```
import numpy as np
from scipy.spatial import distance_matrix as dm
import matplotlib
matplotlib.use('TkAgg')
from matplotlib import pyplot as plt
import random as ran
import math as math
import argparse

def read_and_convert_data_points(filename):
    """
    read_and_convert_data_points is a function that takes  a text file
    as an argument and converts it into an array containing
     all the data points.
    ================================================================
    :param filename: text file name in directory.
    :return: npts x 2 array.
    ================================================================
    """
    splits = []
    with open(filename, 'r') as f:
        data_points = f.readlines()

    for data_point in data_points:
        splits.append(data_point.split())

    return np.array(splits, dtype=np.float)

def scatter_plot(data, output_file_name='none',i=0):
    """
    scatter_plot is a function that plots the data and differentiate up to
    two clusters.
    ================================================================
    :param data: npts x 2, data set array.
    :param output_file_name:
    :param i: idx of the first data-point of the second cluster.
    :return:
    ================================================================
    """
    if i != 0:
        x_1, y_1 = data[0:i].T
        x_2, y_2 = data[i:data.shape[0]].T
        plt.scatter(x_1, y_1, color='r')
        plt.scatter(x_2, y_2, color = 'g')

        if output_file_name =='none':
            plt.show()
        else:
```

```
            plt.savefig(output_file_name)
            plt.show()
    else:
        x, y = data.T
        plt.scatter(x,y)
        if output_file_name =='none':
            plt.show()
        else:
            plt.savefig(output_file_name)
            plt.show()

def initial_clustering_membership_probability(clusters_number, npts):
    """
    initial_clustering_membership_probability is a function that takes
    as an argument the number of clusters 'C' and generates a random
    cluster membership probability per element 'i'
    in the data set composed by 'N'elements.
    The sum of an element cluster membership probabilities is be 1.
    ================================================================
    :param clusters_number: 'C'
    :param npts: Number of data points in the data set.
    :return initial_cond_prob: C x N matrix.
    ================================================================
    """
    initial_cond_prob = np.zeros((npts,clusters_number))
    for i in range(npts):
        x = np.random.rand(clusters_number)
        initial_cond_prob[i] = x/np.sum(x)
    return initial_cond_prob.transpose()


def blahut_arimoto_algorithm_imp(clusters_number,
                                 priors,
                                 distance_matrix,
                                 beta,
                                 npts_,
                                 rounds_,
                                 threshold=1e-10,
                                ):
    """
    the blahut_algorithm_imp function implements the Blahut Arimoto Algorithm
    for a given number of clusters and a given value of beta.
    For each round it will:
    *firstly initialize the clustering membership probabilities randomly,
    *compute the evidence for each cluster 'C'(returns a vector),
    *Compute the posterior probability per cluster: returns a matrix C x Npts
    *Compute the element to cluster distance: returns a matrix Npts x C
    After performing all the previous operations for all clusters 'C' the
```

```
normalization (partition) function is created and posteriori to this the
clustering membership probabilities are updated. The function will later try
to minimize the Lagrange multiplier function until convergence i.e, there is no
improvement or the improvement is less than the threshold. When the function
reaches this condition it gets out of the loop and starts the next round.

==============================================================
:param clusters_number:
:param priors: prior probability for each data point, if uniformly
distributed = 1/Npts
:param distance_matrix: Matrix containing the distance from every vector in x
to every vector in y.
:param beta: distortion trade-off parameter
:param npts_: Number of data points in the data set.
:param rounds_: number on rounds (iterations) perform for averaging results.
:param threshold: if the minimization of the lagrange multiplier result
compared to the previous iteration's result is less than the threshold
the round will exit the loop.
:returns: scalar value, mean of the expected distortion for each round.
==============================================================
"""

d_results=[]

for round_number in range(rounds_):
    d_ = []
    lagrange_multiplier=[]
    diff_elem_value=1
    membership_prob = initial_clustering_membership_probability(
                                                        clusters_number, npts_
                                                                )
    evidence_vector = np.zeros((clusters_number,))
    posteriori_prob = np.zeros((npts_, clusters_number)).transpose()
    element_cluster_distance_matrix = np.zeros((npts_, clusters_number))

    while diff_elem_value > threshold:

        for c in range(clusters_number):

        #     Computes the evidence for each cluster 'C': returns a vector
            evidence_vector[c] = np.sum(np.dot(membership_prob[c], priors))
        #     Computes the posterior probability: returns a matrix C x npts
            posteriori_prob[c] = (np.dot(
                                        priors,
                                        membership_prob[c]
                                        ))/evidence_vector[c]

        #     Compute element to cluster distance:
```

```
                element_cluster_distance_matrix.transpose()[c] =(

                                                np.multiply(
                                                    posteriori_prob[c], distance_matrix
                                                           )
                                                ).sum(axis=1,dtype='float')
            #compute partition function:
            exponential_beta_matrix = np.exp(-beta * element_cluster_distance_matrix)
            normalization_partition_function = (
                                np.multiply(evidence_vector,
                                            exponential_beta_matrix
                                            )).sum(axis=1)

            #update conditional cluster_probability
            membership_prob = (np.multiply
                                (evidence_vector,
                                 exponential_beta_matrix
                                )/ normalization_partition_function[:,None]
                               ).transpose()

            #Compute averaged element to cluster distance:
            joint_probability = np.multiply(evidence_vector, posteriori_prob.transpose())
            d =np.sum(np.multiply(joint_probability,element_cluster_distance_matrix))

            d_.append(d)
            ##Compute Compression rate:
            ratios = posteriori_prob.transpose()/priors
            compression_rate_ = np.sum(np.multiply(joint_probability, np.log2(ratios)))

            ##Compute Lagrange Multiplier:
            lagrange_multiplier_ = compression_rate_ + beta*d
            lagrange_multiplier.append(lagrange_multiplier_)

            if len(lagrange_multiplier) > 1:
                diff_elem_value = lagrange_multiplier[-2]-lagrange_multiplier[-1]

        d_results.append(d_[-1])
    return sum(d_results)/float(rounds_)


def batch_perform_algorithm(
                            range_betas_,
                            prior_,
                            distance_matrix,
                            npts_,
                            rounds_,
                            range_clusters=range(2, 5)
                            ):
```

```
    """
    batch_perform_algorithm will recursively call the
    blahut_arimoto_algorithm_imp function for a number of clusters and
    distortion trade-off values (beta) it will return the coordinates
    of the value that minimizes the expected distortion for a given
    distortion trade-off.
    ============================================================
    :param range_betas_:range of betas = list of the given betas
    (distortion trade-off values)
    that want to be evaluated.
    :param prior_: prior probability for each data point,
    if uniformly distributed = 1/npts
    :param distance_matrix: Matrix containing the distance f
    rom every vector in x to every vector in y.
    :param npts_: Number of data points in the data set.
    :param rounds_: number on rounds (iterations) perform for averaging results.
    :param range_clusters: range of clusters = list of the given range.
    ============================================================
    """
    results = np.zeros((len(range_clusters),len(range_betas_)))
    betas=np.zeros((len(range_betas_)))
    coordinates = np.zeros((len(range_clusters), len(range_betas_),2))

    for idx, cluster_n in enumerate(range_clusters):
        for idx_beta, beta in enumerate(range_betas_):
            results[idx, idx_beta] = blahut_arimoto_algorithm_imp(
                                                                  cluster_n,
                                                                  prior_,
                                                                  distance_matrix,
                                                                  beta,npts_,
                                                                  rounds_
                                                                          )
            betas[idx_beta]=beta

    for idx, cluster in enumerate(range_clusters):
        coordinates[idx].transpose()[0] = (-results[idx])
        coordinates[idx].transpose()[1] = betas

    return coordinates

def plot_batch_information_curve(
                                 coordinates_,
                                 range_clusters=range(2, 5),
                                 out_filename='none'
                                 ):
    """
    plot_batch_information_curve generates a plot with the given
    coordinates, where the x axis is the value of the distortion trade-off
```

```
    parameter and the y axis the (negative) expected distortion.
    ============================================================
    :param coordinates_: output of batch_perform_algorithm function,
    (C, Beta, 2) array.
    :param range_clusters: range of clusters = list of the given range
    :return: information curves plot
    ============================================================
    """
    colors = [
        '#e6194b', '#3cb44b', '#ffe119', '#4363d8',
        '#f58231','#911eb4', '#46f0f0', '#f032e6',
        '#bcf60c','#fabebe', '#008080', '#e6beff',
        '#9a6324', '#fffac8', '#800000', '#aaffc3',
        '#808000', '#ffd8b1', '#000075', '#808080',
        '#ffffff', '#000000']
    #plt.figure(figsize=(10,10))
    plt.style.use('seaborn-darkgrid')
    for i_coord, i_cluster in zip(range(coordinates_.shape[0]), range_clusters):
        y,x = coordinates_[i_coord].T
        plt.plot(x, y, marker='o', linestyle='--',
                 color=colors[i_coord],
                 label= f'Nc:{i_cluster}')
    plt.title('Information Curve')
    plt.xlabel(r'$\beta$')
    plt.ylabel(r'$<-D>$')
    plt.legend(loc='best')
    if out_filename=='none':
        plt.show()
    else:
        plt.savefig(out_filename)
        plt.show()


def main():
    """
    Main function of the program
    """
    parser = argparse.ArgumentParser(description="Blahut-Arimoto Algorithm")

    parser.add_argument("-i", "--input", metavar="INPUT_FILE",
                        help="Input file, Blast output file in xml format")
    parser.add_argument("-o", "--output", metavar="OUTPUT_PLOT_FILE",
                        help="Output information curve plot file in PNG format.",
                        default="none")
    parser.add_argument("-c", "--clusters", metavar="CLUSTERS_RANGE",
                        help="Clusters range, (2,5) if not specified",
                        default=range(2,5))
    parser.add_argument("-b", "--betas", metavar="BETAS_RANGE",
```

```
                    help="Betas range, range(1,55,5) if not specified",
                    default= [1,5,10,15,20,25,30,35,40,45,50])
parser.add_argument("-r", "--rounds", metavar="NUMBER_OF_ROUNDS",
                    help="Number of rounds the algorithm will be executed,
                    default=20)


args = parser.parse_args()
data_points = read_and_convert_data_points(args.input)
npts, Ndim = data_points.shape
range_clusters = (range(2,5) if not args.clusters else args.clusters)
range_betas = ([1,5,10,15,20,25,30,35,40,45,50] if not args.betas else args.betas)
rounds_number = (20 if not args.rounds else args.rounds)
prior = 1 / float(npts)
distance_matrix_ = dm(data_points, data_points)
plot_batch_information_curve(batch_perform_algorithm(
                                                    range_betas,
                                                    prior,
                                                    distance_matrix_,
                                                    npts, rounds_number),
                                                    range_clusters,
                                                    out_filename=args.output
                                                    )

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