Universidad Zaragoza
1542

# Trabajo Fin de Grado

## Procesamiento de lenguaje natural aplicado a Twitter para relacionar usuarios por sus intereses

## Natural language processing applied to Twitter to relate users by their interests


Autor

David Puente Mur

Director

José Ramón Gállego Martínez

Escuela de Ingeniería y Arquitectura
2018

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

# DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

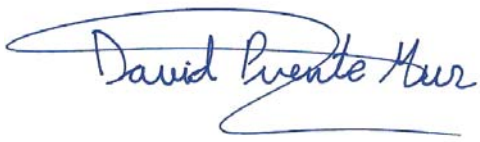
D./Dª. David Puente Mur,

con nº de DNI 18172258G en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Procesamiento de lenguaje natural aplicado a Twitter para relacionar usuarios por sus intereses

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 25 de Junio de 2018



Fdo: David Puente Mur

# Procesamiento de lenguaje natural aplicado a Twitter para relacionar usuarios por sus intereses

## Resumen

El objetivo de este proyecto consiste en tomar la información que se genera en las redes sociales y transformarlo en una herramienta que pueda ser de utilidad. Este objetivo podría aplicarse a otras redes sociales y con otras finalidades, pero en este proyecto nos hemos centrado en los usuarios de Twitter y sus intereses. Lo que se ha hecho es tomar una muestra de usuarios cualesquiera, e inferir relaciones entre ellos en base a sus intereses.

Para ello se han recolectado los tweets de dichos usuarios mediante la API de Twitter y se han procesado con un extractor de entidades como el de Google Cloud. Con estas entidades y mediante un proceso de filtrado se han obtenido los intereses de los usuarios con unos determinados pesos. Una vez identificados sus intereses se ha procedido a relacionarlos tanto directamente como indirectamente, para lo cual se ha usado Wikidata como una ontología que nos ha permitido relacionar los intereses entre sí. Con todas estas relaciones y pesos se ha desarrollado un algoritmo que calcule la afinidad entre 2 usuarios y por tanto poder cruzarlos todos entre ellos. Finalmente para visualizarlo de forma más intuitiva y poder manipular fácilmente toda esta información, se han volcado todas las relaciones, afinidades, intereses y usuarios sobre un servidor de grafos Neo4j.

# Natural language processing applied to Twitter to relate users by their interests

## Abstract

The purpose of this project is to take the information generated in social networks and transform it into a tool that can be useful. This purpose could be applied to other social networks and for other purposes, but in this project we have focused on Twitter users and their interests. What has been done is to take a sample of any users, and infer relations between them based on their interests.

For this, the tweets of these users have been collected through the Twitter API and processed with an extractor of entities such as Google Cloud. With these entities and through a filtering process, the interests of users with certain weights have been obtained. Once their interests have been identified, they have been related directly and indirectly, for which Wikidata has been used as an ontology that has allowed us to relate the interests to each other. With all these relationships and weights an algorithm has been developed to calculate the affinity between 2 users and therefore can cross them all between them. Finally, to visualize it in a more intuitive way and to easily manipulate all this information, all the relationships, affinities, interests and users have been dumped on a Neo4j graph server.

# Índice

# 1 Introducción

## 1.1 Motivación y objetivos

Hoy en día, en un mundo en el que la tecnología ha llegado al bolsillo de todos, y todo el mundo tiene Internet al alcance de su mano, son muchas las personas que pasan gran parte de su tiempo relacionándose a través de las redes sociales, donde se va acumulando una gran cantidad de información que tratada de forma adecuada puede ser muy valiosa. Con este proyecto se quiere demostrar la viabilidad de convertir toda esa información en una herramienta con la que extraer conclusiones y poder utilizarlo para tomar decisiones más fácilmente en un determinado contexto.

La idea de este proyecto surge gracias a los conocimientos adquiridos durante las prácticas realizadas en el departamento de inteligencia artificial de Everis Aragón, donde además de descubrir el potencial que puede llegar a tener la información, se ha participado en proyectos interesantes desempeñando tareas bastante enriquecedoras. Uno de estos proyectos consistía en analizar de forma automática una gran cantidad de comentarios resultantes de unas encuestas. Para ello utilizamos entre otras cosas los servicios de Amazon Web Services (AWS), que nos permitían identificar las palabras clave y sentimientos asociados a los comentarios. Durante este proyecto aprendí tanto las capacidades que tienen los servicios en la nube como AWS, a usarlos, y a manipular y estructurar la información. Posteriormente también participé en otro proyecto para convertir el texto en conocimiento. Una de las partes consistía en adquirir una gran ontología ya existente y tratarla para usar las relaciones extraídas en etapas posteriores. Una de las tareas que desempeñé en este proceso, fue mirar documentación sobre las grandes wikis, tales como Wikidata y DBpedia, aprender cómo está estructurada la información, y desarrollar algoritmos para procesarla.

Con las tareas realizadas, me di cuenta de que haciendo uso de grandes herramientas como las mencionadas, se podían hacer grandes cosas con pocos recursos. Por tanto ahora me parecía viable la idea de procesar las redes sociales para extraer información útil de toda esa información aparentemente inabarcable.

Puesto que el proceso ya es de por sí laborioso, en este proyecto se ha hecho algo de carácter general, ya que hay que recopilar la información de Twitter, hacer el procesado de la misma para identificar los intereses de los usuarios, tratar la base de datos de Wikidata para encontrar relaciones entre los intereses, desarrollar un algoritmo para calcular la afinidad entre usuarios, y representar en forma de grafos las relaciones que nos han llevado a ello.

Todo el código desarrollado puede encontrarse publicado en GitHub para poder seguirlo si se quiere conforme se lee la memoria [16].

## 1.2 Herramientas y plataformas utilizadas

Se ha utilizado Python como lenguaje de programación, Jupyter Notebook como entorno de desarrollo y Anaconda como gestor de entornos, por su versatilidad, librerías existentes y facilidad para la manipulación de datos. En el *anexo 2* puede encontrarse todo lo relativo al entorno. Como ya se ha dicho, se ha tomado como fuente de datos la red social Twitter, y para ello ya te ofrecen en su plataforma de desarrolladores una API con la que interactuar fácilmente con sus servicios y que se ha usado en particular para obtener los tweets de los usuarios de interés. Luego para procesar los tweets se ha utilizado un servicio de extracción de entidades ya existente como el que ofrecen Google Cloud y Amazon Web Services. Para relacionar términos entre sí se ha utilizado Wikidata como ontología, ya que supera los 40 millones de entradas frente a los más de 4 millones de DBpedia, es multilingüe por construcción y a diferencia de Wikipedia contiene los campos básicos. Para la representación, almacenamiento y búsqueda de los resultados finales se ha utilizado un servidor local de Neo4j ya que es una base de datos orientada a grafos.

## 1.3 Descripción general y planificación

Tal como puede verse en la siguiente figura, el proceso consta de 4 grandes bloques, en cada uno de ellos se hace uso de una tecnología distinta, por lo que cada parte ha requerido de una investigación previa, tanto para el aprendizaje como la consulta de las capacidades, limitaciones, y costes que puede suponer su uso.

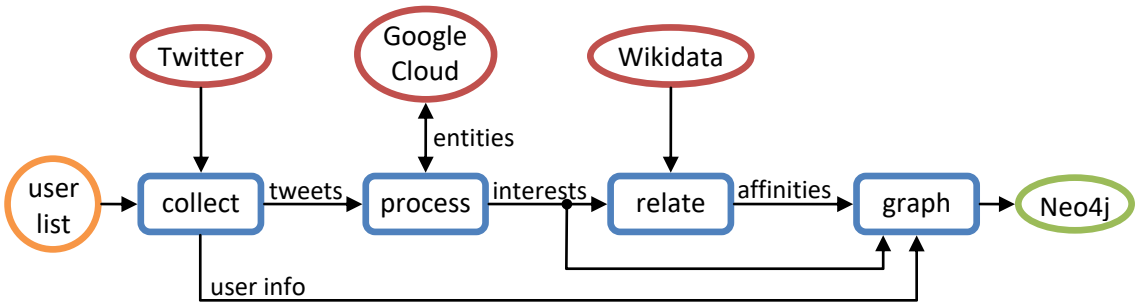



Figura 1. Diagrama de bloques del proyecto

Se parte de una lista de usuarios que se quieren analizar, donde la primera etapa se encarga de recolectar con la API de Twitter sus tweets además de información básica del usuario tal como el nombre completo, número de seguidores,… que posteriormente será de utilidad para filtrar los resultados con ello.

Los tweets recopilados pasan a la etapa de procesado, donde se utiliza el extractor de entidades elegido (hay que decidir entre Google Cloud y Amazon Web Services según el que se adapte mejor a nuestras necesidades) y se hace un filtrado de ello para identificar los intereses de los usuarios.

Una vez tenemos los intereses identificados, hay que relacionar a los usuarios en base a ello, para lo que se utiliza Wikidata como ontología para sacar relaciones entre los intereses. Con todo ello se ha desarrollado un algoritmo que con los pesos, relaciones directas e indirectas acaba traduciendo esa relación entre usuarios en un valor numérico al que hemos llamado afinidad.

Finalmente se ha volcado todo ello sobre un servidor Neo4j con el que poder visualizar de forma gráfica todas esas relaciones en forma de grafos, pudiendo ver así qué es lo que tienen en común los usuarios para haber llegado a calcular esa afinidad y por supuesto hacer cualquier búsqueda sobre ello que nos pueda ser de utilidad.

Dado que cada etapa depende de los resultados de la anterior, su desarrollo en el tiempo tendrá que seguir el mismo orden, y como cada una de ellas supone un ciclo de desarrollo entero, se ha pretendido repartir en principio el tiempo de forma equitativa. De igual forma se va a proceder para la redacción de esta memoria, dividiendo así el desarrollo del proyecto en 4 grandes etapas, explicando a su vez las partes de cada una. A continuación se muestra un diagrama de Gantt con la distribución temporal que finalmente se ha llevado a cabo.

| Tarea / Semana | 12 mar | 19 mar | 26 mar | 02 abr | 09 abr | 16 abr | 23 abr | 30 abr | 07 may | 14 may | 21 may | 28 may | 04 jun | 11 jun | 18 jun | 25 jun | Total (h) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Investigación API Twitter | | | | 3 | 6 | 8 | | | | | | | | | | | 17 |
| Recopilación de tweets | | | | | | | 28 | | | | | | | | | | 28 |
| Investigación Google Cloud | | | | | | | | 18 | | | | | | | | | 18 |
| Extracción de entidades | | | | | | | | 6 | 31 | | | | | | | | 37 |
| Identificación de intereses | | | | | | | | | | 35 | | | | | | | 35 |
| Investigación Wikidata | 7 | 6 | 4 | | | | | | | | | | | | | | 17 |
| Obtención de relaciones | | | | | | | | | | | 35 | 5 | | 3 | | | 43 |
| Cáculo de la afinidad | | | | | | | | | | | 6 | 36 | | | | | 42 |
| Investigación Neo4j | | | | | | | | | | | | | 41 | | | | 41 |
| Representación en grafos | | | | | | | | | | | | | | 32 | | | 32 |
| Elaboración de la memoria | | | | | | | | | | | | | | | 33 | 12 | 45 |
| Realización del proyecto | | | | | | | | | | | | | | | | | 355 |

Tabla 1. Diagrama de Gantt y distribución de horas

# 2 Desarrollo del proyecto

## 2.1 Etapa 1: Recolección de los tweets

### 2.1.1 Investigación previa

Para interactuar con los servicios de Twitter, este ofrece a los desarrolladores una API con una serie de llamadas y sus correspondientes parámetros de entrada y salida que facilitan la tarea del programador. Pero para utilizarla, antes es preciso saber qué hace falta y qué limitaciones tiene.

Como es común en este tipo de servicios, es preciso crearse una cuenta en la plataforma y obtener unas credenciales para identificarse a la hora de usar la API, de esta forma pueden regular el uso que hacemos de ella. En este caso podemos hacer uso de la API de forma gratuita pero con ciertas limitaciones tales como máxima tasa de peticiones o acceso a los tweets más antiguos, que pagando no tendríamos.

Consultando la documentación de Twitter [1] podemos ver que los límites se aplican en intervalos de 15 minutos, y que hay fijados distintos límites para cada tipo de petición. En particular, la petición que nos interesa a nosotros para extraer tweets de un determinado usuario [2], está limitada a 900 llamadas durante esa ventana de 15 minutos. Además en cada petición se puede pedir un máximo de 200 tweets.

Pero hay otra limitación muy importante a tener en cuenta, y es que para evitar abusos, Twitter solo permite extraer los últimos 3200 tweets de un usuario en particular. Lo que suele hacerse para lidiar con ello es ir recopilando los últimos 3200 tweets a lo largo del tiempo, llegando a acumular más de los que pueden obtenerse en un momento dado.

Se ha puesto esa técnica en práctica pero en el tiempo que ha durado este proyecto no es mucha más la cantidad que se ha llegado a acumular. En cualquier caso una muestra del orden de los 3000 tweets es más que suficiente para este propósito.

Otro detalle a tener en cuenta que está algo escondido en la documentación y que tuve que recurrir a terceros [3] para encontrar, es que tras la duplicación del límite de caracteres de los tweets, puedes encontrarte tanto con tweets anteriores de 140 caracteres como de los 280 actuales y resulta que por motivos de retrocompatibilidad, la API sigue devolviéndote un máximo de 140 caracteres, truncándote así los de 280. Entonces para evitar que los tweets se trunquen, hay que agregar el parámetro 'tweet_mode' a la petición con el valor 'extended' y será devuelto como 'full_text'.

Como ya se ha mencionado anteriormente, se ha utilizado Python para todo el desarrollo del proyecto, tanto por su agilidad a la hora de trabajar con etapas que se suceden (ya que con él se tiende a la programación funcional), como por su gran disponibilidad de librerías. En la propia documentación de la API de Twitter encontramos varias librerías recomendadas en Python [4], finalmente se ha escogido Tweepy [5] por su buena documentación y ejemplos.

## 2.1.2 Algoritmo de recopilación

El programa se ha desarrollado completamente en Python como ya se ha mencionado, y más concretamente en forma de notebook, que a grandes rasgos, se ha estructurado en 4 bloques o conjuntos de métodos, tal como se muestra en la *figura 2*, estando cada uno de ellos orientado a una funcionalidad. Estos serían: funciones que interactúan con la API de Twitter para obtener información, funciones encargadas de transformar la información obtenida y quedarse con lo relevante y en el formato deseado, funciones para guardar y recuperar datos de forma ordenada en local, y funciones de mayor abstracción que hacen uso de las anteriores para desempeñar la función final de esta etapa.

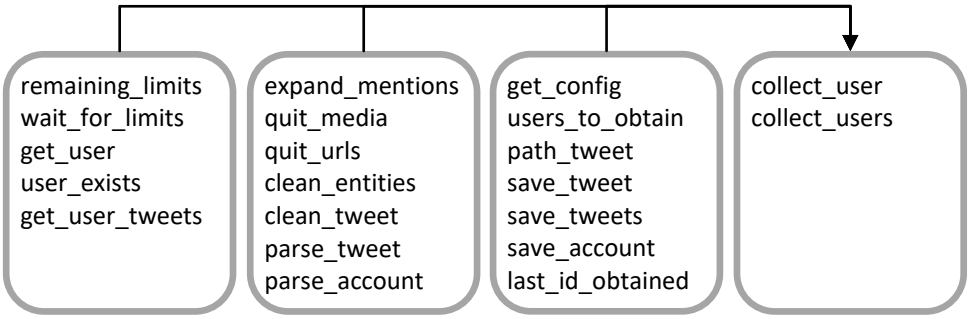



Figura 2. Bloques con las principales funciones de la etapa 1

Empezamos con las funciones para interactuar con la API de Twitter, donde se utiliza la librería Tweepy ya mencionada en el apartado de investigación previa.

Lo primero que se hizo fue un método para gestionar la limitación de 900 llamadas en 15 minutos de manera transparente al resto del programa. Lo que se hacía era consultar el número de peticiones restantes que nos quedan, y en caso de que fuera 0, esperar el tiempo que en esa misma petición nos indicaba que faltaba para el fin de la ventana de 15 minutos. A su vez también había un límite de 180 peticiones por ventana para consultar los límites, así que en vez de hacer esta comprobación en cada petición de tweets, se hacía solo al principio de la recolección de cada usuario, estimando un máximo de 18 peticiones restantes por cuenta.

¿Y de dónde sale el 18? Pues bien, como ya se ha visto en la parte de investigación previa, se pueden llegar a obtener unos 3200 tweets por cuenta (cantidad que no es exacta y normalmente es inferior por tweets que ya no existen, pero a veces es mayor), por tanto si el máximo de tweets por petición es de 200, nos salen en teoría un máximo de 16 peticiones por cuenta, que en la práctica han llegado a ser 17, y dejando 1 de margen, son las 18.

Además del método que se acaba de describir para consultar los límites restantes de la API, aquí es donde se realizan las peticiones de los tweets y además se comprueba si un determinado usuario existe. Hay que decir que no existe en la API una función como tal para verificar la existencia de un usuario dado, lo que se hace es pedir la información básica de ese usuario y controlando la excepción podemos saber si existe o no, y de paso almacenamos esa información para posteriores etapas. En cuanto al método para solicitar los tweets, es un bucle que hace peticiones de 200 tweets hasta recibir una respuesta con 0 tweets. Para solicitar los tweets de un determinado usuario se utiliza su nick, y se parte del id del último tweet obtenido en caso de tener alguno.

Las funciones de "*parseado*[1]" para transformar y adaptar la información obtenida, son la parte más importante de esta etapa, ya que es donde se decide qué guardar y cómo guardarlo de cara a las necesidades de las siguientes etapas. Y es que lo que nos devuelve Twitter no es solo el texto del tweet en sí, contiene otros muchos campos asociados al tweet tales como el id, la fecha de publicación, si es una respuesta o un retweet,… A continuación se muestra un ejemplo de la estructura original, pero la cantidad de información que contiene es tal, que se ha tenido que acotar a las partes más relevantes para poder incluirlo dentro de los límites de este documento. Puede verse un ejemplo de la estructura completa en la documentación de esta petición [2].


```
{
   "created_at": "Wed Apr 04 13:21:55 +0000 2018",
   "id": 981522253534941184,
   "full_text": "The @SpaceX #Dragon cargo vehicle was successfully installed on the
@Space_Station at 9:00am ET, delivering ~5,800 pounds of science and cargo to the
orbiting laboratory. Details: https://t.co/Y1NiDYzxls https://t.co/kt6GwtlBIo",
   "truncated": false,
   "entities": {
      "hashtags": [
         {"text": "Dragon", (...)}
      ],
      "symbols": [],
      "user_mentions": [
         {"screen_name": "SpaceX", "name": "SpaceX", (...)},
         {"screen_name": "Space_Station", "name": "Intl. Space Station", (...)}
      ],
      "urls": [
         {"url": "https://t.co/Y1NiDYzxls", (...)}
      ],
      "media": [
         {"url": "https://t.co/kt6GwtlBIo", "type": "photo", (...)}
      ]
   },
   "in_reply_to_status_id": null,
   "in_reply_to_user_id": null,
   "in_reply_to_screen_name": null,
   "user": {"id": 11348282, "name": "NASA", "screen_name": "NASA", (...)},
   "lang": "en",
   (...)
}
```


Figura 3. Estructura original de un tweet

Se ha elegido este tweet a propósito porque contiene varios elementos especiales: menciones, hashtag, enlace y contenido multimedia.

[1] "Parsear": proceso de analizar una estructura para convertirla al formato deseado.

Como no nos hacen falta la mayoría de los campos adicionales, lo primero que hacemos es seleccionar los que nos interesan o que podrían ser útiles más adelante. Estos campos son: el id del tweet, la fecha de publicación, el id, nick y nombre del usuario que lo ha publicado, si se trata de una respuesta o de un retweet, el idioma en el que está escrito y por supuesto el texto del contenido.

Pero el proceso de "parseado" no es tan sencillo como seleccionar los campos deseados de la estructura original, ya que si se trata de un retweet, la estructura original se complica. Resulta que cuando un usuario hace retweet de un tweet de otro usuario, en realidad se genera un tweet completamente nuevo con fecha e identificador distintos, e incluso figura el usuario que ha hecho el retweet como el usuario que lo ha publicado. Pero se conserva todo el tweet original incrustado dentro de este, donde está el texto original con todos sus metadatos originales, entre ellos los elementos especiales adjuntos y los datos del usuario que lo publicó originalmente.

Entonces, como lo que queremos es quedarnos con la información del usuario que estamos recopilando (ya que para ser coherentes necesitamos el id y fecha de este en vez de los del tweet original) y además queremos el texto y los elementos adjuntos del tweet original (porque ahí es donde sale sin truncar), tenemos que discernir cuando se trata de un retweet fijándonos en si existe el campo "retweeted_status" (donde se encuentra incrustado el tweet original) y extraer lo que nos interesa de donde corresponda, el contenido del tweet de "retweeted_status" si es el caso y sino de la estructura general, y el resto de datos siempre de la estructura general.

Como ya se ha dicho, el texto puede contener elementos especiales que conllevan información adjunta, y antes de desechar el objeto original vamos a tenerlos en cuenta. En el caso de las menciones, se ha hecho un método para expandirlas sustituyendo el nick por el nombre completo (en caso de que sea un personaje público podrá ser posteriormente identificado por su nombre real), y en el caso de los enlaces y contenido multimedia, lo quitamos directamente porque no nos aportan ninguna información de forma directa, mientras que los hashtags, al final se han conservado pese a haberlos borrado originalmente, ya que estos sí que pueden servir para relacionar usuarios. Y para terminar, antes de proceder al guardado de la información ya transformada, limpiamos tanto el texto del tweet como el del nombre de usuario por si contienen emojis que aparte de no aportar nada, pueden darnos problemas de codificación en las posteriores etapas. A continuación puede verse la estructura final.


```
{
   "retweet": false,
   "reply": false,
   "id": 981522253534941184,
   "date": "04-04-2018 13:21:55",
   "lang": "en",
   "text": "The SpaceX #Dragon cargo vehicle was successfully installed on the Intl.
Space Station at 9:00am ET, delivering ~5,800 pounds of science and cargo to the
orbiting laboratory. Details:",
   "user": {"id": 11348282, "nick": "NASA", "name": "NASA"}
}
```


Figura 4. Estructura final de un tweet

La otra parte menos importante pero igual de necesaria, es la de guardar y recuperar la información en local. El método más importante de esta parte sería el que se encarga de guardar los tweets con un formato y orden adecuados para el resto de funciones, para lo cual se hace uso a su vez de un método que genera las rutas.

Para ello se definió una estructura intuitiva y funcional para guardar y recuperar los tweets en local. Intuitiva porque se agrupan todos los tweets de usuario bajo una misma carpeta con el nombre de su nick, e identificando a los tweets por su fecha de creación, pero como la fecha no tiene por qué ser única, se extiende con el id que Twitter tiene del propio tweet. Además, aunque el nick sí que es único en Twitter, este puede ser cambiado por el usuario a lo largo del tiempo, así que se ha puesto como prefijo el id que utiliza Twitter para ello, ya que podría darse el caso de que un usuario libere un nick y posteriormente otro utilice ese mismo nick, llegando por tanto a mezclar tweets de distintas personas. Por tanto la ruta de acceso a un tweet sería la siguiente: "\[user id]@[user nick]\[tweet date]_[tweet id].json". De esta forma será muy sencillo recuperar todos los tweets de un determinado usuario, o saber cuál es el último tweet que tenemos recopilado de un usuario de cara a futuras recopilaciones.

También hay otro método similar para guardar la información básica de usuario siguiendo el mismo formato de "id@nick", el método para obtener la lista de usuarios a procesar del fichero "users.list", y el método para recuperar del fichero "config.json" las credenciales para usar la API.

Para finalizar esta etapa, nos queda el bloque de funciones de mayor abstracción, donde se hace uso de las funciones ya explicadas para realizar el cometido de recopilar los tweets de un conjunto de usuarios. Para empezar tenemos un método para recopilar los tweets de un determinado usuario, donde se llama al método para comprobar que dicho usuario existe, se pasa por la comprobación de los límites, se recupera el id del último tweet obtenido en caso de tener ya alguno, y se hace uso del método para obtener los nuevos tweets del usuario, "parsearlos" y guardarlos.

Luego tenemos otro método que es la generalización del anterior, que obtiene la lista de usuarios a recopilar y aplica dicho método a cada uno de los usuarios.

## 2.2 Etapa 2: Procesado y obtención de los intereses

### 2.2.1 Investigación previa

Para procesar los tweets se va a utilizar un servicio de extracción de entidades de terceros, y los servicios de procesado de lenguaje natural que mejor funcionan y de mayor reputación son los de Amazon Web Services (AWS) y Google Cloud. Así que se van a consultar las métricas y precios de ambas plataformas [6] [7].

Para empezar podemos ver que ambos servicios fijan una cantidad de caracteres como unidad mínima de procesamiento, en el caso de AWS es de 100 caracteres, pero como el cargo mínimo es de 3 unidades, son 300 caracteres mínimo por petición, mientras que con Google Cloud son 1000 caracteres mínimo por petición. Para nuestro caso en particular, donde los tweets pueden ser como mucho de 280 caracteres, nos interesaría más a priori la unidad mínima de AWS para hacer una petición por tweet, pero en realidad los tweets no suelen ser del tamaño máximo y de hecho la gran mayoría sigue siendo del antiguo límite de 140 que tampoco se llenaba. Por tanto con ambas plataformas estaríamos infrautilizando el coste del procesamiento, así que habrá que procesar los tweets de forma agrupada, no siendo por tanto un motivo decisivo para decantarnos por uno o por otro.

El siguiente punto a tener en cuenta es el coste de cada plataforma, puesto que trabajan con distintas unidades mínimas de procesamiento, los precios no son directamente equiparables, y por si no fuera poco, hay distintos tramos en función del volumen de peticiones. A continuación se muestran los tramos de ambas plataformas.

| **Unidades** | **Precio ($)** |
|---|---|
| < 10 millones | 0,0001 |
| 10-50 millones | 0,00005 |
| > 50 millones | 0,000025 |

Tabla 2. Precios AWS

| **Unidades** | **Precio ($)** |
|---|---|
| < 5000 | 0,00 |
| 5k-1 millón | 0,001 |
| 1-5 millones | 0,0005 |
| 5-20 millones | 0,00025 |

Tabla 3. Precios Google Cloud

Para hacer correctamente la comparación y en particular para nuestro caso, vamos a calcular directamente el máximo coste que supondría con cada uno para 100 usuarios. Para calcular el máximo coste, vamos a suponer el máximo número de tweets obtenible por usuario y el máximo número de caracteres por tweet. Entonces tendríamos 100 usuarios x 3200 tweets/usuario x 280 caracteres/tweet = 89,6 millones de caracteres a procesar como mucho. Lo que sería en cada caso un total de unidades:

AWS: 89.600.000 caracteres / 100 caracteres/unidad = 896.000 unidades < 10 millones
Google: 89.600.000 caracteres / 1000 caracteres/unidad = 89.600 unidades < 1 millón

Y teniendo en cuenta los tramos por los que pasan supondría un coste de:

AWS: 896.000 unid. x 0,0001 $/unid. = 89,60 $
Google: 5.000 unid. x 0,00 $/unid. + (89.600-5.000) unid. x 0,001 $/unid. = 84,60 $

Por lo que, aunque salen valores similares, en base al coste elegiríamos el servicio de Google Cloud. En cualquier caso, para este proyecto se va a utilizar Google Cloud de todas formas porque ofrece una prueba gratuita de hasta 300$ de gasto.

## 2.2.2 Extracción de entidades

La extracción de entidades y la identificación de intereses se han desarrollado en forma de dos procesos independientes para "*cachear*[2]" las entidades extraídas y poder repetir todas las veces que haga falta el proceso de identificación de intereses sin incurrir en costes cada vez. Ambos procesos se ejecutan secuencialmente, pero lo que se hace es guardar las entidades extraídas en local y que el segundo proceso lo recupere de disco en vez de pasárselo directamente como variables. De esta forma, cuando se vaya a ejecutar el primer proceso (*figura 5*), si detecta que la ruta de acceso ya existe, lo da ya por hecho y pasa directamente al segundo proceso de identificación.

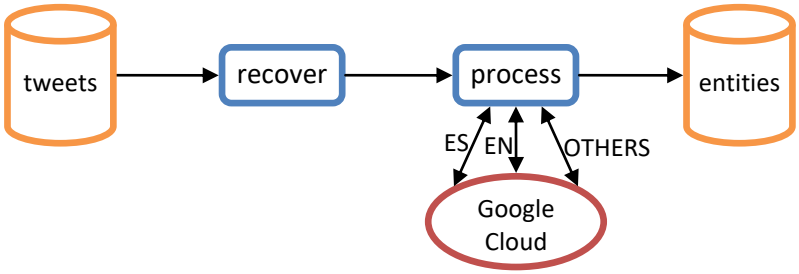



Figura 5. Esquema del proceso de extracción de entidades

Llegados a este punto ha habido que tomar una decisión con la que habrá que ser consecuentes en el resto del proyecto, y es que podemos tener tweets en cualquier idioma, por simplicidad lo hemos acotado a español e inglés, ya que aunque el procesado de lenguaje natural que hace Google soporta multitud de idiomas, no podemos controlarlos todos. Un ejemplo de ello es el catalán, que da un error al intentar procesarlo. Entonces, como tenemos que concatenar los tweets para llegar a las unidades mínimas de 1000 caracteres y hacer el procesado por un determinado idioma, se han separado los tweets en 3 cadenas de texto: español, inglés y otros idiomas. De esta forma se procesan español e inglés de forma independiente, y finalmente se ha optado por procesar también la cadena de otros idiomas, para obtener con ello algunas entidades adicionales, ya que se ha visto que en muchas ocasiones Twitter etiqueta un tweet en un idioma que no se corresponde por faltas de ortografía, perdiendo sino entidades que sí están en español o inglés.

Para las primeras pruebas realizadas en este primer proceso, se ha utilizado un subconjunto de 17 usuarios para evitar costes excesivos. Y en el procesado final de los 100 usuarios, el coste total ha sido bastante inferior al calculado en el apartado de investigación previa, ya que ahí se estaba suponiendo el máximo tamaño de tweet, y la realidad es que el tamaño promedio de los tweets recopilados es de unos 80 caracteres, por lo que el coste real ha sido de 19,71$, menor que los 300$ de prueba.

Si recalculamos los costes con el tamaño promedio real obtenemos:

100 usuarios x 3200 tweets/ususario x 80 caracteres/tweet = 25,6 millones caracteres
25.600.000 caracteres / 1000 caracteres/unidad = 25.600 unidades
5.000 unid. x 0,00 $/unid. + (25.600-5.000) unid. x 0,001 $/unid. = 20,60 $ ≈ 19,71$

[2] "Cachear": Almacenar un resultado para evitar volver a generarlo más adelante.

### 2.2.3 Obtención de intereses

Tanto si se ha realizado el proceso anterior como si se ha saltado porque ya ha sido realizado anteriormente, el segundo proceso parte de las entidades almacenadas. A continuación puede verse el esquema correspondiente al segundo proceso.

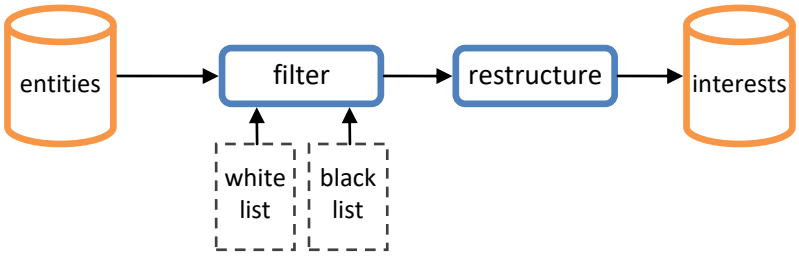



Figura 6. Esquema del proceso de identificación de intereses

Una vez extraídas las entidades, no nos quedamos con todas, seleccionamos las relevantes, ya que aunque Google Cloud sea de los servicios más avanzados en este sector, el procesado de lenguaje natural no es una ciencia exacta y también se comenten errores. Además el extractor de entidades de Google Cloud es de carácter general y devuelve algunas palabras que no entran dentro de lo que se espera en este caso, que no tienen un significado propio, o incluso caracteres sin sentido. Así que para empezar desechamos todas las entidades de menos de 3 caracteres como por ejemplo ";)" o "<3", ya que no son pocas y al repetirse tanto llegan a salir como el máximo interés de algunos usuarios. Pero esta regla de filtrado introduce otro problema, y es que de esta manera se pierden muchas entidades de 2 caracteres muy relevantes, tales como "PP", "PC", "TV", "CV" o "4G". Por ello se ha introducido una lista blanca en forma de fichero "entities_white.list" donde se han guardado todas las entidades que a priori se han considerado relevantes. De igual manera se ha creado también una lista negra "entities_black.list" donde se han puesto todas las entidades recurrentes que han ido saliendo y se consideraban inoportunas, como "algo", "todo", "cosas", "vía",…

Partiendo de las entidades ya filtradas del proceso anterior, vamos a transformarlas en intereses. Para empezar, lo que se hace es juntar todas las entidades detectadas y contar el número de apariciones de cada una de ellas, ordenándolas de más a menos. Esto es tan sencillo como recorrer todas las entidades de un usuario y crear un diccionario clave-valor, donde la clave es la entidad, y cada vez que se vuelve a encontrar, se incrementa el valor asociado a dicha entidad. Después esto se convierte a una lista de objetos, recorriendo el diccionario y creando para cada entrada un objeto que se compone de la entidad y el recuento total, pudiendo así ordenar la lista en orden decreciente por el atributo "count". Posteriormente se unifican los intereses léxicamente parecidos, tanto ignorando las diferencias entre mayúsculas y minúsculas, como las diferencias que puede haber al no acentuar las palabras e incluso considerando similares las palabras que coincidan en más de un 75% de caracteres. A continuación se desechan los intereses que solo aparezcan una vez, ya que son muchísimos y además de poca importancia para el usuario, suelen ser ruido en la mayoría de las ocasiones. Para terminar, se les asigna un peso normalizando respecto del interés más frecuente, siendo 1 el valor del que más le interesa.

## 2.3 Etapa 3: Relaciones y cálculo de la afinidad

### 2.3.1 Relaciones de Wikidata

Para poder relacionar intereses distintos entre sí se va a utilizar Wikidata como ontología para identificar si un interés tiene alguna relación con otro (por ejemplo Elon Musk es el fundador de SpaceX), o si hay alguna relación en común entre ambos intereses (por ejemplo, entre Tesla Motors y SpaceX está Elon Musk).

Como ya se ha comentado anteriormente, se ha escogido Wikidata principalmente por el volumen de datos que tiene, ya que cuenta con más de 40 millones de entradas, mientras que DBpedia pasa de los 4 millones y Wikipedia tan solo roza el millón y medio. Además tanto DBpedia como Wikipedia funcionan de forma separada para cada idioma, lo que dificulta trabajar con varios idiomas a la vez (aunque en este caso solo sea español e inglés). Otra razón por la cual también se descarta Wikipedia, es que está más orientada al usuario, conteniendo mucha más información visual, mientras que Wikidata y DBpedia tienen los campos básicos.

Por tanto, para relacionar intereses entre sí, vamos a coger la base de datos de Wikidata y a procesarla hasta quedarnos con lo mínimo necesario para nuestras necesidades, ya que solo nos interesan las relaciones de las entidades y por simplicidad solo en español e inglés. En la *figura 7* puede verse el proceso seguido.

Si consultamos en Wikidata la entrada correspondiente para la descarga de su base de datos [8], vemos que nos la ofrecen en varios formatos, entre ellos JSON que además de ser el recomendado, lo elegimos por lo fácil que es de manejar en Python. Además nos fijamos también en el apartado de licencia y vemos que está disponible bajo la licencia Creative Commons CC0 [9], por lo que no hay problema en usarlo.

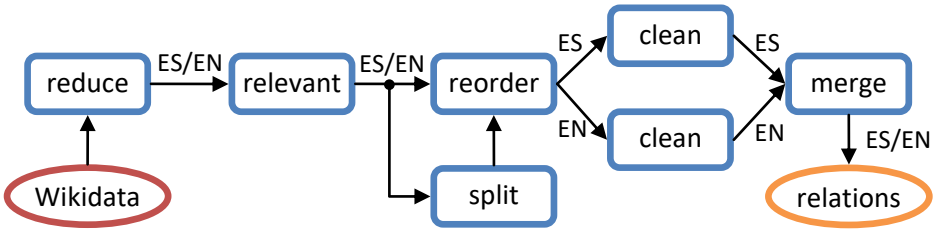



Figura 7. Diagrama de bloques del procesado de Wikidata

Descargamos la base de datos y la descomprimimos en un fichero resultante de 469GB y 48 millones de entidades, hacemos una primera reducción quitando los idiomas sobrantes y simplificando las relaciones en: entidad relacionada, tipo de relación y relevancia. En el *anexo 1* puede verse un ejemplo de la estructura de datos en cada paso.

El problema de procesar un fichero JSON tan grande es que habría que cargarlo entero en memoria, y puesto que no disponemos de 500GB de RAM, ha habido que pensar alternativas. Una posibilidad era leerlo en forma de stream, de esta forma solo se cargaba una pequeña parte en memoria. Aun así era una operación bastante lenta, así que se aprovechó que el fichero está formado con una entrada por línea, lo cual es muy sencillo de leer y a su vez de escribir para guardar el resultado (*figura 8*).

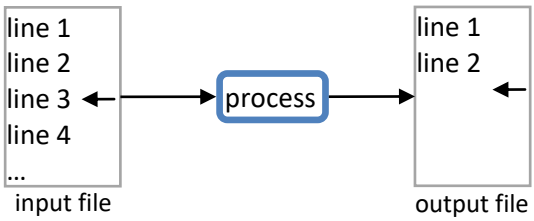



Figura 8. Procesado de un fichero grande por líneas

En el procesado de cada línea, lo que se hace es seleccionar los campos en español e inglés e identificar en las relaciones los campos de interés mencionados. En unas 9h de procesado obtenemos un fichero resultante de 16GB y 36 millones de entidades, ya que se han quitado las que no estuvieran en español o inglés. De ahora en adelante partiremos de este archivo ya que es más manejable.

Como a nosotros no nos hacen falta todos los campos como definición y sinónimos, hacemos otro procesado para quedarnos esta vez solamente con lo que nos interesa. Nuevamente volvemos a procesar el fichero por líneas, ya que 16GB sigue siendo una cantidad excesiva para un ordenador corriente. En este paso seleccionamos solo los campos que nos interesan, descartando así la definición en español e inglés del término, y en las relaciones quedándonos solo con el término relacionado. Tras 1h de procesado nos sale un fichero de 4,8GB y 1,3 millones de entidades, ya que en este paso también hemos filtrado y nos hemos quedado con las relaciones que Wikidata marca como más relevantes y por tanto se han quitado las entidades que se han quedado sin relaciones. Hay que destacar que además de las relaciones marcadas como relevantes, también nos hemos quedado con algunos tipos de relación que pueden verse en el correspondiente *anexo 1*.

Ahora procedemos a reorganizar la estructura, ya que hasta ahora era una lista de objetos en la que para seleccionar una determinada entidad había que recorrerlo todo, y lo convertimos en un diccionario de entidad-relaciones. El problema de ello, es que para sustituir el id del término por el propio valor de la entidad, hay que poder seleccionar una entidad en particular por su id. Así que lo que se hizo de forma provisional, fue partir el fichero de 4.8GB en ficheros individuales para cada término, poniéndole como nombre al archivo, el de su id. De esta forma el acceso es inmediato.

Al pasar a utilizar el propio nombre de la entidad como clave de selección, ha habido que hacer un diccionario a parte para cada idioma. Ya que teníamos el nombre de la entidad en ambos idiomas. Además esta separación por idiomas podría ser útil.

Finalmente limpiamos las relaciones que salen repetidas dentro de una entidad para cada diccionario, y por ahora unificamos las entidades, ya que son muchísimas las que son iguales en español e inglés, como personas. Al final ha resultado en 130MB.

## 2.3.2 Cálculo de la afinidad

El algoritmo para calcular la afinidad entre dos usuarios es relativamente sencillo, ya que lo que se hace es recuperar los intereses de ambos usuarios y cruzarlos para encontrar coincidencias. Las coincidencias pueden ser totales, si es exactamente la misma entidad, directamente relacionadas si una figura como relación de la otra en la ontología de Wikidata, o indirectamente relacionadas si hay otras entidades de por medio. A continuación se muestra un caso de cada una de ellas:

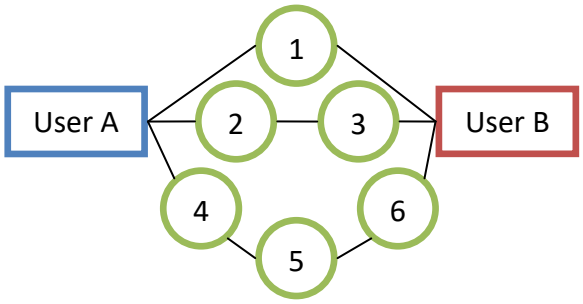



Figura 9. Tipos de relaciones

Es obvio que no todas las coincidencias van a ser igual de relevantes, por ello cada tipo va a tener un valor distinto. Para las coincidencias totales 1.00, para las relaciones directas 0.75, y para las relaciones indirectas de hasta 1 salto 0.50.

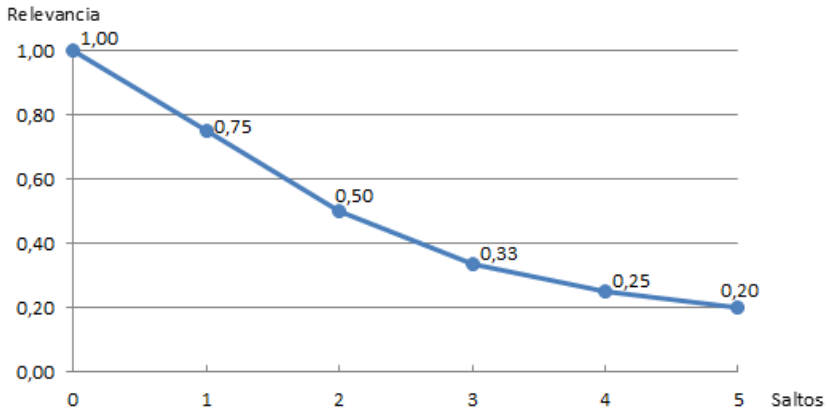



Figura 10. Valor de relevancia de las relaciones

Idealmente el valor de relevancia de las relaciones decaería siguiendo la curva de la función inversa 1/n, ya que cuanto más indirecta es una relación, menos relevante es. La excepción sería obviamente cuando la coincidencia es plena, ya que al ser el mismo interés, la relevancia es máxima, y cuando la relación es directa entre dos intereses, tomamos el valor intermedio entre el máximo y el siguiente. La función inversa entraría ya con las relaciones indirectas, en nuestro caso hasta 2 saltos.

Puesto que para cada usuario no tiene el mismo peso los intereses en común, se toma el primer usuario como el principal y el segundo como el usuario con el que se quiere comparar y se elige como peso de referencia el menor de ambos. Ya que para un determinado interés en común, si el primer usuario tiene un peso de 0.6 y el segundo uno de 1.0, como mucho esa relación será un 0.6 de interesante para ambos, siendo plena para el primero y parcial para el segundo.

Al final lo que se hace es sumar los pesos de todas las relaciones y normalizarlo respecto de la suma de los pesos de todos los intereses del primer usuario, siendo 1.0 la afinidad de un usuario consigo mismo. Por tanto, al comparar un usuario con muchos intereses y otro con muy pocos, si esos pocos intereses los tiene el otro, resultará en una alta afinidad del usuario con pocos intereses hacia el de muchos, y para el de muchos intereses apenas le parecerá interesante el de pocos y por tanto tendrá una baja afinidad hacia él. Así que por definición, la afinidad no tiene por qué ser correspondida. En las siguientes figuras se intenta mostrar de forma gráfica e intuitiva lo que se acaba de explicar:

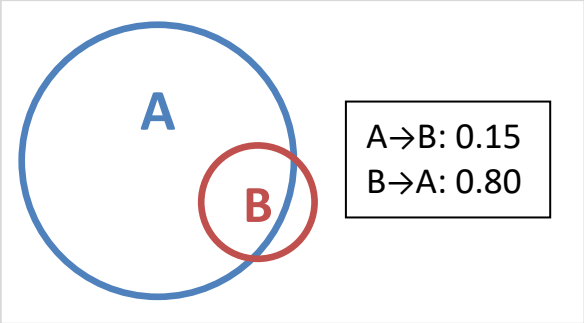



Figura 11. Afinidades entre 2 usuarios

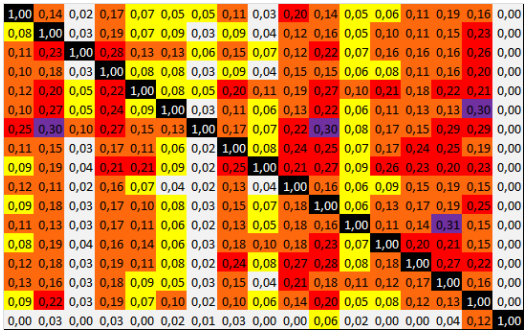


Figura 12. Matriz de afinidades de 17 usuarios

En la figura de la derecha, se ha representado la matriz resultante de cruzar las afinidades de un subconjunto de 17 usuarios. Lo primero que se puede apreciar es que en la diagonal principal son todo unos, y esto es así por lo que se ha comentado de que la afinidad con uno mismo ha de ser 1 por definición. Si la matriz fuera simétrica respecto de la diagonal, esto querría decir que las afinidades son correspondidas. Pero no es el caso y de hecho puede verse como los valores se asemejan por columnas, lo cual quiere decir que el interés que tiene un usuario hacía los demás viene a ser similar y esto seguramente sea porque tenga los mismos intereses en común con la mayoría de usuarios, así que lo interesante es ver las anomalías en las columnas.

Para finalizar se va a hacer un ejemplo completo paso a paso del cálculo de la afinidad entre dos usuarios con distintos tipos de relaciones entre ellos.

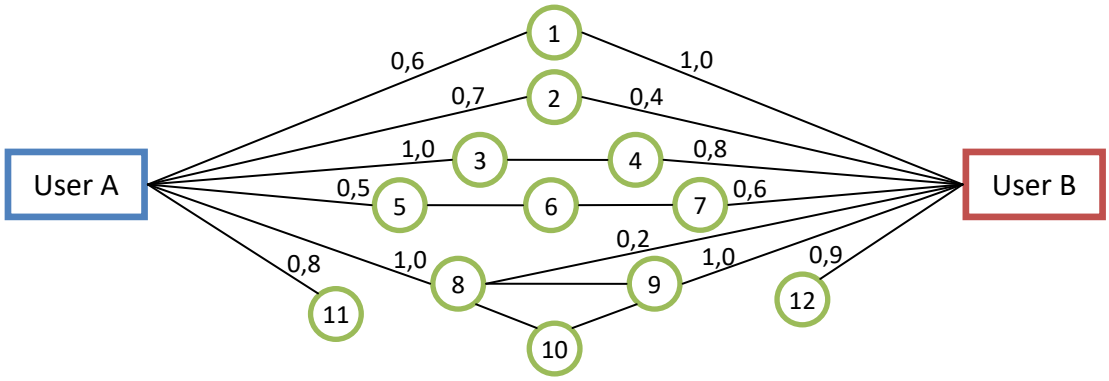



Figura 13. Relaciones entre dos usuarios

Como puede verse en la figura, cada usuario tiene sus propios pesos para cada uno de sus intereses, en algunos intereses la coincidencia es plena (valor de 1.00), en otros es mediante relaciones directas (0.75) y en otros indirectas (0.50). Puesto que la afinidad no tiene por qué ser recíproca, vamos a hacer el cálculo en ambos sentidos.

Afinidad A→B:

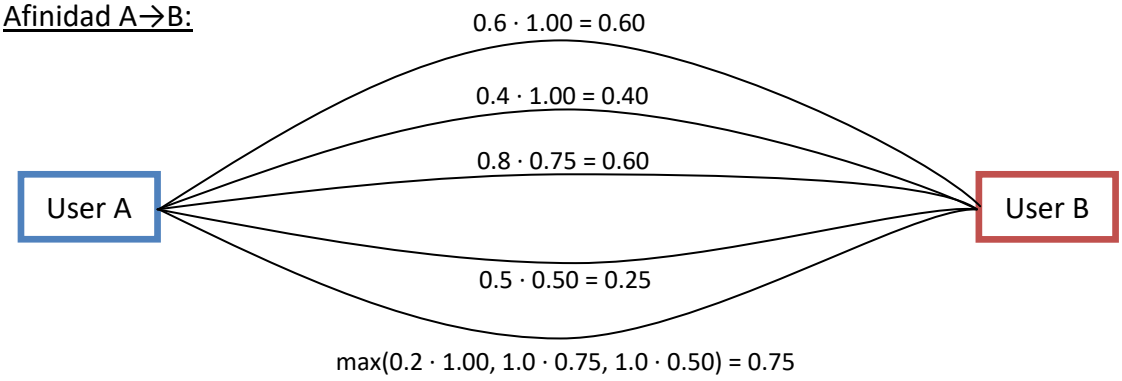



Figura 14. Caminos del usuario A al B

Afinidad = $(0.60+0.40+0.60+0.25+0.75)/\sum w_A = 2.60/(0.6+0.7+1.0+0.5+1.0+0.8) = 0.57$

Afinidad B→A:

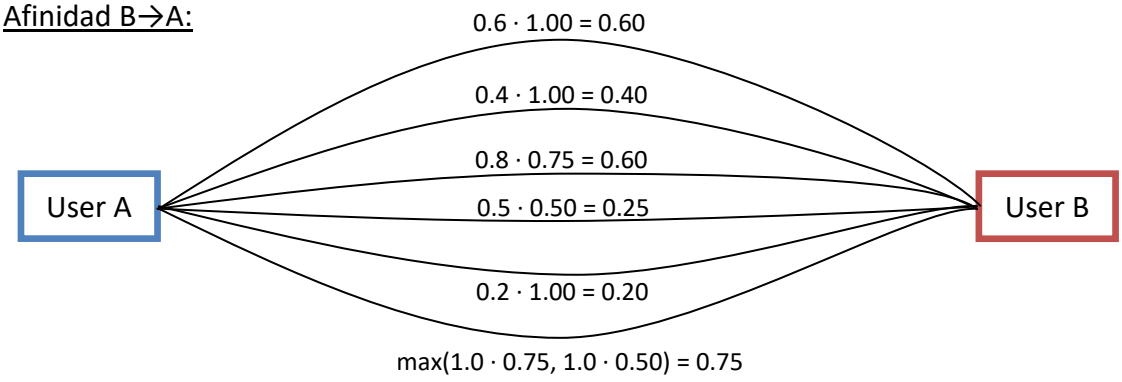



Figura 15. Caminos del usuario B al A

Afinidad = $(0.60+0.40+0.60+0.25+0.20+0.75)/\sum w_B = 2.80/(1.0+0.4+0.8+0.6+0.2+1.0+0.9) = 0.57$

## 2.4 Etapa 4: Representación en grafos y búsquedas

### 2.4.1 Volcado en Neo4j

Antes de realizar el volcado de datos sobre Neo4 es preciso tener corriendo un servidor tal como se explica en el *anexo 2* y guardar las credenciales del mismo en el fichero de configuración "config.json". Para interactuar con el servidor Neo4j desde Python, se ha usado la librería Py2neo [10].

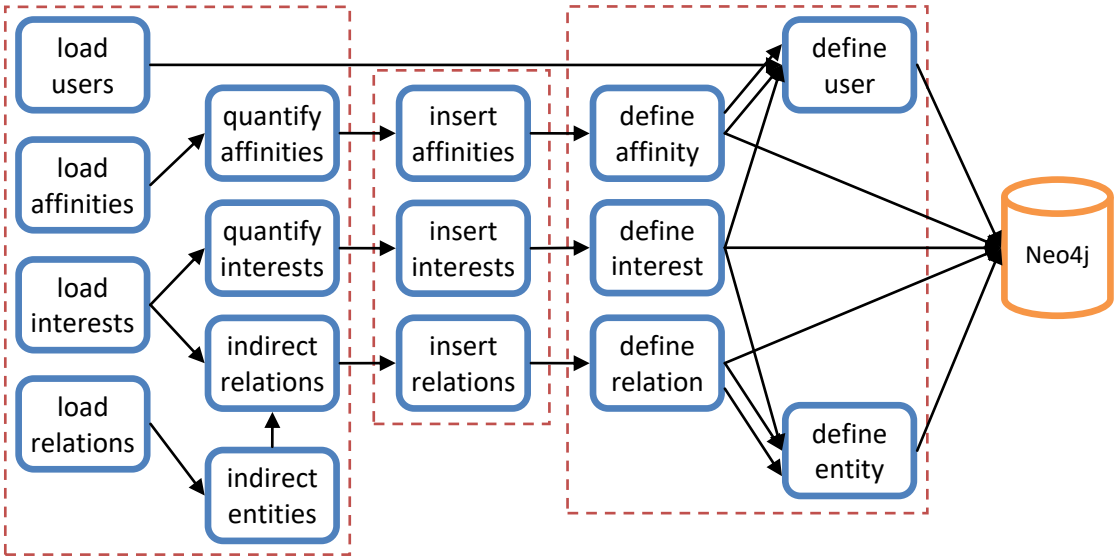



Figura 16. Bloques con las principales funciones de la etapa 4

Al igual que en la etapa de recolección de tweets, se han agrupado las funciones con un determinado fin (*figura 16*). Se ha creado un bloque de funciones para recuperar de local los datos de las etapas anteriores y enriquecerlos con etiquetas, otro para los métodos que interactúan con el servidor Neo4j mediante la librería Py2neo, y otro que crea los objetos con la estructura a insertar.

Lo primero que se ha hecho en esta etapa ha sido recuperar las afinidades, intereses, relaciones de Wikidata e información de usuario de las anteriores etapas. La información de usuario básica, obtenida en la primera etapa junto con la recolección de tweets, se introduce sin tratamiento alguno. En el caso de los intereses y las afinidades, se les ha agregado una etiqueta cuantificando la intensidad de la relación en base a sus pesos. Los valores se han segmentado en una serie de niveles que nos permitirán posteriormente filtrar más fácilmente por mayor o menor relevancia de dichas relaciones. Concretamente se han fijado los siguientes niveles:

| Peso | Etiqueta |
|---|---|
| >= 0.10 | AFFINITY_1 |
| >= 0.20 | AFFINITY_2 |
| >= 0.30 | AFFINITY_3 |

Tabla 4. Niveles afinidad

| Peso | Etiqueta |
|---|---|
| < 0.10 | INTEREST_0 |
| >= 0.10 | INTEREST_1 |
| >= 0.20 | INTEREST_2 |
| >= 0.50 | INTEREST_3 |

Tabla 5. Niveles interés

En el caso de las afinidades, como se cruzan todos los usuarios con todos, estamos forzando a que haya un nivel de afinidad entre dos usuarios aunque este sea muy bajo, por eso descartamos las afinidades inferiores a 0.10 ya que realmente no existe una relación entre ellos. En cuanto al otro extremo, con la muestra de usuarios que se ha estado trabajando, las mayores afinidades que se han obtenido están entre 0.30 y 0.45, por eso se ha fijado >= 0.30 como la más relevante.

Para los intereses, como hay una gran cantidad de ellos que no tienen ningún tipo de relación con otros usuarios o intereses, en vez de descartarlos para evitar que empañen la visión en la representación gráfica, se han conservado con la etiqueta 0.

En cuanto a las relaciones de Wikidata, las cargamos en memoria "cacheándolas" para evitar hacer la lectura en disco cada vez que se usen. Lo que se hace en este paso es juntar los intereses de todos los usuarios en una lista y recorrerla cruzándola consigo misma, y en caso de que los dos términos cruzados figuren en la lista de relaciones cargada, se obtienen las relaciones de ambas entidades y se comparan para extraer las relaciones que tienen en común. Si se han obtenido entidades indirectas entre ambos términos, se almacenan como relaciones indirectas en la lista asociada a cada término.

Una vez tenemos preparados los tres tipos de relaciones, pasamos a las funciones de mayor abstracción, que básicamente son, una para insertar todo en el servidor, y otra para borrar todo lo insertado. Antes de insertar, siempre se hace un borrado para asegurarnos de que los objetos ya existentes se actualicen a los nuevos valores. Para insertar los datos, lo que se hace es recorrer todos los usuarios y recorrer a su vez todos los intereses y afinidades (tras cuantificarlos), insertando en cada ocasión un nodo-relación-nodo, siendo siempre el primer nodo el usuario y en función de si el segundo es un interés u otro usuario, la relación será de tipo interés o afinidad.

Por último se insertan las entidades de las relaciones indirectas, también como un nodo-relación-nodo, pero en este caso entre entidades con un simple "RELATION".

Cada vez que se inserta una relación de afinidad, interés o relación indirecta, se define cada uno de estos tipos construyendo el correspondiente objeto en Neo4j.

Además al crear cualquier tipo de relación, también se definen los correspondientes nodos de usuario o entidad para insertar también dichos objetos en Neo4j. Tanto la definición de los usuarios como la de las entidades, son "cacheadas" para minimizar el tiempo, ya que ambas salen en más de una relación.

## 2.4.2 Resultados en forma de grafos

Tras aproximadamente media hora de inserción, ya tendremos todos los datos con los que se ha estado trabajando desde Python en nuestro servidor Neo4j, donde ya podemos visualizarlo todo de forma gráfica y hacer búsquedas con ello. En el *anexo 5* se explica cómo acceder y visualizar correctamente los resultados desde la interfaz gráfica que trae el servidor Neo4j, y se explica brevemente cómo funciona la sintaxis de las búsquedas. En este apartado solo se van a mostrar los resultados de algunas búsquedas representativas de lo que se ha logrado con este proyecto.

Para empezar vamos a ver cuántos elementos de cada tipo tenemos al final. Como era de esperar, tenemos 100 usuarios, que es la muestra que hemos cogido, entre los cuales hay un total de 2841 relaciones de afinidad. Un total de 16033 entidades entre intereses y entidades de relaciones indirectas, 31232 relaciones de interés, y 2927 relaciones entre entidades de relaciones indirectas.

Ahora vamos a consultar la afinidad entre dos usuarios cualesquiera:


Figura 17. Afinidades entre dos usuarios

Puede verse como en este caso la afinidad no es recíproca, ya que el usuario de la derecha está más interesado en los temas de los que habla el usuario de la izquierda, y al usuario de la izquierda no le interesa tanto lo que dice el de la derecha.

Pero veamos qué intereses tienen en común para obtener esa afinidad:

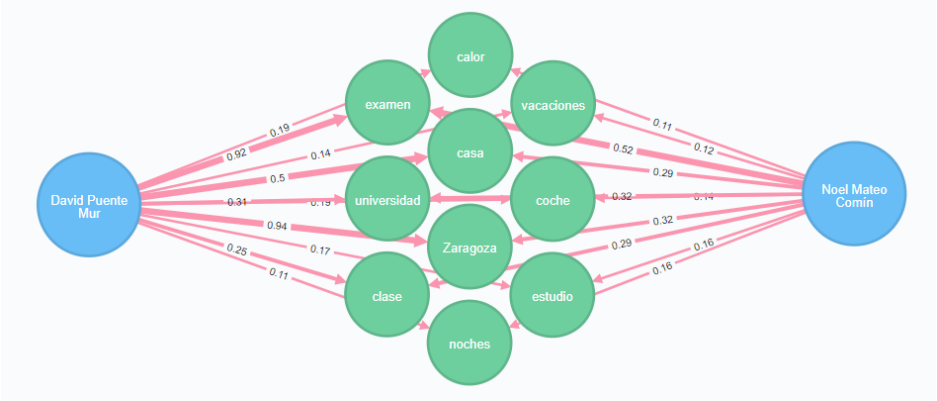



Figura 18. Intereses relevantes en común entre dos usuarios

En esta representación solo se están mostrando los intereses más relevantes para poder ver algo con claridad, pero aunque no estén todos los intereses en común, puede apreciarse que para el usuario de la izquierda tienen más peso que para el usuario de la derecha, es por ello que el usuario de la izquierda se queda insatisfecho con el de la derecha y que al de la derecha le sacia más el poco interés que tiene.

En la anterior figura solo estábamos viendo los intereses en común entre ambos usuarios, pero obviamente hay otros muchos intereses que tenía solo uno del otro y otros tantos que no le resultaban interesantes del otro. Pero también podemos ver todos los intereses que tiene un usuario en particular (filtramos por relevantes):

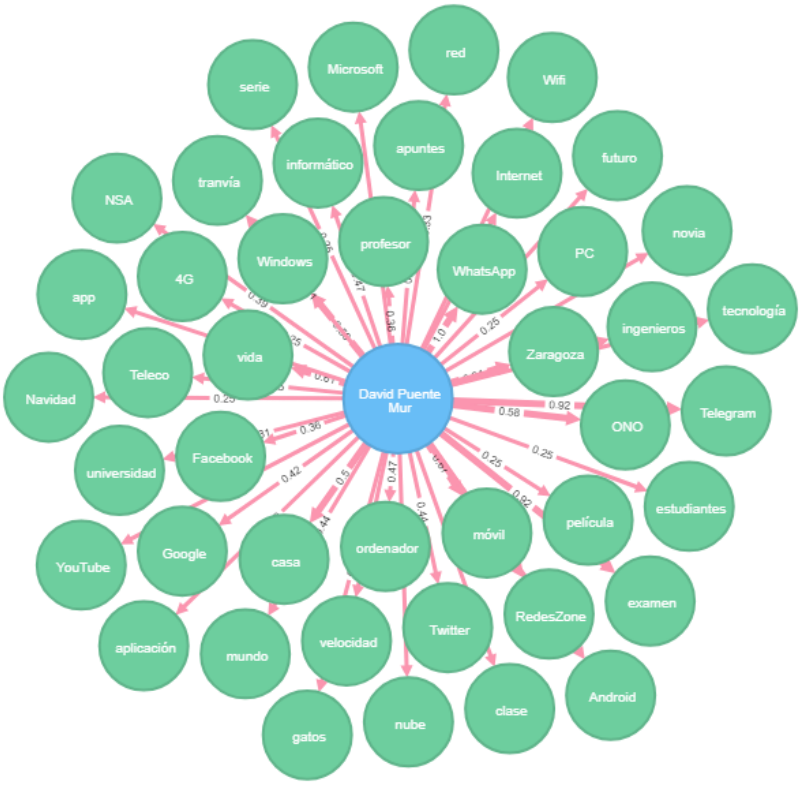



Figura 19. Intereses relevantes de un usuario

Y al igual que podemos ver todos los intereses de un usuario, también podemos ver todos los interesados en una determinada entidad (sin filtrar en esta ocasión):


Figura 20. Interesados en una entidad

Jugando con este tipo de consultas, podríamos hacer búsquedas como la siguiente. Supongamos que somos de recursos humanos de una empresa como Everis y que necesitamos a alguien para un departamento como el de inteligencia artificial. Pues bien, podríamos hacer una búsqueda para ver qué usuarios están interesados en ambos términos (se han incluido las cuentas de dos empleados de Everis a propósito):

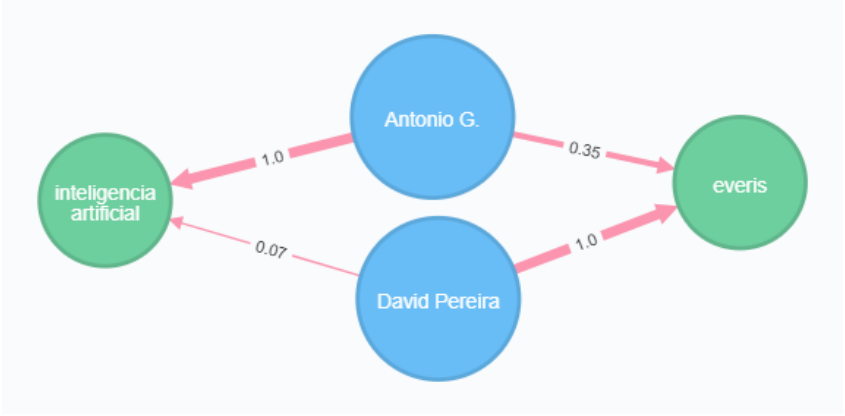



Figura 21. Usuarios interesados en 2 términos

Y con los usuarios obtenidos podríamos investigar el resto de sus intereses.

Habitualmente para valorar la cuenta de un usuario se mira si está verificada y el número de seguidores, pero en ocasiones las cuentas realmente interesantes pasan desapercibidas, así que buscamos el usuario que resultaría interesante a más personas dentro de nuestra muestra de 100 usuarios:

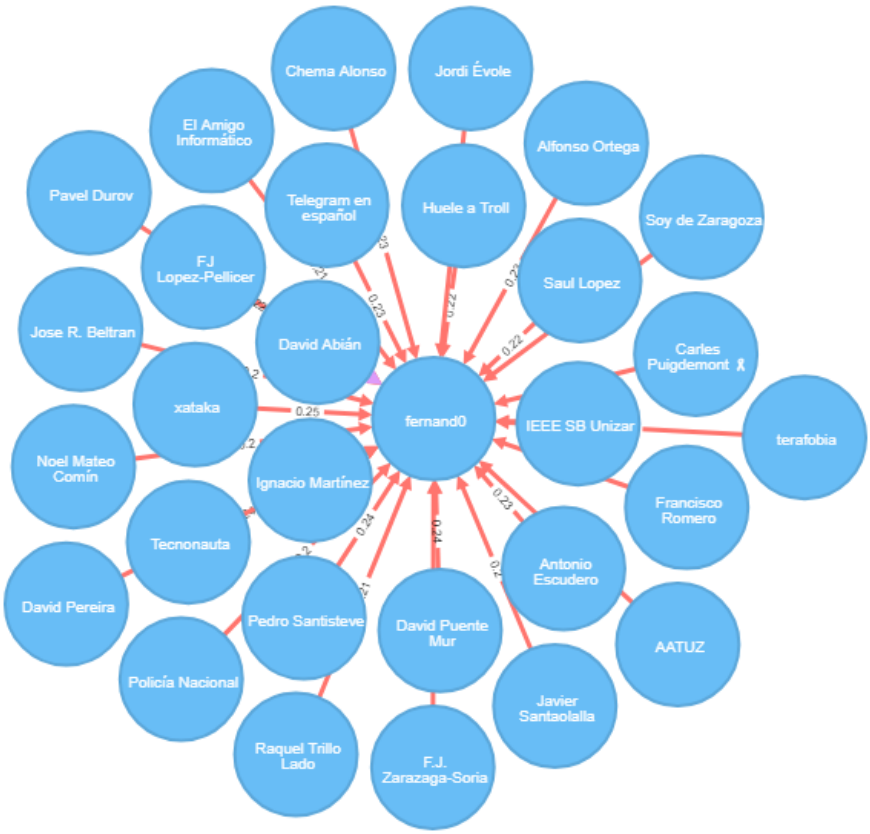



Figura 22. El usuario más interesante de la muestra

# 3 Conclusiones y trabajos futuros

## 3.1 Conclusiones

A lo largo de este proyecto, se ha desarrollado un algoritmo, haciendo uso de la API de Twitter (documentándose sobre su uso y limitaciones), para recopilar los tweets de un conjunto de usuarios de interés. Se ha valorado el uso de un servicio de extracción de entidades basándose en su coste y documentación, para procesar los tweets recopilados y extraer entidades con las que identificar los intereses de los usuarios. Se ha procesado la base de datos de Wikidata, con la que usándola como ontología, se han obtenido relaciones directas e indirectas entre los intereses identificados para posteriormente calcular la afinidad entre dos usuarios en base a esas relaciones. Finalmente se ha usado Neo4j para representar gráficamente en forma de grafos las relaciones directas e indirectas entre intereses y usuarios, y sus afinidades.

No solo se ha cumplido con los objetivos propuestos, sino que además se ha ido más allá de calcular la afinidad entre dos usuarios y se ha convertido toda la información tratada en una herramienta con la que hacer búsquedas útiles.

Durante el desarrollo de este proyecto se ha aprendido a usar la API de Twitter, cómo calcular los costes de usar servicios en la nube como Google Cloud, y a hacer búsquedas en una base de datos orientada a grafos como Neo4j.

También ha habido contratiempos o limitaciones que no se preveían antes de comenzar el proyecto. Por ejemplo, en la recopilación de tweets, hasta que no se consultó la documentación detenidamente, no se supo que no se podía obtener más de 3200 tweets de un usuario. Al final ha sido una muestra más que suficiente, pero de no haber sido el caso habría supuesto un problema para continuar con lo propuesto. Un contratiempo que sí surgió, fue que los tweets de más de 140 caracteres salían truncados por la retrocompatibilidad de la API de Twitter, para lo cual, como ya se ha comentado en la parte correspondiente de investigación, hubo que buscar en páginas de terceros para solucionarlo. Y un aspecto que se subestimó, fue el intentar procesar una gran base de datos como la de Wikidata con un ordenador corriente. Para lo cual hizo falta muchas horas de procesado, y por tanto cada error inesperado que surgía a mitad del proceso, suponía una gran pérdida de tiempo. Además hubo que hacer un esfuerzo optimizando las funciones para reducir considerablemente los tiempos de procesado.

## 3.2 Futuros trabajos

De cara a futuras actualizaciones, hay varios aspectos a mejorar o al menos a contemplar, ya que quizás puedan suponer un tiempo y esfuerzo innecesarios.

Para empezar, actualmente la recopilación de tweets usa el nick del usuario y aunque este cambie, ya los estamos distinguiendo por su id, pero en el tiempo que ha durado este proyecto no se ha dado el caso y de cara a mantener este sistema en el tiempo, habría que ver detalladamente qué implicaciones tendría y seguramente modificar el sistema para basarlo completamente en el id de usuario en vez en el nick.

Otra idea que se ha tenido a lo largo del proyecto, es que además de recopilar los tweets, retweets y respuestas del usuario, se podría hacer lo mismo con los tweets que ha marcado como favoritos. Al igual que con los ya recopilados, hay un límite de 3200 tweets por usuario, por lo que de esta forma lograríamos duplicar la muestra.

Otra forma de aumentar los intereses identificados, sería sacarle más partido a los tweets que ya tenemos, no desechando los enlaces y el contenido multimedia. Ya que aunque a priori no nos aporten información de forma directa, se podría intentar ver si merece la pena consultar los enlaces y extraer al menos el título de la web a la que apuntan. Y en el caso de las imágenes, se podría desarrollar o usar algún servicio ya existente de visión por computación que nos revele los objetos que aparecen.

Una cosa que se quiso hacer desde el principio con los tweets, era volcarlos también sobre la representación de grafos para ver de qué contexto salían los intereses identificados, pero se ha perdido esa trazabilidad al tener que juntar todos los tweets para procesarlos en el extractor de entidades de Google Cloud por cuestión de costes. Pero si se utilizara un extractor de entidades propio ya sería posible hacerlo.

Al hilo de la anterior idea surge la de desarrollar con Machine Learning una pequeña red neuronal que identifique entidades de un texto y de esta forma poder entrenarla para el caso concreto de los tweets y obtener así mejores resultados.

Otra cosa que se ha barajado hacer a posteriori y que finalmente se ha descartado, es usar los sinónimos o “aliases” de Wikidata para agrupar las entidades similares no solo léxicamente sino también un poco más semánticamente. Aunque tampoco terminaría de resolver el problema de que haya una misma entidad escrita de varias formas distintas, quizás también habría que echar mano de redes neuronales…

En la parte de transformar la base de datos de Wikidata, también se podrían hacer algunas mejoras, especialmente en el filtrado de relaciones relevantes, que por tratar de obtener algo manejable no se ha ampliado, pero se debería mirar qué otros tipos de relaciones podrían ser interesantes, por ejemplo: guionista, cantante,…

Además, de cara a representar gráficamente las relaciones indirectas entre entidades, podría conservarse en la adaptación de Wikidata, los nombres de los tipos de relación, ya que actualmente es la única relación que no tiene ninguna propiedad.

Y para finalizar, si se va a incrementar el número de usuarios de forma considerable, habría que optimizar la forma en la que se obtienen las afinidades entre los usuarios, ya que actualmente se cruzan todos con todos cada vez. Lo lógico sería que cuando se introduce un nuevo usuario sea solo este el que se cruce con los demás.

# 4 Referencias


[1] (25/04/2018) Documentación de la API de Twitter
https://developer.twitter.com/en/docs
[2] (25/04/2018) Documentación para obtener los tweets de un usuario
https://developer.twitter.com/en/docs/tweets/timelines/api-reference/get-statuses-user_timeline.html
[3] (25/04/2018) Cómo obtener enteros los tweets de más de 140 caracteres
http://www.barriblog.com/2018/02/bajarse-los-tuits-rts-280-caracteres-desde-las-apis-twitter/
[4] (25/04/2018) Librerías recomendadas para usar la API de Twitter
https://developer.twitter.com/en/docs/developer-utilities/twitter-libraries
[5] (25/04/2018) Documentación de la librería Tweepy para Python
http://docs.tweepy.org/en/v3.6.0/
[6] (30/04/2018) Precios de Amazon Web Services - Comprehend
https://aws.amazon.com/es/comprehend/pricing/
[7] (30/04/2018) Precios de Google Cloud - Natural Language
https://cloud.google.com/natural-language/pricing?hl=es
[8] (20/05/2018) Descargar base de datos de Wikidata
https://www.wikidata.org/wiki/Wikidata:Database_download/es
[9] (20/05/2018) Licencia Creative Commons CC0
https://creativecommons.org/publicdomain/zero/1.0/
[10] (05/06/2018) Documentación de la librería Py2neo
http://py2neo.org/
[11] (20/06/2018) Página de instalación de Anaconda
https://www.anaconda.com/download/
[12] (20/06/2018) Página de instalación de Jupyter Notebook
http://jupyter.org/install
[13] (05/06/2018) Página de descarga de Neo4j
https://neo4j.com/download/?ref=product
[14] (25/04/2018) Gestor de aplicaciones de Twitter
https://apps.twitter.com/
[15] (13/05/2018) Google Cloud
https://console.cloud.google.com/home/dashboard
[16] (24/06/2018) Código del Proyecto en GitHub
https://github.com/davidjaca94/TFG

# 5 Anexos

## 5.1 Anexo 1: Relaciones relevantes y estructuras

Además de las relaciones que Wikidata marca como altamente relevantes, también se han conservado algunas que se han creído necesarias, en general son de distintos tipos de pertenencia:

| Identificador | Etiqueta |
|---|---|
| P36 | capital |
| P108 | empleador |
| P112 | fundador |
| P178 | desarrollador |
| P355 | filial |
| P361 | forma parte de |
| P452 | industria |
| P527 | compuesto de |
| P749 | organización matriz |
| P1056 | producto |
| P1376 | capital de |
| P1830 | propietario de |

Tabla 6. Relaciones relevantes

A continuación se muestra como se ha ido transformando la estructura de la primera entidad extraída de Wikidata a lo largo del procesado del mismo:

```
{
   "type": "item",
   "id": "Q26",
   "labels": {
      "en": {
         "language": "en",
         "value": "Northern Ireland"
      },
      (…)
   },
   "descriptions": {
      "en": {
         "language": "en",
         "value": "region in north-west Europe, part of the United Kingdom"
      },
      (…)
   },
   "aliases": {
      "en": [{
            "language": "en",
            "value": "NIR"
         }, {
            "language": "en",
            "value": "UKN"
         }, {
            "language": "en",
            "value": "North Ireland"
         }
      ],
      (…)
   },
   "claims": {
      "P1151": [{
            "mainsnak": {
               "snaktype": "value",
               "property": "P1151",
               "datavalue": {
                  "value": {
                     "entity-type": "item",
                     "numeric-id": 11045856,
                     "id": "Q11045856"
                  },
                  "type": "wikibase-entityid"
               },
               "datatype": "wikibase-item"
            },
            "type": "statement",
            "id": "Q26$badd854c-42e5-e3cb-426d-36555c8a1469",
            "rank": "normal"
         }
      ],
      (…)
   }
}
```


Figura 23. Estructura original de Wikidata

```
{
   "item": "Q26",
   "labels": {
      "es": "Irlanda del Norte",
      "en": "Northern Ireland"
   },
   "aliases": {
      "en": ["NIR", "UKN", "North Ireland"]
   },
   "descriptions": {
      "es": "nación constituyente del Reino Unido",
      "en": "region in north-west Europe, part of the United Kingdom"
   },
   "relations": [{
         "item": "Q11045856",
         "relation": "P1151",
         "rank": "normal"
      }, {
         "item": "Q27",
         "relation": "P47",
         "rank": "preferred"
      },
      (…)
   ]
}
```


Figura 24. Estructura reducida de Wikidata


```
{
   "item": "Q26",
   "labels": {
      "es": "Irlanda del Norte",
      "en": "Northern Ireland"
   },
   "aliases": {
      "en": ["NIR", "UKN", "North Ireland"]
   },
   "relations": ["Q10686", "Q27"]
}
```


Figura 25. Estructura relevante de Wikidata


```
{
   "Northern Ireland": ["Belfast", "Ireland"],
   "Irlanda del Norte": ["Belfast", "Irlanda"]
}
```


Figura 26. Estructura reordenada de Wikidata

En este caso en particular, no hay relaciones repetidas ni coinciden los nombres en español e inglés por lo tanto el resultado tras la limpieza y mezclado sería el mismo.

# 5.2 Anexo 2: Instalación y configuración del entorno

Como ya se ha dicho, se ha utilizado Python como lenguaje de programación, para lo cual se ha usado Anaconda [11] como gestor de entornos y librerías, y se ha usado Jupyter Notebook [12] como entorno de desarrollo que ya viene con Anaconda.

Se ha creado un entorno con Python3 para el desarrollo de este proyecto y se han instalado las siguientes librerías que se van a usar a lo largo del proyecto:

```
numpy
unidecode
functools
tweepy
google-cloud-language
py2neo
```

Figura 27. Librerías instaladas

Se ha descargado e instalado el servidor de Neo4j [13] que básicamente ha consistido en descomprimirlo y lanzarlo con el comando "neo4j\bin\neo4j console".

Por defecto viene con las credenciales "neo4j" tanto para el usuario como para la contraseña. El usuario "neo4j" es administrador, y te pide cambio de contraseña.

En el directorio de trabajo se ha creado un archivo de configuración "config.json" en el que guardar las distintas credenciales para no "*hardcodearlas*[3]".

```
{
   "twitter":{
      "consumer_key":"(RELLENAR)",
      "consumer_secret":"(RELLENAR)",
      "access_key":"(RELLENAR)",
      "access_secret":"(RELLENAR)"
   },
   "gcloud":{
      "type": "service_account",
      "project_id": "(RELLENAR)",
      "private_key_id": "(RELLENAR)",
      "private_key": "(RELLENAR)",
      "client_email": "(RELLENAR)",
      "client_id": "(RELLENAR)",
      "auth_uri": "https://accounts.google.com/o/oauth2/auth",
      "token_uri": "https://accounts.google.com/o/oauth2/token",
      "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
      "client_x509_cert_url": "(RELLENAR)"
   },
   "neo4j":{
      "host": "localhost",
      "port": 7474,
      "user": "neo4j",
      "password": "(RELLENAR)"
   }
}
```

Figura 28. Estructura de configuración

[3] Hardcodear: incrustar en el código datos que podrían cambiar más adelante.

## 5.3 Anexo 3: Obtención de credenciales

Para usar los servicios de las API's es preciso obtener unas credenciales con las que identificarse y de ese modo poder hacer un uso controlado de dichos servicios.

En el caso de Twitter, es preciso tener una cuenta en esta red social y haber facilitado un número de móvil verificable para evitar hacer cuentas de usar y tirar.

Hay que acceder al gestor de aplicaciones de Twitter [14] y crear una nueva aplicación, al final en la configuración de la nueva aplicación, te mostrará y permitirá generar las credenciales necesarias que hay que rellenar en el archivo "config.json".


Figura 29. Gestor de aplicaciones de Twitter

Para Google Cloud [15] también hay que registrarse aunque no tiene por qué ser con una cuenta de Google. Y por supuesto también será preciso un número de móvil verificable. Habrá que crear un proyecto con el que trabajar y teniéndolo seleccionado, acceder a la pestaña de "APIs y servicios" y ahí en "Credenciales", crear unas nuevas credenciales de tipo "Clave de cuenta de servicio" y al final nos dará un archivo JSON con todas las credenciales necesarias que incluiremos en "config.json".


Figura 30. Gestor de proyectos de Google Cloud

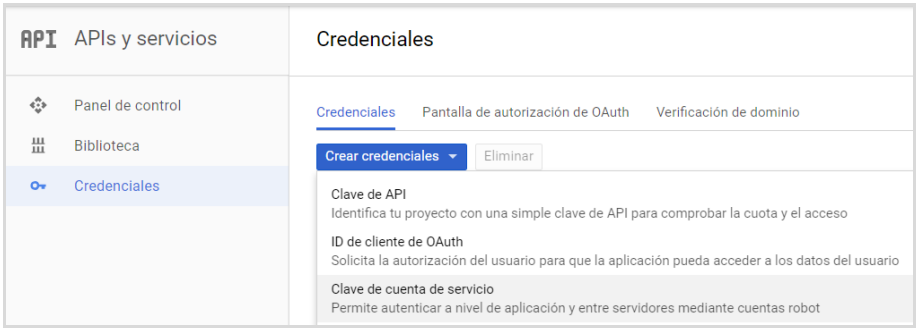



Figura 31. Credenciales de Google Cloud

## 5.4 Anexo 4: Código y puesta en marcha

El código desarrollado durante el proyecto se ha subido a GitHub y está disponible en un repositorio público [16] para consultarlo y ver más en detalle la implementación del proyecto. Puede ser clonado o descargado para replicar todo el proceso explicado y ponerlo en funcionamiento. A continuación se explica cómo.

Antes de comenzar a ejecutar nada, hay que configurar el fichero "config.json" con las credenciales necesarias para interactuar con la API de Twitter, de Google Cloud y el servidor Neo4 que se vaya a utilizar posteriormente. Escribir en el fichero "users.list" la lista de usuarios de los que se quiere recopilar sus tweets, se usa el nick sin arroba, respetando las mayúsculas y poniendo uno por línea.

Ejecutar completo el notebook "collect-tweets.ipynb". En caso de que ya se haya ejecutado previamente simplemente partirá de los tweets que ya tenga para evitar hacer más peticiones de la cuenta y de paso viene bien para ir acumulando más de los últimos 3200 tweets que permite. Puesto que el máximo de tweets por petición es de 200 y solo podemos llegar a pedir 3200 por cuenta, en teoría como mucho se llegan a hacer 16 peticiones por cuenta, pero en la práctica a veces se excede ligeramente ese límite pudiendo hacerse 17 peticiones, 18 dejando margen. Como la API permite 900 llamadas en cada ventana de 15 minutos, si queremos procesar 100 cuentas, lo que supone 1800 llamadas, habrá que consumir 2 ventanas (3 a lo sumo), o lo que es lo mismo, entre 30 y 45 minutos.

Una vez se hayan descargado los tweets en "data\tweets\", ejecutar completo el notebook "process-tweets.ipynb", que en caso de haber sido previamente ejecutado, solo procesará las nuevas cuentas, ya que cuesta tiempo y dinero. Así que si se quiere actualizar tras la descarga de nuevos tweets habrá que borrar las carpetas "data\entities\" (donde se guardan las entidades identificadas) y "data\interests\" (donde se guardan los intereses extraídos de las entidades). Como el extractor de entidades de Google Cloud también comete errores y saca algunas cosas que no concuerdan con lo esperado, se aplican algunos filtros, entre ellos una blacklist "entities_black.list" donde poner términos a ignorar, y una whitelist "entities_white.list" donde poner los términos que se pierden con el filtro de tamaño y queremos conservar. Para volver a hacer el procesado de esta parte valdrá con borrar la carpeta "data\interests\" evitando así tener que volver a extraer las entidades con Google Cloud, que cobra por peticiones, y es por ello que se han creado 2 carpetas, siendo la de entidades a modo de caché y así poder repetir el proceso de la segunda parte sin coste adicional.

Antes de continuar, habría que ejecutar los scripts de Wikidata para generar el archivo de relaciones que hay que ubicar en "data\relations\wikidata_relevant.json". Esto solo habría que hacerlo una vez a no ser que se quiera refinar el procesado.

Una vez generados los intereses en “data\interests\”, procedemos a ejecutar el notebook “relate-interests.ipynb”, que en caso de haber sido previamente ejecutado se sobrescribirán los resultados de la afinidad de cada usuario, ya que cada uno de ellos ha de ser comparado con el resto y no se pueden introducir nuevos usuarios sin recalcular los demás… La operación de calcular la afinidad entre 2 usuarios cuesta poco tiempo (unos 150 ms), pero al cruzarlos a todos ya no es insignificante. En este caso serán 100*100=10.000 cálculos ~ 25 min. Si se deshabilita el cacheo del cálculo de proximidad entre intereses, el tiempo para calcular la afinidad entre 2 usuarios asciende a unos 300 ms, lo que viene a ser el doble y tiene sentido ya que al cruzarse a todos los usuarios, se comparar los intereses de A con B y los de B con A. El cálculo de la afinidad no es simétrico, pero el de la proximidad entre los intereses sí, por ello es imperativo cachearlo, pero fijando un límite ya que puede crecer indefinidamente.

Antes de pasar a ejecutar el notebook “graph-relations.ipynb”, hay que tener corriendo el servidor de Neo4j que inicialmente hemos especificado en el fichero de configuración. Da igual si ya hemos lanzado previamente este notebook, porque antes de insertar los datos en Neo4j, borra todo lo que había para obligar a actualizar los nodos y relaciones que ya había con los nuevos valores. De esta forma podremos luego ver de forma gráfica desde el navegador (localhost:7474) la afinidad de los usuarios desglosada en las relaciones y caminos que hay entre ellos. El tiempo de inserción en base de datos no es directamente proporcional al número de usuarios, ya que la información se almacena en forma de nodos y relaciones, por tanto dependerá de cuántos intereses tenga cada usuario y cuántos de ellos y con cuántos saltos están relacionados entre sí. Obviamente el tiempo será mayor cuantos más usuarios…
En este caso el notebook no va mostrando información del progreso, pero desde la interfaz gráfica de Neo4j se puede ver cómo se va incrementado el número de usuarios, entidades y relaciones con las siguientes consultas:

```
MATCH (n:User) RETURN count(n)
MATCH (n:Entity) RETURN count(n)
MATCH ()-[r:RELATION]->() RETURN count(r)
MATCH ()-[r:AFFINITY_1|AFFINITY_2|AFFINITY_3]->() RETURN count(r)
MATCH ()-[r:INTEREST_0|INTEREST_1|INTEREST_2|INTEREST_3]->() RETURN count(r)
```

Figura 32. Consulta de número de elementos

Saliendo al final para este caso en particular:

```
100 usuarios
16033 entidades
2927 relaciones entre entidades
2841 relaciones de afinidad entre usuarios
31232 intereses de usuarios hacia entidades
```

Figura 33. Número de elementos final

# 5.5 Anexo 5: Visualización y manipulación de grafos

## 5.5.1 Acceso y visualización

Para acceder a la interfaz gráfica solo hay que abrir un navegador web e introducir la dirección localhost:7474. Las credenciales por defecto son “neo4j” como usuario y como contraseña. La primera vez solicitará el cambio de contraseña.

Una vez dentro nos dará acceso a toda la interfaz, arriba está la caja de texto para introducir consultas, y si pulsamos en la izquierda, tenemos este panel lateral:

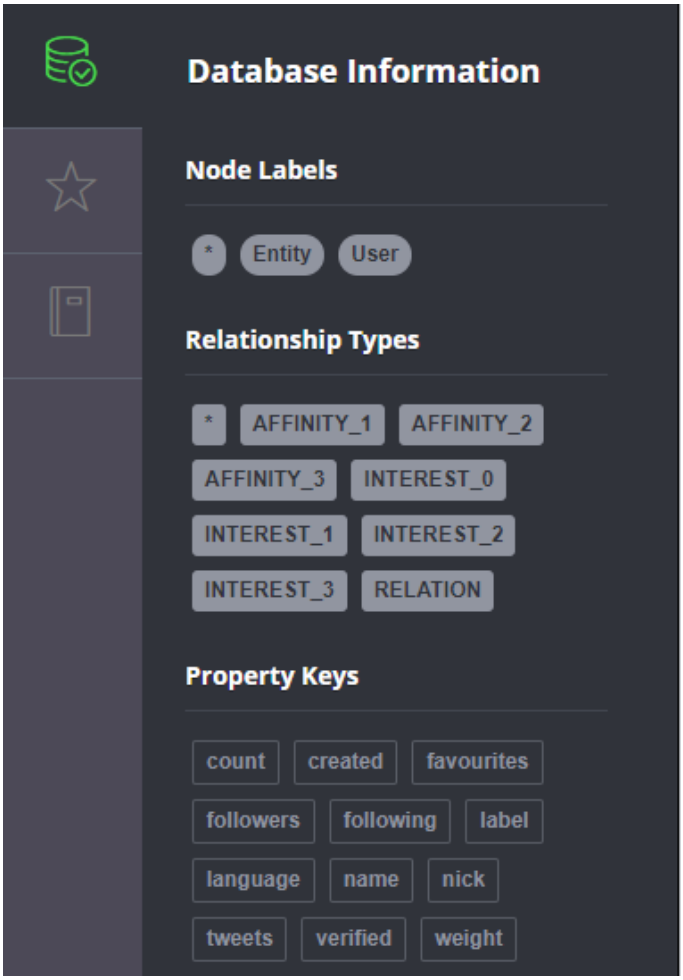



Figura 34. Panel lateral de la interfaz de Neo4j

Aquí podemos ver los tipos de nodos y relaciones que hay así como los atributos que tienen.

Antes de intentar representar nada, como son muchas las relaciones que hay, vamos a la configuración (en el símbolo de engranaje que hay en la parte inferior del panel lateral) y desmarcamos si no lo está ya la opción de abajo del todo “Connect result nodes” para evitar que la interfaz se sature representando todas las relaciones.

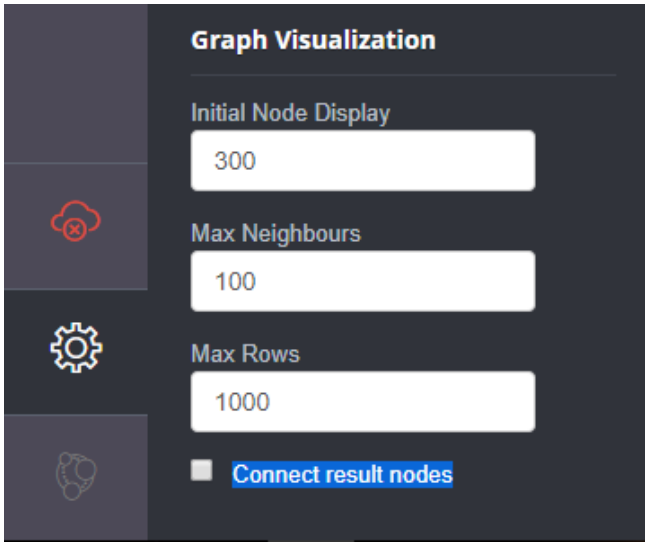


Figura 35. Configuración de la interfaz de Neo4j

Además podemos ajustar la cantidad máxima de resultados que se mostrarán, ya que puede llegar a suponer un gran consumo de RAM por parte del navegador.

Si volvemos a la primera pestaña del panel lateral y pinchamos sobre un tipo de nodo o relación nos sacará todas las que hay de ese tipo pero como puede verse en el comando que ha lanzado, lo ha limitado a 25 resultados, así que para ver todos los usuarios podemos introducir a mano el comando “MATCH (n:User) RETURN n”.

Cada consulta genera una nueva ventana y como son muchos resultados podemos verlo a “ventana completa” dándole a las opciones de la ventana, y luego nos saldrán abajo a la derecha unas lupas para poder cambiar el zoom. Así podremos ver los 100 usuarios que tenemos:

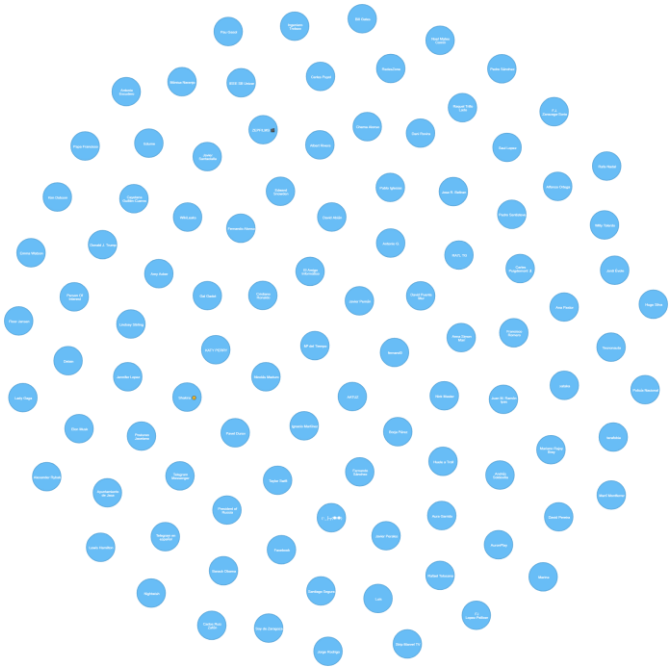


Figura 36. Visualización de todos los usuarios

Las opciones de visualización que vienen por defecto no son las más apropiadas para la correcta visualización, y como se guardan en la caché del navegador de cada usuario, se va a explicar cómo configurarlo a continuación.

Ejecutando el comando “CALL db.schema()” se mostrará la estructura de los distintos tipos de nodos y relaciones que hay.

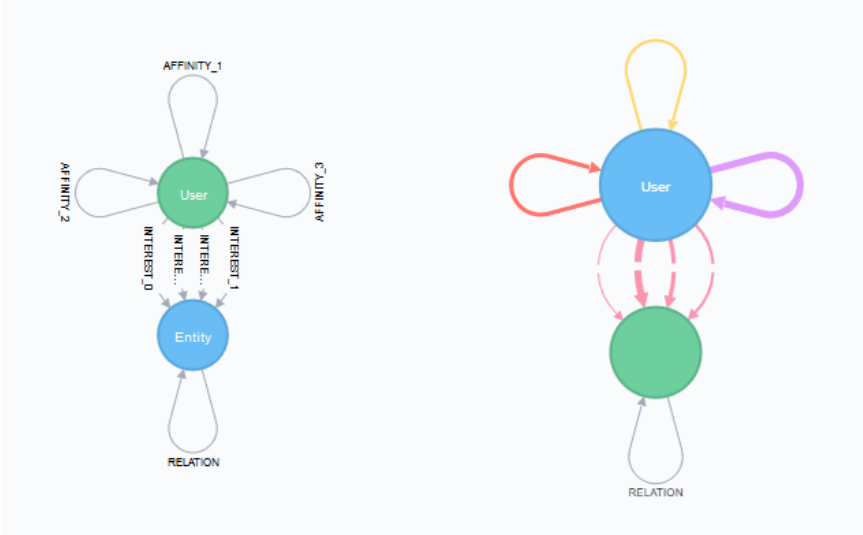



Figura 37. Comparativa estilos de visualización

Si no se ha configurado nada, seguramente se verá algo como lo de la izquierda, y para que sea más legible e intuitivo, debería verse algo como lo de la derecha.

Para cambiar la forma en la que se representa cada objeto basta con pinchar en ese tipo de nodo o relación en las etiquetas que hay en la parte superior de la ventana y se mostrarán las opciones en la parte inferior:


Figura 38. Etiquetas de la parte superior


Figura 39. Opciones de la parte inferior para un nodo


Figura 40. Opciones de la parte inferior para una relación

Tanto en nodos como relaciones se puede escoger entre 7 colores, y luego hay distintos grosores para cada uno, y por supuesto el rótulo a mostrar dependerá de los atributos definidos para cada tipo de objeto. En esta representación en particular solo muestra los atributos por defecto, pero con el resultado de cualquier otra consulta (como la de sacar todos los elementos de un tipo) sí que se podrá cambiar.

A continuación se dejan los valores que se han utilizado para la visualización:

| | Color | Size | Caption |
|---|---|---|---|
| **User** | blue | 5 | name |
| **Entity** | green | 4 | label |
| **AFFINITY_1** | yellow | 2 | weight |
| **AFFINITY_2** | red | 3 | weight |
| **AFFINITY_3** | purple | 4 | weight |
| **INTEREST_0** | pink | 1 | weight |
| **INTEREST_1** | pink | 2 | weight |
| **INTEREST_2** | pink | 3 | weight |
| **INTEREST_3** | pink | 4 | weight |
| **RELATION** | grey | 1 | <type> |

Tabla 7. Valores de visualización

Ahora que ya estamos listos para representar correctamente los grafos, vamos a empezar por ver los usuarios con mayor afinidad entre ellos ejecutando la consulta "MATCH p=()-[r:AFFINITY_3]->() RETURN p"

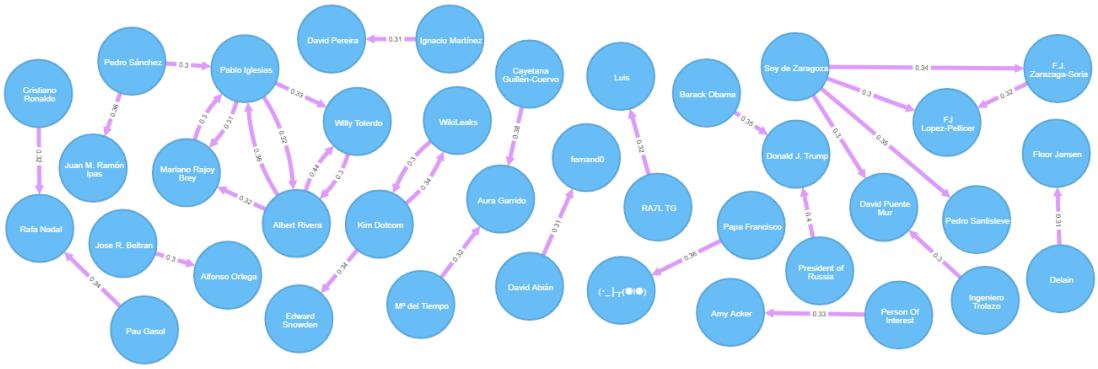



Figura 41. Las mayores relaciones de afinidad

Si se pincha sobre un usuario en particular saldrá un aro con 3 opciones, una de ellas es la de expandir, que añadirá a la actual representación todas las relaciones de ese usuario, y lo mismo ocurre con las entidades.

Pero como son tantas las relaciones que puede llegar a haber, es mejor hacer consultas concretas para mostrar exactamente lo que queramos.

### 5.5.2 Manipulación y consultas

A continuación se explica resumidamente cómo formular las consultas.

Como es de imaginar, “MATCH” se utiliza siempre para realizar búsquedas en la base de datos y “RETURN” para visualizar los resultados que queremos, puede ser un path entero o uno o varios nodos y relaciones.

Los nodos se representan con “()” y las relaciones con “[]”. Si se dejan vacías puede ser cualquiera. Si se quiere especificar el tipo se usa “:” por ejemplo “(:User)” o “[:AFFINITY_3]”, y si se quieren especificar varios tipos se usa “|” como separador, por ejemplo “[:AFFINITY_1| AFFINITY_2|AFFINITY_3]”. Si además se quiere recoger el valor en una variable se pone delante de “:” o el nombre de la variable a secas si no importa el tipo, por ejemplo “(u:User)”, “(u)”, “[r:AFFINITY_1| AFFINITY_2|AFFINITY_3]” o “[r]”, así podríamos devolver solo “u” o “r”.

Para unir los nodos con las relaciones se usa “-“ o “->” si se quiere fijar el sentido de la relación. Entre 2 nodos siempre tiene que haber una relación, y lo mismo pasa con las relaciones, tiene que haber uno entre ellas. Todavía se puede filtrar más fino si es preciso fijando los atributos de los nodos y las relaciones, por ejemplo “(n:User {nick:'david12_45'})” o incluso usando el operador “WHERE” como aquí “MATCH (n:User {verified:true}) WHERE n.followers >= 50000000 RETURN n”


Figura 42. Usuarios verificados con más seguidores

Además de representar de forma gráfica, también se pueden hacer consultas de datos concretos, como por ejemplo el número de elementos de cada tipo:

```
MATCH (n:User) RETURN count(n)
MATCH (n:Entity) RETURN count(n)
MATCH ()-[r:RELATION]->() RETURN count(r)
MATCH ()-[r:AFFINITY_1|AFFINITY_2|AFFINITY_3]->() RETURN count(r)
MATCH ()-[r:INTEREST_0|INTEREST_1|INTEREST_2|INTEREST_3]->() RETURN count(r)
```

Figura 43. Consulta de número de elementos

Para terminar se van a dejar algunas consultas curiosas o a modo de ejemplo y distintas formas de formularlas, así como los resultados de algunas de ellas:

```
// Highest users affinities
MATCH p=()-[:AFFINITY_3]-() RETURN p
MATCH (a:User)-[r:AFFINITY_1|AFFINITY_2|AFFINITY_3]->(b:User) WHERE r.weight >= 0.30 RETURN a,r,b

// A user relevant affinities
MATCH p=(:User {nick:'david12_45'})-[:AFFINITY_2|AFFINITY_3]-() RETURN p
MATCH p=(a:User)-[r]-(:User) WHERE a.nick='david12_45' AND r.weight>=0.20 RETURN p

// Affinity between 2 users
MATCH p=(:User {nick:'david12_45'})-[]-(:User {nick:'fernand0'}) RETURN p

// Affinity hops between 2 users
MATCH p=(:User {nick:'david12_45'})-[:AFFINITY_2|AFFINITY_3*..2]-(:User {nick:'fernand0'}) RETURN p
```


Figura 44. Usuarios afines en común entre 2 usuarios

```
// Corresponded relations
MATCH p=(a:User)<-[:AFFINITY_2|AFFINITY_3]->(b:User)<-[:AFFINITY_2|AFFINITY_3]->(a:User) RETURN p
// Search a specific user
MATCH (n:User {nick:'david12_45'}) return n

// Search a specific entity
MATCH (e:Entity {label:'EINA'}) return e
```

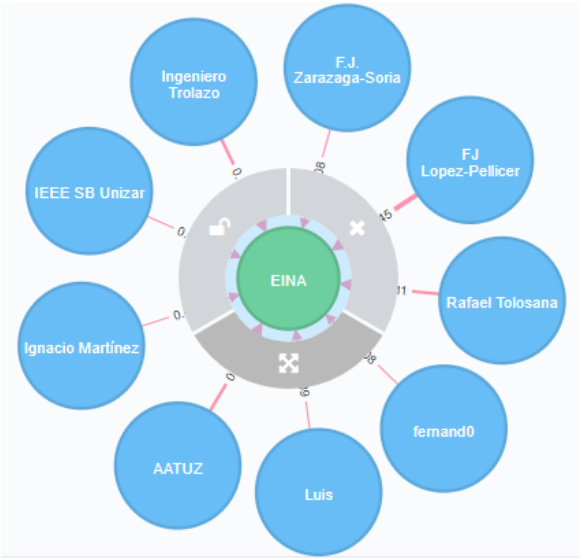



Figura 45. Expansión de una entidad

// A user relevant interests
MATCH p=(:User {nick:'elonmusk'})-[:INTEREST_2|INTEREST_3]->() RETURN p


Figura 46. Intereses relevantes de un usuario

// Highest users interests
MATCH p=()-[r:INTEREST_3]->() RETURN p

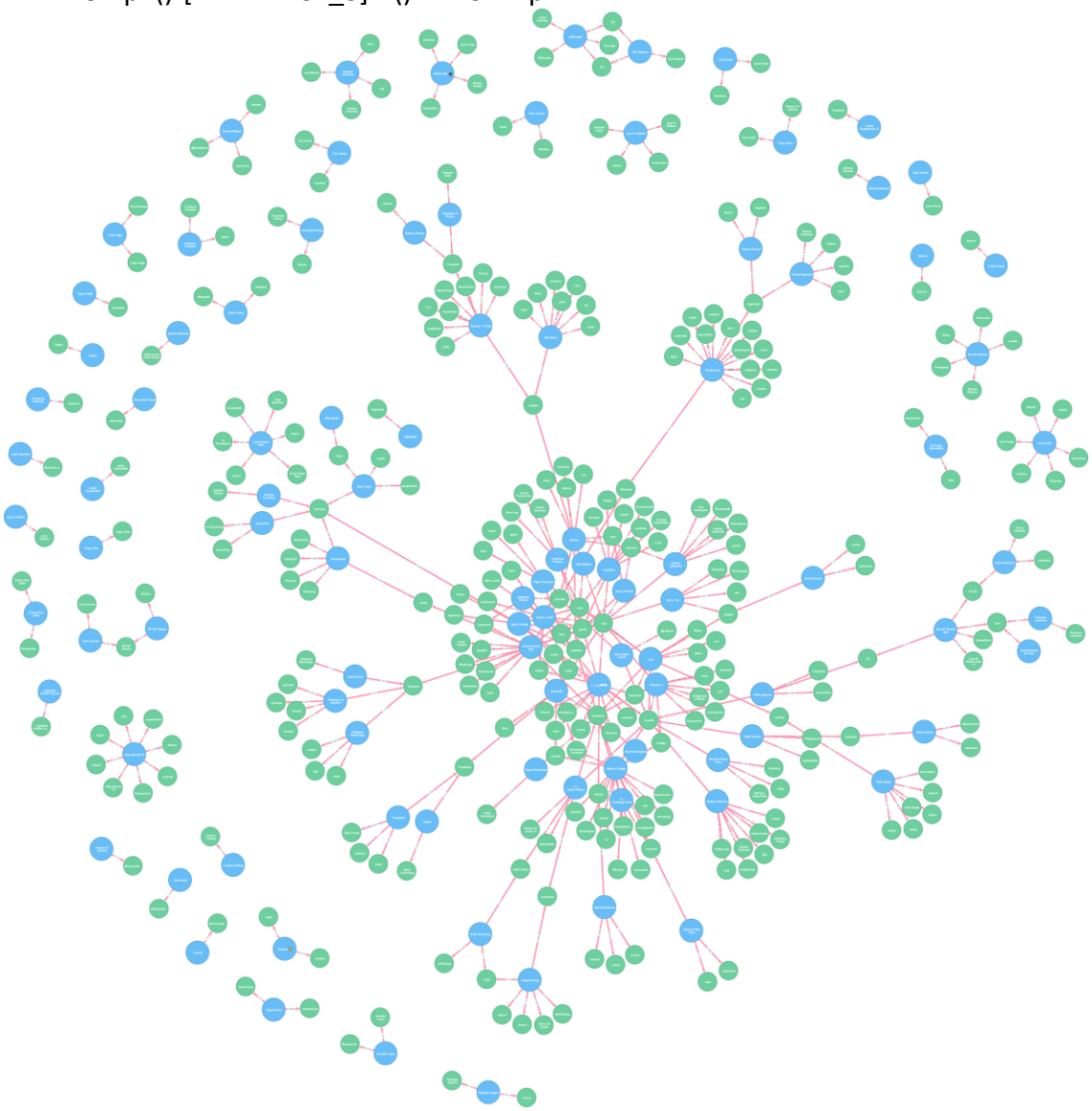


Figura 47. Los intereses más relevantes de todos los usuarios

// Interests between 2 users
MATCH p=(:User {nick:'marianorajoy'})-[:INTEREST_1|INTEREST_2|INTEREST_3*..2]-(:User {nick:'Pablo_Iglesias_'}) RETURN p


Figura 48. Intereses en común entre 2 usuarios

// Interests between 3 users
MATCH p=(a:User)-[:INTEREST_1|INTEREST_2|INTEREST_3]-(:Entity)-[:INTEREST_1|INTEREST_2|INTEREST_3]-(b:User) WHERE a.nick IN ['marianorajoy', 'Pablo_Iglesias_', 'sanchezcastejon'] and b.nick IN ['marianorajoy', 'Pablo_Iglesias_', 'sanchezcastejon'] RETURN p

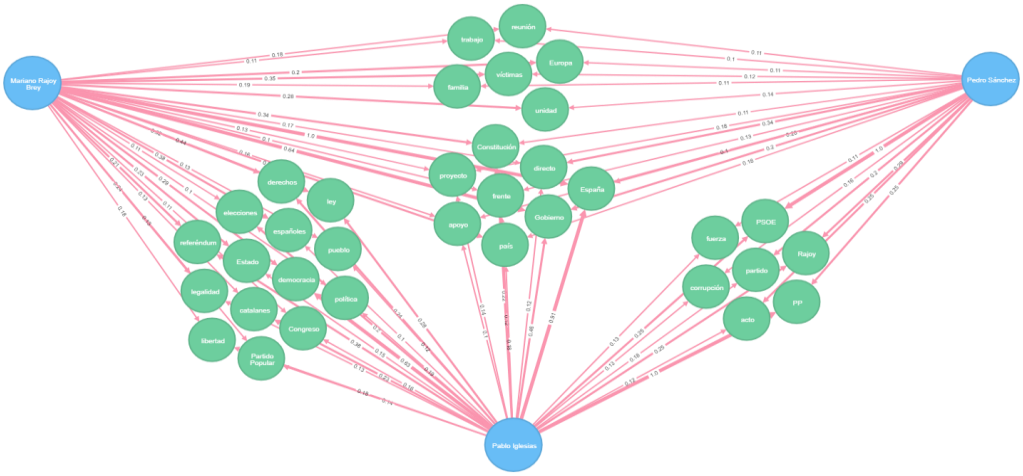



Figura 49. Intereses en común entre 3 usuarios

// Interests between 3 users
MATCH p=(a:User)-[:INTEREST_1|INTEREST_2|INTEREST_3]-(:Entity)-[:INTEREST_1|INTEREST_2|INTEREST_3]-(b:User) WHERE a.nick IN ['marianorajoy', 'Pablo_Iglesias_', 'sanchezcastejon', 'Albert_Rivera'] and b.nick IN ['marianorajoy', 'Pablo_Iglesias_', 'sanchezcastejon', 'Albert_Rivera'] RETURN p

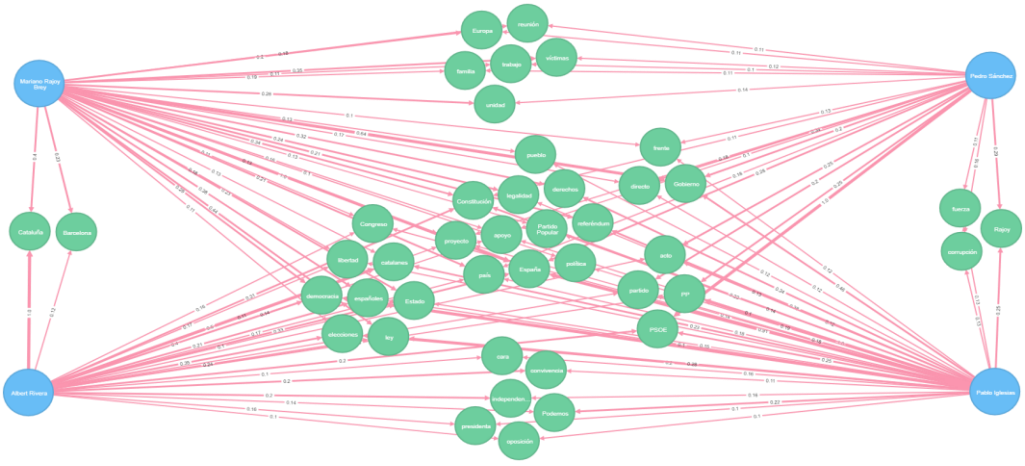



Figura 50. Intereses en común entre 4 usuarios

// Indirect interests between users
MATCH p=(:User)-[:INTEREST_2|INTEREST_3]-(:Entity)-[:RELATION]-(:Entity)-[:INTEREST_2|INTEREST_3]-(:User) RETURN p

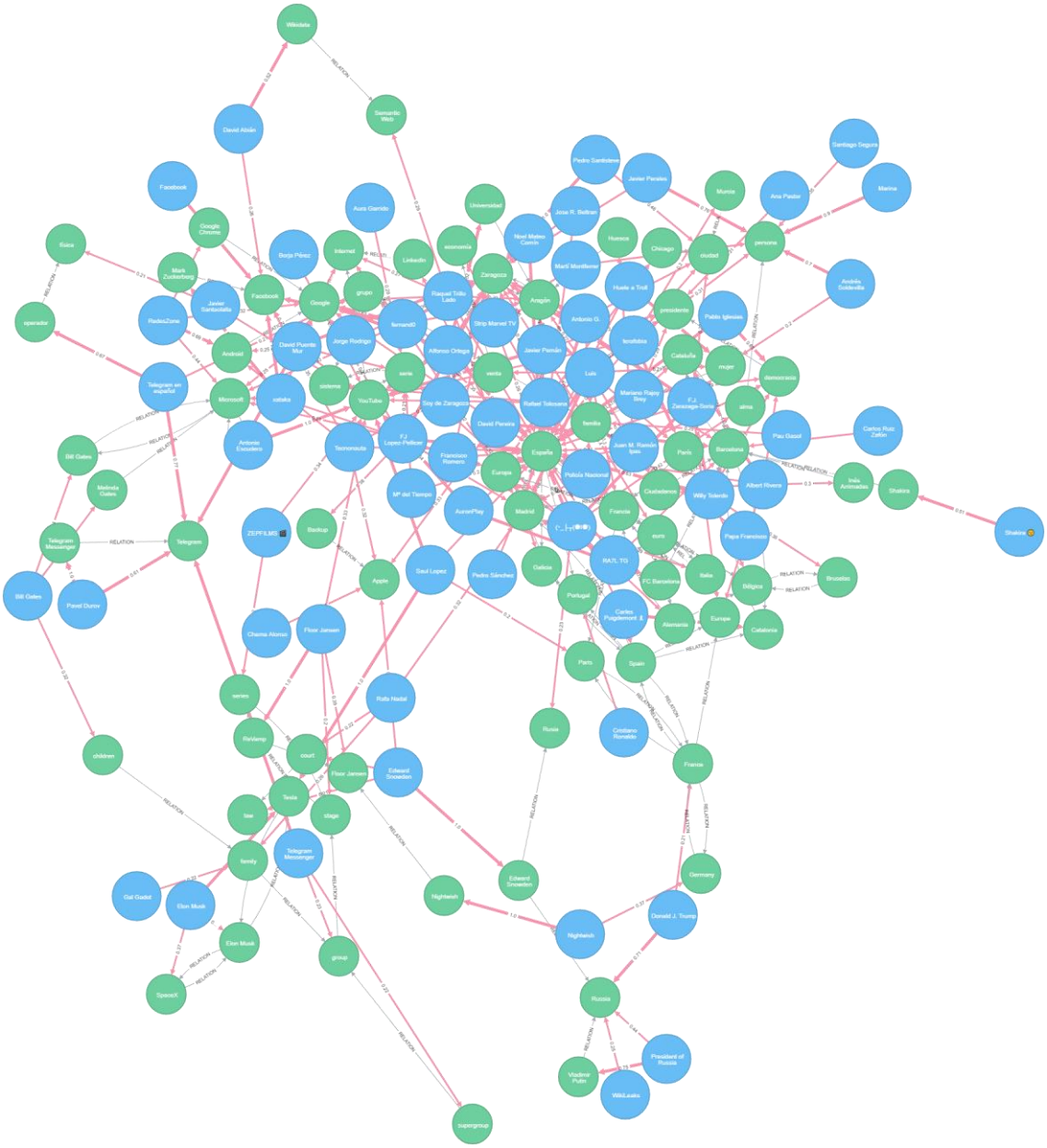


Figura 51. Usuarios relaciones a través de relaciones indirectas entre entidades

```
// Related entities
MATCH p=(:Entity)-[r:RELATION]->(:Entity) RETURN p
```

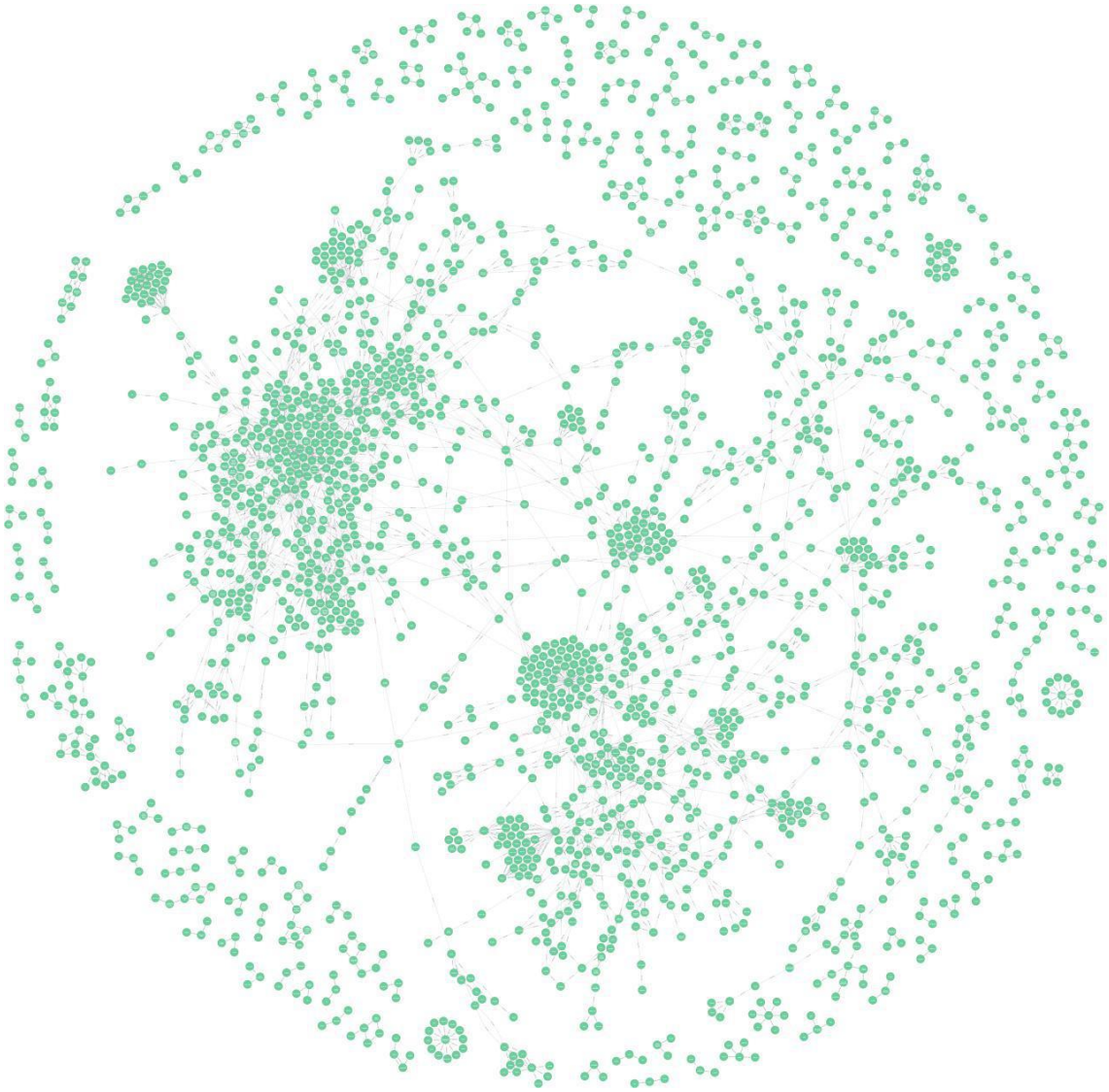


Figura 52. Todas las entidades relacionadas entre sí