挖掘频繁项，项集，子序列或其他子结构通常是分析大规模数据集的第一步，这是数据挖掘多年来一直活跃的研究课题。spark.mllib提供了FP-growth的并行实现，这是一种挖掘频繁项集的流行算法。

# 1.FP-growth

[Han et al., Mining frequent patterns without candidate generation](http://dx.doi.org/10.1145/335191.335372)这篇论文描述了FP-growth算法，论文中“FP”代表频繁模式。给定交易数据集，FP-growth的第一步是计算项目频率并识别频繁项目。与为相同目的而设计的类Apriori算法不同，FP-growth的第二步使用后缀树（FP-tree）结构来编码事务，而不显式生成候选集，这通常很难生成。在第二步之后，可以从FP树中提取频繁项集。在spark.mllib中，在论文 [Li et al., PFP: Parallel FP-growth for query recommendation](http://dx.doi.org/10.1145/1454008.1454027).中我们实现了一个名为PFP的FP-growth的并行版本。PFP基于事务的后缀分配增长的FP树的工作，因此比单机实现更具可扩展性。

spark.mllib的FP-growth实现采用以下（超）参数：

* minSupport：最低支持度

例如，如果一个项目出现在5个交易中的3个中，则它具有3/5 = 0.6的支持。

* numPartitions：用于分发工作的分区数。

FPGrowth类实现了FP-growth算法。它需要一个事务的JavaRDD，其中每个事务都是一个泛型类型的Iterable项。使用事务调用FPGrowth.run会返回FPGrowthModel，该FPGrowthModel存储频繁项目集及其频率。以下示例说明了如何挖掘频繁项集和关联规则：

|  |
| --- |
| **package** com.cb.spark.mllib;  **import** java.util.Arrays;  **import** java.util.List;  **import** org.apache.spark.SparkConf;  **import** org.apache.spark.api.java.JavaRDD;  **import** org.apache.spark.api.java.JavaSparkContext;  **import** org.apache.spark.mllib.fpm.AssociationRules;  **import** org.apache.spark.mllib.fpm.FPGrowth;  **import** org.apache.spark.mllib.fpm.FPGrowthModel;  **public** **class** JavaSimpleFPGrowth {  **public** **static** **void** main(String[] args) {  SparkConf conf = **new** SparkConf().setAppName("JavaSimpleFPGrowth").setMaster("local");  JavaSparkContext sc = **new** JavaSparkContext(conf);  String path = "F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\sample\_fpgrowth.txt";  JavaRDD<String> data = sc.textFile(path);  JavaRDD<List<String>> transactions = data.map(line -> Arrays.*asList*(line.split(" ")));  FPGrowth fpg = **new** FPGrowth();  fpg = fpg.setMinSupport(0.2).setNumPartitions(10);  FPGrowthModel<String> model = fpg.run(transactions);  **for** (FPGrowth.FreqItemset<String> itemset : model.freqItemsets().toJavaRDD().collect()) {  System.***out***.println("[" + itemset.javaItems() + "]," + itemset.freq());  }  **double** minConfidence = 0.8;  **for** (AssociationRules.Rule<String> rule : model.generateAssociationRules(minConfidence).toJavaRDD().collect()) {  System.***out***.println(rule.javaAntecedent() + "=>" + rule.javaConsequent() + "," + rule.confidence());  }  }  } |

# 2.关联规则

AssociationRules实现并行规则生成算法，用于构造具有单个项目作为结果的规则。

|  |
| --- |
| **package** com.cb.spark.mllib;  **import** java.util.Arrays;  **import** org.apache.spark.SparkConf;  **import** org.apache.spark.api.java.JavaRDD;  **import** org.apache.spark.api.java.JavaSparkContext;  **import** org.apache.spark.mllib.fpm.AssociationRules;  **import** org.apache.spark.mllib.fpm.AssociationRules.Rule;  **import** org.apache.spark.mllib.fpm.FPGrowth;  **public** **class** JavaAssociationRulesExample {  **public** **static** **void** main(String[] args) {  SparkConf conf = **new** SparkConf().setAppName("JavaSimpleFPGrowth").setMaster("local");  JavaSparkContext sc = **new** JavaSparkContext(conf);  JavaRDD<FPGrowth.FreqItemset<String>> freqItemSet = sc  .parallelize(Arrays.*asList*(**new** FPGrowth.FreqItemset<>(**new** String[] { "a" }, 15L),  **new** FPGrowth.FreqItemset<>(**new** String[] { "b" }, 35L),  **new** FPGrowth.FreqItemset<>(**new** String[] { "a", "b" }, 12L)));  AssociationRules arules = **new** AssociationRules().setMinConfidence(0.8);  JavaRDD<Rule<String>> results = arules.run(freqItemSet);  **for** (AssociationRules.Rule<String> rule : results.collect()) {  System.***out***.println(rule.javaAntecedent() + "=>" + rule.javaConsequent() + "," + rule.confidence());  }  }  } |