Minería de opiniones mediante análisis de sentimientos y extracción de conceptos en Twitter

José Javier Martínez Pagés

MÁSTER EN INGENIERÍA INFORMÁTICA

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO FIN DE GRADO EN INGENIERÍA INFORMÁTICA

Madrid, 5 de julio de 2017

Director: Enrique Martín Martín

Convocatoria de junio
Calificación: 9,5

Índice

Índice de figuras

Resumen

Debido al auge de las redes sociales en los últimos años, han surgido diversas técnicas de análisis de sentimientos y minería de datos que les son útiles a las empresas para conocer la opinión pública de sus productos.
Sin embargo, la mayoría de las aplicaciones en el mercado que realizan análisis de sentimientos orientados a productos se centran en la opinión sobre el producto en general, y pocas tratan de buscar por qué las opiniones son positivas o negativas.
Un sistema que descubra cómo se perciben en las redes las características de un producto además de la percepción del producto en sí, también puede ayudar a cualquier organización a detectar la percepción que tienen de ella y por qué.
El objetivo de este trabajo es desarrollar un sistema que, usando el entorno de cómputo distribuido Apache Spark, y mediante técnicas de análisis de sentimientos y de extracción de conceptos consiga averiguar la percepción general de los usuarios de Twitter de los distintos componentes, cualidades y características definidas en una jerarquía de entidades definidas por el usuario.

Palabras clave
Twitter, redes sociales, análisis de sentimientos, minería de datos, extracción de conceptos, Big-Data, Spark, Scala.

Abstract

Due to the rise of social networks in recent years, various methods based on sentiment analysis and data mining have arisen. Companies use these useful methods to find out the public opinion of their products.

However, most market applications that perform sentiment analysis for products focus on the overall view of the product, and only a few try to seek why a product is viewed positively or negatively.

Companies would have an easier time identifying the weaknesses of their products if a system that not only could find out the overall public view of a product, but also capable of finding out the public opinion on the specific characteristics of said product. This would also help any organization to find out the opinion people have of it.

The project objective is to develop a system that, using Apache's Spark distributed computing environment, sentiment analysis and concepts extraction techniques, manages to find out the general perception that Twitter users have on the various components, qualities and characteristics of a hierarchy of user defined entities.

Keywords

Twitter, social networks, sentiment analysis, data mining, concept extraction, Big-Data, Spark, Scala.

Capítulo 1 - Introducción

La finalidad de este capítulo es servir como presentación y toma de contacto con el proyecto, como tal, se pretende:

- Exponer la motivación que llevó al desarrollo de este.
- Fijar los objetivos principales.
- Explicar de manera sencilla y breve los principales conceptos con los que se trabajan.
- Realizar un estudio del estado del arte de estos conceptos y un análisis de la competencia de sistemas similares al que se quiere desarrollar.

1.1 Motivación

El aumento de la disponibilidad y de uso de Internet en los últimos años ha supuesto la aparición de una de las fuentes más fructíferas de información y de opinión en la forma de redes sociales. Aprovechar esta información ha sido uno de los principales objetivos de *marketing* de empresas y organizaciones, tanto para conocer la opinión que se tiene de la organización, sus servicios y sus productos, como para usar las redes sociales para publicitarse y mejorar su imagen.

Esta gran cantidad de información generada ha supuesto el nacimiento de nuevos paradigmas en muchos campos de la Informática actual, tales como la seguridad, la inteligencia artificial, el almacenamiento persistente de datos o el diseño de interfaces gráficas. Como es natural, los nuevos paradigmas y técnicas han ido acompañados de nuevas tecnologías y herramientas que los aplican, formándose así un ecosistema extremadamente variado y de rápido crecimiento.

Es extremadamente relevante conocer este ecosistema y saber identificar qué paradigmas, algoritmos y tecnologías aplicar para resolver el problema. En particular, con este proyecto se pretende aprovechar la información generada en Twitter para conocer la opinión de sus usuarios sobre una jerarquía de elementos, aprovechando los nuevos entornos de computación distribuida para tratar los datos con suficiente velocidad, los modelos de bases de datos no relacionales que consigan un almacenamiento rápido y eficiente, y las técnicas de tratamiento de lenguaje natural que permitan averiguar la percepción de los usuarios.

La mayoría de las aplicaciones en el mercado realizan análisis de sentimientos y monitorizan la percepción sobre ciertos organizaciones y productos, pero pocas buscan identificar la causa de esta percepción. Es decir, estos sistemas averiguan que los nuevos móviles Samsung cuentan con mala opinión, pero pocos buscan saber que esto se debe un mal funcionamiento de la batería.

1.2 Objetivos

Los objetivos que se quieren conseguir en el proyecto son los siguientes:

- Desarrollar una aplicación usando el entorno Apache Spark para conseguir una búsqueda y un tratamiento distribuido y rápido de los tuits. La aplicación recibe una jerarquía de elementos de entrada, busca tuits que contengan alguno de estos elementos, y realizan un análisis lingüístico y de sentimientos del tuit.
- Realizar una identificación de conceptos y un análisis de sentimientos de los tuits. Se deberá detallar aquellas las palabras que aparecen relacionadas con los elementos de la jerarquía y realizar un análisis de sentimientos polar de ellas.
- Guardar la información encontrada en una base de datos, de manera rápida y concurrente. Dicha base de datos debe permitir acceder a los datos de manera remota.
- Crear una aplicación cliente que acceda a la base de datos, muestre la información relevante extraída, permita visualizar cómo ha evolucionado la percepción de los elementos de la jerarquía en Twitter, y en la que se puedan realizar búsquedas sobre la base de datos.

1.3 Conceptos

Los principales conceptos relacionados con el presente Trabajo de Fin de Máster están relacionados con análisis de lenguaje natural, análisis de sentimientos, Big Data, la plataforma Spark, el lenguaje Scala y las bases de datos no relacionales. En este apartado se introducirá brevemente estos conceptos, detallados más adelante en el capítulo 2.

Apache Spark es un sistema *open-source* de ejecución distribuida en *clusters*. Ofrece un *framework* al desarrollador para simplificar el proceso de paralelizar y hacer resistente a fallos los programas escritos. Comenzó a ser desarrollado en la universidad de Berkeley, y después pasó a pertenecer a la Apache Fundation; manteniendo en todo momento los principios que regían su desarrollo: rapidez, sencillez y portabilidad.

El lenguaje Scala está especialmente ligado al proyecto Spark. Scala, tal y como su nombre indica (Scalable language) es un lenguaje de programación creado para ser conciso y escalable. Mezcla las ventajas de la programación imperativa orientada a objetos y la programación funcional; además, permite interoperabilidad con clases de Java, ya que Scala también se ejecuta en la Java Virtual Machine.

Dado que para alcanzar los objetivos del proyecto se requiere la recogida de información generada en Twitter en tiempo real, es hora de introducir el concepto Big

Data. Aunque Big Data es actualmente un término que está en boca de todos, no hay una definición estricta, y se suele aplicar el adjetivo Big Data a toda cantidad de datos que no puede almacenarse en un ordenador convencional. Big Data se suele caracterizar también por sus tres **V**s: gran **volumen** de datos, **variedad** en sus características, y **velocidad** a la que se generan y deben ser procesados.

La ineficiencia de las bases de datos relacionales al tratar problemas de Big Data, ha hecho que cobren importancia varios modelos de bases de datos no relacionales, como las orientadas a documentos, las bases de datos clave/valor, las orientadas a objetos y las basadas en grafos. Estos modelos no cuentan con las restricciones que hacen a las bases de datos relaciones tan robustas, pero esto permite que sean más rápidas. A menudo no garantizan consistencia ni atomicidad, y su diseño se realiza no sólo teniendo en cuenta el modelo de datos real, sino también teniendo en cuenta las consultas permitidas.

Debido a esta gran cantidad de datos generados en Internet, ha cobrado importancia una parte del campo del tratamiento de lenguaje natural, aquella que se refiere a averiguar la actitud subjetiva del autor de un texto con respecto a un tema, también llamado análisis de sentimientos. El análisis de sentimientos se suele clasificar en dos tipos:

- De polaridad: clasifica la intención en dos tipos. La frase suele ser positiva, negativa o neutral.
- Más allá de la polaridad: busca el sentimiento del autor. Es decir, si la frase indica tristeza, alegría, enfado, envidia, etc.

En este proyecto se usará un análisis de sentimientos de polaridad.

1.4 Plan de trabajo

En este apartado describiremos cómo se organizó y estructuró el trabajo a realizar durante el desarrollo del proyecto.

La primera etapa del proyecto consistió en un estudio inicial del estado del arte. Este estudio consta de dos partes:

- Estudio del estado del arte: buscar información sobre técnicas de análisis de sentimientos e identificación de conceptos, en particular artículos de investigación y ejemplos prácticos sobre ambos aspectos y traten información conseguida de redes sociales, u opiniones de usuarios en páginas de venta.
- Estudio de la competencia: identificación de aplicaciones o sistemas ya existentes que apliquen lo que se busca hacer en este proyecto. Identificar sus carencias o debilidades si existen.

A continuación se comenzó el desarrollo en sí del proyecto, que empezó con una toma de contacto con las tecnologías básicas que se van a usar. Esto es:

- Instalación y configuración de Spark y de su biblioteca Streaming. Prueba de su funcionamiento en local y estudio de la funcionalidad que proporcionan.
- Creación de una cuenta de de Twitter con capacidad de crear aplicaciones. Familiarizarse con la estructura de los tuits, API, y restricciones de uso.
- Primer contacto con Spark, Streaming y Twitter: inicialización de la transferencia de datos continua desde Twitter, modos de filtro de los tuits recibidos, tratamiento de los tuits en Streaming y almacenamiento y persistencia de los resultados.

El siguiente paso en el desarrollo del proyecto fue el análisis de los sentimientos y la identificación de adjetivos y modificadores de entidades, esto incluye:

- Investigación más profunda de los distintos métodos de análisis de sentimientos e identificación de identidades.
- Búsqueda de bibliotecas que faciliten la implementación de esta fase.
- Implementación de un prototipo funcional probado localmente con frases sencillas preconstruidas.
- Adaptación de este prototipo a Spark, y realización de pruebas con tuits reales.
- Estudio de los resultados y mejora del prototipo.

Una vez alcanzado el objetivo anterior se procedió a la construcción de una jerarquía que guarde la información extraída de la frase:

- Idear un modelo de datos para su representación.
- Diseñar e implementar un algoritmo que permita usar las entidades dadas y los sentimientos identificados (en este caso particular, las partes de la frase usadas para averiguar el sentimiento) de las distintas frases para formar una única jerarquía.
- Probar el modelo de manera local con frases sencillas preconstruidas, y una vez hecho esto, probar su funcionamiento con tuits reales usando Spark Streaming.

La siguiente fase consistió en desarrollar una base de datos para almacenar los resultados:

- Elección del modelo de base de datos que se va a usar (relacional, orientada a columna, etc.).
- Diseño de la base de datos.
- Modificación del software realizado anteriormente para que almacene los resultados en esta base de datos.
- Creación de procedimientos de limpieza y compactación de la base de datos.

Finalmente, se desarrolló la interfaz gráfica del sistema:

- Investigación de paradigmas o técnicas de usabilidad para facilitar la interacción con grafos.

- Diseñar la interfaz en base a la información encontrada, tratando de que sea consistente con el resto de funcionalidad que tiene que ofrecer la interfaz.
- Implementación de la interfaz.
- Prueba de su funcionamiento.

La tarea de redacción de la memoria se llevó a cabo al mismo tiempo que el trabajo anterior.

1.5 Estado del arte

Al inicio del desarrollo del proyecto se realizó una investigación sobre el estado del arte, junto con los diferentes métodos y técnicas para realizar análisis de sentimientos, minería de opiniones e identificación de aspectos en textos.

Dado que el análisis de sentimientos ha cobrado mucha importancia en los últimos años y se encuentra una gran cantidad de información sobre este tema, en esta sección se incluirá únicamente un resumen el estado del arte respecto a esto, y se dará preferencia a todo aquello relacionado con la "identificación de aspectos"; pues son estas técnicas las que buscan averiguar las características de un elemento a partir del análisis de textos, y suelen incluir una fase de análisis de sentimientos.

En [3] se puede encontrar un resumen completo del estado del análisis de sentimientos a día de hoy. En el texto se indica que para la resolución de este problema se puede usar un enfoque basado en el Aprendizaje Automático, usando algoritmos de aprendizaje supervisado (difícil por la falta de ejemplos etiquetados) o algoritmos de aprendizaje no supervisado. Es posible también enfocar la solución del problema mediante análisis del lenguaje natural, aunque resulta más complejo debido al esfuerzo extra que hay que realizar para tratar los casos específicos de conjunciones, disyunciones, etc. y la gran dificultad de detectar ironías, sarcasmos y bromas.

En [4] también se puede encontrar una explicación más detallada del estado del arte del análisis de sentimientos, pero lo que es más importante, en el apartado 5.3 y 5.4 del artículo se habla sobre la "extracción de aspectos". La "extracción de aspectos" se refiere precisamente a encontrar los adjetivos, sustantivos y adverbios de la frase relacionados con el sentimiento encontrado. En el artículo se propone usar las relaciones sintácticas entre las palabras que otorgan un sentimiento a la frase y el resto de palabras para identificar los aspectos y extraerlos. Se expone también que una vez identificados los aspectos se pueden usar estos para encontrar más sentimientos.

Esta identificación de aspectos se trata también en [5], aunque se centra en buscar por qué un producto falla, y se denomina "identificación de debilidades". El

artículo trata de identificar las características que definen a un producto y encontrar la opinión que se tiene sobre ellas. Para la identificación usan un algoritmo de aprendizaje automático supervisado y un diccionario para buscar sinónimos. Los sentimientos de las características se extraen usando análisis de sentimientos, y comparación cruzada entre otras características ya encontradas. Hay que recalcar que la solución adoptada por los autores contiene una fase de etiquetado manual, y no se encuentra cuáles son los motivos por los que un producto falla, por ejemplo, se consigue que la batería de una cámara tiene mala puntuación, pero no se averigua si es por poca duración, porque necesita un largo tiempo para cargarse, porque se calienta en exceso, etc.

Se tratan otros métodos sobre la identificación de aspectos de un producto en [6]. Los autores usan un método de cuatro fases para la identificación: en el primero se identifican los sustantivos que aparecen comúnmente en la misma frase que el producto; en la segunda fase se identifican adjetivos y se eliminan de la lista de sustantivos anterior aquellos que no suelen ir cerca de adjetivos o de otros sustantivos no comunes; tras esto se establecen relaciones entre adjetivos y adverbios (por si se omite el sustantivo o el verbo) dependientes del dominio; y en el último paso se eliminan los sustantivos no importantes usando la medida PMI-IR [7]. Este método no es explícitamente un algoritmo de aprendizaje automático, pero en [6] se menciona que se alimentó al algoritmo en su primera fase con frases correctamente construidas que contenían información veraz, por lo que no es seguro que el algoritmo funcione de la misma manera con frases incorrectas o que contengan información falsa, suponiendo esto que el algoritmo necesita de un cierto "entrenamiento". Además, para el tercer paso del algoritmo ha hecho falta relacionar adjetivos y adverbios para lo cual era necesario conocimiento del dominio.

Asimismo, se realizó también un estudio de sistemas ya existentes en el mercado, con el objetivo de ver la funcionalidad que ofrecen y así tratar de diferenciar este proyecto de la competencia.

El primer sistema encontrado fue **Sproutsocial** [8], un software de manejo de redes sociales. Sproutsocial ofrece una gran cantidad de funcionalidad para gestionar la opinión de una empresa en las redes sociales, y permite programar la publicación de mensajes, realizar un seguimiento de hashtags, cuantificar el tiempo de respuesta a menciones, etc. La funcionalidad más parecida que tiene a la que se busca en este proyecto, es que permite realizar un seguimiento de palabras clave y de *hashtags* en Twitter, realizando también un análisis de sentimientos del tuit en el que aparecen.

En la misma línea está **Falcon.io** [9]. La funcionalidad es similar a Sproutsocial, pero a diferencia de Sproutsocial, Falcon permite construir consultas personalizables para buscar palabras, temas y frases en Twitter, analizar sus sentimientos y organizar los resultados por tema, región o grupos de audiencia personalizables.

Oracle también ha desarrollado su propio producto para facilitar la gestión de redes sociales, **Oracle Social Cloud** [11]. Pertenece al mismo orden que los sistemas mencionados antes, con una diferencia importante para este proyecto, Oracle Social Cloud acepta una lista de palabras sobre las que busca su sentimiento en Twitter, pero no sólo su sentimiento, si no también los adjetivos que más aparecen directamente relacionados con la palabra buscada y la frecuencia con la que aparecen.

Existen muchos otros programas parecidos con funcionalidad similar, como **Agorapulse** [11], **Twazzup** [12], **Lithium** [13] o **Adobe Social** [14]. Respecto al análisis de sentimientos y la identificación de características son también similares a los programas anteriores, y se centran en realizar análisis de sentimientos de palabras o frases que introduce el usuario, no suelen hacer identificación de características, y solo suelen guardar adjetivos directamente relacionados con las palabras a buscar.

La mayoría de los sistemas mencionados (con la excepción de Oracle Social Cloud y Adobe Social) siguen un modelo de negocio *open-source*, aunque los sistemas no son completamente abiertos, sino que solo hay libres ciertos componentes de estos sistemas. Entre estos componentes abiertos, se ha podido observar que usan entornos de computación distribuidos como Hadoop, admitiendo ejecución en la nube de Amazon mediante Amazon Web Services, y usan tano bases de datos relacionales tradicionales como SQL como no relacionales (Apache Cassandra o MongoDB).

Chapter 1 - Introduction

The purpose of this chapter is to serve as a presentation and a first contact with the project, as such, it includes:

- Motivation that led to the development of the project.
- Main objectives.
- Brief and simple explanation of the main concepts this project works with.
- State of the Art and competence analysis of the similar systems that can already be found in the market.

1.1 Motivation

The increase in the availability and use of the Internet in recent years has meant an emergence of one of the most fruitful sources of information and opinion in the form of social networks. Taking advantage of this information has been one of the main marketing objectives of companies and organizations, to know the opinion of the organization, its services and products, and to use social networks to be known and improve their image.

This large amount of information has led to the emergence of new paradigms in many fields of current Computer Science, such as computer security, Artificial Intelligence, persistent data storage or the design of graphical interfaces. Naturally, the new paradigms and techniques have been accompanied by new technologies and tools that apply them, creating an extremely varied and fast growing ecosystem.

It is extremely important to know about this ecosystem and to know what paradigms, algorithms and technologies to apply to solve a specific problem. In particular, this project aims to take advantage of the information generated on Twitter to find the opinions of its users about a hierarchy of elements; Taking advantage of new distributed computing environments to be able to deal with the generated data with sufficient speed, non-relational database models to achieve fast and efficient storage, and natural language processing techniques that allow to extract the opinion of Twitter users.

Most applications on the market perform sentiment analysis and monitor the perception of certain organizations and products, but few seek to identify the cause of the sentiment found. That is, these systems find out that the new Samsung phones have a bad image, but few are looking to know that this is because of a battery malfunction.

1.2 Objectives

The project objectives are:

- Creating an application that runs in the Apache Spark environment. The application receives input files that contain a defined hierarchy of elements. The application is connected to Twitter and receives a stream of tweets, which must contain some of the elements defined in the hierarchy.
- Perform an identification of concepts and a sentiment analysis of the tweets. It will find the words that appear modifying the elements of the hierarchy and t it will make a polar sentiment analysis of them.
- Save the extracted information in a database, in a fast and concurrent way, also allowing to access the data remotely.
- Create a client application that accesses the database, shows the relevant information extracted, allows you to visualize how the perception of the elements of the hierarchy in Twitter has evolved, and also allowing to search the database.

1.3 Concepts

The main concepts needed to understand this thesis are related to natural language analysis, sentiment analysis, Big Data, the Apache Spark platform and the Scala language. A small introduction to said concepts will be included in this section so that reading the memorandum might be easier.

Apache Spark is an open-source system based on distributed execution in clusters. It provides a framework for the developer to simplify the process of parallelizing and writing fault-tolerant programs. Its development was started by the University of Berkeley; tough later Apache Foundation got ownership behind the project, the ideology behind it persisted: speed, simplicity and portability.

The Scala language has a special connection with Apache Spark. The Scala language, just like its name indicates (Scalable language) is a programming language designed and made with the whole purpose of being simple, concise and scalable. It mixes the advantages of object oriented imperative programming and functional programming; additionally, it allows interoperability with Java classes, as Scala also runs on the Java Virtual Machine.

Since achieving the project objectives requires collecting information generated on Twitter at real time, it is time to introduce the concept behind Big Data. Even though Big Data has recently become a trending term, there is not an objective and strict way to define it. Big Data is usually used to refer to any amount of data that can not be stored on a conventional computer. Another way to describe Big Data is using the famous three **V**s: large data **volume**, characteristic **variety**, and **velocity** at which data is generated and must be processed.

The inefficiency of database relationships when dealing with Big Data problems is the main reason why many different kinds of nonrelational databases have become important. These nonrelational databases do not have the constraints that make relational databases so robust, but in exchange allows them to be faster. They often do not guarantee consistency, and usually their design does not only take into account the actual data model, but also the different queries the final program will have.

Due to the large amount of data generated over the Internet, a specific part of the natural language processing field has grown in importance, the one that tries to find the subject opinion of an author over the piece of text he has written, also called sentiment analysis. Sentiment analysis is usually classified into two types:

- Polar: classifies the intention into two types. The phrase is either positive, negative or neutral.
- Beyond polarity: looks for the author's feelings. That is, if the phrase indicates sadness, joy, anger, envy, etc.

This project will use polar sentiment analysis.

1.4 Work plan

The following section will explain how the work plan for the project was organized.

The first stage of the project consisted of an initial study over the state of the art. This study was done in two parts:

- Study of the state of the art: look for information on both sentiment analysis and feature identification, putting particularly interest on texts that contain information about both topics. Also, focusing on what has been done, how it has been done, and use this information to carry out this project and expand the state of the art.
- Competition study: identification of existing applications or systems that already do what this project intends to, while also identifying their deficiencies or weaknesses if they do exist.

Then the development of the project itself began, starting with an initial contact with the basic technologies that were going to be used. That is:

- Doing an installation and a configuration of Spark and the Streaming library, testing their local behavior and studying the functionality they provide.
- Making a Twitter account with enough rights to manage Twitter applications. Studying the structure tweets have and the Streaming library's API, and its usage restrictions.
- First contact with Spark, Streaming and Twitter working together:initializing the continuous data transfer from Twitter, studying the different ways of filtering the receiving tweets, treating the tweets and storing the results.

The next step focused in sentiment analysis and the identification of adjectives and modifiers of entities, that includes:

- Deeper investigation of the different methods of sentiment analysis and entity identification.
- A search for libraries that help to implement the above.
- Implementation of a locally tested functional prototype with simple pre-built phrases.
- Adaptation of this prototype to the Spark platform, and testing it with real tweets.
- Studying the results and improving the prototype.

Once the previous objective had been achieved, the next step was to build a hierarchy capable of storing the information analyzed in the sentence:

- Design a data model that represents the hierarchy.
- Design and implement an algorithm that allows to use the given entities and the identified feelings of the different phrases to form a single hierarchy (particularly, the specific parts of the phrase used to find the author's opinion).
- Test the model locally with simple pre-built phrases, and once that is done, test the program with actual tweets using Spark Streaming.

The next step was to develop a database that stores the previous hierarchy:

- Selecting the database model to be used (relational, column oriented, etc.).
- Designing the database.
- Modify the previous software to store the results in this database.
- Create procedures for cleaning and compacting the database.

Finally, the graphical interface was developed:

- Research of usability paradigms or techniques that facilitate user interaction with graphs.
- Design the interface based on the information found, trying to make a consistent design with the rest of the functionality that the interface has to offer.
- Implementation of the graphical interface.
- Test its behavior.

The memorandum will be written at the same time as the previous work is being done.

1.5 State of the art

An investigation on the state of the art was carried out at the beginning of the project's development, along with a study over the different methods and techniques of performing sentiment analysis, opinion mining and feature extraction.

Since sentiment analysis has become so important in recent time, and given that there is a great deal of information about it, this section will cover only a summary about sentiment analysis. The focus of this section will be put in articles related to feature extraction, as this technique goal is to extract the features of an entity using text analysis.

[3] contains a complete summary of the state of the art on sentiment analysis as of today. The text speaks about the different approaches to the problem, it can be done through Machine Learning, either using supervised algorithms (not easy because of the lack of labeled examples) or unsupervised algorithms. It is also possible to use a natural language analysis approach, although it is certainly more complex due to the extra effort necessary to deal with the specific cases of conjunctions, disjunctions, etc. and the great difficulty of detecting ironies, sarcasms and jokes.

At [4] a more detailed explanation over the state of the art of sentiment analysis can also be found, but the main reason this article is here, is because sections 5.3 and 5.4 of the article deal with feature extraction. "Feature extraction" refers precisely to finding the particular parts of the phrase related to the feeling found. In the article the author uses the syntactic relations between the words responsible of the phrases' sentiment and the rest of words to identify the features and extract them. It is also said that once the features are identified, they can be used to find more opinions.

Feature identification is also addressed in [5], although it focuses specifically on finding out why a product fails, calling it "weakness finder". The author tries to identify the features that define a product and to find the opinions related to them. For the feature identification a supervised machine learning algorithm is used, with the help of a synonym dictionary. Features opinions are extracted using sentiment analysis and crossing information with the other features already found. It should be emphasized that the solution adopted by the authors contains a manual labeling phase, and really it doesn't find why the product is failing, it does find that the battery of a camera model is poorly rated, but it doesn't find that the rating is low because either the battery has a short duration, or because it needs a long time to charge, or if is because the battery overheats, etc.

Other methods used for feature identification are discussed in [6]. The authors use a four-phase method to identify the features: the first method identifies the common nouns that usually appear in the same sentence as the name of product appears; In the second phase, adjectives are identified, and then those nouns that do not usually

appear near adjectives or other nouns are eliminated from previous list of nouns; After this is done, it is necessary to do a mapping between adjectives and adverbs (because the verb or the noun can be omitted) dependent on the domain; And in the last step we remove the non-important nouns using the PMI-IR measure [7]. This method is not explicitly a machine learning algorithm, but it is mentioned in the text that for the first step they fed the algorithm some sentences known to be correct, so in a way it seems that it is necessary to train the algorithm. In addition, for the third step of the algorithm it is necessary to do a manual mapping that needs some knowledge of the domain.

Also, a study of software projects already available in the market was carried out, in order to see the functionality they offer and thus to try to differentiate this project from the competition.

The first system found was Sproutsocial [8], a social network management software. Sproutsocial offers a large amount of functionality for easily managing the opinion of your company in social networks, it allows scheduling the publication of messages, tracking hashtags, measuring the time your team take to answer, etc. It also does something similar to the intended functionality of this project, that is tracking a user defined set of keywords and hashtags on Twitter and doing a sentiment analysis of the tweet in which they appear.

There is also a similar software called Falcon.io [9]. Its functionality is similar to Sproutsocial, but if we focus on the theme of this project, Falcon maintains certain differences from Sproutsocial. Falcon allows you to build customizable queries to search for words, themes and phrases on Twitter, can analyze the found sentiments and organize the results by theme, region or by customizable audience groups.

Oracle has also developed its own product to facilitate social networking management, Oracle Social Cloud [11]. It belongs to the same group as the systems mentioned above, with an important difference, Oracle Social Cloud accepts a list of words to look for on Twitter and analyzes its sentiment and stores the adjectives that appear directly related to the searched word and the frequency with which they appear.

There are many other similar programs, such as Agorapulse [11], Twazzup [12], Lithium [13], Adobe Social [14], etc. Regarding sentiment analysis and feature extraction they do similar to the previous programs, and focus on performing sentiment analysis of words or phrases that the user introduces, but do not usually apply feature extraction, and only usually save adjectives that are directly related to the words to had to be looked for.

Most of the above systems (with the exception of Oracle Social Cloud and Adobe Social) follow an open-source business model, although the systems are not

completely open, but rather only certain components of these systems are. Among these open components, it has been observed that they use distributed computing environments such as Hadoop, supporting Amazon cloud execution using Amazon Web Services, and use traditional relational databases such as SQL and non relational (Apache Cassandra or MongoDB).

Capítulo 2 - Propuesta Software

En este capítulo se pretende incluir una descripción del software construido para el proyecto, comenzando con una visión general del mismo, siguiendo con una explicación de las tecnologías usadas, y finalmente descripciones de las tres aplicaciones distintas que forman el sistema.

2.1 Visión general

El sistema desarrollado cuenta con tres partes diferentes que resuelven tareas distintas. Estas distintas partes no necesitan ser instaladas en la misma máquina, tan solo tener acceso a la base de datos común:

1. Aplicación Spark que recoge tuits en tiempo real que contengan alguna de las palabras de la jerarquía de entrada, los analiza sintácticamente, realiza un análisis de sentimientos y guarda el sentimiento y las palabras relacionadas con los elementos encontrados en los tuits en la base de datos.
2. Conjunto de varios procesos de ejecución periódica que se encargan de limpiar información no relevante de la base de datos y de compactar su información.
3. Aplicación cliente que cuenta con una interfaz gráfica para leer la información de la base de datos y mostrarla al usuario. Permite visualizar la jerarquía de entrada y las palabras extraídas en forma de grafo, obtener gráficas con la evolución de la opinión de los distintos elementos de la base de datos, y realizar búsquedas complejas sobre ésta.

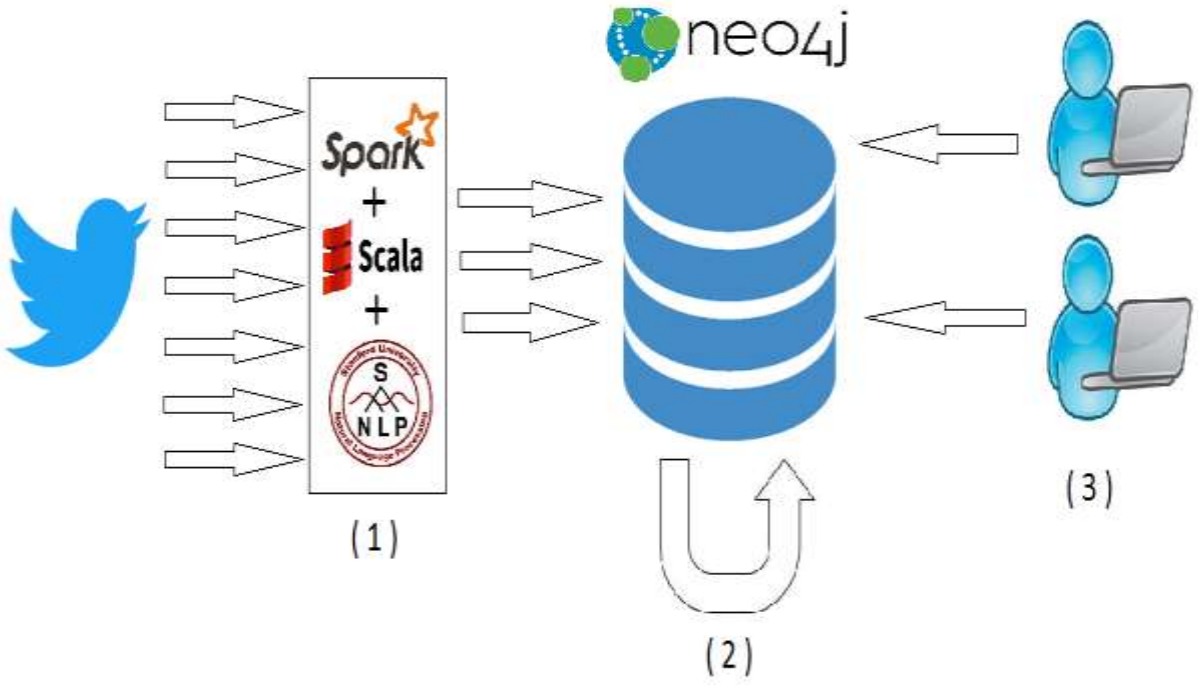

Figura 2.1: Visión general del sistema

2.2 Tecnologías utilizadas

2.2.1 Spark

La introducción a Apache Spark ya se ha realizado en la sección 1.3, así que en este apartado se explica su uso concreto en este proyecto, y las ventajas y limitaciones que ha supuesto.

Spark ofrece un modelo de cómputo basado en RDDs (*Resilient Distributed Datasets*), colecciones inmutables de objetos divididos en varias particiones lógicas, de tal manera que se pueda operar con estas particiones de manera concurrente en distintos nodos de cómputo. A diferencia de Hadoop y del modelo *MapReduce*, Spark tiene un mayor soporte de operaciones iterativas, pretendiendo solucionar el problema que ocasiona *MapReduce* al estar enfocado a operaciones de agrupamiento y reducción. Esto se consigue guardando los resultados de las operaciones en una memoria distribuida, en vez de guardarlo en un sistema de ficheros HDFS.

Este sistema se adapta mejor a lo requerido en el proyecto que otros sistemas como Hadoop, puesto que no hay que realizar operaciones de agregación ni de reducción, pero sí hay que filtrar los datos de entradas y transformarlos de manera iterativa hasta conseguir lo que se desea guardar en la base de datos.

Se ha aprovechado la API Streaming de Spark, que proporciona una extensión de Spark para facilitar el procesamiento de flujos. A efectos del programador, el tratamiento de los distintos flujos se realiza de manera muy parecida al tratamiento de listas, y se pueden definir operaciones de agregación, de reducción, de filtrado o de transformación de los flujos. El parecido que guardan los flujos con una estructura tan conocida como es una lista, permite en gran medida hacer el desarrollo de manera cómoda para alguien que no tenga grandes conocimientos de programación distribuida, pero aún así hay que tener en cuenta que las operaciones de transformación y de filtrado de los flujos se hacen en paralelo, por lo que hay ciertas restricciones con el tipo de clases que se pueden usar dentro de estas operaciones.

Este paralelismo provoca que cada tuit debe ser tratado individualmente e impide que la información extraída se guarde en una estructura de datos común, a no ser que esta esté preparada para atender peticiones concurrentes. Este hecho no ha supuesto grandes limitaciones, ya que la estructura que se modifica, en este caso, es una base de datos no relacional, que fue elegida entre otros motivos, por su capacidad de soportar lecturas y escrituras concurrentes. Se hablará de esto más en profundidad en la sección 2.2.3.

Spark Streaming cuenta con una parte de su API dedicada a recoger un flujo de tuits, e incorpora mecanismos a través de Twitter4J [15] para poder realizar la

autenticación de la aplicación en Twitter y establecer los parámetros necesarios. Como ya ha sido mencionado en la sección 1.2, fue necesario crear una cuenta de Twitter y crear una aplicación de Twitter.

2.2.2 Stanford CoreNLP

El Stanford Natural Processing Language Group [16] es un equipo de la universidad de Stanford que investiga el tratamiento computacional del lenguaje natural, y su aplicación en tecnologías. Uno de los productos de su investigación es la biblioteca para Java Stanford CoreNLP [17], cuya primera versión fue publicada en Noviembre del 2011 y con la que han seguido trabajando hasta ahora. La versión usada en este proyecto es la 3.7, correspondiente a Octubre de 2016.

Esta biblioteca proporciona una amplia variedad de herramientas con las que realizar tratamiento de lenguaje natural en seis idiomas: inglés, español, francés, alemán, chino y árabe. Entre la funcionalidad usada en este proyecto se incluye el reconocimiento de entidades, análisis morfológico de palabras, análisis sintáctico, establecimiento de dependencias entre palabras de una misma frase y finalmente, el análisis de sentimientos de frases y palabras.

Especialmente útil ha sido el establecimiento de dependencias entre palabras, ya que así se consigue en este proyecto encontrar las palabras relacionadas con aquellas buscadas, y así poder luego encontrar su sentimiento.

Es necesario decir que la biblioteca cuenta también con sus limitaciones, y el reconocimiento de entidades no es muy preciso a la hora de buscar entidades cuyos nombres son también sustantivos comunes (ocurre a veces con nombres de determinados móviles, como en “Samsung Galaxy Note”) o cuando se identifican por números (por ejemplo en “iPhone 6”). El funcionamiento del analizador de sentimientos de la biblioteca de Stanford tampoco es perfecto: su analizador usa un modelo entrenado por un algoritmo de aprendizaje automático, y debido a este entrenamiento se obtienen ciertas situaciones no deseadas, como el verbo “like” que tiene asociado un sentimiento neutro, y el verbo “liked” (en pasado) tiene asociado un sentimiento positivo. También ha influido en el proyecto el hecho de que algunas búsquedas realizadas por la biblioteca entre cadenas de texto son sensibles a mayúsculas y minúsculas, por lo que la conversión de una cadena a minúsculas es necesaria en ciertas partes del proyecto.

2.2.3 Neo4J

Neo4J [18] es una base de datos nativa no relacional orientada a grafos. Es altamente escalable, rápida en sus transacciones y cuenta con un control de

transacciones que permite escrituras concurrentes. Neo4J permite almacenar la información en nodos y en aristas mediante diccionarios clave-valor.

Existen restricciones en cuanto a los tipos que los diccionarios admiten: se pueden guardar cadenas de texto, números enteros, decimales y arrays cuyos elementos sean del mismo tipo. No se puede almacenar, por ejemplo tuplas, listas de tuplas o que un elemento del diccionario sea otro diccionario.

Neo4J admite consultas orientadas a grafo, de tal manera que se pueden buscar nodos o aristas no sólo según sus atributos, sino también en función de los caminos formados entre los nodos y las aristas. Las consultas se realizan mediante el lenguaje de consultas Cypher, un lenguaje que aprovecha la familiaridad de SQL y el arte ASCII para facilitar las consultas sobre el grafo.

Neo4J no solo proporciona una representación de los datos en la base de datos similar a la usada fuera de ella, si no que además está construida para poder tratar problemas Big Data; la rapidez con la que se procesan las transacciones y la posibilidad de que sean concurrentes permite que sea totalmente adecuado para usarse desde una aplicación Spark.

Esta necesidad de resolver las consultas de manera rápida obliga a Neo4J a incorporar compatibilidad concurrente a las operaciones de lectura y escritura. Existen ciertas instrucciones Cypher que aseguran que no se van a producir conflictos entre las consultas concurrentes bloqueando solo los nodos implicados, de tal manera que el resto de consultas no peligrosas pueden seguir realizándose a pleno rendimiento. Estas instrucciones son *MERGE* (actualiza un nodo con nueva información si existe y si no lo crea), *CREATE UNIQUE* (crea un nodo si no existe y si existe sobreescribe sus atributos) y *SET* (actualiza un nodo).

También dispone de una capacidad de almacenamiento muy alta, y aunque teóricamente a partir de la versión 3.0 de Neo4J se pueden almacenar un número ilimitado de nodos, fuentes externas a Neo4J cifran en 10^{15} el número máximo de nodos [19], donde cada nodo puede ocupar 4 GB de memoria [20]. Estas limitaciones en cuanto al espacio también se tuvieron en cuenta a la hora de diseñar el sistema.

2.2.4 Anormcypher

Anormcypher [21] es una biblioteca Scala para Neo4J. Creada para seguir el modelo de la biblioteca Anorm [22], la biblioteca proporciona una API para usar Cypher, realizar consultas y obtener sus resultados en tipos admitidos por Scala.

Al igual que Anorm ofrece una API para volver al uso de texto SQL plano en programas Java, sin complicar el guardado de la base de datos con traductores

automáticos entre objetos y tablas SQL; Anormcypher ofrece una API simple con la que conectarse a una base de datos Neo4J y ejecutar consultas en texto Cypher plano.

La biblioteca actúa como interfaz para el programador entre el programa Scala y la base de datos, y no ha supuesto restricciones para el proyecto.

2.2.5 JavaFX

Para la parte gráfica de la aplicación se ha usado JavaFX [23]. JavaFX es la nueva biblioteca gráfica de Oracle para Java, y esta vez ofrece una mayor separación ya integrada entre la interfaz, el controlador y el modelo; y permite además el uso de archivos CSS para definir la interfaz.

Las aplicaciones JavaFX no ejecutan el código de una clase Main como es lo común de las aplicaciones Java, sino que ejecutan el código estático del método start de una clase que herede de la clase Application; esto altera en cierta medida la organización del código de la parte gráfica de este proyecto.

JavaFX también dispone de un componente gráfico denominado WebView, que permite visualizar páginas web. Este componente está basado en WebKit [24], soporta CSS, DOM, HTML, JavaScript, y permite que se realicen tareas propias de navegador tales como ver el historial o ejecutar comandos JavaScript. Por todos estos motivos se usa en el proyecto, sin embargo no es tan potente como los navegadores web usuales.

2.2.6 Alchemy.js

Alchemy.js [25] es una aplicación web para la visualización de grafos. La aplicación funciona con JavaScript, CSS y HTML, y se ha usado en este proyecto en la interfaz gráfica para visualizar el grafo cargado desde la base de datos.

Dado que Alchemy.js es una aplicación web y el proyecto actual está implementado en Scala, se ha usado el componente WebView de JavaFX para poder visualizar la página web generada por Alchemy.js. La falta de potencia de este componente en comparación con un navegador web usual es notoria en este caso, y restringe el tamaño del grafo que se quiere visualizar. Como ejemplo ilustratrivo, en las pruebas realizadas el componente funcionó bien hasta el orden de los siete mil nodos, a partir de este número el componente mostró una pantalla en blanco; sin embargo, si ese mismo grafo se abría mediante el navegador Google Chrome el grafo se mostraba correctamente.

La configuración de Alchemy.js se realiza mediante CSS y JavaScript, y para poder realizarla mediante el proyecto Scala se han tenido que escribir la configuración en forma de un objeto *String* de Scala. El grafo a pintar tiene que estar escrito en formato JSON, y debe estar presente como variable en un archivo HTML concreto; esto significa que en tiempo de ejecución el grafo debe traducirse a JSON y luego escribirse éste en el archivo HTML.

Esta tarea no conlleva una gran carga computacional, y tal y como se puede ver en la sección 3.4, el tiempo acumulado de la escritura de los archivos y el dibujo del grafo suele mantenerse relativamente constante respecto al tiempo que se tarda en realizar las consultas para conseguir los datos que hay que mostrar. Además, en la sección 2.5.2 se establece que la interfaz gráfica dibuje como máximo diez palabras por nodo de la jerarquía, por lo que la complejidad de estas operaciones crece de manera lineal respecto al tamaño de la jerarquía de entrada.

2.3 Aplicación Spark

En este apartado se explicará la arquitectura y las clases que componen la aplicación Spark que recoge tuits, realiza un análisis de sentimientos y guarda la información en la base de datos.

2.3.1 Representación de la jerarquía

Dado que el objetivo general de la aplicación es encontrar información de los elementos que forman una jerarquía de entidades definida por el usuario, la primera tarea que se ha de realizar es diseñar un modelo de datos que represente esta jerarquía. La jerarquía mantiene una estructura de árbol, pero debe permitir que los nodos tengan un número distinto de hijos entre sí y que además los nombres de estas entidades se repitan entre nodos sin relación directa de distintas ramas de la jerarquía.

Se puede encontrar un ejemplo válido de jerarquía en la Figura 2.2.1. En esta figura se puede ver cómo el nodo “Samsung” tiene un solo hijos mientras que el nodo “Galaxy” tiene tres. También se puede observar cómo existen dos nodos con nombre “1”, primos entre ellos.

Debido a estas restricciones, no es posible usar una de las estructuras de árboles ofrecidas en las bibliotecas de Scala, ya que todas las clases que implementan a la clase *Tree* necesitan que no se repitan las claves; así que se creó un objeto *Jerarquia* propio. Este objeto contiene un atributo, una tabla hash que usará de clave el nombre de la entidad, y como valor una tupla (detalla más adelante) que contiene entre sus elementos una referencia a otro objeto *Jerarquia*. Esta definición recursiva permite conseguir la representación de la jerarquía que buscamos. De esta forma, la jerarquía anterior estaría representada como en la figura 2.2.2.

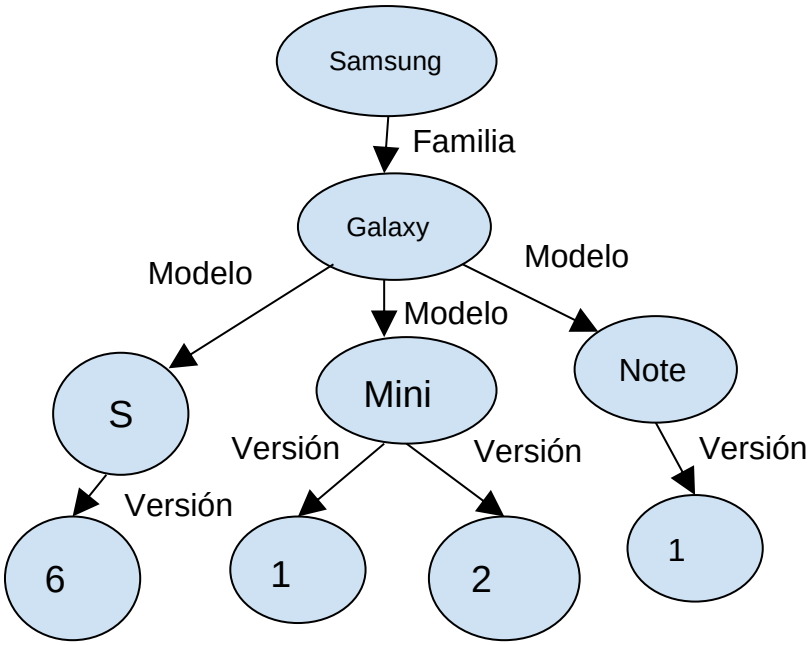

Figura 2.2.1: Ejemplo de jerarquía con elementos repetidos

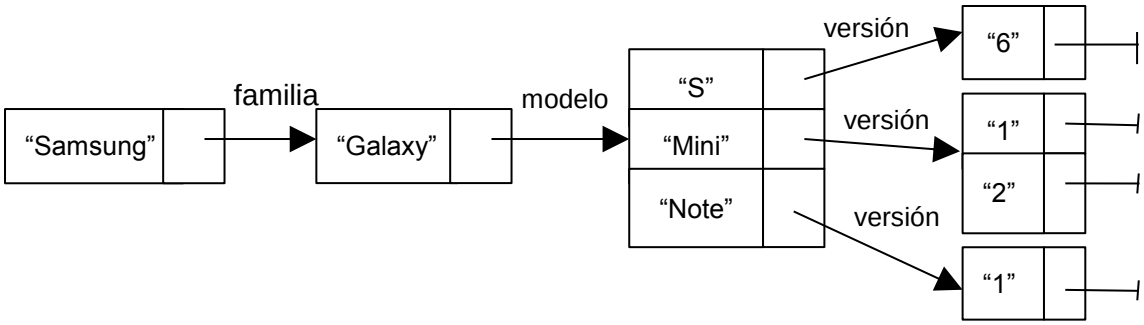

Figura 2.2.2: Ejemplo de jerarquía guardada en una tabla hash

Este objeto no solo tiene como objetivo guardar la jerarquía de entrada del usuario, sino también la jerarquía resultante al conseguir en los tuits las palabras relacionadas con cada entidad y su sentimiento.

Las palabras encontradas relacionadas con cada entidad se pueden guardar de la misma forma que los otros elementos de la tabla hash, sin embargo, hace falta distinguir estas palabras de las entidades. Para realizar esta distinción se usará una cadena de texto que contiene el nombre de la relación, es decir, en el caso del objeto *Jerarquia* que contiene como única entrada en su tabla hash "Galaxy", esta cadena de texto tendría el valor "Familia", ya que este es el nombre de la relación establecido por el usuario. En el caso de las palabras encontradas al analizar las frases, se usará como nombre de la relación "modificador", para así distinguirlas del resto de entidades.

Como también se debe hacer un análisis de sentimientos, es necesario guardar el sentimiento encontrado en cada nodo nodo de la jerarquía. Para almacenar la puntuación se ha añadido otro elemento a la tupla, una referencia a un objeto de clase *Puntuacion*. Esta clase contiene tres atributos, uno que indica el número de veces que se ha encontrado la palabra al analizar el tuit, un número entero que indica el sentimiento asociado al nodo del árbol, y otro número entero que contiene la media de los sentimientos de los hijos que no son modificadores.

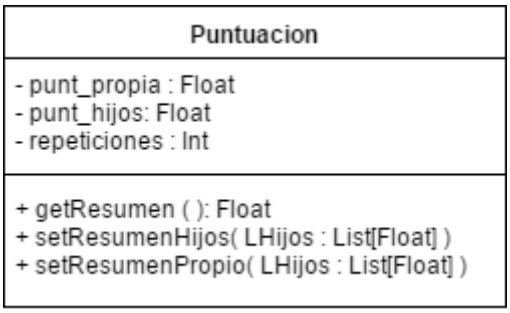

Figura 2.2.3: Clase Puntuacion

Esta clase también posee métodos para conseguir la denominada "puntuación resumen", es decir, la media entre la puntuación propia del nodo y la puntuación de sus hijos. La clase también ofrece métodos para cambiar la puntuación de sus hijos en función de una lista con las puntuaciones de sus hijos entidades, y un método para cambiar la puntuación propia de la entidad a partir de una lista con las puntuaciones de sus hijos no entidades.

Además de la referencia a un objeto *Jerarquia*, la cadena de texto con el nombre de la relación y la referencia al objeto *Puntuacion*, hacen falta dos elementos más en la jerarquía: una cadena de texto que indique si la entidad es débil respecto a su padre, y una cadena de texto que guarde un identificador para la entidad. La necesidad de estos elementos se explica en el apartado 2.3.2.

2.3.2 Representación del modelo de datos para el tratamiento del texto

Aunque el modelo anterior de jerarquía es útil tanto para guardar la jerarquía definida por el usuario, como para guardar la jerarquía definitiva una vez se ha hecho el análisis del texto, no lo es para el paso intermedio en el que se realiza el análisis. Sería lento modificar la jerarquía original cada vez que se encuentra en la frase una palabra relacionada con una entidad de la jerarquía y tener que recalcular su sentimiento; es más rápido encontrar todas las palabras relacionadas, guardarlas en una estructura auxiliar, y finalmente modificar la jerarquía. De este modo se puede calcular el sentimiento de la entidad como la media de los sentimientos de las palabras encontradas relacionadas con él.

La clase que se encarga de guardar estas estructuras de datos necesarias, así como de realizar el post-procesado para modificar la jerarquía original con la nueva información, es la clase *ModeloBajoNivel*. Esta clase necesita de cuatro atributos: una referencia a la jerarquía original, una lista con los intereses definidos por usuario (si le interesa la batería de los móviles, o las pantallas, etc.), una tabla en la que se almacenan las palabras encontradas al analizar el texto, y una referencia a un objeto de la clase *Sentimientos* que se encarga de realizar el análisis de sentimientos y de asignar una puntuación según sea este positivo, negativo o neutro.

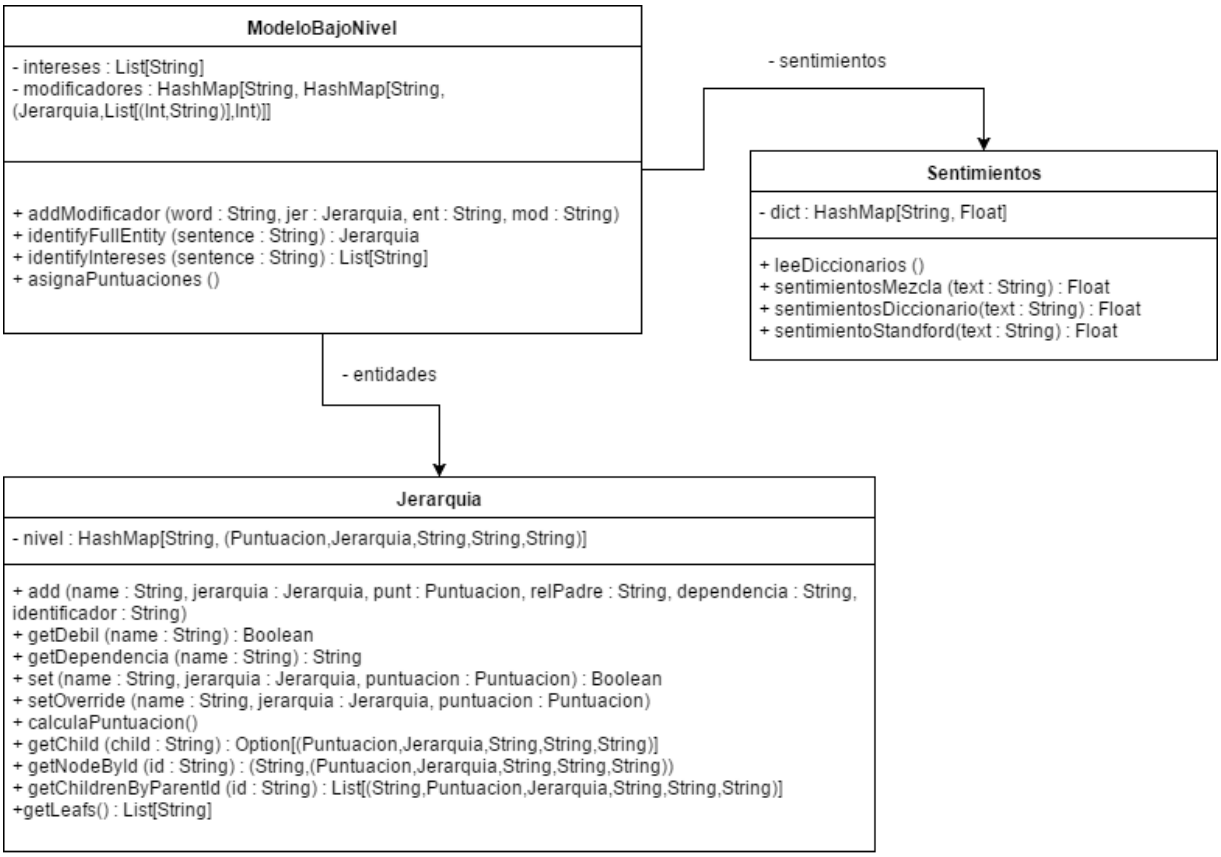

Figura 2.2.4: Clases Jerarquia, ModeloBajoNivel y Sentimientos

Para facilitar la escritura de los siguientes párrafos y evitar redundancia y repetir explicaciones, se va a introducir el concepto “subjerarquía”. Una “subjerarquía” es un sub-árbol del árbol usado para representar la jerarquía, pero que sigue cumpliendo las restricciones establecidas por la clase *Jerarquia*, es decir, admite nodos con nombre repetido y cada nodo puede tener un número de hijos distinto al de los demás.

Las palabras encontradas al realizar el análisis del texto se almacenan en una estructura de datos un tanto compleja. Esta estructura de datos consiste en una tabla *hash* con la siguiente información:

- Clave: elemento de la jerarquía modificado.

- Valor: otra tabla *hash*. usada para referenciar y encontrar en la jerarquía al elemento a la que la palabra encontrada modifica. Esta segunda tabla es esencial si el nombre del nodo se repite bastante en el árbol. Para encontrar la referencia se usa al padre del elemento modificado. Si no se puede encontrar un padre para el nodo en la frase, se usa la misma clave anterior repetida.
    - Clave: padre del elemento modificado
    - Valor: tupla con varios elementos:
        - Subjerarquía a la que pertenecen el elemento modificado y su padre. La identificación de la jerarquía se explica más adelante en esta sección.
        - Lista de pares de la forma: (palabra modificadora, número de veces que aparece)
        - Número de menciones que se ha hecho al elemento modificado de la jerarquía en la frase.

Si se dispone de la jerarquía de la Figura 2.2.5, y de la frase “I love Bill and Hillary Clinton”, la estructura anterior se muestra en la figura 2.2.6.

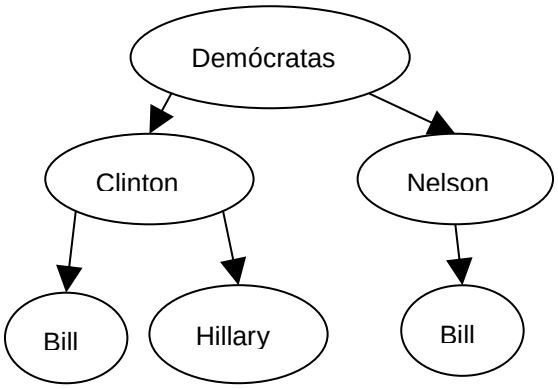

Figura 2.2.5: Jerarquía de ejemplo con repetición del partido Demócrata

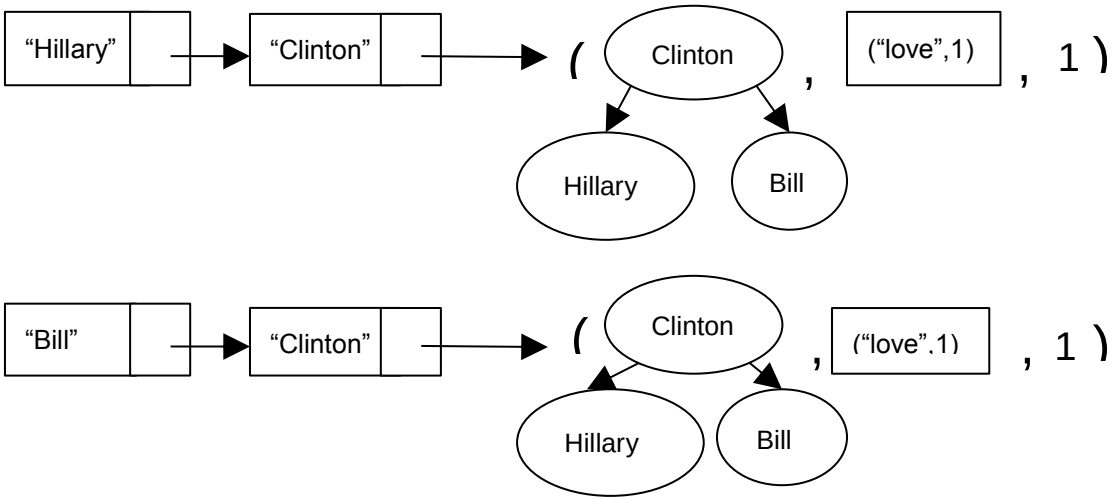

Figura 2.2.6: Estructura de tablas hash intermedia

Esta clase contiene métodos para identificar la posición que ocupan las entidades encontradas en el texto en la jerarquía, guardar sus modificadores en la tabla anterior, identificar sus sentimientos y finalmente modificar la jerarquía original. Ya que en la jerarquía pueden aparecer entidades con nombre repetido, lo primero que hay que hacer es identificar a cuál de todos los que aparecen en la jerarquía se hace referencia.

Al inicio del análisis de un texto, se buscan las palabras que aparecen tanto en este como en la jerarquía, y se busca en la jerarquía las posibles subjerarquías que contengan alguna de estas palabras. Una vez se han encontrado, se comprueba si son disjuntas, si lo son se modifican ambas, si no lo son, se escoge aquella de mayor tamaño, es decir, aquella en la que más palabras encajan; si todas tienen el mismo tamaño, se ha decidido escoger a la más pequeña en orden lexicográfico por ser el primer elemento encontrado en la tabla *hash*, aunque otras estrategias también son válidas. Ilustremos esto con la jerarquía demócrata (Figura 2.2.5) y las frases: “Adoro a Bill y Hillary Clinton”, “Adoro a Bill Clinton y a Bill Nelson” y “Adoro a Bill”.

En la frase “Adoro a Bill y Hillary Clinton” se encontrarían dos subjerarquías (Figura 2.2.7): la de tres nodos de la familia Clinton, y la subjerarquía de un solo nodo de Bill (de la familia Nelson). Aplicando lo anterior, como no son disjuntas, ya que la subjerarquía que tiene solo el nodo Bill es subconjunto de la primera, se elige la subjerarquía más grande, a la de la familia Clinton. En la segunda frase “Adoro a Bill Clinton y a Bill Nelson” se obtienen dos subjerarquías de dos nodos cada una: Clinton→Bill y Nelson→Bill; como ninguna subjerarquía es subconjunto de la otra, se escogen las dos. En la frase “Adoro a Bill” se obtienen dos subjerarquías de un solo nodo completamente iguales, las dos son solo “Bill”; en este caso se elegiría a Bill de la familia Clinton, por ser Clinton menor que Nelson alfabéticamente.

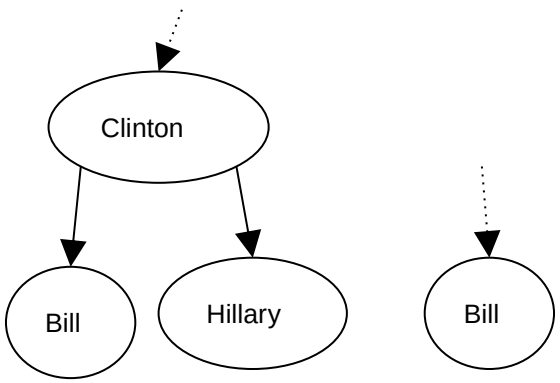

Figura 2.2.7: Subjerarquías encontradas al analizar “Adoro a Bill y Hillary Clinton”

El ejemplo de la frase “Adoro a Bill” sirve también para ilustrar un problema encontrado. Si tuviéramos la jerarquía anterior y buscáramos información de ella en

Twitter con el algoritmo anterior, cada vez que se hablara de un Bill aleatorio, asociaríamos lo que dicen de él a Bill Clinton. Para evitar esto se ha decidido dejar que el usuario defina una entidad como “débil” respecto a su padre, en estos casos solo se modificará el nodo si aparece en el texto analizado junto con su nodo padre. Es decir, que si en la jerarquía inicial definiéramos que Bill es débil respecto a Clinton, la frase “Adoro a Bill” no pasaría el filtro.

Una vez se ha identificado la subjerarquía, se añaden los modificadores que se han encontrado en la tabla explicada anteriormente. Se itera sobre los elementos de la tabla, consiguiendo el sentimiento de los modificadores mediante la clase *Sentimientos*. Una vez se han tratado todos, se calcula su media, se busca la subjerarquía en la jerarquía inicial, y se añaden los modificadores a los nodos que más abajo estén en la subjerarquía (es decir, en la jerarquía de ejemplo anterior, para la frase “Adoro a Bill y Hillary Clinton” se modifican sólo los nodos “Bill” y “Hillary” hijos de “Clinton”, pero no se le añaden modificadores a “Clinton”) y se actualiza la puntuación que tienen.

2.3.3 Análisis de sentimientos

Como se ha dicho antes, la clase encargada de realizar el análisis de sentimientos es la clase *Sentimientos*. Esta clase se encarga de ofrecer métodos que dada una cadena de texto calculan su sentimiento, esto se hace o bien a través de la parte de análisis de sentimientos de la biblioteca Stanford, o bien usando un diccionario que contiene un listado de palabras positivas o negativas, o bien una mezcla de ambas.

Tal y como se ha mencionado en el apartado 2.2.2, la biblioteca de Stanford tiene a veces un comportamiento no intuitivo, y como además es mejor usar diversas fuentes, se decidió añadir la posibilidad de usar un diccionario que contiene palabras positivas y negativas. En el código se inicializa este diccionario usando dos archivos de texto [26] creado por los investigadores Bing Liu, Minqing Hu y Junsheng Cheng, uno de ellos contiene palabras positivas y el otro negativas; aunque siempre sería posible inicializar el diccionario usando otras fuentes similares. El diccionario es una tabla hash que contiene como clave palabras y como valor un número entero que indica su sentimiento asociado, un 3 para las positivas y un 1 para las negativas; si se trata de buscar el sentimiento de una palabra que no se encuentra en el diccionario se le asignará un 2, una puntuación neutra.

La biblioteca de Stanford funciona de manera parecida, y al analizar el sentimiento de una frase devuelve un valor del 0 al 4, siendo 0 un sentimiento bastante negativo y un 4 bastante positivo. La clase *Sentimientos* ofrece métodos para usar solo una de las fuentes, pero también ofrece la posibilidad de usar las dos. Cuando se decide usar las dos fuentes se analiza la palabra individualmente con la biblioteca de Stanford y con los diccionarios y se analiza el valor devuelto por cada una de ellas.

Si el valor de las dos de ellas es neutro (ambos métodos devuelven el valor 2), entonces se asigna un sentimiento neutro de valor 2; si el de un método devuelve que es neutro y el otro no, se da un 80% de peso al valor no neutro y 20% de peso al neutro; en otro caso se calcula una media de los dos valores devueltos. La decisión de asignar mayor peso al valor no neutro corresponde a que es conveniente reflejar que alguna fuente entiende que la palabra no es neutra, si se hiciera una media se conseguiría que palabras vistas como poco positivas por una fuente y como neutras por la otra perdieran casi totalmente el sentimiento que una de las fuentes le ha asignado.

Dado que el análisis se hace sobre palabras individuales, los casos en los que hay negaciones y cuantificadores deben tratarse por separado. Antes de enviar la palabra a analizar a esta clase, se ha averiguado ya si la palabra ha sido negada o cuantificada (más información en el apartado 2.3.4). Si la palabra ha sido negada se habrá añadido a la cadena el prefijo “not” (separado de la palabra por un espacio), y si ha sido cuantificada se le han añadido todos los cuantificadores encontrados. De este modo el análisis de sentimiento en esta clase es sencillo, en el caso de la negación basta con invertir la puntuación de todo aquello encontrado después de la palabra “not”, y en el caso de los cuantificadores solo hay que analizarlos y según sean positivos o negativos, aumentar o disminuir el sentimiento de la palabra a la que modifican.

2.3.4 Análisis de la frase y extracción de modificadores

Ya se ha explicado la estructura de la jerarquía, la estructura para guardar las entidades, los intereses y los modificadores durante el análisis de lenguaje, cómo se identifican las entidades presentes de una frase y cómo se analizan los sentimientos de los modificadores. En este apartado se explicará el paso intermedio, es decir, cómo se analiza la frase para encontrar las palabras modificadoras.

La clase encargada de este análisis será la clase *TrataModelo*. Esta clase cuenta con cinco atributos para realizar su tarea:

- Una referencia al objeto *ModeloBajoNivel* con la jerarquía inicial, la lista de intereses y la tabla en la que se escriben los modificadores encontrados en la frase
- Una referencia a un objeto *Jerarquia* que cambiará por cada frase que se analice y que contiene la subjerarquía que contenga a las entidades que aparecen en la frase
- Una lista con los intereses que aparecen en la frase
- Una lista negra de palabras que interesa que no se analicen,
- Una referencia a un objeto de clase *Sentimientos* para tratar los casos especiales en los que se encuentran negaciones y cuantificadores.

En esta clase el tratamiento del lenguaje se realiza mediante la biblioteca de Stanford NLPCore, principalmente usando el análisis *PartOfSpeech* y el análisis morfológico. El dominio de este proyecto se restringe al análisis de texto en inglés, aunque la biblioteca disponga de herramientas para otros idiomas. El análisis *PartOfSpeech* de una frase devuelve un grafo, en el que los nodos son las palabras de la frase, y las relaciones entre los nodos son las distintas relaciones lingüísticas entre las palabras, es decir, cuáles palabras forman un nombre compuesto entre ellas, qué adjetivos modifican a qué sustantivos, a qué palabras se refieren los determinantes, etc.

El proceso de análisis se realiza de la siguiente manera:

1. Búsqueda de las entidades de la frase y obtención de la subjerarquía mediante una llamada al atributo de la clase *ModeloBajoNivel*.
2. Búsqueda en la frase de las palabras que aparecen en la lista de intereses del atributo de la clase *ModeloBajoNivel*.
3. Generación del grafo con relaciones entre palabras mediante la biblioteca de Stanford NLP.
4. Por cada entidad de la subjerarquía, recorrido del grafo siguiendo determinadas relaciones, si las palabras pasan el filtro (explicado más adelante) se añaden a la tabla de *ModeloBajoNivel*.
5. Por cada interés identificado, buscar con qué entidad está relacionado en la frase, y tras esto realizar el mismo análisis que en el punto anterior.

El recorrido del grafo se realiza manteniendo una lista de nodos visitados, de forma similar a los algoritmos de recorrido de grafos usuales, con la simplificación de que las aristas no tienen coste, y algunas aristas no hace falta recorrerlas. No basta solo con recorrer las relaciones que proceden de las entidades de la subjerarquía, sino que conviene recorrer todos los nodos que contengan un camino a alguna entidad de la subjerarquía, ya que hay relaciones indirectas que son importantes. Un ejemplo para clarificar esta situación es el provocado por la frase: “Los Galaxy S7 son extremadamente peligrosos”, en esta frase el análisis proporcionaría una relación entre “S7” y “peligrosos” de nombre “sujeto nominal” (relaciona el sujeto con el verbo de la frase a no ser que el verbo sea copulativo y entonces lo une con el complemento del verbo, como en este caso), y otra relación entre “peligrosos” y “extremadamente”; en este caso concreto estaríamos interesado también en conseguir la palabra “extremadamente”, y para eso hace falta recorrer las relaciones de “peligrosos”.

La biblioteca de Stanford reconoce muchas relaciones, pero en este proyecto únicamente se utilizan un subconjunto de ellas. Los nombres de las relaciones se escribirán en inglés, ya que el idioma del texto debe ser ese, aunque en la explicación de la relación se tratará de traducir a su correspondiente término en castellano.

- **Nominal subject**: relación de sujeto nominal, entre el sujeto y el verbo. Si el verbo es copulativo, une al sujeto con el complemento del verbo.
  Ej: “Trum defeated Clinton” → *nsubj(defeated, Trump)*
- **Clausal complement**: en una frase compuesta subordinada, relación entre el verbo de la frase principal con el verbo de la subordinada cuando el verbo de la subordinada actúa como objeto o adjetivo de la oración principal.
  Ej: “I am certain that he did it” → *ccomp(certain, did)*
- **Direct object**: relación entre el verbo y su complemento directo.
  Ej: “Santa brought presents” → *dobj(brought, presents)*
- **Open clausual component**: relación presente en las oraciones subordinadas sustantivas de complemento directo, que une el verbo de la principal oración con el verbo de la subordinada, o con su sujeto si el verbo no está presente.
  Ej: “He says you are ready to leave” → *xcomp(ready, leave)*
- **Appositional modifier**: en oraciones compuestas (generalmente coordinadas o yuxtapuestas), relación entre un pronombre de una oración y la palabra a la que hace referencia en la otra.
  Ej: “Trump, the president, is blonde” → *appos(Trump, president)*
- **Clausal modifier of noun**: en oraciones compuestas, relaciones adjetivas entre las frases.
  Ej: “I love the dog that barks to the Moon” → *acl:relcl(dog, barks)*
- **Dependency**: relación asignada entre dos palabras cuando el sistema no puede establecer qué tipo de relación lingüística hay entre ellas.
  Ej: “NEWS: Trump visits Israel http://www.news.fake” → *dep(visits, http...)*
- **Compound**: relación entre varias palabras que forman juntas un nombre compuesto.
  Ej: “Hillary Clinton lost.” → *dep(Hillary, Clinton)*
- **Adjectival modifier**: relación entre un adjetivo y la palabra a la que modifica.
  Ej: “The phone has a big battery.” → *amod(big, battery)*
- **Adverb modifier**: relación entre un adverbio y la palabra a la que modifica.
  Ej: “Windows crashes too much.” → *advmod(too, much)*
- **Nominal modifier**: relaciones nominales, de varios tipos:
  - Temporal modifier: relación entre un verbo y el complemento de tiempo.
    Ej: “I eat every day.” → *nmod:tmod(eat, day)*

    - Possesive: relación de posesión, entre el sujeto y aquello que posee (posesión en el sentido: “Mi teléfono”$= Mi \leftrightarrow teléfono$ o “La batería del móvil”$= móvil \leftrightarrow batería$).

      Ej: “I love my phone” → *nmod:poss(eat, day)*

- **Negation**: cuando se encuentra un adverbio de negación, se forma una relación entre ese adverbio y la palabra que niega.

  Ej: “I don't like centipedes.” → *not(n’t, like)*

Es conveniente mencionar que el manual de dependencias la biblioteca de Stanford NLP no contiene descripción de todos los tipos de relaciones que existen. El proceso de averiguar qué tipos de relaciones usar y cuáles no usar fue en gran parte experimental, y consistió en ejecutar el análisis con distintas frases y comprobar los distintos tipos de relaciones que se conseguían.

Las relaciones de negación y de modificadores adverbiales se tratan de distinta manera a las otras para que la clase *Sentimientos* puede hacer un análisis de sentimientos de forma cómoda. En el caso de la negación, se añade la palabra “not” y un espacio delante de la palabra a la que modifica; en el caso de los adverbios, se colocan delante de la palabra a la que modifican igual que en el caso de la negación; si se dan los dos casos simultáneamente la primera palabra será “not”, seguida de los adverbios, y finalmente la palabra a la que modifican.

Una vez se averiguan las palabras relacionadas con las entidades se efectúa un filtro sobre ellas, para averiguar si merece la pena añadirlas a la tabla de modificadores o no. Este filtro descarta los pronombres personales, los pronombres posesivos, los nombres propios, los determinantes, y teniendo en cuenta que el análisis se realiza con Twitter, también descarta los hashtags, las referencias a usuarios de Twitter y los links a páginas web (estos últimos serán sustituidos por el token <link>).

Un flujo de alto nivel del proceso que se sigue para analizar un texto y modificar la jerarquía original se puede encontrar en la figura 2.2.8.

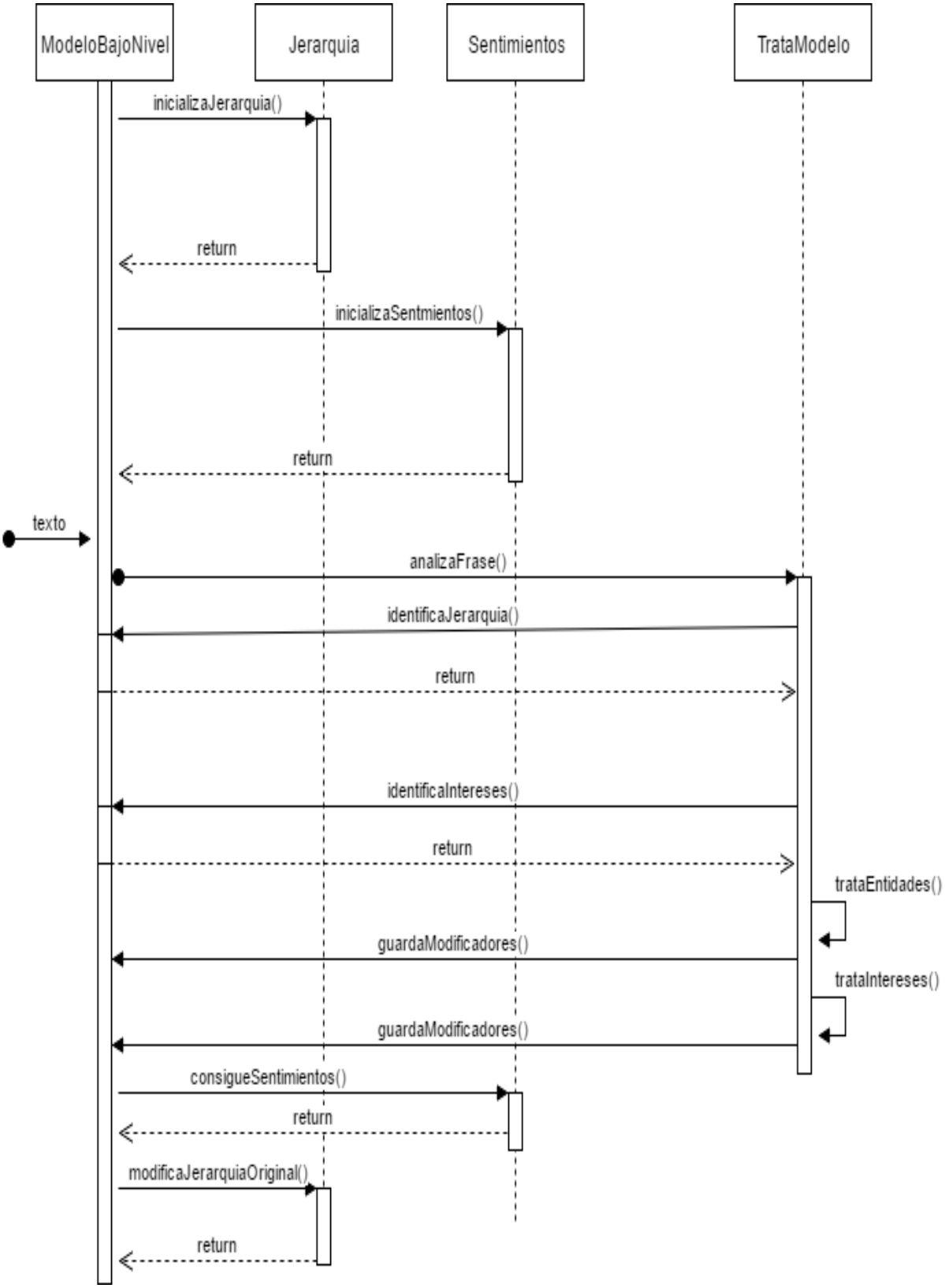

Figura 2.2.8: Flujo de alto nivel de análisis de texto

2.3.5 Definición de la jerarquía inicial del usuario

El programa necesita como entrada la definición de una jerarquía establecida por el usuario. La definición de esta jerarquía se hará usando dos archivos .xml, aprovechando la estructura arbórea que estos permiten.

En un fichero XML se define la jerarquía, estableciendo los nombres de los nodos, los nombres de las relaciones entre los nodos y si un nodo es débil respecto a su padre. En el otro xml se define el tipo de los nodos, si son entidades o intereses; esta distinción es importante, ya que se entiende que aunque los intereses se definan como hijos de una entidad, si al analizar un texto se encuentran modificando a otra entidad del árbol también se añadirá la información encontrada. Esta separación en dos xml diferentes permite, además, que en un futuro se puedan definir más tipos de nodos distintos, no sólo entidades e intereses, y que no haya que alterar la jerarquía original.

El xml en el que se define la jerarquía debe tener el formato siguiente:

```
<jerarquia name=[nombre de la jerarquía]>
    < [nombre de relación 1] name= [nombre del nodo] >
        < [nombre de relación 2] name= [nombre del nodo 2]  weak = [bool]>
            ...
        </ [nombre de relación 2] >
        ...
    </ [nombre de relación 1] >
    ...
</jerarquia>
```

En el segundo xml, por cada tipo de relación definida en el primer xml, hay que establecer en esta si se trata de una relación hacia una entidad o hacia un interés, aquí llamado elemento. El formato que ha de seguir es el siguiente:

```
<plugin name=[nombre de la jerarquía]>
    < relation name= [nombre de relación 1] > [entidad | elemento] </relation >
    < relation name= [nombre de relación 2] > [entidad | elemento] </relation >
    ...
</jerarquia>
```

Al inicio de la aplicación, se realizará una lectura de los dos archivos xml y se guarda la jerarquía generada y los intereses. Los archivos xml del primer tipo deben incluirse en la carpeta del proyecto de nombre “jerarquias” dentro de la carpeta “main/resources”, los archivos xml del segundo tipo deben estar en la carpeta de nombre “plugins” dentro de la carpeta “main/resources”.

Como ejemplo, a continuación se muestran los XML de la jerarquía demócrata representada en la figura 2.2.5 con cuatro intereses añadidos: “amante” y “pelo” para Bill Clinton, “marido” para Hillary, y “puesto” para Bill Nelson.

```
<jerarquia name="Politicos">
	<partido name="Democrata">
		<familia name="Clinton">
			<persona name="Bill">
				<tiene name="pelo"/>
				<tiene name="amante"/>
			</persona>
			<persona name="Hillary">
				<tiene name="marido"/>
			</persona>
		</familia>
		<familia name="Nelson">
			<persona name="Bill">
				<tiene name="puesto"/>
			</persona>
		</familia>
	</partido>
</jerarquia>


<plugin name="Politicos">
	<relation name="partido">entidad</relation>
	<relation name="familia">entidad</relation>
	<relation name="persona">entidad</relation>
	<relation name="tiene">elemento</relation>
</plugin>
```

2.3.6 Base de datos

Una vez se ha modificado la jerarquía original, hay que guardar los datos de forma persistente. El guardado de los datos cuenta con una serie de restricciones: debe ser rápido, flexible, debe permitir guardar grandes cantidades de datos, debe poder permitir transacciones concurrentes y finalmente es aconsejable que sea escalable de forma horizontal.

Estas restricciones hace que el modelo más lógico de datos para usar sea un modelo de base de datos no relacional. Existen multitud de modelos de bases de datos no relacionales, y muchas de ellas cuentan con soporte nativo de Spark, como Hadoop; sin embargo, por la similitud que tiene el modelo de grafos con los datos que se están tratando en este proyecto (árboles), al final se tomó la decisión de usar Neo4J.

Dado que el modelo de datos usado por el programa es tan similar al grafo, el diseño de la base datos fue inmediato. Los nodos de la base de datos cuentan con etiquetas para determinar qué tipo de nodos es, y atributos para guardar su información. Según su etiqueta, un nodo tiene distintos atributos:

- Entidad: las entidades de la jerarquía. Tienen como atributos: nombre, puntuación de los hijos, puntuación propia, número de veces que se ha mencionado y un identificador único.
- Elemento: lo que antes se ha llamado “interés”. Tienen los mismos atributos que un nodo Entidad.
- Modificador: tienen como atributos: puntuación de la palabra y nombre. En este caso no nos interesa un identificador único ya que no querremos que en la base de datos haya más de dos nodos modificadores con el mismo nombre.

Las relaciones entre los nodos mantienen un tipo y contienen atributos. En la base de datos existe un número de tipos de relaciones distintas entre los nodos Entidad igual al número de relaciones definida en la jerarquía original. Las relaciones entre nodos Entidad y nodos Elemento se llama “elemento”, y las relaciones entre nodos Entidad o Elemento y nodos Modificador se llama “modificador”. Tan solo las relaciones “modificador” tienen atributos, y este es el número de veces que ha aparecido el nodo Modificador en una frase asociado a la Entidad o al Elemento con el que está relacionado.

En la base de datos también se deben respetar ciertas restricciones respecto a los nodos y las relaciones:

- De un nodo Entidad pueden salir relaciones hacia nodos Entidad, Elemento y Modificador.
- De un nodo Elemento solo pueden salir relaciones hacia nodos Modificador.
- Si hay dos nodos Entidad o Elemento con el mismo nombre, se crearán dos nodos distintos con distinto identificador.
- No se crearán distintos nodos Modificador con el mismo nombre para así evitar que el número de nodos crezca indefinidamente. Si una palabra se encuentra modificando a más de un nodo Entidad o Elemento, se creará solo un nodo con el nombre de la palabra al que le lleguen relaciones desde los nodos Entidad o Elemento a los que modifica.
- Un nodo Modificador solo existirá en la base de datos si existe como mínimo una relación entre él y otro nodo Entidad o Elemento.

Dado que estas restricciones pueden ser un poco confusas, a continuación se presenta un diagrama Entidad-Relación, que aunque no sirva para representar el modelo en una base de datos de grafos, sí que sirve para aclarar las restricciones anteriores:

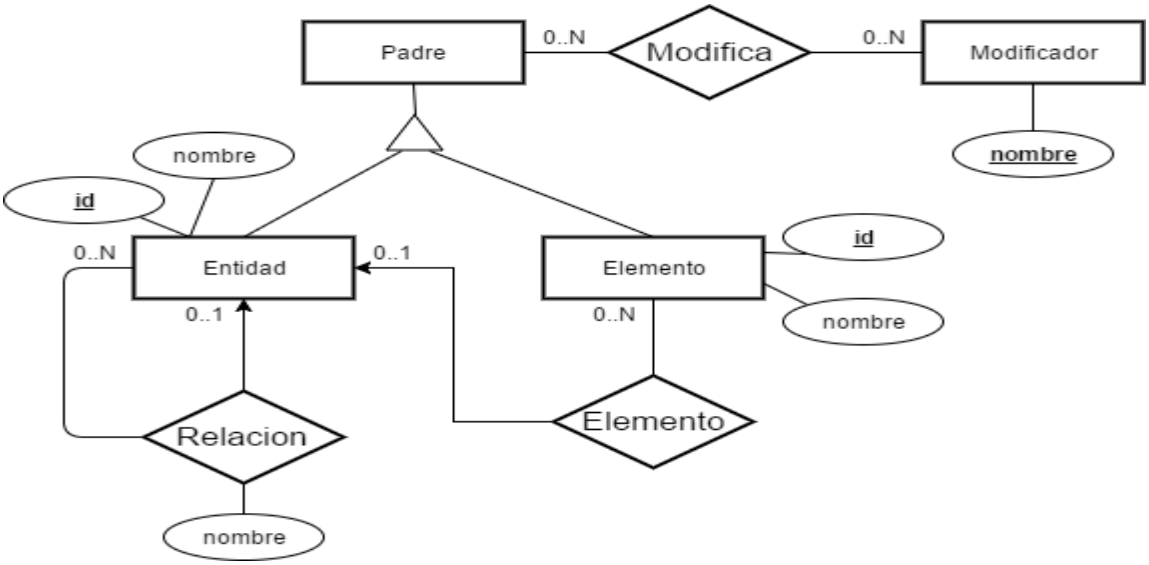

Figura 2.2.9: Modelo Entidad-Relación de los datos

Nótese que la entidad Padre presente en el modelo no existe realmente como tipo de nodo en la base de datos, pero sirve para representar la restricción de que un nodo Modificador necesita que exista una relación entre él y un nodo Entidad o Elemento. También se ha evitado incluir en la figura los atributos que no suponen restricción alguna, como los distintos tipos de puntuación.

Además de la estructura básica de la base de datos, -, sería útil guardar la información por fechas para que se pudiera obtener una evolución de los sentimientos por fecha, ver el número de menciones que ha tenido una entidad durante cierto tiempo, etc. Para esto hay que cambiar varios atributos y cambiar cierto comportamiento de las relaciones.

Lo primero que hay que cambiar si se quiere mantener un control para saber cuántas menciones ha tenido una entidad o un elemento para una fecha determinada es el atributo “menciones” de estos dos nodos. Teniendo en cuenta que Neo4J no acepta guardar arrays de pares, y que los elementos de un array tienen que ser todos del mismo tipo, a partir de ahora el atributo “menciones” será un array en el que las posiciones pares (posición 0 incluida) guardan fechas y las impares el número de menciones, de tal manera que una posición *n* impar contiene el número de veces que ha sido mencionada la entidad para la fecha que ocupa la posición *n-1* del array.

El segundo cambio necesario afecta a las relaciones entre los nodos Entidad y Elemento y los nodos Modificador. A partir de ahora la relación contendrá la fecha en la que el modificador apareció relacionado con la entidad o elemento, y el número de veces que se vió en esa fecha. Esto quiere decir que si un nodo Modificador aparece relacionado con el mismo nodo Entidad o Elemento en tuits escritos en distintas fechas, existirá más de una relación entre el Modificador y la Entidad o Elemento.

Con los dos cambios anteriores ya tenemos todo lo mínimo necesario para poder obtener información de los sentimientos de una entidad según la fecha, ya que disponemos de las menciones que han tenido en cada fecha y de qué modificadores se han encontrado asociados en cada fecha y cuántas veces. El cálculo puede ser costoso si se pregunta por alguno de los nodos de los niveles superiores de la jerarquía y si hay una gran cantidad de relaciones con nodos Modificador, pero tal y como se cuenta en el siguiente apartado, es necesario un control sobre el número de nodos en la base de datos y sobre el número de relaciones.

Con el fin de explicar mejor la organización de la base de datos, en la figura 2.2.10 se muestra el contenido de la base de datos usando el visor web de Neo4J al usar una jerarquía pequeña de la familia de políticos estadounidenses Clinton y las frases “Clinton lost the elections because of her incompetence”, “I hate Hillary Clinton but she is a good woman” y “Bill Clinton's hair has become really white”, escritas todas en la misma fecha.

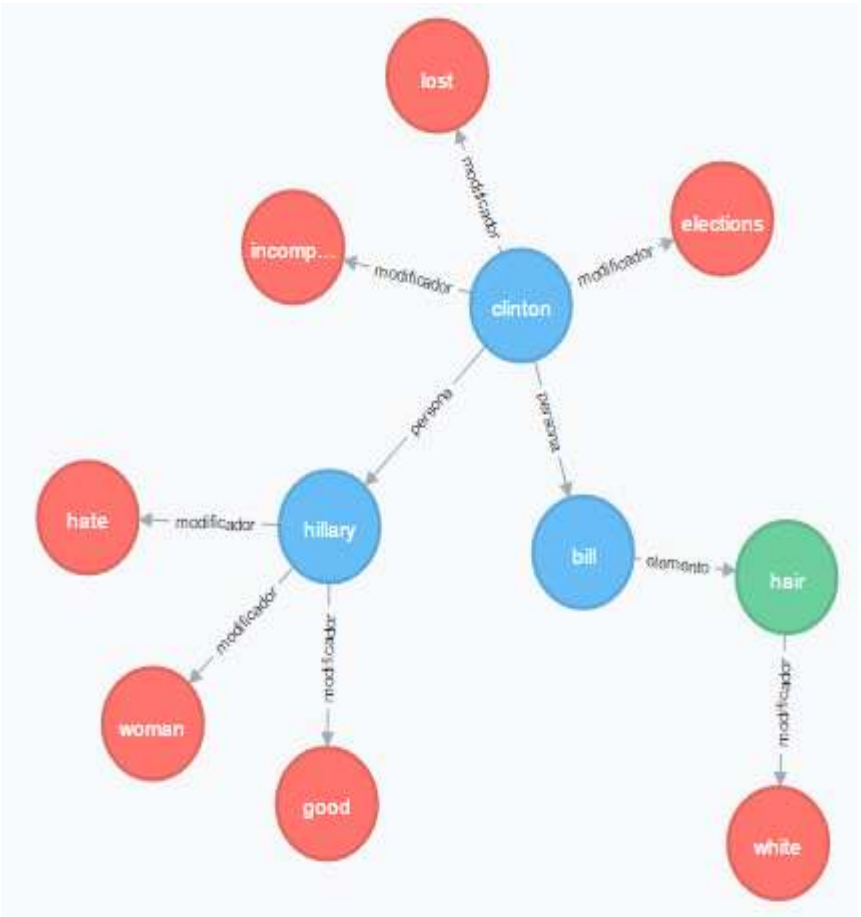

Figura 2.2.10: Contenido de la Base de Datos usando a la familia Clinton

Del guardado en la base de datos se encarga la clase estática (un objeto Scala) *ManejadorBaseDatos*. Esta clase mantiene una conexión con la dirección en la que

se encuentra la base de datos, y contiene un método que recibe un objeto de la clase *Jerarquia* y la guarda en la base de datos, manteniendo los datos anteriores y sobreescribiendo lo necesario. Las puntuaciones de las entidades y su lista de menciones necesitan ser leídas para poder actualizarse, de igual manera, hace falta consultar si un nodo Modificador con un nombre determinado existe para no crear otro igual. Hay que tener en cuenta además que el análisis de los tuits se realizan de manera concurrente debido al modelo de cómputo de Spark, y hay que garantizar que las restricciones mencionadas anteriormente se sigan cumpliendo. Afortunadamente, Neo4J ofrece operaciones que garantizan bloqueos sobre escritura en ciertos nodos y nodos relaciones, tales como *MERGE, CREATE UNIQUE* y *SET*.

2.4 Limpieza y compactación de la base de datos

2.4.1 Problemática

En Twitter se generan una media de 500 millones tuits diarios [27]. Aunque la aplicación sólo recoge información de tuits en inglés y que contengan alguna palabra presente en la jerarquía, la cifra anterior da una idea de la gran cantidad de datos que se generan en Twitter y de que es necesario establecer un cierto control. En el apartado 2.2.3 de la memoria ya se ha hablado de los límites de Neo4J, y a continuación se ilustra un ejemplo de por qué es necesario eliminar algunos nodos y compactar cierta información.

Supongamos que disponemos de una jerarquía inicial de *n* entidades y elementos, y que se han encontrado *m* palabras; en el caso peor, es posible que todos los nodos de la jerarquía estén relacionados con las *m* palabras encontradas, esto supondría tener *n*m* relaciones diferentes en la jerarquía. Hay que tener en cuenta también que se generan distintas relaciones entre un nodo Entidad y un Modificador según la fecha, si se realiza el corte de las fechas según la hora del día (es decir, se supone que un tuit analizado a las 10:12 de un determinado día, y otro a las 10:54 del mismo día se han realizado en la misma fecha), se puede tener que para un número de horas *h* se tengan *h*m*n* relaciones distintas.

Ahora supongamos que se puede aproximar el máximo valor *m* según el número de palabras existentes en el idioma inglés, esto es del orden de $22 * 10^4$palabras [28]. En un día se generarían $24 * 22 * 10^4 * n$.En un año, para una jerarquía con 100 elementos, se llegaría al orden de las $30* 10^{18}$ de relaciones, que teniendo en cuenta la velocidad a la que se generan los datos, es una cifra relativamente cercana a la máxima recomendable.

Dado que restringir el tamaño de la jerarquía que quiere introducir el usuario no es algo recomendable, hay que centrarse en eliminar relaciones y nodos innecesarios, y en compactar relaciones cuando sea posible.

2.4.2 Limpieza

Lo primero en lo que centrarse es en eliminar las relaciones y los nodos no relevantes. Para esto se ofrecen varios métodos en la clase estática *ProcesosPeriódicos* que permiten arrancar un proceso que se ejecute durante con una frecuencia introducida como parámetro. Como no se pueden eliminar nodos Entidad o Elemento o sus relaciones, hay que centrarse en eliminar las relaciones que llegan a nodos Modificador.

Como el número de nodos que puede soportar la base de datos depende en gran medida de las capacidades del sistema en el que esté instalado, se ha decidido dar al usuario varias posibilidades para limpiar modificadores innecesarios:

- Eliminar las relaciones entre nodos Modificador y nodos Entidad y Elemento tales que el número de repeticiones que ha aparecido la palabra del Modificador asociada a la Entidad o el Elemento es menor que un parámetro. Tras esto se eliminan los nodos Modificador que no tengan ninguna relación.
- Por cada nodo Entidad y Elemento, se calcula la media de repeticiones de los Modificadores asociados a ellos, y se eliminan las relaciones con Modificadores que sean menor de la media. Igual que en el método anterior se eliminan los nodos Modificadores sin relaciones.
- Eliminar un porcentaje de las relaciones de los nodos Entidad y Elemento. Se eliminan de menor número de repeticiones a mayor número de repeticiones. Después se eliminan los nodos Modificador inconexos.

2.4.3 Compactación

Otra manera de reducir el número de relaciones es juntando varias relaciones en una. Las relaciones propensas a esto son las relaciones entre una misma Entidad o Elemento y un mismo Modificador, que sólo sean diferentes por la fecha.

La idea es escoger todas las relaciones entre un nodo Entidad y un Modificador cuya fecha difiera en la hora pero comparta el día, mes y año, eliminarlas, y crear una nueva con su atributo de repeticiones igual a la suma de los atributos repeticiones de las relaciones anteriores y con una fecha igual a la anterior pero con la hora fijada a las 00:00. Si se aplicara esto a las relaciones generadas en un año se podría pasar de casi 9000 relaciones a unas 365.

Se ha implementado un método que compacta las relaciones en días para fechas menores que un parámetro dado. Esto mismo se podría realizar por grupos de siete días, por meses, por años, etc. y reducir así todavía más las relaciones que existen

en la jerarquía. Esta decisión depende del comportamiento de la base de datos observado por el administrador y el usuario. También existen otras posibilidades no implementadas, como replicar parte de los contenidos de la base de datos en otro servidor e ir aplicando ya en la nueva los métodos aquí comentados, consiguiendo así que en la antigua se siga manteniendo información que no se quisiera perder.

2.5 Aplicación cliente

En este apartado se detallan las decisiones de diseño tomadas respecto a la aplicación gráfica que se ejecuta en el cliente. Esta aplicación tiene como objetivo conectarse a un servidor con la base de datos, mostrar al usuario un resumen de la situación de la jerarquía, permitirle ver una evolución de la situación de ésta, y darle la posibilidad de realizar ciertas búsquedas en la base de datos.

Teniendo en cuenta los objetivos, se ha decidido que la aplicación tenga tres partes partes principales:

- Visualizador del grafo, mostrando los nodos más relevantes.
- Visualizador de gráficas que muestra la evolución de los sentimientos en función del tiempo, y resúmenes de los sentimientos de los elementos de la jerarquía.
- Buscador complejo de entidades, elementos o modificadores según sus sus atributos.

A continuación se explicarán primero los recursos de bajo nivel, tales como el acceso a la base de datos y la escritura y lectura de archivos; y después se procederá a explicar las distintas partes de la interfaz gráfica y cómo usan estos recursos.

2.5.1 Acceso a la base de datos

Acceder a la base de datos y realizar consultas sobre ella es una labor que deben realizar todos los elementos de la interfaz. Para el acceso a la base de datos se seguirá usando la biblioteca Anormcypher, pero a diferencia de la aplicación Spark, que mayoritariamente realizaba escrituras, esta aplicación realizará lecturas, lo cual implica tener que tratar con los tipos devueltos por la biblioteca y tener que traducirlos.

Para separar las consultas a la base de datos de la traducción de los datos entre los tipos devueltos por Anormcypher y los que usa la aplicación, se han creado dos clases estáticas distintas: *Consultas* y *TraduccionDatos*. De esta manera, cuando una parte de la aplicación necesite conseguir datos de la base de datos, llamará a una función de *TraduccionDatos*, y esta clase se encargará de llamar cuantas veces sea necesario a las funciones que contienen las consultas de la clase *Consultas*, y de devolver los datos a la clase que lo ha llamado con los tipos ya traducidos.

La clase estática *Consultas* contiene una conexión al servidor de la base de datos, que puede cambiar durante el transcurso de la aplicación, y que soporta conexiones con usuario y contraseña. Además de los procedimientos para configurar la conexión, todas las demás funciones contienen sentencias del lenguaje de consultas Cypher que devuelven un flujo con los resultados de la consulta.

Existe también otra clase llamada *RelacionesYAtributos* que se encarga de leer la jerarquía de la base de datos y de guardar los atributos que tienen los nodos según el tipo de relación que les llega, ya que por ejemplo, como los nodos Modificador no tienen hijos tampoco tienen un atributo puntuación propia y otro para la puntuación de sus hijos. Esta clase no es necesaria, porque los atributos solo cambian cuando se trata de nodos Modificador, y todas las Entidades y Elementos tienen los mismos atributos, pero es posible que en un futuro se cambie la aplicación para que las Entidades tengan distintos atributos entre sí. También es innecesaria ya que se inicializa sin acceder a la metainformación de la base de datos, sino estableciendo directamente en el código los atributos que corresponden a cada relación; pero así esta clase sirve como cimientos por si se quiere realizar alguno de los cambios anteriores, pues solo habría que cambiar su inicialización.

2.5.2 Dibujo del grafo

Un objetivo importante de la interfaz gráfica es que el usuario pueda visualizar el grafo. Tras investigar varias bibliotecas que pudieran cumplir esta función, al final se decidió usar Alchemy.js, con las restricciones que eso supone, tal y como se menciona en el apartado 2.2.6.

En ese mismo apartado se explica que Alchemy.js es una aplicación web y necesita HTML y JavaScript para funcionar, y los datos del grafo en formato JSON. El proceso que se realiza para poder mostrar los datos es el siguiente:

- Lectura de los datos de la base de datos.
- Traducción de los datos a formato JSON.
- Escritura de la cadena JSON en un archivo de texto.
- Escritura de la configuración JavaScript y los contenidos del archivo JSON en un archivo HTML.
- Cargado del archivo HTML con un componente JavaFX *WebView*.

La lectura inicial de los datos se realiza mediante la clase *Consulta*. Teniendo en cuenta que en la base de datos pueden existir miles de nodos, no es conveniente dibujar el grafo completo; para esto se realiza una consulta que consigue los diez modificadores más relevantes de cada entidad o elemento, entendiendo por relevantes aquellos que se repitan más veces y tengan puntuaciones más extremas, de tal manera que para dos modificadores con la misma puntuación, tiene el triple de

prioridad un nodo con puntuación buena o mala, o el cuádruple si es muy buena o muy mala.

Una vez se han leído los datos, hay que escribirlos en un archivo JSON, de esto se encarga la clase *EscribeJSON*. Alchemy.js espera recibir en el archivo una definición para las nodos, en el que debe ser obligatorio incluir un atributo identificador, y otra definición para las aristas, en la que cada arista está definida por dos atributos obligatorios que deben indicar los identificadores de los nodos que une la arista. En este caso no hace falta traducir los datos, ya que se va a aprovechar la metainformación de los resultados de la consulta para escribir el JSON. Lo que se ha hecho es devolver en la consulta los resultados con el alias que necesita Alchemy, por lo que al escribirlos en el archivo solo hay que incluir el nombre del atributo seguido de su valor. Es un proceso sencillo, ya que no necesitamos operar con los resultados y solo los escribimos en forma de cadena de texto, sin importar el tipo de éstos, si esto no fuera así operar directamente con la metainformación de la consulta resultaría engorroso.

Otra clase, llamada *GraphFileManager*, se encarga de guardar el código de configuración JavaScript que necesita Alchemy.js en forma de cadena de texto, y de tener los métodos necesarios para escribir en un archivo HTML este código JavaScript y los datos JSON. En la configuración JavaScript figuran funciones para establecer el color de los nodos según su sentimiento (muy verde, verde, blanco, rojo o muy rojo, según la puntuación es buena o mala), el tamaño de los nodos (mayor cuantas más menciones haya tenido la entidad en Twitter), y el nombre que se muestra cuando el ratón pasa por encima de los nodos (el nombre del nodo: en mayúsculas si es una entidad, con la primera letra en mayúscula y las demás en minúscula si es un elemento, y todo en minúsculas si es un modificador). Esta clase contiene además tres atributos, que son las rutas a los archivos JSON, HTML que se crea y el HTML que usa de plantilla para crear el otro.

Una vez se ha escrito en el archivo HTML, se encarga de mostrar su contenido un componente *WebView* de JavaFX. Este componente se encapsula en una clase *Browser* que contiene dos atributos: el componente *WebView* y el motor del *WebView*. Esta clase contiene métodos para crear la vista web, actualizarla y establecer el tamaño de esta. La clase está definida junto a otra, llamada *WebViewSample*, que se usó para probar el funcionamiento.

2.5.3 Visualización del grafo

La pantalla principal una vez se ha iniciado la aplicación y se ha establecido la conexión con la base de datos es la pantalla de visualización del grafo. Para facilitar la explicación, se comenzará mostrando una imagen de la pantalla y luego se procederá a explicar el porqué de su diseño y la programación que va detrás.

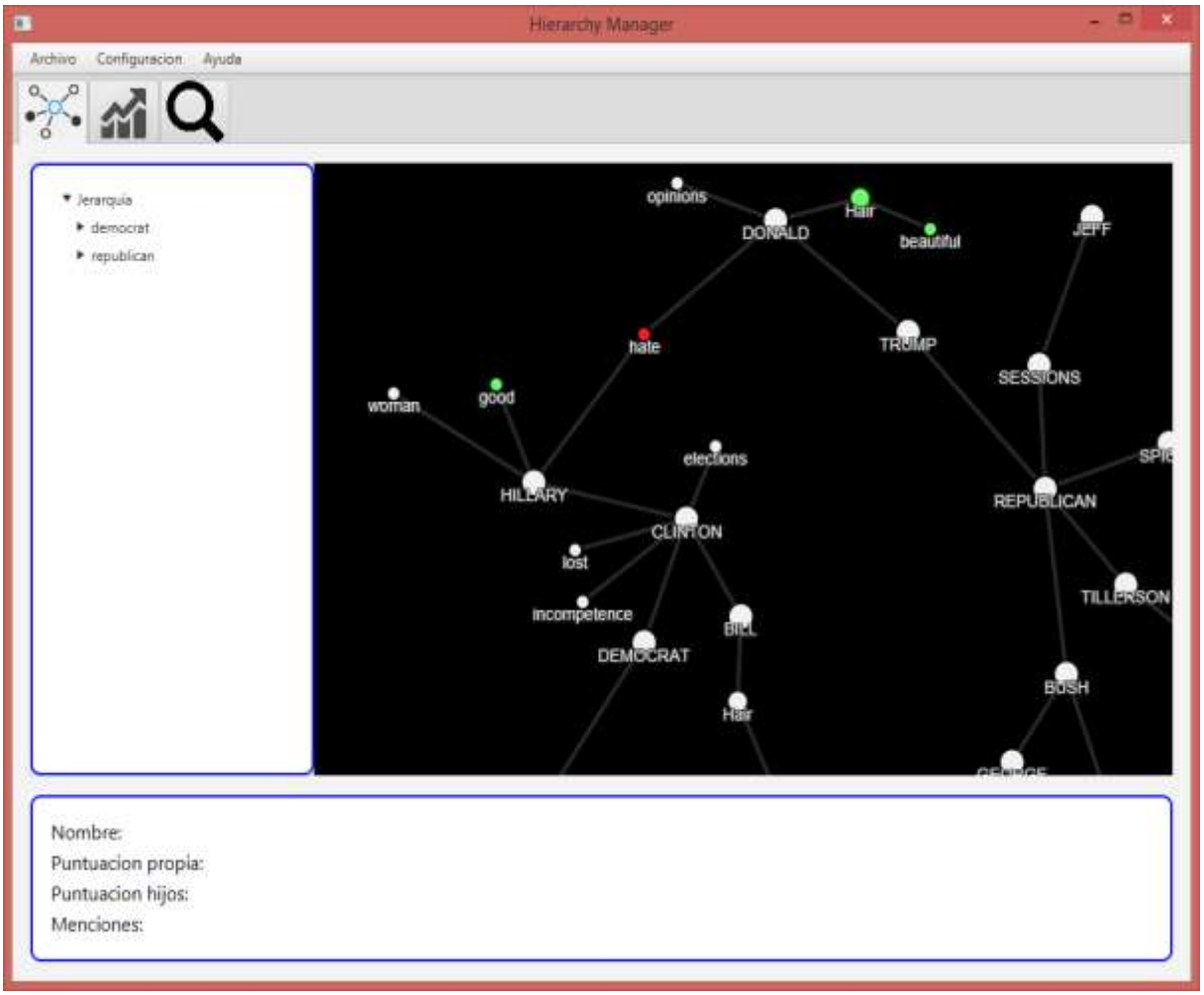

Figura 2.9: Grafo sencillo con el visualizador del grafo

Se realizó una investigación para comprobar el estado del arte respecto a interfaces gráficas que tuvieran que trabajar con consultas sobre grafos, y aunque no se encontró la información exacta que se buscaba, sí que se encontraron ideas de cómo organizar una interfaz para visualizar grafos.

En lo encontrado [29] se recomienda situar el grafo en la parte central de la pantalla, con los demás elementos de la interfaz rodeando a éste, también incluir en la parte izquierda los controles que determinen el grafo que se está dibujando. Tal y como se ve en la figura 2.9 se ha seguido este esquema, y a la izquierda se dispone de un árbol desplegable de la jerarquía, en el que se puede hacer doble clic sobre los nodos de ésta y se dibuja su subgrafo asociado. Si se hace solo un clic sobre alguno de los nodos de la jerarquía, en la parte de abajo se muestra su nombre, su puntuación, la puntuación de sus hijos y el número de veces que ha sido mencionado en Twitter.

El cuadro en el que se visualiza el grafo es el objeto *Browser* que contiene un *WebView* tal y como se ha explicado antes. El cuadro inferior se compone tan solo de componentes *Label* de JavaFX, unos inalterables y otros que cambian su contenido según el nodo del árbol pulsado. El cuadro izquierdo está compuesto por un componente *TreeView* de JavaFX.

El encargado crear la vista de árbol es la clase *TreeCreator*. Esta clase contiene métodos para crear el *TreeView*, formado por elementos *TreeItemPair*. Aunque el árbol solo necesita mostrar objetos *String*, se ha decidido crear una clase propia que contiene el nombre del nodo y su identificador, y como el identificador es único, al hacer clic sobre un elemento del árbol se puede buscar en la base de datos sus modificadores y los de sus hijos más rápidamente.

TreeCreator también tiene un atributo con una referencia a un objeto *TreeController*. Esta clase mantiene el control de toda esta pantalla, y se encarga de actualizar la interfaz cuando se capturan eventos en alguno de los elementos con los que se puede interaccionar, además de sólo poder acceder él a la lógica de la aplicación, siguiendo así el patrón Modelo-Vista-Controlador y separando estas tres partes. Para esto mantiene referencias tanto a los elemento de la lógica de la aplicación como a los de la vista (Figura 2.10).

En el diagrama de clases se puede ver que el controlador mantiene una referencia a un objeto *Jerarquia*. Esta referencia se mantiene para evitar tantos accesos a la base de datos, al cargar la vista se carga también la jerarquía de la base de datos en memoria, con la puntuación y las menciones de sus nodos pero sin información de sus modificadores. Así cuando el usuario haga un clic sobre uno de los elementos de la jerarquía y haya que mostrar su información en el panel inferior no hace falta realizar un acceso a la base de datos.

También contiene una referencia a un objeto *GraphFileManager*. Al realizar doble clic sobre uno de los nodos de la jerarquía se debe mostrar la subjerarquía asociada a él con los modificadores más relevantes. Para esto se realiza una consulta a través de la clase estática *TraduccionDatos* y se escribe en los archivos HTML y JSON que necesita Alchemy.js de la manera ya explicada mediante la clase *GraphFileManager*.

En el diagrama también se pueden observar referencias a objetos *Chart* y métodos que esperan recibir estos objetos. En la siguiente pantalla de la aplicación también se muestra la jerarquía y se puede interaccionar con ella, pero la interacción resulta en cambiar un elemento distinto de la jerarquía que se comenta en el siguiente apartado.

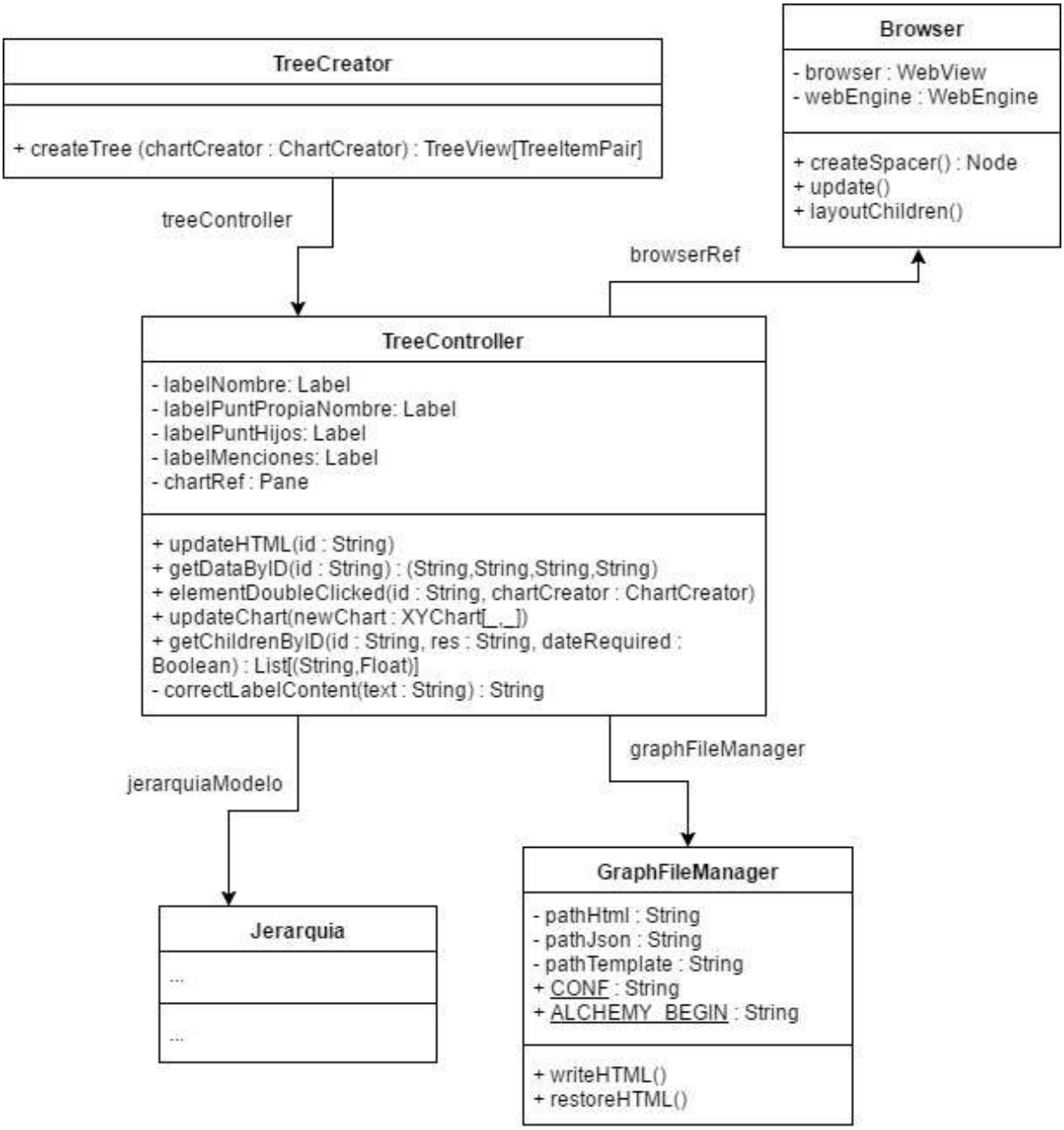

Figura 2.10: Diagrama de clases TreeController

2.5.4 Visualización de gráficas

En esta parte de la aplicación se busca mostrar la evolución de los atributos de los nodos a lo largo del tiempo, y mostrar resúmenes comparativos de la situación actual de estos atributos. Esta pantalla comparte el árbol desplegable de la pantalla anterior, y el cuadro inferior con la información de los nodos.

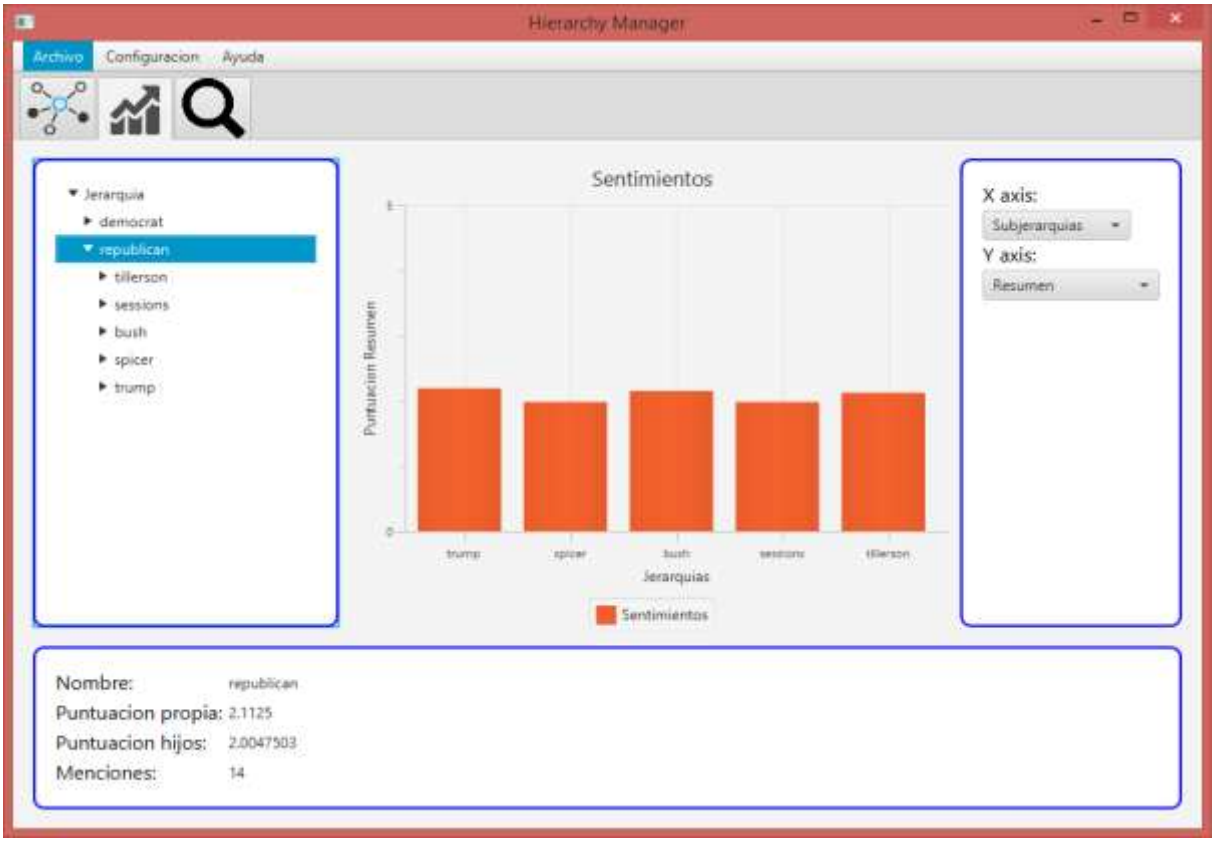

Figura 2.11: visualización del gráfico de barras

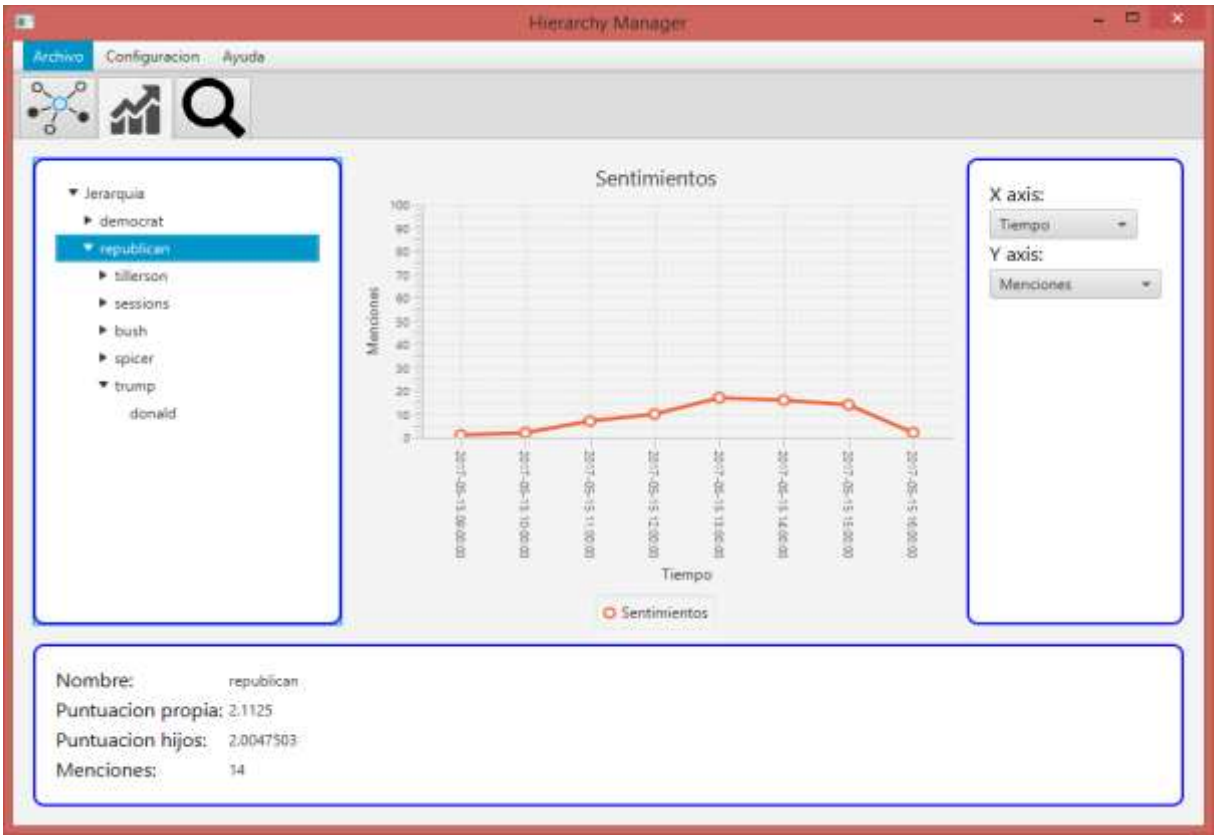

Figura 2.12: visualización del gráfico de línea

El funcionamiento del árbol es similar al de la pantalla anterior. Al hacer un clic sobre un elemento del árbol se muestra su información en el cuadro inferior; si se hace doble clic, en vez de cargar el grafo como se hacía en la otra pantalla, se carga la gráfica asociada a este nodo. La gráfica cargada por defecto es la que muestra un resúmen la media de la puntuación propia y de la puntuación de los hijos de sus nodos hijos.

Se ha añadido un nuevo elemento a la interfaz en la parte derecha de la pantalla, este cuadro contiene dos elementos que se encargan de cambiar la información que se visualiza en la gráfica. Se puede cambiar el atributo que se usa para el eje Y (resumen de la puntuación, puntuación de los hijos, puntuación propia y número de menciones), y también se puede cambiar el eje X, para poder mostrar por cada nodo o bien la información de sus hijos, o bien la evolución de los atributos en el tiempo.

De un modo parecido al que ocurre con la pantalla anterior, de la creación de las gráficas se encarga la clase *ChartCreator*. Al igual que *TreeCreator*, esta clase mantiene una referencia a un objeto *TreeController*, ya que la interacción del árbol cambia la gráfica que hay que dibujar, y además *ChartCreator* necesita acceder a la lógica de la aplicación para conseguir los datos de la gráfica. Cuenta también con dos atributos que establecen lo que se muestra en los ejes X e Y, estos atributos se cambian cuando se cambien los desplegables de la parte derecha de la pantalla.

El proceso de obtención es igualmente similar a los ya mencionados, con la ventaja de que en este caso se puede aprovechar también la información de la jerarquía que tenemos guardada en memoria. Si en el eje X se quiere mostrar la información de las subjerarquías de un nodo, tenemos toda la información de sus atributos guardada en la jerarquía en memoria igual que lo tenemos para mostrar la información en el cuadro inferior. Si en el eje X se quiere mostrar la evolución de los atributos según el tiempo, entonces sí que es necesario realizar un acceso a la base de datos; los datos se piden a *TreeController*, y éste a través de *TraduccionDatos* recoge los resultados en forma de una lista de pares (fecha, valor) que ya puede mostrar la vista.

El tipo de gráfica varía según se quiera ver la evolución de los atributos a lo largo del tiempo, o el valor de los atributos de las subjerarquías de un nodo. En el caso del primero, se ha usado un gráfico de líneas, útil para ver la evolución a lo largo del tiempo; en el segundo, un gráfico de barras, útil para comparar los valores entre las diferentes subjerarquías.

2.5.5 Búsqueda avanzada

La tercera pantalla principal de la aplicación corresponde a una pantalla de búsqueda avanzada. Para el diseño de esta pantalla se buscó si existía algún

paradigma de diseño de interfaces gráficas de búsquedas sobre bases de datos, y aunque no se encontró ninguno, sí que se encontraron ejemplos de búsquedas con lógica booleana. El diseño de esta pantalla se ha basado en la pantalla de búsqueda avanzada de listas de reproducción de iTunes [30].

Tal y como se puede observar en la figura 2.13, la pantalla de búsqueda se compone de una lista de condiciones, al pulsar el botón de “Buscar”, se buscan elementos de la base de datos que cumplan o bien todas las condiciones, o bien alguna de ellas. Las condiciones que un elemento debe cumplir se definen mediante una lista de desplegables que elegir y cuadros de texto que rellenar. Las condiciones son dinámicas, y dependiendo de los valores elegidos en los desplegables la interfaz cambia.

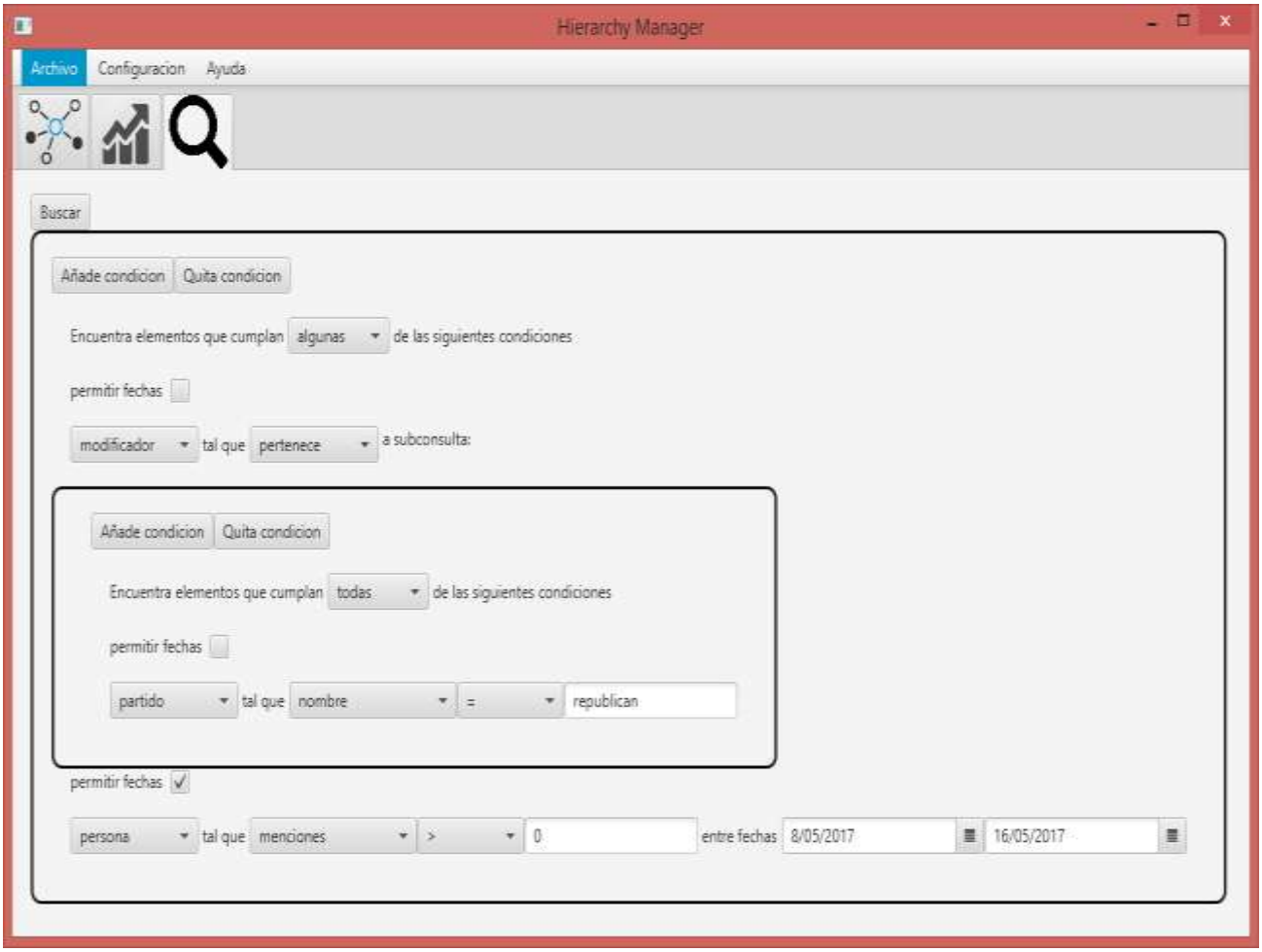

Figura 2.13: Ejemplo de Búsqueda Avanzada

En el primer desplegable se selecciona el tipo de nodos de la base de datos que interesa (los distintos tipos de entidades, elementos o modificadores). Una vez elegido el primero, en el segundo desplegable, o bien se selecciona un atributo del nodo del que se quiere que cumpla una condición, o bien se selecciona la opción “pertenece”:

- Si se selecciona un atributo, se abre un tercer desplegable y un cuadro de texto. En el tercer desplegable se ha de seleccionar el tipo de comprobación que se quiere hacer sobre el atributo (mayor, menor, igual…), y en el cuadro de texto el valor con el que se quiere realizar la comprobación.
- Si se selecciona la opción “pertenece”, se abre una segunda pantalla de búsqueda con más condiciones. La idea de esta opción es permitir que se busquen nodos hijos de padres que se quieren que cumplan ciertas condiciones. Esto permite buscar modificadores de nodos específicos, o elementos y entidades hijos de otros elementos o entidades. Por ejemplo, con esto se podrían buscar las personas del partido Republicano, o los modificadores del partido Republicano.

A las condiciones se les puede incluir también que permitan fechas mediante un checkbox. Esto hace que las condiciones que se incluyan deban cumplirse entre dos fechas que hay que especificar. Es decir, si se dice que las menciones deben ser mayores que 5 entre el 8/05/2017 y el 16/05/2017, se comprueba que durante esos días, el nodo haya sido mencionado más de cinco veces en total.

Una vez se pulsa el botón para realizar la búsqueda, se abre una ventana nueva con los resultados (véase Figura 2.14). La tabla contiene los resultados obtenidos en filas, conteniendo las columnas los valores de los atributos del nodo. Debido a que las entidades y los elementos tienen un atributo más que los modificadores, si el resultado de la búsqueda contiene nodos de ambos tipos, la ventana contiene dos tablas, una con los elementos y entidades y otra con los modificadores. Los resultados de la búsqueda se pueden ordenar haciendo clic sobre el nombre de la columna por el que se quiere que se ordenen.

La pantalla de búsqueda se ha programado usando tres clases:

- *AdvanceSearchWindow* representa la ventana de búsqueda. Contiene de manera separado un grid con el botón “Buscar” y otro grid con los botones de añadir y quitar condiciones, el desplegable que establece si se han de cumplir todas o alguna, y un array de *AdvanceSearch*.
- *AdvanceSearch* contiene el label para permitir fechas y el panel de un *SearchComponent*.
- *SearchComponent*: todos los desplegables, cuadros de texto y cuadros de fecha. También contiene una referencia a un objeto AdvanceSearchWindow que se inicializa si se selecciona la opción “pertenece” en el desplegable.

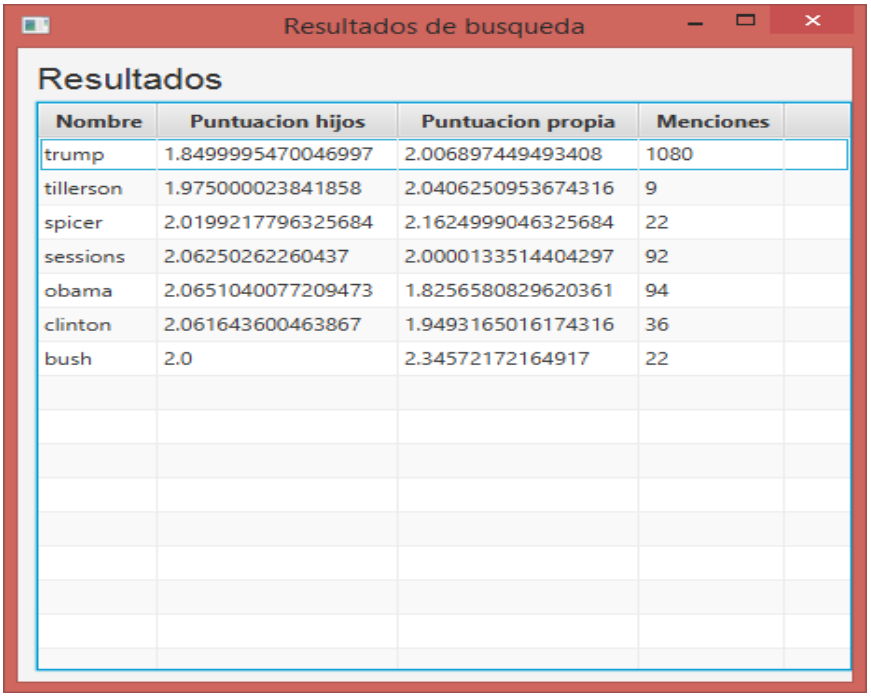

Figura 2.14: Ventana con los resultados de la búsqueda

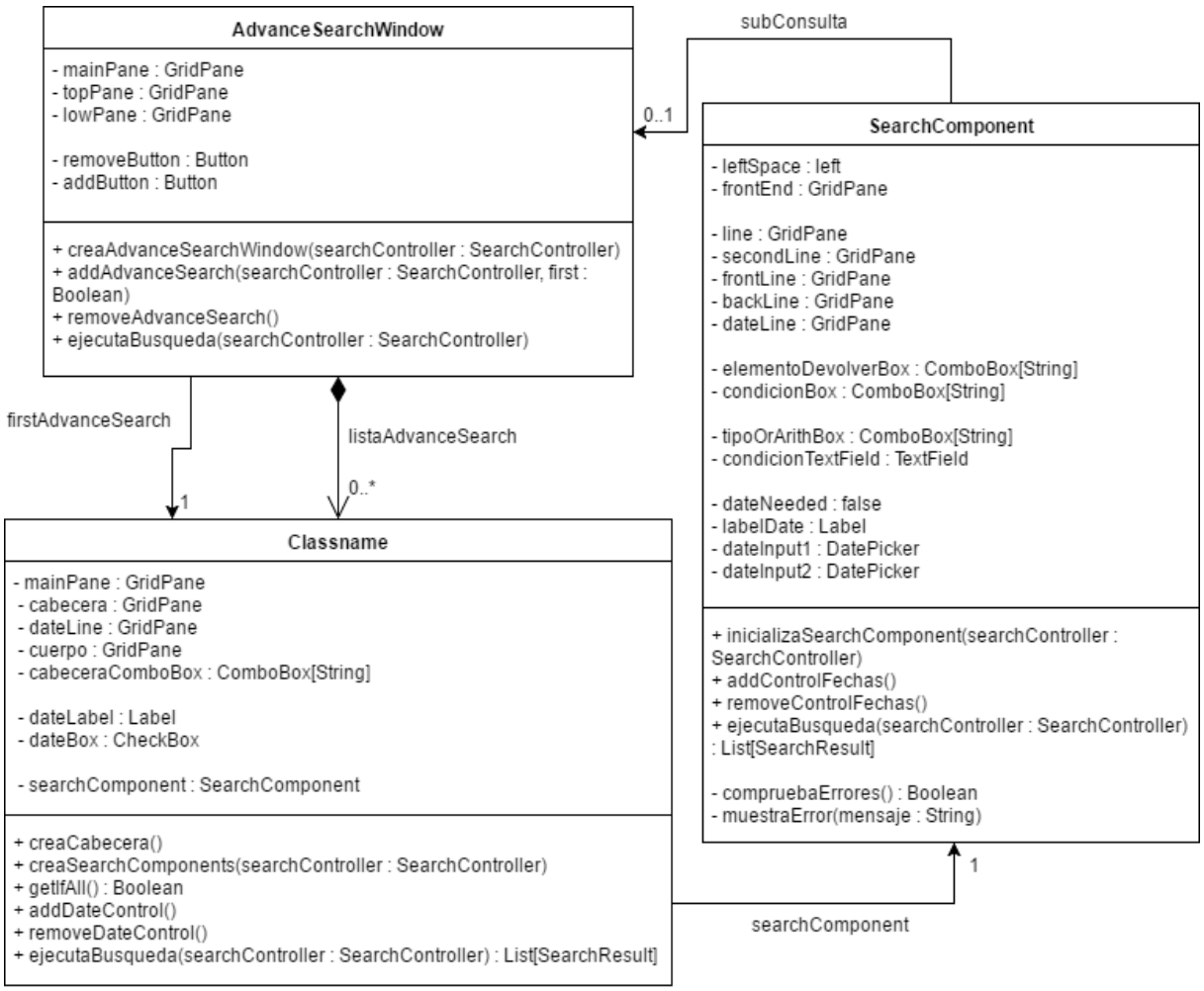

Figura 2.15: Diagrama de clases para la búsqueda avanzada

La ventana con los resultados está contenida en la clase *SearchResultsWindow*. Esta clase contiene las dos tablas que pueden aparecer en la ventana al mostrar los resultados, tal y como se ha dicho antes, una con las entidades y los elementos, y otra con los modificadores. Aunque no se ha podido usar polimorfismo para tener solo una tabla, sí que se ha podido usar para generalizar los métodos de búsqueda.

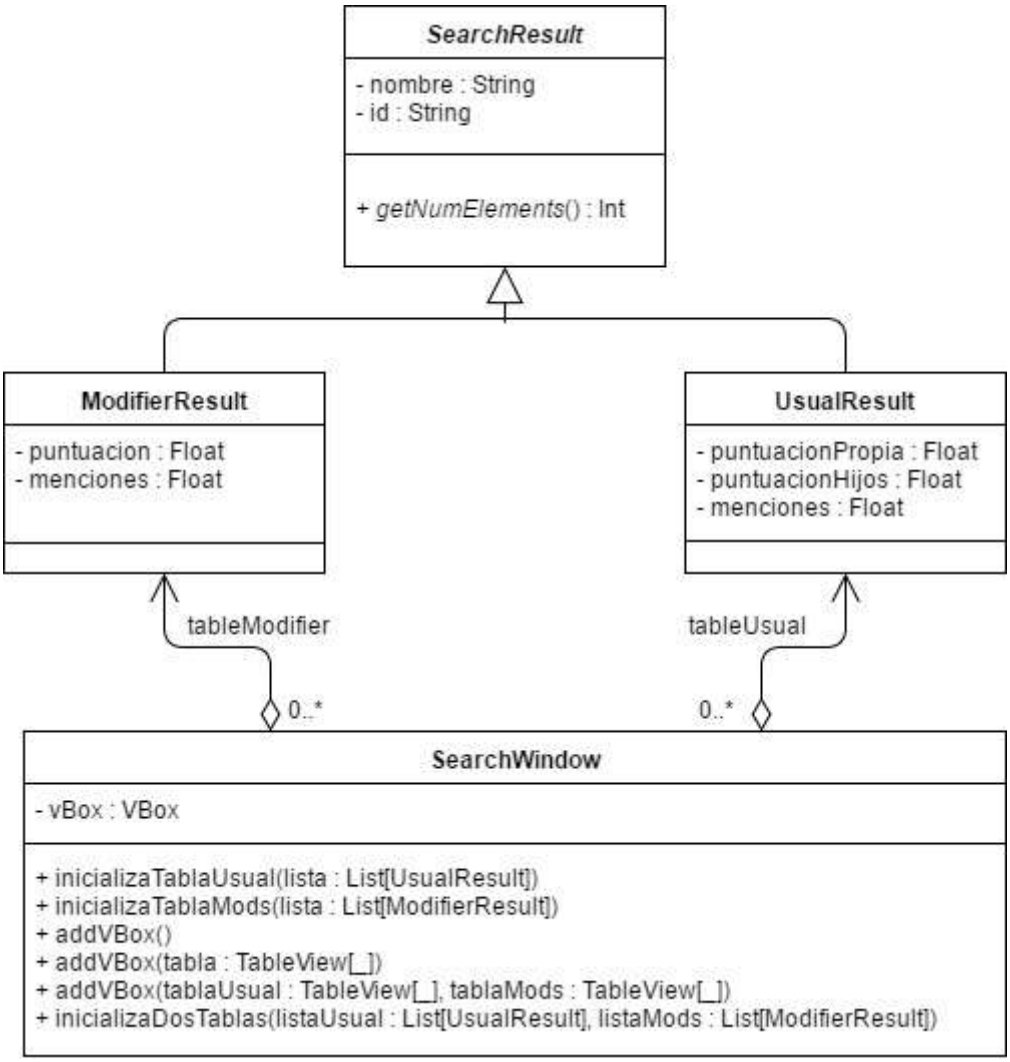

Figura 2.16: Diagrama de clases para los resultados de la búsqueda avanzada

La clase *SearchResult* es una clase abstracta que representa a cualquier resultado de la búsqueda, de esta manera se guardan como atributos el nombre y el identificador del nodo; si el nodo es un Modificador, se usa como identificador su nombre. La clase contiene también un método abstracto para conseguir el número de atributos que contiene la clase. De esta clase abstracta heredan las clases *ModifierResult* y *UsualResult*. La primera representa los resultados de la búsqueda que son modificadores y la segundo los que son entidades y elementos.

Una vez se pulsa el botón, el proceso de búsqueda sigue el siguiente proceso:

- Por cada condición de la pantalla de búsqueda se buscan los elementos que la cumplen. Si se ha seleccionado la opción “pertenece”, se resuelve la subconsulta primero, y después se recogen los elementos que estén relacionados con los obtenidos en la subconsulta.
- Tras conseguir los resultados de cada condición, se unen los resultados.
    - Si se ha seleccionado que los elementos tienen que cumplir todas las condiciones, se eligen solo elementos devueltos que aparezcan en los resultados de todas las búsquedas.
    - Si se ha seleccionado que cumplan alguna, se devuelven todos los elementos.

Para realizar el acceso a la base de datos, se necesita un controlador, llamado *SearchController*. Este controlador dispone de métodos que hacen de interfaz con la clase *TraduccionDatos* para realizar las consultas necesarias a la base de datos para obtener los resultados de las búsquedas. Asimismo, esta clase contiene un atributo que contiene una referencia a un objeto *RelacionesYAtributos*, para que los valores de los desplegables de la interfaz gráfica de búsqueda sean coherentes, y no aparezcan atributos de Entidades y Elementos si se quiere buscar nodos Modificador.

2.5.6 Pantalla de conexión a la base de datos

Es natural que se usen distintas bases de datos para guardar distintas jerarquías. Por esto, al principio de la aplicación se necesita incluir la dirección del servidor de la base de datos, el usuario y la contraseña. Durante la aplicación, en el menú “Archivo”, se puede cambiar la base de datos que se está usando.

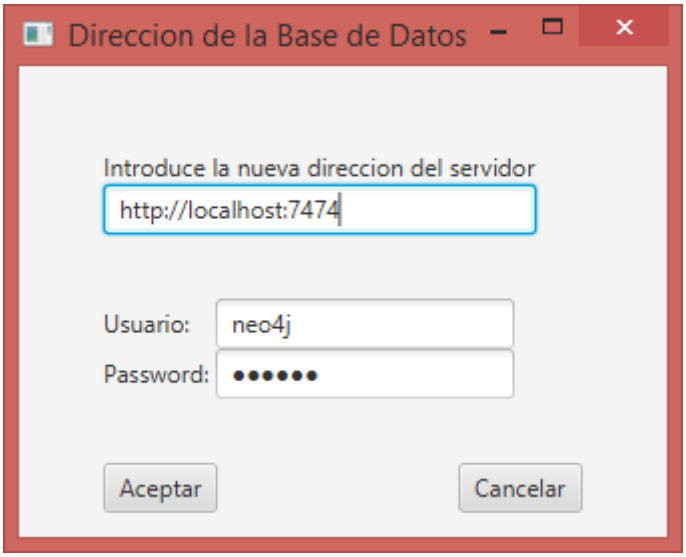

Figura 2.17: Ventana de conexión a la base de datos

Esta ventana está encapsulada en la clase *InputWindow*. La clase contiene como atributos los elementos de la interfaz: la ventana, los cuadros de texto de la dirección y el usuario, y el campo de contraseña para la contraseña. Además necesita un atributo booleano que indique si el cambio se hace de dirección se realiza durante la ejecución del programa o al inicio de la aplicación, y una referencia a un controlador para que pueda indicarle que es necesario recargar la vista y el modelo.

2.5.7 Ventana principal

La ventana principal de la aplicación está formada por dos elementos: un menú desplegable y un panel de pestañas. El panel de pestañas dispone de tres pestañas diferentes, una es el visor de grafos, otra el visor de estadísticas, y la última la pantalla de búsqueda avanzada.

El menú cuenta con tres opciones: “Archivo”, “Configuración” y “Ayuda”. Las dos últimas opciones no son funcionales en la versión final de la aplicación, pero inicialmente se pensó funcionalidad para ellas, y se detalla en el Capítulo 4. Si se interacciona con la opción de “Archivo” se puede seleccionar la primera opción para conectarse a una nueva base de datos, o la opción de “Salir” para cerrar la aplicación.

Para programar el menú se ha creado la clase *MenuConf*, una sencilla clase que presenta un método que devuelve el menú ya creado; y que cuenta con un solo atributo, una referencia al controlador principal de la aplicación, necesario al crear la nueva ventana de conexión a la base de datos.

De la creación de la ventana principal de la aplicación se encarga la clase *MainWindow*. Esta clase cuenta con un método *start* que es equivalente a lo que sería el método *main* en una aplicación Java usual. Este método se encarga de la inicialización de todos los controladores y de los elementos de la vista principal, así como de asignar los controladores con sus respectivas vistas. Un controlador principal se ha creado para evitar que cada controlador acceda al modelo de forma indiscriminada, y si los otros controladores desean acceso a los datos del modelo lógico, los deben conseguir del controlador principal. En el método *start* se inicializa el principal controlador *MainController*, y tras esto se inicializan *TreeController* y *SearchController* usando como referencias el modelo del primero. Se puede ver más información en la Figura 2.18.

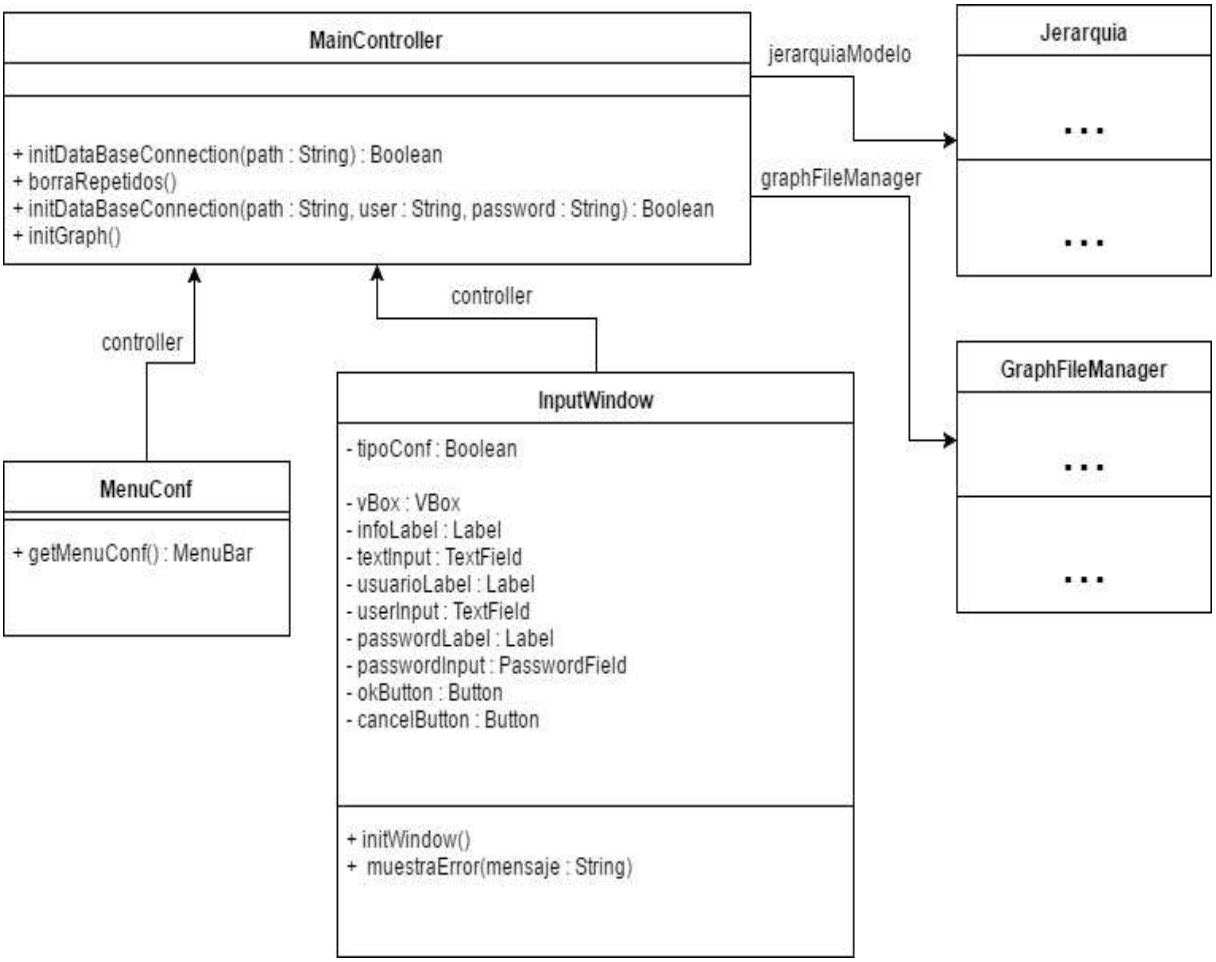

Figura 2.18: Diagrama de clases de MainController

Capítulo 3 - Resultados

El proyecto cuenta con una amplia variedad de metodologías aplicadas en las distintas partes de la aplicación: análisis del texto, análisis de sentimientos, la aplicación Spark, transacciones y almacenamiento en la base de datos, interfaz gráfica, etc. El propósito de este capítulo es presentar los resultados obtenidos en estas partes, analizar si son correctos y buscar en qué fallan si no lo son.

3.1 Análisis del texto

Este apartado se refiere al análisis de los tuits, a la identificación de las entidades de la jerarquía en la frase y a la búsqueda de palabras relacionadas con esta. A continuación se muestran subjerarquías con sus modificadores "relevantes" encontrados tras ejecutar el programa durante 12 horas, en las cuales se recogieron alrededor de 3.200 tuits, mostrados usando la interfaz gráfica del programa.

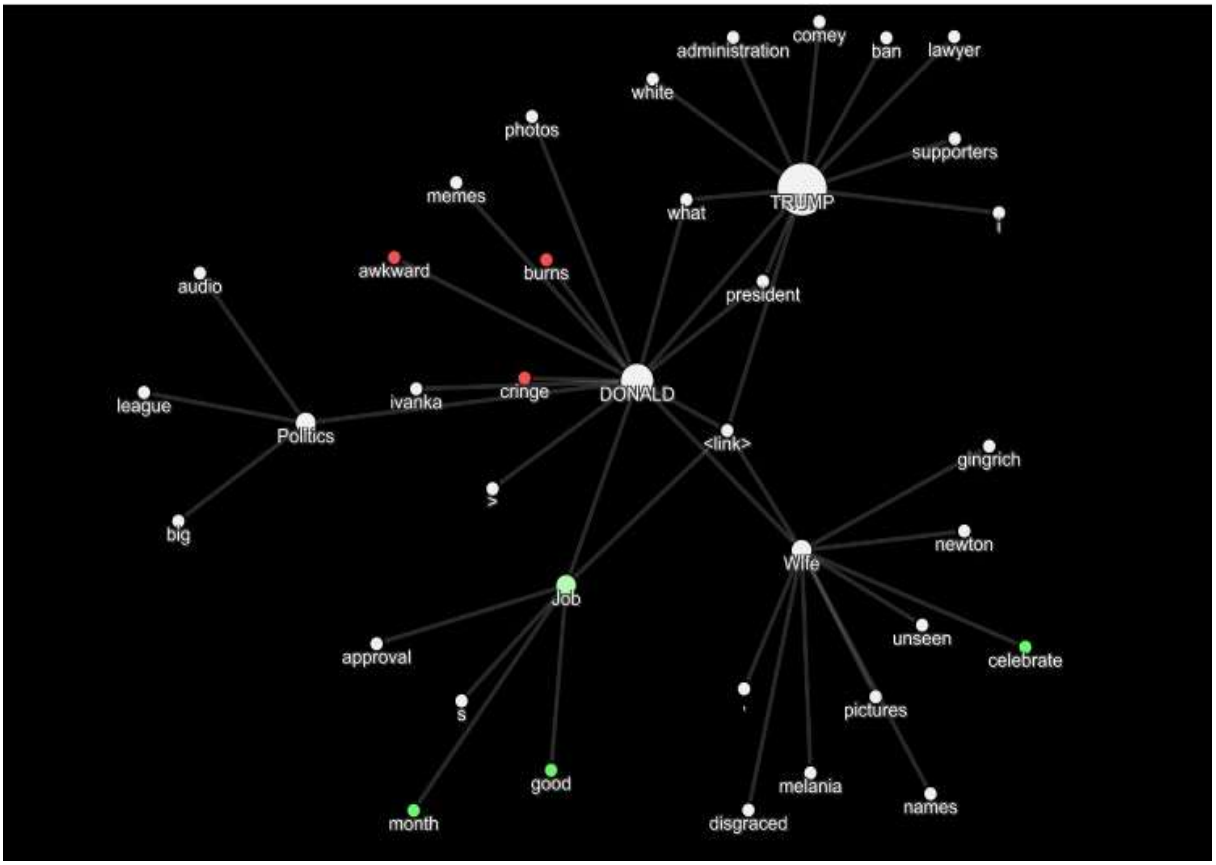

Figura 3.1: Grafo Trump

En este grafo se ve la jerarquía con entidades "Donald" y "Trump", y elementos "Job", "Politics" y "Wife"asociados a "Donald". La mayoría de los modificadores asociados cumplen los objetivos buscados al analizar las frases, sin embargo, se encuentran algunos que no deberían estar. Asociado a "Wife" se encuentra un nodo

con una coma, a “Donald” un nodo con el signa de mayor, a “Job” el signo del dólar, y a Trump el pronombre “I” (“yo” en inglés).

Se ha observado que la biblioteca de Stanford reconoce símbolos como si cumplieran funciones de palabras en ciertas ocasiones, por ejemplo si la palabra es “Donald Trump loves $”, la biblioteca trabaja con el símbolo del dólar y este se usa como complemento directo, explicando que aparezca en el grafo anterior. De igual manera, en una frase del estilo “Donald Trump > Hillary”, aunque Stanford no reconozca qué es exactamente “>”, sí reconoce una dependencia entre ellos, y por eso se incluye en el grafo. Esto también puede ocurrir con frases mal construidas o aquellas con emoticonos. Una posible solución es añadir los símbolos que no se quiere que aparezcan en el grafo en la lista negra que admite el programa.

También se ve en el grafo la palabra “I”, y en la programación se asegura que los pronombres no se guarden como nodos Modificadores. La única posibilidad es que la biblioteca no lo reconozca como pronombre en ciertas frases, ya sea porque de verdad no se usa como pronombre (a veces se usa la “i” mayúscula como número romano) o ya sea porque lo malinterprete.

Tampoco se pretende que nombres propios puedan funcionar como modificadores, sin embargo, se puede observar en el grafo como “Ivanka” es modificador de “Trump”, y “Melania” es modificador de “Wife”. Esto ocurre porque la biblioteca de Stanford no reconoce ciertos nombres propios si no comienzan como mayúscula.

Fenómenos idénticos se observan comprobando los grafos de otros nodos. En la subjerarquía de la familia Obama (Figura 3.2), se puede ver como la palabra “i” también aparece; y se observa que aparecen como modificadores los nombres propios “james” y “michelle” que no han sido reconocido como tales.

SI observamos grafos cuyas entidades han sido mencionadas menos veces en Twitter no encontramos que ocurra lo anterior, sino otros fenómenos. En la Figura 3.3. se puede ver cómo aparecen modificadores como “omg”, “!!!” o “af”. Estas palabras no aportan información relevante, y sería posible añadirlas a la lista negra para que no se trataran; sin embargo, aparecen debido a las pocas menciones que han tenido los nodos, si hubieran sido más mencionados este tipo de palabras desaparecen.

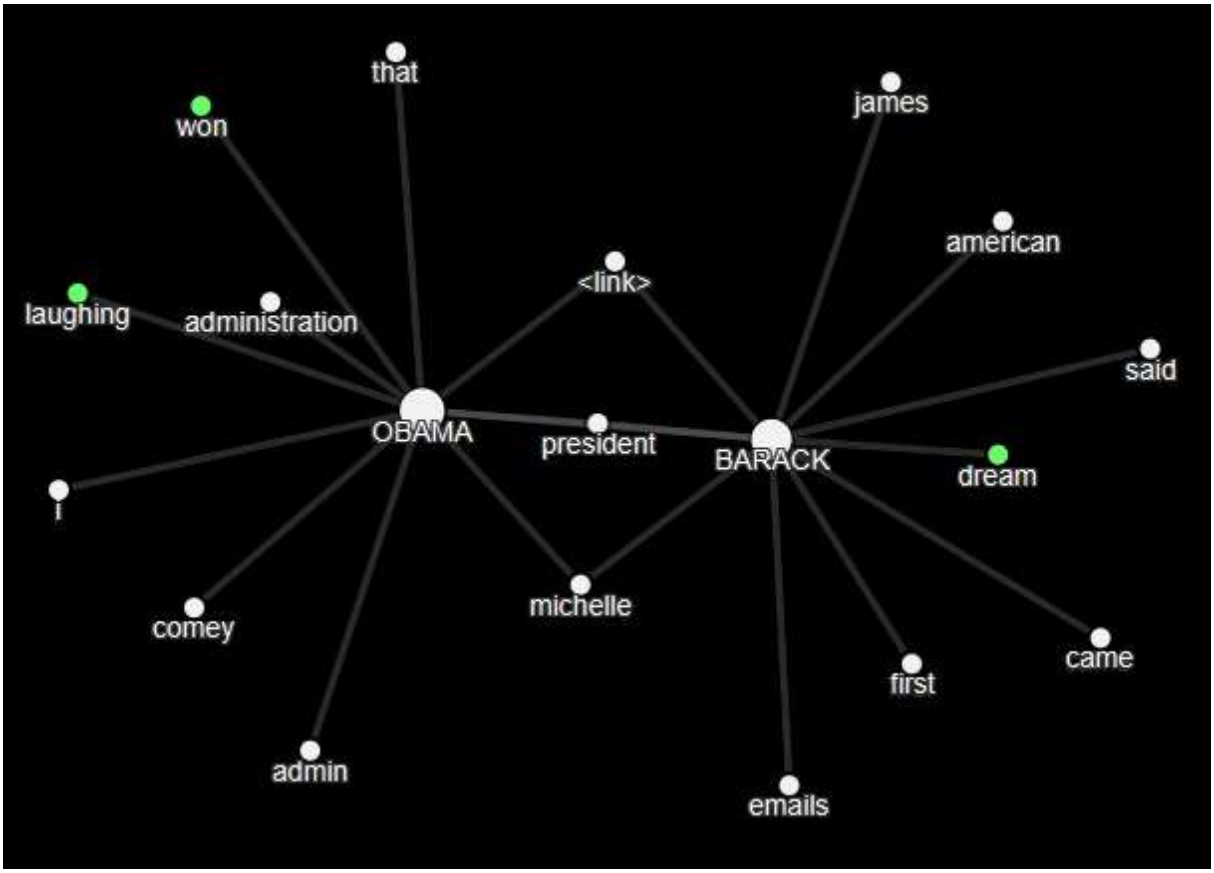

Figura 3.2: Grafo Obama

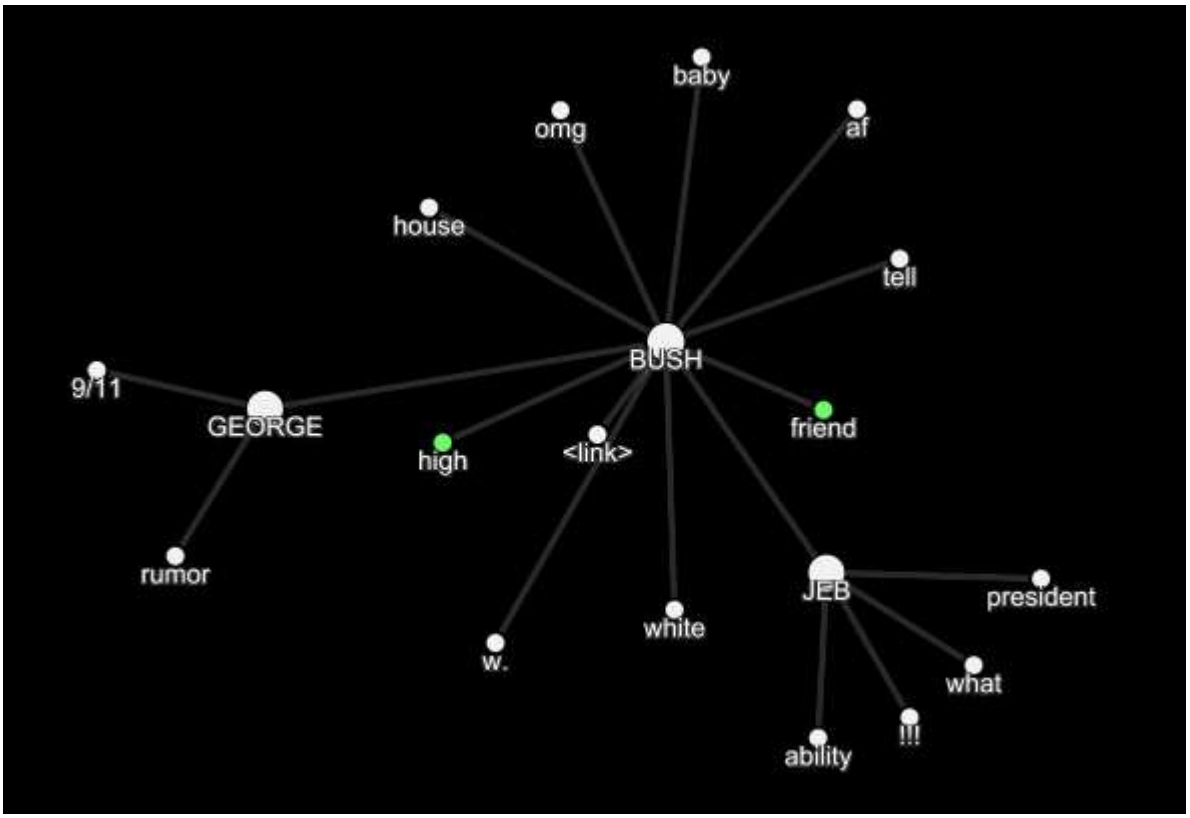

Figura 3.3: Grafo Bush

Se ve también en las figuras 3.1, 3.2 y 3.3 que aparecen las palabras “that” y “what”. Estas palabras tampoco aportan información relevante al grafo en el que aparece, y además es bastante frecuente que aparezcan en frases. Aunque se haya establecido en el código no tratar los artículos, “that” y “what” pueden funcionar también como adverbios, pronombres e incluso adjetivos. Como sí que nos interesan ciertos adverbios y adjetivos, no es posible hacer una regla general para que “that” y “what” no aparezcan, la única manera de garantizar que no aparezcan más sería incluirlas en la lista negra.

3.2 Análisis de sentimientos

Esta sección corresponde al estudio de los resultados centrándose en el análisis de sentimientos, esto quiere decir que se repasarán las palabras encontradas durante la ejecución del programa y se analizará si se les ha asignado un sentimiento correcto, o por lo menos, intuitivo.

En las figuras 3.2. y 3.3 se observan palabras con sentimiento positivo generalmente bien asignado, aunque la palabra “high” puede ser una excepción ya que necesariamente no es algo bueno. En la figura 3.1 se puede ver un resultado parecido, en el que las palabras a las que se ha asignado sentimiento generalmente lo tienen, aunque igual que antes, la palabra “month” no es necesariamente algo positivo. Lo que sí que se encuentra en la figura 3.1 son palabras neutras que sería lógico que fueran positivas o negativas; por ejemplo, “disgrace” asociado a “Wife” sería lógico que tuviera una posición negativa, lo mismo la palabra “ban”; en un grado inferior, también sería posible que las palabras “supporters” y “approval” tuvieran puntuación positiva.

Peores resultados se ven en la figura 3.4. En este caso se puede ver cómo casi todas las palabras que cuentan con sentimiento positivo no parece lógico que lo tengan, “wind” debería ser neutra, y tanto “thriller” como “arresting” tendrían sentido que fueran negativos o neutros.

En la figura 3.5 se encuentra algo más parecido a lo encontrado en la figura 3.1. Se observan varios nodos que debían tener puntuación negativa y tienen puntuación neutra; “punk” usado a veces como insulto podría ser negativo, “trouble”, “worried”, “tough” y “sentencing” también podrían ser negativos.

Aunque en general los resultados son aceptables, existen bastantes casos en los que es posible una mejora. Los dos analizadores de sentimientos que se usan funcionan medianamente bien, pero usar un número mayor de ellos y aplicar un sistema de votación conseguiría una mayor diversidad de sentimientos.

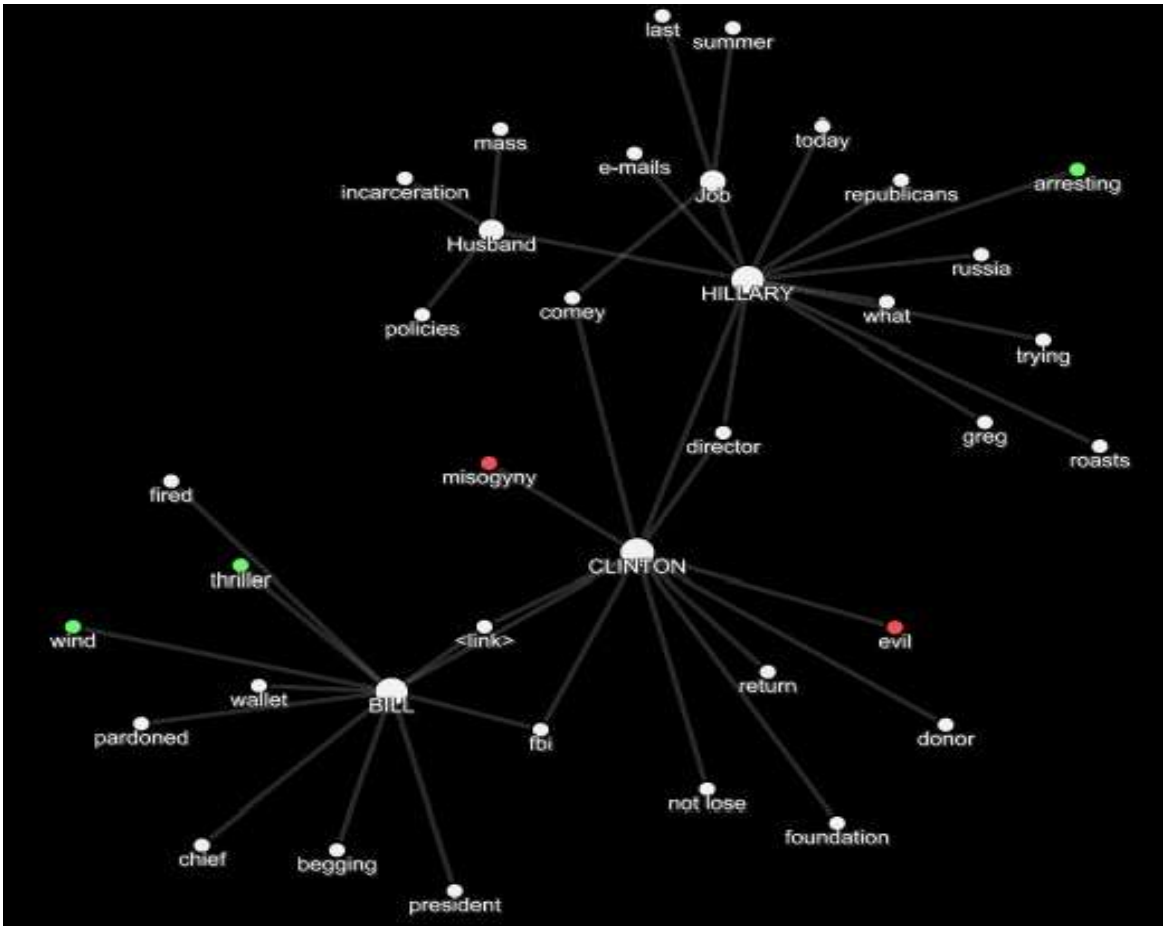

Figura 3.4: Grafo Clinton

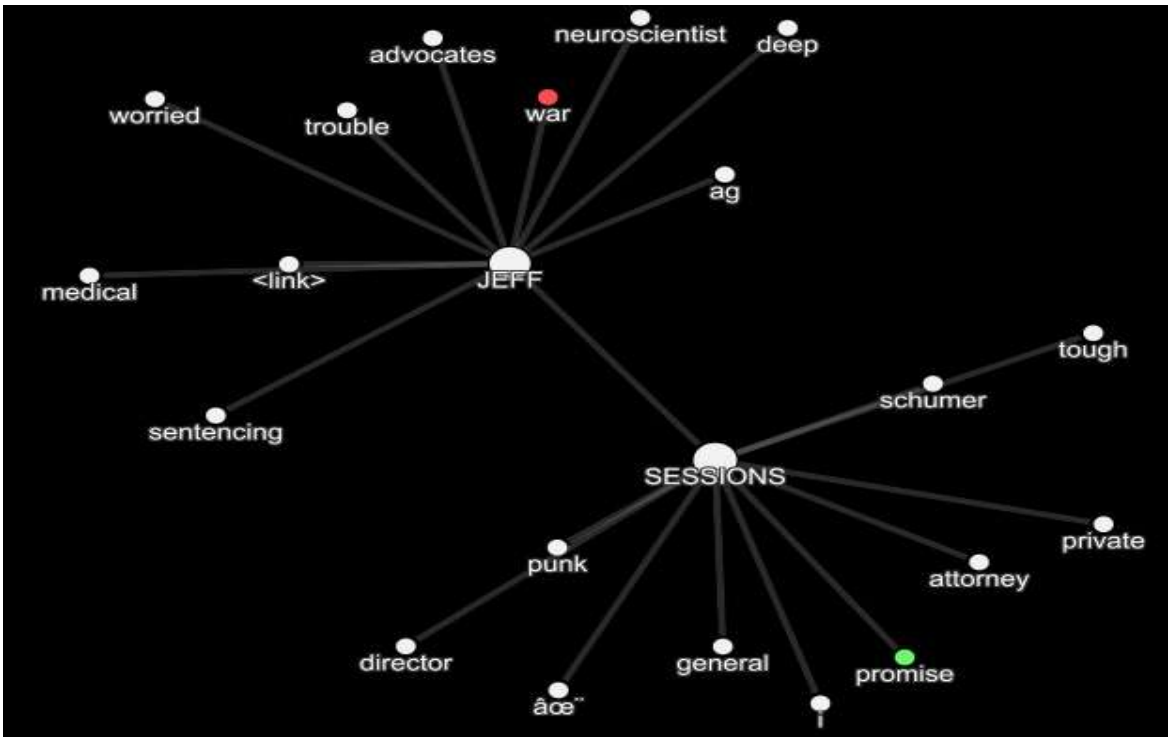

Figura 3.5: Grafo Sessions

3.3 Rendimiento de la aplicación Spark

En este apartado se tiene como objetivo analizar el número de tuits que se recogen usando la aplicación Spark. Hay que tener en cuenta que los resultados estudiados en este apartado dependen en gran manera de el sistema en el que ejecuta. En este caso las pruebas han sido realizadas con un solo ordenador portátil Windows 8 con dos procesadores, y una conexión a Internet de 50 Mb/s. También hay que tener en cuenta que Twitter limita el número de tuits obtenibles a un 1% del total de los tuits generados, lo que suele equivaler a 60 tuits por segundo.

Para poder comparar los resultados con los que se obtengan en otro equipo, primero se va a comprobar recogiendo todos los tuits que estén en inglés, y guardando un nodo por cada uno de ellos en la base de datos. De esta manera se prueba cuántos tuits se consigue independientemente de la jerarquía que se use.

Se aplicó este experimento varias veces durante varios días en las mismas horas, y se obtuvieron siempre los mismos resultados, sólo variando los tuits recogidos en las nueve de la mañana y las nueve de la noche de la noche entre 160 y 360.

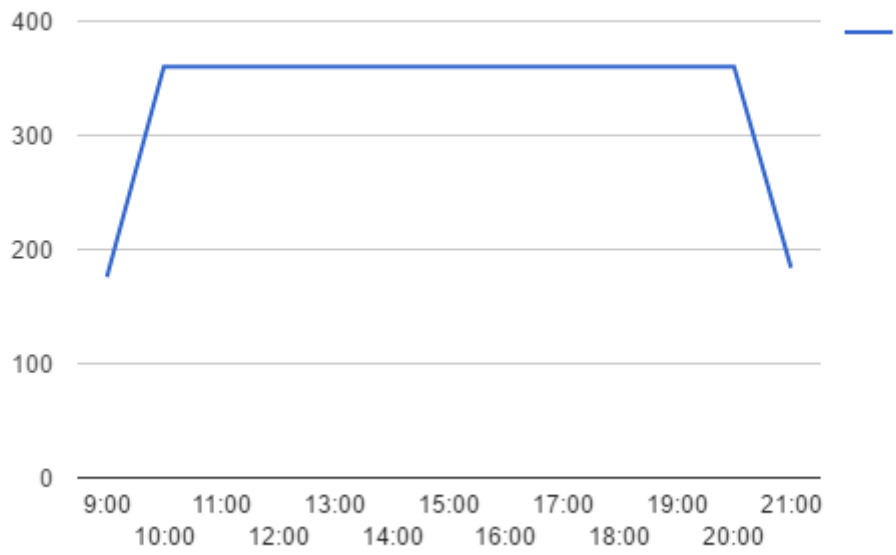

Figura 3.6: Tuits recogidos por hora

Ahora, si usamos la jerarquía de políticos mostrada en los apartados anteriores, se puede ver el número de veces que han sido mencionados los apellidos de los políticos en las siguientes figuras:

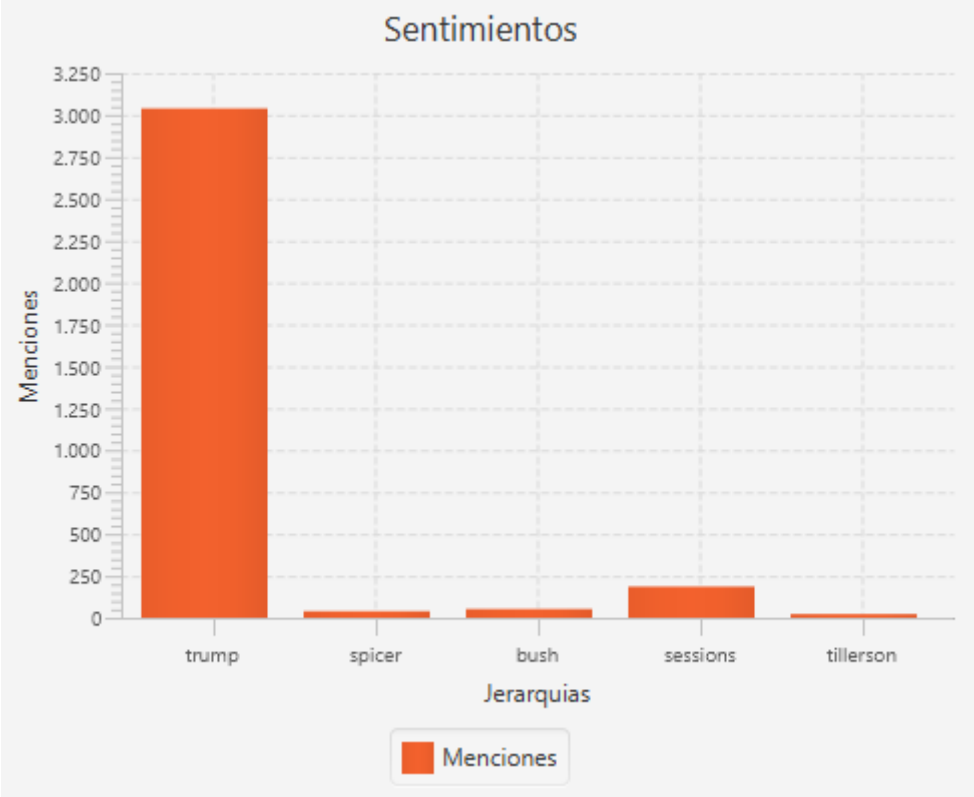

Figura 3.7: Menciones partido Republicano

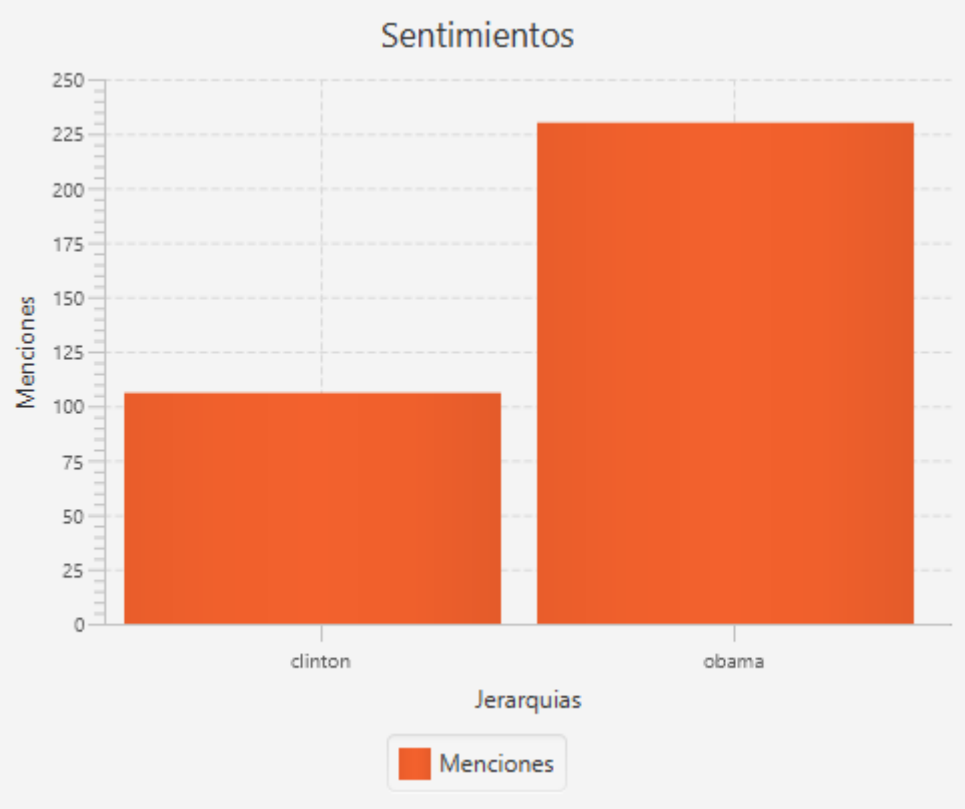

Figura 3.8: Menciones partido Demócrata

En la figura se puede observar que el número de tuits capturados en los que aparece algún miembro de la jerarquía es similar al número de tuits capturados que

no deben cumplir ninguna restricción en particular salvo estar en inglés (de hecho en el caso de Trump se capturan más, algo posible, ya que un tuit puede estar formado por más de una frase, y así se puede obtener más de una mención por tuit). Si se usará un clúster de computadores con una conexión más rápida seguramente la diferencia aumentaría, ya que el sistema consumiría tuits de manera más rápida, y no todos los tuits contienen alguna de las entidades de la jerarquía.

En las figuras también se puede ver cómo varía la relevancia de estas familias de políticos en Twitter. En particular se habla de Trump diez veces más que de secretarios de su gobierno y de los miembros de la familia Bush, y también diez veces más que los miembros de su oposición; es también interesante notar que durante el tiempo que el programa se estuvo ejecutando se habló de los Obama el doble que de los Clinton; de hecho, se habló de los Obama más que de Jeff Sessions, que durante el tiempo en el que se recogieron tuits fue protagonista de una controversia.

En las siguientes figuras (3.9, 3.10, 3.11) se puede observar una evolución de las menciones según la hora. Esto no sólo nos sirve para recabar información particular de los políticos que se han usado para probar el programa, sino también para conseguir información general de Twitter, por ejemplo, comparando gráficas distintas se pueden inferir las horas en los que los tuiteros están más activos.

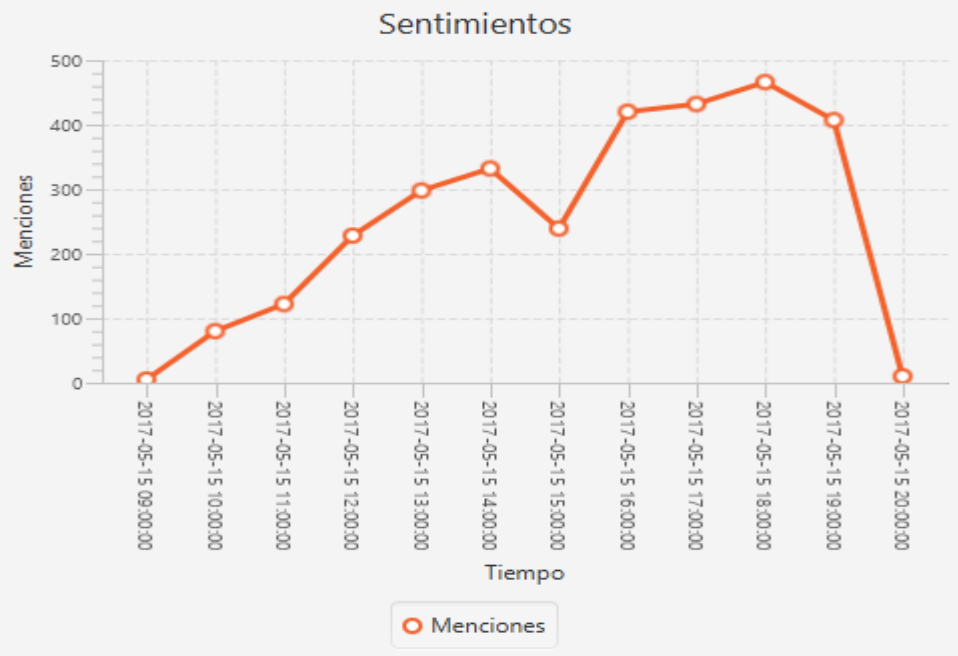

Figura 3.9: Menciones a lo largo del tiempo Trump

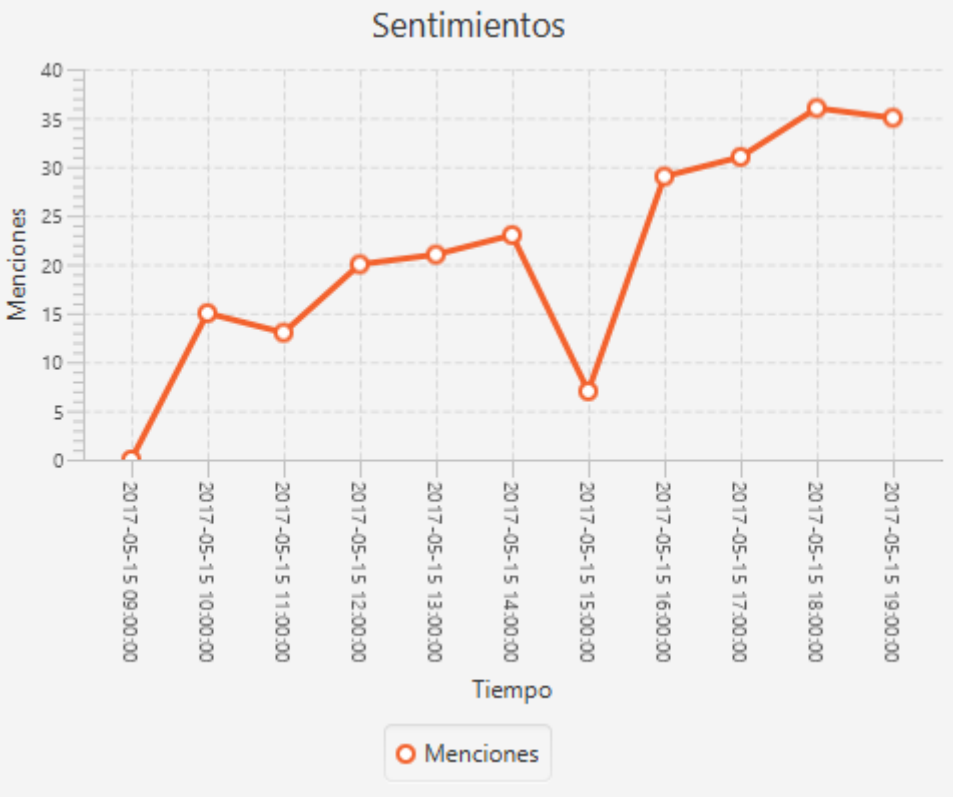

Figura 3.10: Menciones a lo largo del tiempo Obama

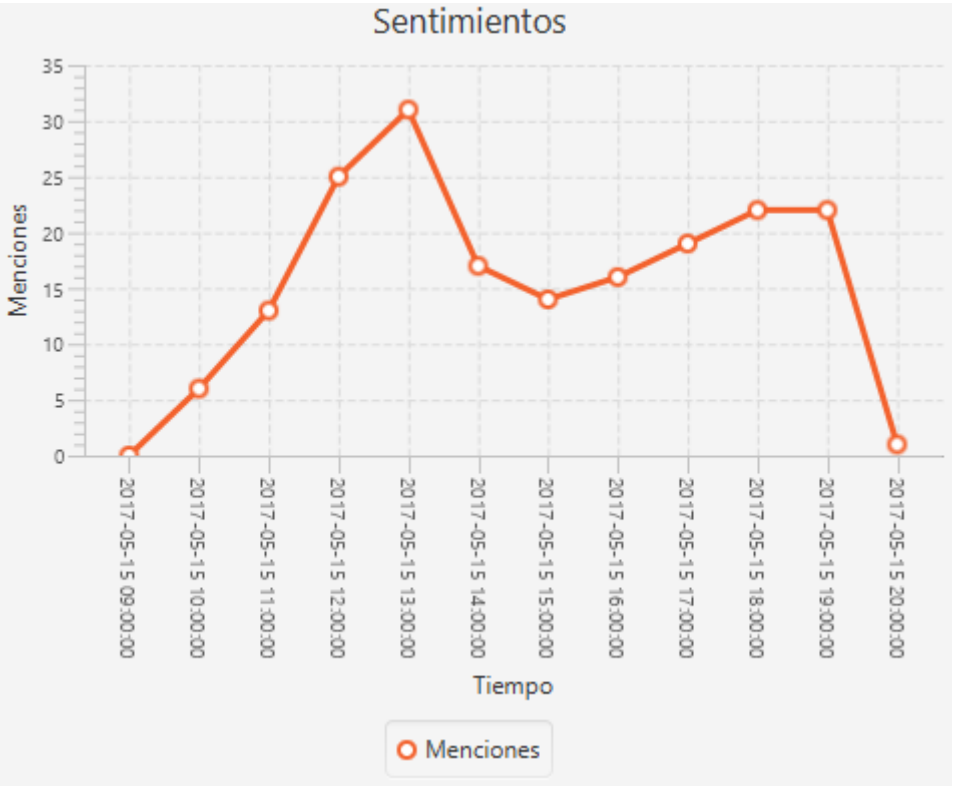

Figura 3.11: Menciones a lo largo del tiempo Sessions

Es importante decir que a veces durante la ejecución de la aplicación Spark, se interrumpe el flujo de datos si tiempo que se tarda en tratar un elemento y recoger el siguiente es mayor de dos segundos. El procesamiento que requiere realizar el análisis sintáctico, de sentimientos y la modificación de la jerarquía tarda a veces más de este tiempo, por lo que se pierde la conexión al flujo y esto provoca una excepción durante la ejecución. La excepción no provoca que el programa deje de funcionar, tan solo provoca que uno de los hilos que recogen información del flujo acabe, y se abra uno nuevo con una nueva conexión.

Estas interrupciones retrasan la velocidad a la que se recogen los tuits, pero si comparamos el número de tuits recogidos sin ser tratados (no se producen estas interrupciones) con el número de menciones de las figuras anteriores, se puede ver que no implica una diferencia grande. Aun así, si se dispusiera de un equipo más potente se reduciría el tiempo anterior y el número de excepciones de este tipo se reduciría.

3.4 Base de datos

Este apartado tiene como fin comprobar si el rendimiento ofrecido por la base de datos y el espacio ocupado son adecuados y cómo mejorarlos. De esta última parte ya se ha hablado en el apartado 2.4, pero en este apartado se centrará en el número de nodos y relaciones generados, y en el espacio absoluto que ocupa la base de datos y estudiar de dónde proviene.

El rendimiento ofrecido por Neo4J es adecuado, ya que garantiza hasta millones de escrituras de registros por segundo; aún así, se puede comprobar usando el lote de pruebas anterior en los que se guarda un nodo en la base de datos por cada tuit en inglés encontrado, de esta forma se comprobó que no se produjo ninguna excepción de las mencionadas antes.

Si analizamos el estado de la base de datos tras ejecutar el programa durante casi 12 horas, se obtiene que el espacio ocupado por la base de datos es de 470 MB, estando la mitad de este espacio ocupado por el log lógico, un historial de las transacciones de la base de datos. En la base de datos se han creado un total de 3.486 nodos Modificador, y hay un total de 17.558 relaciones entre nodos.

Para enlazar este dato con lo comentado en el apartado 4 del capítulo 2, hay 3.975 relaciones con 2.794 nodos Modificador que han sido mencionados menos de cinco veces. Eliminar estas relaciones y los nodos supone reducir en un 23% el número de relaciones y en un 80% el número de nodos Modificador. Tras limpiar estos nodos, se aplicó también un método de compactación de las relaciones, juntando las relaciones del mismo día, y se consiguió reducir el número de relaciones a 1671; es decir aplicando los anteriores procedimientos de limpieza y

compactación se consigue tener diez veces menos relaciones, aunque evidentemente se pierde la información de cómo ha evolucionado la información de las entidades por hora.

Sorprendentemente, el tamaño de la base de datos ha aumentado a 473 MB tras eliminar estas relaciones y nodos. Esto sucede porque la mayoría del tamaño lo ocupa el log lógico (220 MB) como ya se ha dicho antes, y este ha aumentado de tamaño por las transacciones correspondientes a la limpieza y compactación. Es importante notar que se puede limitar la cantidad de logs generados y el espacio máximo que pueden ocupar, en el caso presentado aquí el tamaño es tan alto porque se han realizado varias pruebas con la base de datos durante meses sin establecer un límite de los logs.

El resto del espacio de la base de datos, está ocupado de la siguiente manera:

- 956 KB ocupados por la información de las propiedades de los nodos. Este tamaño tan alto se debe a que los sentimientos de los nodos se guardan usando un número en coma flotante, y que las fechas se guardan un entero largo.
- 160 KB ocupados por los nodos.
- 510 KB ocupados por las relaciones.

Con estos datos se puede concluir que es muy importante la limpieza de nodos no relevantes (al eliminar los nodos se eliminan sus propiedades), y la compactación de relaciones (se reduce no sólo el tamaño ocupado por las relaciones sino también el tamaño ocupado por las propiedades de éstas).

3.5 Interfaz gráfica

La interfaz gráfica accede a la base de datos para mostrar el grafo, mostrar las estadísticas y realizar búsquedas. En el capítulo 2 ya se ha hablado de las limitaciones de la interfaz y de la necesidad de reducir el número de nodos que mostrar en el grafo. En este apartado se va a hablar del tiempo de respuesta de las distintas partes de la interfaz, primero usando la base de datos original tras la ejecución anterior, y después tras limpiar y compactar la base de datos. Además, también se analizará si la estrategia para decidir qué nodos se muestran es buena y si los nodos mostrando realmente aportan información.

Al iniciar la aplicación, una vez se ha establecido la conexión con la interfaz gráfica, usando la base de datos mostrados en los apartados anteriores, sin realizar ningún tipo de compactación ni limpieza, la aplicación tarda 2 minutos y 15 segundos en mostrarse, y tarda otros 10 segundos en dibujar el grafo y responder totalmente a interacciones. Esto se debe principalmente a que busca los 10 nodos Modificador más relevantes entre los 3.500 mencionados antes, y para esto debe recorrer

también las 17.500 relaciones según el nodo al que modifican y la fecha para calcular el número de veces que se han repetido.

Si aplicamos uno de los procedimientos de limpieza, y eliminamos los nodos Modificador que han aparecido menos de cinco veces, la aplicación pasa de 2 minutos y 15 segundos a 22 segundos en mostrarse, y tarda otros 8 segundos en dibujar el grafo y poder responder a interacciones: un cuarto del tiempo que ocupaba originalmente.

SI tras esto aplicamos el método de compactación por día, se consigue reducir el tiempo de inicio a 15 segundos, y el de dibujo del grafo y respuesta a 7 segundos. Se puede comprobar que no es tan importante compactar relaciones como sí lo es eliminar los nodos Modificador no importantes, de cara a la eficiencia de la interfaz gráfica.

Capítulo 4 - Trabajo Futuro

Este capítulo pretende proponer mejoras del proyecto vistos los resultados del capítulo anterior. La sección se organizará de la misma manera que la anterior, estableciendo mejoras para las distintas partes de la aplicación. A lo largo de la memoria ya se han mencionado ciertas posibles mejoras, en este apartado no sólo se pretende ofrecer otras que no han sido mencionadas, sino también ampliar las ya mencionadas.

4.1 Análisis de texto

Hay una gran variedad de mejoras que se pueden aplicar en esta parte de la aplicación, algunas más sencillas de implementar en el estado actual del proyecto, y otras que requerirían más trabajo.

Una mejora inmediata que se podría aplicar es usar llevar un control del número de relaciones que se recorren al analizar el grafo de relaciones de la frase. Lo que se hace actualmente es recorrer de manera transitiva todos los nodos relacionados con las entidades, de tal manera que si la entidad está relacionada con una palabra *x*, y la palabra *x* está relacionada con una *y*, se buscan las palabras que estén relacionadas con *y*, y esto se repite hasta que ya no hay palabras por recorrer. Llevando una cuenta de la longitud del camino se podrían decidir parar una vez el camino supere un límite a definir.

Otra mejora puede ser tratar las relaciones distintamente según su tipo. En la versión actual de la aplicación las únicas relaciones con las que se realiza un tratamiento distinto a las demás son las negaciones y las modificaciones adverbiales. Se podría juntar esta mejora con la anterior, de tal manera que dependiendo del tipo de relación se considere un número distinto de nodos que recorrer. Por ejemplo, se podría restringir el camino a seguir cuando se encuentra una oración subordinada al tratar el sujeto o el complemento directo a dos nodos, pero seguir dejando que al tratar adjetivos y adverbios se siga un camino teóricamente infinito (en la práctica toda frase es finita).

En la descripción del estado del arte se ha hablado de varios métodos de identificación de características. En este proyecto se han usado técnicas de las mencionadas en los artículos para identificar palabras relacionadas con las entidades y para el análisis de sentimientos, sin embargo no se ha aplicado ningún algoritmo específico que busque identificar exclusivamente las características. Podrían elegirse una combinación de algoritmos descritos en los artículos y añadir una fase al análisis del texto que consista en una identificación de características, y luego aplicar el algoritmo actual, tratando a las características encontradas como si fueran los elementos (también llamados intereses en esta memoria) que se definen

en los XML de entrada. Este cambio es sustancial y seguramente implica más esfuerzo que los anteriores.

Mejorar la identificación de nombres propios, símbolos y pronombres es también una parte a mejorar de la aplicación, actualmente se puede resolver el problema añadiendo pronombres, símbolos y nombres propios a la lista negra, pero es una solución poco flexible. Para evitar esto se podría usar, además de la biblioteca de Stanford, otras bibliotecas distintas que ofrezcan reconocimiento de nombres propios y análisis morfológico de frases. El procedimiento a seguir podría ser parecido a lo que se hace actualmente en el proyecto con el análisis de sentimientos, y en caso de que las bibliotecas catalogasen de manera distinta las palabras, se sometería a votación.

Es posible incluso intentar también identificar nodos de la jerarquía según su género. Es decir, si se dispone de la frase “Trump is a really handsome man” y de una jerarquía de entrada en la que el nodo “Trump” tenga como hijos “Donald” e “Ivanka”, debido a que “man” aparece en la frase, saber que se refiere al nodo “Donald”. Esto requeriría cambiar la entrada para que se pudieran definir géneros y otra información para los nodos de la jerarquía, además de cambiar el algoritmo para que hiciera este tipo de reconocimiento.

Nótese que todos estos cambios supondrían también un aumento en la complejidad del cómputo del algoritmo. Un algoritmo más lento provocaría que la aplicación Spark bajara su rendimiento y que se pierdan tuits que analizar. Hay cambios en la arquitectura del proyecto que podrían evitar que esto ocurriese, y que se detallarán más adelante en el siguiente apartado.

4.2 Análisis de sentimientos

Desde el punto de vista de análisis de sentimientos también hay una gran cantidad de mejoras que se pueden aplicar. Una mejora ya mencionada antes es añadir más fuentes distintas de análisis de sentimientos. Ahora se usa el análisis de la biblioteca de Stanford, que usa un algoritmo entrenado de aprendizaje automático; y un diccionario que separa palabras en positivas y negativas. Se podrían añadir fuentes entrenadas de manera distinta a la de Stanford, para así obtener resultados distintos, y usar también más de un diccionario, para obtener también distintos resultados. La mezcla de las distintas fuentes podría ser una media, se podría dar más peso a los resultados positivos o negativas (tal y como se hace ahora), o se podría usar un sistema de votación.

Otra mejora que se podría realizar es usar un reconocedor de emoticones y emojis, un tema muy trabajado últimamente por la enorme popularidad que estos han conseguido, especialmente en redes sociales. Un método a seguir sería realizar

un preprocesado previo previo de la frase, buscar las apariciones de los emoticonos y emojis, y sustituirlos por tokens especiales, de manera que los tokens tengan asociados un sentimiento específico. Otra posibilidad sería sustituirlos por palabras en el diccionario inglés, para que así pudieran guardarse también como nodos Modificador en la base de datos, es decir, por ejemplo sustituir ":)" por "happy", ":(" por "sad", etc.

Podría decidirse también cambiar el método de realizar el análisis de sentimientos, y en vez de realizarlo orientado a las palabras que modifican a la entidad, se podría enfocar a analizar la frase en su conjunto y luego asignarle el sentimiento resultante a la entidad. Realizar un análisis de sentimientos de la frase completa permite identificar si hay ironía o sarcasmo, y existen varios métodos y herramientas ya construidas para realizarlo. La biblioteca de Stanford permite usarse para un análisis de sentimientos de la frase completa, y funciona bien en cuanto a disyunciones, negaciones y conjunciones se refiere, pero el hecho de que funcione mediante un algoritmo de aprendizaje automático entrenado supone que cataloga frases con ciertas palabras de modo incorrecto, tal y como se ha mencionado al inicio del capítulo 2. Por ejemplo, la frase: "The battery life of my Samsung Galaxy is so long", la clasifica de modo negativo, ya que "long" debe haber aparecido como algo negativo en las frases que han usado para entrenar el algoritmo.

En este proyecto se ha usado un análisis de sentimiento de polaridad, sólo se ha distinguido si los sentimientos son buenos, neutros o malos. Una buena ampliación sería incluir también un análisis de sentimientos que detecten tristeza, alegría y emociones en general. Esta mejora va en línea con el proyecto, pues es útil no sólo conocer la opinión que se tiene sobre una entidad, sino también los sentimientos que provocan en ellos. Un cambio de este tipo requiere hacer un algoritmo muy complejo desde cero, y quizá replantearse el modelo de base de datos a usar o el diseño de la que se usa actualmente.

Al igual que se ha comentado en el apartado anterior, todos estos añadidos aumentan la complejidad del algoritmo ejecutado en Spark Streaming y seguramente su tiempo de ejecución. En el siguiente apartado se propone mejoras relativas a esta parte de la aplicación.

4.3 Aplicación Spark

Como ya hemos visto en el capítulo anterior, el algoritmo de Spark Streaming debe ser capaz de tratar los datos a una velocidad lo suficientemente rápida como para no romper el flujo. Las mejoras mencionadas anteriormente supondrían aumentar todavía más el tiempo de tratamiento de cada elemento del flujo, por lo que a continuación se incluye un cambio en la organización del sistema para hacer

que el análisis de texto y de sentimientos no influya en el algoritmo ejecutado en Spark:

- Aplicación Spark Streaming que recoge tuits en un flujo continuo. Se comprueba si en el tuit existe alguna de las entidades presentes en la jerarquía. Si hay alguna se guarda el texto del tuit y la fecha en la que se realizó en un sistema de almacenamiento persistente, ya sea en ficheros de texto, en una base de datos no relacional que permita escrituras y lecturas rápidas, o en una estructura intermedia como una cola distribuida.
- Aplicación que se ejecute de manera regular, y que se encargue del análisis de la frase y de sentimientos que antes se realizaba dentro de la aplicación Spark. Una vez tratados los datos, se guardan en la base de datos Neo4J de la misma manera que hasta ahora.
- Procedimientos periódicos de compactación y limpieza de la base de datos Neo4J.
- Aplicación gráfica cliente.

Separando el tratamiento del texto de la aplicación Spark, se consigue que este funcione más rápido y recoja más tuits, además de hacerla independiente de la complejidad de los algoritmos de análisis de texto y de análisis de sentimiento. La contrapartida es que se necesita más espacio para guardar los tuits antes de tratarlos. La aplicación que lee los tuits y los procesa se beneficiaría de funcionar del modo parecido a Spark Streaming, y funcionar mediante flujos de manera concurrente, contando además con la ventaja de que ahora no hace falta tratar cada elemento en menos de 2.000 milisegundos.

4.4 Base de datos

Existen pocas mejoras aplicables a la base de datos si no se quiere cambiar su diseño o usar otro modelo de base de datos. Las principales mejoras aplicables tienen que ver con reducir el espacio que ocupan los nodos y propiedades, reduciendo el espacio que ocupan sus propiedades.

Las fechas en las que se guardan las menciones de los nodos Entidad y Elemento y las que aparecen como propiedad en las relaciones con los nodos Modificador es un entero largo que guarda el tiempo en horas desde las 00:00 del 1 de enero de 1970. Dado que el programa recoge los tuits en un flujo, y no accede a los tuits pasados, sería posible guardar esta información usando un entero más pequeño, de manera que solo guarde el número de horas pasado desde el primer inicio de la aplicación, para lo cual habría que guardar la fecha del primer inicio de la aplicación. Este cambio no es tan inmediato como parece, ya que la mayoría de las bibliotecas de traducción de fechas funcionan con el formato anterior pero en milisegundos, y tendrían que hacerse conversiones cada vez que se quisiera usar una de estas bibliotecas, aumentando el tiempo de cómputo.

La puntuación de los nodos de la base de datos también puede optimizarse, ahora mismo se usa un tipo *float* tanto en la ejecución del programa como en la base de datos. No tiene mucho sentido usar tantos decimales para guardar el sentimiento, y usar tan solo tres decimales puede ser suficiente. Como el valor no decimal está comprendido entre 0 y 4, podría multiplicarse la puntuación por 1.000 y guardarla como un entero en la base de datos, no solo salvando un poco de espacio sino también agilizando las operaciones realizadas con las puntuaciones.

Existe una alternativa a perder tanta información al aplicar los métodos de compactación en la base de datos, aunque implica cambios en la organización de la base de datos. Se pueden plantear recorrer los nodos Entidad y Elemento de la base de datos Neo4J antes de hacer la compactación, y de cada uno de ellos recoger varios nodos Modificador con sus respectivas fechas de aparición que se consideren importantes (los cinco más mencionados, los cinco con peor/mejor puntuación…). Tras recoger estos datos se guardan en una base de datos, que no necesariamente necesita ser Neo4J u orientada a grafos, de hecho en este caso como tampoco necesitamos alta velocidad ni gran espacio, podría usarse una base de datos relacional. Con esto se consigue mantener ciertos datos relativos a las fechas consiguiendo reducir el tamaño de la base de datos orientada a grafos, pero se aumenta el tamaño total de los datos persistentes.

4.5 Interfaz gráfica

Hay numerosas partes de la interfaz gráfica que admiten mejoras, tanto partes ya incluidas como algunas nuevas que se pueden desarrollar. Desde el punto de vista de la eficiencia de la aplicación se pueden realizar varias mejoras.

Durante el transcurso de la aplicación, tanto el dibujo del grafo como el dibujo de las gráficas requieren acceso a la base de datos y puede llegar a ser lento. Como ya se ha mencionado en el apartado de resultado, al iniciar la aplicación está llega a tardar 2 minutos. Esto no se restringe al inicio de la aplicación, sino que también ocurre cuando se trata de cargar una subjerarquía con muchos nodos Modificador haciendo doble clic en un nodo del árbol. Estas operaciones bloquean la interfaz gráfica, por lo que en estos casos sería posible usar hilos que se encarguen de acceder a estos datos y dibujar el grafo y las gráficas, permitiendo que el resto de la interfaz responda a interacciones mientras ocurre, y que por ejemplo el usuario pueda hacer clic en el árbol y ver la información del nodo en la parte inferior de la pantalla. Nótese que la creación de hilos no es trivial, sino que hay controlar el caso en el que el usuario cambie de pestaña antes de que el hilo acabe de ejecutar, o si hace doble clic en un elemento del árbol antes de que cargue el anterior y se produzca una condición de carrera, etc.

La solución anterior es válida para determinados escenarios, pero no sirve para arreglar la tardanza del inicio de la aplicación, ya que hay que cargar los datos iniciales de la base de datos y la aplicación no puede empezar sin estos. Para paliar este problema se podría añadir algo de *feedback* al usuario, desde algo tan simple como una pantalla que diga “Cargando...”, a algo más complejo como una barra de progreso, que aunque no sea muy precisa por lo menos pueda indicar el porcentaje de nodos Entidad y Elemento que faltan por tratar.

Se pueden completar los elementos que faltan en la interfaz, como las secciones de “Configuración” y “Ayuda” del menú desplegable. En la sección de “Configuración” se podrían añadir opciones para guardar el usuario y la contraseña de la conexión a la base de datos y que esta se autocomplete, añadir una conexión a una base de datos por defecto, y otras opciones relativas a la conexión con las bases de datos. En la sección de “Ayuda” se podría incluir un enlace a un manual de usuario. También se podría añadir a “Archivo” una opción para establecer conexión con las 5 últimas bases de datos conectadas. Es posible incluir más opciones en el menú, como una opción de “Formato” que permita cambiar de fuente el texto, o el color de los nodos del árbol según su tipo, etc.

Añadir una pestaña extra de “Resumen” que se pueda elegir en la ventana principal es también posible, que comparta el árbol de la jerarquía de la parte izquierda y los detalles de los nodos de la parte inferior, y que el centro de la pantalla lo ocupe un cuadro resumen con las palabras más mencionadas de una entidad, o las que tienen puntuación más baja/alta. Esto es posible conseguirlo en la pestaña de búsqueda buscando modificadores y ordenando los resultados, pero sería preferible tener un atajo para esto.

Capítulo 5 - Conclusiones

Como conclusión de esta memoria, este capítulo contiene una enumeración de los distintos objetivos fijados en la introducción del proyecto y un estudio de cómo se han cumplido (en los dos capítulos anteriores hay información más detallada de los resultados y de cómo mejorarlos, respectivamente). También se incluye un apartado en el que se detalla la experiencia adquirida y el grado de satisfacción personal con el desarrollo del proyecto.

5.1 Objetivos

El desarrollo del proyecto ha sido complejo, con muchas partes distintas que se han tenido que integrar y funcionar juntas. En el capítulo 3 ya se han presentado los resultados obtenidos, aquí se realiza una reflexión sobre cómo se han cumplido los objetivos fijados al inicio del proyecto.

Desde el punto de vista del tratamiento de lenguaje se ha realizado un trabajo que ha requerido de análisis sintáctico, morfológico y de reconocimiento de entidades. El análisis es mejorable tal y como se ha mencionado en el capítulo anterior, pero aún así la versión actual es capaz de encontrar las palabras relacionadas de una frase con los elementos de la jerarquía. Las palabras encontradas ofrecen información útil acerca de lo que se está diciendo del elemento en concreto de la jerarquía, información que es usable para encontrar las cualidades que más interesan a la gente, ya sea de manera positiva o negativa.

El análisis de sentimiento de las palabras encontradas también ha demostrado ser útil, pues sirve para identificar cómo se está hablando de los elementos de la jerarquía, y además para averiguar si lo que se está diciendo del elemento es positivo o negativo, sirviendo así para identificar las debilidades aunque la percepción general sea buena. En el proyecto se han representado los sentimientos mediante puntuaciones, y se modifica la puntuación de los elementos de la jerarquía según la puntuación de las palabras que los modifican, permitiendo así ver la puntuación media de estos nodos de un modo rápido.

El uso del entorno de cómputo Apache Spark, y de su API Streaming, ha sido esencial para el desarrollo de este proyecto. El tratamiento en flujo de manera continua de los tuits permite ahorrar espacio teniendo que guardar tuits, y el funcionamiento concurrente que sigue es intuitivo y sencillo de usar. Spark consigue ahorrar el enorme esfuerzo que supondría tener que construir un entorno de ejecución que permitiera computación distribuida, además de que su API Streaming ofrece una conexión con Twitter para conseguir tuits de manera continua. No se ha conseguido crear una aplicación Spark que se ejecute sin perder conexión, pero como ya se ha explicado, esto no ha supuesto un deterioro importante en la

eficiencia de la aplicación y ha supuesto ahorrar espacio, a pesar de esto se ha propuesto también una mejora en el capítulo 4.

Se ha conseguido encontrar un modelo de base de datos no relacional totalmente adecuado para los datos con los que se trabajan, el modelo de grafos. Además, Neo4J ofrece una base de datos de este modelo que permite escrituras rápidas y que se realicen operaciones sobre ella de forma concurrente, siendo perfecto para usarse en una ejecución de Spark Streaming. Los procedimientos creados para el mantenimiento de la base de datos consiguen que la base de datos no crezca de forma incontrolada, y además sirven para establecer un a base sobre la que el administrador del sistema puede experimentar y construir, creando más procedimientos de compactación por fechas usando los ya hechos o ideando algoritmos que los apliquen con distinta frecuencia.

La interfaz gráfica creada permite visualizar un resumen de los nodos más relevantes de la base de datos, ofreciendo a un usuario no iniciado la posibilidad de visualizar los resultados conseguidos por la aplicación sin necesidad de usar consultas de bases de datos, La interfaz permite comparar distintos nodos del árbol, y ver la evolución que han seguido a lo largo del tiempo. También permite realizar búsquedas complejas sobre la base de datos y así poder acceder de una manera cómoda y conocida a la información extraída. Existen también aspectos a mejorar, como el *feedback* que se da al usuario, la estricta dependencia del tiempo de respuesta de la aplicación con el tamaño de la base de datos, las nulas opciones de configurar la interfaz, y lo restrictiva que es a la hora de comparar varios nodos de la jerarquía.

Finalmente, queda concluir que los aspectos principales de los objetivos se han cumplido, y se ha creado un sistema que aglutina un conjunto de distintas técnicas, bibliotecas y tecnologías para resolver un problema Big Data y extraer información interesante para organizaciones de una de las fuentes actuales de información más prolíficas, Twitter.

5.2 Conocimientos adquiridos

Aunque no haya sido mencionado en la introducción ni en el capítulo anterior, uno de los objetivos principales por el que se desarrolló este proyecto fue aprender. Aprender sobre Big Data, análisis de sentimientos, bases de datos no relacionales, redes sociales, etc. todos ellos conceptos conocidos y estudiados pero de manera aislada.

Se buscaba familiarizarse con el proceso de buscar las bibliotecas a usar, elegir un buen entorno de computación distribuida, identificar el modelo de base de datos ideal para almacenar los datos. Y cómo no, se buscaba desarrollar un sistema que

incorpore estos conceptos y pueda resolver un problema real, en el que se puedan ver los resultados obtenidos de manera clara.

Al inicio del desarrollo se tenía cierto conocimiento y un bajo nivel de experiencia con entornos de computación distribuida, pero nunca se había usado Spark. De igual manera, se tenía experiencia con bases de datos no relacionales clave valor, pero no basadas en grafos, y nunca se había usado Neo4J. Durante la carrera académica también se habían adquirido conocimientos sobre el tratamiento de lenguaje natural, pero tampoco se había realizado nunca análisis de sentimientos. El lenguaje Scala, con el que se ha desarrollado todo el proyecto, tampoco se conocía, aunque sí se tenía experiencia con lenguajes imperativos, orientados a objetos y declarativos.

Al final del proyecto se consiguió adquirir experiencia con todo lo anterior, construyendo un sistema funcional y completo, cumpliendo el principal objetivo de este proyecto, el aprendizaje.

Chapter 5 - Conclusions

As a conclusion to this report, this chapter contains an enumeration of the different objectives set in the introduction of the project and a study of how they have been fulfilled (in the two previous chapters there is more detailed information on the results and how to improve them, respectively). This section also includes a description of the experience gained and the degree of personal satisfaction with the development of the project.

5.1 Objectives

Project's development has been complex, with many different parts that have had to be integrated and put to work together. Chapter 3 already contains the obtained results, this chapter focuses on how the objectives set at the beginning of the project were met.

The natural language processing analysis has required syntactic analysis, morphological analysis and entity recognition. This part can be improved as has been mentioned in the previous chapter, but still the current version is able to find the related words in a sentence with the elements of the hierarchy. Words found offer useful information about what is being said about the particular element of the hierarchy, making easy to find what makes the elements of the hierarchy popular, either in a positive or a negative way.

The sentiment analysis of the words found in the sentences has also proved useful, as it serves to identify how the elements of the hierarchy are being discussed, and also to find out if what is being said about the element is positive or negative, thus serving to identify the product weaknesses even if the general perception of the product is good. In the project the sentiments found have been represented by a score system, which is modified according to the punctuation of the words related to them, thus allowing the user to see the average score of these nodes in a fast and simple way.

The use of the Apache Spark computing environment and its Streaming API has been essential to the development of this project. The treatment of the continuous tweet stream allows saving space by not needing to have to save the tweets, and the concurrent way the API works is intuitive and easy to use. Spark manages to save the enormous effort that would be needed to build a distributed computing environment. In addition, the Streaming API offers a connection with Twitter to obtain a stream of tweets. It has not been possible to create a Spark application that runs without never losing the established connection, but like what has already been explained, this has not meant a significant deterioration of the efficiency of the

application and has implied saving some space, despite this some improvements have also been proposed in Chapter 4.

It has been possible to find a non-relational database model that is fully adequate for the data that is used, the graph database. In addition, Neo4J offers a graph oriented database that allows quick writing operations and many different concurrent operations, just making it perfect to use it in a Spark Streaming execution. The procedures created to maintain the database make it so that it does not grow uncontrollably, and it also serves to establish a base on which the system administrator can experiment and expand on it, let it be by creating more compaction procedures by dates or by applying the already made algorithms with different frequency.

The graphical user interface allows viewing a summary of the most relevant nodes in the database, giving an uninitialized user the possibility to visualize the results obtained by the application without the need to use database queries. The interface also allows to see the different nodes of the tree, and see the evolution they have followed over time. It also makes it possible to perform complex searches on the database so that the extracted information can be accessed in a convenient and familiar way. There are also aspects that could be improved, such as the feedback given to the user, the strict dependence of the response time of the application with the size of the database, the non existent ways of configuring the interface, and how restrictive it is about comparing several nodes in the hierarchy.

Finally, it can be concluded that the main aspects of the objectives have been fulfilled, and a system has been created that brings together a set of different techniques, libraries and technologies to solve a Big Data problem and extract interesting information for organizations to use from one of the most prolific current source of information, Twitter.

5.2 Knowledge acquired

Although it has not been mentioned in the introduction or in the previous chapter, one of the main objectives of the project was to learn. To learn about Big Data, sentiment analysis, non-relational databases, social networks, etc. as all of these concepts were already known and studied but in an isolated way.

The objective was to familiarize with the process of looking for the libraries that should be used, choosing a good distributed computing environment, identifying the ideal database model to store the data. And of course, it was sought to develop a

system that incorporated these concepts and could solve a real problem, in which you could see the results obtained after applying the prior concepts clearly.

A bit of knowledge and some experience with distributed computing environments was had at the beginning of the project, but Apache Spark had never been used. Similarly, I had experience with non-relational key value databases, but not with the graph-based model, and Neo4J had also never been used. Knowledge of natural language processing had also been acquired during the academic career, but sentiment analysis had never been applied. The Scala language, with which the whole project was developed, was not known either, although there was experience with imperative, object-oriented and declarative languages.

A lot of experience with the concepts mentioned above was gained by the time the project was finished, making it possible to build a complete and functional system, which shows that the main objective of the project was fulfilled, learning.

Bibliografía

[1] Spark (2014) “Apache Spark: Lightning-fase cluster computing”. Obtenido de: http://spark.apache.org/

[2] Scala (2011) “Scala: What is Scala?”. Obtenido de: http://www.scala-lang.org/what-is-scala.HTML

[3] W. Medhat, A. Hassan, H. Korashy “Sentiment analysis algorithms and applications: A survey” Ain Shams Engineering Journal 5:1093--1113 2014

[4] B. Lu, L. Zhang “A survey of opinion mining and sentiment analysis”, ed. C. C. Aggarwal and C. X. Zhai. Boston, MA: Springer US, 2012, pp. 415—463. ISBN: 978-1-4614-3223-4. DOI=10.1007/978-1-4614-3223-4_13

[5] W. Hongwei, W. Wei, Y. Pei “Prodweakfinder: An information extraction system for detecting product weaknesses in online reviews based on sentiment analysis” Proceedings of The Pacific Asia Conference on Information Systems (PACIS), 2014. ISBN 978-988-8353-22-4. Obtenido de: http://aisel.aisnet.org/pacis2014/101

[6] M. Z. Asghar, A. Khan, S. Ahmad, F. M. Kundi (2014) “A Feature Extraction Process for Sentiment Analysis of Opinions on Services” Proceedings of International Workshop on Web and Text Intelligence (WTI), São Bernardo do Campo, Brazil, 2010, pp. 404--413. Obtenido de: http://www.labic.icmc.usp.br/wti2010/IIIWTI_camera_ready/74769.pdf

[7] P. D. Turney “Thumbs up or thumbs down?: Semantic orientation applied to unsupervised classification of reviews” Proceedings of the 40th Annual Meeting on Association for Computational Linguistics in Morristown, NJ, USA, Association for Computational Linguistics, 2002, pp. 417-424

[8] Sproutsocial (2010) “Sproutsocial. Measure performance with social media analytics”. Obtenido de: http://sproutsocial.com/features/social-media-analytics

[9] Falcon.io (2010) “Falcon.IO: Keep up to date with Social Media monitoring” Obtenido de: https://www.falcon.io/products/social-media-monitoring/

[10] Agorapulse (2010) “Agorapulse: Identify oportunities”. Obtenido de: https://www.agorapulse.com/features/social-media-monitoring

[11] Oracle Social Cloud (2017) “Oracle Customer Experience Social Cloud”. Obtenido de: https://www.oracle.com/es/applications/customer-experience/social/index.HTML

[12] Twazzup (2010) “Twazzu: get real-time insight from twitter”. Obtenido de: http://new.twazzup.com/

[13] Lithium (2001) “Lithium: deliver better customer experiences through digital channels”. Obtenido de: https://www.lithium.com/

[14] Adobe social (2012) “”Adobe Social: get results with your social marketing”. Obtenido de: http://www.adobe.com/es/marketing-cloud/social-media-marketing.HTML

[15] Twitter4J (2007) “Twitter4J”. Obtenido de: http://twitter4j.org/en/index.HTML

[16] Stanford NLP Group (2010) “The Stanford Natural Language Processing Group”. Obtenido de: https://nlp.stanford.edu/

[17] Stanford NLP Core. (2014) “Stanford CoreNLP - Core Natural Language Software”. Obtenido de: https://stanfordnlp.github.io/CoreNLP/

[18] Neo4J (2007) “Neo4J”. Obtenido de: https://neo4j.com/product/

[19] Datanami (2016) “Neo4J pushes graph DB limits past a quadrillion nodes”. Obtenido de: https://www.datanami.com/2016/04/26/neo4j-pushes-graph-db-limits-past-quadrillion-nodes/

[20] Stackexchange (2011) “Amount of data per node in Neo4J”. Obtenido de: https://dba.stackexchange.com/questions/4996/amount-of-data-per-node-in-neo4j

[21] Anormcypher (2015) “Anormcypher”. Obtenido de: http://www.anormcypher.org/

[22] Anorm (2015) “Anorm, simple SQL data access”. Obtenido de: https://www.playframework.com/documentation/2.0.4/ScalaAnorm

[23] JavaFX (2009) “JavaFX Overview”. Obtenido de: http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#JFXST784

[24] WebKit (1998) “WebKit: Open Source Web Browser Engine“. Obtenido de: https://webkit.org/

[25] Alchemy.js (2014) “Alchemy.js: a graph visualization application for the web”. Obtenido de: http://graphalchemist.github.io/Alchemy/#/

[26] B. Liu, M. Hu, J. Cheng. "Opinion Observer: Analyzing and Comparing Opinions on the Web." WWW '05: Proceedings of the 14th international

conference on World Wide Web, in New York, NY, USA, ACM, 2005, pp. 342-351.

[27] Internet Live Stats (2017) “Twitter Usage Statistics”. Obtenido de: http://www.internetlivestats.com/twitter-statistics/

[28] English Oxford living dictionaries (1989) “How many words are there on the English language?”. Obtenido de: https://en.oxforddictionaries.com/explore/how-many-words-are-there-in-the-english-language

[29] J. Zhang, S. S. Bhowmick, H. H. Nguyen, B. Choi, F. Zhu. “DaVinci: Data-driven Visual Interface Construction for Subgraph Search in Graph Databases” IEEE 31st International Conference on Data Engineering (ICDE), in COEX, Samsung-dong, Kangnam-gu, Seoul, Korea (South), 2015, pp. 1500 -1503

[30] Stackexchange (2010) “Intuitive interface for Composing Boolean Logic?”. Obtenido de: https://ux.stackexchange.com/questions/1737/intuitive-interface-for-composing-boolean-logic?noredirect=1&lq=1

Anexo I: Manual de Instalación

I.1 Aplicación ejecutada en Apache Spark

Para la ejecución de este programa, es necesario tener instalado:

- Spark 2.00 o superior con Hadoop 2.7.
- Sbt 0.13.13 (si se quiere reconstruir el proyecto)
- Java 8 o superior.
- Scala 2.11 o superior.
- Neo4j 3.1.1 o superior.

El repositorio que contiene la aplicación a ejecutar en el entorno Spark se encuentra en el repositorio:
https://github.com/josejamp/TFM/tree/master/Entregables/appSpark/TwitterCollect
Esto no contiene ningún ejecutable, sino que hay que crear el proyecto y ejecutarlo usando los comandos de Spark, como se verá en el Anexo II.

I.2 Aplicación de limpieza y compactación

Para la ejecución de este programa, es necesario tener instalado:

- Java 8 o superior.
- Scala 2.11 o superior.
- Neo4j 3.1.1 o superior.

El ejecutable para la ejecución tiene como nombre mantenimiento.jar, y se puede encontrar en
https://github.com/josejamp/TFM/tree/master/Entregables/mantenimiento .

I.3 Aplicación gráfica

Para la ejecución de este programa, es necesario tener instalado:

- Java 8 o superior.
- Scala 2.11 o superior.
- Neo4j 3.1.1 o superior.

El ejecutable tiene como nombre interfazGrafica.jar, y se puede encontrar en: https://github.com/josejamp/TFM/tree/master/Entregables/interfazGrafica . En la misma carpeta se incluye una carpeta llamada webResources. Esta carpeta debe estar presente en el mismo directorio en el que esté el ejecutable.

Anexo II: Manual de Usuario

II.1 Aplicación ejecutada en Apache Spark

Es necesario construir el proyecto y compilar la aplicación mediante sbt, para esto hay que descargarse el proyecto e ir a la carpeta "appSpark/TwitterCollect". Una vez en esta carpeta, se ha de ejecutar el comando "sbt assembly". Se recomienda aumentar la memoria virtual de Java a por lo menos 6 GB.

La aplicación necesita de una conexión a una base de datos Neo4J activa para funcionar correctamente. Para ejecutar la aplicación se debe usar el comando spark-submit de Spark con los siguientes argumentos:

--class	Nombre de la clase a ejecutar, en este caso es la clase Collect: "tweet_analizer.Collect"
--master	Determinar el número de procesadores que ejecutan en paralelo.
*args	Argumentos de la aplicación, se introducen después del argumento de "--master" y van separados por espacios: <path_to_jar> <database_url> <database_user> <database_password> <minutes_of_execution>
--consumerKey	Clave de consumidor de la aplicación de Twitter.
--consumerSecret	Secreto de consumidor de la aplicación de Twitter
--accessToken	Token de consumidor de la aplicación de Twitter
--accessTokenSecret	Secreto del token de consumidor de la aplicación de Twitter

Todos los argumentos son necesarios, el comando final sería:

$ bin\spark-submit --class tweet_analizer.Collect --master local[*] <path_to_jar> <database_url> <database_user> <database_password> <minutes_of_execution> --consumerKey x...xxx --consumerSecret y...yyy --accessToken w..ww --accessTokenSecret z...zzz

II.2 Aplicación de limpieza y compactación

Para ejecutar la aplicación se debe ejecutar el jar "mantenimiento.jar". El ejecutable necesita parámetros, así que debe ser lanzado desde la línea de comandos, y no haciendo clic desde el explorador de archivos.

La aplicación realizará una compactación o una limpieza especificada con una frecuencia en segundos un número determinado de veces. El programa necesita de distintos argumentos dependiendo del tipo de compactación o limpieza que se quiera usar. El formato del comando es el siguiente:
$ java -jar mantenimiento.jar <database_url> <database_user> <database_password> [limpieza | compactacion] [total | media | proporcion | dia>] <delayInicial> <intervalo> <repeticiones> <argumento_especifico>

Entre los argumentos se debe elegir entre si se quiere limpieza o compactación, dependiendo de lo que se elija se deben especificar distintos argumentos:

- Limpieza:
    - total: se eliminan las palabras que hayan aparecido repetidas menos que <argumento_especifico> veces.
    - media: se limpian las palabras de cada entidad que hayan aparecido repetidas un número de veces menor a la media de repeticiones de la entidad. Esta opción no necesita ni debe incluir <argumento_especifico>.
    - proporcion: se limpia un porcentaje de palabras menos repetidas, especificado en el argumento <argumento_especifico>.
- Compactación:
    - dia: se compactan las relaciones por días para una fecha menor que la dada en <argumento_especifico>. La fecha debe estar en formato Epoch en horas, es decir, en el número de horas que han pasado desde las 00:00 del 1 de Enero de 1970.

II.3 Aplicación gráfica

Para ejecutar la aplicación gráfica basta con ejecutar el archivo “interfazGrafica.jar”, ya sea desde el explorador de archivos o usando la línea de comandos. Una vez ejecutado, aparece la pantalla vista en la Figura Anexo.1.

En esta pantalla se ha de indicar la dirección del servidor Neo4J con la base de datos a la que queremos conectarnos, y además indicar el usuario y la contraseña para garantizar que no cualquiera pueda conectarse. Una vez incluidos se pulsa el botón “Aceptar”.
:

Figura Anexo.1: Inicio de la aplicación

Tras pulsar el botón se carga la ventana principal de la aplicación. El tiempo de carga puede ser largo, y dependiendo del tamaño de la base de datos el orden de este tiempo puede llegar a minutos. Una vez haya aparecido la nueva ventana y se haya cargado el grafo, podemos interactuar con ella (Figura Anexo.2).

En esta ventana se pueden observar los siguientes elementos:

1. Menú desplegable. Las opciones de “Configuración” y “Ayuda” no son funcionales. El menú de “Archivo” despliega opciones para salir del programa y para cambiar la base de datos a la que se está conectado.
2. Pestañas: las pestañas indican el contenido mostrado en la ventana principal: la pantalla de visualización de la jerarquía y sus palabras en forma de grafo, la pantalla de visualización de estadísticas, y la pantalla de búsqueda avanzada.
3. Jerarquía: representa el árbol desplegable de la jerarquía. Si se hace un clic sobre uno de los elementos se muestra su información en (4). Si se hace doble clic se carga el grafo asociado a la subjerarquía en (5).
4. Información del nodo de la jerarquía seleccionado.
5. Grafo de la subjerarquía sobre la que se ha hecho doble clic. Se pueden arrastrar nodos con el ratón para cambiar su posición, o pinchar en el espacio negro donde no hay nodos para desplazarse por el visor.

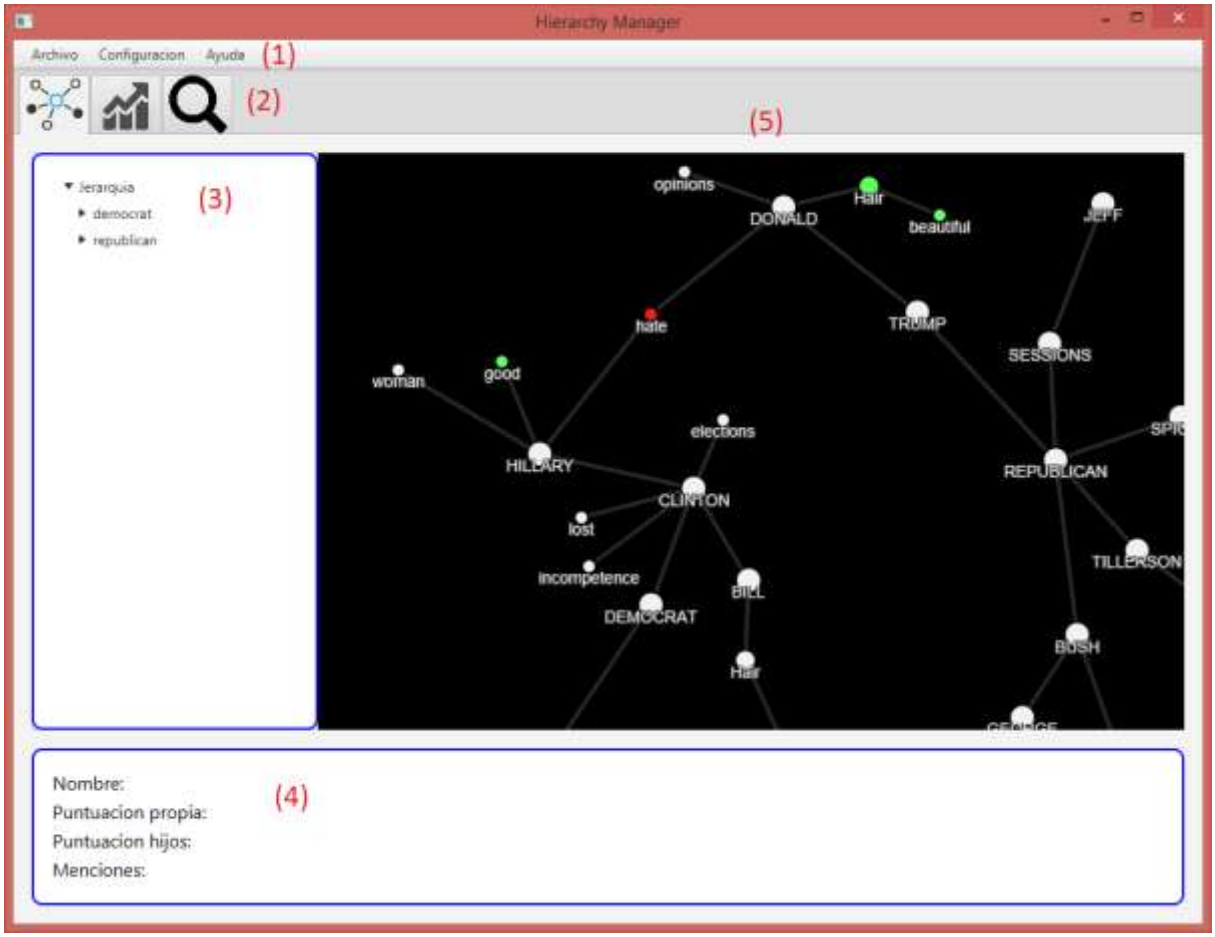

Figura Anexo.2: Ventana principal, vista del grafo

Al pulsar sobre la segunda pestaña, el contenido de la ventana principal cambia al mostrado en la figura Anexo.3. En esta pestaña se pueden ver los siguientes elementos:

1. Jerarquía: al igual que antes, un clic muestra los datos del nodo en la parte de abajo. Doble clic carga las gráficas asociadas al nodo en (2).
2. Gráfica: la parte central la ocupan la gráfica mostrada. La gráfica es de barras si en (3) se elige en “X axis: Subjerarquías” y se muestra la información de los nodos hijos del nodo seleccionado. Si se selecciona en “X axis: Tiempo” se muestra un gráfica de línea que muestra la evolución del atributo a lo largo del tiempo. El atributo se elige en (3) en “Y axis”.
3. Hay dos pestañas que controlan la información que se muestra en el eje x y la que se muestra en el eje y:
    - Eje x:
        i. Subjerarquías: muestra la información del atributo de los hijos del nodo, para así poder compararlos.
        ii. Tiempo: muestra la evolución del atributo a lo largo del tiempo.
    - Eje y:
        i. Resumen: muestra una media de la puntuación propia del nodo y la de sus hijos.

ii. Puntuación propia: la puntuación (sentimientos) de las palabras encontradas para el nodo.
iii. Puntuación hijos: la puntuación de los descendientes del nodo.
iv. Menciones: el número de veces que ha sido mencionado el nodo.

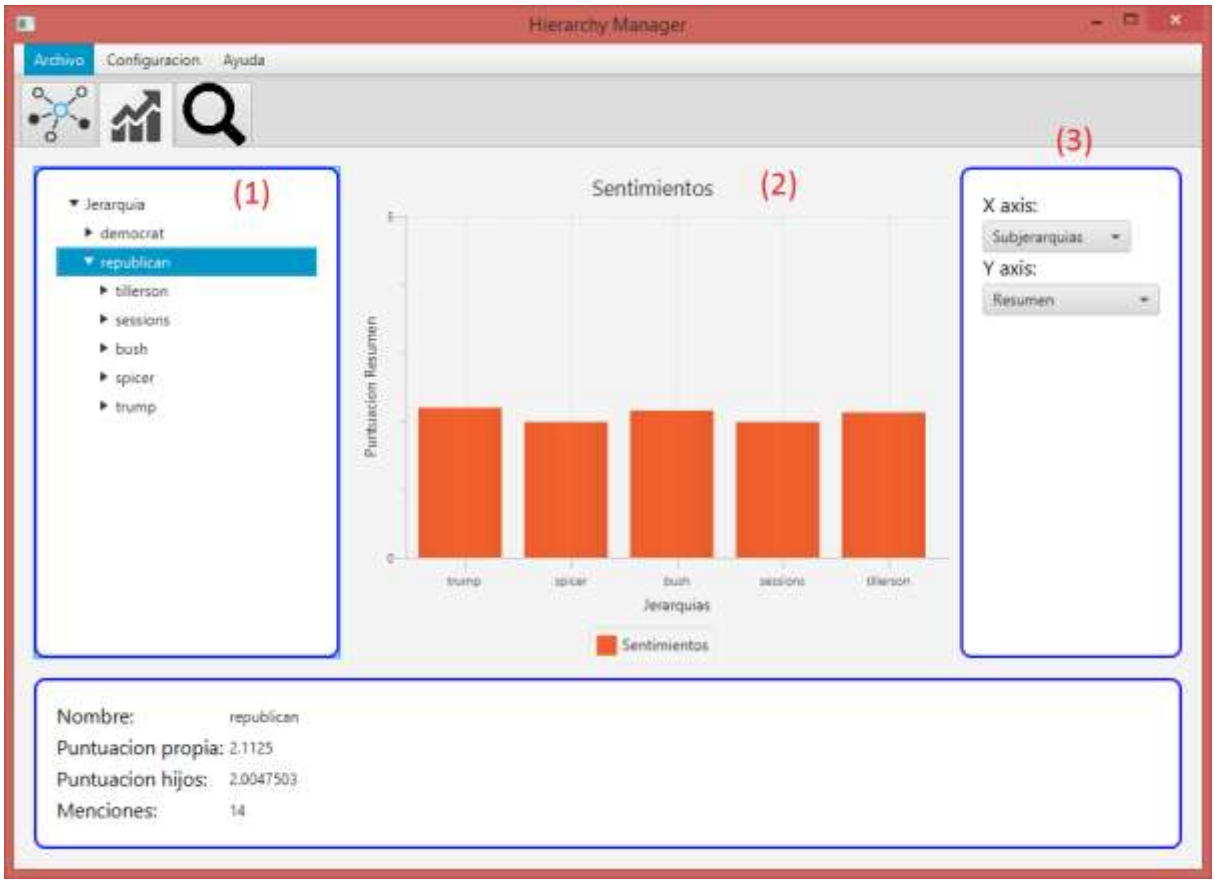

Figura Anexo.3: Visor estadísticas

Al pulsar la pestaña con el icono de la lupa, se abre la ventana de búsqueda avanzada (Figura Anexo.4). Esta pestaña permite realizar búsquedas complejas de los elementos de la base de datos, y esto se realiza mediante los siguientes elementos:

1. Botón de “Buscar”: busca los elementos con las condiciones incluidas en la ventana. Al pulsarlo se abre una ventana nueva con los resultados de la búsqueda.
2. Botones para añadir/quitar condiciones: estos botones permiten añadir o eliminar condiciones de la búsqueda.
3. Desplegable “algunas/todas”: este desplegable indica si los resultados deben cumplir todas o alguna de las condiciones establecidas.
4. Permitir fechas: este cuadro permite marcar si la condición establecida se debe cumplit entre ciertas fechas o no.
5. Selector de elemento a buscar: este desplegable permite seleccionar el elemento que se quiere buscar, se puede elegir entre “modificador”,

"elemento" o el resto de relaciones de la base de datos. Siguiendo el ejemplo de los políticos este resto de relaciones incluyen "partido", "familia" y "nombre".

6. Pestaña de pertenencia: si se selecciona la opción "pertenece" en esta pestaña, se abre un cuadro con una subconsulta. Se buscan los elementos que sean descendientes de los encontrados en la subconsulta.
7. Pestaña de condición de elemento: en vez de seleccionar "pertenece" se puede seleccionar uno de los atributos de los nodos de la base de datos, como su nombre, el número de menciones, su puntuación, etc. y establecer una condición que el atributo debe cumplir.
8. Condición de fecha: en estos cuadros de texto se establece entre qué fechas se debe cumplir la condición.

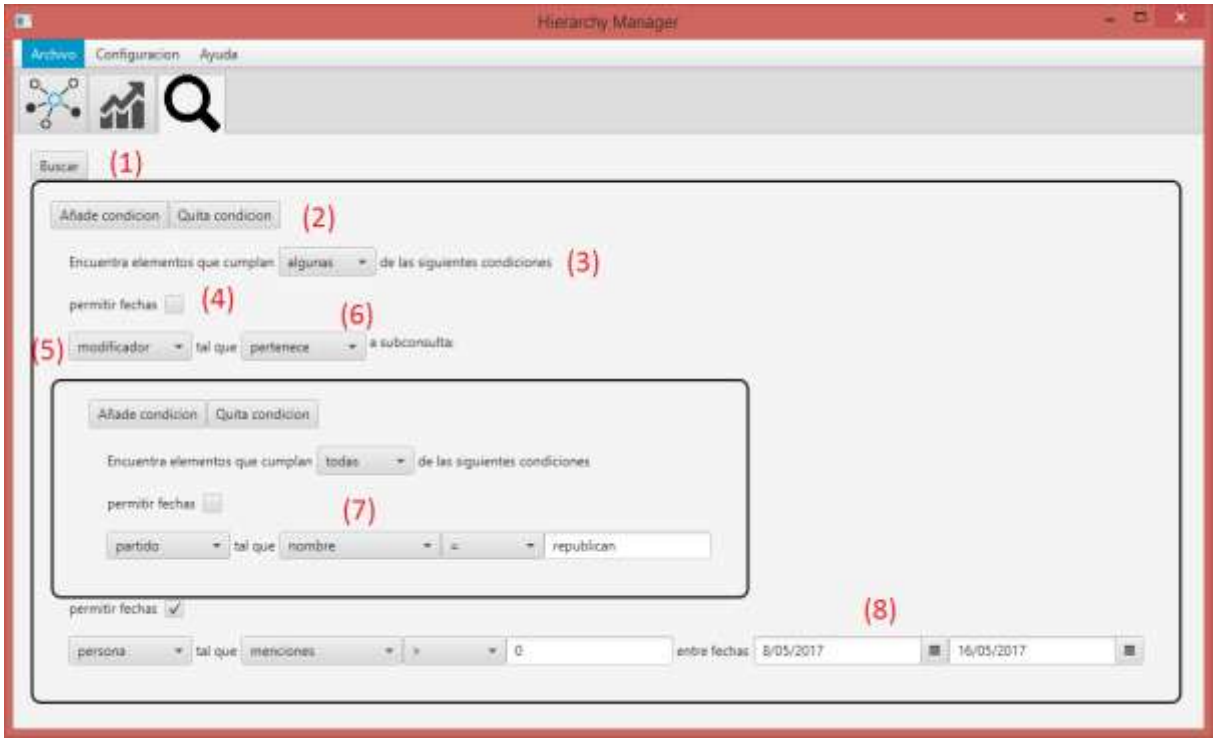

Figura Anexo.4: Búsqueda avanzada

Tras pulsar el botón de "Buscar", se abre una pestaña con los resultados de la búsqueda (Figura Anexo.5). En esta ventana se tiene una tabla o dos tablas que contienen los resultados. Se puede pulsar sobre los nombres de las columnas para establecer cómo aparecen los resultados ordenados.

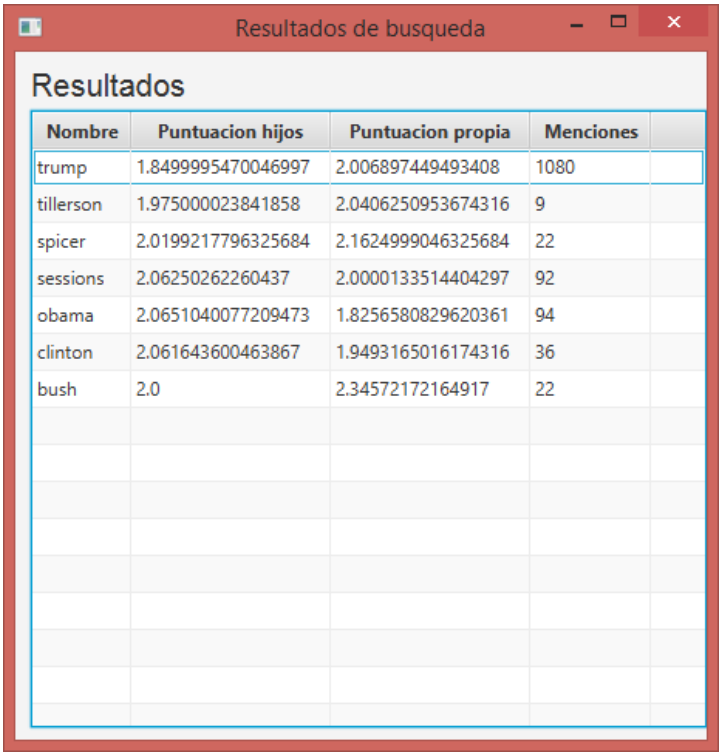

Figura Anexo.4: Tabla de resultados de la búsqueda avanzada