Visualization of the Topographic Product (Coding Topic A)

Ramírez Mejía Lorenzo (00473221), Jean-Baptiste Cellier (12402507)

GitHub Repository: https://github.com/Jean-BaptisteCellier/SOM_TopographicProduct.git

License: Apache License

Description: This repository contains Python code for SOM visualization of the topographic product and provenance generation. **Provenance**: We were not sure if we did this right, the provenance can be found in the provenance folder.

In [240…

```
import numpy as np
import pandas as pdcoding
import gzip
```

In [241…

```
#SOMToolbox Parser
from SOMToolBox_Parse import SOMToolBox_Parse
idata = SOMToolBox_Parse("iris\\iris.vec").read_weight_file()
weights = SOMToolBox_Parse("iris\\iris.wgt.gz").read_weight_file()
```

In [242…

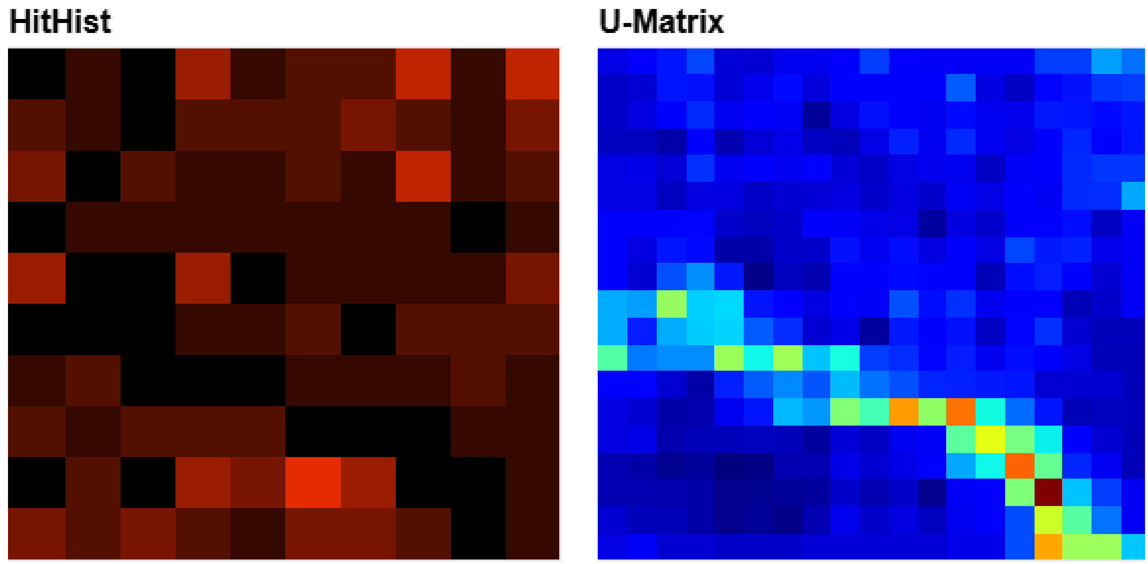
```
#HitHistogram
def HitHist(_m, _n, _weights, _idata):
    hist = np.zeros(_m * _n)
    for vector in _idata:
        position =np.argmin(np.sqrt(np.sum(np.power(_weights - vector, 2), axis=1)
        hist[position] += 1

    return hist.reshape(_m, _n)

#U-Matrix - implementation
def UMatrix(_m, _n, _weights, _dim):
    U = _weights.reshape(_m, _n, _dim)
    U = np.insert(U, np.arange(1, _n), values=0, axis=1)
    U = np.insert(U, np.arange(1, _m), values=0, axis=0)
    #calculate interpolation
    for i in range(U.shape[0]):
        if i%2==0:
            for j in range(1,U.shape[1],2):
                U[i,j][0] = np.linalg.norm(U[i,j-1] - U[i,j+1], axis=-1)
        else:
            for j in range(U.shape[1]):
                if j%2==0:
                    U[i,j][0] = np.linalg.norm(U[i-1,j] - U[i+1,j], axis=-1)
                else:
                    U[i,j][0] = (np.linalg.norm(U[i-1,j-1] - U[i+1,j+1], axis=-1)

    U = np.sum(U, axis=2) #move from Vector to Scalar

    for i in range(0, U.shape[0], 2): #count new values
        for j in range(0, U.shape[1], 2):
            region = []
            if j>0: region.append(U[i][j-1]) #check left border
            if i>0: region.append(U[i-1][j]) #check bottom
```

```
            if j<U.shape[1]-1: region.append(U[i][j+1]) #check right border
            if i<U.shape[0]-1: region.append(U[i+1][j]) #check upper border

            U[i,j] = np.median(region)

    return U

#SDH - implementation
def SDH(_m, _n, _weights, _idata, factor, approach):
    import heapq

    sdh_m = np.zeros( _m * _n)

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

    for vector in _idata:
        dist = np.sqrt(np.sum(np.power(_weights - vector, 2), axis=1))
        c = heapq.nsmallest(factor, range(len(dist)), key=dist.__getitem__)
        if (approach==0): # normalized
            for j in range(factor):  sdh_m[c[j]] += (factor-j)/cs
        if (approach==1):# based on distance
            for j in range(factor): sdh_m[c[j]] += 1.0/dist[c[j]]
        if (approach==2):
            dmin, dmax = min(dist[c]), max(dist[c])
            for j in range(factor): sdh_m[c[j]] += 1.0 - (dist[c[j]]-dmin)/(dmax-d

    return sdh_m.reshape(_m, _n)
```

In [243…

```
import panel as pn
import holoviews as hv
from holoviews import opts
hv.extension('bokeh')

hithist = hv.Image(HitHist(weights['ydim'], weights['ydim'], weights['arr'], idata
um = hv.Image(UMatrix(weights['ydim'], weights['ydim'], weights['arr'], 4)).opts(x
sdh = hv.Image(SDH(weights['ydim'], weights['ydim'], weights['arr'], idata['arr'],

hv.Layout([hithist.relabel('HitHist').opts(cmap='kr'),
           um.relabel('U-Matrix').opts(cmap='jet'), sdh.relabel('SDH').opts(cmap=
```

Out[243…



Data processing

- Compute pairwise distances between weights in "input space", by using Euclidean distance.
- Compute the pairwise SOM distances in the grid ("output space"), by using Manhattan distance.

In [244…

```
from sklearn.metrics.pairwise import euclidean_distances

def compute_pairwise_som_distances(idata, weights):
    idata_arr = idata['arr']
    weights_arr = weights['arr']
    som_xdim, som_ydim = weights['xdim'], weights['ydim']
    grid_positions = np.array([[i, j] for i in range(som_xdim) for j in range(som_
    som_distances = np.abs(grid_positions[:, np.newaxis] - grid_positions).sum(ax
    weights_distances = euclidean_distances(weights_arr)
    return som_distances, weights_distances

idata_arr = idata['arr']
weights_arr = weights['arr']
som_xdim, som_ydim = weights['xdim'], weights['ydim']

# # Pairwise distances in input space
# input_distances = np.linalg.norm(idata_arr[:, np.newaxis] - idata_arr, axis=2)

# Best-matching units for input vectors
# bmus = np.argmin(np.linalg.norm(idata_arr[:, np.newaxis] - weights_arr.reshape(-
# # Pairwise SOM distances (Manhattan distance)
# grid_positions = np.array([[i, j] for i in range(som_xdim) for j in range(som_yd
# bmu_positions = grid_positions[bmus]
# som_distances = np.abs(bmu_positions[:, np.newaxis] - bmu_positions).sum(axis=2)

grid_positions = np.array([[i, j] for i in range(som_xdim) for j in range(som_ydir
som_distances, weights_distances = compute_pairwise_som_distances(idata, weights)
```

In [245…

```
def get_kth_neighbor(unit_i, all_distances, k):
    distances = all_distances[unit_i]
```

```
    nonzero_distances = distances[distances > 0]
    unique_distances = np.unique(nonzero_distances)

    if len(unique_distances) < k:
        raise ValueError(f"Not enough neighbors: k={k} but only {len(unique_distar

    kth_unique_distance = unique_distances[k - 1]
    kth_neighbor_indices = np.where(distances == kth_unique_distance)

    mean_kth_distance = np.mean(distances[kth_neighbor_indices])

    return kth_neighbor_indices, mean_kth_distance
```

P_1: Distortion in input space

Comparison of distances between weights of neighboring units (input space)

Purpose: Measures how well the topological relationships in the input space are preserved in the SOM. If two input vectors are close in the original input space, their corresponding BMUs should also reflect this closeness in the output space.

$$P_1(j,k) = \left(\prod_{l=1}^{k} Q_1(j,l)\right)^{1/k}$$

where

$$Q_1(j,l) = \frac{d^I(\mathbf{w}_j, \mathbf{w}_{n_k^O(j)})}{d^I(\mathbf{w}_j, \mathbf{w}_{n_k^I(j)})}$$

j: reference unit

l: "order" of the neighborhod

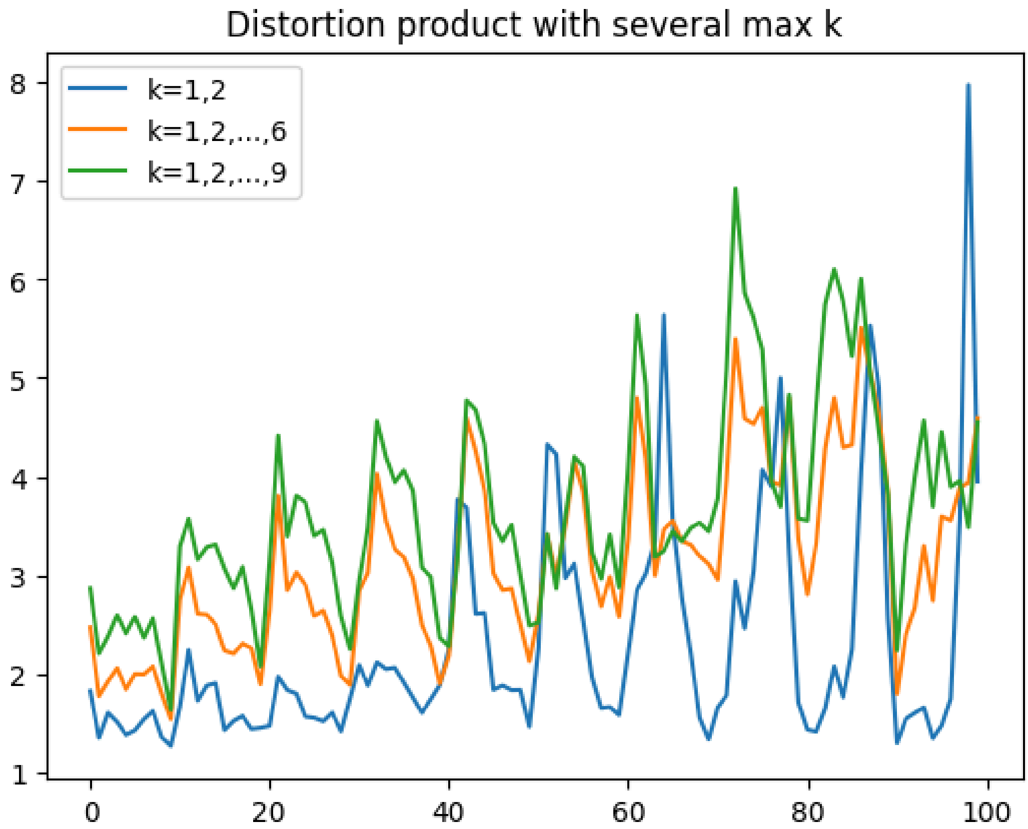
In [246…

```
def distortion_input_space(som_distances, weights_distances, unit_i, max_k):
    p1 = 1.0  # Initialize the distortion product

    for k in range(1, max_k + 1):
        # Distance based on neighbors in SOM grid
        som_k_neighbors, _ = get_kth_neighbor(unit_i, som_distances, k)
        input_dist_output_neighbors = np.mean([weights_distances[unit_i, neighbor]

        # Distance based on neighbors in input space
        _, input_dist_input_neighbors = get_kth_neighbor(unit_i, weights_distances
        if input_dist_input_neighbors == 0:
            q1 = 0
        else:
            q1 = input_dist_output_neighbors / input_dist_input_neighbors

        # Compute Q1 for this k-th neighborhood
        p1 *= q1

    # Final distortion
    p1 = p1 ** (1 / max_k)

    return p1
```

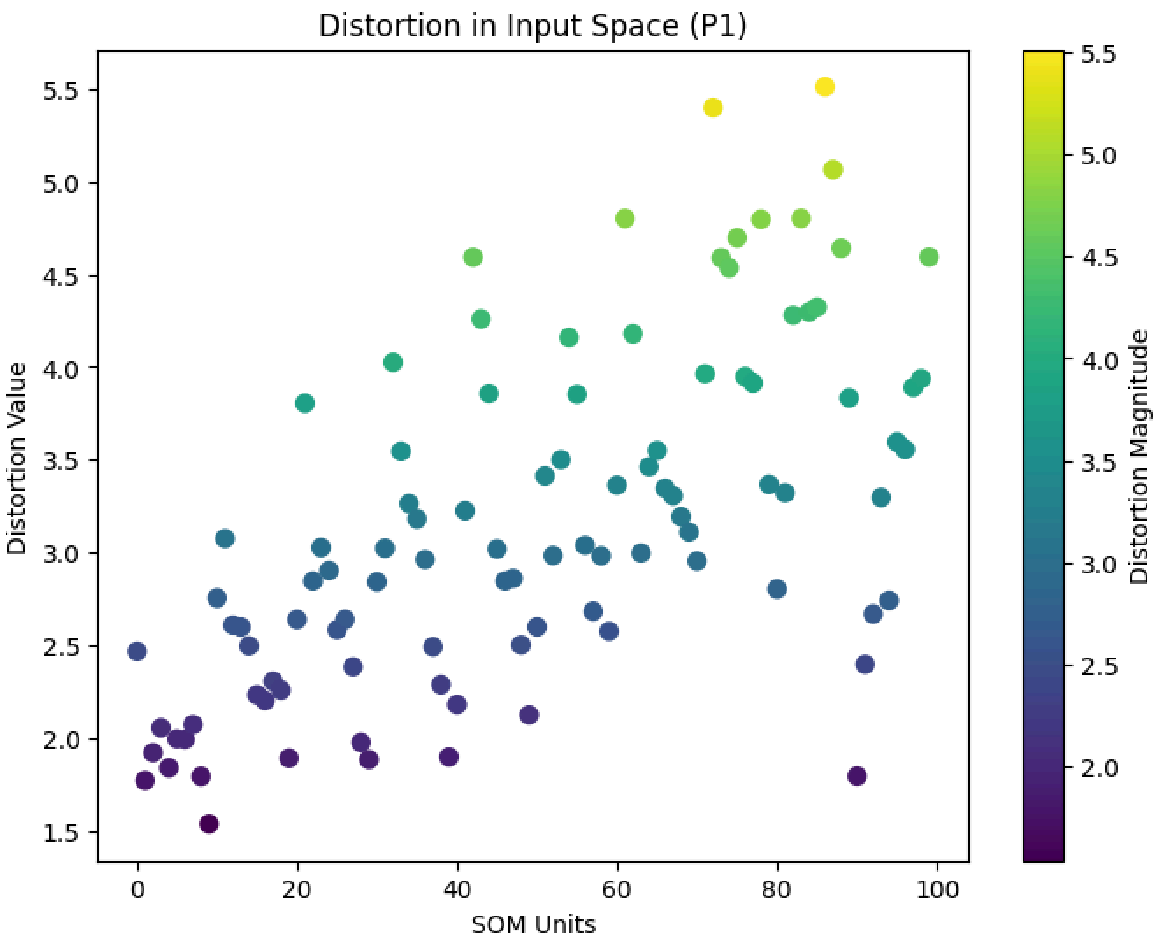
In [247…
```
list_disto_input_2 =[]
list_disto_input_6 = []
list_disto_input_9 = []

dim = som_distances.shape[0]

for i in range(dim):
    list_disto_input_2.append(distortion_input_space(som_distances, weights_distar
    list_disto_input_6.append(distortion_input_space(som_distances, weights_distar
    list_disto_input_9.append(distortion_input_space(som_distances, weights_distar
```

In [248…
```
import matplotlib.pyplot as plt
plt.plot(range(dim), list_disto_input_2, label="k=1,2")
plt.plot(range(dim), list_disto_input_6, label="k=1,2,...,6")
plt.plot(range(dim), list_disto_input_9, label="k=1,2,...,9")
plt.legend()
plt.title("Distortion product with several max k")
```

Out[248…
```
Text(0.5, 1.0, 'Distortion product with several max k')
```

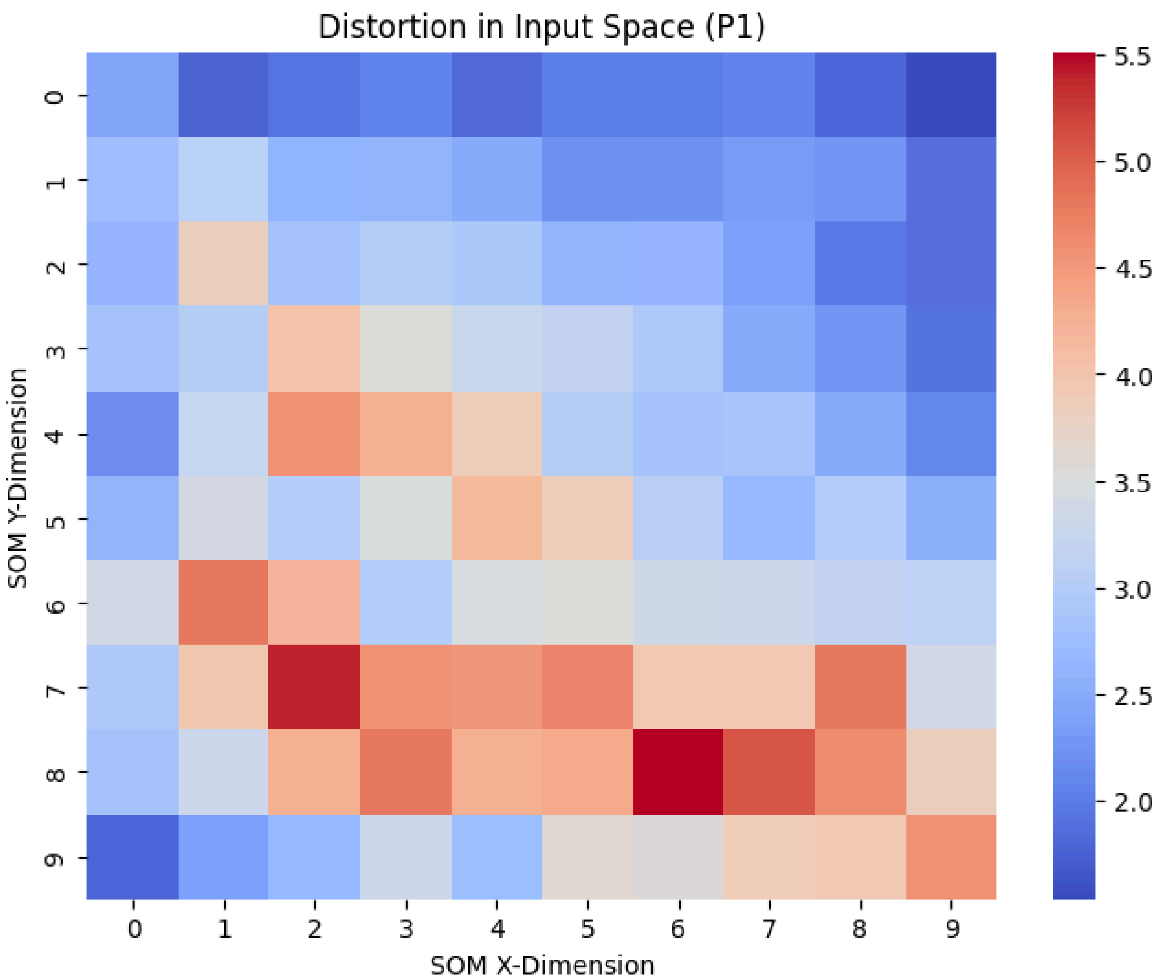


In [249…
```
plt.figure(figsize=(8, 6))
plt.scatter(range(len(list_disto_input_6)), list_disto_input_6, c=list_disto_input
plt.colorbar(label='Distortion Magnitude')
plt.title('Distortion in Input Space (P1)')
plt.xlabel('SOM Units')
plt.ylabel('Distortion Value')
plt.show()
```



In [250…

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

disto_input_6 = np.array(list_disto_input_6)
disto_input_6 = disto_input_6.reshape(som_xdim, som_ydim)

plt.figure(figsize=(8, 6))
sns.heatmap(disto_input_6, cmap='coolwarm', annot=False)
plt.title('Distortion in Input Space (P1)')
plt.xlabel('SOM X-Dimension')
plt.ylabel('SOM Y-Dimension')
plt.show()
```



P_2: Distortion in output space

Comparison of distances between SOM units in the grid (output space)

Purpose: Measures how well the data structure of the grid reflects the original input space distances. If two units are close in the SOM grid, the corresponding weights in input space should also be close.

$$P_2(j,k) = \left(\prod_{l=1}^{k} Q_2(j,l)\right)^{1/k}$$

where

$$Q_2(j,l) = \frac{d^O(j, n_k^O(j))}{d^O(j, n_k^I(j))}$$

j: reference unit
l: "order" of the neighborhod

In [251…

```
def distortion_output_space(som_distances, weights_distances, unit_i, max_k):
    p2 = 1.0

    for k in range(1, max_k + 1):
        # k-th nearest neighbor(s) and distance in SOM space for unit_i
        _, output_dist_output_neighbors = get_kth_neighbor(unit_i, som_distances,
```

```
        # k-th nearest neighbor(s) in input space
        input_k_neighbors, _ = get_kth_neighbor(unit_i, weights_distances, k)
        # Output distance to the k-th nearest in input space
        output_dist_input_neighbors = np.mean([som_distances[unit_i, neighbor] for

        if output_dist_input_neighbors == 0:
            q2 = 1
        else:
            q2 = output_dist_output_neighbors / output_dist_input_neighbors

        # Compute Q2 for this k-th neighborhood

        p2 *= q2

    # Final distortion
    p2 = p2 ** (1 / max_k)

    return p2
```

In [252…

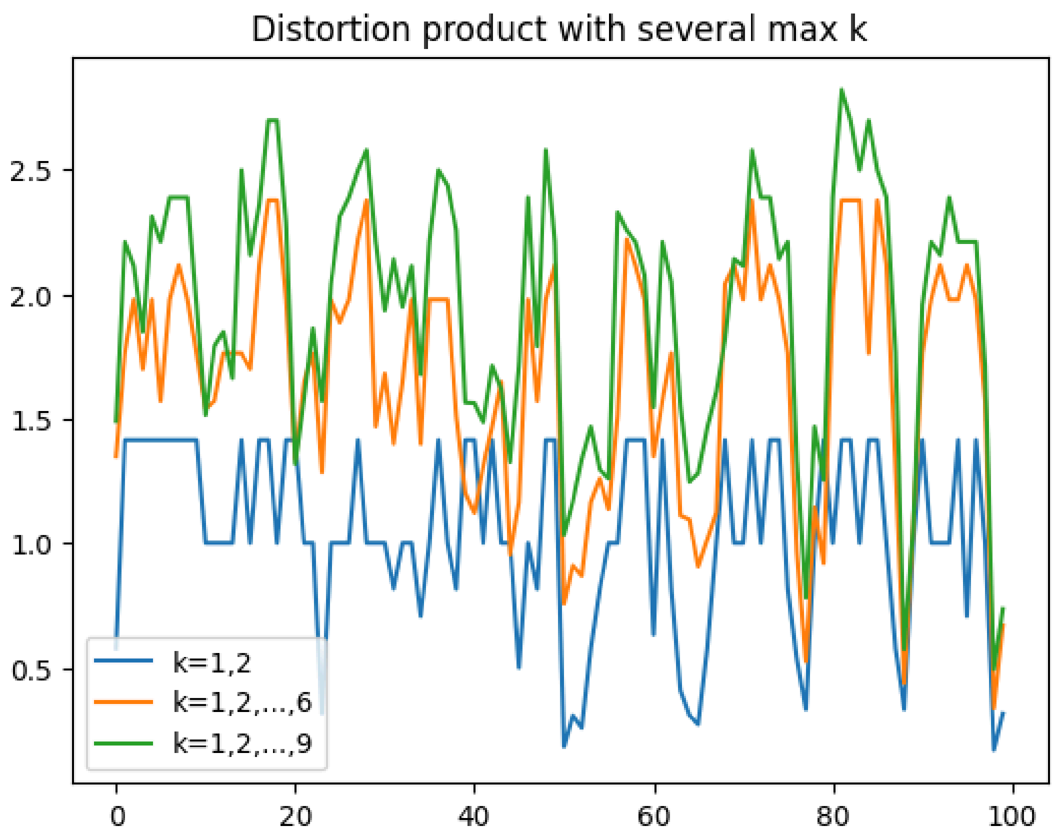
```
list_disto_output_2 = []
list_disto_output_6 = []
list_disto_output_9 = []

for i in range(dim):
    list_disto_output_2.append(distortion_output_space(som_distances, weights_dist
    list_disto_output_6.append(distortion_output_space(som_distances, weights_dist
    list_disto_output_9.append(distortion_output_space(som_distances, weights_dist
```

In [253…

```
import matplotlib.pyplot as plt
plt.plot(range(dim), list_disto_output_2, label="k=1,2")
plt.plot(range(dim), list_disto_output_6, label="k=1,2,...,6")
plt.plot(range(dim), list_disto_output_9, label="k=1,2,...,9")
plt.legend()
plt.title("Distortion product with several max k")
```

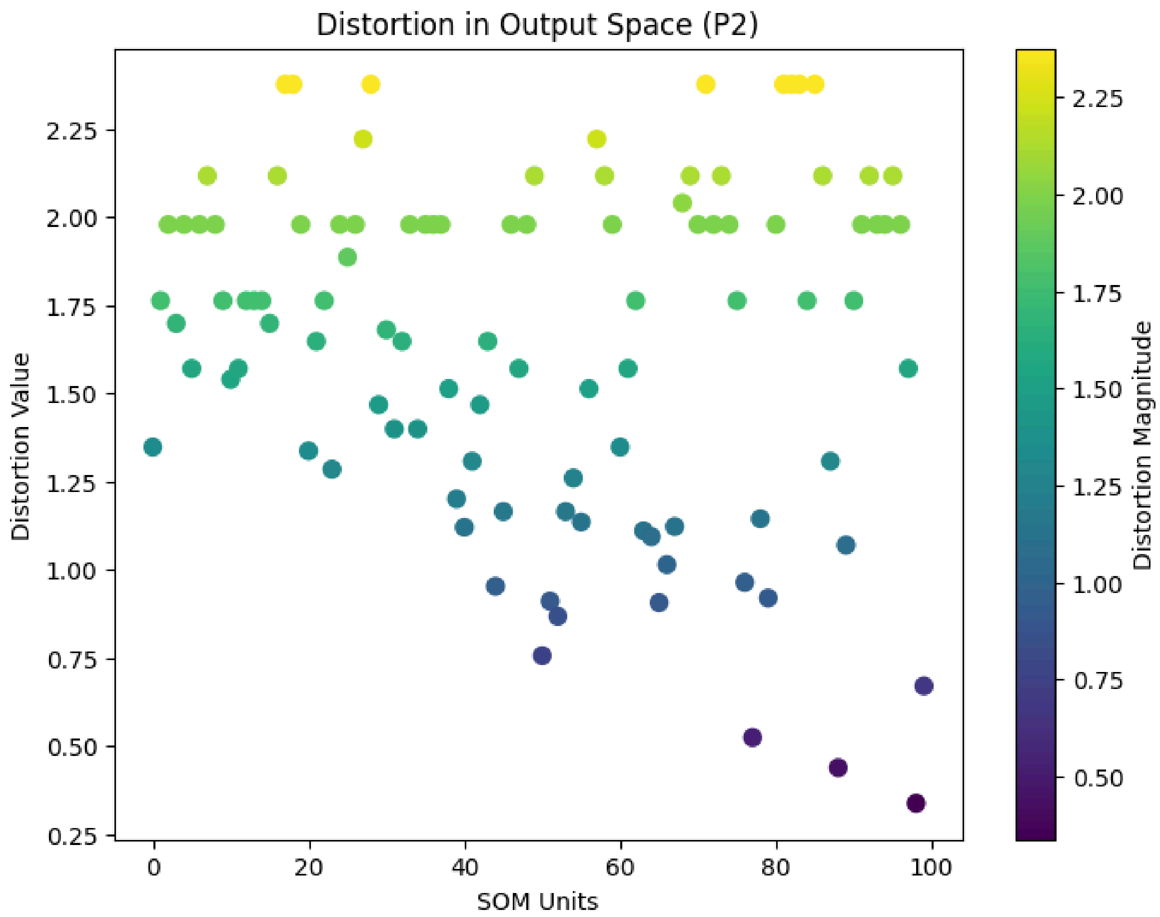
Out[253…

```
Text(0.5, 1.0, 'Distortion product with several max k')
```



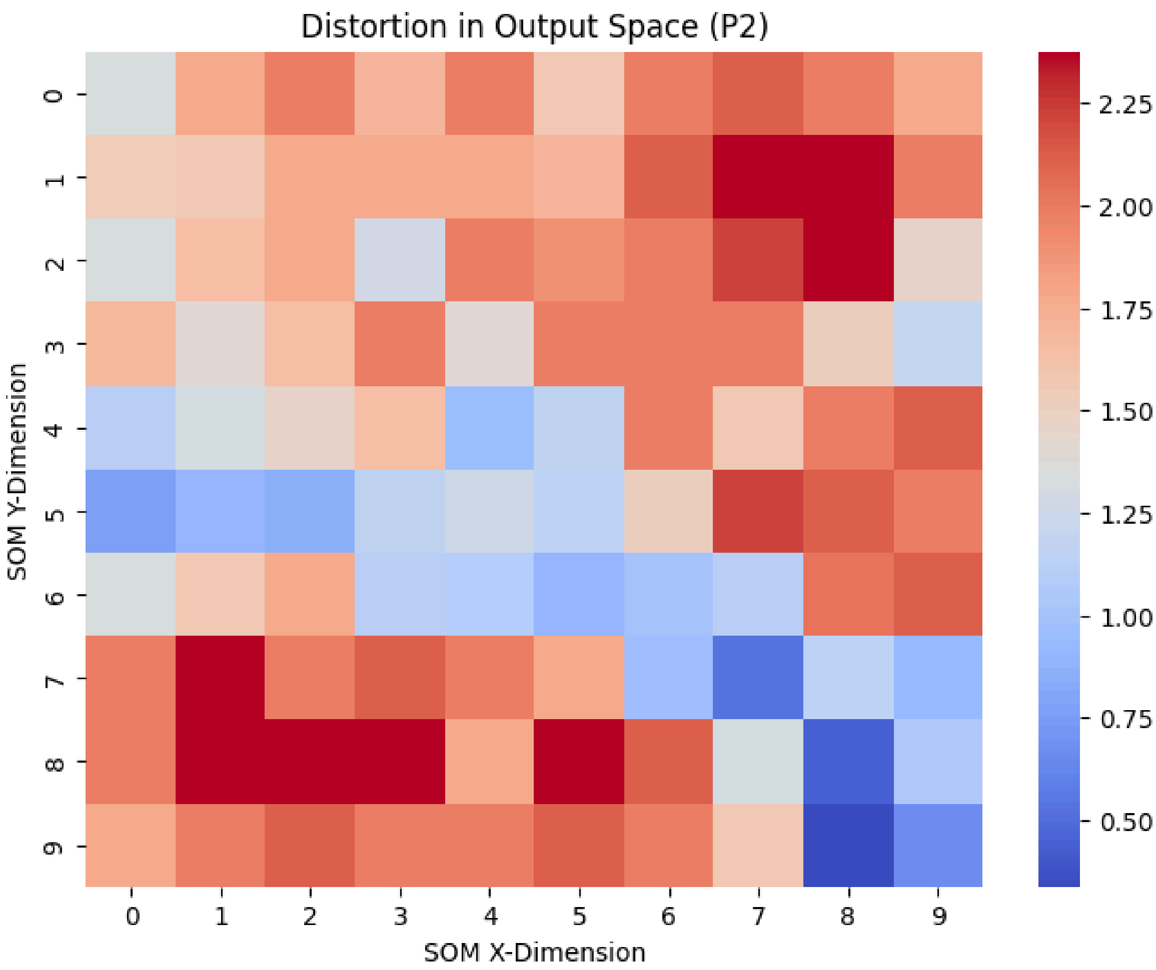
In [254…

```
plt.figure(figsize=(8, 6))
plt.scatter(range(len(list_disto_output_6)), list_disto_output_6, c=list_disto_out
plt.colorbar(label='Distortion Magnitude')
plt.title('Distortion in Output Space (P2)')
plt.xlabel('SOM Units')
plt.ylabel('Distortion Value')
plt.show()
```



In [255…

```
disto_output_6 = np.array(list_disto_output_6)
disto_output_6 = disto_output_6.reshape(som_xdim, som_ydim)

plt.figure(figsize=(8, 6))
sns.heatmap(disto_output_6, cmap='coolwarm', annot=False)
plt.title('Distortion in Output Space (P2)')
plt.xlabel('SOM X-Dimension')
plt.ylabel('SOM Y-Dimension')
plt.show()
```



P_3: Geometric mean of distortions

Geometric mean of P_1 and P_2, capturing an overall sense of topological preservation.

$$P_3(j,k) = \sqrt{P_1(j,k) \cdot P_2(j,k)} = \left(\prod_{l=1}^{k} Q_1(j,l)Q_2(j,l)\right)^{1/2k}$$

In [256…

```
def mean_distortion(som_distances, weights_distances, unit_i, max_k):
    p1 = distortion_input_space(som_distances, weights_distances, unit_i, max_k)
    p2 = distortion_output_space(som_distances, weights_distances, unit_i, max_k)
    return np.sqrt(p1*p2)
```

In [257…

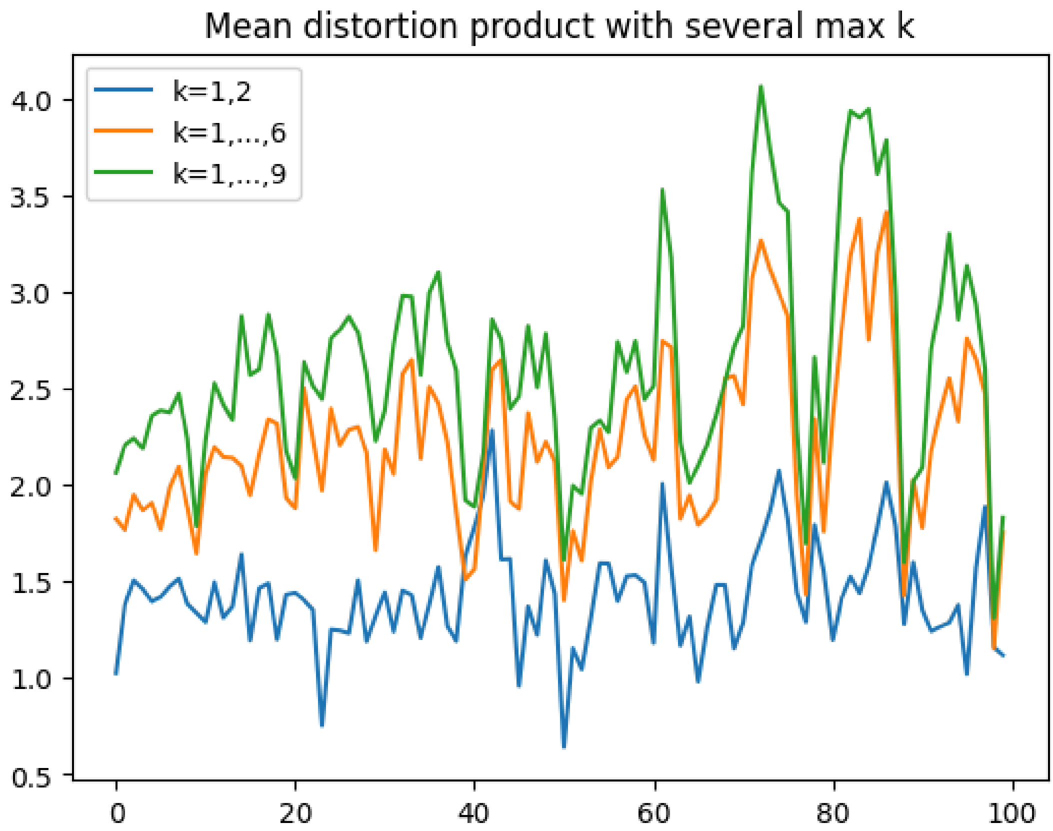
```
list_mean_disto_2 = []
list_mean_disto_6 = []
list_mean_disto_9 = []
for i in range(dim):
    list_mean_disto_2.append(mean_distortion(som_distances, weights_distances, i,
    list_mean_disto_6.append(mean_distortion(som_distances, weights_distances, i,
    list_mean_disto_9.append(mean_distortion(som_distances, weights_distances, i,
```

In [258…

```
plt.plot(range(dim), list_mean_disto_2, label="k=1,2")
plt.plot(range(dim), list_mean_disto_6, label="k=1,...,6")
plt.plot(range(dim), list_mean_disto_9, label="k=1,...,9")
plt.legend()
plt.title("Mean distortion product with several max k")
```

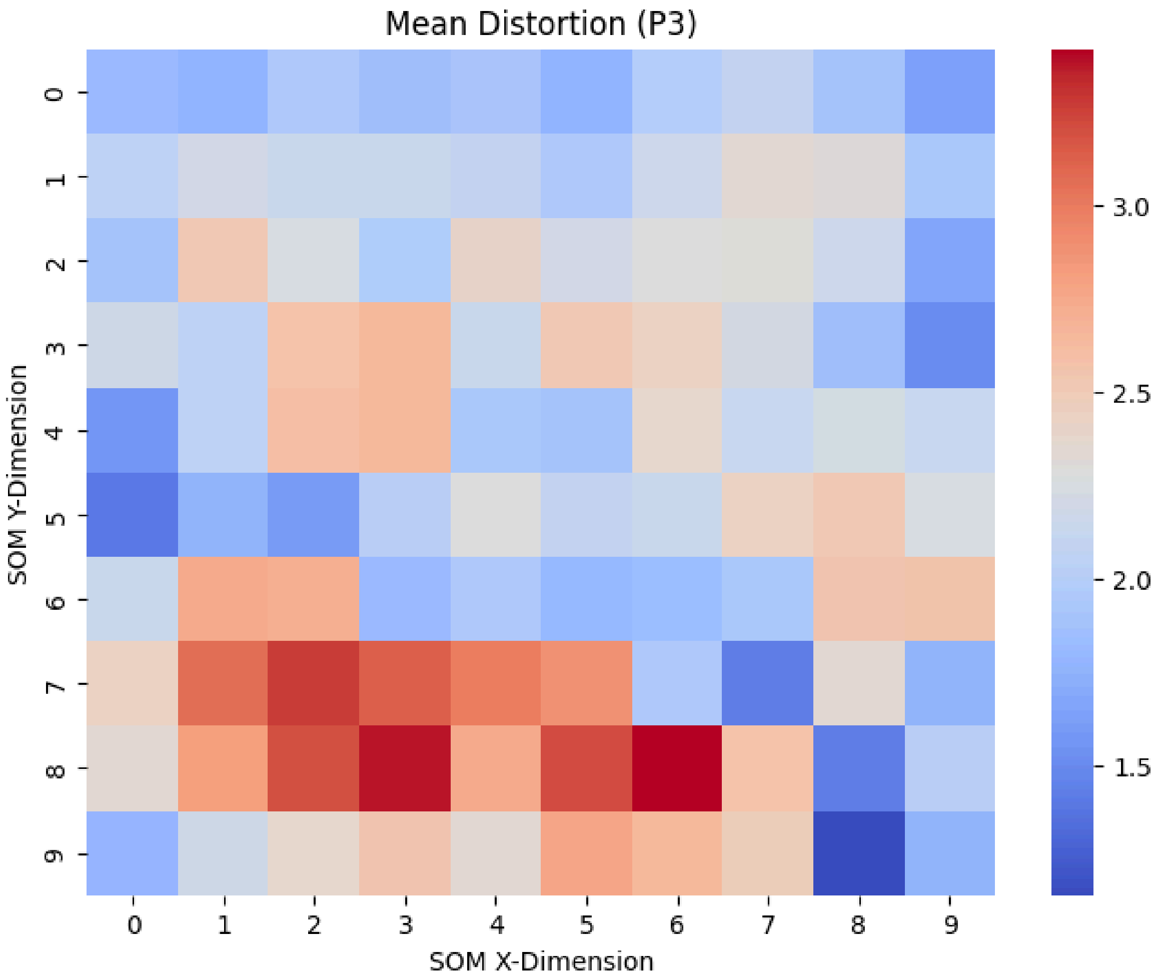
Out[258…

```
Text(0.5, 1.0, 'Mean distortion product with several max k')
```



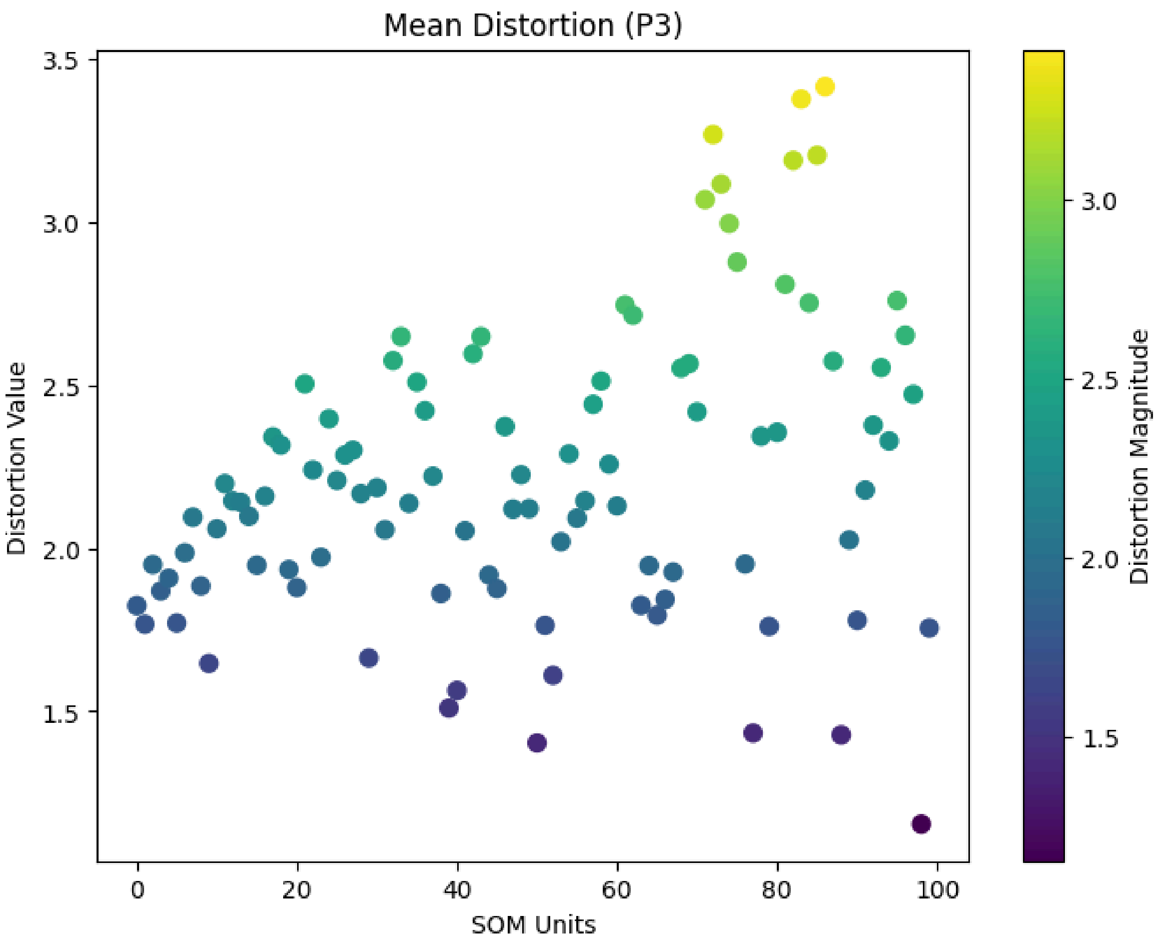
In [259…

```
mean_disto_6 = np.array(list_mean_disto_6)
mean_disto_6 = mean_disto_6.reshape(som_xdim, som_ydim)
plt.figure(figsize=(8, 6))
sns.heatmap(mean_disto_6, cmap='coolwarm', annot=False)
plt.title('Mean Distortion (P3)')
plt.xlabel('SOM X-Dimension')
plt.ylabel('SOM Y-Dimension')
plt.show()
```



In [260…

```
plt.figure(figsize=(8, 6))
plt.scatter(range(len(list_mean_disto_6)), list_mean_disto_6, c=list_mean_disto_6,
plt.colorbar(label='Distortion Magnitude')
plt.title('Mean Distortion (P3)')
plt.xlabel('SOM Units')
plt.ylabel('Distortion Value')
plt.show()
```



P: Topographic product

Overall distortion metric across all units and neighborhood orders, using the logarithmic average of P_3

The exact formula for computing P is the following:

$$P = \frac{1}{N(N-1)} \sum_{j=1}^{N} \sum_{k=1}^{N-1} \log(P_3(j,k))$$

However, in practice, with our implementation, we can't iterate over $N-1$ orders of neighborhoods. Indeed, we are using Manhattan distances to compute the matrix of SOM grid distances, which limits the range of possible k. For example, with the initial example of the notebook (iris dataset with 10×10 SOM), we can't go above 11 neighbors, because some units only have 10 unique SOM distances to their neighbors. Thus, we decide to limit k to this maximum number of available unique neighbors. Nevertheless, this is not very rigorous: we indeed observe (with the initial setting) that P_3 values tend to increase as k increases.

In [261…

```
def topographic_product(som_distances, weights_distances, max_neighb):
    p = 0
    n = som_distances.shape[0]
    for i in range(1,n):
        s = 0
        for k in range(1, max_neighb):
```

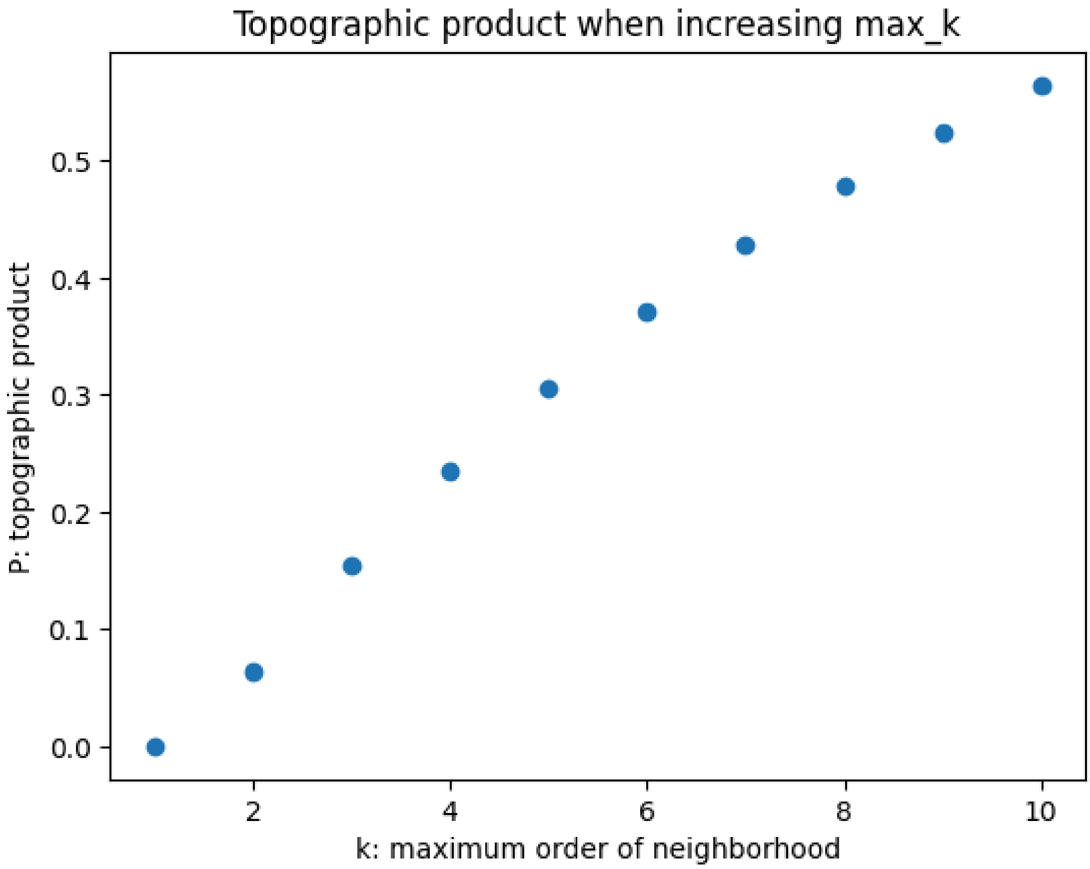
```
            s += np.log(mean_distortion(som_distances, weights_distances, i, k))
        p += s
    return p/(n*max_neighb)
```

In [262…

```
list_p = []
for k in range(1,11):
    list_p.append(topographic_product(som_distances, weights_distances, k))

plt.scatter(range(1,11), list_p)
plt.xlabel("k: maximum order of neighborhood")
plt.ylabel("P: topographic product")
plt.title("Topographic product when increasing max_k")
```

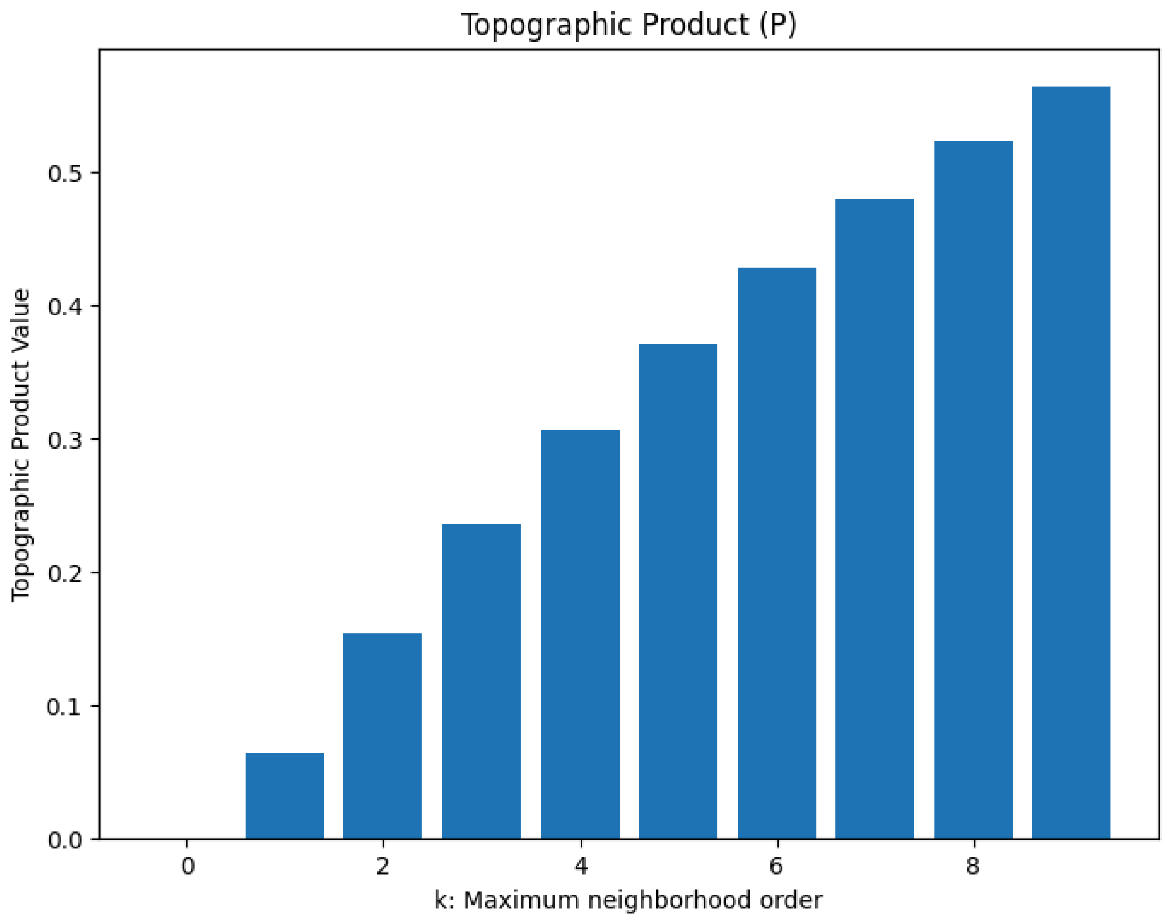
Out[262…

```
Text(0.5, 1.0, 'Topographic product when increasing max_k')
```



In [263…

```
plt.figure(figsize=(8, 6))
plt.bar(range(len(list_p)), list_p)
plt.title('Topographic Product (P)')
plt.xlabel('k: Maximum neighborhood order')
plt.ylabel('Topographic Product Value')
plt.show()
```



In [264…

```
import os
from prov.model import ProvDocument, Namespace
from datetime import datetime

def log_provenance(visualization_name, parameters, output_file):
    # Initialize PROV document
    prov_doc = ProvDocument()
    prov_doc.set_default_namespace('http://www.w3.org/ns/prov#')

    # Define namespaces
    process_ns = Namespace('process', 'http://example.org/process#')
    param_ns = Namespace('param', 'http://example.org/param#')
    output_ns = Namespace('output', 'http://example.org/output#')

    prov_doc.add_namespace(process_ns)
    prov_doc.add_namespace(param_ns)
    prov_doc.add_namespace(output_ns)

    # Define the process
    process = prov_doc.activity(process_ns[visualization_name], datetime.now())

    # Document parameters
    for key, value in parameters.items():
        param_entity = prov_doc.entity(param_ns[key], {'prov:value': value})
        prov_doc.wasGeneratedBy(param_entity, process)

    # Document output file
    output_entity = prov_doc.entity(output_ns[output_file], {'prov:type': 'File'})
    prov_doc.wasGeneratedBy(output_entity, process)

    # Ensure provenance directory exists
    os.makedirs("provenance", exist_ok=True)
```

```
    # Save provenance as a JSON
    prov_doc.serialize(destination=f"provenance/{visualization_name}_provenance.js

    return prov_doc


    return prov_doc
```

C) Evaluation Report

1) Testing

This first module loads the chainlink and 10cluster dataset and is used to test the modules for the Topographic Product based on these two datasets

For testing we create a small example:

In [265…

```
# Small example SOM and weight distance matrices (3x3 neurons)
som_distances = np.array([
    [0.0, 1.0, 2.0],
    [1.0, 0.0, 1.0],
    [2.0, 1.0, 0.0]
])

weights_distances = np.array([
    [0.0, 0.5, 1.5],
    [0.5, 0.0, 1.0],
    [1.5, 1.0, 0.0]
])

get_kth_neighbor_temp = get_kth_neighbor

# Simulated k-neighbor function (for test purposes)
def get_kth_neighbor(unit_i, distances, k):
    sorted_indices = np.argsort(distances[unit_i])  # Sort by distance
    return sorted_indices[:k], np.mean(distances[unit_i, sorted_indices[:k]])




# Dummy mean distortion function (modify for actual use)
def mean_distortion(som_distances, weights_distances, unit_i, k):
    p1 = distortion_input_space(som_distances, weights_distances, unit_i, k)
    p2 = distortion_output_space(som_distances, weights_distances, unit_i, k)
    return (p1 + p2)
```

Test 1: distortion_input_space

This tests whether input space distortion (P1) is non-negative and behaves as expected.

In [266…

```
def test_distortion_input_space():
    unit_i = 1  # Test neuron
    max_k = 2  # Small k for easy verification
```

```
    result = distortion_input_space(som_distances, weights_distances, unit_i, max_
    print(result)
    assert result >= 0, "P1 distortion should be non-negative"
    print(f"Test 1 Passed: distortion_input_space = {result}")

test_distortion_input_space()
```

```
0.0
Test 1 Passed: distortion_input_space = 0.0
```

Here we already found a bug, we previously got a division by 0 if there were no neighbors.

Test 2: distortion_output_space

This verifies that output space distortion (P2) is valid.

In [267…

```
def test_distortion_output_space():
    unit_i = 1  # Test neuron
    max_k = 2

    result = distortion_output_space(som_distances, weights_distances, unit_i, max
    assert result >= 0, "P2 distortion should be non-negative"
    print(f"Test 2 Passed: distortion_output_space = {result}")

test_distortion_output_space()
```

```
Test 2 Passed: distortion_output_space = 1.0
```

Here we found the same division by 0 problem.

Test 3: topographic_product

This tests if the topographic product behaves correctly.

In [268…

```
def test_topographic_product():
    max_neighb = 2  # Small max neighborhood for easy checking

    result = topographic_product(som_distances, weights_distances, max_neighb)

    # TP should normally be non-negative unless something is wrong
    assert result >= 0 or np.isnan(result), "Topographic product should not be neg
    print(f"Test 3 Passed: topographic_product = {result}")

test_topographic_product()
```

```
Test 3 Passed: topographic_product = 0.0
```

Step 3: Edge Cases and Robustness Testing

Edge Case 1: All Neurons Identical

If all neurons have the same weights, distortion should be zero.

In [269...

```
def test_identical_weights():
    n = 3  # Number of neurons
    identical_som_distances = np.zeros((n, n))
    identical_weights_distances = np.zeros((n, n))

    result = topographic_product(identical_som_distances, identical_weights_distar
    assert result == 0, "If all weights are identical, distortion should be zero"
    print(f"Edge Case Test Passed: Identical Weights, TP = {result}")

test_identical_weights()
```

```
Edge Case Test Passed: Identical Weights, TP = 0.0
```

Edge Case 2: Random Large Matrix

This checks if the functions handle large inputs without errors.

In [270...

```
def test_large_matrices():
    # For reproducibility
    np.random.seed(42)
    n = 50
    som_distances_large = np.random.rand(n, n)
    weights_distances_large = np.random.rand(n, n)

    som_distances_large = (som_distances_large + som_distances_large.T) / 2
    weights_distances_large = (weights_distances_large + weights_distances_large.T

    result = topographic_product(som_distances_large, weights_distances_large, max
    print(f"Large Matrix Test Passed: TP = {result}")

test_large_matrices()
```

```
Large Matrix Test Passed: TP = 1.31492325749606
```

In [271...

```
from SOMToolBox_Parse import SOMToolBox_Parse

cluster_idata = SOMToolBox_Parse("10clusters\\10clusters.vec").read_weight_file()
cluster_weights_toolbox = SOMToolBox_Parse("10clusters\\10clusters.wgt.gz").read_w

chainlink_idata = SOMToolBox_Parse("chainlink\\chainlink.vec").read_weight_file()
chainlink_weights_toolbox = SOMToolBox_Parse("chainlink\\chainlink.wgt.gz").read_w
```

We then test parameters for both datasets for the soms of size 10 x 10 and 100 x 60.

We then perform tests on these visualizations with our implementation and the java implementation.

For this we save the resulting weights. We will skip the large ones for testing because they just take too long.

In [272...

```
from minisom import MiniSom
get_kth_neighbor = get_kth_neighbor_temp
def train_som(data, x_dim, y_dim, iterations, sigma, learning_rate):
    som = MiniSom(x_dim, y_dim, data.shape[1], sigma=sigma, learning_rate=learning
    som.train(data, iterations, random_order=True)
    computed_weights = som.get_weights()
```

```
    return computed_weights

def compute_pairwise_som_distances_minisom(weights):
    weights_arr = weights
    som_xdim, som_ydim = weights.shape[0], weights.shape[1]
    grid_positions = np.array([[i, j] for i in range(som_xdim) for j in range(som_
    som_distances = np.abs(grid_positions[:, np.newaxis] - grid_positions).sum(axi
    weights_distances = euclidean_distances(weights_arr.reshape(-1, weights_arr.sh
    return som_distances, weights_distances


def create_topological_product_visualizations(name, weights, k=10, lr=0.5, sigma=1
    # Compute distances and initialize variables
    som_distances, weights_distances = compute_pairwise_som_distances_minisom(weig
    dim = som_distances.shape[0]

    distortion_input_space_list = []
    distortion_output_space_list = []
    distortion_mean_list = []

    # Compute distortions
    for i in range(dim):
        distortion_input_space_list.append(distortion_input_space(som_distances, w
        distortion_output_space_list.append(distortion_output_space(som_distances,
        distortion_mean_list.append(mean_distortion(som_distances, weights_distanc

    list_p = []
    for l in range(1,k+1):
        list_p.append(topographic_product(som_distances, weights_distances, l))

    # Convert lists to numpy arrays and reshape them
    # Convert lists to numpy arrays and reshape them
    distortion_input_space_list = np.array(distortion_input_space_list).reshape(we
    distortion_output_space_list = np.array(distortion_output_space_list).reshape(
    distortion_mean_list = np.array(distortion_mean_list).reshape(weights.shape[0]

    # Create figure with subplots: 3 heatmaps + 1 bar plot
    fig, axes = plt.subplots(1, 4, figsize=(22, 6), gridspec_kw={'width_ratios': [

    fig.suptitle(f'Distortion Analysis and Topographic Product of {name} k:{k} lr:

    # Titles for the heatmaps
    titles = ['Distortion in Input Space (P1)', 'Distortion in Output Space (P2)',
    data = [distortion_input_space_list, distortion_output_space_list, distortion_

    # Generate heatmaps
    for i, ax in enumerate(axes[:3]):
        sns.heatmap(data[i], cmap='coolwarm', annot=False, ax=ax, cbar=False)
        ax.set_title(titles[i])
        ax.set_xlabel('SOM X-Dimension')
        ax.set_ylabel('SOM Y-Dimension')

    # Bar plot on the right
    axes[3].bar(range(len(list_p)), list_p)
    axes[3].set_title(f'Topographic Product')
    axes[3].set_xlabel('k: Maximum neighborhood order')
    axes[3].set_ylabel('Topographic Product Value')

    # Adjust layout and show the figure
    plt.tight_layout()
    plt.show()
```

```python
    # Create all visualizations

data_cluster = cluster_idata['arr']
data_chainlink = chainlink_idata['arr']

def perform_experiment_small(sigma, lr, k):
    weights = train_som(data_cluster, 10, 10, 10000, sigma, lr)
    create_topological_product_visualizations("10clusters", weights, k=k, lr=lr)

    weights = train_som(data_chainlink, 10, 10, 10000, sigma, lr)
    create_topological_product_visualizations("chainlink", weights, k=k, lr=lr)


def perform_experiment_large(sigma, lr, k):
    weights = train_som(data_cluster, 100, 60, 1000, sigma, lr)
    create_topological_product_visualizations("10clusters", weights, k=k, lr=lr)

    weights = train_som(data_chainlink, 100, 60, 1000, sigma, lr)
    create_topological_product_visualizations("chainlink", weights, k=k, lr=lr)

def perform_experiment(sigma, lr, k):
    log_provenance(f"Topographic_Product_Visualization_Experiment_sigma_sigma_{sig
    perform_experiment_small(sigma, lr, k)
    perform_experiment_large(sigma, lr, k)
```

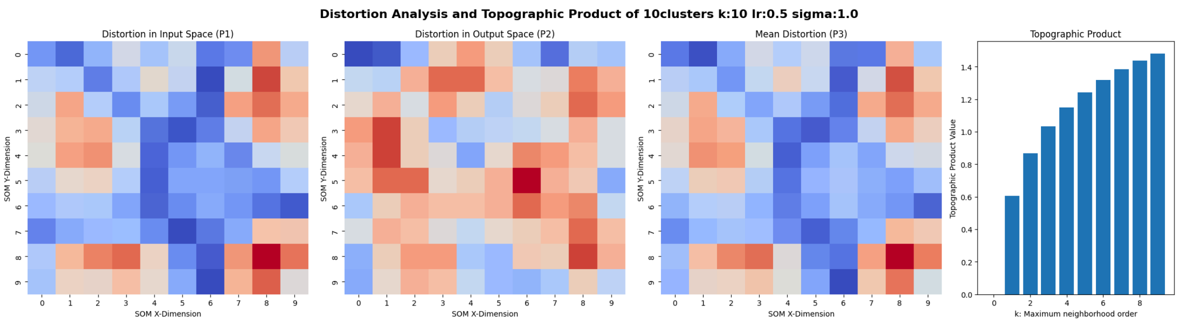
Experiment Setup

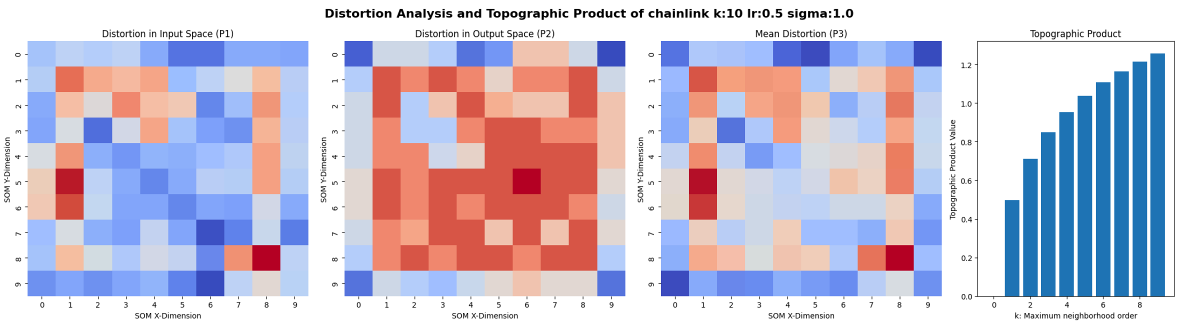
We will run tests on training soms (10, 10) and (100, 60) for both the chainlink and 10 clusters dataset. The resulting visualizations will always be in the order: (10, 10) 10clusters (10, 10) chainlink

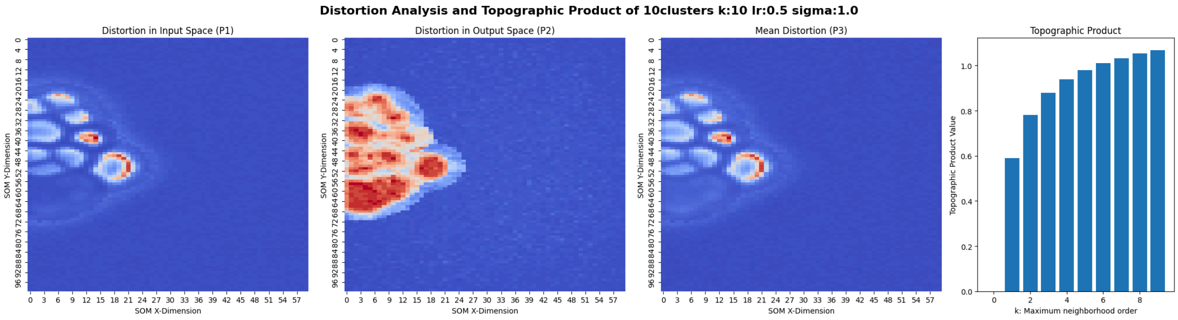
(100, 60) 10clusters (100, 60) chainlink

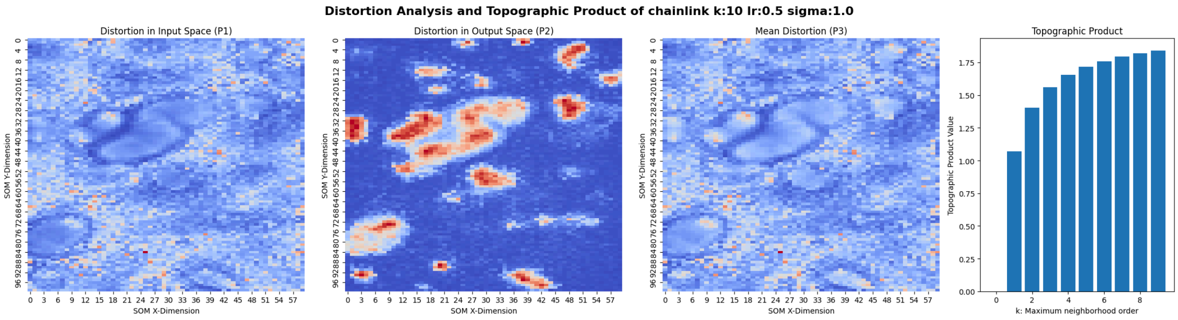
In [273…

```python
perform_experiment(4, 0.5, 10)
```







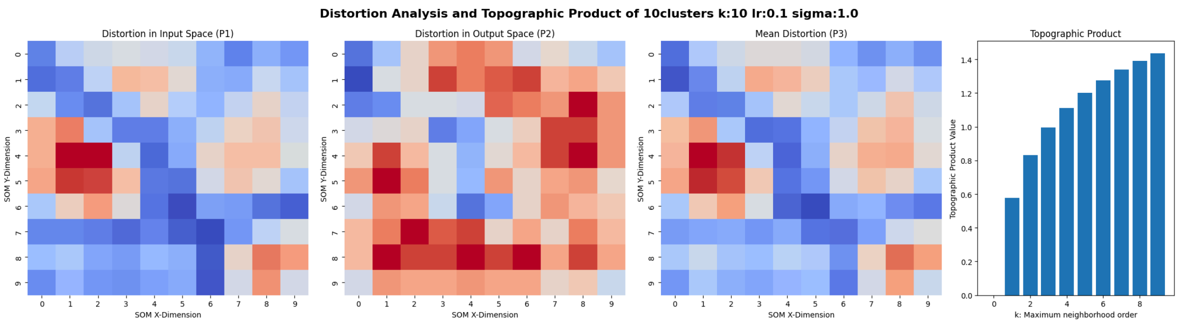


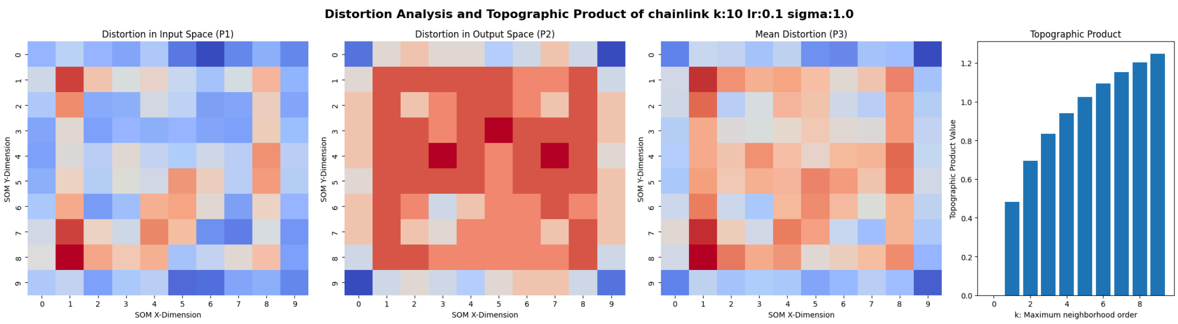
From the results for a moderate learning rate and a high sigma we can see that the topographic product works well for the small soms, but struggles for the large soms as it is negative which is not desired. We also see large distortions in the outupt space indicating that these are not the right parameters.

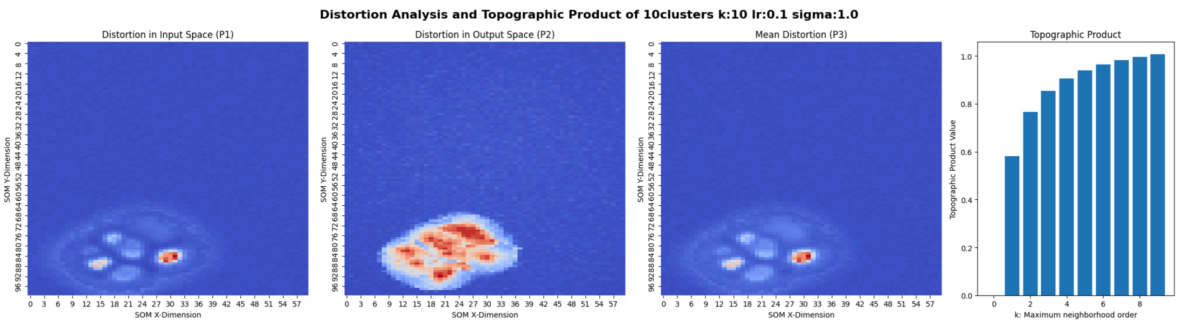
Analysis:

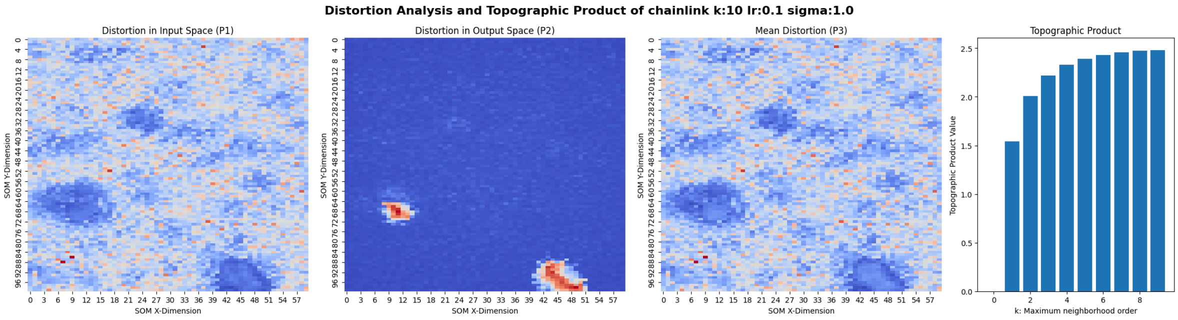
In [274…

```
perform_experiment(4, 0.1, 10)
```





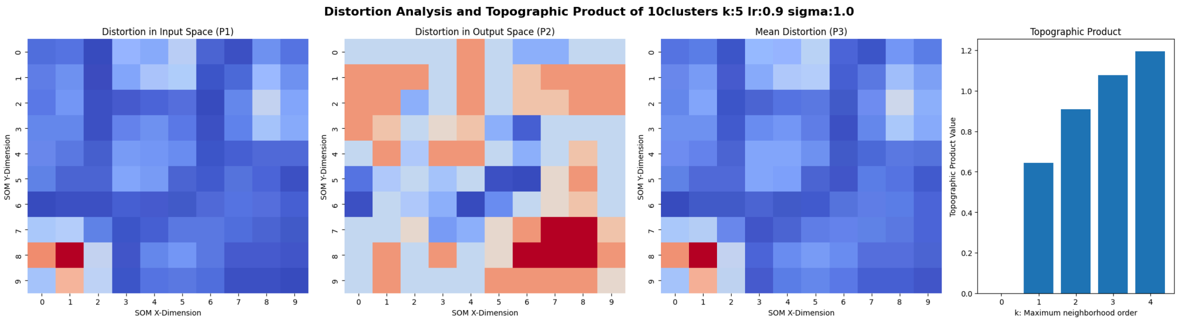


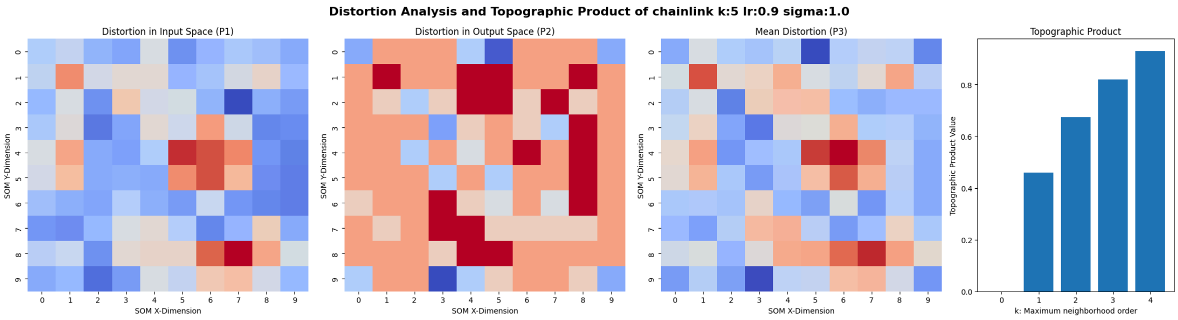


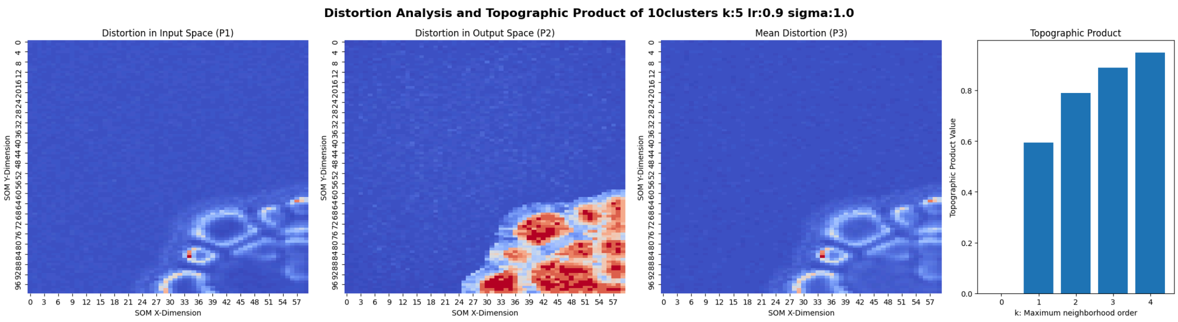
For a small learning rate we see that the input data is not well mapped in the chainlink dataset and that there is one nasty cluster in the output space, the topological product also just became slightly better.

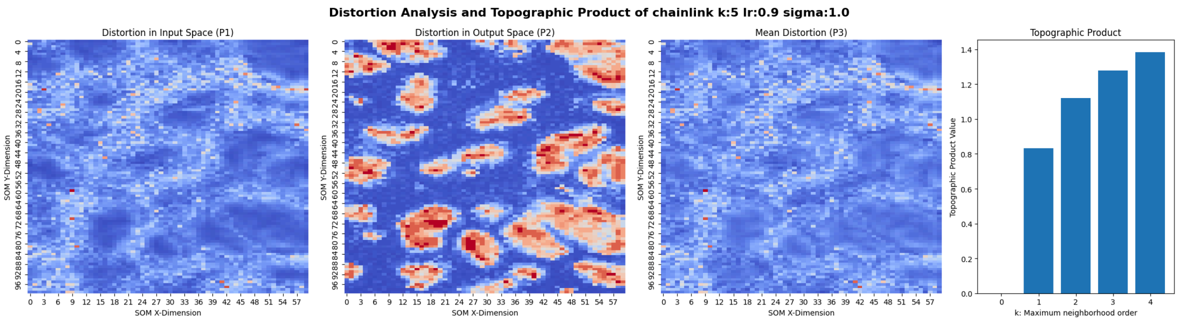
In [275…

```
perform_experiment(4, 0.9, 5)
```







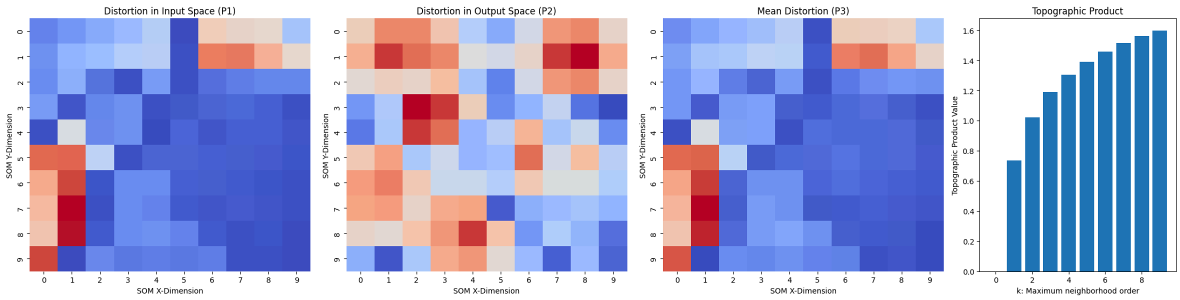


With a large learning rate and a lower k we get large clusters of distortions in the output space, and chainlink is a complete mess this means that it completely fails here. Only the topological product is better and becomes positive for the large soms.

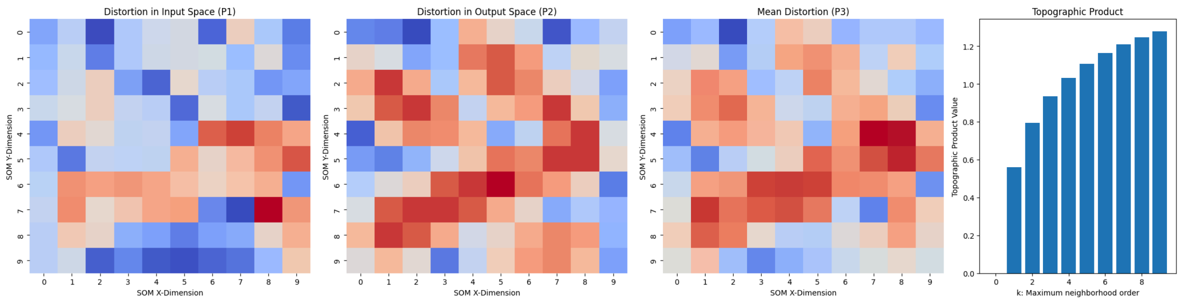
In [276…

```
perform_experiment(1, 0.9, 10)
```

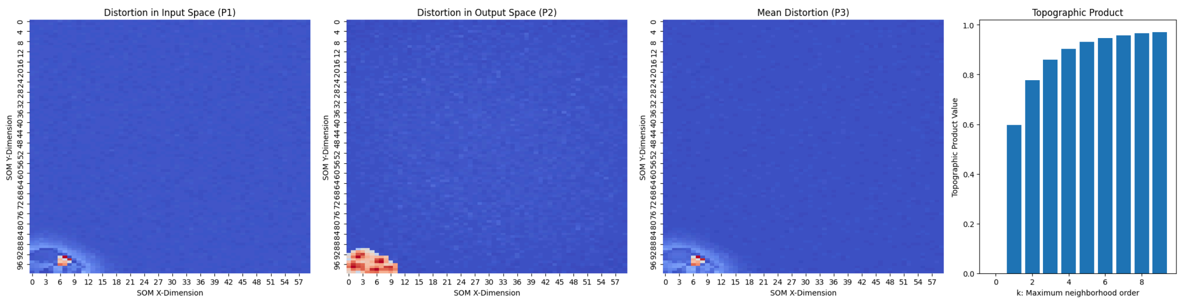
Distortion Analysis and Topographic Product of 10clusters k:10 lr:0.9 sigma:1.0



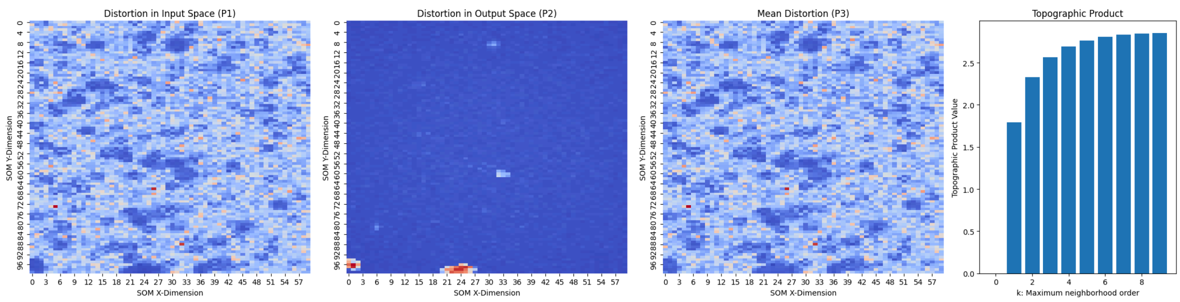
Distortion Analysis and Topographic Product of chainlink k:10 lr:0.9 sigma:1.0



Distortion Analysis and Topographic Product of 10clusters k:10 lr:0.9 sigma:1.0



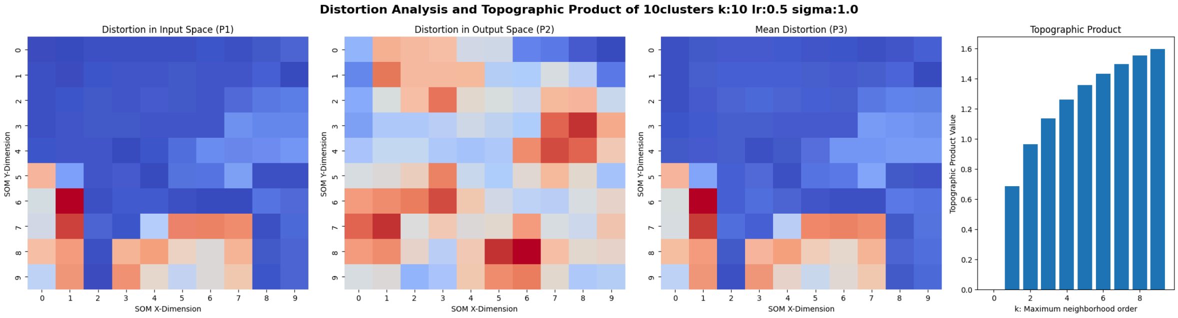
Distortion Analysis and Topographic Product of chainlink k:10 lr:0.9 sigma:1.0

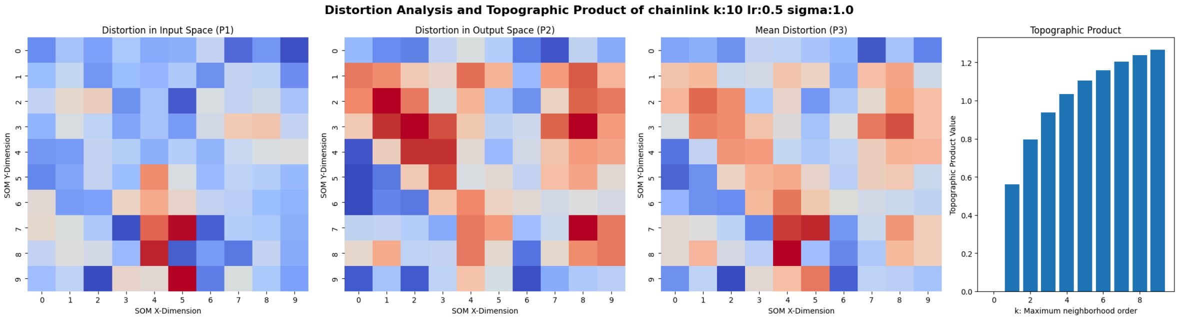


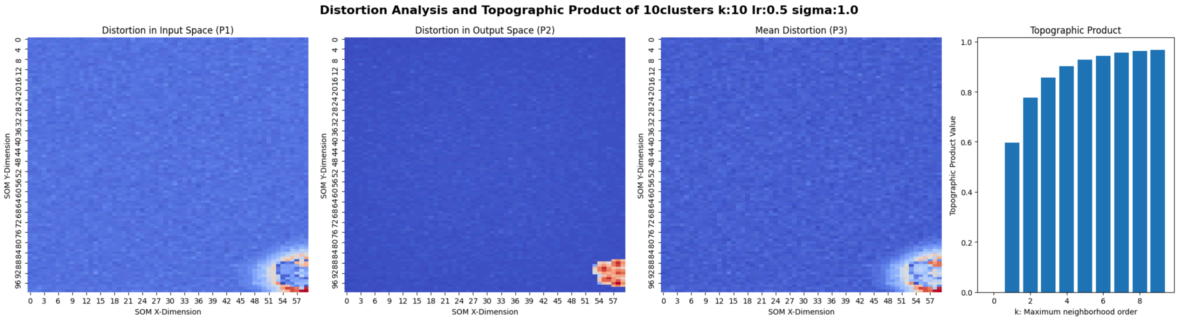
Here we clearly see catastrophic results.

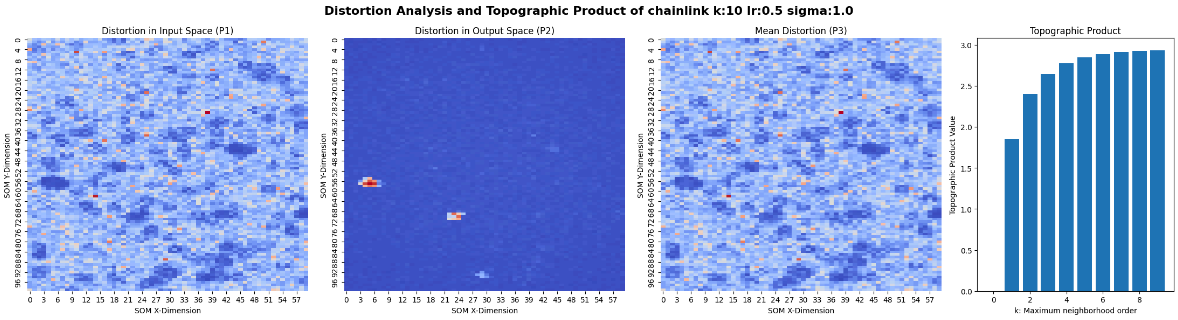
In [277…

```
perform_experiment(1, 0.5, 10)
```





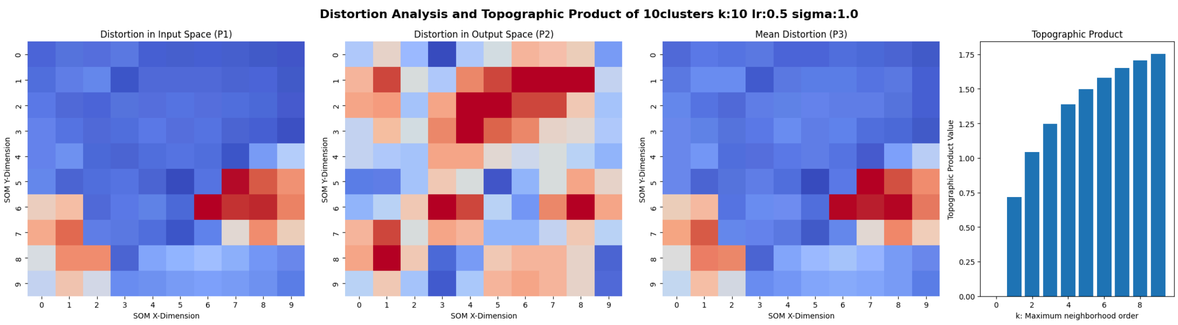


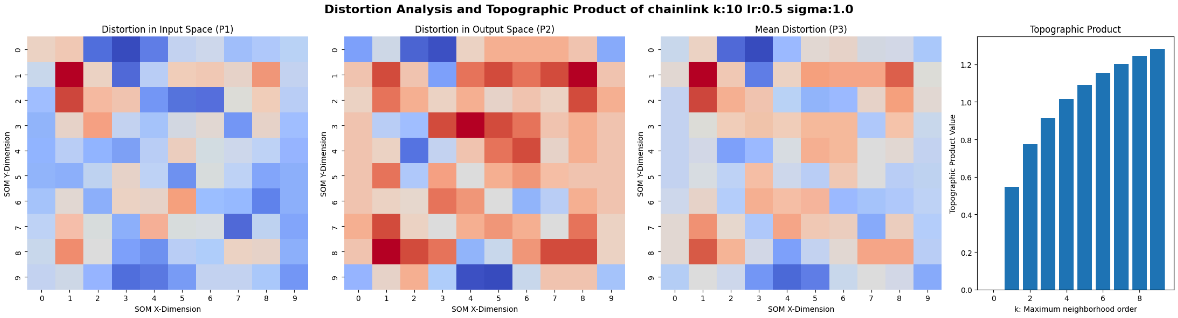


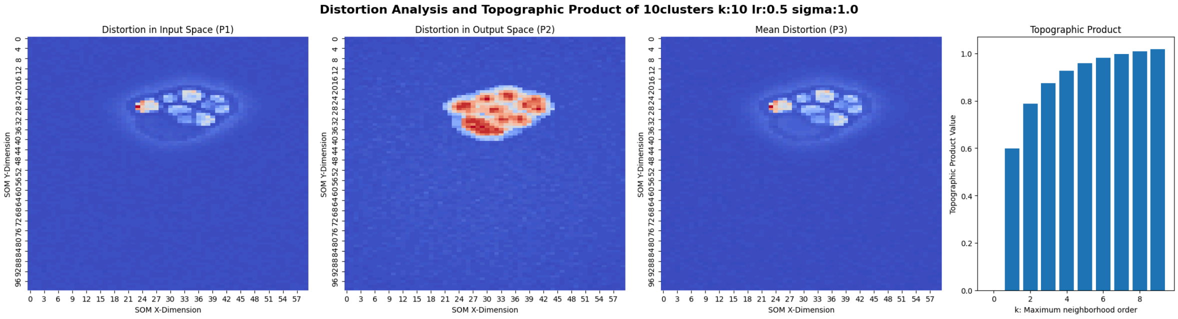
Here by using a lower value for sigma (1) in minisom we get much better results for the chainlink dataset, but the 10clusters dataset performs very poorly in terms of the topological product, for this reason we see that a lower sigma is better but something is still missing for large soms.

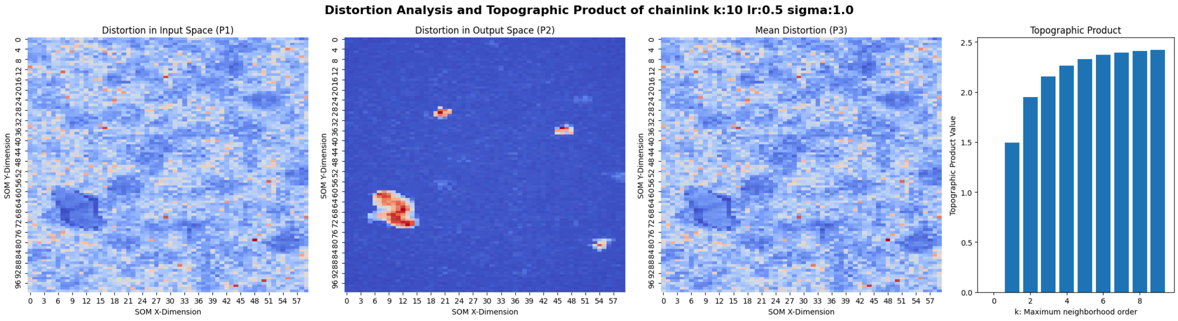
In [278…

```
perform_experiment(2, 0.5, 10)
```





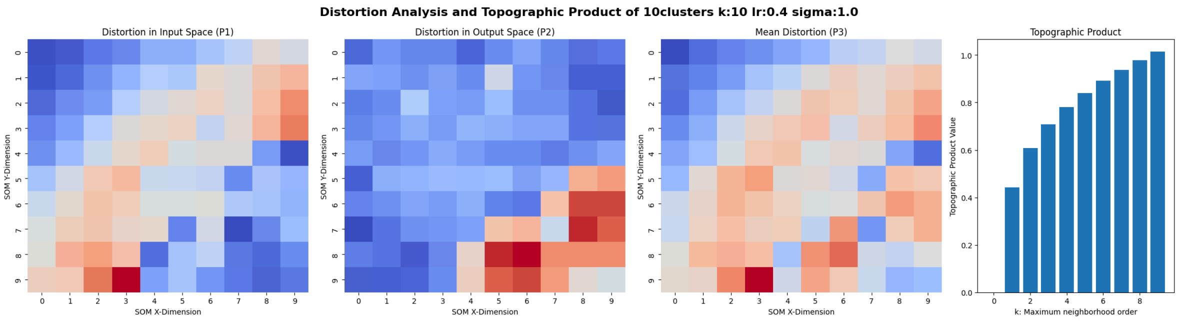


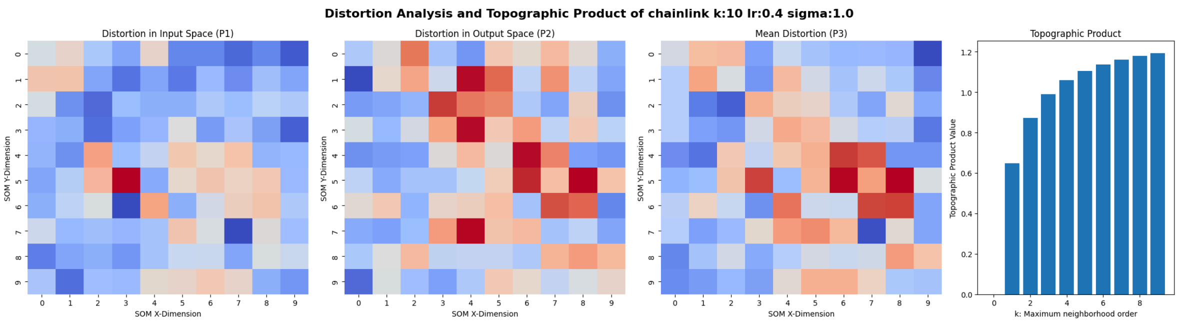


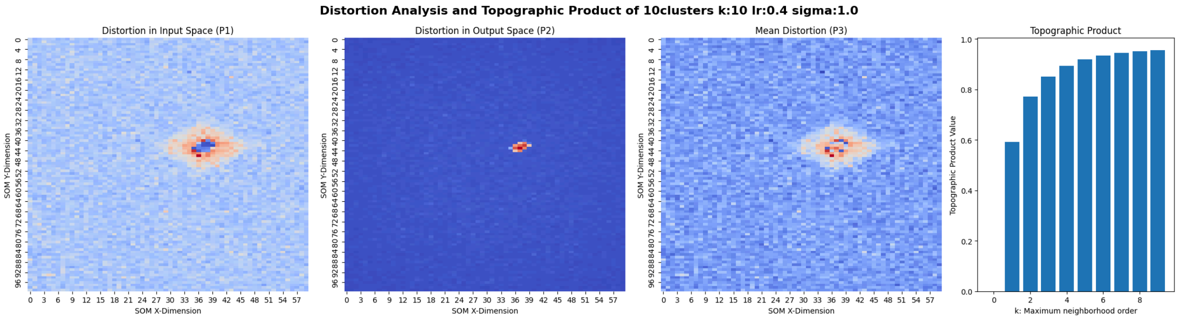
Here is the first time that we get problems with the small soms, the input space is mapped perfectly but the grid structure and neurons are extremely disorganized in the output space. The output for the large soms is similarily extremely poor.

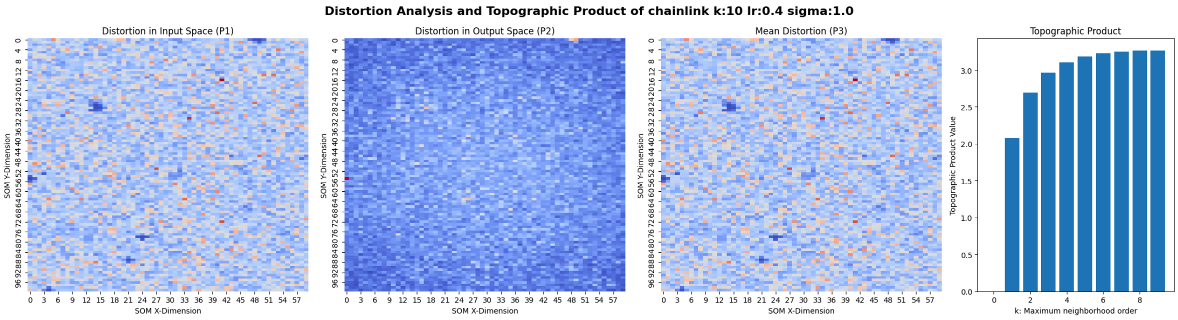
In [279…

```
perform_experiment(0.5, 0.4, 10)
```









For the last test with a very low sigma we finally get very good results for the large soms, indicating that a smaller sigma is more beneficial, we only struggled with the chainlink som where this evidently leads to large distortions and a bad topographic production > 3. For this we see that ideally parameters should be tailored for different datasets.