Universidad del Pais Vasco

Facultad de Ciencia y Tecnología

Proyecto de desarrollo de software: WebScraping a MRW

Eneko Ruiz (Ingeniería Electrónica)

Tecnicas Actuales de Programación (TAP)

Objetivos generales del trabajo

El objetivo de este trabajo es desarrollar una utilidad que realice web scraping en la sección "Historico envíos nacional" de la página web de MRW (el nombre puede llevar a equívocos, ya que también contiene envíos a Portugal y Andorra). Después, dicha información será almacenada en una tabla de MySQL. Posteriormente, a través de una API que nos devuelva las coordenadas de cada ciudad, se calculara la relación entre el tiempo de envió y la distancia para cada envió, con el objetivo de realizar una aplicación que, introduciendo el origen y el destino, sea capaz de predecir cuanto tardará el envió en ser entregado. **Palabras clave: Java, Python, JSOUP, MySQL, WebScraping, RestFUL API, JSON**

Índice

1 Introducción **2**

2 Scraping **2**

2.1 Utilidad de Scraping (Scraper.java) 2

2.2 Utilidad para guardar en la base de datos la información del envió. (DatabaseTranser.java) . 2

2.3 Gestión del scraper (PipelineForScraping.java) 3

3 Obtener distancia y tiempo de envió **5**

3.1 Obtención de las coordenadas (Geoencoder.java) 5

3.2 Calculo de distancias y escritura en la BBDD (FromCityToDistance.java) 5

4 Análisis de los datos **6**

5 Conclusiones **8**

1 Introducción

El objetivo del presente documento es dar cuenta y explicar el software realizado para el objetivo arriba citado. Éste se subdividirá en 3 apartados: primero, explicaremos como y basados en que asunciones hemos programado el *Scraper*, después introduciremos las herramientas que nos han permitido obtener las coordenadas del origen y destino de cada envió, para posteriormente analizar la relación entre distancia entre origen y destino y tiempo transcurrido desde que el paquete es entregado al repartidor en origen hasta que éste lo entrega en destino.

A partir de esa relación, se intentará ajustar un polinomio de la forma

$$f(x) = \sum_{k=0}^{n} a_k x^k$$

que mejor represente nuestros datos. Si esto no se consigue, se intentara con los más estables polinomios de *Legendre*. Si se obtiene un resultado satisfactorio, dicho polinomio será usado en un aplicación móvil (operable en Android), que permitiría predecir la duración de un envió.

2 Scraping

Dentro del proyecto, los ficheros dedicados unicamente a hacer *scraping* a la página web de MRW son: *DatabaseTranser.java*, *Scraper.java* y *PipelineForScraping.java*. También encontramos otra clase, *InformacionEnvio.java*, en cuyos objetos almacenaremos la información del envió.

Esta clase tendrá como atributos el numero de albarán, el lugar de recogida y envió (String) y las fechas de recogida y envió (Date).

2.1 Utilidad de Scraping (Scraper.java)

En *Scraper.java*, definimos la utilidad de *scraping*. Esta clase tendrá como atributos la URL que queremos *scrapear*, un numero de albarán, un servidor proxy y una cabecera (o *header*) *HTTP user-agent*.

En la clase, usaremos el método *scrapingTool*, que nos devolverá un objeto *InformacionEnvio*, tras el proceso de *scraping*. Para descargar el código HTML de la página usaremos el paquete JSOUP. En este paquete, además de la URL, podemos especificar el servidor proxy y el *HTTP header user-agent* que queremos utilizar.

Posteriormente, a través de expresiones regulares (REGEX) buscaremos en el código HTML descargado palabras clave como "entregado", "recogido" o "recoger", parsearemos la información y la guardaremos en un objeto *InformacionEnvio*.

Este método devolverá un objeto de la clase *InformacionEnvio*, exista o no exista pedido con ese albarán. Por lo tanto, antes de insertar nada en la base de datos, debemos comprobar que todos los campos están rellenados, ésto es, que estamos ante un pedido que existe y que se ha entregado (no que este en camino).

2.2 Utilidad para guardar en la base de datos la información del envió. (DatabaseTranser.java)

El objetivo principal de esta clase es recibir un objeto *InformacionEnvio* (obtenido previamente en la utilidad de *scraping*) y, una vez comprobado que todos sus campos están rellenos, insertarlo en la base de datos, en la

tabla INFO_GENERAL_ENVIO. Para ello, previamente deberemos conectarnos a la base de datos, a través del paquete java.sql.

En caso de recibir un objeto que no tenga todos los atributos, no se realizará una inserción. Se ha decidido no lanzar ninguna excepción porque esta utilidad se ejecutara recursivamente (se explicara en la siguiente subsección) y lanzar una excepción detendría dicha ejecución.

2.3 Gestión del scraper (PipelineForScraping.java)

En esta clase, definida como un singleton, se gestionara todo el proceso de scraping. La clase extenderá *TimerTask*, ya que queremos que sea ejecutada continuamente en intervalos de tiempo prefijados.

Los albaranes están formados por doce caracteres numéricos. Si todo el proceso de *scraping* (llamada y inserción en base) tardase un segundo, necesitaríamos 10^{12} segundos, o más de 30 mil años, para *scrapear* todos los pedidos. Por lo tanto, descartamos la posibilidad de recorrer todos los albaranes, ya que nosotros solo tendremos una semana (una vez afinada la herramienta) para recoger información. Realizar llamadas con una frecuencia inferior a un segundo también se descarta, porque podría sobrecargar el servidor. Además, hay que añadir que se han encontrado albaranes alfanuméricos (ésto es, con caracteres alfabéticos), por lo que el tiempo necesario seria considerablemente mayor a los mentados 30 mil años.

Por otra parte, tampoco podremos generar números de longitud 12 aleatoriamente, ya que la mayoría de albaranes, como se ha podido comprobar, están vacíos. A continuación, se explica la estrategia que vamos a seguir.

A partir de un albarán que ya conocemos que contiene un pedido (la web de MRW nos permite buscar por código de envió), variaremos los números del albarán, hasta obtener un numero de albarán no nulo. Por ejemplo, para el albarán no nulo 123456789000, iremos cambiando, empezando por la izquierda, los números del albarán hasta obtener uno no nulo. Si, por ejemplo, 123456779000, es el primer albarán no nulo, tomaremos todos los a la izquierda de éste y los guardaremos en una *seed* (en nuestro ejemplo, la *seed* será 1234567).

A partir de la *seed*, generaremos números aleatorios hasta llegar a 12 cifras. De nuevo en nuestro ejemplo, la *seed* tendrá longitud 7, por lo que tendremos que generar un numero aleatorio de 5 cifras, ésto es, entre 00000 y 999999.

Este método tiene dos principales desventajas. La primera desventaja es que, a pesar de ser más eficiente que generar un numero aleatorio de 12^{12} cifras y probar a ver si existe un pedido con ese albarán, aún son muchos los albaranes vacíos que obtendremos. Por lo tanto, la cantidad de información que consigamos recoger dependerá en gran medida de como de "buena" sea nuestra *seed*.

El segundo problema es que le restará calidad a la información que consigamos obtener. Los números de albaranes no están formados aleatoriamente (el autor ha intentado deducir el formato seguido para generarlos, sin éxito), por lo que los pedidos obtenidos a partir de una misma *seed* suelen compartir características, como el lugar de envió o recogida.

Esto puede solucionarse parcialmente utilizando varias *seeds* y descartando las *seeds* que generen datos más desviados. Por ejemplo, es de esperar que un envió que sale desde Madrid (por la configuración de las carreteras españolas) llegue antes a cualquier punto de nuestra geografía que un envió que sale de cualquier otro lugar, a pesar de que la distancia a cubrir sea inferior. Por lo tanto, descartaremos *seeds* con lugar de envió Madrid o su área metropolitana.

Una vez solucionado este problema, debemos abordar uno de los mayores problemas del *scraping*: evitar ser detectados y bloqueados. Para ello, debemos intentar que el comportamiento de nuestro *bot* de *scraping* sea lo más parecido posible al de un humano (o al de varios humanos).

Para ello, empezaremos añadiendo cabeceras HTTP a las llamadas que realizará nuestro *bot*. Una llamada a través de un cliente web (navegador) contiene en torno a de 6-7 *headers*. Sin embargo, nosotros solo usaremos una: el *user-agent* o agente de usuario. Según la fundación Mozilla, "La solicitud de cabecera del Agente de Usuario contiene una cadena característica que permite identificar el protocolo de red que ayuda a descubrir el tipo de aplicación, sistema operativo, proveedor del software o la versión del software de la petición del agente de usuario." [1].

Aparte de dichas cabeceras, usaremos un servidor proxy. En el caso de que el sistema nos detecte (por ejemplo, por estar recibiendo muchas peticiones desde una misma IP) y bloquee nuestra IP, realmente estará bloqueando la del proxy, por lo que cambiar el proxy serviría para burlar este bloqueo. Un servidor proxy es una especie de intermediario entre nosotros (cliente) y el servidor de destino (la página web), filtrando las llamadas. De esta forma, la página web no conocerá nuestra IP, si no la del proxy.

Una combinación de proxy y *user-agent* aleatorios seria lo ideal para que nuestro *scraper* pase desapercibido y no sea bloqueado. Obtener una cantidad considerable de cabeceras *user-agent* es relativamente sencillo. Sin embargo, conseguir una cantidad considerable de proxys que funcionen, sin recurrir a aplicaciones de pago, es mucho más complicado. Una posibilidad seria utilizar APIs que nos permitan generar nuestros propios proxys[2]. Sin embargo, nosotros seguiremos un modo más "tradicional": buscar proxys en listas gratuitas online, e ir probandolos hasta topar con uno que funcione.

Como la mayoría de proxy obtenidos en sitios gratuitos dejan de funcionar a las pocas horas, a pesar de que existe la utilidad de seleccionar los proxys de un fichero aleatoriamente, nosotros "hardcodeademos" uno que ya sabemos que funciona (y que probablemente tendremos que cambiar si usamos la aplicación tras un par de días). Aún así, la explicación que sigue a continuación asume que disponemos de un fichero con un numero considerable de proxies que sabemos que, al menos la gran mayoría, funcionará correctamente.

Por último, como se ha comentado antes, la clase *PipelineForScrapping* extiende a las clase *TimerTask*. Con ello, invocaremos las utilidades previas con una frecuencia elegida de forma aleatoria (entre 1-5 segundos). Se ha elegido una frecuencia de 1-5 segundos para no sobrecargar el servidor y para mimetizar un comportamiento humano (ya que éste no seria capaz de modificar la URL y actualizar la página cada 10 milisegundos).

Una vez explicado esto, vamos a analizar el funcionamiento de esta clase. *PipelineForScrapping* se instanciará a través de un singleton, que invocara un constructor (privado) que tendrá como atributos la dirección a las listas de *user-agents* y proxys, la URL de MRW, así como la *seed* y un booleano.

La clase, posteriormente, invocará a *Scrapper.java* para obtener un objeto *InformacionEnvio* y a *DatabaseTransfer.java*, para guardarlo en la en la base de datos (BBDD). A su vez, le pasará a *Scrapper.java* un *user-agent* y un proxy seleccionados aleatoriamente del un fichero (atributo de la clase). Antes de enviar la proxy, se comprobara que funcione, y en caso contrario, se eliminara del fichero y se volverá a probar con otra. Por último, tendremos un método que generará números enteros aleatorios para completar la *seed*.

Los albaranes se guardarán en un Set (se ha elegido Set en vez de List porque su complejidad temporal es menor[3]), para evitar insertar dos veces el mismo albarán (en nuestro caso, para evitar provocar una excepción, ya que el numero de albarán será una clave primaria). Este set puede cargarse vació o con los albaranes ya presentes en la tabla INFO_GENERAL_ENVIO (la posibilidad se elegirá con un booleano, que será un atributo del constructor).

El hecho de haber dejado en manos del usuario el usar, en cada ejecución del programa, un set vació o cargar uno con los albaranes presentes en la base de datos se debe a que no siempre nos interesa utilizar un set con, por ejemplo, 10^4 entradas ya que esto cargaría considerablemente nuestra memoria RAM y haría la ejecución del programa mas lento.

[1] https://developer.mozilla.org/es/docs/Web/HTTP/Headers/User-Agent
[2] http://scrapoxy.io/
[3] https://www.baeldung.com/java-collections-complexity

Sin embargo, puede que accidentalmente, en medio de la ejecución del programa, hayamos cancelado la ejecución o nos hayamos quedado sin Internet, y queramos, una vez solucionado el problema, seguir *scrapeando* la URL con la misma *seed*. En este caso, si que seria interesante cargar los albaranes ya almacenados para evitar la arriba citada *SQLException*.

3 Obtener distancia y tiempo de envió

A partir de las utilidades de *scraping*, el paquete incluye dos clases (*FromCityToDistance.java* y *Geoencoder.java*) que permitirán obtener la distancia (tanto temporal como física) del envió.

3.1 Obtención de las coordenadas (Geoencoder.java)

Con esta clase, que se instanciará a partir de un nombre de usuario de GeoNames y el nombre de una ciudad, obtendremos las coordenadas en las que se encuentra esta ultima utilizando dos APIs (GeoNames y de Google).

El motivo por el que usamos dos APIs es el siguiente: ninguna de las dos nos permite especificar más de dos países (por ejemplo, España y Portugal) en los que realizar la búsqueda. Esto no es problema en ciudades como Oporto o Madrid, que están indudablemente en Portugal y España. Sin embargo, en ciudades más pequeñas y no tan conocidas, la API de Google nos devolverá el resultado más parecido en USA. Por ejemplo, el pueblo gallego "Barro" comparte nombre con un restaurante en el condado de Maricopa, Arizona.

Por otra parte, como MRW no nos dice si el envió procede de España o Portugal, es peligroso añadir, en la API de Google, el país en el que buscar. De nuevo con la gallega Barro, en el caso de indicar España, nos devolverá el resultado correcto. Sin embargo, si indicamos Portugal, existe una ciudad portuguesa llamada Barrô, que será la que nos devuelva.

Una forma no demasiado elegante de atajar este problema es la siguiente. Se ha comprobado que si buscamos una ciudad española con nombre parecido en Portugal (por ejemplo, Barro) o viceversa, la API de Google nos devolverá un *JSONArray* con posibles soluciones (en el caso de que se busque la ciudad en el país al que pertenece, solo devolverá un resultado). Por lo tanto, se ha incluido un método que, en el caso de usar la API de Google, compruebe la longitud de la respuesta, y si es mayor que la unidad, cambie la región en la que se busca (de ES a PT, o viceversa).

Sin embargo, esta solución solo es un parche, por lo que, de forma general usaremos la API de GeoNames, que, a pesar de ser más lenta, no es capaz de buscar restaurantes y demás servicios que añadían imprecisión a la API de Google.

Aún así, esta API también tiene problemas. Por ejemplo, ubica Sabadell en Filadelfia (Pensilvania) o es incapaz de encontrar pueblos pequeños. Por lo tanto, como norma general, utilizaremos la API de GeoNames, pero si obtenemos unas coordenadas que no pertenecen ni a España, Portugal, Andorra o Francia u obtenemos una respuesta vacía, cambiaremos a la API de Google, fijando España como país de búsqueda por defecto, y cambiando a Portugal si obtenemos una respuesta con más de un resultado.

3.2 Calculo de distancias y escritura en la BBDD (FromCityToDistance.java)

Esta clase sera la encarga de recorrer la base de datos rellanada por el *scraper*, y tras calcular las distancias física (con la utilidad previa) y temporal, guardar estas, así como el albarán, en una tabla en la BBDD (tabla DISTANCIA_TIEMPO_ENVIO).

La distancia temporal se calculara en horas, a partir de la diferencia entre la fecha de envió y de recogida.

Por su parte, para obtener la distancia física, calcularemos la distancia en linea recta (en kilómetros) desde la ciudad de envió a la de recogida, a partir de las coordenadas que nos devolverá *Geoencoder.java*.

Por la previamente comentada configuración de las carreteras españolas, es muy posible que un pedido tarde menos en recorrer los más de 600 kilómetros que separan Madrid de Barcelona que 200 kilómetros entre dos pueblos de las provincias de Zamora y Salamanca. Sin embargo, en general, es una buena estimación, por lo que nos basaremos en ella.

Por último, al igual que en el *scraper*, usaremos un Set para evitar la excepción que se lanzara si intentamos escribir en la tabla un albarán ya existente (aquí también es el numero de albarán una clave primaria). Al igual que en la utilidad previa, este set también puede cargarse vació o a partir de los albaranes de la BBDD.

Por último, comentar que tanto la utilidad de *scraping* como esta, contarán con su propio método Main (independientes el uno del otro), para que el usuario tenga la posibilidad de ejecutarlas independientemente. Este método Main será una clase privada definido fuera del cuerpo de la clase principal.

4 Análisis de los datos

Una vez guardadas en una base de datos la distancia temporal y física de cada envió, representaremos estados dos variables con la ayuda del paquete *Matplotlib* de *Python*. El resultado (para 1250 envíos elegidos aleatoriamente) es el siguiente:

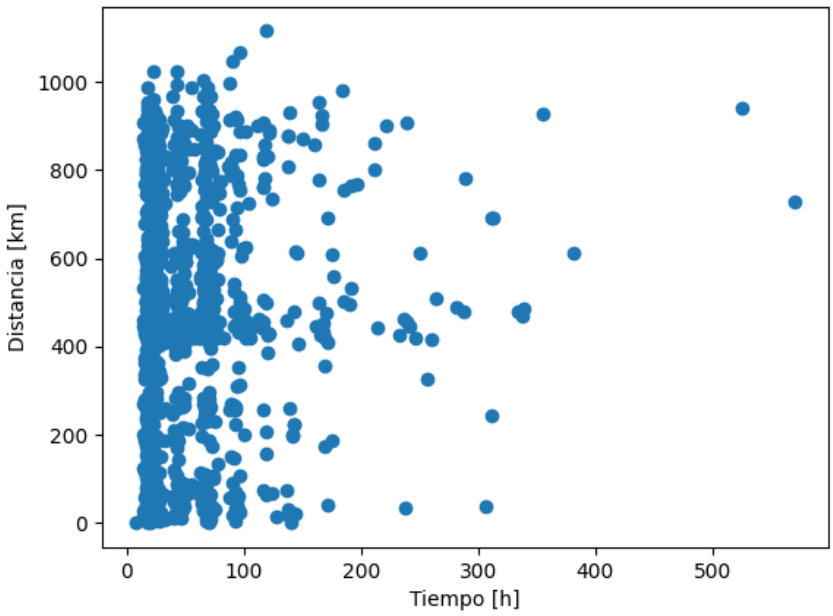

Figure 1: Distancia temporal frente a física.

Si realizamos un análisis algo más pormenorizado, nos encontramos por ejemplo que las distancias temporales entre la gaditanas Arcos de la frontera y Barcelona capital para distintos envíos es:

gen_albaran	gen_lugar_envio	gen_lugar_entrega	env_distancia	env_tiempo
010092736273	ARCOS DE LA FRONTERA	BARCELONA	859.8749208717627	15
010092751683	ARCOS DE LA FRONTERA	BARCELONA	859.8749208717627	40
010092756461	ARCOS DE LA FRONTERA	BARCELONA	859.8749208717627	44
010092767370	ARCOS DE LA FRONTERA	BARCELONA	859.8749208717627	191
010092775953	ARCOS DE LA FRONTERA	BARCELONA	859.8749208717627	49

Figure 2: Tiempos de envió para artículos con origen Arcos de la Frontera y destino Barcelona.

Por lo tanto, hay envíos que tardan menos de un día (15h) en cubrir los casi 860 que separan ambas ciudades y envíos que tardan casi 8 días (191 horas).

Este resultado, a primera vista incongruente, se explica, en opinión del autor, de una forma sencilla. Descartando los retrasos típicos del transporte y la logística, como un almacén ineficiente o retrasos debido al trafico, que explicarían una desviación de como mucho 10-20 horas, es de esperar que un paquete con envió urgente o en un día llegue considerablemente antes que un paquete con envió estándar, ya que se paga precisamente por ello.

Es por ello que la información "cruda" de la página de MRW no es de muy poca utilidad para predecir el tiempo que tardara un paquete en llegar, y por lo tanto construir una aplicación que nos proporcione esa información carecería de sentido.

Sin embargo, estos datos si nos proporcionan información, como que la mayoría de paquetes tardan entre 25 y 75 horas en llegar al destino, independientemente de la distancia, tal y como podemos ver en el siguiente histograma:

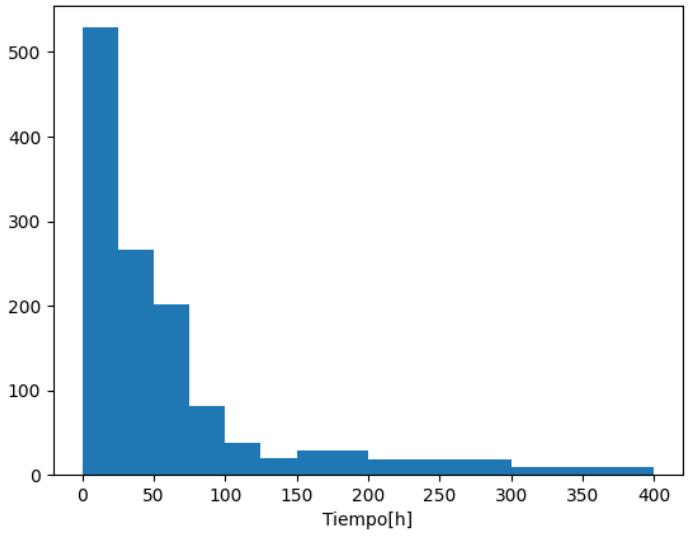

Figure 3: Histograma de tiempos de envió para 1250 envíos.

Es posible ver un descenso exponencial del numero de envíos a medida que aumenta el tiempo.

5 Conclusiones

En el presente trabajo hemos conseguido hacer *scraping* en la página web de MRW, rellenar una base de datos en MySQL, y, con la ayuda de las APIs de Google y GeoNames, calcular la distancia en linea recta entre ambos puntos (en kilómetros) y las horas que tarda un envió en llegar del origen al destino.

Por ultimo, se han analizado gráficamente la distancia entre ambos puntos y el tiempo transcurrido, no observándose ninguna relación aparente entre ambas variables. Sin embargo, si se ha podido comprobar que la gran mayoría de envíos tardan entre 20 y 75 horas en llegar a su destino.