CODE CAMP'18

17 November 2018 | FON University, Skopje

OUR PARTNERS

PLATINUM

GOLD

SILVER

BRONZE

MEDIA PARTNER

SUPPORTING PARTNERS

Ве молиме исклучете ги мобилните уреди

Please turn off your mobile devices

Markov Cluster Algorithm

implementation in Python and real-world application

Andjela Todorovic, Createsi GmbH

CODE CAMP'18
17 November 2018 | FON University, Skopje

macedonian.net user group
The Ultimate .NET User Group and Developers Association
10 YEAR ANNIVERSARY

- *Natural cluster*

- Presence of many edges between the members of the cluster

- High number of higher-length (longer) paths between two arbitrary nodes.

Why? (node pairs lying in different natural clusters)

- Random walks on the graph infrequently go from one natural cluster to another.

- *MCL Algorithm*

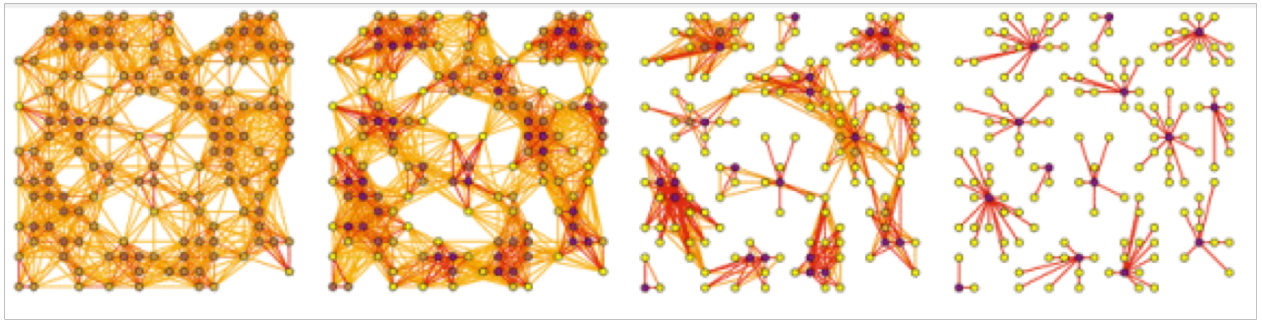
- Finds cluster structure in graphs by a mathematical bootstrapping

- Bootstrapping (*"To lift himself up by his bootstraps."*)

  → *resampling the sample data*

- The process deterministically computes (the probabilities of) random walks through the graph

- Two operators transforming one set of probabilities into another



- *Random walks* ("drunkard's walk")

• The next value in the sequence is a modification of the previous value in the sequence.

• Provides some consistency from step-to-step rather than the large jumps that a series of independent, random numbers provides

• A simple model of a random walk is as follows:

1. Start with a random number of either -1 or 1.

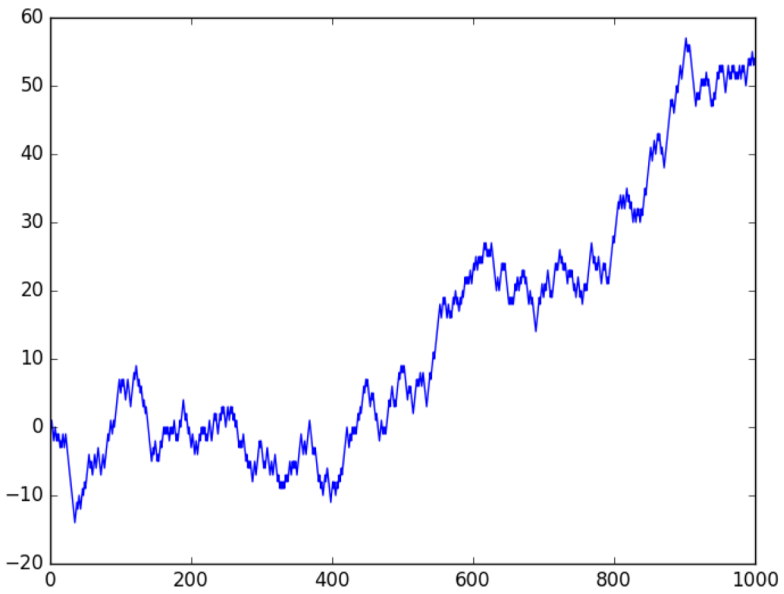
2. Randomly select a -1 or 1 and add it to the observation from the previous time step.

3. Repeat step 2 for as long as you like.

$$y(t) = B0 + B1*X(t-1) + e(t)$$

Random walks

```
from random import seed
from random import random
from matplotlib import pyplot
seed(1)
random_walk = list()
random_walk.append(-1 if random() < 0.5 else 1)
for i in range(1, 1000):
            movement = -1 if random() < 0.5 else 1
            value = random_walk[i-1] + movement
            random_walk.append(value)
pyplot.plot(random_walk)
pyplot.show()
```



MCL Algorithm

- Alternation of two operators called expansion and inflation

- Expansion coincides with taking the power of a stochastic matrix using the normal matrix product (i.e. matrix squaring)

- Inflation corresponds with taking the Hadamard power of a matrix (taking powers entrywise), followed by a scaling step, such that the resulting matrix is stochastic *again (the matrix elements (on each column) correspond to probability values.*)

MCL Algorithm

- A column stochastic matrix is a non-negative matrix with the property that each of its columns sums to 1.

- Let M be column stochastic matrix and r > 1 a real number

- The column stochastic matrix resulting from inflating each of the columns of M with power coefficient r is written $\Gamma_r(M)$ (inflation operator with power coefficient r.)

- Write $\Sigma_{r,j}(M)$ for the summation of all the entries in column j of M raised to the power r (sum after taking powers)

- Then $\Gamma_r(M)$ is defined in an entrywise manner by setting

$\Gamma_r(M_{ij}) = M_{ij}^r / \Sigma_{r,j}(M)$

MCL Algorithm

- Each column j of a stochastic matrix M corresponds with node j of the stochastic graph associated with M.

- Row entry i in column j (i.e. the matrix entry M_{ij}) corresponds with the probability of going from node j to node i.

- It is observed that for values of r > 1, inflation changes the probabilities associated with the collection of random walks departing from one particular node (corresponding with a matrix column) by favoring more probable walks over less probable walks.

MCL Algorithm

- Expansion corresponds to computing random walks of higher length, which means random walks with many steps.

- It associates new probabilities with all pairs of nodes, where one node is the point of departure and the other is the destination.

- Since higher length paths are more common within clusters than between different clusters, the probabilities associated with node pairs lying in the same cluster will, in general, be relatively large as there are many ways of going from one to the other.

- Inflation will then have the effect of boosting the probabilities of intra-cluster walks and will demote inter-cluster walks.

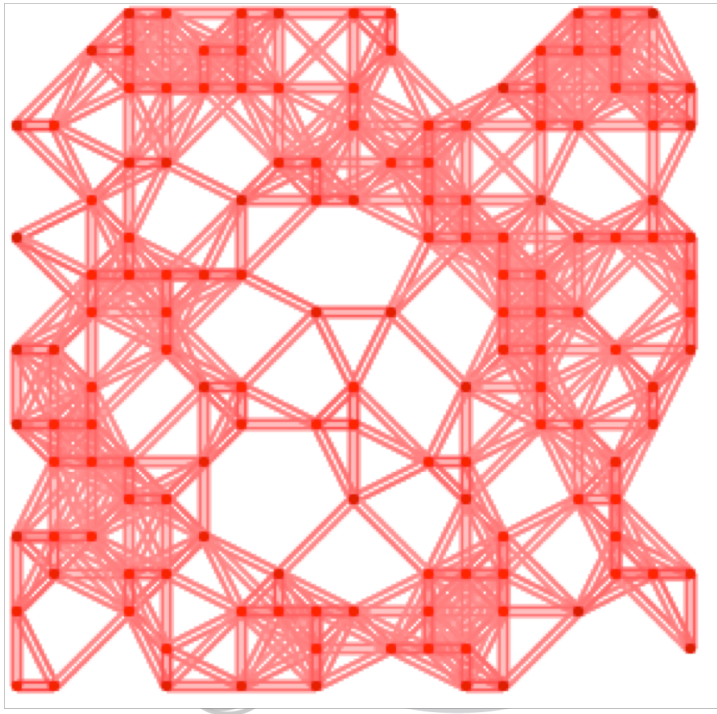
- This is achieved without any a priori knowledge of cluster structure

MCL Algorithm

- Eventually, iterating expansion and inflation results in the separation of the graph into different segments.

- There are no longer any paths between these segments and the collection of resulting segments is simply interpreted as a clustering.

- The inflation operator can be altered using the parameter r. Increasing this parameter has the effect of making the inflation operator stronger, and this increases the granularity or tightness of clusters.

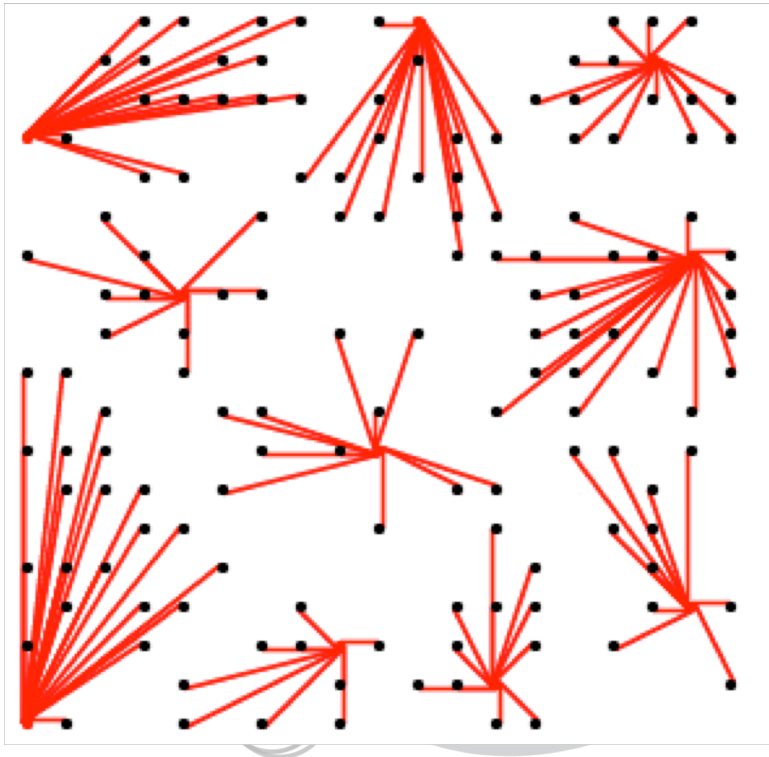


MCL Algorithm

- With this, the MCL algorithm can be written as G is a graph add loops to G affects granularity set M_1 to be the matrix of random walks on G while (change)

{ M_2 = M_1 * M_1 (expansion M_1 = Γ(M_2)) inflation change = difference(M_1, M_2) }

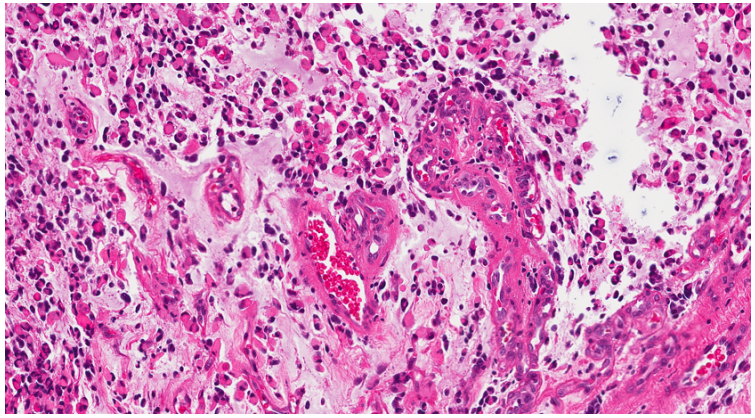
- Set CLUSTERING as the components of M_1



Tissue segmentation

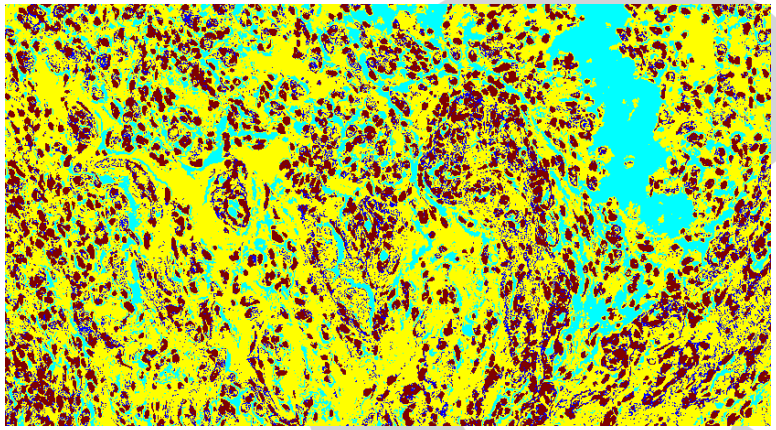
real world application

- Entry dataset consists of the images of the tissue including cytoplasm, cytoplasmic organelles and nucleus.



- The result should be the clustered image where each one of the elements is colored differently, precisely:

Cytoplasm – yellow, Nucleus – brown, Background- blue



The solution

- Generally speaking, any image can be processed as either a function or a graph

- Graph based image processing methods typically operate on pixel adjacency graphs, i.e., graphs whose vertex set is the set of image elements, and whose edge set is given by an adjacency relation on the image elements.

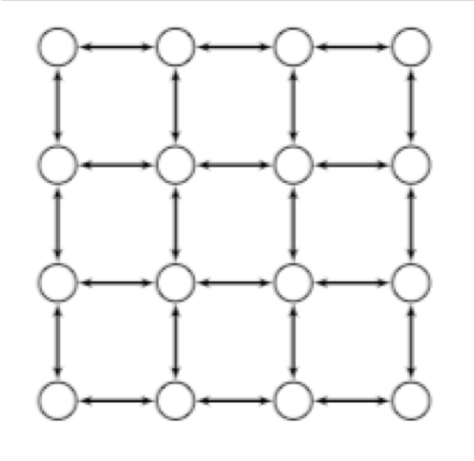
- Commonly, the edge set is defined as all vertices v,w such that $d(v,w) \leq \rho$.

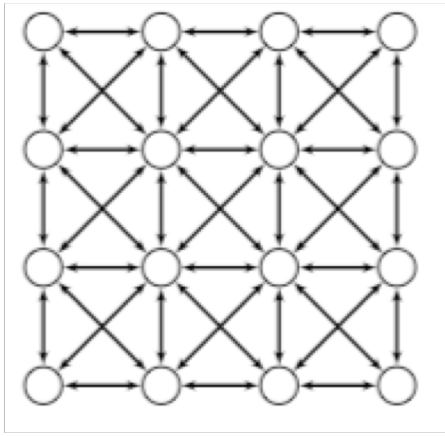
This is called the Euclidean adjacency relation.

- *Connected component labeling*

• Connected components labeling scans an image and groups its pixels into components based on pixel connetivity,

• All pixels in a connected component share similar pixel intensity values and are in some way connected with each other. Once all groups have been determined, each pixel is labeled with a graylevel or a color (color labeling) according to the component it was assigned to.





```
def connected_component_labelling(bool_input_image,
connectivity_type=CONNECTIVITY_8):


if connectivity_type !=4 and connectivity_type != 8: raise
ValueError("Invalid connectivity type (choose 4 or 8)") image_width
= len(bool_input_image[0]) image_height = len(bool_input_image)


labelled_image = np.zeros((image_height, image_width),
dtype=np.int16) uf = UnionFind() # initialise union find data
structure current_label = 1
```

```
if (connectivity_type == CONNECTIVITY_4) or (connectivity_type ==
CONNECTIVITY_8):

# West neighbour

if x > 0: west_neighbour = image[y,x-1]

if west_neighbour > 0: labels.add(west_neighbour)


# North neighbour

if y > 0: north_neighbour = image[y-1,x]

if north_neighbour > 0: labels.add(north_neighbour)


if connectivity_type == CONNECTIVITY_8:

# North-West neighbour

if x > 0 and y > 0: northwest_neighbour = image[y-1,x-1]

if northwest_neighbour > 0:labels.add(northwest_neighbour)


# North-East neighbour

if y > 0 and x < len(image[y]) - 1:northeast_neighbour = image[y-
1,x+1]

if northeast_neighbour > 0: labels.add(northeast_neighbour) else:
print("Connectivity type not found.") return labels
```

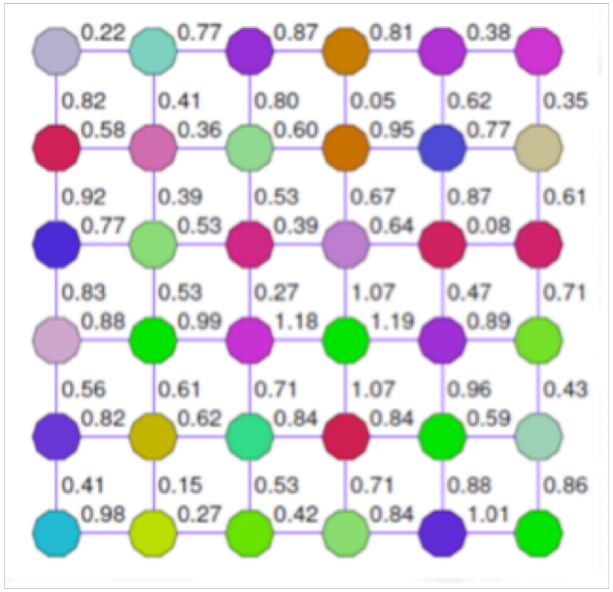
```
def connected_component_labelling(bool_input_image,
connectivity_type=CONNECTIVITY_8):

if connectivity_type !=4 and connectivity_type != 8: raise
ValueError("Invalid connectivity type (choose 4 or 8)")
image_width = len(bool_input_image[0]) image_height =
len(bool_input_image)

labelled_image = np.zeros((image_height, image_width),
dtype=np.int16) uf = UnionFind() # initialise union find data
structure current_label = 1
```

Advanced problem solution:

- Instead of taking into consideration the distance between the nodes for labeling weather there exists an edge between them, we can state that each node is somehow connected ie. there exists either four or eight connectivity.
- In this case we cannot apply CCL, so instead we use Markov cluster, as shown.



```python
class Node(object):

    def __init__(self, value):
        self.value = value
        self.neighbors = {}
        self.inDegree = 0
        self.outDegree = 0

    def updateEdgeWeight(self, key):
        pass

    def getEdgeWeight(self, key):
        if self.isConnectedTo(key):
            return self.neighbors[key
        return None

    def connectTo(self, key, weight):
        self.neighbors[key] = weight

    def isConnectedTo(self, key):
        return key in self.neighbors
```

```python
class Graph(object):
    def __init__(self, directedGraph=False):

        self.directedGraph = directedGraph
        self.nodes = {}
        self.nodesCount = 0
        self.edgeCount = 0

    def addNode(self, key, value):

        if not self.hasNode(key):
            self.nodes[key] = Node(value)
            self.nodesCount += 1

    def addEdge(self, key1, key2, weight=1):

        if not self.hasNode(key1) or not self.hasNode(key2):
            return None
        if not self.hasEdge(key1, key2):
            self.nodes[key1].connectTo(key2, weight)
            self.edgeCount += 1

        if not self.directedGraph:
            if not self.hasEdge(key2, key1):
                self.nodes[key2].connectTo(key1, weight)
```

```
def updateNodeValue(self, key):
    pass
def updateEdge(self, key1, key2, newWeight):
    pass
def removeNode(self, key):
    pass
def removeEdge(self,node1,node2):
    pass


def getNodeValue(self, key):

    if (self.hasNode(key)):
        return self.nodes[key].value
    return None

def getEdgeWeight(self, key1, key2):

    if self.hasNode(key1):
        return self.nodes[key1].getEdgeWeight(key2)
    return None
```

```
def getAllNodes(self):

    res = []
    for k in self.nodes:
        res.append((k,self.getNodeValue(k)))
    return res

def getAllEdges(self):
    res = []
    for k in self.nodes:
        for k1 in self.nodes[k].neighbors:
            if self.directedGraph:
                res.append((k,k1))
            else:
                if (k1, k) not in res:
                    res.append((k, k1))
    return res
```

```
def hasNode(self, key):

    return key in self.nodes

def hasEdge(self, key1, key2):

    if self.hasNode(key1):
        return self.nodes[key1
    return False

def nodesCount(self):
    return self.nodesCount

def inDegree(self, key):
    pass
def outDegree(self, key):
    pass


# Algoritmi
```

```
def getGraphMatrix(self):
    mapKeysToIndexes = {}
    mapIndexesToKeys = [None] * self.nodesCount
    c = 0
    for k in self.nodes:
        mapKeysToIndexes[k] = c
        mapIndexesToKeys[c] = k
        c += 1

    matrix = [[0 for _ in range(self.nodesCount)]
            for _ in range(self.nodesCount)]
    for k in self.nodes:
        for k1 in self.nodes[k].neighbors:
            matrix[mapKeysToIndexes[k]][mapKeysToIndexes[k1]] = \
                    self.nodes[k].neighbors[k1]

    return (matrix, mapIndexesToKeys)

    s = ""
    for n1 in self.nodes:
        s += str(n1) + ": "
        for n2 in self.nodes[n1].neighbors:
            s+= str(n2) + "->"
        s += "\n"
    return s
```

```
import numpy

class MarkovClustering(object):

    def __init__(self, matrix, e, r):

        self.matrix = numpy.array(matrix,dtype=numpy.float64)
        self.e = e
        self.r = r

    def computeClusters(self, T = 100):

        self.addSelfLoops()
        self.normalizeColumns()

        t = 0
        while(t<T):
            lastMatrix = numpy.copy(self.matrix)
            self.powerStep()
            self.inflationStep()
            if self.steadyState(lastMatrix)==True:
                break
            t += 1

        return self.interpretClusters()
```

```
def addSelfLoops(self):

    for i in range(self.matrix.shape[0]):
        self.matrix[i][i] = 1

def normalizeColumns(self):

    s = self.matrix.sum(axis=0)
    for (x,y), _ in numpy.ndenumerate(self.matrix):
        if s[y] != 0:
            self.matrix[x][y] /= float(s[y])

def powerStep(self):
    temp = self.matrix
    for _ in range(self.e-1):
        temp = temp.dot(self.matrix)
    self.matrix = temp

def inflationStep(self):
    self.matrix **= self.r
    self.normalizeColumns();

def steadyState(self, lastMatrix):
    for (x,y), _ in numpy.ndenumerate(self.matrix):
        if self.matrix[x][y]-lastMatrix[x][y] != 0:
            return False
    return True
```

```
def interpretClusters(self):
    res = []
    for i in range(self.matrix.shape[0]):
        cluster = []
        flag = 0
        for z in range(self.matrix.shape[0]):
            if self.matrix[i][z] > 0:
                cluster.append(z)
                flag = 1
        if flag==1:
            res.append(cluster)
    return res
```

Thanks!

Any Questions?

www.createsi.ai
github.com/kobrica/MarkovCluster

linkedin.com/andjelatodorovich
facebook.com/andjelatodorovich
instagram.com/andjelatodorovich