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 tuits, se usa el nick sin arroba, respetando las mayúsculas y poniendo uno por línea. De todas formas al comprobar si la cuenta existe parece que la API no es tan quisquillosa y lo acepta igualmente.

Ejecutar completo el notebook “collect-tweets.ipynb”, en caso de que ya se haya ejecutado previamente simplemente partirá de los tuits que ya tenga para evitar hacer más peticiones de la cuenta y de paso viene bien para ir acumulando más de los últimos 3200 tuits que permite.

Puesto que el máximo de tuits 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), 30’-45’

Hay que mencionar que es complicado sobrepasar las 900 llamadas en 15 minutos, ya que eso supone 1 petición por segundo, y al menos con mi conexión cuesta entre 1 y 2 segundos.

Una vez se hayan descargado los tuits 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 tuits 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 gcloud no es perfecto 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 gcloud, 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.

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](http://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:

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)

Saliendo al final un total de:

100 usuarios

16.033 entidades

2.927 relaciones entre entidades

2.841 relaciones de afinidad entre usuarios

31.232 intereses de usuarios hacia entidades