Getting started with data exploration on the web w/ D3

web science summer school 2014

introduction to D3 ::

D3 is a data-oriented Web micro-framework that allows Web programmers to focus on data rather than everything around it. It follows the tradition of jQuery to provide 'fluent interfaces' that makes code extremely succinct and (sometimes!) easy to read.

Unlike using Matlab, R, SPSS or Excel, however, it is a general-purpose *data drawing* API. This means two things: first, that it takes a bit more time and investment to build things in D3 than it does to plot things in any high level tool or statistical analysis toolkit. But it also means that you have far greater flexibility about what you can build - in fact, you can probably build almost any kind of data visualisation you can think of with the right amount of D3.

This tutorial will walk through the basics of getting started with playing with data in D3. To make this particularly relevant for budding Web Scientists, we will also walk you through a simple way to grab real data from Twitter in the very first example.

prerequisites & background reading ::

This tutorial assumes a familiarity with Web technologies, in particular with prior experience building static Web sties. Proficiency with JS is recommended but not absolutely necessary; a rudimentary knowledge of JS syntax, however, is recommended. I recommend "Eloquent Javascript" (http://eloquentjavascript.net/contents.html) for beginners, and "Javascript: The Good Parts" for advanced JS programmers.

If you would like to familiarise yourself with D3 before getting started, I recommend the simple 17 step Interactive Data Visualisation Tutorials at http://alignedleft.com/tutorials/D3

Part 1 : A Simple Tweet-Scatterplot

Note: the code in this section is available at **https://github.com/sociam/WS3-D3-Tutorial/**

1a- Setting up your environment

Let's set up your development environment. For convenience, we are going to use the node package manager (**npm**) to get us some useful things very quickly. Make sure you have npm installed; consult relevant documentation in order to do this.

Since we're building a static web site, we need two things: a static web server, and a package manager to help us manage our dependencies. For the first we are going to use the nodejs project **http-server** and the latter **bower**.

Create an empty directory (folder) somewhere, and install these using **npm**.

OS X / Linux:

```
sudo npm install -g http-server bower
```

Then, get bower to install some useful libraries for you.

```
bower install jQuery d3 underscore
```

Note: Ensure that you've spelled jQuery with a capital 'Q', or you will experience sadness due to an outstanding bug in the npm entries for jquery.

1b - Get some Tweets!

Twitter makes a huge variety of API endpoint calls available that let you ask for tweets, user profile and other things in various ways. In this example, we will use the Streaming Search API. You will need the free utility **cURL** to do this example on your computer. On OS X, curl is available via **ports**. On Linux, curl is included in most distributions. On Windows, it comes with Cygwin and is also available as a stand-alone download.

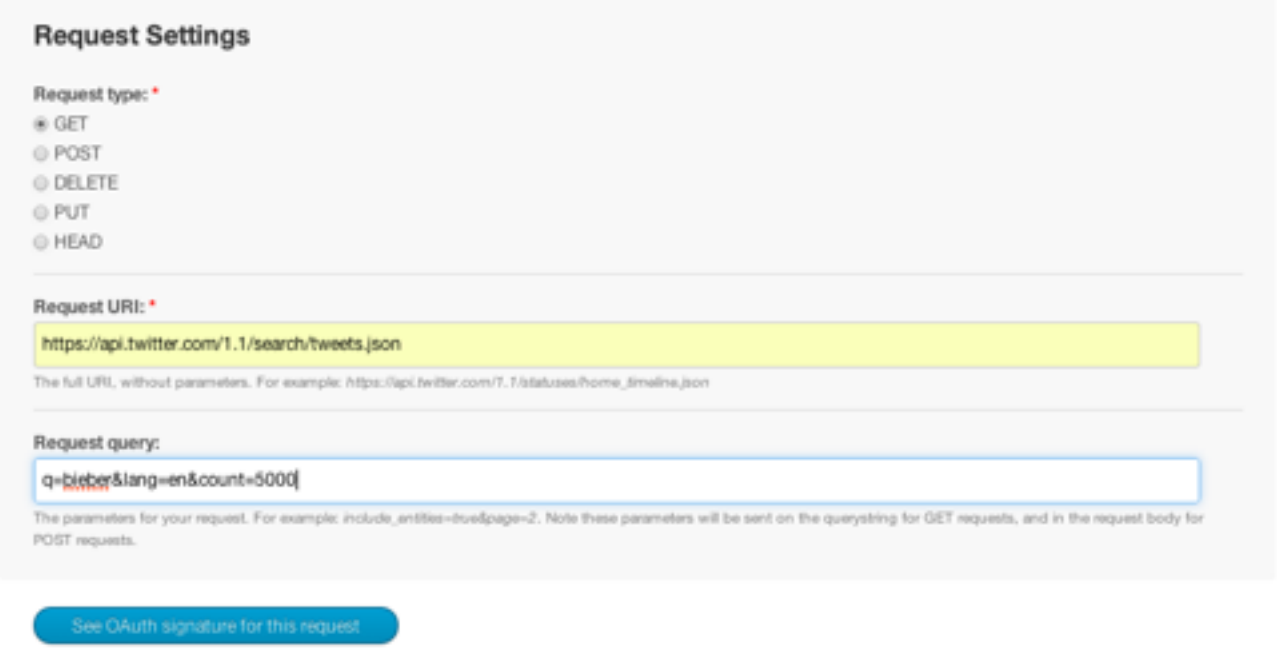
First things first, you need to register as a Twitter developer; go to https://apps.twitter.com/ , log in, and select **Create a New App**. You will have to give your new app a friendly name, description, and URL. Feel free to use your personal URL of your choice.

Then, find the Test OAuth button in the upper right of your app settings. (If the button is not there, find the "OAuth Dashboard" on the Twitter API site)

You should see a page with a box called **Request Settings**. Leave the request type **GET** and fill in for **Request URI**:

```
https://api.twitter.com/1.1/search/tweets.json
```

Then, for Request query, you can supply any queries you want in GET param syntax. To find the list of all possible search queries, refer to the Twitter documentation, in particular `https://dev.twitter.com/docs/api/1.1/get/search/tweets` .



For example, you can use the query `q=bieber&lang=en&count=5000` to search for the term “bieber”, in English, up to the first 5000 hits. Consult the documentation for details on how to make more sophisticated text queries.

Finally, hit **See OAuth signature for this request**

You will get a page with 3 parts, one of which says **cURL Command** .

Copy and paste the whole cURL command to your terminal. If it says “command not found”, ensure that curl is in your path (or specify a full path to curl in your command).

You should see a giant blob of JSON scroll across your terminal, vaguely containing the search keyword you included. You probably want to run the command again and pipe it to a file, like this

```
curl --get (…lots of stuff here…) --verbose > tweets.json
```

You might get an “permission denied” message from the Twitter API. That means your OAuth token has expired and you need to go back a page in your browser and re-issue click **See OAuth signature for this request** again.

Step 1c - Writing some code.

We have the data, we have the environment, now let’s write some code!

Let’s first set up a very bare bones, **index.html**

```
<!doctype html>
<html lang="en">
<head>
 <link href="index.css" rel="stylesheet"/>
 <title> playing with D3 </title>
</head>
<body>
  <script src="bower_components/jquery/jquery.min.js" type="text/javascript"></script>
  <script src="bower_components/underscore/underscore.js" type="text/javascript"></script>
  <script src="bower_components/d3/d3.min.js" type="text/javascript"></script>
  <script src="index.js" type="text/javascript"></script>
</body>
</html>
```

Notice that we have included references to the `bower_components` directory which is where bower put our dependencies. Check to make sure the relative paths are correct.

Now let's create a happy little index.css which we will update at the next stage

```
* {
   box-sizing:border-box;
   font-family:'helvetica-neue';
   font-weight:100;
}

body {
   padding:30px;
   height:100%;
}

svg {
   width:100%;
   height:80%;
   border:1px solid #eee;
   box-shadow: 5px 5px 5px #eee;
}
```

Notice that we've put in a handler for the SVG object. That's going to be our main drawing surface so we have decorated it extra nicely. Now, let's create an **index.js.** For now, keep it simple

```
jQuery(document).ready(function() {
   var svg,
       setup = function() { svg = d3.select('body').append('svg');  };
   $.get('/tweets.json').then(function(tweets) {
       console.log('tweets > ', tweets.statuses);
       setup();
       window.tweets = tweets.statuses;
   }).fail(function(data) {
       console.error('error loading data ', data);
   });
});
```

As you can see, we're simply using jQuery to fetch our tweets, and print them out for us.

Let's start our static HTTP server that we installed. On the terminal which to your base project directory and type : `http-server`

You will also notice that we have already started using D3! We have asked D3 to do something very simple; to create an svg element for us and put it in the body.

Let's give it a whirl to see if it works! Open a terminal, change to the directory that is the base of our project, and start an `http-server` . You should see some notification like this:

```
Starting up http-server, serving ./ on port: 8080
```

Now, point your browser to localhost:8080. You should see a lovely blank SVG canvas. Open the browser console, and you should see a debug message with the list of tweets that you can explore.

Step 1d - Let's do some plotting!

The first thing we will do is establish what we want to measure about the tweets. Let's make two functions which will compute the x and y axes values we want to compare in our scatterplot.

For this very silly example, I'm going to see whether the number of words in a Tweet correlates with the number of retweets it has for our dataset (obviously, it won't - this is merely for explanatory purposes!).

So, for x, we want a function that returns the number of words (tokens):

```
var xval = function(tweet) {
     return tweet.text.split(' ').filter(function(x) { return x.trim().length > 0; }).length;
};
```

Not beautiful, but that will do. For the y-axis we just need to select the retweet count

```
var yval = function(tweet) { return tweet.retweet_count; };
```

Now, let's set up our plotting function. this is where all the magic happens. Let's modify our document ready function to add a few more variable declarations:

var svg,hmargin = 40, vmargin = 40, xscale, yscale, width, height;

(The hmargin/vmargin values give us padding within the SVG to place the axes.)

Next, create a new **plot** function (see the next page for full listing.) The ,function we've written is divided into three sections. The first, **scaling** serves the purpose of declaring D3 scales, which are functions responsible for transforming the raw values we are plotting into SVG coordinates. **D3** uses *scale functions* to make this work; and, since we are plotting two different ranges of values against each other, we need two separate scale functions: one for x (the number of tokens) and one for y (the number of retweets). For each scale, we need to tell them explicitly what the min and max values of the data we will be plotting (the **domain**) is, and the output coordinates in SVG space we desire (the **range**). See the D3 documentation on **d3.scale** for the complete scale API.

The next, **drawing** section is where the party goes on; this simple "single line" function performs the entire scatterplots dataset. This is where D3 really shines. The way to read this is as follows:

1. First, *select* (in a jQuery sense) all circle elements with class pt in the SVG canvas, and *data join* them against the array **tweets**. This means, pair up each element in the selection, one at a time, against the elements of the data array, in order.

```
var plot = function(tweets) {
    width = $(‘svg').width();
    height = $('svg').height();
    xscale = d3.scale.linear().range([hmargin, width-2*hmargin]),
    yscale = d3.scale.linear().range([height-vmargin,vmargin]);

    var xvals = tweets.map(xval),
        yvals = tweets.map(yval);

    // establish the domain here.
    xscale.domain([_.min(xvals), _.max(xvals)]);
    yscale.domain([_.min(yvals), _.max(yvals)]);

    svg.selectAll(‘circle.pt’).data(tweets)
      .enter().append('circle')
      .attr('cx', function(t) { return xscale(xval(t)); })
      .attr('cy', function(t) { return yscale(yval(t)); })
      .attr('r', 3)
      .attr('class', 'pt');

    // add axes:
    var xaxis = d3.svg.axis().scale(xscale).orient('bottom'),
        yaxis = d3.svg.axis().scale(yscale).orient('left');

    svg.append('g')
        .attr('class','xaxis')
        .attr('transform', 'translate(0,'+(height-vmargin)+')')
        .call(xaxis);

    svg.append('g')
        .attr('class','yaxis')
        .attr('transform','translate('+3.0/4*vmargin+',0)')
       .call(yaxis);

}
```

scaling

drawing

draw axes!

The question you might ask yourself is - what if there are *no elements* yet to select?

This is where D3 and jQuery differ. D3 takes note of everything that doesn't exist yet and puts it into what it calls an *enter selection*. To get access to the enter selection, call the *enter()* method, which is what we do next.

Data-binding and enter/exit selections are fundamental to D3, and so please consult the documentation

2. For those elements in **tweets** that don't yet have a corresponding circle.pt element, create a new circle element and **append()** it.

3. Now, for all tweets' circles, set the 'cx' ("center-x") attribute of the SVG circle to the value returned by function(t) { return xscale(xval(t)); } for each tweet provided; note that xscale is the scale value we declared earlier and xval is the function that measures the number of tokens in each tweet.
Similarly, set 'cy' (center y) to the value to function(t) { return yscale(yval(t)); }

4. Set each circle's radius to 3, and its class to 'pt.'

The final section pertains to drawing the axes, using D3's axis convenience methods. The axes directly read the **scale** variables to identify where and how to draw the extrema. Since the x-axis needs to be at the bottom of the display, we use an SVG transform **translate** operation for that.

1e - View it in a browser, and switch to log-plot on the y-axis

Try running the example by refreshing the browser. If you avoided syntax errors, you should see a scatter plot! However, you may notice that a few tweets have gotten many retweets, while a majority have gotten very few. How do we make this more readable? Easy - try switching to a log scale:

```
yscale = d3.scale.log().range([height-vmargin,vmargin]);
```

and change yval to be:

```
var yval = function(tweet) {
      return tweet.retweet_count + 1; // why is this?
};
```

(Why did you need to change yval? What happens when retweet_count is 0?) Try refreshing; does it look more reaosnable this time?

1f - Adding basic interactivity - mouseover based selection

Immediately we notice that it would be fun to figure out which tweets correspond to each dot. Let's add a simple mouseover-display feature to show us the contents of each tweet.

First, let's add a place to display the tweet contents to our **index.html** :

```
<body>
  <div class="tweet-display">
        <div class="author"></div>
        <div class="text"></div>
  </div>
  <script src="bower_components/jQuery/dist/jquery.min.js" type="text/javascript"></script>
….
```

Now, let's handle a mouseover to populate those divs.

```
$('circle.pt').on('mouseover', function(evt) {
     var target = evt.target;
     $('.tweet-display .text').html(target.__data__.text);
     $('.tweet-display .author').html("@"+target.__data__.user.screen_name);
     $(target).attr('class', 'pt selected');
});
$('circle.pt').on('mouseout', function(evt) {
     var target = evt.target;
     $('.tweet-display .text, .tweet-display .author').html('');
     $(target).attr('class', 'pt');
});
```

The two event handlers simply watch for hover events over any of the points. The hovered-over DOM object (circle) is available as **target** on the first argument, which we named **evt**. To get access to the actual data object that D3 has joined to it, we use the **.__data__** field on the DOM object; which gives us access to the raw tweet that we joined against the object.

1g - Extending the Scatterplot

Realistically, you may want to add significantly more interactivity to the plot. How would you go about adding faceted browsing? For example, what if you wanted a range-slider which let you only filter for tweets that had a certain range of retweets? Or a text-search box to filter only for tweets that contained a particular token, aka **@justinbieber?**

Since D3 is data-driven, this should be easy to do. Think about what you need to do to the tweets array and how to re-render things once you make appropriate changes

If you wanted to appropriate the scatterplot for your research, what other kinds of interactivity might you want to support?

Part 2. Mention Forced-Directed Graph

Next, we're going to move on to relational data, starting with the same dataset we created for Part 1.

Our goal here is our first attempt to visualise at the social structure of tweets in our dataset by looking at the mentions relationships of the tweets. We will start with having tweet authors (and mentions as nodes), linked to the author of every tweet only if they were mentioned in at least one tweet by the author. (Note that there are many other ways you might construct this 'tweet graph' - for example, we might want the complete graph of co-occurring mentions - I'll let you figure out how to do that; should lead very easily from this example.)

In this example, we're going to use the fantastic force-layout engine that comes with D3, which is an implementation of an approximate-physics algorithm called Dywer-Jakobsen which avoids an n^2 computational explosion with little accuracy tradeoff. This makes it possible for us to do realistic physics in JS directly in the browser.

We are going to start a basic example by Mike Bostock (author of D3) :

```
http://bl.ocks.org/mbostock/2706022
```

2.1 - Getting started

Make a copy of the example (Open it in a new window to only get the relevant code) to start. Call it pt2.html and view it in a browser.

2.2 - Modifying it for Tweets

Based on the code in this example, we can see that the force directed engine takes in a list of **nodes** and **links** . For version, we're going to have **tweet authors** as nodes, and tweets mentions as links. That is, if author **@justinbieber** tweets at **@harrystyles** (or vice versa), there would be an link object between these nodes.

What if Justin tweets at Harry more than once? Let's also use the weight of the link to reflect that.

Looking carefully at our example, we realise we need our nodes and links to look like the following structures:

```
$.get('/bieber.json').then(function(tweets) {
   var nodes = {}, links = [];

    tweets.statuses.map(function(tweet) {

       var people = [tweet.user.screen_name].concat(
            tweet.entities.user_mentions
                 .map(function(mention) { return mention.screen_name ; }));

        // make nodes:
       people.map(function(p) { return nodes[p] || (nodes[p] = { name: p }); });

        // make links:
       people.slice(1).map(function(recipient) {
         var link = links.filter(function(l) {
            return l.source == tweet.user.screen_name && l.target == recipient;
         });
         if (link && link[0]) {
            link[0].weight += 1.0;
         } else {
           links.push({ source: tweet.user.screen_name, target:recipient, weight: 1.0 });
        }
       });
    });
    console.log('nodes >> ', nodes);
    console.log('links >> ', links);
    plot(nodes,links);
  }).fail(function(data) {
    console.error('error loading data ', data);
  });
```

As we see above, we first grab the screenname of the author from each tweet, and create a node object with **name** field for each.

Then, we create a link for every mention, incrementing the weight if it already existed.

Now, wrap the example force directed plotting code in a **plot** function, and try reloading.

2.3 - Drawing weighted links

To actually add the link weighting, let's modify the link line to reference stroke weight

```
var link = svg.selectAll(".link")
   .data(force.links())
   .enter().append("line")
   .attr("class", "link")
   .attr('style', function(d) { return 'stroke-width:' + d.weight + 'px;'; });
```

The last line added to the link statement references the weight parameter we created earlier when creating the link structures. Try re-rendering the dataset now.

2.4 - Next steps: Adding meaningful interactivity, ways to derive links

What sort of interactivity would you add to make this visualisation more useful? How would you add it?

Second, think about the method we used to create the graph. How might we improve this method to get a better picture of the social closeness of people tweeting?

Part 3. Critiquing the D3 Gallery

One of D3's strengths is the huge variety of code that has already been written for nearly every kind of visualisation we discussed. The D3 Gallery contains annotated examples that can be easily appropriated and used for your own purposes.

https://github.com/mbostock/d3/wiki/Gallery

Today, in lecture, we gave heuristics for reasoning about choosing effective visual dimensions for representing the types and dimensions of multivariate data.

Choose 5 examples from the D3 Gallery. For each example answer the following questions for yourself.

1. What kind of data does it support? If multivariate, how many dimensions, and what is the type of each dimension?

2. What kind of interactions does it support?

3. Based on #2, what kinds of tasks is this visualisation appropriate for?

4. Critique the visualisation's ability to convey information. How effective is it at using shape, colour, texture, line width? What is the balance of usefulness versus aesthetics? How easy is it for an expert to use?

5. Would you consider using this visualisation approach? Why or why not?