CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Trabajo Fin de Grado

Servicio de creación, gestión y utilización de modelos predictivos basados en grafos de conocimiento

Autor: Víctor Fernández Rico

Director: Óscar Corcho García

MADRID, JUNIO DE 2017

Índice general

Índice de figuras

Índice de listados

Agradecimientos

A mi tutor, Óscar Corcho, y a Daniel Vila,
por darme la oportunidad de aprender
y tener la paciencia de enseñar.

Y a mis padres y mi hermano,
por apoyarme en todo momento.

Resumen

Con el auge de las bases de conocimiento colaborativas como Wikidata o DBPedia, se hace necesario crear modelos que sean capaces de explotar la información que almacenan de forma rápida y eficiente. Pese al gran tamaño y complejidad que tienen, es posible representar las entidades y relaciones presentes en estas bases de conocimiento con modelos de factores latentes. Estos modelos de factores latentes permiten codificar de una manera compacta las interacciones y propiedades del grafo de conocimiento y por tanto reducen en varios órdenes de magnitud el tamaño de esta estructura de datos en memoria.

Dado que estas bases de conocimiento están pensadas para ser comprendidas tanto por seres humanos como por sistemas automatizados, es frecuente que estén expresadas utilizando estándares que siguen la filosofía de la web semántica.

En este trabajo se expondrá la solución creada, *KGE-Server*, que permite aplicar las técnicas de aprendizaje automático basadas en modelos de factores latentes sobre grafos de conocimiento creados automáticamente a partir de bases de conocimiento colaborativas, utilizando las herramientas que proporcionan los estándares de la web semántica. Esto permite generar modelos predictivos para encontrar similitud entre entidades o predecir la relación entre dos entidades.

Dicha solución está orientada a ser fácil de utilizar y de desplegar en un entorno real. Provee una API REST, que habilita a cualquier dispositivo capaz de realizar una petición HTTP utilizar dichos modelos predictivos, y ha sido construido de forma modular utilizando contenedores Docker, facilitando la labor de puesta en funcionamiento.

Palabras clave: Grafos de conocimiento, *embeddings*, Web Semántica, modelos de factores latentes, aprendizaje automático, servicio web, API REST.

Abstract

In the last years, with the groing collaborative knowledge datasets like Wikidata or DBPedia, it is necessary to create artificial intelligence models in order to take advantage of the information stored in a fast and efficient way. Although this knowledge datasets are complex and really big, it is possible to represent the entities and relationships stored with latent factor models. Those models allows us to code in a compact way the interactions and properties present on the knowledge graph, and therefore the size this graph occupies in memory is reduced on several magnitude orders.

Due to those knowledge datasets are created to be used by both human and autonomous systems, they are usually represented using standards from the Semantic Web philosophy.

In this work I will show the created solution, called *KGE-Server*, which allows to apply all machine learning techiques based on latent factor models over knowledge graphs automatically created from collaborative knowledge datasets using the tools that Semantic Web standards provides. On the end, we can create predictive models in order to find entity similarity or predict links between entities.

This solution is also oriented to be easy to use and easy to deploy on a real environment. It provides a REST API which allows any device able to build HTTP requests to use those predictive models, and it has been constructed in a modular way using Docker containers, easing the deployment labour.

Keywords: Knowledge Graph, Embeddings, Semantic Web, latent factor models, machine learning, web service, REST API.

1 Introducción

En los últimos años la información presente en Internet ha crecido exponencialmente, y ha estado cada vez más presente en nuestras vidas sobre todo en el ámbito de la comunicación. Pero no ha sido sólo la información relacionada con las redes sociales lo que ha crecido, también lo ha hecho la información con la que trabajamos diariamente, algunas provenientes de proyectos colaborativos y abiertos como Wikipedia. A partir de éste proyecto surgen otros como DBPedia [1] o Wikidata [2] que buscan crear repositorios estructurados y abiertos de información. Éstos pueden ser representados como grafos de conocimiento utilizando estándares propios de la Web Semántica como RDF para representar las relaciones o como SPARQL para realizar consultas a estas enormes bases de conocimiento.

Estas bases de conocimiento están representadas habitualmente en el estándar RDF, que requiere que toda la información esté almacenada en tripletas del tipo: *sujeto*, *predicado* y *objeto*. Es necesario que cada uno de estos elementos sean únicos, y esto se consigue otorgando una URI única que sirve de identificador universal para cada elemento. Este hecho, además, proporciona una característica extra que vertebra los mismos principios de la web semántica, y es que los datos están enlazados entre sí por direcciones accesibles mediante el protocolo HTTP.

Al igual que en una base de datos tradicional es posible acceder a la información almacenada a través de consultas SQL, también es posible realizar consultas a este tipo de bases de datos, utilizando el lenguaje SPARQL. Este lenguaje, al igual que RDF, es un estándar abierto del W3C[3], y compatible con las distintas representaciones de RDF.

Estos grafos son generalmente muy grandes, y contienen información de distintos tipos y muy diversa, por lo que recientemente se están investigando técnicas de aprendizaje automático que puedan aprovechar todo el potencial que tienen y hacerlo de una manera más eficiente. Entre las aplicaciones en las que dichas técnicas son de interés se encuentran la búsqueda semántica, la predicción de enlaces, la recomendación de enti-

[1] `http://wiki.dbpedia.org/`

[2] `https://www.wikidata.org/`

[3] World Wide Web Consortium: `http://www.w3c.org/`

dades o el procesamiento del lenguaje natural.

1.1 Objetivos

El objetivo de este trabajo es utilizar estas bases de conocimiento disponibles de forma totalmente libre en la Web para construir internamente grafos de conocimiento sobre los que poder aplicar las técnicas de aprendizaje automático que se han descrito anteriormente, de forma automatizada y eficiente. Sin embargo, debido a la magnitud de estos grafos, lograr ambos objetivos no es fácil.

Uno de los enfoques con mayor potencial en la actualidad para representar estos grafos de manera eficiente consiste en recurrir a modelos de factores latentes[4] que posibilitan representar el propio grafo en dimensiones más pequeñas y menos costosas computacionalmente en forma de vectores multidimensionales. Estas técnicas se engloban dentro de la rama de aprendizaje relacional estadístico, y se desarrollarán en profundidad en el capítulo 2.2 sobre el Estado del Arte.

Dentro del área del aprendizaje automático se verán técnicas de clasificación no supervisada, con algoritmos como el de los k-vecinos. Es gracias a este algoritmo con el que podemos obtener elementos (representados como vectores de *embeddings*) más parecidos unos con otros, ya sean entidades o relaciones.

Otro punto en el que se ha hecho especial énfasis en este trabajo es la escalabilidad. Se ha diseñado la aplicación pensando en un uso masivo de grafos y técnicas de aprendizaje automático, donde una arquitectura distribuida es clave para poder llevar a cabo muchas tareas de forma concurrente. Esto, unido a otras tecnologías de contenedores como Docker, facilita enormemente la distribución de la computación dentro de un clúster de máquinas, sin entorpecer la facilidad de despliegue de la aplicación.

Es frecuente ver en las empresas y en otros ámbitos similares, que se podrían ver tremendamente beneficiados por el uso de las técnicas anteriormente comentadas, que declinan implementarlas en sus sistemas por la complejidad que suponen, tanto por el tiempo de desarrollo de una solución como por la dificultad de aprovisionar las máquinas y el *hardware* necesario destinado a tal efecto. Es por tanto que la solución que se presenta aquí, a pesar de que involucra a múltiples áreas de la informática, es de fácil despliegue y uso, añadido a que es un producto de software libre (ver punto 3) y posee una extensa documentación para todo aquel que esté interesado en consultarla a fondo.

[4]También conocidos en la literatura anglosajona como *embeddings*

1.2 Estructura del documento

Esta memoria está organizada en distintos capítulos. En el segundo capítulo, que trata sobre el estado del arte, se discutirán las distintas tecnologías existentes y cuáles han sido escogidas para la realización del trabajo. Los siguientes cuatro capítulos explicarán el funcionamiento del sistema en profundidad, mientras que el último hablará sobre las conclusiones del trabajo y los siguientes pasos. Finalmente se presenta un anexo con la documentación del servicio REST desarrollado.

2 Estado del arte

En este capítulo se expondrán los distintos avances existentes en relación a las distintas tencologías que se han utilizado en la realización de todo el proyecto. En la primera sección se detallarán los distintos conceptos de los que consiste la web semántica, incluyendo RDF y SPARQL, en la segunda nos centraremos en explicar qué son los *embeddings* aplicados a los grafos de conocimiento, y en la última sección se comentarán todas las tecnologías disponibles en los distintos campos que se abordan y cuáles son las que se utilizarán en el proyecto.

2.1 Web Semántica

Desde la creación de la web en 1989, Internet ha estado orientado a ser comprensible por y para los humanos, pero no para las máquinas. El enfoque de la web semántica consiste en crear una extensión de la propia Web donde el conocimiento esté expresado de una forma explícita que permita a las máquinas y a las personas trabajar de forma cooperativa [1]. Esto plantea, automáticamente, el problema de la representación del conocimiento para que sea entendible tanto para los humanos como para los sistemas automatizados. Para aplicar este enfoque, el *World Wide Web Consortium* (W3C) ha desarrollado una serie de estándares para la representación y la consulta de datos en el contexto de la web semántica: la especificación RDF y el lenguaje SPARQL.

RDF es la especificación central de toda la familia de recomendaciones que proporciona el W3C para la web semántica. Propone un modelo de datos basado en grafos[1] para representar la información en forma de tripletas compuestas por *sujeto*, *predicado* y *objeto*. Cada uno de estos elementos tiene asociados un identificador único y universal, definido como una URI. La agrupación de varias tripletas genera un grafo de conocimiento, donde podemos identificar a los nodos (que pueden ser tanto el *sujeto* como el *objeto* de las tripletas) y las aristas que unen todas entidades. En adelante, utilizaremos el término *entidad* para referirnos a cualquiera de estos nodos y el término *relación* para

[1]La especificación completa se encuentra en: `https://www.w3.org/TR/rdf11-concepts/`

referirnos a las aristas.

Este modelo de datos propone una sintaxis abstracta para la representación de los datos, pero también provee de distintas serializaciones con las que se puede representar esos modelos en un fichero de texto. La serialización más básica de todas se basa en el estándar XML, sin embargo, existen otras que son más fáciles de leer y de interpretar, como Turtle[2] o JSON-LD[3].

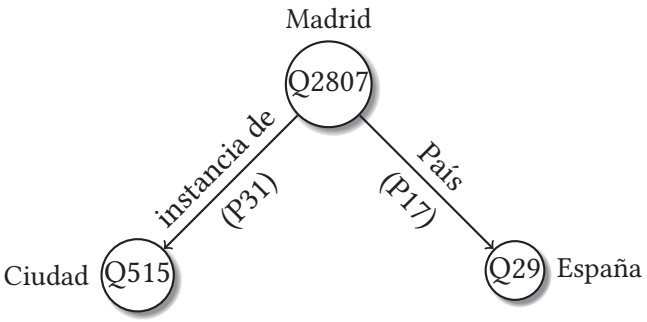

Figura 2.1: Grafo de conocimiento con Madrid como entidad principal

Este pequeño grafo contiene 2 tripletas. El identificador de las entidades y el vocabulario empleado corresponde al utilizado en `http://wikidata.org/`

En la figura 2.1 podemos ver un pequeño ejemplo de un grafo de conocimiento que contiene tres entidades y dos relaciones. Almacena información sobre Madrid, concretamente qué tipo de entidad es (P31: *Instancia de* → *Ciudad*) y dónde se encuentra esa entidad (P17: *País* → *España*). Este grafo se representa con dos tripletas en RDF, tal y como se muestra en el listado 2.1, utilizando la serialización Turtle.

```
@prefix wd: <http://www.wikidata.org/entity/>
@prefix wdt: <http://www.wikidata.org/prop/direct/>

# Primera tripleta: "Madrid" es "Instancia de" "ciudad"
wd:Q2807 wdt:P31 wd:Q515 .

# Segunda tripleta: "Madrid" esta en "pais" "espana"
wd:Q2807 wdt:P17 wd:Q29 .
```

Listado 2.1: Representación del grafo de la figura 2.1 mediante la serialización Turtle de RDF utilizando la ontología de Wikidata

Por otra parte, SPARQL es un lenguaje de consultas desarrollado para poder ser utilizado en bases de conocimiento representadas como RDF. Este lenguaje permite obtener

[2]La especificación completa se encuentra en `https://www.w3.org/TR/turtle/`
[3]La especificación completa se encuentra en `https://www.w3.org/TR/json-ld/`

la información que deseemos de una entidad concreta, obtener una lista de todas las entidades que cumplan ciertas restricciones o, incluso, crear un *nuevo* grafo dadas unas condiciones. Este lenguaje es la pieza clave que permite enlazar las bases de conocimiento ya existentes con el sistema que se presenta en este trabajo. Una muestra de este lenguaje se puede ver en el listado 2.2, cuya consulta nos devolvería todas las entidades que sean ciudades de España. Podemos ver en él que las consultas SPARQL utilizan una sintaxis muy similar a la mostrada en el listado 2.1 utilizando la serialización Turtle.

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>

SELECT ?ciudad
WHERE {
  ?ciudad wdt:P31 wd:Q515 .
  ?ciudad wdt:P17 wd:Q29 .
}
```

Listado 2.2: Consulta SPARQL utilizando la ontología de Wikidata

Un concepto fundamental de la web semántica son las ontologías o vocabularios para representar datos en RDF. Una ontología es la especificación formal y explícita de una conceptualización compartida [2] [3], o dicho de otro modo, es un modelo de las entidades y las restricciones o relaciones que se establecen entre ellos dentro de un dominio específico. El W3C dispone de dos estándares para representar las ontologías en la web: *RDF Schema* (RDFS) y *Ontology Web Language* (OWL). Para la realización del trabajo no se requiere la creación de nuevas ontologías, sino que simplemente se utilizarán las ya existentes.

2.2 Knowledge Graphs Embeddings

Como ya se ha comentado en la introducción, el objetivo principal de utilizar modelos de factores latentes, es reducir las dimensiones del problema original, con el fin de que sea manejable de forma óptima en el sentido del tiempo de procesamiento y utilización de recursos.

Por ejemplo, en algunos de estos grafos de conocimiento[4], podemos hablar de más de 40 millones de entidades y 35.000 relaciones, por lo que el número de tripletas $\mathcal{E} \times \mathcal{R} \times \mathcal{E}$ posibles que pueden ser representadas en un tensor es superior a 10^{19}. En la figura 2.2 podemos ver una representación de las entidades y relaciones de este grafo como un tensor $\mathcal{T}$, donde cada plano i, j es una matriz binaria de la relación k.

[4]Datos pertenecientes a Freebase, según [4]

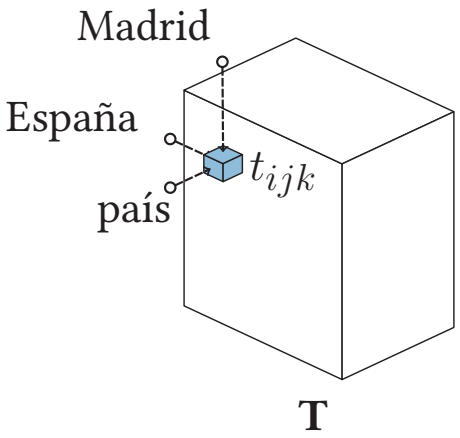

Figura 2.2: Representación de un grafo de conocimiento como un tensor.

El elemento t_{ijk} representa la relación entre *Madrid* (eje x) y *España* (eje y), almacenado en la rodaja *País* del eje z. Fuente: [4].

Los problemas de representar un grafo de conocimiento de esta forma son numerosos. En primer lugar, el tamaño que ocupa en memoria crece de forma cúbica a medida que se añaden entidades o relaciones. Segundo, las operaciones de búsqueda en grafos no son tan eficientes como otras que conocemos, y algunas de ellas son problemas *NP-Completos*, que necesitas recorrerte todo el espacio de búsqueda para encontrar la solución óptima. Por último, y como consecuencia de las anteriores, las operaciones que se quieran realizar serán muy costosas, tanto por el hardware que hay que emplear como por el tiempo que tardarán en realizarse.

2.2.1 Trabajos Previos

Uno de los primeros artículos de investigación en la línea de aprendizaje estadístico de tensores tridimensionales, aplicados a grafos de conocimiento [5], data de 2011. El algoritmo presentado en dicho artículo, conocido como RESCAL, es capaz de factorizar un tensor tridimensional que contenga la información organizada como tripletas de la forma *(sujeto, predicado y objeto)*. Es en ese mismo año cuando otra investigación [6] utiliza técnicas similares sobre bases de conocimiento como DBpedia o Freebase, extrayendo pequeñas cantidades de datos sobre las que actuar.

Sobre esta primera base se han ido desarrollando mejores métodos, unos más eficientes desde el punto de vista computacional, y otros con mejores resultados a la hora del entrenamiento, siendo capaz de converger a una solución más rápidamente. Uno de los modelos más conocidos, después de RESCAL, por la sencillez con la que está construi-

do es TransE [7]. Este modelo proporciona algunas propiedades interesantes [5] debido a cómo se construye: las relaciones son representadas como traslaciones en el espacio *embebido*, haciendo que los vectores que representan al sujeto y al objeto de una relación estén próximos debido a que hay una relación (predicado) que les une.

Otro modelo que ha ganado popularidad, sobre todo recientemente, ha sido HolE [8]. Éste ofrece una complejidad computacional similar a TransE, pero requiere de más memoria para ekjecutar. Sin embargo, es capaz de converger a una solución con mejores resultados en menos pasos que los necesarios para TransE, a costa de necesitar bastante más tiempo para cada una de estos pasos. A pesar de ello, es más eficiente que RESCAL, sobre todo en el tema de la memoria que necesita para ejecutarse, pues para operar, en lugar de utilizar el producto de tensores, utiliza la correlación circular, que reduce cuadráticamente el número de nodos necesarios y, por tanto, la memoria que requiere.

Los dos últimos modelos han sido implementados en `python` en una misma librería [9], y es la que por razones de tiempo se ha utilizado a lo largo del proyecto, ya que no hubiera sido posible realizar la implementación del propio algoritmo desde cero. Como se verá en la sección sobre el soporte tecnológico (punto 2.3.1), es posible, además, distribuir esta librería en varias máquinas para acelerar la computación.

Otros trabajos, como TransR [10] o ProjE [11] no han podido ser explorados con la profundidad requerida por razones de tiempo o por falta de implementaciones disponibles con las que poder ejecutar la generación de los modelos predictivos.

2.2.2 Propiedades

Los vectores resultantes de aplicar un algoritmo de los mencionados anteriormente son lo que conocemos como *embeddings*. Tras la ejecución del modelo, cada entidad y cada relación del grafo viene representada por un vector multidimensional, que contiene valores numéricos

Estos vectores tienen varias propiedades que los hacen interesantes para poder explotarlos con técnicas de aprendizaje automático. Una de ellas es que la distancia que les separa en el nuevo espacio *multidimensional* se corresponde, *idealmente*, con la distancia semántica que tienen cada uno de los conceptos. Dicho de otra forma, un elemento (representado por un vector de *embedding*) es más similar a otro cuanto menor sea la distancia que los separa. Asimismo, si dos elementos están muy separados entre sí, significará que semánticamente tienen poco en común.

En la figura 2.3 podemos visualizar estos conceptos de forma más sencilla. Se han generado vectores de factores latentes de tres dimensiones, ya que no es posible visualizar

[5] véase sección 2.2.2

las más de cien dimensiones que suelen tener en la literatura. El dataset utilizado incluye, entre otros conceptos, películas, ciudades y países. Se puede ver que las películas, que son semánticamente parecidas entre sí, están muy juntas. Los países y las ciudades están separados respecto de las películas, pero relativamente juntos entre sí, pues estos conceptos son ambos del ámbito geográfico.

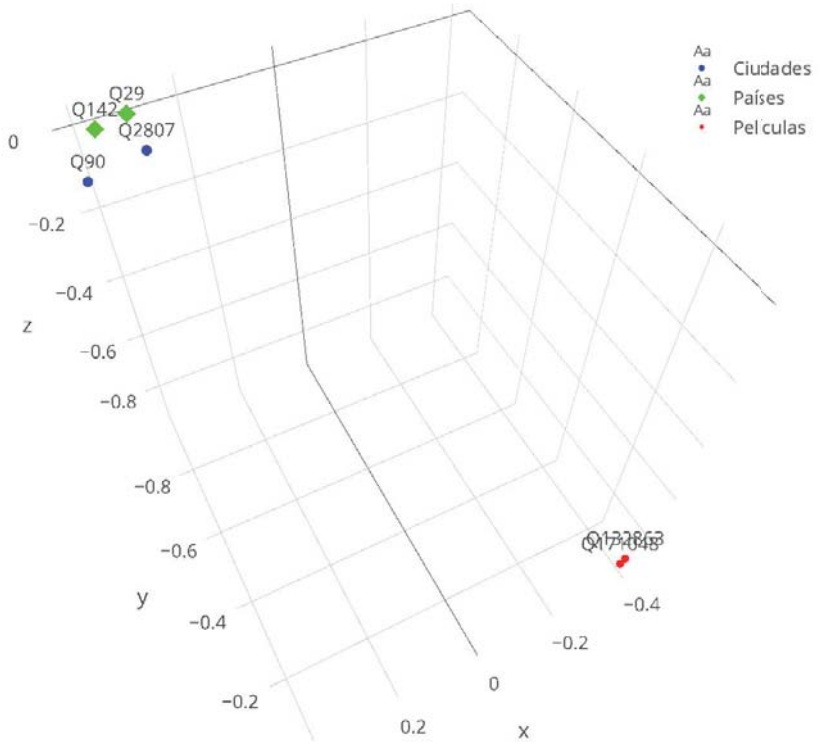

Figura 2.3: Visualización de un espacio de *embedding* tridimensional

La etiqueta de cada una de las entidades corresponde al identificador de Wikidata. Se han generado embeddings de tres dimensiones para facilitar la visualización de los datos, aunque en la práctica se utilizarán vectores bastante más grandes, imposibles de visualizar.

Quizá la forma más inmediata de explotar esta característica es utilizar el algoritmo de los k-vecinos más cercanos (KNN[6]). Este algoritmo, utilizado tanto para técnicas de clasificación supervisada como para clasificación no supervisada con el objetivo de hacer diferentes *clústers*, utiliza la distancia del coseno entre todos los elementos para definir la relación de cercanía o lejanía entre los distintos elementos. Es destacable que la distancia del coseno no es única de los espacios bidimensionales o tridimensionales, sino que es extrapolable a prácticamente cualquier dimensión, confirmando que se trata de una buena opción para nuestro objetivo.

[6]Por sus siglas en inglés: *k-nearest neighbor*

Adicionalmente, la forma de construir estos embeddings puede proporcionar otras propiedades adicionales, como ocurre con TransE. Esta particular forma de representar la base de conocimiento original, como se ha descrito en el apartado 2.2.1, permite realizar inferencias sobre un conocimiento sencillo. Concretamente: es posible utilizar los vectores en el espacio *embebido* de entidades y realizar con ellos operaciones de suma o resta, llegando así a otro vector que representa otra entidad, y a la vez guardando la semántica de la operación realizada. Por ejemplo: es posible *extraer* el concepto de «capital» a un vector ($Capital = E_{paris} - E_{france}$) y sumarle el vector que representa al país ($Madrid \approx Capital + E_{spain}$), obteniendo así una aproximación del vector que representa la capital de ese país.

2.2.3 Limitaciones

Como se ha visto hasta ahora, estos modelos de representacion son bastante potentes y eficaces, pero para poder aplicarlos es necesario disponer de una representación de la base de conocimiento como un tensor. Sin embargo esta representación no la hace adecuada para almacenar datos de tipo numérico o de cadena de texto, por lo que es necesario eliminar estos tipos de datos mientras se construye el tensor.

En este trabajo se propone utilizar bases de conocimiento expresadas en RDF, como puede ser Wikidata, y realizar la transformación a un tensor automáticamente, para poder aplicar las técnicas de aprendizaje automático ya comentadas de forma simple y directa.

2.3 Soporte tecnológico

A continuación se exponen las últimas tecnologías, relacionadas con el trabajo que se presenta, en distintos ámbitos, como lo son los ámbitos de la computación distribuida o el despliegue de entornos de desarrollo y producción, así como de distintas bases de datos, especialmente las no relacionales.

2.3.1 Computación distribuida

Uno de los objetivos del proyecto ha sido diseñar una solución escalable y que se pueda distribuir fácilmente entre un cluster de máquinas. De cara a las técnicas de la computación de alto rendimiento (HPC, por sus siglas en inglés), todas las tecnologías más utilizadas se centran en C o C++, y por tanto, en manejar a muy bajo nivel accesos a memoria de los distintos hilos o máquinas.

MPI y OpenMP

Aunque en realidad son bastante distintas entre sí, ambas se centran en directivas o funciones a bajo nivel para el compilador de C o C++, y aprovechan características como paralelismo a bajo nivel entre hilos o procesos, y vectorización de bucles.

Es una opción ampliamente utilizada a la hora de programar algoritmos para computación distribuida o aplicaciones para HPC. Sin embargo, se requiere de unos altos conocimientos y experiencia para poder crear un programa que realmente aproveche todas las ventajas, y por supuesto, un hardware disponible que sea capaz de ejecutarlo de esa forma.

Asimismo, como se verá más adelante, el proyecto está desarollado prácticamente en su totalidad utilizando Python como lenguaje de programación, por lo que intentar desarrollar una solución basada en estas tecnologías hubiera sido poco viable para la finalización del mismo.

Celery

Celery es una librería escrita en Python que proporciona herramientas para crear y gestionar una cola de procesos y distribuirla en una o varias máquinas. Es una solución bastante distinta a las anteriores, sobre todo, porque permite trabajar con el lenguaje Python de forma nativa.

Quizás el uso principal para el que fue pensado no es el de computación de alto rendimiento, pero sin duda por diseño, es perfectamente válido para aplicar una tecnología de este tipo. Tiene una arquitectura similar a la de editor-subscriptor, donde hay un único nodo que conoce qué tareas hay que ejecutar y cuántos nodos *trabajadores* hay disponibles para ejecutar las distintas tareas.

Esta librería necesita de otras tecnologías como RabbitMQ o Redis para gestionar la cola de las tareas, sincronizarlas, distribuirlas y ejecutarlas en los distintos nodos que componen el clúster de Celery. Dado que Redis es una base de datos de tipo *noSQL* y almacena datos en forma de parejas de clave - valor, se ha decidido utilizar ésta para poder aprovechar esta característica con otros servicios.

2.3.2 Desarrollo web

Para el desarrollo de la aplicación web, hoy en día existen multitud de librerías y *frameworks* que permiten tanto agilizar el desarrollo como conseguir unos mejores resultados en materia de rendimiento y presentación para el usuario final.

Del lado de los *frameworks*, tenemos como al principal exponente a AngularJS, que provee una solución completa del tipo MVC (*Model - View - Controller*). En su última

versión, AngularJS 2, utiliza el lenguaje *Typescript* y entre otras cosas permite declarar y utilizar tipos, para utilizarse sobre todo en tiempo de compilación.

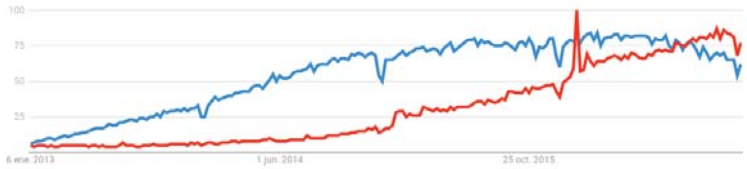

Figura 2.4: Análisis de Google Trends sobre el interés de ambas herramientas

En esta gráfica se puede observar la evolución de los términos de búsqueda relacionados con AngularJS (azul) y ReactJS (rojo). Fuente: Google Trends.

Sin embargo, en los últimos años, el interés de AngularJS se ha visto incapaz de competir con la expectación causada en torno a ReactJS (ver figura 2.4), sobre todo por ser la primera una opción demasiado pesada para muchas aplicaciones, cargando módulos innecesarios y ralentizando la experiencia de usuario. ReactJS, por su parte, se trata de una librería, no de un *framework*, por lo que para muchas de las cosas que AngularJS ofrece por defecto, es necesario hacer uso de librerías extra para utilizar junto con ReactJS.

Una de las razones por las que está ganando terreno en el mercado se debe a la particular forma que tiene de manipular el DOM. Esta librería mantiene internamente un DOM virtual, de tal forma que todos los cambios que se realicen primero se aplican al DOM virtual, y después se llevan a cabo en el DOM real, pero modificando solamente los elementos que han cambiado en lugar de recargar todo el DOM de nuevo. Esto agiliza enormemente cualquier cambio, ya que evita tener que recargar un DOM por completo, que es muy costoso computacionalmente y cambia sólo lo estrictamente necesario.

Lenguajes de desarrollo de aplicaciones web

Aunque actualmente los navegadores Web sólo interpretan el lenguaje Javascript, éste tiene varias limitaciones a la hora de desarrollar aplicaciones complejas que puede incrementar el tiempo de programación y dificultar la comprensión del código. Aunque varios de estos problemas han sido solucionados con la llegada del estándar ECMAScript 6[7], en los últimos años han proliferado algunos lenguajes orientados a facilitar la programación web, que son posteriormente compilados a un lenguaje Javascript comprensible por el navegador. Principalmente podemos señalar estos dos:

[7]https://www.ecma-international.org/ecma-262/6.0/

TypeScript Este lenguaje, desarrollado con el apoyo de Microsoft, proporciona una sintaxis muy similar a Javascript, pero añade tipos a las variables, opción que para aplicaciones grandes con muchos componentes, puede ser de mucha ayuda. Sin embargo requiere de un compilador a Javascript que realice todas las comprobaciones de tipos en tiempo de compilación.

CoffeeScript Al contrario que TypeScript, este lenguaje modifica bastante la sintaxis original de Javascript, proporcionando a cambio un código más limpio de ver y mantener. También es posible utilizar estructuras que no existen en Javascript, como objetos, y utilizar variables declaradas como constantes. Requiere también de un compilador que convierta a Javascript el código escrito en CoffeeScript.

El nuevo estándar ECMAScript 6 incluye muchas de las mejoras que contienen estos lenguajes, y es previsible que los navegadores implementarán estas funcinalidades con el tiempo, haciendo innecesario el uso de un compilador. Por contra, hasta que ese momento llegue, será necesario el uso de un *transpilador* que convierta el código a un lenguaje comprensible a los navegadores. Para el desarrollo de la aplicación web se ha decidio optar por este lenguaje por las razones comentadas.

2.3.3 Persistencia de los datos

Existen diversas tecnologías en todo lo relacionado a la persistencia de los datos, ya sean éstos datos relacionales, accesibles mediante una base de datos SQL o datos binarios, como los que disponemos a la hora de generar los grafos y los distintos modelos. A continuación se exponen las distintas soluciones propuestas y si finalmente han sido implementadas en el proyecto.

Binarios

Durante la ejecución de las distintas partes del proyecto se generan archivos binarios, ya sea durante la creación del dataset, después del entrenamiento, o para la indexación necesaria para ofrecer los servicios de una forma eficaz. Estos archivos, por su tipología, deben ser almacenados en el sistema de ficheros del sistema operativo, en lugar de ir a bases de datos, principalmente, porque tienen un tamaño muy grande (en proporción al *dataset*), y porque otras librerías auxiliares esperan que estos datos estén como ficheros.

NoSQL

Existen muchas bases de datos para el almacenamiento de información de forma no relacional. De ellas, varias están orientadas a trabajar con clave-valor, otras con grafos,

y otras orientadas a trabajar con documentos (generalmente descritos con el lenguaje JSON).

Redis Esta base de datos está pensada para almacenar solamente pares de clave - valor, y la principal ventaja que ofrece es la rapidez de respuesta. Generalmente aloja toda la base de datos en memoria RAM, para que el acceso sea lo más rápido posible, con un coste muy alto, el de tener mucha memoria en uso y quizás sin que sea accedida.

Está escrito en lenguaje C y su uso se ha extendido bastante, por lo que es posible acceder a numerosas facilidades que permiten utilizar esta base de datos casi con cualquier tecnología: hay disponibles tanto librerías compatibles con `python` como contenedores ya construidos, listos para utilizar con `docker`, entre otras muchas otras cosas.

ElasticSearch Para realizar el almacenamiento de datos semiestructurados tenemos la posibilidad de utilizar ElasticSearch. Sin embargo, donde se puede aprovechar todo el potencial de esta base de datos es utilizándola como motor para búsquedas dentro de la aplicación.

Está basado en un desarrollo previo de la Fundación Apache: Lucene. Ambos están escritos en Java, pero proveen, entre otros métodos, una interfaz HTTP REST para acceder a la base de datos, ya sea para realizar consultas o para insertar datos. Como otros desarrollos de esta fundación, esta herramienta es fácilmente escalable a medida que los datos o los clientes crezcan.

Tanto por ser escalable, como por ser fácilmente accesible, se ha decidido usar esta base de datos para el servicio de búsquedas de los elementos en la parte Web de la aplicación. Posibilita a los usuarios encontrar el identificador de una entidad conociendo únicamente el nombre o la descripción.

MongoDB Otra de las tecnologías que se basan en almacenar datos semiestructurados es MongoDB. Al igual que Redis se ha ido convirtiendo en una opción cada vez más válida para nuevos desarrollos. Proporciona mucha rapidez a la hora de acceder a los distintos datos, y al contrario que ElasticSearch, en vez de estar escrito en Java, el desarrollo se ha llevado a cabo mayoritariamente en C++.

Sin embargo no proporciona muchos valores añadidos, por lo que en lugar de diversificar los distintos métodos de almacenamiento, se ha decidido no utilizar.

SQLite

Quizás no sea la opción más conocida en lo que a soluciones SQL se refiere, pero es una solución muy ligera y sencilla de utilizar y de implementar. Además carece de un

modelo de cliente - servidor y almacena todos los datos en un único fichero de disco.

Esta solución permite crear prototipos de forma rápida, y posteriormente es bastante sencillo cambiar a otra solución basada en SQL más robusta, como puede ser PostgreSQL o MySQL, que sí tienen un servidor central al que acceden varios clientes.

2.3.4 Despliegue

Un punto muy importante dentro del ciclo de vida de un software es el despliegue a un entorno de producción. Esta aplicación posee varios servicios que deben estar conectados entre sí, y sin duda no es algo fácil de mantener, pues obliga a estar pendiente de muchos de estos servicios a la vez.

Dado que prácticamente todo el desarrollo que se está llevando a cabo se realiza con herramientas multiplataforma, y todas son compatibles con los sistemas Linux, se ha optado por una topología basada en contenedores, y para facilitar todas las tareas de creación de contenedores y su puesta en funcionamiento, se ha decidido utilizar Docker. Muchas de las aplicaciones con las que debe estar conectada nuestra aplicación disponen de un contenedor Docker mantenido por los desarrolladores originales, lo que permite conectar los distintos servicios fácilmente y utilizarlos configurando lo mínimo posible.

3 Trabajo realizado

La totalidad del proyecto realizado se puede dividir en tres grandes partes, que se ha decidido exponer en diferentes capítulos en el presente documento

4 Biblioteca KGE-Server Esta primera parte se puede considerar como el **núcleo del proyecto** en sí mismo, que ha sido desarrollado como una biblioteca escrita en `python` y que se puede ejecutar en cualquier lugar donde se disponga del código y del entorno de desarrollo.

Como entorno de desarrollo se ha utilizado un contenedor Docker, facilitando la portabilidad del código a cualquier máquina compatible con esta tecnología.

5 Servicio REST Esta segunda parte permite acceder a todas las funciones que proporciona la biblioteca a través de peticiones HTTP hacia un único servidor que es el que está ejecutando el código remotamente por medio de una API REST.

6 Aplicación Web Esta última parte se ha desarrollado como complemento al servicio REST mencionado anteriormente. Permite realizar todas las operaciones que se proporcionan de una forma más amigable al usuario, y proporciona métodos para realizar búsquedas por texto y no sólo por entidad.

Dado los importantes aportes en materia de facilidad de despliegue y en uso de la aplicación que se han proporcionado con los dos primeros componentes, se ha decidido liberarlos como Software libre como una forma de devolver a la comunidad los beneficios que ésta ha proporcionado en forma de herramientas a la hora de desarrollar todo el código. Esta operación se ha llevado a cabo en la plataforma GitHub[1], haciendo uso de la herramienta de control de versiones Git.

Se ha elegido una licencia permisiva como la LGPLv3[2], ya que permite utilizar el código de estos componentes en proyectos de software privados siempre y cuando sean utilizados como una biblioteca y el código no cambie. De utilizar la licencia GPLv3, se

[1] `https://www.github.com/vfrico/kge-server`
[2] `https://www.gnu.org/licenses/lgpl.html`

estaría imposibilitando el uso de estos componentes en cualquier proyecto que no fuese liberado a la comunidad, impidiendo así uno de los objetivos de este proyecto, que es poder ser utilizado por cualquier entidad. Asimismo, la licencia escogida sí que obliga a que los cambios que se realicen en el software reviertan a la comunidad en beneficio de todos, en contra de otras licencias aún más permisivas como Apache o MIT.

Apoyado por el hecho de liberar el sofware, también se ha desarrollado una extensa documentación que puede ser consultada en formato HTML[3]. Explica en profundidad el propósito de la librería, como se ha desarrollado, los componentes que tiene, y sobre todo cómo funciona cada uno de forma detallada. Prácticamente todas las funciones desarrolladas en Python tienen expuesto su funcionamiento para facilitar que cualquier persona interesada en el proyecto pueda comprender cada componente y desarrollar las funciones que necesiten.

[3] `https://vfrico.github.io/kge-server/`

4 Biblioteca KGE-Server

Esta es la parte principal del proyecto, la que se encarga de llevar a cabo todo el proceso de gestión y de decisión. Se ha decidido desarrollar como si fuera una biblioteca de funciones escrita en `python`. Principalmente, tiene tres módulos distintos:

Dataset Este módulo se encarga de gestionar la **creación**, el **guardado** y la **carga** de los conjuntos de datos que serán utilizados en los siguientes módulos del proyecto.

Algorithm El módulo Algorithm se encarga de definir el algoritmo que se utilizará para el modelo y el entrenamiento del mismo.

Server El módulo Server se encarga de proveer al usuario los servicios de predicción de tripletas y búsqueda por similitud semántica.

Para una visión más clara de la estructura de esta biblioteca, se puede ver la figura 4.1, donde se muestra un diagrama con los distintos componentes que la conforman, y cómo estan conectados entre sí. Estos tres componentes se van a definir a continuación, especificando qué hacen con los datos recibidos del paso anterior, y qué ofrecen como resultado.

4.1 Dataset

Este componente es el más básico para el funcionamiento del resto de la aplicación. Es capaz de crear un objeto `Dataset` sin importar de dónde vengan los datos: permite cargar las tripletas desde un fichero CSV o TSV, desde un archivo JSON, o desde incluso una consulta SPARQL. Esto provee al siguiente nivel de una abstracción del origen del grafo de conocimiento, permitiendo aplicar los mismos métodos con independencia del origen de los datos.

Puesto que el objeto `Dataset` es fundamentalmente una clase abstracta válida para cualquier conjunto de datos, es necesario extenderla para crear distintos objetos que hagan uso de las distintas fuentes. Durante la realización del proyecto nos hemos centrado

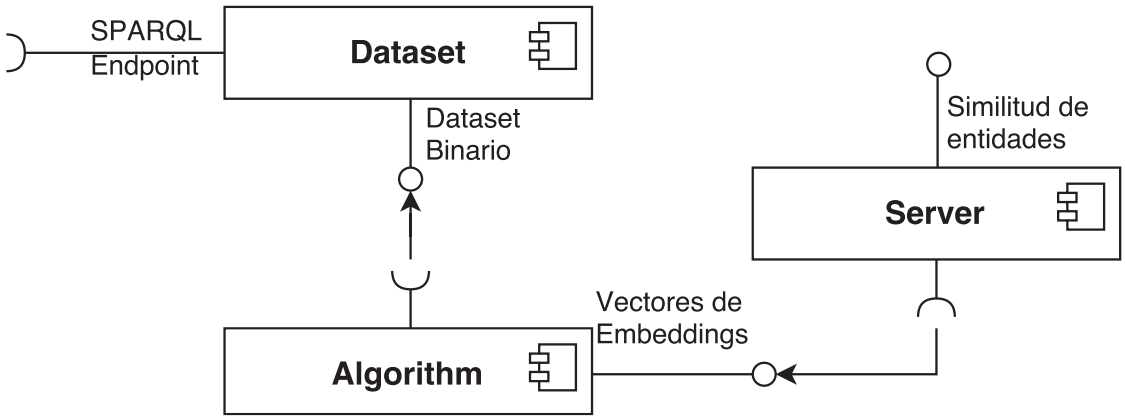

Figura 4.1: Estructura modular de la biblioteca KGE-Server.

en utilizar como fuente de datos el *endpoint* SPARQL provisto por *Wikidata*, por lo que se ha creado una clase `WikidataDataset`, válida para generar conjuntos de datos a partir de este endpoint. A modo de pruebas, también se ha creado una clase, válida para el `endpoint` SPARQL de *DBPedia* en Español, que permite generar datasets a partir de esta fuente. Adicionalmente, dado que este servicio utiliza vocabularios estándares del W3C, sería fácilmente portable a otros conjuntos de datos.

4.1.1 Encadenamiento de consultas

Con el objetivo de obtener un conjunto de datos mucho más rico y con más conexiones que las que otorga una simple consulta SPARQL, los datasets que hacen uso de este servicio pueden realizar consultas encadenadas para extraer más información acerca de cada elemento, y generar niveles extra de relaciones. Esto permite al algoritmo de generación de *embeddings* encontrar más nodos para relacionar entre sí, y como consecuencia, los resultados de las predicciones serán más precisas.

Podemos ver un ejemplo de cómo se realiza este encadenamiento en la figura 4.2. Para obtener un conjunto de datos con todas las ciudades de Europa se hará una primera consulta que extraerá todas las ciudades a través de una sola consulta SPARQL. Llamaremos a esto *nivel semilla* o *nivel 0*, ya que partiremos de éste para encadenar el resto de consultas. El siguiente *nivel* se obtendría haciendo una nueva consulta por cada ciudad, que la relacionará con nuevas entidades a las que está conectada, como por ejemplo, el país al que pertenece. Este conjunto de entidades se considerarían como de *nivel 1* y a partir de ellas se podría realizar de nuevo el mismo encadenamiento, y obtener un *nivel 2* con entidades de diverso tipo.

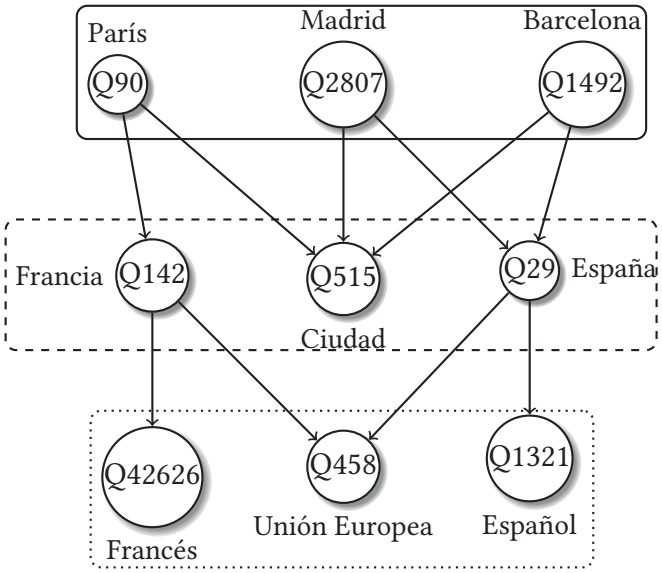

Figura 4.2: Grafo creado a través del encadenamiento de consultas.

Las entidades agrupadas con la línea recta son consideradas como el *nivel semilla*, mientras que las entidades agrupadas con la línea de guiones corresponderían al *nivel 1* y las entidades agrupadas con la línea de puntos corresponderían al *nivel 2*.

Este proceso, sin embargo, no es barato de realizar, en términos computacionales. Por cada entidad que se añade al conjunto de datos es necesario realizar una consulta al servicio SPARQL. Por ello, para aprovechar la capacidad actual de múltiples hilos de ejecución que tienen los procesadores, se abrirán varias conexiones al mismo servidor, con el objetivo de acelerar el proceso (a costa de un alto tráfico de red). Es posible configurar el número de *niveles* que se desean encadenar para obtener las entidades, aunque lo recomendado sería moverse entre 2 y 3 para conjuntos de datos comunes.

4.1.2 Creación del *graph_pattern*

Para poder generar el encadenamiento mencionado anteriormente, es preciso proporcionar a la biblioteca el *graph_pattern* que debe seguir para generar el *nivel semilla* y encadenar correctamente el resto de niveles. Como en última instancia el software va a realizar una consulta SPARQL, la sintaxis utilizada será muy similar.

Internamente, la biblioteca utilizará un esqueleto para la consulta SPARQL muy básico para obtener todas las entidades relacionadas con un único elemento. Este esqueleto lo podemos ver en el listado 4.1, pero es sólo aplicable a los conjuntos de datos obtenidos de Wikidata, porque utiliza un vocabulario propio. Se puede observar que necesita tres

variables: `?subject`, `?predicate` y `?object`, que debemos hacer que tomen algún valor a partir del *graph_pattern* que utilicemos.

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>

SELECT ?subject ?predicate ?object
WHERE {
  # Graph Pattern
}
```

Listado 4.1: Esqueleto de consulta encadenada para Wikidata

Por ejemplo, partamos de una consulta SPARQL simple y conocida, como la consulta 2.2 mostrada en el capítulo sobre el estado del arte, a partir de la cual somos capaces de extraer todas las ciudades que hay en España. Simplemente realizaba una operación SELECT sobre la variable `?ciudades`. El grafo que crearemos tendrá como sujeto de las tripletas a las propias ciudades, por lo que para el *graph_pattern* utilizaremos la variable `?subject` en lugar de `?ciudades`. El resto de las variables `?predicate` y `?object` las asignaremos en una tripleta extra que añadimos: `?subject ?predicate ?object`. Finalmente nuestro *graph_pattern* quedaría así:

```
?subject wdt:P31 wd:Q515 . ?subject wdt:P17 wd:Q29 . ?subject ?
    predicate ?object
```

Listado 4.2: Ejemplo de *graph_pattern* para obtener un dataset de Wikidata

4.2 Algorithm

El componente `Algorithm` es el más relevante, pues es el encargado de crear un modelo de *embeddings* acorde con el conjunto de datos para poder aplicar métodos de predicción con ellos.

Como se ha mencionado en el capítulo 2.2 del estado del arte, existen numerosos modelos de factores latentes para aplicar a grafos de conocimiento, sin embargo, a la hora de utilizarlos en el proyecto, se ha decidido centrarse sólo en uno: TransE. Es un algoritmo que ofrece unas buenas métricas sin tener un coste computacional especialmente alto, como sí lo tienen otros algoritmos homólogos.

Debido a que el tiempo de realización del proyecto es limitado, se ha optado por tomar una implementación existente, la de la librería `scikit-kge` [8] [9], y adaptarla a las necesidades del proyecto. Esta libería permite la reutilización de su código, gracias a la licencia MIT con la que está publicado. Adicionalmente cuenta con el algoritmo

HolE, por lo que sería razonablemente sencillo extender el proyecto para utilizar este otro algoritmo en un futuro.

El usuario debe proveer de una serie de parámetros que necesita el algoritmo para poder ejecutarse. Entre ellos están el *embedding_size*, que dice qué tamaño debe de tener el vector de factores latentes, *max_epochs*, que especifica el número máximo de iteraciones del algoritmo, y *margin*, que indica el margen de error que debe tenerse en cuenta para durante el entrenamiento. Para la generación del modelo, el usuario también deberá decir a qué conjunto de datos se debe aplicar (devuelto por el componente anterior, `Dataset`).

Puesto que este componente se engloba dentro del aprendizaje estadístico, en el estado del arte se mencionan distintos métodos para evaluar el rendimiento del modelo, tales como la predicción de enlace o el *rank@10*. Hubiera sido muy deseable haber incluido distintos métodos para realizar esta evaluación, y poder probar distintos parámetros de entrenamiento para poder decidir cuáles son esos parámetros que mejor responden a nuestro conjunto de datos. Sin embargo, debido al largo tiempo que toma en evaluar el modelo, se ha preferido dejar a un lado esta parte y generar un modelo con todos los parámetros de entrada.

Como salida, este método genera dos matrices M, $\mathcal{E} = M(N_{ent} \times s)$ y $\mathcal{R} = M(N_{rel} \times s)$, que contienen los vectores de embeddings de tamaño s para cada una de las entidades N_{ent} y relaciones N_{rel}.

4.3 Server

Este último componente, `Server`, provee de distintos servicios de predicción respecto el dataset, como predicción de entidades similares, similitud entre dos elementos, o predicción de enlace. Los servicios que están implementados y funcionan completamente son los de similitud de dos elementos y predicción de entidades similares.

Ya que contamos con los vectores de *embeddings* del componente anterior, podemos utilizar el algoritmo KNN para obtener los elementos más cercanos en el espacio, siendo éstos los elementos más similares en términos semánticos (véase sección 2.2.2 del capítulo sobre el estado del arte). Del mismo modo, podemos calcular la distancia entre dos elementos para conocer cómo de parecidos son ambos.

Sin embargo, este algoritmo, para decir quién está más cercano que otros necesita calcular las distancias de ese elemento con todos. Esto es un problema cuando se habla de grandes bases de conocimiento, a las cuales está orientado todo el desarrollo del trabajo. Para evitar esto, se ha decidido utilizar una biblioteca para hacer el cálculo de distancias de estos embeddings: `Annoy` [12]. Esta biblioteca requiere un tiempo de ejecu-

ción previo al de la consulta de distancias, en el que precalculará un árbol de búsqueda que contendrá las zonas contiguas. Esto ayudará a la hora de calcular las distancias, ya que en lugar de hacerlo en todo el espacio *multidimensional*, sólo lo deberá de realizar en las regiones contiguas a las que se encuentra el punto de origen. Con este enfoque reducimos los cálculos necesarios a un espacio tan pequeño que los resultados se obtienen prácticamente al instante.

También es posible obtener una métrica de distancia entre dos entidades, para conocer la distancia relativa entre ambas. Esto puede ser de utilidad en ciertos casos, por lo que se ha implementado una función para darle soporte.

5 Servicio REST

Como ya se ha comentado antes, el servicio REST permite acceder a todas las funciones de la biblioteca sin necesidad de tenerlo instalado en tu ordenador, simplemente con peticiones HTTP.

5.1 Creación de los métodos HTTP

En primer lugar ha sido necesario definir las tareas que se deben implementar en este servicio. Se ha hecho un análisis de las partes que conlleva toda la ejecución de la librería, y sigue siempre el orden en el que se desarrollaron los módulos de la biblioteca KGE-Server:

1. Primero es necesario crear un objeto `Dataset` y alimentarlo con una consulta para insertar tripletas en el conjunto de datos.
2. Después, una vez que este proceso haya finalizado, es preciso ejecutar la clase `Algorithm` para generar los vectores de *embedding* con unos hiperparámetros determinados.
3. Una vez que tenemos el modelo entrenado, debemos generar un índice de búsqueda con la clase `Server`, que permitirá realizar las predicciones en un tiempo prácticamente instantáneo.
4. Finalmente, después de realizar todos estos pasos, el módulo `Server` estará preparado para contestar llamadas a la función de obtener entidades similares.

De esta serie de pasos se puede deducir lo siguiente:

- Necesitaremos de una colección llamada `/dataset/` que permita gestionar todo lo relacionado con la creación, modificación y exploración de los conjuntos de datos, así como de la predicción de entidades similares.
- A causa de que las tareas de entrenamiento se alargan en el tiempo y no pueden ser respondidas en una única petición HTTP, se ha decidido tomar un modelo de

petición asíncrono[1] para liberar el socket HTTP entre la petición y la respuesta. Para realizar esto ha sido necesario crear una colección `/tasks/` que almacenarán todo tipo de información sobre cada tarea en ejecución.

- Para facilitar la reutilización de los parámetros de entrenamiento entre distintos conjuntos de datos se ha decidido crear la colección `/algorithm/` permitiendo así disponer de un objeto con todos los hiperparámetros necesarios, y simplemente habrá que proporcionar un identificador de algoritmo para empezar la generación de *embeddings*.
- *Adicionalmente*, para facilitar el desarrollo de la aplicación web, se ha implementado un servicio de búsqueda de entidades, orientado a abstraer al usuario final del identificador real de un recurso, utilizando únicamente el nombre común.

En el anexo A se pueden consultar todos los métodos que se han implementado finalmente, junto con la documentación original del proyecto acerca de la utilización de cada método. Se provee un ejemplo de uso para cada llamada, con la petición HTTP y la respuesta del servidor.

5.2 Implementación del servicio

Después de haber decidido cuáles son todos los métodos que debería tener el servicio, es necesario implementarlo. A continuación se expondrán los distintos componentes que conforman el servicio REST desarrollado y cómo se conectan unos con otros. En la figura 5.1 se puede observar un diagrama de las interfaces que provee cada componente y cómo se relacionan entre sí.

Para comenzar a desarrollar este servicio, necesitaremos una forma de recibir y enviar peticiones HTTP a través de un servidor web. Por facilidad, ya que todo el proyecto está desarrollado utilizando `python` como lenguaje de programación, se utilizará `Falcon`, un *framework* que permite realizar de forma simple un servicio REST y puede hacer uso de la biblioteca sin realizar apenas ningún cambio. El servidor que se recomienda utilizar en estos casos es *Gunicorn*, que se combina fácilmente con este *framework* y permite configurar el número de procesos y threads dedicados a escuchar peticiones HTTP.

Como hemos podido ver en la sección anterior, el servidor permite realizar tanto la creación de conjuntos de datos como el entrenamiento de modelos para la obtención de *embeddings* o la predicción de tripletas similares basándose en estos modelos. Sin embargo, el entrenamiento de los conjuntos de datos ha quedado reducido exclusivamente

[1]La documentación sobre cómo llevarlo a cabo y otras recomendaciones se encuentra en: `http://restcookbook.com/Resources/asynchroneous-operations/`

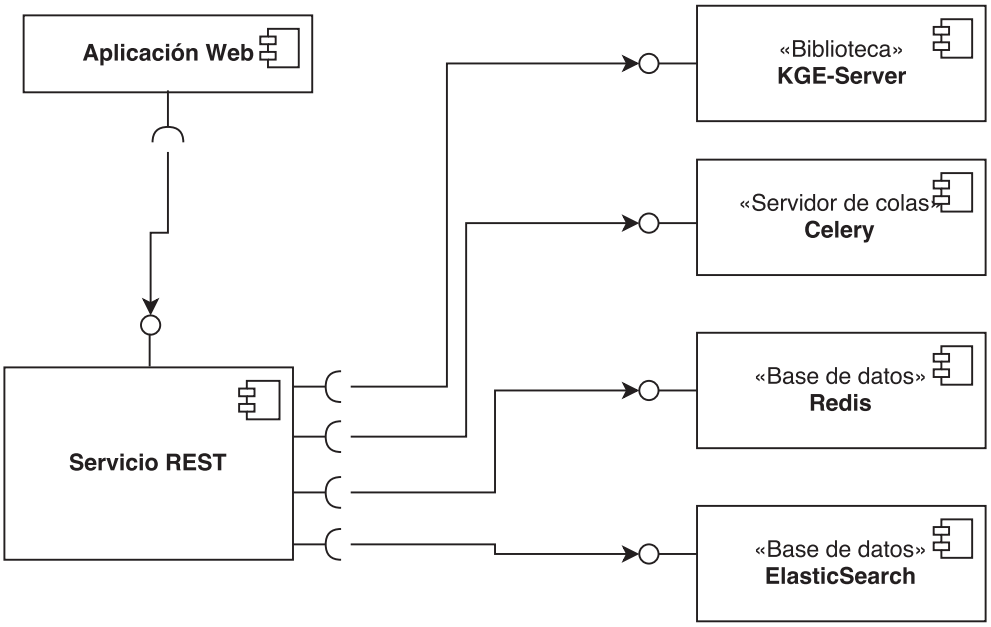

Figura 5.1: Arquitectura del servicio REST y de la conexión de la aplicación web al resto del sistema.

al uso del algoritmo TransE [7], que es bastante más rápido que HolE [8] y además, por cómo está modelado, posee unas propiedades que pueden ser interesantes para aplicarlas en ciertos casos, como se ha visto en la sección 2.2.2.

5.2.1 Soluciones HTTP utilizadas

Varias de las funcionalidades que tardan mucho en ejecutarse se han implementado como peticiones asíncronas [13]; esto es, cuando se realice una petición que puede durar potencialmente varias horas, o al menos, superar el tiempo de respuesta máximo que tiene el servidor, en lugar de devolver la respuesta, se devuelve un nuevo recurso creado, del tipo tarea. Esta recurso permite, fácilmente, realizar un seguimiento de cómo progresa la ejecución de la tarea en el servidor. Cuando se haya completado, la propia tarea redirigirá con el estado `303` al recurso creado con la finalización de la tarea.

Funcionamiento de las peticiones asíncronas

Por ejemplo, si queremos crear una tarea para insertar tripletas en el dataset, necesitaremos hacer la siguiente petición:

```
GET /datasets/1/generate_triples HTTP/1.1
Content-type: application/json
```

```
{
  "generate_triples": {
    "levels": 2,
    "graph_pattern": "?subject wdt:P31 wd:Q131212 . ?subject ?
predicate ?object ",
  }
}
```

Listado 5.1: Cuerpo de la petición POST

En lugar de que el servidor nos devuelva un código de estado `200 OK` recibiremos un `202 ACCEPTED` y tanto en el cuerpo de la petición como en las cabeceras HTTP veremos los detalles de la nueva tarea recibida.

```
HTTP/1.1 202 ACCEPTED
Content-type: application/json
Location: /tasks/6

{
  "task": {
    "id": 6
  },
  "status": 202,
  "message": "Task 6 created successfuly"
}
```

Listado 5.2: Respuesta como un recurso tarea

En adelante podremos consultar el estado de la tarea con la siguiente petición HTTP, utilizando la misma cabecera `Location` recibida en la respuesta del servidor.

```
GET /tasks/6 HTTP/1.1
```

Y obtendremos como respuesta el recurso `task`, que nos informa sobre la progresión de la tarea en el servidor. Dependiendo de la tarea concreta informará con más o menos detalle del progreso.

```
HTTP/1.1 200 OK
Content-type: application/json

{
  "task": {
        "id": 6,
    "state": "STARTED",
```

```
    "progress": {
      "current": 499,
      "total_steps": 2,
      "total": 13467,
      "current_steps": 1
    }
  }
}
```

Listado 5.3: Cuerpo de la respuesta al recurso tarea

Cuando esta tarea finalice, en lugar de obtener el código de estado `200 OK`, si todo ha funcionado bien obtendremos el código `303 SEE OTHER`, con la cabecera `Location` apuntando al recurso que fue creado o en este caso, modificado (el dataset). Dependiendo del comportamiento del agente de usuario, podrías ser redirigido automáticamente al nuevo recurso indicado por la cabecera `Location`.

```
HTTP/1.1 303 SEE OTHER
Content-type: application/json
Location: /datasets/1

{
  "task": {
    "state": "SUCCESS",
    "id": 6
  }
}
```

Listado 5.4: Respuesta de una tarea que ha finalizado

Si hubiera algún error durante la ejecución de esta tarea, se informará al usuario cambiando el valor del estado que devuelve (parámetro `task.state`), e incluyendo una traza de la excepción producida. El usuario necesitará realizar la llamada original que generó la tarea para volver a ejecutarla en el servidor.

Para poder realizar estas tareas de forma *asíncrona*, se utiliza la biblioteca `celery` [14], que permite crear colas de tareas para ejecutar en una o varias máquinas. Esto permitiría, en un futuro, distribuir estas tareas en un *cluster* de máquinas.

Mecanismo CORS

Es destacable que se ha necesitado un estudio del mecanismo CORS[2] que aplican los distintos navegadores web para bloquear las peticiones a servidores distintos de los que

[2]Se puede consultar detalles del funcionamiento de este mecanismo en: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS`

están mostrando la información en la web. Tras este breve estudio, se ha optado por implementar cabeceras que sigan las directivas establecidas, para facilitar en la medida de lo posible el desarrollo de la aplicación web, y propiciar que el servicio creado sea portable independientemente de dónde se encuentre el servidor que lo proporcione.

Gracias a la biblioteca `Falcon` que se ha utilizado, simplemente se ha tenido que instalar un paquete que configura las cabeceras mencionadas en todas las peticiones automáticamente, facilitando nuevamente el desarrollo.

5.2.2 Servicio de búsqueda y autocompletado

El servicio de búsqueda se ha implementado en favor de una mejor funcionalidad de la aplicación web. De esta forma se facilita a los usuarios la utilización de los distintos modelos y aplicación directa sin necesidad de que los usuarios conozcan exactamente el identificador de la entidad que desean consultar.

Este servicio se ha decidido implementar utilizando ElasticSearch. Este almacén de datos necesita documentos en un formato estructurado como JSON, y a partir de un modelo de búsqueda que se debe indicar, es capaz de proporcionar servicios HTTP para autocompletar un cuadro de texto y proporcionar información adicional sobre las entidades.

Internamente, realiza consultas contra el *endpoint* SPARQL asociado al conjunto de datos y recoge las relaciones que almacenan la etiqueta de cada entidad y la descripción, que se corresponden con las relaciones `rdfs:label` y `rdfs:comment`. Adicionalmente, el lenguaje RDF también permite expresar el idioma en el que están escritos estos textos, por lo tanto a la hora de construir este índice de autocompletado es posible indicar el idioma o idiomas que deseamos configurar para proporcionar este servicio.

6 Aplicación Web

Por último, tenemos la parte de la aplicación web. Se ha optado por realizar una interfaz web con la que manejar el sistema tanto por la facilidad de desarrollo y las posibilidades que ofrece para poder ejecutarse prácticamente en cualquier parte. Además, como ya contamos con una parte de la aplicación que funciona mediante peticiones HTTP siguiendo el patrón de diseño REST, es bastante sencillo construir una aplicación web que haga uso de este sistema.

Después de una reflexión (que puede ser consultada en el punto 2.3.2, dentro del capítulo sobre el estado del arte) sobre las distintas opciones que existen a día de hoy para desarrollar aplicaciones web, se ha optado por utilizar la librería `ReactJS` para manejar los distintos componentes visuales, y librerías auxiliares para manejar el estado y las distintas pantallas de las que dispone la aplicación.

En cuanto al lenguaje de programación para esta parte se ha optado por utilizar Javascript en, al menos, su versión ES6, que dispone de características muy necesarias, como paradigmas de programación orientado a objetos o nuevas API's nativas. Para que sea compatible con navegadores que aún no soportan la última versión del estandar de JavaScript, se utiliza la librería `babel`, que es capaz de convertir el código existente a una versión anterior de Javascript.

6.1 Desarrollo de la aplicación

Como se ha mencionado ya en el punto 2.3.2 sobre el estado del arte, ReactJS no es un framework, sino una librería, y como tal no es posible generar un proyecto con él. Ha sido necesario utilizar una plantilla[1] con todas las bibliotecas Javascript ya preparadas para empezar a programar la aplicación.

Se ha decidido seguir una metodología de desarrollo web orientada a componentes, esto es, separar toda la página web original en diferentes módulos independientes. De esta forma conseguimos un alto grado de reutilización de código a lo largo de la aplica-

[1] Proyecto disponible en `https://github.com/jpsierens/webpack-react-redux`

ción, ya que es frecuente que un componente se encuentre en varios sitios dentro de la aplicación.

Aparte de ReactJS, ha habido otras dos bibliotecas Javascript que han sido de vital importancia para el desarrollo de la web. Estas han sido material-ui y la otra ha sido Redux. La primera es principalmente una biblioteca que provee de componentes que siguen las diretrices del diseño *Material Design* [2], mientras que la segunda es un complemento a ReactJS para implementar un modelo de estados en la aplicación.

Se ha decidido implementar un modelo de estados utilizando Redux para facilitar el desarrollo de la aplicación, y porque tiene numerosas ventajas para el usuario, desde el punto de vista de navegación web. Este modelo permite separar los componentes visuales de las acciones que se realizan, y así poder tener un grado de desacoplamiento entre los componentes pensado, nuevamente, para evitar duplicidades en el código.

6.2 Caso de uso

A continuación veremos un ejemplo de uso de la aplicación web, en el que generaremos un conjunto de datos relativo a todas las ciudades de España que hay almacenadas en Wikidata. Generaremos un grafo de conocimiento asociado, realizaremos un entrenamiento y finalmente ejecutaremos el servicio de predicción de tripletas similares y evaluaremos estos resultados.

Lo primero que haremos será hacer click en el botón inferior derecho para crear un nuevo conjunto de datos vacío, y rellenaremos el nombre y la descripción que queremos que tenga. En nuestro caso pondremos como título *Ciudades_ES* y como descripción *Ciudades de España*. En la figura 6.1 podemos ver el resultado. Hacemos click y veremos aparecer una nueva tarjeta en la página principal.

[2]Material Design guidelines: `https://material.io/guidelines/`

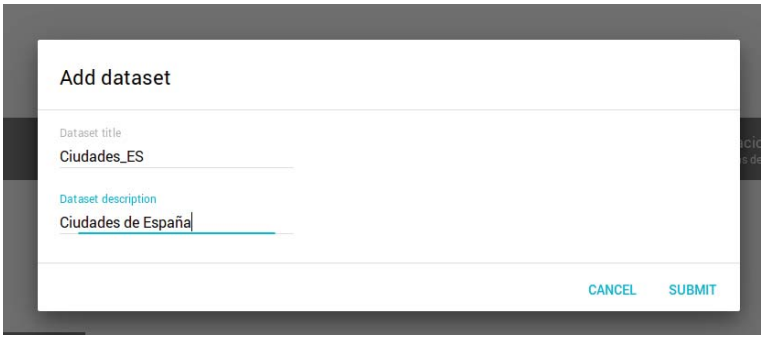

Figura 6.1: Añadir un dataset a la aplicación

Se creará un dataset vacío, sólo con el nombre y la descripción.

Si hacemos click en esa tarjeta podremos ver toda la información y acciones asociada en una nueva pantalla (figura 6.2). Esta es la página del *dataset*, donde podemos consultar estadísticas sobre nuestro dataset como de cuántas tripletas o entidades está compuesta o ver el progreso de una tarea que se esté ejecutando. También podemos ver en la parte de estado los servicios que tiene disponibles para utilizar.

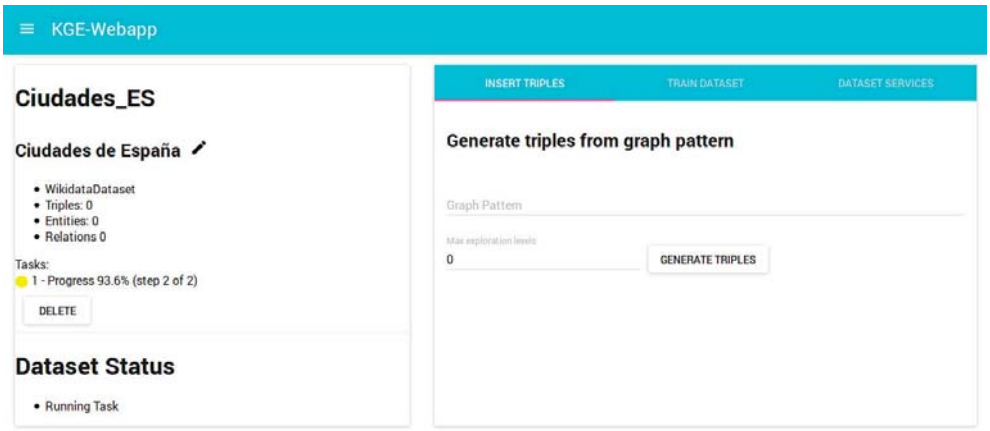

Figura 6.2: Vista principal de un dataset en la aplicación

Se puede ver que hay principalmente dos columnas. La izquierda, que indica el estado del dataset y algunas estadísticas, y la derecha, a partir de la cual podremos realizar operaciones con el conjunto de datos.

En este punto vemos que disponemos de una segunda columna, que tiene todas las operaciones disponibles con este *dataset* en concreto:

- Generar un grafo de conocimiento a través de un *graph_pattern*

- Realizar la generación de *embeddings*
- Utilizar el servicio de predicción de entidades similares

En este momento nos centraremos en generar el grafo necesario para nuestro ejemplo. Dado que el propósito es el mismo que en el punto 4.1.2, utilizaremos el mismo *graph_pattern*. Necesitaremos dar al algoritmo otro parámetro extra, el número de niveles que deseamos generar, que lo pondremos a dos, ya que obtenemos una profundidad adecuada para el propósito que deseamos sin realizar un número muy elevado de consultas. Podemos ver cómo quedaría en la figura 6.3

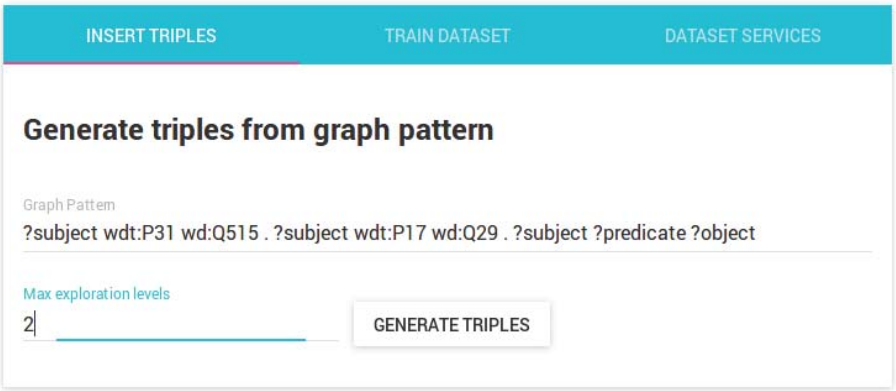

Figura 6.3: Generar grafo de conocimiento a partir de un *graph_pattern*

Veremos que en ese momento el estado del dataset cambia a *amarillo*, puesto que hay una tarea en ejecución. Cuando acabe podremos ir a la siguiente pestaña, donde configuraremos nuestro algoritmo de generación de *embeddings* y lo pondremos en marcha (figura 6.4). Puesto que es la primera vez que lo ejecutaremos, deberemos crear un nuevo conjunto de parámetros. Presionaremos el botón de *Add Algorithm* y configuraremos los parámetros siguientes: vectores de *embedding* de 100 dimensiones, hasta 500 iteraciones y un margen con valor 2. Nuevamente pondremos una tarea en ejecución y deberemos de esperar hasta su finalización. Una vez acabada, deberemos poner a ejecutar también el proceso de generación de índice y la configuración del autocompletado.

Figura 6.4: Crear nuevo algoritmo de entrenamiento

Tras realizar esta serie de pasos ya tendremos listo nuestro conjunto de datos para poder ejecutar el servicio de similitud de entidades. En la parte superior se muestra una tarjeta con la entidad que habremos escogido a través del cuadro de búsqueda de abajo, gracias al autocompletado, como se puede ver en la figura 6.5. En este ejemplo probaremos introduciendo *Madrid* en el cuadro de búsqueda y seleccionando la que corresponde a la capital española (identificador en Wikidata Q2807). Al pulsar el botón, podremos ver en la parte inferior (figura 6.6) las entidades más similares a Madrid.

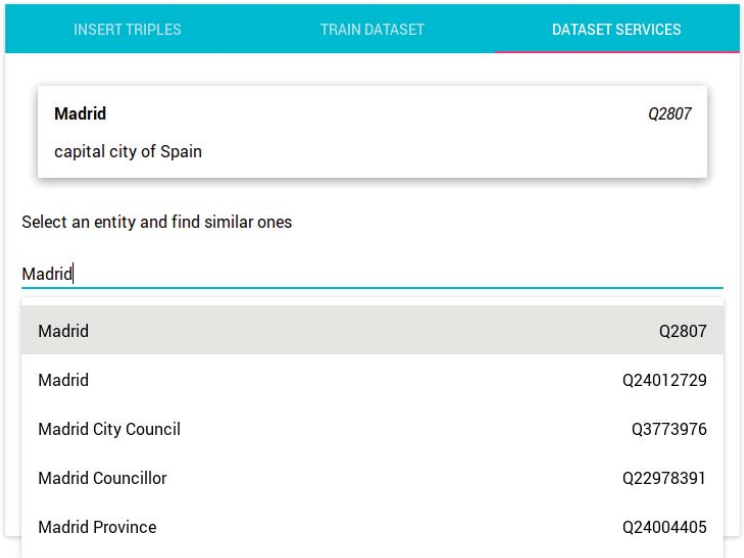

Figura 6.5: Ventana de autocompletado

En esta figura se puede ver el cuadro de autocompletado, y la entidad que con la que se quieren buscar similitudes arriba, una vez escogida la entidad del desplegable.

En la figura 6.6 vemos las cuatro primeras entidades que nos ha seleccionado el al-

goritmo como *más similares*. Como se puede observar, son todo del mismo tipo, ciudades, pero comparten que todas ellas son capitales de algún territorio y poseen muchos habitantes. Los dos primeros resultados son similares probablemente debido a que son capitales de las dos comunidades autónomas más pobladas de España. Aparecen después dos ciudades más, la capital de Alemania, Berlín, y la capital de República Dominicana, Santo Domingo, probablemente por el hecho de ser capitales de sus respectivos países.

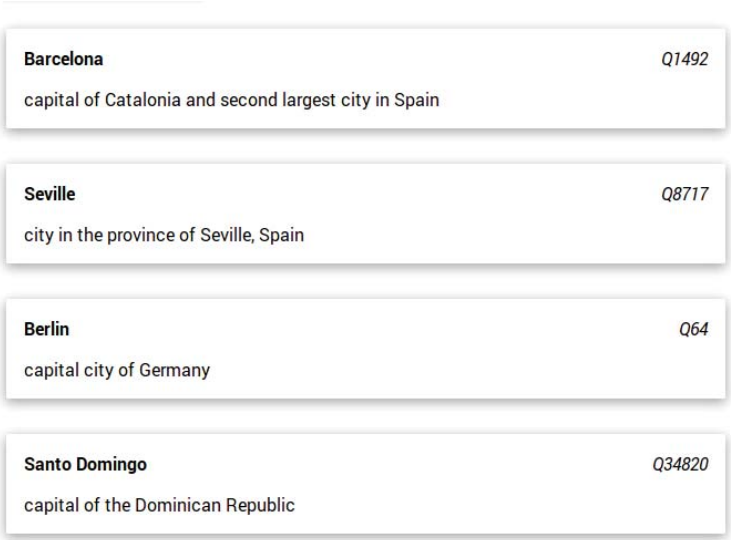

Figura 6.6: Entidades similares a Madrid

Probaremos otra ejecución con el mismo conjunto de datos, esta vez, para obtener las entidades más parecidos a Estados Unidos (figura 6.7). Es importante destacar aquí que aunque en el *graph_pattern* inicial no figuraba esta entidad, ha sido incluida en el grafo de conocimiento gracias al encadenamiento de consultas explicado en la sección 4.1.1. De esta forma, el conjunto total de entidades que lo forman es mucho mayor que el original y las consultas que se ejecutan pueden devolver resultados mucho más ricos.

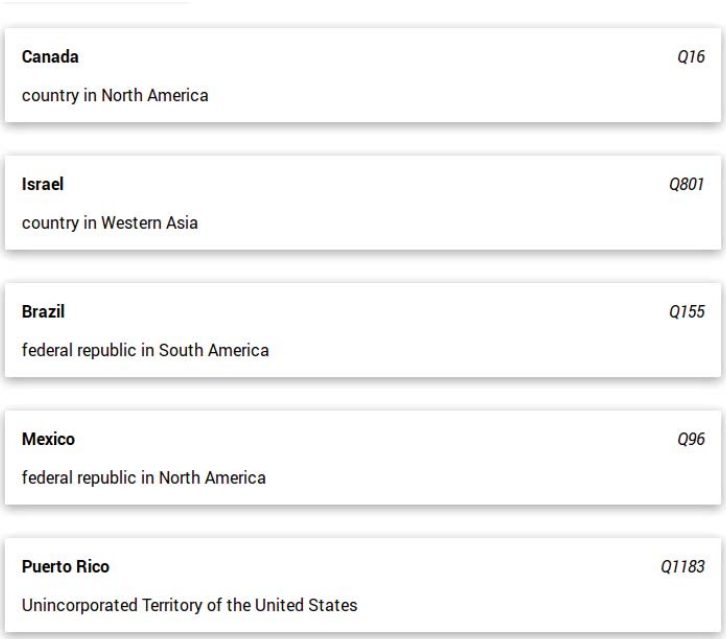

Figura 6.7: Entidades similares a Estados Unidos

Nuevamente vemos que los resultados son bastante buenos. De esta lista de 5 resultados, cuatro de ellos son países que están en América, y salen dos de los países que hacen frontera con él, México y Canadá. En este *ranking* también aparece Puerto Rico, que se trata de un estado libre asociado a Estados Unidos.

7 Conclusión

A lo largo del trabajo se ha presentado un trabajo bastante completo que abarca muchas áreas de la informática, desde la Inteligencia Artifical hasta la administración de sistemas, pasando por la Ingeniería de Software. Es un trabajo que permite la utilización de las últimas técnicas en el campo de la inteligencia artificial, como son los *embeddings* sin necesidad de instalar y configurar todos los módulos que son requeridos en ocasiones para experimentar con este tipo de técnicas. Permite, asimismo, la explotación del conocimiento de grandes bases de datos expresadas en RDF y consultables con SPARQL de una forma cómoda.

Los resultados, además, son bastante alentadores, ya que sin construir un modelo óptimo para cada conjunto de datos y aplicando una configuración básica se obtienen resultados buenos, o, que al menos, se comportan dentro de los límites del sentido común. Se pueden conseguir distintos resultados en la similitud de entidades modificando los parámetros en tiempo de entrenamiento y es recomendable hacerlo para encontrar cuáles son los parámetros que obtienen los mejores resultados.

Gracias a que el código de la mayor parte del sistema está publicado como software libre, se facilita el acceso a estas técnicas y se promueve el trabajo colaborativo y abierto en beneficio de la comunidad.

7.1 Trabajos futuros

A pesar de que se presenta un trabajo muy completo, en el campo de la informática pocas veces se puede dar un trabajo como completamente finalizado, pues siempre es posible mejorar el producto, ya sea desde el punto de vista del rendimiento computacional, de la mejora de los algoritmos utilizados para aumentar la precisión o los cambios a nivel de diseño que pueda sufrir la aplicación web para adaptarla más al usuario final.

Desde el punto de vista del campo de la inteligencia artificial, el software aquí utilizado genera modelos, pero no es posible evaluarlos, puesto que debido a la magnitud que tienen, toman mucho tiempo para hacerlo. Se debería investigar alguna técnica capaz de

llevar a cabo esta evaluación en un tiempo más razonable, o intentar analizar el software para poder ver qué mejoras se pueden llevar a cabo.

El tiempo de computación de los *embeddings* suele ser bastante elevado y además crece exponencialmente a medida que se agregan entidades y relaciones. Para aplicaciones en tiempo real que manejen grandes cantidades de datos, este procesamiento no podrá ser hecho instantáneamente, y estas técnicas pierden interés. Por eso, se podrían revisar los algoritmos utilizados y utilizar otros que se basen realmente en modelos de computación distribuida.

Como se ha mencionado, en este trabajo sólo nos hemos centrado finalmente en un único algoritmo, TransE, pero como se ha visto en el estado del arte, hay otros muchos que se han quedado sin probar. También, desde el punto de vista de las propiedades que tienen los *embeddings*, y tomando las ideas que ya se llevan a cabo en proyectos como Word2vec [15], podría analizarse si alguna de estas ideas se puede trasladar al campo de los grafos de conocimiento.

A Métodos HTTP REST del servicio KGE-Server

En este anexo se detallarán todos los métodos HTTP que utiliza el servicio KGE-Server, que se pueden conocer detalles sobre la implementación en el punto 5. Dado que la documentación de este trabajo se ha realizado como parte de un proyecto de software libre y el propósito es que sea consultada por cualquier persona sin que la barrera lingüística suponga un problema, está escrita en inglés.

El servicio se compone principalmente de las siguientes colecciones:

Dataset La colección `/dataset/` es probablemente la más importante en el servicio. Realiza las labores de gestión de dataset y permite ejecutar diferentes tareas y servicios de predicción sobre un conjunto de datos determinado.

Algorithm Esta colección permite gestionar los distintos algoritmos que se usarán para entrenar los modelos predictivos y de generación de vectores de *embedding*.

Tasks Debido al enfoque *asíncrono* que se ha dado a la ejecución de las tareas que más tiempo tardan en ejecutar, esta colección almacena información sobre la ejecución de dicha tarea, e informa de si está en ejecución, si ha finalizado correctamente o con errores.

Los datos que se envíen en el cuerpo de la petición deberán estar serializados como JSON, y deben proporcionar la cabecera indicándolo adecuadamente: `Content-type: application/json`. De la misma forma, todas las respuestas que proporcione el servidor serán serializadas como JSON y vendrán señaladas con la misma cabecera.

A.1 Administración de conjuntos de datos

Esta sección contiene todos los métodos que crean o alteran conjuntos de datos.

GET /datasets/

Get all datasets available on the system

Response


```
[
    {
        "dataset": {
            "relations": 536,
            "name": "animacion2",
            "status": 6,
            "dataset_type": "WikidataDataset",
            "task": 7,
            "algorithm": {
                "embedding_size": 10,
                "max_epochs": 100,
                "margin": 2.0,
                "id": 2
            },
            "triples": 370334,
            "id": 4,
            "entities": 126526,
            "description": "Peliculas de animacion_maspeq"
        }
    }
]
```


POST /datasets/

Creates a new and empty dataset. To fill in you must use other requests.

You also must provide dataset_type query param. This method will create a WikidataDataset (id: 1) by default, but you also can create different datasets providing a different dataset_type.

Inside the body of the request you can provide a name and/or a description for the dataset. The name must be unique. For example:

Request

POST `/datasets/?dataset_type=1`

Response The location: header of the response will contain the relative URI for the created dataset. Additionally, the body of the response will contain a dataset object with only id argument filled in:

```
location: /datasets/32
```

```
{
    "dataset": {
        "id": 32
    }
}
```


GET /datasets/: *dataset_id*

Get all the information about a dataset, given a *dataset_id*

Request

POST /datasets/1/

Response


```
{
    "dataset": {
            "relations": 655,
            "triples": 3307248,
            "algorithm": {
                    "id": 2,
                    "embedding_size": 100,
                    "max_epochs": null,
                    "margin": 2
            },
            "entities": 651759,
            "status": 2,
            "name": "Films",
            "id": 1
    }
}
```


PUT /datasets/: *dataset_id*

Edits the description from a existing dataset.

Request

PUT /datasets/1


```
{"description": "A dataset with most awarded films"}
```


Response


```
{
    "dataset": {
            "relations": 655,
            "triples": 3307248,
```

```
            "algorithm": {
                    "id": 2,
                    "embedding_size": 100,
                    "max_epochs": null,
                    "margin": 2
            },
            "entities": 651759,
            "status": 2,
            "name": "Films",
            "description": "A dataset with most awarded films",
            "id": 1
    }
}
```


A.2 Operar con los conjuntos de datos

POST `/datasets/`: *dataset_id*`/train`

Train a dataset with a given algorithm id. See `/algorithm` collection to learn how to manage them. The training process can be quite large, so this REST method uses a asynchronous model to perform each request.

The response of this method will only be a `202 ACCEPTED` status code, with the `Location:` header filled with the task path element. See `/tasks` collection to get more information about how tasks are managed on the service.

The dataset must be in a 'untrained' (0) state to get this operation done. Also, no operation such as `add_triples` must be being processed. Otherwise, a `409 CONFLICT` status code will be obtained.

Request

POST `/datasets/`: *dataset_id*`/train?algorithm=1`

Response


```
Location: /tasks/16
```

```
{
  "message": "Task 16 created successfuly",
  "status": 202,
    "task": {
        "id": 16
    }
}
```

POST /dataset/: *dataset_id*/triples

Adds a triple or a list of triples to the dataset. You must provide a JSON object on the request body, as shown below on the example. The name of the JSON object must be triples and must contain a list of all entities to be introduced inside the dataset. These entities must contain `"subject", "predicate", "object"` params. This notation is similar to other known as head, label and tail.

Only triples can be added on a untrained (0) dataset.

Request

POST /datasets/: *dataset_id*/triples

```
{"triples": [
        {
            "subject": {"value": "Q1492"},
            "predicate": {"value": "P17"},
            "object": {"value": "Q29"}
        },
        {
            "subject": {"value": "Q2807"},
            "predicate": {"value": "P17"},
            "object": {"value": "Q29"}
        }
    ]
}
```

POST /datasets/: *dataset_id*/embeddings

Retrieve from the trained dataset the embeddings from a list of entities.

If on the request list the user requests for a entity that does not exist, the response won't contain that element. The 404 error is limited to the dataset, not the entities inside the dataset.

The dataset must be in trained status (>= 1), because a model must exist to extract triples from. If not, a `409 CONFLICT` will be returned.

This could be useful if it is used with `/similar_entities` endpoint, to find similar entities given a different embedding vector.

Request

POST /datasets/4/embeddings

```
{"entities": [
    "http://www.wikidata.org/entity/Q1492",
    "http://www.wikidata.org/entity/Q2807",
    "http://www.wikidata.org/entity/Q1" ]
```

```
}
```

Response *Note: The shown vectors are only shown as illustrative, they are not real values*

```
{ "embeddings": [
    [
        "Q1",
        [0.321, -0.178, 0.195, 0.816]
    ],
    [
        "Q2807",
        [-0.192, 0.172, -0.124, 0.138]
    ],
    [
        "Q1492",
        [0.238, -0.941, 0.116, -0.518]
    ]
  ]
}
```

POST `/datasets/`: *dataset_id*`/generate_triples`

Adds triples to dataset doing a sequence of SPARQL queries by levels, starting with a seed vector. This operation is supported only by certain types of datasets (the default one, type=1)

The request will use asyncronous operations. This means that the request will not be satisfied on the same HTTP connection. Instead, the service will return a `/task` resource that will be queried with the progress of the task.

The `graph_pattern` argument must be the where part of a SPARQL query. It must contain three variables named as `?subject`, `?predicate` and `?object`. The service will try to make a query with these names.

You also must provide the levels to make a deep lookup of the entities retrieved from previous queries.

The optional param `batch_size` is used on the first lookup for SPARQL query. For big queries you must tweak this parameter to avoid server errors as well as to increase performance. It is the LIMIT statement when doing this queries.

Request

```
{
    "generate_triples":
        {
```

```
            "graph_pattern": "SPARQL Query",
            "levels": 2,
            "batch_size": 30000
        }
}
```

Response

```
location: /tasks/32
```

```
{
    "message": "Task 32 created successfuly",
    "status": 202,
    "task": {
        "id": 32
    }
}
```

POST /datasets/: *dataset_id*/generate_autocomplete_index

Creates a task to build an autocomplete index

The task will perform a request to SPARQL endpoint for each entity. This will extract the labels, description and altLabels and store it on an Elasticsearch database.

It is also possible give the languages desired to build the autocomplete index, allowing not only having english language, but others available on the endpoint. You must specify in the body a param named langs with a list with all language codes in ISO 639-1 format.

Request

POST /datasets/6/generate_autocomplete_index

```
{
    "langs" :  [
        "en", "es"
    ]
}
```

Response

```
{
    "status": 202,
    "message": "Task 73 created successfuly",
    "task": {
        "id": 73
    }
}
```

A.3 Servicio de predicción de los conjuntos de datos

POST /datasets/: $dataset_id$/similar_entities

Get the limit entities most similar to a entity inside a dataset_id. The given number in limit excludes the entity given itself.

The POST method allows any representation of the wanted resource. See the example below. You can provide an entity as an URI or other similar representation, even an embedding. The type param inside entity JSON object must be "uri" for a URI or similar representation and "embedding" for an embedding.

The search_k param is used to tweak the results of the search. When this value is greater, the precission of the results are also greater, but the time it takes to find the response is also bigger.

Request

POST /datasets/4/similar_entities?limit=1&search_k=10000

```
{ "entity":
  {"value": "http://www.wikidata.org/entity/Q1492", "type": "uri"}
}
```

Response

```
{    "similar_entities":
     {    "response":
          [
               {"distance": 0, "entity": "http://www.wikidata.org/
entity/Q1492"},
               {"distance": 0.8224636912345886, "entity": "http://www.
wikidata.org/entity/Q15090"}
          ],
          "entity": "http://www.wikidata.org/entity/Q1492",
          "limit": 2
     },
     "dataset": {
          "entities": 664444,
          "relations": 647,
          "id": 1,
          "status": 2,
          "triples": 3261785,
          "algorithm": 100
     }
}
```

POST /datasets/: $dataset_id$/distance

Returns the distance between two elements. The lower the number is, most probable to be both the same triple. The minimum distance is 0.

Request

POST /datasets/7/distance

```
{
    "distance": [
        "http://www.wikidata.org/entity/Q1492",
        "http://www.wikidata.org/entity/Q5682"
    ]
}
```

Response

```
{
    "distance": 1.460597038269043
}
```

POST /datasets/: $dataset_id$/suggest_name

Gives a list of autocomplete suggestions. For each entity, this will show labels on every language available, descriptions and altLabels.

If any suggestion is available, this will return an empty list.

Request

POST /datasets/7/suggest_name

```
{
    "input": "human"
}
```

Response

```
[
    {
        "text": "humano",
        "entity": {
            "alt_label": {
                "es": [
                    "humano",
                    "Homo sapiens sapiens",
                    "persona",
                ],
                "en": [
                    "people",
```

```
                    "person",
                    "human being"
                ]
            },
            "label": {
                "en": "human",
                "es": "ser humano",
            },
            "entity": "Q5",
            "description": {
                "en": "common name of Homo sapiens (Q15978631),
unique extant species of the genus Homo",
                "es": "especie animal perteneciente a la familia
Hominidae, unica superviviente del genero Homo",
            }
        }
    }
]
```


A.4 Gestión de algoritmos de entrenamiento

The algorithm collection is used mainly to create and see the different algorithms created on the server. The hyperparameters that are allowed currently to tweak are:

embedding_size The size of the embeddigs the trainer will use

margin The margin used on the trainer

max_epochs The maximum number of iterations of the algorithm

GET /algorithms/

Gets a list with all the algorithms created on the service.

Request

GET /algorithms/

Response


```
[
  {
    "id": 1,
    "max_epochs": 100,
    "margin": 2.0,
    "embedding_size": 50
```

```
    },
      {
          "id": 2,
          "max_epochs": 500,
          "margin": 1.4,
          "embedding_size": 100
      }
]
```

GET /algorithm/: *algorithm_id*

Gets only one algorithm

Request

GET /algorithms/2

Response

```
    {
        "id": 2,
        "max_epochs": 500,
        "margin": 1.4,
        "embedding_size": 100
    }
```

POST /algorithms/

Create one algorithm on the service. On success, this method will return a 201 CREATED status code and the header parameter Location: filled with the relative path to the created resource. The body of the request must contain all parameters for the new algorithm. See the example below:

Request

POST /algorithm/

```
{
    "algorithm": {
            "embedding_size": 50,
            "margin": 2,
            "max_epochs": 80
    }
}
```

Response The response when creating a new algorithm gives the location header filled with the URI of the new resource. It also returns the HTTP 201 status code, and the body has information about the request in json format.

```
location: /algorithm/3
```

```
{
    "status": 201,
    "algorithm": {
        "id": 3
    },
    "message": "Algorithm 3 created successfuly"
}
```

A.5 Gestión de tareas asíncronas

The task collection stores all the information that async request need. This collection are made mainly to get the actual state of tasks, but no to edit or delete tasks.

GET `/tasks/`: *task_id*

Shows the progress of a task with a `task_id`. The finished tasks can be deleted from the system without previous advise.

Some tasks can inform to the user about its progress. It is done through the progress param, which has current and total relative arguments, and `current_steps` and `total_steps` absolute arguments. When a task involves some steps and the number of small tasks to be done in next step cannot be discovered, the current and total will only indicate progress in current step, and will not include previous step, expected to be already done, or next step which is expected to be empty.

The resource has two optional parameters: `get_debug_info` and `no_redirect`. The first one, `get_debug_info` set to true on the query params will return additional information from the task. The other param, `no_redirect` will avoid send a 303 status to the client to redirect to the created resource. Instead it will send a simple 200 status code, but with the location header filled.

Request

GET `/tasks/12`

Response

```
{
  "task": {
    "state": "STARTED",
    "progress": {
      "current": 499,
      "total_steps": 2,
```

```
      "total": 13467,
      "current_steps": 1
    },
    "id": 12
  }
}
```

8 Bibliografía

[1] T. Berners-Lee, J. Hender y O. Lassila, «The Semantic Web», *Scientific American*, vol. 284, n.º 5, págs. 34-43, mayo de 2001.

[2] T. R. Gruber, «A translation approach to portable ontology specifications», *Knowledge Acquisition*, vol. 5, n.º 2, págs. 199-220, 1993.

[3] R. Studer, V. R. Benjamins y D. Fensel, «Knowledge engineering: Principles and methods», *Data Knowl. Eng*, vol. 25, n.º 1-2, págs. 161-197, 1998.

[4] M. Nickel, K. Murphy, V. Tresp y E. Gabrilovich, «A review of relational machine learning for knowledge graphs», *ArXiv e-prints*, mar. de 2015. arXiv: 1503.00759 [stat.ML].

[5] M. Nickel, V. Tresp y H.-p. Kriegel, «A three-way model for collective learning on multi-relational data», en *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, L. Getoor y T. Scheffer, eds., New York, NY, USA: ACM, 2011, págs. 809-816. dirección: http://www.icml-2011.org/papers/438_icmlpaper.pdf.

[6] A. Bordes, J. Weston, R. Collobert y Y. Bengio, «Learning structured embeddings of knowledge bases», en *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, ép. AAAI'11, San Francisco, California: AAAI Press, 2011, págs. 301-306. dirección: http://dl.acm.org/citation.cfm?id=2900423.2900470.

[7] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston y O. Yakhnenko, «Translating embeddings for modeling multi-relational data», en *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani y K. Q. Weinberger, eds., Curran Associates, Inc., 2013, págs. 2787-2795. dirección: http://papers.nips.cc/paper/5071-translating-embeddings-for-modeling-multi-relational-data.pdf.

[8] M. Nickel, L. Rosasco y T. A. Poggio, «Holographic embeddings of knowledge graphs», *CoRR*, vol. abs/1510.04935, 1 de nov. de 2015. dirección: http://arxiv.org/abs/1510.04935.

[9] M. Nickel, *Knowledge graph embeddings*, feb. de 2016. dirección: https://github.com/mnick/scikit-kge.

[10] Y. Lin, Z. Liu, M. Sun, Y. Liu y X. Zhu, «Learning entity and relation embeddings for knowledge graph completion», en *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, ép. AAAI'15, Austin, Texas: AAAI Press, 2015, págs. 2181-2187, ISBN: 0-262-51129-0. dirección: http://dl.acm.org/citation.cfm?id=2886521.2886624.

[11] B. Shi y T. Weninger, «Proje: Embedding projection for knowledge graph completion», *CoRR*, vol. abs/1611.05425, 2016. dirección: http://arxiv.org/abs/1611.05425.

[12] E. Bernhardsson, *Annoy (approximate nearest neighbors oh yeah)*, 2013. dirección: https://github.com/spotify/annoy.

[13] S. Allamaraju, *RESTful Web Services Cookbook*. O'Reilly, 2010, págs. 19-22, ISBN: 978-0-596-80168-7. dirección: http://www.oreilly.de/catalog/9780596801687/index.html.

[14] C. project, *Celery: Distributed task queue*, 2016. dirección: https://github.com/celery/celery/.

[15] T. Mikolov, K. Chen, G. Corrado y J. Dean, «Efficient estimation of word representations in vector space», *CoRR*, vol. abs/1301.3781, 2013. dirección: http://arxiv.org/abs/1301.3781.

Este documento esta firmado por

Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
Fecha/Hora	Thu Jun 08 22:25:14 CEST 2017
Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
Numero de Serie	630
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)