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 mediante Procesamiento de Lenguaje Natural de frases sustantivas, en particular, direcciones domiciliarias. 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 especifica 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 escrita en lenguaje Python. 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]. 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 sin descartar las modificaciones anteriores. El objetivo de esto es,eventualmente, poder tener acceso a la historia de todo el procesamiento y facilitar la adicción de nuevas operaciones. Por ejemplo: el texto original puede guardarse asociado a la key 'text'; posteriormente, sus modificaciones, asociadas a keys 'tagged', 'chunked', 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,etc.), abreviaturas, etc. Esta ampliación de vocabulario vuelve la tarea de tokenización más desafiante conforme aumenten los tokens que deseemos detectar o transformar. Por ende, y para conservar en forma intacta dichos tokens correctamente es imprescindible definir *expresiones regulares* que puedan conocer dichos tokens de forma particular (y tratar el resto que no es de utilidad) a fin de que etapas posteriores mejoren la precisión/recall de sus ejecuciones.

El pipeline propuesto permite definir estas expresiones regulares fuera del código fuente, y asociar operaciones

1 http://en.wikipedia.org/wiki/Natural_language_processing
2 http://es.wikipedia.org/wiki/JSON

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. Este último 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 precisión 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 en participio.

Cuando se prueba el etiquetado en los textos de tweets esos inconvenientes persisten y adicionalmente, la precisión 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 ambiguas: 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 que contienen verbos,adverbios y contracciones de uso común.

- Palabras fuera del vocabulario: En la etapa de tokenización se definieron expresiones regulares para detectar tokens que de otro modo se podrían haber desmenuzado 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 (urls,formato de horas, adverbios terminados 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 flexible: si bien no podremos llegar a una precisión de 100% contra datos de prueba muy variados, ante la necesidad el pipeline permite añadir más etapas o añadir responsabilidad a las que se pretende ajustar los n-gramas fallidos (e infrecuentes respecto a un texto correctamente escrito) y corregirlos mediante una heurística adaptada a los casos particulares. Por ejemplo: al clasificar un adjetivo, asegurarnos que efectivamente no sea una conjugación de un verbo en participio invocando una estrategia o política que encapsula un Web Service Client para desambigüar una palabra, o bien determinar la similitud respecto a clusters de palabras símiles. 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

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

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; y su alternativa: 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 precisión resulta una opción muy adecuada para chunks que aparecen como patrones, pero la especifidad de esta técnica conlleva a un **trade-off** entre recall y 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 de la frase con reglas menos privilegiadas o restrictivas resultando en falsos negativos. Es decir, ante el deseo de mantener precisión y mejorar el recall, podemos perder ambos.

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

Es importante aclarar que las direcciones postales no siguen patrones estrictamente definidos (pueden ser números, sustantivos propios/comunes, incluir verbos,etc). Más bien, en nuestra forma de hablar, 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, no es suficiente para detectar la variedad de frases que corresponden a direcciones. 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 a demanda). Las ventajas de este método reside en que no es sintáctico, como el caso de gramáticas, entonces nos permite 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.

El corpus actual se formó a partir de tweets de tránsito automovilístico procesados con ayuda de la técnica de gramáticas y un posterior etiquetado/corrección manual iterativo e incremental. 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**
transitoBsAs ***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 forma parte del mismo chunk, y **O** que no corresponde a una frase significativa.

Por defecto, la aplicación provee un modelo **Naive Bayes**[7] provisto por la librería **NLTK**[8] para extraer frases. El modelo está entrenado sobre el corpus que contiene poco más de 1000 tweets. 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 precisión en IOB tags de hasta 86.% cuando el modelo se entrenó con un 70% de los

6 http://es.wikipedia.org/wiki/Colocación
7 http://es.wikipedia.org/wiki/Clasificador_bayesiano_ingenuo
8 http://www.nltk.org/

datos del corpus, respecto de la alternativa con 75.9%. Adicionalmente, la precisión y recall del modelo entrenado siempre está sujeto a la calidad y variedad del corpus, y las características variables que deseemos extraer de los datos entrenados (si extraemos características que no aportan a la búsqueda, tendremos resultados pobres, si extraemos sólo pos tags, no se reconocerán frases).

Por otro lado, en términos de desarrollo del software, está técnica se impone sobre la técnica de extracción de chunks por gramáticas pues el proceso para extraer más direcciones *no cambia programáticamente*, sino que sólo implica enriquecer el corpus con aquellas instancias de datos que deseemos clasificar.

Entonces, NERIT ofrece dos métodos para extraer frases. Si se desea extraer otro tipo de frase o aplicar correcciones, estos métodos pueden combinarse e intercambiara: podemos extraer las direcciones probabilísticamente y, por dar un ejemplo arbitrario, fechas mediante gramáticas. 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. Las siguientes tablas muestran la confusión del clasificador Naive Bayes al momento de asignar el tag de direcciones LOCATION. Sólo aparecen las frases candidatas más frecuentes, aquellas cuya varianza entre las posibles etiquetas del corpus son muy cercanas a LOCATION y la clase gramatical de las palabras que provocan etiquetados incorrectos. Al mismo tiempo, los experimentos muestran el efecto gradual de combinar ambos clasificadores utilizando el corpus elaborado y las gramáticas; estas últimas son las que provee la aplicación por defecto (archivos: grammars_fix y grammars).

Palabras\Sintagmas	Dirección	Preposicional	Sustantiva	Evento	Conjunción
Preposiciones	0,68	0,32			
Numéricos	0,53		0,3	0,17	
Artículos	0,86		0,07	0,06	
Conjunciones	0,88		0,01	0,07	0,04

Tabla 1. TP:0.82 FN:0.18 Entrenado al 0.4 sin utilizar gramáticas

Palabras\Sintagmas	Dirección	Preposicional	Sustantiva	Evento	Conjunción
Preposiciones	0,88	0,12			
Numéricos	0,5		0,34	0,16	
Artículos	0,96		0,02	0,02	
Conjunciones	0,95		0,01	0,04	0,01

Tabla 2 .TP:0.838 FN:0.162 Entrenado al 0.4 apoyado en gramáticas sólo de direcciones

Palabras\Sintagmas	Dirección	Preposicional	Sustantiva	Evento	Conjunción
Preposiciones	0,91	0,09			
Numéricos	0,5		0,34	0,16	
Artículos	0,97			0,03	
Conjunciones	0,9			0,04	0,06

Tabla 3. TP: 0.86 FN:0.14 Entrenado al 0.8 y gramáticas sólo de direcciones

Palabras\Sintagma	Dirección	Preposicional	Sustantiva	Evento	Conjunción
Preposiciones	0,97	0,09			
Numéricos	0,96		0,03	0,01	
Artículos	0,94		0,02	0,02	
Conjunciones	0,97			0,01	0,02

Tabla 4. TP: 0.869 FN:0.13. Entrenado al 0.8 y utilizando gramáticas para atrapar direcciones más variadas y eventos de tránsito.

Key:

- **TP**: Verdaderos positivos: instancias correctamente clasificadas en la clase que pertenecen.
- **FN:** Falsos negativos: instancias incorrectametne clasificadas que deberían y no están en su clase.

De las tablas se infiere que cuando el modelo tiene datos de muy símiles, la clasificación se vuelve nebulosa. Una forma de sesgar a una categoría particular es aumentar datos de entrenamiento para decantar o direccionar la certidumbre al clasificar, así los falsos negativos se vuelven sensibles más sensibles o maleables en el proceso. Como corolario de esto surge algo llamativo: los modelos pueden sobre entrenarse, esto es que el clasificador aparentemente llega a un máximo y ante más datos merma su precisión pues vamos de nuevo a una situación de datos símiles. Otra observación es que ante la falta de datos de entrenamiento, utilizar gramáticas favorece la efectividad confinada *exclusivamente a* las frases para la cual se ha programado, a raíz de esto no se ven mejoras globales en el recall del algorítmo. Por otro lado, (y siempre apuntando a direcciones domiciliarias) el efecto de las gramáticas es inverso al de un modelo probabilístico en función de su tamaño y calidad; de ahí en más la predicción tiende a ser guiada por los datos opacando dichas expresiones.

3. Diseño de clases de NERIT.

A continuación se expone una justificación concisa sobre las clases relevantes de la aplicación. En los diagramas sólo aparecen los métodos destacados.

Source: es esperable que la aplicación sea capaz de obtener datos desde diferentes fuentes, esto promueve la creación de una interface estándar a la cual adhieran las distintas vías de captación de datos. Para reaccionar ante los

datos que arrivan y procesarlos, NERIT instancia el patrón Observer. Así el fetching de información queda desacoplado de los receptores, pudiendo estos variar en número y comportamiento independientemente entre ellos y de las fuentes de datos. Pipeline el único suscrito a los eventos emitidos en etapa de fetching. Dado que esta clase plasma un comportamiento de ejecución fijo y de tipo Pipe and Filters, añadiendo posibilidad de una operación específica desconocida a priori, se propone el patrón de diseño Template Method. De esta manera, las etapas que integren el pipe seguirán su operatoria quedando confinadas a un marco de trabajo. Al culminar el procesamiento es probable requerir su notificación, en consecuencia Pipeline es al mismo tiempo una instancia del patrón Observer. Por lo tanto, está clase tiene embebido un comportamiento de Observer y Observable.

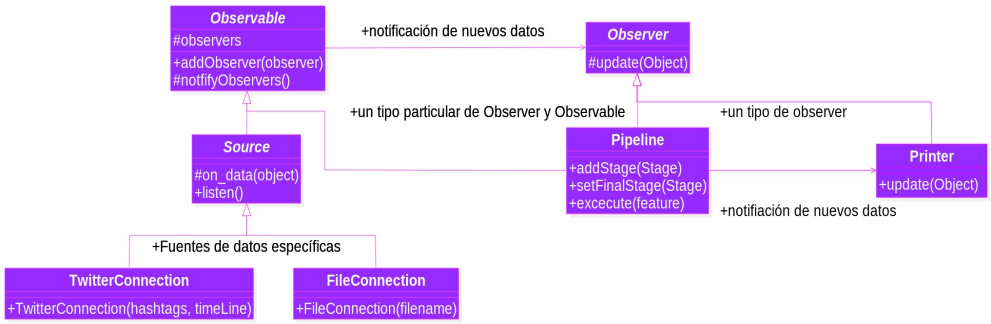
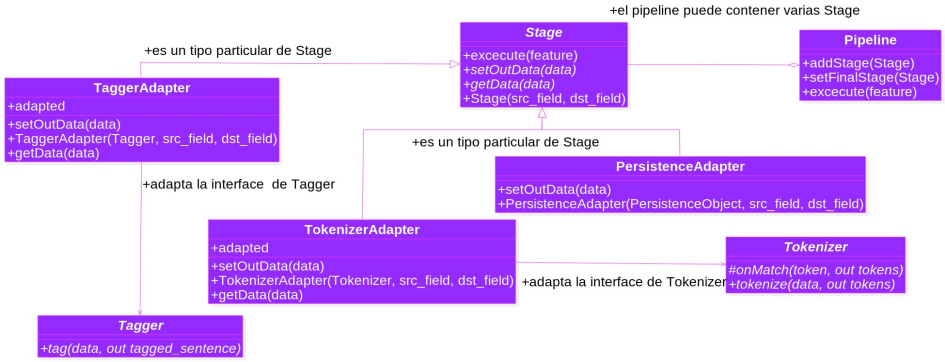


Fig 1. Patrones Observer y Template Method

Printer, sólo una clase de aplicación específica que muestra el resultado del Pipe.

Stage: el procesado de los tweets requiere pasos heterogéneos. En general, etapas diferentes tienen signaturas diferentes y será estrictamente necesario cambiar sus interfaces e invocaciones para interoperar dentro del Pipeline. La solución en NERIT no requiere cambios sino extensiones a las clases. Dicho esto, se utilizarán los patrones Adapter y Template Method correspondientes a cada nueva etapa.

Fig 2. Patrones Adapter y Template Method



Las etapas de tokenización de texto tienen un comportamiento común, pues todas dividen el texto ante espacios. Las transformaciones y/o filtrado del texto para determinados grupos de tokens son especificas individualmente. Esta problemática suguiere el patrón de diseño Template Method.

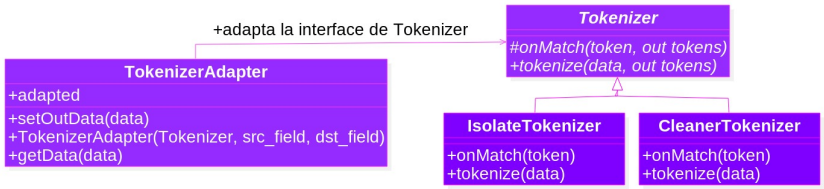


Fig 3. Patrón Template Method

El dilema de reconcer frases fue inicialmente encarado utilizando gramáticas sobre las palabras y pos tags. Conforme aparecían nuevos casos, la gramática se volvía más compleja, así mismo la adición de nuevas por las relaciones de prioridad entre ellas. Para reducir la complejidad se optó por encontrar primero los sintagmas corrientes y comunes del idioma Español (frases sustantivas, adverbiales,etc) y, luego ,sobre estos buscar las frases concretas o bien aplicar acciones correctivas. De la experiencia de los resultados se ve que hay responsabilidad que va creciendo y combinándose conforme se conocen más datos a incluir. Como respuesta a este desafío se exponen las soluciones de diseño Decorator y Strategy. En la figura RegexpChunker es el tipo de chunker que trabaja con expresiones regulares, y CustomModelTagger y ModelChunker sobre modelos entrenados. Con este diseño, estudiando los casos de fallas o bien reconocer frases no contempladas en los datos de entrenamiento tiende a ser viable mediante combinaciones apropiadas de sus instancias.

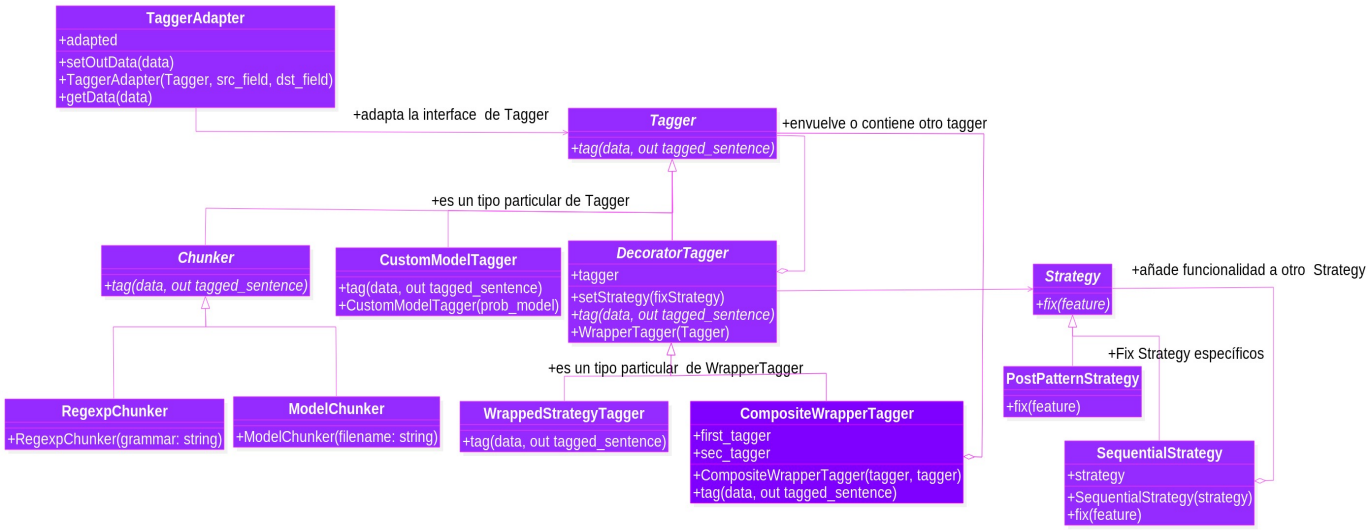


Fig 3. Patrón Decorator y Strategy

Evidentemente la instanciación de objetos deriva en un problema si es proporcional al número y construcción de objetos cuando estos establecen relaciones de composición o deben instanciarse de forma relacionada. Recordando que las etapas que intervienen en la secuencia serán distintas y algunas compuestas, se propone simplificar esta manipulación mediante fábricas de objetos, que permitan eventualmente crear objetos compuestos (decorados) o bien

más grandes (sencuencias de Tokenizers) y en cualquier caso con interfaces acondicionadas. En este contexto, presentamos una combinación de Factory Method y Facade que convergen en la solución de diseño Abstract Factory; uno por cada etapa de uso *frecuente y escencial* para la funcionalidad básica de la aplicación : TokenizerFactory, ModelChunkerFactory, RegexpChunkerFactory y TaggerFactory son ejemplos de aplicación y se muestran en la Fig 4.

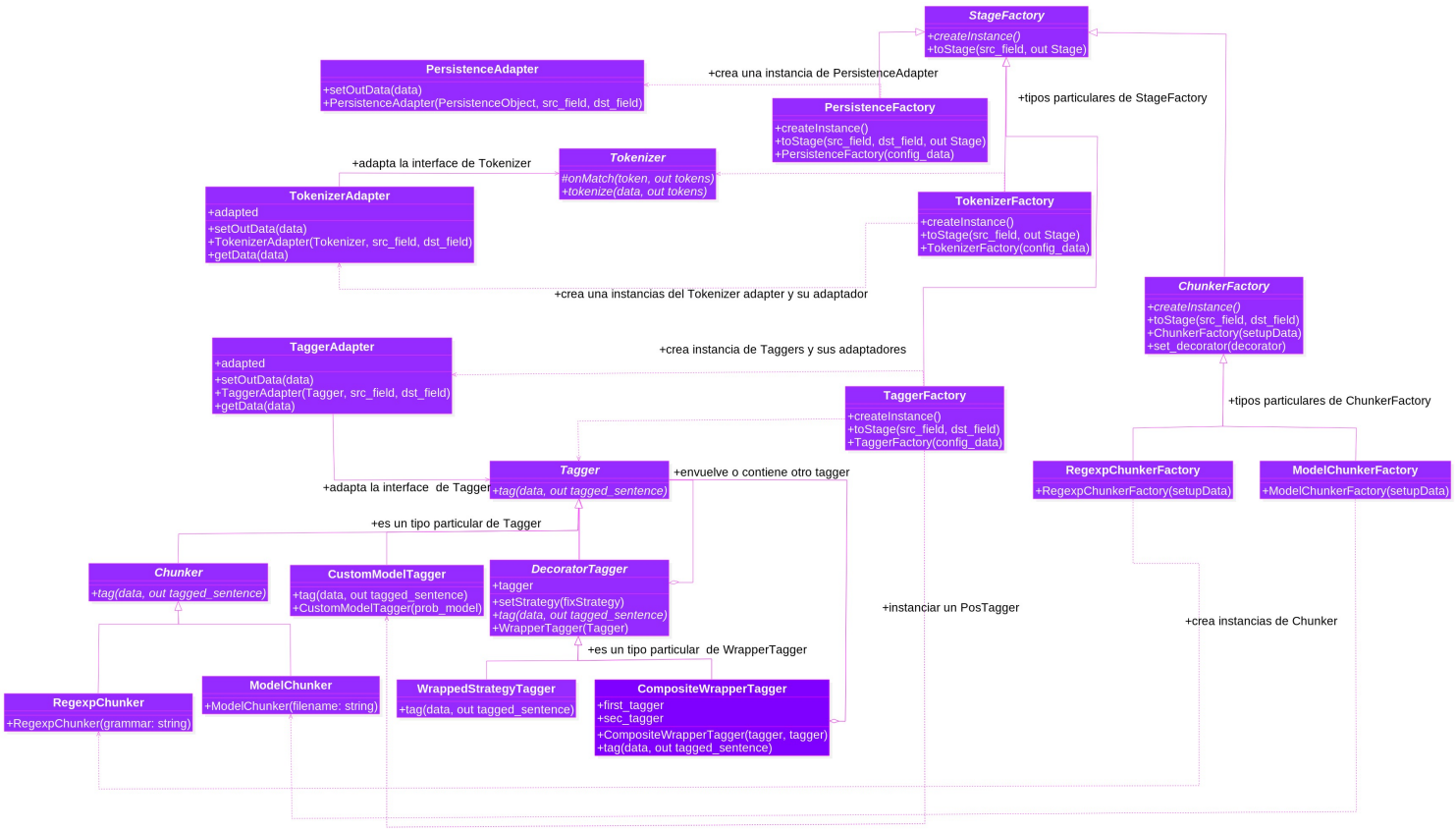


Fig 4. Entre otros, en esta imagen se intenta destacar el Patrón Abstract Factory

Finalmente, el uso de una configuración de clases semejantes da lugar a la confusión de uso. Se hace necesario proveer un nivel de abstracción más alto para acceder a toda la funcionalidad de manera más simple y memorable al momento de dar un uso concreto. El desenlace de esta problemática propone el uso del patrón Facade, en este escenario es NeritFacade; a través de esta fachada es posible configurar el toda la funcionalidad del Pipeline de una manera más sencilla. A esta altura, se presenta el diagrama de clases final de NERIT.

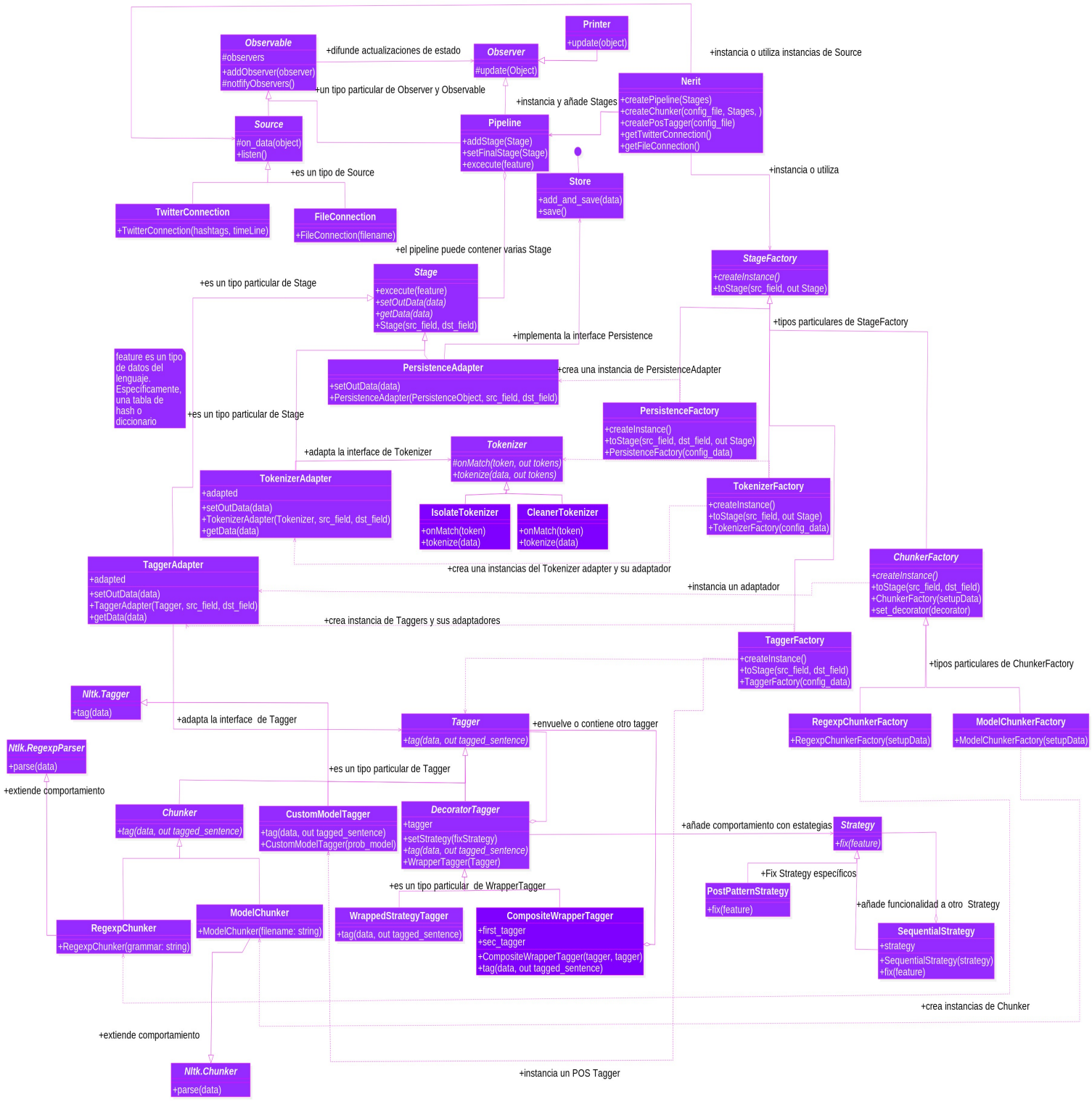


Fig.5. Diagrama de clases de la aplicación. Aparecen las librerias principales de NLTK

4.Conclusion académica.

Durante el desarrollo del proyecto se ha intentado transmitir la dificultad de extraer información de los tweets, para esto se implementaron las etapas distinguidas y se enfatizó en la complejidad de cada una.
De las técnicas para extraer frases del lenguaje natural se evaluaron dos de ellas, una requirió armar un conjunto de datos anotado casi manualmente. De esta experiencia se observó las limitaciones y bondades de cada una: queda a la luz el hecho de que trabajar con datos aprendidos y serializados en un objeto no solo captura mucha de la experiencia del campo que queremos detectar si no que una vez listo, disminuye el tiempo de desarrollo; esto resalta contra utilizar gramáticas y la atención que requiere mantenerlas e implementarlas. Finalmente destaco el uso de patrones de diseño, ya que facilitó subir el nivel de abstracción y referirme a subconjuntos del proyecto con una palabra y hablar en sus términos permitió hacer combinaciones fáciles en el código, tales como realizar las evaluaciones de los algorítmos del proyecto que dieron contexto a este informe.

5.Biliografía.

- Python Text Processing with NLTK 2.0 – Jacob Perkins . Ed Packt Publishing
- Design Patterns “Elements of Reusalbe Object-Oriented Software – Erich Gamma, Ralph Johnson John Vlissides. Ed Addison Wesley

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 que tamaño, archivos de expresiones regulares, etc.

Ejemplos de uso mediante consola de comandos:

Ante de ejecutar la aplicación se necesita activar el entorno.
En el directorio raíz del proyecto, hacer:

$> source bin/activate

Luego, en el directorio donde se encuentra la apliación (comunmente src/) ya puede ejecutarse.

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