ДОДАТОК А

ЛІСТИНГ ПРОГРАМНОГО ЗАСТОСУНКУ

```
from copy import copy
import numpy as np
import math
import random
import logging

from pandas import DataFrame
from sklearn.preprocessing import MinMaxScaler

SMALL_VALUE = 0.00001


class FCM:

    def __init__(self, n_clusters=2, m=2, max_iter=50):
        self.n_clusters = n_clusters
        self.n_features = None
        self.cluster_centers_ = None
        self.n_points = None
        self.u = None  # The membership
        self.m = m  # the fuzziness, m=1 is hard not fuzzy. see the paper for more info
        self.max_iter = max_iter
        self.logger = logging.getLogger(__name__)
        self.logger.addHandler(logging.NullHandler())

    def get_logger(self):
        return self.logger

    def set_logger(self, tostdout=False, logfilename=None,
level=logging.WARNING):
        if tostdout:
            self.logger.addHandler(logging.StreamHandler())
        if logfilename:
            self.logger.addHandler(logging.FileHandler(logfilename))
        if level:
            self.logger.setLevel(level)

    def compute_cluster_centers(self, X):
        """
        :param X:
```

```
        :return:

        vi = (sum of membership for cluster i ^ m  * x ) / sum of membership for
cluster i ^ m  : for each cluster i

        """
        centers = []
        for c in range(self.n_clusters):
            sum1_vec = np.zeros(self.n_features)
            sum2_vec = 0.0
            for i in range(self.n_points):
                interm1 = (self.u[i][c] ** self.m)
                interm2 = interm1 * X[i]
                sum1_vec += interm2
                sum2_vec += interm1
                if np.any(np.isnan(sum1_vec)):
                    self.logger.debug("compute_cluster_centers> interm1 %s" %
str(interm1))
                    self.logger.debug("compute_cluster_centers> interm2 %s" %
str(interm2))
                    self.logger.debug("compute_cluster_centers> X[%d] %s" % (i,
str(X[i])))
                    self.logger.debug("compute_cluster_centers> loop sum1_vec %s" %
str(sum1_vec))
                    self.logger.debug("compute_cluster_centers> loop sum2_vec %s" %
str(sum2_vec))
                    self.logger.debug("X: [%d] %s" % (i-1, X[i-1]))
                    self.logger.debug("X: [%d] %s" % (i+1, X[i+1]))
                    self.logger.debug("X: ")
                    self.logger.debug(X)
                    raise Exception("There is a nan in compute_cluster_centers method
if")
            if sum2_vec == 0:
                sum2_vec = 0.000001
            centers.append(sum1_vec/sum2_vec)
        self.cluster_centers_ = np.array(centers)
        return centers

    def distance_squared(self, x, c):
        """
        Compute the Euclidean distance
        :param x: is a single point from the original data X
        :param c: is a single point that represent a center or a cluster
        :return: the distance
        """
```

```
        sum_of_sq = 0.0
        for i in range(len(x)):
            sum_of_sq += (x[i]-c[i]) ** 2
        return sum_of_sq

    def compute_membership_single(self, X, datapoint_idx, cluster_idx):
        """
        :param datapoint_idx:
        :param cluster_idx:
        :return: return computer membership for the given ids
        """
        clean_X = X
        d1 = self.distance_squared(clean_X[datapoint_idx],
self.cluster_centers_[cluster_idx])
        sum1 = 0.0
        for c in self.cluster_centers_:  # this is to compute the sigma
            d2 = self.distance_squared(c, clean_X[datapoint_idx])
            if d2 == 0.0:
                d2 = SMALL_VALUE
            sum1 += (d1/d2) ** (1.0/(self.m-1))
            if np.any(np.isnan(sum1)):
                self.logger.debug("nan is found in compute_membership_single")
                self.logger.debug("d1: %s" % str(d1))
                self.logger.debug("sum1: %s" % str(sum1))
                self.logger.debug("d2: %s" % str(d2))
                self.logger.debug("c: %s" % str(c))
                self.logger.debug("X[%d] %s" % (datapoint_idx,
str(clean_X[datapoint_idx])))
                self.logger.debug("centers: %s" % str(self.cluster_centers_))
                raise Exception("nan is found in computer_memberhip_single method in
the inner for")
        if sum1 == 0:  # because otherwise it will return inf
            return 1.0 - SMALL_VALUE
        if np.any(np.isnan(sum1 ** -1)):
            self.logger.debug("nan is found in compute_membership_single")
            self.logger.debug("d1: %s" % str(d1))
            self.logger.debug("sum1: %s" % str(sum1))
            self.logger.debug("X[%d] %s" % (datapoint_idx,
str(clean_X[datapoint_idx])))
            self.logger.debug("centers: %s" % str(self.cluster_centers_))
            raise Exception("nan is found in computer_memberhip_single method")
        return sum1 ** -1

    def update_membership(self, X):
        """
```

```
        update the membership matrix
        :param X: data points
        :return: nothing

        For performance, the distance can be computed once, before the loop instead
of computing it every time
        """
        for i in range(self.n_points):
            for c in range(len(self.cluster_centers_)):
                self.u[i][c] = self.compute_membership_single(X, i, c)

    def fit(self, X: np.ndarray):
        """
        :param X
        :return: self
        """
        self.n_points = X.shape[0]
        self.n_features = X.shape[1]
        self.random_cluster_centers()
        self.u = np.zeros((self.n_points, self.n_clusters))
        centers_history = []
        centers_history.append(self.cluster_centers_.copy())
        i = 0
        while not self.check_error(centers_history):
            self.update_membership(X)
            self.compute_cluster_centers(X)
            centers_history.append(self.cluster_centers_.copy())
            self.logger.info("updated membership is: ")
            self.logger.info(self.u)
            self.logger.info("updated cluster centers are: ")
            self.logger.info(self.cluster_centers_)
            i += 1
        self.logger.info(f'took {i} iterations')
        return self

    def check_error(self, centers_history):
        if len(centers_history) < 2:
            return False
        last_centers = self.cluster_centers_
        pre_last_centers = centers_history[-2]
        for last_cluser_center, prelast_cluster_center in zip(last_centers,
pre_last_centers):
            for i in range(len(last_centers)):
                if math.fabs(last_cluser_center[i] - prelast_cluster_center[i]) > 0.0001:
                    return False
```

```
        return True


    def random_cluster_centers(self):
        centers = []
        for i in range(self.n_clusters):
            coordinates = []
            for c in range(self.n_features):
                point = random.uniform(-1, 1)
                coordinates.append(point)
            centers.append(coordinates)
        self.cluster_centers_ = np.array(centers)
        return centers

    def credibilistic_recalculation(self):
        max_credibiliscic = self.u.max()
        for i in range(len(self.u)):
            for j in range(len(self.u[i])):
                self.u[i][j] = (self.u[i][j] + 1 - max_credibiliscic) / 2

def normalize_input_data(data: DataFrame):
    new_data = copy(data)
    # clear any string features, remain only numeric
    for col_name in new_data.columns:
        if any(type(value) is str for value in data[col_name]):
            new_data.drop(col_name, axis = 1, inplace = True)
    # delete id if exist
    new_data.drop('Id', axis=1, inplace=True, errors='ignore')
    # make normalization in range -1 to 1
    for feature_name in new_data.columns:
        column_feature = new_data[[feature_name]]
        scaler = MinMaxScaler((-1, 1)).fit_transform(column_feature)
        new_data.drop(column_feature, axis = 1, inplace = True)
        new_data[feature_name] = scaler.flatten()
    return new_data


import base64
from io import BytesIO
from matplotlib import pyplot as plt
from itertools import combinations

def draw_model_2d(fcm, test_data, name, headers):
    rgb_spectre = fcm.u.tolist()
    for i in range(len(rgb_spectre)):
```

```
        if len(rgb_spectre) == 2:
            rgb_spectre[i].append(0)
    comb = list(combinations(range(0, len(fcm.cluster_centers_[0])), 2))
    transponate_input = test_data.transpose()
    all_combination_features = list(combinations(transponate_input, 2))
    fig, axes = plt.subplots(len(all_combination_features), 2, figsize=(16, 5 *
len(all_combination_features)))
    plt.subplots_adjust(top=0.99)
    fig.suptitle(f'{name} fuzzy clustering result', fontsize=16, y=1)
    i = 0
    for features_to_compare, centers_i in zip(all_combination_features, comb):
        axes[i, 0].scatter(features_to_compare[:][0], features_to_compare[:][1],
alpha=1)
        axes[i, 0].set_ylabel(headers[centers_i[0]])
        axes[i, 0].set_xlabel(headers[centers_i[1]])
        axes[i, 1].scatter(features_to_compare[:][0], features_to_compare[:][1], c =
rgb_spectre, alpha=1)
        axes[i, 1].set_ylabel(headers[centers_i[0]])
        axes[i, 1].set_xlabel(headers[centers_i[1]])
        for cluster in fcm.cluster_centers_:
            axes[i, 1].scatter(cluster[centers_i[0]], cluster[centers_i[1]], marker="+",
s=150, c='b')
        i += 1
    tmpfile = BytesIO()
    fig.savefig(tmpfile, format='png')
    encoded = base64.b64encode(tmpfile.getvalue()).decode('utf-8')

    html = '<img src=\'{}\'>'.format(encoded)

    with open(f'{name}.html','w') as f:
        f.write(html)

import pandas as pd
import numpy as np
import logging

from fuzzycmeans.fuzzy_clustering import FCM, normalize_input_data
from fuzzycmeans.visualization import draw_model_2d


def run():
    df = pd.read_csv('top10s.csv', sep=';')
    normal_data = normalize_input_data(df)
    headers = normal_data.columns
    test_data = normal_data.to_numpy()
```

```
    fcm = FCM(n_clusters=3)
    fcm.set_logger(tostdout=True, level=logging.DEBUG)
    fcm.fit(test_data)
    print("\n\Original data")
    print(df)
    print("\n\nNormalized data")
    print(normal_data)
    print("\nCluster centers after FCM")
    print(fcm.cluster_centers_)
    draw_model_2d(fcm, test_data,'Fcm', headers)
    fcm.credibilistic_recalculation()
    fcm.compute_cluster_centers(test_data)

    df["Cluster"] = [np.argmax(row) + 1 for row in fcm.u]
    print("\n\Original data with result")
    print(df)
    print("\nCluster centers after credibilistic")
    print(fcm.cluster_centers_)

    draw_model_2d(fcm, test_data, 'Credibilistic', headers)


run()
```

Requirements.txt:

```
appdirs==1.4.4
argh==0.26.2
boto==2.49.0
entrypoints==0.3
et-xmlfile==1.1.0
fonttools==4.25.0
mkl-fft==1.3.1
mkl-service==2.4.0
mpmath==1.2.1
nltk==3.6.5
numpy==1.20.3
pandas==1.3.4
pathspec==0.7.0
patsy==0.5.2
pep8==1.7.1
Pillow==8.4.0
pkginfo==1.7.1
ply==3.11
pycosat==0.6.3
pycurl==7.44.1
```

pytz==2021.3
PyYAML==6.0
scikit-image==0.18.3
scikit-learn==0.22.1
simplegeneric==0.8.1
unicodecsv==0.14.1