Mining Massive Datasets

Lecture 3

Artur Andrzejak
http://pvs.ifi.uni-heidelberg.de

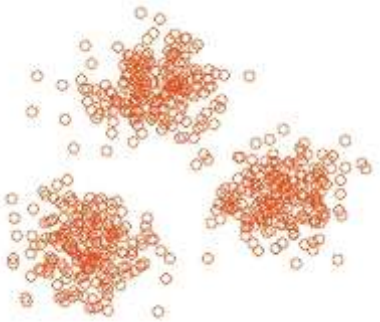
Note on Slides

A substantial part of these slides come (either verbatim or in a modified form) from the book Mining of Massive Datasets
by Jure Leskovec, Anand Rajaraman, Jeff Ullman (Stanford University).
For more information, see the website accompanying the book: http://www.mmds.org.

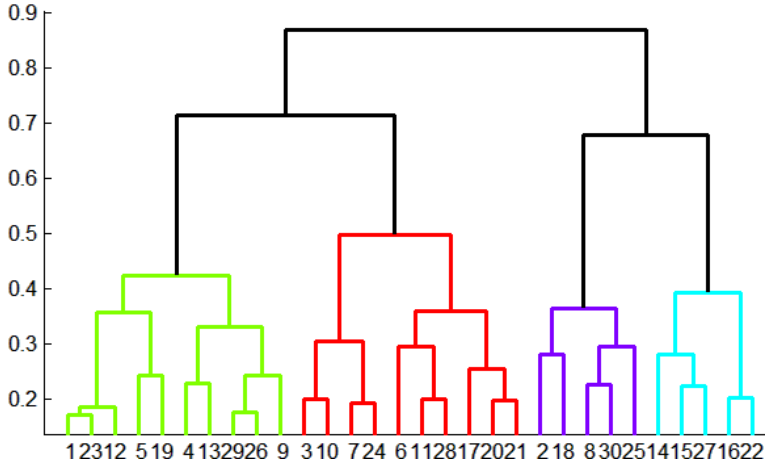
Hierarchical Clustering

Overview: Methods of Clustering

- **Partitioning** approaches (e.g. k-means):
  - Iteratively find k cluster "centers"
  - Points belong to "nearest" center

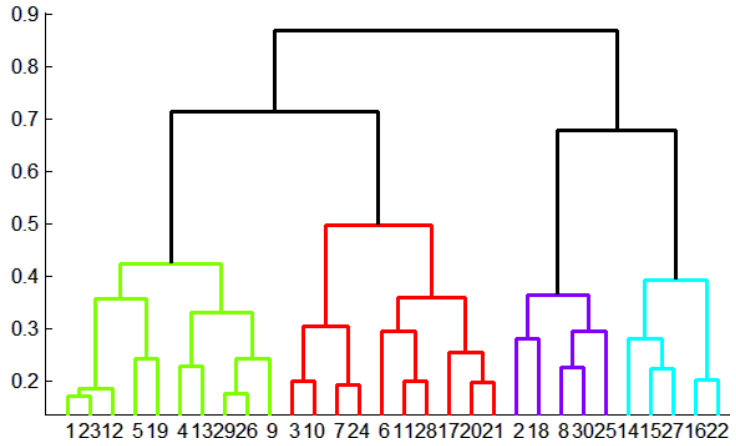


- **Hierarchical:**
  - Create a hierarchy of subdivisions by **merging** or **splitting** clusters
  - **Agglomerative** (bottom up):
    - Initially, each point is a cluster
    - Repeatedly combine the two "nearest" clusters into one
  - **Divisive** (top down):
    - Start with one cluster and recursively split it



Agglomerative Clustering

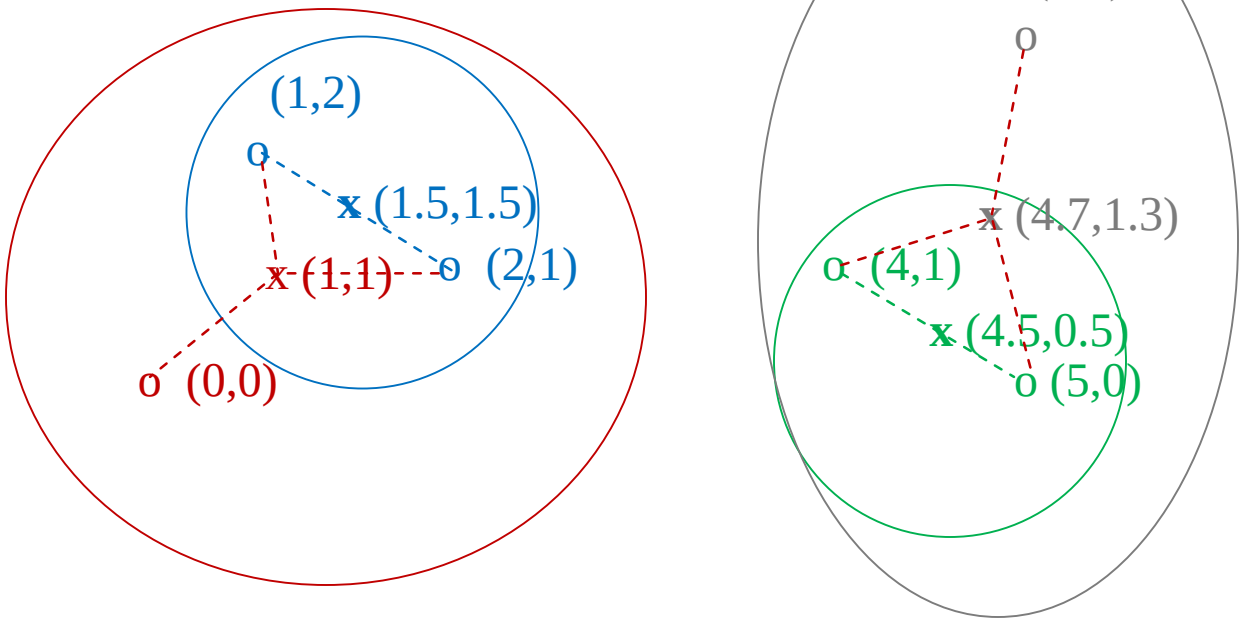
- **Key operation:**
  Repeatedly combine two nearest clusters



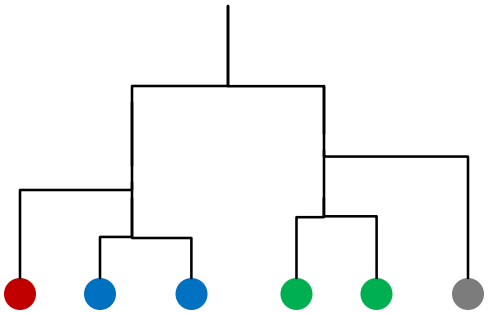
- **Three important questions:**
  1. How do you represent a cluster of more than one point? => **centroids**, **clustroids**
  2. How do you determine the "nearness" of clusters?
  3. When to stop combining clusters?

centroid = **average** of cluster's (data) points (seen as vectors)
clustroid = (data) point "**closest**" to other points in the cluster

Example: Hierarchical clustering



Data:
o … data point
x … centroid



Dendrogram

Pseudocode P. 1: Merge Clusters

- def (outCluster:list, outMerged:pair) =
  merge(inCluster:list):
  - Compute pairwise distances between all clusters in inCluster # define distance later
  - Find a pair (P, Q) of clusters with the smallest distance
  - Merge P and Q into cluster R
  - outCluster = (inCluster \ {P, Q}) $\cup$ {R}
  - outMerged = (P, Q) (or indices of P, Q)
  - return (outCluster, outMerged)

Pseudocode Part 2: Main Loop

main():

- inCluster = RDD with pairs (point_index, point)
- merged:list = []
- while inCluster.count() > 1:
  - outCluster:list, outMerged:pair = **merge**(inCluster)
  - merged.append(outMerged)
  - inCluster = outCluster # safe: RDDs are read-only
- print merged

Defining "Nearness" of Clusters

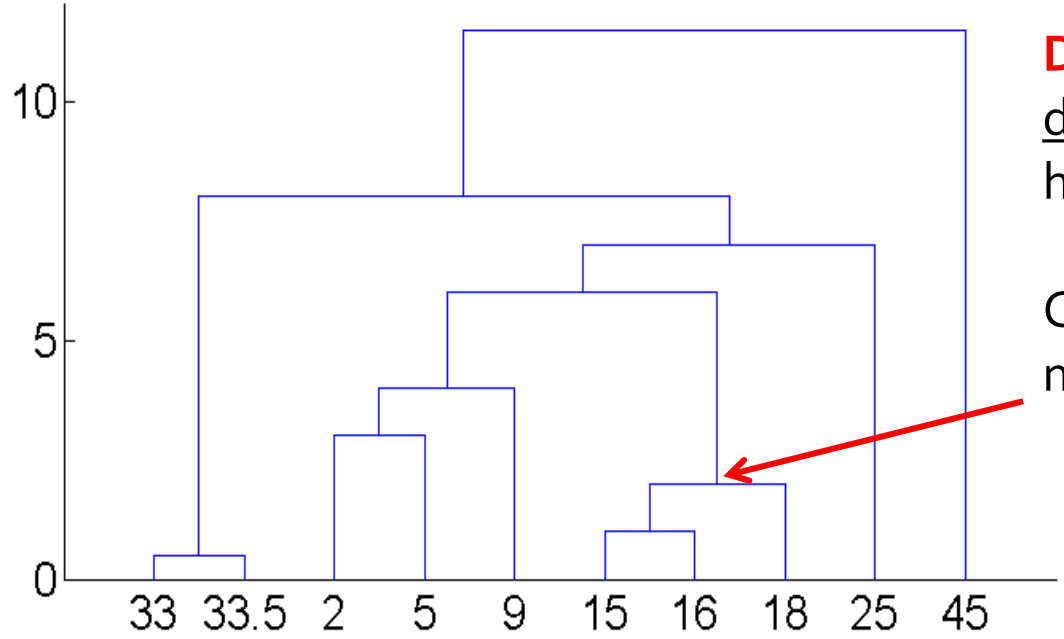
- **How do you determine the distance of clusters?**
  - **Approach 1 (centroid distance)**:
    - Measure cluster distances by distances of **centroids**
  - **Approach 2 (intercluster distance)**:
    - **Single-linkage method**: minimum of the distances between any two points, one from each cluster
    - **Complete-linkage method**: maximum of …
  - **Approach 3 (cohesion quality)**:
    - Pick a notion of a "**cohesion**" of clusters
      - *e.g.*, a maximum distance from the **clustroid**
        - **clustroid** = (real) point "**closest**" to other points in the cluster
    - Merge clusters whose union is most cohesive

Zusammenhang, Zusammenhalt

Example: Single-Linkage

Agglomerative, single-linkage, Euclidean distance metrics

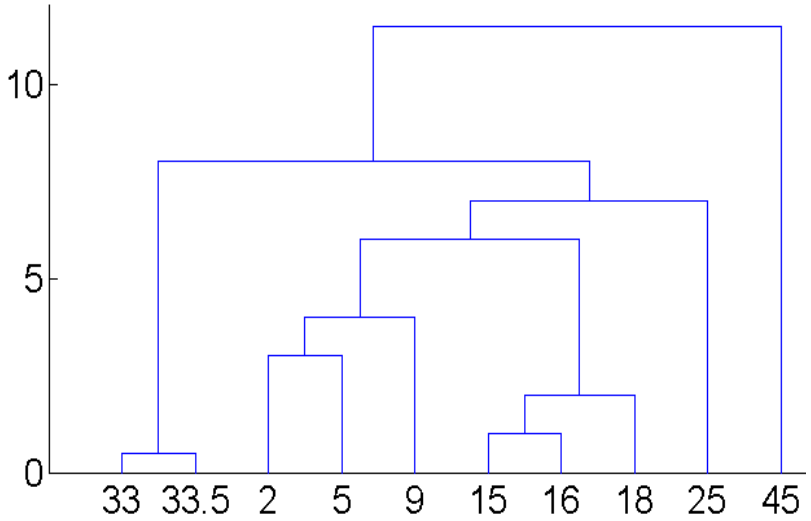
- … = Minimum of the distances between any two points
- Data: [2; 5; 9; 15; 16; 18; 25; 33; 33.5; 45]; dim d=1, N = 10
  - Which clusters are created first?



Dendrogram: y-axis shows the distance at which two clusters have been merged

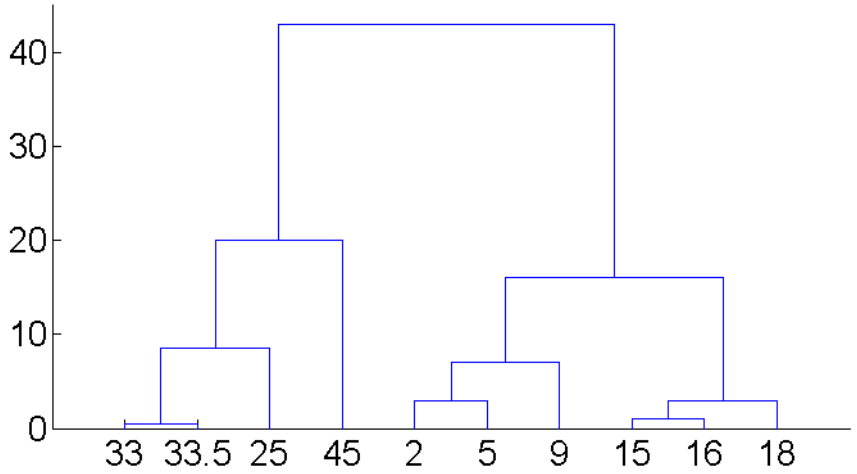
Clusters {15, 16} and {18} merged at distance = 2

Comparison Single-Linkage vs. Complete-Linkage



Single-Linkage

= minimum of the distances between any two points



Complete-Linkage

= maximum of the distances between any two points

Cohesion Metrics: Option

- **Approach 3.1:** Use the **diameter** of the merged cluster = maximum distance between points in the cluster
- **Approach 3.2:** Use the **average distance** between points in the cluster
- **Approach 3.3:** Use a **density-based approach**
  - Take the diameter or avg. distance, e.g., and divide by the number of points in the cluster

Runtime - Sequential Version

- What is the (sequential) runtime in *O*()-notation?
  - Assume that each cluster is represented by a centroid
  - Count distance computation **d**(*, *) as one step
    - * = point or centroid
- At each of $O(N)$ merge-steps …
- … compute pairwise distances between all $O(N^2)$ pairs of clusters, then merge a pair
- => in total, $O(\boldsymbol{N^3})$
- Careful implementation using priority queue can reduce time to $O(N^2 \log N)$
  - Still too expensive for really big datasets!

Hierarchical Clustering in Spark

Implementing Subroutine merge

Recall: Subr. Merge (Adapted)

- def (outCluster:list, outMerged:pair) =
  merge(inCluster:list):
  - Compute pairwise distances between all ~~clusters~~ now: **centroids** in inCluster
  - Find a pair of **centroids** (p, q) with the smallest distance (the centroids are P, Q)
  - Compute **centroid** r of cluster R = P ∪ Q
  - outCluster = (inCluster \ {p, q}) ∪ {r}
  - outMerged = (p, q) (or indices of them)
  - return (outCluster, outMerged)

Merge in Spark: Step 1 of 4

1. Compute pairwise distances between all centroids in inCluster
2. Find a pair (p,q) of centroids with the smallest distance

- Assume that inCluster is an RDD with pairs (centroid_index, centroid), i.e. (0,centroid0); (1; centroid1) (2; c2), …
- Compute RDD distances with triples: **(** D(p,q), (p,q), (ip,iq) **)**
  - where **ip** = centroid_index of p; **iq** = … of q

```
N = inCluster.count()
distances = sc.parallelize([])
for cIndex in range(N):
    iq, q = inCluster.filter(lambda idx, p: cIndex==idx ).collect()
    partialDistances = inCluster.map(lambda ip, p: ( D(p,q), (p,q), (ip,iq) ) )
    partialDistances = partialDistances.filter(lambda d, arg2, arg3: d > 0)
    distances = distances.union(partialDistances)
```

A bit better approach in: "Pairwise Element Computation with MapReduce" by T. Kiefer, P.B. Volk, W. Lehner, HPDC'10, 2010, goo.gl/58HESM

Remove pairs (p, p) with distance = 0.0

Merge in Spark: Step 2 of 4

1. Compute pairwise distances between all centroids in inCluster
2. Find a pair (p,q) of centroids with the smallest distance

- We need to find in distances a pair of different centroids with the smallest distance d (but d > 0.0)
- First solution (A):

```
bestPair = distances.sortBy ( lambda x: x[0] ).first()
# See Docs
```

- Output is a triple **(** D(p,q), (p,q), (ip,iq) **)**

Merge in Spark: Step 2 of 4

1. Compute pairwise distances between all centroids in inCluster
2. Find a pair (p,q) of centroids with the smallest distance

- We need to find in distances a pair of different centroids with the smallest distance
- Other solution (B): compute without sorting

```
def getMin (left, right):
    # left and right are each ( D(p,q), (p,q), (ip,iq) )
    d0, pair0, indices0 = left              # unwrap tuple
    d1, pair1, indices1 = right             # unwrap tuple
    result = left if d0 <= d1 else right
    return result     # output: ( D(p,q), (p,q), (ip,iq) )

bestPair = distances.reduce(getMin)
```

You can do this on partialDistances in step 1 => eliminates RDD distances

Merge: Step 3 of 4

Simple version: if we just need the dendrogram, but no cluster membership

3. Compute centroid r of cluster R = P ∪ Q
4. outCluster = (inCluster \ {p, q}) ∪ {r}

- We know bestPair = a triple **(** D(p,q), (p,q), (ip,iq) **)**
  - => So we know centroids (vectors!) p and q
- Is r' = (p **+** q)/2 a centroid of R = P ∪ Q? (vector **+**)
- No! E.g. if P has 1 point, and Q many points
- Better:
  - Store each p, q as a pair: (sum, count), i.e. (vector sum of points in cluster, #points in cluster)
- Then:
  - r = (p.sum + q.sum)/(p.count + q.count)
  - We store:
    - r.sum = p.sum + q.sum; r.count = p.count + q.count

Merge: Step 3 of 4

Simple version: if we just need the dendrogram, but no cluster membership

3. Compute centroid r of cluster R = P ∪ Q
4. outCluster = (inCluster \ {p, q}) ∪ {r}

- We know bestPair = a triple **(** D(p,q), (p,q), (ip,iq) **)**
- Store each p, q as a pair: (sum, count), i.e. (vector sum of points in cluster, #points in cluster)
- Code:
  - d, (p,q), (ip,iq) = bestPair # and p, q are each (sum, count)
  - sum = p[0] **+** q[0] # vector **+**
  - count = p[1] + q[1] # scalar +
  - r = (sum, count) # r is also (sum, count)

Merge: Step 4 of 4

Simple version: if we just need the dendrogram, but no cluster membership

3. Compute centroid r of cluster R = P ∪ Q
4. outCluster = (inCluster \ {p, q}) ∪ {r}

- Recall: inCluster is RDD of (centroid_index, centroid)
- We know bestPair = a pair (p, q) of centroids
- We computed r = (sum, count) = (p[0]+q[0], p[1]+q[1])
- Code for outCluster = (inCluster \ {p, q}) ∪ {r}:
  - d, (p,q), (ip,iq) = bestPair
  - inCluster = inCluster.filter(**lambda** idx, p:
    (idx! = ip) **and** (idx != iq))
  - <Pseudocode: Re-number centroid_index in inCluster
    - From 0 to **N** := inCluster.count()-1 >
  - outCluster = inCluster.union(sc.parallelize(**(N**+1, r**)**))

Main Loop in Spark: Problem Set

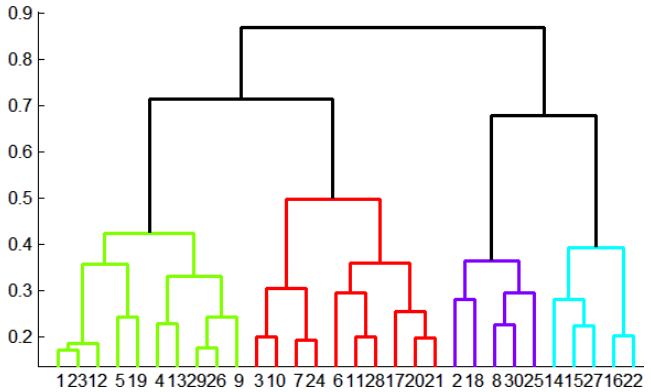
main():

- inCluster = RDD with pairs (point_index, point)
- merged:list = []
- while inCluster.count() > 1:
  - outCluster:list, outMerged:pair = **merge**(inCluster)
  - merged.append(outMerged)
  - inCluster = outCluster # safe: RDDs are read-only
- print merged

Scalability: Bad News, Good News

- Does this algorithm scale?
- In practice, no: we need $O(N^2)$ RDDs partialDistances
  - For $N > 10^4$, there are better sequential algorithms
- In general, algorithms with runtime $O(N^2)$ or worse (here: $O(N^3)$) are hopeless for "big data"

- Good news: hierarchical clustering on large data is pointless
  - Usually done for the dendrogram, but how to show a tree with $>10^6$ leaves?
  - Better: run parallel k-means for k = 100…1000, then run sequential hierarchical clustering on centroids



Clustering:
Metrics and Evaluation

Distance Metrics

- We have been using the **Euclidean distance**

$$D_2(x,y) = \left(\sum_{i=1}^{d} (x_i - y_i)^2 \right)^{1/2}$$

- "**City block**"/"**Manhattan**" Metric **L_1**

$$D_1(x,y) = \sum_{i=1}^{d} |x_i - y_i|$$

- Generalization of both: **Minkowski**-Metric

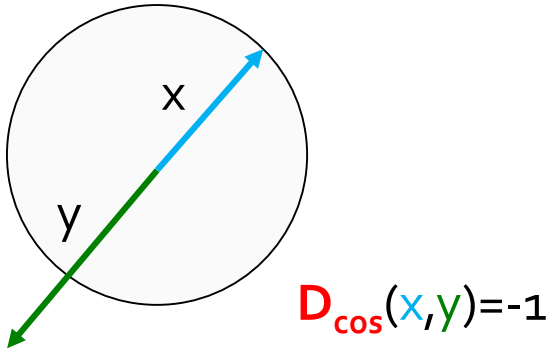
$$D_p(x,y) = \left(\sum_{i=1}^{d} |x_i - y_i|^p \right)^{1/p}$$

Other Metrics

- **Hamming distance D_H**
  - For vectors x, y of 0/1s, then $D_H(x, y)$ is the number of divergent components in x, y
  - This is one of the previous metrics D_x – which one?

- **Cosine similarity metric D_{cos}**

$$D_{\cos}(x, y) = \frac{\sum_{i=1}^{d} x_i \cdot y_i}{\sqrt{\sum_{i=1}^{d} x_i^2 \sum_{i=1}^{d} y_i^2}}$$



  - Measures the degree of (geometric) parallelism of vectors x, y
  - 1, if y = cx, constant c > 0; and -1, if y = cx, c < 0

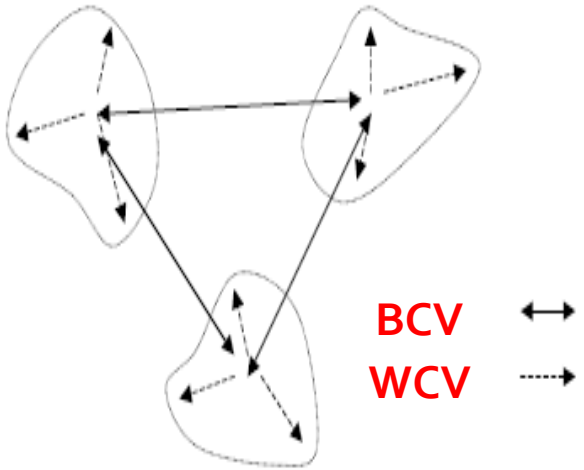
Metrics for Partitioning Quality

- We used for k-means termination:
  - Sum of squared distances by which each centroid has „moved“ between iterations
- Alternative: **sum of *squared* errors (SSE)**
  - **SSE** is sum of *squared* distances between each sample in cluster C_i and its centroid M_i, summed over all clusters
  - Formal definition: (C_i is cluster, M_i its centroid)

$$SSE = \sum_{i=1}^{k} \sum_{s \in C_i} D(s, M_i)^2$$

Evaluating Partitioning Quality

- Intuitively, in clustering we want to:
  - Maximize the distance between clusters
  - Minimize the variance (of distance) within each cluster
- **=> BCV: between-cluster variation**
- **=> WCV: within-cluster variation**
- So the **quotient BCV/WCV** can be used for estimating the quality of partitioning (= result of clustering)



Computing BCV/WCV

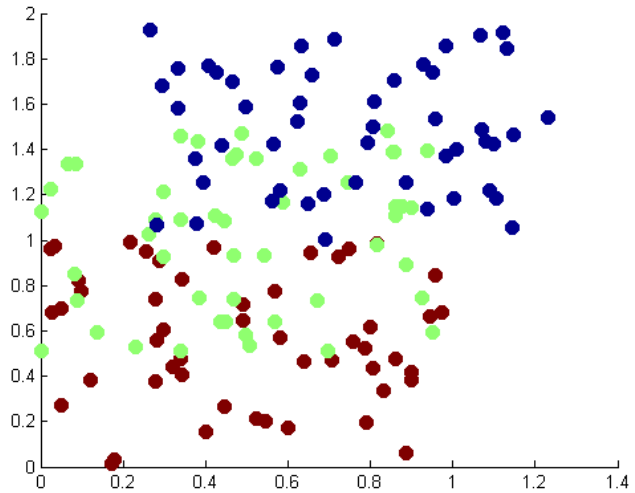
- Approximating BCV/WCV (M_i = Centroid of cluster C_i):
  - BCV: We use sum of squared distances between centroids

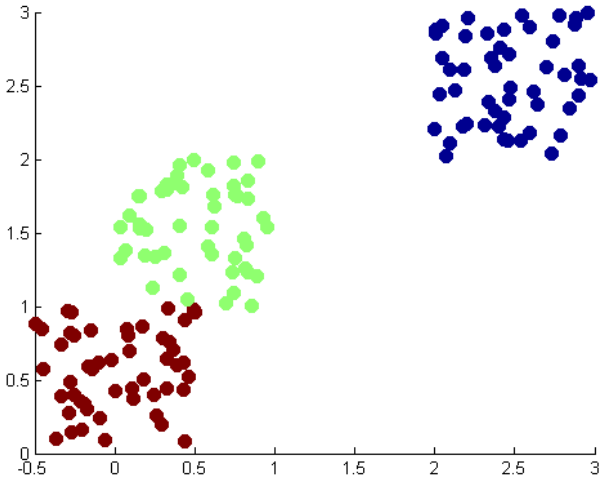
$$BCV = \sum_{i=1}^{k} \sum_{j>i}^{k} D(M_i, M_j)^2$$

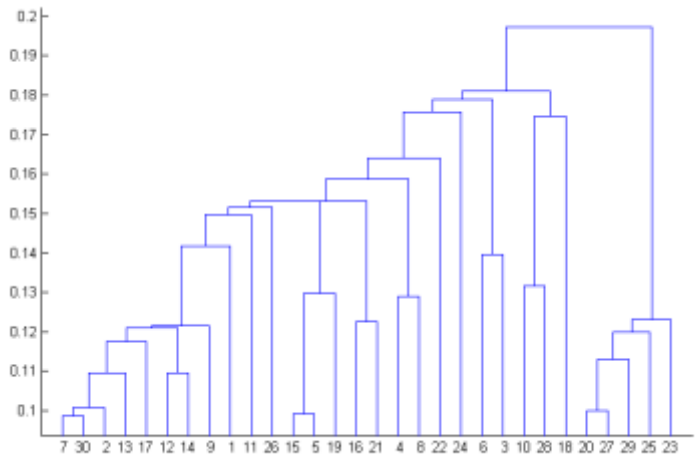
  - And WCV?
  - We use **SSE** (sum of squared distances between a sample in C_i and its centroid M_i, over all clusters)
    - „Error“ is the distance between sample and centroid

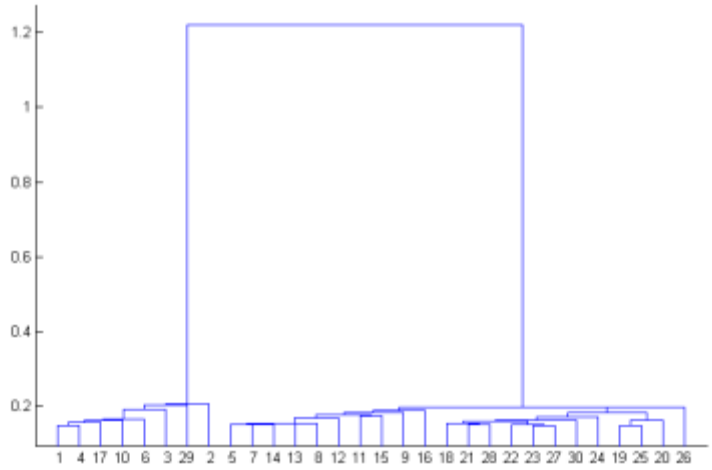
$$WCV = SSE = \sum_{i=1}^{k} \sum_{s \in C_i} D(s, M_i)^2$$

Evaluating with Dendrograms /1







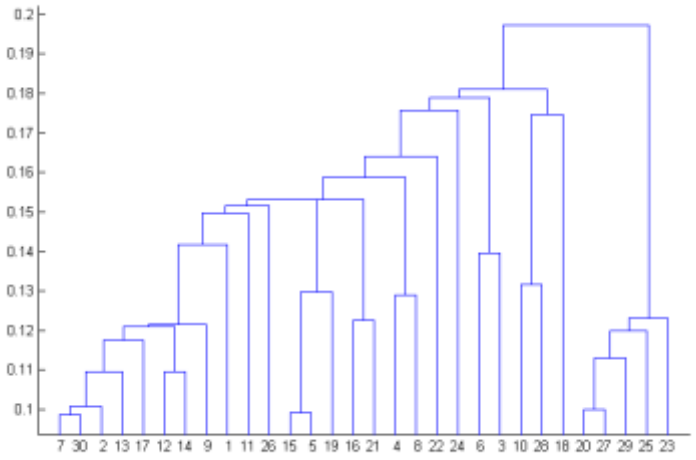


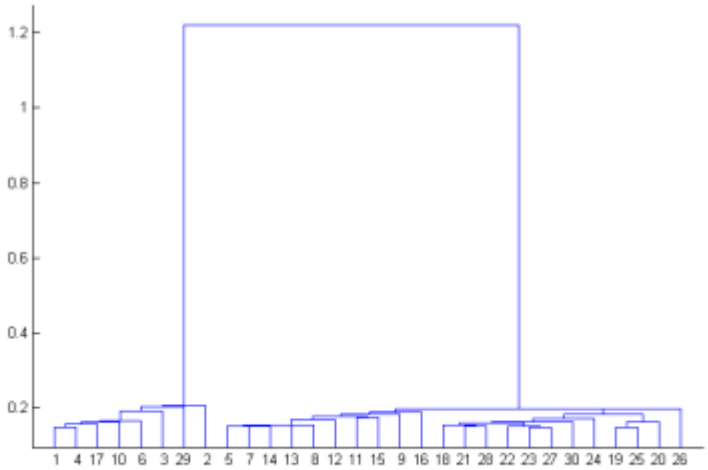
Samples are hard to separate

Samples are well separable

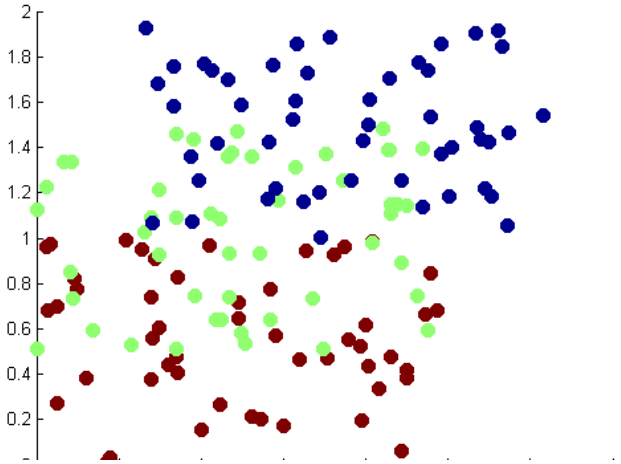
Evaluating with Dendrograms /2

- Samples are hard to separate
  - Tree is „uniformly distributed“ at the y-axis
  - I.e. small incremental distances between merge levels
- Samples are well separable
  - Tree is „dense“ at bottom and „thin“ at the top

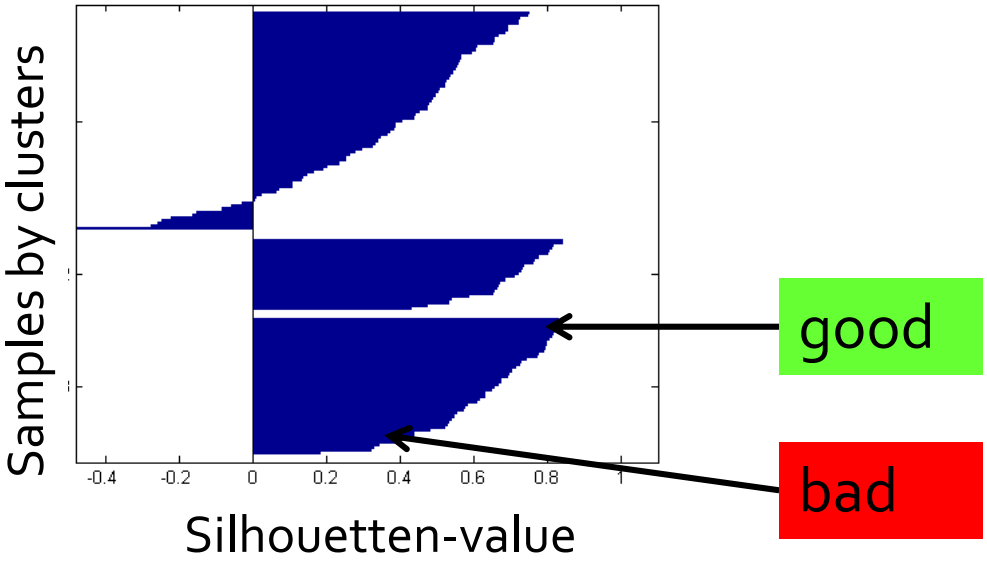




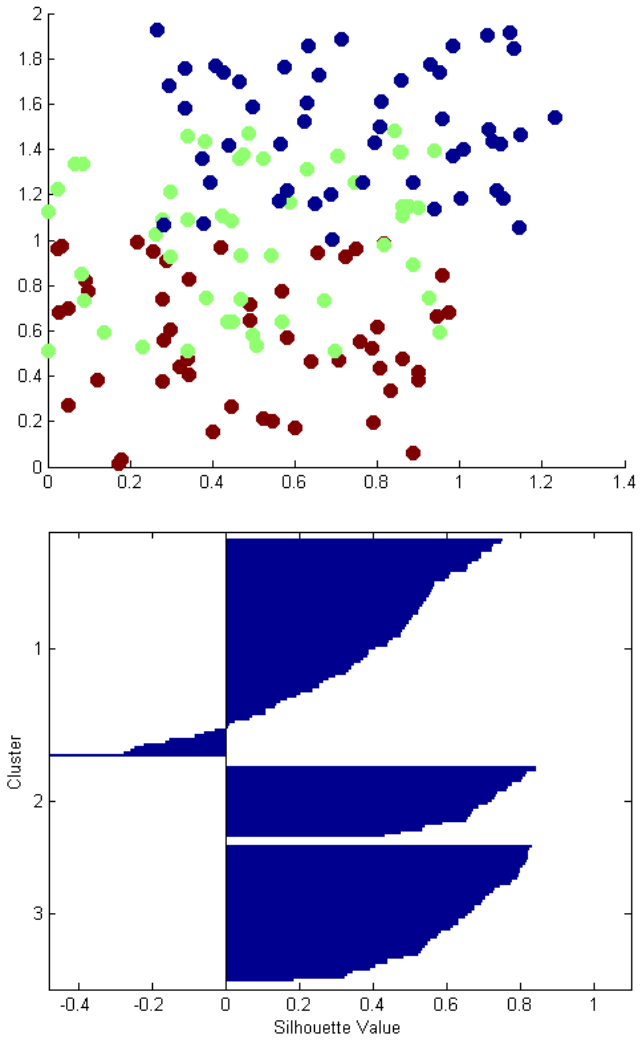
Evaluating with Silhouetten-Plots



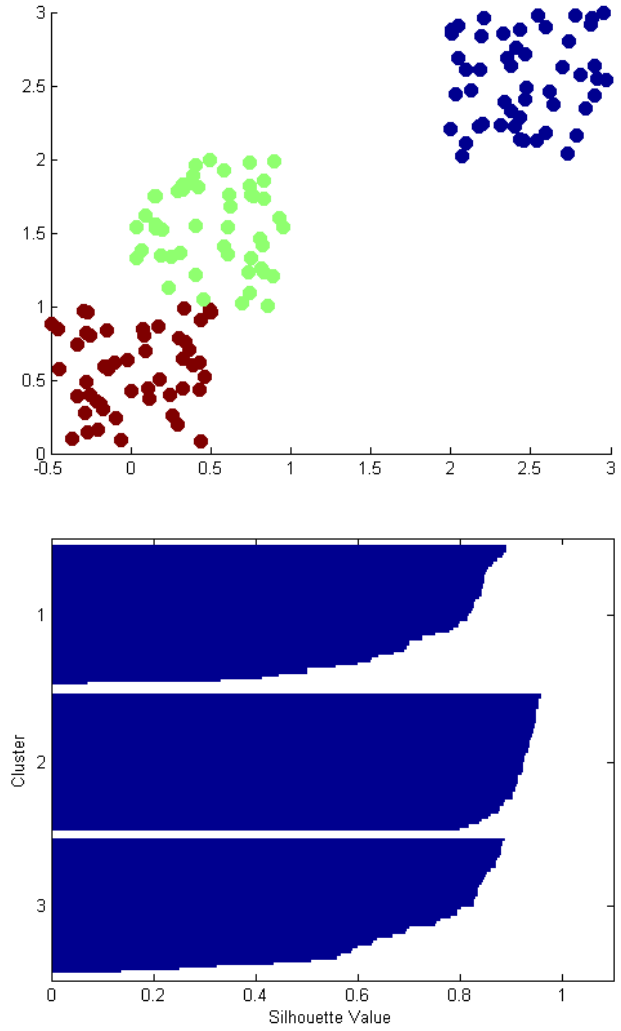
Essentially: a large x-value for sample s tells us that s is far away from other („foreign“) clusters



Evaluating with **Silhouetten**-Plots



Samples are hard to separate



Samples are well separable

Crash Course on NumPy

What is NumPy?

- **NumPy**: an extension of the programming language Python
  - Initiated in 2005 by Jim Hugnin (Link)
- Provides efficient data structures and operations for numerical matrices
  - Technically: C-code with Python-wrappers => fast!
- Syntax/programming: similar to Matlab
- Base for further extensions like SciPy, matplotlib or Pandas

NumPy /1

NumPy

from numpy import *

- a = array ([[1.0, 2.0],
- … [3.0, 4.0]])
- b = arange(10)
- c = zeros(4)
- d = zeros((2, 3))
- d = zeros(6).reshape(2,3)
- e = ones((3,4))

- f = linspace(0, 2, 5)
  - => array ([0., 0.5, 1., 1.5, 2.])

- a = matrix with rows [1.0, 2.0] and [3.0, 4.0]
- b = vector 0,1,…,9
- c = row vec. with 4 zeros
- d = a 2-by-3 matrix with zeros
- e = a 3-by-4 matrix with 1s

- f = 5 „equidistant“ numbers between 0 and 2

NumPy /2

NumPy: Operators work element-wise

- g = 3 * ones((2, 2))
- h = 2 * ones((2, 2))
- print g * h
  - => array ([[6., 6.] , [6., 6.]])
- Matrix-mult: **dot**( )
- print **dot**(g, h)
  - => array([[12., 12.],
  - [12., 12.]])
- print sqrt(g), exp(h)

- g = a 2-by-2 matrix of 3.0s
- h = a 2-by-2 matrix of 2.0s
- „*“ multiplies matrices element-wise!
- dot(g, h) yields the „classical“ matrix multiplication

- sqrt(g): square root, exp(h): e^h

NumPy /3

NumPy: Indexing

a = linspace(10, 60, 6)

a[**0**] => 10	a = 10, 20, 30, .., 60
a[**-1**] => 60	First index is „0“
a[**3:5**] = 33	„-j“ is j-th el. from the end
a[a > 30] = 1.0	Set 40, 50, 60 to 33
a[**:**] = 2.0	Set all el‘s > 30 to 1.0
a.fill(0.0)	Set all el‘s to 2.0
a.**shape** = (2, 3)	Set all el‘s to 0.0
print a[:, 1] => 2nd col	Change a to a 2-by-3 matrix
print a[1, [0, 2]]	Gives column with index 1
a[:,0] = 0.0	Gives two el’s from 2nd row
	Set whole first col to 0.0

NumPy – Useful Idioms

- In-place operations (*=, -=, /=, +=, **=)
  - a = 3*a -1 creates 2 temp arrays (3*a, 3*a-1)
  - Faster and saves memory: a ***=** 3; a **-=** 1;
  - Alternative: a[:] = 3*a -1 (a = 3*a -1 creates a new object)
- Dimension-related operations (let be a = zeros([2, 3]))
  - a.ndim => 2 (number of dimensions)
  - a.shape => (2, 3) (tuple with all dimensions)
  - a.size => 6 (number of elements)
  - a.sum (axis = 0) => sum of each column elem‘s (=1 for rows)
- Creating a matrix with a function
  - def myfun (i, j):
    - return (i+1)*(j+4-i)
  - a = **fromfunction** (myfun, (3, 6)) => a 3-by-6 matrix

NumPy vs. Matlab /1

NumPy

from numpy import *

- a = array ([[1.0, 2.0],
- … [3.0, 4.0]])
- b = zeros(4)
- c = arange(10)
- d = zeros((2, 3))
- d = zeros(6).reshape(2,3)
- e = ones((3,4))
- f = linspace(0, 2, 5)
  - => array ([0., 0.5, 1., 1.5, 2.])

Matlab

- a = [1 2; 3 4];
- b = zeros(1,4);
- c = 0:9;
- d = zeros(2,3);
- e = ones(3,4);
- f = linspace(0,2,5)

NumPy vs. Matlab /2

NumPy: Operators work element-wise

- g = 3 * ones((2, 2))
- h = 2 * ones((2, 2))
- print g * h
  - => array ([[6., 6.] , [6., 6.]])
- Matrix-product: **dot**( )
- print **dot**(g, h)
  - => array([[12., 12.],
  - [12., 12.]])
- print sqrt(g), exp(h)

Matlab: Operators work vector-wise

- g = 3 .* ones(2, 2);
- h = 2 .* ones(2, 2);
- display g .* h
  - …
- Matrix-product: *
- display g * h
  - …
  - In ML: **dot**() scalar-product!
- display sqrt(g), exp(h)

NumPy vs. Matlab /3

NumPy: Indexing

a = linspace(10, 60, 6)

- a[**0**] => 10
- a[**-1**] => 60
- a[**3:5**] = -1
- a[a < 0] = 1.0
- a[**:**] = 2
- a.fill(0)
- a.**shape** = (2, 3)
- print a[:, 1] => 2nd col
- print a[1, [0, 2]]
- a[:,0] = 0

Matlab: Indexing

- a = 10:10:60;
- a(**1**) => 10;
- a(**end**) => 60
- a(**4:6**) = -1;
- a (a < 0) = 1;
- a(**:**) = 2;
- --
- a = **reshape**(a, 2, 3);
- a(:, 2)
- a(2, [1 3])
- a(:,1) = zeros(2,1);

Thank you.

Questions?