PhUSE 2015

Paper CS07

Create interactive web graphics out of your SAS or R datasets

Patrick René Warnat, HMS Analytical Software GmbH, Heidelberg, Germany

ABSTRACT

Several commercial software products allow the creation of interactive graphics. For some tasks, open source software solutions can be sufficient to present data in an interactive way. In this talk it is shown for one example how an ensemble of plots (bar chart, pie chart...) can be created for display in a web browser, visualizing data delivered by a SAS Stored Process or by a HTTP service interface to R. The ensemble of plots is interactive in the way that by clicking and thus selecting different parts of one graph, all other graphs are updated and filtered to show only the corresponding sub-part of the data. The solution enables users to interactively drill down into sub-parts of a data, providing a flexible graphical examination of a data set.

INTRODUCTION

Data visualization is a powerful tool for the analysis of quantifiable information, because it allows using a particular strength of human beings: the visual perception. Through the graphic representation of data, relationships can be identified faster and easier than without visualization. This effect is further intensified by using visualization software that supports interactive work with graphics.

There are several commercial software products available that provide tools for generation of interactive graphics that can be used in a wide variety of use cases. For some tasks, open source software solutions can be sufficient to present data in an interactive way.

In this paper, it is shown how to graphically present variables of a dataset in an interactive way using ubiquitously available web technologies and open source software libraries. An ensemble of plots (bar chart, pie chart...) is created for display in a web browser using HTML, CSS and JavaScript. In the simplest use case the visualized data is provided as a flat file, as an alternative it is shown how data delivery could be implemented with an R or SAS based server backend. The ensemble of plots is interactive in the way that by clicking and thus selecting different parts of one graph, all other graphs are updated and filtered to show only the corresponding sub-part of the data. The solution enables users to interactively drill down into sub-parts of a data, providing a flexible graphical examination of a data set.

The solution shown here for one particular example dataset is easily applicable to other datasets. It could be used as a blueprint for a quick and easy solution to provide interactive diagrams with the optional possibility to connect to different data serving backend technologies.

The remainder of this paper is structured as follows: First the specific example scenario is described in detail, second, the technical details for the web based interactive graphics are explained, then two possible variants for data providing backend solutions are outlined, and finally a conclusion is given.

EXAMPLE SCENARIO

As an example, data from the website ClinicalTrails.gov were used: Descriptive data for studies found on ClinicalTrials.gov by search term”Influenza” (search results retrieved at 13/August/2015).

For this search, n = 2513 studies were found and descriptive data was downloaded as a “tsv” file (tab separated values) utilizing the download feature provided at the search results page of ClinicalTrails.gov. The downloaded file contains 2514 rows, first row with column headers and every following row describing one study. Out of the available variables (columns) describing the found studies, the following four categorical variables where used:

- Study type: observational or interventional
- Study results availability: results available at ClinicalTrails.gov or not
- Study phase
- Age group: investigated age group

WEB FRONTEND WITH INTERACTIVE GRAPHICS

In order to create interactive diagrams for the example data set, HTML, CSS and JavaScript were used. More specifically, a custom web page was created on an HTTP server that uses the open source Java Script library “dc.js” [1] to create an interactive panel of diagrams that can be used by opening an HTML file in a web browser (see figure 1). All plots are interactive in the way that by clicking and thus selecting different parts of one graph, all other graphs are updated and filtered to show only the corresponding sub-part of the data. Figure 2 shows the panel of charts after selection of a slice in the second pie chart, namely only studies for which results are available at ClinicalTrials.gov. A number of 415 studies are selected by this interaction and all other charts are automatically

updated to show only the corresponding sub-part of the data. A line of text informs about of number of currently selected records and allows for a reset of all selections.

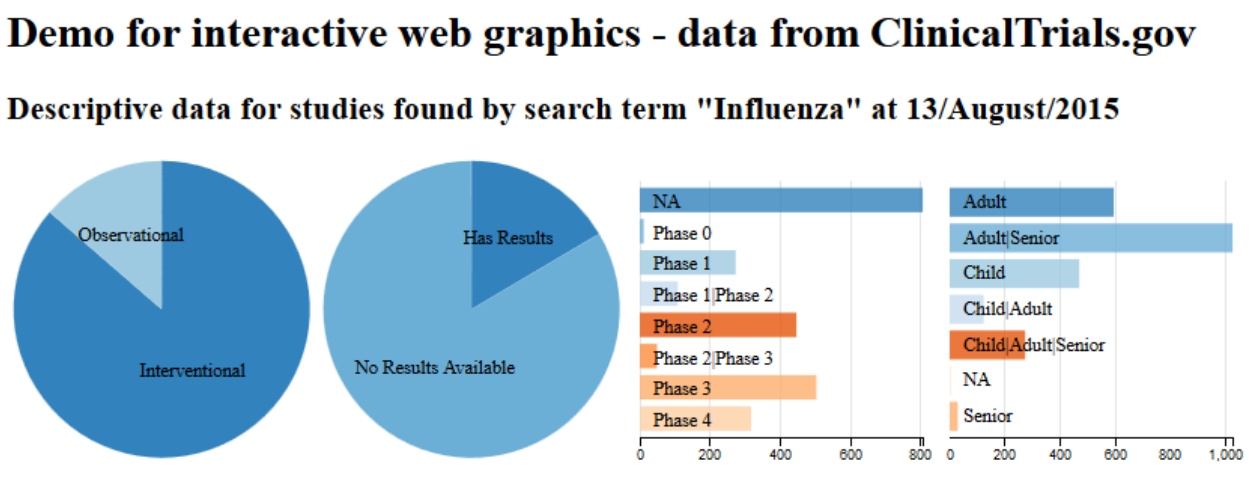

Figure 1: Demo panel of interactive graphics as rendered in a web browser. The diagrams display the number of category occurrences for four different attributes.

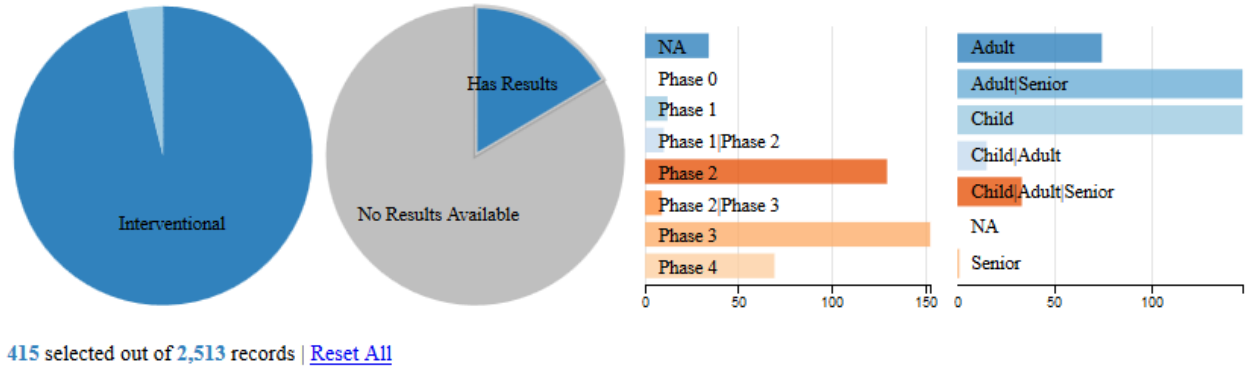

Figure 2: Demo panel after interaction. In the second pie chart, the slice “Has Results” was clicked, thereby selecting only studies where results are available. All other charts are automatically updated to show only the corresponding sub-part of the data.

The following files are used to create the panel of interactive diagrams:

```
index.html

—css
     dc.css
     style.css

—data
     study_fields.tsv

—js
     crossfilter.min.js
     d3.min.js
     dc.min.js
```

The file index.html contains the HTML for the web page structure and the JavaScript Code to read in the data and to create the diagrams utilizing the library dc.js. The dc.js library is located in the js subdirectory as a minified version, along with the two other libraries of which dc.js is dependent (crossfilter.js [2] and d3.js [3]). The data is contained as a tabulator separated file in subdirectory data. The css subdirectory contains two files: dc.css is provided together with dc.js and style.css was created in order to modify the styling of the diagrams, in particular the size and color of the text labels.

Full source code of the files styles.css and index.html are printed in the appendix. The contents of the file index.html can be summarized with a list of code blocks as follows:

```
- HTML Head with title definition and links (imports) of CSS files
- HTML Body:
    - Headlines definitions
    - several Div-Blocks for definition of the different panel elements
    - links (imports) of js libraries
    - custom Java Script code to read in data and define the diagrams:
          - function replaceMissingWithMarkerNA: simple replacement of empty
          strings with string NA, used during data import
          - function createCharts: used to define the interactive diagrams using
          dc.js, this function is designed to be used as a callback of an
          d3 data import function (see below)
          - a call to the function d3.tsv, a function that reads tabular separated
          files, allows definition of preprocessing (here a call to
          replaceMissingWithMarkerNA) and which calls function createCharts when
          data is read and available.
```

The most interesting part of the JavaScript source code is the function 'createCharts'. In this function the shown diagrams are defined using a declarative syntax. The following example shows the steps necessary to define one pie chart.

```
// the variable data contains the tabular input data read by function d3.tsv as a
// list of JSON objects
// the crossfilter function takes a list of JSON objects, and creates an crossfilter
// object
var   crf               = crossfilter(data);
// using the crossfilter object, we define the column 'types' of the data as a
// dimension, which can be used to group or filter data
var   typesDimension    = crf.dimension(function(d) {return d.types;});
// the group function constructs a new grouping for the given dimension, according
to a specified groupValue function. The groupValue function is optional; if not
specified, as it is the case here, the number of records per group will be counted.
var   typesGroup        = typesDimension.group();
//define a pie chart; the referenced HTML DIV element defines where on the page the
diagram will be located; width, height and radius define the size of the diagram and
dimension and group define the shown information
var typesPieChart  = dc.pieChart('#chart-pie-types');
           typesPieChart
              .width(230)
              .height(230)
              .radius(110)
              .dimension(typesDimension)
              .group(typesGroup)
           ;
// finally, a function call to render the diagram on the page
dc.renderAll();
```

All other charts are defined in a similar manner. By using the same crossfilter object to define dimensions and groups for the different diagrams, they all are interconnected as described further above. Thus, no explicit programming is necessary to create a panel of interactive graphics, this functionality is completely provided by the dc.js library.

In the example above, the data is provided as a tabular separated text file (tsv), directly located at the HTTP server that provides the files for the front end (HTML, CSS, JavaScript). As described, this file was manually downloaded from ClinicalTrials.gov for this demo. In other scenarios, the date file could be automatically generated or updated by scheduled backend processes like scheduled execution of SAS or R programs. As an alternative, the requested data could be provided on-the-fly by HTTP based services, as the data reading functions of the d3 library, like the d3.tsv functions are based on HTTP GET requests. The following two paragraphs give an overview on how such a data providing service could be implemented using SAS or R based technologies.

SAS STORED PROCESS AS DATA PROVIDING BACKEND

Utilizing the SAS Stored Process Web Application [4], SAS Stored Processes (STP) can be invoked directly using HTTP GET and STPs can return data as part of corresponding HTTP Response. For usage with the d3.tsv (or similarly d3.csv) data import function as used in the example above, it would be possible to implement a STP that

directly returns a data set in tsv format. In the front end JavaScript code call of the d3.tsv function, the relative path of a tsv file at the web server would be replaced with the URL of the STP that is providing the data. Please note that it is assumed for this example scenario that the web server providing the front end code and the Stored Process Web Application providing the STP HTTP interface run on the same host as otherwise most browsers will deny the call to the STP by default, due to a violation of the same-origin policy [5]. As an alternative, it is possible to provide the front end code itself as the result of an STP or make use of JSONP, a method to encapsulate the client-server communication.

HTTP SERVICE INTERFACE TO R AS DATA PROVIDING BACKEND

There are several software solutions available to implement a data providing HTTP service based on R. One of these solutions is OpenCPU [6]. OpenCPU is a system that provides a HTTP API to an R installation, providing ways to call R functions or R scripts and/or to retrieve data over HTTP. The OpenCPU system is available in two variants: The first variant is a R package that can be used in a local R installation for development, the second variant is a Linux server installation package for use in production. In addition, there is a publicly accessible server installation available on the domain opencpu.org. For example, using the public OpenCPU server that has the R package MASS installed, the URL to retrieve the data set Cars93 of the package MASS as a csv file is:
`http://public.opencpu.org/ocpu/library/MASS/data/Cars93/csv`
A URL like this can be directly used with the d3 import functions like d3.csv to import data sets as described in the example scenario. Be aware that the same-origin policy [5] as implemented in web browser needs to be taken into consideration here as well. A possible solution could be to configure the web server providing the front end code that it proxies the requests to a local OpenCPU server instance (see figure 3).

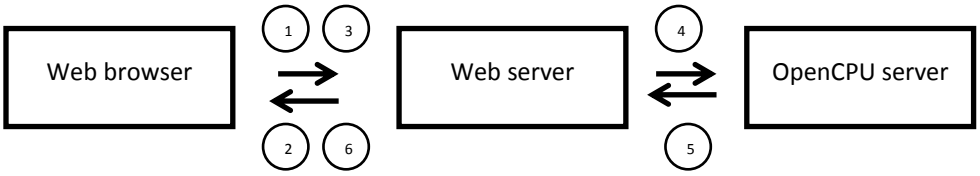

Figure 3: Possible scenario of using an OpenCPU server together with a web server. (1) The web browser requests the index.html page. (2) The webserver provides the front end Code (HTML, CSS and JavaScript) to the web browser. (3) The delivered Java Script code executed at the web browser fetches data (e.g. by a call to the d3.csv function) using an URL on the web server. (4) The web server acts as a reverse proxy to the OpenCPU server. (5) The OpenCPU server is providing data out of an R installation over HTTP. (6) The reverse proxy feature of the web server passes the data to the web browser.

CONCLUSION

Graphical presentation of data is very helpful, especially in order to find and interpret relationships. Interactivity of data visualizations enable users to a certain degree to select on which aspects of the presentation they want to concentrate on or in which order they explore different aspects.
Interactive diagrams can be created with several commercial software packages. For some tasks, open source software solutions can be sufficient to present data in an interactive way, and in this paper one way was shown to accomplish this with minimal effort. The presented example can be enhanced in several ways and combined with different data providing backends and can be integrated in existing frontend code.

REFERENCES

web links as accessed on 2nd September 2015:

[1] http://dc-js.github.io/dc.js/
[2] http://square.github.io/crossfilter/
[3] http://d3js.org/
[4] http://support.sas.com/documentation/cdl/en/stpug/68399/HTML/default/viewer.htm#n1gt44n8wc0la0n18s9kajwq0o2q.htm
[5] https://en.wikipedia.org/wiki/Same-origin_policy
[6] https://www.opencpu.org/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Dr. Patrick R. Warnat
HMS Analytical Software GmbH
Rohrbacher Str. 26
69115 Heidelberg
Germany
http://www.analytical-software.de/en/

APPENDIX

Full source code of the files index.html and styles.css as described in paragraph “Web Frontend with interactive graphics” of this paper.

File index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Demo for interactive web graphics - data from ClinicalTrials.gov</title>
    <link rel="stylesheet" href="./css/dc.css">
    <link rel="stylesheet" href="./css/style.css">
  </head>
  <body>
    <h1>Demo for interactive web graphics - data from ClinicalTrials.gov</h1>
    <h2>Descriptive data for studies found by search term "Influenza" at 13/August/2015</h2>

    <div id="chart-pie-types"></div>
            <div id="chart-pie-results"></div>
    <div id="chart-row-phases"></div>
    <div id="chart-row-ageGroups"></div>

    <div class="dc-data-count">
      <span class="filter-count"></span> selected out of <span class="total-count"></span> records | <a
        href="javascript:dc.filterAll(); dc.renderAll();">Reset All</a>
    </div>

    <script type="text/javascript" src="./js/d3.min.js"></script>
    <script type="text/javascript" src="./js/crossfilter.min.js"></script>
    <script type="text/javascript" src="./js/dc.min.js"></script>
    <script type="text/javascript">

              //simple replacement of empty   strings with string NA
                        //used during data import
      replaceMissingWithMarkerNA = function(value) {
        var res = "NA";
        if (value){
          res = value;
        };
        return(res);
      };

                        //function to define the interactive diagrams using dc.js,
                        //this function is designed to be used as a callback of an
                        //d3 data import function (see below)
      createCharts = function(data) {
        // the variable data contains the tabular input data read by function d3.tsv as a
        // list of JSON objects
        // the crossfilter function takes a list of JSON objects, and creates an crossfilter
        // object
                                  var   crf             = crossfilter(data);
        var   all             = crf.groupAll();

        // using the crossfilter object, we define selected columns of the data as a
        // dimension, which can be used to group or filter data
                                  // the group function constructs a new grouping for the given dimension,
                                  // according to a specified groupValue function. The groupValue function
                                  // is optional; if not specified, as it is the case here, the number of
                                  // records per group will be counted.
        var   resultsDimension   = crf.dimension(function(d) {return d.results;});
        var   resultsGroup       = resultsDimension.group();
        var   typesDimension     = crf.dimension(function(d) {return d.types;});
        var   typesGroup         = typesDimension.group();
        var   phasesDimension    = crf.dimension(function(d) {return d.phases;});
        var   phasesGroup        = phasesDimension.group();
```

```
  var   ageGroupsDimension = crf.dimension(function(d) {return d.ageGroups;});
  var   ageGroupsGroup     = ageGroupsDimension.group();

                    //define a pie chart; the referenced HTML DIV element defines where on the page
                            //the diagram will be located; width, height and radius define the size of the
                            //diagram and dimension and group define the shown information
  var resultsPieChart  = dc.pieChart('#chart-pie-results');
  resultsPieChart
    .width(230)
    .height(230)
    .radius(110)
                              .dimension(resultsDimension)
    .group(resultsGroup)
  ;
  //define a pie chart
  var typesPieChart  = dc.pieChart('#chart-pie-types');
  typesPieChart
    .width(230)
    .height(230)
    .radius(110)
    .dimension(typesDimension)
    .group(typesGroup)
  ;
 //define a row chart (horizontal bar chart)
  var phasesRowChart  = dc.rowChart('#chart-row-phases');
  phasesRowChart
    .width(230)
    .height(230)
    .margins({top: 20, left: 10, right: 10, bottom: 20})
    .dimension(phasesDimension)
    .group(phasesGroup)
    .elasticX(true)
    .xAxis().ticks(4);
  ;
                            //define a row chart (horizontal bar chart)
  var ageGroupsRowChart  = dc.rowChart('#chart-row-ageGroups');
  ageGroupsRowChart
    .width(230)
    .height(230)
    .margins({top: 20, left: 10, right: 10, bottom: 20})
    .dimension(ageGroupsDimension)
    .group(ageGroupsGroup)
    .elasticX(true)
    .xAxis().ticks(4);
  ;
  //define a data count for display of the numer of selected
                            //and the total number of items
  var   selectedDataCount = dc.dataCount('.dc-data-count');
  selectedDataCount
    .dimension(crf)
    .group(all)
  ;

                            // finally, a function call to render the diagram on the page
  dc.renderAll();
};

//read in data
d3.tsv(
  //data source url, can be local flat file or file from server
  "./data/study_fields.tsv",
  //accessor function for data row processing
                            //it is defined which colums are read, and that they are preprocessed
                            //with function replaceMissingWithMarkerNA
  function(d) {
    return {
```

```
            types     : replaceMissingWithMarkerNA(d["Study Types"]),
            results   : replaceMissingWithMarkerNA(d["Study Results"]),
            phases    : replaceMissingWithMarkerNA(d["Phases"]),
            ageGroups : replaceMissingWithMarkerNA(d["Age Groups"])
          };
        },
        //callback function which is called when the data is available
        function (data) {
          createCharts(data);
        }
      );
    </script>
  </body>
</html>
```

File style.css

```
#chart-pie-results .pie-slice {
        fill: black;
        font-size: 14px;
}
#chart-pie-types .pie-slice {
        fill: black;
        font-size: 14px;
}
#chart-row-phases .row text {
   fill: black;
        font-size: 14px;
}
#chart-row-ageGroups .row text {
   fill: black;
        font-size: 14px;
}
```