Datavarehus Øving 3

Alexander Fredheim
TDT4300 - Datavarehus og Datagruvedrift
NTNU

06.03.2020

2

a)

Explain the Hierarchical Agglomerative Clustering (HAC) and the difference between MIN-link and MAX-link.

Hierarkisk agglomerativ klynging (HAC) er en klyngingsmetode der vi først ser på hvert eneste datapunkt som en egen klynge. Deretter ser vi på alle klyngene våres og slår sammen de to "likeste" eller "nærmeste" klyngene. Etter sammenslåingen har vi nå et nytt sett med "n-1" klynger og vi gjennomfører prosessen rekursivt til vi kun gjenstår med alle punktene i én klynge. Gitt at minst en klynge i ett "klynge-par" har mer enn ét datapunkt i settet vil interne datapunkter i en klynge har forskjellige distanser eller "nærheter" til interne datapunkter i andre klynger. Det er her MIN-link og MAX-link kommer inn.

MIN-link slår sammen to klynger ved å se på alle klynger i settet og velge å slå sammen to klynger der vi finner den **laveste distansen**, gitt at alle klynger velger sine *mest* optimale interne datapunkter.

MIN-link slår sammen to klynger ved å se på alle klynger i settet og velge å slå sammen to klynger der vi finner den **laveste distansen**, gitt at alle klynger velger sine *minst* optimale interne datapunkter.

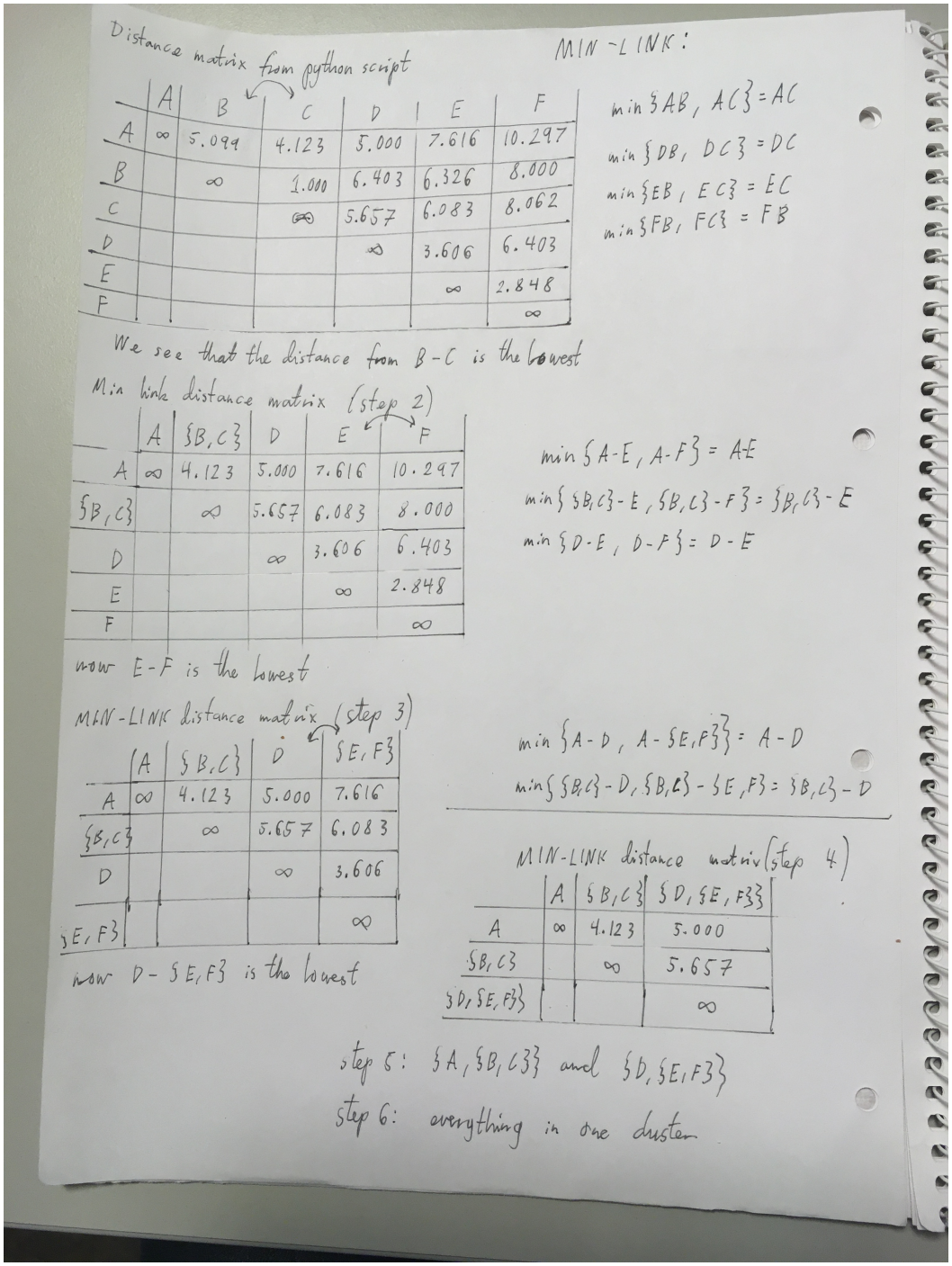
b)

Under er prosessen forklart steg for steg. Distanse-matrisen er generert via python, der koden er å finne i jupyter dokumentet som er lagt ved i øvingen. Vi bruker grådighetsegenskapen i forholdet mellom klyngene og forholder oss til oppdaterte tall for sammenligning hver gang vi er et skritt videre

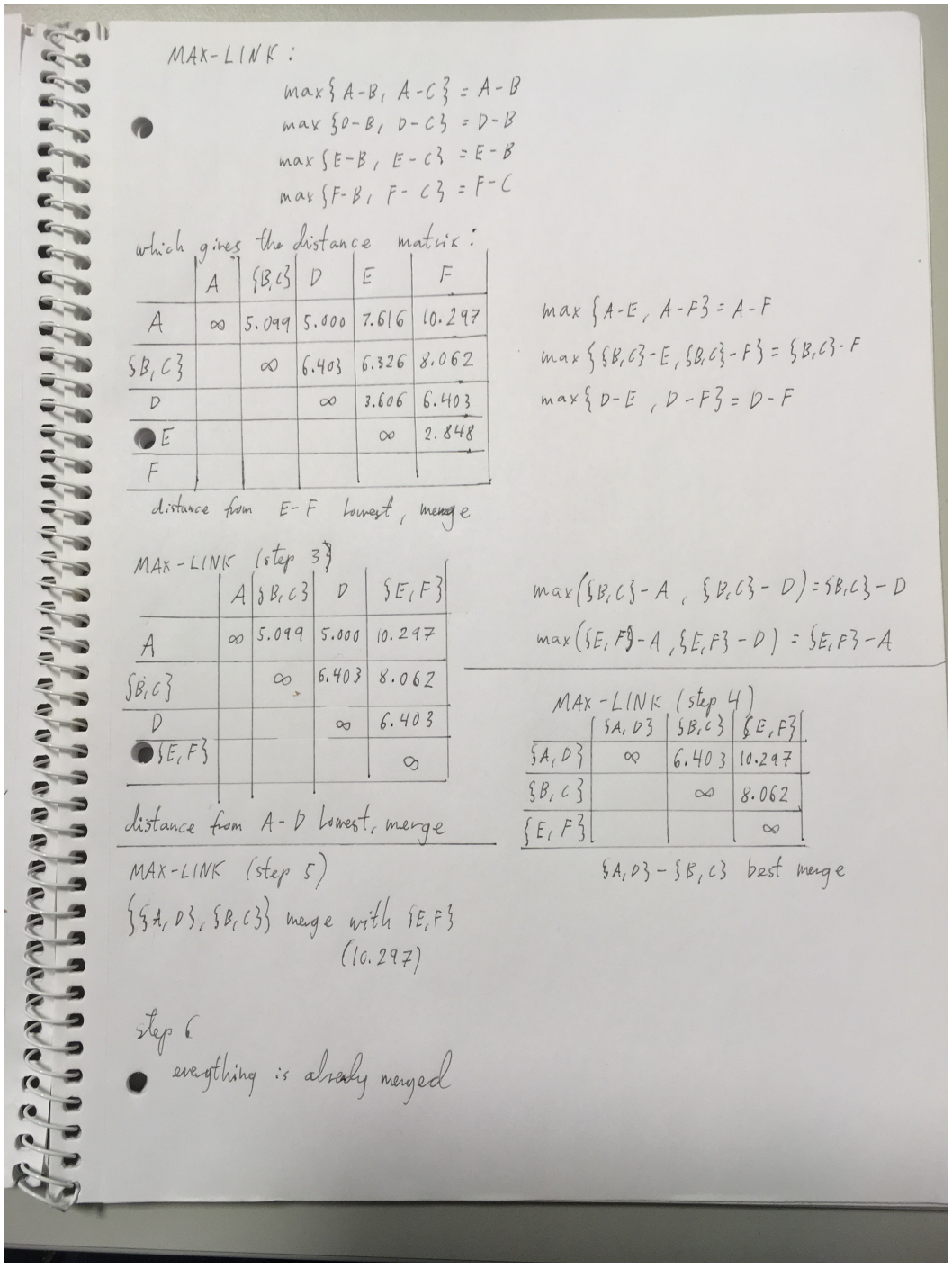
1. laveste distanse er B-C, ser på alle distanser fra de andre datapunktene til nodene B og C, og sammenligner parvis hvilket av de to valgene som vil gi Min-Link, altså kortest distanse mellom datapunkter i klyngene.

2. vi oppdaterer tabellen med de oppdaterte lokalt-beste klyngedistansene fra alle andre noder til vår nye klynge {B,C}-klyngen. Ser nå at laveste distance er E-F. gjør tilsvarende som for forrige punkt for {E,F}-klyngen

3. vi fortsetter denne prosessen for de gjenstående klyngene, til vi har alle datapunktene representert i én klynge



Her ser vi tilsvarende for Max-link, der vi oppdaterer tabellen med de lokalt lengste distansene mellom datapunkter i to klynger.



MAX-LINK:

$\max\{A-B, A-C\} = A-B$
$\max\{D-B, D-C\} = D-B$
$\max\{E-B, E-C\} = E-B$
$\max\{F-B, F-C\} = F-C$

which gives the distance matrix:

	A	{B,C}	D	E	F
A	∞	5.099	5.000	7.616	10.297
{B,C}		∞	6.403	6.326	8.062
D			∞	3.606	6.403
E				∞	2.848
F					

distance from E-F lowest, merge

$\max\{A-E, A-F\} = A-F$
$\max\{\{B,C\}-E, \{B,C\}-F\} = \{B,C\}-F$
$\max\{D-E, D-F\} = D-F$

MAX-LINK (step 3)

	A	{B,C}	D	{E,F}
A	∞	5.099	5.000	10.297
{B,C}		∞	6.403	8.062
D			∞	6.403
{E,F}				∞

distance from A-D lowest, merge

$\max(\{B,C\}-A, \{B,C\}-D) = \{B,C\}-D$
$\max(\{E,F\}-A, \{E,F\}-D) = \{E,F\}-A$

MAX-LINK (step 4)

	{A,D}	{B,C}	{E,F}
{A,D}	∞	6.403	10.297
{B,C}		∞	8.062
{E,F}			∞

{A,D} - {B,C} best merge

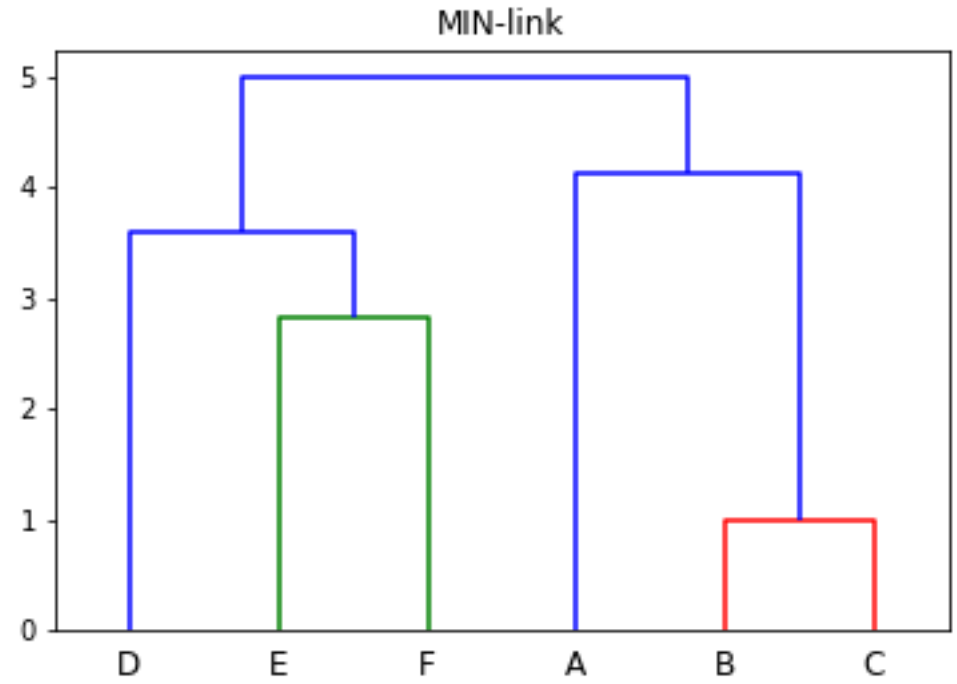
MAX-LINK (step 5)

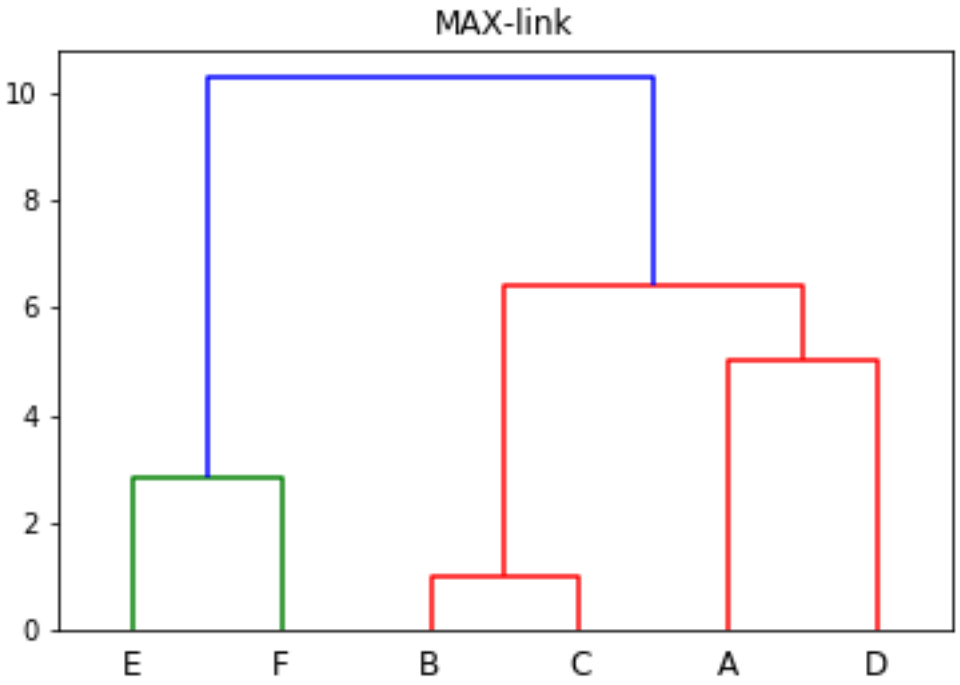
{{A,D}, {B,C}} merge with {E,F}
(10.297)

step 6

everything is already merged

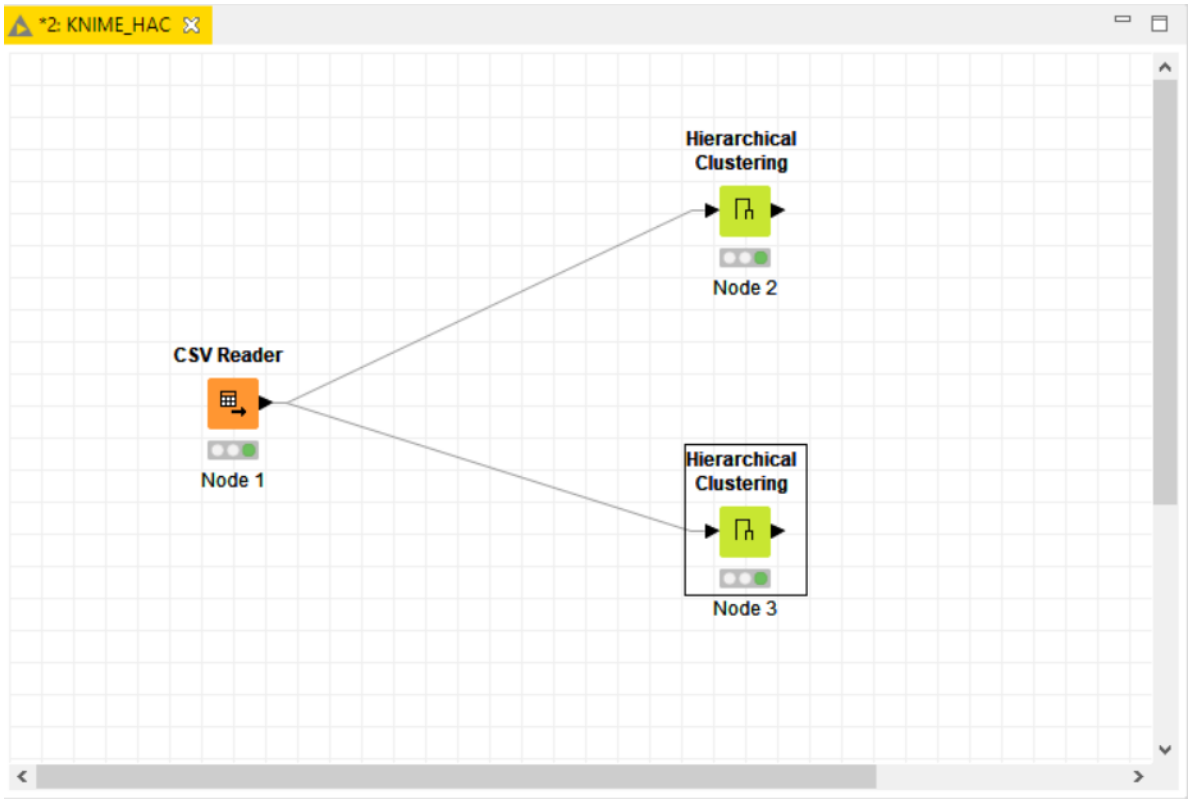
Under ser vi dendogrammer fra både Min-Link og Max-Link. Koden for å generere disse er å finne i Jupyter dokumentet

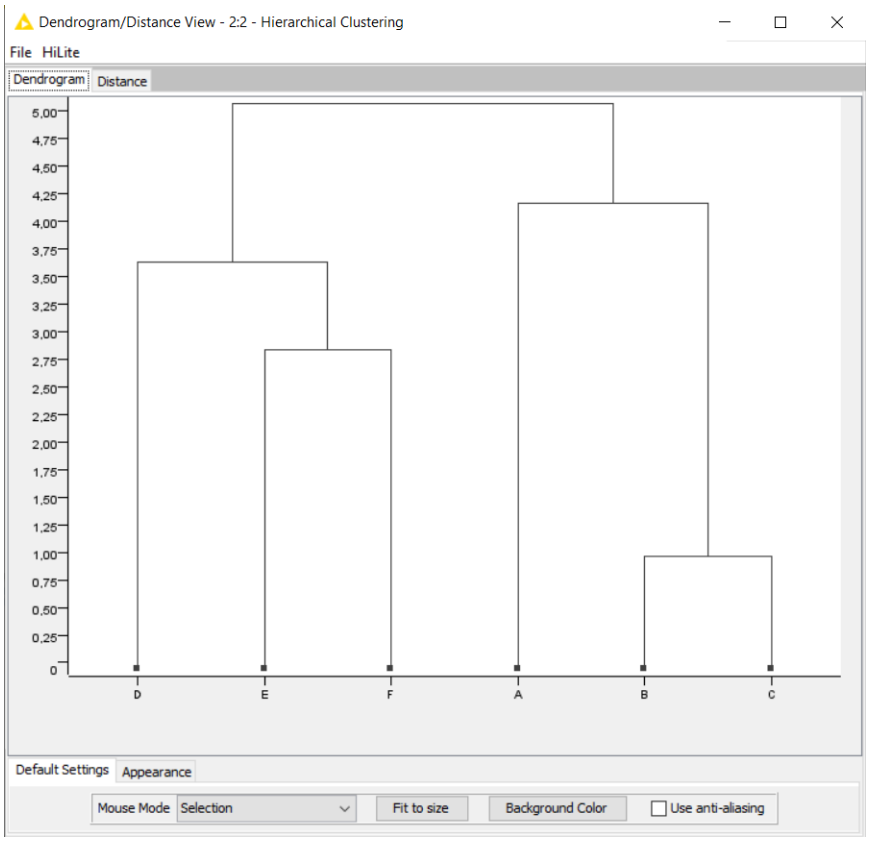


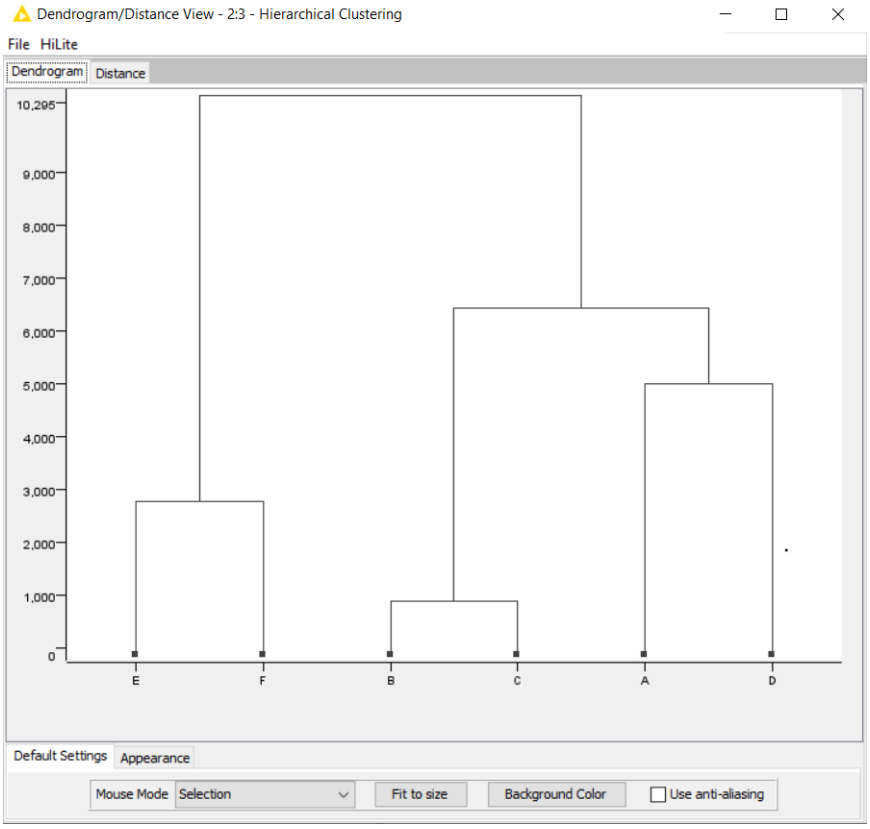


c)

Under ser vi resultater av samme oppgave gjennom bruk av KNIME




Dendrogram/Distance View - 2:2 - Hierarchical Clustering
File HiLite
Dendrogram Distance
5,00
4,75
4,50
4,25
4,00
3,75
3,50
3,25
3,00
2,75
2,50
2,25
2,00
1,75
1,50
1,25
1,00
0,75
0,50
0,25
0
D
E
F
A
B
C
Default Settings Appearance
Mouse Mode Selection
Fit to size
Background Color
Use anti-aliasing


Dendrogram/Distance View - 2:3 - Hierarchical Clustering
File HiLite
Dendrogram Distance
10,295
9,000
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
0
E
F
B
C
A
D
Default Settings Appearance
Mouse Mode Selection
Fit to size
Background Color
Use anti-aliasing

3

a)

```
P_1 = [1,1] # Datapoints
P_2 = [14,8]
P_3 = [6,12]
P_4 = [3,1]
P_5 = [5,11]
P_6 = [13,6]
P_7 = [4,12]
P_8 = [12,8]
P_9 = [1,3]
P_10 = [8,1]
P_11 = [5,9]
P_12 = [10,12]
P_13 = [14,5]
P_14 = [2,4]
P_15 = [8,6]
P_16 = [4,3]
P_17 = [12,5]
P_18 = [14,14]

# generate euclidean distance between all datapoints
def distance_matrix(dataset):
    distance_matrix = []
    for i in range(len(dataset)):
        row = []
        point = np.array(dataset[i])
        for j in range(len(dataset)):
            comp_point = np.array(dataset[j])
            distance = np.linalg.norm(point-comp_point)
            row.append(distance)
        distance_matrix.append(row)
    return distance_matrix

# we use combined data from proximity list and distance matrix to
# recognize what kind of points we have
def identify_border_points(prox_list, dist_matrix, eps):
    border_list = []
    for i in range(len(dist_matrix)):
        # if we know the point is a core point we don't need to check
        if(prox_list[i]>=3):
            border_list.append('C')
        else:
            counter = 0
            for j in range(len(dist_matrix)):
                if(prox_list[j]<3):
                    None
                else:
                    if(dist_matrix[i][j] <= eps):
                        counter += 1
            border_list.append(counter)
    return border_list

# we use the distance matrix and simply count how many instances
# a datapoints euclidean distance to another is less than eps
def proximity_list(matrix, eps):
    prox_list = []
    for row in matrix:
        counter = 0
        for col in row:
            if(col <= eps):
                counter += 1
        prox_list.append(counter)
    return prox_list
```

```
63
64
65
66 eps = 2
67
68 all_points = [P_1,P_2,P_3,P_4,P_5,P_6,P_7,P_8,P_9,
69 P_10,P_11,P_12,P_13,P_14,P_15,P_16,P_17,P_18]
70 dm2 = distance_matrix(all_points)
71
72 prox_list = proximity_list(dm2, eps)
73
74 # prox list produces [3, 2, 3, 2, 4, 3, 3, 2, 3, 1, 2, 1, 3, 2, 1, 1, 3, 1]
75
76 # from looking at the proximity list we have
77 # P_1, P_3, P_5, P_6, P_7, P_9, P_13 and P_17
78 # that have 3 nodes within the proximity required by the given eps (eps=2)
79
80 bord_list = identify_border_points(prox_list,dm2, eps)
81
82 # border list produces ['C', 0, 'C', 1, 'C', 'C', 'C', 0, 'C'
83 #                         , 0, 1, 0, 'C', 1, 0, 0, 'C', 0]
84 # this leaves P_4, P_11 & P_14 as border points
85
86 # this means that P_2, P8, P_10, P_12, P_15, P_16 & P_18 are noice points
87
88 noiceless_list = [P_1, P_3, P_4, P_5, P_6, P_7, P_9, P_11, P_13, P_14, P_17]
89
90
91 dm3 = distance_matrix(noiceless_list)
92
93 # first row (P_1) gives the connections P_1-P_4-P_9
94 # second row (P_3) gives the connections P_3-P_5-P_7
95 # third row (P_4) gives no new connections
96 # fourth row (P_5) gives a new connection in P_5-P_11,
97 # transitively giving us the cluster P_3-P_5-P_7-P_11
98
99 # fifth row (P_6) gives P_6-P_13-P_17
100 # sixth row (P_7) gives no new connections
101 # seventh row (P_9) gives a new connection in P_14,
102 # transitively giving us the cluster P_1-P_4-P_9-P_14
103
104 # eight row (P_11) gives no new connections
105 # ninth row (P_13) gives no new connections
106 # tenth row (P_14) gives no new connections
107
108 # we end up with the clusters {P_1-P_4-P_9-P_14}, {P_3-P_5-P_7-P_11}, {P_6-P_13-P_17}
```

Listing 1: Python code

Printing options for this code is in the jupyter document

b)

Under ser vi resultater av samme oppgave gjennom bruk av KNIME

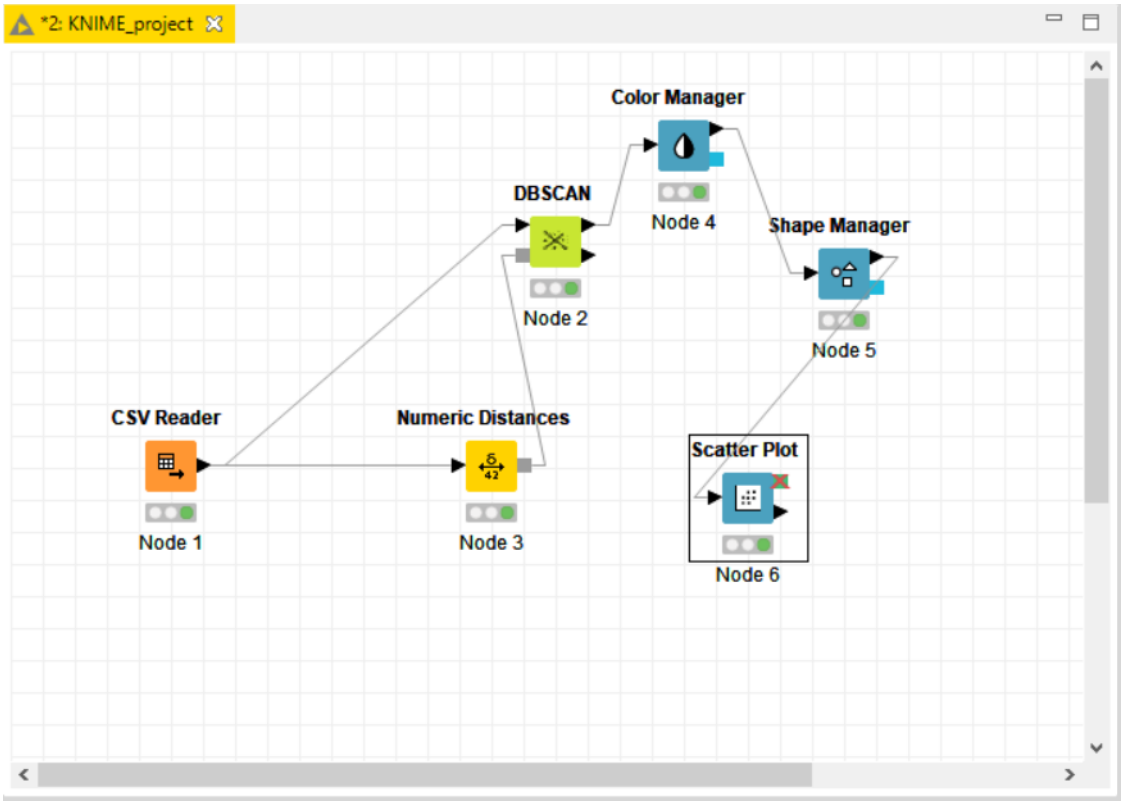
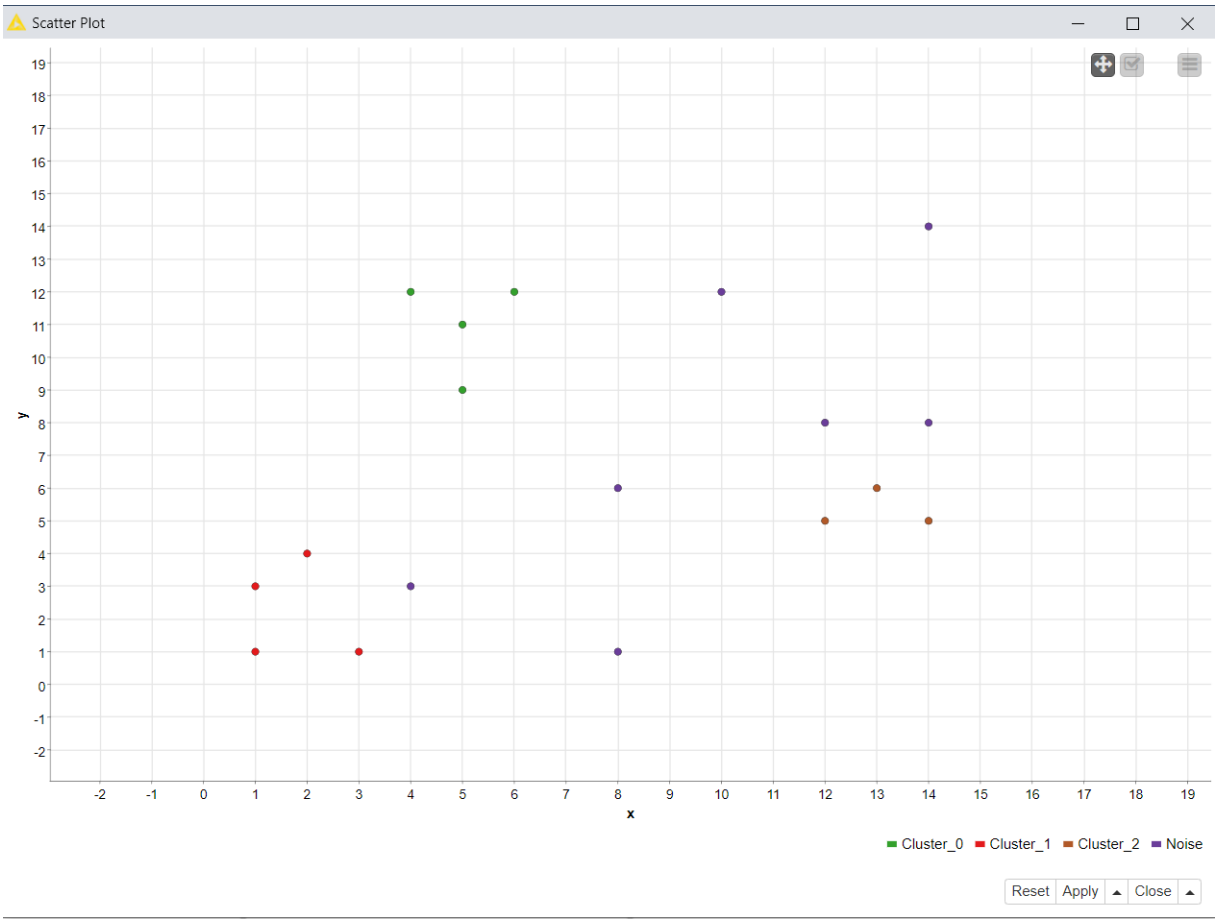


Table "default" - Rows: 18 | Spec - Columns: 4 | Properties | Flow Variables

Row ID	x	y	Cluster	Selecte...
P1	1	1	Cluster_1	false
P2	14	8	Noise	false
P3	6	12	Cluster_0	false
P4	3	1	Cluster_1	false
P5	5	11	Cluster_0	false
P6	13	6	Cluster_2	false
P7	4	12	Cluster_0	false
P8	12	8	Noise	false
P9	1	3	Cluster_1	false
P10	8	1	Noise	false
P11	5	9	Cluster_0	false
P12	10	12	Noise	false
P13	14	5	Cluster_2	false
P14	2	4	Cluster_1	false
P15	8	6	Noise	false
P16	4	3	Noise	false
P17	12	5	Cluster_2	false
P18	14	14	Noise	false


Scatter Plot
y
x
Cluster_0
Cluster_1
Cluster_2
Noise
Reset
Apply
Close