CUSP-GX-6006.001: Data Visualization

SPRING 2018

Lab 7 – End-to-End Data Visualization

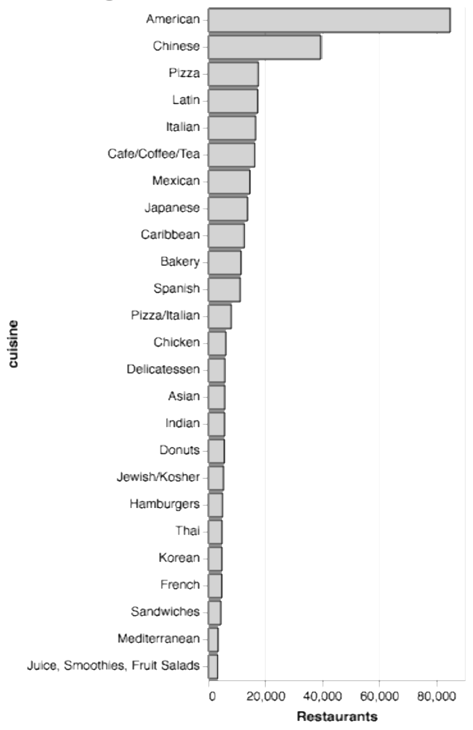
Part 1 – User Interaction

Due Date: 11:59PM on Apr 12, 2018

OBJECTIVES: this is the first part of the two-part lab that aims to give an overview of the entire data visualization pipeline. We will first touch on adding user interface for our visualization on the front-end (this part), and then, how to connect them to a data back-end that supports dynamic analytic results (part two). In the first part, we will look at the following:

- The differences of using a high-level grammar library, e.g. *vega-lite*, versus a low-level library like D3.
- How to build a high-level pipeline in Python with *Altair*.
- Using React.js, a user-interface library, to add inputs and interactions to a web-visualization.

1. Vega-lite



♦ **Motivation:** We have been using D3.js and JavaScript to build our visualization so far. Though powerful, using D3 to visualize data can be quite tedious. Users have to get it right in both the design and the implementation (aka. writing JavaScript code) to produce a desired visualization. This leads to the development of many high-level visualization libraries such as vega-lite.js. With vega-lite, users can specify the description of a visualization using the terminologies of visualization design, e.g. marks, channels, and encodings, etc. All of these can be specified in a data-exchange format, JSON, thus, it is less worrisome about the syntax of JavaScript coding.

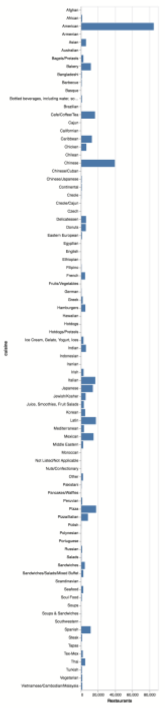
In this task, we will produce one of cuisine plot in HW2 (shown on the left) using vega-lite. Before proceed, please take some time to go through the introduction slides from one of the library's authors. Please also take a look at their online tutorials for additional details.

♦ **Setup:** For this task, we will be using the Vega editor to walk through the example. Please open your browser to: https://vega.github.io/editor/#/custom/vega-lite

The left side of the editor is the Vega-lite description in JSON. The right side is the visualization that reflects the specification on the left. We start with this simple snippet to build a bar plot from the cuisine data set.

NOTE: you can copy and paste this into the editor panel.

```
{
  "$schema": "https://vega.github.io/schema/vega-lite/v2.json",
  "data": {
    "url": "https://raw.githubusercontent.com/hvo/datasets/master/nyc_restaurants_by_cuisine.json",
    "format": { "type": "json" }
  },
  "mark": {
    "type": "bar"
  },
  "encoding": {
    "y": {
      "field": "cuisine",
      "type": "ordinal"
    },
    "x": {
      "field": "total",
      "type": "quantitative",
      "axis": {
      "title": "Restaurants"
      }
    }
  }
}
```



♦ **Vega-lite Pipeline:** From the code above, you should see a very long bar chart, similar to the tiny image on the bottom right. This is already very close to the visualization that we want, and with quite little efforts:

1. Set the language specification
2. Set the data source
3. Set the marks
4. Set the encoding, and channels

In particular, we specify the "language" to be parsed by the editor using the schema command. Here, we're using vega-lite version 2:

```
  "$schema": "https://vega.github.io/schema/vega-lite/v2.json",
```

Then, we set the data source, and tell Vega that it is in JSON format.

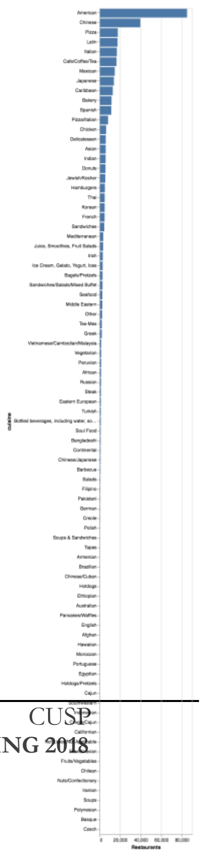
```
"data": {
  "url": "https://raw.githubusercontent.com/hvo/datasets/master/nyc_restaurants_by_cuisine.json",
  "format": { "type": "json" }
},
```

The mark can then be specified through the “mark” property. For a bart chart, we use “bar” as the mark.

```
"mark": {
  "type": "bar"
},
```

Finally, we set the encodings using the “encoding” property. Note that, the channels are implicitly interpreted by Vega based on the mark type. In this case, since we’re using the “bar” marks, we’re expected to use the position channel. Hence, we only need set the “x” and “y” property of the encoding for it to work. In each of the encoding, we need to specify the field, and type, and whether we need an axis label.

```
"encoding": {
  "y": {
    "field": "cuisine",
    "type": "ordinal",
  },
  "x": {
    "field": "total",
    "type": "quantitative",
    "axis": {
      "title": "Restaurants"
    }
  }
}
```

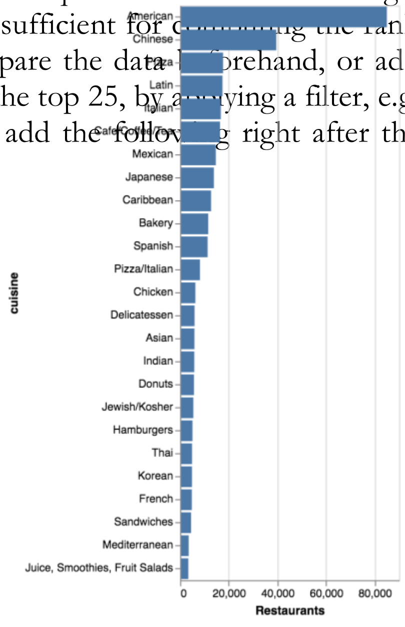


♦ **Ordering Data:** However, our bar plot is sorted by the ordinal field by default. We would like to sort them by the “total” field instead. In this case, we could add a sort description to the “y” field, makes it become the following:

```
  "y": {
    "field": "cuisine",
    "type": "ordinal",
    "sort": {"field": "total", "op": "argmax"}
  },
```

Our plot should now be sorted by the "total" field. Note that "argmax" is an operation to be applied on group of "cuisine". This is needed since there is no guarantees that our data only contain unique cuisine names. For example, if there are multiple "American" entries, then, the "argmax" operation will select the one with the maximum "total" across these operations. We should now get the plot on the right.

♦ **Data Transformation:** Our plot is still showing too many entries. We would like to shorten the list of cuisines to only the top 25 entries. A natural way of doing this is to sort the cuisine by total, and pick the top one. Though this could be done in Vega (the superset library of vega-lie), it requires some tricks with vega-lite, unfortunately. This is because the operations that are supported is not sufficient for computing the rank (using the total), and then filter the top 25. In practice, we can either prepare the data beforehand, or add additional fields that can be used for filtering. In our case, we can compute the top 25, by applying a filter, e.g. saying that only selecting cuisines with 3,300 or more restaurants. Please add the following right after the "data" section, and before the "mark":



```
...
 "data": {
   "url":
"https://raw.githubusercontent.com/hvo/datasets/master/nyc_restaurants_by_cui
sine.json",
   "format": { "type": "json" }
 },
 "transform": [
   {"filter": {"field": "total", "range": [3300,null]}}
 ],
 "mark": {
   "type": "bar",
 },
...
```

In the "range" value, "null" indicates Infinity since we do not have a cap on the upper limit. The expected result is shown on the right. It's worth noting again that the flexibility in data transformation is one of the limitations of vega-lite, and/or other high-level visualization libraries. Though this works, it requires knowing the value 3,300 beforehand. In the Selection section, we'll look at a workaround for this issue.

♦ **Styling:** In this part, to stay true to the original visualization, we will color all bars as Light Grey, and adding the highlight capability that changes the bar color to Steel Blue when being hovered.

First, we add a "color" channel right at the beginning of the encoding to make all bars become gray:

```
...
 "encoding": {
   "color": {
     "value": "LightGrey"
   },
   "y": {
...
```

We also need to add a black border like we had in the homework. This is part of the styling process. With D3, we can use CSS. For vega-lite, we can either define a custom "style" property for mark or simply adding basic style properties, such as "stroke" to the mark definition itself:

```
...
"mark": {
  "type": "bar"
},
...
```

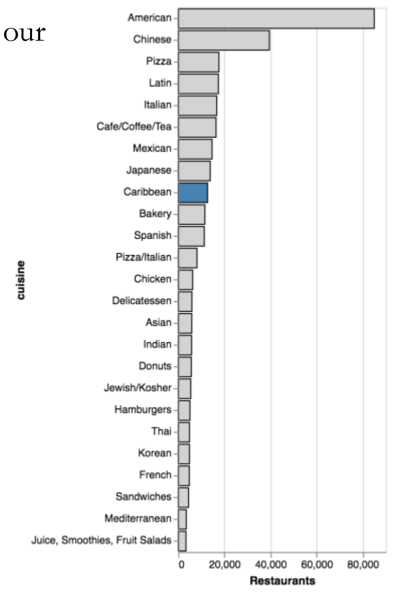
```
...
"mark": {
    "type": "bar",
    "stroke": "black"
 },
...
```

♦ **Selection:** By default, vega-lite already includes highlighting feature. However, if we hover the mouse over an entry, it only shows a tooltip showing detailed info about the datum, fields associated with the record. You can try this with the existing code. In order to change the color when the mouse hover a mark, we need to add selections into our visualization. More info on selections in vega-lite can be found here. We would like our selection "type" to be "single" (selecting one mark at a time), triggered "on" the event of "mouseover", and with an "empty" selection, "none" will be highlighted (vega-lite also allows "all" to be selected with an empty selection). Let's add the following between the definition of "transform" and "mark":

```
...
 "transform": [
   {"filter": {"field": "total", "range": [3300,100000]}}
 ],
 "selection": {
   "highlight": {
     "type": "single",
     "on": "mouseover",
     "empty": "none"
   }
 },
 "mark": {
...
```

The selection has been named "highlight", which can be used to style our mark's color encoding, by adding the following to the "color" definition:

```
...
    "color": {
      "condition": {
        "selection": "highlight",
        "value": "SteelBlue"
      },
      "value": "LightGrey"
    },
...
```



Now you should see the visualization similar to the one on the right.

♦ **Workaround for top-25 filter:** as mentioned before, it's impractical to provide a value to filter out data outside of the top 25 values. This could be done if we can have a "row id" for each record, and only keep the row with ids not larger than 25 after sorted. Since vega-lite doesn't provide a row id, this cannot be achieved. However, once we add selections to our system, if you hover over a mark, you will notice that there's now a new field called "_vgsid_", meaning vega selection id. This is indeed equivalent to the row id field that we're looking for. Thus, now we can change our filter to filter out all "_vgsid_" larger than 25 to keep only the top 25 items. Please change your filter to the following, and verify that it's still working.

Changing:

```
{"filter": {"field": "total", "range": [3300,null]}}
```

to either

```
{"filter": {"field": "_vgsid_", "range": [null, 25]}}
```

or simply:

```
{"filter": "datum._vgsid_<=25"}
```

♦ **User Interface:** so far, we have built a static visualization with vega-lite. Next, we look at building user interface for interactions. Let's say we would like to build a search box that allows users to enter partial text of

a cuisine name, and we then highlight all the matching bars. For example, if a user types “izza”, both the “Pizza”, and “Pizza/Italian” entries will be highlighted.
With vega-lite, selections are the basic building blocks for interactions. To achieve this, we need to create a selection that binds to a text input, and conditionally set our color based on this text. First, add a new selection, named “search”, as follows:

```
...
 "transform": [
  {"filter": "datum._vgsid_<=25"}
 ],
 "selection": {
  "search": {
   "bind": {
    "input": "input"
   },
   "empty": "none",
   "on": "mouseover",
   "fields": ["term"],
   "type": "single"
  },
  "highlight": {
   "type": "single",
   "on": "mouseover",
   "empty": "none"
  }
...
```

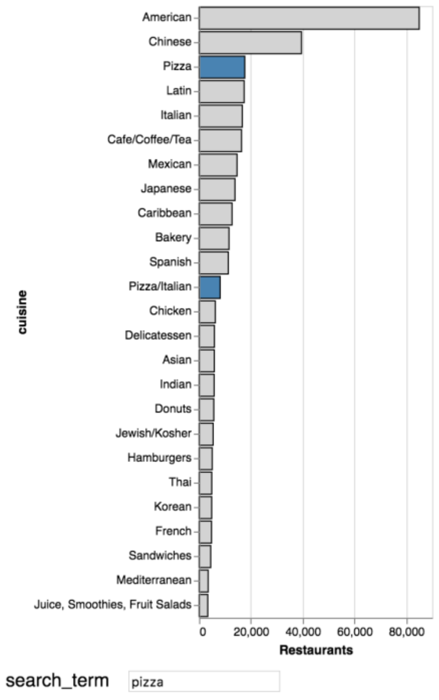
After this, there should be a text input at the bottom of the visualization. This text can be referenced as “search.term” as the combination of the selection name, and the “fields” that we define. Now, in order to highlight the right word. We need to change the condition of the color encoding to reflect this logic. Our test expression is:

```
"(indexof(lower(datum.cuisine), lower(search.term))>=0) || (highlight._vgsid_==datum._vgsid_)"
```

This means that we convert the cuisine name for each entry to lower cases, then saying that it’s a match if we could find the typed text (also in lower case) in the cuisine name. Note that, we still allow mouse hovering highlight, however, this will invalidate our search term whenever it occurs.

```
...
  "color": {
    "condition": {
      "selection": "highlight",
      "value": "SteelBlue"
    },
    "value": "LightGrey"
  },
...
```

```
...
  "color": {
    "condition": {
      "test": "(indexof(lower(datum.cuisine),
lower(search.term))>=0) ||
(highlight._vgsid_==datum._vgsid_)
",
      "value": "SteelBlue"
    },
    "value": "LightGrey"
  },
...
```



Your output now should look like the one on the right. Another note is that "indexof" and "lower" are special functions for vega expressions. Vega supports quite a few of these functions, however, they are simply not as extensive as in JavaScript or Python. This is a trade-off in flexibility when using a high-level grammar, e.g. vega-lite, versus a low-level one like D3.js.

♦ **Export:** similar to our JS Bin labs, we can export vega-lite visualizations to a GIST, and make it accessible publicly. This can be done by extracting the JSON that we build with vega editor, and vega-embed library. Here are the steps to have your visualization published on bl.ocks.org:

1. Using your GitHub account to log in to GitHubGist, and create a "New Gist".

2. Add a new file named "index.html" as follows (using the vega-embed template on the link above). Note that this assumes our vega-lite JSON will be stored as "DV_Lab7.vg.json" in the same Gist:

```
<!DOCTYPE html>
<html>
<head>
  <script src="https://cdn.jsdelivr.net/npm/vega@3"></script>
  <script src="https://cdn.jsdelivr.net/npm/vega-lite@2"></script>
  <script src="https://cdn.jsdelivr.net/npm/vega-embed@3"></script>
```

```
</head>
<body>
<div id="vis"></div>
<script type="text/javascript">
vegaEmbed('#vis', "DV_Lab7.vg.json");
</script>
</body>
</html>
```

3. Add a new file named "DV_Lab7.vg.json" with the JSON contents copied from your vega editor.

4. Click on create GIST (either public or secret is okay).

5. After that, open your GIST, and copy the URL, e.g., mine is:
   https://**gist.github.com**/hvo/8febd426e3d12bd430aedc0cd8dd1d41

6. Replace "gist.github.com" with "bl.ocks.org". In my case, the new URL is:
   https://**bl.ocks.org**/hvo/8febd426e3d12bd430aedc0cd8dd1d41

7. Try to open the link to verify that your visualization is correctly display.
   TURN IN: Please turn in your bl.ocks.org link as part of the lab submission.

2. Altair: Vega-lite in Python

An advantage of high-level visualization library such as vega-lite is that it is not locked into any specific programming languages like JavaScript, or Python. In fact, because it is a data exchange standard, i.e. JSON, for describing visualization designs, it can easily be used in multiple language. In the above, we already see that it Vega-lite can be shown through a website. In this part, we will visit Altair, an open-source framework that allows users to describe Vega-lite visualization pipeline in Python.

♦ **Install Altair:** please follow Altair's installation instructions to setup your Python environment. Be sure to try their example in your notebook. If you run into issues, below are some useful troubleshooting step:

- Update your conda/jupyter notebook to the latest version before installing Altair
- If you cannot see the chart output in your notebook, verify that the package "vega3" is already installed through pip. You can then manually enable the package with your notebook by:

```
jupyter nbextension enable vega3 --py --sys-prefix
```

- If you still cannot see your chart (but a JSON description) when running the example, and have already tried their troubleshooting guide, ***the last resource*** is to build a virtual environment for Altair. You still have to register the extension in this case. For example, with conda, run in a terminal:

```
$ conda create -n vis
$ source activate vis
$ pip install notebook vega3 altair==2.0.0rc2 ipykernel
$ $(dirname ${CONDA_EXE})/pip install vega3
$ jupyter nbextension install --sys-prefix --py vega3
$ jupyter nbextension enable vega3 --py --sys-prefix
$ ipython kernel install --user --name=vis
```

After this, you should be able to open a kernel "vis" in your notebook and run Altair.

♦ **Getting hands-on with Altair:** our goal is to re-create the final visualization in Task 1 using Altair and Python. For this task, please go over the Altair's notebook that is accompanied with this lab, named **DV_Lab7_Altair.ipynb**.

TURN IN: Please turn in your bl.ocks.org link as specified in the notebook.

You are encouraged to look at the examples for both vega-lite and Altair to learn more about them.

3. React.JS: adding user interface (UI) to your web visualization

Putting your visualization on the web is a popular and quick way to present your results for a broad audience target. In this case, UI is needed, particularly for interactive visualization, to get user feedback and allow users to drive the visualization. Visualizations generated from high-level libraries such as vega-lite and Altair can be embedded to a web page and with certain UI capabilities. Examples are the search text box in our example, or other type of inputs that can be bound to a selection. However, these can be limited in cases where your application requires complex interactions that need to operate not only on the visualization, e.g. control the layout of your application, highlight with a custom animation, linking to other libraries, or perform additional data filtering steps, etc. There is often a need for an independent UI component in a web application that efficiently manage all UI operations. One such library is React.JS.

In this part, we will look at how to use React.JS to add the keyword search feature to our HW2 solution. The behavior of the cuisine highlight will be identical to our previous two tasks. The original idea was to update the map based on the selection as well, but we're leaving that part to a later session. You are not required to turn in anything for this last task. However, I strongly suggest you to read through my commented JS Bin, and the Quick Start for React.JS. I expect these will be very helpful not only for your final projects, but also for your capstones (if you have a web deployment component).

Again, please go through these two set of notes:

HW 2 Solution++	: http://jsbin.com/ficepeq/edit?js
React's Quick Start	: https://reactjs.org/docs/hello-world.html

(**please remember to click Next** at the end, and at least go through "Hello World", "Introducing JSX", "Rendering Elements", and "Components and Props").