

Partitioned Local Depth (PaLD) Community Analyses in R

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

Abstract Partitioned Local Depth (PaLD) is a framework for a holistic consideration of the community structure of distance-based data. This paper describes an R package, [pald](#), for calculating Partitioned Local Depth (PaLD) probabilities, implementing community analyses, and creating data visualizations to display community structure. We describe how to use the package as well as walk through several examples.

Introduction

Partitioned Local Depth (PaLD) is a framework for a holistic consideration of community structure for 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 via 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, as shown in [Berenhaut et al. \(2022\)](#), 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 there is heterogeneity in density 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 a straightforward yet elegant threshold for distinguishing between strongly and weakly cohesive pairs.

Throughout this paper, we will 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 community “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 consider using community detection methods (such as spectral clustering or the Louvain algorithm), as available, say, in the [igraph](#) package. 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 Depth (PaLD) probabilities, implementing community analyses, and creating data visualizations to display community structure. This paper will describe how to use the package, walk through several examples, and compare the method to commonly used techniques in R. Together, these demonstrate both the novelty of the method and utility of the implementation in package described.

pald

The main functions in the [pald](#) package can be split into 3 categories:

1. A function for computing the cohesion matrix
2. Functions for extracting useful information from the cohesion matrix, such as local depths, neighbors, community clusters, and graph objects
3. Plotting functions for community graphs

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, data from the World Values Survey ([Inglehart et al., 2014](#)) on cultural values regarding family, religion, education, and institutions for several regions ([Muthukrishna et al., 2020](#)), 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

The input for the Partitioned Local Depths (PaLD) computations is a distance matrix or `dist` object. Note that the collection of input distances (or dissimilarities) does not need to satisfy the triangle inequality nor be symmetric.

For demonstration purposes, we will show how one can compute a distance matrix from an input data frame with, say, two variables `x1` and `x2`. The input data may be of any dimension; in fact the PaLD framework provides advantages when considering high-dimensional data (see the **Examples** section as well as [Berenhaut et al. \(2022\)](#)).

```
library(pald)
df <- data.frame(
  x1 = c(6, 8, 8, 16, 4, 14),
  x2 = c(5, 4, 10, 8, 4, 10)
)
rownames(df) <- c("A", "B", "C", "D", "E", "F")
```

The `dist()` function converts an input data frame into a distance matrix, as demonstrated below. If the data are already provided as a distance matrix (or `dist` object), the user can skip to the next step. Note that the distance matrix needed for the subsequent functions doesn't need to be a `dist` object and *need not* be symmetric.

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

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

This `dist` object, or 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 within local regions. It captures aspects of the relative positioning of points and accounts for varying density across the space.

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

#>           A         B         C         D         E         F
#> A 0.25000000 0.18333333 0.06666667 0.0000000 0.18333333 0.0000000
#> B 0.14000000 0.24000000 0.05000000 0.0000000 0.10666667 0.0000000
#> C 0.07333333 0.07333333 0.20333333 0.0000000 0.03333333 0.0800000
#> D 0.00000000 0.00000000 0.00000000 0.2333333 0.00000000 0.1333333
#> E 0.14000000 0.10666667 0.03333333 0.0000000 0.24000000 0.0000000
#> F 0.00000000 0.00000000 0.05000000 0.1400000 0.00000000 0.2400000
#> 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 0.25000000 0.18333333 0.06666667 0.0000000 0.18333333 0.0000000
```

```
#> B 0.14000000 0.24000000 0.05000000 0.0000000 0.10666667 0.0000000
#> C 0.07333333 0.07333333 0.20333333 0.0000000 0.03333333 0.0800000
#> D 0.00000000 0.00000000 0.00000000 0.2333333 0.00000000 0.1333333
#> E 0.14000000 0.10666667 0.03333333 0.0000000 0.24000000 0.0000000
#> F 0.00000000 0.00000000 0.05000000 0.1400000 0.00000000 0.2400000
#> attr(,"class")
#> [1] "cohesion_matrix" "matrix"           "array"
```

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

Functions for extracting information from the cohesion matrix

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

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

The `local_depths()` function calculates the *depth* 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., $d(z, x) < d(z, y)$, see [Berenhaut et al. \(2022\)](#)).

```
local_depths(cohesion)

#>      A       B       C       D       E       F
#> 0.6833333 0.5366667 0.4633333 0.3666667 0.5200000 0.4300000
```

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 of the cohesion matrix ([Berenhaut et al., 2022](#)), and is a threshold that may be used to distinguish between strong and weak ties.

```
strong_threshold(cohesion)

#> [1] 0.1172222
```

In this case, the threshold is a little above 0.117.

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, using the entry-wise (parallel) minimum of the cohesion matrix and its transpose, with the default parameter `symmetric = TRUE`.

```
cohesion_strong(cohesion)

#>      A       B       C       D       E       F
#> A 0.25 0.14 0.0000000 0.0000000 0.14 0.0000000
#> B 0.14 0.24 0.0000000 0.0000000 0.00 0.0000000
#> C 0.00 0.00 0.2033333 0.0000000 0.00 0.0000000
#> D 0.00 0.00 0.0000000 0.2333333 0.00 0.1333333
#> E 0.14 0.00 0.0000000 0.0000000 0.24 0.0000000
#> F 0.00 0.00 0.0000000 0.1333333 0.00 0.2400000
#> attr(,"class")
#> [1] "cohesion_matrix" "matrix"           "array"
```

The `community_graphs()` function takes the cohesion matrix and creates `igraph` objects, graphs that describe 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
- `G_strong`: the weighted (community) graph consisting of edges for which mutual (symmetrized) cohesion (i.e. the minimum of the two directed cohesion values for any given pair) 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 f9306e2 UNW- 6 3 --
#> + attr: name (v/c), weight (e/n)
#> + edges from f9306e2 (vertex names):
#> [1] A--B A--E D--F

```

Here we see that there are three connected components, ties A-B and A-E which form the first cluster, and the tie D-F which forms another.

The `any_isolated()` function will check whether there are any isolated points.

```
any_isolated(cohesion)
```

Here, there are no isolated points, i.e. points having zero cohesion with all other points in the data (an extreme form of outlier).

The “community clusters” identified by PaLD are the connected components of the graph of strong ties, `G_strong`. To directly calculate them, 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 (a cluster of size 1) and points D and F fall into cluster 3.

Plotting functions

The final category of function is that for data visualization. We can begin by visualizing the points in the data frame `df` (Figure 1). When visualizing these points, it is important to have the aspect ratio of the x and y axes equal to 1 so as to not distort distances. When using the `ggplot2` package for this visualization, one can use the `coord_fixed(ratio = 1)` function. If using the `plot()` function included in the base library, one can use the `asp = 1` argument.

```

library(ggplot2)
ggplot(df, aes(x1, x2)) +
  geom_text(label = rownames(df)) +
  coord_fixed(ratio = 1) +
  xlim(c(4, 16)) +
  ylim(c(4, 16))

```

We can pass the cohesion matrix to the `plot_community_graphs()` function to view the relationship between points (Figure 2). Notice in this plot the layout does not match that of the original data frame as seen in Figure 1. Since our original data is two dimensional, it may be reasonable to use this as the layout. Figure 3 will make this update as well as update some of the aesthetics, such as employing more readable labels.

```
plot_community_graphs(cohesion)
```

The `layout` argument allows the user to pass a matrix to dictate the 2-dimensional layout of the graph. For example, if we wanted the graph to match the visualization displayed in Figure 1, we could pass `as.matrix(df)`, or a matrix of the data frame `df` to the `layout` argument (Figure 3). Additionally, the `plot_community_graphs()` function will also permit parameters that can be passed to the `plot.igraph()` function. We can pass arguments to the `plot.igraph` function via the `...` argument; for example to increase the vertex size and change the vertex label color, we can specify `vertex.size = 100` and `vertex.label.color = "white"`. Additionally, to allow axes, we use `axes = TRUE`, and to put them back on the original scale we set `rescale = FALSE`, resetting the axis limits using `xlim` and `ylim`. The `par(pty = "s")` function forces the subsequent plot to be square.

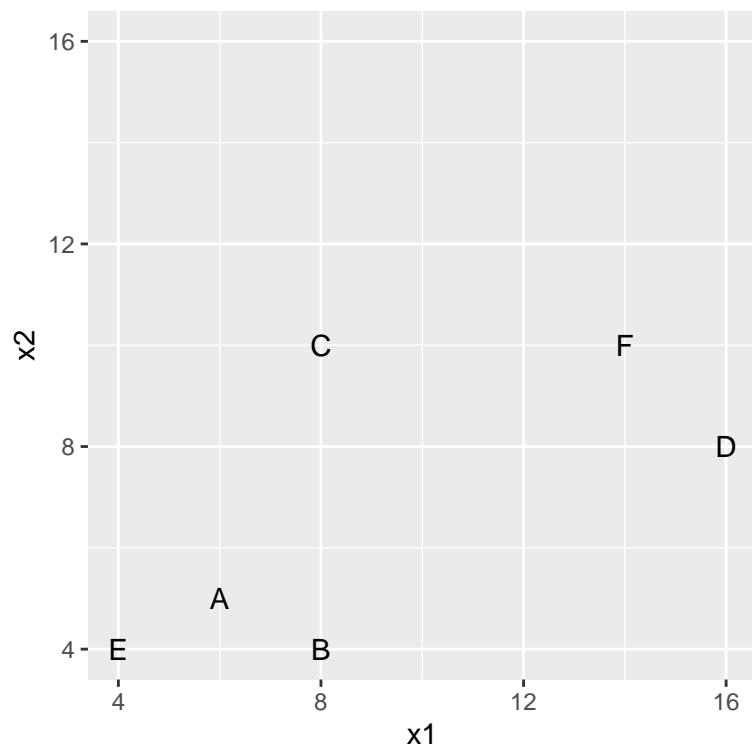


Figure 1: Visualization of the points from the data frame ‘df’

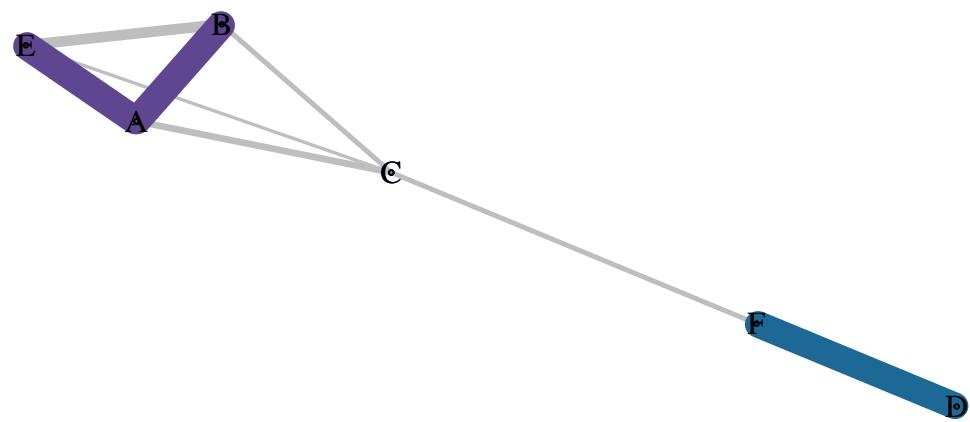


Figure 2: PaLD graph displaying the relationship between the points in data frame ‘df’

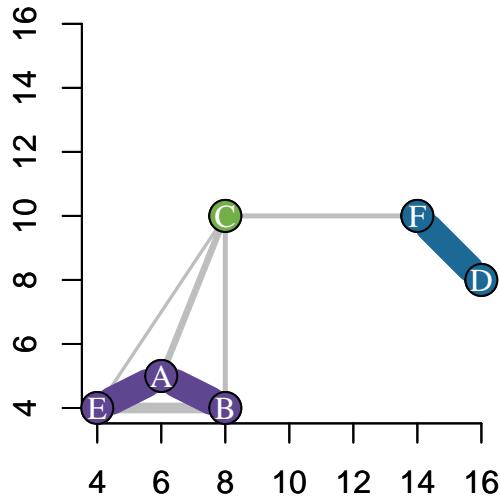


Figure 3: PaLD graph displaying the relationship between the points in data frame ‘df’, matching the original layout in Figure 1

```
par(pty = "s")

plot_community_graphs(cohesion,
                      layout = as.matrix(df),
                      vertex.size = 100,
                      vertex.label.color = "white",
                      axes = TRUE,
                      rescale = FALSE,
                      asp = 1,
                      xlim = c(4, 16),
                      ylim = c(4, 16))
```

Examples

We will demonstrate the utility of the `pald` package in three illustrative examples.

Clustering tissue gene expression data

The first example is from a subset of 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) consisting of 22,215-dimensional gene expression data from 189 tissue samples. A `dist` object was created using this data set and is included in 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)
```

The `community_clusters()` function can be used to identify the community cluster corresponding to each tissue sample. Since the output is a data frame, we can summarize the result using commonly

employed 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 1 consists of two types of tissue, the kidney and endometrium. Cluster 2 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 to the `plot.igraph` function (via the `...` argument). Similarly, we can add a legend using the `legend()` function, as you would for an `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, coded using 2,655-dimensional binary vectors. A `dist` object was created from this data set and is 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 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"]]
```

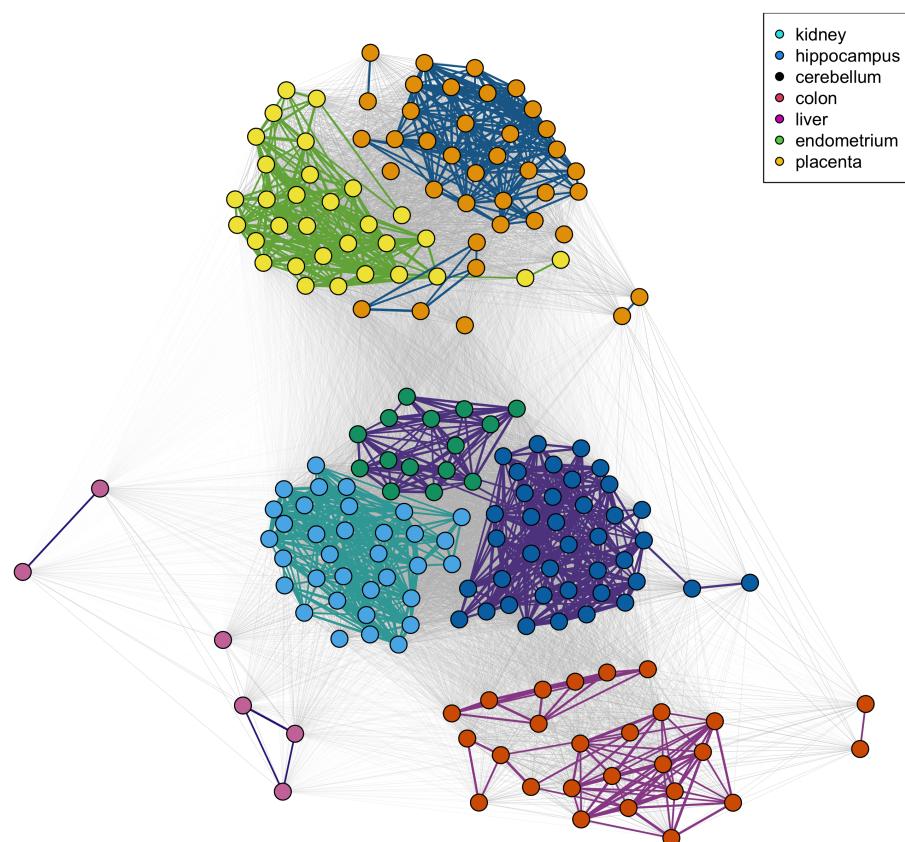


Figure 4: Community cluster network for the tissue data. The line colors indicate the PaLD clusters, the point colors indicate the tissue classification.

We can then use the `neighbors()` function from the `igraph` package to extract the strong neighbors in this graph. For example, if we wanted to extract all neighbors for the language “French”, we could run the following.

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

#> + 8/87 vertices, named, from 2660ad9:
#> [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
```

We can again use the `plot_community_graphs()` function to visualize the community clusters (Figure 5). One may note the commonly identifiable language clusters and that, under a slight rotation, some of the underlying geography is mirrored in the plot.

```
plot_community_graphs(
  cognate_cohesion,
  edge_width_factor = 30,
  emph_strong = 3,
  vertex.size = 3,
  vertex.label.cex = 0.7,
  vertex.label.dist = 1
)
```

Community analysis for generated data

The `pald` package includes three randomly generated data frames corresponding to plots from Berenhaus et al. (2022):

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

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 using complete linkage (`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 6).

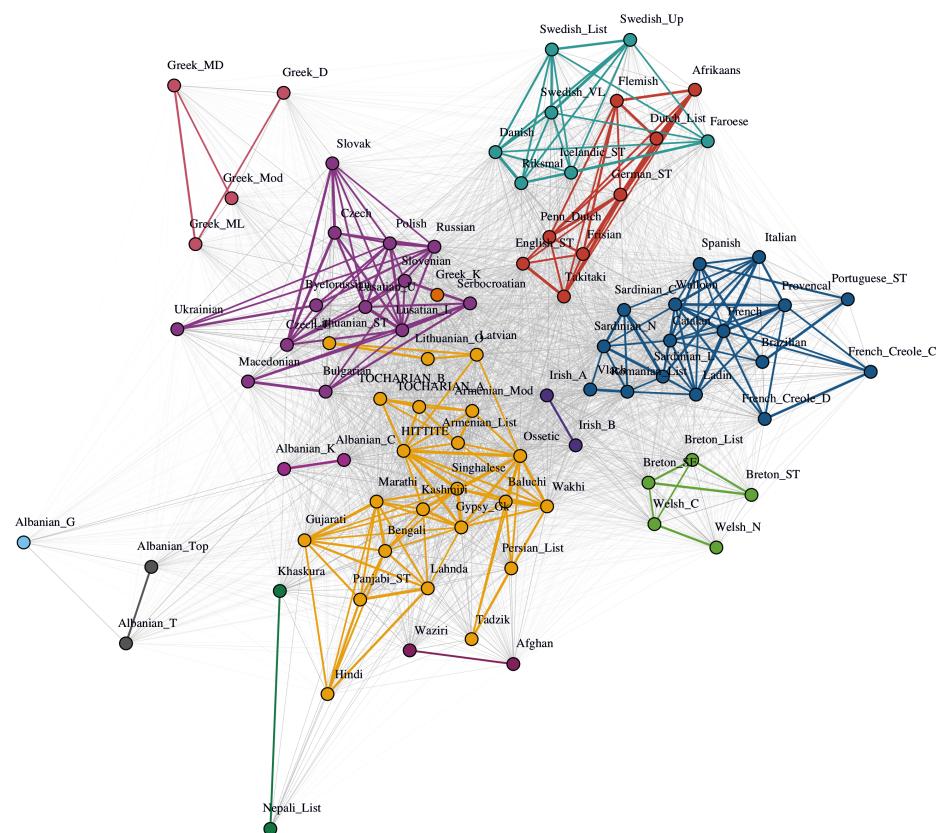


Figure 5: Community structure for 87 Indo-European languages, which employs cognate information that was coded via 2,665-dimensional binary vectors. Commonly identifiable language clusters arise along with informative inter- and intra-cluster structure. Several ancient languages are centrally located.

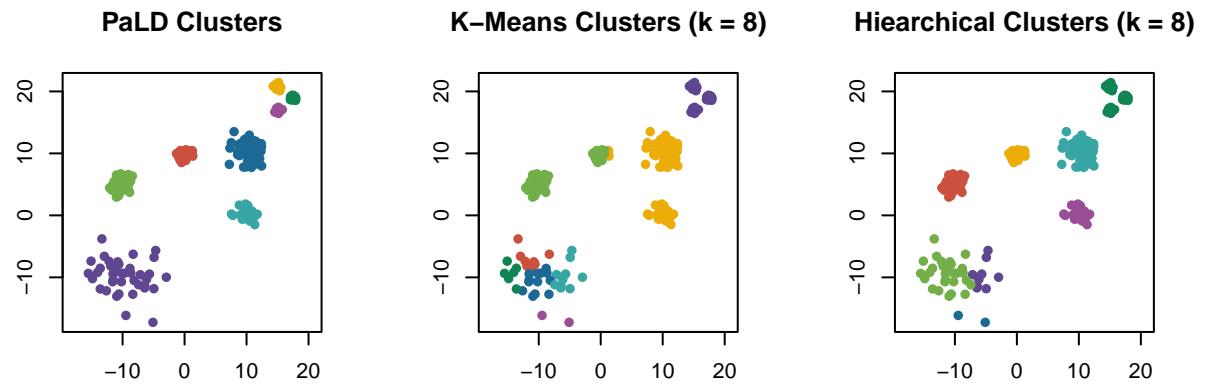


Figure 6: PaLD clustering of randomly generated example data (from Figure 4D from Berenhaut et al. (2022)) compared to k-means and hierarchical clustering with $k = 8$.

```
par(mfrow = c(1, 3), pty = "s")
plot(
  exdata3,
  pch = 16,
  col = pald_colors[exdata_pald],
  xlab = "",
  ylab = "",
  main = "PaLD Clusters",
  asp = 1
)
plot(
  exdata3,
  pch = 16,
  col = pald_colors[exdata_kmeans],
  xlab = "",
  ylab = "",
  main = "K-Means Clusters (k = 8)",
  asp = 1
)
plot(
  exdata3,
  pch = 16,
  col = pald_colors[exdata_hclust],
  xlab = "",
  ylab = "",
  main = "Hierarchical Clusters (k = 8)",
  asp = 1
)
```

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 the user input of the “correct” number of clusters (i.e., $k = 8$) both k -means and hierarchical

clustering did not give the desired result. For further examples, discussion, and theoretical results, see Berenhaut et al. (2022).

Summary

This paper introduces the `pald` package, demonstrating its utility for providing parameter-free clustering which can easily be implemented for a variety of data sets. We provide example code as well as compare the method to commonly used clustering techniques, *k*-means and hierarchical clustering.

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, 2, 3, 9, 11, 12]
- I. Dyen, J. B. Kruskal, and P. Black. An indo-european classification: A lexicostatistical experiment. *Transactions of the American Philosophical Society*, 82(5):iii, 1992. doi: 10.2307/1006517. [p2, 7]
- A. Gionis, H. Mannila, and P. Tsaparas. Clustering aggregation. *ACM Transactions on Knowledge Discovery from Data*, 1:1–30. [p2]
- R. Inglehart, C. Haerpfer, A. Moreno, C. Welzel, K. Kizilova, J. Díez-Medrano, M. Lagos, P. Norris, E. Ponarin, B. Puranen, et al. World values survey: All rounds–country-pooled datafile 1981-2014. madrid: Jd systems institute, 2014. 2014. [p2]
- M. Love and R. Irizarry. *tissuesGeneExpression*. Subset of gene expression data, 2015. URL <https://github.com/genomicsclass/tissuesGeneExpression>. [p2, 6]
- 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. [p6]
- 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. [p6]
- M. Muthukrishna, A. V. Bell, J. Henrich, C. M. Curtin, A. Gedranovich, J. McInerney, and B. Thue. Beyond western, educated, industrial, rich, and democratic (weird) psychology: Measuring and mapping scales of cultural and psychological distance. *Psychological Science*, 31(6):678–701, 2020. [p2]
- 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. [p2]
- M. J. Zilliox and R. A. Irizarry. A gene expression bar code for microarray data. *Nature Methods*, 4(11):911–913, 2007. [p6]

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

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

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