CLUSTERING

Team Members

Anjaly Mehla

Antra Grover

Damini Singal

Jyoti Lakra

INTRODUCTION TO CLUSTERING

What is clustering

Applications

Defining the dataset

Various Clustering strategies

CLUSTERING

- **Clustering** is the classification of objects into different groups, or more precisely, the partitioning of a data set into subsets (clusters), so that the data in each subset (ideally) share some common trait - often according to some defined distance measure.

- Dealing with real life situations
  - Very large data
  - High dimensional spaces

APPLICATIONS

- Social network analysis
- Search result grouping
- Flickr's slippy map organization
- Recommendation systems
- Image segmentation and object recognition
- Human genetic clustering
- Data Mining
- Market Research and analysis

DEFINING THE DATASET

- Points
  - Objects associated with a feature vector, each feature as a vector component
- Spaces
  - A universal set of points from which the points in the dataset are drawn
  - Euclidean(flat) and non-Euclidean(non-flat spaces)
- Distances
  - All spaces for which we can perform clustering MUST have a distance measure
  - Euclidean space: Eucledian, Manhattan, L-infinity
  - Non Euclidean space: Jaccard, cosine, hamming, edit

CLUSTERING STRATEGIES

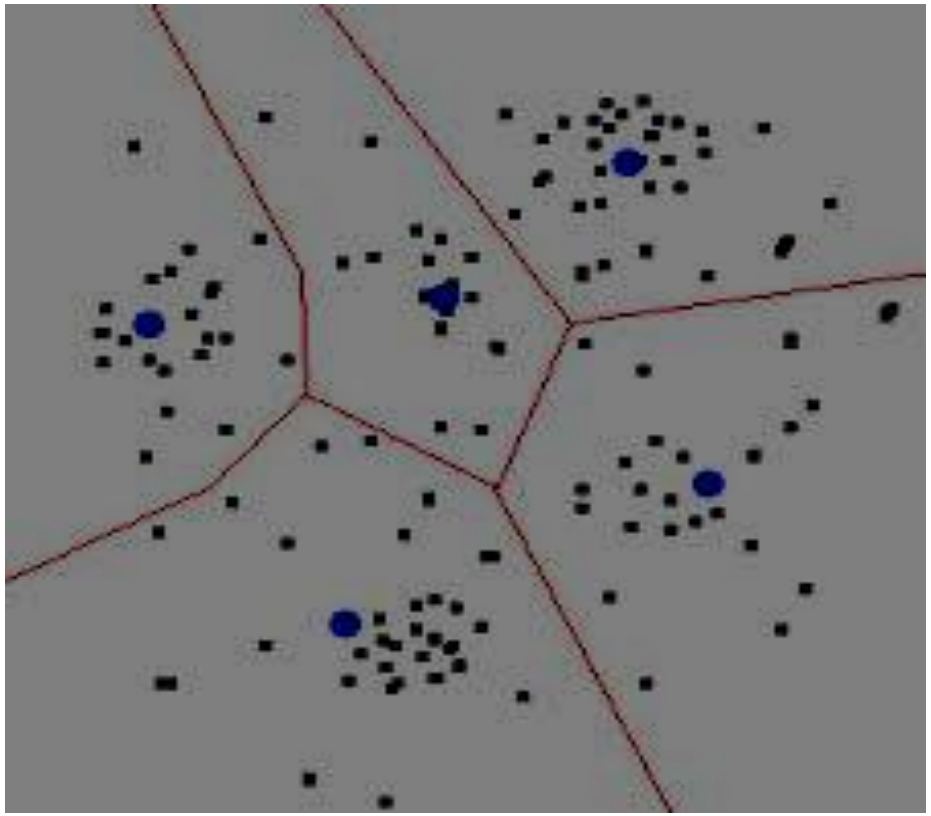
- Two ways of “clustering” the clustering algos
- Point assignment
  - Initially clusters are estimated by carrying out hierarchical clustering on a very small sample of the real dataset
  - Remaining points are then assigned to clusters to which it best fits
  - Determines all clusters at once
  - Kmeans and derivatives, fuzzy c-means etcetera

CLUSTERING STRATEGIES contd

- Hierarchical or agglomerative clustering
  - Each point initialized as a cluster
  - Clusters are merged based on their “closeness”
  - Combination stops when further clustering leads to undesirable results

```
WHILE it is not time to stop DO
    pick the best two clusters to merge;
    combine those two clusters into one cluster;
END;
```

K MEANS CLUSTERING



Description

Algorithm

Illustration

Prerequisites

Pseudocode

Complexity Analysis

Performance analysis

WHAT IS KMEANS CLUSTERING?

- Given a set of observations (x_1, x_2, …, x_n), where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into k sets ($k \leq n$ and $k>0$) $S = \{S_1, S_2, \ldots, S_k\}$ based on features/attributes of data points so as to minimize the within-cluster sum of squares (WCSS):

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in S_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2$$

- where μ_i is the geometric centroid of data points in S_i and x_j is the vector representing jth data point.

ALGORITHM

- K-Means clustering involve the following logical steps
  - Determine the value of k
  - Determine the initial k centroids
  - Repeat until convergence or until limit of max iterations has reached
    - Determine membership: Assign each point to the closest centroid
    - Update centroid position: Compute new centroid position from assigned members

ILLUSTRATION (using K=2)

Individual	Variable 1	Variable 2
1	1.0	1.0
2	1.5	2.0
3	3.0	4.0
4	5.0	7.0
5	3.5	5.0
6	4.5	5.0
7	3.5	4.5

Step 1:

Initialization: Randomly we choose following two centroids (k=2) for two clusters.

In this case the 2 centroid are: m1=(1.0,1.0) and m2=(5.0,7.0).

Individual	Variable 1	Variable 2
1	1.0	1.0
2	1.5	2.0
3	3.0	4.0
4	5.0	7.0
5	3.5	5.0
6	4.5	5.0
7	3.5	4.5

	Individual	Mean Vector
Group 1	1	(1.0, 1.0)
Group 2	4	(5.0, 7.0)

Step 2:

Thus, we obtain two clusters containing:

{1,2,3} and {4,5,6,7}.

Their new centroids are:

$$m_1 = \left(\frac{1}{3}(1.0+1.5+3.0), \frac{1}{3}(1.0+2.0+4.0)\right) = (1.83, 2.33)$$

$$m_2 = \left(\frac{1}{4}(5.0+3.5+4.5+3.5), \frac{1}{4}(7.0+5.0+5.0+4.5)\right)$$

$$= (4.12, 5.38)$$

Individual	Centroid 1	Centroid 2
1	0	7.21
2 (1.5, 2.0)	1.12	6.10
3	3.61	3.61
4	7.21	0
5	4.72	2.5
6	5.31	2.06
7	4.30	2.92

$$d(m_1, 2) = \sqrt{|1.0-1.5|^2 + |1.0-2.0|^2} = 1.12$$

$$d(m_2, 2) = \sqrt{|5.0-1.5|^2 + |7.0-2.0|^2} = 6.10$$

Step 3:

Now using these centroids we compute the Euclidean distance of each object, as shown in table.

Individual	Centroid 1	Centroid 2
1	1.57	5.38
2	0.47	4.28
3	2.04	1.78
4	5.64	1.84
5	3.15	0.73
6	3.78	0.54
7	2.74	1.08

Therefore, the new clusters are:

{1,2} and {**3**,4,5,6,7}

Next centroids are: m1=(1.25,1.5) and m2 = (3.9,5.1)

Step 4 :

The clusters obtained are:

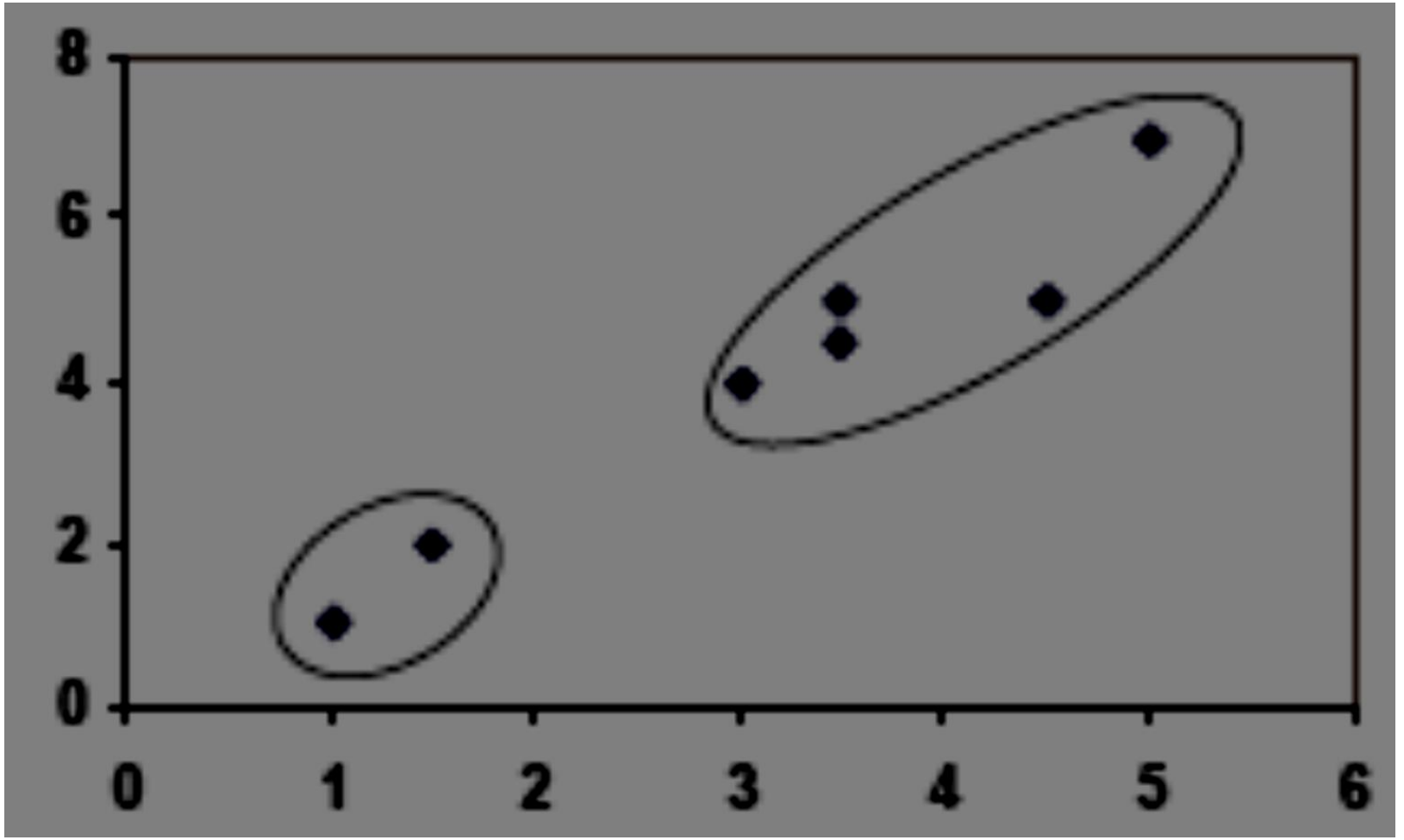
{1,2} and {3,4,5,6,7}

Therefore, there is no change in the cluster.

Thus, the algorithm comes to a halt here and final result consist of 2 clusters {1,2} and {3,4,5,6,7}.

Individual	Centroid 1	Centroid 2
1	0.58	5.02
2	0.58	3.92
3	3.05	1.42
4	6.66	2.20
5	4.16	0.41
6	4.78	0.61
7	3.75	0.72

PLOT



DECISIONS PRIOR CLUSTERING

- Representation of initial clusters
- Choosing an optimum number of clusters
- Deciding which point to be assigned to which cluster
- When to stop merging clusters
- Size of the dataset
- Nature of domain space

PSEUDOCODE

```
Input:
        E = {e[1], e[2]....e[n]}        #set of entities to be clustered
        k                               #number of clusters
        maxIter                         #termination criteria

Output:
        C = {c[1],c[2]....c[k]}         #set of cluster centroids
        L = {l[1],l[2]....l[n]}         #set of cluster labels corresponding to each entity

Psuedocode:

        E' = E;                         #to keep track of entities chosen as initial centroids
        iter = 1;
        jter = rand(|E'|);

        #initializing centroids of k clusters
        c[iter] = e[jter];              #Pick the first entity at random from the entity set
        E' = E' - {e[jter]};            #remove this entity so as not to reallocate as centroid of another cluster

        WHILE iter<=k DO
                Add the entity from E' whose minimum distance from the selected entities is as large as possible;
                remove that entity from E';
                iter++;
        END;

        #assign initial cluster labels to each entity
        FOR i=1 to |E|
                l[i] = argminDistance(e[i],c[j]) where(j = 1 to k);     #assigns that value of j to l[i] for which distance is min
        END
```

PSEUDOCODE contd

```
#implementing k-means
change = false;                 #termination condition
kter = 0;

repeat
        change = false;
        #recalculating cluster centroids
        FOR i = 1 to k
                mean, count = 0;
                FOR j = 1 to |E|
                        IF l[j] == i
                                mean = mean + e[j];
                                count = count + 1;
                        END
                END
                c[i] = mean/count;
        END

        #reallocating entities to k clusters
        FOR i = 1 to |E|
                temp = argminDistance(e[i],c[j]) where(j = 1 to k);
                IF temp != l[i]
                        l[i] = temp;
                        change = true;
                END
        END

        kter++;
until change=true && kter<=maxITer
```

COMPLEXITY ANALYSIS

- The runtime complexity of the algorithm is O(tkn), where,

t = number of iterations

k = number of clusters

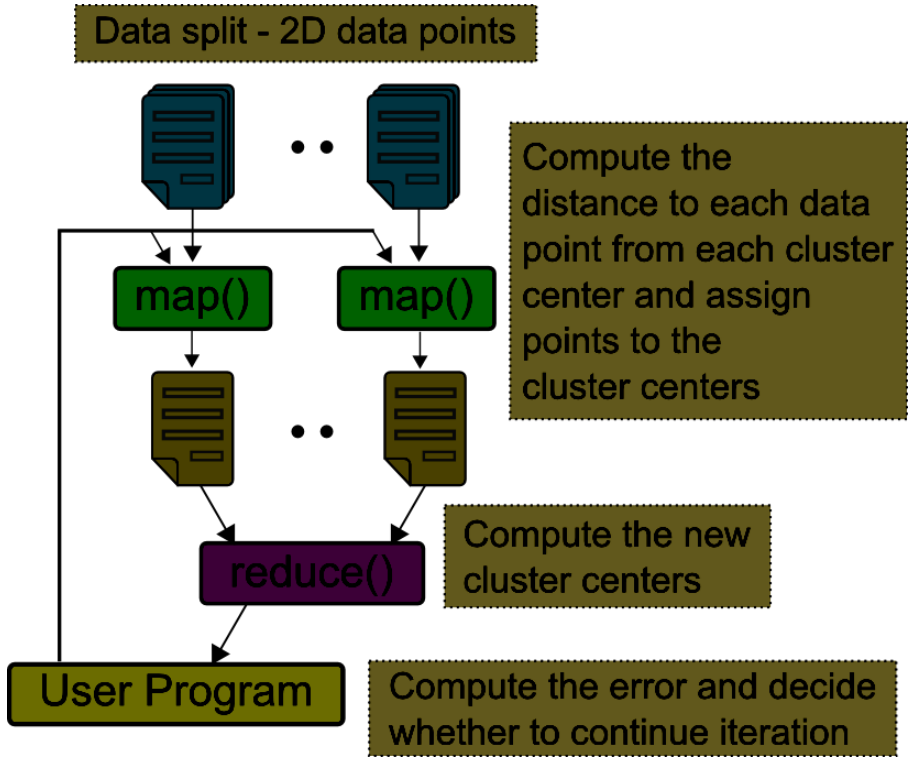
n = number of objects

PERFORMANCE ANALYSIS

- STRENGTH
  - Relatively efficient: O(tkn), t --> #iterations, k --> #clusters and n--> #objects
  - Often terminates at “local” optimum but “global” can be easily found using genetic algorithms
- WEAKNESS
  - Applicable only when the concepts of mean and distances are clearly defined
  - Dependence on initial conditions and inconsistency in results b/c of unordered data input
  - Lack of precision with non-convex surfaces, outliers and noisy data

MAPREDUCE ON KMEANS

- Kmeans as a candidate of mapreduce
- Map Algorithm
- Reduce Algorithm
- Illustration
- Complexity Analysis
- Performance Analysis



KMEANS AS A MAPREDUCE CANDIDATE

- Mapreduce will be used for “Centroid calculation” and “Cluster allocation” for each iteration until convergence
- Kmeans is an iterative process
- Vast amount of data independence in both the steps

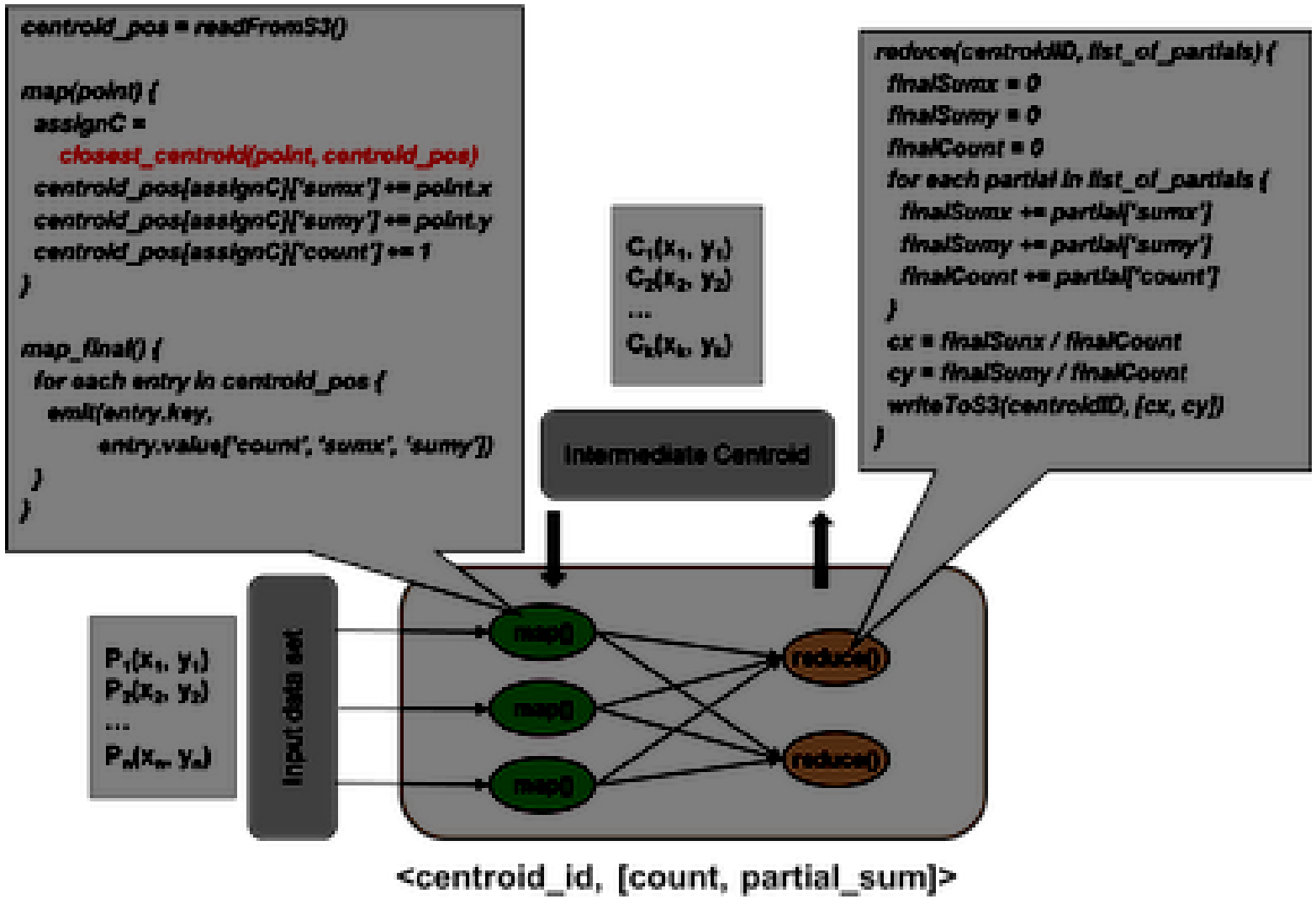
MAP ALGORITHM

```
Input:
        key: void

        value:
        E = {e[1], e[2]....e[n]}        #set of entities to be clustered
        C = {c[1],c[2]....c[k]}         #set of cluster centroids, sum initialized to entity value and cnt to 1 in the beginning

Output:
        cluster id;
        partial sum of member points corresponding to that cluster id;

Psuedocode:
        #initializing new clusters to keep partial cluster sum
        C' = {c'[1],c'[2]....c'[k]};
        FOR i = 1 to |C'|
                c'[i].sum=0;
                c'[i].cnt=0;
        END

        #computing partial cluster sum
        FOR i = 1 to |E|
                temp = argminDistance(e[i],c[j].sum) where(j = 1 to k);
                c'[temp].sum += e[i];
                c'[temp].cnt += 1;
                l[i] = temp;
        END

        #returning output
        FOR i = 1 to |C'|
                EmitIntermediate(i,c'[i]);
        END
```

REDUCE ALGORITHM

```
Input:
        key: id                                         #cluster id
        value: PS = {ps[1], ps[2]....ps[m]}             #list of partial sums corresponding to that particular id

Output:
        centroid of the input cluster

Psuedocode:
        Centroid final;
        final.sum=0;
        final.cnt=0;

        FOR i = 1 to |PS|
                final.sum += ps[i].sum;
                final.cnt += ps[i].cnt;
        END

        final.sum = final.sum/final.cnt;
        return final;
```

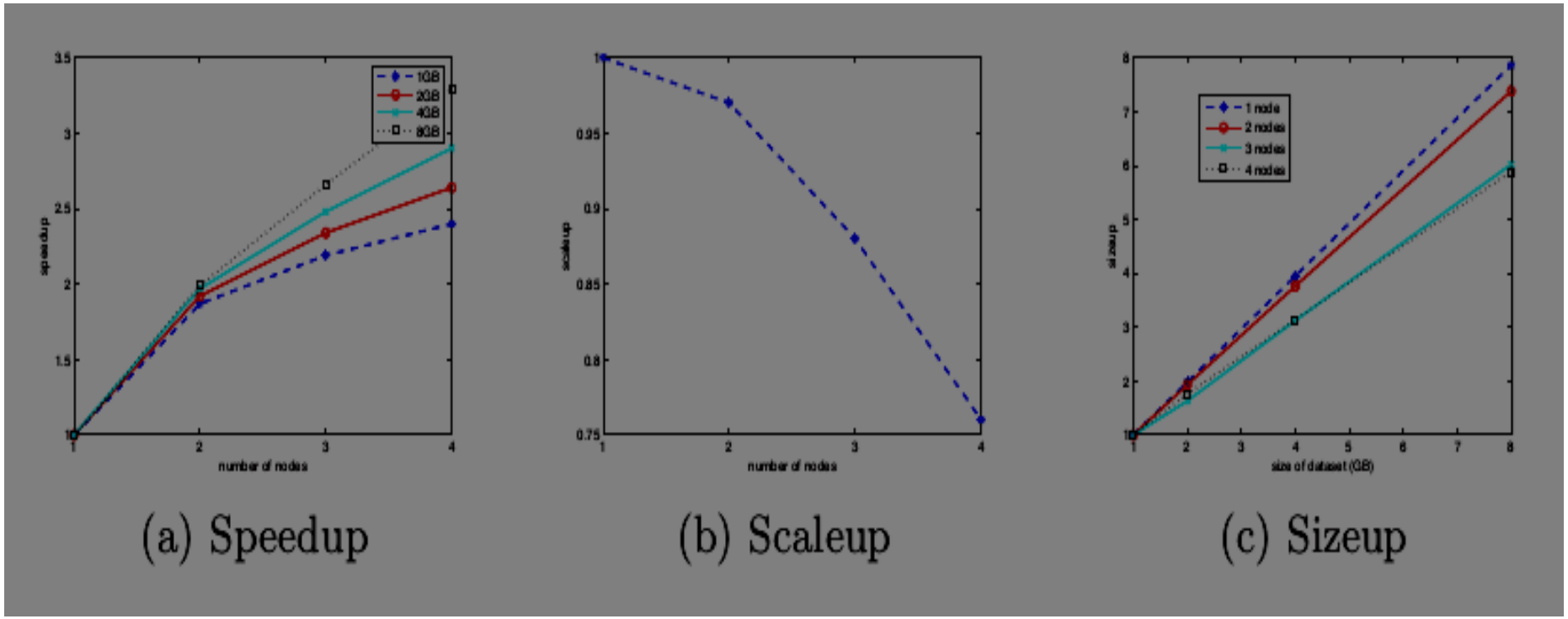
ILLUSTRATION



COMPLEXITY ANALYSIS

- The runtime complexity of mapreduce on kmeans will be O(k*n/p), where,
- k is number of clusters
- n is number of data points
- p is number of machines.

PERFORMANCE ANALYSIS



- *Reference: Parallel K -Means Clustering Based on MapReduce*
- *Weizhong Zhao , Huifang Ma, and Qing He*

Graph Clustering

Graph Clustering

- Given data points X_1, …, X_n and similarities $w(X_i,X_j)$, partition the data into groups so that points in a group are similar and points in different groups are dissimilar.

For e.g. If we are given a set of points, we can convert them into graph clustering problem by finding:

Similarity Graph: G(V,E,W)

where V – Vertices (Data points)

E – Edge if similarity > 0

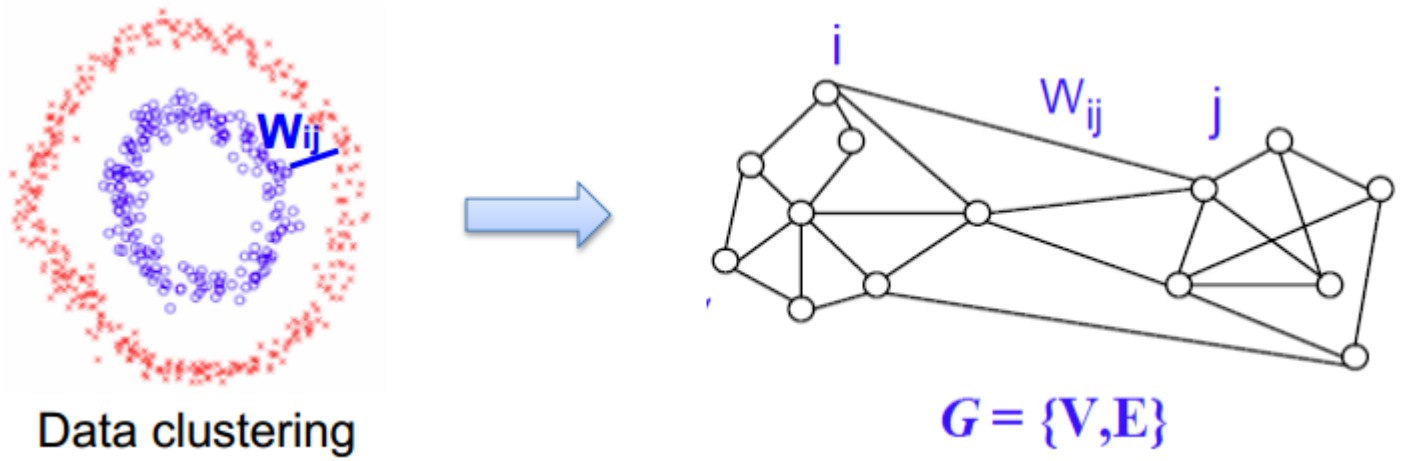
W - Edge weights (similarities)

Calculating weights

- Similarity can be calculated using this Gaussian function:

$w_{ij} = e^{-\|x_i - x_j\|^2/(2\sigma^2)}$

W_{ij} here inversely proportional to the distance between the two nodes

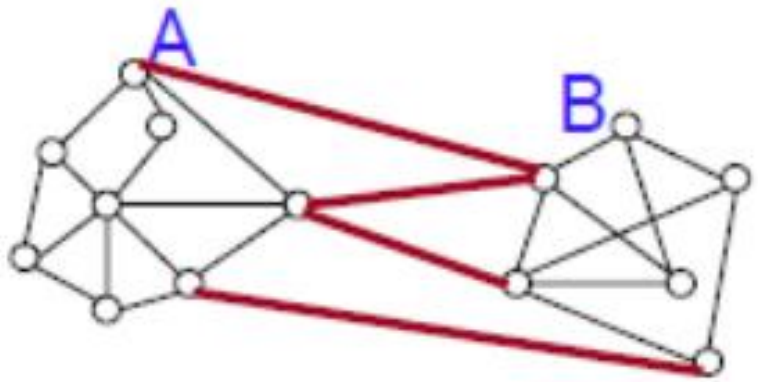


Data clustering

Creating Clusters

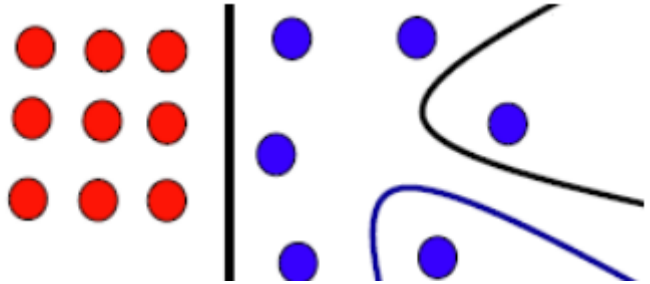
- Now we will like to cut those edges to form the cluster, such that

Cut(A,B) = $\sum w_{ij}$ where $i \in A$, $j \in B$



Problem with the Cut

- It can lead to this problem:



- Solution: Normalized Cut

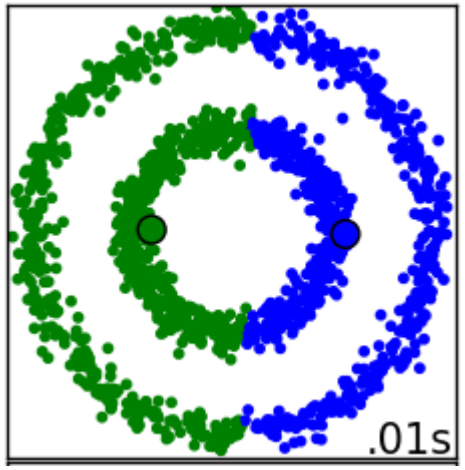
$$\text{Ncut}(A, B) := \text{cut}(A, B)\left(\frac{1}{\text{vol}(A)} + \frac{1}{\text{vol}(B)}\right)$$

where vol(A)= $\sum d_i$ where i ϵ A, d is degree.

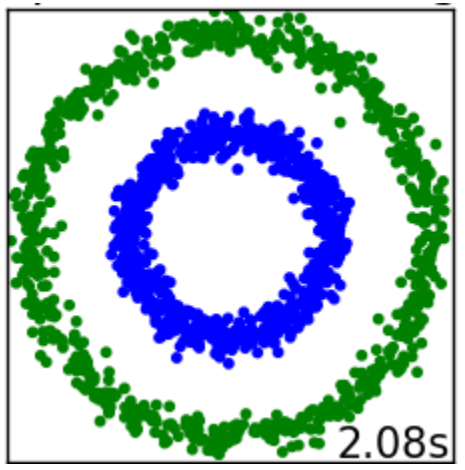
But this is a NP-hard problem.

Spectral Clustering

Spectral clustering uses the spectrum (eigenvalues) of the similarity matrix of the data to perform dimensionality reduction before clustering in fewer dimensions



Compactness
vs.
Connectivity



Variable Computation Required

- Form **Similarity Matrix,** W from the distance matrix by applying Gaussian similarity function element-wise

  We take the fully connected component graph as our similarity graph.

$$s(x_i, x_j) = e^{-\|x_i - x_j\|^2/(2\sigma^2)}$$

- **Degree matrix, D** is defined as the diagonal matrix with the degrees $d_1 \dots d_n$ on the diagonal.

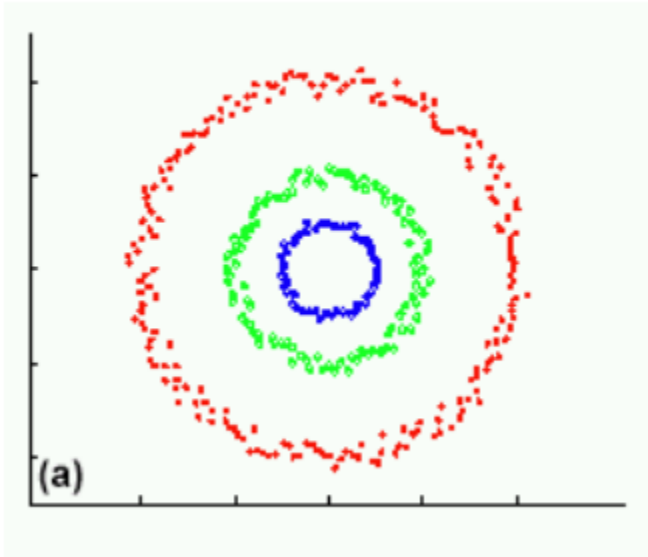
  $d_i = \sum w_{ij}$ where $1 <= j <= n$

- **Laplacian matrix**, L = D - W

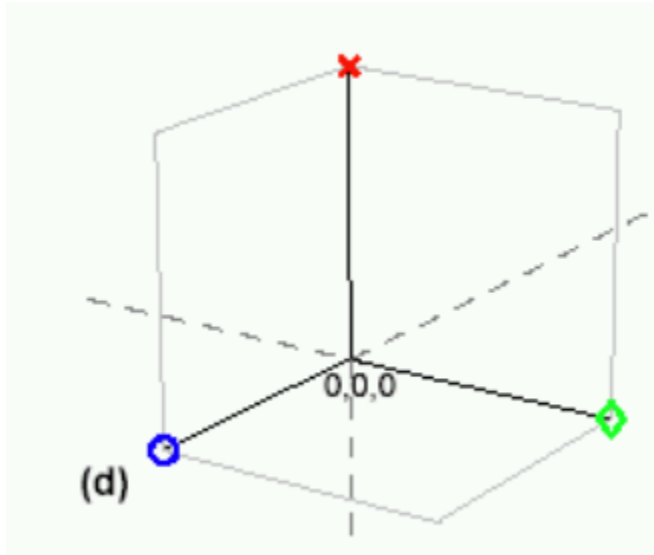
Overview

1. Given n data points, construct n-by-n distance matrix.
2. Form similarity matrix (W) from the distance matrix.
3. Form Laplacian matrix L from the similarity matrix
4. Compute the smallest k eigenvectors u_1, u_2,…,u_k of L.
5. Form a n-by-k matrix U containing the vectors u_1, u_2,…,u_k as columns.
6. Form a matrix Y by normalizing each of the U's rows
7. Treat each row of Y as a point on the data set and cluster them in k clusters via k-means clustering method.

Original data



Projected data



Broad division of the problem

- Finding similarity matrix
- Finding eigenvectors
- Applying k-means (other proximity clustering algorithm)

Calculating Eigenvectors

- The three most efficient ways of calculating eigenvectors for large datasets are :-
  - Power Method
  - Simultaneous Iteration
  - Lanczos Algorithm

Power Method

- We choose an initial approximation x_0 of one of the dominant eigenvectors (with maximum eigenvalue) of A. This initial approximation must be a nonzero vector in R_n.
- Finally we form the sequence given by

$A\,x_{n+1}=A\,x_n$

- *Limitation* : Gives only one dominant eigenvector.

Algorithm of Power Method

$$\begin{aligned}
\mathbf{x}_1 &= A\mathbf{x}_0 \\
\mathbf{x}_2 = A\mathbf{x}_1 &= A(A\mathbf{x}_0) = A^2\mathbf{x}_0 \\
\mathbf{x}_3 = A\mathbf{x}_2 &= A(A^2\mathbf{x}_0) = A^3\mathbf{x}_0 \\
&\vdots \\
\mathbf{x}_k = A\mathbf{x}_{k-1} &= A(A^{k-1}\mathbf{x}_0) = A^k\mathbf{x}_0.
\end{aligned}$$

Power method iteratively multiplies the randomly chosen X_0 vector and scales the output matrix (with lowest element) until it finally converges to some constant vector.

Simultaneous Iteration

- Similar to Power Iteration
- Takes n random vectors (each orthogonal to the other) and obtain an orthogonal matrix which increases the chances of obtaining different eigenvectors.

- *Limitation*: Requires to do matrix-matrix multiplication which is computationally expensive than matrix-vector multiplications.

Algorithm of Simultaneous Iteration

- Take n n-dimension vectors orthonormal to each other
- Pick a starting basis $\{v_1^{(0)}, \ldots, v_n^{(0)}\}$ of R_n.
- Build the matrix $V = [v_1^{(0)} | \cdots | v_n^{(0)}]$
- Obtain the factors $Q^{(0)}R^{(0)} = V^{(0)}$
- For k = 1,2, . . .

  Let $W = AQ^{(k-1)}$

  Obtain the factors $Q^{(k)}R^{(k)} = W$

Algorithm: Let $A_0 = A$, we iterate

$i = 0$

repeat

$A_i = Q_i R_i$ (QR decomposition)

$A_{i+1} = R_i Q_i$

$i = i + 1$

until convergence

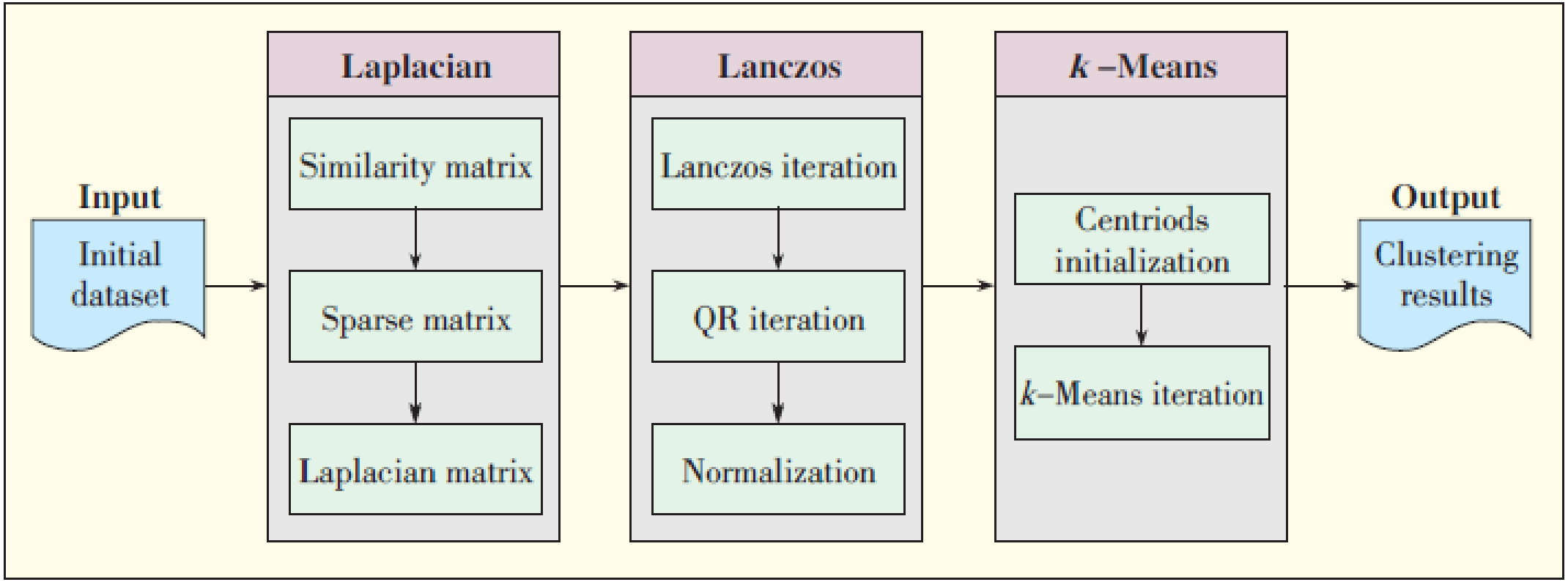
Lanczos Iteration

- Uses matrix-vector multiplication
- Matrix-matrix multiplications needed for matrix with significantly lower number of rows and columns.

- Finally, using eigenvector matrix(Y) of the Tridiagonal matrix obtained, find $V_m Y$.

Input: Matrix $A^{n \times n}$,
random n-vector b,
number of steps m
Output: Orthogonal matrix $V_m^{n \times m} = [v_1 \dots v_m]$,
coefficients $\alpha[1..m]$ and $\beta[1..m-1]$
$\beta_0 \leftarrow 0, v_0 \leftarrow 0, v_1 \leftarrow b/\|b\|$;
for $i = 1, ..m$ **do**
  $v \leftarrow Av_i$;
  $\alpha_i \leftarrow v_i^T v$;
  $v \leftarrow v - \beta_{i-1} v_{i-1} - \alpha_i v_i$; // make a new basis
  $\beta_i \leftarrow \|v\|$;
  if $\beta_i = 0$ **then**
    break for loop;
  end if
  $v_{i+1} \leftarrow v/\beta_i$;
end for

Overview


Input
Initial dataset
Laplacian
Similarity matrix
Sparse matrix
Laplacian matrix
Lanczos
Lanczos iteration
QR iteration
Normalization
k –Means
Centriods initialization
k–Means iteration
Output
Clustering results

Laplacian matrix using MapReduce

Map

- *Input*:
  Key => ($Index_i$, $Features_i$),
  Value => List of ($Index_k$, $Features_k$) where 1 <= k <= n; k!=I

- *Output*:
  Key =>($Index_i$),
  Value => List of ($Index_k$, $Similarity_{ik}$)

Reducer : Identity

Note: **Reducer** is identity for fully-connected graph. It can be used to make sparse matrix.

Map: Identity

Reducer

- *Input:*
  Key=> ($Index_i$),
  Value=> List of ($Index_k$, $Similarity_{ik}$)

- *Output:*
  Key=> ($Index_i$),
  Value=> $Degree_i$

Laplacian Matrix = D - W

```
Mapper :

Input:
        key=>{ Index(i) , features(i)};
        value=>list { index(k),features(k)};
                                where k=1..n && k!=i;

Output:
        key=>{index(i)}
        value=>list {index(k),similarity(ik)}

Pseudocode:
        input_content=getfeatures of input_index(i);
        foreach k in input_value_list:
                k_content=getfeaturesof index(k);
                similarity(ik)=CalculateSimmilarity(i_content,k_content);
        emit(index(i),List {index(k),similarity(ik)});


Reducer:
Input:
        Key=>{index(i)};
        value=>list {index k, similarity(ik)};

Output:
        key=>{index(i)};
        value=>{ degree(i)};


Pseudocode:
        foreach k in input_value_list:
                degree(i)=Sum (similarity(ik))
        emit {index(i),degree(i)};
```

Matrix-Vector Multiplication using MapReduce

```
Mapper:

Input :
        key=>NULL
        Value=>{i,j,a_ij,v_j};

Output:
        key=>{i};
        value=>{product };
map(key, value):
    for (i, j, a_ij) in value:
        emit(i, a_ij * v[j])
```

```
Reducer:

Input:
        key=> {i}
        value=>{product}
Output:
        key=>{i}
        valure={summation}
reduce(key, values):
    result = 0
    for value in values:
        result += value
    emit(key, result)
```

Markov Clustering

- MCL is a graph clustering algorithm which finds out the clusters of vertices by simulating random walks.

- Random walk : - Considering a graph, there will be many links within a cluster, and fewer links between clusters.

So that means if you were at a node, and then you have to randomly travel to a connected node, you’re more likely to stay within a cluster than travel between.

- Now how the simulation of random walk is done?

Markov Chain

- Random walk is simulated and calculated in MCL by using Markov Chain.
- Markov Chain is presented as a sequence of states X1, X2, X3, etc. Given the present state, the past and future states are independent, which means that the next state of system depends on and only depends on the current state of system.
- This property can be formally described as below:
- $Pr (X_{n+1} = x \mid X_1 = x_1, X_2 = x_2, X_3 = x_3, \ldots, X_n = x_n) = Pr (X_{n+1} = x \mid X_n = x_n)$
- In MCL, the states are represented as probability matrix. In this
- matrix, if the value of each element indicates the possibility to “walk” through a certain edge, then it is called probability matrix.

Probability Transition Matrix

An example :-

0.6	0.2
0.4	0.8

Next step – t0 → t1 → t2

1 -> 1 -> 1 + 1 -> 2 -> 1

.6 * .6 + .4 * .2 = .44

0.6	0.2
0.4	0.8

0.6	0.2
0.4	0.8

→

0.44	0.28
0.56	0.72

→

0.35	0.32
0.65	0.68

Graph to probability matrix

- Given an adjacency matrix with weights :-

- Normalize the values by dividing each value by the sum of all values in the column of that value. So in each column the sum of all value is 1 and hence it becomes probability matrix.

0	2	1	3
2	0	0	2
1	0	0	0
3	2	0	0

0	1/2	1	3/5
1/3	0	0	2/5
1/6	0	0	0
1/2	1/2	0	0

Expansion and Inflation

- In MCL, the following two processes are alternated between repeatedly:
  - Expansion (taking the Markov Chain transition matrix powers)
  - Inflation
- The expansion operator is responsible for allowing flow to connect different regions of the graph.
- The inflation operator is responsible for both strengthening and weakening of current.

Expansion and Inflation Contd.

- The power represents the length of the random walk and A[i][j], where A is the matrix, is probability of the random walk ending at ith node given that it started from jth node.

- By taking power of the probability matrix, we are actually expanding the length of the random walk and MCL will act a random walk to allow the flow going between vertices which are not directly connected. This operation is called expansion in MCL.

- Inflation : - The inflation operator is responsible for both strengthening and weakening of current and also inflation parameter, r, controls the extent of this strengthening / weakening.

MCL Algorithm

- Input is an undirected graph, power parameter e, and inflation parameter r.
- Create the associated matrix and add self loops and normalize it.
- Expand by taking the eth power of the matrix.
- Inflate by taking inflation of the resulting matrix with parameter r.
- Repeat steps 5 and 6 until a steady state is reached (convergence).
- Interpret resulting matrix to discover clusters.

MCL Convergence

- The matrix converges to a steady state where all the values of each row contain some probability and most of the other probabilities go down to approximately 0.

- The algorithm converges nearly always to a "doubly idempotent" matrix:
  1. It's at steady state.
  2. Every value in a single column has the same number (homogeneous).

1			1		1
	1				
		0.5		0.5	
		0.5		0.5	

MCL interpretation

- To interpret clusters, the vertices are split into two types.
- Attractors (row numbers representing vertices), which attract other vertices, and vertices (column numbers) that are being attracted by the attractors.
- Attractors have at least one positive flow value within their corresponding row (in the steady state matrix).
- Each attractor is attracting the vertices which have positive values within its row.
- Attractors and the elements they attract are swept together into the same cluster.
- So clusters of this matrix are :- {1,4,6}, {2}, {3,5}

Map reduce - MCL

- Firstly we are dividing the problem into 3 sub parts :-
  a) How to express a matrix for MCL problem.
  b) How to compute power of a matrix in a manner that scales.
  c) How to compute Hadamard power(inflation) of matrix in a scalable way.

Map Reduce – Matrix Pseudocode

Map-Reduce Input: connectivity matrix

Map-Reduce output: probability matrix

// entry.id is the row number of that entry

```
class MarkovMapper
    method map(column)
    sum=sum(column)
    for all entry ∈ column do
        entry.value=entry.value/sum
        emit {column.id, {“column”, entry.id, entry. Value}}
        emit {entry.id, {“row”, column.id, entry.value}}
```

- Map-Reduce execution framework does the sorting and shuffling part and each reducer gets a list of values under the same key.

```
class MarkovReducer
        method reduce(key, list {sub-key, id, value}})
        newcolumn=[]
        newrow=[]

        for all element ∈ list do
                if sub-key is “column” then newcolumn.add({key, id, value})
                else newrow.add({id, key, value})
        emit newcolumn
        emit newrow
```

Reduce function fetches all of the values under a same key, and
puts it back to file system, in our case, the new rows and columns with probability values will be put back to file system.

After that, the data probability matrix will look like the following:

- Column 0-> {0 0.5, 1 0.25, ….}
- Column 1-> {0 0.25, 1 0.5, ….}
- ….
- Row 0-> {0 0.5, 1 0.25, ….}

Expansion Map-reduce

Map-Reduce Input: markov matrix

Map-Reduce output: expanded matrix

```
class ExpansionMapper
        method map(column, row)
                sum=0
                for all entry ∈ column do
                        if entry.id is contained in row.idSet then
                                sum=sum+entry.value*row.get(entry.id).value
                if sum!=0 then emit <key: column.id, value : { row.id, sum}>

class ExpansionReducer
        method reduce(key, list {row.id, value}})
        newcolumn=[]
        for all element ∈ list do
                newcolumn.add({key, row.id, value})
        emit newcolumn
```

Map-reduce : Inflation

Map-Reduce Input: expanded matrix

Map-Reduce output: markov (probability) matrix

```
class InflationMapper
        method map(column,r)
        sum=sum(all column.entry.value^r)
        for all entry ∈ column do
                entry.value=entry.value^r/sum
        emit {column.id, {“column”, entry.id, entry. value}}
        emit {entry.id, {“row”, column.id, entry.value}}
class InflationReducer
        method reduce(key, list {sub-key, id, value}})
        newcolumn=[]
        newrow=[]
        for all element ∈ list do
                if sub-key is “column” then newcolumn.add({key, id, value})
                else newrow.add({id, key, value})
        emit newcolumn
        emit newrow
```

Graph Data Structure

- Two kinds of structure of graph files are taken into consideration:
  1. Single File: use one file to contain all of the information of the graph.
  2. Multiple Files: one file only contains information of one vertex, thus, the number of file is equal to the number of vertices in the graph.

- In single file, the graph is represented as:

<Vertex ID> <Edge 1 ID> <Edge 1 Weight> <Edge 2 ID> <Edge 2 Weight>… <Edge n ID>

- But in multiple file format, the ID of vertex will be used as the file name, and in the file, the graph will be shown as:

<Edge 1 ID> <Edge 1 Weight> <Edge 2 ID> <Edge 2 Weight>… <Edge n ID> <Edge n Weight>.

Intermediate Result

- To store the intermediate result following format is followed. Files have blocks of data. For example, a sub-block is described as following:

<Column From, Column To>

<Column ID> <Row ID 1> <Value 1> <Row ID 2> <Value 2>… <Row ID n> <Value n> *

<Row From, Row To>

<Row ID> <Column ID 1> <Value 1> <Column ID 2> <Value 2>… <Column ID n> <Value n>*

<End>

- Expansion and Inflation will be applied alternately until termination. Intermediate result will be stored using block format on HDFS and be taken as the input for the next job.

Termination

- There are 2 ways to decide the termination of algorithm:

    1. When all of elements in the matrix stop changing.
    2. When the number of output records in reduce task is the number of vertices.

- The first way is the most accurate method to decide when MCL should terminate. But comparing two large matrices after every map-reduce task would be a huge overhead.

- The second way is a pretty smart way to decide when to terminate. It borrows the built-in counter feature of Hadoop, and doesn't cause any extra work load. But it works only when all the values converge to 1.

- So :D we decide to keep the number of expansion-inflation iterations constant for now.