MASTER'S THESIS

Interuniversity Master in Statistics and Operations Research UPC-UB

Title: Multidimensional Scaling for Big Data

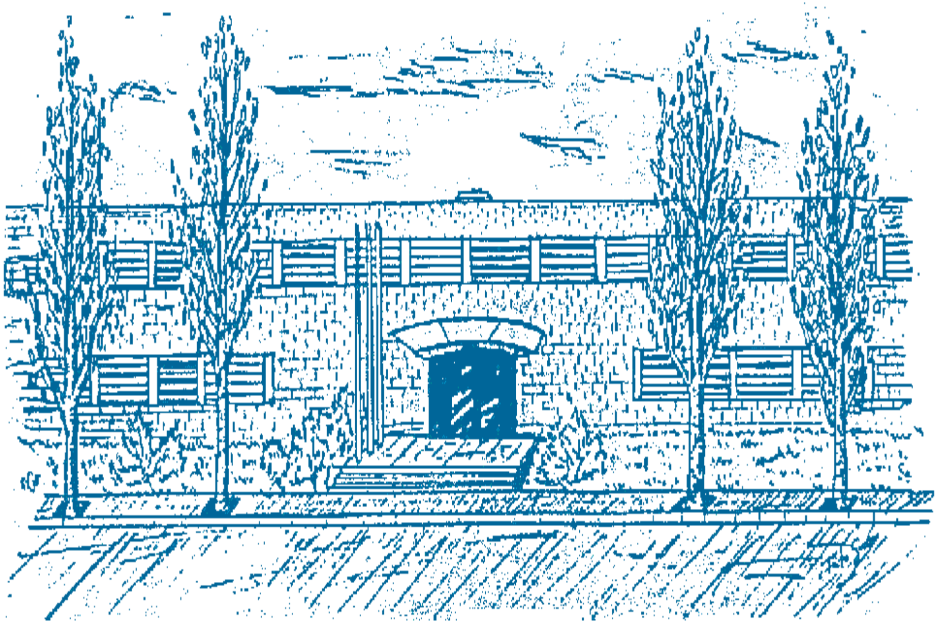
Author: Cristian Pachón García

Advisor: Pedro Delicado Useros

Department: Dept. d'Estadística i Investigació Operativa

University: Universitat Politècnica de Catalunya - Universitat de Barcelona

Academic year: 2018-2019



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

UNIVERSITAT DE BARCELONA

Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Master's thesis

Multidimensional Scaling for Big Data

Cristian Pachón García

Advisor: Pedro Delicado Useros

Dept. d'Estadística i Investigació Operativa

Abstract

We present a set of algorithms for *Multidimensional Scaling* (MDS) to be used with large datasets.

MDS is a statistic tool for reduction of dimensionality, using as input ~~data~~ a distance matrix of dimensions $n \times n$. When n is large, classical algorithms suffer from computational problems and MDS configuration can not be obtained.

In this thesis we address these problems by means of three algorithms: *Divide and Conquer MDS, Fast MDS* and *MDS based on Gower interpolation*. The main idea of these methods is based on partitioning the dataset into small pieces, where the classical methods can work.

In order to check the performance of the algorithms as well as to compare them, we do a simulation study. This study points out that *Fast MDS* and *MDS based on Gower interpolation* are appropriated to use when n is large.

Contents

Chapter 1

Classical Multidimensional Scaling

1.1 Introduction to Multidimensional Scaling

Multidimensional Scaling (MDS) is a family of methods that represents measurements of dissimilarity (or similarity) among pairs of objects as Euclidean distances between points of a low-dimensional space. The data, for example, may be correlations among intelligence tests and the MDS representation is a plane that shows the tests as points. The graphical display of the correlations provided by MDS enables the data analyst to literally "look" at the data and to explore their structure visually. This often shows regularities that remain hidden when studying arrays of numbers.

Given a square matrix $\mathbf{D}$ $n \times n$, the goal of MDS is to obtain a configuration matrix $\mathbf{X}$ $n \times p$ with orthogonal columns that can be interpreted as the matrix of p variables for the n observations, where the Euclidean distance between the rows of $\mathbf{X}$ is approximately equal to $\mathbf{D}$. The columns of $\mathbf{X}$ are called *principal coordinates.*

This approach arises two questions: is it (always) possible to find this low dimensional configuration $\mathbf{X}$? How is it obtained? In general, it is not possible to find a set of p variables that reproduces *exactly* the initial distance. However, it is possible to find a set of variables which distance is approximately the initial distance matrix $\mathbf{D}$.

As classical example, consider the distances between European cities as in the Table 1.1. One would like to get a representation in a 2-dimensional space such that the distances would be almost the same as in the Table 1.1. The representation of these coordinates are displayed in Figure 1.1.

	Athens	Barcelona	Brussels	Calais	Cherbourg	$\cdots$
Athens	0	3313	2963	3175	3339	$\cdots$
Barcelona	3313	0	1318	1326	1294	$\cdots$
Brussels	2963	1318	0	204	583	$\cdots$
Calais	3175	1326	204	0	460	$\cdots$
Cherbourg	3339	1294	583	460	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Table 1.1: Distances between European cities (just 5 of them are shown).

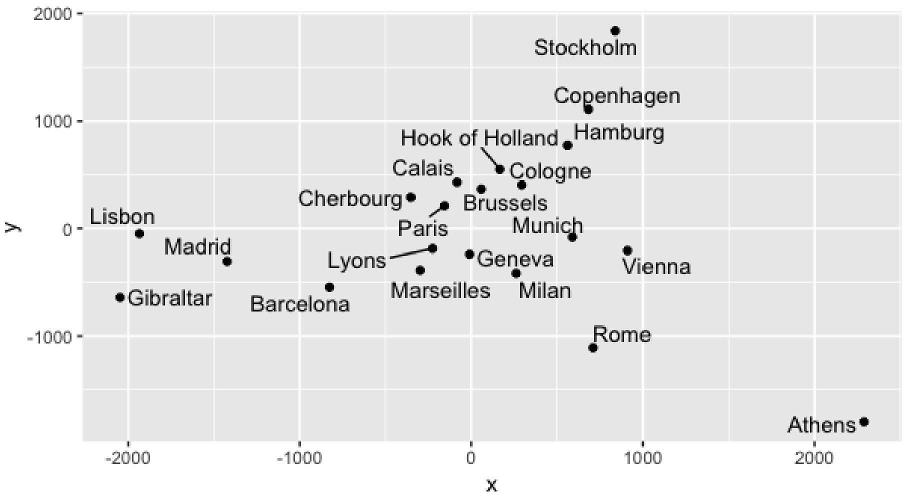


Figure 1.1: MDS on the European cities.

1.2 Principal coordinates

Given a matrix $\mathbf{X}$ $n \times p$, the matrix of n individuals over p variables, it is possible to obtain a new one with mean equal to 0 by column from the previous one:

$$\widetilde{\mathbf{X}} = \Big(\mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}'\Big)\mathbf{X} = \mathbf{P}\mathbf{X},$$

where

$$\mathbf{P} = \Big(\mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}'\Big).$$

This new matrix, $\widetilde{\mathbf{X}}$, has the same dimensions as the original one but its columns mean is $\mathbf{0}$. From this matrix, it is possible to build two square semi-positive definite matrices: the covariance matrix $\mathbf{S}$, defined as $\widetilde{\mathbf{X}}'\widetilde{\mathbf{X}}/n$ and the cross-products matrix $Q = \widetilde{\mathbf{X}}\widetilde{\mathbf{X}}'$. The last matrix can be interpreted as a similarity matrix between the n elements. The term ij is obtained as follows:

$$q_{ij} = \sum_{s=1}^{p} x_{is}x_{js} = \mathbf{x_i}'\mathbf{x_j} \tag{1.1}$$

where $\mathbf{x_i}'$ is the i-th row from $\widetilde{\mathbf{X}}$.

Given the scalar product formula, $\mathbf{x_i}'\mathbf{x_j} = \mid \mathbf{x_i} \mid\mid \mathbf{x_i} \mid \cos\theta_{ij}$, if the elements i and j have similar coordinates, then $\cos\theta_{ij} \simeq 1$ and q_{ij} will be large. On the contrary, if the elements are very different, then $\cos\theta_{ij} \simeq 0$ and q_{ij} will be small. So, $\widetilde{\mathbf{X}}\widetilde{\mathbf{X}}'$ can be interpreted as the similarity matrix between the elements.

The distances between elements can be deduced from the similarity matrix. The Euclidean distance between two elements is calculated in the following way:

$$d_{ij}^2 = \sum_{s=1}^{p}(x_{is} - x_{js})^2 = \sum_{s=1}^{p} x_{is}^2 + \sum_{s=1}^{p} x_{js}^2 - 2\sum_{s=1}^{p} x_{is}x_{js}. \tag{1.2}$$

This expression can be obtained directly from the matrix $\mathbf{Q}$:

$$d_{ij}^2 = q_{ii} + q_{jj} - 2q_{ij}. \tag{1.3}$$

We have just seen that, given the matrix $\widetilde{\mathbf{X}}$, it is possible to get the similarity matrix $\mathbf{Q} = \widetilde{\mathbf{X}}\widetilde{\mathbf{X}}'$ and from it, to get the distance matrix $\mathbf{D}$. Let diag($\mathbf{Q}$) be the vector that contains the diagonal terms of $\mathbf{Q}$ and $\mathbf{1}$ be the vector of ones, the matrix $\mathbf{D}$ is given by

$$\mathbf{D} = \operatorname{diag}(\mathbf{Q})\mathbf{1}' + \mathbf{1}\operatorname{diag}(\mathbf{Q})' - 2\mathbf{Q}.$$

The problem we are dealing with goes in the opposite direction. We want to rebuild $\widetilde{\mathbf{X}}$ from a square distance matrix $\mathbf{D}$, with elements d_{ij}^2. The first step is to obtain $\mathbf{Q}$ and afterwards, to get $\widetilde{\mathbf{X}}$. The theory needed to get the solution can be found in (Peña 2002). Here, we summarise it.

The first step is to find out a way to obtain the matrix $\mathbf{Q}$ given $\mathbf{D}$. We can assume without loss of generality that the mean of the variables is equal to 0. This is a consequence of the fact that the distance between two points remains the same if the variables are expressed in terms of the mean:

$$d_{ij}^2 = \sum_{s=1}^{p}(x_{is} - x_{js})^2 = \sum_{s=1}^{p}[(x_{is} - \overline{x_s}) - (x_{js} - \overline{x_s})]^2. \tag{1.4}$$

The previous condition means that we are looking for a matrix $\widetilde{\mathbf{X}}$ such that $\widetilde{\mathbf{X}}'\mathbf{1} = 0$. It also means that $\mathbf{Q}\mathbf{1} = 0$, i.e, the sum of all the elements of a column of $\mathbf{Q}$ is 0. Since the matrix is symmetric, the previous condition should state for the rows as well.

To establish these constrains, we sum up (1.3) at row level:

$$\sum_{i=1}^{n} d_{ij}^2 = \sum_{i=1}^{n} q_{ii} + nq_{jj} = t + nq_{jj} \tag{1.5}$$

where $t = \sum_{i=1}^{n} q_{ii} = \text{Trace}(\mathbf{Q})$, and we have used that the condition $\mathbf{Q}\mathbf{1} = 0$ implies $\sum_{i=1}^{n} q_{ij} = 0$. Summing up (1.3) at column level:

$$\sum_{j=1}^{n} d_{ij}^2 = t + nq_{ii}. \tag{1.6}$$

Summing up (1.5) we obtain:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}^2 = 2nt \tag{1.7}$$

Replacing in (1.3) q_{jj} obtained in (1.5) and q_{ii} obtained in (1.6), we have the following expression:

$$d_{ij}^2 = \frac{1}{n}\sum_{i=1}^{n} d_{ij}^2 - \frac{t}{n} + \frac{1}{n}\sum_{j=1}^{n} d_{ij}^2 - \frac{t}{n} - 2q_{ij} \tag{1.8}$$

Let $d_{i.}^2 = \frac{1}{n}\sum_{j=1}^{n} d_{ij}^2$ and $d_{.j}^2 = \frac{1}{n}\sum_{i=1}^{n} d_{ij}^2$ be the row-mean and column-mean of the elements of $\mathbf{D}$. Using (1.7), we have that

$$d_{ij}^2 = d_{i.}^2 + d_{.j}^2 - d_{..}^2 - 2q_{ij} \tag{1.9}$$

where $d_{..}$ is the mean of all the elements of $\mathbf{D}$, given by

$$d^2_{..} = \frac{1}{n^2}\sum\sum d^2_{ij}.$$

Finally, from (1.9) we get the expression:

$$q_{ij} = -\frac{1}{2}(d^2_{ij} - d^2_{i.} - d^2_{.j} + d^2_{..}). \tag{1.10}$$

The previous expression shows how to build the matrix of similarities $\mathbf{Q}$ from the distance matrix $\mathbf{D}$.

The next step is to obtain the matrix $\mathbf{X}$ given the matrix $\mathbf{Q}$. Let's assume that the similarity matrix is positive definite of range p. Therefore, it can be represented as

$$\mathbf{Q} = \mathbf{V\Lambda V}'$$

where $\mathbf{V}$ is a $n \times p$ matrix that contains the eigenvectors with not nulls eigenvalues of $\mathbf{Q}$. $\mathbf{\Lambda}$ is a diagonal matrix $p \times p$ that contains the eigenvalues.

Re-writing the previous expression, we obtain

$$\mathbf{Q} = (\mathbf{V\Lambda}^{1/2})(\mathbf{\Lambda}^{1/2}\mathbf{V}'). \tag{1.11}$$

Getting

$$\mathbf{Y} = \mathbf{V\Lambda}^{1/2}.$$

We have obtained a matrix with dimensions $n \times p$ with p uncorrelated variables that reproduce the initial metric. It is important to notice that if one starts from $\mathbf{X}$ (i.e $\mathbf{X}$ is known) and calculates from these variables the distance matrix in (1.2) and after that it is applied the method explained, the matrix obtained is not the same as $\mathbf{X}$, but its principal components. This happens since the distance between the rows in a matrix does not change if:

- The row-mean values are modified by adding the same row vector to all the rows in $\mathbf{X}$.
- Rows are rotated, i.e, $\mathbf{X}$ is postmultiplied by an orthogonal matrix.

By (1.3), the distance is a function of the terms of the similarity matrix $\mathbf{Q}$ and this matrix is invariant given any rotation, reflection or translation of the variables

$$\mathbf{Q} = \widetilde{\mathbf{X}}\widetilde{\mathbf{X}'} = \widetilde{\mathbf{X}}\mathbf{AA}'\widetilde{\mathbf{X}'}$$

for any orthogonal $\mathbf{A}$ matrix. The matrix $\mathbf{Q}$ only contains information about the space generated by the variables $\mathbf{X}$. Any rotation, reflection or translation preserves the distance. Therefore, the solution is not unique.

1.3 Building principal coordinates

Let $\mathbf{D}$ be a square distance matrix. The process to obtain the *principal coordinates* is as follows:

1. Build the matrix $\mathbf{Q} = -\frac{1}{2}\mathbf{PDP}$ of cross-products.

2. Obtain the eigenvalues of $\mathbf{Q}$. Take the r greatest eigenvalues. Since $\mathbf{P1} = 0$, where $\mathbf{1}$ is a vector of ones, range($\mathbf{Q}$) $= n - 1$, being the vector $\mathbf{1}$ an eigenvector with eigenvalue 0.

3. Obtain the coordinates of the rows in the variables $\mathbf{v_i}\sqrt{\lambda_i}$, where λ_i is an eigenvalue of $\mathbf{Q}$ and $\mathbf{v_i}$ is the associated unitary eigenvector. This implies that $\mathbf{Q}$ is approximated by

$$\mathbf{Q} \approx (\mathbf{V_r}\mathbf{\Lambda}^{1/2})(\mathbf{\Lambda_r}^{1/2}\mathbf{V}'_\mathbf{r}).$$

4. Take as coordinates of the points the following variables:

$$\mathbf{Y_r} = \mathbf{V_r}\mathbf{\Lambda_r}^{1/2}.$$

The method can also be applied if the initial information is not a distance matrix but a similarity matrix. A *similarity function* is characterized by the following properties (s_{ij} denotes the similarity between element i and j):

- $s_{ii} = 1$.
- $0 \le s_{ij} \le 1$.
- $s_{ij} = s_{ji}$.

If the initial information is $\mathbf{Q}$, a similarity matrix, then $q_{ii} = 1$, $q_{ij} = q_{ji}$ and $0 \le q_{ij} \le 1$. The associated distance matrix is

$$d_{ij}^2 = q_{ii} + q_{jj} - 2q_{qij} = 2(1 - q_{ij}),$$

and it is easy to see that $\sqrt{2(1 - q_{ij})}$ is a distance and it verifies the triangle inequality.

1.4 Procrustes transformation

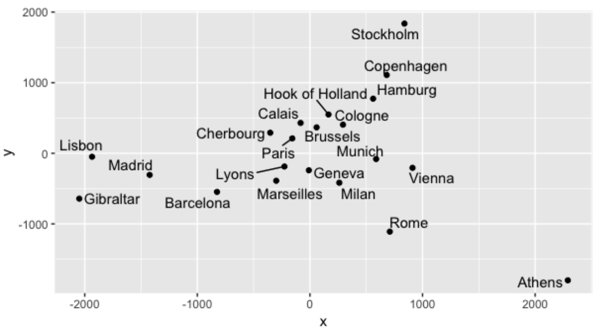
As we have mentioned before, the MDS solution is not unique. One example of it is shown in Figure 1.2.

Since rotations, translations and reflections are distance-preserving functions, one can find two different MDS configurations for the same set of data. How is it possible to align both solutions? *Align both solutions* (or multiple ones) means to find a common coordinate system for all the solutions, i.e, let $\mathbf{MDS}_1$ and $\mathbf{MDS}_2$ be two MDS solutions of dimensions $n \times r$. We say they are aligned if the coordinates of row i are the same in both solutions:

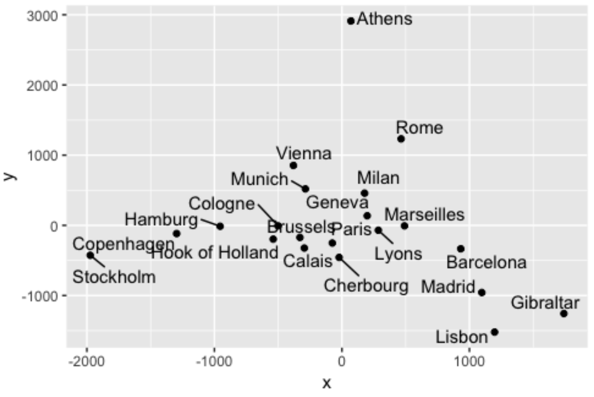
$$mds_{i1}^1 = mds_{i1}^2, \ldots, mds_{ir}^1 = mds_{ir}^2$$

where mds_{ij}^k is the coordinates j for the individual i given the solution k, $j \in \{1, \ldots r\}$, $i \in \{1, \ldots, n\}$ and $k \in \{1, 2\}$.

This problem is solved by means of *Procrustes transformations.* The Procrustes problem is concern with fitting a configuration (testee) to another (target) as closely as possible. In the simple case, both configurations have the same dimensionality and the same number of points, which can be brought into 1-1 correspondence. Under orthogonal



(a)



(b)

Figure 1.2: Two different solutions of MDS.

transformations, the testee can be fitted it to the target. In addition to such rigid motions, one may also allow for dilations and for shifts.

All the details are developed in Borg and Groenen (2005). This is out of the scope of this thesis. However, since it has been a repeatedly used tool, we briefly summarise it.

Let **A** and **B** be two different MDS configurations of dimensions $n \times t$ for the same set of data. Without loss of generality, let's assume that the target is **A** and the testee is **B**. One wants to obtain $s \in \mathbb{R}$, $\mathbf{T} \in \mathrm{M}_{r\times r}(\mathbb{R})$ and $\mathbf{t} \in \mathbb{R}^r$ such that

$$\mathbf{A} = s\mathbf{B}\mathbf{T} + \mathbf{1}\mathbf{t}'$$

where **T** is an orthogonal matrix. As mentioned before, in Borg and Groenen (2005) are all the details needed to estimate these parameters.

1.5 Multidimensional Scaling with R

All the algorithms have been coded in R. We have used two packages for developing our MDS approaches:

- Package: stats. From this one we have used the function cmdscale to do the MDS. The output of this function is a list of two elements:
  - The first r principal coordinates for the individuals, i.e, the low-dimensional configuration for the data.

  - All the eigenvalues found. If the dimensions of the initial dataset are $n \times k$, then there are n eigenvalues.

- Package: MCMCpack. From this one we have used the function procrustes to do the Procrustes transformation. The output of this function is a list of three elements:

  - The dilation coefficient s.
  - The orthogonal matrix $\mathbf{T}$.
  - The translation vector $\mathbf{t}$.

Chapter 2

Algorithms for Multidimensional Scaling with Big Data

2.1 Why is it needed?

In this chapter we present the algorithms developed so that MDS can be applied when we are dealing with large datasets. The natural question here is why we need them if there are already some implementations. To answer this question, let's take a look at the Figures 2.1 and 2.2.

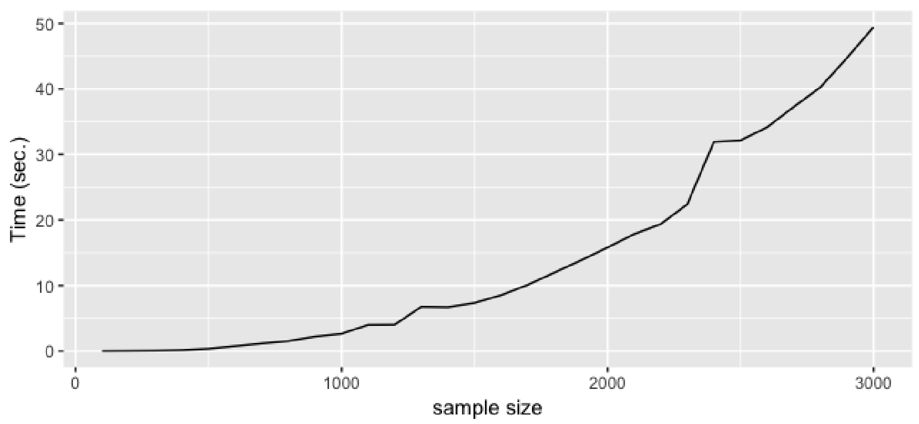


Figure 2.1: Elapsed time to compute MDS.

Figure 2.1 shows the time needed to compute MDS as a function of the sample size. As we can see, the time grows considerably as the sample size increases when using `cmdscale` function. Apart of the time issue, there is another one related to the memory needed to compute the distance matrix. Figure 2.2 points out that it is required at least 400MB to store the distance matrix when the dataset is close to 10^4 observations.

In order to solve these problems, we have considered to work on three algorithms:

- *Divide and Conquer MDS:* Before this thesis, *Pedro Delicado* had done some work about MDS with big datasets and he had already created a first approach, which is this one. The algorithm is based on the idea of dividing and conquering. Given a big dataset, it is divided into p partitions. After that, MDS is performed over all the partitions and, finally, the p solutions are stitched so that all the points lie on the same coordinate system.

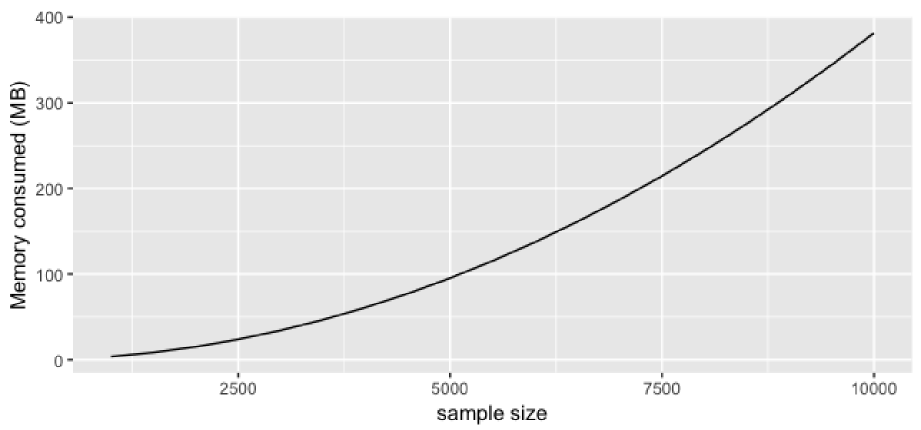


Figure 2.2: Memory consumed to compute distance.

- *Fast MDS:* during the phase of gathering information, we found an article that solved the problem of scalability (Tynia, Jinze, Leonard, and Wei 2006). The authors use a divide and conquer idea together with recursive programming.
- *MDS based on Gower interpolation:* this algorithm uses Gower interpolation formula, which allows to add a new set of points to an existing MDS configuration. For further details see, for instance, the Appendix of (Gower and Hand 1996).

In the next sections we provide a description of the algorithms. If further details about the implementation are needed, the code is provided in Appendix D.

2.2 Divide and Conquer MDS

2.2.1 Algorithm

- The first step is to divide the original dataset into p partitions: $\mathbf{X_1}, \ldots, \mathbf{X_p}$. The number of partitions, p, is also the number of steps needed to compute the algorithm.
- Calculate the MDS for the first partition: $\mathbf{MDS(1)}$. This solution will be used as a guide to align the MDS for the remaining partitions. We use a new variable, **cum-mds**, that will be growing as long as new partitions are used. Before adding a new MDS, it is aligned and, after that, added.
- Define **cum-mds** equal to $\mathbf{MDS(1)}$ and start iterating until the last partition is reached.
- Given a step k, $1 < k \leq p$, partitions k and *k-1* are joint, i.e, $\mathbf{X_k} \cup \mathbf{X_{k-1}}$. MDS is calculated on this union, obtaining $\mathbf{MDS_{k,k-1}}$. In order to add the rows of the *k-th* partition to **cum-mds**, the following steps are performed:
  - Take the rows of the partition *k-1* from $\mathbf{MDS_{k,k-1}}$: $\mathbf{MDS_{k,k-1}}\Big|_{k-1}$.
  - Take the rows of the partition *k-1* from **cum-mds**: **cum-mds** $\Big|_{k-1}$.

– Apply Procrustes to align both solutions. It means that a scalar number s, a vector $\mathbf{t}$ and an orthogonal matrix $\mathbf{T}$ are obtained so that

$$\text{cum-mds}\Big|_{k-1} \approx s\mathbf{MDS_{k,k-1}}\Big|_{k-1}\mathbf{T} + \mathbf{1t}'.$$

– Take the rows of the partition k from $\mathbf{MDS_{k,k-1}}$: $\mathbf{MDS_{k,k-1}}\Big|_{k}$.

– Use the previous Procrustes parameters to add the rows of $\mathbf{MDS_{k,k-1}}\Big|_{k}$ to **cum-mds**:

$$\text{cum-mds}_k := s\mathbf{MDS_{k,k-1}}\Big|_{k}\mathbf{T} + \mathbf{1t}'.$$

– Add the previous dataset to **cum-mds**, i.e:

$$\text{cum-mds} = \text{cum-mds} \cup \text{cum-mds}_k$$

As we have seen, the algorithm depends on p, the number of partitions. How many of them are needed? To answer this question, let $l \times l$ be the size of the largest matrix that allows to run MDS efficiently, i.e, in a reasonable amount of time. If n is the number of rows of $\mathbf{X}$, then p is $\frac{2n}{l}$.

2.2.2 Some indicators about the performance of the algorithm

The aim of this section is to show some indicators about the performance of the algorithm. A deeper analysis is done in Chapter 3, where more details are provided.

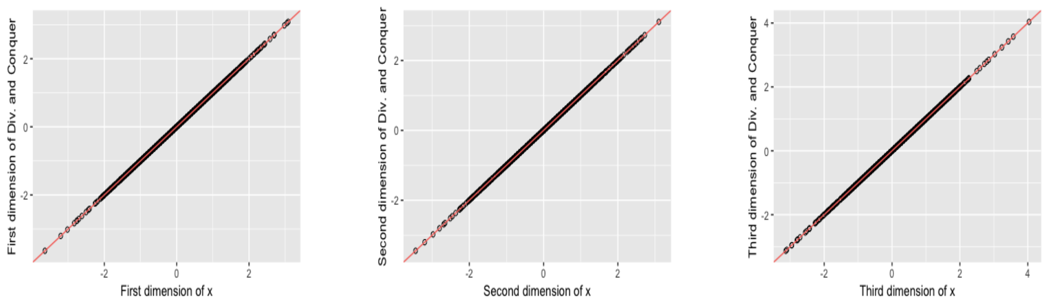
We have generated a matrix $\mathbf{X}$ with 3 independent *Normal* distributions ($\mu = 0$ and $\sigma = 1$) and 10^3 rows with l equals to 500. Afterwards, we have run the algorithm. We have required the algorithm to return 3 columns. So, a new matrix with 3 columns and 10^3 rows ($\mathbf{MDS_{Div}}$) has been obtained. Both matrices should be "equal" with an exception of either a rotation, translation or reflection, but not a dilation. We have not allowed dilations to see that the distance is preserved.

To align the matrices we have performed a Procrustes transformation, but setting the the dilation parameter (s) equal to 1. After that, we have compared the three columns (we refer to the columns as *dimensions*). Figure 2.3 shows the dimension i of $\mathbf{X}$ against the dimension i of $\mathbf{MDS_{Div}}$, $i \in \{1, 2, 3\}$.

As we can see, the algorithm is able to capture the dimensions of the original matrix. We do not show cross-dimensions (i.e dimension i of $\mathbf{X}$ against dimension j of $\mathbf{MDS_{Div}}$ $i \neq j$), but Table 2.1 contains the cross-correlation matrix. The results show that dimension i of $\mathbf{MDS_{Div}}$ captures dimension i of $\mathbf{X}$ and just dimension i. So, it seems that the algorithm, for this particular case, has a good performance.

	1	2	3
1	1	0.02	-0.04
2	0.02	1	0.02
3	-0.04	0.02	1

Table 2.1: Cross-correlation of $\mathbf{X}$ and $\mathbf{MDS_{Div}}$.



(a) (b) (c)

Figure 2.3: Dimension 1,2 and 3 of $\mathbf{X}$ against dimensions 1,2 and 3 of $\mathbf{MDS_{Div}}$. In red, the line $x = y$.

2.3 Fast MDS

During the process of gathering information about work previously done around MDS with big datasets, we found that Tynia, Jinze, Leonard, and Wei (2006) already proposed an algorithm, they named it *Fast Multidimensional Scaling*.

2.3.1 Algorithm

- Divide $\mathbf{X}$ into $\mathbf{X_1}, \dots, \mathbf{X_p}$.
- Compute MDS for each $\mathbf{X_i}$: $\mathbf{MDS_1}, \dots, \mathbf{MDS_p}$. These individuals MDS solutions are stitched together by sampling s points (rows) from each submatrix $\mathbf{X_i}$ and putting them into an alignment matrix $\mathbf{M_{align}}$ of size $sp \times sp$. In principle, s should be at least 1 plus the estimated dimensionality of the dataset. In practice, they oversample by a factor of 2 or more.
- MDS is run on $\mathbf{M_{align}}$. After this, it is obtained **mMDS**. Given a sampled point, there are two solutions of MDS: one from $\mathbf{X_i}$ and another one from $\mathbf{M_{align}}$.
- The next step is to compute the Procrustes transformation to line these two sets of solutions up in a common coordinate system:

$$\mathbf{mMDS_i} = s_i \mathbf{dMDS_i T_i} + \mathbf{1 t_i}'$$

  where:

  - $\mathbf{dMDS_i}$ is $\mathbf{MDS_i}$ but taking into account just the subset of the sample points that belongs to partition i.
  - $\mathbf{mMDS_i}$ is **mMDS** but taking into account just the subset of the sample points that belongs to partition i

- After doing the previous part, we obtain a set of p Procrustes parameters $(s_i, \mathbf{T_i}, \mathbf{t_i})$. So, the next step is to apply this set of parameters to each $\mathbf{MDS_i}$, i.e,

$$\mathbf{MDS_i}^a := s_i \mathbf{MDS_i T_i} + \mathbf{1 t_i'}.$$

- The last step is to join $\mathbf{MDS_1}^a, \dots, \mathbf{MDS_p}^a$ all together, i.e,

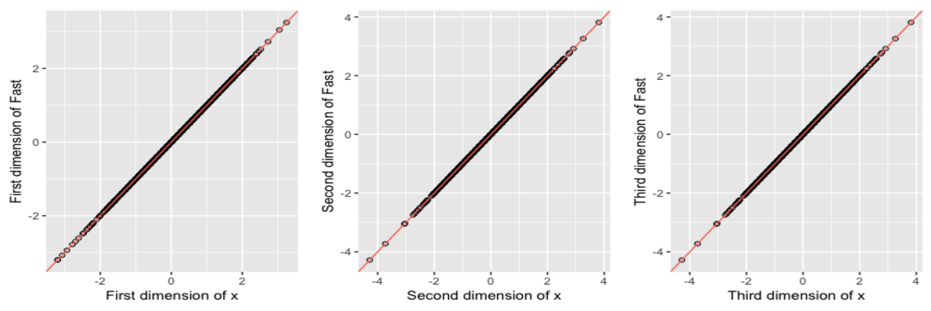
$$\mathbf{MDS_X} := \mathbf{MDS_1}^a \cup \dots \cup \mathbf{MDS_p}^a.$$

They apply this process recursively until the size of $\mathbf{X_i}$ is optimal to run MDS on. They find the stop condition as follows. Let $l \times l$ be the size of the largest matrix that allow MDS to be executed efficiently, i.e, in a reasonable amount of time. There are two issues that impact the performance of the algorithm: the size of each submatrix after subdivision and the number of submatrices, p, that are stitched together at each step. Ideally, the size of each submatrix after division should be as large as possible without exceeding l. By the same token, the size of $\mathbf{M_{align}}$ should be bounded by l. The number of submatrices to be stitched together, p, should be the largest number such that $sp \leq l$.

2.3.2 Some indicators about the performance of the algorithm

As we have done in Section 2.2.2, we present some visual results of this algorithm. The data used are the same as in Section 2.2.2. We call $\mathbf{MDS_{Fast}}$ the result that provides the previous algorithm.

Figure 2.4 shows that, for this particular case, the algorithm captures quite well the dimensions of the original data, providing a good performance. In addition, dimension i of $\mathbf{MDS_{Fast}}$ fits perfectly the same dimensions i of $\mathbf{X}$ and just this one, as we can see in Table 2.2.



(a) (b) (c)

Figure 2.4: Dimensions 1,2 and 3 of $\mathbf{X}$ against dimensions 1,2 and 3 of $\mathbf{MDS_{Fast}}$. In red, the line $x = y$.

	1	2	3
1	1	0.02	0
2	0.02	1	0.02
3	0	0.02	1

Table 2.2: Cross-correlation of $\mathbf{X}$ and $\mathbf{MDS_{Fast}}$.

2.4 MDS based on Gower interpolation

Gower interpolation formula (see Appendix of (Handl 1996)) allows to add a new set of points to a given MDS configuration. Given a matrix $\mathbf{X}$ $n \times p$, a MDS configuration for this matrix of dimension $n \times c$ and a matrix $\mathbf{X_{new}}$ $m \times p$, one wants to add these new m rows to the existing MDS configuration. So, after adding this new rows, the MDS configuration will have $n + m$ rows and c columns. We briefly summarise how to do so:

- Obtain $\mathbf{J} = \mathbf{I_n} - \frac{1}{n}\mathbf{1}\mathbf{1}'$, where $\mathbf{I_n}$ is the identity matrix $n \times n$.
- Given the distance matrix $\mathbf{D}$ of the rows of $\mathbf{X}$, calculate $\mathbf{\Delta} = (\delta_{ij}^2)$.
- Calculate $\mathbf{G} = -\frac{1}{2}\mathbf{J}\mathbf{\Delta}\mathbf{J}'$
- Let $\mathbf{g}$ be the diagonal of $\mathbf{G}$, i.e, $\mathbf{g} = \mathrm{diag}(\mathbf{G})$. We treat $\mathbf{g}$ as a vector.
- Let $\mathbf{A}$ be the distance matrix between the rows of $\mathbf{X}$ and the rows of $\mathbf{X_{new}}$. $\mathbf{A}$ has dimensions $m \times n$. Let $\mathbf{A}^2$ be the matrix of the square elements of $\mathbf{A}$, i.e, $\mathbf{A}^2 = (a_{ij}^2)$.
- Let $\mathbf{M}$ and $\mathbf{S}$ be the MDS for $\mathbf{X}$ and the variance-covariance matrix of the c columns of $\mathbf{M}$.
- The interpolated coordinates for the new m observations are given by

$$\frac{1}{2n}(\mathbf{1}\mathbf{g}' - \mathbf{A}^2)\mathbf{M}\mathbf{S}^{-1}. \tag{2.1}$$

The resulting MDS for the m observations of $\mathbf{X_{new}}$ is in the same coordinate system as $\mathbf{M}$. So, here it is not needed to do any Procrustes transformation.

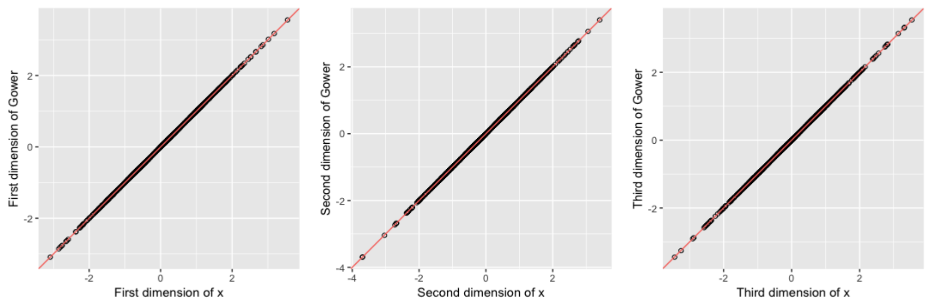
2.4.1 Algorithm

- Divide $\mathbf{X}$ into p partitions $\mathbf{X_1}, \ldots, \mathbf{X_p}$. We use the procedure explained above, being $\mathbf{X_{new}} := \mathbf{X_k}$, $k \in \{1, \ldots, p\}$.
- Calculate $\mathbf{J}$, $\mathbf{\Delta}$, $\mathbf{G}$, $\mathbf{g}$, $\mathbf{A}$, $\mathbf{M}$ and $\mathbf{S}$ according to the above formulas.
- Obtain MDS for the first partition $\mathbf{X_1}$.
- Given a partition $1 < k \leq p$, do the following steps to get the related MDS:
  - Calculate the distance matrix between the rows of $\mathbf{X_1}$ and $\mathbf{X_k}$ and calculate the square of each element of this matrix. Let $\mathbf{A}^2$ be this matrix (same as above).
  - Use Gower interpolation formula (2.1) to obtain MDS for partition k.
  - Accumulate this solution.

As in the previous two algorithms, there is a key parameter to choose: p, the number of partitions. For this algorithm, p is set in the following way. Let $l \times l$ be the size of the largest distance matrix that a computer can calculate efficiently, i.e, in a reasonable amount of time. The value of p is set as n/p.

2.4.2 Some indicators about the performance of the algorithm

We repeat the same as in Section 2.2.2. Figure 2.5 and Table 2.3 shows that the algorithm, for this particular case, captures quite well the dimensions of the original data, providing a good performance.



(a) (b) (c)

Figure 2.5: Dimensions 1,2 and 3 of $\mathbf{X}$ against dimensions 1,2 and 3 of $\mathbf{MDS_{Gower}}$. In red, the line $x = y$.

	1	2	3
1	1	0	-0.04
2	0	1	-0.0
3	-0.04	-0.03	1

Table 2.3: Cross-correlation of $\mathbf{X}$ and $\mathbf{MDS_{Gower}}$.

2.5 Comparison of the algorithms

The three previous algorithms share the same goal: obtaining a MDS configuration for a given big dataset. However, there are some differences between the approaches that impact the performance of the algorithms. The main differences between them are:

- *Divide and Conquer MDS* uses a guide (the first subset, $\mathbf{X_1}$) to align the solutions as well as it uses the whole partition $\mathbf{X_i}$ to find the Procrustes parameters. However, *Fast MDS* does not use a guide an it uses a set of subsamples to find the Procrustes parameters.
- *Fast MDS* is based on recursive programming. It divides until a manageable dimensionality is found. However, *Divide and Conquer MDS* finds the number of partitions without applying recursive programming.
- *MDS based on Gower interpolation* does not need any Procrustes transformation.

The fact that we found three algorithms to compute MDS possesses some questions that need to be answered:

- Are these algorithms able to capture the data dimensionality as good as classical MDS does?
- Which is the fastest method?
- Can they deal with big datasets in a reasonable amount of time?
- How are they performing when dealing with big data sets?

All these questions and are answered in Chapter 3.

2.6 Output of the algorithms

The three algorithms have the same type of output. It consists on a list of two parameters.

The first parameter is the MDS configuration calculated by the algorithm. It is a matrix of n rows and c columns, where n is the number of rows of the input data and c is the number of dimensions the user has required.

The second parameter is a list of eigenvalues. This list is built as follows:

- All the algorithms divide the initial data into a set of p partitions.
- Given a partition i, a distance matrix of dimensions $m_i \times m_i$ is calculated: $\mathbf{D_i}$.
- Over $\mathbf{D_i}$ a singular value decomposition is performed, providing a list of length m_i that contains all the eigenvalues of the previous decomposition: $list_i$.
- Let $norm\_eigenvalues_i$ be $list_i/m_i$, i.e, each eigenvalue is divided by the number of rows of $\mathbf{D_i}$.
- The algorithms return $norm\_eigenvalues_1 \cup \cdots \cup norm\_eigenvalues_p$. We refer to this union as the *normalized eigenvalues.*

Chapter 3

Simulation study

3.1 Design of the simulation

Given the three algorithms, we would like to ~~know how they perform~~. There are two issues to study:

- Performance ~~of the algorithms~~: are they able to captu[illegible]a dimensionality?
- Performance in terms of time: are they " fast" enough? Which one is the fastest?

To test the algorithms under different conditions, a simulation study has been carried out. The scenarios are obtained as combinations of:

- *Sample size*: we use different sample sizes, combining small data sets and big ones. A total of six sample sizes are used, which are: $10^3, 3 \cdot 10^3, 5 \cdot 10^3, 10^4, 10^5, 10^6$.
- *Data dimensions*: we generate a matrix with two different number of columns: 10, 100.
- *Main dimensions*: given the data matrix $\mathbf{X}$ $n \times k$[1], it is postmultiplied by a diagonal matrix that contains k values, $\lambda_1, \dots, \lambda_k$. The first values are much higher than the rest. The idea of this is to see if the algorithms are able to capture the main dimensions of the original dataset, i.e, the columns with the highest variance. We set 5 combinations for this variable, which are:
  - All the columns with the same values of λ: $\lambda_1 = \cdots = \lambda_k = 1$.
  - One main dimension with $\lambda_1 = 15$ and $\lambda_2 = \cdots = \lambda_k = 1$.
  - Two main dimensions of the same value λ: $\lambda_1 = \lambda_2 = 15$, $\lambda_3 = \cdots \lambda_k = 1$.
  - Two main dimensions of different values λ: $\lambda_1 = 15$, $\lambda_2 = 10$, $\lambda_3 = \cdots \lambda_k = 1$.
  - Four main dimensions of the same value λ: $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = 15$, $\lambda_5 = \cdots \lambda_k = 1$.
- As probabilistic model, we use a Normal distribution with $\mu = 0$ and $\sigma = 1$. With this distribution, we generate a matrix of n observations and k columns, being the k columns independent .After generating the dataset $\mathbf{X}$, it is postmultiplied by the diagonal matrix that contains the values of λ's.

[1] $n \in \{10^3, 3 \cdot 10^3, 5 \cdot 10^3, 10^4, 10^5, 10^6\}$ and $k \in \{10, 100\}$

There is a total of 60 scenarios to simulate. Given a scenario, it is replicated 100 times. For every simulation, it is generated a dataset (according to the scenario), and all the algorithms are run using this dataset. So, a total of 6000 simulations are carried out.

Table 3.1 shows the configuration of each scenario. Given a scenario, *scenario_id* identifies it. We refer to a scenario by its *scenario_id*.

	scenario_id	sample_size	n_dimensions	value_primary_dimensions
1	1	10^3	10	NULL
2	2	10^3	100	NULL
3	3	10^3	10	15
4	4	10^3	100	15
5	5	10^3	10	c(15, 15)
6	6	10^3	100	c(15, 15)
7	7	10^3	10	c(15, 10)
8	8	10^3	100	c(15, 10)
9	9	10^3	10	c(15, 15, 15, 15)
10	10	10^3	100	c(15, 15, 15, 15)
11	2000	$3 \cdot 10^3$	10	NULL
12	2001	$3 \cdot 10^3$	100	NULL
13	2002	$3 \cdot 10^3$	10	15
14	2003	$3 \cdot 10^3$	100	15
15	2004	$3 \cdot 10^3$	10	c(15, 15)
16	2005	$3 \cdot 10^3$	100	c(15, 15)
17	2006	$3 \cdot 10^3$	10	c(15, 10)
18	2007	$3 \cdot 10^3$	100	c(15, 10)
19	2008	$3 \cdot 10^3$	10	c(15, 15, 15, 15)
20	2009	$3 \cdot 10^3$	100	c(15, 15, 15, 15)
21	4000	$5 \cdot 10^3$	10	NULL
22	4001	$5 \cdot 10^3$	100	NULL
23	4002	$5 \cdot 10^3$	10	15
24	4003	$5 \cdot 10^3$	100	15
25	4004	$5 \cdot 10^3$	10	c(15, 15)
26	4005	$5 \cdot 10^3$	100	c(15, 15)
27	4006	$5 \cdot 10^3$	10	c(15, 10)
28	4007	$5 \cdot 10^3$	100	c(15, 10)
29	4008	$5 \cdot 10^3$	10	c(15, 15, 15, 15)
30	4009	$5 \cdot 10^3$	100	c(15, 15, 15, 15)
31	6000	10^4	10	NULL
32	6001	10^4	100	NULL
33	6002	10^4	10	15
34	6003	10^4	100	15
35	6004	10^4	10	c(15, 15)
36	6005	10^4	100	c(15, 15)
37	6006	10^4	10	c(15, 10)
38	6007	10^4	100	c(15, 10)
39	6008	10^4	10	c(15, 15, 15, 15)

40	6009	10^4	100	c(15, 15, 15, 15)
41	20000	10^5	10	NULL
42	20001	10^5	100	NULL
43	20002	10^5	10	15
44	20003	10^5	100	15
45	20004	10^5	10	c(15, 15)
46	20005	10^5	100	c(15, 15)
47	20006	10^5	10	c(15, 10)
48	20007	10^5	100	c(15, 10)
49	20008	10^5	10	c(15, 15, 15, 15)
50	20009	10^5	100	c(15, 15, 15, 15)
51	30000	10^6	10	NULL
52	30001	10^6	100	NULL
53	30002	10^6	10	15
54	30003	10^6	100	15
55	30004	10^6	10	c(15, 15)
56	30005	10^6	100	c(15, 15)
57	30006	10^6	10	c(15, 10)
58	30007	10^6	100	c(15, 10)
59	30008	10^6	10	c(15, 15, 15, 15)
60	30009	10^6	100	c(15, 15, 15, 15)

Table 3.1: Scenarios simulated

Note that scenarios 1, 2, 2000, 2001, 4000, 4001, 6000, 6001, 20000, 20001, 30000, 30001 are pure noise. We refer to them as *noisy scenarios*.

In order to test the performance of the algorithms as well as the time needed to compute the MDS configuration, some metrics are calculated. These metrics are the following ones:

- Performance: in terms of performance two metrics are calculated, which are:
  - Correlation between the main dimensions of the data and the main dimensions after applying the algorithms. We get the diagonal of the correlation matrix, i.e, the correlation between dimension i of the data and the dimension i of the algorithm.
  - *Normalized eigenvalues* as an approximation of the standard deviation of the variables of $\mathbf{X}$.
- Elapsed Time to get the MDS configuration: Given an algorithm, we compute and store the elapsed time to get the corresponding MDS configuration.

We do it in this way because we want to check some hypothesis. We expect the three algorithms to behave "correctly". By "correctly" we mean that the behavior should be the same as if classical MDS were run. Therefore, we expect that the correlation between the main dimensions of the data and the main dimensions of the MDS of each algorithm is close to 1 when the dimension asked to MDS is k, i.e, the same dimension as the original dataset.

In addition, the variance of the original data should be captured. So, given the highest *normalized eigenvalues*, we expect that its square root is approximately 15 or 10 when the scenarios are not the *noisy scenarios*.

For the time of the algorithms, we have done some tests and it seems that *MDS based on Gower interpolation* seems to be the fastest. So, it will be tested.

Given a scenario, the steps that we have performed to calculate and to store all the data needed are:

1. Generate the data according to the scenario.

2. For each algorithm, we do the following steps:

   (a) Run the algorithm and get MDS configuration for the algorithm ($\mathbf{MDS_{alg}}$).
   (b) Get the elapsed time to compute MDS configuration and store it.
   (c) Get *normalized eigenvalues* and store them.
   (d) Align $\mathbf{MDS_{alg}}$ and $\mathbf{X}$ using Procrustes.
   (e) Get the correlation coefficients between the main dimensions of $\mathbf{MDS_{alg}}$ and $\mathbf{X}$ and store it.

There are some important details that affect the results of the simulations, which are:

- When running the algorithms, we ask for as many columns as the original data has, i.e, k. Therefore, the low-dimensional space has the same dimension as the original dataset.

- For the *normalized eigenvalues*, we just store 6 eigenvalues instead of the full list of eigenvalues (otherwise we would store n eigenvalues, which is memory consuming).

- For Procrustes we dot not allow dilations, otherwise distance could not be preserved. In addition, we do not use all the columns to do the alignment, we select the main dimensions. If there is not any main dimension, i.e it is one of the *noisy scenarios*, we just select 4 columns.

- To avoid memory problems with the alignment when n is greater or equal to 10^5, Procrustes is done in the following way:

  1. Create p partitions of $\mathbf{X}$ and the result of a given MDS algorithm ($\mathbf{MDS_{alg}}$). Both sets of partitions contain exactly the same observations.
  2. For each partition get the Procrustes parameters without dilations.
  3. Accumulate the parameters iteration after iteration. So, at the end, we obtain $\mathbf{R} = \sum_{i=1}^{p} \mathbf{R_i}$ and $\mathbf{t} = \sum_{i=1}^{p} \mathbf{t_i}$.
  4. $\mathbf{R} = \mathbf{R}/p$ and $\mathbf{t} = \mathbf{t}/p$.
  5. Apply these parameters to $\mathbf{MDS_{alg}}$ so that $\mathbf{X}$ and $\mathbf{MDS_{alg}}$ are in the same coordinate system and they can be compared, i.e

$$\mathbf{X_{Procrustes}} = \mathbf{XR} + \mathbf{1t}'.$$

The algorithms have as input values a set of variables. The input matrix is already explained, but there is another parameter that has been used in the description of the algorithms (see Chapter 2): l. The meaning of l is a little bit different in each algorithm, but for simplicity we set this value equals to 500.

Fast MDS has an extra parameter: the amplification parameter. It is used the value that they used to test this algorithm, i.e, a value of 3. So, for each partition, it is taken 30 (when the original matrix has 10 columns) or 300 (when the original matrix has 100 columns) points for every partition to build $\mathbf{M_{align}}$.

Finally, there is an extra ~~"~~parameter~~"~~ to take into account: the machine used to do the simulations. Since a total of 6000 simulations are performed and some of them include big datasets, we use *Amazon Web Services* (AWS) to carry out the simulations. 10 servers of the same type are used: *c5n.4xlarge*. It has 16 cores and 42 GB of memory RAM. We use this server because it is designed for applications like batch and log processing, distributed and or real-time analytics.

3.2 Correlation coefficients

In this section we provide ~~some~~ results for the correlation coeffi~~cients based on the simulations~~. Given a scenario and its dataset $\mathbf{X}$ $n \times k$, the correlation matrix between the main dimensions of $\mathbf{X}$ and the main dimensions of $\mathbf{MDS_{alg}}$ is computed. We are interested in the diagonal of the correlation matrix, expecting that the values are close to 1. We expect 1 as a correlation coefficient because the number of dimensions required to the MDS configuration is the same as the number of dimensions of the original dataset.

The length of the diagonal correlation matrix is not static. Depending on the scenario it can be:

- 1, when $\lambda_1 = 15$ and $\lambda_2 = \cdots = \lambda_k = 1$.
- 2, when$\lambda_1 = \lambda_2 = 15$, $\lambda_3 = \cdots \lambda_k = 1$ or $\lambda_1 = 15$, $\lambda_2 = 10$, $\lambda_3 = \cdots \lambda_k = 1$.
- 4, when $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = 15$, $\lambda_5 = \cdots \lambda_k = 1$.
- 0, when $\lambda_1 = \cdots = \lambda_k = 1$, i.e, *noisy scenarios*.

Figure 3.1 shows some boxplots of the correlation coefficients between the main dimensions of the data $\mathbf{X}$ and the main dimensions of $\mathbf{MDS_{Div}}$, where $\mathbf{MDS_{Div}}$ is the MDS configuration for *Divide and Conquer MDS*. As we can see, all the values are close to 1, showing a good performance.

Note that before calculating the correlation matrix, Procrustes transformation is performed (dilations are not allowed) so that both coordinate systems are the same.

Figure 3.1 just shows the correlation coefficients for $n = 10^3$. The remaining figures are in Appendix A.1. All of them have the same shape, independently of the scenario: there is always a high correlation between the data and the MDS. ~~Actually, the correlation is so high that it is not needed any hypothesis test to check if the value is 1 or not.~~

Figure 3.2 shows the boxplot for the *noisy scenarios*. It has been taken 4 dimensions to do the alignment, instead of all the dimensions (10 or 100). Because of this, we expect a random distribution between them within [0,1] interval. If we had taken all the dimensions, we would be seeing higher correlation values.

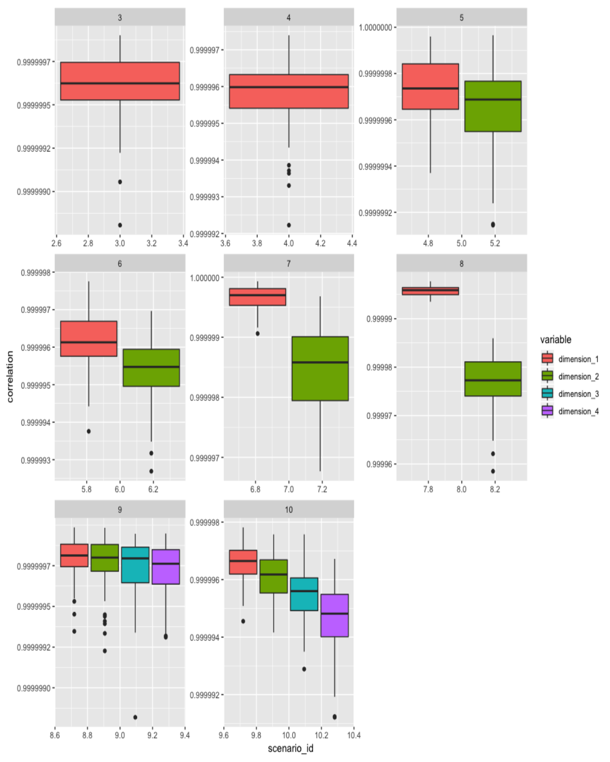


Figure 3.1: Correlation for $n = 10^3$ and MDS Divide and conquer algorithm

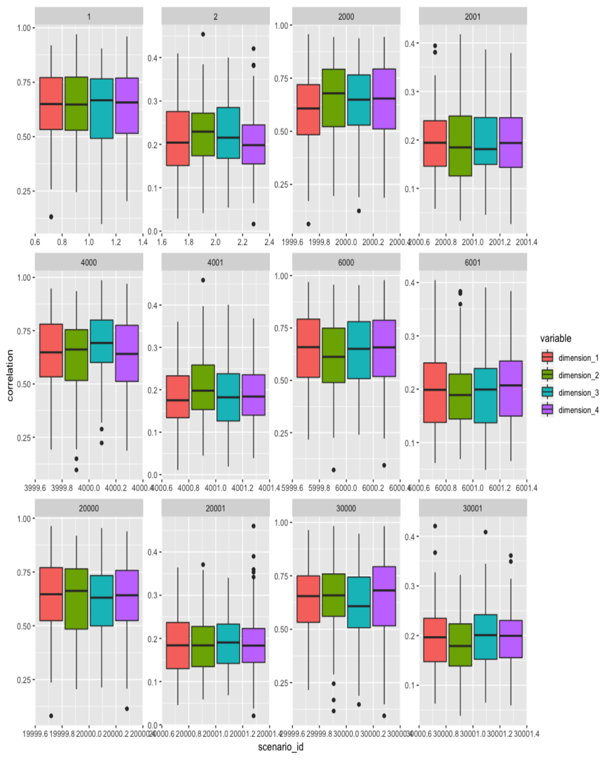


Figure 3.2: Correlation for *noisy scenarios* and MDS Divide and conquer algorithm

We do the same for *Fast MDS* algorithm. Figure 3.3 shows the boxplot of the correlation coefficients between the data and the MDS of this algorithm. Again, there is high correlation between them. Figure 3.4 shows the boxplot for the *noisy scenarios*. The remaining plots are in Appendix A.2.

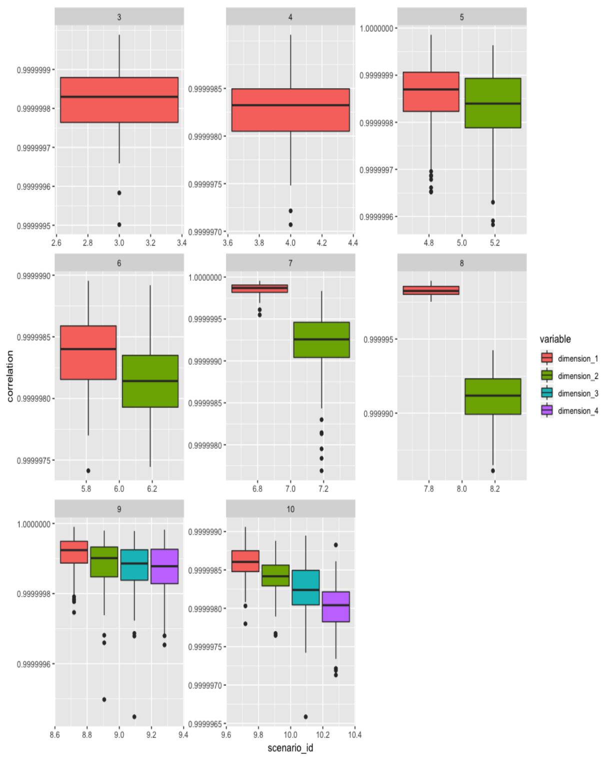


Figure 3.3: Correlation for $n = 10^3$ and MDS Fast algorithm.

We do the same for *MDS based on Gower interpolation* algorithm. Figure 3.5 shows the boxplot of the correlation coefficients between the data and the MDS of this algorithm. Again, there is high correlation between them. Figure 3.6 shows the boxplot for the *noisy scenarios*. The remaining plots are in Appendix A.3.

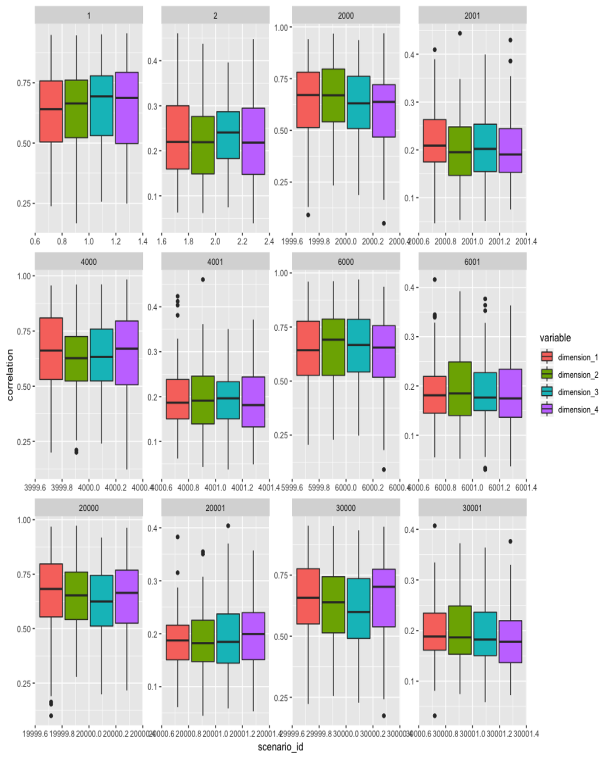


Figure 3.4: Correlation for *noisy scenarios* and MDS Fast algorithm.

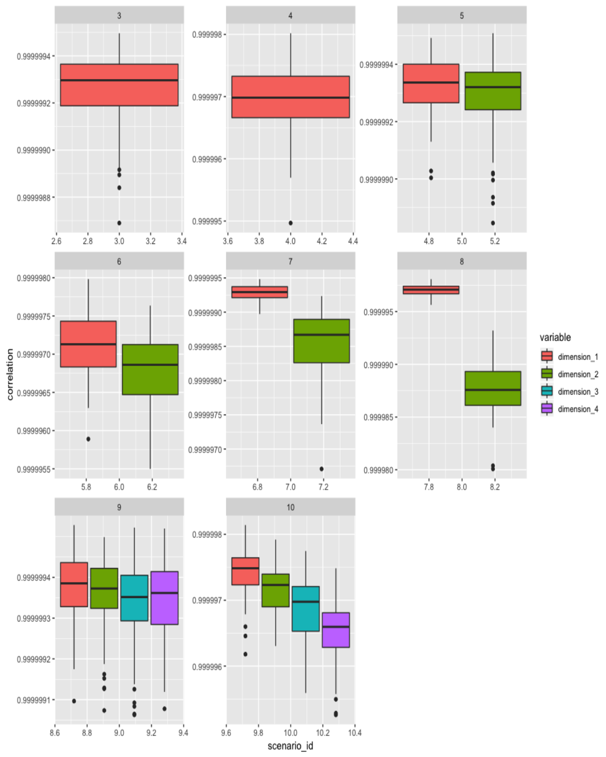


Figure 3.5: Correlation for $n = 10^3$ and MDS based on Gower algorithm.

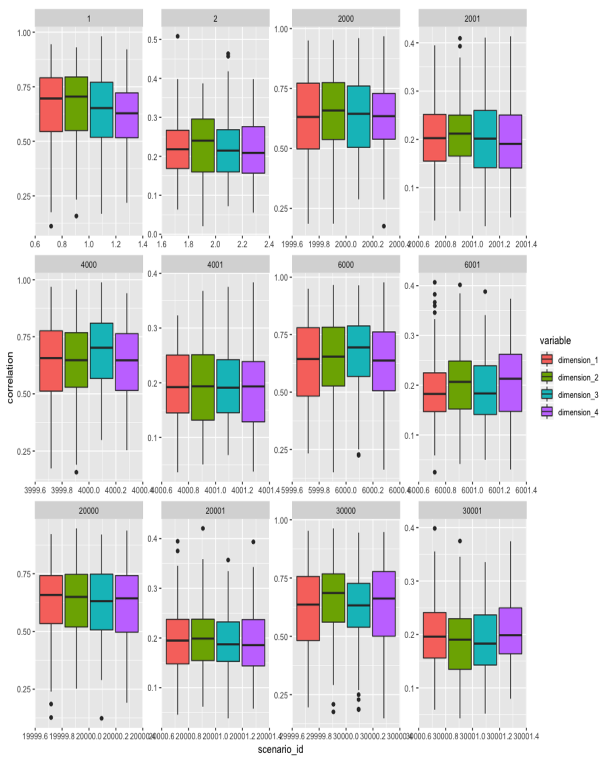


Figure 3.6: Correlation for *noisy scenarios* and MDS based on Gower algorithm.

3.3 Eigenvalues

In this section we analyse how the eigenvalues approximate the standard deviation of the original variables. Given $\mathbf{X}$ $n \times k$, we say that the original data has k variables. So, we treat the columns as variables.

Since the original dataset, $\mathbf{X}$, is postmultiplied by a diagonal matrix $k \times k$ that contains $\lambda_1, \dots, \lambda_k$, then $\mathrm{var}(X_i) = \lambda_i^2$ and $\mathrm{sd}(X_i) = \lambda_i$.

MDS should be able to capture the variance of the main dimensions through the eigenvalues. Let $\phi_1, \dots, \phi_t$ be the *normalized eigenvalues* of the MDS such that $\phi_1 > \phi_2 > \cdots > \phi_t$. The first highest *normalized eigenvalues* have to verify $\sqrt{\phi_j} \approx \lambda_j$.

Given a scenario, the number of main dimensions can be:

- 1 if there is one main dimension with $\lambda_1 = 15$.
- 2 if there are two main dimensions with $\lambda_1 = \lambda_2 = 15$ or with $\lambda_1 = 15, \lambda_2 = 10$.
- 4 if there are four main dimensions with $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = 15$.
- 0 if it is a *noisy scenario*, i.e, $\lambda_1 = \lambda_k = \cdots = \lambda_k = 1$.

To check how the algorithms approximate the variance of the original data, we compute the bias and mean square error (MSE) for each scenario. We do not include the noisy ones. Remember that~~,~~ bias $= \frac{1}{m}\sum_{i=1}^{m}\sqrt{\phi_{ij}} - \lambda_j = \overline{\sqrt{\phi_j}} - \lambda_j$ and MSE $= \frac{1}{m}\sum_{i=1}^{m}(\lambda_j - \sqrt{\phi_{ij}})^2$. Since we have performed 100 simulations, $m = 100$. Depending on the scenario, there can be 1, 2 or 4 estimators.

Table 3.2 shows the bias and MSE for *Divide and Conquer MDS* and scenarios with one main dimension $\lambda = 15$. As we can see, the bias and MSE ~~is~~ "low" for these scenarios and this algorithm.

The remaining cases are in Appendix B.1. As long as number of dimensions increases, the bias and MSE does the same. However, it seems to be in an acceptable range.

	scenario_id	$\sqrt{\phi_1}$	bias_1	MSE_1
1	3	14.98	-0.02	0.03
2	4	15.03	0.03	0.11
3	2002	15.00	-0.00	0.00
4	2003	14.96	-0.04	0.16
5	4002	14.99	-0.01	0.02
6	4003	14.99	-0.01	0.01
7	6002	14.99	-0.01	0.01
8	6003	14.99	-0.01	0.00
9	20002	14.99	-0.01	0.01
10	20003	14.99	-0.01	0.01
11	30002	14.98	-0.02	0.03
12	30003	14.99	-0.01	0.01

Table 3.2: MSE for scenarios with one main dimension $\lambda_1 = 15$ for *Divide and Conquer MDS*.

Table 3.3 shows the bias and MSE for *Fast MDS* and scenarios with one main dimension $\lambda = 15$. There is one scenario (*scenario_id = 6002*) for which the MSE seems to be high. We do not know the reason, since it is one particular case. Apart from this scenario, the other ones have a small value for the bias and MSE, but a little bit higher than *Divide and Conquer MDS*.

The remaining cases are in Appendix B.2. As long as number of dimensions increases, the bias and MSE does the same. However, it seems to be in an acceptable range.

Table 3.4 shows the bias and the MSE for *MDS based on Gower interpolation* and scenarios with one main dimension $\lambda = 15$. Again both values are small, but a little bit higher than *Divide and Conquer MDS*.

The remaining cases are in Appendix B.3. As long as number of dimensions increases, the bias and MSE does the same. However, it seems to be in an acceptable range.

The algorithm that has the lowest error is the *Divide and Conquer MDS*. Even though the other ones have higher errors, we consider that they are acceptable.

Note that, we do not consider the *noisy scenarios*, since all the directions have the same variance.

3.4 Time to compute MDS

In this section we ~~test~~ if there exists an algorithm that is faster than the other ones. Our intuition is that *MDS based on Gower interpolation* is the fastest algorithm. Our intuition is based ~~in~~ the fact the it has the lowest mean value, as we can see in Table 3.5.

Table 3.5 provides a rank between the methods: it seems that *MDS based on Gower interpolation* is the fastest one. In second position it would be *Fast MDS* and finally *Divide and Conquer*.

	scenario_id	$\sqrt{\phi_1}$	bias_1	MSE_1
1	3	14.85	-0.15	2.27
2	4	15.01	0.01	0.01
3	2002	14.91	-0.09	0.76
4	2003	15.10	0.10	0.93
5	4002	14.96	-0.04	0.14
6	4003	15.03	0.03	0.07
7	6002	14.33	-0.67	44.82
8	6003	15.09	0.09	0.76
9	20002	15.00	-0.00	0.00
10	20003	15.00	0.00	0.00
11	30002	14.86	-0.14	1.88
12	30003	14.90	-0.10	1.02

Table 3.3: MSE for scenarios with one main dimension $\lambda_1 = 15$ for *Fast MDS*.

	scenario_id	$\sqrt{\phi_1}$	bias_1	MSE_1
1	3	15.05	0.05	0.22
2	4	15.02	0.02	0.04
3	2002	14.94	-0.06	0.36
4	2003	15.04	0.04	0.20
5	4002	14.98	-0.02	0.04
6	4003	15.02	0.02	0.05
7	6002	14.99	-0.01	0.01
8	6003	15.06	0.06	0.31
9	20002	15.04	0.04	0.19
10	20003	14.97	-0.03	0.07
11	30002	14.98	-0.02	0.06
12	30003	14.90	-0.10	1.07

Table 3.4: MSE for scenarios with one main dimension $\lambda_1 = 15$ for *MDS based on Gower interpolation*.

~~We do an hypothesis test to check it.~~ We do an ANOVA test using ~~two~~ factors: the sample size (which has 6 levels) and the number of dimensions (which has 2 levels). Instead of using the *elapsed time* variable, we use its logarithm.

Given the results of the ANOVA test, which are in Table 3.6, we can reject the null hypothesis. ~~So, there exists τ_{ij} such that $\tau_{ij} \neq 0$.~~

We fit a linear regression with these variables and see the magnitude of the coefficients. In addition, we plot the distribution of log(elapsed_time) for all the algorithms.

Table 3.7 contains the value of the coefficients. As long as either the sample size or the data dimensions increase the coefficients do the same (and so the time needed). Looking at the values for the *algorithm* variable, it seems that *MDS base on Gower interpolation* is the fastest, being the coefficient equal to 0. On the other hand, *Divide and Conquer* is the slowest one.

Figure 3.7 shows the estimated density of the elapsed time for each algorithm and each *scenario_id*. As we can see, *MDS based on Gower interpolation* is the fastest algorithm, especially when 100 dimensions are required. Figure 3.7 contains the *scenario_id*'s just

	sample_size	n_dimensions	mean_divide_conquer	mean_fast	mean_gower
1	10^3	10	0.27	0.14	0.10
2	10^3	100	0.78	0.69	0.28
3	$3 \cdot 10^3$	10	0.78	0.32	0.16
4	$3 \cdot 10^3$	100	2.50	3.14	0.52
5	$5 \cdot 10^3$	10	1.37	0.54	0.20
6	$5 \cdot 10^3$	100	4.25	5.69	0.84
7	10^4	10	2.60	1.81	0.31
8	10^4	100	8.85	11.79	1.37
9	10^5	10	28.10	11.46	2.44
10	10^5	100	106.30	116.46	18.02
11	10^6	10	420.29	106.59	53.15
12	10^6	100	2365.46	1070.19	813.15

Table 3.5: Mean of elapsed time (in seconds) to compute each algorithm.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
algorithm	2	9283.73	4641.86	32143.99	$< 2e-16$
sample_size	5	108572.93	21714.59	150369.26	$< 2e-16$
n_dimensions	1	12868.36	12868.36	89110.86	$< 2e-16$
Residuals	17991	2598.05	0.14		

Table 3.6: Results for ANOVA test for differences in log(elapsed_time) using algorithms, sample size and num. dimensions as factors.

for the sample size $n = 10^3$. The remaining figures are in Appendix C. One interesting thing that we can observe from Appendix C is the fact that the elapsed time grows as long as the sample size does, especially form *Divide and Conquer MDS*. However, *MDS based on Gower interpolation* and *Fast MDS* provide a really good time even though the sample size is big, being *MDS based on Gower interpolation* the fastest one. So, we can consider both algorithms efficient, since they are able to compute MDS is a reasonable amount of time.

After seeing the estimated density, we would like to test if there exists any difference between the scenarios To test that, we perform, again, an ANOVA test. This test includes two factors: *scenario_id*, which has 60 levels, and algorithms, which has 3 levels. As a response, we use the logarithm of the elapsed time.

Table 3.8 contains the result for the hypothesis test. We can reject the null hypothesis, at least one level is different from the other ones in terms of elapsed time.

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-1.4058	0.0085	-165.44	$< 2e-16$
algorithmfast	-0.4313	0.0069	-62.17	$< 2e-16$
algorithmgower	-1.6926	0.0069	-243.96	$< 2e-16$
sample_size3000	0.9473	0.0098	96.54	$< 2e-16$
sample_size5000	1.4434	0.0098	147.10	$< 2e-16$
sample_size10000	2.1505	0.0098	219.17	$< 2e-16$
sample_size1e+05	4.4286	0.0098	451.35	$< 2e-16$
sample_size1e+06	7.2782	0.0098	741.78	$< 2e-16$
n_dimensions100	1.6910	0.0057	298.51	$< 2e-16$

Table 3.7: Linear model for response log(elapsed_time).

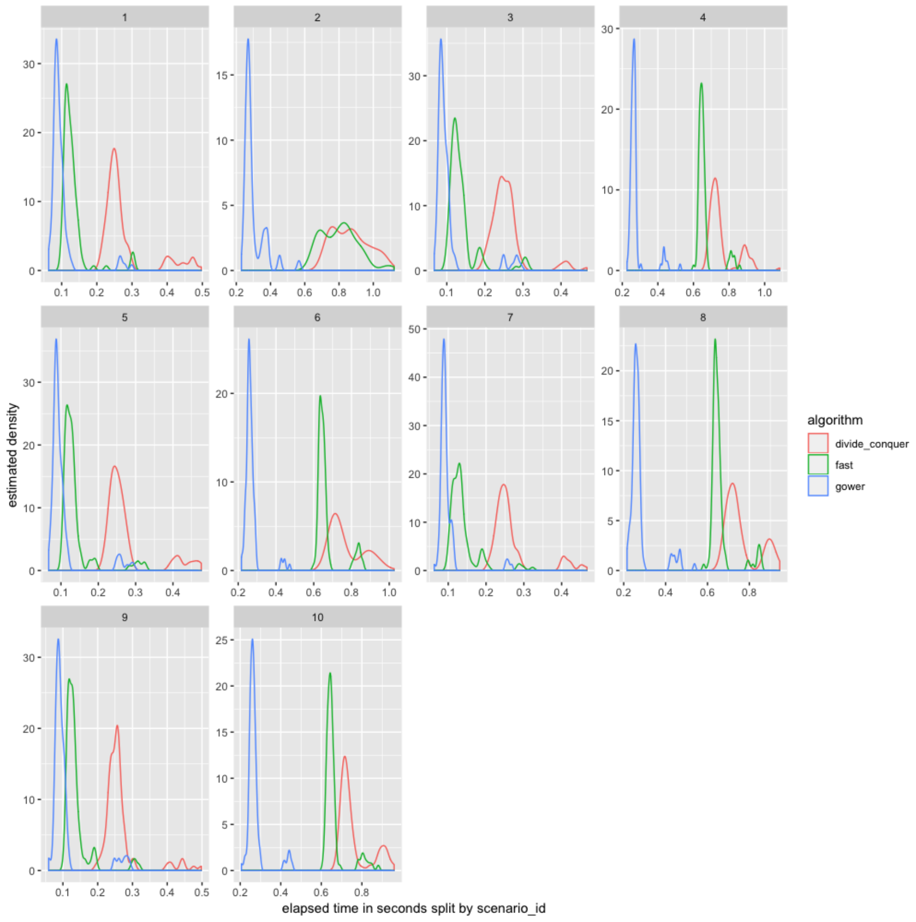


Figure 3.7: Elapsed time for each algorithm and each *scenario_id* of $n = 10^3$.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
scenario_id	59	121912.97	2066.32	17431.46	$< 2e-16$
algorithm	2	9283.73	4641.86	39158.68	$< 2e-16$
Residuals	17938	2126.37	0.12		

Table 3.8: Results for ANOVA test for differences in log(elapsed_time) using *scenario_id* and algorithms as factors.

Chapter 4

Conclusions

The goal of this master's thesis was to find an algorithm able to compute a MDS configuration when dealing with a big dataset in an "efficient" way, i.e, such an algorithm should be fast enough to get the configuration.

The first algorithm developed, *Divide and Conquer MDS*, does not achieve the goal. Although we have managed to get a MDS configuration, too much time is required. However, it has a really good property that the other two algorithms do not have: it is able to capture the variance of the original data quite well.

Even though *Fast MDS* solves the problem of time, it is based on recursive programming. The problem with these kind of algorithms is that they can consume a lot of memory.

A really good algorithm is *MDS based on Gower interpolation*, since it provides a MDS configuration in small amount of time with low errors and its implementation is easy. Apart from this, it does not need to do any Procrustes transformation, which save time and memory. Therefore, this would be the algorithm to use if MDS had to be computed with large datasets.

Problem encountered during the development of the thesis

Basically we have faced two kind of problems, which are:

- Computational problems: when working with big datasets, we suffered from consuming all RAM of the servers. Especially when doing Procrustes for aligning the original data and the MDS for n large. The solution to it was to partition the process into pieces that the servers could manage without consuming all RAM.

- Procrustes packages: even though R has a lot of packages that allows to compute Procrustes, they did not fulfilled our goals. The reasons are either because the output is not well specified or because some of the transformations (mainly dilations or translations) were not included. We recommend to use MCMCpack package.

Future research

The algorithms that we have developed are implemented in R, which is a good language to do prototypes. However, this is not the best programming language to be used. So, we recommend to implement them in a robust programming language such as C, C++ or Java.

Given that the world is talking about Big Data, it is a good opportunity to challenge the algorithms and use them with Spark/Hadoop. Actually the initial idea of this thesis was to use them with a Spark DB that contains millions of chess games. Due to lack of time, we could not do so.

Acknowledgements

I would like to express my gratitude to *Pedro Delicado*, since he helped me to do this thesis. There have been some difficult issues in which his help allowed me to move on and solve them. Also, I would like to thank *Roger Devesa*, since he has helped me with reviewing recursive programming and with implementing *Fast MDS*.

Bibliography

Borg, I. and P. Groenen (2005). *Modern Multidimensional Scaling: Theory and Applications.* Springer.

Gower, J. C. and D. J. Hand (1996). *Biplots* (1st ed ed.). London ; Melbourne : Chapman and Hall. Includes index and bibliographical references.

Handl, A. (1996, October). Biplots : J. C. Gower and D. J. Hand (1996) London: Chapman & Hall, ISBN 0-412-71630-5, [pound sign] 32.00, pp. 277. *Computational Statistics & Data Analysis 22*(6), 655–651.

Peña, D. (2002). *Análisis de datos multivariantes.* Madrid, Spain: McGraw Hill.

Tynia, Y., L. Jinze, M. Leonard, and W. Wei (2006). A fast approximation to multidimensional scaling.

Appendix A

Correlation coefficients boxplot

A.1 Divide and Conquer MDS

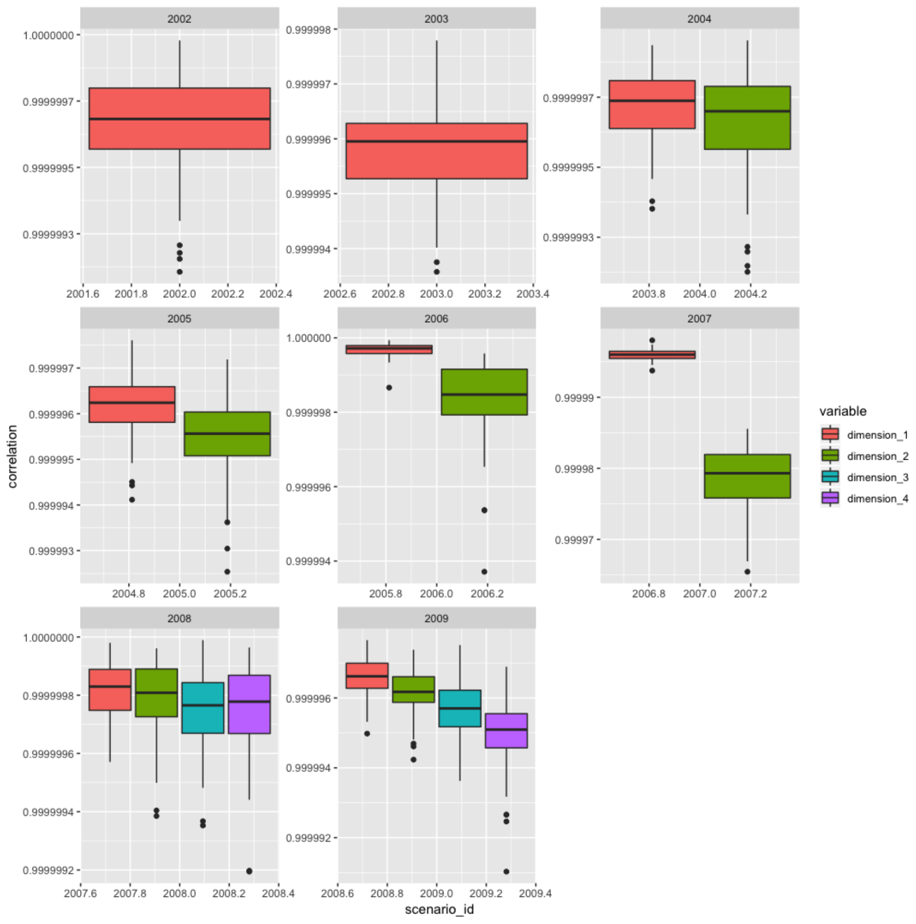


Figure A.1: Correlation for $n = 3 \cdot 10^3$ and MDS Divide and conquer algorithm

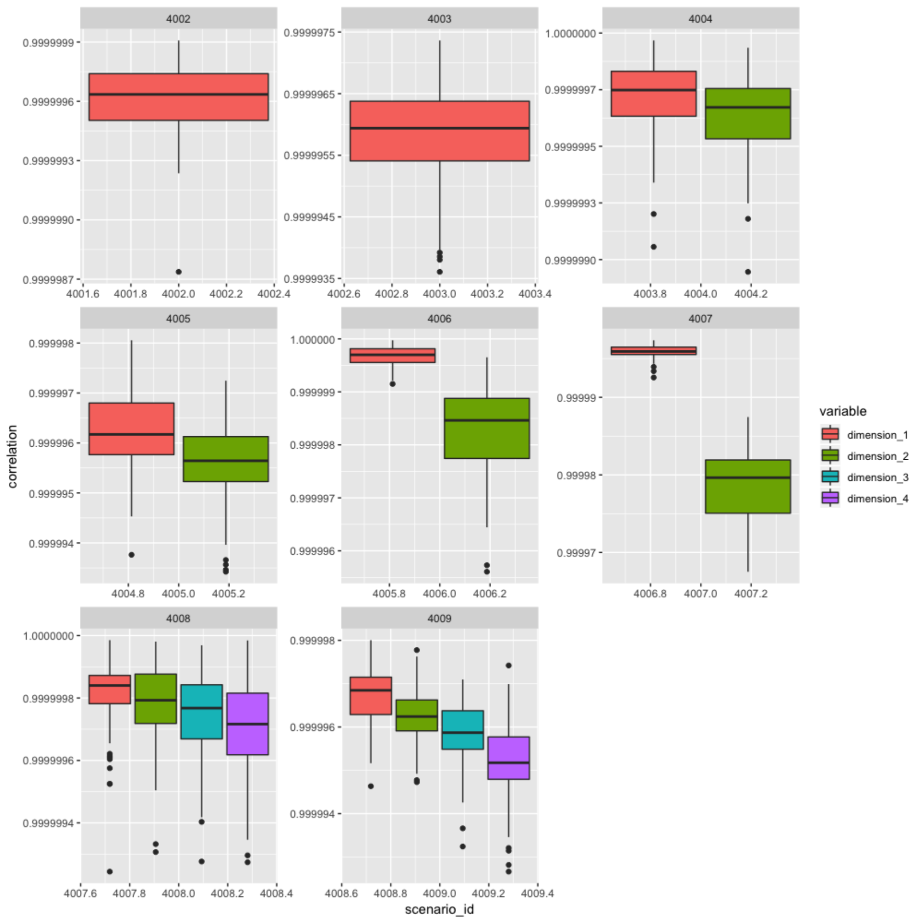


Figure A.2: Correlation for $n = 5 \cdot 10^3$ and MDS Divide and conquer algorithm

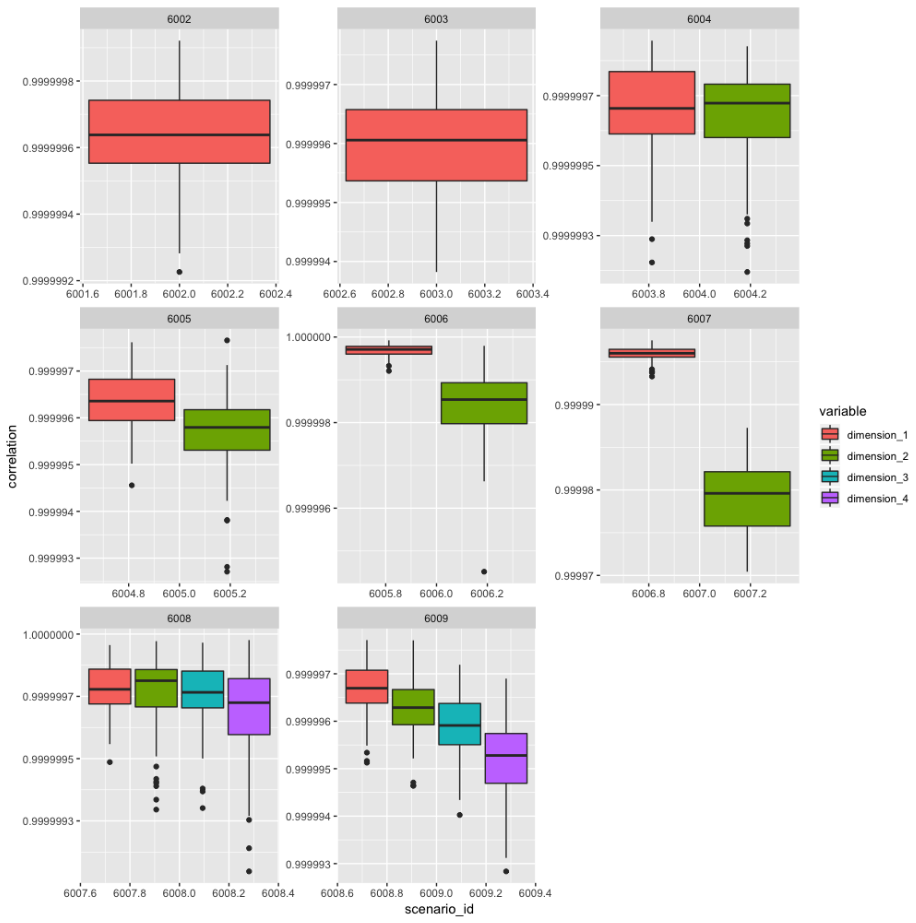


Figure A.3: Correlation for $n = 10^4$ and MDS Divide and conquer algorithm

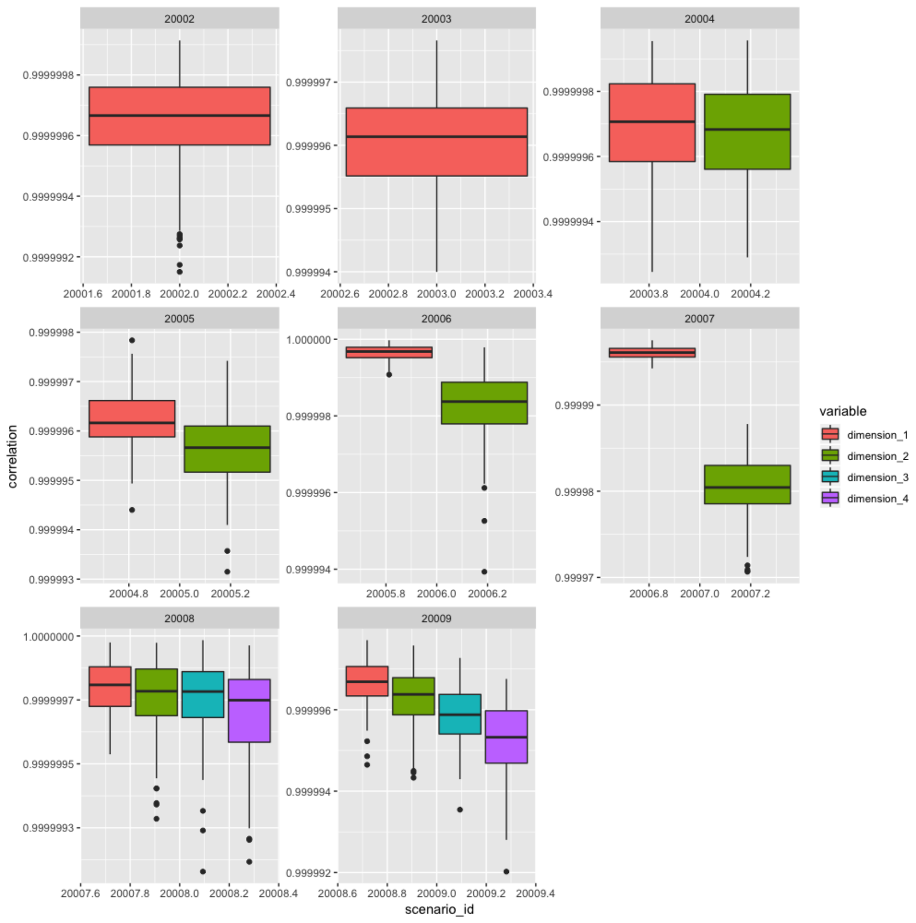


Figure A.4: Correlation for $n = 10^5$ and MDS Divide and conquer algorithm

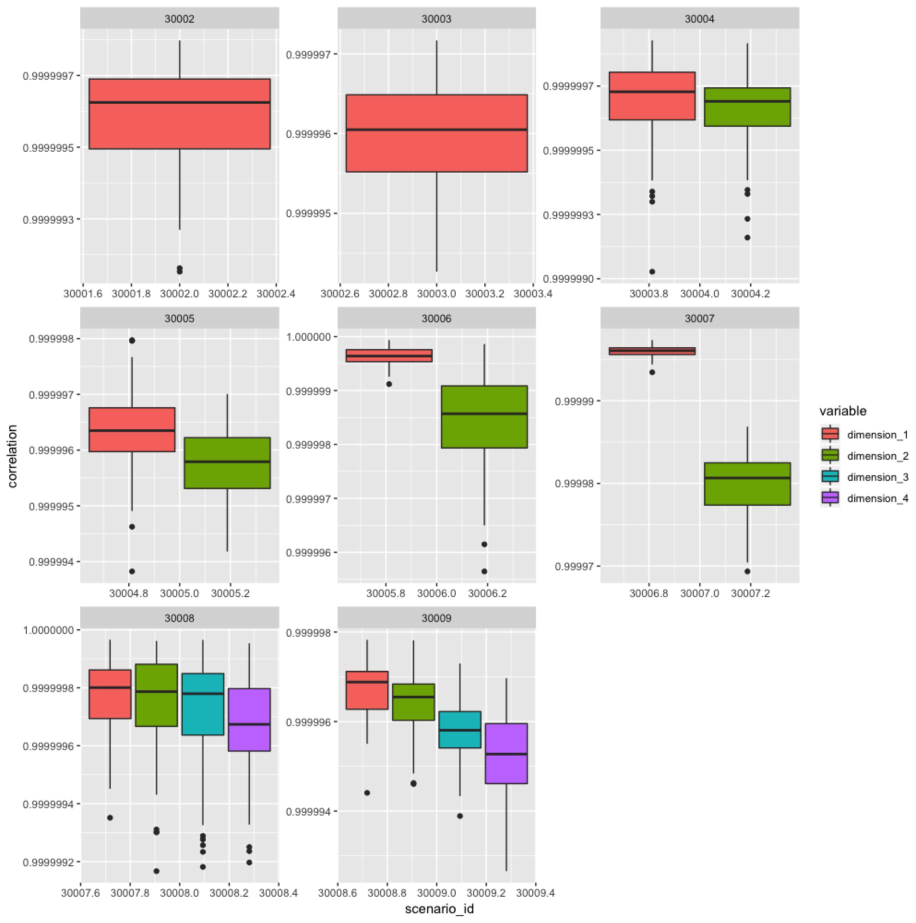


Figure A.5: Correlation for $n = 10^6$ and MDS Divide and conquer algorithm

A.2 Fast MDS

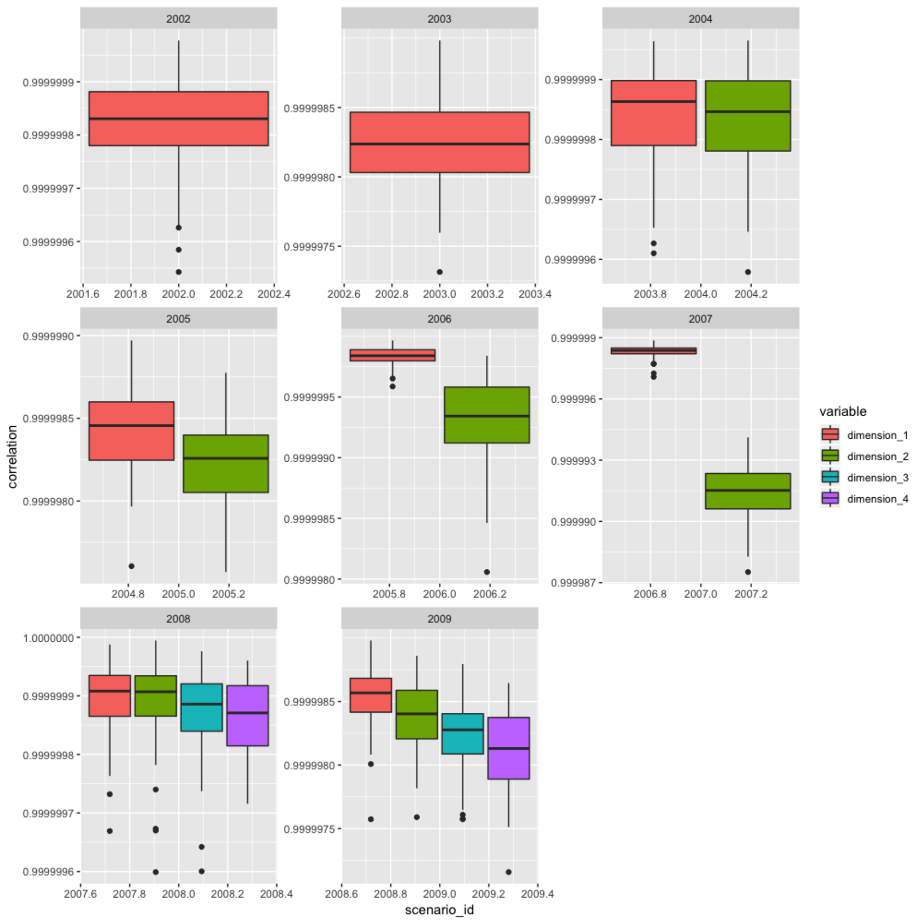


Figure A.6: Correlation for $n = 3 \cdot 10^3$ and MDS Fast algorithm.

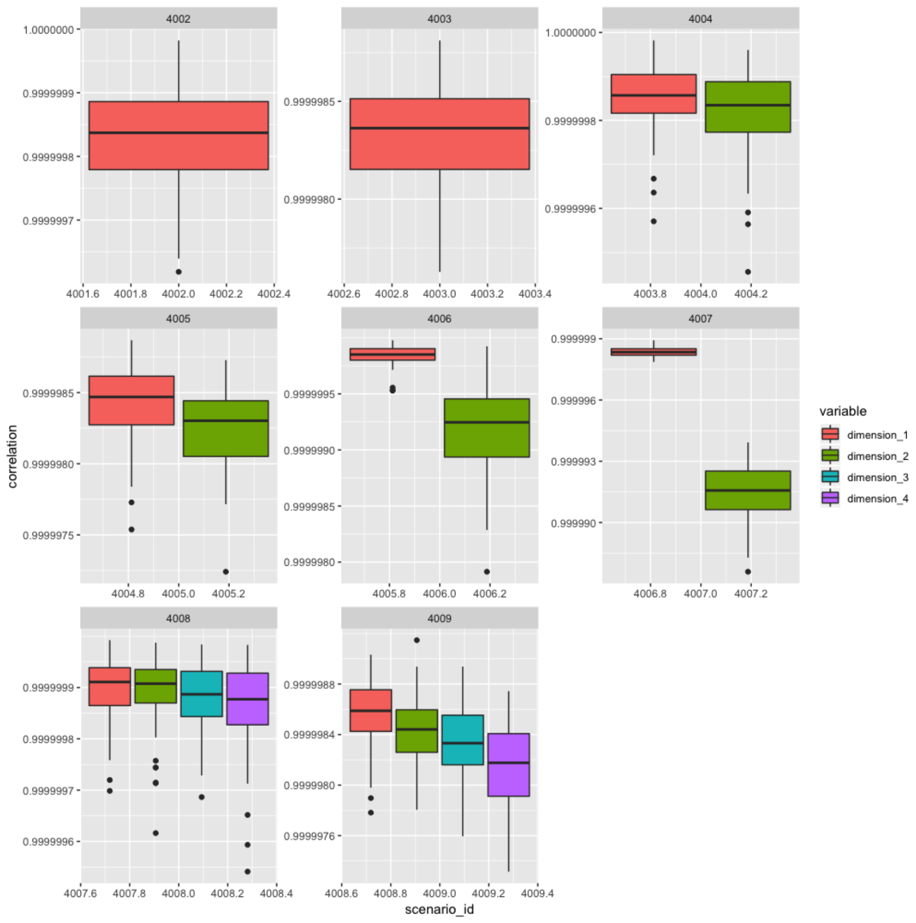


Figure A.7: Correlation for $n = 5 \cdot 10^3$ and MDS Fast algorithm.

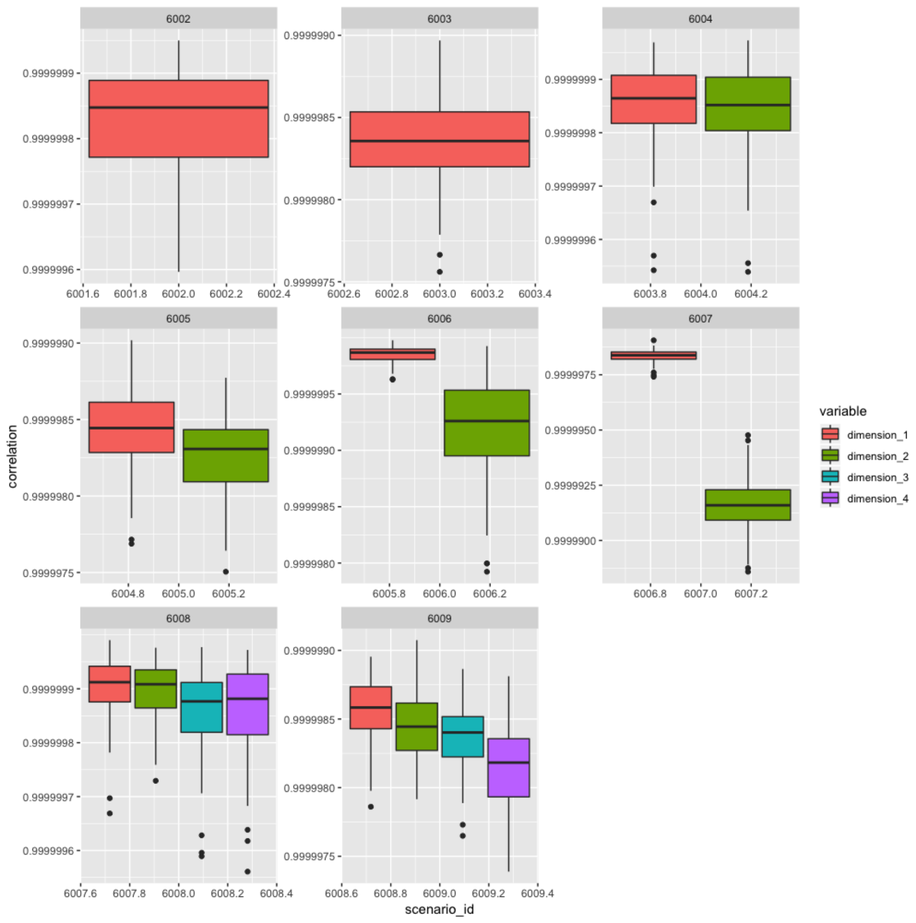


Figure A.8: Correlation for $n = 10^4$ and MDS Fast algorithm.

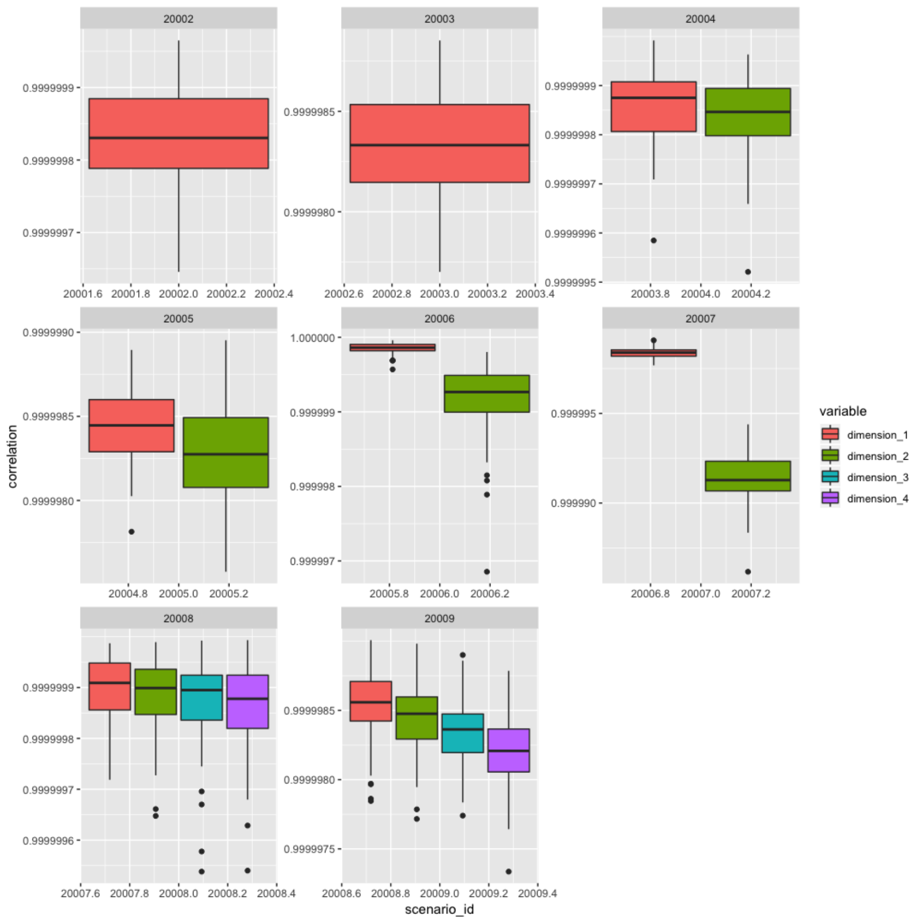


Figure A.9: Correlation for $n = 10^5$ and MDS Fast algorithm.

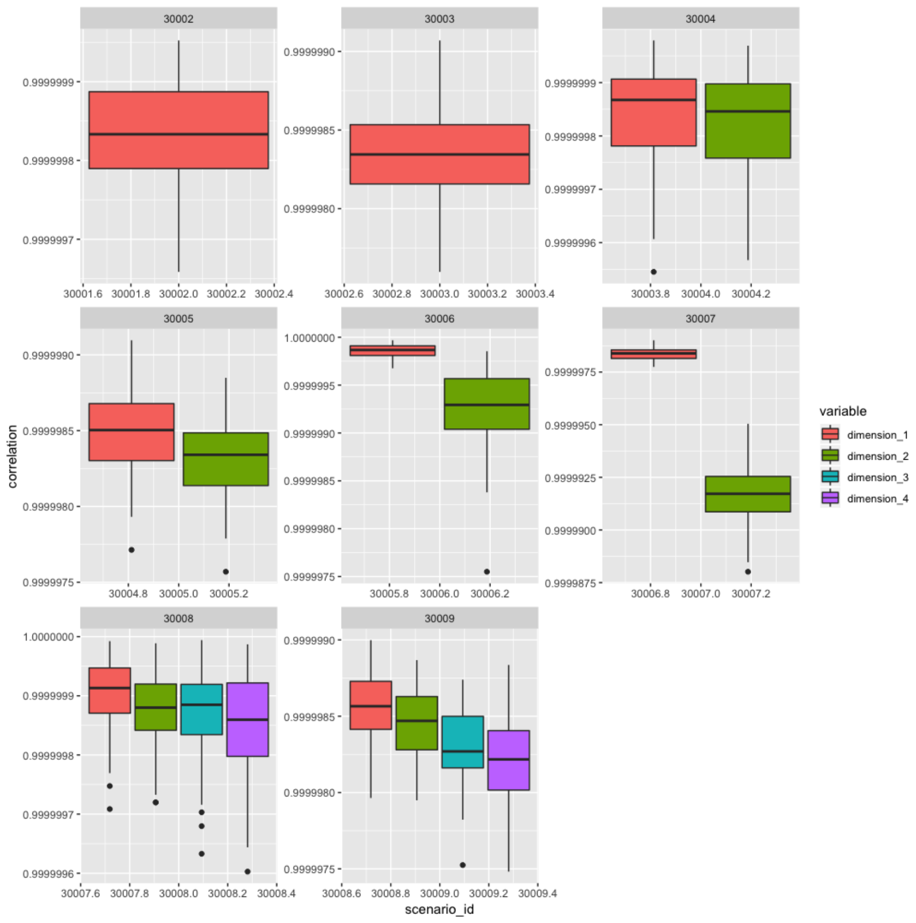


Figure A.10: Correlation for $n = 10^6$ and MDS Fast algorithm.

A.3 MDS based on Gower interpolation

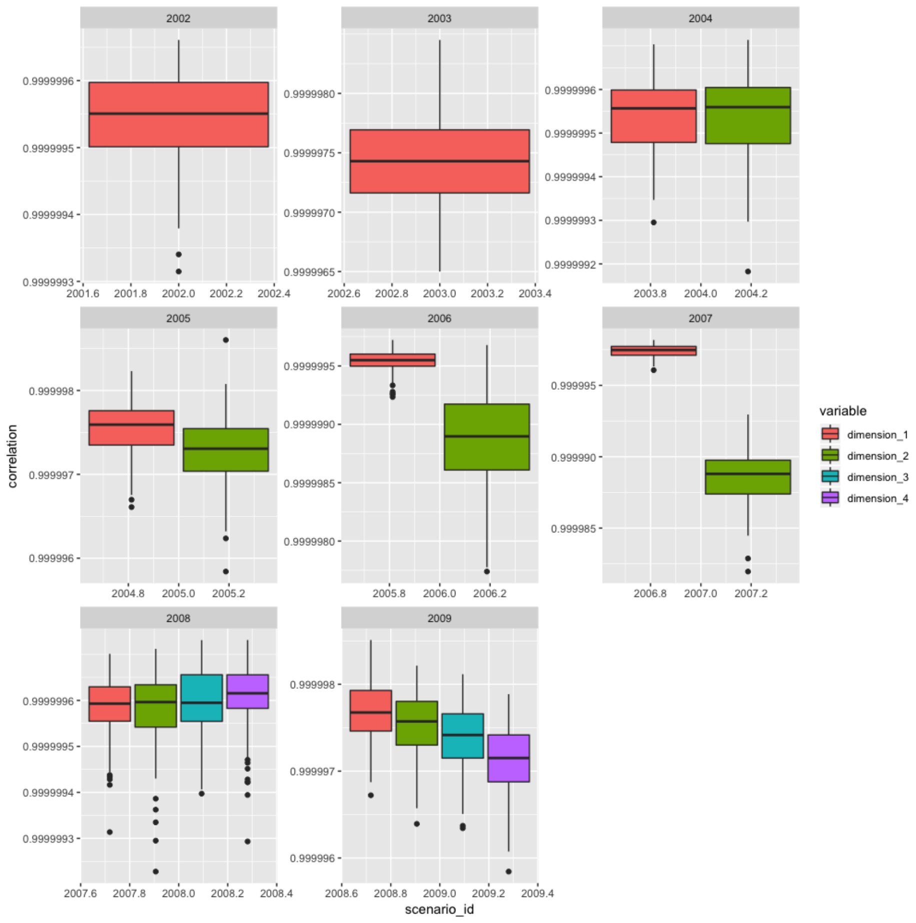


Figure A.11: Correlation for $n = 3 \cdot 10^3$ and MDS based on Gower interpolation algorithm.

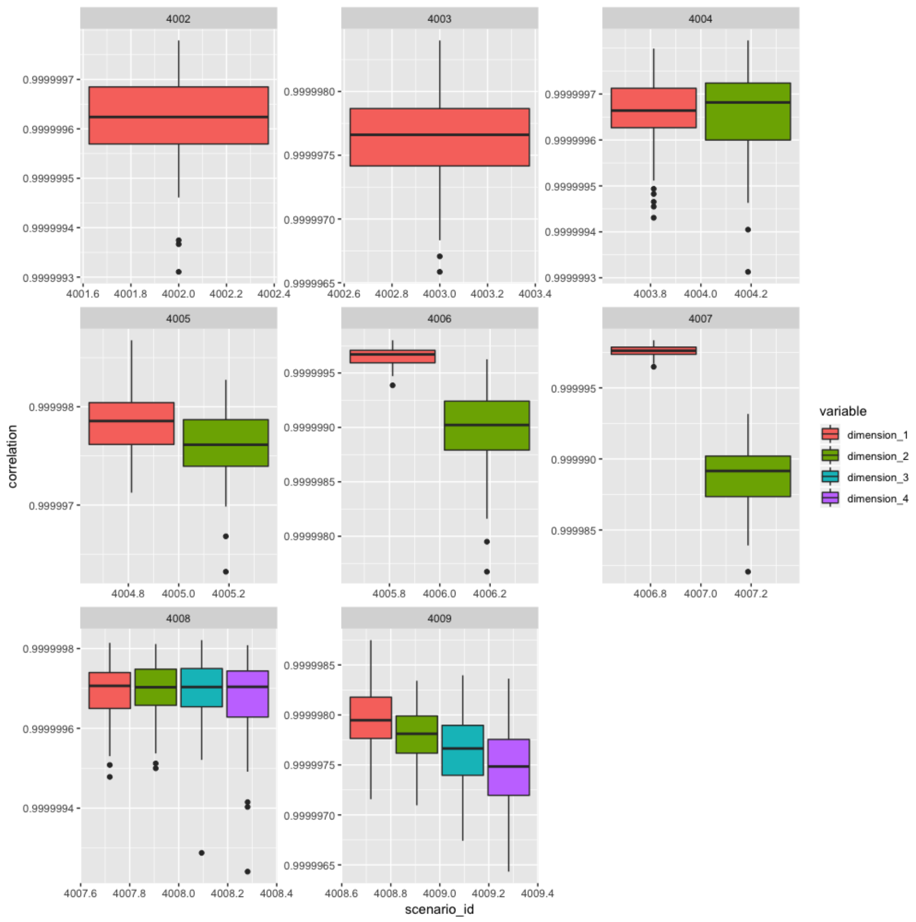


Figure A.12: Correlation for $n = 5 \cdot 10^3$ and MDS based on Gower interpolation algorithm.

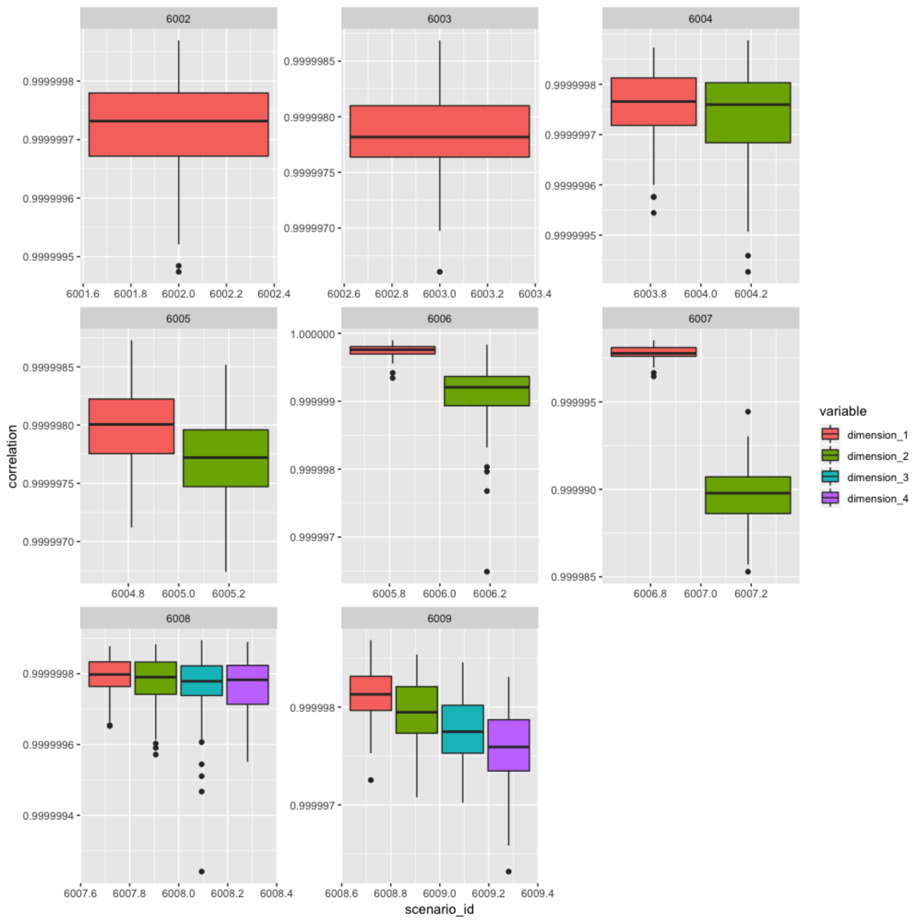


Figure A.13: Correlation for $n = 10^4$ and MDS based on Gower interpolation algorithm.

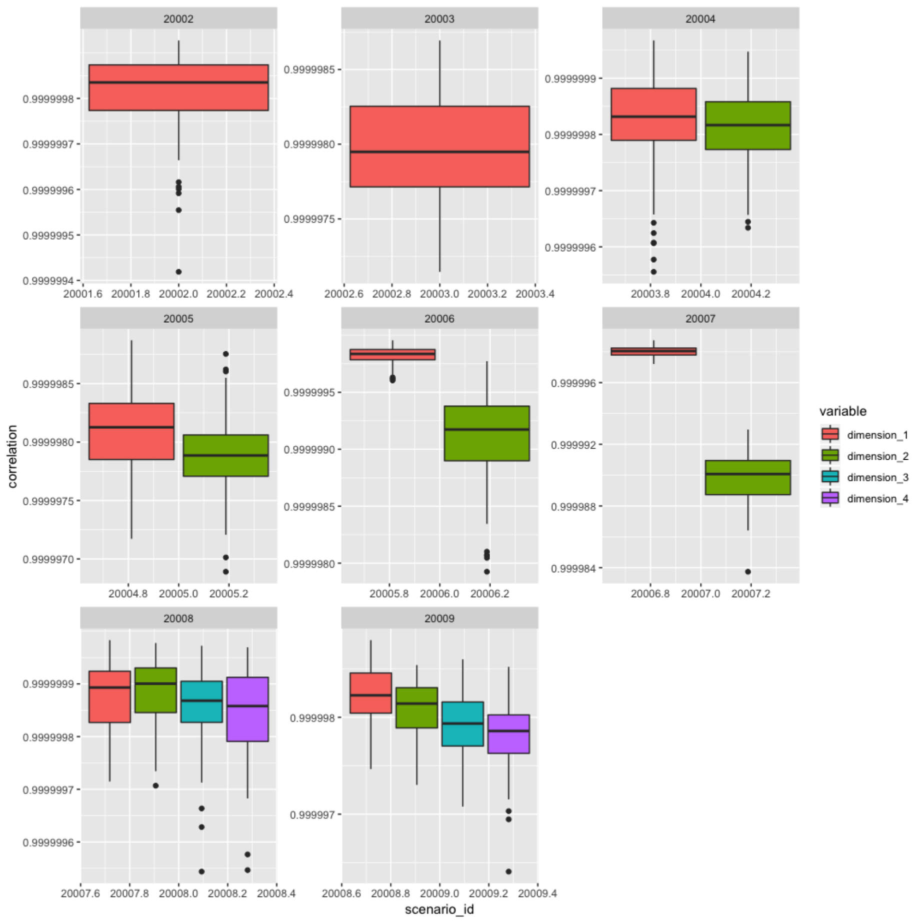


Figure A.14: Correlation for $n = 10^5$ and MDS based on Gower interpolation algorithm.

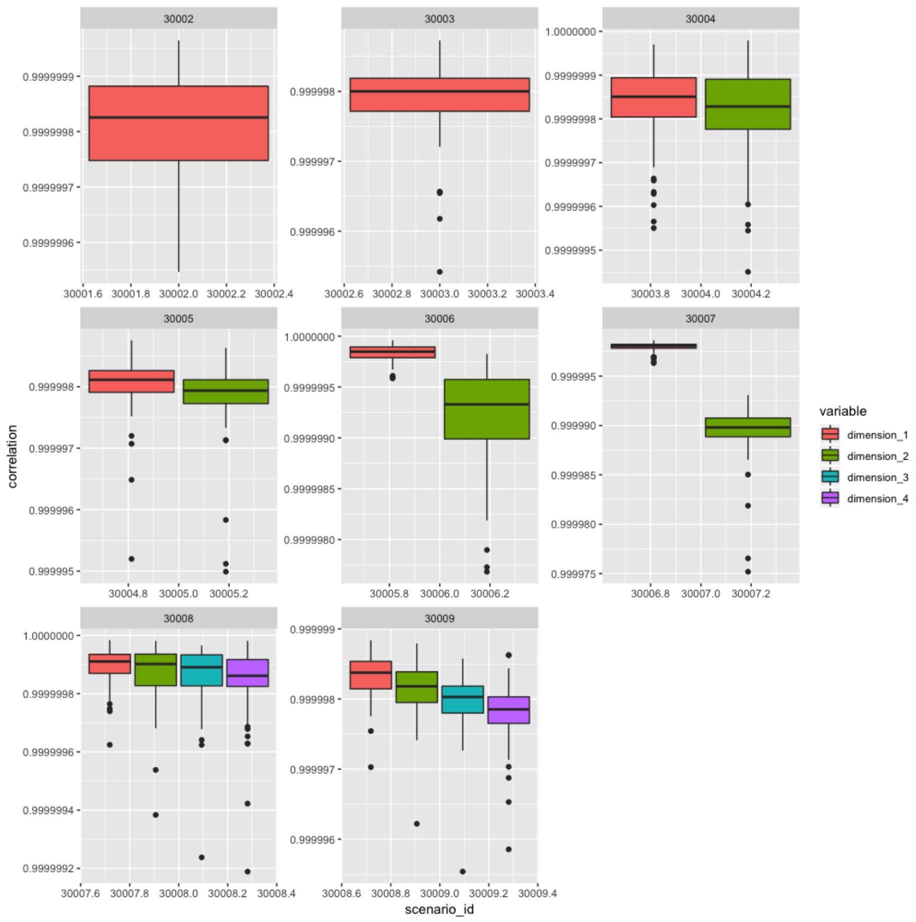


Figure A.15: Correlation for $n = 10^6$ and MDS based on Gower interpolation algorithm.

Appendix B

Bias and MSE for eigenvalues

B.1 Divide and Conquer MDS

	scenario_id	$\overline{\sqrt{\phi_1}}$	bias_1	MSE_1	$\overline{\sqrt{\phi_2}}$	bias_2	MSE_2
1	5	15.41	0.41	17.12	14.59	0.41	16.99
2	6	15.41	0.41	17.08	14.54	0.41	20.88
3	2004	15.40	0.40	15.99	14.55	0.40	19.89
4	2005	15.39	0.39	15.03	14.54	0.39	21.59
5	4004	15.41	0.41	16.71	14.55	0.41	20.01
6	4005	15.41	0.41	16.94	14.57	0.41	18.79
7	6004	15.39	0.39	15.57	14.54	0.39	21.13
8	6005	15.42	0.42	17.60	14.57	0.42	18.22
9	20004	15.40	0.40	15.77	14.56	0.40	19.42
10	20005	15.41	0.41	16.85	14.56	0.41	19.15
11	30004	15.40	0.40	16.00	14.56	0.40	19.52
12	30005	15.41	0.41	16.73	14.56	0.41	18.96

Table B.1: Estimator, bias and MSE for scenarios with two main dimensions $\lambda_1 = 15$ and $\lambda_2 = 15$ for *Divide and Conquer MDS*.

	scenario_id	$\overline{\sqrt{\phi_1}}$	bias_1	MSE_1	$\overline{\sqrt{\phi_2}}$	bias_2	MSE_2
1	7	14.96	-0.04	0.15	9.95	-0.05	0.27
2	8	14.98	-0.02	0.02	9.92	-0.08	0.57
3	2006	15.01	0.01	0.02	9.98	-0.02	0.05
4	2007	15.03	0.03	0.08	9.98	-0.02	0.04
5	4006	15.00	0.00	0.00	9.99	-0.01	0.02
6	4007	15.01	0.01	0.01	9.97	-0.03	0.11
7	6006	15.01	0.01	0.01	9.97	-0.03	0.06
8	6007	15.01	0.01	0.00	10.00	-0.00	0.00
9	20006	15.00	0.00	0.00	9.98	-0.02	0.05
10	20007	15.01	0.01	0.00	9.98	-0.02	0.04
11	30006	15.00	-0.00	0.00	9.97	-0.03	0.08
12	30007	15.00	0.00	0.00	9.98	-0.02	0.03

Table B.2: Estimator, bias and MSE for scenarios with two main dimensions $\lambda_1 = 15$ and $\lambda_2 = 10$ for *Divide and Conquer MDS.*

	scenario_id	$\overline{\sqrt{\phi_1}}$	bias_1	MSE_1	$\overline{\sqrt{\phi_2}}$	bias_2	MSE_2	$\overline{\sqrt{\phi_3}}$	bias_3	MSE_3	$\overline{\sqrt{\phi_4}}$	bias_4	MSE_4
1	9	15.88	0.88	77.09	15.26	0.26	6.94	14.71	-0.29	8.40	14.08	-0.92	84.94
2	10	15.89	0.89	80.08	15.26	0.26	6.52	14.70	-0.30	8.82	14.09	-0.91	83.59
3	2008	15.90	0.90	80.98	15.26	0.26	6.87	14.68	-0.32	10.37	14.04	-0.96	91.72
4	2009	15.88	0.88	76.61	15.25	0.25	6.02	14.70	-0.30	9.07	14.04	-0.96	91.55
5	4008	15.87	0.87	76.05	15.26	0.26	6.58	14.68	-0.32	9.99	14.06	-0.94	87.55
6	4009	15.87	0.87	75.86	15.24	0.24	5.83	14.68	-0.32	10.21	14.07	-0.93	86.76
7	6008	15.89	0.89	79.01	15.25	0.25	6.07	14.69	-0.31	9.80	14.06	-0.94	87.95
8	6009	15.91	0.91	82.54	15.25	0.25	6.32	14.69	-0.31	9.64	14.06	-0.94	87.92
9	20008	15.89	0.89	78.58	15.25	0.25	6.10	14.68	-0.32	9.98	14.06	-0.94	87.77
10	20009	15.89	0.89	79.78	15.25	0.25	6.30	14.69	-0.31	9.69	14.07	-0.93	86.87
11	30008	15.89	0.89	78.65	15.25	0.25	6.09	14.68	-0.32	9.95	14.06	-0.94	88.05
12	30009	15.89	0.89	79.73	15.25	0.25	6.40	14.69	-0.31	9.57	14.07	-0.93	86.68

Table B.3: Estimator, bias and MSE for scenarios with four main dimensions $\lambda_i = 15$ $i \in \{1, 2, 3, 4\}$ for *Divide and Conquer MDS.*

B.2 Fast MDS

	scenario_id	$\overline{\sqrt{\phi_1}}$	bias_1	MSE_1	$\overline{\sqrt{\phi_2}}$	bias_2	MSE_2
1	5	16.06	1.06	112.12	13.79	-1.21	145.86
2	6	15.41	0.41	17.21	14.56	-0.44	19.44
3	2004	15.71	0.71	49.84	14.31	-0.69	46.93
4	2005	15.40	0.40	15.93	14.39	-0.61	37.06
5	4004	15.50	0.50	24.80	14.43	-0.57	32.32
6	4005	15.45	0.45	20.39	14.46	-0.54	29.07
7	6004	16.52	1.52	231.64	13.35	-1.65	271.33
8	6005	15.46	0.46	21.30	14.49	-0.51	25.90
9	20004	15.45	0.45	20.62	14.37	-0.63	40.08
10	20005	15.41	0.41	17.03	14.55	-0.45	20.34
11	30004	15.67	0.67	45.00	14.27	-0.73	52.78
12	30005	15.35	0.35	12.11	14.57	-0.43	18.11

Table B.4: Estimator, bias and MSE for scenarios with two main dimensions $\lambda_1 = 15$ and $\lambda_2 = 15$ for *Fast MDS*.

	scenario_id	$\overline{\sqrt{\phi_1}}$	bias_1	MSE_1	$\overline{\sqrt{\phi_2}}$	bias_2	MSE_2
1	7	15.00	0.00	0.00	9.82	-0.18	3.22
2	8	14.96	-0.04	0.17	9.93	-0.07	0.56
3	2006	15.14	0.14	2.00	9.88	-0.12	1.37
4	2007	14.99	-0.01	0.00	9.98	-0.02	0.04
5	4006	14.91	-0.09	0.75	9.95	-0.05	0.21
6	4007	14.89	-0.11	1.29	9.90	-0.10	0.92
7	6006	15.10	0.10	0.93	9.64	-0.36	13.12
8	6007	14.97	-0.03	0.10	9.96	-0.04	0.12
9	20006	15.02	0.02	0.05	9.93	-0.07	0.47
10	20007	14.99	-0.01	0.02	10.02	0.02	0.03
11	30006	14.88	-0.12	1.38	9.91	-0.09	0.74
12	30007	15.03	0.03	0.09	9.99	-0.01	0.02

Table B.5: Estimator, bias and MSE for scenarios with two main dimensions $\lambda_1 = 15$ and $\lambda_2 = 10$ for *Fast MDS*.

	scenario_id	$\overline{\sqrt{\phi_1}}$	bias_1	MSE_1	$\overline{\sqrt{\phi_2}}$	bias_2	MSE_2	$\overline{\sqrt{\phi_3}}$	bias_3	MSE_3	$\overline{\sqrt{\phi_4}}$	bias_4	MSE_4
1	9	17.43	2.43	590.46	15.58	0.58	33.32	13.74	-1.26	158.12	11.99	-3.01	903.29
2	10	15.90	0.90	80.54	15.25	0.25	6.49	14.70	-0.30	8.79	14.08	-0.92	84.90
3	2008	16.40	1.40	195.78	15.27	0.27	7.32	14.31	-0.69	47.00	13.33	-1.67	280.08
4	2009	16.01	1.01	102.73	15.32	0.32	10.08	14.65	-0.35	11.97	13.87	-1.13	127.87
5	4008	16.14	1.14	130.55	15.29	0.29	8.64	14.53	-0.47	22.40	13.81	-1.19	141.75
6	4009	16.02	1.02	104.48	15.32	0.32	10.27	14.63	-0.37	13.72	14.00	-1.00	99.58
7	6008	17.96	2.96	875.12	15.53	0.53	27.71	13.70	-1.30	169.28	11.42	-3.58	1281.54
8	6009	16.01	1.01	102.49	15.29	0.29	8.26	14.69	-0.31	9.35	13.93	-1.07	114.86
9	20008	16.14	1.14	129.03	15.39	0.39	15.54	14.63	-0.37	13.90	13.94	-1.06	112.16
10	20009	15.98	0.98	96.04	15.25	0.25	6.28	14.61	-0.39	15.18	13.97	-1.03	106.71
11	30008	16.29	1.29	165.16	15.22	0.22	4.73	14.37	-0.63	39.55	13.44	-1.56	243.94
12	30009	15.98	0.98	95.75	15.33	0.33	10.87	14.70	-0.30	9.21	14.06	-0.94	87.68

Table B.6: Estimator, bias and MSE for scenarios with four main dimensions $\lambda_i = 15$ $i \in \{1, 2, 3, 4\}$ for *Fast MDS*.

B.3 MDS based on Gower interpolation

	scenario_id	$\overline{\sqrt{\phi_1}}$	bias_1	MSE_1	$\overline{\sqrt{\phi_2}}$	bias_2	MSE_2
1	5	15.45	0.45	20.49	14.63	-0.37	13.52
2	6	15.41	0.41	17.04	14.56	-0.44	19.36
3	2004	15.42	0.42	17.23	14.59	-0.41	16.71
4	2005	15.35	0.35	12.08	14.47	-0.53	28.25
5	4004	15.45	0.45	19.82	14.55	-0.45	19.91
6	4005	15.40	0.40	16.06	14.52	-0.48	23.04
7	6004	15.36	0.36	13.23	14.53	-0.47	22.56
8	6005	15.40	0.40	15.75	14.53	-0.47	22.29
9	20004	15.36	0.36	12.93	14.47	-0.53	27.88
10	20005	15.38	0.38	14.27	14.60	-0.40	16.20
11	30004	15.37	0.37	13.86	14.52	-0.48	22.97
12	30005	15.33	0.33	11.15	14.57	-0.43	18.74

Table B.7: Estimator, bias and MSE for scenarios with two main dimensions $\lambda_1 = 15$ and $\lambda_2 = 15$ for *MDS based on Gower interpolation.*

	scenario_id	$\overline{\sqrt{\phi_1}}$	bias_1	MSE_1	$\overline{\sqrt{\phi_2}}$	bias_2	MSE_2
1	7	14.95	-0.05	0.25	9.95	-0.05	0.24
2	8	14.96	-0.04	0.18	9.93	-0.07	0.51
3	2006	15.05	0.05	0.27	9.98	-0.02	0.06
4	2007	15.02	0.02	0.04	9.99	-0.01	0.01
5	4006	14.98	-0.02	0.05	9.98	-0.02	0.03
6	4007	14.92	-0.08	0.63	9.90	-0.10	1.05
7	6006	14.98	-0.02	0.02	9.98	-0.02	0.04
8	6007	14.97	-0.03	0.07	9.97	-0.03	0.11
9	20006	15.05	0.05	0.28	9.97	-0.03	0.07
10	20007	15.00	-0.00	0.00	10.00	-0.00	0.00
11	30006	14.89	-0.11	1.19	9.98	-0.02	0.05
12	30007	15.03	0.03	0.09	9.98	-0.02	0.06

Table B.8: Estimator, bias and MSE for scenarios with two main dimensions $\lambda_1 = 15$ and $\lambda_2 = 10$ for *MDS based on Gower interpolation.*

	scenario_id	$\sqrt{\phi_1}$	bias$_1$	MSE$_1$	$\sqrt{\phi_2}$	bias$_2$	MSE$_2$	$\sqrt{\phi_3}$	bias$_3$	MSE$_3$	$\sqrt{\phi_4}$	bias$_4$	MSE$_4$
1	9	15.89	0.89	79.83	15.28	0.28	7.68	14.73	-0.27	7.38	14.08	-0.92	84.17
2	10	15.90	0.90	80.72	15.26	0.26	7.00	14.70	-0.30	9.07	14.10	-0.90	81.70
3	2008	15.83	0.83	68.94	15.20	0.20	4.00	14.63	-0.37	13.55	14.03	-0.97	94.05
4	2009	15.86	0.86	73.48	15.24	0.24	5.66	14.67	-0.33	10.78	14.02	-0.98	96.87
5	4008	15.88	0.88	76.87	15.24	0.24	5.91	14.66	-0.34	11.64	14.08	-0.92	84.24
6	4009	15.93	0.93	86.05	15.27	0.27	7.03	14.67	-0.33	10.97	14.12	-0.88	77.32
7	6008	15.87	0.87	75.93	15.23	0.23	5.30	14.67	-0.33	10.59	14.05	-0.95	91.13
8	6009	15.90	0.90	80.39	15.26	0.26	6.61	14.68	-0.32	10.22	14.07	-0.93	86.34
9	20008	15.97	0.97	94.74	15.30	0.30	9.03	14.70	-0.30	8.94	14.13	-0.87	74.89
10	20009	15.88	0.88	77.34	15.22	0.22	5.03	14.66	-0.34	11.33	14.10	-0.90	81.27
11	30008	15.88	0.88	77.43	15.20	0.20	4.12	14.63	-0.37	13.67	14.03	-0.97	94.66
12	30009	15.97	0.97	94.19	15.33	0.33	10.85	14.70	-0.30	9.29	14.07	-0.93	86.59

Table B.9: Estimator, bias and MSE for scenarios with four main dimensions $\lambda_i = 15$ $i \in \{1, 2, 3, 4\}$ for *Fast MDS*.

Appendix C

Time required to compute MDS configuration

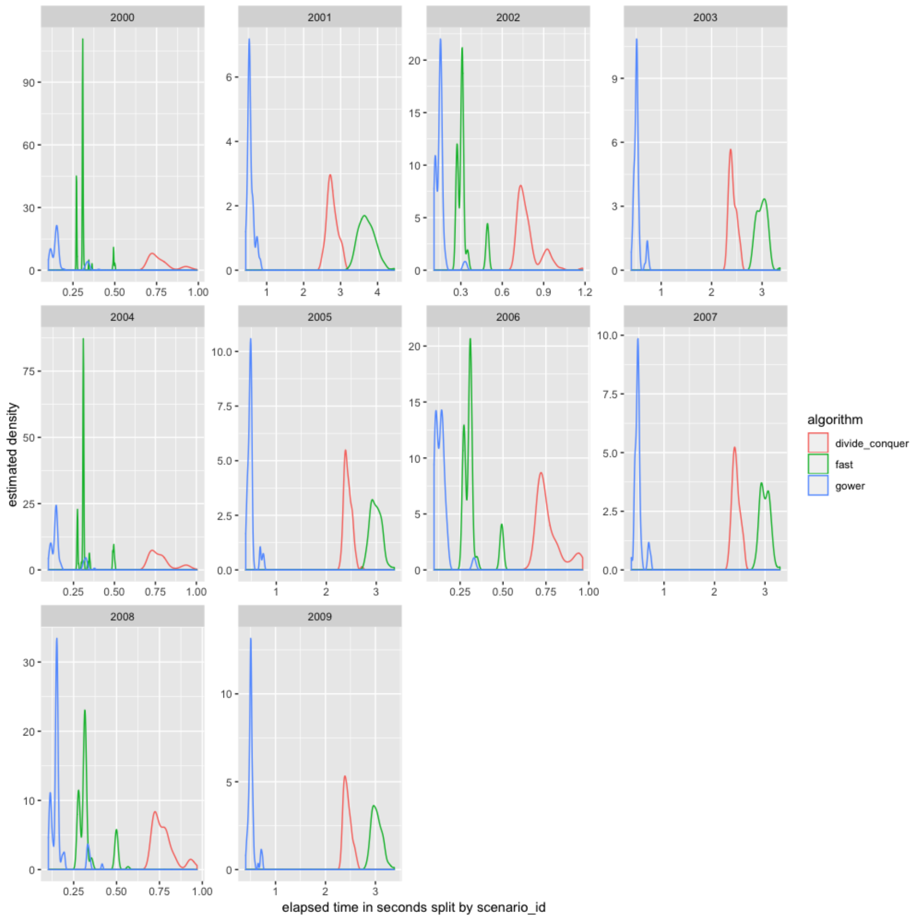


Figure C.1: Elapsed time for each algorithm and each *scenario_id* of $n = 3 \cdot 10^3$.

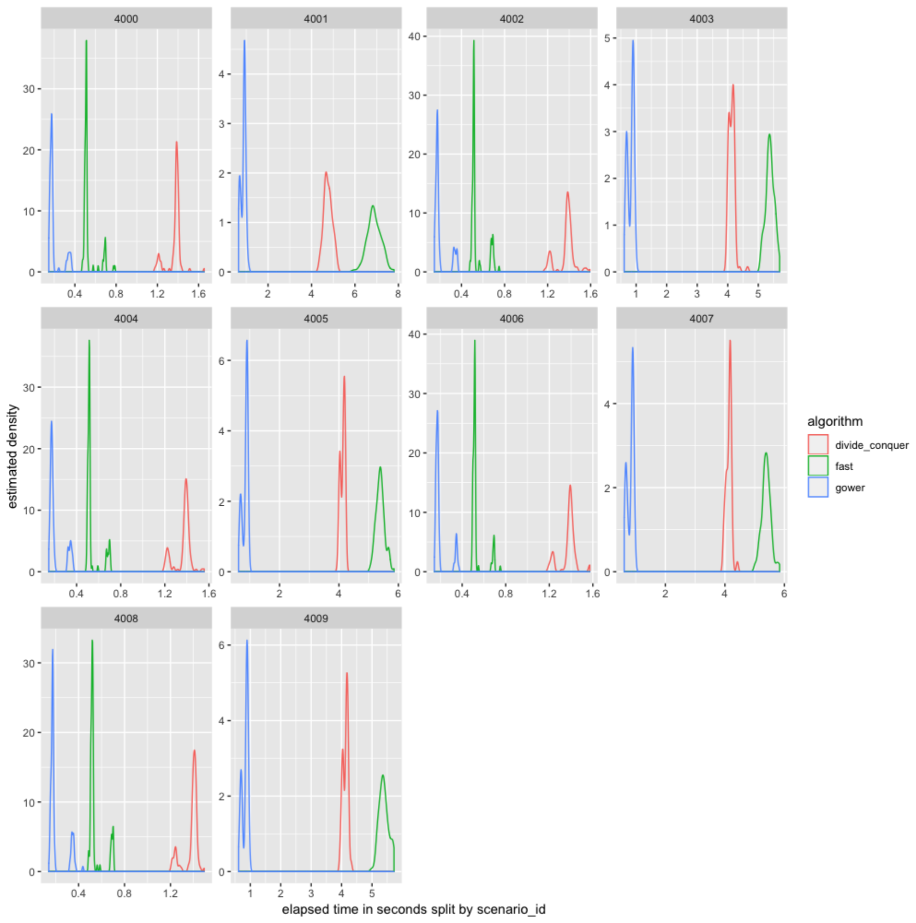


Figure C.2: Elapsed time for each algorithm and each *scenario_id* of $n = 5 \cdot 10^3$.

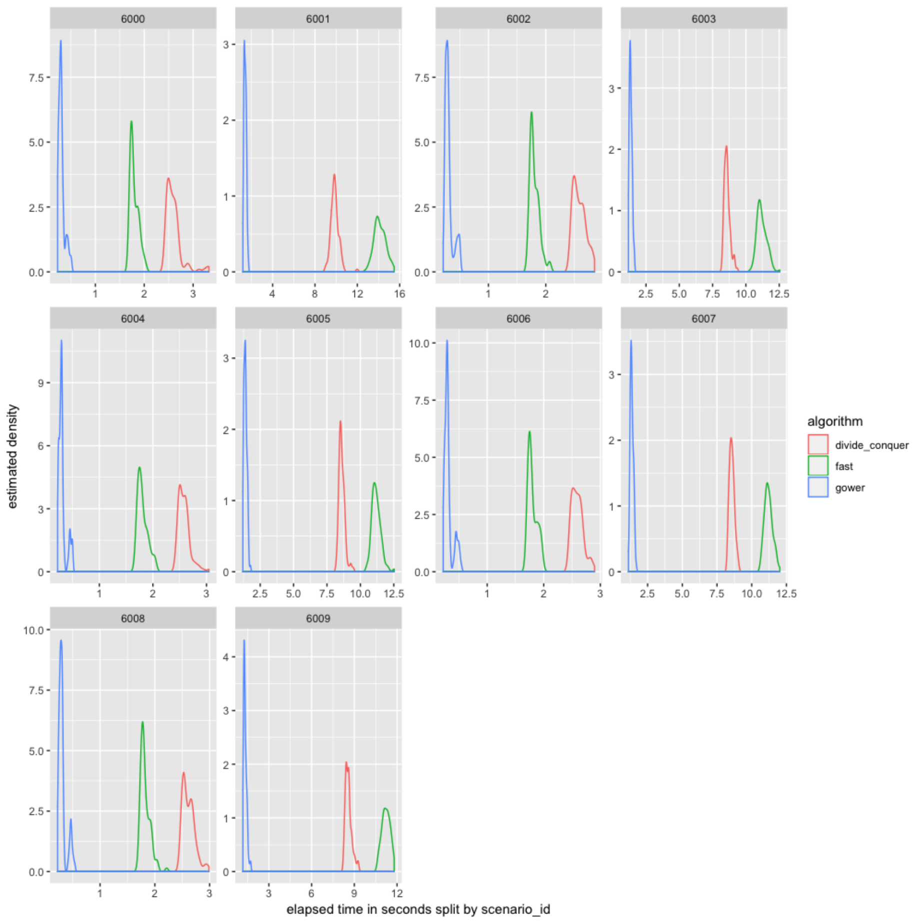


Figure C.3: Elapsed time for each algorithm and each *scenario_id* of $n = 10^4$.

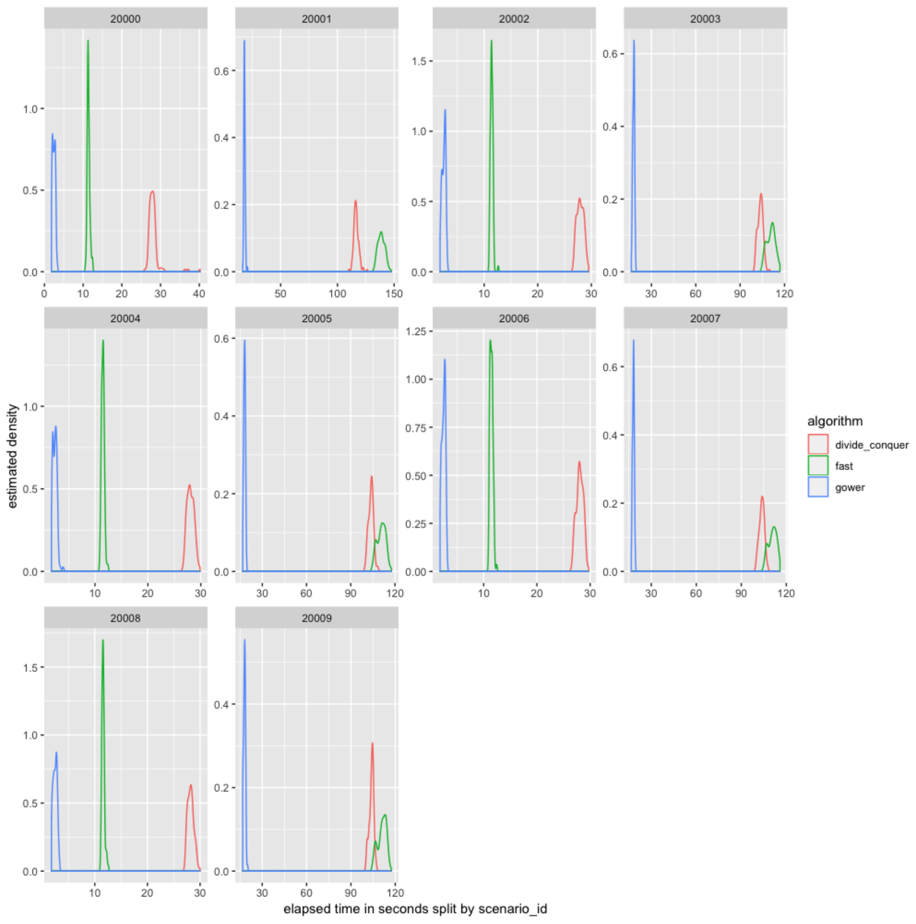


Figure C.4: Elapsed time for each algorithm and each *scenario_id* of $n = 10^5$.

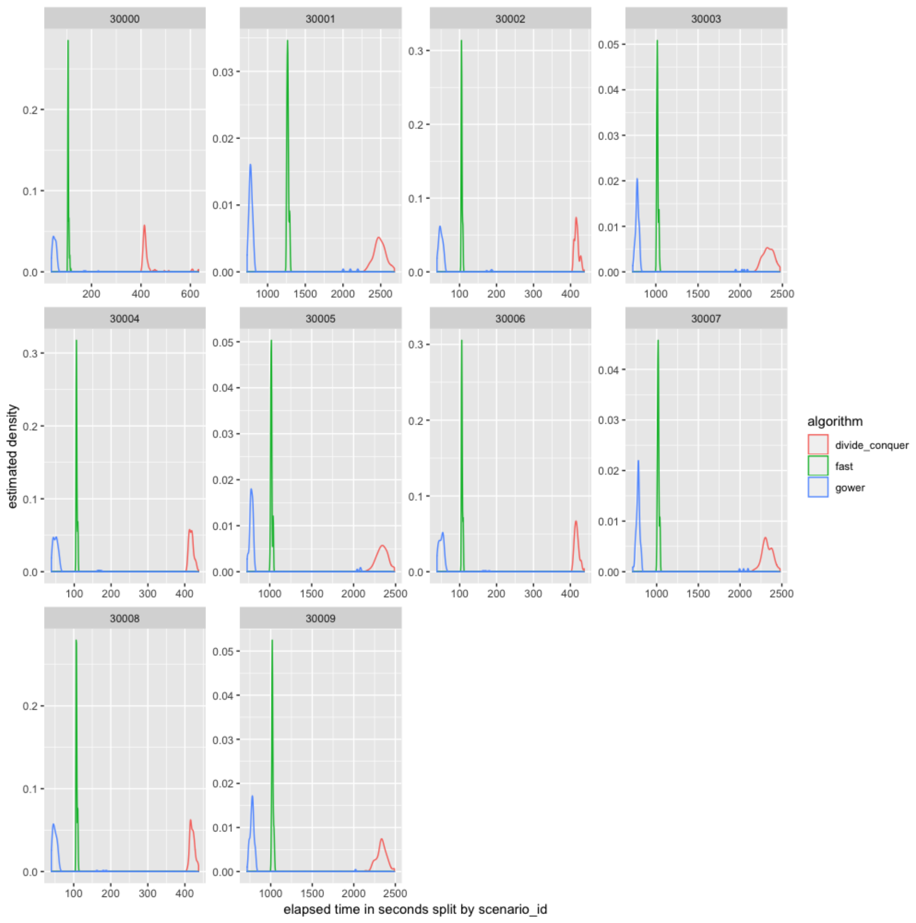


Figure C.5: Elapsed time for each algorithm and each *scenario_id* of $n = 10^6$.

Appendix D

Code

D.1 Divide and Conquer MDS

```
divide_conquer.mds <- function(
  x,
  l,
  s,
  metric
){

  # List positions
  ls_positions = list()
  list_eigenvalues = list()
  i_eigen = 1

  # Initial parameters
  p = ceiling(2*nrow(x)/l)
  groups = sample(x = p, size = nrow(x), replace = TRUE)
  groups = sort(groups)
  unique_group = unique(groups)
  total_groups = length(unique_group)

  for(k in 1:total_groups){
    # Getting the group that is being processed
    current_group = unique_group[k]
    x_positions_current_group = which(groups == current_group)
    ls_positions[[k]] = x_positions_current_group

    # Take the data in the following way:
    #   If it is the first iteration, take the data from first
      group
    #   else, take the data from k-1 and k groups
    if(k == 1){
      filter_rows_by_position = x_positions_current_group
      rows_processed = x_positions_current_group
    }else{

      rows_processed = c(
        rows_processed,
        x_positions_current_group
      )
      previous_group = unique_group[k-1]
      x_positions_previous_group =  which(groups == previous_group
         )

      # Rows to be filtered
      filter_rows_by_position = c(
        x_positions_previous_group,
        x_positions_current_group
      )

    }

    # Matrix to apply MDS
    submatrix_data = x[filter_rows_by_position, ]

    # Calculate distance
    distance_matrix = cluster::daisy(
      x = submatrix_data,
      metric = metric
    )

    # Applying MDS to the submatrix of data
    cmd_eig = stats::cmdscale(
      d = distance_matrix,
      k = s,
      eig = TRUE
    )

    mds_iteration = cmd_eig$points
    if(p%%2 == 0){
      if(k%%2 == 0){
        list_eigenvalues[[i_eigen]] = cmd_eig$eig/nrow(
            submatrix_data)
        i_eigen = i_eigen + 1
      }
    }else{
      if(k %% 2 == 1){
        list_eigenvalues[[i_eigen]] = cmd_eig$eig/nrow(
            submatrix_data)
        i_eigen = i_eigen + 1
      }
    }

        row.names(mds_iteration) = row.names(submatrix_data)

    if(k == 1){
      # Define cum-MDS as MDS(1)
      cum_mds = mds_iteration

    }else{
      # Take the result of MDS(k-1) obtained with k-1 and k
      rn_prev = row.names(x)[x_positions_previous_group]
      rn_cur = row.names(x)[x_positions_current_group]
      pos_previous_group_current_mds = row.names(mds_iteration) %
         in% rn_prev
```

```r
      pos_current_group_current_mds = row.names(mds_iteration) %in% rn_cur
      mds_previous = mds_iteration[pos_previous_group_current_mds,]
      mds_current = mds_iteration[pos_current_group_current_mds,]

      # From cum-MDS take the result of group k-1
      positions_cum_sum_previous = which(row.names(cum_mds) %in% rn_prev
      cum_mds_previous = cum_mds[positions_cum_sum_previous, ]


      # Apply Procrustes transformation
      procrustes_result =  MCMCpack::procrustes(
        X = mds_previous, #The matrix to be transformed
        Xstar = cum_mds_previous, # target matrix
        translation = TRUE,
        dilation = TRUE
      )

      rotation_matrix = procrustes_result$R
      dilation = procrustes_result$s
      translation = procrustes_result$tt
      ones_vector = rep(1, nrow(mds_current))
      translation_matrix = ones_vector %*% t(translation)

      # Transform the data for the k-th group
      cum_mds_current = dilation * mds_current %*% rotation_matrix +
        translation_matrix

      cum_mds = rbind(
        cum_mds,
        cum_mds_current
      )

    }

  }

  # Reordering
  reording_permutation = match(1:nrow(x), rows_processed)
  cum_mds = cum_mds[reording_permutation, ]


  return(
    list(
      points = cum_mds,
      eig = list_eigenvalues
    )
  )
}
```

D.2 Fast MDS

```r
fast_mds <- function(
  x,
  l,
  s,
  k,
  metric
){

  # Initial parameters
  list_matrix = list()
  list_index = list()
  list_mds = list()
  list_mds_align = list()

  sub_sample_size = k * s
  n = nrow(x)

  # Division into p matrices
  # When doing the partitions it can happen that there are so many
      matrices
  # that s*k < nrow(x_i). In this case, we do a sampling again

  p = ceiling(l/sub_sample_size)
  observations_division = sample(x = p, size = nrow(x), replace =
     TRUE)
  observations_division = sort(observations_division)
  min_sample_size = min(table(observations_division))

  while( min_sample_size < sub_sample_size && p > 1){
    p = p - 1
    observations_division = sample(x = p, size = nrow(x), replace
       = TRUE)
    observations_division = sort(observations_division)
    min_sample_size = min(table(observations_division))
  }



  # Partition into p submatrices
  for(i_group in 1:p){
    ind = which(observations_division == i_group)
    list_matrix[[i_group]] = x[ind, ]
  }

  able_to_do_mds = n/p <= l | p == 1


  # We can do MDS
  if(able_to_do_mds == TRUE){
    message(paste0("Non-recursive!!"))
    for (i_group in 1:p) {

      matrix_filter = list_matrix[[i_group]]

      # MDS for each submatrix
      distance_matrix = daisy(
        x = matrix_filter,
        metric = metric
      )

      list_mds[[i_group]] = stats::cmdscale(
        d = distance_matrix,
        k = s
      )


      # Subsample
      sample_size = sub_sample_size
      if(sample_size > length( row.names(matrix_filter ) ) ){
        sample_size = length( row.names(matrix_filter ) )
      }


      list_index[[i_group]] = sample(
        x = row.names(matrix_filter),
        size = sample_size,
        replace = FALSE
      )


      # Building x_M_align
      ind_M = which(row.names(x) %in% list_index[[i_group]])
      if(i_group == 1){
        x_M_align = x[ind_M, ]
      }else{
        x_M_align = rbind(
          x_M_align,
          x[ind_M, ]
        )
      }

    }

    # M_align: MDS over x_M_align
    distance_matrix_M = distance_matrix = daisy(
      x = x_M_align,
      metric = metric
    )

    M_align = stats::cmdscale(
      d = distance_matrix_M,
      k = s
    )

    # Global alignment
    for(i_group in 1:p){
      row_names = list_index[[i_group]]

      ind_M = which(row.names(M_align) %in% row_names)
      M_align_filter =  M_align[ind_M, ]

      di = list_mds[[i_group]]
      ind_mds = which(row.names( di ) %in% row_names)
      di_filter = di[ind_mds, ]

      # Alignment
      procrustes_result =  MCMCpack::procrustes(
        X = di_filter, #The matrix to be transformed
        Xstar = M_align_filter, # target matrix
        translation = TRUE,
        dilation = TRUE
      )

      rotation_matrix = procrustes_result$R
      dilation = procrustes_result$s
      translation = procrustes_result$tt
      ones_vector = rep(1, nrow(di))
      translation_matrix = ones_vector %*% t(translation)


      tranformation_di = dilation * di %*% rotation_matrix + translation_matrix


      # Append
      if(i_group == 1){
        Z = tranformation_di
      } else{
        Z = rbind(
          Z,
          tranformation_di
        )

      }
    }

    row.names(Z) = row.names(x)

  }else{
    message("Recursive!!!")
    list_zi ← list()
    list_index ← list()

    for(i_group in 1:p){
      # Apply the algorithm
      list_zi[[i_group]] = fast_mds(
        x = list_matrix[[i_group]],
        n = nrow(list_matrix[[i_group]]),
        l = l,
        s = s,
        k = k,
        metric = metric
      )

      #Take a subsample
      list_index[[i_group]] = sample(
        x = row.names( list_zi[[i_group]] ),
        size = k * s,
        replace = FALSE
      )

      ind = which( row.names( list_zi[[i_group]] ) %in% list_index
         [[i_group]])
      submatrix = list_matrix[[i_group]][ind, ]


      if(i_group == 1){
        x_M_align = submatrix
      } else{
        x_M_align = rbind(
          x_M_align,
          submatrix
        )
      }
    }

    message(paste0("         At the end x_M_align has ", nrow(
       x_M_align), " rows"))


    distance_matrix_M  = daisy(
      x = x_M_align,
      metric = metric
    )

    M_align = stats::cmdscale(
      d = distance_matrix_M,
      k = s
    )


    # Global alignment
    for(i_group in 1:p){
      row_names = list_index[[i_group]]

      ind_M = which(row.names(x_M_align) %in% row_names)
      M_align_filter =  M_align[ind_M, ]

      di = list_zi[[i_group]]
      ind_mds = which(row.names( di ) %in% row_names)
      di_filter = di[ind_mds, ]

      # Alignment
      procrustes_result =  MCMCpack::procrustes(
        X = di_filter, #The matrix to be transformed
        Xstar = M_align_filter, # target matrix
        translation = TRUE,
        dilation = TRUE
      )

```

```r
      rotation_matrix = procrustes_result$R
      dilation = procrustes_result$s
      translation = procrustes_result$tt
      ones_vector = rep(1, nrow(di))
      translation_matrix = ones_vector %*% t(translation)


      tranformation_di = dilation * di %*% rotation_matrix +
        translation_matrix


      # Append
      if(i_group == 1){
        Z = tranformation_di
      } else{
        Z = rbind(
          Z,
          tranformation_di
        )

      }
    }

    row.names(Z) = row.names(x)

  }

  return(
    list(
      Z = Z,
      eigenvalues = NA
    )
  )
}
```

D.3 MDS based on Gower interpolation

```r
gower.interpolation.mds ← function(
  x,
  l,
  s
){

  nrow_x = nrow(x)
  p = ceiling(nrow_x/l)
  if(p<1) p = 1

  if( p>1 ){
    # Do MDS with the first group and then use the Gower
       interpolation formula
    sample_distribution = sample(x = p, size = nrow_x, replace =
       TRUE)
    sample_distribution = sort(sample_distribution)

    # Get the first group
    ind_1 = which(sample_distribution == 1)
    n_1 = length(ind_1)

    # Do MDS with the first group
    submatrix_data = x[ind_1, ]

    distance_matrix = cluster::daisy(
      x = submatrix_data,
      metric = "euclidean"
    )

    distance_matrix = as.matrix(distance_matrix)

    # MDS for the first group
    cmd_eig = stats::cmdscale(
      d = distance_matrix,
      k = s,
      eig = TRUE
    )

    M = cmd_eig$points
    eigen = cmd_eig$eig/nrow(M)
    cum_mds = M

    # Calculations needed to do Gower interpolation
    delta_matrix = distance_matrix^2
    In = diag(n_1)
    ones_vector = rep(1, n_1)
    J = In - 1/n_1*ones_vector %*% t(ones_vector)
    G = -1/2 * J %*% delta_matrix %*% t(J)
    g_vector = diag(G)
    # S = cov(M)
    S = 1/(nrow(M)-1)*t(M) %*% M
    S_inv = solve(S)

    # For the rest of the groups, do the interpolation
    for(i_group in 2:p){
      # Filtering the data
      ind_i_group = which(sample_distribution == i_group)
      submatrix_data = x[ind_i_group, ]


      # A matrix
      distance_matrix_filter = pdist::pdist(
        X = submatrix_data,
        Y = x[ind_1, ]
      )

      distance_matrix_filter = as.matrix(distance_matrix_filter)
      A = distance_matrix_filter^2
      ones_vector = rep(1, length(ind_i_group))
      MDS_i_group = 1/(2*n_1)*(ones_vector %*%t(g_vector) - A) %*%
          M %*% S_inv
      cum_mds = rbind(
        cum_mds,
        MDS_i_group
      )
    }
  }else{
    # It is possible to run MDS directly
    distance_matrix = cluster::daisy(
      x = x,
      metric = "euclidean"
    )

    distance_matrix = as.matrix(distance_matrix)

    # MDS for the first groups
    cmd_eig = stats::cmdscale(
      d = distance_matrix,
      k = s,
      eig = TRUE
    )

    cum_mds = cmd_eig$points
    eigen = cmd_eig$eig/nrow_x
  }

  return(
    list(
      points = cum_mds,
      eig = eigen
    )
  )
}
```