

Continuous Distribution Goodness-of-Fit in MLlib

Jose Cambronero

Data's portrait

Data can come in many shapes and forms, with an equally large amount of ways to describe them. Most people have dealt with averages and standard deviation. These are in effect *shorthand* for that description. Sometimes we might want more information, more perspective into that data's profile. In a situation like that we might look at other measures, such as [skewness](#) and [kurtosis](#). However, sometimes we can provide a much neater description for data by stating it comes from a given distribution, which not only tells us things like the average value that we should expect, but effectively gives us the data's "recipe" so that we can compute all sorts of useful information from it. As part of my summer internship at Cloudera I've added implementations to Spark's MLlib library of various statistical tests that can help us draw conclusions regarding how well a distribution fits data. Specifically, the implementations pertain to the JIRAS: [1-sample, two-sided Kolmogorov-Smirnov test](#), [2-sample, two-sided Kolmogorov-Smirnov test](#), and [1-sample, two-sided Anderson-Darling test](#).

Testing in big-data world

The world of small-data analytics has many tools to accomplish this, ranging from quantitative measures to more graphical approaches. R, a popular statistical programming language, provides many of the usual measures out-of-the box, and there are a plethora of packages that provide additional goodness-of-fit tests and visualization tools to aid data scientists in determining whether a sample comes from a given distribution (or isn't different enough to warrant ruling that out). The same is true for Python, using libraries such as SciPy.

However, these tools are rarely able to natively handle the volume of data associated with big-data problems. Unless users want to go with the alternative of testing subsamples of their data, they have to turn to platforms designed to for such big-data tasks. For example, consider a sample of 10,000,000 observations. This immediately rules out using any graphical tools unless you perform something like binning, and then you're potentially throwing out information (which was the whole point of collecting that much data to begin with!).

Apache Spark's MLlib can help analytics' users scratch this itch by providing implementations of various popular statistical measures that can handle large-scale data. For example, the Statistics object in MLlib provides various implementations of the Chi-Square test, both as a test for independence and as a goodness-of-fit test. For example, if you were to collect values and wanted to test these against expected values from a given discrete distribution, in order to draw conclusions about whether the data stems from that distribution, we could use the `ChiSqTest(Vector, Vector)` implementation.

As useful and powerful as the Chi-Squared test is with discrete data, applying it to continuous data is inappropriate. Thus there has been a need for continuous distribution goodness-of-fit testing for big-data needs.

Innocent until proven guilty (read: null until proven alternate!)

TODO: trim down

We might have various distributions that we think fit our data. For example, we might have just fit a linear regression to a dataset, and believe that our residuals are white noise (i.e. standard normal), and we would now like some assurance that this is the case and we're not missing some important features in our model. We now have the task of testing whether our empirical data actually follow these theoretical distributions.

It is often difficult to conclude that a given hypothesis is correct. However, the converse, concluding that a given hypothesis is wrong, is actually much simpler. This idea might initially seem a bit counterintuitive, but a simple example can clear things up. Let's say a bad back is a common (clearly not professional!) diagnosis for back pain. If I have back pain, it is possible that I might have a bad back. It is also possible that I simply sat for too long. However, if I don't have back pain, it is fairly certain that I don't have a bad back. This drives the underlying intuition behind the concept of a null hypothesis and statistical tests for that hypothesis.

We can use this hammer and nail, a statistic and a hypothesis test, to do something similar with our potential distributions. In fact, there are various standard statistical tests that allow us to analyze whether a given sample might come from a given theoretical distribution (or alternatively, whether 2 samples come from the same, unknown, distribution). We'll discuss 2 of these in depth in this blog post: the Kolmogorov-Smirnov test, and the Anderson-Darling test. While "passing" the test doesn't guarantee that the data comes from that distribution, failing it is a pretty surefire way to demonstrate that it does not.

Statistics: Goodness-of-Fit tests

Kolmogorov-Smirnov

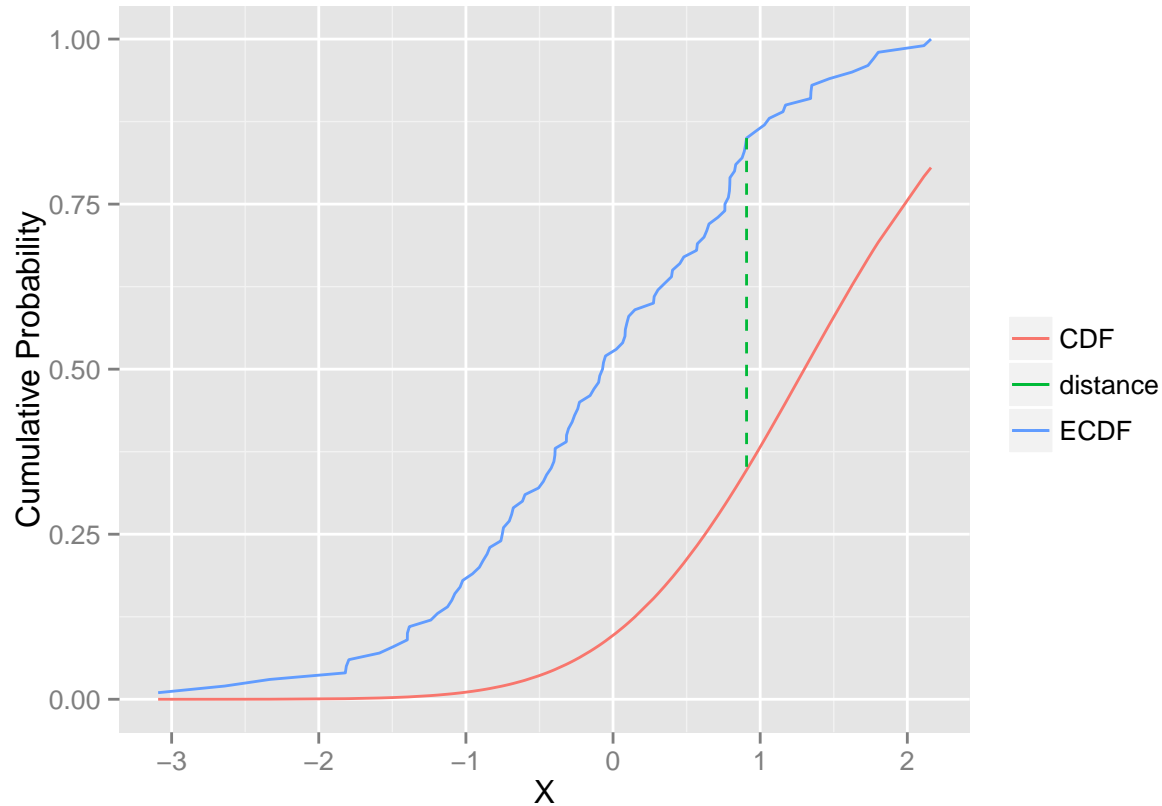
This might be one of the most popular goodness-of-fit tests for continuous data out there, with an implementation in pretty much every popular analytics platform. The intuition behind the test centers around comparing the largest deviation between the cumulative distribution at a given value X under the theoretical distribution and the [empirical cumulative distribution](#).

The Kolmogorov-Smirnov statistic for the 1-sample, 2-sided statistic is defined as

$$D = \max_{i=1}^n \left(\Phi(Z_i) - \frac{i-1}{n}, \frac{i}{n} - \Phi(Z_i) \right)$$

where N is the size of the sample, Z is the sorted sample, and Φ represents the cumulative distribution function for the theoretical distribution that we want to test.

The general intuition is captured by the graphic below. The test tries to capture the largest difference between the 2 curves. Then, given the distribution of the statistic itself, we can make some claims regarding how likely such a distance would be assuming that the null hypothesis (i.e. that the data comes from the distribution) holds.



One of the main appeals of this test centers around the fact that the test is distribution-agnostic. By that we mean that the statistic can be calculated and compared the same way regardless of the theoretical distribution we are interested in comparing against (or the actual distributions of the 1 or 2 samples, depending on the test variant being performed).

Anderson-Darling

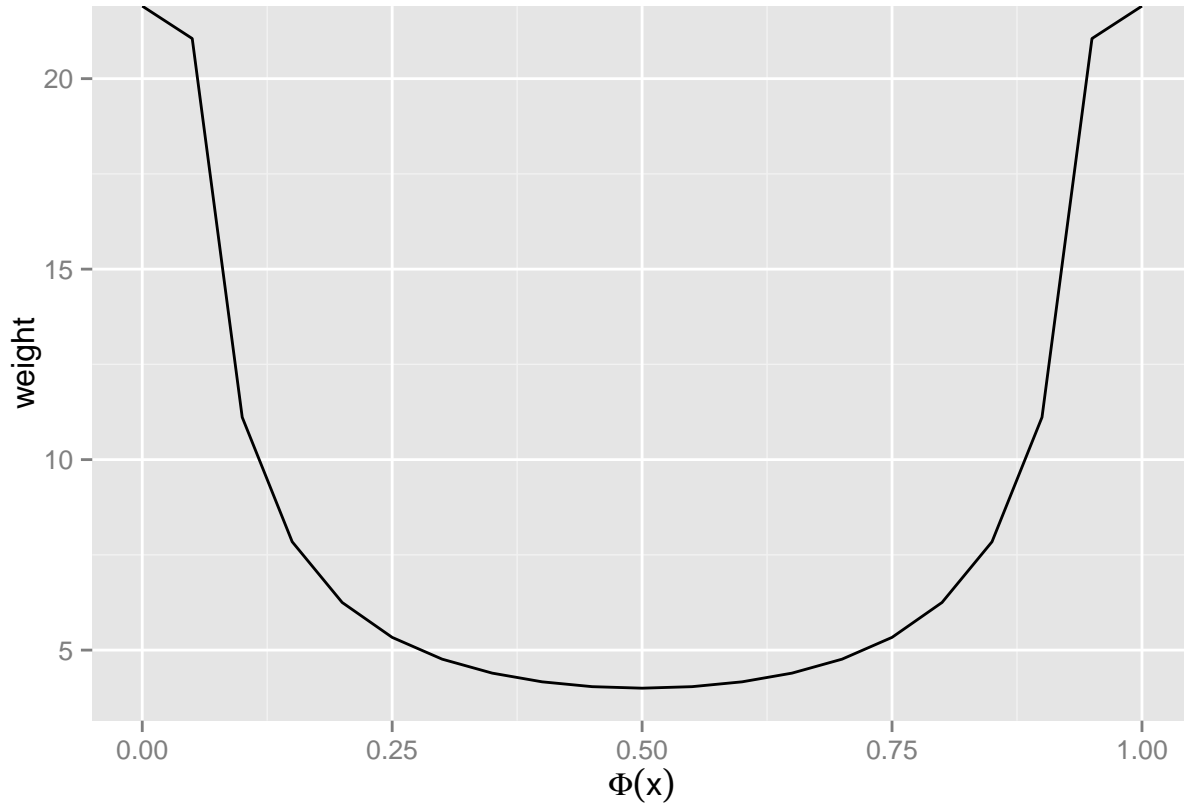
Anderson-Darling is often proposed as an alternative to the Kolmogorov-Smirnov statistic, with the advantage that it is better suited to identify departures from the theoretical distribution at the tails, and is more robust towards the nuances associated with estimating the distribution's parameters directly from the sample that we're trying to test.

In contrast to the Kolmogorov-Smirnov test, the Anderson-Darling test for 1-sample has critical values (i.e. the reference points that we will use to accept or dismiss our null hypothesis) that depend on the distribution we are testing against.

The informal intuition behind Anderson-Darling is that *not all distances are equal*, meaning that a deviation of size m between the empirical CDF curve and the CDF curve should carry different importance depending on where it happens on the curve. Indeed, a small deviation at the start of the CDF or the end could signal that the sample doesn't really stem from that distribution. Why is that? Well, consider that the CDF tends to 0 at the left, and 1 at the right, so there **shouldn't** be any real discrepancies there, for the most part.

Specifically, the Anderson-Darling test weighs the square of the deviations by $\frac{1}{\Phi(x)(1-\Phi(x))}$, where Φ is once again the CDF.

The graph below shows the point we made earlier, departures at the tails are weighed more heavily, and contribute to the overall sum, even if they're nominally small.



Let's take a look at the formula definition of Anderson-Darling (for brevity sake, we will solely consider the computational formula, not the formal version which involves an integral from $-\infty$ to ∞).

$$A = -N - \frac{1}{N} \sum_{i=1}^n (2i+1) [\ln(\Phi(Z_i)) + \ln(1 - \Phi(Z_{n+1-i}))]$$

Implementations

Distilling distributed distribution (statistics) (now say that three times fast...)

Calculating both of these statistics is extremely straightforward when performed in memory. If we have all our data, we can simply traverse it as needed and calculate what we need at each point. Performing the same calculations in a distributed setting can take a bit more craftiness.

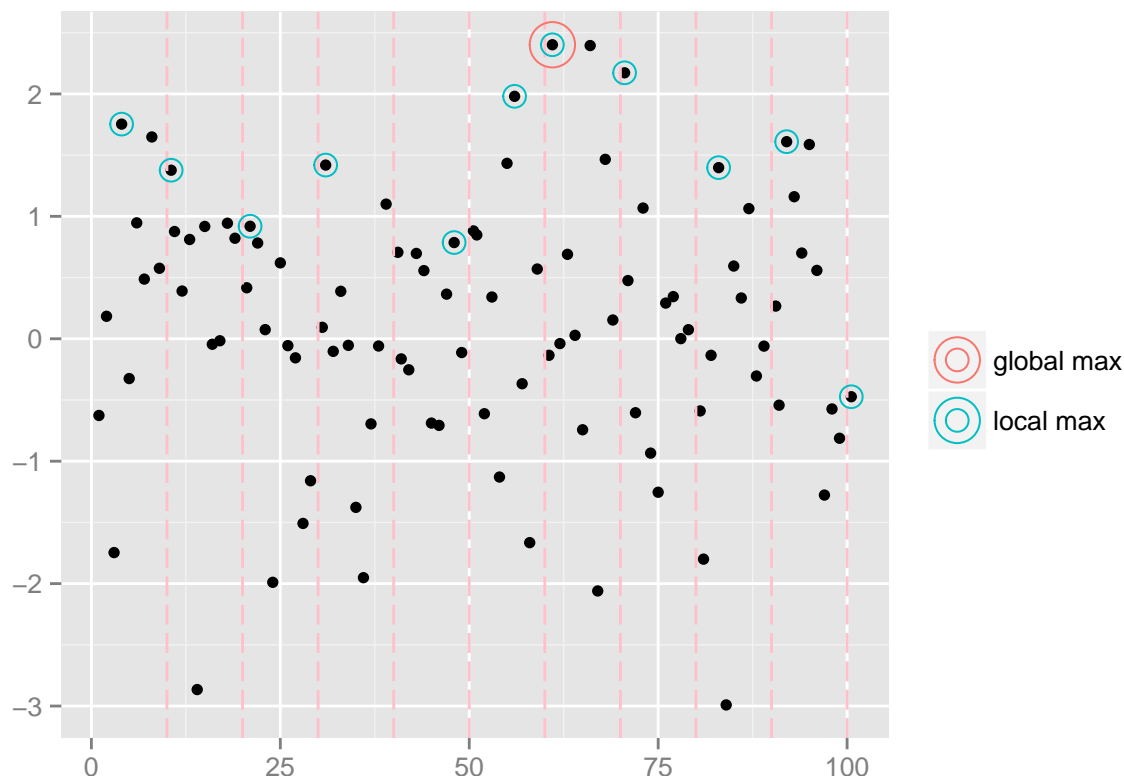
Cloudera has contributed distributed implementations of these tests to MLlib. You can now find in the Statistics object: (TODO: update this with a final list of functionality)

The remainder of this blog focuses on explaining the implementation details, along with some use-case examples for 2 of these tests: the 2-sample Kolmogorov-Smirnov test and the 1-sample Anderson-Darling test. We've chosen these two as they arguably provide the most interesting implementations.

Locals and Globals

The general intuition behind the strategy applied to solve these problems is: we can often arrive at a global value by working with local values. For example, consider the problem of finding the global maximum in the graph below. We could keep track of the last seen maximum as we traverse the graph from left to right. But,

if the graph were divided into parts, we could just as easily take the maximum of each part (local maximum, and potentially the global maximum) and then compare at the end and pick the winner.

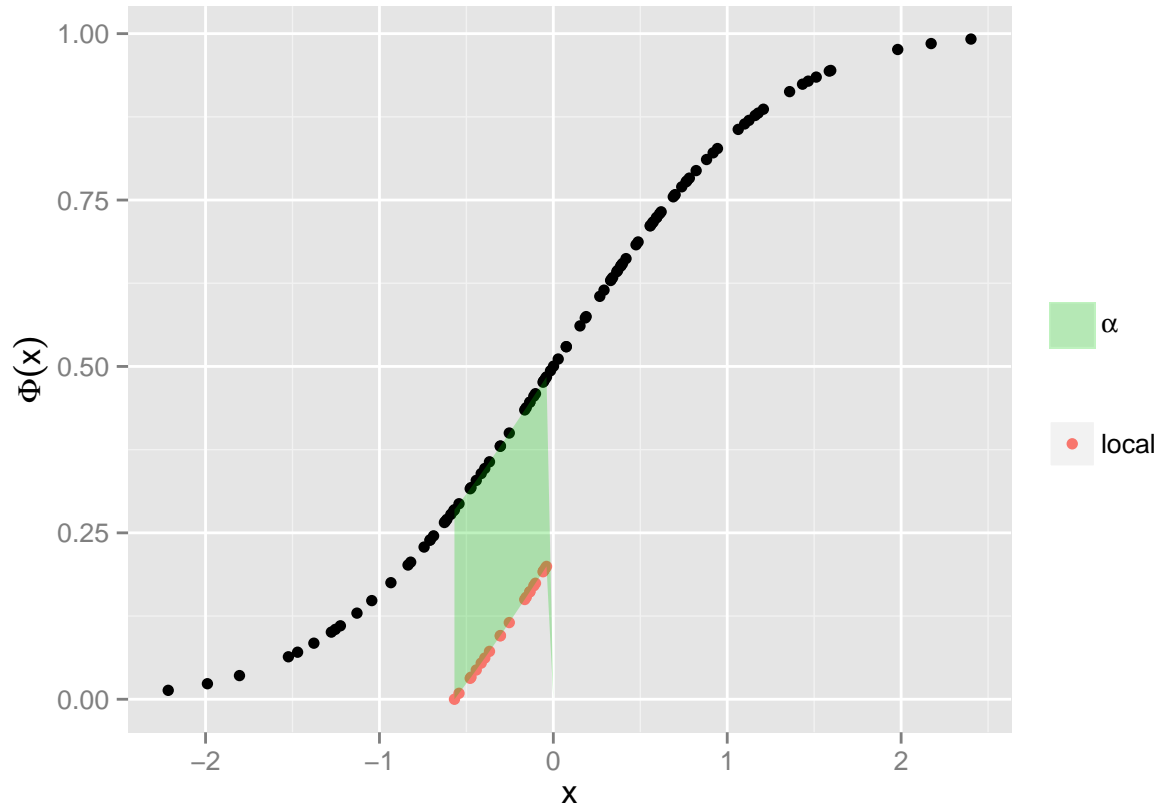


2-sample Kolmogorov-Smirnov test

The 2-sample variant of the Kolmogorov-Smirnov(KS) test allows us to test the hypothesis that both samples stem from the same distribution. Its definition is a nice, straightforward extension of the 1-sample variant: we are looking for the largest distance between the 2 empirical CDF curves. The process for generating the 2 curves is quite simple. You create a co-sorted group Z , and then for each element Z_i you calculate 2 values, y_{i1} and y_{i2} , which are simply the number of elements in each respective sample with a value less than or equal to Z_i , divided by the number of elements in Z . This in effect is the CDF curve for Z assuming we view sample 1 and sample 2 as some sort of *step-wise theoretical CDFs*. We can then subtract these 2 values to compare the “distance” between the 2 CDFs.

Implementing this for a single machine is very easy. The paragraph above effectively gives you the steps. But now let’s think about calculating this in a distributed fashion. In general, when conceiving distributed algorithms, the goal is to minimize the required communication between nodes. In the context of Spark this usually means minimizing the number of operations that induce shuffles. In this case, the algorithm must require at least a single shuffle - there is no getting around the fact that we must union and sort the data, but need we do more?

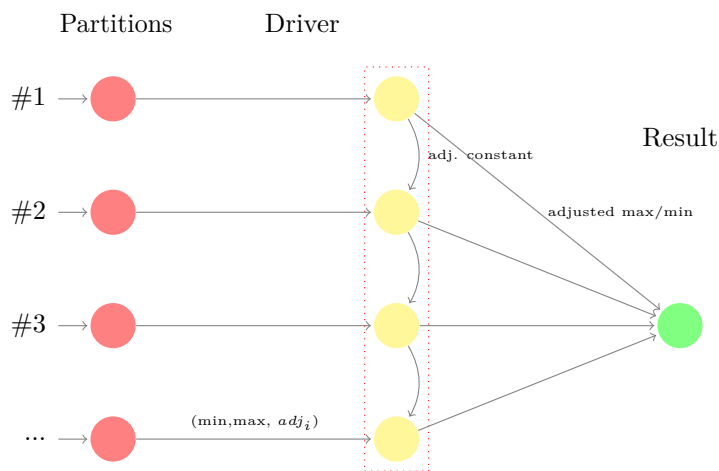
If we compute the empirical CDFs under each sample within a partition, the values for sample 1 will be off by α and the values for sample 2 will be off by β . However, note that all values for each sample within that partition will be off by that same amount. Since both are off by constants, the vertical distance between them is also off by a constant ($\alpha - \beta$). The graph below shows this idea for the case of 1 sample. The 2-sample case is analogous.



This fact allows us to simply concentrate on finding the largest deviation within each partition, adjusting these later by the appropriate constant (α is really α_i for each partition i , similarly for β) once we've collected the values on the driver, and then comparing the results for the maximum.

So in the case of the 2 sample KS test we want each partition to compute - The maximum and minimum distance between the local empirical CDFs - The count of elements in sample 1 - The count of elements in sample 2

We actually combine the last 2 points into 1 adjustment constant, to minimize the number of elements we need to keep track of. The diagram below visualizes the steps described prior.



Running our test's implementation on simulated OLS residuals In this case, we've decided to simulate data, from which we then create 2 very simplistic linear regression models using ordinary least squares (OLS).

If you recall from your statistics classes, OLS imposes various assumptions on the data that you're trying to model and on the resulting model. To the extent that the assumptions don't hold, you can end up with poor results, and in some cases you might not even know they're necessarily poor. One of those restrictions is that the residuals (which are defined as the observed value minus the predicted value) should be *white noise*, which means they follow a normal distribution, are uncorrelated, and have constant variance. Under some circumstances, you can keep your OLS model, even if not all boxes are checked off, but in this case we'll assume we're being sticklers.

We'll begin by generating our data: 2 regressors, which are combined to create a dependent variable. We can very easily generate a distributed sample from a normal distribution by using Spark's `RandomRDD`, which is part of `MLlib`.

```
val n = 1e6.toLong
val x1 = normalRDD(sc, n, seed = 10L)
val x2 = normalRDD(sc, n, seed = 5L)
val X = x1.zip(x2)
val y = X.map(x => 2.5 + 3.4 * x._1 + 1.5 * x._2)
val data = y.zip(X).map { case (yi, (xi1, xi2)) => (yi, xi1, xi2) }
```

Once we've defined our data, we specify 2 models. One which is clearly wrong. In turn, this means that the residuals for the incorrect model should not satisfy the white noise assumption.

```
// This model specification captures all the regressors that underly
// the true process
def goodSpec(obs: (Double, Double, Double)): LabeledPoint = {
  val X = Array(1.0, obs._2, obs._3)
  val y = obs._1
  LabeledPoint(y, Vectors.dense(X)) //
}

// this is a bad model
def badSpec(obs: (Double, Double, Double)): LabeledPoint = {
  val X = Array(1.0, obs._2, obs._2 * obs._1)
  val y = obs._1
  LabeledPoint(y, Vectors.dense(X))
}
```

We specified the models as simple functions, since we can then map over the observations RDD and obtain an `RDD[LabeledPoint]`, which is exactly what `MLlib`'s [linear regression implementation](#) expects.

```
val iters = 100
val goodModelData = data.map(goodSpec)
val goodModel = LinearRegressionWithSGD.train(goodModelData, iters)

val badModelData = data.map(badSpec)
val badModel = LinearRegressionWithSGD.train(badModelData, iters)
```

Now all we need are our residuals. We can obtain these easily using the models' `predict` function.

```
val goodResiduals = goodModelData.map(p => p.label - goodModel.predict(p.features))
val badResiduals = badModelData.map(p => p.label - badModel.predict(p.features))
```

In a small-data environment, one of the common ways to start testing the white-noise assumption is to perform various plots of the residuals and see if there are any glaring departures from normality. However, in

a big-data world, such tools might not be feasible or necessarily all that useful. We can, however, leverage other measures, such as the 2-sample Kolmogorov-Smirnov statistic. We could have naturally used the 1-sample Kolmogorov-Smirnov test, however, this is meant to provide a more general example of how we might use the 2-sample Kolmogorov-Smirnov test under circumstances where we might not have access to the theoretical cdf, but we do have access to a sample draw from the distribution. The function below performs this test for us.

```
def testResiduals(residuals: RDD[Double]): KolmogorovSmirnovTestResult = {
  val stdNormSample = normalRDD(sc, residuals.count())
  val mean = residuals.mean()
  val variance = residuals.map(x => (x - mean) * (x - mean)).mean()
  val standardizedResiduals = residuals.map(x => (x - mean) / math.sqrt(variance))
  // TODO: uncomment below once 2-sample included in MLlib
  //Statistics.kolmogorovSmirnovTest2(residuals, stdNormSample)
  Statistics.kolmogorovSmirnovTest(standardizedResiduals, "norm")
}
```

Once we check the test results, we can see that the badly specified model has residuals that do not satisfy the white noise assumption.

```
println(testResiduals(goodResiduals))
println(testResiduals(badResiduals))
```

Note that the test is meant for continuous distributions, so any ties in ranking will affect the test's power.

1-sample Anderson-Darling test

The implementation of the 1-sample Anderson-Darling test provides yet another productive strategy for implementing algorithms in a distributed setting: algebraic manipulations that isolate global vs. local components can be useful in maximizing how much we can calculate locally and how much information we need to carry to adjust globally later. The Anderson-Darling statistic is frequently formulated as shown below:

$$A = -N - \frac{1}{N} \sum_{i=1}^n (2i + 1) [\ln(\Phi(Z_i)) + \ln(1 - \Phi(Z_{n+1-i}))]$$

By having a term that involved Z_i and Z_{n+1-i} , this requires that we have access to elements that may or may not be in the same partition. But we can clearly reformulate this in a way that every term in the statistic solely deals with Z_i , thus getting around this issue. We can re-index the second terms so that the sum is based on element i , rather than $n + 1 - i$. The final form of the statistic is then:

$$A = -N - \frac{1}{N} \sum_{i=1}^n (2i + 1) \ln(\Phi(Z_i)) + (2(n - i) + 1) \ln(1 - \Phi(Z_i))$$

For our example, we'll be following inspiration from [this paper](#) which studied airplane departure delay distributions in Indonesia. The researchers studied how to fit various types of distributions to both the length of the delay and the time of day during which the delay occurred. We'll be focusing on the former of these.

We'll be using the 2008 portion of the [Airline on-time performance](#) data made available by the [American Statistical Association](#) back in 2009 for a data expo. There are data for every year between 1987 and 2008, but for simplicity we focus on 2008.

We take advantage of Scala's case classes to store each observation in a more easily readable format upon parsing, given that we can use named accessors rather than have to rely on positional accessors in a normal tuple.

```
case class AirlineObs(  
  timestamp: DateTime, // year(1) + month(2) + day of month(3) + scheduled departure time ((6)  
  carrier: String, // UniqueCarrier(9)  
  tailNum: String, // plane tail number, unique identifier for plane (11)  
  departureDelay: Int, // delays in minutes (16)  
  origin: String, // airport code for origin (17)  
  cancelled: Boolean // was flight canceled (22)  
)
```

Spark makes it very easy to read text files, and directories. In this case, we've unzipped the original download and now have a simple csv file. We simply read this file in as text and parse using a simple function. We remove all failures as they are a result of NA departure delay information.

```
// Source: http://stat-computing.org/dataexpo/2009/the-data.html  
val path = "/Users/josecambroner/Projects/gof/data/2008.csv"  
val airLineRaw = sc.textFile(path)  
// wrap parser in Try to catch any exceptions, then get successes and extract  
val parsed = airLineRaw.map(x => Try(parseAirlineObs(x))).filter(_.isSuccess).map(_.get)
```

It stands to reason that not all airports will have the same distribution of delays, so we focus our efforts on a single airport: SFO. Additionally, we exclude weekends and Federal holidays, which might unduly influence our numbers. Finally, given the research results in the paper mentioned earlier, we also acknowledge that the distribution of delays is likely to change throughout the day (not only in terms of occurrence but also in terms of length). Given this, we only consider delays that take place between 9am and noon, daily.

```
val targetAirport = "SFO"  
val sfoDelayData = parsed.filter { obs =>  
  val isSFO = obs.origin == targetAirport  
  val isWorkWeek = isWeekDay(obs.timestamp)  
  val isMorning = inTimeSlot(obs.timestamp, new LocalTime(9, 0, 0), new LocalTime(12, 0, 0), 1)  
  val isDelayed = obs.departureDelay > 0  
  val isNotFederalHoliday = !isFederalHoliday(obs.timestamp)  
  isSFO && isWorkWeek && isMorning && isNotFederalHoliday && isDelayed  
}
```

We will be analyzing each carrier in isolation, as it stands to reason that not all carriers will have the same distribution of delays. Furthermore, the insights from modeling a delay distribution are likely to be most beneficial when we concentrate on a single carrier, since that allows us to provide actionable business insight such as using delay distributions to improve scheduling, as the research paper suggests.

In this case, we focus on the top 5 carriers with delays in 2008. Note that this is not normalized, and thus is both a function of a carrier's proclivity for delays but also of a carrier's flight volume. Indeed, most of the names in the top 5 are carriers one would expect to see there solely due to their size: United Airlines, Skywest Airlines, Southwest Airlines, American Airlines, Continental Airlines.

```
val topCarriers = sfoDelayData.groupBy(_.carrier).map(x => (x._2.size, x._1)).top(5).map(_.._2)  
val dataForTop = sfoDelayData.filter(x => topCarriers.contains(x.carrier))
```

The analysis will require that we frequently query `dataForTop` for each of the carriers. So it makes sense for us to cache that data set, and avoid having it recomputed each time we use it.

```
dataForTop.cache()
```

For each carrier, we extract all relevant delays. Note that given the precision that was reported for delay length (minutes) there will inevitably be ties. However, Anderson-Darling (and Kolmogorov-Smirnov) are both meant for continuous data and thus ties can result in poor results. Given this, we add a bit of noise to our numbers to avoid ties (ala R's `jitter`). The noise is uniformly distributed between 0 and 1 and we take 1/100 for each observation. We then produce MLE estimate for the scale and shaper paramaters for a Log-Normal distribution. Finally, we test our data against this distribution by using the Anderson-Darling 1-sample, 2-sided test.

```
def testLogNormDist(
  data: RDD[AirlineObs],
  carrier: String,
  rand: RandomGenerator): KolmogorovSmirnovTestResult = {
  val delays = data.filter(_.carrier == carrier).map(_.departureDelay.toDouble)
  val hasTies = delays.distinct().count < delays.count
  val jittered = if (hasTies) delays.map(x => x + rand.nextDouble / 100) else delays
  val jitteredLog = jittered.map(math.log)
  // MLE estimates
  val scale = jitteredLog.mean()
  val shape = math.sqrt(jitteredLog.map(x => math.pow(x - scale, 2)).mean())
  val logNormDist = new LogNormalDistribution(scale, shape)
  // TODO: replace this with AD eventually
  val testResult = Statistics.kolmogorovSmirnovTest(
    jittered,
    (x: Double) => logNormDist.cumulativeProbability(x)
  )
  testResult
}
```

We note that the results tell us that the samples do not come from the distributions specified. This result can come from various sources:

- The delay length provided was discretized and thus we might have lost information. Our “jitter” trick is really a poor man’s solution to avoid ties. The real way would be to have obtained the original measures with increased precision.
- The log-normal parameters estimated with MLE are known to be biased by outliers, so it is possible that our current estimation method is not the best.
- The results reported by the research paper were based on 10-minute binned data, so it is possible that the Log-Normal is not an appropriate model in the absence of binning.

Although the test’s rejection can be initially disappointing for a data scientist (because, let’s be honest... we all want our first model to be the “right” model), the test has served its purpose and has made us rethink our current modeling approach and ask some important questions in the process.

To see the full code for both examples, please visit [TODO:ADD LINK](#).

While in this blog post we have used the Scala API, all these tests have also been made available in PySpark, allowing data scientists to calculate familiar statistics on big-data sized samples. As usual, all feedback is welcome. Happy hypothesis-testing!