**Module II-Mapping with Leaflet Cont.**

**Part I:**

In the last module we learned how to use the geoJSON file format with our maps. Although geoJSON is a very common file format, one of the most common formats of data is still the CSV. So how would we go about incorporating a CSV file into an interactive web map? Great question! Lucky for us, Leaflet has a prepared library called “omnivore”, which is very good at digesting not only CSV, but many other data types! What does this mean? You guessed it, you are going to have to set up a link to this library! To do this, open up your index.html code and add the following code in the head:

<script src='<https://api.mapbox.com/mapbox.js/plugins/leaflet-omnivore/v0.2.0/leaflet-omnivore.min.js>'></script>

Awesome! Now we can get into the good stuff. Included with this module is a data set called CheckCashing.csv, if you open it up you will see that it contains data about some check cashing locations. The data we are interested in here are the latitude and longitudes (of course, we are making maps!), but we are also interested in the data about the amount cashed at each location, and the total number of customers at each location. Ideally we would like to represent both of these values at once, so lets make it our goal to create a map that has circles representing each check cashing location that are colored by the total amount cashed and which are scaled in size by the total number of customers!

To start off lets open up our java.js file, you will notice we already have the map tiles (and their attribution!) set up. Right below that we can create two new functions, one to create a color scale and one to create a size (radius) scale based on a locations check cashing amounts and their number of customers:

function fillColor(d) {

return d > 500000 ? '#006d2c' :

d > 250000 ? '#31a354' :

d > 100000 ? '#74c476' :

d > 50000 ? '#a1d99b' :

d > 10000 ? '#c7e9c0' :

'#edf8e9';

}

function radius(d) {

return d > 9000 ? 20 :

d > 7000 ? 12 :

d > 5000 ? 8 :

d > 3000 ? 6 :

d > 1000 ? 4 :

2 ;

}

Notice that we got the break points for these functions by examining our data and identifying the proper ranges for both the check cashing amounts, and the total number of customers. Ok, now with those two functions in place we can set up a function to actually apply them! Lets add this code in:

var checkCashingStyle = function (feature, latlng){

var checkCashingMarker = L.circleMarker(latlng, {

stroke: false,

fillColor: fillColor(feature.properties.amount),

fillOpacity: 1,

radius: radius(feature.properties.customers)

});

return checkCashingMarker;

}

Above you will see that the function is telling leaflet to use the circle marker to display our data, and to define the color and the size of each marker by calling the two functions (“fillColor” and “radius”) we built earlier! Great! But now we want the user experience to include more than just varying colors and sizes of circles, we need to spruce this up a bit. Lets construct a large function that will handle how the user interacts with the data. Under your last function add this code:

var checkCashingInteraction = function(feature,layer){

You will notice that this line just begins a new function called “checkCashingInteraction” but does not close it. We will build this function out one piece at a time and then close it when we are finished! Down to business then, first lets build three styles for when a user mouses over a data point, clicks on a data point, or is not on a data point at all:

var highlight = {

stroke: true,

color: '#ffffff',

weight: 3,

opacity: 1,

};

var clickHighlight = {

stroke: true,

color: '#f0ff00',

weight: 3,

opacity: 1,

};

var noHighlight = {

stroke: false,

};

Great! Above you can see that the first style (called “highlight”) creates a border or stroke around the circle that will be solid white. The second style is for when a user clicks on a data point, which will then turn the border yellow, and finally the last style just removes the border which we will use when someone is not mousing over or clicking on a data point. Ok now lets set up the interactions that will use these styles. First we can set up the interaction for when someone mouses over a data point, add this code now:

layer.on('mouseover', function(e) {

layer.setStyle(highlight);

if (!L.Browser.ie && !L.Browser.opera) {

layer.bringToFront();

}

});

Here we have told Leaflet to fire off our “highlight” style anytime there is a mouse over event. Note that the “e” in the function is standard notation representing what ever data has been “”moused-over”. Also notice that in this function we have told the data point to “bringToFront”, however we are telling it to do this as long as the browser is not Internet explorer or Opera, which have some buggy issues. Next we can add the function for when there is no mouse over taking place at a data point, add this code now:

layer.on('mouseout', function(e) {

layer.setStyle(noHighlight);

});

Nice and simple! This function simply fires off our “noHighlight” style anytime a data point is not being moused over. But what about when someone clicks a data point? Sure we want to tell it to go to our highlighted style, but we also want our pop-up to appear! To do this we can add the following code:

layer.on("click",function(e){

layer.bindPopup('<div class="popupStyle"><h3>' + feature.properties.name + '</h3><p>'+ feature.properties.address + '<br /><strong>Amount:</strong> $' + feature.properties.amount + '<br /><strong>Customers:</strong> ' + feature.properties.customers + '</p></div>').openPopup();

layer.setStyle(clickHighlight);

});

Ok, lets go through the code above. First we are looking for anytime someone clicks on a data point, if that even occurs we want the following thins to happen, first we want to bind a pop-up which gives the details about our check cashing location, including the name of the property, the address, the total amount, and the total number of customers. Second we want to fire off our style “clickHighlight”, so the border of the circle that was clicked is highlighted yellow! Great! Now we just need to add this closing bracket, which will close our big “checkCashingInteraction” function!:

}

I know what your thinking, we built all these fancy functions for the user interactions, but what about the CSV?? We were supposed to find a way to read the CSV data in right? Of course! Lets start by first creating a “shell” L.geojson function that will apply all our new fancy styles to any data that gets passed through. Add this code below your last function:

var checkCashingCustomStuff = L.geoJson(null, {

pointToLayer: checkCashingStyle,

onEachFeature: checkCashingInteraction

});

And with that in place we can finally use our new omnivore library to read in our CSV file and convert it to a geoJSON file, which will then get passed to the shell we just created! To do this just add the following code:

var checkCashingLayer = omnivore.csv('CheckCashing.csv', null, checkCashingCustomStuff).addTo(map);

Great! Lets check it out, open the map in your browser to view your handy work!

**Part II**

In part I we were able to create a fantastic map using CSV data and some cool stylizations. However it would be a lot better for the user (and good map making practice) to include a legend to help folks make sense of everything! Lets build one! To start we need to allocate some space on our map for the legend itself, luckly leaflet has some built-ins to help us. Go back to your java.js file and add this code:

var legend = L.control({position: 'bottomright'});

Here we have allocated the bottom-right-hand side of the map for a “control”, which is where we will place our legend. As it turns out there is a Leaflet function for adding a legend which is just what we need! First we start the function by adding the following code:

legend.onAdd = function (map) {

Next we use another Leaflet built-in that allow us to create a “div” element. Add the following code:

var div = L.DomUtil.create('div', 'legend'),

This creates a new div element we are giving the ID ‘legend’. Next we will create two vectors containing the ranges of data we will need by adding these lines of code:

amounts = [0, 10000, 50000, 100000, 250000, 500000],

customers = [0, 1000, 3000, 5000, 7000, 9000];

Great! Now lets first work on the amounts of checks cashed in our legend. We will start with the label by adding this line of code:

div.innerHTML += '<p><strong>Amounts</strong></p>';

That simply adds some bold text to the div element that was just created. Then we can use a loop to generate the fill colors we need for the legend:

for (var i = 0; i < amounts.length; i++) {

div.innerHTML +=

'<i style="background:' + fillColor(amounts[i] + 1) + '"></i> ' +

amounts[i] + (amounts[i + 1] ? '&ndash;' + amounts[i + 1] + '<br />' : '+<br />');

}

Lets break that loop down. We can see we are looping through the length of the vector “amounts” that we made previously. During every iteration of the loop we creating a new html section of code for the corresponding amount fill color, the actual label representing the lower amount of the fill color range, followed by a dash, then followed by the high range of the fill color range! Great! Now lets do the same for the total number of customers:

//+= is the add into the HTML

div.innerHTML += '<p><strong>Customers</strong></p>';

//Loop through the customers array. For each within the array set the border radius based on the larger part of the range (+1 makes sure that it is within the right size bucket in the radius function at the top).

//Within the div add

for (var i = 0; i < customers.length; i++) {

var borderRadius = radius(customers[i] + 1);

var widthHeight = borderRadius \* 2;

div.innerHTML +=

'<i class="circle" style="width:' + widthHeight + 'px; height:' + widthHeight + 'px; -webkit-border-radius:' + borderRadius + 'px; -moz-border-radius:' + borderRadius + 'px; border-radius:' + borderRadius + 'px;"></i> ' +

customers[i] + (customers[i + 1] ? '&ndash;' + customers[i + 1] + '<br />' : '+<br />');

}

Ok, we have generated all the symbols, colors, and labels that we need for our legend, whats next? Oh snap! Whats next is close the function!! Dont forget that all of this code was sitting inside the legend.on.Add function and we need to close the function for it to work! Lets make sure we return the new “div” before we close:

return div;

};

And of course we need to add this to our map by adding the following line:

legend.addTo(map);

That was a lot of work! But we are still not done, we need to take care of some last details, specifically we need to create some new CSS styling for the div elements that are created by that previous loop! To do this crack open your stylesheet.css code and add the following:

.legend {

line-height: 18px;

padding: 6px 8px;

background: white;

background: rgba(255,255,255,0.8);

box-shadow: 0 0 15px rgba(0,0,0,0.2);

border-radius: 5px;

}

This bit of CSS sets up the style of the legend container that will be created. Awseome! But what about the contents of the legend? No worries! Lets take care of that with the following code:

.legend i {

width: 18px;

height: 18px;

float: left;

margin-right: 8px;

opacity: 0.7;

}

Here we are making sure all the i’th html elements generated in our loops from the previous steps are given the proper style. However if you look at our last loop we gave the i’th elements that were generated a class of “circle”, which we need to define. So lets add the following code to our CSS:

.circle {

background-color: #909090;

}

Great! That was simple enough! Do you know what time that makes it? Time to check out your handy work! Refresh your web browser to see your new legend!