Supplementary File: Data Analysis and Codes (All codes are in Python)

CONTENTS

APPENDIX A
DATA ANALYSIS:

We conducted our data analysis in two phases. In the first phase we tested $\mathcal{I}^*$ index with XB, DB and Dunn's index in batch clustering setup (i.e., when the complete data set is available) to see which one performs best. In the second phase we tested incremental $\mathcal{I}^*$ index with incremental XB and incremental DB index in online streaming data setup to check which one performs best. For both the cases we used synthetic data along with real data to perform our comparative analysis.

A. Batch Clustering Setup:

We used 6 different synthetic data of varying dimensions for this setup and 3 different real data sets. Real datasets can be found in UCI repository. Table 1 contains a summary of the datasets used for the experimental results. For each of the synthetic data sets we generated a random positive definite covariance matrix and random data points for each cluster center from multivariate Gaussian distributions with varying mean and covariance. Now using that covariance matrix and cluster centers obtained we generated data points for each cluster from another multivariate normal distribution with mean and covariance specified above. For datasets S-1, S-2 and S-3 we generated balanced clusters (i.e., each cluster have roughly similar number of data points). On the other hand, S-4, S-5 and S-6 have unbalanced clusters (i.e., one or two clusters contain majority of the points).

Real datasets are Iris, Cervical Cancer and Gene Expression datasets. Iris data has 4 attributes and all of them are numerical. Iris data has 3 different labels I.e., 3 different clusters. Cervical Cancer dataset has 19 numeric attributes and 2 different clusters while Gene expression dataset has 20531 numeric attributes and 5 different clusters. The number of clusters indicated by the different indices for the nine data sets are tabulated in Table 2.

Name	Dimension	Cluster	points
S-1	3	3	1500
S-2	2	5	1000
S-3	3	10	1500
S-4	4	7	3500
S-5	20	4	65000
S-6	113	6	3000
Iris	4	3	150
Cervical Cancer	19	2	72
Gene Expression	20531	5	801

TABLE I: Batch Clustering Dataset Description

Name	Cluster	XB	DB	$\mathcal{I}^*$	Dunn
S-1	3	3	3	3	2
S-2	5	5	5	5	5
S-3	10	9	8	10	10
S-4	7	6	6	7	3
S-5	4	2	2	4	2
S-6	6	2	2	5	2
Iris	3	2	2	3	3
Cervical Cancer	2	2	3	2	4
Gene Expression	5	2	2	2	2

TABLE II: Values of the four indices for the different datasets

Name	Dim	Cluster	Points	Data Flow
S-7	3	2-4-5	5000	2K-2K-1K
S-8	10	2-6-8	8000	4K-3K-1K
S-9	5	2-3-5-7	7000	3K-1K-1K-2K
S-10	111	2-3-4	5000	1K-3K-1K
S-11	3	3	1500	1.5K
S-12	2	2-3-4-5	16000	4K-4K-4K-4K
S-13	3	2-5-7	20000	10K-5K-5K
S-14	2	6	6000	6K
Iris	4	2-3	150	50-100
Cervical Cancer	19	2	72	72

TABLE III: Streaming data description

As we can see barring from S-6 and Gene Expression data $\mathcal{I}^*$ index predicts the number of clusters accurately for each of the datasets. In contrast, Xie-Beni index fails to predict the accurate number of clusters 6 times. DB and Dunn's index also fail to predict the accurate number of clusters 6 times each. In this context, $\mathcal{I}^*$ index clearly outperforms all the other indices. Note that for the S-6 dataset, $\mathcal{I}^*$ index prediction is the closest to the actual number of clusters. The above result gives us confidence to extend the $\mathcal{I}^*$ index in an online setup to get a better index than IXB and IDB indices.

B. Online Streaming Data Setup:

We used 8 synthetic datasets and 2 real datasets for this part. Our primary goal was to see how the different indices perform when the number of clusters changes over time. In Table 3 we have noted the key characteristics of each of the datasets (i.e., their dimensions, number of clusters over time, number of datapoints and how the data is arriving). We used 4 different streaming data clustering techniques, which are SK-means, SK-means with exponentially fading memory, Inverse Weighted K-means for online clustering version-1 and Inverse Weighted K-means for online clustering version-2, for this purpose.

The dataflow column in the table-3 represents how the data is coming in the stream. For example, for S-10 cluster set is given by 2-3-4, and dataflow is given by 1K-3K-1K, which implies that the first 1K data points are coming from 2 clusters, the next 3K data points are coming from 3 clusters (first two clusters included and a new cluster joined), and the final 1K data points are coming from 4 clusters (3 clusters of the previous stage included and a new cluster added to the list). MacQueen's SK-mean algorithm needs only one parameter: the number of clusters, so we do not need to worry about it. On the other hand, SK-mean with exponentially fading memory requires another parameter, namely the forgetting factor. We currently fix the forgetting factor value to 0.9 for clarity, but for any general value of the forgetting factor, the indices can be obtained (We have provided the codes in supplementary resources). For the other two clustering algorithms, IWK-ov1 and IWK-ov2, the two parameters n and p are set to n=2 and p=6, respectively, but again we can evaluate the indices for any positive values of n and p if n¡p (codes for these algorithms are provided in supplementary resources as well). For datasets S-7, S-8, S-9, S-10, and S-11, we generated random mean vectors from gamma distributions and generated random positive definite covariance matrices. Using those, we generated clusters from multivariate normal distributions. For S-12 and S-13, we generated circularly shifting data clusters from multivariate normal distributions with an added noise cluster in the center. S-14 is linearly shifted Gaussian data with one noise cluster lying outside the chain of the clusters (see fig-1)

1) Performance Analysis::

1) S-7: IDB performs the worst for each of the 4 algorithms it predicts that there are around 9-11 clusters for different time points. While IXB with SK-mean performs moderately well predicting in the range of 3-9 clusters. $\mathcal{I}^*$ index with SK-mean performs best in this scenario predicting 2/4 clusters in different time points. $\mathcal{I}^*$ with exponential weight decay also performs reasonably well predicting 4-7 clusters in different time points.
2) S-8: IDB again performs the worst predicting 10- 11 clusters for each time point for all the setups. IXB performs reasonably well predicting 7-11 clusters for 3 different setups but for exponential weight decay it only predicts 2 clusters. $\mathcal{I}^*$ index with exponential weight decay predicts 3-9 clusters for each data point (with number of clusters increasing gradually) and $\mathcal{I}^*$ index with IWK-ov2 predicts 3-6 clusters in different time points

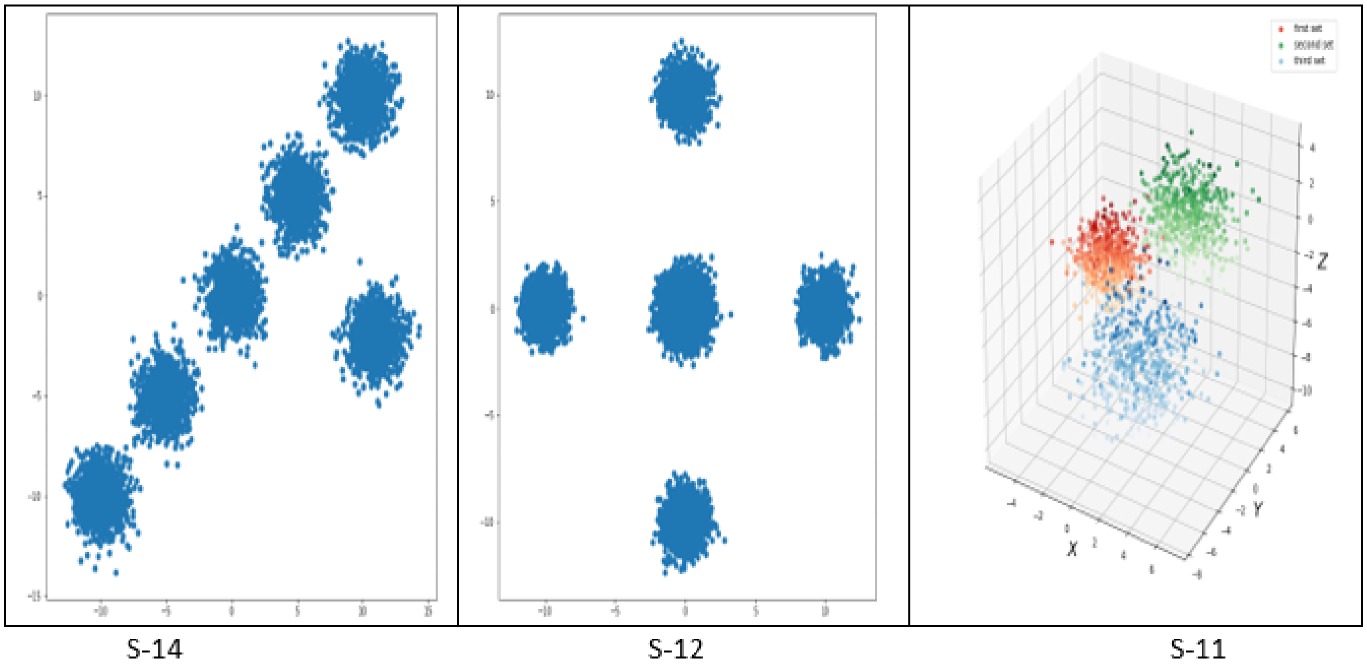

Fig. 1: Data Plots

Name	Best Performer	2^{nd} Best Performer
S-7	$\mathcal{I}^*$-SK-mean	$\mathcal{I}^*$-Exp Weight Decay
S-8	XB-IWK-ov1	$\mathcal{I}^*$-Exp Weight Decay
S-9	$\mathcal{I}^*$-IWK-ov1	XB-SK-mean
S-10	$\mathcal{I}^*$-Exp Weight Decay	$\mathcal{I}^*$-IWK-ov2
S-11	$\mathcal{I}^*$-IWK-ov2	$\mathcal{I}^*$-Exp Weight Decay
S-12	$\mathcal{I}^*$-IWK-ov2	$\mathcal{I}^*$-Exp Weight Decay
S-13	XB-SK-mean	DB with IWK-ov1
S-14	$\mathcal{I}^*$-IWK-ov1	$\mathcal{I}^*$-IWK-ov2
Iris	$\mathcal{I}^*$-Exp Weight Decay	XB-Exp Weight Decay
Cervical Cancer	$\mathcal{I}^*$-Exp Weight Decay	$\mathcal{I}^*$-IWK-ov2

TABLE IV: Top Performers for Streaming Data

3) S-9: IXB with SK mean predicts that 3 or 6 cluster at each time point and $\mathcal{I}^*$ index with IWK-ov1 predicts 3-7 clusters at each time point. For all the other cases each of the indices fails to produce a reasonably good estimate of the true number of clusters.
4) S-10: $\mathcal{I}^*$ index with exponential weight decay outperforms all other indices predicting correctly for most of the time points. $\mathcal{I}^*$ index with IWK-ov2 also performs reasonably well but IXB and IDB produces estimates in the range of 8-11.
5) S-11: $\mathcal{I}^*$ index with IWK-ov2 performs best it predicts 3 clusters for almost all the time points. $\mathcal{I}^*$ index with exponential wight decay also produces good estimate. IXB and IDB produces estimates which are much higher than 3(true number of clusters)
6) S-12: $\mathcal{I}^*$ index with IWK-ov2 performs best. $\mathcal{I}^*$ index with exponential wight decay also produces good estimate predicting the change points accurately as well. IXB with SK-mean and IWK-ov1 also give good estimates of the actual number of clusters for different time points.
7) S-13: IXB with SK-mean and IDB with IWK-ov1 produces best estimates for this data. $\mathcal{I}^*$ with exponential weight decay also produces good estimates
8) S-14: $\mathcal{I}^*$ index with IWK-ov1 and IWK-ov2 outperforms all other indices for this data set.
9) Iris: $\mathcal{I}^*$ index with Exponential Weight decay and IXB with exponential Weight decay performs reasonably well. Other indices fail to produce a good estimate of the number of clusters for different time points.
10) Cervical Cancer: $\mathcal{I}^*$ index with Exponential Weight decay and $\mathcal{I}^*$ index with IWK-ov2 predicts 2 clusters correctly. All other indices predicts that there is 8+ clusters, which is far away from the actual number of clusters.

As can be seen from Table-4, the $\mathcal{I}^*$ index with exponential weight decay and $\mathcal{I}^*$ index with IWK-ov2 outperforms all other indices. The difference between the indices gets starker when the number of dimensions increases. IXB index also produces reasonable estimates of the actual number of clusters in many situations, but IDB fails to produce reasonable estimates in almost all cases.

APPENDIX B
LIBRARIES:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
from mpl_toolkits.mplot3d import Axes3D
import sklearn
from sklearn.cluster import KMeans
from sklearn.metrics import davies_bouldin_score
import jqmcvi
from jqmcvi import base
from sklearn.preprocessing import OneHotEncoder
from scipy import random, linalg
```

APPENDIX C
SK-MEAN:

```
def seqKmean(k,X):
  if len(X)<k:
    print("length is an issue")
  else:
    centroids = X[:k]
    counter = np.ones(k)
    for x_n in X[k:]:
      closest_k = find_closest_cluster(x_n,centroids)
      counter[closest_k] +=1
      temp = np.subtract(x_n,centroids[closest_k])
      temp = np.divide(temp,counter[closest_k])
      centroids[closest_k] = np.add(centroids[closest_k],temp)
    return centroids , counter
#closest cluster
def find_closest_cluster(x, centroids):
  closest_cluster, closest_distance = 999999, 999999 # initially invalid
  K = len(centroids)
  for k in range(K): # no. clusters
    temp = 0
    for j in range(len(centroids[k])):
      temp = temp + (x[j]-centroids[k][j])**2
    distance = temp # Euclidean distance
    if distance < closest_distance:
      closest_cluster  = k
      closest_distance = distance
  return closest_cluster
```

APPENDIX D
FINDING E-VALUES UNDER GENERAL SETUP:

```
    #forumla to evaluate e_K for any given positive integer K , K = cluster number , X = data
    returns an array of values
def eK(K,X):
  if len(X)<K:
    print("length is an issue")
  else:
    centroids = X[:K]
    counter = np.ones(K)
    p = np.zeros(K)
    temp_len = len(X) - K
    eK = np.zeros(temp_len)
    y = len(X[K])
    temp_len = (K,y)
    g = np.zeros(temp_len)

    for j in range(K,len(X),1):
      closest_K = find_closest_cluster(X[j],centroids)
      m = counter.copy()
      counter[closest_K] +=1

      temp3 = np.subtract(counter,m)
      temp_centroids = centroids.copy()
      temp = np.subtract(X[j],temp_centroids[closest_K])
      temp = np.divide(temp, counter[closest_K])
      temp_centroids[closest_K] = np.add(temp_centroids[closest_K],temp)
      #update stage
      for i in range(K):
        #print("Cluster " + str(i))
        temp1 = np.subtract(centroids[i],temp_centroids[i])

        temp1 = np.transpose(temp1)
        q = np.matmul(temp1,g[i])
        #print(q[i])
        b = norm(np.subtract(centroids[i],temp_centroids[i]))**2
        #print(b[i])
        a = temp3[i] * (norm(np.subtract(X[j],temp_centroids[i]))**2)
        #print(a[i])
        p[i] = p[i] + (2*q) + (b*m[i]) + a
        #print(p[i])
        g[i] = g[i] + np.multiply(m[i],np.subtract(centroids[i],temp_centroids[i]) +
        np.multiply(temp3[i],np.subtract(X[j],temp_centroids[i]))

      centroids = temp_centroids.copy()


      eK[j-K] = np.sum(p)
  return eK
```

APPENDIX E
FINDING E-VALUES UNDER EXPONENTIAL DECAY SITUATION:

```
  def eK(K,X,l):
if len(X)<K:
  print("length is an issue")
else:
  centroids = X[:K]
  counter = np.ones(K)
  p = np.zeros(K)
  temp_len = len(X) - K
  eK = np.zeros(temp_len)
  temp_len = (K,len(X[K]))
  g = np.zeros(temp_len)

  for j in range(K,len(X),1):
    closest_K = find_closest_cluster(X[j],centroids)
    m = counter.copy()
```

```
      counter[closest_K] +=1

      temp3 = np.subtract(counter,m)
      temp_centroids = centroids.copy()
      temp = np.subtract(X[j],temp_centroids[closest_K])
      temp = np.multiply(temp, l)
      temp_centroids[closest_K] = np.add(temp_centroids[closest_K],temp)
      #update stage
      for i in range(K):
        #print("Cluster " + str(i))
        temp1 = np.subtract(centroids[i],temp_centroids[i])

        temp1 = np.transpose(temp1)
        q = np.matmul(temp1,g[i])
        #print(q[i])
        b = norm(np.subtract(centroids[i],temp_centroids[i]))**2
        #print(b[i])
        a = temp3[i] * (norm(np.subtract(X[j],temp_centroids[i]))**2)
        #print(a[i])
        p[i] = (l*p[i]) + (2*q*l) + (b*m[i]*l) + a
        #print(p[i])
        g[i] = (l*g[i]) + np.multiply((l*m[i]),np.subtract(centroids[i],temp_centroids[i])) +
        np.multiply(temp3[i],np.subtract(X[j],temp_centroids[i]))

      centroids = temp_centroids.copy()


      eK[j-K] = np.sum(p)
  return eK
```

Appendix F
Functions for Xie-Beni, $\mathcal{I}^*$ index , Dunn's index and DB index using K-means

```
    #Functions for evaluating XB, I* , dunn and DB index in general setting using regular
    k means function
def ivalueg(data):
  kmeans = KMeans(n_clusters=1, random_state=0).fit(data)
  e1 = kmeans.inertia_
  output = []
  for k in range(2,15,1):
    kmeans = KMeans(n_clusters=k, random_state=0).fit(data)
    p = kmeans.labels_
    temp = kmeans.cluster_centers_
    eKtemp = kmeans.inertia_
    temp2 = 0
    for h in range(len(temp)):
        for x in range(len(temp)):
            v = np.linalg.norm(np.subtract(temp[h],temp[x]))
            if v>temp2:
                temp2 = v
    dmax = temp2
    t = np.divide(e1,eKtemp)
    t = np.multiply(t,np.multiply(dmax,dmax))
    t = t /((k+1)**2)
    output = output + [t]
  plt.plot(range(2,15,1),output,label = "Plot for I* index")
  plt.xlabel("Cluster Number")
  plt.legend()
```

```
def xbvalueg(data):
  output = []
  for k in range(2,15,1):
    kmeans = KMeans(n_clusters=k, random_state=0).fit(data)
    p = kmeans.labels_
    temp = kmeans.cluster_centers_
    eKtemp = kmeans.inertia_
    temp2 = 999999
    for h in range(len(temp)):
        for x in range(len(temp)):
            v = np.linalg.norm(np.subtract(temp[h],temp[x]))
            if v<temp2 and x!=h:
                temp2 = v
    dmax = temp2
    t = eKtemp/(len(data)*temp2*temp2)
    output = output + [t]
  plt.plot(range(2,15,1),output,label = "Plot for XB index")
  plt.xlabel("Cluster Number")
  plt.legend()

def dunnvalueg(data):
  output = []
  for k in range(2,15,1):
    kmeans = KMeans(n_clusters=k, random_state=0).fit(data)
    p = kmeans.labels_
    t = base.dunn_fast(data, p)
    output = output + [t]
  plt.plot(range(2,15,1),output,label = "Plot for Dunn's index")
  plt.xlabel("Cluster Number")
  plt.legend()

def dbvalueg(data):
  output = []
  for k in range(2,15,1):
    kmeans = KMeans(n_clusters=k, random_state=0).fit(data)
    p = kmeans.labels_
    t = davies_bouldin_score(data, p)
    output = output + [t]
  plt.plot(range(2,15,1),output,label = "Plot for DB index")
  plt.xlabel("Cluster Number")
  plt.legend()
def generalPlot(data):
  plt.subplot(2,2,1)
  ivalueg(data)
  plt.subplot(2,2,2)
  xbvalueg(data)
  plt.subplot(2,2,3)
  dunnvalueg(data)
  plt.subplot(2,2,4)
  dbvalueg(data)
  plt.show()
```

Appendix G
Functions for $\mathcal{I}^*$ index under SK-mean, exponential weight decay , IWK-ov1 and IWK-ov2

```
    #Function-1: Finding the closest cluster of a point given the point and centroids
def find_closest_cluster(x, centroids):
    closest_cluster, closest_distance = 999999, 999999 # initially invalid
    K = len(centroids)
    for k in range(K): # no. clusters
        temp = 0
        for j in range(len(centroids[k])):
            temp = temp + (x[j]-centroids[k][j])**2
        distance = temp # Euclidean distance
        if distance < closest_distance:
            closest_cluster  = k
            closest_distance = distance
    return closest_cluster
#Function-2: Sequential K-means
def sKmean(k,x_n,counter,centroids):
    closest_k = find_closest_cluster(x_n,centroids)
    counter[closest_k] +=1
    temp = np.subtract(x_n,centroids[closest_k])
    temp = np.divide(temp,counter[closest_k])
    centroids[closest_k] = np.add(centroids[closest_k],temp)
    return centroids, counter

def seqKmean(k,X):
    if len(X)<k:
        print("length is an issue")
    else:
        centroids = X[:k]
        counter = np.ones(k)
        for x_n in X[k:]:
            closest_k = find_closest_cluster(x_n,centroids)
            counter[closest_k] +=1
            temp = np.subtract(x_n,centroids[closest_k])
            temp = np.divide(temp,counter[closest_k])
            centroids[closest_k] = np.add(centroids[closest_k],temp)
        return centroids , counter

#Function-3: Calculating Euclidean Norm
def norm(v):
    temp = 0
    for i in range(len(v)):
        temp = temp + (v[i]**2)
    temp = temp**0.5
    return temp

#Function-4: Exponentially fading E_K
def eK2(K,X,l):
    if len(X)<K:
        print("length is an issue")
    else:
        centroids = X[:K]
        counter = np.ones(K)
        p = np.zeros(K)
        temp_len = len(X) - K
        eK = np.zeros(temp_len)
        temp_len = (K,len(X[K]))
        g = np.zeros(temp_len)

        for j in range(K,len(X),1):
            closest_K = find_closest_cluster(X[j],centroids)
            m = counter.copy()
            counter[closest_K] +=1

            temp3 = np.subtract(counter,m)
            temp_centroids = centroids.copy()
            temp = np.subtract(X[j],temp_centroids[closest_K])
            temp = np.multiply(temp, l)
            temp_centroids[closest_K] = np.add(temp_centroids[closest_K],temp)
            #update stage
            for i in range(K):
                temp1 = np.subtract(centroids[i],temp_centroids[i])
                temp1 = np.transpose(temp1)
                q = np.matmul(temp1,g[i])
                b = norm(np.subtract(centroids[i],temp_centroids[i]))**2
                a = temp3[i] * (norm(np.subtract(X[j],temp_centroids[i]))**2)
                p[i] = (l*p[i]) + (2*q*l) + (b*m[i]*l) + a
                g[i] = (l*g[i]) + np.multiply((l*m[i]),np.subtract(centroids[i],temp_centroids
+
                np.multiply(temp3[i],np.subtract(X[j],temp_centroids[i]))
            centroids = temp_centroids.copy()
            eK[j-K] = np.sum(p)
        return eK

#Function-5: Finding d_max for exponentially fading setting
def dmax2(K,X,l):
    if len(X)<K:
        print("length is an issue")
    else:
        d = []
        centroids = X[:K]
        counter = np.ones(K)
        for x_n in X[K:]:
            closest_k = find_closest_cluster(x_n,centroids)
            counter[closest_k] +=1
            temp = np.subtract(x_n,centroids[closest_k])
            temp = np.multiply(temp,l)
            centroids[closest_k] = np.add(centroids[closest_k],temp)
            for h in range(len(centroids)):
                temp2 = 0
                for x in range(len(centroids)):
                    v = np.linalg.norm(np.subtract(centroids[h],centroids[x]))
                    if v>temp2:
                        temp2 = v
            d = d + [temp2]
        return d
#Function-6: Finding E_K for general setting
def eK(K,X):
    if len(X)<K:
        print("length is an issue")
    else:
        centroids = X[:K]
        counter = np.ones(K)
        p = np.zeros(K)
        temp_len = len(X) - K
        eK = np.zeros(temp_len)
        y = len(X[K])
        temp_len = (K,y)
        g = np.zeros(temp_len)
        for j in range(K,len(X),1):
            closest_K = find_closest_cluster(X[j],centroids)
            m = counter.copy()
            counter[closest_K] +=1

            temp3 = np.subtract(counter,m)
            temp_centroids = centroids.copy()
            temp = np.subtract(X[j],temp_centroids[closest_K])
            temp = np.divide(temp, counter[closest_K])
            temp_centroids[closest_K] = np.add(temp_centroids[closest_K],temp)
            #update stage
            for i in range(K):
                #print("Cluster " + str(i))
                temp1 = np.subtract(centroids[i],temp_centroids[i])

                temp1 = np.transpose(temp1)
                q = np.matmul(temp1,g[i])
                #print(q[i])
                b = norm(np.subtract(centroids[i],temp_centroids[i]))**2
                #print(b[i])
                a = temp3[i] * (norm(np.subtract(X[j],temp_centroids[i]))**2)
                #print(a[i])
                p[i] = p[i] + (2*q) + (b*m[i]) + a
                #print(p[i])
                g[i] = g[i] + np.multiply(m[i],np.subtract(centroids[i],temp_centroids[i]))
+
                np.multiply(temp3[i],np.subtract(X[j],temp_centroids[i]))

            centroids = temp_centroids.copy()


            eK[j-K] = np.sum(p)
        return eK
#Function-7: Calculating d_max in general setting
def dmax(K,X):
    if len(X)<K:
        print("length is an issue")
    else:
        d = []
        centroids = X[:K]
        counter = np.ones(K)
        for x_n in X[K:]:
            closest_k = find_closest_cluster(x_n,centroids)
            counter[closest_k] +=1
            temp = np.subtract(x_n,centroids[closest_k])
            temp = np.divide(temp,counter[closest_k])
            centroids[closest_k] = np.add(centroids[closest_k],temp)
            for h in range(len(centroids)):
                temp2 = 0
                for x in range(len(centroids)):
                    v = np.linalg.norm(np.subtract(centroids[h],centroids[x]))
                    if v>temp2:
                        temp2 = v
            d = d + [temp2]
        return d

#Function-8: Generating plot for I2-I11 in general setting
def final(name,data):
```

```
t = np.divide(eK(1,data)[1:],eK(2,data))
d = dmax(2,data)
t = np.multiply(t,np.multiply(d,d))
t = t/4
I2= t

t = np.divide(eK(1,data)[2:],eK(3,data))
d = dmax(3,data)
t = np.multiply(t,np.multiply(d,d))
t = t/9
I3= t
t = np.divide(eK(1,data)[3:],eK(4,data))
d = dmax(4,data)
t = np.multiply(t,np.multiply(d,d))
t = t/16
I4= t
t = np.divide(eK(1,data)[4:],eK(5,data))
d = dmax(5,data)
t = np.multiply(t,np.multiply(d,d))
t = t/25
I5= t
t = np.divide(eK(1,data)[5:],eK(6,data))
d = dmax(6,data)
t = np.multiply(t,np.multiply(d,d))
t = t/36
I6 = t
t = np.divide(eK(1,data)[6:],eK(7,data))
d = dmax(7,data)
t = np.multiply(t,np.multiply(d,d))
t = t/49
I7= t
t = np.divide(eK(1,data)[7:],eK(8,data))
d = dmax(8,data)
t = np.multiply(t,np.multiply(d,d))
t = t/64
I8= t
t = np.divide(eK(1,data)[8:],eK(9,data))
d = dmax(9,data)
t = np.multiply(t,np.multiply(d,d))
t = t/81
I9= t
t = np.divide(eK(1,data)[9:],eK(10,data))
d = dmax(10,data)
t = np.multiply(t,np.multiply(d,d))
t = t/100
I10= t
t = np.divide(eK(1,data)[10:],eK(11,data))
d = dmax(11,data)
t = np.multiply(t,np.multiply(d,d))
t = t/121
I11= t

I2 = I2[9:]
I3 = I3[8:]
I4 = I4[7:]
I5 = I5[6:]
I6 = I6[5:]
I7 = I7[4:]
```

```
    I8 = I8[3:]
    I9 = I9[2:]
    I10 = I10[1:]

    temp = [I2,I3,I4,I5,I6,I7,I8,I9,I10,I11]
    temp = np.asarray(temp)
    temp2 = np.argmax(temp, axis=0)

    temp2 = np.add(temp2,np.repeat(2,len(temp2)))


    plt.plot(temp2,label = "Expected cluster number: SK-mean")


    plt.legend()

def final2(l,name,data):
    t = np.divide(eK2(1,data,l)[1:],eK2(2,data,l))
    d = dmax2(2,data,l)
    t = np.multiply(t,np.multiply(d,d))
    t = t/4
    I2= t

    t = np.divide(eK2(1,data,l)[2:],eK2(3,data,l))
    d = dmax2(3,data,l)
    t = np.multiply(t,np.multiply(d,d))
    t = t/9
    I3= t
    t = np.divide(eK2(1,data,l)[3:],eK2(4,data,l))
    d = dmax2(4,data,l)
    t = np.multiply(t,np.multiply(d,d))
    t = t/16
    I4= t
    t = np.divide(eK2(1,data,l)[4:],eK2(5,data,l))
    d = dmax2(5,data,l)
    t = np.multiply(t,np.multiply(d,d))
    t = t/25
    I5= t
    t = np.divide(eK2(1,data,l)[5:],eK2(6,data,l))
    d = dmax2(6,data,l)
    t = np.multiply(t,np.multiply(d,d))
    t = t/36
    I6 = t
    t = np.divide(eK2(1,data,l)[6:],eK2(7,data,l))
    d = dmax2(7,data,l)
    t = np.multiply(t,np.multiply(d,d))
    t = t/49
    I7= t
    t = np.divide(eK2(1,data,l)[7:],eK2(8,data,l))
    d = dmax2(8,data,l)
    t = np.multiply(t,np.multiply(d,d))
    t = t/64
    I8= t
    t = np.divide(eK2(1,data,l)[8:],eK2(9,data,l))
    d = dmax2(9,data,l)
    t = np.multiply(t,np.multiply(d,d))
    t = t/81
    I9= t
    t = np.divide(eK2(1,data,l)[9:],eK2(10,data,l))
    d = dmax2(10,data,l)
    t = np.multiply(t,np.multiply(d,d))
    t = t/100
    I10= t
    t = np.divide(eK2(1,data,l)[10:],eK2(11,data,l))
    d = dmax2(11,data,l)
    t = np.multiply(t,np.multiply(d,d))
    t = t/121
    I11= t
    plt.plot(I2[10:],label = "cluster-2")
    plt.plot(I3[9:],label = "cluster-3")
    plt.plot(I4[8:],label = "cluster-4")
    plt.plot(I5[7:],label = "cluster-5")
    plt.plot(I6[6:],label = "cluster-6")
    plt.plot(I7[5:],label = "cluster-7")
    plt.plot(I8[4:],label = "cluster-8")
    plt.plot(I9[3:],label = "cluster-9")
    plt.plot(I10[2:],label = "cluster-10")
    plt.plot(I11[1:],label = "cluster-11")
    plt.title("Exponential Weight Decay")
    plt.legend()
    plt.savefig(name)

def iwek(K,X,p,n):
  d = []
  s = (K,len(X[0]))
  centroids = np.zeros(s)
  v = np.ones(K)
  pu = np.zeros(K)
  temp_len = len(X)
  eK = np.zeros(temp_len)
  y = len(X[0])
  temp_len = (K,y)
  g = np.zeros(temp_len)
  j=0
  for x_n in X:
    closest_K = find_closest_cluster(x_n,centroids)
    a = np.ones(K)

    temp3 = 0


    for k in range(K):
      if k != closest_K:
        temp3 = temp3 + 1/((np.linalg.norm(np.subtract(x_n,centroids[k])))**p)
        temp = p* (((np.linalg.norm(np.subtract(x_n,centroids[closest_K])))**n)/((np.linalg.no
        a[k] = temp
        #print(centroids[k])
        #print(v[k])
        temp5 = np.multiply(centroids[k],v[k])
        temp6 = np.multiply(a[k],x_n)
        temp7 = np.add(temp5,temp6)
        temp7 = np.divide(temp7,(v[k]+a[k]))
        #print(temp7)

        temp1 = np.subtract(centroids[k],temp7)
        temp1 = np.transpose(temp1)
        q = np.matmul(temp1,g[k])
        b = norm(np.subtract(centroids[k],temp7))**2
        au = a[k] * (norm(np.subtract(X[j],temp7))**2)
        pu[k] = pu[k] + (2*q) + (b*v[k]) + au
        g[k] = g[k] + np.multiply(v[k],np.subtract(centroids[k],temp7)) + np.multiply(a[k],np.
        centroids[k] = temp7
        v[k] = v[k] + a[k]
    temp1 = (np.linalg.norm(np.subtract(x_n,centroids[k])))**(n-p-2)
    temp1 = (p-n)*temp1
    temp2 = np.linalg.norm(np.subtract(x_n,centroids[closest_K]))
    temp2 = temp2**(n-2)
    temp2 = (-n)*temp2
    temp4 = temp1 + temp2
    a[closest_K] = temp4
    k = closest_K

    temp5 = np.multiply(centroids[k],v[k])
    temp6 = np.multiply(a[k],x_n)
    temp7 = np.add(temp5,temp6)
    temp7 = np.divide(temp7,(v[k]+a[k]))
    temp1 = np.subtract(centroids[k],temp7)
    temp1 = np.transpose(temp1)
    q = np.matmul(temp1,g[k])
    b = norm(np.subtract(centroids[k],temp7))**2
    au = a[k] * (norm(np.subtract(X[j],temp7))**2)
    pu[k] = pu[k] + (2*q) + (b*v[k]) + au
    g[k] = g[k] + np.multiply(v[k],np.subtract(centroids[k],temp7)) + np.multiply(a[k],np.subt
    centroids[k] = temp7
    v[k] = v[k] + a[k]

    eK[j] = np.sum(pu)
    j=j+1
    for h in range(len(centroids)):
      temp2 = 0
      for x in range(len(centroids)):
        vu = np.linalg.norm(np.subtract(centroids[h],centroids[x]))
        if vu>temp2:
          temp2 = vu
    d = d + [temp2]

  return eK,d
#Function for e1
def iwe1(X,p,n,K=1):
  d = []
  s = (1,len(X[0]))
  centroids = np.zeros(s)
  v = 1
  pu = 0
  temp_len = len(X)
  eK = np.zeros(temp_len)
  y = len(X[0])
  temp_len = (K,y)
  g = np.zeros(temp_len)
  j=0
  for x_n in X:
    closest_K = find_closest_cluster(x_n,centroids)
    a = np.ones(K)
```

```
      temp3 = 0

      k = closest_K
      temp7 = np.multiply(j,centroids[k])
      temp7 = np.add(temp7,x_n)
      temp7 = np.divide(temp7,(j+1))



      temp1 = np.subtract(centroids[k],temp7)
      temp1 = np.transpose(temp1)
      q = np.matmul(temp1,g[k])
      b = norm(np.subtract(centroids[k],temp7))**2
      au = a[k] * (norm(np.subtract(X[j],temp7))**2)
      pu = pu + (2*q) + (b*v) + au
      g[k] = g[k] + np.multiply(v,np.subtract(centroids[k],temp7)) + np.multiply(a[k],np.subtrac
      centroids[k] = temp7


      v = v +1
      eK[j] = np.sum(pu)
      j=j+1


    return eK

  #Function for eK and dmax
  def tiwek(K,X,p,n):
    d = []
    s = (K,len(X[0]))
    centroids = np.zeros(s)
    v = np.ones(K)
    pu = np.zeros(K)
    temp_len = len(X)
    eK = np.zeros(temp_len)
    y = len(X[0])
    temp_len = (K,y)
    g = np.zeros(temp_len)
    j=0
    for x_n in X:
      closest_K = find_closest_cluster(x_n,centroids)
      a = np.ones(K)

      temp3 = 0


      for k in range(K):
        temp3 = temp3 + 1/((np.linalg.norm(np.subtract(x_n,centroids[k])))**p)
        temp = p* (((np.linalg.norm(np.subtract(x_n,centroids[closest_K])))**n)/((np.linalg.norm
        a[k] = temp
          #print(centroids[k])
          #print(v[k])
        temp5 = np.multiply(centroids[k],v[k])
        temp6 = np.multiply(a[k],x_n)
        temp7 = np.add(temp5,temp6)
        temp7 = np.divide(temp7,(v[k]+a[k]))
          #print(temp7)
        temp1 = np.subtract(centroids[k],temp7)
        temp1 = np.transpose(temp1)
        q = np.matmul(temp1,g[k])
        b = norm(np.subtract(centroids[k],temp7))**2
        au = a[k] * (norm(np.subtract(X[j],temp7))**2)
        pu[k] = pu[k] + (2*q) + (b*v[k]) + au
        g[k] = g[k] + np.multiply(v[k],np.subtract(centroids[k],temp7)) + np.multiply(a[k],np.su
        centroids[k] = temp7
        v[k] = v[k] + a[k]




      eK[j] = np.sum(pu)
      j=j+1
      for h in range(len(centroids)):
        temp2 = 0
        for x in range(len(centroids)):
          vu = np.linalg.norm(np.subtract(centroids[h],centroids[x]))
          if vu>temp2:
            temp2 = vu
      d = d + [temp2]

    return eK,d
  def iwkPlot(data,p,n):
    a1,d1 = iwek(2,data,p,n)
    f1 = iwe1(data,p,n)
    t1 = np.divide(f1,a1)
    t1 = np.multiply(t1,np.multiply(d1,d1))
    t1 = t1/4
    I2= t1

    a,d = iwek(3,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/9
    I3= t

    a,d = iwek(4,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/16
    I4= t

    a,d = iwek(5,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/25
    I5= t

    a,d = iwek(6,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/36
    I6= t

    a,d = iwek(7,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/49
    I7= t

    a,d = iwek(8,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/64
    I8= t

    a,d = iwek(9,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/81
    I9= t

    a,d = iwek(10,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/100
    I10= t

    a,d = iwek(11,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/121
    I11= t

    temp = [I2,I3,I4,I5,I6,I7,I8,I9,I10,I11]
    temp = np.asarray(temp)
    temp2 = np.argmax(temp, axis=0)

    temp2 = np.add(temp2,np.repeat(2,len(temp2)))


    plt.plot(temp2,label = "Expected cluster number: IWK-ov-1")


    plt.legend()

  def tiwkPlot(data,p,n):
    a1,d1 = tiwek(2,data,p,n)
    f1 = iwe1(data,p,n)
    t1 = np.divide(f1,a1)
    t1 = np.multiply(t1,np.multiply(d1,d1))
    t1 = t1/4
    I2= t1

    a,d = tiwek(3,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/9
    I3= t

    a,d = tiwek(4,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/16
```

```
    I4= t

    a,d = tiwek(5,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/25
    I5= t

    a,d = tiwek(6,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/36
    I6= t

    a,d = tiwek(7,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/49
    I7= t

    a,d = tiwek(8,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/64
    I8= t

    a,d = tiwek(9,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/81
    I9= t

    a,d = tiwek(10,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/100
    I10= t

    a,d = tiwek(11,data,p,n)
    t = np.divide(f1,a)
    t = np.multiply(t,np.multiply(d,d))
    t = t/121
    I11= t
    temp = [I2,I3,I4,I5,I6,I7,I8,I9,I10,I11]
    temp = np.asarray(temp)
    temp2 = np.argmax(temp, axis=0)
    temp2 = np.add(temp2,np.repeat(2,len(temp2)))
    plt.plot(temp2,label = "Expected cluster number: IWK-ov-2")
    plt.legend()
  def Istar(data,p=2,n=6,l=0.9):
    plt.title("Plots for I* index")
    plt.subplot(2,2,1)
    final("First",data)
    plt.subplot(2,2,2)
    final2(l,"Second",data)
    plt.subplot(2,2,3)
    iwkPlot(data,p,n)
    plt.subplot(2,2,4)
```

```
tiwkPlot(data,p,n)
plt.show()
```

Appendix H
Additional Plots for Online Streaming Data Analysis

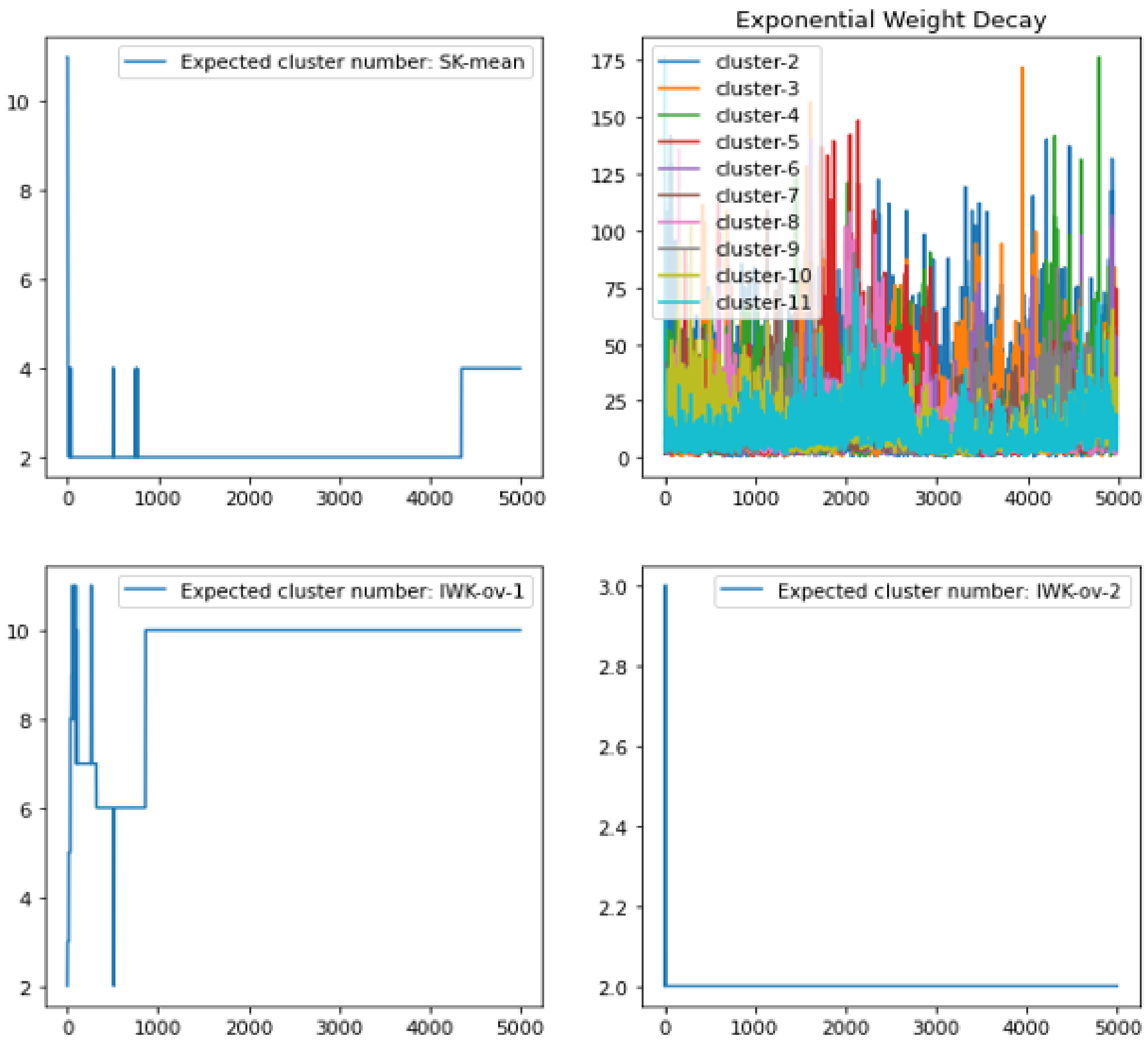

Fig. 2: S7: I* Index

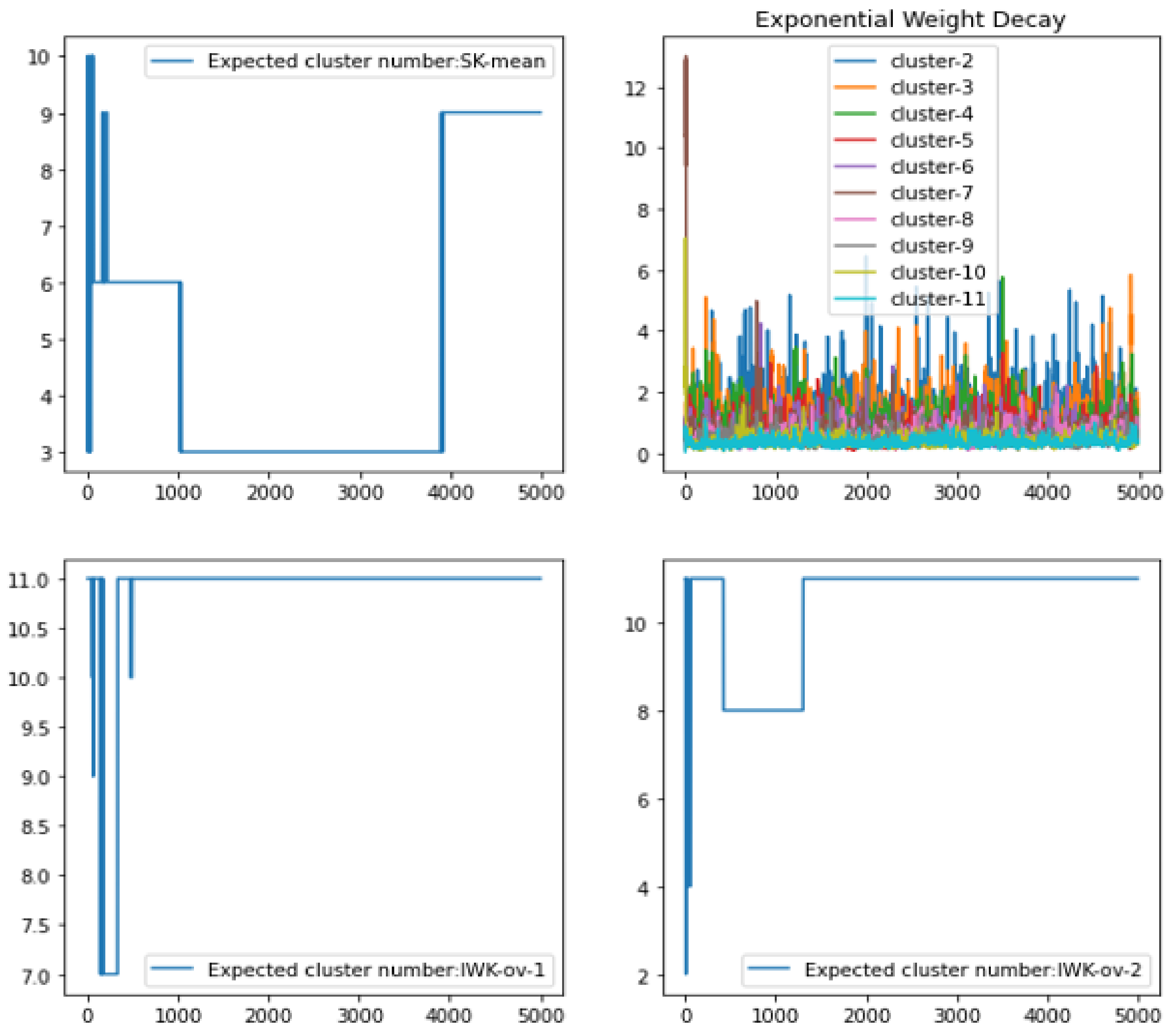

Fig. 3: S7: XB Index

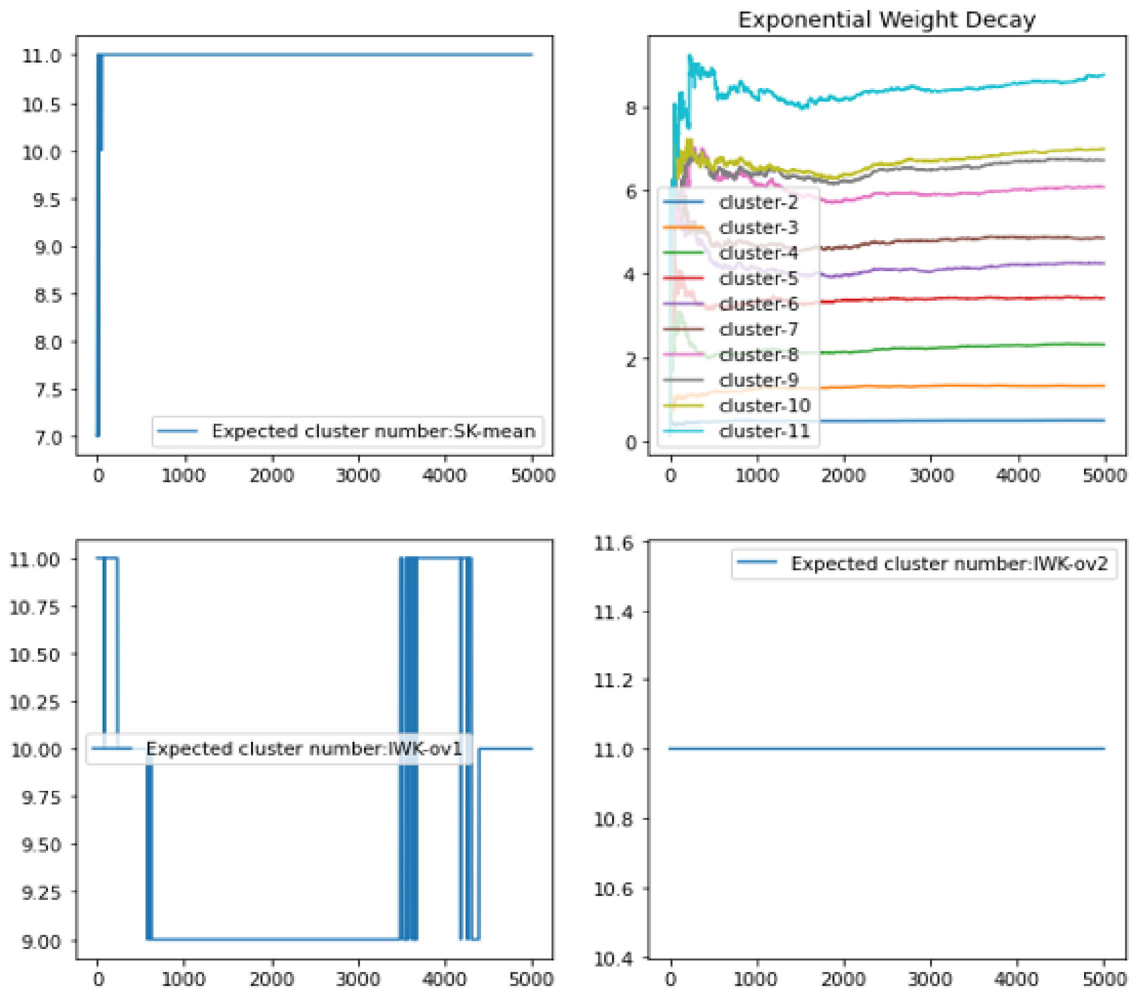

Fig. 4: S7: DB Index

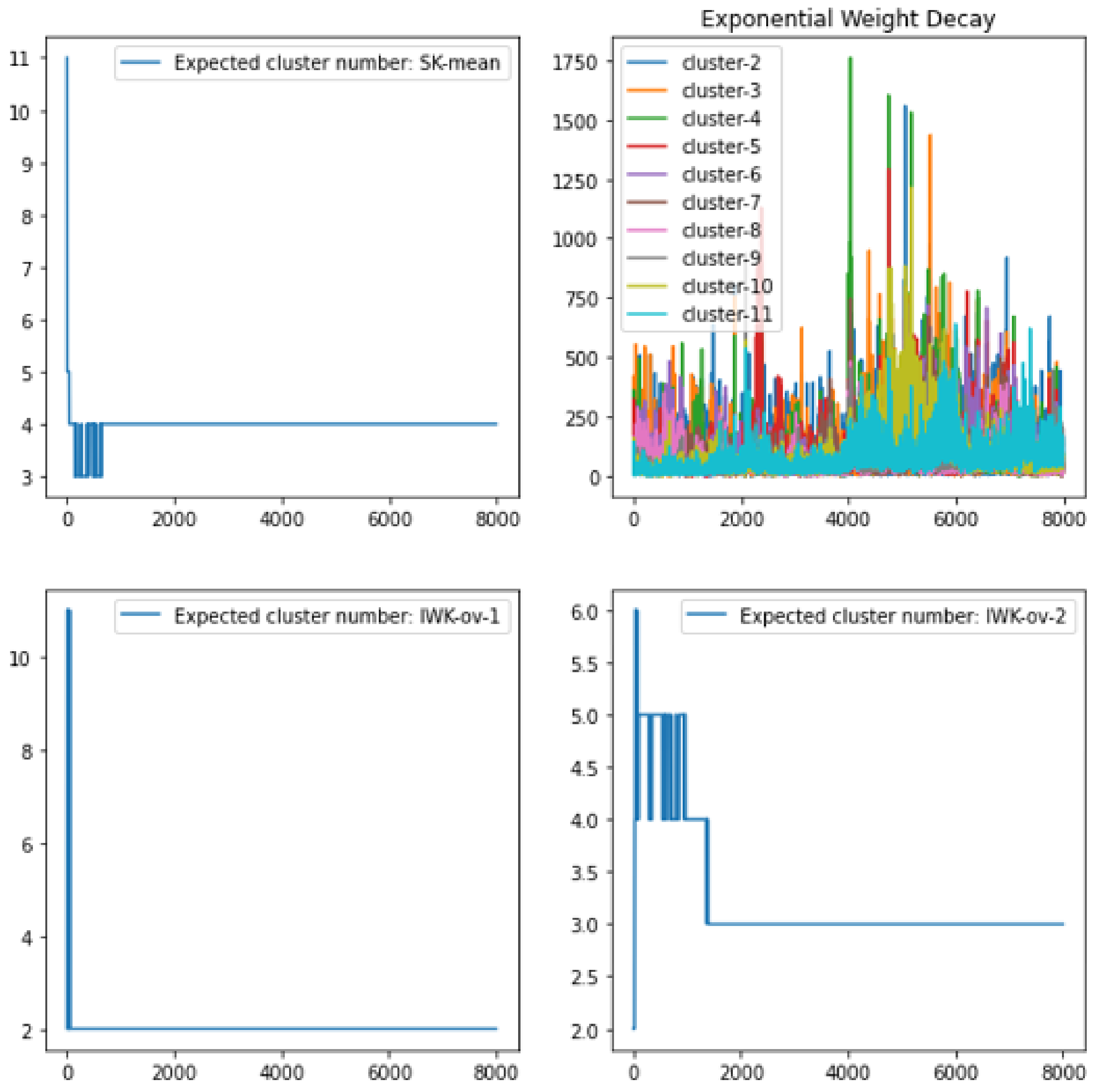

Fig. 5: S8: I* Index

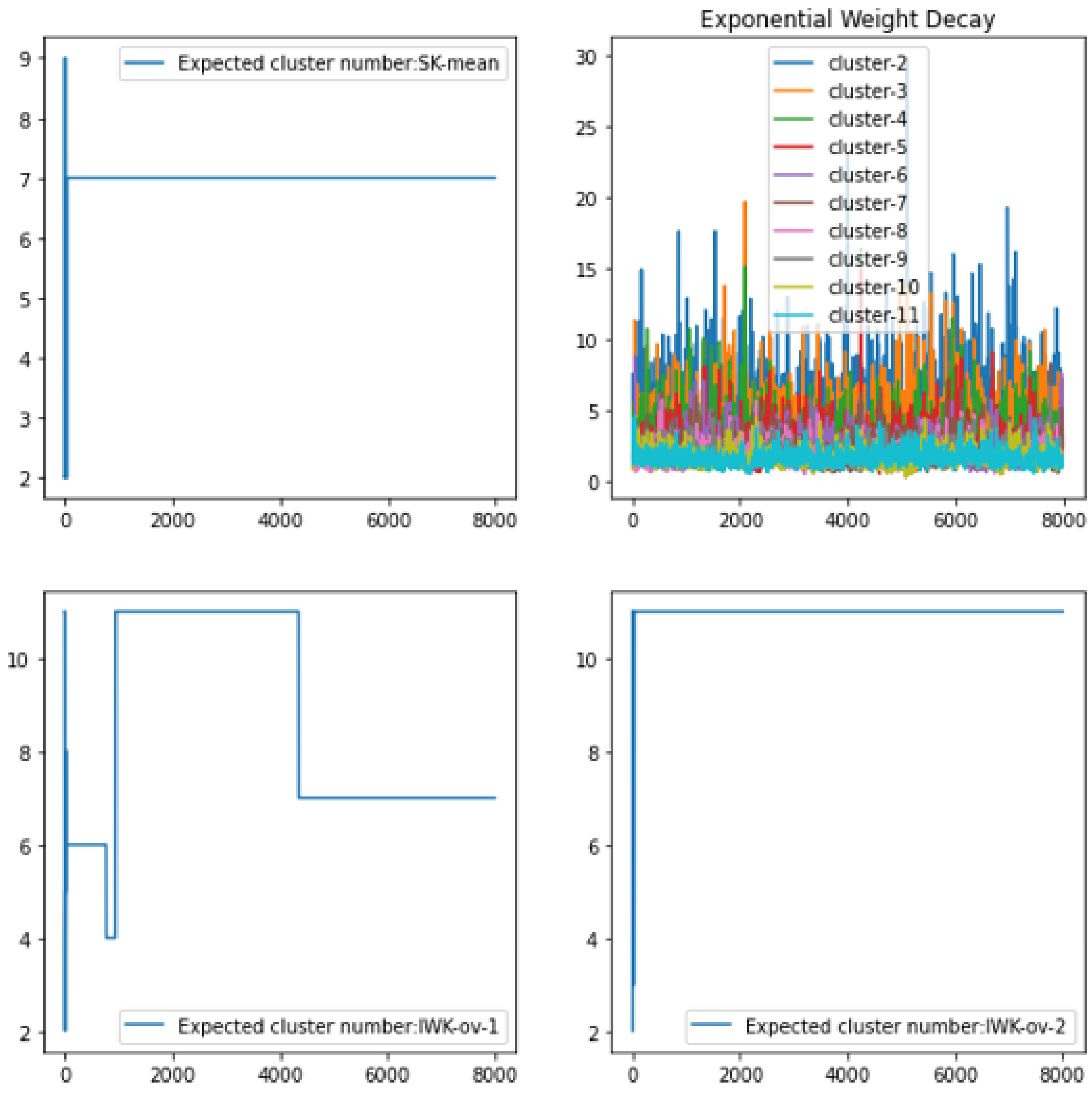

Fig. 6: S8: XB Index

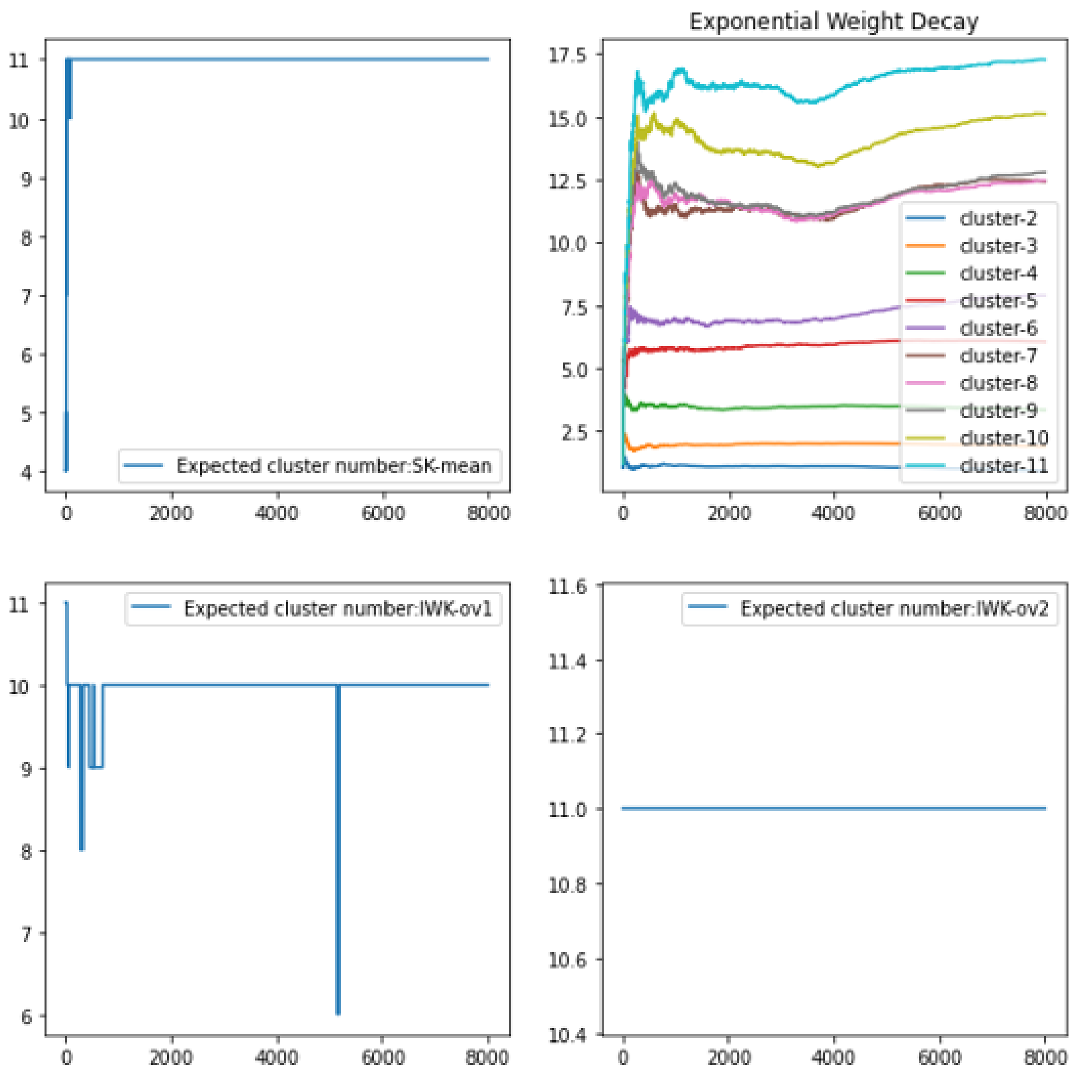

Fig. 7: S8: DB Index

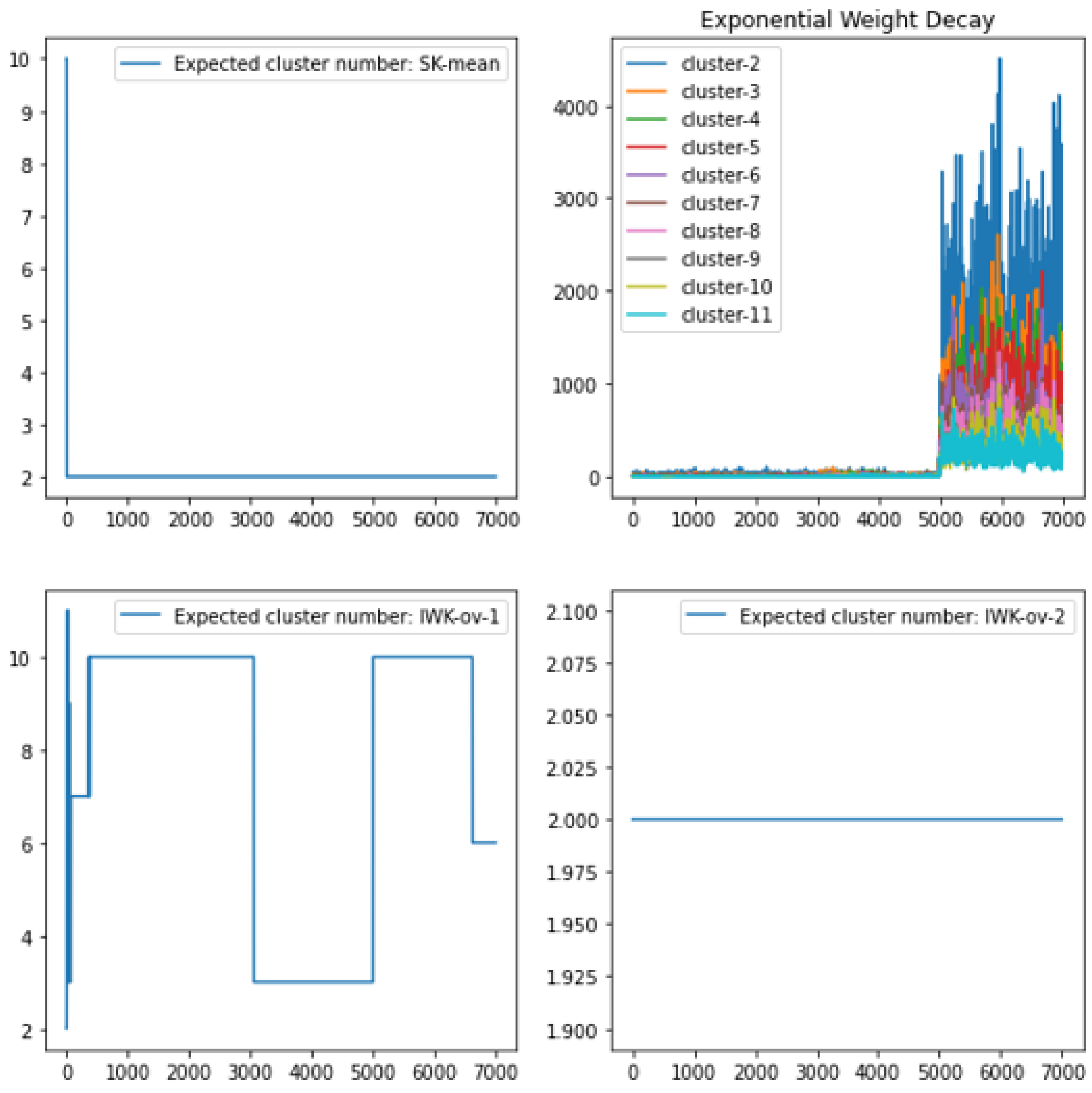

Fig. 8: S9: I* Index

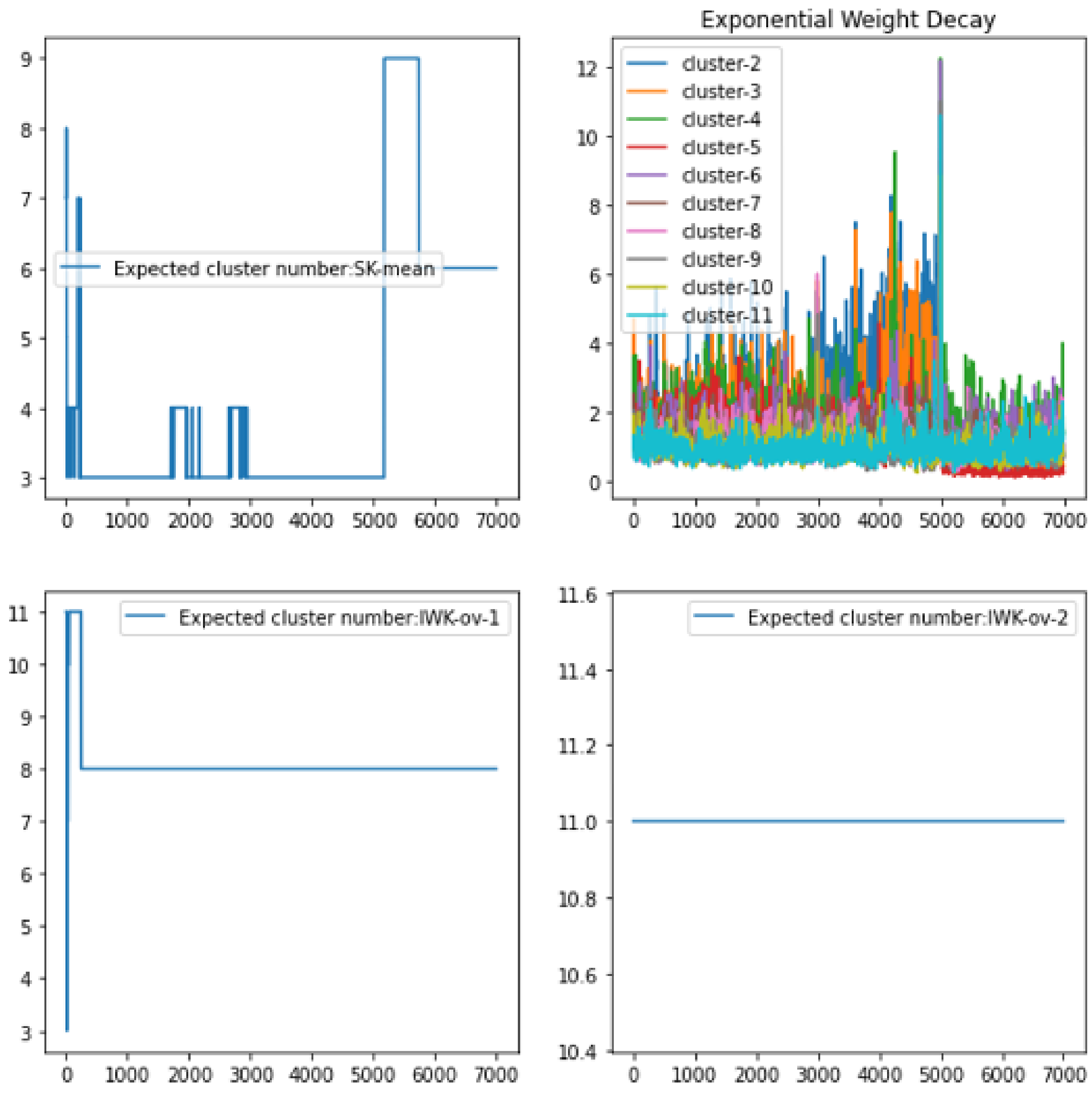

Fig. 9: S9: XB Index

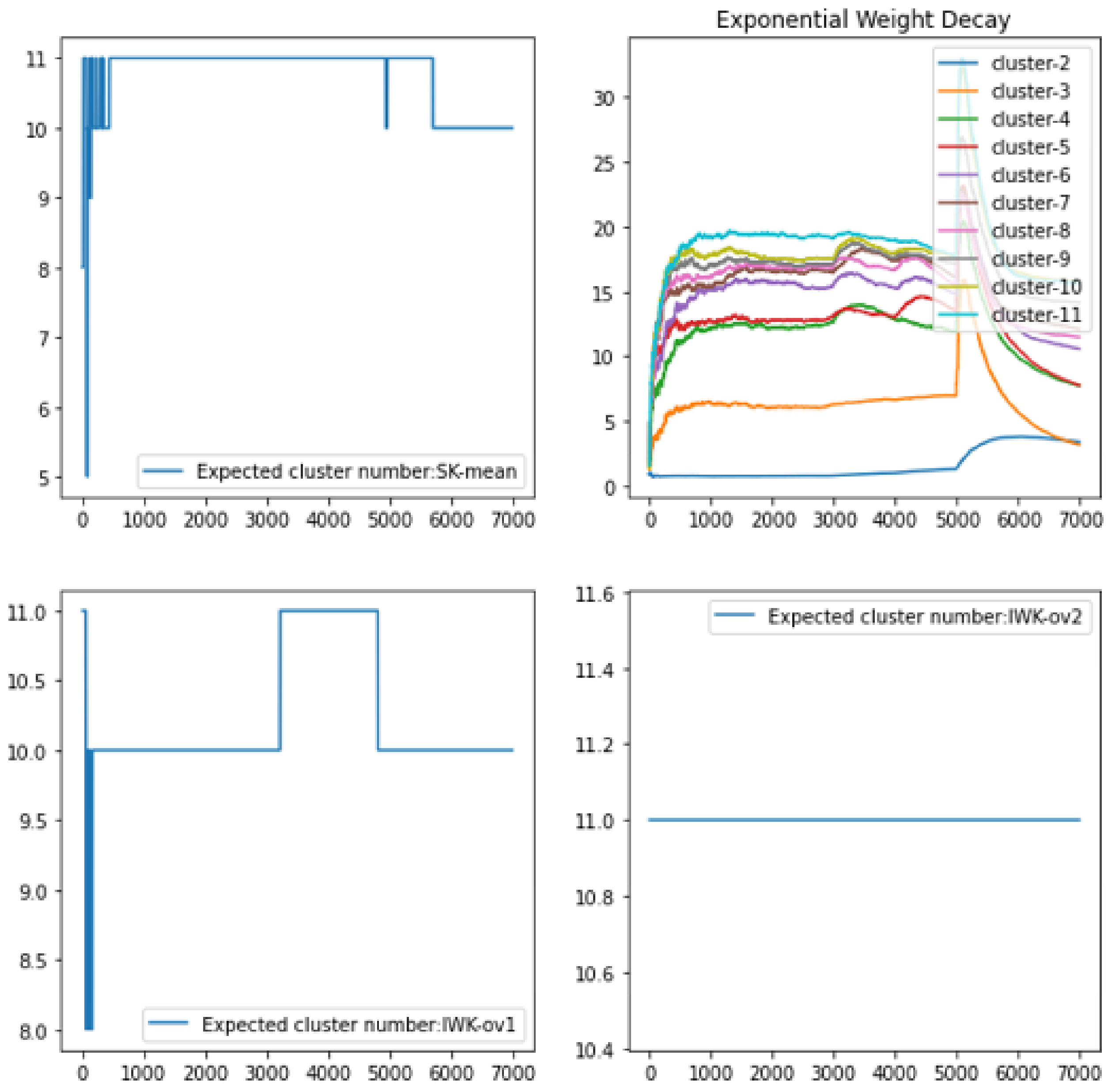

Fig. 10: S9: DB Index

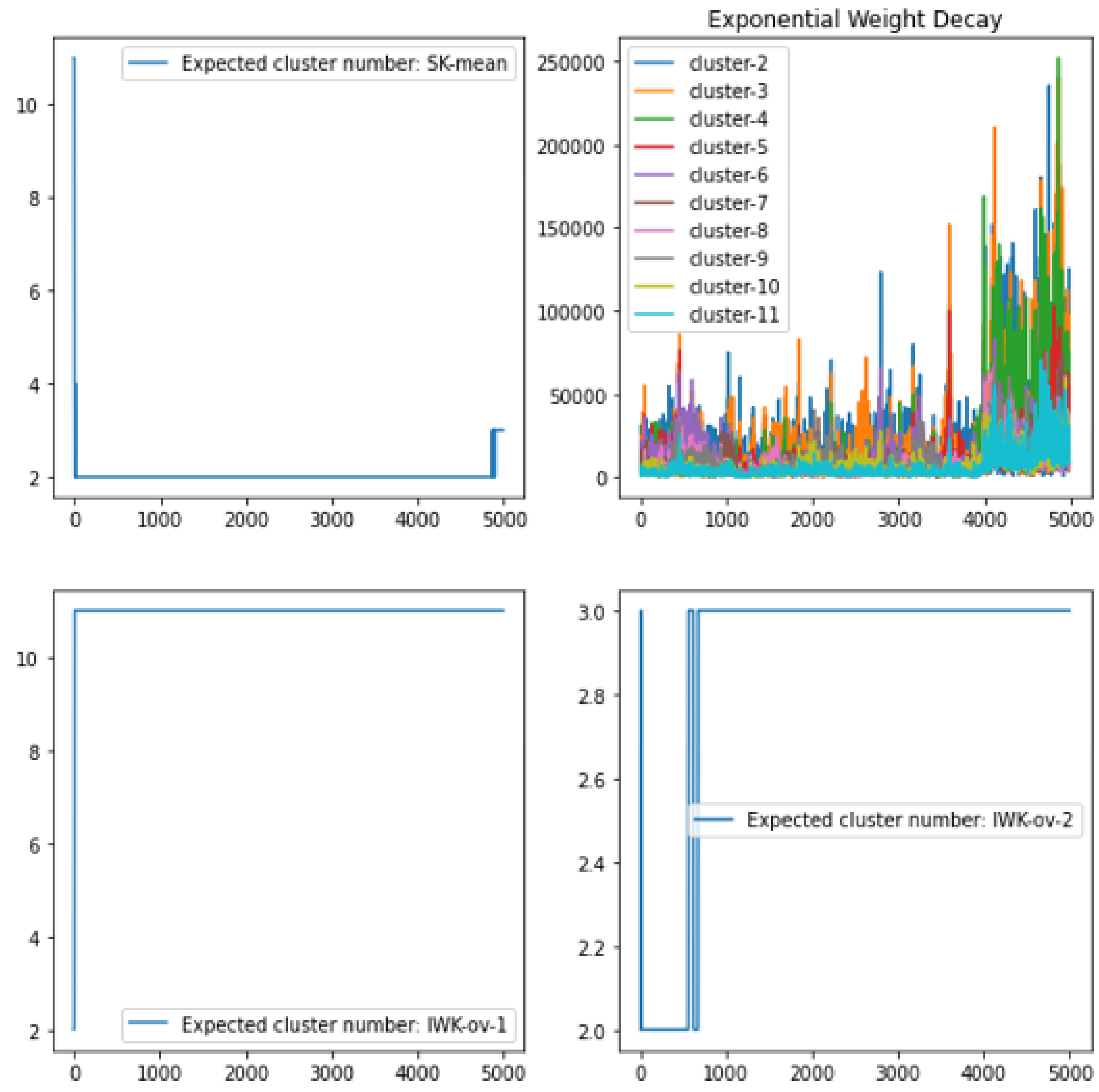

Fig. 11: S10: I* Index

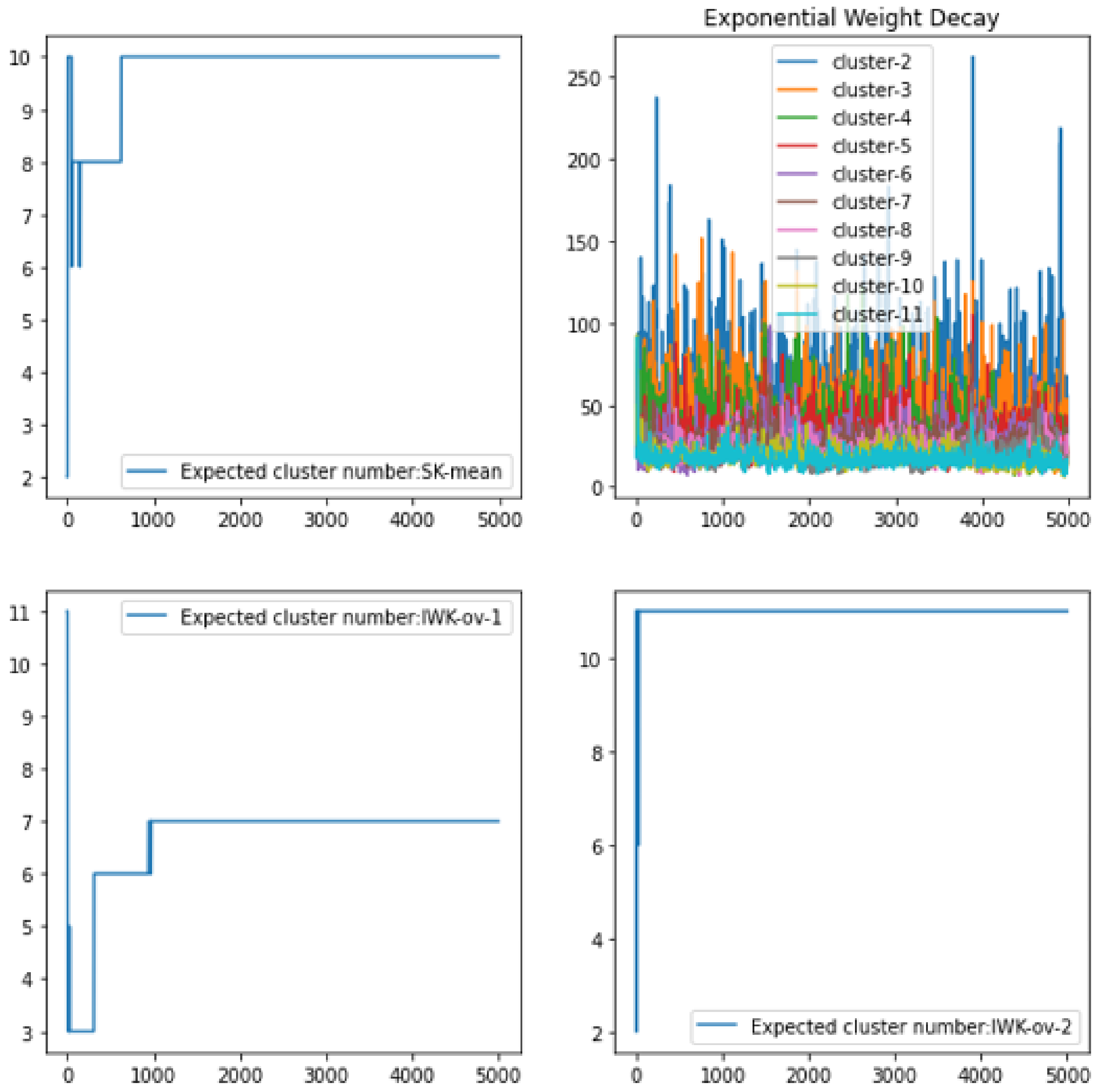

Fig. 12: S10: XB Index

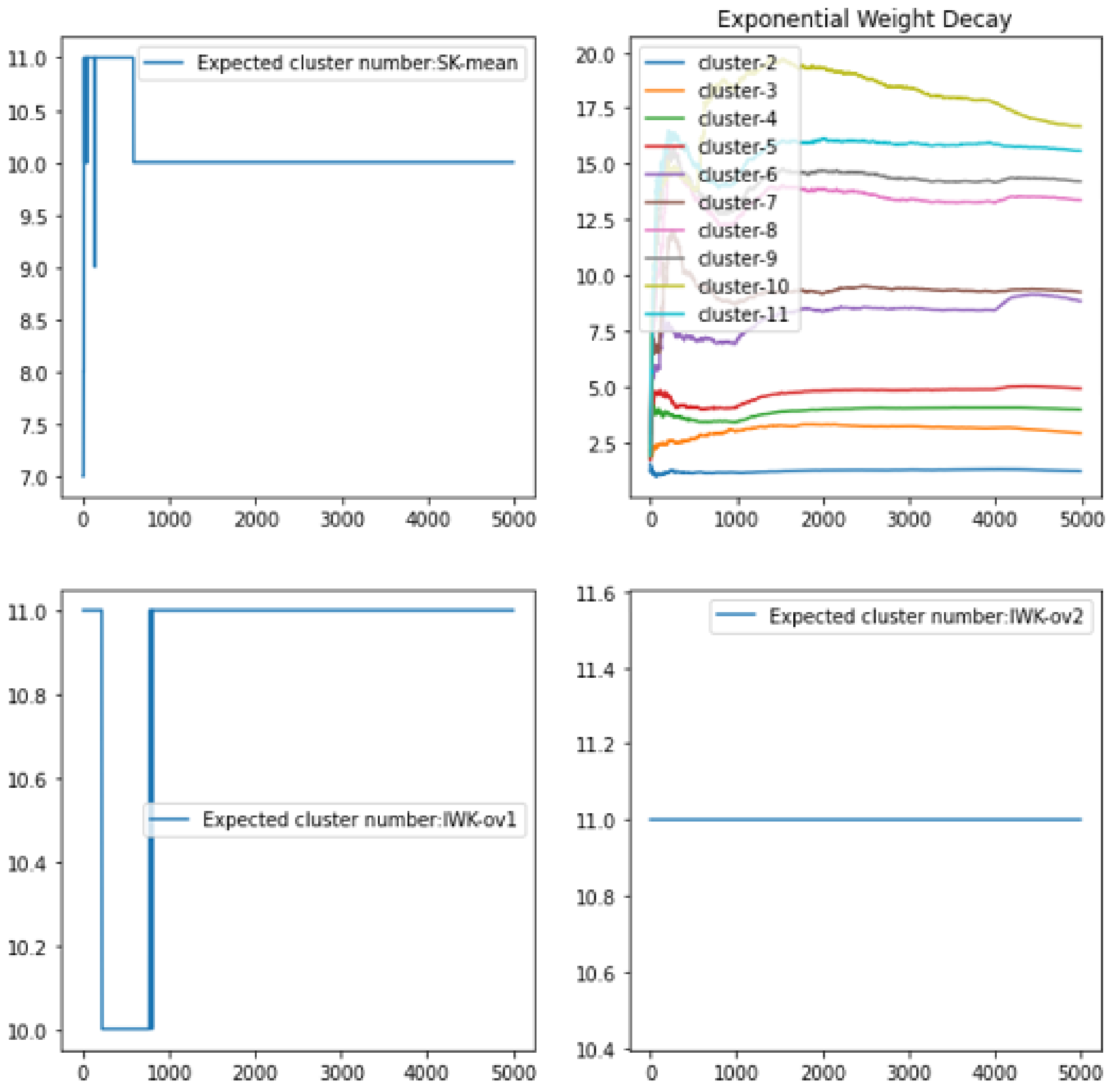

Fig. 13: S10: DB Index

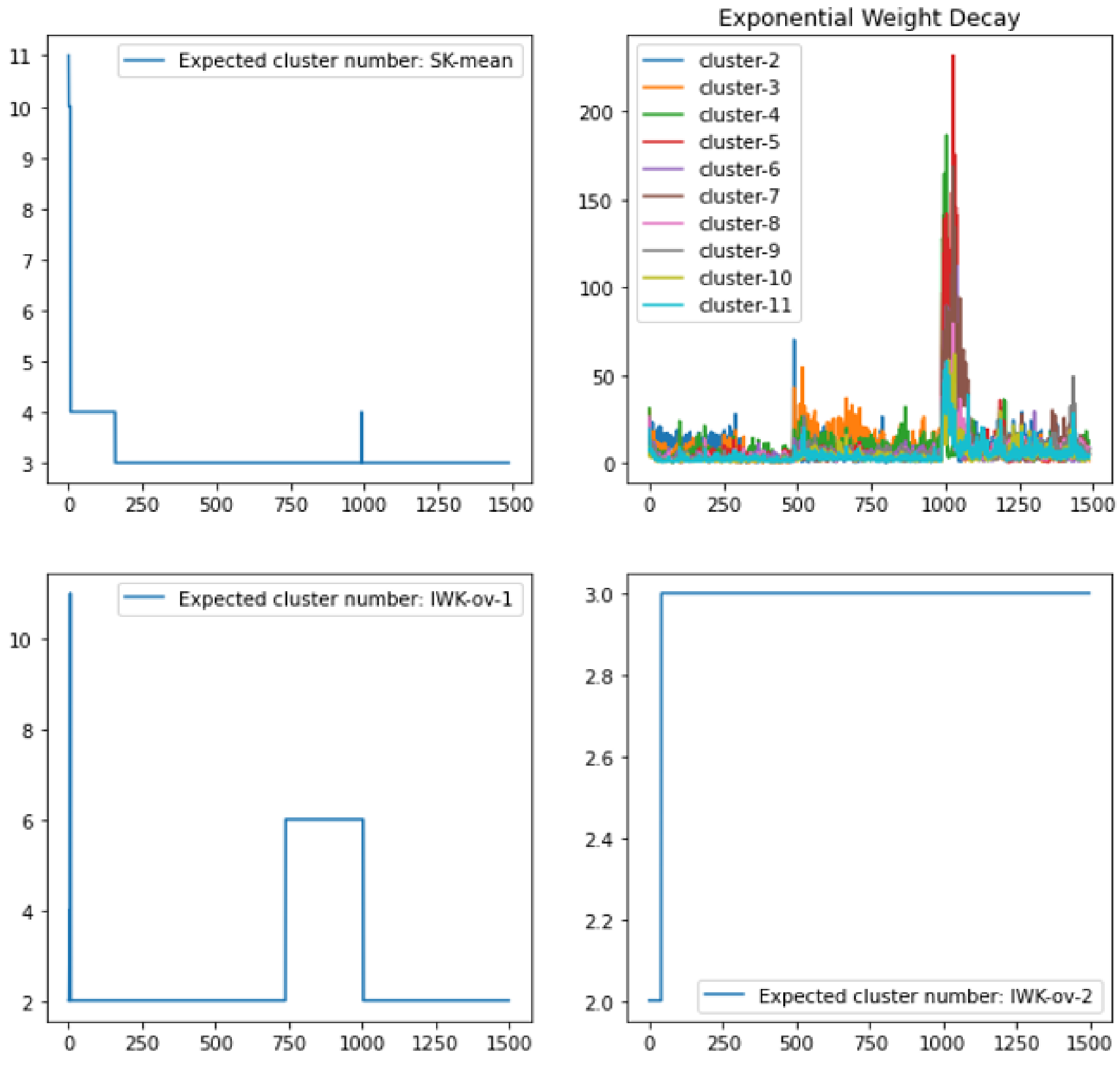

Fig. 14: S11: I* Index

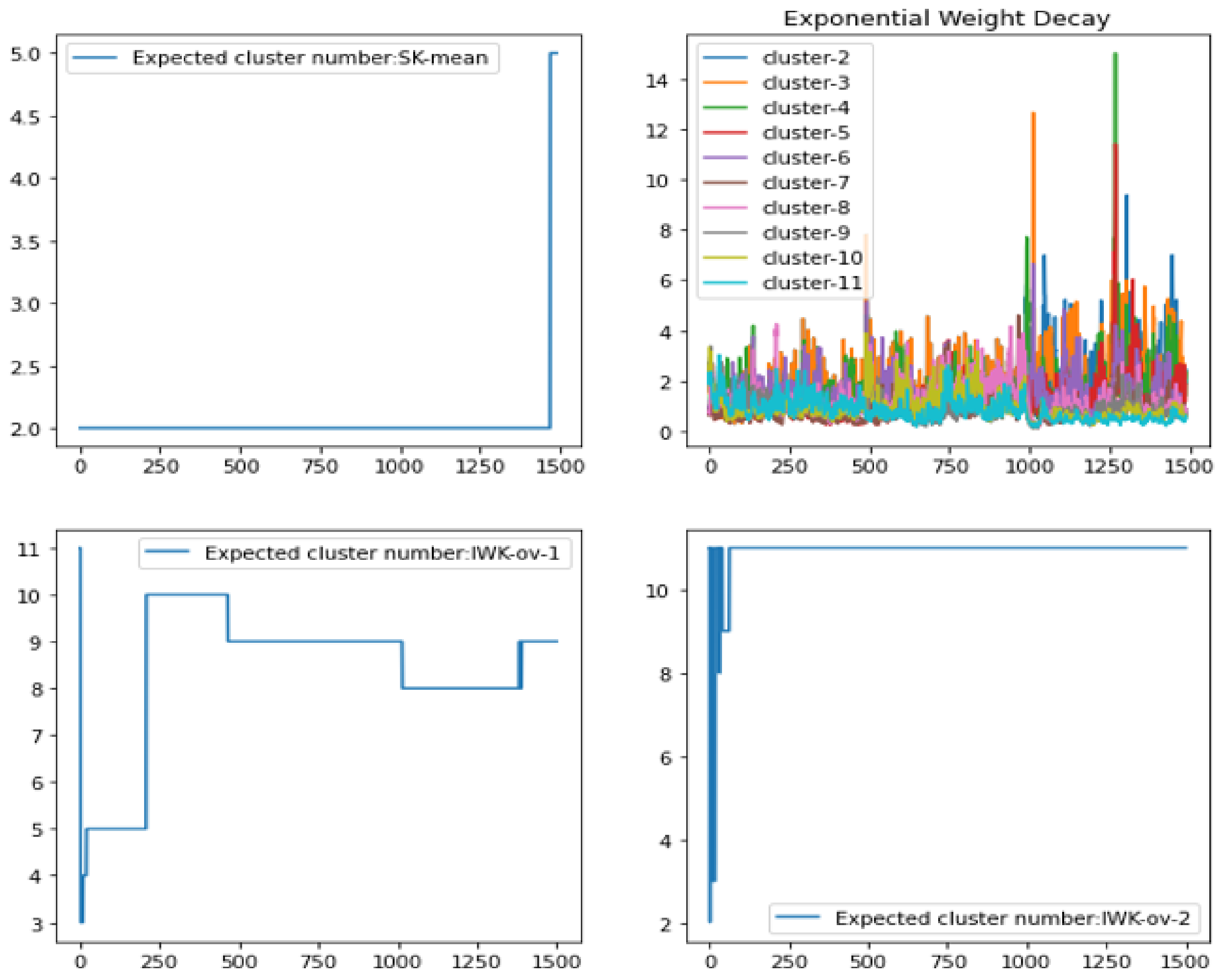

Fig. 15: S11: XB Index

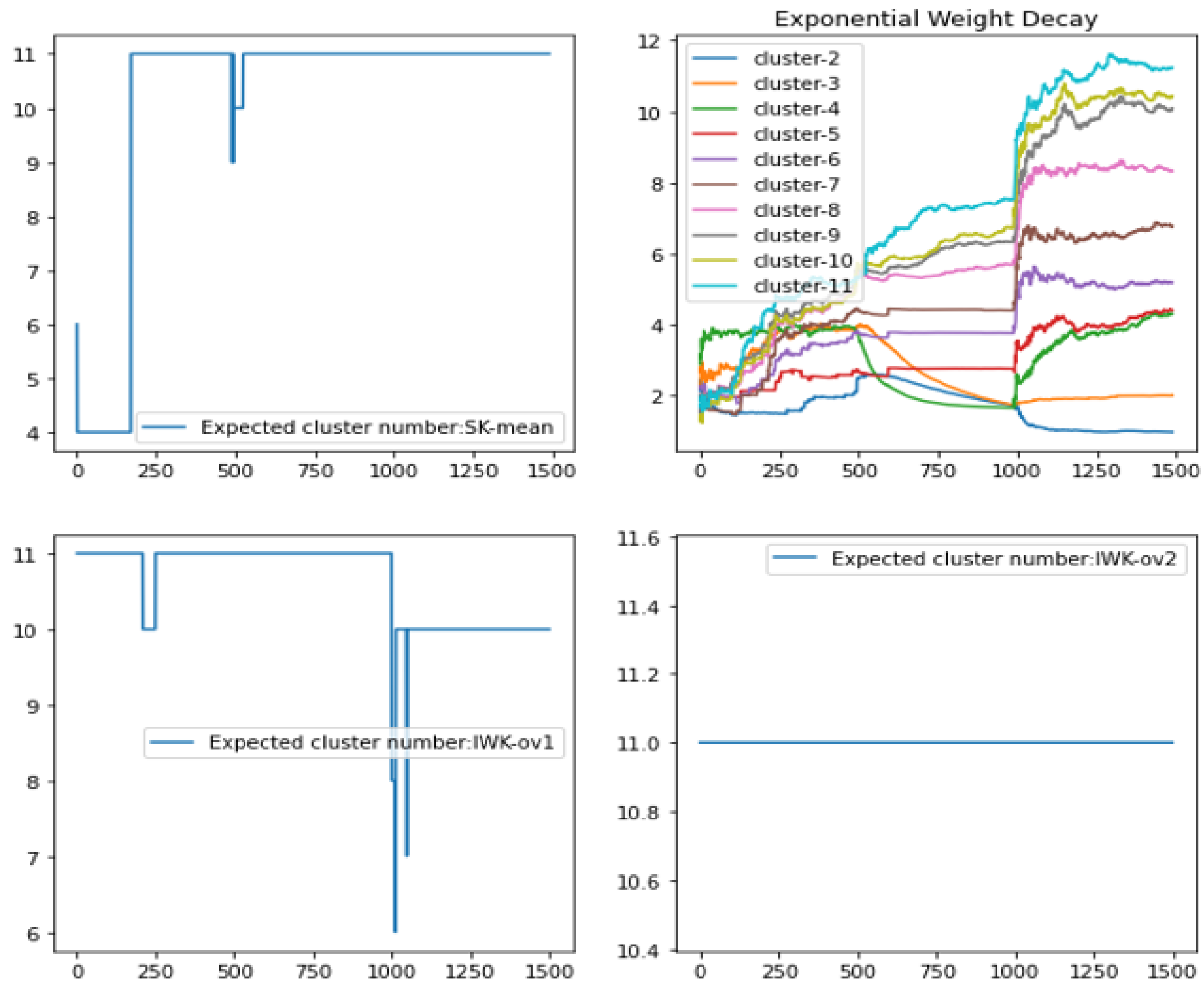

Fig. 16: S11: DB Index

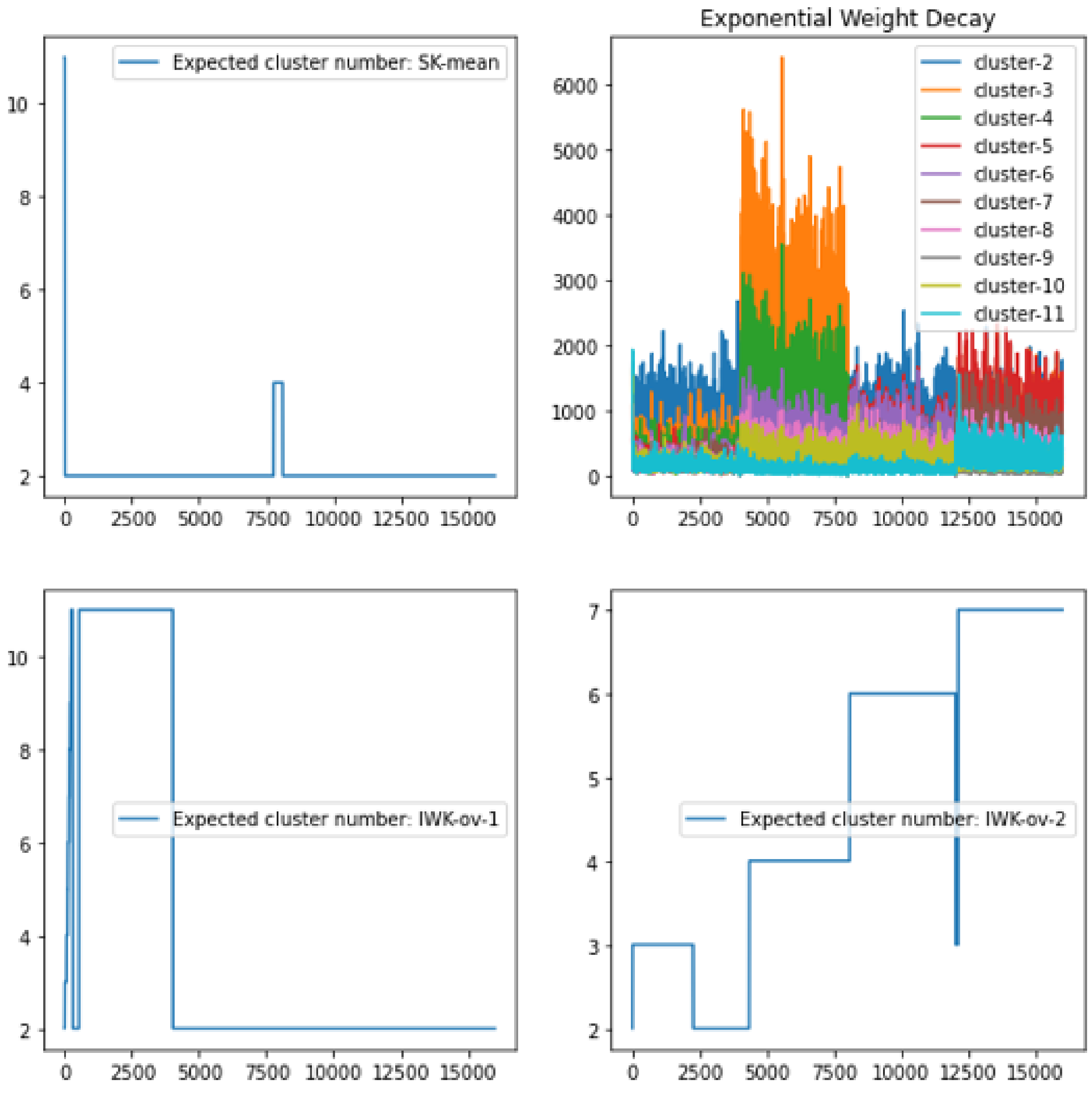

Fig. 17: S12: I* Index

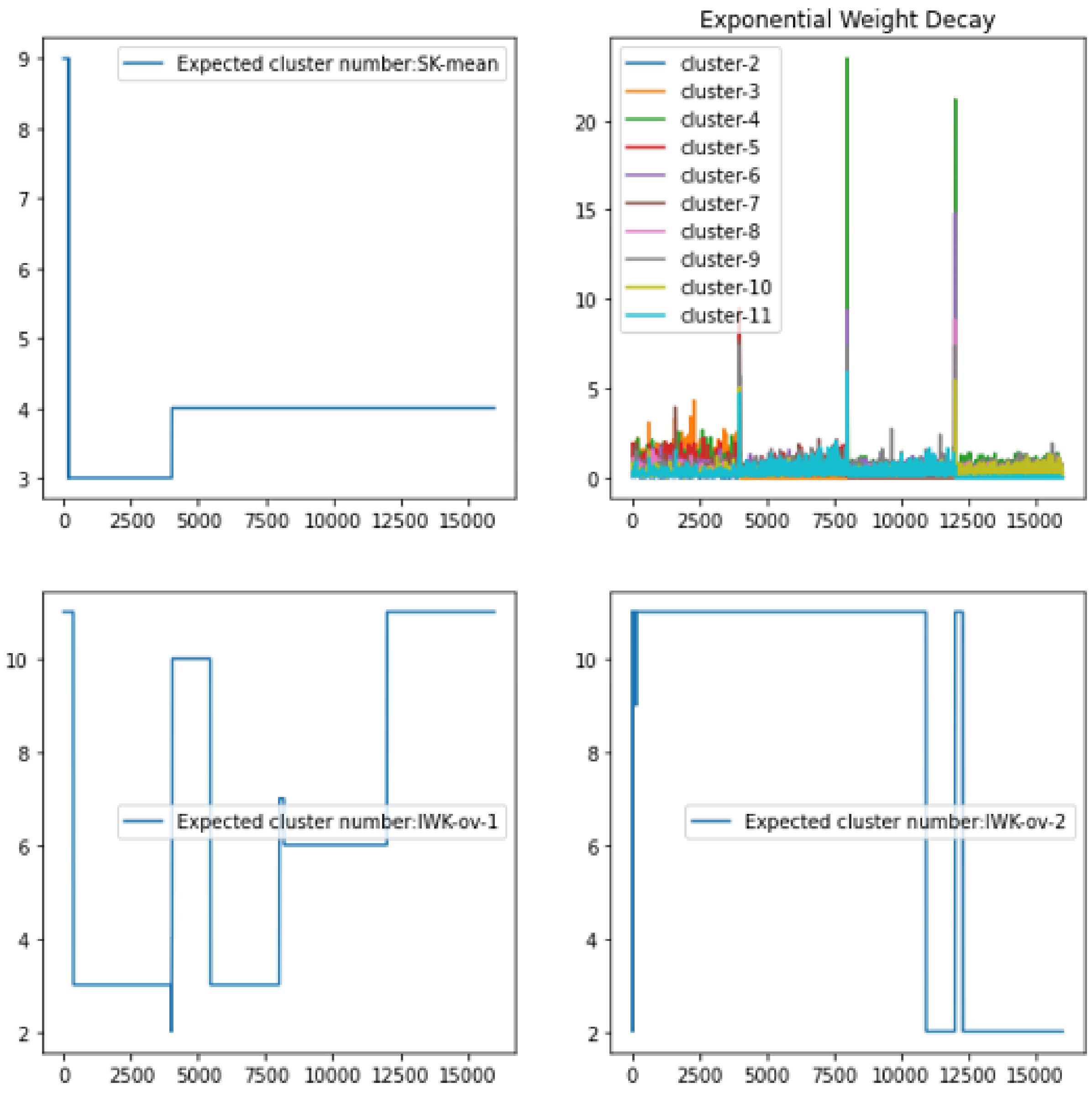

Fig. 18: S12: XB Index

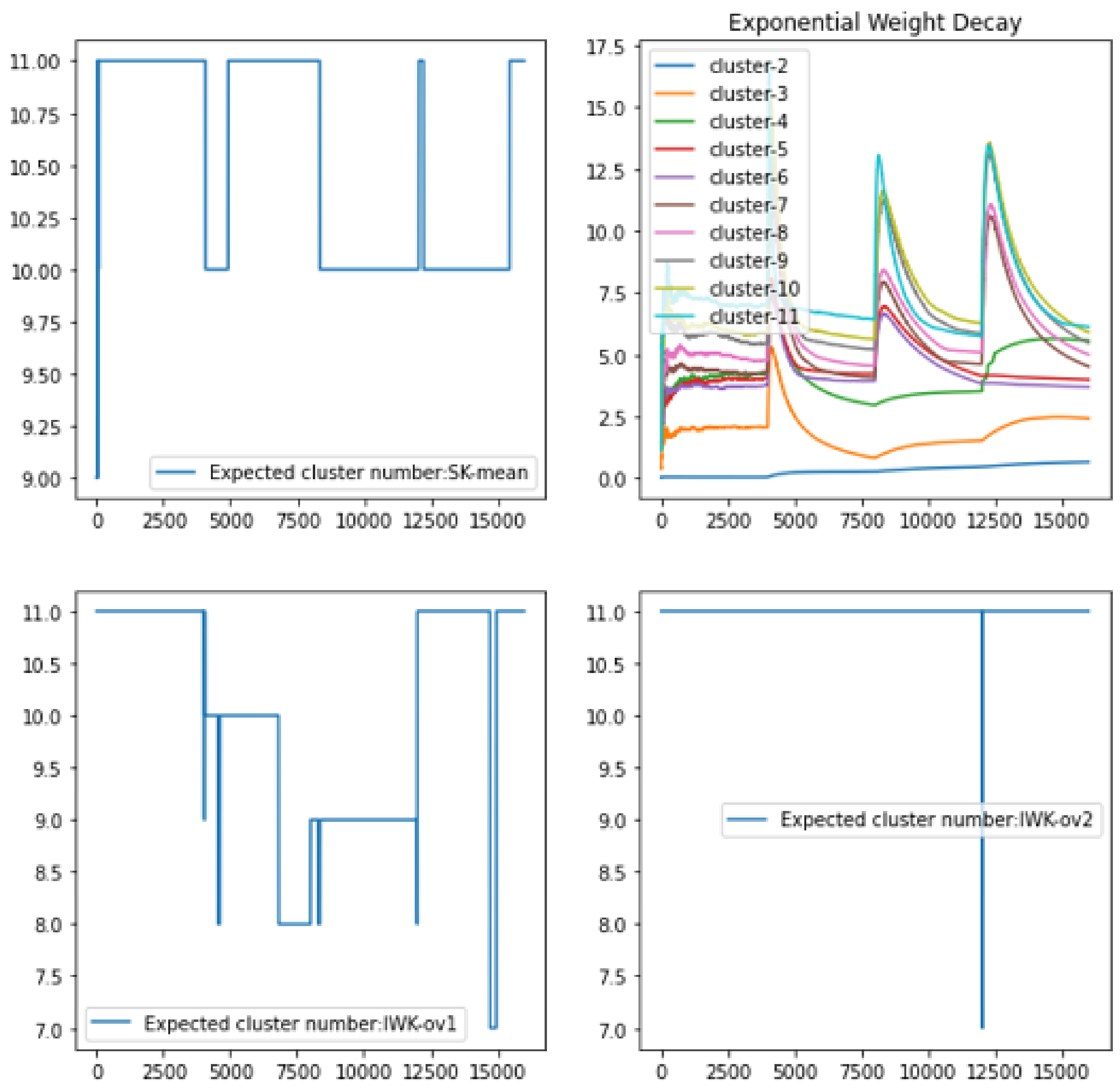

Fig. 19: S12: DB Index

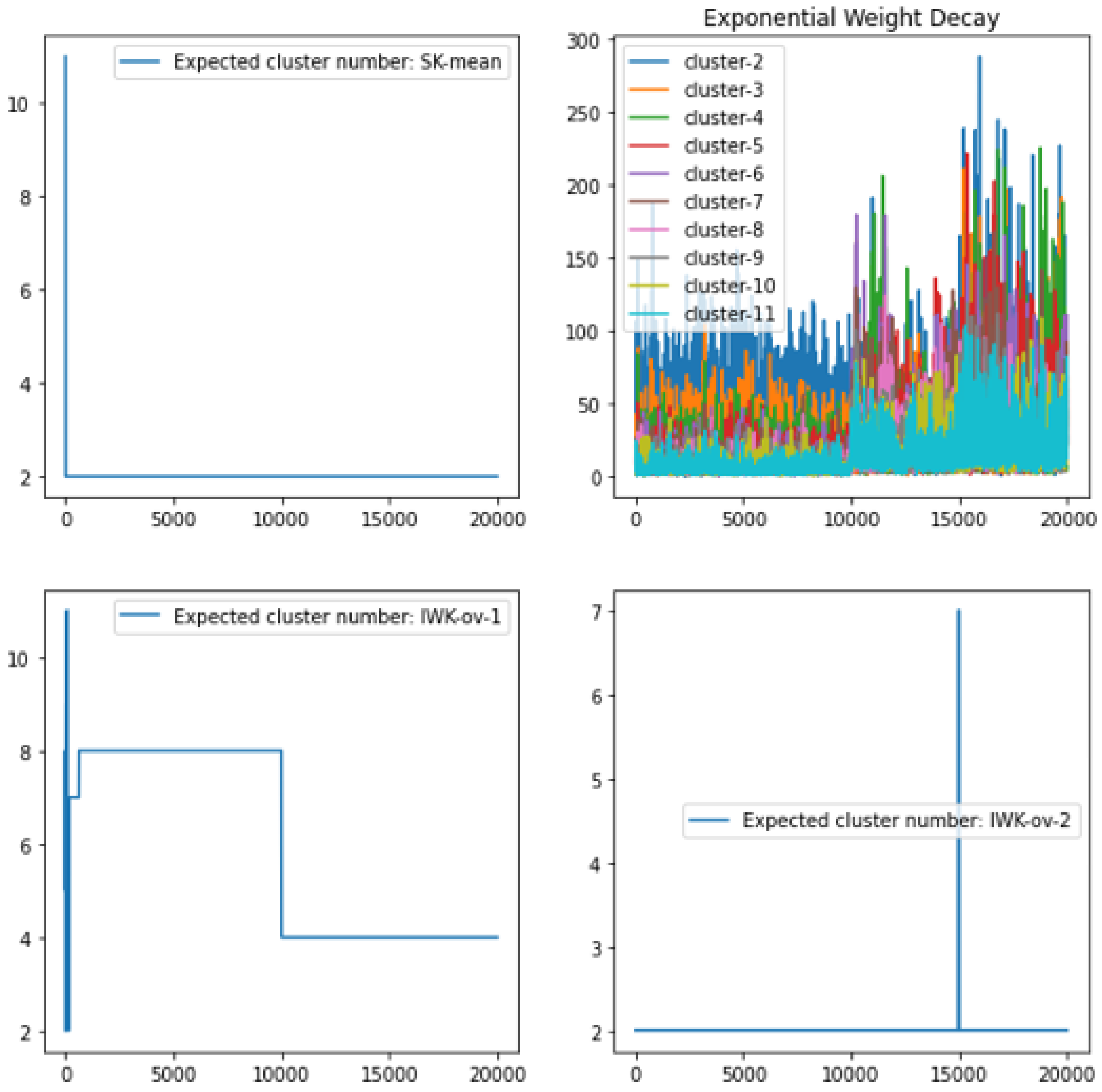

Fig. 20: S13: I* Index

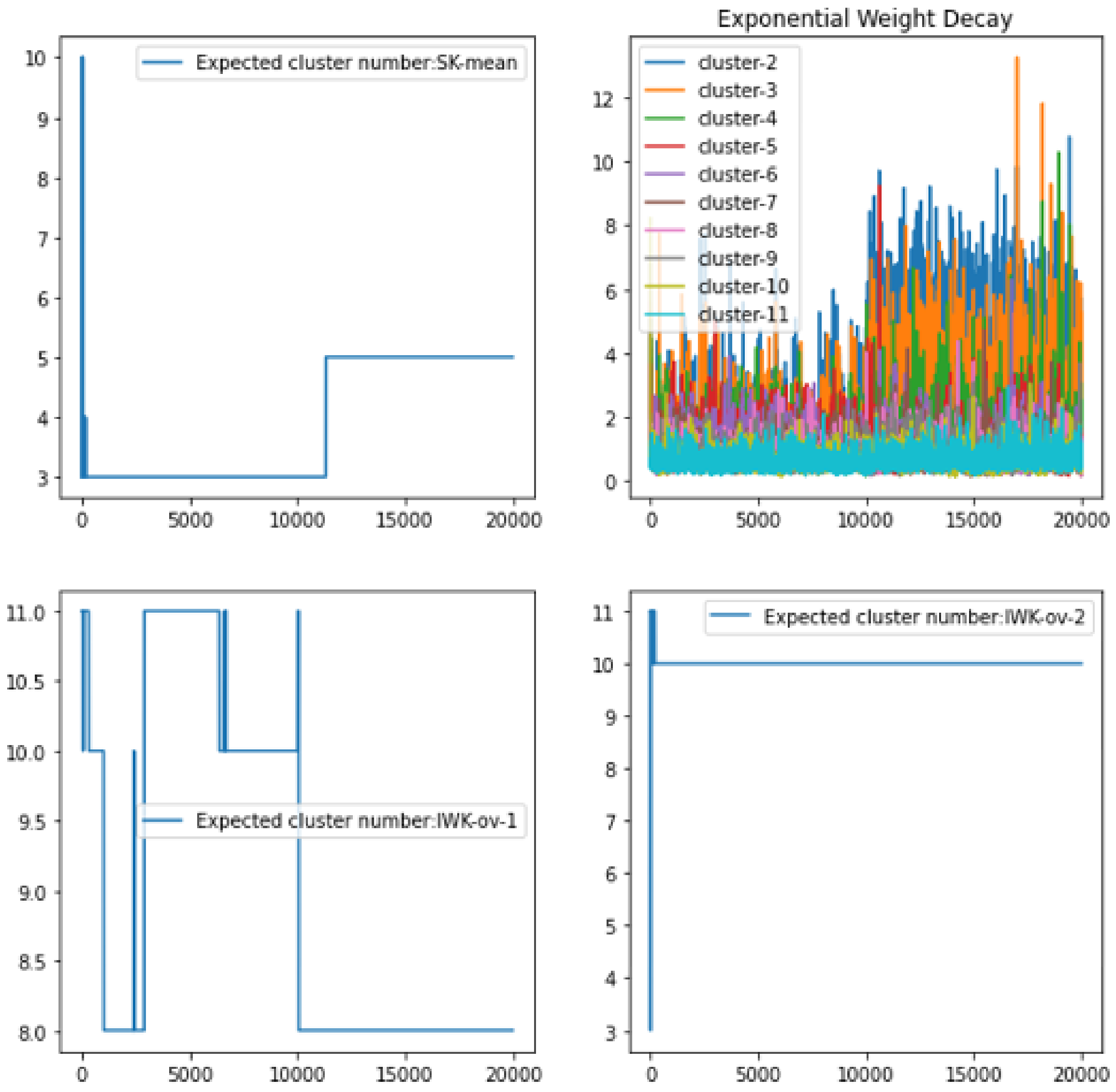

Fig. 21: S13: XB Index

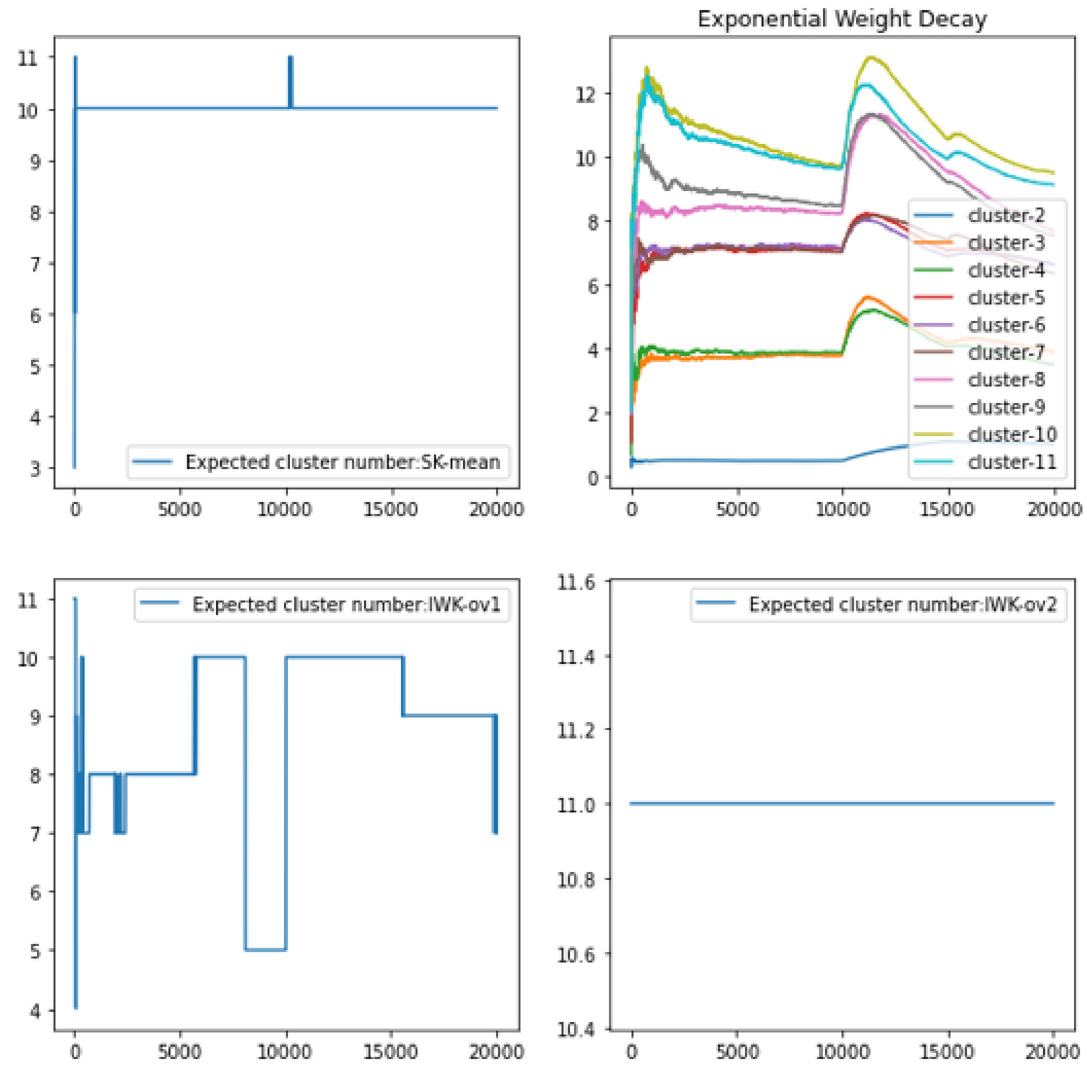

Fig. 22: S13: DB Index

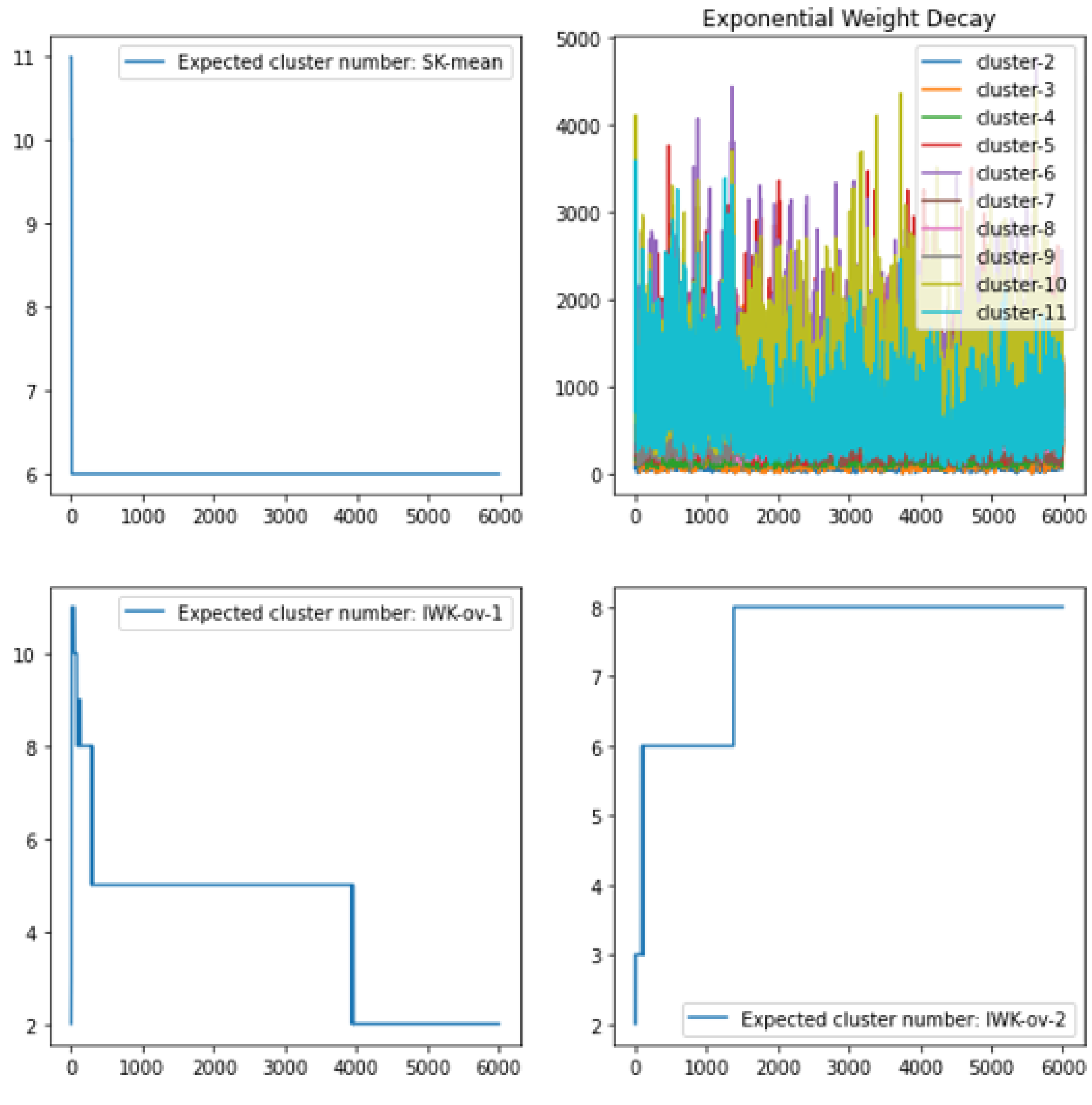

Fig. 23: S14: I* Index

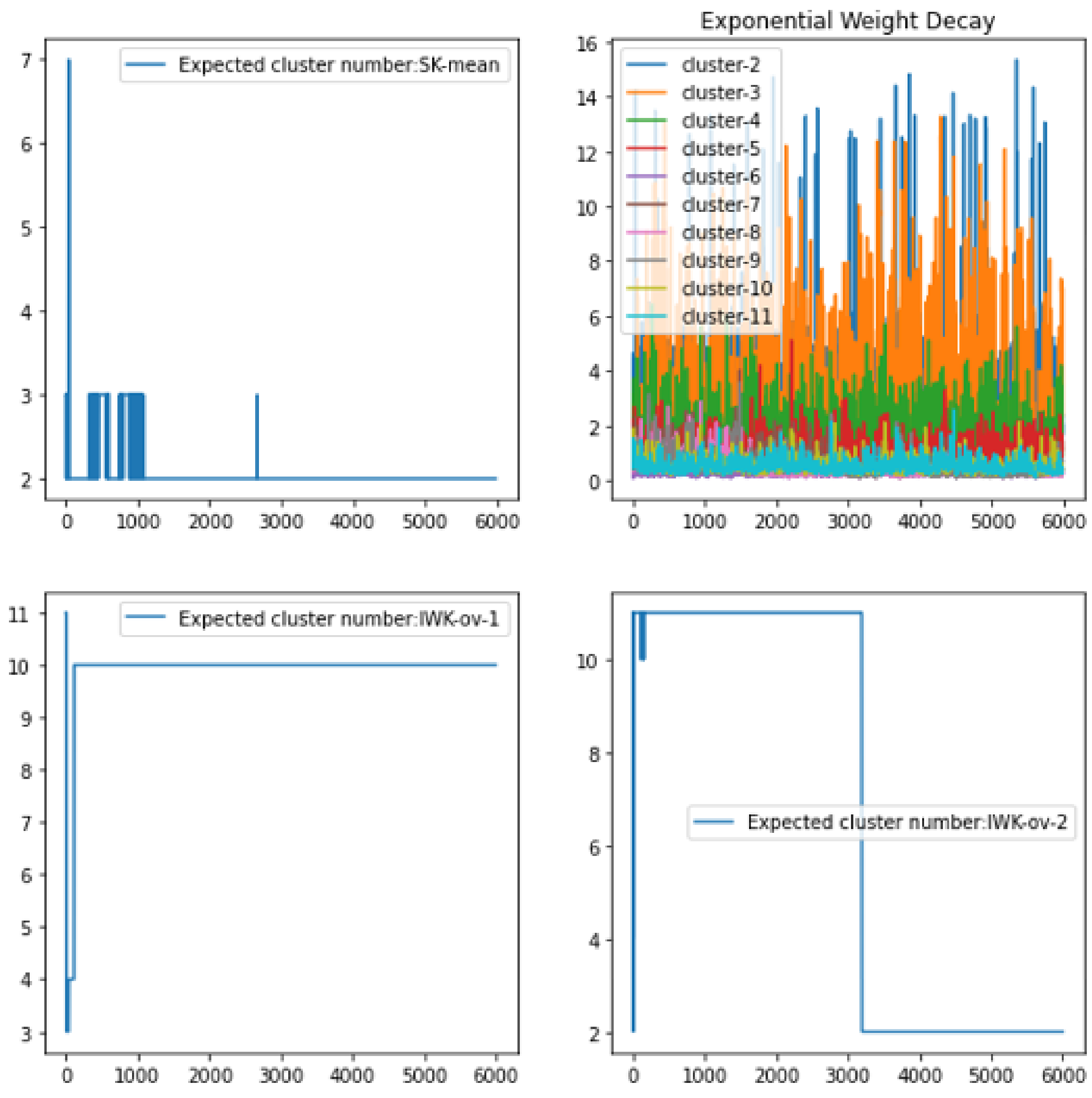

Fig. 24: S14: XB Index

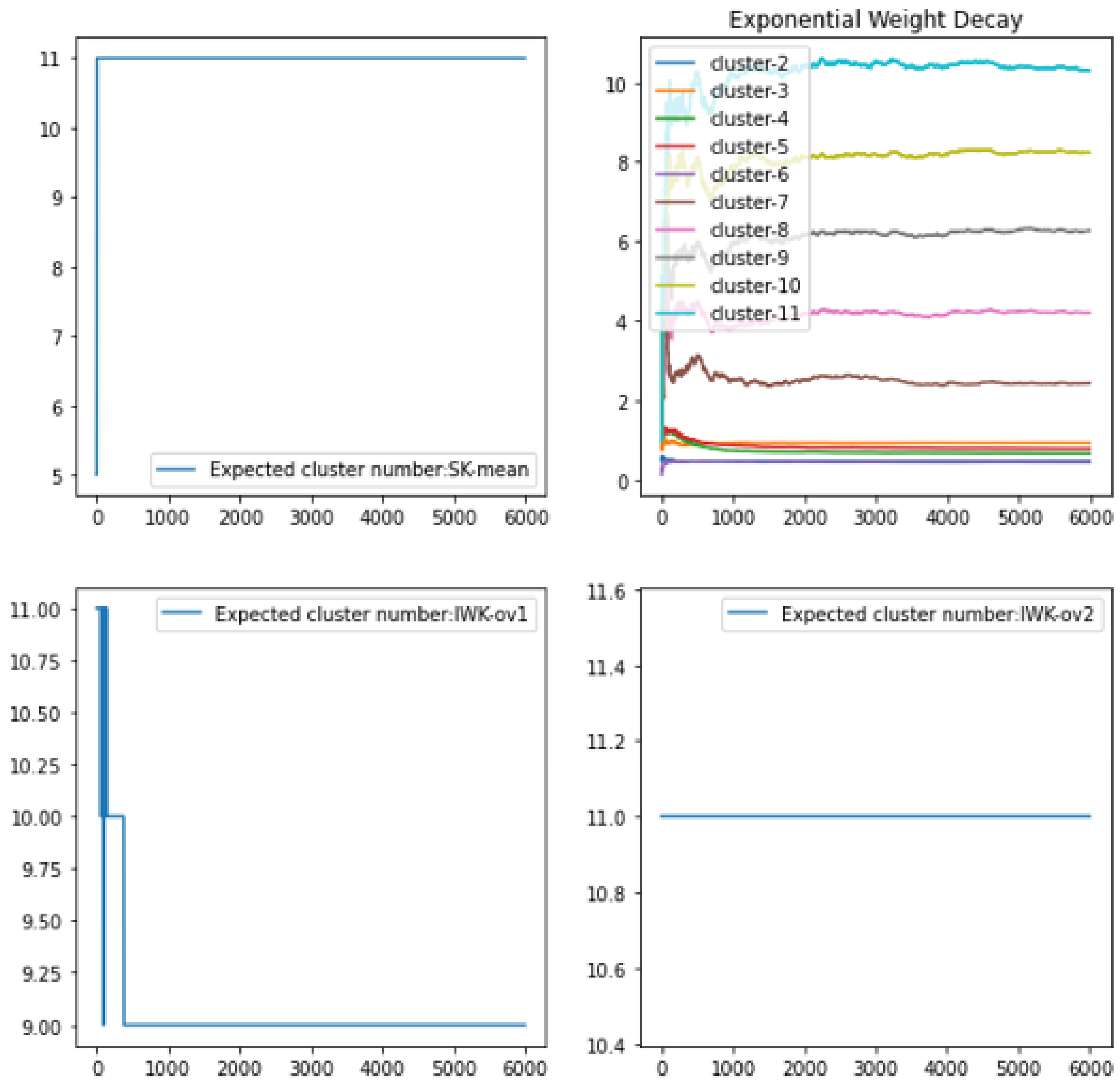

Fig. 25: S14: DB Index

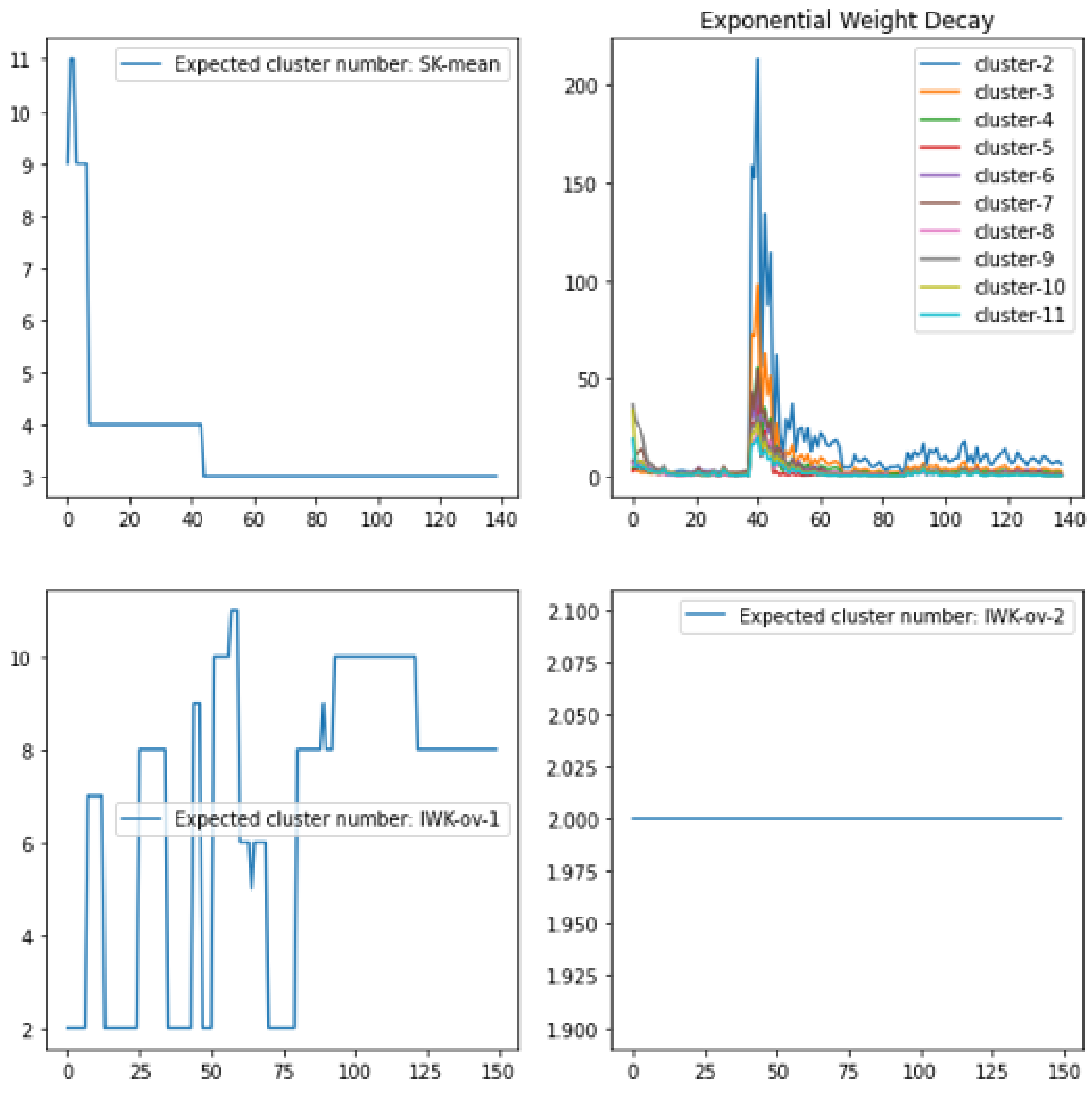

Fig. 26: Iris: I* Index

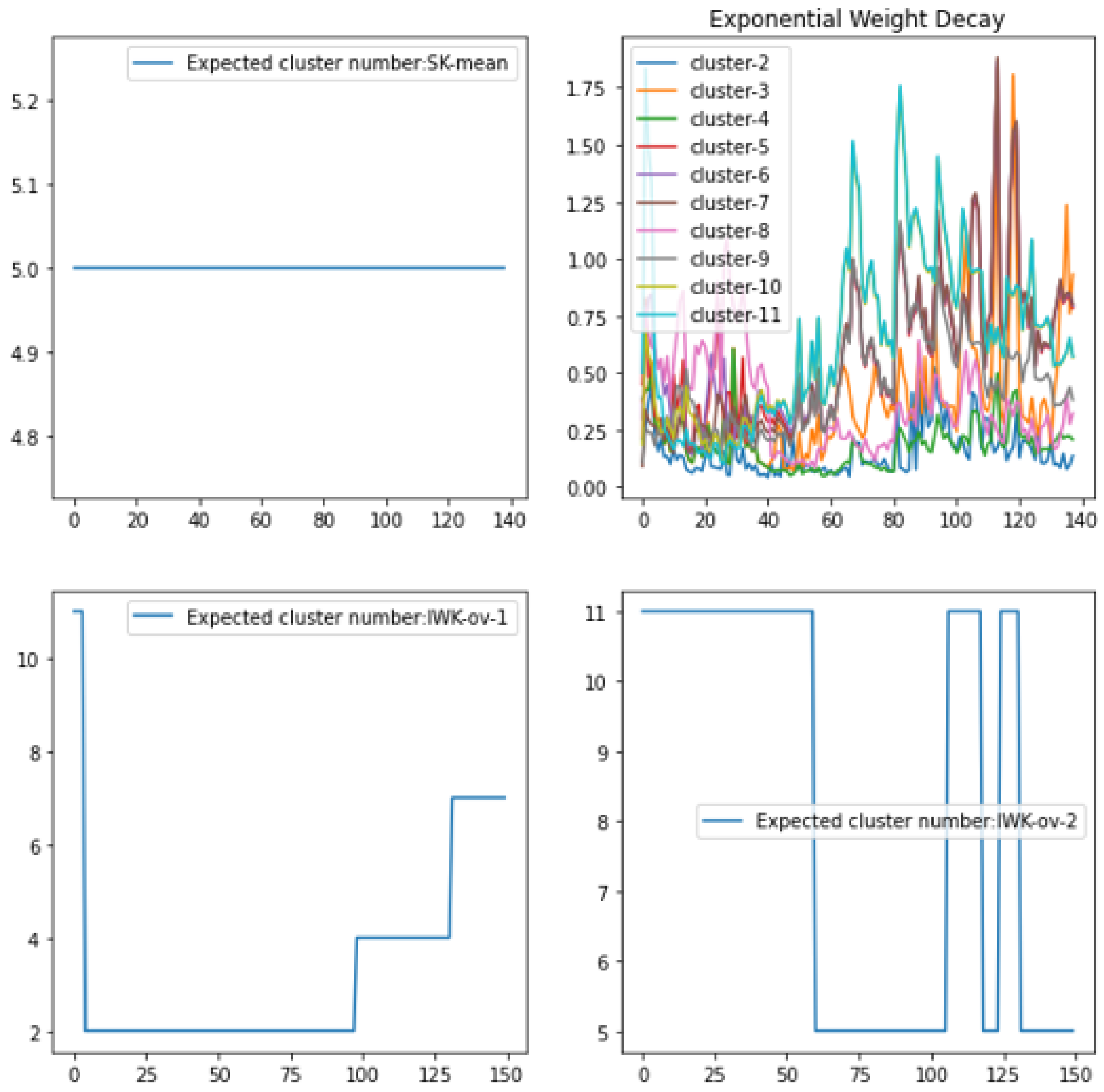

Fig. 27: Iris: XB Index

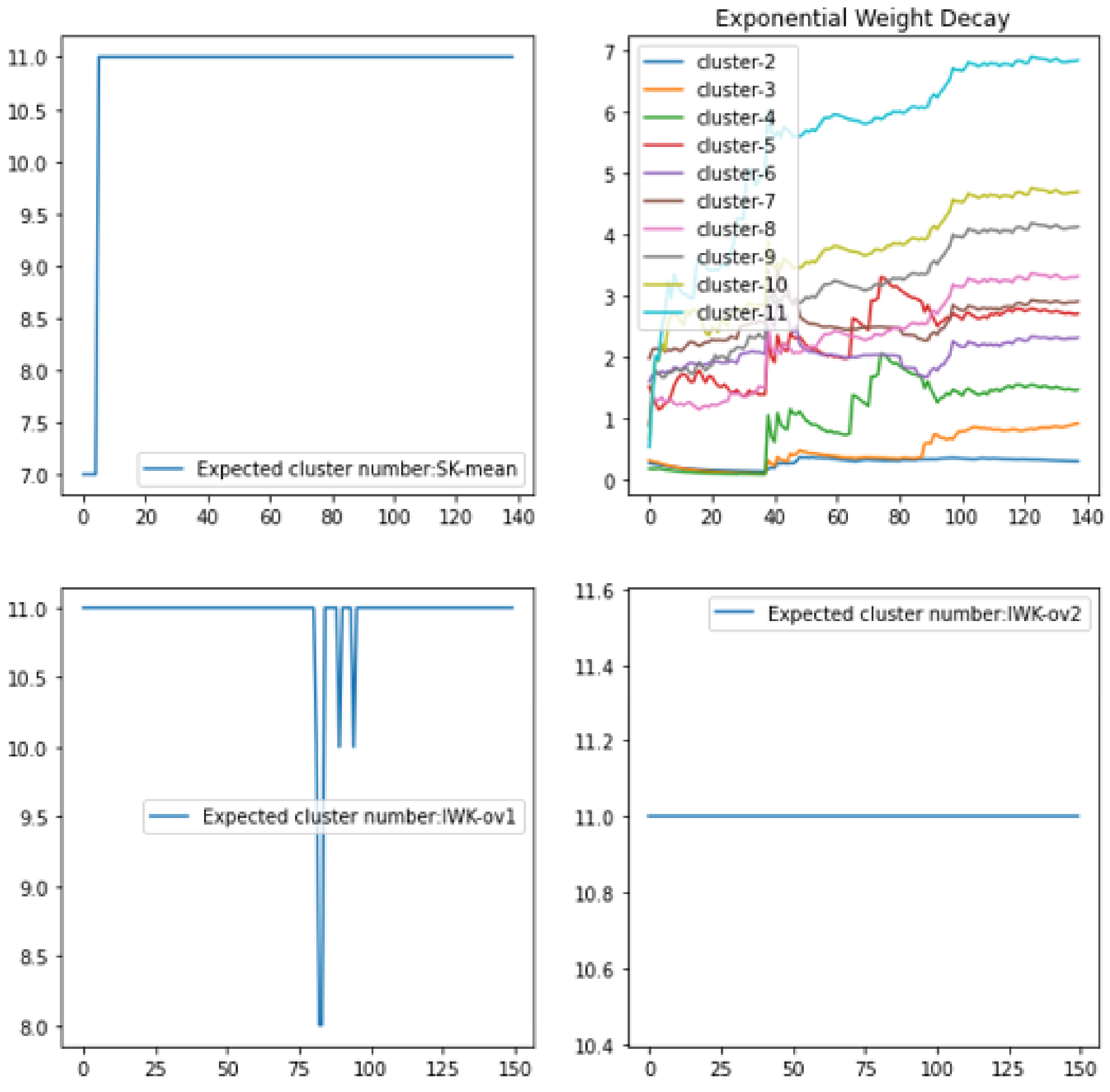

Fig. 28: Iris: DB Index

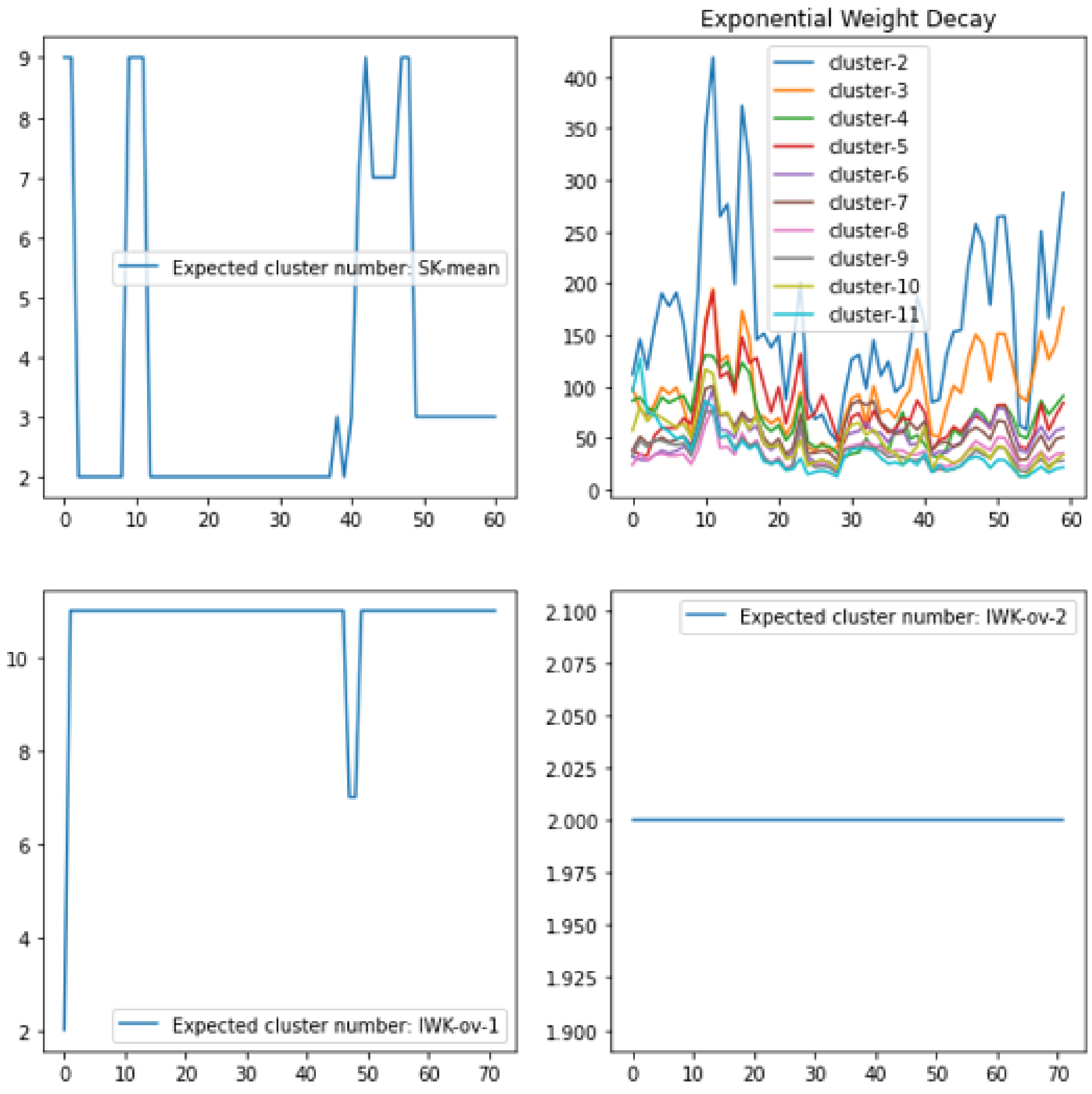

Fig. 29: Cervical Cancer: I* Index

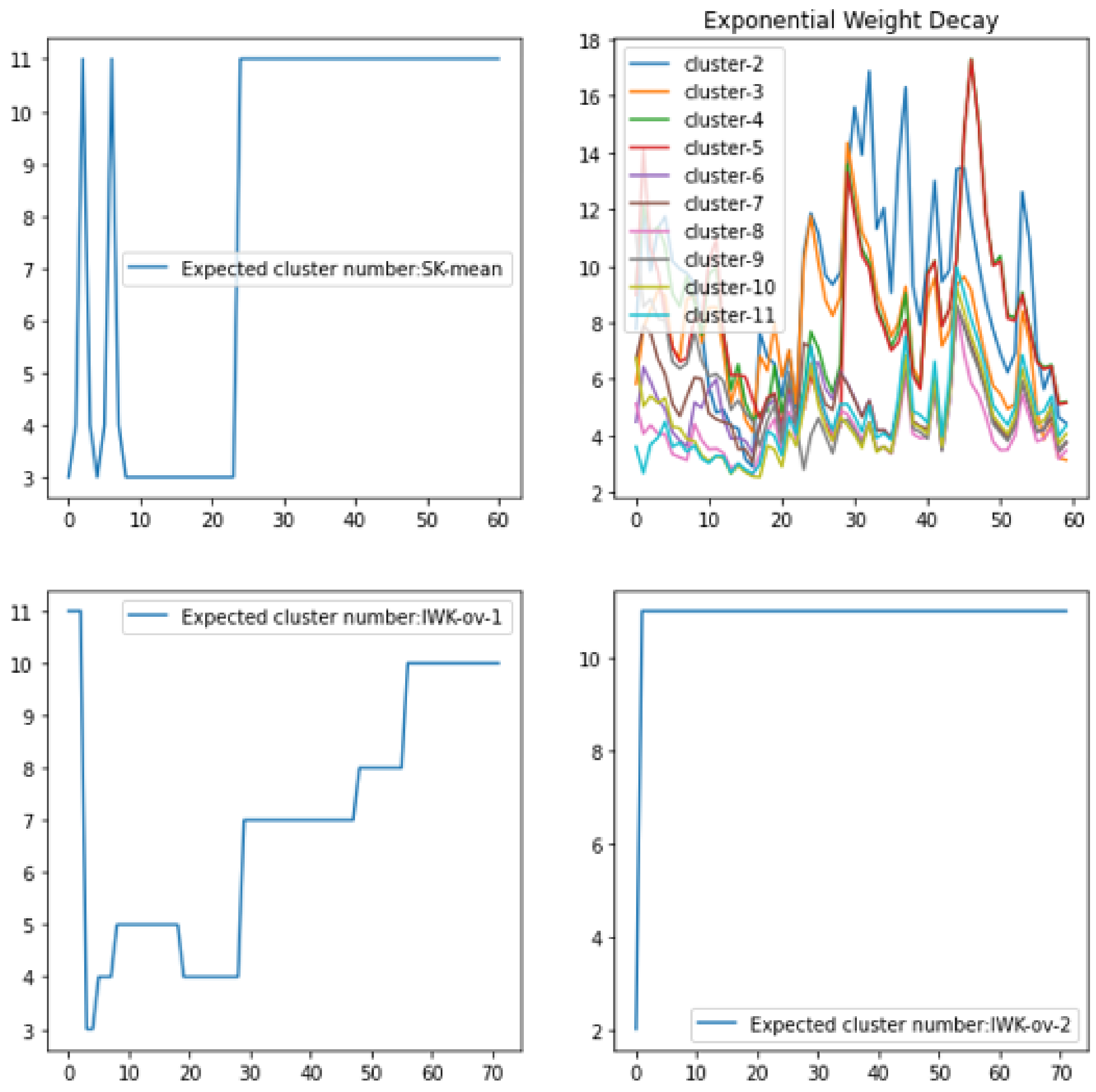

Fig. 30: Cervical Cancer: XB Index

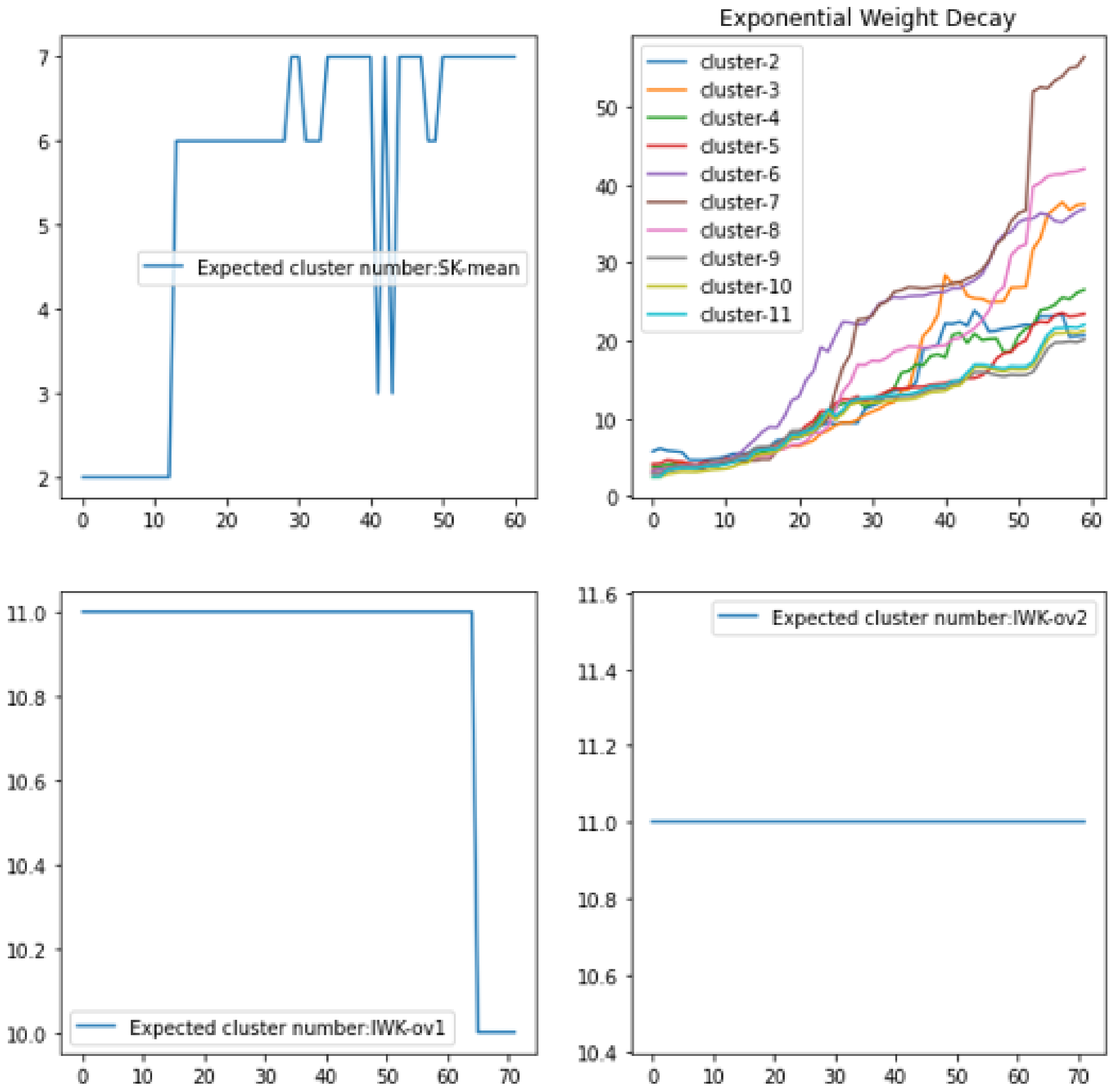

Fig. 31: Cervical Cancer: DB Index