题目　Clustering

姓名 骆克云 学号 MG1633052 邮箱 streamer.ky@foxmail.com 联系方式18115128082

(南京大学 计算机科学与技术系, 南京　210093)

1　实现细节

1.1　　读取文件，返回数据和标签（projectutil.py）

使用 Python 中的 Pandas 库读取 CSV 文件：将每一行数据按逗号分隔，最后一个作为标签，前面的作为数据

```
def get_handwritten_digits(filename):
  file_path = project_dir + os.sep + "data/Clustering/" + filename + ".txt"
  handwritten_digits = pd.read_csv(file_path, header=None)
  label_column = handwritten_digits.columns[-1]
  label = handwritten_digits[label_column]
  handwritten_digits = handwritten_digits.drop(label_column, axis=1)
  return handwritten_digits, label
```

1.2　　KMedoids算法实现(assignment3/kmedoids.py,score.py)

K-中心聚类是基于 Kmeans 聚类算法的改进，相关算法如下：

1）初始化(__init__)：读取文件，真实标签：true_label，分类数：c，分类数对应的标签：label, 最大迭代次数:max_iterator，计算距离矩阵(scipy cdist 函数加速，曼哈顿距离):distance，重复次数:repeat,算法名：alg;

2）重初始化(reinit)：被谱聚类调用，传入新的数据，重新计算任意两点间的距离；

3）取样(init_centers)：初始化 c 个样本作为中心点,并分配标签：centers, labels;

4）指派新的标签(assign_label)：对所有点到各个中心点的距离排序，取最短的中心点的标签作为点新的标签，将所有数据重新划分成 c 个类；

5）类内代价(cost)：计算一个类中某一点到其他点的曼哈顿距离和；

6）聚类代价(manhadun_cost)：计算所有类的类内代价求和；

7）交换数据(exchange_value)：在一个类中，选取一个点和该类的中心点交换；

8）交换所有值，替换中心点(exchange_all)：对于所有的类，每一次选取该类中的所有点逐个替换中心点，如果类内代价减小，则实施替换，否则撤销替换。所有的替换都完成后，指派新的标签，返回聚类代价；

9）随机交换部分值，替换中心点(exchange_random)：对于所有的类，每一次选取该类中的一部分随机点逐个替换中心点，如果类内代价减小，则实施替换，否则撤销替换。所有的替换都完成后，指派新的标签，返回聚类代价；

10）交换算法(exchange)：根据类的 alg 选择使用随机交换还是全部交换算法， 默认为随机交换；

11）聚类效果量化评分(score.py/score):计算聚类后的聚类效果，clusters_true ：真实标签，clusters_alg ：算法计算出来的标签，Ni: 本身为第 i 类实际被分到第 j 类，Mj: 实际被分到第 j 类本身为第 i 类，Pj: 主导类，Gj: Gini 系数，纯度计算公式为：purity_value = 1.0 * sum(Pj[j] for j in clusters_alg) / sum(Mj[j] for j in clusters_alg)， gini 系数计算公式为：gini_value = 1.0 * sum(Gj[j]*Mj[j] for j in clusters_alg) / sum(Mj[j] for j in

clusters_alg)；

12）Kmedoids 算法(kmedoids_alg): 计数 count=0,首先根据中心点分配标签，然后循环：计算老的聚类代价 old_cost，运行交换算法，得到新的聚类代价 new_cost，当 new_cost >= old_cost 或 count > .max_iterator 时循环停止，进行聚类效果量化评分，否则 count 自增 1；

13）主方法(main_func)：初始化中心点，运行 Kmedoids 算法；

14）多核并发运行(multi_core):使用 multiprocessing 进程池 Pool 和重复次数 repeat 并发运行主方法；

15）Run: 如果重复次数为 0 则运行主方法，否则运行多核并发方法。

伪代码：

Algorithm 1 K-Medoids 核心算法伪代码

输入:

真实标签: true_label; 分类数: c; 分类数对应的标签: label; 最大迭代次数:max_iterator
计算距离矩阵 (scipy cdist 函数加速，曼哈顿距离):distance; 重复次数:repeat; 算法名: alg;

输出:

聚类准确度: 纯度和基尼系数;

1: init all values in class;
2: **if** $repeat \neq 0$ **then**
3: $self.multicore(main\_func)$;
4: **end if**

```
def multi_core(self):
        pool = Pool()
        pool.map(self.main_func, range(self.repeat))
        pool.close()
        pool.join()

def main_func(self, thread=0):
        centers, labels = self.init_centers(thread)
        self.kmedoids_alg(centers, labels, thread)
```

5: $count = 0$;
6: self.assign_label(centers, labels);
7: **repeat**
8: old_cost = self.manhadun_cost(centers, labels)
9: new_cost = self.exchange(centers, labels, alg)

```
def exchange(self, centers, labels, alg="random"):
        if alg == "random":
            return self.exchange_random(centers, labels)
        return self.exchange_all(centers, labels)
```

10: count += 1
11: **until** $new\_cost \geq old\_cost$ *or* $count > self.max\_iterator$
12: purity_value, gini_value = score(labels, self.true_label)
13: **return** purity_value, gini_value

Algorithm 2 随机交换替换中心点算法

输入:

中心点：centers；标签：labels；随机率：r_ration；

输出:

聚类代价;

```
for each label ∈ self.label do
  indexes = labels[labels == label].index ;
  selected_indexes = random.sample(indexes.tolist(), int(r_ration*len(indexes)));
  for each point_index ∈ selected_indexes do
    old_cost = self.cost(centers, labels, label)
```

```
def cost(self, centers, labels, label):
      return sum(self.distance[centers[label]][point_index] for point_index in
          labels[labels == label].index)
```

```
    old_center = centers[label]
    centers = self.exchange_value(old_center, point_index, centers, labels)
```

```
def exchange_value(self, X_Old, X_New, centers, labels):
      XOld_label = labels[X_Old]
      labels[X_Old] = labels[X_New]
      labels[X_New] = XOld_label
      centers[XOld_label] = X_New
      return centers
```

```
    new_center = centers[label]
    new_cost = self.cost(centers, labels, label)
    if new_cost > old_cost then
      centers = self.exchange_value(new_center, old_center, centers, labels)
    end if
  end for
end for
labels = self.assign_label(centers, labels)
```

```
def assign_label(self, centers, labels):
      center_indexes = centers.values.tolist()
      for point_index in labels.index:
         index = center_indexes[np.argmin(self.distance[point_index][center_indexes])]
         labels[point_index] = labels[index]
      return labels
```

```
return self.manhadun_cost(centers, labels)
```

1.3 Spectral谱聚类（assignment3/spectralclustering.py）

把数据映射到一个新空间，该空间里具有约化的维度，使得相似性更加显而易见，然后对新空间的数据进行聚类。

基本步骤：

1） 初始化(__init__)：读取文件，真实标签：true_label，分类数：k，计算距离矩阵(scipy cdist 函数加速，曼哈顿距离):distance，近邻数:knn;

2） 构建带权图相似度矩阵 W：使用 k 近邻算法，对每个点选取前 k 个邻居为 1，其余为 0，并对称化该矩阵；

3） 构建拉普拉斯图矩阵 L：构建 W 的对角元素矩阵 D($d_i = \sum_iW_{ij}$), L = I - D^(-0.5)WD^(-0.5)

4） 特征值分解，得到新的数据：eig_values, eig_vectors = np.linalg.eigh(L)，将特征值排序，选取特征值最小的 k 个对应的特征向量列组成显得数据；

5） 调用 KMedoids 算法，重新初始化 KMedoids 算法中的距离矩阵，运行 KMedoids 算法，得出结果。

伪代码：

Algorithm 3 Spectral-Clustering 核心算法伪代码

输入:

真实标签: true_label; 分类数: k; 计算距离矩阵 (scipy cdist 函数加速, 曼哈顿距离):distance; 近邻数:knn;

输出:

聚类准确度: 纯度和基尼系数;

1: init all values in class;

2: build_graph_weights;

```
k = self.knn + 1, N = self.length
W = np.full((N, N), fill_value=0, dtype=float)
for i in range(N):
    index_sort = np.argsort(self.distance[i])
    for j in index_sort[:k]:
        W[i, j] = 1, W[i, i] = 0
return np.maximum(W, W.T)
```

3: build_matrix

```
W = self.build_graph_weights(), N = self.length
D = np.full((N, N), fill_value=0, dtype=float)
for i in range(N):
    D[i, i] = sum(W[i][j] for j in range(N))
    D_sym = np.full((N, N), fill_value=0, dtype=float)
    for i in range(N):
        D_sym[i, i] = np.power(D[i, i], -0.5)
    L_sym = np.eye(N) - D_sym.dot(W).dot(D_sym)
return L_sym
```

4: laplas

```
L = self.build_matrix()
eig_values, eig_vectors = np.linalg.eigh(L)
indexes = np.argsort(eig_values)
eig_vectors = eig_vectors[:, indexes]
return eig_vectors[:, :self.k]
```

5: run_kmedoids

```
L = self.laplas()
kmedoids = KMedoids(self.filename, repeat=4, alg="Spectral-Clustering-knn-%d" % self.knn)
kmedoids.reinit(L)
kmedoids.run()
```

6: **return** purity_value, gini_value

1.4 运行

对于每个文件（"german","mnist"），分别运行 KMedoids(repeat=10,重复 10 次),Spectral(k=3,6,9，调用 kmedoid

重复 4 次)得出结果。

```
def run():
  for file in ["german","mnist"]:   # , "mnist" "german",
      kmedoids = KMedoids(file, repeat=10)
      kmedoids.run()
      spectral = Spectral(file,knn=3)
      spectral.run()
      spectral = Spectral(file,knn=6)
      spectral.run()
      spectral = Spectral(file,knn=9)
      spectral.run()
```

2 结果

2.1 实验设置

数据来源：Clustering,包含如下文件：german.txt，mnist.txt

数据预处理：projectutil.py 中 get_handwritten_digits(filename)可给定文件名返回数据及标签。

2.2 实验结果

1. 正确率比较

Purity of different algorithms

	k-medoids	Spectral(n=3)	Spectral(n=6)	Spectral(n=9)
german.txt	0.7	0.7	0.7	0.7
mnist.txt	0.514600	0.720500	0.762200	0.737700

Gini index of different algorithms

	k-medoids	Spectral(n=3)	Spectral(n=6)	Spectral(n=9)
german.txt	0.407687	0.419239	0.417718	0.418907
mnist.txt	0.639545	0.354718	0.340989	0.353832

2. 结果反思

Spectral 聚类方法总体好于 k-medoids，并且 Spectral 聚类方法的好坏并不随 k 近邻的大小有直接的相关性。k-medoids 算法运行缓慢，通常为了跳出局部最优需要重复运行多次，花费时间特别长，因此使用了多个进程同时运行 k-medoids 算法。

注意：

1. 最终提交的报告最好保存为 pdf 格式
2. 压缩格式为 zip 格式，请勿使用需要安装特定软件才能打开的压缩方式
3. 作业的文件夹目录请按照网页要求，代码、结果放在不同子文件夹中。作业网页上给出的数据不需要再次提交