SwiftVis2: Plotting with Spark using Scala

Mark C. Lewis¹, Lisa L. Lacher²

¹Department of Computer Science, Trinity University, San Antonio, TX, USA
²College of Science and Engineering, University of Houston - Clear Lake, Houston, TX, USA

Abstract—This paper explores the development of a plotting package for Scala called SwiftVis2 and its integration with Spark. While there are many design goals for SwiftVis2, the goal we focus on in this paper is to enable the early stages of data exploration to be easily done with Scala. We compare SwiftVis2 to other plotting libraries for Scala and present a general overview of the library. We then look at the facade and fluent interface that have been created to make the library more usable. We also discuss the Spark integration and how features particular to the Scala language enable plotting with Spark in a way that is natural and expressive. We finish by comparing the current speed and output quality to a few other libraries.

Keywords: plotting, big data, Apache Spark

1. Introduction

Apache Spark has become one of the primary frameworks used for big data analytics. While Spark has bindings for Java, Python, and R, the language it is implemented in, and the one in which all features are always available, is Scala. Despite this, much of the data science (exploratory and model fitting) done with Spark is done using Python and R. It is only once an application has passed gotten to data engineering (implementing the large data applications) that Scala and Java become the primary languages. There are a number of reasons for this. The one that is most frequently cited is the dynamic nature of Python and R and the feeling that they make the data science phase easier. While this definitely puts Python and R ahead of Java, the Scala syntax is as expressive as those scripting languages, and it has a scripting environment and a REPL. Scala’s use of type inference also makes Scala programs feel like those written in dynamically typed languages without giving up the safety and performance aspects of static typing.

The other significant factor is the quality and availability of plotting. The early stages of a data science project involve a lot of data exploration. While summary statistics can reveal certain things, being able to visualize data is an essential part of the process. Data plotting is an area that hasn’t been as well addressed on the Java Virtual Machine (JVM) as on platforms for Python and R. This is largely because Java is largely used as a server-side language where plotting isn’t as necessary, so libraries like JFreeChart[1] have been sufficient for most applications. Unfortunately, JFreeChart has the verbosity that is common to many Java libraries and it lacks some of the visualization options that are helpful for statistical or scientific plotting.

The rise of Spark has led to the development of a number of libraries to enable better plotting in Scala. The subject of this paper is a library called SwiftVis2[2] that is aimed primarily at improving scientific plotting on the JVM with an eye to performance and the creation of plots with large amounts of data. Development also includes Spark integration to simplify the creation of plots when using Spark. The primary motivation is to enable the full data processing pipeline, from data cleaning and exploration to final application development, to happen efficiently in a single language to avoid the costs that are unavoidable when switching languages. Allowing data exploration in Scala also insures that it can be done using the full feature set of Spark, avoiding the delays bringing features to the other language bindings.

1.1 Apache Spark

Apache Spark has become one of the leading frameworks for big data analysis. The Apache Spark developers bill it as “a fast and general engine for large-scale data processing” [3]. It was created by Matei Zaharia at UC Berkeley in 2009. It became open source in 2010 and was donated to the Apache Software Foundation in 2013.

In 2014, Spark was used to set a new record in large-scale sorting and achieved a 100TB sort in twenty-three minutes in the Daytona GraySort contest. In that same contest, Hadoop MapReduce, using significantly more machines, took seventy-two minutes to sort 100TB of data [4]. This feat helped Spark begin to dominate Hadoop MapReduce.

Spark can perform batch processing, but it excels at interactive queries, machine-based learning, and streaming workloads. Spark is most noted for its real-time data processing capability. It runs one hundred times faster than Hadoop MapReduce in memory and ten times faster than MapReduce’s disk-bound, batch processing engine. Some of the strengths of Spark includes its rich API, and that it is compatible with Hadoop and its modules.

1.2 Existing Plotting Options for Scala

Data visualization is an integral part of data science. Plotting is often the quickest and most convenient way of analyzing data. There are many popular libraries for plotting, such as JFreeChart for Java, matplotlib for Python, and ggplot2 for R. However, there isn’t currently a de facto option
for Scala. Notebooks have built-in visualization tools for the different languages they support. However, notebooks offer environments better suited for quick exploration tasks and collaboration with a small number of plotted elements. There are many limitations in the current browser based notebook implementations. These limitations prevent them from offering enjoyable environments to develop code for production data engineering work. Thus, we will focus our search on visualization libraries that can be used in a general programming environments outside of a browser.

One of the libraries written specifically for Scala related to machine learning is ScalaNLP, which has three sub-libraries for various aspects of NLP. The component that contains the mathematical components is called Breeze which a JFreeChart wrapper called Breeze-viz[5]. While JFreeChart does not have as many features as matplotlib, it does contain a wide range of chart types and support for many output types such as image files, vector graphics file formats, and Swing components. Breeze-viz helps to tame the verbosity of using JFreeChat in Java, but it is unfortunately rather poorly documented and doesn’t add any additional functionality.

A second alternative, scala-chart [6], is also a Scala wrapper for JFreeChart. It provides a more convenient interface for working with JFreeChart than using the Java library directly, allowing one to accomplish everything that JFreeChart does with fewer lines of code. Like Breeze-viz, it does not provide any additional functionality or plot types beyond what is in JFreeChart.

One thing that is significant to note is that JFreeChart uses Swing, the primary graphics library for Java for most of the history of the language. Beginning with Java 7, JavaFX was introduced as a replacement for Swing. Unlike Swing and its predecessor AWT, JavaFX was built from the ground up. Certain aspects of the design of both AWT and Swing prevented the library from offloading a lot of work to the graphics card. JavaFX was built with this capability in mind, giving it better support for 3D graphics and potentially better overall performance. Oracle now encourages all graphical applications built since Java 8 to use JavaFX instead of Swing or AWT, and future improvements to graphics on the JVM will focus on JavaFX[7].

The third plotting package for Scala that plots to the desktop is the New Scala Plotting Library (NSPL)[8]. Unlike Breeze-viz and scala-chart, NSPL is a 2D plotting library written from scratch in Scala. It renders plots both to vector (pdf, svg, eps) and raster formats (png, jpg). It also includes support for Java Swing and the HTML5 Canvas. This broad support for formats allows it to run on the JVM or in the browser with minimal dependencies.

There are also a number of other plotting libraries with Scala bindings that are designed for plotting specifically to browsers without direct desktop plotting support. This generally means that the data being plotted needs to go though a JSON text intermediate format. That adds a lot of overhead to plots that include a large number of elements. As one of our objectives of SwiftVis2 is fast plotting of large amounts of data, these libraries aren’t true alternatives, so we only list them briefly.

- Plotly[10] – Commercial plotting library with Scala support.

Table 1 summarizes the plotting libraries mentioned here and includes information on their feature support. The column for Spark support indicates if the library has specific code for making it easier to plot data from Spark. Any plotting library can be used to plot data from Spark if the user performs manual collection or aggregation operations to bring the data into standard Scala, but we are interested in library code that goes beyond this.

### 1.3 SwiftVis and SwiftVis2

This work grows out of older work on a data analysis and plotting package called SwiftVis[13] that was initially created to help planetary scientists work with the output of the Swift[14] planetary integrator. SwiftVis is a tool focused on doing analysis with a data flow model that could be used by scientists who were not proficient programmers. SwiftVis2 has a goal of reproducing this capability with additional error handling and better support for parallelism[15], but such work is beyond the scope of this paper.

### 2. Plotting in SwiftVis2

The SwiftVis2 plotting features are based on a few key classes: Plot, PlotGrid, PlotStyle, and Axis. There are other elements, but these are the most significant ones, and hence they are the only ones we will consider here.

**Plot**

This is the top level construct that holds all the information that is to be rendered. It includes PlotGrids as well as text and other elements.

---

### Table 1

<table>
<thead>
<tr>
<th>Tool</th>
<th>Web</th>
<th>Desktop</th>
<th>Spark</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breeze-viz</td>
<td>x</td>
<td></td>
<td>x</td>
<td>free</td>
</tr>
<tr>
<td>NSPL</td>
<td></td>
<td></td>
<td>x</td>
<td>free</td>
</tr>
<tr>
<td>scala-chart</td>
<td>x</td>
<td></td>
<td>x</td>
<td>free</td>
</tr>
<tr>
<td>Bokah</td>
<td>x</td>
<td></td>
<td></td>
<td>free</td>
</tr>
<tr>
<td>Plotly</td>
<td>x</td>
<td></td>
<td>x</td>
<td>$500-$1000/user/year</td>
</tr>
<tr>
<td>Vegas</td>
<td>x</td>
<td></td>
<td>x</td>
<td>free</td>
</tr>
<tr>
<td>WISP</td>
<td>x</td>
<td></td>
<td></td>
<td>free</td>
</tr>
</tbody>
</table>

This table compares current options that exist for plotting with Scala. The columns indicate whether the library has web plotting support, desktop plotting support, special bindings for Spark, and cost.
PlotGrid
This represents a grid of rectangular plot regions where each row and each column can have two axes, one on the either side of the plot. In each cell of the grid there can be multiple PlotStyles.

PlotStyle
A style that represents the visual display of the data. Examples include scatter plots, histograms, and box plots.

Axis
SwiftVis2 currently has support for numeric and categorical axes. The numeric axes can use linear or logarithmic scales.

Each PlotStyle is added to a PlotGrid along with information saying what axes it will use for x and y. This allows styles to share axes or use separate ones for either x or y independently. Data can be overlaid by stacking multiple plot styles in a single grid cell.

All of these are written using a Scala construct called a case class, are immutable classes that have appropriate implementations of standard methods for the JVM including equals, hashCode, and toString. In addition, case class have code that allows them to be used with Scala’s pattern matching and to be instantiated without new. Most importantly for SwiftVis2, case classes have a copy method that allows for modified versions to be created easily by providing only the fields being altered using named arguments. This simplifies the creation of a fluent interface.

We use types called PlotSeries to represent data. There are currently three subtypes of PlotSeries named PlotDoubleSeries, PlotIntSeries, and PlotStringSeries. Each of these is a function from an Int to the type listed in the name, and the PlotSeries type adds methods for minimum and maximum indices to allow offsets or slices of data that are not zero-indexed.1

The following code demonstrates the low-level usage of SwiftVis2 to create and display a scatter plot with data coming from sequences named x and y. The plot it produces is shown in Fig. 1.

```scala
import swiftdsa.plotting._
import swiftdsa.plotting.renderer.FXRenderer
import swiftdsa.plotting.styles.ScatterStyle

val font = new Renderer.FontData("Arial", Renderer.FontStyle.Plain)
val style = ScatterStyle(x, y)
val p2d = Plot2D(style, "x", "y")
val xAxis = NumericAxis(tickLabelInfo = Some(Axis.LabelSettings(90, font, "%1.1f")), name = Some(Axis.NameSettings("X", font)))
val yAxis = NumericAxis(tickLabelInfo = Some(Axis.LabelSettings(0, font, "%1.1f")), name = Some(Axis.NameSettings("Y", font)))

val grid = PlotGrid(Seq(Seq(Seq(p2d))), Map("x" -> xAxis, "y" -> yAxis), Seq(1.0), Seq(1.0))
val plot = Plot(grid = Map("main" -> Plot.GridData(grid, Bounds(0.0, 0.05, 0.95, 0.95))), name = Some(Axis.NameSettings("X", font)))

FXRenderer(plot, 600, 600)
```

This example begins with a number of import statements that are generally placed at the top of the file, and which would not be duplicated for each plot. The second and third imports should look familiar to anyone with a Java background. The first one makes more sense once you know that _ character is used as a wildcard in Scala and replaces the Java use of *. What isn’t as clear though is that the primary purpose of the first import statement is to bring into scope a number of implicit conversions that convert from Scala collection types, like Seq[Double] and Array[Double], to the PlotDoubleSeries that is used to represent data in SwiftVis2. Without these implicit conversions, the line creating the ScatterPlot would be significantly more complex.

Currently we have implemented plotting styles for scatter plots, bar charts, histograms, box plots, and violin plots, and have plans to add a number of other options. It is worth mentioning that the scatter plot supports the ability to vary the size of the points independently in the X and Y directions as well as to set their colors, add error bars, and connect points with lines. As such, this one plot style serves the

---

1The PlotSeries is not defined as PlotSeries[A] that is a function from Int -> A as this would force Scala to box the Double and Int values instead of using primitives. This can have a significant impact on performance, and leaving it out doesn’t have negative impacts on the programming of the library or its usage.
role of what would often be multiple different types of plots such as bubble plots and line charts. In the example above, all those other features take default values, so the dots are a uniform size, drawn in black, and have no error bars or lines connecting them.

Even without those options, there is a lot of overhead in this example that is needed to give the full library power, but which is only boilerplate for general usage. To reduce this overhead in common cases, we have implemented a facade[16] for common uses.

### 2.1 Plotting Facade

The facade for SwiftVis2 is built as a number of methods in the Plot companion object. These methods act like static factory methods to create instances of Plot.

The facade includes methods built specifically for different types of plot styles. This included basic methods like scatterPlot, barPlot, and histogramPlot as well as methods that take advantage of the PlotGrid capabilities such as scatterPlotGrid. The following example shows the use of scatterPlotGrid. This example also includes a color gradient. Fig. 2 shows the plot that might result for some particular values of the x and y data.

```scala
val cg = ColorGradient(0.0 -> BlackARGB, 0.5 -> RedARGB, 1.0 -> WhiteARGB)
Plot.scatterPlotGrid(Seq(
  Seq((x1, y1, BlackARGB, 5), (x2, y2, BlueARGB, 5)),
  Seq((x3, y3, cg(c3), 10), (x4, y4, GreenARGB, 5)),
  "Plot_Grid", "Shared_X", "Shared_Y"
))
```

The primary limitations of this method are that all of the plots must be scatter plots, there can only be one scatter plot per cell, and all the plots share the same axes, which automatically scale to the minimum and maximum values for any of the data points.

There are also facade methods that can handle a combination of different types of plots. The following code shows the use of a facade method called row which will plot a single row in the grid with mixed plot styles. Figure 3 shows the resulting plot for a particular set of values for xs and ys.

```scala
Plot.row(Seq(
  ScatterStyle(xs, ys, symbolWidth = 5, symbolHeight = 5),
  HistogramStyle(bins, Seq(HistogramStyle.DataAndColor(counts, GreenARGB))), binsOnX = false),
  BoxPlotStyle(Array("Distrib"), Array(ys)),
  ViolinPlotStyle(Array("Distrib"), Array(ys)), "Distributions", "Num_X", "Categories", "Y")
```

The left-most plot in the row is a scatter plot with points distributed uniformly in x, and non-uniformly in y. The second plot is a histogram of that non-uniform distribution in y. The third and fourth plots are a box plot and a violin plot of the same distribution to illustrate some of the other plotting styles currently implemented in SwiftVis2.

### 2.2 Fluent Interface

The facade has been designed to include simple access to most of the methods that we expect user to want to plot on a regular basis. However, it is not only impossible to predict all potential use cases, it would be bad design to attempt to do so. For this reason, we also wish to provide methods that allow users to easily adjust the plots created by the facade without having to use the lower-level API to get that expressivity.

The approach we have chosen for this is to add a fluent interface[17] to the various components of SwiftVis2, starting with the top level Plot type. As was mentioned at the beginning of this section, the components in SwiftVis2 are built from immutable case classes. Each call in the fluent API returns a new instance Plot with the appropriate changes made. This allows calls to be chained together in a manner that is easy to read and work with, which is why such approaches are called fluent interfaces.

2The Scala language provides a construct of an object declaration that creates singleton objects. Instead of having static methods, object declarations that share the same name as a class and appear in the same file are called companion objects. Companion objects have access to private data of instances of the class in the same manner as static methods.

3This might sound like it uses a lot of memory, but the way that the copy method works on a case class, only the elements that are changed are altered. All other elements refer to the old values. One advantage of immutability is that this can be done safely without defensive copying.
3. Spark Integration

Unlike Hadoop's implementation of MapReduce, Spark's functional interface makes it feasible to use Spark for the entire data processing pipeline. Hadoop is most useful at the end for creating big data applications, but Spark's design makes it work well for the whole process from data cleaning, exploration, and model building on through data application creation. This makes plotting data used in Spark particularly advantageous. This is why projects like Vegas-viz[11] and SwiftVis2 aim to integrate Spark support directly in the project.

3.1 Spark SQL

To understand the Spark integration in SwiftVis2, it is helpful to give a brief introduction to the DataFrame functionality of Spark. Spark DataFrames are similar to those in R or the Pandas library for Python, only they do their processing in parallel, distributed across a cluster. DataFrames were introduced as part of what is called Spark SQL, and as the name implies, the DataFrames can be operated on using SQL or functional methods that are related to SQL operations.

A DataFrame can be pictured like a table in a relational database, and there is a type called Column that represents the columns on a DataFrame as well as many operations that can be done on the values in the columns. For example, if we have a DataFrame called people that has columns named “weight” and “height” in units of kilograms and meters, we could calculate the BMI of

```
Plot.stacked(Seq(  
  ScatterStyle(temp, alt, lines=ScatterStyle.  
    connectAll),  
  ScatterStyle(pressure, alt, lines=ScatterStyle.  
    connectAll, symbol = Rectangle)),  
  "Temp and Pressure", "Temperature [C]", "  
  Altitude [km]").  
withModifiedAxis[NumericAxis]("x", "pressure",  
  asMaxSideXAxis  
  .updatedScaleStyle(Axis.ScaleStyle.LogSparse)  
  .updatedName("Pressure [Pa]"))  
  .updatedStyleXAxis("pressure", stack = 1)
```

Here the `Plot.stacked` method from the facade is called first setting up a plot that has both temperature and pressure plotted in a single cell with the data points connected by lines. The result of that call would have temperature and pressure share the same axis, which makes the temperature values nearly impossible to see. The fluent interface of `Plot` is then used to create a new axis called “pressure” that is drawn at the top of the plot using a log scale. The result of rendering this plot is shown in figure 4.
everyone in the table with `people.select('weight / (’height * ’height))`. This type of value could be brought into a normal Scala array with a line like:

```scala
val bmi = df.select('weight / ('height * 'height)).as[Double].collect(), and the user could plot that, but this leads to very verbose code with declarations that collect each value we wish to use in the plot separately.
```

### 3.2 Integration with Implicits

This usage of `Columns` is the natural way to operate on `DataFrames` in Spark, and we wanted to preserve that in the SwiftVis2 integration. To do this, we need to make it so that it is easy to move from `Columns` to the `PlotSeries` that are used by SwiftVis2. That is accomplished by a Scala language feature called implicit conversions. An implicit conversion does much what the name implies, it causes one type to be converted to another implicitly if such a conversion allows the code to compile. Unlike C++, Scala is strict about when implicit conversions are applied, which helps make their usage safer. The following code shows one implicit conversion that can turn a Spark `Column` into a SwiftVis2 `PlotDoubleSeries`. The Swift binding contains similar conversions for other series types as well as for Scala symbols so that programmers can use the syntax that they are familiar with.

```scala
implicit class ColumnToDoubleSeries(col: Column){
  implicit ds: Dataset[_] extends PlotDoubleSeries {
    import ds.sparkSession.implicits._
    val data = ds.select(col).as[Double].collect()
    def minIndex: Int = 0
    def maxIndex: Int = data.length
    def apply(i: Int) = data(i)
  }
}
```

These implicit conversions can be brought into scope with the `import` statement.

```scala
import swiftvis2.spark._
```

Unfortunately, a Spark `Column` does not map to data on its own. It requires a `DataFrame` in order to get actual data for that column. This is the reason for the second argument to `ColumnToDoubleSeries`, which includes a second meaning of implicit in the Scala language. This is an implicit argument that will be provided automatically, assuming that one is in scope. The result is that a SwiftVis2 `Plot` can be constructed for a particular `DataFrame` using code like the following, which assumes the `DataFrame` has columns named `x`, `y`, and `z`.

```scala
implicit val df = spark.createDataset(pnts)
val cg = ColorGradient(0.0 -> RedARGB, 100.0 -> GreenARGB)
Plot.simple(ScatterStyle('x, 'y), colors = cg(doubles(df, 'z))), "Color_Grad", "X", "Y")
```

The implicit declaration of `df` enables the implicit conversions. The curly braces are put there to limit the scope of that implicit declaration. The key point we want to highlight is that the natural integration of Spark into the library is really made possible by the usages of implicit in the Scala language. Without these constructs, the binding would be more verbose.

The above example only works if all the data is coming from a single `DataFrame`. That should be the case the vast majority of the time, but in order to handle the situations where it isn’t, we also have functions with names like `doubles` that allow a concise way of getting data out of `DataFrames` by their column names, as shown in the following code.

```scala
Plot.simple(ScatterStyle(doubles(df, 'x), doubles(df, 'y), colors = cg(doubles(df, 'z))), "Color_Grad", "X", "Y")
```

In this example it would be a simple matter to pull the different values from separate `DataFrames`. Without the implicits, this is the usage that would always be required.

### 4. Performance and Quality

While the implementation of SwiftVis2 is still rather nascent, and as such, no attempts at optimization have yet been made, it is still interesting to compare its performance to some of the other libraries. The two other libraries that we chose were Breeze-viz and NSPL, both of which are also young libraries that aren’t all that well documented. To test the performance, we used data from a simulation of rings around the Centaur Chariklo. The simulation included just under 9 million particles. To make the plot more reasonable, we included every 10th particle in the plot. All three plotting libraries multithread their work, so they can’t be timed in the code. Instead, the timings were done by hand. After the initial render was done, we maximized the window and timed how long it took for the maximized image to appear. Figures 5 and 6 show the plots for SwiftVis2 and Breeze-vis respectively. Note that SwiftVis2 shows axis labels by default and Breeze-viz does not. The documentation for Breeze-viz did not make it clear how to add them, so we left them off here as they shouldn’t have a significant impact on render time. A figure for NSPL isn’t shown because it wasn’t clear they currently had the functionality to scale the points to the proper sizes for the particles. The images were exported at PNG files with a resolution of 2000x1500 pixels.

The timing results are shown in table 2. The time for NSPL comes from a usage without the points being scaled. It isn’t clear what impact this has on performance in NSLP, but in Breeze-viz, the change from a plot with simple dots
to one with scaled points took the timing from 10 seconds to the 15 seconds shown here. A similar scaling for NSLP would put it very close in performance to what we are seeing currently with SwiftVis2.

The fact that Breeze-viz performs well is not surprising, as it is simply a wrapper for the very mature and well optimized JFreeChart library. The question of output quality, being subjective, is left up to the reader to decide.

5. Conclusions

One of the major goals of the SwiftVis2 project is to make the use of Scala more viable for data exploration and general data science. There are still many improvements that need to be made, especially in the areas of the fluent interface and performance. The current implementation runs into issues with the JavaFX pipeline when there are too many strokes. This is a known issue with JavaFX as all operations are buffered for pipelining before being rendered. We will have to explore options for overcoming this limitation. Fortunately, the design of SwiftVis2, like many of the other plotting libraries referred to in this paper, allows us to easily implement different rendering schemes and for users to choose different rendering schemes. Note that figures 1 to 4 look different from figure 5 because the earlier ones were created using an SVG based renderer intended for web integration while figure 5 is drawn as an SVG image to handle the extremely large number of items being drawn.

References