机器学习与数据挖掘 ~ Homework2

19335084 黄梓浩

1 实验目的和实验要求

1.1 实验目的

实现并测试K-means聚类算法

1.2 实验要求

- 实现K-means算法，采用三种不同初始化方法。
- 聚类个数设定为正确类个数情况下，三种不同初始化方法版本的聚类效果，使用NMI (Normalized Mutual Information) 度量效果。
- 设置不同聚类个数K的情况下，三种不同初始化方法版本的"目标函数|随着聚类个数变化的曲线。

2 算法原理

2.1 聚类问题

聚类问题 属于 无监督学习，机器学习主要分为 有监督学习 和 无监督学习，其中无监督学习指的是 从没有标注的数据中学习模型的机器学习问题，实际中体现在数据中没有标签项（没有label项用于给初始样本分类），因此样本的类是从数据中得到的。

而聚类问题的目标在于，将样本集合中相似的样本通过某种方法分配到同一个类下，不相似的样本则分到不同的类下。其中，样本所属于的类可以是不固定的，如果一个样本只能属于一个类的模型，称为硬聚类，一个样本能属于多个类称为软聚类。

2.2 K - Means

K - Means 是一种常用的聚类算法。对有n个样本数据的集合X，K - Means 的目标是找到一种最佳的分类方式，让 n 个样本分到 K 个不同的类中，这使得类中的样本尽量相似，不同类的样本的差异尽量大。

使用特征向量来代表各个样本，设第 K 个类的中心特征向量为 $\boldsymbol{\mu}_k$，同类下的样本差异要小，而不同类的样本差异要大。

我们可以用 欧氏距离平方 来定义样本间的差异：$\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\|^2$

因此可以定义损失函数为：$\sum_{k=1}^{K}\sum_{n=1}^{N} r_{nk}\|\boldsymbol{x}^{(n)} - \boldsymbol{\mu}_k\|^2$

令该函数最小即可找到最佳划分，通过以下迭代的方式得到：

- 通过给定聚类中心，将各样本分配到各个类； 分析该函数可知，只有当各个样本被分配到和它最相近的中心样本的类中，才能使函数最小化。

- 通过上一步得到的分配结果，更新各个类的聚类中心； 分析可知，当聚类中心为该类中的样本均值时，类中的样本和聚类中心的距离之和最小，可得聚类中心的更新公式为：

$$\boldsymbol{\mu}_k = \frac{\sum_{n=1}^{N} r_{nk}\boldsymbol{x}_n}{\sum_{n=1}^{N} r_{nk}}$$

重复上述步骤改变参数，可使得分类不再改变，损失函数收敛到最小。

在开始迭代前，需要初始化各个类的聚类中心，通常有三种方法初始化聚类中心：

- **[random]** 随机选 K 个样本作为初始聚类中心。
- **[distance]** 先随机选 1 个样本作为新聚类中心，再不断选 K-1 个离当前所有聚类中心最远的样本作为新的聚类中心。
- [random+distance] 先随机选 1 个样本作为新聚类中心，再从离当前所有聚类中心最远的样本中随机选 1 个样本作为新聚类中心，直到选出 K 个聚类中心为止。

初始化完成即可进行迭代完成模型训练。

3 代码

3.1 预处理数据集

本次实验我们使用的数据集：

```
https://archive.ics.uci.edu/ml/datasets/Travel+Reviews
https://archive.ics.uci.edu/ml/datasets/Facebook+Live+Sellers+in+Thailand
https://archive.ics.uci.edu/ml/datasets/Online+Shoppers+Purchasing+Intention+Dat
aset
https://archive.ics.uci.edu/ml/datasets/Tarvel+Review+Ratings
https://archive.ics.uci.edu/ml/datasets/Turkiye+Student+Evaluation
```

写一个 `trans.py` 用于预处理这些数据集，使之变为 numpy 容易处理的矩阵文件格式。

可以得到 data\下 1_image~ 5_image，1_label ~ 5_label 的 npy文件，分别表示5个数据集的特征集和标签集，在学习中可以直接用 numpy库 的 load方法 方便读取数据集。

3.2 读数据

主要是有一个寻找数据的 正确类大小 的过程，这个变量将用于完成第一个对比实验。

因为标签的数据是已经处理好的，里面都是从1到 K 的连续不重复整数。

```
data_num = 5

images = np.load('data/'+str(data_num)+'_image.npy')
raw_label = np.load('data/'+str(data_num)+'_label.npy')

row = len(raw_label)
labels = np.zeros(row,dtype=np.int64)

# 从给的标签中找出正确的类大小
K_true = 0
for i in range(0,row):
    labels[i] = raw_label[i]-1
```

```
    if(labels[i]>K_true):
        K_true = labels[i]
K_true = K_true+1
print(K_true)
```

3.3 K - Means 模型代码：

设置 kmeans类，用于K-Means 聚类模型的学习：

```
class kmeans():
    def __init__(self, data, K=2, method='random'):
    # 初始化聚类中心，method可选 random|distance|random+distance 三种方法
    def init_centers(self, data, method):
    # 返回各个类中的所有样本和聚类中心的欧式距离平方总和
    def get_distance(self, data):
    # 返回样本所属聚类集
    def get_cluster(self, data):
    # 返回新的聚类中心集
    def get_center(self, data, clusters):
    # 模型训练过程
    def train(self, data):
    # 返回实验所需要的NMI和J值
    def ret(self, data, labels):
```

初始化聚类中心的函数，可以通过改变 method的字符串值 来选择初始化的方法：

```
# 初始化聚类中心，method可选 random|distance|random+distance 三种方法
def init_centers(self, data, method):
    # 生成一个随机化的序列作为索引
    shuff_num = np.arange(data.shape[0])
    np.random.shuffle(shuff_num)

    if method=='random':
        # 随机选K个作为聚类中心
        self.centers = np.zeros((self.K, data.shape[1]))
        for i in range(self.K):
            self.centers[i] = data[shuff_num[i]]

    elif method=='distance':
        self.centers = np.zeros((1, data.shape[1]))
        # 记录已选的样本序号
        visted = []
        # 先随机选一个样本为第一个聚类中心，
        self.centers[0] = data[shuff_num[0]]
        visted.append(shuff_num[0])
        # 再选K-1个离当前聚类中心最远的样本为聚类中心
        for i in range(1,self.K):
            distance_arr = self.get_distance(data)
            sum_dis = np.sum(distance_arr, axis=1)
            # 从大到小排序距离
            shuff_num = np.argsort(-sum_dis)
            for i in shuff_num:
```

```
                if i not in visted:
                    self.centers = np.insert(self.centers,
obj=self.centers.shape[0], values=data[i], axis=0)
                    visted.append(i)
                    break

    elif method=='random+distance':
        self.centers = np.zeros((1, data.shape[1]))
        visted = np.zeros(data.shape[0], dtype=int)
        # 先随机选一个样本为第一个聚类中心，
        self.centers[0] = data[shuff_num[0]]
        visted[shuff_num[0]] = 1
        # 然后在离当前聚类中心最远的点中 随机选 新的聚类中心
        for i in range(1,self.K):
            # 没选的样本序列
            notvisted = np.argsort(visted)
            notvisted = notvisted[:-i]
            # 计算没选的样本离聚类中心的总距离
            distance_arr = self.get_distance(data[notvisted,:])
            sum_dis = np.sum(distance_arr, axis=1)
            # 从远到近排序
            shuff_num = np.argsort(-sum_dis)
            while True:
                i = notvisted[random.randint(0,notvisted.size-1)]
                # 随机选点，如果没被选过，而且距离超过一半的点就是新的中心
                if i in
notvisted[shuff_num[:int(np.ceil(float(shuff_num.size)/2))]]:
                    self.centers = np.insert(self.centers,
obj=self.centers.shape[0], values=data[i], axis=0)
                    visted[i] = 1
                    break
```

用于计算样本和聚类中心的欧氏距离平方的函数：

```
    # 返回各个类中的所有样本和聚类中心的欧式距离平方总和
    def get_distance(self, data):
        distance_arr = np.zeros((data.shape[0], self.centers.shape[0]))
        for i in range(data.shape[0]):
            distance_arr[i] = np.sum((self.centers - data[i])**2, axis=1)**0.5
        return distance_arr
```

通过该距离来把各样本分配到各个类下，得到各样本的聚类集：

```
    # 返回样本所属聚类集
    def get_cluster(self, data):
        distance_arr = self.get_distance(data)
        clusters = np.argmin(distance_arr, axis=1)
        sum_dis = np.sum(np.min(distance_arr, axis=1))
        return clusters, sum_dis
```

根据新的分配聚类集，更新新的聚类中心：

```
    # 返回新的聚类中心集
    def get_center(self, data, clusters):
        now_cluster = np.zeros((data.shape[0], self.K))
        now_cluster[clusters[:,None]==np.arange(self.K)] = 1
        # print(now_cluster)
        return np.dot(now_cluster.T, data)/np.sum(now_cluster,
axis=0).reshape((-1,1))
```

训练过程，基本是按照 第2节 的方法步骤进行，先把各样本分配到聚类，然后通过各样本的聚类更新聚类中心；

```
    # 模型训练过程
    def train(self, data):
        clusters, _ = self.get_cluster(data)
        newcenters = self.get_center(data, clusters)
        d_val = np.sum((newcenters-self.centers)**2)**0.5
        self.centers = newcenters
        return d_val
```

返回度量值的方法：

```
    # 返回实验所需要的NMI和J值
    def ret(self, data, labels):
        clusters, sum_dis = self.get_cluster(data)
        return normalized_mutual_info_score(clusters, labels), sum_dis
```

3.4 对比实验 1 过程

聚类个数设定为正确类个数情况下，三种不同初始化方法版本的聚类效果，使用NMI度量效果。

主要步骤是轮流使用不同方式初始化

实验过程代码如下：

```
# 聚类个数设定为正确类个数情况下，三种不同初始化方法版本的聚类效果
plt.figure()
plt.xlabel("epochs")
plt.ylabel("NMI")
plt.xticks(range(1,epochs+1))

method_str = ['random','distance','random+distance']
color_str = ['red','blue','black']
# 轮流选三种方法初始化
for r in range(3):
    nmi_arr = []
    print('method = ' + method_str[r] + ' : ')
    # 类设定为正确类个数
    train_model = kmeans(data=images, K=K_true, method=method_str[r])
    # 训练 epochs 轮，记录每轮的 NMI 值
    for i in range(epochs):
```

```
        dif = train_model.train(images)
        nmi_val, distance = train_model.ret(images, labels)
        print('J = {:.4f} NMI = {:.4f}'.format(distance , nmi_val))
        nmi_arr.append( nmi_val )
    # 将该method的 NMI 值变化加入图中
    plt.plot(range(1,epochs+1), nmi_arr, c=color_str[r], label=method_str[r])

plt.legend()
plt.grid()
plt.savefig('nmi_'+str(data_num)+'.jpg', dpi = 1800)
plt.show()
```

3.5 对比实验 2 过程

设置不同聚类个数K的情况下，三种不同初始化方法版本的 目标函数J 随着聚类个数变化的曲线，过程代码如下：

```
# 设置不同聚类个数K的情况下，三种不同初始化方法版本的 目标函数J 随着聚类个数变化的曲线。
plt.figure()
plt.xlabel("K")
plt.ylabel("J_val")
plt.xticks(range(1,epochs+1))

method_str = ['random','distance','random+distance']
color_str = ['red','blue','black']
for r in range(3):
    J_arr = []
    print('method = ' + method_str[r] + ' : ')
    # 聚类个数变化在 2~K
    for K_num in range(2,K_true+1):
        train_model = kmeans(data=images, K=K_num, method=method_str[r])
        for i in range(epochs):
            dif = train_model.train(images)
            # 差别小到一定程度，结束训练
            if dif < 1e-6:
                endi = epochs-i-1
                break
        nmi_val, j_val = train_model.ret(images, labels)
        print('J = {:.4f} NMI = {:.4f}'.format(j_val , nmi_val))
        J_arr.append( j_val )

    plt.plot(range(2,K_true+1), J_arr, c=color_str[r], label=method_str[r])

plt.legend()
plt.grid()
plt.savefig('J'+str(data_num)+'.jpg', dpi = 1800)
# plt.show()
```

4 结果分析

4.1 对比实验 1

聚类个数设定为正确类个数情况下，三种不同初始化方法版本的聚类效果、

设定 epochs为 50 ，使得每种方法下的 K-Means 模型训练50轮。

```
epochs = 50
```

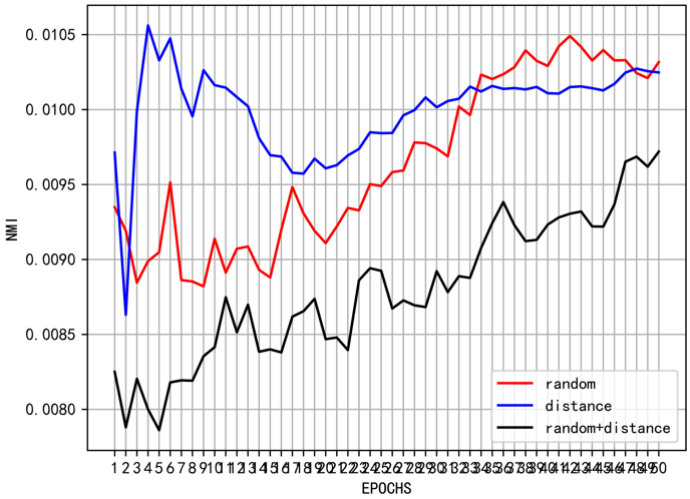
直接使用 sklearn 的库函数 `normalized_mutual_info_score(clusters, labels)` 来求得 NMI信息，clusters 为当前的所有样本所属聚类的集合，labels 是源数据的分类标签。

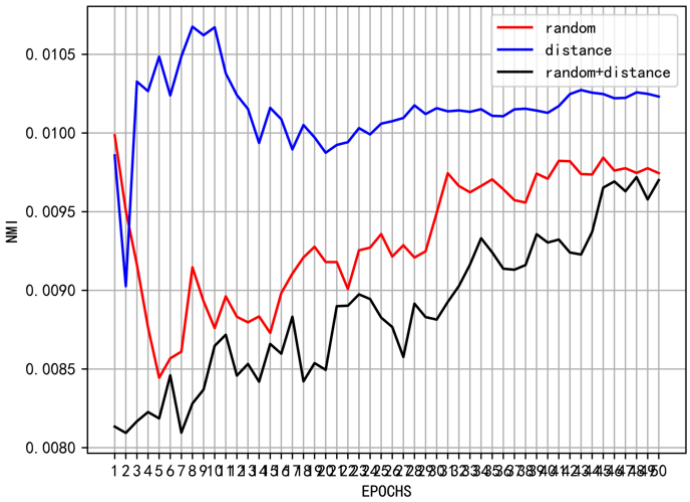
```
from sklearn.metrics.cluster import normalized_mutual_info_score
...
return normalized_mutual_info_score(clusters, labels), sum_dis
```

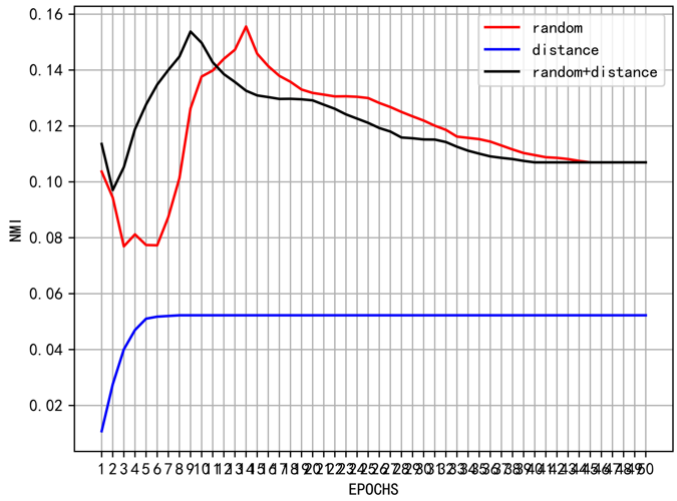
轮流使用三种不同初始化方法：

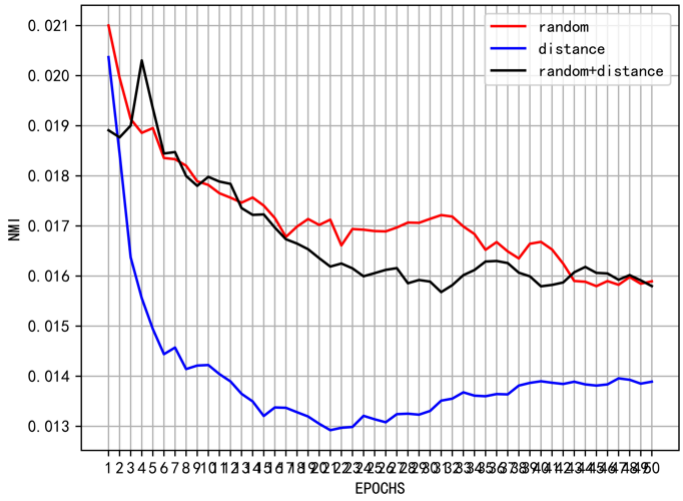
```
method_str = ['random','distance','random+distance']
```

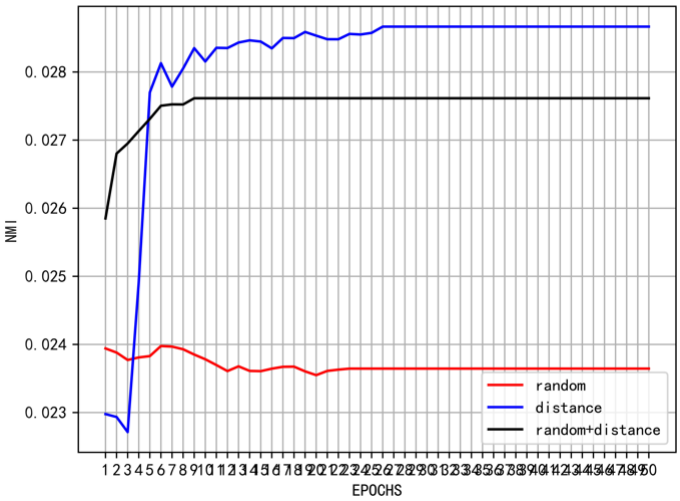
对选用的 5 个数据集分别进行实验，结果绘制成折线图，5 个数据集的结果如下：


0.0105
0.0100
0.0095
0.0090
0.0085
0.0080
NMI
random
distance
random+distance
EPOCHS


0.0105
0.0100
0.0095
0.0090
0.0085
0.0080
NMI
random
distance
random+distance
EPOCHS


0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
NMI
random
distance
random+distance
EPOCHS


0.021
0.020
0.019
0.018
0.017
0.016
0.015
0.014
0.013
NMI
random
distance
random+distance
EPOCHS



图上 红色、蓝色、黑色折线分别代表使用 **random、 distance、 random+distance** 方法初始化后训练得到的 NMI值，纵轴是 epochs，表示训练轮数。

对不同的数据集，不同的初始化方法体现的结果差异比较明显，但可以看出的是，随着训练轮数（epochs）的增加，NMI结果是逐步趋于稳定的，大多数情况下，NMI的值是曲折上升的，但在某些数据集下NMI也会随着训练轮次的增加而下降，这可能是 **选择数据集的聚类效果不明显** 所导致的。

如果我们认为，随着训练轮次增加，NMI稳步上升的模型，可以说明该训练集的聚类效果较好，那么对于 第1数据集、第2数据集、第5数据集，可以认为其聚类效果较好，在这种情况下，使用 **distance** 方法可以达到更好的效果。

对于 **distance** 方法，我们一开始就选择聚类中心彼此距离更远、就更容易区分不同的样本，很容易得出这种方法的巧妙和优越性，但是这种方法也有其缺点，就是这使得同一个类的样本间距离会更加大，这将体现在下一个对比实验2 之中。

4.2 对比实验 2

设置不同聚类个数K的情况下，三种不同初始化方法版本的 目标函数 J 随着聚类个数变化的曲线。

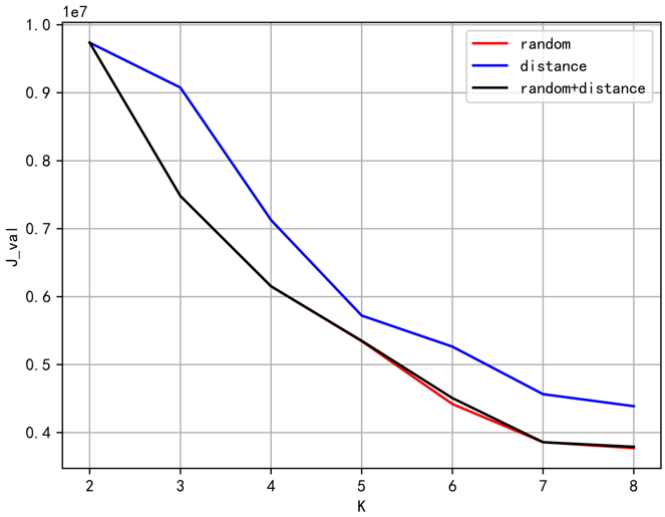
目标函数 J： $J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \left\| \boldsymbol{x}^{(n)} - \boldsymbol{\mu}_k \right\|^2$

目标函数 J 即欧氏距离平方总和，返回该项值即可。

设定参数 epochs 为20，每个模型训练 20 轮。

```
epochs = 20
```

如果使用更大的训练轮数（如使用epochs = 50），折线会过于拟合，难以表现差异性：

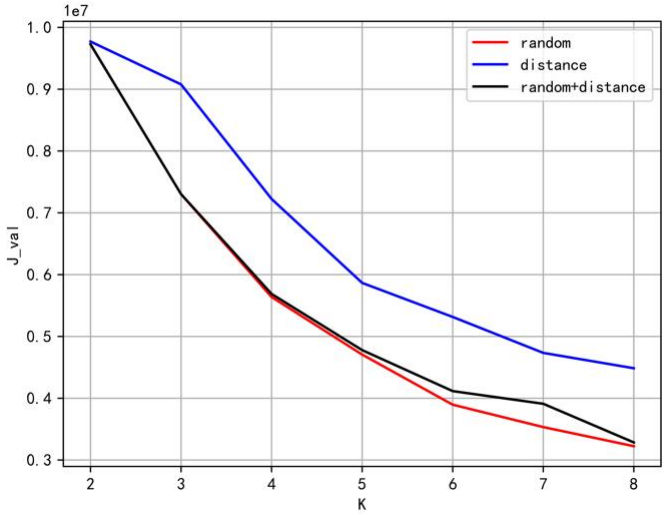


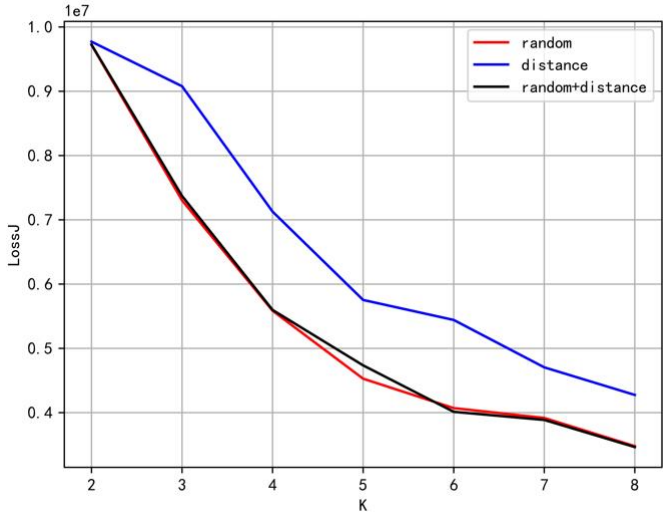
(50轮的训练折线)

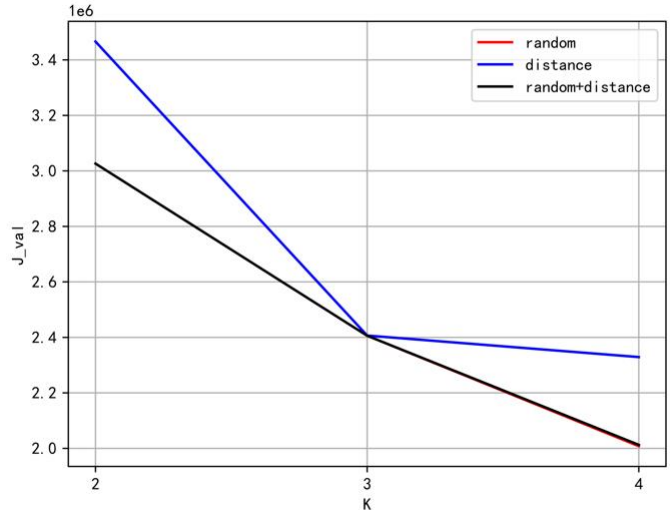
轮流使用三种不同的初始化方法下，从 2 ~ K 轮流设置模型的类的个数（K是正确类个数）

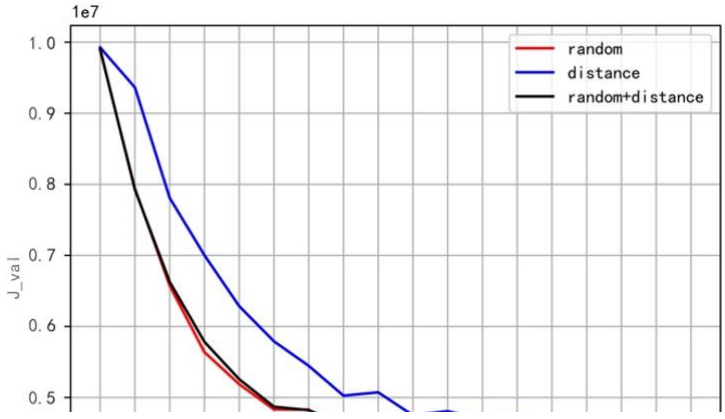
```
method_str = ['random', 'distance', 'random+distance']
color_str = ['red', 'blue', 'black']
for r in range(3):
    # 聚类个数变化在 2~K
    for K_num in range(2, K_true+1):
        train_model = kmeans(data=images, K=K_num, method=method_str[r])
...
```

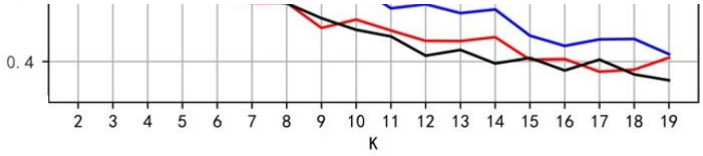
对选用的 5 个数据集分别进行实验，结果绘制成折线图，5 个数据集的结果如下：

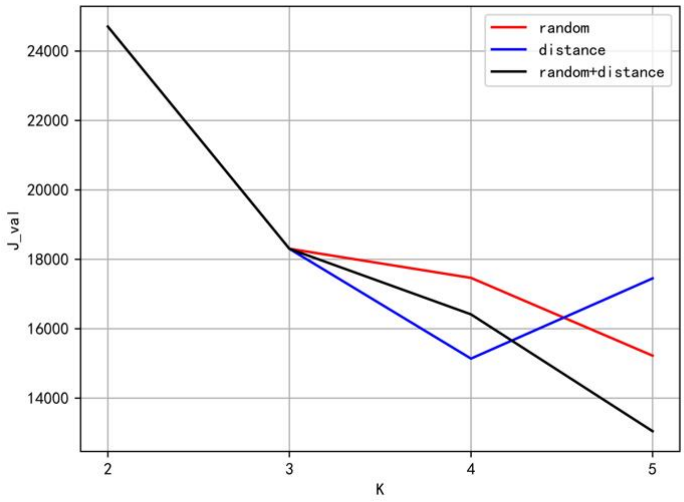

1e7
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
J_val
2
3
4
5
6
7
8
K
random
distance
random+distance


1e7
1.0
0.9
0.8
0.7
0.6
0.5
0.4
LossJ
2
3
4
5
6
7
8
K
random
distance
random+distance


1e6
3.4
3.2
3.0
2.8
2.6
2.4
2.2
2.0
J_val
2
3
4
K
random
distance
random+distance


1e7
1.0
0.9
0.8
0.7
0.6
0.5
J_val
random
distance
random+distance





图上 红色、蓝色、黑色折线分别代表使用 **random、 distance、 random+distance** 方法初始化后训练得到的 目标函数 J 值，纵轴是 K，表示本模型的类的个数。（ 第三幅图中，红线和黑线重叠）

对于所选用的所有数据集，其 目标函数 J 呈现的结果都是，随着分类的个数 K 的增大，J（样本间欧式距离平方和）会越来越小，因为类的个数变多，所以样本能更好地聚拢在一起，所以结果是很容易理解的。

但是对于使用不同聚类中心初始化方法的比较，对大多数据集，**distance**的 目标函数 J 都较大于另外两种方法，这是由于在 **distance** 方法中，初始的聚类中心选择都是相距彼此较远的，这使得后来的更新聚类中心后，同一个类中的样本间距会更加大。

对于 **random** 和 **random + distance** 方法的比较，两种方法差别不大，在 目标函数 J 的大小上，总体都小于 使用 **distance** 方法的。