Chapter 13: Part 3

**Deep Learning**

**SOM: How it Works**

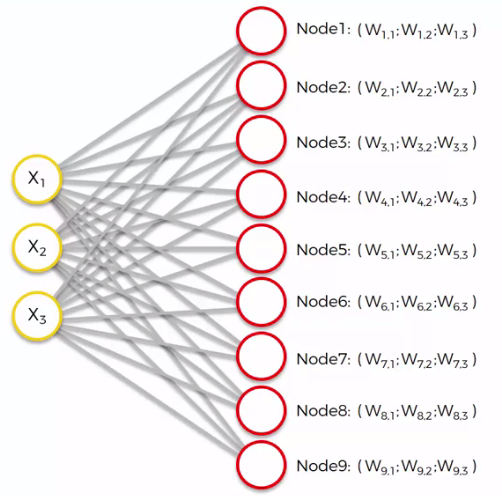
**13.3.1 How SOMs work**

|  |  |
| --- | --- |
| In this section we want to find out how SOMs learn.   * Here we've got a very simple example of a SOM. We've got three features (3 columns) in our input vector, and we've got *nine* *nodes* in the output. Here *each* *node* represents a *data-point* (a *single* *row*). * Don't let this representation confuse your understanding of SOM. Here we have *3 columns of features* and we might have *thousands* of *rows* (each row has 3 columns, represents a data-point). * We map these rows (data-points) in a 2D map and Group (cluster) them using SOM. * It means that our *input data set* is actually *three dimensional* (we reduce this dimension), whereas our *output data set* in a *SOM* is always a *two-dimensional map*, and therefore we are reducing the dimensionality from *3D* to *2D*. |  |
| * Now we're going to turn this SOM into an Network representation. Right-hand image shows us what it would look like. * It is the *same network* (as *above*), the only difference is how we've *positioned* the *nodes*. * We still have the *same* *amount* of *connections*, *inputs* and *outputs*, it's just the *visual* *representation* has changed so that it would be easier for us to understand what's going on. * Also note that, SOMs are different than NN. * First of all, SOMs are much easier than other NN-supervised-techniques. The whole concept behind them is very simple and straightforward. * Secondly it's also important to note that SOMs are different than ANN/CNN/RNN. * The concepts that might have the same names/terms (such as *weights* and *synapses*) have different meanings so don’t get confused with ANN/CNN/RNN terms. * Just be careful when we're talking about things like *weights* and *synapses* and other things, those are different than *ANN/CNN/RNN*. |  |
| * Let's consider the *top* *node* of our *output-nodes*. Notice the three synapses connecting it to our three input-nodes. * We saw in ANN, *weights* were used to multiply the *value* of a node, we *added them up*, and then we *applied* an *activation* *function*. * Weights in SOMs: The weights in SOMs are different and there is *no activation function*. Here weights are *characteristics* of the *node* *itself*. * In *SOM* the weights are act as *coordinates*. For example, , and as an *input* *vector*, in 3D input space. * There are three weights for those 3 synapses , and . Now   *First* *index* means that it's the first *output-node*. *Second* *index* means the *input nodes*.   * Here the first node trying to fit in our *input* *space*. |  |

|  |  |
| --- | --- |
| * So in *twenty-dimensional input space*, there are 20 columns in your inputs. Then each output-node would have 20 weights and act as a *20-dimensional-vector* in the *20D-input-space*. * Basically just think of these *output* *nodes*, each one of them is a *imaginary* *data* *point* in our *input* *space*. * Same thing applied to 2nd , 3rd output-nodes and all other output-nodes. * So, each one of the nodes, in our case **nine** (there could be many more), has its own weights. At the start of the algorithm, those weights are randomly selected a small value near ***0***. * Each one of these nodes has its own imaginary place in the input space. |  |

|  |  |
| --- | --- |
|  |  |

* After assigning all weights, finally we get Following.

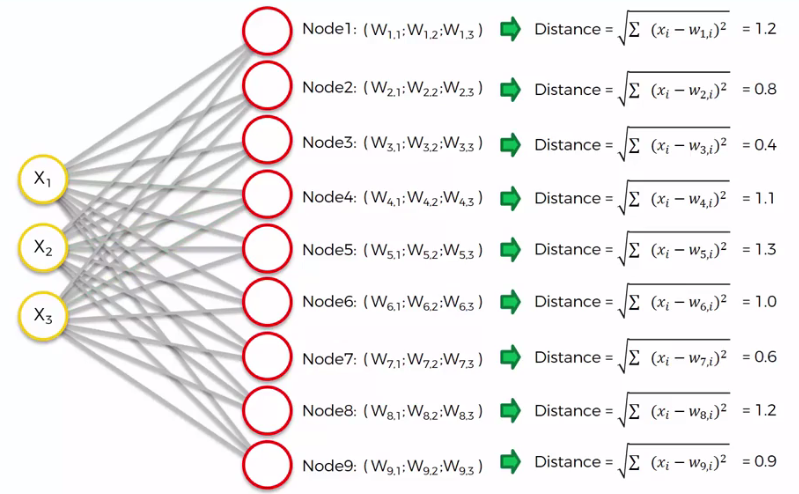


**13.3.2 How SOMs organize itself: Distance calculation**

Now we go through each of *rows* of our *data* *set*, and we're going to find out which of the weightassigned *output-nodes* (from SOM's first iteration) is closest to each of our rows (inputs) in our data set.

* Let's start with row number one:
* We first put the first row's *3 values* into our *3 input nodes*.
* After we've inputted *first* *row* from our *data* *set* into our *input* *nodes*. We'll go through all 9 output-nodes and calculate the distance from the *first* *row* in *input* *nodes*.

|  |  |
| --- | --- |
|  |  |



* Remember, here inside *input-nodes* is the inputs from *first-row* of our *Data-set*. This inside *input-nodes* does not changes while we are calculating distance from SOMs output-nodes. We're not changing the row.

|  |  |
| --- | --- |
| * We're going to go through every single one of these nodes, and find out which of these is the *closest* in that *original* *input* *space*, which of these nodes is *closest* to our *first-row*. * We calculate the *distance* as a *Euclidean* *distance*. * Feature scaling: Also note that we should get a value close to 1. Because our inputs are between 0 and 1, to make this algorithm work properly. * So we need to apply *normalization* or *standardization* to our data-set. Then we input the *data* into the *SOM*. |  |

* After calculating distance for all 9-output-nodes, we can see that *first-row* or *first-input* in our data-set is *three times closer* to *3rd output-node* than *1st output-node*.
* Best Matching Unit (BMU): Now we calculated all of the *distances* between *first-row (first-input)* and *9-output-nodes* then we found that the closest one is *3rd output-node*.
* Here *3rd output-node* is the BMU, or the best matching unit.

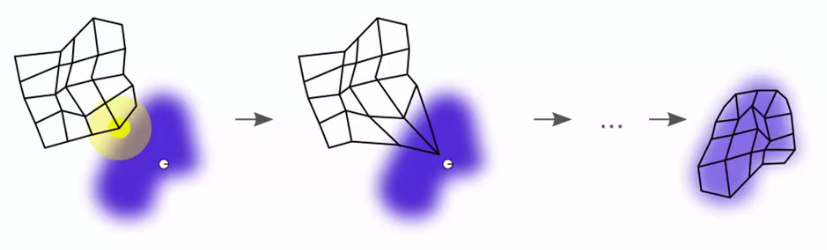
**13.3.3 SOM moves toward the Data-points**

Let's look at a larger SOM of 9x12 size (i.e. 108 output-nodes).

* Let's say in this larger map we found the BMU for first-row (the Green colored point).
* Next SOM is actually going to update the weights, according to BMU, so that the output-node gets even closer to our first-input (first row) in our ***data-set***.
* The reason we are updating the weights is because we *don't have control* of our *inputs*, the only thing that we can control in that formula are the *weights*.

|  |  |
| --- | --- |
|  |  |

* In simple words, the *SOM* (white structure/grids) is coming *closer* to that *data* *point* (*purple* colored *cloud*). In the following image we can see that the *SOM* is moving over the *data-point-cloud* (*Purple* colored).
* At the very start *SOM* is *initialized* with some *weights* and then it *updates* the *weights* and move over the *data-points*.



* Setting radius (how grouping happens): In the next step we set a radius around this BMU, and every single point (output-node) of our SOM that falls inside that *radius* is going to have its weight updated to come *closer* to that *row* that we *matched* *up* with.
* The *closer* to the *BMU*, the *heavier* are the *weights* being updated. So weights *near* *BMU* are going to be *updated* the *most*, weights *far* from *BMU* are going to be updated *less*.

|  |  |
| --- | --- |
|  |  |
| * Think it as the nodes are dragging each other. Whole structure near the BMU is slowly pulled towards the same direction. * The closer nodes are to this BMU, the harder they will get pulled towards that row (input) that the BMU matched up with.   So that's how the radius concept works. |  |

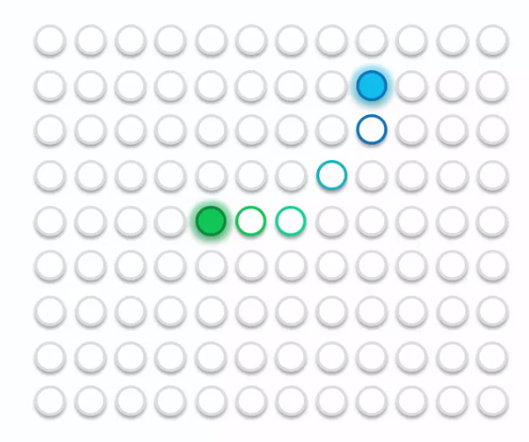
* How nodes fall into different groups: Now let's have a look at *2nd-row* (2nd-input). Let's say *2nd-row* has its BMU at the *Blue-Colored* node.
* Here again this BMU drags all the nodes that are close to it that are falls into BMU's radius.

|  |  |  |
| --- | --- | --- |
|  |  |  |

* So for above two BMUs the nodes closer to Green BMU falls in "Green-group" and nodes are closer to Blue BMU falls in "Blue-group". It's pretty simple.
* For example, a node far away from the green *BMU*, is close to the *blue* *BMU*, it is pulled much harder with the *blue* *BMU*, and therefore it becomes like the blue *BMU*.

So now we've got some idea about how this SOM works.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |



**13.3.4 How SOM updates itself**

* Here we have 5 BMUs. Now each these BMUs are going to be *updated*.
* Then each one of these *BMUs* is going to be *assigned* an *area* around it, and that area is going to be calculated through a *radius*:
* Here we can see that there's some values that don't fall under the radius, that usually doesn't happen in SOM this is just our visual example.
* Normally the radius at the start is selected as quite large so that it covers nearly the whole *self-organizing map*.

Here all of the nodes that fall into these areas are updated.

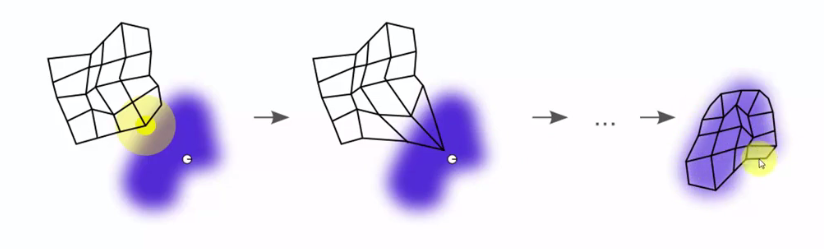
|  |  |
| --- | --- |
|  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

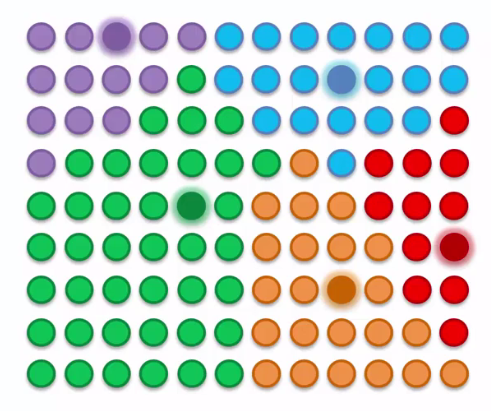
|  |  |
| --- | --- |
| * So *BMUs* dragged the *nearby* *nodes* closer and closer and there's a *competition* between them each node feels attraction for multiple BMUs. That's normal, that's what happens in the *SOM*. * BMUs radiuses shrink: One ***epoch*** completes after you go through *all of your rows* in your *dataset* and all of these *updates* happen. In a new epoch when you go through your rows again, a unique feature of the Kohonen Learning Algorithm is applied and makes all the BMUs *radiuses* *shrink*. * When the radiuses of all BMUs become a bit smaller, and this time when you're going through your dataset, your BMUs are pulling *less* *nodes* than previous epoch. * And again, the *radiuses* *shrink*. Then again *less* *nodes* are *pulled* towards the BMUs. The process becomes more and more *accurate* as epoch passed. |  |

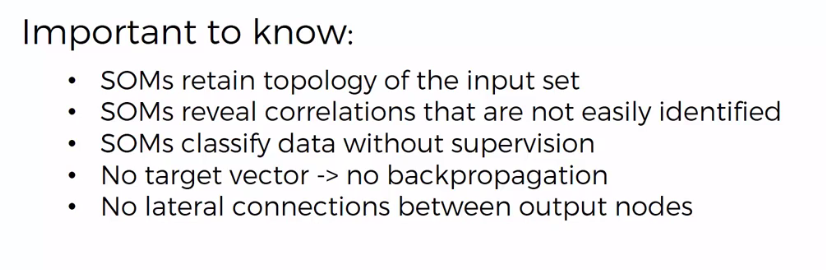
|  |  |  |
| --- | --- | --- |
|  |  |  |

* At the *very-start* you were just trying to *get* all of your *SOMs* very *close* to your *data*. Then as you go through more and more and more *epochs* you are adjusting your *SOM* in a more *precise* manner. As a result *SOM* slowly *goes over* your *data* and becomes kind of musk for your data, as we can see in this example down here.



* Then through lots and lots of epochs, our SOM might look something like this:





Note:

A couple of things that are important to know:

1. SOMs retain topology of your input set: The map is slowly become a musk of your data. Your data might have some topology, some interrelations in your data. SOM retain those.
2. SOMs actually reveal correlations between data that are not easily identified: Say you hundreds of columns in your dataset it can be very challenging to find kind of correlations or similarities that might be present in your dataset.

* A SOM can neatly analyze all that for you, and then put all of that for you into a map.

1. SOMs classify data without supervision: SOM is unsupervised Deep learning technique. SOMs don't need any labeled data.

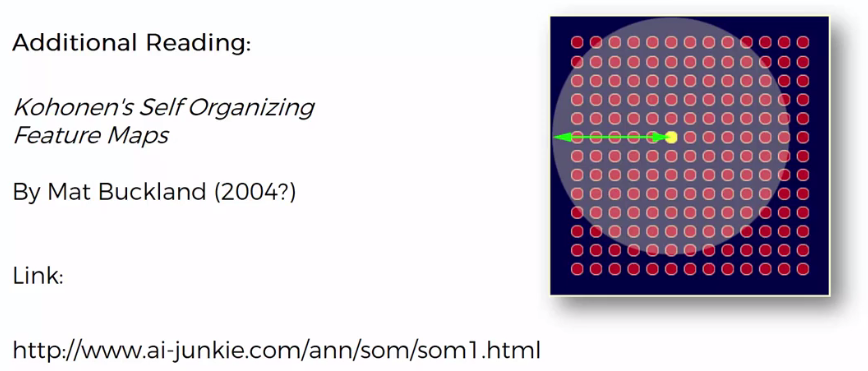
* It doesn't need any supervision, the SOM will, extract features on its own, show us features, dependencies, correlations, and similarities. They can be used in *scenarios* where you don't actually know *what* you're *looking* *for* but you want to find any kind of *correlations* in your data.

1. SOMs don't require a target vector: There is no backpropagation in the training of a SOM (unlike ANN). SOM doesn't have a target vector. There is no error to backpropagate.

* In *ANN* the *data* would go through the *NN*, we'd get a result we'd *compare* it to the *target* *vector*, we'd find the *error* and then we'd *backpropagate* that *error* through the *NN* to update the *weights*.

1. There is no lateral connections between output-nodes: We didn't actually need any connection between the nodes.

* The only thing that happens between the nodes is: when you pull on one node, the other close ones get pulled.
* When we say that ***there's no lateral connections between the output-nodes*** means that there's no actual NN type of connections there's no activation functions between them and so on.
* Sometimes you'll see images where the *output-nodes* are *connected*. There's a *grid* behind the *nodes*. They're just showing that these are output-nodes on a SOM. But there is *no* actual *formulas* or *equations* going on between those *output-nodes*.
* Additional reading: If you'd like to study with some soft introduction into mathematics behind self-organizing maps, about how the radius changes and how the weights are updated based on how close on the proximity to your best-matching unit. Read the blog " *Kohonen's Self Organizing Feature Maps*" by *Mat* *Buckland*.
* You also get some introduction to programming SOMs.
* Several useful blogs are available at ***ai-junkie.com***.

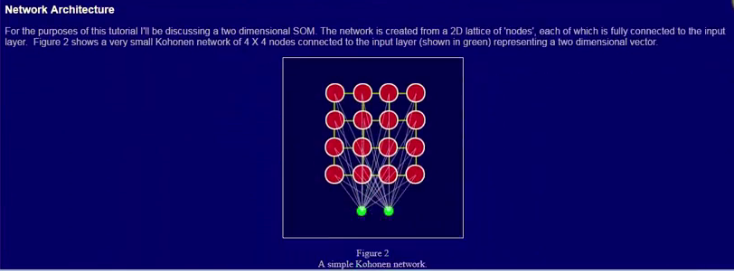


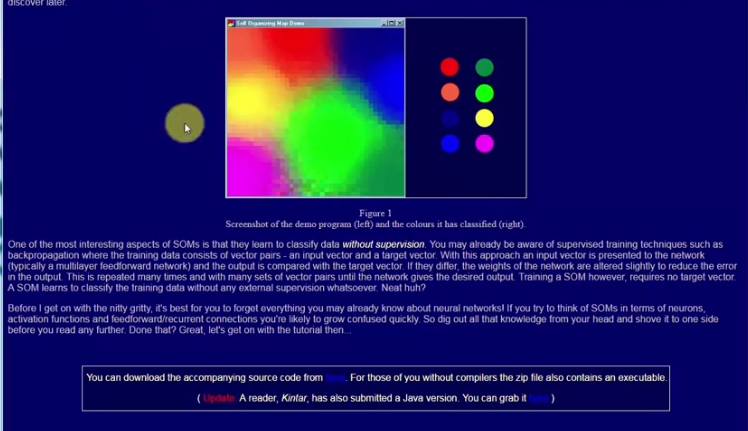
**13.3.5 Example: Live example of a SOM**

Now we are going to see a live example of a SOM. Here we use .exe (executable) file provided by AI junkie. ***ai-junkie.com*** is the blog that (we mentioned above section).

* In this blog you can get a brief introduction into *SOMs* and the *mathematics* behind them, and as well get some *programming* examples.
* You will be able to download source codes for the example we're going to be looking at.







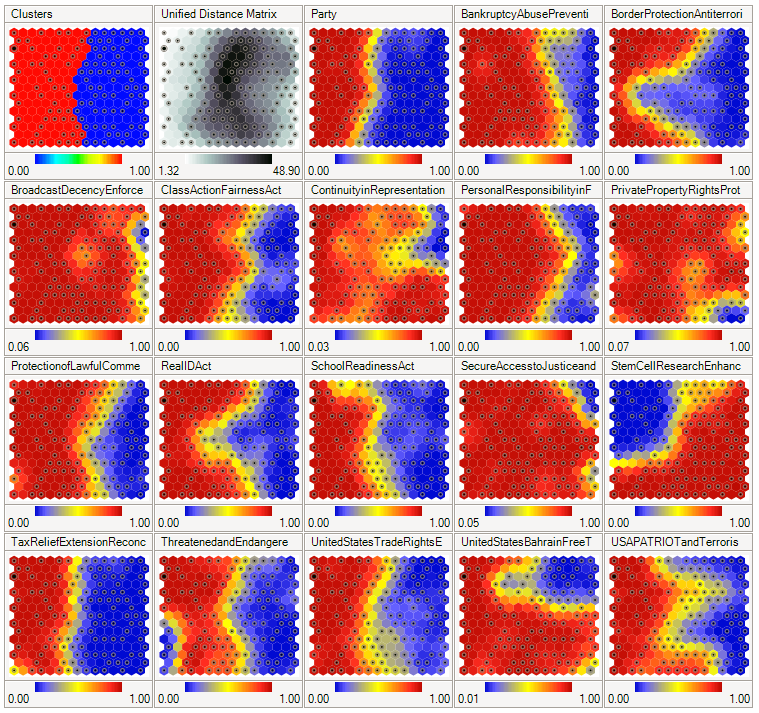
|  |  |
| --- | --- |
| * We're going to be creating a self-organizing map like this by running the provided .exe file. * Once we run it, this app will create a SOM, which has, as inputs, these eight colors: red, orange, dark blue, light blue, dark green, light green, yellow and magenta. * So, each of the *rows* in the data set, is *R-G-B* code for each of these *colors*. And those codes are actually *normalized* on scale on *range* **[0, 1]**. * The point is we have got eight rows on our dataset (8-colors) and each one of them has three columns (**R, G, B**) and we are going to put them into a SOM.   Once you run it, you will see that there is the number of iterations and then you can press R to retrain this SOM. |  |

|  |  |
| --- | --- |
|  |  |
|  |  |

* You will notice that it *organizes* itself every time *differently* that's because your *weights* are *initialized* at *random*.
* But the main point is: it's putting a ***dark blue*** next to the ***light*** ***blue;*** and the ***light*** ***green*** next to the ***dark*** ***green;***, and most of the time *magenta* *red*, *orange* together. That's what we mean that *SOM* does *preserve* *topology* or it does *find* and *preserve* *similarities* in your data set. So here SOM groups similar colors together.
* Once again, check out the website, here at AI junkie, and you will be able, get the codes here, you can actually get it in Java as well. And you can get some lovely examples of how all this works.
* You'll see that self-organizing maps are not that hard mathematics and to code.

|  |  |
| --- | --- |
|  |  |

**13.3.6 Example: Reading a SOM**



* Here we're going to have a look at how to read advanced SOM. You might come across a *SOM* that looks like *above*. There's actually a *lots* of *representations* inside this *SOM*. This is an example from *Wikipedia*, and is a *SOM* of *voting* results/ *patterns* in the *US* *Congress*.
* The *input* *data* for this map was like a *data* *sets* of *Members* of *Congress* in the *US* *Congress* like over *500* or *535* *Members*. Where each Member say about a certain question that they were voting on. Did they say yes/ no/absent from voting.
* And based on that information, the *SOM* group them which *Members* of *Congress* are *close*, are *similar* to each other, which are *dissimilar*, and place them onto a *map*.

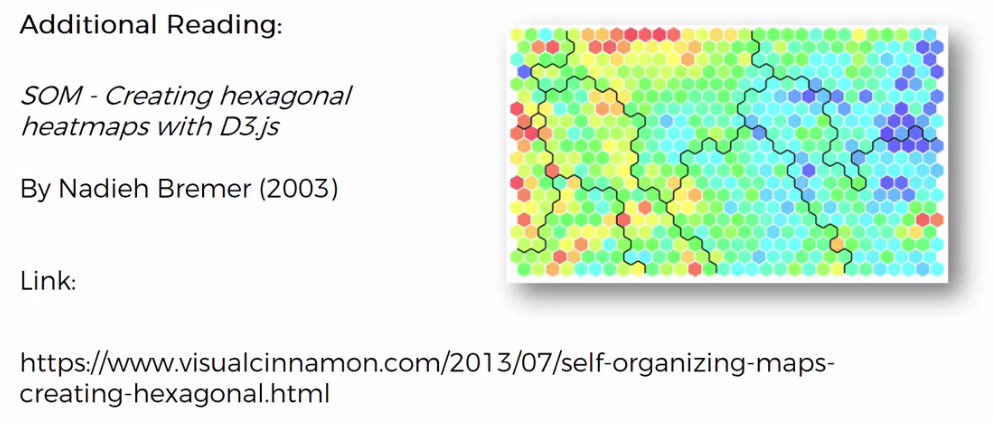
|  |  |
| --- | --- |
| * CLUSTERS: So here we've got a map of the *Members* of Congress, where *SOM* splits the data set into *two* *classes*. * This is the overall cluster, it is bit different. This is how the SOM, splits the Members of Parliament, just based on all of the things that they voted on all of the different questions (it is *not* *solely* *based* on Republican -Democratic). * Here red doesn't mean Republican or blue doesn't mean Democratic, just red and blue other two colors that are used to identify the two clusters. * This cluster is based on overall results on the voting about: 17 topics –***Bankruptcy Abuse Prevention, Border Protection Anti-terrorist, Broadcast Decency Enforce, Class Action Fairness Act, Continuity in Representation, Personal Responsibility inF, Private Property Rights Prot*** etc (these are the actual topics that being voted). |  |
| * PARTY: The actual *split* between the *parties* is modeled over here. So when you ask the SOM which Member belongs to which party, the answer is like this, so *red* here is *Republican* and *blue* is *Democratic* Party. * This is the clusters of members that belonging to either Democratic Party or the Republican Party. |  |
| * UNIFIED DISTANCE MATRIX: In the second representation, we see the *unified distance matrix*, is also called the U matrix and it shows is the distance between points/nodes on the SOM. * At the *middle* it's *darker*, means that these points are *further* *apart* from each other, *lighter*-points are *close* together. * It makes sense over here, because the *darker* *points* are exactly on that *ridge* (border between two clusters) and there's a lot of *dissimilarity*. |  |
| * Bankruptcy abuse prevention: in this case Republicans voted yes and maybe that is a corporate bankruptcy, rather than an individual purpose bankruptcy. * Here is an invisible line (in some SOM represented by a black line/border) is the border between Republican –Democratic. It can show us: that most of the Republicans according to SOM voted yes, some of the Democrats voted yes and other Democratic voters voted no. * red = yes, and blue = no. |  |
| * Border Protection, anti terrorism: Here you can see a very interesting split (keep in mind that there is invisible border from the main cluster: "Clusters", the first map.). * We can see that some Democrats voted no over here, but *most* *Democrats* voted *yes*, and then *some* even *Republicans* voted *yes*. * red = yes, and blue = no. |  |
| * Broadcast Decency Enforce: Looks like there's a lot of yes-vote on this question. All of the Republicans definitely voted yes. Most of the Democrats voted yes. Some people voted no. * You can see a bit of yellow. The reason for this most likely, is that here we've got nodes. So that means we have about 225 nodes in our SOM, whereas in the US Congress, you have over 500 Members. So meaning that the way this map has been overlaid over our data, is not a one to one relationship. There are several Members of Congress represented within every node. * This yellow basically means that maybe *most* are *yes-voting*, but there was like one or two people who voted *no*. And same thing over other yellow areas, means there's kind of *inconsistency* there. * red = yes, and blue = no. |  |

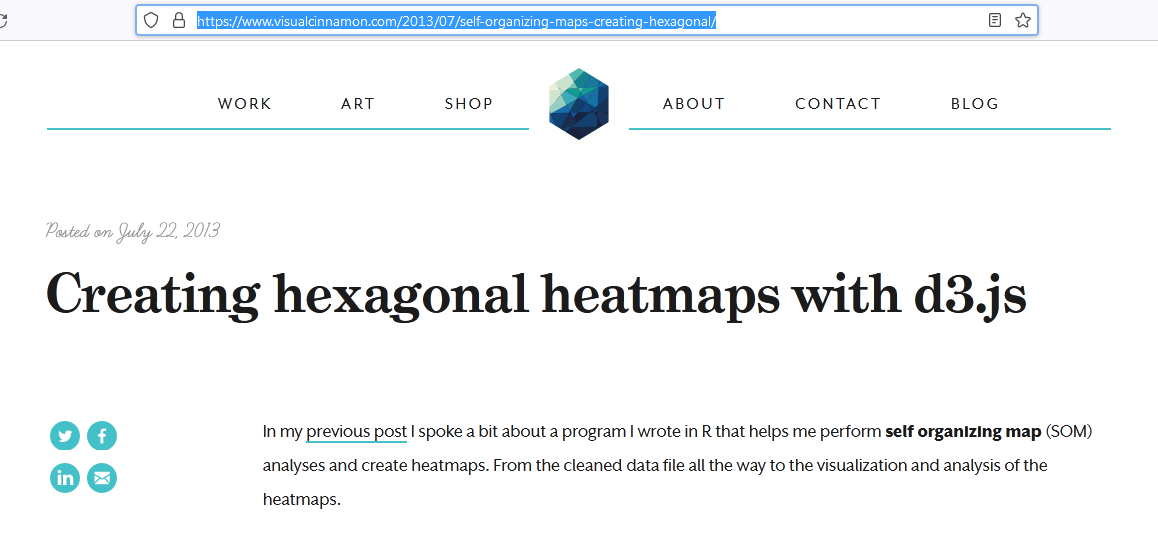
And that’s how we read an Advanced SOM.

|  |  |
| --- | --- |
| * Because of the simplicity of SOMs, you will find that there are lots and lots of different versions and variations of implementations of SOMs. * Here's another example from another website, ***boyeltab.org*** |  |

|  |  |
| --- | --- |
| * So this one is from ***R-bloggers***, it shows you can create a self organizing map in R. |  |
| * This is a self organizing map which we'll use in ***this*** ***Chapter***. |  |
| * This SOM from ***StackOverflow***. |  |
| * This one from ***viscovery***. * You can see these are the clusters that have been identified and then you have separate maps for each one of your features. Like the voting-SOM from the Wikipedia. |  |
| * Another from ***visualcinnamon.com***. * Here you can actually see those lines for highlights and identify the clusters. And then as you go through those separate *features* of your *data* *set*, you still keep these *clusters* in mind. |  |

* Additional readings: We reference this one below because it's actually quite a cool representation. This one is coded in ***D3.js***, so it's a ***JavaScript*** library.
* This is by *Nadieh* *Bremer* from *Netherlands* *Amsterdam* and she was kind enough to actually explain how she created this visualization and all the hexagons and you've got the codes here.
* <https://www.visualcinnamon.com/2013/07/self-organizing-maps-creating-hexagonal/>





|  |  |
| --- | --- |
|  |  |
|  |