COMPUTATIONAL INTELLIGENCE

Assignments of Associative Systems

Adrian Horzyk

horzyk@agh.edu.pl

AGH University of Science and Technology Krakow, Poland

Assignment

Implement a chosen associative structure (AGDS, MAGDRS) and use it to a selected subgroup (choose at least one of the following tasks) of CI problems:

- Recommendation system.
- Searching for similar objects to any given input context (data).
- Searching for similar objects to a given group of the other objects, computing the most similar objects to all of them as an aggregative association calculated on the basis of the individual similarities to each of these objects (the calculated values are summed up in the nodes).
- Find missing values of attributes for a given subset of input features.
- Implement an associative classifier based on the associations between objects and classes stored in the network.
- Implement a clustering based on the AGDS structure and connections representing similarities and accelerating the search for the clusters.
- Implement inferences on an associatively transformed database.

Implementation of AGDS

1. Try to use numpy, vectors, and parallelism where possible.
2. Assuming that we usually work on the chosen static data, we do not always need to work on the dynamic data structures like lists but we can use constant-size structures like arrays.
3. Choose the structure for representing attribute data and mechanism for searching for them: sorted arrays, sorted lists, dictionaries, hash-tables, AVB+trees etc.
4. The graph structure can be implemented using a list of edges (list or array), an adjacency matrix (2D array), an incidence matrix (2D array) or an adjacency list (an array of lists, a list of arrays, an array of arrays, or a list of lists).

Graph as an Adjacency Matrix

The adjacency (neighborhood) matrix is a square (2D) matrix with a degree equal to the number of vertices of the graph.

Each vertex of the graph is indexed by one of the consecutive numbers from 0 to N-1, where N specifies the number of vertices of such graph. You can also use labels as presented below.

The row indexes of the adjacency matrix correspond to the indexes of the start vertices v_i.

Adjacency matrix column indexes correspond to end point indexes v_j.

Each cell of such a matrix represents one possible connection between the vertices v_i and v_j :

- 0 - means no edge
- 1 - means existing edge or w - means weight of the edge

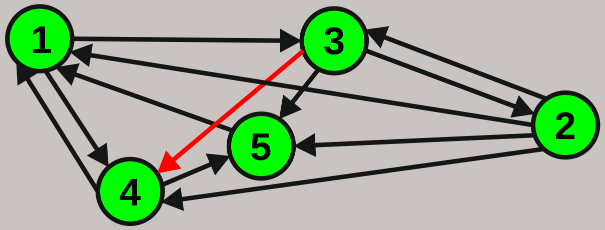
The i-th row of such matrix is a list of vertex indexes to which the edges from the vertex with the index i lead to, when the values in the cell are nonzero.

3	0	1	0	1	1

The j-th column of such matrix is a list of vertex indexes from which edges lead to the vertex with index j when the values in the cell are non-zero.

3
1
1
0
0
0

The adjacency matrix for an undirected graph is symmetrical about the main diagonal, because if there is an edge (v_i, v_j), then there is also an edge (v_j, v_i).



INCIDENCE MATRIX		Vj - END VERTEX				
		1	2	3	4	5
Vi - START VERTEX	1	0	0	1	1	0
	2	1	0	1	1	1
	3	0	1	0	1	1
	4	1	0	0	0	1
	5	1	0	0	0	0

Graph as an Adjacency Matrix

The degree and vertex of an undirected graph can be determined by counting the number of non-zero cells in the i-th row of the adjacency (neighborhood) matrix.

The degree and vertex of the directed graph will be determined by counting the number of non-zero cells in the i-th row and j-th column of the adjacency matrix.

We can also separately count the number of outgoing edges (in rows) and the number of incoming edges (in columns).

Indexes of neighbors of the vertex of the i-th can be found in the i-th row if the field value is nonzero.

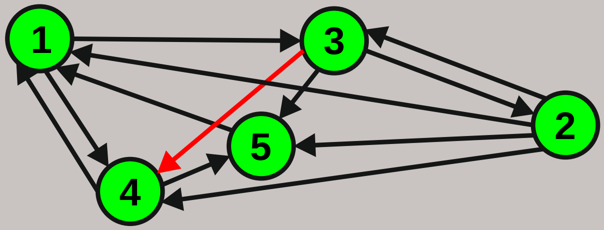
the i-th vertex is isolated if both the i-th row as well as the j-th column contain all zeros.

Vertices containing loops have non-zero values on the diagonal of the adjacency matrix.

Checking the existence of edges between vertices in the adjacency matrix is an operation of O (1) complexity.

The disadvantage of this representation is the square memory complexity.

Many fields of such matrices contain zeros, except in the case of full graphs (clique).



INCIDENCE MATRIX		Vj - END VERTEX				
		1	**2**	**3**	**4**	**5**
Vi - START VERTEX	**1**	0	0	1	1	0
	2	1	0	1	1	1
	3	0	1	0	1	1
	4	1	0	0	0	1
	5	1	0	0	0	0

Graph as an Incidence Matrix

The incidence matrix is a matrix with the dimension N x K, where N - means the number of vertices of the graph and K the number of its edges. Each row maps one vertex of the graph, and each column one its edge. The cell content of such a matrix determines the association of the i-th vertex with the k-th edge as follows for the directed graph:

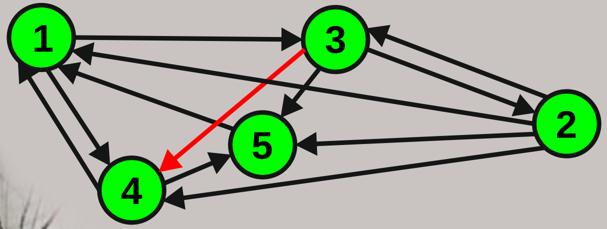
- 0 - means that the i-th vertex is neither the beginning nor the end of the k-th edge,
- 1 or w - means that the i-th vertex is the beginning of the k-th edge (weight w is optional),
- -1 - means that the i-th vertex is the end of the k-th edge.

If the graph is undirected, then the beginning and end is marked with 1.

If the graph contains loops, then the beginning and end fall at the same vertex, which we mark in the graph with the value 2. Multiple edges can be represented without difficulty.

Due to the fact that each edge has exactly one beginning and one end, in each column of such matrix there is exactly one 1 and one -1, and the other values are 0.

Each row of such a matrix can contain many 1 and -1, because it can be the beginning or end of many edges.



INCIDENCE LIST		Vk - EDGE											
		1	2	3	4	5	6	7	8	9	10	11	11
Vi - START VERTEX	1	1	1	-1	0	0	0	0	0	0	-1	0	-1
	2	0	0	1	1	1	1	-1	0	0	0	0	0
	3	-1	0	0	-1	0	0	1	1	1	0	0	0
	4	0	-1	0	0	-1	0	0	-1	0	1	1	0
	5	0	0	0	0	0	-1	0	0	-1	0	-1	1

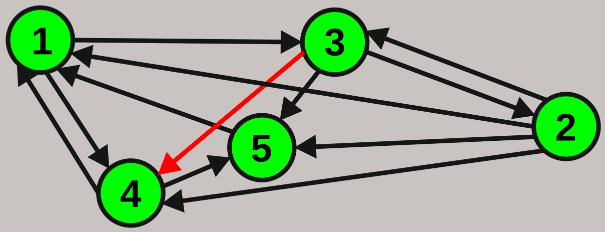
Graph as an Adjacency List

The adjacency (neighborhood) list is implemented as a list array (in Python as a list list), i.e. each element of the list of starting vertices contains a list of end vertices, i.e. a list of neighbors with which it is connected by an edge.

Slightly more difficult in the directed graph we find the vertices from which the edge leads to a given vertex, because you need to search all the lists containing the index of the final vertex O (K), but more often we are interested in the list of neighbors, i.e. to which vertices the edge from a given vertex leads. This problem does not occur with non-directed graphs.

This is by far the most effective (in terms of memory) and the most commonly used method of graph implementation, as it does not contain unused memory cells (except for isolated vertices).

To save the edge weight, use the list of object (class) lists whose key is the index (label) of the target vertex, and the weight is saved inside the class.



ADJACENCY L.		Vj - END VERTEX			
Vi - START VERTEX	1	3	4		
	2	1	3	4	5
	3	2	4	5	
	4	1	5		
	5	1			

Walking around a Graph

The vertices of the graph cannot be traversed in a linear way, so the most widely used are the trees spread (spanned) on the graph and the algorithms used to search the trees.

However, there may be cycles and loops in the graphs, which could cause such an algorithm to loop and spin around the same vertices many times.

To avoid this, we use an additional Boolean parameter, in which we mark whether the vertex has already been visited / searched or not.

So, if we enrich the tree search algorithms with such a parameter, we will be able to use the algorithms:

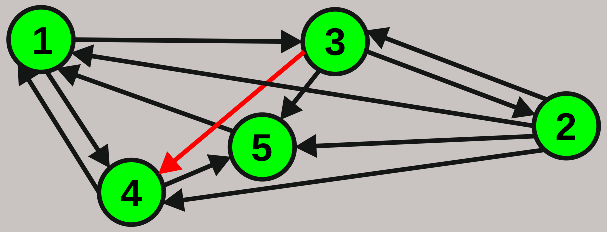
- DFS - depth first search - depth search,
- BFS - breadth first search - width search

also for searching graphs using this spanned tree on a graph.

The graph traversal transition is done using the DFS algorithm as follows:

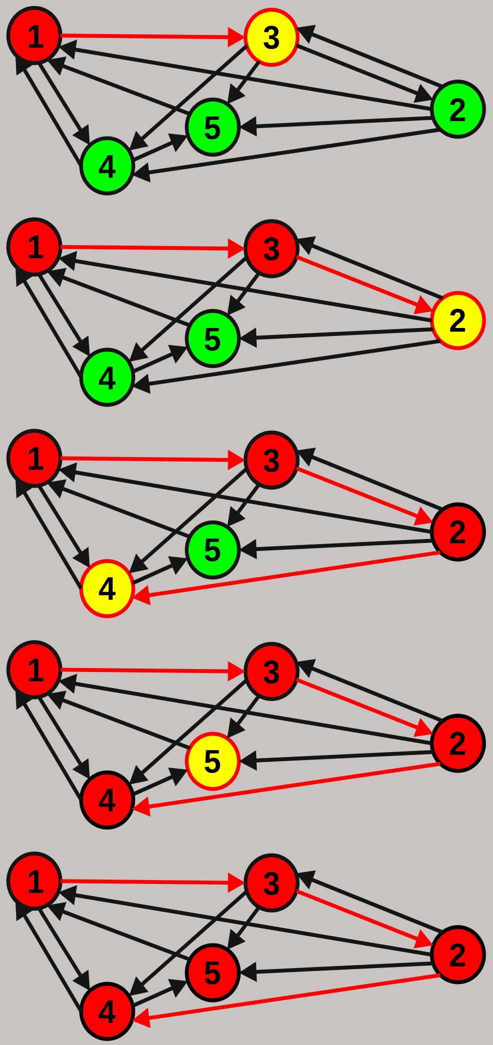
1. We search the current vertex.
2. We mark the current vertex as visited.
3. We go to the first not visited yet incident vertex of the current vertex.
4. If all incidental vertices have already been visited, then we return to the vertex from which we went to this vertex and perform point 3 in it.
5. We finish searching the graph when the vertex does not have a predecessor from which we entered.

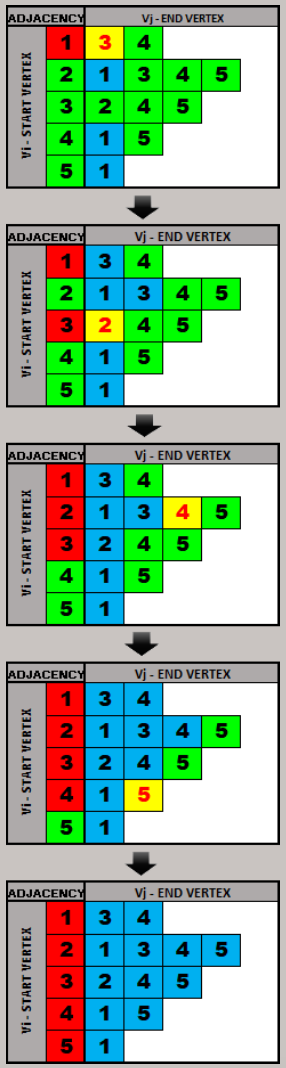
	ADJACENCY L.	Vj - END VERTEX			
Vi - START VERTEX	1	3	4		
	2	1	3	4	5
	3	2	4	5	
	4	1	5		
	5	1			



Walking DFS around a Graph

1. We start the graph search from vertex number 1.
2. We go to its first incidental and not yet visited vertex, i.e. to vertex No. 3.
3. From vertex 3. we go to its first incidental, unvisited vertex No. 2.
4. When searching the list of incidental vertices of vertex No. 2, we skip vertices No. 1 and No. 3, because they were already visited, finding vertex No. 4 as the first not visited yet.
5. At the vertex no. 4 on the incidence list, we skip the vertex no. 1 we have visited before and we reach the yet unvisited vertex no. 5 and go to it.
6. On the list of incidents in vertex no. 5 there is no unvisited vertex yet, so we retreat.
7. We proceed similarly in vertices No. 4, 2, 3, 1, where we also find no unvisited vertex.
8. We are finishing the graph search.





ADJACENCY	Vj - END VERTEX			
Vi - START VERTEX 1	3	4		
2	1	3	4	5
3	2	4	5	
4	1	5		
5	1			

ADJACENCY	Vj - END VERTEX			
Vi - START VERTEX 1	3	4		
2	1	3	4	5
3	2	4	5	
4	1	5		
5	1			

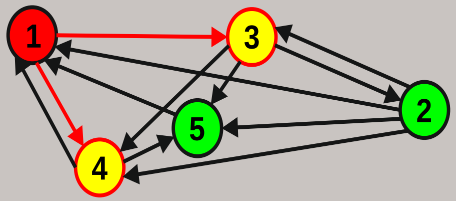
ADJACENCY	Vj - END VERTEX			
Vi - START VERTEX 1	3	4		
2	1	3	4	5
3	2	4	5	
4	1	5		
5	1			

ADJACENCY	Vj - END VERTEX			
Vi - START VERTEX 1	3	4		
2	1	3	4	5
3	2	4	5	
4	1	5		
5	1			

ADJACENCY	Vj - END VERTEX			
Vi - START VERTEX 1	3	4		
2	1	3	4	5
3	2	4	5	
4	1	5		
5	1			

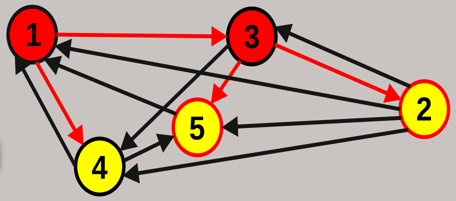
Walking BFS around a Graph

1. Graph transition with the BFS algorithm is done as follows:
2. We create an empty queue of searched vertices.
3. We add to the queue the vertex from which we start the graph search and mark it as included in the search (visited).
4. We take the first vertex from the queue and search it.
5. In the queue of searched vertices, enter all of its incidental vertices that have not yet been searched (visited) and mark them as included in the search.
6. We go to step 3.
7. We finish searching the graph when the queue is empty.
8. The number of potentially added items to the queue can be O (N), so the BFS algorithm is burdened with linear memory complexity. Usually fewer items are added to the queue, but in the case of a full graph, N-1 items (neighbors) will be added.



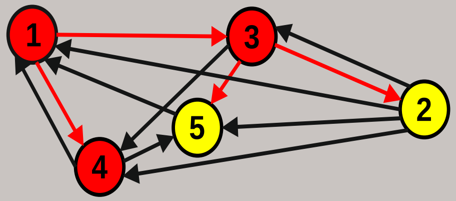
ADJACENCY	Vj - END VERTEX			
Vi - START VERTEX 1	3	4		
2	1	3	4	5
3	2	4	5	
4	1	5		
5	1			

K: 3 4



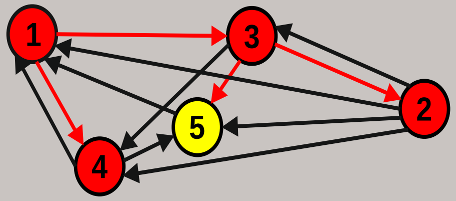
ADJACENCY	Vj - END VERTEX			
Vi - START VERTEX 1	3	4		
2	1	3	4	5
3	2	4	5	
4	1	5		
5	1			

K: 4 2 5



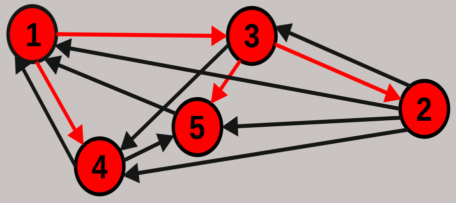
ADJACENCY	Vj - END VERTEX			
Vi - START VERTEX 1	3	4		
2	1	3	4	5
3	2	4	5	
4	1	5		
5	1			

K: 2 5



ADJACENCY	Vj - END VERTEX			
Vi - START VERTEX 1	3	4		
2	1	3	4	5
3	2	4	5	
4	1	5		
5	1			

K: 5



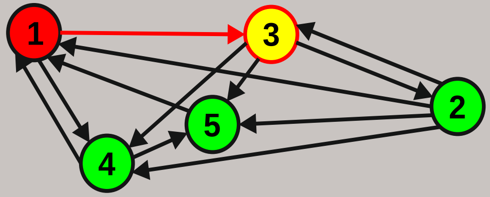
ADJACENCY	Vj - END VERTEX			
Vi - START VERTEX 1	3	4		
2	1	3	4	5
3	2	4	5	
4	1	5		
5	1			

K:

Comparison of DFS and BFS

DFS

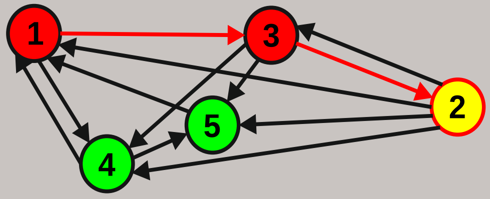
BFS



ADJACENCY		Vj - END VERTEX			
Vi - START VERTEX	1	3	4		
	2	1	3	4	5
	3	2	4	5	
	4	1	5		
	5	1			

ADJACENCY		Vj - END VERTEX			
Vi - START VERTEX	1	3	4		
	2	1	3	4	5
	3	2	4	5	
	4	1	5		
	5	1			

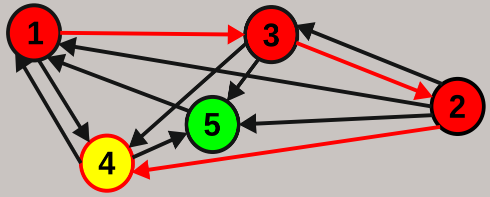
K: 3 4



ADJACENCY		Vj - END VERTEX			
Vi - START VERTEX	1	3	4		
	2	1	3	4	5
	3	2	4	5	
	4	1	5		
	5	1			

ADJACENCY		Vj - END VERTEX			
Vi - START VERTEX	1	3	4		
	2	1	3	4	5
	3	2	4	5	
	4	1	5		
	5	1			

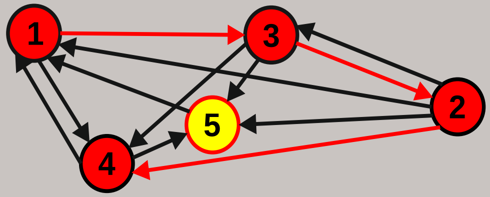
K: 4 2 5



ADJACENCY		Vj - END VERTEX			
Vi - START VERTEX	1	3	4		
	2	1	3	4	5
	3	2	4	5	
	4	1	5		
	5	1			

ADJACENCY		Vj - END VERTEX			
Vi - START VERTEX	1	3	4		
	2	1	3	4	5
	3	2	4	5	
	4	1	5		
	5	1			

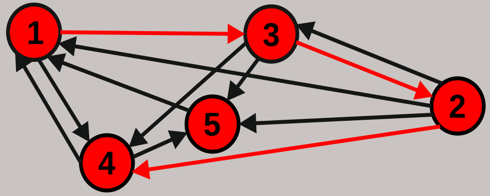
K: 2 5



ADJACENCY		Vj - END VERTEX			
Vi - START VERTEX	1	3	4		
	2	1	3	4	5
	3	2	4	5	
	4	1	5		
	5	1			

ADJACENCY		Vj - END VERTEX			
Vi - START VERTEX	1	3	4		
	2	1	3	4	5
	3	2	4	5	
	4	1	5		
	5	1			

K: 5



ADJACENCY		Vj - END VERTEX			
Vi - START VERTEX	1	3	4		
	2	1	3	4	5
	3	2	4	5	
	4	1	5		
	5	1			

ADJACENCY		Vj - END VERTEX			
Vi - START VERTEX	1	3	4		
	2	1	3	4	5
	3	2	4	5	
	4	1	5		
	5	1			

K:

AGDS Implementation

Implement the simplistic version of an AGDS structure for the chosen dataset.

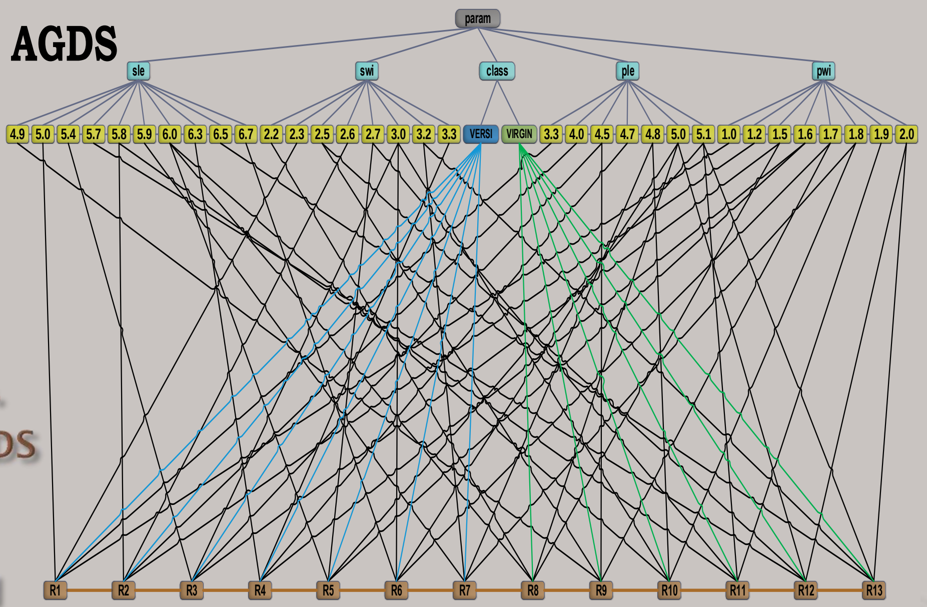
You can but there is no need to use AVB+trees to create it.
You can use sorted lists or tables of Value Vertices, hash-tables, or dictionaries, but the result should be similar and correct.

You don't need to present your results or a network structure graphically,

Only text form of results is required.

Independently from your choice of implementation AGDS use the selected one to:

1. Find the most similar object(s) to the given object or a group of objects.
2. Find the most similar object(s) to any combination of input data given on the input(s).
3. Using associations generate the list of all objects sorted by the similarity to the give object (Task 1) or any combination of input data (Task 2).
4. Try to implement a classifier using AGDS structure.
5. Compute min, max, average, medians using classic algorithms and computed the time to similar operations on AGDS structures.
6. Implement knowledge-based inferences on AGDS structures.

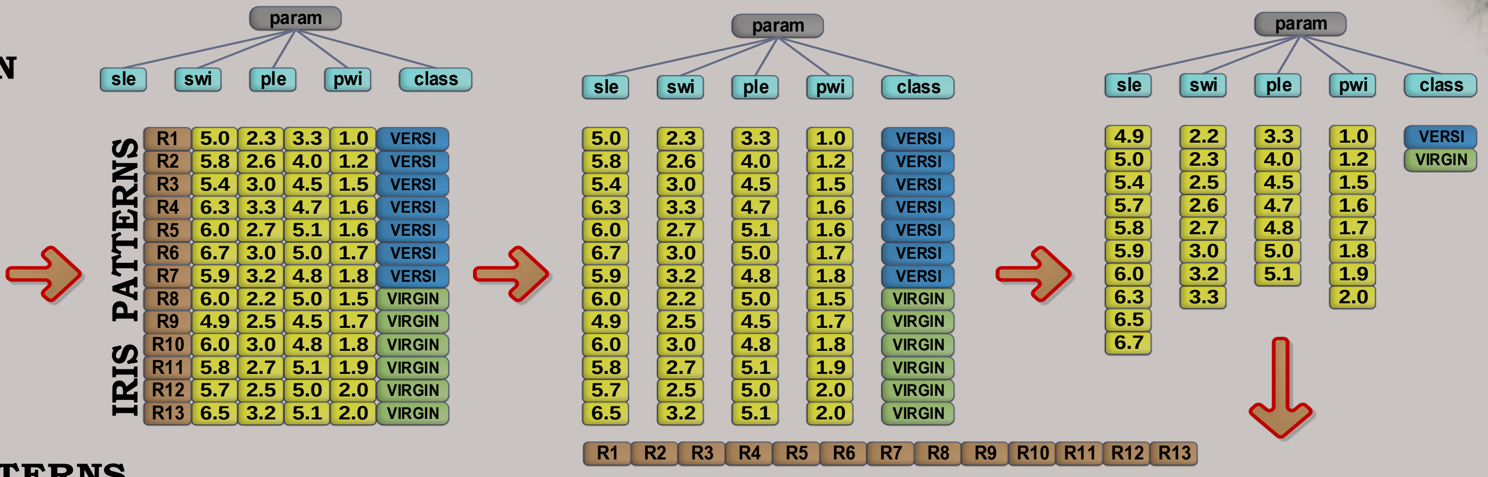


AGDS Implementation

ASSOCIATIVE TRANSFORMATION

IRIS PATTERNS

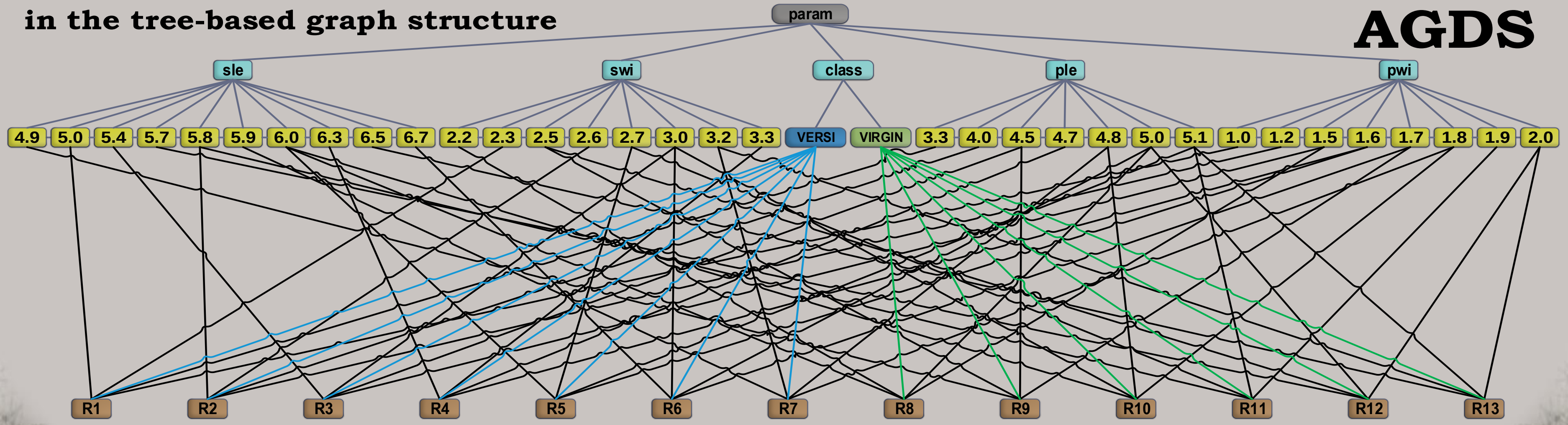
param	sle	swi	ple	pwi	class
R1	5.0	2.3	3.3	1.0	VERSI
R2	5.8	2.6	4.0	1.2	VERSI
R3	5.4	3.0	4.5	1.5	VERSI
R4	6.3	3.3	4.7	1.6	VERSI
R5	6.0	2.7	5.1	1.6	VERSI
R6	6.7	3.0	5.0	1.7	VERSI
R7	5.9	3.2	4.8	1.8	VERSI
R8	6.0	2.2	5.0	1.5	VIRGIN
R9	4.9	2.5	4.5	1.7	VIRGIN
R10	6.0	3.0	4.8	1.8	VIRGIN
R11	5.8	2.7	5.1	1.9	VIRGIN
R12	5.7	2.5	5.0	2.0	VIRGIN
R13	6.5	3.2	5.1	2.0	VIRGIN



param
sle
swi
ple
pwi
class
IRIS PATTERNS
R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13
sle: 4.9 5.0 5.4 5.7 5.8 5.9 6.0 6.3 6.5 6.7
swi: 2.2 2.3 2.5 2.6 2.7 3.0 3.2 3.3
ple: 3.3 4.0 4.5 4.7 4.8 5.0 5.1
pwi: 1.0 1.2 1.5 1.6 1.7 1.8 1.9 2.0
class: VERSI VIRGIN

IRIS PATTERNS
in the tree-based graph structure

AGDS



To represent Value Nodes, use sorted lists or tables together with the half-search algorithm or (the more ambitious solution: with the AVB+trees) which will allow you to find two closest values when the searched one will not be found. Standard hash-tables and sorted-lists have no such functionality!

AGDS Implementation

AGDS nodes representing neighboring (subsequent) values of each attribute a_k are connected and the weight of this connection (edge) is computed after the following formula:

$$w_{v_i^{a_k}, v_j^{a_k}} = 1 - \frac{\left|v_i^{a_k} - v_j^{a_k}\right|}{r^{a_k}}$$

weights do not need to be stored in this structure

but computed on demand when necessary.

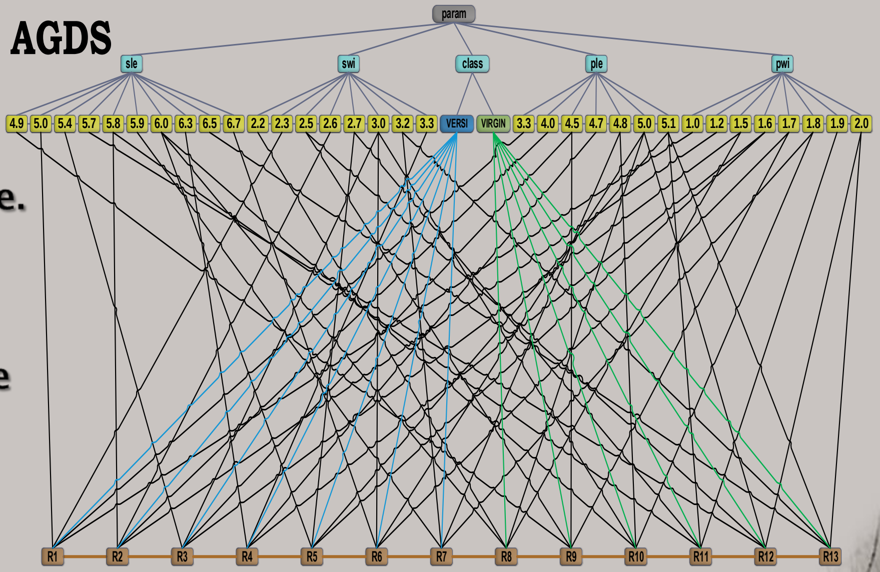
In this case, you don't need to update weights when adding new objects to this structure, where

$v_i^{a_k}$, $v_j^{a_k}$ - are values represented by the neighboring attribute nodes, which are connected by an edge in the AGDS graph,

$r^{a_k} = v_{max}^{a_k} - v_{min}^{a_k}$ - is the current range of values of the attribute a_k.

The weight of the connection from the value node $v_i^{a_k}$ of the attribute a_k to the object node R_m is determined after the number of occurrences $N_i^{a_k}$ of this value ($v_i^{a_k}$) in all objects:

$$w_{v_i^{a_k}, R_m} = \frac{1}{N_i^{a_k}} = \frac{1}{\left\|v_i^{a_k}\right\|}$$

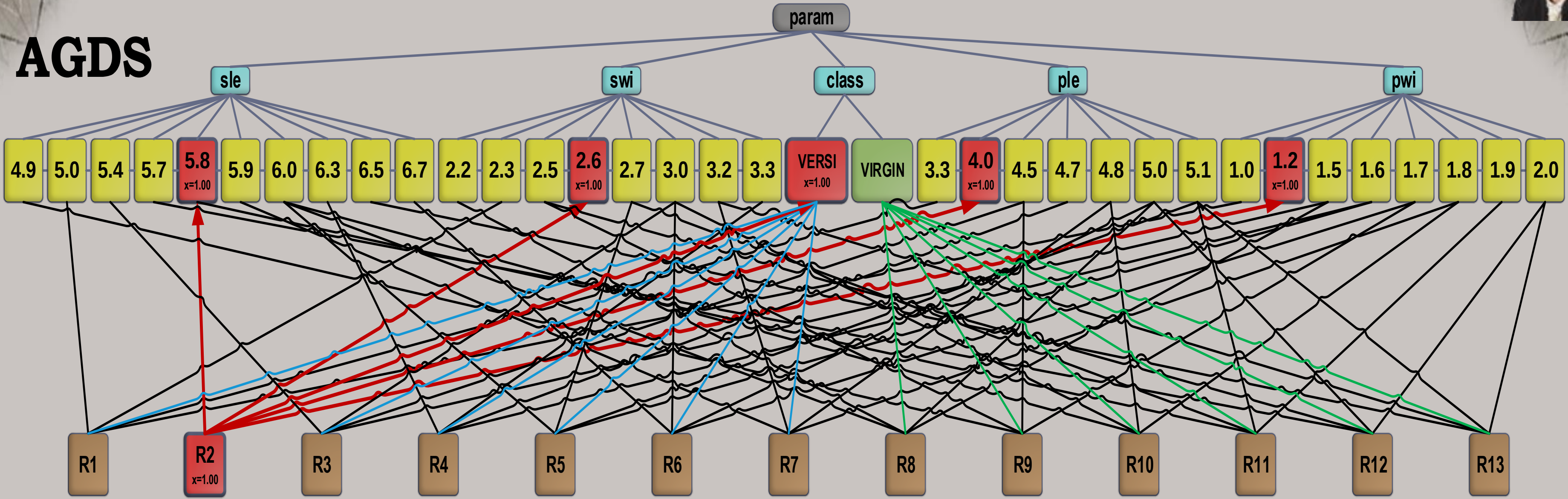


These numbers ($N_i^{a_k} = \left\|v_i^{a_k}\right\|$) are stored in the individual value nodes of each attribute. This number is equal to the number or all connections of this value node to all object nodes if there are no duplicated objects in the table used to create the AGDS structure.

In the opposite direction, the weights of connections from the object nodes to the value nodes are always equal to one:

$$w_{R_m, v_i^{a_k}} = 1$$

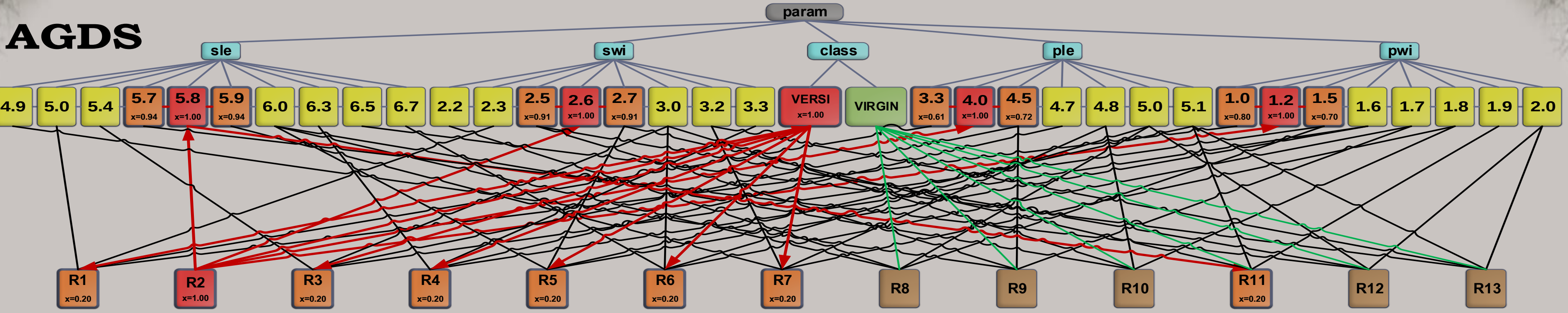
AGDS Associative Inference

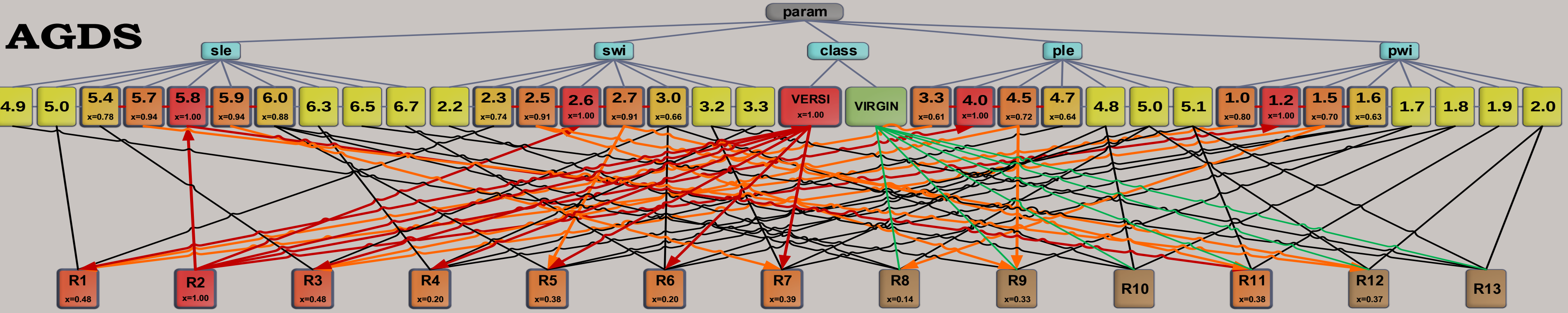


Associative data structures AGDS can be now used for associative inference, which is based on moving along the connections to
the connected nodes and computing some values in these nodes on the basis of the send values multiplied by weights of these connections. In such a way we get the information about e.g. similarity of objects represented by other nodes of the same kind or about the objects that satisfy some given conditions defined by the represented attribute values. Let's use our AGDS graph created for 13 Irises for such inference looking for objects (Irises) Rx which are most similar to R2.

1. **We start in the node R2 which assumes the similarity value x=1.0, because this node is 100% similar to itself.**
2. **Next, we assign values x of the connected nodes representing the following values: 5.8, 2.6, VERSI, 4.0, and 1.2 by multiplying the value coming from the node R2 with the connection weights, which are equal 1.0.**
   So, as a result, we achieve x=1.0 for all these connected nodes.

AGDS Associative Inference

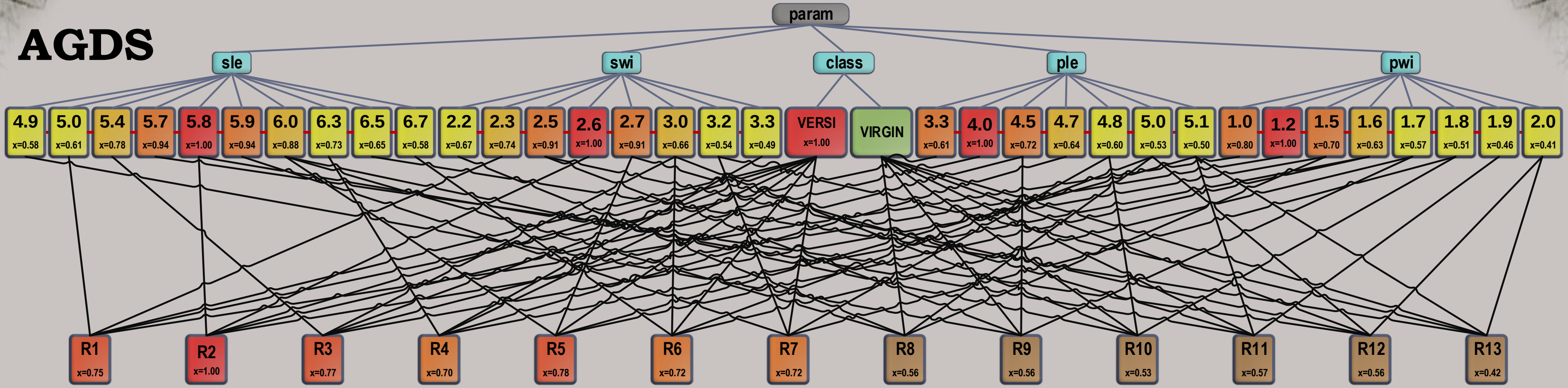




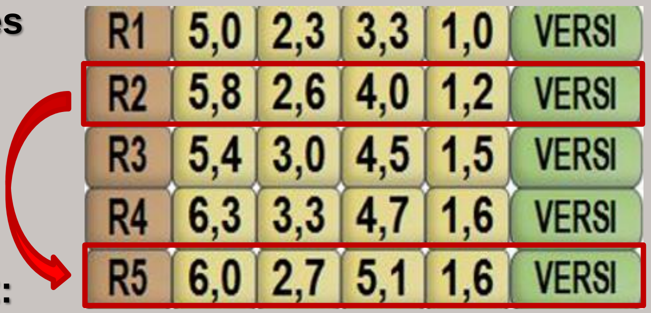
3. Subsequently, the values computed for these nodes are multiplied by next connection weights and send to the neighbor connected value nodes, for which we also compute their similarity values x.
4. Similarly, we compute the similarity values x for connected object nodes with regards to the necessity to add the passed weighted values to the sums already stored in these nodes, e.g. for the node R3, we compute $x = 1.0 * 0.2 + 0.72 * 0.2 + 0.7 * 0.2 = 0.48$

AGDS Associative Inference

AGDS



5. **Finally, when we go through all the connected (associated) values nodes computing theirs values of similarities by multiplying the sender similarity values by connection weights.**
   We also computed weighted sums for all object nodes, where these weights are here equal w = 1/5 = 0.2.
   The computed similarity values for the nodes Rx can be used to compare and designate the most similar objects to the object R2: R5 (78%), R3 (77%), R1 (75%), …



R1	5,0	2,3	3,3	1,0	VERSI
R2	5,8	2,6	4,0	1,2	VERSI
R3	5,4	3,0	4,5	1,5	VERSI
R4	6,3	3,3	4,7	1,6	VERSI
R5	6,0	2,7	5,1	1,6	VERSI

Let's start with powerful computations!

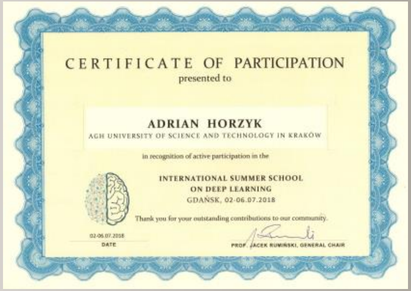
Bibliography and Literature

1. Nikola K. Kasabov, *Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence*, In Springer Series on Bio- and Neurosystems, Vol 7., Springer, 2019.
2. Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep Learning*, MIT Press, 2016, ISBN 978-1-59327-741-3 or PWN 2018.
3. Holk Cruse, *Neural Networks as Cybernetic Systems*, 2nd and revised edition
4. R. Rojas, *Neural Networks*, Springer-Verlag, Berlin, 1996.
5. *Convolutional Neural Network* (Stanford)
6. *Visualizing and Understanding Convolutional Networks*, Zeiler, Fergus, ECCV 2014
7. IBM: https://www.ibm.com/developerworks/library/ba-data-becomes-knowledge-1/index.html
8. NVIDIA: https://developer.nvidia.com/discover/convolutional-neural-network
9. JUPYTER: https://jupyter.org/
10. https://www.youtube.com/watch?v=XNKeayZW4dY
11. https://victorzhou.com/blog/keras-cnn-tutorial/
12. https://github.com/keras-team/keras/tree/master/examples
13. https://medium.com/@margaretmz/anaconda-jupyter-notebook-tensorflow-and-keras-b91f381405f8
14. https://blog.tensorflow.org/2019/09/tensorflow-20-is-now-available.html
15. http://coursera.org/specializations/tensorflow-in-practice
16. https://udacity.com/course/intro-to-tensorflow-for-deep-learning
17. MNIST sample: https://medium.com/datadriveninvestor/image-processing-for-mnist-using-keras-f9a1021f6ef0
18. Heatmaps: https://towardsdatascience.com/formatting-tips-for-correlation-heatmaps-in-seaborn-4478ef15d87f

Adrian Horzyk

horzyk@agh.edu.pl

Google: Horzyk



CERTIFICATE OF PARTICIPATION
presented to

ADRIAN HORZYK
AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY IN KRAKÓW

in recognition of active participation in the

INTERNATIONAL SUMMER SCHOOL
ON DEEP LEARNING
GDAŃSK, 02-06.07.2018

Thank you for your outstanding contributions to our community.

02-06.07.2018
DATE

PROF. JACEK RUMIŃSKI, GENERAL CHAIR

AGH

University of Science and Technology in Krakow, Poland