Análisis de datos masivos

Técnicas avanzadas

PID_00250692

Jordi Casas Roma
Francesc Julbe
Mario Macias Lloret
Jordi Conesa Caralt

Tiempo mínimo de dedicación recomendado: 5 horas

Universitat Oberta de Catalunya

Índice

Introducción

Las herramientas y tecnologías para el análisis de datos masivos (*big data*) se encuentran en un momento de alta ebullición. Continuamente aparecen y desaparecen herramientas para dar soporte a nuevas necesidades, a la vez que las existentes van incorporando nuevas funcionalidades. Por eso, el objetivo principal de este módulo no es enumerar las tecnologías y herramientas disponibles, sino proporcionar las bases teóricas necesarias para entender este complejo escenario y ser capaz de analizar de forma propia las distintas alternativas existentes.

Aun así, sí que veremos con cierto grado de detalle la implementación, el funcionamiento y las características de algunas de las herramientas más importantes y estables dentro de este complejo escenario, como pueden ser las principales herramientas del ecosistema de Apache Hadoop o las librerías de Apache Spark.

En este módulo revisaremos las principales técnicas relacionadas con el análisis de grafos (*graph mining*), la minería de textos (*text mining*) y el procesamiento de datos en *streaming*, así como sus implicaciones cuando trabajamos con grandes volúmenes de datos.

Iniciaremos este módulo didáctico revisando las bases esenciales de teoría de grafos, que nos permitirán entender cómo se representan los datos semiestructurados empleando grafos. A continuación, veremos algunos de los principales algoritmos para el análisis de grafos y sus correspondientes implementaciones en los ecosistemas Hadoop y Spark.

En el segundo apartado introduciremos las bases de la minería de texto. Detallaremos la metodología y los procesos relacionados, y finalizaremos viendo dos de los ejemplos más empleados en la actualidad, como son Apache Solr y Elasticsearch.

Finalizaremos este módulo didáctico presentando el análisis de datos continuos, o datos en *streaming*. En primer lugar, discutiremos el concepto de datos continuos, para entender el escenario y las posibles soluciones que se plantean. A continuación, veremos cuatro implementaciones líderes en distintos tipos de análisis continuo, como son Apache Storm, Spark Streaming, Apache Flink y los sistemas de gestión de *streams* (flujo) de datos.

Objetivos

En los materiales didácticos de este módulo encontraremos las herramientas indispensables para asimilar los objetivos siguientes:

1. Conocer las bases teóricas del análisis de grafos o redes, así como sus principales problemas de paralelización.
2. Entender la problemática del análisis de grafos, así como las soluciones existentes para trabajar con grandes volúmenes de datos.
3. Comprender la problemática de la minería de textos y las soluciones existentes para trabajar con grandes volúmenes de datos.
4. Conocer los principales entornos de trabajo (*frameworks*) existentes para procesado distribuido de grafos, el análisis de textos y el procesamiento de datos en tiempo real *streaming*.

1. Análisis de grafos

En los últimos años la representación de datos en formato de red ha experimentado un importante auge en todos los niveles. Este formato permite representar estructuras y realidades más complejas que los tradicionales datos relacionales, que utilizan el formato de tuplas. En un formato semiestructurado cada entidad puede presentar, igual que los datos relacionales, una serie de atributos en formato numérico, nominal o categórico. Pero, además, el formato de red permite representar de un modo más rico las relaciones que pueden existir entre las distintas entidades que forman el conjunto de datos. Un claro ejemplo de esta situación lo presentan las redes sociales.

La literatura utiliza los términos red y grafo de manera indistinta. Generalmente podemos encontrar referencias a redes o a grafos sin apenas matices, sin diferencias importantes en su significado. En este texto se utilizan los términos red y grafo indistintamente.

El estudio de este tipo de datos requiere de algoritmos específicos, que permitan explotar la capacidad de expresión de este formato de datos. En este apartado veremos los principales algoritmos de análisis de datos en formato de grafos, que son conocidos como minería de grafos (*graph mining*).

1.1. Introducción a la teoría de redes y grafos

En este subapartado introduciremos la definición y notación básica de la teoría de grafos. Los grafos son la forma más natural de representación de las redes reales, y es en este sentido en el que necesitamos introducir los conceptos básicos para poder representar las redes reales.

Un grafo es una pareja de conjuntos $G = (V,E)$, donde $V = \{v_1,v_2...,v_n\}$ es el conjunto de nodos o vértices y $E = \{e_1,e_2,\ldots,e_m\}$ es un conjunto de aristas que unen dos nodos $e_i = \{v_i,v_j\}$ de forma bidireccional, es decir, el nodo v_i está conectado al nodo v_j y viceversa. En este caso, hablamos de **grafos no dirigidos**, **bidireccionales** o **simétricos**.

Cuando las relaciones no son bidireccionales, hablaremos de **grafos dirigidos**, unidireccionales o asimétricos. En este caso, se representa el grafo como pareja de conjuntos $G = (V,A)$, donde $V = \{v_1,v_2...,v_n\}$ es el conjunto de nodos o vértices, igual que en el caso anterior, y $A = \{a_1,a_2,\ldots,a_m\}$ es un conjunto

de arcos que unen dos nodos $a_i = \{v_i, v_j\}$ de forma unidireccional, es decir, el nodo v_i está conectado al nodo v_j.

Se llama **orden** de G a su número de nodos, $|V|$, que por convenio es referenciado por la letra n. Asimismo, el número de aristas, $|E|$, es referenciado por la letra m y se le llama **tamaño** del grafo.

Lectura complementaria

Pérez-Solà, C.; Casas-Roma, J. (2016). *Análisis de datos de redes sociales*. Barcelona: Editorial UOC.

Los **nodos adyacentes** o vecinos, denotados como $\Gamma(v_i)$, se definen como el conjunto de nodos unidos a v_i por medio de una arista. En este caso, el **grado** de un nodo se define como el número de nodos adyacentes, es decir, $|\Gamma(v_i)|$, aunque generalmente el grado del vértice v_i se denota como $deg(v_i)$. La **secuencia de grados** (*degree sequence*) es una secuencia numérica de n posiciones en que cada posición i indica el grado del nodo v_i.

En un grafo dirigido G se define a los sucesores de un nodo v_i, $\Gamma(v_i)$ como el conjunto de nodos a los cuales se puede llegar usando un arco desde v_i. Se define el grado exterior de un nodo como el número de sucesores $|\Gamma(v_i)|$. De forma similar, se puede definir a los antecesores de un nodo v_i, $\Gamma^{-1}(v_i)$ como el conjunto de nodos desde los cuales es posible llegar a v_i usando un arco. Se define el grado interior de un nodo como el número de antecesores, es decir, $|\Gamma^{-1}(v_i)|$.

En el caso de las redes sociales, los datos se suelen representar utilizando los grafos, dado que permiten una representación natural de las relaciones existentes entre un conjunto de usuarios (representados mediante nodos o vértices en el contexto de los grafos) de la red. Existen varios formatos de grafo que permiten representar cada una de las redes existentes en la realidad y que se adaptan a las particularidades de cada una de ellas. Por ejemplo, podemos encontrar redes con relaciones simétricas, como por ejemplo Facebook, donde si el usuario A es amigo del usuario B, necesariamente B es amigo de A. Por otro lado, podemos encontrar ejemplos de redes asimétricas, como por ejemplo Twitter, en que se representa mediante grafos dirigidos o asimétricos, y donde la relación de «seguir» del usuario A hacia un usuario B no tiene que ser recíproca.

Ejemplo: representación mediante grafos

Vamos a suponer que deseamos modelar las relaciones entre los usuarios de Twitter, mostrando la relación «seguir» que se establece entre dos usuarios de la red. Para representar la información que nos interesa podemos utilizar un grafo dirigido o asimétrico, en donde creamos un arco entre los nodos A y B si el usuario A «sigue» al usuario B. La figura 1 muestra un posible escenario donde podemos ver que los usuarios A, B y C «siguen» al usuario D. Por su parte, el usuario D «sigue» a los usuarios A, E y F.

A continuación, veremos cómo modelar las relaciones en una red simétrica o no dirigida, como puede ser, por ejemplo, Facebook. En esta red los usuarios establecen relaciones de «amistad» bidireccionales, es decir, si un usuario A es «amigo» de un usuario B, entonces implícitamente el usuario B también es «amigo» del usuario A. La figura 2 muestra un posible ejemplo en el que vemos las relaciones de amistad entre siete usuarios. Podemos ver que el usuario A es amigo de D, el cual es también amigo de A y de C, E y F.

Figura 1. Grafo dirigido o asimétrico

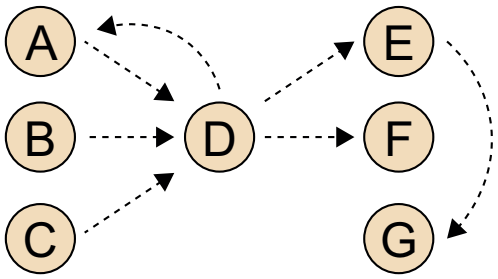
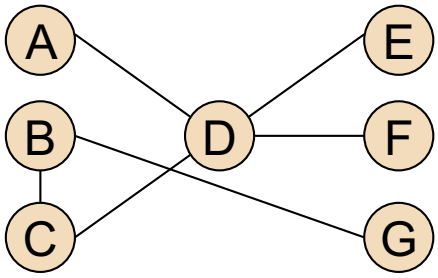


Figura 2. Grafo no dirigido o simétrico



En estos casos, la propia estructura de la red contiene información de gran utilidad para el el análisis y el estudio de las redes.

1.1.1. Matriz de adyacencia

El grafo es una herramienta con una potente representación visual para entender relaciones entre nodos, sin embargo las dependencias entre ellos para procesarlos requiere de una notación matemática, que ayude a su procesado desde un punto de vista algorítmico.

Dado un grafo G, su matriz de adyacencia $A = (a_{i,j})$ es cuadrada de orden n y viene dada por:

$$a_{i,j} = \begin{cases} p & \text{si } \exists(v_i, v_j) \\ 0 & \text{caso contrario} \end{cases} \quad (1)$$

Así, siendo n el número de nodos en un grafo, podemos representarlo con una matriz de n^2 elementos. Si un elemento dado v_i está unido al elemento v_j por una arista de peso p, el elemento $m_{i,j}$ de la matriz toma el valor p. Para cual-

quier par de elementos que no están unidos por una arista, su correspondiente valor en la matriz es igual a 0.

La matriz de adyacencia del grafo G, representada en la figura 1, es:

$$A(G) = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (2)$$

Nótese que la matriz de adyacencia de un grafo no dirigido es simétrica, mientras que no tiene por qué serlo en el caso de un grafo dirigido.

Desde un punto de vista computacional la matriz permite:

- Algoritmo de complejidad $O(1)$ para identificar si dos nodos están conectados.

- Para matrices densas, el consumo de memoria es un problema de $O(n^2)$. Las matrices que representan grafos con pocas conexiones entre nodos son llamadas dispersas (*spare*). Es decir, la mayoría de elementos de la matriz son 0.

1.1.2. Lista de adyacencia

Una alternativa interesante para representar un grafo es mediante la lista de adyacencia. En esta estructura se asocia a cada vértice v_i una lista que contiene todos los vértices adyacentes $v_j|\{v_i,v_j\} \in E$. En el caso de un grafo dirigido, se asocia a cada vértice v_i una lista que contiene todos los vértices accesibles desde él, es decir, $v_j|(v_i,v_j) \in A$. La principal ventaja de este método de representación es el ahorro de espacio para almacenar la información.

Ejemplo

Supongamos que queremos almacenar un grafo no dirigido de 100 vértices y 1000 aristas (n = 100 y m = 1000). Si utilizamos la matriz de adyacencia necesitaremos una matriz cuadrada de $n \times n$, es decir, 10^4 posiciones. En el caso de emplear la matriz de incidencia deberemos reservar $n \times m = 10^5$ posiciones. Finalmente, utilizando la lista de adyancencia requeriremos una posición para cada vértice del grafo y otras dos para cada arista (debido a que representamos la adyacencia entre ambos vértices), es decir, $n + 2m$ = 2.100.

Esta representación cuenta con las siguientes propiedades:

- permite una fácil iteración entre nodos relacionados;
- las listas tienen longitud variable;
- requiere menos memoria que la matriz de adyacencia para ser procesada, siendo proporcional a la suma de nodos y aristas.

Desde un punto de vista programático, una lista de adyacencia es una estructura que puede implementarse utilizando *arrays*, ya que se conoce *a priori* el número de elementos y la longitud de cada *array*.

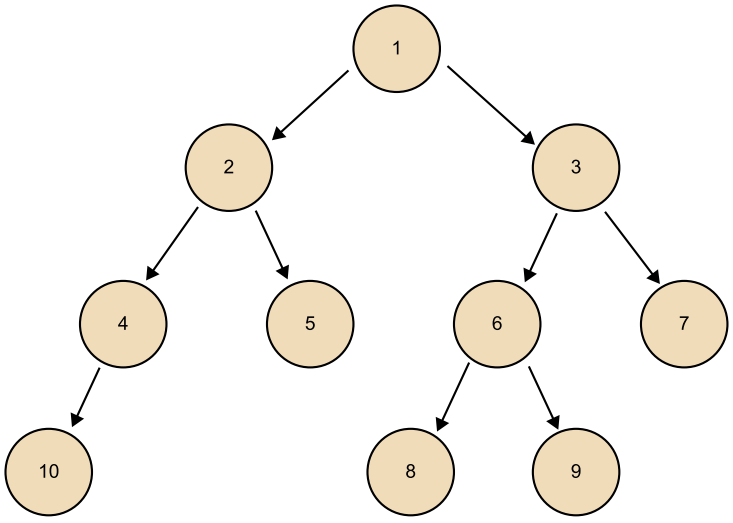
1.1.3. Tipos de grafos

Los grafos permiten representar y resolver muchos problemas habituales encontrados en muchos ámbitos del día a día, más allá de las ciencias de la computación.

Árbol

Un árbol es un grafo que no tiene caminos cerrados en el que existe exactamente un camino entre cada par de puntos. Los árboles son un tipo de grafo que permiten resolver gran cantidad de problemas habituales relacionados con estructuras jerárquicas.

Figura 3. Grafo en forma de árbol



Técnicas de aprendizaje automático basadas en la construcción de estructuras en árbol, como *random forest*, pueden representarse adecuadamente utilizando este tipo de grafos.

Grafo acíclico dirigido

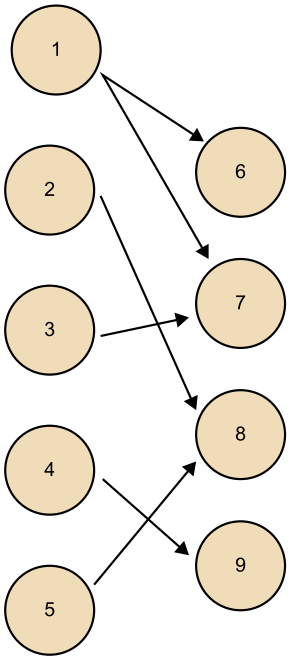
Un grafo acíclico dirigido (*directed acyclic graph*, DAG) es un tipo de grafo que ya se ha estudiado en subapartados anteriores, debido a su aplicación para la ejecución de tareas en Apache Spark.

La característica principal en un DAG es que dado un vértice v_i, no hay un camino de relaciones que vuelvan a v_i en ningún momento. Es decir, no se producen ciclos. Así, en un DAG existe una dirección definida, con un inicio y un final determinados.

Grafo bipartito

Un grafo bipartito es aquel en el que los nodos pueden separarse en dos grupos, nodos de inicio y nodos finales, tal y como muestra la figura 4.

Figura 4. Grafo bipartito



Si el número de elementos en cada subgrupo es idéntico nos referiremos a él como un grafo bipartito balanceado. Si todos los elementos de cada grupo se relacionan con todos los elementos del segundo grupo, diremos que el grafo es completo.

1.2. Procesamiento de grafos

Los grafos son una importante herramienta para la resolución de problemas de procesamiento complejo. Así, al representar un problema en forma de grafo podemos aplicar algunos algoritmos de manejo de grafos.

A continuación, vamos a presentar algunos de ellos que, en muchas ocasiones, constituyen subrutinas incluidas en tareas de procesamiento más complejas. Los algoritmos presentados en este subapartado son los elementos que ayudarán a tareas de procesado más complejas sobre grafos, de forma similar a la manera como las tareas de ordenación de series contribuyen a las tareas de procesamiento que manejan largas colecciones de datos.

1.2.1. Algoritmos para el recorrido de grafos

Para examinar la estructura de un grafo, es común tener que realizar un recorrido por todo el grafo. Las dos estrategias más comunes para tal fin son:

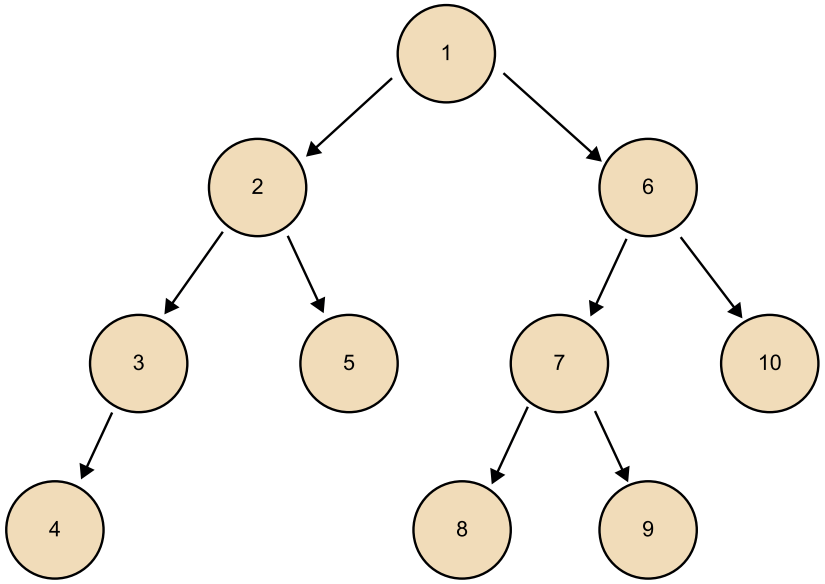
- DFS (*depth first search*) o búsqueda en profundidad prioritaria. Este método prioriza la apertura de una única vía de exploración a partir del nodo actual.

- BFS (*breadth first search*) o búsqueda en anchura prioritaria. Este método prioriza la búsqueda en paralelo de todas las alternativas posibles desde el nodo actual.

Búsqueda en profundidad prioritaria

En la búsqueda en profundidad prioritaria (DFS) se recorre un grafo desde cada nodo hasta el último nodo de una rama. Una vez se llega al final, vuelve hacia atrás hasta encontrar otra rama en el nodo siguiente. Es una búsqueda ordenada, tal y como muestra la figura 5.

Figura 5. Búsqueda en profundidad: la numeración de los nodos corresponde a su orden en el proceso de búsqueda



Desde un punto de vista de programación, es un patrón utilizado muy habitualmente para recorrer arquitecturas jerárquicas, como podría ser un árbol de directorios. Este tipo de algoritmos tienen un marcado carácter recurrente, lo que facilita la implementación utilizando dichos patrones secuenciales de diseño e ingeniería de software. La figura 5 muestra un ejemplo de búsqueda en profundidad prioritaria.

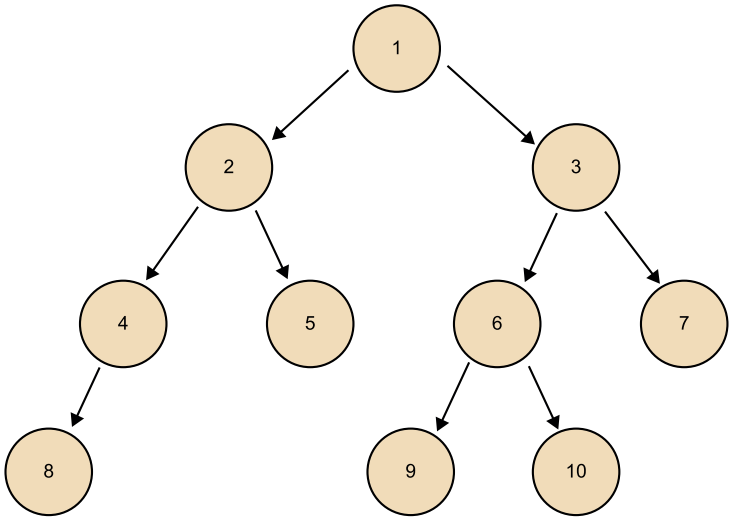
Análisis de paralelismo: DFS tiene una importante componente de ordenación recursiva, recorriendo cada rama del grafo antes de recorrer el siguiente. Esta componente compromete su capacidad de desarrollar algoritmos de procesamiento de grafos en paralelo, ya que el «etiquetado» propio de la ordenación obliga a un sincronismo entre tareas que dificultan su procesado. Para resolver dicha dificultad, existen propuestas de DFS no ordenado.

Es este caso, el principal objetivo es visitar todos los nodos y verificar sus relaciones. Sin embargo, si la ordenación no es un requisito de nuestro análisis, podríamos utilizar estructuras de monitorización de nodos visitados y nodos por visitar, de modo que cada tarea paralela puede recorrer aquellas que quedan por visitar.

Búsqueda en anchura prioritaria

Contrariamente a la búsqueda en profundidad prioritaria, en este algoritmo recorremos solo los nodos adyacentes a cada nodo. Es decir, no recorremos todo el camino hasta los vértices terminales, sino que lo recorremos en anchura y profundizando en cada nueva iteración, tal y como muestra la figura 6.

Figura 6. Búsqueda en anchura: la numeración de los nodos corresponde al orden en que el algoritmo los recorre



Análisis de paralelismo: el número de vértices es finito, de modo que a partir de un nodo inicial vamos a recorrer todos los nodos que se van encontrando

por niveles. Siguiendo con el ejemplo de la figura, recorremos el nodo inicial (primer nivel), a continuación los nodos conectados a este (segundo nivel), y así sucesivamente hasta completar el árbol hasta su máxima profundidad.

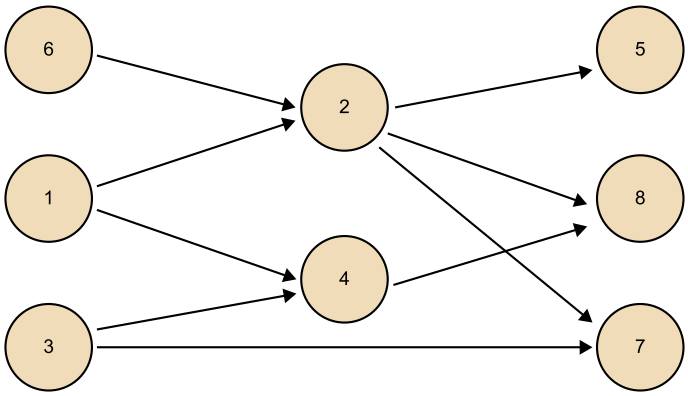
Este es un proceso iterativo que permite más paralelismo, ya que diferentes procesos se pueden ir ejecutando en paralelo y verificando que cada vértice se visita una vez y se le asigna una profundidad mediante una etiqueta. Desde un punto de vista de diseño de programación, podríamos utilizar hilos de procesamiento (*threads*) que analicen las relaciones y vayan etiquetando/marcando aquellos nodos a medida que se recorren, utilizando estructuras de monitorización y almacenamiento intermedio, tales como colas, para etiquetar los nodos visitados y nodos pendientes.

1.2.2. Ordenación topológica

La ordenación topológica de grafos (*topological sort*) es un concepto importante, especialmente en grafos del tipo DAG (directos y sin ciclos), ya que cada nodo viene precedida de otra tarea y es requisito de la siguiente. Mediante la ordenación topológica se identifica la dirección y secuencia del grafo.

Un nodo sin una arista de entrada es un nodo origen, mientras que un nodo sin ninguna arista de salida es un nodo final.

Figura 7. Grafo DAG sobre el que realizar una ordenación topológica



En un grafo puede haber múltiples ordenaciones topológicas válidas. Por ejemplo, en el grafo de la figura 7, son válidas las ordenaciones {6,1,3,2,7,4,5,8} y {1,6,2,3,4,5,7,8}.

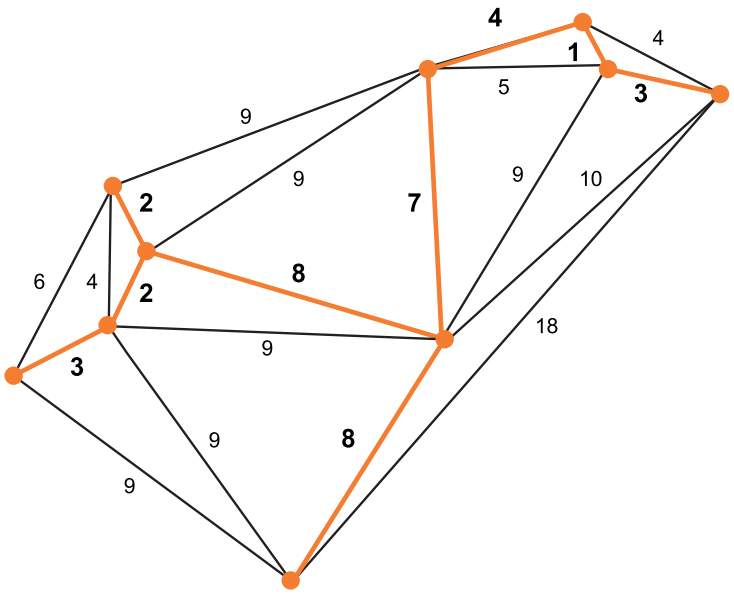
La complejidad de la ordenación topológica puede llegar a ser alta. Una ordenación topológica hace uso de un algoritmo de recorrido de un árbol, sin embargo, al ser la ordenación un factor importante, en este caso se utilizarían técnicas de DFS, siendo un problema de difícil paralelización.

1.2.3. Árbol recubridor mínimo

El árbol recubridor mínimo (*minimum spanning tree*, MST) es aquel camino mínimo que recorre todos los nodos del grafo utilizando el menor número de vértices (o camino en el que la suma de pesos es menor).

Este tipo de operaciones sobre grafos es de especial importancia en escenarios como la minimización de distancias y longitudes en redes de telecomunicaciones o la optimización de potencias y consumos en redes de energía eléctrica.

Figura 8. Ejemplo de MST de un grafo



Existen varios algoritmos para resolver este tipo de problemas:

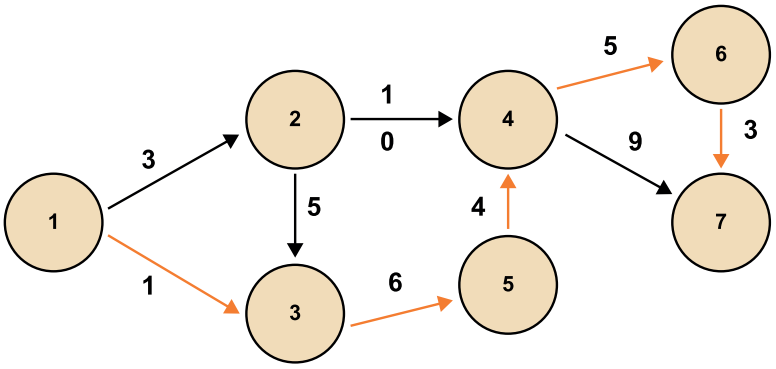
- Algoritmo de Kruskal: recorre cada arista a partir de un nodo, las ordena por peso y selecciona las de menor coste con dos premisas, que son en primer lugar no repetir nodos y, en segundo lugar, no hacer ningún bucle hasta el final.

- Algoritmo de Prim: utiliza la estrategia de buscar las distancias mínimas de cada nodo. Empieza por un vértice inicial arbitrario y va añadiendo vértices al recorrido, tomando aquellos que están a menor distancia. Cuando ya no quedan más nodos por añadir, entiende que ha finalizado.

1.2.4. El camino más corto

El algoritmo del camino más corto (*shortest path*, SP) es aquel que encuentra la distancia mínima entre dos vértices.

Figura 9. Grafo con el camino más corto marcado en azul, en el cual la suma de pesos es mínima para llegar del vértice inicial (1) al vértice final (7)



Existen, una vez más, diferentes algoritmos para resolver este problema, siendo el algoritmo de Dijkstra uno de los más populares.

En este caso, el algoritmo recorre todos los nodos del grafo ordenadamente, añadiendo a una lista aquellos que ya ha visitado y eliminándolos de otra lista en la que se incluyen todos los nodos pendientes de visitar. Además, va añadiendo en una tercera lista solo aquellos que suponen una menor distancia entre dos nodos. A medida que recorre todos los nodos va eliminando elementos de la lista. Una vez ha completado todos los pasos, devuelve la lista con el camino o los caminos más cortos entre los vértices inicial y final.

1.2.5. Importancia de los vértices

El problema de gran relevancia dentro del análisis de grafos es determinar la importancia de los vértices que lo forman. Esto se puede realizar de muchas formas distintas, pero uno de los algoritmos más relevantes es el PageRank, desarrollado por Google para indicar la importancia de una determinada página web dentro de la WWW.

A continuación, describiremos un ejemplo típico de uso de PageRank.

PageRank

PageRank es un algoritmo utilizado por Google para clasificar páginas web en base a las referencias cruzadas entre ellas. La idea que subyace es que las páginas web más importantes tienen más referencias de otras páginas web.

Ejemplo: algoritmo PageRank

En el ejemplo siguiente, consideraremos cuatro páginas que se referencian entre sí (página A, B, C y D). Cada una contribuye a las otras con su peso (valor dentro del círculo o vértice de la figura 10) dividido por el número de referencias a las otras (si una página tiene un peso de 1.0 y dos referencias a otras páginas, contribuye con 0.5 a cada una de ellas).

Las contribuciones se ajustan por el factor de amortiguamiento (*damping*), un ajuste estadístico que tiene un valor aproximadamente de 0.85.

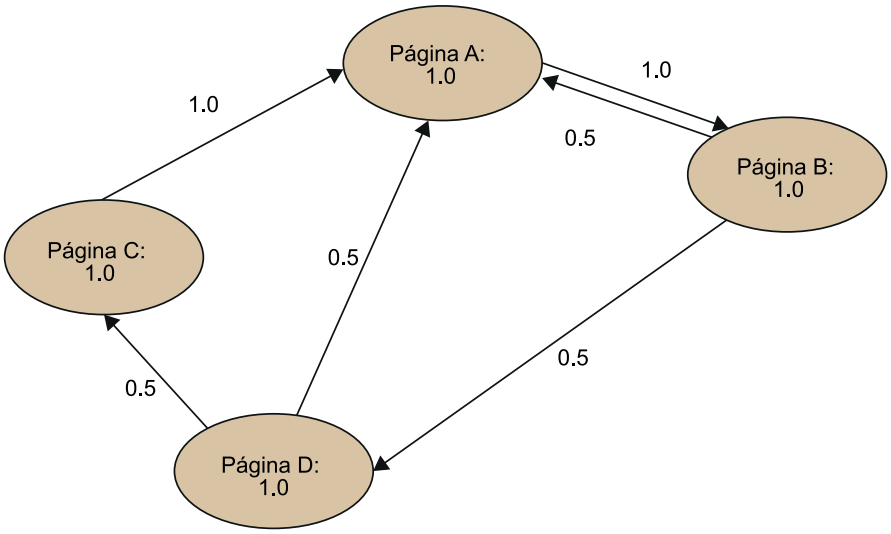
Así, en cada iteración i el peso de cada página se actualiza como su peso P_i, menos el factor de *damping* d, más la suma de los pesos del resto de páginas que la referencian, aplicándole el factor d:

$d = 0.85$
P_i = Peso de la página en la iteración i

Damping

Damping es la probabilidad de que un usuario vaya a dejar de navegar por las páginas referenciadas entre ellas.

Figura 10. Representación gráfica del ejemplo PageRank



Para calcular el peso de la página A en la iteración $i + 1$ utilizamos la ecuación siguiente:

$$P_{i+1}(A) = (P_i(A) - d) + d(P_i(B) + P_i(C) + P_i(D)) \quad (3)$$

En nuestro ejemplo, se aplicaría de la forma siguiente:

$P_1(A) = (1 - 0.85) + 0.85(1.0 + 0.5 + 0{,}5) = 1.85$

Al aplicarse a todo el grafo, tras la primera iteración obtenemos:

$P_1(A) = 1.85$
$P_1(B) = 1.0$
$P_1(C) = 0.58$
$P_1(D) = 0.58$

Tras la segunda:

$P_2(A) = 1.31$
$P_2(B) = 1.7$
$P_2(C) = 0.39$
$P_2(D) = 0.57$

Y finalmente, tras la iteración 10:

$P_{10}(A) = 1.43$
$P_{10}(B) = 1.38$
$P_{10}(C) = 0.46$
$P_{10}(D) = 0.73$

La solución va a converger cuando el incremento en cada iteración sea muy pequeño. El número de iteraciones va a depender de la precisión que deseamos obtener: cuantas más iteraciones, mayor tiempo de computación y mayor precisión.

Actualmente existen dos grandes implementaciones en el entorno de datos masivos para lidiar con grafos. Por un lado, encontramos Apache Giraph, que es parte del ecosistema de Apache Hadoop. Y por otro lado, tenemos GraphX, que es una librería de Apache Spark especializada en la representación y análisis de grafos.

1.3. Hadoop Giraph

Apache Giraph* es un sistema iterativo de procesamiento de grafos creado para entornos altamente escalables. Giraph se originó como la equivalente de código abierto de Pregel, la arquitectura de procesamiento de grafos desarrollada en Google. Giraph utiliza la implementación MapReduce de Apache Hadoop para procesar los grafos. Actualmente es utilizado, entre muchos otros, por Facebook para analizar el grafo social formado por los usuarios y sus conexiones.

* http://giraph.apache.org

Lectura complementaria

Malewicz y otros (2010). *Pregel: A System for Large-Scale Graph Processing*. En: Actas de la conferencia internacional de la gestión de datos 2010 ACM SIGMOD (págs. 135-146). EE. UU. DOI: 10.1145/1807167.1807184

Giraph es un motor de procesado de grafos cuyos cálculos se realizan sobre una máquina *bulk synchronous parallel* (BSP) a gran escala que se ejecuta enteramente sobre Hadoop. El modelo BSP para diseñar algoritmos paralelos no garantiza la comunicación y sincronización entre nodos. Dicho modelo consiste en:

1) componentes con capacidad de proceso y realizado de transacciones en la memoria local (por ejemplo, una CPU),

2) una red que encamina mensajes entre pares de los mencionados componentes de procesado,

3) un entorno hardware que permite la sincronización de todos o de un subconjunto de los componentes de procesado.

Una máquina BSP suele estar implementada como un conjunto de procesadores que ejecutan diferentes hilos de computación, equipados con una memoria local rápida e interconectados por una red de comunicación. La comunicación es un componente esencial en un algoritmo BSP, donde cada proceso que se lleva a cabo en una serie de **superpasos** que se repiten, y están compuestos por tres subfases:

- **Computación concurrente**: en esta fase cada procesador realiza cálculos de manera local y asíncrona accediendo solo a los valores guardados en la memoria local.

- **Comunicación**: fase mediante la cual los procesos intercambian datos entre ellos para facilitar la persistencia remota de datos.

- **Sincronización de barrera**: cuando un proceso alcanza un punto de barrera, espera hasta que los demás procesos hayan alcanzado dicha barrera.

La computación y la comunicación no tienen por qué estar ordenadas temporalmente. La comunicación típicamente es unidireccional, es decir, que el proceso pide y envía datos a una memoria remota (nunca recibe datos si no los pide). La posterior sincronización de barrera asegura que todas las comunicaciones unidireccionales de los procesos han finalizado satisfactoriamente.

Giraph define los sistemas de procesado como un grafo compuesto por vértices unidos por aristas dirigidas. Por ejemplo, los vértices podrían representar a una persona y las aristas las peticiones de amistad en una red social. La computación se lleva a cabo mediante una serie de superpasos anteriormente descritos.

A la hora de implementar vértices, el usuario puede decidir las características, la memoria, y la velocidad de cada uno de estos. Esto se puede conseguir implementando varias de las superclases abstractas de *Vertex* que Giraph provee. Mientras los usuarios expertos pueden implementar sus propios vértices a bajo nivel, el equipo de Giraph recomienda a los usuarios noveles/intermedios implementar subclases de *Vertex* y adaptar los ejemplos que ellos mismos proveen.*

*** http://bit.ly/2lvR2ML**

1.3.1. Tipos de grafos soportados por Giraph

El tipo de grafo por defecto en Giraph son grafos dirigidos, en los que todas las aristas se consideran aristas de salida que salen del vértice en el cual se han definido. A partir de ahí, existe también la posibilidad de definir otros tipos de grafos:

- **Grafos no dirigidos**. El programador debe asegurarse de que por cada arista de salida que sale de un vértice A a un vértice B, hay otra arista que sale de B a A. Giraph no detectará si el grafo simple no está compuesto como tal.

- **Grafos no etiquetados**. Para ello, el programador deberá extender las clases que se encargan de leer la información de los vértices (*VertexInputFormat*/*VertexReader*) para que lea las aristas y no espere ninguna información, o definir los parámetros genéricos de un vértice que definen el tipo de peso como *NullWritable*.

- **Grafos etiquetados**. Se definen de manera análoga a los no etiquetados, pero implementando las clases asociadas para que lean los datos pertinentes de las aristas, o definiendo el tipo de datos genérico de estas.

- **Multigrafos**. El equipo de Giraph ha anunciado que recientemente soportan este tipo de grafos, aunque en el momento de escribir este documento, su uso no está documentado.

1.3.2. Agregadores

Los agregadores permiten realizar computaciones de manera global, sobre todos los nodos. Se pueden usar para comprobar que una condición global se cumple, o para calcular algunas estadísticas (por ejemplo, contar elementos).

Durante un superpaso, los vértices asignan valores a los agregadores, que luego son agregados (valga la redundancia) por el sistema, haciendo que los resultados estén disponibles para todo el sistema en el siguiente superpaso. Por ello, la operación realizada por un agregador debe cumplir la propiedad conmutativa y asociativa (por ejemplo, una suma, un AND o OR booleano, etc).

El usuario puede crear sus propios agregadores, pero Giraph provee algunos agregadores básicos: encontrar el máximo o mínimo entre dos números, sumatorios, etc.

Giraph es un motor de computación de grafos, y no una biblioteca de algoritmos, por lo que los algoritmos para tratar grafos son presentados como «ejemplos» en el paquete *org.apache.giraph.examples*, entre los cuales se encuentran:

- **BrachaTouegDeadlockComputation*** para detectar situaciones de interbloqueo (*deadlock*), en las que un vértice A necesita para continuar un recurso proporcionado por un vértice B, pero B no puede proporcionarlo porque necesita un recurso que solo A le puede proporcional.

* http://bit.ly/2CoWKcI

- **ConnectedComponentsComputation**** detecta componentes que conectan varios vértices y asigna a cada vértice un identificador del componente al que pertenecen.

** http://bit.ly/2CsedOZ

- **MaxComputation**, que detecta el valor máximo de un grafo.

- **PageRankComputation**, que implementa el algoritmo *PageRank**** de Google para puntuar la importancia de los vértices de un grafo sobre la base de cuántos vértices lo enlazan y qué características tienen, entre otros.

*** http://bit.ly/1sai4J1

- **RandomWalkComputation**, que ejecuta un «paseo aleatorio» por los vértices de un grafo.

- **SimpleShortestPathsComputation**, para hallar el camino más corto entre dos vértices de un grafo, basándose en el número de aristas y el peso asignado a estas.

Además, es fácil encontrar librerías de terceros que agrupan algoritmos para Giraph, por ejemplo las del grupo de investigación en bases de datos de la universidad de Leipzig,**** tales como:

**** http://bit.ly/2EpAOMX

- **BTG**: extrae grafos de transacciones de negocio (*business transactions graphs*, BTG).

- **Propagación de etiquetas**: encuentra comunidades dentro de redes mediante la propagación por los vértices de etiquetas identificativas de la co-

munidad. Los vértices migran a la comunidad representada por la mayoría de etiquetas enviadas por sus vecinos.

- **Repartición adaptativa**: particiona un grafo utilizando propagación de etiquetas.

Lectura complementaria

A. Petermann; M. Junghanns; R. Müller; E. Rahm (2014). *BIIIG: Enabling business intelligence with integrated instance graphs. Data Engineering Workshops (ICDEW)*. En: IEEE 30.ª Conferencia internacional de Chicago, IL (págs. 4-11). doi: 10.1109/ICDEW.2014.6818294

1.4. Spark GraphX

GraphX* es la API de Spark para el procesado y la computación paralela de grafos. Tal y como hemos visto con las otras API de Spark, en GraphX la capa de abstracción RDD es la llamada *resilient distributed property graph* y expone una serie de operadores y algoritmos específicos para simplificar el trabajo con grafos.

Lectura complementaria

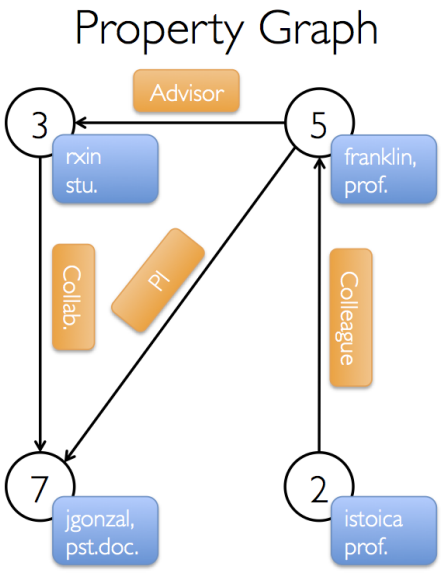
L. M. Vaquero; F. Cuadrado; D. Logothetis; C. Martella (2014). *Adaptive Partitioning for Large-Scale Dynamic Graphs. Distributed Computing Systems (ICDCS)*. En: IEEE 34.ª Conferencia internacional de Madrid (págs. 144-153). doi: 10.1109/ICDCS.2014.23

GraphX es la API de Spark para extender sus funcionalidades para trabajar con grafos, que a su vez son representables como tablas, unificando el procesado de datos y el de grafos en paralelo. En ocasiones el usuario va a preferir representar los datos como tablas en vez de vértices y ejes, de modo que GraphX permite trabajar con RDD que pueden visualizarse a la vez como componentes de un grafo (vértices – VertexRDD; aristas – EdgeRDD) y sus características como colecciones (RDD propiamente dichos).

Por tanto, un grafo en GraphX es solo una vista de los mismos datos, y cada vista tiene sus operadores específicos. Como muestra la figura 11, un mismo grafo puede almacenarse utilizando el formato de vértices o aristas, o bien como una colección de datos tabulares.

* http://bit.ly/1U6KBID

Figura 11. Representaciones de un grafo en GraphX y sus posibles vistas



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Fuente: http://spark.apache.org/graphx/

Apache GraphX, además de las API para escribir programas para analíticas de grafos, distribuye implementaciones de algunos de los algoritmos esenciales para Grafos. En su versión 1.6.1, GraphX soporta los siguientes algoritmos:

1) ***PageRank***, también implementado en Giraph, mide la importancia de cada vértice en un grafo, asumiendo que una arista de un vértice A a un vértice B comporta una validación de la importancia de B para A. Por ejemplo, si una página web es enlazada por muchas otras páginas, esta será catalogada como página importante (motivo para que aparezca en las primeras posiciones de un buscador).

GraphX implementa versiones estáticas y dinámicas de *PageRank*. Las versiones estáticas ejecutan un número fijo de iteraciones, mientras que las dinámicas se ejecutan hasta que los ránquines convergen hasta llegar a una solución aceptable.

GraphX incluye un ejemplo basado en los datos de una red social sobre la que ejecutar *PageRank*.

2) Componentes conectados, conocido como *Connected Components* en inglés, y también implementado en Giraph, es un algoritmo que etiqueta a cada componente (grupo de vértices) conectado del grafo con el identificador del vértice etiquetado con un número menor. Por ejemplo, en una red social, el algoritmo de componentes conectados puede detectar grupos de usuarios (conocidos como clústeres) con algún tipo de afinidad.

GraphX usa los anteriormente mencionados datos de una red social para proporcionar un ejemplo de ejecución del análisis de componentes conectados.

3) Conteo de triángulos. Un vértice forma parte de un triángulo cuando tiene dos vértices adyacentes, también unidos por una arista entre sí. El algoritmo *TriangleCount* de GraphX determina el número de triángulos de los que forma parte un vértice, y se usa para detectar agrupaciones o clústeres.

4) Propagación de etiquetas, también disponible en Giraph, encuentra comunidades dentro de redes mediante la propagación por medio de los vértices de etiquetas identificativas de la comunidad. En dicho algoritmo, cada vértice es inicialmente asignado a una comunidad propia. En cada iteración, los nodos envían su afiliación comunitaria a los vecinos y actualizan su estado según las afiliaciones que reciben de otros vértices.

El algoritmo de propagación de etiquetas es posiblemente el más común a la hora de detectar comunidades en grafos, ya que es computacionalmente muy barato. No obstante, tiene dos desventajas:

- La convergencia hacia una solución aceptable no está garantizada.
- Puede converger hacia una solución trivial (cada nodo forma parte de su propia comunidad individual).

1.5. Visualización de redes

La visualización de redes o grafos es un área situada a caballo entre las matemáticas y las ciencias de la computación, que combina los métodos de la teoría de grafos y la visualización de la información para derivar representaciones bidimensionales de grafos. La visualización de una red es una representación pictórica de los vértices y las aristas que forman el grafo. Este gráfico o representación no debe confundirse con el grafo en sí mismo; configuraciones gráficas muy distintas pueden corresponder al mismo grafo. Durante los procesos de manipulación de los grafos, lo que importa es qué pares de vértices están conectados entre sí mediante aristas. Sin embargo, para la visualización de los grafos, la disposición de estos vértices y aristas dentro del gráfico afecta a su comprensibilidad, a su facilidad de uso y a su estética.

1.5.1. Estrategias de visualización

En este subapartado veremos algunas de las principales estrategias de visualización de redes enfocadas a la distribución de los nodos en un espacio bidimensional. Es decir, estas estrategias se centran en asignar posiciones (o coordenadas) a los nodos de la red con la finalidad de facilitar la visualización de la red, pero no consideran aspectos como el color o el tamaño de los nodos.

Se han definido múltiples métricas para evaluar de forma objetiva la calidad de una estrategia de visualización de grafos. Algunas de estas métricas, además, son utilizadas por las estrategias de disposición como una función objetivo que se trata de optimizar durante la colocación de los vértices para mejorar la visibilidad de la información. Algunas de las métricas más relevantes son las siguientes:

- El número de cruces se refiere al número de pares de aristas que se cruzan entre sí. Generalmente no es posible crear una disposición sin cruces de aristas, pero si se intenta minimizar el número de cruces se suelen obtener representaciones más sencillas y, por lo tanto, más fáciles de interpretar de forma visual.

- El área de representación del grafo es el tamaño del marco delimitador más pequeño. Las representaciones con un área más pequeña son, en general, preferibles a aquellas con un área mayor, ya que permiten que las características del grafo sean más legibles.

- La simetría permite hallar grupos con cierta simetría dentro del grafo dado y mostrar estas simetrías en la representación del grafo, de forma que se puedan identificar rápidamente en una observación visual.

- Es importante representar las aristas y los vértices utilizando formas simples, para facilitar la identificación visual. Generalmente se aconseja representar las aristas utilizando formas rectas y simples, evitando las curvas innecesarias.

- La longitud de las aristas también influye en la representación de un grafo. Es deseable minimizar la longitud de las aristas y, además, mantener longitudes uniformes para todas las aristas que se representen.

- La resolución angular mide los ángulos entre las aristas que salen (o llegan) a un mismo nodo. Si un grafo tiene vértices con un grado muy grande (*hubs*), entonces la resolución angular será pequeña, pero en caso contrario se debe mantener una proporcionalidad que ayuda a la estética y a la legibilidad del grafo.

Existen múltiples estrategias de visualización de grafos, y de forma similar a la mayoría de escenarios, cada una posee ciertos puntos fuertes y débiles. Consecuentemente, es importante conocer estas estrategias básicas para escoger en cada momento la que mejor se adapte al planteamiento y a los datos con los que se está trabajando.

1) La **disposición aleatoria** (*random layout*) es la estrategia más simple de visualización. Consiste en distribuir los vértices de forma aleatoria en el espacio bidimensional y añadir las aristas correspondientes entre los pares de vértices.

2) La **disposición circular** (*circular layout*) coloca los vértices del grafo en un círculo, eligiendo el orden de los vértices alrededor del círculo intentando reducir los cruces de aristas y colocando los vértices adyacentes cerca el uno del otro.

3) Los **diagramas de arcos** (*arc diagrams*) sitúan los vértices sobre una línea imaginaria. A continuación, las aristas se pueden dibujar como semicírculos por encima o por debajo de la línea.

4) Los sistemas de **disposición basados en fuerzas** (*force-based layout*) modifican de forma continua una posición inicial de cada vértice de acuerdo a un sistema de fuerzas basado en la atracción y repulsión entre vértices. Típicamente, estos sistemas combinan fuerzas de atracción entre los vértices adyacentes y las fuerzas de repulsión entre todos los pares de vértices, con el fin de buscar una disposición en la que las longitudes de las aristas sean pequeñas y los vértices estén bien separados para facilitar la visualización.

Matriz de Laplace

La matriz de Laplace es una matriz cuadrada de orden $n \times n$ tal que $L = D - A$, donde D es la matriz diagonal que contiene la suma de filas a lo largo de la diagonal y 0 en las demás posiciones, y A es la matriz de adyacencia.

5) La **disposición basada en el espectro** del grafo (*spectral layout*) utiliza los vectores propios de la matriz de Laplace, u otras matrices relacionadas con la matriz de adyacencia del grafo, como coordenadas de los vértices en el espacio bidimensional.

6) Los métodos de **disposición ortogonal** (*orthogonal layout*) fueron diseñados originalmente para problemas de diseño de VLSI.* Típicamente emplean un enfoque de múltiples fases, en las que se intenta reducir el número de cruces entre aristas, trazar aristas que sean lo más rectas posibles y reducir el espacio de representación del grafo.

*** Integración en escala muy grande (*very large scale integration*) de sistemas de circuitos basados en transistores en circuitos integrados.**

7) La **disposición basada en capas** (*layered-based layout*), a menudo también llamados de estilo Sugiyama, son los más adecuados para grafos dirigidos acíclicos o grafos casi acíclicos. En estos métodos, los vértices del grafo están dispuestos en capas horizontales, de tal manera que la mayoría de aristas se encuentran en posición vertical desde una capa a la siguiente.

Método GEM

Arne Frick, Andreas Ludwig, Heiko Mehldau (1995): *A Fast Adaptive Layout Algorithm for Undirected Graphs*, Proc. Graph Drawing 1994, LNCS 894, pp. 388-403.

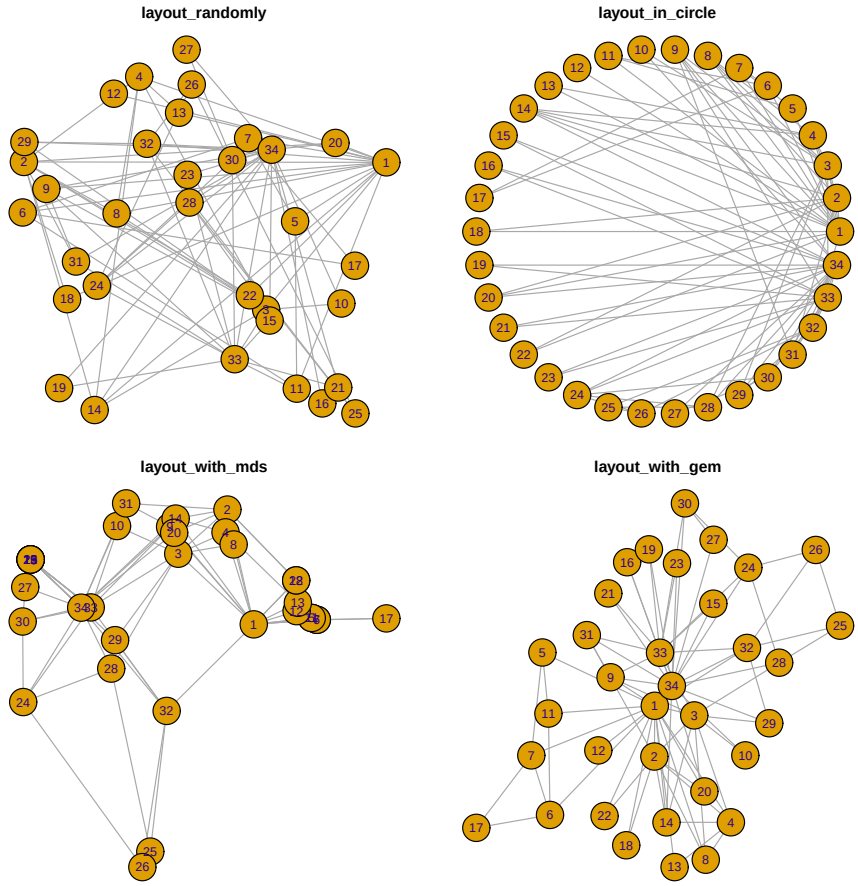
Ejemplo de representaciones de redes

Existe una gran cantidad de algoritmos de visualización de grafos. En este ejemplo, veremos algunos de ellos, que presentamos brevemente a continuación:

- *layout_randomly*: esta disposición coloca los vértices de forma aleatoria en el espacio de dos dimensiones.
- *layout_in_circle*: los vértices son ubicados de forma equidistante entorno a un círculo.
- *layout_with_mds*: el método de escalado dimensional sitúa los puntos de un espacio dimensional grande ($\geq$ 3) en un espacio bidimensional, de manera que se intenta mantener la distancia entre los puntos en el espacio original.
- *layout_with_gem*: este algoritmo sitúa los vértices en el plano utilizando la disposición basada en fuerzas del método GEM.

La figura 12 muestra el resultado de los cuatro algoritmos descritos anteriormente. En primer lugar, se puede ver el algoritmo aleatorio, que muestra los vértices sin ningún tipo de agrupamiento, lo que dificulta su visibilidad. En segundo lugar, vemos la disposición en círculo de los vértices. Esta disposición puede ser útil en algunos casos donde existen pocas aristas y el grafo no presenta ningún tipo de estructura interna. En tercer y cuarto lugar, podemos ver los algoritmos de escalado multidimensional y basado en fuerzas. La disposición de los vértices en ambos casos ayuda a una correcta visualización de la red.

Figura 12. Ejemplos de representación de una red



1.5.2. La problemática del orden

La representación de datos mediante gráficos presenta algunos problemas, especialmente cuando queremos representar grandes volúmenes de datos. La visualización de grafos suele ser muy efectiva cuando se manejan conjuntos de datos de un tamaño razonable, pero la complejidad aumenta mucho cuando se pretenden representar grandes conjuntos de datos, es decir, grafos con un orden (número de vértices) elevado.

Existen dos metodologías básicas para tratar con grafos de orden elevado:

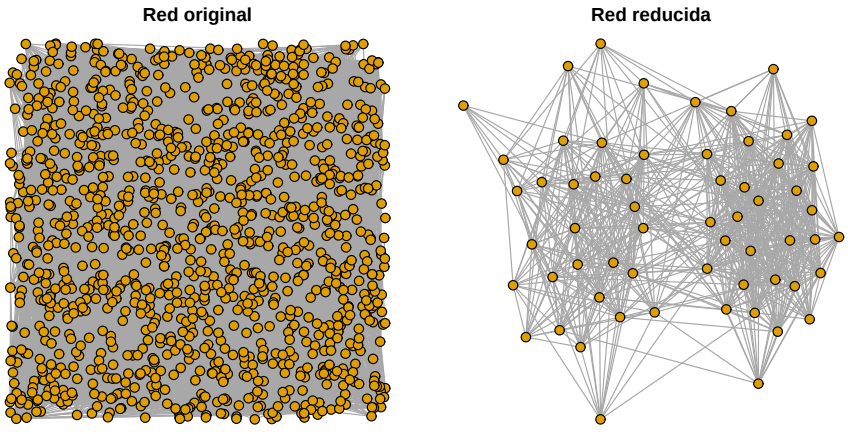
- Una primera alternativa consiste en representar, no el conjunto entero de todos los vértices, sino un subconjunto reducido de valores agregados a partir de los datos originales. De esta forma se pretende reducir el tamaño de los datos y facilitar su visualización.

- Un segundo enfoque consiste en utilizar una estrategia de visualización dinámica, que permita un cierto granulado de la información. Por ejemplo, se puede visualizar la red en un primer momento mostrando solo los vértices más importantes (por ejemplo, los vértices con un grado elevado), y permitir al usuario focalizar la visualización en un punto de la red concreto, mostrando los vértices de menor grado en este punto. Es decir, se trata de hacer un tipo de «zoom» de las distintas zonas que permita ver los «detalles» que puedan quedar ocultos al mostrar solo los vértices más importantes.

Ejemplo del problema de visualización de grafos grandes

La figura 13 muestra la estructura de dos redes. Como se puede observar, la figura de la izquierda muestra una red de tamaño medio, con 1.224 usuarios y 16.715 relaciones entre ellos. Como se puede ver en el gráfico, es muy difícil, si no imposible, obtener información alguna de los datos presentados en esta figura. El número de nodos y aristas es demasiado grande, resultando imposible identificar la estructura de la red, las comunidades que la componen o los nodos principales con un simple análisis visual.

Figura 13. Gráfico de una red social 1.224 vértices (izquierda) y una versión reducida que incluye los sesenta vértices más importantes en función del grado (derecha)



Por otro lado, la figura de la derecha muestra un resumen de esta misma red social, donde solo se muestran los vértices más importantes. En este caso se ha definido la importancia de un vértice utilizando una métrica basada en el grado. Solo los vértices con grado superior a cien se muestran en esta red. Obtenemos, por lo tanto, un resumen de sesenta vértices y setecientas dos relaciones entre ellos. Dado el nuevo volumen de datos mostrados, es mucho más fácil identificar claramente la estructura de la red; así, conseguimos identificar también las distintas comunidades que la forman y los nodos principales, es decir, los individuos que están conectados a un gran número de usuarios y que actúan como concentradores de información (*hubs*).

2. Minería de textos

La minería de textos (*text mining*) es el descubrimiento mediante tareas de análisis de datos de nueva información, previamente desconocida, mediante la extracción automática de información de una cantidad usualmente grande de diferentes recursos textuales no estructurados. En las definiciones habituales de minería de textos se suele hacer hincapié en el factor «descubrimiento de nueva información o conocimiento» resultado del análisis del texto analizado.

Así, por extensión diremos que la minería de textos incluye también un conjunto de herramientas y técnicas que permiten convertir datos de texto no estructurados o semiestructurados en otros formatos, generalmente estructurados, de modo que algoritmos matemáticos de análisis de datos se pueden aplicar sobre dichos datos y procesar grandes volúmenes de información proveniente de formato textual.

2.1. Introducción a la minería de textos

A modo de ejemplo, imaginemos un archivo médico en el cual se mezclan artículos científicos y registros médicos con información acerca de diferentes tratamientos realizados sobre pacientes con patologías similares, entradas de texto realizadas por los propios médicos, tratamientos propuestos y, finalmente, el resultado de su seguimiento.

Un análisis de las diferentes fuentes de texto implicadas (texto introducido por los médicos, tratamientos propuestos, artículos científicos, etc.) podría acabar deduciendo nuevas correlaciones entre síntomas de afecciones o enfermedades, tratamientos y posibles causas de dichas afecciones que *a priori* no eran evidentes o manifiestas. En principio, este conocimiento no está presente en los documentos y recursos de texto, sino que ha sido «descubierto» a partir del análisis de contenido de todas las fuentes de información, descubriendo conexiones y/o correlaciones ocultas, es decir, generando nuevo conocimiento.

Sin embargo, previamente a todo este análisis, es necesario analizar los datos y sus formatos, y ser capaces de extraer la información relevante para cada caso de estudio. Así, como primer objetivo de la minería de textos y previo al descubrimiento de conocimiento, debemos ser capaces de procesar y mostrar la información disponible de las diferentes y extensas fuentes de información en un formato que facilite su comprensión y análisis. Podríamos decir que la

minería de textos es también el proceso de analizar colecciones de materiales textuales para capturar conceptos y temas clave, y descubrir relaciones y tendencias ocultas sin la necesidad de que se conozcan las palabras o los términos exactos que los autores han utilizado para expresar esos conceptos.

La minería de textos a veces se confunde con la recuperación de información. Si bien la recuperación y el almacenamiento exacto de la información es un enorme desafío, la extracción y gestión del contenido, la terminología y las relaciones de calidad contenidos en la información son procesos cruciales y críticos, así podríamos decir que la minería de texto extiende o incluye la extracción de información en su proceso de ejecución, como veremos más adelante.

La minería de textos tiene algunos puntos en común con otras disciplinas muy habituales, como son:

- **Minería de datos**: podríamos decir que la minería de textos es una disciplina complementaria a la minería de datos, con la diferencia principal de que no trabaja con datos estructurados provenientes de (grandes) bases de datos o repositorios más o menos estructurados, sino con fuentes de datos de texto no estructuradas o semiestructuradas. Así, comparten el mismo objetivo y utilizan técnicas parecidas, solo que la fuente de datos objeto del estudio es de naturaleza distinta.

- **Recuperación de documentos e información**: la búsqueda y la recuperación de información cubre la indexación, la búsqueda y la recuperación de documentos de texto sobre grandes bases de datos utilizando consultas mediante palabras clave. Con el surgimiento de poderosos motores de búsqueda en internet, como por ejemplo Google o Yahoo!, la búsqueda y recuperación de información se convirtió en una funcionalidad muy extendida y familiar para la mayoría de la gente. Actualmente casi todas las aplicaciones existentes, desde el correo electrónico al procesamiento de textos incluye una función de búsqueda. Sin embargo, la recuperación textual no pretende facilitar el proceso de análisis ni la extracción de nuevo conocimiento, como sí pretende la minería de textos, sino que el objetivo último es identificar los documentos relevantes para un usuario dentro de una colección, lo que sería un paso previo a la realización de minería de textos.

- **Lingüística computacional**: agrupa una serie de técnicas para procesar textos y tratar de hacerlos comprensibles para un ordenador, permitiendo el análisis sintáctico y gramatical de textos en formato electrónico, la alineación e identificación de correspondencias entre textos escritos en diferentes idiomas, etc. Su aplicación más conocida son los sistemas de traducción automática. En este caso, si bien la minería de textos ha adoptado algunas de las técnicas desarrolladas por esta disciplina, sus objetivos son claramente diferentes.

2.2. Aplicaciones de la minería de textos

Para comprender la minería de textos nos será útil presentar algunas de sus aplicaciones más habituales:

- **Extracción de información**: se refiere a la localización de datos específicos en documentos de texto no estructurados de los cuales se extrae información estructurada. Típicamente la información «extraída» será almacenada en repositorios de datos estructurados, como bases de datos, para su posterior estudio mediante herramientas de minería de datos tradicionales. A modo de ejemplo, se podrían querer extraer información de un texto referente a los nombres de personas, empresas, cargos y datos de contactos.

  Si bien es posible que esta extracción de datos relevantes (también llamados «hechos» o *facts*) se apoye en diccionarios y listados existentes, las aplicaciones de minería de textos deberían ser capaces de identificar de forma automática y no supervisada (sin clasificación previa) aquellos fragmentos o cadenas de texto que hagan referencia a personas, organizaciones y eventos de los que no se tengan referencias previas.

- **Clasificación de documentos**: también llamada categorización de texto, es la clasificación de documentos con formato de texto libre no estructurado a partir de unas clasificaciones previamente definidas. El proceso se realiza de forma automática y clasifica los documentos a partir de su contenido u otro criterio.

  La clasificación no se restringe a todo el contenido de un documento, sino que pueden utilizarse párrafos, resúmenes de artículos u otras porciones de texto para la categorización. Un ejemplo de esta aplicación sería la asignación automática de una categoría o materia que un sistema de clasificación bibliográfico asigna a un documento.

  Existe una analogía evidente con los llamados sistemas de clasificación supervisada en minería de datos, y es que se trata de la misma tarea pero realizada sobre datos con formato textual no estructurado. Así, la clasificación automática requiere de un entrenamiento previo del programa encargado de realizarla. Como en todo proceso de clasificación, es necesario contar con una muestra de documentos previamente clasificados para entrenar nuestro modelo de predicción. De esta forma, el programa podrá analizar las características que determinan la asignación de los documentos a una u otra clase, clasificando los documentos por su similitud.

  La categorización puede ser tan granular como se quiera, desde una sola categoría a un amplio conjunto de categorías que incluyan múltiples niveles, es decir, subcategorías dentro de otras categorías.

Para poder categorizar un texto es necesario representarlo de algún modo, esto es, saber cuáles son las características de dicho texto. Si son palabras, es necesario utilizar alguna representación del documento o porción de texto, como podría ser el modelo bolsa de palabras (*bag of words*) o el modelo vectorial, que introduciremos en apartados posteriores.

- **Agrupación de documentos**: así como la clasificación de documentos es equivalente a la clasificación supervisada en minería de datos, la agrupación de documentos (*document clustering*) es el aprendizaje «no supervisado», en el que no hay categorías o clases predefinidas, sino que los documentos se agrupan a partir de categorías y características generadas automáticamente por el modelo de predicción. Los documentos «similares» se agrupan, de modo que el elemento clave es poder caracterizar correctamente nuestros documentos para identificar de forma correcta cuál es la característica que determina qué documentos son similares.

  Aunque son técnicas particularmente interesantes debido a que no necesitan un entrenamiento previo, sus aplicaciones reales en el campo de la minería de textos no han sido tan extendidas como la clasificación supervisada comentada en el punto anterior. El tiempo de procesamiento puede ser particularmente significativo en aplicaciones de agrupamiento con datos textuales, en el cual las instancias pueden ser descritas por cientos o miles de atributos, incluir atributos «conceptuales» o ambigüedad semántica.

- **Minería de webs** (*web mining*) es la aplicación de las técnicas de minería de datos a los datos generados por el tráfico y uso de internet, con el objetivo de descubrir de nueva información derivada de su uso.

  Este es un campo tan amplio que puede segmentarse en diferentes campos de estudio, como serían:

  1) **Minería del contenido web (*web content mining*)**: focalizada en el análisis del contenido de una web, también se le llama minería de textos web, ya que sobre el contenido textual de una web se aplican las técnicas ya mencionadas (clasificación supervisada o no supervisada, extracción de información, procesamiento de lenguaje natural, etc).
  2) **Minería del uso web (*web usage mining*)**: se basa en realizar minería sobre el registro de navegación de una web y sus incidencias. Permite analizar registros de navegación de usuarios, incidencias relativas a la seguridad, intentos de acceso a webs por parte de usuarios no autorizados o, desde un punto de vista más orientado al marketing, permite modelizar perfiles de usuario y los contenidos a los que acceden.
  3) **Minería de la estructura web (*web structure mining*)**: existe una tercera rama de aplicación de la minería de webs que permite el análisis de la estructura de una web. A partir de teoría de grafos y del análisis de

nodos y aristas de dichos grafos, se pueden encontrar nuevas relaciones entre páginas webs diversas.

- **Procesamiento de lenguaje natural** (*natural language processing*, NLP): tiene una historia relativamente larga, tanto en lingüística como en computación. Este campo de estudio combina las técnicas de aprendizaje automático e inteligencia artificial con la lingüística, con el objetivo de procesar el lenguaje humano por parte de un computador. Su campo de actividad es variado e incluye, por ejemplo, los sistemas de traducción automáticos o la generación de textos automáticos.

  El desarrollo de las aplicaciones de la NLP es un desafío porque los ordenadores tradicionalmente requieren que los seres humanos «hablen» con ellos en un lenguaje que sea preciso, inequívoco y altamente estructurado o, quizás por medio de un número limitado de comandos de voz claramente enunciados. El discurso humano, sin embargo, no siempre es preciso; a menudo es ambiguo y la estructura lingüística puede depender de muchas variables complejas, como la jerga, los dialectos regionales, el contexto social, la ironía, etc.

  NLP también puede ser útil para proporcionar variables de entrada útiles para la minería de textos. Así, las tareas comunes de NLP en los programas de software de hoy incluyen la segmentación de oraciones, reconocimiento de voz, análisis sintáctico, extracción de conceptos o resolución de ambigüedades, entre muchos otros.

- **Extracción de conceptos y hechos**: además de información relevante de un documento o texto podemos extraer «conceptos» de un texto en forma de relaciones conceptuales a partir de la extracción de palabras y un léxico inicial, agrupando palabras o frases a partir de criterios semánticos. Por ejemplo, no es lo mismo decir que «una nube de crecimiento vertical puede provocar una tormenta» que «vamos a subir nuestros archivos a la nube».

  Esta función permitiría extraer los principales temas o ideas tratados en los documentos. No se trata de un proceso de clasificación sino de extraer un conjunto de términos que son representativos del contenido de los documentos, lo que sería de mucha utilidad para una clasificación posterior.

  Así, a diferencia de los mecanismos de indexación automática que extraen términos como aparecen en los documentos o textos, la identificación de conceptos es más compleja y representaría la capacidad de identificar una idea sobre la que trata un documento a partir de la identificación de determinados términos y combinaciones de términos en el documento.

2.3. Metodología y procesos de la minería de textos

Una vez descritos los principales campos de aplicación de la minería de textos, vamos a describir cuál sería la metodología que seguir en un proceso e minería de textos, en el cual identificaremos algunas de las aplicaciones anteriores.

2.3.1. Preprocesamiento

Como ya se ha comentado, las fuentes de texto son no estructuradas, ambiguas, etc. Así pues, la etapa de preprocesamiento tiene una gran importancia para el desarrollo de nuestra cadena de trabajo de minería, ya que un mal preprocesamiento puede comprometer el resto de etapas. El preprocesamiento incluye:

- **Limpieza**: debemos imponer reglas de procesamiento, tales como la limpieza del texto y otras fuentes de datos que no sean relevantes en el contexto del estudio (¿el texto contiene tablas? ¿figuras? ¿anuncios?).

- **Tokenización**: es el proceso de separación del texto en palabras. ¿Son los espacios relevantes? ¿Cómo tratamos los apóstrofes? Las palabras con un mismo origen (alumno/alumna) ¿deben tener el mismo tratamiento o las consideramos palabras completamente distintas (lematización)?

- **Desambiguación del significado de las palabras**. ¿Cuál es el significado más probable de una palabra dado el contexto? Utilizando las técnicas descritas en el subapartado anterior debemos determinar las estructuras semánticas. Relacionado con la contextualización, identificamos segmentos de texto que tienen un significado específico que, en caso de separar las palabras sin mantener un contexto, perderían el significado que quiere transmitir.

2.3.2. Transformación del texto

El texto o documento se representa como un conjunto de palabras (o características, utilizando la terminología de minería de datos), así como las veces que estas se repiten en dicho texto. Sin embargo, para parametrizar nuestro documento para posteriores estudios podemos utilizar un modelo de representación de documentos, conocido como «modelo bolsa de palabras» (*bag of words*). El documento se representa como un listado de palabras ignorando el orden de estas y el número de veces que aparecen en un documento.

Si queremos comparar varios documentos a partir del número de ocurrencias de cada palabra, normalizaremos el peso de las ocurrencias por el número total de palabras para unificar el criterio de comparación.

Cada documento estará compuesto por un listado de palabras y su peso relativo en el documento, lo que llamaremos vector. La suma de los pesos de todos los elementos de cada vector (un vector por cada documento), es igual a 1 a causa de la normalización. Este es el llamado modelo vectorial.

Este modelo nos permite caracterizar documentos a partir de su «vector», permitiendo, por ejemplo, la comparación entre documentos y su clasificación.

Un segundo paso de la transformación del texto es la selección de características (palabras). Una vez representado el documento, podemos seleccionar aquellas palabras que van a ser más necesarias en nuestro análisis. Así, palabras como: *la*, *a*, *como*, *una*, aparecerán muchas veces en nuestro texto y en muchos documentos, pero probablemente aportarán poca información a nuestro análisis. Una vez eliminadas dichas características puede volver a normalizarse el vector.

Con el texto caracterizado y habiendo seleccionado las características que mejor definen nuestros documentos, podemos aplicar un clasificador de documentos, tal y como se ha presentado en el apartado anterior, por ejemplo.

En cualquier caso, una vez completado este paso, los datos se almacenarán en estructuras que permitan el análisis estructurado de la información, tales como bases de datos relacionales o repositorios propios de datos masivos como sistemas de ficheros distribuidos (HDFS) o bases de datos NoSQL.

2.3.3. Selección de atributos y reducción de dimensionalidad

Puede ser útil discriminar, en un proceso de selección de características más fino, aquellas palabras que tienen poco peso en el análisis que estamos realizando mediante un proceso de reducción de dimensionalidad (por ejemplo, empleando el algoritmo de análisis de componentes principales o PCA), lo que permite reducir el número de dimensiones de nuestro problema.

Este paso se distingue de la limpieza de palabras anterior en que aquí tenemos en cuenta palabras que tienen un peso moderado en los documentos que estamos analizando (a diferencia de los determinantes o las preposiciones, por ejemplo) pero que, tras un análisis de dimensionalidad, las podemos descartar porque hemos determinado que tienen poco peso en la caracterización del problema. Esto nos permite reducir la dimensionalidad, que suele ser una de las mayores dificultades con las que nos encontramos en tareas de minería de datos.

2.3.4. Minería de datos y descubrimiento de patrones

Una vez llegados a este punto el documento ya se ha parametrizado y representado de una forma precisa, de modo que a partir de aquí se aplicarán las técnicas de minería de datos y de aprendizaje automático necesarias para

resolver el problema actual, como por ejemplo una clasificación no supervisada (*clustering*) sobre multitud de documentos o una clasificación utilizando un conjunto predeterminado de documentos de entrenamiento. Así, hemos transformado el problema en una tarea de minería de datos clásico sobre el cual se pueden aplicar las técnicas habituales.

2.3.5. Interpretación y evaluación

Una vez completado al análisis podemos realizar las evaluaciones necesarias para volver a ejecutar la tarea para mejorar alguno de los pasos (visualizando el resultado), o finalizar la cadena de procesamiento, llegando a los resultados y a las conclusiones pertinentes.

2.4. Apache Solr y Elasticsearch

Elasticsearch* es un motor de búsqueda basado en Apache Lucene.** Proporciona un motor de búsqueda de texto completo distribuido y accesible para múltiples usuarios con una interfaz web HTTP y documentos JSON. Elasticsearch está desarrollado en Java y publicado como código abierto bajo los términos de la licencia de Apache.

*** http://www.elastic.co**
**** http://lucene.apache.org**

Elasticsearch se desarrolla junto con un motor de recopilación de datos y análisis de registros llamado Logstash, y una plataforma de análisis y visualización llamada Kibana. Los tres productos están diseñados para su uso como una solución integrada, conocida como Elastic Stack.

Apache Solr*** es una plataforma de búsquedas *open source*, también basada en el proyecto Apache Lucene. Se trata de un entorno desarrollado en Java que se ejecuta como un servidor de búsqueda de texto completo independiente dentro de un contenedor de *servlets*, como podría ser Apache Tomcat o Jetty. Utiliza una biblioteca Java para la indexación de texto completo y de búsqueda. Además, proporciona una sólida base de API, lo que permite una fácil integración con casi cualquier lenguaje de programación.

***** http://bit.ly/1nrcsFV**

Sus características principales incluyen: motor de búsqueda de texto completo, análisis de textos, resaltado de afectados, agrupamiento dinámico, integración de bases de datos y manipulación de documentos de texto enriquecido (por ejemplo, Microsoft Word o PDF). Proporciona búsqueda distribuida, replicación de índices y es altamente escalable. Más concretamente, Solr permite:

- realizar peticiones HTTP para indexar o consultar documentos al estilo REST permitiendo la recuperación de documentos en formato XML y JSON;
- manejar documentos de texto enriquecido;

- soportar múltiples sintaxis de consulta sobre documentos, y hacer consultas y combinación de consultas, similares a los *join* en SQL, sobre documentos;

- buscar de forma «facetada», esto es, búsqueda de documentos categorizados; esta es una funcionalidad muy importante, ya que permite aplicar algunas de las técnicas ya descritas en el proceso de minería de textos, como la clasificación de documentos o la búsqueda de documentos similares;

- indexar documentos mediante ficheros XML, JSON y CSV;

- llevar a cabo funcionalidades avanzadas de minería de textos como, por ejemplo, detección de idioma de un documento, traducción, funcionalidades NLP y tareas de clasificación de documentos;

- SorlCloud* permite el despliegue de Sorl en entornos distribuidos.

* http://bit.ly/2CgDbUJ

Elasticsearch y Solr son, actualmente, los dos entornos de trabajo líderes en procesamiento y búsqueda de texto. Ambos proporcionan un amplio conjunto de operaciones sobre textos. Aunque un análisis detallado de las diferencias entre ambos escapa al objetivo de este texto, se pueden encontrar artículos interesantes que comparan ambos que nos pueden ayudar a decidirnos por uno u otro sistema dado un problema concreto.

Enlace de interés

Para una comparación entre Elasticsearch y Solr podéis consultar la siguiente dirección web: http://bit.ly/1gQc10h

3. *Streaming*: análisis de datos en tiempo real

En el caso del procesamiento de datos en *stream*, los datos deberán ser procesados secuencial e incrementalmente a medida que vayan llegando. La granularidad de los datos por procesar pueden variar según el sistema usado, desde un procesado registro a registro a un procesado por ventanas de tiempo predeterminadas. El tipo de análisis que se puede hacer sobre estos tipos de datos es equivalente a otros análisis que hemos visto en otros contextos (agregación, agrupamiento, clasificación, etc.), pero cuando tratamos con datos en *stream* nos solemos encontrar con una característica distintiva: la necesidad de obtener información en tiempo real. En consecuencia, muchos de los sistemas de procesamiento de *streams* están pensados para realizar análisis simples en tiempo real.

Conceptualmente, la manera más simple de procesar un flujo de datos continuo sería agregando su contenido. En tal caso, habrá que identificar los indicadores de interés y ir recalculándolos a medida que se vayan leyendo los datos de origen. Vamos a ver un ejemplo de cómo podría hacerse.

Supongamos que tenemos un *stream* donde nos llega información sobre los «me gusta» que se han realizado sobre recursos propios. Cada entrada del *stream* contiene el identificador del recurso donde se ha hecho el «me gusta», el identificador del usuario que lo ha hecho y la marca de tiempo que indica cuándo se hizo. Supongamos que queremos identificar el número total de «me gusta» que ha recibido cada uno de nuestros recursos. Supongamos también que tenemos una tabla que almacena la información de interés de forma agregada, compuesta por dos campos: un identificador del recurso y el número total de «me gusta» que se han hecho sobre el mismo. En este caso el proceso de análisis sería simple: por cada entrada del *stream* leída se incrementaría en uno el número total de «me gusta» de la tabla de agregados para el identificador de recurso indicado. En un caso como este, no habría la necesidad de guardar los datos recibidos por el *stream*, sino un resumen de los mismos.

La aproximación anterior no es válida en todos los casos. En otros casos, el sistema deberá almacenar los datos generados (o parte de ellos) para poder hacer analíticas complejas, para proveer de datos a terceros o simplemente para poblar un almacén de datos (*data warehouse*). Además, existen algunos tipos de análisis para los cuales calcular el valor del estado final de un conjunto de datos es insuficiente, porque se requiere conocer el estado final junto con todos los cambios que han conducido a dicho estado. Un ejemplo lo tendría-

mos al realizar analíticas sobre la cesta de la compra de un portal de venta en línea. Suponed que queremos analizar la cesta de la compra para identificar compradores dudosos y animarlos a comprar más unidades de productos. Los compradores dudosos son aquellos que, antes de realizar la compra, tienden a reducir y aumentar la cantidad de los productos de la cesta. Para un sistema como el propuesto, sería necesario analizar, en tiempo real, qué cambios de estado realiza el usuario en la cesta de la compra. Por tanto, se debería implementar un proceso que lea los datos del *stream* (que indican los cambios realizados en la cesta de la compra) y tome decisiones en tiempo real para garantizar que el usuario no reduce el numero de unidades antes de la compra. Probablemente en este contexto sería necesario también almacenar todos los cambios realizados en la cesta de la compra y el resultado final (número de unidades vendidas) para entrenar sistemas de aprendizaje de máquina que nos permitan realizar analíticas más potentes al respecto.

Estos dos ejemplos resumen dos de las actividades principales que puede realizar un sistema consumidor al procesar un conjunto de datos en *streaming*: por un lado, agregación y resumen de datos y, por otro, almacenamiento y análisis de los datos del *stream*. Entre estos dos extremos existen soluciones mixtas, como por ejemplo la creación de nuevos *streams* que resuman los datos originales o que los enriquezcan con datos de negocio, datos calculados o datos provenientes de otros *streams*.

Los escenarios que motivan el análisis de datos mediante herramientas de *streaming* son:

- La necesidad de calcular estadísticas sobre datos que se generan al vuelo.

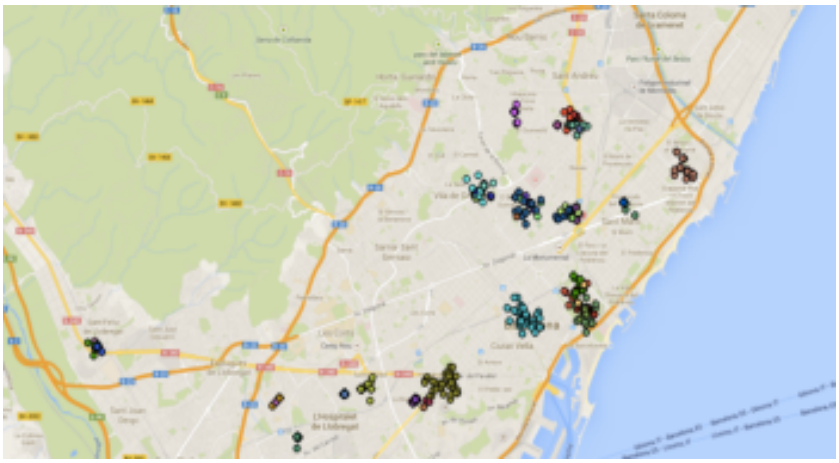
- La necesidad de calcular estadísticas sobre un conjunto de datos cuando este es demasiado grande. Una solución es ir precomputando los datos a medida que se generan.

Cuando se adopta un flujo de trabajo en modo de paquetes por lotes (también conocido como *batch*), no existe una garantía de que estos vayan a ser procesados en un tiempo corto, por lo que los datos que vamos a calcular pueden estar desactualizados.

Ejemplo: las fiestas de la Mercè

Como ejemplo imaginario, durante las fiestas de la Mercè, el ayuntamiento de Barcelona pretende localizar los recursos asistenciales y de seguridad (policía, ambulancias...) allá donde se detecte una mayor concentración de personas. Dicha concentración puede cambiar a lo largo de la noche (cuando acaban los conciertos del Fòrum, algunas personas irán al parque de La Ciutadella y otras a Plaça Catalunya). En un alarde de optimización, el ayuntamiento pretende saber al momento cuáles son las mayores concentraciones de personas, y para ello procesará los datos geolocalizados que los asistentes a las fiestas generan en cada momento (tuits, fotografías de Instagram, comentarios de Facebook, etc).

Figura 14. Concentración de personas en las fiestas de la Mercè de 2014, sobre la base de 65.918 tuits



En una solución de paquetes por lotes, tal como cualquiera de las que implemente el modelo MapReduce, el sistema debería ir almacenando los datos y, llegado un momento, procesar las estadísticas sobre estos. El ayuntamiento podrá tener una evolución de las concentraciones de personas a lo largo de la noche, pero es posible que no tenga tiempo de reacción ante cambios repentinos en la distribución de las personas, ya que casi siempre habrá un número considerable de eventos «no absorbidos» por el sistema de análisis de datos.

Los motores de procesado de datos por flujos (también conocidos como *streaming*) surgen para habilitar el análisis de datos en tiempo real (es decir, se consumen y procesan inmediatamente después de que sean producidos).

Las características deseables de estos son:

- Garantizan que los datos van a ser procesados en un tiempo razonable.
- Son horizontalmente escalables: añadiendo más recursos de computación podremos procesar más datos.
- Son tolerantes a fallos: si un nodo de computación falla, los siguientes nodos pueden absorber su carga trabajo.
- Son rápidos.

En este apartado se muestran las características de cuatro herramientas para el procesado de *streaming* que actualmente gozan de gran difusión y proyección:

- Apache Hadoop Storm
- Apache Spark Streaming
- Apache Flink
- Data Stream Management Systems

3.1. Hadoop Storm

Storm es el motor de procesado en *streaming* para Apache Hadoop que relaciona los datos con sus funciones de procesado según un grafo acíclico dirigido.

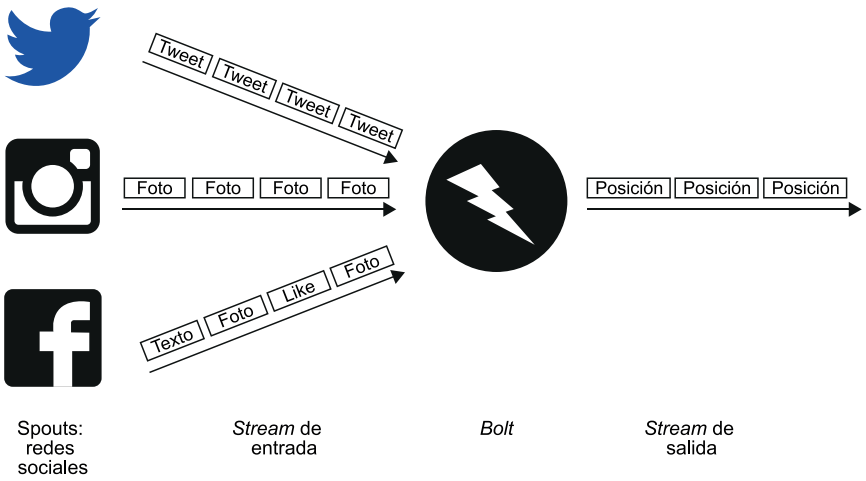
La unidad básica de datos es la *n*-tupla, un conjunto de datos que representan, por ejemplo, un evento. En el ejemplo al inicio de este apartado, una *n*-tupla podría guardar la latitud y la longitud de una foto geolocalizada, así como el identificador único de su usuario.

Los flujos o *streams* serían una secuencia no acotada de tuplas. Y un *spout* (canalón, en español) una fuente de flujos de datos.

Un *bolt* (algo así como una vía de desagüe) consume datos de uno o varios *streams* y produce nuevos flujos de datos.

La figura 15 muestra visualmente los conceptos anteriormente descritos, así como la relación entre estos.

Figura 15. Un ejemplo de topología Storm para el procesado de eventos



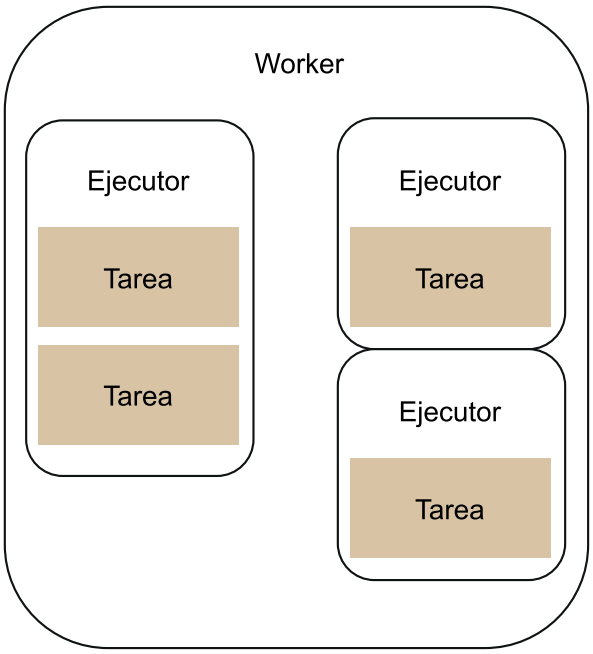
A la hora de especificar la implementación de la topología de un sistema Storm (es decir, qué componentes desplegar y cómo se relacionan entre estos), se pueden especificar tres tipos de entidades (figura 16):

- **Tareas** que ejecutan el procesado de datos. Cada *spout* o *bolt* implementado en el código ejecuta tareas. El número de tareas de un componente siempre es el mismo durante la vida de una topología.

- **Ejecutores**, hilos que pueden ejecutar una o varias tareas de un mismo componente (*spout* o *bolt*). El número de ejecutores puede variar en el tiempo.

- Procesos **worker** (trabajador), que ejecutan un subconjunto de una topología, pudiendo ejecutar uno o varios ejecutores que a su vez contienen uno o varios componentes (*spouts* o *bolts*). Una topología consistirá en varios procesos *worker* ejecutándose en varias máquinas de un clúster.

Figura 16. Arquitectura de un proceso *worker*



3.1.1. Topologías predefinidas

Apache Storm no es más que un motor de procesado, por lo que no provee herramientas analíticas, aunque sí anexa muchos ejemplos e implementaciones de métodos comunes que pueden ser adaptados por el programador, como por ejemplo:

- **ReachTopology** crea una topología que computa, mediante paralelización y en tiempo real, el alcance (a cuántas personas llega y cómo se transmite entre seguidores) de cualquier URL en Twitter. Para computar esta métrica, conocida como *reach*, es necesario obtener a toda la gente que tuiteó la URL, obtener todos los seguidores de estas personas y crear un conjunto sin duplicados con estas personas. Es una computación tan intensa que puede requerir miles de llamadas a bases de datos y decenas de millones de registros de seguidores.

- **RollingTopWords** computa continuamente las principales palabras que han pasado por la topología en términos de cardinalidad. El algoritmo está implementado de una manera escalable y puede adaptarse para computar otros conceptos como temas o imágenes populares del momento en una red social.

- **SlidingWindowTopology** demuestra el uso de ventanas de tiempo para realizar cálculos sobre un conjunto de las últimas tuplas procesadas.

- **SlidingTupleTsTopology**, como el anterior, pero ordenando las tuplas por el momento en que fueron emitidas, en vez de cuándo han sido procesadas.

- **TransactionalGlobalCount** es un ejemplo básico de topología transaccional, que guarda la cuenta del número de tuplas vistas hasta el momento en una base de datos.

- **TransactionalWords** usa como base la implementación anterior de *TransactionalGlobalCount* para procesar un flujo de palabras y producir dos salidas:

  1) cuántas veces aparece una palabra en el texto;

  2) el número de palabras que aparecen un rango de veces, es decir, cuántas palabras aparecen de 0 a 9 veces, cuántas aparecen de 10 a 19 veces, etc.

- **WordCountTopology** demuestra las capacidades que Storm proporciona para agrupar *streams*.

3.1.2. Tipos de *bolts*

Apache Storm provee numerosos *bolts* que facilitan y agilizan el diseño de topologías dedicadas al análisis de datos. Además de los *bolts* que conectan con sistemas de persistencia (HBase, HDFS, Cassandra...), Storm provee también los siguientes *bolts*:

- **AbstractRankerBolt**, que implementa el comportamiento básico y común de los **bolts** que categorizan objetos según su numerosidad. Permite que las implementaciones de esta clase abstracta especifiquen cómo las tuplas recibidas deben ser procesadas (por ejemplo, cómo los objetos contenidos en dichas tuplas deben ser extraídos y contados).

- **IntermediateRankingBolt**, que implementa *AbstractRankerBolt* y categoriza objetos según su numerosidad. Asume que las tuplas de entrada tienen el formato (objeto, numero_objetos, campoAdicional1, campoAdicional2, . . . , campoAdicionalN).

- **TotalRankingBolt**, que recibe ránquines intermedios como los generados por *IntermediateRankingBolt* y los mezcla en un ranquin final, consolidado.

- **RollingCountBolt**, que cuenta los objetos que se han enumerado en una ventana de tiempo definida. Los objetos más viejos que dicha ventana de tiempo serán eliminados de la cuenta.

- **SlidingWindowSumBolt**, que realiza un sumatorio de los valores de tuplas recibidas en una ventana de tiempo recibida. Los valores más viejos que dicha ventana de tiempo serán eliminados del sumatorio.

3.1.3. Agrupamiento de *streams*

Storm también provee algunas primitivas básicas para agrupar los diferentes flujos de datos y, dada una tupla que se pretende emitir, ayuda a decidir a qué tarea debe ir a parar.

Las agrupaciones provistas son:

- **Mezcla** aleatoria, típicamente usada para emitir de manera distribuida.
- **Por campos**, según el valor de un campo de una tupla (equivalente al SQL GROUP BY).
- **A todas**, replicará un flujo hacia todos los *bolts* conectados. Se debe usar con cuidado para no sobrecargar el sistema.
- **Directa**, donde el productor de las tuplas decidirá qué tarea será la receptora de estas.
- **LocalOrShuffle** (local o mezclada), si el *bolt* de destino alberga varias tareas en el mismo proceso *worker*, las tuplas serán mezcladas aleatoriamente dentro de esas tareas, si no, se comportará como una mezcla aleatoria normal.

3.2. Spark Streaming

El hecho de que Spark proporcione en un mismo entorno de trabajo los dos modos de procesado (por lotes o *batch* y *streaming*) es uno de sus puntos fuertes frente a otras alternativas especializadas en uno u otro (por ejemplo Hadoop para el modo *batch* o Storm para el modo *streaming*). La integración de ambos paradigmas convierte a Spark en una implementación de lo que a nivel teórico se conoce como arquitectura *lambda*.

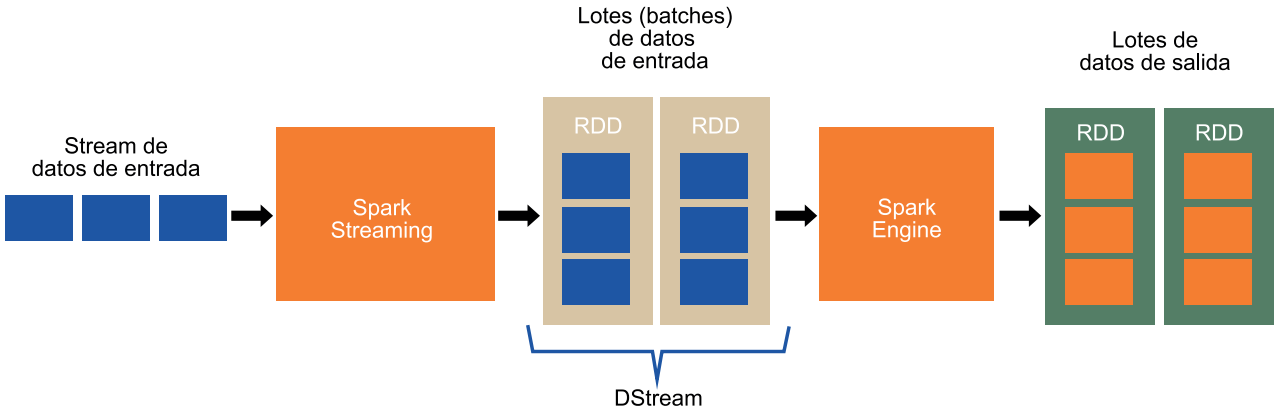
RDD

RDD es la abreviatura de Resilient-Distributed Dataset, o conjunto de datos resiliente y distribuido. La unidad mínima de procesado para Spark

La manera en que Spark incorpora el procesado de flujos de datos sin la necesidad de rehacer todos los demás componentes de la plataforma consiste en capturar pequeños paquetes de datos del flujo de entrada cada cierto intervalo de tiempo y presentarlos como un RDD. De este modo, el flujo de entrada toma la forma de una serie continua de RDD, lo que se denomina un DStream (del inglés *discretized stream*), el concepto esencial y sobre el que gira todo Spark Streaming.

Del mismo modo que ocurre con los RDD, se expresará el procesado del flujo de datos mediante transformaciones sobre los DStreams. Existen una serie de transformaciones estándar para su procesado, aunque también se podrán aplicar transformaciones genéricas de RDD sobre los RDD que los componen.

Figura 17. Esquema del funcionamiento de Spark Streaming y un DStream



Cuando se procesan flujos de datos cobra mucha importancia el modo en que los datos entran y salen del sistema. En el procesado de flujos de datos normalmente intervienen múltiples sistemas, y conviene encontrar una manera cómoda y eficiente de intercambiar los datos entre ellos. Spark facilita esta tarea proporcionando utilidades para la recepción de datos de muchas fuentes (Apache Kafka, Apache Flume, Amazon Kinesis, HDFS, *sockets* y más) y para el envío de datos con todas las alternativas ya descritas para los RDD. Aparte de los conectores que ya proporciona Spark Streaming, existe la posibilidad de definir conectores a medida para interactuar con sistemas todavía no soportados.

Igual que con Apache Storm, Spark Streaming es solo un motor de procesado de flujos de datos, y su núcleo no proporciona herramientas para análisis estadístico. No obstante, el paquete de clases *org.apache.spark.examples.streaming* proporciona muchos ejemplos que pueden utilizarse como punto de partida para la creación de aplicaciones de análisis estadístico de flujos de datos en tiempo real. Cabe destacar:

- ***WordCount**, que implementa receptores de *streams* para diferentes fuentes de datos (Amazon Kinesis, Kafka, HDFS, Flume...) y cuenta las palabras emitidas por estas.

- ***EventCount**, que implementa receptores de *streams* para diferentes fuentes de datos y cuenta los diferentes eventos que pueden ser emitidos por estos.

- **PageViewStreams**, que analiza un flujo de visitas a páginas web, demostrando además varios tipos de operadores disponibles en Spark Streaming, tales como contadores, contadores según valor, *joins*, agrupaciones por clave, etc.

3.3. Apache Flink

Apache Flink, pese a ser más reciente e inmaduro que Storm y Spark Streaming, se está abriendo paso debido a sus ventajas para el procesado de flujos de datos. Mientras Spark y Storm procesan en *micro-batch* (es decir, pequeños paquetes de lotes), Flink procesa en *streaming* real, lo cual conlleva una menor latencia en el procesado en tiempo real.

Otras ventajas de Flink son:

- Se adapta automáticamente a los conjuntos de datos que le van llegando, y Spark/Storm necesitan optimizar y ajustar los trabajos manualmente a conjuntos individuales.
- Puede proveer resultados intermedios cuando se requiera.
- Provee una interfaz web para mandar y ejecutar trabajos remotamente mediante Apache Zeppelin.

Cada tipo de procesado (*batch* o *streaming*) tiene sus ventajas y sus inconvenientes, y son más adecuados para diferentes tipos de situación. Streaming suele ser más adecuado cuando tenemos alguna de las situaciones siguientes:

- Datos de entrada que llegan en forma de registros, en una secuencia específica.
- Se requiere una salida de datos tan pronto como sea posible, aunque no antes del tiempo requerido para analizar/verificar la secuencia.
- La salida no necesita ser modificada una vez ha sido escrita.

Por ejemplo, se puede considerar la media móvil del tiempo que cada visitante permanece visitando una página web. En el caso de usar *batch*, el número de visitas puede ser actualizado cada minuto, hora, e incluso día. El problema es que sería difícil definir cuándo empieza o acaba la sesión, así como los períodos de inactividad, ya que estos puntos que marcan el inicio/fin de las sesiones podría caer en diferentes lotes. Por lo que, en este caso, se requiere procesar datos en tiempo real mediante *streaming*, en el que se puede ver el tiempo como un continuo.

Hay otras situaciones en las que tanto el modelo de *streaming* y *batch* son útiles. Por ejemplo, si se quisiera calcular el mensual móvil de ventas en intervalos de días. En este caso sería necesario computar las ventas diarias totales y al final hacer una suma acumulativa de los últimos treinta días. En este caso, puede ser suficiente con el procesado en *batch* de las ventas según sus respectivas fechas. Incluso es aceptable que pueda haber latencia en los datos, pudiendo agregar registros tardíos a los futuros lotes de procesado.

3.4. Sistemas de gestión de *streams* de datos

El área de procesamiento de datos en *stream* es un ámbito en constante evolución y con múltiples líneas de trabajo. De hecho, podemos encontrar distintos términos y aproximaciones al respecto en Stream Analytics, Stream Processing, Complex Event Processing, Data Stream Management Systems, etc. En este subapartado nos centraremos en la aproximación más parecida a un enfoque de bases de datos: los sistemas de gestión de *streams* de datos (o *data stream management systems (DSMS)* en inglés). Los DSMS se definen como el conjunto de sistemas que se encargan de gestionar y analizar datos en *stream*.

Los sistemas DSMS serían el equivalente a un sistema gestor de bases de datos pero en un entorno donde los datos se proveen de forma continua en forma de *streams*. Estos sistemas permiten definir la estructura de los datos que se obtienen de un determinado *stream* (utilizando estructuras parecidas a las vistas del modelo relacional) y realizar consultas directamente sobre los *streams* de datos utilizando lenguajes de consulta de alto nivel (en muchos casos parecidos a SQL).

Las consultas que se realizan en estos sistemas no suelen ser puntuales, es decir, consultas que se hacen en un cierto instante del tiempo, sino continuas. Estas consultas se «instalan» en un *stream* de datos (o en un conjunto de ellos) y una vez «instaladas» se ejecutan de forma continua. La periodicidad de la ejecución la suele marcar el usuario en función de una ventana de tiempo (cada *x* segundos por ejemplo) o de datos (al llegar una cantidad determinada de registros o al suceder un evento, por ejemplo).

Las características principales de estos sistemas que los diferencian de los sistemas gestores de bases de datos son las siguientes:

- **Acceso secuencial**: los datos se acceden secuencialmente, a medida que van llegando. Normalmente las consultas permiten indicar qué datos queremos obtener y qué condiciones deben cumplir; no obstante, estos datos deberán estar definidos en el contexto de una ventana temporal concreta. Por tanto, una consulta en estos sistemas consultará **solo** datos recientes.

- **Consultas continuas**: las consultas se «instalan» en uno (o más) *streams* de datos y devuelven resultados cuando haya nuevos datos de interés o cuando haya pasado el intervalo de tiempo indicado por el usuario que instaló la consulta.

- **Orden de los datos**: en la mayoría de los sistemas vistos hasta ahora lo importante es conocer el estado actual de los datos (la dirección de una persona, su edad o su estado civil). No obstante, en estos sistemas la importancia no recae tanto en el estado final de los datos sino en los distintos estados por los que ha ido evolucionando hasta llegar al estado final.

- **Frecuencia de actualización**: a diferencia de los sistemas tradicionales, los sistemas en *streaming* proveen de información de forma continua. Por tanto, estos sistemas deberán proveer mecanismos para garantizar una rápida escritura de datos.

- **Tamaño de los datos**: el tamaño de los registros proveídos por el *stream* suele ser pequeño. Los sistemas deberán estar optimizados para procesar continuas actualizaciones o inserciones de datos de reducido tamaño.

- **Tiempo real**: en muchos casos estos sistemas requieren responder preguntas analíticas en tiempo real.

- **Espacio de almacenamiento**: debido a la potencial rapidez con que llegan los datos y con que tienen que analizarse, estos sistemas suelen realizar el procesamiento en memoria. Por tanto, hay que tener en cuenta las limitaciones de la misma.

- **Completitud de los datos**: normalmente, las bases de datos asumen que los datos son consistentes (exactos, actualizados, representan fielmente la realidad, etc). En estos sistemas no es así, ya que al llegar los datos de forma secuencial, asumimos que no tenemos todos los datos necesarios en cada momento y que estos pueden ser incompletos.

Existen distintos sistemas que podrían englobarse en esta categoría y que permiten consumir y analizar datos en *streaming* de forma eficiente. Algunos de ellos son VoltDB*, que utiliza una aproximación denominada *Fast Data* para proveer análisis simples en tiempo real a la vez que permite almacenar la información para realizar análisis más complejos de forma diferida; MapR-DB**, que permite representar tanto *streams* de datos como procesar sus datos de forma eficiente; PipelineDB,*** que extiende PostgreSQL;**** InfuxDB,***** que utiliza una aproximación más enfocada a bases de datos temporales, y SQLStream,****** que permite gestionar *streams* de datos mediante una extensión de SQL llamada StreamSQL.

*** https://www.voltdb.com/**
**** http://bit.ly/2CtxTop**
***** http://bit.ly/2DIvdQq**

****** http://bit.ly/2CxHBU9**
******* http://bit.ly/2lxmYAb**
******** http://sqlstream.com**

3.4.1. Ejemplos utilizando StreamSQL

A continuación, vamos a ver un par de ejemplos simples sobre StreamSQL para mostrar más claramente cómo funcionan los sistemas DSMS. Para ello utilizaremos un ejemplo basado en el envío de un *stream* de datos con información de un sensor de temperatura que envía la información siguiente:

```
(<25 grados, 70 % humedad>, 1483309623)
(<25 grados, 70 % humedad>, 1483309624)
(<26 grados, 60 % humedad>, 1483309625)
(<15 grados, 90 % humedad>, 1483309626)
(<10 grados, 95 % humedad>, 1483309627)
```

Supongamos para simplificar que dichos datos se proporcionan mediante las siguientes tripletas en un sólo *stream* de datos, tal y como se muestra a continuación:

```
(25 grados, 70 % humedad, 1483309623)
(25 grados, 70 % humedad, 1483309624)
(26 grados, 60 % humedad, 1483309625)
(15 grados, 90 % humedad, 1483309626)
(10 grados, 95 % humedad, 1483309627)
```

A continuación veremos cómo integrar este *stream* en StreamSQL y explotar sus datos.

StreamSQL utiliza el lenguaje de consulta StreamSQL, que es una extensión de SQL. Para poder procesar datos de un *stream* en StreamSQL debemos hacer tres cosas:

StreamSQL

Podéis encontrar más información sobre la sintaxis del lenguaje en http://bit.ly/2zXDrlA.

1) Definir el enlace entre el *stream* externo y StreamSQL,

2) definir la estructura del *stream*,

3) realizar las consultas que calculen la información de interés.

Para simplificar el ejemplo, en este apartado trabajaremos únicamente los puntos 2 y 3.

Para definir un *stream* de datos se utiliza la sentencia `CREATE STREAM`. Esta sentencia permite definir un *stream* indicando la estructura de sus datos. Como podemos ver a continuación, la sentencia para definir el *stream* de datos es muy simple y se asemeja a la que utilizaríamos para la creación de una tabla o una vista.* En la consulta se define un nuevo *stream* de datos llamado *TempAndHumData* que contiene los tres campos del *stream* de origen.

*** En las consultas del ejemplo se ha obviado el uso de esquemas de datos con el objetivo de simplificar su comprensión.**

```
CREATE OR REPLACE FOREIGN STREAM "TempAndHumData" (
"temperature"    SMALLINT NOT NULL,
"humidity"       SMALLINT NOT NULL,
"MesuredTime"    TIMESTAMP NOT NULL
) DESCRIPTION 'Datos de temperatura y humedad';
```

Una vez definido el *stream* y enlazado con el *stream* externo (no realizado en el ejemplo) se podrían realizar consultas al *stream* de datos. Estas consultas utilizan SQL con algunas funciones añadidas que permiten gestionar ventanas de validez que limiten los datos que se consultan (por tiempo, por número de registros o por ambos factores). A continuación, podemos ver tres consultas de ejemplo.

```
// Consulta 1: Seleccionar las medidas de temperaturas y
        humedad del último día
SELECT  STREAM temperature, humidity
FROM  TempAndHumData
RANGE INTERVAL '1' DAY PRECEDING;

// Consulta 2: Seleccionar las medidas de temperaturas y
        humedad de los cinco últimos registros
SELECT  STREAM temperature, humidity
FROM  TempAndHumData
RANGE  ROWS 5 PRECEDING;

// Consulta 3: Seleccionar las medias de temperatura y
        humedad del último día
SELECT  STREAM AVG(temperature), AVG(humidity)
FROM  TempAndHumData
RANGE INTERVAL '1' DAY PRECEDING;
```

La primera consulta devolvería los datos de temperatura y humedad facilitados en el último día. La ventana de tiempo se define mediante la cláusula `RANGE INTERVAL 1 DAY PRECEDING`, que indica que queremos los registros que se hayan generado dentro del día en curso (ahora - 24 horas). Para gestionar la ventana temporal, StreamSQL utiliza el valor del campo ROWTIME. Este campo (parecido al ROWID en las bases de datos relacionales) se almacena para cada registro de un *stream* y representa el momento del tiempo en que los datos se generaron (o en su defecto se insertaron en la base de datos). La segunda consulta crea una ventana de trabajo que tiene en cuenta solo los cinco últimos registros, por tanto devolvería las cinco últimas lecturas de humedad y temperatura. Finalmente, la consulta número tres devolvería la temperatura y humedad medias del último día.

Notad que todas las consultas se ejecutan sobre una ventana determinada, por tanto su ejecución devolverá resultados distintos a medida que lleguen nuevos datos por el *stream*.

Resumen

En este módulo didáctico hemos descrito técnicas avanzadas de minería de datos en entornos de los datos masivos. Concretamente, nos hemos centrado en tres análisis específicos, pero que han ganado muchísima popularidad recientemente: el análisis de grafos, la minería de textos y el análisis de datos en *streaming*.

En primer lugar, los grafos presentan una potente forma de representación de datos semiestructurados, pero la complejidad de realizar análisis sobre estructuras basadas en grafos es elevada. En este sentido, existen multitud de algoritmos desarrollados específicamente para trabajar con grafos, aunque desgraciadamente no todos ellos son paralelizables. En este módulo hemos presentado las bases del análisis de grafos, así como las implementaciones disponibles en los dos grandes entornos de datos masivos, como son Apache Hadoop y Apache Spark.

Por otro lado, la minería de textos también es extremadamente importante en el momento actual. Hemos revisado las principales disciplinas dentro de esta área, así como el procedimiento general para lidiar con datos masivos en formato no estructurado, como son los textos. Hemos presentado dos de las principales herramientas para lidiar con datos textuales en entornos de datos masivos, como son Apache Solr y Elasticsearch.

Finalmente, en la última parte de los materiales se han tratado los sistemas que permiten procesar, analizar y almacenar los datos en *streaming*. Hemos revisado las características y peculiaridades que confirman un escenario típico de procesamiento en *streaming*, aunque la diversidad de datos y escenarios hace inviable contemplar todo el abanico posible de opciones. A continuación, hemos profundizado en tres herramientas importantes, como son Apache Storm, Spark Streaming y Apache Flink. Para finalizar, se ha explicado el funcionamiento y las principales características de los sistemas de gestión de *stream* de datos, y se ha mostrado cómo definir y consultar un *stream* en este tipo de sistemas mediante un ejemplo en StreamSQL.

Glosario

conjunto de datos resiliente y distribuido *m* Unidad mínima de procesado para Spark. *en* Resilient-Distributed Dataset.

damping *m* Véase **factor de amortiguamiento**.

escalabilidad horizontal *f* Sistema que mejora su rendimiento al agregarse más nodos.

factor de amortiguamiento *m* Probabilidad de que un usuario vaya a dejar de «navegar» por las páginas referenciadas entre ellas. *en* damping

hilo *m* Proceso que se ejecuta en paralelo o de forma asíncrona a otros procesos que forman una aplicación. *en* thread

matriz de Laplace *f* Matriz cuadrada de orden $n \times n$ tal que $L = D - A$, donde D es la matriz diagonal que contiene la suma de filas a lo largo de la diagonal y 0 en las demás posiciones, y A es la matriz de adyacencia.

PageRank Algoritmo utilizado por Google para clasificar páginas web sobre la base de las referencias cruzadas entre ellas. La idea que subyace es que las páginas web más importantes tienen más referencias de otras páginas web.

Resilient-Distributed Dataset Véase **conjunto de datos resilientes y distribuido**.

StreamSQL Lenguaje de consulta que amplía SQL con el objetivo de procesar flujos de datos en tiempo real. Podéis encontrar más información sobre la sintaxis del lenguaje en http://sqlstream.com/docs/sqlrf_index.html.

thread Véase **hilo**.

Bibliografía

Betts, R.; Hugg, J. (2015). *Fast Data: Smart and at Scale Design Patterns and Recipes*. Boston: O'Reilly Media, Inc.

Fleischner, H. (2016). *Algorithms in Graph Theory*. Viena: Algorithms and Complexity Group.

Garofalakis, M.; Gehrke, J.; Rastogi, R. (2016). *Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications)*. Berlín: Editorial Springer.

Hearst, M. (2003). *Untangling Text Data Mining*. En: Actas del ACL'99: 37.º encuentro anual de la Asociación para la Lingüística Computacional.

Miner, G.; Elder, J.; Fast, A.; Hill, T.; Nisbet, R.; Delen, D. (2012). *Practical Text Mining and Statistical Analysis for Non-Structured Text Data Applications*. Boston: Elsevier.

Pérez-Solà, C.; Casas-Roma, J. (2016). *Análisis de datos de redes sociales*. Barcelona: Editorial UOC.

Witten, I. H. (2005). «Text mining». *Practical handbook of internet computing*. M.P. Singh (ed.). Boca Raton (Florida): Chapman & Hall/CRC Press (págs. 14(1)-14(22)).

White, T. (2015). *Hadoop: The Definitive Guide, 4th Edition*. Boston: O'Reilly Media.

Zaharia, M.; Chambers, B. (2017). *Spark: The Definitive Guide, Big Data Processing Made Simple*. Boston: O'Reilly Media.