San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

2010

Clustering and Validation of Microarray Data Using Consensus Clustering

Sarbinder Kallar
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Computer Sciences Commons

Recommended Citation
Kallar, Sarbinder, "Clustering and Validation of Microarray Data Using Consensus Clustering" (2010). *Master's Projects*. 61.
DOI: https://doi.org/10.31979/etd.ek6y-js4j
https://scholarworks.sjsu.edu/etd_projects/61

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Clustering and Validation of Microarray Data Using Consensus Clustering

A Report

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Sarbinder Kallar

April 2010

© 2010

Sarbinder Kallar

ALL RIGHTS RESERVED

Abstract

Clustering and Validation of Microarray Data Using Consensus Clustering

by Sarbinder Kallar

Clustering is a popular method to glean useful information from microarray data. Unfortunately the results obtained from the common clustering algorithms are not consistent and even with multiple runs of different algorithms a further validation step is required. Due to absence of well defined class labels, and unknown number of clusters, the unsupervised learning problem of finding optimal clustering is hard. Obtaining a consensus of judiciously obtained clusterings not only provides stable results but also lends a high level of confidence in the quality of results. Several base algorithm runs are used to generate clusterings and a co-association matrix of pairs of points is obtained using a configurable majority criterion. Using this consensus as a similarity measure we generate a clustering using four algorithms. Synthetic as well as real world datasets are used in experiment and results obtained are compared using various internal and external validity measures. Results on real world datasets showed a marked improvement over those obtained by other researchers with the same datasets.

ACKNOWLEDGEMENTS

I am grateful to Dr. Khuri for introducing me to the interesting research areas of bioinformatics. Without his patient guidance, advice, and encouragement, I could not have completed this endeavor.

I am grateful to Dr. Chun for accepting to serve on my committee. His advice and feedback vastly improved the quality of this report.

I am grateful to Mr. Butt for his advice, support and encouragement.

TABLE OF CONTENTS

1. INTRODUCTION 1

2. CLUSTERING 3

2.1 Data: Intensity Matrix 3

2.2 Clustering 4

2.3 Distance Measures 4
2.3.2 Pearson Correlation Coefficient 4
2.3.3 Spearman Rank Correlation Coefficient 5
2.3.4 Kendall tau Rank Correlation Coefficient 5

2.4 Linkage Rules 6
2.4.1 Single Linkage 6
2.4.2 Complete Linkage 6
2.4.3 Average Linkage 6
2.4.4 Centroid Linkage 6

2.5 Types of Clusterings 7

2.6 Clustering Algorithms 7
2.6.1 K-Means Algorithm 8
2.6.2 Hierarchical Algorithm 9

3. CLUSTERING VALIDATION 10

3.1 Motivation 10
3.1.1 Cluster Stability 10
3.1.2 Significance of clusters 11
3.1.3 Number of Clusters 11
3.1.4 Identifying better clusters 12

3.2 Internal Validation Indexes 13
3.2.1 Dunn’s Validity Index 13
3.2.2 Silhouette Value 14
3.2.3 Hubert Gamma Statistic 14

3.3 External Validation Indexes 15
3.3.1 Jaccard Index 15
3.3.2 Rand Index 15
3.3.3 Adjusted Rand Index 16
3.3.4 Variation of Information 16
3.3.5 Kappa Statistic 16

3.4 Index Performance 17

4. CONSENSUS CLUSTERING 18

4.1 Clustering Aggregation **18**
4.1.1 Algorithm 3: Best Cluster 19

4.2 Consensus Clustering **20**
4.2.1 Algorithm 4: Agglomerative Clustering Algorithm 20
4.2.2 Algorithm 5: Local Search Algorithm 21
4.2.3 Algorithm 6: Greedy Search Algorithm 21
4.2.4 Algorithm 7: Consensus Clustering 22
4.2.5 Algorithm 8: Weighted Consensus Clustering 22
4.2.6 Algorithm 9: Kappa Statistic 23

5. EXPERIMENTS AND RESULTS 25

5.1 Scalability **25**

5.2 Datasets used **25**

5.3 Evaluation Criteria **26**

5.4 Comparative Methods **26**

5.5 Experimental Results **27**
5.5.1 Consensus on Hierarchical Clustering (melanoma dataset) 27
5.5.2 Consensus on Artificial Dataset 30
5.5.3 Number of Clusters 32
5.5.4 Consensus on K-Means clustering (Yeast dataset) 34

5.6 Conclusion and Future Work **41**

APPENDIX A: SOURCE CODE 43

KMeans.java **43**

BestCluster.java **50**

LocalSearch.java **54**

MUtils.java **57**

Rnd.java **63**

GreedySearch.java **64**

Agglomerative.java **66**

BIBLIOGRAPHY 75

LIST OF TABLES

Table 1. Intensity Matrix (Yeast dataset Eisen 1998) 3
Table 2. Interpretation of κ (Viera & Garrett, 2005) 17
Table 3. Clustering Aggregation 19
Table 4. Analysis of Algorithms 35

LIST OF FIGURES

Figure 1. Linkage Rules 7
Figure 2. Mountain Visualization (5 clusters) 12
Figure 3. Clustering Aggregation 19
Figure 4. Complete Linkage (Correlation Centered) 28
Figure 5. Centroid Linkage (Correlation uncentered) 29
Figure 6. Average Linkage (Spearman Rank) 29
Figure 7. Average Linkage (Kendall's tau) 30
Figure 8. K-Means clusterings (k=4 & k=5) 31
Figure 9. Consensus Clustering 31
Figure 10. Validation Indexes vs. Number of Clusters 32
Figure 11. Hubert Gamma Coefficient vs. Number of Clusters 33
Figure 12. Dunn Index vs. Number of Clusters 33
Figure 13. Mountain Visualization (Number of clusters) 34
Figure 14. Avg. Silhouette Width of Clusters 36
Figure 15. Average Toother of Clusters 36
Figure 16. Hubert Gamma Coefficient 37
Figure 17. Average Silhouette Width 37
Figure 18. Entropy of distribution into clusters 38
Figure 19. Dunn Index 38
Figure 20. Sum of squares (within-cluster) 39
Figure 21. Average with/Average between 39
Figure 22. Dunn Index With Varying Clusters 40
Figure 23. Hubert Gamma Coefficient With Varying Clusters 41
Figure 24. Entropy of Cluster Distribution with Varying Clusters 41

1. Introduction

Clustering is a method to discern hidden patterns in data without the need for any supervision and in absence of any prior knowledge. Clustering is a popular method for analysis of microarray data. There are several challenges to clustering of microarray data. The high number of objects and the high number of attributes and attribute types make it difficult to analyze the quality of results. Every clustering algorithm makes assumptions regarding the data model. When the assumptions are not satisfied the clustering results become unreliable. The information regarding data domain is not always available. It has been shown that most deviations in clustering results are due to a small proportion of noisy data which could not be filtered out (McShane 2002). Moreover the different runs of the same or different algorithms deviate in different directions. A judicious selection of algorithms can guarantee that most results are near-optimal most of the times. Thus there is a strong motivation to combine the various clusterings so that the non-standard deviations cancel out. By using a mixture of algorithms, the strength of each algorithm is leveraged.

The consensus clustering approach is based on combining results from multiple runs of the same or different clustering algorithms on the same data. This approach has several advantages over base clustering algorithms. Consistent results provide stable clusters which are dense and well-separated. A high level of confidence can be attributed to the results. Novel results such as outliers and new clusters are obtained which could not have been attained by any base algorithm alone. Consensus algorithms can be highly

optimized for parallel operation. The base algorithms can be run simultaneously and the results combined.

In section 2 some of the clustering algorithms and their strengths and weaknesses are described. In section 3 common methods for validation of clustering results are reviewed. Some algorithms such as K-Means require the number of clusters as an input parameter. In section 4 consensus clustering algorithm is reviewed. Experimental results are discussed in section 5.

2. Clustering

2.1 Data: Intensity Matrix

The microarray data contains test and reference samples. The ratio of test to control gene expression datasets is preprocessed using background correction, log transformation and filtering or replacement of missing data. The data may additionally be centered such that mean of a column value is 0 and standardized to make variance 1. Such standardization results in continuous data distribution with Gaussian shape. Table 1 displays a part of gene expression intensity matrix from the yeast dataset (Eisen 1998). The dataset contains 2467 genes (rows) under 79 biological conditions (columns). Approximately one percent of values is missing and can be replaced using average values.

Table 1. Intensity Matrix (Yeast dataset Eisen 1998)

ORF	alpha 0	alpha 7	alpha 14	alpha 21	alpha 28	alpha 35	alpha 42	alpha 49
YBR166C	0.33	-0.17	0.04	-0.07	-0.09	-0.12	-0.03	-0.2
YOR357C	-0.64	-0.38	-0.32	-0.29	-0.22	-0.01	-0.32	-0.27
YLR292C	-0.23	0.19	-0.36	0.14	-0.4	0.16	-0.09	-0.12
YGL112C	-0.69	-0.89	-0.74	-0.56	-0.64	-0.18	-0.42	-0.34
YIL118W	0.04	0.01	-0.81		-0.3	0.49	0.08	0.19
YDL120W	0.11	0.32	0.03	0.32	0.03	-0.12	0.01	-0.36
YHL025W	-0.47	1	-0.51	-0.25	-0.71	-0.22	-0.3	-0.36
YGL248W	-0.25	0.26	0.01	-0.06	-0.42	-0.07	-0.3	-0.18
YIL146C	-0.58	-0.29	-0.45	-0.15	-0.86	-0.36	-0.54	-0.47
YJR106W	-0.36	-0.17	-0.22	-0.34	-0.36	0.03	-0.2	-0.42
YNL272C	0.31	0.12	0.34	0.61	0.18	0.28	0.14	
YBR123C	-0.17	-0.32	-0.34	-0.42	-0.25	-0.3	0.19	0.26
YCR040W	-0.29	0.31	-0.2	-0.04	-0.38	0.11	-0.2	-0.4
YHR047C	-0.29	-0.07	-0.34	-0.34	-0.36	-0.43	-0.4	-0.25

2.2 Clustering

Clustering is the process of finding patterns or natural groups in datasets. It can be used as an exploratory mechanism for discovering interesting relationships between genes. Clustering can also be used to group experiments e.g. when predicting net survival rates of patients from some disease.

2.3 Distance Measures

Clustering algorithms group genes based on similarity (or dissimilarity) between genes. Similarity is measured using distances between pairs of genes in the multidimensional space. Some common distance measures are:

2.3.1 Euclidean Distance

The straight line geometric distance between points a and b in n-dimensional space is calculated using Pythagorean Theorem (Jain 1999).

Euclidean Distance (a, b) = $\sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$

2.3.2 Pearson Correlation Coefficient

Pearson Correlation Coefficient is a value for the quality of finding best-fit by minimizing sum of squares from the best-fitting curve. For two variables it is defined as the ratio of covariance of the variables to product of their standard deviations (Jain 1999).

$$r = \frac{1}{n-1}\sum_{j=1}^{n}\frac{(X_j - \overline{X})(Y_j - \overline{Y})}{Sx.Sy}$$

2.3.3 Spearman Rank Correlation Coefficient

Spearman Rank Correlation Coefficient is a nonparametric procedure of measuring dependence between variables. It is similar to Pearson correlation coefficient except that it works on rank-order of variables. It is less sensitive to outliers and independent of assumptions about distribution of data.

$$\rho = 1 - \frac{6 \sum_{j=1}^{n} {d_j}^2}{n(n^2-1)}$$

2.3.4 Kendall tau Rank Correlation Coefficient

Kendall tau Rank Correlation Coefficient is another nonparametric procedure for measuring dependence of variables using hypothesis test. It is more intuitive and easier to calculate than Spearman Rank Correlation Coefficient. A pair of data points is considered concordant if the values increase (or decrease) in all dimensions. If the value of one point is higher in one dimension while that of other point is higher in another dimension, the pair is called discordant.

$$\tau = \frac{n_c - n_d}{\frac{n(n-1)}{2}}$$

where n_c= number of concordant nodes

n_d= number of discordant nodes

2.4 Linkage Rules

There are several rules to determine how to apply the distance metric for finding distance between objects and intermediate clusters or the distance between clusters. Figure 1 displays the linkage rules.

2.4.1 Single Linkage

The distance between two nearest neighbors in different clusters is considered the distance between the clusters.

2.4.2 Complete Linkage

The distance between two farthest neighbors in different clusters is considered the distance between the clusters.

2.4.3 Average Linkage

For any pair of clusters, average linkage is the average of distances between all element pairs such that the element pair comprises of one element from each cluster.

2.4.4 Centroid Linkage

The distance between two clusters is the distance between the centroids of the clusters (Bolshakova 2002).

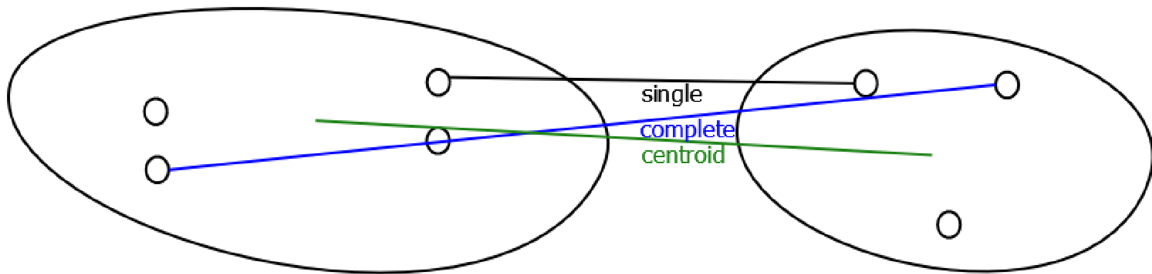

Figure 1. Linkage Rules

2.5 Types of Clusterings

If the elements in a cluster can belong to only one cluster the clustering is considered hard or exclusive. When clusters are allowed to overlap the clustering is considered soft or fuzzy.

When not all elements belong to a cluster (outliers or unclustered), the clustering is considered as partial. When all elements belong to a cluster, the clustering is considered complete.

When clusters could be nested (subclusters), hierarchical clustering is obtained while unnested clusters result in partitioned clustering.

All the features are generally used simultaneously to calculate distances (polythetic) but features are used sequentially (monothetic) by some algorithms (Jain 1999).

2.6 Clustering Algorithms

Several clustering algorithms are available based on clustering types and methodologies (model-based, grid-based, density-based, agglomerative, divisive etc.). K-

Means and Hierarchical algorithms are two commonly used approaches for clustering gene expression data.

2.6.1 K-Means Algorithm

The K-Means algorithm is a partitioning algorithm where the number of clusters, k, is provided a priori. The algorithm initializes k elements as cluster centroids and iteratively adds elements to the nearest centroid. The centroids are updated and the steps are repeated until centroids stabilize.

2.6.1.1 Algorithm 1: K-Means

```
Input: Gene Expression Array G (double [][])
           Number of clusters k

Output: Set of Clusters C (int [])

Randomly assign k elements as centroids

Repeat Until centroids stabilize
     Assign each element to cluster with closest centroid
     Recalculate centroids


Proximity Measure: Euclidean Distance

Objective Function: Assign centroids such that the scatter

(within-cluster sum of squared errors) is minimized.
```

The centroid that optimizes the scatter has been shown to be the mean of cluster elements(Berkhin, 2002). K-Means algorithm is easy to implement and works well for large datasets where partitions are well separated but it is sensitive to noise. Also the algorithm depends on initial choice of partition and converges to local minima which may not be optimal. Several modifications have been proposed to overcome the tendency

of local minima in K-Means algorithms. Multiple runs using different initial clusters can still result in local minima since the number of true partitions is not known especially with high dimensional microarray data. Several methods to overcome the problem have been proposed. Deterministically generating centroids using hierarchical algorithm or incrementally adding cluster centers one at a time (Likas 2001) has been proposed. We found that selecting a centroid that is at least a distance d_e away from all existing centroids results in reasonable accuracy without sacrificing performance. The distance d_e is found by dividing the distance d_{max} between farthest points by k. The distance is halved if new centroid could not be allocated.

2.6.2 Hierarchical Algorithm

Although a divisive (top-down) approach is sometimes used, the agglomerative approach is more common.

2.6.2.1 Algorithm 2: Agglomerative Hierarchical

```
Input: Gene Expression Array G (double [][])

Output: Set of Clusters C (int [])

Assign each element to its own cluster (n clusters)

Repeat Until all elements merged into one cluster
      Merge the two closest clusters
      Recalculate proximity matrix


Proximity Measure: Average Linkage using correlation
```

Hierarchical clustering deterministically returns clustering solution for small datasets. However for large datasets the algorithm performs poorly. The algorithm returns a dendogram but there is no criterion for cutting the tree to determine cluster

membership. Cut is made using visual inspection with the knowledge that cut is made at (1-correlation) height when correlation is used as the distance function.

Clustering algorithms always return a result. The quality of the result is dependent on various factors such as distribution of data, input parameters, starting condition etc. Since multiple runs of even the same algorithm can return different results, an independent evaluation of the results is required. In next section several methods to validate the results from clustering algorithm are reviewed.

3. Clustering Validation

3.1 Motivation

There are several motivations for validating clustering results. Issues such as well-separateness, optimum number of clusters, significance and reproducibility of results must be considered when validating clustering results.

3.1.1 Cluster Stability

Reproducible clustering results can be used to validate results. Reproducibility of individual clusters can be more significant when considering microarray data (McShane 2002). Most algorithms will cluster noisy data in either one of the existing clusters or create an additional cluster (outliers). This implies that the compactness and the number of clusters will depend on abundance of such noisy data. Thus validation using the number of clusters and statistical indexes of complete clusterings is not always reliable. Focusing on individual cluster properties has been shown to provide more reliable results (Kerr 2001). Data are repeatedly clustered after introduction of artificial noise (perturbation) and similarity of results is measured.

3.1.2 Significance of clusters

Most clustering algorithms return results for any input data. Hence there is a need to consider whether any real clusters are present in data. For global test it is difficult to identify a null model for the hypothesis that no clustering exists. McShane (2002) showed that global test can be performed reliably by clustering only the first three principal components. By considering only three dimensions, errors in null distribution due to high dimensionality can be reduced.

3.1.3 Number of Clusters

Milligan (1985) reviewed 30 clustering indexes and showed that optimizing the indexes could identify correct number of clusters. Using predictive power of clustering such as leave-one-out (Yeung 2001) can also be used to find number of clusters. Other methods include using stability of clusters with perturbation of data and specifying new indexes such as GAP statistics (Tibshirani 2001). Most validation indexes monotonically increase (or decrease) with k, the number of clusters. Optimizing the first or second difference of the index provides a good estimate of the number of clusters (Mirkin 2005).

Mountain visualization uses volume, color, height and location of peak to represent a cluster (Rasmussen 2004). The centroid of a cluster is used as location of the peak. The similarity between clusters is represented by the distance between mountain peaks. Internal similarity of a cluster is calculated by averaging the pair-wise similarity and represented as height of peaks. The peaks with red color represent low internal standard deviation of within-cluster objects while high within-cluster deviation is

represented on the blue end of spectrum. The number of objects in a cluster is represented by the volume of the peak. Tall red peaks represent clusters with highly similar objects having low deviation. Looking at closely formed groups of peaks it is possible to estimate the number of stable clusters even though the number of clusters obtained is much higher. Using this approach requires looking at several clusterings with varying number of clusters. A solution representing five clusters is shown in Figure 2 with clusters labeled. The clusters numbered one and four are overlapping and a four cluster solution is expected to be optimal.

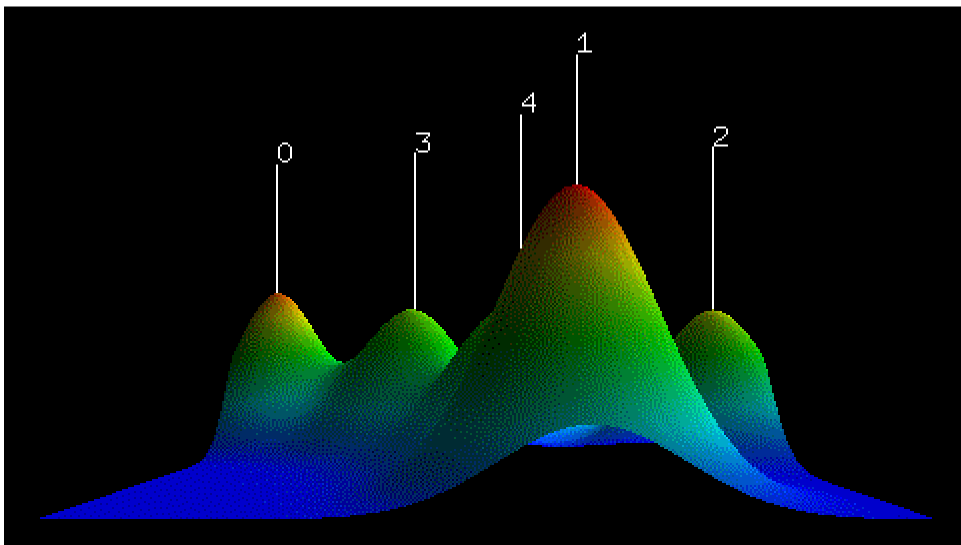

Figure 2. Mountain Visualization (5 clusters)

3.1.4 Identifying better clusters

When different results are obtained by iteratively running same or different algorithms, quality measures to identify better results are needed.

Several statistical indexes have been proposed for measuring the quality of clusterings. The validation indexes can be divided into external and internal validation indexes.

3.2 Internal Validation Indexes

These methods validate individual clustering using the clustering result and input data. Clusters are expected to be compact (low within-cluster distances) and well scattered (high between cluster distances).

3.2.1 Dunn's Validity Index

Dunn's Index measures how compact and well-separated clusters within a clustering are. Higher value of Dunn's index implies that clusterings are more compact and separated(Dunn, 1974).

$$\text{DI}=min_{1\le j\le n}\langle min_{1\le k\le n}\left\{ d_{c_j c_k/(max_{1\le l\le n}d'(C_l))}\right\}\rangle$$

where

$dc_j c_k$ = distance between clusters k and j

$d'(c_l)$ = intercluster distance of cluster l

n = number of clusters

3.2.2 Silhouette Value

For any element the Silhouette value shows ratio of measures by which average between cluster distance exceeds within cluster distance (Bolshakova 2002).

$$Silhouette_i = \frac{dist_{i\ Cluster\ neighbor} - dist_{i\ Cluster\ same}}{\max(dist_{i\ Cluster\ neighbor}, dist_{i\ Cluster\ same})}$$

where

$dist_{i\ Cluster\ neighbor}$ = average distance of element i to other elements in same cluster

$dist_{i\ Cluster\ same}$ = average distance of element i to elements in its nearest neighboring cluster

3.2.3 Hubert Gamma Statistic

Hubert Γ is defined(Halkidi, Batistakis, & Vazirgiannis, 2002) as

$$\Gamma = \frac{2}{N(N-1)} \sum_{i=1}^{N-1} \sum_{k=i+1}^{N} d_{ik}\ Cl_{ik}$$

where

d_{ik} = distance between elements i and k

Cl_{ik} = distance between clusters to which elements i and k belong (represented by centroids)

Entropy:

Assuming that a point has equal probability of belonging to any cluster, the entropy of a clustering is defined as(Meila, 2007):

$$H(C) = -\sum_{i=1}^{k} P(i) \log P(i)$$

where P(i) = $\frac{n_i}{n}$

k = number of clusters

3.3 External Validation Indexes

These methods validate two clustering solutions obtained by different algorithms or different runs of the same algorithm (with some input parameters changed).

3.3.1 Jaccard Index

Jaccard Index measures fraction of element pairs that are placed in same cluster by both clustering. It ignores pairs that are not clustered together in either clustering.

$$JI = \frac{N_{11}}{N_{11} + N_{01} + N_{10}}$$

where

N_{11} = number of pairs of points clustered together in both clusterings

N_{10} = number of pairs clustered in first but not second clustering

N_{01} = number of pairs clustered in second but not first clustering

3.3.2 Rand Index

Rand Index is the fraction of agreements with respect to element pairs that are either clustered together in both clusterings or clustered apart in both clusterings.

$$RI = \frac{N_{11} + N_{00}}{N_{11} + N_{00} + N_{01} + N_{10}}$$

where

N_{00} = number of element pairs that both clusterings did not cluster together

3.3.3 Adjusted Rand Index

The Rand Index has been adjusted such that the normalized index has expected value 0 and value cannot exceed 1 (Meila 2007).

$$\text{ARI} = \frac{\text{RI} - \text{E[R]}}{1 - \text{E[R]}}$$

3.3.4 Variation of Information

It is a measure of information contained in one clustering about the other clustering (Meila 2007).

$$\text{MI} = \sum_{i=1}^{k} \sum_{i'=1}^{k'} P(i, i') \log \frac{P(i,i')}{P(i)P'(i')}$$

where P(i,i') = probability that element i belongs to cluster C_i in one clustering and $C'_{i'}$ in second clustering.

3.3.5 Kappa Statistic

Kappa statistic is a measure of agreement between clustering solutions. The statistic is corrected for chance agreement(Viera & Garrett, 2005).

$$\kappa = \frac{P_o - P_e}{1 - P_e}$$

where P_e = observed agreement probability

P_e = expected agreement probability

A value of $\kappa=1$ implies not only complete agreement but zero probability of agreement happening by chance. Typically a value $\kappa > 0.2$ is considered fair.

Table 2. Interpretation of κ (Viera & Garrett, 2005)

K	interpretation
< 0.0	poor
< 0.2	slight
< 0.4	fair
< 0.6	medium
< 0.8	significant
< 0.99	perfect match

3.4 Index Performance

Most validation indexes are not invariant of the number of clusters and must be scaled and shifted before comparison (Meila 2007). The null model used for rescaling is not intuitive. The Variation of Information does not require any adjustments and has been shown to be more discriminative (Meila 2007). For complex models, external indexes perform better than internal indexes. Validation of clustering results can be used to decide if some minimum criteria such as $\kappa > 0.2$ are being met. When results from two runs of same or different algorithm return identical results, the results are of higher quality. The optimal value of validation indexes is not clear and the indexes are used for comparison purposes only. When one of clusterings is chosen as the best based on validation, the knowledge contained in other clusterings is ignored. To obtain robust results which can be accepted with high level of confidence, methods to aggregate clustering results are reviewed in next section.

4. Consensus Clustering

4.1 Clustering Aggregation

The results from clustering algorithms are not consistent and it is difficult to ascribe any level of confidence to the results. Hierarchical clusterings are not suitable for larger datasets although for small datasets the results are reproducible. Partitioning algorithms such as K-Means and EM perform well on large datasets but results are not consistent since the algorithms converge to local minima. K-Means algorithm performs poorly with noisy data. Using validation indexes clustering quality can be accessed and clusterings can be compared. Outliers often can distort results but constitute very small part of the data. Using repeated runs and reasonably good starting criteria (for K-Means) there is an incentive to aggregate the clusterings.

Clustering Aggregation is the process of aggregating clusterings such that disagreements are minimized. Formally the disagreement $d(C_1, C_2)$ between clusterings C_1 and C_2 can be defined as the number of element pairs that are clustered together in C_1 but not in C_2 and vice versa. Clustering Aggregation algorithms find a clustering C that minimizes the disagreement over all input clusterings $\sum_{i=1}^{n} d(C, C_i)$ (Gionis 2005).

In table 3, four clusterings C_1, C_2, C_3, and C_4 placed four elements e_1, e_2, e_3, and e_4 in one of two clusters. Ignoring minor disagreements, clustering C represents the aggregation of clusterings. The resultant clustering C has the least number of disagreements with the input clusterings (figure 3).

Table 3. Clustering Aggregation

	C_1	C_2	C_3	C_4	**C**
e_1	1	2	1	1	**1**
e_2	1	2	2	2	**2**
e_3	1	1	1	1	**1**
e_4	2	2	1	2	**2**

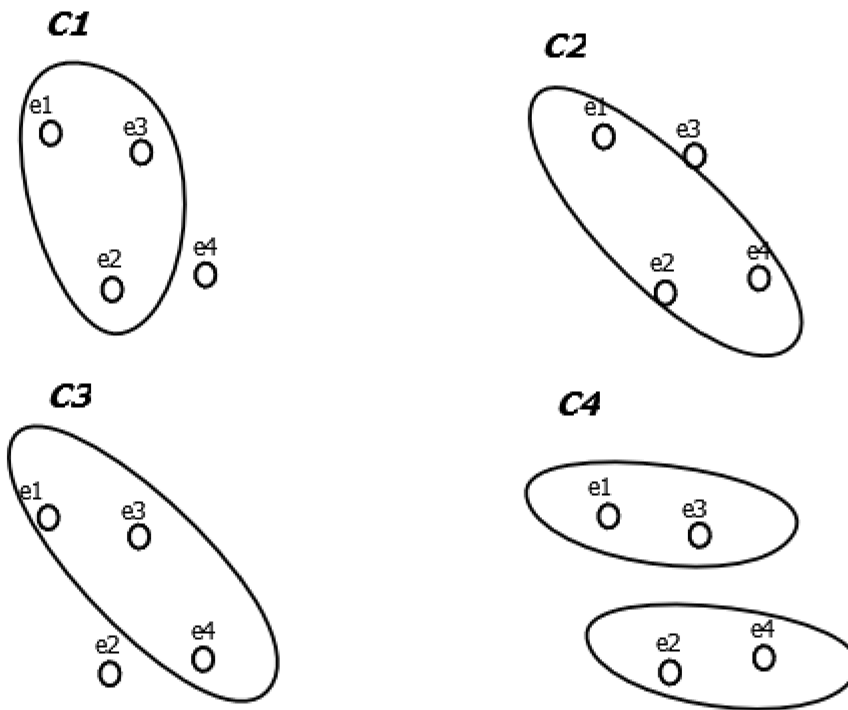

Figure 3. Clustering Aggregation

4.1.1 Algorithm 3: Best Cluster

```
Input: Sets of clustering solutions S (int [][])

Output: Set of Clusters C as best solution (int [])

Generate Similarity Matrix
Calculate cumulative distance of each solution to rest of
solutions in S.
Return the solution with least cumulative distance as best set.
```

Best cluster algorithm is a 2-approximation algorithm.

Given k input clusterings where any clustering can have at most m clusters, the run-time is O(mk).

4.2 Consensus Clustering

The problem of finding a clustering that minimizes disagreements with a given set of clusterings can be generalized. Each object to be clustered can be considered as the vertex of a graph with weighted edges connecting it to other objects. The weight of an edge represents the fraction of input clusterings that place the two vertices it connects in different clusters. Consensus clustering is an optimization strategy wherein edges with high weights (>0.5) are cut while trying to preserve edges with low weights. Individual clustering results can contain random errors. When several runs of different algorithms are made, the systemic errors in experiment can be distributed in results. Since the erroneous output is less common and error distribution varies between results, a consensus can filter out the errors and consistently return results that are nearly optimal.

4.2.1 Algorithm 4: Agglomerative Clustering Algorithm

```
Input: Sets of clustering solutions S (int [][])

Output: Set of Clusters C as consensus solution (int [])

Generate distance matrix D using dissimilarity metric for each
gene pair of S
Initialize solution C such that each gene is in its own cluster
If the proportional dissimilarity distance of a pair < 0.5, merge
the pair into one cluster.
Merge in the increasing order of dissimilarity distances.
Recalculate proportional dissimilarity distance to merged cluster
Stop when no more merge possible i.e. each cluster at distance >
```

0.5.

The agglomerative consensus algorithm generates a true consensus by using majority vote (dissimilarity proportion < 0.5). At worst it is a 2-approximation algorithm. The algorithm has a runtime of O (n^2 log n). Normalized Kappa Statistic is optionally used to calculate the relative significance of input clusterings. Clusterings with $\kappa < 0.0$ are assigned a weight 0 (pruned). The remaining clusterings are weighted using the normalized κ. The calculation of Kappa Statistic does not affect run-time.

4.2.2 Algorithm 5: Local Search Algorithm

```
Input: Sets of clustering solutions S (int [][])

Output: Set of Clusters C as locally optimal solution (int [])

Generate Similarity Matrix
Obtain initial solution C_loc using BestCluster
Do Forever
  For each element in nxn Similarity Matrix
     For each cluster in C_loc
          Move element to next cluster
          Accept move if cumulative distance reduced
  Terminate if cumulative distance cannot be improved
```

Local Search algorithm uses Best Cluster to obtain a starting partition. The starting condition is important since Local Search algorithm iterates until no further improvement in cumulative distance to input clusterings is possible. The algorithm is computation intensive and has a O (n!) run-time. The algorithm is not suitable for large dataset due to computational constraints.

4.2.3 Algorithm 6: Greedy Search Algorithm

```
Input: Sets of clustering solutions S (int [][])
```

```
Output: Set of Clusters C as greedy optimal solution (int [])

Generate Similarity Matrix
Obtain initial solution C_greedy using BestCluster
 For each element in nxn Similarity Matrix
      For each cluster in C_greedy
            maxImprovement=0
           If curImprovement> maxImprovement
                maxImprovement= curImprovement
      If maxImprovement > 0
           Move element
```

The Greedy Search algorithm is a simplification of Local Search algorithm. The algorithm only uses one best possible move for each element and has a O(mn) run-time.

4.2.4 Algorithm 7: Consensus Clustering

```
Input: Gene Expression Array G (double [][])

Output: Set of Clusters C (int [])

For each Base Clustering Algorithm i=1 to K
      *Bootstrapping Step: Resample G using perturbation
        Substitute missing values using average
      Execute using G as input and construct clustering soln Si
      Union with comprehensive clustering soln S

Using S as input construct dissimilarity matrix M for each pair
of genes (See agreement criteria)
Generate distance matrix D based on proportion of disagreement
between sets in S
Execute consensus clustering algorithm using D as input and
output solution C
```

4.2.5 Algorithm 8: Weighted Consensus Clustering

```
Input: Gene Expression Array G (double [][])

Output: Set of Clusters C (int [])

For each Base Clustering Algorithm i=1 to K
```

```
    *Bootstrapping Step: Resample G using perturbation
      Substitute missing values using average
    Execute using G as input and construct clustering soln Si
    Union with comprehensive clustering soln S

Using S as input construct dissimilarity matrix M for each pair
of genes (See agreement criteria)
Generate distance matrix D based on proportion of disagreement
between sets in S
Generate normalized kappa coefficient for each clustering soln
Execute weighted consensus clustering algorithm using D as input
with weight using kappa coefficient and output solution C


Agreement Criteria:
Two sets of clusters agree on a gene pair if both place the pair
in same cluster or if they both place the pair in different
clusters (Gionis 2005).

Distance Metric:
Proportion of clustering solutions that added the gene pair in
same cluster.

Metric Definition:
Triangle Inequality is satisfied by above distance metric(Zuylen
& Williamson, 2008).
```

4.2.6 Algorithm 9: Kappa Statistic

Input: Sets of clustering solutions S (int [][])

Output: Array of normalized kappa (float [])

Generate Agreement Matrix M_i for each clustering
Calculate Agreement Probability Pr_i for each clustering
Calculate Expected Probability $P_e = \prod_{i=1}^{n} Pr_i + \prod_{i=1}^{n} (1 - Pr_i)$
Calculate Observed Probability P_o by comparing corresponding M_i
Return $\kappa = \frac{P_o - P_e}{1 - P_e}$

The Kappa Statistic is in range $-1.0 < \kappa < 1.0$. A clustering with value less than 0.0 can be considered insignificant and thus ignored (pruned). The clusterings with higher κ are considered more significant i.e. less likely to appear by chance. Weights can be assigned to clusterings using normalized κ.

None of the clustering algorithms can be assigned high confidence levels based on a single run. Multiple runs using judicious choice of algorithms and input parameters is recommended. Consensus algorithms can preserve the strengths of the clusterings obtained from these runs while removing noise and erroneous outputs. In next section experimental results using such a strategy are evaluated.

5. Experiments and Results

5.1 Scalability

There is noticeable performance degradation for Local Search (O (n!)) and Agglomerative algorithms with datasets greater than 5000 genes which can be solved by sampling very large datasets before running these algorithms. Very large datasets were not used in this experiment and applying the algorithm to a sub-sample can be a future enhancement. For performance reasons max heap memory must be set to a value close to physical memory size since frequent java garbage collection was found to degrade performance. Program was optimized for parallel operation using localized arrays and reordered indexes (Moreira 2000).

5.2 Datasets used

The yeast dataset from the seminal work by Eisen et al (Eisen, Spellman, Brown, & Botstein, 1998) was used as a real world dataset. Less that 1% of values was missing and was replaced by average values. The yeast dataset contained 2467 genes and 79 experiments. A variety of Synthetic datasets were randomly generated for well-defined as well as loosely defined clusters. In addition melanoma dataset (Bittner, et al., 2000) was used. The Bittner dataset contained 3614 genes and 31 experiments. The original Bittner paper used control datasets (7) and originally contained 8150 cDNAs of which 6971 were unique genes. Only 3613 genes were found to have measurable gene expressions. The ratios were log transformed and genes were normalized using median log-ratios such that median log-ratio for each experiment was 0.

5.3 Evaluation Criteria

The following statistical indexes (Mirkin, 2005) were applied to results for comparison of cluster quality generated:

Hubert Gamma Coefficient

Entropy

Dunn's Index

Within /Between ratio

Avg silhouette width

Avg toother

Within cluster sum of squares

Avg. toother

Corrected Rand index

Variation of information

Milligan(1981) showed Hubert Gamma as the best internal quality measure out of 30 measures considered.

5.4 Comparative Methods

In addition to results of implemented algorithms, results from Eisen paper and Bittner paper were used for comparison. The clusterings generated were compared to the optimal results and various evaluation criteria were applied. Different distance metrics such as Spearman Rank correlation, Kendall's tau coefficient, Euclidean distance were

used when performing Hierarchical clustering on the melanoma dataset. Two separate analyses were performed on yeast datasets. During first analysis the number of clusters was maintained at 5 while K-Means algorithm was run 8-times. Each K-Means algorithm run terminated when two results were matching (+-1%) or the algorithm had run 10 times. For the second analysis 29 results of K-Means algorithm with number of clusters between 5 and 10 were used.   Visual inspections were performed on synthetic datasets to evaluate results.

5.5 Experimental Results

5.5.1 Consensus on Hierarchical Clustering (melanoma dataset)

Bittner et al (Bittner M, 2000) determined 19 samples to be clustered and 12 samples to be unclustered. Out of 3613 genes, 182 genes were identified to be significant by assigning weights to genes that would result in compact clusters with high inter-cluster distances. The weight function used was similar to t-statistics (but adding square roots instead of root of sums squared).The author predicted the metastatic ability of cancer based on the membership in cluster. Using survival information available on 15 patients, authors noted that 7 out of 10 patients survived from the 19-membered tight cluster of less-invasive form of melanoma while only one out of 5 survived from remaining group. With a p-value of 0.135 (<0.05 is norm), the statistical significance is low although this could be due to low event rate and sample size(8 survivals out of 15 patients whose information was available).Using the datasets and various linkage rules, very similar results were obtained using cluster (Eisen, Eisen Lab: Maple Tree Cluster, 2010). The only variation observed was for case# M93-007, which was not found to be a

member of the tight melanoma cluster (figs. 4 & 5) except when distance metric was changed to Spearman Rank coefficient (fig. 6). Thus the consensus result also does not cluster case# M93-007 to be part of the tight melanoma cluster. Unfortunately Bittner et al did not publish the details on case-by-case basis regarding survival of patients and it could not be ascertained if this result improves the prediction about metastatic ability.

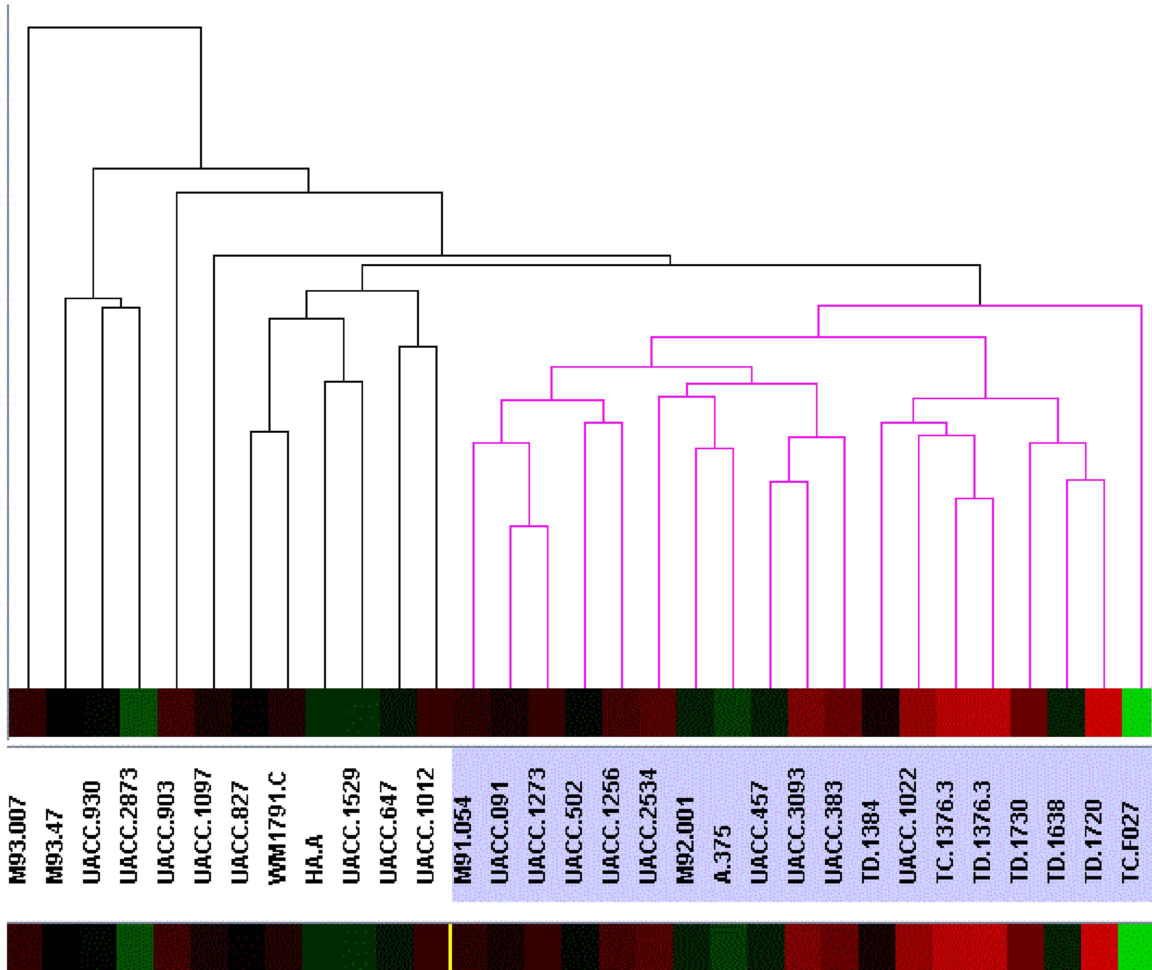

Figure 4. Complete Linkage (Correlation Centered)

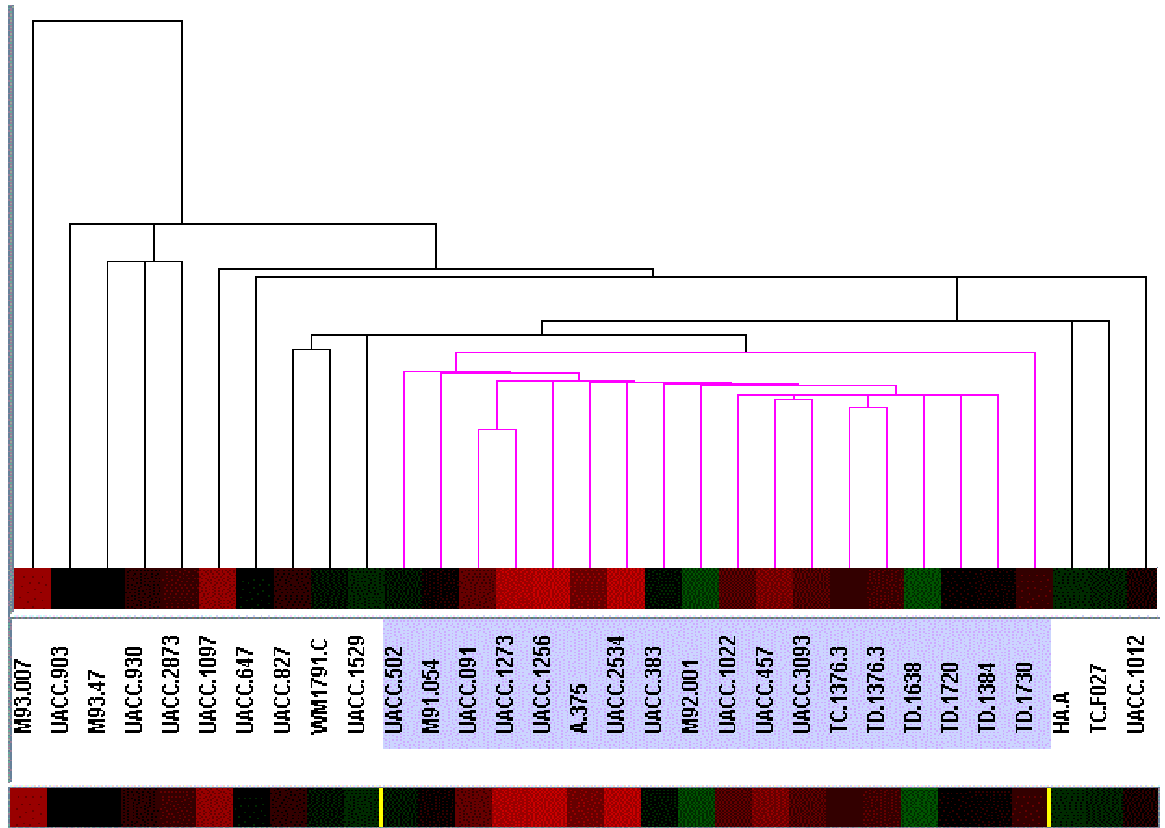

Figure 5. Centroid Linkage (Correlation uncentered)

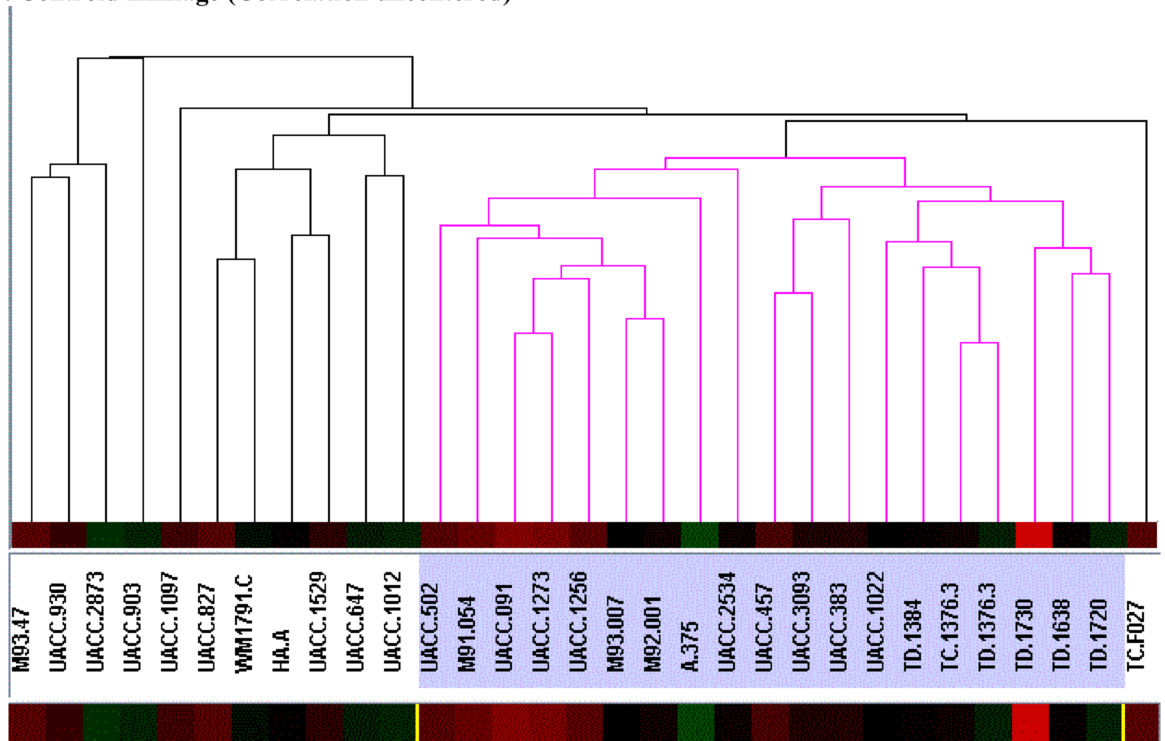

Figure 6. Average Linkage (Spearman Rank)

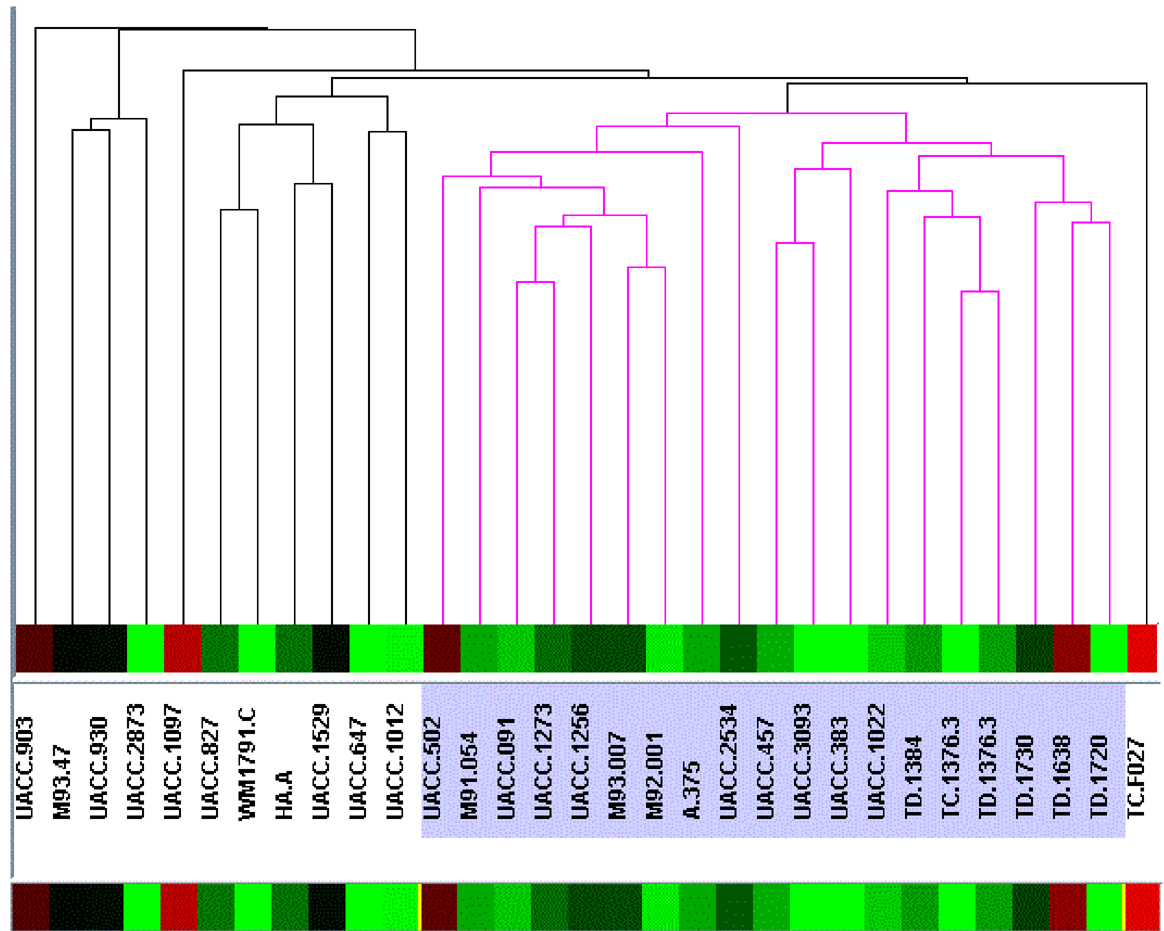

Figure 7. Average Linkage (Kendall's tau)

5.5.2 Consensus on Artificial Dataset

Multiple runs of K-Means algorithm with value of k (number of clusters) ranging from four to six were performed on artificial dataset. The artificial dataset was generated to have six tight clusters by randomly generating values around six well separated points in two-dimensions. As expected when K-Means algorithm was run with $k < 6$, neighboring clusters were merged (fig. 8). Since the merges were random and multiple runs generated merger of different clusters, the consensus clustering correctly identified

the six clusters (fig. 9). Twenty runs of K-Means algorithm were performed and consensus obtained.

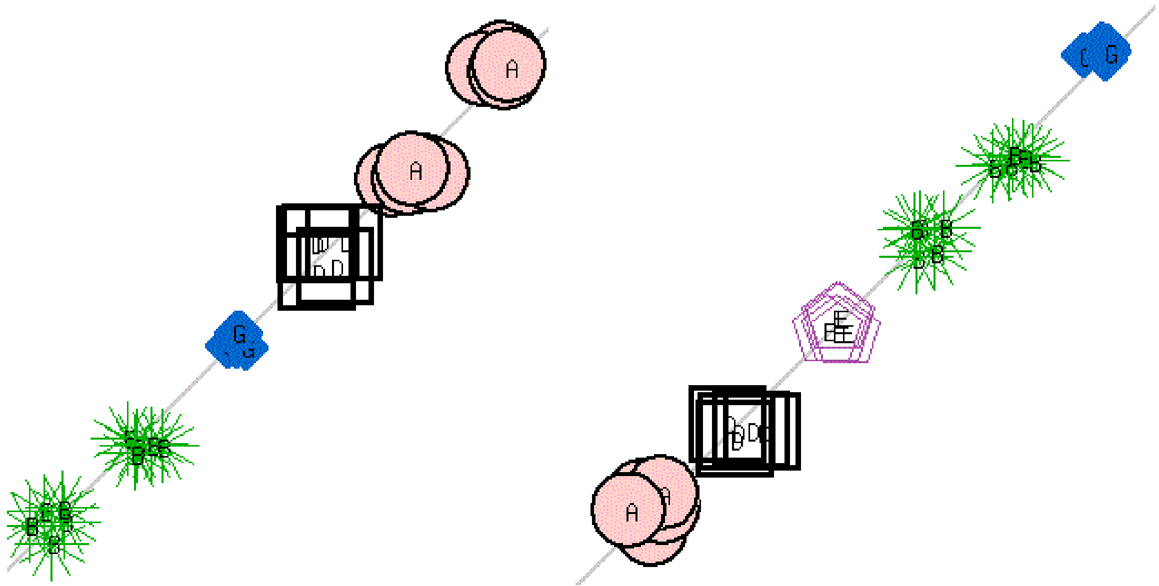

Figure 8. K-Means clusterings (k=4 & k=5)

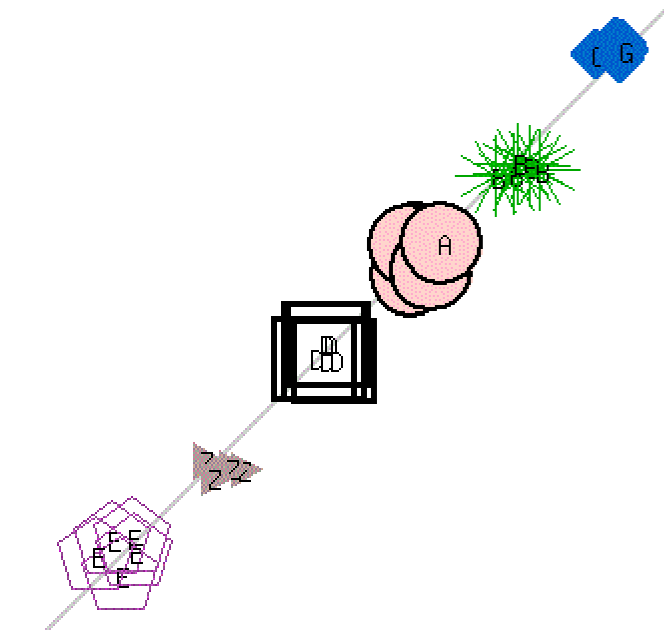

Figure 9. Consensus Clustering

5.5.3 Number of Clusters

The number of clusters was estimated for yeast dataset by generating clusterings with different number of clusters (fig. 10) and applying validation indexes (Hennig 2010). As the figures show, the optimum values are obtained when 4-5 clusters are generated (figs. 11 & 12). This result agrees with the Brown observation (5 clusters).

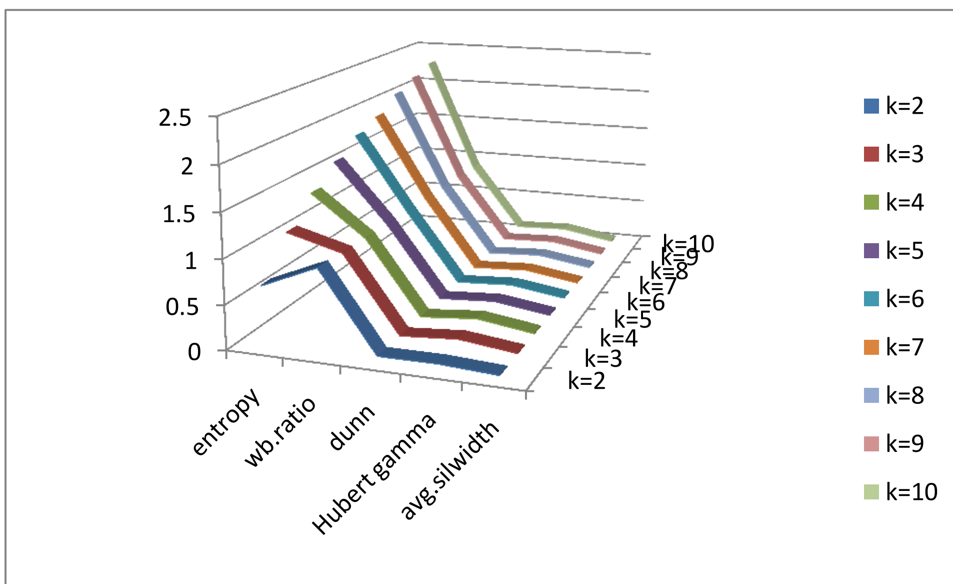

Figure 10. Validation Indexes vs. Number of Clusters

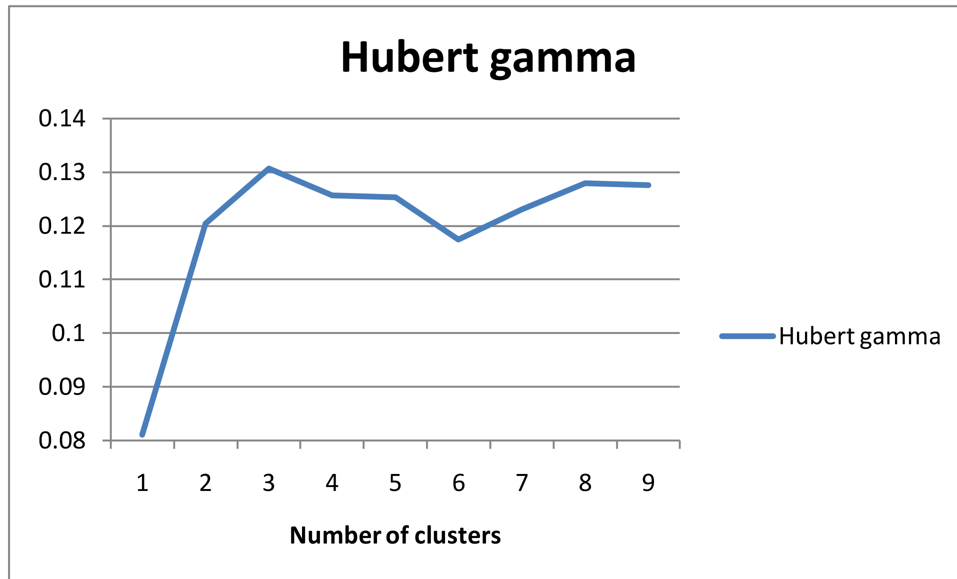

Figure 11. Hubert Gamma Coefficient vs. Number of Clusters

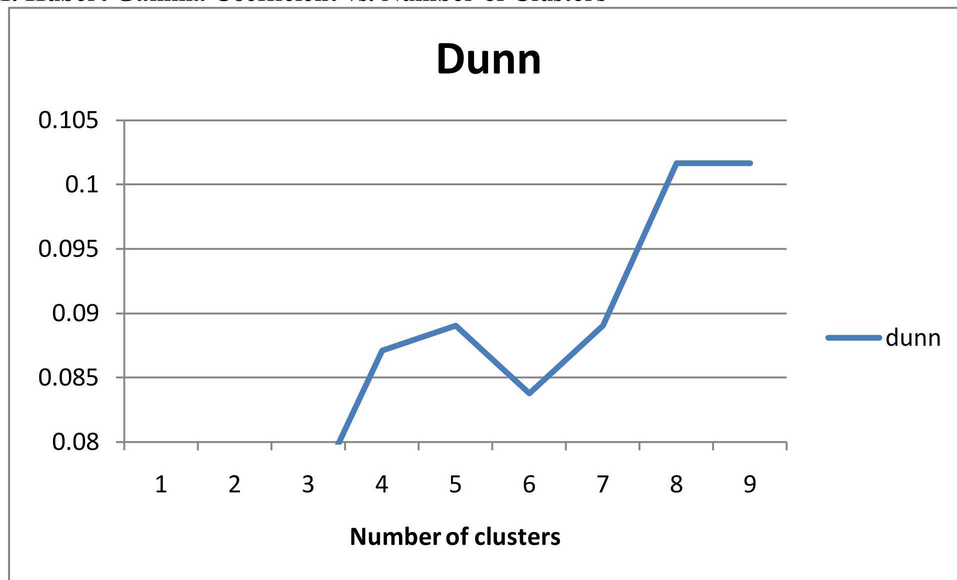

Figure 12. Dunn Index vs. Number of Clusters

Additionally several runs of K-Means algorithm with k=3, 4, 5,6,7,8,9,10 and 11 were used to generate mountain visualizations using gCLUTO (Rasmussen 2004). Even with high values of k, the peaks were identifiable in groups of 3 to 4 (fig. 13). This result was further confirmed when the clusterings were used as input to consensus algorithms.

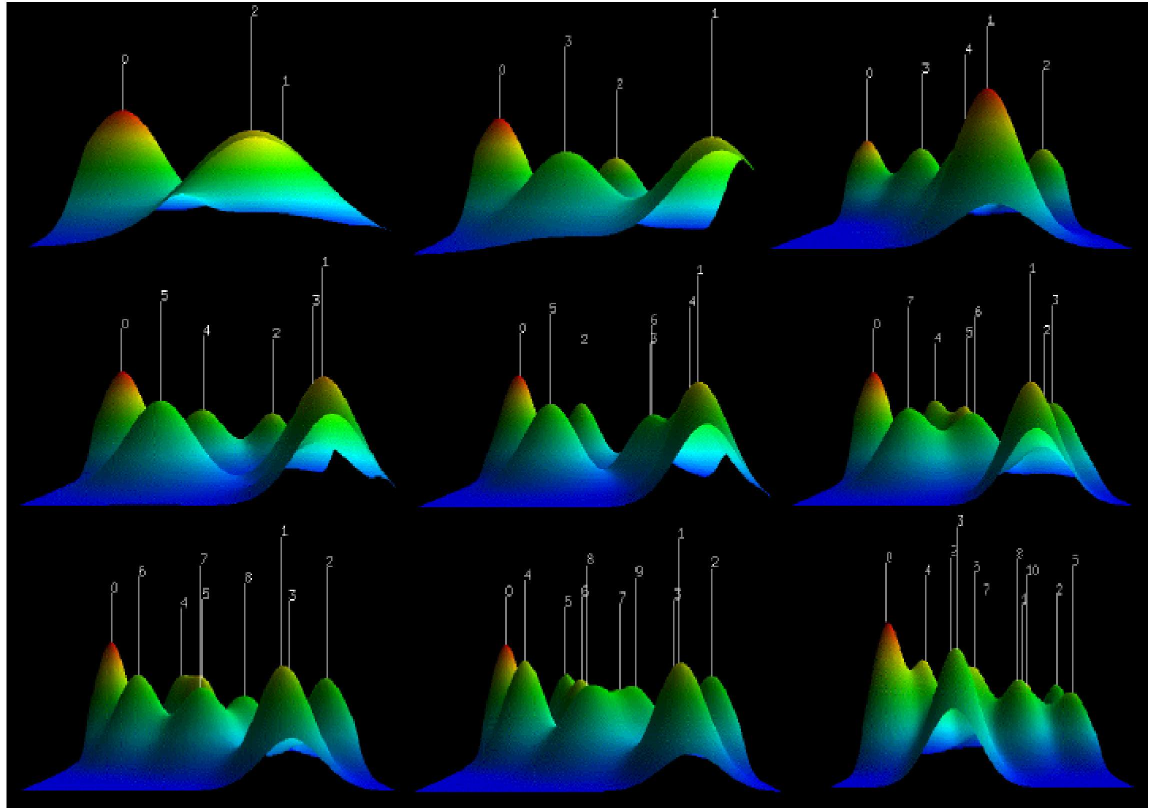

Figure 13. Mountain Visualization (Number of clusters)

5.5.4 Consensus on K-Means clustering (Yeast dataset)

The consensus on K-Means algorithm was performed using fixed as well as varying number of clusters.

5.5.4.1 Analysis 1: Fixed Number of Clusters (k=5)

The K-Means algorithm (Eisen 1998) was run with k=5 (Number of clusters) eight times and the resulting clusterings were used to generate a consensus. Different distance measures used were: Euclidean distance, Manhattan Distance, Uncentered correlation (absolute and standard), Pearson correlation (absolute and standard), Spearman's rank correlation, and Kendall's tau correlation. The analysis of results using

various validation indexes is shown in Table 4. The Best Cluster Algorithm selected clustering that used uncentered correlation (standard). The Best and Local Search algorithms were run on the same set of clusterings. The Local Search algorithm resulted in good entropy and Dunn Index value (figs. 18 & 19). The validation results from consensus algorithm were found to be optimal for Hubert Gamma Statistic and average within/between ratio. The Best algorithm generated results that were almost as good as those from consensus algorithm for Hubert Gamma Statistic and entropy (figs. 16 & 21). This result is not surprising since the input clusterings were from a narrow field of options (all input clusterings had 5 clusters).The weighted consensus algorithm performed best with lowest sum of squares error (SSE), and optimal values for entropy, Dunn's index, average within/between ratio, and average Silhouette Width (fig. 14 & 17). In all cases the results were superior to those from Brown paper.

Table 4. Analysis of Algorithms

	K-Means	Best	Local	Consensus	Consensus_Wt
hubertgamma	0.1165932	0.1414726	0.125488	0.1547022	0.1337955
avg.silwidth	0.0231681 6	0.0286981 6	0.0262698 9	0.0415417 4	0.04154174
entropy	1.567404	1.580375	1.60319	1.336454	1.769071
Dunn	0.0649534 4	0.0781762 9	0.0715259 5	0.0645985	0.07930587
SSE	170853.6	166882.7	168535.5	167983.8	164659.7
Avg. within/bet	0.9212384	0.9039512	0.9131768	0.9046992	0.904024

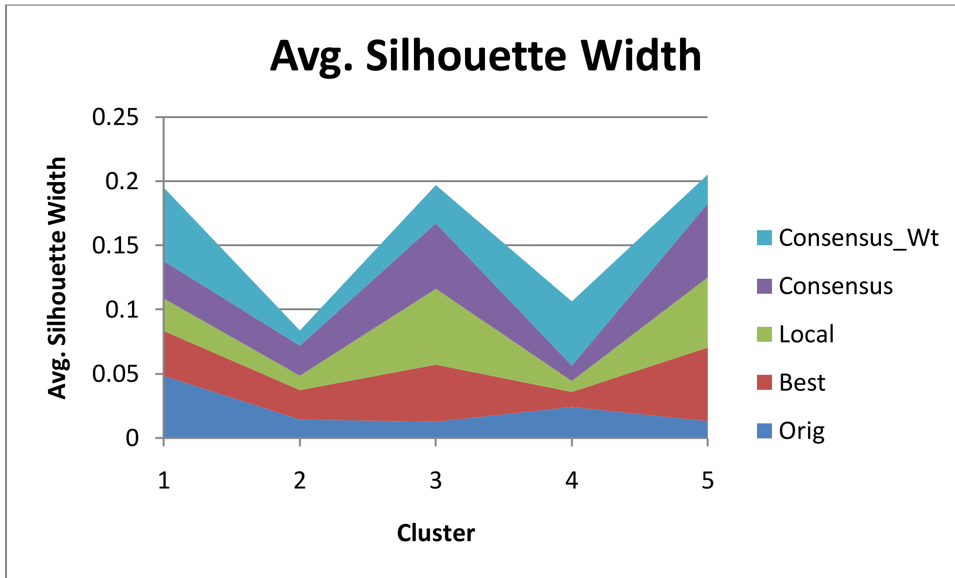

Figure 14. Avg. Silhouette Width of Clusters

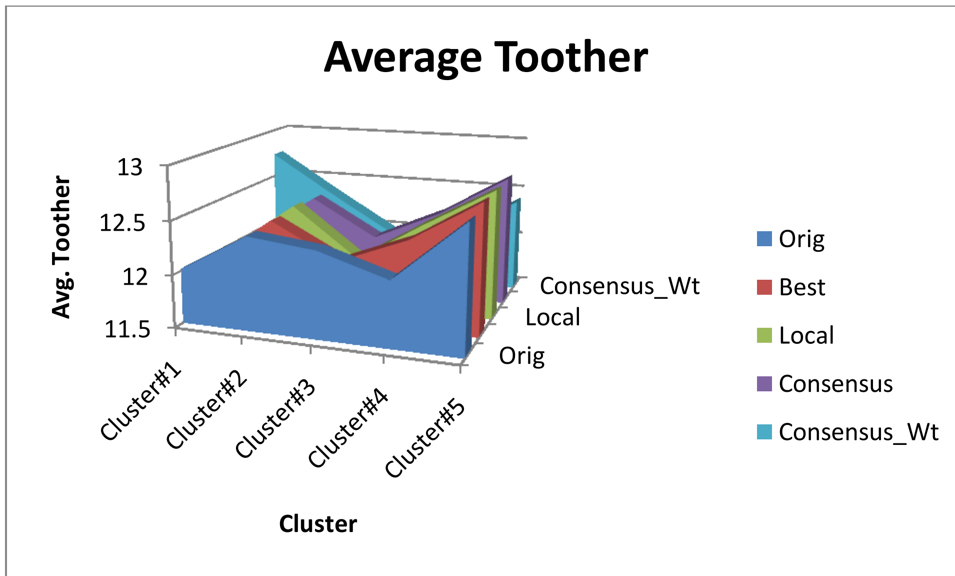

Figure 15. Average Toother of Clusters

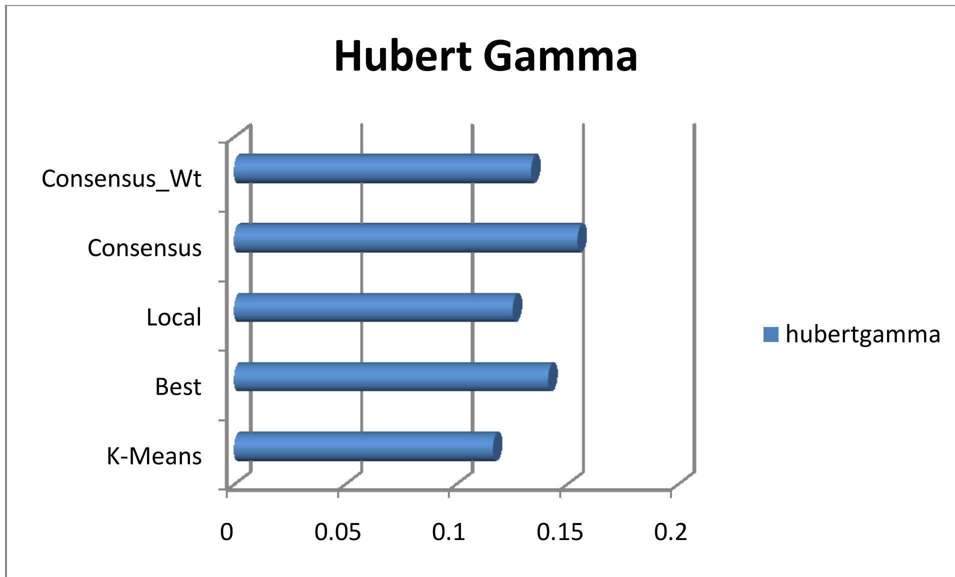

Figure 16. Hubert Gamma Coefficient

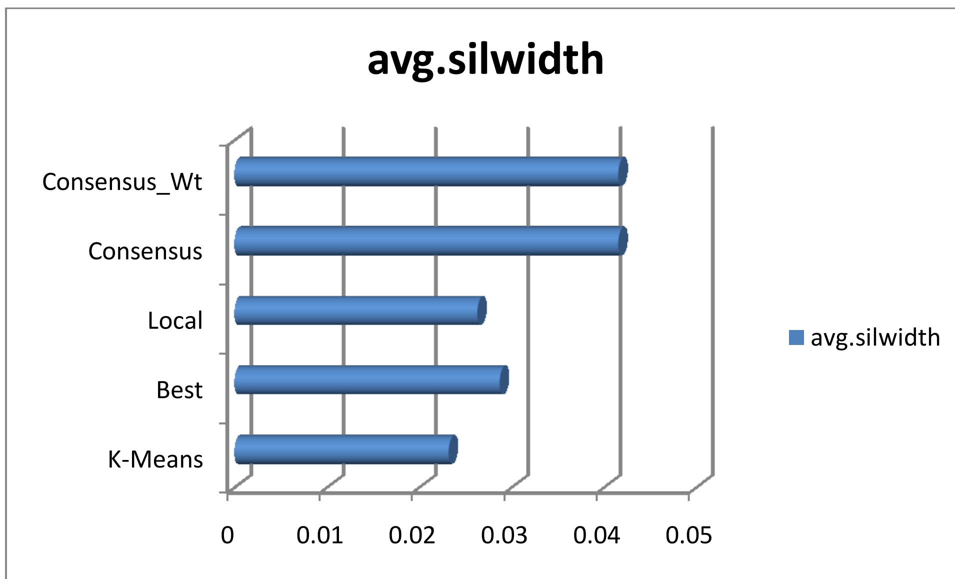

Figure 17. Average Silhouette Width

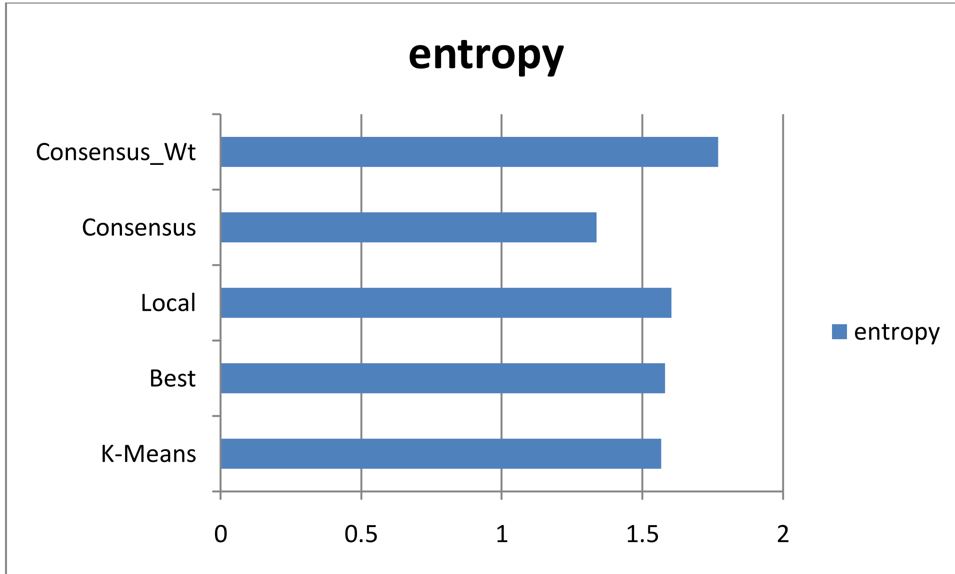

Figure 18. Entropy of distribution into clusters

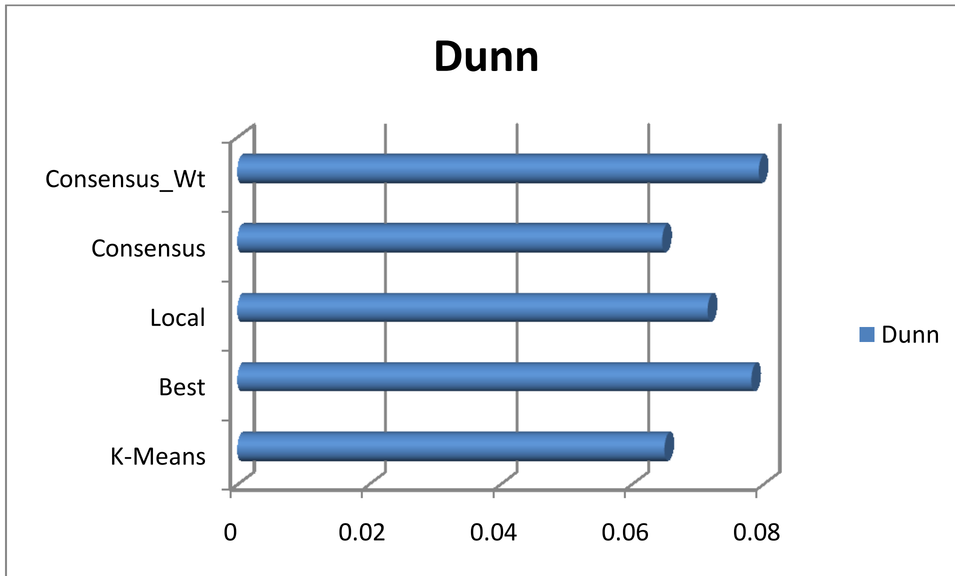

Figure 19. Dunn Index

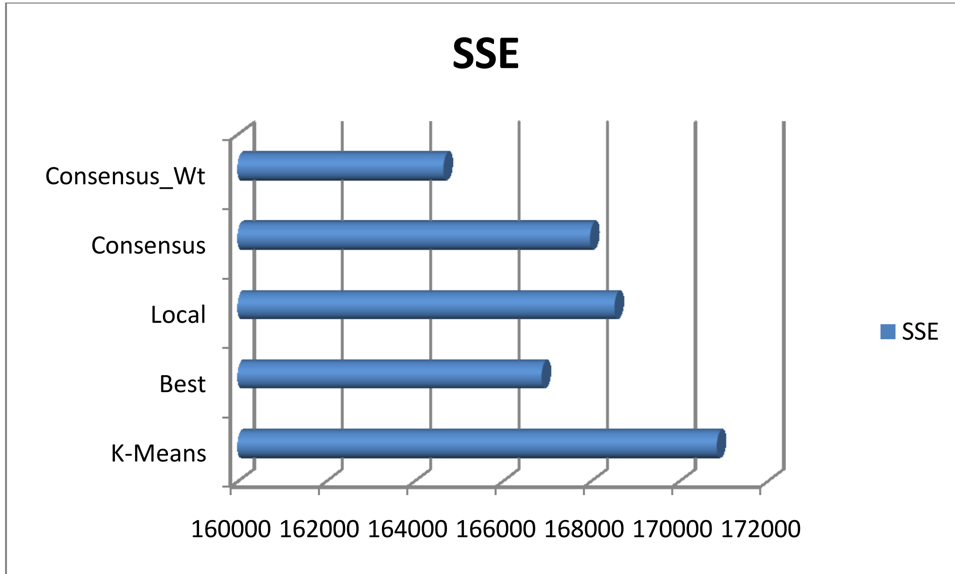

Figure 20. Sum of squares (within-cluster)

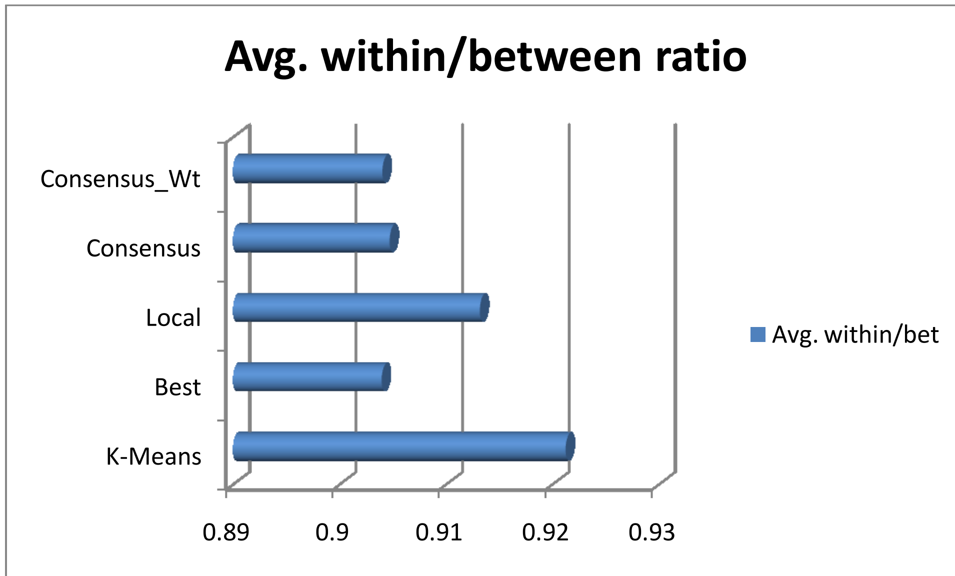

Figure 21. Average with/Average between

5.5.4.2 Analysis 2: Varying Number of Clusters (k=3 to k=10)

The Number of clusters in the yeast dataset was expected to be in the range of 4-5 clusters. 30 iterations of K-Means algorithm were performed with k=3 to k=10. Most runs were in the expected middle range (4-7 clusters). The resultant consensus was found to contain 4 clusters while the Best and Local Search yielded 3 cluster solutions. The validity indexes for consensus algorithm result were consistently superior (figs. 22 & 23). The entropy is not invariant of the number of clusters. The five cluster K-Means algorithm yielded higher entropy as expected.

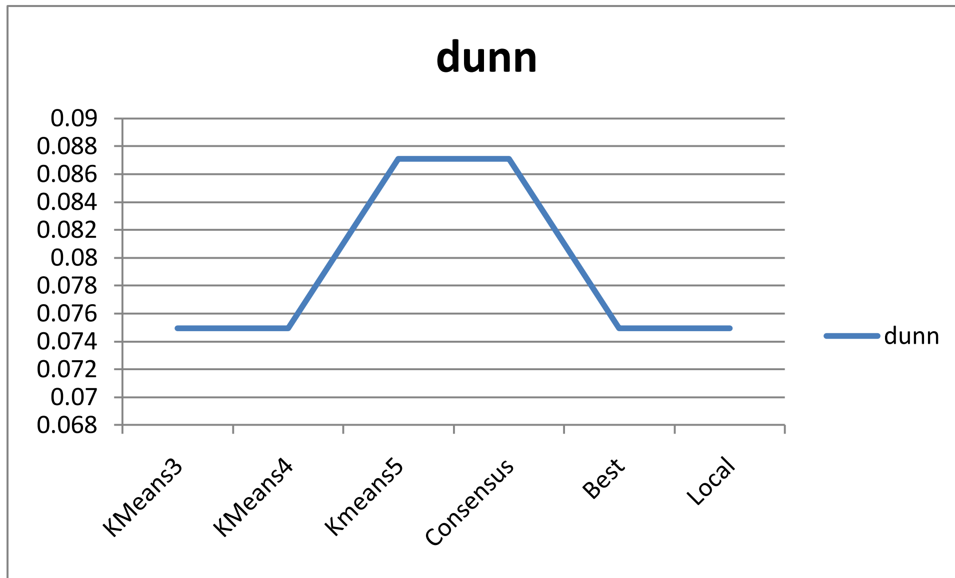

Figure 22. Dunn Index With Varying Clusters

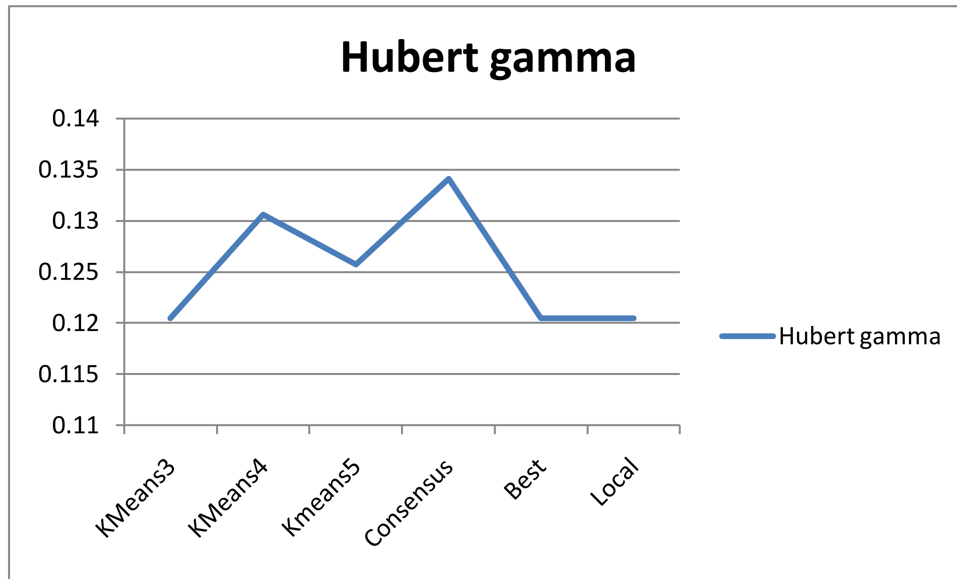

Figure 23. Hubert Gamma Coefficient With Varying Clusters

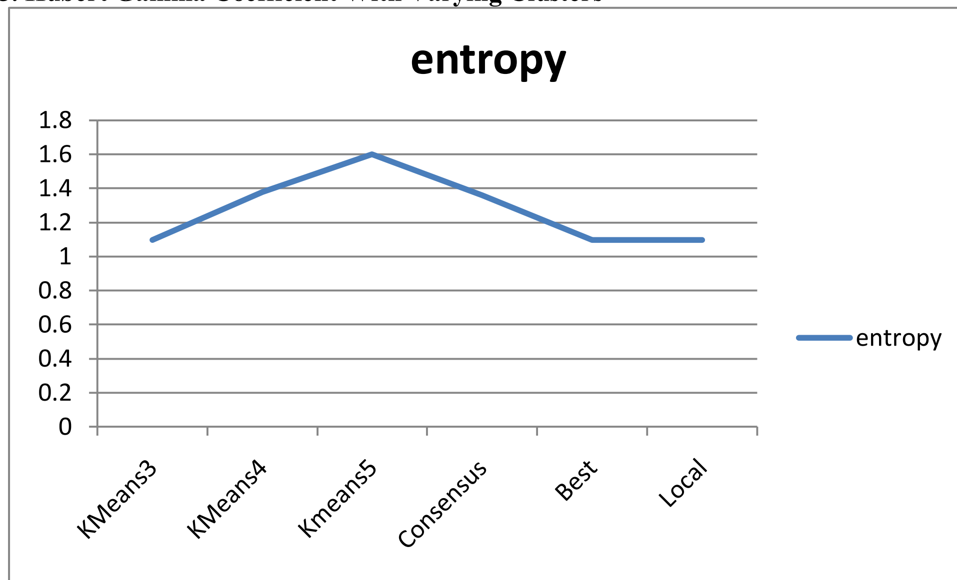

Figure 24. Entropy of Cluster Distribution with Varying Clusters

5.6 Conclusion and Future Work

We compared the results of applying consensus clustering on yeast and melanoma datasets against the results obtained from base algorithms (K-Means and Hierarchical algorithms). The samples and genes were clustered. The size of datasets can be

considered as medium size but the algorithms can scale to large sized dataset. The number of clusters were estimated to be less than 10 using Dunn's Index and Hubert Gamma coefficient. A consensus matrix using dissimilarity was used, and the input was treated as data for consensus clustering. This allowed generation of tight clusters without large number of outliers. Using several performance criteria the results obtained using consensus algorithm was shown to be an improvement. Results obtained from consensus clustering are consistent and more accurate than results from base algorithms. The consensus algorithm can identify the number of clusters and detect outliers. The consensus clustering results provide a high level of confidence in the results. The results were compared to (Eisen, Spellman, Brown, & Botstein, 1998) and Bittner (2000) and better quality of results were found to be obtained using consensus clustering.

APPENDIX A: SOURCE CODE

KMeans.java

```
/**

   COPYRIGHT (C) 2010 Sarbinder Kallar. All Rights Reserved.

   K Means Algorithm implementation.

   @author skallar

   @version 1 2010/04/22

*/
import java.io.*;

/*java -Xmx1024m  KMeans*/
import java.util.*;


/**

K Means algorithm.
*/
class KMeans {
    DataPoint[] dataMatrix;
    int numClust;
    int numVars;
    int bestClust;
    int bestDist;
    int cumulativeDist;
    int maxClust = 0;
    int nextAvailableClusterNo;
    ArrayList<ArrayList> CNode2Array;
    ArrayList CNode2List;
    int mrgNode1;
    int mrgNode2;
    Cluster[] clustering;
    float[][] distMatrix;
    double epsilon = 0.01;
    double gamma = 105.51;

    /**

       Reads file, creates and populates a matrix of Datapoints.

       @param filename name of input file

       Note: Input parm must specify fully qualified domain name.
             Otherwise file is assumed to be in current directory.

    */
    public void readFile(String filename) {
        try {
            BufferedReader inbuf = new BufferedReader(new FileReader(filename));
            int eof = 0;

            String strLine;
```

```
            try {

                numVars = Integer.parseInt(inbuf.readLine());
            } catch (NumberFormatException ne) {
                System.out.println("Illegal number in line:1 ");
            }

            dataMatrix = new DataPoint[numVars];

            for (int i = 0; i < numVars; i++) {
                dataMatrix[i] = new DataPoint();
                strLine = inbuf.readLine();

                StringTokenizer st = new StringTokenizer(strLine);

                while (st.hasMoreTokens()) {
                    dataMatrix[i].point.add(Float.parseFloat(st.nextToken()));
                }
            }


            }
        } catch (IOException ie) {
            System.out.println("I/O Error ");
        }
    }

    /**

       Initialize clusters to trigger start of algorithm.


    */
    public void initCluster() {
        int[] taken = new int[numVars];
        int pos;
        boolean done = false;

        for (int i = 0; i < numVars; i++)
            taken[i] = -1;

        for (int i = 0; i < numClust; i++) {
            done = false;
            clustering[i] = new Cluster();

            while (!done) {
                pos = (int) (Math.random() * numVars);

                if (taken[pos] < 0) {
                    done = true;
                    taken[pos] = 1;
                    clustering[i].centroid = new ArrayList((Collection)
dataMatrix[pos].point);
                    clustering[i].pointsInCluster = 1;
                    dataMatrix[pos].clusterNo = i;
                }
            }
        }
    }

    /**
```

```
        Main driver for algorithm.


     */
     public void runCluster() {
         float maxCentroidShift = Float.POSITIVE_INFINITY;
         float centroidShift = 0;
         float minDist = Float.POSITIVE_INFINITY;
         float maxDist = Float.NEGATIVE_INFINITY;
         int count = 0;
         int minClust;
         int maxClust;

         while ((maxCentroidShift > epsilon) && (count < 1000)) {
             int pos;
             minClust = -1;
             maxClust = -1;
             count++;

             for (int i = 0; i < numVars; i++) {
                 minClust = -1;
                 minDist = Float.POSITIVE_INFINITY;
                 maxClust = -1;
                 maxDist = Float.NEGATIVE_INFINITY;

                 for (int j = 0; j < numClust; j++) {

                     if (dataMatrix[i].distance(clustering[j].centroid) < minDist) {
                         minDist = dataMatrix[i].distance(clustering[j].centroid);
                         minClust = j;
                     }

                     if ((dataMatrix[i].distance(clustering[j].centroid) > maxDist) &&
                             (dataMatrix[i].clusterNo == j)) { //
                         maxDist = dataMatrix[i].distance(clustering[j].centroid);
                         maxClust = j;
                     }
                 }

                 if ((maxDist > gamma) && (dataMatrix[i].clusterNo != -1) &&
                         (clustering[dataMatrix[i].clusterNo].pointsInCluster > 1)) {
                     dataMatrix[i].clusterNo = -1;
                     clustering[maxClust].detach(dataMatrix[i].point);

                 }

                 if ((minClust == dataMatrix[i].clusterNo) || (minClust == -1)) {

                     continue;
                 }

                 if (dataMatrix[i].clusterNo != -1) {
                     clustering[dataMatrix[i].clusterNo].pointsInCluster--;
                 }

                 dataMatrix[i].clusterNo = minClust;
                 centroidShift =
clustering[minClust].updateCentroid(dataMatrix[i].point);

                 if (centroidShift < maxCentroidShift) {
```

```
                maxCentroidShift = centroidShift;
            }
        }
    }
}

/**

   Output formatted result in format needed for visualization of results.

*/
public void printResults() {
    //ArrayList<ArrayList> outArray;
    ArrayList[] dataList = new ArrayList[numClust];

    //DataPoint dataPoint[]=new DataPoint[numVars];
    int maxsz = 0;

    for (int i = 0; i < numClust; i++)
        dataList[i] = new ArrayList();

    for (int i = 0; i < numVars; i++) {
        dataList[dataMatrix[i].clusterNo].add(dataMatrix[i].getString());
    }

    String outln = "data:\t";
    BufferedWriter outbuf = null;

    for (int i = 0; i < numClust; i++) {
        System.out.println(dataList[i]);

        if (dataList[i].size() > maxsz) {
            maxsz = dataList[i].size();
        }
    }

    System.out.println("maxsz" + maxsz);

    try {
        outbuf = new BufferedWriter(new FileWriter("outdata" + numClust));

        System.out.println("data stored in file:outdata" + numClust);

        for (int i = 0; i < maxsz; i++) {
            for (int j = 0; j < numClust; j++) {
                if (dataList[j].size() > i) {

                    outln += dataList[j].get(i);
                } else {
                    outln += "-99\t-99\t";
                }
            }

            outbuf.write(outln);
            outbuf.newLine();
            outln = "";
        }
    } catch (IOException ie) {
        System.out.println("I/O Error ");
    } finally {
        try {
```

```
                if (outbuf != null) {
                    outbuf.flush();
                    outbuf.close();
                }
            } catch (IOException ex) {
            }
        }
    }

    public static void main(String[] args) {
        KMeans a = new KMeans();

        a.numClust = 3; //3

        if (args.length > 0) {
            try {
                a.numClust = Integer.parseInt(args[0]);
            } catch (Exception e) {
                System.err.print("Invalid input:");
                System.err.println(e.getMessage());
                System.out.println("Usage: java KMeans nn");
                System.out.println("       where nn=No. of clusters");
                throw new RuntimeException(e);
            }
        }

        a.clustering = new Cluster[a.numClust];
        a.readFile("datapoint2");
        a.initCluster();
        a.runCluster();

        for (int i = 0; i < a.numVars; i++)
            System.err.print(a.dataMatrix[i]);

        System.err.println();
        a.printResults();
    }
}


/**

DataPoint class represents individual data point to be clustered.
It can handle n dimensions as defined in input.
Assumption: All data points have same number of dimensions.

*/
class DataPoint {
    ArrayList point = new ArrayList();
    int clusterNo = -1;

    public float distance(ArrayList cPoint) {
        float distance = 0;

        for (int i = 0; i < cPoint.size(); i++) {
            distance += ((((Float) cPoint.get(i)).floatValue() -
            ((Float) point.get(i)).floatValue()) * (((Float)
cPoint.get(i)).floatValue() -
            ((Float) point.get(i)).floatValue()));
        }
```

```
        distance = (float) Math.sqrt(distance);

        return distance;
    }

    /**

       String representation of DataPoint

    */
    public String toString() {
        String tmpString = clusterNo + "\t";

        //for (int i=0;i<point.size();i++)
        //   tmpString+=(Float)point.get(i)+",";
        return tmpString;
    }

    /**

        DataPoint formatted for printing.

    */
    public String getString() {
        String tmpString = "";

        for (int i = 0; i < point.size(); i++)
            tmpString += ((Float) point.get(i) + "\t");

        return tmpString;
    }

    /**

        Datapoint comparison.

    */
    public boolean equals(ArrayList cPoint) {
        boolean result = true;

        if (point == cPoint) {
            return true;
        }

        for (int i = 0; i < cPoint.size(); i++) {
            result &= (((Float) cPoint.get(i)).floatValue() == ((Float)
point.get(i)).floatValue());
        }

        return result;
    }
}


/**

   Generic class for cluster.

*/
class Cluster {
    ArrayList centroid = new ArrayList();
```

```
    int pointsInCluster = 0;

    /**

       Adjust inter-cluster distances when cluster is updated by addition.

       @param point Arraylist representing a point represented in n-dimension to be
added to current cluster

    */
    public float updateCentroid(ArrayList point) {
        float shift = 0;

        for (int i = 0; i < centroid.size(); i++) {
            shift += (((-((Float) centroid.get(i)).floatValue() +
            ((Float) point.get(i)).floatValue()) / (pointsInCluster + 1)) * ((-((Float)
centroid.get(i)).floatValue() +
            ((Float) point.get(i)).floatValue()) / (pointsInCluster + 1)));
            centroid.set(i,
                ((((Float) centroid.get(i)).floatValue() * pointsInCluster) +
                ((Float) point.get(i)).floatValue()) / (pointsInCluster + 1));
        }

        pointsInCluster++;

        return (float) Math.sqrt(shift);
    }

    /**

       Adjust inter-cluster distances when cluster is updated by deletion.

       @param point Arraylist representing a point represented in n-dimension to be
detached from current cluster

    */
    public void detach(ArrayList point) {
        for (int i = 0; i < centroid.size(); i++) {
            centroid.set(i,
                ((((Float) centroid.get(i)).floatValue() * (pointsInCluster -
                1)) - ((Float) point.get(i)).floatValue()) / (pointsInCluster));
        }

        pointsInCluster--;

        return;
    }

    /**

       String representation of a cluster


    */
    public String toString() {
        String tmpString = "" + pointsInCluster + ":";

        for (int i = 0; i < centroid.size(); i++)
            tmpString += ((Float) centroid.get(i) + ",");

        return tmpString;
```

```
    }
}
```

BestCluster.java

```
/**

   COPYRIGHT (C) 2010 Sarbinder Kallar. All Rights Reserved.

   Best clustering Algorithm implementation.

   @author skallar

   @version 1 2010/04/22

*/
import java.io.*;

import java.util.*;


/**

   Best cluster algorithm. The algorithm returns the best candiadate from
   set of input clusterings.


*/
class BestCluster {
    int[][] clustMatrix;
    int numClust;
    int numVars;
    int bestClust;
    int bestDist;
    int cumulativeDist;
    int maxClust = 0;

    /**

       Read input file and populate input clustering matrix.

       @param filename name of file to read

    */
    public void readFile(String filename) {
        try {
            BufferedReader inbuf = new BufferedReader(new FileReader(filename));
            int eof = 0;

            String strLine;

            try {
                numClust = Integer.parseInt(inbuf.readLine().trim());
                numVars = Integer.parseInt(inbuf.readLine().trim());
            } catch (NumberFormatException ne) {
                System.out.println("Illegal number in line:1 " +
                    ne.getMessage());
            }

            System.out.println("Num: " + numClust);
```

```
        clustMatrix = new int[numClust][numVars];

        for (int i = 0; i < numClust; i++) {
            strLine = inbuf.readLine();

            StringTokenizer st = new StringTokenizer(strLine);

            for (int j = 0; (j < numVars) && st.hasMoreTokens(); j++) {
                clustMatrix[i][j] = Integer.parseInt(st.nextToken());

                if (clustMatrix[i][j] > maxClust) {
                    maxClust = clustMatrix[i][j];
                }
            }
        }

        for (int i = 0; i < numClust; i++) {
            for (int j = 0; j < numVars; j++) {
                System.out.print(clustMatrix[i][j] + "\t");
            }

            System.out.println();
        }
    } catch (IOException ie) {
        System.out.println("I/O Error ");
    }
}

/**

   No parm invocation of bestClust using current clustering matrix.

*/
public void bestClust() {
    bestClust(clustMatrix);
}

/**

   Calls getBest method to obtain best clustering

   @param inMatrix matrix of input clusterings

*/
public void bestClust(int[][] inMatrix) {
    bestClust = Integer.MAX_VALUE;
    bestDist = Integer.MAX_VALUE;
    cumulativeDist = 0;
    System.out.println("numClust=" + numClust);
    System.out.println("numVars=" + numVars);

    getBest(inMatrix);
}

/**

   Finds best clustering from set of input clusterings

   @param inMatrix matrix of input clusterings

*/
```

```java
public int getBest(int[][] inMatrix) {
    bestClust = Integer.MAX_VALUE;
    bestDist = Integer.MAX_VALUE;
    cumulativeDist = 0;

    for (int i = 0; i < numClust; i++) {
        cumulativeDist = 0;

        for (int j = 0; j < numClust; j++) {
            cumulativeDist += getDistance(inMatrix, i, j, numVars);
        }

        if (cumulativeDist < bestDist) {
            bestClust = i;
            bestDist = cumulativeDist;
        }
    }

    System.out.println("Bext cluster is: " + bestClust + " with dist=" +
        bestDist);

    for (int j = 0; j < numVars; j++)
        System.out.print(clustMatrix[bestClust][j] + "\t");

    return bestDist;
}

/**

   Finds distance between two clusterings.

   @param inMatrix matrix of input clusterings

   @param clust1 index of first clustering

   @param clust2 index of first clustering

   @param numVars number of variables to be clustered

*/
public int getDistance(int[][] inMatrix, int clust1, int clust2, int numVars) {
    int distance = 0;

    if (clust1 == clust2) {
        return 0;
    }

    for (int i = numVars - 1; i > 0; i--)
        for (int j = 0; j < i; j++) {
            /* If the clusterings disagree add 1 to distance  */
            if (!(((inMatrix[clust1][i] == inMatrix[clust1][j]) &&
                    (inMatrix[clust2][i] == inMatrix[clust2][j])) ||
                    ((inMatrix[clust1][i] != inMatrix[clust1][j]) &&
                    (inMatrix[clust2][i] != inMatrix[clust2][j])))) {
                distance += 1;
            }
        }

    return distance;
}
```

```
/**

   Finds distance between two individual clusterings.

   @param inclust1 first clustering

   @param inclust2 second clustering

   @param numVars number of variables to be clustered

*/
public int getDistance(int[] inclust1, int[] inclust2, int numVars) {
    int distance = 0;

    if (inclust1 == inclust2) {
        return 0;
    }

    for (int i = numVars - 1; i > 0; i--)
        for (int j = 0; j < i; j++) {
            /* If the clusterings disagree add 1 to distance  */
            if (!(((inclust1[i] == inclust1[j]) &&
                    (inclust2[i] == inclust2[j])) ||
                    ((inclust1[i] != inclust1[j]) &&
                    (inclust2[i] != inclust2[j])))) {
                distance += 1;
            }
        }

    return distance;
}

/**

   No parm method to return distance matrix


*/
public float[][] getDistanceMatrix() {
    return getDistanceMatrix(clustMatrix, numClust, numVars);
}

/**

   Returns distance matrix

   1-none of clusters put them together, 0-all clusters put the pairs together

   @param inMatrix input clusterings

   @param numClust number of input clusterings

   @param numVars number of variables to be clustered

*/
public float[][] getDistanceMatrix(int[][] inMatrix, int numClust,
    int numVars) {
    float[][] distMatrix = new float[numVars][numVars];
    int notInSameCluster = 0;

    for (int i = 0; i < (numVars - 1); i++) {
```

```
            for (int j = i + 1; j < numVars; j++) {
                for (int h = 0; h < numClust; h++) {
                    if (inMatrix[h][i] != inMatrix[h][j]) {
                        notInSameCluster += 1;
                    }
                }

                if (notInSameCluster != 0) {
                    distMatrix[i][j] = distMatrix[j][i] = ((float) 1.0 *
notInSameCluster) / numClust;
                } else {
                    distMatrix[i][j] = distMatrix[j][i] = 0;
                }

                notInSameCluster = 0;
            }
        }

        return distMatrix;
    }

    public static void main(String[] args) {
        BestCluster b = new BestCluster();
        b.readFile("input2");
        b.bestClust();

        b.getDistanceMatrix();
    }
}


/**

     Cluster Node represenation

*/
class CNode2 {
    String label;
    int clusterNo;
    int numElements;
    float distance;
    boolean clustered;

    /**

       String representation of cluster node


    */
    public String toString() {
        return new String(clusterNo + "::" + distance);
    }
}
```

LocalSearch.java

```
/**

   COPYRIGHT (C) 2010 Sarbinder Kallar. All Rights Reserved.

   Local Search clustering Algorithm implementation.
```

```
   @author skallar

   @version 1 2010/04/22

*/

/**

   Local Search algorithm. For large datasets the algorithm must be invoked with heap
value half of
   physical memory(using Xmx option).


*/
class LocalSearch {
    int[] perturbedMatrix;
    BestCluster b;
    int curDist;
    int curBestDistance;
    int numClust;
    int numVars;
    int curBest;
    int maxClust = 0;
    boolean better;

    /**

       No-parm explicit constructor.


    */
    public LocalSearch() {
        b = new BestCluster();
        b.readFile("input2");
        numClust = b.numClust;
        numVars = b.numVars;
        curBestDistance = b.getBest(b.clustMatrix);
        curBest = b.bestClust;
        maxClust = b.maxClust;
        perturbedMatrix = new int[numVars];
    }

    /**

       Driver for Local Search. Data perturbation is continued until no further
optimization is possible.


    */
    public void runLocalSearch() {
        getperturbedMatrix(b.clustMatrix, curBest);
        System.out.println("Bettered!!=" + curBestDistance);

        for (int j = 0; j < numVars; j++)
            System.out.print(perturbedMatrix[j] + "\t");
    }

    /**

       At local level perturb a cluster by making best possible local move for a point
```

```
        @param inMatrix array of input clusterings

        @param bestClust current best cluster obtained from Best Cluster algorithm


     */
    public void getperturbedMatrix(int[][] inMatrix, int bestClust) {

        for (int i = 0; i < perturbedMatrix.length; i++) {
            perturbedMatrix[i] = inMatrix[bestClust][i];
        }

        curDist = 0;

        int res_pt = -1;

        for (int j = 0; j < numVars; j++) {
            better = false;
            res_pt = perturbedMatrix[j];

            curDist = 0;

            for (int k = 0; k <= maxClust; k++) {
                if (k == perturbedMatrix[j]) {
                    continue;
                }

                curDist = 0;
                perturbedMatrix[j] = k;

                for (int i = 0; i < numClust; i++)
                 {
                    curDist += b.getDistance(perturbedMatrix, inMatrix[i],
                        numVars);
                }

                if ((curBestDistance - curDist) > (numVars / 20)) {
                    System.out.println(">>CURR>" + curDist + "  >>PrevBest>" +
                        curBestDistance);
                    curBestDistance = curDist;
                    better = true;
                    inMatrix[bestClust] = perturbedMatrix;
                }
            }

            if (!better) {
                perturbedMatrix[j] = res_pt;
            }
        }
    }

    public static void main(String[] args) {
        LocalSearch l = new LocalSearch();

        l.runLocalSearch();

    }
}
```

MUtils.java

```
/**

   COPYRIGHT (C) 2010 Sarbinder Kallar. All Rights Reserved.

   Utility methods used for ranking and weighting

   @author skallar

   @version 1 2010/04/22

*/
public class MUtils {
    static int numOfExperiments;
    static int numOfUniqueClasses;
    int numBinsX;
    int numBinsY;

    public static void main(String[] args) {
        MUtils MUtils = new MUtils();

        double[][] val = {
                { 1, 2, 2, 1, 1, 3, 5, 2, 2, 1 },
                { 2, 1, 1, 2, 2, 3, 2, 1, 1, 2 },
                { 1, 1, 1, 1, 1, 2, 2, 2, 2, 2 }
            };
        double[][] val1 = MUtils.normalize(val);

        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 5; j++) {
                System.out.print("\t" + val1[i][j]);
            }

            System.out.println();
        }

        double[] valu1 = { 1, 0, 1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        double[] valu2 = { 1, 0, 1, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        System.out.println(MUtils.kappa(valu1, valu2));

        double[] val4 = { 0.3, 0.2, 0.2 };
        System.out.println("\nBEFORE: ");

        for (int i = 0; i < 3; i++) {
            System.out.print("\t" + val4[i]);
        }

        System.out.println("\n AFTER: ");
        MUtils.normalize(val4);

        for (int i = 0; i < 3; i++) {
            System.out.print("\t" + val4[i]);
        }

        double[] norm = new double[3];
        norm = MUtils.getNormalizedKappa(val);
        System.out.println("\n NKAP: ");

        for (int i = 0; i < 3; i++) {
            System.out.print("\t" + norm[i]);
```

```
    }
}

/**

   Returns Fleiss Kappa coefficient for multiple set values

   @param value matrix of dataset values

*/

//A generalization of pi coefficient(Fleiss)
public double fKappa(double[][] value) {
    int csize = value[0].length;
    int rsize = value.length;
    int numCases = (csize * (csize - 1)) / 2;
    double kappa = 0;
    double curVal = 0;
    double[] chancePr = new double[rsize];
    double[] curRow = new double[csize];
    int[] sameClust = new int[numCases];
    int[] difClust = new int[numCases];
    normalize(value);

    for (int i = 0; i < numCases; i++) {
        sameClust[i] = 0;
        difClust[i] = 0;
    }

    double agreementPr = 0;
    int agreeCount = 0;
    int l = 0;

    for (int i = 0; i < rsize; i++) {
        l = 0;
        curRow = value[i];

        for (int j = 0; j < csize; j++) {
            curVal = curRow[j];

            for (int k = 0; k < csize; k++) {
                if (k == j) {
                    continue;
                }

                if (curRow[k] == curVal) {
                    (sameClust[l])++;
                } else {
                    (difClust[l])++;
                }

                l++;
            }
        }
    }

    return kappa;
}

/**
```

```
   Reorder class labels so two parttions can be compared

   @param value matrix of dataset values

*/
public double[][] normalize(double[][] value) {
    int csize = value[0].length;
    int rsize = value.length;
    int curClNo = 0;
    double[][] lvalue = value;
    double curVal;
    int[] isReplaced = new int[csize];
    double[] curClustering = new double[csize];

    for (int i = 0; i < rsize; i++) {
        for (int j = 0; j < csize; j++)
            isReplaced[j] = 0;

        curClNo = 0;

        for (int j = 0; j < csize; j++) {
            if (isReplaced[j] != 0) {
                continue;
            }

            curVal = value[i][j];

            for (int k = j; k < csize; k++) {
                if (value[i][k] == curVal) {
                    if (isReplaced[k] != 0) {
                        continue;
                    }

                    lvalue[i][k] = curClNo;
                    isReplaced[k] = 1;
                }
            }

            curClNo++;
        }
    }

    return lvalue;
}

/**

   Returns normalied Kappa coefficient for each set value

   @param value matrix of dataset values

*/
public double[] getNormalizedKappa(final double[][] cvalue) {
    int numParts = cvalue.length;
    double[] normalizedKappa = new double[numParts];

    for (int i = 0; i < numParts; i++) {
        normalizedKappa[i] = 0;

        for (int j = 0; j < numParts; j++) {
            if (i == j) {
```

```
                continue;
            }

            normalizedKappa[i] += kappa(cvalue[i], cvalue[j]);
        }
    }

    normalize(normalizedKappa);

    return normalizedKappa;
}

/**

   Returns normalied Kappa coefficient for each set value

   @param value matrix of dataset values

*/
public double[] getNormalizedKappa(final int[][] cvalue) {
    int numParts = cvalue.length;
    int numElements = cvalue[0].length;
    double[][] dvalue = new double[numParts][numElements];

    for (int i = 0; i < numParts; i++) {
        for (int j = 0; j < numParts; j++) {
            dvalue[i][j] = (double) cvalue[i][j];
        }
    }

    double[] normalizedKappa = new double[numParts];

    for (int i = 0; i < numParts; i++) {
        normalizedKappa[i] = 0;

        for (int j = 0; j < numParts; j++) {
            if (i == j) {
                continue;
            }

            normalizedKappa[i] += kappa(dvalue[i], dvalue[j]);
        }
    }

    normalize(normalizedKappa);

    return normalizedKappa;
}

/**

   Normalize Kappa coefficient. Also reset negative values to 0.

   @param value matrix of dataset values

*/
public void normalize(double[] value) {
    int rsize = value.length;

    double sumOfRows = 0;
```

```
        for (int i = 0; i < rsize; i++) {
            if (value[i] > 0) {
                sumOfRows += value[i];
            } else {
                value[i] = 0;
                System.out.println("negative kappa = " + i);
            }
        }

        for (int i = 0; i < rsize; i++) {
            value[i] = value[i] / sumOfRows;
        }
    }

    /**

       Returns Kappa coefficient between two partitions

       interpretation of kappa. <0 poor;
                                <0.2 slight;
                                                            <0.4 fair;
                                                            <0.6 significant;
                                                            else substantial

       @param value1 matrix of first dataset values

       @param value2 matrix of second dataset values

    */
    public double kappa(final double[] value1, final double[] value2) {
        double[][] simMatrix1 = getSimMatrix(value1);
        double[][] simMatrix2 = getSimMatrix(value2);

        int csize = value1.length;
        double chancePr = 0;
        double chanceP1 = 0;
        double chanceP2 = 0;
        double agreementPr = 0;

        double count1 = countIn(simMatrix1);
        double count2 = countIn(simMatrix2);

        chanceP1 = count1 / (csize * (csize - 1));
        chanceP2 = count2 / (csize * (csize - 1));
        chancePr = (chanceP1 * chanceP2) + ((1 - chanceP1) * (1 - chanceP2));
        agreementPr = (1.0 * agreementCount(simMatrix1, simMatrix2)) / (csize * (csize
-
            1));

        return (agreementPr - chancePr) / (1 - chancePr);
    }

    /**

       Count number of agreements between two partitions

       @param value1 matrix of first dataset values

       @param value2 matrix of second dataset values
```

```
*/
public int agreementCount(double[][] value1, double[][] value2) {
    int csize = value1[0].length;
    int rsize = value1.length;
    int curCount = 0;

    for (int i = 0; i < rsize; i++)
        for (int j = 0; j < csize; j++) {
            if ((i != j) && (value1[i][j] == value2[i][j])) {
                curCount++;
            }
        }

    //System.out.println("\tcur:"+curCount);
    return curCount;
}

/**

   Print matrix in rectangular format

   @param value1 matrix of dataset values

*/
public void printMatrix(double[][] value1) {
    int csize = value1[0].length;
    int rsize = value1.length;
    System.out.println();

    for (int i = 0; i < rsize; i++) {
        for (int j = 0; j < csize; j++) {
            System.out.print("\t" + value1[i][j]);
        }

        System.out.println();
    }
}

/**

   Returns number of points clustered together

   @param value matrix of dataset values

*/
public double countIn(double[][] value) {
    int csize = value[0].length;
    int rsize = value.length;
    double curCount = 0;

    for (int i = 0; i < rsize; i++)
        for (int j = 0; j < csize; j++) {
            curCount = curCount + value[i][j];
        }

    return curCount;
}

/**

   Returns Similarity Matrix for pairs of points from partition
```

```
        @param value matrix of dataset values

     */
     public double[][] getSimMatrix(final double[] value1) {
         int size = value1.length;
         double[][] retval = new double[size][size];

         for (int i = 0; i < size; i++)
             for (int j = 0; j < size; j++) {
                 if ((i != j) && (value1[i] == value1[j])) {
                     retval[i][j] = 1;
                 }
             }

         return retval;
     }
}
```

Rnd.java

```
/**

   COPYRIGHT (C) 2010 Sarbinder Kallar. All Rights Reserved.

   Utility class to generate random numbers

   @author skallar

   @version 1 2010/04/22

*/
import java.util.Random;


/**

   Generates randome points for clustering algorithms.main method invocation.

*/
public class Rnd {
    public static void main(String[] args) {
        int limit = 0;

        if (args.length > 0) {
            try {
                limit = Integer.parseInt(args[0]);
            } catch (Exception e) {
                System.err.print("Invalid input:");
                System.err.println(e.getMessage());
                System.out.println("Usage: java Rnd nn");
                System.out.println("       where nn=No. of values");
                throw new RuntimeException(e);
            }
        }

        System.out.println(limit);

        Random r = new Random();
```

```
        for (int i = 0; i < limit; i++) {
            System.out.println(Math.abs(r.nextGaussian() % 1 * 12) + " " +
                Math.abs(r.nextGaussian() % 1 * 12));
        }
    }
}
```

GreedySearch.java

```
/**

   COPYRIGHT (C) 2010 Sarbinder Kallar. All Rights Reserved.

   Greedy Search clustering

   @author skallar

   @version 1 2010/04/22

*/
class GreedySearch {
    int[] perturbedMatrix;
    BestCluster b;
    int curDist;
    int curBestDistance;
    int numClust;
    int numVars;
    int curBest;
    int maxClust = 0;
    boolean better;

    /**

       Explicit no arg constructor

    */
    public GreedySearch() {
        b = new BestCluster();
        b.readFile("input");
        numClust = b.numClust;
        numVars = b.numVars;
        curBestDistance = b.getBest(b.clustMatrix);
        curBest = b.bestClust;
        maxClust = b.maxClust;
        perturbedMatrix = new int[numVars];
    }

    /**

       Driver for greedy search algorithm

    */
    public void runGreedySearch() {
        getperturbedMatrix(b.clustMatrix, curBest);

        System.out.println("Bettered!!=" + curBestDistance);

        for (int j = 0; j < numVars; j++) {
            System.out.print(perturbedMatrix[j] + "\t");
```

```java
        }
    }

    /**

       Perturb original matrix and make best one element moves

       @param inMatrix input clusterings

       @param bestClust best clustering from input

    */
    public void getperturbedMatrix(int[][] inMatrix, int bestClust) {
        for (int i = 0; i < perturbedMatrix.length; i++) {
            perturbedMatrix[i] = inMatrix[bestClust][i];
        }

        curDist = 0;

        int res_pt = -1;

        for (int j = 0; j < numVars; j++) {
            better = false;
            res_pt = perturbedMatrix[j];

            curDist = 0;

            for (int k = 0; (k <= maxClust) && !better; k++) {
                if (k == perturbedMatrix[j]) {
                    continue;
                }

                curDist = 0;
                perturbedMatrix[j] = k;

                for (int i = 0; i < numClust; i++) {
                    curDist += b.getDistance(perturbedMatrix, inMatrix[i],
                        numVars);
                }

                if (curDist < curBestDistance) {
                    //              System.out.println(">>CURR>" + curDist + "
>>PrevBest>" + curBestDistance);
                    curBestDistance = curDist;
                    better = true;
                    inMatrix[bestClust] = perturbedMatrix;
                }
            }

            if (!better) {
                perturbedMatrix[j] = res_pt;
            }
        }
    }

    public static void main(String[] args) {
        GreedySearch l = new GreedySearch();

        l.runGreedySearch();
    }
}
```

Agglomerative.java

```
import java.io.*;

/**

   COPYRIGHT (C) 2010 Sarbinder Kallar. All Rights Reserved.

   Agglomerative consensus clustering algorithm

   @author skallar

   @version 1 2010/04/22

*/
import java.util.*;


/**

   Consensus clustering algorithm. The algorithm returns the consensus of values
   from input clusterings.


*/
public class Agglomerative {
    static final double pluralityRatio = 0.5; //0.55;
    int[][] clustMatrix;
    int numClust;
    int numClustFound = 0;
    int numVars;
    int bestClust;
    int bestDist;
    int cumulativeDist;
    int maxClust = 0;
    int nextAvailableClusterNo;
    boolean isDemo = false;
    //CNode2 [][] CNode2Array;
    ArrayList<ArrayList> CNode2Array;
    ArrayList CNode2List;
    double[] penalMatrix;
    int mrgNode1;
    int mrgNode2;
    int counter;
    float[][] distMatrix;
    int[] isClustered;
    double[] distToCluster;
    int[] itemsInCluster;
    int[] isClusteredOrder;

    /**

       Read input file and populate input clustering matrix.

       @param filename name of file to read

    */
    public void readFile(String filename) {
```

```
    try {
        BufferedReader inbuf = new BufferedReader(new FileReader(filename));
        int eof = 0;

        String strLine;

        try {
            numClust = Integer.parseInt(inbuf.readLine().trim());
            numVars = Integer.parseInt(inbuf.readLine().trim());
        } catch (NumberFormatException ne) {
            System.out.println("Illegal number in line:1 " +
                ne.getMessage());
        }

        System.out.println("Num: " + numClust);
        clustMatrix = new int[numClust][numVars];
        itemsInCluster = new int[numVars];
        distToCluster = new double[numVars];

        for (int i = 0; i < numClust; i++) {
            strLine = inbuf.readLine();

            StringTokenizer st = new StringTokenizer(strLine);

            for (int j = 0; (j < numVars) && st.hasMoreTokens(); j++) {
                clustMatrix[i][j] = Integer.parseInt(st.nextToken());

                if (clustMatrix[i][j] > maxClust) {
                    maxClust = clustMatrix[i][j];
                }
            }
        }

    } catch (IOException ie) {
        System.out.println("I/O Error ");
    }
}

/**

   No parm method to get distance matrix


*/
public float[][] getDistanceMatrix() {
    return getDistanceMatrix(clustMatrix, numClust, numVars);
}

/**

   Returns distance matrix from input clusterings

   @param inMatrix input clusterings

   @param numClust number of input clusterings

   @param numVars number of variables to be clustered

*/
public float[][] getDistanceMatrix(int[][] inMatrix, int numClust,
    int numVars) {
```

```
        distMatrix = new float[numVars][numVars];

        float notInSameCluster = 0;
        System.out.println("Penal Matrix");

        for (int h = 0; h < numClust; h++)
            System.out.print(penalMatrix[h] + "\t");

        System.out.println("nUmCLust" + numClust);

        for (int i = 0; i < (numVars - 1); i++) {
            for (int j = i + 1; j < numVars; j++) {
                for (int h = 0; h < numClust; h++) {
                    if (inMatrix[h][i] != inMatrix[h][j]) {
                        notInSameCluster += (numClust * penalMatrix[h]); // 1
                    }
                }

                if (notInSameCluster != 0) {
                    distMatrix[i][j] = distMatrix[j][i] = ((float) 1.0 *
notInSameCluster) / numClust;
                } else {
                    distMatrix[i][j] = distMatrix[j][i] = 0;
                }

                notInSameCluster = 0;
            }
        }

        return distMatrix;
    }

    /**

       Initialize cluster node


    */
    public void initCNode2() {
        CNode2Array = new ArrayList<ArrayList>();
        isClustered = new int[numVars];

        for (int i = 0; i < numVars; i++) {
            ArrayList<CNode2> CNode2List = new ArrayList<CNode2>();

            for (int j = 0; j < numVars; j++) {
                CNode2 CNode2 = new CNode2();
                CNode2.label = "" + j; // label file
                CNode2.clusterNo = -1; // begin as outlier
                CNode2.distance = distMatrix[i][j];
                CNode2.clustered = false;
                CNode2.numElements = 1;
                CNode2List.add(CNode2);
                isClustered[i] = -1;
            }

            CNode2Array.add(CNode2List);
        }
    }

    /**
```

```
   Main driver to build consensus clustering


*/
public void agglomerate() {
    counter = 0;

    while (findMinPair())
        mergeNodes(mrgNode1, mrgNode2);
}

/**

   Returns true if a pair exists that can minimize consensus distance


*/
public boolean findMinPair() {
    mrgNode1 = -1;
    mrgNode2 = -1;

    float minDistPair = Float.POSITIVE_INFINITY;

    for (int i = 0; i < numVars; i++) {

        for (int j = 0; j < numVars; j++) {
            if ((j == i) ||
                    ((isClustered[j] != -1) &&
                    (isClustered[j] == isClustered[i]))) {

                continue;
            }


            if ((((distMatrix[i][j] + distToCluster[j]) < pluralityRatio) ||
                    ((distMatrix[i][j] + distToCluster[i]) < pluralityRatio)) &&
                    (distMatrix[i][j] < minDistPair))
             {
                minDistPair = distMatrix[i][j];

                if (isClustered[j] == -1) {
                    mrgNode1 = i;
                    mrgNode2 = j;
                } else {
                    mrgNode1 = j;
                    mrgNode2 = i;
                }
            }
        }
    }

    System.out.print("FOUND:" + mrgNode1 + ":" + mrgNode2 + ":\t");

    if (mrgNode1 >= 0) {
        return true;
    }

    return false;
}
```

```
    /**

       Merge a pair of nodes(called if consensus criterion satisfied)

       @param tgt target cluster

       @param src source cluster

    */
    public void mergeNodes(int tgt, int src) {
        if ((isClustered[tgt] != -1) && (isClustered[src] != -1)) {
            collapseNodes(tgt, src);
        }

        if (isClustered[tgt] == -1) {
            isClustered[tgt] = nextAvailableClusterNo;
            itemsInCluster[nextAvailableClusterNo]++;
            nextAvailableClusterNo++;
            numClustFound++;
            distToCluster[tgt] = 0.5 * distMatrix[src][tgt]; //0.5*
        } else if (isClustered[src] == -1) {
            distToCluster[tgt] = ((itemsInCluster[isClustered[tgt]] *
distToCluster[tgt]) +
                distMatrix[src][tgt]) / (itemsInCluster[isClustered[tgt]] + 1);

            for (int i = 0; i < numVars; i++) {
                if (isClustered[i] == isClustered[tgt]) {
                    distToCluster[i] = distToCluster[tgt];
                }
            }
        }

        isClustered[src] = isClustered[tgt];
        itemsInCluster[isClustered[tgt]]++;

        distToCluster[src] = distToCluster[tgt];
    }

    /**

       Merge two clusters and update distances

       @param tgt target cluster

       @param src source cluster

    */
    public void collapseNodes(int tgt, int src) {
        int clusterToCollapse = isClustered[src];
        int itemsInClusterToCollapse = itemsInCluster[isClustered[src]];
        System.out.print("\tcollapse:" + tgt + ":" + src + ":\t");
        distToCluster[tgt] = (0.5 * distMatrix[tgt][src]) +
            (((itemsInCluster[isClustered[tgt]] * distToCluster[tgt]) +
            (itemsInCluster[isClustered[src]] * distToCluster[src]) +
            distMatrix[src][tgt]) / (itemsInCluster[isClustered[tgt]] +
            itemsInCluster[isClustered[src]]));

        for (int i = 0; i < numVars; i++) { // get(0)?

            if (isClustered[i] == clusterToCollapse) {
                isClustered[i] = isClustered[tgt];
```

```
            distToCluster[i] = distToCluster[tgt];

        } else if (isClustered[i] == isClustered[tgt]) {
            distToCluster[i] = distToCluster[tgt];
        }
    }

    numClustFound--;
    itemsInCluster[isClustered[tgt]] += (itemsInCluster[clusterToCollapse] -
    1);
    itemsInCluster[clusterToCollapse] = 0;
}

/**

   Compact cluster numbers to make cluster numbers contiguous

   @param rankArray array for reverse lookup of compacted cluster numbers

*/
public void rank(int[] rankArray) {
    //remove duplicates & sort array. Needed since cluster# may not be continuous
    TreeSet<Integer> tsIsClustered = new TreeSet<Integer>();

    for (int i = 0; i < numVars; i++) {
        //
        if (isClustered[i] >= 0) {
            tsIsClustered.add(isClustered[i]);
        }
    }

    //isClusteredOrder = new int[tsIsClustered.size()];
    int i = 0;
    Iterator<Integer> tsIsClusteredItr = tsIsClustered.iterator();

    while (tsIsClusteredItr.hasNext() && (i < rankArray.length)) {
        rankArray[tsIsClusteredItr.next().intValue()] = i;
        i++;
    }
}

/**

   Print results in required format


*/
public void printResults() {
    numClust = numClustFound;
    System.out.println("itemsInCluster:" + Arrays.toString(itemsInCluster));

    ArrayList[] dataList = new ArrayList[numClust];
    int[] clustRank = new int[nextAvailableClusterNo];
    rank(clustRank);

    DataPoints[] DataPoints = new DataPoints[numVars];
    BufferedReader inbuf = null;

    //initialize
    for (int i = 0; i < numClust; i++)
        dataList[i] = new ArrayList();
```

```
        CNode2List = CNode2Array.get(0);

        for (int j = 0; j < CNode2List.size(); j++) {
            System.out.println("Cl#" + ((CNode2) CNode2List.get(j)).clusterNo);

            if (((CNode2) CNode2List.get(j)).clusterNo != -1) {
                dataList[clustRank[((CNode2)
CNode2List.get(j)).clusterNo]].add(DataPoints[j]);
            } else {
                System.out.print("\t outlier");
            }
        }

        //   }
        for (int j = 0; j < CNode2List.size(); j++) {
            System.out.print("\t" + ((CNode2) CNode2List.get(j)).clusterNo);
        }

        System.out.println();

        for (int j = 0; j < CNode2List.size(); j++) {
            if (((CNode2) CNode2List.get(j)).clusterNo != -1) {
                System.out.print("\t" +
                    clustRank[((CNode2) CNode2List.get(j)).clusterNo]);
            } else {
                System.out.print("\t-1");
            }
        }

        System.out.println();

        for (int j = 0; j < CNode2List.size(); j++) {
            System.out.print("\t" + isClustered[j]);
        }

        System.out.println();

        if (isDemo != true) {
            return;
        }

        try {
            inbuf = new BufferedReader(new FileReader("DataPoint2"));
            System.out.println("numVars:" + numVars);

            String strLine;
            StringTokenizer st;
            strLine = inbuf.readLine();

            if (numVars != Integer.parseInt(strLine)) {
                throw new RuntimeException("Internal Inconsistency");
            }

            numVars = Integer.parseInt(strLine.trim());

            for (int j = 0; j < numVars; j++) {
                //
                DataPoints[j] = new DataPoints();
                strLine = inbuf.readLine();
                st = new StringTokenizer(strLine);
```

```
                DataPoints[j].x = Float.parseFloat(st.nextToken());
                DataPoints[j].y = Float.parseFloat(st.nextToken());
            }
        } catch (IOException ie) {
            System.out.println("I/O Errors ");
        }

        int maxsz = 0;
        String outln = "data:\t";
        BufferedWriter outbuf = null;
        System.out.println("numClust:" + numClust);

        for (int i = 0; i < numClust; i++) {
            System.out.println(dataList[i]);

            if (dataList[i].size() > maxsz) {
                maxsz = dataList[i].size();
            }
        }

        System.out.println("maxsz" + maxsz);

        try {
            outbuf = new BufferedWriter(new FileWriter("outdata"));

            System.out.print("data:\t");

            for (int i = 0; i < maxsz; i++) {
                for (int j = 0; j < numClust; j++) {
                    if (dataList[j].size() > i) {
                        outln += dataList[j].get(i);
                    } else {
                        outln += "-99\t-99\t";
                    }
                }

                outbuf.write(outln);
                outbuf.newLine();
                outln = "";
            }
        } catch (IOException ie) {
            System.out.println("I/O Error ");
        } finally {
            try {
                if (outbuf != null) {
                    outbuf.flush();
                    outbuf.close();
                }
            } catch (IOException ex) {
            }
        }
    }

    public static void main(String[] args) {
        Agglomerative a = new Agglomerative();

        if (args.length > 0) {
            a.isDemo = true;
        }

        a.readFile("input2");
```

```
        a.penalMatrix = new MUtils().getNormalizedKappa(a.clustMatrix);
        ;
        a.getDistanceMatrix();
        a.initCNode2();
        a.agglomerate();
        System.out.println(a.CNode2Array);
        a.printResults();

        System.exit(0);
    }
}
```

BIBLIOGRAPHY

Berkhin, P. (2002). A Survey of Clustering Data Mining Techniques. (J. Kogan, C. Nicholas, & M. Teboulle, Eds.) *Grouping Multidimensional Data* , pp. 1-56.

Bittner, M., Meltzer, P., Chen, Y., Jiang, Y., Seftor, E., Hendrix, M., et al. (2000). Molecular classification of cutaneous malignant melanoma by gene expression profiling. *Nature , 406* (6795), 536-40.

Bolshakova, N., & Azuaje, F. (2002). Cluster Validation Techniques for Genome Expression Data. *Signal Processing , 83* (4), 825-833.

Dunn, J. C. (1974). Well-Separated Clusters and Optimal Fuzzy Partitions. *Cybernetics and Systems , 4* (1), 95-104.

Eisen, M. B. (2010). *Eisen Lab: Maple Tree Cluster*. Retrieved April 12, 2010, from Eisen Lab: Evolution of Gene Expression and Gene Regulation in Flies, Fungi and Beyond: http://rana.lbl.gov/EisenSoftware.htm

Eisen, M. B., Spellman, P. T., Brown, P. O., & Botstein, D. (1998). Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences of the United States of America , 95* (25), Proceedings of the National Academy of Sciences of the United States of America.

Gionis, A., Mannila, H., & Tsaparas, P. (2005). Clustering Aggregation. *In Proceedings of the 21st International Conference on Data Engineering , 1* (1), 341-352.

Halkidi, M., Batistakis, Y., & Vazirgiannis, M. (2002). Clustering validity checking methods: part II. *ACM SIGMOD Record archive , 31* (3), 19-27.

Hennig, C. (n.d.). *fpc: Fixed point clusters, clusterwise regression and discriminant plots*. Retrieved Apr 12, 2010, from The Comprehensive R Archive Network: http://cran.r-project.org/web/packages/fpc/index.html

Jain, A. K., Murty, M. N., & Flynn, P. J. (1999). Data Clustering: A Review. *ACM Computing Surveys , 31* (3), 264-323.

Kerr, M. K., & Churchill, G. A. (2001). Bootstrapping cluster analysis: assessing the reliability of conclusions from microarray experiments. *Proceedings of the National Academy of Sciences , 98* (16), 8961–8965.

Likas, A., Vlassis, N., & Verbeek, J. J. (2001). The Global K-Means Clustering Algorithm. *Pattern Recognition , 36*, 451-461.

McShane, L. M., Radmacher, M. D., Friedlin, B., Yu, R., Li, M. C., & Simon, R. (2002). Methods for assessing reproducibility of clustering patterns observed in analyses of microarray data. *Bioinformatics , 18* (11), 1462-1469.

Meila, M. (2007). Comparing clusterings—an information based distance. *Journal of Multivariate Analysis , 98* (5), 873-895.

Milligan, G. (1981). A Review of Monte Carlo Tests of Cluster Analysis. *Multivariate Behavioral Research , 16*, 379–407.

Mirkin, B. (2005). *Clustering for data mining : a data recovery approach.* Boca Raton, Florida: Chapman & Hall/CRC Computer Science & Data Analysis.

Moreira, J. E., Midkiff, S. P., Gupta, M., Artigas, P. V., Snir, M., & Lawrence, R. D. (2000). Java programming for high-performance numerical computing. *IBM Systems Journal , 39* (1), 21-56.

Rasmussen, M., & Karypis, G. (2004). gcluto: An interactive clustering, visualization and analysis system. *CSE/UMN Technical Report , 04* (21).

Tan, P., Steinbach, M., & Kumar, V. (2006). *Introduction to Data Mining.* Reading: Addison-Wesley.
Tibshirani, R., Walther, G., & Hastie, T. (2001). Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology) , 63* (2), 411-423.

Viera, A. J., & Garrett, J. M. (2005). Understanding Interobserver Agreement: The Kappa Statistic. *Family Medicine , 37* (5), 360-363.

Yeung, K. Y., Haynor, D. R., & Ruzzo, W. L. (2001). Validating clustering for gene expression data. *Bioinformatics , 17* (4), 309-318.

Zuylen, A. V., & Williamson, D. P. (2008, February 09). Deterministic Algorithms for Rank Aggregation and Other Ranking and Clustering Problems. (C. Kaklamanis, & M. Skutella, Eds.) *Approximation and Online Algorithms* , pp. 260-273.