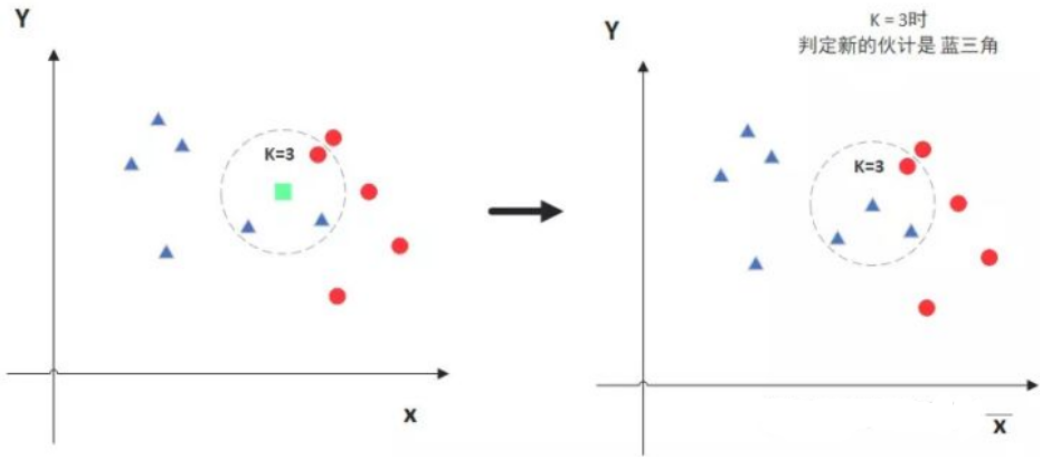
中期考核文档

1. KNN算法的总结

- **KNN的简介：**
  - K近邻算法（K-Nearest Neighbor，简称KNN）是一种非常简单的分类和监督学习算法，他同时也是一种基于实例的学习。**其实啊，KNN的原理就是当预测一个新的值x的时候，根据它距离最近的K个点是什么类别来判断x属于哪个类别**，其基本流程如下：
    1. 计算测试数据与各个训练数据之间的距离；（一般为**欧式距离**或**曼哈顿距离**）
    2. 按照距离的递增关系进行排序；（这个过程会加权重，越近的权重越大）
    3. 选取距离最小的K个点；（K 为用户给定的值）
    4. 确定前K个点所在类别的出现频率；
    5. 返回前K个点中出现频率最高的类别作为测试数据的预测分类。



- **KNN算法的特点：**

  KNN是一种**非参的**，**惰性**的算法模型。什么是非参，什么是惰性呢?

  1. **非参**的意思并不是说这个算法不需要参数，而是意味着这个模型不会对数据做出任何的假设，与之相　　对的是线性回归（我们总会假设线性回归是一条直线）。也就是说KNN建立的模型结构是根据数据来　　决定的，这也比较符合现实的情况，毕竟在现实中的情况往往与理论上的假设是不相符的。

  2. **惰性**又是因为KNN算法不需要先对数据进行大量训练（tranning），它没有明确的训练数据的过程，　　或者说这个过程很快。

- **KNN算法的优势和劣势**
  - **KNN算法的优点**
    1. 简单易用，相比其他算法，KNN算是比较简洁明了的算法。即使没有很高的数学基础也能搞清楚它的原理。
    2. 模型训练时间快，上面说到KNN算法是惰性的，这里也就不再过多讲述。
    3. 预测效果较好。
    4. 对异常值（噪声）不敏感
  - **KNN算法的缺点**

1. 对内存要求较高，因为该算法存储了所有训练数据
2. 预测阶段可能很慢
3. 对不相关的功能和数据规模敏感

- **额外拓展：（分类与聚类 Classification & Clustering）**
  - **分类（Classification）：**

    分类简单来说，就是根据文本的特征或属性，划分到已有的类别中。也就是说，这些类别是已知的，通过对已知分类的数据进行训练和学习，找到这些不同类的特征，再对未分类的数据进行分类。属于典型的监督学习。

    常见的分类算法有：

    - **K近邻（KNN）**
    - **决策树 （Decision Tree）**
    - **朴素贝叶斯 （Naive Bayes）**
    - **逻辑回归 （Logistic Regression）**
    - **支持向量机（SVM）**
    - **随机森林 （Random Forest）**
  - **聚类（Clustering）：**

    聚类的理解更简单，就是你压根不知道数据会分为几类，通过聚类分析将数据或者说用户聚合成几个群体，那就是聚类了。聚类不需要对数据进行训练和学习，属于典型的无监督学习。

    常见的聚类算法有：

    - **K均值（K-means）**
    - **DBSCAN**
    - **……**
- **KNN实现的感悟：**

```python
import csv   # 导入csv库
import random

# 读取鸢尾花数据集
with open("E:\\All_Test_Files\\Py_File\\data\\Iris.csv", 'r') as Iris:
    reader = csv.DictReader(Iris)
    data = [row for row in reader]

# 划分测试集和训练集
random.shuffle(data)  # 打乱顺序
n = len(data) // 3
test_set = data[0:n]  # 划分1/3为测试集
train_set = data[n:]  # 划分1/3为训练集

# knn的实现
# 欧式距离
def distance(d1, d2):
    res = 0  # 初始化欧氏距离为零

    # 对 花萼的长度&宽度 和 花瓣的长度&宽度 取点算欧式距离
    for key in ("SepalLengthCm", "SepalWidthCm", "PetalLengthCm",
"PetalWidthCm"):
        res += (float(d1[key]) - float(d2[key]))**2
```

```
23
24      return res**(0.5)  # 开平方
25
26  k = 3  # 取k值为3（当然也可取其他正整数值）
27  def KNN(data):
28      # 距离
29      res = [
30              {"result" : train['Species'], "distance" : distance(data,
    train)}
31              for train in train_set
32            ]
33      # print(res)  # check
34
35      # 对所得到目标点的距离升序排列
36      res = sorted(res, key = lambda item : item['distance'])
37
38      # 取前 k 个为有效
39      resO2 = res[0:k]
40      # print(resO2)
41
42      # 先初始化各种类的个数
43      result = {'Iris-setosa':0, 'Iris-versicolor':0, 'Iris-virginica':0}
44
45      # 算总距离
46      sum = 0
47      for dis in resO2:
48          sum += dis['distance']
49
50      for disO2 in resO2:
51          result[disO2['result']] += 1 - dis['distance']/sum # 加权
52
53      # print(result)
54      # print("This species: ", data['Species'])
55
56      # 将加权后距离经行比较，然后判断归属于哪一类
57      if (result['Iris-setosa'] > result['Iris-versicolor']) &
    (result['Iris-setosa'] > result['Iris-virginica']):
58          return 'Iris-setosa'
59      elif (result['Iris-virginica'] > result['Iris-versicolor']) &
    (result['Iris-virginica'] > result['Iris-setosa']):
60          return 'Iris-virginica'
61      elif (result['Iris-versicolor'] > result['Iris-virginica']) &
    (result['Iris-versicolor'] > result['Iris-setosa']):
62          return 'Iris-versicolor'
63      else: return -1
64
65  # 测试模型的预测度
66  correct = 0
67  for test in test_set:
68      result = test['Species'] # 真实结果
69      resultO2 = KNN(test)
70
71      if result == resultO2:
72          correct += 1
73
74  print(correct)
75  print(len(test_set))
76  print("成功率: ", (correct / len(test_set)) * 100, "%")
```

```
77
78  '''
79  Ps:
80  这套模型代码的准确度有一定的浮动，但是预测度总体较好，
81  当 K = 3 时，三次运行的结果成功率分别为 96%、94%、94%，
82  当 K = 4 时，三次运行的结果成功率分别为 98%、90%、94%
83  当 K = 1 时，三次运行的结果成功率分别为 0%、0%、0%
84  当 K = 2 时，三次运行的结果成功率分别为 86%、88%、92%
85  可见 K 的选值是非常重要的
86  '''
87
```

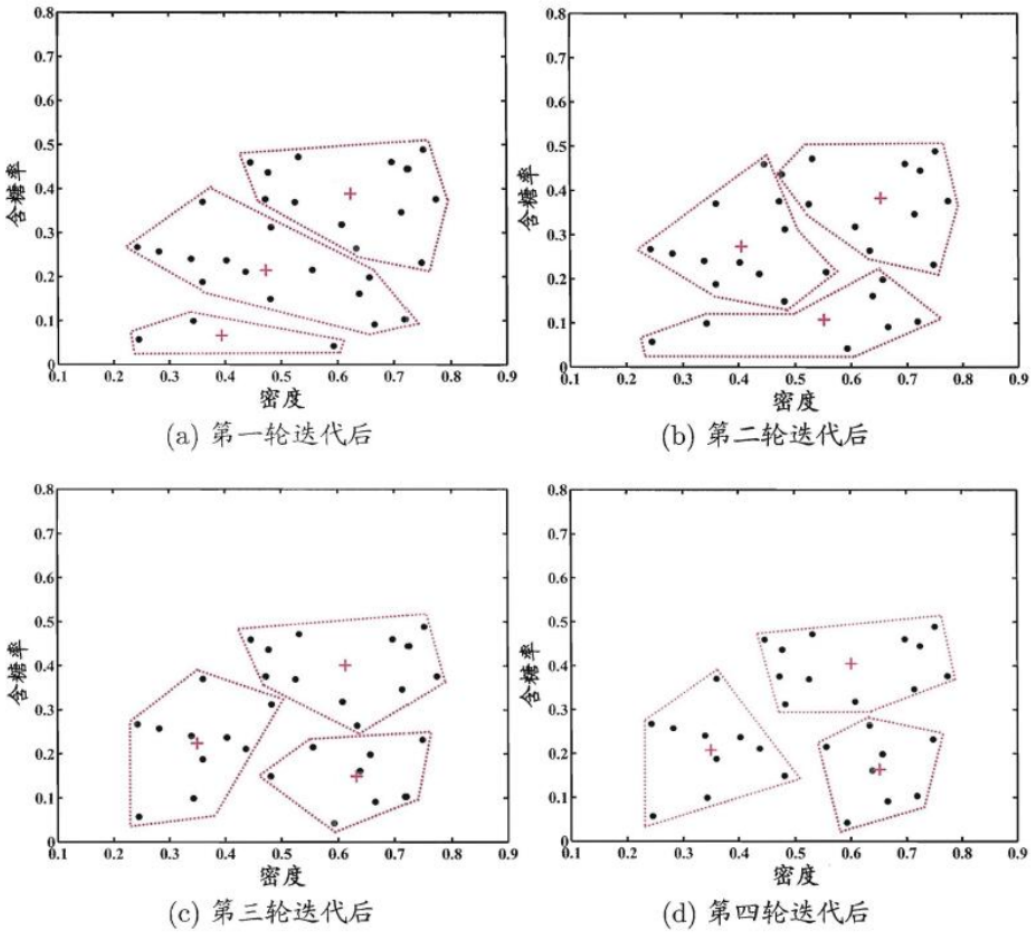
总的来说，KNN算法我认为是最简单的一种机器学习算法，因为它的整个思路就是取K值、算距离、加权距离后再比较，最后分类，在整个实现过程最舒服的感觉就是思路清晰简单，但实话实说，由于我使用的是鸢尾花数据集，所以大概是因为数据量不够大，导致我上文所说的某些缺点并不是能很好地体会，我相信以后再接触这个算法时体会会逐渐加深，但是有一点是值得肯定的——整个算法中 K 值的选取，对整个算法的成功率至关重要！此外由上文的结果，KNN算法不太受噪音的干扰也能初窥端倪。

2. K-means算法的总结

- **K-means算法的简介**
  - K-means算法又名k均值算法。其**算法思想**大致为：先从样本集中随机选取 k 个样本作为**簇中心**，并计算所有样本与这 k 个“簇中心”的距离，对于每一个样本，将其划分到与其**距离最近**的“簇中心”所在的簇中，对于新的簇计算各个簇的新的“簇中心”。

    根据以上描述，我们大致可以猜测到实现K-means算法的主要几点：

    1. 随机选择 K 个点作为点的聚类中心，这表示我们要将数据分为 K 类。
    2. 遍历所有的点 P, 算出 P 到每个聚类中心的距离，将 P 放到最近的聚类中心的点集中。遍历结束后我们将得到 K 个点集。
    3. 遍历每一个点集，算出每一个点集的中心位置，将其作为新的聚类中心。
    4. 重复步骤 2 和步骤 3进行迭代，直到聚类中心位置不再移动。
  - 例子如下（初始 k = 3）：



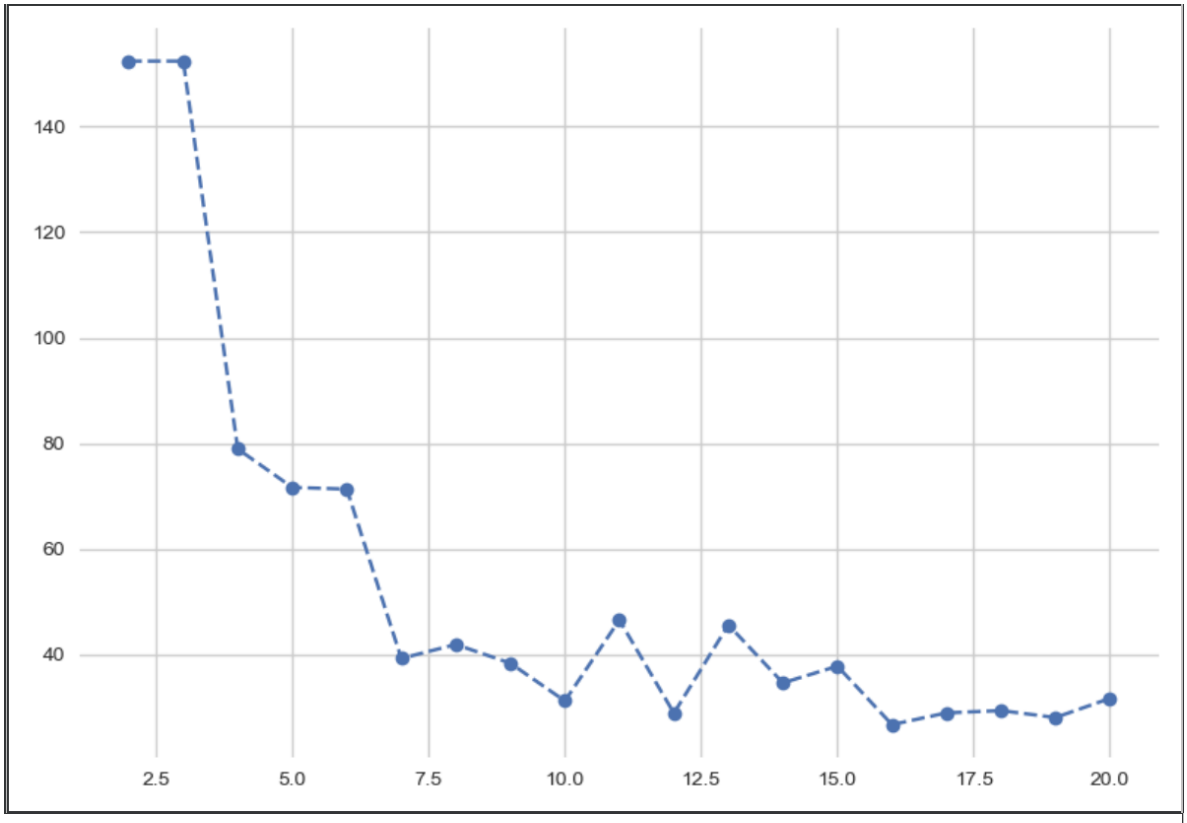
第四轮后还会重复迭代直到每个质心收敛为止。

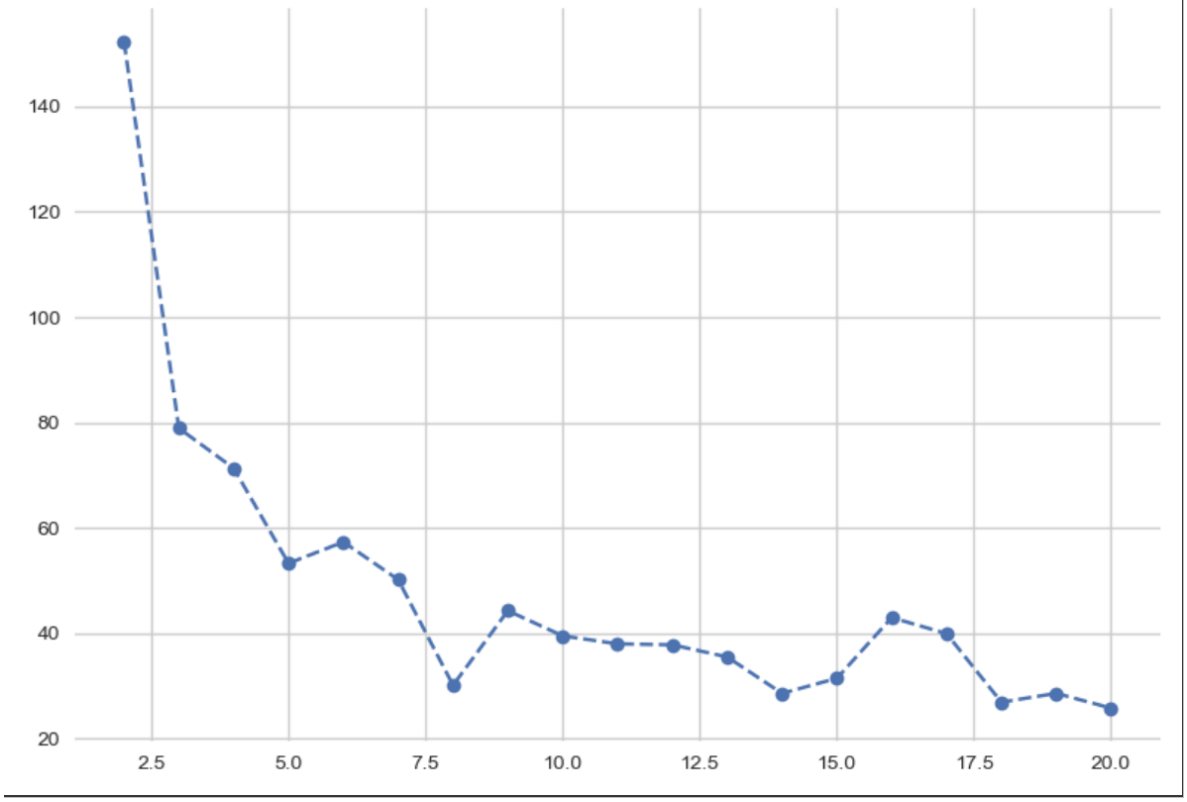
- **K-means算法的优缺点**
  - **K-means算法优点：**
    - 容易理解，聚类效果不错，虽然是局部最优， 但往往局部最优就够了；
    - 处理大数据集的时候，该算法可以保证较好的伸缩性；
    - 算法复杂度低。
  - **K-means算法缺点：**
    - K 值需要人为设定，不同 K 值得到的结果不一样，同时对异常值（噪声）敏感；
    - 对初始的簇中心敏感，不同选取方式会得到不同结果；
    - 样本只能归为一类，不适合多分类任务（即只属于n个类别中的某一类）；
    - 不适合太离散的分类、样本类别不平衡的分类、非凸形状的分类。
    - 在大规模的数据集上，该算法的收敛速度是比较慢的！
- **对K-means的补足：**
  1. 在初始化质心的时候，尽可能地让质心更加分散；
  2. 或在观察玩数据集后，人工地设置初始质心；
  3. 增量地更新质心，一种常见的方法即**二分K-均值法**（为克服K-Means算法收敛于局部最小值问题）其主要参考指标为**误差平方和**（SSE, Sum of Squard Error），二分K-Means算法**首先将所有点作为一个簇**，然后将该簇一分为二。之后选择其中一个簇继续进行划分，选择哪一个簇进行划分取决于对其划分是否可以最大程度降低SSE的值。上述基于SSE的划分过程不断重复，直到得到用户指定的簇数目为止。
- **对K-means实现的感悟：**

```
# 手动实现
# 导入文件
```

```
iris = pd.read_csv("E:\\All_Test_Files\\Py_File\\data\\iris.txt", header = None)
# print(iris.head())
# print(iris.shape)

# 计算欧式距离
def distance(arrA, arrB):
    d = arrA - arrB
    dist = np.sum(np.power(d, 2), axis = 1)   # 对列求和
    return dist
# 这里不开方是为了方便后面算SSE

# 随机生成 k 个质心
def randCent(dataset, k):
    n = dataset.shape[1]
    data_min = dataset.iloc[:, :n-1].min()
    data_max = dataset.iloc[:, :n-1].max()
    # 在列中的最小值和最大值之间选取随机质心
    data_cent = np.random.uniform(data_min, data_max, (k, n - 1))
    # 几个特征就几列
    return data_cent
# iris_cent = randCent(iris, 3)
# print(iris_cent)

# 算法主体
def KMeans(dataset, k, distMeans = distance, createCent = randCent):
    m, n = dataset.shape
    centroids = createCent(dataset, k)
    clusterAss = np.zeros((m, 3))  # 初始化
    clusterAss[:, 0] = np.inf # 第 0 列的距离初始化为无穷大
    clusterAss[:, 1: 3] = -1
    # 拼接数据
    result_set = pd.concat([dataset, pd.DataFrame(clusterAss)], axis = 1, ignore_index = True)
    clusterChanged = True
    while clusterChanged:
        clusterChanged = False
        for i in range(m):
            # 计算距离
            dist = distMeans(dataset.iloc[i, :n - 1].values, centroids)
            result_set.iloc[i, n] = dist.min()
            result_set.iloc[i, n + 1] = np.where(dist == dist.min())[0]
        # 取反:
        clusterChanged = not (result_set.iloc[:, -1] == result_set.iloc[:, -2]).all()
        # 最后一列放的是上一次迭代结果，倒数第二列放的是本次迭代结果，比较他们的误差是否足够小（相等）:
        if clusterChanged:
            cent_df = result_set.groupby(n + 1).mean() # 质心更新
            centroids = cent_df.iloc[:, :n-1].values
            result_set.iloc[: , -1] = result_set.iloc[:, -2]  # 迭代结果更新
    return centroids, result_set

iris_cent, iris_result = KMeans(iris, 3)
#print(iris_cent)
#print(iris_result.iloc[:, -1].value_counts())

```

```
56
57  # 模型评估
58  print("误差平方和：", iris_result.iloc[:, 5].sum())
59
60  # 刻画学习曲线，学习二十次
61  def KMLearningCurve(dataset, cluster = KMeans, k = 20):
62      n = dataset.shape[1]
63      SSE = []
64      for i in range(1, k):
65          centroids, result_set = cluster(dataset, i + 1)
66          SSE.append(result_set.iloc[:, n].sum())
67      plt.plot(range(2, k + 1), SSE, '--o')
68      # 直接从第二次开始画
69      plt.show()
70      return SSE
71
72  KMLearningCurve(iris)
73
74  '''
75  最终的曲线刻画出来的结果是不太稳定的（在3-4之间），故k选3和选4都可以
76  '''
77
```





本套K-means算法是最基础的模式，并未采用二分k均值法，总体思路已在前文大致地进行了阐述，在这个过程中，我觉得最精髓的就是对末尾两次迭代结果的比较看看是否质心收敛，然后不收敛的话就来一个取反判断。同时，这个算法是比较耗时的，特别是在最后刻画学习曲线时就可以体会到其收速度的快慢。

3. Apriori算法的总结：

- **先引入几个概念：**
    - **支持度（Support）**

关联规则A->B的支持度support=P(AB)，指的是事件A和事件B同时发生的概率（相当于联合概率）。

同理多个事件的支持度等于，多个时间同时发生的概率。在实际使用过程中，我们需要先设置一个支持度的阈值来进行项集的选择。

- **置信度（confidence)**

  置信度confidence=P(B|A)=P(AB)/P(A),指的是发生事件A的基础上发生事件B的概率（相当于条件概率）。置信度confidence=P(B|A)=P(AB)/P(A),指的是发生事件A的基础上发生事件B的概率（相当于条件概率）。

- **频繁k项集**

  顾名思义，频繁项集表示的就是在数据集中频繁出现的项集（可以是一个，也可以是多个）如果事件A中包含k个元素，那么称这个事件A为k项集，并且事件A满足最小支持度阈值的事件称为频繁k项集。

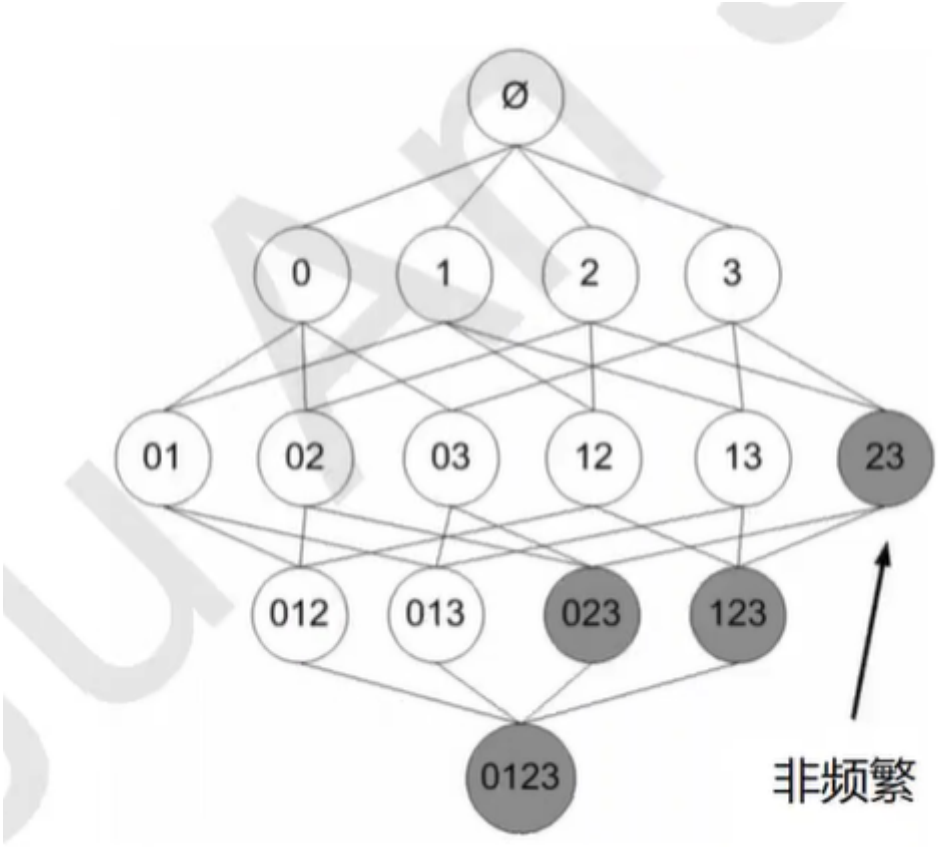
- **Apriori算法原理**

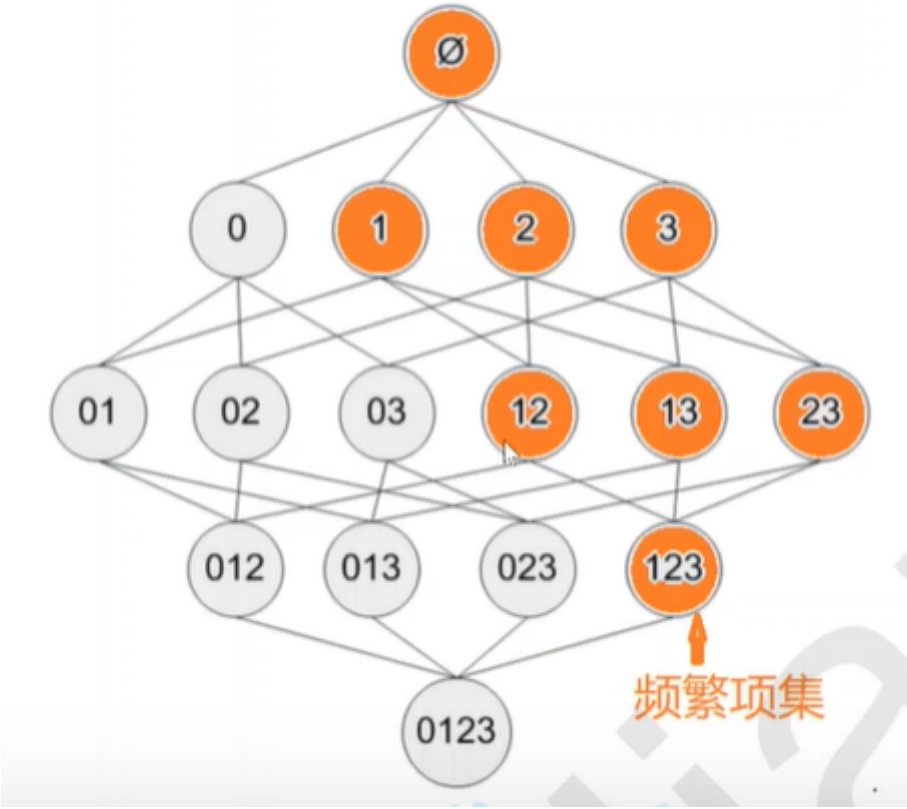
Apriori算法是一种关联规则挖掘算法，是为了找出物品之间的关联规则（潜在联系），通过上面我们知道，支持度越高，说明越相关联。那么支持度怎么决定呢？这个是我们主观决定的，我们会给Apriori提供一个最小支持度参数，然后Apriori会返回比这个最小支持度高的那些频繁项集。但是实际上，我们不大可能遍历计算所有的关联规则（量太大），所以就有了下面这句核心的话语：

某个项集是频繁的，那么它的所有子集也是频繁的

这句话看起来是没什么用，但是反过来就很有用了：

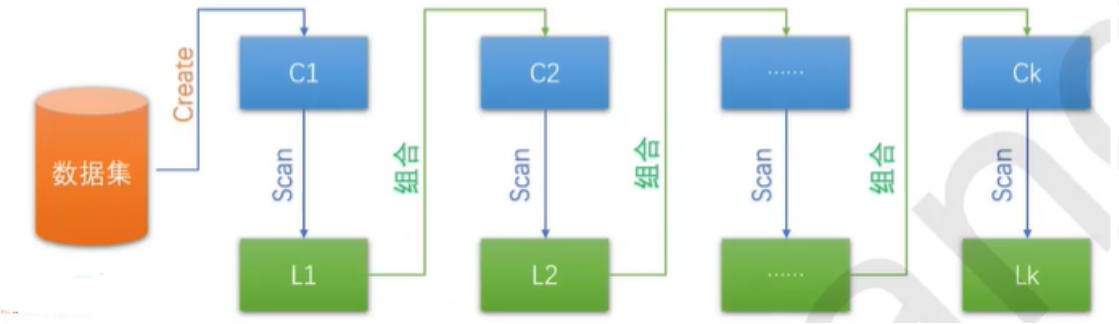
如果一个项集是 非频繁项集，那么它的所有超集也是非频繁项集。





- **算法流程：**

  要使用Apriori算法，我们需要提供两个参数，**数据集和最小支持度**。我们从前面已经知道了Apriori会遍历所有的物品组合，怎么遍历呢？答案就是递归。先遍历1个物品组合的情况，剔除掉支持度低于最小支持度的数据项，然后用剩下的物品进行组合。遍历2个物品组合的情况，再剔除不满足条件的组合。不断递归下去，直到不再有物品可以组合。



- **算法实现：**

```
def createC1(dataSet):
    # 构建初始候选项集的列表，即所有候选项集只包含一个元素，C1是大小为1的所有候选项集的集合

    C1 = []
    #遍历数据集合，并且遍历每一个集合中的每一项，创建只包含一个元素的候选项集集合
    for transaction in dataSet:
        for item in transaction:
            # 如果没有在C1列表中，则将该项的列表形式添加进去
            if not [item] in C1:
                C1.append([item])
    # 对列表进行排序
    C1.sort()
    # 固定列表C1，使其不可变
```

```
    return list(map(frozenset, C1))


def scanD(D,Ck,minSupport):      # 创建满足支持度要求的候选键集合

    # 定义存储每个项集在消费记录中出现的次数的字典
    ssCnt={}
    # 遍历这个数据集，并且遍历候选项集集合，判断候选项是否是一条记录的子集
    for tid in D:
        for can in Ck:
            if can.issubset(tid):
                # 如果是则累加其出现的次数
                if not can in ssCnt:
                    ssCnt[can]=1
                else: ssCnt[can]+=1
    # 计算数据集总及记录数
    numItems=float(len(D))
    # 定义满足最小支持度的候选项集列表
    retList = []
    # 用于所有项集的支持度
    supportData = {}
    # 遍历整个字典
    for key in ssCnt:
        # 计算当前项集的支持度
        support = ssCnt[key]/numItems
        # 如果该项集支持度大于最小要求，则将其头插至L1列表中
        if support >= minSupport:
            retList.insert(0,key)
        # 记录每个项集的支持度
        supportData[key] = support
    return retList, supportData


def aprioriGen(Lk, k):
    # 根据上一函数，创建由L1->C2的函数，也就是说需要将每个项集集合元素加1
    # Lk: 频繁项集列表
    # k: 项集元素个数

    # 存储Ck的列表
    retList = []
    # 获取lkPri长度，便于在其中遍历
    lenLk = len(Lk)
    # 两两遍历候选项集中的集合
    for i in range(lenLk):
        for j in range(i+1, lenLk):
            # 因为列表元素为集合，所以在比较前需要先将其转换为list,选择集合中前k-2个元素进行比较，如果相等，则对两个集合进行并操作
            # 这里可以保证减少遍历次数，并且可保证集合元素比合并前增加一个
            L1 = list(Lk[i])[:k-2]; L2 = list(Lk[j])[:k-2]
            # 对转化后的列表进行排序，便于比较
            L1.sort(); L2.sort()
            if L1==L2: #若两个集合的前k-2个项相同时,则将两个集合合并
                retList.append(Lk[i] | Lk[j]) #set union
    return retList


def apriori(dataSet, minSupport = 0.5):
    # 生成所有频繁项集函数
```

```
71
72      # 创建C1
73      C1 = createC1(dataSet)
74      # 对数据集进行转换，并调用函数筛选出满足条件的项集
75      D = list(map(set, dataSet))
76      L1, supportData = scanD(D, C1, minSupport)    # 单项最小支持度判断
    0.5，生成L1
77      # 定义存储所有频繁项集的列表
78      L = [L1]
79      k = 2 # 从2开始方便计算
80      # 迭代开始，生成所有满足条件的频繁项集（每次迭代项集元素个数加1）
81      # 迭代停止条件为，当频繁项集中包含了所有单个项集元素后停止
82      while (len(L[k-2]) > 0):#创建包含更大项集的更大列表,直到下一个大的项集为空
83          Ck = aprioriGen(L[k-2], k)
84          Lk, supK = scanD(D, Ck, minSupport)
85          supportData.update(supK)
86          # 更新supportData
87          # 不断的添加以项集为key，以项集的支持度为value的元素
88          # 将此次迭代产生的频繁集集合加入L中
89          L.append(Lk)
90          k += 1
91      return L, supportData
92
93  #生成关联规则
94  def genRules(L, supportData, minConf=0.7):
95      #频繁项集列表、包含那些频繁项集支持数据的字典、最小可信度阈值
96      bigRuleList = [] #存储所有的关联规则
97      for i in range(1, len(L)):  #只获取有两个或者更多集合的项目，从1,即第二个
    元素开始，L[0]是单个元素的
98          # 两个及以上的才可能有关联一说，单个元素的项集不存在关联问题
99          for freqSet in L[i]:
100             H1 = [frozenset([item]) for item in freqSet]
101             #该函数遍历L中的每一个频繁项集并对每个频繁项集创建只包含单个元素集合
    的列表H1
102             if (i > 1):
103             #如果频繁项集元素数目超过2,那么会考虑对它做进一步的合并
104                 rulesFromConseq(freqSet, H1, supportData, bigRuleList,
    minConf)
105             else:#第一层时，后件数为1
106                 calcConf(freqSet, H1, supportData, bigRuleList,
    minConf)# 调用函数
107     return bigRuleList
108
109 #生成候选规则集合：计算规则的可信度以及找到满足最小可信度要求的规则
110 def calcConf(freqSet, H, supportData, brl, minConf=0.7):
111     #针对项集中只有两个元素时，计算可信度
112     prunedH = []#返回一个满足最小可信度要求的规则列表
113     for conseq in H:#后件，遍历 H中的所有项集并计算它们的可信度值
114         conf = supportData[freqSet]/supportData[freqSet-conseq] #可信度
    计算，结合支持度数据
115         if conf >= minConf:
116             print (freqSet-conseq,'-->',conseq,'conf:',conf)
117             #如果某条规则满足最小可信度值,那么将这些规则输出到屏幕显示
118             brl.append((freqSet-conseq, conseq, conf))#添加到规则里，brl
    是前面通过检查的 bigRuleList
119             prunedH.append(conseq)#同样需要放入列表到后面检查
120     return prunedH
121
```

```python
#合并
def rulesFromConseq(freqSet, H, supportData, brl, minConf=0.7):
    #参数:一个是频繁项集,另一个是可以出现在规则右部的元素列表 H
    m = len(H[0])
    if (len(freqSet) > (m + 1)): #频繁项集元素数目大于单个集合的元素数
        Hmp1 = aprioriGen(H, m+1)#存在不同顺序、元素相同的集合，合并具有相同部分的集合
        Hmp1 = calcConf(freqSet, Hmp1, supportData, brl, minConf)#计算可信度
        if (len(Hmp1) > 1):
        #满足最小可信度要求的规则列表多于1,则递归来判断是否可以进一步组合这些规则
            rulesFromConseq(freqSet, Hmp1, supportData, brl, minConf)

# 创建函数进行测试
if __name__ == '__main__':
    dataSet = loadDataSet()
    dataSet = dataSet.values
    dataSet = dataSet.tolist()
    t1 = time()
    L, suppData = apriori(dataSet, minSupport=0.9)
    rules = genRules(L, suppData, minConf=0.9)
    t2 = time()
    time = t2 - t1
    print(f"耗时：{time}秒")

# Apriori 由于因为是逐层搜索(可能的关联规则非常多），所以耗时特别大。时间复杂度比较大

'''
本测试数据来自mushroom数据集
经过多次支持度和置信度的修改，其运行耗时有比较明显的不同，但总的来说
支持度越低耗时越久，因为这种情况下关联规则多，计算耗时长!
'''
```

这套算法最精髓的地方就在于对频繁项集的确定以及关联规则的生成，事实上大体思路是清晰的，即扫描-->判断-->合并-->迭代至整个算法结束，但这套算法在实际应用中有一个缺点就是有可能我们得到的关联规则是无用的，即使他们之间的关联非常强，如闹钟和电池，通过这套算法，我们想真正的到的有用信息是那些以前从未出现过的关联规则，这才是这套算法应用的价值所在。