

K-means算法实验报告

201250205 郭慕凡

K-means算法实验报告

1.基础实现

1.1 k-means算法原理

1.2 详细步骤说明

1.2.1 输入图像预处理：归一化

1.2.2 k-means算法实现

1.3 运行结果及截图

1.3.1 输入可视化

1.3.2 输出可视化 k=2

1.3.3 输出可视化 k=5

1.3.4 性能可视化

2.拓展实现1：gpu并行计算

2.1 gpu代码

2.2 性能对比：

3.拓展实现2：k-mean++算法

1.基础实现

1.1 k-means算法原理

k-means算法是无监督学习中的一种“基于原型的聚类”算法。基于原型的聚类算法指的是聚类的结构可以通过一组原型刻画，k-means算法即假设数据分布的结构为k个有中心的聚类簇。

给定样本集：

$$D = \{x_1, x_2, x_3, \dots, x_m\}$$

k-means算法针对聚类的簇划分

$$C = \{C_1, C_2, \dots, C_k\}$$

最小化平方误差：

$$E = \sum_{i=1}^k \sum_{x \in C_i} \|x - \mu_i\|_2^2$$

其中，

$$\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$$

为聚类 C_i 的均值向量（中心或“质心”），最小化误差 E 即追求最大的簇内相似度。

由于最小化该式理论上需要考虑样本集的所有可能划分，这是一个NP难问题，因此我们采取动态划分，迭代优化动态求解该式。基本思想是：首先选择 k 个样本点作为聚类中心，再利用距离最小化原则，使样本点向中心聚集；判断初始分类是否合理，若不合理，则修改聚类。

具体的算法步骤可以表达如下：

1. 确定聚类数量k
2. 随机取k个样本点，将它们视作k个聚类的初始中心
3. 对每个样本点，计算样本点到k个聚类中心的距离，将样本点分到距其距离最小的聚类中心所属的聚类中
4. 更新聚类中心，更新值为3后每个聚类的样本均值
5. 重复2-4，直至聚类中心不再变化

这是一个贪心算法。

1.2 详细步骤说明

1.2.1 输入图像预处理：归一化

```
# 读入图片
image = imageio.imread("./input.png")
```

```
# 图片预处理
def pre_process(image):
    # 1. squeeze the image to [-1, 3]
    input = np.reshape(image, [-1, 3])
    input = input/255
    return input
```

首先将输入图像从[w, h, 3]的三维数组转换为[l, 3]的二维数组，方便统一处理，再将RGB值从[0, 255]归一化到[0, 1]，避免可能的数值溢出问题

1.2.2 k-means算法实现

代码说明：使用python的第三方库：imageio, numpy

1. 初始化k个随机整数作为初始聚类中心

```
def init_k(length, k):
    ...
    @param: length: 数组长度
    @param: k: 聚类数
    @return: 长度为k的数组
    ...
    init_nums = np.random.randint(0, length-1, k)
    return init_nums
```

2. 计算距离：定义聚类为RGB向量差的2范数（RGB向量的欧式距离）

```
def dist(i, j):
    ...
    @param i,j: RGB vector
    ...
    return np.power(i-j, 2).sum()/3 #利用numpy的广播机制
```

3. 更新样本所属聚类及样本中心

```
def update(input, centers, k, length, belongs):
    count = np.ones(k)
    colors = np.zeros([k, 3])+centers
```

```

"""计算最小距离"""
for i in range(0, length):
    min_dist = 100
    min_cnum = 0
    for j in range(0, k):
        if dist(input[i], centers[j]) < min_dist:
            min_dist = dist(input[i], centers[j])
            min_cnum = j
    count[min_cnum] += 1
"""更新所属聚类"""
belongs[i] = min_cnum
for w in range(0, 3):
    colors[min_cnum][w] += input[i][w]
"""更新聚类中心"""
for r in range(0, k):
    colors[r] = colors[r] / count[r]
return colors

```

4. 如何判断循环终止？利用epsilon

在实际操作中，想让更新前后的中心值完全相等是很困难的，而且当误差足够小的时候，继续精化的边际效益较低，整体算法效率较低，因此我们通过设置一个epsilon值，当更新前后满足：

$$\sum_{i=1}^k \|C_i - \hat{C}_i\|_2^2 < \epsilon$$

我们即可判断终止，跳出循环

epsilon的取值为多少比较合适？：当RGB的每个分量差值为1时，在视觉上基本一致，由于在输入预处理时我们将RGB的值压缩到了[0, 1]，对应差值为1/255 约为0.004；此时上式误差约为1e-4，因此取epsilon为1e-4

1.3 运行结果及截图

1.3.1 输入可视化

选取南大北大楼的经典图片，图片参数为

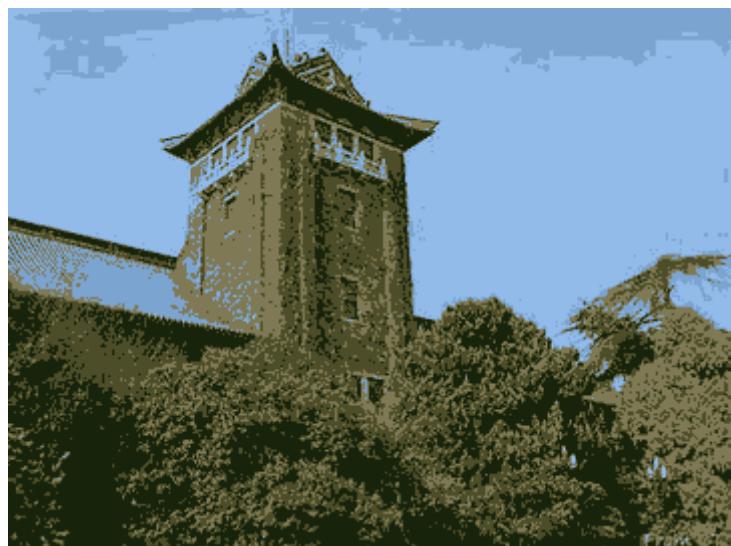


1.3.2 输出可视化 k=2

(每个聚类的颜色为所有样本点颜色的均值)



1.3.3 输出可视化 k=5



1.3.4 性能可视化

k=2

```
loop: 0 done | loss: 0.4328773267351747 | using time(s): 5.786073684692383
loop: 1 done | loss: 0.0007291451682320771 | using time(s): 5.6664958000183105
total time(s): 17.885477542877197

Process finished with exit code 0
```

k=5

```
loop: 0 done | loss: 0.05272054394193749 | using time(s): 10.747315883636475
loop: 1 done | loss: 0.0021894152461439794 | using time(s): 10.81178879737854
loop: 2 done | loss: 0.00028479448345017366 | using time(s): 10.557894468307495
total time(s): 43.55551528930664

Process finished with exit code 0
```

2.拓展实现1：gpu并行计算

2.1 gpu代码

上述算法的性能较差，在 $k=5$ ，图片像素为90000时，分类一次要43s左右，如果面对 $k=10$ 甚至 $k=100$ ；图片分辨率更高的情况，那么算法的执行时间将是无法忍受的。

上述算法的性能瓶颈主要在于循环遍历每个样本点计算距离并更新所属聚类，而这正适用于gpu并行计算来加快

这里采用python的第三方库numba.cuda，硬件需要为nvidia显卡，选用该库的原因是其封装较好，不需要写c语言的硬件代码

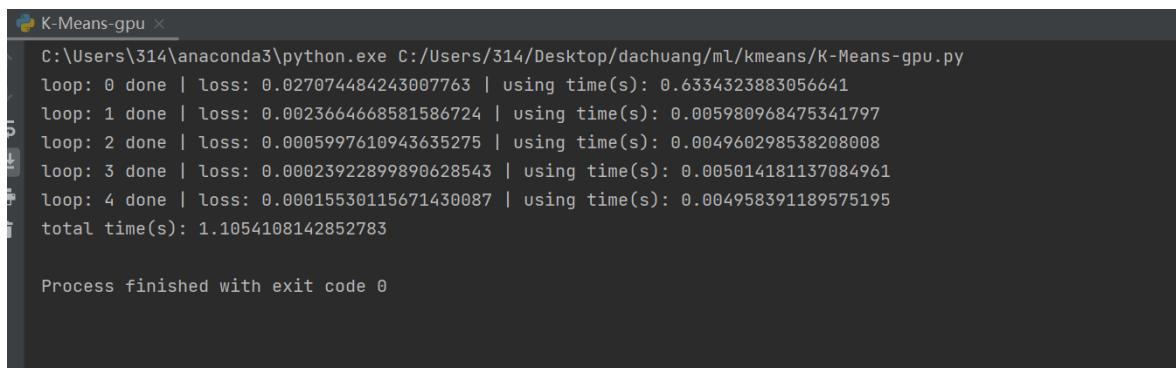
代码更新：

```
@cuda.jit() #调用cuda的标记
def classify(input, centers, k, length, belongs, relations):
    i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x #并行索引
    if i < length:
        min_dist = 100
        min_cnum = 0
        for j in range(0, k):
            if dist(input[i], centers[j]) < min_dist:
                min_dist = dist(input[i], centers[j])
                min_cnum = j
        relations[min_cnum][i] = 1
        belongs[i] = min_cnum

@cuda.jit(device=True) #硬件函数
def dist(i, j):
    result = 0
    for w in range(0, 3):
        result += pow(i[w]-j[w], 2)
    return result
```

2.2 性能对比：

$k=5$



```
K-Means-gpu x
C:\Users\314\anaconda3\python.exe C:/Users/314/Desktop/dachuang/ml/kmeans/K-Means-gpu.py
loop: 0 done | loss: 0.027074484243007763 | using time(s): 0.6334323883056641
loop: 1 done | loss: 0.0023664668581586724 | using time(s): 0.005980968475341797
loop: 2 done | loss: 0.0005997610943635275 | using time(s): 0.004960298538208008
loop: 3 done | loss: 0.00023922899890628543 | using time(s): 0.005014181137084961
loop: 4 done | loss: 0.00015530115671430087 | using time(s): 0.004958391189575195
total time(s): 1.1054108142852783

Process finished with exit code 0
```

与1.3.4中展示的 $k=5$ 的运算时间缩短至2.5%

甚至，可以快速计算 $k=100$

$k=100$:

```
loop: 39 done | loss: 0.0001879952555585555 | using time(s): 0.0877575602722108
loop: 40 done | loss: 0.00019675518631042727 | using time(s): 0.08577084541320801
loop: 41 done | loss: 0.00013855634181278475 | using time(s): 0.08776545524597168
loop: 42 done | loss: 0.00013158217738876802 | using time(s): 0.0858011245727539
loop: 43 done | loss: 0.00012453134095022882 | using time(s): 0.0877225399017334
loop: 44 done | loss: 0.00010712493193150362 | using time(s): 0.08973288536071777
total time(s): 4.822860956192017
```

```
Process finished with exit code 0
```

k=100效果图：



原图：



可见，在我采用：每个聚类的颜色为所有样本点颜色的均值时，k=100与原图已经十分接近了

3.拓展实现2：k-mean++算法

进一步尝试了一下k-means++算法，算法实现如下：

```
def init_kpp(length, k, input_device):
    inits = np.zeros(k).astype(int)
    init_first = np.random.randint(0, length - 1)
    """首先初始化一个结果"""
    inits[0] = init_first
    """循环得到k-1个结果"""
    for i in range(1, k):
```

```

for i in range(1, k):
    """首先计算概率"""
    possibilities = np.zeros(length)
    possibilities = cuda.to_device(possibilities)
    """gpu函数代替循环"""
    cal_p[1024, 1024](input_device, inits, length, possibilities, i)
    cuda.synchronize()
    possibilities = possibilities.copy_to_host()
    sum = possibilities.sum()
    possibilities /= sum
    """ 排序 """
    possibilities = np.sort(possibilities)
    """通过轮盘算法选出新的中心"""
    r = np.random.randint(0, 1)
    new = RWS(possibilities, r)
    inits[i] = k

```

其中，轮盘法实现如下：

```

def RWS(possibilities, r):
    """轮盘算法"""
    q = 0 # 累计概率
    for i in range(1, possibilities.shape[0] + 1):
        q += possibilities[i - 1] # P[i]表示第i个个体被选中的概率
        if r <= q: # 产生的随机数在m~m+P[i]间则认为选中了i
            return i

```