In [67]:

```
import pandas as pd
import numpy as np
import gzip


class SOMToolBox_Parse:

    def __init__(self, filename):
        self.filename = filename

    def read_weight_file(self,):
        df = pd.DataFrame()
        if self.filename[-3:len(self.filename)] == '.gz':
            with gzip.open(self.filename, 'rb') as file:
                df, vec_dim, xdim, ydim = self._read_vector_file_to_df(df, file)
        else:
            with open(self.filename, 'rb') as file:
                df, vec_dim, xdim, ydim = self._read_vector_file_to_df(df, file)

        file.close()
        return df.astype('float64'), vec_dim, xdim, ydim


    def _read_vector_file_to_df(self, df, file):
        xdim, ydim, vec_dim, position = 0, 0, 0, 0
        for byte in file:
            line = byte.decode('UTF-8')
            if line.startswith('$'):
                xdim, ydim, vec_dim = self._parse_vector_file_metadata(line, xdim, y
                if xdim > 0 and ydim > 0 and len(df.columns) == 0:
                    df = pd.DataFrame(index=range(0, ydim * xdim), columns=range(0,
            else:
                if len(df.columns) == 0 or vec_dim == 0:
                    raise ValueError('Weight file has no correct Dimensional informa
                position = self._parse_weight_file_data(line, position, vec_dim, df)
        return df, vec_dim, xdim, ydim


    def _parse_weight_file_data(self, line, position, vec_dim, df):
        splitted=line.split(' ')
        try:
            df.values[position] = list(np.array(splitted[0:vec_dim]).astype(float))
            position += 1
        except: raise ValueError('The input-vector file does not match its unit-dime
        return  position


    def _parse_vector_file_metadata(self, line, xdim, ydim, vec_dim):
        splitted = line.split(' ')
        if splitted[0] == '$XDIM':      xdim = int(splitted[1])
        elif splitted[0] == '$YDIM':    ydim = int(splitted[1])
        elif splitted[0] == '$VEC_DIM': vec_dim = int(splitted[1])
        return xdim, ydim, vec_dim
```

Report - Clustering of SOM

Group G2

Aron Ingruber (01634059), Dominik Veselý (01633647)

Link to public repository: https://github.com/aroningruber/SOS-Exercise3-Group-G2

Implementation

We extended the class `SomViz` in the following cell with additional functions that add clustering of the SOM visualizations. For these visualizations, the weight vectors of those SOM units that are the best-matching unit for at least one input vector (i.e. interpolating units are omitted) are clustered. The SOM is then visualized with the units colored according to the cluster they belong to. Interpolating units are not colored and are thus left black.

As per requirement, four clustering methods are provided, namely k-means, as well as agglomerative clustering with the three linkage criterions "single", "complete" and "WARD". All of our implemented methods take the same arguments:

- `n_clusters` : The number of clusters (integer).
- `idata` : The input vectors (the data the SOM should be mapped to)
- `title` : The title for the produced visualization.

The k-means implementation additionally accepts a `random_state` argument to ensure determinism.

K-Means

The K-Means clustering algorithm minimizes the within-cluster sum-of-squares. Each data point is assigned to the cluster with the nearest mean (also referred to as cluster centroid), thus partitioning the data space into Voronoi cells. k denotes the number of clusters and determines the number of centroids that are created (usually by randomly selecting k data points) in the first stage of the algorithm. These centroids are iteratively adapted by clustering the data and shifting the centroids to the centers of gravity within each cluster. This iterative process stops when the means do not change anymore.

Disadvantages:

- Depends on initially selected centroids.
- Noise has a lot of influence on clustering results.

We use `scikit-learn` 's implementation of the k-means algorithm.

The method signature is `kmeans(self, n_clusters: int, idata=[], title="", random_state=42)` .

Hierarchical Clustering (Agglomerative)

In hierarchical agglomerative clustering, all data points are first assigned to their own cluster. These clusters are then iteratively merged (which can be used to produce a dendogram as a by-product). There are different criteria that can be used for deciding, which clusters to merge. As already mentioned, we implement three of them:

- Single Linkage: minimizes the distance between the closest vectors of pairs of clusters.

- Complete Linkage (or Maximum Linkage): minimizes the maximum distance between the closest vectors of pairs of clusters.
- WARD: minimizes the sum-of-squares within each cluster (thus making it a hierarchical variant of the k-means algorithm)

We use `scipy` 's implementation of the hierarchical clustering. This requires the computation of the pairwise distances between vectors, for which we also use `scipy` 's implementation.

The method signatures are:

- `single_linkage(self, n_clusters: int, idata=[], title="")`
- `complete_linkage(self, n_clusters: int, idata=[], title="")`
- `ward_linkage(self, n_clusters: int, idata=[], title="")`

Helper Methods

All four clustering methods above call our `plot_clusters(self, n_clusters: int, labels, mask, title="")` method. This method is essentially an adaptation of the existing `plot()` method. It also uses `plotly` 's Heatmap, but adds a grid to it, by creating gaps between units. It also colors the background (and thus interpolating, empty units, which are not part of the clustering) in black. The color scale to use depends on the number of clusters. Since the heatmap uses continuous color scales, the individual clusters can be difficult to distinguish visually when too many clusters are created. However, one can still hover over each unit to see the label of the respective cluster it belongs to.

We have also created the helper functions `get_mask(self, idata=[])` and `get_hist(self, idata=[])` . The latter is code extracted from the existing Hit Histogram visualization (`hithist()`). The former uses `get_hist()` to create a boolean mask, which separates interpolating units and those that are BMU for at least one input vector. This mask is used in the clustering methods to only cluster non-interpolating units.

In [68]:

```
import numpy as np
from scipy.cluster import hierarchy
from scipy.spatial import distance_matrix, distance
from ipywidgets import Layout, HBox, Box, widgets, interact
import plotly.express as px
import plotly.graph_objects as go
from sklearn import cluster
import matplotlib.pyplot as plt


from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))

class SomViz:

    def __init__(self, weights=[], m=None, n=None):
        self.weights = weights
        self.m = m
        self.n = n

    def get_hist(self, idata=[]):
        hist = [0] *self.n *self.m
        for v in idata:
            position =np.argmin(np.sqrt(np.sum(np.power(self.weights - v, 2), axis=1
```

```
            hist[position] += 1
        return np.array(hist)

    def get_mask(self, idata=[]):
        hist = self.get_hist(idata)
        mask = hist > 0
        return mask

    def umatrix(self, som_map=None, color="Viridis", interp = "best", title=""):
        um =np.zeros((self.m *self.n, 1))
        neuron_locs = list()
        for i in range(self.m):
            for j in range(self.n):
                neuron_locs.append(np.array([i, j]))
        neuron_distmat = distance_matrix(neuron_locs,neuron_locs)

        for i in range(self.m * self.n):
            neighbor_idxs = neuron_distmat[i] <= 1
            neighbor_weights = self.weights[neighbor_idxs]
            um[i] = distance_matrix(np.expand_dims(self.weights[i], 0), neighbor_wei

        if som_map==None: return self.plot(um.reshape(self.m,self.n), color=color, i
        else: som_map.data[0].z = um.reshape(self.m,self.n)

    def hithist(self, som_map=None, idata = [], color='RdBu', interp = "best", title
        hist = self.get_hist(idata)

        if som_map==None: return self.plot(hist.reshape(self.m,self.n), color=color,
        else:  som_map.data[0].z = np.array(hist).reshape(self.m,self.n)

    def component_plane(self, som_map=None, component=0, color="Viridis", interp = "
        if som_map==None: return self.plot(self.weights[:,component].reshape(-1,self
        else:  som_map.data[0].z = self.weights[:,component].reshape(-1,n)

    def sdh(self, som_map=None, idata=[], sdh_type=1, factor=1, draw=True, color="Ci

        import heapq
        sdh_m = [0] *self.m *self.n

        cs=0
        for i in range(0,factor): cs += factor-i

        for vector in idata:
            dist = np.sqrt(np.sum(np.power(self.weights - vector, 2), axis=1))
            c = heapq.nsmallest(factor, range(len(dist)), key=dist.__getitem__)
            if (sdh_type==1):
                for j in range(0,factor):  sdh_m[c[j]] += (factor-j)/cs # normalized
            if (sdh_type==2):
                for j in range(0,factor): sdh_m[c[j]] += 1.0/dist[c[j]] # based on d
            if (sdh_type==3):
                dmin = min(dist)
                for j in range(0,factor): sdh_m[c[j]] += 1.0 - (dist[c[j]]-dmin)/(ma

        if som_map==None: return self.plot(np.array(sdh_m).reshape(-1,self.n), color
        else: som_map.data[0].z = np.array(sdh_m).reshape(-1,self.n)

    def project_data(self,som_m=None, idata=[], title=""):

        data_y = []
        data_x = []
        for v in idata:
            position =np.argmin(np.sqrt(np.sum(np.power(self.weights - v, 2), axis=1
            x,y = position % self.n, position // self.n
            data_x.extend([x])
            data_y.extend([y])
```

```
        if som_m!=None: som_m.add_trace(go.Scatter(x=data_x, y=data_y, mode = "marke

    def time_series(self, som_m=None, idata=[], wsize=50, title=""): #not tested

        data_y = []
        data_x = [i for i in range(0,len(idata))]

        data_x2 = []
        data_y2 = []

        qmin = np.Inf
        qmax = 0

        step=1

        ps = []
        for v in idata:
            matrix = np.sqrt(np.sum(np.power(self.weights - v, 2), axis=1))
            position = np.argmin(matrix)
            qerror = matrix[position]
            if qmin>qerror: qmin = qerror
            if qmax<qerror: qmax = qerror
            ps.append((position, qerror))

        markerc=[]
        for v in ps:
            data_y.extend([v[0]])
            rez = v[1]/qmax

            markerc.append('rgba(0, 0, 0, '+str(rez)+')')

            x,y = v[0] % self.n, v[0] // self.n
            if    x==0: y = np.random.uniform(low=y, high=y+.1)
            elif  x==self.m-1: y = np.random.uniform(low=y-.1, high=y)
            elif  y==0: x = np.random.uniform(low=x, high=x+.1)
            elif  y==self.n-1: x = np.random.uniform(low=x-.1, high=x)
            else: x,y = np.random.uniform(low=x-.1, high=x+.1), np.random.uniform(lo

            data_x2.extend([x])
            data_y2.extend([y])

        ts_plot = go.FigureWidget(go.Scatter(x=[], y=[], mode = "markers", marker_co
        ts_plot.update_xaxes(range=[0, wsize])


        ts_plot.data[0].x, ts_plot.data[0].y = data_x, data_y
        som_m.add_trace(go.Scatter(x=data_x2, y=data_y2, mode = "markers",))

        som_m.layout.height = 500
        ts_plot.layout.height = 500
        som_m.layout.width = 500
        ts_plot.layout.width = 1300

        return HBox([go.FigureWidget(som_m), go.FigureWidget(ts_plot)])

    def plot(self, matrix, color="Viridis", interp = "best", title=""):
        return go.FigureWidget(go.Heatmap(z=matrix, zsmooth=interp, showscale=False,

    def plot_clusters(self, n_clusters: int, labels, use_mask, mask, title=""):
        matrix = labels.reshape(self.m,self.n)
        if use_mask:
            mask = mask.reshape(self.m, self.n)
            matrix[~mask] = None
        xgap = 2
```

```
        ygap = 2
        if n_clusters == 2:
            color = "Bluered"
        else:
            color = "Rainbow"
        return go.FigureWidget(
            go.Heatmap(
                z=matrix,
                showscale=False,
                colorscale=color,
                xgap=xgap,
                ygap=ygap
            ),
            layout=go.Layout(
                width=700,
                height=700,
                title=title,
                title_x=0.5,
                plot_bgcolor="black",
                xaxis=dict(showgrid=False, zeroline=False),
                yaxis=dict(showgrid=False, zeroline=False)
            )
        )

    def kmeans(self, n_clusters: int, idata=[], title="", use_mask=True, random_stat
        if use_mask:
            mask = self.get_mask(idata)
            weights = self.weights[mask]
        else:
            mask = None
            weights = self.weights

        # Train k-means classifier
        k_means = cluster.KMeans(n_clusters=n_clusters, n_init=1, max_iter=1000, ran

        if use_mask:
            labels = np.zeros(mask.shape)
            labels[mask] = k_means.labels_
        else:
            labels = k_means.labels_

        return self.plot_clusters(n_clusters, labels, use_mask=use_mask, mask=mask,

    def single_linkage(self, n_clusters: int, idata=[], title="", use_mask=True):
        if use_mask:
            mask = self.get_mask(idata)
            weights = self.weights[mask]
        else:
            mask = None
            weights = self.weights

        # Calculate distance matrix
        y = distance.pdist(weights)
        # Calculate single linkage
        Z = hierarchy.single(y)

        # Get labels of weight vector
        # (minimum label is 1, therefore 1 must be subtracted)
        if use_mask:
            labels = np.zeros(mask.shape)
            labels[mask] = hierarchy.fcluster(Z, n_clusters, criterion="maxclust") -
        else:
            labels = hierarchy.fcluster(Z, n_clusters, criterion="maxclust") - 1

        return self.plot_clusters(n_clusters, labels, use_mask=use_mask, mask=mask,
```

```
    def complete_linkage(self, n_clusters: int, idata=[], title="", use_mask=True):
        if use_mask:
            mask = self.get_mask(idata)
            weights = self.weights[mask]
        else:
            mask = None
            weights = self.weights

        # Calculate distance matrix
        y = distance.pdist(weights)
        # Calculate single linkage
        Z = hierarchy.complete(y)

        # Get labels of weight vector
        # (minimum label is 1, therefore 1 must be subtracted)
        if use_mask:
            labels = np.zeros(mask.shape)
            labels[mask] = hierarchy.fcluster(Z, n_clusters, criterion="maxclust") -
        else:
            labels = hierarchy.fcluster(Z, n_clusters, criterion="maxclust") - 1

        return self.plot_clusters(n_clusters, labels, use_mask=use_mask, mask=mask,

    def ward(self, n_clusters: int, idata=[], title="", use_mask=True):
        if use_mask:
            mask = self.get_mask(idata)
            weights = self.weights[mask]
        else:
            mask = None
            weights = self.weights

        # Calculate distance matrix
        y = distance.pdist(weights)
        # Calculate single linkage
        Z = hierarchy.ward(y)

        # Get labels of weight vector
        # (minimum label is 1, therefore 1 must be subtracted)
        if use_mask:
            labels = np.zeros(mask.shape)
            labels[mask] = hierarchy.fcluster(Z, n_clusters, criterion="maxclust") -
        else:
            labels = hierarchy.fcluster(Z, n_clusters, criterion="maxclust") - 1

        return self.plot_clusters(n_clusters, labels, use_mask=use_mask, mask=mask,
```

The following function can be used to load and pre-process (i.e. scale with `sklearn` 's min-max-scaling) three different datasets, namely the Iris, the Chainlink and the 10 Clusters dataset. The functions also print the `shape` of the data, thus implicitly informing us about the nature of the SOM (emergent or not, average number of datapoints per unit).

In [69]:

```
from sklearn import datasets, preprocessing

# Get pre-precessed datasets.

def get_iris():
    iris = datasets.load_iris().data
    print("Shape of the Iris data: {}".format(iris.shape))
    return preprocessing.MinMaxScaler().fit_transform(iris)

def get_chainlink():
```

```
    chainlink = pd.read_table("chainlink.vec", sep = " ", header = None, index_col =
    print("Shape of the Chainlink data: {}".format(chainlink.shape))
    return preprocessing.MinMaxScaler().fit_transform(chainlink)

def get_ten_clusters():
    ten_clusters = pd.read_table("10clusters.vec", sep = " ", header = None, index_c
    print("Shape of the 10 Clusters data: {}".format(ten_clusters.shape))
    return preprocessing.MinMaxScaler().fit_transform(ten_clusters)
```

The following cell contains the Iris dataset example.

In [70]:

```
import pandas as pd
import minisom as som
#interp: False, 'best', 'fast',
#color = 'viridis': https://plotly.com/python/builtin-colorscales/



##############################
######## miniSOM ############1/0
##############################
m=10
n=10

# Pre-processing
iris = get_iris()

# Train
s = som.MiniSom(m, n, iris.shape[1], sigma=0.8, learning_rate=0.7)
s.train_random(iris, 10000, verbose=False)

# Visualizaton
viz_miniSOM = SomViz(s._weights.reshape(-1,4), m, n)
um1 = viz_miniSOM.umatrix(color='magma', interp='best', title='U-matrix miniSOM')


##########################################
######## read from SOMToolBox ############
##########################################
trainedmap = SOMToolBox_Parse('iris.norm.vec')
idata, idim, idata_x, idata_y = trainedmap.read_weight_file()

smap = SOMToolBox_Parse('iris.wgt.gz')
smap, sdim, smap_x, smap_y = smap.read_weight_file()

# Visualizaton
viz_SOMToolBox = SomViz(smap.values.reshape(-1,sdim), smap_y, smap_x)
# um2 = viz_SOMToolBox.umatrix(color='viridis', interp=False, title='U-matrix SOMToo
um2 = viz_SOMToolBox.umatrix(color='magma', interp='best', title='U-matrix SOMToolBo

display(HBox([um1, um2]))
```

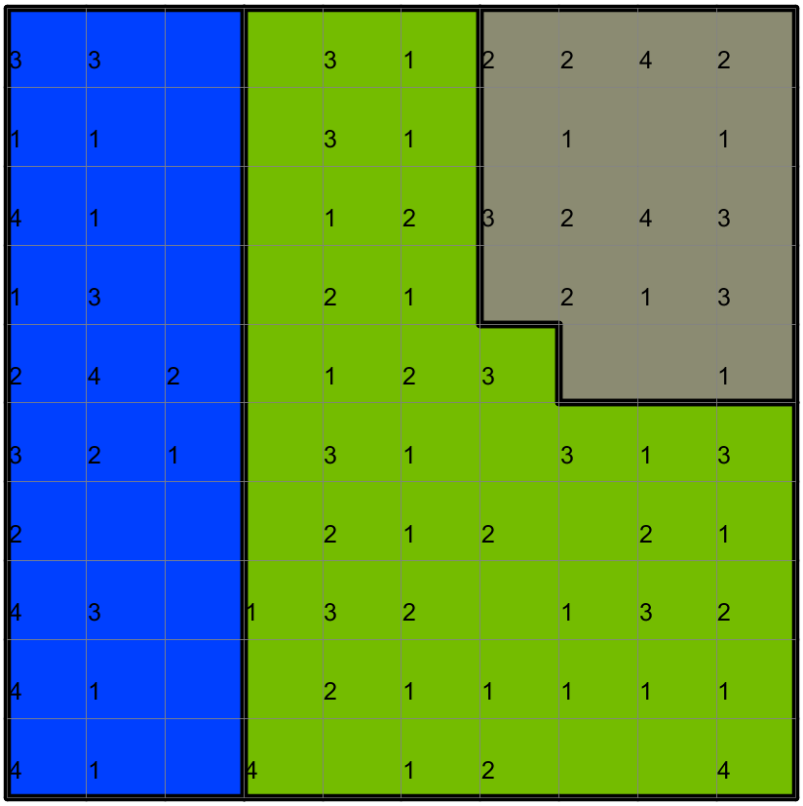
```
Shape of the Iris data: (150, 4)
```

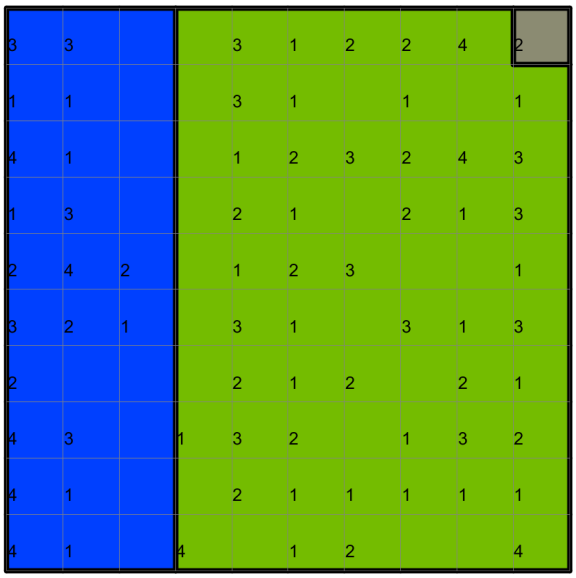
Comparison with Java SOMToolbox

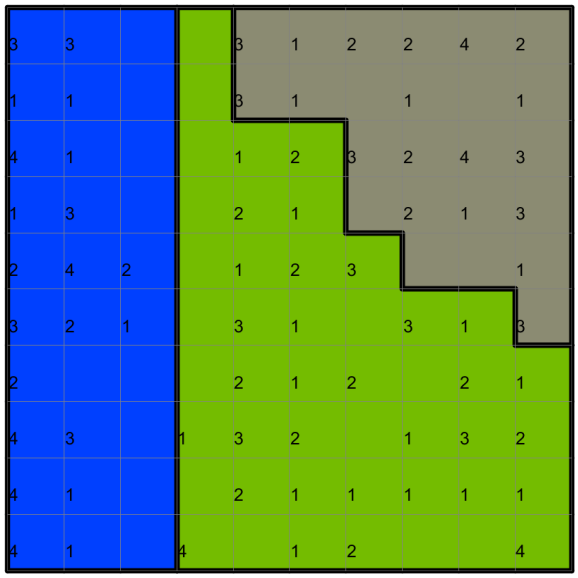
For the comparison with the Java SOMToolbox, we have decided to work with the Iris dataset and a 10x10 SOM. The SOMToolbox seems to have troubles with visualizations for the large SOMs that we have created below. Furthermore, in our opinion, the comparison on a small SOM,

using a well-understood dataset, should be sufficient to validate the correct behaviour of our visualizations.

The following cell showcases our four clustering visualizations on a small 10x10 SOM of the Iris dataset, of course with 3 clusters. Below, we have displayed the four corresponding visualizations produced by the SOM Toolbox, with the exception of the k-means clustering, which unfortunately did not work, as the program crashes as a consequence. With the exception of the colors, of course, we observe that the visualizations are equal, albeit mirrored along the horizontal axis. Therefore, we conclude that our clustering visualizations are correct.







In [71]:

```
m=10
n=10

trainedmap = SOMToolBox_Parse('iris.norm.vec')
idata, idim, idata_x, idata_y = trainedmap.read_weight_file()
```

```
smap = SOMToolBox_Parse('iris.wgt.gz')
smap, sdim, smap_x, smap_y = smap.read_weight_file()

# Visualization
viz = SomViz(smap.values.reshape(-1,sdim), smap_y, smap_x)
n_clusters = 3

viz_kmeans = viz.kmeans(n_clusters, idata=idata, title=f"k-means - Iris ({m}x{n})",
viz_ward = viz.ward(n_clusters, idata=idata, title=f"WARD - Iris ({m}x{n})", use_mas

viz_single = viz.single_linkage(n_clusters, idata=idata, title=f"Single linkage - Ir
viz_complete = viz.complete_linkage(n_clusters, idata=idata, title=f"Complete linkag

display(HBox([viz_kmeans, viz_ward]))
display(HBox([viz_single, viz_complete]))
```

Specifically for the task of producing clustering visualizations, we wrote the `SOMWrapper` class. It can be initialized with the SOM-specific parameters (size m x n, sigma, learning rate) as well as the number of clusters to produce in the visualizations. The class also contains a `train()` function which trains the SOM (for 10000 iterations by default).

In [72]:
```
from minisom import MiniSom

class SOMWrapper:
    def __init__(self, name="", data=[], m=10, n=10, n_clusters=3, sigma=1.0, learni
        self.name = name
        self.data = data
        self.m = m
        self.n = n
        self.n_clusters = n_clusters
        self.sigma = sigma
        self.learning_rate = learning_rate

        self.som = MiniSom(m, n, data.shape[1], sigma=sigma, learning_rate=learning_

    def train(self, num_iterations=10000, verbose=False):
        self.som.train_random(self.data, num_iterations, verbose=verbose)
        self.viz = SomViz(self.som._weights.reshape(-1, self.data.shape[-1]), self.m
```

In the following cell we create two SOM's for the Chainlink dataset. A small SOM (40x20, 800 units) and a large SOM (100x60, 6000 units). We use a sigma of 2.0 and a learning rate of 0.07. Each SOM is trained for 10000 iterations.

In [73]:
```
#
# Train SOMs on chainlink dataset
#

soms_chainlink = [
    # Small SOM
    SOMWrapper(
        name="Chainlink",
        data=get_chainlink(),
        m=40,
        n=20,
        n_clusters=2,
        sigma=2,
        learning_rate=0.07
    ),
```

```
    # Large SOM
    SOMWrapper(
        name="Chainlink",
        data=get_chainlink(),
        m=100,
        n=60,
        n_clusters=2,
        sigma=2,
        learning_rate=0.07,
    )
]

for som in soms_chainlink:
    som.train(num_iterations=10_000, verbose=False)
```

```
Shape of the Chainlink data: (1000, 3)
Shape of the Chainlink data: (1000, 3)
```

We now create the visualizations for both the small and the large SOM on the Chainlink dataset. Aside from the four different clustering visualizations that we implemented, we also show a Hit-histogram and a U-matrix visualization.

In [74]:

```
#
# Visualizations on chainlink dataset
#
from ipywidgets import HBox, VBox
import random

for som in soms_chainlink:

    viz_hit = som.viz.hithist(idata=som.data, interp=False, title=f"Hit-histogram -
    viz_um = som.viz.umatrix(color='viridis', interp=False, title=f"U-matrix - {som.

    viz_kmeans = som.viz.kmeans(som.n_clusters, idata=som.data, title=f"k-means - {s
    viz_ward = som.viz.ward(som.n_clusters, idata=som.data, title=f"WARD - {som.name

    viz_single = som.viz.single_linkage(som.n_clusters, idata=som.data, title=f"Sing
    viz_complete = som.viz.complete_linkage(som.n_clusters, idata=som.data, title=f"

    display(HBox([viz_hit, viz_um]))
    display(HBox([viz_kmeans, viz_ward]))
    display(HBox([viz_single, viz_complete]))
```

To ensure that the chosen parameters for our SOMs for the chainlink dataset are appropriate, we train additional SOMs with some extreme values for sigma and the learning rate.

In [75]:

```
soms_chainlink_extreme = [
    SOMWrapper(
        name="Chainlink",
        data=get_chainlink(),
        m=40,
        n=20,
        n_clusters=2,
        sigma=0.01,
        learning_rate=0.01,
    ),
    SOMWrapper(
```

```
        name="Chainlink",
        data=get_chainlink(),
        m=40,
        n=20,
        n_clusters=2,
        sigma=10,
        learning_rate=10,
    ),
    SOMWrapper(
        name="Chainlink",
        data=get_chainlink(),
        m=40,
        n=20,
        n_clusters=2,
        sigma=1,
        learning_rate=0.5,
    ),
]

for som in soms_chainlink_extreme:
    som.train(num_iterations=10_000, verbose=False)
```

```
Shape of the Chainlink data: (1000, 3)
Shape of the Chainlink data: (1000, 3)
Shape of the Chainlink data: (1000, 3)
```

In [76]:

```
from ipywidgets import HBox, VBox

som_normal = soms_chainlink[0]
som_ext1 = soms_chainlink_extreme[0]
som_ext2 = soms_chainlink_extreme[1]
som_ext3 = soms_chainlink_extreme[2]

viz_normal = som_normal.viz.hithist(idata=som_normal.data, interp=False, title=f"{so
viz_ext1 = som_ext1.viz.hithist(idata=som_ext1.data, interp=False, title=f"{som_ext1
viz_ext2 = som_ext2.viz.hithist(idata=som_ext2.data, interp=False, title=f"{som_ext2
viz_ext3 = som_ext3.viz.hithist(idata=som_ext3.data, interp=False, title=f"{som_ext3

display(HBox([viz_normal, viz_ext1]))
display(HBox([viz_ext2, viz_ext3]))
```

In the following cell we create two SOM's for the 10-clusters dataset. A small SOM (40x20, 800 units) and a large SOM (100x60, 6000 units). We use a sigma of 2.0 and a learning rate of 0.07. Each SOM is trained for 10000 iterations.

In [77]:

```
#
# Train SOMs on 10-clusters dataset
#

soms_ten_clusters = [
    SOMWrapper(
        name="10-clusters",
        data=get_ten_clusters(),
        m=40,
        n=20,
        n_clusters=10,
        sigma=2,
        learning_rate=0.07,
    ),
    SOMWrapper(
        name="10-clusters",
```

```
        data=get_ten_clusters(),
        m=100,
        n=60,
        n_clusters=10,
        sigma=2,
        learning_rate=0.07,
    ),
]

for som in soms_ten_clusters:
    som.train(num_iterations=10_000, verbose=False)
```

```
Shape of the 10 Clusters data: (850, 10)
Shape of the 10 Clusters data: (850, 10)
```

We now create the visualizations for both the small and the large SOM on the 10 clusters dataset. Aside from the four different clustering visualizations that we implemented, we also show a Hit-histogram and a U-matrix visualization.

In [78]:

```
#
# Visualizations on 10-clusters dataset
#
from ipywidgets import HBox, VBox

for som in soms_ten_clusters:

    viz_hit = som.viz.hithist(idata=som.data, interp=False, title=f"Hit-histogram -
    viz_um  = som.viz.umatrix(color='viridis', interp=False, title=f"U-matrix - {som

    viz_kmeans = som.viz.kmeans(som.n_clusters, idata=som.data, title=f"k-means - {s
    viz_ward   = som.viz.ward(som.n_clusters, idata=som.data, title=f"WARD - {som.na

    viz_single = som.viz.single_linkage(som.n_clusters, idata=som.data, title=f"Sing
    viz_complete = som.viz.complete_linkage(som.n_clusters, idata=som.data, title=f"

    display(HBox([viz_hit, viz_um]))
    display(HBox([viz_kmeans, viz_ward]))
    display(HBox([viz_single, viz_complete]))
```

One interesting observation about the clustering of the 10-clusters dataset is that none of the clustering algorithms is able to properly separate the 10 clusters. Although humans may be able to spot the ten different clusters, the algorithms seem to have problems with it. We assume that this is caused be the non-convex mapping of the data to the SOM.

To ensure that the chosen parameters for our SOMs for the 10-clusters dataset are appropriate, we train additional SOMs with some extreme values for sigma and the learning rate.

In [79]:

```
soms_ten_clusters_extreme = [
    SOMWrapper(
        name="10-clusters",
        data=get_ten_clusters(),
        m=40,
        n=20,
        n_clusters=10,
        sigma=0.01,
        learning_rate=0.01,
```

```
    ),
    SOMWrapper(
        name="10-clusters",
        data=get_ten_clusters(),
        m=40,
        n=20,
        n_clusters=10,
        sigma=10,
        learning_rate=10,
    ),
    SOMWrapper(
        name="10-clusters",
        data=get_ten_clusters(),
        m=40,
        n=20,
        n_clusters=10,
        sigma=1,
        learning_rate=0.5,
    ),
]

for som in soms_ten_clusters_extreme:
    som.train(num_iterations=10_000, verbose=False)
```

```
Shape of the 10 Clusters data: (850, 10)
Shape of the 10 Clusters data: (850, 10)
Shape of the 10 Clusters data: (850, 10)
```

In [80]:

```
from ipywidgets import HBox, VBox

som_normal = soms_ten_clusters[0]
som_ext1 = soms_ten_clusters_extreme[0]
som_ext2 = soms_ten_clusters_extreme[1]
som_ext3 = soms_ten_clusters_extreme[2]

viz_normal = som_normal.viz.hithist(idata=som_normal.data, interp=False, title=f"{so
viz_ext1 = som_ext1.viz.hithist(idata=som_ext1.data, interp=False, title=f"{som_ext1
viz_ext2 = som_ext2.viz.hithist(idata=som_ext2.data, interp=False, title=f"{som_ext2
viz_ext3 = som_ext3.viz.hithist(idata=som_ext3.data, interp=False, title=f"{som_ext3

display(HBox([viz_normal, viz_ext1]))
display(HBox([viz_ext2, viz_ext3]))
```

In [ ]: