数据挖掘 hw1 q5

Careful K-mediod思路概述

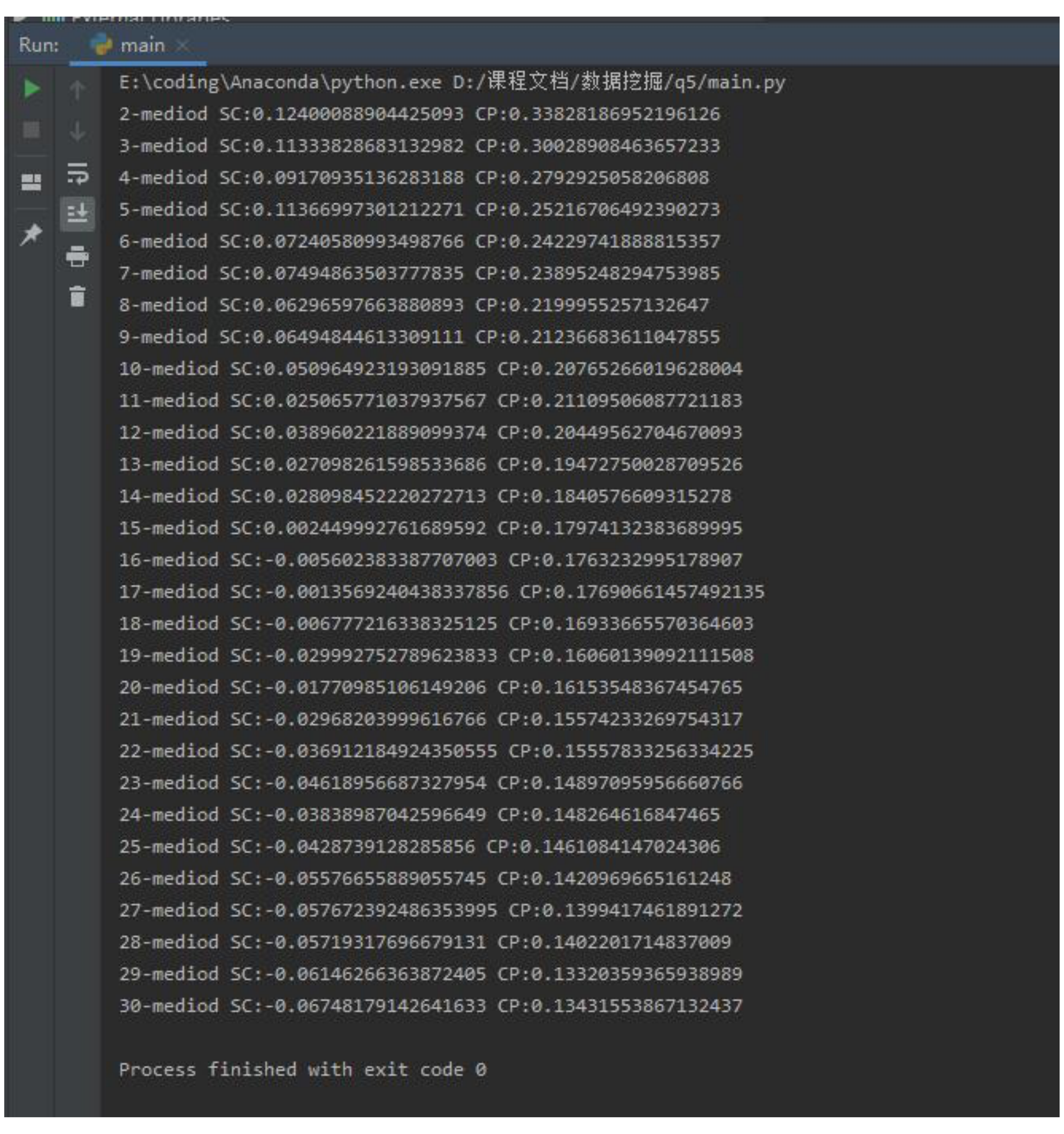
在q3中，我们尝试复现了作者的论文。我认为作者的FTC_tree 具有可行性，虽然距离定义还有点问题，但我并没有想到比作者更好的方法，只是取质心的方式没有逻辑，这也是论文最大的问题。我想到了，在只有距离矩阵的情况下且没有合适的定义质心的方法的情况下， $K-mediod$ 能够很好地代替 $K-means$,采用中位点代替簇内点的平均值，至少能够保证该点是一个离簇内其他点距离之和比较近的点，虽然并不是最优解

代码实现

在保留了 q3 中的FTC_tree 的情况下,实现$K-mediod$ 算法。

In [1]:
```python
def transaction_kmediod(self, dm, k):
    #caereful-k-mediod聚类算法
    # k       - 指定分簇数量
    # dm      - distance_matrix
    n_sample, n_feature = dm.shape  # m: 样本数量, n: 每个样本的属性值个数
    result = np.empty(n_sample, dtype=np.int)  # m个样本的聚类结果
    # cores = np.empty((k, 1))  # k个质心,每个放入

    cores = self.get_k_initial_points(dm, k)

    # 无放回抽取质心
    max_iter = 40
    for itr in range(max_iter + 1):  # 迭代计算
        distance = dm[:, cores]
        index_min = np.argmin(distance, axis=1)  # 每个样本距离最近的质心索引序号

        if (index_min == result).all():
            return result, cores

        result = index_min  # 更新聚类结果
        for i in range(k):  # 遍历质心集
            # 获得这个簇里的点在所有点里的索引
            cluster_index = np.array([s for s in range(n_sample)])[result == i]
            # 找出对应当前质心的子样本集
            items = dm[result == i]
            items = items[:, result == i]
            med_index = cluster_index[np.argmin(np.sum(items, axis=1))]
            cores[i] = med_index
```

其中对于 $get\_k\_initial\_points$, 我们采用先采用随机取一个点，后每个初始质心取距离已有质心距离之和最远的点的$K-mediod++$\ 随后采用自定义的一套规则，观察SC和CP变化情况\ 对于每个k，取50次平均，观察SC和CP变化情况。



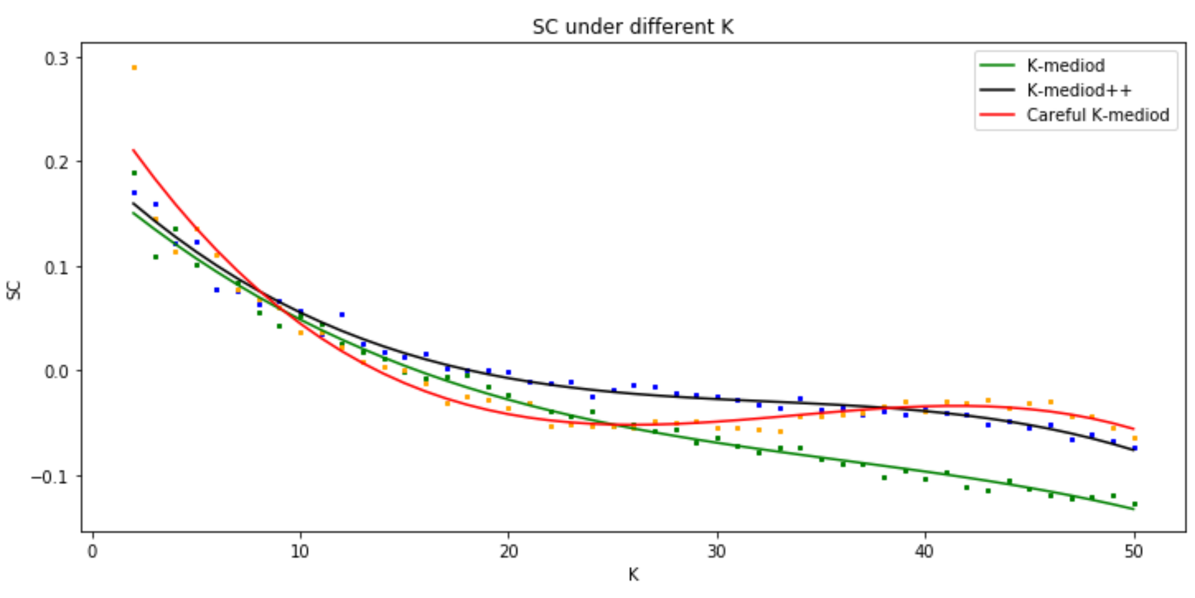
```
E:\coding\Anaconda\python.exe D:/课程文档/数据挖掘/q5/main.py
2-mediod SC:0.12400088904425093 CP:0.33828186952196126
3-mediod SC:0.11333828683132982 CP:0.30028908463657233
4-mediod SC:0.09170935136283188 CP:0.2792925058206808
5-mediod SC:0.11366997301212271 CP:0.25216706492390273
6-mediod SC:0.07240580993498766 CP:0.24229741888815357
7-mediod SC:0.07494863503777835 CP:0.23895248294753985
8-mediod SC:0.06296597663880893 CP:0.2199955257132647
9-mediod SC:0.06494844613309111 CP:0.21236683611047855
10-mediod SC:0.050964923193091885 CP:0.20765266019628004
11-mediod SC:0.025065771037937567 CP:0.21109506087721183
12-mediod SC:0.038960221889099374 CP:0.20449562704670093
13-mediod SC:0.027098261598533686 CP:0.19472750028709526
14-mediod SC:0.028098452220272713 CP:0.1840576609315278
15-mediod SC:0.002449992761689592 CP:0.17974132383689995
16-mediod SC:-0.005602383387707003 CP:0.1763232995178907
17-mediod SC:-0.0013569240438337856 CP:0.17690661457492135
18-mediod SC:-0.006777216338325125 CP:0.16933665570364603
19-mediod SC:-0.029992752789623833 CP:0.16060139092111508
20-mediod SC:-0.01770985106149206 CP:0.16153548367454765
21-mediod SC:-0.02968203999616766 CP:0.15574233269754317
22-mediod SC:-0.036912184924350555 CP:0.15557833256334225
23-mediod SC:-0.04618956687327954 CP:0.14897095956660766
24-mediod SC:-0.03838987042596649 CP:0.148264616847465
25-mediod SC:-0.042873912828585 6 CP:0.146108414702430 6
26-mediod SC:-0.05576655889055745 CP:0.1420969665161248
27-mediod SC:-0.057672392486353995 CP:0.1399417461891272
28-mediod SC:-0.05719317696679131 CP:0.1402201714837009
29-mediod SC:-0.06146266363872405 CP:0.13320359365938989
30-mediod SC:-0.06748179142641633 CP:0.13431553867132437

Process finished with exit code 0
```

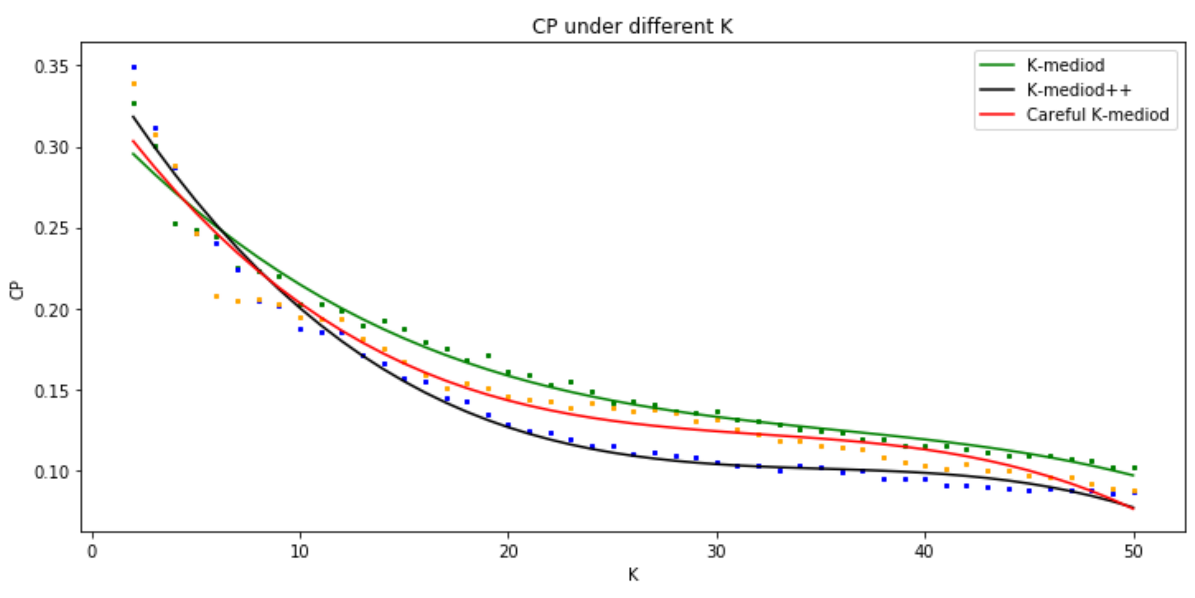
做出随机取$K-mediod$ 初始点和 用一套规则取初始点,分别对应 $Careful$ 和 $Random$

实验结果

In [4]:
```python
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
import pickle

with open("total_sc.pkl","rb") as f:
    ord_sc=pickle.load(f)

with open("total_cp.pkl","rb") as f:
    ord_cp=pickle.load(f)


with open("plus_sc.pkl","rb") as f:
    total_sc=pickle.load(f)

with open("plus_cp.pkl","rb") as f:
    total_cp=pickle.load(f)

with open("careful_sc.pkl","rb") as f:
    care_sc=pickle.load(f)

with open("careful_cp.pkl","rb") as f:
    care_cp=pickle.load(f)

plt.figure(figsize=(12,12))


plt.subplot(211)
x=list(ord_sc.keys())
y=list(ord_sc.values())
plt.scatter(x,y,s=5,c='green',marker=(10,1),alpha=1 ,lw=1,facecolors='none')
psc = np.polyfit(x, y, 3)
f = np.poly1d(psc) # 拼接方程
plt.plot(x, f(x),"green",label="K-mediod")


x=list(total_sc.keys())
y=list(total_sc.values())
plt.subplot(211)
plt.scatter(x,y,s=5,c='b',marker=(10,1),alpha=1 ,lw=1,facecolors='none')
psc = np.polyfit(x, y, 3)
f = np.poly1d(psc) # 拼接方程
plt.plot(x, f(x),"black",label="K-mediod++")

x=list(care_sc.keys())
y=list(care_sc.values())
plt.scatter(x,y,s=5,c='orange',marker=(10,1),alpha=1 ,lw=1,facecolors='none')
psc =np.polyfit(x,y,3)
f = np.poly1d(psc) # 拼接方程
plt.plot(x, f(x),"red",label="Careful K-mediod")
plt.title("SC under different K")
plt.legend()
plt.xlabel("K")#x轴上的名字
plt.ylabel("SC")#y轴上的名字

plt.subplot(212)

x=list(ord_cp.keys())
y=list(ord_cp.values())
plt.scatter(x,y,s=5,c='green',marker=(10,1),alpha=1 ,lw=1,facecolors='none')
pcp = np.polyfit(x, y, 3) # n=1为一次函数，返回函数参数
f = np.poly1d(pcp) # 拼接方程
plt.plot(x, f(x),"green",label="K-mediod")


x=list(total_cp.keys())
y=list(total_cp.values())
plt.scatter(x,y,s=5,c='b',marker=(10,1),alpha=1 ,lw=1,facecolors='none')
pcp = np.polyfit(x, y, 3) # n=1为一次函数，返回函数参数
f = np.poly1d(pcp) # 拼接方程
plt.plot(x, f(x),"black",label="K-mediod++")

x=list(care_cp.keys())
y=list(care_cp.values())
plt.scatter(x,y,s=5,c='orange',marker=(10,1),alpha=1 ,lw=1,facecolors='none')
psc =np.polyfit(x,y,3)
f = np.poly1d(psc) # 拼接方程
plt.plot(x, f(x),"red",label="Careful K-mediod")
plt.title("CP under different K")
plt.legend()
plt.xlabel("K")#x轴上的名字
plt.ylabel("CP")#y轴上的名字
plt.show()
```

```
E:\coding\Anaconda\lib\site-packages\ipykernel_launcher.py:39: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance.  In a future version, a new instance will always be created and returned.  Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
```





可以看出,当使用原始的$K-mediod$算法时，SC和CP指标相比于$q1$和$q2$有了显著提升。而使用优化初始点选取后的$K-mediod++$在两项指标上都超过了前者。而使用我自己实现的 $CarefulK-mediod$其中SC在$k=2$时达到了 0.2905,然而在$k=[10,20]$范围内SC和CP不如$K-mediod++$

$get_k_initial_points$代码

In [5]:
```python
    def get_k_initial_points(self, dm, num_k, alpha):
        assert dm is not None
        vipnos = list(sorted(self.FTC_trees.keys()))
        n_vipno = dm.shape[0]
        ipts = list()

        row_index = [i for i in range(n_vipno)]
        np.random.shuffle(row_index)
        for row in row_index:
            vtree = self.FTC_trees[vipnos[row]]
            if len(vtree.root.children) > 12:
                start_pt = row
                ipts.append(start_pt)
                break

        ratio = (1 - alpha) * num_k / n_vipno
        for k in range(1, num_k):
            sub_dm = dm[ipts]
            dist_to_centroid = np.sum(sub_dm, 0)
            dist_to_centroid[ipts] = -1
            # print(dist_to_centroid)
            new_cols = np.where(dist_to_centroid > max(dist_to_centroid) * (alpha + ratio))

            good_col, lst_child_num = new_cols[0][0], 0
            biggest = -1
            # print("new_cols:", new_cols)
            for col in new_cols[0]:
                vtree = self.FTC_trees[vipnos[col]]
                if np.sum(dm[col]) >= (n_vipno - 1) * alpha * (1-(num_k-2)/(n_vipno-2)) and len(vtree.root.children) >
= lst_child_num:
                    good_col = col
                    biggest = np.sum(dm[col])
            # print("good_col:", np.sum(dm[good_col]))
            ipts.append(good_col)

        #print(ipts, dm[ipts[0], ipts[1]])
        #print(len(self.FTC_trees[vipnos[ipts[0]]].root.children))
        #print(len(self.FTC_trees[vipnos[ipts[1]]].root.children))
        return ipts
```

选取初始点逻辑:

1. 确定簇数k，第一个点选取一个不离群的点，可直接选取一个一层子节点较多的点。
2. 确定阈值 $alpha$
3. 选取与其距离超过 $alpha+(1-alpha)*\frac{t}{nvipno}$ 的所有点
4. 在这些点中选取一层子节点最多且到之和点距离大于$nvipno \times alpha(1-\frac{t}{nvipno})$的点作为新质心

其中 $nvipno$ 是数据总数，t是第t次选取中心点

思考

K-mediod 算法选取初始点对离群点非常敏感，如果按照$K-mediod++$取最远点的做法，会取到那些只买了一种商品的用户，这些用户与其他所有用户的距离都接近1，因此我放宽了对距离的约束，争取寻找到离质心比较远但是不是离所有点都远的用户做质心。而第一层子节点多的节点大概率不离群，如果离已有质心远就适合做新质心，也因此SC能达到0.29。但随着K增大，单个离群点的影响变小，K-mediod表现更好，因此后续如何做到权衡，把深层子节点的数目和重复性纳入考虑，是可以研究的方向。