Додаток А

Вихідний код програми для імітаційного моделювання

```
from typing import overload
import numpy as np
from scipy.sparse import dok_matrix
from sklearn.base import BaseEstimator,
ClusterMixin


class FSoinn(BaseEstimator, ClusterMixin):

     NOISE_LABEL = -1

    def __init__(self, delete_node_period=10,
    max_edge_age=5,  init_node_num=2, sigma=0.05):
    """ :param
    delete_node_period:
      A period deleting nodes. The nodes that
      doesn't satisfy some condition are deleted
      every this period.
      :param max_edge_age:  The maximum of
      edges' ages. If an edge's age is more than
      this, the edge is deleted. :param
      init_node_num:  The number of nodes used
      for initialization.
      :param sigma: A scale parameter for Cauchy
      distribution."""
      self.delete_node_period =
      delete_node_period self.max_edge_age =
      max_edge_age self.init_node_num =
      init_node_num
      self.sigma = sigma
      self.min_degree = 1
      self.num_signal = 0
      self._reset_state()
      self.distance = 0

    def _reset_state(self):
      self.dim = None self.nodes = np.array([], dtype=np.float64)
      self.winning_times = []
      self.adjacent_mat = dok_matrix((0, 0), dtype=np.float64)
      self.node_labels = []
      self.labels_ = []
      self.labels_probability = []

    def fit(self, X):
      """ train data in batch manner
       :param X: array-like or ndarray """
      self._reset_state() for x in X:
      self.input_signal(x)
      self.labels_ = self.__label_samples(X)
      self.labels_probability=self.__label_samples_probability(X)
      return self

    def fit_predict(self, X, y=None):
      """ train data and predict cluster
      index for each sample. :param X:
      array-like or ndarray :rtype list:
      :return:
      cluster index for each sample. if a sample
      is noise, its index is FSoinn.NOISE_LABEL.
      """
```

```
  return self.fit(X).labels_

def input_signal(self, signal: np.ndarray):
  signal = self.__check_signal(signal)
  self.num_signal += 1
  if self.nodes.shape[0] < self.init_node_num:
    self.__add_node(signal)
    return
  winner, dists = self.__find_nearest_nodes(2, signal)
  sim_thresholds = self.__calculate_similarity_thresholds(winner)
  if dists[0] >= sim_thresholds[0] or dists[1] >= sim_thresholds[1]:
     self.__add_node(signal)
  else:
    self.__add_edge(winner) self.__increment_edge_ages(winner[0])
    winner[0] = self.__delete_old_edges(winner[0])
    self.__update_winner(winner[0], signal)
    self.__update_adjacent_nodes(winner[0], signal)

  if self.num_signal % self.delete_node_period == 0:
    self.__delete_noise_nodes()

@overload def __check_signal(self, signal: list) -> None:
  .
  .
  .


def __add_node(self, signal: np.ndarray):
  n = self.nodes.shape[0]
  self.nodes.resize((n + 1, self.dim))
  self.nodes[-1, :] = signal
  self.winning_times.append(1)
  self.adjacent_mat.resize((n + 1, n + 1))

def __find_nearest_nodes(self, num: int, signal: np.ndarray):
  n = self.nodes.shape[0] indexes = [0] * n
  sq_dists = [0] * n
  D = np.sum((self.nodes - np.array([signal] * n)) ** 2, 1)
  self.distance = (1/sum(D**2))
  for i in range(n):
    indexes[i] = np.nanargmin(D)
    sq_dists[i] = D[indexes[i]] D[indexes[i]] = float('nan')

  return indexes, sq_dists

def __calculate_similarity_thresholds(self, node_indexes):
  sim_thresholds = [] for i in node_indexes:
    pals = self.adjacent_mat[i, :]
    if len(pals) == 0:
      idx, sq_dists = self.__find_nearest_nodes(2, self.nodes[i, :])
    sim_thresholds.append(sq_dists[1]) else:
      pal_indexes = []
      for k in pals.keys():
         pal_indexes.append(k[1]) sq_dists =
         np.sum((self.nodes[pal_indexes] np.array([self.nodes[i] *
         len(pal_indexes)])) ** 2, 1)
         sim_thresholds.append(np.max(sq_dists))
return sim_thresholds

def __add_edge(self, node_indexes):
```

```
  self.__set_edge_weight(node_indexes, 1)



def __increment_edge_ages(self, winner_index):
  for k, v in self.adjacent_mat[winner_index, :].items():
    self.__set_edge_weight((winner_index, k[1]), v + 1)



def __delete_old_edges(self, winner_index):
  candidates = [] for k, v in
  self.adjacent_mat[winner_index,
  :].items():
  if v > self.max_edge_age + 1:
   candidates.append(k[1])
   self.__set_edge_weight((winner_index, k[1]), 0)
   delete_indexes = []
   for I in candidates:
    if len(self.adjacent_mat[I, :]) == 0:
       delete_indexes.append(i)
      self.__delete_nodes(delete_indexes)
      delete_count = sum([1 if i < winner_index else 0 for i in
    delete_indexes])
    return winner_index - delete_count

def __set_edge_weight(self, index, weight):
  self.adjacent_mat[index[0], index[1]] = weight
  self.adjacent_mat[index[1], index[0]] = weight

def __update_winner(self, winner_index, signal):
  self.winning_times[winner_index] += 1
  w = self.nodes[winner_index]
  member_f = 1 / (1 + (signal - w) ** 2 / self.sigma)
  self.nodes[winner_index] = w + (signal - w) * member_f /
  self.winning_times[winner_index]

def __update_adjacent_nodes(self, winner_index, signal):
  pals = self.adjacent_mat[winner_index]
  for k in pals.keys():
    i = k[1]
    w = self.nodes[i]
    member_f = 1 / (1 + (signal - w) ** 2 /
    self.sigma)
    self.nodes[i] = w + (signal - w) * member_f / (100 *
    self.winning_times[i])


def __delete_nodes(self, indexes):
  if not indexes:
     return n = len(self.winning_times)
    self.nodes = np.delete(self.nodes, indexes, 0)
    remained_indexes = list(set([i for i in range(n)]) - set(indexes))
    self.winning_times = [self.winning_times[i]
    for i in remained_indexes]
       self.__delete_nodes_from_adjacent_mat(indexes, n,
  len(remained_indexes))
```

```
def __label_samples(self, X: np.ndarray):
  """ :param X: (n, d) matrix whose rows are
  samples. :rtype list: :return list of labels
  """

  self.__label_nodes()
  n = len(X)
  a = 1

  labels = np.array([FSoinn.NOISE_LABEL for _ in range(n)], dtype='i')
  for i, x in enumerate(X):
    i_nearest, dist = self.__find_nearest_nodes(a, x)
    sim_threshold = self.__calculate_similarity_thresholds(i_nearest)
    if dist < sim_threshold:
      labels[i] = self.node_labels[i_nearest[0]]
  return labels

def __label_samples_probability(self, X: np.ndarray):
  """ :param X: (n, d) matrix
  whose rows are samples. :rtype
  list: :return list of labels """
  self.__label_nodes()
  n = len(X)
  a = 1
  labels = []
  for i, x in enumerate(X):
    i_nearest, dist = self.__find_nearest_nodes(a, x)
    res = []
    for node in i_nearest:
      res.append(self.node_labels[node])
     filled = []
     filledidx = []
     distance = 0
     for idx, r in enumerate(res):
     if r not in filled:
       filled.append(r)
      filledidx.append(idx)
      distance += dist[idx]
      label = []
    for ii, fill in enumerate(filledidx):
      label.append([filled[ii], dist[fill] / distance])
      labels.append(label)
    return
    labels
```

Додаток Б

Відомість атестаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	75с.
		.
	Інші документи	
2.	Презентаційні матеріали	13 плакатів

					Самонавчанна нейро-фаззі система для кластерування великих масивів даних	
.	.	Прізвище та ініціали.	Підп.	Дата		
Розробив		Іванова Є.В.				Шифр групи: Код напр./спец.
Перевірив		Бодянський Є.В.				СШІм-18-2: 122
Н.контр.		.				ХНУРЕ кафедра ШІ
Затв.		Філатов В.О.				