rCharts Documentation

Release 0.1.0

Ramnath Vaidyanathan

November 14, 2016

Contents

rCharts is an R package to create, customize and publish interactive javascript visualizations from R using a familiar lattice style plotting interface.

CHAPTER 1

Quick Start

You can install rCharts from github using the devtools package

```
require(devtools)
install_github('rCharts', 'ramnathv')
```

The design philosophy behind rCharts is to make the process of creating, customizing and sharing interactive visualizations easy.

Create

rCharts uses a formula interface to specify plots, just like the lattice package. Here are a few examples you can try out in your R console.

```
require(rCharts)

## Example 1 Facetted Scatterplot
names(iris) = gsub("\\.", "", names(iris))
rPlot(SepalLength ~ SepalWidth | Species, data = iris, color = 'Species', type = 'point')

## Example 2 Facetted Barplot
hair_eye = as.data.frame(HairEyeColor)
rPlot(Freq ~ Hair | Eye, color = 'Eye', data = hair_eye, type = 'bar')
```

Customize

rCharts supports multiple javascript charting libraries, each with its own strengths. Each of these libraries has multiple customization options, most of which are supported within rCharts.

Share

rCharts allows you to share your visualization in multiple ways, as a standalone page, embedded in a shiny application, or embedded in a tutorial/blog post.

Publish to Gist/RPubs

```
names(iris) = gsub("\\.", "", names(iris))
r1 <- rPlot(SepalLength ~ SepalWidth | Species, data = iris,
  color = 'Species', type = 'point')
r1$publish('Scatterplot', host = 'gist')
r1$publish('Scatterplot', host = 'rpubs')
```

Use with Shiny

rCharts is easy to embed into a Shiny application using the utility functions renderChart and showOutput. Here is an example of an rCharts Shiny App.

```
## server.r
require(rCharts)
shinyServer(function(input, output) {
  output$myChart <- renderChart({
    names(iris) = gsub("\\.", "", names(iris))
    p1 <- rPlot(input$x, input$y, data = iris, color = "Species",
      facet = "Species", type = 'point')
    return(p1)
  })
})

## ui.R
require(rCharts)
shinyUI(pageWithSidebar(
  headerPanel("rCharts: Interactive Charts from R using polychart.js"),

  sidebarPanel(
    selectInput(inputId = "x",
     label = "Choose X",
     choices = c('SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth'),
     selected = "SepalLength"),
    selectInput(inputId = "y",
      label = "Choose Y",
      choices = c('SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth'),
      selected = "SepalWidth")
  ),
  mainPanel(
    showOutput("myChart", "polycharts")
  )
))
```

1.1 Credits

Most of the implementation in `rCharts` is inspired by rHighcharts and rVega. I have reused some code from these packages verbatim, and would like to acknowledge the efforts of its author Thomas Reinholdsson, who has since merged his rHighcharts package into rCharts. I would also like to thank @timelyportfolio for adding Dimple JS to rCharts, as well as for his contagious enthusiasm, which has egged me on constantly.

1.2 License

rCharts is licensed under the MIT License. However, the JavaScript charting libraries that are included with this package are licensed under their own terms. All of them are free for non-commercial and commercial use, with the exception of **Polychart** and **Highcharts**, both of which require paid licenses for commercial use. For more details on the licensing terms, you can consult the `License.md` file in each of the charting libraries.

1.3 See Also

There has been a lot of interest recently in creating packages that allow R users to make use of Javascript charting libraries.

- ggvis by RStudio

- clickme by Nacho Caballero

CHAPTER 2

Getting Started

You can install rCharts from github using the devtools package

```
require(devtools)
install_github('rCharts', 'ramnathv')
```

The design philosophy behind rCharts is to make the process of creating, customizing and sharing interactive visualizations easy.

2.1 Create

rCharts uses a formula interface to specify plots, just like the lattice package. Here are a few examples you can try out in your R console.

Note: Every example comes with an edit button that allows you to experiment with the code online. The online playground was built using OpenCPU

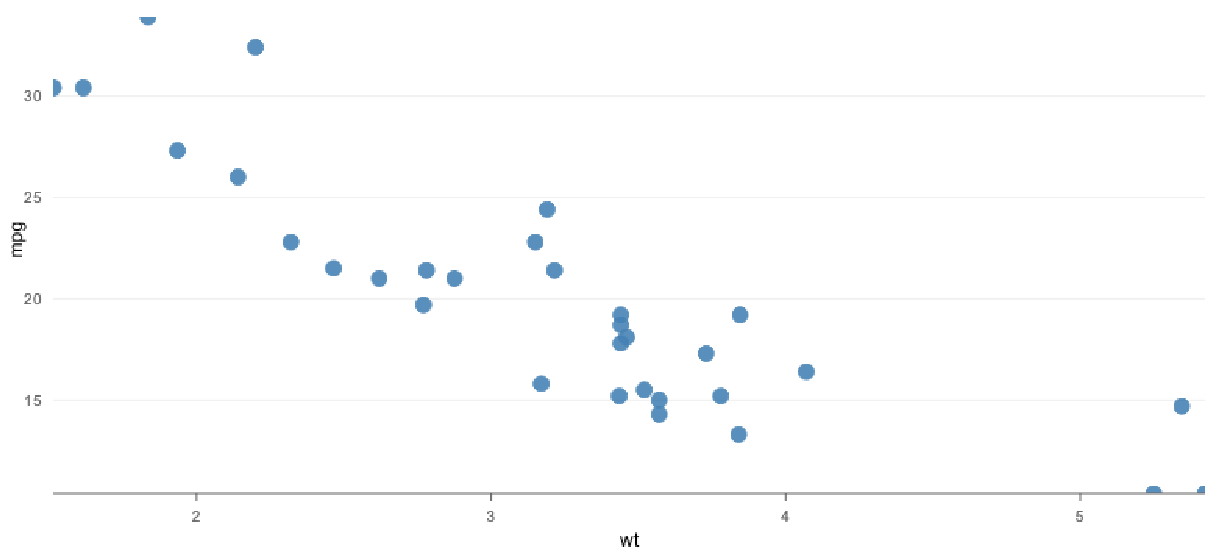
First let us load the `rCharts` package

```
library(rCharts)
```

2.1.1 Polycharts

```
r1 <- rPlot(mpg ~ wt, data = mtcars, type = 'point')
r1
```

```
Standalone
```



2.1.2 NVD3

```r
hair_eye = as.data.frame(HairEyeColor)
p2 <- nPlot(Freq ~ Hair, group = 'Eye',
  data = subset(hair_eye, Sex == "Female"),
  type = 'multiBarChart'
)
p2$chart(color = c('brown', 'blue', '#594c26', 'green'))
p2
```

intro/_downloads/intro-nvd3.pdf

View Interactive

2.1.3 Morris

```r
data(economics, package = "ggplot2")
econ <- transform(economics, date = as.character(date))
m1 <- mPlot(x = "date", y = c("psavert", "uempmed"), type = "Line", data = econ)
m1$set(pointSize = 0, lineWidth = 1)
m1
```

Standalone

2.1.4 Highcharts

```r
h1 <- hPlot(x = "Wr.Hnd", y = "NW.Hnd",
  data = MASS::survey,
  type = c("line", "bubble", "scatter"),
  group = "Clap",
  size = "Age"
)
h1
```

Standalone

2.1.5 Rickshaw

```
usp = reshape2::melt(USPersonalExpenditure)
usp$Var2 <- as.numeric(as.POSIXct(paste0(usp$Var2, "-01-01")))
p4 <- Rickshaw$new()
p4$layer(value ~ Var2, group = "Var1", data = usp, type = "area")
p4$set(slider = TRUE)
p4
```

Standalone

2.1.6 xCharts

```
require(reshape2)
uspexp <- melt(USPersonalExpenditure)
names(uspexp)[1:2] = c('category', 'year')
x1 <- xPlot(value ~ year, group = 'category', data = uspexp,
  type = 'line-dotted')
x1
```

Standalone

2.2 Share

Any visualization is useful only when you are able to share it. rCharts tries to make it really easy to share the visualizations you create. Let us first create a simple interactive scatterplot to illustrate the different sharing mechanisms built into rCharts

- *Save*
- *Publish*
- *Embed*

```
library(rCharts)
r1 <- rPlot(mpg ~ wt, data = mtcars, type = 'point')
```

2.2.1 Save

You can save your chart using the `save` method. The additional parameters passed to the `save` method determine how the js/css assets of the javascript visualization library are served. You can now email your visualization or embed it in a blog post as an iframe.

```
# link js/css assets from an online cdn
r1$save('mychart1.html', cdn = TRUE)
# create standalone chart with all assets included directly in the html file
r1$save('mychart2.html', standalone = TRUE)
```

2.2.2 Publish

Sometimes, you may want to directly publish the visualization you created, without having to bother with the steps of saving it and then uploading it. rChart has you covered here, and provides a `publish` method that combines these two steps. It currently supports publishing to RPubs and Gist and I expect to add more providers over time.

```
# the host defaults to 'gist'
r1$publish("My Chart")
r1$publish("My Chart", host = 'rpubs')
```

Publishing a chart saves the html in a temporary file, uploads it to the specified `host`, and returns a link to where the chart can be viewed. There are many gist viewers out there, and rCharts uses a custom viewer http://rcharts.io/viewer, designed specifically for rCharts, and is a modified version of another excellent gist viewer http://www.pagist.info/. Another popular gist viewer is http://blocks.org, built by Mike Bostock, the creator of d3.js.

If you wish to simply **update** a visualization you have already created and shared, you can pass the gist/rpubs id to the `publish` method, and it will update instead of uploading it as a brand new chart.

```
r1$publish("My Chart", id = 9253202)
```

While using a provider like Gist that allows multiple files to be uploaded, you can use the `extras` argument to add additional files that you want to upload. This is especially useful, if you want to provide a `README.md` or upload external assets like js/css/json files that are required for your chart to render.

```
r1$publish("My Chart", id = 9253202, extras = "README.md")
```

2.2.3 Embed

RMarkdown

Suppose you wish to embed a visualization created using rCharts in an Rmd document.

IFrame

One way to do this would be to use the `save` method to save the chart, and then embed it as an iframe. rCharts saves you the steps by allowing you to use the `show` method and specify that you want the chart to be embedded as an `iframe`.

We need to set the chunk options `comment = NA` and `results = "asis"` so that the resulting html is rendered asis and not marked up (which is the default in knitr).

~~~
```{r results = "asis", comment = NA}
r1$show('iframe', cdn = TRUE)
```
~~~

If you have several charts in your Rmd document, you can set these options globally in a setup chunk. Make sure to set `cache = F` for this chunk so that it is always run.

~~~
```{r setup, cache = F}
options(rcharts.mode = 'iframe', rcharts.cdn = TRUE)
knitr::opts_chunk$set(results = "asis", comment = NA)
```
~~~

You can now rewrite the earlier sourcecode chunk simply as

~~~
```{r}
r1
```
~~~

I prefer this style when writing, since it allows a user to simply copy paste sourcecode from the html and run it in their R console.

IFrame Inline

The `iframe` mode requires users to upload the additional chart html files along with their document. This introduces additional steps, and in the case of some providers like Rpubs, is not even possible. Hence, rCharts provides an additional mode named `iframesrc` that embeds the chart as an inline iframe, which makes your document self contained.

~~~
```{r results = "asis", comment = NA}
r1$show('iframesrc', cdn  = TRUE)
```
~~~

This option has the advantage of keeping the html standalone, but isolating the chart from the html on the page, thereby avoiding css and js conflicts. However, this feature is not supported by IE and Opera.

Inline

A third option to embed an rCharts created visualization is to inline the chart directly. Note that you need to add `include_assets = TRUE`, only the first time you are creating a chart using a specific library.

~~~
```{r chart3}
r1$show('inline', include_assets = TRUE, cdn = TRUE)
```
~~~

This approach should work in all browsers, however, it is susceptible to css and js conflicts.

If you are using Slidify to author your Rmd, then you can specify the charting library as `ext_widgets` in the YAML front matter. Here is a minimal reproducible example.

Note how you did not have to specify `include_assets = TRUE`. This is because slidify uses the `ext_widgets` property to automatically pick up the required assets and include them in the header of the resulting html page.

Shiny

It is easy to embed visualizations created using rCharts into a Shiny application. The main idea is to make use of the utility functions `renderChart()` and `showOutput()`. The shiny application created using the code below, can be seen here

```
## server.r
require(rCharts)
shinyServer(function(input, output) {
  output$myChart <- renderChart({
    names(iris) = gsub("\\.", "", names(iris))
    p1 <- rPlot(input$x, input$y, data = iris, color = "Species",
      facet = "Species", type = 'point')
    p1$addParams(dom = 'myChart')
    return(p1)
  })
})

## ui.R
require(rCharts)
```

```
shinyUI(pageWithSidebar(
  headerPanel("rCharts: Interactive Charts from R using polychart.js"),

  sidebarPanel(
    selectInput(inputId = "x",
     label = "Choose X",
     choices = c('SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth'),
     selected = "SepalLength"),
    selectInput(inputId = "y",
      label = "Choose Y",
      choices = c('SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth'),
      selected = "SepalWidth")
  ),
  mainPanel(
    showOutput("myChart", "polycharts")
  )
))
```

CHAPTER 3

Libraries

rCharts supports multiple javascript visualization libraries

3.1 NVD3

NVD3 is an elegant visualization library that provides re-usable chart components based on d3.js. Here is an excerpt directly taken from the NVD3 website.

> "This project is an attempt to build re-usable charts and chart components for d3.js without taking away the power that d3.js gives you. This is a very young collection of components, with the goal of keeping these components very customizeable, staying away from your standard cookie cutter solutions."

3.1.1 Create

The NVD3 library supports most of the common chart types.

- *Scatter Chart*
- *Multibar Chart*
- *Multibar Horizontal Chart*
- *Pie Chart*
- *Donut Chart*
- *Line Chart*
- *Line with Focus Chart*
- *Stacked Area Chart*
- *Multi Chart*

You can create an interactive plot making use of the NVD3 library using the `nPlot()` function.

Argument	Type	Description
x	formula	A formula of the form y ~ x, with column names from the data frame.
data	data frame	A data frame containing the data to be plotted
type	string	The type of chart to plot
group	string	Name of column based on which data should be grouped.

Scatter Chart

```r
p1 <- nPlot(mpg ~ wt, group = 'cyl', data = mtcars, type = 'scatterChart')
p1$xAxis(axisLabel = 'Weight')
p1
```

```
Standalone
```

Multibar Chart

```r
hair_eye = as.data.frame(HairEyeColor)
p2 <- nPlot(Freq ~ Hair, group = 'Eye',
  data = subset(hair_eye, Sex == "Female"),
  type = 'multiBarChart'
)
p2$chart(color = c('brown', 'blue', '#594c26', 'green'))
p2
```

```
Standalone
```

Multibar Horizontal Chart

```
Standalone
```

Pie Chart

```r
p4 <- nPlot(~ cyl, data = mtcars, type = 'pieChart')
p4
```

```
Standalone
```

Donut Chart

```r
p5 <- nPlot(~ cyl, data = mtcars, type = 'pieChart')
p5$chart(donut = TRUE)
p5
```

```
Standalone
```

Line Chart

```r
data(economics, package = 'ggplot2')
p6 <- nPlot(uempmed ~ date, data = economics, type = 'lineChart')
p6
```

```
Standalone
```

Line with Focus Chart

```
ecm <- reshape2::melt(
  economics[,c('date', 'uempmed', 'psavert')],
  id = 'date'
)
p7 <- nPlot(value ~ date, group = 'variable',
  data = ecm,
  type = 'lineWithFocusChart'
)
p7
```

```
Standalone
```

Stacked Area Chart

```
dat <- data.frame(
  t = rep(0:23, each = 4),
  var = rep(LETTERS[1:4], 4),
  val = round(runif(4*24,0,50))
)
p8 <- nPlot(val ~ t, group =  'var', data = dat,
 type = 'stackedAreaChart', id = 'chart'
)
p8
```

```
Standalone
```

Multi Chart

```
p12 <- nPlot(value ~ date, group = 'variable', data = ecm, type = 'multiChart')
p12$set(multi = list(
  uempmed = list(type="area", yAxis=1),
  psavert = list(type="line", yAxis=2)
))
p12$setTemplate(script = system.file(
  "/libraries/nvd3/layouts/multiChart.html",
  package = "rCharts"
))
p12
```

```
Standalone
```

Tutorials

4.1 Visualizing Strikeouts

This tutorial explains in detail, how I used `rCharts` to replicate this NY times interactive graphic on strikeouts in baseball. The end result can be seen here as a `shiny` application.

4.1.1 Data

The first step is to get data on strikeouts by team across years. The NY Times graphic uses data scraped from baseball-reference, using the `XML` package in R. However, I will be using data from the R package Lahman, which provides tables from Sean Lahman's Baseball Database as a set of data frames.

The data processing step involves using the plyr package to create two data frames:

1. `team_data` containing `SOG` (strikeouts per game) by `yearID` and team `name`
2. `league_data` containing `SOG` by *yearID* averaged across the league.

```
require(Lahman) ; require(plyr); library(ascii)
dat = Teams[,c('yearID', 'name', 'G', 'SO')]
team_data = na.omit(transform(dat, SOG = round(SO/G, 2)))
league_data = ddply(team_data, .(yearID), summarize, SOG = mean(SOG))
ascii(head(team_data), type = 'rst')
```

	yearID	name	G	SO	SOG
1	1871.00	Boston Red Stockings	31.00	19.00	0.61
2	1871.00	Chicago White Stockings	28.00	22.00	0.79
3	1871.00	Cleveland Forest Citys	29.00	25.00	0.86
4	1871.00	Fort Wayne Kekiongas	19.00	9.00	0.47
5	1871.00	New York Mutuals	33.00	15.00	0.45
6	1871.00	Philadelphia Athletics	28.00	23.00	0.82

4.1.2 Charts

We will start by first creating a scatterplot of *SOG* by *yearID* across all teams. We use the *rPlot* function which uses the PolyChartsJS library to create interactive visualizations. The formula interface specifies the x and y variables, the data to use and the type of plot. We also specify a *size* and *color* argument to style the points. Finally, we pass a *tooltip* argument, which is a javascript function that overrides the default tooltip to display the information we require. You will see below the R code and the resulting chart.

```
require(rCharts)
p1 <- rPlot(SOG ~ yearID, data = team_data,
  type = "point",
  size = list(const = 2),
  color = list(const = "#888"),
  tooltip = "#! function(item){
    return item.SOG + ' ' + item.name + ' ' + item.yearID
   } !#"
)
p1
```

`Standalone`

tutorials/_downloads/chart1.pdf

Now, we need to add a line plot of the average `SOG` for the league by `yearID`. We do this by adding a second layer to the chart, which copies the elements of the previous layer and overrides the `data`, *type*, `color` and `tooltip` arguments. The R code is shown below and you will note that the resulting chart now shows a blue line chart corresponding to the league average `SOG`.

```
p1$layer(data = league_data, type = 'line',
  color = list(const = 'blue'), copy_layer = T, tooltip = NULL)
p1
```

`Standalone`

tutorials/_downloads/chart2.pdf

Finally, we will overlay a line plot of `SOG` by `yearID` for a specific team *name*. Later, while building the shiny app, we will turn this into an input variable that a user can choose from a dropdown menu. We use the layer approach used earlier and this time override the *data* and *color* arguments so that the line plot for the team stands out from the league average.

```
myteam = "Boston Red Sox"
p1$layer(data = team_data[team_data$name == myteam,],
  color = list(const = 'red'),
  copy_layer = T)
p1$set(dom = 'chart3')
p1
```

`Standalone`

tutorials/_downloads/chart3.pdf

Let us add a little more interactivity to the chart. To keep it simple, we will use handlers in PolychartJS to initiate an action when a user clicks on a point. The current handler is a simple one, which just displays the name of the team clicked on. If you are familiar with Javascript event handlers, the code should be self explanatory.

```
p2 <- p1$copy()
p2$setTemplate(afterScript = '
  <script>
    graph_chart3.addHandler(function(type, e) {
      var data;
      data = e.evtData;
      if (type === "click") {
        return alert("You clicked on the team: " + data.name["in"][0]);
      }
    });
  </script>
')
p2
```

```
Standalone
```

4.1.3 Application

Now it is time to convert this into a Shiny App. We will throw the data processing code into *global.R* so that it can be accessed both by *ui.R* and *server.R*. For the dropdown menu allowing users to choose a specific team, we will restrict the choices to only those which have data for more than 30 years. Accordingly, we have the following *global.R*.

```
## global.R
require(Lahman); require(plyr)
dat = Teams[,c('yearID', 'name', 'G', 'SO')]
team_data = na.omit(transform(dat, SOG = round(SO/G, 2)))
league_data = ddply(team_data, .(yearID), summarize, SOG = mean(SOG))
THRESHOLD = 30
team_appearances = count(team_data, .(name))
teams_in_menu = subset(team_appearances, freq > THRESHOLD)$name
```

For the UI, we will use a bootstrap page with controls being displayed in the sidebar. Shiny makes it really easy to create a page like this. See the annotated graphic below and the *ui.R* code that accompanies it to understand how the different pieces fit together.

We now need to write the server part of the shiny app. Thankfully, this is the easiest part, since it just involves wrapping the charting code inside *renderChart* and replacing user inputs to enable reactivity. We add a few more lines of code to set the height and title and remove the axis titles, since they are self explanatory.