The bigMap R-package: quick reference

J.Garriga; F.Bartumeus
ICREA Movement Ecology Laboratory (CEAB-CSIC)
jgarriga@ceab.csic.es

Version 2.0.0, mar 2019

Abstract

The bigMap R-package integrates a *mapping method* (*i.e.* a multi-step unsupervised clustering protocol) for large-scale structured data. The protocol follows three steps: a dimensionality reduction of the data, a density estimation over the low dimensional representation of the data, and a final segmentation of the density landscape. For the dimensionality reduction step we use a parallelized implementation of the well-known *t-Stochastic Neighbouring Embedding* (t-SNE) algorithm that significantly alleviates some inherent limitations, while improving its suitability for large datasets. For the clustering we use the *perplexity-adaptive Kernel Density Estimation* (paKDE) algorithm, an adaptive KDE particularly coupled with the t-SNE framework in order to get accurate density estimates out of the embedded data, and the *watertrack transform* (WTT) a variant of the *rainfalling watershed* algorithm to identify clusters within the density landscape.

Contents

1 bigMap: Big data mapping with parallelized t-SNE 3

1.1 Package dependencies . . . 3

2 Using the package 3

2.1 Mapping data . . . 5

2.1.1 Initialization: bdm.init() . . . 5

2.1.2 First step: bdm.ptsne() . . . 6

2.1.3 Second step: bdm.pakde() . . . 10

2.1.4 Third step: bdm.wtt() . . . 11

2.1.5 bdm.labels() . . . 12

2.2 Visual assessment of the output. . . . 12

2.2.1 bdm.cost() . . . 12

2.2.2 bdm.plot() . . . 13

2.2.3 bdm.qMap() . . . 15

2.2.4 bdm.dMap() . . . 15

2.2.5 bdm.boxp() . . . 16

3 Working with remote multiprocessor systems 17

3.1 Easy managing of ptSNE runs. . . . 17

3.1.1 bdm.mybdm() . . . 17

3.1.2 bdm.fName() . . . 18

3.1.3 bdm.save() . . . 18

3.1.4 bdm.local() . . . 18

3.1.5 bdm.scp() . . . 18

3.2 Summary script . . . 18

3.2.1 Intra-node (socket) parallelization . . . 19

3.2.2 Inter-node (MPI) parallelization . . . 20

References 21

1 bigMap: Big data mapping with parallelized t-SNE

The **bigMap** R-package integrates an improved *mapping method* (MM (Todd, Kain, and Bivort 2017, Garriga and Bartumeus (2018))) within R's (R Core Team 2018) environment. The *bigMap* protocol is based on the following three steps:

- dimensionality reduction down to 2D, based on a parallelized implementation of the t-Stochastic Neighbouring Embedding algorithm (t-SNE (vdMaaten and Hinton 2008), ptSNE (Garriga and Bartumeus 2018));
- kernel density estimation (KDE (Terrell and Scott 1992)) over the embedding space, by means of the perplexity-adaptive KDE (paKDE (Garriga and Bartumeus 2018)) with adaptive bandwidth;
- cluster identification by means of the water-track transform (WTT (Garriga and Bartumeus 2018)), a variant of the watershed transform family of algorithms (Meyer 1994, Stoev and Straßer (2000), De Bock, De Smet, and Philips (2005)) that figures out the water tracks (*i.e* the riverbeds) along the valleys of the density landscape.

1.1 Package dependencies

The bigMap R-package is intended to be used in multi-core platforms. The most computationally expensive parts of the algorithm are implemented using shared memory and C++ implementations. Thus, important dependences of the *bigMap* package are:

- **Rcpp**, *Seamless R and C++ Integration* (Eddelbuettel and François 2011);
- **RcppArmadillo**, *Accelerating R with high-performance C++ linear algebra* (Eddelbuettel and Sanderson 2014);
- **bigmemory**, *Scalable Strategies for Computing with Massive Data* (Kane, Emerson, and Weston 2013);
- **parallel**, *Support for Parallel computation in R* (R Core Team 2018);
- **snow**, *Simple Network of Workstations* (Tierney et al. 2016).

Further dependencies of the package are:

- **colorspace**, (Zeileis, Hornik, and Murrell 2009);
- **RColorBrewer**, (Neuwirth 2014);

2 Using the package

```
library(bigMap)
```

The workflow of the package is structured around a central object (a common R's *list()*) that represents an instance of a mapping of a dataset. We name this object *bdm* from *big data mapping* and so we will do throughout this document. The names of the commands in the package also stem from this acronym.

A *bdm* is initialized with an element containing the input data. Afterwards, each down-stream step in the protocol attaches the output as a new element in the list, which, in turn, constitutes the input for the next step. Consequently, the protocol is build up of an **ordered sequence of steps** that must be preserved. Let's load the example included in the package,

```
bdm.example()
```

```
## Loading objects:
##   exMap
```

This example illustrates the structure of a *bdm* object after going through all the protocol,

```r
str(exMap)
```

```
## List of 9
##  $ dSet       : chr "exMap"
##  $ data       : num [1:5000, 1:4] 0.0681 1.6987 8.7789 2.4453 2.4284 ...
##  $ lbls       : num [1:5000] 1 5 12 4 8 6 3 6 1 3 ...
##  $ is.distance: logi FALSE
##  $ Xdata      :List of 2
##   ..$ whiten   : num 4
##   ..$ input.dim: int 4
##  $ Xbeta      :List of 4
##   ..$ B  : num [1:5000] 7.97 1.84 4.93 3.09 7.6 ...
##   ..$ ppx: num 200
##   ..$ itr: num 100
##   ..$ tol: num 1e-05
##  $ ptsne      :List of 8
##   ..$ threads: num 10
##   ..$ layers : num 2
##   ..$ rounds : int 2
##   ..$ boost  : num 1
##   ..$ alpha  : num 0.5
##   ..$ Y      : num [1:5000, 1:4] -13.14 23.75 10.89 3.67 27.48 ...
##   ..$ cost   : num [1:10, 1:141] 0.916 0.916 0.919 0.915 0.916 ...
##   ..$ size   : num [1:2, 1:141] 2.81 2.81 2.81 2.81 3.04 ...
##  $ pakde      :List of 2
##   ..$ :List of 6
##   .. ..$ ppx  : num 200
##   .. ..$ beta : num [1:5000] 0.3668 0.0282 0.4195 0.3033 0.5556 ...
##   .. ..$ g.exp: num 3
##   .. ..$ x    : num [1:200] -37.2 -36.9 -36.5 -36.2 -35.8 ...
##   .. ..$ y    : num [1:200] -33 -32.7 -32.4 -32.1 -31.7 ...
##   .. ..$ z    : num [1:200, 1:200] 4.46e-61 1.62e-60 1.10e-59 1.33e-58 1.88e-57 ...
##   ..$ :List of 6
##   .. ..$ ppx  : num 200
##   .. ..$ beta : num [1:5000] 0.346 0.028 0.504 0.224 0.477 ...
##   .. ..$ g.exp: num 3
##   .. ..$ x    : num [1:200] -37.5 -37.2 -36.8 -36.4 -36.1 ...
##   .. ..$ y    : num [1:200] -31.6 -31.3 -31 -30.7 -30.3 ...
##   .. ..$ z    : num [1:200, 1:200] 4.55e-55 1.30e-54 3.61e-54 9.95e-54 2.83e-53 ...
##  $ wtt        :List of 2
##   ..$ :List of 5
##   .. ..$ grid: num [1:2, 1:6] 200 200 -38 -34 34 ...
##   .. ..$ P   : num [1:16, 1] 35462 16620 25050 12712 8541 ...
##   .. ..$ M   : num [1:16, 1:2] 62 20 50 112 141 134 180 139 93 61 ...
##   .. ..$ C   : num [1:40000, 1] 10 10 10 10 10 10 10 10 10 10 ...
##   .. ..$ s   : int 16
##   ..$ :List of 5
##   .. ..$ grid: num [1:2, 1:6] 200 200 -38 -32 34 ...
##   .. ..$ P   : num [1:15, 1] 35462 7741 16220 24851 11912 ...
##   .. ..$ M   : num [1:15, 1:2] 62 141 20 51 112 134 139 180 94 61 ...
##   .. ..$ C   : num [1:40000, 1] 10 10 10 10 10 10 10 10 10 10 ...
##   .. ..$ s   : int 15
```

2.1 Mapping data

A mapping instance has several blocks that result from the four main commands of the package. Let's see how we build it.

2.1.1 Initialization: bdm.init()

Let's consider a matrix of raw data (here we take it from the *exMap* object itself). This is a small synthetic dataset with $n = 5000$ observations drawn from a 4-variate Gaussian mixture model with 16 Gaussian components,

```
X <- exMap$data      # let's assume this is our raw data matrix
dim(X)
```

```
## [1] 5000    4
```

A *bdm* is initialized through the *bdm.init()* command with, at least, an input dataset (either a *matrix* or a *data.frame*) and a name to identify the dataset (the utility of the dataset name is explained later in Section 3.1.2). So, in our case,

```
myMap <- bdm.init('myDataset', X)
str(myMap)
```

```
## List of 3
##  $ dSet       : chr "myDataset"
##  $ data       : num [1:5000, 1:4] 0.0681 1.6987 8.7789 2.4453 2.4284 ...
##  $ is.distance: logi FALSE
```

myMap is now a *bdm* instance with three basic elements: *myMap$dSet* a name identifying the dataset, a matrix *myMap$data* with the raw data, and a logical value *myMap$is.distance* (*FALSE* by default) indicating that the input matrix does not contain distances but raw observations. As the ptSNE algorithm works with distances among observations, we can forward a matrix of precomputed distances as well including the command's argument *is.distance* = *TRUE*,

```
myDistMap <- bdm.init('myDataset', X, is.distance = TRUE)      # don't run
```

In case (exceptional) of a supervised dataset, we can also supply a vector of labels. This is the case of our synthetic dataset (again we will take the labels from the *exMap* instance),

```
L <- exMap$lbls      # assume this is our vector of labels
```

Then, we would use the argument *labels* = *L* in the initialization function,

```
myMap <- bdm.init('myDataset', X, labels = L)
str(myMap)
```

```
## List of 4
##  $ dSet       : chr "myDataset"
##  $ data       : num [1:5000, 1:4] 0.0681 1.6987 8.7789 2.4453 2.4284 ...
##  $ lbls       : num [1:5000] 1 5 12 4 8 6 3 6 1 3 ...
##  $ is.distance: logi FALSE
```

In vector L, we may supply any external labelling that we may want to map in the landscape. Labels can be of whatever type that can be factorized. Labels are transformed by means of *as.numeric(as.factor())* into a vector of integer labels and attached afterwards as a new element *myMap$lbls*.

2.1.2 First step: bdm.ptsne()

bdm.ptsne() performs the dimensionality reduction by means of the ptSNE algorithm. The ptSNE runs several instances of the t-SNE on different chunks of the data (partial t-SNEs) using an alternating scheme of short runs and mixing of the partial solutions. The performance of the ptSNE algorithm is governed by the parameters described below. Further details can be found in (Garriga and Bartumeus 2018).

run and mixing scheme

Threads The number of partial t-SNEs to be run. The ptSNE splits the dataset into this number of elementary chunks. The larger the number of threads, the faster the computation of the final solution but the slower their convergence to a global solution. Note that the number of threads must not necessarily match the number of physical cores available. Indeed, it can be higher (what is known as multi-threading). Using multi-threading to run ptSNE on multiprocessor systems can yield significant reductions in the computational times.

Layers Each one of the threads above does not run a single chunk of data. Instead, the threads are chained cyclically such that each thread includes the number of chunks specified by *layers* sharing with another thread as many chunks of data as specified by *layers-1*. This chained structure with overlapping chunks of data creates a link between successive threads which facilitates convergence towards a global solution. As a consequence of this overlapping each data-point is taking part in several partial t-SNEs and thus, after pooling all partial solutions we have as many global solutions as *layers*. The relation *layers/threads* determines the thread-size $z = n\ layers/threads$ (where n is the dataset size). The t-SNE algorithm is of order quadratic with respect to the size of the dataset. By making $z \ll n$ we overcome the unsuitability of the t-SNE algorithm for large-scale datasets.

Epochs The run and mixing scheme is cyclical and each cycle is structured in three phases involving a master process and several worker processes (the threads): (i) the master process mixes the data and defines the data chunks; (ii) each worker runs a partial t-SNE with the chunk of data that it has been assigned; (ii) the master pools the partial solutions from the workers into a global solution. This sequence is called an *epoch*. The global solution of one *epoch* is used as the starting condition for the new *epoch*.

Rounds The number of epochs is set to $\sqrt{n}$ (where n is the dataset size), and the number of iterations per epoch (epoch length) is set to $\sqrt{z}$ (where z is the thread-size). Scaling the epoch length to the thread-size avoids getting too divergent solutions from each partial t-SNE. This predetermined setup, with $\sqrt{n}$ epochs and $\sqrt{z}$ iterations per epoch, is a *round*. In general, the algorithm reaches a stable solution in one single round. If not, the ptSNE can run some extra rounds to refine the mapping, although the improvement achieved is usually low with respect to the computational time required.

computing similarities

Perplexity This parameter sets a balance between finding global and local structure in the data (corresponding respectively to high and low values of perplexity). It does not exist any rule-of-thumb to set this value. In general, we will have to perform several runs testing different values. A good advice is to **start with a high value of perplexity (about half or a quarter of the thread.size) to first unveil the global structure and decrease it gradually in subsequent runs to refine the local structure** as long as the global structure is more or less preserved (depending on our requirements).

```
myMap <- bdm.ptsne(myMap, threads = 8, layers = 2, rounds = 2, ppx = 200)
```

```
+++ running 8 threads
+++ processing data
+++ Computing Betas, perplexity 200
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.2163  0.2974  0.3514  0.3745  0.4367  0.8412
+++ computing ptSNE
... threads 8, size 1250, epochs 142, iters 36
+++ epoch 0000/0142  9.1793e-01  1.9914e+00
--- round 01/02        <Cost>      <Size>      epSecs  time2End  eTime
+++ epoch 0001/0142  9.3225e-01  1.9914e+00    1.7064  00:04:01  11:39
+++ epoch 0002/0142  9.4205e-01  2.0900e+00    1.5366  00:03:47  11:39
+++ epoch 0003/0142  9.4732e-01  2.2707e+00    1.5169  00:03:41  11:39
+++ epoch 0004/0142  9.4656e-01  2.3904e+00    1.5226  00:03:37  11:39
+++ epoch 0005/0142  9.3815e-01  2.5407e+00    1.5370  00:03:34  11:38
+++ epoch 0006/0142  9.2341e-01  2.7160e+00    1.5481  00:03:32  11:38
+++ epoch 0007/0142  9.0513e-01  3.4047e+00    1.5327  00:03:30  11:38
... (output suppressed)
+++ epoch 0070/0142  6.7653e-01  4.1372e+01    1.5313  00:01:52  11:38
+++ epoch 0071/0142  6.7578e-01  4.1679e+01    1.5367  00:01:50  11:38
--- round 02/02        <Cost>      <Size>      epSecs  time2End  eTime
+++ epoch 0072/0142  6.7551e-01  4.1722e+01    1.5572  00:01:48  11:38
+++ epoch 0073/0142  6.7410e-01  4.1941e+01    1.5465  00:01:47  11:38
... (output suppressed)
+++ epoch 0141/0142  6.4650e-01  5.4606e+01    1.5350  00:00:02  11:38
+++ epoch 0142/0142  6.4643e-01  5.4385e+01    1.5537  00:00:00  11:38
... threads 8, size 1250, epochs 142, iters 36
+++ epoch 0142/0142  6.4643e-01  5.4385e+01    1.5499  00:03:40
```

Note that the first parameter is *myMap* itself and the output value is a new *bdm* instance. This is the general rule for most of the commands of the package: **the command's first parameter is a *bdm*** (a list) and **the value returned is a copy of the input *bdm* but including the new output** (a copy of the list with a new element attached where the output of the command is included). As we did in the example above, we can overwrite the input object with the output object, so that the resulting effect is a single list that keeps growing as we go ahead with the protocol.

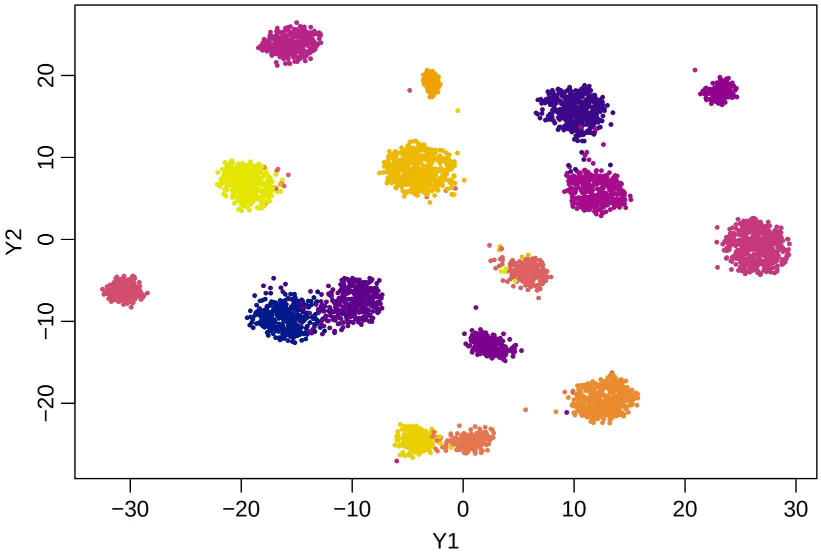
```
str(myMap$ptsne)
```

```
## List of 8
##  $ threads: num 10
##  $ layers : num 2
##  $ rounds : int 2
##  $ boost  : num 1
##  $ alpha  : num 0.5
##  $ Y      : num [1:5000, 1:4] -13.14 23.75 10.89 3.67 27.48 ...
##  $ cost   : num [1:10, 1:141] 0.916 0.916 0.919 0.915 0.916 ...
##  $ size   : num [1:2, 1:141] 2.81 2.81 2.81 2.81 3.04 ...
```

This command adds new elements to the list. The element *myMap$ptsne* includes a matrix *myMap$ptsne$Y*(5000, 4) with the mapping positions $\left(y_i^1, y_i^2\right)$. The positions of the mapped data-points are given by layers as succesive pairs of columns (*i.e.* columns 1 and 2 in *myMap$ptsne$Y* are the mapped positions for layer 1).

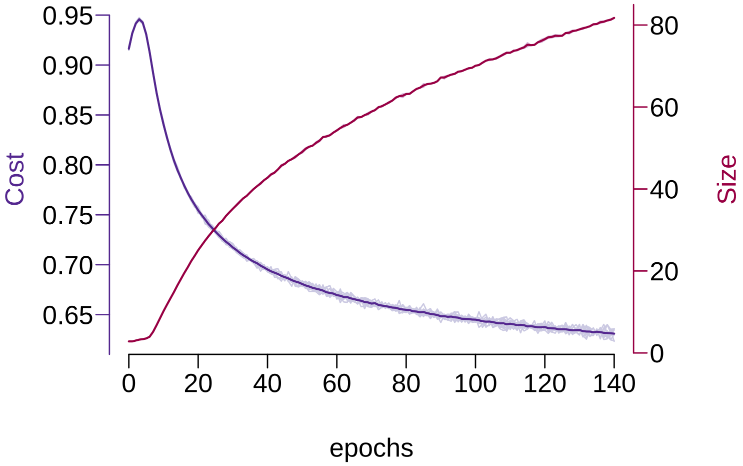
We can visualize the embedding with *bdm.plot()* (Section 2.2.2),

```
bdm.plot(myMap)
```



The elements *myMap$ptsne$cost* and *myMap$ptsne$size* include the values of the embedding cost and the embedding size functions (Garriga and Bartumeus 2018). These elements are matrices with one column per epoch. The rows in *myMap$ptsne$cost* are the embedding cost values per thread (each **partial** t-SNE). The rows in *myMap$ptsne$size* are the embedding size per layer (each **global** t-SNE). The command *bdm.cost()* (Section 2.2.1) plots the embedding cost and embedding size functions,

```
bdm.cost(myMap)
```



It is important to note that the *bdm.ptsne()* silently performs a complex sequence of operations (Garriga and Bartumeus 2018), therefore it has far more parameters than those shown in the example (though most of them work well with default values for the most of the cases):

1. Setting up the running (parallel) environment (*threads = 4*, *type = 'SOCK'*, *layers = 2*, *boost = 2*, *rounds = 2*):

   - *threads* sets the number of partial t-SNEs that will be simultaneously running, each one with a subset of data of size z (the thread size) that results from the combination of *threads* and *layers*, given by $n\,layers/threads$.
   - *type* sets the type of parallelization, intra-node parallelization (*type = 'SOCK'*), or inter-node parallelization (*type = 'MPI'*). This is related with the *makeCluster()* command of the *snow* R-package (Tierney et al. 2016). To use inter-node parallelization a version of the *message passing interface* (MPI) must be up and running in the system (see section 3.2.2).
   - *rounds* sets the number of rounds.
   - *boost* changes the default run-and-mixing scheme. The number of epochs is set to $1/\textit{boost}\,\sqrt{n}$ and the number of iterations per epoch to $\textit{boost}\,\sqrt{z}$. The total number of iterations remains as $\sqrt{n\,z}$ but the ptSNE running time is reduced in as much as the number of inter-epoch phases is reduced (specially when using MPI parallelization). The counterpart is that convergence might be compromised if $\textit{boost} \gg 1$.

2. Pre-processing of the raw data (*whiten = 4*, *input.dim = 30*):

   - By default the command performs a whitening of the raw data (*i.e.* computing the principal components and dividing them by the eigen values to equalize the variances).If *whiten = 3*, only PCA is performed (with no whitening). In both cases, at most, a subset of the first *input.dim* components is used. If *whiten = 2* raw data is only centered and scaled. If *whiten = 1* raw data is only centered. If *whiten = 0* no preprocessing of raw data is performed at all. With *whiten < 3*, all dimensions are considered (*input.dim* has no effect).

3. Computing the variances of the input data kernels (*ppx = 100*, *itr = 100*, *tol = 1e-05*):

   - *ppx* sets the value of *perplexity* used to compute the input similarities (as explained above)
   - *itr* and *tol* refer to the precision in the computation of the variances and in general work well with default values.

4. Running the partial t-SNEs (*alpha = 0.5*, *Y.init = NULL*):

   - *alpha* sets the momentum in the gradient function. The momentum is a fraction *alpha* of the position gradient computed in the last iteration that is added to the position gradient computed in the current iteration. In general, the momentum works well with the default value 0.5.
   - *Y.init* allows to start the embedding with a given initial mapping (by default, the starting positions of the embedding are randomly initialized). If we deem that a ptSNE run has not yet reach a stable solution, we can use the output of this run to initialize the embedding of a new run and save computational time.

5. Showing intermediate information (*info = 1*):

   - *info* sets the level of intermediate information. At the lowest level (*info = 0*), the algorithm shows the values of the cost and size of the embedding along with an estimation of the remaining time at the end of each epoch. With *info = 1* (default value) the algorithm saves the current state of the embedding as an *.RData* file at the end of each round (Section 3.1.2). Thus we can forward some insights about a long ptSNE run. For instance, we can detect that the current solution is already stable or that the parameterization is not appropriate.

6. Close the parallel environment (no parameters).

2.1.3 Second step: bdm.pakde()

bdm.pakde() computes a density estimation over the embedding area based on the paKDE algorithm.

```
myMap <- bdm.pakde(myMap, threads = 4, ppx = 200, g.exp = 2)
```

```
+++ running 4 threads
+++ paKDE for layer 1/2 +++
+++ Computing Betas, perplexity 200
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
 0.7883  1.0309  1.2018  1.3939  1.4282  3.9641 
 computing grid cell densities ...
+++ cdf 0.9997
```

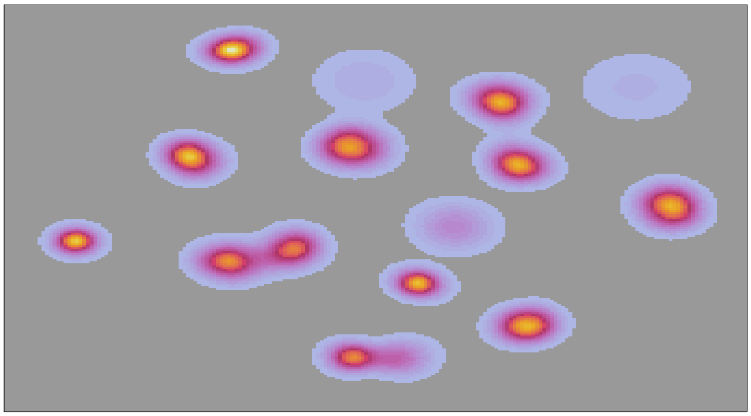
The output is attached to the list as a new element named *myMap$pakde*. This element is itself a list of lists (with one element per layer). By default, the command computes the paKDE for layer one, which is attached to *myMap$pakde[[1]]*. The argument *layer = l* allows computing the paKDE for any other layer, and the output, in this case, is attached to *myMap$pakde[[l]]*.

```
str(myMap$pakde)
```

```
## List of 1
##  $ :List of 6
##   ..$ ppx  : num 200
##   ..$ beta : num [1:5000] 0.3668 0.0282 0.4195 0.3033 0.5556 ...
##   ..$ g.exp: num 3
##   ..$ x    : num [1:200] -37.2 -36.9 -36.5 -36.2 -35.8 ...
##   ..$ y    : num [1:200] -33 -32.7 -32.4 -32.1 -31.7 ...
##   ..$ z    : num [1:200, 1:200] 4.46e-61 1.62e-60 1.10e-59 1.33e-58 1.88e-57 ...
```

The element *myMap$pakde[[1]]* is a raster over the embedding area. The coordinates of the raster are saved in *myMap$pakde[[1]]$x* and *myMap$pakde[[1]]$y*, and the density values are stored in *myMap$pakde[[1]]$z*. We can visualize a heat-map of the density estimation with,

```
bdm.plot(myMap) # by default, bdm.plot() shows the output of the last step,
```



The *bdm.pakde()* command performs a sequence of operations (Garriga and Bartumeus 2018) with corresponding arguments:

1. Setting up the running environment (*threads = 2*, *type = ‘SOCK’*):

    - *threads* and *type* specify the number of threads that will be spawned and the type of parallelization used, intra-node parallelization (*type = ‘SOCK’*), or inter-node parallelization (*type = ‘MPI’*), (as explained for the *bdm.ptsne()* command). In this case, multi-threading does not help to increase efficiency as this algorithmic process is of $\mathcal{O}(n)$, so we would typically specify as many threads as the number of physical cores available.

2. Computing the variances of the local kernels of the mapped data-points (*ppx = 100*, *itr = 100*, *tol = 1e-05*):

    - the meaning of these parameters is exactly as explained for the *bdm.ptsne()* command, though in this case the perplexity refers to the computation of similarities in the low dimensional space (the mapped data-points). **A good advice is to start with the same value of perplexity used in the high dimensional space (*i.e.* assume a similar degree of similarity in both, high and low dimensional spaces) and lower it gradually if the density function does not look reasonably tight to the shape of the mapping.**

3. Computing the density estimation (*g = 200*, *g.exp = 4*):

    - *g* defines the grid size (the length of the raster side in number of cells). The embedding tends to be almost circular in general, thus we always consider a square grid of *g* times *g* cells.
    - *g.exp* is a factor to expand the limits of the embedding size expressed in times σ. By default (*g.exp = 4*) the expansion is such to include 4 times σ of the most extrem kernels (minimum and maximum in both directions), that is 0.9999367 of the probability mass of the kernel. In the example shown above, we set *g.exp = 2* thus the cumulated density function does not add up to one (note the last line in the command's output, +++ cdf 0.9997375). **If the input dataset includes a few outliers, those data-points will likely be mapped far away from the rest. This situation will force an unnecessary expansion of the limits and will cause a distortion of the density estimation.** We can overcome this using lower values of *g.exp*. With *g.exp = 0* the limits of the grid will match those of the embedding.

4. Close the parallel environment (no parameters).

2.1.4 Third step: bdm.wtt()

bdm.wtt() identifies the clusters by means of the WTT algorithm (Garriga and Bartumeus 2018). This algorithm figures out the water tracks (i.e. river beds) in the valleys of the density landscape. Clusters are numbered following the height of the defining peaks. *bdm.wtt()* is a very fast command and has no parameters.

```r
myMap <- bdm.wtt(myMap)
```

```
+++ WTT for layer 1/2 +++
clusters: 16
```

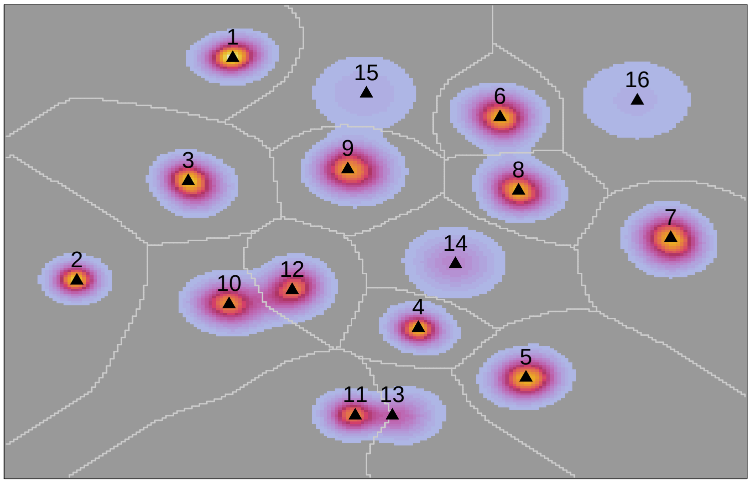
This command attaches a new element *myMap$wtt* to the list. This element is itself a list of lists (with an element per layer). By default, the *bdm.wtt()* command computes only the WTT for the first layer which is attached to *myMap$wtt[[1]]*. Using *layer = l* allows computing the WTT for any other layer. In this case the output is attached to *myMap$wtt[[l]]*.

```r
str(myMap$wtt)
```

```
## List of 1
##  $ :List of 5
##   ..$ grid: num [1:2, 1:6] 200 200 -38 -34 34 ...
##   ..$ P   : num [1:16, 1] 35462 16620 25050 12712 8541 ...
##   ..$ M   : num [1:16, 1:2] 62 20 50 112 141 134 180 139 93 61 ...
##   ..$ C   : num [1:40000, 1] 10 10 10 10 10 10 10 10 10 10 ...
##   ..$ s   : int 16
```

At this stage the lines that delimit the clusters are added to the density heat-map image.

```
bdm.plot(myMap)
```



2.1.5 bdm.labels()

The *bdm.wtt()* identifies the clustering at grid cell levels (Section 2.1.4). The element *myMap$wtt[[1]]$C* is a numeric vector with **clustering labels at grid cell level**. To get a vector of **data-point clustering labels** we use the *bdm.labels()* function,

```
myDataLabels <- bdm.labels(myMap)
```

This is a very fast function. By default the function computes the data-point labels from layer 1. We can get the vector of labels for any other layer using the argument *layer*,

```
myDataLabels2 <- bdm.labels(myMap, layer = 2)
```

2.2 Visual assessment of the output.

2.2.1 bdm.cost()

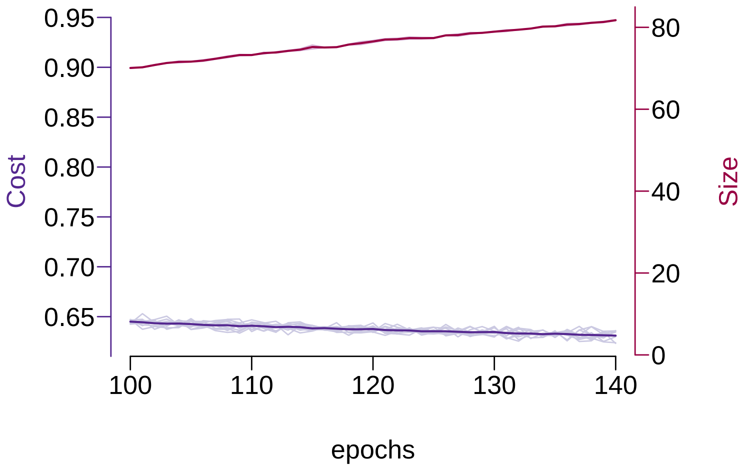
bdm.cost() depicts the embedding cost and the embedding size as a function of the epochs (Garriga and Bartumeus 2018):

- The cost of the embedding evaluates how well the low dimensional mapping matches the distances between data-points in the high dimensional space. This is a normalized value such that the lower it is the cost the better it is the matching.

- The size of the embedding is the measure of the diagonal of the embedding area which keeps growing as the mapping improves.

bdm.cost() has an *offset* argument (NULL by default) to better asses the last part of the curves,

```
bdm.cost(myMap, offset = 100)
```



The shadowed area (scarcely visible in this case because of the strong convergence) around the solid blue line depicts the embedding cost values by thread (each partial t-SNE). The solid blue line depicts the average cost. The shadowed area (not visible in this case because of the strong convergence) around the solid red line depicts the embedding size values by layers (each global t-SNE). The solid red line depicts the average size.

Both functions together assess the validity of the solution: **if both achieve a stable value then the solution is stable** and the mapping will hardly improve for longer we keep iterating. Nonetheless, it is possible that the size function keeps growing even though the cost function has reached a stable solution. This occurs when the relative position of the mapped data-points is fairly correct but the embedding still needs to grow to fairly represent the highest dissimilarities. In terms of clustering **this is not significant unless we are interested in the quantitative semantics of the density estimation**. If this is the case, we would be better off by letting the algorithm iterate until the size function reaches also a stable value.

bdm.cost() admits a list of *bdms* as input as well yielding a multi-plot to compare the output of different parameterizations. Note that higher values of perplexity will inevitably lead to higher stabilization cost values (Garriga and Bartumeus 2018).

2.2.2 bdm.plot()

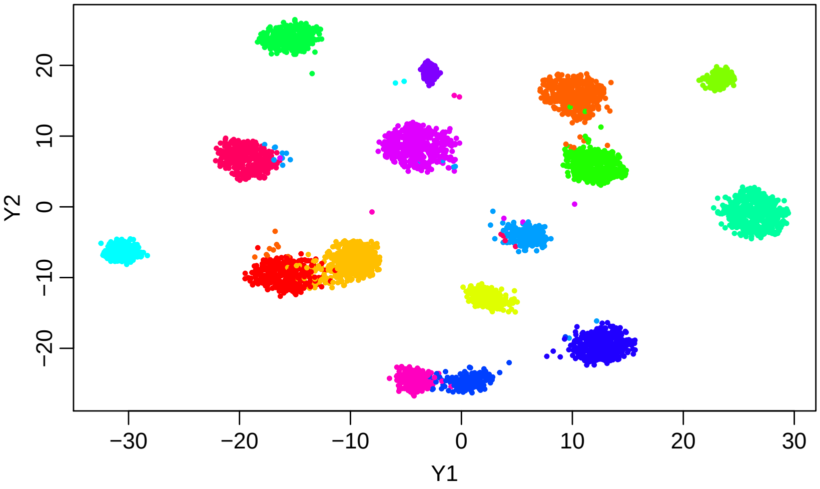
bdm.plot() is a general command to depict the output after each step in the mapping protocol, plotting by default the output of the last step computed. Please, refer to the package help to see some step specific arguments to minimally tweak the plots (*e.g* colour, size). We just note the following:

1. In case that *$lbls* is not *NULL* the function will use these labels to colour the data-points in the *bdm.ptsne()* output plot. If not, the function will use the clustering labels (from *bdm.labels()*).

2. Once the *bdm.pakde()* has been performed, we can still plot the *bdm.ptsne()* output setting the parameter *ptsne = TRUE*.

3. An example of the *bdm.ptsne()* output plot with some tweaking to increase the size of the data-points and using a different colour palette (see the package help for further information),

```
bdm.plot(myMap, ptsne = TRUE, ptsne.cex = 0.6, ptsne.pltt = rainbow(16), layer = 2)
```

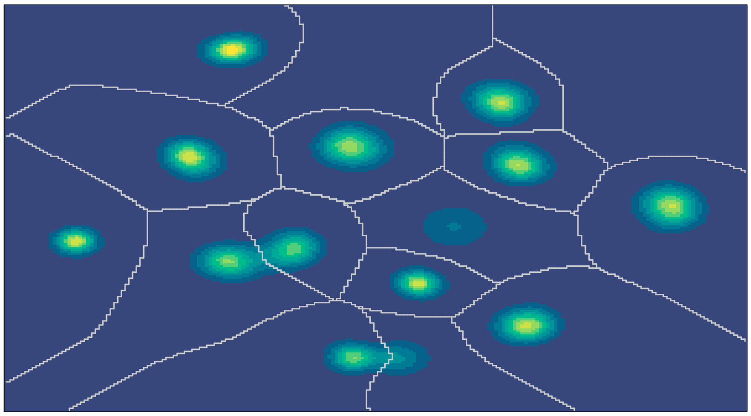


4. After *bdm.pakde()* or *bdm.wtt()* we get a heat-map image of the density function, in the latter case including the water-track lines that delimit the clusters and the labels assigned to the clusters.

5. An example with some tweaking to change the colour palette and the number of levels of the heat-map and hiding the labels of the clusters (see the package help for further information),

```
mypalette <- colorspace::heat_hcl(12, h = c(300, 75), c. = c(35, 95), l = c(15, 90),
    power = c(0.8, + 1.2), fixup = TRUE, gamma = NULL, alpha = 1)[3:12]
```

```
bdm.plot(myMap, pakde.pltt = mypalette, pakde.lvls = 10, plot.peaks = FALSE)
```



2.2.3 bdm.qMap()

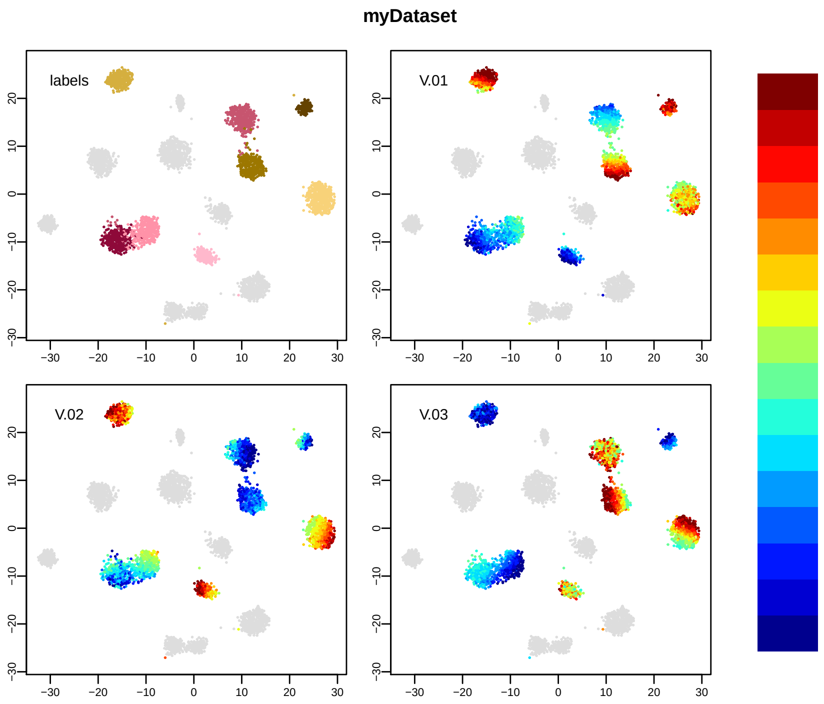
bdm.qMap() maps numeric variables onto the embedding, depicting its value as a quantile-map. A quantile-map is similar to a heat-map. The difference is that in a quantile-map colours indicate quantiles instead of ranges. In this way we always get a clear depiction of the distribution of the variable throughout the embedding. By plotting the input variables we can detect relevant/irrelevant features in determining the clusters.

bdm.qMap() generates a multi-plot layout, with the first plot depicting the embedding (as with *bdm.plot(. . . , ptsne = TRUE)*) and the rest depicting the mapping of one variable each (with a maximum of 15 variables).

By defaut, *bdm.qMap()* plots the input variables (the first 15 at most). In case we have more than 15 input variables, the argument *data* allows passing a matrix with a selection of the variables that we may want to map. This matrix can also include any **numeric** covariate of interest.

Using the argument *subset* we can restrict the mapping of the given variables to a subset of the input data, given as a vector of row indexes of the input data matrix. The data-points that are not in *subset* are shown in grey. This is useful to highlight the distribution of the given variables in relation to some **discrete** covariate of interest.

```r
bdm.qMap(myMap, data = myMap$data[, 2:4], subset = which(myMap$lbls %in% 1:8))
```



2.2.4 bdm.dMap()

bdm.dMap() visualizes the distribution of a **discrete** covariate (or class variable) throughout the embedding. In particular, *bdm.dMap()* computes the join distribution $P\left(V = v_i, C = c_j\right)$ where $V = \{v_1, \ldots, v_l\}$ is the class/covariate and $C = \{c_1, \ldots, c_g\}$ are the grid cells of the paKDE raster. The result is a fuzzy distribution of the class/covariate at each cell.

Computing the join distribution $P(V = v_i, C = c_j)$ entails an intensive computation. Thus *bdm.dMap()* performs the computation and stores the result in a dedicated element named *$dMap*.

Afterwards the density maps can be visualized with the *bdm.dMap.plot()* function. The *bdm.dMap.plot()* yields a multi-plot layout where the first plot shows the dominating value of the covariate (or dominating class) in each cell, and the rest of the plots show the density map of each covariate value (or class) (see details in the package help). The multi-plot layout can be limited to a subset of the values of the covariate (or subset of classes).

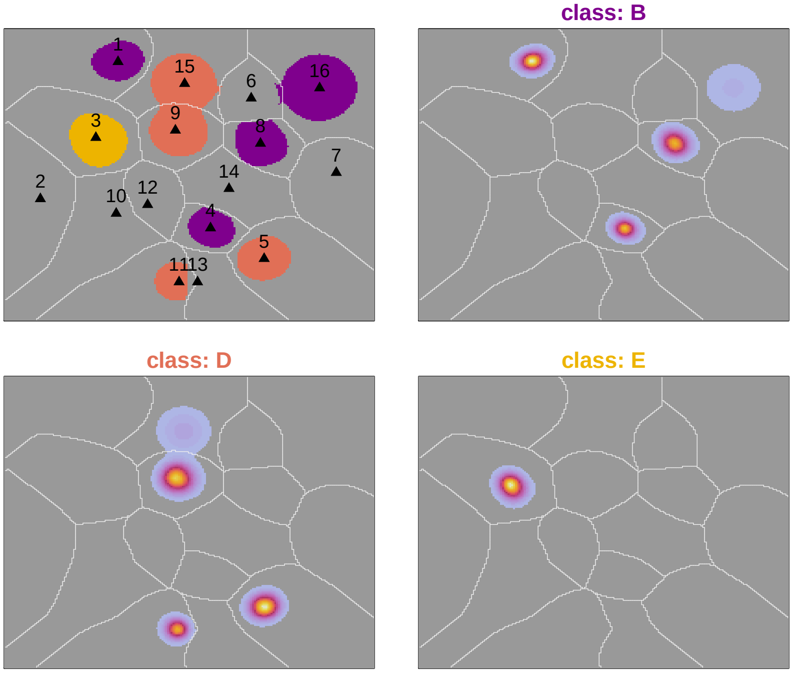
```r
# assume these are our class covariates
myclasses <- c('A', 'B', 'C', 'D', 'E')[(myMap$lbls %/% 4) +1]
```

```r
myMap <- bdm.dMap(myMap, threads = 4, labels = myclasses)
```

```
+++ running 4 threads
+++ cdf 0.9997
```

We can now plot the class density maps,

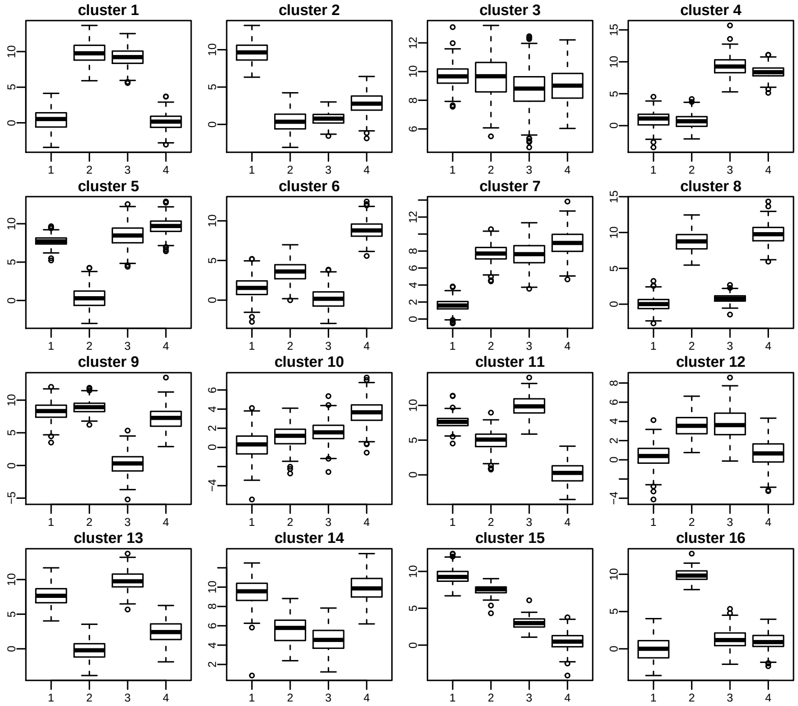
```r
bdm.dMap.plot(myMap, classes = c('B', 'D', 'E'))
```



2.2.5 bdm.boxp()

bdm.boxp() depicts a multi-plot layout with a box-plot with input variable statistics per cluster (for the first 25 clusters as a maximum). By default box-plots are grouped by cluster. Using *byVars = TRUE* box-plots are grouped by input feature.

```
bdm.boxp(myMap)
```



3 Working with remote multiprocessor systems

3.1 Easy managing of ptSNE runs.

The *bigMap* mapping protocol entails a computationally intensive part (*i.e.* ptSNE, pKDE and WTT) and a posterior visualization to analyze the results. The former imply strong hardware requirements and is likely to be run in a mutiprocessor-system (*e.g.* a cluster of multiple inter-connected nodes with several physical cores each). The latter requires easy graphical interactivity, which is not typically available when working with remote multi-processor systems. Therefore, the *bigMap* package usually involves a cyclic process of submitting a job to the remote system, pooling the result to the local machine, draw some conclusion and start again with different parameters, until getting to a satisfactory result. In order to ease this task, the *bigMap* package includes some simple functions to systematically save the results (Functions 3.1.1, 3.1.2, 3.1.3). Also, users working in a local machine with a *secure file transfer* client (scp) can set up the *bigMap* environment so that the results are automatically transferred to the user's local machine (Functions 3.1.4, 3.1.5).

3.1.1 bdm.mybdm()

This function sets/gets a default path to save results,

```
bdm.mybdm('~/myPath/')
```

```
## [1] "~/myPath/"
```

3.1.2 bdm.fName()

This function returns a default file name to save a *bdm* object (an *.RData file). The file name stems from the dataset name stored in *$dSet* (Section 2.1.1) and keeps track of the main parameters used for the mapping,

```
bdm.fName(myMap)
```

```
## [1] "~/myPath/myDataset_z10_l2_b1_r2_p200_k16.RData"
```

The name includes the number of threads (*_zxx*), the number of layers (*_lxx*), the boost factor (*_bxx*), the number of rounds (*_rxx*), the perplexity used for the ptSNE (*_pxxx*) and the number of clusters found (*_kxxx*).

3.1.3 bdm.save()

This function saves a *bdm* object as an *.RData file with default file name (as given by *bdm.fName()*) and default path (as given by *bdm.mybdm()*),

```
bdm.save(myMap)
```

```
## +++ saved to ~/myPath/myDataset_z10_l2_b1_r2_p200_k16.RData
```

3.1.4 bdm.local()

This function sets/gets the IP address of the local machine to which we would like the remote system to send the results,

```
bdm.local('xxx.xxx.xxx.xxx')
```

```
## [1] "xxx.xxx.xxx.xxx"
```

3.1.5 bdm.scp()

This function saves and transfers a *bdm* object. The object is saved in the remote system with default path and default file name, and is transfered to the local machine and saved again with default path and default file name. **The necessary condition is to keep the same path structure in both machines**. A further condition is to have ssh private key authentication access.

```
bdm.scp(myMap)
```

```
## +++ saved to ~/myPath/myDataset_z10_l2_b1_r2_p200_k16.RData
```

3.2 Summary script

The *bigMap* mapping process can be summarized in a template of a simple R script (bdmScript.R),

```
library(bigMap)
# set bigMap environment
bdm.local('xxx.xxx.xxx.xxx')
bdm.mybdm('~/myPath/')
# load dataset
X <- read.csv('~/myPath/myDataSet.csv')
# mapping protocol
```

```
myMap <- bdm.init('myDataSet', X)
myMap <- bdm.ptsne(myMap, threads = 64, type = 'SOCK', layers = 2, boost = 2,
    rounds = 4, whiten = TRUE, input.dim = 10, ppx = 300, progress = 1)
myMap <- bdm.pakde(myMap, threads = 16, type = 'SOCK', ppx = 300)
myMap <- bdm.wtt(myMap)
# save and transfer to local machine
bdm.scp(myMap)
```

Let's assume we have a dataset in file *~/myPath/myDataSet.csv* with $n = 64000$ rows (and whatever number m of dimensions). This *bdmScript.R* will run 64 processes (partial t-SNEs) with a thread-size of $z = 64000/64 * 2 = 2000$. The raw data X will be preprocessed performing a PCA and a whitening and only the first ten components (assuming $m > 10$) will be used as input data. The ptSNE algorithm will perform 4 rounds of $\sqrt{64000} = 253$ epochs per round and $\sqrt{2000} = 45$ iterations per epoch. As we set *progress = 1*, the program will save a *bdm* object with the current embedding, in folder '~/myPath/' and file name 'myDataSet_z64_l03_r01_p300.RData' after the first round, 'myDataSet_z64_l03_r02_p300.RData' after the second round, and so on. At the same time, the intermediate results are transferred to the local machine with IP address xxx.xxx.xxx.xxx. After the last round, the program will run the pKDE and the WTT. Finally, it will save a *bdm* object with the mapping protocol completed, with file name 'myDataSet_z64_l03_r02_p300_kss.RData' (where *ss* stands for the number of clusters found). This file will also be transferred and saved to the local machine (assuming that the path '~/myPath/' exists in the local machine).

Typically, the *bdmScript.R* would be submitted as a batch job to a multiprocessor-system via *qsub* or alike (depending on the platform), specifying the type of parallelization interface to be used. **Submitting jobs to a multiprocessor-system is quite platform dependent.**

3.2.1 Intra-node (socket) parallelization

When using intra-node (or *socket*) parallelization all worker processes (64 threads in our example) are spawned within the same single node where the master process is submitted. With intra-node parallelization memory access is super fast but computation power is limited to the number of physical cores available in that node (usually 16, 20 or 64 at the best). For the case of our example (with 64 threads), submitting the job to a node with 16 cores overloads each core with 4 processes, *i.e* each core will have to run 4 threads at the same time. This is known as multi-threading and obviously results in longer running times in comparison with single-threading (1 thread per core). Nonetheless, because of the quadratic complexity of the t-SNE algorithm, multi-threading is in general a good choice, that is, **given a fixed number of physical cores, the higher it is the number of threads, the lower it is the thread-size accounting for significant savings in computational time** (Garriga and Bartumeus 2018).

In the following, we show how we would submit our example script using *SOCK* parallelization (**this example is for illustration purposes only and may work differently in many platforms**):

1. We (typically) must set some path environment variables to point to the programs we need to run (**this is platform dependent**),

```
# point to R
~$ export PATH=/home/soft/R-3.5.1/bin:$PATH
```

2. We submit the job with,

```
~$ qsub -pe make 16 -l h_vmem=8G R -q --vanilla -f bdmScript.R
```

This command submits the job to a node with 16 cores with 8GB of memory each and calls R to run the *bdmScript.R*. Note that the argument *-pe make 16* specifies the number of physical cores that we want available in our socket parallel environment, regardless of the number of threads that this set of cores will be running simultaneously (64 in our case).

3.2.2 Inter-node (MPI) parallelization

The use of inter-node (or *message passing interface*, MPI) parallelization is somewhat more contrived and subject to the existence of a low-latency/high-bandwidth system network (*e.g.* InfiniBand) and a high performance MPI messaging module (*e.g.* ompi, mpich2) installed on the system. When using MPI parallelization the worker processes are spawned through all the nodes of the cluster, independently of the node where the master process resides, and sharing data between node implies message passing through the network. Thus, memory access is much slower than in the case of socket parallelization. The counterpart is that we can (potentially) make use of as many physical cores as we have in the cluster. In other words, if we need to spawn 200 threads to map a large dataset we can set up an MPI parallel environment with 200 physical cores to bind one core per thread.

Using MPI we must make some major changes into our *bdmScript.R* (**this example is for illutration purposes only and may work differently in many platforms**):

1. For technical reasons, **an MPI cluster can be started only once per job**. This means that we must split the *bdmScript.R* into two parts:

   - Part 1 (*bdmScript1.R*)

   ```
   library(bigMap)
   # set bigMap environment
   bdm.local('xxx.xxx.xxx.xxx')
   bdm.mybdm('~/myPath/')
   # load dataset
   X <- read.csv('~/myPath/myDataSet.csv')
   # mapping protocol (step 1: ptSNE)
   myMap <- bdm.init('myDataSet', X)
   myMap <- bdm.ptsne(myMap, threads = 64, type = 'MPI', layers = 2, boost = 2,
       rounds = 4, whiten = TRUE, input.dim = 10, ppx = 300, progress = 1)
   # save and transfer to local machine
   bdm.scp(myMap)
   ```

   - Part 2 (*bdmScript2.R*)

   ```
   library(bigMap)
   # set bigMap environment
   bdm.local('xxx.xxx.xxx.xxx')
   bdm.mybdm('~/myPath/')
   # load previously saved result
   myMap <- load('~/myPath/myDataSet.RData')
   # mapping protocol (steps 2 and 3: paKDE, WTT)
   myMap <- bdm.pakde(myMap, threads = 16, type = 'MPI', ppx = 300)
   myMap <- bdm.wtt(myMap)
   # save and transfer to local machine
   bdm.scp(myMap)
   ```

2. Note that we must **change the argument *type = 'SOCK'* to *type = 'MPI'* in both commands, *bdm.ptsne()* and *bdm.pakde()***.

3. To submit the jobs we must write a job-script file. We suggest to use a job-script template, namely *snow.sh*, that we can use to submit any R script based on the *snow* R-package. This is our *snow.sh* job-script template (**this is platform dependent**),

```
#!/bin/bash
# execute as:
# qsub -l h_vmem=XG -pe ompi np snow path_to_myRscript/myRscript.R
# job name
#$ -N snowjob
# Use current working directory
#$ -cwd
# Set path environment variables
# MPI
export PATH=/home/soft/openmpi-2.1.5/bin:$PATH
export LD_LIBRARY_PATH=/home/soft/openmpi-2.1.5/lib:$LD_LIBRARY_PATH
# R
export PATH=/home/soft/R-3.5.1/bin:$PATH
# snow R-package
export PATH=/home/soft/R-3.5.1/lib64/R/library/snow:$PATH
# Execute
mpirun -np $NSLOTS RMPISNOW --no-save -q -f $1
```

3. We submit the first job with,

```
~$ qsub -pe ompi 65 snow.sh bdmScript1.R
```

4. Once the first job is finished, we submit the second job with,

```
~$ qsub -pe ompi 65 snow.sh bdmScript2.R
```

The commands in 3 and 4 create an MPI parallel environment with 65 physical cores and run the RMPISNOW script (from R-package **snow**) at each one. The RMPISNOW script is responsible to start an R environment at each core. One of them is configured as the master process and runs the bdmScript.R and the rest (64) are configured as worker processes to run one thread each one.

References

De Bock, Johan, Patrick De Smet, and Wilfried Philips. 2005. "A Fast Sequential Rainfalling Watershed Segmentation Algorithm." In *Advanced Concepts for Intelligent Vision Systems*, edited by Jacques Blanc-Talon, Wilfried Philips, Dan Popescu, and Paul Scheunders, 476–82. Berlin, Heidelberg: Springer Berlin Heidelberg.

Eddelbuettel, Dirk, and Romain François. 2011. "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software* 40 (8): 1–18. doi:10.18637/jss.v040.i08.

Eddelbuettel, Dirk, and Conrad Sanderson. 2014. "RcppArmadillo: Accelerating R with High-Performance C++ Linear Algebra." *Computational Statistics and Data Analysis* 71 (March): 1054–63. http://dx.doi.org/10.1016/j.csda.2013.02.005.

Garriga, Joan, and Frederic Bartumeus. 2018. "BigMap: Big Data Mapping with Parallelized t-SNE." *Journal of Statistical Software* 14 1: 015002.

Kane, Michael J., John Emerson, and Stephen Weston. 2013. "Scalable Strategies for Computing with

Massive Data.” *Journal of Statistical Software* 55 (14): 1–19. http://www.jstatsoft.org/v55/i14/.

Meyer, Fernand. 1994. “Topographic Distance and Watershed Lines.” *Signal Process.* 38 (1). Amsterdam, The Netherlands, The Netherlands: Elsevier North-Holland, Inc.: 113–25. doi:10.1016/0165-1684(94)90060-4.

Neuwirth, Erich. 2014. *RColorBrewer: ColorBrewer Palettes*. https://CRAN.R-project.org/package=RColorBrewer.

R Core Team. 2018. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Stoev, Stanislav L., and Wolfgang Straßer. 2000. “Extracting Regions of Interest Applying a Local Watershed Transformation.” In *Proceedings of the Conference on Visualization ’00*, 21–28. VIS ’00. Los Alamitos, CA, USA: IEEE Computer Society Press. http://dl.acm.org/citation.cfm?id=375213.375214.

Terrell, George R., and David W. Scott. 1992. “Variable Kernel Density Estimation.” *Ann. Statist.* 20 (3). The Institute of Mathematical Statistics: 1236–65. doi:10.1214/aos/1176348768.

Tierney, Luke, A. J. Rossini, Na Li, and H. Sevcikova. 2016. *Snow: Simple Network of Workstations*. https://CRAN.R-project.org/package=snow.

Todd, Jeremy G, Jamey S. Kain, and Benjamin Lovegren de Bivort. 2017. “Systematic Exploration of Unsupervised Methods for Mapping Behavior.” *Physical Biology* 14 1: 015002.

vdMaaten, Laurens, and G.E. Hinton. 2008. “Visualizing High-Dimensional Data Using T-Sne.” *Journal of Machine Learning Research* 9: 2579–2605.

Zeileis, Achim, Kurt Hornik, and Paul Murrell. 2009. “Escaping RGBland: Selecting Colors for Statistical Graphics.” *Computational Statistics & Data Analysis* 53 (9): 3259–70. doi:10.1016/j.csda.2008.11.033.