Basic Statistics - RDD-based API

- Summary statistics
- Correlations
- Stratified sampling
- Hypothesis testing
  - Streaming Significance Testing
- Random data generation
- Kernel density estimation

Summary statistics

We provide column summary statistics for `RDD[Vector]` through the function `colStats` available in `Statistics`.

Scala **Java** **Python**

`colStats()` returns an instance of `MultivariateStatisticalSummary`, which contains the column-wise max, min, mean, variance, and number of nonzeros, as well as the total count.

Refer to the `MultivariateStatisticalSummary` Python docs for more details on the API.

```python
import numpy as np

from pyspark.mllib.stat import Statistics

mat = sc.parallelize(
    [np.array([1.0, 10.0, 100.0]), np.array([2.0, 20.0, 200.0]), np.array([3.0, 30.0, 300.0])]
)  # an RDD of Vectors

# Compute column summary statistics.
summary = Statistics.colStats(mat)
print(summary.mean())  # a dense vector containing the mean value for each column
print(summary.variance())  # column-wise variance
print(summary.numNonzeros())  # number of nonzeros in each column
```

Find full example code at "examples/src/main/python/mllib/summary_statistics_example.py" in the Spark repo.

Correlations

Calculating the correlation between two series of data is a common operation in Statistics. In `spark.mllib` we provide the flexibility to calculate pairwise correlations among many series. The supported correlation methods are currently Pearson’s and Spearman’s correlation.

Scala **Java** **Python**

`Statistics` provides methods to calculate correlations between series. Depending on the type of input, two `RDD[Double]`s or an `RDD[Vector]`, the output will be a `Double` or the correlation `Matrix` respectively.

Refer to the `Statistics` Python docs for more details on the API.

»

```python
from pyspark.mllib.stat import Statistics

seriesX = sc.parallelize([1.0, 2.0, 3.0, 3.0, 5.0])  # a series
# seriesY must have the same number of partitions and cardinality as seriesX
seriesY = sc.parallelize([11.0, 22.0, 33.0, 33.0, 555.0])

# Compute the correlation using Pearson's method. Enter "spearman" for Spearman's method.
# If a method is not specified, Pearson's method will be used by default.
print("Correlation is: " + str(Statistics.corr(seriesX, seriesY, method="pearson")))

data = sc.parallelize(
    [np.array([1.0, 10.0, 100.0]), np.array([2.0, 20.0, 200.0]), np.array([5.0, 33.0, 366.0])]
)  # an RDD of Vectors

# calculate the correlation matrix using Pearson's method. Use "spearman" for Spearman's method.
# If a method is not specified, Pearson's method will be used by default.
print(Statistics.corr(data, method="pearson"))
```

Find full example code at "examples/src/main/python/mllib/correlations_example.py" in the Spark repo.

Stratified sampling

Unlike the other statistics functions, which reside in `spark.mllib`, stratified sampling methods, `sampleByKey` and `sampleByKeyExact`, can be performed on RDD's of key-value pairs. For stratified sampling, the keys can be thought of as a label and the value as a specific attribute. For example the key can be man or woman, or document ids, and the respective values can be the list of ages of the people in the population or the list of words in the documents. The `sampleByKey` method will flip a coin to decide whether an observation will be sampled or not, therefore requires one pass over the data, and provides an *expected* sample size. `sampleByKeyExact` requires significant more resources than the per-stratum simple random sampling used in `sampleByKey`, but will provide the exact sampling size with 99.99% confidence. `sampleByKeyExact` is currently not supported in python.

Scala **Java** **Python**

`sampleByKey()` allows users to sample approximately $\lceil f_k \cdot n_k \rceil \, \forall k \in K$ items, where f_k is the desired fraction for key k, n_k is the number of key-value pairs for key k, and K is the set of keys.

Note: `sampleByKeyExact()` is currently not supported in Python.

```python
# an RDD of any key value pairs
data = sc.parallelize([(1, 'a'), (1, 'b'), (2, 'c'), (2, 'd'), (2, 'e'), (3, 'f')])

# specify the exact fraction desired from each key as a dictionary
fractions = {1: 0.1, 2: 0.6, 3: 0.3}

approxSample = data.sampleByKey(False, fractions)
```

Find full example code at "examples/src/main/python/mllib/stratified_sampling_example.py" in the Spark repo.

Hypothesis testing

Hypothesis testing is a powerful tool in statistics to determine whether a result is statistically significant, whether this result occurred by chance or not. `spark.mllib` currently supports Pearson's chi-squared (χ^2) tests for goodness of fit and independence. The input data types determine whether the goodness of fit or the independence test is conducted. The goodness of fit test requires an input type of `Vector`, whereas the independence test requires a `Matrix` as input.

»

`spark.mllib` also supports the input type `RDD[LabeledPoint]` to enable feature selection via chi-squared independence tests.

Scala **Java** **Python**

`Statistics` provides methods to run Pearson's chi-squared tests. The following example demonstrates how to run and interpret hypothesis tests.

Refer to the `Statistics` Python docs for more details on the API.

```
from pyspark.mllib.linalg import Matrices, Vectors
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.stat import Statistics

vec = Vectors.dense(0.1, 0.15, 0.2, 0.3, 0.25)  # a vector composed of the frequencies of events

# compute the goodness of fit. If a second vector to test against
# is not supplied as a parameter, the test runs against a uniform distribution.
goodnessOfFitTestResult = Statistics.chiSqTest(vec)

# summary of the test including the p-value, degrees of freedom,
# test statistic, the method used, and the null hypothesis.
print("%s\n" % goodnessOfFitTestResult)

mat = Matrices.dense(3, 2, [1.0, 3.0, 5.0, 2.0, 4.0, 6.0])  # a contingency matrix

# conduct Pearson's independence test on the input contingency matrix
independenceTestResult = Statistics.chiSqTest(mat)

# summary of the test including the p-value, degrees of freedom,
# test statistic, the method used, and the null hypothesis.
print("%s\n" % independenceTestResult)

obs = sc.parallelize(
    [LabeledPoint(1.0, [1.0, 0.0, 3.0]),
     LabeledPoint(1.0, [1.0, 2.0, 0.0]),
     LabeledPoint(1.0, [-1.0, 0.0, -0.5])]
)  # LabeledPoint(feature, label)

# The contingency table is constructed from an RDD of LabeledPoint and used to conduct
# the independence test. Returns an array containing the ChiSquaredTestResult for every feature
# against the label.
featureTestResults = Statistics.chiSqTest(obs)
```

```
for i, result in enumerate(featureTestResults):
    print("Column %d:\n%s" % (i + 1, result))
```

Find full example code at "examples/src/main/python/mllib/hypothesis_testing_example.py" in the Spark repo.

Additionally, `spark.mllib` provides a 1-sample, 2-sided implementation of the Kolmogorov-Smirnov (KS) test for equality of probability distributions. By providing the name of a theoretical distribution (currently solely supported for the normal distribution) and its parameters, or a function to calculate the cumulative distribution according to a given theoretical distribution, the user can test the null hypothesis that their sample is drawn from that distribution. In the case that the user tests against the normal distribution (`distName="norm"`), but does not provide distribution parameters, the test initializes to the standard normal distribution and logs an appropriate message.

Scala Java **Python**

`Statistics` provides methods to run a 1-sample, 2-sided Kolmogorov-Smirnov test. The following example demonstrates how to run and interpret the hypothesis tests.

Refer to the `Statistics` Python docs for more details on the API.

```
from pyspark.mllib.stat import Statistics

parallelData = sc.parallelize([0.1, 0.15, 0.2, 0.3, 0.25])

# run a KS test for the sample versus a standard normal distribution
testResult = Statistics.kolmogorovSmirnovTest(parallelData, "norm", 0, 1)
# summary of the test including the p-value, test statistic, and null hypothesis
# if our p-value indicates significance, we can reject the null hypothesis
# Note that the Scala functionality of calling Statistics.kolmogorovSmirnovTest with
# a lambda to calculate the CDF is not made available in the Python API
print(testResult)
```

Find full example code at "examples/src/main/python/mllib/hypothesis_testing_kolmogorov_smirnov_test_example.py" in the Spark repo.

Streaming Significance Testing

`spark.mllib` provides online implementations of some tests to support use cases like A/B testing. These tests may be performed on a Spark Streaming `DStream[(Boolean,Double)]` where the first element of each tuple indicates control group (`false`) or treatment group (`true`) and the second element is the value of an observation.

Streaming significance testing supports the following parameters:

- `peacePeriod` - The number of initial data points from the stream to ignore, used to mitigate novelty effects.
- `windowSize` - The number of past batches to perform hypothesis testing over. Setting to `0` will perform cumulative processing using all prior batches.

Scala Java

`StreamingTest` provides streaming hypothesis testing.

```
val data = ssc.textFileStream(dataDir).map(line => line.split(",") match {
  case Array(label, value) => BinarySample(label.toBoolean, value.toDouble)
})

val streamingTest = new StreamingTest()
  .setPeacePeriod(0)
  .setWindowSize(0)
  .setTestMethod("welch")

val out = streamingTest.registerStream(data)
out.print()
```

»

Find full example code at "examples/src/main/scala/org/apache/spark/examples/mllib/StreamingTestExample.scala" in the Spark repo.

Random data generation

Random data generation is useful for randomized algorithms, prototyping, and performance testing. `spark.mllib` supports generating random RDDs with i.i.d. values drawn from a given distribution: uniform, standard normal, or Poisson.

Scala Java **Python**

`RandomRDDs` provides factory methods to generate random double RDDs or vector RDDs. The following example generates a random double RDD, whose values follows the standard normal distribution `N(0, 1)`, and then map it to `N(1, 4)`.

Refer to the `RandomRDDs` Python docs for more details on the API.

```
from pyspark.mllib.random import RandomRDDs

sc = ... # SparkContext

# Generate a random double RDD that contains 1 million i.i.d. values drawn from the
# standard normal distribution `N(0, 1)`, evenly distributed in 10 partitions.
u = RandomRDDs.normalRDD(sc, 1000000L, 10)
# Apply a transform to get a random double RDD following `N(1, 4)`.
v = u.map(lambda x: 1.0 + 2.0 * x)
```

Kernel density estimation

Kernel density estimation is a technique useful for visualizing empirical probability distributions without requiring assumptions about the particular distribution that the observed samples are drawn from. It computes an estimate of the probability density function of a random variables, evaluated at a given set of points. It achieves this estimate by expressing the PDF of the empirical distribution at a particular point as the mean of PDFs of normal distributions centered around each of the samples.

Scala Java **Python**

`KernelDensity` provides methods to compute kernel density estimates from an RDD of samples. The following example demonstrates how to do so.

»

Refer to the `KernelDensity` Python docs for more details on the API.

```
from pyspark.mllib.stat import KernelDensity

# an RDD of sample data
data = sc.parallelize([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])

# Construct the density estimator with the sample data and a standard deviation for the Gaussian
# kernels
kd = KernelDensity()
kd.setSample(data)
kd.setBandwidth(3.0)

# Find density estimates for the given values
densities = kd.estimate([-1.0, 2.0, 5.0])
```

Find full example code at "examples/src/main/python/mllib/kernel_density_estimation_example.py" in the Spark repo.