Estrategia para la recuperación de tuits diarios por país

Recuperación de información y recomendaciones en la web 2017
Docente: Libertad Tansini
Facultad de Ingeniería
Universidad de la República

Matías Lugli - matiaslugli08@gmail.com
Mauro Picó - mpico@fing.edu.uy || mauro@enia.io

Introducción

En el marco de un trabajo en conjunto con investigadores de las áreas de economía, sociología y estadística aplicada, se trabaja en la tarea de recuperar tuits generados dentro de diez países de América Latina y el Caribe. Para esto trabajamos realizando una comparación entre la API de búsqueda y la API de streaming de Twitter, para luego consumir información de alguna de ellas con el fin de recuperar los documentos. Estos tuits serán utilizados para construir un modelo hedónico que permita cuantificar el sentimiento asociado a la información. El modelo estadístico que permite realizar el análisis de sentimiento es privativo y el mismo no será abordado en este trabajo.

La recuperación de la información del corpus de Twitter, se encuentra sujeta a un conjunto de restricciones las cuales se deberán de congeniar en un prototipo funcional que logre realizar las capturas. Las restricciones del problema se listan a continuación:

- Se cuenta con una infraestructura básica. Servidor físico con linux, con conectividad limitada a la conectividad de un hogar. Con 500Gb de espacio de almacenaje y 4Gb de memoria RAM.
- Se deben de recuperar la mayor cantidad de tuits posibles. Se estima que será necesario recuperar en el entorno de 10.000 tuits para países de la escala de Uruguay en cuanto a cantidad de habitantes.
- Los tuits deberán estar distribuidos a lo largo de las 24 horas del día, durante un período de 180 días.
- Se debe recuperar información de 10 países de América Latina: Uruguay, Argentina, Chile, Paraguay, Colombia, Bolivia, Perú, Ecuador, Venezuela y México.

Luego de la conformación de un prototipo funcional para la captura de datos que deberá funcionar durante un período de 6 meses continuos, se trabaja en el diseño de una Arquitectura de la aplicación, que permite escalar rápidamente para ser utilizado en modelos comerciales.

APIs de Twitter

A continuación se realizará una descripción de la API de streaming y de la API de búsqueda de Twitter a partir de la investigación realizada para cada una de ellas. Así también se decidirá, en función de los resultados de la investigación y de las restricciones del problema, cuál será la más apropiada para realizar el trabajo.

API de streaming

Esta API permite obtener en tiempo real los tuits que los usuarios van generando. En particular, este enfoque es el que permitiría obtener mayor cantidad de documentos. Cuando se consulta a la API, se genera una conexión HTTP persistente, y la API devuelve constantemente tuits. Así mismo, si se intenta abrir una nueva conexión, desde la misma IP origen, la conexión actual vigente se cerrará dejando paso a la nueva conexión persistente.[1]

Nos interesa evaluar cómo la API devuelve tweets de determinadas regiones. La API de Streaming devuelve datos en función de un parámetro llamado *location*, el cual asume valores separados por coma de la siguiente manera -122.75,36.8,-121.75,37.8. Estos valores definen los vértices de una rectángulo en una región espacial, y la API devolverá aquellos tuits que hayan sido originados dentro de esos límites. Particularmente se devolverán únicamente aquellos tuits que fueron geo referenciados a la hora de ser publicados por el usuario (esto se configura como una opción a la hora de publicar el tuit), utilizando la siguiente heurística interna de la api:

- Si el campo "coordinates" del tuit está populado, se verifica que el mismo esté en el rectángulo definido y si es así, se devuelve.

- Si el campo "coordinates" no está populado, se verificará que el campo "place" tenga información, ese campo define una región, y si esa región tiene intersección alguna con el rectángulo definido en el parámetro "location" el tweet se devuelve.

- Si ninguno de estos campos tiene información, el tweet no será devuelto.

En previas versiones de la API, el campo "geo" se verificaba en el tercer punto del algortimo mencionado en los puntos anteriores, antes de descartar el tweet. En la versión actual este campo está discontinuado.

Esta API está orientada a la completitud, devolviendo todos los tuits que sean relevantes para Twitter (no está claro el significado de "relevante" para twitter, ya que no se aclara en la documentación oficial a qué se refiere con eso). Existe otro enfoque no orientado a la completitud que es el utilizado por la API de Búsqueda.

---

[1] https://dev.twitter.com/streaming/overview - accedido el 2017-04-25

API de búsqueda

Otro estrategia para recuperar tweets, es utilizando la API de búsqueda. Esta API permite realizar la búsqueda de la misma manera que se podría realizar en el buscador del sitio de Twitter, agregando beneficios de parámetros temporales y espaciales, entre otros.

En primer lugar, esta API no requiere de una conexión HTTP Persistente y soporta varias conexiones simultáneas, únicamente afectadas por los Rate Limits[2] que son explicados en la sección siguiente.

Así también, se pueden obtener tweets de zonas definidas mediante el parámetro 'geocode', este parámetro tiene que ser definido con una latitud y una longitud que representarán un punto, y una distancia en millas o kilómetros, que define el radio de un círculo concéntrico en el punto. La solicitud devolverá todos los tweets que se encuentren dentro de ese rango y cumplan con las propiedades de los otros parámetros.

Dentro de esos otros parámetros, encontramos uno de particular valor, `query`, parámetro por el cual se envían consultas que pueden ser implementadas a mano. Por ejemplo, si se quiere que la API devuelva todos los tweets que contengan la palabra "X", se deberá llamar a la API con q="X". Así mismo el parámetro puede ser utilizado para consultas más complejas que contengan operadores lógicos, por ejemplo q="X OR Y".

También podemos encontrar parámetros que definen fechas específicas para la captura de datos. Por ejemplo, `since` determinará la fecha desde la cual se comenzarán a tomar datos inclusive, 'until' determinará la fecha hasta la cual se tomarán datos inclusive. La fecha deberá de ser pasada con el siguiente formato para ambos parámetros: "AAAA-MM-DD". En caso de no definir un límite superior con 'until', la API devolverá resultados desde la fecha definida en 'since' hasta el momento de la consulta a la API. En caso de no definir 'since' ni 'until', la API traerá datos desde 7-9 días hacia atrás desde el momento de la consulta, hasta el momento de la consulta. El parámetro 'result_type', determina el orden de captura, si este parámetro tiene el valor 'recent', se traerán los tuits más recientes primero.

El parámetro 'lang=es' determina tweets en idioma español.

A diferencia de la API de streaming que está enfocada a la completitud, la API de búsqueda devuelve aquellos tweets que Twitter entiende relevantes, los cuales son un muestreo del universo total de tweets. El criterio con el que se realiza este muestreo no está especificado en la documentación oficial de Twitter.

[2] https://developer.twitter.com/en/docs/basics/rate-limiting - accedido

Rate Limits y Autenticación

En particular, a la hora de realizar llamadas a la API de búsqueda, se deberá tener en cuenta que la cantidad de requests están limitados a determinado número por ventana, y cada ventana es de 15 minutos. Al alcanzar los límites la API responde con el mensaje:

```
{"errors":[{ "code": 88, "message": "Rate limit exceeded"}]}
```

Este puede ser utilizado para monitorear el rate limit y detener la aplicación por el tiempo que sea necesario.

La cantidad de request aceptados depende del endpoint REST que se esté utilizando. En particular, método `GET search/tweets` por ventana de los 15 minutos puede aceptar hasta 450 request para las aplicaciones autenticadas en la API, y 150 para los usuarios autenticados en la API.

La API acepta autenticación como usuario de Twitter y autenticación como aplicación registrada por un usuario de Twitter. Los servicios de autenticación usan OAuth como servicio de autenticación.[3]

API de búsqueda y estrategia de captura

En la siguiente sección se explicita por qué se realiza la elección de la API de búsqueda para llevar adelante el desarrollo y la forma en la cual se realizará para cumplir con las restricciones definidas, así como pruebas realizadas mediante la construcción de la estrategia.

Consideraciones generales

En cuanto a la infraestructura, contamos con conectividad de un hogar, lo que implica que contamos con una única IP dinámica, lo cual es un problema a la hora de paralelizar capturas de diez países de forma simultánea. En caso de utilizar la API de streaming, deberíamos de tener diez conexiones persistentes, provenientes de IPs diferentes, que deberían de recuperar información constante a lo largo de las 24 horas del día, lo cual excede ampliamente las características de la infraestructura. Esto ha sido determinante a la hora de la elección de la API de búsqueda para llevar adelante el desarrollo.

La contracara de la API de búsqueda es que la misma no está apuntada a obtener completitud de tuits (cantidad), sino que apunta a relevancia, con un criterio poco transparente en la documentación oficial. Así mismo, si bien las conexiones entre nuestro sistema y la API no son persistentes, la API acumula las cantidades recuperadas de tuits

[3] https://oauth.net/

desde la misma IP-ID de aplicación, lo cual implica que no se podrán obtener datos de forma paralela.

Con el fin de obtener tuits suficientes distribuidos durante las 24 horas del día, la API de búsqueda es de utilidad dado que a la misma, se le pueden consultar rangos de fechas. Para un rango de 24 horas, mediante esta API, se obtienen 360000, lo que hace inviable que la misma sea utilizada a partir de la consulta por fechas, si tenemos en cuenta que Uruguay es de los países más pequeños, y se esperan mínimo 10 veces más para países como Argentina.

Para obtener tuits georeferenciados, la api de búsqueda permite definir superficies geográficas circulares, obteniendo aquellos tuits relevantes (según un criterio desconocido aplicado por Twitter) dentro de esa zona. Esto puede ser utilizado para, de una manera especial, obtener tuits asociados a un determinado país.

Lo expuesto anteriormente sumado a que la API de Búsqueda también está sometida a los límites de captura, lleva a diseñar una estrategia de captura particular para así cumplir con las restricciones.

Estrategia de Captura y Pruebas

Almacenar la cantidad total de tuits obtenidos mediante un llamado solicitando información dentro de determinado rango de fechas, es inviable puesto que nuestras estimaciones iniciales, indican que almacenar el total de tuits que se obtienen ocuparían un total aproximado de 3.2Gb por día para todos los países objetivo.

Por este motivo se decide almacenar tuits en intervalos de tiempo, esto es, se realizará un muestreo de 5 minutos de tuits de cada país, aproximadamente cada 3 horas. Este intervalo de tiempo (las 3 horas entre muestreos) es aproximado porque depende de la ejecución del de los scrapers. Por limitaciones presupuestales y de tiempo, simplemente se iteran las capturas sobre los países, uno a uno, al culminar, se retoma nuevamente las capturas desde el país inicial.

Para agregar variabilidad, el momento de captura no es el mismo para cada país. Es decir, un ciclo de captura involucra la ejecución del scraper para 10 países, el orden en el cual se obtienen los datos de cada país se elije de manera aleatoria. Lo que sí, respetamos que el intervalo analizado de 5 minutos sea el mismo intervalo de tiempo en todos los países, para así tener congelado el estado de sentimiento de los 10 países en el mismo momento.

Para explicar mejor el ciclo de captura, presentamos el llamado a la api, el cual se observa a continuación:

```
new_tweets = api.search(q=query, geocode=geocode, count=count, max_id=str(last_id - 1),
result_type='recent')
```

El primer parámetro `query` se encuentra vacío durante toda la ejecución, ya que no se realiza una query exacta para traer los documentos. Se utiliza el comportamiento por defecto de la api, la cual obtiene todos los documentos sin aplicar filtros. Por otro lado, el parámetro `geocode` es una string con la siguiente forma

```
'-36.717554,-60.317591,320km'
```

En ese string se observan en los dos primeros lugares, latitud y longitud que definen un punto en el planeta, y el último lugar, define un radio. Esto es utilizado para definir un círculo en el espacio para el cual se recuperará la información. Para la captura, se definen círculos que cubren de manera estratégica cada país, conteniendo ciudades principales manteniendo cuidado de no salir de las fronteras. La figura 1 muestra la primera aproximación en base a círculos realizada para Argentina.

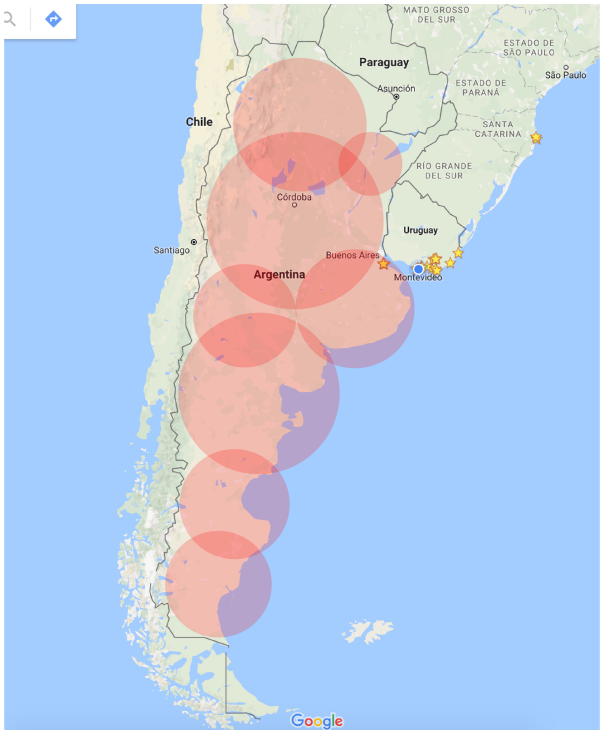

Figura 1 - Primera aproximación de círculos para Argentina, teniendo en cuenta las principales ciudades de interés. Se puede descargar el mapa dinámico en la referencia[4]

A medida que se fue generando la estrategia de captura, se fueron realizando pruebas para comprobar que la misma era adecuada para los objetivos planteados. Se realizaron cinco oleadas de captura de alrededor de 2.000 tuits por círculo cada una. Es decir, se le solicitó a Twitter que devuelva documentos hacia atrás en el tiempo desde el momento en que arranca la captura y se interrumpió la ejecución cuando se alcanza dicho número de capturas. En los círculos en los cuales hay una menor densidad de publicaciones, Twitter nos devolvió una muestra de mayor amplitud en el tiempo. Esto se puede ver en la gráfica de la figura 2.

[4] https://drive.google.com/file/d/1xJjjfci-1DWlCM7fvqbkogQBvecUteQG/view?usp=sharing

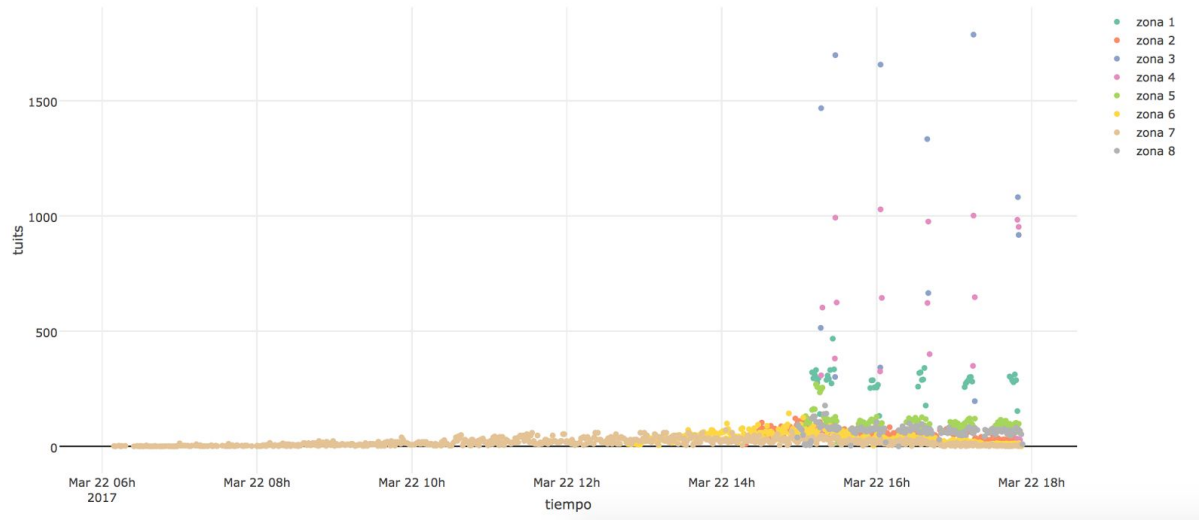

Figura 2 - Gráfica de primeras capturas en argentina

Otra cuestión que se detectó fue el importante número de tuits duplicados que se recibieron. Esto tiene que ver con dos cosas. Una de ellas, la yuxtaposición de círculos y la otra (y más importante) es que en las zonas en las cuales la densidad de tuits es muy baja, las distintas capturas (espaciadas en alrededor de media hora) cubrían lapsos de tiempo yuxtapuestos. Es decir, para agotar los 2.000 tuits que le pedíamos, Twitter tenía que ir más y más atrás en el tiempo, lo cual lo obligaba a barrer varias veces el mismo período. Una vez eliminadas las publicaciones duplicadas (el tuit original se asignó a la primera zona ordenada de menor a mayor en su numeración) la base se redujo de 93.819 registros a 56.578 (-40%).

Esta es la distribución por zona de los tuits, como porcentaje del total, antes y después de este proceso:

Var1	Freq
1	13.1
2	12.6
3	12.8
4	11.7
5	12.8
6	12.8
7	12.8
8	11.4

Var1	Freq
1	21.2
2	9.5
3	21.1
4	10.4
5	19.5
6	6.5
7	4.9
8	6.9

Figura 3 - Distribución por zona de tuits antes y después de eliminar duplicados

Se aprecia claramente que el tamaño del círculo y sobre todo, la densidad de tuits esperados para cada zona afectan su distribución una vez que se depuran los duplicados.

Para evitar la captura de elementos duplicados se redefinieron los círculos, de tal manera que los mismos sean disjuntos. Esto se muestra en la figura 4.

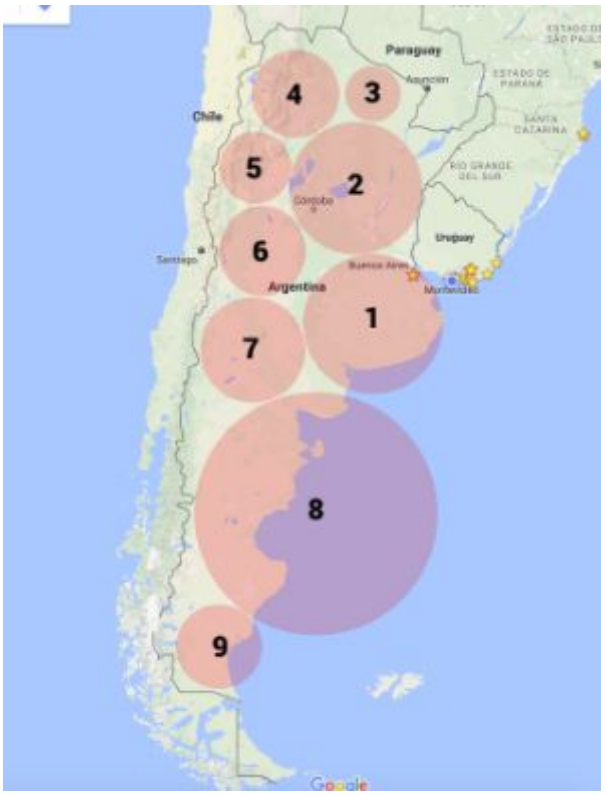

Figura 4 - Aproximación de círculos para Argentina, teniendo en cuenta las principales ciudades de interés.

Con esta aproximación, se eliminaron los duplicados. Además se dejó en ejecución el scraper, por un período de 24 horas atendiendo al muestreo antes planteado, para comprobar la cantidad total de tuits obtenidos en este período de tiempo obteniendo por sobre 100000 tuits, como nuestro benchmark para la cantidad de tuits es Uruguay, realizamos el mismo procedimiento para Uruguay, el cual nos da una cantidad por sobre los 11000 tuits, cantidad suficiente para realizar el análisis estadístico inicial.

Nos enfrentamos a una dificultad en este punto, de naturaleza técnica, con el manejo de la RAM. El framework de tweepy define una clase Cursor, que es la que facilita la iteración en los resultados, manteniendo el control de qué documentos recuperar. En esencia, Twitter requiere que se le diga, no únicamente la cantidad de documentos a recuperar, sino a partir de qué id de documento obtenerlos. Esta información es manejada de manera automática por la clase dicha del framework. El problema es que esta clase, almacena en RAM los resultados obtenidos de la API, no solamente la información básica asociada al tuit, sino que todos los metadatos. haciendo que para capturas extensas, se superaran los límites de nuestros escasos recursos, lo que llegó a realizar la implementación de nuestra propia

gestión de la iteración en resultados, solicitando ‘a mano’ los id de cada documento, agregando complejidad a la solución. Esto es un bug conocido de tweepy [5]

Luego de estabilizado el sistema y demonizado, dejamos ejecutando durante 5 días los scrapers, para analizar su comportamiento y además analizar los datos obtenidos. Los resultados de esta ejecución se muestra a continuación en la figura 5:

Argentina	Bolivia	Chile	Colombia	Ecuador	México
84304	5240	28325	71915	10437	121892
99122	4998	27832	64732	9930	107482
104721	5291	42671	81709	9800	119614
105886	5212	27058	61444	8414	105984
98579	5115	32730	82811	10258	130012

Perú	Paraguay	Uruguay	Venezuela	Total tuits
17126	8067	12256	36206	395768
14397	5446	11935	29466	375340
16946	7187	14291	42750	444980
15263	6677	11553	36610	384101
15773	9340	12945	43569	441132

Figura 5 - Totales de Captura para 5 días de todos los países objetivo

Los números en general van de acuerdo a nuestras expectativas. Nos llama la atención el caso de chile, para el cual esperamos aproximadamente 5 veces más que los tuits obtenidos para uruguay dada la penetración de Twitter similar y la población 5 veces mayor. Ante esto se realizaron pruebas asumiendo que tal vez se encontraba algún servidor caché intermedio que hacía que las muestras de los países no fueran representativas, para esto se realizaron pruebas desde servidores externos (AWS EC2) creados específicamente para esto, los cuales fueron eliminados luegos de su uso para evitar gastos desproporcionados. Los resultados fueron similares. Al momento no hemos podido explicar el fenómeno de Chile, y el mismo ha quedado publicado en foros de asistencia[6].

De todas formas, las cantidades de documentos obtenidas, son suficientes para realizar los ajustes sobre los modelos hedónicos. La figura 6 muestra un mes de ejecución del scraper para Uruguay, la figura 7 muestra el mismo mes para Argentina.

[5] https://github.com/tweepy/tweepy/issues/714

[6] https://stackoverflow.com/questions/43614115/tweeter-api-returns-low-quantity-of-tweets-for-south-america-chile

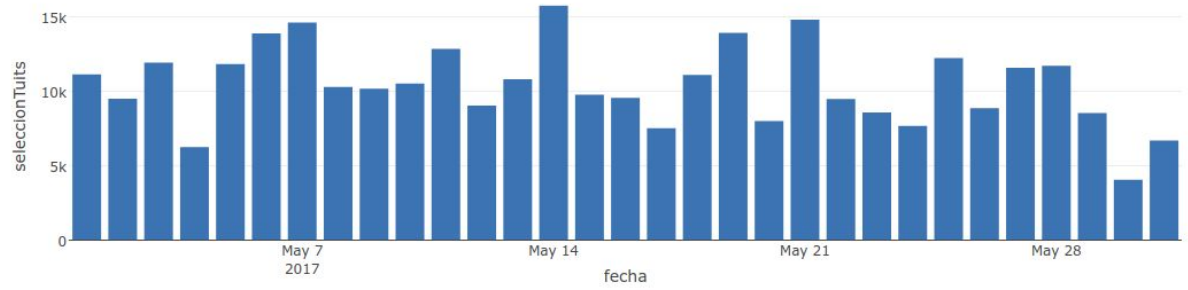

Figura 6 - Capturas correspondientes al mes de Mayo para Uruguay

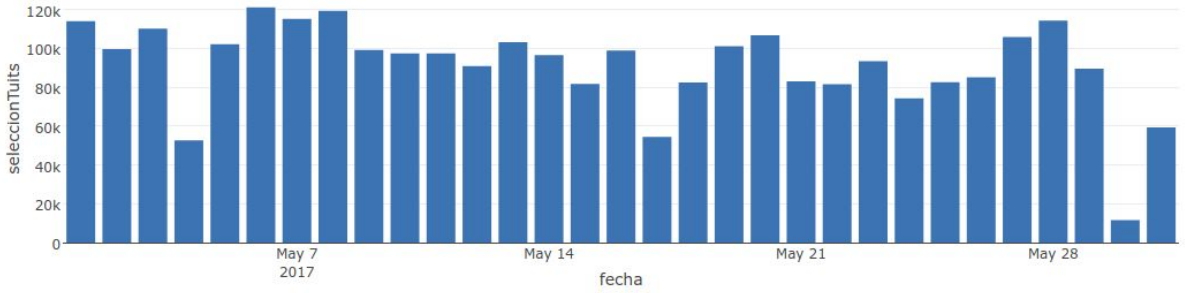

Figura 7 - Capturas correspondientes al mes de Mayo para Argentina

Sobre las arquitecturas

A continuación se plantea una descripción de la arquitectura del sistema como prototipo, esta arquitectura refleja los componentes de software y bases de datos utilizadas, que se encuentran deployadas en un servidor único, local, básico. Así mismo se presentará una propuesta de arquitectura distribuida que permita escalar el sistema.

Arquitectura Actual

La figura 8 muestra el diagrama de arquitectura del sistema propuesta para el prototipo.

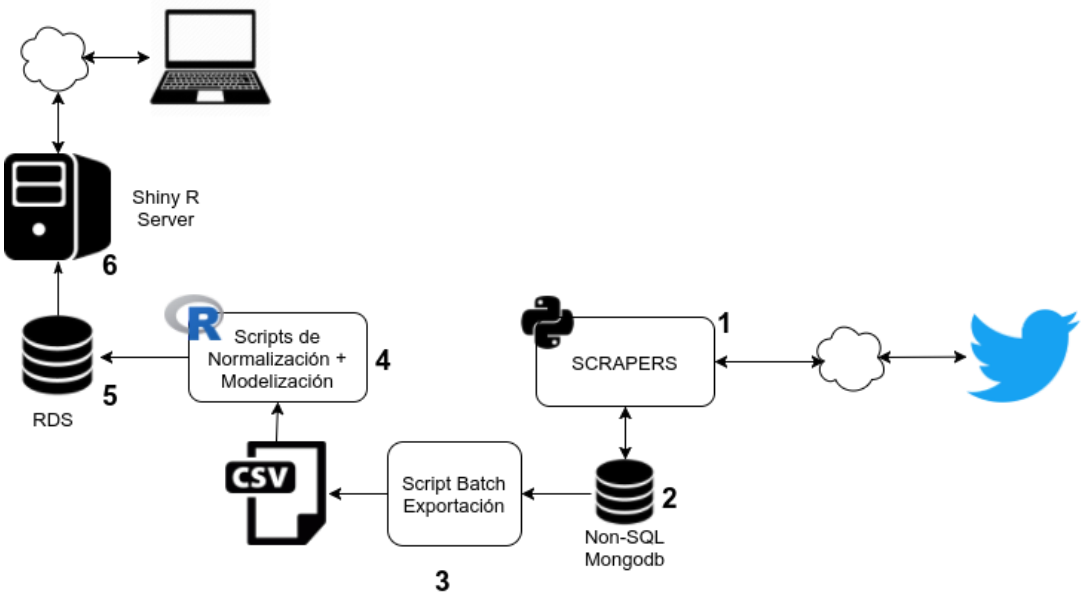

Figura 8 - Diagrama de arquitectura para el prototipo

Se explicará punto a punto de acuerdo a la numeración del diagrama.

1. Scrapers definidos. Mediante internet (representado con el símbolo de la nube) se integra con twitter para recuperar los documentos.
2. El mismo scraper almacena los documentos en una base de datos no relacional, local, MongoDB.
3. En horas definidas de la noche, se ejecutan procesos batch que exportan los datos del día más reciente capturado y lo guardan a csv. Esto se realiza porque para los scripts de 4, que se encuentran codificados en R, es trivial el manejo de csv e integrarse con una base no relacional requería del uso de conectores y librerías particulares.
4. Scripts de R que se ejecutan luego de la exportación. Estos scripts normalizan los datos, separando por palabras los documentos y se guarda en formato RDS.
5. El formato de RDS[7] es un formato que lo provee R, el cual utiliza compresión gzip de forma serializada, siendo una mejor alternativa al almacenamiento en csv en términos de espacio de disco. Además, este formato, al ser nativo del servidor Shiny (de RStudio), leer los datos procesados es más performante. Utilizamos este formato, porque la parte de procesamiento y consumo de los datos es en R, de otra manera se deberá utilizar otra solución.
6. Servidor shiny para la publicación de frontends web basados en R[8]. Este servidor web es ideal para el prototipado y evaluar usabilidad de aplicaciones web orientadas al uso intensivo de datos. El problema con este sistema es que no escala por su intensivo consumo de recursos y además tiene un no muy buen manejo de la concurrencia.

[7] https://www.r-bloggers.com/remember-to-use-the-rds-format/

[8] https://www.rstudio.com/products/shiny/shiny-server/

Es importante resaltar, que luego de finalizado el prototipo, con la metodología propuesta y utilizando la compresión provista por los RDS, 1 día completo de capturas de los 10 países, ocupa un total de 160MB aproximadamente.

Así mismo, la base de datos no relacional que contiene los tuits en bruto, al momento, luego de poco más de 6 meses de ejecución ocupa 12,4GB.

En cuanto a la arquitectura de la infraestructura, ya se comentó que se dispone únicamente de un servidor básico con conectividad a internet de un hogar, disponiendo de una única IP dinámica.

Propuesta de Arquitectura/Infraestructura

En esta sección, se aborda una definición de arquitectura de software e infraestructura que permita escalar el sistema. La idea de esta arquitectura es encontrar un balance entre la performance y el costo de mantener el sistema.

A continuación presentamos los aspectos críticos a tener en cuenta para la realización de una nueva arquitectura.

Partiendo de nuestra arquitectura actual, se estima que se el volumen de datos diarios a alojar en nuestra base de datos RDS es de 160 MB (aproximadamente 5GB mensuales), cabe destacar que esta base de datos normalizada alojará la información ya procesada, contrario a la base de datos No-SQL con información en bruto con un estimado de 2GB mensuales, es por eso que una de las variables más importantes a tener en cuenta para realizar una nueva arquitectura es el volumen de datos a alojar.

A su vez se propone mejorar la performance de los scrapers, actualmente se cuenta con una sola instancia en la cual existen bloqueantes tanto tanto en la cantidad de request diarios contra la API de twitter como en el procesamiento de información no utilizable como se detalla más arriba.

Con respecto a los componentes de software, se propone la arquitectura que se muestra en la figura 9.

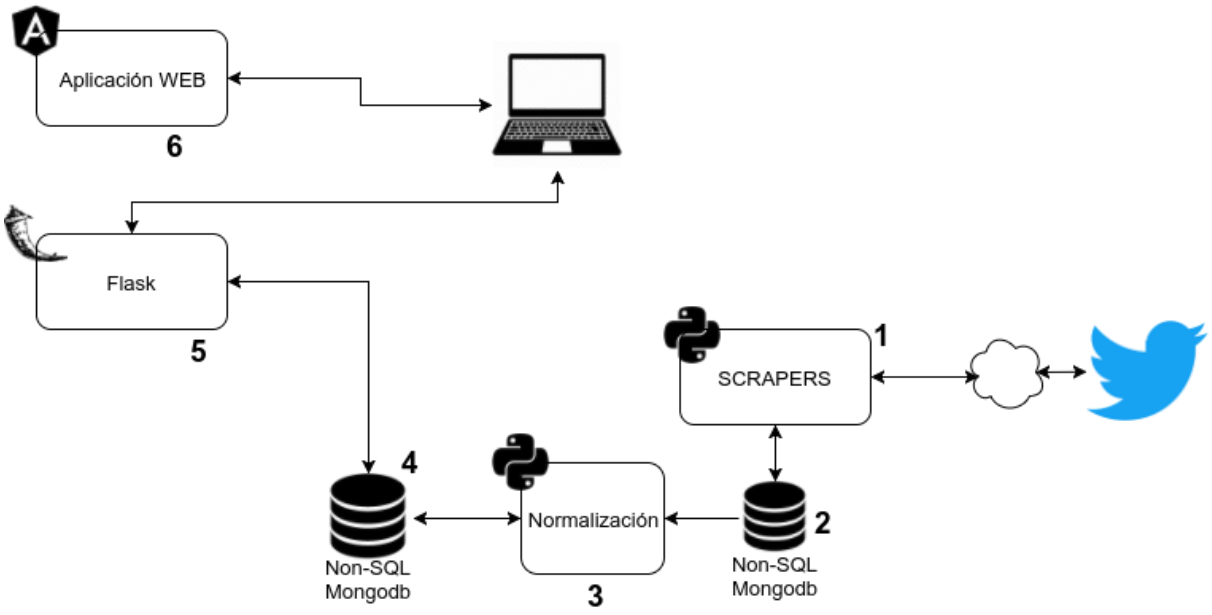

Figura 9 - Arquitectura propuesta

1. Los scrapers seguirán siendo implementados en Python, con el agregado de que contaremos con más de una instancia (o thread) del mismo con el fin de aumentar la eficiencia y solucionar tanto el problema de la captura de información no útil como el del bloqueo por parte de la API de twitter en cuanto a la cantidad de request por dia.
2. La base de datos donde alojaremos la información en bruto seguirá siendo MongoDB. Para la implementación eficiente de este componente se necesita de suficiente memoria RAM para el manejo de datos realizado ya que MongoDB mantiene índices y metadatos en memoria para rápidas respuestas.
3. Se considera como procedimiento común en el desarrollo de software implementar los procesos de normalización en Python. En este componente, es un aspecto clave contar con suficiente capacidad de procesamiento de datos (CPU).
4. La base de datos con la información normalizada será MongoDB. Al igual que en el componente con la información en bruto, es necesario contar con suficiente memoria RAM para el manejo de datos.

Es importante remarcar que cada uno de los elementos mencionados anteriormente serán implementados en máquinas virtuales independientes.

5. Se utilizará una aplicación implementada en el microframework de Python Flask donde se publicaran los servicios web que se conectará con la información ya procesada. Se utiliza Flask, porque el mismo está apuntado a la creación de

interfaces web con la flexibilidad de Python, además de que las aplicaciones en Flask son de fácil deploy en contenedores de Amazon Web Services y otros proveedores de Cloud.

6. Para presentar la información contaremos con una aplicación web implementada mediante el framework Angular 2+. Para esto se deberán analizar en detalle librerías como NVD3, especializadas para realizar las visualizaciones de la información.

Luego de la arquitectura de software planteada, se propone definir los elementos de hardware a ser utilizados teniendo en cuenta los puntos claves de cada uno de los componentes de software.

Consideramos que existen al menos dos enfoques que contemplen los requerimientos mencionados más arriba, a continuación realizaremos una explicación de ambos, detallando cómo será estructurado cada uno de ellos .

Los componentes que entendemos transversales a ambas arquitecturas son los 1, 3, 5 y 6 de la figura 9.

Para el primer componente necesitamos poder contar con diferentes instancias o threads del scraper, como vimos esto soluciona tanto el problema de recuperación de datos no utilizables como el de la restricción de acceso por parte de la API de Twitter, es por esto que optamos por un servidor local, donde se ejecutarán los diferentes threads de la aplicación, pudiendo así obtener los datos en tiempo real para cada uno de los países. Así mismo, para el componente 3, es necesario contar con manejo total de librerías de Python, mediante el uso del mismo servidor local existe ahorro en lo que refiere a costos de mantenimiento, en contraposición a usar servidores remotos bajo demanda como puede ser EC2 de AWS, dado que las actualizaciones realizadas sobre las librerías utilizadas, deberán ser replicadas en el otro servidor.

Por otro lado para los componentes 5 y 6 de la figura 9, utilizaremos contenedores de AWS, en uno se pondrá en producción una aplicación Flask y en otro una aplicación Web en el framework AngularJs.

El primer enfoque consiste en alojar los componentes 2 y 4 de la figura 9, en un servidor local (seguimos manteniendo máquinas virtuales independientes para cada uno). Este enfoque implica horas hombre dedicadas al mantenimiento de servidor, a su vez deberá contar con los aspectos claves mencionados más arriba, donde se deberá de contar con suficiente CPU para la normalización y memoria RAM tanto para la recuperación como el procesamiento de la información.

El segundo enfoque consiste alojar los componentes 2 y 4 de la figura 9 en servidores amazon S3. Los costos de los mismos fueron evaluados llegando a la conclusión de que la relación precio/volumen de datos /mantenimiento, puede llegar a ser un factor determinante en comparación con el primer enfoque. La investigación realizada para el precio del uso de los servidores S3, arroja que los primeros 50 TB/mes, equivale a 0.0405 dólares. Dada las

dimensiones de los datos que tenemos, generando en torno de 7GB mensuales, y cada 1000 requests 0,007 dólares. Estos precios son manejables.

Estos datos indican que cualquiera de los dos enfoques es viable y la decisión deberá de ser tomada en función de las horas hombres que se le quiera dedicar al manejo de la infraestructura. Se evitarían los costos del segundo enfoque, si ya se dispone de recursos capacitados en el manejo de infraestructura que esté calificado para el manejo de virtualización y manejo de RAID. En caso de no disponer de recursos, el primer enfoque es más adecuado por la facilidad del uso de los servicios Cloud y la construcción del servidor necesario es mucho más simple ya que son procesos python simples siendo ejecutados en un sistema operativo linux genérico, sin necesidad de virtualización intensiva ni de almacenaje.

Trabajo a futuro

El sistema construído intenta validar una estrategia de captura que consiga suficientes datos, no pretende implementar una arquitectura final. La arquitectura final expuesta en este trabajo deberá ser implementada y probada, así también modificada de acuerdo a los resultados de esas pruebas.

Por otro lado, deberán ser realizadas modificaciones en el código fuente de los scrapers. Al momento, para obtener el mismo instante de capturas para cada país de américa latina, se desperdician una cantidad no menor de datos, así mismo, esto aumenta de manera no menor la demora en las capturas. La arquitectura presentada pretende resolver esto mediante el uso de varias IP en simultáneo, pero de todas maneras deberán ser realizadas modificaciones sustanciales en los scripts para soportar multithreading.

Así también, deberemos desacoplar el sistema del uso de R, se deberá realizar la normalización y modelado en python, homogeneizando el sistema como pieza de software, y aumentando su performance.

La construcción de un frontend como aplicación web será una tarea que insumirá su tiempo, más dada las características del proyecto, donde se pretende mostrar información accionable apuntando a la comunicación de la misma. Para esto se deberán analizar diferentes librerías javascript y eventualmente modificarlas, para brindar la mejor experiencia de comunicación de la información.

Un sistema de la índole, autónomo, con fines comerciales, requerirá de constante monitoreo y control, el trabajo no concluirá con la construcción del sistema. Así mismo, eventualmente las APIs podrán cambiar y con esto se deberán de repensar componentes del sistema.

También sería interesante tender a la implementación de un framework de captura propio, que no requiera el uso del framework tweepy.

Conclusion

La API de Twitter ofrece información explotable, que mediante técnicas de recuperación de información, procesamiento de datos, PLN y machine learning, pueden dar lugar a generar investigaciones y productos innovadores. La integración con la misma para obtener una gran cantidad de datos útiles, sin caer en sobre costos, no es trivial. En este trabajo se analiza una de un sin fin de posibles estrategias de captura.

Nuestra estrategia está fuertemente atada a los requerimientos iniciales y ha demostrado (en los últimos meses) ser adecuada en términos de infraestructura y en términos de generar los datos necesarios para la generación del modelo hedónico que se encuentra por sobre el sistema.

Al momento el sistema se encuentra ejecutando con interrupciones mínimas, en general dadas por la precaria infraestructura, los últimos 7 meses. Seguirá expuesto el sistema, capturando datos, hasta que se degrade la arquitectura y el sistema deje de funcionar.

El uso del framework tweepy fue útil para facilitar los pedidos a la api, en esto radicó su utilidad. Para la iteración de resultados no nos resultó útil, ya que hace un no muy buen manejo de la RAM. En términos de flexibilidad tal vez pueda ser una limitante en desarrollos futuros.

Una vez más R ha demostrado ser una muy buena herramienta para el manejo de datos en términos exploratorios y de prototipación. Por las características del funcionamiento de R, no permite la construcción de sistemas que escalen para ser comercializados. Pero para sistemas con fines de investigación y academia, es ideal. Permite construir frontends orientados al manejo de datos de una forma muy simple, para así obtener validaciones inmediatas y saber sin mayor demora si las visualizaciones generadas a partir de los análisis realmente aportan información útil.