

Universitatea „Alexandru Ioan Cuza” din Iași  
Facultatea de Informatică



LUCRARE DE LICENȚĂ  
Vizualizarea și analiza datelor  
spațiale

lucrare propusă de  
**Sebastian-Adrian Ciobanu**

**Sesiunea: iulie, 2017**

Coordonator științific:  
**Conf. Dr. Mihaela Breabăn**

**Universitatea „Alexandru Ioan Cuza” din Iași  
Facultatea de Informatică**

# **Vizualizarea și analiza datelor spațiale**

**Sebastian-Adrian Ciobanu**

**Sesiunea: iulie, 2017**

**Coordonator științific:  
Conf. Dr. Mihaela Breabăn**

# DECLARAȚIE PRIVIND ORIGINALITATE ȘI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licență cu titlul „*Vizualizarea și analiza datelor spațiale*” este scrisă de mine și nu a mai fost prezentată niciodată la o altă facultate sau instituție de învățământ superior din țară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- ☐ toate fragmentele de text reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și dețin referința precisă a sursei;
- ☐ reformularea în cuvinte proprii a textelor scrise de către alți autori deține referința precisă;
- ☐ codul sursă, imaginile etc. preluate din proiecte *open-source* sau alte surse sunt utilizate cu respectarea drepturilor de autor și dețin referințe precise;
- ☐ rezumarea ideilor altor autori precizează referința precisă la textul original.

Iași, 30 iunie 2017

Absolvent Sebastian-Adrian Ciobanu,

---

(semnătura în original)

# DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Vizualizarea și analiza datelor spațiale*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași, 30 iunie 2017

Absolvent Sebastian-Adrian Ciobanu,

---

(semnătura în original)

## Cuprins

Introducere.....	2
1. Motivație .....	2
2. Soluții existente .....	3
Contribuții.....	5
I. Tehnologii folosite .....	6
1. Algoritmul SOM (Self Organizing Maps).....	6
i. Necesitate.....	6
ii. Prezentare .....	7
iii. Grafice .....	9
iv. SOM și culorile de pe hartă .....	11
v. Măsurile de eroare .....	12
2. R Shiny .....	14
i. De ce R? .....	14
ii. Introducere în shiny.....	14
iii. Cum se face legătura dintre ui și server? .....	16
iv. Reactivitate (reactive programming) .....	18
v. Unde intervin HTML, CSS și Javascript?.....	21
vi. Modularizarea aplicațiilor shiny – modulele shiny .....	24
vii. Pachete complementare shiny .....	26
3. Alte pachete folosite .....	26
4. Chestiuni relevante în R .....	28
II. Ghid de utilizare a aplicației și alte mențiuni legate de implementare/arhitectură.....	31
1. Mini-arhitectura aplicației .....	31
2. Aplicația pas cu pas .....	33
i. Available Data .....	33
ii. Map .....	39
iii. Postfilter .....	42
iv. County Aggregates .....	44
Concluzii și idei de viitor .....	45
Bibliografie .....	46

# Introducere

## 1. Motivație

În zilele noastre, numărul surselor de date a crescut exponențial față de câteva decenii în urmă. Acum, datele pot fi preluate fie de la distribuitori specializați (de obicei, contra cost), fie de pe site-uri asociate domeniului datelor, fie din chestionare de specialitate etc.

Un lucru este cert: datele devin din ce în ce mai multe. Ele servesc totuși aceluiași scop: să fie manipulate într-un mod util, lucru care devine dificil în acest context dacă ne gândim să rezolvăm problema manual. Din acest motiv au apărut direcții noi în prelucrarea datelor.

Una dintre aceste direcții este Vizualizarea datelor. Datele nu mai sunt doar un tabel cu numere și text, ci capătă o altă dimensiune, una vizuală, prin diverse grafice, imagini, aplicații interactive etc.

Lucrarea de față se dorește a fi un instrument util pentru persoanele care lucrează cu date spațiale (**puncte** având latitudine și longitudine sau **zone** descrise într-un anumit format). În principiu, datele vor căpăta dimensiunea vizuală prin plasarea pe hartă, fiind colorate corespunzător nevoii utilizatorului.

Să luăm, spre exemplu, contextul imobiliar din Statele Unite ale Americii. Casele se vând și se cumpără cu un anumit preț care depinde de mai mulți factori. Dacă la un moment dat, o casă apare pe piață se poate pune întrebarea cu cât s-ar vinde aceasta. Evident, ar trebui să ne uităm la casele asemănătoare și să descoperim astfel prețul. În acest sens, avem nevoie de date despre case pentru a vedea care seamănă: numărul de dormitoare, de băi, prețul de cumpărare inițial, dacă s-au făcut renovări etc. În continuare, în ajutorul nostru vin algoritmii de Învățare automată pentru a prezice un astfel de preț.

O posibilă problemă aici este riscul de a lua în seamă doar factori ce țin exclusiv de casă, în genul celor menționați mai sus. Există cel puțin o altă dimensiune a casei: cea spațială. Intuitiv, o casă din Los Angeles este mai scumpă decât o casă din Memphis chiar dacă ele sunt la fel din punctul de vedere al caracteristicilor. Astfel, demersurile de până acum pentru aflarea prețului nu sunt îndeajuns. Trebuie inclusă și analiza spațială. Pe lângă calcule de distanțe dintre case, de un real ajutor ar fi o hartă unde acestea să fie vizualizate. Spre exemplu, punctele de pe hartă ar putea fi colorate după un anumit atribut și acest lucru ar facilita descoperirea zonelor în care atributul respectiv are o valoare aproximativ constantă.

## 2. Soluții existente

Una dintre primele întrebări care vin în minte având o asemenea problemă este: Există software deja dezvoltat pentru așa ceva? Răspunsul ar duce la două tipuri de soluții:

- **Soluții totale** – aici intră sistemele de tip GIS (Geographic Information System; Sistem Informațional Geografic)

Un sistem informațional geografic este un sistem care permite vizualizarea mai multor straturi pe hartă. Spre exemplu, lumea reală este compusă din mai multe straturi: geografic, al transporturilor, al adreselor etc. la care poți adăuga stratul rezultat de datele tale. Un exemplu este [www.giscloud.com](http://www.giscloud.com) care prezintă numeroase opțiuni de manipulare a datelor după cum se poate vedea în Figura 1:

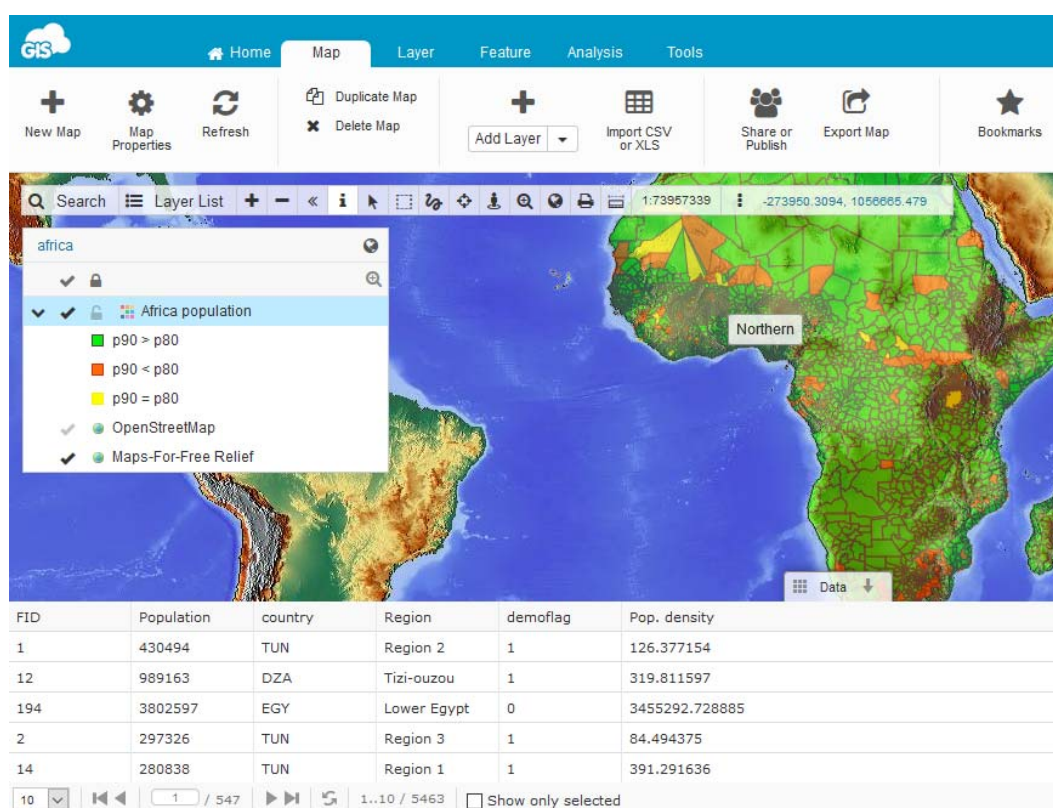


Figura 1: Exemplu GIS de pe [www.giscloud.com](http://www.giscloud.com)

- **Soluții parțiale** – aici intră bibliotecile din limbaje de programare

Utilizatorul ar putea rula un program folosind o bibliotecă pentru hărți oricând are noi date la dispoziție. Soluția este parțială din lipsa flexibilității și necesitatea ca utilizatorul să cunoască programare. Exemple de biblioteci sunt:

*ggmap*, *RgoogleMaps* – pentru hărți statice; *Leaflet* în JavaScript (care are biblioteci corespondente în Python și R), *Mapbox* în Javascript – pentru hărți interactive.

În alegerea unei soluții pentru rezolvarea problemei, am considerat necesare și următoarele: interactivitatea și dorința de a colora punctele/zonile de pe hartă după culori care să fie reprezentative din punctul de vedere al mai multor atribute ale datelor. Astfel, un GIS nu rezolva problema colorării și nu a putut fi ales ca soluție finală. În final, s-a ales biblioteca **leaflet** din limbajul de programare **R** pe baza căreia s-a construit întreaga aplicație. Dacă interactivitatea a fost trecută la necesități și dacă, în limbajul R, *leaflet* este singura bibliotecă ce furnizează hărți interactive, atunci ar fi limpede de ce s-a ales în acest fel dacă s-ar cunoaște motivele alegerii limbajului R, motive care vor fi regăsite mai ales în capitolul **Tehnologii folosite**.

După cum se poate vedea din *Cuprins*, lucrarea prezintă două mari capitole: *Tehnologii folosite* și *Ghid de utilizare a aplicației și alte mențiuni legate de implementare/arhitectură*. Dacă în primul capitol se prezintă mai detaliat cele două tehnologii de bază folosite (algoritmul SOM și pachetul *shiny* din R) cu surprinderea unor chestiuni relevante în implementarea aplicației (aici, accentul cade pe tehnologie), în capitolul al doilea se prezintă totul dinspre aplicație înspre mențiuni legate de implementare/arhitectură (aici, accentul cade pe aplicație).



## Contribuții

Lucrarea a vizat implementarea unei aplicații web pentru analiza vizuală a datelor spațiale existente într-o bază de dată relațională sau încărcate la cerere de către utilizator din fișiere text. Punctual, sunt aduse următoarele contribuții:

- integrarea în aplicație a algoritmului SOM (Self Organizing Maps) pentru realizarea de “*heatmap-uri*” și detectarea de zone geografice omogene din punctul de vedere al atributelor de interes
- postprocesarea rezultatului algoritmului SOM prin adăugarea de culori pe grilă
- calculul manual al erorii topografice pentru un algoritm SOM
- a fost posibilă apelarea unor funcții din *leafletJS* pe un obiect *leaflet* din R (funcțiile respective nu aveau corespondent în pachetul R)
- crearea de module *shiny*
- integrarea funcționalităților unor biblioteci JS/R într-o singură aplicație
- modificarea comportamentului implicit al unor elemente de interfață (ex.: legendele adăugate pe hartă au fost făcute să poată fi mutate)

## I. Tehnologii folosite

În acest capitol vor fi prezentate tehnologiile care au stat la baza dezvoltării aplicației. Acestea sunt împărțite în două: **algoritmice** și **de implementare**. În cadrul celor algoritmice se va prezenta algoritmul **SOM**, iar în cadrul celor de implementare se va prezenta pachetul **shiny** (și nu numai) din limbajul de programare **R**.

### 1. Algoritmul SOM (Self Organizing Maps)

#### i. Necesitate

Să ne imaginăm că lucrăm cu date ca cele din Tabelul 1.

HouseID	Longitude	Latitude	LandSqft	LandValue
1	-86.871199	33.753735	287496	51299
2	-86.833221	33.68803	827640	126299
3	-86.616408	33.753159	20000	23899

Tabelul 1: Trei rânduri dintr-un tabel cu date ce prezintă coordonate spațiale (longitudine și latitudine).

Din tabel, se poate observa ușor că, având longitudinea și latitudinea, putem plasa ușor fiecare casă pe hartă ca un punct. În continuare vine întrebarea cu ce culoare să colorăm fiecare punct. Un posibil răspuns este: toate cu aceeași culoare. Astfel, s-ar putea observa unde sunt mai aglomerate punctele, ceea ce ar ajuta. Totuși mai putem adăuga informație utilă pe hartă prin culori diferite ale punctelor care să exprime și altă informație decât cea de dispunere spațială.

Aceste culori ar putea fi generate, să spunem, de atributul *LandValue*. Se poate realiza un grafic de tip “*heatmap*” care utilizează o paletă de culori în degradé de la galben (valoarea minimă) la roșu (valoarea maximă) și astfel se pot indica zonele pe hartă cu valori *LandValue* scăzute/ridicate. Aceste zone pot fi considerate de sine stătătoare și analiza de date va continua doar pe acele zone.

Același lucru se poate realiza și cu atributul *LandSqft*, însă zonele, în general, vor fi altele. Dacă avem mai multe atribute, ar trebui să facem acest lucru pentru fiecare în parte și să intuim care ar fi zonele cu care ar trebui să lucrăm în continuare. Pentru un număr de atribute egal cu 50, procesul devine destul de incomod. Ideal ar fi să combinăm atributele care ne interesează pe noi (în exemplul nostru: *LandSqft* și *LandValue*) într-un singur atribut (numeric sau, direct, de tip culoare) pe care să-l plasăm pe hartă. O astfel de soluție este dată de folosirea algoritmului SOM, urmată de o postprocesare.

## ii. Prezentare

Algoritmul SOM, cunoscut și sub numele de *hărți Kohonen* [1], primește ca **input** date (de obicei) cu mai mult de două atribute, iar ca **output** returnează o reprezentare (de obicei) bidimensională a datelor. Este astfel, o modalitate de reducere a dimensionalității datelor. Acea reprezentare bidimensională nu reprezintă altceva decât o **matrice (grilă)**. Fiecare instanță din datele de intrare este asignată unui **element (nod)** din matrice. Un nod poate conține mai multe instanțe asignate. Outputul prezintă o **proprietate importantă** în condițiile în care algoritmul converge: date asemănătoare (spre exemplu, distanță euclidiană mică) din punctul de vedere al atributelor vor fi asignate unor noduri de pe grilă care sunt apropiate (poate chiar în același nod), iar date diferite (distanță euclidiană mare) vor fi asignate unor noduri depărtate unele de altele. De pildă, dacă instanța A a fost asignată nodului stânga sus, iar instanța B, nodului dreapta jos, atunci ar trebui ca datele A, B să aibă o distanță mare între ele în spațiul din care provin (cel al datelor de intrare).

Algoritmul SOM poate fi privit din cel puțin două puncte de vedere:

- a. Ca metodă din cadrul învățării nesupervizate a **rețelelor neuronale**
  - b. Ca o generalizare a algoritmului **k-means**
- a. Ca metodă din cadrul învățării nesupervizate a **rețelelor neuronale** [37]

Prezintă următoarea structură:

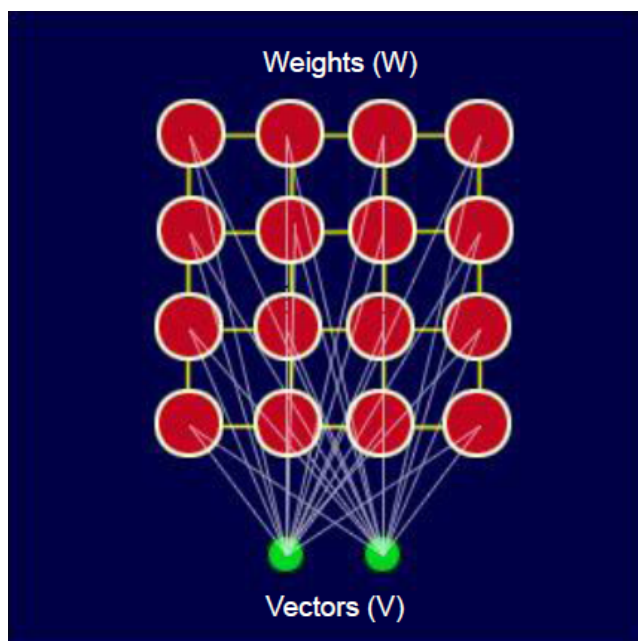


Figura 2: Structura rețelei neuronale din cadrul algoritmului SOM [37]

Vectorii (V; neuronii input ai rețelei neuronale; primul strat de neuroni) reprezintă atributele din datele de intrare (de exemplu, ar putea fi vorba de cele două atribute din Tabelul 1: *LandSqft* și *LandValue*). Fiecare nod din grilă este conectat cu fiecare neuron input, însă nu

este conectat cu niciun nod al grilei. Avem astfel câte o pondere de la fiecare atribut de intrare la fiecare nod de pe grilă. Fiecare nod de pe grilă are, deci, un număr de ponderi egal cu numărul de atribute de intrare. Când se menționează că se calculează o distanță de la o instanță input la un nod de pe grilă, nodul este privit ca vector având componentele date de ponderi.

Ca algoritm de învățare automată, prezintă două faze: **antrenare** și testare. În cadrul problemei noastre, ne va interesa doar faza de antrenare (nu vor exista noi instanțe cărora să le găsim un corespondent pe grilă).

Algoritmul de antrenare este următorul:

- Ponderile rețelei sunt inițializate (de obicei, aleatoriu)
- Se alege o instanță aleator din datele de intrare
- Se găsește nodul de pe grilă cel mai apropiat de această instanță (calculându-se distanțe). Nodul găsit este deseori numit BMU (Best Matching Unit)
- Se găsește raza care dă vecinătatea corespunzătoare acestui BMU pe grilă. Raza va descrește de la o iterație la alta. (Sau, altfel spus, se calculează ponderea asociată actualizării unui nod de pe grilă, în funcție de numărul iterației și distanța – calculată din punctul de vedere al grilei – față de BMU.)
- Fiecare nod din vecinătatea găsită se va actualiza astfel încât să devină mai apropiat de instanța aleator aleasă. Nodurile mai apropiate de BMU vor fi ajustate mai mult, decât cele mai depărtate (este vorba de ponderea mai sus menționată).
- Se repetă pașii începând cu pasul 2 pentru un număr de iterații.

b. Ca o generalizare a algoritmului **k-means**

Algoritmul k-means pornește cu un număr (k) de centroizi inițializați aleator și apoi aceștia sunt actualizați, apropiindu-se de datele de intrare. La fiecare iterație, pentru fiecare instanță se caută centroidul cel mai apropiat. Astfel, fiecare centroid va avea asignate mai multe instanțe. Noua valoare a centroidului ca fi media instanțelor asignate. Avem de-a face cu o formulă de tipul următor, unde j reprezintă indexul centroidului, iar  $n_j$ , numărul de instanțe (x) asignate lui  $C_j$ :

$$C_j = \frac{x_1 + x_2 + \dots + x_{n_j}}{n_j}$$

Rescriind formula obținem:

$C_j = \frac{1*x_1 + 1*x_2 + \dots + 1*x_{n_j} + 0*x_{n_{j+1}} + \dots + 0*x_n}{1+1+1+\dots+0+\dots+0}$ , unde cu  $x_{n_{j+1}}, \dots, x_n$  am notat instanțele care nu sunt asigurate centroidului.

Algoritmul SOM prezentat la subpunctul a este algoritmul original SOM. Se poate observa că acesta este un algoritm online (aleg o instanță => modificare parametri). Există, ca la alți algoritmi de învățare automată, o variantă de tip batch (iau tot setul de date => modificare parametri)[4]:

- Ponderile rețelei sunt inițializate (de obicei, aleatoriu)
- Pentru fiecare instanță din datele de intrare, se găsește BMU
- Pentru fiecare pereche (x,y) de noduri de pe grilă se calculează ponderea asociată actualizării nodului x, în funcție de numărul iterației și distanța, calculată din punctul de vedere al grilei, față de y. Fie aceasta  $h_t(x,y)$ .

Obs.:  $h_t(x,y) = h_t(y,x)$

- Ponderile rețelei se actualizează astfel:

$$W_j = \frac{\sum_{i=1}^n h_t(BMU(x_i), j) x_i}{\sum_{i=1}^n h_t(BMU(x_i), j)}$$

- Se repetă pașii începând cu pasul 2 pentru un număr de iterații.

Făcând legătura dintre SOM și k-means, reprezentarea unui nod de pe grilă în vector de ponderi este dată de centroid. În k-means, nu există conceptul de vecinătate, deci nu există conceptul de grilă. De fapt, algoritmul SOM ne servește ca soluție pentru problema noastră tocmai datorită introducerii ideii de vecinătate pe grilă. Altfel, după cum se va vedea imediat, outputul nu ar fi fost altul decât cel al unui k-means (centroizi ai unor clustere).

Recapitulând, formulele de actualizare sunt:

$$C_j = \frac{1*x_1 + 1*x_2 + \dots + 1*x_{n_j} + 0*x_{n_{j+1}} + \dots + 0*x_n}{1+1+1+\dots+0+\dots+0} - \text{pentru k-means}$$

$$W_j = \frac{\sum_{i=1}^n h_t(BMU(x_i), j) x_i}{\sum_{i=1}^n h_t(BMU(x_i), j)}; \text{pentru SOM de tip batch.}$$

Se observă cum pentru k-means funcția  $h_t$  este definită:

$$h_t(x, y) = \begin{cases} 1, & x = y \\ 0, & x \neq y \end{cases}$$

Astfel, algoritmul SOM devine o generalizare a algoritmului k-means.

### iii. Grafice

După cum am menționat, ca output, algoritmul SOM furnizează o grilă (matrice). În 2D, de obicei grila poate fi în formă **dreptunghiulară** sau în formă **hexagonală**. Diferența

este aceea că un nod de pe grilă va avea maxim 4 vecini (dreptunghi) sau 6 vecini (hexagon). În calculul unei măsuri de eroare pentru SOM, acest lucru va conta.

Graficele standard utilizate în cadrul aplicației au fost de trei tipuri [38, 40]:

- **Codes plot** (vezi Figura 3)

La finalul algoritmului, fiecare nod din grilă are asociate niște ponderi. Aceste ponderi sunt prezentate vizual prin graficul de tip codes. Dacă fiecare nod are în corespondență o arie pe care se poate desena (un cerc, după cum se vede în Figura 3), atunci acea arie este împărțită în mod egal la numărul de atribute (3, în Figura 3), fiecare subarie astfel obținută fiind repartizată unui singur atribut. Ponderile sunt desenate în funcție de nodul și atributul la care corespund (=> suprafața unde se va desena) și valoarea (=> cât din suprafața găsită se va desena).

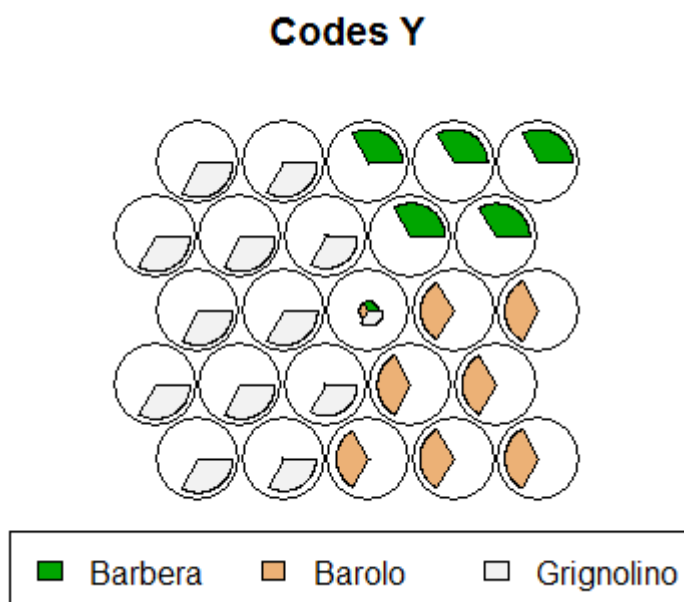


Figura 3: Grafic de tip codes pentru SOM. Grila este de tip hexagonal.

- **Heatmap plot** (vezi Figura 4)

Graficul anterior este destul de folositor în cazul în care numărul de atribute este până în 10, să zicem. De la un număr de atribute în sus, nu ne mai ajută. În acest sens, există graficele heatmap prin intermediul cărora putem vizualiza un singur atribut. Urmărind scala asociată graficului ne dăm seama, pentru un nod de pe grilă, de valoarea atributului ales pentru a fi reprezentat prin heatmap (adică de ponderea rezultată în urma algoritmului pentru un nod și un atribut). Vizualizând mai multe astfel de grafice, se obțin aceleași informații dacă am fi putut vizualiza direct un *Codes plot*.

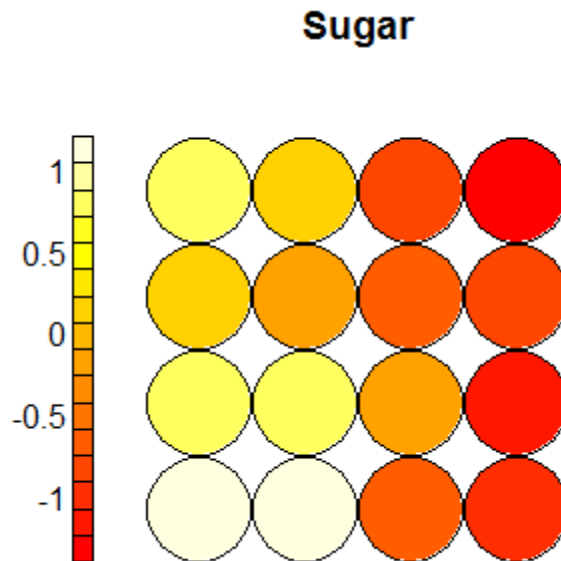
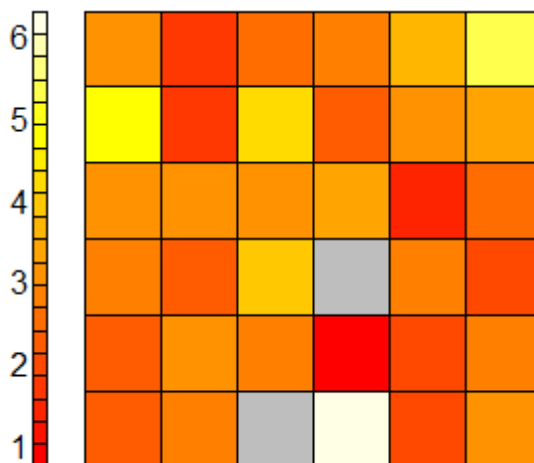


Figura 4: Grafic de tip heatmap pentru SOM pentru atributul *Sugar*. Grila este de tip dreptunghiular.

- **Quality plot** (vezi Figura 5)

Acest grafic este în același stil ca și graficul heatmap, însă valoarea asignată nodului este media distanțelor dintre fiecare instanță asociată unui nod și nodul respectiv. Graficul ne ajută să percepem calitatea grilei rezultate în urma algoritmului, după cum indică și numele de *quality*. Nodurile având culori



asociate valorilor mici de pe scală sunt bine formate. Ideal, ar fi ca mai toate nodurile să aibă o culoare asociată minimului de pe scală.

Figura 5: Grafic de tip quality pentru SOM. Grila este de tip dreptunghiular. Majoritatea nodurilor au culoarea portocalie (pe la jumătatea scalei). Probabil algoritmul SOM ar trebui rulat din nou cu mai multe iterații.

#### iv. SOM și culorile de pe hartă

După cum am precizat deja, subproblema colorării după mai multe attribute a datelor de intrare s-a rezolvat prin algoritmul SOM, urmat de o **postprocesare** a rezultatului.

Deși în subsecțiunea iii. avem de-a face cu grafice în care apar diverse culori, am putea fi tentați să asociem instanțelor din același nod din grilă culoarea corespunzătoare nodului. Dacă am prelua culorile de la un heatmap, am face aproximativ același lucru ca la

colorarea punctelor pe hartă după un anumit atribut (deci, algoritmul SOM ar fi fost rulat inutil). Dacă am prelua culorile de la graficul de tip *quality*, am vizualiza puncte pe hartă care, în cazul în care algoritmul SOM a fost rulat cu un număr semnificativ de iterații, ar fi colorate toate după aproximativ aceeași culoare (minimul de pe scala afișată pe graficul de tip *quality*).

În subsecțiunea ii am menționat o **proprietate importantă** de care nu ne-am folosit încă: date depărtate, respectiv apropiate, unele de altele în spațiul atributelor de intrare vor fi depărtate, respectiv apropiate, și în spațiul bidimensional dat de grilă. Ne vom folosi de acest lucru în felul următor: Vom genera o paletă de culori în degradé 2D pe care o vom suprapune peste grila rezultată din SOM. O astfel de suprapunere este vizibilă în Figura 6.

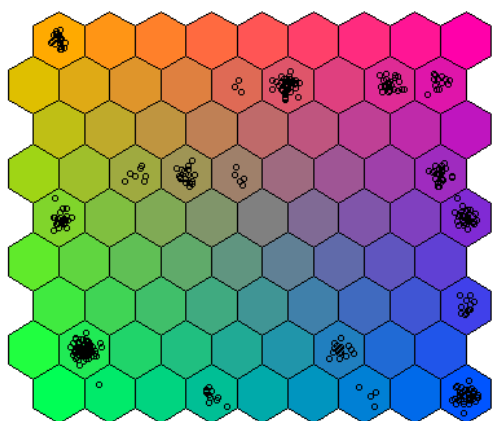


Figura 6: Paletă de culori în degradé 2D suprapusă peste o grilă rezultată în urma algoritmului SOM. Punctele negre sunt instanțe asociate unui nod. Spre exemplu, nodul gălbui cu cel albastru vor corespunde instanțelor foarte depărtate în spațiul datelor de intrare. Cele având nuanțe de verde vor corespunde instanțelor foarte apropiate în spațiul datelor de intrare.

După cum se menționează în adnotarea Figurii 6, prin această postprocesare, culorile apropiate estetic (nuanțe de verde) vor corespunde instanțelor apropiate. Invers, se întâmplă pentru culorile depărtate estetic (galben, albastru). În acest fel, putem asocia culori pentru instanțele de pe hartă (preluând culoarea, în urma postprocesării, corespunzătoare nodului la care instanța a fost asignată).

Trebuie menționat un lucru foarte **important**: dacă în cazul colorării unui singur atribut, culorile de pe hartă exprimă valori mari sau mici ale atributului selectat, în cazul colorării după mai multe atribute, culorile de pe hartă nu exprimă nimic legat de cantitate (adică o culoare puternică, în genul roșului, NU exprimă o valoare medie ridicată a atributelor selectate), ci redau asemănările/deosebiriile dintre instanțe (două culori asemănătoare vizual indică două instanțe asemănătoare; două culori diferite vizual indică două instanțe diferite).

#### v. Măsuri de eroare

Există două măsuri de eroare tipice pentru a vedea calitatea grilei rezultate în urma algoritmului SOM. De obicei, când valoarea uneia nu este scăzută, algoritmul se rulează din nou cu alți parametri (de obicei, un număr mai mare de iterații sau se modifică dimensiunile grilei). Aceste măsuri de eroare sunt [5]:



- **Eroarea medie de cuantizare**<sup>1</sup> (Average Quantization Error)

$$E_q = \frac{\sum_{i=1}^n \text{dist}(x_i, BMU(x_i))}{n}$$

Această eroare prezintă informația din graficul de tip quality printr-un singur număr. În acel tip de grafic, pentru fiecare nod  $j$  avem următoarea valoare:

$$E_{qj} = \frac{\sum_{i=1}^{n_j} \text{dist}(x_i, j)}{n_j}$$

Sunt luate în considerare doar nodurile  $x_i$  asiguate nodului  $j$  ( $BMU(x_i) = j$ ).

- **Eroarea topografică** (Topographic Error)

$$E_t = \frac{\sum_{i=1}^n f(x_i)}{n}$$

$$f(x) = \begin{cases} 0, & BMU(x) \text{ și } secondBMU(x) \text{ sunt vecini pe grilă} \\ 1, & \text{altfel} \end{cases}$$

Această eroare verifică oarecum dacă grila prezintă proprietatea importantă pe care am mai menționat-o legată de menținerea depărtării/apropierii din spațiul datelor de intrare prin plasarea pe grilă.

Tot aici, este relevant dacă grila este de tip dreptunghi sau hexagon în verificarea existenței vecinătății dintre două noduri.

---

<sup>1</sup> După [49]: cuantizarea este procesul de convertire a mulțimii datelor de intrare, într-o mulțime de cardinal mai mic, astfel încât valorile obținute ca date de ieșire aproximează valorile datelor de intrare. La un nivel foarte general, cuantizare = aproximare.

## 2. R Shiny

Subsecțiunile acestui capitol descriu arhitectura unei aplicații *shiny*. Subsecțiunile ii și iii utilizează în mare măsură informațiile prezentate în [17], subsecțiunea iv, informațiile din [19, 20, 23], iar subsecțiunea vi, informațiile din [18].

### i. De ce R?

Am ales limbajul R pentru a implementa marea parte a aplicației datorită faptului că aceasta implică analiza datelor, iar R facilitează foarte mult operațiile asupra datelor (preluarea datelor din fișiere CSV (Comma-separated values)/TSV (Tab-separated values), filtrarea datelor preluate etc.; existența tipului de date *data.frame* ajută foarte mult aici). În plus, în R există pachete (biblioteci) care implementează algoritmul SOM și realizează grafice destul de ușor. Datorită faptului că s-a adăugat o bibliotecă pentru lucrul cu hărți interactive (*leaflet*) în R, a faptului că există posibilitatea creării de interfețe grafice Web în R (pachetul *shiny*) și a faptului că pachetul *leaflet* își are originea în biblioteca JS *leafletJS* (deci, integrarea cu *shiny* este posibilă), alegerea finală a limbajului a fost R.

### ii. Introducere în shiny

*Shiny* reprezintă un pachet în R prin intermediul căruia se pot construi aplicații Web. Pentru instalare trebuie executată următoarea comandă în sesiunea R:

```
install.packages("shiny")
```

În prezentări ale acestui pachet este regăsită ideea că pentru dezvoltarea unei aplicații *shiny* (*shiny* app) nu este nevoie de cunoștințe de HTML/CSS/Javascript. Totuși, este bine să fie cunoscute aceste tehnologii pentru situațiile în care apar limitări în cadrul pachetelor R de dezvoltare de *shiny* apps, deoarece multe elemente pot fi extinse sau personalizate prin HTML/CSS/Javascript [14].

*Shiny* se bazează pe partea de front-end pe Bootstrap<sup>2</sup> (=> responsive) și pe partea de back-end pe Web Sockets<sup>3</sup> (=>interactivitate).

Aplicația *shiny* de bază are două forme, la fel cum orice aplicație *shiny* poate fi scrisă în două forme, vizibile în Tabelul 2:

---

<sup>2</sup> Framework de HTML, CSS, JS pentru dezvoltarea aplicațiilor responsive pe Web

<sup>3</sup> Tehnologie în care, față de HTTP, serverul nu închide conexiunea imediat după transmiterea datelor către client și nu este nevoit să primească un request de la client pentru a-i trimite ceva acestuia.

Forma single-file app.R	Forma multi-file		
<pre>library(shiny) ui &lt;- fluidPage() server &lt;- function(input, output) {} shinyApp(ui=ui, server=server)</pre>	ui.R	server.R	<pre>shiny::runApp(".") #"." = directorul în care se află app.R, respectiv ui.R și server.R (trebuie neapărat să fie ambele în același director).</pre>
	<pre>fluidPage()</pre>	<pre>function(input, output) {}</pre>	

Tabelul 2: Aplicația *shiny* de bază, în cele două forme posibile

Pentru aplicații simple este preferată forma single-file, iar în rest se apelează la forma multi-file (care mai poate conține, opțional, și fișierul *global.R*).

După cum se poate observa pentru rularea aplicației este nevoie de două argumente: ***ui*** și ***server***.

- ***ui*** – argument reprezentând pagina HTML statică (în principiu, doar conținutul *body* din cadrul unei pagini HTML) care apare imediat după rularea aplicației; aceasta poate fi generată în cel puțin două moduri:
  - stocarea textului HTML într-un string și apelarea funcției *HTML* pe acel string; de exemplu, `HTML('<div> Primul program </div>')`
  - folosirea de funcții din R pentru generarea într-o manieră mai simplă a elementelor HTML; de exemplu, `tags$div('Primul program')`

Ca regulă de bază, pentru a genera codul HTML corespunzător unui element, se folosesc funcțiile din obiectul *tags* din R care poartă numele elementului. Apelul unei astfel de funcții se face cu parametri de tip valoare (aceștia vor deveni conținut al elementului respectiv) sau cu parametri de tip cheie-valoare (aceștia vor deveni attribute ale elementului respectiv). Rezultatul aplicării acestor funcții se poate vedea la consola R. De exemplu,

```
tags$div(id='my_div', 'content')
```

va fi corespondent pentru

```
<div id='my_div'>content</div> .
```

Următorul apel este și el valid:

```
tags$div(id='my_div', 'content', tags$div('content'))
```

fiind corespondent pentru

```
<div id='my_div'>content<div>inner content</div></div>
```

- server – o funcție cu 3 parametri: input, output și session. Ultimul parametru este opțional.

Ordinea de execuție a codului pentru o aplicație *shiny* app în forma multi-file este următoarea:

- global.R – fiind primul fișier al cărui cod se execută, aici pot fi incluse apelurile de tip *library()* (includerea unor biblioteci în R), codul de conexiune la o bază de date, citirea datelor dintr-un fișier, crearea variabilelor care să fie folosite atât în ui.R, cât și în server.R
- ui.R – doar ultima expresie din fișier va reprezenta obiectul ui asociat aplicației *shiny* (acesta va fi referit de-a lungul lucrării prin *ui* sau *obiectul ui*); în rest, înainte de ultima expresie poate exista cod
- server.R – doar ultima expresie din fișier va reprezenta obiectul server asociat aplicației *shiny*; în rest, înainte de ultima expresie poate exista cod

Observație: codul din funcția din server.R este executat la fiecare conexiune a unui client la server, iar tot celălalt cod este executat o singură dată, la lansarea aplicației *shiny*.

### iii. Cum se face legătura dintre ui și server?

De obicei, în aplicațiile Web, datele de intrare sunt plasate în elemente HTML de tip input. Nici *shiny* nu se îndepărtează de la regulă. Astfel pentru a primi date de intrare de la utilizator pot fi apelate în ui funcții precum *tags\$input()* sau funcții specializate de tip *\*Input()*, unde *\** poate fi înlocuită prin *dateRange*, *checkbox*, *file*, *select*, *checkboxGroup*, *numeric*, *slider*, *date*, *password*, *text*, sau alte funcții specializate: *radioButtons()*, *actionButton()*.

Toate funcțiile specializate de mai sus generează HTML-ul corespunzător și prezintă cel puțin două argumente: *inputId* și *label*. *Label* are doar rol estetic, însă *inputId* este foarte important pentru comunicarea cu serverul. După cum am văzut deja, serverul este o funcție cu 3 parametri (input, output, session). Intuitiv, doar primul parametru are legătură cu datele primite de la utilizator. Datele de intrare care ajung la un element input din ui cu *inputId* = “*x*” vor putea fi accesate în server prin expresia *input\$x*. Un necunoscător al pachetului *shiny*, ar încerca în acest moment, după ce am prezentat aceste idei, un exemplu asemănător celui din Tabelul 3:

```
library(shiny)
ui <- fluidPage(textInput(inputId = "text", label = "Introdu text:"))
server <- function(input,output,session) {
  print(input$text)
}
shinyApp(ui=ui, server=server)
```

Tabelul 3: Exemplu de cod în R *shiny*

La rulare însă, va apărea o eroare: *Error in .getReactiveEnvironment()\$currentContext: Operation not allowed without an active reactive context. (You tried to do something that can only be done from inside a reactive expression or observer.)*. Problema a fost la apelul funcției `print`. Mai multe informații în acest sens vor fi furnizate în subsecțiunea următoare.

Momentan, nu putem verifica dacă în `input$text` avem efectiv datele de intrare de la utilizator. Voiam să verificăm acest lucru prin afișarea la consolă. O alternativă pe care o putem încerca ar fi să afișăm clientului textul introdus. Pentru aceasta, de obicei, într-o aplicație Web, pe partea de front trebuia să creăm un element HTML în care să introducem mesajul primit de la server. De data aceasta, *shiny* se abate de la regulă și vine cu următoarea soluție: în ui să se plaseze elementul dat de apelul la o funcție specializată de tip *\*Output*, unde *\** poate fi înlocuită prin *dataTable*, *html*, *image*, *plot*, *table*, *text*, *ui*, *verbatimText*. Primul argument al acestor funcții este un *ID*, care poate fi referit în server prin `output$ID`. În server, pentru a asigura o valoare acestui output, nu vom scrie direct `output$ID <- x` (pentru a nu primi eroarea anterioară), ci `output$ID <- render*({x})`, unde *\** va reprezenta aproximativ cu ce a fost înlocuită steluța în linia de cod `*Output("ID")` din ui (*DataTable*, *Image*, *Plot*, *Print*, *Table*, *Text*, *UI*). Exemplul din Tabelul 3 devine acum:

```
library(shiny)
ui <- fluidPage(
  textInput(inputId = "text_in", label = "Introdu text:"),
  textOutput("text_out")
)
server <- function(input,output,session) {
  output$text_out <- renderText({
    input$text_in
  })
}
shinyApp(ui=ui, server=server)
```

Tabelul 4: Exemplu de cod în R *shiny* funcțional

#### iv. Reactivitate (reactive programming)

##### 1. În general

Fie următoarele linii de cod în R:

```
a <- 1  
b <- a  
a <- 2
```

Tabelul 5: Exemplu de cod simplu (doar atribuiri) în R

Valoarea lui  $b$  în urma executării celor trei instrucțiuni este 1. În schimb, dacă variabila  $a$  ar fi fost reactivă, atunci, la schimbarea valorii sale prin instrucțiunea 3, variabila  $b$  trebuia să fie notificată că ar trebui să-și schimbe valoarea, pentru că  $b$  depinde de  $a$ , iar  $a$  s-a schimbat (ar trebui, deci, ca instrucțiunea 2 să se reexecute).

##### 2. În shiny

În *shiny*, există trei tipuri de obiecte reactive. Mai întâi vom prezenta modelul și apoi implementarea acestora.

Modelul obiectelor reactive cuprinde: **surse reactive**, **conductori reactivi** și **puncte finale reactive** (Reactive Endpoints). Acestea vor fi noduri ale unui graf orientat, iar între ele vor putea exista arce: de la surse/conductori la conductori/puncte finale. Dacă există arc de la A la B, atunci spunem că B depinde de A. Acest graf este folosit în următorul context:

- La lansarea aplicației, se realizează graful de reactivitate (noduri + arce)
- La o modificare a sursei reactive X (din interfața aplicației sau din server), toți descendenții (direcți sau indirecti) nodului X vor fi marcați drept invalidați, iar toate arcele plecând fie din X, fie dintr-un descendent al lui X vor fi șterse
- Toate punctele finale reactive invalidate vor fi reexecutate, într-o ordine aleatorie
- În codul specific unui punct final vor (putea) exista referințe la surse și/sau conductori. Când se găsește o astfel de referință X în codul punctului final Y, un arc de la X la Y va fi adăugat. Valoarea lui X va fi preluată imediat, dacă este vorba despre o sursă. Dacă este vorba de un conductor, mai întâi se va verifica dacă el este invalidat sau nu. Dacă este invalidat, atunci conductorul este reexecutat în maniera descrisă mai sus în care este reexecutat un punct final. Dacă nu este invalidat, atunci valoarea este preluată imediat (valoarea calculată la ultima execuție a codului din conductor)

Un exemplu simplu este prezent în Figura 7:

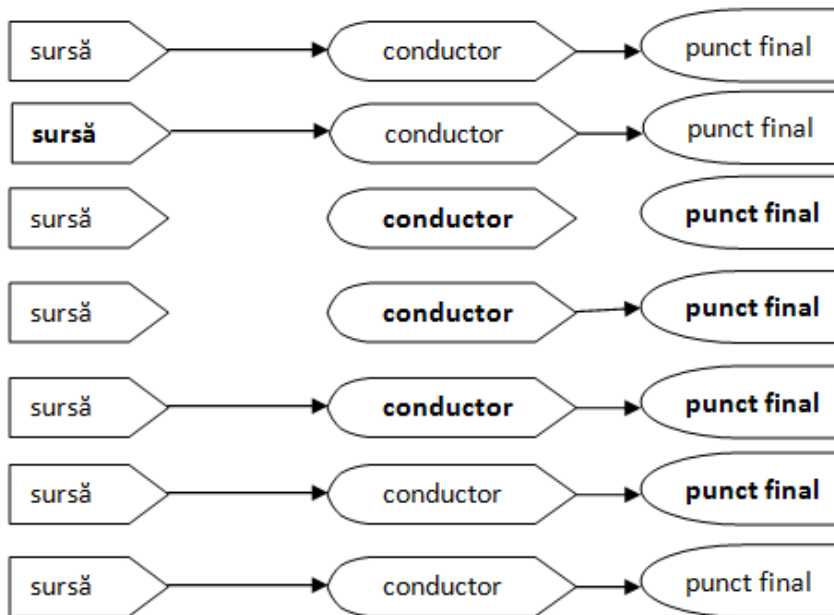


Figura 7: Starea grafului de reactivitate. La început, se formează graful. Apoi, utilizatorul manipulează interfața în așa fel încât sursa reactivă este modificată. Apoi, toți descendenții sursei vor fi invalidați (au fost marcați prin îngroșare). Începe reexecuția punctului final. Acesta în cod are o referință către conductor și încearcă să-i preia valoarea (apare și arcul de la conductor la punctul final). Conductorul este invalidat și trebuie și el reexecutat. În codul lui întâlnește o referință către sursă și îi preia valoarea (apare și arcul de la sursă la conductor). Acum, conductorul nu mai este invalidat, iar valoarea sa o trimite la punctul final, care termină execuția codului și nu mai este invalidat.

Ca implementare, există următoarea corespondență: sursă reactivă – **variabilă reactivă**, conductor reactiv – **expresie reactivă**, punct final – **observator**.

Mai există termenul de **context reactiv** care de obicei se referă la codul conținut în expresia reactivă sau în observator.

Exemplul tipic de variabilă reactivă este orice element al listei *input* (de exemplu: *input\$text\_in*).

Exemplul tipic de punct final este orice element la listei *output* (de exemplu: *output\$text\_out*).

Exemplul tipic de context reactiv este codul dat ca parametru funcției **render\*** (ceea ce va returna *render\** va fi atribuit unui element din *output*).

Un dezavantaj al elementelor listei *input* este faptul că pot fi modificate doar din interfață. Dacă se dorește a se modifica o variabilă reactivă în server (pentru a declanșa procesul corespunzător grafului de reactivitate), atunci se va apela funcția **reactiveValues**:

```
x <- reactiveValues(flag = TRUE)
```

x va fi o listă de variabile reactive. Momentan, există o singură variabilă reactivă (*flag*).

Pe lângă observatorii generați de funcțiile de tip *render\**, există și observatori generați de funcția *observe*. Față de *render\**, rezultatul lui *observe* nu este stocat în niciun element din output (de obicei, în nicio variabilă).

Expresii reactive sunt generate de funcția *reactive*. Aceasta va returna un obiect (să zicem, X) în genul unei funcții (pentru a primi rezultatul trebuie făcut un apel la X, adică referințele din cod pentru X vor fi X() și nu, simplu, X; cazul cel din urmă nu va avea niciun efect asupra grafului de reactivitate).

Pentru funcția *observe* există varianta *observeEvent()* care ca prim parametru primește variabila/expresia reactivă care să invalideze acest observator. Codul dat la al doilea parametru nu va declanșa crearea de arce pe graful de reactivitate. La fel, se întâmplă în cazul funcției *reactive* pentru care există varianta *eventReactive()*.

Funcțiile *observeEvent* și *eventReactive* prezintă doi parametri opționali prin care se elimină niște cazuri în care s-ar produce invalidarea: *ignoreNULL* și *ignoreInit*. Dacă *ignoreNULL* este setat pe TRUE, atunci variabila/expresia reactivă nu va invalida dacă este de tip NULL (în cazul unui buton, acest lucru este echivalent cu faptul că valoarea corespunzătoare din lista *input* este 0: prin apăsarea unui buton, se incrementează valoarea din lista *input*). Dacă *ignoreInit* este setat pe TRUE, atunci la crearea obiectului (prin apelul funcției *observeEvent/eventReactive*), variabila/expresia reactivă nu va invalida.

Observații:

- Există și alt context reactiv, dat de funcția *isolate*. Nu se produce nicio modificare asupra grafului de reactivitate prin includerea unui apel *isolate* în cod.
- O variabilă/expresie reactivă poate fi folosită **doar** într-un context reactiv. Altfel, vom primi eroarea din subsecțiunea iii.
- Când graful de reactivitate este realizat la lansarea aplicației, se procedează astfel: punctele finale și conductorii sunt invalidați, iar apoi procesul este cel normal (ca plecând din pasul 3 din Figura 7).
- Dacă un conductor nu are descendenți (de tip punct final), atunci el nu se va executa niciodată.
- Din punct vedere practic, expresiile reactive sunt necesare pentru a nu executa același cod (care poate fi costisitor ca timp sau spațiu) de mai multe ori. Totodată, reprezintă o manieră de a **partaja** o valoare în cadrul observatorilor/altor expresii reactive.



- Deși expresiile reactive și observatorii seamănă, ei au fost concepuți cu scopuri diferite: expresiile - pentru **memorarea** unei valori; observatorii - pentru **efecte secundare** (side effects; de obicei este folosit *observeEvent* care are ca prim argument un element din lista *input* corespunzător unui *actionButton*).
- Codul corespunzător expresiilor reactive sau observatorilor trebuie privit drept cod ce **se va executa** și nu drept cod ce se execută. În exemplul din Tabelul 4, pe partea de server, mai întâi este executat codul din funcția *server* (atribuirea), iar apoi abia (când în browser, elementul *text\_out* este vizibil) este executat codul din *renderText*. Acest cod va fi executat de fiecare dată când *text\_in* va fi modificat.

## v. Unde intervin HTML, CSS și Javascript?

### 1. Nivel de bază

După cum am menționat, elemente **HTML** pot fi incluse în interfața utilizator prin intermediul funcțiilor din obiectul *tags*. *tags\$head* este folosită pentru a include cod HTML în elementul *head* al paginii din interfață. Este recomandabil ca fiecare apel al funcției *tags\$head* să fie combinat cu un apel al funcției *singleton* (în genul *singleton(tags\$head(...))*), pentru ca, de-a lungul aplicației, același apel la *tags\$head* să nu introducă de două ori același conținut în elementul *head* al paginii din interfață, ci doar o singură dată.

În cazul în care se dorește gruparea mai multor elemente HTML direct din R acestea pot fi introduse ca argumente în *tags\$div* sau, specific *shiny*, în *tagList*.

Funcția *includeHTML* este folosită pentru a citi cod HTML dintr-un fișier (echivalentul unui string).

Proprietăți **CSS** asupra elementului HTML *X* pot fi introduse prin adăugarea unui argument de tip cheie-valoare (cheia = *style*) apelului funcției *tags\$X* (de exemplu, *tags\$div(style="color:red")*). Funcția *includeCSS* este folosită pentru a citi cod CSS dintr-un fișier (echivalentul unui string).

Cod **Javascript** poate fi introdus prin apelul funcției *tags\$script*. Funcția *includeScript* este folosită pentru a citi cod Javascript dintr-un fișier (echivalentul unui string).

Deși există funcția *tags\$head* care este folosită pentru a include în antet foi de stiluri externe (prin apelul de genul

```
tags$head(tags$link(rel="stylesheet",
                  type="text/css",
```

`href="style.css"))`, unde *style.css* trebuie să se afle în directorul *www* din cadrul directorului aplicației, adică acolo unde se află *app.R* sau *ui.R* și *server.R*) sau scripturi Javascript (prin apelul de genul

`tags$head(tags$script(src="script.js"))`, unde pentru *script.js* sunt aceleași mențiuni ca și pentru *style.css*), există o altă metodă pentru a face acest lucru (în pachetul *htmltools*, funcția *htmlDependency*) care are cel puțin două avantaje:

- Poate include dintr-un singur apel mai multe foi de stiluri și scripuri Javascript.
- Nu este obligatoriu ca fișierele respective să fie în directorul *www* din cadrul directorului aplicației.

Rezultatul returnat de *htmlDependency* trebuie să fie atașat unui element HTML. Cea mai simplă soluție este să se creeze un *tagList* în care să fie incluse atât elementul, cât și dependența.

## 2. Nivel mediu

Tot în categoria HTML/CSS/JS (mai concret, JS) se încadrează și următoarele două pachete: *htmlwidgets* și *shinyjs*, pe care le-am folosit în dezvoltarea aplicației *shiny*.

Pachetul *htmlwidgets* are rolul de a integra o bibliotecă JS în R (grosier vorbind, este un fel de RPC - Remote Procedure Call). Deși, nu am realizat o astfel de sarcină, am folosit mai multe pachete care au făcut-o (pachetul *leaflet*, *DT* etc.). Din păcate, pentru *leaflet* cel puțin, nu au fost implementate toate funcționalitățile din biblioteca JS originală. Astfel, a fost nevoie de o modalitate de a putea apela funcții JS direct pe obiectul JS *leaflet* prin intermediul codului scris în R.

Pentru a rezolva acest neajuns s-a folosit pachetul *htmlwidgets* care conține funcția *onRender* și de pachetul *shinyjs* prin care serverul poate trimite cod JS clientului, pe care să-l execute.

Funcția ***onRender*** are 3 argumente în această ordine: un obiect *htmlwidget*, un string *jsCode* și un argument opțional *data* (listă cu index de tip nume). Stringul *jsCode* va conține definiția unei funcții JS care primește trei argumente (*el*, *x*, *data*) – *el* este elementul HTML corespunzător obiectului *htmlwidget*, *data* este chiar al treilea argument dat funcției *onRender*. Funcția JS va fi apelată după ce obiectul *htmlwidget* va fi afișat (în browser). În codul JS al funcției, obiectul de tip *htmlwidget* din JS este referit prin *this*. Acest lucru ne permite să avem acces în JS la obiectul creat din R. În continuare problema ar fi: dacă ulterior vom avea nevoie de această referință în JS (a obiectului creat din R), cum o vom recupera? De

menționat este faptul că *onRender* poate fi apelat de mai multe ori (spre exemplu, prima dată pentru *leaflet* și apoi, pentru a modifica ceva în acel obiect, pentru *leafletProxy*), însă efectul se va vedea doar prima dată când obiectul respectiv va fi afișat (în browser; prin apelul *leaflet*) și nu când va fi actualizat (prin apeluri *leafletProxy*). Astfel, când se apelează prima dată *onRender* pe un obiect *htmlwidgets*, este de preferat să stocăm instanța. O modalitate elegantă de stocare este următoarea:

```
$('#' + data.id).data(el, this); unde id este identificatorul (outputId) pentru elementul htmlwidget, furnizat ca parametru pentru onRender. Pentru a prelua ulterior într-un cod JS variabila stocată, se va apela:
```

```
var widget = $('#' + data.id).data();
```

În ghiduri de utilizare a pachetului *htmlwidgets* se recomandă ca cel ce integrează biblioteca JS în R să ofere o modalitate ușoară de găsim instanței. Astfel, în cazul pachetului *leaflet*, există o modalitate:

```
var map = HTMLWidgets.find('#' + id).getMap();
```

Pentru a rula cod JS în browser atunci când serverul dorește (spre exemplu, la simularea unui click sau la schimbarea opacității unui strat de date de pe o hartă – instanța hărții o vom prelua printr-una dintre metodele mai sus menționate), există cel puțin două variante:

- Varianta directă din pachetul *shiny* [22]

În ui trebuie introdus următorul script:

```
Shiny.addCustomMessageHandler(jsHandlerType, function(jsArgs)
{...});
```

Când serverul va dori să se execute la client codul ... va apela:

```
session$sendCustomMessage(type = rHandlerType, rArgs)
```

De menționat este faptul că *jsHandlerType* și *rHandlerType* reprezintă același șir de caractere, iar *rArgs* reprezintă o listă cu index de tip nume, în timp ce *jsArgs* va fi JSON-ul corespunzător.

- Varianta mai simplă din pachetul *shinyjs*

În *shinyjs*, se poate declanșa rularea de cod JS prin două modalități (înainte de toate în cadrul obiectului ui, trebuie inclus apelul la funcția *useShinyjs*):

- Prin apelul funcției ***runjs*** care primește ca argument un șir de caractere reprezentând codul JS ce se va executa la client
- Prin crearea unui fișier js (să zicem *shinyjs.js*) în care se vor scrie funcții ce se doresc a fi apelate din codul R. Va mai trebui să includem

în obiectul `ui`, apelul la funcția ***extendShinyjs*** cu primul argument setat pe calea către fișierul *shinyjs.js*. Exemplul clasic de utilizare este în Tabelul 6, unde mențiunile pentru *jsArgs* și *rArgs* rămân cele de la varianta directă prezentată mai sus:

shinyjs.js	R
<pre>shinyjs.f = function(jsArgs) {...}</pre>	<pre>js\$f(rArgs)</pre>

Tabelul 6: Exemplu utilizare *shinyjs* cu *extendShinyjs*

Uneori (de fapt, în mult mai puține cazuri), este nevoie de comunicarea în celălalt sens (de la client la server), dar nu direct prin plasarea unor elemente `ui` de tip `input` în interfață. Astfel, în `ui` trebuie introdus următorul script: `Shiny.onInputChange("id", value);`, iar în server valoarea *value* va fi preluată astfel: `input$id`. În exemplul anterior, șirul de caractere *id* poate fi înlocuit cu un altul [22].

#### vi. Modularizarea aplicațiilor *shiny* – modulele *shiny*

În cazul în care aplicația *shiny* devine din ce în ce mai complexă este bine ca aceasta să fie modularizată pentru o mai bună organizare a codului. În acest sens, vin în ajutor modulele *shiny*.

Un modul este format din două părți: `ui` și `server`. Astfel, poate fi privit ca o miniaplicație *shiny*. Șablonul de utilizare a unui modul *shiny* este prezentat în Tabelul 7.

```
library(shiny)
fUI <- function(id, ...) {
  ns <- NS(id)
  *Input/Output(ns("id"))
}
f <- function(input,output,session, ...) {
  ns <- session$ns
  #... variabile/expresii reactive/observatori
  input/output$id
  *Input/*Output(ns("id2"))
  input/output$id2
  #... variabile/expresii reactive/observatori
  return(to.return.value)
}
ui <- fUI("module")
server <- function(input,output,session) {
```

```

value <- callModule(f, "module")
}
shinyApp(ui=ui, server=server)

```

Tabelul 7: Șablonul pentru un modul R care este folosit într-o aplicație *shiny*

Avem următoarele observații de făcut:

- La nivelul cel mai de sus (ui, server pentru *shiny* app) totul devine foarte ușor de urmărit, datorită abstractizării
- În *fUI*, a fost creată variabila *ns* care are semnificația de namespace. Pentru orice element input/output folosit aici, id-ul corespunzător va trebui să fie „împachetat” într-un apel *ns*. De fapt, *ns* va fi o funcție simplă care va transforma un șir de caractere în același șir de caractere, dar precedat de șirul dat ca argument lui *NS* și de “-”. De exemplu, dacă `ns <- NS("a")` și `x <- ns("b")`, atunci *x* va fi “a-b”. Aceeași observație se păstrează și pentru server, cu excepția că variabila *ns* nu mai trebuie generată prin apelul funcției *NS*, ci trebuie doar copiată din obiectul *session*. Valoarea variabilei *ns* va fi cea din *fUI*.
- În *f*, accesarea în lista *input* a elementului cu id-ul `ns("id")` se face astfel: `input$nsid`. Acest lucru este posibil, pentru că *f* nu a fost apelată ca o funcție normală din R, ci s-a folosit funcția *callModule*.
- În general (dacă în id-uri nu folosim caracterul “-”), cât timp nu avem conflicte de nume la nivel de modul sau la nivelul cel mai de sus al aplicației, sigur nu vom avea conflicte de nume de-a lungul rulării aplicației.
- Pentru o comunicare între nivelul cel mai de sus al aplicației și un modul, cel mai probabil se va alege ca funcția de tip server din modul să returneze o expresie reactivă (nu rezultatul apelului ei).
- Dacă se dorește ca funcția de tip server din modul să primească o variabilă reactivă (din lista *input*) ca parametru, atunci se va proceda astfel: `callModule(f, "module", reactive(input$X))` și nu simplu: `callModule(f, "module", input$X)`.
- Un modul poate apela un alt modul, însă trebuie ca, în funcția *ui* din modulul care apelează, pentru orice element input/output, id-ul corespunzător să fie „împachetat” într-un apel *ns*.

### vii. Pachete complementare *shiny*

Odată cu apariția pachetului *shiny*, au apărut și alte pachete în strânsă legătură care vin cu noi funcționalități. Pachetele complementare *shiny* și folosite în aplicație au fost:

- **shinydashboard**: folosit pentru un design în stilul unui dashboard
- **shinytoastr**: folosit pentru trimiterea dispre server spre client a notificărilor de patru tipuri (folosește biblioteca *toastr* din JS): succes, eroare, informație, avertisment
- **shinyBS**: folosit pentru posibilitatea adăugării de tooltip-uri (Bootstrap)
- **shinyjs**: folosit, pe lângă mențiunile de mai sus, pentru a ascunde sau a face vizibil un element input *shiny* (funcțiile *hide*, *show* și *toggle* cu argumentul *condition*)
- **shinyjsky**: folosit pentru afișarea unei imagini de încărcare pe partea de client, atunci când serverul are instrucțiuni de executat
- **shinyWidgets**: folosit pentru butoane radio mai atractive din punctul de vedere al aspectului și pentru inputul de tip *select* care permite, pentru opțiunea *multiple*, selectarea tuturor opțiunilor și deselectarea tuturor opțiunilor
- **colourpicker**: pentru inputul de tip culoare

### 3. Alte pachete folosite

În afară de *shiny* și pachetele complementare lui, în cadrul aplicației s-au folosit și următoarele:

- **leaflet**: pentru folosirea hărților interactive direct din R; funcțiile relevante folosite au fost
  - *leafletProxy* (pentru modificarea unei hărți deja afișate), *setView* (pentru a seta centrul și zoomul hărții), *addTiles* (pentru a adăuga hărți geografice din diverse surse specificate de utilizator), *addProviderTiles* (pentru a adăuga hărți geografice din surse deja existente în pachet), *addControl* (pentru a adăuga elemente HTML deasupra hărții), *addScaleBar* (pentru a adăuga scala hărții), *addLayersControl* (pentru manipularea straturilor adăugate pe hartă: harta geografică de fundal sau datele spațiale), *fitBounds* (pentru a încadra harta într-un chenar specificat prin latitudini și longitudini), *clearGroup* (pentru a elimina un strat de date de pe hartă; ca observație, *group* reprezintă o etichetă partajată pentru mai multe elemente de pe hartă, iar *layerId* reprezintă o

etichetă pentru un element de pe hartă; ambele pot fi atribuite elementelor ce sunt adăugate pe hartă), *removeControl* (pentru a șterge elementele HTML adăugate prin *addControl*), *addPolygons* (pentru a adăuga zone pe hartă), *addLegend* (pentru a adăuga o legendă pe hartă sub forma unei palete de culori în dégradé), *addCircleMarkers* (pentru a adăuga puncte pe hartă; există opțiunea de clusterizare a punctelor:

- algoritmul de clusterizare pentru un zoom este unul de tip greedy (este nevoie de o rază  $r$  prestabilită) [48]:
  - inițial, nu există clustere
  - se alege un punct aleator
  - toate punctele (care nu aparțin deja unui cluster) aflate la o distanță mai mică decât  $r$  față de punctul ales formează un cluster
  - procesul se repetă până când toate punctele aparțin unui cluster
  - la trecerea mouse-ului peste un cluster (reprezentat prin cerc), apare înfășurătoarea convexă a clusterului respectiv care mereu este un poligon convex; suprapunerea (uneori) acestor zone apărute se explică tocmai prin algoritmul greedy de clusterizare (dacă un punct este într-un cluster, se poate să aparțină și înfășurătorii convexe corepunzătoare altui cluster)
- algoritmul de clusterizare per total este denumit *greedy ierarhic*, pentru că inițial se formează clusterul greedy la cel mai mare zoom, iar apoi pentru fiecare cluster format va exista un centroid care va ține loc de punct pentru un zoom mai mic cu o unitate decât cel actual; pentru acest zoom mai mic se face clusterizarea greedy și se continuă astfel până la cel mai mic zoom)
- **leaflet.extras**: pentru funcția *addFullscreenControl* aplicabilă unui obiect leaflet
- **DT**: pentru vizualizarea tabelor în interfață

- **kohonen**: pentru rularea algoritmului SOM, pentru desenarea ploturilor și pentru calculul erorii medii de cuantizare
- **RMySQL**: pentru conectarea și interogarea bazelor de date MySQL
- **sqldf**: pentru transformarea unui tabel (*data.frame*) din R într-o tabelă într-o bază de date SQLite stocată în memoria RAM; s-a realizat acest lucru pentru filtrarea prin SQL a datelor din obiectele *data.frame* (de către utilizatorul aplicației)
- **pool**: pentru crearea unui pool de conexiuni la o bază de date
- **data.table**: pentru filtrări (agregări) mai ușoare a unor obiecte *data.frame*
- **rgeos** și **sf**: pentru convertirea din șiruri de caractere (în format WKT - Well-known text; exemplu: POINT (30 10)) în obiecte spațiale
- **ggmap** + **ggplot2**: pentru hărți statice pe care au fost plasate doar date de tip punct (longitudine, latitudine)
- **dygraphs**: pentru grafice interactive
- **xts**: pentru convertirea într-un obiect de tip serie de timp

#### 4. Chestiuni relevante în R

Pentru dezvoltarea unei aplicații (nu numai *shiny*) în R, următoarele idei trebuie cunoscute.

**Evaluarea argumentelor unei funcții în R este lazy** [16]: la apelul unei funcții, argumentele nu sunt evaluate (argumentul neevaluat se numește promisiune (Promise) și stochează de fapt expresia ce a fost primită ca argument și environmentul în care expresia a fost creată și în care va fi evaluată). Abia prima dată când o promisiune este accesată în cod, expresia este evaluată în environmentul corespunzător, iar rezultatul este reținut, pentru ca, la noi acceasări ale promisiunii, rezultatul să fie returnat imediat. Astfel,

```
f<-function(x) {}
f(v)
```

nu va furniza nicio eroare chiar dacă nu există variabila *v* în environmentul curent, iar

```
f<-function(x) {x}
f(v)
```

va furniza o eroare dacă nu există variabila *v* în environmentul curent pentru că expresia *v* va fi evaluată în cadrul funcției *f*. Probleme apar de obicei în cazul închiderilor (Closures). În *shiny*, avem de-a face cu închideri, deci, este bine să nu uităm de acest detaliu. Exemplul cel mai relevant este atunci când se apelează funcția *callModule* într-un for, după cum se poate vedea în Tabelul 8.



```

library(shiny)
fUI <- function(id) {
  ns <- NS(id)
  actionButton(ns("go"), "Go")
}
f <- function(input, output, session, parameter) {
  # parameter
  # force(parameter)
  observeEvent(input$go, {
    print(parameter)
  })
  return()
}
ui <- fluidPage(
  fUI("1"),
  fUI("2")
)
server <- function(input, output) {
  for(i in 1:2) {
    callModule(f, as.character(i),
              parameter = i)
  }
}
shinyApp(ui=ui, server=server)

```

Tabelul 8: Exemplu de problemă la apelarea *callModule* într-un for. Prin apăsarea oricărui buton, la consolă se va afișa 2, ceea nu era de așteptat (abia când se va ajunge la linia de cod *print(parameter)* se va evalua expresia *i* din env. curent, care va fi 2 la acel moment). Dacă vom comenta prima linie din funcția *f*, comportamentul este cel așteptat, pentru că promisiunea *parameter* va executa și reține expresia *i* din env curent (1 pentru primul buton, 2 pentru al doilea). O altă soluție este să comentăm doar a doua linie din funcția *f*. Această soluție este preferată, pentru că se va cunoaște intenția programatorului (de a evalua imediat expresia data ca argument) care nu va fi confundată cu o greșeală (de a scrie, simplu, *parameter* pe o linie).

Dacă avem următoarele linii de cod: *b<-1*, *a<- b*, atunci în *a* nu se copiază tot obiectul *b*, ci *a* va ști că valoarea sa se află în locația de memorie în care este stocat *b*-ul (*a* memorează în spate doar un pointer către *b*). Totuși, dacă în continuare, *b*-ul se modifică (*b<-2*), atunci *a* va copia conținutul anterior al lui *b* (1). Acest lucru se întâmplă și la prima accesare a promisiunilor (grosier spus, parametrii formali ai funcției nu copiază parametrii actuali, ci, mai degrabă țin o referință către ele). Acest mecanism se numește **copy-on-modify** și se aplică majorității obiectelor R.

O ultimă idee este faptul că orice proces R este **single-threaded**. Astfel, procesul în care rulează aplicația *shiny* este și el single-threaded. Dacă, spre exemplu, un utilizator intră în aplicație și pornește, prin interfață, o funcție pe server care durează 20 de secunde, iar un alt utilizator intră pe site-ul aplicației în timp ce acea funcție se execută, atunci cel din urmă utilizator va trebui să aștepte ca funcția să se termine de executat, pentru a se putea conecta la aplicație. O soluție ar fi ca pentru fiecare utilizator (sau pentru fiecare *n* utilizatori) să avem

câte un proces corespunzător unei aplicații *shiny*. Acest tip de soluție este întâlnit în programul *Shiny Server Pro* care are rolul de a facilita lansarea aplicațiilor *shiny*.

## II. Ghid de utilizare a aplicației și alte mențiuni legate de implementare/arhitectură

### 1. Mini-arhitectura aplicației

Aplicația a fost structurată în mai multe module *shiny*:

- **csvFile**: folosit pentru încărcarea datelor în aplicație din fișiere CSV/TSV
- **queryDb**: folosit pentru încărcarea datelor în aplicație dintr-o bază de date MySQL
- **databox**: folosit pentru crearea unei structuri pentru fiecare tabel încărcat în aplicație (funcționalități: previzualizare/vizualizare tabel, filtrare, sumarizare date)
- **layeredMap**: folosit pentru crearea unei hărți interactive peste care se pot adăuga diverse straturi (rezultate din datele încărcate în aplicație; datele încărcate în aplicație vor trebui să fie neapărat date spațiale)
- **postfilter**: folosit pentru filtrarea datelor deja existente ca straturi pe hartă
- **countiesAggregates**: folosit pentru crearea unei hărți interactive peste care să apară doar regiunile de tip county din SUA, colorate după date preexistente (rezultatul modulului poate fi privit ca o zonă de testare/încercare a aplicației în cazul în care utilizatorul nu dispune de date)
- **countiesAggregates2**: folosit pentru crearea unei hărți interactive peste care să apară doar regiunile de tip county din SUA, colorate după date spațiale (de tip latitudine, longitudine); în cadrul acestui modul a fost nevoie de mai multe funcții de agregare; pentru a defini mai multe funcții de agregare manual, se va deschide directorul *functions* din directorul asociat modulului, iar în fișierul *functions.txt* se vor introduce numele funcțiilor de agregare din R (funcția trebuie să permită două argumente: un vector de date asupra căruia se face agregarea și *na.rm* cu semnificația uzuală); dacă se doresc a fi implementate alte funcții care să fie referite în *functions.txt*, acestea pot fi definite în fișierul *other\_functions.R*

Fiecare modul a fost plasat într-un director separat care conține, în general, următoarele:

- *numeModulGlobal.R* – echivalentul lui *global.R* pentru un modul
- *numeModulServer.R* – funcția server din cadrul modulului
- *numeModulUI.R* – funcția ui din cadrul modulului

- *numeModulModule.R* – conține 4 linii de cod: memorarea într-o variabilă a directorului modulului și trei apeluri source: la *numeModulGlobal.R*, *numeModulServer.R*, *numeModulUI.R*
- directorul *src*
  - fișiere R/js ajutătoare (funcții în special)
  - directorul *server* în care există, de obicei, câte un fișier pentru fiecare expresie reactivă/observator din acest modul; aceste fișiere au fost incluse în *numeModulServer.R* prin intermediul funcției *source* (`[[1]]`) a fost folosit pentru că *source* returnează o listă din care doar primul element ne interesează):

```
observe(Event)/reactive/eventReactive/render*(... ,{
source(paste0(numeModulModule.wd, "/src/server/fișier.R"), local = TRUE)[[1]]
})
```

La nivelul cel mai de sus al aplicației există următoarele:

- *global.R*
- *ui.R* și fișiere *ui\_\*.R*
- *server.R*
- *credentials.cnf* – conține datele de conectare la baza de date MySQL
- directorul *R*
  - directorul *shiny*
    - directorul *modules* – conține modulele folosite, fiecare într-un director separat
    - directorul *server* – în care există, de obicei, câte un fișier pentru fiecare expresie reactivă/observator folosită la acest nivel al aplicației
  - directorul *src* – conține fișiere cu funcții folosite la acest nivel al aplicației
- directorul *www*
  - directorul *img* – pentru imaginile din aplicație
  - directorul *src* – pentru fișiere CSS/JS necesare

Pentru integrarea unui modul s-a procedat astfel:

- dacă modulul respectiv utiliza pachetul *toastr*, atunci în *ui* s-a introdus apelul la *useToastr*

- dacă modulul respectiv utiliza pachetul *shinyjs*, atunci în ui s-a introdus apelul la *useShinyjs*; în *global.R* a fost furnizată calea către fișierul js (cu funcțiile pentru *shinyjs*) din directorul modulului respectiv pentru a se crea un singur fișier a cărui cale să fie dată ca parametru funcției *extendShinyjs*
- dacă modulul respectiv a inclus imagini în aplicație de pe mașina locală, atunci imaginile au fost plasate în directorul *www/img*
- în *global.R* a fost inclusă următoarea linie de cod:  

```
source("R/shiny/modules/numeModulModule/numeModulModule.R",
local = TRUE, chdir = TRUE). Parametrul local a fost folosit pentru
consistență (la orice apel al funcției source a fost utilizat). Dacă este setat pe
TRUE, atunci codul R din fișierul de la calea respectivă este executat în
environmentul local și nu în cel global (doar că în acest caz, cele două
coincid). Parametrul chdir a fost folosit pentru a se schimba pe moment
directorul de lucru. Fișierul numeModulModule.R conține ca primă linie de
cod: numeModulModule.wd <- getwd() ,
iar variabila numeModulModule.wd va fi folosită în codul modulului
numeModul, deci, are sens ca parametrul chdir al funcției source să fie setat pe
TRUE.
```

## 2. Aplicația pas cu pas

Înainte de rularea aplicației, în cazul în care datele de conectare la serverul de baze de date MySQL trebuie schimbate, se vor modifica informațiile din fișierul *credentials.cnf*.

### i. Available Data

După rularea aplicației, orice client se poate conecta, rezultatul fiind, spre exemplu cel din Figura 8:

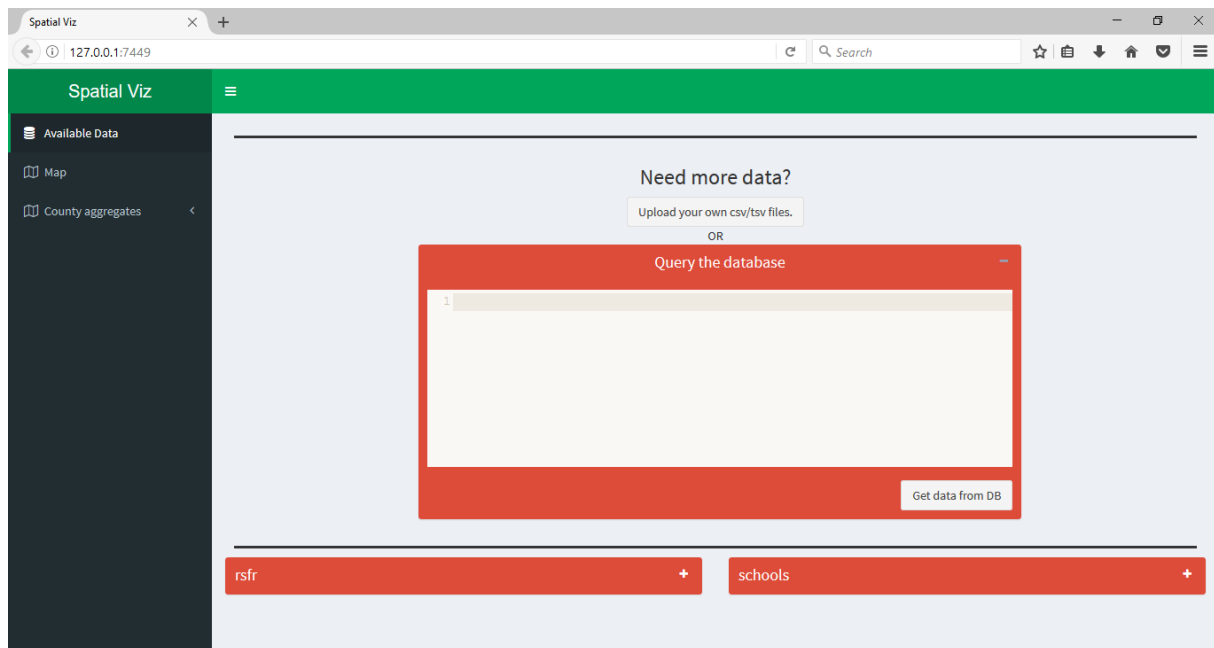


Figura 8: Ce vede clientul când se conectează la aplicație

Se poate observa că în partea de jos a paginii au apărut două bare, fiecare cu câte un titlu. De fapt, a fost interogată schema bazei de date MySQL (cea referită în *credentials.cnf*), iar tabelele existente au fost afișate în roșu.

De aici, utilizatorul este ghidat ce să facă pentru a adăuga date în aplicație. Utilizatorul are două opțiuni: să încarce fișiere CSV/TSV și să interogheze baza de date care conține tabelele apărute deja cu roșu în partea de jos a paginii.

Dacă se dorește încărcarea unui fișier CSV/TSV, atunci utilizatorul face click pe butonul *Upload your own csv/tsv files.* (vezi Figura 9). El va fi întrebat de calea către fișierul de încărcat, apoi de opțiuni ale fișierului (dacă primul rând din datele încărcate trebuie să fie considerat antet de tabel și dacă este vorba despre un fișier CSV sau TSV), va fi anunțat că se poate ca unele rânduri sau coloane să fie eliminate<sup>4</sup> și va fi notificat dacă s-au făcut astfel de eliminări din date. În continuare, utilizatorului îi apar maxim 6 rânduri din tabelul tocmai încărcat și i se cere să schimbe tipul coloanelor, în cazul în care nu este mulțumit de tipurile deja identificate (implicit, tipurile identificate sunt doar *Character* și *Numeric*). Apare și o notificare despre cum se poate specifica tipul unei coloane (*Numeric*, *Character*, *Date* urmat

<sup>4</sup> Eliminările sunt necesare pentru a avea date care să se comporte normal în cadrul aplicației. Dacă există nume de coloane duplicate sau numele unei coloane este *\_rowid\_*, se afișează un mesaj de eroare. Se elimină liniile care prezintă doar valori NA pe fiecare coloană. Se elimină coloanele care prezintă valori NA pe fiecare linie sau care prezintă o singură valoare pe toate liniile. Dacă în urma eliminării nu mai există date (0 linii sau 0 coloane) se afișează un mesaj de eroare.

### Step 1: Choose a csv/tsv file

Browse...

No file selected

Cancel

Next

### Step 2: Choose file options

☒ Header

**Separator**  
☒ Comma  
☐ Tab

We will remove some rows and columns with missing values...

Cancel

Previous

Next

### Data information

**Data ID:**

**Data description:**

**Type of data**  

☒ Point
 ☐ Zone

**Latitude column name**

**Longitude column name**

Cancel

Go

### Step 3: Modify existing column types

mx_id	nces_disid	nces_schid	urbanrural	students
Character	Numeric	Character	Numeric	Numeric
WA-AUBURN S-17426-WA-PB90919	5300300	5300300000029	21	469
WA-AUBURN S-17426-WA-PB90923	5300300	5300300000034	13	317
WA-AUBURN S-17426-WA-PB90924	5300300	5300300000035	13	432
WA-AUBURN S-17426-WA-PB90925	5300300	5300300000036	13	363
WA-AUBURN S-17426-WA-PB90926	5300300	5300300000037	13	464
WA-AUBURN S-17426-WA-PB90927	5300300	5300300000038	13	683

Possible types:
 

- Numeric
- Character
- Date format (eg: 'Date %Y-%m-%d')

Possible date formats (all R date formats, including):
 

- %d - Day of the month (01-31)
- %m - Month (01-12)
- %y - Year without century (00-99)
- %Y - Year with century

Columns that were dropped since they were empty / constant:  
 gender, state, statefips, county, countyfips, relver, yearclass, ext\_relver, relver.L

Figura 9: Interfața cu utilizatorul când este apăsat butonul *Upload your own csv/tsv files*.

de format).<sup>5</sup> Până acum, utilizatorul a putut reveni la pasul anterior. Apoi, acestuia i se cere un identificator, o descriere și un tip pentru date. Regulile de specificare a identificatorului și la ce va fi el de folos apar ca o notificare. La alegerea tipului de date, există două opțiuni: *Point* și *Zone*. Scopul aducerii datelor în aplicație este plasarea lor pe hartă, deci, are sens ca datele acceptate să conțină coloane prin care să poată fi introduse pe hartă. Dacă se alege *Point*, trebuie să se specifice numele coloanelor care dau Longitudinea și Latitudinea. Dacă se alege *Zone*, trebuie să se specifice numele coloanei care conține șiruri de caractere ce ar reprezenta date spațiale în formatul WKT.

Dacă se dorește preluarea de date din baza de date, atunci se vor interoga tabelele existente din josul paginii acolo unde scrie *Query the database*. Mai întâi, este recomandabil ca utilizatorul să își amintească structura tabelelor, dând click pe plusul din dreapta unei bare roșii (obținând informații despre tabelul respectiv; vezi Figura 10). De aici, se pot obține următoarele informații: descrierea (dată de comentariul introdus la crearea tabelului în BD), previzualizarea tabelului (maxim 10 linii), datele rezultate în urma filtrării printr-o clauză WHERE din SQL (pentru a filtra datele, în tabul *Filter*, se va da click pe butonul *Go*; dacă butonul *Filter* este setat pe ON, filtrarea va avea loc, iar dacă este setat pe OFF, clauza WHERE nu se va lua în seamă: interogarea va fi `SELECT * FROM tabel`). Utilizatorul este informat în legătură cu aceste lucruri.

După ce utilizatorul investighează tabelele, și-a amintit lucruri despre informația cuprinsă în BD și poate aduce în aplicație datele dorite (vezi Figura 11). În continuare, pașii ce trebuie urmați de utilizator seamănă cu cei din cazul preluării datelor din fișiere CSV/TSV. O diferență este faptul că, dacă în BD, există coloane ale căror nume conțin șirul de caractere *Date* (case-sensitive), atunci acele coloane vor fi automat convertite la tipul *Date* cu formatul `%Y-%m-%d`.

Datele din aplicație vor fi afișate în interfața grafică imediat sub tabelele din BD (după barele roșii). Cele din fișiere CSV/TSV sunt bare de culoare albastră, iar cele din BD sunt bare de culoare verde. Datele de tip *Point* sunt diferențiate de cele de tip *Zone*, prin adăugarea șirului de caractere (*p*) după identificatorul înscris pe bară (cele de tip *Zone*, prezintă sufixul (*z*)). Ceea ce apare nou în expandarea barelor este faptul că datele nu mai sunt previzualizate în maxim 10 rânduri, ci întregul tabel poate fi vizualizat. Pentru că datele temporale și cele spațiale ne vor interesa pentru mai târziu, la vizualizarea datelor (tabul *Data*), coloanele

---

<sup>5</sup> În plus, dacă nu există nicio coloană numerică se afișează o eroare. Dacă facem o proiecție doar pe coloanele numerice, în tabelul rezultat nu acceptăm linii doar cu NA-uri și, deci, liniile corepunzătoare din tabelul original vor dispărea. Dacă în urma eliminării nu mai există date (0 linii) se afișează un mesaj de eroare.



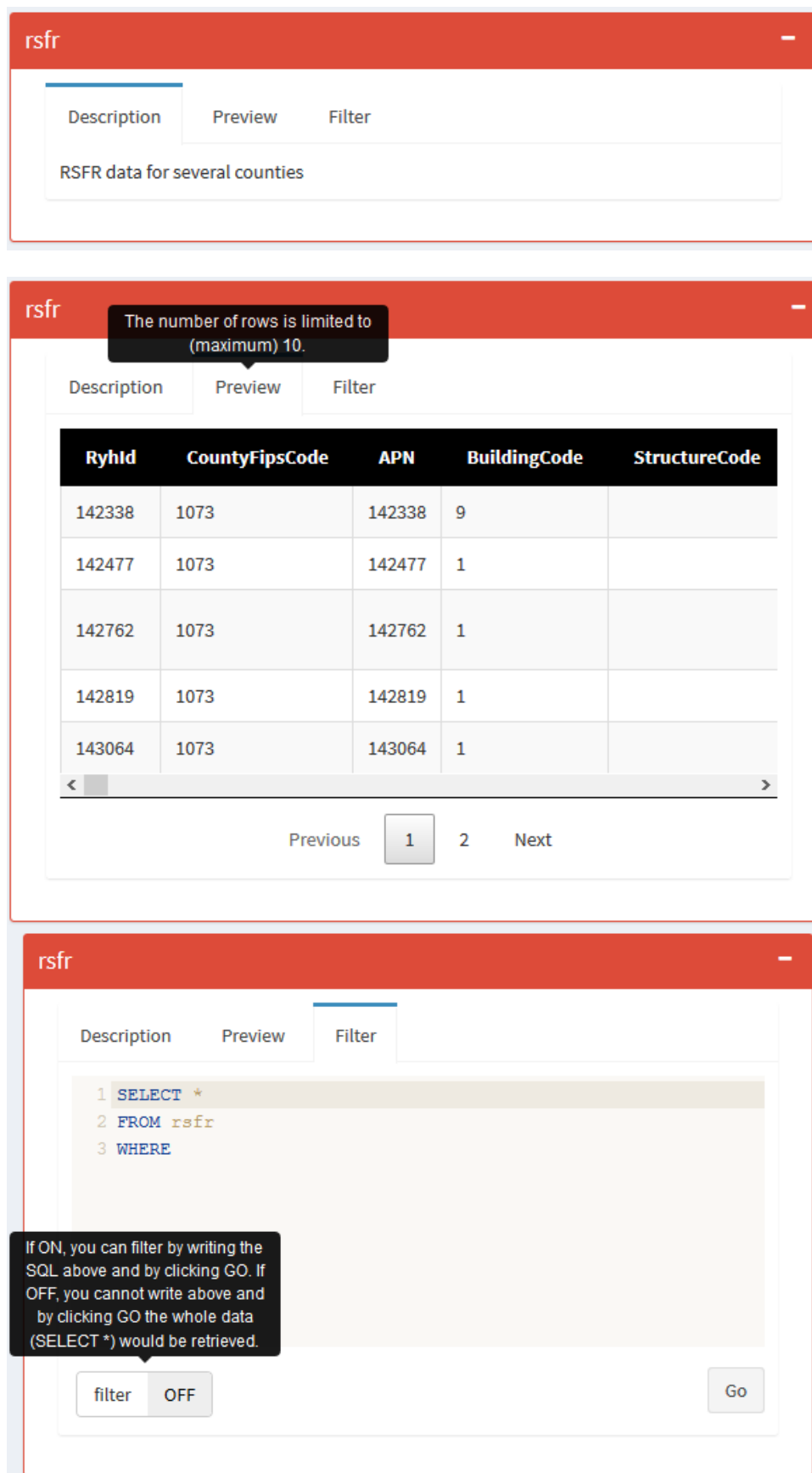


Figura 10: Interfața cu utilizatorul când este apăsat butonul + al unei bare roșii

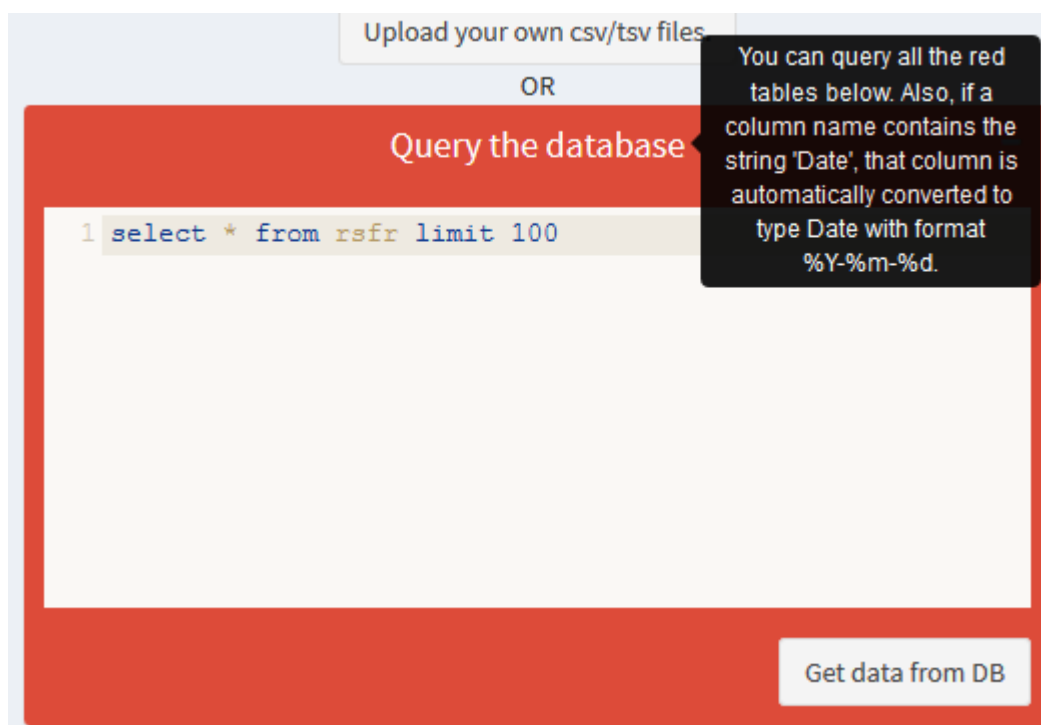


Figura 11: Interfața cu utilizatorul când se dorește a se prelua date din BD

scoli (z)

Description	Data	Summary	Filter
mx_id	nces_disid	nces_schid	
Length:376	Min. :5300001	Length:376	
Class :character	1st Qu.:5303540	Class :character	
Mode :character	Median :5304230	Mode :character	
	Mean :5304786		
	3rd Qu.:5307710		
	Max. :5308760		
students	teachers	kinder	fir
Min. : 0.0	Min. : 0.00	Min. : 0.00	Min.
1st Qu.: 412.0	1st Qu.:22.48	1st Qu.: 0.00	1st Qu.
Median : 516.5	Median :27.05	Median : 60.00	Median
Mean : 622.4	Mean :31.21	Mean : 48.84	Mean
3rd Qu.: 664.5	3rd Qu.:34.12	3rd Qu.: 77.25	3rd Qu.
Max. :2224.0	Max. :96.80	Max. :145.00	Max.

Figura 12: Interfața cu utilizatorul când se dorește un sumar al unui tabel din aplicație

temporale au fost marcate cu albastru, iar cele spațiale, cu verde. Totodată, apare un nou tab (*Summary*) prin care se afișează sumarul datelor (vezi Figura 12).

Tabelele adăugate sunt reținute într-o listă *filtered.reactives*, iar informațiile despre tabele sunt reținute într-un *data.frame* denumit *session.tables.info* cu următoarele coloane: *table\_id*, *table\_desc*, *table\_type* (p sau z), *table\_col1* (numele coloanei ce reprezintă longitudinea – pentru tipul *p* – sau șirul de caractere corespunzător tipul *z*), *table\_col2* (numele coloanei ce reprezintă latitudinea – doar pentru tipul *p*).

## ii. Map

În continuare, dând click pe Tabul *Map* din partea stângă a aplicației, se obține următorul rezultat (Figura 13):

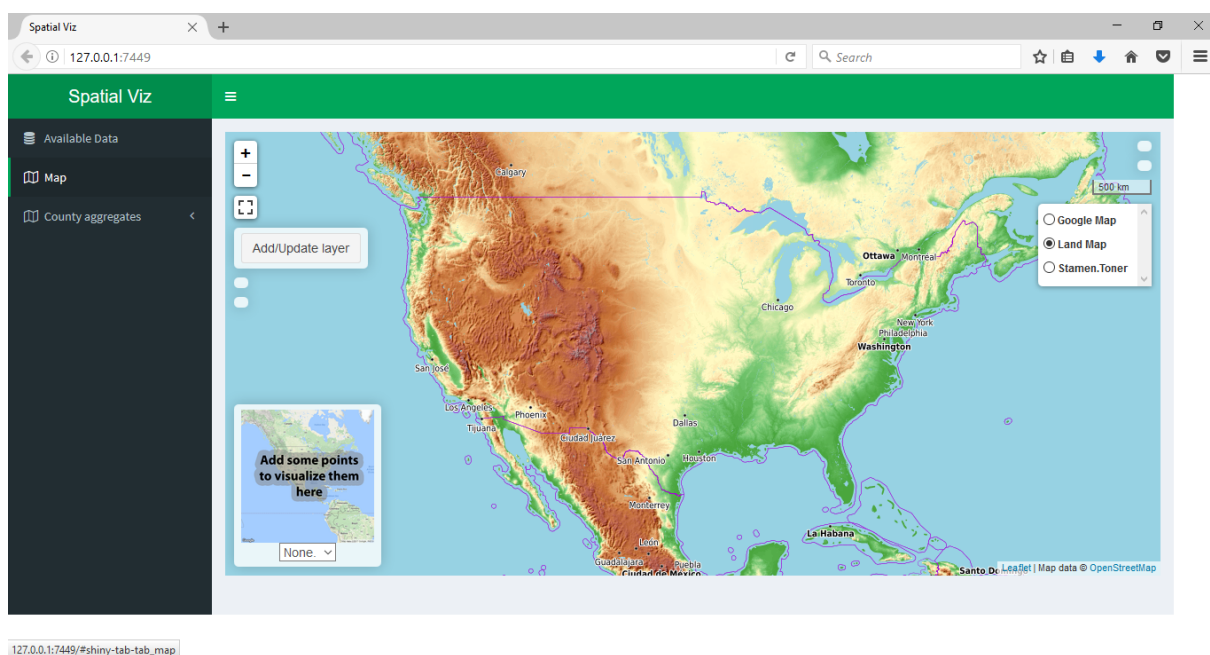


Figura 13: Inițial, tabul *Map*

Deocamdată în partea din dreapta sus, există un control pentru a seta harta de fundal (la adăugarea de straturi pe hartă, numele acestora vor apărea tot aici și va exista posibilitatea ascunderii lor sau a reafîșării lor; în plus, prin ascunderea și reafîșarea unui strat, acesta va deveni cel mai de sus – relevant când există straturi suprapuse). În partea din stînga sus, există controale de Zoom in/out și pentru modul Fullscreen. În partea din stînga jos, pot fi vizualizate imagini cu o lupă, dacă se trece cu mouse-ul peste imaginea în miniatură (pentru zoom in/out se va folosi mouse-ul). Acest control va avea sens atunci când vor fi adăugate date de tip *Point* pe hartă. Acestea vor fi afișate în clustere (de la un număr de puncte în sus, browserul ocupă prea multă memorie pentru afișarea punctelor, dacă ele nu ar fi clusterizate), iar, ca ajutor pentru utilizatorul care vrea să vadă imaginea de ansamblu a datelor de tip *Point* neclusterizate, în controlul de care vorbim va apărea această imagine de ansamblu

(cuprinzând doar un strat de pe hartă). Imaginile acestea de ansamblu vor fi reținute pe parcurs, în așa fel încât dacă utilizatorul adaugă mai multe date de tip *Point* pe hartă, poate reveni la oricare imagine de ansamblu prin controlul de tip select-input aflat în josul imaginii de tip miniatură.

Butonul *Add/Update layer* apare doar dacă există date în aplicație și are rolul de a adăuga datele pe hartă. Se alege identificatorul pentru tabelul din aplicație, apoi se aleg coloanele numerice după care se dorește ca datele să fie colorate. În cazul în care numărul de coloane selectat este 1 (vezi Figura 14), atunci se va crea o paletă de culori în degradé cu care vor fi colorate datele. Utilizatorul poate să introducă culorile de la capete și cele intermediare utilizate în generarea paletelor. Dacă numărul de coloane selectat este mai mare decât 1 (vezi Figura 15), atunci se va aplica algoritmul SOM peste tabelul original proiectat pe coloanele selectate. Utilizatorul va trebui să furnizeze numărul de linii și coloane pentru grilă (implicit [39], numărul de noduri este  $5 * \text{numărul de linii din tabel}$ , numărul de linii a fost considerat egal cu numărul de coloane, iar numărul de iterații este  $500 * \text{numărul de linii} * \text{numărul de coloane}$ ; la o modificare a numărului de linii/coloane, se va actualiza automat valoarea pentru numărul de iterații) și să aleagă o paletă de culori 2D prestabilită. Utilizatorului i se sugerează faptul că ar trebui să aleagă paleta, știind că probabil instanțele cu valori maxime ale atributelor se vor îndrepta spre colțul stânga sus, iar instanțele cu valorile minime ale atributelor se vor îndrepta spre colțul dreapta jos. Spunem *probabil*, pentru că s-a folosit o euristică (în unele cazuri funcționează, alteori, nu): inițializăm ponderile nodului din stânga sus cu valorile maxime ale tuturor atributelor, iar ponderile nodului dreapta jos cu valorile minime ale tuturor atributelor. În continuare, utilizatorul poate selecta numele unor coloane de tip *Character*. Valorile acestora, corespunzătoare unei instanțe, vor fi afișate (pe lângă coloanele numerice alese și coloanele temporale) pe hartă când se va click pe forma geometrică respectivă.

Exemple de hărți cu date adăugate ca straturi sunt prezente în figurile 16 și 17.

Datele din *Available Data* pot fi filtrate ulterior și pot fi adăugate pe hartă direct filtrate. De aceea, eliminări de linii în stilul celor ce apar la preluarea datelor în aplicație sunt posibile și aici.

Imediat se observă că au apărut noi controale. În stânga au apărut butoanele *Remove layer* (prin care se poate șterge un strat de pe hartă), *Fit bounds* (prin care un set de date este

**Data ID**  

scoli

**Numeric non-constant variables:**  

students

**Choose the colors to generate the palette (from lowest to highest):**  

#FFFF00

#FF0000

Add color

Delete color

**Text variables (will appear in data popups):**  

Nothing selected

Cancel

SOM it!

**Data ID**  

scoli

**Numeric non-constant variables:**  

urbanrural, students

**Number of lines for grid:**  

9

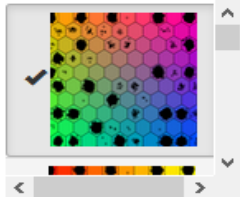
**Number of columns for grid:**  

9

**Number of iterations:**  

40500

**Choose colors:**  



**Text variables (will appear in data popups):**  

Nothing selected

Cancel

SOM it!

Figura 14 (stânga): Interfața cu utilizatorul când se adaugă date ale aplicației pe hartă (# coloane = 1)

Figura 15 (dreapta): Interfața cu utilizatorul când se adaugă date ale aplicației pe hartă (# coloane > 1)

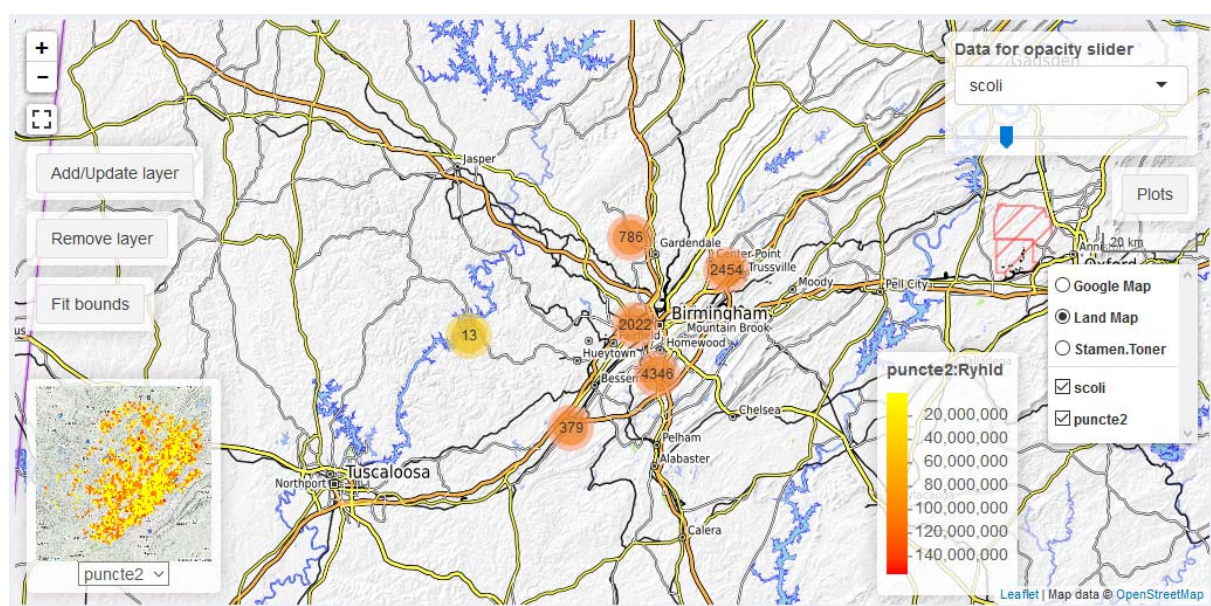


Figura 16: Interfața cu utilizatorul când s-au adăugat date ale aplicației pe hartă (# coloane = 1, tip Point)

vizualizat complet pe hartă). Controlul hărții de fundal a fost augmentat. În cazul în care nu s-a aplicat SOM, a apărut o legendă (cuprinzând scala de culori) care poate fi mutată, iar în celălalt caz a apărut butonul *Plots* în dreapta. Datele tocmai adăugate pe hartă au fost colorate după acestea. Dând click pe *Plots* vor putea fi vizualizate grafice (despre care s-a vorbit în secțiunea I.1.iii) și vor putea fi cunoscute valorile erorilor (discutate în secțiunea I.1.v). Dacă datele adăugate sunt de tip *Point*, imaginea din stânga jos a fost actualizată corespunzător. În dreapta sus, a apărut un control prin care se poate modifica opacitatea fiecărui strat de pe hartă (în Figura 17, opacitatea este mai mică decât valoarea implicită). Dând click pe o figură geometrică de pe hartă (fie că este punct sau zonă), apar valori ale instanței pentru atributele după care s-a făcut colorarea (sunt marcate cu portocaliu), atributele temporale (marcate cu albastru) și atributele textuale selectate.

Pe o hartă, un set de date poate apărea o singură dată. Există nevoia de a aplica algoritmul SOM de mai multe ori (de obicei pe atribute diferite), iar astfel pe hartă apar datele colorate după ultimul SOM. Dacă se dorește a se reveni la o colorare printr-un SOM deja executat, s-ar reexecuta algoritmului SOM. Atunci când numărul de iterații este mare, rularea algoritmului durează. Ca soluție, s-a implementat un mecanism de reținere a rezultatului ultimului obiect SOM pentru un tabel dat și pentru un set de coloane dat. Astfel, dacă utilizatorul ajunge în interfață să aleagă să coloreze datele după un SOM deja executat, el este întrebat dacă dorește să-l utilizeze pe acela sau nu (îi sunt afișate și informații relevante despre obiectul SOM: număr linii, număr coloane, număr iterații, paleta de culori, cele două erori ale modelului).

### iii. Postfilter

Dacă pe hartă există măcar un strat adăugat, în stânga apare și un nou tab: *Postfilter*. Rolul acestuia este de a filtra datele dispuse deja pe hartă.

Odată cu adăugarea unui strat pe hartă, apare și o secțiune în tabul *Postfilter* corepunzătoare stratului respectiv (vezi Figura 18). Dacă tabelul corespunzător are date temporale, atunci va apărea un grafic. Pe axa Ox va putea fi reprezentat orice atribut numeric după care s-a făcut colorarea, iar pe axa Oy va putea fi reprezentat orice atribut temporal. Graficul este interactiv și se poate selecta o perioadă de timp (zoom in; pentru zoom out se va da un dublu click pe grafic). O perioadă de timp este reținută pentru fiecare coloană temporală. Aceste perioade de timp vor genera o parte dintr-o interogare SQL, afișată sub grafic. Modificări posibile ale interogării sunt asupra datelor (exemplu: textul *1958-02-06* va



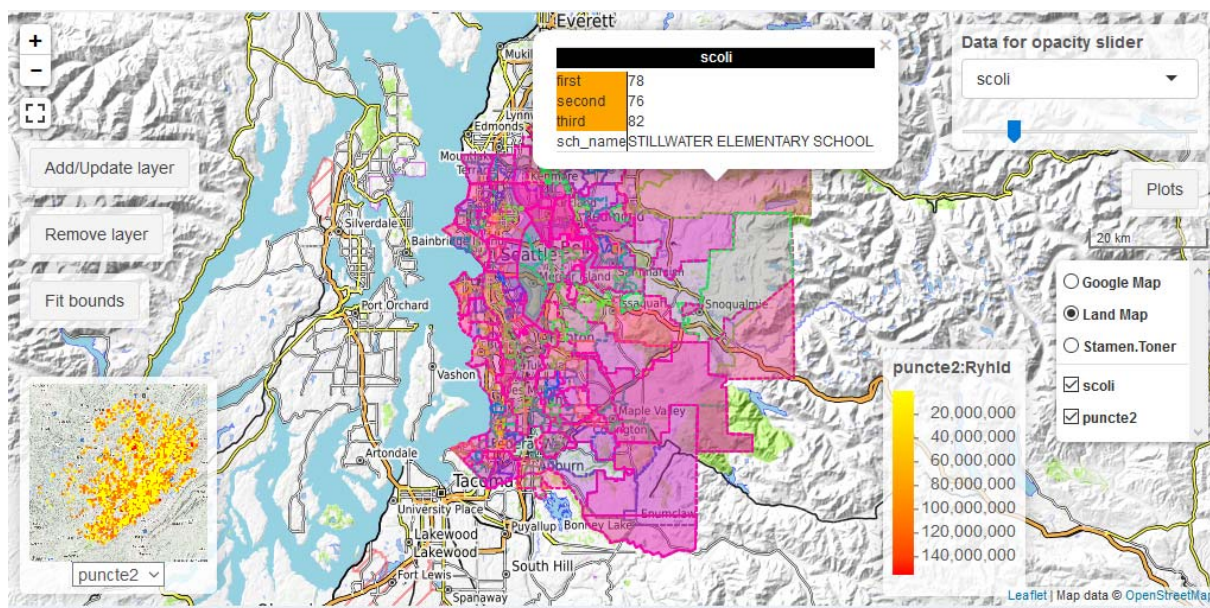


Figura 17: Interfața cu utilizatorul când s-au adăugat date ale aplicației pe hartă (# coloane > 1, tip Zone, opacitate scăzută)

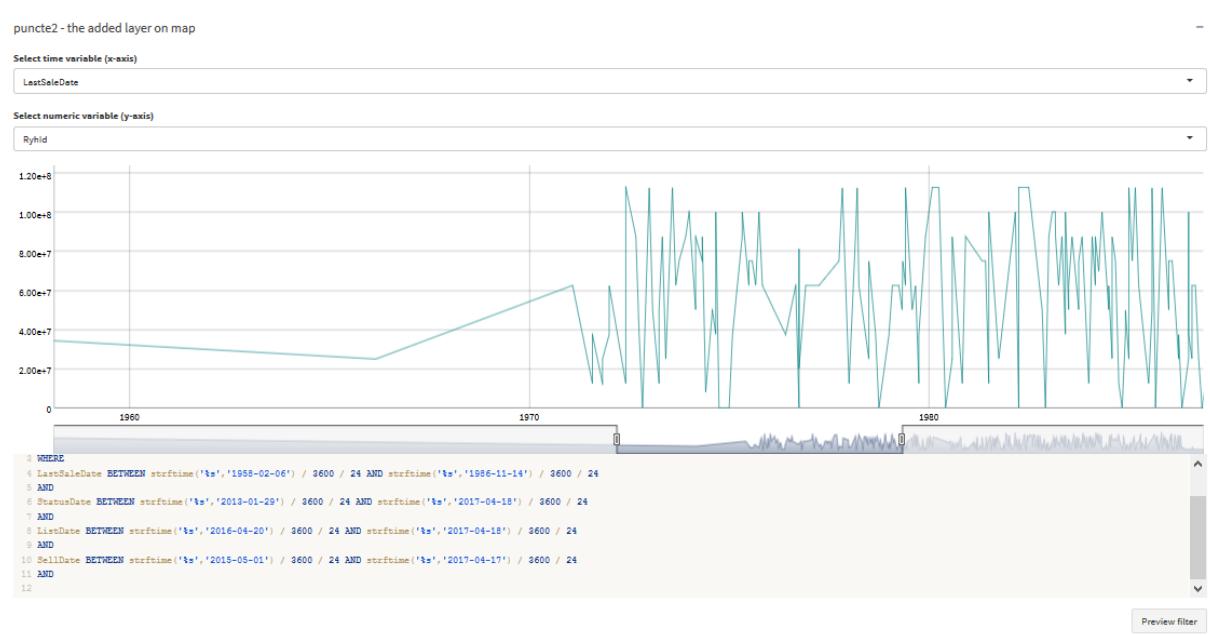


Figura 18: Interfața cu utilizatorul când s-a adăugat un strat pe hartă și s-a accesat tabul *Postfilter* (datele respective aveau măcar un atribut temporal; în caz contrar, nu apărea niciun grafic)

putea fi modificat) sau asupra finalului interogării (ea va putea fi completată cu alte filtre, de obicei, asupra altor atribute: coloanele pentru colorare și cele de tip text selectate).

Rezultatul interogării generate și modificate de utilizator poate fi previzualizat prin apăsarea butonului *Preview filter*. Apoi, prin apăsarea butonului *Filter data on map*, setul de date referit în interogarea SQL este actualizat pe hartă, conform rezultatului interogării.

#### iv. County Aggregates

Tabul *County Aggregates* conține două subtaburi: *Playground area* și *Map counties from data*.

În *Playground area* (vezi Figura 19), se poate lucra fără a avea de dinainte un set de date încărcat în aplicație. Utilizatorul se poate familiariza cât de cât cu aplicația intrând aici (de unde și denumirea). Datele sunt despre regiuni de tip county din SUA. Doar unele funcționalități generale din tabul *Map* au fost implementate aici. Există totuși o funcționalitate în plus: schimbarea culorilor după colorarea în funcție de un SOM.

În *Map counties from data*, avem de-a face cu o hartă ca cea din tabul *Map*. Doar date de tip *Point* pot fi adăugate, însă nu acestea vor fi reprezentate pe hartă. Pentru fiecare instanță se caută regiunea county (din SUA) din care face parte. Datele vor fi agregate după county (funcțiile de agregare pentru fiecare atribut vor fi selectate de utilizator) și deci pentru fiecare county va exista câte o valoare pentru fiecare atribut numeric din datele de tip *Point*. Noile date (despre regiunile de tip counties) vor fi plasate pe hartă, în aceeași manieră ca în tabul *Map*. Deci, deși ca input, se doresc date de tip *Point*, ca output (pe hartă) apar date de tip *Zone*.

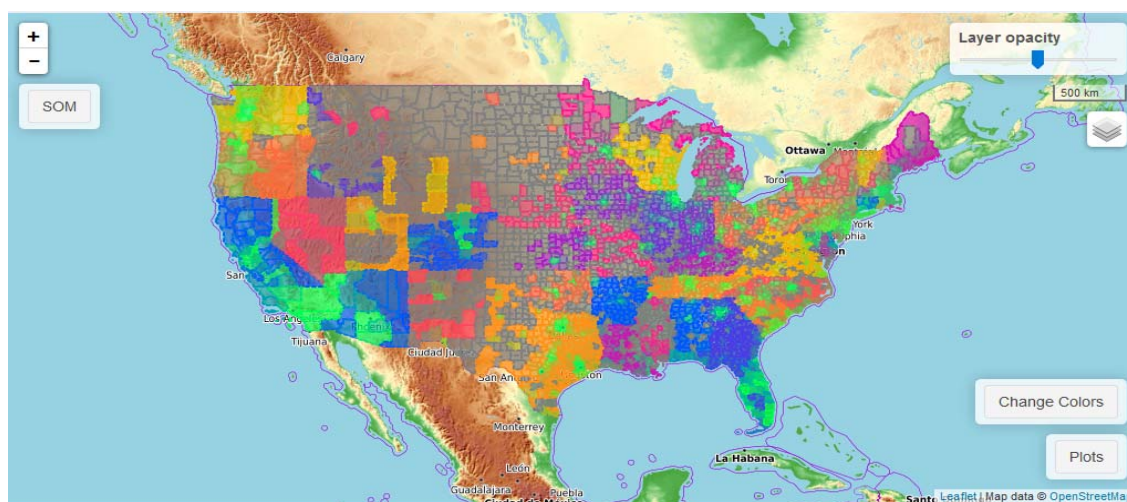


Figura 19: Interfața cu utilizatorul din tabul *Playground area* (s-a colorat după un SOM)



## Concluzii și idei de viitor

Scopul aplicației este de a vizualiza pe hartă date ce prezintă dimensiuni spațiale, în scopul înțelegerii distribuției geografice a acestora în raport cu alte variabile de interes. Aceste date spațiale pot fi de două tipuri: *Point* (latitudine, longitudine) sau *Zone* (format WKT). Datele pot proveni dintr-o baza de date MySQL sau din fișiere CSV/TSV. Unele rânduri/coloane din datele adăugate în aplicație vor fi eliminate automat. Datele pot fi colorate în două moduri: după un singur atribut (după o paletă de culori în degradé 1D: culorile exprimă valoarea unui atribut) sau după mai multe atribute (după o paletă de culori în degradé 2D: culorile exprimă asemănarea dintre instanțe; culori apropiate cromatic vor corespunde instanțelor asemănătoare). Filtrările din cadrul aplicației sunt de două tipuri: prefiltrarea (filtrarea înainte de a plasa date pe hartă; se realizează în tabul *Available data*; dacă se prefiltrează date deja existente pe hartă, acest lucru nu va avea niciun efect asupra hărții) și postfiltrarea (filtrarea după ce datele au fost plasate pe hartă; se realizează în tabul *Postfilter*; orice postfiltrare va modifica harta). Partea de *County Aggregates* poate fi folosită pentru utilizatorul începător (*Playground area*) sau pentru a compara regiuni de tip county din SUA după date de tip *Point* încărcate în aplicație.

În cadrul aplicației se lucrează cu trei tipuri de coloane: *Numeric*, *Character*, *Date*. Rolurile acestora trei sunt: numeric – pentru a colora instanțele; text – pentru a filtra datele și de a afișa informații atunci când se selectează o instanță de pe hartă; dată – pentru a vizualiza graficele de timp din tabul *Postfilter*.

Încărcarea datelor de către utilizator este destul de facilă, iar aplicația reprezintă un instrument folositor pentru lucrul cu date spațiale. Prin plasarea pe hartă a datelor și colorarea lor se pot observa proprietăți ale datelor care pot fi foarte greu identificate din forma tabelară. Indirect, aplicația oferă și suport pentru o analiză minimală a seriilor de timp (prin graficele din tabul *Postfilter*).

Ca idei de viitor, mecanismul de cache al obiectelor SOM ar putea fi remodelat (momentan, pentru date puține în aplicație nu este nimic grav, dar dacă datele devin din ce în ce mai multe, atunci și SOM-urile executate vor fi din ce în ce mai multe, și obiectele SOM reținute vor fi mai numeroase). Apoi, aplicația depinde de o conexiune la o bază de date MySQL. Dacă nu există o astfel de bază de date, aplicația nu poate fi rulată. Se poate ca pe viitor conexiunea la BD să nu fie doar la nivel de aplicație, ci și la nivel de sesiune (fiecare client să se poată conecta la 0/1/mai multe BD proprii sau la BD asociată aplicației). Apoi, momentan, datele de tip *Zone* trebuie să fie în format WKT. Pe viitor, s-ar putea introduce funcționalitatea ca utilizatorul să selecteze formatul datelor de tip *Zone*.

## Bibliografie

### Articole și cărți

1. Kohonen, Teuvo, "Self-Organizing Maps" (1995, 1997, 2001)
2. Vesanto, Juha, et al. "SOM toolbox for Matlab 5." Helsinki University of Technology, Finland (2000).  
(<https://pdfs.semanticscholar.org/9b4f/6595ab9b851d851a440fe480f3b3bf7ad092.pdf>)
3. Vesanto, Juha, et al. "Self-organizing map in Matlab: the SOM Toolbox." Proceedings of the Matlab DSP conference. Vol. 99. 1999.  
([http://cda.psych.uiuc.edu/matlab\\_class/martinez/edatoolbox/Docs/toolbox2paper.pdf](http://cda.psych.uiuc.edu/matlab_class/martinez/edatoolbox/Docs/toolbox2paper.pdf))
4. Matsushita, Haruna, and Yoshifumi Nishio. "Batch-learning self-organizing map with weighted connections avoiding false-neighbor effects." Neural Networks (IJCNN), The 2010 International Joint Conference on. IEEE, 2010. (<http://www.eng.kagawa-u.ac.jp/~haruna/PDF-file/CONF/C041.pdf>)
5. Gabrielsson, Sam, and Stefan Gabrielsson. "The Use of self-organizing maps in recommender systems." Master's Thesis, Computer Science, Uppsala University (2006). (<http://rslab.movsom.com/paper/somrs/somrs.pdf>)
6. Wehrens, Ron, and Lutgarde MC Buydens. "Self-and super-organizing maps in R: the Kohonen package." J Stat Softw 21.5 (2007): 1-19.  
(<https://core.ac.uk/download/pdf/6303136.pdf>)
7. Mitchell, Tom M. "Machine learning. WCB." (1997).

### Resurse Web

#### Resurse R

8. <https://cran.r-project.org/web/packages/RMySQL/RMySQL.pdf>
9. <https://cran.r-project.org/web/packages/sqldf/sqldf.pdf>
10. <https://cran.r-project.org/web/packages/data.table/data.table.pdf>
11. <https://cran.r-project.org/web/packages/rgeos/rgeos.pdf>
12. <https://cran.r-project.org/web/packages/sf/sf.pdf>
13. <https://cran.r-project.org/web/packages/ggmap/ggmap.pdf>
14. <https://cran.r-project.org/web/packages/ggplot2/ggplot2.pdf>
15. <https://cran.r-project.org/web/packages/xts/xts.pdf>
16. <http://adv-r.had.co.nz/Functions.html>

## Resurse *shiny*

17. <http://shiny.rstudio.com/tutorial/video/>
18. <https://shiny.rstudio.com/articles/modules.html>
19. <https://shiny.rstudio.com/articles/reactivity-overview.html>
20. <https://shiny.rstudio.com/articles/execution-scheduling.html>
21. <https://ibiostat.be/seminar/uploads/introdcution-r-shiny-package-20160330.pdf>
22. <https://ryouready.wordpress.com/2013/11/20/sending-data-from-client-to-server-and-back-using-shiny/>
23. <http://deanattali.com/blog/building-shiny-apps-tutorial/#10-reactivity-101>
24. <https://cran.r-project.org/web/packages/shiny/shiny.pdf>
25. <https://cran.r-project.org/web/packages/shinydashboard/shinydashboard.pdf>
26. <https://cran.r-project.org/web/packages/shinytoastr/shinytoastr.pdf>
27. <https://cran.r-project.org/web/packages/shinyBS/shinyBS.pdf>
28. <https://cran.r-project.org/web/packages/shinyjs/shinyjs.pdf>
29. <https://github.com/AnalytixWare/ShinySky>
30. <https://cran.r-project.org/web/packages/shinyWidgets/shinyWidgets.pdf>
31. <https://cran.r-project.org/web/packages/colourpicker/colourpicker.pdf>
32. <https://cran.r-project.org/web/packages/leaflet/leaflet.pdf>
33. <https://cran.r-project.org/web/packages/leaflet.extras/leaflet.extras.pdf>
34. <https://cran.r-project.org/web/packages/DT/DT.pdf>
35. <https://shiny.rstudio.com/articles/pool-basics.html>
36. <https://cran.r-project.org/web/packages/dygraphs/dygraphs.pdf>

## Resurse SOM

37. <http://chem-eng.utoronto.ca/~datamining/Presentations/SOM.pdf>
38. <https://www.r-bloggers.com/self-organising-maps-for-customer-segmentation-using-r/>
39. [http://www.improvedoutcomes.com/docs/WebSiteDocs/SOM/Performing\\_a\\_SOM\\_Experiment.htm](http://www.improvedoutcomes.com/docs/WebSiteDocs/SOM/Performing_a_SOM_Experiment.htm)
40. <https://cran.r-project.org/web/packages/kohonen/kohonen.pdf>

## Resurse Javascript

41. <http://leafletjs.com/reference-1.1.0.html>
42. <http://codemirror.net/> - ca bibliotecă JS preluată

- 43. <http://www.elevateweb.co.uk/image-zoom> - ca bibliotecă JS preluată
- 44. <https://jquery.com/>
- 45. <https://jqueryui.com/>

#### **Alte resurse**

- 46. <https://www.nationalgeographic.org/encyclopedia/geographic-information-system-gis/>
- 47. <https://www.pubnub.com/blog/2015-01-05-websockets-vs-rest-api-understanding-the-difference/>
- 48. <https://www.mapbox.com/blog/supercluster>
- 49. <https://www.lifewire.com/quantization-defined-3134737>