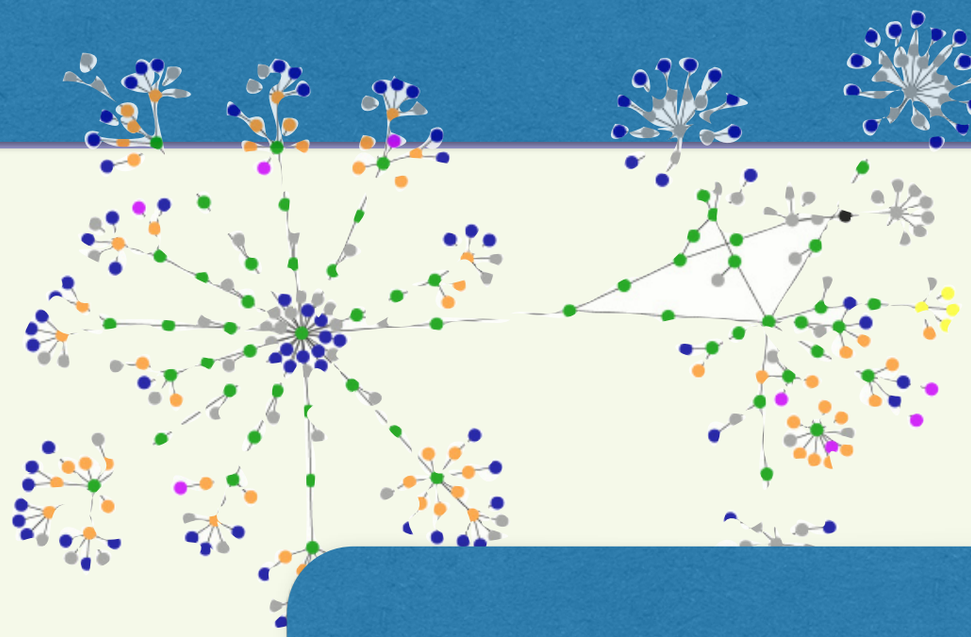
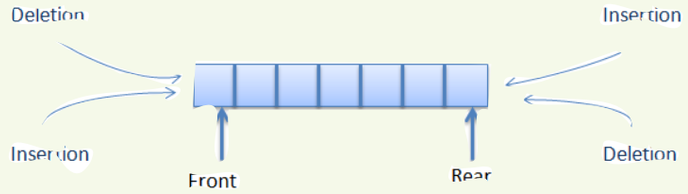
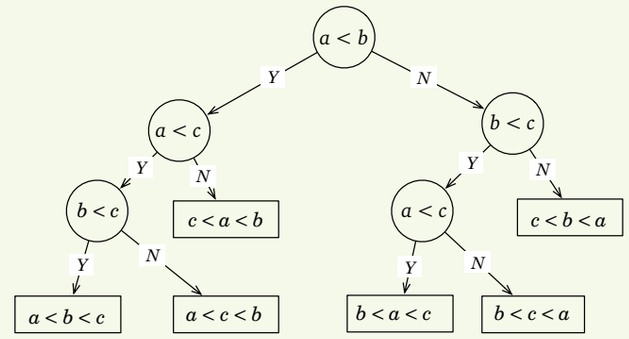




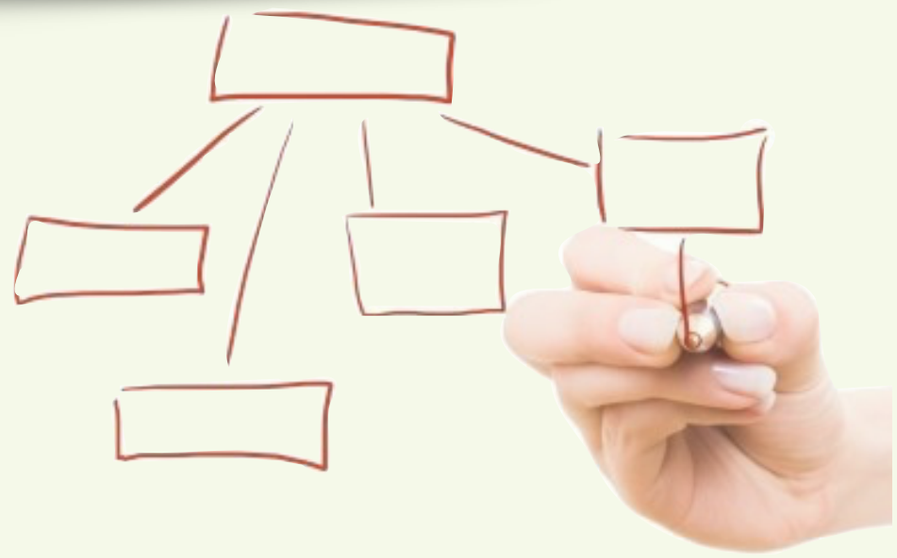


COURSE
“Técnicas Matemáticas para Big Data”
University of Aveiro

Algorithm 2.1: INSERTION-SORT(A)

```
1  for j ← 2 to A.size do
2      key ← A[j]
       // Insert A[j] into the sorted sequence A[1 . . j − 1]
3      i ← j − 1
4      while i > 0 and A[i] > key do
5          A[i + 1] ← A[i]
6          i−−
7      A[i + 1] ← key
```

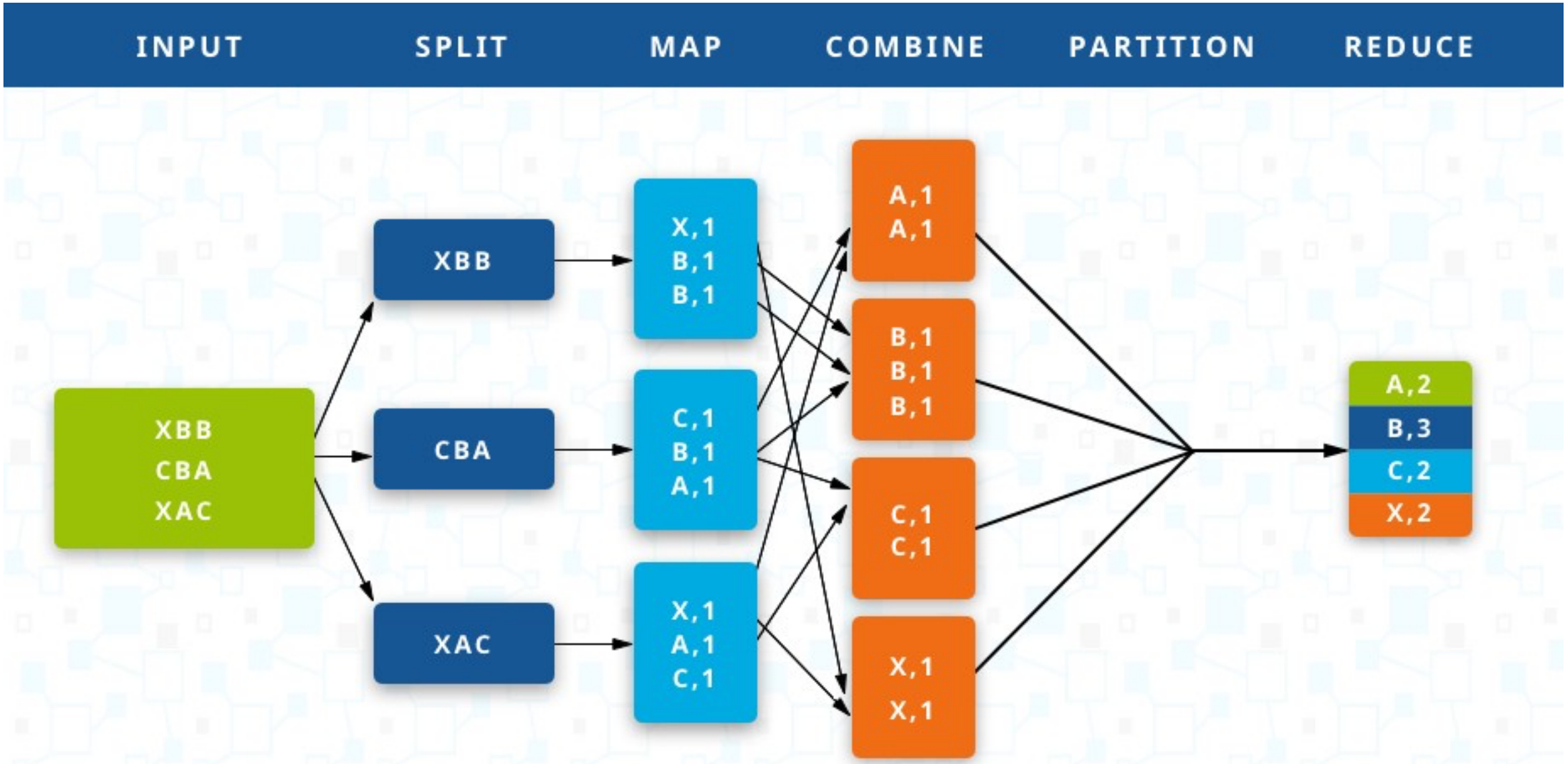
Master in Mathematics and Applications
2020/2021



MapReduce Scheme

A year after Google published a white paper describing the MapReduce framework (2004), Doug Cutting and Mike Cafarella created Apache Hadoop. Hadoop has moved far beyond its beginnings in web indexing and is now used in many industries for a huge variety of tasks that all share the common theme of variety, volume and velocity of structured and unstructured data. Hadoop is increasingly becoming the go-to framework for large-scale, data-intensive deployments.

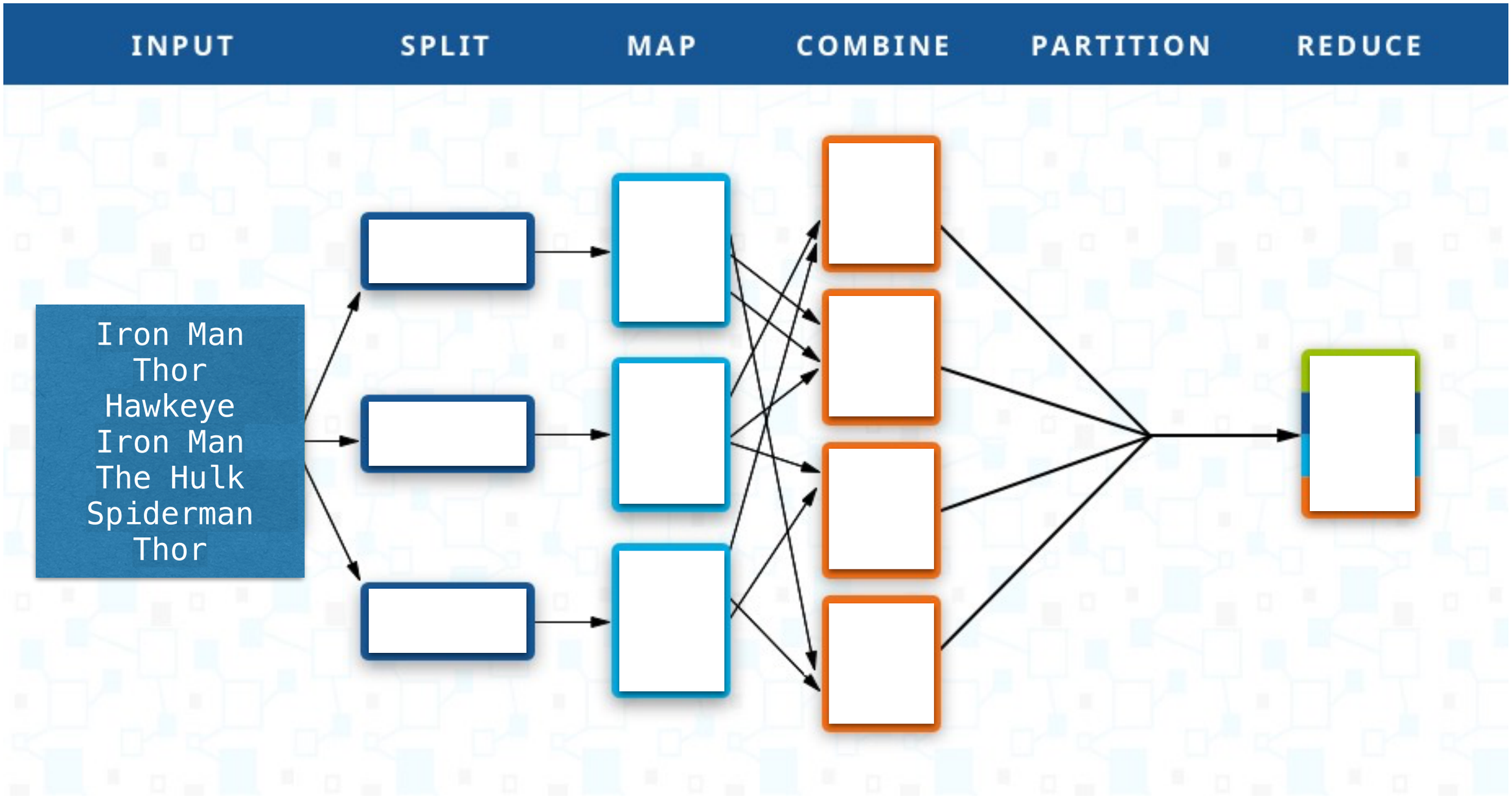
Problem: count letters



(key,value)

MapReduce is a big data processing technique, and a model for how to programmatically implement that technique. Its goal is to sort and filter massive amounts of data into smaller subsets, then distribute those subsets to computing nodes, which process the filtered data in parallel.

Problem: find duplicates



Implementations

The following software and data systems implement MapReduce:

- **Hadoop** — Developed by the Apache Software Foundation. Written in Java, with a language-agnostic API.
- **Spark** — Developed by AMPLab at UC Berkeley, with APIs for Python, Java, and Scala.
- **Disco** — A MapReduce implementation originally developed by Nokia, written in Python and Erlang.
- **MapReduce-MCI** — Developed at Sandia National Laboratories, with bindings for C, C++, and Python.
- **Phoenix** — A threaded MapReduce implementation developed at Stanford University, written in C++.

Limitations

Due to its constant shuffling of data, MapReduce is poorly suited for iterative algorithms, such as those used in ML (machine learning).

Alternatives

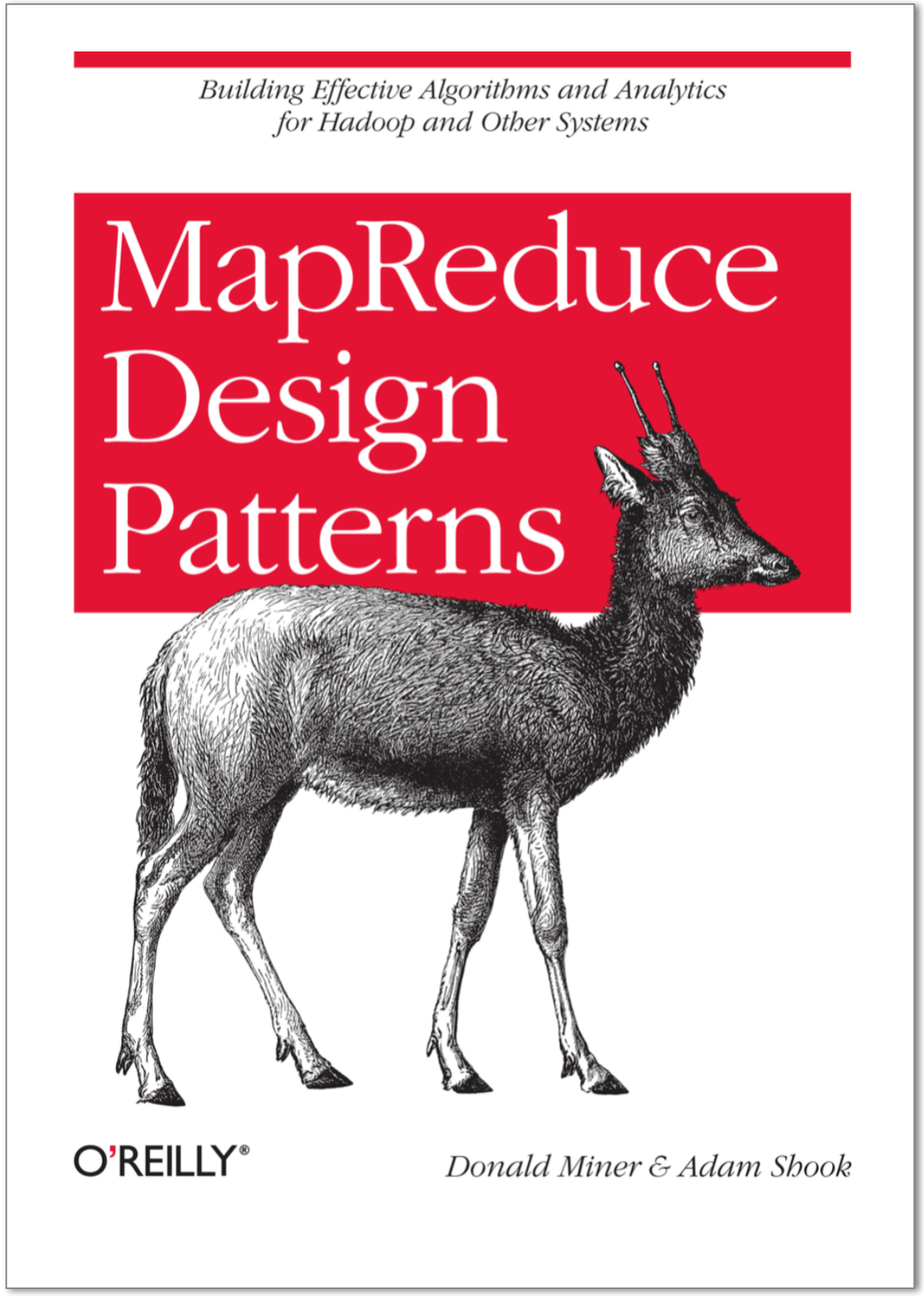
Alternatives to traditional MapReduce, that aim to alleviate these bottlenecks, include:

- **Amazon EMR** — High-level management for running distributed frameworks on the AWS (Amazon Web Services) platform, such as Hadoop, Spark, Presto, and Flink.
- **Flink** — An open-source framework for stream processing of incomplete or in-transit data. Optimized for large datasets, developed by Apache.
- **Google Cloud Dataflow** — Distributed data processing for GCP (Google Cloud Platform).
- **Presto** — An open-source distributed SQL query engine for big data

- **Spark** — RDD and DGG approach (current).

Book Recommendation:

Try some exercises

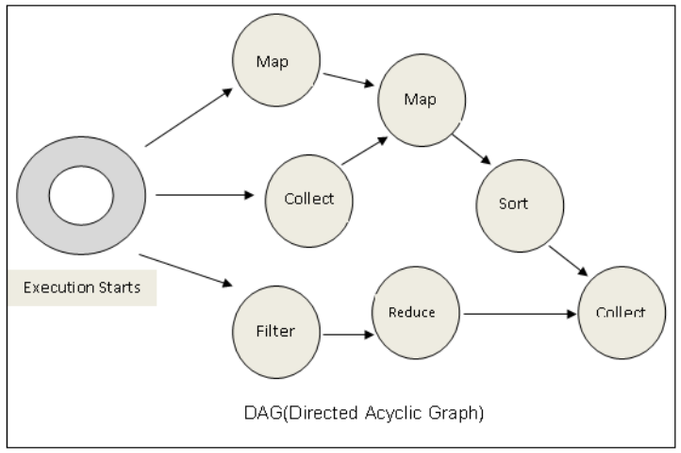


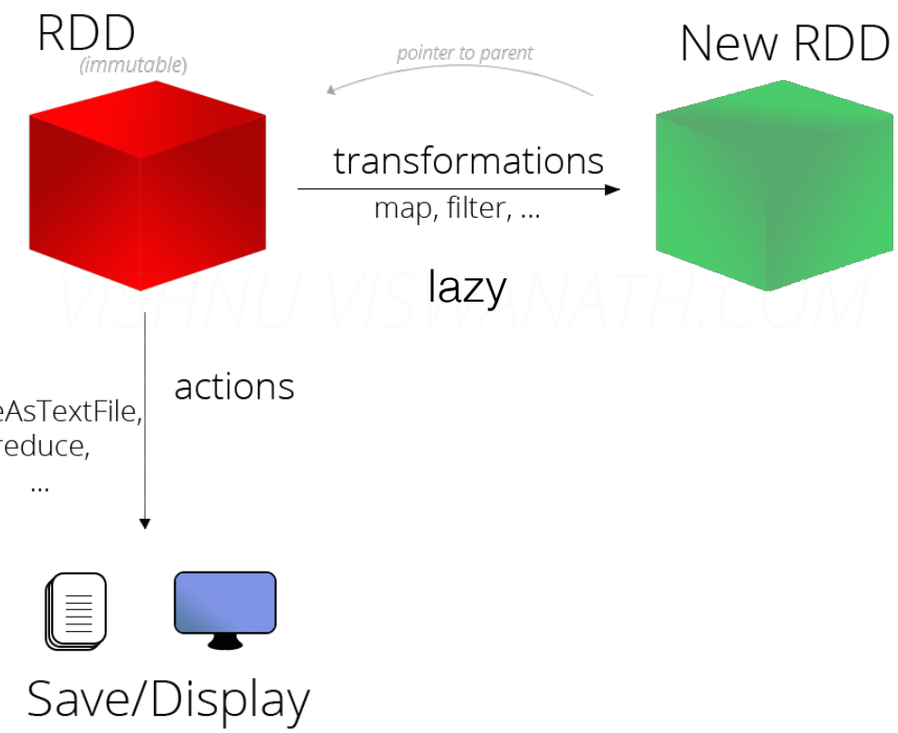
2013 with 251 pg

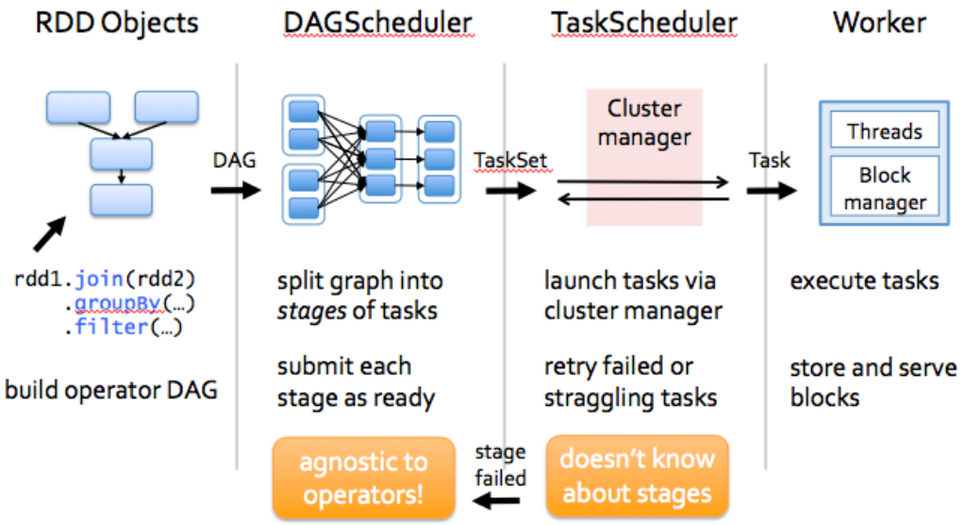
1. Resilient Distributed Datasets (RDD)

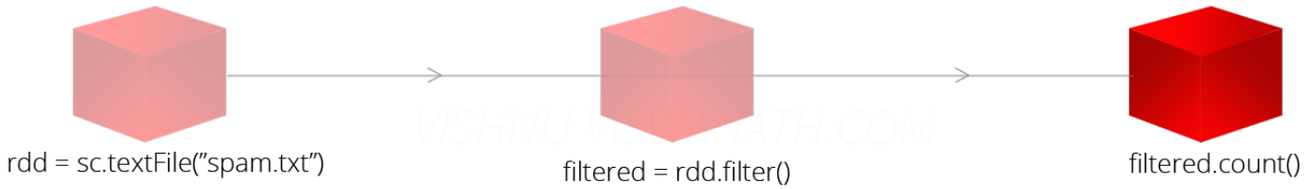
- RDDs are divided into smaller chunks called **Partitions;**
- Executing some **action**, a task is launched per partition;
- More partitions then more **parallelism** (Spark automatically decides the number of partitions);
- Partitions of an RDD are **distributed** across all the nodes in the network.

2. Directed Acyclic Graph (DAG)



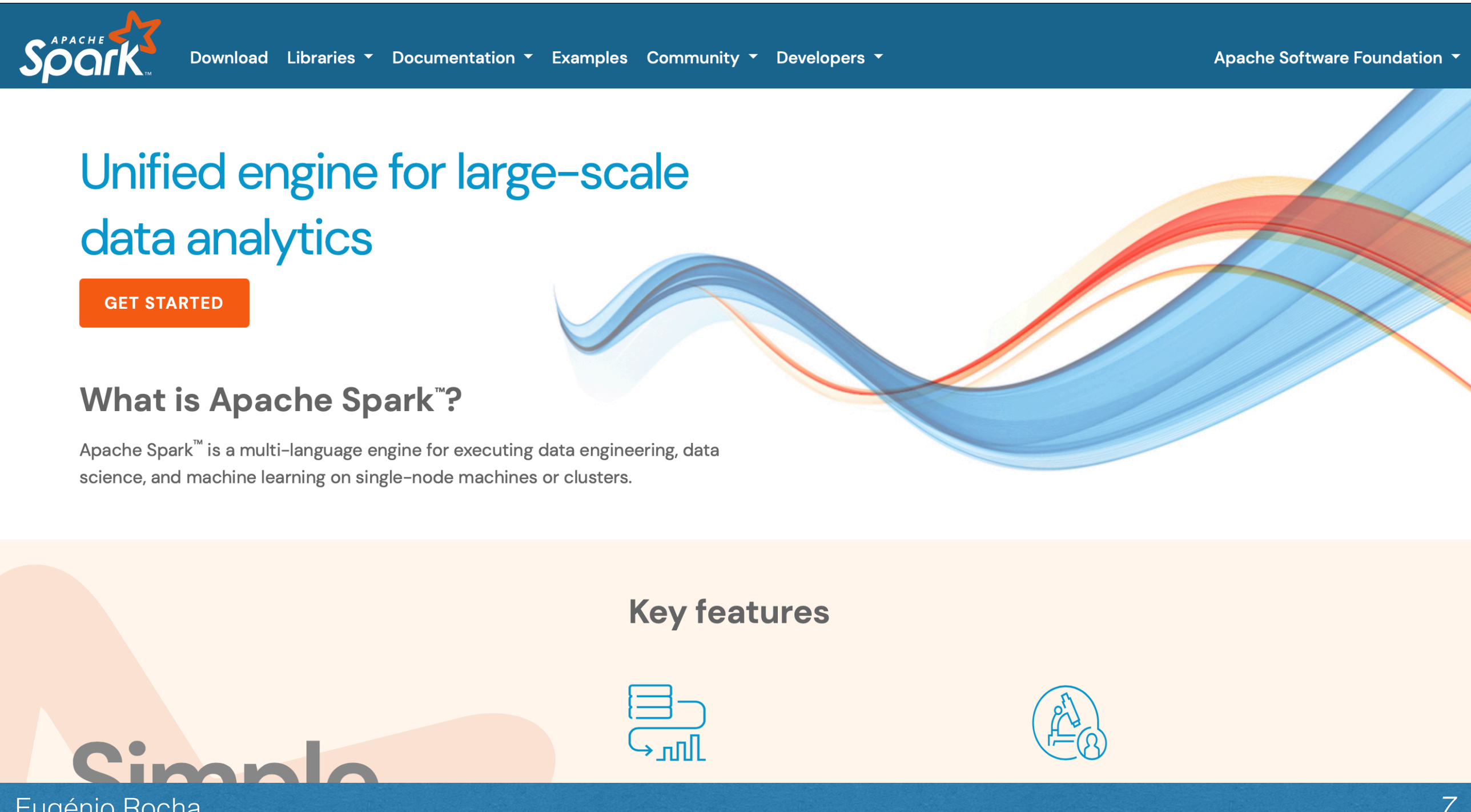







```
val rdd = sc.textFile("spam.txt")
val filtered = rdd.filter(line => line.contains("money"))
filtered.count()
```

https://spark.apache.org



RECALL

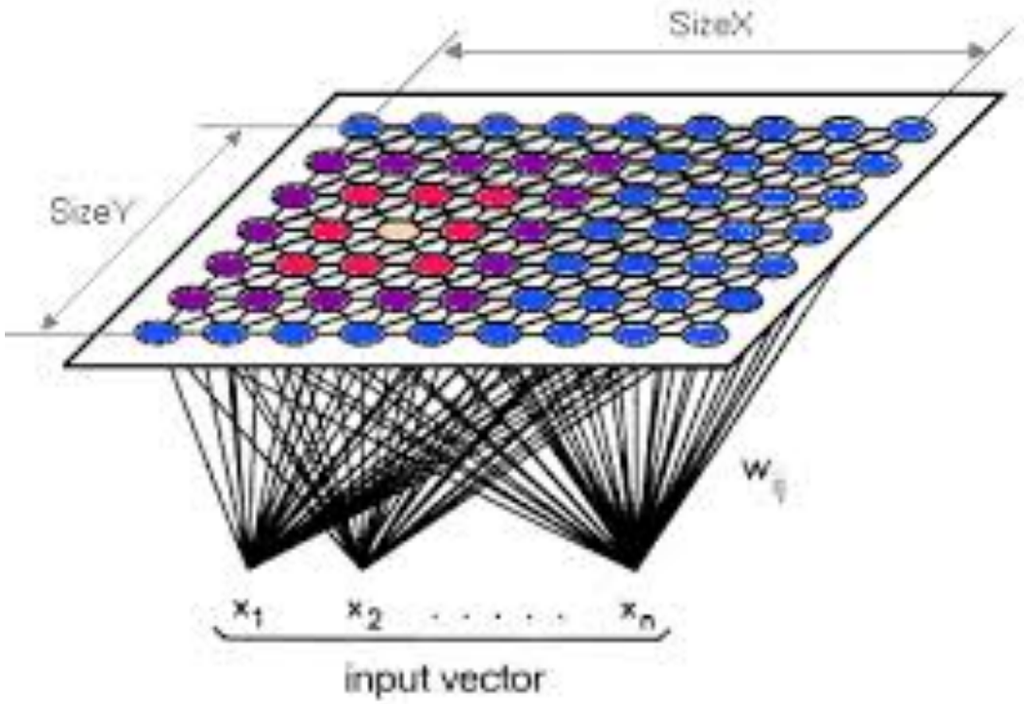
Unsupervised learning problems can be maily grouped into:

- **Clustering**: Discover the inherent groupings in the data.
- **Association**: Discover rules that describe large portions of your data, such as people that buy X also tend to buy Y.

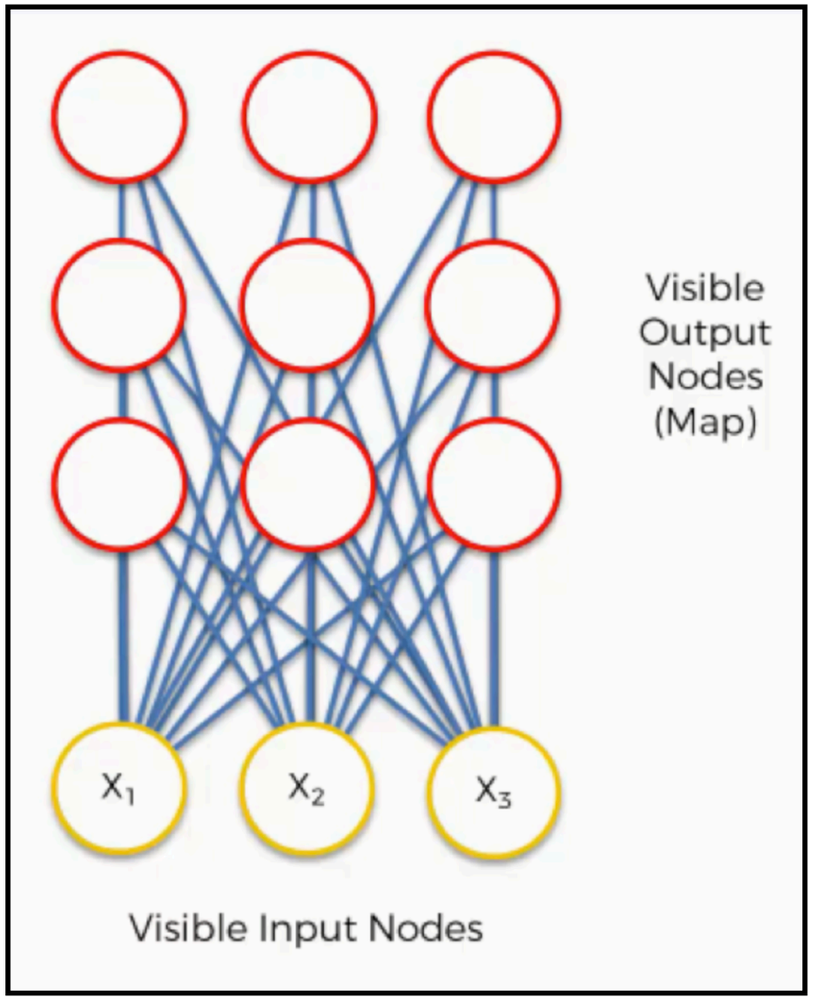
Self Organising Maps (Kohonen's maps)

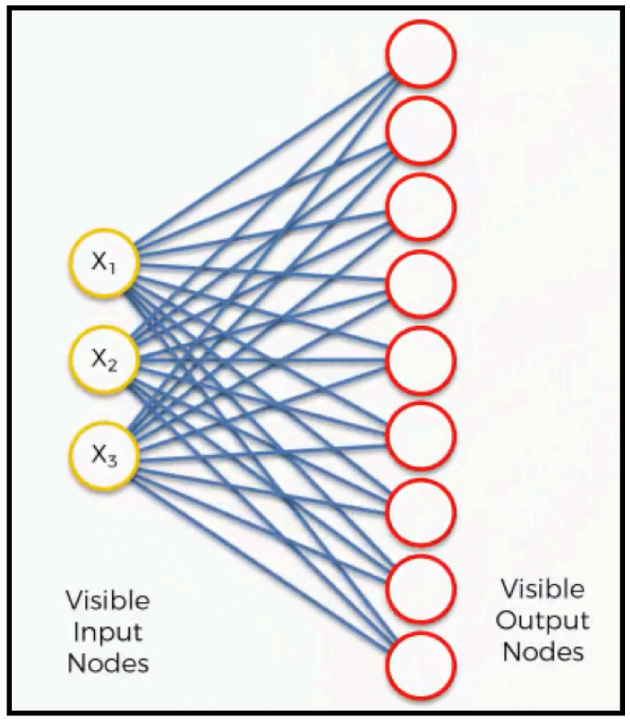
Introduced by the Finnish professor Teuvo Kohonen in the 1980s.

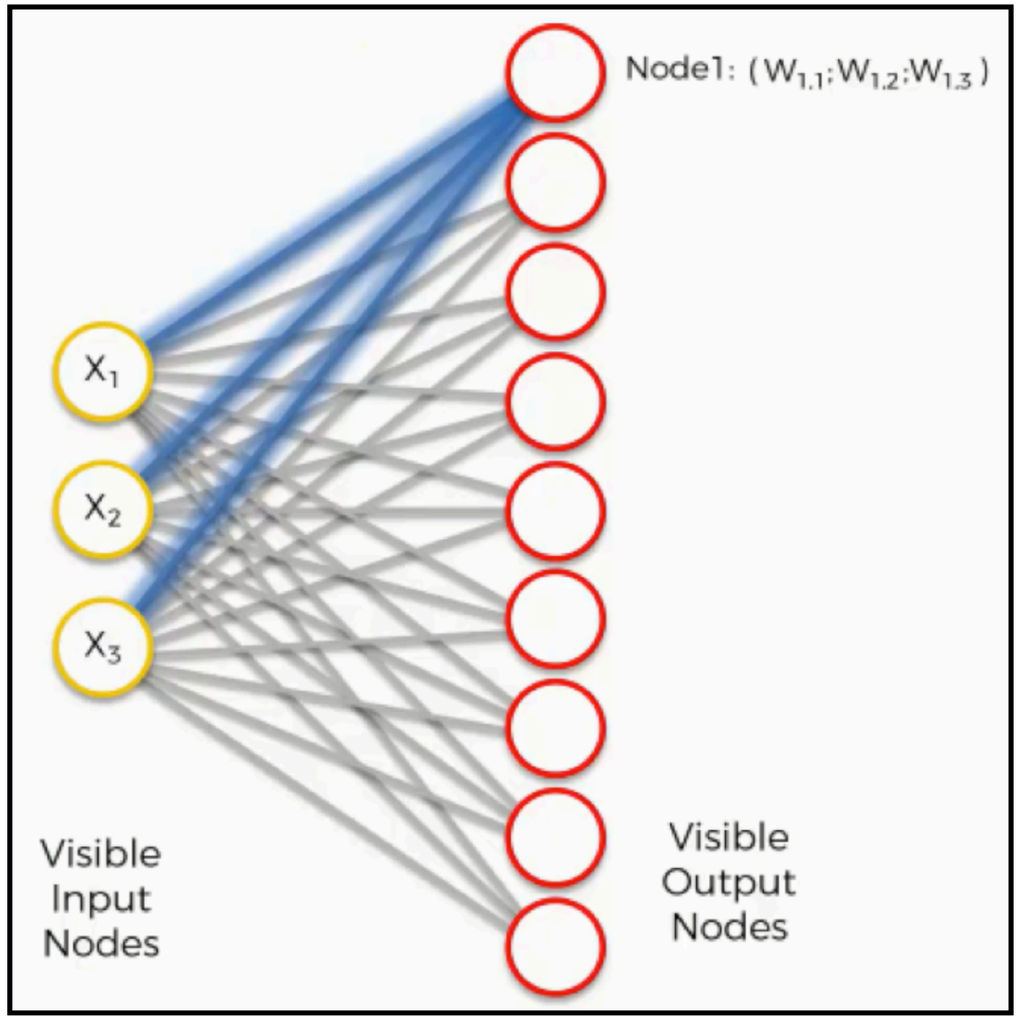
According to Wikipedia, “A **self-organising map** (**SOM**) or **self-organising feature map** (**SOFM**) is a type of artificial neural network (ANN) that is trained using unsupervised learning to produce a low-dimensional (typically two-dimensional), discretised representation of the input space of the training samples, called a **map**, and is therefore a method to do dimensionality reduction.”



SOM sketch

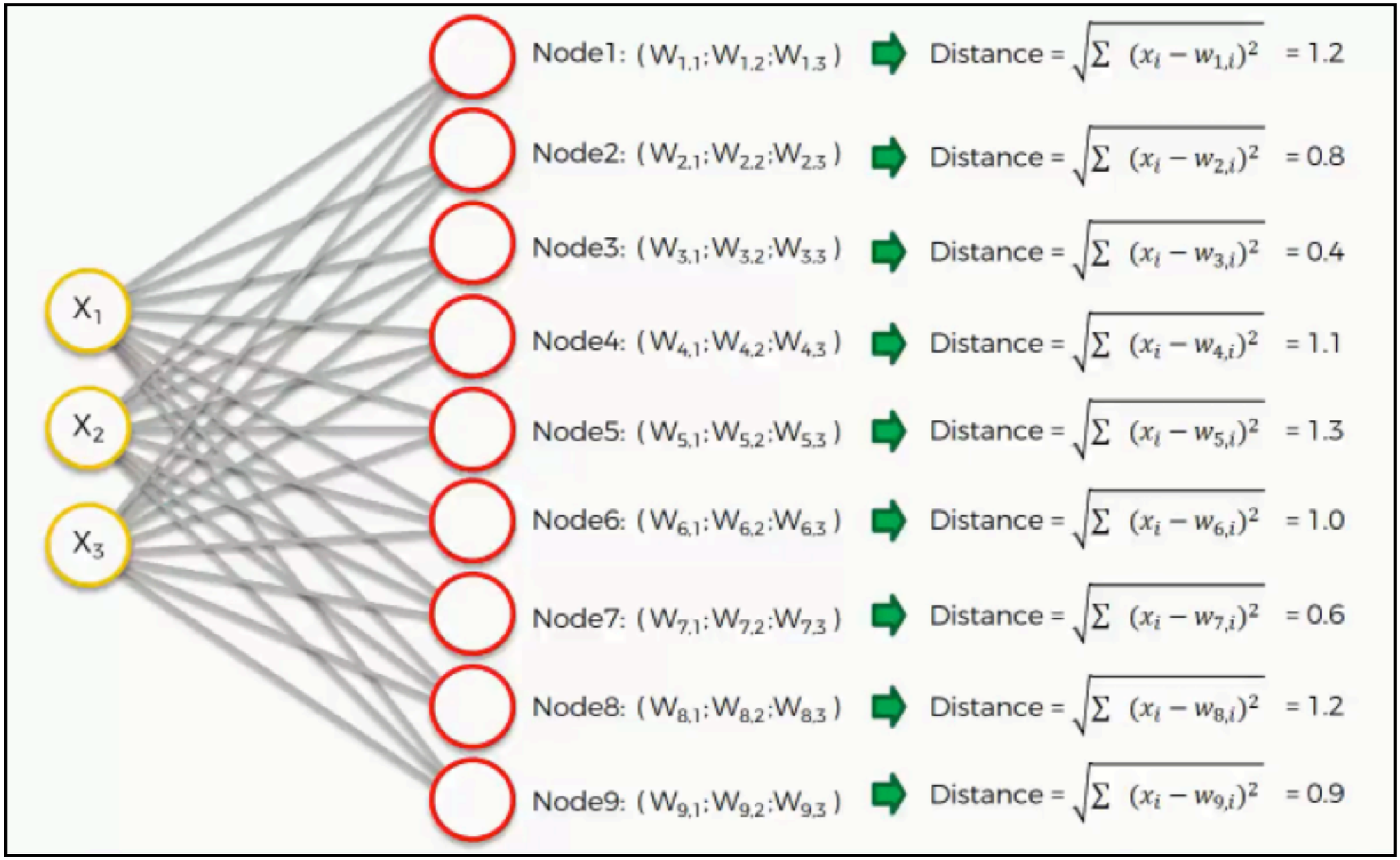


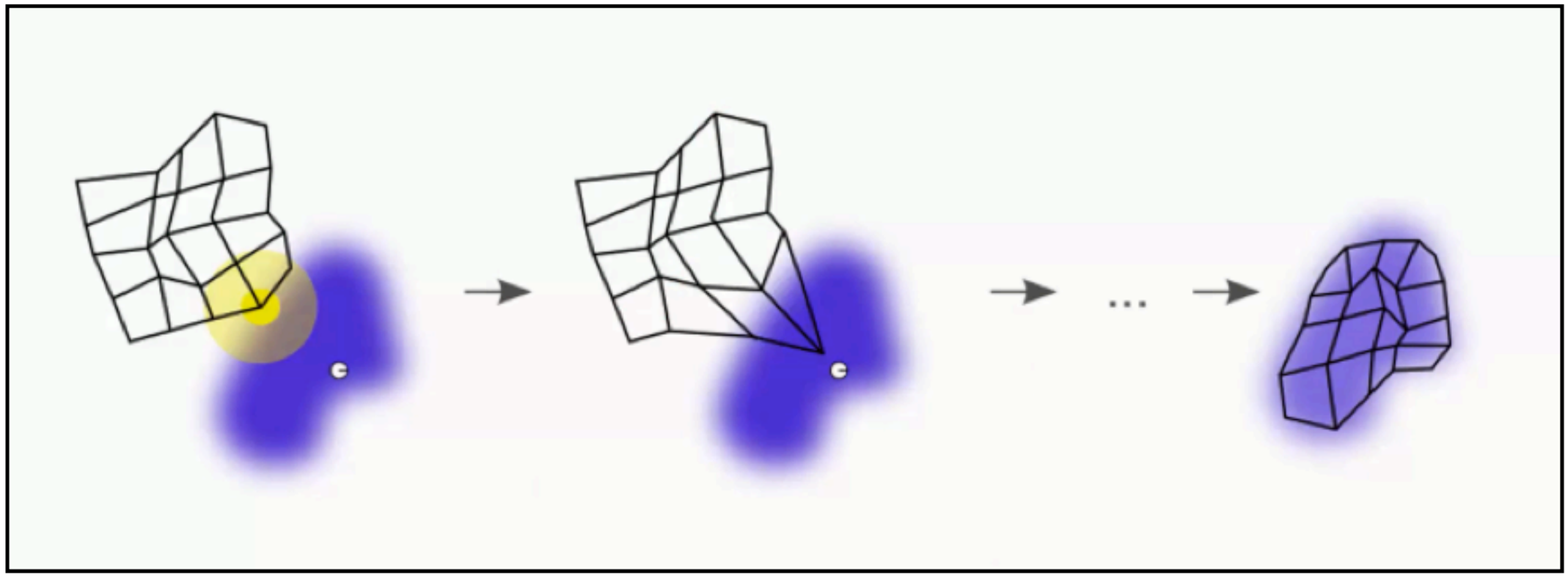


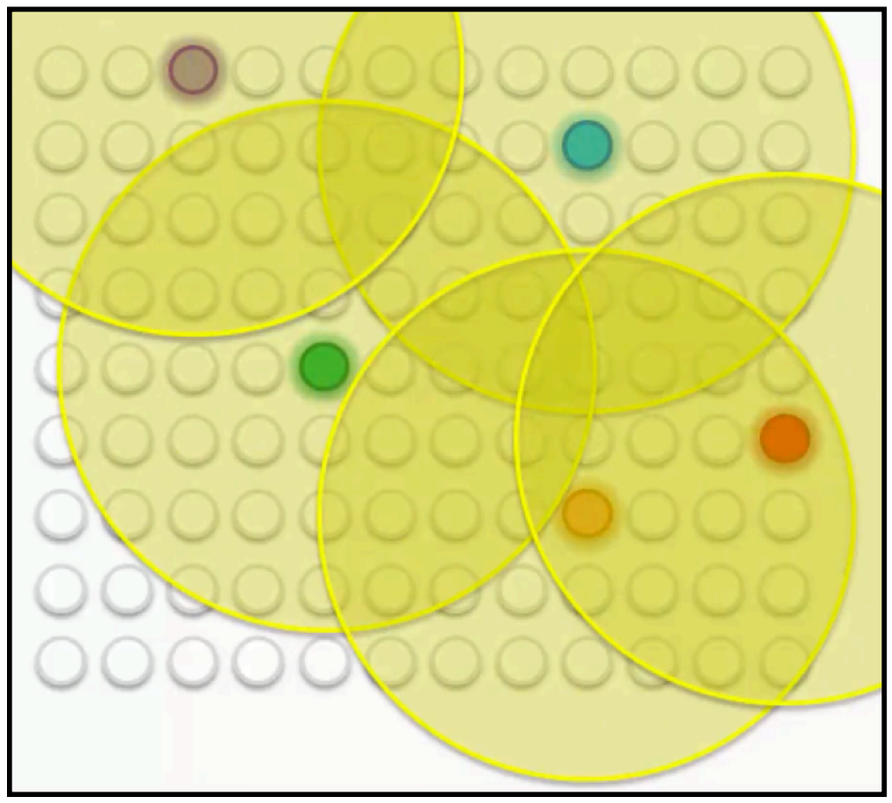


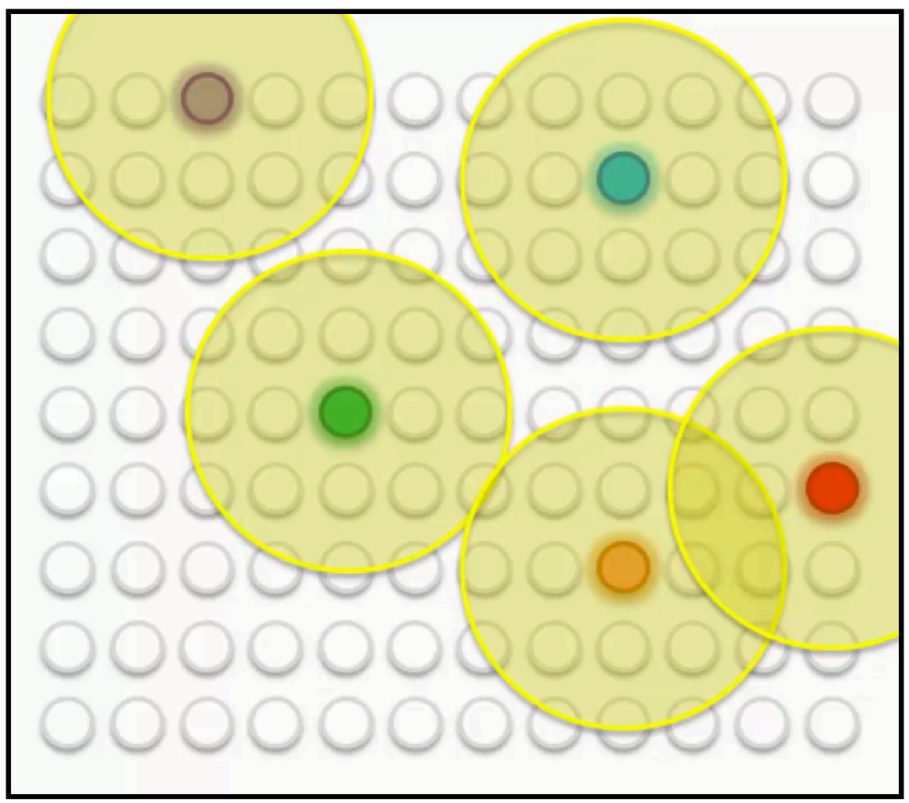
The word "weight" in SOM carries a whole other meaning than it does with artificial and convolutional neural networks. For instance, with ANN, we multiplied the input node's value by the weight and, finally, applied an activation function. With SOMs, on the other hand, **there is no activation function**.

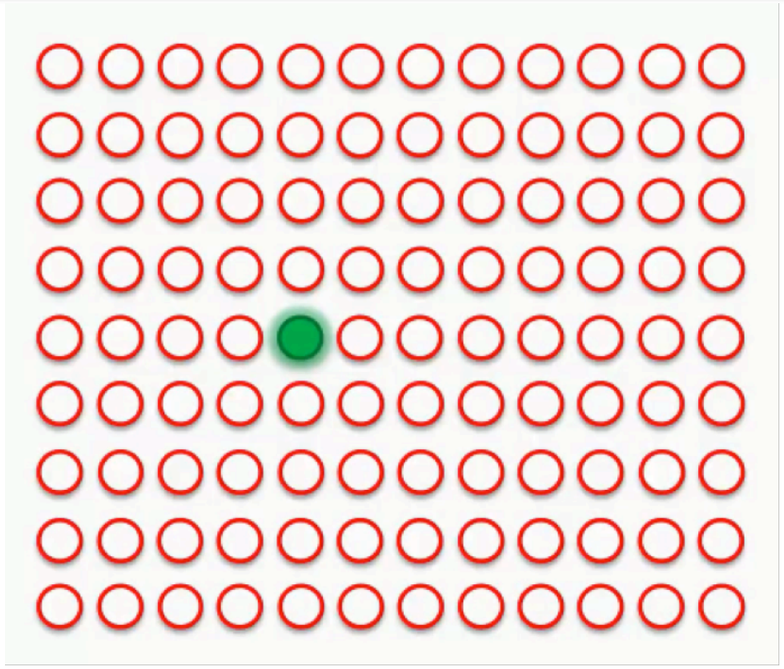
Weights are not separate from the nodes here. In an SOM, the weights belong to the output node itself. Instead of being the result of adding up the weights, the output node in an **SOM contains the weights as its coordinates**. Carrying these weights, it sneakily tries to find its way into the input space.


X1
X2
X3
Node1: (W1,1;W1,2;W1,3) Distance = $\sqrt{\sum (x_i - w_{1,i})^2}$ = 1.2
Node2: (W2,1;W2,2;W2,3) Distance = $\sqrt{\sum (x_i - w_{2,i})^2}$ = 0.8
Node3: (W3,1;W3,2;W3,3) Distance = $\sqrt{\sum (x_i - w_{3,i})^2}$ = 0.4
Node4: (W4,1;W4,2;W4,3) Distance = $\sqrt{\sum (x_i - w_{4,i})^2}$ = 1.1
Node5: (W5,1;W5,2;W5,3) Distance = $\sqrt{\sum (x_i - w_{5,i})^2}$ = 1.3
Node6: (W6,1;W6,2;W6,3) Distance = $\sqrt{\sum (x_i - w_{6,i})^2}$ = 1.0
Node7: (W7,1;W7,2;W7,3) Distance = $\sqrt{\sum (x_i - w_{7,i})^2}$ = 0.6
Node8: (W8,1;W8,2;W8,3) Distance = $\sqrt{\sum (x_i - w_{8,i})^2}$ = 1.2
Node9: (W9,1;W9,2;W9,3) Distance = $\sqrt{\sum (x_i - w_{9,i})^2}$ = 0.9









There are a few points to bear in mind here:

- **SOMs retain the interrelations and structure of the input dataset**.

The SOM goes through this whole process precisely to get as close as possible to the dataset. In order to do that, it has to adopt the dataset's topology.

- **SOMs uncover correlations that wouldn't be otherwise easily identifiable**.

If you have a dataset with hundreds or thousands of columns, it would be virtually impossible to draw up the correlations between all of that data. SOMs handle this by analyzing the entire dataset and actually mapping it out for you to easily read.

- **SOMs categorize data without the need for supervision**.

As we mentioned in our introduction to the SOM section, self-organizing maps are an unsupervised form of deep learning. You do not need to provide your map with labels for the categories for it to classify the data.

Features of SOM

Dimensionality Reduction: The n dimensional data from input layer will be represented by two-dimensional output layer. It can be also said that SOM is a form of compress data by dimensionality reduction.

Competitive Learning: Unlike most of other ANNs, SOM uses competitive learning which means only one neuron gets activated for an input vector. The winner neuron is called Best Matching Unit (BMU).

Topological Preservation: In SOM, the clusters of similar features are near each other. In other words, the data points that are nearby in high dimensional space will be somehow near in low dimensional representation, hence preserving the topology. This is made possible by neighbourhood weight adjustments. When an input vector excites the BMU, the weights of BMU as well as neighbouring neurons are adjusted (depending on a radius from the BMU).

Algorithm

The SOM algorithm that we will apply is called on-line SOM algorithm [1] because the weights are updated after each row or line and the update is recursive in nature. Squared Euclidian distance is used as measure of similarity between each input vector x and each weight vector w of neurons by using the following equation

$$d_k(t) = ||x(t) - w_k(t)||^2 \quad (1)$$

where t denotes training instance (i.e. each row) and k denotes each neuron of SOM.

[1] Lawrence, R.D. & Almasi, G.S. & Rushmeier, Holly. (1999). A Scalable Parallel Algorithm for Self-Organizing Maps with Applications to Sparse Data Mining Problems. Data Mining and Knowledge Discovery. 3. 171-195. 10.1023/A:1009817804059.

The winning or **best-matching unit** (denoted by subscript) is determined by

$$d_c(t) = min(d_k(t)) \tag{2}$$

The weight vectors are **updated** using

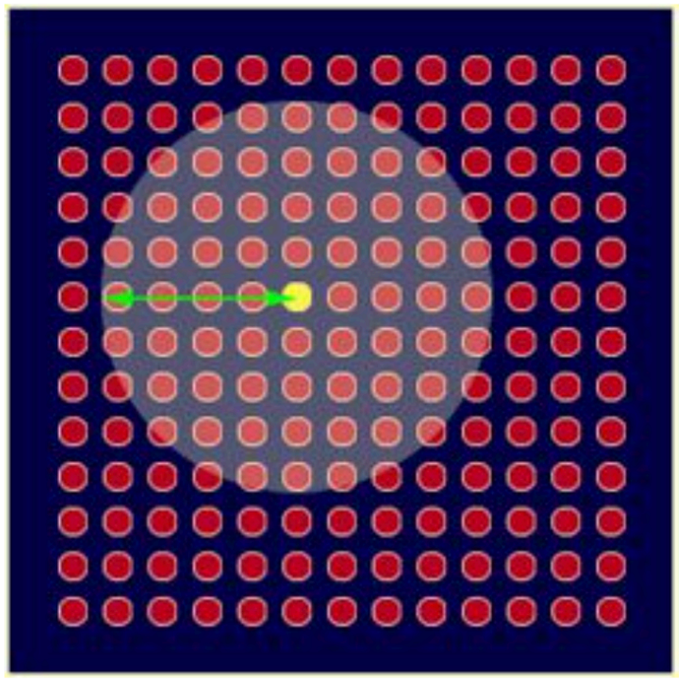
$$w_k(t+1) = w_k(t) + \alpha(t)h_{ck}(t)[x(t) - w_k(t)] \tag{3}$$

where $\alpha(t)$ is the learning-rate factor, and $h_{ck}(t)$ is the neighborhood function.

Normally, standard Gaussian neighbourhood function is used i.e.

$$h_{ck}(t) = \exp(-||r_k - r_c||^2/\sigma(t)^2)$$

where r_k and r_c denote the coordinates of nodes k and c, respectively, on the two-dimensional lattice. The width $\sigma(t)$ of the neighborhood function decreases during training, from an initial value comparable to the dimension of the lattice to a final value effectively equal to the width of a single cell.



The winning or **best-matching unit** (denoted by subscript) is determined by

$$d_c(t) = min(d_k(t)) \tag{2}$$

The weight vectors are **updated** using

$$w_k(t+1) = w_k(t) + \alpha(t)h_{ck}(t)[x(t) - w_k(t)] \tag{3}$$

where $\alpha(t)$ is the learning-rate factor, and $h_{ck}(t)$ is the neighborhood function.

Normally, standard Gaussian neighbourhood function is used i.e.

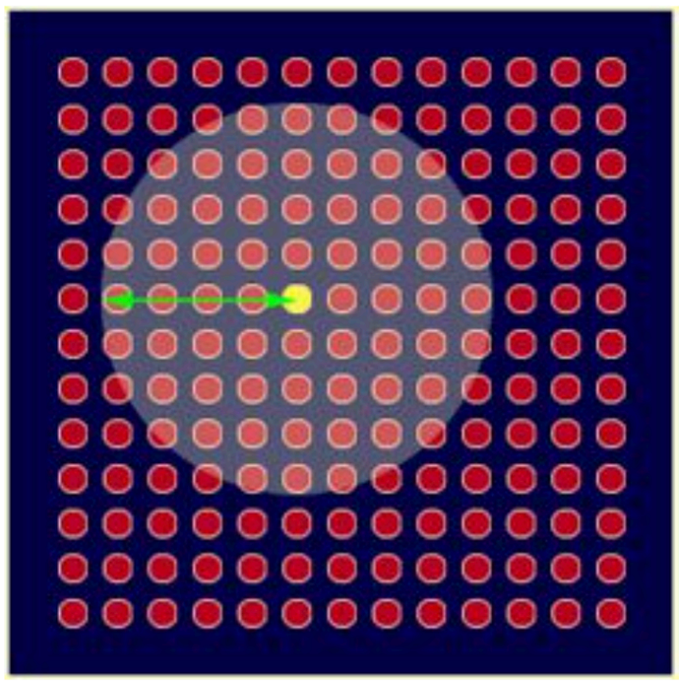
$$h_{ck}(t) = \exp(-||r_k - r_c||^2/\sigma(t)^2)$$

Algorithm

```
begin
    initialize weight vector
    t ⟵ 0
    for epoch ← 1 to N_epochs do
        interpolate new values for α(t) and σ(t)
        for record ← 1 to N_records do
            t ⟵ t + 1
            for k ← 1 to K do
                compute distances d_k using Eq. (1)
            end
            compute winning node c using Eq. (2)
            for k ← 1 to K do
                update weight vectors w_k using Eq. (3)
            end
        end
    end
end
```

spectively, on the two-dimensional lattice. The width $\sigma(t)$ of the

itial value comparable to the dimension of the lattice to a final value



Limitations

- This particular algorithm is serial and on-line in nature which means it is not useful for Big data / distributed computing (large datasets). To handle this problem, there is another algorithm “Batch SOM".

- The learning rate parameter should be decided and it affects the convergence of algorithm. Batch SOM is free from learning rate parameter.

- The initialisation also affects the process of learning. Although the topology is preserved, the positions of clusters may vary on different training. An alternative of random initialisation is Principal Component Initialisation (PCI) [https://arxiv.org/pdf/1210.5873.pdf].

- The order of training samples also affects the clustering process as neighbouring neurons weights are also adjusted and data coming towards the end of the training may influence the final results. To handle this, the training data is shuffled in every epoch. But data shuffling will decrease the performance of algorithm. Batch SOM gets rid of this problem as well.

Notebook: SOM1.ipynb

Recall “Métodos Numéricos”

Método do ponto fixo: convergência

Em que condições é que, dada uma aproximação inicial x_0, o método iterativo $x_k = g(x_{k-1}),\ \ k = 1, 2, \ldots$, converge para a raiz α de $f(x) = 0$ no intervalo $I = [a, b]$?

Teorema: Seja $g \in C^1(I)$ e α o único zero de f em I.

Se

$$g(I) \subset I \ \ \text{(i. e., } \forall x \in I,\ g(x) \in I)$$

e

$$0 < M = \max_{x \in I} |g'(x)| < 1,$$

então g possui um único ponto fixo α no intervalo I e a sucessão de aproximações $\{x_k\}_{k \in \mathbb{N}}$ gerada por $x_k = g(x_{k-1})$, $k = 1, 2, \ldots$, converge para α, qualquer que seja a aproximação inicial $x_0 \in I$.

Recall “Métodos Numéricos”

Método de Newton: condições de convergência

Em que condições a sucessão $\{x_k\}_{k\in\mathbb{N}_0}$ gerada pelo método de Newton $x_{k+1} = x_k - \dfrac{f(x_k)}{f'(x_k)},\ k = 0, 1, 2, \ldots,$ converge para α?

Teorema (condições suficientes de convergência):

Seja $f \in C^2([a, b])$, tal que $f(a)f(b) < 0$, se

i) $f'(x) \neq 0,\ \forall x \in [a, b]$,

ii) $f''(x) \neq 0,\ \forall x \in [a, b]$,

iii) $\left|\dfrac{f(a)}{f'(a)}\right| < b - a$ e $\left|\dfrac{f(b)}{f'(b)}\right| < b - a$,

então o método de Newton converge para a única raiz α de $f(x) = 0$ no intervalo $[a, b]$, qualquer que seja a aproximação inicial $x_0 \in [a, b] = I$.

Recall “Métodos Numéricos”

Métodos de passo único: método de Euler

MN 2018/19

Consideremos o PVI (1)-(2). Se $y = y(x) \in C^2([a,b])$ e x_n, $x_{n+1} \in [a,b]$, pelo desenvolvimento de y em série de Taylor, vem

$$y(x_{n+1}) = y(x_n) + h\, y'(x_n) + \frac{h^2}{2} y''(\xi_n),\ x_n < \xi_n < x_{n+1}.$$

Atendendo a (1), a igualdade anterior pode ser escrita na forma

$$y(x_{n+1}) = y(x_n) + h\, f(x_n, y(x_n)) + \frac{h^2}{2} y''(\xi_n),\ x_n < \xi_n < x_{n+1}. \quad (3)$$

Se h for pequeno, o termo $\frac{h^2}{2} y''(\xi_n)$ será desprezável e, portanto, da fórmula anterior deduz-se a chamada fórmula de Euler

$$y_{n+1} = y_n + h\, f(x_n, y_n),\ n = 0, 1, 2, \ldots, \quad (4)$$

sendo y_0 dado pela condição inicial (2).

Online Algorithms

The ingredients necessary to the definition of these algorithms are:

- An initial value of the d-dimensional code vectors $X_0(i), i \in I$, where I is the set of units;
- A sequence $(\omega_t)_{t\geq 1}$ of observed data which are also d-dimensional vectors;
- A symmetric neighborhood function $\sigma(i, j)$ which measures the link between units i and j;
- A gain (or adaptation parameter) $(\varepsilon_t)_{t\geq 1}$, constant or decreasing.

Then the algorithm works in 2 steps:

1. Choose a winner (in the winner-take-all version) or compute an activation function (we mention this case, but we will concentrate only on the previous winner-take-all version)
    - winner : $i_0(t+1) = arg\min_i\ dist(\omega_{t+1}, X_t(i))$, where $dist$ is generally the Euclidean distance, but could be any other one;

- activation function, for instance :

$$\phi(i, X_t) = \frac{\exp(\frac{1}{T})dist(\omega_{t+1}, X_t(i))}{\sum_{j \in I} \exp(\frac{1}{T})dist(\omega_{t+1}, X_t(j))}.$$

In this expression, if we let the positive parameter T going to infinity, we retrieve the winner-take-all case.

2. Modify the code vectors (or weight vectors) according to a reinforcement rule: the closer to the winner, the stronger is the change.

- $X_{t+1}(i) = X_t(i) + \varepsilon_{t+1}\sigma(i_0(t+1), i)(\omega_{t+1}(i) - X_t(i))$ in the winner-take-all case or
- $X_{t+1}(i) = X_t(i) + \varepsilon_{t+1} \sum_j \sigma(j, i)\phi(j, X_t)(\omega_{t+1}(i) - X_t(i))$ in the activation function case.

The case of the SCL algorithm (i.e. only quantization) is obtained by taking $\sigma(i, i) = 1$ and $\sigma(i, j) = 0, i \neq j$. It can be viewed as a 0-neighbor Kohonen algorithm.

Note: for the definition Simple Competitive Learning Algorithms (SCL) see J. Hertz et al., Introduction to the theory of Neuron Computation, 1991. SOM is considered a powerful extension of SCL.

SOM further math details

From now and for simplicity, we choose the Euclidean distance to compute the winner unit. For any set of code vectors $x = (x(i)), i \in I$, we put

$$C_i(x) = \{\omega / ||x(i) - \omega|| = \min_j ||x(j) - \omega||\},$$

that is the set of data for which the unit i is the winner. The set of the $(C_i(x)), i \in I$ is called the Voronoï tessellation defined by x.

When it is convenient, we write $x(i,t)$ instead of $x_t(i)$ and $x(.,t)$ for $x_t = (x_t(i), i \in I)$.

Then the ODE (in the winner-take-all case) reads :

$$\forall i \in I, \frac{dx(i,u)}{du} = -\sum_{j \in I} \sigma(i,j) \int_{C_j(x(.,u))} (x(i,u) - \omega)\mu(d\omega),$$

where μ stands for the probability distribution of the data set and where the second member is the expectation of $\sigma(i_0(t), i)(\omega_{t+1}(i) - X_t(i))$ with respect to the distribution μ.

In practice, distribution μ looks like a discrete one, because we only use a finite number of "examples" or at least a denumerable set. But generally the good way to model the actual set of possible data is to consider a continuous probability distribution (for a frequency, some Fourier coefficients or coordinate in an Hilbertian base, a level of gray, ...).

When μ is discrete (in fact has a finite support), one can see that the ODE locally derives from an energy function (it means that the second member can be seen as the opposite gradient of this energy function, which is then decreasing along the trajectories of the ODE), that we call *extended distortion*, in reference to the classical distortion in the SCL case. See [19] for elements of proof.

In the SCL case, the (classical) distortion is

$$D(x) = \sum_{i \in I} \int_{C_i(x)} \|x(i) - \omega\|^2 \mu(d\omega) \quad (1)$$

$$= \int \min_i \|x(i) - \omega\|^2 \mu(d\omega). \quad (2)$$

In the general case of SOM, the extended distortion is

$$D(x) = \sum_{i \in I} \sum_{j} \sigma(i,j) \int_{C_j(x)} \|x(i) - \omega\|^2 \mu(d\omega).$$

When μ is diffuse (or has a diffuse part), then D is no more an energy function for the ODE, because there are spurious (or parasite) terms due to the neighborhood function (see [9] for proof). Then when μ is diffuse, the only case where D is still an energy function for the ODE is the SCL case.

Nevertheless, it has to be noticed that

- when D is an actual energy function, then it is not everywhere differentiable and we only get local information which is not what is expected from such a function. In particular, the existence of this energy function is not sufficient to rigorously prove the convergence !
- when D is everywhere differentiable, then it is no more an actual energy function even if it gives a very good information about the convergence of the algorithm.

In any case, if they would be convergent, both algorithms (SOM and SCL) would converge toward an equilibrium of the ODE (even if this fact is not mathematically proved). These equilibria are defined by

$$\forall i \in I, \sum_j \sigma(i,j) \int_{C_j(x^*)} (x^*(i) - \omega)\mu(d\omega) = 0.$$

This also reads

$$x^*(i) = \frac{\sum_j \sigma(i,j) \int_{C_j(x^*)} \omega \ \mu(d\omega)}{\sum_j \sigma(i,j)\mu(C_j(x^*))} \tag{3}$$

In the SCL case, it means that all the $x^*(i)$ are the centers of gravity of their Voronoï tiles. In the statistical literature, it is called self-consistent point. More generally, in the SOM case, $x^*(i)$ is the center of gravity (for the weights which are given by the neighborhood function) of all the centers of the tiles.

From this remark, the definitions of the batch algorithms can be derived.

THE BATCH ALGORITHMS

Using (3), we immediately derive the definitions of the batch algorithms. Our aim is to find a solution of (3) through a deterministic iterative procedure. In a first approach, let us define a sequence of code vectors by:

- x^0 is an initial value
- and

$$x^{k+1}(i) = \frac{\sum_j \sigma(i,j) \int_{C_j(x^k)} \omega \ \mu(d\omega)}{\sum_j \sigma(i,j) \mu(C_j(x^k))}.$$

Now if the sequence converges (or at least for any sub-sequence that converges, which always exists if $(x^k)_{k\geq 1}$ is bounded), it converges toward a solution of (3).

This defines exactly the Kohonen Batch algorithm when μ weights only a finite number N of data. It reads :

$$x_N^{k+1}(i) = \frac{\sum_j \sigma(i,j) \sum_{l=1}^N \omega_l \ \mathbf{1}_{C_j(x^k)}(\omega_l)}{\sum_j \sigma(i,j) \sum_{l=1}^N \mathbf{1}_{C_j(x^k)}(\omega_l)}.$$

Recall that $\mathbf{1}_{C_j(x^k)}(\omega_l)$ is 1 if ω_l belongs to $C_j(x^k)$ and 0 elsewhere.

Moreover, if we assume that when N goes to infinity, $\mu_N = \frac{1}{N}\sum_{l=1}^{N}\delta_{\omega_l}$ (δ_ω stands for the Dirac measure at ω) weakly converges toward μ, we then have (under mild conditions)

$$\lim_{N\to\infty}\lim_{k\to\infty} x_N^{k+1}(i) = x^*(i),$$

where x^* is a solution of (3).

The extended distortion is piecewise convex since the changes of convexity take place on the median hyperplanes which are the frontiers of the Voronoï tessellation. And at this stage, we may notice that the Kohonen Batch algorithm is nothing else but a "quasi-Newtonian" algorithm which minimizes the extended distortion D associated with μ_N.

In fact, when there is no data on the borders of the Voronoï tessellation, it is possible to verify that

$$x_N^{k+1} = x_N^k - \left(diag\nabla^2 D(x_N^k)\right)^{-1}\nabla D(x_N^k)$$

where $diagM$ is the diagonal matrix made with the diagonal of M, ∇D is the gradient of D and $\nabla^2 D$ is the Hessian of D, which exists. It is a "quasi-Newtonian" algorithm because it uses only the diagonal part of the Hessian matrix and not the full matrix.

This proves that in every convex set where D is differentiable, (x_N^k) converges toward a x^* minimizing D. Unfortunately, there are many such disjoint sets and in each of them there is a local minimum of D, even if we do not take into account the possible symmetries or geometric transformations letting μ invariant.

Batch SOM

In batch SOM algorithm [1], the weights are updated only at the end of each epoch according to following equation:

$$w_k(t_f) = \frac{\sum_{t=t_0}^{t=t_f} h_{ck}(t)x(t)}{\sum_{t=t_0}^{t=t_f} h_{ck}(t)} \tag{1}$$

where t_0 and t_f denote the start and finish of the present epoch, respectively, and $w_k(t_f)$ are the weight vectors computed at the end of the present epoch.

The best matching unit (BMU) for each data row/input vector is determined using:

$$d_k(t) = ||x(t) - w_k(t_0)||^2 \tag{2}$$

$$d_c(t) = min(d_k(t)) \tag{3}$$

where $w_k(t_0)$ are the weight vectors computed at the end of the previous epoch.
The calculation for neighborhood function is similar to on-line training algorithm.

$$h_{ck}(t) = \exp(-||r_k - r_c||^2/\sigma(t)^2) \tag{4}$$

where r_k and r_c denote the coordinates of nodes k and c, respectively, on the two-dimensional lattice. The width $\sigma(t)$ of the neighborhood function decreases during training, from an initial value comparable to the dimension of the lattice to a final value effectively equal to the width of a single cell.

Algorithm

begin
  initialize weight vectors identically in all tasks
  $t \longleftarrow 0$
  for $epoch \leftarrow 1$ **to** N_{epochs} **do**
    interpolate new value for $\sigma(t)$
    initialize numerator and denominator in Eq. (1) to 0
    for $record \in [recordsprocessedbymytask]$ **do**
      $t \longleftarrow t + 1$
      for $k \leftarrow 1$ **to** K **do**
        compute distances d_k using Eq. (2)
      end
      compute winning node c using Eq. (3)
      for $k \leftarrow 1$ **to** K **do**
        accumulate local sums for numerator and denominator in Eq. (1)
      end
    end
    reduce to combine local sums for numerator and denominator into global sum in all tasks
    for $k \leftarrow 1$ **to** K **do**
      calculate new weights vector w_k using combined sums in Eq.(1)
    end
    update the weight vectors identically in all tasks
  end
end

How can be implemented in Spark ?

Notebook: SOM2.ipynb

Questões para SOM1:

Q1: Em relação ao SOM1, qual o critério numérico que usaria para medir uma melhoria num algoritmo alterado?

Q2: Escreva a versão SOM1A, onde altera a curva de interpolação do learning factor. Consegue obter melhorias?

Q3: Escreva a versão SOM1B, onde altera a curva de interpolação do desvio padrão. Consegue obter melhorias?

Q4: Escreva a versão SOM1C, onde altera a distribuição normal para outra distribuição. Consegue obter melhorias?

Q5*: Determine a condição matemática que garante a convergência da equação (3), na página 13 dos slides A06-TMBD.pdf.

Q6: Conforme explicado na aula, o SOM usa implicitamente o método de integração de Euler para a ODE associada. Determine o erro (global) cometido após N epochs.

Q7*: Como poderia alterar o SOM proposto para usar o método de Runge-Kutta de segunda ordem? Houve melhorias? De que tipo de melhorias estaremos à espera?

Q8*: Determine o erro cometido após N epochs em Q7.

Q9: Como combinaria as respostas às questões anteriores para melhorar o seu algoritmo, nomeadamente, garantindo uma melhor separação da variável colheitas no dataset original?