k-means clustering k均值聚类

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming, trimester 1, 2019

```
In [2]: from collections import namedtuple, defaultdict
        from math import hypot
        import matplotlib.pyplot as plt
```

坐标

A point on the plane is defined by its x- and y-coordinates; it can therefore be represented by a 2-element list or tuple, but it is more elegant to take advantage of the `namedtuple` class from the `collections` module. The point coordinates can be referred to thanks to a more natural syntax, though indexes can still be used:

平面上的点由其x坐标和y坐标定义；因此，它可以表示为一个2元素的列表或元组，但是利用来自的namedtuple类更优雅收藏模块。 不过，由于更自然的语法，可以参考点坐标索引仍然可以使用

```
In [3]: Point = namedtuple('Point', 'x y')
        # Alternatively:
        # Point = namedtuple('Point', ['x', 'y'])

        pt = Point(3, 5.5)

        pt
        pt.x, pt[0]
        pt.y, pt[1]
```

```
Out[3]: Point(x=3, y=5.5)

Out[3]: (3, 3)

Out[3]: (5.5, 5.5)
```

给定正整数k，k-聚类是一种将平面上的点分组为最多k个聚类的技术，从k个给定点开始，质心
每个质心C与最接近C的点集合SC相关联，而不是与任何其他质心相关联。 如果一个点最接近一个以上的质心，则可以任意选择一个。 质心可以使得每个点最接近另一个质心，在这种情况下，如此创建的非空集的数量小于k。

对于具有C初始质心的SC形式的每个非空集，计算SC的重心并且成为新的质心。 使用新质心而不是原始质心重复该过程

该过程可能一次又一次地重复，直到阶段结束时计算的质心与开始时使用的质心相同的阶段。 与最终质心之一相关联的每组点是最终聚类。

Given a positive integer k, k-clustering is a technique to group points on the plane into at most k clusters, starting with k given points, the centroids.

- Each centroid C is associated with the set S_C of points that are closest to C than to any other centroid. In case a point is closest to more than one centroid, one is chosen arbitrarily. A centroid can be such that every point is closest to another centroid, in which case the number of nonempty sets so created is smaller than k.
- For each nonempty set of the form S_C with C an initial centroid, the centre of gravity of S_C is computed and becomes a new centroid. The procedure is repeated using the new centroids instead of the original ones.
- The procedure is possibly repeated again and again, up to the stage where the centroids computed at the end of the stage are the same as those used at the beginning. Each set of points associated with one of the final centroids is a final cluster.

比较距离相当于比较距离的平方，因此我们选择更经济的计算并实现一个函数square_of_distance（）来计算后者。 如果我们必须计算距离，我们可以使用math模块中的hypot（）函数：

Comparing distances is equivalent to comparing squares of distances, so we opt for a more economical computation and implement a function, `square_of_distance()`, to compute the latter. If we had to compute distances, we could use the `hypot()` function from the `math` module:

```
In [4]: def square_of_distance(point_1, point_2):
            return (point_1.x - point_2.x) ** 2 + (point_1.y - point_2.y) ** 2

        pt_1 = Point(65, 82)
        pt_2 = Point(52, 93)
        square_of_distance(pt_1, pt_2)
        hypot(pt_1.x - pt_2.x, pt_1.y - pt_2.y) ** 2

Out[4]: 290

Out[4]: 290.00000000000006
```

Let us define a number of points and centroids:

```
In [5]: points = {Point(20, 40), Point(20, 42), Point(20, 44), Point(20, 46),
                  Point(20, 46), Point(20, 52), Point(20, 90), Point(38, 85),
                  Point(83, 95), Point(53, 87), Point(39, 98), Point(44, 73),
                  Point(65, 82), Point(52, 93), Point(63, 34), Point(71, 27)
                 }
        centroids = {Point(20, 95), Point(20, 5), Point(52, 83),
                     Point(73, 33), Point(85, 38)
                    }
```

To compute the member of `centroids` that is closest to a given point P, one can start with the square of the distance from P to some centroid, and update that value whenever a new centroid C is processed and the square of the distance between C and P is found out to be smaller than the smallest value recorded so far:

为了计算最接近给定点P的质心成员，可以从P到某个质心的距离的平方开始，并且每当处理新的质心C并且C和C之间的距离的平方时更新该值。 发现P小于目前记录的最小值：

```
In [6]: point = Point(70, 80)
        centroids_list = list(centroids)
        minimal_squared_distance = square_of_distance(point, centroids_list[0])
        print('Smallest squared distance so far:', minimal_squared_distance)
        print('Closest centroid so far:', centroids_list[0])
        for centroid in centroids_list[1: ]:
            squared_distance = square_of_distance(point, centroid)
            if squared_distance < minimal_squared_distance:
                minimal_squared_distance = squared_distance
                print('\nSmallest squared distance now:', minimal_squared_distance)
                print('Closest centroid now:', centroid)

Smallest squared distance so far: 2218
Closest centroid so far: Point(x=73, y=33)

Smallest squared distance now: 1989
Closest centroid now: Point(x=85, y=38)
```

```
Smallest squared distance now: 333
Closest centroid now: Point(x=52, y=83)
```

更好的方法是使用足够大的值初始化minimal_squared_distance；float(`inf')非常适用于此目的（nan代表“不是数字”；实际上，减去无穷大加无穷大是未定义的，而所有其他计算都是定义的）：

A better approach is to initialise `minimal_squared_distance` with a large enough value; `float(`inf')` is perfect for that purpose (`nan` stands for "not a number"; indeed, minus infinity plus infinity is undefined, whereas all other computations are defined):

```
In [7]: float('-inf') <= float('-inf') < -(10 ** 300) <\
                        10 ** 300 < float('inf') <= float('inf')
        min(float('inf'), 10 ** 3)
        float('inf') - 20
        float('-inf') * -5
        float('-inf') + float('inf')

Out[7]: True

Out[7]: 1000

Out[7]: inf

Out[7]: inf

Out[7]: nan
```

此外，min()函数可以系统地应用，而无需测试是否minimal_squared_distance需要更改为一个新值：

Also, the `min()` function can be applied systematically without testing whether `minimal_squared_distance` needs to be changed to a new value:

```
In [8]: point = Point(70, 80)
        minimal_squared_distance = float('inf')
        for centroid in centroids:
            minimal_squared_distance = min(minimal_squared_distance,
                                           square_of_distance(point, centroid)
                                          )

        minimal_squared_distance

Out[8]: 333
```

We need to eventually know, for a given point P, which centroid C is closest to P rather than what is the minimal squared distance between P and some centroid; so we still use an `if` statement in the code fragment below:

我们需要最终知道，对于给定的点P，哪个质心C最接近P而不是P与某些质心之间的最小平方距离；所以我们仍然在下面的代码片段中使用if语句：

```
In [9]: clusters = defaultdict(set)
        for point in points:
            min_squared_distance = float('inf')
            for centroid in centroids:
                squared_distance = square_of_distance(point, centroid)
                if squared_distance < min_squared_distance:
```

```
                    min_squared_distance = squared_distance
                    closest_centroid = centroid
            clusters[closest_centroid].add(point)

    clusters
```

```
Out[9]: defaultdict(set,
                    {Point(x=20, y=95): {Point(x=20, y=52),
                      Point(x=20, y=90),
                      Point(x=39, y=98)},
                     Point(x=52, y=83): {Point(x=38, y=85),
                      Point(x=44, y=73),
                      Point(x=52, y=93),
                      Point(x=53, y=87),
                      Point(x=65, y=82),
                      Point(x=83, y=95)},
                     Point(x=20, y=5): {Point(x=20, y=40),
                      Point(x=20, y=42),
                      Point(x=20, y=44),
                      Point(x=20, y=46)},
                     Point(x=73, y=33): {Point(x=63, y=34), Point(x=71, y=27)}})
```

For every member `centroid` of `centroids`, if `centroid` is one of `clusters`'s keys then `clusters[centroid]` is a nonempty set of points whose centre of gravity should be computed. It will be used in place of `centroid`, but both could be identical:

```
In [10]: centroid = Point(x = 20, y = 95)

         clusters[centroid]
         [c for c in zip(*clusters[centroid])]
         x, y = [sum(c) for c in zip(*clusters[centroid])]; x, y
         Point(x / len(clusters[centroid]), y / len(clusters[centroid]))
```

```
Out[10]: {Point(x=20, y=52), Point(x=20, y=90), Point(x=39, y=98)}
```

```
Out[10]: [(20, 20, 39), (90, 52, 98)]
```

```
Out[10]: (79, 240)
```

```
Out[10]: Point(x=26.333333333333332, y=80.0)
```

Going one step further, we can create a new dictionary similar to `clusters`, except that for any cluster C that is one of `clusters`'s values, the key K in `clusters` that is associated with C is replaced by C's centre of gravity (which might be no different to K). Such is the purpose of the following function, which starts with the previous code fragment, then computes the new centroids and redefines `clusters` accordingly, and eventually returns both `clusters` and `True` or `False` depending on whether the set of new centroids is different to the one passed as argument:

更进一步，我们可以创建一个类似于集群的新字典，但对于作为集群值之一的任何集群C，与C关联的集群中的密钥K将被替换为 C的重心（可能与K没有什么不同）。 这就是以下函数的目的，它从前面的代码片段开始，然后计算新的质心并重新定义集群因此，最终返回两个簇和True或False，具体取决于新质心的集合是否与作为参数传递的质心不同

```
In [11]: def cluster_with(centroids):
             clusters = defaultdict(set)
             for point in points:
                 min_squared_distance = float('inf')
                 for centroid in centroids:
                     squared_distance = square_of_distance(point, centroid)
                     if squared_distance < min_squared_distance:
                         min_squared_distance = squared_distance
                         closest_centroid = centroid
                 clusters[closest_centroid].add(point)
             new_to_old = {}
             for centroid in clusters:
                 nb_of_points = len(clusters[centroid])
                 x, y = [sum(c) for c in zip(*clusters[centroid])]
                 new_to_old[Point(x / nb_of_points, y / nb_of_points)] = centroid
             return {centroid: clusters[new_to_old[centroid]]
                                     for centroid in new_to_old
                    }, any(new_to_old[centroid] != centroid
                                     for centroid in new_to_old
                           )

         cluster_with(centroids)

Out[11]: ({Point(x=26.333333333333332, y=80.0): {Point(x=20, y=52),
            Point(x=20, y=90),
            Point(x=39, y=98)},
           Point(x=55.833333333333336, y=85.83333333333333): {Point(x=38, y=85),
            Point(x=44, y=73),
            Point(x=52, y=93),
            Point(x=53, y=87),
            Point(x=65, y=82),
            Point(x=83, y=95)},
           Point(x=20.0, y=43.0): {Point(x=20, y=40),
            Point(x=20, y=42),
            Point(x=20, y=44),
            Point(x=20, y=46)},
           Point(x=67.0, y=30.5): {Point(x=63, y=34), Point(x=71, y=27)}},
          True)
```

现在让我们定义一个显示聚类的函数。 matplotlib.pyplot模块中的subplot（）函数有三个参数：

Let us now define a function to display the clusters. The `subplot()` function from the `matplotlib.pyplot` module takes three arguments:

- a strictly positive integer meant to denote a number of rows n_R, 一个严格正整数，意味着表示行数nR
- a strictly positive integer meant to denote a number of columns n_C, and 一个严格正整数，意味着表示多个列nC
- an integer between 1 and $n_R \times n_C$ meant to denote an index n. 1和nR之间的整数×nC意味着表示索引n

`subplot()` returns a so-called **axes**, meant to refer to part of the figure to draw: representing that picture as a grid of size n_R by n_C, the returned axes is supposed to occupy the nth cell of the grid. For

subplot（）返回一个所谓的轴，意思是指要绘制的图形的一部分：将该图像表示为大小为nR，nC的网格，返回的轴应该占据网格的第n个单元格。

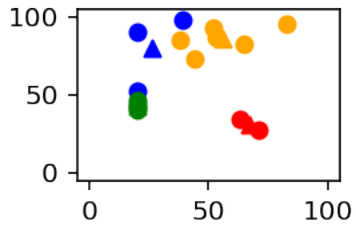
例如，如果nR，nC和n分别是3,3和2，那么返回的轴应该占据顶行中间的单元。

instance, if n_R, n_C and n are 3, 3 and 2, respectively, then the returned axes is supposed to occupy the cell in the middle of the top row.

When all three arguments are single digits, they can also be "glued" as a single integer or a single string, so 3, 3 and 2 can also be glued as `332` or `'332'`. We intend to first draw the original points, all in one colour, and all original centroids, all in another colour. We then intend to compute clusters and new centroids stage by stage, until the centroids do not change, and display every new set of clusters; such is the purpose of function `plot_clusters()`, whose first argument is meant to denote a dictionary of the kind returned by `cluster_with()`, and whose second argument is meant to denote an axes's index. The purpose of `plot_clusters()` is to draw the points, using one colour per cluster, as well as the associated centroids, using a different shape but the same colour as that of the associated cluster. We limit the number of computation stages to 8 at most, so we use a grid of size 3 by 3, reserve the top left cell of the grid to display the points and original centroids before cluster computation starts, and use some of the 8 remaining cells to display the first and every new set of clusters and associated centres of gravity, starting with the middle cell on the top row, and moving left to right and top to bottom. We limit the number of original centroids to 10 at most, define a sequence of 10 colours, and let `iter()` return an iterator for this sequence so that `next()` can be called and yield a new colour for every new cluster. We set the range of x- and y-values to display along the x- and y axes with `set_xlim()` and `set_ylim()`, respectively. We use the default value of the keyword argument `marker` of `matplotlib.pyplot`'s `scatter()` function to draw the points as disks, while we change it to `'^'` to draw the centres of gravity as triangles; the `c` keyword argument can be made more explicit as `color`. Calling `plot_clusters()` only once does not show how the axes is positioned on a grid of size 3 by 3:

```
In [12]: def plot_clusters(clusters, i):
             colours = iter(('blue', 'orange', 'green', 'red', 'purple',
                             'brown', 'pink', 'gray', 'olive', 'cyan'
                            )
                           )
             ax = plt.subplot(f'33{i}')
             ax.set_xlim(-5, 105)
             ax.set_ylim(-5, 105)
             for centroid in clusters:
                 colour = next(colours)
                 plt.scatter(*centroid, marker = '^', c = colour)
                 plt.scatter(*zip(*clusters[centroid]), c = colour)

         plot_clusters(cluster_with(centroids)[0], 2)
```



The function that follows completes the task of calling `cluster_with()` for long enough, but no more than 8 times, to compute the next set of clusters and associated centres of gravity, and in case the latter

后面的函数完成调用cluster_with（）的任务足够长，但不超过8次，以计算下一组聚类和相关的重心，并且如果后者与上面的质心集不同。 确定聚类集合的基础，在图像的适当部分显示聚类并更新质心。 在不太可能的情况下，原始质心将是首先计算的聚类集合的重心，聚类仍将显示在12英寸×12英寸大小的图片的第一行的中间：

are different to the set of centroids on the basis of which the set of clusters was determined, display the clusters in the appropriate part of the picture and update the centroids. In the unlikely case the original centroids would be the centres of gravity of the set of clusters first computed, the clusters would still be displayed in the middle of the first row of a picture of size 12 inches by 12 inches:

```
In [13]: def iterate_and_plot_clusters(centroids):
             if len(centroids) > 10:
                 print('At most 10 centroids accepted')
                 return
             plt.figure(figsize = (12, 12))
             ax = plt.subplot(331)
             ax.set_xlim(-5, 105)
             ax.set_ylim(-5, 105)
             plt.scatter(*zip(*centroids), marker = '^', color = 'cyan')
             plt.scatter(*zip(*points), color = 'olive')
             for i in range(2, 10):
                 clusters, new_centroids = cluster_with(centroids)
                 if i == 2:
                     plot_clusters(clusters, i)
                 if new_centroids:
                     centroids = clusters.keys()
                     if i > 2:
                         plot_clusters(clusters, i)
                 else:
                     break

         iterate_and_plot_clusters(centroids)
```