Una Implementación del Modelo DMV+CCM para Parsing No Supervisado

Franco M. Luque
Universidad Nacional de Córdoba & CONICET
Córdoba, Argentina
francolq@famaf.unc.edu.ar

Resumen

En este trabajo describimos los modelos de parsing no supervisado de Klein y Manning (2004) CCM, DMV y DMV+CCM, y presentamos una implementación de éstos en lenguaje Python y sobre la plataforma NLTK. El software posee un diseño orientado a objetos que permite definir fácilmente diversas variantes de los modelos. Además, define las abstracciones necesarias para proveer una plataforma de experimentación completa, con baselines, lectores de corpus, árboles de distintos tipos, parsers y evaluadores. Experimentos con el corpus de inglés WSJ10 permiten ver que nuestra implementación reproduce con bastante aproximación los resultados originales. El software se distribuye bajo licencia GNU GPL v3, siendo a la fecha la única implementación de DMV+CCM libremente disponible para la comunidad.

1. Introducción

Uno de los problemas centrales de la Lingüística Computacional es el desarrollo de parsers, esto es, analizadores sintácticos de oraciones de una lengua en particular. Los parsers supervisados aprenden la gramática de una lengua a partir de un corpus de oraciones previamente analizadas por lingüistas. En cambio, los parsers no supervisados sólo aprenden a partir de texto sin analizar, de la misma manera que lo hacen los seres humanos [9].

La investigación en el área de parsing no supervisado recibió un importante impulso a partir de la publicación de los modelos CCM, DMV y DMV+CCM [6, 7, 8, 5], ya que fueron los primeros modelos en superar en performance a los *baselines*, logrando resultados muy superiores para los idiomas inglés, alemán y chino. Actualmente, los sistemas que son estado del arte en cuanto a parsing no supervisado de dependencias están basados en DMV [2, 4, 12].

CCM y DMV son modelos probabilísticos generativos que modelan diferentes aspectos sintácticos, y DMV+CCM es la combinación de éstos. CCM (*Constituent-Context Model*, o modelo de constituyentes y contextos) modela árboles de constituyentes sin etiquetas, mientras que DMV (*Dependency Model with Valence*, o modelo de dependencias con valencia) modela estructuras de dependencias proyectivas. Los modelos se entrenan con el algoritmo de maximización de esperanza EM, por lo que la selección apropiada de una instancia inicial dentro de los modelos es esencial para obtener buenos resultados.

En este trabajo presentamos una implementación de los modelos CCM, DMV y DMV+CCM. Los aspectos generales de los modelos son descriptos con bastante claridad en [5]. Sin embargo, para lograr una implementación completa que reproduzca, al menos aproximadamente, los resultados publicados, se debieron tomar varias decisiones que no estaban especificadas. En este trabajo hacemos primero una descripción general de los modelos, y luego describimos los detalles de implementación.

La implementación de los parsers se incluye en el paquete de software `lq-dmvccm`. Éste depende del paquete `lq-nlp-commons` que, junto con el Natural Language Toolkit (NLTK) [1], proveen una plataforma para el desarrollo de parsers y la experimentación con corpus. Todo el software se distribuye bajo licencia GNU GPL v3. [1] En este trabajo también hacemos una descripción de la arquitectura y el diseño de nuestro sistema, y cómo se integra éste con Python y NLTK.

[1] Disponible en `http://www.cs.famaf.unc.edu.ar/~francolq/en/proyectos/dmvccm`.

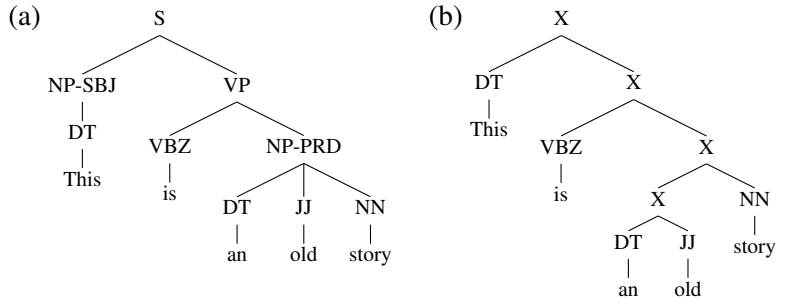

Figura 1: Ejemplos de (a) árbol de constituyentes clásico y (b) bracketing binario.

Finalmente, realizamos experimentos con el corpus del inglés WSJ10 para comparar nuestros resultados con los resultados publicados originalmente. Del análisis de los resultados se desprende que, si bien no se han logrado reproducir exactamente todos los resultados, la implementación funciona correctamente, y en algunos casos con mejores resultados que los originales.

2. Los Modelos

2.1. CCM

CCM es un parser de árboles binarios de constituyentes sin etiquetar, también llamados bracketings binarios o, en este trabajo, simplemente bracketings (Figura 1(b)). Un bracketing de una oración s puede ser visto como una matriz booleana B, a dónde B_{ij} indica si la subcadena ${}_is_j = s_i \dots s_{j-1}$ es un constituyente. Para definir un árbol binario, B debe tener una cantidad maximal de constituyentes que no se cruzan entre sí. [2] Las subcadenas de una oración que no son constituyentes son llamadas distituyentes.

CCM es un modelo generativo, que define la probabilidad $P(s|B)$ de generar una oración en términos de un bracketing dado. Cada elemento B_{ij} de la matriz genera una subcadena ${}_is_j$ y un contexto (s_{i-1}, s_j)[3], con una distribución condicionada en el valor binario de B_{ij}. Como las subcadenas y los contextos se solapan, sólo algunas formas de generarlos resultan en oraciones válidas, por lo que $P(s|B)$ es en realidad deficiente en masa.

Más formalmente,

$$P(s,B) = P_{bin}(B)P(s|B)$$

a dónde P_{bin} es la distribución uniforme sobre bracketings binarios y

$$P(s|B) = \prod_{i,j:i\leq j} P_{span}({}_is_j|B_{ij})P_{ctx}(s_{i-1}, s_j|B_{ij}).$$

Los parámetros del modelo son los valores de las cuatro distribuciones $P_{span}(\alpha|b)$, $P_{ctx}(\beta|b)$ con $b \in \{true, false\}$. Son aprendidos iterativamente vía EM de manera de maximizar la likelihood $\prod P(s)$ de un conjunto observado de oraciones, considerando a los bracketings como los datos no observados. En cada paso E del algoritmo, se calcula para cada oración la probabilidad de cada bracket (i, j),

$$P_b(i, j|s) = \sum_{B:B_{ij}} P(B|s)$$

en términos de los valores anteriores de los parámetros. Esto se puede hacer con un programa dinámico que calcula las probabilidades *inside* y *outside* para todos los brackets [9].

[2] Las subcadenas vacías nunca son consideradas constituyentes, mientras que las subcadenas de largo uno siempre lo son.

[3] Definimos $s_{-1} = s_{|s|} = \diamond$.

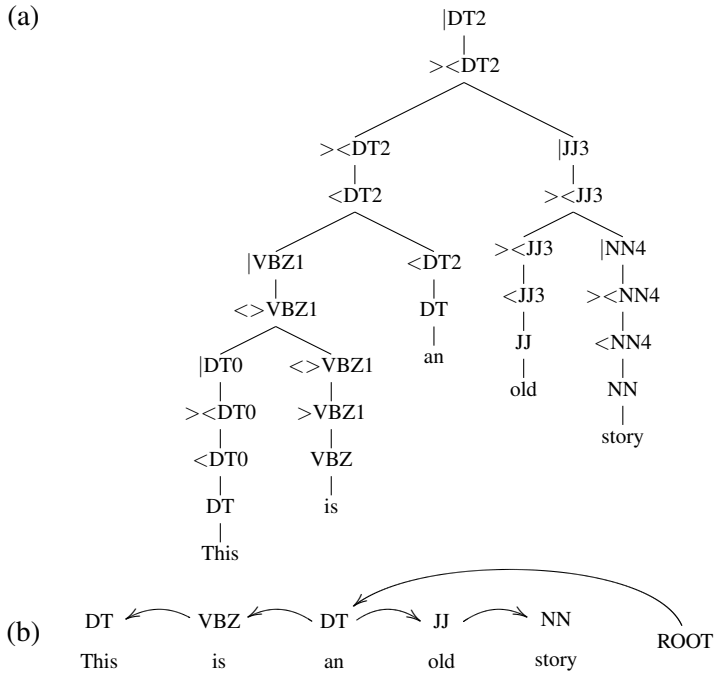

Figura 2: (a) Ilustración del proceso generativo de DMV para árboles de dependencias proyectables. (b) Estructura de dependencias resultante (sintácticamente incorrecta, por cierto).

En el primer paso E, en vez de usar $P(B|S)$ con valores iniciales para los parámetros, se utiliza una distribución independiente de las oraciones $P_{split}(B)$. Esta distribución favorece los árboles desbalanceados de una manera independiente del lenguaje.

En el paso M del algoritmo, los parámetros son reestimados usando las fórmulas

$$P_{span}(\alpha|true) = \frac{\sum_{i,j,s:\ _is_j=\alpha} P_b(i,j|s)}{\sum_{i,j,s} P_b(i,j|s)}$$

$$P_{span}(\alpha|false) = \frac{\sum_{i,j,s:\ _is_j=\alpha} (1-P_b(i,j|s))}{\sum_{i,j,s} (1-P_b(i,j|s))} \quad (1)$$

para cada subcadena α, y usando fórmulas análogas para cada contexto β.

2.2. DMV

DMV codifica un proceso generativo top-down (de arriba hacia abajo) de una estructura de dependencias, a donde las palabras generan sus dependientes en ambas direcciones hasta que deciden detenerse.

Existen dos variantes del modelo DMV, dependiendo de si se utiliza o no la restricción denominada *"one-side-first"* (un lado primero). La variante sin la restricción se encuentra muy vagamente explicada en [5], y no hemos sido capaces de encontrar una definición precisa que dé resultados satisfactorios, por lo que nos concetraremos en la otra variante.

En la variante con la restricción *"one-side-first"* , cada palabra h elige primero, con probabilidad $P_{order}(dir|h)$, una dirección en la que generar dependientes ($dir \in \{$*left-first, right-first*$\}$). Luego, decide qué dependientes generar en esa dirección y cuándo parar, con probabilidades dependientes de la palabra y de la dirección. La decisión de parar se toma con probabilidad $P_{stop}(true|h,dir,adj)$, a dónde $dir \in \{l,r\}$ y adj es un booleano que indica si todavía no se ha generado ningún dependiente. Si decide no

parar, se genera un dependiente a con probabilidad $P_{attach}(a|h,dir)$. Cuando decide parar, h pasa a generar dependientes en la otra dirección, utilizando el mismo proceso.

La Figura 2 ilustra este proceso, utilizando los marcadores $<,>$ y $|$ sobre los no terminales para indicar el estado en el proceso generativo en cada momento. La marca $|$ señala el estado inicial, mientras que $<$ y $>$ se utilizan para señalar el orden y la dirección en la que se está generando. Por ejemplo, $><$DT2 indica que DT2 decide generar primero hacia la derecha, y $<$DT2 indica que terminó de generar hacia la derecha y se encuentra generando hacia la izquierda.

Si definimos $D(h,dir)$ como la secuencia de dependientes de h en la dirección dir, podemos resumir la probabilidad de generar todos los dependientes de h como

$$P(D(h)) = P_{order}(o|h) \prod_{dir \in \{l,r\}} \left(\prod_{a \in D(h,dir)} P_{stop}(f|h,dir,adj) P_{attach}(a|h,dir) \right) P_{stop}(t|h,dir,adj).$$

De esta manera, la probabilidad de un árbol de dependencias completo D sobre una oración s es

$$P(D) = P(D(\text{ROOT})) \prod_{h \in s} P(D(h)).$$

Al igual que con CCM, los parámetros de DMV también pueden ser aprendidos utilizando el algoritmo EM. En este caso, en cada paso E se calcula para cada oración s la probabilidad $c_s(x:i,j)$ de que determinado no-terminal x, con su respectiva marca, genere el substring ${}_i s_j$, En el paso M, los valores de los parámetros son reestimados utilizando estas cantidades. Las fórmulas de reestimación se pueden encontrar en el Apéndice A.2 de [5].

2.3. DMV+CCM

DMV+CCM no es otra cosa que la combinación de CCM y DMV. Que esta combinación se puede realizar fácilmente se puede ver en el hecho de que las estructuras de dependencias modeladas por DMV son también árboles binarios, y por lo tanto se pueden incorporar al proceso generativo los factores correspondientes a los constituyentes y contextos. Todos los aspectos del entrenamiento EM se pueden adaptar a este esquema sin mayores cambios.

3. Detalles de Implementación

3.1. Inicialización

A diferencia de CCM, la inicialización de DMV se encuentra definida en [5] con muy poca precisión, por lo que fue necesario explorar diferentes posibilidades hasta encontrar una que permitiera al algoritmo EM alcanzar los resultados esperados. El objetivo de la inicialización es guiar al entrenamiento hacia estructuras lingüísticas correctas, en un sentido general e independiente del idioma en cuestión.

Por ejemplo, inicialmente se desea asignar más probabilidad a dependencias de corto rango que a dependencias de largo rango. Para ello, inicialmente se condiciona la generación de dependencias utilizando un parámetro adicional de distancia $dist$, y cuando $h \neq$ ROOT, se define

$$P_{attach}(a|h,dist,dir) = \frac{1}{1+dist}.$$

Si $h =$ ROOT, P_{attach} es uniforme.

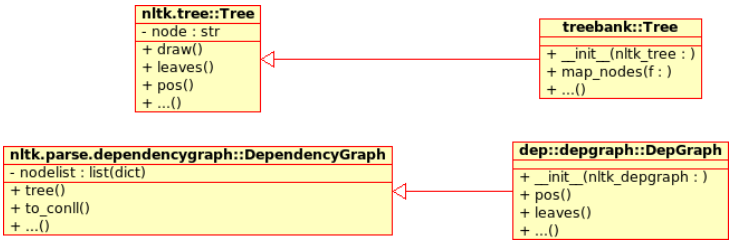

Figura 3: Estructuras de constituyentes y de dependencias de NLTK y `lq-nlp-commons`.

También se desea asignar más probabilidad de detenerse cuando ya se ha generado un dependiente que cuando no se ha generado ninguno. En nuestra implementación, elegimos inicializar con

$$P_{stop}(true|h,dir,adj) = \begin{cases} 0{,}25 & \text{si } adj \\ 0{,}75 & \text{si no } adj \end{cases}$$

En el caso del orden de generación no se pueden hacer consideraciones generales, por lo que inicialmente definimos $P_{order}(dir|h) = 0{,}5$.

3.2. Evaluación

Las métricas de evaluación utilizadas se describen en [5], faltando definir sólo algunos detalles menores.

Como los parsers obtenidos devuelven tanto estructuras de constituyentes como de dependencias, se pueden evaluar los sistemas utilizando las métricas existentes para ambos formalismos.

En el caso de las estructuras de constituyentes, las métricas utilizadas son *unlabeled precision* (UP), *unlabeled recall* (UR) y la media harmónica entre éstas (UF_1). Las oraciones de una sola palabra no son consideradas en la evaluación. Los árboles del corpus de referencia, llamado *gold standard*, son primero convertidos a conjuntos de bracketings (no necesariamente binarios), eliminando los brackets producto de producciones unarias. Los brackets correspondientes a las oraciones completas siempre son considerados en la evaluación, y por lo tanto esto resulta en valores más altos para las medidas, ya que siempre son clasificados correctamente.

Para las estructuras de dependencias, se utilizan las métricas de precisión dirigida (*Dependency Accuracy*, o Dir) y precisión no-dirigida (*Undirected Accuracy*, o Undir).

4. Arquitectura y Diseño

La implementación de DMV+CCM se apoya en otro paquete de software de nuestra autoría, denominada `lq-nlp-commons`, que reúne aquellas herramientas que fuimos desarrollando y que pueden ser útiles para tareas de Procesamiento de Lenguaje Natural en general. Apoyándose en la plataforma NLTK [1], `lq-nlp-commons` define un conjunto de abstracciones que permiten definir y evaluar diferentes modelos de parsing, trabajando con corpus de diferentes tipos e idiomas.

4.1. Estructuras de Datos Básicas

En la Figura 3 se puede ver un diagrama de clases con las estructuras de datos para árboles de constituyentes y de dependencias. A la izquierda se encuentran las clases básicas definidas por NLTK, y algunos métodos y atributos que poseen. A la derecha se encuentran subclases definidas en `lq-nlp-commons`,

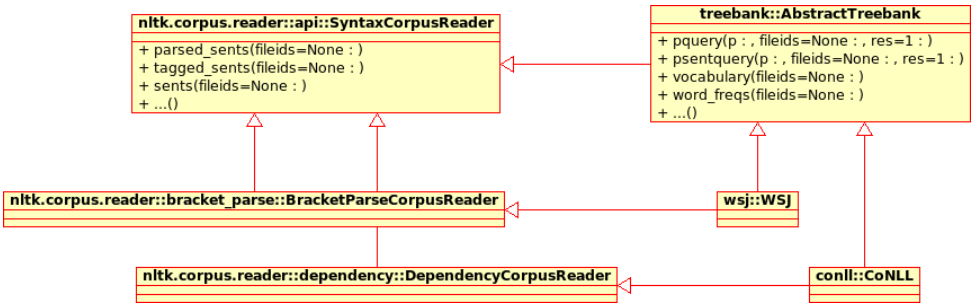

Figura 4: Lectores de corpus de NLTK y `lq-nlp-commons`

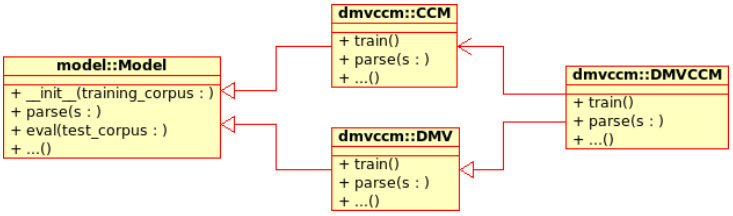

Figura 5: Clase abstracta `Model`, de `lq-nlp-commons`, y subclases concretas de `lq-dmvccm`.

que actúan como envoltorios o *wrappers* de las anteriores, agregando funcionalidad a través de nuevos métodos.

4.2. Corpus

En la Figura 4 podemos ver un diagrama con algunas de las clases utilizadas para leer datos de corpus. La interfaz básica `SyntaxCorpusReader` define los métodos de la forma `*_sent`, todos éstos devolviendo iteradores sobre las oraciones del corpus. En `sents`, los ítems son listas de palabras que representan las oraciones; en `tagged_sents`, son listas de pares de la forma (palabra, tag); en `parsed_sents`, son las estructuras sintácticas, cuyo tipo va a depender del corpus, como se explica a continuación.

Para leer corpus de constituyentes en formato similar al Penn Treebank [10], se usa la clase `BracketParseCorpusReader`, en cuyo caso los ítems devueltos por `parsed_sents` serán árboles de constituyentes de tipo `Tree` de NLTK. Para leer corpus de dependencias en formato similar al de los corpus CoNLL 2006 y 2007, se usa la clase `DependencyCorpusReader`, y en este caso los ítems devueltos por `parsed_sents` serán de instancias de la clase `DependencyGraph` de NLTK.

Las clases `WSJ` y `CoNLL` de `lq-nlp-commons` son dos ejemplos de corpus concretos definidos a partir de los lectores de corpus de NLTK. Gracias a la herencia múltiple, los corpus definidos heredan funcionalidad adicional definida en `AbstractTreebank`, como por ejemplo funciones de búsqueda de oraciones o estadísticas de palabras. Además, estas clases se encargan de "envolver" los ítems devueltos por `parsed_sents` con las clases propias de `lq-nlp-commons` presentadas en la sección anterior.

4.3. Modelos

En la Figura 5 se puede ver el diagrama de clases para los modelos DMV, CCM y DMV+CCM. Todos heredan de la superclase `Model` de `lq-nlp-commons`, que define una interfaz general para modelos de parsing y provee el sistema de evaluación, entre otras cosas. La evaluación se hace de manera inteligente, dependiendo de la naturaleza del parser y del corpus de evaluación provisto. De esta manera, puede devolver métricas para constituyentes, para dependencias, o para ambas estructuras.

DMV+CCM se define como una subclase de DMV. Utiliza internamente una instancia de CCM y redefine algunos de los elementos heredados para incorporar los parámetros de CCM.

5. Experimentos

A continuación presentamos los resultados obtenidos en los experimentos realizados con el corpus WSJ10 y los comparamos con los resultados presentados por Klein en [5]. Dicho corpus está conformado por las 7422 oraciones del WSJ Penn Treebank [10] cuyo largo es de a lo sumo 10 palabras luego de eliminar signos de puntuación y moneda. [4] Los modelos son entrenados y evaluados sobre las oraciones de etiquetas POS. Las entradas léxicas son ignoradas. Además, vale aclarar que se utiliza el conjunto completo de oraciones tanto para entrenar como para evaluar.

Para obtener las dependencias del WSJ10, se utilizó una implementación propia de un *head-finder* con las reglas de Yamada y Matsumoto [13]. En [5] se utilizan las reglas de Collins [3], pero no se especifica la implementación utilizada. Los resultados para los baselines RHEAD y LHEAD sugieren que de todas maneras los conjuntos de dependencias utilizados no difieren demasiado.

Los resultados pueden verse en el Cuadro 1. DMV alcanza el valor mostrado en la 20-ésima iteración del entrenamiento. CCM converge a los valores dados desde la 40-ésima iteración. La performance de DMV+CCM empieza a decrecer a partir de la novena iteración, alcanzando un pico de $UF_1 = 76.1\,\%$, Dir = 47.2 %. Los resultados de la tabla fueron tomados arbitrariamente de la décima iteración.

Lo más llamativo es la gran diferencia que hay en las métricas de constituyentes para DMV, ya que estamos obteniendo resultados muy superiores (UF_1 de 65.3 % vs. 52.1 %). Este fenómeno ya ha sido observado en otros trabajos que reproducen los resultados para DMV [11]. Creemos que la diferencia puede ser debido a que los números de Klein corresponden a la versión sin la restricción *"one-side-first"* o a una versión con una mala inicialización.

Los resultados casi idénticos para CCM muestran que posiblemente hemos podido reflejar con exactitud el modelo propuesto por Klein. Lo que no pudimos entender del todo es el origen de los resultados de CCM para dependencias (Dir y Undir), ya que CCM es un parser sólo de constituyentes. Entendemos que sería posible parsear dependencias con CCM, aunque obteniendo resultados bastante aleatorios, pero no hemos realizado experimentos al respecto.

Para DMV+CCM, puede verse que nuestros resultados están aún por debajo de los originales (UF_1 de 75.9 % vs. 77.6 %). En primer lugar, suponemos que el número publicado por Klein corresponde al pico de performance, ya que no se da ninguna información acerca de la iteración elegida, y está claro que el entrenamiento empieza a diverger en determinado momento. De todas formas nuestro pico de $UF_1 = 76.1\,\%$ sigue por debajo. Una posible fuente de la diferencia es la inicialización de DMV. Otra, puede ser que los resultados originales correspondan a la versión sin la restricción *"one-side-first"*.

6. Conclusiones

Presentamos una implementación de los modelos DMV, CCM y DMV+CCM, y una plataforma de software para el desarrollo de modelos de parsing y la experimentación con corpus. Encontramos la forma de definir los detalles necesarios para completar la implementación, principalmente en cuanto a la inicialización de los modelos. Describimos los componentes del software, mostrando las diferentes abstracciones de datos y cómo éstas permiten definir parsers y utilizar corpus de diferentes idiomas. En nuestros experimentos, logramos reproducir exactamente los resultados para CCM, mientras que superamos ampliamente los de DMV, y estuvimos sólo un poco por debajo de los resultados originales para

[4] Este preprocesamiento está descripto en detalle en [5].

	Klein (2005) [5]					`lq-dmvccm`				
	UP	UR	UF1	Dir	Undir	UP	UR	UF1	Dir	Undir
LBRANCH/RHEAD	25.6	32.6	28.7	33.6	56.7	25.7	32.6	28.7	33.5	56.4
RBRANCH/LHEAD	55.1	70.0	61.7	24.0	55.9	55.2	70.0	61.7	23.7	55.6
DMV	46.6	59.2	52.1	43.2	62.7	58.3	74.1	65.3	49.0	65.5
CCM	64.2	81.6	71.9	23.8	43.3	64.3	81.6	71.9		
DMV+CCM	69.3	88.0	77.6	47.5	64.5	67.9	86.2	75.9	47.0	64.5

Cuadro 1: Comparación de los resultados de Klein con los resultados propios, para los baselines y los modelos sobre el corpus WSJ10.

DMV+CCM. Hasta donde sabemos, no se encuentran disponibles para la comunidad otras implementaciones de estos modelos.

Agradecimientos

Este trabajo fue financiado en parte por el proyecto PAE-PICT 2007-02290 de la Agencia Nacional de Promoción Científica y Técnológica (ANPCyT), Argentina.

Referencias

[1] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, 1 edition, July 2009.

[2] Shay B. Cohen and Noah A. Smith. Shared logistic normal distributions for soft parameter tying in unsupervised grammar induction. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL '09, pages 74–82, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[3] Michael J. Collins. *Head-driven statistical models for natural language parsing*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1999.

[4] William P. Headden, Mark Johnson, and David McClosky. Improving unsupervised dependency parsing with richer contexts and smoothing. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL '09, pages 101–109, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[5] Dan Klein. *The Unsupervised Learning of Natural Language Structure*. PhD thesis, Stanford University, 2005.

[6] Dan Klein and Christopher D. Manning. Natural language grammar induction using a Constituent-Context model. In Thomas G. Dietterich, Suzanna Becker, and Zoubin Ghahramani, editors, *Advances in Neural Information Processing Systems (NIPS) 14*, pages 35–42. MIT Press, 2001.

[7] Dan Klein and Christopher D. Manning. A generative Constituent-Context model for improved grammar induction. In *ACL*, pages 128–135, 2002.

[8] Dan Klein and Christopher D. Manning. Corpus-Based induction of syntactic structure: Models of dependency and constituency. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, pages 478–485, Morristown, NJ, USA, 2004. Association for Computational Linguistics.

[9] Christopher D. Manning and Hinrich Schtze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1 edition, June 1999.

[10] Mitchell P. Marcus, Beatrice Santorini, and Mary A. Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1994.

[11] Valentin I. Spitkovsky, Hiyan Alshawi, and Daniel Jurafsky. Baby Steps: How "Less is More" in unsupervised dependency parsing. In *NIPS: Grammar Induction, Representation of Language and Language Learning*, 2009.

[12] Valentin I. Spitkovsky, Hiyan Alshawi, and Daniel Jurafsky. From baby steps to leapfrog: how "less is more"in unsupervised dependency parsing. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, pages 751–759, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[13] Hiroyasu Yamada and Yuji Matsumoto. Statistical dependency analysis with support vector machines. In *In Proceedings of IWPT*, pages 195–206, 2003.