Partitioned Local Depth (PaLD) Clustering Analyses in R

by Lucy D'Agostino McGowan, Katherine Moore, Kenneth Berenhaut

Abstract An abstract of less than 150 words.

Introduction

- Describe PaLD (Ken & Kate?)
- Cite PaLD

We present a new package, **pald**, for calculating partitioned local depth (PaLD) probabilities, implementing clustering analyses, and creating data visualizations to represent the clusters. This paper will describe how to use the package as well as walk through two examples.

pald

The main functions in **pald** package can be split into 3 categories:

1. Helper functions to organize data into the correct format, into distance matrices and then contribution matrices
2. Functions that convert a contribution matrix into a variety of useful formats, including partitioned local depths, clusters, and graphs
3. Plotting functions

In addition, the package provides a number of pertinent example data sets commonly used to demonstrate cluster algorithms, including a synthetic data set of two-dimensional points created by Gionis et al. to demonstrate clustering aggregation, a sample of walking distances from a pump in the infamous cholera outbreak (Peter Li, 2019), clustering data generated from the scikit-learn Python package (Pedregosa et al., 2011), and data compiled by Love and Irizarry (2015) of tissue gene expressions.

While it is not a necessity, the **pald** package is designed to function well with the pipe operator, `%>%`, from the Bache and Wickham (2014) **magrittr** package in that the first argument of each function is the data. This functionality will be demonstrated below.

Helper functions to create contribution matrix

For demonstration purposes, below is a sample data frame with two variables, `x1` and `x2`. The methods put forth here work on data frames with higher dimensions, as described in the **Examples** section, we are simply choosing a small data frame here for demonstration purposes.

```
library(pald)
d <- data.frame(
  x1 = c(6, 8, 11, 16, 4),
  x2 = c(5, 4, 13, 7, 18)
)
rownames(d) <- c("A", "B", "C", "D", "E")
```

The first step needed to calculate the partitioned local depths is to construct a *distance matrix*. If the data are already in this form, the user can skip to the next step. The `get_distance_matrix()` function converts an input data frame into a distance matrix, as demonstrated below.

```
get_distance_matrix(d)
```

```
#>           A         B        C        D        E
#> A 0.0000000 0.4584177 1.717380 2.158418 2.229205
#> B 0.4584177 0.0000000 1.644296 1.778777 2.505929
#> C 1.7173802 1.6442957 0.000000 1.468398 1.713327
#> D 2.1584182 1.7787772 1.468398 0.000000 3.158040
#> E 2.2292047 2.5059291 1.713327 3.158040 0.000000
```

This will create an $n \times n$ distance matrix, where n corresponds to the number of observations in the original data frame, in this example $n = 5$. By default, the distance matrix is *scaled*, is possible to create a distance matrix that is not scaled by changing the `scale` argument to `FALSE`.

```
get_distance_matrix(d, scale = FALSE)
```

```
#>           A         B        C         D         E
#> A  0.000000  2.236068 9.433981 10.198039 13.152946
#> B  2.236068  0.000000 9.486833  8.544004 14.560220
#> C  9.433981  9.486833 0.000000  7.810250  8.602325
#> D 10.198039  8.544004 7.810250  0.000000 16.278821
#> E 13.152946 14.560220 8.602325 16.278821  0.000000
```

As mentioned previously, the functions in this package are designed to work with the **magrittr** pipe operator, `%>%`, so the same code as above could be written utilizing this format.

```
d %>%
  get_distance_matrix()
```

```
#>           A         B        C        D        E
#> A 0.0000000 0.4584177 1.717380 2.158418 2.229205
#> B 0.4584177 0.0000000 1.644296 1.778777 2.505929
#> C 1.7173802 1.6442957 0.000000 1.468398 1.713327
#> D 2.1584182 1.7787772 1.468398 0.000000 3.158040
#> E 2.2292047 2.5059291 1.713327 3.158040 0.000000
```

This distance matrix can then be passed to the `get_contribution_matrix()` function in order to calculate a matrix of contributions. Again, if the user begins with a distance matrix, they can skip the first step and simple input the contribution matrix into this function.

```
distance_matrix <- get_distance_matrix(d)
get_contribution_matrix(distance_matrix)
```

```
#>        1      2     3     4      5
#> A 0.2875 0.1625 0.000 0.050 0.0000
#> B 0.1750 0.3000 0.050 0.050 0.0000
#> C 0.0000 0.0625 0.300 0.175 0.0500
#> D 0.0500 0.0500 0.175 0.300 0.0000
#> E 0.0000 0.0000 0.050 0.000 0.2125
```

Again, the **magrittr** pipe can be used as follows.

```
d %>%
  get_distance_matrix() %>%
  get_contribution_matrix()
```

```
#>        1      2     3     4      5
#> A 0.2875 0.1625 0.000 0.050 0.0000
#> B 0.1750 0.3000 0.050 0.050 0.0000
#> C 0.0000 0.0625 0.300 0.175 0.0500
#> D 0.0500 0.0500 0.175 0.300 0.0000
#> E 0.0000 0.0000 0.050 0.000 0.2125
```

The *contribution matrix* output by the `get_contribution_matrix()` is the main input for the majority of the remaining functions.

Functions that convert a contribution matrix into useful formats

From the *contribution matrix*, a variety of useful quantities can be calculated. Below, we create a contribution matrix using the functions described in the previous section.

```
d %>%
  get_distance_matrix() %>%
  get_contribution_matrix() -> contribution_matrix
```

To calculate the *clusters* that each point will fall into, we can use the `get_clusters()` function. This will output a data frame with two columns, the first will correspond to the `point`, as identified by the row name of the original input data frame, `d`, the second will identify the `cluster` that each point belongs to.

```
get_clusters(contribution_matrix)
```

```
#> # A tibble: 5 x 2
#>   point cluster
#>   <chr>   <dbl>
#> 1 A           1
#> 2 B           1
#> 3 C           2
#> 4 D           2
#> 5 E           3
```

In this example, three clusters are identified with these five points. Points `A` and `B` fall into cluster 1. Points `C` and `D` into cluster 2, and point `E` in cluster 3.

The `get_depths()` function calculates the *depths* of each point, outputting a data frame with two columns, `point` indicating the point, as identified by the row name of the original data frame, `d`, and `depth` indicating the depth of the point. The data frame is arranged by depth, with the deepest point listed first.

```
get_depths(contribution_matrix)
```

```
#> # A tibble: 5 x 2
#>   point depth
#>   <chr> <dbl>
#> 1 C     0.588
#> 2 B     0.575
#> 3 D     0.575
#> 4 A     0.5
#> 5 E     0.262
```

In this case, the deepest point is `C`.

The `get_bound()` function will calculate the *bound* of the contribution matrix.

```
get_bound(contribution_matrix)
```

```
#> [1] 0.14
```

In this case, the bound is 0.14.

The `any_isolated()` function will check whether there are any isolated points that will inadvertently be dropped by a graph.

```
any_isolated(contribution_matrix)
```

```
#> [1] FALSE
```

In this case, there are no isolated points.

The `get_pald_cluster_matrix()` will calculate a matrix of partitioned local depth clusters. This function contains an argument `keep_all_edges` that indicates whether all edges should be kept. The default value is `FALSE`, indicating that edges that are less than the expectation will be dropped, allowing a clear clustering of points. Changing this argument to `TRUE` results in the pairwise minimum of the contribution matrix and the transpose of the contribution matrix.

```
get_pald_cluster_matrix(contribution_matrix, keep_all_edges = TRUE)
```

```
#>        1      2     3     4      5
#> A 0.2875 0.1625 0.000 0.050 0.0000
#> B 0.1625 0.3000 0.050 0.050 0.0000
#> C 0.0000 0.0500 0.300 0.175 0.0500
#> D 0.0500 0.0500 0.175 0.300 0.0000
#> E 0.0000 0.0000 0.050 0.000 0.2125
```

Leaving the `keep_all_edges` as the default, `FALSE` will take this pairwise minimum seen above and drop the edges that are less than the expectation.

Figure 1: Visualize the points from data frame 'd'

```
get_pald_cluster_matrix(contribution_matrix)
```

```
#>        1      2     3     4 5
#> A 0.0000 0.1625 0.000 0.000 0
#> B 0.1625 0.0000 0.000 0.000 0
#> C 0.0000 0.0000 0.000 0.175 0
#> D 0.0000 0.0000 0.175 0.000 0
#> E 0.0000 0.0000 0.000 0.000 0
```

Finally, the `get_pald_graph()` function takes the contribution matrix and creates an **igraph** object, a graph that describes the relationship between the points.

```
get_pald_graph(contribution_matrix)
```

```
#> IGRAPH 2c0b54a UNW- 5 2 --
#> + attr: name (v/c), weight (e/n)
#> + edges from 2c0b54a (vertex names):
#> [1] A--B D--C
```

Here we see that there are two connected components, points `A` and `B`, which form the first cluster, and points `D` and `C` which form the second.

Plotting functions

The final category of function is functions for data visualization. We can begin by visualizing the points in data frame `d` (Figure 1).

```
library(ggplot2)
ggplot(d, aes(x1, x2)) +
  geom_text(label = rownames(d))
```

We can then pass the contribution matrix to the `plot_pald_graph()` function to view the relationship between points (Figure 2). By default, this will invoke the `layout_nicely()` function from **igraph** to determine the layout of the graph.

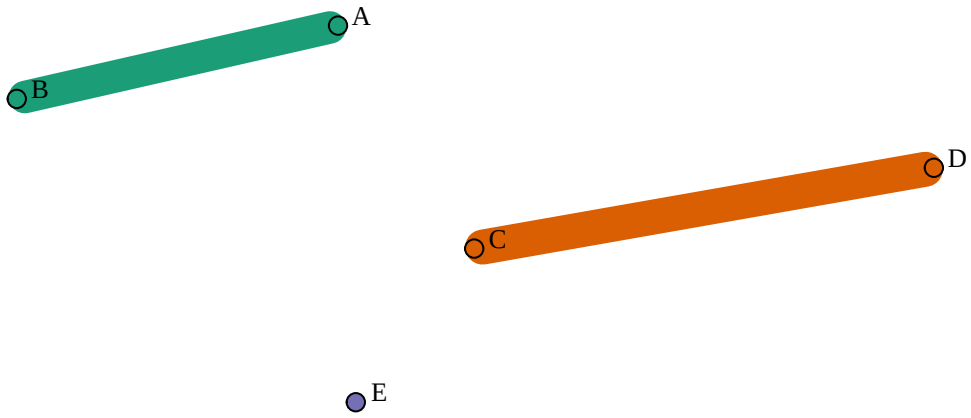

Figure 2: PaLD graph displaying the relationship between the points in data frame 'd'

```
d %>%
  get_distance_matrix() %>%
  get_contribution_matrix() %>%
  plot_pald_graph()
```

The `layout` argument allows the user to pass a matrix to dictate the layout of the graph. For example, if we wanted the graph to match the visualization displayed in Figure 1, we can pass `as.matrix(d)`, or a matrix of the data frame `d` to the `layout` argument (Figure 3.

```
d %>%
  get_distance_matrix() %>%
  get_contribution_matrix() %>%
  plot_pald_graph(layout = as.matrix(d))
```

This `plot_pald_graph()` function will also permit parameters that can be passed to the `plot.igraph()` function. For example, to add axes to the graph, the user can pass the `axes = TRUE` argument to the `...` in the `plot_pald_graph()` function (Figure 4).

```
d %>%
  get_distance_matrix() %>%
  get_contribution_matrix() %>%
  plot_pald_graph(layout = as.matrix(d),
                  axes = TRUE)
```

Examples

We will demonstrate the utility of the **pald** package in two clustering examples.

Clustering tissue gene expression data

The first example data frame is a subset of tissue gene expression data from Zilliox and Irizarry (2007), McCall et al. (2011), and McCall et al. (2014), obtained from the **tissuesGeneExpression** bioconductor package (Love and Irizarry, 2015). This data set is included in the **pald** package in an object called `tissue`.

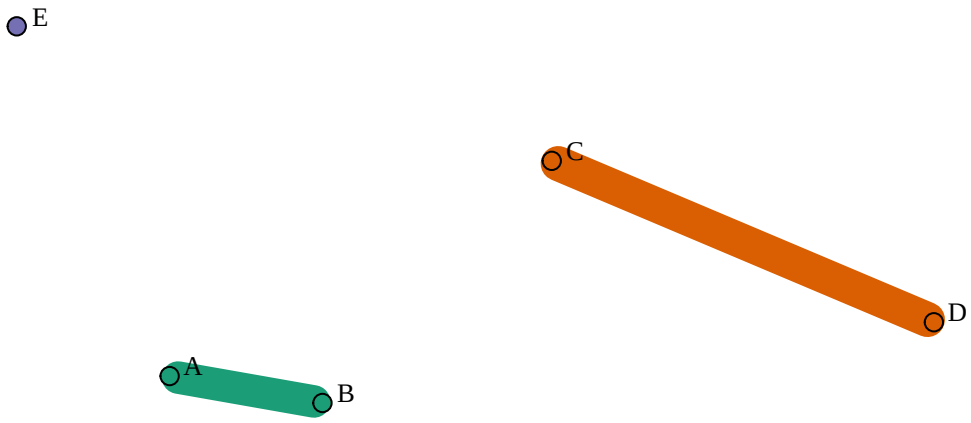

Figure 3: PaLD graph displaying the relationship between the points in data frame 'd', matching the original layout in Figure 1

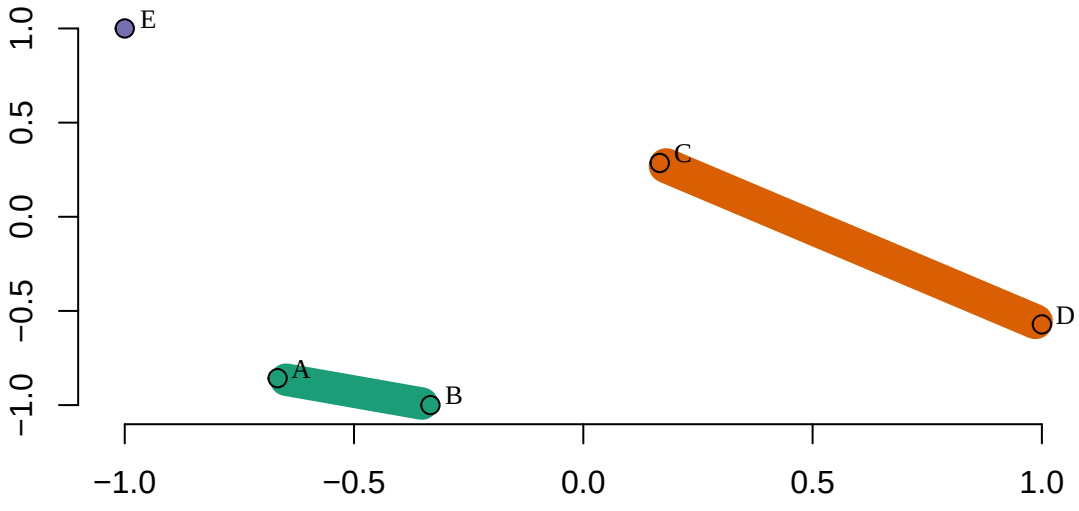

Figure 4: PaLD graph displaying the relationship between the points in data frame 'd', matching the original layout in Figure 1, adding axes

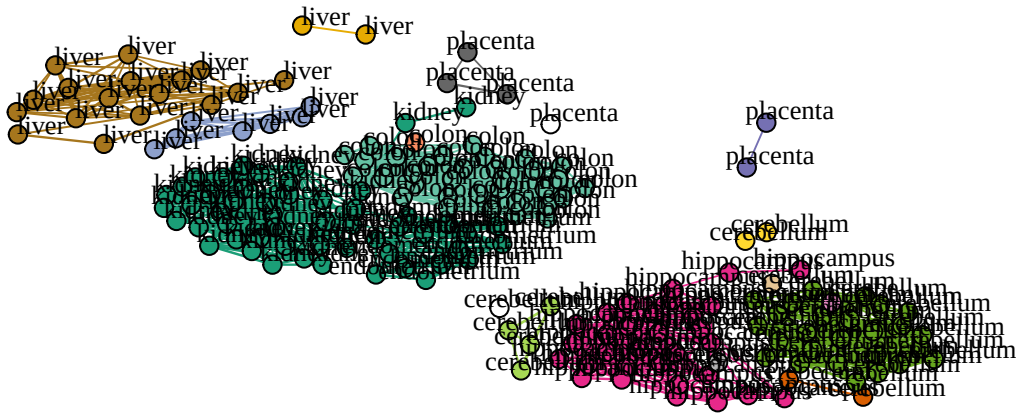

Figure 5: PaLD clustering of tissue data

The `tissue` object is a matrix with 189 rows, each with a corresponding tissue, such as `colon`, `kidney` or `cerebellum`. There are 22,215 columns corresponding to gene expression data from each of these rows.

We will first calculate the contribution matrix for this data frame.

```
tissue_contribution_matrix <- tissue %>%
  get_distance_matrix(scale = FALSE) %>%
  get_contribution_matrix()
```

We can then use this contribution matrix to display the relationship between tissue samples using the `plot_pald_graph()` function (Figure 5).

```
plot_pald_graph(tissue_contribution_matrix)
```

The `get_clusters()` function can be used to identify the clusters of each tissue sample. Since the output is a data frame, we can summarize the clusters using commonly used data analysis techniques. For demonstration purposes, we will use the **dplyr** package to summarize the contribution of clusters.

```
library(dplyr)
get_clusters(tissue_contribution_matrix) %>%
  group_by(cluster, point) %>%
  count()
```

```
#> # A tibble: 19 x 3
#> # Groups:   cluster, point [19]
#>    cluster point           n
#>      <dbl> <chr>       <int>
#>  1       1 endometrium    15
#>  2       1 kidney         39
#>  3       2 hippocampus    31
#>  4       3 cerebellum     26
#>  5       4 cerebellum      1
#>  6       5 colon          33
#>  7       6 colon           1
#>  8       7 liver           7
```

```
#>  9       8 cerebellum      1
#> 10       9 liver          17
#> 11      10 cerebellum      2
#> 12      11 liver           2
#> 13      12 cerebellum      1
#> 14      13 cerebellum      4
#> 15      14 cerebellum      2
#> 16      15 cerebellum      1
#> 17      16 placenta        2
#> 18      17 placenta        1
#> 19      18 placenta        3
```

From this, we can glean that cluster one consists of two types of tissue, the kidney and endometrium. Cluster two is comprised of only the hippocampus.

Clustering generated data

The **pald** includes two data frames generated from the scikit-learn Python package (Pedregosa et al., 2011), `noisy_moons` and `noisy_circles`.

The `noisy_moons` data frame consists of 500 rows and two columns.

```
moons_contribution_matrix <- noisy_moons %>%
  get_distance_matrix() %>%
  get_contribution_matrix()
```

When plotting the `noisy_moons` PaLD graph, we want the layout to match the layout of the original data, so we will pass `as.matrix(noisy_moons)` to the `layout` parameter in the `plot_pald_graph()` function. Additionally, here the row names are meaningless, they just correspond to the location of the generated data, so we can remove the labels on the plot by passing `vertex.label = NA` to the `...` of the `plot_pald_graph()` function.

```
plot_pald_graph(moons_contribution_matrix,
                layout = as.matrix(noisy_moons),
                vertex.label = NA)
```

The `noisy_circles` data frame consists of 500 rows and two columns. We can create the contribution matrix for this data frame using the same methods as used for the `noisy_moons` data.

```
circles_contribution_matrix <- noisy_circles %>%
  get_distance_matrix() %>%
  get_contribution_matrix()
```

Similarly, we will pass the layout and remove vertex labels for the noisy circles PaLD plot.

```
plot_pald_graph(circles_contribution_matrix,
                layout = as.matrix(noisy_circles),
                vertex.label = NA)
```

Because these are **igraph** objects, they can be combined as you would normally combine a plot. For example, we could add the line `par(mfrow = c(1, 2))` above the plot calls to create an output where both plots are displayed side by side (Figure 6).

```
par(mfrow = c(1, 2))
plot_pald_graph(moons_contribution_matrix,
                layout = as.matrix(noisy_moons),
                vertex.label = NA)
plot_pald_graph(circles_contribution_matrix,
                layout = as.matrix(noisy_circles),
                vertex.label = NA)
```

The ability of the PaLD algorithm to discern clusters is demonstrated here.

Summary

This paper introduces the **pald** package, demonstrating it's utility for providing a parameter-free clustering algorithm that can easily be applied to any data set.

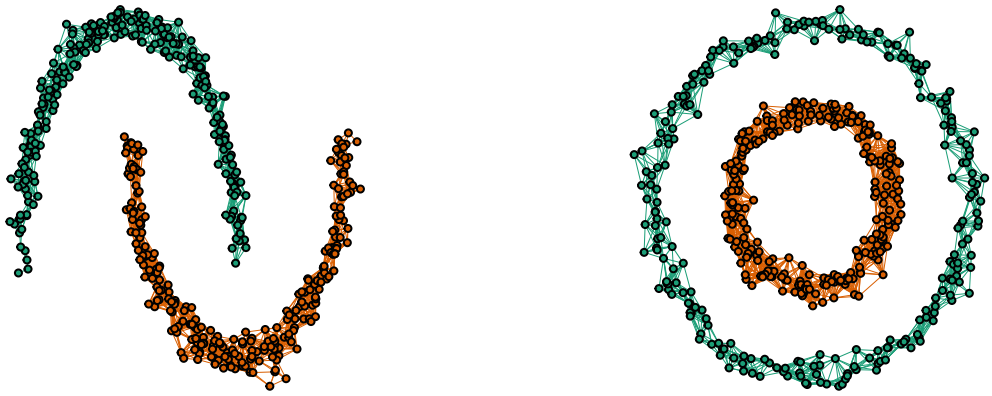

Figure 6: PaLD clustering of noisy moons (left) and noisy circles (right) data

Bibliography

S. M. Bache and H. Wickham. *magrittr: A Forward-Pipe Operator for R*, 2014. URL https://CRAN.R-project.org/package=magrittr. R package version 1.5. [p1]

A. Gionis, H. Mannila, and P. Tsaparas. Clustering aggregation, acm transactions on knowledge discovery from data. *vol*, 1:1–30. [p1]

M. Love and R. Irizarry. *tissuesGeneExpression. Subset of gene expression data*, 2015. URL https://github.com/genomicsclass/tissuesGeneExpression. [p1, 5]

M. N. McCall, K. Uppal, H. A. Jaffee, M. J. Zilliox, and R. A. Irizarry. The gene expression barcode: leveraging public data repositories to begin cataloging the human and murine transcriptomes. *Nucleic acids research*, 39(suppl_1):D1011–D1015, 2011. [p5]

M. N. McCall, H. A. Jaffee, S. J. Zelisko, N. Sinha, G. Hooiveld, R. A. Irizarry, and M. J. Zilliox. The gene expression barcode 3.0: improved data processing and mining tools. *Nucleic acids research*, 42 (D1):D938–D943, 2014. [p5]

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011. [p1, 8]

Peter Li. *cholera: Amend, Augment and Aid Analysis of John Snow's Cholera Map*, 2019. URL https://CRAN.R-project.org/package=cholera. R package version 0.7.0. [p1]

M. J. Zilliox and R. A. Irizarry. A gene expression bar code for microarray data. *Nature methods*, 4(11): 911–913, 2007. [p5]

Lucy D'Agostino McGowan
Wake Forest University
Winston-Salem, NC
27106
mcgowald@wfu.edu

Katherine Moore
Wake Forest Unversity
Winston-Salem, NC
27106
mooreke@wfu.edu

Kenneth Berenhaut
Wake Forest University
Winston-Salem, NC
27106
berenhks@wfu.edu