k-means 算法实现

1. 概念

K-means 算法是硬聚类算法，是基于原型的目标函数聚类方法的代表，它是数据点到原型的某种距离作为优化的目标函数，利用函数求极值的方法得到迭代运算的调整规则。K-means 算法以欧式距离作为相似度测度，它是求对应某一初始聚类中心向量最优分类，使得评价指标最小。用误差平方和准则函数作为聚类准则函数。

2. 算法原理

随机选取 k 个初始均值向量，然后对数据集进行遍历，每次找到一个距离某个均值向量最近的点，将该数据集加入该均值向量所在的 cluster，重复迭代多次，直到 k 个均值向量均不会更新为止（均值向量都相等），算法步骤如下所示：

输入: 样本集 $D = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$;
聚类簇数 k.
过程:
从 D 中随机选择 k 个样本作为初始均值向量 $\{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_k\}$
repeat
  令 $C_i = \varnothing \ (1 \leqslant i \leqslant k)$
  for $j = 1, 2, \ldots, m$ **do**
    计算样本 $\boldsymbol{x}_j$ 与各均值向量 $\boldsymbol{\mu}_i \ (1 \leqslant i \leqslant k)$ 的距离: $d_{ji} = \|\boldsymbol{x}_j - \boldsymbol{\mu}_i\|_2$;
    根据距离最近的均值向量确定 $\boldsymbol{x}_j$ 的簇标记: $\lambda_j = \arg\min_{i \in \{1,2,\ldots,k\}} d_{ji}$;
    将样本 $\boldsymbol{x}_j$ 划入相应的簇: $C_{\lambda_j} = C_{\lambda_j} \bigcup \{\boldsymbol{x}_j\}$;
  end for
  for $i = 1, 2, \ldots, k$ **do**
    计算新均值向量: $\boldsymbol{\mu}'_i = \frac{1}{|C_i|} \sum_{\boldsymbol{x} \in C_i} \boldsymbol{x}$;
    if $\boldsymbol{\mu}'_i \neq \boldsymbol{\mu}_i$ **then**
      将当前均值向量 $\boldsymbol{\mu}_i$ 更新为 $\boldsymbol{\mu}'_i$
    else
      保持当前均值向量不变
    end if
  end for
until 当前均值向量均未更新
输出: 簇划分 $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$

算法的评价函数(误差平方和)公式如下所示

给定样本集 $D=\{\boldsymbol{x}_1,\boldsymbol{x}_2,\ldots,\boldsymbol{x}_m\}$，“$k$ 均值”(k-means)算法针对聚类所得簇划分 $\mathcal{C}=\{C_1,C_2,\ldots,C_k\}$ 最小化平方误差

$$E=\sum_{i=1}^{k}\sum_{\boldsymbol{x}\in C_i}\|\boldsymbol{x}-\boldsymbol{\mu}_i\|_2^2, \tag{9.24}$$

其中 $\boldsymbol{\mu}_i=\frac{1}{|C_i|}\sum_{\boldsymbol{x}\in C_i}\boldsymbol{x}$ 是簇 C_i 的均值向量. 直观来看, 式(9.24) 在一定程度上刻画了簇内样本围绕簇均值向量的紧密程度, E 值越小则簇内样本相似度越高.

3. 算法源码

1.首先就是对本地文件系统数据集的加载

```
#加载本地数据集
def loadDataSet():
    sets = pd.read_csv("D://melon.csv",sep="\t",names=["x1","x2"])
    dataSet = []
    for i in range(len(sets)):
        initDataSet = [round(float(x),3) for x in sets.loc[i].tolist()]
        dataSet.append(initDataSet)
    return dataSet
```

2.从数据集里边随机选取 k 个均值向量作为初始均指向量

```
#随机抽取k个向量作为均值向量进行聚类
def getRandomMeanValues(k,dataSet):
    setLens = len(dataSet)
    meanVec = []
    # 从range()里边随机抽取k个数
    ranList = rd.sample(range(setLens), k)
    for i in ranList:
        meanVec.append(dataSet[i])
    return meanVec
```

3.遍历数据集，迭代划分类

```
#k 均值算法,迭代更新均值向量，直至均值向量都不更新为止
def calDistanaceAndIteration(dataSet,k):
    #先找出均值向量
    meansVec = getRandomMeanValues(k,dataSet)

    cluster = []
    for e in range(k):
        cluster.append([])
```

```
    nums = 0
    count = 0
    sortArray = []
    while count == 0:
        nums += 1

        # cluster = [[], [], []]  #每次都需要清空

        cluster = []
        for e in range(k):
            cluster.append([])

        for data in dataSet:
            min = 0

            #当前均值向量的位置

            p = 0
            for pos in range(len(meansVec)):
                vecData = np.array(data)
                vecMean = np.array(meansVec[pos])
                dist = np.sqrt(np.sum(np.square(vecData -
vecMean)))
                sortArray.append(dist)

            min = sortArray[0]
            for m in range(len(sortArray)):
                if sortArray[m] < min:
                    min = sortArray[m]
                    p = m
            cluster[p].append(data)
            sortArray = []

        # 计算新的均值向量,如果每次三个均值都更新的话，count 每次

更新都加 1，否则就是没更新，继续循环，划分新的均指向量

        pos = 0
        # print(meansVec)
        tempList = []
        for data in cluster:
            sumx1 = 0
            sumx2 = 0
            lens = len(data)

            # 计算初始的一个 cluster 的新的均值

            for d in data:
```

```
                sumx1 += d[0]
                sumx2 += d[1]
            meanX1, meanX2 = round(float(sumx1 / lens),
3), round(float(sumx2 / lens), 3)
            tempList.append(meanX1)
            tempList.append(meanX2)
            # print(meanX1, meanX2)
            # print(tempList)

            #不等于，表示还得继续循环

            if meansVec[pos] != tempList:
                meansVec[pos] = tempList

                count += 1  #要是变，说明有一个更新了，还需要继
续循环
            else:
                pass
            pos += 1
            tempList = []
        if count != 0:

            #还需要继续循环

            count = 0

        else:  #不需要循环,给定一个较大的数值,让他直接退出

            count = 100
        # print(meansVec)
        # print(count)
    return cluster,meansVec,nums,k
```

4. k-means 算法评价函数(cluster 之间的最小化误差平方和)，绘制 E-K 图,找到最优 k 值(E-K 第一个拐点处的位置) 如下所示

```
#cluster之间的最小化误差平方和
def calEvaluateMethod(cluster,meanVec,k):
    pos = 0
    sums = 0
    for data in cluster:
        for ele in data:
            vecData = np.array(ele)
            vecMean = np.array(meanVec[pos])
            dist = np.sum(np.square(vecData - vecMean))
            # print(dist)
            # print(np.sqrt(dist),dist)
        sums += dist
        pos += 1
    sums = round(float(sums),3)
    # print(sums)
    return sums,k
```

```
for i in range(2,10):
    cluster, meanVec, nums, k = calDistanaceAndIteration(dataSet,
i)
    sums,k = calEvaluateMethod(cluster,meanVec,i)
    eArray.append(sums)
    kArray.append(i)
drawEkCurve(eArray,kArray)
```

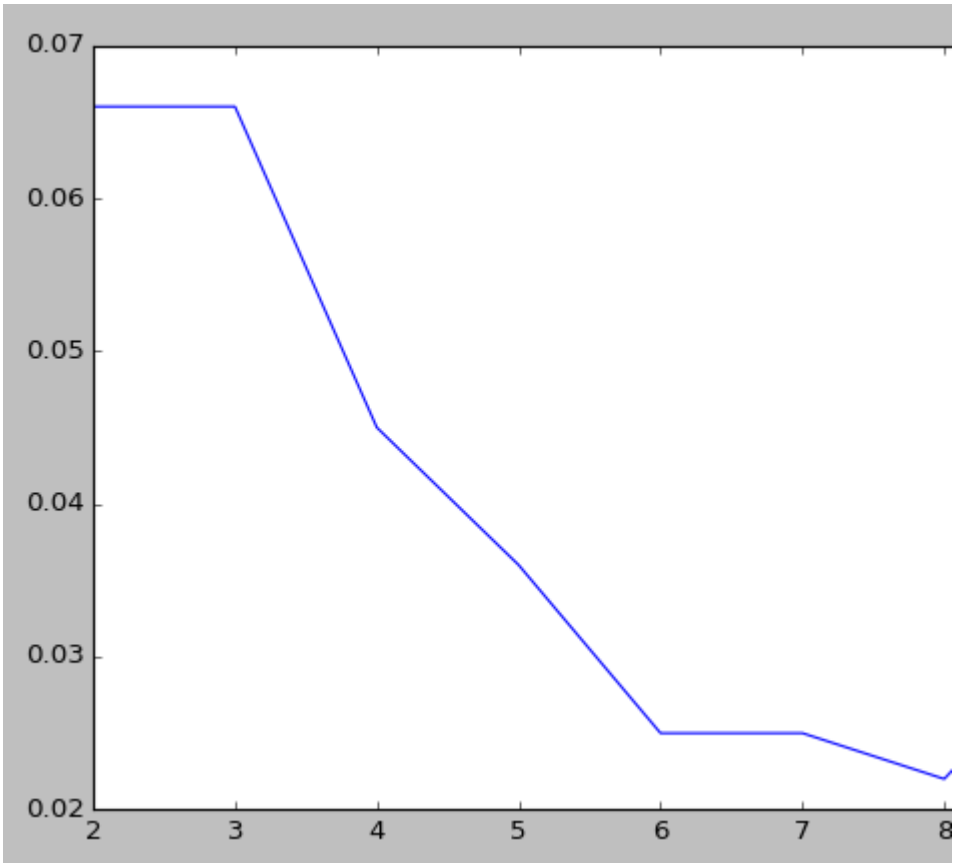
#根据多次执行算法，绘制出 E-K 得知，k = 4 的时候聚类效果是最好的

5.随机选取多个 k 值，迭代多次，计算误差平方和 E，划出 E-K 图

```
  #聚类的个数

# 从 range()里边随机抽取 1 个数
k = rd.sample(range(2,10), 1)
k = k[0]
# print(k)
bestK = 0

#E,k 保存的位置
eArray = []
kArray = []
dataSet = loadDataSet()


# i 表示随机产生的 k 值
for i in range(2,10):
    cluster, meanVec, nums, k =
```

```
calDistanaceAndIteration(dataSet, i)
    sums,k = calEvaluateMethod(cluster,meanVec,i)
    eArray.append(sums)
    kArray.append(i)
drawEkCurve(eArray,kArray)

#根据多次执行 E-K 得知，k = 4 的时候聚类效果是最好的

#所以令 k = 4 的时候,迭代聚类
bestK = 4
cluster, meanVec, nums, k =
calDistanaceAndIteration(dataSet, bestK)
drawImage(cluster,bestK)
```

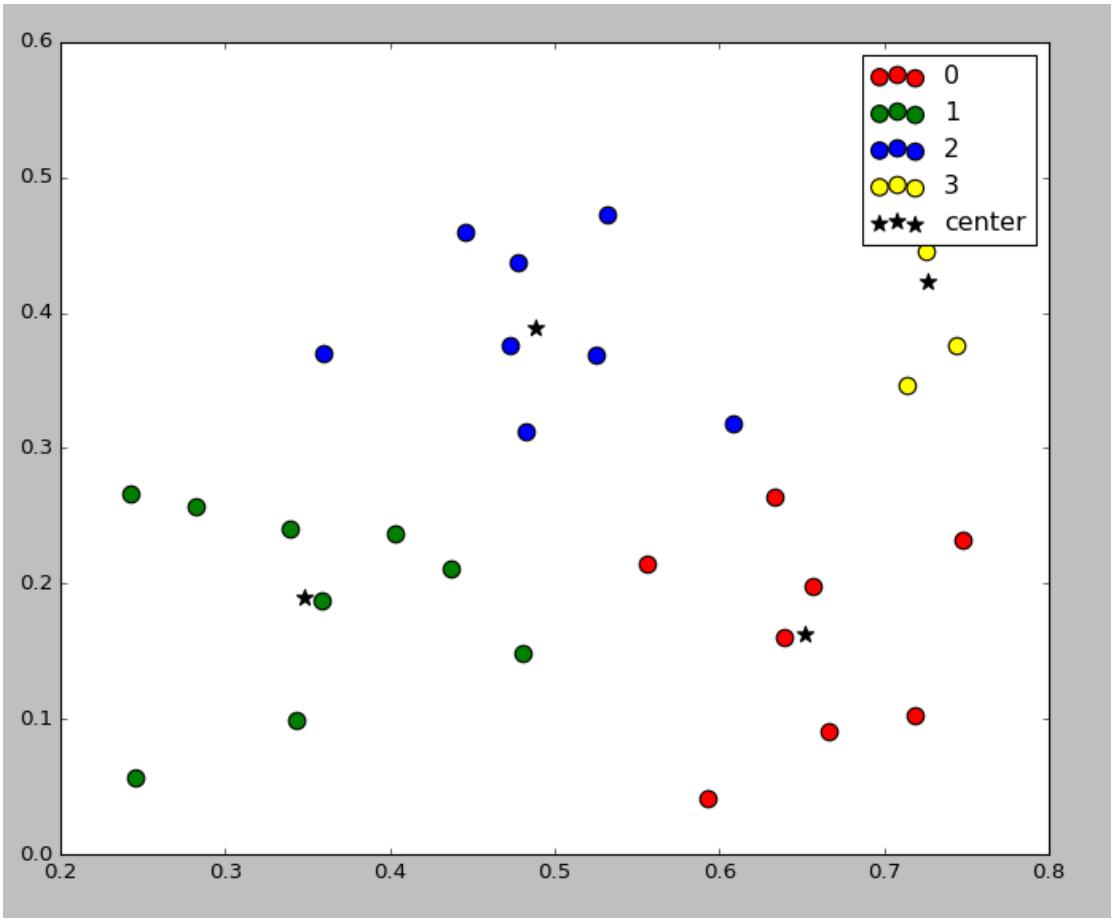
迭代多次求得的 E-K 图如下图所示



可以看出当 k = 4 的时候聚类效果是最好

4．算法执行结果

当 k = 4 的时候聚类效果如下图所示：



5. 总结

聚类是无监督学习的一种，而 K-means 是聚类最常见的一种算法。 K-means 只能处理连续型属性的样本，若有标称型或离散型属性要先将其转化为连续型数值（用 one-hot 编码）。基本简单的实现了 k-means 算法，由于 k-means 算法需要重复迭代多次，所以算法的效率不是太高，在本次算法中，未对噪声点进行处理，还有待处理，算法效率还需要进一步优化。

EM 算法实现

1．概念

当有部分数据缺失或者无法观察到时，EM 算法提供了一个高效的迭代程序用来计算这些数据的最大似然估计。在每一步迭代分为两个步骤：期望（Expectation）步骤和最大化（Maximization）步骤，因此称为 EM 算法，EM 算法解决这个的思路是使用启发式的迭代方法，既然我们无法直接求出模型分布参数，那么我们可以先猜想隐含数据（EM 算法的 E 步），接着基于观察数据和猜测的隐含数据一起来极大化对数似然，求解我们的模型参数（EM 算法的 M 步)。由于我们之前的隐藏数据是猜测的，所以此时

得到的模型参数一般还不是我们想要的结果。不过没关系，我们基于当前得到的模型参数，继续猜测隐含数据（EM 算法的 E 步），然后继续极大化对数似然，求解我们的模型参数（EM 算法的 M 步)。以此类推，不断的迭代下去，直到模型分布参数基本无变化，算法收敛，找到合适的模型参数。

2．算法原理

假设全部数据 Z 是由可观测到的样本 X={X1, X2，……, Xn}和不可观测到的样本 Z={Z1, Z2，……, Zn}组成的，则 Y = X∪Z。EM 算法通过搜寻使全部数据的似然函数 Log(L(Z; h))的期望值最大来寻找极大似然估计，注意此处的 h 不是一个变量，而是多个变量组成的参数集合。此期望值是在 Z 所遵循的概率分布上计算，此分布由未知参数 h 确定。然而 Z 所遵循的分布是未知的。EM 算法使用其当前的假设 h`代替实际参数 h，以估计 Z 的分布。

$$Q(h`|h) = E[\ln P(Y|h`) | h, X]$$

EM 算法重复以下两个步骤直至收敛。

步骤 1：估计（E）步骤：使用当前假设 h 和观察到的数据 X 来估计 Y 上的概率分布以计算 Q(h` | h)。

$$Q(h` | h) \leftarrow E[\ln P(Y|h`) | h, X]$$

步骤 2：最大化（M）步骤：将假设 h 替换为使 Q 函数最大化的假设 h:

$$h \leftarrow \text{argmax} Q(h` | h)$$

3．算法源码

1. 数据集的加载

```
def readDataset():
    sets = pd.read_csv("D://fiveSet.csv", sep=" ", names=["x1", "x2"])
    dataSet = []
    for i in range(len(sets)):
        initDataSet = [round(float(x), 3) for x in sets.loc[i].tolist()]
        dataSet.append(initDataSet)
    return dataSet
```

2.迭代计算数据集合和初始簇中心的的隶属度，就是计算概率

```
#迭代计算数据集合和初始簇中心的的隶属度，就是计算概率
def calMembershipDegree(dataSet):

    xAndYArray = []
    xArray,yArray = getXandY(dataSet)
```

```
    xAndYArray.append(xArray)
    xAndYArray.append(yArray)

    dataLen = len(dataSet)

    #放置最终结果概率的数组，

    cluster = [[],[]]

    meanVec = [[],[]]

    #迭代次数

    num = 0

    #初始类中心

    clusterCenter = [[3.0,3.0],[4.0,10.0]]

    #放置距离的数组

    distance = []
    outDistance1 = []
    outDistance2 = []
    outDistance = [[],[]]

    posOut = 0

    #循环条件

    count = 0
    while count == 0:
        cluster = [[], []]

        #迭代产生的均值向量

        meanVec = [[], []]
        num += 1
        for data in dataSet:
            posIn = 0

            for ele in clusterCenter:
                # distance = []
                if data in clusterCenter:
                    if data == ele:
                        cluster[posIn].append(1)
                    else:
```

```
                    cluster[posIn].append(0)
                posIn += 1
            else:
                vecData = np.array(data)
                centors = np.array(ele)
                dist = np.sum(np.square(vecData -
centors))
                distance.append(dist)
                # cluster[posIn].append(dist)
                posIn += 1
        flag = data in clusterCenter

        #E步  计算向量之间的距离，以便于后边方面进行计算

        if flag is False:
            sums = distance[0]+distance[1]

outDistance1.append(round(float(distance[1]/sums),2))
            outDistance2.append(round(float(distance[0]
/ sums),2))
            distance = []
            posOut += 1
    # print(outDistance1,outDistance2)
    outDistance[0].append(outDistance1)

    outDistance[1].append(outDistance2)
    outDistance1 = []
    outDistance2 = []
    for i in range(2):
        for x in outDistance[i]:
            cluster[i].extend(x)
    # print(cluster)
    outDistance = [[],[]]

    #M步

    for ins in range(2):

        #计算概率

        for out in range(2):
            aa =
[(cluster[ins][n]**2)*xAndYArray[out][n] for n in
range(dataLen)]
            bb = [cluster[ins][m]**2 for m in
range(dataLen)]
            # print(aa)
```

```
                # print(bb)
                aa = ft.reduce(add,aa)
                bb = ft.reduce(add,bb)
                result = retainDecimal(aa / bb)
                meanVec[ins].append(result)

        for i in range(2):
            vecData = np.array(meanVec[i])
            cent = np.array(clusterCenter[i])
            #计算新的均值向量和旧的初始向量的差别，判断是否需要迭代
            oDist = np.sum(np.square(vecData - cent))
            if oDist < 0.0001:  #说明均值接近，可不用在迭代
                count += 1
            else:
                pass
        if count >= 2:  #两个新的均值向量和两个旧的均值向量都接近，无需迭代，直接退出迭代
            count = 100
        else:  #还需要进行迭代，迭代完成之后更新均指向量
            count = 0
            clusterCenter[0] = meanVec[0]
            clusterCenter[1] = meanVec[1]

    return meanVec,cluster,num
```

3.主函数如下，k 的个数为 2

```
def main():
    dataSet = readDataset()
    pos = 0
    meanVec,cluster,num = calMembershipDegree(dataSet)
    print("迭代次数为:",num)
    print("最后产生的簇中心向量为:",meanVec)
    #里边是每个样本属于该类别的概率
    print("最终形成的分类数组为:",cluster)
    for x in cluster[0]:
        if x > 0.5:
            print('样本%s' % dataSet[pos],'的类别为1')
        else:
            print('样本%s' % dataSet[pos],'类别为2')
        pos += 1
```

4．算法执行结果

```
迭代次数为：7
最后产生的簇中心向量为：[[5.24, 6.34], [17.84, 8.73]]
最终形成的分类数组概率为：[[0.94, 0.93, 0.86, 0.16, 0.03, 0.05], [0.06, 0.07, 0.14, 0.84, 0.97, 0.95]]
样本[3.0, 3.0] 的类别为1
样本[4.0, 10.0] 的类别为1
样本[9.0, 6.0] 的类别为1
样本[14.0, 8.0] 类别为2
样本[18.0, 11.0] 类别为2
样本[21.0, 7.0] 类别为2
```

5．总结

EM 算法里已知的是观察数据，未知的是隐含数据和模型参数，在 E 步，就是把模型参数的值固定，优化隐含数据的分布，而在 M 步，我们所做的事情是固定隐含数据分布，优化模型参数的值。