Reconstructing Pairwise Comparisons Matrices Based on Differential Evolution: A Monte Carlo Study

by

Zhangao Lu

A thesis submitted in partial fulfilment

of the requirements for the degree of

Master of Science (MSc) in Computational Science

The Faculty of Graduate Studies

Laurentian University

Sudbury, Ontario, Canada

THESIS DEFENCE COMMITTEE/COMITÉ DE SOUTENANCE DE THÈSE
Laurentian Université/Université Laurentienne
Faculty of Graduate Studies/Faculté des études supérieures

Title of Thesis Titre de la thèse	Reconstructing Pairwise Comparisons Matrices Based on Differential Evolution: A Monte Carlo Study	
Name of Candidate Nom du candidat	Lu, Zhangao	
Degree Diplôme	Master of Science	
Department/Program Département/Programme	Computational Sciences	Date of Defence Date de la soutenance May 28, 2021

APPROVED/APPROUVÉ

Thesis Examiners/Examinateurs de thèse:

Dr. Waldemar W. Koczkodaj
(Supervisor/Directeur(trice) de thèse)

Dr. Miroslaw Mazurek
(Committee member/Membre du comité)

Dr. Mariusz Pelc
(External Examiner/Examinateur externe)

Approved for the Faculty of Graduate Studies
Approuvé pour la Faculté des études supérieures
Tammy Eger, PhD
Vice-President Research (Office of Graduate Studies)
Vice-rectrice à la recherche (Bureau des études supérieures)
Laurentian University / Université Laurentienne

ACCESSIBILITY CLAUSE AND PERMISSION TO USE

Abstract

Pairwise comparisons have been used in the decision-making process since antiquities. However, it is a substantial challenge to generate a PC matrix from noisy or incomplete real-life input data. This study aims to investigate the reconstruction of pairwise comparisons matrices from not-so-inconsistent pairwise comparisons matrices by an optimization method based on differential evolution. A distance-based objective function is defined as a function of the inconsistency indicator and the distance metric. Monte Carlo experiments are designed to illustrate the research outcomes. The experimental results show that this method convergence quickly. It also provides comparisons of several traditional metrics.

Keywords

pairwise comparisons, pairwise comparisons matrix, inconsistency, differential evolution, optimization, Monte Carlo, metric.

Acknowledgments

I would like to express my sincere gratitude and appreciation to my supervisor Dr. Waldemar Koczkodaj for his support and advice during my studies. His explicit knowledge in the field helped me through to the end.

I would like to thank my wife for her constant support and patience throughout this process.

I would like to thank my parents for their interest and support throughout my degree.

I would like to thank all my friends who never wavered in their support.

Contents

List of Tables

List of Figures

1 Introduction

In nature, pairs occur everywhere. A pair of binary digits is the foundation of computers. We compare objects or concepts in pairs more frequently than we realize. The pairwise comparisons (PC or PCs depending on the context) method deserves more attention than it is currently getting. There are many kinds of research for consistent PC matrices. Nevertheless, not all PC matrices are consistent matrices in practice. To generate a consistent PC matrix from a "not-so-inconsistent" PC matrix or NSI PC matrix, which was introduced in [9], is worth considering. Saaty proposed a method to solve that problem in 1977 [17]. After that, several methods have been raised [18] [4]. However, there is no decisive proof of which one is best until now. Therefore, it becomes an optimization problem. In this study, a new method is proposed based on Differential Evolution (DE) with the tolerance according to Kii (Koczkodaj inconsistency indicator [11]) and several distance measures.

2 Pairwise Comparisons

2.1 Pairwise Comparisons Basics

Pairwise comparisons method, described by Ramond Llull in the 13th century, was used for deciding elections. As a scientific method, it has evolved over hundreds of years and gained considerable importance to model inconsistency. Input data is usually represented by a square matrix with elements that are ratios between compared entities. The matrix is called a pairwise comparisons matrix (PC matrix for short). The ratio definitions imply that PC matrix elements are strictly positive real numbers. Extensions to fuzzy numbers and interval numbers have been analyzed in [22] but under the constraints outlined in [15].

In this study, only strictly positive real numbers, as PC matrix elements, will be considered. If needed, they can be generalized in time. Consider a 3×3 PC matrix:

$$M = \begin{bmatrix} 1 & m_{12} & m_{13} \\ m_{21} & 1 & m_{23} \\ m_{31} & m_{32} & 1 \end{bmatrix}$$

its elements are assumed to be reciprocal: $m_{ij} = 1/m_{ji}$ since the ration $x/y = 1/(y/x)$. It implies that PC matrix elements on the main diagonal are equal to 1 ($x/x = 1$ for $x > 0$).

Using PC matrix M elements, we can express ratios $[A/B] = m_{12}$, $[B/C] = m_{23}$,$[A/C] = m_{13}$ where A, B, and C are entities. The object $T = (m_{12}, m_{13}, m_{23})$ is called a triad, and its elements create the triangular above the main diagonal of PC matrix M. Since the elements above the main diagonal create a cycle, there may be a contradiction in the real-life situation:

$$[A/B] * [B/C] \neq [A/C]$$

where $[*]$ denotes a ratio. The ratios can be obtained by expert opinion where division operation may not be applicable (e.g., when comparing non-functional software attributes such as software reliability and software safety).

To focus our attention, assume the triad (2,5,3) in the PC matrix M above the main diagonal. It is represented by dotted arrows with solid arrowheads in Fig. 1.

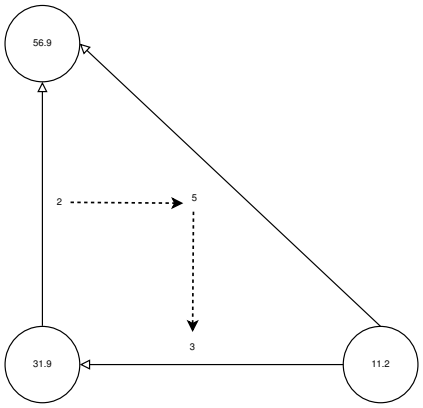


Figure 1: Triad

For better research in consistency, W.Koczkodaj proposed an indicator to measure the inconsistency illustrated in Fig. 1 in 1993 [11]. Kii is defined based on the absolute value of distances between the ratios of triads and the constant 1:

$$Kii(M) = \max_{i<j<k} \min \left(\left| 1 - \frac{m_{ik}}{m_{ij}m_{jk}} \right|, \left| 1 - \frac{m_{ij}m_{jk}}{m_{ik}} \right| \right)$$

where M is any reciprocal matrix and m_{ij}, m_{jk}, and m_{ik} are its elements. Kii is simplified in 2014 [13]:

$$Kii(M) = 1 - \min_{i<j<k} \left(\frac{m_{ik}}{m_{ij}m_{jk}}, \frac{m_{ij}m_{jk}}{m_{ik}} \right) \tag{2.1}$$

The range of Kii is $[0, 1)$. It means the matrix is consistent if $Kii = 0$, and

the matrix is inconsistent when $Kii \to 1$. Moreover, this indicator guarantees monotonicity. If N is a PC submatrix of M, we have $Kii(N) \leq Kii(M)$ [14].

2.2 Problem Definition

This paper will address reconstructing PC matrices from NSI PC matrices. An example of this is people living in a bartering economy where they exchange goods for goods. In this circumstance, people need to remember plenty of rules about exchanging goods. For example, two tomatoes equal a chicken or two chickens equal three fish. For five goods, there are $5^2 - 5 = 20$ rules between them. These rules are symmetrical, therefore, people only need to remember 10 of them. See Fig. 2. Things become more difficult when the number of goods increases to 10, meaning there are now 45 rules to remember. By comparison, for 20 items, the number of rules increases to 140 and although this is still a small number compared to the number of merchandise items in life, the increase is substantial. In this situation, the probability of contradictions will increase significantly. Considering a practical example, some people claim that two tomatoes can exchange one chicken and three chickens are equal to one fish. However, they hold that five tomatoes equal one fish. There is a classic $(2, 3, 5)$ contradiction that sets up some interesting

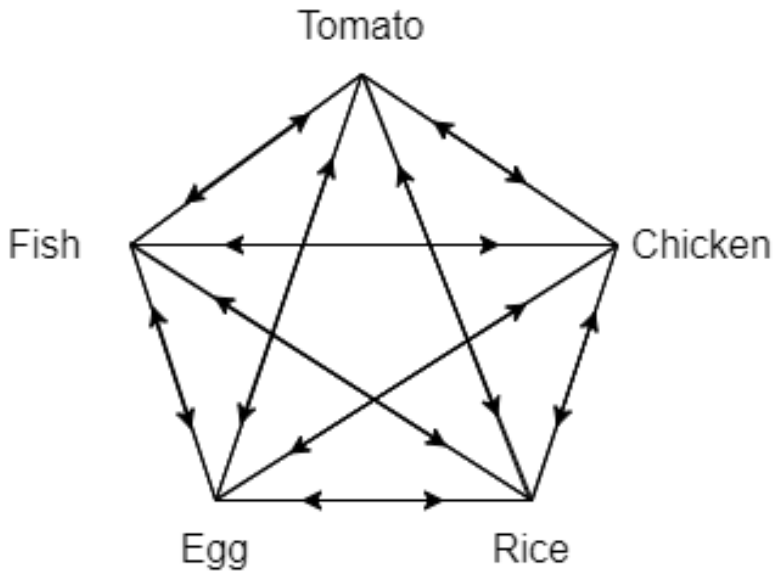


Figure 2: Sample Input Data

arbitrage opportunities. The rule can be rebuilt to solve this problem. For example, six tomatoes can exchange one fish. Now it looks like the problem is solved, and the rule is changed from $(2, 3, 5)$ to $(2, 3, 6)$. But why must it be $(2, 3, 6)$? Triad $(2, 2.5, 5)$ or triad $(1, 5, 5)$ can also satisfy the consistency condition. Moreover, if there are 40 contradictions in 140 rules, it will be tough to change the rules with all constraints.

A formal definition of this problem is that if there is an NSI PC matrix M'. The goal is to find a PC matrix M that differs from the original NSI PC matrix M' as little as possible. There are two constraints here: the

new matrix must be a consistent PC matrix, and this new matrix should be close to the original NSI matrix as much as possible. In other words, the distance between the two matrices should incline to zero. [12] proposed a distance-based inconsistency reduction algorithm with Kii, which has a quick convergence rate. This algorithm generates the consistency PC matrix with less than ten reductions in most cases if the Kii of the original NSI PC matrix is lower than $\frac{1}{3}$. However, this algorithm is developed only based on Kii from the beginning, and it cannot optimize other indicators or metrics together. Therefore, a new method should be designed to tackle mixed problems.

3 Differential Evolution

Differential evolution is a population-based evolutionary meta-heuristic, introduced by Storn and Price in 1996 [20]. Lampinen and Storn illustrated that DE was more accurate than some other optimization algorithms like simulated annealing and evolutionary programming in 2004 [16]. This method is widely applied in numerous branches of science. It is also used for solving engineering problems since the late 1990s, as documented in [5].

Although it does not guarantee a globally optimal solution, DE is regarded as a robust and powerful method with good convergence speed. Furthermore, it does not require the objective (goal) function to be differentiable, while differentiability is an essential condition for most of the classic global optimization methods (e.g., gradient descent). DE can be used to find approximate solutions to non-linear, non-convex, and non-differentiable objective functions. Generally, the optimization goal of the DE algorithm is to minimize the objective function:

$$f(X) : \mathbb{R}^n \to \mathbb{R}$$

$$X = [x_1, x_2, \cdots, x_n], X \in \mathbb{R}^n$$

by optimizing its argument X and get X^*:

$$f(X^*) \leq f(X), \forall X \in \mathbb{R}^n$$

X is an n-dimension vector, and its elements are subject to some boundary constraints:

$$L_i \leq x_i \leq U_i, i = 1, 2, \cdots, n$$

DE meta-heuristic can be described as four steps:

1. Initialization
2. Mutation
3. Crossover
4. Selection

See Fig. 3 for the block diagram of it.

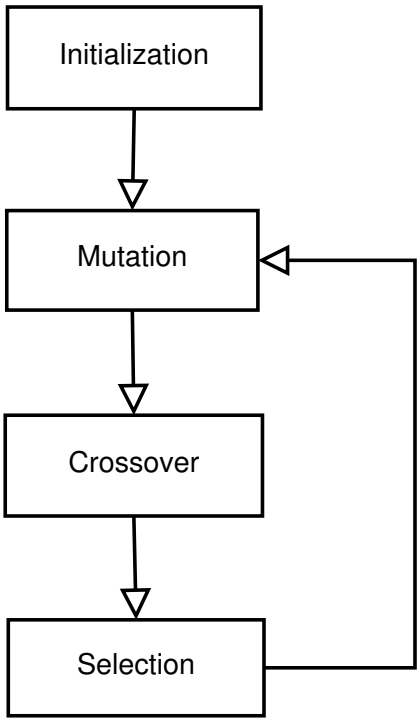


Figure 3: Differential Evolution Processes

Initialization: In the absence of the initial solution, the meta-heuristic may randomly select parameter vectors. Each vector represents a candidate solution for the objective function. We denote iterations in DE by $t = 1, 2, \cdots, t_{max}$. The pth vector of the population for the iteration t can be denoted by:

$$X_p^{(t)} = [x_{1,p}^{(t)}, x_{2,p}^{(t)}, \cdots, x_{n,p}^{(t)}]$$

where $x_{i,p}^{(t)}$ is a uniformly distributed number between L_i and U_i and can be represented by:

$$x_{i,p}^{(t)} = L_i + rand_{ip}[0,1](U_i - L_i)$$

Mutation: For each target vector $X_p^{(t)}$ in the current iteration t, DE generates a mutant vector $V_{p+1}^{(t)}$. Besides, the component of the mutant can be computed by:

$$v_{i,p}^{(t+1)} = x_{r_1,p}^{(t)} + F \cdot (x_{r_2,p}^{(t)} - x_{r_3,p}^{(t)}),\ F \in [0,2]$$

where r_1, r_2 and r_3 are randomly selected from $\{1, 2, \cdots, n\}$ and $r_1 \neq r_2 \neq r_3$. Crossover: For diversity of the parameters, the donor vector $V_{p+1}^{(t)}$ combines its entries with the target vector $X_p^{(t)}$. Hence, we generate a trial vector $U_p^{(t+1)}$, where its components can be denoted by:

$$u_{ip}^{(t+1)} = \begin{cases} v_{i,p}^{(t+1)}, & if\ rand_{ip}[0,1] \leq CR, \\ x_{i,p}^{(t)}, & otherwise \end{cases}$$

where CR or crossover rate is a pre-fixed constant $\in [0, 1]$.

Selection: DE decides whether the target vector $X_p^{(t)}$ or the trail vector $U_p^{(t+1)}$ exists in the next iteration based on:

$$X_p^{(t+1)} = \begin{cases} U_p^{(t+1)}, & if\ f(U_p^{(t+1)}) \leq f(X_p^{(t)}), \\ X_p^{(t)}, & otherwise \end{cases}$$

where f is the objective function.

DE repeats Mutation, Crossover and Selection until some threshold is reached. Subsequently, the components of the vector $X_p^{(t)}$ are the optimized parameters for the objective function.

4 Problem Formulation

4.1 The NSI PC Matrices

The "not-so-inconsistent" or NSI PC matrices should be generated randomly with some criteria before optimized. It is clear that there is no scientific merit to optimize completely random matrices. There are numerous solutions for completely random matrices. Since there are no original PC matrices for these completely random matrices, the solution PC matrices cannot be compared with the original PC matrix. Thus, it is unknown which solution is the best and closest to the original matrix. [12] created these matrices by deviating a consistent PC matrix M randomly: $M' = M * \text{rand}()$. Meanwhile, [8] proposed a different formula: $M' = M * (1 \pm \rho D)$, where $\rho \in [0, 1]$ and D is a given constant. The former method built matrices by multiplying a fixed constant and did not consider the inter elements difference. The latter solved that problem using a random number ρ but has not dealt with possible negative numbers. Thus, a new method based on the distribution of errors is proposed to build these NSI matrices.

The elements m_{ij} of the PC matrix M are defined as ratios of entities, said V_i. Hence, there must be some errors if $m_{ik} \neq m_{ij} * m_{jk}$. According to the

central limit theorem, each error e_{ij} follows a normal distribution. In this paper, for convenience of computation, normal distributions with $\mu = 0$ are applied when computing each error e_{ij}. However, it is not easy to set an appropriate value for standard deviation. Therefore, a Monte Carlo experiment is designed to find the best estimate for the standard deviation. First, we generate 100,000 PC matrices for each order. Then the standard deviation of the normal distribution is defined here to deviate these PC matrices:

$$\sigma = \rho * m_{ij}$$

where $\rho \in (0, 1]$ and m_{ij} is the corresponding element of the PC matrix. It is a reasonable estimate for standard deviation because of the 68-95-99.7 rule. Next, 100,000 NSI PC matrices were generated by $m_{ij}+e_{ij}$ (If $m_{ij}+e_{ij} <= 0$, we discard e_{ij} and generate a new one to replace it). After that, Kii for all NSI PC matrices were computed, and the arithmetic mean of these Kii were determined. Finally, a diagram to illustrate the results was created. See Fig. 4.

The diagram shows ρ and the arithmetic mean of Kii are linearly dependent when $order = 3$. However, the curve is more like a parabola when $order > 3$.

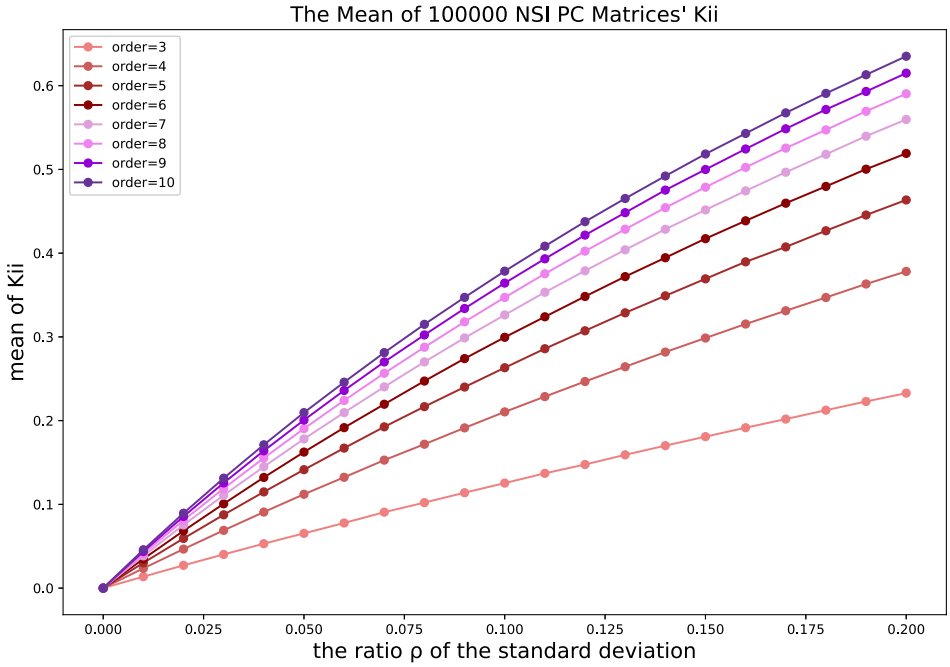


Figure 4: The Mean of 100,000 NSI PC Matrices' Kii

Therefore, we can fit the curve with the quadratic function: $\text{mean}(Kii) = a\rho^2 + b\rho$, where a and b are constants. The result is displayed in Fig. 5. The dashed lines represent graphs of quadratic functions and fit the original curve almost perfectly. In addition, if we set a threshold for Kii of NSI PC matrices, like 0.1, we can compute the value of ρ by the equation $0.1 = a\rho^2 + b\rho$. See Table. 1. Subsequently, we can generate NSI PC matrices quantitatively using the value of ρ in this table.

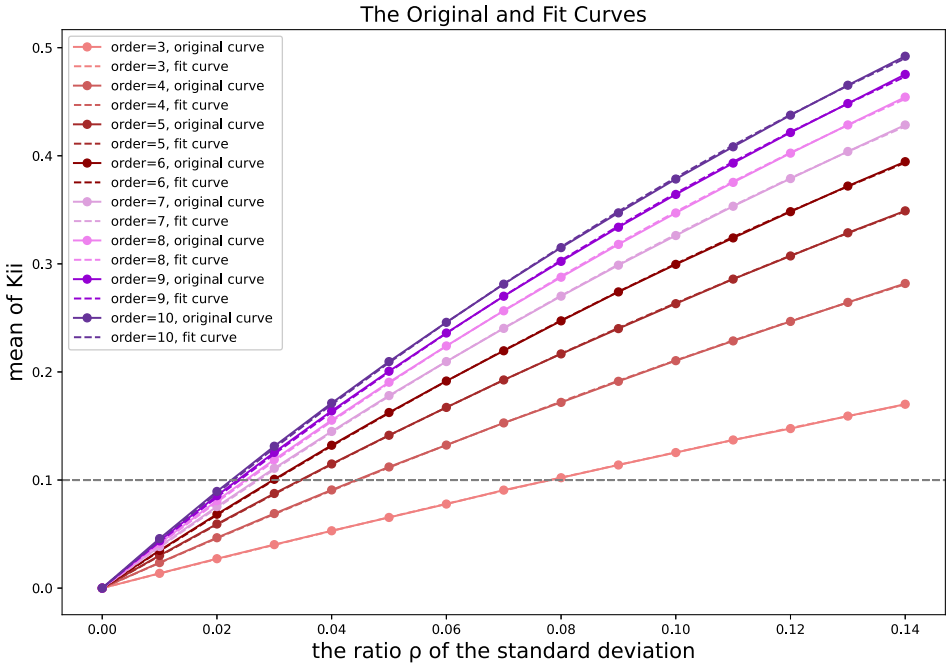


Figure 5: The Original and Fit Curve

4.2 Selection of Metrics

Now, 100,000 not-so-inconsistent PC matrices (The mathematical expectation of these matrices' Kii equals 0.1) is generated. And the indicator Kii can be used to measure a matrix's consistency. However, there is no standard measure of "close". In other words, to measure the distance between NSI PC matrices and PC matrices, some metrics should be defined or applied here. It is essential to compare the element-wise matrix distance metrics before

Table 1: The Coefficients of Quadratic Function

Threshold (Mean of Kii)	Order	ρ	a	b
0.1	3	0.0781	-1.0915	1.3653
0.1	4	0.0446	-2.4776	2.3549
0.1	5	0.0347	-3.7161	3.0093
0.1	6	0.0300	-4.7447	3.4753
0.1	7	0.0272	-5.5261	3.8247
0.1	8	0.0253	-6.2311	4.1051
0.1	9	0.0239	-6.8591	4.3401
0.1	10	0.0229	-7.4362	4.5422

applying the differential evolution algorithm to reconstruct PC matrices. In this sense, measuring the matrix distance is equivalent to computing the distance or similarity between vectors flattened from the matrix. There are several metrics to measure the distance. See Table. 2. It is worth noting that not all common metrics have been listed. For example, Minkowski distance, Manhattan distance or Pearson correlation coefficient has been tested and removed. Minkowski distance and Manhattan distance have similar characteristics to Euclidean distance. For Pearson correlation coefficient, the result

shows that Cosine similarity is generally superior to Pearson correlation coefficient.

Table 2: Several Metrics

Name	Formula	Range
Bray–Curtis distance	$d(u,v) = \frac{\sum_i(\lvert u_i - v_i\rvert)}{\sum_i(\lvert u_i + v_i\rvert)}$	$[0,1]$
Canberra distance	$d(u,v) = \sum_{i=1}^{n} \frac{\lvert u_i - v_i\rvert}{\lvert u_i\rvert + \lvert v_i\rvert}$	$[0,n]$
Chebyshev distance	$d(u,v) = \max_i \lvert u_i - v_i\rvert$	$[0,\infty)$
Cosine similarity	$\cos\theta = 1 - \frac{u\cdot v}{\lvert\lvert u\rvert\rvert_2 \lvert\lvert v\rvert\rvert_2}$	$[0,\ 1]$
Euclidean distance	$d(u,v) = (\sum_i \lvert u_i - v_i\rvert^2)^{\frac{1}{2}}$	$[0,\infty)$
Jensen–Shannon divergence	$JSD(P\lvert\lvert Q) = \frac{1}{2}D(P\lvert\lvert R) + \frac{1}{2}D(Q\lvert\lvert R)$	$[0,1)$
Kullback-Leibler divergence	$D(P\lvert\lvert Q) = \sum_i P(x)\log\frac{P(x)}{Q(x)}$	$[0,\infty)$

For better display, a Monte Carlo experiment is designed to compare these metrics in Table. 2. First, 100,000 random PC matrices are created for each order between 3 and 10. The NSI PC matrix corresponding to each PC matrix is then generated through the method introduced before. This method also can ensure the matrix's Kii is equal to 0.1 by setting the ratio ρ according to Table. 1. After that, the distance or similarity between each

pair of PC matrices and NSI PC matrices is computed. Finally, the figures are drawn to scale the density and distribution of these metrics. Letter-Value box plots instead of box plots are used here. There are 700,000 samples or points that need to be displayed in the same diagram. Meanwhile, the box plot doesn't work well with a large number of outliers, and the Letter-Value box plot is designed to handle big data [7]. Fig. 6 to Fig. 12 are Letter-Value box plots. The X-axis denotes these matrices' order, while the Y-axis refers to the values of these metrics. The diamond points are the outliers. The black line in the middle of the most oversized box is the median of these metrics with respect to the specific order. The upper and lower limit of the most oversized box denotes 75% and 25%. For the second biggest box, the limits are 87.5% and 12.5%. At last, the box widths are proportional to the number of inside points. Here are the analysis for each metric:

i. Bray–Curtis distance

   Bray–Curtis distance considers the vector space as grids. Similar to Manhattan distance, it computes the distance with absolute values. Fig. 6 shows that the values of this metric locate in a small range, and the range converges as the matrix order increases. It is not a suitable property here since it converges too fast to show the differences between

orders. Also, there are some outliers in Fig. 6. The outliers approach the median of distances as order increases.

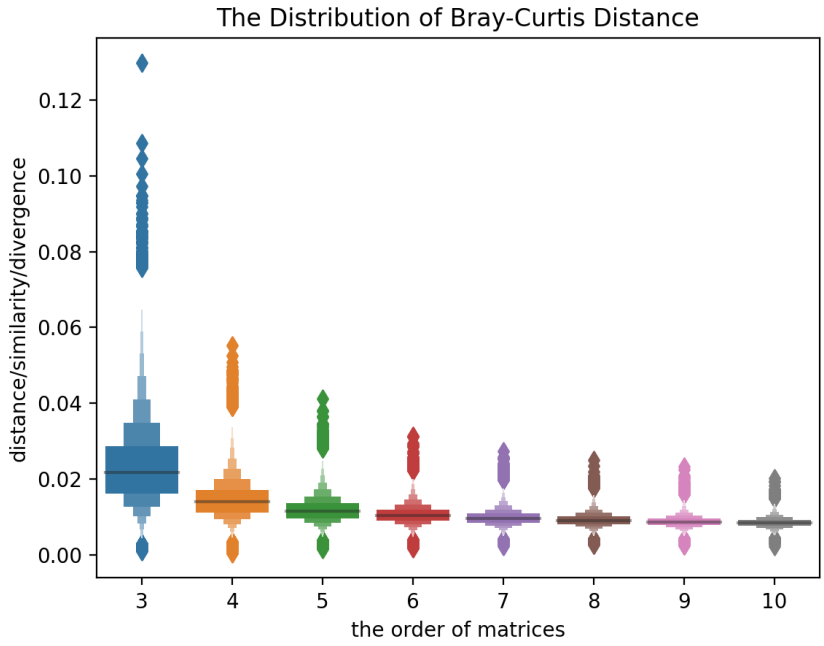


Figure 6: The Distribution of Bray-Curtis Distance

ii. Canberra distance

Canberra distance also applies absolute values to measure the distance. It may have comparable properties with Bray–Curtis distance or Manhattan distances. However, Fig. 7 presents a divergent view. This metric is highly distinguishable for each order. The graphs of each or-

der are similar, and the only difference is the values of mathematical expectations. Although the range of Canberra distance is $[0, n]$ where n is the number of matrix elements, it can be treated as $[0, 1]$ here. The figure illustrates that nearly all distances are lower than one when the matrix order is not greater than ten.

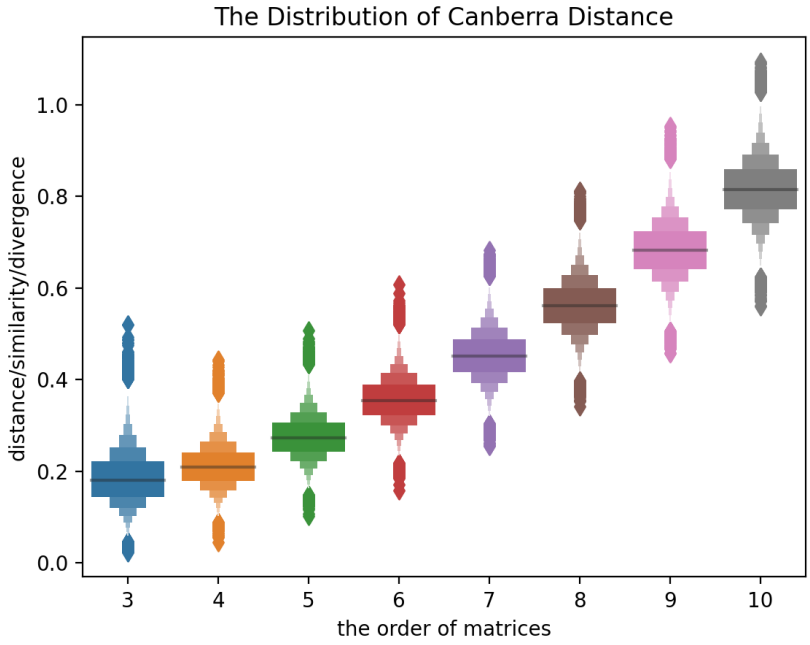
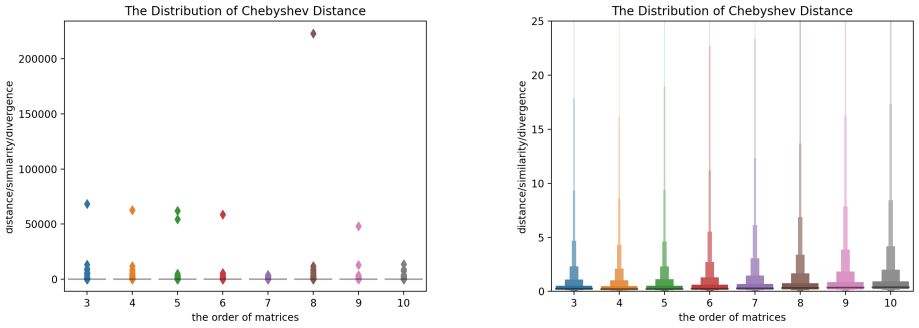


Figure 7: The Distribution of Canberra Distance

iii. Chebyshev distance

Chebyshev distance is a metric to compute the maximum element-wise

difference. It can be seen that the outliers will be a serious problem since this distance only calculates the absolute value of their differences. Fig. 8 illustrates that. Fig. 8(a) shows the maximum value is over 200,000 while the median of distances for $order = 8$ is almost zero. With the exception of outliers, the Chebyshev distance is stable. Most of the distances are located in $[0, 5]$, no matter which order the matrices have. Fig. 8(b) is included to provide more detail.



(a) Full view (b) Partial Enlarged View

Figure 8: The Distribution of Chebyshev Distance

iv. Cosine similarity

Cosine similarity is applied to measure the similarity between two vectors.The range of this metric is also [0, 1]. Fig. 9 demonstrates its graph is quite similar to Fig. 6. However, its value range converges more rapidly than Bray-Curtis distance as the order increases. Be-

sides, it also has more outliers.

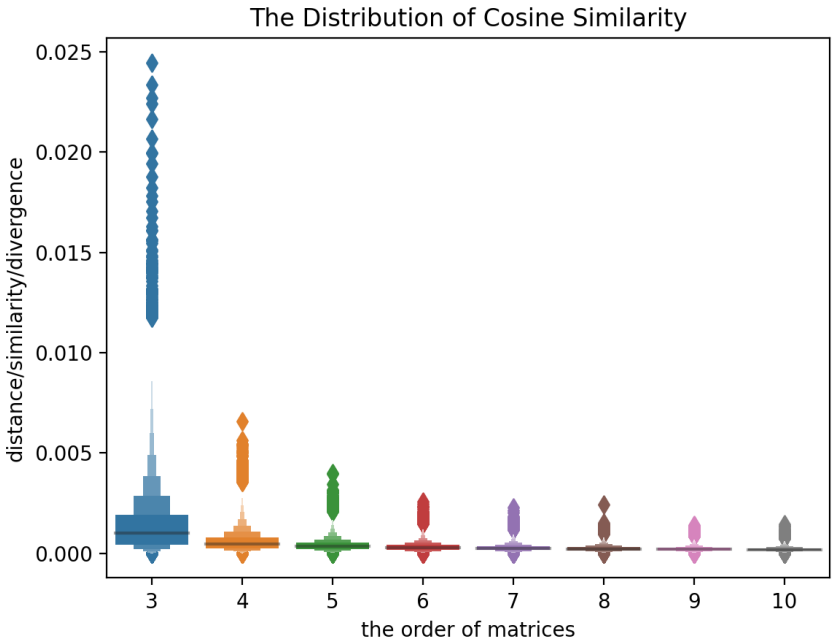
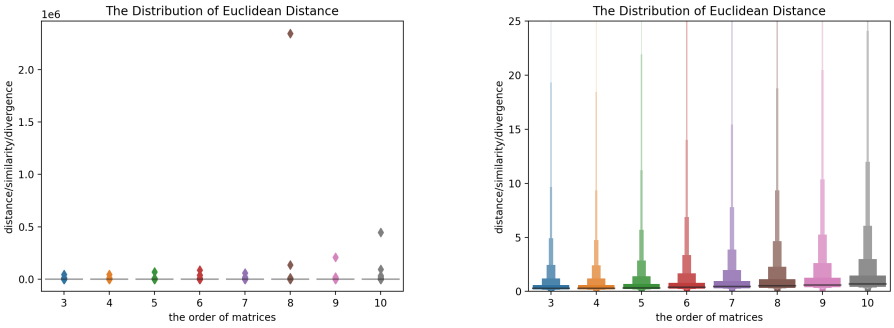


Figure 9: The Distribution of Cosine Similarity

v. Euclidean distance

Euclidean distance is the most popular distance metric. It is defined by the length of a line connected to two points. Obviously, Euclidean distance has the same problem as Chebyshev distance. Of note, the figures are notably similar. Fig. 10 demonstrates that. Nevertheless, Fig. 10(a) shows a higher density of data. All of the boxes are around zero. At last, Fig. 10(b) illustrated Euclidean distance is stable with

orders which are similar to Chebyshev distance.



(a) Full view (b) Partial Enlarged View

Figure 10: The Distribution of Euclidean Distance

vi. Jensen–Shannon divergence

Jensen–Shannon divergence or information radius is a metric to compute the similarity between two probability distributions. Although its range is $[0, 1]$, Fig. 11 shows that the value will be lower than 0.1 when the order is from $[3, 10]$. Moreover, the whole graph illustrates an explicit trend of the means, which is similar to the figure of Cosine similarity or Bray–Curtis distance, although these metrics have different theories and algorithms.

vii. Kullback-Leibler divergence

Kullback-Leibler divergence or relative entropy is commonly used as the loss function in DNNs(Deep Neural Networks). It is also the base

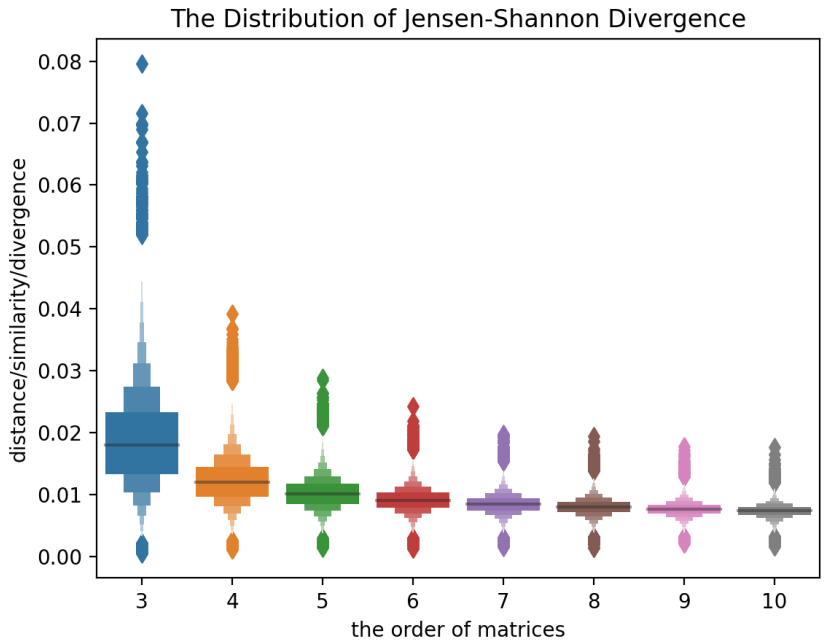


Figure 11: The Distribution of Jensen-Shannon Divergence

and precondition of Jensen–Shannon divergence. However, it does not perform well here. There are more outliers in Fig. 12, and the range convergences too fast.

In addition to these graphs, Table. 3 and Table. 4 analyze these distances statistically. To support this further, the distance data set generated by the matrices whose orders equal 4 and 8 are analyzed. It is clear that the standard deviation of Chebyshev distance and Euclidean distance are incredibly high, which indicates these distances are spread out widely. The maximum

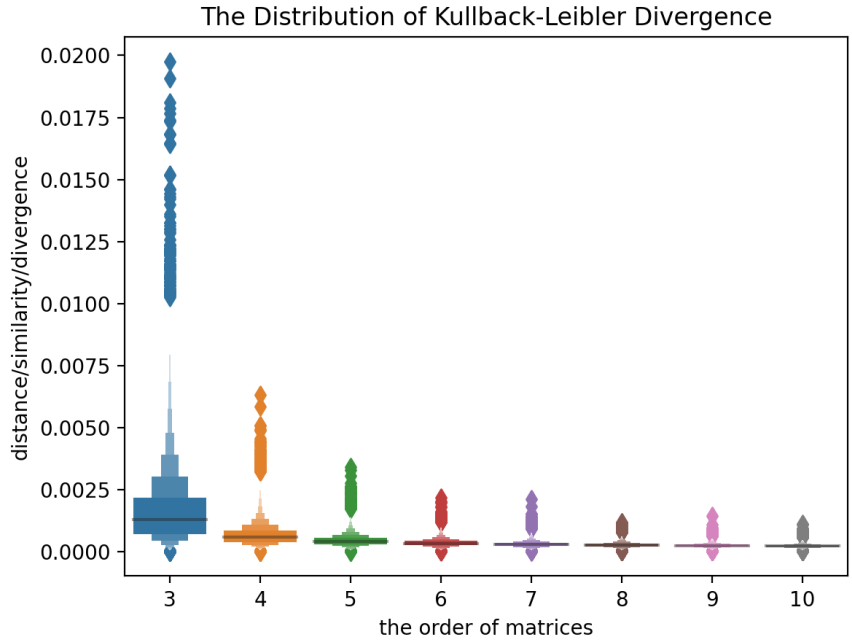


Figure 12: The Distribution of Kullback-Leibler Divergence

value denotes that on the other side. With regard to the above mentioned, it is not a good idea to set Chebyshev distance or Euclidean distance as the distance metric. For Cosine similarity and Kullback-Leibler divergence, there is another problem. Their maximum values are lower than 10^{-2}, which means they are hardly distinguishable for different orders. Thus, Bray-Curtis distance, Canberra distance and Jensen-Shannon divergence are kept for further research.

Table 3: Statistical Analysis for Order=4 Matrices

Name	Mean	SD	Min	25%	50%	75%	Max
Bray-Curtis	0.0146	0.005	0.0004	0.0112	0.0139	0.0172	0.0552
Canberra	0.2112	0.0465	0.0455	0.1787	0.209	0.2414	0.4436
Chebyshev	2.3262	207.7218	0.021	0.1265	0.2106	0.4761	62732
Cosine	0.0006	0.0005	0.0	0.0003	0.0005	0.0008	0.0066
Euclidean	2.2778	158.2255	0.0475	0.1927	0.2882	0.5825	46397
Jensen-Shannon	0.0122	0.0037	0.0011	0.0097	0.012	0.0145	0.0392
Kullback-Leibler	0.0007	0.0004	0.0	0.0004	0.0006	0.0008	0.0063

4.3 Metric Monotonicity

In this subsection, the distributions of these metrics concerning different means of Kii are discussed. Here and subsequently, we denote the mean of Kii briefly by κ. κ can be represented by $\kappa = \frac{\sum_{i=1}^{n} Kii}{n}$. The above research focuses on the distances, similarities or divergences with respect to the same κ, which is 0.1. However, it is unknown whether these metrics are increasing or not as the κ is increasing. To address this question, another form is applied to represent the distance function D. It is recognized that the NSI PC matrix M' is generated from a PC matrix M by setting ratio ρ. Furthermore, ρ is

Table 4: Statistical Analysis for Order=8 Matrices

Name	Mean	SD	Min	25%	50%	75%	Max
Bray-Curtis	0.0092	0.0017	0.0026	0.0081	0.0091	0.0101	0.0251
Canberra	0.5619	0.057	0.3415	0.5228	0.5608	0.5996	0.8117
Chebyshev	4.7233	708.8502	0.0387	0.1696	0.3173	0.7656	223137
Cosine	0.0002	0.0001	0.0	0.0002	0.0002	0.0003	0.0024
Euclidean	28.1736	7427.259	0.1358	0.334	0.5223	1.1145	2344745
Jensen-Shannon	0.008	0.0014	0.0014	0.0071	0.008	0.0088	0.0194
Kullback-Leibler	0.0003	0.0001	0.0	0.0002	0.0003	0.0003	0.0012

determined by the mean of Kii κ. Hence, we have:

$$\begin{aligned} D(M, M') &= D(M, h(M, \rho)) \\ &= D(M, h(M, l(\kappa))) \\ &= g(\kappa) \end{aligned} \tag{4.3}$$

where M is a constant matrix. Consequently, it is equivalent to check whether $g(\kappa) = D(M, M')$ is increasing or decreasing on an interval $\kappa \in [a, b]$. If this function $g(\kappa)$ is a monotonic function or the mean of $g(\kappa)$ is a monotonic

function when the sample size is quite large, the goal can be achieved by reconstructing a PC matrix M from an NSI PC matrix M' by optimizing the goal function $f(M, M') = Kii(M') + \alpha D(M, M')$, where α is a constant. In order to check the graph visually, 100,000 random NSI PC matrices are created for each order between 3 and 10 and each κ between 0.8 and 1.3. There are 4,200,000 matrices in total. The mean and standard deviation for every 100,000 random NSI PC matrices is computed. For the purpose of illustrating results more clearly, bubble charts are used to display the relations between each parameter. See Fig. 13 to Fig. 16. It is obvious that X-axis refers to the order of the matrices, and Y-axis is defined as the mean of metric values. Besides, the radius of each bubble is the standard deviation of metrics. Hence, if bubble A is higher and bigger than bubble B, it implies that the mean and standard deviation of these metrics denoted by A are larger than these statistical measures of B. In other words, the metric values of A are larger than B on average and are more spread out. Here are the analyses for each metric:

i. Bray–Curtis distance

   The Fig. 13 shows $g(\kappa) = D_{Bray-Curtis}(M, M')$ is increasing as κ increases. Moreover, the distances converge to their mean as the order

increases, which is consistent with Fig. 6. It is impossible to prove

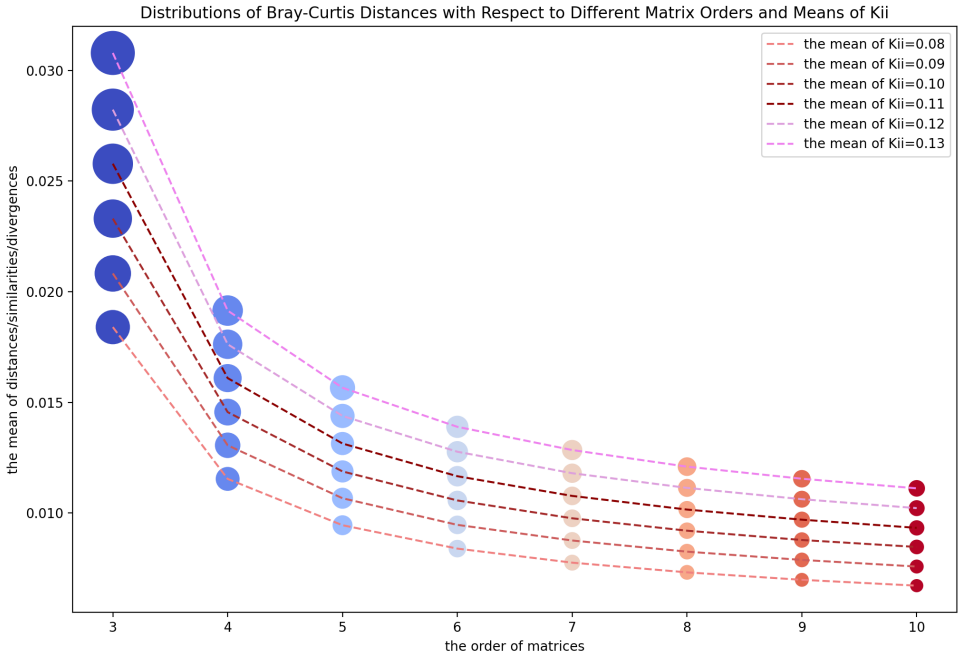


Figure 13: Distributions of Bray-Curtis distances with respect to Different Matrix Orders and Means of Kii

$g(\kappa)$ is a monotonic function. However, it is also worth knowing that the mean of $g(\kappa)$ is monotonically increasing or not when the sample set is large enough. In order to prove $\frac{\sum_k g(\kappa)}{k}, k \to \infty$ is a monotonically increasing function, we assume that ρ is proportional to κ, which is illustrated in Fig. 4. When κ increases, ρ increases. For now on, we denote PC matrix $M = [m_{ij}] \in \mathbb{R}_+^{n \times n}$ and the NSI PC matrix

$M' = [m'_{ij}] \in \mathbb{R}_+^{n \times n}$. According to the method mentioned above, we have:

$$\begin{aligned} m'_{ij} &= m_{ij} + random(\mathcal{N}(\mu, \sigma^2)) \\ &= m_{ij} + random(\mathcal{N}(0, (\rho m_{ij})^2)) \\ &= m_{ij} + random(\mathcal{N}(0, (l(\kappa) m_{ij})^2)) \end{aligned} \tag{4.31a}$$

where $\rho = l(\kappa)$ and $l(\kappa)$ is an increasing function on the interval $\kappa \in [0, 1)$. Based on the definition of Bray-Curtis distance, we have:

$$D(M, M') = \frac{\sum_{ij}(|m_{ij} - m'_{ij}|)}{\sum_{ij}(|m_{ij} + m'_{ij}|)} \tag{4.31b}$$

From (4.31a) and (4.31b), we conclude that:

$$\begin{aligned} D(M, M') &= \frac{\sum_{ij}(|m_{ij} - m_{ij} - random(\mathcal{N}(0, (l(\kappa) m_{ij})^2))|)}{\sum_{ij}(|m_{ij} + m_{ij} + random(\mathcal{N}(0, (l(\kappa) m_{ij})^2))|)} \\ &= \frac{\sum_{ij} |random(\mathcal{N}(0, (l(\kappa) m_{ij})^2))|}{\sum_{ij} |2m_{ij} + random(\mathcal{N}(0, (l(\kappa) m_{ij})^2))|} \end{aligned}$$

For abbreviation, $random(\mathcal{N}(0, (l(\kappa) m_{ij})^2)$ is denoted by n_{ij}. We get:

$$D(M, M') = \frac{\sum_{ij} |n_{ij}|}{\sum_{ij} |2m_{ij} + n_{ij}|} \tag{4.31c}$$

The task is now to find how n_{ij} changes while κ increases. Let Δn_{ij} represents the change of n_{ij} when κ increases. It is clear that the probability of generating bigger random numbers is increasing as the standard deviation $l(\kappa) * m_{ij}$ increases because of the definition of Normal distribution. Consider the law of large numbers, we obtain:

$$\lim_{i,j\to\infty} \frac{\sum_{ij} |n_{ij} + \Delta n_{ij}|}{ij} \geq \lim_{i,j\to\infty} \frac{\sum_{ij} |n_{ij}|}{ij}$$

If we denote the new distance by $D(M, M'')$, it can be represented by adding a positive number c_{ij} to the numerator and denominator of (4.31c):

$$D(M, M'') = \frac{\sum_{ij} |n_{ij}| + \sum_{ij} c_{ij}}{\sum_{ij} |2m_{ij} + n_{ij}| + \sum_{ij} c_{ij}}$$

Thus, according to the mediant inequality and $m_{ij} > 0$, we have:

$$\frac{\sum_{ij} |n_{ij}|}{\sum_{ij} |2m_{ij} + n_{ij}|} < \frac{\sum_{ij} |n_{ij}| + \sum_{ij} c_{ij}}{\sum_{ij} |2m_{ij} + n_{ij}| + \sum_{ij} c_{ij}} < \frac{\sum_{ij} c_{ij}}{\sum_{ij} c_{ij}} = 1$$

This is to say,

$$D(M, M') < D(M, M'')$$

where M, M' and M'' are r by r matrices and $r \to \infty$. Hence, consider

the mean of k matrices which their orders are small but $k \to \infty$, we have:

$$\frac{\sum_k D(M, M')}{k} < \frac{\sum_k D(M, M'')}{k}$$

Moreover, it is proved that $D(M, M')$ can be represented by the function of κ, see (4.3). There is:

$$\frac{\sum_k g(\kappa_1)}{k} < \frac{\sum_k g(\kappa_2)}{k}, if\ \kappa_1 < \kappa_2$$

where $k \to \infty$, and the proof is complete.

ii. Canberra distance

The Fig. 14 shows $g(\kappa) = D_{Canberra}(M, M')$ is increasing as κ increases. Generally, there is no change for the distances as the order increases, which is also consistent with Fig. 7. The proof for Canberra distance is similar. Based on the definition of Canberra distance, we have:

$$D(M, M') = \sum_{ij} \frac{|m_{ij} - m'_{ij}|}{|m_{ij}| + |m'_{ij}|} \tag{4.32a}$$

According to (4.31a), we can substitute m'_{ij} with $m_{ij}+random(\mathcal{N}(0, (l(\kappa)m_{ij})^2))$

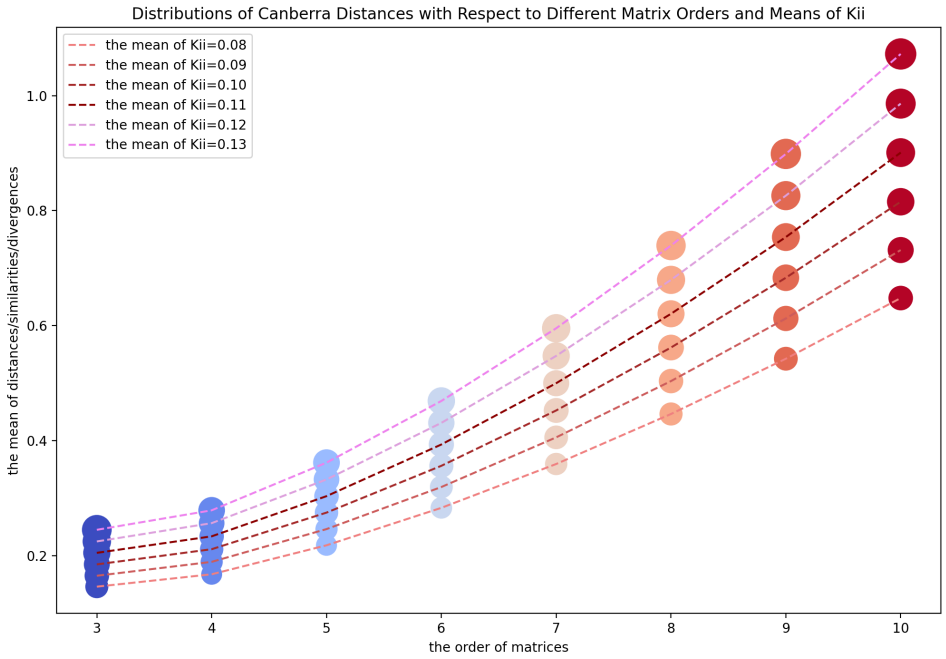


Figure 14: Distributions of Canberra Distances with respect to Different Matrix Orders and Means of Kii

and get:

$$D(M, M') = \sum_{ij} \frac{|random(\mathcal{N}(0, (l(\kappa)m_{ij})^2))|}{|m_{ij}| + |m_{ij} + random(\mathcal{N}(0, (l(\kappa)m_{ij})^2))|}$$

And again, we denote $random(\mathcal{N}(0, (l(\kappa)m_{ij})^2)$ by n_{ij}:

$$D(M, M') = \sum_{ij} \frac{|n_{ij}|}{|m_{ij}| + |m_{ij} + n_{ij}|} \tag{4.32b}$$

Thus, to prove $g(\kappa)$ is an increasing or decreasing function, it equals to prove the formula below:

$$
\begin{aligned}
g(\kappa_2) - g(\kappa_1) &= D(M, M'') - D(M, M') \\
&= D([m_{ij}], [m''_{ij}]) - D([m_{ij}], [m'_{ij}]) \\
&= \sum_{ij} \frac{|n_{ij} + \Delta n_{ij}|}{|m_{ij}| + |m_{ij} + n_{ij} + \Delta n_{ij}|} - \sum_{ij} \frac{|n_{ij}|}{|m_{ij}| + |m_{ij} + n_{ij}|} \\
&> 0 \ \textit{or} \\
&< 0
\end{aligned}
$$

where $m_{ij} > 0$, $m'_{ij} = m_{ij} + n_{ij} > 0$ and $m''_{ij} = m_{ij} + n_{ij} + \Delta n_{ij} > 0$. Consider four cases:

$$
\begin{cases}
\Delta n_{ij} = 0 \\
n_{ij} = 0, \Delta n_{ij} \neq 0 \\
n_{ij} \Delta n_{ij} > 0 \\
n_{ij} \Delta n_{ij} < 0
\end{cases}
$$

Suppose that $\Delta n_{ij} = 0$, then we obtain:

$$\begin{aligned}
&\frac{|n_{ij} + \Delta n_{ij}|}{|m_{ij}| + |m_{ij} + n_{ij} + \Delta n_{ij}|} - \frac{|n_{ij}|}{|m_{ij}| + |m_{ij} + n_{ij}|} \\
&= \frac{|n_{ij} + 0|}{|m_{ij}| + |m_{ij} + n_{ij} + 0|} - \frac{|n_{ij}|}{|m_{ij}| + |m_{ij} + n_{ij}|} \\
&= 0
\end{aligned} \tag{4.32c}$$

In a similar way, suppose that $n_{ij} = 0$, $\Delta n_{ij} \neq 0$, then we obtain:

$$\begin{aligned}
&\frac{|n_{ij} + \Delta n_{ij}|}{|m_{ij}| + |m_{ij} + n_{ij} + \Delta n_{ij}|} - \frac{|n_{ij}|}{|m_{ij}| + |m_{ij} + n_{ij}|} \\
&= \frac{|0 + \Delta n_{ij}|}{|m_{ij}| + |m_{ij} + 0 + \Delta n_{ij}|} - \frac{0}{|m_{ij}| + |m_{ij} + n_{ij}|} \\
&> 0
\end{aligned} \tag{4.32d}$$

Suppose $n_{ij}\Delta n_{ij} > 0$, according to the mediant inequality, $m_{ij}+n_{ij} > 0$

and $m_{ij} + n_{ij} + \Delta n_{ij} > 0$, we get:

$$\begin{aligned}
&\frac{|n_{ij} + \Delta n_{ij}|}{|m_{ij}| + |m_{ij} + n_{ij} + \Delta n_{ij}|} - \frac{|n_{ij}|}{|m_{ij}| + |m_{ij} + n_{ij}|} \\
&= \frac{|n_{ij}| + |\Delta n_{ij}|}{m_{ij} + m_{ij} + n_{ij} + \Delta n_{ij}} - \frac{|n_{ij}|}{m_{ij} + m_{ij} + n_{ij}} \\
&\geq \frac{|n_{ij}| + |\Delta n_{ij}|}{m_{ij} + m_{ij} + n_{ij} + |\Delta n_{ij}|} - \frac{|n_{ij}|}{m_{ij} + m_{ij} + n_{ij}} \\
&> 0
\end{aligned} \tag{4.32e}$$

However, the last case is extremely complicated. Suppose $n_{ij}\Delta n_{ij} < 0$, we have:

$$\begin{aligned}
&\frac{|n_{ij} + \Delta n_{ij}|}{|m_{ij}| + |m_{ij} + n_{ij} + \Delta n_{ij}|} - \frac{|n_{ij}|}{|m_{ij}| + |m_{ij} + n_{ij}|} \\
&= \frac{|n_{ij} + \Delta n_{ij}|}{2m_{ij} + n_{ij} + \Delta n_{ij}} - \frac{|n_{ij}|}{2m_{ij} + n_{ij}}
\end{aligned} \tag{4.32f}$$

It is seen that (4.32f) can be positive, negative or equal to zero. Therefore, there is no mathematical proof for the monotonicity of $g(\kappa)$. Nevertheless, according to (4.32d), (4.32e) and (4.32f), it is obvious that $g(\kappa)$ is an increasing function in most cases. In other words, let A, which is the increment of $g(\kappa)$, be a random variable defined on the probability space $(\Omega, \mathcal{F}, \mathcal{P})$, we have $P(A \geq 0) > P(A < 0)$. Thus, it

is essential to design a Monte Carlo experiment and demonstrate the data distribution for $\lim_{i,j\to\infty} \sum_{i,j}(g(\kappa_i) - g(\kappa_j))$, $\kappa_i > \kappa_j$. See Fig. 15. The X-axis refers to the n-th trial, and Y-axis is defined as the increment $\Delta g(\kappa)$. Like other heat maps, darker color refers to more points located in this area. It can be seen that $\Delta g(\kappa)$ is a direct ratio to $\Delta\kappa$. In the last graph, the positive area is insignificantly larger than the negative one. Besides, the upper limit in the last graph is almost 0.75, while the color of the negative parts is lighter than the color in the first graph. In a nutshell, it cannot be proved that $g(\kappa)$ is an increasing function mathematically on the one hand. On the other hand, the Monte Carlo experiment shows it is probably true.

iii. Jensen–Shannon divergence

The Fig. 16 shows $g(\kappa) = D_{Jensen-shannon}(M, M')$ is increasing as κ increases. Moreover, this graph is similar to Fig. 13. Based on the definition of Jensen–Shannon divergence, we have:

$$\begin{aligned} JSD(P||Q) &= \frac{1}{2}D(P||R) + \frac{1}{2}D(Q||R) \\ R &= \frac{1}{2}(P+Q) \end{aligned} \tag{4.33a}$$

where D refers to the Kullback–Leibler divergence. Since P, Q and

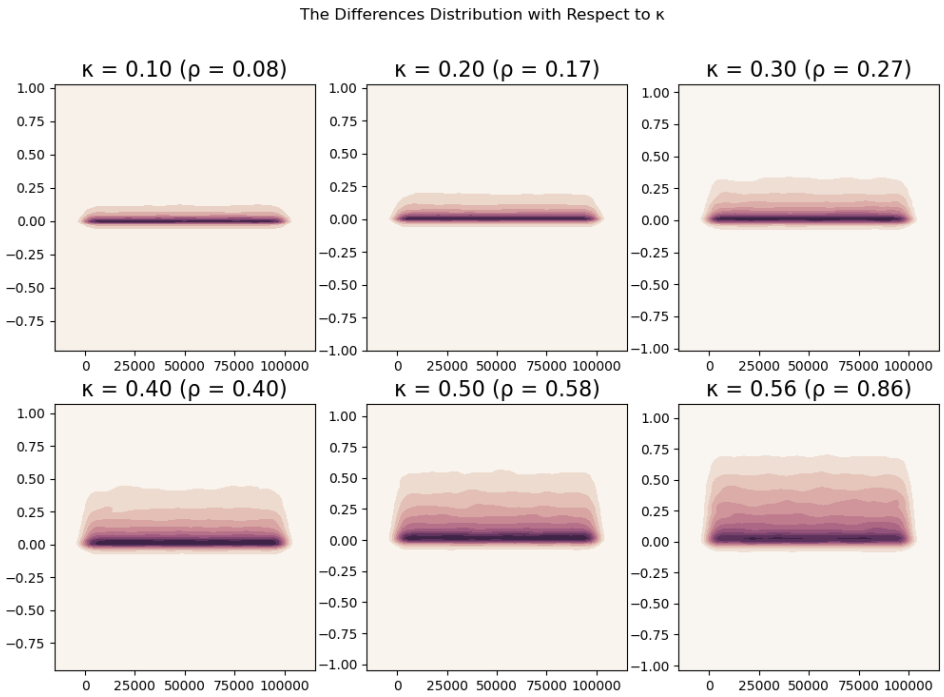


Figure 15: The Differences of $g(\kappa)$ with respect to κ. The ratios between κ and ρ are computed based on the matrix order = 3.

R are distributions for a continuous random variable defined on the probability space $(\Omega, \mathcal{F}, \mathcal{P})$, the KL divergence is defined as the integral:

$$D(P||R) = \int_{-\infty}^{\infty} p(x) log(\frac{p(x)}{r(x)})\, dx \tag{4.33b}$$

$$D(Q||R) = \int_{-\infty}^{\infty} q(x) log(\frac{q(x)}{r(x)})\, dx \tag{4.33c}$$

KL divergence also can be represented by the differences of the cross-

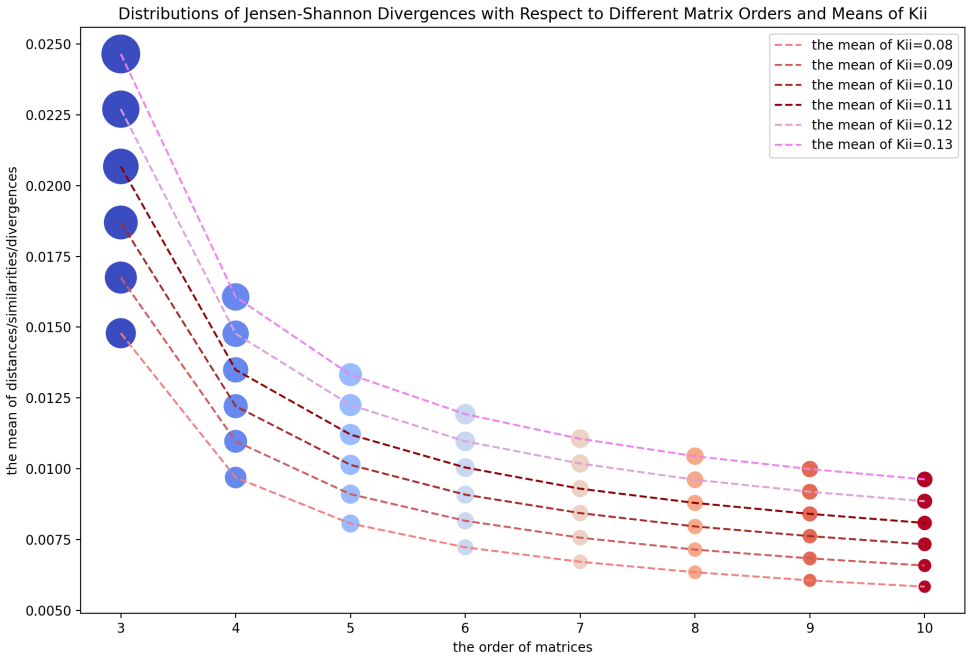


Figure 16: Distributions of Jensen–Shannon Divergences with respect to Different Matrix Orders and Means of Kii

entropy and the entropy:

$$D(P||R) = \int_{-\infty}^{\infty} p(x) log(\frac{1}{r(x)})\, dx - \int_{-\infty}^{\infty} p(x) log(\frac{1}{p(x)})\, dx$$
$$D(Q||R) = \int_{-\infty}^{\infty} q(x) log(\frac{1}{r(x)})\, dx - \int_{-\infty}^{\infty} q(x) log(\frac{1}{q(x)})\, dx$$

That is the reason why it is also called relative entropy [3]. In this study, the random numbers are generated from normal distributions. Then we denote these two distributions by $p(x) \sim \mathcal{N}(0, \sigma_p^2)$ and $q(x) \sim \mathcal{N}(0, \sigma_q^2)$.

From (4.33a), we have $R = \frac{1}{2}(P + Q)$. Thereby, $r(x)$ also follows a normal distribution $\mathcal{N}(0, \sigma_r^2)$ where $\sigma_r = \frac{\sqrt{\sigma_p^2+\sigma_q^2}}{2}$. In general, their probability density functions are written as:

$$\begin{aligned} p(x) &= \frac{1}{\sigma_p\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x}{\sigma_p})^2} \\ q(x) &= \frac{1}{\sigma_q\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x}{\sigma_q})^2} \\ r(x) &= \frac{1}{\frac{\sqrt{\sigma_p^2+\sigma_q^2}}{2}\sqrt{2\pi}} e^{-\frac{1}{2}\frac{2x^2}{\sigma_p^2+\sigma_q^2}} \end{aligned} \tag{4.33d}$$

Substituting (4.33d) into (4.33b), we obtain:

$$\begin{aligned} D(P||R) &= \int_{-\infty}^{\infty} p(x) log \frac{\frac{1}{\sqrt{2\pi}\sigma_p} e^{-\frac{1}{2}\frac{x^2}{\sigma_p^2}}}{\frac{1}{\sqrt{2\pi}\sigma_r} e^{-\frac{1}{2}\frac{x^2}{\sigma_r^2}}} \\ &= \int_{-\infty}^{\infty} p(x)(log\frac{\sigma_p}{\sigma_r} + log\frac{e^{-\frac{1}{2}\frac{x^2}{\sigma_p^2}}}{e^{-\frac{1}{2}\frac{x^2}{\sigma_r^2}}})dx \\ &= log\frac{\sigma_r}{\sigma_p}\int_{-\infty}^{\infty} p(x)dx + \int_{-\infty}^{\infty} p(x)(-\frac{x^2}{2\sigma_p^2} + \frac{x^2}{2\sigma_q^2})dx \\ &= log\frac{\sigma_r}{\sigma_p} - \frac{1}{2\sigma_p^2}\int_{-\infty}^{\infty} p(x)x^2dx + \frac{1}{2\sigma_r^2}\int_{-\infty}^{\infty} p(x)x^2dx \\ &= log\frac{\sigma_r}{\sigma_p} - \frac{\sigma_p^2}{2\sigma_p^2} + \frac{\sigma_p^2}{2\sigma_r^2} \\ &= log\frac{\sigma_r}{\sigma_p} + \frac{\sigma_p^2}{2\sigma_r^2} - \frac{1}{2} \end{aligned} \tag{4.33e}$$

Similarly, substituting (4.33d) into (4.33c), we have:

$$\begin{aligned} D(Q||R) &= \int_{-\infty}^{\infty} q(x) log \frac{\frac{1}{\sqrt{2\pi}\sigma_q} e^{-\frac{1}{2}\frac{x^2}{\sigma_q^2}}}{\frac{1}{\sqrt{2\pi}\sigma_r} e^{-\frac{1}{2}\frac{x^2}{\sigma_r^2}}} \\ &= log\frac{\sigma_r}{\sigma_q} + \frac{\sigma_q^2}{2\sigma_r^2} - \frac{1}{2} \end{aligned} \tag{4.33f}$$

Combine (4.33a), (4.33e) and (4.33f), we conclude that

$$\begin{aligned} JSD(P||Q) &= \frac{1}{2}D(P||R) + \frac{1}{2}D(Q||R) \\ &= \frac{1}{2}\left(log\frac{\sigma_r}{\sigma_p} + \frac{\sigma_p^2}{2\sigma_r^2} - \frac{1}{2} + log\frac{\sigma_r}{\sigma_q} + \frac{\sigma_q^2}{2\sigma_r^2} - \frac{1}{2} \right) \\ &= \frac{1}{2}(log\frac{\sqrt{\sigma_p^2+\sigma_q^2}}{2\sigma_p} + log\frac{\sqrt{\sigma_p^2+\sigma_q^2}}{2\sigma_q} + \frac{\sigma_p^2}{\sqrt{\sigma_p^2+\sigma_q^2}} \\ &\quad + \frac{\sigma_q^2}{\sqrt{\sigma_p^2+\sigma_q^2}} - 1) \\ &= \frac{1}{2}\left(log\frac{\sigma_p^2+\sigma_q^2}{4\sigma_p\sigma_q} + \sqrt{\sigma_p^2+\sigma_q^2} - 1 \right) \end{aligned} \tag{4.33g}$$

We can now proceed analogously to the proof of the monotonicity of $g(\kappa) = JSD(P||Q)$. When κ increases, ρ increases. So is σ_q. Besides, we have $\sigma_q > \sigma_p$ because $\sigma = \rho * m_{ij}$ and $m_{ij} > 0$. Under this circumstance, it is important to check the first-order partial derivative of $g(\kappa)$

by substituting (4.33g):

$$\begin{aligned}\frac{\partial g(\kappa)}{\partial \sigma_q} &= \frac{\partial g(\sigma_p, \sigma_q)}{\partial \sigma_q} \\ &= \frac{1}{2}\left(\frac{2\sigma_q}{\sigma_p^2 + \sigma_q^2} - \frac{1}{\sigma_q} + \frac{1}{2\sqrt{\sigma_p^2 + \sigma_q^2}}\right) \\ &= \frac{2\sigma_q^2 - 2\sigma_p^2 + \sigma_q\sqrt{\sigma_p^2 + \sigma_q^2}}{4\sigma_q(\sigma_p^2 + \sigma_q^2)} \\ &> 0\end{aligned}$$

So $g(\kappa)$ is an increasing function, which completes the proof.

4.4 Discussion

To summarize, Bray-Curtis distance, Canberra distance and Jensen-Shannon divergence are all reliable metrics. The distributions of Bray-Curtis distance and Jensen-Shannon divergence are similar, although there is a big gap between their theories. They both have a problem that the value range convergences so fast as the matrix order increases. However, they are much better than Cosine similarity or Kullback-Leibler divergence, which have higher convergence rates. Meanwhile, they have fewer outliers than the Chebyshev distance or Euclidean distance. Canberra distance, has a unique distribution graph and excellent properties to measure the matrices' distance. However,

its monotonicity cannot be proved mathematically. Besides, its distance range is not strictly $[0, 1]$. Nevertheless, it is still a good indicator. It has excellent distinguishability with different matrix orders. Simultaneously, the Monte Carlo experiment shows it is worth considering. Therefore, all of them will be used to optimize the objective function and reconstruct the matrices in the next section.

5 Reconstruct the Matrices with Differential Evolution

5.1 Weight Coefficient

According to the previous sections, for a l by l NSI PC matrix $M = [m_{pq}]\, p, q \in \{1, 2, \cdots, l\}$, the distance-based objective function is defined as:

$$f(M, M') = Kii(M) + \alpha D(M, M') \tag{5.1}$$

where $\alpha \in [0, \infty]$. According to 2.1, 4.31c, 4.32b and 4.33g, we obtain:

$$f_{BC}(M, M') = 1 - \min_{i<j<k} \left(\frac{m_{ik}}{m_{ij}m_{jk}}, \frac{m_{ij}m_{jk}}{m_{ik}} \right) + \alpha \frac{\sum_{ij} |n_{ij}|}{\sum_{ij} |2m_{ij} + n_{ij}|}$$

$$f_{CA}(M, M') = 1 - \min_{i<j<k} \left(\frac{m_{ik}}{m_{ij}m_{jk}}, \frac{m_{ij}m_{jk}}{m_{ik}} \right) + \alpha \sum_{ij} \frac{|n_{ij}|}{|m_{ij}| + |m_{ij} + n_{ij}|}$$

$$f_{JS}(M, M') = 1 - \min_{i<j<k} \left(\frac{m_{ik}}{m_{ij}m_{jk}}, \frac{m_{ij}m_{jk}}{m_{ik}} \right) + \frac{\alpha}{2} \left(log \frac{\sigma_p^2 + \sigma_q^2}{4\sigma_p\sigma_q} + \sqrt{\sigma_p^2 + \sigma_q^2} - 1 \right)$$

subject to constrains $m_{pq} > 0$. Of note, none of these functions is continuous or differentiable. Thus, the stochastic gradient descent algorithm, which is a well-known and effective optimization algorithm, or other first-order algorithms can not be used to optimize these functions. Nevertheless, there are many problem-independent algorithms for these optimization problems, for example, the pattern search method or heuristic algorithms [10]. In these algorithms, differential evolution algorithm is a popular derivative-free heuristic algorithm. It can be used to optimize non-differentiable, discontinuous or noisy objective functions by searching wide spaces of candidate solutions. Thereby, DE is used here to optimize above the objective functions.

The next concern is the value of α. The role of this parameter is to balance the weights between two parts of the objective function f. In other words, the inconsistent indicator Kii and the metric between two matrices $D(M, M')$

have the same weight exactly if α equals a threshold $\hat{\alpha}$. Also, $\alpha > \hat{\alpha}$ or $\alpha < \hat{\alpha}$ means one of them is more important than the other one. Fig. 6 illustrates that the mean of the Bray-Curtis distances is different although there are no big gaps between these means. Similarly, Fig. 7 demonstrates that the means of Jensen-Shannon divergences are also different and converge to 0.01 as order increases. Meanwhile, the differences between Canberra distances are even higher when comparing the other two metrics. See Fig. 11. Thus, the same α cannot be applied for metrics with different matrix orders. For each combination of order and metric, there is a unique α:

$$\alpha = \frac{\text{the mean of } Kii}{\text{the mean of metrics}}$$

where the mean of Kii is set as 0.1 in this section. See Table. 5. Here is an example. The threshold of α is 4.4459 when the order is three and the metric is Bray-Curtis distance in this table. So the objective function is defined as:

$$f_{BC}(M, M') = 1 - \min_{i<j<k} \left(\frac{m_{ik}}{m_{ij}m_{jk}}, \frac{m_{ij}m_{jk}}{m_{ik}} \right) + 4.4459 * \frac{\sum_{ij} |n_{ij}|}{\sum_{ij} |2m_{ij} + n_{ij}|}$$

when three by three matrices are optimized based on Bray-Curtis distance.

Table 5: The Threshold of α with respect to the Matrix Order and Metric

Order	Bray-Curtis Distance	Canberra Distance	Jensen-Shannon Divergence
3	4.4459	0.5499	5.3696
4	6.7535	0.4745	8.1333
5	8.7276	0.3723	9.9943
6	9.3913	0.2801	11.0695
7	10.285	0.2229	12.0492
8	10.9508	0.1804	12.6775
9	11.592	0.1487	13.3177
10	12.0805	0.125	13.8052

5.2 Analysis of the Results

100,000 NSI PC matrices have been generated and used to compare the distributions of metrics in the last section. Also, 10,000 matrices are chosen to be included in the Monte Carlo experiments in this section because the DE program is time-consuming. After optimizing and analyzing these NSI PC matrices, the results show that the Kii of all optimized matrices are zero,

which is as expected. There are numerous solutions for $Kii = 0$. There is no doubt that the DE algorithm can find them. Therefore, the critical point is the performance of these algorithms in metrics. In what follows, M' denotes the NSI PC matrix generated from the original PC matrix M^*, and M stands for the new PC matrix, which is optimized by the DE algorithm.

i. Bray–Curtis distance

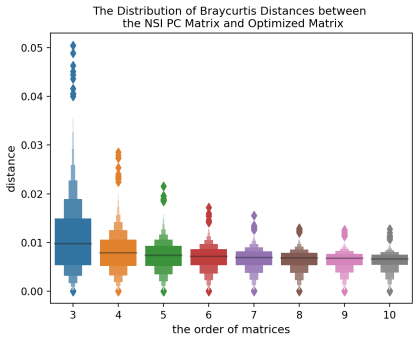
Fig. 17 illustrates the distribution of distances about the optimized matrix M. The X and Y axes are kept the same for these two subplots as well as the axes in Fig. 6. In other words, the X-axis denotes the orders of matrices, and the Y-axis refers to the distances between different matrices.

For Fig. 17(a), the distances are computed between the optimized matrix M and the NSI PC matrix M'. Compared with Fig. 6, which shows the distances between the NSI PC matrix M' and the original PC matrix M^*. It is unambiguous the new matrix M is much closer to M' than M^* which meets our goal. The most significant outlier is three times smaller now. Also, the median values of new distances are much closer to zero.

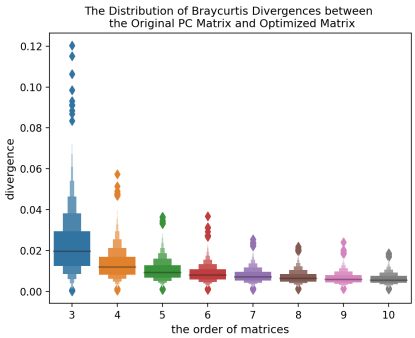
Fig. 17(b) demonstrates the distances between M and M^*. The corre-

sponding experiment is apparently a control group. There is no further information about the original matrix M^* in real-life optimization problems. Thus, the distribution of this distance between M^* and M is hard to predict since they are almost two random matrices. However, they still have one thing in common: they can converge to the same NSI PC matrix M' in some way. This subplot here is considerably similar to Fig. 6. The data shows similarity with the exception of fewer outliers in Fig. 17(b).

Several statistical measurements (mean, stand deviation, minimum and maximum value) of the experiment results have been detailed in Table. 6, in this way that they can be compared visibly. As mentioned before, all statistical measurements for distances between M' and M are smaller than these indicators for metrics between M' and M^* except the standard deviation (SD, for short). Besides, the most exciting result here is the minimum values of distances between M' and M are almost zero for all matrix orders (If the value is zero in this table, it means the exact value is lower than 0.00005). It implies that the optimization algorithm works well and obtains the significantly "close" PC matrices corresponding to some NSI PC matrices.



(a) Distances between M' and M



(b) Distances between M^* and M

Figure 17: The New Distribution of Bray-Curtis Distance

ii. Canberra distance

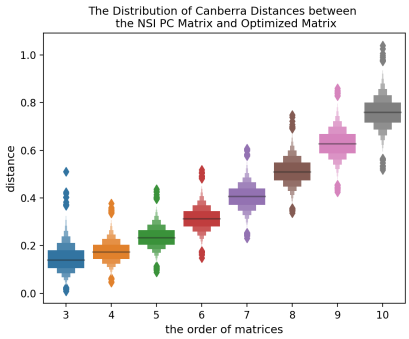
Canberra distance, which has unique properties for measuring matrix distance, is still differently distributed here. Compared with Fig. 17, Fig. 18 comes to a contrary conclusion. Fig. 18(a) and Fig. 7 are almost identical. Nevertheless, Table. 7 illustrates that all values, even the standard deviation, have decreased slightly, which proves that the DE algorithm performs a function in optimizing the Canberra distances.

For another subplot Fig. 18(b), it is interesting that all indicators are decreased dramatically. For example, the median distance is lower than 0.33 when the matrix order is 10. Meanwhile the value for distances between M' and M^* is 0.8198. It looks like the algorithm optimize the distances between M and M^* instead of the distances between M and

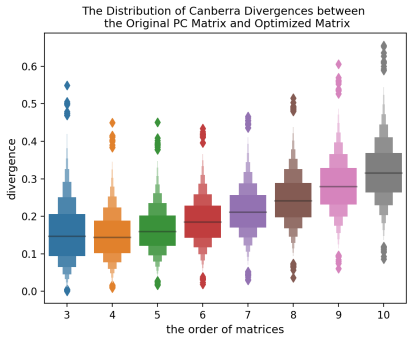
Table 6: Statistical Measurements for Several Bray-Curtis Distances

Bray-Curtis Distance		Matrix Order							
		3	4	5	6	7	8	9	10
Distances between M' and M^*	Mean	0.0236	0.015	0.0122	0.0106	0.01	0.0095	0.009	0.0087
	SD	0.0108	0.005	0.0033	0.0026	0.0021	0.0019	0.0016	0.0014
	Min	0.0028	0.0047	0.0035	0.0035	0.004	0.0027	0.0046	0.0047
	Max	0.0949	0.0405	0.027	0.0222	0.0216	0.0204	0.0174	0.0161
Distances between M' and M	Mean	0.0106	0.0079	0.0072	0.0068	0.0065	0.0063	0.0062	0.0062
	SD	0.007	0.0041	0.003	0.0027	0.0023	0.0022	0.002	0.0019
	Min	0.0	0.0	0.0	0.0001	0.0	0.0001	0.0001	0.0001
	Max	0.0393	0.0246	0.0188	0.0149	0.0134	0.0126	0.0103	0.0121
Distances between M and M^*	Mean	0.0224	0.0133	0.0101	0.0082	0.0078	0.0072	0.0066	0.0061
	SD	0.014	0.0068	0.0049	0.0037	0.0034	0.0032	0.0029	0.0026
	Min	0.0006	0.0012	0.0012	0.001	0.0015	0.002	0.0017	0.0017
	Max	0.1319	0.0454	0.0308	0.0279	0.0228	0.0236	0.0195	0.0181

M'. It is unknown why Canberra distance has this property, but this property can be widely used in real-life problems and help approach the latent original matrix.



(a) Distances between M' and M



(b) Distances between M^* and M

Figure 18: The New Distribution of Canberra Distance

iii. Jensen-Shannon divergence

The analysis results for Jensen-Shannon divergence shows both of the two divergences are dropped rapidly. All medians except the one for order=3 are lower than 0.1. Also, there are fewer outliers in Fig. 19. Table. 8 provides more details on the comparisons between these two divergences. It shows that the divergences between M and M^* are lower than those between M and M', which are the same as those in Table. 7. Hence, the abnormal property of Canberra Distance is not an isolated case. Similarly, it can also obtained the latent origin matrix by optimizing the objective function based on Jensen-Shannon divergence.

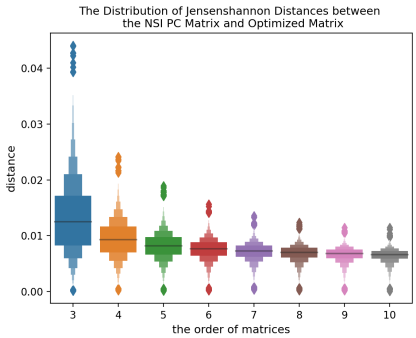
Table 7: Statistical Measurements for Several Canberra Distances

Canberra Distance		Matrix Order							
		3	4	5	6	7	8	9	10
Distances between M' and M^*	Mean	0.1838	0.2134	0.2783	0.3578	0.4538	0.565	0.6841	0.8198
	SD	0.0589	0.0474	0.0475	0.0512	0.0541	0.058	0.062	0.0676
	Min	0.0466	0.0898	0.1492	0.2195	0.2896	0.3993	0.4956	0.5869
	Max	0.4784	0.4007	0.4277	0.5574	0.6224	0.7224	0.8785	1.0693
Distances between M' and M	Mean	0.1434	0.1739	0.2373	0.3143	0.4037	0.5128	0.6263	0.7548
	SD	0.0535	0.0455	0.0474	0.05	0.0524	0.0562	0.0597	0.0661
	Min	0.0326	0.0464	0.1032	0.1789	0.2531	0.3366	0.4719	0.5535
	Max	0.3664	0.3318	0.399	0.4873	0.5785	0.6822	0.8482	1.0171
Distances between M and M^*	Mean	0.1539	0.1463	0.163	0.184	0.214	0.2436	0.279	0.3244
	SD	0.0819	0.0606	0.0591	0.0597	0.0643	0.0668	0.0713	0.079
	Min	0.0049	0.0129	0.0191	0.033	0.0618	0.0852	0.087	0.1175
	Max	0.4485	0.344	0.3612	0.4066	0.4561	0.4765	0.5354	0.6277

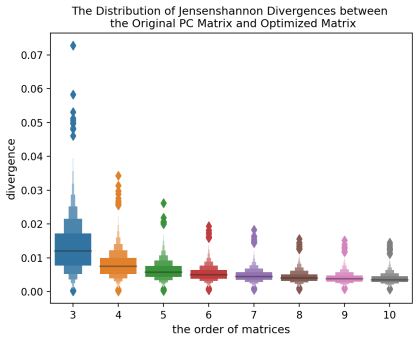
5.3 Discussion

All results based on different metrics have been analyzed and compared yet.

The algorithm based on Bray-Curtis distances cannot minimize the distances



(a) Distances between M' and M



(b) Distances between M^* and M

Figure 19: The New Distribution of Jensen-Shannon Distance

between M and M^* by optimizing the distances between M and M'. For Canberra distance, things are the opposite. The algorithm is designed to minimize the distance between M and M'. However, the result shows that it only optimizes this metric slightly and minimizes another metric inadvertently. Besides, the range of this metric is not strictly $[0, 1]$. Also, there is no mathematical proof for its monotonicity. The algorithm based on Jensen-Shannon Divergence performances much better than the other two metrics. The figure and table illustrate it can minimize two divergences simultaneously, although only one of them is the target. As mentioned above, the range of Jensen-Shannon divergence is $[0, 1]$, and it is a monotonically increasing function with respect to the mean of Kii. In a nutshell, it is sufficient to say

Table 8: Statistical Measurements for Several Jensen-Shannon Divergences

Jensen-Shannon Divergence		Matrix Order							
		3	4	5	6	7	8	9	10
Divergences between M' and M^*	Mean	0.0191	0.0128	0.0105	0.0093	0.0087	0.0083	0.0079	0.0076
	SD	0.008	0.0038	0.0025	0.002	0.0016	0.0015	0.0012	0.0011
	Min	0.0013	0.0036	0.0019	0.003	0.0035	0.0027	0.0036	0.0041
	Max	0.0684	0.0325	0.0216	0.0183	0.0154	0.0181	0.0127	0.0132
Divergences between M' and M	Mean	0.0129	0.0094	0.0082	0.0074	0.0071	0.0068	0.0066	0.0064
	SD	0.0067	0.0036	0.0025	0.002	0.0017	0.0015	0.0014	0.0012
	Min	0.0002	0.0003	0.0002	0.0012	0.0006	0.0008	0.0007	0.001
	Max	0.0449	0.0213	0.017	0.0136	0.012	0.0111	0.0103	0.0105
Divergences between M and M^*	Mean	0.0128	0.008	0.0061	0.0051	0.0048	0.0044	0.0041	0.0038
	SD	0.0072	0.0037	0.0026	0.0021	0.0018	0.0018	0.0015	0.0015
	Min	0.0005	0.0009	0.0002	0.0012	0.0011	0.0012	0.0009	0.0013
	Max	0.0532	0.0284	0.017	0.0157	0.0151	0.0155	0.0115	0.0111

that:

$$f_{JS}(M, M') = \max_{i<j<k} \min \left(\left| 1 - \frac{m_{ik}}{m_{ij}m_{jk}} \right|, \left| 1 - \frac{m_{ij}m_{jk}}{m_{ik}} \right| \right) + \frac{\alpha}{2} \left(log \frac{\sigma_p^2 + \sigma_q^2}{4\sigma_p\sigma_q} + \sqrt{\sigma_p^2 + \sigma_q^2} - 1 \right)$$

is the best-suited objective function for the differential evolution algorithm, which is applied to find the closest PC matrix.

6 Conclusion and Future Work

6.1 Conclusion

In this paper, an optimization method was proposed in accordance with differential evolution and matrix metrics to reconstruct pairwise comparisons matrices. The issue which reconstructs a PC matrix from an NSI PC matrix is introduced and defined in the first and second section. The previous method to solve this problem is the distance-based inconsistency reduction algorithm. It is a simple and straightforward design proposed by Koczkodaj in 2015 [12]. The basic concepts of pairwise comparisons are also presented in the second section. PC matrix, NSI PC matrix, consistency and Kii are all the bases of this unsolved problem. In section 3, the origin and history of the differential evolution algorithm are introduced at the beginning. The processes and details of this algorithm are also presented in this section.

There are three subsections for Problem Formulation. A new method to generate random NIS PC matrices is proposed in the first subsection and compared with several other methods. The correlation between the parameters of this new method and Kii has been investigated. The examined result shows there is an approximate linear or quadratic correlation when the Kii

is lower than a threshold. In the second subsection, several common metrics are proposed and compared based on a Monte Carlo experiment, which is designed to illustrate the distributions of these metrics with the same Kii and different matrix orders. The results demonstrate that Bray-Curtis distance, Canberra distance and Jensen-Shannon divergence have suitable properties to measure the matrix distances. In the last subsection, mathematical proofs are given to ensure the monotonicity of the metric function including Bray-Curtis distance and Jensen-Shannon divergence when the number or matrices or the order of a large matrix tends to infinity. For Canberra distance, there is no mathematical proof. However, a figure based on a Monte Carlo experiment demonstrates that the differences between two matrices tend to be bigger than zero.

In the first subsection of section 5, the value of the weight coefficient α is discussed. Also, a table of the thresholds of α is proposed, and these values are applied in the subsequent experiments. The differential evolution algorithm is used for optimization in the second section. The analysis of the optimized matrices shows that the algorithm, which its objective function is based on Jensen-Shannon divergence, is steady and has good performance in comparison to these algorithms based on other metrics.

6.2 Future Work

The following further research is proposed based on previous results. Firstly, all the research in this paper is based on inconsistency indicators. Koczkodaj inconsistency index is used here as the standard of inconsistency because it is straightforward and easy to compute. Besides, the range of Kii is $[0, 1]$, which is a suitable property for an objective function. However, there are some other indicators to measure the inconsistency of a PC matrix, like the GW index in 1989 [6], relative error in 1998 [2], Geometric Consistency Index in 2003 [1], Harmonic Consistency Index in 2007 [19] or K-Index in 2020 [21]. Thus, extending the work to those indicators is reasonably straightforward. Secondly, only seven common metrics are compared in section 4. There are also a lot of other metrics as well as inconsistency indicators. With that said, there might be other metrics which have higher performance than Jensen-Shannon divergence since this divergence is an unexpected optimal solution. Finally, it is worth noting that Canberra distance and Jensen-Shannon divergence have some anomalous properties. The DE algorithm based on these metrics can minimize two distances or divergences at the same time, which is designed to optimize only one of them. What's more, the optimization algorithm does not receive any information for another distance

or divergence. One possible reason is that there are some latent connections between the NSI PC matrix and its original PC matrix. More experiments will be designed to reveal these connections in the future.

Appendix

A The Core Part of the Python Program

```
# coding:utf8

"""
@author: Zhangao Lu
@contact: zlu2@laurentian.ca
@time: 2021/2/24
@description:
1. Generate NSI PC matrices, save and test them.
2. Fit the curve which is used to display the relations between rho
and mean of Kii.
"""

import copy
import numpy as np
import matplotlib.pyplot as plt
import scipy
from itertools import combinations
from scipy.optimize import curve_fit
from collections import OrderedDict
from config import config
from utils.pairwise_comparison_tools import compute_kii
from utils.gerenal_tools import open_pickle, save_pickle, save_hickle, open_hickle


class GenerateMatrices(object):
    def __init__(self, order=3, iterations=1000, std_rate=1):
        """

        :param order: int, default = 3
                The order of generated matrices.
```

```
31          :param iterations: int, default = 1000
32                 The number of generated matrices.
33          :param std_rate: float, default = 1
34                 It is the parameter \rho in the thesis.
35                 sigma = self.std_rate * origin_num.
36          """
37          self.order = order
38          self.iterations = iterations
39          self.std_rate = std_rate
40          # A ordered dict to save the results
41          self.result = OrderedDict()
42          # Save the generated matrices with file name
               below.
43          # The Kii threshold will always be 0.1,
44          # which is determined in the thesis.
45          self.file_name_of_pc = "%d pc matrices with
               order=%d kii_threshold=%0.1f.pkl" % \
46                                 (self.iterations, self.
                                    order, 0.1)  # for
                                    PC matrices
47          self.file_name_of_nsi_pc = "%d nsi pc matrices
               with order=%d kii_threshold=%0.1f.pkl" % \
48                                     (self.iterations,
                                        self.order, 0.1)
                                          # for NSI PC
                                        matrices
49
50      @staticmethod
51      def random_numbers(sigma, origin_num, mu=0):
52          """
53          A static method used to generate errors for
               the elements of the original PC matrices.
54          Errors followed normal distribution with mean
               = mu, standard deviation = sigma.
55          And make sure origin_num + error > 0
56          :param sigma: float
57                 Standard deviation of the normal
                    distribution
58          :param origin_num: float
```

```
59                The elements in the PC matrix. The
                     value must be greater than 0.
60          :param mu: float, default = 0
61                Mean of the normal distribution. The
                     value is zero and will not be
                     changed during this experiment.
62          :return: error, float
63                   A float number which refers to the
                        random error of the PC matrices'
                        elements.
64          """
65          while 1:
66              # numpy.random.randn() can return a sample
                    from the standard normal distribution.
67              # So for random samples from N(\mu, \sigma
                    ^2), they are sigma * np.random.randn()
                    + mu.
68              error = sigma * np.random.randn() + mu
69              # The error must make sure the sum of the
                    error and original element is greater
                    than zero.
70              # If the error meets the requirement, then
                    break.
71              if origin_num + error > 0:
72                  break
73          return error
74
75      def generate_matrix(self):
76          """
77          Generate PC matrices and NSI PC matrices.
78          :return: dict
79                  {"NSI_PC": nsi_pc, "PC": pc}
80          """
81          # numpy.random.rand(n) can generate a random
                array with shape (n, 1).
82          # However, the elements of this array can be
                zero. So if it happens, the array should be
                 discarded.
83          while 1:
```

```
84              vector = np.random.rand(self.order)  # The
                    range of the samples is [0, 1).
85              if 0 not in vector:  # If 0 in the array,
                   repeat the process. Otherwise,
                   terminate the loop.
86                  break
87          # np.eye() can return a 2-D array with ones on
                the diagonal and zeros elsewhere.
88          pc = np.eye(self.order)
89          # Use copy.deepcopy here to create another
               matrix.
90          nsi_pc = copy.deepcopy(pc)
91          """
92          Permutations and combinations are itertools
               functions, which are designed to return
               successive elements
93          in the iterable.
94          permutations(range(0, 2), 2) ==> (0, 1), (0,
               2), (1, 0), (1, 2), (2, 0), (2, 1).
95          combinations(range(0, 2), 2) ==> (0, 1), (0,
               2), (1, 2).
96          """
97          temp = combinations(range(0, self.order), 2)
98          """
99          Generate the PC matrix according the array
               vector. The element of the PC matrix a_{ij}
                is equal to
100         vector_i / vector_j. Because I use
               combinations here, the iterable only has
               half of the needed elements.
101         So two elements of the PC matrices, a_{ij} and
                a_{ji}, must be generated in one loop.
102         The O(n) for permutations is n(n-1).
103         The O(n) for combinations is n(n-1)/2.
104         """
105         for elm in temp:
106             i = elm[0]
107             j = elm[1]
108             tmp1 = vector[i] / vector[j]
109             tmp2 = vector[j] / vector[i]
```

```
110             pc[i, j] = tmp1  # The PC matrix's element
                    : a_{ij} = vector_i / vector_j.
111             pc[j, i] = tmp2  # The PC matrix's element
                    : a_{ji} = vector_j / vector_i.
112             # The NSI PC matrix's element: b_{ij} =
                    vector_i / vector_j + error.
113             nsi_pc[i, j] = tmp1 + self.random_numbers(
                    self.std_rate * tmp1, tmp1) if self.
                    std_rate else tmp1
114             # The NSI PC matrix's element: b_{ji} =
                    vector_j / vector_i + error.
115             nsi_pc[j, i] = tmp2 + self.random_numbers(
                    self.std_rate * tmp2, tmp2) if self.
                    std_rate else tmp2
116         return {"NSI_PC": nsi_pc, "PC": pc}
117
118     def generate_with_rho(self, start=0, end=21, step
            =1):
119         """
120         Generate NSI PC matrices with different
                std_rate, then compute and compare the
                matrices' Kii.
121         It is used to draw the graph in the thesis.
122         :param start: int, default = 0
123         :param end: int, default = 21
124         :param step: int, default = 1
125         :return: None
126         """
127         # np.array(range(0, 21, 1)) / 100.0 will
                create a numpy.array: [0, 0.01, 0.02, ...,
                0.2]
128         for self.std_rate in np.array(range(start, end
                , step)) / 100.0:
129             kii_list = list()  # Store the values of
                    Kii temporarily.
130             for _ in range(self.iterations):
131                 m = self.generate_matrix()  # Generate
                        a PC and NSI PC matrix.
132                 kii_list.append(compute_kii(m["NSI_PC
                        "]))  # Choose the NSI one and
```

```
                    compute its Kii.
133             mean_of_kii = float(np.mean(kii_list))  #
                    Compute the mean of all values of Kii.
134             self.result[self.std_rate] = mean_of_kii
                    # Then save the mean into the ordered
                    dict: self.result.
135
136     def generate_and_save(self):
137         """
138         Generate NSI PC matrices and save them with
                pickle for further research.
139         Generally, it is designed to generate matrices
                with Kii = threshold by setting different
                values
140         for self.std_rate.
141         In default, self.std_rate = 1
142         :return: None
143         """
144         pc_list = list()  # To save PC matrices
                temporarily.
145         nsi_pc_list = list()  # To save NSI PC
                matrices temporarily.
146         for _ in range(self.iterations):
147             m = self.generate_matrix()
148             """
149             np.expand_dims(arr, axis=2): add a new
                    dimension for a two dimension np.array
                    arr, then the np.array list
150             can be merged in to a big 3d array in next
                    steps.
151             The shape of arr is changed from shape (n,
                    n) to shape (n, n, 1).
152             """
153             pc_list.append(np.expand_dims(m["PC"],
                    axis=2))
154             nsi_pc_list.append(np.expand_dims(m["
                    NSI_PC"], axis=2))
155         """
156         Concatenate a list of arrays, which the shape
                is (n, n, 1), into one big array,
```

```
157         and its shape is (n, n, self.iterations). Then
                the big array can be saved by a faster
            tool: hickle,
158         """
159         pc_array = np.concatenate(tuple(pc_list), axis
                =2)
160         nsi_pc_array = np.concatenate(tuple(
                nsi_pc_list), axis=2)
161         # If the file is a numpy array, I can use
                hickle to accelerate and save spaces and
                time.
162         save_hickle(pc_array, config.path_for_thesis +
                 self.file_name_of_pc)
163         save_hickle(nsi_pc_array, config.
                path_for_thesis + self.file_name_of_nsi_pc)
164
165     def read_array(self):
166         """
167         Read hickle files and decompose into array
                list.
168         :return: pc_list, list
169                  nsi_pc_list, list
170                  The lists of PC matrices and NSI PC
                     matrices.
171         """
172         self.file_name_of_pc = "%d pc matrices with
                order=%d kii_threshold=%0.1f.pkl" %\
173                                (self.iterations, self.
                                   order, 0.1)
174         self.file_name_of_nsi_pc = "%d nsi pc matrices
                 with order=%d kii_threshold=%0.1f.pkl" %\
175                                    (self.iterations,
                                       self.order, 0.1)
176         pc_array = open_hickle(config.path_for_thesis
                + self.file_name_of_pc)
177         nsi_pc_array = open_hickle(config.
                path_for_thesis + self.file_name_of_nsi_pc)
178         pc_list = np.split(pc_array, pc_array.shape
                [2], axis=2)  # (n, n, iterations) -> [(n,
                n, 1)], len()=1000
```

```
179         nsi_pc_list = np.split(nsi_pc_array,
                nsi_pc_array.shape[2], axis=2)
180         return pc_list, nsi_pc_list
181
182     def print_results(self):
183         """
184         Print all values of mean of Kii for further
                research.
185         :return:
186         """
187         # self.result is a ordered dict to save
                different values for mean of kii. Their
                keys are self.std_rate.
188         for key in self.result:
189             mean_of_kii = self.result[key]
190             if key == 0:
191                 print("the mean of %d %d by %d PC
                        matrices' Kii is %f" %
192                     (self.iterations, self.order,
                            self.order, mean_of_kii))
193             else:
194                 print("standard deviation == %0.2f *
                        m_{ij}, the mean of %d %d X %d NSI
                        PC matrices' Kii is %f" %
195                     (key, self.iterations, self.
                            order, self.order,
                            mean_of_kii))
196
197
198 def draw_means_graph(is_fit=True, is_save=False,
        is_generate=False, iterations=100000):
199     """
200     Draw graphs to display the relations between mean
            of Kii and the ratio \rho (or self.std_rate) of
201     the standard deviation.
202     :param is_fit: boolean
203             If True, fit the curve. Otherwise, draw the
                    original curve.
204     :param is_save: boolean
```

```
205             If True, save the graph. Otherwise, show
                    the graph.
206     :param is_generate: boolean
207             If True, generate these matrices and save.
                    Otherwise, access from the hard drive.
208     :param iterations: int, default = 1000
209             The number of generated matrices.
210     :return:
211     """
212     rho_collection = dict()
213     # To display the graph better, set the figsize to
            (12, 8).
214     fig = plt.figure(figsize=(12.00, 8.00))
215     # fig = plt.figure(figsize=(19.20, 10.80))
216     cnt = 0  # Counter of the loop, used to choose
            different colors.
217     for order in range(3, 11, 1):  # Matrix order [3,
            11]
218         if is_generate:  # Run once.
219             gm = GenerateMatrices(order=order,
                        iterations=iterations)
220             gm.generate_with_rho(end=15)  # Change end
                        from 21 to 15 for better display.
221             gm.print_results()
222             # Save these matrices to hard drive.
223             save_pickle(gm.result, config.
                        path_for_thesis + "mean_of_kii_order=%d
                        .pkl" % order)
224             res = gm.result
225         else:
226             # Access the matrices from hard drive.
227             res = open_pickle(config.path_for_thesis +
                        "mean_of_kii_order=%d.pkl" % order)
228
229         if is_fit:  # If fit, draw the original curve
                and the fit curve.
230             x = list(res.keys())[: 15]  # Select some
                        std_rate/rho to analyze.
231             y = list(res.values())[: 15]  # Select
                        some mean of Kii to analyze.
```

```
232             # Draw the original curve at first.
233             plt.plot(x, y, color=config.color_list[cnt
                    ],
234                      marker="o", linestyle="-", label
                            ="order=%d, original curve " %
                             order)
235             popt, pcov = compute_curve(x, y)  # popt =
                    (x, y) = (std_rate/rho, mean of Kii)
236             rho_collection[order] = list()  #
                    rho_collection = {order: []}
237             for threshold in np.array(range(5, 16, 1))
                    / 100.0:  # threshold = [0.05, 0.06,
                    ..., 0.15]
238                 # scipy.optimize.fsolve is used to
                        find the roots (\rho) of the lambda
                         function:
239                 # a * \rho ^ 2 + b \rho - threshold =
                        0
240                 rho = scipy.optimize.fsolve(lambda k:
                        popt[0] * (k ** 2) + popt[1] * k -
                        threshold, np.array([0]))[0]
241                 # Print the results and round to four
                        decimal places.
242                 print("order:", order, round(popt[0],
                        config.decimal_places), round(popt
                        [1], config.decimal_places),
243                            round(rho, config.decimal_places
                                ), popt[0] * rho ** 2 + popt
                                [1] * rho)
244                 # rho_collection = {order: [{"rho":, "
                        a":, "b":, "threshold":}, {}]}
245                 rho_collection[order].append({"rho":
                        rho, "a": popt[0], "b": popt[1], "
                        threshold": threshold})
246             # Draw the fit curve. *popt = popt[0],
                    potp[1].
247             plt.plot(x, fit_function(np.array(x), *
                    popt), color=config.color_list[cnt],
                    linestyle="--",
```

```
248                         label="order=%d, fit curve" %
                               order)
249             # Draw a line for the threshold, 0.1.
250             plt.axhline(y=config.threshold, color='
                   grey', linestyle='--')
251         else:  # If not fit, draw the original curve.
252             x = list(res.keys())
253             y = list(res.values())
254             # For this case, only draw the original
                   curve.
255             plt.plot(x, y, color=config.color_list[cnt
                   ], marker="o", linestyle="-", label="
                   order=%d" % order)
256         cnt += 1
257
258     # Set different titles
259     if is_fit:
260         title = 'The Original and Fit Curves'
261     else:
262         title = 'The Mean of %d NSI PC Matrices\' Kii'
                % iterations
263     plt.title(title, config.ft)
264     plt.xlabel('the ratio %s of the standard deviation
        ' % chr(961), config.ft)  # The label of X-axis
        .
265     plt.ylabel('mean of Kii', config.ft)  # The label
       of Y-axis.
266     plt.legend()  # Display the legend of the graph.
267     if is_save:
268         if is_fit:
269             # Eps file for latex. Pdf file for
                   checking.
270             fig.savefig(config.path_for_thesis + "
                   the_original_and_fit_curve.pdf",
271                         format="pdf", dpi=1200)
272             fig.savefig(config.path_for_thesis + "
                   the_original_and_fit_curve.eps",
273                         format="eps", dpi=1200)
274         else:
```

```
275             fig.savefig(config.path_for_thesis + "
                    the_mean_of_NSI_PC_matrices_Kii.pdf",
276                         format="pdf", dpi=1200)
277             fig.savefig(config.path_for_thesis + "
                    the_mean_of_NSI_PC_matrices_Kii.eps",
278                         format="eps", dpi=1200)
279     else:
280         plt.show()  # If not is_save, show the graph.
281     # Print specific sentences, which is designed for
            writing the table in Latex.
282     if rho_collection:
283         # print(rho_collection)
284         for key in rho_collection:  # {order: [{"rho
                ":, "a":, "b":, "threshold":}, {}]}
285             for elm in rho_collection[key]:
286                 if elm["threshold"] == 0.1:
287                     print("0.1 & %d & %0.4f & %0.4f &
                            %0.4f \\\\" % (key, elm["rho"],
                             elm["a"], elm["b"]))
288         save_pickle(rho_collection, config.
                path_for_thesis + "rho.pkl")
289
290
291 def fit_function(x, a, b):
292     """
293     y = a * x ^ 2 + b, the function used to fit the
            curve.
294     :param x: float
295             \rho/std_rate
296     :param a: float
297             coefficient
298     :param b: float
299             coefficient
300     :return: y float
301              y is the mean of Kii
302     """
303     y = a * x ** 2 + b * x
304     return y
305
306
```

```
307 def compute_curve(x, y):
308     """
309     There are some relations between x (std_rate or \
            rho in the thesis) and y (mean of Kii).
310     So fit a curve function for it.
311     :param x: std_rate, list
312     :param y: mean of Kii, list
313     :return: popt float
314              coefficient a
315              pcov float
316              coefficient b
317     """
318     popt, pcov = curve_fit(fit_function, x, y)
319     return popt, pcov
320
321
322 if __name__ == '__main__':
323     # gm = GenerateMatrices(order=3)
324     # tmp = gm.generate_matrix()
325     # for key in tmp:
326     #     print(tmp[key])
327     # for order in range(3, 11):
328     #     # gm = GenerateMatrices(order=order,
            iterations=1000, std_rate=config.rho_table[
            order])
329     #     # gm = GenerateMatrices(order=order,
            iterations=100000, std_rate=config.rho_table[
            order])
330     #     gm = GenerateMatrices(order=order,
            iterations=10000, std_rate=config.rho_table[
            order])
331     #     gm.generate_and_save()
332
333     # draw_means_graph(is_fit=False, is_save=False,
            is_generate=False)
334     draw_means_graph(is_fit=True, is_save=False,
            is_generate=False)
```

```
1 # coding:utf8
2
```

```
3  """
4  @author: Zhangao Lu
5  @contact: zlu2@laurentian.ca
6  @time: 2021/2/27
7  @description:
8  1. Use several methods to compute matrix distance.
9  Euclidean distance
10 Chebyshev distance
11 ...
12 2. Analyze the results.
13 """
14
15 import numpy as np
16 import pandas as pd
17 import matplotlib.pyplot as plt
18 import seaborn as sns
19 import matplotlib.cm as cm
20 from scipy.spatial.distance import cdist
21 from scipy.stats import entropy
22 from config import config
23 from utils.gerenal_tools import open_pickle,
      save_pickle
24 from pairwise_comparison.generate_NSI_PC_matrices
      import GenerateMatrices
25 from utils.pairwise_comparison_tools import
      compute_kii
26
27
28 class MatrixDistance(object):
29     def __init__(self):
30         pass
31
32     @staticmethod
33     def compute_distance(m1, m2, metric="euclidean",
          **kwargs):
34         """
35         Compute distance between each pair of the two
              collections of inputs.
36         :param m1: ndarray
37                    It is a np.array with shape (n, n)
```

```
                    here.
38          :param m2: ndarray
39                     It is a np.array with shape (n, n)
                       here.
40          :param metric: string, default = "euclidean".
41                     The distance function can be "
                          braycurtis", "canberra", "
                          chebyshev", "cityblock",
42                     "correlation", "cosine", "dice",
                           "euclidean", "hamming", "
                          jaccard", "jensenshannon", "
                          kulsinski",
43                     "mahalanobis", "matching", "
                          minkowski", "rogerstanimoto",
                           "russellrao", "seuclidean",
44                     "sokalmichener", "sokalsneath",
                          "sqeuclidean", "wminkowski",
                          "yule", "KLdivergence".
45          :return: ds float
46                   The value of distance.
47          """
48          if metric == "KLdivergence":
49              """
50              entropy(): Calculate the entropy of a
                   distribution for given probability
                   values.
51              m1.shape = (n, n)
52              m1.reshape(1, -1).shape = (1, n^2)
53              np.squeeze(m1.reshape(1, -1)).shape = (n
                   ^2,)
54              """
55              ds = entropy(np.squeeze(m1.reshape(1, -1))
                   , np.squeeze(m2.reshape(1, -1)))
56          else:
57              ds = cdist(m1.reshape(1, -1), m2.reshape
                   (1, -1), metric=metric)
58              ds = ds[0][0]  # cdist will return a
                   ndarray, so use ds[0][0] to get a float
                    number.
59          return ds
```

```
60
61
62 def create_rho_table():
63     """
64     Create a table about matrix order, threshold and \
          rho, then save it in hard drive.
65               threshold_1  threshold_2
66     order_1  rho_{11}     rho_{12}
67     order_2  rho_{21}     rho_{22}
68     :return: None
69     """
70     temp_dict = dict()
71     tmp = open_pickle(config.path_for_thesis + "rho.
          pkl")  # Access
72     for key in range(3, 11):
73         # print(key, tmp[key])
74         # [{'rho': 0.038956560762328174, 'a':
              -1.4810377291603385, 'b':
              1.3411769934668802, 'threshold': 0.05}, {}]
75         for elm in tmp[key]:
76             if elm["threshold"] not in temp_dict:
77                 temp_dict[elm["threshold"]] = [elm["
                      rho"]]
78             else:
79                 temp_dict[elm["threshold"]].append(elm
                      ["rho"])
80     # The indices are orders, the columns are
          thresholds, and the elements are values of \rho
          .
81     df = pd.DataFrame(temp_dict, index=range(3, 11))
82     print(df)
83     save_pickle(df, config.path_for_thesis + "rho and
          order table.pkl")
84
85
86 def generate_and_compute_the_distances(metric_list,
       iterations=1000):
87     """
88     Generate PC matrices and NSI PC matrices with
          different constraint: \rho (the mean of Kii).
```

```
        Then compute the
89      distances between them. Finally, save the results.
90      :param metric_list: list
91              A list of metrics. Check config.mc_para_1.
92      :param iterations: int, default = 1000
93              Check config.mc_para_1.
94      :return: None
95      """
96      md = MatrixDistance()
97      tb = open_pickle(config.path_for_thesis + "rho and
            order table.pkl")  # Get the table about the \
        rho.
98      print(tb)
99      res = dict()
100     for metric in metric_list:  # For each metric in
        the list
101         res[metric] = dict()  # res = {metric1: {},
            metric2: {}, ... }
102         for ind in tb.index:  # tb is the table of \
            rho and order, and the index of tb is the
            matrix order.
103             order = ind  # Rename it for better
                understanding.
104             res[metric][order] = dict()
105             # Threshold is the mean of Kii, which is
                set in advance.
106             for threshold in np.array(range(8, 14, 1))
                    / 100.0:
107                 print(metric, ind, threshold)
108                 rho = tb.loc[ind][threshold]  # Select
                        the \rho from the table.
109                 # Generate matrices.
110                 gm = GenerateMatrices(order=order,
                        iterations=iterations, std_rate=rho
                        )
111                 # Compute all the distances in loops
112                 res[metric][order][threshold] = list()
                        # res = {metric1: {order1:{
                        threshold1: []}}}
113                 for _ in range(iterations):
```

```
114                     # generate and get the PC and NSI
                           PC matrix
115                     tmp = gm.generate_matrix()
116                     pc = tmp["PC"]
117                     nsi_pc = tmp["NSI_PC"]
118                     # Compute the distance between PC
                           matrices and NSI PC matrices
119                     dt = md.compute_distance(pc,
                           nsi_pc, metric)
120                     res[metric][order][threshold].
                           append(dt)
121     save_pickle(res, config.path_for_thesis + "
           distances for %d matrices" % iterations)
122
123
124 def analysis_results_chart1(metric_list, iterations,
       is_show=False, dpi=600):
125     """
126     Draw Letter-Value Plots for Section 4.2 in the
           thesis.
127     :param metric_list: list
128             A list of metrics. Check config.mc_para_1.
129     :param iterations: int
130             Check config.mc_para_1.
131     :param is_show: boolean, default = False
132             Show the plot or not. Check config.
                mc_para_1.
133     :param dpi: int, default = 600
134             When dpi is very high, the speed that latex
                    compile the file is very slow. 200 is
                recommended for test.
135             Check config.mc_para_1.
136     :return: None
137     """
138     # Access the results computed through function:
           generate_and_compute_the_distances
139     # res = {metric: {order: {threshold: []}}}
140     res = open_pickle(config.path_for_thesis + "
           distances for %d matrices" % iterations)
141
```

```
142     for metric in metric_list:
143         tmp = dict()
144         for order in res[metric]:  # res[metric] = {
               order: {threshold: []}}
145             # All experiments are base on threshold =
                   0.1. So select the data where threshold
                    = 0.1.
146             tmp[order] = res[metric][order][config.
                   threshold]
147         df = pd.DataFrame(tmp)  # Convert to pandas.
               Dataframe.
148         order = list(range(3, 11))  # The X-axis of
               the graph.
149         plot_name = config.printed_metric[metric]  #
               Covert the metric names to print.
150         """
151         For Chebyshev distance and Euclidean distance,
               there are two graphs need to draw.
152         The first one is the whole graph, and the
               second one the partial enlarged view.
153         """
154         if metric in ["chebyshev", "euclidean"]:
155             sns.boxenplot(data=df, order=order)  #
                   Letter-Value Plot
156             plt.title("The Distribution of %s" %
                   plot_name)  # Set the title of the
                   graph.
157             plt.xlabel('the order of matrices')  # Set
                    the label of X-axis.
158             plt.ylabel('distance/similarity/divergence
                   ')  # Set the label of Y-axis.
159             # If is_show, then show the graph.
                   Otherwise, save it.
160             if is_show:
161                 plt.show()
162             else:
163                 plt.savefig(config.path_for_thesis + "
                       distribution_of_distances_%s_a.png"
                        % metric,
164                             format="png",
```

```
165                                 dpi=dpi)
166
167             sns.boxenplot(data=df, order=order,
                    showfliers=False)  # Letter-Value Plot
168             plt.ylim([0, config.max_ylim[metric]])  #
                    Limit the Y-axis to show more details.
169             plt.title("The Distribution of %s" %
                    plot_name)
170             plt.xlabel('the order of matrices')
171             plt.ylabel('distance/similarity/divergence
                    ')
172             if is_show:
173                 plt.show()
174             else:
175                 plt.savefig(config.path_for_thesis + "
                        distribution_of_distances_%s_b.png"
                         % metric,
176                                 format="png",
177                                 dpi=dpi)
178         else:
179             sns.boxenplot(data=df, order=order)  #
                    Letter-Value Plot
180             plt.title("The Distribution of %s" %
                    plot_name)
181             plt.xlabel('the order of matrices')
182             plt.ylabel('distance/similarity/divergence
                    ')
183             if is_show:
184                 plt.show()
185             else:
186                 plt.savefig(config.path_for_thesis + "
                        distribution_of_distances_%s.png" %
                         metric, format="png", dpi=dpi)
187
188
189 def analysis_results_table(metric_list, iterations,
        need_order):
190     """
191     Generate the tables to show the statistical
            indicators for different orders, which is
```

```
        displayed in
192     Section 4.2 of the thesis.
193     :param metric_list: list
194             A list of metrics. Check config.mc_para_1.
195     :param iterations: int
196             Check config.mc_para_1.
197     :param need_order: int
198             Check config.mc_para_1.
199             I only set order=4 or order=8 for my thesis
                .
200     :return: None
201     """
202     # Access the results computed through function:
           generate_and_compute_the_distances
203     # res = {metric: {order: {threshold: []}}}
204     res = open_pickle(config.path_for_thesis + "
           distances for %d matrices" % iterations)
205     need_merge = list()
206     for metric in metric_list:
207         tmp = dict()
208         for order in res[metric]:  # res[metric] = {
               order: {threshold: []}}
209             # All experiments are base on threshold =
                   0.1. So select the data where threshold
                    = 0.1.
210             tmp[order] = res[metric][order][config.
                   threshold]
211         df = pd.DataFrame(tmp)
212         """
213         An example of df.describe()
214                                3              4   ...
                                                   9
                                                    10
215         count  100000.000000  100000.000000  ...
               100000.000000  100000.000000
216         mean        0.024043       0.014955  ...
                    0.009011       0.008663
217         std         0.010873       0.005100  ...
                    0.001548       0.001369
218         min         0.000922       0.000649  ...
```

```
                    0.001851        0.002561
219         25%        0.016504        0.011468  ...
                    0.008016        0.007785
220         50%        0.022294        0.014261  ...
                    0.008873        0.008547
221         75%        0.029504        0.017615  ...
                    0.009832        0.009401
222         max        0.141391        0.073217  ...
                    0.022877        0.022084
223         """
224         need_merge.append(df.describe()[need_order])
225     mdf = pd.concat(need_merge, axis=1)  # Merge all
           pandas.Series and get a big matrix or pandas.
           Dataframe.
226     mdf.columns = [config.printed_metric[elm] for elm
           in metric_list]  # Set the column names.
227     mdf = mdf.T  # Transpose the matrix.
228     mdf = mdf[["mean", "std", "min", "25%", "50%",
           "75%", "max"]]  # Select needed statistical
           measurements.
229     # Print specific sentences, which is designed for
           writing the table in Latex.
230     print(" name & " + " & ".join(list(mdf.columns)) +
            "\\\\" + " \\hline")
231     for ind in mdf.index:
232         print(ind.split()[0] + " & " + " & ".join(map(
               lambda x: str(round(x, 4)), list(mdf.loc[
               ind]))) + "\\\\" + " \\hline")
233
234
235 def analysis_results_chart2(metric_list, iterations,
       is_show=False, dpi=600):
236     """
237     Draw bubble charts for Section 4.3 in the thesis.
238     :param metric_list: list
239             A list of metrics. Check config.mc_para_2.
240     :param iterations: int
241             Check config.mc_para_2.
242     :param is_show: boolean, default = False
243             Show the plot or not. Check config.
```

```
            mc_para_2.
    :param dpi: int, default = 600
           When dpi is very high, the speed that latex
               compile the file is very slow. 200 is
              recommended for test.
           Check config.mc_para_2.
    :return: None
    """
    # Access the results computed through function:
       generate_and_compute_the_distances
    # res = {metric: {order: {threshold: []}}}
    res = open_pickle(config.path_for_thesis + "
       distances for %d matrices" % iterations)
    for metric in metric_list:
        fig = plt.figure(figsize=(12.00, 8.00))
        # Means are used to set the points of bubbles
           while standard deviations are used to set
           the size of bubbles.
        tmp1 = dict()  # To save the values of mean
        tmp2 = dict()  # To save the values of std
        plot_name = config.printed_metric[metric]
        for order in res[metric]:  # res[metric] = {
           metric: {order: {threshold: []}}}
            tmp1[order] = dict()
            tmp2[order] = dict()
            for threshold in res[metric][order]:  #
               res[metric][order] = {threshold: []}
                tmp1[order][threshold] = np.mean(res[
                   metric][order][threshold])
                tmp2[order][threshold] = np.std(res[
                   metric][order][threshold])
        df1 = pd.DataFrame(tmp1)
        """
        An example of df1
        The indices are the thresholds. The columns
           are matrices' orders. The elements are mean
            of distances.
                    3         4         5   ...
                              8         9         10
        0.08  0.019007  0.011864  0.009661  ...
```

```
            0.007491   0.007152   0.006878
270         0.09   0.021488   0.013406   0.010922   ...
            0.008462   0.008074   0.007772
271         0.10   0.024043   0.014955   0.012183   ...
            0.009432   0.009011   0.008663
272         0.11   0.026739   0.016543   0.013473   ...
            0.010423   0.009945   0.009572
273         0.12   0.029309   0.018190   0.014790   ...
            0.011426   0.010894   0.010493
274         0.13   0.031895   0.019769   0.016071   ...
            0.012431   0.011862   0.011411
275         """
276         df2 = pd.DataFrame(tmp2)
277         """
278         An example of df2
279         The indices are the thresholds. The columns
               are matrices' orders. The elements are mean
                of distances.
280                      3         4         5   ...
                                   8         9         10
281         0.08   0.008581   0.004023   0.002682   ...
               0.001405   0.001231   0.001086
282         0.09   0.009759   0.004555   0.003033   ...
               0.001592   0.001385   0.001232
283         0.10   0.010873   0.005100   0.003373   ...
               0.001766   0.001548   0.001369
284         0.11   0.012127   0.005634   0.003737   ...
               0.001955   0.001699   0.001509
285         0.12   0.013237   0.006177   0.004099   ...
               0.002150   0.001867   0.001656
286         0.13   0.014382   0.006742   0.004443   ...
               0.002327   0.002038   0.001806
287         """
288         cnt = 0  # counter
289         for ind in df1.index:
290             x = df1.columns
291             y = df1.loc[ind]
292             # The original size of the bubbles are too
                    small. So set a ratio to zoom in it.
293             size = df2.loc[ind] * config.size_dict[
```

```
                metric]
294             plt.scatter(x, y, size, c=x, cmap=cm.
                    get_cmap("coolwarm"))
295             plt.plot(x, y, config.color_list[cnt],
                    linestyle="--", label="the mean of Kii
                    =%0.2f" % ind)
296             cnt += 1
297         plt.title("Distributions of %ss with Respect
                to Different Matrix Orders and Means of Kii
                " % plot_name)
298         plt.xlabel('the order of matrices')
299         plt.ylabel('the mean of distances/similarities
                /divergences')
300         plt.legend()
301         if is_show:
302             plt.show()
303         else:
304             fig.savefig(config.path_for_thesis + "
                    thresholds_%s_distribution.png" %
                    metric, format="png", dpi=dpi)
305
306
307 def func_canberra_distance(error, m):
308     """
309     A quick method to compute the canberra distance
            between two numbers.
310     n = m + error, m > 0 and n > 0
311     d = |n-m| / (|m| + |n|) = |e| / (2m + error)
312     :param error: float
313     :param m: float
314     :return: float
315              The canberra distance.
316     """
317     return abs(error) / (2 * m + error)
318
319
320 def differences_between_canberra_distances(sigma,
        delta_sigma):
321     """
322     Generate two random errors from two different
```

```
        normal distributions with the original number m
        ,
323     then compute the canberra distances between two
        random samples and m.
324     After that, return the differences between two
        canberra distances.
325     :param sigma: float
326             The standard deviation of the normal
                distribution.
327     :param delta_sigma: float
328             Measure the change of sigma.
329     :return: float
330              The differences between two canberra
                 distances.
331     """
332     while 1:
333         m = np.random.random() / np.random.random()  #
                m is the elements of any PC matrices.
334         gm = GenerateMatrices()
335         error = gm.random_numbers(sigma, m, 0)  #
            Generate a random error from original
            distribution.
336         new_error = gm.random_numbers(sigma +
            delta_sigma, m, 0)  # Generate another
            error from the new distribution.
337         if m > 0 and m + error > 0 and m + error +
            new_error > 0:  # All random values should
            be greater than zero.
338             return func_canberra_distance(new_error, m
                ) - func_canberra_distance(error, m)
339
340
341 def analysis_result_canberra_distance(iterations,
    is_show=False):
342     """
343     Draw a heat map for canberra distance which is
        also used in Section 4.3 of the thesis.
344     The heat map demonstrates the distributions of the
         distances when order = 3.
345     :param iterations: int
```

```
346             Check config.mc_para_3.
347     :param is_show: boolean, default = False
348             Show the plot or not. Check config.
                    mc_para_3.
349     :return: None
350     """
351     # sigma = \rho * origin_num, [0.1, 0.2, ... 1]
352     fig = plt.figure(figsize=(12.00, 8.00))
353     cnt = 1
354     # \rho - \kappa table when order = 3, kappa is
            defined in thesis as the mean of Kii.
355     kappas = [0.1, 0.2, 0.3, 0.4, 0.5, 0.56]
356     rho_table = {0.1: 0.0781, 0.2: 0.1705, 0.3: 0.274,
            0.4: 0.4003, 0.5: 0.5783, 0.56: 0.8608}
357     for kappa in kappas:
358         x = list(range(iterations))
359         y = list()
360         for _ in range(iterations):
361             value =
                    differences_between_canberra_distances(
                    config.sigma, delta_sigma=rho_table[
                    kappa])
362             y.append(value)
363         df = pd.DataFrame({'x': x, 'y': y, 'color': pd
                .cut(y, 10, labels=range(1, 11))})
364         print(df)
365         plt.subplot(2, 3, cnt)  # Set 6 sub-plots.
366         cmap = sns.cubehelix_palette(start=0.1, light
                =1, as_cmap=True)
367         sns.kdeplot(x, y, cmap=cmap, shade=True, cut
                =5)  # Draw heat maps.
368         plt.title("%s = %0.2f (%s = %0.2f)" % (chr
                (954), kappa, chr(961), rho_table[kappa]),
                config.ft)
369         cnt += 1
370     plt.suptitle("The Differences Distribution with
            Respect to %s" % chr(954))  # Set the sub
            titles.
371     if is_show:
372         plt.show()
```

```
373     else:
374         # fig.savefig(config.path_for_thesis + "
              distributions_of_differences_cd_highDPI.png
              ", format="png", dpi=200)
375         fig.savefig(config.path_for_thesis + "
              distributions_of_differences_cd.png",
              format="png", dpi=100)
376
377
378 def create_table_for_alpha(metric_list, iterations):
379     """
380     Create a alpha-order table, which will be used in
          reconstruct.py and Section 5.1 of the thesis.
381     :param metric_list: list
382             A list of metrics. Check config.mc_para_2.
383     :param iterations: int
384             Check config.mc_para_3.
385     :return: None
386     """
387     table = dict()
388     # Access the results computed through function:
          generate_and_compute_the_distances
389     # res = {metric: {order: {threshold: []}}}
390     res = open_pickle(config.path_for_thesis + "
          distances for %d matrices" % iterations)
391     # Print specific sentences, which is designed for
          writing the table in Latex.
392     print("Order & Bray-Curtis Distance & Canberra
          Distance & Jensen-Shannon Divergence" + "\\\\"
          + " \\hline")
393     for order in range(3, 11):
394         for metric in res:
395             if metric in metric_list:
396                 table[metric] = dict()
397                 rm = GenerateMatrices(iterations=
                      iterations, order=order)
398                 arrays = rm.read_array()  # Access the
                      NPI PC matrices from hard drive.
399                 nsi_pc_list = arrays[1]
400                 tmp = list()
```

```
401                 for ind in range(len(nsi_pc_list)):
402                     m_prime = np.squeeze(nsi_pc_list[
                            ind])  # (n, n, 1) -> (n, n)
403                     tmp.append(compute_kii(m_prime))
                            # Compute the kii of the
                            matrices
404                 table[metric][order] = {"mean of kii":
                        np.mean(tmp),
405                                         "mean of
                                            distances":
                                             np.mean(
                                            res[metric
                                            ][order][
                                            config.
                                            threshold])
                                            ,
406                                         "ratio": np.
                                            mean(tmp) /
                                             np.mean(
                                            res[metric
                                            ][order][
                                            config.
                                            threshold])
407                                         }
408         # Print specific sentences, which is designed
                for writing the table in Latex.
409         print(" & ".join([str(order), str(round(table
                ["braycurtis"][order]["ratio"], config.
                decimal_places)),
410                           str(round(table["canberra"][
                                order]["ratio"], config.
                                decimal_places)),
411                           str(round(table["
                                jensenshannon"][order]["
                                ratio"], config.
                                decimal_places))])
412               + "\\\\" + " \\hline")
413     save_pickle(table, config.path_for_thesis + "alpha
            table for %d matrices" % iterations)
414
```

```
415
416 if __name__ == '__main__':
417     # m1 = np.array(range(1, 10)).reshape(3, 3)
418     # m2 = m1 + np.random.randn()
419     # print(m1, "\n", m2)
420     # md = MatrixDistance()
421     # # md.compute_distance(m1, m2)
422     # create_rho_table()
423
424     # generate_and_compute_the_distances(config.
            mc_para_1["metrics"], config.mc_para_1["
            iterations"])
425     # analysis_results_chart1(config.mc_para_1["
            metrics"], config.mc_para_1["iterations"],
426     #                         config.mc_para_1["
            is_show"], config.mc_para_1["dpi"])
427     # analysis_results_table(config.mc_para_1["metrics
            "],
428     #                        config.mc_para_1["
            iterations"],
429     #                        config.mc_para_1["
            need_order"])
430
431     # analysis_results_chart2(config.mc_para_2["
            metrics"], config.mc_para_2["iterations"],
432     #                         config.mc_para_2["
            is_show"], config.mc_para_2["dpi"])
433
434     # analysis_result_canberra_distance(config.
            mc_para_3["iterations"], config.mc_para_3["
            is_show"])
435     create_table_for_alpha(config.mc_para_3["metrics
            "], config.mc_para_3["iterations"])
```

```
1 # coding:utf8
2
3 """
4 @author: Zhangao Lu
5 @contact: zlu2@laurentian.ca
6 @time: 2021/3/16
```

```
7  @description:
8  1. Reconstruct PC matrix from NSI PC matrix.
9  """
10
11 import pandas as pd
12 import matplotlib.pyplot as plt
13 import time
14 import seaborn as sns
15 from scipy import optimize
16 from pairwise_comparison.matrix_distance import
       MatrixDistance
17 from pairwise_comparison.generate_NSI_PC_matrices
       import GenerateMatrices
18 from utils.pairwise_comparison_tools import *
19 from multiprocessing import cpu_count
20 from utils.gerenal_tools import open_pickle,
       save_pickle, open_hickle, save_hickle, my_round
21 from utils.printing_format import PrintingFormat
22 from config import config
23 from config.config import key_names
24
25 np.set_printoptions(suppress=True)  # Do not use
       scientific notation when printing matrix.
26 np.set_printoptions(threshold=np.inf)  # Do not use
       Ellipsis when printing matrix.
27
28
29 class ReconstructMatrices(MatrixDistance,
       GenerateMatrices):
30
31     def __init__(self):
32         super().__init__()
33         GenerateMatrices.__init__(self)
34         self.alpha = 1.0  # The weight coefficient in
               the objective function, see Section 5.1 in
               the thesis.
35         self.m_origin = np.array([])  # PC matrix
36         self.v_origin = []  # original vector
37         self.metric = "braycurtis"
38         self.metric_list = config.mc_para_2["metrics"]
```

```
            # The three metrics needed to analyze in section 5.
        self.alpha_plan = ""
        self.file_name_of_reconstruct_result = "%d matrices reconstructed result alpha plan=%s.pkl" % \
                                               (self.iterations, str(self.alpha_plan))
        self.file_name_of_new_pc = "%d new pc matrices with order=%d metric=%s alpha=%0.4f.pkl" % \
                                   (self.iterations, self.order, self.metric, self.alpha)
        self.is_show = False  # If is_show is True, show the graph.
        self.dpi = 200  # Set the dpi of the graphs.
        self.alpha_table = {}

    def objective_function(self, v):
        """
        The objective function: f(m') = Kii(m') + \
           alpha * D(m', m)
        :param v: list
               v is a vector, and will be converted in to the matrix m'
        :return: float
                 The number computed by the objective function.
        """
        m_prime = vector_to_pc_matrix(v)  # Convert the vector into a PC matrix
        kii = compute_kii(m_prime)  # Compute the Kii
```

```
            of the matrix.
58          dt = self.compute_distance(m_prime, self.
               m_origin, self.metric)
59          goal = kii + self.alpha * dt
60          return goal
61
62      def reconstruct_matrix(self, m, beta=0.2, maxiter
           =1000, disp=False):
63          """
64          Reconstruct the PC matrix based on DE.
65          :param m: np.array
66                 original matrix, its shape is (n, n),
                      cannot be (n, n, 1)
67          :param beta: float, default = 0.2
68                 beta is the coefficient of the bounds.
69                 bounds = [v_i - beta * v_i, v_i + beta
                      * v_i], v_i is a element of the
                      original vector
70          :param maxiter: int, default = 1000
71                 The maximum number of generations.
72          :param disp: boolean
73                 Prints the evaluated function at every
                      iteration.
74          :return: np.array
75                 The reconstruct matrix.
76          """
77          self.m_origin = m  # Rename the matrix, then
               it can be printed in the loop.
78          bounds = list()  # Bounds for variables.
79          self.v_origin = pc_matrix_to_vector(self.
               m_origin)
80          for elm in self.v_origin:
81              bounds.append((elm - elm * beta, elm + elm
                    * beta))
82          r = optimize.differential_evolution(self.
               objective_function, bounds, workers=
               cpu_count(), maxiter=maxiter,
83                                              updating="
                                                   deferred
                                                   ", disp
```

```
                                                    =disp)
84          new_v = r.x
85          return vector_to_pc_matrix(new_v)
86
87      def run(self, iterations=1000, alpha_plan="plan1")
            :
88          """
89          Run the main function of this algorithm. It
               will reconstruct matrices and save the
               results.
90          :param iterations: int, default = 1000
91          :param alpha_plan: string, default = "plan1"
92                 It refers to a ratio used to change the
                    values of \alpha.
93          :return: None
94          """
95          self.alpha_table = self.read_alpha_table()
96          self.alpha_plan = alpha_plan
97          self.iterations = iterations
98          for self.order in range(3, 11):
99              arrays = super().read_array()
100             pc_list = arrays[0]  # List of PC matrices
                    .
101             nsi_pc_list = arrays[1]  # List of NSI PC
                    matrices.
102             for self.metric in self.metric_list:
103                 # Select the values of \alpha with
                        respect to different metrics and
                        orders
104                 self.alpha = self.alpha_table[self.
                        metric][self.order]["ratio"] *
                        config.alpha_table[self.alpha_plan]
105                 self.alpha = round(self.alpha, config.
                        decimal_places)
106                 print(self.order, self.metric, "%0.4f"
                        % self.alpha)
107                 new_pc_list = list()
108                 for ind in range(len(nsi_pc_list)):
109                     m_prime = np.squeeze(nsi_pc_list[
                            ind])  # (n, n, 1) -> (n, n)
```

```
110                     new_m_prime = self.
                            reconstruct_matrix(m_prime)
111                     new_pc_list.append(np.expand_dims(
                            new_m_prime, axis=2))
112                 # Save the optimized matrices for
                        further research.
113                 new_pc_array = np.concatenate(tuple(
                        new_pc_list), axis=2)
114                 self.file_name_of_new_pc = "%d new pc
                        matrices with order=%d metric=%s
                        alpha=%0.4f.pkl" % (
115                      self.iterations, self.order, self.
                            metric, self.alpha)
116                 save_hickle(new_pc_array, config.
                        path_for_thesis + self.
                        file_name_of_new_pc)
117
118     def read_new_array(self):
119         """
120         Access the reconstructed PC matrices from the
                hard drive.
121         :return: list
122                  The list of reconstructed PC matrices
                        .
123         """
124         self.file_name_of_new_pc = "%d new pc matrices
                with order=%d metric=%s alpha=%0.4f.pkl" %
                \
125                                    (self.iterations,
                                        self.order, self
                                        .metric, self.
                                        alpha)
126         new_pc_array = open_hickle(config.
                path_for_thesis + self.file_name_of_new_pc)
127         new_pc_list = np.split(new_pc_array,
                new_pc_array.shape[2], axis=2)  # (3, 3,
                1000) -> [(3, 3, 1)], len()=1000
128         return new_pc_list
129
130     def read_alpha_table(self):
```

```
131         """
132         Access the \alpha table from hard drive.
133         :return: None
134         """
135         # return open_pickle(config.path_for_thesis +
                "alpha table for %d matrices" % 1000)
136         return open_pickle(config.path_for_thesis + "
                alpha table for %d matrices" % self.
                iterations)
137 
138     def check_and_draw(self, iterations=1000, readable
            =True, alpha_plan="plan1", is_show=True):
139         """
140         Generate a complicated dict for drawing graphs
                .
141         res =
142         {self.order:
143             {self.metric: {"kn": [],
144                  "knn": [],
145                  "dnp": [],
146                  "dnnn": [],
147                  "dnnp": [],
148                 }
149             }
150         }
151         :param iterations: int, default = 1000
152         :param readable: int, default=True
153                If True, access the results from the
                       hard drive. Otherwise, compute them.
154         :param alpha_plan: string, default = "plan1"
155                It refers to a ratio used to change the
                        values of \alpha.
156         :param is_show: boolean, default = False
157                Show the plot or not.
158         :return: None
159         """
160         self.iterations = iterations
161         self.is_show = is_show
162         self.alpha_table = self.read_alpha_table()
163         self.alpha_plan = alpha_plan
```

```
164         res = dict()
165         self.file_name_of_reconstruct_result = "%d
               matrices reconstructed result alpha plan=%s
               .pkl" % \
166                                                 (self.
                                                       iterations
                                                       ,
                                                       str(
                                                       self
                                                       .
                                                       alpha_plan
                                                       ))
167         if readable:
168             res = open_pickle(config.path_for_thesis +
                    self.file_name_of_reconstruct_result)
169         else:  # If not readable, compute and save the
                data for further research.
170             for self.order in range(3, 11):
171                 res[self.order] = dict()  # res = {
                       self.order: {}}
172                 for self.metric in self.metric_list:
173                     self.alpha = self.alpha_table[self
                           .metric][self.order]["ratio"] *
                            config.alpha_table[
174                            self.alpha_plan]  # Select the
                                values of \alpha with
                               respect to different
                               metrics and orders
175                     new_pc_list = self.read_new_array
                           ()  # The list of Reconstructed
                            PC matrices.
176                     arrays = super().read_array()
177                     pc_list = arrays[0]  # The list of
                            original PC matrices.
178                     nsi_pc_list = arrays[1]  # The
                           list of NSI PC matrices.
179                     res[self.order][self.metric] =
                           dict()  # res = {self.order: {
                           self.metric: {}}}
180                     # For short, use "kn", "knn" and
```

```
                    so on. The full name is
                    displayed in config.py.
181                 temp_dict = {key_names["kn"]: [],
182                              key_names["knn"]: [],
183                              key_names["dnp"]: [],
184                              key_names["dnnn"]:
                                    [],
185                              key_names["dnnp"]: []
186                              }
187                 for ind in range(len(new_pc_list))
                        :
188                     m_prime = np.squeeze(
                            nsi_pc_list[ind])  # (n, n,
                             1) -> (n, n)
189                     m_origin = np.squeeze(pc_list[
                            ind])
190                     new_m_prime = np.squeeze(
                            new_pc_list[ind])
191                     # The full name of keys in
                            config.py has explained the
                             meaning of next 5
                            sentences.
192                     temp_dict[key_names["kn"]].
                            append(compute_kii(m_prime)
                            )
193                     temp_dict[key_names["knn"]].
                            append(compute_kii(
                            new_m_prime))
194                     temp_dict[key_names["dnp"]].
                            append(self.
                            compute_distance(m_origin,
                            m_prime, self.metric))
195                     temp_dict[key_names["dnnn"]].
                            append(self.
                            compute_distance(m_prime,
                            new_m_prime, self.metric))
196                     temp_dict[key_names["dnnp"]].
                            append(self.
                            compute_distance(m_origin,
                            new_m_prime, self.metric))
```

```
197                     res[self.order][self.metric] =
                            temp_dict
198             save_pickle(res, config.path_for_thesis +
                    self.file_name_of_reconstruct_result)
199
200         self.create_data_tables(res)
201
202         self.choose_data_to_draw(res, "alpha+metric+
                order+key-name.dnp")
203         self.choose_data_to_draw(res, "alpha+metric+
                order+key-name.dnnn")
204         self.choose_data_to_draw(res, "alpha+metric+
                order+key-name.dnnp")
205
206     def choose_data_to_draw(self, res, gtype):
207         """
208         Draw three letter-value plots for each metric.
                For example:
209         metric=braycurtis. X-axis: order. Y-axis:
                key_names.dnp.
210         metric=braycurtis. X-axis: order. Y-axis:
                key_names.dnnn.
211         metric=braycurtis. X-axis: order. Y-axis:
                key_names.dnnp.
212         :param res: dict
213                 A complicated dict.
214         res =
215         {self.order:
216             {self.metric: {"kn": [],
217                  "knn": [],
218                  "dnp": [],
219                  "dnnn": [],
220                  "dnnp": [],
221                 }
222             }
223         }
224         :param gtype: string
225                 Graph types. There are three gtypes
                        here:
226                 1. alpha+metric+order+key-name.dnp
```

```
227                 2. alpha+metric+order+key-name.dnnn
228                 3. alpha+metric+order+key-name.dnnp
229         :return: None
230         """
231         xlabel = "the order of matrices"
232         plot_format = "png"
233 
234         order_list = list(range(3, 11))  # The X-axis
               of the graph.
235         if gtype == "alpha+metric+order+key-name.dnp":
236             for self.metric in self.metric_list:
237                 tmp = dict()
238                 for self.order in res:
239                     tmp[self.order] = res[self.order][
                           self.metric][key_names["dnp"]]
240                 df = pd.DataFrame(tmp)
241                 ylabel = "distance"
242                 plot_name = "The Distribution of %s
                       Distances between \n the NSI PC
                       Matrices and the Original PC " \
243                             "Matrices" % self.metric.
                                capitalize()
244                 # gtype.split(".")[1] = "dnp"
245                 plot_saved_path = "
                       new_distribution_of_distances_%s_%s
                       .png" % (self.metric, gtype.split
                       (".")[1])
246                 self.draw_box_plots(data=df, xaxis=
                       order_list, plot_name=plot_name,
                       xlabel=xlabel, ylabel=ylabel,
247                                     plot_saved_path=
                                           plot_saved_path
                                           , plot_format=
                                           plot_format)
248 
249         elif gtype == "alpha+metric+order+key-name.
               dnnn":
250             for self.metric in self.metric_list:
251                 tmp = dict()
252                 for self.order in res:
```

```
253                     tmp[self.order] = res[self.order][
                           self.metric][key_names["dnnn"]]
254                 df = pd.DataFrame(tmp)
255                 ylabel = "distance"
256                 plot_name = "The Distribution of %s
                       Distances between \n the NSI PC
                       Matrix and Optimized Matrix" \
257                             % self.metric.capitalize()
258                 plot_saved_path = "
                       new_distribution_of_distances_%s_%s
                       .png" % (self.metric, gtype.split
                       (".")[1])
259                 self.draw_box_plots(data=df, xaxis=
                       order_list, plot_name=plot_name,
                       xlabel=xlabel, ylabel=ylabel,
260                                         plot_saved_path=
                                               plot_saved_path
                                               , plot_format=
                                               plot_format)
261
262         elif gtype == "alpha+metric+order+key-name.
                dnnp":
263             for self.metric in self.metric_list:
264                 tmp = dict()
265                 for self.order in res:
266                     tmp[self.order] = res[self.order][
                           self.metric][key_names["dnnp"]]
267                 df = pd.DataFrame(tmp)
268                 ylabel = "divergence"
269                 # plot_name = "alpha=%0.4f x %d
                       metric=%s" % (self.alpha_table[self
                       .metric][self.order]["ratio"],
270                 #

                       config.alpha_table[self.alpha_plan
                       ], self.metric)
271                 plot_name = "The Distribution of %s
                       Divergences between \n the Original
                        PC Matrix and Optimized " \
272                             "Matrix" % self.metric.
```

```
                                    capitalize()
273                 plot_saved_path = "
                       new_distribution_of_distances_%s_%s
                       .png" % (self.metric, gtype.split
                       (".")[1])
274                 self.draw_box_plots(data=df, xaxis=
                       order_list, plot_name=plot_name,
                       xlabel=xlabel, ylabel=ylabel,
275                                        plot_saved_path=
                                              plot_saved_path
                                              , plot_format=
                                              plot_format)
276
277     def create_data_tables(self, res):
278         """
279         create a complicated table, see table 6 in
               Section 5.2 of the thesis.
280         :param res: dict
281                A complicated dict. res =
282         {self.order:
283             {self.metric: {"kn": [],
284                  "knn": [],
285                  "dnp": [],
286                  "dnnn": [],
287                  "dnnp": [],
288                 }
289             }
290         }
291         :return: None.
292         """
293         for self.metric in self.metric_list:
294             # Print the data with some specific format
                   , which is used to create tables in the
                    LaTex file.
295             np1 = PrintingFormat()
296             np2 = PrintingFormat()
297             np3 = PrintingFormat()
298
299             np1.for_reconstruct()
300             np2.for_reconstruct()
```

```
301             np3.for_reconstruct()
302             for self.order in res:
303                 tmp1 = dict()
304                 tmp2 = dict()
305                 tmp3 = dict()
306                 tmp1[self.order] = res[self.order][
                        self.metric][key_names["dnp"]]
307                 tmp2[self.order] = res[self.order][
                        self.metric][key_names["dnnn"]]
308                 tmp3[self.order] = res[self.order][
                        self.metric][key_names["dnnp"]]
309
310                 df1 = pd.DataFrame(tmp1)
311                 df2 = pd.DataFrame(tmp2)
312                 df3 = pd.DataFrame(tmp3)
313                 # print(df1.describe().loc[["mean", "
                        std", "min", "max"]])
314                 # print(df2.describe().loc[["mean", "
                        std", "min", "max"]])
315                 # print(df3.describe().loc[["mean", "
                        std", "min", "max"]])
316
317                 np1.for_reconstruct(mean=df1.describe
                        ()[self.order]["mean"],
318                                     std=df1.describe()
                                        [self.order]["
                                        std"],
319                                     min_value=df1.
                                        describe()[self
                                        .order]["min"],
320                                     max_value=df1.
                                        describe()[self
                                        .order]["max"])
321                 np2.for_reconstruct(mean=df2.describe
                        ()[self.order]["mean"],
322                                     std=df2.describe()
                                        [self.order]["
                                        std"],
323                                     min_value=df2.
                                        describe()[self
```

```
                                                .order]["min"],
324                                         max_value=df2.
                                                describe()[self
                                                .order]["max"])
325                 np3.for_reconstruct(mean=df3.describe
                        ()[self.order]["mean"],
326                                         std=df3.describe()
                                                [self.order]["
                                                std"],
327                                         min_value=df3.
                                                describe()[self
                                                .order]["min"],
328                                         max_value=df3.
                                                describe()[self
                                                .order]["max"])
329             np1.for_reconstruct(end=True)
330             np2.for_reconstruct(end=True)
331             np3.for_reconstruct(end=True)
332
333             print(self.metric)
334             print("=" * 100)
335             np1.print_need_print()
336             print("=" * 100)
337             np2.print_need_print()
338             print("=" * 100)
339             np3.print_need_print()
340             print("=" * 100)
341
342     def draw_box_plots(self, data, xaxis, plot_name,
            xlabel, ylabel, plot_saved_path, plot_format):
343         """
344         Draw box plots according to different
                parameters.
345         :param data: pandas.Dataframe
346                 The data used to draw the graph.
347         :param xaxis: list
348                 The values of X-axis.
349         :param plot_name: string
350                 The name of this graph.
351         :param xlabel: string
```

```
352                 The label of X-axis.
353         :param ylabel: string
354                 The label of Y-axis.
355         :param plot_saved_path: string
356                 The file path to save the graph.
357         :param plot_format: string
358                 png or other formats.
359         :return: None
360         """
361         sns.boxenplot(data=data, order=xaxis)  # Draw
               the box plot.
362         plt.title(plot_name, config.new_ft)
363         plt.xlabel(xlabel, config.new_ft)
364         plt.ylabel(ylabel, config.new_ft)
365         if self.is_show:
366             plt.show()
367         else:
368             plt.savefig(config.path_for_thesis +
                   plot_saved_path, format=plot_format,
369                         dpi=self.dpi)
370             plt.close()
371
372     def check_outliers_of_kii(self, alpha_plan):
373         """
374         Check if there is any Kii of reconstructed PC
               matrix != 0.
375         :param alpha_plan: String
376                 plan1, plan2, plan3
377         :return:
378         """
379         self.alpha_table = self.read_alpha_table()
380         self.alpha_plan = alpha_plan
381         self.file_name_of_reconstruct_result = "%d
               matrices reconstructed result alpha plan=%s
               .pkl" \
382                                                % (self
                                                      .
                                                   iterations
                                                   ,
                                                   str(
```

```
                                                        self
                                                        .
                                                        alpha_plan
                                                        ))
383             for self.order in range(3, 11):
384                 for self.metric in self.metric_list:
385                     self.alpha = self.alpha_table[self.
                           metric][self.order]["ratio"] *
                           config.alpha_table[
386                            self.alpha_plan]
387                     new_pc_list = self.read_new_array()
388                     arrays = super().read_array()
389                     pc_list = arrays[0]
390                     nsi_pc_list = arrays[1]
391                     for ind in range(len(new_pc_list)):
392                         m_prime = np.squeeze(nsi_pc_list[
                               ind])  # (n, n, 1) -> (n, n)
393                         new_m_prime = np.squeeze(
                               new_pc_list[ind])
394                         kii1 = compute_kii(m_prime)
395                         kii2 = compute_kii(new_m_prime)
396                         if kii2 != 0:  # If kii2 !=0,
                               which means the DE algorithm
                               hasn't been converged correctly
                               .
397                             print(kii1, kii2)
398
399
400 if __name__ == '__main__':
401     rm = ReconstructMatrices()
402     # rm.run(10000, "plan1")
403     # rm.check_and_draw(iterations=10000, readable=
            False, alpha_plan="plan1", is_show=False)
404     # rm.check_and_draw(readable=False, alpha_plan="
            plan3", is_show=True)
405
406     # rm.check_outliers_of_kii("plan1")
```

References

[1] Juan Aguarón and José Mariéa Moreno-Jiménez. The geometric consistency index: Approximated thresholds. *European Journal of Operational Research*, 147(1):137–145, 2003.

[2] Jonathan Barzilai. Consistency measures for pairwise comparison matrices. *Journal of Multi-Criteria Decision Analysis*, 7(3):123–132, 1998.

[3] Thomas M Cover and Joy A Thomas. Entropy, relative entropy and mutual information. *Elements of information theory*, 2(1):12–13, 1991.

[4] GB Crawford. The geometric mean procedure for estimating the scale of a judgement matrix. *Mathematical Modelling*, 9(3-5):327–334, 1987.

[5] Swagatam Das and Ponnuthurai Nagaratnam Suganthan. Differential evolution: A survey of the state-of-the-art. *IEEE transactions on evolutionary computation*, 15(1):4–31, 2010.

[6] Bruce L Golden and Qiwen Wang. An alternate measure of consistency. In *The analytic hierarchy process*, pages 68–81. Springer, 1989.

[7] Hofmann Heike, H Wickham, and K Kafadar. Letter-value plots: Boxplots for large data. *J. Comput. Graph. Stat*, 26:469–477, 2017.

[8] Michael W Herman and Waldemar W Koczkodaj. A monte carlo study of pairwise comparisons. *arXiv preprint arXiv:1505.01888*, 2015.

[9] Włodzimierz Holsztyński and Waldemar W Koczkodaj. Convergence of inconsistency algorithms for the pairwise comparisons. *Information Processing Letters*, 59(4):197–202, 1996.

[10] Mykel J Kochenderfer and Tim A Wheeler. *Algorithms for optimization*. Mit Press, 2019.

[11] Waldemar W Koczkodaj. A new definition of consistency of pairwise comparisons. *Mathematical and computer modelling*, 18(7):79–84, 1993.

[12] Waldemar W Koczkodaj, Marek Kosiek, Jacek Szybowski, and Ding Xu. Fast convergence of distance-based inconsistency in pairwise comparisons. *Fundamenta Informaticae*, 137(3):355–367, 2015.

[13] Waldemar W Koczkodaj and Ryszard Szwarc. On axiomatization of inconsistency indicators for pairwise comparisons. *Fundamenta Informaticae*, 132(4):485–500, 2014.

[14] Waldemar W Koczkodaj and Jacek Szybowski. On the convergence of the pairwise comparisons inconsistency reduction process. *arXiv preprint arXiv:1505.01325*, 2015.

[15] WW Koczkodaj, F Liu, VW Marek, J Mazurek, Marcin Mazurek, L Mikhailov, C Özel, W Pedrycz, A Przelaskowski, A Schumann, et al. On the use of group theory to generalize elements of pairwise comparisons matrix: A cautionary note. *International Journal of Approximate Reasoning*, 124:59–65, 2020.

[16] Jouni Lampinen and Rainer Storn. Differential evolution. In *New optimization techniques in engineering*, pages 123–166. Springer, 2004.

[17] Thomas L Saaty. A scaling method for priorities in hierarchical structures. *Journal of mathematical psychology*, 15(3):234–281, 1977.

[18] Thomas L Saaty and Luis G Vargas. Comparison of eigenvalue, logarithmic least squares and least squares methods in estimating ratios. *Mathematical modelling*, 5(5):309–324, 1984.

[19] William E Stein and Philip J Mizzi. The harmonic consistency index for the analytic hierarchy process. *European journal of operational research*, 177(1):488–497, 2007.

[20] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[21] Jacek Szybowski, Konrad Kułakowski, and Anna Prusak. New inconsistency indicators for incomplete pairwise comparisons matrices. *Mathematical Social Sciences*, 2020.

[22] Peter JM Van Laarhoven and Witold Pedrycz. A fuzzy extension of saaty's priority theory. *Fuzzy sets and Systems*, 11(1-3):229–241, 1983.