NERIT: Extracción de Entidades con Nombre en Tweets

Diseño de Sistemas de Software

Marcelo Javier Rodríguez
Leg: 245-221

mailto:mrodriguez@alumnos.exa.unicen.edu.ar

Profesor: Doctor Ing. Álvaro Soria.

Resumen

En este documento se exponen las decisiones tomadas para la construcción de un pipeline mediante el cual procesar tweets. El objetivo fundamental del informe se dirige a la extracción de frases sustantivas, en particular, direcciones postales. El resto del documento enfatiza la solución escogida en cada etapa, limitaciones encontradas y propuestas de mejora.

1.Motivación.

Twitter es, hoy en día, el servicio de microblogging más utilizado en el mundo. Actualmente los usuarios escriben tweets en los que comparten variados aspectos de su vida personal,opiniones de distintos tópicos,etc. Esta fuente de información representa una oportunidad interesante dada su potencial utilidad para informar sobre una problemática en particular; p.ej inferir preferencias/perfiles de comunidades de usuarios,reacción ante un proyecto social, organización y difusión de eventos en tiempo real,etc. Debido al inmenso volumen de información que provee Twitter, la aplicación de técnicas de Extracción de Información para atacar una cuestión singular resulta una alternativa prometedora para arribar a una solución específica.

En este contexto, la intención de este proyecto es crear **NERIT**, una aplicación que se basa en aplicar técnicas de NLP[1] sobre los tweets mayoritariamente referentes al estado de tránsito, y extraer las frases sustantivas referentes a direcciones postales o edificios conocidos/lugares de interés.

En este documento se especifíca la construcción de NERIT enfatizando la dificultad que impone tratar con contenido web y texto escrito informalmente.

2. NERIT

La aplicación NERIT se organiza en una arquitectura tipo pipeline. En un pipeline, cada etapa recibe como datos de entrada la salida de la etapa precedente, y envía su salida como entrada a la etapa que le precede ; a su vez, estas fases o etapas deben poder ser utilizadas individualmente, y particularmente en este caso, componer etapas de mayor granularidad a fin de obtener/combinar distintos resultados durante el procesamiento del texto.

El pipeline propuesto utiliza diccionarios, listas de expresiones regulares y las mencionadas etapas. A continuación, se explica concisamente las etapas en su orden de necesidad y desarrollo en el pipeline.

2.1. Etapa de captación de datos

Para iniciar el procesamiento es imprescindible tener la habilidad de tomar datos desde una colección. Las distintas fuentes de datos se integran al pipeline extendiendo clases de modo que cualquier fuente (base de datos, archivo de texto,web u otros) pueda ser procesada siempre que esta adhiera a la interface requerida del pipeline.

Los tweets son entregados desde la API de Twitter en formato **JSON**[2], y dentro del pipeline las etapas determinan explícitamente sobre cúales campos realizarán su procesamiento; el tipo de datos final guarda la evolución del procesamiento entre etapas: Particularmente se utiliza una estructura de datos tipo Hash, que por ejemplo almacenará el texto original asociado a la clave *'text'*; su correspondiente el correspondiente etiquetado gramatical asociado a la clave *'tagged'*, etc.

2.1. Etapa de tokenización.

El texto de los tweets no sigue una ortografía y estructura gramatical correcta o formal, sino que es escrito en forma libre. Una simple tokenización por espacios no es suficiente, y tokenizar por signos de puntuación no es viable dado el contenido de los mismos: a diferencia del vocabulario natural, aquí se incluyen elementos web (hashtags, menciones de usuarios, urls, emails entre otros), abreviaturas, etc. Esta ampliación de vocabulario vuelve la tarea de tokenización más desafiante conforme aumenten los tokens que

1 http://en.wikipedia.org/wiki/Natural_language_processing

2 http://es.wikipedia.org/wiki/JSON

deseemos detectar o transformar. Por ende, y para consevar dichos tokens correctamente es imprescindible definir *expresiones regulares* que puedan detectar dichos tokens de forma particular (y tratar el resto que no es de utilidad) a fin de que etapas posteriores mejoren la presición/recall de sus ejecuciones.

El pipeline propuesto permite definir estas expresiones regulares fuera del código fuente, y asociar operaciones que pueden dispararse cuando un token adhiere a dicha expresión. Es decir, por ejemplo: cuando se encuentra un hashtag[3], eliminar **#** y si el hashtag está escrito en forma **CamelCase**[4], separar sus palabras; sea "#LaPlata", su procesamiento quedará como "La Plata"; eliminar signos de puntuación ligados a palabras del español: "chocaron**...**", su procesamiento será "chocaron" ;etc. El último ejemplo tendrá como resultado en su clasificación gramatical un verbo, cuando al estar ligado a signos de puntuación hubiere clasificado como "no reconocido-fuera del vocabulario" o cualquier falso negativo.

2.2. Etapa de etiquetado gramatical

Al momento de realizar la clasificación gramatical o **Part Of Speech**[5] de las palabras en español se obtienen resultados satisfactorios siempre que las palabras hayan sido vistas en los datos de entrenamiento del etiquetador; aún así, la presición de esta etapa se ve afectada por las palabras cuya clase gramática es sensible al contexto. Ej: "puente *circular*" y "*circular* por "; se tiene la misma palabra, en el primer caso como adjetivo; y en el segundo, como un verbo.

Cuando se prueba el etiquetado en los textos de tweets esos inconvenientes persisten y adicionalmente, la presición se reduce dramáticamente dado que quedan muchos tokens fuera del vocabulario del modelo entrenado debido a la naturaleza del tweet.

Para mitigar las situaciones descriptas de propone:

- Palabras ambigüas: una clasificación por n-gramas. Fundamentalmente esto es encadenar modelos de datos entrenados sobre secuencias de n-tokens, y predecir la clase de la última palabra dado el contexto que le precede. También se utilizan diccionarios (o listas) de adverbios, contracciones del español,verbos,etc. Cuando un etiquetador de n-gramas no determina efectivamente una categoría para una palabra, delega esta responsabilidad al etiquetador de un contexto más corto, y así sucesivamente. Los etiquetadores por unigramas se sirven de los diccionarios definidos.

- Palabras fuera del vocabulario: En la etapa de tokenización se definieron expresiones regulares para detectar tokens que de otro modo se podrían haber tokenizado o bien contener signos de puntuación que fuerzan a un etiquetado gramatical erróneo. Todos estos tokens son definidos morfológica o sintácticamente y reconocidos con total precisión, por ende, estas expresiones y otras (adverbios: todas terminadas en -mente; verbos, terminados en e/ando) también pueden ser añadidas a la cadena de etiquetadores gramaticales para formar parte de un etiquetador por unigramas para obtener resultados aún más precisos

- Diseño de clases utilizando Decorator y Strategy Pattern: si bien no podremos llegar a una presición de 100% contra corpus de prueba muy variados, ante la necesidad, el pipeline permite añadir más etapas o enriquecer funcionalidad de las que se pretende ajustar los n-gramas fallidos (e infrecuentes dentro de un texto correctamente escrito) y corregirlos mediante una heurística adaptada a los casos

3 http://es.wikipedia.org/wiki/Hashtag
4 http://es.wikipedia.org/wiki/CamelCase
5 http://es.wikipedia.org/wiki/Etiquetado_gramatical

particulares. Por ejemplo: al encontrar un adjetivo, asegurarnos que no sea una conjugación de un verbo en participio. Una configuración de clases que brinde tal flexibilidad es necesaria para acercarnos a resultados aún más satisfactorios.

2.3. Etapa de extracción de chunks

Los chunks son secuencias de etiquetas gramaticales. La extracción de chunks es el proceso de detectar patrones de estas etiquetas; en adelante pos tags (part of speech tags). La idea subyacente de esta etapa es que podemos extraer frases significativas observando patrones específicos de tags. Entre otras técnicas de extracción de chunks, el presente informe acata dos de ellas. Como primer instancia, el reconocimiento de estas frases mediante gramáticas basadas en palabras-tags;la otra,predicción probabilística.

2.3.1.Extraer frases mediante gramáticas.

El uso de gramáticas o expresiones regulares para extraer frases requiere de la observación de los tipos de instancias de tweets que nos interesaría atrapar y, para cada una, definir la expresión que permita su captura. Por su alta presición resulta una opción muy adecuada para chunks particulares, pero la especifidad de esta técnica conlleva a un trade-off entre recall-presición ya que las frases que no son explícitamente programadas quedarán fuera de los resultados. En consecuencia, para superar este límite es necesario adicionar expresiones. Esta incorporación gradual exige más atención por parte del desarrollador; no es un proceso trivial ni complejo,pero se debe advertir el orden de prioridad entre sus definiciones y la potencial superposición entre ellas. Perder esta pista puede resultar en frases incompletas a causa de emparejamiento con reglas menos restrictivas, resultando en falsos negativos. Es decir, ante el deseo de mantener presición y mejorar el recall, podemos perder ambos.

2.3.2.Preparación de un corpus y modelo para extraer frases.

Las direcciones postales no siguen patrones estrictamente definidos (pueden ser números, sustantivos propios/comunes, incluir verbos,etc). Más bien, las reconocemos por el contexto en el que aparecen: colocaciones[6] de preposiciones, adverbios y verbos (frente a,cruzando,desde,hasta) y otras combinaciones de palabras invariables e independientes del género.

Si bien la técnica de gramáticas es adecuada, exponer sus limitaciones en el contexto de este problema particular, hace evidente la presentación de su alternativa en aras de mejores resultados.

La alternativa de detección de frases vía predicción probabilística requiere de ejemplos a partir de los cuales entrenar un clasificador. La preparación de estos ejemplos conforma un corpus: un corpus, no es más que una colección de datos (y tags u otros adicionales). Las ventajas de este método permiten reconocer tantas frases como características se pueden extraer de las instancias de datos de entrenamiento; así, cuanto más calidad y variedad de muestras tenga nuestro corpus, con más sensibilidad y precisión se comportará el clasificador.

6 http://es.wikipedia.org/wiki/Colocación

El corpus actual se formó a partir de tweets de tránstico procesados con ayuda de la técnica de gramáticas y un posterior etiquetado y corrección manual. Una instancia de entrenamiento en dicho corpus puede ser:

corte *NN* **B-EVENT**
9 *CD* **B-LOCATION**
de *IN* **I-LOCATION**
julio *NNP* **I-LOCATION**
bsastransit *NN* **O**

Los tags intermedios de cada línea son *pos tags*. Los pos tags NN y NNP representan sustantivos comunes y propios; IN, preposiciones; CD datos numéricos. Los tags en negrita determinan los chunks en formato **IOB**, donde:

B indica el comienzo de un chunk, **I** determina que el tag es interno o forma parte del mismo chunk, y **O** que no corresponde a una frase o chunk.

Por defecto, la aplicación provee un modelo Naive Bayes[7] para extraer frases entrenado sobre el corpus que contiene poco más de 600 tweets referentes al tránsito automovisístico. Durante el desarrollo de la aplicación se realizaron tests comparativos entre ambas técnicas para extraer las frases de direcciones domiciliarias. La técnica probabilística resultó tener una presición en IOB tags de hasta 83.7% cuando el modelo se entrenó con un 70% de los datos del corpus, respecto de la otra con 75.9%. Adicionalmente, la presición y recall del modelo entrenado siempre está sujeto a la calidad y variedad del corpus, y las carácterísticas variables, que deseemos extraer de los datos entrenados (si extraemos características que no aportan a la búsqueda, tendremos resultados pobres).

Esta técnica también representa una ventaja en téminos de desarrollo, pues el proceso para extraer más direcciones no cambia programáticamente, sino que sólo implica enriquecer el corpus sin requerir escribir nuevos algorítmos.

Entonces, NERIT ofrece dos métodos para extraer frases. Si se desea extraer otro tipo de frase, estos métodos pueden combinarse e intercambiarse de forma transparente: podemos extraer las direcciónes probabilísticamente y, por decir, fechas mediante gramaticas. El intercambio o instanciación en la aplicación se logra mediante los patrones de diseño Factory Method y Adapter; la combinación o composición de funcionalidad adicional se obtiene mediante Decorator Pattern y Strategy.

3. Diseño de clases de NERIT.

A continuación se expone una justificación concisa sobre las clases relevantes.

- **Source:** esta clase se implementa como un Observer. Extendiendo esta clase, se puede notificar a cualquier suscripto ante un cambio. Por ende, si escuchamos por tweets desde Twitter o los tomamos desde un archivo, o base de datos, al arrivar un texto podemos notificar dicho evento a un numero arbitrario de receptores.

- **Pipeline:** este objeto representa una secuencia de Stages que se van ejecutando secuencialmente. Extiende de la clase observer, de esta manera puede ir procesando los datos que le llegan por

7 http://es.wikipedia.org/wiki/Clasificador_bayesiano_ingenuo

- difusión de notificaciones. Adicionalmente, puede registrar subscriptores para la etapa final (determinar qué operaciones hacer al finalizar un proceso). Cualquier objeto que desee incorporarse como etapa del pipeline debe extender la clase Stage, o bien adaptar su interface.

- **Stage:** simplemente toma un dato, invoca las operaciones de un objeto y retorna resultados. Stage sigue al patrón de diseño Template Method, las implementaciones particulares siguen al Adapter Pattern. Esta configuración facilita a las clases con diferentes signaturas en sus métodos participar en el pipeline, ejecutando en secuencia e invocar sus métodos útiles de forma transparente en cada paso.

- **Tokenizers:** Se tiene un conjunto de clases Tokenizers, modeladas mediante Template Method Design Pattern. La elección se justifica en el funcionamiento básico de la tokenización: tomar secuencias de caracteres,su filtrado o modificación; en consecuencia, cada subclase de tokenizer está confinada a un marco de trabajo con diferencias particulares propias de cada subclase. Las implementaciones de cada tokenizer se corresponden con un conjunto de expresiones regulares definidas fuera del source code. La secuencia de tokenizers puede realizarse envolviendo las instancias mediante Decorator Design Pattern o bien disponiendo de una secuencia o lista de tokenizadores y conectando sus entradas y salidas.

- **Taggers:** Los taggers también se modelan mediante Decorator Design Pattern de modo que sea posible añadir responsabilidad dinámicamente. La motivación principal, como en todas las etapas viene de la observación de las frases extraídas: queremos ir haciendo mejoras sobre el texto en forma incremental. Adicionalmente, las instacias de taggers "envueltas" pueden utilizar el patrón strategy para encapsular técnicas las heurísticas a aplicar.

- **Chunkers:** el diseño aplicado esta etapa es análoga a la anterior, sólo que utiliza un nivel de abstracción más alto: consideramos secuencias de pos tags, chunks.

Finalmente, para simplificar la interacción y creación de instancias de estos subsistemas se propone utilizar Facade Design Pattern combinado con Abstract Factory (conformado de Factory Method para crear parejas de objetos que deben crearse juntas: Ej Tagger y su Adaptador), tal como se expone en las clases Nerit y TokenizerFactory, ChunkerFactoy,etc.

4.Conclusion académica.

En el diagrama se muestran las clases relevantes y sus relaciones. Se pretende que el modelo sea cerrado para modificaciones y abierto para extender su funcionalidad.

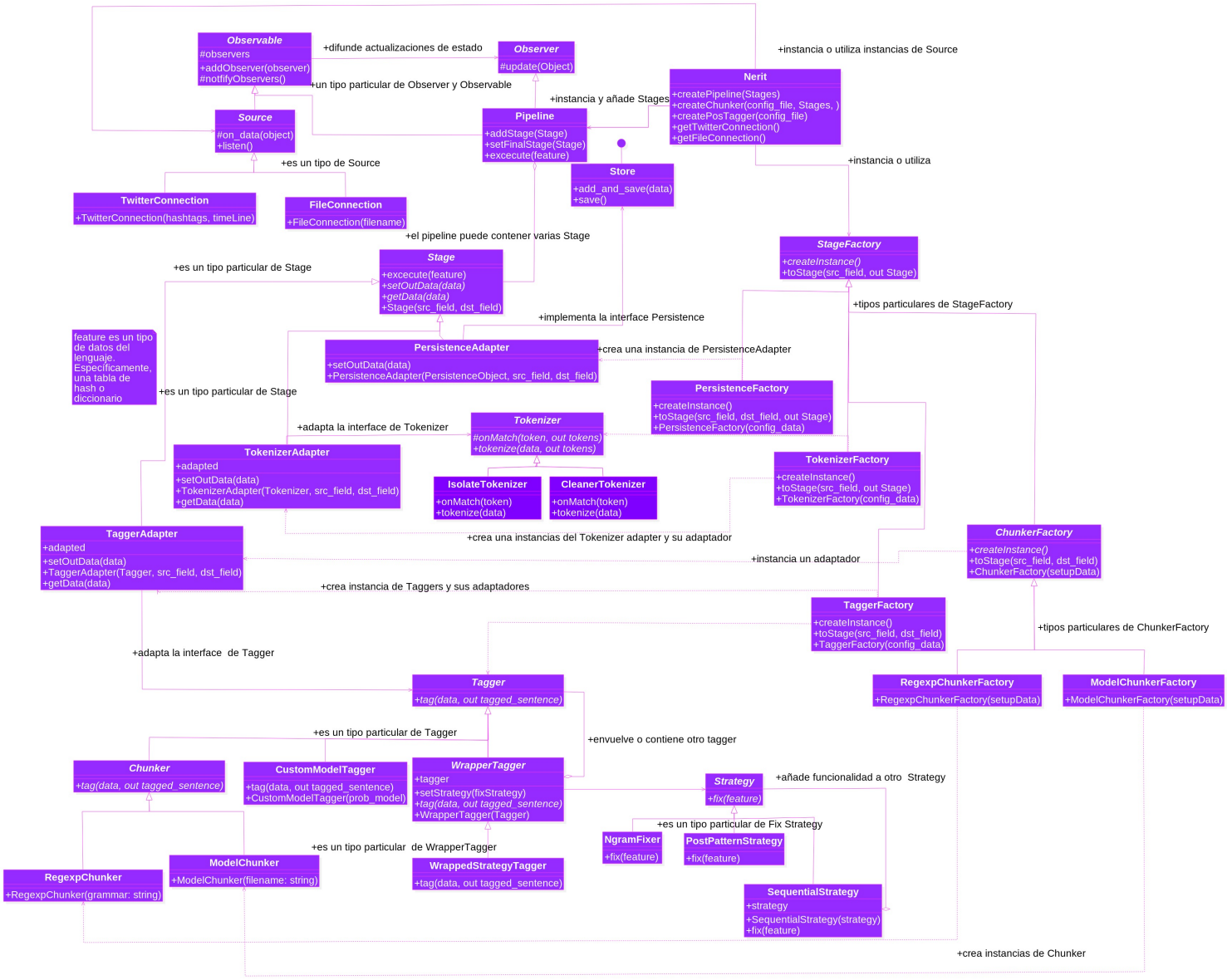

Fig.1. Diagrama de clases de la aplicación.

Apéndice.

1.Archivos de configuración de la aplicación:

1.a Expresiones regulares para tokenización.

En **/data/tokenizers.re** se puede encontrar un archvio de expresiones regulares en formato JSON. En este archivo se puede definir qué tokens se quiere tratar de manera particular. Los distintos tokenizers consultan dichas propiedades y actúan en consecuencia.

"hashtag":

{

***"regex":"**(?P<hashtag>#[\\wá-úñü0-9Á-ÚÜÑ]+)",*

***"replace":"**\\g<hashtag>",*

***"isolate":"**True",*

***"post":"**HTAG"*

}

Según este ejemplo una instancia de la clase IsolateTokenizer, apartará o no dividirá en el texto (*isolate=True*) a los tokens que adhieren con la expresión regular *"regex"*. Adicionalmente, un TitleizeTokenizer puede aplicar la expresión que dicta *"replace"* a los tokens que se emparejan con *"regex"*. Vale destacar, que en la etapa de Part of Speech Tagging, los etiquetadors morfológicos pueden etiquetar los tokens que adhieren a las expresiones regulares *"regex"* con la etiqueta "post". En general, a una misma expresión regular, se le pueden asociar diversas operaciones mediante propiedades.

1.b Expresiones regulares para abreviaturas.

En /**data/abbrebiations.re** puede escribir expresiones regulares para abreviaturas comunes. De este modo, al momento de tokenizar se puede consultar si el token corrriente es una abreviatura.

1.c Creación de modelo de datos para entrenamiento: Part of Speech Tagging y Chunking

En **/config/taggers.ini** se encuentra un archivo de configuración, en el cúal podemos indicar los corpus de datos a utilizar como una lista de archivos, porcentaje de datos para entrenamiento y test, adicionar expresiones regulares, directorio dónde almacenar los modelos creados, extensión de los archivos generados,etc.

2. Uso de la aplicación y parámetros.

La aplicación puede extraer frases mediante un modelo probabilístico o mediante patrones de chunks tags expresados por gramamáticas. En **/src/grammars.py** puede escribir las gramáticas deseadas.
El archivo **/config/nerit.ini** permite indicar la técnica a utilizar; en ausencia de un modelo, se utilizan por defecto las gramáticas definidas. En este archivo puede indicar el modelo a utilizar, si generar un corpus o no y de cúantos datos, archivos de expresiones regulares, etc.

Ejemplos de uso mediante consola de comandos:

Extrayendo frases desde el timeline de una cuenta de Twitter

$> ./nerit.py -tc

Extrayendo frases según hashtags

$> ./nerit.py -tc 'tránsito choque autovía'

Crear un clasificador para etiquetado gramaticales

$>./nertic.py -cpt

y una modelo para extraer frases

$>./nertic.py -cct