# Partitioned Local Depth (PaLD) Clustering Analyses in R

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

**Abstract** An abstract of less than 150 words.

## Introduction

Partitioned Local Depths (PaLD) is a framework for a holistic consideration of the community structure of distance-based data. Leveraging a socially inspired perspective, the method provides network-based community information which is founded on a new measure of local depth and pairwise cohesion (partitioned local depth). The method does not require distributional assumptions, optimization criteria, nor extraneous inputs. A complete description of the perspective, together with a discussion of the underlying social motivation, theoretical results, and applications to additional data sets is provided in Berenhaut et al. (2022).

Building on existing approaches to (global) depth, local depth expresses features of centrality as an interpretable probability which is free of parameters and robust to outliers. Then, partitioning the probability which defines local depth, we obtain a measure of cohesion between pairs of points. Both local depth and cohesion reflect aspects of relative position (rather than absolute distance) and provide a straightforward way to account for varying density across the space. Specifically, provided that two sets are separated (in the sense that the minimum between-set distance is greater than the maximum within-set distance), cohesion is invariant under the contraction and dilation of the distances within each set. This property may be particularly valuable when one has reason to believe that the magnitude of distances varies across the space.

As cohesion captures a sense of the relationship strength between points, we can then visualize the resulting community structure with a network whose edges are weighted by (mutual) cohesion. The underlying social framework motivates an elegant threshold for distinguishing between strongly and weakly cohesive pairs.

As seen throughout this paper, we display the network obtained from cohesion using a force-directed graph drawing algorithm and emphasize the strong ties (colored by connected component). We refer to the connected components of the network of strong ties as "clusters." Note that to qualify as a cluster in this definition, one may not have any strong ties with those outside the cluster, and thus the existence of disjoint groups is a strong signal for separation. Here, clusters are identified without additional user inputs nor optimization criteria. If one wishes to further break the community graph into groups, one may use community detection methods for networks (such as spectral clustering or the Louvain algorithm). Though only briefly considered here, one may also use the collection of strong ties in place of (weighted) k-nearest neighbors in settings such as classification and smoothing. Overall, the structural information obtained from local depth, cohesion and community graphs can provide a holistic perspective on the data which does not require the use of distributional assumptions, optimization criteria nor additional user inputs.

We present a new package, **pald**, for calculating Partitioned Local Depths (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 cohesion matrices
2. Functions that convert a cohesion 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, clustering data generated from the scikit-learn Python package (Pedregosa et al., 2011), data describing cognate relationships between words across 87 Indo-European languages (Dyen et al., 1992), data compiled by Love and Irizarry (2015) of tissue gene expressions, and three example data sets generated for the Berenhaut et al. (2022) paper.

While it is not a necessity, the **pald** package is designed to function well with the pipe operator, `|>`. This functionality will be demonstrated below.

### Creating the cohesion 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)
df <- data.frame(
  x1 = c(6, 8, 11, 16, 4, 14),
  x2 = c(5, 4, 13, 7, 6, 10)
)
rownames(df) <- c("A", "B", "C", "D", "E", "F")
```

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 `dist()` function converts an input data frame into a distance matrix, as demonstrated below.

```
d <- dist(df)
```

This will create a `dist` object. If converted to a matrix, this will be a $n \times n$ distance matrix, where $n$ corresponds to the number of observations in the original data frame, in this example $n = 6$.

This dist object, or equivalently a distance matrix, can then be passed to the `cohesion_matrix()` function in order to calculate the pairwise cohesion values. Cohesion is an interpretable probability that reflects the strength of alignment of two points. Again, if the user begins with a distance matrix, they can skip the first step and simply input the distance matrix into this function.

```
d <- dist(df)
cohesion_matrix(d)

#>           A         B         C         D         E         F
#> A 0.2333333 0.1666667 0.0000000 0.0000000 0.1666667 0.0000000
#> B 0.1333333 0.2333333 0.0000000 0.0000000 0.1000000 0.0000000
#> C 0.0000000 0.0000000 0.2333333 0.1000000 0.0000000 0.1000000
#> D 0.0000000 0.0000000 0.1000000 0.2666667 0.0000000 0.1666667
#> E 0.1333333 0.1000000 0.0000000 0.0000000 0.2333333 0.0000000
#> F 0.0000000 0.0000000 0.1000000 0.1666667 0.0000000 0.2666667
#> attr(,"class")
#> [1] "cohesion_matrix" "matrix"          "array"
```

Equivalently, the user can use the native pipe `|>` as follows.

```
df |>
  dist() |>
  cohesion_matrix()

#>           A         B         C         D         E         F
#> A 0.2333333 0.1666667 0.0000000 0.0000000 0.1666667 0.0000000
#> B 0.1333333 0.2333333 0.0000000 0.0000000 0.1000000 0.0000000
#> C 0.0000000 0.0000000 0.2333333 0.1000000 0.0000000 0.1000000
#> D 0.0000000 0.0000000 0.1000000 0.2666667 0.0000000 0.1666667
#> E 0.1333333 0.1000000 0.0000000 0.0000000 0.2333333 0.0000000
#> F 0.0000000 0.0000000 0.1000000 0.1666667 0.0000000 0.2666667
#> attr(,"class")
#> [1] "cohesion_matrix" "matrix"          "array"
```

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

### Functions that convert a cohesion matrix into useful formats

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

```
df |>
  dist() |>
  cohesion_matrix() -> cohesion
```

To calculate the *clusters* that each point will fall into, we can use the `community_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, `df`, the second will identify the `cluster` that each point belongs to.

```
community_clusters(cohesion)
```

```
#>   point cluster
#> A     A       1
#> B     B       1
#> C     C       2
#> D     D       3
#> E     E       1
#> F     F       3
```

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

The `local_depths()` function calculates the *depths* of each point, outputting a vector of local depths. Local depth is an interpretable probability which reflects aspects of relative position and centrality via distance comparisons (i.e., $d(z,x) < d(z,y)$).

```
local_depths(cohesion)
```

```
#>         A         B         C         D         E         F
#> 0.5666667 0.4666667 0.4333333 0.5333333 0.4666667 0.5333333
```

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

The `strong_threshold()` function will calculate the cohesion threshold for strong ties. This is equal to half the average of the diagonal cohesion matrix. This is a threshold that may be used to distinguish between strong and weak ties.

```
strong_threshold(cohesion)
```

```
#> [1] 0.1222222
```

In this case, the threshold is `0.12`.

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

```
any_isolated(cohesion)
```

Here, there are no isolated points.

The function `cohesion_strong()` will update the cohesion matrix to set all weak ties to zero (via the `strong_threshold()` function). Optionally, the matrix will also be symmetrized, with the default parameter `symmetric = TRUE`.

```
cohesion_strong(cohesion)
```

```
#>           A         B         C         D         E         F
#> A 0.2333333 0.1333333 0.0000000 0.0000000 0.1333333 0.0000000
#> B 0.1333333 0.2333333 0.0000000 0.0000000 0.0000000 0.0000000
#> C 0.0000000 0.0000000 0.2333333 0.0000000 0.0000000 0.0000000
#> D 0.0000000 0.0000000 0.0000000 0.2666667 0.0000000 0.1666667
#> E 0.1333333 0.0000000 0.0000000 0.0000000 0.2333333 0.0000000
#> F 0.0000000 0.0000000 0.0000000 0.1666667 0.0000000 0.2666667
#> attr(,"class")
#> [1] "cohesion_matrix" "matrix"          "array"
```

Finally, the `community_graphs()` function takes the cohesion matrix and creates **igraph** objects, graphs that describes the relationship between the points. This function will output a list of three objects:

- `G`: the weighted (community) graph whose edge weights are mutual cohesion

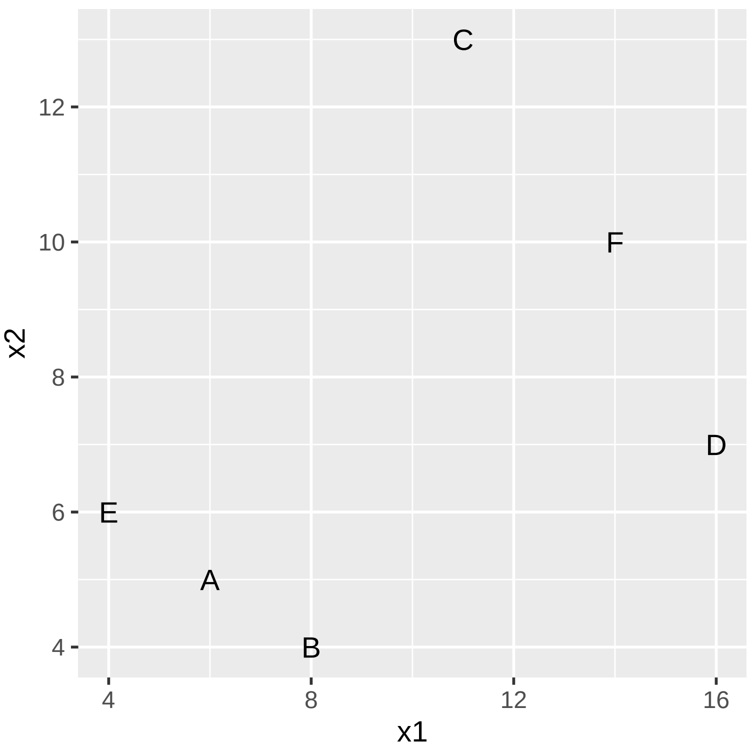

**Figure 1:** Visualize the points from data frame 'df'

- `G_strong`: the weighted (community) graph consisting of edges for which mutual cohesion is greater than the threshold for strong ties
- `layout`: the graph layout, using the Fruchterman Reingold (FR) force-directed graph drawing for the graph `G`

```
graphs <- community_graphs(cohesion)
graphs[["G_strong"]]

#> IGRAPH b3faedc UNW- 6 3 --
#> + attr: name (v/c), weight (e/n)
#> + edges from b3faedc (vertex names):
#> [1] A--B A--E D--F
```

Here we see that there are three connected components, points `A` and `B`, and `A` and `E` which form the first cluster, and points `D` and `F` which form another.

## Plotting functions

The final category of function is functions for data visualization. We can begin by visualizing the points in data frame `df` (Figure 1). When visualizing these points, it is important to have a square plot so as to not distort the distances. When using the **ggplot2** package for this visualization, you can use the `aspect.ratio = 1` argument in the `theme()` function. If using the `plot()` function included in the base library, you can set the plot area to be a square by using `par(pty = "s")` prior to calling the `plot()` function, as shown below.

```
library(ggplot2)
ggplot(df, aes(x1, x2)) +
  geom_text(label = rownames(df)) +
  theme(aspect.ratio = 1)
```

We can pass the cohesion matrix to the `plot_community_graphs()` function to view the relationship between points (Figure 2).

```
plot_community_graphs(cohesion)
```

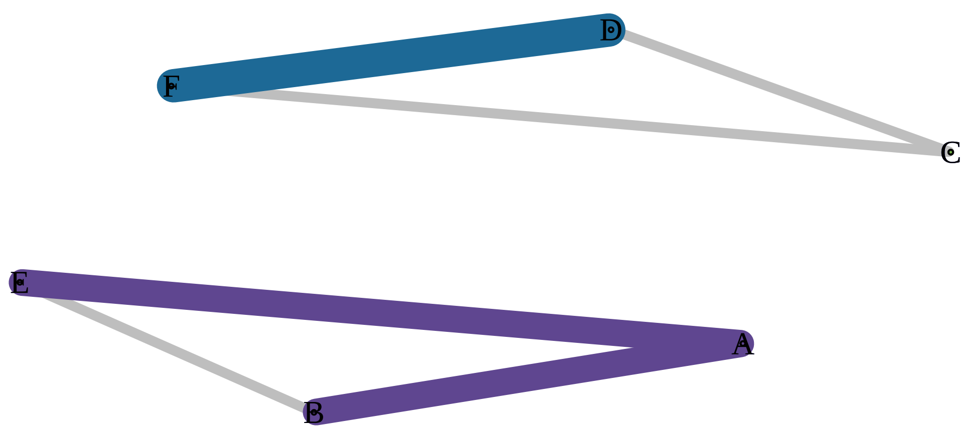

**Figure 2:** PaLD graph displaying the relationship between the points in data frame 'df'

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(df)`, or a matrix of the data frame `df` to the `layout` argument (Figure 3. Notice in the code below, we set `par(pty = "s")` in order to ensure that we have a square plotting region. Additionally, this `plot_community_graphs()` function will also permit parameters that can be passed to the `plot.igraph()` function. For example, we can pass arguments to the `plot.igraph` function via the `...`, for example to increase the vertex size and change the vertex label color, we can specify `vertex.size = 15` and `vertex.label.color = "white"`.

```
par(pty = "s")

plot_community_graphs(cohesion,
                      layout = as.matrix(df),
                      vertex.size = 15,
                      vertex.label.color = "white")
```

## Examples

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

### Clustering tissue gene expression data

The first example if from 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). A `dist` object was created using this data set and is included the **pald** package in an object called `tissue_dist`.

The `tissue_dist` object is a `dist` object resulting in a distance matrix with 189 rows and 189 columns.

We can create the cohesion matrix using the `cohesion_matrix` function.

```
tissue_cohesion <- cohesion_matrix(tissue_dist)
```

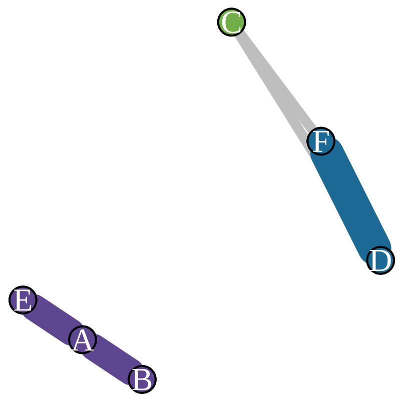

**Figure 3:** PaLD graph displaying the relationship between the points in data frame 'd', matching the original layout in Figure 1

The `community_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.

```
community_clusters(tissue_cohesion) |>
  dplyr::count(cluster, point)
```

```
#>    cluster       point  n
#> 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.

We can also display the relationships between tissue samples using the `plot_community_graphs()` function (Figure 4). For clarity of the display, we show how to remove the labels using `show_labels = FALSE`. We will instead color by the labels by passing these to the `vertex.color` parameter through

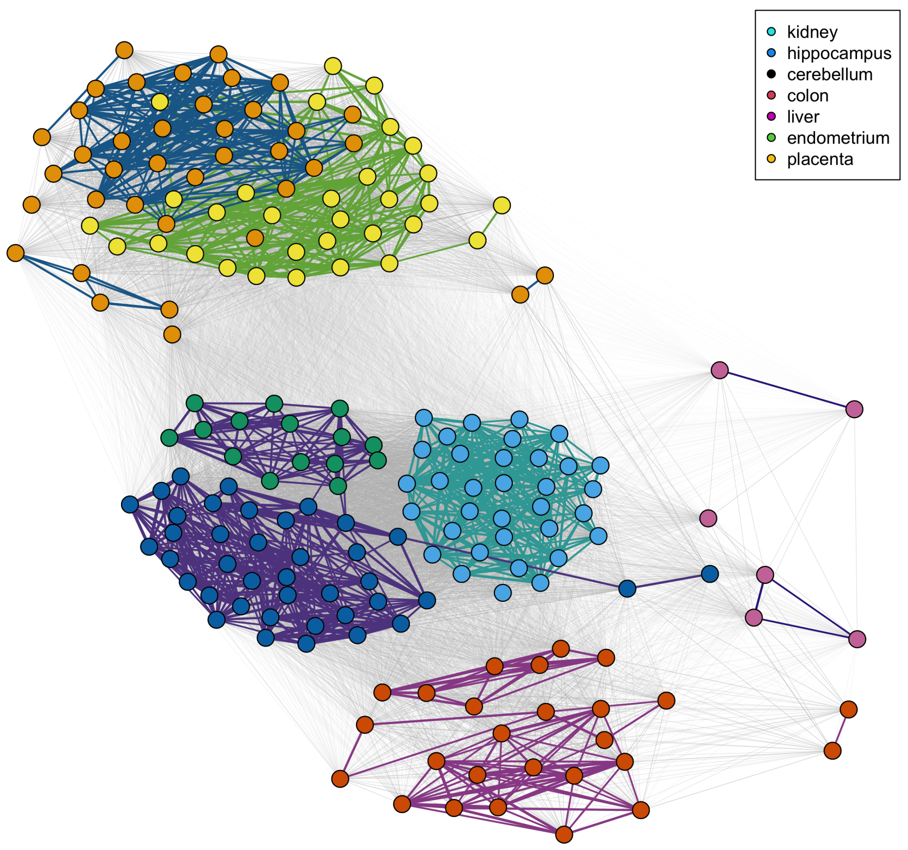

**Figure 4:** PaLD clustering of tissue data. The line colors indicate the PaLD clusters, the point colors indicate the tissue classification.

the ... to the `plot.igraph` function. Similarly, we can add a legend using the `legend()` function, as you would for a **igraph** visualization. Additionally, we use the `edge_width_factor` and `emph_strong` arguments to adjust the width of the lines between and within PaLD clusters.

```
labels <- rownames(tissue_cohesion)
plot_community_graphs(tissue_cohesion,
                      show_labels = FALSE,
                      vertex.size = 4,
                      vertex.color = as.factor(labels),
                      edge_width_factor = 35,
                      emph_strong = 5)
legend("topleft",
       legend = unique(as.factor(labels)),
       pt.bg = unique(as.factor(labels)),
       col = "black",
       pch = 21)
```

## Cognate-based Language Families

This example performs a PaLD analysis on a data set from Dyen et al. (1992) that examines the relationship between 87 Indo-European languages from the perspective of cognates. A `dist` object was created from this data set and included in the **pald** package in an object called `cognate_dist`.

This example will demonstrate how you can apply functions in the **igraph** package to objects output from the **pald** package. We can first use the `cohesion_matrix()` function to calculate the cohesion matrix and the `community_graphs()` function to create a list with the weighted community graph, the weighted community graph with only strong ties included (and all others set to 0), and the layout. From this, we can extract the graph with only the strong ties, here called `cognate_graph_strong`.

```
cognate_cohesion <- cohesion_matrix(cognate_dist)
cognate_graphs <- community_graphs(cognate_cohesion)

cognate_graph_strong <- cognate_graphs[["G_strong"]]
```

We can then use the `neighbors()` function from the **igraph** package to extract the neighbors in this graph. For example, if we wanted to extract all neighbors where the language is "French", we would run the following.

```
french_neighbors <- igraph::neighbors(cognate_graph_strong, "French")
french_neighbors

#> + 8/87 vertices, named, from 46d19b2:
#> [1] Italian          Ladin            Provencal        Walloon
#> [5] French_Creole_C French_Creole_D Spanish          Catalan
```

Similarly, we can print the associated neighborhood weights by subsetting the cohesion matrix.

```
cognate_cohesion["French", french_neighbors]

#>         Italian           Ladin       Provencal         Walloon French_Creole_C
#>      0.01997696      0.02094596      0.02871174      0.03258771      0.02406057
#> French_Creole_D         Spanish         Catalan
#>      0.02406057      0.01679733      0.01859688
```

## Clustering generated data

The **pald** includes three randomly generated data frames corresponding to plots from Berenhaut et al. (2022):

- `exdata1` is a data set consisting of 8 points to recreate Figure 1 in Berenhaut et al. (2022)
- `exdata2` is a data set consisting of 16 points to recreate Figure 2 in Berenhaut et al. (2022)
- `exdata3` is a data set consisting of 240 points to recreate Figure 4D in Berenhaut et al. (2022)

Here, we will demonstrate how to use `exdata3`. These points were generated from bivariate normal distributions with varying means and variances. There are eight "true" clusters.

We will demonstrate how we can compare PaLD to two clustering methods: *k*-means and hierarchical clustering. The code below calculates the cohesion matrix (`exdata_cohesion`) as well as the clusters via PaLD (`exdata_pald`), *k*-means (`exdata_kmeans`) and hierarchical clustering (`exdata_hclust`).

```
exdata_cohesion <- exdata3 |>
  dist() |>
  cohesion_matrix()

exdata_pald <- community_clusters(exdata_cohesion)$cluster

exdata_kmeans <- kmeans(exdata3, 8)$cluster

exdata_hclust <- exdata3 |>
  dist() |>
  hclust() |>
  cutree(k = 8)
```

We can compare this to the clustering generated by *k*-means and hierarchical clustering (Figure 5).

```
par(mfrow = c(1, 3), pty = "s")
plot(
  exdata3,
  pch = 16,
  col = pald_colors[exdata_pald],
```

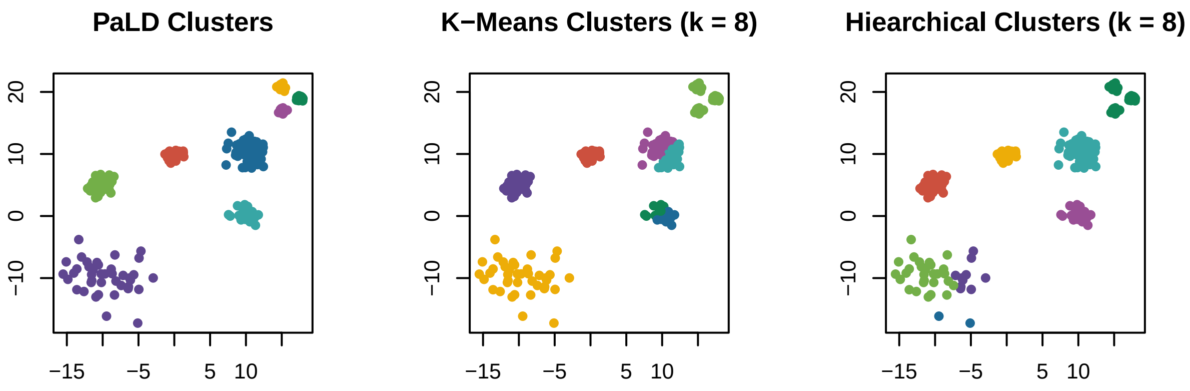

**Figure 5:** PaLD clustering of randomly generated example data (from Figure 4D from Berenhaut et al. (2022)) compared to *k*-means and hierarchical clustering with k = 8.

```
  xlab = "",
  ylab = "",
  main = "PaLD Clusters"
)
plot(
  exdata3,
  pch = 16,
  col = pald_colors[exdata_kmeans],
  xlab = "",
  ylab = "",
  main = "K-Means Clusters (k = 8)"
)
plot(
  exdata3,
  pch = 16,
  col = pald_colors[exdata_hclust],
  xlab = "",
  ylab = "",
  main = "Hiearchical Clusters (k = 8)"
)
```

Cohesion is particularly useful when considering data with varying local density, see discussion in (Berenhaut et al., 2022). Note that the PaLD algorithm is able to detect the eight natural groups within the data without the use of any additional inputs (e.g., number of clusters) nor optimization criteria. Despite providing the "correct" number of clusters (i.e., $k = 8$) both $k$-means and hierarchical clustering did not give the desired result.

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.

## Bibliography

K. S. Berenhaut, K. E. Moore, and R. L. Melvin. A social perspective on perceived distances reveals deep community structure. *Proceedings of the National Academy of Sciences*, 119(4), 2022. [p1, 8, 9]

I. Dyen, J. B. Kruskal, and P. Black. An indoeuropean classification: A lexicostatistical experiment. *Transactions of the American Philosophical Society*, 82(5):iii, 1992. doi: 10.2307/1006517. [p1, 7]

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]

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`