Showcase your skillset with an interactive colorful D3.js tag cloud

by Benny Bottema - Wednesday, March 02, 2016

http://www.bennybottema.com/2016/03/02/showcase-your-skillset-with-an-interactive-colorful-d3-js-tag-cloud/

What do you do when you want to show someone all your skills ever? You make a tag cloud ofcourse!

Here’s mine:

Now, mine works by trying to fit all words in the small space I give it and if it fails, it retries with a reduced size. This continues until everything fits. But that’s just my version though; you can configure your own to just draw what it can fit, or you can reduce words instead of size.

Also, check out the sample project I made for this

Contents

Credits where credit's due

To begin with, most of this work is not my own. I took it from Jason Davies, who was inspired by Wordle tag clouds and then used the awesome D3.js framework to make a feature library for this. I just took this working example, cleaned it up and completely changed the configuration with my own formula's for word placement, size, zoomfactor and how skills can be represented based on years of experience and relevancy.

aksahyPrabs's version

Jason's version

So how do we call this script? You call it's cloud() function and provide the box size, a list of words to lay out (text / size per word), allowed rotations and finally which font to use:

First define the words and their properties

```
var skillsToDraw = [
 { text: 'javascript', size: 80 },
 { text: 'D3.js', size: 30 },
 { text: 'coffeescript', size: 50 },
 { text: 'shaving sheep', size: 50 },
 { text: 'AngularJS', size: 60 },
 { text: 'Ruby', size: 60 },
 { text: 'ECMAScript', size: 30 },
 { text: 'Actionscript', size: 20 },
 { text: 'Linux', size: 40 },
 { text: 'C++', size: 40 },
 { text: 'C#', size: 50 },
 { text: 'JAVA', size: 76 }
];
```

This just defines a hard set of words to draw in a cloud, where I approximated the numbers based on how much I want to flaunt a skill. We'll get to a more advanced version for displaying your skillset later where it is based on actual years of experience and a relevancy factor.

Just so you know, you can actually use this library completely serverside using nodejs, but we are doing this in the browser.

Then invoke the cloud() script to calculate a layout

```
d3.layout.cloud()
 .size([width, height])
 .words(skillsToDraw)
 .rotate(function() {
  return ~~(Math.random() * 2) * 90;
 })
 .font("Impact")
 .fontSize(function(d) {
  return d.size;
 })
 .start();
```

The size determines the canvas size that will be inserted to your container div. The one in the top of this

post is 600px wide by 200px high.

The rotation is a function that returns a random angle in steps of 90 degrees (0 * 90 or 1 * 90, randomly). The weird ~~ operator is a peculiar one, it is a speed optimized replacement for Math.floor().

Draw the word cloud

Finally, an event handler is needed for when the script is done calculating stuff and is gathered data is ready to be drawn using D3.js graphics library:

```
 ...
 .on("end", drawSkillCloud)
 .start();

// apply D3.js drawing API
function drawSkillCloud(words) {
 d3.select("#cloud").append("svg")
  .attr("width", width)
  .attr("height", height)
  .append("g")
  .attr("transform", "translate(" + ~~(width / 2) + "," + ~~(height /
2) + ")")
  .selectAll("text")
  .data(words)
  .enter().append("text")
  .style("font-size", function(d) {
   return d.size + "px";
  })
  .style("-webkit-touch-callout", "none")
  .style("-webkit-user-select", "none")
  .style("-khtml-user-select", "none")
  .style("-moz-user-select", "none")
  .style("-ms-user-select", "none")
  .style("user-select", "none")
  .style("cursor", "default")
  .style("font-family", "Impact")
  .style("fill", function(d, i) {
   return fill(i);
  })
  .attr("text-anchor", "middle")
  .attr("transform", function(d) {
   return "translate(" + [d.x, d.y] + ")rotate(" + d.rotate + ")";
  })
  .text(function(d) {
   return d.text;
  });
```

```
}
```

What we have so far

Problems with the standard approach

Often not all tags fit inside the cloud. What people often is filtering the list of words before creating a cloud with it. Jason's layout library actually sorts on size first, starts by placing the biggest words first and add more words as they become smaller. If it fails to find enough space, it is skipped.

So one optimization has been applied already, but it's not very smart yet. For example, the library will try to place a word in one angle only. So if it might fit in another angle, tough luck. So what can we do to compensate? Well, without changing the library itself it becomes a bit tricky; I implemented a retry mechanism that reduces the size of every word with each retry cycle, until everything definitely fits inside the given space.

Here's the retry mechanism:

```
var MAX_TRIES = 5;

generateSkillCloud();

function generateSkillCloud(retryCycle) {
 d3.layout.cloud()
  .size([width, height])
  .words(skillsToDraw)
  .rotate(function() {
   return ~~(Math.random() * 2) * 90;
  })
  .font("Impact")
```

```
  .fontSize(function(d) {
   // reduce size of every words based on the current retry cycle
   return d.size * ((MAX_TRIES - retryCycle) / MAX_TRIES);
  })
  //.on("end", drawSkillCloud)
```

Each time the layout library was unable to fit everything, we retry and make everything a little bit smaller. The layout library itself will try different angles as well each time.

Finally, let’s integrate our skillset in the cloud!

Alright, we now have a way of generating word clouds, we know how to influence word angles and sizes and we have a way to make sure everything fits. Now for the math to calculate word size based on years of experience and relevancy and while we’re at it, let’s make the differences between sizes more pronounced, or else everything will still equally important.

So let’s start with the skills again:

```
// example list of skills, years of experience and relevancy (possibly
 inflated for more visibility)
var skills = [
 { name: 'javascript', years: 10, relevancy: .6 }, // relevant, but no
t as much as individual frameworks
 { name: 'D3.js', years: 1, relevancy: 1 }, // baseline importance
 { name: 'coffeescript', years: 1, relevancy: .3 },
 { name: 'shaving sheep', years: 1, relevancy: 1.5 }, // very importan
t skill obviously
 { name: 'AngularJS', years: 4, relevancy: 1 },
 { name: 'Ruby', years: 1, relevancy: .5 },
 { name: 'ECMAScript', years: 7, relevancy: .2 },  // not very importa
nt
 { name: 'Actionscript', years: 10, relevancy: .2 },
 { name: 'Linux', years: 10, relevancy: .5 },
 { name: 'C++', years: 1, relevancy: .2 },
 { name: 'C#', years: 1, relevancy: .2 },
 { name: 'JAVA', years: 10, relevancy: .4 },
 { name: 'REST', years: 2, relevancy: 1.2 } // my interviewer will loo
k for this, let's boost it's presence
];
// normally you would a lot more skills, so let's fill up the cloud a
bit artificially
var skillsToDraw = skills.concat(skills).concat(skills);
```

Ok, awesome. Now for the trick to get them into the skill cloud in the right size, we need to do three things:

1. Convert the skills to layout objects (with a text and size property, as before)
2. Calculate the size based on years and relevancy
3. Apply an exponential to expand the difference between item sizes

Convert the skills to layout objects

I've used the awesome Lodash library to transform all the items, but you can do it any other way:

```
// convert skill objects into cloud layout objects
function transformToCloudLayoutObjects(skills, retryCycle) {
 return _.map(skills, function(skill) {
  return {
   text: skill.name.toLowerCase() + ':' + skill.years + 'y',
   size: toFontSize(skill.years, skill.relevancy, retryCycle)
  };
 });
}
```

Calculate the size based on years, relevancy and retries

We have two scales between which we have to translate: a minimum to maximum fontsize scale (18 – 35 seems to work nicely) and a minimum to maximum years of experience scale, based on the entire set of skills. So the item with the least amount of years of experience should have the smallest font size, while the skill with the most years of experience should be the smallest. At the same time, the relevancy factor needs to be taken into account:

Here's the math for that:

$$linearSize(y, relevancy) = fontMin + \\ left(\\ fracy - yMinyMax - yMin * (fontMax - fontMin) \\ right) * relevancy$$

Now expand the differences between small and large font sizes to create nice effect that highlights the relevant skills that you excel at:

$$polarizedSize(linearSize) = left(fraclineairSize8 right)^3$$

And finally take into account the retry mechanism to reduce the lot in case there is not enough space:

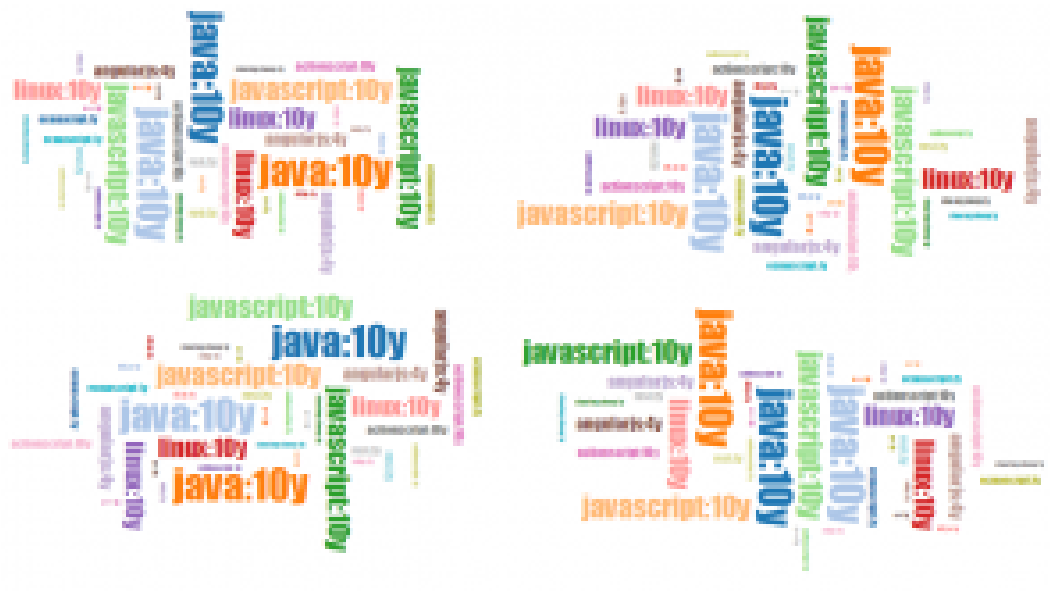
$$reducedSize(polarizedSize, retryCycle) = polarizedSize * left(fractriesMax - retryCycletriesMax right)$$

This is how it looks in code:

```
// again use awesome Lodash to get min/max years of experience
var minyears = _.min(_.map(skills, 'years'));
var maxyears = _.max(_.map(skills, 'years'));

// these are 'magical' numbers, but they seem to work nicely
var minfont = 18;
var maxfont = 35;

// let's make a wide cloud
var width = 600;
var height = 300;

function toFontSize(years, relevancy, retryCycle) {
 // translate years scale to font size scale and apply relevancy facto
r
 var lineairSize = (((years - minyears) / (maxyears - minyears)) * (ma
xfont - minfont) * relevancy) + minfont;
 // make the difference between small sizes and bigger sizes more pron
ounced for effect
 var polarizedSize = Math.pow(lineairSize / 8, 3);
 // reduce the size as the retry cycles ramp up (due to too many words
 in too small space)
 var reduceSize = polarizedSize * ((MAX_TRIES - retryCycle) / MAX_TRIE
S);
 return ~~reduceSize; // get rid of decimals and return result
}
```

Confirm it looks awesome

And this is the result!



Looks pretty good, right?

The only differences with the cloud in the top of this page is the angles used. Till now we have used two random possible angles: 0 degrees and 90 degrees (~(Math.random() * 2) * 90), but as the cloud gets bigger it becomes repetitive very quickly. Here's the version I used in the top version: (Math.random() * 6 - 2.5) * 30. This results in the following possible angles: -75, -45, -15, 15, 45, 75.

Make the cloud searchable

The last step is pretty straightforward. We take everything we already have, add an input and on a keyup event remove the old cloud and add the new one. Filter the skills based on the input and voilà, magic.

```
function generateSkillCloud() {
 var skillsToDraw = transformToCloudLayoutObjects(filterSkills(skills)
, retryCycle);

 // filter skills based on user input (again, using Lodash)
 function filterSkills(skills) {
  var textfilter = document.getElementById('filter').value;
  return _.filter(skills, function(skill) {
   return !textfilter || skill.name.toLowerCase().indexOf(textfilter.t
oLowerCase()) >= 0;
  });
 }

 // ... the rest you know now
}
```

Checkout the demo code

That’s it. Thanks for reading along with me so far and don’t forget the checkout the working sample code:

https://github.com/bbottema/d3-tag-skills-cloud

If you have suggestions or found a bug, leave a comment or send me an email using the form below.
