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

Tesis de grado presentada en conformidad a los requisitos para obtener el título de Ingeniero Civil en Informática

Santiago – Chile
2016

Dedicado a...

AGRADECIMIENTOS

Agradezco a

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 propone un sistema capaz de recoger la información desde *Twitter* — los denominados *tweets* — y procesarla 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 será capáz 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 lograr lo expuesto anteriormente, se ha construido un sistema cuya principal característica está en identificar, a partir de la información contenida en un *tweet*, si éste hace referencia o no a una necesidad y a dónde corresponde. (**PRINCIPALES RESULTADOS Y CONCLUSIONES ACA**).

Palabras Claves: Palabras; Claves

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 proposed a system able to get 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 has been made a system whose main characteristics are identify, by the *tweet*'s given metadata, if it references a need and where it corresponds.

Keywords: Key; words

TABLA DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE ALGORITMOS

CAPÍTULO 1. INTRODUCCIÓN

1.1 ANTECEDENTES Y MOTIVACIÓN

Los desastres naturales en el país han sido frecuentes en los últimos años. Sólo por mencionar algunos de los más recientes y recordados: la erupción del volcán Chaitén (Mayo, 2008), terremoto en Tocopilla (Noviembre, 2007), terremoto Concepción (2010), incendio de las Torres del Paine (Diciembre, 2011), incendio en Valparaíso (Abril, 2014), erupción volcán Villarrica (Marzo, 2015), 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 no resulta beneficioso para las personas que intentan sobrellevar de la mejor manera posible la crisis y esto se complica aún más cuando la necesidad involucra una necesidad básica, como la falta de agua, donde la vida de los afectados puede correr riesgo. No es problema sólo para las autoridades, Imran et al. (2014) señalan que el comportamiento humano ante crisis como éstas no es de quedarse esperando o huir en pánico, sino que intentan tomar decisiones rápidas en base a la información que conocen. Esto quiere decir que existe gente dispuesta a ayudar, aun siendo ellos los mismos afectados, pero no siempre disponen de la información necesaria para saber dónde apuntar sus esfuerzos. Sería útil, dado 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.

El grupo RESPOND de la Universidad de Santiago de Chile se ha adjudicado fondos para el desarrollo de un proyecto de dos años de duración el cual consiste en el desarrollo de una plataforma de *streaming* a escala nacional, enfocada en el procesamiento de datos en caso de crisis. Esta plataforma hará uso de la información generada por los usuarios en redes sociales como fuente de datos. Se espera que esta plataforma provea de herramientas para que cualquier persona pueda desarrollar nuevas aplicaciones para atender las diversas problemáticas que puedan existir cuando el país se enfrente a catástrofes.

Para ayudar a difundir la plataforma se requiere construir tres aplicaciones, una que apoye la coordinación de voluntarios, una segunda que difunda noticias y mensajes y, finalmente, una que permita detectar necesidades de la población, todas ellas al presentarse escenarios de catástrofes naturales.

En particular, para este trabajo, se espera atacar 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

Se requiere hacer uso de la información generada por la población en *Twitter* para que, en caso de alguna emergencia de carácter nacional, pueda prestarse apoyo a las autoridades encargadas de la toma de decisiones, por ejemplo, dándoles a conocer en qué lugar en particular se requiere asistir a la población con un determinado tipo de ayuda según la necesidad que se presente. ¿Cómo puede usarse la información disponible en *Twitter* para que, en casos de emergencia, ésta sea útil para ir en directo beneficio de la población en la que se generó satisfaciendo la necesidad específica que presentan?

1.3 SOLUCIÓN PROPUESTA

Se propone una aplicación, la cual estará recogiendo constantemente, en tiempo real, publicaciones desde *Twitter* y analizando si corresponde o no a una necesidad existente en el conjunto de necesidades detectables y, en casos afirmativos, mostrarlas durante un intervalo de tiempo sobre un mapa geográfico del país haciendo uso de los metadatos asociados al tweet, en el caso en que se encuentren disponibles o hacer uso del contenido para inferir sus ubicaciones si es posible.

Al tratarse de una aplicación que recogerá grandes cantidades de información, el desempeño que ésta tendrá ha de ser considerado, por ello, se hará uso de un *framework* de computación distribuida para procesar las grandes cantidades de tweets de la manera más eficiente posible. Lo anterior quiere decir que la aplicación tendrá, internamente, forma de grafo dirigido; cada nodo de este grafo corresponderá a un operador por el que la información fluirá. Estos operadores serán aquellos que la literatura señale como los apropiados para el caso, posibles operadores podrían ser: filtro de *stopwords*, filtro de *spam*, corrector ortográfico, detector de sentimientos, etcétera.

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 que serán detectadas.
3. Diseñar e implementar el clasificador de necesidades.
4. Definir de 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 podría ser el caso de una emergencia nacional.

1.4.3 Alcances

Se utilizarán 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 detectará no serán todas del universo posible de necesidades existentes, sino de un subconjunto que se considere más importante tanto por el equipo que está trabajando en el proyecto FONDEF; agua, vivienda o luz eléctrica, por ejemplo. De esta forma se logra acotar el problema reduciendo la cantidad de categorías y permitir una mayor precisión en la clasificación (trabajos similares han bordeado una precisión entre el sesenta y ochenta por ciento, pero estos resultados van de la mano con la cantidad de datos utilizados

para entrenar), entendiendo la precisión como la relación de elementos clasificados correcta o incorrectamente.

Se considera para la construcción del clasificador un subconjunto de un *dataset* de cuatro millones setecientos mil *tweets* recogidos desde *Twitter* correspondientes al terremoto ocurrido el 2010 en Chile. Este conjunto de datos contiene mensajes en distintos idiomas y ha sido filtrada llegando a aproximadamente un millón y medio de tweets; de aquel conjunto se obtendrá un subconjunto para realizar el etiquetado y ser usado como datos de entrenamiento. Este trabajo no surge de la nada, busca mejorar un trabajo anterior hecho en el marco de un proyecto PMI por Gonzales & Wladdimiro (2014).

1.5 METODOLOGÍA Y HERRAMIENTAS UTILIZADAS

1.5.1 Metodología

Este trabajo considera dos partes, la primera es la generación del clasificador y la segunda la construcción de la aplicación, donde la segunda depende de haber completado la primera, por ello la principal prioridad será desarrollar este clasificador.

Para realizar el clasificador se tiene considerado el proceso de KDD Fayyad & Uthurusamy (1995), acrónimo de Knowledge Discovery in Databases o, simplemente, Descubrimiento (o extracción) de conocimiento en bases de datos. Se refiere al “proceso no-trivial de descubrir conocimiento, patrones e información potencialmente útiles dentro de los datos contenidos en algún repositorio” Han & Kamber (2000). Este proceso iterativo diseñado para explorar grandes volúmenes de datos. Consta de cinco fases:

- Selección de datos: Se determinan las fuentes de datos y el tipo de información a utilizar. Se extraen los datos útiles de las fuentes de datos.
- Pre-procesamiento: Los datos se preparan y limpian. Se utilizan estrategias para rellenar los datos en blanco o con información faltante. Finalmente en esta etapa se obtiene una estructura de datos adecuada para ser transformada, posteriormente.
- Transformación: Consiste en el tratamiento preliminar de datos, transformación y generación de nuevas variables a partir de las ya existentes.
- Data Mining: Fase de modelamiento propiamente tal. Se utilizan métodos para obtener o detectar patrones que están “ocultos” en los datos.

- Interpretación y evaluación: Se identifican los patrones y se analizan por alguna métrica y se evalúan los resultados obtenidos.

En segundo lugar se tiene la aplicación propiamente tal que será dividida en dos, por un lado se tendrá la aplicación que llamaremos el núcleo que se encargará de recepcionar la información y el visualizador o interfáz que la mostrará por pantalla.

Para realizar lo anterior se hará el uso de *Extreme Programming* (en adelante XP), presentando avances semanales y discutiendo cambios a realizar en la aplicación, tanto visuales como de funcionamiento interno.

Para manejar las tareas se hará uso de un tablero kanban de cuatro columnas: ”Por hacer”, ”Haciendo”, ”Por Revisar” y ”Completo” donde una tarea sólo podrá considerarse completa habiendo pasado por la revisión y haber sido aceptada.

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.

1.5.2.1 Herramientas de software

Se han se utilizar las siguientes herramientas de software para la construcción de la aplicación:

- Java como lenguaje de programación.
- Apache Storm (1.0.1), como *framework* de computación distribuida.
- Apache Zookeeper (3.4.8), como herramienta para mantener la configuración
- Mallet (2.0.7), como herramienta de *Data Mining* para la construcción del clasificador.
- MongoDB (3.2.6), para la persistencia de datos.
- Play Framework (2.5.3), como *framework* para el desarrollo de aplicaciones Java. En particular, la construcción de la aplicación que permitirá visualizar los datos.
- 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).
- Linux Mint 17.3 (x86).
- Oracle VirtualBox (5.0.14).

1.5.2.2 Herramientas de hardware

Se utilizará el 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

A continuación se presentan a grueso modo los capítulos que componen el presente documento:

El capítulo 2: Marco Teórico presenta una serie de definiciones detalladamente para ayudar a comprender de mejor manera el problema y los elementos utilizados para su resolución.

El capítulo 3: Requerimientos detallan el proceso de toma de requerimientos de la aplicación.

El capítulo 4: Diseño e implementación presenta las decisiones de diseño que se tomaron para solucionar los problemas encontrados en el desarrollo del la aplicación como el proceso de implementación.

El capítulo 5: Evaluación del sistema detalla cómo se evaluó la solución desarrollada, el por qué se decidió una topología en particular y su nivel de replicación de operadores.

Finalmente en el capítulo 6: Conclusiones presenta las conclusiones del trabajo realizado, el cumplimiento de objetivos y trabajo futuro.

CAPÍTULO 2. MARCO TEÓRICO

Este capítulo busca dar a conocer al lector lo último en las áreas que el presente trabajo está inmerso, además de realizar una contextualización de los conceptos necesarios para entender el problema que se está tratando por medio de una breve reseña de cada uno.

2.1 ESTADO DEL ARTE

Los tópicos que se tratan en esta sección son variados, se comenzará 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.

2.1.1 Trabajo previo

En el marco de las jornadas chilenas de la computación Gonzales & Wladdimiro (2014) propusieron un modelo, desarrollado para el proyecto PMI USA1024, para detectar necesidades de la población ante escenarios de desastres naturales. 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: Utiliza un clasificador *Naïve Bayes* para identificar si un *tweet* es subjetivo o no.
- 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 sería el adecuado o cuándo replicar.

2.1.2 La problemática de *Twitter*

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 podra 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.

2.1.3 Procesamiento de la información

Para procesar datos, como los del *stream* de *Twitter*, donde los datos llegan en ráfagas, pueden ser de gran o poco volúmen y requieren de una respuesta inmediata, Harwood (2014),hace falta un cambio de paradigma respecto al procesamiento *batch* o procesamiento por lotes, donde un programa ejecuta procesos sin la intervención de terceros desde una base de datos o fichero, por uno donde los datos sean continuos y sin final conocido.

Se consideraron tres *frameworks* de computación distribuida: *Apache Storm*, *Apache Spark* y *Apache S4*. El primero presenta una solución basado en el modelo *MapReduce* que toma datos estructurados de la forma (clave, valor) para llevaros a una lista de valores y se usa

típicamente para procesar grandes cantidades de datos en distintos nodos que pueden o no estar cercanos físicamente. Los otros dos presentan un modelo basado en el procesamiento de eventos en tiempo real. El problema particular que presenta *S4* es la falta de avances en su desarrollo, el cual ha estado paralizado desde el año 2013.

Storm es un *framework* de computación distribuida para trabajar datos en tiempo real de múltiples fuentes de manera distribuida, tolerante a fallos y de alta disponibilidad. Su funcionamiento se divide en dos elementos: Por un lado existen los *Spout*, encargados de recoger el flujo de entrada de datos, y en segundo, los denominados *bolts*, encajados de procesamiento o transformación de los datos. Por recomendación un *bolt* sólo ha de realizar una tarea. *Storm* puede funcionar de dos formas: En modo *cluster* o modo local; en este último se simula un *thread* por nodo y es utilizado para realizar pruebas locales.

2.1.4 Clasificación de textos

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). Éste 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. De igual forma Gonzales & Wladdimiro (2014) 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*.

2.2 MINERÍA DE DATOS

A veces llamada como ”descubrimiento de información o conocimiento”, es el proceso de analizar información de diferentes perspectivas y transformarlo en información de utilidad. Puede ser aplicado a distintas fuentes de datos como: bases de datos, imágenes, internet, etc. Es un campo multidiciplinal que involucra el aprendizaje de máquina, la estadística, bases de datos, la inteligencia artificial y la recuperación de información.

Siendo distintos los usos que pueden dársele se pueden generalizar cuatro etapas:

- **Determinación de objetivos**: Delimitar los objetivos que se esperan alcanzar con el proceso de minado.
- **Preprocesamiento de datos**: Se refiere a la limpieza, reducción y transformación de las bases de datos. Es, generalmente, el subproceso que utiliza la mayor cantidad de tiempo.
- **Determinación del modelo**: Aplicación de algoritmos para generar un modelo que cumpla los objetivos planteados. Se genera nuevo conocimiento o se descubre un patrón.
- **Análisis de los resultados**: Se verifica si el conocimiento es útil.

Dentro de las tareas que pueden realizarse utilizando *data mining* pueden encontrarse tales como: Aprendizaje supervisado o clasificación, no supervisado o clustering y reglas de asociación.

2.2.1 Minería de la Web

La aplicación de la minería de datos al contenido que se encuentra en línea es conocida como Minería de la *web* o *web mining*. Se diferencia de la minería de datos tradicional en que ésta última utiliza repositorios de datos; en cambio, la minería *web*, hace uso de información extraída directamente desde la *web*.
Sus métodos son similares en cuanto a sus etapas:

- Selección de las fuentes: referencia al proceso de recuperación de los datos.
- Selección y pre-procesamiento: Incluye cualquier transformación o pre-procesamiento que puedan realizárseles a los datos, por ejemplo, eliminar elementos, como palabras, aplicación de correctores de datos, etc.
- Generalización: Etapa donde se realiza el proceso de minería en sí.
- Análisis: Desarrolla técnicas para utilizar o visualizar el conocimiento adquirido.

La información obtenida puede ser utilizara para analizar tanto el contenido de la web (*Web content mining*) como sus enlaces (o relaciones) (*Web structure mining*) y/o el registro de navegación de los usuarios (*Web usage mining*).

La primera se refiere a búsqueda entre documentos *web* (texto o imágenes), es decir, analiza los documentos y no la relación entre ellos.

La segunda se dedica a analizar la topología de los vínculos existentes y/o analizar la estructura interna de la página *web* y descrbir el *HTML* o el *XML* de la misma.

En particular dentro de la minería de contenido encontramos la minería de texto o *text mining*. Ésta tiene como objetivo el descubrir nueva información a partir colecciones de documentos de texto no estructurado, es decir, texto libre (lenguaje natural, generalmente), aunque también es aceptable otro tipo de información textual como un código fuente. Lo más habitual es trabajar el texto para categorizarlo (Asignar una o más categorías a un documento), clasificarlo (Asignar sólo una clase a un documento) y/o agruparlo (organizar en torno a una jerarquía basado en alguna similitud).

El primer paso para comenzar a trabajar haciendo uso de minería de texto es representar los datos de alguna manera para luego dárselo a los algoritmos adecuados. Algunas de estas representaciones pueden ser las siguientes:

- Bolsas de palabras (*Bag of words*): Representar el texto como un vector de largo n, donde n corresponde al número de palabras, así cada palabra corresponde a un elemento del vector.
- Frases: Considera el texto, simplemente, como una frase sintáctica. Así se permite conservar el contexto.
- N-gramas: Consideran la información de la posición de la palabra en el texto mediante secuencias de longitud n (n-gramas).

Habiendo realizado la representación, el paso siguiente es reducir el conjunto de características. La literatura indica que los métodos más frecuentados son la eliminación de palabras que no aportan información, llevar las palabras a una palabra raíz (*stemming*), entre otros. Son (2006).

2.3 APRENDIZAJE SUPERVISADO

Se caracteriza por ser un proceso de aprendizaje en el que éste se realiza mediante un entrenamiento controlado por un agente externo, el que determina qué respuesta debería generarse a partir de una entrada determinada. Cica (2000).

Se asocia al concepto de *machine learning* con la minería de datos; la primera busca patrones conocidos y predecir en base a ellos mientras que la segunda busca patrones con anterioridad desconocidos, es decir, la primera tiene una función focalizada en la predicción mientras que la segunda realiza una función exploratoria.

Los datos son denominados instancias, ejemplares, casos o vectores, donde una instancia corresponde a cada uno de los datos disponibles para el análisis.

Los datos poseen atributos son los elementos dentro de las instancias. Una instancia puede tener asociado un elemento de otro conjunto de atributos llamado ”Clase”, correspondiente a etiquetas de identificación.

Teniendo en cuenta los elementos vistos con anterioridad, se define el objetivo del proceso de aprendizaje como construir una función que relacione las instancias con las clases llamada modelo o, en este caso, clasificador.

Se le llama conjunto de entrenamiento al conjunto de datos utilizado para el aprendizaje. Este conjunto es entregado como entrada al algoritmo de aprendizaje y construcción del modelo. Para realizar la evaluación de la calidad del modelo se utiliza un segundo conjunto de instancias llamado datos de validación. Se espera que estos datos no hayan sido vistos con anterioridad por máquina y así obtener la confianza, es decir, la probabilidad de acierto que calcula el sistema para cada predicción.

Lo ventajoso de este método es que se podrá clasificar una instancia sin haberla visto nunca, pero la desventaja principal es la que han de utilizarse una gran cantidad de instancias para el proceso de entrenamiento. El proceso de entrenamiento y evaluación se ilustra en la Figura 2.1.

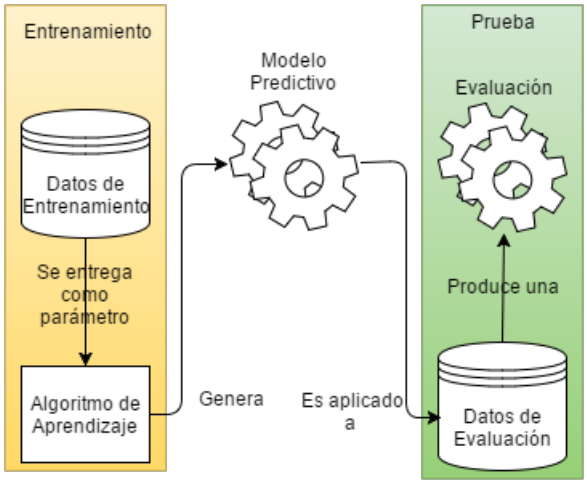

Figura 2.1: Proceso de entrenamiento y prueba del modelo.

Dentro de los algoritmos utilizados para la construcción de clasificadores se encuentra *Naïve Bayes* Russell (2003) a continuación se realiza una descripción de este.

2.3.1 Naïve Bayes

La clasificación puede verse como una función γ que asigna etiquetas a observaciones Izquierdo & Díaz (2012), es decir:

$$\gamma : (x_1, ..., x_n) \rightarrow \{1, 2, ...r_0\}$$

Existe una matriz de costo $cos(r, s)$ con $r, s = 1, ..., r_0$ en el cual se refleja el costo asociado a las clasificaciones incorrectas. En concreto $cos(r, s)$ indica el costo de clasificar un ejemplo de la clase r como de la clase s. En el caso especial de la función de pérdida 0/1, se tiene:

$$cos(r, s) = \begin{cases} 0 & \text{si } r \neq s \\ 1 & \text{si } r = s \end{cases}$$

Subyacente a las observaciones suponemos la existencia de una distribución de probabilidad conjunta:

$$p(x_1, ..., x_n, c) = p(c|x_1, ..., x_n)p(x_1, ..., x_n) = p(x_1, ..., x_n|c)p(c)$$

La cual es desconocida. El objetivo es construir un clasificador que miniza el coste total de los errores cometidos, y esto se consigue, Langley & Sage (2013) por medio del clasificador de Bayes:

$$\gamma(x) = argmin_k \sum_{c=1}^{r_0} cos(k, c)p(c|x_1, ..., x_n)$$

En el caso que la función de pérdida sea 0/1, el clasificador de Bayes se convierte en asignar al ejemplo $x = (x_1, ..., x_n)$ la clase con mayor probabilidad a posteriori. Es decir:

$$\gamma(x) = argmax_c p(c|x_1, ..., x_n)$$

En la práctica la función de distribución conjunta $p(x_1, ..., x_n, c)$ es desconocida, y puede ser estimada a partir de una muestra aleatorea simple $\{(x^{(1)}, c^{(1)}), ..., (x^{(N)}, C^{(N)})\}$ extraida de dicha función de distribución conjunta.

El paradigma clasificatorio en el que se utiliza el teorema de Bayes en conjunción con la hipótesis de independencia condicional de las variables predictorias dada la clase se conoce como Naïve Bayes, Langley & Sage (2013).

Finalmente el Teorema de Bayes está representado por la expresión CHristianCH

(2013):

$$P(c_i|d_j) = \frac{P(d_j|c_i) \cdot P(c_i)}{P(d_j)}$$

Donde c_i corresponde al atributo Clase y d_j al conjunto de documentos. El término $P(d_j)$ suele omitirse, pues no aporta mucha información para la clasificación. Habiendo realizado lo anterior y tomando en cuenta la hipótesis de independencia se obtiene:

$$P(d_j|c_i) = P(c_i) \prod_{j=1}^{n} P(a_j|c_i)$$

Pero siempre se considera la mayor probabilidad de c_i, por ello podemos añadir un nuevo elemento llegando a la fórmula de Naïve Bayes:

$$P(d_j|c_i) = ArgMaxk^n P(c_i) \prod_{j=1}^{n} P(a_j|c_i)$$

En términos simples el crear un modelo de clasificación Naïve Bayes se puede resumir en el Algoritmo 2.1.

Algoritmo 2.1: Algoritmos Naïve Bayes.

Entrada: Clases $C = \{c_0, \ldots, c_n\}$.

Entrada: Datos de entrenamiento $D = \{d_0, \ldots, d_n\}$

Salida: Modelo de clasificación M.

Para Clase c_i perteneciente a C **Hacer:**

Calcular las probabilidades a priori para cada clase (Probabilidad de que un dato cualquiera pertenezca a c_i.

Fin Para

Para Clase c_i perteneciente a C **Hacer:**

Realizar un recuento de los valores v_i de atributos que toma cada ejemplo d_i, guardarlos en V.

Fin Para

Para Valores v_i perteneciente a V **Hacer:**

Aplicar corrección de Laplace a v_i.

Fin Para

Para Valores v_i perteneciente a V **Hacer:**

Normalizar para obtener un rango de valores [0,1].

Fin Para

Para utilizar el modelo generado por el Algoritmo 2.1, se hace uso del Algoritmo 2.2

sobre un dato de entrada d_j.

Algoritmo 2.2: Algoritmos Naïve Bayes.

Entrada: Clases $C = \{c_0, \ldots, c_n\}$.

Entrada: Dato d_j.

Salida: Etiqueta del Dato d_i.

Para Clase c_i perteneciente a C **Hacer:**

Calcular las probabilidades a priori para cada clase (Probabilidad de que un dato cualquiera pertenezca a c_i.

Fin Para

retornar $ArgMax^n P(c_i) \prod_{j=1}^{n} P(d_j|c_i)$.

2.4 METODOLOGÍA

Para la realización de este trabajo se utilizarán dos metodologías, en esta sección se ambas son definidas.

2.4.1 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 provado exitosamente en compañias e industrias de todos los tamaños. Wells (2013).

Su éxito se debe a que hace especial incapié 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: Constantemente se comunica al equipo de desarrollo con el cliente. Se intenta mantener el diseño lo más simple y sencillo posible. Se obtiene retroalimentación desde las pruebas desde el día uno. Se les entrega el software al cliente lo más pronto posible con los cambios solicitados. El éxito depende, en gran medida, del respeto y comunicación de los miembros del equipo y los clientes. Implementando XP el equipo puede responder a los cambios sin temor.

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 si 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 hará lo que se solicitó, pero no más. Ésto maximiza el valor entregado dado una fecha límite. Nuestras metas se alcanzarán por medio de pequeños pasos para mitigar errores tan pronto ocurran. Crearemos algo de lo que estemos orgullosos y lo mantendremos en el tiempo a costos razonables.
- Comunicación: Todos somos partes de un equipo y nos comunicamos cara a cara a diario. Trabajaremos juntos en todo: desde la toma de requerimientos hasta la implementación. Crearemos la mejor solución posible al problema.
- Retroalimentación: Cada iteración será completada seriamente entregando *software* funcional. Mostraremos nuestro *software* a menudo y prontamente para luego escuchar y aplicar los cambios solicitados. Hablaremos de nuestro proyecto y adaptaremos 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: Diremos la verdad sobre el progreso y nuestras estimaciones. No se documentan excusas por si se falla porque se planea tener éxito. No tenemos porque no trabajamos solos. Nos adaptaremos a los cambio cuando ocurran.

El proceso de XP puede puede ser aprecido en la Figura 2.2.

Figura 2.2: Diagrama de flujo de Programación Extrema

2.4.2 *Knowledge Discovery in Databases* (KDD)

Es definido 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 cantidades 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 utilizarán 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 podrían 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 puede verse esquematizado en la Figura 2.3.

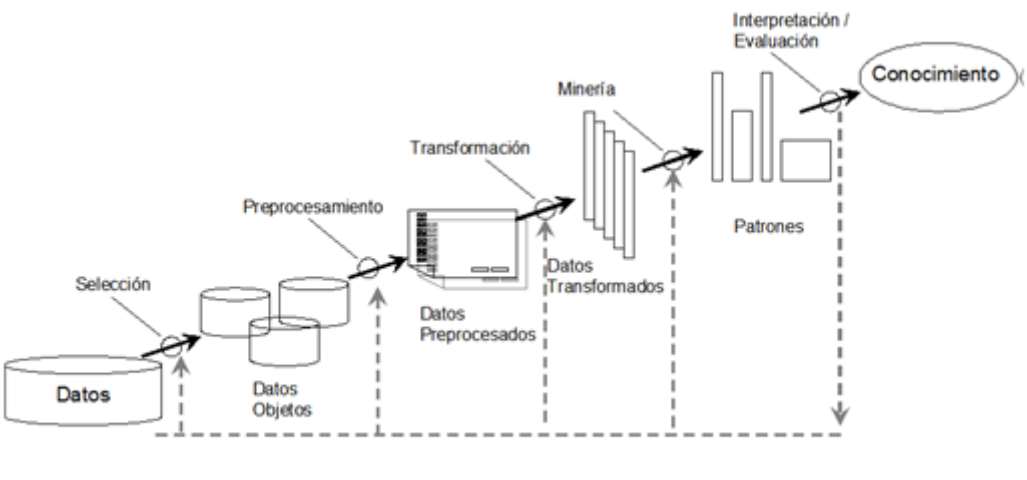

Figura 2.3: Proceso KDD.

2.5 HERRAMIENTAS

Esta sección busca dar a conocer al lector las herramientas de *software* utilizadas para desarrollar el sistema propuesto.

2.5.1 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 ”Convención sobre configuración”, el cual apunta a reducir la toma de decisiones que debe tomar el desarrollador sin perder flexibilidad.

Se enfoca en la productividad a aplicaciones *RESTful*.

Elimina la desventaja de desarrollo al utilizar *Java* dada por el continuo ciclo de compilar-empaquetamiento-despliegue. Al detectar cambios en el código realiza inmediatamente

la compilación y actualiza en la JVM sin necesidad de reiniciar el servidor.

Play no utilizar sesiones en su funcionar, privilegiando el uso de almacenamiento *offline* o el uso de peticiones *Ajax* para resolver problemas del lado del cliente.

2.5.2 Apache Storm

Apache Storm es un sistema de computación en tiempo real de código abierto. Simplifica el problema de flujos (*streams*), de datos sin que estos tengan fin.

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.

Se compone principalmente de dos partes. La primera es denominada *Spout* y es la encargada de recoger el flujo de datos de entrada. La segunda es denominada *Bolt* y es la encargada de la transformación o procesado de los datos.

Oficialmente es representado como puede verse en la Figura 2.4. donde 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.

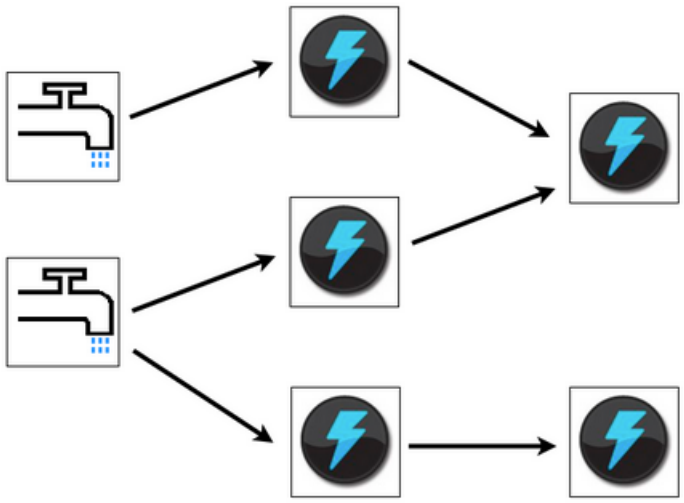

Figura 2.4: Representación del funcionamiento de Apache Storm.

Uno de los puntos fuertes que tiene este sistema 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.

Al trabajar con *Apache Storm*, como ya se ha descrito, se han de construir dos

elementos: *spout* y *bolt*. Éstos elementos se construyen realizando heredando desde desde *BaseRichSpout*, para el caso de *spout* e implementando *IRichBolt* para los *bolt*. La descripción de los elementos de éstas clases se presentan en las secciones 2.5.2.1 y 2.5.2.2.

2.5.2.1 Spout

```
1.  public class StormSpout extends BaseRichSpout{
2.
3.      @Override
4.      public void declareOutputFields(OutputFieldsDeclarer ofd) { }
5.
6.      @Override
7.      public void open(Map map, TopologyContext tc, SpoutOutputCollector soc) { }
8.
9.      @Override
10.     public void nextTuple() { }
11.
12. }
```

Figura 2.5: Construcción de un Spout.

En la figura 2.5 muestra la base para la implementación de un *spout*. Cuenta con tres métodos:

- *declareOutputFields*: declara nombres para las etiquetas que tendrá el objeto emitido.
- *open*: Inicializa todos los elementos utilizados en el spout.
- *nextTuple*: es llamado desde storm al recibir una nueva tupla para realizar una tarea.

2.5.2.2 Bolt

```java
public class TestBolt implements IRichBolt {

    @Override
    public void prepare(Map map, TopologyContext tc, OutputCollector oc) { }

    @Override
    public void execute(Tuple tuple) { }

    @Override
    public void cleanup() { }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer ofd) { }

    @Override
    public Map<String, Object> getComponentConfiguration() { }

}
```

Figura 2.6: Construcción de un Bolt.

La figura 2.6 presenta la base para la implementación de un *bolt*. Sus métodos se explican a continuación:

- *prepare*: Similar al método *open* para los *spout*, realiza la misma función.
- *execute*: Cumple la misma función que el método *nextTuple*.
- *cleanup*: Al finalizar una topología se llama este método para cerrar el *bolt*.
- *declareOutputFields*: declara nombres para las etiquetas que tendrá el objeto emitido.
- *getComponentConfiguration*: Usado al querer cambiar una configuración del sistema.

2.5.2.3 Topología

Una topología de Storm es similar a un grafo. 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).

2.5.2.4 Cluster de Storm

Un cluster de Storm no muere, se queda siempre en espera de nuevos datos de entrada mientras el proceso siga activo.

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 corriendo. En caso de fallo de uno *Worker Node*, Nimbus se dará cuenta y redirigirá 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*.

2.5.2.5 Modos de funcionamiento

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 'modo de producción' y es el modo donde el código es distribuido en máquinas diferentes dentro del Cluster.

2.5.2.6 Storm grouping

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. El valor puede ser cualquier tipo, para ello se ha de implementar un serializador. 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 gruoping: 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.

2.5.3 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 2.7, parte de una colección llamada ’Personas’ en MongoDB:

Figura 2.7: Documento en MongoDB.

Una consulta para encontrar este elemento dentro de la colección se daría de la forma aprecida en la Figura 2.8

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

Figura 2.8: Consulta en MongoDB.

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.

CAPÍTULO 3. REQUERIMIENTOS

Éste capítulo detallará los requisitos de las aplicaciones, descritos como historias de usuario. Éstos señalan las necesidades de los clientes - Profesores guía y co-guía - expresadas en sucesivas reuniones mantenidas en el Departamento de Ingeniería Informática de la universidad donde se mostró, semana a semana, avances en la aplicación y se señalaron los cambios que el sistema necesitaba. Éstos cambios o sugerencias, en general, nuevas características para ser agregadas al sistema, permitieron, en el tiempo, agregar nuevas y redefinir las historias de usuario que definen al mismo.

Estas historias de usuario tienen la siguiente nomenclatura para su identificación: Aquellos que guarden relación con la aplicación de detección se identificarán como ’HU-cXX’ donde XX corresponderá al número del requisito; ’HU-vYY’ para aquellas que correspondan a la aplicación interfáz donde, al igual que en el caso anterior, YY corresponderá al número del requisito.

La Tabla 3.1 presenta las historias de usuario correspondientes a los requisitos de todo el sistema en general, sin hacer referencia al cómo está dividido éste.

Tabla 3.1: Historias de usuario

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 añadirá la información de los nuevos eventos.

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

Faltan criterios, conversarlo con profes

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

Ya se ha mencionado que el sistema completo constará de dos aplicaciones: el visualizador que, además permitirá actualizar el clasificador, y la aplicación detectora de necesidades. Considerando lo anterior se presenta la arquitectura del sistema en la Figura 4.1. En ella se aprecia que los datos llegan al sistema desde *Twitter* y, utilizando Apache Storm para soportarlo, además del clasificador, son clasificados, detectando necesidades y almacenando, en la base de datos, los nuevos marcadores, luego, en la aplicación visualizadora, haciendo uso de Google Maps, son mostrados en el cliente *web* por medio de su navegador.

El sistema de detección de necesidades, en su conjunto, constará de dos aplicaciones independientes que trabajarán unidas para realizar el proceso de detección de eventos. Estas aplicaciones corresponden, por un lado, a aquella que estará encargada de realizar el proceso de clasificación propiamente tal la que se denominará ”Detector de eventos”, éste detector hará uso de *Apache Storm* para transformar el *stream* de *Twitter* a marcadores dispuestos en un mapa geográfico del país.

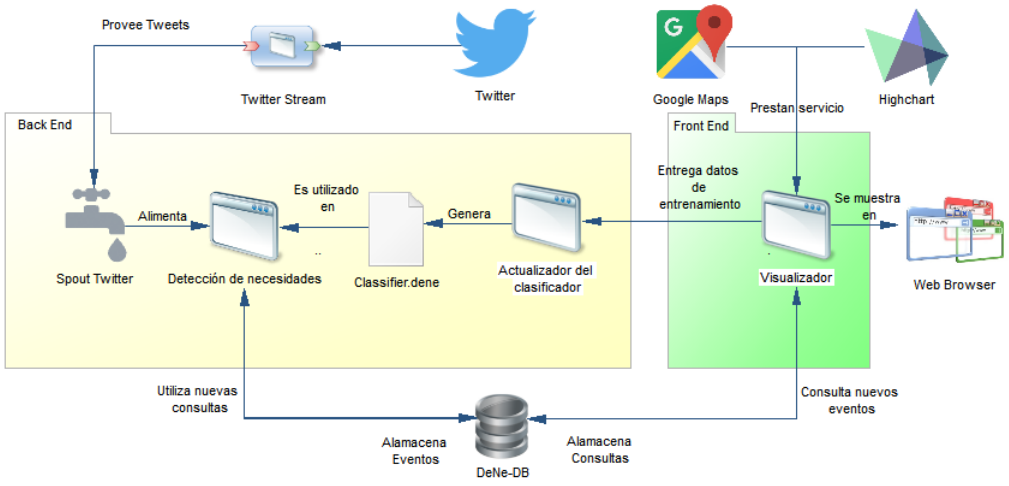

Figura 4.1: Arquitectura del sistema.

La figura 4.1 presenta la arquitectura del sistema. Éste, como se mencionó, está compuesto de dos aplicaciones. Todo el sistema es alimentado por el *stream* provisto por el servicio de *streaming* de *Twitter*, este *stream* ingresa, por medio de un *spout*, a la aplicación detectora de necesidades la que clasifica cada *tweet* del *stream* y lo transforma, de ser posible, en un marcador almacenado en la base de datos, desde donde es leído por la aplicación visualizadora la que, utilizando la API de *Google Map* y *Highcharts* para mostrarlos al usuario por medio de su navegador *web*.

Se hizo referencia a que un *tweet* puede no terminar como un marcador en la base de datos, ésto es producto de la serie de filtros que implementala aplicación de detección, los cuales se detallan en las siguientes secciones.

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 base de datos presta una función escencial al funcionamiento del sistema. Es aquí 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.

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.

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 3.1. 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 2.5.2, 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 constituirá 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.

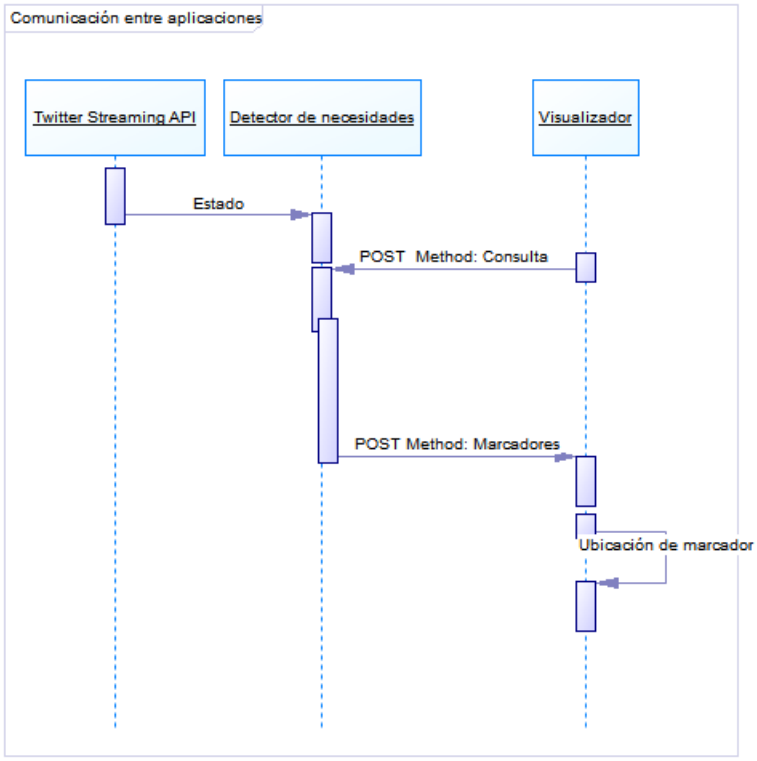

Figura 4.2: Esquema que representa la comunicación entre aplicaciones en primeras etapas del desarrollo.

Esta aproximación no consideraba la existencia de un sistema de persistencia de información; al considerar la persistencia la comunicación ya no se realizará por medio de peticiones REST, sino que se utilizará 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.

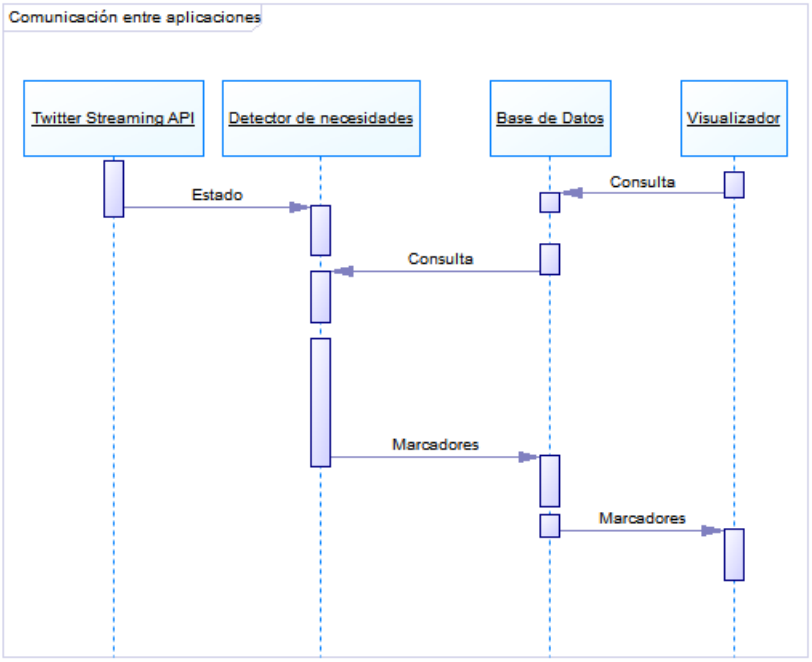

Figura 4.3: Esquema que representa la comunicación entre aplicaciones del sistema detector de necesidades.

4.3.2 Persistencia

Al hacer referencia a información pasada en la historia de usuario HU-v04 se infiere la necesidad de la implementación de un sistema de persistencia de datos. Es, principalmente, por esta razón que se modificó la forma de comunicación inicial presentada en la sección 4.3.1.

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 utilizará, 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 operará 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 utilizarán, 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 almacenarán sólo restaría 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”.

```
{
    "_id": objectId("_MongoDB_ID"),
    "contenido": "Contenido del tweet",
    "categoría": "Categoría del tweet",
    "latitud": "Coordenada Latitud",
    "longitud": "Coordenada Longitud",
    "generatedAt": ISODate("YYYY-mm-ddTHH:mm:sssZ")

}
```

Figura 4.4: Ejemplo de documento en la colección Markers.

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 esta historia de usuario, se requiere de un sistema capaz de procesar eventos en tiempo real que, dado el *peak* de información que recibirá el sistema deberá ser escalable. El problema en este punto es el cómo construir un sistema que cumpla capáz 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 si un componente (nodo) falla, otro será capaz de reemplazarlo. IPN (2013). En un inicio se consideraron tres plataformas sobre las cuales podría 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, 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 estan 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 2.4 en la página 21, 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 utilizará *Twitter* como fuente de información, así la unidad de información pasará, 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*

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 Anexo **??**.

4.3.5 Especificación de términos de búsqueda

Twitter4J, la herremienta que se menciona en la sección 4.4.2.1 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 estará encargado de realizar el filtrado de acuerdo a términos y llevar a cabo la operaciones descritas en la sección 4.4.2.3 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.

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 4.4.2.3.

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, prometía 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, será 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 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 Alvarado (2015) 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 mostrarán 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), como se mencionó en en el capítulo 1, 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 2.3.1. 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 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 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 separarán 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 permitirá la creación de un nuevo modelo clasificador.

Figura 4.6: Formato archivo de entrada.

1. Corresponderá a un identificador arbitrario, pero necesario para la herramienta de clasificación Mallet.
2. Corresponderá 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 construirá el clasificador y en otra se utilizará surge el problema de cómo realizar la comunicación entre ellas. Para solucionar este inconveniente se utilizará una carpeta compartida por ambas aplicaciones. En el caso de sistemas Unix se utilizará el directorio $/opt/DeNe$, mientras que para Windows se utilizará $C:/DeNe/$. En estos directorios se almacenará un fichero con el clasificador serializado.

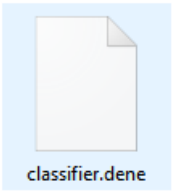

Figura 4.7: Fichero clasificador en $C:/DeNe/$.

Cada vez que se actualice el clasificador se contrastará el nuevo con el ya existente, de encontrar mayor precisión en el primero, se reemplazará en la carpeta antes mencionada, según el sistema operativo de la máquina que se esté utilizando. En caso contrario , se mantendrá al anterior. En ambos escenarios se le dará 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.

4.4.1.1 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 existirán 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.

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 y K no es ”todas las categorías” **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

Para realizar la selección del intervalo mencionado en la HU-v04 se solicitó, por parte del equipo FONDEF, 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 permitía la implementación de un histograma externo.

Figura 4.8: Selectorde fechas JDateRangeSlider.

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 arrojara 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.

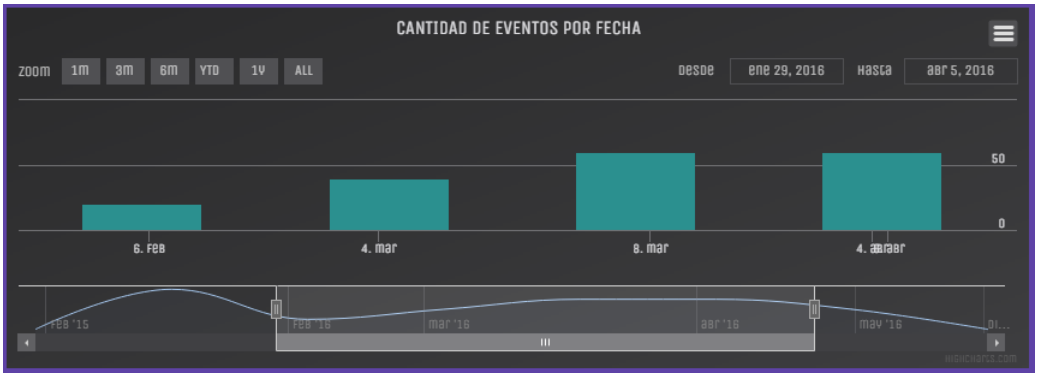

Figura 4.9: Selector de fechas presente en la aplicación.

```
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.

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 refería. Los diseños finales son presentados en las Figuras de la 4.11 a la 4.16.

Figura 4.11: Icono categoría agua.

Figura 4.12: Icono categoría alimento.

Figura 4.13: Icono electricidad agua.

Figura 4.14: Icono categoría comunicacion.

Figura 4.15: Icono categoría personas.

Figura 4.16: Icono categoría seguridad.

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*, **??**, 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.

Figura 4.18: Icono categoría agua para cluster medio.

Figura 4.19: Icono categoría agua para cluster grande.

Figura 4.20: Icono categoría alimento para cluster pequeño.

Figura 4.21: Icono categoría alimento para cluster medio.

Figura 4.22: Icono categoría alimento para cluster grande.

Figura 4.23: Icono categoría electricidad para cluster pequeño.

Figura 4.24: Icono categoría electricidad para cluster medio.

Figura 4.25: Icono categoría electricidad para cluster grande.

Figura 4.26: Icono categoría comunicación para cluster pequeño.

Figura 4.27: Icono categoría comunicación para cluster medio.

Figura 4.28: Icono categoría comunicación para cluster grande.

Figura 4.29: Icono categoría personas para cluster pequeño.

Figura 4.30: Icono categoría personas para cluster medio.

Figura 4.31: Icono categoría personas para cluster grande.

Figura 4.32: Icono categoría seguridad para cluster pequeño.

Figura 4.33: Icono categoría seguridad para cluster medio.

Figura 4.34: Icono categoría seguridad para cluster grande.

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 eliminará todo marcador del mapa y se reubicarán en el todos los que cumplan con los parámetros de búsqueda.

4.4.1.2 Estadísticas de procesamiento

Específicamente se solicitaron tres tipos de estadísticasa 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 incapáz 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 agregará 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.

Para la tercera estadística la colección de marcadores no sería útil, pues no reflejaría la cantidad de tweets procesados, para ello sería 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 tendrá 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.

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 4.4.2.3 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 dará 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 permitirá 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.

4.4.1.3 Configuración

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

- Modo tiempo real: Cuando el sistema esté en funcionamiento y cada cierto tiempo, t_1, se actualizarán los marcadores de los nuevos eventos y éstos se mostrarán 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.

4.4.2.1 Entrada de datos al sistema

Conociendo desde donde se obtendrá 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 surtirá 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.

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 2.4 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 4.4.2.1 corresponderá a aquel que recibe la información desde el *stream* de *Twitter*, pero ¿cuál será 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 desarrollará un operador y éstos, trabajando en conjunto, producirán la información necesaria para ser almacenada como un documento 'marcador' en la colección 'Markers' descrita en la Figura 4.35.

4.4.2.2 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 utilizarán.

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 2.5.2.2.

```
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.

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.

4.4.2.3 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 corresponderá 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 considerarán 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 sería del todo util. En su lugar se desarrolló 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 llegarán los mismos estados a todas las instancias del nodo, por ello este proceso se realizará de manera única para cada instancia en función de los estados que hayan llegado a él. El Algoritmo 4.4 agregará a los términos de búsqueda de cada instancia las palabras más frecuentes y, siguiendo el ejemplo de *Twitter* con sus *thrending topics* tendrá 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.

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.

4.4.2.4 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 4.4.1.2 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 corresponderá a normalizador de texto, el cual reemplazará menciones a usuarios, *hashtags* y URLs, todas ellas variables, por palabras constantes, el reemplazo a realizarse se muestra en la Tabla 4.2. Esto se fundamenta en los resultados descritos en el capítulo 5.

Tabla 4.2: Reemplazo de entidades en texto

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.

4.4.2.5 Operador geolocalizador

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

Ha sido señalado por Imran et al. (2014) que menos del 1% de los *tweets* contienen datos en sus campos correspondientes a geolocalización. En un experimento (véase anexo **??**) 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 continuarán en la línea de procesamiento. En el capítulo 5 realiza una evaluación sobre la efectividad del operador.

Los operadores descritos en las secciones 4.4.2.6 y 4.4.2.7 tienen relación con la labor del señalado en 4.4.2.8, las razones que llevan a la construcción de estos tres son expuestan en la sección **??**, mientras tanto, al igual que los operadores anteriores se detallará el cómo fueron diseñados.

4.4.2.6 Operador removedor de stopword

Éste operador hará uso de una lista de palabras denominadas *stopwords* o palabras vacías, éstas corresponden a palabras sin significado, como artículos, pronombres, preposiciones, etcétera. Éstas palabras serán eliminadas del texto que se está procesando, para ello se utilizará 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'

4.4.2.7 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'

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

4.4.2.8 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.

4.4.2.9 Operador persistencia

Este operador se encargará de conectarse a la base de datos y almacenar el nuevo marcador. Los datos recibidos desde la cadena de procesamiento se llevarán 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 transformará en BSON para almacenarlo en la base de datos en MongoDB.

4.4.2.10 Clasificación

Hasta ahora se han tocado, prácticamente, todos los temas que se relacionan con el funcionamiento del sistema de detección, desde donde se obtendrán los datos, cómo se procesaran e incluso cómo se almacenaran, pero no se ha especificado cómo se realizará la

clasificación, es decir, cómo dado un texto de entrada se conseguirá discriminar en qué categoría encaja. Ésta sección especifica lo que se contextualizó en la sección 4.4.2.8.

El concepto que involucra la construcción de un clasificador es el de ”aprendizaje supervizado”, que fue mencionado en la sección 2.3, en este tipo de aprendizaje se requiere de un conjunto de datos de entrada, denominados conjunto de entrenamiento, que ha de pasar por el algormito, en este caso *Naïve Bayes*, que ha de conocer previamente, la salida esperada para cada elemento del conjunto. El resultado esperado es un clasificador capáz de predecir, en este caso, a qué categoría pertenece un texto sometido a su evaluación.

Según la metodología KDD existen cinco pasos en la búsqueda de conocimiento en bases de datos, estos fueron descritos en la sección 2.4.2.

La selección de datos se ha llevado a cabo. Se cuenta con más de 2000 estados recogidos desde *Twitter* previo, durante y posterior al terremoto de Concepción el año 2010. Éstos datos se han etiquetado por el autor, dándoles la categoría a la que hacen referencia.

El siguiente paso a realizarse según la metodología es realizar un preprocesamiento. Según lo descrito por Manning et al. (2008), se sugiere emplear una serie de técnicas tanto para preprocesar como transformar los datos, éstas son:

- Tokenización: Dada una secuencia de carácteres y una unidad definida (una palabra en este caso), la tarea es cortar esa secuencia en *Tokens* y, a la vez, eliminar ciertos carácteres (por ejemplo, signos de puntuacion).
- Eliminación de stopwords: Son elementos que no aportan a la selección en el proceso de clasificación, usualmente se mantienen en una lista y son eliminados de la secuencia.
- Normalización: Incluye los procesos de llevar todo el texto a minúsculas o mayúsculas, llevar términos a equivalencias, seleccionar el idioma, etcétera.
- *Stemming*: Llevar los términos a una raíz común.

Estas transformaciones han sido descritas por los operadores anteriores con excepción de aquellos que tienen que ver con el filtrado de consultas, la ubicación de *tweets*, el etiquetado propiamente tal y la persistencia.

Para la construcción del clasificador, descrita en la sección 4.4.2, cada conjunto de datos pasará obligatoriamente por todos estos procesos.

4.4.2.11 Topología del sistema

Habiendo definido los elementos de procesamiento, los operadores o *bolts*, se está en condicion de definir la topología. La topología que utilizará 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 emitirán 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.
- *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, continuarán 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 esta posición, pues debe ir previo a la eliminación de *stopwords*, de no ser así ubicaciones, como por ejemplo, ”los vilos”, no serían detectados al realizar la comparativa en éste operador.
- *Bolt* Eliminador de *stopword*: Se realiza previo al *Stemming* para reducir la carga computacional, pues el operador de *stemming* llevaría 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.

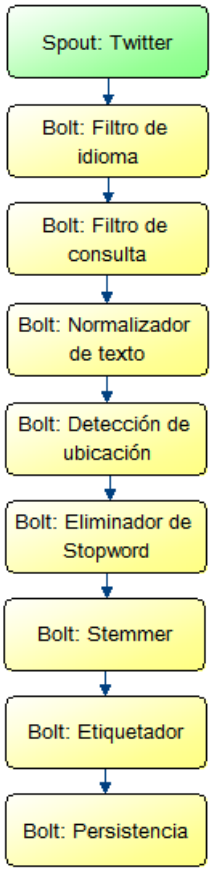

Figura 4.40: Topología general del sistema.

CAPÍTULO 5. EVALUACIÓN DEL SISTEMA

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

5.1 TOPOLOGÍA Y REPLICACIÓN

En la sección 4.4.2.11 se explicitó cómo estarían 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 tendrá un nivel de operadores. Para explicar lo anterior se utilizarán las figuras 5.1 y 5.2 que muestran la implementación del la topología y una esquematización de cómo funcionaría en la peor situación, es decir, cuando el sistema determine que el nivel de replicación debería 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 5.1: Implementación topología de detección de necesidades.

Cada elemento de procesamiento, *bolt*, es instanciado, el valor que acompaña a

cada uno de ellos es el numero máximo de nodos que tendrá el sistema y seguido del modo de agrupamiento, en este caso, *shuffle grouping* para balancear la carga en cada nodo.

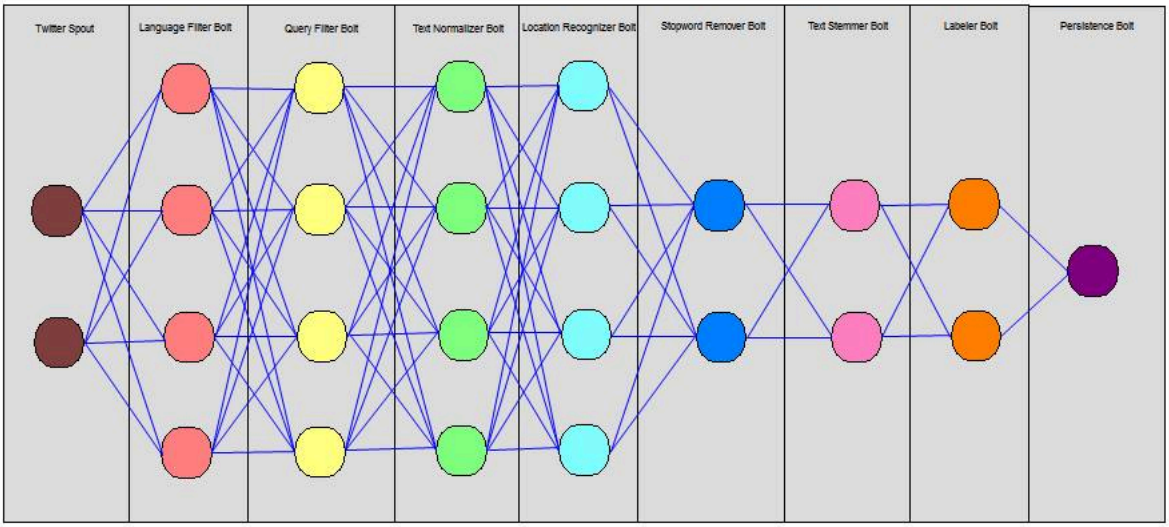

Figura 5.2: Esquema de la topología en el caso de máxima actividad.

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 enviará, *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 para 1000, 2000, 4000 y 8000 estados

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)	513	28999	1713114	38064
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)	908	77259	1415104	46093
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)	1657	47225	2437992	63437
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)	3658	55311	4681626	92825

Con estos resultados se busca definir los balores para la cantidad de nodos por cada nivel de bolts, de la tabla podemos concluir lo siguiente:

- En relación al normalizador, eliminador y stemmer (no se incluyó en la tabla pues su comportamiento es idéntico a las dos anteriores), la entrada y salida es 1 a 1, es decir, por cada elemento que entra, saldrá un elemento. Por ello, y para no originar cuellos de botella, tendrán la misma cantidad de nodos que el *bolt* anterior.
- Para un aumento exponencial de datos, el filtro de idioma es, aproximadamente, constante en permitir el paso del 40% de los estados. Esto quiere decir que el filtro siguiente, debería contener el 40% de los *bolts* del filtro de idioma.
- Para un aumento exponencial de datos, el filtro por ubicación, aproximadamente, permite el paso del 50% de los datos. Es decir, el *bolt* siguiente debería tener el 50% de los nodos que tiene el nivel de ubicación.
- El tiempo de ejecución del filtro de ubicación es el más alto de todos, debido a ello pueden producirse cuellos de botella, es recomendable, entonces, aumentar el número de elementos de procesamiento.

En función de lo anterior, y tomando como solución inicial lo expuesto en la figura 5.2 la configuración de la topología del detector de necesidades quedaría de la siguiente forma:

1. *Spout Twitter*: 1 nodo.
2. *Bolt* Idioma: 4 nodos.
3. *Bolt* Filtro de Consultas: 2 nodos.
4. *Bolt* Normalizador: 2 nodos.
5. *Bolt* Ubicación: 2 nodos.
6. *Bolt Stopword*: 1 nodo.
7. *Bolt Stemmer*: 1 nodo.
8. *Bolt* Etiquetador: 1 nodo.
9. *Bolt* Persistencia: 1 nodo.

5.2 FUNCIONAMIENTO EN ALTO TRÁFICO

CAPÍTULO 6. CONCLUSIONES

placeholder

REFERENCIAS BIBLIOGRÁFICAS

Alvarado, S. (2015). Clasificación.

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

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.
URL https://carta-natal.es/ciudades/Chile/

CHristianCH (2013). Clasificador naïve bayes. ¿cómo funciona? Accedido: 01/07/2016.
URL http://naivebayes.blogspot.cl/

Cica, T. (2000). Características de las redes neuronales. Accedido: 30/06/2016.

Dev.twitter.com (2016). Developer agreement and policy - twitter developers.
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

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

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*.

Han, J., & Kamber, M. (2000). *Data Mining: Concepts and Techniques*. Morgan Kaufmann.

Harwood, P. (2014). *Manifestation of real world social events on Twitter*. Master's thesis, Radboud University, Comeniuslaan 4, 6525 HP Nijmegen, Países Bajos. Máster en Ingeniería de Sistemas e Informática.

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

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

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

Izquierdo, O. J. P., & Díaz, R. C. (2012). Aprendizaje bayesiano. Accedido: 15/06/2016.

Langley, P., & Sage, S. (2013). Induction of selective bayesian classifiers. *CoRR*, *abs/1302.6828*.
URL http://arxiv.org/abs/1302.6828

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.

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

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

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

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

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.

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

Son, N. H. (2006). Data cleaning and data preprocessing.

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/

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.

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

ANEXO A. ANEXO DE EJEMPLO

Cómo obtener claves twitter
Glosario