* [Summary statistics](http://spark.apache.org/docs/latest/mllib-statistics.html#summary-statistics)
* [Correlations](http://spark.apache.org/docs/latest/mllib-statistics.html#correlations)
* [Stratified sampling](http://spark.apache.org/docs/latest/mllib-statistics.html#stratified-sampling)
* [Hypothesis testing](http://spark.apache.org/docs/latest/mllib-statistics.html#hypothesis-testing)
  + [Streaming Significance Testing](http://spark.apache.org/docs/latest/mllib-statistics.html#streaming-significance-testing)
* [Random data generation](http://spark.apache.org/docs/latest/mllib-statistics.html#random-data-generation)
* [Kernel density estimation](http://spark.apache.org/docs/latest/mllib-statistics.html#kernel-density-estimation)

1. 概要统计(Summary statistics)

我们通过Statistics中提供的函数colStats为RDD [Vector]提供列摘要统计信息。

colStats（）返回MultivariateStatisticalSummary的一个实例，其中包含列的max，min，mean，variance和非零数，以及总计数。

|  |
| --- |
| SparkConf conf = **new** SparkConf().setMaster("local").setAppName("BasicStatistics");  JavaSparkContext context = **new** JavaSparkContext(conf);  JavaRDD<Vector> mat = context.parallelize(Arrays.*asList*(Vectors.*dense*(1.0, 10.0, 100.0),  Vectors.*dense*(2.0, 20.0, 200.0), Vectors.*dense*(3.0, 30.0, 300.0)));  // 计算列的摘要统计  MultivariateStatisticalSummary summary = Statistics.*colStats*(mat.rdd());  System.***out***.println(summary.mean());// 密集向量，表示每一列的平均值  System.***out***.println(summary.variance()); // 列方差  System.***out***.println(summary.numNonzeros());// 每一列的非零值数目    context.stop(); |

1. 相关性

计算两个数据系列之间的相关性是统计学中的常见操作。在spark.mllib中，我们提供了计算许多系列之间成对相关性的灵活性。目前支持的相关方法是Pearson和Spearman的相关性。

[Statistics](http://spark.apache.org/docs/latest/api/java/org/apache/spark/mllib/stat/Statistics.html) 提供了计算序列之间相关性的方法。根据输入类型，两个JavaDoubleRDD或JavaRDD <Vector>，输出将分别为Double或相关矩阵。

|  |
| --- |
| // 相关性计算  JavaDoubleRDD seriesX = context.parallelizeDoubles(Arrays.*asList*(1.0, 2.0, 3.0, 3.0, 5.0));  JavaDoubleRDD seriesY = context.parallelizeDoubles(Arrays.*asList*(11.0, 22.0, 33.0, 33.0, 555.0));  **double** correlation = Statistics.*corr*(seriesX.srdd(), seriesY.srdd(), "pearson");// 默认使用pearson相关性  System.***out***.println("相关性为：" + correlation);  JavaRDD<Vector> data = context.parallelize(Arrays.*asList*(Vectors.*dense*(1.0, 10.0, 100.0),  Vectors.*dense*(2.0, 20.0, 200.0), Vectors.*dense*(5.0, 33.0, 366.0)));  // 计算相关性矩阵  Matrix matrix = Statistics.*corr*(data.rdd());  System.***out***.println(matrix); |

1. 分层抽样(Stratified sampling)

与驻留在spark.mllib中的其他统计函数不同，可以对RDD的键值对执行分层抽样方法sampleByKey和sampleByKeyExact。对于分层抽样，可以将键视为标签，将值视为特定属性。例如，键可以是人或女人，或文档ID，并且相应的值可以是人口中的人的年龄列表或文档中的单词列表。sampleByKey方法将翻转硬币以决定是否对样本进行采样，因此需要对数据进行一次传递，并提供预期的样本大小。sampleByKeyExact比sampleByKey中使用的每层简单随机抽样需要更多的资源，但是会提供99.99％置信度的精确抽样大小。 python目前不支持sampleByKeyExact。

sampleByKeyExact（）允许用户准确地采样⌈fk⋅nk⌉∀k∈K项，其中fk是键k的期望分数，nk是键k的键值对的数量 ，K是一组键。

|  |
| --- |
| // 分层抽样  List<Tuple2<Integer, Character>> list = Arrays.*asList*(**new** Tuple2<Integer, Character>(1, 'a'),  **new** Tuple2<Integer, Character>(1, 'b'), **new** Tuple2<Integer, Character>(2, 'c'),  **new** Tuple2<Integer, Character>(2, 'd'), **new** Tuple2<Integer, Character>(2, 'e'),  **new** Tuple2<Integer, Character>(3, 'f'));  JavaPairRDD<Integer, Character> data1 = context.parallelizePairs(list);  ImmutableMap<Integer, Double> fractions = ImmutableMap.*of*(1, 0.1, 2, 0.6, 3, 0.3);  JavaPairRDD<Integer, Character> javaPairRDD = data1.sampleByKey(**false**, fractions);  JavaPairRDD<Integer, Character> javaPairRDD2 = data1.sampleByKey(**false**, fractions);  javaPairRDD.foreach(x -> System.***out***.print(x + " "));  System.***out***.println();  javaPairRDD2.foreach(x -> System.***out***.print(x + " "));  System.***out***.println(); |

1. 假设检验

假设检验是统计学中一种强有力的工具，用于确定结果是否具有统计显着性，无论该结果是否偶然发生。spark.mllib目前支持Pearson的卡方（χ2）检验，以确保拟合度和独立性。输入数据类型确定是否进行拟合优度或独立性测试。拟合优度测试需要输入类型的Vector，而独立性测试需要Matrix作为输入。spark.mllib还支持输入类型RDD [LabeledPoint]，以通过卡方独立测试启用特征选择。

[Statistics](http://spark.apache.org/docs/latest/api/java/org/apache/spark/mllib/stat/Statistics.html)提供了运行Pearson卡方检验的方法。 以下示例演示了如何运行和解释假设检验。

|  |
| --- |
| //假设检验  Vector vec=Vectors.*dense*(0.1,0.15,0.2,0.3,0.25);  //计算适合度。 如果未提供要测试的第二个向量作为参数，则测试针对均匀分布运行。  ChiSqTestResult chiSqTestResult=Statistics.*chiSqTest*(vec);  System.***out***.println(chiSqTestResult+"\n");  //创建应变矩阵（（1.0,2.0），（3.0,4.0），（5.0,6.0））  Matrices.*dense*(3, 2, **new** **double**[]{1.0,3.0,5.0,2.0,4.0,6.0});  //对输入应变矩阵进行Pearson独立性检验  ChiSqTestResult chiSqTestResult2=Statistics.*chiSqTest*(matrix);  System.***out***.println(chiSqTestResult2+"\n"); |

1. 流重要性测试

spark.mllib提供了一些测试的在线实现，以支持A / B测试等用例。这些测试可以在Spark Streaming DStream [（Boolean，Double）]上执行，其中每个元组的第一个元素表示控制组（false）或处理组（true），第二个元素是观察值。

流式重要性测试支持以下参数：

peacePeriod - 要忽略的流中的初始数据点数，用于缓解新奇效应。

windowSize - 执行假设检验的过去批次数。 设置为0将使用所有先前批次执行累积处理。

StreamingTest提供流式假设测试。

|  |
| --- |
| JavaDStream<BinarySample> data = ssc.textFileStream(dataDir).map(line -> {  String[] ts = line.split(",");  boolean label = Boolean.parseBoolean(ts[0]);  double value = Double.parseDouble(ts[1]);  **return** **new** BinarySample(label, value);  });  StreamingTest streamingTest = **new** StreamingTest()  .setPeacePeriod(0)  .setWindowSize(0)  .setTestMethod("welch");  JavaDStream<StreamingTestResult> out = streamingTest.registerStream(data);  out.print(); |

1. 随机数据生成

随机数据生成对于随机算法，原型设计和性能测试非常有用。spark.mllib支持使用i.i.d从给定分布绘制的值：均匀，标准正太或泊松分布生成随机RDD。RandomRDDs提供工厂方法来生成随机doubleRDD或vecors RDD。以下示例生成随机doubleRDD，其值遵循标准正态分布N（0,1），然后将其映射到N（1,4）。

|  |
| --- |
| // 随机数据生成  JavaDoubleRDD javaDoubleRDD = RandomRDDs.*normalJavaRDD*(context, 1000000L, 10);// 10  // partitions.  javaDoubleRDD.mapToDouble(x -> 1.0 + 2.0 \* x); |

1. 核密度估计

[Kernel density estimation](https://en.wikipedia.org/wiki/Kernel_density_estimation) 是一种可用于可视化经验概率分布的技术，无需假设观察到的样本的特定分布。它计算随机变量的概率密度函数的估计值，在给定的一组点处进行评估。它通过将特定点的经验分布的PDF表示为以每个样本为中心的正态分布的PDF的平均值来实现该估计。KernelDensity提供了从样本的RDD计算核密度估计的方法。 以下示例演示了如何执行此操作。

|  |
| --- |
| // 核密度估计  JavaRDD<Double> data2= context  .parallelize(Arrays.*asList*(1.0, 1.0, 1.0, 2.0, 3.0, 4.0, 5.0, 5.0, 6.0, 7.0, 8.0, 9.0, 9.0));  //使用样本数据和高斯核的标准偏差构造密度估计器  KernelDensity density=**new** KernelDensity().setSample(data2).setBandwidth(3.0);  //计算给定值的密度估计  **double**[] densitys=density.estimate(**new** **double**[]{-1.0,2.0,5.0});  System.***out***.println(Arrays.*toString*(densitys)); |