# ДОДАТОК А

## Код програми

```
from abc import ABCMeta, abstractmethod
from copy import copy
from mayavi import mlab
import operator
import imageio
from collections import OrderedDict
from scipy.spatial.distance import euclidean
from sklearn import preprocessing
import csv
import numpy as np
import networkx as nx
import re
import os
import shutil
import sys
import glob
from future.utils import iteritems
import time

def sh(s):
    sum = 0
    for i, c in enumerate(s):
        sum += i * ord(c)
    return sum

def create_data_graph(dots):
    """Create the graph and returns the networkx version of it 'g'."""

    count = 0

    g = nx.Graph()

    for i in dots:
        g.add_node(count, pos=i)
        count += 1
    return g

def get_ra(ra=0, ra_step=0.3):
    while True:
        if ra >= 360:
            ra = 0
        else:
            ra += ra_step
        yield ra

def shrink_to_3d(data):
    result = []

    for i in data:
        depth = len(i)
        if depth <= 3:
            result.append(i)
        else:
            sm = np.sum([(n) * v for n, v in enumerate(i[2:])])
            if sm == 0:
                sm = 1

            r = np.array([i[0], i[1], i[2]])
            r *= sm
```

```
                r /= np.sum(r)

                result.append(r)

        return preprocessing.normalize(result, axis=0, norm='max')

    def draw_dots3d(dots, edges, fignum, clear=True,
                    title='',
                    size=(1024, 768), graph_colormap='viridis',
                    bgcolor=(1, 1, 1),
                    node_color=(0.3, 0.65, 0.3), node_size=0.01,
                    edge_color=(0.3, 0.3, 0.9), edge_size=0.003,
                    text_size=0.14, text_color=(0, 0, 0), text_coords=[0.84, 0.75], text={},
                    title_size=0.3,
                    angle=get_ra()):

        # numpy array of x, y, z positions in sorted node order
        xyz = shrink_to_3d(dots)

        if fignum == 0:
            mlab.figure(fignum, bgcolor=bgcolor, fgcolor=text_color, size=size)

        # Mayavi is buggy, and following code causes sockets leak.
        #if mlab.options.offscreen:
        #    mlab.figure(fignum, bgcolor=bgcolor, fgcolor=text_color, size=size)
        #elif fignum == 0:
        #    mlab.figure(fignum, bgcolor=bgcolor, fgcolor=text_color, size=size)

        if clear:
            mlab.clf()

        # the x,y, and z co-ordinates are here
        # manipulate them to obtain the desired projection perspective

        pts = mlab.points3d(xyz[:, 0], xyz[:, 1], xyz[:, 2],
                            scale_factor=node_size,
                            scale_mode='none',
                            color=node_color,
                            #colormap=graph_colormap,
                            resolution=20,
                            transparent=False)

        mlab.text(text_coords[0], text_coords[1], '\n'.join(['{} = {}'.format(n, v) for n, v
in text.items()]), width=text_size)

        if clear:
            mlab.title(title, height=0.95)
            mlab.roll(next(angle))
            mlab.orientation_axes(pts)
            mlab.outline(pts)

        pts.mlab_source.dataset.lines = edges
        tube = mlab.pipeline.tube(pts, tube_radius=edge_size)
        mlab.pipeline.surface(tube, color=edge_color)

        #mlab.show() # interactive window

    def draw_graph3d(graph, fignum, *args, **kwargs):
        graph_pos = nx.get_node_attributes(graph, 'pos')
        edges = np.array([e for e in graph.edges()])
        dots = np.array([graph_pos[v] for v in sorted(graph)], dtype='float64')

        draw_dots3d(dots, edges, fignum, *args, **kwargs)

    def generate_host_activity(is_normal):
        # Host loads is changed only in 25% cases.
        attack_percent = 25
        up_level = (20, 30)
```

```
            # CPU load in percent.
            cpu_load = (10, 30)
            # Disk IO per second.
            iops = (10, 50)
            # Memory consumption in percent.
            mem_cons = (30, 60)
            # Memory consumption in Mb/s.
            netw_act = (10, 50)

            cur_up_level = 0

            if not is_normal and np.random.randint(0, 100) < attack_percent:
                cur_up_level = np.random.randint(*up_level)

            cpu_load = np.random.randint(cur_up_level + cpu_load[0], cur_up_level + cpu_load[1])
            iops = np.random.randint(cur_up_level + iops[0], cur_up_level + iops[1])
            mem_cons = np.random.randint(cur_up_level + mem_cons[0], cur_up_level + mem_cons[1])
            netw_act = np.random.randint(cur_up_level + netw_act[0], cur_up_level + netw_act[1])

            return cpu_load, iops, mem_cons, netw_act

        def     read_ids_data(data_file,     activity_type='normal',     labels_file='NSL_KDD/Field
Names.csv', with_host=False):
            selected_parameters  =  ['duration',  'protocol_type',  'service',  'flag',  'src_bytes',
'dst_bytes', 'land',
                                     'wrong_fragment', 'urgent', 'serror_rate',
                                     'diff_srv_rate',   'srv_diff_host_rate',   'dst_host_srv_count',
'count']

            # "Label" - "converter function" dictionary.
            label_dict = OrderedDict()
            result = []

            with open(labels_file) as lf:
                labels = csv.reader(lf)
                for label in labels:
                    if len(label) == 1 or label[1] == 'continuous':
                        label_dict[label[0]] = lambda l: np.float64(l)
                    elif label[1] == 'symbolic':
                        label_dict[label[0]] = lambda l: sh(l)

            f_list = [i for i in label_dict.values()]
            n_list = [i for i in label_dict.keys()]

            if activity_type == 'normal':
                data_type = lambda t: t == 'normal'
            elif activity_type == 'abnormal':
                data_type = lambda t: t != 'normal'
            elif activity_type == 'full':
                data_type = lambda t: True
            else:
                raise ValueError('`activity_type` must be "normal", "abnormal" or "full"')

            print('Reading {} activity from the file "{}" [generated host data {} included]...'.
                  format(activity_type, data_file, 'was' if with_host else 'was not'))

            with open(data_file) as df:
                # data = csv.DictReader(df, label_dict.keys())
                data = csv.reader(df)
                for d in data:
                    if data_type(d[-2]):
                        # Skip last two fields and add only specified fields.
                        net_params = tuple(f_list[n](i) for n, i in enumerate(d[:-2]) if n_list[n]
in selected_parameters)

                        if with_host:
                            host_params = generate_host_activity(activity_type != 'abnormal')
```

```
                        result.append(net_params + host_params)
                    else:
                        result.append(net_params)
            print('Records count: {}'.format(len(result)))
            return result

        class NeuralGas:
            __metaclass__ = ABCMeta

            def __init__(self, data, surface_graph=None, output_images_dir='images'):
                self._graph = nx.Graph()
                self._data = data
                self._surface_graph = surface_graph
                # Deviation parameters.
                self._dev_params = None
                self._output_images_dir = output_images_dir
                # Nodes count.
                self._count = 0

                if os.path.isdir(output_images_dir):
                    shutil.rmtree('{}'.format(output_images_dir))

                print("Output images will be saved in: {0}".format(output_images_dir))
                os.makedirs(output_images_dir)

                self._start_time = time.time()

            @abstractmethod
            def train(self, max_iterations=100, save_step=0):
                raise NotImplementedError()

            @property
            def number_of_clusters(self):
                return nx.number_connected_components(self._graph)

            @property
            def clusters(self):
                return nx.connected_components(self._graph)

            def detect_anomalies(self, data, threshold=5, train=False, save_step=100):
                anomalies_counter, anomaly_records_counter, normal_records_counter = 0, 0, 0
                anomaly_level = 0

                start_time = self._start_time = time.time()

                for i, d in enumerate(data):
                    risk_level = self.test_node(d, train)
                    if risk_level != 0:
                        anomaly_records_counter += 1
                        anomaly_level += risk_level
                        if anomaly_level > threshold:
                            anomalies_counter += 1
                            #print('Anomaly was detected [count = {}]!'.format(anomalies_counter))
                            anomaly_level = 0
                    else:
                        normal_records_counter += 1

                    if i % save_step == 0:
                        tm = time.time() - start_time
                        print('Abnormal records = {}, Normal records = {}, Detection time = {} s,
Time per record = {} s'.
                              format(anomaly_records_counter,  normal_records_counter,  round(tm,
2), tm / i if i else 0))

                tm = time.time() - start_time

                print('{} [abnormal records = {}, normal records = {}, detection time = {} s, time
per record = {} s]'.
```

```
                    format('Anomalies were detected (count = {})'.format(anomalies_counter)
                           if anomalies_counter else 'Anomalies weren\'t detected',
                           anomaly_records_counter, normal_records_counter, round(tm, 2), tm /
len(data)))

                return anomalies_counter > 0

            def test_node(self, node, train=False):
                n, dist = self._determine_closest_vertice(node)
                dev = self._calculate_deviation_params()
                dev = dev.get(frozenset(nx.node_connected_component(self._graph, n)), dist + 1)
                dist_sub_dev = dist - dev
                if dist_sub_dev > 0:
                    return dist_sub_dev

                if train:
                    self._dev_params = None
                    self._train_on_data_item(node)

                return 0

            @abstractmethod
            def _train_on_data_item(self, data_item):
                raise NotImplementedError()

            @abstractmethod
            def _save_img(self, fignum, training_step):
                """."""
                raise NotImplementedError()

            def _calculate_deviation_params(self, distance_function_params=None):
                if self._dev_params is not None:
                    return self._dev_params

                clusters = {}
                dcvd = self._determine_closest_vertice
                dlen = len(self._data)
                #dmean = np.mean(self._data, axis=1)
                #deviation = 0

                for node in self._data:
                    n = dcvd(node) #, **distance_function_params)
                    cluster                                                                    =
clusters.setdefault(frozenset(nx.node_connected_component(self._graph, n[0])), [0, 0])
                    cluster[0] += n[1]
                    cluster[1] += 1

                clusters = {k: np.sqrt(v[0]/v[1]) for k, v in clusters.items()}

                self._dev_params = clusters

                return clusters

            def _determine_closest_vertice(self, curnode):
                """."""

                pos = nx.get_node_attributes(self._graph, 'pos')
                kv = list(zip(*pos.items()))
                distances = np.linalg.norm(kv[1] - curnode, ord=2, axis=1)

                i0 = np.argsort(distances)[0]

                return kv[0][i0], distances[i0]

            def _determine_2closest_vertices(self, curnode):
                """Where this curnode is actually the x,y index of the data we want to analyze."""

                pos = nx.get_node_attributes(self._graph, 'pos')
```

```
                l_pos = len(pos)
                if l_pos == 0:
                    return None, None
                elif l_pos == 1:
                    return pos[0], None

                kv = list(zip(*pos.items()))
                # Calculate Euclidean distance (2-norm of difference vectors) and get first two
indexes of the sorted array.
                # Or a Euclidean-closest nodes index.
                distances = np.linalg.norm(kv[1] - curnode, ord=2, axis=1)
                i0, i1 = np.argsort(distances)[0:2]

                winner1 = tuple((kv[0][i0], distances[i0]))
                winner2 = tuple((kv[0][i1], distances[i1]))

                return winner1, winner2

        class GNG(NeuralGas):
            """Growing Neural Gas multidimensional implementation"""

            def __init__(self, data, surface_graph=None, eps_b=0.05, eps_n=0.0006, max_age=15,
                         lambda_=20, alpha=0.5, d=0.005, max_nodes=1000,
                         output_images_dir='images'):
                """."""
                NeuralGas.__init__(self, data, surface_graph, output_images_dir)

                self._eps_b = eps_b
                self._eps_n = eps_n
                self._max_age = max_age
                self._lambda = lambda_
                self._alpha = alpha
                self._d = d
                self._max_nodes = max_nodes
                self._fignum = 0

                self.__add_initial_nodes()

            def train(self, max_iterations=10000, save_step=50, stop_on_chi=False):
                """."""

                self._dev_params = None
                self._save_img(self._fignum, 0)
                graph = self._graph
                max_nodes = self._max_nodes
                d = self._d
                ld = self._lambda
                alpha = self._alpha
                update_winner = self.__update_winner
                data = self._data
                CHS = self.__calinski_harabaz_score
                old = 0
                calin = CHS()
                start_time = self._start_time = time.time()
                train_step = self.__train_step

                for i in range(1, max_iterations):
                    tm = time.time() - start_time
                    print(f'Training time = {round(tm, 2)} s, ' +
                          f'Time per record = {tm / len(data)} s ' +
                          f'Training   step   =   {i}   /   {max_iterations},   Clusters   count   =
{self.number_of_clusters}, ' +
                          f'Neurons = {len(self._graph)}'
                          )
                    for x in data:
                        update_winner(x)

                    train_step(i, alpha, ld, d, max_nodes, True, save_step, graph, update_winner)
```

```
                    old = calin
                    calin = CHS()

                    # Stop on the enough clusterization quality.
                    if stop_on_chi and old - calin > 0:
                        break
                print(f'Training complete, clusters count = {self.number_of_clusters}, ' +
                      f'training time = {round(time.time() - start_time, 2)} s')

            def __train_step(self, i, alpha, ld, d, max_nodes, save_img, save_step, graph,
update_winner):
                g_nodes = graph.nodes

                # Step 8: if number of input signals generated so far
                if i % ld == 0 and len(graph) < max_nodes:
                    # Find a node with the largest error.
                    errorvectors = nx.get_node_attributes(graph, 'error')
                    node_largest_error = max(errorvectors.items(), key=operator.itemgetter(1))[0]

                    # Find a node from neighbor of the node just found, with a largest error.
                    neighbors = graph.neighbors(node_largest_error)
                    max_error_neighbor = 0
                    max_error = -1

                    for n in neighbors:
                        ce = g_nodes[n]['error']
                        if ce > max_error:
                            max_error = ce
                            max_error_neighbor = n

                    # Decrease error variable of other two nodes by multiplying with alpha.
                    new_max_error = alpha * errorvectors[node_largest_error]
                    graph.nodes[node_largest_error]['error'] = new_max_error
                    graph.nodes[max_error_neighbor]['error'] = alpha * max_error

                    # Insert a new unit half way between these two.
                    self._count += 1
                    new_node = self._count
                    graph.add_node(new_node,

pos=self.__class__.__get_average_dist(g_nodes[node_largest_error]['pos'],

g_nodes[max_error_neighbor]['pos']),
                                   error=new_max_error)

                    # Insert edges between new node and other two nodes.
                    graph.add_edge(new_node, max_error_neighbor, age=0)
                    graph.add_edge(new_node, node_largest_error, age=0)

                    # Remove edge between old nodes.
                    graph.remove_edge(max_error_neighbor, node_largest_error)

                    if True and i % save_step == 0:
                        self._fignum += 1
                        self._save_img(self._fignum, i)

                    # step 9: Decrease all error variables.
                    for n in graph.nodes():
                        oe = g_nodes[n]['error']
                        g_nodes[n]['error'] -= d * oe

            def _train_on_data_item(self, data_item):
                """IGNG training method"""

                np.append(self._data, data_item)

                graph = self._graph
```

```
                max_nodes = self._max_nodes
                d = self._d
                ld = self._lambda
                alpha = self._alpha
                update_winner = self.__update_winner
                data = self._data
                train_step = self.__train_step

                #for i in xrange(1, 5):
                update_winner(data_item)
                train_step(0, alpha, ld, d, max_nodes, False, -1, graph, update_winner)

            def _calculate_deviation_params(self):
                return super(GNG, self)._calculate_deviation_params()

            def __add_initial_nodes(self):
                """Initialize here"""

                node1 = self._data[np.random.randint(0, len(self._data))]
                node2 = self._data[np.random.randint(0, len(self._data))]

                # make sure you don't select same positions
                if self.__class__.__is_nodes_equal(node1, node2):
                    raise ValueError("Rerun ---------------> similar nodes selected")

                self._count = 0
                self._graph.add_node(self._count, pos=node1, error=0)
                self._count += 1
                self._graph.add_node(self._count, pos=node2, error=0)
                self._graph.add_edge(self._count - 1, self._count, age=0)

            @staticmethod
            def __is_nodes_equal(n1, n2):
                return len(set(n1) & set(n2)) == len(n1)

            def __update_winner(self, curnode):
                """."""

                # find nearest unit and second nearest unit
                winner1, winner2 = self._determine_2closest_vertices(curnode)
                winner_node1 = winner1[0]
                winner_node2 = winner2[0]
                win_dist_from_node = winner1[1]
                graph = self._graph
                g_nodes = graph.nodes

                # Update the winner error.
                g_nodes[winner_node1]['error'] +=  + win_dist_from_node**2

                # Move the winner node towards current node.
                g_nodes[winner_node1]['pos']      +=      self._eps_b      *      (curnode      -
g_nodes[winner_node1]['pos'])

                eps_n = self._eps_n

                # Now update all the neighbors distances.
                for n in nx.all_neighbors(graph, winner_node1):
                    g_nodes[n]['pos'] += eps_n * (curnode - g_nodes[n]['pos'])

                # Update age of the edges, emanating from the winner.
                for e in graph.edges(winner_node1, data=True):
                    e[2]['age'] += 1

                # Create or zeroe edge between two winner nodes.
                graph.add_edge(winner_node1, winner_node2, age=0)

                # if there are ages more than maximum allowed age, remove them
                age_of_edges = nx.get_edge_attributes(graph, 'age')
```

```
            max_age = self._max_age
            for edge, age in age_of_edges.items():
                if age >= max_age:
                    graph.remove_edge(edge[0], edge[1])

            # If it causes isolated vertix, remove that vertex as well.
            for node in g_nodes:
                if not graph.neighbors(node):
                    graph.remove_node(node)

        @staticmethod
        def __get_average_dist(a, b):
            """."""

            return (a + b) / 2

        def __calinski_harabaz_score(self):
            graph = self._graph
            nodes = graph.nodes
            extra_disp, intra_disp = 0., 0.

            # CHI = [B / (c - 1)]/[W / (n - c)]
            # Total numb er of neurons.
            #ns = nx.get_node_attributes(self._graph, 'n_type')
            c = len(nodes)
            # Total number of data.
            n = len(self._data)

            # Mean of the all data.
            mean = np.mean(self._data, axis=1)

            pos = nx.get_node_attributes(self._graph, 'pos')

            for node, k in pos.items():
                mean_k = np.mean(k)
                extra_disp += len(k) * np.sum((mean_k - mean) ** 2)
                intra_disp += np.sum((k - mean_k) ** 2)

        def _save_img(self, fignum, training_step):
            """."""

            title = 'Growing Neural Gas for the network anomalies detection'

            if self._surface_graph is not None:
                text = OrderedDict([
                    ('Image', fignum),
                    ('Training step', training_step),
                    ('Time', '{} s'.format(round(time.time() - self._start_time, 2))),
                    ('Clusters count', self.number_of_clusters),
                    ('Neurons', len(self._graph)),
                    ('Connections', len(self._graph.edges)),
                    ('Data records', len(self._data))
                ])

                draw_graph3d(self._surface_graph, fignum, title=title)

            graph = self._graph

            if len(graph) > 0:
                draw_graph3d(graph, fignum, clear=False, node_color=(1, 0, 0),
                             title=title,
                             text=text)

            mlab.savefig("{0}/{1}.png".format(self._output_images_dir, str(fignum)))


    def sort_nicely(limages):
        """Numeric string sort"""
```

```
            def convert(text): return int(text) if text.isdigit() else text

            def alphanum_key(key): return [convert(c) for c in re.split('([0-9]+)', key)]
            limages = sorted(limages, key=alphanum_key)
            return limages

        def convert_images_to_gif(output_images_dir, output_gif):
            """Convert a list of images to a gif."""

            image_dir = "{0}/*.png".format(output_images_dir)
            list_images = glob.glob(image_dir)
            file_names = sort_nicely(list_images)
            images = [imageio.imread(fn) for fn in file_names]
            imageio.mimsave(output_gif, images)

        def    test_detector(use_hosts_data,    max_iters,    alg,    output_images_dir='images',
output_gif='output.gif'):
            """Detector quality testing routine"""

            #data = read_ids_data('NSL_KDD/20 Percent Training Set.csv')
            frame = '-' * 70
            training_set = 'NSL_KDD/Small Training Set.csv'
            #training_set = 'NSL_KDD/KDDTest-21.txt'
            testing_set = 'NSL_KDD/KDDTest-21.txt'
            #testing_set = 'NSL_KDD/KDDTrain+.txt'

            print('{}\n{}\n{}'.format(frame,   '{}   detector   training...'.format(alg.__name__),
frame))
            data = read_ids_data(training_set, activity_type='normal', with_host=use_hosts_data)
            data  =  preprocessing.normalize(np.array(data,  dtype='float64'),  axis=1,  norm='l1',
copy=False)
            G = create_data_graph(data)

            gng = alg(data, surface_graph=G, output_images_dir=output_images_dir)
            gng.train(max_iterations=max_iters, save_step=50)

            print('Saving GIF file...')
            convert_images_to_gif(output_images_dir, output_gif)

            print('{}\n{}\n{}'.format(frame, 'Applying detector to the normal activity using the
training set...', frame))
            gng.detect_anomalies(data)

            for a_type in ['abnormal', 'full']:
                print('{}\n{}\n{}'.format(frame, 'Applying detector to the {} activity using the
training set...'.format(a_type), frame))
                d_data         =         read_ids_data(training_set,         activity_type=a_type,
with_host=use_hosts_data)
                d_data   =   preprocessing.normalize(np.array(d_data,   dtype='float64'),   axis=1,
norm='l1', copy=False)
                gng.detect_anomalies(d_data)

            dt = OrderedDict([('normal', None), ('abnormal', None), ('full', None)])

            for a_type in dt.keys():
                print('{}\n{}\n{}'.format(frame, 'Applying detector to the {} activity using the
testing set without adaptive learning...'.format(a_type), frame))
                d = read_ids_data(testing_set, activity_type=a_type, with_host=use_hosts_data)
                dt[a_type]  =  d  =  preprocessing.normalize(np.array(d,  dtype='float64'),  axis=1,
norm='l1', copy=False)
                gng.detect_anomalies(d, save_step=1000, train=False)

            #for a_type in ['full']:
            #    print('{}\n{}\n{}'.format(frame, 'Applying detector to the {} activity using the
testing set with adaptive learning...'.format(a_type), frame))
            #    gng.detect_anomalies(dt[a_type], train=True, save_step=1000)
```

```
        def main():
            """Entry point"""

            start_time = time.time()

            mlab.options.offscreen = True
            test_detector(use_hosts_data=False,              max_iters=7000,              alg=GNG,
output_gif='gng_wohosts.gif')
            print('Working time = {}'.format(round(time.time() - start_time, 2)))
            test_detector(use_hosts_data=True,               max_iters=7000,              alg=GNG,
output_gif='gng_whosts.gif')
            print('Working time = {}'.format(round(time.time() - start_time, 2)))
            return 0


        if __name__ == "__main__":
            exit(main())
```

| *№ док.* | *Позначення* | *Найменування* | *Дод. відом.* |
|---|---|---|---|
| | | | |
| 1 | | Текстові документи | |
| | | | |
| | ГЮІК.ХХХХХХ.1649Ст.21ПЗ | Пояснювальна записка | 86 с. |
| | | | |
| 2 | | Графічні матеріали | |
| | | | |
| | | Презентаційний матеріал | 15 слайдів |
| | | | |
| 3 | | Інші документи | |
| | | Рецензія | 1 с. |
| | | Відгук керівника | 1 с. |
| | | Електрона версія ПЗ | 1 шт. |

| Зм. | Арк. | № докум. | Підпис | Дата | ГЮІК.ХХХХХХ.1649Ст.21ПЗ | | |
|---|---|---|---|---|---|---|---|
| Розробив | | Груш В.Є. | | | Методи виявлення аномалій трафіку за допомогою нейронних мереж | Літ. | Арк. 1 / Арк. 1 |
| Перевірив. | | Федюшин О.І. | | | | | |
| Н. контр. | | Конєва Н.Ф. | | | | ХНУРЕ Кафедра БІТ | |
| Затвердив | | Халімов Г.З. | | | | | |