is larger than 0.5). Nevertheless, even excellent classification methods may behave worse than that if the training elements are incorrectly labelled or little representative. It is easy to imagine atypical elements in both training samples such that the rule is affected by a *training weakening effect* and therefore tends to misclassify most of the testing elements; the criterion can even be switched. Alternatively, a *training strengthening effect* appears when the atypical elements benefit the classification rule. These effects can be noticed by comparing the output of the calls of the simulation exercise in section 6.1.3. The weakening—even switching—effect can also be noticed by comparing the output of the calls 1 and 4 of the simulation exercise in section 6.1.2.

Testing Samples: Strenghening and Weakening Effects

Equivalently, we can talk about the difficulty of some elements to be classified. It is easy to imagine elements that are on "the worse side" of the representative element of their class. With many of these elements in the testing samples, the process will be affected by a *testing weakening effect*. Alternatively, a *testing strengthening effect* will appear when there are many elements on "the better side" of the representative element of their class. These effects can spoil the estimation of either the error rates or the labels. These effects can be noticed in practice by comparing the output of the three calls of the simulation exercise in section 6.1.3.

Truncated Distributions

The two previous sections highlight, on the one hand, the importance of the quality of the samples, not of the classification methods themselves; and, on the other hand, the convenience of evaluating the methods not only with several pairs of population models but also after removing the least representative data. Our depth-based robustifying technique, described in 2.6.2, can be used to avoid these effects. The technique leaves out the elements with smallest depth from both the training and the testing samples of the inner loop and from the training samples of the outer loop, while all the final testing elements are classified.

Theoretically, the "natural" proportion of little representative elements in both the training and the testing samples should be that determined by the tails of the probability distribution of the underlying stochastic models. Since obtaining these elements is little probable, leaving them out can introduce several advantages. To begin with, ignoring these elements would prevent us from working with little representative, probable samples. These samples are legitimate from a probabilistic point of view, but not the best framework to train and test procedures. Besides, in filtering the samples outliers and atypical data would be removed at the same time. To end with, the attention is focused on the most representative elements, the

kind of element with which the method has most times to deal. In short, the behaviour of the classification method can improve.

The described in the previous paragraph should not be seen as an ad-hoc manipulation of the samples but as being working with the truncated distribution of the underlying models. These distributions are widely used and has great theoretical importance in Probability Theory and Statistics. On the one hand, a disadvantage to working with them is that it is necessary to determine whether a datum is representative or not, task for which the concept of *depth measure* can play a crucial role. Our depth-based robustifying approach can be thought of as a way of working with the truncated probability distributions of the underlying stochastic models.

A.1.2 Quantity of Data

Once the data are supposed to have a "reasonable" quality, caring about the quantity of data makes sense.

Training Samples: Under- and Overfitting Effects

These effects are caused by the number of training elements. For good statistical inference procedures, the amount of data is directly related to the quality, and the *training-sample-size effect* is usually nonlinear, that is, the quality varies nonlinearly with sample size.

When there are few training elements, it is not possible to carry out a reasonable classification process, regardless the method and the quality of the data. This *underfitting effect* can be noticed in practice by running the same simulation exercise twice: with few and many training data.

Nevertheless, perhaps the most important effect related to the training process is the well-known *overfitting effect*. It appears when the same few data are used so intensively that the method behave poorly when it is applied to new data.

Testing Samples: Discreteness Effect

The amount of testing data is also related to the quality of the estimations of either the error rates and the labels. This can be called *testing-sample-size effect*.

If there are few testing data, the discretization of the possible estimations of the error rate is poor:

$$\hat{c} = \frac{k}{m_1 + m_2} \tag{38}$$

with $k \in \{0, 1, ..., m_1 + m_2\}$ being the number of misclassified elements. The step between contiguous possible values is $1/(m_1 + m_2)$. The discreteness can be noticed by running a simulation exercises with few testing sample sizes and runs. As a consequence, when searching the optimum parameter value or method, the decision-making process may be affected by a "dubiously situation" among several parameter values or methods that tend to provide the same or close estimation of the error rates—this can be called *discreteness effect*.

However, the average estimation of B runs is usually considered:

$$\hat{C} = \frac{1}{B}\sum_{i=1}^{B} \hat{c}_i = \frac{1}{B}\sum_{i=1}^{B} \frac{k_i}{m_{i,1} + m_{i,2}} \tag{39}$$

with $k_i \in \{0, 1, ..., m_{i,1} + m_{i,2}\}$. Hence, now the step between contiguous possible values is $1/(B \cdot (m_1 + m_2))$. This run-averaging weakens the effect, although it may still persit for few runs. Them, if possible we

should avoid small values for the testing sample sizes, especially for few runs; otherwise the effect, which may happen in both the inner and the outer loops, can spoil the procedure. This effect is related to the estimation of the error rates, so we need not care about the sizes of the final testing samples in the scripts for new data. To prevent this effect, our implementation includes specific code.

On the one hand, for the inner or nested loop to be applicable with small data sets, the training data of the outer cross-validation loop can optionally be used for both optimizing the parameter and estimating the final misclassification error rates (by setting `paramDataReuse = true`). Nevertheless, a too intensive reuse of the data can cause overfitting—see the training-sample-size effects of this section.

On the other hand, in the inner loop of the two first schemes of figure 1, section 2.2, we have implemented a specific approach to postpone the decision and select the parameter value or the classification method in the main loop. Concretely, for both parameter optimization and method selection a margin over the minimum is considered; that is, the parameter values or methods with estimations inside the close interval $[minimum, minimum \cdot (1 + discretenessMargin)]$ are registered to be considered again, for a second chance, in the main loop. The user can disable this interval by setting `discretenessMargin = 0`. In applying this two-step approach, some time is spent in the outer—instead of in the outer—loop. Since the probability of ties is tiny for "reasonable" sample sizes, number of runs and classification methods, this part of the code is seldom executed.

In short, there appears to be three possible solutions to mitigate the discreteness effect:

1. Increasing the testing sample sizes.
2. Increasing the number of runs.
3. Implementing the two-step approach of sections 2.5.1 and 2.5.2.

The two first proposals are statistical and can be applied only in some situations. The third is methodological and can be applied when the decision can be made in two steps, as it happens in our scripts for simulated and real data.

A.1.3 Length of the Data

For longitudinal data, we can talk about the *data-length effect* due to the length T of the stochastic processes or stochastic functions. Notice that this effect is different to those related to the methodology, that is, to the possible tasks that can be done for data of a given length—see section A.1.4.

A.1.4 Methodology Effect

When a methodology is designed, its steps introduce inevitable conditions. For example, stochastic tasks—e.g., splitting a sample—increase the variability, which can be called *randomness effect*.

When generating time series, the values with which the calculations are initialized may have an effect for short series. Therefore, to avoid this effect in the functions `funcAR2tv` and `funcARMApq` series of longer than T are generated so that to take only the last T values.

Also for times series, the length T is closely related to the quality of the periodogram as an estimator of the spectral density function. As we state in Alonso et al. (2008): *It is worth mentioning that there are two opposite effects as a consequence of splitting: one is that the narrower the blocks are, the closer we are to the locally stationary assumption; the other one is that when the length of the blocks decreases, the quality*

of the integrated periodogram as an estimator of the integrated spectrum also decreases. To show this effect, we have included some additional calls in simulation exercise of section 6.1.1.

Another example: For functions, the number of points T is closely related to the quality of the derivatives and differentials, and even some characteristics of the functions may go unnoticed for not large enough T.

Finally, it is important not to confuse the two cases just presented with that due to the training-sample--size reduction—see section A.1.2.

A.1.5 Averaging Effects

To understand—through the mean square error—the *averaging effect* of combining several classifiers, we consider the easiest case. Let e be a new element of group g_e. For independent and identically distributed classifiers with expectation $\mu(e)$ and variance $\sigma^2(e)$, the sample mean has expectation $\mu(e)$ and variance $\sigma^2(e)/B$. Thus, for this particular case, if each classifier $c_b(e)$ has mean square error

$$MSE(c_b(e)) = bias(c_b(e))^2 + var(c_b(e)) = [\mathbb{E}(c_b(e)) - g_e]^2 + var(c_b(e)) = [\mu(e) - g_e]^2 + \sigma^2(e) \tag{40}$$

the sample mean has mean square error

$$MSE(C(e)) = bias(C(e))^2 + var(C(e)) = [\mathbb{E}(C(e)) - g_e]^2 + var(C(e)) = [\mu(e) - g_e]^2 + \frac{\sigma^2(e)}{B} \tag{41}$$

In this case, for the latter classifier the bias is equal while the variance is smaller. Variability by itself does not change the mean value of a set of variables, it changes the distribution of these values around that mean.

Obviously, calculations for dependent or differently distributed classifiers c_b are quite more complicated than the previous.

On the other hand, when averaging classifiers c_b can be equal but based on different samples or, alternatively, different but based on the same samples.

What: Error Rates or Labels

Both the estimated error rates or the estimated labels can be averaged. Error rates are averaged in the inner loop of all scripts and in the outer loop of scripts for simulated and real data. On the contrary, labels are averaged in the outer loop of the scripts for new data.

Where: Testing Samples or Runs

As regards the place, it is worth highlighting that in estimating the overall misclassification error rate individual error rates are implicitly averaged when the testing samples have more than one datum—this can be called *sample-averaging effect*.

Besides, if this estimation of the overall error rate is repeated and averaged, an explicit *run-averaging effect* appears.

Number of Runs and Estimations

On the one hand, the number of runs, say B, should not be too small for several reasons:

- To avoid the discreteness effect described in section A.1.2, especially for small testing samples.

- To guarantee a minimum quality of the estimated error rates and labels, since the error of an estimation must be of smaller order of magnitude than the estimation itself. Let Q be a quantity with mean $\mu = \mathbb{E}(Q)$ and variance mean $\sigma^2 = Var(Q)$; if its mean is being estimated through the sample mean $\bar{Q} = \frac{1}{B}\sum_{i=1}^{B} Q_i$, where $\{Q_i\}$ is a simple random sample, the standard error of the estimation is defined as the standard deviation of the sample mean, that is,

$$SEE(Q) = \sqrt{\frac{\sigma^2}{B}} = \frac{\sigma}{\sqrt{B}}, \tag{42}$$

  which is in practice—unknown σ—approximated by

$$S\hat{E}E(Q) = \frac{s}{\sqrt{B}}, \tag{43}$$

  where s is the sample variance.

(This formula holds for independent and identically distributed classifiers, basically for procedures where methods do not change the samples.) From this formula it is easy to see that $B = 100$ causes the standard error to be one order of magnitude smaller than the estimation (error around 10% of the quantity itself), the value $B = 400$ would imply the order of magnitude to be reduced in one degree and the magnitude to be divided by 2, the value $B = 10000$ would imply a standard error with an order of magnitude two degrees smaller than the estimation, et cetera.

On the other hand, the number of runs should not be too large for other reasons:

- When the same data are subsequently used, to avoid the overfitting effect described in section A.1.2.
- For computational, practical reasons.

Run-Averaging of the Labels: Majority-Vote Classifier

In a two-group problem, a classifier can be seen as a function that takes the value -1 or $+1$, that is, $c(e) \in \{-1, +1\}$. For subsequent estimations, say B, the final unweighted majority-vote final classifier is given by

$$C(e) = sign[\sum_{b=1}^{B} c_b(e)]. \tag{44}$$

If the B runs are independent, they can be considered simultaneous; otherwise, they are dependent. As an example of the latter case, the *boosting* techniques generate and combine a sequence of classifiers—based on dependent modifications of the samples—in a weighted summation $C(e) = sign[\sum_{b=1}^{B} \alpha_b c_b(e)]$. As Hastie et al. (2001) state: *The motivation for boosting was a procedure that combines the outputs of many "weak" classifiers to produce a powerful "committee"*. Averaging "weak" or unstable classifiers can reduce the variance of the final decision.

The result of expression 44 can be thought of as either the sample mean or the median of variables taking the value -1 or $+1$. It is well-known the robustness of the median, while, for a variable taking the values mentioned, the sample mean is in $[-1, +1]$. We have carried out a simulation exercise to calculate the mean square error of c_b and C—see section 6.1.4.

A.2 On the Computational Time

The following time effects can be negligible when their order of magnitude is quite bigger than the order of magnitude of the most lasting step of an algorithm, or when we shall pay attention to the averaged computational time of many runs. Alternatively, the effects are difficult to detect for fast methods—so happens with our algorithms *WI* and *WD*.

A.2.1 Sample-Size Time Effect

Computational time depends on the sample sizes. In general, the more data in the samples, the bigger the necessary computational time to extract the information from them.

The, as a consequence of using fewer—but usually better—elements in the samples, there may be a time-saving effect if the time spent calculating the depth is compensated, especially for methods that use each datum individually during the calculations, while the quality of the classification is not affected.

A.2.2 Auxiliary-Method Effect

When a technique calls other "secondary" or auxiliary procedures, they may introduce perturbations—*auxiliary-method effects*—in the computational effort of the technique itself. For example, when working with the periodogram (it is an estimator of the power spectral density), the length T of the processes or series is recommended to be a power of two due to the fact that in this case the Fast Fourier Transform can be applied to calculate it—see, for example, section 6.1.3 of Priestley (1981). Hence, if the user splits the data into blocks such that some of them verify that condition while some others do not, there will appear an effect inherited from the way of calculating the periodogram.

A.2.3 Call-Order Effect

In any multistep procedure, some calculations can be reused when:

1. Several methods are applied to just the same data.
2. Consecutive methods have common initial steps.

With such an implementation, the order in which the methods are called may introduce a perturbance—a *call-order effect*—in the computational times. We have accelerated the calculations in this way as described in section 2.8; however, to avoid the perturbance our code registers the computational time spent by the method that did the calculations. To take advantage of the implementation, methods should be implemented for them to share the first steps, for example, `[1 1 2 1; 1 1 3 2, 1 1 3 4]`.

A.2.4 Inherent Effect

The computer, the operating system and the programming language introduce inevitable perturbances—*inherent effects*—in the computational times.

While preparing this package, I have met an interesting example that at the beginning I thought it was a programming error. Concretely, the first time some code is run in a `for`-loop, it may last more than in posterior times. A tricky solution consists in doing the calculations one more time than necessary; if the user needs fast code or has no interest in the accuracy of the computational time, this trick can be removed. See the simulation exercise in section 6.1.6 for a particular case of this *beginner effect*.

A.3 Total Effect

So far we have described isolated effects. Nevertheless, they are usually involved in the *total effect* of a complex process. Some individual effects are positive and some others are negative, in the sense of being or not beneficial for the process. Although it is usually impossible to quantify the total effect in a process, we can apply some qualitative reasoning so that to understand the whole process. For this purpose, it is possible to relate the elements of a methodology or scheme with some of the effects, for example:

- In both the inner and the outer loops, data are split in each run. Thus, there are a negative randomness effect, negative training- and testing-sample-size effects, a positive run-averaging effect, etc.
- When leave-one-out is applied, the reduction of the sample sizes is minimum, but so is the sample--averaging effect.
- Reusing or not the data in the inner loop can involve training- and testing-sample-size effects, but also an overfitting effect.
- The depth-based robustifying technique introduces weak negative training- and testing-sample-size effects in the inner loop and a training-sample-size effect in the outer loop, but also reduces the training and testing strengthening and weakening effects, and implies a sample-size time effect.
- For time series, in the calculation of the periodogram there is an auxiliary-method effect and, when many blocks are considered, also a methodology effect.
- The schemes themselves and their calls may imply a call-order effect, inherent effects, et cetera.

B Ways of Using the Data: Expanded Figures

References

[1] Alonso, A.M., D. Casado, S. López-Pintado and J. Romo (2008) A Functional Data Based Method for Time Series Classification. *Working paper*. Departamento de Estadística. Universidad Carlos III de Madrid. Available at `http://hdl.handle.net/10016/3381`.

[2] Alonso, A.M., D. Casado and J. Romo (2012) Supervised Classification for Functional Data: A Weighted Distance Approach. *Computational Statistics and Data Analysis*. 56 (7), 2334–46

[3] Casado, D. (2010). Classification Techniques for Time Series and Functional Data. *Doctoral Thesis*. Available at `http://www.Casado-D.org/edu/Casado-Thesis.pdf`.

[4] Ferraty, F., and P. Vieu (2003). Curves Discrimination: A Nonparametric Functional Approach. *Computational Statistics and Data Analysis*. 44, 161–173.

[5] Gerald, C.F., and P.O. Wheatley (1999). *Applied Numerical Analysis*. Addison-Wesley

[6] Gibaldi, J. (1998). *MLA Style Manual and Guide to Scholarly Publishing*. The Modern Language Association of America (New York)

[7] Hastie, T., R. Tibshirani and J. Friedman (2001). *The Elements of Statistical Learning*. Springer

Figure 19: A way of using simulated data samples

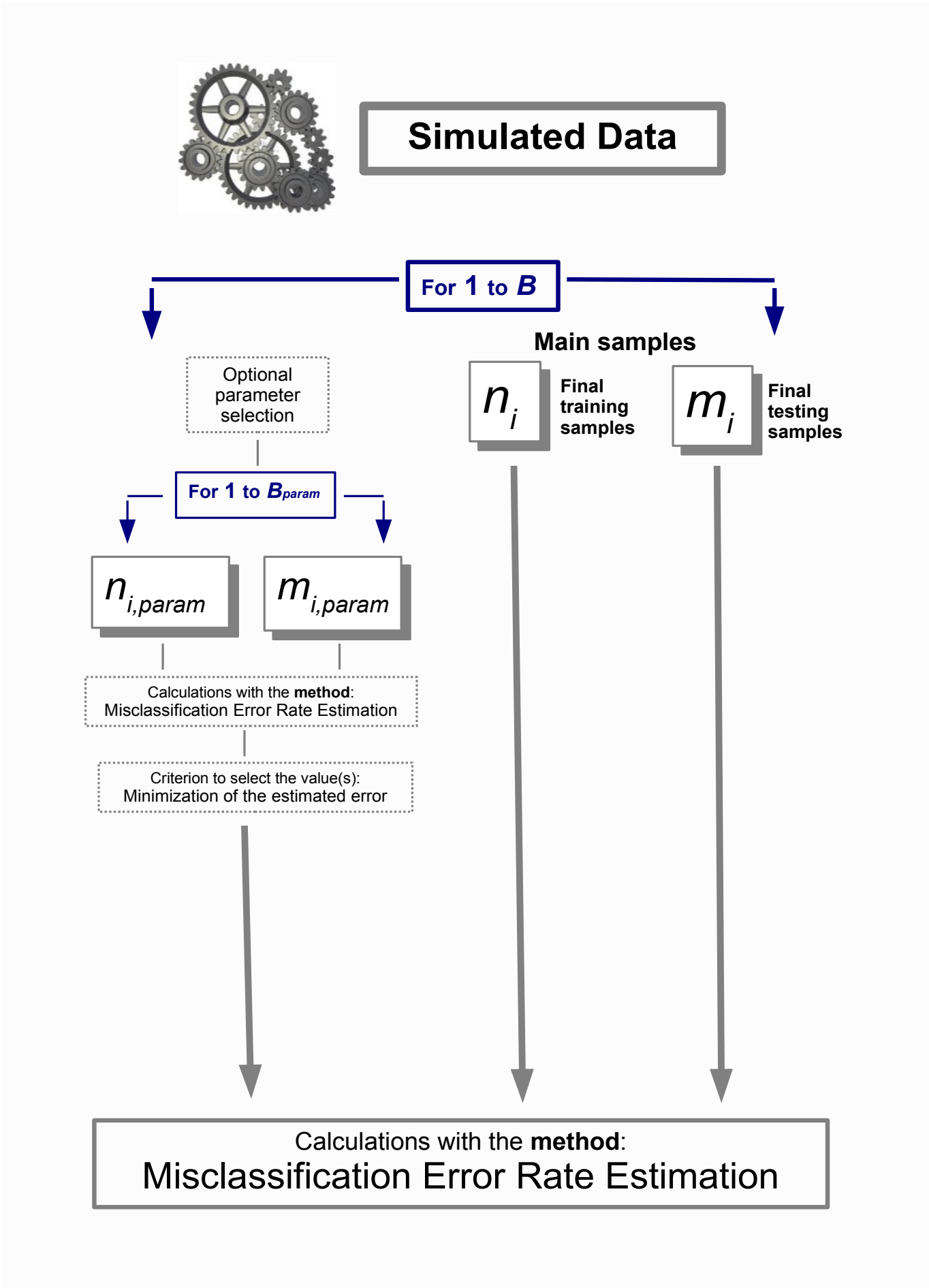


Figure 20: A way of using real data samples

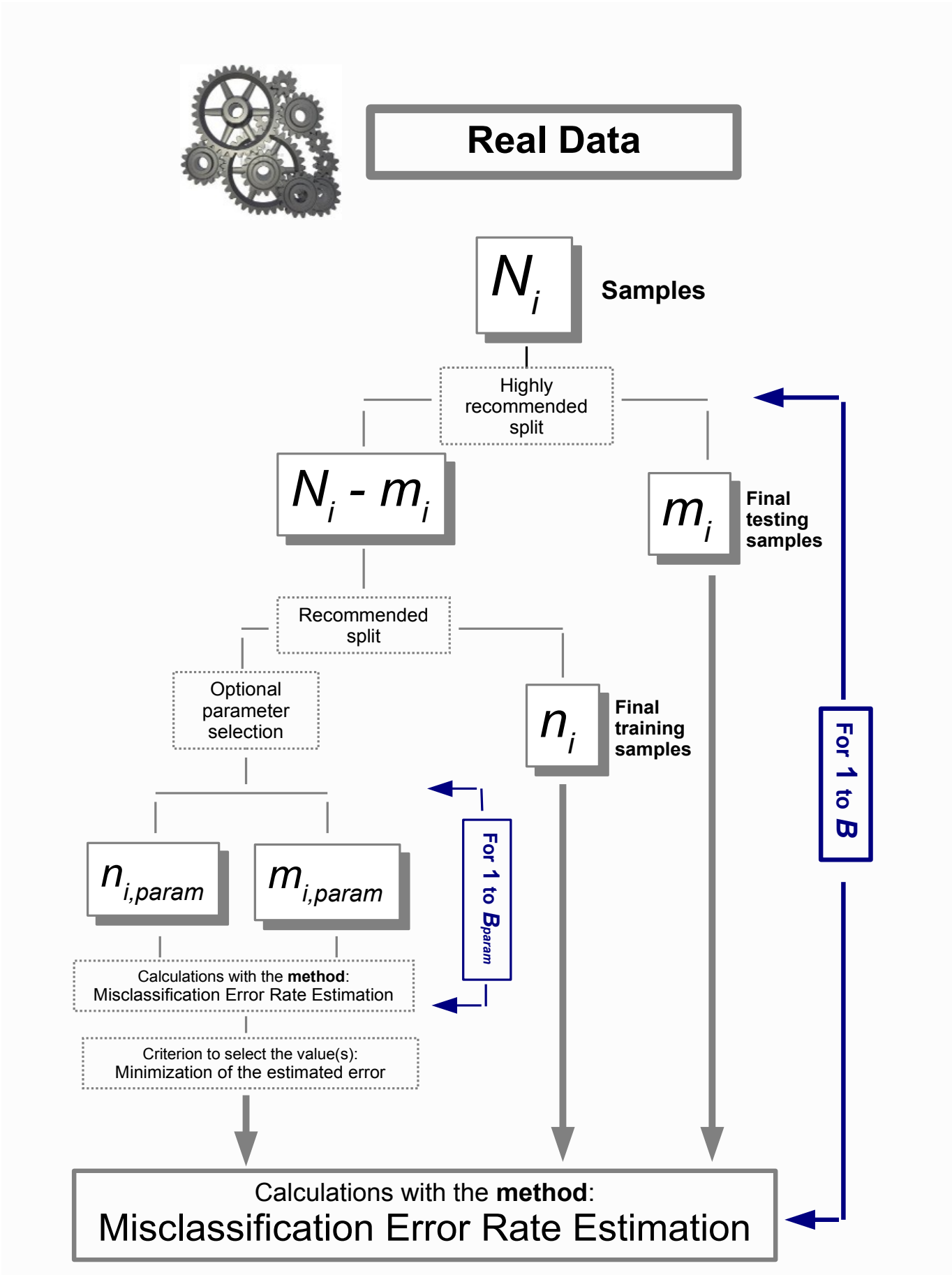
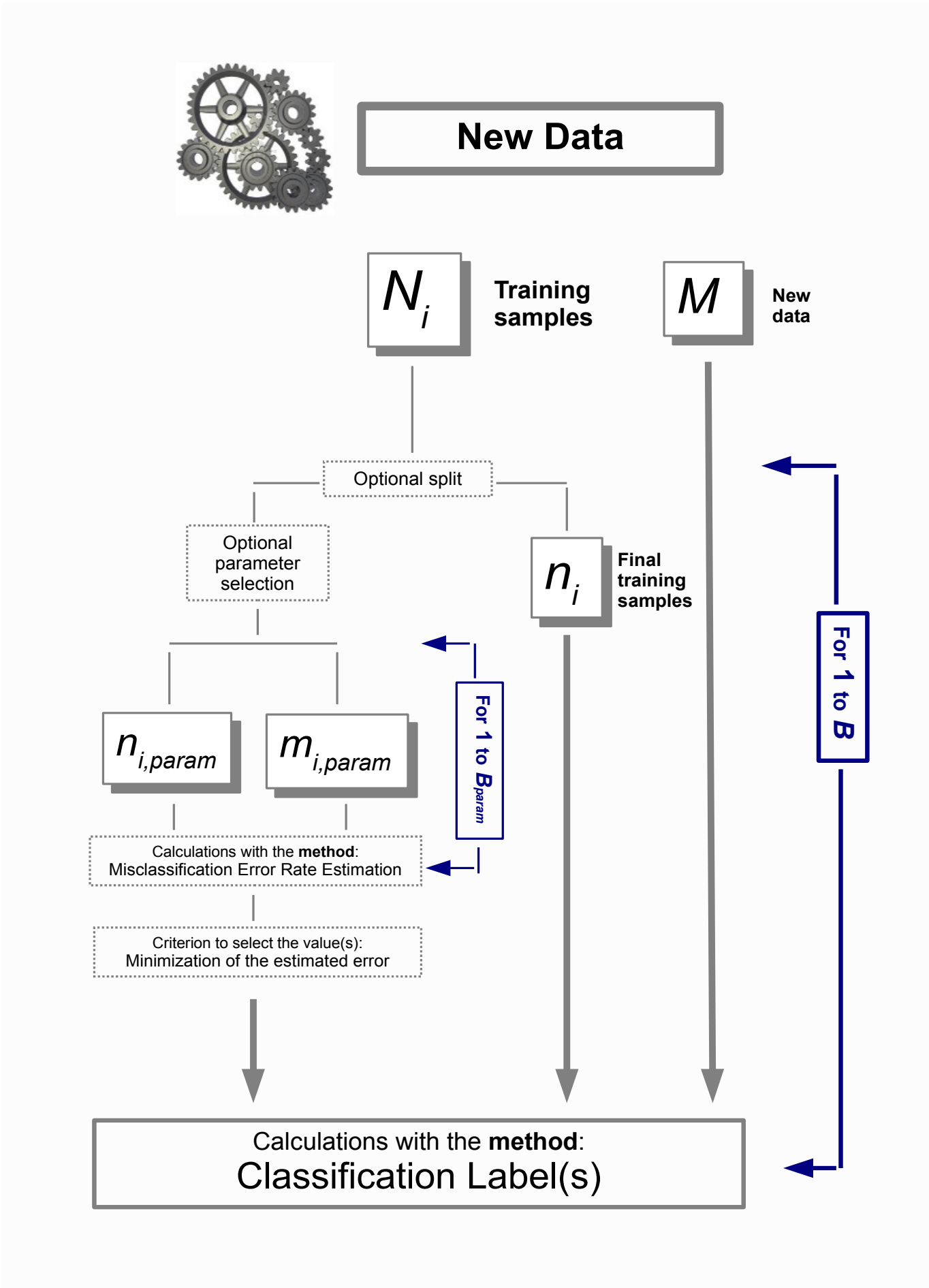


Figure 21: A way of using training samples to classify new data



[8] Huang, H., H. Ombao and D.S. Stoffer (2004). Discrimination and Classification of Nonstationary Time Series Using the SLEX Model. *Journal of the American Statistical Association*. 99 (467), 763–774.

[9] López-Pintado, S., and J. Romo (2006). Depth-Based Classification for Functional Data. In *Data Depth: Robust Multivariate Analysis, Computational Geometry and Applications*. American Mathematical Society. DIMACS Series, 72, 103–121. (R. Liu, R. Serfling and D.L. Souvaine [eds]).

[10] Priestley, M.B. (1981). *Spectral Analysis and Time Series*. Elsevier.

[11] MathWorks, (2010). MATLAB *Software*. Version R2010b. `http://www.matlab.com`.

[12] Swan, M. (1995). *Practical English Usage*. Oxford University Press.