多视角下的古代玻璃制品数据相关性分析

摘要

古代玻璃制品是中华文明的重要瑰宝之一，它不但记载着中国古代与外界往来的历史，也是先民劳动智慧的重要结晶。同为硅酸盐制品，陶瓷有着致密的釉面防止其侵蚀，而玻璃则不同，在长久的岁月洗刷下，古代玻璃制品大多遭到了严重的风化。正因于此，寻找一个古代玻璃制品的聚类方案，研究其类别与风化的关系乃至于对严重风化的玻璃制品进行风化前成分预测显得十分重要。

针对问题一，本文通过观察分析玻璃各特征的特点，将玻璃表面风化情况与玻璃纹饰、玻璃种类、玻璃颜色这种一对多的关系转化成四个数据两两之间的相关性，形成数据之间的列联表，并引入不确定系数作为数据相关性的重要参考标准，从而实现对各组分相关性的梳理，最终得到结论：玻璃颜色、种类与纹饰均与玻璃表面风化情况成正相关，其中，玻璃颜色的相关性最强，玻璃种类次之，玻璃纹饰最小。

同时，我们还引入了斯皮尔曼相关系数，对玻璃的化学组分进行了分析，深入探讨了各组分与风化程度之间的关系，并通过多元线性回归得出不同种玻璃在风化前后成分变化的重要数据，成功预测了风化玻璃在风化前的组分数据。

针对问题二、三，我们根据已知数据对玻璃进行了聚类，为了降低聚类的难度，我们针对玻璃的各项参数进行了降维。最后根据聚类结果，我们将各类玻璃共分为 9 个大类，即：高铜高钾玻璃、高铜铅钡玻璃、高铝高钾玻璃、铝磷高钾玻璃、高锶铅钡玻璃、铝镁铅钡玻璃、低硫铅钡玻璃、钙镁铅钡玻璃、风化高钾玻璃。而后，我们通过多元线性回归，对表格三种的玻璃分类进行了总结：A2、A3、A8 系铅钡玻璃，其余系高钾玻璃。

针对问题四，我们综合了前面问题的相关方法与答案，综合了自然与化学因素

最后，我们对整个模型进行了敏感性分析和检验。在本文的分析模型中，关于最核心的部分——即聚类部分，是通过机器学习反复迭代而得到的结果，数据结果在统计学意义上具有一定的稳定性，但是，一旦出现了与附件所给数据差异较大的玻璃制品时，聚类结果可能不理想。另外，在本模型的建立中，我们假设了玻璃风化的过程不会反作用于玻璃化学成分的变化，但是在数据分析中，二氧化硅与风化程度的关系出现了不正常的相关性，这很有可能是因为二氧化硅作为玻璃的主要载体，玻璃的风化对其成分流失存在影响。

关键词：不确定系数（Uncertainty coefficient），斯皮尔曼相关系数，层次分析法，逻辑斯蒂回归，K-Means 聚类

1. 问题重述

1.1 问题背景

玻璃制品在我国拥有悠久的历史。早期的玻璃虽是由中、西亚传入我国 [1] [2]，其制作工艺在经本土改良后却与过去形成了较大差异。两者烧制出来的玻璃制品尽管外形相似，但化学组成上却截然不同 3。不同于陶瓷制品，硅酸盐玻璃表面存在硅氧不饱和键，这些不饱和键会吸附大气中的水分而断键，玻璃制品缺少釉面的保护，极易受到外界环境的侵蚀 [3]。正因如此，玻璃相关文物的化学成分分析对文化溯源、文化分类有着重要意义 [4]，但由于玻璃制品的成分极易受环境影响而风化，寻找预测玻璃风化前化学成分的方法显得至关重要 [5]。

1.2 题目概括

（1）分析玻璃表面风化情况与玻璃各性质的关系，统计不同类型玻璃表面风化与否的成分差异，并基于此预测出已风化的玻璃在风化前的化学成分。

（2）通过已知的玻璃化学成分数据，对玻璃种类进行化学成分上的细分，并分析该划分方案的合理性与敏感性。

（3）基于已有方案，对未知种类的玻璃进行类别划定并分析该方案的敏感性。

（4）分析不同类别玻璃化学成分之间的关联关系，并比较其化学成分关联关系的差异性。

2. 问题分析

关于玻璃各部分数据的关系、分类、预测问题，实质上是综合分析已知数据，求出现有数据互相之间的相关性，并进一步向外推广，利用相关性来处理未知数据的问题。分析模型的核心在于相关性分析的精确性以及推广的普适性。

2.1 问题一分析

问题一首先已给定玻璃文物的类型、纹饰、颜色和表面风化情况，要分析它们之间的关系，就要对这四项没有明确数值的数据进行数据分析，建立、使用有效的相关性参数来推断相关性的高低情况。

对于文物表面有无风化的化学成分变化规律，我们可以将数据分为四个大类，即高钾玻璃无风化、高钾玻璃已风化、铅钡玻璃无风化、铅钡玻璃已风化。通过这四个大类的数据统计，我们可以近似得出玻璃风化前后的数据变化，从而对已风化玻璃风化前的数据进行预测。

2.2 问题二分析

问题二要根据已知分类：高钾玻璃和铅钡玻璃，分析出玻璃的分类依据，从而进一步对玻璃进行亚类分类，属于聚类问题。对玻璃的分类最终需要有较高的敏感性和合理性，应尽可能使得分类模型具有现实意义。

2.3 问题三分析

问题三实际上是对问题二的延伸，需要根据已知化学成分来推断玻璃的类别，是聚类后所

得模型的具体应用，在问题三的解决过程中，需要对数据进行前两问同样的数据处理，避免处理后的数据无法对应。

2.4 问题四分析

问题四是问题一、二的综合，既要考虑到玻璃各组分之间的相关性，又要使得所得结果在不同种类的玻璃之中具有区分度。在这一点上不但要考虑各成分在数值上的对应关系，也需要考虑在自然和化学上的客观事实，从而得出最优解。

3. 模型假设

假设一：附件中给出的数据均是从大规模的总体中随机抽取来的，数据具有统计学意义；

假设二：未能通过技术手段检测出的成分视为含量极低，数值上接近于 0；

假设三：风化结果不会反作用于玻璃组成成分的变化。

4. 名词解释和符号说明

4.1 名词解释

数据类型 [6]：按照 S. Stevens 的理论，数据共有四种类型：定类、定序、定距、定比。其中，定距和定比数据是常用的统计数据，在符合数据条件的情况下可以进行计算。而定类、定序两类数据通常情况下不能直接进行计算。

4.2 符号说明

表 1

符号	意义	数据类型
X_1	文物纹饰，由附件给出的A、B、C表示	定类
X_2	文物类型，0 代表高钾玻璃，1 代表铅钡玻璃	定类
X_3	文物颜色，由列出颜色的英文缩写或首字母表示	定类
Y	表面风化情况，0 代表未风化，1 代表风化	定距
Z	风化严重系数，0 代表未风化，1 代表风化，2 代表严重风化	定序
p	概率分布	定比
H	信息熵	定比
I	信息增益	定距
U	不确定系数	定距
R	位次序数	定序

ρ	斯皮尔曼相关系数	定距
J	损失函数	定比

5. 模型建立

5.1 数据预处理

对于附件表单 1 中给出的数据，本文为表面风化情况设为 Y，其值为 0 或 1。文物纹饰、文物类型、文物颜色分别设为X_1、X_2、X_3。

其中，文物纹饰X_1的三种类型已经由附件给出 A、B、C 三种代表形式，文物类型X_2同样设值为 0 或 1。文物颜色X_3涉及变量较多，为避免混淆，由字母缩写进行代替，其值为“AQ”（蓝绿色）、“D”（黑色）、“DB”（深蓝色）、“DG”（深绿色）、“G”（绿色）、“LB”（浅蓝色）、“LG”（浅绿色）、“V”（紫色），没有给出颜色的数据视为无效数据。

对于附件表单 2 中给出的数据，本文将未检出的成分含量视为 0，并舍弃了总成分比例低于 85%的两项数据（文物采样点 15、17）。并基于此设立风化严重系数 Z，其值为 0,1 或 2。

5.2 模型

5.2.1 基于玻璃相关性质的列联表

由题目给出的附件可以看出，玻璃纹饰、类型、颜色与表面风化情况是一组多维离散随机变量，显然本文可以通过列联表的方式来对数据相关性进行求解。

但这些离散随机变量中有的是二元的，如玻璃类型和表面风化情况，有些则是非二元的，如玻璃纹饰和玻璃颜色。在这四个变量之中，只有表面风化情况的数据类型可以视为定距数据，其他三类皆为定类数据。而且，在玻璃颜色这一变量中，一些数据出现的频次太少，不适合进行卡方检验[7][8][9]。

这无疑加大了数据的分析难度，在常用的数据分析系数中，皮尔森系数（Pearson correlation coefficient）[10]需要算出变量的总体均值，可定类数据无法得出一个有效且令人信服的均值来。以玻璃颜色为例，色彩感受是很主观的一类数据，在各种不同的颜色中，本文难以找到一个公认的光谱区间作为它的数据代表。而克兰姆系数（Cramér's V）[11]作为衡量分类数据之间相关量的相关系数，其对于两组变量列成列联表的行列数有较大的依赖[12][13]，而本题中十分容易出现行列数差距较大的情况，是以本文认为克莱姆系数在问题的解决中也不适用。

5.2.1.1 不确定系数

因此，我们决定引用不确定系数（Uncertainty coefficient）作文本模型的分析系数，它是亨利·泰尔（Henri Theil）利用信息熵提出的针对定类数据的一种数学工具。不确定系数的取值区间为[-0.1,+0.1]，值为-0.1 代表完全负相关，值为+0.1 代表完全正相关，其值为 0 是代表两者无关联。

对于本题中的某一种变量X_i或Y，其本身的值均具有不确定性，针对于此，我们可以求出该变量的基本信息熵H[14]：

$$H(A) = -\sum_{a \in A} p_A(a)\, log_c\big(p_A(a)\big) \tag{1}$$

其中，c是常数，其值只影响信息熵的单位，$p_A(a)$是关于X的概率分布：

$$p_X(x) = P(X = x) \qquad\qquad -\infty < x < +\infty \tag{2}$$

$$\sum_{x} p_X(x) = 1 \qquad\qquad p(x) > 0 \tag{3}$$

信息熵所描述的是信息的不确定程度，而在多个随机变量两两列联的情况下，可以在列联表中通过条件信息熵看出两者关系 [14]：

$$H(A|B) = -\sum_{A,B} P_{A,B}(a,b)\, log_c P_{A|B}(a|b) \tag{4}$$

条件信息熵的意义为：在随机变量B的值已知的情况下，随机变量A的不确定程度。

当条件信息熵的值与基本信息熵的值不同时，说明随机变量B对随机变量A产生了影响，该影响可以通过两者之间的差值I来表示 [15]：

$$I(A;B) = H(A) - H(A|B) = \sum_{a \in A} \sum_{b \in B} p(a,b)\, log_c\left(\frac{p(a,b)}{p(a)p(b)}\right) \tag{5}$$

I称为信息增益，它表示观察变量B对减小变量A的不确定程度的能力，相应的，我们用信息增益除以基本信息熵，最终可以得到一个比值，该比值描述的是观察变量B对减小变量A的不确定程度的比例 [16]：

$$U(X|Y) = \frac{H(X) - H(X|Y)}{H(X)} = \frac{I(X;Y)}{H(X)} \tag{6}$$

该比值即为不确定系数，又称为信息增益率。从公式中可以看出，分子分母均为底数为c的对数，由换底公式可知，不确定系数的最终结果与常数c无关。

显然地，由于信息熵恒为正值，所以不确定系数的正负只与信息增益有关，当B对A表现为正增益时，不确定系数就为正值，反之则为负值。不确定系数不受不同类别相对比例的影响，对本题解答十分有帮助 [17]。

5.2.1.2 玻璃相关性质之间的不确定系数

为了得出X_1、X_2、X_3与Y之间的关系，我们将它们两两之间列联后分别求它们的不确定系数，这样不但可以通过不确定系数看出自变量与因变量之间的关系，还可以看出自变量互相之间是否存在影响。我们将最终的结果绘制成热力图如下：

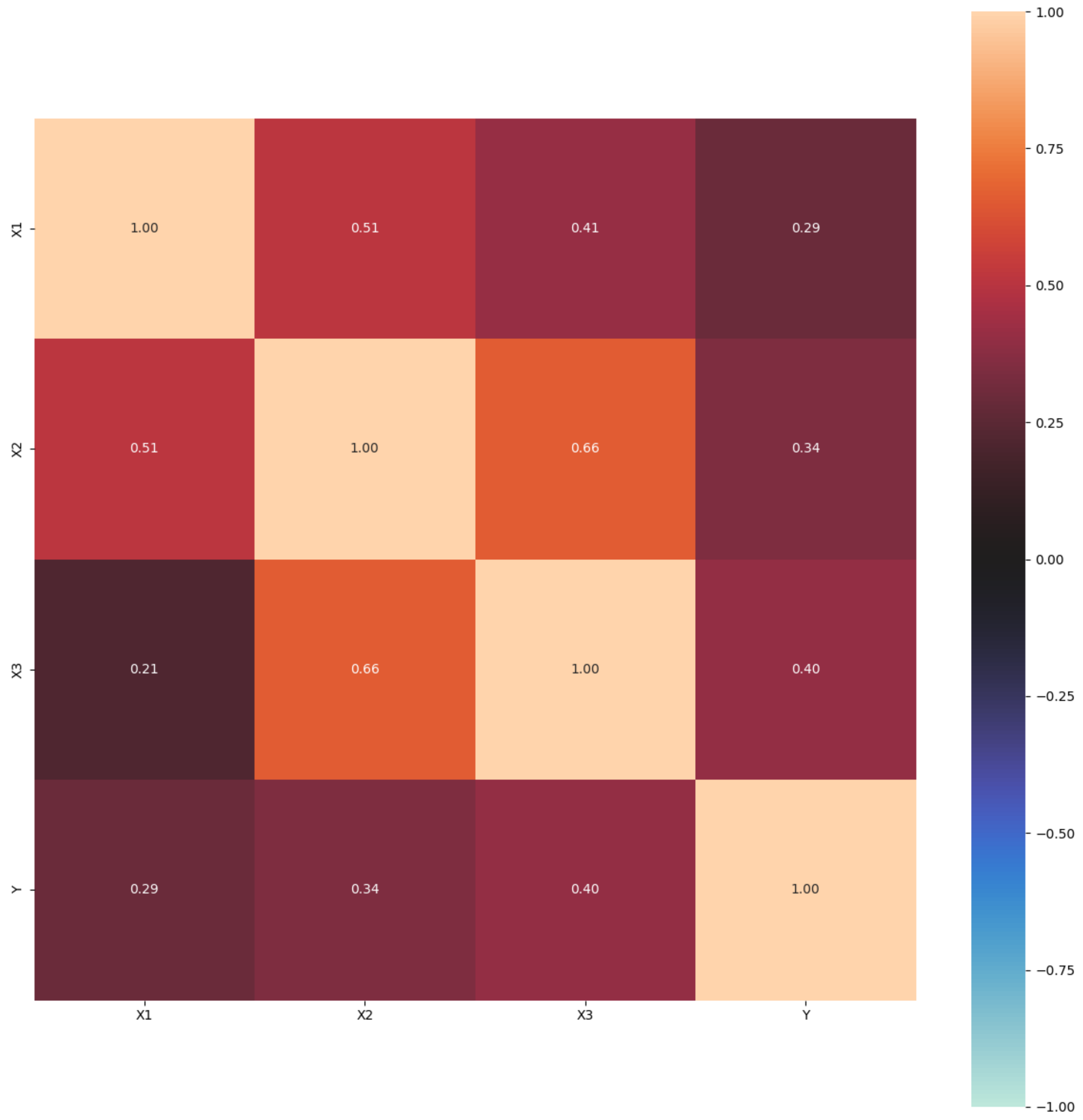


图 1：玻璃相关性质之间的不确定系数

由热力图可以看出，玻璃纹饰、玻璃种类和玻璃颜色都对玻璃最终的表面风化情况呈正相关，其中，玻璃颜色与表面风化情况的相关性最高，玻璃种类次之，玻璃纹饰的相关性在三者之中最小。

但是，三者的相关性均小于 0.5，属于低度线性相关。

5.2.2 文物风化化学成分含量的规律总结

文物样品的风化程度系数显然并非是连续的，因为其值是定序的，且只有 0,1,2 三个数据，是以想要量化地分析文物样品表面有无风化化学成分含量的统计规律是存在困难的。

5.2.2.1 斯皮尔曼相关系数

基于此，为了分析化学成分与风化情况的关系，我们引入斯皮尔曼相关系数[9]，这是一种

基于皮尔逊系数的相关系数，它在使用时会将同一检测点两项数据在总体中的大小排名来代替数据本身，常用于比较非正态分布下数据的相关性。即，斯皮尔曼相关系数是用来分析定序数据的相关系数。而在本文当中，我们将各监测点的各化学成分数据进行排名，并挨个分析各成分与风化严重系数的相关性，从而得出文物样品表面有无风化化学成分含量的统计规律。

斯皮尔曼相关系数的计算公式为：

$$\rho = \rho_{\mathrm{R}(X),\mathrm{R}(Y)} = \frac{\mathrm{COV}(\mathrm{R}(X),\mathrm{R}(Y))}{\sigma_{\mathrm{R}(X)}\sigma_{\mathrm{R}(Y)}} \tag{7}$$

其中，$R(X)$代表X的位次，这里代表含量从多到少的排名，COV是协方差，σ是标准差。显然斯皮尔曼相关系数可以在方差公式的基础上进一步简化，即：

$$\rho = 1 - \frac{6\sum d_i^2}{n(n^2-1)} \tag{8}$$

其中，d_i表示同一检测点两项数据的位次之差，n 表示样本总量，即检测点的总数量。

5.2.2.1 化学成分相关性分析

下图中给出了二氧化硅、氧化钾和氧化铅的频率分布直方图，可见其分布规律并不完全呈正态分布，甚至有的近似呈指数分布：

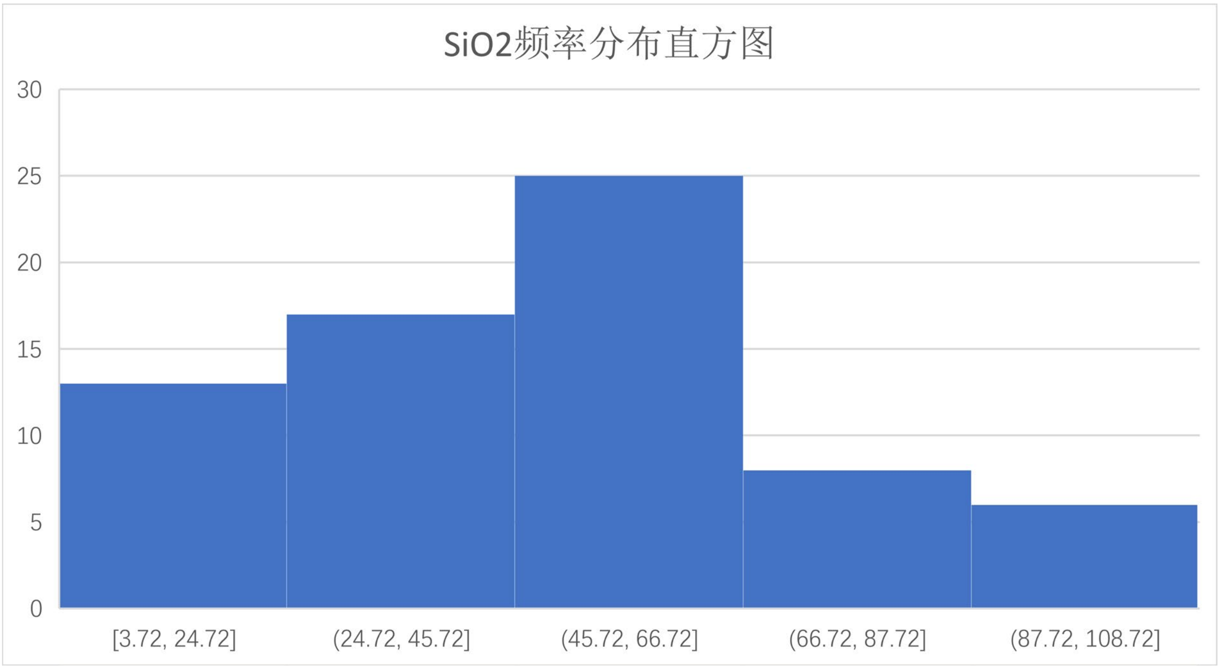


图 2：SiO_2的频率分布直方图

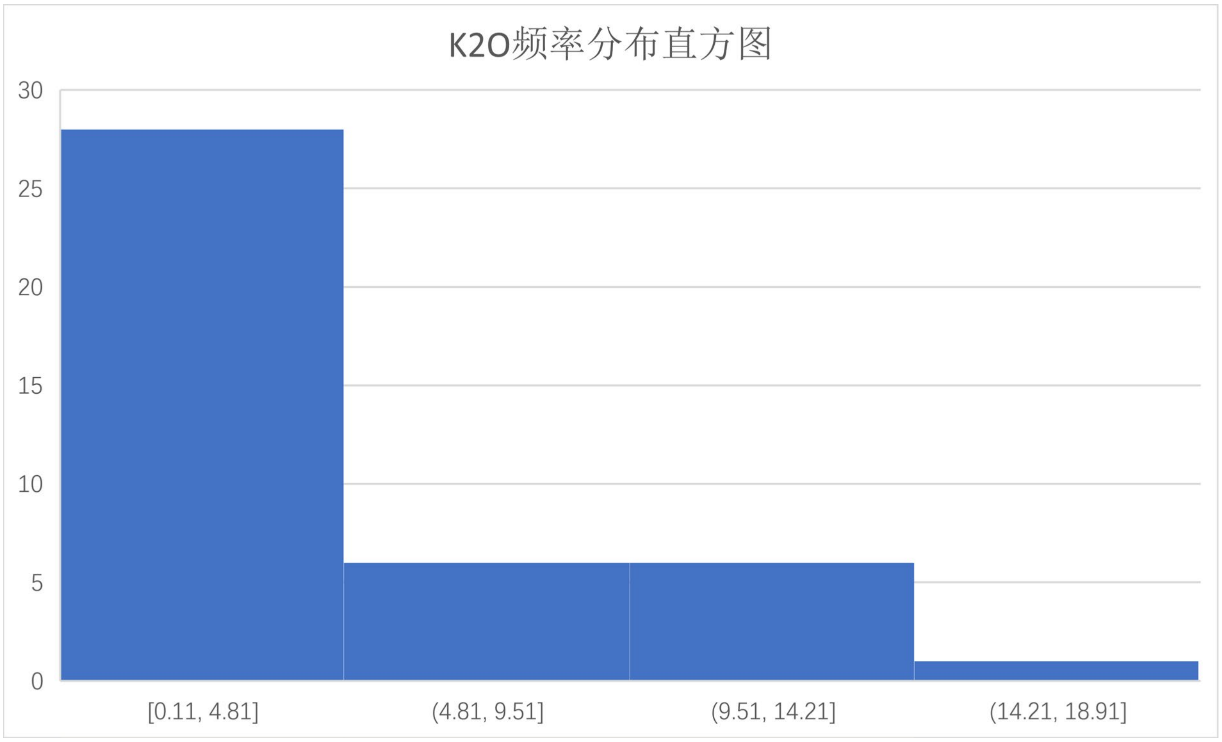


图 3：K_2O的频率分布直方图

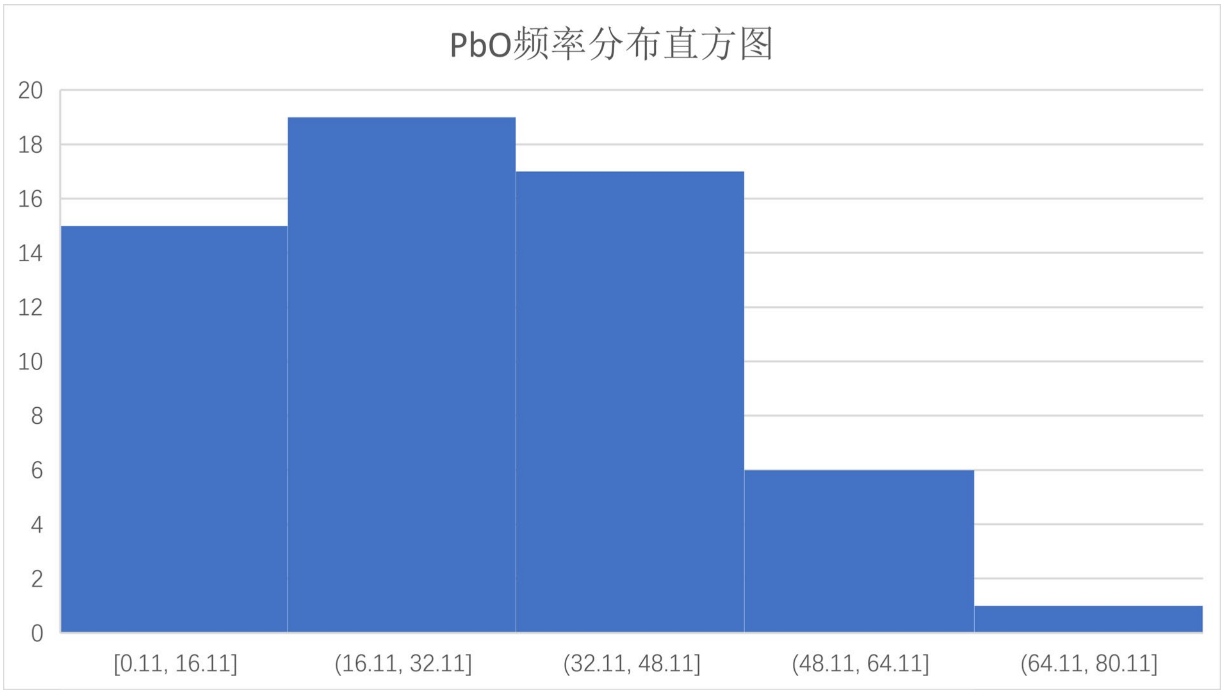


图 4：PbO的频率分布直方图

因此，在化学成分的相关性分析中，斯皮尔曼系数同样适用。

5.2.2.2 有无风化对化学成分含量的统计规律

根据数据类别之间两两进行斯皮尔曼系数的计算，我们最终得出了数据之间斯皮尔曼系数的热力图如下：

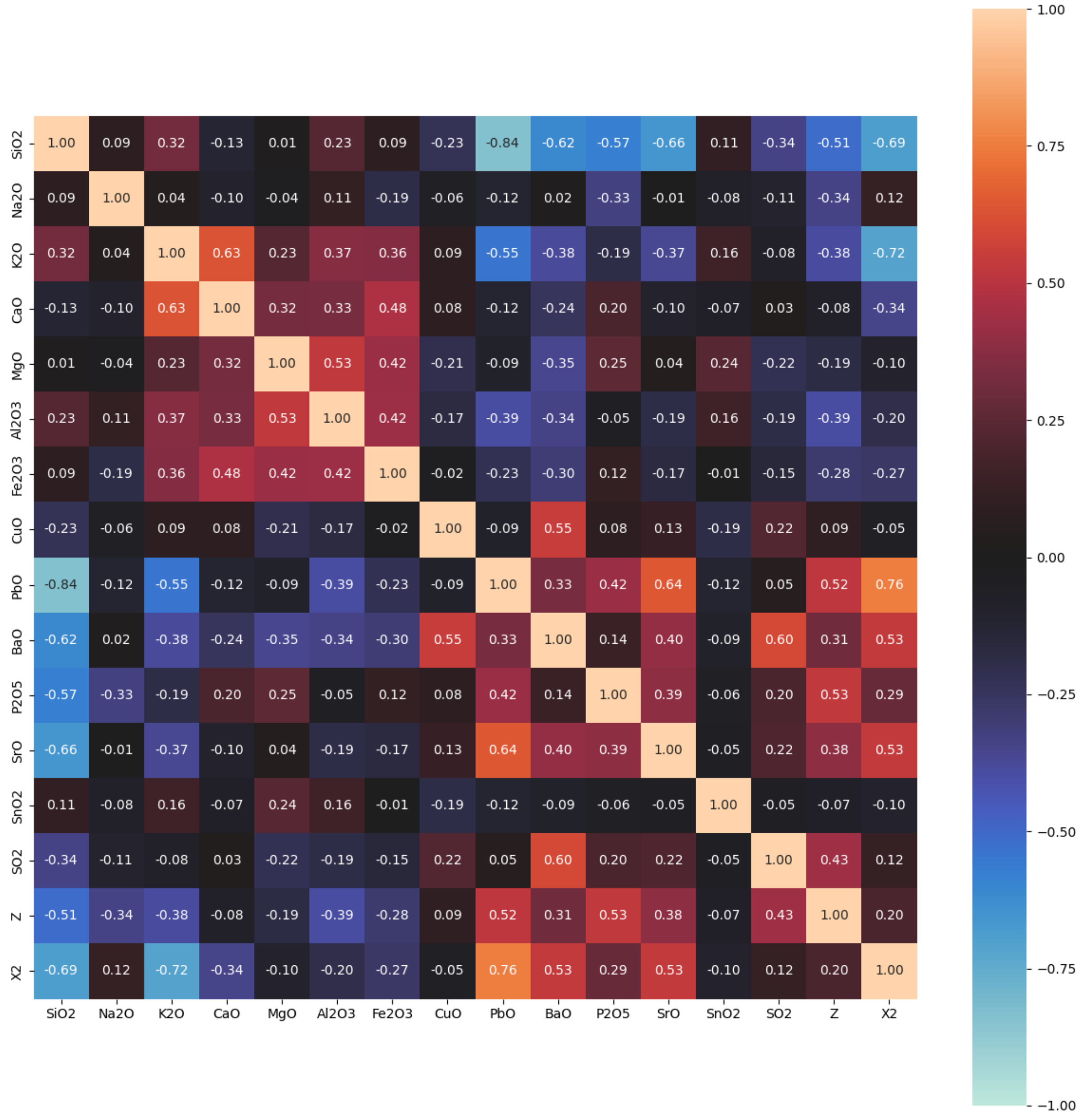


图 5：样品各化学成分与玻璃种类、风化严重系数之间斯皮尔曼系数的热力图

图 5 是排除无效数据后所有检测点总体的斯皮尔曼系数热力图，从图中可以看出，铅、钡硫、锶、磷对于风化的严重程度存在明显的正相关，其中尤其以铅与磷的影响显著。而钾的含量对于风化的严重程度呈明显的负相关。诸多化学元素之间，钙与钾之间存在较高的相关性、铝与镁存在较高的相关性、钡与铜、钡与硫、铅与锶均有较高的相关性。

但是，由于该结果是从整体的角度出发进行计算的，并没有对玻璃进行分类对待，该结果只能作为一个参考标志，并不能作为显著结论。

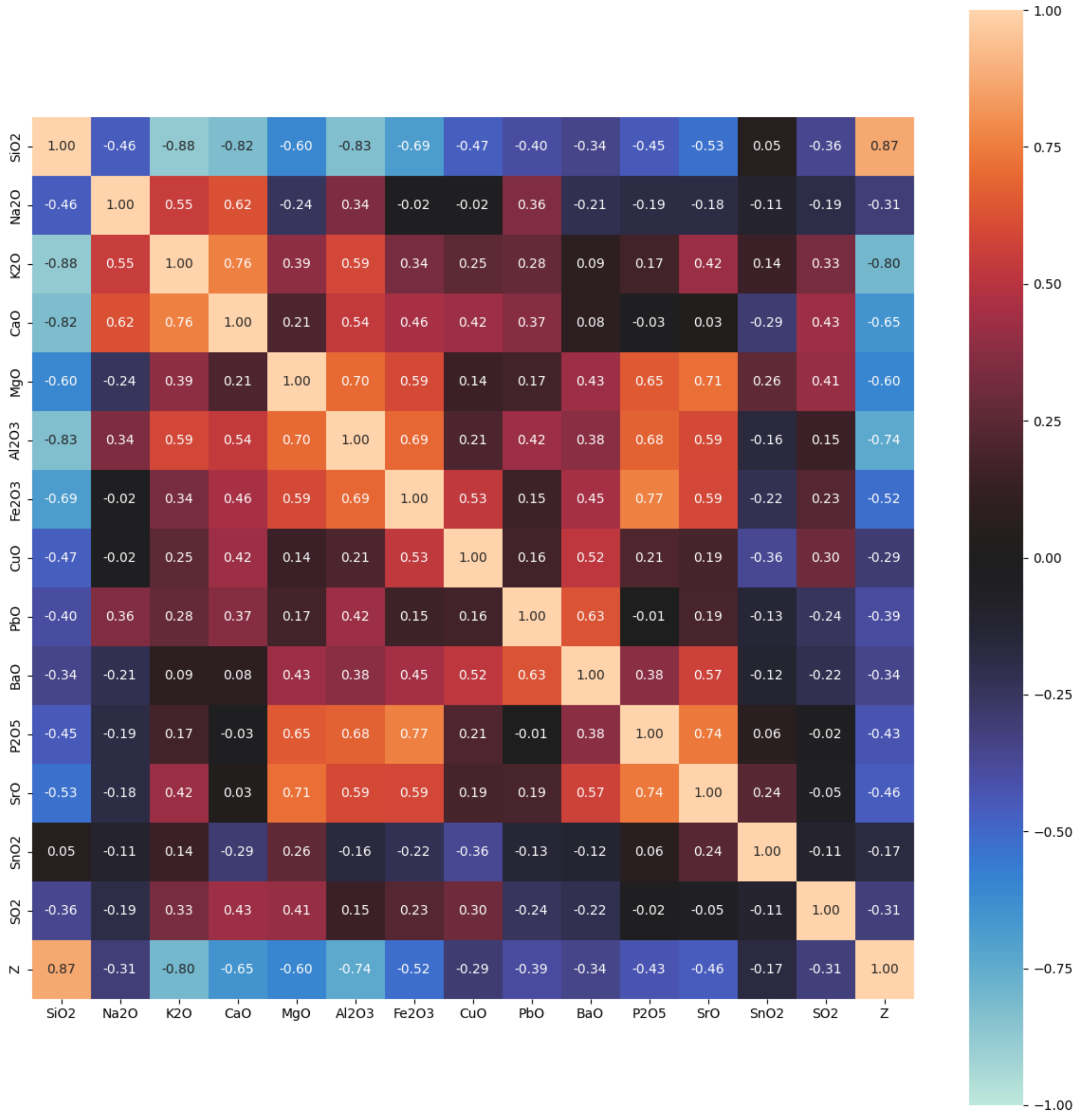
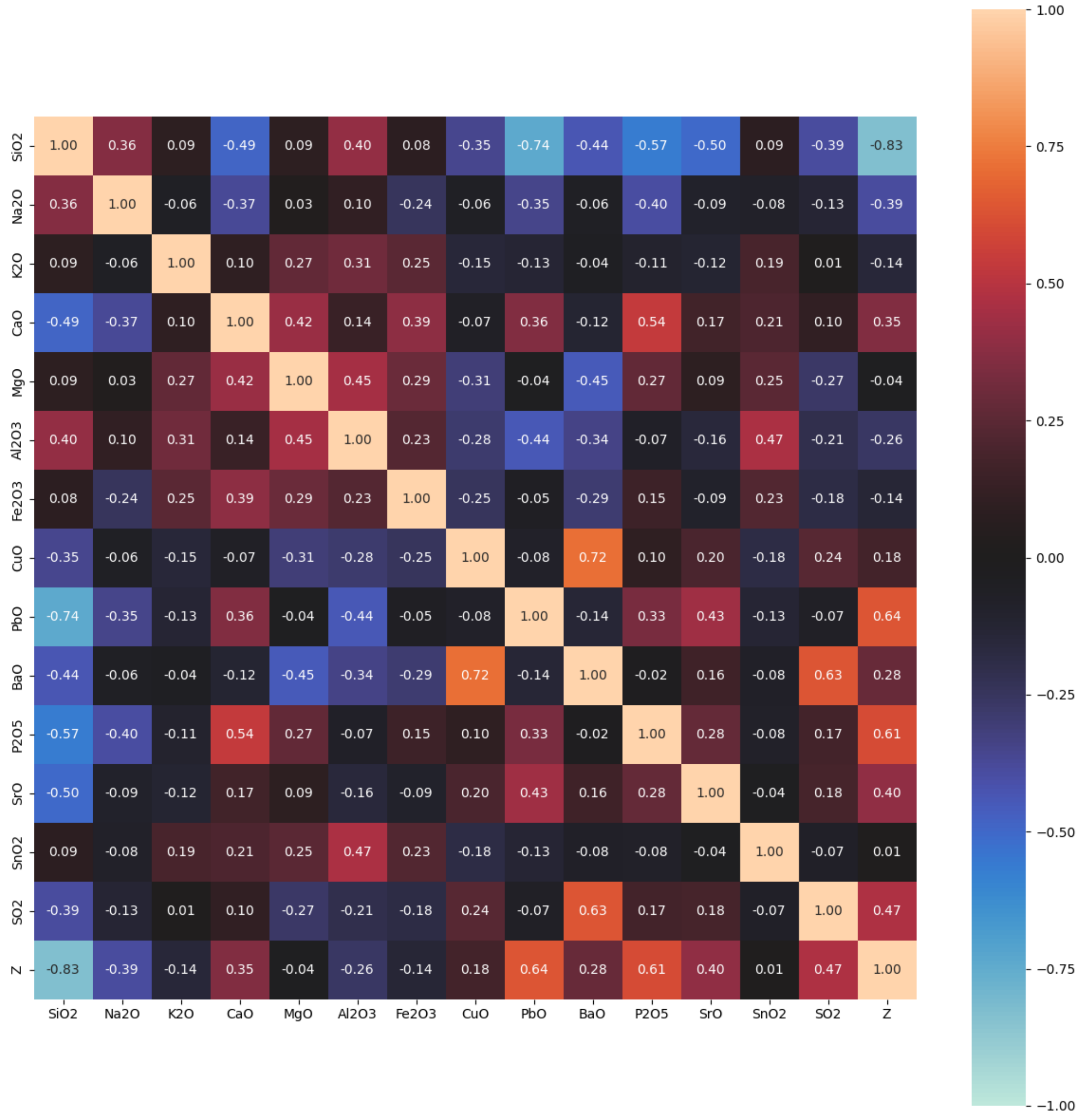


图 6：高钾玻璃各化学成分与风化严重系数之间斯皮尔曼系数的热力图

图 6 是针对高钾玻璃的斯皮尔曼系数，图中硅对玻璃风化情况形成了明显的正相关，但硅本身即是玻璃的主要载体，其成分与风化的关系只能说明风化流失的元素是否是硅，而不能表示硅对玻璃风化的影响。同时还可以看出，钾、铝、钙、镁对于玻璃的风化情况成显著的负相关，而这几种元素之间也存在较高的相关性。

图中还可以看出，在高钾玻璃当中，钾与钠存在较高的相关性，磷、锶同镁、铝、铁、钡、铅也有较高的相关性。



	SiO2	Na2O	K2O	CaO	MgO	Al2O3	Fe2O3	CuO	PbO	BaO	P2O5	SrO	SnO2	SO2	Z
SiO2	1.00	0.36	0.09	-0.49	0.09	0.40	0.08	-0.35	-0.74	-0.44	-0.57	-0.50	0.09	-0.39	-0.83
Na2O	0.36	1.00	-0.06	-0.37	0.03	0.10	-0.24	-0.06	-0.35	-0.06	-0.40	-0.09	-0.08	-0.13	-0.39
K2O	0.09	-0.06	1.00	0.10	0.27	0.31	0.25	-0.15	-0.13	-0.04	-0.11	-0.12	0.19	0.01	-0.14
CaO	-0.49	-0.37	0.10	1.00	0.42	0.14	0.39	-0.07	0.36	-0.12	0.54	0.17	0.21	0.10	0.35
MgO	0.09	0.03	0.27	0.42	1.00	0.45	0.29	-0.31	-0.04	-0.45	0.27	0.09	0.25	-0.27	-0.04
Al2O3	0.40	0.10	0.31	0.14	0.45	1.00	0.23	-0.28	-0.44	-0.34	-0.07	-0.16	0.47	-0.21	-0.26
Fe2O3	0.08	-0.24	0.25	0.39	0.29	0.23	1.00	-0.25	-0.05	-0.29	0.15	-0.09	0.23	-0.18	-0.14
CuO	-0.35	-0.06	-0.15	-0.07	-0.31	-0.28	-0.25	1.00	-0.08	0.72	0.10	0.20	-0.18	0.24	0.18
PbO	-0.74	-0.35	-0.13	0.36	-0.04	-0.44	-0.05	-0.08	1.00	-0.14	0.33	0.43	-0.13	-0.07	0.64
BaO	-0.44	-0.06	-0.04	-0.12	-0.45	-0.34	-0.29	0.72	-0.14	1.00	-0.02	0.16	-0.08	0.63	0.28
P2O5	-0.57	-0.40	-0.11	0.54	0.27	-0.07	0.15	0.10	0.33	-0.02	1.00	0.28	-0.08	0.17	0.61
SrO	-0.50	-0.09	-0.12	0.17	0.09	-0.16	-0.09	0.20	0.43	0.16	0.28	1.00	-0.04	0.18	0.40
SnO2	0.09	-0.08	0.19	0.21	0.25	0.47	0.23	-0.18	-0.13	-0.08	-0.08	-0.04	1.00	-0.07	0.01
SO2	-0.39	-0.13	0.01	0.10	-0.27	-0.21	-0.18	0.24	-0.07	0.63	0.17	0.18	-0.07	1.00	0.47
Z	-0.83	-0.39	-0.14	0.35	-0.04	-0.26	-0.14	0.18	0.64	0.28	0.61	0.40	0.01	0.47	1.00

图 7：铅钡玻璃各化学成分与风化严重系数之间斯皮尔曼系数的热力图

由图七可以看出，铅钡玻璃中，铅、磷、硫对风化的严重程度呈正相关，铅钙之间、硫钡之间、铜钡之间、磷钙之间、镁铝之间、镁钙之间存在较高的相关性。相比于高钾玻璃，铅钡玻璃中的大多数元素之间互不相干。

这一点似乎可以用两种玻璃的制作与案例来解释：高钾玻璃的辅料是草木灰，是有机物焚烧后的产物，而铅钡玻璃的辅料则是矿石，是以高钾玻璃中各元素的成分会更复杂，各元素之间也会由于植物生长的需求而表现出一定的相关性，而矿石的成分复杂性相比于草木灰则要低上许多，成分之间的相关性主要来源于伴生矿[18]。

5.2.3 基于风化点数据的多元线性回归

风化往往是不可逆的过程，而风化的过程中也往往受多因素的影响[5]，是以，文物风化后的相关数据其实是受多元自变量影响的因变量，即：

$$y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots \beta_n X_n + e \tag{9}$$

其中，y 是因变量，在这里是风化后数据，X_i是风化前数据。

我们可以通过微分方程和最小二乘法对该回归进行求解。

5.2.3.1 逻辑斯蒂回归

结果保存在支撑文件 fenghua_result.CSV 中

5.2.4玻璃种类划分的聚类模型

5.2.4.1 K-Means 聚类

K-Means 聚类是一种常用的无监督机器学习方法，它通过机器反复迭代运算对数据进行分类，并使得最终得出的结果在同类之间相似度较高。应该承认，不管使用怎样的分类方法，总会不可避免地使得样本数据特点出现一定的损失，而 K-Means 聚类为此定义了一个损失函数：

$$J(c,\mu) = \sum_{i=1}^{M} \left\| x_i - \mu_{c_i} \right\| \tag{10}$$

其中，x_i代表第i个样本，c_i是样本所属的类别，μ_{c_i}是类别的中心均值，M则是样本总数。

K-Means 聚类需要人工预设一个类别总量 k，而后由程序不断地确定 k 个中心均值μ_{c_i}，而后将样本分为 k 个小类，再通过损失函数来判断是否需要调整中心均值、各样本的类别是否需要变换的交替循环过程，直到机器找到了使得损失函数J最小的分类方法为止。

5.2.4.2 主成分分析法

主成分分析法（PCA）是一种数据降维方案，它可以将数据从高维向低维进行映射，从而筛选掉不那么重要的数据，留下差异性较大的“主成分”，以便于模型的分析的推广。附件中给出的化学成分数据多达 14 个维度，在聚类过程中进行适当的筛选是十分有必要的。

我们以数据种类为列，数据来源为行，可以通过附件表格二轻易构建出一个 67 行 14 列的矩阵，通过矩阵可以计算出它的协方差：

$$cov(X,Y) = E(X-\mu_X)(Y-\mu_Y) \tag{11}$$

而当样本是 n 维数据时，得到的将不再是协方差，而是协方差矩阵：

$$Cov(X,Y,Z) = \begin{bmatrix} Cov(x,x) & Cov(x,y) & Cov(x,z) \\ Cov(y,x) & Cov(y,y) & Cov(y,z) \\ Cov(z,x) & Cov(z,y) & Cov(z,z) \end{bmatrix} \tag{12}$$

对于协方差矩阵来说，它与矩阵的散度矩阵存在代数关系，这可以方便计算：

$$Cov = \frac{1}{n-1}AA^T \tag{13}$$

根据线性代数，不难得出协方差矩阵的特征值与对应的特征向量，再将特征值由大到小排列，其对应的特征向量也相应排列组合成特征向量矩阵P，将特征向量矩阵与原矩阵相乘，即可得到降维后的向量矩阵：

$$B = PA \tag{14}$$

降维后的矩阵将大大减小聚类的计算量，有助于聚类的进行。

5.2.4.3 聚类结果

将数据矩阵进行 PCA 后，所得特征矩阵保存在支撑材料的 pca_base_data.CSV 中，而后我们将之进行聚类。

在聚类过程中我们需要凭借经验确定 k 的值，但我们可以提前通过模拟其统计量来选取 k 的值：

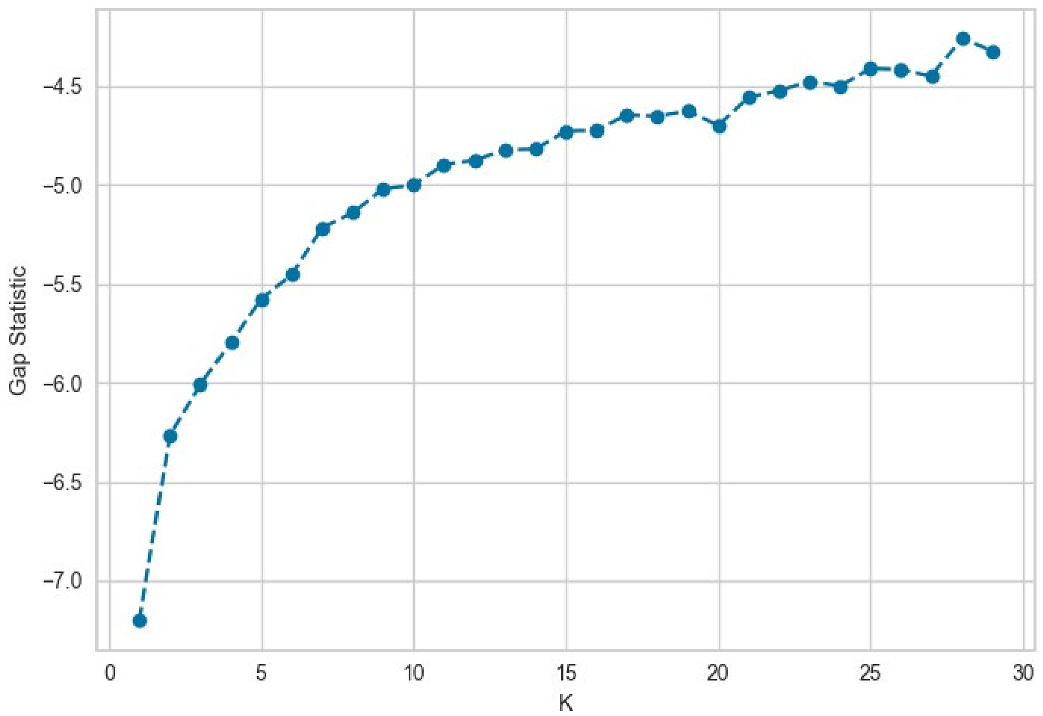
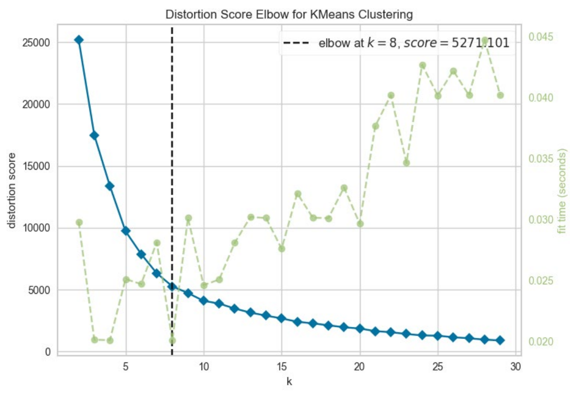


图 8：k 值大小与统计量的关系

由图片可以看出：当 k>8 时，已经不会对统计量有明显增益，故这里取 k=8。



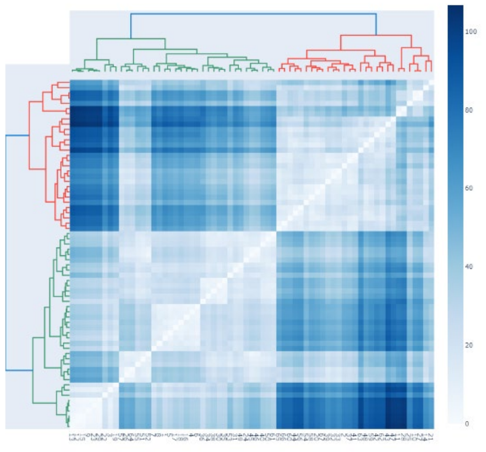


图 9：聚类结果

聚类结果保存在 cluster_result_all.CSV 中。

由聚类结果本文可以继续细分玻璃类型：高铜玻璃（编号 0）、高铝钾玻璃（编号 1）、铝磷高钾玻璃（编号 2）、高锶铅钡玻璃（编号 3）、铝镁铅钡玻璃（编号 4）、低硫铅钡玻璃（编号 5）、钙镁铅钡玻璃（编号 6）、风化高钾玻璃（编号 7，即含钾较少的高钾玻璃）。其中，高铜玻璃在高钾玻璃、铅钡玻璃中均有分布，故一共可分为 9 类。

6. 模型评价

在本文的分析模型中，关于最核心的部分——即聚类部分，是通过机器学习反复迭代而得到的结果，数据结果在统计学意义上具有一定的稳定性，但是，一旦出现了与附件所给数据差异较大的玻璃制品时，聚类结果可能不理想。另外，在本模型的建立中，我们假设了玻璃风化的过程不会反作用于玻璃化学成分的变化，但是在数据分析中，二氧化硅与风化程度的关系出现了不正常的相关性，这很有可能是因为二氧化硅作为玻璃的主要载体，玻璃的风化对其成分流失存在影响。

参考文献

1. 于国成. 也谈中国古代玻璃的起源. 玻璃与搪瓷. 1987;4.
2. Fuxi G. The Glass and Jade Road—the Cultural and Technical Exchange of Silicate Based Artefacts with Foreign Countries Before Qin Dynasty. *J Chinese Ceram Soc*. 2013;41(4):458-466.
3. Melcher M, Schreiner M. Statistical evaluation of potash-lime-silica glass weathering. *Anal Bioanal Chem*. 2004;379(4):628-639.
4. Fuxi G. The silk road and ancient Chinese glass. *Anc Glas Res along Silk Road, Singapore*. Published online 2009:41-108.
5. 薛吕. 玻璃文物保护与修复. 中国文物保护技术协会第七次学术年会论文集. Published online 2012.
6. Stevens SS. On the theory of scales of measurement. *Science (80-)*. 1946;103(2684):677-680.
7. Janse RJ, Hoekstra T, Jager KJ, et al. Conducting correlation analysis: important limitations and pitfalls. *Clin Kidney J*. 2021;14(11):2332-2337. doi:10.1093/ckj/sfab085
8. Kozak M. What is strong correlation? *Teach Stat*. 2009;31(3):85-86.
9. Myers JL, Well AD, Lorch RF. *Research Design and Statistical Analysis*. Routledge; 2013.
10. Schober P, Boer C, Schwarte LA. Correlation coefficients: appropriate use and interpretation. *Anesth Analg*. 2018;126(5):1763-1768.
11. Cramer H. Mathematical methods of statistics. Princeton: Princeton Univer. Published online 1946.
12. Cohen J. *Statistical Power Analysis for the Behavioral Sciences*. Routledge; 2013.
13. Smith SM, Albaum GS. *Fundamentals of Marketing Research*. Sage; 2005.
14. Shannon CE. A mathematical theory of communication. *ACM SIGMOBILE Mob Comput Commun Rev*. 2001;5(1):3-55.
15. Press WH, Teukolsky SA, Vetterling WT, Flannery BP. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge university press; 2007.
16. White J V, Steingold S, Fournelle CG. Performance metrics for group-detection algorithms. *Proc Interface*. 2004;2004.
17. Mills P. Efficient statistical classification of satellite measurements. *Int J Remote Sens*. 2011;32(21):6109-6132.
18. 干福熹. 中国古代玻璃的起源和发展. 自然杂志. 2006;28(4):187-193.

附录

聚类：

main：

```
from pathlib import Path
import pandas as pd
from phik.binning import bin_data
from phik.report import plot_correlation_matrix
from scipy.sparse import data
from scipy.cluster.hierarchy import linkage, dendrogram
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.graph_objects as go
import plotly.figure_factory as ff
import numpy as np
from scipy.spatial.distance import pdist, squareform

from clustering.dendrogram_plot import dend_plot

if __name__ == '__main__':
    # create the file handles for each list
    path_root = str(Path.cwd())
    data_file_root = path_root.rstrip("clustering") + "raw_data_source"
    data_handle_1 = data_file_root + "\\raw_data_form_1.CSV"
    data_handle_2 = data_file_root + "\\raw_data_form_2_preprocessed.CSV"
    data_handle_2_X2_0 = data_file_root + "\\pre_processed_X2_0.CSV"
    data_handle_2_X2_1 = data_file_root + "\\pre_processed_X2_1.CSV"
    data_handle_2_color_append = data_file_root +
"\\raw_data_form_2_preprocessed_color_append.CSV"
    data_handle_3 = data_file_root + "\\raw_data_form_3.CSV"
    data_handle_2_pca_processed_data = data_file_root + "\\pca_processed_form_2.CSV"
```

```
            data_handle_pca_processed_data_form_3 = data_file_root +
"\\pca_processed_form_3.CSV"
            data_handle_2_X2_0_pca = data_file_root + "/pca_processed_X2_0.CSV"
            data_handle_2_X2_1_pca = data_file_root + "/pca_processed_X2_1.CSV"


            # identify the columns
            columns = ['PC1', 'PC2', 'PC3', 'PC4', 'PC5', 'PC6', 'PC7', 'PC8', 'PC9', 'PC10']
            columns_1 = ["index"]
            columns_2 = ["Number"]


            # read in the data files
            df_2_pca_processed = pd.read_csv(data_handle_2_pca_processed_data, header=0,
encoding='ISO-8859-1',
                                             usecols=columns, )
            df_2_pca_processed = df_2_pca_processed.fillna(0)


            df_2_X2_0_pca_processed = pd.read_csv(data_handle_2_X2_0_pca, header=0,
encoding='ISO-8859-1',
                                                  usecols=columns, )
            df_2_X2_0_pca_processed = df_2_X2_0_pca_processed.fillna(0)


            df_2_X2_1_pca_processed = pd.read_csv(data_handle_2_X2_1_pca, header=0,
encoding='ISO-8859-1',
                                                  usecols=columns, )
            df_2_X2_1_pca_processed = df_2_X2_1_pca_processed.fillna(0)


            df_3_pca_processed = pd.read_csv(data_handle_pca_processed_data_form_3, header=0,
encoding='ISO-8859-1',
                                             usecols=columns, )
            df_3_pca_processed = df_3_pca_processed.fillna(0)
```

```
        df_only_index_X2_0 = pd.read_csv(data_handle_2_X2_0, header=0, encoding='ISO-8859-
1', sep="\t",
                                          usecols=columns_1, )
        df_only_index_X2_0 = df_only_index_X2_0.fillna(0)


        df_only_index_X2_1 = pd.read_csv(data_handle_2_X2_1, header=0, encoding='ISO-8859-
1', sep="\t",
                                          usecols=columns_1, )
        df_only_index_X2_1 = df_only_index_X2_1.fillna(0)


        # get the data within the data files
        samples_2_total = df_2_pca_processed.iloc[:, :].values


        samples_2_X2_0 = df_2_X2_0_pca_processed.iloc[:, :].values


        samples_2_X2_1 = df_2_X2_1_pca_processed.iloc[:, :].values


        samples_2_total


        labels_2_total = []
        for i in range(len(df_2_pca_processed)):
            labels_2_total.append(i + 1)


        labels_2_X2_0 = df_only_index_X2_0.iloc[:, :].values


        labels_2_X2_1 = df_only_index_X2_1.iloc[:, :].values




        #dend_plot(samples_2_total, labels_2_total, path_root, "fig_total")
        #dend_plot(samples_2_X2_0, labels_2_X2_0, path_root, "fig_X2_0")
        #dend_plot(samples_2_X2_1, labels_2_X2_1, path_root, "fig_X2_1")
    dend_plot()
```

dendrogram_plot：

```
        from pathlib import Path
    import pandas as pd
    from phik.binning import bin_data
    from phik.report import plot_correlation_matrix
    from scipy.sparse import data
    from scipy.cluster.hierarchy import linkage, dendrogram
    import matplotlib.pyplot as plt
    import seaborn as sns
    import plotly.graph_objects as go
    import plotly.figure_factory as ff
    import numpy as np
    from scipy.spatial.distance import pdist, squareform



    def dend_plot(data_array, labels, path_root, fig_name):
        fig = ff.create_dendrogram(data_array, orientation='bottom', labels=labels)
        for i in range(len(fig['data'])):
            fig['data'][i]['yaxis'] = 'y2'

        # Create Side Dendrogram
        dendro_side = ff.create_dendrogram(data_array, orientation='right')
        for i in range(len(dendro_side['data'])):
            dendro_side['data'][i]['xaxis'] = 'x2'

        # Add Side Dendrogram Data to Figure
```

```
for data in dendro_side['data']:
    fig.add_trace(data)

# Create Heatmap
dendro_leaves = dendro_side['layout']['yaxis']['ticktext']
dendro_leaves = list(map(int, dendro_leaves))
data_dist = pdist(data_array)
heat_data = squareform(data_dist)
heat_data = heat_data[dendro_leaves, :]
heat_data = heat_data[:, dendro_leaves]

heatmap = [
    go.Heatmap(
        x=dendro_leaves,
        y=dendro_leaves,
        z=heat_data,
        colorscale='Blues'
    )
]

heatmap[0]['x'] = fig['layout']['xaxis']['tickvals']
heatmap[0]['y'] = dendro_side['layout']['yaxis']['tickvals']

# Add Heatmap Data to Figure
for data in heatmap:
    fig.add_trace(data)

# Edit Layout
fig.update_layout({'width': 1000, 'height': 1000,
                         'showlegend': False, 'hovermode': 'closest',
                    })
# Edit xaxis
fig.update_layout(xaxis={'domain': [.15, 1],
                        'mirror': False,
                        'showgrid': False,
                        'showline': False,
                        'zeroline': False,
                        'ticks': ""})
# Edit xaxis2
fig.update_layout(xaxis2={'domain': [0, .15],
                         'mirror': False,
                         'showgrid': False,
                         'showline': False,
                         'zeroline': False,
                         'showticklabels': False,
                         'ticks': ""})

# Edit yaxis
fig.update_layout(yaxis={'domain': [0, .85],
                        'mirror': False,
                        'showgrid': False,
                        'showline': False,
                        'zeroline': False,
                        'showticklabels': False,
                        'ticks': ""
                        })
# Edit yaxis2
fig.update_layout(yaxis2={'domain': [.825, .975],
                         'mirror': False,
                         'showgrid': False,
                                    'showline': False,
                                    'zeroline': False,
                                    'showticklabels': False,
                                    'ticks': ""})

    # Plot!
    fig.show()
#fig.write_image(path_root + "\\" + fig_name + ".png")
```

相关分析：

main：

```
    from pathlib import Path
import pandas as pd
from phik.binning import bin_data
from phik.report import plot_correlation_matrix
from scipy.sparse import data



from correlation_analysis.uncertainty_correlation import *

if __name__ == '__main__':
    # create the file handles for each list
    path_root = str(Path.cwd())
    data_file_root = path_root.rstrip("correlation_analysis") + "raw_data_source"
    data_handle_1 = data_file_root + "\\raw_data_form_1.CSV"
    data_handle_2 = data_file_root + "\\raw_data_form_2_preprocessed.CSV"
    data_handle_2_X2_0 = data_file_root + "\\pre_processed_X2_0.CSV"
    data_handle_2_X2_1 = data_file_root + "\\pre_processed_X2_1.CSV"
    data_handle_2_color_append = data_file_root +
"\\raw_data_form_2_preprocessed_color_append.CSV"
            data_handle_3 = data_file_root + "\\raw_data_form_3.CSV"

            df_1 = pd.read_csv(data_handle_1, header=0, encoding='ISO-8859-1', usecols=["X1", "X2",
"X3", "Y"], )

            df_1.columns = [
                "X1",
                "X2",
                "X3",
                "Y",
            ]

            df_2 = pd.read_csv(data_handle_2, header=0, encoding='ISO-8859-1', usecols=['SiO2',
'Na2O', 'K2O', 'CaO',

'MgO', 'Al2O3', 'Fe2O3', 'CuO', 'PbO',

'BaO', 'P2O5', 'SrO', 'SnO2', 'SO2',

'Z', 'X2'], )

            df_2.columns = [
                'SiO2', 'Na2O', 'K2O', 'CaO', 'MgO', 'Al2O3',
                'Fe2O3', 'CuO', 'PbO', 'BaO', 'P2O5', 'SrO', 'SnO2', 'SO2', 'Z', 'X2'
            ]

            df_2_X2_0 = pd.read_csv(data_handle_2_X2_0, header=0, encoding='ISO-8859-1',
usecols=['SiO2', 'Na2O', 'K2O', 'CaO',

'MgO', 'Al2O3', 'Fe2O3',
'CuO', 'PbO',

'BaO', 'P2O5', 'SrO', 'SnO2',

'SO2',

'Z', ], )

            df_2_X2_0.columns = [
                'SiO2',
                'Na2O',
                'K2O',
                'CaO',
                'MgO',
                'Al2O3',
                'Fe2O3',
                'CuO',
                'PbO',
                'BaO',
                'P2O5',
                'SrO',
                'SnO2',
                'SO2',
                'Z',
            ]

            df_2_X2_1 = pd.read_csv(data_handle_2_X2_1, header=0, encoding='ISO-8859-1',
                                    usecols=['SiO2', 'Na2O', 'K2O', 'CaO',
                                             'MgO', 'Al2O3', 'Fe2O3', 'CuO', 'PbO',
                                             'BaO', 'P2O5', 'SrO', 'SnO2', 'SO2',
                                             'Z', ], )
            df_2_X2_1.columns = df_2_X2_0.columns


            df_2_color_append = pd.read_csv(data_handle_2_color_append, header=0, encoding='ISO-
8859-1',
                                                usecols=['SiO2', 'Na2O', 'K2O', 'CaO',
                                                         'MgO', 'Al2O3', 'Fe2O3', 'CuO', 'PbO',
                                                         'BaO', 'P2O5', 'SrO', 'SnO2', 'SO2',
                                                         'color', ], )
            df_2_color_append.columns = ['SiO2', 'Na2O', 'K2O', 'CaO', 'MgO', 'Al2O3', 'Fe2O3', 'CuO',
'PbO',
                                            'BaO', 'P2O5', 'SrO', 'SnO2', 'SO2', 'color', ]


            df_2_Z = pd.read_csv(data_handle_2, header=0, encoding='ISO-8859-1', usecols=['SiO2',
'Na2O', 'K2O', 'CaO',

'MgO', 'Al2O3', 'Fe2O3', 'CuO', 'PbO',

'BaO', 'P2O5', 'SrO', 'SnO2', 'SO2',

'Z'], )


            df_2_Z.columns = [
                'SiO2', 'Na2O', 'K2O', 'CaO', 'MgO', 'Al2O3',
                'Fe2O3', 'CuO', 'PbO', 'BaO', 'P2O5', 'SrO', 'SnO2', 'SO2', 'Z'
            ]


            df_2_X2_0_only_components = pd.read_csv(data_handle_2_X2_0, header=0,
encoding='ISO-8859-1',
                                                        usecols=['SiO2', 'Na2O', 'K2O', 'CaO',
                                                                 'MgO', 'Al2O3', 'Fe2O3', 'CuO',
'PbO',
                                                    'BaO', 'P2O5', 'SrO', 'SnO2',
'SO2', ], )
        df_2_X2_0_only_components.columns = [
            'SiO2', 'Na2O', 'K2O', 'CaO', 'MgO', 'Al2O3',
            'Fe2O3', 'CuO', 'PbO', 'BaO', 'P2O5', 'SrO', 'SnO2', 'SO2'
        ]

        df_2_X2_1_only_components = pd.read_csv(data_handle_2_X2_1, header=0,
encoding='ISO-8859-1',
                                              usecols=['SiO2', 'Na2O', 'K2O', 'CaO',
                                                    'MgO', 'Al2O3', 'Fe2O3', 'CuO',
'PbO',
                                                    'BaO', 'P2O5', 'SrO', 'SnO2',
'SO2', ], )
        df_2_X2_1_only_components.columns = [
            'SiO2', 'Na2O', 'K2O', 'CaO', 'MgO', 'Al2O3',
            'Fe2O3', 'CuO', 'PbO', 'BaO', 'P2O5', 'SrO', 'SnO2', 'SO2'
        ]

        # calculate the theil_u coefficient of data file 1
        association_computation_main(dataset=df_1, nominal_nominal_assocication="theil",
plot_save_filename="figure_1",
                                     figure_size=(15, 15), heatmap_vmin=0)
        association_computation_main(dataset=df_2,
nominal_nominal_assocication="spearman_rank_correlation",
                                     plot_save_filename="figure_2",
                                     figure_size=(15, 15), heatmap_vmin=0)
        association_computation_main(dataset=df_2_X2_0,
nominal_nominal_assocication="spearman_rank_correlation",
                                     plot_save_filename="figure_2_X2_0",
                                     figure_size=(15, 15), heatmap_vmin=0)
        association_computation_main(dataset=df_2_X2_1,
nominal_nominal_assocication="spearman_rank_correlation",
                                        plot_save_filename="figure_2_X2_1",
                                        figure_size=(15, 15), heatmap_vmin=0)
        association_computation_main(dataset=df_2_color_append,
nominal_numerical_association="correlation_ratio",
                                        plot_save_filename="figure_3",
                                        figure_size=(15, 15), heatmap_vmin=0)
        association_computation_main(dataset=df_2_X2_0_only_components,
numerical_numerical_association="pearson",
                                        plot_save_filename="figure_4_X2_0",
                                        figure_size=(15, 15), heatmap_vmin=0)
        association_computation_main(dataset=df_2_X2_1_only_components,
numerical_numerical_association="pearson",
                                        plot_save_filename="figure_4_X2_1",
                                        figure_size=(15, 15), heatmap_vmin=0)
        # dataset: NumPy array/Pandas DataFrame
        # nominal_columns: string/list/NumPy ndarray, names of columns of the dataset holding
categorical values
        # numerical_columns: string/list/NumPy ndarray, names of columns of the dataset holding
numerical values
        # mark_columns: Boolean, default = False, if True, output's columns' names will have a
suffix of '(nom)' or '(con)'
        # nominal_nominal_assocication: callable/string, default = 'cramer'
        # options: 'cramer', 'theil'
        # numerical_numerical_association: callable/string, default = 'pearson'
        # options: 'pearson', 'spearman', 'kendall', 'spearman_rank_correlation'
        # nominal_numerical_association: callable/string, default = 'correlation_ratio'
        # options: 'correlation_ratio'
        # symmetric_nominal_nominal: Boolean, default = True, declare whether the function is
symmetric
        # symmetric_numerical_numerical: Boolean, default = True, declare whether the function is
symmetric
```

# display_rows: list/string, default = 'all', choose which of the dataset's features will be displyed in the output's correlations table rows

# display_columns: list/string, default = 'all', choose which of the dataset's features will be displyed in the output's correlations table columns

# hide_rows: list/string, default = None, choose which of the dataset's features will not be displyed in the output's correlations table rows.

# hide_columns: list/string, default = None, choose which of the dataset's features will not be displyed in the output's correlations table column

# cramers_v_bias_correction: Boolean, default = True, use bias correction for Cramer's V from Bergsma and Wicher,

# nan_value_strategy: string, default = 'replace'

# nan_replace_value: any, default = 0.0

# figure_ax: matplotlib figure_ax, default = None, Matplotlib Axis on which the heat-map will be plotted

# figure_size: (int,int) or None, default = None A Matplotlib figure-size tuple. If 'None', falls back to Matplotlib's default. Only used if 'figure_ax=None'.

# plot_annotation: Boolean, default = True, Plot number annotations on the heat-map

# annotation_string_format: string, default = '.2f' String formatting of annotations

# plot_color_map: Matplotlib colormap or None, default = None A colormap to be used for the heat-map. If None, falls back to Seaborn's heat-map default

# sv_color: string, default = 'silver', the color to be used when displaying single-value features over the heat-map

# color_bar_display_flag: Boolean, default = True Display heat-map's color-bar

# heatmap_vmax: float, default = 1.0, set heat-map heatmap_vmax option

# heatmap_vmin: float or None, default = None Set heat-map heatmap_vmin option. If set to None, heatmap_vmin will be chosen automatically between 0 and -1, depending on the types of associations used (-1 if Pearson's R is used, 0 otherwise)

# plot: Boolean, default = True, plot a heat-map of the correlation matrix. If false, the all axes and plotting still happen, but the heat-map will not be displayed.

# compute_only: Boolean, default = False Use this flag only if you have no need of the plotting at all. This skips the entire plotting mechanism.

# clustering: Boolean, default = False If True, hierarchical clustering is applied in order to sort features into meaningful groups

# plot_title: string or None, default = None Plotted graph plot_title

# plot_save_filename: string or None, default = None If not None, plot will be saved to the given file name

# multiprocessing_flag: Boolean, default = False If True, use 'multiprocessing_flag' to speed up computations. If None, falls back to single core computation

# max_cpu_cores: int or None, default = None If not None, ProcessPoolExecutor will use the given number of CPU cores

uncertainty_correlation：

```
import concurrent.futures as cf
import math
import warnings
from collections import Counter
from itertools import repeat

import matplotlib.pyplot as plt
import numpy as np
import numpy.ma as ma
import pandas as pd
import scipy.cluster.hierarchy as sch
import scipy.stats as ss
import seaborn as sns
from psutil import cpu_count
import scipy

from correlation_analysis.data_plot import histogram_plot
from correlation_analysis.data_preprocess import convert, remove_incomplete_samples, replace_none_value_with_value_for_2_files, identify_columns_by_type

# define the parameters for future convenience
_REPLACE = "replace"
```

```
_DROP = "drop"

_DROP_SAMPLES = "drop_samples"

_DROP_FEATURES = "drop_features"

_SKIP = "skip"

_DEFAULT_NAN_REPLACE_VALUE = 0.0

# 'drop': remove samples with missing values

# 'replace': replace all missing values with the nan_replace_value

# the missing values are None and np.nan


_PRECISION = 1e-13

_NO_OP = "no-op"

_SINGLE_VALUE_COLUMN_OP = "single-value-column-op"

_I_EQ_J_OP = "i-equal-j-op"

_ASSOC_OP = "assoc-op"



def check_the_number_inf_none_value(input_num):

    if np.isnan(input_num):

        return "NaN"

    elif abs(input_num) == np.inf:

        return "inf"

    else:

        return ""


# Function check_the_number_inf_none_value checks if the input_num is NaN or inf, else return
""

# input_num: the input number



def conditional_entropy(
    input_x,
    input_y,
    nan_value_strategy=_REPLACE,
    nan_replace_value=_DEFAULT_NAN_REPLACE_VALUE,
    log_base: float = math.e,
):
    # handle the NaN values
    if nan_value_strategy == _REPLACE:
        input_x, input_y = replace_none_value_with_value_for_2_files(input_x, input_y,
nan_replace_value)
    elif nan_value_strategy == _DROP:
        input_x, input_y = remove_incomplete_samples(input_x, input_y)

    y_counter = Counter(input_y)
    xy_counter = Counter(list(zip(input_x, input_y)))
    total_occurrences = sum(y_counter.values())

    # calculate the conditional entropy
    entropy = 0.0
    for xy in xy_counter.keys():
        p_xy = xy_counter[xy]/total_occurrences
        p_y = y_counter[xy[1]]/total_occurrences
        entropy += p_xy * math.log(p_y/p_xy, log_base)
    return entropy

# Function conditional_entropy() calculates the conditional entropy of x given y and returns the
calculated entropy
# input_x: list/NumPy ndarray/Pandas Series
# input_y: list/NumPy ndarray/Pandas Series
# nan_value_strategy: string, default = 'replace'
# nan_replace_value: any, default = 0.0
# Only applicable when nan_value_strategy is set to 'replace'.
# log_base: float, default = e specifying base for calculating entropy, default base e.


def cramers_v(
    input_x,
    input_y,
    bias_correction=True,
    nan_value_strategy=_REPLACE,
    nan_replace_value=_DEFAULT_NAN_REPLACE_VALUE,
):
    if nan_value_strategy == _REPLACE:
        input_x, input_y = replace_none_value_with_value_for_2_files(input_x, input_y,
nan_replace_value)
    elif nan_value_strategy == _DROP:
        input_x, input_y = remove_incomplete_samples(input_x, input_y)
    confusion_matrix = pd.crosstab(input_x, input_y)
    chi_square = ss.chi2_contingency(confusion_matrix)[0]
    n = confusion_matrix.sum().sum()
    phi_square = chi_square/n
    r, k = confusion_matrix.shape
    if bias_correction:
        phi2corr = max(0, phi_square - ((k - 1) * (r - 1))/(n - 1))
        rcorr = r - ((r - 1) ** 2)/(n - 1)
        kcorr = k - ((k - 1) ** 2)/(n - 1)
        if min((kcorr - 1), (rcorr - 1)) == 0:
            warnings.warn(
                "Unable to calculate Cramer's V using bias correction",
                RuntimeWarning,
            )
            return np.nan
        else:
            v = np.sqrt(phi2corr/min((kcorr - 1), (rcorr - 1)))
    else:
        v = np.sqrt(phi_square/min(k - 1, r - 1))
    if -_PRECISION <= v < 0.0 or 1.0 < v <= 1.0 + _PRECISION:
        rounded_v = 0.0 if v < 0 else 1.0
        warnings.warn(
            f"Rounded V = {v} to {rounded_v}",
            RuntimeWarning,
        )
        return rounded_v
    else:
        return v

# Function cramers_v() calculates Cramer's V statistic for categorical-categorical association
# and returns a float in the range of [0,1]
# input_x: list/NumPy ndarray/Pandas Series
# input_y: list/NumPy ndarray/Pandas Series
# bias_correction: Boolean, default = True
# Use bias correction from Bergsma and Wicher,
# Journal of the Korean Statistical Society 42 (2013): 323-328.
# nan_value_strategy: string, default = 'replace'

def spearman_correlation(
    input_x,
    input_y,
    nan_value_strategy=_REPLACE,
    nan_replace_value=_DEFAULT_NAN_REPLACE_VALUE,
):
    if nan_value_strategy == _REPLACE:
        input_x, input_y = replace_none_value_with_value_for_2_files(input_x, input_y,
nan_replace_value)
    elif nan_value_strategy == _DROP:
        input_x, input_y = remove_incomplete_samples(input_x, input_y)
    v, p_value_1 = scipy.stats.spearmanr(input_x, input_y)
    print("the spearmanr_p_value is ", p_value_1)

    if -_PRECISION <= v < 0.0 or 1.0 < v <= 1.0 + _PRECISION:
        rounded_v = 0.0 if v < 0 else 1.0
        warnings.warn(
            f"Rounded V = {v} to {rounded_v}",
            RuntimeWarning,
        )
        return rounded_v
    else:
        return v



def pointbiseral_correlation(
    input_x,
    input_y,
    nan_value_strategy=_REPLACE,
    nan_replace_value=_DEFAULT_NAN_REPLACE_VALUE,
):
    if nan_value_strategy == _REPLACE:
        input_x, input_y = replace_none_value_with_value_for_2_files(input_x, input_y,
nan_replace_value)
```

```
        elif nan_value_strategy == _DROP:
            input_x, input_y = remove_incomplete_samples(input_x, input_y)
        v, p_value_1 = scipy.stats.pointbiserialr(input_x, input_y)
        print("the spearmanr_p_value is ", p_value_1)

        if -_PRECISION <= v < 0.0 or 1.0 < v <= 1.0 + _PRECISION:
            rounded_v = 0.0 if v < 0 else 1.0
            warnings.warn(
                f"Rounded V = {v} to {rounded_v}",
                RuntimeWarning,
            )
            return rounded_v
        else:
            return v




    def theils_u(
        input_x, input_y, nan_value_strategy=_REPLACE,
nan_replace_value=_DEFAULT_NAN_REPLACE_VALUE
    ):
        if nan_value_strategy == _REPLACE:
            input_x, input_y = replace_none_value_with_value_for_2_files(input_x, input_y,
nan_replace_value)
        elif nan_value_strategy == _DROP:
            input_x, input_y = remove_incomplete_samples(input_x, input_y)
        s_xy = conditional_entropy(input_x, input_y)
        x_counter = Counter(input_x)
        total_occurrences = sum(x_counter.values())
        p_x = list(map(lambda n: n/total_occurrences, x_counter.values()))
        s_x = ss.entropy(p_x)
        if s_x == 0:
            return 1.0
        else:
            u = (s_x - s_xy)/s_x
            if -_PRECISION <= u < 0.0 or 1.0 < u <= 1.0 + _PRECISION:
                rounded_u = 0.0 if u < 0 else 1.0
                warnings.warn(
                    f"Rounded U = {u} to {rounded_u}",
                    RuntimeWarning,
                )
                return rounded_u
            else:
                return u


    # Function theils_u() calculates Theil's U statistic for x given y
    # and returns a float in the range of [0,1]
    # input_x: list/NumPy ndarray/Pandas Series
    # input_y: list/NumPy ndarray/Pandas Series
    # nan_value_strategy: string, default = 'replace'
    # nan_replace_value: any, default = 0.0


    def correlation_ratio(
        categories,
        measurements,
        nan_value_strategy=_REPLACE,
        nan_replace_value=_DEFAULT_NAN_REPLACE_VALUE,
    ):
        if nan_value_strategy == _REPLACE:
            categories, measurements = replace_none_value_with_value_for_2_files(
                categories, measurements, nan_replace_value
            )
        elif nan_value_strategy == _DROP:
            categories, measurements = remove_incomplete_samples(
                categories, measurements
            )
        categories = convert(categories, "array")
        measurements = convert(measurements, "array")
        fcat, _ = pd.factorize(categories)
        cat_num = np.max(fcat) + 1
        y_avg_array = np.zeros(cat_num)
        n_array = np.zeros(cat_num)
        for i in range(0, cat_num):
            cat_measures = measurements[np.argwhere(fcat == i).flatten()]
            n_array[i] = len(cat_measures)
            y_avg_array[i] = np.average(cat_measures)
        y_total_avg = np.sum(np.multiply(y_avg_array, n_array))/np.sum(n_array)
        numerator = np.sum(
            np.multiply(n_array, np.power(np.subtract(y_avg_array, y_total_avg), 2))
        )
        denominator = np.sum(np.power(np.subtract(measurements, y_total_avg), 2))
        if numerator == 0:
            return 0.0
        else:
            eta = np.sqrt(numerator/denominator)
            if 1.0 < eta <= 1.0 + _PRECISION:
                warnings.warn(
                    f"Rounded eta = {eta} to 1",
                    RuntimeWarning,
                )
                return 1.0
            else:
                return eta


    def identify_nominal_columns(dataset):
        return identify_columns_by_type(dataset, include=["object", "category"])

    # Function identify_nominal_columns() identifies the categorical columns from the given dataset
    # and returns a list of the names of categorical columns
    # dataset: NumPy array/Pandas DataFrame



    def identify_numeric_columns(dataset):
        return identify_columns_by_type(dataset, include=["int64", "float64"])

    # Function identify_numeric_columns identifies the numeric columns from the given dataset
    # and returns a list of the names of numerical columns
    # dataset: NumPy array/Pandas DataFrame



    def association_computation_main(
        dataset,
        nominal_columns="auto",
        numerical_columns=None,
        mark_columns=False,
        nominal_nominal_assocication="cramer",
        numerical_numerical_association="pearson",
        nominal_numerical_association="correlation_ratio",
        symmetric_nominal_nominal=True,
        symmetric_numerical_numerical=True,
        display_rows="all",
        display_columns="all",
        hide_rows=None,
        hide_columns=None,
        cramers_v_bias_correction=True,
        nan_value_strategy=_REPLACE,
        nan_replace_value=_DEFAULT_NAN_REPLACE_VALUE,
        figure_ax=None,
        figure_size=None,
        plot_annotation=True,
        annotation_string_format=".2f",
        plot_color_map=None,
        sv_color="silver",
        color_bar_display_flag=True,
        heatmap_vmax=1.0,
        heatmap_vmin=None,
        plot=True,
        compute_only=False,
        clustering=False,
        plot_title=None,
        plot_save_filename=None,
        multiprocessing_flag=False,
        max_cpu_cores=None,
    ):
        dataset = convert(dataset, "dataframe")
```

```
if numerical_columns is not None:
    if numerical_columns == "auto":
        nominal_columns = "auto"
    elif numerical_columns == "all":
        nominal_columns = None
    else:
        nominal_columns = [
            c for c in dataset.columns if c not in numerical_columns
        ]

# handle the NaN values
if nan_value_strategy == _REPLACE:
    dataset.fillna(nan_replace_value, inplace=True)
elif nan_value_strategy == _DROP_SAMPLES:
    dataset.dropna(axis=0, inplace=True)
elif nan_value_strategy == _DROP_FEATURES:
    dataset.dropna(axis=1, inplace=True)

# identify the categorical columns
columns = dataset.columns
auto_nominal = False
if nominal_columns is None:
    nominal_columns = list()
elif nominal_columns == "all":
    nominal_columns = columns
elif nominal_columns == "auto":
    auto_nominal = True
    nominal_columns = identify_nominal_columns(dataset)
```

```
# select the rows and columns to be displayed
if hide_rows is not None:
    if isinstance(hide_rows, str) or isinstance(hide_rows, int):
        hide_rows = [hide_rows]
    display_rows = [c for c in dataset.columns if c not in hide_rows]
else:
    if display_rows == "all":
        display_rows = columns
    elif isinstance(display_rows, str) or isinstance(display_rows, int):
        display_columns = [display_rows]

if hide_columns is not None:
    if isinstance(hide_columns, str) or isinstance(hide_columns, int):
        hide_columns = [hide_columns]
    display_columns = [c for c in dataset.columns if c not in hide_columns]
else:
    if display_columns == "all":
        display_columns = columns
    elif isinstance(display_columns, str) or isinstance(
        display_columns, int
    ):
        display_columns = [display_columns]

if (
    display_rows is None
    or display_columns is None
    or len(display_rows) < 1
    or len(display_columns) < 1
):
    raise ValueError(
                "display_rows and display_columns must have at least one element"
            )
        displayed_features_set = set.union(set(display_rows), set(display_columns))

        # convert timestamp columns to numerical columns to perform correlation
        datetime_dtypes = [
            str(x) for x in dataset.dtypes if str(x).startswith("datetime64")
        ]  # finding all timezones
        if datetime_dtypes:
            datetime_cols = identify_columns_by_type(dataset, datetime_dtypes)
            datetime_cols = [c for c in datetime_cols if c not in nominal_columns]
            if datetime_cols:
                dataset[datetime_cols] = dataset[datetime_cols].apply(
                    lambda col: col.view(np.int64), axis=0
                )
                if auto_nominal:
                    nominal_columns = identify_nominal_columns(dataset)

        # store associations values
        corr = pd.DataFrame(index=columns, columns=columns)

        # this dataframe is used to keep track of invalid association values, which will be placed on
top
        # of the corr dataframe. It is done for visualization purposes, so the heatmap values will
remain
        # between -1 and 1
        inf_nan = pd.DataFrame(
            data=np.zeros_like(corr), columns=columns, index=columns
        )
```

```
# find the single-value columns
single_value_columns_set = set()
for c in displayed_features_set:
    if dataset[c].unique().size == 1:
        single_value_columns_set.add(c)

# find the number of physical cpu cores available
n_cores = cpu_count(logical=False)

# multiprocess implementation
if multiprocessing_flag and n_cores > 2:
    # find out the list of cartesian products of the column indices
    number_of_columns = len(columns)
    list_of_indices_pairs_lists = [
        (i, j)
        for i in range(number_of_columns)
        for j in range(number_of_columns)
    ]

    if max_cpu_cores is not None:
        max_cpu_cores = min(32, min(max_cpu_cores, n_cores))
    else:
        max_cpu_cores = min(32, n_cores)

    # submit each list of cartesian products of column indices to separate processes
    with cf.ProcessPoolExecutor(max_workers=max_cpu_cores) as executor:
        results = executor.map(
            _compute_associations,
            list_of_indices_pairs_lists,
            repeat(dataset),
            repeat(displayed_features_set),
            repeat(single_value_columns_set),
            repeat(nominal_columns),
            repeat(symmetric_nominal_nominal),
            repeat(nominal_nominal_assocication),
            repeat(cramers_v_bias_correction),
            repeat(numerical_numerical_association),
            repeat(nominal_numerical_association),
            repeat(symmetric_numerical_numerical),
            chunksize=max(
                1, len(list_of_indices_pairs_lists) // max_cpu_cores
            ),
        )
else:
    results = []

    for i in range(0, len(columns)):
        for j in range(i, len(columns)):
            results.append(
                _compute_associations(
                    [i, j],
                    dataset,
                    displayed_features_set,
                    single_value_columns_set,
                    nominal_columns,
                    symmetric_nominal_nominal,
                    nominal_nominal_assocication,
                    cramers_v_bias_correction,
                    numerical_numerical_association,
                    nominal_numerical_association,
                    symmetric_numerical_numerical,
                )
            )

# fill the correlation Dataframe with the results
for result in results:
    try:
        if result[0] == _NO_OP:
            pass
        elif result[0] == _SINGLE_VALUE_COLUMN_OP:
            i = result[1]
            corr.loc[:, columns[i]] = 0.0
            corr.loc[columns[i],:] = 0.0
        elif result[0] == _I_EQ_J_OP:
            i, j = result[1:]
            corr.loc[columns[i], columns[j]] = 1.0
        else:
            # assoc_op
            i, j, ij, ji = result[1:]
            corr.loc[columns[i], columns[j]] = (
                ij if not np.isnan(ij) and abs(ij) < np.inf else 0.0
            )
            corr.loc[columns[j], columns[i]] = (
                ji if not np.isnan(ji) and abs(ji) < np.inf else 0.0
            )
            inf_nan.loc[columns[i], columns[j]] = check_the_number_inf_none_value(ij)
            inf_nan.loc[columns[j], columns[i]] = check_the_number_inf_none_value(ji)
    except Exception as exception:
        raise exception
```

```
corr.fillna(value=np.nan, inplace=True)

if clustering:
    corr, _ = cluster_correlations(corr)
    inf_nan = inf_nan.reindex(columns=corr.columns).reindex(
        index=corr.index
    )

    # rearrange the dispalyed rows and columns according to the clustered order
    display_columns = [c for c in corr.columns if c in display_columns]
    display_rows = [c for c in corr.index if c in display_rows]

# keep only displayed columns and rows
corr = corr.loc[display_rows, display_columns]
inf_nan = inf_nan.loc[display_rows, display_columns]

if mark_columns:
    def mark(col):
        return (
            "{} (nom)".format(col)
            if col in nominal_columns
            else "{} (con)".format(col)
        )

    corr.columns = [mark(col) for col in corr.columns]
    corr.index = [mark(col) for col in corr.index]
    inf_nan.columns = corr.columns
    inf_nan.index = corr.index
    single_value_columns_set = {
        mark(col) for col in single_value_columns_set
    }
    display_rows = [mark(col) for col in display_rows]
    display_columns = [mark(col) for col in display_columns]

if not compute_only:
    if figure_ax is None:
        plt.figure(figsize=figure_size)
    if inf_nan.any(axis=None):
        inf_nan_mask = np.vectorize(lambda x: not bool(x))(inf_nan.values)
        figure_ax = sns.heatmap(
            inf_nan_mask,
            plot_color_map=["white"],
            plot_annotation=inf_nan if plot_annotation else None,
            annotation_string_format="",
            center=0,
            square=True,
            figure_ax=figure_ax,
            mask=inf_nan_mask,
            color_bar_display_flag=False,
        )
    else:
        inf_nan_mask = np.ones_like(corr)

    if len(single_value_columns_set) > 0:
        sv = pd.DataFrame(
            data=np.zeros_like(corr), columns=corr.columns, index=corr.index
        )
        for c in single_value_columns_set:
            if c in display_rows and c in display_columns:
                sv.loc[:, c] = " "
            sv.loc[c,:] = " "
            sv.loc[c, c] = "SV"
        elif c in display_rows:
            sv.loc[c,:] = " "
            sv.loc[c, sv.columns[0]] = "SV"
        else:    # c in display_columns
            sv.loc[:, c] = " "
            sv.loc[sv.index[-1], c] = "SV"
    sv_mask = np.vectorize(lambda x: not bool(x))(sv.values)
    figure_ax = sns.heatmap(
        sv_mask,
        plot_color_map=[sv_color],
        plot_annotation=sv if plot_annotation else None,
        annotation_string_format="",
        center=0,
        square=True,
        figure_ax=figure_ax,
        mask=sv_mask,
        color_bar_display_flag=False,
    )
else:
    sv_mask = np.ones_like(corr)

mask = np.vectorize(lambda x: not bool(x))(inf_nan_mask) + np.vectorize(
    lambda x: not bool(x)
)(sv_mask)

heatmap_vmin = heatmap_vmin or (
    -1.0
    if len(displayed_features_set) - len(nominal_columns) >= 2
```

```
                else 0.0
            )

            figure_ax = sns.heatmap(
                corr,
                cmap=plot_color_map,
                annot=plot_annotation,
                fmt=annotation_string_format,
                center=0,
                vmax=heatmap_vmax,
                vmin=heatmap_vmin,
                square=True,
                mask=mask,
                ax=figure_ax,
                cbar=color_bar_display_flag,
            )
            plt.title(plot_title)
            if plot_save_filename:
                plt.savefig(plot_save_filename)
            if plot:
                plt.show()

        return {"corr": corr, "figure_ax": figure_ax}

# Function association_computation_main() calculates the correlation in dataset
# and returns: a dictionary with the following keys:
# 'corr': A DataFrame of the correlation among all features
# 'figure_ax': A Matplotlib Axe

# dataset: NumPy array/Pandas DataFrame
```

# nominal_columns: string/list/NumPy ndarray, names of columns of the dataset holding categorical values

# numerical_columns: string/list/NumPy ndarray, names of columns of the dataset holding numerical values

# mark_columns: Boolean, default = False, if True, output's columns' names will have a suffix of '(nom)' or '(con)'

# nominal_nominal_assocication: callable/string, default = 'cramer'

# options: 'cramer', 'theil'

# numerical_numerical_association: callable/string, default = 'pearson'

# options: 'pearson', 'spearman', 'kendall'

# nominal_numerical_association: callable/string, default = 'correlation_ratio'

# options: 'correlation_ratio'

# symmetric_nominal_nominal: Boolean, default = True, declare whether the function is symmetric

# symmetric_numerical_numerical: Boolean, default = True, declare whether the function is symmetric

# display_rows: list/string, default = 'all', choose which of the dataset's features will be displyed in the output's correlations table rows

# display_columns: list/string, default = 'all', choose which of the dataset's features will be displyed in the output's correlations table columns

# hide_rows: list/string, default = None, choose which of the dataset's features will not be displyed in the output's correlations table rows.

# hide_columns: list/string, default = None, choose which of the dataset's features will not be displyed in the output's correlations table column

# cramers_v_bias_correction: Boolean, default = True, use bias correction for Cramer's V from Bergsma and Wicher,

# nan_value_strategy: string, default = 'replace'

# nan_replace_value: any, default = 0.0

# figure_ax: matplotlib figure_ax, default = None, Matplotlib Axis on which the heat-map will be plotted

# figure_size: (int,int) or None, default = None A Matplotlib figure-size tuple. If 'None', falls back to Matplotlib's default. Only used if 'figure_ax=None'.

# plot_annotation: Boolean, default = True, Plot number annotations on the heat-map

# annotation_string_format: string, default = '.2f' String formatting of annotations

# plot_color_map: Matplotlib colormap or None, default = None A colormap to be used for the heat-map. If None, falls back to Seaborn's heat-map default

# sv_color: string, default = 'silver A Matplotlib color. The color to be used when displaying single-value features over the heat-map

# color_bar_display_flag: Boolean, default = True Display heat-map's color-bar

# heatmap_vmax: float, default = 1.0, set heat-map heatmap_vmax option

# heatmap_vmin: float or None, default = None Set heat-map heatmap_vmin option. If set to None, heatmap_vmin will be chosen automatically between 0 and -1, depending on the types of associations used (-1 if Pearson's R is used, 0 otherwise)

# plot: Boolean, default = True, plot a heat-map of the correlation matrix. If false, the all axes and plotting still happen, but the heat-map will not be displayed.

# compute_only: Boolean, default = False Use this flag only if you have no need of the plotting at all. This skips the entire plotting mechanism.

# clustering: Boolean, default = False If True, hierarchical clustering is applied in order to sort features into meaningful groups

# plot_title: string or None, default = None Plotted graph plot_title

# plot_save_filename: string or None, default = None If not None, plot will be saved to the given file name

# multiprocessing_flag: Boolean, default = False If True, use 'multiprocessing_flag' to speed up computations. If None, falls back to single core computation

# max_cpu_cores: int or None, default = None If not None, ProcessPoolExecutor will use the given number of CPU cores

```
def nominal_numerical_correlation_calculation(nominal_column, numerical_column, dataset, nominal_numerical_association, nominal_nominal_assocication):
    if callable(nominal_numerical_association):
        cell = nominal_numerical_association(dataset[nominal_column], dataset[numerical_column])
        ij = cell
        ji = cell
    elif nominal_numerical_association == "correlation_ratio":
        cell = correlation_ratio(
            dataset[nominal_column], dataset[numerical_column], nan_value_strategy=_SKIP
            )
            ij = cell
            ji = cell
        else:
            raise ValueError(
                f"{nominal_nominal_assocication} is not a supported nominal-numerical
association"
            )
        return ij, ji


    # Function nominal_numerical_correlation_calculation computes the nominal-numerical
association value



    def _compute_associations(
        indices_pair,
        dataset,
        displayed_features_set,
        single_value_columns_set,
        nominal_columns,
        symmetric_nominal_nominal,
        nominal_nominal_assocication,
        cramers_v_bias_correction,
        numerical_numerical_association,
        nominal_numerical_association,
        symmetric_numerical_numerical,
    ):
        columns = dataset.columns

        i, j = indices_pair
```

```
if columns[i] not in displayed_features_set:
    return _NO_OP, None
if columns[i] in single_value_columns_set:
    return _SINGLE_VALUE_COLUMN_OP, i

if (
    columns[j] in single_value_columns_set
    or columns[j] not in displayed_features_set
):
    return _NO_OP, None
elif i == j:
    return _I_EQ_J_OP, i, j
else:
    if columns[i] in nominal_columns:
        if columns[j] in nominal_columns:
            if callable(nominal_nominal_assocication):
                if symmetric_nominal_nominal:
                    cell = nominal_nominal_assocication(
                        dataset[columns[i]], dataset[columns[j]]
                    )
                    ij = cell
                    ji = cell
                else:
                    ij = nominal_nominal_assocication(
                        dataset[columns[i]], dataset[columns[j]]
                    )
                    ji = nominal_nominal_assocication(
                        dataset[columns[j]], dataset[columns[i]]
                    )
            elif nominal_nominal_assocication == "theil":
```

```
        ij = theils_u(
            dataset[columns[i]],
            dataset[columns[j]],
            nan_value_strategy=_SKIP,
        )
        ji = theils_u(
            dataset[columns[j]],
            dataset[columns[i]],
            nan_value_strategy=_SKIP,
        )
    elif nominal_nominal_assocication == "spearman_rank_correlation":
        ij = spearman_correlation(
            dataset[columns[i]],
            dataset[columns[j]],
            nan_value_strategy=_SKIP,
        )
        ji = spearman_correlation(
            dataset[columns[j]],
            dataset[columns[i]],
            nan_value_strategy=_SKIP,
        )
    elif nominal_nominal_assocication == "cramer":
        cell = cramers_v(
            dataset[columns[i]],
            dataset[columns[j]],
            bias_correction=cramers_v_bias_correction,
            nan_value_strategy=_SKIP,
        )
        ij = cell
        ji = cell
```

```
                else:
                    raise ValueError(
                        f"{nominal_nominal_assocication} is not a supported nominal-
nominal association"
                    )
            else:
                ij, ji = nominal_numerical_correlation_calculation(
                    nominal_column=columns[i],
                    numerical_column=columns[j],
                    dataset=dataset,
                    nominal_numerical_association=nominal_numerical_association,
                    nominal_nominal_assocication=nominal_nominal_assocication,
                )
        else:
            if columns[j] in nominal_columns:
                ij, ji = nominal_numerical_correlation_calculation(
                    nominal_column=columns[j],
                    numerical_column=columns[i],
                    dataset=dataset,
                    nominal_numerical_association=nominal_numerical_association,
                    nominal_nominal_assocication=nominal_nominal_assocication,
                )
            else:
                if callable(numerical_numerical_association):
                    if symmetric_numerical_numerical:
                        cell = numerical_numerical_association(
                            dataset[columns[i]], dataset[columns[j]]
                        )
                        ij = cell
                        ji = cell
                    else:
                        ij = numerical_numerical_association(
                            dataset[columns[i]], dataset[columns[j]]
                        )
                        ji = numerical_numerical_association(
                            dataset[columns[j]], dataset[columns[i]]
                        )
                else:
                    if numerical_numerical_association == "pearson":
                        cell, _ = ss.pearsonr(
                            dataset[columns[i]], dataset[columns[j]]
                        )
                    elif numerical_numerical_association == "spearman":
                        cell, _ = ss.spearmanr(
                            dataset[columns[i]], dataset[columns[j]]
                        )
                    elif numerical_numerical_association == "kendall":
                        cell, _ = ss.kendalltau(
                            dataset[columns[i]], dataset[columns[j]]
                        )
                    elif numerical_numerical_association == "pointbiseral_correlation":
                        cell, _ = ss.pointbiserialr(
                            dataset[columns[i]], dataset[columns[j]])
                    else:
                        raise ValueError(
                            f"{numerical_numerical_association} is not a supported
    numerical-numerical association"
                        )
                    ij = cell
                    ji = cell
                    return (_ASSOC_OP, i, j, ij, ji)


            # Function _compute_associations()
            # and returns a list containing tuples. All tuples have one of the following strings in the 0-th index:
            # _NO_OP
            # _SINGLE_VALUE_COLUMN_OP
            # _I_EQ_J_OP
            # _ASSOC_OP


            # indices_pair: Tuple[int, int], the tuple of indices pairs (i, j)
            # dataset: pandas.Dataframe, the pandas dataframe
            # displayed_features_set: Set[str], the set of { display_rows } U { display_columns }
            # single_value_columns_set: Set[str], the set of single-value columns
            # nominal_columns: string/list/NumPy ndarray, default = 'auto', names of columns of the dataset which hold categorical values
            # symmetric_nominal_nominal: Boolean, default = True, declare whether the function is symmetric
            # nominal_nominal_assocication: callable/string, default = 'cramer'
            # options: 'cramer', 'theil'
            # numerical_numerical_association: callable/string, default = 'pearson'
            # options: 'pearson', 'spearman', 'kendall'
            # nominal_numerical_association: callable/string, default = 'correlation_ratio'
            # options: 'correlation_ratio'
            # symmetric_numerical_numerical: Boolean, default = True, declare whether the function is symmetric
            # cramers_v_bias_correction: Boolean, default = True, use bias correction for Cramer's V from Bergsma and Wicher,
    def numerical_encoding(
        dataset,
        nominal_columns="auto",
        drop_single_label=False,
        drop_fact_dict=True,
        nan_value_strategy=_REPLACE,
        nan_replace_value=_DEFAULT_NAN_REPLACE_VALUE,
    ):
        dataset = convert(dataset, "dataframe")
        if nan_value_strategy == _REPLACE:
            dataset.fillna(nan_replace_value, inplace=True)
        elif nan_value_strategy == _DROP_SAMPLES:
            dataset.dropna(axis=0, inplace=True)
        elif nan_value_strategy == _DROP_FEATURES:
            dataset.dropna(axis=1, inplace=True)
        if nominal_columns is None:
            return dataset
        elif nominal_columns == "all":
            nominal_columns = dataset.columns
        elif nominal_columns == "auto":
            nominal_columns = identify_nominal_columns(dataset)
        converted_dataset = pd.DataFrame()
        binary_columns_dict = dict()
        for col in dataset.columns:
            if col not in nominal_columns:
                converted_dataset.loc[:, col] = dataset[col]
            else:
                unique_values = pd.unique(dataset[col])
```

```
                if len(unique_values) == 1 and not drop_single_label:
                    converted_dataset.loc[:, col] = 0
                elif len(unique_values) == 2:
                    (
                        converted_dataset.loc[:, col],
                        binary_columns_dict[col],
                    ) = pd.factorize(dataset[col])
                else:
                    dummies = pd.get_dummies(dataset[col], prefix=col)
                    converted_dataset = pd.concat(
                        [converted_dataset, dummies], axis=1
                    )
        if drop_fact_dict:
            return converted_dataset
        else:
            return converted_dataset, binary_columns_dict
```

# Function numerical_encoding() encodes a dataset with mixed data to a numerical-only dataset

# categorical with only a single value will be marked as zero (or dropped, if requested)

# categorical with two values will be replaced with the result of Pandas 'factorize'

# categorical with more than two values will be replaced with the result of Pandas 'get_dummies', numerical columns will not be modified and returns DataFrame or (DataFrame, dict)

# If 'drop_fact_dict' is True, returns the encoded DataFrame, else, returns a tuple of the encoded DataFrame and dictionary

# each key is a two-value column, and the value is the original labels, as supplied by Pandas 'factorize'. Will be empty if no two-value columns are present in the dataset

# dataset: NumPy array/Pandas DataFrame

# nominal_columns: string/list/NumPy ndarray, names of columns of the dataset holding categorical values

# nan_value_strategy: string, default = 'replace'

```
# nan_replace_value: any, default = 0.0

# drop_single_label: Boolean, default = False, if True, nominal columns with only one single value will be dropped

# drop_fact_dict: Boolean, default = True, if True, the return value will be the encoded DataFrame alone

# If False, it will be a tuple of the DataFrame and the dictionary of the binary factorization



def cluster_correlations(correlation_matrix, indices=None):

    if indices is None:

        X = correlation_matrix.values

        d = sch.distance.pdist(X)

        L = sch.linkage(d, method="complete")

        indices = sch.fcluster(L, 0.5 * d.max(), "distance")

    columns = [

        correlation_matrix.columns.tolist()[i] for i in list((np.argsort(indices)))

    ]

    correlation_matrix = correlation_matrix.reindex(columns=columns).reindex(index=columns)

    return correlation_matrix, indices


# Function cluster_correlations applies agglomerative clustering to sort a correlation matrix

# and returns corr: a sorted correlation matrix and indices: cluster indices based on the original dataset

# correlation_matrix: a square correlation matrix (pandas DataFrame)

# indices: cluster labels [None]; if not provided we'll do an aglomerative clustering to get cluster labels.


data_plot：

import matplotlib.pyplot as plt
```

```
def histogram_plot(
        input_dataset,
        values,
        split_by,
        plot_title="",
        x_axis_label="",
        y_axis_label=None,
        figure_size=None,
        figure_legend="best",
        plot=True,
        **hist_kwargs,
):
    # figure size
    plt.figure(figure_size=figure_size)

    split_vals = input_dataset[split_by].unique()
    data_split = list()
    for val in split_vals:
        data_split.append(input_dataset[input_dataset[split_by] == val][values])
    hist_kwargs["label"] = split_vals
    plt.hist(data_split, **hist_kwargs)

    # figure_legend
    if figure_legend:
        plt.figure_legend(loc=figure_legend)

    # figure x axis label
    if x_axis_label is not None:
        if x_axis_label == "":
                x_axis_label = values
            plt.x_axis_label(x_axis_label)

        # figure plot_title
        if plot_title is not None:
            if plot_title == "":
                plot_title = values + " by " + split_by
            plt.plot_title(plot_title)

        # figure y axis label
        plt.y_axis_label(y_axis_label)
        figure_ax = plt.gca()

        # display the figure
        if plot:
            plt.show()
        return figure_ax



    # Function histogram_plot() plots a histogram of values from a given dataset, split by the values of a chosen column
    # and returns a Matplotlib figure

    # dataset: Pandas DataFrame
    # values: The column name of the values to be displayed in the histogram
    # split_by: The column name of the values to split the histogram by
    # plot_title: string or None, default = '' The plot's plot_title. If empty string, will be '{values} by {split_by}'
    # x_axis_label: string or None, default = '', if empty string, will be '{values}'
    # y_axis_label: y-axis label
```

```
# figure_size: (int,int), a Matplotlib figure-size tuple
# figure_legend: A Matplotlib figure_legend location string
# plot: Boolean, default = True, plot the histogram
# hist_kwargs: A key-value pairs to be passed to Matplotlib hist method.

data_preprocess：
import numpy as np
import pandas as pd



def convert(input_data, output_format, copy=True):
    converted = None

    if output_format == "array":
        if isinstance(input_data, np.ndarray):
            converted = input_data.copy() if copy else input_data
        elif isinstance(input_data, pd.Series):
            converted = input_data.values
        elif isinstance(input_data, list):
            converted = np.array(input_data)
        elif isinstance(input_data, pd.DataFrame):
            converted = input_data.values()

    elif output_format == "list":
        if isinstance(input_data, list):
            converted = input_data.copy() if copy else input_data
        elif isinstance(input_data, pd.Series):
            converted = input_data.values.tolist()
        elif isinstance(input_data, np.ndarray):
            converted = input_data.tolist()
    elif output_format == "dataframe":
        if isinstance(input_data, pd.DataFrame):
            converted = input_data.copy(deep=True) if copy else input_data
        elif isinstance(input_data, np.ndarray):
            converted = pd.DataFrame(input_data)

    else:
        raise ValueError("Unknown data conversion: {}".format(output_format))
    if converted is None:
        raise TypeError(
            "cannot handle data conversion of {} to {}".format(
                type(input_data), output_format
            )
        )

    else:
        return converted
# the function convert() convert the input data input_data into the format of output_format
# output_format: "array", "dataframe", "list"
# otherwise cannot be converted, return error
# input_data: the data that willing to convert
# output_format: the desired convert format


def remove_incomplete_samples(input_1, input_2):
    input_1 = [iter_1 if iter_1 is not None else np.nan for iter_1 in input_1]
    input_2 = [iter_1 if iter_1 is not None else np.nan for iter_1 in input_2]
    arr = np.array([input_1, input_2]).transpose()
    arr = arr[~np.isnan(arr).any(axis=1)].transpose()
```

```
        if isinstance(input_1, list):
            return arr[0].tolist(), arr[1].tolist()
        else:
            return arr[0], arr[1]
    # function remove_incomplete_samples removes the incomplete samples
    # input_1, input_2: the input data



    def replace_none_value_with_value_for_2_files(input_1, input_2, replace_value):


        input_1 = np.array(
            [iter_1 if iter_1 == iter_1 and iter_1 is not None else replace_value for iter_1 in
input_1]
        )


        input_2 = np.array([iter_1 if iter_1 == iter_1 and iter_1 is not None else replace_value for
iter_1 in input_2])
        return input_1, input_2
    # function replace_none_value_with_value_for_2_files replaces the NaNs with replace_value
    # replace_value: the value used for replace
    # input_1, input_2: the input data


    def replace_none_value_with_value(input_1, replace_value):


        input_1 = np.array(
            [iter_1 if iter_1 == iter_1 and iter_1 is not None else replace_value for iter_1 in
input_1]
        )


        return input_1
    def identify_columns_by_type(input_dataset, include):
        input_dataset = convert(input_dataset, "dataframe")
        columns = list(input_dataset.select_dtypes(include=include).columns)
        return columns

    # Function identify_columns_by_type identifies the columns including the desired
    # and returns the column titles
    # input_dataset: the input dataset
    # include: list of strings containing desired column types


    def identify_columns_with_none_value(dataset):
        dataset = convert(dataset, "dataframe")
        na_count = [sum(dataset[cc].isnull()) for cc in dataset.columns]
        return (
            pd.DataFrame({"column": dataset.columns, "na_count": na_count}).query("na_count >
0").sort_values("na_count",

ascending=False)
        )

    # Function identify_columns_with_none_value identifies the columns having NA values, sorted
in descending order by #NA
    # dataset: NumPy array/Pandas DataFrame
    # Returns a DataFrame of two columns (['column', 'na_count']),
    # 'column' consists of only the names of columns with NA values, sorted by their number of NA
values

    def one_hot_encode(arr, classes=None):
        arr = convert(arr, "array").astype(int)
        if not len(arr.shape) == 1:
            raise ValueError(
                f"array must have only one dimension, but has shape: {arr.shape}"
            )
        if arr.min() < 0:
            raise ValueError("array cannot contain negative values")
        classes = classes if classes is not None else arr.max() + 1
        h = np.zeros((arr.size, classes))
        h[np.arange(arr.size), arr] = 1
        return h

    # Function one_hot_encode one-hot encodes a array and returns a 2D one-hot encoded array
    # arr: array-like, an array to be one-hot encoded, contain only non-negative integers
    # classes: int or None, the number of classes, if None, replace with the max value of the array
```

聚类：

main:

```
from pathlib import Path
import pandas as pd
from scipy.sparse import data
from scipy.cluster.hierarchy import linkage, dendrogram
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.graph_objects as go
import plotly.figure_factory as ff
import numpy as np
from scipy.spatial.distance import pdist, squareform
from sklearn.cluster import KMeans
import plotly as py
import plotly.graph_objs as go
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
import csv
from yellowbrick.cluster import KElbowVisualizer

from kmeans.kmeans_function import optimalK, kmeans_assignment
from kmeans.k_elbow_main import k_elbow_calculate

if __name__ == '__main__':
    # create the file handles for each list
    path_root = str(Path.cwd())
    data_file_root = path_root.rstrip("kmeans") + "raw_data_source"
    data_handle_1 = data_file_root + "\\raw_data_form_1.CSV"
    data_handle_2 = data_file_root + "\\raw_data_form_2_preprocessed.CSV"
    data_handle_2_X2_0 = data_file_root + "\\pre_processed_X2_0.CSV"
    data_handle_2_X2_1 = data_file_root + "\\pre_processed_X2_1.CSV"
    data_handle_2_color_append = data_file_root +
"\\raw_data_form_2_preprocessed_color_append.CSV"
    data_handle_3 = data_file_root + "\\raw_data_form_3.CSV"
    data_handle_2_pca_processed_data = data_file_root + "\\pca_processed_form_2.CSV"
    data_handle_pca_processed_data_form_3 = data_file_root +
"\\pca_processed_form_3.CSV"
    data_handle_2_X2_0_pca = data_file_root + "\\pca_processed_X2_0.CSV"
    data_handle_2_X2_1_pca = data_file_root + "\\pca_processed_X2_1.CSV"

    # identify the columns
    columns = ['PC1', 'PC2', 'PC3', 'PC4', 'PC5', 'PC6', 'PC7', 'PC8', 'PC9', 'PC10']
    columns_1 = ["index"]
    columns_2 = ["Number"]
```

```
        # read in the data files
        df_2_pca_processed = pd.read_csv(data_handle_2_pca_processed_data, header=0, 
encoding='ISO-8859-1',
                                         usecols=columns, )
        df_2_pca_processed = df_2_pca_processed.fillna(0)

        df_2_X2_0_pca_processed = pd.read_csv(data_handle_2_X2_0_pca, header=0, 
encoding='ISO-8859-1',
                                              usecols=columns, )
        df_2_X2_0_pca_processed = df_2_X2_0_pca_processed.fillna(0)

        df_2_X2_1_pca_processed = pd.read_csv(data_handle_2_X2_1_pca, header=0, 
encoding='ISO-8859-1',
                                              usecols=columns, )
        df_2_X2_1_pca_processed = df_2_X2_1_pca_processed.fillna(0)

        df_3_pca_processed = pd.read_csv(data_handle_pca_processed_data_form_3, header=0, 
encoding='ISO-8859-1',
                                         usecols=columns, )
        df_3_pca_processed = df_3_pca_processed.fillna(0)

        df_only_index_X2_0 = pd.read_csv(data_handle_2_X2_0, header=0, encoding='ISO-8859-
1', sep="\t",
                                         usecols=columns_1, )
        df_only_index_X2_0 = df_only_index_X2_0.fillna(0)

        df_only_index_X2_1 = pd.read_csv(data_handle_2_X2_1, header=0, encoding='ISO-8859-
1', sep="\t",
                                         usecols=columns_1, )
        df_only_index_X2_1 = df_only_index_X2_1.fillna(0)

        df_only_index_all = pd.read_csv(data_handle_2, header=0, encoding='ISO-8859-1',
                                            usecols=columns_1, )
        df_only_index_all = df_only_index_all.fillna(0)


        df_only_index_3 = pd.read_csv(data_handle_3, header=0, encoding='ISO-8859-1',
                                          usecols=columns_2, )


        # get the data within the data files
        samples_2_total = df_2_pca_processed.iloc[:, :].values


        samples_2_X2_0 = df_2_X2_0_pca_processed.iloc[:, :].values


        samples_2_X2_1 = df_2_X2_1_pca_processed.iloc[:, :].values


        # samples_2_total


        labels_2_total = []
        for i in range(len(df_2_pca_processed)):
            labels_2_total.append(i + 1)


        labels_2_X2_0 = df_only_index_X2_0.iloc[:, :].values


        labels_2_X2_1 = df_only_index_X2_1.iloc[:, :].values

##########################################################################################
############################
        score_g, df_result_all, kmeans_obj = optimalK(samples_2_total, nrefs=5, maxClusters=30)
        plt.plot(df_result_all['clusterCount'], df_result_all['gap'], linestyle='--', marker='o', color='b')
        plt.xlabel('K')
        plt.ylabel('Gap Statistic')
        # plt.savefig("score_g.png")
```

```
            # plt.show()
            row_title = ['index', 'cluster']


############################################################################################
############################


            # k is range of number of clusters.
            k_means_model_all = k_elbow_calculate(samples_2_total, df_only_index_all, 30, 8,
path_root, "all samples",
                                                                    data_file_root, row_title, 'cluster_result_all')


############################################################################################
############################
            k_means_model_X2_0 = k_elbow_calculate(samples_2_X2_0, df_only_index_X2_0, 10, 4,
path_root, "samples with X2=0",
                                                                    data_file_root, row_title,
'cluster_result_X2_0')

            k_means_model_X2_1 = k_elbow_calculate(samples_2_X2_1, df_only_index_X2_1, 10, 5,
path_root, "samples with X2=1",
                                                                    data_file_root, row_title,
'cluster_result_X2_1')

            list_3_assign_to_X2_0, distance_X2_0 =
kmeans_assignment(centroids=k_means_model_X2_0.cluster_centers_,

points=df_3_pca_processed.iloc[:, :].values)
            list_3_assign_to_X2_1, distance_X2_1 =
kmeans_assignment(centroids=k_means_model_X2_1.cluster_centers_,

points=df_3_pca_processed.iloc[:, :].values)
```

```
row_title_1 = ["Number", 'cluster_in_0', "cluster_in_1"]
X3_info = df_only_index_3
X3_info['cluster_in_0'] = list_3_assign_to_X2_0
X3_info["cluster_in_1"] = list_3_assign_to_X2_1

with open(data_file_root + "/" + "cluster_result_3" + ".CSV", 'w') as f:
    write = csv.writer(f)
    write.writerow(row_title_1)
    write.writerows(X3_info.values)



##########################################################################################
############################
row_title_2 = []
row_title_3 = []

with open(data_file_root + "/" + "distance_in_X2_0" + ".CSV", 'w') as f_1:
    for i in range(4):
        row_title_2.append("centroid " + str(i+1))
    write = csv.writer(f_1)
    write.writerow(row_title_2)
    write.writerows(distance_X2_0)

with open(data_file_root + "/" + "distance_in_X2_1" + ".CSV", 'w') as f_2:
    for i in range(6):
        row_title_3.append("centroid " + str(i+1))
    write = csv.writer(f_2)
    write.writerow(row_title_3)
    write.writerows(distance_X2_1)
```

```
import numpy as np
import pandas as pd
import sklearn
from sklearn.cluster import KMeans


# Gap Statistic for K means
def optimalK(data, nrefs=3, maxClusters=15):
    gaps = np.zeros((len(range(1, maxClusters)),))
    resultsdf = pd.DataFrame({'clusterCount': [], 'gap': []})
    for gap_index, k in enumerate(range(1, maxClusters)):
        # Holder for reference dispersion results
        refDisps = np.zeros(nrefs)
        # For n references, generate random sample and perform kmeans getting resulting
dispersion of each loop
        for i in range(nrefs):
            # Create new random reference set
            randomReference = np.random.random_sample(size=data.shape)

            # Fit to it
            km = KMeans(k)
            km.fit(randomReference)

            refDisp = km.inertia_
            refDisps[i] = refDisp
        # Fit cluster to original data and create dispersion
        km = KMeans(k)
        km.fit(data)
```

```
            origDisp = km.inertia_
            # Calculate gap statistic
            gap = np.log(np.mean(refDisps)) - np.log(origDisp)
            # Assign this loop's gap statistic to gaps
            gaps[gap_index] = gap

            resultsdf = resultsdf.append({'clusterCount': k, 'gap': gap}, ignore_index=True)

        return (gaps.argmax() + 1, resultsdf, km)

""""
        Calculates KMeans optimal K using Gap Statistic
        Params:
            data: ndarry of shape (n_samples, n_features)
            nrefs: number of sample reference datasets to create
            maxClusters: Maximum number of clusters to test for
        Returns: (gaps, optimalK)
""""


def kmeans_assignment(centroids, points):
    num_centroids, dim = centroids.shape
    num_points, _ = points.shape

    # Tile and reshape both arrays into `[num_points, num_centroids, dim]`.
    centroids = np.tile(centroids, [num_points, 1]).reshape([num_points, num_centroids, dim])
    points = np.tile(points, [1, num_centroids]).reshape([num_points, num_centroids, dim])

    # Compute all distances (for all points and all centroids) at once and
    # select the min centroid for each point.
    distances = np.sum(np.square(centroids - points), axis=2)
    return np.argmin(distances, axis=1), distances


from pathlib import Path
import pandas as pd
from phik.binning import bin_data
from phik.report import plot_correlation_matrix
from scipy.sparse import data
from scipy.cluster.hierarchy import linkage, dendrogram
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.graph_objects as go
import plotly.figure_factory as ff
import numpy as np
from scipy.spatial.distance import pdist, squareform
from sklearn.cluster import KMeans
import plotly as py
import plotly.graph_objs as go
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
import csv
from yellowbrick.cluster import KElbowVisualizer



def k_elbow_calculate(input_sample, sample_index, a, b,path_root, figure_name, data_file_root, row_title, result_file_name):
        k_means_model = KMeans()
        visualizer = KElbowVisualizer(k_means_model, k=(2, a), timings=True)
        visualizer.fit(input_sample)    # Fit data to visualizer
        visualizer.show()    # Finalize and render figure
        k_means_model = KMeans(b)
```

```
k_means_model.fit(input_sample)
X2_0_result = k_means_model.labels_


X2_0_index = sample_index
X2_0_index['cluster'] = X2_0_result
with open(data_file_root + "/" + result_file_name + ".CSV", 'w') as f:
    write = csv.writer(f)
    write.writerow(row_title)
    write.writerows(X2_0_index.values)


return k_means_model
```