Setting up TDA

Introduction

Recall the broad steps of TDA:

1. Start with the data. Apply a filter function.
2. Create overlapping bins of the data using the image of the filter function.
3. Cluster the data in the overlapping bins of the data (aka the pre-image).
4. Create a directed graph.

We must choose the following methods and parameters. These correspond to the first three steps of TDA.

1. The method for scaling the data and the filter function
2. Parameters or (a) number of bins and (b) percent overlap of the bins
3. The clustering algorithm and the parameters

We already decided on PCA2 for our filter function because...

Summary

This notebook is about selecting the methods and parameters within TDA. Specifically, we go through

1. Scaling the data -- we decide on Robust Scaling over Standard Scaling
2. Clustering -- we decide on DBscan clustering and the parameters epsilon = xx and min_sample_size = xx.
3. TDA parameters -- we decide on n_cubes = xx and perc_overlap = xx.

Heuristics of parameter and method selection

1. Each cluster has at most 10% of the data
2. The Mapper output represents at least 90% of the data (no more than 10% of the original data, aka 7600 data points, are lost as noise).

The print statements about the maximum data points in a node and number of unique samples address each of these heuristics, repsectively.

Notes

- COND is removed in this analysis.
-

Load libraries

In [1]:
```
import kmapper as km
# import sklearn

from sklearn.cluster import DBSCAN # clustering algorithm
from sklearn.decomposition import PCA # projection (lens) creation
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.compose import ColumnTransformer

import hdbscan

# from sklearn import ensemble
# from sklearn.manifold import MDS

import plotly.graph_objs as go
# from ipywidgets import interactive, HBox, VBox, widgets, interact # ?
# import dash_html_components as html # ?
# import dash_core_components as dcc # ?

from kmapper.plotlyviz import * # static and interactive plots
import psutil # for plotlyviz
import kaleido # for plotlyviz
# import networkx # ?

# import dash # ?
import warnings #?
warnings.filterwarnings("ignore")

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
import json as js
import pickle as pk
```

Read data

In [3]:

```
water20 = pd.read_csv("../../LTRM data/RF interpolation/water_full.csv")
```

In [4]:

```
water20.head()
```

Out[4]:

	SHEETBAR	DATE	LATITUDE	LONGITUDE	FLDNUM	STRATUM	LOCATCD	TN	TP	TEMP	DO	TURB	COND	VEL	SS	WDP	CH
0	41000065	07/26/1993	44.571864	-92.510970	Lake City, MN	Main channel	9312103	3.955	0.228	23.0	6.6	28	550.0	0.50	42.3	2.2	9.4
1	41000066	07/26/1993	44.575497	-92.518497	Lake City, MN	Main channel	9312002	4.876	0.229	23.0	6.6	28	554.0	0.72	37.6	8.2	8.2
2	41000067	07/26/1993	44.573718	-92.523549	Lake City, MN	Main channel	9312102	3.955	0.220	22.9	6.3	24	564.0	0.66	34.1	4.3	8.7
3	41000068	07/26/1993	44.566588	-92.541238	Lake City, MN	Main channel	9312003	4.257	0.212	22.9	6.4	28	563.0	0.69	33.4	9.1	8.4
4	41000069	07/26/1993	44.568419	-92.548780	Lake City, MN	Main channel	9312104	4.030	0.237	23.0	6.6	33	556.0	0.68	48.0	6.7	9.5

Scaling

Standard scaling uses mean and standard deviation. Robust scaling uses median and IQR.

The motivation for scaling the data prior to running TDA is that the units of each continuous variable is different. Thus, the ranges are different. For example, SECCHI and SS have ranges from 0 to 120, while VEL and TP all have values less than 20.

It is beneficial to have continuous variables in the same "units" by subtracting the mean or median and dividing by the standard deviation or IQR.

First, it is beneficial for PCA, which finds axes of maximum variance. Without standardizing the variables, SECCHI was consistently a PCA... xx.

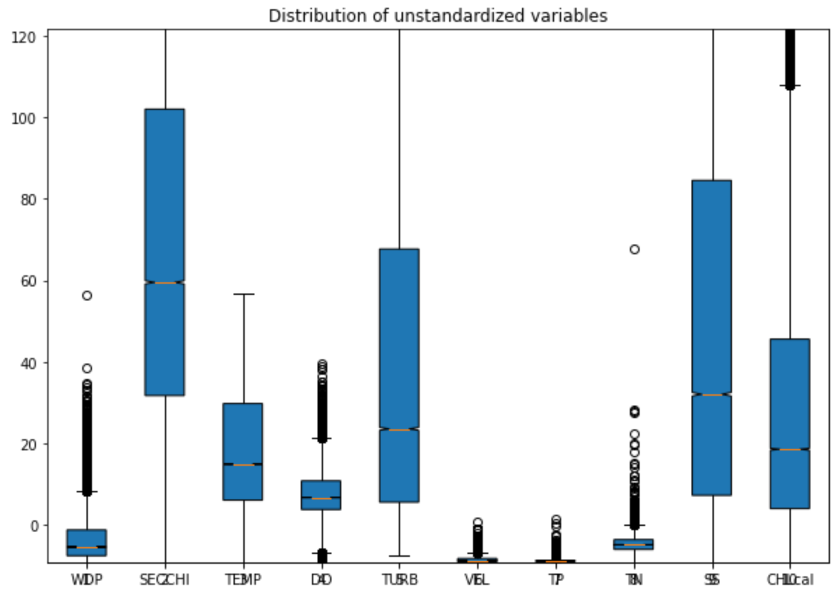
Clustering algorithms use some form of distance metric, so standardizing is also helpful here. "If one of your features has a range of values much larger than the others, clustering will be completely dominated by that one feature." from this Stack Overflow thread. (If we were to cluster data on weight and height, we would want to cluster by kilograms and meters but not grams and meters.)

We let X denote the data with robust scaling and Z denote the data with standard scaling.

We choose robust scaling over standard scaling because the distributions among the continuous variables are more similar with robust scaling.

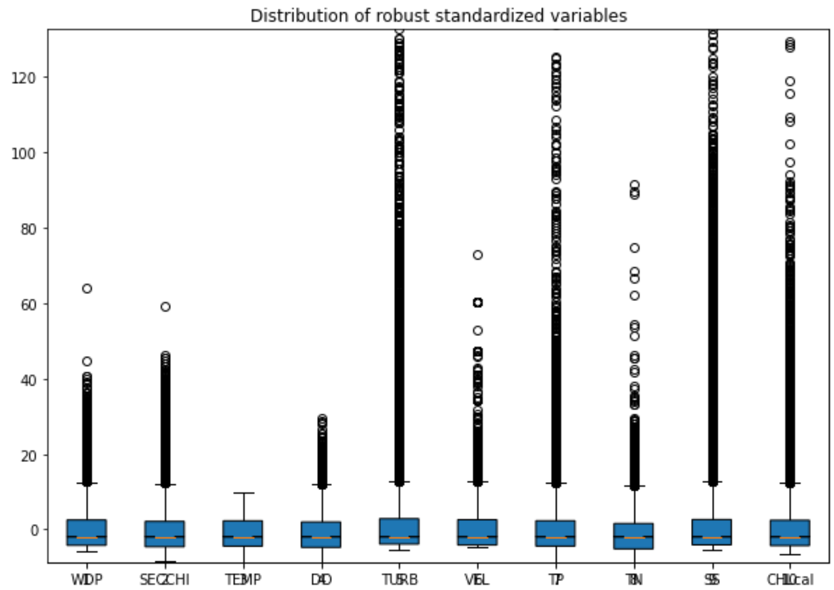
In [38]:

```
fig = plt.figure(figsize =(10, 7))

# Creating plot
plt.boxplot(water20[["WDP", "SECCHI", "TEMP", "DO", "TURB",
             "VEL", "TP", "TN", "SS", "CHLcal"]])
ax = fig.add_subplot(111)

bp = ax.boxplot(water20[["WDP", "SECCHI", "TEMP", "DO", "TURB",
             "VEL", "TP", "TN", "SS", "CHLcal"]], patch_artist = True,
                notch ='True', vert = 1)


ax.set_xticklabels(["WDP", "SECCHI", "TEMP", "DO", "TURB",
                    "VEL", "TP", "TN", "SS", "CHLcal"])

ax.set_ylim(0, 80)
ax.get_yaxis().set_visible(False)

#ax.set_yticklables([-5, 0, 5, 10, 15, 20])

plt.title("Distribution of unstandardized variables")


# show plot
plt.show(bp)
```



Robust scaling

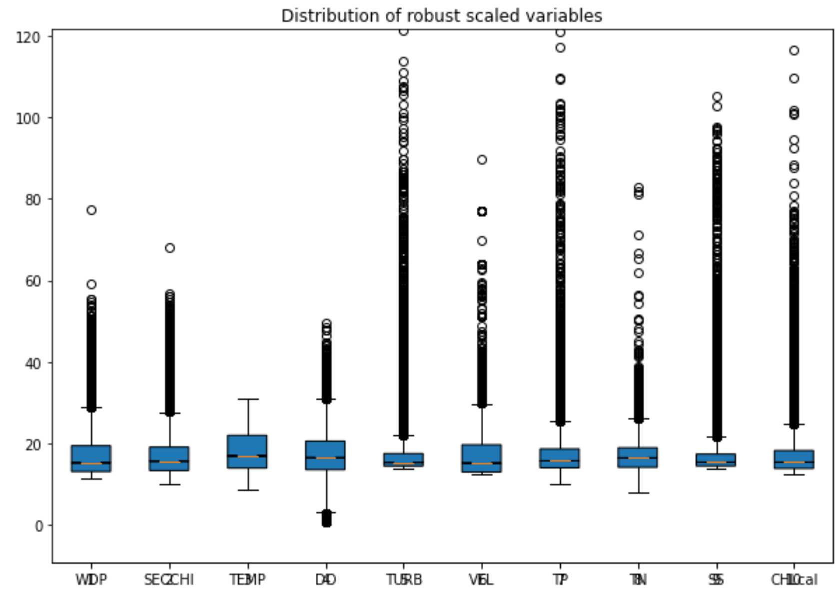
In [55]:
```
X = water20[["WDP", "SECCHI", "TEMP", "DO", "TURB",
             "VEL", "TP", "TN", "SS", "CHLcal",
             "YEAR", "SEASON", "FLDNUM", "STRATUM"]]

ct = ColumnTransformer([
        ('somename', RobustScaler(), ["WDP", "SECCHI", "TEMP", "DO", "TURB",
                                      "VEL", "TP", "TN", "SS", "CHLcal"])
    ], remainder='passthrough')

X = pd.DataFrame(ct.fit_transform(X), columns = ["WDP", "SECCHI", "TEMP", "DO", "TURB",
                                                 "VEL", "TP", "TN", "SS", "CHLcal",
                                                 "YEAR", "SEASON", "FLDNUM", "STRATUM"])

X = pd.DataFrame(X, columns = ["WDP", "SECCHI", "TEMP", "DO", "TURB",
                               "VEL", "TP", "TN", "SS", "CHLcal"])

fig = plt.figure(figsize =(10, 7))

# Creating plot
plt.boxplot(X)
ax = fig.add_subplot(111)

bp = ax.boxplot(X, patch_artist = True,
                notch ='True', vert = 1)


ax.set_xticklabels(["WDP", "SECCHI", "TEMP", "DO", "TURB",
                    "VEL", "TP", "TN", "SS", "CHLcal"])

ax.set_ylim(-1, 20)
ax.get_yaxis().set_visible(False)

#ax.set_yticklables([-5, 0, 5, 10, 15, 20])

plt.title("Distribution of robust standardized variables")


# show plot
plt.show(bp)
```



Standard scaling

In [56]:

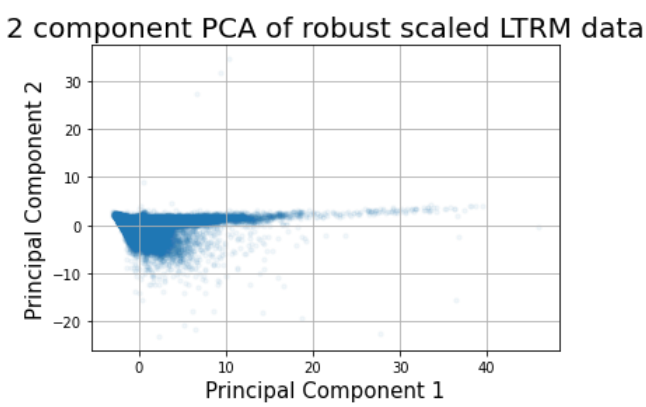
```
Z = water20[["WDP", "SECCHI", "TEMP", "DO", "TURB",
             "VEL", "TP", "TN", "SS", "CHLcal",
             "YEAR", "SEASON", "FLDNUM", "STRATUM"]]

ct = ColumnTransformer([
        ('somename', StandardScaler(), ["WDP", "SECCHI", "TEMP", "DO", "TURB",
                                        "VEL", "TP", "TN", "SS", "CHLcal"])
    ], remainder='passthrough')

Z = pd.DataFrame(ct.fit_transform(Z), columns = ["WDP", "SECCHI", "TEMP", "DO", "TURB",
                                                 "VEL", "TP", "TN", "SS", "CHLcal",
                                                 "YEAR", "SEASON", "FLDNUM", "STRATUM"])

Z = pd.DataFrame(Z, columns = ["WDP", "SECCHI", "TEMP", "DO", "TURB",
                               "VEL", "TP", "TN", "SS", "CHLcal"])


fig = plt.figure(figsize =(10, 7))

# Creating plot
plt.boxplot(Z)
ax = fig.add_subplot(111)

bp = ax.boxplot(Z, patch_artist = True,
                notch ='True', vert = 1)


ax.set_xticklabels(["WDP", "SECCHI", "TEMP", "DO", "TURB",
                    "VEL", "TP", "TN", "SS", "CHLcal"])

ax.set_ylim(-5, 20)
ax.get_yaxis().set_visible(False)

#ax.set_yticklables([-5, 0, 5, 10, 15, 20])

plt.title("Distribution of robust scaled variables")


# show plot
plt.show(bp)
```



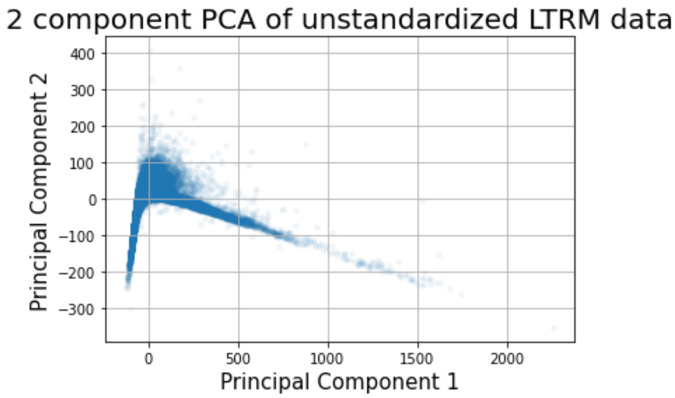
Filter function: PCA2 projection

We can see that the ranges of PCA1 and PCA2 are within much more reasonable ranges after scaling. The unscaled PCA has

In [57]:
```
pca = PCA(n_components = 2)
lens = pca.fit_transform(X)

fig, ax = plt.subplots()
ax.scatter(lens[:,0], lens[:,1], s = 10, alpha = 0.047)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 component PCA of robust scaled LTRM data', fontsize = 20)
ax.grid(True)

plt.show()
```



In [58]:
```
pca = PCA(n_components = 2)
lens = pca.fit_transform(water20[["WDP", "SECCHI", "TEMP", "DO", "TURB",
                                  "VEL", "TP", "TN", "SS", "CHLcal"]])

fig, ax = plt.subplots()
ax.scatter(lens[:,0], lens[:,1], s = 10, alpha = 0.047)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 component PCA of unstandardized LTRM data', fontsize = 20)
ax.grid(True)

plt.show()
```



Define mapper functions

The first one uses DBscan as the clustering algorithm. The second one uses HDBscan as the clustering algorithm.

In [19]:

```
def mapper_pca2_db(df, DBSCAN_EPSILON = 10, DBSCAN_MIN_SAMPLES = 20,
                   N_CUBES = [10,10], PERC_OVERLAP = [.25,.25], return_with_df = False, print_pca_info = False):
    """
    """

    X = df[["WDP", "SECCHI", "TEMP", "DO", "TURB",
            "VEL", "TP", "TN", "SS", "CHLcal"]]

    # for discerning primary variables in PCA
    continuous_variables = ["WDP", "SECCHI", "TEMP", "DO", "TURB",
                            "VEL", "TP", "TN", "SS", "CHLcal"]

    var_to_index = {continuous_variables[i] : i for i in range(len(continuous_variables))}
    projected_vars = continuous_variables
    projected_var_indices = [var_to_index[var] for var in projected_vars]

#     if X.shape[0]<10:
#         #print(X)
#         print("Not enough data in ", title, "_size = ", X.shape[0])
#         return(X.shape[0])

    # to match up indices in scomplex with the original dataframe X
    X.reset_index(drop = True, inplace = True)

    # create instance of clustering alg
    cluster_alg = DBSCAN(eps = DBSCAN_EPSILON, min_samples = DBSCAN_MIN_SAMPLES,
                         metric='euclidean')

    # instantiate kepler mapper object
    mapper = km.KeplerMapper(verbose = 0)

    # defining filter function as projection on to the first 2 component axis
    pca = PCA(n_components = 2)
    lens = pca.fit_transform(X)

    if print_pca_info:
        for j in range(2):
            pc_j = pca.components_[j]
            largest_magnitude = max(abs(pc_j))
            idx_magnitude = np.where(abs(pc_j) == largest_magnitude)[0][0]

            print("*** PCA", j+1, " ***")
            print("Primary variable: ", continuous_variables[idx_magnitude])
            print("Corresponding component: ", pc_j[idx_magnitude])
            print("Explained variance: ", pca.explained_variance_ratio_[j])

    summary_variable = mapper.project(np.array(X), projection=projected_var_indices, scaler=None)
    # similar to fit transform

    # Generate the simplicial complex
    scomplex = mapper.map(lens, X,
                          cover=km.Cover(n_cubes = N_CUBES, perc_overlap = PERC_OVERLAP),
                          clusterer = cluster_alg)

    if return_with_df:
        return(scomplex, X)

    return(scomplex)
```

In [18]:

```
def mapper_pca2_hdb(df, HDB_MIN_CLUSTER = 45, HDB_MIN_SAMPLES = 10, HDB_EPSILON = 1,
```

```
                    N_CUBES = [10,10], PERC_OVERLAP = [.25,.25], return_with_df = False, print_pca_info = False):
    """
    """

    X = df[["WDP", "SECCHI", "TEMP", "DO", "TURB",
            "VEL", "TP", "TN", "SS", "CHLcal"]]

    # for discerning primary variables in PCA
    continuous_variables = ["WDP", "SECCHI", "TEMP", "DO", "TURB",
                            "VEL", "TP", "TN", "SS", "CHLcal"]

    var_to_index = {continuous_variables[i] : i for i in range(len(continuous_variables))}
    projected_vars = continuous_variables
    projected_var_indices = [var_to_index[var] for var in projected_vars]

#     if X.shape[0]<10:
#         #print(X)
#         print("Not enough data in ", title, "_size = ", X.shape[0])
#         return(X.shape[0])

    # to match up indices in scomplex with the original dataframe X
    X.reset_index(drop = True, inplace = True)

    # create instance of clustering alg
    cluster_alg = hdbscan.HDBSCAN(min_cluster_size = HDB_MIN_CLUSTER, min_samples = HDB_MIN_SAMPLES,
                                  cluster_selection_epsilon= HDB_EPSILON, cluster_selection_method = 'eom')

    # instantiate kepler mapper object
    mapper = km.KeplerMapper(verbose = 0)

    # defining filter function as projection on to the first 2 component axis
    pca = PCA(n_components = 2)
    lens = pca.fit_transform(X)

    if print_pca_info:
        for j in range(2):
            pc_j = pca.components_[j]
            largest_magnitude = max(abs(pc_j))
            idx_magnitude = np.where(abs(pc_j) == largest_magnitude)[0][0]

            print("*** PCA", j+1, " ***")
            print("Primary variable: ", continuous_variables[idx_magnitude])
            print("Corresponding component: ", pc_j[idx_magnitude])
            print("Explained variance: ", pca.explained_variance_ratio_[j])

    summary_variable = mapper.project(np.array(X), projection=projected_var_indices, scaler=None)
    # similar to fit transform

    # Generate the simplicial complex
    scomplex = mapper.map(lens, X,
                          cover=km.Cover(n_cubes = N_CUBES, perc_overlap = PERC_OVERLAP),
                          clusterer = cluster_alg)

    if return_with_df:
        return(scomplex, X)

    return(scomplex)
```

Clustering in TDA

Running DBScan and TDA

TDA parameters

In [74]:

```
n_cubes_lst = []

for n in [100, 125, 150]:
    n_cubes_lst.append([n, n])

perc_overlap_lst = []

for perc in [0.4, 0.5]:
    perc = round(perc, 2)
    perc_overlap_lst.append([perc, perc])

# tda params
print(n_cubes_lst) # need higher cubes after standardizing
print(perc_overlap_lst) # redundancy is okay
```

```
[[100, 100], [125, 125], [150, 150]]
[[0.4, 0.4], [0.5, 0.5]]
```

DBScan parameters

In [22]:

```python
# dbscan parameters

eps_lst = [0.9, 1] # variables are standardized
min_samples_lst = [10]
print(eps_lst)
print(min_samples_lst)
```

```
[0.8, 0.9, 1]
[10]
```

In [75]:

```python
db_params = []
db_scomplex = []

for epsilon in eps_lst:

    for min_samples in min_samples_lst: # min_samples = 10

        for n_cubes in n_cubes_lst:

            for perc in perc_overlap_lst: # perc_overlap = [0.5, 0.5]

                db_params.append('epsilon = ' + str(epsilon) + ', min_samples = ' + str(min_samples) +
                                 ', n_cubes = ' + str(n_cubes) + ", and perc_overlap = " + str(perc))

                scomplex, df = mapper_pca2_db(X, DBSCAN_EPSILON = epsilon, DBSCAN_MIN_SAMPLES = min_samples,
                                              N_CUBES = n_cubes, PERC_OVERLAP = perc, return_with_df = True)

                db_scomplex.append(scomplex)

                idx = len(db_params)-1

                print("*** ", idx, " ***")
                print(db_params[idx])

                all_nodes = db_scomplex[idx].get('nodes')

                obsv_per_node = []

                for node in all_nodes:
                    obsv_per_node.append(len(all_nodes.get(node)))

                print("The maximum data points in a node is ", max(obsv_per_node))
                print("The minimum data points in a node is ", min(obsv_per_node))
                print("The number of unique samples is ", get_mapper_graph(db_scomplex[idx])[1]["n_unique"])
```

```
***  0  ***
epsilon = 0.8, min_samples = 10, n_cubes = [100, 100], and perc_overlap = [0.4, 0.4]
The maximum data points in a node is  10752
The minimum data points in a node is  4
The number of unique samples is  69065
***  1  ***
epsilon = 0.8, min_samples = 10, n_cubes = [100, 100], and perc_overlap = [0.5, 0.5]
The maximum data points in a node is  14690
The minimum data points in a node is  4
The number of unique samples is  69704
***  2  ***
epsilon = 0.8, min_samples = 10, n_cubes = [125, 125], and perc_overlap = [0.4, 0.4]
The maximum data points in a node is  7057
The minimum data points in a node is  7
The number of unique samples is  68265
***  3  ***
epsilon = 0.8, min_samples = 10, n_cubes = [125, 125], and perc_overlap = [0.5, 0.5]
The maximum data points in a node is  9873
The minimum data points in a node is  4
The number of unique samples is  69285
***  4  ***
epsilon = 0.8, min_samples = 10, n_cubes = [150, 150], and perc_overlap = [0.4, 0.4]
The maximum data points in a node is  5083
The minimum data points in a node is  3
The number of unique samples is  67227
***  5  ***
epsilon = 0.8, min_samples = 10, n_cubes = [150, 150], and perc_overlap = [0.5, 0.5]
The maximum data points in a node is  7021
The minimum data points in a node is  3
The number of unique samples is  68621
***  6  ***
epsilon = 0.9, min_samples = 10, n_cubes = [100, 100], and perc_overlap = [0.4, 0.4]
The maximum data points in a node is  10813
The minimum data points in a node is  7
The number of unique samples is  71149
***  7  ***
epsilon = 0.9, min_samples = 10, n_cubes = [100, 100], and perc_overlap = [0.5, 0.5]
The maximum data points in a node is  14757
The minimum data points in a node is  4
The number of unique samples is  71721
***  8  ***
epsilon = 0.9, min_samples = 10, n_cubes = [125, 125], and perc_overlap = [0.4, 0.4]
```

```
The maximum data points in a node is  7105
The minimum data points in a node is  1
The number of unique samples is  70477
***  9  ***
epsilon = 0.9, min_samples = 10, n_cubes = [125, 125], and perc_overlap = [0.5, 0.5]
The maximum data points in a node is  9934
The minimum data points in a node is  6
The number of unique samples is  71267
***  10  ***
epsilon = 0.9, min_samples = 10, n_cubes = [150, 150], and perc_overlap = [0.4, 0.4]
The maximum data points in a node is  5112
The minimum data points in a node is  5
The number of unique samples is  69565
***  11  ***
epsilon = 0.9, min_samples = 10, n_cubes = [150, 150], and perc_overlap = [0.5, 0.5]
The maximum data points in a node is  7060
The minimum data points in a node is  4
The number of unique samples is  70743
***  12  ***
epsilon = 1, min_samples = 10, n_cubes = [100, 100], and perc_overlap = [0.4, 0.4]
The maximum data points in a node is  10833
The minimum data points in a node is  8
The number of unique samples is  72428
***  13  ***
epsilon = 1, min_samples = 10, n_cubes = [100, 100], and perc_overlap = [0.5, 0.5]
The maximum data points in a node is  14788
The minimum data points in a node is  3
The number of unique samples is  72981
***  14  ***
epsilon = 1, min_samples = 10, n_cubes = [125, 125], and perc_overlap = [0.4, 0.4]
The maximum data points in a node is  7116
The minimum data points in a node is  7
The number of unique samples is  71822
***  15  ***
epsilon = 1, min_samples = 10, n_cubes = [125, 125], and perc_overlap = [0.5, 0.5]
The maximum data points in a node is  9949
The minimum data points in a node is  2
The number of unique samples is  72494
***  16  ***
epsilon = 1, min_samples = 10, n_cubes = [150, 150], and perc_overlap = [0.4, 0.4]
The maximum data points in a node is  5137
The minimum data points in a node is  7
The number of unique samples is  71106
***  17  ***
epsilon = 1, min_samples = 10, n_cubes = [150, 150], and perc_overlap = [0.5, 0.5]
The maximum data points in a node is  7075
The minimum data points in a node is  6
The number of unique samples is  71999
```

Parameters for DBscan, TDA mapper

With these parameters,

- Epsilon = 1
- min_samples = 10
- n_cubes = [125, 125]
- perc_overlap = [0.4, 0.4],

Each node represents at most 10% of the data, and about 5% of the data is lost as noise. The graph layout is one large connected component.

In [76]:

```
plotlyviz(db_scomplex[14], title = db_params[14],
          graph_layout='fr', dashboard = True)
# CHOSEN ONE
```

In [77]:

```
plotlyviz(db_scomplex[14], title = db_params[14],
          graph_layout='kk', dashboard = True)
```

Running HDBScan and TDA

Use the cluster selection epsilon method so that it is a hybrid of DBscan and HDBscan. We will see that HDBscan graph outputs are not as desirable because xx.

HDBScan parameters

In [25]:

```
min_cluster_lst = [10]
print(min_cluster_lst)
print(min_samples_lst)
```

```
[10]
[10]
```

In [63]:

```
# hdbscan

hdb_params = []
hdb_scomplex = []

for min_clust in min_cluster_lst: # min_clust = 10

    min_clust = int(min_clust)

    for min_samples in min_samples_lst: # min_samples = 10

        min_samples = int(min_samples)

        for n_cubes in n_cubes_lst:

            for perc in perc_overlap_lst: # perc_overlap = [0.5, 0.5]

                hdb_params.append('min_clust = ' + str(min_clust) + ', min_samples = ' + str(min_samples) +
                                 ', n_cubes = ' + str(n_cubes) + ", and perc_overlap = " + str(perc))

                hdb_scomplex.append(mapper_pca2_hdb(X, HDB_MIN_CLUSTER = min_clust, HDB_MIN_SAMPLES = min_samples,
                                                    HDB_EPSILON = 1,N_CUBES = n_cubes, PERC_OVERLAP = perc,
                                                    return_with_df = False))

                idx = len(hdb_params)-1

                print("*** ", idx, " ***")
                print(hdb_params[idx])

                all_nodes = hdb_scomplex[idx].get('nodes')

                obsv_per_node = []

                for node in all_nodes:
                    obsv_per_node.append(len(all_nodes.get(node)))

                print("The maximum data points in a node is ", max(obsv_per_node))
                print("The minimum data points in a node is ", min(obsv_per_node))
                print("The number of unique samples is ", get_mapper_graph(hdb_scomplex[idx])[1]["n_unique"])
```

```
***  0  ***
min_clust = 10, min_samples = 10, n_cubes = [100, 100], and perc_overlap = [0.5, 0.5]
The maximum data points in a node is  12639
The minimum data points in a node is  10
The number of unique samples is  67094
***  1  ***
min_clust = 10, min_samples = 10, n_cubes = [125, 125], and perc_overlap = [0.5, 0.5]
The maximum data points in a node is  8491
The minimum data points in a node is  10
The number of unique samples is  63616
***  2  ***
min_clust = 10, min_samples = 10, n_cubes = [150, 150], and perc_overlap = [0.5, 0.5]
The maximum data points in a node is  6458
The minimum data points in a node is  10
The number of unique samples is  63490
```

Deciding not to use HDBscan

xx.

In [64]:

```
plotlyviz(hdb_scomplex[0], title = hdb_params[0],
          graph_layout='fr', dashboard = True)
# chosen one
```

In [65]:

```
plotlyviz(hdb_scomplex[1], title = hdb_params[1],
          graph_layout='fr', dashboard = True) #alternative
```

Save selected parameters into .html and .json

In [68]:

```
mapper = km.KeplerMapper(verbose=0)

db_mapper = mapper.visualize(db_scomplex[8], path_html=db_params[8] + '.html')
```

In [69]:

```
db_scomplex_final, dbscan_df = mapper_pca2_db(X, DBSCAN_EPSILON = 1, DBSCAN_MIN_SAMPLES = 10,
                                        N_CUBES = [75, 75], PERC_OVERLAP = [0.5, 0.5],
                                        return_with_df = True, print_pca_info = True)
```

```
*** PCA 1  ***
```

```
Primary variable:  SS
Corresponding component:  0.6636335560275306
Explained variance:  0.5253705542119079
*** PCA 2  ***
Primary variable:  CHLcal
Corresponding component:  -0.7604895384683665
Explained variance:  0.14469860960939684
```

In [70]:
```
json = js.dumps(db_scomplex_final)
f = open("water_dbscan.json","w")
f.write(json)
f.close()
```

In [72]:
```
pk.dump(dbscan_df, open("water_dbscan_df.p","wb"))
```

Old

In [ ]:
```
#     # the rest of this is for coloring

#     pl_brewer = [[0.0, '#006837'],
#              [0.1, '#1a9850'],
#              [0.2, '#66bd63'],
#              [0.3, '#a6d96a'],
#              [0.4, '#d9ef8b'],
#              [0.5, '#ffffbf'],
#              [0.6, '#fee08b'],
#              [0.7, '#fdae61'],
#              [0.8, '#f46d43'],
#              [0.9, '#d73027'],
#              [1.0, '#a50026']]

#     color_values = lens[:,0] - lens[:,0].min() # changes if PCA1 or PCA1 and PCA2
#     # can change to other variables
#     color_function_name = ['Distance to x-min'] # set name of color function
#     my_colorscale = pl_brewer
#     kmgraph, mapper_summary, colorf_distribution = get_mapper_graph(scomplex,
#                                                                     color_values,
#                                                                     color_function_name=color_function_name,
#                                                                     colorscale=my_colorscale)

#     plotly_graph_data = plotly_graph(kmgraph, graph_layout='fr', colorscale=my_colorscale,
#                                      factor_size=2.5, edge_linewidth=0.5)

#     plot_title = str(DBSCAN_EPSILON) + str(DBSCAN_EPSILON) + ', MIN_SAMPLES ' + str(DBSCAN_MIN_SAMPLES)

#     layout = plot_layout(title=plot_title,
#                          width=620, height=570,
#                          annotation_text=get_kmgraph_meta(mapper_summary))

#     # FigureWidget is responsible for event listeners

#     fw_graph = go.FigureWidget(data=plotly_graph_data, layout=layout)
#     fw_summary = summary_fig(mapper_summary, height=300)

#     dashboard = hovering_widgets(kmgraph, fw_graph, member_textbox_width=600)

#     # DESIRED FILE PATH, CHANGE TO FIT YOUR LOCAL MACHINE
#     directory_path = "mapper outputs"

#     #Update the fw_graph colorbar, setting its title:
#     fw_graph.data[1].marker.colorbar.title = 'dist to<br>x-min'
#     html_output_path = directory_path + 'Eps_' + str(DBSCAN_EPSILON) +'_MinS_' + str(DBSCAN_MIN_SAMPLES) + '_NCubes_' +
#     mapper.visualize(scomplex, color_values=color_values, color_function_name=color_function_name,
#                      path_html=html_output_path, lens = summary_variable, lens_names = projected_vars)

#     return scomplex, X
```