# 数据挖掘 hw1 q3

## 代码概述

q3 代码分为 $main.py$ 和 $toolkit.py$ 两个包， 其中 $main.py$ 是主程序所在文件, $toolkit.py$ 包含了一些辅助函数和其他用到的类 主要有3个类：

- Solution

  > 包含算法主要流程所有的操作，存储了所有的数据点

- FTC_Tree

  > 由每一个 $vipno$ 的交易记录组成

- Node

  > 代表FTC_Tree中的每一个节点

## 数据预处理

读取数据，将 $vipno$ 和 $pluno$ 转成字符串，并将购买日期 $sldatime$ 转换成级别 $level$，在建树的时候直接退化

```
dataset = pd.read_csv("trade_new.csv").fillna(0)
# 处理类别
dataset["vipno"] = dataset["vipno"].astype("str")
# dataset.loc[np.where(dataset["amt"] < 0)]["amt"] = 0
dataset["amt"] = np.abs(dataset["amt"])
dataset["pluno"] = dataset["pluno"].astype("str")
dataset["pluno5"] = [t[:5] for t in dataset['pluno']]
# 处理时间
dataset['sldatime'] = [datetime.datetime.strptime(t, '%Y-%m-%d %H:%M:%S') for t in dataset['sldatime']]
# 将时间转化为级别
dataset['level'] = datetime.datetime.now() - dataset['sldatime']
dataset['level'] = [int(t.days / 30) for t in dataset['level']]
dataset['level'] = dataset['level'] - np.min(dataset['level']) + 1
dataset['level'] = 4 - np.log2(dataset['level']).astype("int")
original_dataset = deepcopy(dataset)
```

## 算法流程

1. 创建 $datasets$ 和 $finals$ 两个列表，分别维护待进一步分裂和已经分为一个簇的数据集，同时储存他们的当前质心
2. 从 $datasets$ 中取出第一个数据集，进行论文中的聚类方法，将其分为两个簇。
3. 计算聚类前和聚类后的 $bic$ 变化， 3.1. 如果变大，正式分裂，分裂后的两个数据集放回 $datasets$ 内 3.2. 如果变小，原先的一个数据集标记为一个簇放入 $finals$
4. 重复上述步骤直到 $finals$ 为空

```
# 开始分裂
    while not len(datasets) == 0:
        print("corruption !")
        dataset, cur_core = datasets.pop().values()
        solution.preprocess(dataset)
        # end_early 用于避免只有1个数据点，导致k=2无法找到两个初始质心
        end_early, cores, results = solution.transaction_kmeans(2)
        if end_early:
            finals.append({"ds": dataset, "core": cur_core})
            continue
        else:
            dataset1, dataset2 = split_dataset(dataset, results)

        new_bic = solution.bic(results, cores, dataset1, dataset2, 2)
        current_bic = solution.bic(None, None, dataset, 1)
        print(new_bic, current_bic)
        if new_bic > current_bic:
            datasets.append({"ds": dataset1, "core": cores[0]})
            datasets.append({"ds": dataset2, "core": cores[1]})
        else:
            finals.append({"ds": dataset, "core": cur_core})
# 将拆分以后的数据集提取vipno并合并成最终结果
    final_results = dict()
    final_cores = list()
    category = 0
    for final in finals:
        for vipno in np.unique(final["ds"]["vipno"]):
            final_results[vipno] = category
        category = category + 1
        final_cores.append(final["core"])
```

## 实现细节

代码按照 $Solution \rightarrow FTC_tree \rightarrow Node$自顶向下层层封装和调用，主要操作全部采用递归实现。这里展示几个主要的操作实现，其他函数具体实现可以看代码。

#### k-means实现细节

```
class solution:
    def transaction_kmeans(self, k):
        vipnos = list(self.FTC_trees.keys())
        # 如果数据数少于聚数，提前结束
        if len(vipnos) < num_k:
            return True, None, None
        results = dict()
        flag = False
        # 选取初始质心
        idx0, idx1 = choose_random_k(vipnos, k)
        cores = [copy_tree(self.FTC_trees[idx0]), copy_tree(self.FTC_trees[idx1])]
        # 迭代
        max_iter = 30
        for itr in range(1, max_iter + 1):
            vipno_min = dict()
            for vipno, vipno_tree in self.FTC_trees.items():
                dist0 = vipno_tree.distance(cores[0])
                dist1 = vipno_tree.distance(cores[1])
                vipno_min[vipno] = 0 if dist0 < dist1 else 1

            if vipno_min == results:
                return flag, cores, results
            results = vipno_min
            #每次迭代优次更新质心
            for i in range(k):
                cvs = [vipno for vipno, result in results.items() if result == i]
                if not len(cvs) == 0:
                    cpts = {cv: ct for cv, ct in self.FTC_trees.items() if cv in cvs}
                    cores[i] = self.get_centroid_tree(cpts)
                else:
                    print("early end")
                    flag = True
                    return flag, results, cores
        return flag, cores, results
```

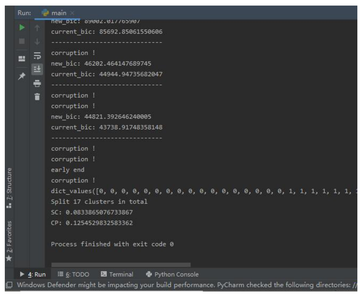
#### 质心树选取\ 质心树选取主要依赖于上一次聚类的result产生的簇内点和 $FTC\_tree$ 中的 $union$ 和 $intersection$ 函数

```
class solution:
    def get_centroid_tree(self, cpts):
        union_tree, centroid_tree = None, None
        for v in cpts:
            union_tree = copy_tree(cpts[v]) if union_tree is None else union_tree.union(cpts[v])
        cur_amt, mindist = 1, float("inf")
        amt_step = union_tree.get_avg_amt()
        amt_end = union_tree.get_max_amt()
        centroid_tree = copy_tree(union_tree)
        while cur_amt < amt_end:
            union_tree.tune(cur_amt)
            dist = sum([cpt.distance(union_tree) for cpt in cpts.values()])
            if dist < mindist:
                mindist = dist
                centroid_tree = copy_tree(union_tree)

            cur_amt = cur_amt + amt_step
        return centroid_tree
```

## 聚类结果

最终在随机取初始点的情况下，$SC$分布在$[0, 0.1]$之间，$CP$分布在$[0.1, 0.15]$\\ 每条分裂线代表一次成功分裂，当簇内用户数小于2时，将停止对簇继续分裂，因而输出会有多个$corruption!$\\ $dict_values$是对每一个用户具体分到的簇的标号，由于论文采用$bic$作为分裂准则，所以无法准确估计k的个数并做出$SC$，$CP$随 $k$ 的变化趋势，但是距离的分布情况是可以分析的。

#### 程序运行截图

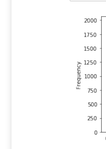



#### 距离分布情况

为了和q1,q2比较距离分布情况，我用$pickle$ 将距离矩阵转存到文件再读取作图

```
In [1]:
import matplotlib.pyplot as plt
import matplotlib as mpl

def draw_trend(x,y,n,lbx,lby,title):
    fig =plt.figure(2)
    ax = fig.add_subplot(1,1,1)
    ax.scatter(x,y,s=5,c='b',marker=(10,1),alpha=1 ,lw=1,facecolors='none')
    ax.set_title(title)
    plt.xlabel(lbx)#x轴上的名字
    plt.ylabel(lby)#y轴上的名字

    parameter = np.polyfit(x, y, n) # n=1为一次函数，返回函数参数
    f = np.polyld(parameter) # 拼接方程
    plt.plot(x, f(x),'black')

    plt.show()

def draw_dist_freq(dm):
    flatten_hist = list()
    for i in range(dm.shape[0]):
        for j in range(i+1):
            flatten_hist.append(dm[i,j])

    hist,bins = np.histogram(flatten_hist,bins = 200)

    plt.bar(list(bins)[1:],hist,width=0.01)
    plt.title("Scale of Distance")
    plt.xlabel("distance")#y轴上的名字
    plt.ylabel("Frequency")#y轴上的名字
    plt.show()
```

```
In [2]:
import pickle
import numpy as np
with open('dm.pkl','rb') as f:
    dm = pickle.load(f)
draw_dist_freq(dm)
```




#### 结论

可以看到，相比于q1,q2.距离的分布变得分散，而不是集中在$[0.8, 1.0]$,证明论文里的距离计算有一定的合理性。因此聚类效果更好。