UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

Departamento de Informática

Sistema escalable para la detección de necesidades en escenarios de catástrofe natural

Esteban Andrés Abarca Rubio

Profesor guía: Nicolás Hidalgo Castillo

Profesor co-guía: Erika Rosas Olivos

Trabajo de titulación presentado en conformidad a los requisitos para obtener el título de Ingeniero Civil en Informática

2016

RESUMEN

Twitter es una red social que cuenta con millones de usuarios en todo el mundo y, en Chile, alcanza cerca de los 1.700.000 accesos diariamente. Sus usos van desde ser un *microblog* personal hasta la entrega de información o comunicación entre pares. Es por ello que, en épocas de necesidad, como lo es el periodo inmediato luego de la ocurrencia de una catástrofe natural, las personas tienden a publicar sus experiencias dentro de éste servicio.

Teniendo en cuentra lo anterior es que se construyó un sistema, basado en el paradigma de procesamiento de *streams*, capaz de recoger información de manera automática desde *Twitter* — los denominados *tweets* — y procesarlos a fin de detectar si es que un *tweet* corresponde a uno en el que el usuario haga mención alguno de los tipos de necesidad que el sistema es capaz de detectar y, finalmente, hacer uso de la información implícita (contenido del *tweet* o metadatos), para presentar la necesidad como un punto en un mapa geográfico del país, de modo que la información obtenida pueda ser tomada por las autoridades correspondientes para que, de esta forma, facilite el proceso de toma de desiciones en cuanto al envío de ayuda a una determinada área dadas las necesidades expresadas por la población.

Para lograrlo se hizo uso de programación extrema en conjunto con la metodología KDD para construir una aplicación que haga uso de un clasificador de textos para la detección de necesidades, mediante los cuales se logra ubicar y clasificar correctamente datos obtenidos del terremoto de Concepción ocurrido el 27 de febrero del año 2010. Además se prueba la efectividad del operador detector de ubicación para suplir la carencia de datos de geolocalización en los metadatos de *Twitter*.

Este trabajo se enmarca en el proyecto FONDEF IDeA (Dos etapas) código ID15I10560 en el que participa un equipo de investigación de la universidad.

Palabras Claves: Programación extrema; KDD; Clasficador; *Twitter*; Detección de necesidades; Geolocalización; *Stream processing*; Clasificador de texto; Redes sociales; Herramienta de apoyo a desastres

ABSTRACT

Twitter is a social network that already has millions of users worldwide and, in Chile, reach about of 1.7 millions of accesses daily. Its uses range from being a personal microblog up to information delivery and comunication between peers. It's because of this that in emergencies, such as the inmediate period after a natural catastrophe, people tends to post their experiences on this service.

With this in mind is that is built a system, based on stream processing paradigm, that it's able to get, automatically, information from *Twitter* — as *tweets* — and process it to detect if a user's *tweet* mentions one of the needs that the system can handle and, finally, use the implicit information in the tweet (metadata) and render the need as a geographical position in country's map, thus authority can use the given information and ease the desition making process of sending help to affected areas with the information given by the population.

To achieve these statements previously exposed extreme programming has been used in conjunction with KDD in order to build a text's classifier to detect people's needs. With these two methods was possible to classify and place correctly the data obtained from Concepción's earthquake at February the 27, 2010. It was proved the effectiveness of the proposed location recognizer created to supply the lack of geolocalization data in Twitter's metadata.

This project is within the FONDEF IDeA (Two stages), code: ID15I10560, where a University's research team is working on.

Keywords: Extreme programming; KDD; Classifier; Twitter; Detect people's Need; Geolocalization; Stream processing; Text's classification; Social networks; Post-disaster support tool

Dedicado a...

AGRADECIMIENTOS

Agradezco a

TABLA DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE ALGORITMOS

CAPÍTULO 1. INTRODUCCIÓN

1.1 ANTECEDENTES Y MOTIVACIÓN

Los desastres naturales en Chile han sido frecuentes en los últimos años. Sólo por mencionar algunos de los más recientes: la erupción del volcán Chaitén (Mayo, 2008), el terremoto en Tocopilla (Noviembre, 2007), el terremoto en Concepción (2010), el incendio de las Torres del Paine (Diciembre, 2011), el incendio en Valparaíso (Abril, 2014), la erupción del volcán Villarrica (Marzo, 2015), los aluviones en el norte (Marzo, 2015), entre otros. Dependiendo de las características de la emergencia, surgen en la población diversos tipos de necesidades: alimentos, agua, luz eléctrica, refugio, rescate o comunicación. Muchas veces éstas pueden no ser detectadas por las autoridades; al menos, no de forma expedita, lo que resulta perjudicial para las personas que intentan sobrellevar, de la mejor manera posible, la crisis cuando se ve involucrada una necesidad básica, como la falta de agua, donde la vida de los afectados puede verse comprometida. El problema anteriormente descrito, no es sólo para las autoridades; Imran et al. (2014a), señalan que el comportamiento humano ante crisis como éstas, no es quedarse esperando por ayuda o huir en pánico, sino de intentar tomar decisiones rápidas en base a la información que conocen. Esto quiere decir que existe gente dispuesta a ayudar, aun siendo ellos mismos los afectados; pero no siempre disponen de la información necesaria para saber hacia dónde apuntar sus esfuerzos. Será útil, dado todo lo anterior, tener algún medio que concentre las necesidades que pueda tener una población dentro del país, para acudir en su auxilio posterior a la ocurrencia de una emergencia catastrófica, como las mencionadas anteriormente.

Los académicos del proyecto FONDEF IDeA de la Universidad de Santiago de Chile, están trabajando en el desarrollo de una plataforma de procesamiento de flujo de datos generados en contextos de desastres; se busca proveer la infraestructura necesaria para la generación de herramientas capaces de apoyar la toma de decisiones en escenarios de desastres. Esta plataforma hace uso de la información generada por los usuarios en redes sociales como fuente de datos. Para realizar la prueba de concepto de la plataforma se planea desarrollar tres aplicaciones base, orientadas a facilitar la coordinación de voluntarios, detección de necesidades y difusión de información de interés.

En particular, para este trabajo, se ataca el problema de la detección de necesidades de la población y servir de apoyo para la construcción de la plataforma de *streaming*, en relación a qué operadores se han de construir y cómo ha de estructurarse el sistema para operar sobre datos nacionales.

1.2 DESCRIPCIÓN DEL PROBLEMA

El problema que aborda esta memoria, es el hacer uso de la información generada por la población por medio de *Twitter* para que, en caso de alguna emergencia de carácter nacional, pueda prestarse apoyo en tiempo real a las autoridades encargadas de la toma de decisiones; por ejemplo, dándoles a conocer en qué lugar exactamente se requiere asistir a la población con un determinado tipo de ayuda, según la necesidad que se presente. ¿Cómo detectar y localizar en tiempo real las necesidades expresadas por la población en redes sociales basadas en texto de manera de entregar generar una nueva fuente de datos para mejorar la toma de decisiones?

1.3 SOLUCIÓN PROPUESTA

Se propone un sistema capaz de recoger y analizar de manera automática los eventos generados en la red social *Twitter* en tiempo real, de manera que, al ocurrir un escenario de desastre, determine si la publicación expresa una necesidad y, en caso de que así sea, determine su posición geográfica.

La solución propuesta consta de dos partes: Por una lado la plataforma y lógica de procesamiento capaz de realizar la labor antes mencionada y, por otro, la visualización de estos datos.

Dado que se utiliza *Twitter* para obtener los datos de entrada, los cuales llegan mediante el *stream* producido por la API de esta red social, la plataforma de procesamiento consiste en un sistema de procesamiento de *streams*, construido utilizando el sistema de computación distribuida de Apache, *Storm*, el cual basa su procesamiento en transformar el problema en un grafo donde cada nodo aporta realizando sólo una tarea, pero que pueden existir múltiples instancias de aquel nodo y éstos pueden estar en diferentes máquinas. Así, internamente, la plataforma está compuesta de operadores que discriminan cuándo evento debe ser entregado a la aplicación de visualización para ser mostradas al usuario. Uno de estos operadores permitirá realizar la categorización del texto de entrada por medio un un clasificador bayesiano.

1.4 OBJETIVOS Y ALCANCE DEL PROYECTO

1.4.1 Objetivo general

Construir un sistema escalable para la detección de necesidades de la población en tiempo real, para escenarios de desastre natural haciendo uso de *Twitter*.

1.4.2 Objetivos específicos

1. Implementar un método encargado de la recolección de tweets generados dentro del territorio nacional haciendo uso de la API pública de Twitter.
2. Especificar la taxonomía de las necesidades detectadas.
3. Diseñar e implementar el clasificador de necesidades.
4. Definir los elementos de procesamiento para la construcción del sistema capaz de trabajar los datos obtenidos a gran escala.
5. Implementar una arquitectura escalable que soporte la aplicación.
6. Evaluar la aplicación bajo condiciones de alto tráfico, como es el caso de una emergencia nacional.

1.4.3 Alcances

Se utilizan las publicaciones de *Twitter* para llevar a cabo el procesamiento de la información y no se considera, en el marco de este trabajo, el uso de una red social alternativa, no porque no sea posible, sino que con el motivo de acotar el problema.

Las necesidades que la aplicación detecta no son una lista exhaustiva de las posibles existentes, sino de un subconjunto que se ha considerado más importante en el equipo de trabajo del proyecto. De esta forma se logra acotar el problema reduciendo la cantidad de categorías y permitir una mayor precisión en la clasificación, entendiendo la precisión como la relación de elementos clasificados correctamente sobre el total.

Se considera, para la construcción del clasificador, un subconjunto de un *dataset* compuesto de cuatro millones de *tweets* recogidos durante y posteriormente al terremoto en

Concepción el 27 de Febrero del 2010. Éste conjunto de datos ha de ser limpiado previamente pues contiene *tweets* en idiomas diferentes al idioma objetivo de este trabajo.

El sistema sólo trabaja en la detección con *tweets* que estén en español.

La validación se realiza a partir de *datasets* con *tweets* reales, sin embargo los flujos generados son sintéticos y no obtenidos de manera online desde *Twitter*.

1.5 METODOLOGÍAS Y HERRAMIENTAS UTILIZADAS

1.5.1 Metodología

Para la realización de este trabajo se utilizan dos metodologías, la primera está enfocada a la búsqueda de información en bases de datos para realizar la construcción del clasificador de texto, mientras que la segunda, una metodología de desarrollo de aplicaciones ágil está enfocada en la construcción en sí de las aplicaciones, tanto de la plataforma de procesamiento como de la aplicación de visualización. Ambas son definidas a continuación.

Programación Extrema

La Programación Extrema (*Extreme Programming*, XP desde ahora en adelante), comenzó como un proyecto el 6 de Marzo de 1996. Es uno de los procesos ágiles más populares y ha sido probado exitosamente en compañias e industrias de todos los tamaños. Wells (2013).

Su éxito se debe a que hace especial hincapié en la satisfacción del cliente por sobre la entrega de todo lo el software posible.

Aporta cinco formas escenciales para mejorar el proceso de desarrollo de software: comunicación, simplicidad, retroalimentación, respeto y coraje. Se busca en establecer una estrecha comunicación entre el equipo de desarrollo y el cliente apuntanto, paralelamente, a evitar el sobre-diseño y obtener retroalimentación de modo que los cambios puedan realizarse lo antes posible.

La metodología implementa unas simples reglas de trabajo, las que se dividen en cinco grandes áreas las que se detallarán a continuación.

1. Planeación:
    - Se escriben Historias de usuario.
    - Se crea un plan de *releases*.
    - Se planifican liberaciones pequeñas y frecuentes.

- Se divide el proyecto en iteraciones.
- Al comienzo de cada iteración se planea cómo será.

2. Manejo:
    - Se le da al equipo una área de trabajo.
    - Se realizan reunione del tipo *stand up meeting* a díario.
    - Se mide la velocidad del proyecto.
    - Se mueven a las personas de sus puestos (para que todo el equipo pueda trabajar en todo).
    - Se solucionan problemas que instroduzcan quiebres en la metodología.
3. Diseño:
    - Simplicidad. El mejor diseño es el más simple.
    - Se crean *spikes* para reducir el riesgo.
    - No se agregan funcionalidades antes de tiempo.
    - Hacer uso de técnicas de *refactoring*, cada vez que sea posible.
4. Implementación:
    - El cliente siempre está disponible.
    - El código debe ser escrito bajo estándares.
    - Se hace uso de *Test Driven Development* (TDD).
    - Todo el código debe hacerse haciendo uso de *pair programming*.
    - Sólo una pareja integra código a la vez.
    - Integración a menudo.
    - Se cuenta con un equipo dedicado a la integración.
    - El código es de todos.
5. Prueba:
    - Todo el código debe tener pruebas unitarias.
    - Todas las pruebas deben ser pasadas antes de una liberación.
    - Cuando se encuentra un *bug*, se crean pruebas.
    - Los *test* de aceptación se corren a menudo y sus resultados son publicados.

Éstas reglas por sí solas pueden carecer de sentido, pero se apoyan en los **valores** que la metodología quiere entregar y que fueron mencionadas anteriormente, pero ahora son detalladas:

- Simplicidad: Se hace lo que se solicitó, pero no más. Ésto maximiza el valor entregado dado una fecha límite. Nuestras metas se alcanzan por medio de pequeños pasos para mitigar errores tan pronto ocurran. Se crea algo de lo que se esté orgullosos y lo se mantiene en el tiempo a costos razonables.
- Comunicación: Todos somos partes de un equipo y nos comunicamos cara a cara a diario. Se trabaja juntos en todo: desde la toma de requerimientos hasta la implementación. Se crea la mejor solución posible al problema.
- Retroalimentación: Cada iteración es completada seriamente entregando *software* funcional. Mostraremos nuestro *software* a menudo y prontamente para luego escuchar y aplicar los cambios solicitados. Se habla de nuestro proyecto y se adapta nuestro proceso a el, no al revéz.
- Respeto: Todos dan y reciben el respeto que merecen como miembros del equipo. Todos contribuyen con valor así sea simple entusiasmo. Los desarrolladores respetan la experiencia del cliente y viceversa.
- Coraje: Se dice la verdad sobre el progreso y nuestras estimaciones. No se documentan excusas por si se falla, pues se planea tener éxito. No tenemos porque no trabajamos solos. Nos adaptaremos a los cambio cuando ocurran.

El proceso de XP puede puede ser apreciado en la Figura 1.1.

Figura 1.1: Diagrama de flujo de Programación Extrema.
Fuente: Wells (2013)

Knowledge Discovery in Databases (KDD)

Metodología de trabajo para la búsqueda de información en bases de datos, es definida por Fayyad & Uthurusamy (1995) como ”el proceso no trivial de identificar patrones válidos, nuevos, potencialmente útiles y en ultima instancia comprensible en los datos”, surge de la necesidad de manejar grandes volúmenes de datos e involucra simultaneamente varias disciplinas de investigación tales como el aprendizaje automático, la estadística, inteligencia artificial, sistemas de gestión de bases de datos, sistemas de apoyo a la toma de decisiones, entre otras.

Si bien puede variar el usuario, quien es aquel que determina el domino de la aplicación, es decir, cómo se utilizan los datos, el proceso generalmente considera las siguientes etapas:

1. Selección de datos: Consiste en buscar el objetivo y las herramientas del proceso de minería, identificando los datos que han de ser extraídos, buscando atributos apropiados de entrada y la información de salida para representar la tarea. Esto quiere decir, primero se debe tener en cuenta lo que se sabe, lo que se quiere obtener y cuáles son los datos que nos facilitarán esa información para poder llegar a nuestra meta, antes de comenzar el proceso como tal.

2. Limpieza de datos: En este paso se limpian los atributos sucios, incluyendo datos incompletos, el ruido y datos inconsistentes. Estos datos sucios, en algunos casos, deben ser eliminados, pues pueden contribuir a un análisis inexacto y resultados incorrectos.

3. Integración de datos: Combina datos de múltiples procedencias incluyendo múltiples bases de datos, que pueden tener diferentes contenidos y formatos.

4. Transformación de datos: Consiste en modificaciones sintácticas llevadas a cabo sobre los datos sin que suponga un cambio en la técnica de minería aplicada. Tiene dos caras, por un lado existen ventajas en el sentido de mejorar la interpretación de las reglas descubiertas y reduce el tiempo de ejecución, por el otro puede llevar a la pérdida de información.

5. Reducción de datos: Reducción del tamaño de los datos, encontrando características más significativas dependiendo del objetivo del proceso.

6. Minería de datos: Consiste en la búsqueda de patrones de interés que puedan expresarse como un modelo o dependencia de los datos. Se ha de de especificar un criterio de preferencia para seleccionar un modelo de un conjunto de posibles modelos. Además se ha de especificar la estrategia de búsqueda (algoritmo), a utilizar.

7. Evaluación de los patrones: Se identifican patrones interesantes que representan conocimiento utilizando diferentes técnicas incluyendo análisis estadísticos y lenguajes de

consulta.

8. Interpretación de resultados: Consiste en entender resultados de análisis y sus implicaciones y puede llevar a regresar a algunos pasos anteriores.

La representacíón del proceso descrito por la metodología KDD es presentada en la Figura 1.2.

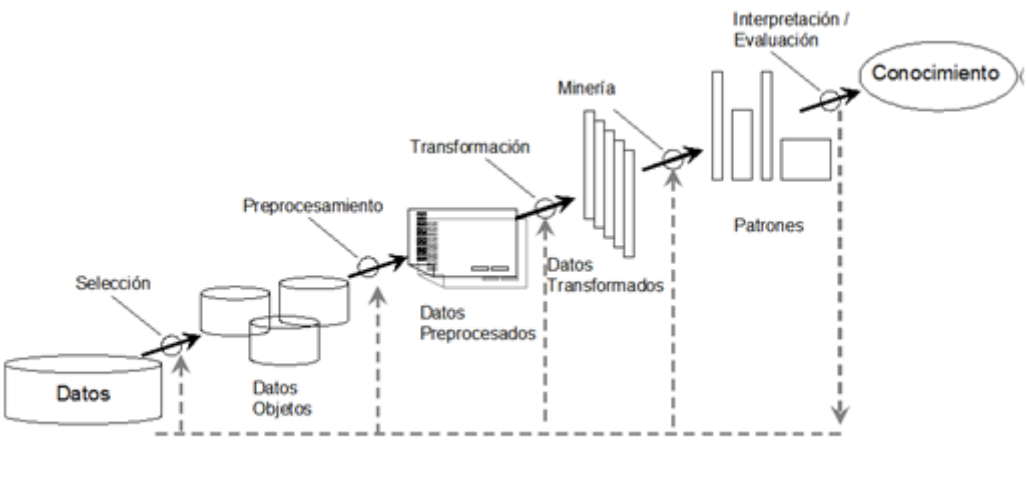

Figura 1.2: Proceso para el manejo y tratamiento de datos según la metodología KDD. Fuente: Iribarra (2015)

1.5.2 Herramientas de desarrollo

A continuación se presentan las herramientas, tanto de *software* como de *hardware* utilizadas para la contrucción del sistema de detección de necesidades.

Se ha se utilizado diversas herramientas de software para la construcción de la aplicación, éstas son descritas a continuación haciendo especial énfasis en aquellas de gran importancia dentro del desarrollo del proyecto

- Apache Storm: sistema de procesamiento distribuido que basa su procesamiento en dividir el procesamiento en nodos de un grafo dirigido. Permite el procesamiento de eventos en tiempo real.
- MongoDB: sistema de gestión de bases de datos no-SQL capaz de lidiar con altas tasas de tráfico y respuestas rápidas para aplicaciones en tiempo real.
- Mallet: biblioteca de Java que contiene herramientas para el procesamiento de lenguaje natural, clasificación de documentos, extracción de información y otras herramientas de

aprendizaje automático sobre texto.

- Play Framework: *framework* para la contrucción tanto de aplicaciones Java como Scala, utiliza el modelo de arquitectura de diseño modelo-vista-controlador. Está orientado a la construcción de aplicaciones REST y hace hincapié en la productividad de los desarrolladores.

Además de las herramientas descritas se hace uso de las una lista de herramientas comunes para el desarrollo de proyectos de *software* presentada a continuación:

- NetBeans (8.1), como herramienta de apoyo a la construcción de la aplicación.
- Sublime Text 3 (Build 3103), como editor de textos.
- MiKTex (XeLaTeX), para la escritura de la memoria.
- PowerDesigner 16, para la elaboración de diagramas.
- Bitbucket (Git), como repositorio de todo lo referente al proyecto (detector de necesidades, visualizador y documento de memoria).
- Windows 10 Home Edition (x64).
- YourKit 1.8.0_92 64 bits, para evaluación de *performance* de la aplicación.
- Linux Mint 17.3 (x86).
- Oracle VirtualBox (5.0.14).

Herramientas de hardware

Se hace uso del equipo del autor de este trabajo cuyas características técnicas son descritas a contunuación:

- Procesador Intel Core i5 2.2 Ghz.
- 8 GB de memoria RAM.
- 1 TB de disco duro.

1.6 ORGANIZACIÓN DEL DOCUMENTO

El resto del documento se organiza de la siguiente manera:

El Capítulo 2, presenta el marco teorico que sustenta la solución implementada y un análisis del estado del arte en términos de las herramientas y tecnicas que han sido utilizadas para dar solucion a problemas similares.

El Capítulo 3 describe el proceso de toma de requerimientos de la aplicación. Para ello, siguiendo la metodología XP, se usan de historias de usuario y sus correspondientes criterios de aceptación.

El Capítulo 4, se presenta la arquitectura del sistema y las decisiones que llevaron a que se se optara por ésta además de describir la implementación de las aplicaciones visualizador y detector de necesidades, incluyendo los elementos que las componen y, en el caso de esta última aplicación, el porqué del uso de una topología en particular.

El Capítulo 5, describe la completitud de las historias de usuario, evalúa el nivel de replicación de los operadores del sistema y su rendimiento en una simulación de una situación real como fue el terremoto de Concepción en febrero del año 2010.

Finalmente en el Capítulo 6, presenta las conclusiones del trabajo realizado, el cumplimiento de objetivos tanto general como específicos, los resultados de los experimentos y trabajo futuro.

CAPÍTULO 2. MARCO TEÓRICO Y ESTADO DEL ARTE

Este capítulo busca realizar una contextualización de los conceptos necesarios para entender el problema que se está tratando por medio de una breve reseña de cada uno, además de a conocer la revisión de los trabajos existentes capaces de realizar tareas similares o basadas en principios similares a los planteados por la solución expuesta de éste trabajo.

2.1 MARCO TEÓRICO

Se presentan las definiciones de herramientas y técnicas presentados y que sustentan a la solución implementada.

2.1.1 Sistemas de procesamiento de *streams*

El sistema tradicional de procesamiento, el trabajo por lotes (*batch*), está pensado en que primero los datos son recopilados y almacenados, una vez que ésto se ha hecho pasan a ser procesados, pero ¿qué pasa si este enfoque no sirve y los datos han de ser procesados conforme éstos van llegando?

Para realizar esto es que existe otro enfoque de procesamiento, el procesamiento de *streams*, este enfoque no quita que se puedan almacenar los datos para un análisis *a posteriori*, pero está pensado para trabajar con datos que llegan al sistema directamente desde la fuente y en gran cantidad y son procesados uno por uno.

Un sistema de procesamiento de *streams* se caracteriza, principalmente, por lo siguiente:

- En memoria: procesamiento contínuo en flujos de datos en series de tiempo.
- Escalable: arquitectura optimizado para latencia cercana a cero en grandes volúmenes de datos.
- Escalabilidad a través de la distribución eficiente en múltiples procesadores o servidores.

Su funcionamiento es caracterizado por información llegando y es asignada a un nodo, donde sufre alguna transformación al pasar por ellos.

Conceptualmente su arquitectura es la de una red de elementos ejecutándose de forma concurrente y que operan el flujo de información que va llegando. La Figura 2.1 presenta cómo es una típica arquitectura de éstos sistemas, SQLStream (2015).

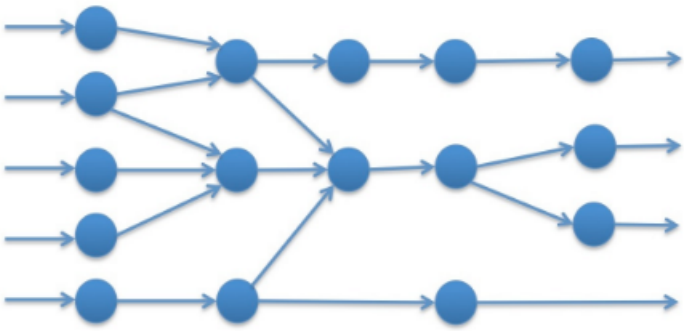

Figura 2.1: Arquitectura típica de sistemas de procesamiento de *streams*.
Fuente: SQLStream (2015)

Uno de los sistemas de procesamiento de *stream* que ha cobrado relevancia en el último tiempo es *Apache Storm*.

Apache Storm

Apache Storm es un sistema que sirve para recuperar *streams* de datos en tiempo real desde una o múltiples fuentes de manera distribuida, tolerante a fallos y en alta disponibilidad. *Storm* está principalmente pensado para trabajar con datos que deben ser analizados en tiempo real, Ramos (2015).

Es escalable, tolerante a fallos y garantiza que toda la información será procesada. Presenta *Benchmarks* que señalan que por nodo es capaz de procesar más de un millón de tuplas por segundo.

Un sistema construido haciendo uso de *storm* está compuesto por elementos procesadores de dos tipos. El primero es denominada *Spout* y es el encargada de recoger el flujo de datos de entrada. El segundo es denominada *Bolt* y es el encargada de la transformación o procesado de los datos.

Oficialmente, una aplicación *storm* es representada como puede verse en la Figura 2.2, allí los *Spouts* son representados simulando ser llaves de agua desde donde fluyen los datos al sistema y los *Bolts* como rayos donde se procesa el flujo.

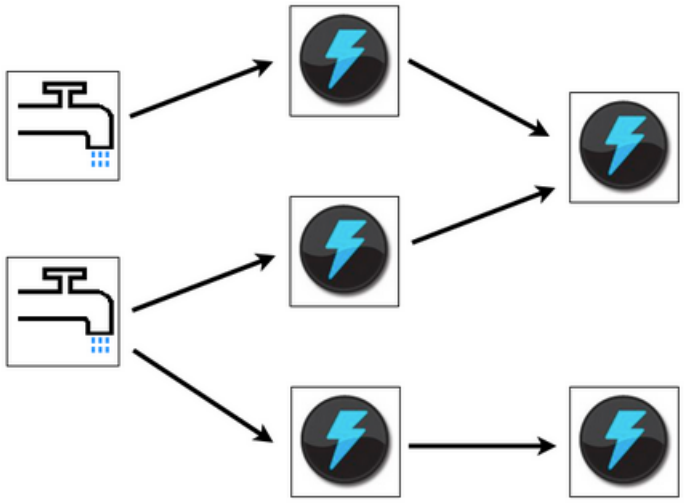

Figura 2.2: Representación del funcionamiento de Apache Storm.
Fuente: Foundation (2016)

Uno de los puntos fuertes que tiene este sistema, y que está en línea con lo señalado para sistemas de procesamiento de *stream*, es que al crear una topología donde se instancian *Bolts* y *Spouts*, Storm se encarga de escalar el sistema distribuyendo los elementos en sus componentes.

Spout y *bolts* se unen para dar origen a una topología. Una topología de Storm es similar a un grafo acíclico dirigido, donde cada nodo se encarga de procesar una determinada información y le pasa el testigo al siguiente nodo. Está compuesta por *Spouts* y *Bolts*, Ramos (2015).

Se refiere a la forma en la que se van a compartir los datos entre los componentes. Como modelo de datos, Storm utiliza tuplas que son listas de valores con un nombre específico, Ramos (2015).

- Shuffle grouping: *storm* decide de forma *round robin* la tarea a la que se va a enviar la tupla, de manera que la distribución sea equivalente entre todos los nodos.
- Fields grouping: se agrupan los *streams* por un determinado campo de manera que se distribuyen los valores que cumplen una determinada condición a la misma tarea.
- All grouping: el *stream* pasa por todas las tareas haciendo multicast.
- Grobal grouping: el *stream* se envía al *bolt* con ID más bajo.
- None grouping: es un *Shuffle grouping* donde el orden no es importante.
- Direct grouping: la tarea es la encargada de decidir hacia donde emitir especificando el ID del destinatario.

- Local grouping: se utiliza el mismo *bolt* si tiene una o más tareas en el mismo proceso.

Storm puede funcionar de dos modos: local y *cluster*. El primero es útil para el desarrollo, pues ejecuta toda la topología en una única JVM, por lo que pueden realizarse fácilmente pruebas de integración, depurar código, etcétera. Este modo simula, haciendo uso de *threads*, cada nodo del Cluster, Ramos (2015).

El modo Cluster es considerado el ẗmodo de producciónÿ es el modo donde el código es distribuido en máquinas diferentes dentro del Cluster.

La arquitectura de Storm se divide en tres componentes:

- Master Node: ejecuta el demonio llamado Nimbus, el cual es responsable de distribuir el código a través del cluster. Realiza la asignación y monitorización de tareas en las distintas máquinas del cluster.
- Worker Node: ejecutan el demonio Supervisor, el cual se encarga de recoger y procesar los trabajos asignados en la máquina donde está siendo ejecutado. En caso de fallo de uno *Worker Node*, Nimbus observa esto y redirige el trabajo a otro.
- Zookeeper: si bien no es un componente propio de Storm, es necesario para su funcionamiento, pues se encarga de coordinar Nimbus y Supervisor, además de mantener sus estados, pues ambos son *stateless*.

2.1.2 Minería de texto

La minería de textos o *text mining* es una rama de la lingüística computacional que trata de obtener información y conocimiento a partir de conjuntos de datos que en principio no tienen un orden o no están dispuestos en origen para transmitir esa informacion, Fernández (2011).

Para introducir este concepto es necesario haber pasado primero por entender el concepto de minería de datos o *data mining*.

Minería de datos

Para la minería de datos los datos son la materia prima, ésta se convierte en información que posteriormente es tratada y utilizada para convertirla en conocimiento. Reune áreas como la estadística, inteligencia artificial, bases de datos (la materia prima), y procesamiento masivo. Molina (2002) la define como Ïla integración de un conjunto de áreas que tienen como propósito la identificación de un conocimiento obtenido a partir de las bases de datos que aporten un sesgo hacia la toma de decisión.¨

La principal diferencia de la minería de datos con respecto a la minería de textos es que la primera utiliza bases de datos como materia prima, mietras la segunda hace uso de documentos para ello.

Una de las aplicaciones de la minería de textos es la clasificación de documentos. Esta práctica consiste en el uso de técnicas de aprendizaje automático para asignar categorías a un determinado texto, para ello existen algoritmos de clasificación con los cuales pueden construirse clasificadores.

Clasificadores

En el campo del procesamiento de lenguaje natural el detectar patrones es una parte central. La clasificación es la tarea de seleccionar la etiqueta correcta para una entrada dada. Cada entrada es considerada como aislada de las demás, y el set de etiquetas (o categorías), es definido con anterioridad.

Un clasificador es llamado šupervisadoši ha sido construido a partir de un conjunto de datos de entrenamiento o *corpus* de entrenamiento. La Figura 2.3 muestra el proceso de entrenamiento y uso de un clasificador. (a) Durante el entrenamiento, un estractor de características es utilizado para convertir cada entrada a un conjunto de característica, éstos capturan la información básica de cada conjunto de entrada que es usada para clasificar. Pares de éstos conjuntos y etiquetas son la entrada para el algoritmo que contruye el modelo de clasificación. (b) Durante la predicción se realiza el mismo procedimiento, pero sin entregar como entrada la etiqueta correspondiente, se pasa al modelo generado para obtener las etiquetas predictas.

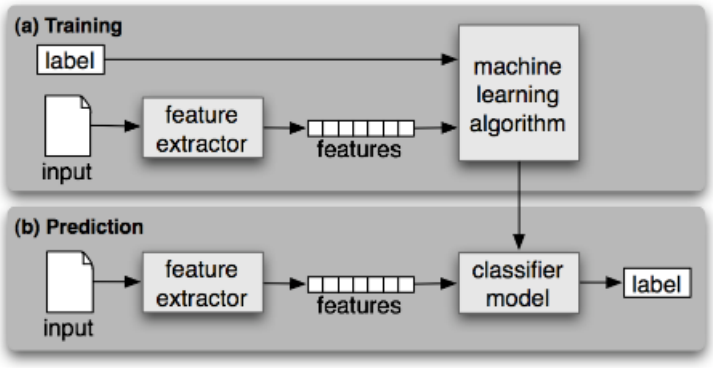

Figura 2.3: Clasificación supervisada.
Fuente: Bird et al. (2009)

Para decidir si el modelo de clasificación construido es preciso, éste debe evaluarse. Para ello se ha de tener un conjunto de datos de evaluación. Típicamente corresponde a la misma entrada, donde se evalúa el resultado obtenido por el clasificador con el definido en ésta,

separando el conjunto de entrenamiento en dos categorías: los datos de entrenamiento y los datos de evaluación. Comúnmente, se utiliza un 10% del conjunto total para formar los datos de evaluación.

La métrica más simple para evaluar un clasificador es la precisión (*accuracy*), ésta corresponde al ratio de entradas clasificadas correctamente. Matemáticamente:

$$Accuracy = \frac{TP}{TP + FP}$$

TP y FP corresponden respectivamente a valores correctamente clasificados (*true positives*) y valores clasificados incorrectamente (*false positives*), también conocidos como Ërrores tipo I.¨ Adicionalmente a éstos existen otros dos: TN y FN que, respectivamente corresponden a valores correctamente no clasificados como elementos de una etiqueta específica (*true negatives*) y los conocidos como Ërrores tipo II,¨ valores que no pertenecen a una etiqueta y han sido clasificados como tales. Con estos valores se pueden obtener dos nuevas métricas: El *recall* y el medida F (*F-Score*).

El *recall*, indica el ratio de elementos clasificados correctamente y matemáticamente se define como sigue:

$$Recall = \frac{TP}{TP + FN}$$

Por otro lado la Medida F, combina precisión y *recall* para entregar un puntaje. Está dado, en general, dado por la siguiente función:

$$F_\beta = (1 + \beta^2)\frac{Accuracy \cdot Recall}{(\beta^2 \cdot Accuracy) + Recall}$$

Típicamente se utiliza la medida armónica donde β toma el valor 1, quedando la función como sigue:

$$F1 = 2\frac{Accuracy \cdot Recall}{Accuracy + Recall}$$

Se dice que es la medida armónica, pues pondera de igual manera ambos ratios.

Existen diferentes tipos de métodos de aprendizaje de máquina utilizados para construir clasificadores, dentro de ellos existen: *Naïve Bayes*, máxima entropía y árboles de decisión.

En cuanto a los árboles de decisión, éstos corresponden a un diagrama de flujo con el que se decide la etiqueta para una entrada. Los diagramas de flujo consisten en nodos de decisión, los que comprueban los conjuntos de características y las hojas que asignan las etiquetas. El más simple de ellos es llamado un *decision stump*, el cual consiste en sólo un nodo de decisión

y múltiples hojas, para construirlo se construye para todas las posibles etiquetas y se calcula la precisión para quedarse con aquellos que alcancen la más alta. Así es como se generaliza, construyendo múltiples *decision stumps* y usando aquelos con más alta probabilidad.

Para el caso de los clasificadores *Naïve Bayes*, Russell (2003), cada elemento del vector de características aporta en la determinación de qué etiqueta ha de ser utilizada. Para cada etiqueta se calcula su probabilidad *a priori*, utilizando como referencia la frecuencia de ésta en el conjunto de entrenamiento. Esta probabilidad en conjunto con el aporte de cada elemento del vector determina una probabilidad de pertenencia, se le asigna a la entrada la etiqueta que tenga mayor probabilidad de pertenencia.

Para el caso de los clasificadores mñaxima entropía, son similares a los clasificadores bayesianos, pero en lugar de utilizar las probabilidades establecer los parámetros del modelo, utiliza técnicas de búsqueda para encontrar parámetros que maximicen el desempeño del clasificador mediante técnicas de optimización iterativa, las que inician los parámetros en valores aleatoreos y los refina mientras se va acercando al óptimo. Su principal desventaja es que toman mucho tiempo en realizar el aprendizaje por el periodo de iteración antes mencionado, Bird et al. (2009).

MongoDB

Base de datos no relacional (NoSQL) de código abierto escrita en C++ y está orientada al trabajo en documentos. Lo anterior quiere decir que, en lugar de guardar los datos en registros, lo hace en documentos y éstos son almacenada en una representación binaria de JSON conocida como BSON.

Una de las diferencias fundamentales con respecto a las bases de datos relacionales es que no es necesario que se siga un esquema; en una misma colección - concepto similar a una tabla en las bases de datos relacionales - pueden tener distintos esquemas.

MongoDB fue creado para brindar escalabilidad, rendimiento y disponibilidad. Puede ser utilizado en un servidor único como en múltiples. Esto se logra dado que MongoDB brinda un elevado rendimiento, tanto para lectura como para escritura, potenciando la computación en memoria.

Las consultas en MongoDB se realizan como si se tratase de Javascript entregando como parámetro un objeto JSON. Por ejemplo, dado el documento presentado en la Figura 2.4, parte de una colección llamada Ṗersonas”¨ en MongoDB:

Figura 2.4: Ejemplo de representación de un documento en MongoDB.
Fuente: Elaboración Propia, (2016)

Una consulta para encontrar este elemento dentro de la colección se da de la forma aprecida en la Figura 2.5

```
db.Personas.find({Nombre:"Juan"});
```

Figura 2.5: Ejemplo de consulta en MongoDB.
Fuente: Elaboración Propia, (2016)

En pruebas realizando operaciones habituales dentro de las bases de datos, Macool (2013), demostró que el tiempo de ejecución de MongoDB, como base de datos NoSQL, aventaja significativamente a las bases de datos relacionales más populares como lo son MySQL y PostgreSQL.

Play Framework

Es un *framework* de código abierto para aplicaciones *web* escrito en *Java* y *Scala*, el cual sigue el patrón de arquitectura *Modelo-Vista-Controlador* (MVC). Utiliza el paradigma de diseño C̈onvención sobre configuración¨ el cual apunta a reducir la toma de decisiones que debe tomar el desarrollador sin perder flexibilidad.

Se enfoca en la productividad, simplificando el desarrollo da aplicaciones mediante procesos automáticos de construcción y despliegue al elimina la desventaja de compilar, empaquetar e implementar el código cuando se realizan cambios, procesos requeridos al utilizar IDEs tradicionales. Los errores son mostrados inmediatamente por medio del navegador. En *Play*

por defecto soporta aplicaciones REST y no guarda el estado, permitiendo escalar mediante el uso de múltiples instancias simultaneas, Play (2015).

Mallet

Fué desarrollada por McCallum (2002), en la Universidad de Massachusetts Amherst, es una herramienta de Java para el procesamiento de lenguaje natural, clasificación de documentos, *clustering*, extracción de información y otras aplicaciones de aprendizaje de máquina sobre texto.

Cuenta con implementaciones de una variedad de algoritmos entre los cuales se encuentran: *Naïve Bayes*, máxima entropía y árboles de decisión. Además incluye herramientas para evaluar el desempeño de clasificadores mediante el uso de métricas más utilizadas.

2.2 ESTADO DEL ARTE

El problema a abordar consta de dos tópicos centrales: el procesamiento escalable de eventos en tiempo real y clasificación de eventos. Se comienza señalando desde donde inicia este trabajo, seguido de las impresiones de distintos autores respecto al trabajo en redes sociales (*Twitter* específicamente), continuando con sistemas de procesamiento para flujos de información para concluir con la construcción de clasificadores para etiquetado de datos.

2.2.1 Plataformas de procesamiento para desastres

El *Qatar Computing Research Institute*, es líder a nivel mundial en la creación de herramientas para dar soporte a desastres naturales usando tecnologías de la información, colaborando con grandes organizaciones como Naciones Unidas, Cruz Roja Internacional y UNICEF. Recientemente, han elaborado distintas aplicaciones (*MicroMappers*, *AIDR*, *UAViators*, etc.), para ayudar a darle sentido al gran flujo de información que reciben los centros de ayuda humanitaria (*Big crisis data*), generada por redes sociales, SMS, imágenes satelitales y aéreas tomadas por drones, que de otra forma serían incapaces de analizar. Tomando la información, estas aplicaciones unen la ayuda de voluntarios y profesionales digitales, quienes están dispuestos a colaborar a través de internet realizando tareas como identificación o

clasificación. Con ésta y el poder del aprendizaje de máquina, se entrenan algoritmos para analizar millones de datos.

Micromappers es una aplicación, creada por *Qatar Computing Research Institute*, que permite a voluntarios digitales etiquetar diferente tipos de información para la ayuda en el proceso de respuesta a desastres y desplegarlos en un mapa. En el caso del terremoto de Nepal por ejemplo, los voluntarios han contribuido revisando miles de *tweets* e imágenes para entregar una evaluación de impacto que ayuda a la toma de decisiones. La evolución de esta herramienta está orientada a integrarse con *AIDR* para que la información obtenida de las personas a través de *crowdsourcing* pueda servir para la automatización de proceso de clasificación mediante aprendizaje de máquina, y así subir al análisis de miles a millones de datos generados situaciones de emergencia.

AIDR (*Artificial Intelligence for Disaster Response*), Imran et al. (2014b), delega al aprendizaje de máquina la identificación de contenido durante desastres en *Twitter*. Va más allá de un simple filtro por palabras clave que limitan la búsqueda a ellas y al lenguaje. Está compuesto de tres partes: El colector, el entrenador y el etiquetador. El primero recolecta y almacena los datos, el entrenador construye un etiquetador automático y permite a usuarios realizar esto dado un conjunto de *tweets* capturados por el colector, el etiquetador analiza los *tweets* clasificados por humanos para automáticamente etiquetar nuevos *tweets*.

UAViators, iniciativa también creada por *Qatar Computing Research Institute*, reúne información generada por drones que entregan imágenes para crear conocimiento sobre el área del desastre en tiempo real. Su procesamiento puede entregar información como una estimación de la población afectada, daños en infraestructura de edificios, líneas de electricidad, carreteras, campamentos base, entre otro (http://uaviators.org/).

2.2.2 Procesamiento de datos a gran escala

Se presentan a continuación los sistemas de procesamiento de datos a gran escala más utilizados en la actualidad, Nicolás Hidalgo Castillo (2014).

MapReduce

El *framework* de Google *MapReduce*, Dean & Ghemawat (2008), es un modelo de programación diseñado para procesar conjuntos de datos a gran escala en una modalidad orientada al lote (*batch*) o procesamiento *online*. Este sistema esta orientado a programación funcional y utiliza funciones del tipo *map* y *reduce*. *MapReduce* ha logrado gran popularidad en aplicaciones orientadas a grandes volúmenes de datos dado su modelo de programación basado

en clave/valor y su escalabilidad. Las soluciones presentadas en y, Verma et al. (2013), están orientadas al procesamiento de flujos de datos en línea basadas en *MapReduce*. Los autores en Condie et al. (2010), proponen modificaciones a *Hadoop*, que es una implementación de *MapReduce* de código abierto, y que permite que los datos sean canalizados entre los operadores haciéndolo mas adecuado a aplicaciones con requerimientos de tiempo real.

Apache S4

S4, Neumeyer et al. (2010), o *Simple Scalable Streaming System*, es un sistema de propósito general, distribuido y escalable que permite que aplicaciones puedan procesar flujos de datos de forma continua y sin restricciones. S4 está inspirado en *MapReduce*, y fue diseñado en el contexto de minería de datos y algoritmos de aprendizaje de máquina en *Yahoo! Labs* para sistemas de publicidad *online*. Cada evento en S4 es descrito como un par (clave, atributo). La unidad básica son los elementos de procesamiento (PEs) y los mensajes que son intercambiados entre ellos. Los PEs pueden emitir o pueden publicar resultados y son alojados en servidores llamados nodos de procesamiento (PNs). Los PNs son responsables de escuchar eventos, rutear eventos a los PEs del nodo y despachar eventos a través de la capa de comunicación. Los eventos son encaminados usando una función de *hashing* sobre los valores de los atributos hacia el PE apropiado. Por otro lado, la capa de comunicación utiliza Zookeeper, Hunt et al. (2010), el cual provee manejo de clusters y reemplazo automático de nodos que fallan. S4 usa encaminamiento estático, es parcialmente tolerante a fallas, y no posee mecanismos de balanceo dinámico de carga.

Actualmente S4 está en fase de incubación en la fundación Apache, pero no ha tenido avances en el proyecto desde el año 2013.

Apache Storm

Storm de *Twitter* es una plataforma similar a S4, la cual es publicada como una API para la computación con flujos de datos en tiempo real y de forma escalable. El modelo de programación de Storm está basado en dos primitivas básicas para la transformación de flujos de datos que deben ser implementados de acuerdo a la lógica de las aplicaciones: *spouts* y *bolts*. Un *spout* es una fuente de flujo de datos y un *bolt* hace una transformación de un solo paso sobre el flujo de datos, creando un nuevo flujo basado en la entrada que recibe. Transformaciones complejas requieren múltiples *bolts*, los cuales crean topologías o grafos, el nivel más alto de abstracción en *Storm*. La plataforma soporta tolerancia a fallas a través de un proceso maestro llamado *Nimbus*, el cual garantiza el procesamiento de todos los mensajes a través del uso de una base de datos para el almacenamiento. Sin embargo, esta base de datos es su mayor desventaja respecto de S4 puesto que no es completamente distribuido. Storm define diferentes técnicas

para el particionamiento de *streams* de datos y para la paralelización de *bolts*. Sin embargo, se complejiza el desarrollo de aplicaciones. Al igual que S4, Storm usa *Zookeeper* ,Hunt et al. (2010), en la capa de comunicación.

Apache Spark

Spark, Apache (2016), es una plataforma de computación de código abierto para análisis y procesos avanzados, que tiene muchas ventajas sobre *Hadoop*. Desde el principio, *Spark* fue diseñado para soportar en memoria algoritmos iterativos que se pudiesen desarrollar sin escribir un conjunto de resultados cada vez que se procesaba un dato. Esta habilidad para mantener todo en memoria es una técnica de computación de alto rendimiento aplicado al análisis avanzado, la cual permite que *Spark* tenga unas velocidades de procesamiento que sean 100 veces más rápidas que las conseguidas utilizando *MapReduce*. Esta plataforma asegura a los usuarios la consistencia en los resultados a través de distintos tipos de análisis, Tirados (2014).

2.2.3 Escalabilidad de sistema de procesamiento de *streams*

En los sistemas de procesamiento de *streams*, la escalabilidad puede lograrse usando: (1) paralelización de los elementos de procesamiento y (2) creación y eliminación de nodos de procesamiento. Lo ideal es utilizar ambas técnicas de escalabilidad para lidiar con *stream* masivos de datos que cambian su tamaño en el tiempo.

- Paralelización: Si el volumen del stream de datos es muy grande, un elemento de procesamiento en una máquina no puede lidiar con el procesamiento, por lo que se requiere paralelizar horizontalmente los operadores. *StreamCloud GulisanoStreamCloud* por ejemplo, puede distribuir cualquier elemento de procesamiento en un conjunto de nodos independientes (*share-nothing*). El objetivo es dividir el conjunto de datos en múltiples *sub streams* que fluyen en paralelo. Ésta paralelización es transparente al usuario y se hace en tiempo de compilación. *StreamCloud* agrega un balanceador de carga al inicio de cada división de flujo para redistribuir la salida de los nodos. Además, agrega un agregador de entradas para recibir lo elementos desde los balanceadores de carga. Estos componentes son especializados en el tipo de operador que se quiere paralelizar. Por otro lado, S4 ,Neumeyer et al. (2010), y *Storm* logran paralelismo horizontal entregándole esta tarea al programador, usando un modelo de programación especialmente diseñado para distribuir elementos de procesamiento. En *Storm*, el *stream* de datos se distribuye equitativamente entre los trabajadores disponibles y en S4 se distribuye generalmente usando una función

de *hash* para que los eventos con un mismo valor sean procesados por el mismo elemento de procesamiento.

- Elasticidad: En el contexto de *cloud computing*, hay un costo asociado al uso de máquinas que no están siendo utilizadas. Por otro lado, si se tiene menos capacidad de la necesaria para el procesamiento se produce sobrecarga de los nodos lo que conlleva la pérdida de datos afectando el rendimiento. Es por eso que agregar y eliminar máquinas en tiempo de ejecución para ajustarse al procesamiento es de suma importancia. En Esc, Satzger et al. (2011), para lograrlo existe un módulo administrador que es responsable de mantener estadísticas sobre la carga de las máquinas y el tamaño de los buffers. Esta información es usada para la adaptabilidad y distribución tomando en cuenta los costos monetarios. La función de *hash* se reemplaza con nuevos parámetros para lograr la nueva distribución. Esc introduce el concepto de reescritura del grafo, donde un PE puede dividir y agregar en otros PE. Similarmente, *TimeStream*, Qian et al. (2013), propone usar una substitución resiliente de cada PE, usando el particionamiento del hash, paralelización de los PE y unión del procesamiento. SEEP, Castro Fernandez et al. (2013), por otro lado, tiene un mecanismo para aumentar el número de nodos, detectando cuellos de botella, compartiendo la carga del operador entre un conjunto de nuevos operadores particionados. El estado de la partición se mantiene y se reparticiona el espacio de claves de las tuplas procesadas. Finalmente, Gedik et al. (2014), propone auto-paralelizar en tiempo de ejecución. usando migración de estado y monitoreo del flujo de datos.
- Localización: Otros trabajo se enfocan en optimizar la localización de elementos de procesamiento en los nodos de procesamiento, como el trabajo realizado en el contexto del proyecto FONDEF Idea y T-Storm, Xu et al. (2014), que modifica *Storm* de modo de disminuir al máximo el tráfico entre nodos de procesamiento.

2.2.4 Tolerancia a fallas en sistema de procesamiento de *streams*

Se puede analizar la tolerancia a fallas a nivel de recuperación de nodos de procesamiento caídos o a nivel de procesamiento de *stream*. A nivel del procesamiento de *stream* hay sistemas que proveen la seguridad que una tupla se va a procesar exactamente una vez como *Storm* y otros como S4 Neumeyer et al. (2010) son adecuados en escenarios donde perder una tupla no tiene mayor impacto en el resultado.

- En los esquemas basados en replicación Hwang et al. (2005), Shah et al. (2004) y Balazinska et al. (2008), el sistema de procesamiento de stream crea un número de copias o réplicas

que le permiten tolerar un número igual de fallas simultáneas. En el caso de que una falla ocurra, el operador involucrado es reemplazado por una de sus réplicas la cual posee el mismo estado que el operador fallado. Por otro lado, las técnicas basadas en replicación son capaces de reducir el costo de almacenamiento, sin embargo introducen un alto costo en términos de memoria puesto que deben mantener más recursos de procesamientos sumados a la mantención de la réplica.

- *Checkpointing*: En los esquemas basados en *checkpoint* Hwang et al. (2005), Wang et al. (2012), cada recurso de procesamiento o operador puede llevar a cabo dos acciones: 1) almacenar periódicamente su estado o 2) retener las tuplas de salida en un *buffer* hasta que estas han sido almacenadas en el siguiente operador. En el caso de que exista una falla, el operador es reiniciado y se le copia el último estado almacenado o checkpoint. S4, Neumeyer et al. (2010), usa un sistema de archivos para implementar almacenamiento de *checkpointing*. El *checkpointing* puede ser seguido de re-procesar eventos desde el *checkpointing*, sin embargo, puede ser que la complejidad del procesamiento, la masividad de los datos y sobrecarga no lo permita y se requieran aumentar el número de nodos de procesamiento. CEC, Sebepou & Magoutis (2011), propone un mecanismo que garantiza tolerancia a fallas al tomar *checkpointing* del estado de los PE de manera incremental, entre *checkpoints* se mantiene un *log* de las tuplas en un *buffer*, el cual puede ser restrictivo. *TimeStream*, Qian et al. (2013), al igual que *Storm* y *MillWheel*, Akidau et al. (2013), provee garantías que las tuplas serán procesadas exactamente una vez, al almacenar estados y las dependencias de salida de cada operador. *Zaharia*, Zaharia et al. (2013), usa técnicas de procesamiento de batch para mantener la tolerancia a fallas de manera asíncrona y en paralelo, al igual que en SEEP, Castro Fernandez et al. (2013), donde cada operador es tratado como una entidad independiente.
- Monitoreo: En Esc, Satzger et al. (2011), se utilizan árboles de supervisión de los procesos, donde el padre monitorea a los hijos y los reinicia cuando fallan. El monitoreo se realiza usando mensajes. Otros sistemas confían en sistemas como *Zookeeper* para mantener la tolerancia a fallas del sistema y reemplazar nodos caídos por otros que realicen el procesamiento.

2.2.5 Clasificación de eventos

Por otra parte, la clasificación de texto en servicios de *microblogging*, como *Twitter* es un problema cuya solución tiene diferentes puntos de vista, los métodos tradicionales incluyen hacer uso de una bolsa de palabras para clasificar según el contenido del texto, construcción

de n-gramas para clasificar según términos co-ocurrentes o ubicar el texto en una categoría haciendo uso de técnicas de aprendizaje de máquina o *Machine Learning*, Nguyen & Jung (2015). Este último método ya ha sido comprobado por diversos autores, entre ellos Maldonado (2012), quien utilizó este método para realizar su memoria donde clasificaba *tweets* según sentimientos positivos, negativos o neutros.

En el marco de las jornadas chilenas de la computación Gonzales & Wladdimiro (2014), propusieron un modelo, desarrollado para el proyecto PMI USA 1204: Despliegue agil de aplicaciones para desastres Nicolás Hidalgo Castillo (2014), donde analizaron el rendimiento de una aplicación de clasificación de necesidades básica al ser implementada en un sistema de procesamiento de *streams* como S4. En se propone un modelo basado en *Yahoo! S4* donde haciendo uso del paradigma de procesamiento de *streams* de datos se forma un grafo cuyos nodos (Elementos de procesamiento o PE, por sus siglas en inglés), dividen el procesamiento en pequeñas tareas fácilmente replicables para paralelizar el *pepeline*. En esa ocación desarrollaron distintos tipos de operadores mencionados a continuación:

- Recolector: haciedo uso de la API de *Twitter* obtiene el *stream* de datos del mismo.
- *Scheduler*: discrimina cada *tweet* según la categoría que pertenece (Información, agua, electricidad o alimento), mediante el uso de una bolsa de palabras y la distancia *Hamming*.
- Filtrado: utilizaron en su trabajo un clasificador basado en *machine learning* para verificar la subjetividad de un *tweet*, por lo que ya esta demostrado que esta herramienta es capaz de categorizar texto, por lo que puede ser aplicada para las entradas de *Twitter*.
- Relevancia: identificar si una información es o no confiable haciendo uso de la cantidad de publicaciones del usuario, sus seguidores y a quienes sigue para estimar una reputación del autor.
- Ranking: hace uso de la información anterior, decidiendo a qué le entrega mayor importancia.

Los autores concluyeron basándose en la carga computacional la importancia de una replicación adecuada para distribuirla entre los PE, pero no fueron concluyentes en cuánto o qué nivel de replicación es el adecuado o cuándo replicar.

A modo de comentario, diversos autores, entre los que podemos mencionar a Valer (2011), Weng & Lee (2011), Maldonado (2012), han señalado las dificultades que se presentan al trabajar utilizando como entradas los estados públicos (*tweet*) de los usuarios de *Twitter*, dentro de las dificultades señaladas se encuentran, por ejemplo, el acceso a la información; si bien existen accesos públicos a la información éstos son restringidos tanto en cantidad como en tiempo: Este punto de acceso permite acceder a un 1% de la información generada en un instante, es decir, por cada cien *tweets* sólo puede accederse a uno de ellos. Sólo se permite realizar 180 consultas

cada quince minutos (aproximadamente 12 consultas por minuto) y, en el caso de ser un usuario identificado, se aumenta a 450 consultas dentro del mismo intervalo de tiempo (aproximadamente 30 consultas por minuto). Por otro lado existe un punto de acceso pagado denominado *FireHose* el cual entrega libre acceso a la información.

Por otro lado Valer (2011), señalan que la dificultad radica en el hecho de que cualquier persona puede realizar publicaciones en esta red social, induciendo ruido en la información (considerando el ruido como toda información que aparece junto a la deseada, pero no aporta nueva), además de, al ser publicaciones de máximo 140 caracteres es complejo contextualizar el contenido.

CAPÍTULO 3. REQUERIMIENTOS

Éste capítulo detalla los requisitos de las aplicaciones, descritos como historias de usuario. Éstos señalan las necesidades de los clientes.

El cliente es uno de los clientes del proyecto FONDEF IDeA, código ID15I10560, proyecto orientado a generar herramientas para la gestión de desastres naturales. Los requerimientos se levantan en reuniones con los investigadores y responsables de aplicaciones del proyecto donde se estipulan las funcionalidades deseables, dichas reuniones tuvieron lugar en el Departamente de Ingeniería informática de la Universidad de Santiago e Chile.

3.1 PROCESO DE TOMA DE REQUERIMIENTOS

Las primeras reuniones tuvieron un caracter exploratorio, en ellas se discutieron aspectos referentes al estado del arte y se acotaron los alcances del proyecto. En reuniones subsecuentes y mediante conversaciones con el equipo del proyecto FONDEF IDeA se encontraron los requisitos descritos en la tabla 3.1 mediante los cuales fue posible construir las historias de usuarios necesarias para la metodología que se está empleando. Los requisitos mostrados a continuación se corresponden con el identificador RFXX para los requerimientos funcionales, donde XX corresponde al número del requisitos y RNFYY, para el caso de los requerimientos no funcionales donde YY corresponde al número del requisito.

3.2 HISTORIAS DE USUARIO Y CRITERIOS DE ACEPTACIÓN

Los requerimientos antes mencionados se agruparon para, según lo descrito por la metodología *extreme programming*, como historias de usuario. Estas historias de usuario tienen la siguiente nomenclatura para su identificación: Aquellos que guarden relación con la aplicación de detección se identifican como ’HU-cXX’, donde XX corresponde al número del requisito; ’HU-vYY’ para aquellas que correspondan a la aplicación interfaz, donde, al igual que en el caso anterior, YY corresponden al número del requisito.

La Tabla 3.2 presenta las historias de usuario correspondientes a los requerimientos de todo el sistema en general, sin hacer referencia al cómo está dividido éste.

Tabla 3.1: Requisitos encontrados
Fuente: Elaboración Propia, (2016)

Identificador	Requisito	Dependencia
RNF00	Se ha de implementar un sistema de procesamiento de {streams}.	-
RNF01	La fuente de datos debe ser {Twitter}.	-
RNF02	Debe obtenerse la información de {Twitter} en tiempo real.	RNF01
RNF03	El sistema debe ser escalable	RNF00
RNF04	El sistema debe presentar una interfaz donde el usuario pueda opearar.	-
RNF05	La interfaz debe poder recargar su información sin actualizar la misma	RNF04
RNF06	Los parámetros utilizados para operar el sistema deben ser modificables.	-
RNF07	Se debe presentar un mapa para el despliegue de la información.	RNF04
RNF08	Cada categoría debe tener un icono particular.	RF05
RNF09	La información en el mapa debe agruparse según nivel de acercamiento de éste.	RNF08, RNF04
RNF10	Se debe poder elegir qué eventos se muestran por categoría	RNF08, RNF04
RNF11	Se deben mostrar estadísticas de procesamiento para la consulta actual.	RF02, RNF04
RNF12	Se debe mostrar desde donde viene un evento, es decir, qué estado lo generó al visualizarlo en el mapa.	RNF10, RNF09, RNF04
RNF13	Debe mostrarse una línea temporal para seleccionar el intervalo.	RF08
RNF14	Debe mostrarse un histograma con la cantidad de eventos pasado por fecha.	RF08
RF00	Se deben detectar las necesidades expresadas en los {tweets}	RNF00, RNF01, RNF02
RF01	Los operadores del sistema deben especificarse como {bolts}	RNF00
RF02	Se debe poder especificar términos de búsqueda,	RNF00, RNF04,
RF03	Los términos ingresados deben expandirse, automáticamente, para abarcar nuesvos términos útiles.	RF03
RF04	Sólo se trabajaran {tweets} en español.	RF01, RNF00
RF05	La detección de necesidades debe realizarse de forma dinámica, no utilizando bolsas de palabras.	-
RF06	Se debe poder acceder a eventos pasados.	-
RF07	Debe existir un sistema de persistencia de datos.	RF06
RF08	Se debe poder especificar un intervalo dentro del cual mostrar datos.	RF06
RF09	El sistema debe funcionar de forma continua	-

Tabla 3.2: Historias de usuario.
Fuente: Elaboración Propia, (2016)

Identificador	Historia de usuario
HU-c00	Como cliente quiero capturar necesidades de la población en tiempo real cuando el país se encuentre en un escenario de catástrofe natural para poder contar con información para asistir a la población afectada.
HU-c01	Como cliente quiero que las necesidades detectadas se recojan desde la información generadas en redes sociales redes sociales para que las personas sean la fuente primaria.
HU-c02	Como cliente quiero que la búsqueda de necesidades se vea enriquecida para abarcar nuevos términos de búsqueda para abarcar un conjunto mayor de información.
HU-v00	Como usuario quiero una interfaz donde pueda visualizar el comportamiento del sistema de detección para poder interactuar con el.
HU-v01	Como cliente quiero que las necesidades detectadas puedan ser asociadas a un punto en un mapa geográfico para poder identificar el lugar físico de su fuente.
HU-v02	Como usuario quiero que puedan aplicarse filtros a la visualización de los puntos de modo que según la distancia entre ellos, cuáles se quieran mostrar y el nivel de acercamiento que tenga el mapa se entreguen diferentes formas de mostrar la información para que la información se visualice con facilidad.
HU-v03	Como usuario quiero que la visualización de eventos se realice en tiempo real para tomar decisiones rápidas cuando la situación lo amerite.
HU-v04	Como usuario quiero visualizar eventos pasados, además quiero poder seleccionar un intervalo de tiempo y que el sistema muestre todos los eventos que se hayan detectado dentro de aquel intervalo de modo que pueda realizarse una análisis a posteriori de la emergencia.
HU-v05	Como usuario quiero poder especificar términos de búsqueda para acotar la búsqueda a aquel contenido que contenga elementos que se correspondan con ellos.
HU-v06	Como usuario quiero que cada punto, correspondiente a una necesidad específica, tenga un diseño particular fácilmente identificable.
HU-07	Como usuario quiero que sea posible visualizar estadísticas del procesamiento de la aplicación por consulta.
HU-08	Como usuario quiero poder modificar cuánto tiempo se visualizará un evento antes de que sea considerado antiguo y cada cuánto tiempo se ha de añadir la información de los nuevos eventos.

Estas historias de usuario se corresponden con los criterios de aceptación descritos en la Tabla 3.3 que se presenta a continuación.

Tabla 3.3: Criterios de aceptación.
Fuente: Elaboración Propia, (2016)

Identificador	Criterio de aceptación
HU-c00	· Debe contruirse el sistema de procesamiento de stream. · Debe capturarse la información desde Twitter. · Debe usarse la información que llega al sistema y no almacenarla para trabajarla por lotes. · El sistema debe concluir con un dato etiquetado en la base de datos.
HU-c01	· Twitter debe ser desde donde se obtienen los datos de entrada del sistema.
HU-c02	· Deben implementarse técnicas para realizar expansión de la consulta.
HU-v00	· Debe contarse con una interfaz que muestre los eventos detectados por el sistema.
HU-v01	· El mapa de eventos debe mostrar los eventos, asociados a un punto geográfico.
HU-v02	· Debe poder filtrarse por modos de agrupamiento. · Debe poder filtrarse por categorías. · El mapa debe permitir modificar su nivel de acercamiento.
HU-v03	· El mapa debe actualizarse automáticamente.
HU-v04	· Debe poder seleccionarse el intervalo. · Los eventos en el mapa deben modificarse según el intervalo que se seleccione. · Debe seleccionarse el intervalo mediante una línea de tiempo. · Debe mostrarse un histograma con los eventos pasados presentes en el sistema.
HU-v05	· Debe existir un lugar donde especificar éstos términos. · El sistema debe mostrar qué términos se están utilizando para la búsqueda actual.
HU-v06	· Cada evento de categorización diferente debe tener un icono particular.
HU-v07	· Deben mostrarse la cantidad de usuarios diferentes que han emitido estados detectados. · Deben mostrarse la cantidad de necesidades detectadas. · Deben mostrarse la cantidad de eventos procesados.
HU-v08	· Debe existir un lugar donde realizar el cambio de parámetros. · Debe ser explícito el cambio de los parámetros de funcionamiento en la interfaz.

CAPÍTULO 4. DISEÑO E IMPLEMENTACIÓN

Este capítulo detalla la fase de construcción de la aplicación se señalan las decisiones tomadas a lo largo del desarrollo para cumplir con lo solicitado, descrito en el capítulo anterior.

Yendo desde lo general a lo particular se presenta la arquitectura del sistema para continuar con las decisiones que llevaron al sistema a ser lo que es.

4.1 ARQUITECTURA DEL SISTEMA

El sistema de detección de necesidades, en su conjunto, conta de dos aplicaciones independientes que trabajan unidas para realizar el proceso de detección de eventos.

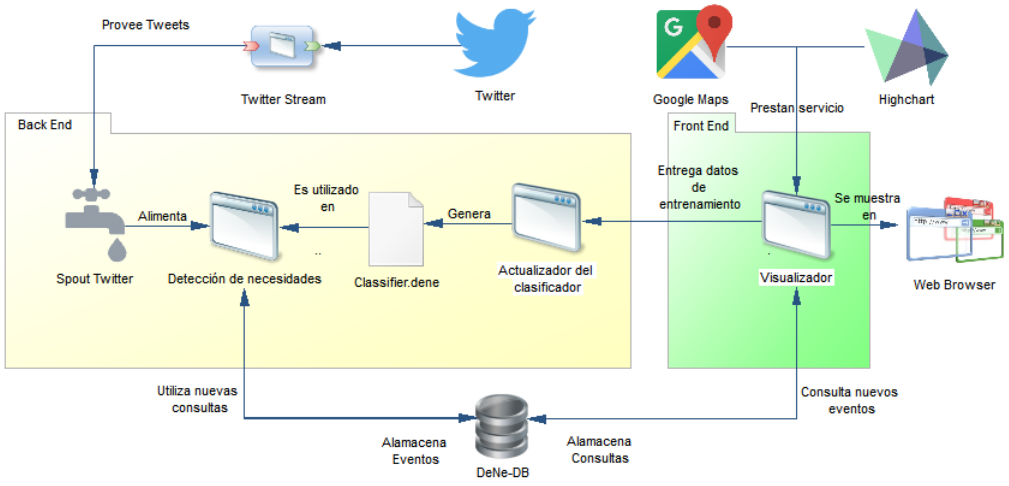

Figura 4.1: Arquitectura del sistema.
Fuente: Elaboración Propia, (2016)

La figura 4.1 presenta la arquitectura del sistema. Éste está compuesto de dos aplicaciones. La arquitectura del sistema pasa a describirse a continuación.

Todo el sistema es alimentado por el *stream* provisto por el servicio de *streaming* de *Twitter*, pues ésta red social genera más de 140 millones de *tweets* diariamente, Jones (2013), y su uso se dispara en períodos de crisis, Olteanu et al. (2015), por ello es una buena fuente de información primaria, es decir, que la información viene desde la misma población afectada.

Para ser capaz de procesar el flujo de información que llega de manera continua se hace uso *Apache Storm*, el sistema de procesamiento de *streams* que fue presentado en la sección 2.1.1, el cual divide el procesamiento en pequeñas tareas, en el caso del flujo desde *Twitter*, éste ingresa, por medio de un *spout*, a la aplicación detectora de necesidades la que procesa y clasifica

cada *tweet* del *stream*, ésto se hace por medio de una serie de operadores denominados *bolts* y lo transforma, de ser posible, en un marcador almacenado en la base de datos. La serie de pasos o *bolts* por los que pasa un un *tweet* son descritos en la sección 4.4.2.

Los datos almacenados en la base de datos son leídos por la visualizadora la que se encarga de mostrar al usuario, mediante su navegador *web*, la ubicación y cantidad de eventos tanto en tiempo real, como históricos. La visualización de éstos últimos se realiza por medio de las API proporcionadas por *Google Maps* y *Highcharts*. El primero se utiliza para obtener un mapa geográfico con la capacidad de ubicar geográficamente puntos por medio de marcadores y modificarlos a voluntad, mientras que el segundo para transformar los datos a una línea temporal que muestre al usuario la cantidad de eventos por cada período.

Para realizar el etiquetado de los datos, es decir, la categorización del texto, se utiliza un clasificador *Naïve Bayes*, el cual está almacenado en un fichero que permite ser modificado, siempre y cuando, se haga entrega de un archivo de entrada con un formáto específico para realizar un entrenamiento del clasificador. La elaboración de éste clasificador se detalla en la sección 4.3.8.

La base de datos presta una función escencial al funcionamiento del sistema. Es allí donde se almacenan las consultas realizadas, con las cuales filtran, según las necesidades del usuario, el *stream*, además almacena los elementos que son mostrados por el visualizador en el mapa. Resumiendo lo anterior, la base de datos presta servicios de comunicación al sistema. Las aplicaciónes y justificación del uso de esta base de datos se presenta en la sección 4.3.1.

El sistema descrito está diseñado para operar de forma continua en condiciones de alto tráfico como lo es al ocurrir una emergencia del tipo catastrófica en el país. A pesar de lo anterior, se asume la existencia un sistema externo encargado de detectar cuándo se produce una situacion de las características antes mencionadas e inicie el sistema de detección, pues no está dentro de los alcances de este trabajo el que se detecte cuándo es que se ha producido un evento y la población comience a producir nuevos estados en las redes sociales.

El desarrollo de la aplicación se guia por el cumplimiento de las historias de usuario descritas en la Tabla 3.2. Para ello se trata cada historia como un problema individual que luego han de ser constituidas en la aplicación final.

4.2 CARACTERÍSTICAS DEL SISTEMA

A continuación se realiza mencionan las características que ha de tener el sistema, se hace esto para tener en cuenta en las decisiones tomadas en las siguientes secciones.

En primer lugar, es necesario hacer hincapié en el contexto en que el sistema

opera. *Twitter* es un servicio que cuenta con millones de usuarios activos, los que generan constantemente nuevo contenido que es emitido por la API de *streaming*, lo que significa que la aplicación está sometida a un gran estrés cuando se encuentra en funcionamiento. Dado el contexto descrito la aplicación ha de ser capaz de soportar ese flujo de información.

En segundo, y como se explicó en la sección 1.5.2, el funcionamiento interno de las aplicaciones construidas con *storm* se puede esquematizar por medio de un grafo dirigido donde los nodos se corresponden con los operadores definidos en la topología y que pueden tener diferente cantidad de elementos por procesar y tardar tiempos distintos en realizar su labor. Lo anterior sugiere que pueden existir niveles en los que se producen cuellos de botella en el *pipeline* de procesamiento. Considerando lo anteriormente expuesto el sistema ha de estar preparado para responder de la mejor forma posible cuando se produzcan estas obstrucciones en el proceso.

En tercer lugar, el uso de un clasificador de texto involucra que la calidad del etiquetado está dada por el cómo éste se construyó. La construcción está dada por los datos de entrenamiento; mientras más datos se entreguen, probablemente, la calidad del clasificador sea mayor. Esto quiere decir que el clasificador puede ser mejorado y que consituye una limitante el mantener éste estático.

4.3 DECISIONES DE DISEÑO

Esta sección presenta las decisiones tomadas por el autor al momento de diseñar las aplicaciones que componen el sistema de detección de necesidades.

4.3.1 Comunicación

Se ha de justificar lo expuesto hasta ahora al hacer mención de la existencia de dos aplicaciones que componen el sistema de detección de necesidades. El uso del sistema de procesamiento distribuido, *storm*, dificultaba en su momento la integración con un *framework* para aplicaciones Java y, dado que el sistema ha de poseer una interfáz donde el usuario pueda visualizar los eventos detectados, por ello se decidió construir ambos módulos; detección y visualización en aplicaciones separadas. En un primer momento se pensó comunicar ambas aplicaciones por medio de peticiones REST cuyo contenido fuesen tanto las consultas ingresadas por el usuario para realizar una búsqueda más exhaustiva, como los datos correspondientes a marcadores ubicados en el mapa del visualizador. Esta aproximación de solución se esquematiza en la Figura 4.2, presentada a continuación.

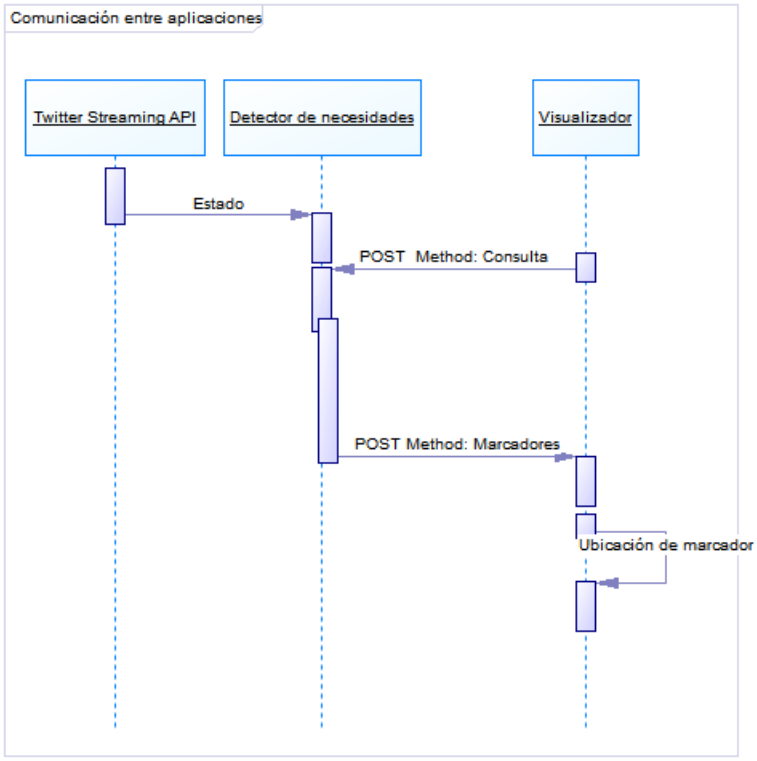

Figura 4.2: Esquema que representa la comunicación entre aplicaciones en primeras etapas del desarrollo.
Fuente: Elaboración Propia, (2016)

Esta aproximación no consideraba la existencia de un sistema de persistencia de información; al considerar la persistencia la comunicación ya no se realiza por medio de peticiones REST, sino que se utiliza la base de datos como intermediario, así al momento de realizar una consulta, ésta es almacenada en la base de datos y recodiga por el sistema de clasificación; cuando el sistema clasifica correctamente un evento, éste también es almacenado en la base de datos y recogido por la aplicación visualizadora. De esta forma la comunicación del sistema pasa a realizarse de forma que se esquematiza en la Figura 4.3.

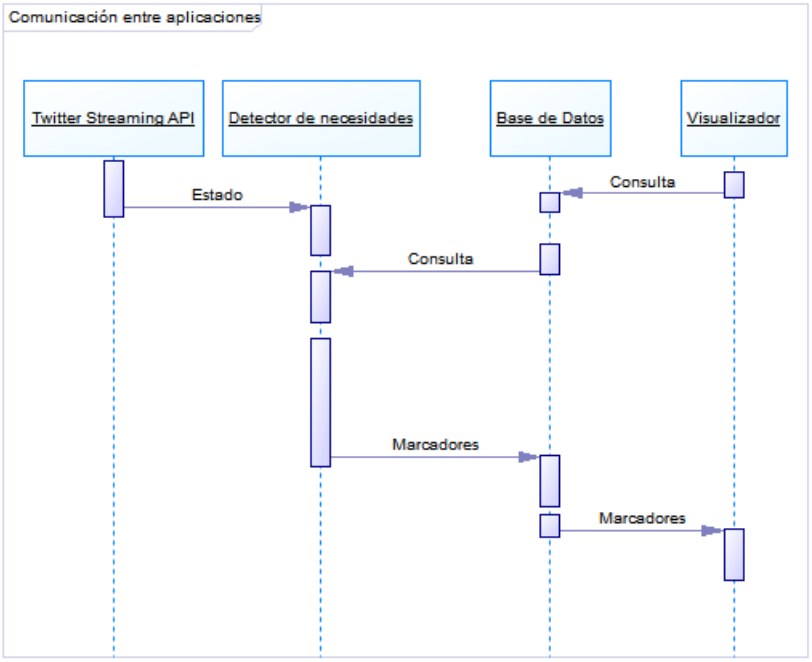

Figura 4.3: Esquema que representa la comunicación entre aplicaciones del sistema detector de necesidades.
Fuente: Elaboración Propia, (2016)

En la historia de usuario HU-v04, al hacer referencia a información pasada, se infiere la necesidad de la implementación de un sistema de persistencia de datos. Es, principalmente, por esta razón que llevó a que modificara la primera forma de comunicación.

4.3.2 Persistencia

Si bien está decidida la implementación de un sistema de persistencia en la aplicación, no se ha definido cuál ha de ser el sistema de gestión de base de datos que se ha de utilizar, es por ello que en ésta sección se presenta la decisión tomada con respecto a este tema.

Se consideraron los principales sistemas de bases de datos utilizados y conocidos por el autor, dentro de los cuales se encontraban herramientas como: MySQL, PostgreSQL, SQL Server, MongoDB, entre otras. Dadas las características y las condiciones con las cuales opera el sistema de detección se requiere de un DBMS con rápido tiempo de respuesta en operaciones lectura/escritura; la decisión se tomó en base a los datos que se manejaran, pues no se apreció necesidad de implementar una base de datos relacional, de esta forma y teniendo en cuenta los resultados presentados en pruebas empíricas realizadas por Macool (2013) en las cuales mostró que el tiempo de respuesta (en operaciones de lectura) es significativamente menor en MongoDB

que en dos de los DBMS más conocidos como MySQL y PostgreSQL. Lo anterior, sumado al hecho de la capacidad de escalar de MongoDB reportada en fuentes oficiales o por diversos desarrolladores como Tobin (2016) que han compartido sus experiencias en la *web*, llevaron a decidir que MongoDB debiera ser el sistema de gestión de base de datos que se utilizase en el sistema.

Para realizar la conexión de MongoDB y el *framework* se utiliza, específicamente, dos bibliotecas: La primera corresponde a un ORM (Mapeo Objeto-Relacional), denominado Jongo, la cual hace uso de la segunda llamada Jackson, para realizar la conversion de JSON a objeto.

Volviendo a la historia de usuario que originó la necesidad de contar con un sistema de persistencia de datos, habiendo resuelto lo anterior la siguiente problemática se presenta como ¿qué datos han de guardarse? Según la definición de la historia en la que se señalan ”eventos pasados dentro de un intervalo de tiempo”, se infiera que ha de guardarse tanto el contenido visible del dato, la clasificación que se le asignó y la fecha en que se identificó, para ello y dado que se seleccionó MongoDB, y aunque no es necesario, se especificó un esquema para los documentos de la colección, dados los datos que se almacenan sólo resta tener la información correspondiente a la ubicación, por lo que el esquema se definió como se presenta en la Figura 4.4 correspondiente a la colección ”Markers”.

```
1. {
2.     "_id": objectId("_MongoDB_ID"),
3.     "contenido": "Contenido del tweet",
4.     "categoría": "Categoría del tweet",
5.     "latitud": "Coordenada Latitud",
6.     "longitud": "Coordenada Longitud",
7.     "generatedAt": ISODate("YYYY-mm-ddTHH:mm:sssZ")
8.
9. }
```

Figura 4.4: Ejemplo de documento en la colección Markers.
Fuente: Elaboración Propia, (2016)

4.3.3 Sistema de procesamiento

Ya se mencionó que se seleccionó storm para construir el detector de necesidades, pero no se ha especificado el porqué de ello. Por este motivo en los siguientes párrafos se exponen las razones por las cuales se tomó esta decisión.

Dado el contexto del funcionamiento del sistema, éste ha de entregar respuestas

rápidas ante una emergencia; para ello, y como es descrito en la historia de usuario HU-c00, se requiere de un sistema capaz de procesar eventos en tiempo real que, dado el *peak* de información que recibe el sistema éste debe ser escalable. El problema en este punto es el cómo construir un sistema que cumpla capaz de identificar necesidades y que posea esta, no menor, característica.

Se consideraron sistemas de procesamiento distribuido; estos sistemas tienen la particularidad de ser una red de computadores (nodos, en general), que el usuario percibe como un solo gran sistema. Estos sistemas pueden ser de diversos tamaños, y suelen ser confiables, pues en caso de que un componente (nodo) falle, otro es capaz de reemplazarlo. IPN (2013). En un inicio se consideraron tres plataformas sobre las cuales puede construirse un sistema que pudiese cumplir con lo solicitado; dichas plataformas fueron Apache S4, Apache Storm y Apache Spark.

Apache S4, pese a su simplicidad, no continuó con su desarrollo luego del año 2013 y nunca tuvo una versión estable 1.0, razones por las cuales se dejó como segunda opción. Apache Spark, pese a contar con continuos *releases*, una comunidad de desarrolladores no menor y permitir la elaboración de sistemas escalables no era lo que se buscaba en aquel momento como herramienta de desarrollo, pues está orientado al procesamiento por lotes y no en tiempo real. Al momento de consultar con los, en este caso, clientes, éstos esperaban que el sistema, internamente, se comportara según el paradigma de procesamiento de *streams*, por medio de operadores dispuestos en un grafo. Por ello finalmente se optó por *Apache Storm*.

Storm permite construir sistemas que cumplan con las características de un sistema distribuido, como lo son: Escalabilidad (Tanto horizontal como vertical) y tolerancia a fallos (como la capacidad de un sistema para realizar correctamente y en todo momento aquello para lo que fue diseñado). Estos sistemas están compuestos por dos tipos de elementos: *Spout* y *Bolt*, que fueron descritos en el capítulo 2. Al combinar esos elementos se da origen a un grafo dirigido, como el presentado en la Figura 2.2 en la página 13, donde cada elemento de procesamiento (*bolt*), cumple con una determinada tarea utilizando como entrada la salida del elemento anterior.

4.3.4 Obtención de datos para el funcionamiento del sistema

La historia de usuaio HU-c01 refleja desde dondé se han de obtener los datos, pero es necesario especificar más aún. Como se señaló al momento de definir los alcances de este trabajo, sólo se utiliza *Twitter* como fuente de información, así la unidad de información pasa, desde ahora, a llamarse como se habitúa en aquella red social: el *Tweet*.

El asunto es, entonces, cómo obtener la información que está produciéndose en *Twitter* en tiempo real. Esta red social ha implementado una serie de interfaces para permitir

a los desarrolladores acceder a sus datos; en particular, la *Streaming API*, es aquella que permite acceder a la información de *Twitter* con baja latencia. Twitter (2016).

Existen tres tipos de *streaming endpoints* disponibles, cada uno para un caso de uso particular y son descritos en la Tabla 4.1.

Tabla 4.1: *Streaming endpoints* de *Twitter*.
Fuente: Elaboración Propia, (2016)

Público	Stream del que fluye la información pública de Twitter. Casos de uso: Seguimiento de usuarios o tópicos específicos o minería de datos.
Usuario	Flujo que toda la información correspondiente a un usuario.
Sitio	Versión multi-usuario de la anterior.

Para esta aplicación la adecuada corresponde a la API pública. En ésta, a la vez, existen dos puntos de acceso; el público y *firehose*. El acceso público es gratuito y permite el acceso a un 1% de la información que se genera en tiempo real y para acceder a el basta con crear una aplicación dentro de *Twitter*. En cambio para acceder a *firehose*, el cual permite acceso total a la información, debe comprarse el acceso. Dadas estas condiciones se seleccionó, previo acuerdo con los clientes, el uso de la API pública.

Para hacer uso de la API descrita con anterioridad es necesario obtener cuatro claves de acceso: *Access Token*, *Access Token Secret*, *Consumer Key (API Key)* y *Consumer Secret (API Secret)*. Para más información sobre cómo conseguir estas claves consulte el Apéndice b..

4.3.5 Especificación de términos de búsqueda

Twitter4J, la herremienta que se menciona en la sección 4.4.2 como aquella que permite obtener el flujo de información desde *Twitter* implementa una forma de filtrado mediante el uso de palabras clave, pero posee una limitante al momento de modificar la búsqueda, deben instanciarse nuevamente los objetos con los cuales se realiza la conexión a la API de *Twitter*, eso se traduce en tiempo de procesamiento perdido, para solucionar este inconveniente se decidió implementar un operador, descrito en la sección 4.4.2, el cual está encargado de realizar el filtrado de acuerdo a términos y llevar a cabo la operaciones descritas en la sección 4.4.2 referente a la expansión de la consulta, pero al ser un operador significa que opera con un nivel de replicación determinado, es decir, existen multiples instancias de operador al mismo tiempo ¿Cómo comunicar el estado de una consulta y que todos los operadores utilicen el mismo filtro?.

Nuevamente la respuesta consistió en recurrir a la base de datos; almacenar la consulta y asignar un estado para controlar el comportamiento del operador. Así el esquema en la base de datos queda tal y como se presenta en la Figura 4.5, documento de la colección

”Queries”.

```
{
    "_id": ObjectID("MongoDB_ID"),
    "terminos": [
        "término A",
        "término B"
    ],
    "estado": "Estado",
    "generatedAt": ISODate("YYYY-mm-ddTHH:mm:sssZ")
}
```

Figura 4.5: Ejemplo de documento en la colección queries.
Fuente: Elaboración Propia, (2016)

Donde la propiedad ”estado” puede tomar dos valores: ”actual” o ”antiguo”, reflejan si una consulta se está llevando a cabo o no. Estos valores son asignados por la aplicación responsable de la interfaz la responsable de recibir los términos de búsqueda por parte del usuario.

Para implementar este operador se utilizaron dos clases llamadas *Current-QueryChecker* y *QueryExpander* desarrolladas, la primera, para detectar cuándo y si es que ha cambiado una consulta en la base de datos y la segunda para desarrollar la labor descrita por el algoritmo de expansión descrito en la sección 4.4.2.

4.3.6 Interfáz del sistema

Teniendo en consideración la característica del desarrollo de esta aplicación como un proyecto ágil con un mínimo de personal para desarrollar se requería de un *framework* que contribuyera a acelerar la construcción de la aplicación. Tras considerar las alternativas más conocidas como *Spring*, *Hibernate* o *JSF* que tienen una curva de aprendizaje elevada, se optó por utilizar un cuarto *framework* que aunque desconocido, promete una simplicidad en su uso. *Play Framework*, construido haciendo uso de Scala y Java permite construir aplicaciones ligeras (tamaño en disco), sin estado (no guarda configuraciones de una sesión para ser utilizadas luego) y por defecto RESTful, ideal para la comunicación entre aplicaciones. Éste *framework* sigue el patrón de arquitectura Modelo-vista-controlador (MVC). Cuenta con un compilador en tiempo real (compila y realiza el despliegue de la aplicación cuando detecta un cambio en el código), lo que agiliza en gran medida el desarrollo, pues al automatizar este proceso mantiene la atención en lo que se está desarrollando.

Para visualizar los puntos encontrados por el detector de necesidades se decidió utilizar la API de *Google Maps* la que permite la colocación de los denominados ’marcadores’

en un punto específico del mapa y asociar a ellos algún tipo de información. Así, aunque el funcionamiento interno esté dirigido por *Play*, la principal funcionalidad del sistema, mostrar el mapa con sus marcadores, es implementada utilizando Javascript.

Estos marcadores, ya ubicados en el mapa, tienen asociado un cuadro de texto dentro del cual refleja el la categoría a la que pertenece y el *tweet* original, el texto, que lo generó. Esto tiene como objetivo permitir decidir, en última instancia, al usuario si ha sido correctamente clasificado.

Según lo solicitado en la historia de usuario HU-v02 se prepararon dos tipos de filtros a la interfaz para la visualización de eventos en el mapa: El primero considera el agrupamiento o *clustering* de marcadores, mientras que el segundo considera el tipo de marcador o marcadores que se desean visualizar.

Para el caso del agrupamiento se definieron tres modos de funcionamiento las cuales se describen a continuación:

1. No agrupar: Mostrar todos los marcadores que correspondan en el mapa de acuerdo al punto geográfico que corresponda en su definición.
2. Agrupar por distancia: Define una grilla invisible en el mapa donde los elementos que calcen en una cudrícula son agregados a un *cluster* y visualizados como tal.
3. Agrupar por categoría: De igual forma que el agrupamiento por distancia, pero sólo agrega elementos que comparan categoría.

Para el segundo caso sólo se definieron dos reglas de funcionamiento las cuales se describen a continuación:

1. Mostrar todos: Muestra elementos de todas las categorias existentes.
2. Mostrar categoría: Para cada categoría mostrar sólo los elementos de aquella categoría.

Al combinar ambos tipos de filtros se tienen potencialmente seis modos de funcionamiento, pero considerando las categorías descritas en al sección 4.3.7 ese número se expande a veintiún modos de funcionamiento del visualizador.

4.3.7 Categorización de necesidades

La definición de las categorías es un punto importante dentro de la construcción de la aplicación. Teóricamente en función a la cantidad de clases (categorías), el tamaño del conjunto de entrenamiento ha de ser mayor o menor.

Inicialmente se consideró la taxonomía definida por Olteanu et al. (2015), pero el equipo del proyecto FONDEF IDeA estableció que no era conveniente utilizar, pues presentaba gran cantidad de categorías y era demasiado específica, se sugirió en su lugar utilizar la clasificación realizada por Nicolás Hidalgo Castillo (2014) en la que se presentaba una categorización de cinco categorías, ellas eran:

1. Necesidades básicas: *Tweet* que entregara o solicitara información sobre serviciós básicos: Agua potable, electricidad y abastecimiento de alimentos.
2. Comunicación: *Tweet* que entregue o solicite información sobre alguna localidad.
3. Seguridad: *Tweet* que señale un riesgo para la población.
4. Personas: *Tweet* que haga referencia al hallazgo o búsquead de una persona desaparecida.
5. Irrelevante: Cualquier otro *tweet*.

Acordando con el equipo, se decidió separar el primer ítem en los tres elementos que lo componen: Agua, alimentos y electricidad. Así, finalmente, se obtienen siete categorías de clasificación.

Los elementos clasificados como ”Irrelevantes” no se muestran en el mapa de eventos, pues hacen referencia a eventos que, pese a haber pasado por todos los operadores anteriormente descritos, no guardan relación con el evento o sus consecuencias.

4.3.8 Clasificador

El proyecto PMI realizado por Gonzales & Wladdimiro (2014), abordó una solución inicial para el problema de la detección de necesidades. En aquella oportunidad se utilizó una bolsa de palabras con términos asociados a las categorías para etiquetar los estados de *Twitter*. Este trabajo busca mejorar aquello realizando la clasificación mediante una técnica de clasificación más poderosa.

En su libro, *Introduction to Information Retrival*, Manning et al. (2008), proponen el uso de *Naïve Bayes*, para realizar clasificación de texto. Esto se logra realizando una implementación del algoritmo descrito en la sección **??**. Este ya ha sido implementado en diversas herramientas como: RapidMiner, Weka (acrónimo de *Waikato Envoirment for Knowledge Analysis*), Mallet (acrónimo de *MAchine Learning for LanguagE Toolkit*).

Tras haber realizado pruebas con ellas, enfocándo la atención en se decidió utilizar Mallet por su simplicidad de implementación junto con la mayor precisión de sus resultados.

La sección 4.2 ya hace referencia a la inconveniencia de utilizar un clasificador estático para el etiquetado de nuevos eventos, al tratarse de un aprendizaje supervisado, donde se requiere de una persona entregue la respuesta esperada para realizar el entrenamiento, no es posible realizar este proceso de manera automática.

Al no poder automatizar el proceso antes señalado se consultó con el equipo del proyecto FONDEF IDeA si es que era factible la implementación de un actualizador manual del clasificador, lo cual fue aceptado.

Segun la metodologia KDD, los pasos que se siguen para construir un nuevo clasificador son los siguientes:

Los datos son seleccionados por el usuario, estos datos se agruparan en un archivo de texto, un archivo CSV en el cual los elementos se separaran utilizando el caracter punto y coma (;). El formato que se utiliza para el archivo de entrada se muetra en la Figura 4.6 , así cualquier archivo que cumpla con el formato permite la creación de un nuevo modelo clasificador.

Figura 4.6: Formato archivo de entrada.
Fuente: Elaboración Propia, (2016)

1. Corresponde a un identificador arbitrario, pero necesario para la herramienta de clasificación Mallet.
2. Corresponde a la etiqueta que categoriza al contenido.
3. Contenido del *tweet* propiamente tal.

El preprocesamiento y transformación de los datos está dado por la definición de los operadores presentados en la sección 4.4.2.

Teniendo en consideración que se utilizan dos aplicaciones distintas, donde en una se construye el clasificador y en otra donde es utilizado. Surge el problema de cómo realizar la comunicación entre ellas. Para solucionar este inconveniente se utiliza una carpeta compartida por ambas aplicaciones. En el caso de sistemas Unix se utiliza el directorio $/opt/DeNe$, mientras que para Windows se utiliza $C : /DeNe/$. En estos directorios se almacena un fichero con el clasificador serializado.

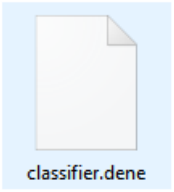

Figura 4.7: Fichero clasificador en $C:/DeNe/$.
Fuente: Elaboración Propia, (2016)

Cada vez que se actualice el clasificador se contrasta el nuevo con el ya existente, de encontrar mayor precisión en el primero, se reemplaza en la carpeta antes mencionada, según el sistema operativo de la máquina que se esté utilizando. En caso contrario, se mantiene al anterior. En ambos escenarios se le da a conocer al usuario la precisión de ambos.

4.4 IMPLEMENTACIÓN DEL SISTEMA

Ésta sección detalla las particularidades de presentadas en la implementación de ambos sistemas, tanto del detector de sistema, como del visualizador.

4.4.1 Visualizador

La implementación del visualizador de eventos se realizó haciendo uso del *framework* de Java *Play*, éste *framework*, por defecto, crea aplicaciones que siguen el patrón de diseño MVC, por lo tanto se tienen tres niveles dentro de la aplicación:

- Modelo: donde están los elementos que permiten interactual con la base de datos.
- Controlador: presentando los métodos de reacción ante los eventos detonados en el nivel de presentación.
- Vista o presentación: muestra las interfaces *web* diseñadas para que el usuario interactúe con el sistema.

Filtrado de marcadores

Dentro del nivel de presentación se encuentra el mapa, proporcionado por la API de Google Maps como se mencionó en la sección 4.3.6. Allí, también, se señaló que existen filtros para la visualización de eventos de manera que se la presentación de éstos se apegue

a las necesidades del usuario. Para implementar estos filtros, internamente, la aplicación hace uso de $n+1$ *clusters*, donde n corresponde al al número de categorías y el cluster extra es para agruparlos a todos. Así, en el caso de querer ver los eventos agrupados, y dependiendo si se quiere o no agruparlos sin discriminación de categoría, se llenan los *clusters* pertenecientes a la visualización general o a la visualización por categoría. Para el caso de querer mostrar sólo una categoría en particular, sólo se permite que los *cluster* se llenen con los elementos de la categoría seleccionada.

Lo anteriormente descrito es presentado a continuación en el Algoritmo 4.1 para facilitar la comprensión de la lógica interna de los filtros presentados.

Para realizar la selección del intervalo mencionado en la HU-v04 se solicitó, por parte del equipo FONDEF IDeA, el uso de una línea de tiempo con intervalo deslizante que, además, mostrase la cantidad de eventos detectados por fecha por medio de un histograma. Para ello se utilizó, inicialmente, se utilizó *JDateRangeSlider*, de la biblioteca Javascript JQRangeSlider, Gautreau (2010). Ésta biblioteca era suficiente para seleccionar el intervalo de fechas y detectar cambios producidos en la línea de tiempo para actualizar los valores, mas no permite la implementación de un histograma externo.

Figura 4.8: Selectorde fechas JDateRangeSlider.
Fuente: Gautreau (2010)

Para lograr implementar ambas dos se utilizó una biblioteca Javascript distinta. La Figura 4.9 presenta la implementación utilizando *HighCharts*, Hønsi & Hjetland (2006). Ésta, al contrario de la anterior, no permitía capturar los cambios en el histograma, para solucionarlo se implementó una función javascript que recogiese los valores del intervalo y arroja un evento cuando se produjece un cambio, este evento se asoció al eje x de la linea temporal, cambiando el valor de la variable *valuesOfAxis* cada vez que se moviese el eje, éste evento es descrito en la Figura 4.10.

Inicialmente este histograma sólo está disponible en inglés, pero permite cambiar todas sus etiquetas manualmente, así, buscando la usabilidad de la aplicación, se modificaron todos los textos y el resultado está visible en la Figura 4.9.

Algoritmo 4.1: Algoritmos de utilización de filtros

Entrada: Tipo de agrupamiento A.

Entrada: Discriminador de categoría K.

Entrada: Marcadores M.

Lista de marcadores L.

Clusters de marcadores $C = \{c_0, \ldots, c_{n+1}\}$.

Para cada m_i perteneciente a M **Hacer:**

Si la categoría de m_i no es "irrelevante" y la categoría de m_i es igual a K **entonces:**

añadir el marcador a L.

Sino: Si K es "todas las categorías" **entonces:**

añadir el marcador a L.

Fin Si

Fin Para

Si A es "no agrupar" **entonces:**

Para cada l_i perteneciente a L **Hacer:**

posicionar l_i en el mapa.

Fin Para

Sino: Si A es "agrupar todos" **entonces:**

Para cada l_i perteneciente a L **Hacer:**

añadir l_i al clister c_0.

Fin Para

posicionar c_0 en el mapa.

Sino:

Para cada l_i perteneciente a L **Hacer:**

añadir l_i al cluster c_{i+1}

Fin Para

Para cada c_i perteneciente a $C - \{c_0\}$ **Hacer:**

añadir l_i al cluster c_i

Fin Para

Fin Si

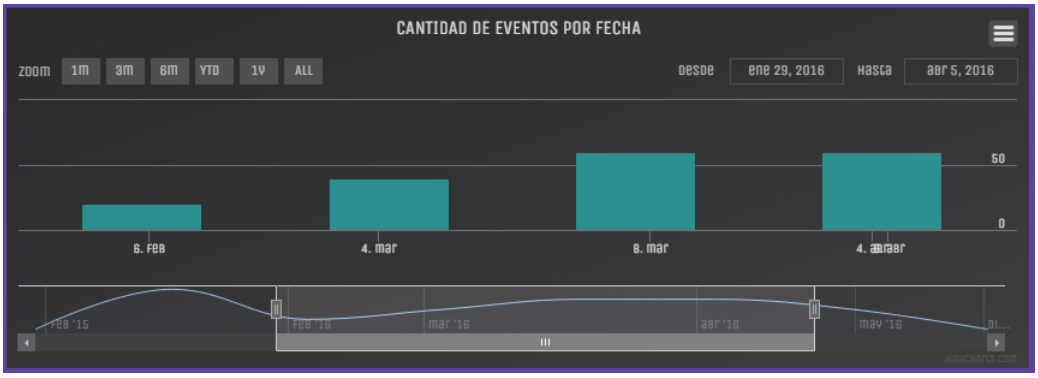

Figura 4.9: Selector de fechas presente en la aplicación.
Fuente: Elaboración Propia, (2016).

```
xAxis: {
    events: {
        setExtremes: function (e) {
            valuesOfAxis[0] = Highcharts.dateFormat(null, e.min);
            valuesOfAxis[1] = Highcharts.dateFormat(null, e.max);
            $('#histograma2').trigger('change');
            }
    }
}
```

Figura 4.10: Implementación de evento de detección de cambios en la línea temporal.
Fuente: Elaboración Propia, (2016)

Tras la selección de intervalo dentro del cual se desea que el sistema muestre los estados recibidos, se implementó un servicio REST, donde mediante una consulta del tipo POST con parámetros fecha inicial y final, retornase una lista con todos los marcadores encontrados.

Las categorías mencionadas a continuación son descritas en la sección 4.3.7. Los iconos correspondientes a las categorías que soporta el programa se definieron mediante la combinación de dos imágenes para cada categoría: un marcador de mapa, similar a los definidos en la API de Google Maps y una que sugiriera al usuario el tipo al cual se refiere. Los diseños finales son presentados en las Figuras de la 4.11 a la 4.16.

Figura 4.11: Icono categoría agua.
Fuente: Elaboración Propia, (2016)

Figura 4.12: Icono categoría alimento.
Fuente: Elaboración Propia, (2016)

Figura 4.13: Icono electricidad agua.
Fuente: Elaboración Propia, (2016)

Figura 4.14: Icono categoría comunicacion.
Fuente: Elaboración Propia, (2016)

Figura 4.15: Icono categoría personas.
Fuente: Elaboración Propia, (2016)

Figura 4.16: Icono categoría seguridad.
Fuente: Elaboración Propia, (2016)

Se consideró apropiado, además, diseñar un icono que representara la densidad de marcadores al momento de realizar el agrupamiento por categorías descrito en ésta sección para ello y siguiendo la combinación de colores utilizada por la biblioteca *MarkerClusterer*, Inc (2014), donde se muestra un cluster azul cuando es un cluster pequeño; amarillo para uno medio y rojo para uno grande. El tamaño de cada uno de estos es especificado internamente por la biblioteca:

- Azul: De dos a diez elementos.
- Amarillo: De once a cien elementos amarillo.
- Rojo: Desde cien elementos.

Se prepararon, entonces, tres iconos adicionales a cada categoría para reemplazar los íconos por defecto de la biblioteca, las que pueden verse en las Figuras 4.17. a la 4.34.

Figura 4.17: Icono categoría agua para cluster pequeño.
Fuente: Elaboración Propia, (2016)

Figura 4.18: Icono categoría agua para cluster medio.
Fuente: Elaboración Propia, (2016)

Figura 4.19: Icono categoría agua para cluster grande.
Fuente: Elaboración Propia, (2016)

Figura 4.20: Icono categoría alimento para cluster pequeño.
Fuente: Elaboración Propia, (2016)

Figura 4.21: Icono categoría alimento para cluster medio.
Fuente: Elaboración Propia, (2016)

Figura 4.22: Icono categoría alimento para cluster grande.
Fuente: Elaboración Propia, (2016)

Figura 4.23: Icono categoría electricidad para cluster pequeño.
Fuente: Elaboración Propia, (2016)

Figura 4.24: Icono categoría electricidad para cluster medio.
Fuente: Elaboración Propia, (2016)

Figura 4.25: Icono categoría electricidad para cluster grande.
Fuente: Elaboración Propia, (2016)

Figura 4.26: Icono categoría comunicación para cluster pequeño.
Fuente: Elaboración Propia, (2016)

Figura 4.27: Icono categoría comunicación para cluster medio.
Fuente: Elaboración Propia, (2016)

Figura 4.28: Icono categoría comunicación para cluster grande.
Fuente: Elaboración Propia, (2016)

Figura 4.29: Icono categoría personas para cluster pequeño.
Fuente: Elaboración Propia, (2016)

Figura 4.30: Icono categoría personas para cluster medio.
Fuente: Elaboración Propia, (2016)

Figura 4.31: Icono categoría personas para cluster grande.
Fuente: Elaboración Propia, (2016)

Figura 4.32: Icono categoría seguridad para cluster pequeño.
Fuente: Elaboración Propia, (2016)

Figura 4.33: Icono categoría seguridad para cluster medio.
Fuente: Elaboración Propia, (2016)

Figura 4.34: Icono categoría seguridad para cluster grande.
Fuente: Elaboración Propia, (2016)

Dado que se solicitó que la interfaz no se recargue cada vez que se produzca un cambio dado por un nuevo evento detectado o el modificación en el intervalo de visualización. Para ello se utilizaron las facultados de Javascript y AJAX capturando los cambios en la línea temporal, descrita en esta sección. Cada vez que se detecte un cambio, se elimina todo marcador del mapa y se reubican en el todos los que cumplan con los parámetros de búsqueda.

Estadísticas de procesamiento

Específicamente se solicitaron tres tipos de estadísticas que han de ser mostradas por consulta, estas se definen a continuación:

1. Cantidad de eventos detectados, es decir, *tweets* que fueron clasificados.
2. Cantidad de usuarios distintos identificados en aquellos eventos.
3. Cantidad total de *tweets* que han pasado por el sistema desde el inicio de la consulta actual.

Para cumplir lo solicitado hacía falta añadir elementos no considerados en la base de datos; hace falta conocer al usuario y contar los *tweets* ingresados desde *Twitter4J*.

Para completar esta historia se realizaron modificaciones al esquema previamente definido en la sección 4.3.2, este de por si era suficiente para cumplir con la estadística número uno, pero incapaz de realizar las otras dos. Para la segunda estadística se consideró que bastaba con guardar al usuario junto con la colección de marcadores. De acuerdo a Dev.twitter.com (2016) en su sección F. *Be a Good Partner to Twitter*, se insta a los desarrolladores que almacenen contenido *offline* de *Twitter*, a almacenar sólo el ID del usuario o del *tweet*, por ello y siguiendo

estos lineamientos se agrega el campo ”userID” al esquema marcadores, pasando a quedar como se aprecia en la Figura 4.35.

```
{
    "_id": objectID("_MongoDB_ID"),
    "contenido": "Contenido del tweet",
    "categoría": "Categoría del tweet",
    "latitud": "Coordenada Latitud",
    "longitud": "Coordenada Longitud",
    "userID": "Identificación del usuario en Twitter",
    "generatedAt": ISODate("YYYY-mm-hhTHH:mm:sssZ")
}
```

Figura 4.35: Ejemplo de documento en la colección Markers.
Fuente: Elaboración Propia, (2016)

Para la tercera estadística la colección de marcadores no sería útil, pues no refleja la cantidad de tweets procesados, para ello es necesario implementar una tercera colección de documentos en la base de datos y almacenarlos antes de la aplicación de cualquier tipo de filtro. Esta colección tiene el esquema presente en la Figura 4.36.

```
{
    "_id": ObjectId("MongoDB_ID"),
    "tweetText": "Contenido del Tweet",
    "timestamp": ISODate("YYYY-mm-ddTHH:mm:sssZ")
}
```

Figura 4.36: Ejemplo de documento en la colección Status.
Fuente: Elaboración Propia, (2016)

Al almacenar sólo el contenido del texto no viola las políticas de uso descritas de *Twitter*, sólo es necesario la fecha para la estadística realizar la estadística, pero resulta útil almacenar el contenido para realizar la expansión de la consulta descrita en la sección 4.4.2 y no aumentar la latencia almacenando el ID y realizando una nueva consulta a la API de *Twitter*.

En general la obtención de estas estadísticas se da utilizando el Algoritmo 4.2 descrito a continuación.

Algoritmo 4.2: Algoritmos de generación de primera y tercera estadística.

Entrada: Colección c

Entrada: Fecha de la consulta actual f

Salida: Contador de eventos $counter$

$counter = 0$

Para Documento d_i en la colección c **Hacer:**

Si fecha de d_i es posterior a f **entonces:**

$counter = counter + 1$

Fin Si

Fin Para

Retornar $counter$

Este algoritmo, como se mencionó, es de uso general y permite cumplir tanto la primera como la tercera estadística, para el caso de la segunda se requiere una modificación, pues se solicitó conocer los usuarios diferentes, el algoritmo 4.3 presenta el algoritmo modificado para la segunda estadística.

Algoritmo 4.3: Algoritmos de generación de segunda estadísticas.

Entrada: Colección c

Entrada: Fecha de la consulta actual f

Salida: Lista de usuarios vacía $list$

Para Documento d_i en la colección c **Hacer:**

Si fecha de d_i es posterior a f **entonces:**

Si ID del usuario de d_i no está en $list$ o $list$ es vacía **entonces:**

Añadir d_i a $list$

Fin Si

Fin Si

Fin Para

Retornar Cantidad de elementos en $list$

Adicionalmente a lo anteriormente descrito hace falta un medio para obtener los datos, para ello se implementó un servicio REST donde utilizando un método GET se obtienen todos los datos correspondientes a la actual consulta activa en el sistema, para conocer cuál fue la última consulta del sistema se utiliza la clase *CurrentQueryChecker*, la cual provee de un método para obtener la fecha de la última consulta.

Configuración

Esta historia nace producto de la HU-v01. El visualizador de eventos tiene dos maneras de comportarse:

- Modo tiempo real: Cuando el sistema esté en funcionamiento y cada cierto tiempo, t_1, se actualizan los marcadores de los nuevos eventos y éstos se muestran durante un tiempo, t_2.
- Modo línea de tiempo: Funcionamiento basado en lo descrito en HU-v04.

Los tiempos t_1 y t_2, inicialmente fueron decididos de manera arbitraria, pero al mostrar su funcionamiento se sugirió que estos parámetros fuesen modificados, por ello, se implementó una sección de configuración dentro de la aplicación de visualización para permitir el cambio de estos valores.

4.4.2 Detector de necesidades

El detector de necesidades, al estar construido sobre *Apache Storm*, está compuesto de múltiples operadores que trabajan en conjunto.

Entrada de datos al sistema

Conociendo desde donde se obtiene la información y teniendo acceso a ella resta conocer cómo realizar la conexión. Para ello se selecciono utilizar *Twitter4J*, una biblioteca no oficial de Java para las API de *Twitter*. Para su funcionamiento sólo requiere del uso de Java en su version 5 o superior.

La implementación de lo anteriormente descrito se realiza utilizando una instancia del objeto *TwitterStream*, el cual captura el flujo público de *Twitter*, almacenando cada estado recibido en una cola. Con esto en mente se construyó el primer operador del sistema correspondiente al *Spout* que surte de datos al sistema.

```java
@Override
public void nextTuple() {
     //Mientras la cola no esté vacía: Busy waiting.
     while(queue.isEmpty()){
         Utils.sleep(50);
     }
     Status status = queue.poll();

     if (status == null) {
        Utils.sleep(50);
     } else {
         /*
              Guarde los ID y fecha de recepción del estado,
              luego emite a la topología.
         */
        statusPersistence.saveStatus(status);
        _collector.emit(new Values(status));
     }
}
```

Figura 4.37: Implementación del *Spout* del sistema.
Fuente: Elaboración Propia, (2016)

Considerando lo recién expuesto la Figura 4.37 muestra cómo los estados son emitidos por el *spout* al sistema basándose en la cola (*queue*) para manejar lo que llega desde el *stream*

Tal y como se mencionó en el capítulo 2 una topología *storm* funciona estableciendo un grafo donde cada nodo corresponde a un operador o *bolt*, la Figura 2.2 muestra como los *bolt* se unen con el propósito de procesar las entradas entregadas por los *spout* que, en este caso y tal como se señaló en el sección 4.4.2 corresponde a aquel que recibe la información desde el *stream* de *Twitter*, pero ¿cuál es la función de cada uno de los *bolts*?

La pregunta anteriormente planteada abre un nuevo abanico de problemas, para cada uno de los cuales se desarrolla un operador y éstos, trabajando en conjunto, producen la información necesaria para ser almacenada como un documento 'marcador' en la colección 'Markers' descrita en la Figura 4.35.

Operador idioma

La primera de las problemáticas a tratar es el idioma. De acuerdo a Statista (2016), existen actualmente 310 millones de usuarios activos en *Twitter* (a enero del 2016), de los cuales 65 millones pertenecen a los Estados Unidos según Smith (2016), y se estima que este año, en latinoamérica, Brasil alcance los 15 millones de usuarios según eMarketer (2015), sin considerar

paises árabes o asiáticos ya se tiene cerca del 30% de los usuarios activos de *Twitter* hablan, en general, idiomas distintos al Español. Dado que el sistema está pensado para operar dentro de Chile donde el idioma oficial es el Español hace necesario que uno de los operadores, el primero, se el filtrado por idioma, ¿Por qué el primero? Para no realizar procesamiento innecesario con datos que no se han de utilizar.

Nakatani (2010) desarrolló, haciendo uso de un clasificador *Naïve Bayes*, un módulo escrito en Java el cual es capaz de detectar con éxito 49 idiomas dentro del texto con un 99.8% de precisión y fue la primera opción para resolver el problema del idioma, pero al analizar el cuerpo de un estado de *Twitter* se encontró que uno de sus campos, precisamente, correspondía al idioma en el que estaba escrito el *tweet*, por ello, y con objeto de no realizar cálculos innecesarios, se optó por utilizar este campo.

La Figura 4.38 presenta el código de la implementación de este *bolt*, correspondiente a su método *execute*, descrito en la sección 1.5.2.

```
1. @Override
2. public void execute(Tuple tuple) {
3.     Status status = (Status) tuple.getValueByField("status");
4.     if(status.getLang().equals("es"))
5.     {
6.         this.collector.emit(new Values(status));
7.     }
8. }
```

Figura 4.38: Implementación del método *execute* del *bolt* de idioma.
Fuente: Elaboración Propia, (2016)

Aunque simple, éste operador filtra un gran número de estados, dado que según lo dicho anteriormente, la mayoría de los usuarios de *Twitter* no son hispano-hablantes.

Operador filtro de consultas

El segundo problema corresponde a que aunque se tengan los mensajes en el idioma correcto existen mensajes que el usuario desea que sean priorizados, para ello, y como se definió en la sección 4.3.5, son entregados términos de búsqueda. Es así como el segundo nivel de operadores corresponda a aplicar los filtros de búsqueda ingresados por el usuario, si existiesen, aplicando el algoritmo descrito anteriormente y discriminando aquellos estados que contengan las palabras buscadas.

Adicionalmente, la historia HU-c02 que guarda relación con la HU-v05, menciona la necesidad de incrementar los términos de búsqueda para enriquecerla y abarcar la mayor información posible dado una consulta. Para realizar esto se consideró una práctica del

procesamiento de lenguaje natural como es la denominada *Query Expansion* (QE). Según lo descrito por Manning et al. (2008) son técnicas comunes al utilizar QE la búsqueda de sinónimos (uso de diccionarios priviamente establecidos), diccionarios basados en la minería de los elementos previamente hayados, creación de diccionadios basados en la co-ocurrencia de términos, es decir, términos que suelen venir juntos o un vocabulario mantenido por editores humanos. Para este trabajo sólo se consideran las dos primeras: Búsqueda por diccionario de sinónimos y una implementación que encuentra los términos más frecuentes dentro de los resultados de la búsqueda.

El diccionario de sinónimos es básicamente una bolsa de palabras asociadas a una semilla, es decir, dado un término de búsqueda agregar todos los términos asociados a el en el diccionario.

En el caso de la búsqueda de términos frecuentes se sugirió integrar un proyecto *storm* ya desarrollado el cuál tiene por finalidad la búsqueda de los denominados *threnting topics*, es decir, aquellos términos de los que se realizan más menciones en un determinado instante, pero aquella implementación sólo consideraba los denominados *hashtag*, un marcador de palabras concatenadas que inician por el caracter ”#”. Siendo ese el caso el uso de esta topología storm no es del todo util. En su lugar se desarrolla un contador de frecuencias para palabras con un funcionamiento similar, dicha implementación se aprecia en el Algoritmo 4.4.

Algoritmo 4.4: Algoritmos de términos recurrentes.

Entrada: Estados $E = \{e_1, \ldots, e_n\}$.

Salida: Terminos frecuentados $T = \{t_1, \ldots, t_{10}\}$.

Lista de terminos: l.

Para Estado: e_i **Hacer:**

Dividir estado por palabra.

Eliminar *stopword* de las palabras.

Para Palabra: w_i en e_i **Hacer:**

Si w_i está en l_i **entonces:**

aumentar contador de w_i en l_i.

Sino:

agregar w_i a l_i con contador en 1.

Fin Si

Fin Para

Fin Para

Si l_i tiene menos de 10 elementos **entonces:**

Retornar l_i

Sino:

Retornar los 10 primeros elementos de l_i.

Fin Si

Dado que el operador puede estar replicado no se reciben los mismos estados a todas las instancias del nodo, por ello este proceso se realiza de manera única para cada instancia en función de los estados que hayan llegado a él. El Algoritmo 4.4 agrega a los términos de búsqueda de cada instancia las palabras más frecuentes y, siguiendo el ejemplo de *Twitter* con sus *thrending topics* tiene un máximo de diez nuevas palabras.

```
1.  @Override
2.  public void execute(Tuple tuple) {
3.      Status status = (Status) tuple.getValueByField("status");
4.      CurrentQueryChecker cqc = new CurrentQueryChecker();
5.      /*Revisa la última query*/
6.      cqc.check();
7.      if(this.checkQueryMatch(queryExpander.expandQuery(cqc), status)){
8.          this.collector.emit(new Values(status));
9.      }
10. }
```

Figura 4.39: Implementación del método *execute* del *bolt* del filtro de consultas.
Fuente: Elaboración Propia, (2016)

La Figura 4.39 muestra la implementación del filtro de consultas. Hace uso de instancias de los objetos descritos en HU-v05 para encontrar la última consulta en el sistema y expande la consulta según ésta y los resultados obtenidos en los estados recibidos. Finalmente y si el estado contiene algunos los términos especificados, éste es emitido al siguiente nivel de operadores.

Operador normalizador de texto

El tercer problema es inherente a *Twitter*: En esta red social es común referenciar un estado a un determinado tema, he ahí el uso de los conocidos *Hashtag* que, como se mencionó en la sección 4.4.1 corresponden a palabras concatenadas antecedidas por el caracter #. Otro problema común corresponde a la mención de usuarios, ésta corresponde a un llamado al nombre de usuario dentro de la aplicación, antecedida por el caracter ”@”, usualmente usada para el envío de mensajes entre pares. Diversos autores, entre ellos, Lynn et al. (2015), Arshi Saloot et al. (2015) y Bonzanini (2015), han señalado que la existencia de estos elementos significan una disminución en la precisión de los elementos descritos en la sección 4.3.8. Por ello él tercer operador corresponde a normalizador de texto, el cual reemplaza menciones a usuarios, *hashtags* y URLs, todas ellas variables, por palabras constantes, el reemplazo a realizarse se muestra en la Tabla 4.2.

Tabla 4.2: Reemplazo de entidades en texto.
Fuente: Elaboración Propia, (2016)

Entidad	**Reemplazo**
@usuario	USUARIO
#hashtag	HASHTAG
http://var.foo/	URL

La implementación de éste operador se realizo utilizando expresiones regulares para detectar cuándo se está haciendo referencia a uno de los elementos anteriores y luego aplicar su reemplazo.

Operador geolocalizador

El cuarto y mayor problema presentado tiene relación, principalmente, con la historia HU-v01. Si bien se mencionó cómo se realiza la visualización, no se señaló cómo es que se obtienen tanto la coordenadas geográficas, latitud y longitud, para ubicar geográficamente un evento.

Ha sido señalado por Imran et al. (2014a) que menos del 1% de los *tweets* contienen datos en sus campos correspondientes a geolocalización. En un experimento (véase Apéndice a.) realizado utilizando la herramienta *RapidMiner* se obtuvo una muestra de 67.789 *tweets*

directamente desde el *stream* sin utilizar filtros de búsqueda, de esos *tweets* 67.475 no contaban con los datos correspondientes a la ubicación geográfica, es decir, el 0.46% de los datos de aquella muestra cuentan con la información requerida, lo que hace creer que lo presentado por los autores, antes mencionados, está en lo correcto.

Siendo la geolocalización un elemento de suma importancia para el funcionamiento de la aplicación, se ha de intentar obtener este dato de alguna forma. Así es como surge la posibilidad de usar el contenido del *tweet* para obtener la ubicación, para ello se preparó un diccionario con las comunas del país y sus coordenadas geográficas, Carta-natal (2016) y se diseñó el Algoritmo 4.5. De esta manera existe una aproximación para detectar la ubicación a la que un *tweet* hace referencia.

Algoritmo 4.5: Algoritmos de ubicación geoográfica.

Entrada: Lista de ciudades $C = \{c_1, \ldots, c_n\}$.

Entrada: Tweet t.

Salida: Coordenadas geográficas $P = \{latitud, longitud\}$.

Si t está geolocalizado **entonces:**

Si Está dentro del territorio chileno **entonces:**

Retornar Coordenadas del t.

Sino:

Retornar Fuera de Chile.

Fin Si

Sino:

Si El texto de t contiene elementos presentes en C **entonces:**

Retornar Coordenadas de c_i.

Sino:

Retornar No geolocalizable.

Fin Si

Fin Si

Para detectar cuándo una ubicación está en Chile, se generó un cuadro en el mapa donde se delimita todo el territorio Chileno, incluyendo Isla de Pascua.

Haciendo uso del algoritmo desarrollado es posible aumentar la cantidad de elementos que continúan en la línea de procesamiento. En el capítulo 5 realiza una evaluación sobre la efectividad del operador.

Los operadores descritos en las secciones 4.4.2 y 4.4.2 tienen relación con la labor del señalado en 4.4.2, las razones que llevan a la construcción de estos tres son expuestan en la sección 4.4.2, mientras tanto, al igual que los operadores anteriores se detalla el cómo fueron

diseñados.

Operador removedor de stopword

Éste operador hace uso de una lista de palabras denominadas *stopwords* o palabras vacías, éstas corresponden a palabras sin significado, como artículos, pronombres, preposiciones, etcétera. Éstas palabras son eliminadas del texto que se está procesando, para ello se utiliza el Algoritmo 4.6.

Algoritmo 4.6: Algoritmos de eliminiación de *stopwords*.

Entrada: Lista de *stopwords* $S = \{s_1, \ldots, s_n\}$.

Entrada: Texto T.

Salida: Texto T'.

T' = T

Para cada palabra de T, t_i **Hacer:**

Si t_i está contenida en S **entonces:**

T' = T' - t_i.

Fin Si

Fin Para

Retornar T'

Operador raíz de texto

Este operador hace uso del algoritmo de Porter (1979), para extraer prefijos y sufijos de palabras y llevarlas a una raíz común, Ramírez (2012), son ejemplos de este proceso, denominada *stemming*, las palabras presentadas en la Tabla 4.3

Tabla 4.3: Ejemplo de *stemming* para la palabra 'presentar'.
Fuente: Elaboración Propia, (2016)

Palabra	Combinaciones de Sufijos
Presentarla	arla
Presentarlas	arlas
Presentarle	arle
Presentarles	arles
Presentarlo	arlo
Presentarlos	arlos
Presentarse	arse
Presentase	ase
Presentásemos	ásemos
Presente	e
Presentémonos	émonos

Operador etiquetador

Este operador hace uso del clasificador generado mediante las técnicas descritas en la sección 4.3.7, para etiquetar el texto de acuerdo a la categoría a la que corresponda. Una vez realizado este proceso se cuenta con todos los datos necesarios para generar completamente un documento de la colección 'Markers', presentada en la Figura 4.35.

Operador persistencia

Este operador se encarga de conectarse a la base de datos y almacenar el nuevo marcador. Los datos recibidos desde la cadena de procesamiento se lleva a una instancia de objeto Java, llamado "Marker", el cual contiene los mismos elementos descritos para un documento de la colección "Markers", a un objeto JSON utilizando *Jackson* y finalmente, utilizando *Jongo*, lo transforma en BSON para almacenarlo en la base de datos en MongoDB.

Construcción del clasificador

Hasta ahora se han tocado, prácticamente, todos los temas que se relacionan con el funcionamiento del sistema de detección, desde donde se obtienen los datos, cómo se procesan e incluso cómo se almacenan, pero no se ha especificado cómo se realiza la clasificación, es decir, cómo dado un texto de entrada se consigue discriminar en qué categoría encaja. Ésta sección especifica lo que se contextualizó en la sección 4.4.2.

El concepto que involucra la construcción de un clasificador es el de "aprendizaje supervizado", que fue mencionado en la sección **??**, en este tipo de aprendizaje se requiere de un conjunto de datos de entrada, denominados conjunto de entrenamiento, que ha de pasar por el algoritmo, en este caso *Naïve Bayes*, que ha de conocer previamente, la salida esperada para cada elemento del conjunto. El resultado esperado es un clasificador capaz de predecir, en este caso, a qué categoría pertenece un texto sometido a su evaluación.

Según la metodología KDD existen subprocesos en la búsqueda de conocimiento en bases de datos, estos fueron descritos en la sección 1.5.1, para este caso particular se describe cómo fue realizado cada uno de estos subprocesos para la construcción del clasificador con el que cuenta el sistema.

El subproceso de selección de datos se llevó a cabo extrayendo un subconjunto del *dataset* mencionado en la sección 1.4.3. Éste conjunto, de exactamente 2234 *tweets* correspondientes al del terremoto de Concepción el año 2010, todos ellos en español, pero no todos hacen referencia al evento, pues éste coincidió con la realización de la LI versión del Festival Internacional de la Canción de Viña del Mar y muchos de estos *tweets* hacen referencia a este último. Los datos en este punto de proceso cuentan con los campos correspondientes a un *tweet*

de la época, es decir: ”ID_unit”, ”day”, ”date”, ”time_zone”, ”time”, ”tweet_it”, ”user_id”, ”name”, ”screen_name”, ”friends_count”, ”follower_count”, ”text” y ”value”. Todos ellos separados por coma (,).

En cuanto al subproceso de preprocesamiento de datos, el primer paso corresponde a la limpieza de los datos, en conjunto de datos que se está utilizando contenía elementos incompletos que no presentaban texto, estos fueron eliminados, pues no resultaban útiles sin el componente texto, quedando así un total de 2187 *tweets* con datos útiles.

El siguiente paso, correspondiente al suproceso de transformación de datos, del conjunto de datos útiles se extrajo el texto y se eliminaron todos los demás componentes. Llegados a este punto sólo se contaba con una lista de textos de *tweets* en español. Lo siguiente a realizar corresponde al proceso de etiquetado, para ello se leyó cada texto y según su contenido se ubicó en alguna de las categorías descritas en la sección 4.3.7, además se asignó un identificador a cada texto, basado en su número, para ser ingresados a la herramienta Mallet, encargada de la construcción del clasificador, así se obtuvo un archivo con los datos formateados según lo descrito en la sección 4.3.8, es decir, con los campos ”Identificador”, ”Etiqueta”, ”Contenido”. En este punto los datos fueron ingresados al sistema para la aplicación de los operadores descritos en la sección 4.4.2, en particular la eliminación de *stopwords*, normalización de texto y *stemming*.

El cuarto subproceso es automatizado por Mallet y corresponde al minado de datos en sí, para entregarle los datos a Mallet primero han de construirse como un objeto ”*Instance*”, lo cual se realiza entregándole cada uno de los elementos preparados en la sección anterior: el identificador, etiqueta y contenido que ya ha pasado por las operaciones que componene el subproceso de transformación. Como resultado de este proceso se obtiene un objeto *Classifier*, el cual puede ser serializado y almacenado como un archivo denominado, en este caso, ”classifier.dene”.

Finalmente el subproceso de evaluación también es llevado a cabo por la herramienta Mallet, que implementa un objeto denominado ”*Trial*” el cual, utilizando un clasificador y un conjunto de datos de prueba. Como resultado de este subproceso se obtienen las métricas comunes de evaluación: *accuracy*, *recall* y *F-1 score*. La evaluación del clasificador construido se presenta en la sección 5.2 del Capítulo 5.2.

Topología del sistema

Habiendo definido los elementos de procesamiento, los operadores o *bolts*, se está en condicion de definir la topología. La topología que utiliza el sistema, en términos generales de la aplicación está definida en la Figura 4.40.

La razón de este orden en la topología se debe a varias razones y se justifican a continuación:

- *Spout Twitter*: Es el eslabón principal de la cadena. Desde aquí se emiten los nuevos estados al sistema y todo el sistema depende de el.
- *Bolt* Filtro de idioma: Ocupa la primera posición de los operadores del sistema, dado que se espera que el *stream* reciba estados de todo el mundo y no sólo en español. Al estar este operador en primer lugar se asegura de reducir el flujo en gran medida, lo cual puede comprobarse por los resultados obtenidos en el Capítulo 5.
- *Bolt* Filtro de consulta: Habiendo filtrado sólo aquellos estados cuyo lenguage sea el español es necesario filtrar aun más el *stream* valiéndose de las restricciones especificadas por el usuario. Así sólo los estados que contengan términos especificados por el usuario, o el sistema de expansión, continúan en el sistema.
- *Bolt* Normalizador de texto: Previo al detector de ubicación para evitar posibles confusiones que pueda acarrear la existencia de nombres de lugares en elementos como nombres de usuario, *hashtags* o enlaces.
- *Bolt* Detección de ubicación: Ocupa ésta posición, pues debe ir previo a la eliminación de *stopwords*, de lo contrario ubicaciones, como por ejemplo ”los vilos”, válida dentro de chile, es ignorada por el sistema.
- *Bolt* Eliminador de *stopword*: Se realiza previo al *Stemming* para reducir la carga computacional, pues el operador de *stemming* lleva a palabras raíz estos términos que no son necesarios.
- *Bolt Stemmer*: Es la única ubicación posible para este operador, pues el siguiente paso es etiquetar el estado.
- *Bolt* Etiquetador: Aplica el modelo al estado, el estado ha de tener su correspondiente etiqueta antes de ser almacenado.
- *Bolt* Persistencia: último eslabón de la cadena. Ingresa un nuevo documento a la colección de marcadores.

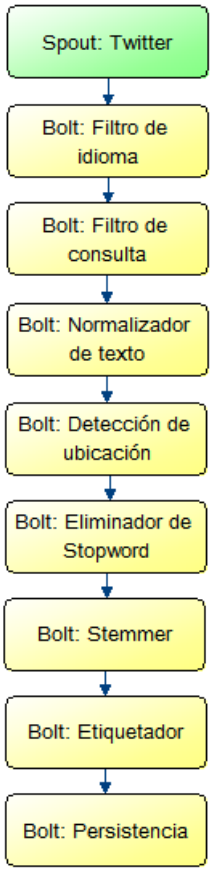

Figura 4.40: Topología general del sistema.
Fuente: Elaboración Propia, (2016)

CAPÍTULO 5. EVALUACIÓN DEL SISTEMA

Con el sistema construido resta someterlo a evaluaciones. En especial se evalúa el sistema de detección de necesidades, pues es el corazón del sistema y ha de operar con un flujo de datos constante.

5.1 CUMPLIMIENTO DE REQUERIMIENTOS

En cuanto a la construcción del *software*, se identificador 12 historias de usuario y se especificaron cada uno de sus criterios de aceptación. Se pasa a detallar por cada historia si ésta fue cumplida o no.

- HU-c00: Mediante la construcción de un operador, *spout*, que haciendo uso del *stream* de datos proporcionado por *Twitter* recogiese los datos en tiempo real y la contrucción de un clasificador de texto es que se completo esta historia.
- HU-c01: De igual manera que la historia anterior, haciendo uso de la información de *Twitter*, ésta historia se completó.
- HU-c02: Esta historia se completó mediante la construcción del operador filtro de consulta, donde se añadió la capacidad de realizar *query expansion*.
- HU-v00: Con la contrucción de la aplicación visualizadora, la que permite el despliegue de una interfaz web al usuario esta historia se marcó como completada.
- HU-v01: Mediante el operador de ubicación ésta historia de usuario se completó.
- HU-v02: Se implementaron filtros haciendo uso de Javascript a la interfaz, permitiendo realizar veintiún formas de visualización diferentes.
- HU-v03: Al igual que la historia anterior, se utilizó Javascript, en específico AJAX, para que, mediante un servicio REST se pudiese actualizar los nuevos marcadores.
- HU-v04: Se utilizó una librería externa, *HighCharts*, para la implementación de una línea de tiempo con intervalo deslizante y, mediante un servicio REST, obtener los datos pertenecientes al intervalo seleccionado y cumplir así ésta historia de usuario.
- HU-v05: Se implementó junto con la historia de usuario HU-c02. El filtro de consultas permitía el paso de los estados que tuviesen parte de su contenido alguno de los términos especificados por el usuario.

- HU-v06: Se diseñaron 7 iconos para corresponder a cada una de las categorías y completar esta historia, la descripción de estos es entregada por medio de la aplicación de visualización al usuario final.
- HU-v07: Mediante la construcción de tres servicios rest que cuenten la cantidad de eventos desde la última consulta se completó esta historia de usuario.
- HU-v08: Se permitió al usuario el parametrizar la configuración de la aplicación visualizadora, completando así esta historia de usuario.

5.2 EVALUACIÓN DEL CLASIFICADOR

Las métricas resultados de la construcción del clasificador en la sección 4.4.2, correspondientes al clasificador actual se presentan a continuación en la Figura 5.1

Clase	Accuracy (%)	Recall (%)	F-1 Score
Agua	87,11	8,00	14,81
Electricidad		5,00	9,52
Alimento		0,00	0,00
Comunicación		15,79	26,86
Seguridad		36,00	52,83
Personas		65,58	77,09
Irrelevante		99,47	92,50

Figura 5.1: Métricas del clasificador.
Fuente: Elaboración Propia, (2016)

Los valores expuestos anteriormente son interpretados, respectivamente, como sigue a continuación.

- *Accuracy*: Este valor quiere decir qué con, aproximadamente, un 87% al decir que un elemento pertenece a una determinada clase esa predicción es correcta.
- *Recall*: Este valor quiere decir que para una clase en particular, es posible identificar es posible identificar en un determinado porcentaje p, correspondiente al valor presentado en la Figura 5.1, de los elementos pertenecientes a aquella clase.
- *F-1 Score*: Corresponde al *trade-off* entre *accuracy* y *recall*, al incrementar uno, el otro disminuye en un F% descrito por los valores en la Figura 5.1.

En particular este clasificador cuenta con alta precisión y bajo *recall*, eso significa que el es preciso para clasificar, pero no es capaz de clasificar algunos de los casos particulares de

cada clase. Esto se debe al conjunto de entrenamiento utilizado, los datos no están balanceados para cada clase, es decir, la clase A, no tiene los mismos elementos que la clase B, ésto repercute en que debido a las pocas instancias que tiene el algoritmo para aprender no es capaz de reconocer elementos de la clase con menos elementos. En particular el caso de la clase ”Alimentos”, los datos que se utilizaron son del periodo inmediato al evento, por ello no se encontraron demasiados elementos que hagan referencia a la falta de alimento en una población.

5.3 TOPOLOGÍA Y REPLICACIÓN

En la sección 4.4.2 se explicitó cómo están dispuestos los operadores en la topología, pero se ha de recordar que el sistema está pensado para operar en casos de emergencia y ha de ser capáz de escalar de acuerdo a las necesidades de la situación.

Apache Storm es capaz de realizar lo anterior, pero se ha de especificar el máximo número de nodos que tiene cada nivel de operadores. Para explicar lo anterior se utilizan las Figuras 5.2 y 5.3 que muestran la implementación del la topología y una esquematización de cómo se comporta en la peor situación, es decir, cuando el sistema determine que el nivel de replicación debe ser máximo.

```
1.  public static void main(String[] args) {
2.      TopologyBuilder builder = new TopologyBuilder();
3.
4.      builder.setSpout("TwitterSpout", new TwitterSpout(), 2);
5.
6.      builder.setBolt("LanguageFilter", new LanguageFilter(), 4).shuffleGrouping("TwitterSpout");
7.      builder.setBolt("QueryFilter", new QueryFilter(), 4).shuffleGrouping("LanguageFilter");
8.      builder.setBolt("TextNormalizer", new TextNormalizer(), 4).shuffleGrouping("QueryFilter");
9.      builder.setBolt("LocationRecognizer", new LocationRecognizer(),
    4).shuffleGrouping("TextNormalizer");
10.     builder.setBolt("StopwordRemover", new StopwordRemover(),
    2).shuffleGrouping("LocationRecognizer");
11.     builder.setBolt("TextStemmer", new TextStemmer(), 2).shuffleGrouping("StopwordRemover");
12.     builder.setBolt("Labeler", new Labeler(), 2).shuffleGrouping("TextStemmer");
13.     builder.setBolt("Persistence", new Persistence(), 1).shuffleGrouping("Labeler");
14.
15.     Config conf = new Config();
16.     conf.setDebug(false);
17.
18.     LocalCluster cluster = new LocalCluster();
19.     cluster.submitTopology("Deteccion-Necesidades", conf, builder.createTopology());
20. }
```

Figura 5.2: Implementación topología de detección de necesidades.
Fuente: Elaboración Propia, (2016)

Cada elemento de procesamiento, *bolt*, es instanciado, el valor que acompaña a cada uno de ellos es el numero máximo de nodos que tiene el sistema y seguido del modo de agrupamiento, en este caso, *shuffle grouping* para balancear la carga en cada nodo.

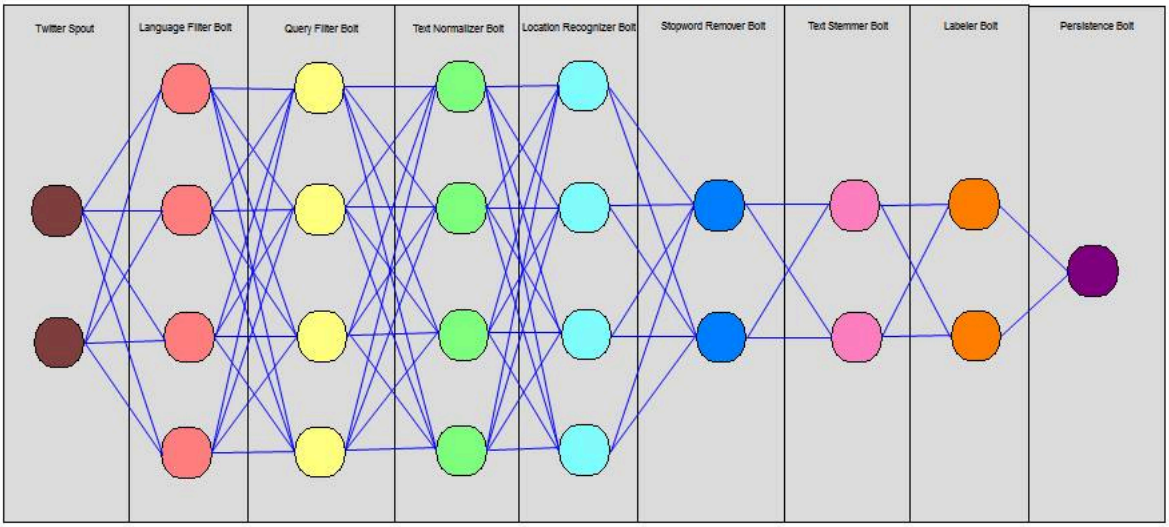

Figura 5.3: Esquema de la topología en el caso de máxima actividad.
Fuente: Elaboración Propia, (2016)

Cada línea de este esquema señala comunicación de izquierda a derecha. En el caso de que el sistema trabaje al máximo de su capacidad cada nodo envía, *round robin*, estados al siguiente nivel.

Esta implementación y diagrama reflejan la solución inicial la cual fue decidida arbitrariamente para probar el sistema.

Tempranamente se detectó que el hecho de tener dos *spout* resultaba contraproducente, pues enviaba, en repetidas ocaciones, el mismo estado al sistema, es decir, cuando el *spout* A enviaba el estado e_0, probablemente el *spout* B enviase el mismo estado e_0. Por ello se decidió eliminar el segundo *spout* y limitarlo sólo a uno.

Se utilizó el tiempo de ejecución para 1000, 2000, 4000 y 8000 estados, pertenecientes al terremoto de Concepción el año 2010 para seleccionar cuán numeroso debería ser un nivel de nodos. Sus resultados son expuestos en la tabla 5.1.

Tabla 5.1: Estadísticas de los operadores diversos estados.
Fuente: Elaboración Propia, (2016)

Entradas (estados)	Métricas	Operadores			
		Idioma	Normalizador	Ubicación	Stopword
1000	Procesados	1000	1000	1000	1000
	Emitidos	402 (40.20%)	1000 (100%)	623 (62.30%)	1000 (100%)
	Descartados	598 (59.80%)	0 (0%)	377 (37.70%)	0 (0%)
	Tiempo (ms)	0.77	36,45	2111.73	30.54
2000	Procesados	2000	2000	2000	2000
	Emitidos	807 (40.35%)	2000 (100%)	1058 (52.90%)	2000 (100%)
	Descartados	1193 (59.65%)	0 (0%)	942 (47.10%)	0 (0%)
	Tiempo (ms)	0.84	56.07	1314.53	45.23
4000	Procesados	4000	4000	4000	4000
	Emitidos	1673 (41.83%)	4000 (100%)	1985 (49.63%)	4000 (100%)
	Descartados	2327 (58.17%)	0 (0%)	2015 (50.37%)	0 (0%)
	Tiempo (ms)	2.08	49.09	2155.30	71.09
8000	Procesados	8000	8000	8000	8000
	Emitidos	3101 (38.76%)	8000 (100%)	4113 (51.41%)	8000 (100%)
	Descartados	4899 (61.24%)	0 (%)	3887 (48.59%)	0 (0%)
	Tiempo (ms)	3.37	59.53	4442.00	87.24

Con estos resultados se busca definir los valores para la cantidad de nodos por cada nivel de *bolts*, de los resultados presentados en la Tabla 5.1 podemos concluir lo siguiente:

Para el caso del operador filtro de idioma, para diferentes tamaños de conjuntos de estados, es emitido, aproximadamente, un 40.28% del flujo de información que llega a aquel operador. Tiene un tiempo de ejecución reducido en comparación a los demás operadores.

En operadores normalizadores de texto y eliminación de *stopwords* todo estado que llega es emitido.

El operador de ubicación admite, aproximadamente, un 54.06% y presenta un elevado tiempo de ejecución.

Estos resultados muestran el comportamiento individual de cada uno de éstos bolts, pero su operación no es de esta forma, por ello se preparó el mismo conjunto de pruebas de 1000, 2000, 4000 y 8000 datos aleatoreos del conjunto de datos del terremoto de Concepción el año 2010 y se pasó al sistema para ver la cantidad de estados emitidos en cada caso. Para el caso del operador de consulta se utilizaron los términos ”terremoto”, ”concepción” y ”Chile”. Se restringió el paralelismo de la topología completa para realizar estas pruebas, es decir, cada operador tuvo, como máximo, una instancia trabajando durante todo el proceso.

Tabla 5.2: Prueba sistema completo utilizando 1000 estados.
Fuente: Elaboración Propia, (2016)

Cantidad	**1000**	
Operador	**Estados recibidos**	**Estados emitidos**
Spout	1000	1000
Idioma	1000	402
Consulta	402	123
Normalizador	123	123
Ubicacion	123	0
Stopword	0	0
Stemmer	0	0
Etiquetador	0	0
Persistencia	0	0

Tabla 5.3: Prueba sistema completo utilizando 2000 estados.
Fuente: Elaboración Propia, (2016)

Cantidad	**2000**	
Operador	**Estados recibidos**	**Estados emitidos**
Spout	2000	2000
Idioma	2000	807
Consulta	807	78
Normalizador	78	78
Ubicacion	78	0
Stopword	0	0
Stemmer	0	0
Etiquetador	0	0
Persistencia	0	0

Tabla 5.4: Prueba sistema completo utilizando 4000 estados.
Fuente: Elaboración Propia, (2016)

Cantidad	**4000**	
Operador	**Estados recibidos**	**Estados emitidos**
Spout	4000	4000
Idioma	4000	1673
Consulta	1673	39
Normalizador	39	39
Ubicacion	39	0
Stopword	0	0
Stemmer	0	0
Etiquetador	0	0
Persistencia	0	0

Tabla 5.5: Prueba sistema completo utilizando 8000 estados.
Fuente: Elaboración Propia, (2016)

Cantidad	**8000**	
Operador	**Estados recibidos**	**Estados emitidos**
Spout	8000	8000
Idioma	8000	3101
Consulta	3101	151
Normalizador	151	151
Ubicacion	151	0
Stopword	0	0
Stemmer	0	0
Etiquetador	0	0
Persistencia	0	0

Las Tablas 5.2, 5.3, 5.4 y 5.5 muestran respectivamente los resultados para los datos antes mencionados, estos resultados muestran la dificultad que tiene un estado para completar el proceso, pero hace presumir un error de implemenación en el operador de ubicación. Para descargar que se deba a los datos se incrementó en gran medida el número de éstos conformando los resultados entregados en la Tabla 5.6

Tabla 5.6: Prueba sistema utilizando 30000 estados.
Fuente: Elaboración Propia, (2016)

Cantidad	**30000**	
Operador	**Estados recibidos**	**Estados emitidos**
Spout	30000	30000
Idioma	30000	5372
Consulta	5372	883
Normalizador	883	883
Ubicacion	883	0
Stopword	0	0
Stemmer	0	0
Etiquetador	0	0
Persistencia	0	0

Ninguno de los datos contenía información sobre la ubicación, por ello el operador debe inferirla con respecto al texto, pero casos como por ejemplo el mostrado en la Tabla 5.7 muestran que no se está realizando la labor como debiese.

Tabla 5.7: Resultado esperado del operador de ubicación.
Fuente: Elaboración Propia, (2016)

Estado	**Resultado esperado**	**Resultado obtenido**
@tvn_mauricio x fin internet desde el movil, ciudad satelite maipu, sin luz ni agua desde el terremoto casi 30 hrs	Emitir(-33.51667, -70.76667)	No emitido

La solución a ésto fue reescribir el código del *bolt* de ubicación, pues contenía un error de evaluación dentro de las condiciones para hallar la localidad contenida en el texto, además de un problema de conexión del colector de eventos para el operador de *stopwords*. Habiendo corregido esto se volvió a realizar la última prueba correspondiente a la todos los estados obteniendo los resultados presentados en las Tabla 5.8. Cabe destacar que estos datos son influenciados por el operador consulta, mientras el filtro de ubicación sólo permite el paso de elementos que hagan referencia a ubicaciones dentro del país y el filtro de idioma sólo permite el paso de aquellos que cumplan con estar en español, el filtro de consulta está influenciado en gran medida por la decisión del usuario; mientras más términos o mientras más generales éstos sean, mayor cantidad de elementos ingresa al sistema.

Tabla 5.8: Nueva prueba al sistema con 30000 estados.
Fuente: Elaboración Propia, (2016)

Cantidad	**30000**	
Operador	**Estados recibidos**	**Estados emitidos**
Spout	30000	30000
Idioma	30000	5372
Consulta	5372	883
Normalizador	883	883
Ubicacion	883	743
Stopword	743	743
Stemmer	743	743
Etiquetador	743	743
Persistencia	743	743

Estos resultados distan de los realizados para los operadores individuales y muestran que el flujo de información en el sistema haciendo uso de datos reales. Teniendo en cuenta estos resultados se modifica la topología inicial presentada en la figura 5.3 y pasa a utilizarse la presentada en la Tabla 5.10, sus valores se basan en el porcentaje de datos capaces de pasar el operador, siendo estos representados en la Tabla 5.9.

Tabla 5.9: Nivel general de replicación de la topología.
Fuente: Elaboración Propia, (2016)

Operador	**Nivel de replicación**
Spout	1
Idioma	N
Consulta	C = $\lceil 18\% \cdot N \rceil$
Normalizador	$\lceil 16\% \cdot C \rceil$
Ubicación	U = $\lceil 16\% \cdot C \rceil$
Stopword	S = $\lceil 84.14\% \cdot U \rceil$
Stemmer	S
Etiquetador	S
Persistencia	S

Donde N es un valor arbitrario determinado por el desarrollador. Para determinar

un nivel de replicación para el sistema se determinó $N = 10$, así el sistema se entrega como el siguiente nivel de replicación máximo en sus operadores:

Tabla 5.10: Nivel máximo de replicación de la topología.
Fuente: Elaboración Propia, (2016)

Operador	Nivel de replicación
Spout	1
Idioma	10
Consulta	2
Normalizador	2
Ubicación	2
Stopword	1
Stemmer	1
Etiquetador	1
Persistencia	1

5.4 FUNCIONAMIENTO EN ALTO TRÁFICO

La siguiente gráfica presentada en la Figura 5.4 presenta el flujo de mensajes de *Twitter* entre las fechas 16 de febrero y 2 de marzo del año 2010 perteneciente al conjunto de datos utilizado, en ella se aprecia que se produjo un *peak* de *tweets* al momento de producirse el evento terremoto.

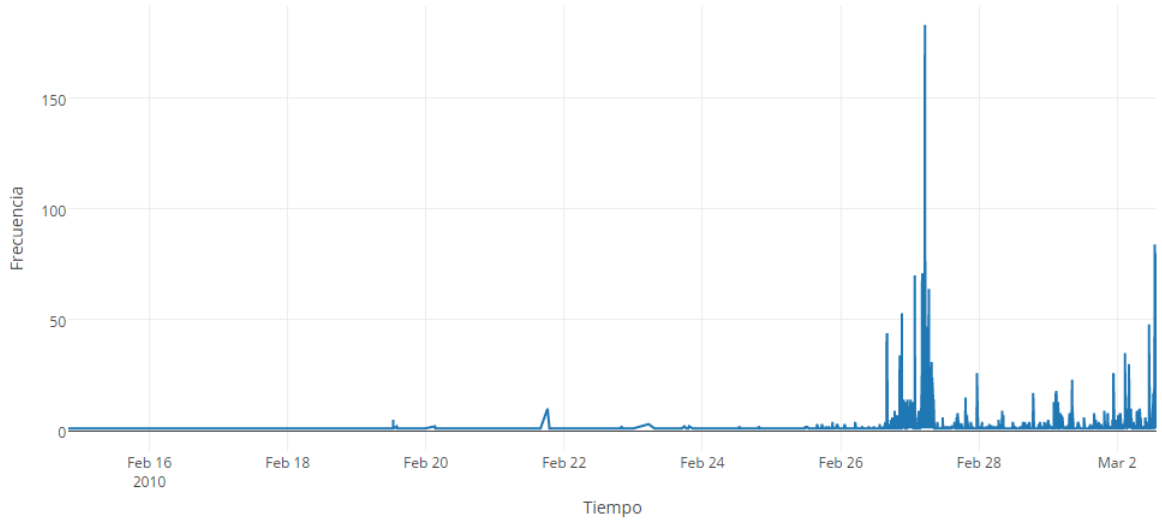

Figura 5.4: Distribución de estados presente en el conjunto de datos utilizado para evaluar.
Fuente: Elaboración Propia, (2016)

Este *peak* de información sólo se ve reflejado en el sistema como en un posible aumento de los mensajes con contenido del evento, mas no una cantidad de mensajes, pues la

herramienta, Twitter4J, realiza un muestreo constante.

Teniendo en consideración lo anterior en cuanto a la cantidad de mensajes, se realizó una recopilación de eventos por un periodo de una hora del *stream* actual de eventos para obtener un total de 150.800 *tweets*. El gráfico presentado en la Figura 5.5 presenta la cantidad eventos acumulados en función del tiempo cuya pendiente indica que se reciben 42 estados por segundo.

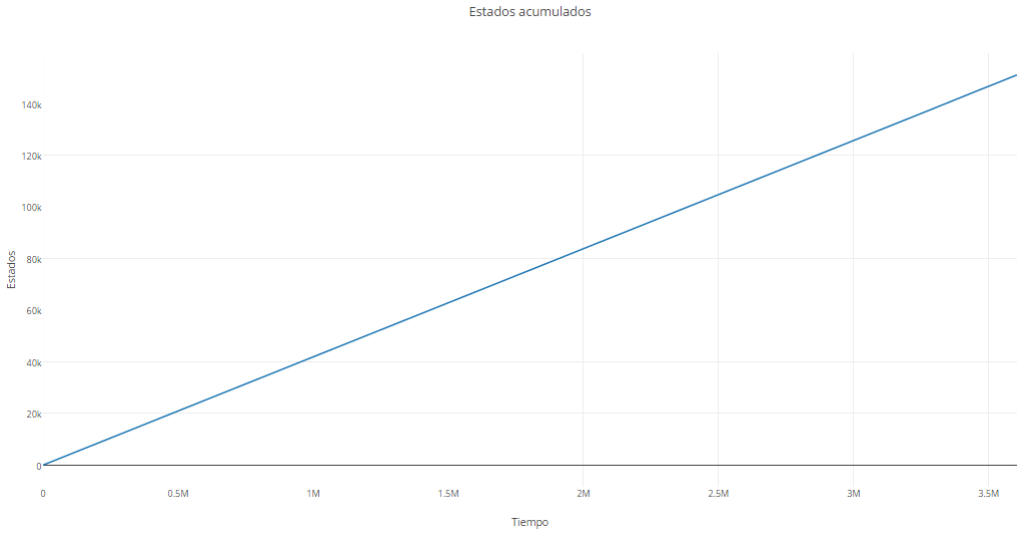

Figura 5.5: Recopilación de datos desde el *stream*.
Fuente: Elaboración Propia, (2016)

Haciendo uso de éstos resultados se simuló la llegada de los estados contenidos dentro de los datos de prueba sobre el sistema. Los resultados de esta simulación del comportamiento de la aplicación al operar como si fuese un evento real se muestran a continuación. La máquina para realizar estas pruebas fue la descrita en la sección 1.5.2 y se utilizó la herramienta YourKit. Se utilizaron 100.000 datos para realizar esta simulación y no se limitó la cantidad de estados por segundo para probar el comportamiento del sistema con una carga de trabajo mayor a la habitual.

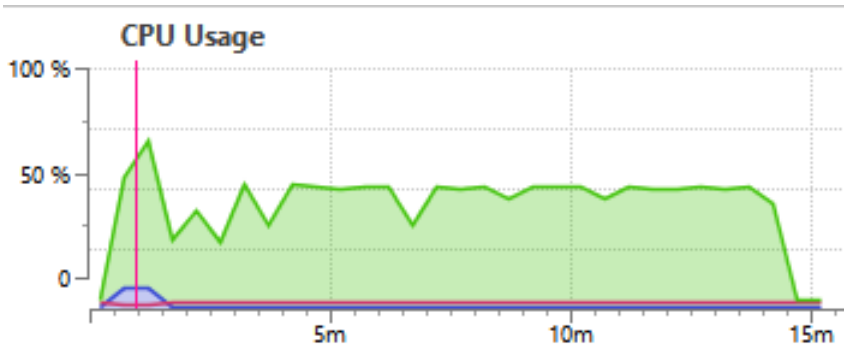

Figura 5.6: Uso de CPU durante la simulación.
Fuente: Elaboración Propia, (2016)

La gráfica presentada en la Figura 5.6 muestra el uso de la CPU del proceso *DeNe-Core* durante la simulación realizada, se ve gran variación del uso de CPU al comienzo de la operación, ésto se debe al uso de un segundo *spout* encargado de emitir los *tweets* almacenados en un archivo del tipo JSON correspondientes al evento del año 2010, esto se aprecia en la Figura 5.7 que muestra la cantidad de archivos abiertos en el tiempo. Posterior a la carga de los datos al sistema, sin considerar el primer segmento, el uso de CPU se mantiene constante al rededor del 49% hasta aproximadamente 10 minutos después de comenzar que terminan de procesarse todos los datos. Teniendo en consideración que se entregaron 100.000 datos se procesan alrededor de 166 estados por minuto en esta prueba, eso es aproximadamente un 400% más de lo que es posible recoger desde el *stream* de *Twitter*, por lo que se espera que en condiciones normales de operación, y dada la limitante del tiempo real, el uso del CPU sea menor al presentado en esta prueba. Siempre teniendo en cuenta que la cantidad de mensajes que alcanzan la mayor parte del sistema están dados por qué halla especificado el usuario en sus términos de búsqueda. Para éste caso, se utilizaron los mismos términos especificados anteriormente: ”terremoto”, ”Chile” y ”Concepción.”

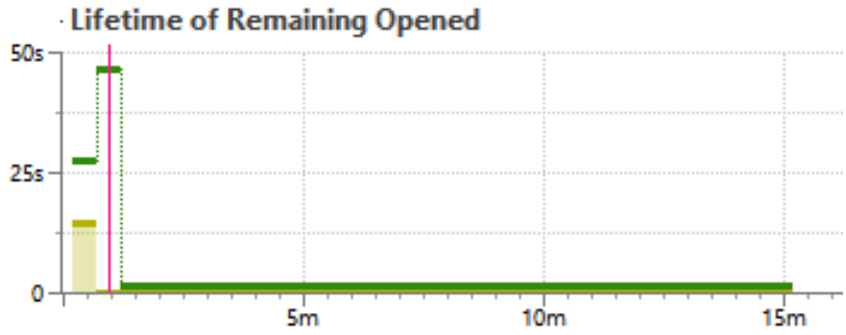

Figura 5.7: Archivos abiertos durante la simulación.
Fuente: Elaboración Propia, (2016)

CAPÍTULO 6. CONCLUSIONES

Twitter al entregar a los desarrolladores un punto de acceso a la información que está fluyendo por la aplicación en tiempo real entrega una potente herramienta para la búsqueda de información, pues sus datos pueden ser objeto de análisis en diferentes áreas, por ejemplo, búsqueda de tendencias, política, eventos, etcétera. Lo que hace esta red social tan especial es que cualquiera puede acceder a ella, no en vano cuenta con millones de usuarios en todo el mundo y mediante su API posibilita obtener información. de primera fuente. En las pruebas realizadas no se vio influencia negativa dada la cuota máxima de accesos impuesta por *Twitter*, pero es un ítem que se ha de tener en cuenta al desplegar la aplicación para funcionar en modo producción, pues las esperas pueden significar datos perdidos.

El objetivo de este trabajo fue, en todo momento, lograr la detección de necesidades en los estados publicados en esta red social; los *tweets*. Para ello se utilizó técnicas de aprendizaje de máquina, en este caso, minería *web*. Ésta se diferencia de la minería tradicional dado que su fuente de información está en línea y no en bases de datos almacenadas para ser analizadas. Para el caso de *Twitter*, presenta un desafío pues su información, el texto del *tweet* en sí, no está estructurada, éste texto puede tener cualquier formato dentro de sus 140 carácteres.

Para solucionar éste inconveniente antes planteado y se utilizó la metodología KDD, que mediante sus cinco etapas permitió la construcción de un clasificador de texto utilizando el algoritmo de *Naïve Bayes*, con el cual es posible etiquetar nuevos datos para la detección de necesidades. Si bien, como se mencionó en la sección 5.2, el clasificador puede no ser el mejor, de hecho, presenta gran precisión, pero no es exhaustivo al momento de clasificar (bajo *recall*). La necesidad de tener un cuerpo de entrenamiento con gran cantidad de instancias fue un problema, pues el tiempo utilizado para etiquetar datos fue muy grande, para el caso del cuerpo utilizado de aproximadamente 2000 instancias se tardó dos horas continuas, para no retrasar el desarrollo se optó por no etiquetar nuevos datos para incrementar el tamaño del conjunto de entrenamiento y en su lugar, para solventar esta debilidad del sistema, se construyó un módulo que permitiese la actualización del clasificador.

El sistema de detección de necesidades consistió en dos aplicaciones, comunicadas por medio de la base de datos, así es posible manejar grandes cantidades de datos sin que el sistema colapse la memoria del equipo.

Sobre los objetivos específicos podemos señalar, respectivamente, lo siguiente con respecto a su completitud:

- Se implementó un *spout* capaz de obtener el *stream* de *Twitter*, para seleccionar aquellas que hicieran referencia al territorio nacional se utilizó un segundo operador, el detector de ubicación.

- Se inició con el trabajo realizado por Olteanu et al. (2015), pero finalmente y avalado por el equipo de trabajo del proyecto FONDEF IDeA, se decidió utilizar una modificación de la categorización realizada por **?**.
- Se construyó un cuerpo de entrenamiento, con el cual se diseñó un clasificador bayesiano haciendo uso de la herramienta Mallet.
- Mediante *Apache Storm* se definieron 8 elementos, además del *spout* para obtener los datos, con los cuales se dió soporte al sistema de detección de necesidades con capacidad de procesar grandes cantidades de datos.
- La escalabilidad está dada en primer lugar, y principalmente, por el uso de *Apache Storm*, pues la aplicación de detección es aquella que realiza todo el procesamiento del sistema, en tanto la aplicación visualizadora se vale de los tiempos de respuesta de la base de datos y, precisamente, en segundo lugar la rapidez en las respuestas a las consultas realizadas a la base de datos de MongoDB permiten que no se produzcan respuestas lentas al manejar un gran volúmen de datos.
- Se simuló una condición en tiempo real de la aplicación haciendo uso de los datos recopilados en el terremoto de Concepción del año 2010 presentando en la sección 5.4 los resultado de ésta simulación.

Finalmente podemos señalar que, por medio de la concreción de los objetivos específicos señalados el sistema de detección de necesidades está completo y el objetivo general: ”Construir un sistema escalable para la detección de necesidades de la población en tiempo real para escenarios de desastre natural haciendo uso de Twitter.” se completó, presentando este sistema listo recolectar los datos desde *Twitter* y detectar los eventos categorizados como necesidades que allí aparezcan.

Habiendo concluido el desarrollo de la aplicación queda como trabajo futuro fundamentalmente dos puntos: Por un lado, la construcción de un conjunto de datos de entrenamiento más grande y cuyos elementos por clases estén balanceados como señala la teoría para permitir generar un mejor modelo de clasificación. Por otro la implementación de una API que dinámicamente encuentre los sinónimos de los términos de búsqueda para expandir de mejor manera las consultas, en lugar de hacer uso de un diccionario de sinónimos estáticos con sólo ciertos términos.

Es importante señalar que aunque se utilizó para el desarrollo de éste trabajo un conjunto de datos correspondientes al terremoto de febrero del año 2010, el sistema no sólo sería capaz de detectar necesidades en estas circunstancias, es de propósito general, pero limitado por el clasificador que puede, como se mencionó, ser mejorado.

GLOSARIO

- API: acrónimo inglés para Application Programming Interface o Interfaz de programación de aplicaciones.
- BSON: es un formato de intercambio de datos usado principalmente para su almacenamiento y transferencia en la base de datos MongoDB. Es una representación binaria de estructuras de datos y mapas. El nombre BSON está basado en el término JSON y significa *Binary JSON* (JSON Binario).
- Computación distribuida: computación haciendo uso de distintos nodos para el procesamiento en lugar de la programación clásica en un único nodo.
- *Framework*: estructura que sirve de base para la organización y desarrollo del software. Puede proveer de soporte a programas y bibliotecas entro otros.
- *Hashtag*: secuencia de palabras concatenadas antecedidas por un gato (#), actúa como etiqueta.
- *Hot-spot* o cuello de botella: se refiere a nodos donde se producen cuellos de botella producto de que recibe información de muchos otros nodos y donde sólo este puede procesar la información.
- JSON: acrónimo de *JavaScript Object Notation*, es un formato de texto ligero para el intercambio de datos.
- JVM: una máquina virtual Java (JVM), es una máquina virtual de proceso nativo, es decir, ejecutable en una plataforma específica, capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial el cual es generado por el compilador del lenguaje Java.
- *REST*: arquitectura de software presentada por Roy Fielding, para más información refiérase a su tesis doctoral. Fielding (2000).
- *Stream*: se refiere al flujo de información, en este contexto, al flujo de mensajes que se están produciendo en cada momento.
- *Trending Topic*: palabras o frases más repetidas en un momento en concreto.
- *Tradeoff*: anglicismo. se refiere a que debe sacrificarse algo para mejorar en otro.
- *Tweet*: los mensajes de *Twitter* son denominados como *tweets*. Tienen una longitud máxima de 140 caracteres.

- *Twitter*: servicio de microblogging. Permite enviar mensajes de texto de corta longitud.
- URL: secuencia de caracteres que siguen un formato estándar que asigna recursos en una red.
- *Performance*: anglicismo. Se refiere al rendimiento.
- *Stopwords*: anglicismo. Se traduce como Palabra vacía. Consiste en palabras sin significados como artículos, pronombres, preposiciones, etc. Suelen filtrarse para realizar procesamiento de lenguaje natural.

REFERENCIAS BIBLIOGRÁFICAS

Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., & Whittle, S. (2013). Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, *6*(11), 1033–1044.
URL http://dx.doi.org/10.14778/2536222.2536229

Apache (2016). Apache spark™ is a fast and general engine for large-scale data processing. Accedido el: 18/07/2016.
URL http://spark.apache.org/

Arshi Saloot, M., Idris, N., Shuib, L., Gopal Raj, R., & Aw, A. (2015). Toward tweets normalization using maximum entropy. In *Proceedings of the Workshop on Noisy User-generated Text*, (pp. 19–27). Beijing, China: Association for Computational Linguistics.
URL http://www.aclweb.org/anthology/W15-4303

Balazinska, M., Balakrishnan, H., Madden, S. R., & Stonebraker, M. (2008). Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, *33*(1), 3:1–3:44.
URL http://doi.acm.org/10.1145/1331904.1331907

Bird, S., Klein, E., & Loper, E. (2009). *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. Beijing: O'Reilly.
URL http://www.nltk.org/book

Bonzanini, M. (2015). Mining twitter data with python (part 2: Text pre-processing).
URL https://marcobonzanini.com/2015/03/09/mining-twitter-data-with-python-part-2/

Carta-natal (2016). Ciudades de chile. Accedido: 09/05/2016.
URL https://carta-natal.es/ciudades/Chile/

Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., & Pietzuch, P. (2013). Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (pp. 725–736). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/2463676.2465282

Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., & Sears, R. (2010). Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, (pp. 21–21). Berkeley, CA, USA: USENIX Association.
URL http://dl.acm.org/citation.cfm?id=1855711.1855732

Dean, J., & Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, *51*(1), 107–113.
URL http://doi.acm.org/10.1145/1327452.1327492

Dev.twitter.com (2016). Developer agreement and policy - twitter developers. Accedido: 05/07/2016.
URL https://dev.twitter.com/overview/terms/agreement-and-policy

eMarketer (2015). Latin america to register highest twitter user growth worldwide in 2015. Accedido: 07/07/2016.
URL http://www.emarketer.com/Article/Latin-America-Register-Highest-Twitter-User-Growth-World
1012498

Fayyad, U. M., & Uthurusamy, R. (Eds.) (1995). *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95), Montreal, Canada, August 20-21, 1995*. AAAI Press.
URL http://www.aaai.org/Library/KDD/kdd95contents.php

Fernández, I. (2011). Data mining: torturando a los datos hasta que confiesen. Accedido: 18/07/2016.
URL http://textmining.galeon.com/

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis. AAI9980887.

Foundation, A. S. (2016). Apache storm. Accedido: 18/07/2016.
URL http://storm.apache.org/

Gautreau, G. (2010). jqrangeslider.
URL http://ghusse.github.io/jQRangeSlider/index.html

Gedik, B., Schneider, S., Hirzel, M., & Wu, K. L. (2014). Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, *25*(6), 1447–1463.

Gonzales, P., & Wladdimiro, D. (2014). Online data processing on s4 engine: A study case on natural disasters. *IV Workshop de Sistemas Distribuidos y Paralelismo*, *4*.

Hønsi, T., & Hjetland, G. (2006). Column - highcharts. Accedido: 15/06/2016.
URL http://www.highcharts.com/stock/demo/column

Hunt, P., Konar, M., Junqueira, F. P., & Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, (pp. 11–11). Berkeley, CA, USA: USENIX Association.
URL http://dl.acm.org/citation.cfm?id=1855840.1855851

Hwang, J.-H., Balazinska, M., Rasin, A., Cetintemel, U., Stonebraker, M., & Zdonik, S. (2005). High-availability algorithms for distributed stream processing. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, (pp. 779–790). Washington, DC, USA: IEEE Computer Society.
URL http://dx.doi.org/10.1109/ICDE.2005.72

Imran, M., Castillo, C., Diaz, F., & Vieweg, S. (2014a). Processing social media messages in mass emergency: A survey. *CoRR*, *abs/1407.7071*.
URL http://arxiv.org/abs/1407.7071

Imran, M., Castillo, C., Lucas, J., Meier, P., & Vieweg, S. (2014b). Aidr: Artificial intelligence for disaster response. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, (pp. 159–162). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/2567948.2577034

Inc, G. (2014). Marker clusterer – a google maps javascript api utility library. Accedido: 01/06/2016.
URL https://github.com/googlemaps/js-marker-clusterer

IPN (2013). Sistemas de procesamiento distribuido. Accedido: 03/07/2016.
URL http://bit.ly/29xAkUN

Iribarra, F. (2015). Descubrimiento del conocimiento (kdd) : "el proceso de minería". Accedido: 01/05/2016.
URL http://mineriadatos1.blogspot.cl/2013/06/descubrimiento-del-conocimiento-kdd-el.html

Jones, M. T. (2013). Procese big data en tiempo real con twitter storm. Accedido: 15/07/2016.
URL https://www.ibm.com/developerworks/ssa/library/os-twitterstorm/

Lynn, T., Scannell, K., & Maguire, E. (2015). Minority language twitter: Part-of-speech tagging and analysis of irish tweets. In *Proceedings of the Workshop on Noisy User-generated Text*, (pp. 1–8). Beijing, China: Association for Computational Linguistics.
URL http://www.aclweb.org/anthology/W15-4301

Macool (2013). Mysql vs postgresql vs mongodb (velocidad). Accedido: 28/05/2016.
URL http://macool.me/mysql-vs-postgresql-vs-mongodb-velocidad/04

Maldonado, L. (2012). *Análisis de sentimiento en el sistema de red social Twitter*. Master's thesis, Universidad de Santiago de Chile, Av. Libertador Bernardo O'Higgins 3363, Santiago, Región Metropolitana. Memoria de pregrado.

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press.

McCallum, A. K. (2002). Mallet: A machine learning for language toolkit. Http://mallet.cs.umass.edu.

Molina, L. (2002). Data mining: torturando a los datos hasta que confiesen. Accedido: 18/07/2016.
URL http://www.uoc.edu/web/esp/art/uoc/molina1102/molina1102.html

Nakatani, S. (2010). Language detection library for java.
URL https://github.com/shuyo/language-detection

Neumeyer, L., Robbins, B., Nair, A., & Kesari, A. (2010). S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, (pp. 170–177). Washington, DC, USA: IEEE Computer Society.
URL http://dx.doi.org/10.1109/ICDMW.2010.172

Nguyen, D. T., & Jung, J. J. (2015). Real-time event detection on social data stream. *MONET*, *20*(4), 475–486.
URL http://dx.doi.org/10.1007/s11036-014-0557-0

Nicolás Hidalgo Castillo, E. R. (2014). Despliegue Ágil de aplicaciones de apoyo a la gestión de desastres de origen natural y caso de estudio en sismología.

Olteanu, A., Vieweg, S., & Castillo, C. (2015). What to expect when the unexpected happens: Social media communications across crises. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing, CSCW 2015, Vancouver, BC, Canada, March 14 - 18, 2015*, (pp. 994–1009).
URL http://doi.acm.org/10.1145/2675133.2675242

Play (2015). The high velocity web framework for java and scala. Accedido: 18/07/2016.
URL https://www.playframework.com/

Porter, M. (1979). The porter stemming algorithm. Accedido: 01/06/2016.
URL https://tartarus.org/martin/PorterStemmer/

Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., Zhou, L., Yu, Y., & Zhang, Z. (2013). Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (pp. 1–14). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/2465351.2465353

Ramos, J. A. (2015). Introducción a apache storm. Accedido: 15/04/2016.
URL https://www.adictosaltrabajo.com/tutoriales/introduccion-storm/

Ramírez, K. (2012). Stemming – lematización. UCR – ECCI CI-2414 Recuperación de Información.

Russell, S. (2003). *Artificial intelligence : a modern approach*. Upper Saddle River, N.J: Prentice Hall/Pearson Education.

Satzger, B., Hummer, W., Leitner, P., & Dustdar, S. (2011). Esc: Towards an elastic stream computing platform for the cloud. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, CLOUD '11, (pp. 348–355). Washington, DC, USA: IEEE Computer Society.
URL http://dx.doi.org/10.1109/CLOUD.2011.27

Sebepou, Z., & Magoutis, K. (2011). Cec: Continuous eventual checkpointing for data stream processing operators. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, (pp. 145–156). Washington, DC, USA: IEEE Computer Society.
URL http://dx.doi.org/10.1109/DSN.2011.5958214

Shah, M. A., Hellerstein, J. M., & Brewer, E. (2004). Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, (pp. 827–838). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/1007568.1007662

Smith, C. (2016). By the numbers: 170+ amazing twitter statistics. Accedido: 07/07/2016.
URL http://bit.ly/1bSfjNi

SQLStream (2015). Stream processing explained. Accedido: 18/07/2016.
URL http://www.sqlstream.com/stream-processing/

Statista (2016). Number of monthly active twitter users worldwide from 1st quarter 2010 to 1st quarter 2016 (in millions). Accedido: 07/07/2016.
URL http://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/

Tirados, M. (2014). Apache spark, la nueva estrella de big data. Accedido el: 18/07/2016.
URL http://www.bigdatahispano.org/noticias/apache-spark-la-nueva-estrella-de-big-data/

Tobin, J. (2016). Myth busting: Mongodb scalability (it scales!). Accedido: 04/05/2016.
URL https://www.percona.com/blog/2016/02/19/myth-busting-mongodbs-scalability/

Twitter, I. (2016). The streaming apis. Accedido: 03/07/2016.
URL https://dev.twitter.com/streaming/overview

Valer, S. (2011). *EVALUACIÓN DE TÉCNICAS Y SISTEMAS DE PROCESAMIENTO DE DATA STREAMS*. Master's thesis, Universidad de Zaragoza, Calle de Pedro Cerbuna, 12, 50009 Zaragoza, España. Master Thesis Computer Science.

Verma, A., Cho, B., Zea, N., Gupta, I., & Campbell, R. H. (2013). Breaking the mapreduce stage barrier. *Cluster Computing*, *16*(1), 191–206.
URL http://dx.doi.org/10.1007/s10586-011-0182-7

Wang, H., Peh, L.-S., Koukoumidis, E., Tao, S., & Chan, M. C. (2012). Meteor shower: A reliable stream processing system for commodity data centers. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, (pp. 1180–1191). Washington, DC, USA: IEEE Computer Society.
URL http://dx.doi.org/10.1109/IPDPS.2012.108

Wells, D. (2013). Extreme programming: A gentle introduction. Accedido: 01/07/2016.
URL http://www.extremeprogramming.org/

Weng, J., & Lee, B. (2011). Event detection in twitter. In *Proceedings of the Fifth International Conference on Weblogs and Social Media, Barcelona, Catalonia, Spain, July 17-21, 2011*.
URL http://www.aaai.org/ocs/index.php/ICWSM/ICWSM11/paper/view/2767

Xu, J., Chen, Z., Tang, J., & Su, S. (2014). T-storm: Traffic-aware online scheduling in storm. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ICDCS '14, (pp. 535–544). Washington, DC, USA: IEEE Computer Society.
URL http://dx.doi.org/10.1109/ICDCS.2014.61

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., & Stoica, I. (2013). Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (pp. 423–438). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/2517349.2522737

A. RECOLECCIÓN DE *TWEETS* CON RAPIDMINER

La recolección de *tweets* para comprobar el dato teórico correspondiente a la existencia de menos de un 1% de los *tweets* de *Twitter* contienen datos sobre su ubicación geográfica, se llevó a cabo haciendo uso de RapidMiner. Ésta herramienta permite recolectar *tweets* desde la API pública de *Twitter* y generar estadísticas en base a ello.

Para su recolección se definió un proceso, éste, descrito en la Figura a..1 corresponde a una iteración de n veces sobre el subproceso descrito en la Figura a..2, que es, en realidad, aquel que realiza la búsqueda de nuevos *tweets*.

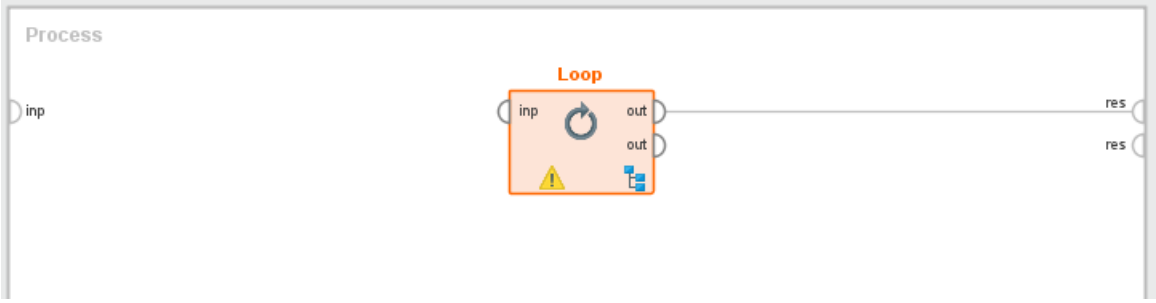

Figura a..1: Proceso de iteración en RapidMiner.
Fuente: Elaboración Propia, (2016)

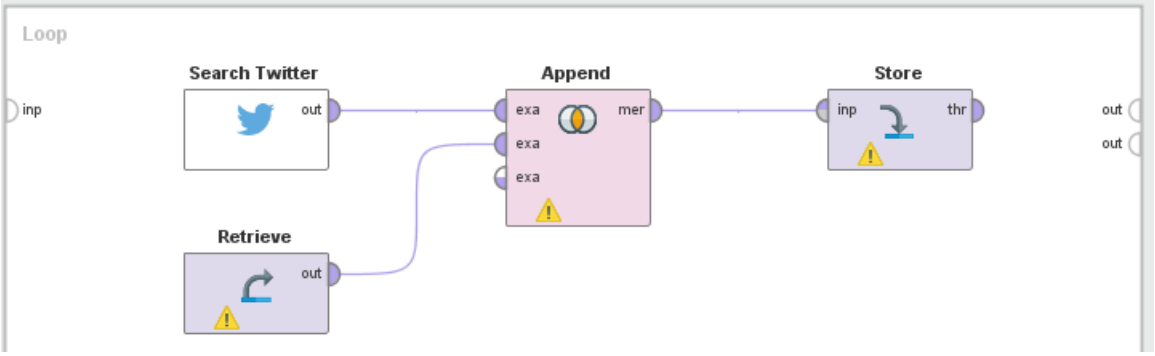

Figura a..2: Subproceso de recolección de estados en RapidMiner.
Fuente: Elaboración Propia, (2016)

Se recolectaron 67.789 *tweets* sólo con el idioma español en diferentes fechas, pues no se era capaz de obtener gran cantidad de datos debido a las limitaciones de *Twitter*. Sin importar el resto de los campos sólo se centra la atención en los correspondientes a la geolocalización presentados en la Figura a..3. La columna *Missing* indica que ese campo se encuentra vacío.

Name	Type	Missing	Statistics	Filter (2 / 12 attributes):	
Geo-Location-Latitude	Numeric	67475	Min -42.484	Max 59.625	Average -1.624
Geo-Location-Longitude	Numeric	67475	Min -118.136	Max 48.053	Average -65.020

Figura a..3: Estadísticas de *tweets* con geolocalización.
Fuente: Elaboración Propia, (2016)

Éstos resultados señalan que sólo un 0.46% de los datos allí muestreados cuentan con el campo de geolocalización con datos, lo que lleva a hacer creer que el valor descrito en la literatura es cierto y justifica aún más la necesidad del operador presentado en la sección 4.4.2.

B. CLAVES PARA EL USO DE LA *STREAM* API DE *TWITTER*

Para el uso de la API de *Twitter* se requiere de cuatro *tokens* de acceso. Éstos pueden ser obtenidos por cualquier persona que posea una cuenta en la red social. Para obtenerlos se debe acceder a la página: `https://apps.twitter.com/`. Allí en la parte superior derecha se encuentra la opción *"Create New App"* como se muestra en la Figura b..4.

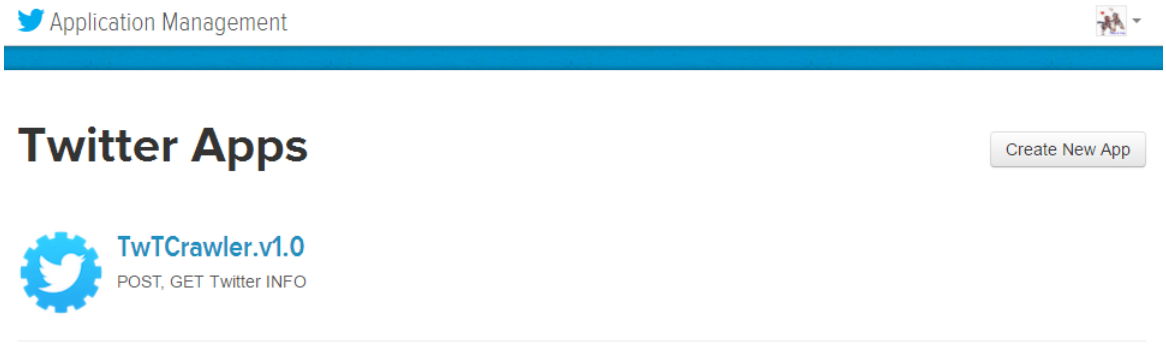

Figura b..4: Obtención de claves paso uno.
Fuente: Elaboración Propia, (2016)

Una vez realizado lo anterior corresponde llenar los datos solicitados como se muestra en la Figura b..5, aceptar los términos de desarrollador expuestos al final de la página y presionar el botón *"Create your Twitter application"*.

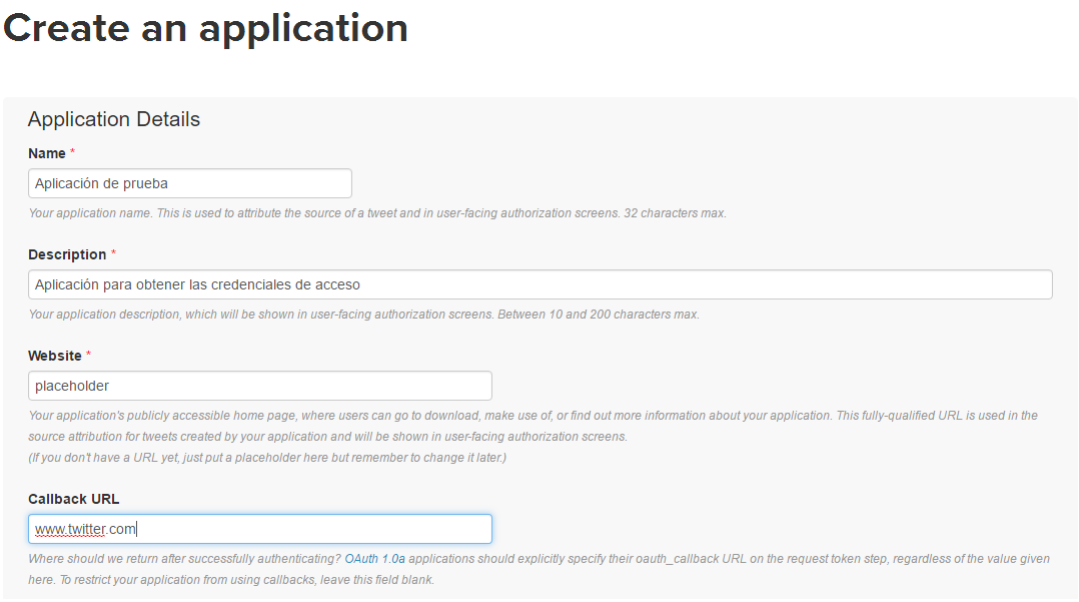

Figura b..5: Obtención de claves paso dos.
Fuente: Elaboración Propia, (2016)

La Figura b..6 muestra el siguiente paso en la pestaña *"Keys And Access Tokens"* donde se obtendrán las primeras dos claves de acceso. La tercera y cuarta claves se obtienen presionando el botón *"Regenerate Consumer Key and Secret"*, se solicitará confirmación y posterior a ello se redirige a la vista presentada en la Figura b..6. Para obtener las claves debe

presionarse, finalmente, el botón *"Test OAuth"*, se redirigirá a la página presentada en la Figura b..7 donde se muestran todas las claves.

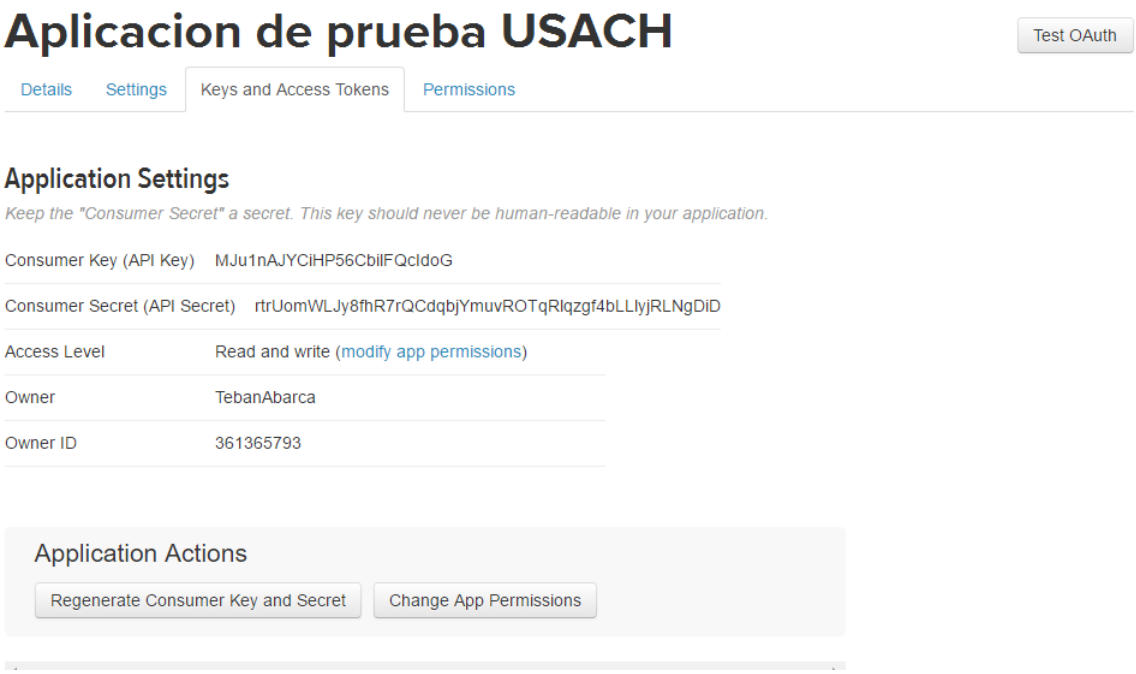

Figura b..6: Obtención de claves paso tres.
Fuente: Elaboración Propia, (2016)

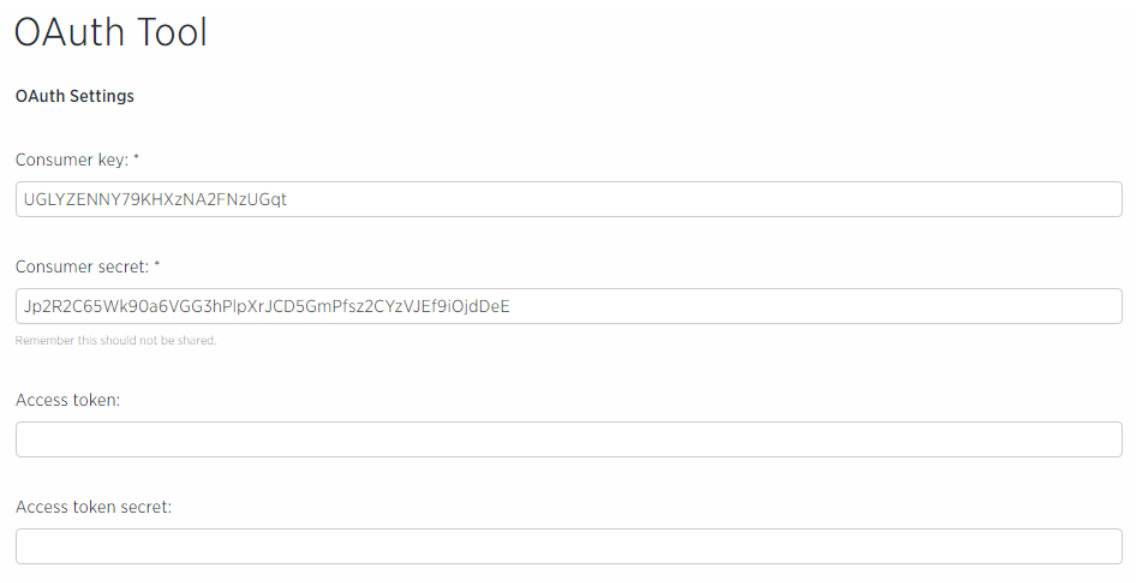

Figura b..7: Obtención de claves paso final.
Fuente: Elaboración Propia, (2016)