quollr: An R Package for Visalizing 2D Models in High Dimensional Space

by Jayani P.G. Lakshika, Dianne Cook, Paul Harrison, Michael Lydeamore, and Thiyanga S. Talagala

Abstract An abstract of less than 150 words.

```
#library(quollr)
library(knitr)
library(kableExtra)
library(readr)
library(ggplot2)
library(dplyr)
library(ggbeeswarm)
library(Rtsne)
library(umap)
library(phateR)
library(reticulate)
library(rsample)

set.seed(20230531)

use_python("~/miniforge3/envs/pcamp_env/bin/python")
use_condaenv("pcamp_env")

reticulate::source_python(paste0(here::here(), "/scripts/function_scripts/Fit_PacMAP_code.py"))
reticulate::source_python(paste0(here::here(), "/scripts/function_scripts/Fit_TriMAP_code.py"))

library(scales)
s_curve_noise_umap$UMAP1 <- rescale(s_curve_noise_umap$UMAP1)

y_min <- -sqrt(3)/2
y_max <- sqrt(3)/2

s_curve_noise_umap$UMAP2 <- ((s_curve_noise_umap$UMAP2 - min(s_curve_noise_umap$UMAP2))/(max(s_curve_noise_uma

# Define the desired size of the hexagons
hex_size <- 0.2

# Compute the horizontal spacing (dx) and vertical spacing (dy) between hexagon centers
dx <- sqrt(3) * hex_size
dy <- 1.5 * hex_size

x_length <- 1
y_length_scaled <- diff(range(s_curve_noise_umap$UMAP2))

# Compute the number of bins along the x-axis and y-axis
# num_bins_x <- ceiling(x_length / dx)
# num_bins_y <- ceiling(y_length_scaled / dy)

# s_curve_noise_umap[,1:(NCOL(s_curve_noise_umap) - 1)] <- scale(s_curve_noise_umap[,1:(NCOL(s_curve_noise_uma
```

1 Introduction

2 Methodology

Usage

- dependencies

Table 1: quollr datasets

data	explanation
s_curve_noise	Simulated 3D S-curve data with additional four noise dimensions.
s_curve_noise_training	Training data derived from S-curve data.
s_curve_noise_test	Test data derived from S-curve data.
s_curve_noise_umap	UMAP 2D embedding data of S-curve data (n_neighbors: 15, min_dist: 0.1).

```
library(tools)
package_dependencies("quollr")
```

- basic example

Datasets

The `quollr` package comes with several data sets that load with the package. These are described in Table 1.

Compute hexagonal bin configurations

Hexagonal binning is a powerful technique for visualizing the density of data points in a 2-d space. Unlike traditional rectangular bins, hexagonal bins offer several advantages, including a more uniform representation of density and reduced visual bias. However, to effectively do the hexagonal binning, it's essential to compute the appropriate configurations based on the characteristics of the dataset.

Before computing hexagonal bin configurations, we need to determine the range of data along the x and y axes to establish the boundary for hexagonal binning. Additionally, choosing an appropriate hexagon size (the radius of the outer circle) is essential. By default, the `calculate_effective_x_bins()` and `calculate_effective_y_bins()` functions use a hexagon size of 1.07457. However, users can adjust the hexagon size to fit the data range and achieve regular hexagons without overlapping.

The hexagon size directly affects the number of bins generated. A higher hexagonal size will result in fewer bins, while a lower hexagonal size will lead to more bins. Therefore, there's always a trade-off depending on the dataset used. Users should consider their specific data characteristics when selecting the hexagon size.

```
num_bins_x <- calculate_effective_x_bins(.data = s_curve_noise_umap,
                                         x = "UMAP1", hex_size = 0.2)
num_bins_x

#> [1] 4

num_bins_y <- calculate_effective_y_bins(.data = s_curve_noise_umap,
                                         y = "UMAP2", hex_size = 0.2)
num_bins_y

#> [1] 7
```

Generate full hex grid

Generating full hexagonal grid contains main three steps:

1. Generate all the hexagonal bin centroids

Steps:

- First compute hex grid bound values along the x and y axis and generate the all the points within the hex box
- Second for each x-value, find which y values are in the even row

- Then, shift the x values of the even rows

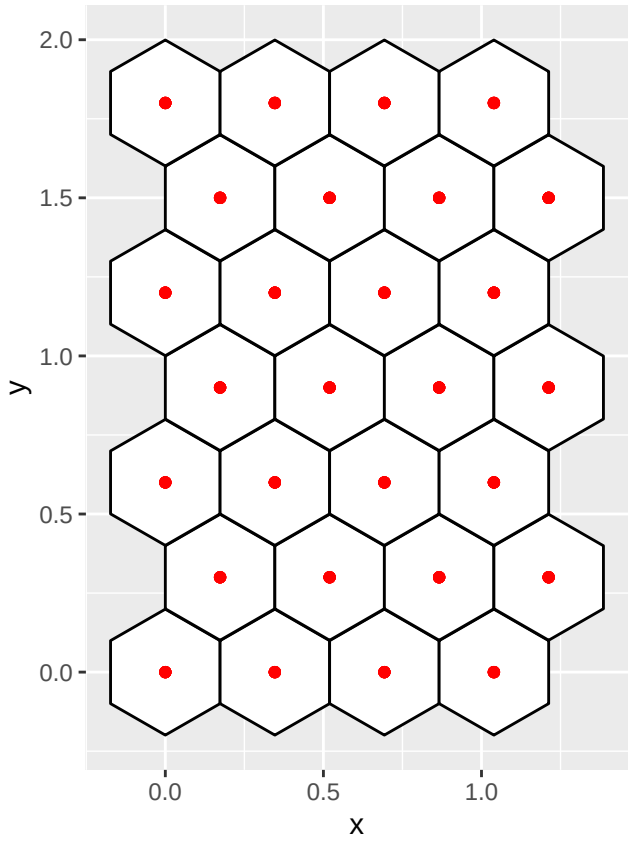
```
all_centroids_df <- generate_full_grid_centroids(nldr_df = s_curve_noise_umap,
                                                 x = "UMAP1", y = "UMAP2",
                                                 num_bins_x = num_bins_x,
                                                 num_bins_y = num_bins_y,
                                                 x_start = NA, y_start = NA,
                                                 hex_size = 0.2)

glimpse(all_centroids_df)

#> Rows: 28
#> Columns: 2
#> $ x <dbl> 0.0000000, 0.3464102, 0.6928203, 1.0392305, 0.1732051, 0.5196152, 0.~
#> $ y <dbl> 0.0, 0.0, 0.0, 0.0, 0.3, 0.3, 0.3, 0.3, 0.6, 0.6, 0.6, 0.6, 0.9, 0.9~
```

2. Generate hexagonal coordinates

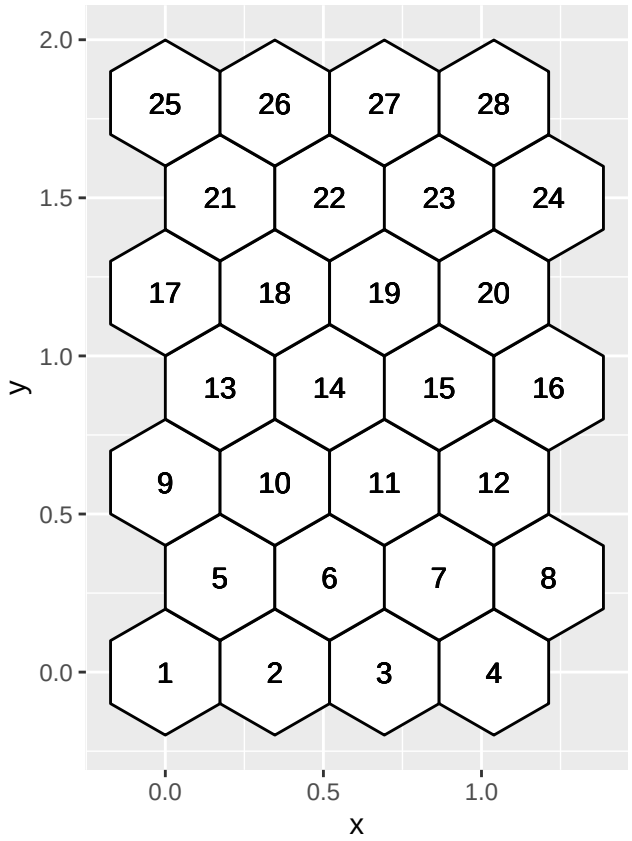
Steps: - Compute horizontal width of the hexagon

- Compute vertical width of the hexagon and multiply by a factor for overlapping ($sqrt(3)/2 * 1.15$)
- Obtain hexagon polygon coordinates
- Obtain the number of hexagons in the full grid
- Generate the coordinates for the hexagons

```
hex_grid <- gen_hex_coordinates(all_centroids_df, hex_size = 0.2)
glimpse(hex_grid)

#> Rows: 168
#> Columns: 6
#> $ c_x   <dbl> 0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000~
#> $ c_y   <dbl> 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,~
#> $ x     <dbl> 0.1732051, 0.1732051, 0.0000000, -0.1732051, -0.1732051, 0.00000~
#> $ y     <dbl> 0.09959292, -0.09959292, -0.19918584, -0.09959292, 0.09959292, 0~
#> $ id    <int> 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4~
#> $ hexID <int> 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4~

ggplot(data = hex_grid, aes(x = x, y = y)) + geom_polygon(fill = "white", color = "black", aes(group = id)) +
  geom_point(aes(x = c_x, y = c_y), color = "red") +
  coord_fixed()
```




```
full_grid_centroids_with_hexbin_id <- hex_grid |>
  dplyr::select("c_x", "c_y", "hexID") |>
    dplyr::distinct()

ggplot(data = hex_grid, aes(x = x, y = y)) + geom_polygon(fill = "white", color = "black", aes(group = id)) +
  geom_text(aes(x = c_x, y = c_y, label = hexID)) +
  coord_fixed()
```



5. Assign data into hexagons

- Compute distances between nldr coordinates and hex bin centroids
- Find the hexagonal centroid that have the minimum distance

```
s_curve_noise_umap_with_id <- assign_data(s_curve_noise_umap, full_grid_centroids_with_hexbin_id)
```

6. Compute standardized counts

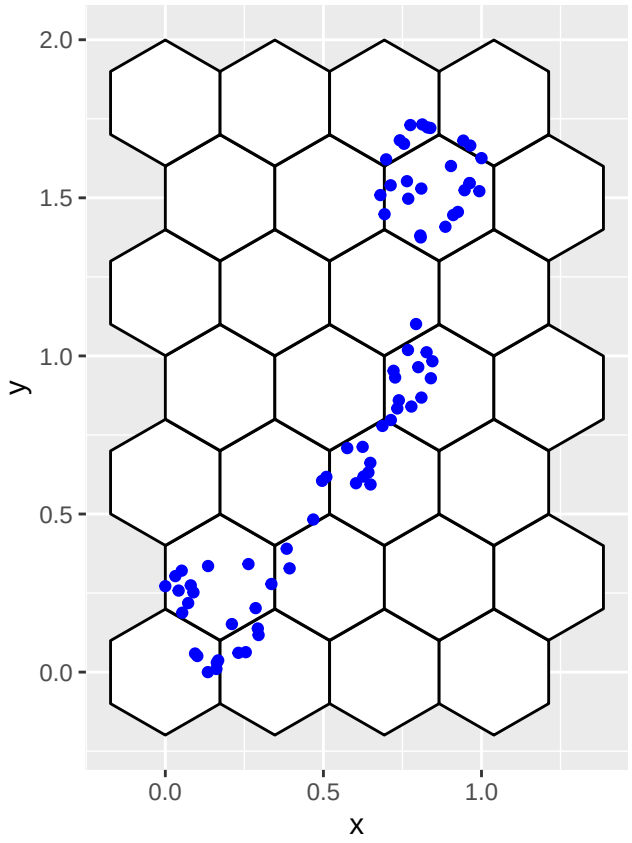
- Compute number of data points within each hexagon
- Compute standardise count by dividing the counts by the maximum

```
df_with_std_counts <- compute_std_counts(nldr_df = s_curve_noise_umap_with_id)
```

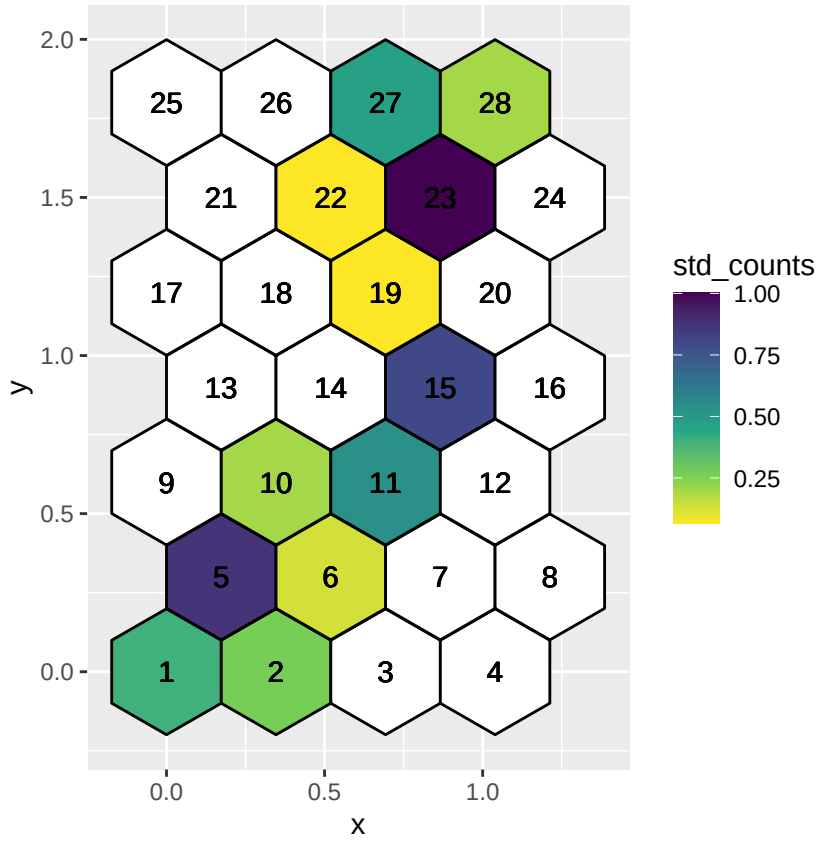
7. Extract full grid info

- Assign standardize counts for hex bins
- Join with the hexagonal coordinates

```
hex_full_count_df <- generate_full_grid_info(hex_grid, df_with_std_counts)

ggplot(data = hex_grid, aes(x = x, y = y)) + geom_polygon(fill = "white", color = "black", aes(group = id)) +
  geom_point(data = s_curve_noise_umap, aes(x = UMAP1, y = UMAP2), color = "blue") +
  coord_fixed()
```




```
ggplot(data = hex_full_count_df, aes(x = x, y = y)) +
  geom_polygon(color = "black", aes(group = id, fill = std_counts)) +
  geom_text(aes(x = c_x, y = c_y, label = hexID)) +
  scale_fill_viridis_c(direction = -1, na.value = "#ffffff") +
  coord_fixed()
```



Buffer size When generating hexagonal bins in R, a buffer is often included to ensure that the data points are evenly distributed within the bins and to prevent edge effects. The buffer helps in two main ways:

1. **Preventing Edge Effects**: Without a buffer, the outermost data points might fall near the boundary of the hexagonal grid, leading to incomplete bins or uneven distribution of data. By adding a buffer, you create a margin around the outer edges of the grid, ensuring that all data points are fully enclosed within the bins.
2. **Ensuring Even Distribution**: The buffer allows for a smoother transition between adjacent bins. This helps in cases where data points are not perfectly aligned with the grid lines, ensuring that each data point is assigned to the nearest bin without bias towards any specific direction.

Overall, including a buffer when generating hexagonal bins helps to produce more accurate and robust binning results, particularly when dealing with real-world data that may have irregular distributions or boundary effects.

Construct the 2D model with different options

```
df_bin_centroids <- hex_full_count_df[complete.cases(hex_full_count_df[["std_counts"]]), ] |>
  dplyr::select("c_x", "c_y", "hexID", "std_counts") |>
  dplyr::distinct() |>
  dplyr::rename(c("x" = "c_x", "y" = "c_y"))

df_bin_centroids
```

```
#>            x   y hexID std_counts
#> 1  0.0000000 0.0     1 0.40000000
#> 2  0.3464102 0.0     2 0.26666667
#> 3  0.1732051 0.3     5 0.86666667
#> 4  0.5196152 0.3     6 0.13333333
#> 5  0.3464102 0.6    10 0.20000000
#> 6  0.6928203 0.6    11 0.53333333
#> 7  0.8660254 0.9    15 0.80000000
#> 8  0.6928203 1.2    19 0.06666667
#> 9  0.5196152 1.5    22 0.06666667
#> 10 0.8660254 1.5    23 1.00000000
#> 11 0.6928203 1.8    27 0.46666667
#> 12 1.0392305 1.8    28 0.20000000
```

Construct the high-D model with different options

```
## To generate a data set with high-D and 2D training data
df_all <- training_data |> dplyr::select(-ID) |>
  dplyr::bind_cols(s_curve_noise_umap_with_id)

## To generate averaged high-D data

df_bin <- avg_highD_data(.data = df_all, column_start_text = "x") ## Need to pass ID column name
```

Generate the triangular mesh

```
tr1_object <- triangulate_bin_centroids(df_bin_centroids, x = "x", y = "y")
tr_from_to_df <- generate_edge_info(triangular_object = tr1_object)
```

Compute parameter defaults

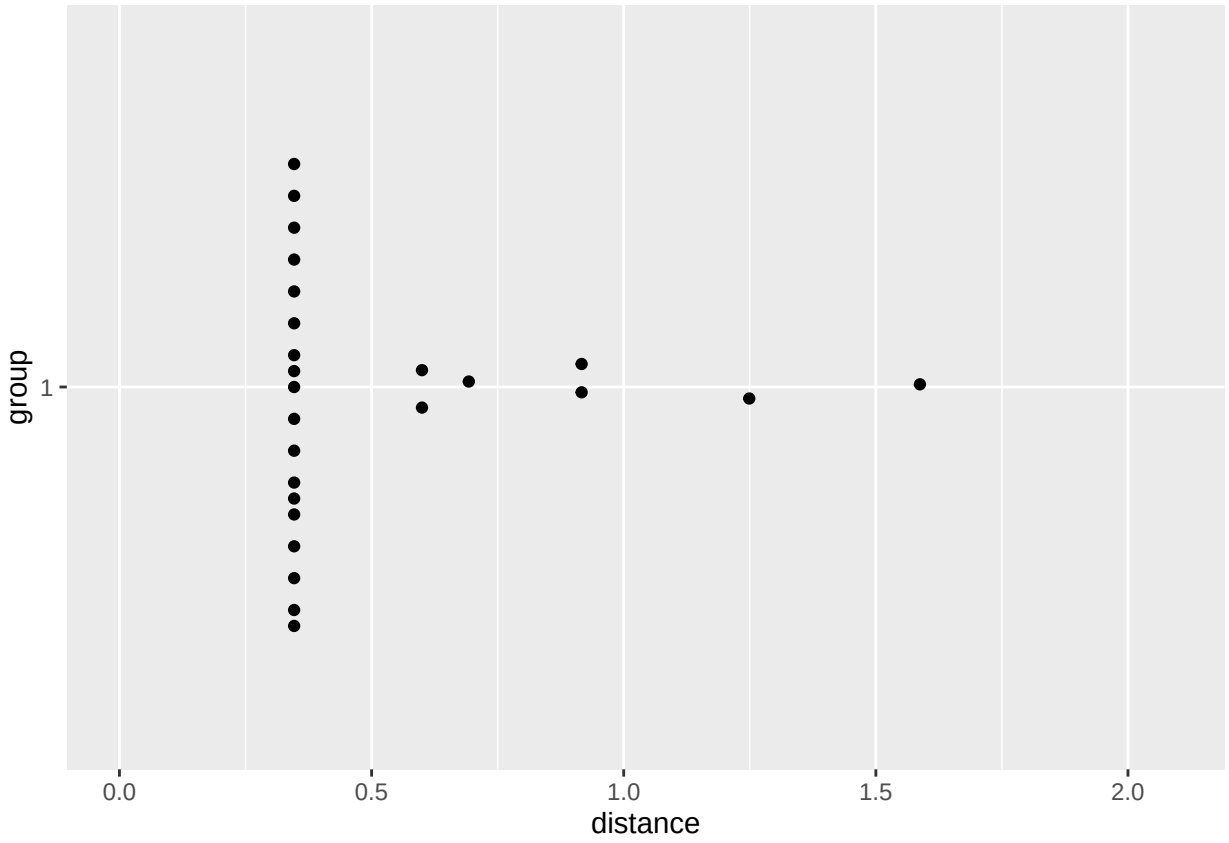
Shift the hexagonal grid origin If shift_x happen to the positive direction of x it should input as a positive value, if not other way If shift_y happen to the positive direction of y it should input as a positive value, if not other way

1. Assign shift along the x and y axis (limited the amount should less than the cell_diameter)
2. Generate bounds with shift origin

Benchmark value to remove the low-density hexagons

Benchmark value to remove the long edges

```
## Compute 2D distances
distance <- cal_2d_dist(tr_from_to_df_coord = tr_from_to_df)

## To plot the distribution of distance
plot_dist <- function(distance_df){
  distance_df$group <- "1"
  dist_plot <- ggplot(distance_df, aes(x = group, y = distance)) +
    geom_quasirandom()+
    ylim(0, max(unlist(distance_df$distance))+ 0.5) + coord_flip()
  return(dist_plot)
}

plot_dist(distance)
```




```
benchmark <- find_benchmark_value(distance_edges = distance, distance_col = "distance")
benchmark

#> [1] 0.6
```

Model function

`fit_high_d_model()` function is used to generate the 2D model and the high-D model.

```
## To generate a data set with high-D and 2D training data
df_all <- training_data |> dplyr::select(-ID) |>
  dplyr::bind_cols(s_curve_noise_umap_with_id)

## To generate averaged high-D data

df_bin <- avg_highD_data(.data = df_all, column_start_text = "x") ## Need to pass ID column name
```

Predict 2D embeddings

To predict the 2D embeddings for a new data point using the trained high-D model, we follow a series of steps outlined below:

1. **Compute high-dimensional Euclidean distance**: Calculate the Euclidean distance between each new data point and the lift model's coordinates in the high-dimensional space. This distance metric helps identify the nearest lift model coordinates to each new data point.
2. **Find the nearest lift model coordinates**: Determine the lift model's high-dimensional coordinates that are closest to each new data point based on the computed distances. This step helps establish a correspondence between the new data points and the lift model's representation in the high-dimensional space.
3. **Map the hexagonal bin ID**: Assign each new data point to a hexagonal bin based on its nearest lift model coordinates. This mapping ensures that each data point is associated with a specific region in the hexagonal grid representation of the high-dimensional space.
4. **Map the hexagonal bin centroid coordinates in 2D**: Transform the hexagonal bin centroid coordinates from the high-dimensional space to the 2D embedding space using the trained lift model. This mapping provides the 2D embeddings for the new data points, allowing them to be visualized and analyzed in a lower-dimensional space.

The `predict_2d_embeddings()` function is used to predict 2D embeddings for a new data point. The inputs for the function are test data, bin centroid coordinates in 2D, lifting model coordinates in high-D, and the type of the NLDR technique. The output contains the predicted 2D embeddings along with the predicted hexagonal id and the ID of the test data.

```
pred_df_test <- predict_2d_embeddings(test_data = training_data,
df_bin_centroids = df_bin_centroids, df_bin = df_bin, type_NLDR = "UMAP")

glimpse(pred_df_test)

#> Rows: 75
#> Columns: 4
#> $ pred_UMAP_1 <dbl> 0.1732051, 0.8660254, 0.8660254, 0.0000000, 0.5196152, 0.6~
#> $ pred_UMAP_2 <dbl> 0.3, 0.9, 0.9, 0.0, 0.3, 1.8, 0.9, 0.6, 0.9, 1.8, 0.6, 0.3~
#> $ ID          <int> 1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 14, 15, 16, 17, 19, 20, 21~
#> $ pred_hb_id  <int> 5, 15, 15, 1, 6, 27, 15, 11, 15, 28, 11, 5, 23, 5, 11, 22,~
```

Making summaries

There are two important summaries that should be made when constructing the model on a dataset using a specific NLDR technique, the Mean Square Error (MSE) and Akaike Information Criterion (AIC) which measure prediction accuracy. The function that perform the summaries called `generate_eval_df()`. The output of this function is a list which contains MSE and AIC. The code below uses `generate_eval_df()` to find the MSE and the AIC values for the model constructed on UMAP 2D embeddings generated from the `s_curve_noise` training data.

```
generate_summary(test_data = training_data, prediction_df = pred_df_test,
                 df_bin = df_bin, col_start = "x")

#> $mse
#> [1] 0.2612147
#>
#> $aic
#> [1] -536.7666
```

Visualizations

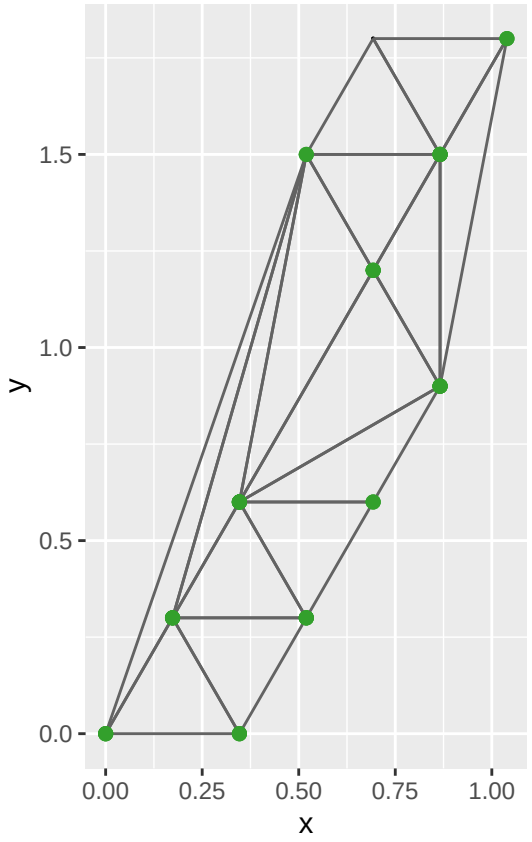
We use static visualizations to understand the model constructed in 2D. On the other hand, dynamic visualization is used to see how the model looks in high-D space.

Static visualizations Static visualizations main involves two types of results. One is the triangulation and the other is the long edge removal. Both types of visualizations provide ggplot objects.

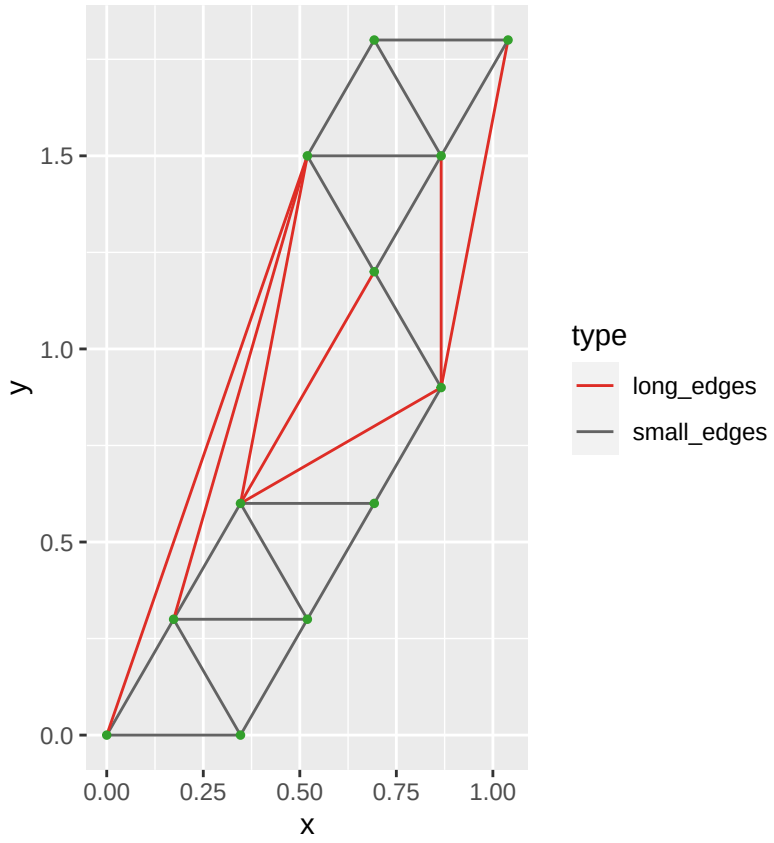
- Triangulation result: To visualize the results of triangulation, we input a dataset containing hexagonal bin centroid coordinates where 2D embedding data exists. `geom_trimesh()` is used to visualize this result.
- Long edge removal: The long edge removal process involves identifying and removing long edges from the triangular mesh. Table shows the main arguments of the functions. We offer two functions for visualizing this process:
- `colour_long_edges()`: This function colors the long edges within the triangular mesh by red.
- `remove_long_edges()`: After identifying long edges, this function draws the triangular mesh without the long edges.

Dynamic visaulizations The `show_langevitour()` function enables dynamic visualization of the 2D model alongside the high-dimensional (high-D) data in its original space. This visualization is facilitated by langevitour object, allowing users to interactively explore the relationship between the 2D embeddings and the underlying high-dimensional data. The main arguments are shown in Table

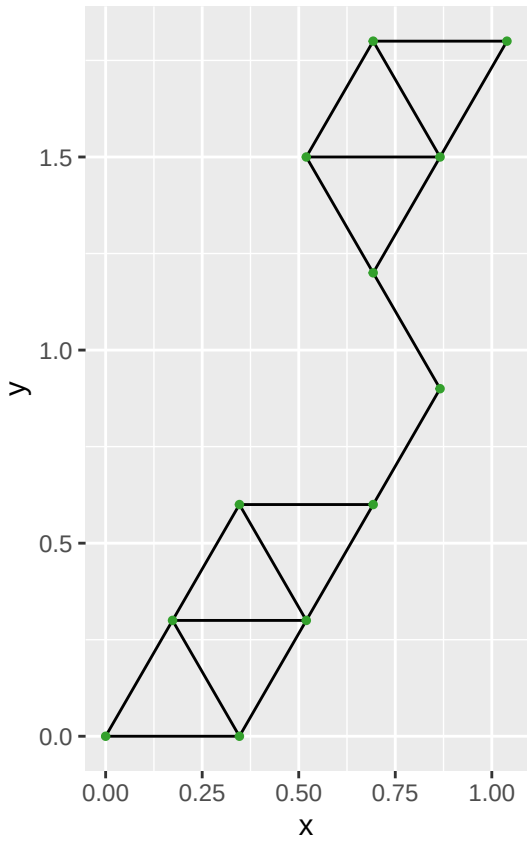
```
trimesh <- ggplot(df_bin_centroids, aes(x = x, y = y)) +
  geom_point(size = 0.1) +
  geom_trimesh() +
  coord_equal()

trimesh
```




```
trimesh_gr <- colour_long_edges(distance_edges = distance, benchmark_value = benchmark,
                        tr_from_to_df_coord = tr_from_to_df, distance_col = "distance")

trimesh_gr
```




```
trimesh_removed <- remove_long_edges(distance_edges = distance, benchmark_value = benchmark,
                          tr_from_to_df_coord = tr_from_to_df, distance_col = "distance")
trimesh_removed
```




```
tour1 <- show_langevitour(df_all, df_bin, df_bin_centroids,
```

```
                          benchmark_value = benchmark,
                          distance = distance, distance_col = "distance",
                          use_default_benchmark_val = FALSE,
                          column_start_text = "x")
tour1
```

Find non-empty bins

Tests

All functions have tests written and implemented using the **testthat** (Wickham 2011) in R.

3 Application

4 Conclusion

5 Acknowledgements

This article is created using **knitr** (Xie 2015) and **rmarkdown** (Xie, Allaire, and Grolemund 2018) in R with the `rjtools::rjournal_article` template. The source code for reproducing this paper can be found at: https://github.com/JayaniLakshika/paper-quollr.

References

Wickham, Hadley. 2011. "Testthat: Get Started with Testing." *The R Journal* 3: 5–10. https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf.

Xie, Yihui. 2015. *Dynamic Documents with R and Knitr*. 2nd ed. Boca Raton, Florida: Chapman; Hall/CRC. https://yihui.name/knitr/.

Xie, Yihui, J. J. Allaire, and Garrett Grolemund. 2018. *R Markdown: The Definitive Guide*. Boca Raton, Florida: Chapman; Hall/CRC. https://bookdown.org/yihui/rmarkdown.

Jayani P.G. Lakshika
Monash University
Department of Econometrics and Business Statistics, VIC 3800 Australia
https://jayanilakshika.netlify.app/
ORCiD: 0000-0002-6265-6481
jayani.piyadigamage@monash.edu

Dianne Cook
Monash University
Department of Econometrics and Business Statistics, VIC 3800 Australia
http://www.dicook.org/
ORCiD: 0000-0002-3813-7155
dicook@monash.edu

Paul Harrison
Monash University
MGBP, BDInstitute, VIC 3800 Australia
ORCiD: 0000-0002-3980-268X
paul.harrison@monash.edu

Michael Lydeamore
Monash University
Department of Econometrics and Business Statistics, VIC 3800 Australia
ORCiD: 0000-0001-6515-827X
michael.lydeamore@monash.edu

Thiyanga S. Talagala
University of Sri Jayewardenepura
Department of Statistics, Gangodawila, Nugegoda 10100 Sri Lanka
https://thiyanga.netlify.app/
ORCiD: 0000-0002-0656-9789
ttalagala@sjp.ac.lk