CSC 468 Lab 5

We will work on this in class, including a demonstration. The part at the end labeled “Assignment” must be completed and turned in to the submission box on D2L (due date marked on the box).

In this lab, we will connect a D3 visualization to a python server. First, we’ll revisit the way we have been serving data to these visualizations all quarter, meaning we will finally get around wrapping csv files in JS code. Next, we’ll use a slightly more sophisticated back-end to run clustering on our data and show the result on the front-end.

Resources

I took the scatterplot we’re using from: http://bl.ocks.org/weiglemc/6185069

The book DVwP&JS covers much of this material in Chapter 12. If you are interested in using an actual database, e.g. with SQL queries or via MongoDB, you can learn about how to connect those in Chapter 13.

We need a python environment and I recommend Anaconda, which is free, has convenient installers, package management and environment management: https://www.anaconda.com/download/ . Note that python underwent an interesting revolution some years back when it transitioned from major version 2 to 3. They broke backwards compatibility, so people still use 2 and that is why it is offered. With Anaconda, you can make multiple environments with different versions of python if you need, so let’s start with version 3.

The Flask library can be installed using the python package manger included with Anaconda: http://flask.pocoo.org/docs/0.12/installation/ . They recommend making an environment, which is a good idea, but we will use the basic approach of installing the library globally.

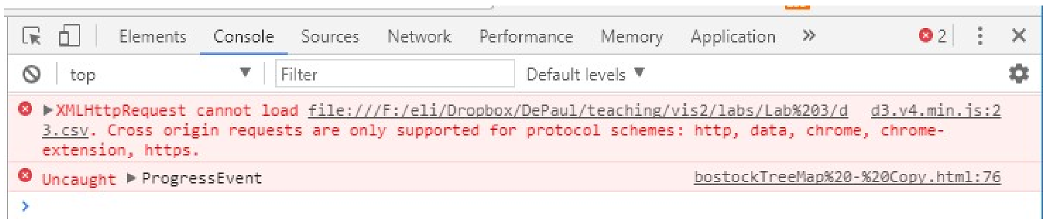
This is a quick getting-started tutorial on Flask: http://flask.pocoo.org/docs/0.12/quickstart/

And here is a more complete one: https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world

As usual, the Lab 5 folder includes the starting point and data. I have modified and extended an example scatterplot so that it is modular and we can adjust the data and mappings after it is loaded. These were substantial modifications, so you should take a look and make sure you understand that code. I will go over it in the lab session. The final result of this lab is in the *app* folder, so you can check your work against it and test that it works with your installation of python.

Loading Data

The first thing we will use our back-end for is loading data. So far we have been wrapping csv data in JS files, which is not efficient or elegant. The reason we do this came up in Lab 3 – if you run normal example D3 code from an example you find, normally the data is loaded with a call like *d3.csv*. That is the function that asks a server for a csv file. But if you just have a file open in your browser, there is no server to connect to:



```
▶XMLHttpRequest cannot load file:///F:/eli/Dropbox/DePaul/teaching/vis2/labs/Lab%203/d   d3.v4.min.js:2
3.csv. Cross origin requests are only supported for protocol schemes: http, data, chrome, chrome-
extension, https.
Uncaught ▶ProgressEvent                                      bostockTreeMap%20-%20Copy.html:76
```

Let's create a server. Remember that while 'server' is usually used to refer to some abstract computer that grants access to some web/cloud resource, its core functionality is down to some software running on a computer somewhere that receive requests through a network and responds to them. Python has the ability to create such functionality very quickly. Open up a prompt (hit the Window key and type *cmd* and hit enter) and then type the following command:

```
python -m http.server
```

or if you are using python 2

```
python -m SimpleHTTPServer
```

Now you can open http://localhost:8000/ in your browser. The ':8000' is the port number. The browser sends a request to your own server, and the server replies. The simple server just serves files from the folder in which it is run. This functionality itself can be useful, but it is not going to fulfill all our needs. The basic page shows a directory listing. Click *scatterplot.html* and you'll see the path in the browser change and note that we're actually serving up our page now. Also look at the console window and you will see log messages confirming that the page is being loaded piece by piece. If you're not seeing what you expect, make sure you ran the python command *in the same directory as the lab code.* If you need help with navigating directories at a prompt, look up the *cd* command and talk to me for further resources.

As noted in Lab 3:
in modern web pages and web apps, it is common practice to load data and content after the web page has loaded. This is done by sending another HTTP request to the server ('another' meaning besides the one the browser sent to get the web page itself). This technology is generally called AJAX and typically the mechanism is the XMLHttpRequest (XHR) mentioned in the error message above. This is happening because in the code, they use d3.csv to get the data.

We no longer need the work-around. Let's change the code back to using *d3.csv.* Remove the line that defines the data variable and then wrap the code that creates the scatterplot in a call to *d3.csv*. Now what we're doing is asking the server for a csv file and when it comes back, we create the scatterplot.

```
d3.csv("cereal.csv", function(error, data) {
    // if the request failed, throw exception
    if (error) throw error;

    SP = ScatterplotVis(svg, data);
    SP.drawScatterplot();
});
```

Pretty cool! Note: if you find yourself puzzled by an error, look at the source code in the browser. Browsers like Chrome use caching (storing a local copy) to improve performance, but sometimes they don't update as often as they should. In Chrome, for example, you can get past this with the Ctrl + F5 keyboard shortcut, which reloads everything from scratch, including JS files that get loaded by the page. Also make sure to save your files in the editor before you expect to see the result when reloading in the browser. It sounds like a stupid reminder, but everyone does it once in a while.

Upgrading the Server

We have this code running on our own server, but the server is pretty basic. It doesn't give us control over what files we serve and doesn't let us do anything besides serve files. Let's install the python library Flask, which we'll use as a server. The resources list includes some more extensive instructions, but we'll get by typing *pip install Flask* in our console window. When that completes, we have the ability to *import flask* in python. Then, let's make a directory for the Flask 'app' version of this by typing *mkdir app* in the lab 5 directory and then creating a file called *server.py*:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

This will run a Flask server and define one specific *route.* The route specifies a path where the server will respond to requests. It won't respond to other paths (except with a 404 error).

Run the server by typing on the command prompt. We'll replace the *export* with *set* because we're using Windows. Our file is *server.py* as opposed to *hello.py* as well. Also note the $ is the prompt; don't copy that.

```
$ export FLASK_APP=hello.py
$ python -m flask run
 * Running on http://127.0.0.1:5000/
```

Direct your browser to localhost:5000 and smile!

Of course, one route that gets us a little bit of raw text won't be that helpful. To really use Flask, you'll want to copy their directory structure to organize elements like JavaScript and CSS separately. Since we don't have time to cover the full power of Flask, I'll leave you to the tutorials and use a very simple version to get us to the next step. First copy all the files we need for our scatterplot page to the app directory, then change the app code to this:

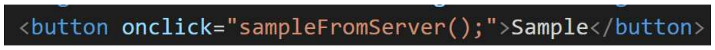
```
from flask import Flask, send_from_directory
app = Flask(__name__)

@app.route('/<path:path>')
def startup(path):
    return send_from_directory('.', path)
```

There are a few notable changes. The import statement includes a new function used to send a static file from some location (*send_from_directory).* The route now has a special component inside <> marks. This says that the item following the / must be a valid file path for this route to activate, and that the filename that is the path should be bound to the variable *path.* Then, the function that runs in response to this route uses the *path* variable to get access to the path that was part of the request URL. We use it simply to send the file. In your browser, open *localhost:5000/scatterplot.html* now and everything should be back in order. Look at both the JavaScript console and the console window where the server is run to see error messages.

Make the Server do Something

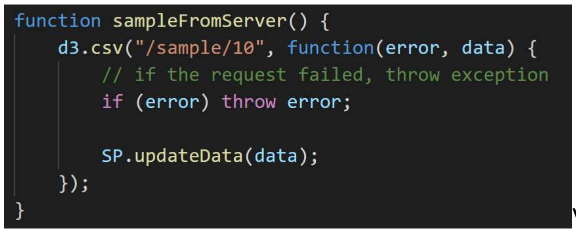
Let's actually use the fact that we're running a python server to get data. As a simple example, we'll make a button that gets a new sample of the data. That's not necessary in this case, but it shows off what we can do. When there is a python representation of the data and we ask for the data as a remote call to the server, we can use python to manipulate and send.

Add a button in the html file, just below the svg:



```
<button onclick="sampleFromServer();">Sample</button>
```

When it's clicked, it will run a function that updates the data, which we'll add to the script tag in the html file:



```
function sampleFromServer() {
    d3.csv("/sample/10", function(error, data) {
        // if the request failed, throw exception
        if (error) throw error;

        SP.updateData(data);
    });
}
```

\

Of course, that does nothing right now – we need related functionality on the server, so let's add this code up near the top of the server.py file (note that this is not an image and can be copy-pasted):

```
# get the data into memory
# (this could be done in a separate module)
import csv
import random
reader = csv.reader(open("cereal.csv"))
header = reader.__next__()
stringData = [ row for row in reader ]

# return a sample set of rows of the cereal data
def dataSample(numSamples):
    random.shuffle(stringData)
    return stringData[:numSamples]

# take a list of data rows and make CSV text
def dataToCSVStr(header, dataList):
    csvStr = ",".join(header) + "\n"
    strData = [ ",".join([str(x) for x in data])
                for data in dataList ]
    csvStr += "\n".join(strData)

    return csvStr
```

First, we use python's csv reading functionality to parse the data file. The *header* is the first row (column labels) and *stringData* holds the actual data. The usage of __*next*__ followed by what looks like a list comprehension that does nothing is to load the data out of the iterator that gets lines of the file and into an in-memory list copy. Note that each data line is represented by a list of strings as opposed to floating point numbers (hence the variable name). **The two variables we have created will stay in scope in this file, making them available in-memory to use in any request to the server.**

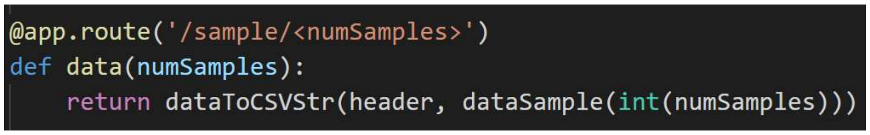
The *dataSample* function can be used to sample the data by choosing a subset of the data rows. It shuffles the data row and then returns a portion of the appropriate size (see python list slices, NB: this will not throw an error if there isn't enough data, it will just return all it has).

When we get a request for a sample from the front-end, the response needs to look like the data did in the first place, i.e. the format in which the front-end first got the data. We have a list of data file lines and we need to stick the header back at the front. That will be done by *dataToCSVStr.* It builds up the string csvStr and then returns it. The first piece is the header and the *join* function turns the list of strings into one string with the original list elements separated by commas (look this function up if you don't know it). The next part does the same thing for the data lines. It is a nested list comprehension and if you don't know about python list comprehensions I strongly recommend learning about them; they're a fantastic tool. Note that the nesting is to take care of the fact that the elements of the data list might be numbers instead of strings. This isn't the case now but it will be after our next modification. If we weren't converting from strings, this would be sufficient:

```
[ ",".join(data) for data in dataList ]
```

The *strData* variable now has a list of lines of our output csv file, so the last step is to join those together with '\n', putting a newline between them. We accomplish that with another join. All in all, *dataToCSVStr* will take a header, and data in nested list format and turn it into a csv formatted string.

Here is how we use our new functionality to respond to the front-end:



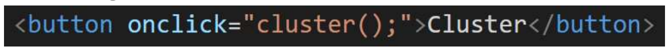
```
@app.route('/sample/<numSamples>')
def data(numSamples):
    return dataToCSVStr(header, dataSample(int(numSamples)))
```

The route definition includes a variable input, *numSamples*. The function we provide for processing can use that as a parameter. When the route is activated by a request to the server, whatever is in that part of the URL gets bound to that variable. All we have to do is take a subset of the data and then put it into the csv format we need to export. Fortunately, that's exactly what the functions we just discussed do.

Rerun the server or see the Flask startup guide and run in debug mode so that it restarts when you edit the source files. Then reload the web page in your browser.

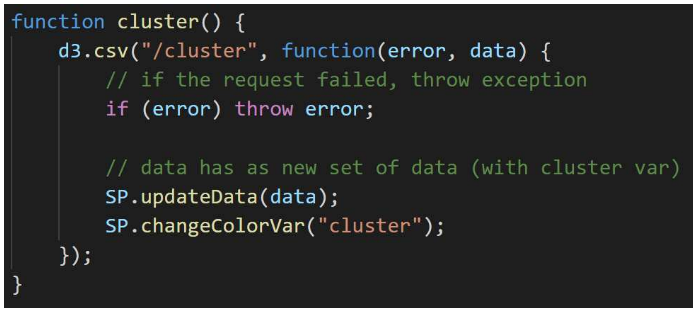
Connect Some Machine Learning

One more thing – let's leverage some more of what python can give us and do a clustering in the background that gets applied on the front-end as a recoloring. First let's create the front-end bit. Add a button again:



```
<button onclick="cluster();">Cluster</button>
```

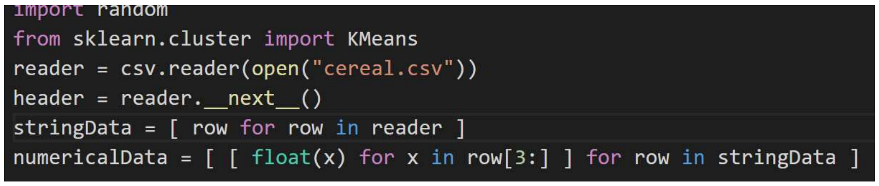
Now we'll code what that button will do from the front-end.



```
function cluster() {
    d3.csv("/cluster", function(error, data) {
        // if the request failed, throw exception
        if (error) throw error;

        // data has as new set of data (with cluster var)
        SP.updateData(data);
        SP.changeColorVar("cluster");
    });
}
```

We will call a yet-to-be-written back-end function via the "/cluster" URL. The *d3.csv* function will make a request to the internet to load the specified URL, wait for a response, and then call the function we provide with the response. If there is an error, we simply throw it as an exception. On success, we get data parsed as csv. We update the scatterplot's data and then tell the scatterplot to use the "cluster" variable to color the data points. Make note for when we code the back-end that it is going to need to add a variable to the data with the cluster assignment and then build a csv formatted string with this variable included.

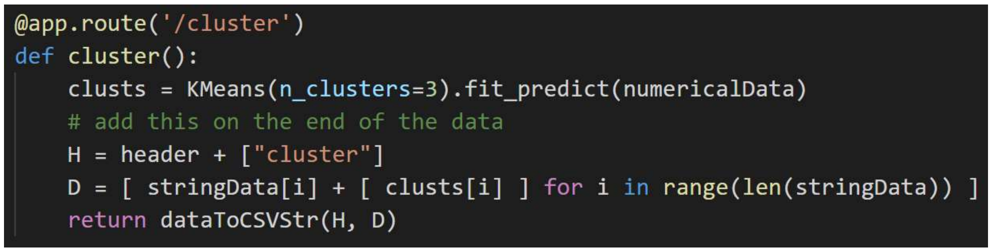
The last remaining step is to actually build that back end. First we will revisit the top of *server.py.* To do clustering, we will use the popular python machine learning library *sklearn.* If you don't have this library, then the first thing to do is install it. Fortunately, this is easy with python's package manager that's included with Anaconda. Type *pip install sklearn* at the command prompt. When the library is installed we can reference it to get the KMeans clustering routine.



```
import random
from sklearn.cluster import KMeans
reader = csv.reader(open("cereal.csv"))
header = reader.__next__()
stringData = [ row for row in reader ]
numericalData = [ [ float(x) for x in row[3:] ] for row in stringData ]
```

The only other change is the last line in the above snippet – we need a numerical version of the data to feed to clustering. It will not cluster vectors like ['2', '3'] and ['2.2', '5']. The conversion code is tailor made for this data, relying on the fact that the first three columns are not numerical variables, but the rest are. It uses nested list comprehension, like the code for *dataToCSVString*. It is not difficult to write a more complete, general version of this code that figures out which variables are categorical and which are numerical. I'd recommend doing that, but actually you can get this functionality from some other libraries now, like Pandas, which is written about in the DVwP&JS book.

The data are ready, time to cluster. We'll make a call to the KMeans algorithm right in the route that responds to "/cluster".



```
@app.route('/cluster')
def cluster():
    clusts = KMeans(n_clusters=3).fit_predict(numericalData)
    # add this on the end of the data
    H = header + ["cluster"]
    D = [ stringData[i] + [ clusts[i] ] for i in range(len(stringData)) ]
    return dataToCSVStr(H, D)
```

The first line gets us the clustering result. That consists of a list of N numbers, each number in the range [0 to (n_clusters – 1)], where N is the number of data rows. Each number in the list represents the cluster id that a given data point is assigned by the clustering algorithm. For more on this, read up on clustering and try this out on some sample data or just try an sklearn tutorial.

The front-end needs a csv file of the data where the cluster id is just one more column. The rest of the code does that. Note that adding two lists in python concatenates them, so `[a] + [b] => [a, b]`. We use this to add “cluster” onto the list of column headers. We use the same trick to add the actual cluster ids onto the end of the list of original data values per row. The list comprehension for *D* replaces each row of data with one that has the appropriate cluster assignment tacked on. Finally, we use *dataToCSVStr* to format the resulting data lists into a csv file. Good thing we made sure it could work with numerical data!

For future reference, note that if you return a python dictionary it maps to JSON data on the JavaScript end, and you can use *d3.json* to retrieve it.

Assignment

The assignment component is again straightforward – take one of your previous lab assignment and make it work from a flask server.