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.

Testing in big-data world

The world of small-data analytics has many tools to accomplish this. Tools can range 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 aide 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 the libraries such as SciPy.

However, these tools are rarely able to natively handle the volume of data associated with big-data problems. And thus 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 throwing out information (which was the whole point of collecting that much data to begin with!). The

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 draw values from a discrete distribution and wanted to test these against expected values, we could then use the `ChiSqrTest(Vector, Vector)` implementation.

```scala
import org.apache.spark.SparkContext
import org.apache.spark.mllib.stat.Statistics
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.random.RandomRDDs.uniformRDD
import scala.math.random

val sc = new SparkContext("local", "test")
val trialsMult = 1e487
// We have a bernoulli RV with p = 0.3
// each experiment is of size trialsMult * x
val p = 0.3
val draw = (x: Int) => {
  Array.fill(trialsMult.toInt * x)(if (random <= p) 1.0 else 0.0).sum
}
val domain = (1 to 100).toArray
val observed = sc.parallelize(domain).map(draw).collect()
val expected = domain.map(x => trialsMult * x * p)
val test = Statistics.chiSqTest(Vectors.dense(observed), Vectors.dense(expected))
test.pValue > 0.05
```

In this case, we were able to use the Chi-Square test to show that the divergence between our observed and expected counts is not sufficient to conclude that the process underlying our experiments is not actually Binomial with success probability 30%.

As useful and powerful as this 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. Cloudera has most recently contributed various tests (TODO: label exactly which) which fulfill this need and allow users of the platform to use standard goodness-of-fit tests across their applications.

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

Jose Question: Should we eliminate this section for brevity? I assume most readers would know this..It might even come across as condescending to include. Thoughts? The only argument I currently have for keeping it is the section title, that I thought was funny

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. Or we might have measured the wait time for the 6 train in NYC at the City Hall stop (protip: it's forever…just come to terms with that) and we'd like to model this using an exponential distribution. 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

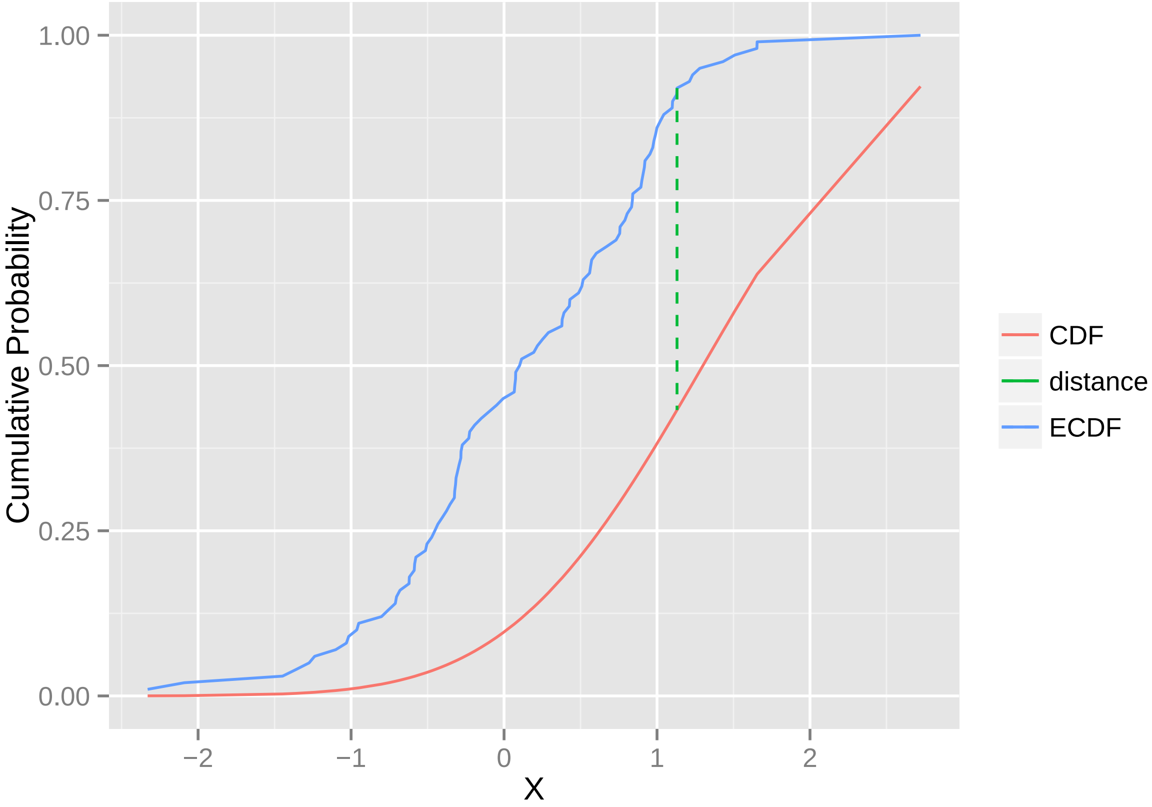
This might be one of the most popular goodness-of-fit tests for continuous data out there, with an implementation in pretty much every popular analytics platform. The intuition behind the test centers around comparing the largest deviation between the cumulative distribution at a given value X under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

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

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

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

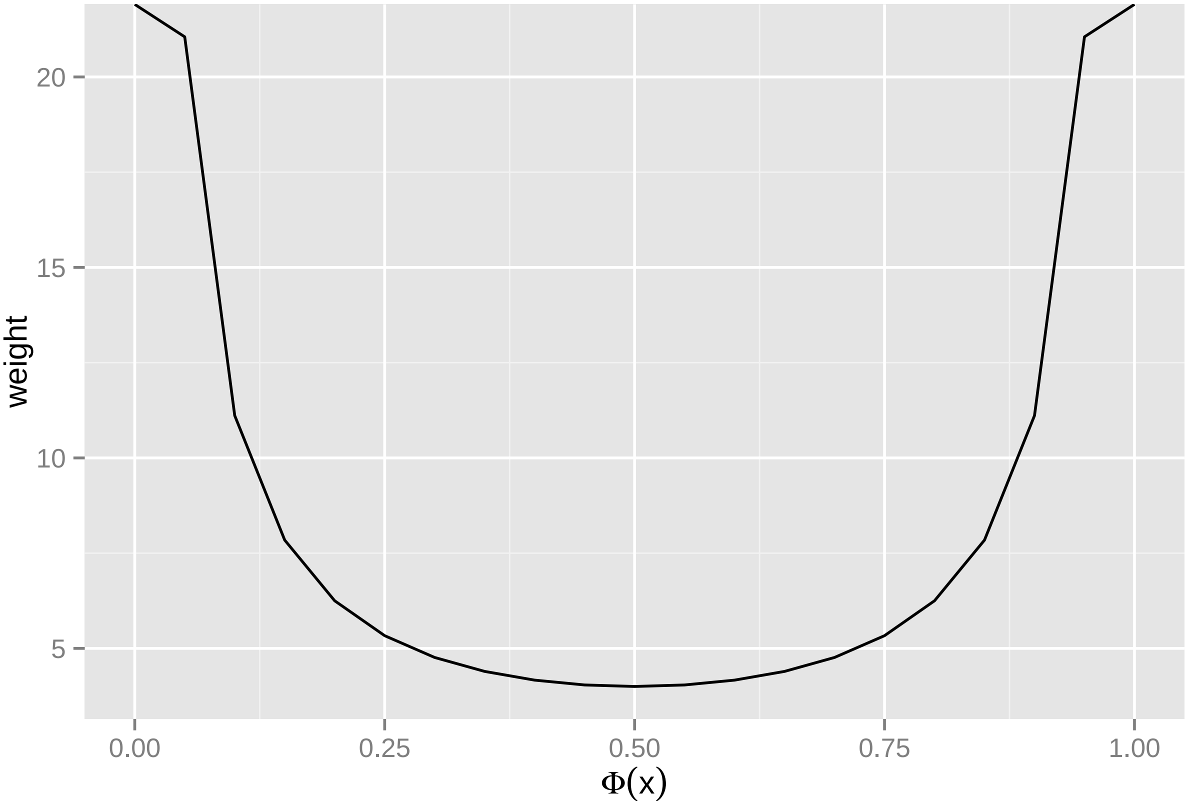
Anderson-Darling is often proposed as an alternative to the Kolmogorov-Smirnov statistic, with the advantage that it is better suited to identify departures from the theoretical distribution at the tails, and is more robust towards the nuances associated with estimating the distribution's parameters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

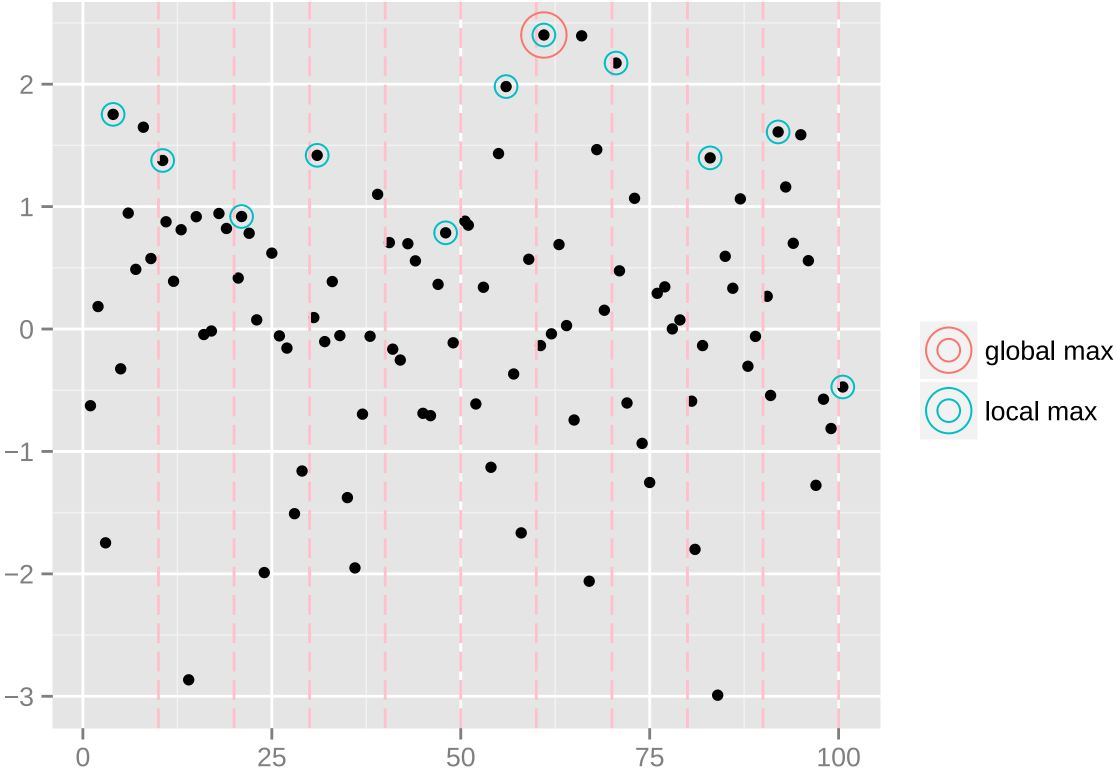
Calculating both of these statistics is extremely straightforward when performed in memory. If we have all our data, we can simply traverse it as needed and calculate what we need at each point. Performing the same calculations in a distributed setting can take a bit more craftiness.

Cloudera has contributed distributed implementations of these tests to MLlib. You can now find in the Statistics object: the 1-sample, 2-sided Kolmogorov-Smirnov test; the 2-sample, 2-sided, Kolmogorov-Smirnov test; the 1-sample Anderson-Darling test; and the k-sample Anderson-Darling test. (TODO: update this with a final list of functionality, some of these might not make it in)

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 simply take the maximum of each part (local maximum, and potentially the global maximum) and then compare at the end and pick the winner.

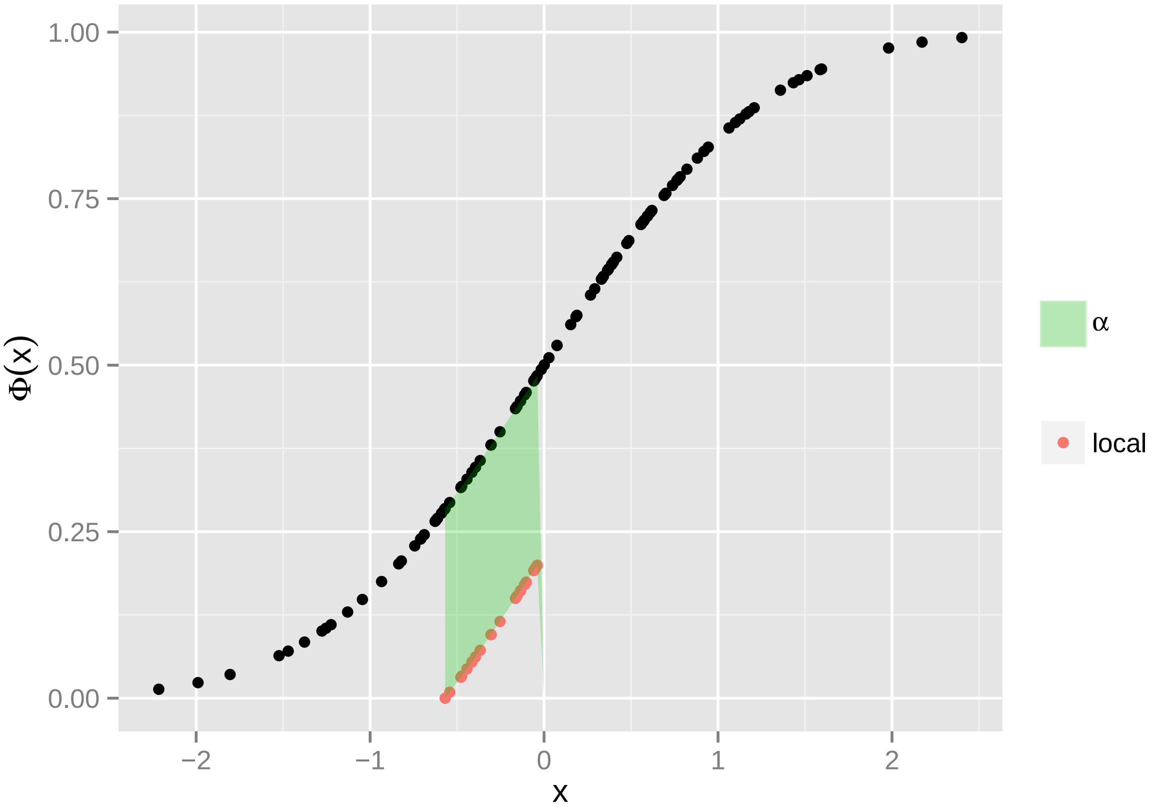


2-sample Kolmogorov-Smirnov test

The 2-sample variant of the 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*. It makes sense then, that we could in turn compare these curves to test whether the 2 CDFs (and thus the distributions) are the same.

Implementing this in-memory is very easy. The paragraph above effectively gives you the steps. But now let's think about calculating this in a distributed fashion. We are going to have to combine the data and sort it to create Z. There is no way for us to get around that. Once we have that, however, we'd like to minimize the amount of unnecessary shuffles (recall, this requires writing out data to disk, which is expensive) and jobs (which isn't quite a shuffle, but still consumes resources).

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.

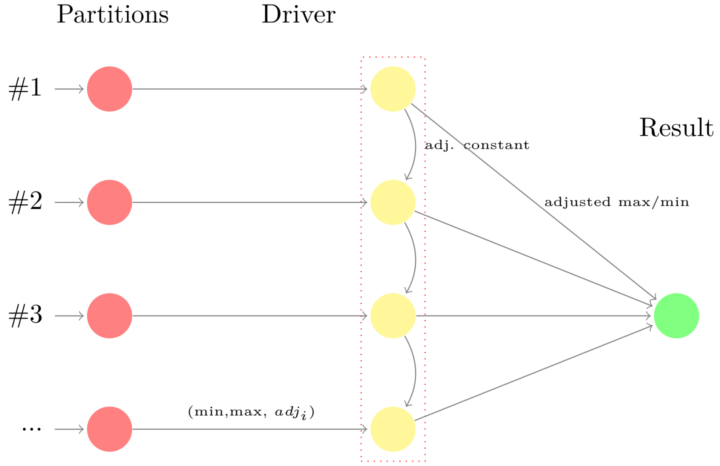


This fact allows us to simply concentrate on finding the largest deviation within each partition, adjusting these later by the appropriate constant (α is really α_i for each partition i, similarly for β), and then comparing the results for the maximum.

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.



Example 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-noie 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: algebraic transformations 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 test how well an exponential distribution fits the waiting time between arrivals of the NYC 1 train (downtown bound) at different stops. Such a test might be a natural first line of attack for someone looking to model delays in the subway. For example, if the tests tell us that an exponential distribution cannot be ruled out, then we might decide to use the exponential's CDF to identify unusually long times, and warn users about potential issues on the track.

We'll be using the May 2011 MTA ATS data. The files are available as csv downloads, with each row providing: the date, train identifier, direction of the train, timestamp of the event, the type of event (arrival or departure), route identifier, stop identifier, and track identifier.

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

```
case class TrainEvent(
  timestamp: DateTime,
  train: Int,
  dir: Int,
  eventType: Int,
  route: String,
  stop: Int,
  track: String)
```

Spark makes it very easy to read a directory of files (which is what we get from MTA) into an RDD. Once we have that, it is just a matter of parsing each line into a `TrainEvent` instance.

```
  val path = "/Users/josecambronero/Projects/gof/data/mta_may_2011"
  val mtaRaw = sc.textFile(path)
```

As usual with real-world data, there are likely to be various parsing errors. Some lines might not fit them schema. Some numbers might be smudged with letters. The possibilities are endless. Since we don't quite care to correct these mistakes, given the volume of data we might have, we have decided to simply throw away data that fails to parse. This can be done very easily with Scala's `Try` (I swear this is the only time you'll hear me say this) monad. The general idea behind `Try` is that we can return a value regardless of whether or not there was an exception by wrapping the value (or exception) in a container that indicates what's inside. This allows us to chain operations cleanly. In this case, there are 2 container: `Success` and `Failure`. The latter holds an exception, and the former a value. In this case, we simply filter out `Failure` and extract the value of the `Success` to obtain an `RDD[TrainEvent]`.

```
val parsed = mtaRaw.map(x => Try(parseTrainEvent(x))).filter(_.isSuccess).map(_.get)
```

Users of MTA's system know that train arrival frequency is not constant throughout the day (scheduled arrivals are less frequent at *odd* hours, think 3am on a Monday). So in order for our analysis to be valid, we need to restrict ourselves to a particular time of day. In reality, picking the span of time should be done more methodically, observing the MTA's schedule. However, we'll just pick a reasonable timeframe. We'll focus on the 5pm to 8pm time slot, since that is rush hour, and when delays are most troublesome for both users and service providers.

Similarly, we need to distinguish between weekdays and weekends. Train service is frequently disrupted over the weekend for repairs etc. Holidays can also be another source of trouble. The data used is all for May 2011, so we'll only worry about filtering out 1 federal holiday: Memorial Day (May 30th, 2011).

```
    val rushHourData = parsed.filter { event =>
    val dateTime = event.timestamp
    val weekDay = isWeekDay(dateTime)
    val minTol = 1
    val rushHour = inTimeSlot(
      dateTime,
      new LocalTime(5, 0, 0),
      new LocalTime(8, 0, 0),
      minTol
    )
    // Memorial day
    val holiday = new DateTime(2011, 5, 30, 0, 0, 0)
    val notHoliday = !dateTime.withTimeAtStartOfDay.isEqual(holiday)
    weekDay && rushHour && notHoliday
  }
```

Now that we have our `RDD[TrainEvent]` we will want to focus on just 1 train line for this exercise. We'll pick the downtown (trains in Manhattan have 2 directions: up/down town) 1 train. Why? Subway trains aren't necesssarily "regular" in terms of their schedule. Sometimes some trains run *express* and skip certain stops, even if they're usually *local*, meaning they make every stop. The 1 train is one of the more regular lines, with no scheduled express times.

Downtown direction is represented as `1` in the MTA files. The same is the case for arrivals, which are encoded in the `eventType` field. The fact that we've filtered for 1 train, downtown, should be sufficient. However, it seems that when you do so some track ids remain which are incongruent with that, or have an id that is not explicitly explained in the ATS data specfication. We've decided to be a bit conservative and only take those observations whose track id ends in `'1'`, which ATS states is generally the local southbound track.

We also want to make sure we have as many obervations from the same day close together, since timespans need to be calculated intra-day. To achieve this, we sort the data by timestamp.

```
  val train1Down = rushHourData.filter (x =>
    x.train == 1 && x.dir == 1 && x.eventType == 1 && x.track.last == '1'
  ).sortBy(x => x.timestamp.getMillis)
```

The analysis will require that we frequently query `train1Down` for various stops, as we can only compare arrival times on a per-stop basis. So it makes sense for us to cache that data set, and avoid having it recomputed each time we use it.

```
train1Down.cache()
```

Now, for each stop we perform the same steps, so we can take the function described below, on a step-per-step basis, and simply map over all the stops.

First, we want to filter out all data not pertaining to the stop we're currently focusing on.

```
val stopData = data.filter(_.stop == stop).map(_.timestamp)
```

Now we want to compare arrival times within that stop, and we can solely compare times intra-day. The times are spread across partitions, and thus it is certainly possible that the first observation in partition i is part of the last date seen in partition $i - 1$. However, rather than collect the last date in each partition and share with the next, the simplest approach to take is to simply treat each partition separately, and deal with the fact that there will be some observations we need to throw away as we won't be able to calculate a time span. Namely, the first observation of each date in each partition will not have a timespan from the previous arrival. This kind of decision needs to be made on a case-specific basis, however, in this case, we don't lose enough information to justify increased computational complexity.

```
    val waitingTimes = stopData.mapPartitions { part =>
      val local = part.toArray
      // comparisons should only take place between arrival times in the same day
      local.groupBy(_.withTimeAtStartOfDay).flatMap { case (date, intraDayTimes) =>
        // sort times within a day, Scala's groupBy makes no guarantees on ordering
        val sortedMillis = intraDayTimes.map(_.getMillis).sortBy(x => x)
        sortedMillis.zip(sortedMillis.drop(1)).map { case (pt, t) =>
          (t - pt).toDouble / 6e4
        }
      }.filter(_ > 0).iterator
    }
```

In the prior code snippet, you can see that in a given partition, we group arrivals by date. Within each group we have to sort the timestamps again, since Scala's `groupBy` makes no guarantees regarding order. Once we've done that, we can simply pair up each timestamp with the following and calculate the difference.

There might be some supurious timestamps in there as well, so for good measure we kick out anything with a negative or zero time span (unfortunately MTA has not yet developed time-traveling technologies, so we know these to be data errors).

Now that we have an `RDD[Double]` representing the minutes between arrivals for this stop (i.e. the waiting times between trains), we can test the distribution of values against an Exponential distribution.

We calculate the mean for the distribution by using the mean of the data. Finally, given this, we can apply MLlib's `AndersonDarlingTest(RDD[Double], String, Double*)`, specifying that this is an exponential distribution and passing in the mean of the distribution.

```
   val mean = waitingTimes.mean()
   val result = Statistics.andersonDarlingTest(waitingTimes, "exp", mean)
   (stop, result)
```

Anderson-Darling is robust to estimating distribution parameters from the data. This is not the case with other tests necessarily. So in this case, Anderson-Darling is great option to use.

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!