Statistical and Mathematical Functions with DataFrames in Spark

June 2, 2015 by Burak Yavuz and Reynold Xin

Join us at Spark Summit to hear more about new functionalities of Apache Spark. Use the code *Databricks20* to receive a 20% discount!

We introduced DataFrames in Spark 1.3 to make Apache Spark much easier to use. Inspired by data frames in R and Python, DataFrames in Spark expose an API that's similar to the single-node data tools that data scientists are already familiar with. Statistics is an important part of everyday data science. We are happy to announce improved support for statistical and mathematical functions in the upcoming 1.4 release.

In this blog post, we walk through some of the important functions, including:

1. Random data generation
2. Summary and descriptive statistics
3. Sample covariance and correlation
4. Cross tabulation (a.k.a. contingency table)
5. Frequent items
6. Mathematical functions

We use Python in our examples. However, similar APIs exist for Scala and Java users as well.

1. Random Data Generation

Random data generation is useful for testing of existing algorithms and implementing randomized algorithms, such as random projection. We provide methods under sql.functions for generating columns that contains i.i.d. values drawn from a distribution, e.g., uniform (`rand`), and standard normal (`randn`).

```
In [1]: from pyspark.sql.functions import rand, randn
In [2]: # Create a DataFrame with one int column and 10 rows.
In [3]: df = sqlContext.range(0, 10)
In [4]: df.show()
+--+
|id|
+--+
| 0|
| 1|
| 2|
| 3|
| 4|
| 5|
| 6|
| 7|
| 8|
| 9|
+---+

In [4]: # Generate two other columns using uniform distribution and normal d
istribution.
In [5]: df.select("id", rand(seed=10).alias("uniform"), randn(seed=27).alias
("normal")).show()
+---+-------------------+--------------------+
|id|            uniform|              normal|
+---+-------------------+--------------------+
| 0| 0.7224977951905031| -0.1875348803463305|
| 1| 0.2953174992603351|-0.26525647952450265|
| 2| 0.4536856090041318| -0.7195024130068081|
| 3| 0.9970412477032209|  0.5181478766595276|
| 4|0.19657711634539565|  0.7316273979766378|
| 5|0.48533720635534006| 0.07724879367590629|
| 6| 0.7369825278894753| -0.5462256961278941|
| 7| 0.5241113627472694| -0.2542275002421211|
| 8| 0.2977697066654349| -0.5752237580095868|
| 9| 0.5060159582230856|  1.0900096472044518|
+---+-------------------+--------------------+
```

2. Summary and Descriptive Statistics

The first operation to perform after importing data is to get some sense of what it looks like. For numerical columns, knowing the descriptive summary statistics can help a lot in understanding the distribution of your data. The function `describe` returns a DataFrame containing information such as number of non-null entries (count), mean, standard deviation, and minimum and maximum value for each numerical column.

```
In [1]: from pyspark.sql.functions import rand, randn
In [2]: # A slightly different way to generate the two random columns
In [3]: df = sqlContext.range(0, 10).withColumn('uniform', rand(seed=10)).wi
thColumn('normal', randn(seed=27))
```

```
In [4]: df.describe().show()
+-------+------------------+-------------------+--------------------+
|summary|                id|            uniform|              normal|
+-------+------------------+-------------------+--------------------+
|  count|                10|                 10|                  10|
|   mean|               4.5| 0.5215336029384192|-0.01309370117407197|
| stddev|2.8722813232690143|  0.229328162820653|  0.5756058014772729|
|    min|                 0|0.19657711634539565| -0.7195024130068081|
|    max|                 9| 0.9970412477032209|  1.0900096472044518|
+-------+------------------+-------------------+--------------------+
```

If you have a DataFrame with a large number of columns, you can also run describe on a subset of the columns:

```
In [4]: df.describe('uniform', 'normal').show()
+-------+-------------------+--------------------+
|summary|            uniform|              normal|
+-------+-------------------+--------------------+
|  count|                 10|                  10|
|   mean| 0.5215336029384192|-0.01309370117407197|
| stddev|  0.229328162820653|  0.5756058014772729|
|    min|0.19657711634539565| -0.7195024130068081|
|    max| 0.9970412477032209|  1.0900096472044518|
+-------+-------------------+--------------------+
```

Of course, while describe works well for quick exploratory data analysis, you can also control the list of descriptive statistics and the columns they apply to using the normal select on a DataFrame:

```
In [5]: from pyspark.sql.functions import mean, min, max
In [6]: df.select([mean('uniform'), min('uniform'), max('uniform')]).show()
+------------------+-------------------+------------------+
|      AVG(uniform)|       MIN(uniform)|      MAX(uniform)|
+------------------+-------------------+------------------+
|0.5215336029384192|0.19657711634539565|0.9970412477032209|
+------------------+-------------------+------------------+
```

3. Sample covariance and correlation

Covariance is a measure of how two variables change with respect to each other. A positive number would mean that there is a tendency that as one variable increases, the other increases as well. A negative number would mean that as one variable increases, the other variable has a tendency to decrease. The sample covariance of two columns of a DataFrame can be calculated as follows:

```
In [1]: from pyspark.sql.functions import rand
In [2]: df = sqlContext.range(0, 10).withColumn('rand1', rand(seed=10)).with
Column('rand2', rand(seed=27))

In [3]: df.stat.cov('rand1', 'rand2')
Out[3]: 0.009908130446217347

In [4]: df.stat.cov('id', 'id')
Out[4]: 9.166666666666666
```

As you can see from the above, the covariance of the two randomly generated columns is close to zero, while the covariance of the id column with itself is very high.

The covariance value of 9.17 might be hard to interpret. Correlation is a normalized measure of covariance that is easier to understand, as it provides quantitative measurements of the statistical dependence between two random variables.

```
In [5]: df.stat.corr('rand1', 'rand2')
Out[5]: 0.14938694513735398

In [6]: df.stat.corr('id', 'id')
Out[6]: 1.0
```

In the above example, id correlates perfectly with itself, while the two randomly generated columns have low correlation value.

4. Cross Tabulation (Contingency Table)

Cross Tabulation provides a table of the frequency distribution for a set of variables. Cross-tabulation is a powerful tool in statistics that is used to observe the statistical significance (or independence) of variables. In Spark 1.4, users will be able to cross-tabulate two columns of a DataFrame in order to obtain the counts of the different pairs that are observed in those columns. Here is an example on how to use crosstab to obtain the contingency table.

```
In [1]: # Create a DataFrame with two columns (name, item)
In [2]: names = ["Alice", "Bob", "Mike"]
In [3]: items = ["milk", "bread", "butter", "apples", "oranges"]
In [4]: df = sqlContext.createDataFrame([(names[i % 3], items[i % 5]) for i
in range(100)], ["name", "item"])
```

```
In [5]: # Take a look at the first 10 rows.
In [6]: df.show(10)
+-----+-------+
| name|   item|
+-----+-------+
|Alice|   milk|
|  Bob|  bread|
| Mike| butter|
|Alice| apples|
|  Bob|oranges|
| Mike|   milk|
|Alice|  bread|
|  Bob| butter|
| Mike| apples|
|Alice|oranges|
+-----+-------+

In [7]: df.stat.crosstab("name", "item").show()
+---------+----+-----+------+------+-------+
|name_item|milk|bread|apples|butter|oranges|
+---------+----+-----+------+------+-------+
|      Bob|   6|    7|     7|     6|      7|
|     Mike|   7|    6|     7|     7|      6|
|    Alice|   7|    7|     6|     7|      7|
+---------+----+-----+------+------+-------+
```

One important thing to keep in mind is that the cardinality of columns we run crosstab on cannot be too big. That is to say, the number of distinct “name” and “item” cannot be too large. Just imagine if “item” contains 1 billion distinct entries: how would you fit that table on your screen?!

5. Frequent Items

Figuring out which items are frequent in each column can be very useful to understand a dataset. In Spark 1.4, users will be able to find the frequent items for a set of columns using DataFrames. We have implemented an one-pass algorithm proposed by Karp et al. This is a fast, approximate algorithm that always return all the frequent items that appear in a user-specified minimum proportion of rows. Note that the result might contain false positives, i.e. items that are not frequent.

```
In [1]: df = sqlContext.createDataFrame([(1, 2, 3) if i % 2 == 0 else (i, 2
* i, i % 4) for i in range(100)], ["a", "b", "c"])
```

```
In [2]: df.show(10)
+-+--+-+
|a| b|c|
+-+--+-+
|1| 2|3|
|1| 2|1|
|1| 2|3|
|3| 6|3|
|1| 2|3|
|5|10|1|
|1| 2|3|
|7|14|3|
|1| 2|3|
|9|18|1|
+-+--+-+

In [3]: freq = df.stat.freqItems(["a", "b", "c"], 0.4)
```

Given the above DataFrame, the following code finds the frequent items that show up 40% of the time for each column:

```
In [4]: freq.collect()[0]
Out[4]: Row(a_freqItems=[11, 1], b_freqItems=[2, 22], c_freqItems=[1, 3])
```

As you can see, “11” and “1” are the frequent values for column “a”. You can also find frequent items for column combinations, by creating a composite column using the struct function:

```
In [5]: from pyspark.sql.functions import struct

In [6]: freq = df.withColumn('ab', struct('a', 'b')).stat.freqItems(['ab'],
0.4)

In [7]: freq.collect()[0]
Out[7]: Row(ab_freqItems=[Row(a=11, b=22), Row(a=1, b=2)])
```

From the above example, the combination of “a=11 and b=22”, and “a=1 and b=2” appear frequently in this dataset. Note that “a=11 and b=22” is a false positive.

6. Mathematical Functions

Spark 1.4 also added a suite of mathematical functions. Users can apply these to their columns with ease. The list of math functions that are supported come from this file (we will also post prebuilt documentation once 1.4 is released). The inputs need to be columns functions that take a single argument, such as `cos`, `sin`, `floor`, `ceil`. For functions that take two arguments as input, such as `pow`, `hypot`, either two columns or a combination of a double and column can be supplied.

```
In [1]: from pyspark.sql.functions import *
In [2]: df = sqlContext.range(0, 10).withColumn('uniform', rand(seed=10) * 3
.14)

In [3]: # you can reference a column or supply the column name
In [4]: df.select(
   ...:   'uniform',
   ...:   toDegrees('uniform'),
   ...:   (pow(cos(df['uniform']), 2) + pow(sin(df.uniform), 2)). \
   ...:     alias("cos^2 + sin^2")).show()

+--------------------+------------------+------------------+
|             uniform|  DEGREES(uniform)|     cos^2 + sin^2|
+--------------------+------------------+------------------+
|  0.7224977951905031| 41.39607437192317|               1.0|
|  0.3312021111290707|18.976483133518624|0.9999999999999999|
|  0.2953174992603351|16.920446323975014|               1.0|
|0.018326130186194667| 1.050009914476252|0.9999999999999999|
|  0.3163135293051941|18.123430232075304|               1.0|
|  0.4536856090041318| 25.99427062175921|               1.0|
|   0.873869321369476| 50.06902396043238|0.9999999999999999|
|  0.9970412477032209| 57.12625549385224|               1.0|
| 0.19657711634539565| 11.26303911544332|1.0000000000000002|
|  0.9632338825504894| 55.18923615414307|               1.0|
+--------------------+------------------+------------------+
```

What's Next?

All the features described in this blog post will be available in Spark 1.4 for Python, Scala, and Java, to be released in the next few days. If you can't wait, you can also build Spark from the 1.4 release branch yourself: https://github.com/apache/spark/tree/branch-1.4

Statistics support will continue to increase for DataFrames through better integration with Spark MLlib in future releases. Leveraging the existing Statistics package in MLlib, support for feature selection in pipelines, Spearman Correlation, ranking, and aggregate functions for covariance and correlation.

At the end of the blog post, we would also like to thank Davies Liu, Adrian Wang, and rest of the

Spark community for implementing these functions.

databricks.com

https://databricks.com/blog/2015/06/02/statistical-and-mathematical-functions-with-dataframes-in-spark.html

http://goo.gl/ffgS