

The bigMap R-package: quick reference

J.Garriga; F.Bartumeus

ICREA Movement Ecology Laboratory (CEAB-CSIC)

ggarriga@ceab.csic.es

Version 2.3.1, nov 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	4
2.1.1	Initialization: <code>bdm.init()</code>	4
2.1.2	First step: <code>bdm.ptsne()</code>	5
2.1.3	Second step: <code>bdm.pakde()</code>	9
2.1.4	Third step: <code>bdm.wtt()</code>	10
2.1.5	<code>bdm.labels()</code>	11
2.2	Visual assessment of the output.	11
2.2.1	<code>bdm.cost()</code>	11
2.2.2	plotting functions	12
2.2.3	<code>bdm.qMap()</code>	13
2.2.4	<code>bdm.dMap()</code>	14
2.2.5	<code>bdm.boxp()</code>	15
3	Post-processing	15
3.1	Spatial merging of clusters	15
3.1.1	<code>bdm.optk.s2nr()</code>	16
3.1.2	<code>bdm.merge.s2nr()</code>	17
4	Working with remote multiprocessor systems	18
4.1	Easy managing of ptSNE runs.	18
4.1.1	<code>bdm.mybdm()</code>	18
4.1.2	<code>bdm.fName()</code>	18
4.1.3	<code>bdm.save()</code>	18
4.1.4	<code>bdm.local()</code>	18
4.1.5	<code>bdm.scp()</code>	19
4.2	Summary script	19
4.2.1	Intra-node (socket) parallelization	19
4.2.2	Inter-node (MPI) parallelization	20
	References	22

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: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:2000, @DeBock: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 Sander-son 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. The mapping protocol is build up of an **ordered sequence of steps** that must be preserved: each step attaches the output as a new element in the list, and the new list constitutes the input for the next step. 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,

```
str(exMap)
```

```
## List of 9
## $ dSet      : chr "exMap"
## $ data      : num [1:1000, 1:4] 9.23 8.39 10.13 1.84 4.33 ...
## $ lbls      : num [1:1000] 16 12 16 2 2 4 7 3 15 7 ...
## $ is.distance: logi FALSE
## $ Xdata     :List of 2
## ..$ whiten  : num 4
## ..$ input.dim: int 4
## $ Xbeta     :List of 4
## ..$ B       : num [1:1000] 8.16 5.23 5.27 10.04 6.6 ...
## ..$ ppx     : num 25
## ..$ itr     : num 100
## ..$ tol     : num 1e-05
## $ ptsne     :List of 8
## ..$ threads: num 4
## ..$ layers  : num 2
## ..$ rounds  : int 2
## ..$ boost   : num 1
## ..$ alpha   : num 0.5
## ..$ Y       : num [1:1000, 1:4] -11.3 9.76 -13.31 6.32 2.81 ...
## ..$ cost    : num [1:4, 1:63] 0.907 0.907 0.906 0.905 0.929 ...
## ..$ size    : num [1:2, 1:63] 2.8 2.81 2.8 2.81 2.99 ...
## $ pakde     :List of 1
## ..$ :List of 6
## .. ..$ ppx  : num 50
## .. ..$ beta : num [1:1000] 0.0828 0.113 0.0981 0.151 0.1067 ...
## .. ..$ g.exp: num 2
## .. ..$ x    : num [1:200] -37.7 -37.3 -37 -36.6 -36.3 ...
## .. ..$ y    : num [1:200] -37.6 -37.2 -36.8 -36.4 -36 ...
## .. ..$ z    : num [1:200, 1:200] 8.99e-10 1.15e-09 1.47e-09 1.87e-09 2.36e-09 ...
## $ wtt       :List of 1
## ..$ :List of 5
## .. ..$ grid: num [1:2, 1:6] 200 200 -38 -38 35 ...
## .. ..$ P    : num [1:16, 1] 17351 9715 23777 27739 26090 ...
## .. ..$ M    : num [1:16, 1:2] 151 115 177 139 90 90 110 59 70 138 ...
## .. ..$ C    : num [1:40000, 1] 15 15 15 15 15 15 15 15 15 ...
## .. ..$ s    : int 16
```

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 $n = 1000$ 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] 1000    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 4.1.2). So, in our case,

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

```
## List of 3
## $ dSet      : chr "myDataset"
## $ data      : num [1:1000, 1:4] 9.23 8.39 10.13 1.84 4.33 ...
## $ 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 pair-wise distances, we can forward a matrix of precomputed distances including the command's argument *is.distance = TRUE*,

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

In the case of a supervised dataset, we can supply a vector of labels to be mapped. 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 [1:1000, 1:4] 9.23 8.39 10.13 1.84 4.33 ...
## $ lbls      : num [1:1000] 16 12 16 2 2 4 7 3 15 7 ...
## $ is.distance: logi FALSE
```

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&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 $z = n \text{ layers}/\text{threads}$ (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; (iii) 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.

- 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.

```
myMap <- bdm.ptsne(myMap, threads = 4, layers = 2, rounds = 2, ppx = 25)
```

```
+++ running 4 threads
+++ processing data
+++ Computing Betas, perplexity 200
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.1631 0.2389  0.2832  0.2939  0.3375  0.7031
+++ computing ptSNE
... threads 4, size 500, epochs 62, iters 23
+++ epoch 0000/0062  9.0644e-01  2.8029e+00
--- round 01/02      <Cost>      <Size>      epSecs  time2End  eTime
+++ epoch 0001/0062  9.2820e-01  2.8029e+00    0.1136  00:00:07  09:48
+++ epoch 0002/0062  9.3100e-01  3.0781e+00    0.0705  00:00:06  09:48
... (output truncated)
```

```

+++ epoch 0031/0062  6.2072e-01  5.1665e+01    0.1072  00:00:03  09:48
--- round 02/02      <Cost>      <Size>      epSecs  time2End  eTime
+++ epoch 0032/0062  6.1840e-01  5.2777e+01    0.1042  00:00:03  09:48
... (output truncated)
+++ epoch 0062/0062  5.6600e-01  7.9076e+01    0.1131  00:00:00  09:48
... threads 4, size 500, epochs 62, iters 23
+++ epoch 0062/0062  5.6600e-01  7.9076e+01    0.1113  00:00:07

```

Note that the first parameter is *myMap* itself and the output value is a new *bdm* instance. This is the general rule for 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)
```

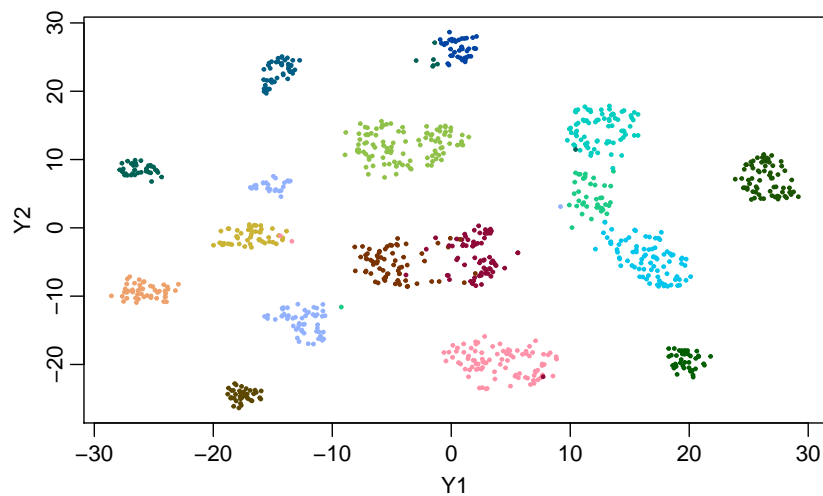
```

## List of 8
## $ threads: num 4
## $ layers : num 2
## $ rounds : int 2
## $ boost  : num 1
## $ alpha  : num 0.5
## $ Y      : num [1:1000, 1:4] -11.3 9.76 -13.31 6.32 2.81 ...
## $ cost   : num [1:4, 1:63] 0.907 0.907 0.906 0.905 0.929 ...
## $ size   : num [1:2, 1:63] 2.8 2.81 2.8 2.81 2.99 ...

```

The new element *myMap\$ptsne* includes a matrix *myMap\$ptsne\$Y*(1000, 4) with the mapping positions (y_i^1, y_i^2) . The positions of the mapped data-points are given by layers as successive 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.ptsne.plot()*,

```
bdm.ptsne.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).

Note that `bdm.ptsne()` performs a complex sequence of operations (Garriga and Bartumeus 2018) and therefore it has more parameters than those shown in the example (though most of them work well with default values):

- 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 $n \text{ layers} / \text{threads}$.
 - `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 4.2.2).
 - `rounds` sets the number of rounds.
 - `boost` changes the default run-and-mixing scheme. The number of epochs is set to $1/\text{boost} \sqrt{n}$ and the number of iterations per epoch to $\text{boost} \sqrt{z}$. 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 $\text{boost} \gg 1$.
- 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).
- 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.
- 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.
- 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 4.1.2). Thus, in case of a long ptSNE, we can anticipate things like a wrong parametric configuration.
- 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 = 50, g.exp = 2)
```

```
+++ running 4 threads
+++ paKDE for layer 1/2 +++
+++ Computing Betas, perplexity 50
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.362  1.923   2.283   2.697  3.449   5.593
  computing grid cell densities ...
+++ cdf 0.999
```

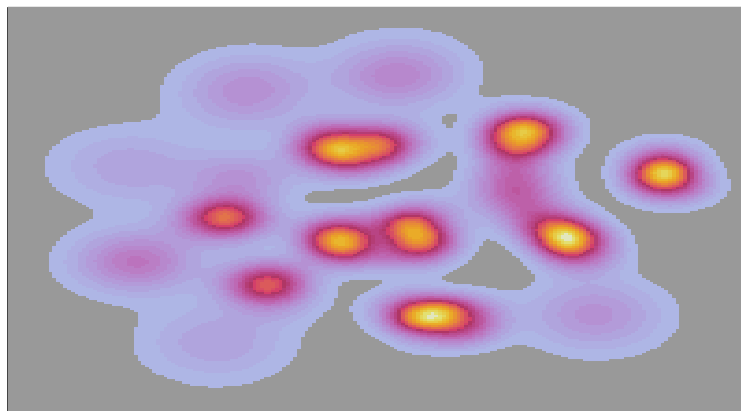
The output is attached to the list as a new element named `myMap$pakde`. This element is itself a list of lists (with one element per layer). By default, the command computes the paKDE for layer one, which is attached to `myMap$pakde[[1]]`. The argument `layer = l` allows computing the paKDE for any 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 50
## ..$ beta : num [1:1000] 0.0828 0.113 0.0981 0.151 0.1067 ...
## ..$ g.exp: num 2
## ..$ x : num [1:200] -37.7 -37.3 -37 -36.6 -36.3 ...
## ..$ y : num [1:200] -37.6 -37.2 -36.8 -36.4 -36 ...
## ..$ z : num [1:200, 1:200] 8.99e-10 1.15e-09 1.47e-09 1.87e-09 2.36e-09 ...
```

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.pakde.plot(myMap)
```



The `bdm.pakde()` command performs the following sequence of operations (Garriga and Bartumeus 2018):

- Setting up the running environment (`threads = 2`, `type = 'SOCK'`):
 - `threads` and `type` specify the number of threads that are 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 increase efficiency, so we would typically specify as many threads as the number of physical cores available.
- 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.**
- Computing the density estimation (`g = 200`, `g.exp = 4`):
 - `g` defines the grid size (the length of the raster side in number of cells). 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 0.9989962`). **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 issue using lower values of `g.exp`. With `g.exp = 0` the limits of the grid will match those of the embedding.
- 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.

```
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]]`.

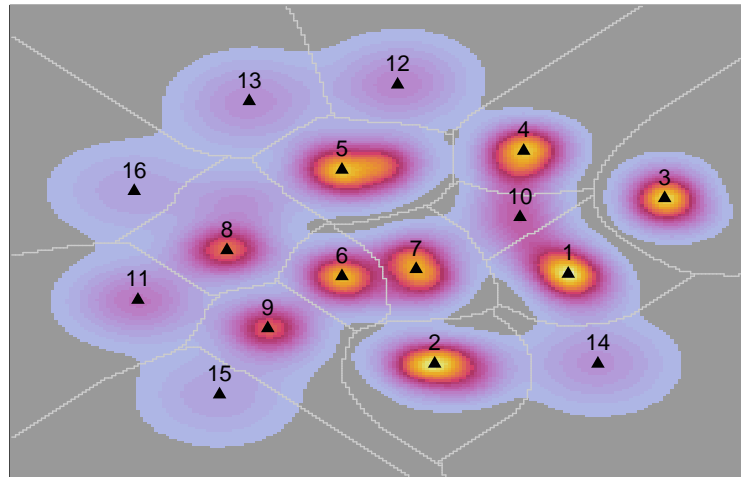
```
str(myMap$wtt)
```

```
## List of 1
## $ :List of 5
## ..$ grid: num [1:2, 1:6] 200 200 -38 -38 35 ...
## ..$ P : num [1:16, 1] 17351 9715 23777 27739 26090 ...
```

```
## ..$ M : num [1:16, 1:2] 151 115 177 139 90 90 110 59 70 138 ...
## ..$ C : num [1:40000, 1] 15 15 15 15 15 15 15 15 15 15 ...
## ..$ s : int 16
```

At this stage the lines that delimit the clusters are added to the density heat-map image. We can visualize the clustering with,

```
bdm.wtt.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)
```

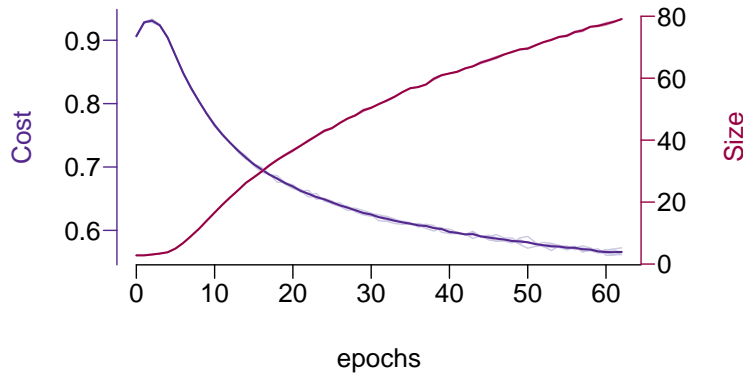
2.2 Visual assessment of the output.

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(myMap)
```



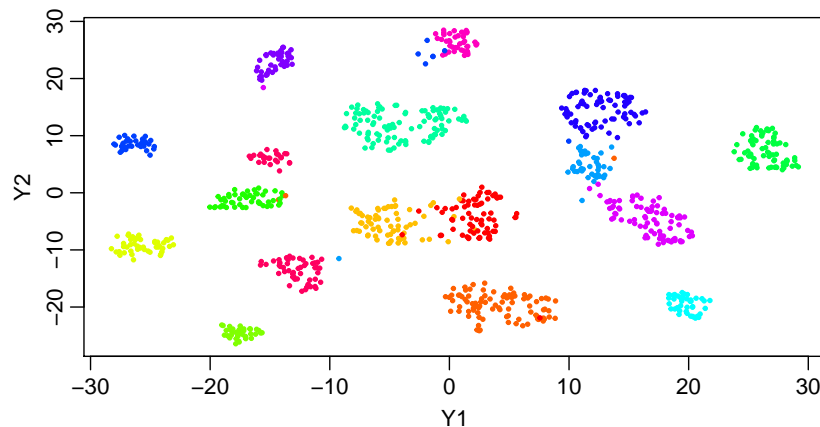
The solid blue line depicts the average cost. The light blue lines around the solid blue line depict the embedding cost values by thread (each partial t-SNE). The solid red line depicts the average size. The light red lines around the solid red line depict the embedding size values by layers (each global t-SNE).

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 with more iterations. 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 should better let the algorithm iterate until reaching a stable embedding size.

2.2.2 plotting functions

Please, refer to the package help to see specific arguments to minimally tweak the plots (*e.g* class-palette, background colour, data-point size). Here are some examples,

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



2.2.3 bdm.qMap()

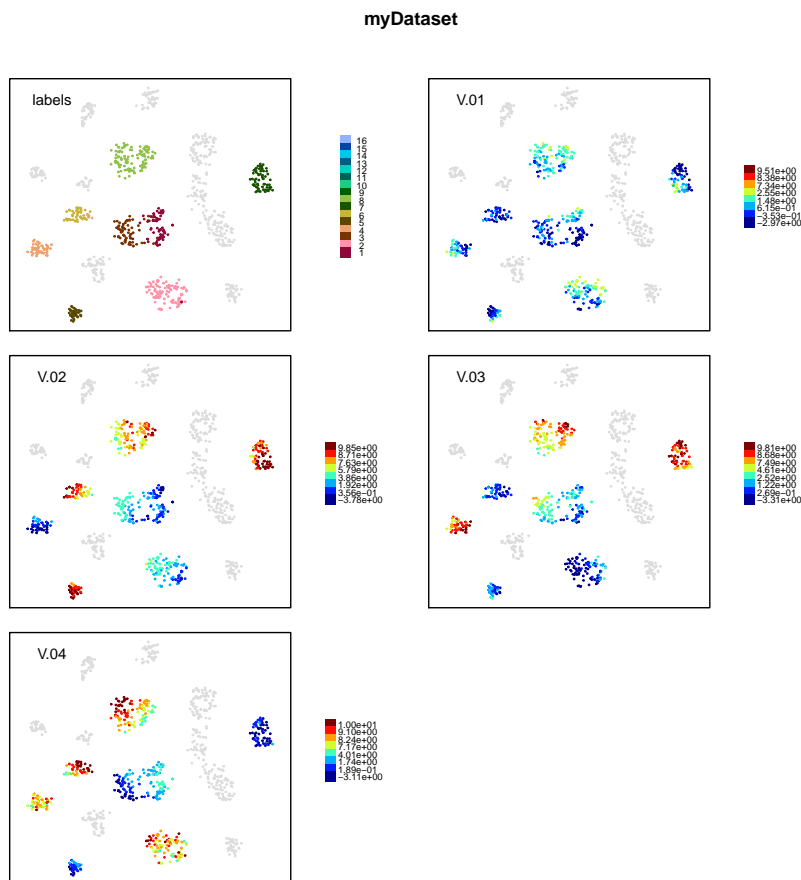
bdm.qMap() maps numeric variables onto the embedding, depicting its value as a quantile-map. A quantile-map is similar to a heat-map. The difference is that in a quantile-map colours indicate quantiles instead of ranges. In this way we always get a clear depiction of the distribution of the variable 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 *bdm.ptsne.plot()*) and the rest depicting the mapping of one variable each (with a maximum of 15 variables).

By default, *bdm.qMap()* plots *\$data* (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 want to map. This matrix can include any **numeric** covariate of interest.

The argument *subset*, given as a vector of row indexes of the input data matrix, restricts the mapping of the given variables to a subset of the input data. 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.

```
bdm.qMap(myMap, data = myMap$data[, 1:4], subset = which(myMap$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 joint distribution $P(V = v_i, C = c_j)$ where $V = \{v_1, \dots, v_l\}$ is the class/covariate and $C = \{c_1, \dots, c_g\}$ are the grid cells of the paKDE raster. The result is a fuzzy distribution of the class/covariate at each cell.

Computing the joint 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).

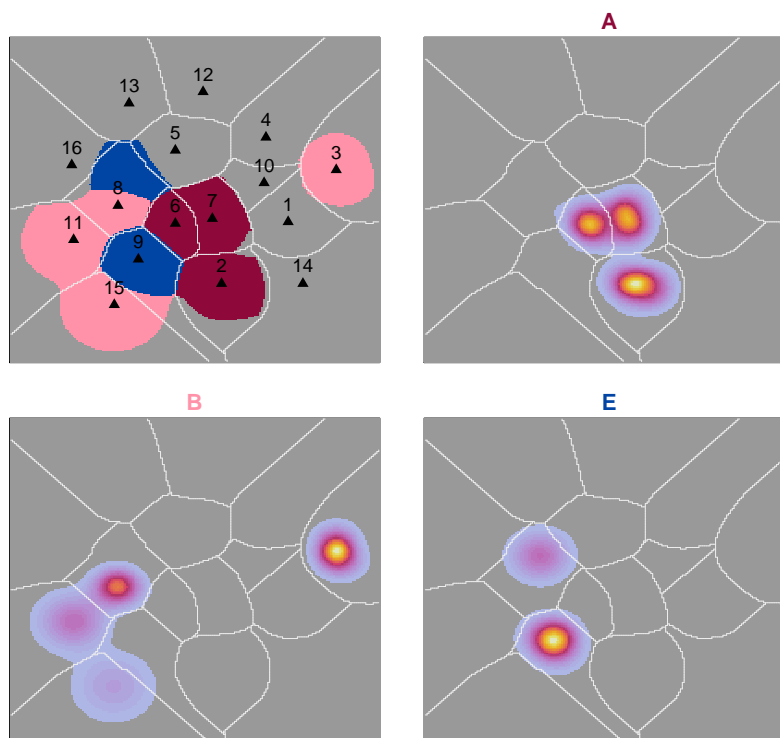
```
# assume these are our class covariates
myclasses <- c('A', 'B', 'C', 'D', 'E')[myMap$lbls %/% 4 + 1]
```

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

```
+++ running 4 threads
+++ cdf 0.9999
```

We can now plot the class density maps,

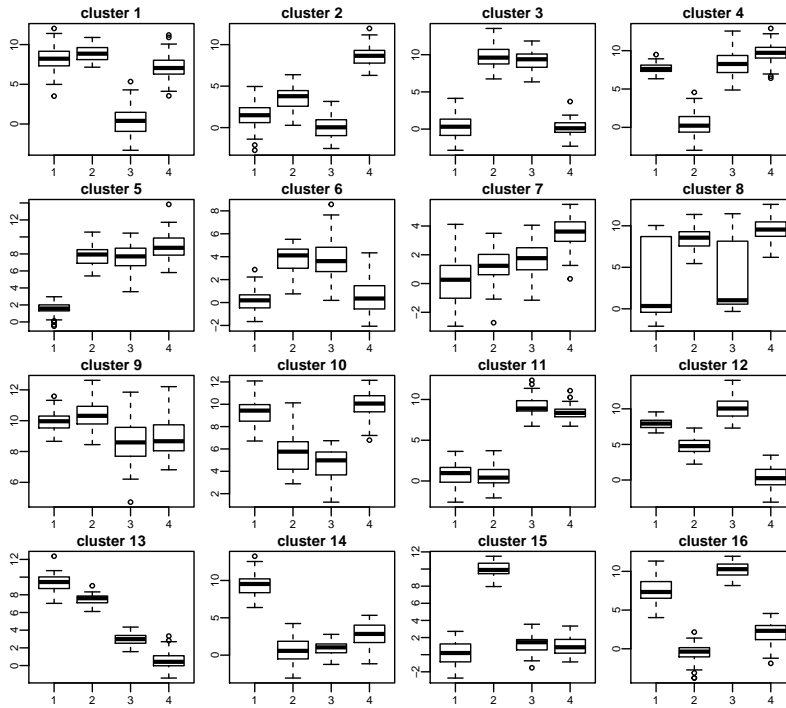
```
bdm.dMap.plot(myMap, classes = c('A', 'B', 'E'))
```



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 Post-processing

Given the values of perplexity for both, `bdm.ptsne()` and `bdm.pakde()`, the `bdm.wtt()` function returns a clustering at the finest grain, meaning that, however small is a peak in the density landscape, it defines its own cluster. Though many of these clusters might be irrelevant for our purposes, this is the base line to get to coarser classifications by hierarchically merging clusters. Nonetheless, merging of clusters can be based on different criteria (*e.g.* spatial, temporal) and can be implemented using different heuristics. A relevant parameter here is the number of clusters of the final clustering.

3.1 Spatial merging of clusters

These are functions to merge clusters based on its spatial proximity. The heuristic that drives the merging is based on minimum loss of signal-to-noise-ratio (S2NR). The S2NR is the explained/unexplained variance ratio measured in the high dimensional space based on the given low dimensional clustering.

3.1.1 bdm.optk.s2nr()

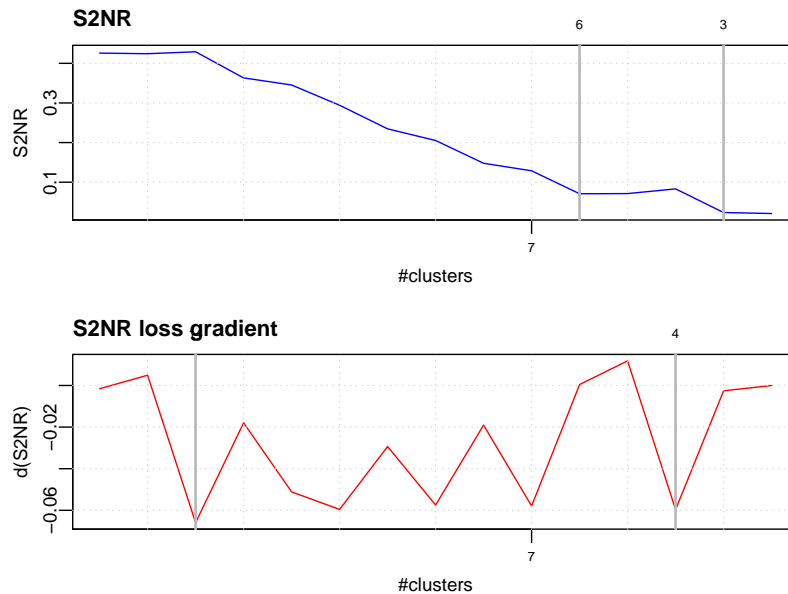
This function performs a recursive merging of clusters until reaching a configuration of only 2 clusters and the S2NR is measured at each step.

```
myMap <- bdm.optk.s2nr(myMap, plot.optk = F, ret.optk = T)
```

```
+++ processing data
+++ computing clustering hierarchy ...
+++ initial S2N ratio ...    4.2579e-01    16
+++ merging 10 into 1    4.2423e-01    15
+++ merging 7 into 6    4.2917e-01    14
... (output truncated)
+++ merging 3 into 1    7.0791e-02    6
+++ blocked hierarchy level reached at 6 clusters !!!
+++ next level ...
+++ merging 8 into 6    7.1264e-02    5
+++ merging 2 into 1    8.3086e-02    4
+++ merging 4 into 1    2.3446e-02    3
+++ blocked hierarchy level reached at 3 clusters !!!
+++ next level ...
+++ merging 6 into 5    2.0888e-02    2
+++ hierarchy levels .... :    6    3
+++ S2NR significant loss :    14    4
```

For large datasets this computation can take a while, so we can save this result by setting `ret.optk = TRUE`. Afterwards, we can plot it to see the optimal number of clusters in terms of S2NR,

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



3.1.2 bdm.merge.s2nr()

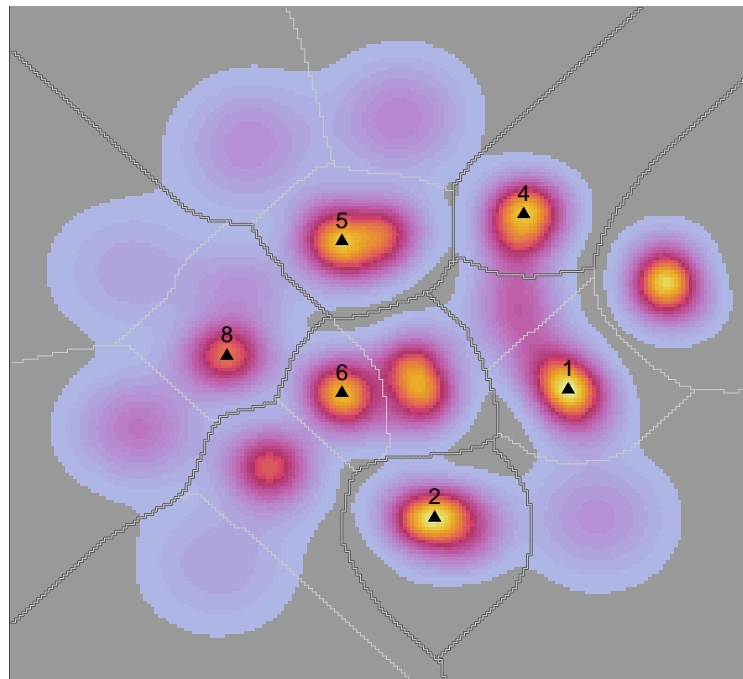
Once we know the number of clusters we run this function to get the desired merged configuration.

```
myMap <- bdm.merge.s2nr(myMap, k = 6, plot.merge = F, ret.merge = T)
```

```
## +++ processing data
## +++ computing clustering hierarchy ...
## +++ initial S2N ratio ... 4.2579e-01 16
## +++ merging 10 into 1 4.2423e-01 15
## +++ merging 7 into 6 4.2917e-01 14
## +++ merging 14 into 1 3.6323e-01 13
## +++ merging 11 into 8 3.4528e-01 12
## +++ merging 16 into 8 2.9416e-01 11
## +++ merging 12 into 5 2.3458e-01 10
## +++ merging 13 into 5 2.0524e-01 9
## +++ merging 15 into 9 1.4778e-01 8
## +++ merging 9 into 6 1.2873e-01 7
## +++ merging 3 into 1 7.0791e-02 6
```

By default, (i.e. *plot.merge = T*), the merged configuration is shown on top of the original clustering with thicker lines. The result can be saved by setting *ret.merge = T* and shown afterwards by means of the *bdm.wtt.plot()* function.

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



4 Working with remote multiprocessor systems

4.1 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 4.1.1, 4.1.2, 4.1.3). 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 4.1.4, 4.1.5).

4.1.1 `bdm.mybdm()`

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

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

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

4.1.2 `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_z4_12_b1_r2_p25_m06.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*).

4.1.3 `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_z4_12_b1_r2_p25_m15.RData
```

4.1.4 `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"
```

4.1.5 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 transferred 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_z4_l2_b1_r2_p25_m15.RData
```

4.2 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, info = 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.**

4.2.1 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**),

```
~$ 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 will be running (64 in our case).

4.2.2 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.* omp, 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 illustration 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 `snw`) 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. <https://doi.org/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: 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): 113–25. [https://doi.org/10.1016/0165-1684\(94\)90060-4](https://doi.org/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): 1236–65. <https://doi.org/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. <https://doi.org/10.1016/j.csda.2008.11.033>.