User Manual for KCC—a MATLAB package for K-means-based Consensus Clustering

Hao Lin[1], Hongfu Liu[2], Junjie Wu[3], Hong Li[4], Stephan Günnemann[5]

1. INTRODUCTION

This user manual systematically presents the usage of the MATLAB package[6] on K-means-based Consensus Clustering (KCC) accompanying the following paper:

> Hao Lin, Hongfu Liu, Junjie Wu, Hong Li, and Stephan Günnemann. 2022. Algorithm xxxx: KCC: A MATLAB Package for K-means-based Consensus Clustering. *ACM Trans. Math. Softw.*

For package installation, you need to first unpack the compressed archive into your current directory. It consists of a *source code* folder Matlab, a folder userManual with this comprehensive *user manual*, and a *license* file LICENSE indicating that the package is distributed under GNU GENERAL PUBLIC LICENSE (Version 3). Then under the Matlab folder, you need to add one of its subfolder, i.e., the Src folder, to the MATLAB path. The directory structure of the Matlab folder is described as follows.

- Src (core functions for conducting KCC)
  - BasicCluster_RFS.m (function to generate BPs with RFS)
  - BasicCluster_RPS.m (function to generate BPs with RPS)
  - Preprocess.m (function to prepare for consensus clustering)
  - KCC.m (consensus function)
  - exMeasure.m (function to compute external validity scores)
  - inMeasure.m (function to compute internal validity scores)
  - load_sparse.m (auxiliary function to load input text data as a sparse matrix)
  - hungarian.m (auxiliary function for cluster label assignment)
  - BasicCluster_RPS_missing.m (auxiliary function to generate IBPs with strategy-I)
  - addmissing.m (auxiliary function to generate IBPs using strategy-II)
  - distance_* (distance functions)
  - gClusterDistribution.m (auxiliary function to calculate cluster distribution for BPs)
  - Ucompute.m, Ucompute_miss.m (auxiliary function for utility calculation)
  - gCentroid.m, gCentroid_miss.m (auxiliary function for centroid update)
  - sCentroid.m, sCentroid_miss.m (auxiliary function for centroid initialization)
- Drivers (illustrative examples)
  - data (input data for illustration)
  - demo.m (function for KCC with different utility functions)
  - demoIBPI.m (function for KCC with IBPs generated by strategy-I)
  - demoIBPII.m (function for KCC with IBPs generated by strategy-II)

[1]Department of Informatics, Technical University of Munich, Germany, linh@in.tum.de.
[2]Michtom School of Computer Science, Brandeis University, USA, hongfuliu@brandeis.edu.
[3]School of Economics and Management, Beihang University, China, wujj@buaa.edu.cn.
[4]School of Economics and Management, Beihang University, China, hong_lee@buaa.edu.cn.
[5]Department of Informatics, Technical University of Munich, Germany, guennemann@in.tum.de.
[6]The software package is publicly available at https://gitee.com/linhaobuaa/KCC.

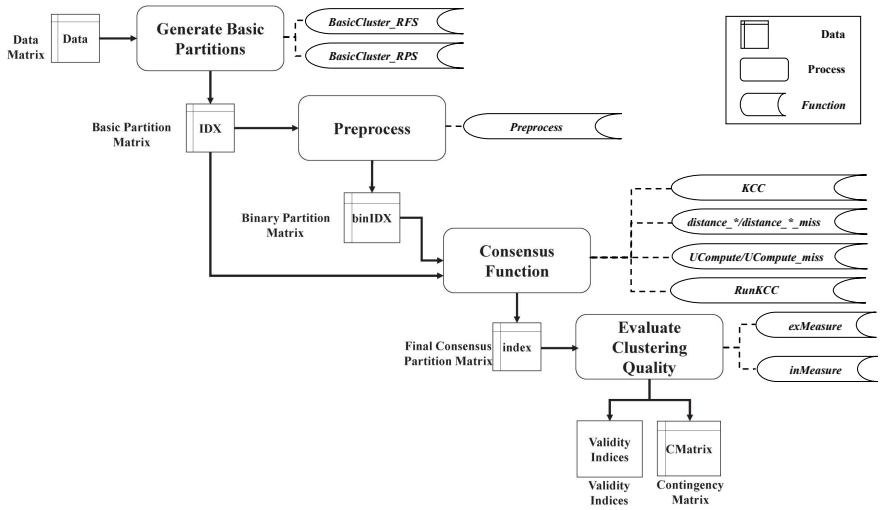


Fig. 1: Typical program flow of KCC package.

demoNumberBP.m (function for KCC with varying number of BPs)
demoStrategyBP.m (function for KCC with RFS strategy for BP generation)
demoEvacluster.m (function for cluster evaluation and selection of K)
demoEvaTimeMem.m (function for recording execution time and memory usage)

The package was developed and tested in MATLAB R2022a under Linux. To those without access to MATLAB and those who prefer to use free open source software, we also investigate the usage of KCC with OCTAVE. Generally, the Octave supports drop-in compatibility with the MATLAB scripts in our KCC package. To use the KCC package with OCTAVE, the users need to firstly do an additional installation step, i.e., installing the statistics and io packages on the OCTAVE command line using "pkg install -forge statistics;" and "pkg install -forge io;", respectively. Then, the users should type the commands "pkg load statistics;" and/or "pkg load io;" on the OCTAVE command line before executing the functions/scripts in the KCC package if needed. All demonstration scripts with the prefix named "demo" under the Matlab/Drivers folder have passed the test on a personal computer with GNU OCTAVE 7.1.0 and macOS 12.4.

In Figure 1, we show the package's typical workflow, which provides a quick overview for using the package. The workflow starts by inputing a real-world data matrix Data into the basic partition generation functions in order to generate a basic partition matrix IDX. The basic partition matrix is then input to a Preprocess function to produce a sparse representation of $\mathcal{X}^{(b)}$, i.e., the matrices, Ki, sumKi, and binIDX. They are then input to the final consensus clustering, i.e., the consensus function, which produces a consensus partition matrix index. Lastly, the clustering quality is evaluated with an inMeasure/exMeasure function, which outputs multiple internal/external validity indices.

Organization of this user manual. In Section 2, we give some typical examples with MATLAB code snippets, tables and figures to help the readers to quickly understand how to work properly and effectively with the package. Next, we provide details for each of the package's available functions, such as syntax, input and output, in Section 3.

	$attr_1$	$attr_2$	$attr_3$	$attr_4$
$data_1$	5.1	3.5	1.4	0.2
$data_2$	4.9	3.0	1.4	0.2
$data_3$	4.7	3.2	1.3	0.2
⋮	⋮	⋮	⋮	⋮
$data_{150}$	5.9	3.0	5.1	1.8

(a) A $n \times d$ attribute matrix for *iris* data set, where $n = 150$ and $d = 4$.

	2521	126373	490062		
$data_1$	556	1	...	238	1
$data_2$	755	1	...	195	1
$data_3$	2202	2	...	7848	1
⋮	⋮	⋮	⋮	⋮	⋮
$data_{2521}$	595	1	...	20246	3

(b) A $n \times d$ attribute matrix for *mm* data set, where $n = 2521$ and $d = 126373$.

Table I: Illustration of the data attribute matrix for two data sets.

2. ILLUSTRATIVE EXAMPLES

2.1 A quick introductory example for illustrating the typical workflow

Here we give illustrative examples with UCI and TREC data sets to show how the KCC package works following the typical workflow as previously described.

Importing data. Firstly, we should import the data set into the MATLAB environment. For a UCI data set such as *iris*, the package KCC includes two files under the **Matlab/Drivers/data** folder, i.e., **iris.dat** and **iris_rclass.dat**, which correspond to the attribute information and ground truth cluster labels of data instances, respectively. As illustrated in Listing 1, the data attributes and ground truth cluster labels can be imported with the built-in MATLAB function **load**. For ease of understanding, we also illustrate the attribute information of the *iris* data set in Table Ia.

```
1 data = load('data/iris.dat');
2 true_label = load(strcat('data/', strcat('iris', '_rclass.dat')));
```

Listing 1: Using **load** to import data attributes and ground truth cluster labels.

For TREC data sets such as *mm*, as indicated in Listing 2, we use the function **load_sparse** implemented in the KCC package to import the data attributes as a sparse matrix `sp_mtx`, and use the built-in function **load** to import the ground truth cluster labels. The attribute matrix of the *mm* data set input to **load_sparse** is illustrated in Table Ib. We can see that the input attribute matrix is in a CLUTO sparse matrix format. The first line gives some meta information of the data set, which includes the number of instances n, the number of features d, and the total number of non-zero elements if recovering the data set as a $n \times d$ matrix. The file's remaining n lines preserve the non-zero values of the matrix in a sparse manner [Karypis 2002].

```
1 [sp_mtx, n, m, count] = load_sparse('data/mm.mat');
2 data = sp_mtx;
3 true_label = load(strcat('data/',strcat('mm', '_rclass.dat')));
```

Listing 2: Using **load_sparse**/**load** to import data attributes/ground truth cluster labels.

Generating basic partitions. As indicated in Figure 1, we should then generate basic partitions (BPs). Concretely, we can utilize the **BasicCluster_RPS** function to produce BPs with Random Parameter Selection (RPS) strategy. We set the number of BPs to 100 [Luo et al. 2011; Liu et al. 2016], because this setting can usually produce good consensus clustering results. For the data set *iris*, we use squared euclidean distance as the distance metric of clustering algorithm to generate BPs, while we use cosine distance for

	BP_1	BP_2	BP_3	BP_4	BP_5	BP_6	BP_7	BP_8	BP_9	...	BP_{100}
$data_1$	1	1	1	4	7	4	6	1	10	...	7
$data_2$	1	8	4	3	1	2	4	1	1	...	7
$data_3$	1	8	10	3	1	2	4	1	6	...	7
$data_4$	1	8	10	3	1	2	4	1	6	...	7
$data_5$	1	1	1	4	7	4	6	1	10	...	7
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$data_{150}$	2	3	3	1	8	1	5	2	11	...	8

Table II: A $n \times r$ basic partition matrix `IDX` for the *iris* data set, where $n = 150$ and $r = 100$.

the *mm* data set. The code snippet is shown in Listing 3. We also illustrate the obtained basic partition matrix `IDX` for the *iris* data set in Table II. Each entry in `IDX` represents a cluster label that the corresponding data object is assigned to in a basic partition.

```
IDX = BasicCluster_RPS(data, 100, 3, 'sqEuclidean', 1); % for iris
IDX = BasicCluster_RPS(data, 100, 2, 'cosine', 1); % for mm
```

Listing 3: Using **`BasicCluster_RPS`** to generate BPs.

For further consideration of efficiency, we can also utilize the multicore-processor architecture to reduce the execution time of the KCC package. In fact, the parallel 'for-loop' function, i.e., **`parfor`**, in MATLAB Parallel Computing Toolbox can be employed in the KCC package for the computation acceleration purpose. Listing 4 demonstrates how the **`parfor`** function can be used to help accelerate the generation process of basic partitions in the **`BasicCluster_RPS`** function.

```
parfor i = 1:r
    IDX(:, i) = kmeans(Data, Ki(i), 'distance', dist, 'emptyaction', 'singleton', 'replicates', 1);
end
```

Listing 4: Using **`parfor`** to accelerate the process of generating BPs.

Preprocessing data and conducting consensus function. As a next step of the workflow in Figure 1, the function **`RunKCC`** is executed with the basic partition matrix `IDX` as the input. An illustrated code to conduct consensus clustering on the *iris* data set with U_h utility function and $K = 3$ is shown in Listing 5. The **`RunKCC`** is composed of two key steps, which involves executing the **`Preprocess`** function to prepare input for consensus clustering and executing the core function **`KCC`** 10 times to find the best consensus partition `pi_index`, which has the best objective function value.

```
K = 3;
U = {'U_h', 'std', []};
r = size(IDX, 2);
w = ones(r, 1);
rep = 10;
maxIter = 40;
minThres = 1e-5;
utilFlag = 1;
[pi_sumbest, pi_index, ~, ~~, ~~~] = RunKCC(IDX, K, U, w, rep, maxIter, minThres, utilFlag);
```

Listing 5: Using **`RunKCC`** to conduct consensus clustering.

Evaluating clustering quality. As a last step of the workflow in Figure 1, users can utilize the external validity metrics to evaluate the clustering quality with the **exMeasure** function in KCC package. The code to launch the **exMeasure** function is shown in Listing 6, which saves values of five external validity measurements to a result matrix file for further comparison.

```
[Acc, Rn, NMI, VIn, VDn, labelnum, ncluster, cmatrix] = exMeasure(pi_index, true_label);
filename = strcat('iris', strcat('_', lower(U{1, 1})));
filename = strcat(filename, strcat('_', lower(U{1, 2})));
if ~isempty(U{1, 3})
    filename = strcat(filename, strcat('_', num2str(lower(U{1, 3}))));
end
filename = strcat(filename, '.mat');
save(filename, 'Acc', 'VIn', 'VDn', 'Rn', 'NMI');
```

Listing 6: Using **exMeasure** to evaluate the clustering quality.

When the ground truth cluster labels are not available in real-world settings, we can utilize the internal validity metrics to evaluate a clustering with the **inMeasure** function in KCC package. An illustrated code snippet is shown in Listing 7.

```
[Distortion, Silhouette, CH] = inMeasure(X, cluster, k);
```

Listing 7: Using **inMeasure** to evaluate the clustering quality.

2.2 An illustrative example of choosing effective and efficient utility function

Choosing effective and efficient utility function is a common need for the KCC package in practical clustering applications. In order to achieve this purpose, the KCC package provides demonstration scripts to compare clustering quality and efficiency with different utility functions on 11 real-world data sets from UCI and TREC repositories.

For effectiveness comparison, the KCC package provides the script **Matlab/Drivers/demo.m**. We briefly explain the main steps in this illustration. The first step utilizes an RPS strategy to generate basic partitions, which calls the **BasicCluster_RPS** function. The next step applies the **RunKCC** function to the obtained basic partition matrix to obtain the consensus partition. This step also involves executing consensus clustering with 10 different utility functions, namely U_c, U_H, U_{cos}, U_{L_5}, U_{L_8} and their corresponding normalized versions NU_x. Then the script conducts clustering quality evaluation and reports five external validity indices, including CA, NMI, R_n, VI_n and VD_n. For robustness concern, the script conducts the **RunKCC** function 10 times to obtain the average validity values. In the following, we show the most important code snippet for this illustration.

```
U_array = {{'U_H','std',[]} {'U_H','norm',[]} {'U_c','std',[]} {'U_c','norm',[]} {'U_cos','std',[]}
{'U_cos','norm',[]} {'U_lp','std',[5]} {'U_lp','norm',[5]} {'U_lp','std',[8]} {'U_lp','norm',[8]}};
IDX = BasicCluster_RPS(data, r, K, dist_of_basic_cluster, randKi);
output_foldername='ResultDemo/';
mkdir ResultDemo;

for uidx = 1:length(U_array)
    avgAcc = 0;
    avgRn = 0;
    avgNMI = 0;
```

	U_c	U_H	U_{cos}	U_{L5}	U_{L8}	NU_c	NU_H	NU_{cos}	NU_{L5}	NU_{L8}
breast_w	0.6403	0.9624	0.7172	0.6820	0.6896	0.6260	**0.9639**	0.6838	0.6820	0.6820
ecoli	0.5639	0.5789	0.5651	0.5756	0.5590	0.5831	0.5578	**0.5880**	0.5605	0.5657
iris	0.8940	0.8973	**0.9000**	**0.9000**	0.8940	0.8933	0.8907	**0.9000**	**0.9000**	0.8880
pendigits	0.6526	0.6584	**0.6837**	0.6422	0.6520	0.5947	0.6443	0.6833	0.6600	0.6533
satimage	0.5829	0.6571	0.6149	0.6060	0.6290	0.5229	0.6425	**0.6594**	0.6006	0.5597
dermatology	0.2913	**0.3128**	0.2729	0.2701	0.2723	0.3075	0.2975	0.2802	0.2760	0.2899
wine	0.5135	**0.5247**	0.5180	0.5112	0.5112	0.5208	0.5079	0.5219	0.5157	0.5152
mm	0.9337	0.9497	0.9532	0.9336	**0.9551**	0.7797	0.9495	0.9415	0.9439	0.9093
reviews	0.6153	0.6593	0.6313	0.6407	0.6569	0.6086	**0.6698**	0.6030	0.6353	0.6198
la12	0.4626	0.4912	**0.5064**	0.4755	0.4690	0.4349	0.4890	0.4853	0.4491	0.4476
sports	0.4497	**0.4858**	0.4715	0.4642	0.4502	0.4592	0.4584	0.4764	0.4693	0.4461
score	0.9182	**0.9898**	0.9462	0.9283	0.9314	0.8910	0.9707	0.9477	0.9265	0.9135

Table III: Clustering quality of KCC with different utility functions in terms of CA.

	U_c	U_H	U_{cos}	U_{L5}	U_{L8}	NU_c	NU_H	NU_{cos}	NU_{L5}	NU_{L8}
breast_w	0.2430	0.7558	0.2575	0.2024	0.1998	0.2326	**0.7690**	0.1850	0.2024	0.2024
ecoli	0.5809	0.5941	0.5861	0.5911	0.5855	0.5937	0.5922	**0.5952**	0.5839	0.5916
iris	0.7905	0.7937	**0.7981**	**0.7981**	0.7905	0.7908	0.7767	**0.7981**	**0.7981**	0.7829
pendigits	0.6978	0.6775	**0.7194**	0.6756	0.6663	0.6732	0.6749	0.7062	0.6816	0.6670
satimage	0.5388	0.5747	0.5695	0.5695	0.5839	0.4699	0.5790	**0.5840**	0.5469	0.5273
dermatology	0.1211	**0.1417**	0.1031	0.0990	0.0916	0.1395	0.1328	0.1238	0.0955	0.1083
wine	0.1675	0.1697	0.1689	0.1669	0.1519	0.1697	0.1511	**0.1701**	0.1682	0.1680
mm	0.6741	0.7249	0.7338	0.6501	**0.7487**	0.3531	0.7211	0.6924	0.7028	0.5949
reviews	0.4879	0.5347	0.5052	0.5375	**0.5575**	0.4538	0.5467	0.4765	0.5147	0.5091
la12	0.2646	**0.3371**	0.3095	0.2882	0.2764	0.2256	0.3234	0.2911	0.2515	0.2630
sports	0.4406	0.4641	0.4531	**0.4754**	0.4715	0.4319	0.4499	0.4693	0.4563	0.4472
score	0.8638	**0.9820**	0.8884	0.8671	0.8647	0.8070	0.9612	0.8842	0.8508	0.8408

Table IV: Clustering quality of KCC with different utility functions in terms of NMI.

```
    avgVIn = 0;
    avgVDn = 0;
    U = U_array{1,uidx};
    for num = 1 : 10
        [pi_sumbest,pi_index,pi_converge,pi_utility,~] = RunKCC(IDX,K,U,w,rep,maxIter,minThres,utilFlag);
        [Acc, Rn, NMI, VIn, VDn, labelnum, ncluster, cmatrix] = exMeasure(pi_index, true_label);
        avgAcc = avgAcc + Acc;
        avgRn = avgRn + Rn;
        avgNMI = avgNMI + NMI;
        avgVIn = avgVIn + VIn;
        avgVDn = avgVDn + VDn;
    end
    avgAcc = avgAcc / num;
    avgRn = avgRn / num;
    avgNMI = avgNMI / num;
    avgVIn = avgVIn / num;
    avgVDn = avgVDn / num;
    filename = strcat([output_foldername '/' datafile],strcat('_',lower(U{1,1})));
    filename = strcat(filename,strcat('_',lower(U{1,2})));
    if ~isempty(U{1,3})
        filename = strcat(filename,strcat('_',num2str(lower(U{1,3}))));
    end
    filename1 = strcat(filename,'.mat');
    save(filename1,'avgAcc', 'avgVIn', 'avgVDn', 'avgRn', 'avgNMI');
end
```

Listing 8: Using **`demo.m`** to evaluate the clustering quality of KCC with different utility functions.

	U_c	U_H	U_{cos}	U_{L5}	U_{L8}	NU_c	NU_H	NU_{cos}	NU_{L5}	NU_{L8}
breast_w	0.0687	0.8537	0.2104	0.1283	0.1391	0.0495	**0.8597**	0.1307	0.1283	0.1283
ecoli	0.3970	0.4230	0.4021	0.4128	0.4021	0.4213	0.4119	**0.4342**	0.4007	0.4110
iris	0.7352	0.7445	**0.7455**	**0.7455**	0.7352	0.7323	0.7287	**0.7455**	**0.7455**	0.7249
pendigits	0.4799	0.5236	**0.5662**	0.5073	0.5010	0.3996	0.5084	0.5551	0.5130	0.4961
satimage	0.3989	**0.5018**	0.4663	0.4650	0.4790	0.2704	0.4908	0.4958	0.4319	0.3955
dermatology	0.0428	**0.0553**	0.0305	0.0259	0.0275	0.0537	0.0442	0.0419	0.0285	0.0352
wine	0.1454	**0.1497**	0.1471	0.1447	0.1375	0.1480	0.1355	0.1484	0.1461	0.1459
mm	0.7564	0.8092	0.8215	0.7522	**0.8284**	0.3454	0.8081	0.7799	0.7893	0.6791
reviews	0.4231	0.5193	0.4797	0.4877	0.5305	0.3738	**0.5334**	0.4403	0.4821	0.4625
la12	0.2046	0.2671	**0.2770**	0.2344	0.2397	0.1574	0.2436	0.2449	0.2059	0.2063
sports	0.2639	0.3193	0.2989	0.3210	0.3229	0.2387	0.3002	**0.3283**	0.3081	0.3038
score	0.7834	**0.9798**	0.8578	0.8185	0.8339	0.6934	0.9342	0.8655	0.8062	0.7910

Table V: Clustering quality of KCC with different utility functions in terms of R_n.

	U_c	U_H	U_{cos}	U_{L5}	U_{L8}	NU_c	NU_H	NU_{cos}	NU_{L5}	NU_{L8}
breast_w	0.7572	0.2442	0.7425	0.7977	0.8003	0.7677	**0.2310**	0.8151	0.7977	0.7977
ecoli	0.4215	0.4079	0.4164	0.4113	0.4169	0.4083	0.4099	**0.4072**	0.4184	0.4109
iris	0.2096	0.2064	**0.2020**	**0.2020**	0.2096	0.2093	0.2234	**0.2020**	**0.2020**	0.2172
pendigits	0.3032	0.3227	**0.2808**	0.3247	0.3340	0.3289	0.3254	0.2940	0.3187	0.3333
satimage	0.4613	0.4253	0.4305	0.4306	**0.4161**	0.5311	0.4210	**0.4161**	0.4532	0.4727
dermatology	0.8790	**0.8584**	0.8970	0.9010	0.9084	0.8605	0.8672	0.8762	0.9045	0.8917
wine	0.8325	0.8303	0.8311	0.8331	0.8481	0.8303	0.8489	**0.8299**	0.8318	0.8320
mm	0.3259	0.2752	0.2662	0.3499	**0.2513**	0.6505	0.2789	0.3077	0.2972	0.4052
reviews	0.5125	0.4660	0.4956	0.4631	**0.4433**	0.5465	0.4539	0.5242	0.4857	0.4915
la12	0.7358	**0.6630**	0.6905	0.7119	0.7236	0.7754	0.6767	0.7090	0.7485	0.7370
sports	0.5612	0.5392	0.5497	**0.5278**	0.5315	0.5691	0.5534	0.5339	0.5467	0.5555
score	0.9060	**0.8145**	0.8721	0.8951	0.8869	0.9780	0.8256	0.8874	0.9027	0.9295

Table VI: Clustering quality of KCC with different utility functions in terms of VI_n.

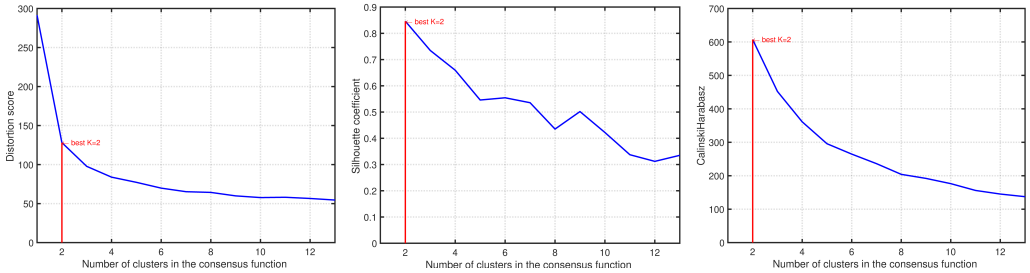
	U_c	U_H	U_{cos}	U_{L5}	U_{L8}	NU_c	NU_H	NU_{cos}	NU_{L5}	NU_{L8}
breast_w	0.9511	0.1087	0.7664	0.8541	0.8378	0.9720	**0.1041**	0.8500	0.8541	0.8541
ecoli	0.4312	0.4323	0.4279	**0.4148**	0.4278	0.4161	0.4248	0.4173	0.4262	0.4270
iris	0.1729	0.1675	**0.1622**	**0.1622**	0.1729	0.1739	0.1779	**0.1622**	**0.1622**	0.1837
pendigits	0.3380	0.3386	**0.3006**	0.3418	0.3467	0.3859	0.3444	0.3133	0.3410	0.3582
satimage	0.4549	**0.3829**	0.4184	0.4198	0.4036	0.5686	0.3980	0.3859	0.4487	0.4805
dermatology	0.9170	**0.8877**	0.9342	0.9370	0.9328	0.9000	0.8977	0.9205	0.9293	0.9246
wine	0.7516	**0.7399**	0.7526	0.7512	0.7570	0.7531	0.7571	0.7534	0.7521	0.7520
mm	0.1546	0.1083	0.1016	0.1504	**0.0968**	0.5553	0.1091	0.1254	0.1244	0.2181
reviews	0.4749	0.4042	0.4089	0.4094	0.3866	0.4915	**0.3799**	0.4496	0.4315	0.4380
la12	0.7167	0.6286	**0.6270**	0.6695	0.6765	0.7898	0.6485	0.6725	0.7091	0.7066
sports	0.6014	0.5335	0.5512	0.5342	0.5372	0.6236	0.5555	**0.5295**	0.5360	0.5644
score	0.8801	**0.7425**	0.8054	0.8314	0.8250	0.9877	0.7536	0.8211	0.8423	0.8821

Table VII: Clustering quality of KCC with different utility functions in terms of VD_n.

	U_c	U_H	U_{cos}	U_{L5}	U_{L8}	NU_c	NU_H	NU_{cos}	NU_{L5}	NU_{L8}
breast_w	0.83	1.95	0.72	0.88	0.76	0.82	0.99	**0.64**	0.76	0.86
ecoli	1.61	3.50	1.24	1.67	1.51	**1.16**	2.06	1.23	1.50	1.40
iris	2.00	7.06	1.61	1.51	1.51	1.49	3.39	1.48	**1.46**	1.48
pendigits	**10.55**	12.87	11.45	13.84	11.87	11.39	13.00	11.19	14.01	13.12
satimage	1.25	1.80	**1.19**	1.42	1.73	1.22	1.62	1.22	1.58	1.96
dermatology	**1.11**	3.59	1.47	1.37	1.15	1.24	1.90	1.20	1.23	1.29
wine	1.86	6.55	2.30	1.84	1.61	1.80	3.28	1.78	1.56	**1.53**
mm	622.61	771.60	733.84	**571.96**	677.55	660.21	739.98	774.26	656.56	588.42
reviews	733.01	1209.34	769.65	693.93	647.68	977.17	1017.01	767.63	756.06	**629.17**
la12	255.44	302.76	283.09	258.23	252.70	266.36	**249.24**	256.89	267.51	252.64
sports	1043.65	1701.98	954.64	1008.88	929.79	989.38	1032.93	**899.55**	952.23	980.68
score	0.55	0.98	0.57	0.57	0.56	0.56	0.69	**0.54**	0.58	0.57

Table VIII: Full execution time of the KCC package with different utility functions. We collect the raw execution time in milliseconds, and normalize it by the number of data objects in the corresponding data set.

The clustering results in terms of the five external validity indices are presented in Tables III-VII. From these tables, we can see that 7 out of 10 utility functions achieve the best clustering performance over at least 1 data set, which is indicated with bold font. This suggests providing flexible utility functions can be crucial to accurate ensemble clustering in real-world applications. In practice, we can hardly know which utility function should



(a) Distortion Score (b) Silhouette Coefficient (c) Calinski and Harabasz index

Fig. 2: Impact of number of clusters in the consensus partition on the *iris* data set.

be used in consensus clustering. We rate utility functions over a testbed, and select the utility function achieving the best score. More specifically, a final score is defined to assess the overall performance of an utility function on a set of data sets. The score is calculated as $score(U_i) = \frac{1}{11}\sum_j \frac{V(U_i,D_j)}{\max_i V(U_i,D_j)}$, where $V(U_i, D_j)$ denotes the clustering validity score obtained by applying U_i on dataset D_j. As can be seen, U_H obtains the best score on all five validity metrics, closely followed by NU_H. As such, we take U_H as the default choice for the KCC package unless otherwise specified.

For efficiency comparison, the KCC package provides the script **Matlab/Drivers/demoEvaTimeMem.m**. We show the full execution time of the KCC with different utility functions in Table VIII. In the computation of execution time, we consider the whole process of using the package, including loading data, generating basic partitions, conducting consensus function, and evaluating the clustering quality. As can be seen, NU_{cos} achieves the best score in terms of efficiency on the 11 data sets.

2.3 An illustrative example of determining the best number of clusters

Automatically determining the best number of clusters is another important need for the KCC package in practical clustering applications. The KCC package provides a demonstration script **Matlab/Drivers/demoEvacluster.m** for this purpose. The script is mainly composed of the following steps. The first step is to utilize three internal metrics (i.e., Distortion Score, Silhouette Coefficient, and Calinski and Harabasz index) to evaluate the quality of clustering results produced by KCC with varying number of clusters. Take the *iris* data set as an example, we show the results in Figure 2. As can be seen, on the iris data set, all three internal metrics generally decrease with the increase of K. The next step involves a selection process to find the best K based on the curve of each metric in Figure 2. For Distortion Score, the Elbow method [Bholowalia and Kumar 2014] is used to pick the elbow point of the curve as the the best K. For Silhouette Coefficient, we choose the best K that produces a clustering solution with the maximum value of average silhouette coefficient. For Calinski and Harabasz index, we choose the best K that produces a clustering solution with the maximum value of Calinski and Harabasz index. As indicated by the red line in Figure 2, all three methods consistently find the best number of clusters as $K = 2$ on the *iris* data set. The most important code snippet is shown below for this illustration.

```
MaxK = ceil(sqrt(size(data, 1))); % max number of clusters to choose from
distortions=zeros(MaxK, 1); % vectors storing the Distortion value for each K
```

```
silhouettes=zeros(MaxK, 1); % vectors storing the Silhouette value for each K
chs=zeros(MaxK, 1);
executiontimes=zeros(MaxK, 1); % vectors storing the execution time of running KCC
for K=1:MaxK % for each K
    tic; % record started computation time in seconds
    [pi_sumbest,pi_index,~,~~,~~~] = RunKCC(IDX,K,U,w,rep,maxIter,minThres,utilFlag);
    t = toc;
    [Distortion, Silhouette, CH] = inMeasure(data, pi_index, K);
    distortions(K,1) = Distortion;
    silhouettes(K,1) = Silhouette;
    chs(K,1) = CH;
    executiontimes(K,1)=t;
end
```

Listing 9: Using **demoEvacluster.m** to determine the best number of clusters for KCC.

3. MATLAB FUNCTIONS THAT ARE PROVIDED TO THE USER

In this section, we give the details of each function, including brief description of the function, syntax of invoking the function, input parameters, output parameters, and a discussion regarding the important details of the function.

3.1 BasicCluster_RFS

This function generates basic partition results using K-means as a basic clustering algorithm with Random Feature Selection (RFS) strategy.

Syntax:

```
IDX = BasicCluster_RFS(Data, r, K, dist, nFeature)
```

Input parameters:

`Data` : input data matrix
`r` : the predefined number of basic partitions in the cluster ensemble
`K` : the predefined number of clusters in the basic partitions
`dist` : the distance measure used in K-means clustering
`nFeature` : the number of randomly selected partial features

Output parameters:

`IDX` : a matrix indicating the cluster labels for data points in basic partitions

Discussion:

The parameter `Data` is an n × p matrix of data, whose rows correspond to n observations, and columns correspond to p features. The available options of the parameter `dist` can be found from the official documentation of the MATLAB kmeans function, and the most widely used distance metric is `'sqEuclidean'`, which denotes the squared Euclidean distance. For `'sqEuclidean'`, the centroid for each cluster is calculated as the mean of the data points in that cluster. For selecting features, we first sample nFeature values uniformly at random without replacement from the integers 1 to p, where p is the dimension of all features in the input data. Then the sampled values are used as the indices of

the selected features. The output `IDX` is a n × r cluster labels matrix for n data points in r basic partitions. Note that this function is a non-deterministic function. Each call may yield a different output matrix IDX, due to the random feature sampling process, and the random initializations in the K-means algorithm.

3.2 BasicCluster_RPS

This function generates basic partition results using K-means as a basic clustering algorithm with Random Parameter Selection (RPS) strategy.

Syntax:

```
IDX = BasicCluster_RPS(Data, r, K, dist, randKi)
```

Input parameters:

`Data` : input data matrix
`r` : the predefined number of basic partitions in the cluster ensemble
`K` : the groundtruth number of clusters for the input data
`dist` : the distance measure used in K-means clustering
`randKi` : the parameter regarding the number of clusters in the basic partitions

Output parameters:

`IDX` : a n × r cluster labels matrix for n data points in r basic partitions

Discussion:

The parameter `K` indicates the groundtruth number of clusters for the input data, which is used as the lower bound of the randomized number of cluster for each basic partition when `randKi == 1`. The parameter randKi has the following options. If `randKi == 1`, this function generates a random number of cluster ranging from `K` to `sqrt(n)`, where n is the number of input data instances; if `randKi` is set to a r × 1 vector, this function produces r basic partitions of which the i-th BP's number of clusters is RandKi(i); otherwise, this function produces r basic partitions with each partition having equal number (i.e., K) of clusters. Note that this function is a non-deterministic function. Each call may yield a different output matrix IDX, due to the random parameter selection process, and the random initializations in the K-means algorithm.

3.3 Preprocess

This function conduct some preprocessing on the input basic partition matrix `IDX` to produce the input for the final consensus clustering, as well as some auxiliary output variables that can help to save memory and accelerate computations.

Syntax:

```
 Ki, sumKi, binIDX, missFlag, missMatrix, distance, Pvector, weight
=
Preprocess(IDX, U, n, r, w, utilFlag)
```

Input parameters:

`IDX` : the input basic partition matrix

`U` : the parameter regarding the chosen utility function

`n` : the number of data instances

`r` : the number of basic partitions in the cluster ensemble

`w` : the weight vector for all basic partitions

`utiFlag` : whether to calculate utility function

Output parameters:

`Ki` : a row vector indicating the number of clusters in all basic partitions

`sumKi` : a matrix indicating the starting indexes for all basic partitions

`binIDX` : a sparse representation of binarization of IDX

`missFlag` : whether the input IDX matrix contains IBPs or not

`missMatrix` indices matrix of the non-zero entries in IDX if there exists IBPs

`distance` : the distance function corresponding to a specific utility function

`Pvector` : a row vector calculated from the contingency matrix

`weight` : an adjusted weight vector adapted for convenient distance calculation

Discussion:

The parameter U is a 1×3 cell array. Its first cell U{1,1} defines the chosen type of the KCC utility function. It currently supports four different types of utility functions which correspond to four different K-means point-to-centroid distance functions, i.e., 'U_c' for Euclidean distance, 'U_H' for Kullback-Leibler Divergence, 'U_cos' for cosine similarity, and 'U_Lp' for Lp-norm. It is worth noting that 'U_Lp' corresponds the distance measure in L_p spaces, which are function spaces defined using a natural generalization of the p-norm for finite-dimensional vector spaces. The second cell U{1,2} is a parameter specifying the chosen form of the KCC utility function, i.e., 'std' for the Standard Form, and 'norm' for the Normalized Form. The third cell U{1,3} is only required to be set when 'U_Lp' is chosen as the utility function. The settings of $p = 1$, $p = 2$, $p \to \infty$ correspond to the Manhattan distance, the euclidean distance and the chebyshev distance, respectively. The parameter w is $r \times 1$ weight vector, of which each entry indicates the weight value assigned to each basic partition in the optimization objective of consensus clustering. The parameter `utiFlag` is a variable indicating whether to calculate utility function during the iterative process of K-means clustering.

The output variable missFlag $\in \{0, 1\}$ indicates whether the input IDX matrix contains incomplete basic partitions (IBPs) or not. The output missMatrix is a mask matrix which represents the indices of the non-zero entries in IDX if there exists IBPs. The output variable distance determines the corresponding distance function to deal with the specific utility function defined by U{1,1}. The output vector Pvector is a $1 \times r$ row vector calculated from the contingency matrix, i.e., $P_k^{(i)}$, which can later be used in calculating distance and utility functions. The output vector weight is a $r \times 1$ adjusted weight vector adapted for convenient distance calculation in later K-means heuristic.

3.4 KCC

This function employs the K-means heuristic to generate a final consensus partition.

Syntax:

```
  sumbest, index, converge, utility = KCC(IDX, K, U, w, weight, distance,
maxIter, minThres, utilFlag, missFlag, missMatrix, n, r, Ki, sumKi,
binIDX, Pvector)
```

Input parameters:

`IDX` : the input basic partition matrix
`K` : the number of clusters in the consensus partition
`U` : the parameter regarding the chosen utility function
`w` : the weight vector for all basic partitions
`weight` : an adjusted weight vector adapted for convenient distance calculation
`distance` : the distance function corresponding to a specific utility function
`maxIter` : the maximum number of iterations for the convergence
`minThres` : the minimum reduced threshold of objective function
`utiFlag` : whether to calculate utility function
`missFlag` : whether there exist IBPs in IDX
`missMatrix` the indices matrix of the non-zero entries in IDX if there exists IBPs
`n` : the number of data points
`r` : the number of basic partitions in the cluster ensemble
`Ki` : a row vector indicating the number of clusters in all basic partitions
`sumKi` : a matrix indicating the starting indexes for all basic partitions
`binIDX` : a sparse representation of binarization of IDX
`Pvector` : a row vector calculated from the contingency matrix

Output parameters:

`sumbest` : the optimal value of the consensus clustering's objective function
`index` : the label vector for the final consensus partition
`converge` : the iterative values of objective function
`utility` : the final utility function value

Discussion:

For convenience of computation, KCC function achieves the consensus clustering under four different conditions, e.g., (a) utility calculation is enabled and there are missing values (utilFlag==1 && missFlag==1); (b) utility calculation is enabled and there are no missing values (utilFlag==1 && missFlag==0); (c) utility calculation is disabled and there are missing values (utilFlag==0 && missFlag==1); (d) utility calculation is disabled and there are no missing values (utilFlag
==0 && missFlag==0). Notably, we introduce a parameter called utilFlag in the KCC function to control whether the K-means heuristic computes the value of the KCC utility function, since the value of the KCC utility function might be of interest to some users

in using the package. The users can choose to the enable or disable the computation of the value of the KCC utility function since it does not affect the process of K-means heuristic. Disabling the computation of the value of the KCC utility function does not affect the clustering performance and can accelerate the computation of KCC. We have added more elaborations on the usage of the parameter utilFlag in the function KCC in the user manual.

3.5 distance_euc

This function performs point-to-centroid distance calculation using euclidean distance measure.

Syntax:

```
D = distance_euc(U, C, weight, n, r, K, sumKi, binIDX)
```

Input parameters:

`U` : the 1×3 utility parameter cell array
`C` : the centroid matrix
`weight` : an $r \times 1$ adjusted weight vector
`n` : the number of data points
`r` : the predefined number of basic partitions in the cluster ensemble
`K` : the predefined number of clusters in the consensus partition
`sumKi` : a matrix indicating the starting indexes for all basic partitions
`binIDX` : a sparse representation of binarization of IDX

Output parameters:

`D` : an $n \times K$ point-to-centroid distance matrix

3.6 distance_cos

This function performs point-to-centroid distance calculation using cosine distance measure.

Syntax:

```
D = distance_cos(U, C, weight, n, r, K, sumKi, binIDX)
```

Input parameters:

`U` : the 1×3 utility parameter cell array
`C` : the centroid matrix
`weight` : an $r \times 1$ adjusted weight vector
`n` : the number of data points
`r` : the predefined number of basic partitions in the cluster ensemble
`K` : the predefined number of clusters in the consensus partition
`sumKi` : a matrix indicating the starting indexes for all basic partitions
`binIDX` : a sparse representation of binarization of IDX

Output parameters:

`D` : an $n \times K$ point-to-centroid distance matrix

3.7 distance_kl

This function performs point-to-centroid distance calculation using KL-divergence measure.

Syntax:

```
D = distance_kl(U, C, weight, n, r, K, sumKi, binIDX)
```

Input parameters:

`U` : the 1 × 3 utility parameter cell array

`C` : the centroid matrix

`weight` : an r × 1 adjusted weight vector

`n` : the number of data points

`r` : the predefined number of basic partitions in the cluster ensemble

`K` : the predefined number of clusters in the consensus partition

`sumKi` : a matrix indicating the starting indexes for all basic partitions

`binIDX` : a sparse representation of binarization of IDX

Output parameters:

`D` : an n × K point-to-centroid distance matrix

3.8 distance_lp

This function performs point-to-centroid distance calculation using L_p norm measure.

Syntax:

```
D = distance_lp(U, C, weight, n, r, K, sumKi, binIDX)
```

Input parameters:

`U` : the 1 × 3 utility parameter cell array

`C` : the centroid matrix

`weight` : an r × 1 adjusted weight vector

`n` : the number of data points

`r` : the predefined number of basic partitions in the cluster ensemble

`K` : the predefined number of clusters in the consensus partition

`sumKi` : a matrix indicating the starting indexes for all basic partitions

`binIDX` : a sparse representation of binarization of IDX

Output parameters:

`D` : an n × K point-to-centroid distance matrix

3.9 distance_euc_miss

This function performs point-to-centroid distance calculation over data with IBPs using euclidean distance measure.

Syntax:

```
D = distance_euc_miss(U, C, weight, n, r, K, sumKi, binIDX, missMatrix)
```

Input parameters:

`U` : the 1 × 3 utility parameter cell array
`C` : the centroid matrix
`weight` : an r × 1 adjusted weight vector
`n` : the number of data points
`r` : the predefined number of basic partitions in the cluster ensemble
`K` : the predefined number of clusters in the consensus partition
`sumKi` : a matrix indicating the starting indexes for all basic partitions
`binIDX` : a sparse representation of binarization of IDX
`missMatrix` the indices matrix of the non-zero entries in IDX if there exists IBPs

Output parameters:

`D` : an n × K point-to-centroid distance matrix

3.10 distance_cos_miss

This function performs point-to-centroid distance calculation over data with IBPs using cosine distance measure.

Syntax:

```
D = distance_cos_miss(U, C, weight, n, r, K, sumKi, binIDX, missMatrix)
```

Input parameters:

`U` : the 1 × 3 utility parameter cell array
`C` : the centroid matrix
`weight` : an r × 1 adjusted weight vector
`n` : the number of data points
`r` : the predefined number of basic partitions in the cluster ensemble
`K` : the predefined number of clusters in the consensus partition
`sumKi` : a matrix indicating the starting indexes for all basic partitions
`binIDX` : a sparse representation of binarization of IDX
`missMatrix` the indices matrix of the non-zero entries in IDX if there exists IBPs

Output parameters:

`D` : an n × K point-to-centroid distance matrix

3.11 distance_kl_miss

This function performs point-to-centroid distance calculation over data with IBPs using KL-divergence measure.

Syntax:

```
D = distance_kl_miss(U, C, weight, n, r, K, sumKi, binIDX, missMatrix)
```

Input parameters:

`U` : the 1 × 3 utility parameter cell array
`C` : the centroid matrix
`weight` : an r × 1 adjusted weight vector
`n` : the number of data points
`r` : the predefined number of basic partitions in the cluster ensemble
`K` : the predefined number of clusters in the consensus partition
`sumKi` : a matrix indicating the starting indexes for all basic partitions
`binIDX` : a sparse representation of binarization of IDX
`missMatrix` the indices matrix of the non-zero entries in IDX if there exists IBPs

Output parameters:

`D` : an n × K point-to-centroid distance matrix

3.12 distance_lp_miss

This function performs point-to-centroid distance calculation over data with IBPs using L_p norm measure.

Syntax:

```
D = distance_lp_miss(U, C, weight, n, r, K, sumKi, binIDX, missMatrix)
```

Input parameters:

`U` : the 1 × 3 utility parameter cell array
`C` : the centroid matrix
`weight` : a r × 1 adjusted weight vector
`n` : the number of data points
`r` : the number of basic partitions in the cluster ensemble
`K` : the predefined number of clusters
`sumKi` : a matrix indicating the starting indexes for all basic partitions
`binIDX` : the sparse representation matrix
`missMatrix` the indices matrix of the non-zero entries in IDX if there exists IBPs

Output parameters:

`D` : an n × K point-to-centroid distance matrix

3.13 UCompute

This function performs utility calculation on data sets without missing values.

Syntax:

```
util = UCompute(index, U, w, C, n, r, K, sumKi, Pvector)
```

Input parameters:

`index` : an n × 1 cluster assignment matrix
`U` : the 1 × 3 utility parameter cell array
`w` : the r × 1 adjusted weight parameter vector
`C` : the centroid matrix
`n` : the number of data points
`r` : the number of basic partitions in the cluster ensemble
`K` : the predefined number of clusters in the consensus partition
`sumKi` : a matrix indicating the starting indexes for all basic partitions
`Pvector` : a 1 × r row vector calculated from the contingency matrix

Output parameters:

`util` : the utility values calculated from the optimization objective of consensus clustering

Discussion:

The output `util` is a 2 × 1 cell array including a utility gain and an adjusted utility value.

3.14 UCompute_miss

This function performs utility calculation on data sets with missing values.

Syntax:

```
util = UCompute_miss(index, U, w, C, n, r, K, sumKi, Pvector, M)
```

Input parameters:

`index` : an n × 1 cluster assignment matrix
`U` : the 1 × 3 utility parameter cell array
`w` : the r × 1 adjusted weight parameter vector
`C` : the centroid matrix
`n` : the number of data points
`r` : the number of basic partitions in the cluster ensemble
`K` : the predefined number of clusters in the consensus partition
`sumKi` : a matrix indicating the starting indexes for all basic partitions
`Pvector` : a 1 × r row vector calculated from the contingency matrix
`M` : a mask matrix indicating the indices of the non-zero entries in IDX

Output parameters:

`util` : the utility values calculated from the optimization objective of consensus clustering

3.15 RunKCC

This function combines the two functions, i.e., Preprocess and KCC, for finding the consensus partition in one step.

Syntax:

```
[pi_sumbest, pi_index, pi_converge, pi_utility, t] = RunKCC(IDX, K, U, w, rep, maxIter, minThres, utilFlag)
```

Input parameters:

- `IDX` : the basic partition matrix
- `K` : the predefined number of clusters in the consensus partition
- `U` : the 1×3 utility parameter cell array
- `w` : the $r \times 1$ adjusted weight parameter vector
- `rep` : the number of repeated KCC experiments for selecting the best result
- `maxIter` : the maximum number of iterations for the convergence
- `minThres` : the minimum reduced threshold of objective function
- `utiFlag` : whether to calculate utility function

Output parameters:

- `pi_sumbest`: the value of objective function for the best KCC experiment
- `pi_index` : the cluster assignment matrix in the consensus partition for the best KCC experiment
- `pi_converge`: the iterative values of objective function for the best KCC experiment
- `pi_utility`: the utility values for the best KCC experiment
- `t` : the execution time cost of the whole process for this function

3.16 exMeasure

This function assesses the clustering quality of the results obtained by a clustering algorithm with external validity indices.

Syntax:

```
[Acc, Rn, NMI, VIn, VDn, labelnum, ncluster, cmatrix] = exMeasure(cluster, true_label)
```

Input parameters:

- `cluster` : a $n \times 1$ cluster assignment matrix returned by a clustering algorithm
- `true_label`: the true class labels for the data points

Output parameters:

- `Acc` : the classification accuracy
- `Rn` : the normalized Rand statistic
- `NMI` : the normalized mutual information
- `VIn` : the normalized Variation of Information
- `VDn` : the normalized van Dongen criterion
- `labelnum` : the number of unique labels in the groundtruth data

`ncluster` : the number of clusters returned by the algorithm
`cmatrix` : a ncluster × labelnum contingency matrix

Discussion:

This function implements five external validity indices, including classification accuracy [Nguyen and Caruana 2007] (CA), the normalized mutual information [Cover and Thomas 2012] (NMI), the normalized Rand statistic [Rand 1971] (R_n), the normalized van Dongen criterion [Dongen 2000] (VD_n), and the normalized Variation of Information [Cover and Thomas 2012] (VI_n). In implementing the exMeasure function, we utilize a function called bestMap written by [Cai et al. 2005], in which the Hungarian method [Carpaneto and Toth 1980] is employed to resolve the label assignment issue in clustering.

3.17 inMeasure

This function assesses the clustering quality of the results obtained by a clustering algorithm with internal validity indices.

Syntax:

```
[Distortion, Silhouette] = inMeasure(IDX, cluster, U)
```

Input parameters:

`IDX` : the input basic partition matrix for KCC
`cluster` : the clustering decision matrix returned by KCC
`U` : the 1 × 3 utility parameter cell array

Output parameters:

`Distortion` the distortion score
`Silhouette` the average silhouette coefficient value of all data objects

Discussion:

This function implements two internal validity indices, including Distortion Score [Bradley and Fayyad 1998] and Silhouette Coefficient [Kaufman and Rousseeuw 2009]. Distortion Score corresponds to the sum of the distance squared between the data objects and the centroid of their assigned cluster.

REFERENCES

Purnima Bholowalia and Arvind Kumar. 2014. EBK-means: A clustering technique based on elbow method and k-means in WSN. *International Journal of Computer Applications* 105, 9 (2014).

Paul S Bradley and Usama M Fayyad. 1998. Refining initial points for k-means clustering.. In *ICML*, Vol. 98. Citeseer, 91–99.

D. Cai, X. He, and J. Han. 2005. Document Clustering Using Locality Preserving Indexing. *IEEE Trans. Knowl. Data Eng.* 17, 12 (Dec 2005), 1624–1637.

Giorgio Carpaneto and Paolo Toth. 1980. Algorithm 548: Solution of the assignment problem [H]. *ACM Transactions on Mathematical Software (TOMS)* 6, 1 (1980), 104–111.

Thomas M Cover and Joy A Thomas. 2012. *Elements of Information Theory*. John Wiley & Sons.

Stijn Dongen. 2000. *Performance Criteria for Graph Clustering and Markov Cluster Experiments*. Technical Report. Amsterdam, The Netherlands, The Netherlands.

George Karypis. 2002. *CLUTO-a clustering toolkit.* Technical Report. MINNESOTA UNIV MINNEAPOLIS DEPT OF COMPUTER SCIENCE.

Leonard Kaufman and Peter J Rousseeuw. 2009. *Finding groups in data: an introduction to cluster analysis.* John Wiley & Sons.

Hongfu Liu, Ming Shao, Sheng Li, and Yun Fu. 2016. Infinite Ensemble for Image Clustering. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. ACM, New York, NY, USA, 1745–1754. https://doi.org/10.1145/2939672.2939813

Dijun Luo, Chris Ding, Heng Huang, and Feiping Nie. 2011. Consensus Spectral Clustering in Near-linear Time. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, Washington, DC, USA, 1079–1090. https://doi.org/10.1109/ICDE.2011.5767925

Nam Nguyen and Rich Caruana. 2007. Consensus Clusterings. In *Proceedings of the 2007 Seventh IEEE International Conference on Data Mining (ICDM '07)*. IEEE Computer Society, Washington, DC, USA, 607–612. https://doi.org/10.1109/ICDM.2007.73

William M Rand. 1971. Objective Criteria for the Evaluation of Clustering Methods. *J. Am. Stat. Assoc.* 66, 336 (1971), 846–850.