PRÁCTICA 1: Entrenamiento y Evaluación de PoS Taggers y Parsers

David Pérez Román

April 13, 2024

Contents

1 Introducción

En esta practica se pide completar una serie de ejercicios pertenecientes al ámbito del Procesamiento del Lenguaje Natural, en concreto se pide familiarizarse con diferentes herramientas y recursos para entrenar PoS Taggers, Parsers de Constituyentes y Parsers de Dependencias.

Antes de dar comienzo a la resolución de los ejercicios propuestos, se presentará el entorno empleado para la realización de los mismos, puesto que ha sido el mismo para todos.

2 Entorno

El entorno empleado para la ejecución del código empleado para completar los ejercicios es un Notebook en *Google Colaboratory*. La razón por la que se ha escogido este entorno es porque, para algunos de los parsers escogidos, es necesario emplear procesamiento con GPU.

Google Colaboratory ofrece la opción de entorno con soporte de GPU, aunque con carácter limitado, lo que provocará que sea necesario adaptar los entrenamientos de los modelos para cumplir con estas limitaciones (sobretodo en cuanto al tiempo disponible para entrenar los modelos).

El análisis de los resultados se hará acorde con estas limitaciones, siempre buscando obtener resultados razonables con los recursos disponibles.

Finalmente, el Notebook con todo el código fuente de la practica podrá ser consultado siguiendo el enlace del mismo en el Apéndice A.

3 Entrenamiento y Etiquetación con PoS Taggers

Para este ejercicio se ha decidido emplear la librería *SpaCy V3* (Explosion, 2021b). SpaCy permite entrenar modelos empleando CPU (modelos basados en Tok2Vec), y también empleando GPU (modelos basados en Transformers, en concreto redes BERT).

Se decide emplear la modalidad de entrenamiento con CPU, de este modo se mitigan las limitaciones del entorno en cuanto al tiempo disponible de uso de los recursos.

3.1 Con modelos pre-entrenados

3.1.1 Inglés

Como se ha expuesto al inicio, se ha decidido emplear SpaCy para completar este ejercicio, para el idioma Inglés se ha decidido emplear el modelo pre-entrenado *en_core_web_sm*. Este modelo esta optimizado para su uso empleando CPU, entrenado empleando textos comunes en la web (blogs, noticias, comentarios).

Dada la índole de los textos empleados en el entrenamiento del modelo, se empleará un corpus acorde para que el modelo lo etiquete, minimizando los posibles errores durante el etiquetado, dicho corpus es el *Blog Authorship Corpus* (Schler et al., 2006). Este corpus contiene publicaciones de 19.320 blogueros recopiladas en blogger.com en agosto de 2004. El corpus incorpora un total de 681.288 publicaciones y más de 140 millones de palabras, o aproximadamente 35 publicaciones y 7250 palabras por persona.

Desgraciadamente, es difícil descargar este corpus de la fuente original (la pagina web es antigua y, al intentar descargar el archivo, se produce un error) por lo que se ha buscado una fuente alternativa para este corpus.

El corpus se ha descargado de *Kaggle* (Tatman, 2017), al descargarlo de una fuente alternativa, el formato no es el mismo que el original. En la versión de Kaggle, todos los archivos se han agrupado en un único CSV, esto facilita el preprocesado del dataset.

3.1.1.1 Preprocesado del dataset Se implementa un preprocesado sencillo del dataset.

El CSV original contiene 6 columnas:

- id
- genero (masculino o femenino)
- edad
- tópico
- signo del zodiaco
- fecha
- texto

De estas columnas, unicamente es necesario leer la sexta columna, que es la que corresponde con el texto escrito por el usuario.

Una vez leído el CSV, se limpia el Dataframe (se eliminan filas vacías, se impone el tipo de dato de la columna y se eliminan espacios al principio y al final de cada texto)

Adicionalmente, se añade una columna adicional al dataset, esta columna contiene el numero de palabras de cada texto.

Finalmente, se genera un nuevo dataset con las filas del anterior dataset reordenadas de forma aleatoria. Con este nuevo dataset, se genera el archivo *INPUT_RAW.txt* para que este contenga a aproximadamente 10.000 palabras.

3.1.1.2 Etiquetado El etiquetado con SpaCy es bastante sencillo, únicamente es necesario establecer el modelo (en este caso uno pre-entrenado), alimentarlo con un texto a procesar y, finalmente, escribir un archivo de salida con el texto y la etiqueta.

3.1.2 Castellano

En el caso del Castellano, se sigue el mismo proceso que con el Inglés. En este caso se emplea el modelo *es_core_news_sm*, también optimizado para su ejecución con CPU, pero esta vez entrenado empleando noticias. Por este motivo se va a emplear un dataset que recoja un corpus compuesto por noticias en Español. El dataset escogido se trata de *Spanish News Classification* (Morgado, 2023), este dataset fue construido con una herramienta de web scraping para el Dataton 2022 de Bancolombia para entrenar modelos supervisados para usar en una recomendación de noticias.

3.1.2.1 Preprocesado del dataset Del mismo modo que con el idioma anterior, se implementa un preprocesado sencillo del dataset.

En este caso el CSV unicamente contiene 3 columnas:

- url de la que proviene la noticia
- texto
- categoría de la noticia

El preprocesado del dataset es el mismo que el que se ha seguido para el idioma Inglés: Se limpia el dataset, se añade una columna con el número de palabras

por fila, se reordena el dataset de forma aleatoria y se construye el archivo de *INPUT_RAW.txt*.

3.1.2.2 Etiquetado De nuevo, el proceso es sencillo y vuelve a ser el mismo que con el Inglés. Se establece un modelo, se alimenta con un texto y se extraen el texto y las etiquetas.

3.1.3 Resultados

Los resultados obtenidos para ambos idiomas son satisfactorios, como cabría esperarse de modelos pre-entrenados y optimizados. Adicionalmente, el uso de datasets que coincidan con el tipo de texto empleado durante el entrenamiento ha contribuido a la calidad de los resultados obtenidos.

En el fichero de salida se puede apreciar, en ambos idiomas, la presencia de líneas en blanco con la etiqueta *(SPACE)*, esto es debido a la presencia de saltos de línea en el texto de entrada *(\n)*.

3.2 Entrenando los modelos

En este apartado del ejercicio se ha empleado de nuevo SpaCy, de hecho, el proceso ha sido el mismo para los 3 idiomas para los que se ha entrenado un etiquetador, por lo que antes de describir características concretas de cada idioma, se van a definir los procedimientos de preprocesado de los datos y entrenamiento de los modelos, puesto que son los comunes para los idiomas entrenados (excepto alguna característica especifica que se explicará en cada idioma).

3.2.1 Preprocesado de los datos

Para entrenar los modelos, se han empleado los treebanks de *Universal Dependencies* (UD) (Universal Dependencies Consortium, 2022), puesto que sus treebanks recogen también las etiquetas morfosintácticas de las palabras del texto. En concreto se ha empleado el compendio de UD v2.13.

En los directorios de UD para cada idioma se pueden encontrar diferentes archivos, comúnmente se encontrarán 3 archivos .conllu y 3 archivos .txt, que corresponden con las particiones Train, Dev y Test. De estos archivos se emplearán las particiones Train y Dev en formato conllu, y la partición Test en formato .txt (que será el archivo que actuará como *INPUT_RAW.txt* tras procesarlo).

SpaCy no trabaja con archivos en formato .conllu, sino que trabaja con archivos binarios formato .spacy, por lo que es necesario hacer una conversión, por suerte SpaCy tiene un comando para hacer precisamente eso: *spacy convert*. Empleando este comando en CLI se convierten todos los ficheros train.conllu y dev.conllu a train.spacy y dev.spacy.

Para los textos de test, se emplean los archivos test.txt de UD (siempre y cuando tengan una longitud apropiada). Como no es necesario hacer un preprocesado extenso, se ajustarán cuando se lean para hacer los test de los modelos.

3.2.2 Entrenamiento en SpaCy

Para entrenar modelos con SpaCy es necesario generar un archivo de configuración que contendrá la información del pipeline del modelo. Como únicamente es necesario etiquetar los textos, se puede incluir únicamente el componente *tagger* en el archivo de configuración (Además del componente *Tok2Vec*, que es la base del modelo). El proceso de generación de archivos de configuración consta de dos partes, la primera es la generación del un archivo preliminar, para este paso SpaCy ofrece una plantilla configurable en su web (Explosion, 2021a). La segunda, consiste en emplear un comando de SpaCy en CLI para terminar de completar el archivo de configuración: *spacy init fill-config*, este comando es útil cuando se quiere emplear la configuración por defecto.

Una vez obtenido el archivo de configuración, ya se puede dar paso al entrenamiento de los modelos. Para ello, únicamente es necesario llamar al comando *spacy train* en CLI, especificando diferentes rutas a archivos necesarios: config.cfg (creado en el paso anterior), output del modelo (donde se guardará el modelo entrenado), train.spacy y dev.spacy.

3.2.3 Testeo de los modelos

Para obtener el *OUTPUT_RAW.txt* que solicita el ejercicio, se seguirá un método similar al del apartado 3.1, con algunos cambios para poder leer correctamente los textos.

El texto de entrada se limpiará con una *Regular Expression* con la que se limpiarán los espacios en blanco, en concreto únicamente los dobles espacios, presentes en algunos puntos de los corpus de UD.

3.2.4 Japonés

Para el idioma Japonés se ha empleado el conjunto de UD *JA-GSD*. Adicionalmente, es necesario instalar dependencias adicionales para poder entrenar los modelos, estas dependencias son *sudachipy* y *sudachidict_core* que son un analizador morfosintáctico y un conjunto de diccionarios respectivamente, del japonés. No es necesario utilizar estas dependencias directamente, basta con tenerlas instaladas para que SpaCy acceda a ellas.

Una vez entrenado el modelo, es necesario añadir un paso adicional al testeo, la razón es porque SpaCy tiene una limitación en algunos idiomas (el japonés siendo uno de ellos) en los que el tamaño máximo del texto que se le pasa al modelo no se ajusta al de la documentación de la librería, por lo que hay que separarlo y pasárselo al etiquetador en varias partes. Este problema nace de una de las dependencias de SpaCy, no de la propia librería. En el caso del Japones, el problema se localiza en la librería *sudachipy*. Para poder cortar de forma optima el texto, se cuentan los caracteres del texto tras codificarlos en UTF-8 y se divide la longitud total del texto por la recién obtenida. Para conocer el tamaño máximo permitido solo es necesario observar el Traceback del error y anotar el valor máximo esperado, que en el caso del japones es 49149 Bytes.

Para resolver este error se emplea una función obtenida del repositorio de SpaCy (JWittmeyer, 2023).

Con este problema resuelto, se puede seguir el proceso convencional ya descrito anteriormente.

3.2.5 Ruso

Para el idioma Ruso se ha empleado el conjunto de UD *RU-TAIGA* y se ha entrenado y testeado siguiendo el proceso descrito anteriormente, sin necesidad de añadir pasos adicionales.

3.2.6 Chino

Para el idioma Chino se ha empleado el conjunto de UD *ZH-GSD* y se ha entrenado y testeado siguiendo el proceso descrito anteriormente, sin necesidad de añadir pasos adicionales.

3.2.7 Resultados

A diferencia del apartado 3.1, en este apartado podemos consultar las métricas que SpaCy ofrece durante el entrenamiento, que aunque básicas, sirven para poder juzgar el desempeño del modelo. Las métricas obtenidas para cada idioma se pueden consultar en la Tabla 1.

	Epochs	#	LOSS TOK2VEC	LOSS TAGGER	TAG_ACC	SCORE
Japonés	2	1600	315.59	2131.50	76.98	0.77
Ruso	12	4000	141.45	1646.80	89.71	0.90
Chino	22	9000	541.54	2456.54	88.50	0.88

Table 1: Métricas durante entrenamiento

Como se puede ver en la tabla, el Japonés se ha entrenado durante menos epochs que los otros dos idiomas. La razón por la que se ha entrenado menos es porque ha llegado al limite de steps sin mejora. En el archivo de configuración, hay un parámetro que indica la paciencia antes de terminar el entrenamiento debido a la no mejora de la métrica, este parámetro es 1600 steps para los tres idiomas. Como para el Japonés no se observa una mejora en el *SCORE*, al llegar al límite de steps, como se ve en la columna *#*, se interrumpe el entrenamiento.

Por otro lado, tanto para el Ruso como para el Chino se ha alcanzado un SCORE satisfactorio considerando el tiempo y recursos empleados para el entrenamiento, siendo el primero de no más de 10 minutos por modelo, y el segundo siendo un entorno con soporte de CPU (no GPU o TPU).

Finalmente, como se puede apreciar en la Tabla 1, en el caso del Chino se ha obtenido un resultado similar al del Ruso, pese a que se ha entrenado durante casi el doble de Epochs.

4 Entrenamiento y evaluación de parsers de constituyentes

Para este ejercicio se ha decidido emplear la librería *Stanza* (Qi et al., 2020e), puesto que ofrece una implementación para poder entrenar parsers de constituyentes. Asimismo, el parser de Stanza tiene integrada la herramienta de evaluación *EVALB*, por lo que podrán evaluarse los modelos sin necesidad de recurrir a otros métodos.

El parser de constituyentes de Stanza esta basado en una arquitectura *Shift-Reduce*, el parser incorpora diferentes modelos para funcionar correctamente, como una red *BI-LSTM* como word input, otra red *BI-LSTM* para realizar la combinación de varios constituyentes en uno solo, etc. Mas información sobre la estructura el parser se puede obtener del código fuente de la librería (Qi et al., 2020c).

Adicionalmente, Stanza emplea *Character-Aware Neural Language Models* o charlm (Kim et al., 2015) para capturar información morfológica y de subpalabras directamente del texto de entrada mediante el procesamiento de caracteres individuales o sub-secuencias de caracteres. El parser de constituyentes de Stanza también ofrece la posibilidad de añadir arquitecturas transformer, modelos BERT específicamente, externas para mejorar los parsers entrenados.

Stanza es una librería para la que es necesario disponer de soporte de GPU para entrenar los modelos en tiempos razonables. Como se expuso el inicio de la practica, este soporte tiene carácter limitado en el entorno de trabajo empleado, y esto condicionará los entrenamientos de los modelos. Tras diversas pruebas, se ha detectado que es posible emplear el entorno con soporte de GPU de *Google Colaboratory* durante periodos de una hora sin que se interrumpa la ejecución por exceso de consumo. Por lo tanto, se ajustará el numero de epochs para que los entrenamientos de los modelos no duren mas de 1 hora, y en algunos casos se interrumpirá el entrenamiento debido a estas limitaciones.

El proceso de preparación del treebank y entrenamiento es común para todos los idiomas, siendo diferente únicamente los componentes empleados para la preparación de los treebanks (que en algunos casos es necesario hacerlos desde 0), dichos componentes se explicarán en cada idioma. Por lo tanto, se detallará el proceso general a continuación.

4.1 Entrenamiento de parser de constituyentes en Stanza

Para entrenar los modelos, se ha seguido la guía que ofrece Stanza en su pagina web (Qi et al., 2020a), en la que se detalla tanto el formato del Treebank que espera Stanza, que se trata del formato PennTreeBank (PTB); así como el proceso para generar los conjuntos de entrenamiento y gold standards y, finalmente, como abordar el proceso de entrenamiento de los modelos.

Todo el proceso se realiza mediante comando en CLI, por lo que es necesario modificar las variables de entorno para añadir directorios de interés, en el caso de *Google Colaboratory* se emplea un archivo de configuración con las variables y se ejecutará en cada ejecución. Este archivo de configuración se puede consultar

en el Anexo B.1

Primero, se debe preparar el treebank, para esto se emplea un script de Stanza *prepare_con_dataset*, este script tiene una selección de treebanks de diferentes idiomas ya implementada, pero para treebanks o idiomas que no estén presentes será necesario modificar el script para añadirlos, de este modo se puede seguir empleando el script y mantener el mismo proceso para todos los casos.

Una vez preparado el treebank se puede proceder al entrenamiento del modelo empleando el script *run_constituency*, aquí se especificará el número de Epochs, que es de 20 epochs para mantener la duración del entrenamiento en aproximadamente 1 hora.

Una vez entrenado el modelo, se puede realizar una evaluación sobre los conjuntos de dev y test llamando al mismo script *run_constituency*, pero esta vez con el argumento *--score_dev* o *--score_test*, donde se Stanza emplea el script de evaluación EVALB y devuelve las métricas del mismo.

4.2 Inglés + lengua romance

4.2.1 Inglés

Para el idioma Inglés, se ha empleado el treebank *The Manually Annotated Sub-Corpus (MASC)* (American National Corpus Consortium, 2015), en concreto se ha descargado la versión *Penn Treebank constituency annotation of entire MASC in original PTB bracket format*.

Este treebank tiene formato Penn Treebank, pero aun así es ligeramente distinto al formato que espera Stanza. La diferencia principal es que Stanza espera que las oraciones estén en una única línea, mientras que el formato de este treebank tiene las oraciones con el formato siguiente:

```
( (S (NP-SBJ (NP (PRP$ Your) (NN contribution))
             (PP (TO to)
                 (NP (NNP Goodwill))))
     (VP (MD will)
         (VP (VB mean)
             (NP (NP (JJR more))
                 (SBAR (IN than)
                       (S (NP-SBJ (PRP you))
                          (VP (MD may)
                              (VP (VB know)))))))))
     (. .)))
```

Adicionalmente, el treebank esta dividido en varios archivos con extensión .mrg, por lo que será necesario leerlos todos y ajustar el formato para obtener las particiones train, dev y test.

Para procesar el treebank se modificará el script de Stanza *prepare_con_dataset* y se generará un nuevo script para este treebank: *convert_en_masc.py*. En el primero, simplemente se añade una función al script que llame al segundo script, tomando como ejemplo el resto de idiomas ya implementados. En el caso del segundo script, se implementa una lectura de los archivos y se procesa el texto leído para transformarlo a una estructura de una única línea. Para que este script funcione, es necesario que el formato de los archivos leídos sea constante, lo cual se cumple en gran parte de los casos, pero en algunos archivos se detecta que el formato no es el mismo que en los demás, en estos casos se eliminarán los archivos y no se procesarán (Se ha tomado esta decisión tras detectar que los archivos con formato distinto eran una pequeña parte y contenían un número reducido de árboles). Los dos Scripts se pueden consultar en los Anexos B.2 y B.3, respectivamente.

Una vez implementados los Scripts, se puede proceder con el entrenamiento empleando el procedimiento descrito en el Apartado 4.1.

Tras procesar el treebank, se obtienen 5201 árboles, los cuales se dividen de modo que 4160 train, 520 dev y 521 test (particiones 80/20 train/test).

Durante el proceso de entrenamiento del treebank se detectan errores en 2 árboles (2 oraciones) en las que el proceso de validación de los árboles falla, como es únicamente en 2 de las oraciones en la partición de train, se decide eliminarlas. Estos arboles pueden ser consultados en el Anexo C.

4.2.2 Portugués

Para el idioma Portugués se ha empleado el *CINTIL-TreeBank* (PortulanCLARIN Consortium, 2012). Este treebank está compuesto por 10.039 frases y 110.166 tokens extraídos de diferentes fuentes y dominios: noticias (8.861 frases; 101.430 tokens), novelas (399 frases; 3.082 tokens).

En este caso, el script de Stanza ya está preparado para preprocesar este treebank, por lo que no es necesario añadir o modificar scripts, únicamente es necesario llamar al script de preparación del dataset especificando el nombre del treebank con el indicador que espera Stanza: *pt_cintil*. Tras procesar el treebank, se obtienen 9948 arboles, los cuales se dividen de modo que 8743 train, 995 dev y 995 test.

Una vez obtenidas las particiones del dataset se sigue el proceso de entrenamiento descrito en el Apartado 4.1.

4.2.3 Resultados

En este apartado se analizarán los resultados obtenidos para los parsers de constituyentes de los idiomas Inglés y Portugués, ambos obtenidos empleando la librería Stanza, en concreto, empleando el comando *run_constituency* con el argumento *--score_test*.

Idioma	LP	LR	F1	Exact	N	Entrenamiento
Inglés	84.84	85.27	85.05	33.58	521	90 min
Portugués	90.31	90.11	90.21	55.77	995	70 min

Table 2: Métricas EVALB en la partición Test para Inglés y Portugués

Como se puede observar en la Tabla 2, Stanza devuelve las métricas de EVALB para diferentes configuraciones, de entre ellas nos centraremos en *Probabilistic context-free grammar (PCFG)*, para la cual se han obtenido resultados prometedores. En el caso del Inglés, se observa que tanto el *Labeled Precision (LP)* como el *Labeled Recall (LR)* son razonables. Por el contrario, se observa como el porcentaje de árboles en los que tanto LP como LR son 100% es, de hecho, algo pobre (33.58%).

Por otro lado, en el caso del Portugués se observan resultados mejores, tanto LP como LR obtienen aproximadamente un 90%, y el número de árboles donde ambos son 100% llega hasta al 55%.

La diferencia entre los dos idiomas puede ser debida al tamaño del treebank, donde el Portugués tenía 4000 árboles más en total, consiguiendo así mejores resultados. Cabe destacar que, pese a que el Inglés se entrenase durante más tiempo, el número de Epochs en ambos casos es el mismo, como se ha expuesto al inicio del ejercicio.

4.3 Otros parsers y/o idiomas

Para este apartado se seguirá la misma estrategia que con el anterior, pero esta vez con dos idiomas nuevos, por lo que se aplicará la estrategia expuesta en el Apartado 4.1.

4.3.1 Japonés

El treebank empleado para el Japonés es el *Asian Language Treebank (ALT) Project* (Miyama, 2015), se trata de un Treebank anotado para aproximadamente

20000 traducciones japonesas de oraciones de artículos de Wikinews en inglés.

Para este treebank es necesario descargar 3 archivos adicionales: *URL-train.txt*, *URL-dev.txt* y *URL-test.txt*. Estos archivos se emplean como índices, en ellos viene indicado el ID de la oración y la página web de la que se obtuvo. Basta con dejar los archivos en el directorio del treebank para que el Script de Stanza los lea.

Tras procesar el treebank, se obtienen 19060 árboles, los cuales se dividen de modo que 17195 train, 934 dev y 931 test.

Por desgracia, debido a las limitaciones de *Google Colaboratory*, se ha interrumpido el entrenamiento del parser tras haber transcurrido únicamente 5 Epochs. A diferencia de los idiomas entrenados en el Apartado 4.2, en los que cada Epoch se entrenaba en aproximadamente 3 minutos, en el caso de Japonés cada Epoch tardaba 9.5 minutos en entrenarse, más de 3 veces más que los idiomas anteriores. Por este motivo, aunque se ha entrenado el modelo durante los 60 minutos que ha permitido el entorno, el número de Epochs ha sido menor.

4.3.2 Indonesio

El treebank empleado para el idioma Indonesio es el *ICON: A Large-Scale Benchmark Constituency Treebank for the Indonesian Language* (AI Singapore, 2023). Este Treebank ya cuenta con el formato PTB esperado por Stanza. Aun así, Stanza tiene implementado el procesado de este Treebank en su Script *prepare_con_dataset*, del mismo modo que el Japonés o el Portugués vistos anteriormente.

Tras procesar el treebank, se obtienen 10000 árboles, los cuales se dividen de modo que 8000 train, 1000 dev y 1000 test.

Por desgracia, debido a las limitaciones de *Google Colaboratory*, se ha interrumpido el entrenamiento del parser tras haber transcurrido únicamente 12 Epochs. Este idioma tarda los mismo que los del Apartado 4.2, pero se ha entrenado menos. La razón es que, del mismo modo que con el Japonés, se ha interrumpido el entrenamiento por un límite de consumo. Cabe destacar, que *Google Colaboratory* limita los recursos cuanto más se usen, es decir, que tras varias ejecuciones se recorta cada vez más el tiempo disponible para entrenar los modelos, provocando situaciones como esta.

4.3.3 Resultados

En este apartado se analizarán los resultados obtenidos para los parsers de constituyentes de los idiomas Japonés e Indonesio, ambos obtenidos empleando la

librería Stanza, en concreto, empleando el comando *run_constituency* con el argumento *--score_test*.

Idioma	LP	LR	F1	Exact	N	Entrenamiento
Japonés	87.29	85.03	86.15	17.61	931	60 min
Indonesio	82.26	83.87	83.05	29.8	1000	60 min

Table 3: Métricas EVALB en la partición Test para Japonés e Indonesio

Como se puede observar en la Tabla 3, pese a que los parsers entrenados obtienen metricas *Labeled Precision (LP)*, *Labeled Recall (LR)* y F1 razonables (entorno al 85%), se puede observar el efecto negativo de las interrupciones prematuras de los entrenamientos en la métrica *Exact*, donde se observan unos resultados muy pobres, especialmente en el caso del Japonés, donde el número de oraciones en las que tanto LP como LR son 100% no alcanza el 20%. Mientras que para el Indonesio, se queda a 0.2% de alcanzar el 30%.

5 Entrenamiento y evaluación de parsers de dependencias

Para completar este ejercicio se han empleado dos librerías: *SpaCy* (Explosion, 2021b) y *Stanza* (Qi et al., 2020e). SpaCy se entrenará empleando CPU, mientras que Stanza se entrenará empleando GPU.

Como el proceso de entrenamiento y posterior validación de los modelos es común para los idiomas entrenados, se describirá a continuación el proceso seguido para preparar los treebanks y entrenar los modelos en ambas librerías, dejando los apartados de cada idioma para información adicional o características específicas de los mismos.

5.1 Preprocesado de los datos

Para entrenar los modelos, se han empleado los treebanks de *Universal Dependencies* (UD) (Universal Dependencies Consortium, 2022). En concreto se ha empleado el compendio de UD v2.13.

En los directorios de UD para cada idioma se pueden encontrar diferentes archivos, comúnmente se encontrarán 3 archivos .conllu y 3 archivos .txt, que corresponden con las particiones Train, Dev y Test. Se emplearán las 3 particiones en

formato .conllu y la partición de Test también en formato .txt. El archivo Test.txt se empleará como el texto sobre el que hay que hacer el análisis de Dependencias, mientras que Test.conllu se empleará como Gold Standard para la evaluación de los modelos.

SpaCy no trabaja con archivos en formato .conllu, sino que trabaja con archivos binarios formato .spacy, por lo que es necesario hacer una conversión, por suerte SpaCy tiene un comando para hacer precisamente eso: *spacy convert*. Empleando este comando en CLI se convierten todos los ficheros train.conllu y dev.conllu a train.spacy y dev.spacy.

En el caso de Stanza, se pueden emplear los archivos en formato .conllu directamente.

5.2 Entrenamiento en SpaCy

Para entrenar modelos con SpaCy es necesario generar un archivo de configuración que contendrá la información del pipeline del modelo. Para poder entrenar el parser de dependencias es necesario añadir otros componentes al pipeline puesto que el parser depende de ellos, por lo tanto se incluirán los componentes: *tagger*, *morphologizer*, *trainable_lemmatizer* y *parser* en el archivo de configuración (Además del componente *Tok2Vec*, que es la base del modelo). El proceso de generación de archivos de configuración consta de dos partes, la primera es la generación del un archivo preliminar, para este paso SpaCy ofrece una plantilla configurable en su web (Explosion, 2021a). La segunda, consiste en emplear un comando de SpaCy en CLI para terminar de completar el archivo de configuración: *spacy init fill-config*, este comando es útil cuando se quiere emplear la configuración por defecto.

Una vez obtenido el archivo de configuración, ya se puede dar paso al entrenamiento de los modelos. Para ello, únicamente es necesario llamar al comando *spacy train* en CLI, especificando diferentes rutas a archivos necesarios: config.cfg (creado en el paso anterior), output del modelo (donde se guardará el modelo entrenado), train.spacy y dev.spacy.

SpaCy no cuenta con la opción de exportar el resultado del parser en formato .conllu. Para obtener el formato deseado se emplea la librería *spacy_conll* (Vanroy, 2021). Para emplearla solo es necesario incluir un elemento al final del pipeline de SpaCy, ajustando la configuración de modo que incluya los headers en el documento.

5.3 Entrenamiento en Stanza

Del mismo modo que SpaCy, si se quiere entrenar un parser de dependencias empleando Stanza, es necesario añadir mas componentes. Stanza descargará automáticamente los componentes que no encuentre de forma local, pero entrenar dichos componentes puede resultar beneficioso para mejorar los resultados obtenidos. Por lo tanto, se entrenarán: *Tokenizer*, *Tagger* y *Parser*.

Para entrenar los modelos se han seguido los pasos descritos en la pagina oficial de Stanza (Qi et al., 2020b) y en su repositorio para entrenamiento de modelos (Qi et al., 2020d).

Para entrenar los diferentes componentes se emplean 2 comandos por CLI. El primero, *prepare_component_treebank* prepara los treebanks para que Stanza pueda emplearlos, solo es necesario substituir *component* por el componente que se vaya a preparar, por ejemplo en el caso de la preparación del treebank para el parser de dependencias *prepare_depparse_treebank*.

Una vez preparados los treebanks, se puede emplear el comando *run_component* para iniciar el entrenamiento. Del mismo modo que con el anterior comando, se substituye *component* por el componente que se desee entrenar. Por ejemplo, una vez mas, con el parser de dependencias *run_depparse*.

A estos comandos se les pasa como argumento el identificador del idioma que se esté entrenando. Adicionalmente, al segundo comando se le pueden pasar argumentos para configurar los hiperparámetros del entrenamiento. Como el entorno es limitado, se limitan los Steps de entrenamiento para los modelos, resumidos en la Tabla 4.

	Tokenizador	Tagger	Parser
Steps	1000	500	1000

Table 4: Steps durante entrenamiento

Con estos Steps reducidos se asegura que *Google Colaboratory* no interrumpa el entrenamiento por exceso de tiempo o consumo de recursos. Estos hiperparámetros serán comunes para todos los idiomas entrenados.

5.4 Evaluación de los modelos

Para evaluar los modelos se ha empleado el Script de evauluación *CoNLL Shared Task 2018* (Universal Dependencies Consortium, 2018). El script tomará el out-

put de los parsers en formato .conllu y lo comparará con el Gold Standard proporcionado.

El script evalúa y muestra distintas métricas, que evalúan diferentes aspectos de la anotación. Las tres métricas principales son:

- **Labeled Attachment Score (LAS)**: El porcentaje de palabras a las que se les asigna tanto el encabezado sintáctico correcto como la etiqueta de dependencia correcta.
- **Morphology-Aware Labeled Attachment Score (MLAS)**: Variación de LAS para palabras de contenido donde, además de la relación principal y de dependencia, también se deben predecir correctamente la etiqueta POS universal, las características morfológicas seleccionadas y las dependencias funcionales particulares.
- **Bilexical dependency score (BLEX)**: Similar a MLAS en cuanto a que se centra en las relaciones entre palabras de contenido. En lugar de rasgos morfológicos, incorpora lematización en la evaluación.

Las tres métricas reflejan la segmentación de palabras y las relaciones entre las palabras del contenido. LAS también incluye relaciones entre otras palabras, pero ignora la morfología y los lemas. Las otras dos métricas están más cercanas al contenido y al significado; Las puntuaciones del MLAS también deberían ser más comparables entre idiomas tipológicamente diferentes.

EL script se puede emplear tanto por CLI, como en el propio pipeline del proyecto, para este caso se ha decidido emplear la segunda opción, para ello es necesario añadir el script al entorno de trabajo e importarlo para poder emplearlo.

Para emplear el script se pueden ejecutar las funciones del mismo, pero para mostrar las métricas por pantalla no hay una función, únicamente se puede acceder mediante el pipeline principal del script, por este motivo se decide extraer el proceso y, empleando un wrapper, utilizarlo como una función por separado. La función generada se puede consultar en el Anexo B.4.

5.5 Inglés + lengua romance

5.5.1 Inglés

Para el idioma Inglés se ha empleado el conjunto de UD *EN-EWT*. El proceso de preparación de los datos y de entrenamiento es el descrito al inicio del ejercicio.

Durante el testeo del parser entrenado empleando Stanza, se observa un error cuando el script de evaluación valida los .conllu. El script comprueba que la secuencia de Tokens del archivo predicho y el gold standard coincidan, si la secuencia de tokens no coincide, el proceso falla. En el caso de SpaCy no aparece este problema, pero con Stanza se observa la presencia del token <*UNK*> en el .conllu. Este token es empleado cuando el parser no reconoce el token leído y lo identifica como desconocido (*Unknown*). Como únicamente se detecta la presencia de dos de estos tokens, el primero teniendo como token original un hipervínculo a una página web, y el segundo teniendo como token original la combinacion de los caracteres *'m* en la secuencia: *Hi I'm from Brazil and I want to know of book 06*. Se decide modificar el output de Stanza, substituyendo los dos tokens <*UNK*> por los originales en el texto. Tras este cambio, el script de evaluación se ejecuta sin problemas.

5.5.2 Italiano

Para el idioma Italiano se ha empleado el conjunto de UD *IT-ISDT*. El proceso de preparación de los datos y de entrenamiento es el descrito al inicio del ejercicio.

En este idioma, a diferencia del anterior, no se ha detectado ningún error en el proceso de evaluación de ninguno de los parsers.

5.5.3 Resultados

Los resultados obtenidos para ambos idiomas en ambos parser se han incluido en forma de tablas. Como se ha comentado en el Apartado 5.4, las métricas principales son LAS, MLAS y BLEX. Por lo tanto, se han incluido únicamente estas en las tablas de resultados. Las tablas completas pueden consultarse en el Anexo D.

	Precision	Recall	F1 Score	Aligned Acc
LAS	69.77	70.93	70.34	72.22
MLAS	64.50	56.65	60.32	57.77
BLEX	68.46	60.13	64.03	61.32

Table 5: Métricas principales en idioma Inglés empleando SpaCy

	Precision	Recall	F1 Score	Aligned Acc
LAS	83.57	83.48	83.53	84.36
MLAS	78.46	78.10	78.28	79.00
BLEX	76.75	76.39	76.57	77.27

Table 6: Métricas principales en idioma Inglés empleando Stanza

Primero, en el idioma Inglés se puede ver como los resultados en Stanza (Tabla 6) son mejores que los obtenidos en SpaCy (Tabla 5), esto es debido a que SpaCy se entrena con CPU, por lo que para poder ser entrenado en un tiempo razonable, emplea un modelo más sencillo y que obtiene resultados más modestos en comparación con Stanza, que necesita soporte de GPU y emplea varios modelos para mejorar el pipeline, como por ejemplo CHARLMs. En general, Stanza obtiene un mínimo de 10% más en todas las métricas, donde en alguna obtiene resultados mucho mejores, como en el Aligned Accuracy y en el Recall para la métrica MLAS, donde Stanza consigue mejorar más de un 20% el resultado de SpaCy.

	Precision	Recall	F1 Score	Aligned Acc
LAS	73.00	67.79	70.30	79.09
MLAS	62.92	55.99	59.25	57.24
BLEX	75.57	67.25	71.17	68.75

Table 7: Métricas principales en idioma Italiano empleando SpaCy

	Precision	Recall	F1 Score	Aligned Acc
LAS	87.59	87.43	87.51	88.06
MLAS	81.55	81.14	81.35	82.02
BLEX	80.22	79.82	80.02	80.68

Table 8: Métricas principales en idioma Italiano empleando Stanza

En el caso del Italiano, se observa la misma situación que con el idioma Inglés, donde Stanza (Tabla 8) obtiene mejores resultados que SpaCy (Tabla 7). Stanza muestra resultados mejores especialmente en la métrica MLAS, del mismo modo que con el Inglés, esta métrica es casi 25% mejor en todas las categorías, indicando que Stanza es mejor en cuanto a la anotación completa del token (POS, dependencia, etc.).

5.6 Otros parsers y/o idiomas

En este apartado se han entrenado los parsers para tres idiomas más: Griego (SpaCy + Stanza), Chino (SpaCy) y Ruso (Stanza). El proceso seguido ha sido el mismo que en el apartado anterior, únicamente cambiando el idioma entrenado, por lo que los pasos descritos en los apartados 5.1, 5.2, 5.3 y 5.4, siguen siendo relevantes para el siguiente apartado.

5.6.1 Griego

Se ha decidido entrenar ambos parsers para el idioma Griego, de este modo se puede realizar una comparación más entre ellos en una lengua con alfabeto diferente al Latín.

Para el idioma Griego se ha empleado el conjunto de UD *EL-GDT*. El proceso de preparación de los datos y de entrenamiento es el descrito al inicio del ejercicio.

En este idioma, no se ha detectado ningún error en el proceso de evaluación de ninguno de los parsers.

5.6.2 Chino

El Chino únicamente se ha entrenado empleando SpaCy para probar como responde el modelo cuando se entrena con un idioma complejo como lo es el Chino y comprobar si, pese a que los resultados sean modestos, el modelo presenta una calidad razonable que justifique el uso de SpaCy por encima de Stanza más allá de las limitaciones de recursos de Stanza.

Para el idioma Chino se ha empleado el conjunto de UD *ZH-GSD*. El proceso de preparación de los datos y de entrenamiento es el descrito al inicio del ejercicio.

En este idioma, no se ha detectado ningún error en el proceso de evaluación.

5.6.3 Ruso

Para el idioma Ruso se ha empleado el conjunto de UD *RU-GSD*. El proceso de preparación de los datos y de entrenamiento es el descrito al inicio del ejercicio.

En este idioma, no se ha detectado ningún error en el proceso de evaluación.

5.6.4 Resultados

Del mismo modo que en el Apartado 5.5.3, únicamente se mostrarán los resultados de las métricas LAS, MLAS y BLEX en este apartado, pero las tablas completas

se pueden consultar en el Anexo D.

	Precision	Recall	F1 Score	Aligned Acc
LAS	75.68	73.97	74.81	77.72
MLAS	60.03	55.03	57.42	55.12
BLEX	63.36	58.09	60.61	58.18

Table 9: Métricas principales en idioma Griego empleando SpaCy

	Precision	Recall	F1 Score	Aligned Acc
LAS	86.38	86.55	86.47	86.79
MLAS	79.68	79.64	79.66	79.98
BLEX	76.09	76.05	76.07	76.38

Table 10: Métricas principales en idioma Griego empleando Stanza

Del mismo modo que en el Apartado 5.5.3, Stanza (Tabla 10) vuelve a obtener mejores resultados que SpaCy (Tabla 9) en todas las métricas, obteniendo diferencias de hasta un 24% en algunos casos, como el Aligned Acc MLAS.

	Precision	Recall	F1 Score	Aligned Acc
LAS	6.75	10.79	8.30	21.05
MLAS	2.58	4.68	3.32	16.51
BLEX	3.15	5.72	4.06	20.18

Table 11: Métricas principales en idioma Chino empleando SpaCy

En cuanto al Chino, los resultados obtenidos son bastante pobres (Tabla 11), donde apenas de obtiene un 20% en cada métrica en el mejor de los casos. Estos resultados no son una sorpresa, como se ha expuesto anteriormente se esperaba que los resultados fuesen pobres, el objetivo era comprobar si con un idioma complejo, como lo es el Chino, los resultados obtenidos serían razonables (que no buenos), para comprobar si la ventaja en cuanto al tiempo de entrenamiento y los recursos empleados para el mismo justificaba el uso de SpaCy por encima de Stanza. Por desgracia, los resultados no cumplen con las expectativas, corroborando que el uso de GPU para entrenar modelos más sofisticados durante más tiempo, es preferible a otras alternativas en cuanto a Tiempo y recursos en el entrenamiento / Calidad de los resultados obtenidos.

	Precision	Recall	F1 Score	Aligned Acc
LAS	83.92	83.96	83.94	84.38
MLAS	80.51	80.58	80.55	81.01
BLEX	75.37	75.43	75.40	75.84

Table 12: Métricas principales en idioma Ruso empleando Stanza

Finalmente, el Ruso obtiene resultados (Tabla 12) que están alineados con el resto de parsers para los demás idiomas entrenados en Stanza.

References

AI Singapore. (2023). ICON: A Large-Scale Benchmark Constituency Treebank for the Indonesian Language [Accessed: 2024-03-18]. https://github.com/aisingapore/seacorenlp-data

American National Corpus Consortium. (2015). The manually annotated subcorpus (masc) [Accessed: 2024-03-10]. https://anc.org/data/masc/

Explosion. (2021a). SpaCy training [Accessed: 2024-03-04]. https://spacy.io/usage/training#quickstart

Explosion. (2021b). *SpaCy: Industrial-strength natural language processing in Python* (Version 3.7.4). https://spacy.io

JWittmeyer. (2023). Issue 13207: Max_length of nlp pipline for e.g. japanese [Accessed: 2024-03-06]. https://github.com/explosion/spaCy/issues/13207

Kim, Y., Jernite, Y., Sontag, D., & Rush, A. M. (2015). Character-aware neural language models.

Miyama, M. (2015). Asian language treebank (alt) project [Accessed: 2024-03-15]. https://www2.nict.go.jp/astrec-att/member/mutiyama/ALT/

Morgado, K. (2023). Spanish news classification dataset [Accessed: 2024-03-04]. https://www.kaggle.com/datasets/kevinmorgado/spanish-news-classification

PortulanCLARIN Consortium. (2012). Cintil treebank [Accessed: 2024-03-12]. https://hdl.handle.net/21.11129/0000-000B-D2FE-A

Qi, P., Zhang, Y., Zhang, Y., Bolton, J., & Manning, C. D. (2020a). Stanza documentation: New language: Constituency parsing [Accessed: 2024-03-08]. https://stanfordnlp.github.io/stanza/new_language_constituency.html

Qi, P., Zhang, Y., Zhang, Y., Bolton, J., & Manning, C. D. (2020b). Stanza Documentation: Training and Evaluation [Accessed: 2024-03-12]. https://stanfordnlp.github.io/stanza/training_and_evaluation.html

Qi, P., Zhang, Y., Zhang, Y., Bolton, J., & Manning, C. D. (2020c). Stanza source code: Constituency parser [Accessed: 2024-03-08]. https://github.com/stanfordnlp/stanza/blob/main/stanza/models/constituency_parser.py

Qi, P., Zhang, Y., Zhang, Y., Bolton, J., & Manning, C. D. (2020d). Stanza Train Repository [Accessed: 2024-03-12]. https://github.com/stanfordnlp/stanza-train

Qi, P., Zhang, Y., Zhang, Y., Bolton, J., & Manning, C. D. (2020e). Stanza: A Python natural language processing toolkit for many human languages. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*.

Schler, J., Koppel, M., Argamon, S., & Pennebaker, J. W. (2006). Blog authorship corpus [Accessed: 2024-03-03]. https://u.cs.biu.ac.il/~koppel/BlogCorpus.htm

Tatman, R. (2017). Blog authorship corpus [Accessed: 2024-03-03]. https://www.kaggle.com/datasets/rtatman/blog-authorship-corpus

Universal Dependencies Consortium. (2018). CoNLL 2018 Shared Task Evaluation [Accessed: 2024-03-12]. http://universaldependencies.org/conll18/evaluation.html

Universal Dependencies Consortium. (2022). Universal dependencies [Accessed: 2024-03-04]. https://universaldependencies.org/

Vanroy, B. (2021). SpaCy conll module [Accessed: 2024-03-12]. https://github.com/BramVanroy/spacy_conll

A Enlaces

En este apéndice se pueden encontrar los enlaces a diferentes recursos empleados en la práctica, por conveniencia se incluye un hipervínculo embebido y el enlace (para que este pueda ser copiado, en caso de que el hipervínculo no funcione)

A.1 Notebook en Google Colaboratory

Enlace al notebook alojado en Google Colaboratory. Este notebook contiene el código fuente de la practica:

Notebook
https://colab.research.google.com/drive/1rthASBWhaL68sKUrzXsgksMB-3opKlWo

B Scripts Adicionales

En este apéndice se pueden encontrar scripts adicionales creados (o modificados) que no podían ser añadidos al Notebook con el resto del código.

B.1 Config.sh para parser de constituyentes

A continuación se presenta el script de configuración de variables de entorno empleado durante el Ejercicio de parsers de constituyentes.

```
#!/bin/bash

# Set the Constituency paths
export CONSTITUENCY_BASE=/content/gdrive/MyDrive/PLN/2/
    constituency-parser
export CONSTITUENCY_DATA_DIR=/content/gdrive/MyDrive/PLN/2/
    constituency
export CORENLP_HOME=/content/gdrive/MyDrive/PLN/2/CoreNLP
```

Listing 1: config.sh

B.2 prepare_con_dataset.py para parser de constituyentes

A continuación se presenta el script prepare_con_dataset.py modificado para añadir idiomas y treebanks adicionales en el Ejercicio de parsers de constituyentes. Se ha eliminado la cabecera del archivo puesto que esta no se ha modificado.

```
import argparse
import os
import random
import sys
import tempfile

from tqdm import tqdm

from stanza.models.constituency import parse_tree
import stanza.utils.default_paths as default_paths
from stanza.models.constituency import tree_reader
from stanza.models.constituency.parse_tree import Tree
from stanza.server import tsurgeon
from stanza.utils.datasets.constituency import utils
from stanza.utils.datasets.constituency.convert_alt import
    convert_alt
from stanza.utils.datasets.constituency.convert_arboretum import
     convert_tiger_treebank
from stanza.utils.datasets.constituency.convert_cintil import
    convert_cintil_treebank
import stanza.utils.datasets.constituency.convert_ctb as
    convert_ctb
from stanza.utils.datasets.constituency.convert_it_turin import
    convert_it_turin
from stanza.utils.datasets.constituency.convert_it_vit import
    convert_it_vit
from stanza.utils.datasets.constituency.convert_en_masc import
    convert_en_masc
from stanza.utils.datasets.constituency.convert_starlang import
    read_starlang
from stanza.utils.datasets.constituency.utils import SHARDS,
    write_dataset
import stanza.utils.datasets.constituency.vtb_convert as
    vtb_convert
import stanza.utils.datasets.constituency.vtb_split as vtb_split

class UnknownDatasetError(ValueError):
    def __init__(self, dataset, text):
        super().__init__(text)
        self.dataset = dataset

def process_it_turin(paths, dataset_name, *args):
    """
    Convert the it_turin dataset
    """
```

```
    assert dataset_name == 'it_turin'
    input_dir = os.path.join(paths["CONSTITUENCY_BASE"], "italian")
    output_dir = paths["CONSTITUENCY_DATA_DIR"]
    convert_it_turin(input_dir, output_dir)

def process_it_vit(paths, dataset_name, *args):
    # needs at least UD 2.11 or this will not work
    # in the meantime, the git version of VIT will suffice
    assert dataset_name == 'it_vit'
    convert_it_vit(paths, dataset_name)

def process_vlsp09(paths, dataset_name, *args):
    """
    Processes the VLSP 2009 dataset, discarding or fixing trees when needed
    """
    assert dataset_name == 'vi_vlsp09'
    vlsp_path = os.path.join(paths["CONSTITUENCY_BASE"], "vietnamese", "VietTreebank_VLSP_SP73", "Kho ngu lieu 10000 cay cu phap")
    with tempfile.TemporaryDirectory() as tmp_output_path:
        vtb_convert.convert_dir(vlsp_path, tmp_output_path)
        vtb_split.split_files(tmp_output_path, paths["CONSTITUENCY_DATA_DIR"], dataset_name)

def process_vlsp21(paths, dataset_name, *args):
    """
    Processes the VLSP 2021 dataset, which is just a single file
    """
    assert dataset_name == 'vi_vlsp21'
    vlsp_file = os.path.join(paths["CONSTITUENCY_BASE"], "vietnamese", "VLSP_2021", "VTB_VLSP21_tree.txt")
    if not os.path.exists(vlsp_file):
        raise FileNotFoundError("Could not find the 2021 dataset in the expected location of {} - CONSTITUENCY_BASE == {}".format(vlsp_file, paths["CONSTITUENCY_BASE"]))
    with tempfile.TemporaryDirectory() as tmp_output_path:
        vtb_convert.convert_files([vlsp_file], tmp_output_path)
        # This produces a 0 length test set, just as a placeholder until the actual test set is released
        vtb_split.split_files(tmp_output_path, paths["CONSTITUENCY_DATA_DIR"], dataset_name, train_size=0.9, dev_size=0.1)
```

```
69      _, _, test_file = vtb_split.create_paths(paths["
        CONSTITUENCY_DATA_DIR"], dataset_name)
70      with open(test_file, "w"):
71          # create an empty test file - currently we don't have
        actual test data for VLSP 21
72          pass
73
74  def process_vlsp22(paths, dataset_name, *args):
75      """
76      Processes the VLSP 2022 dataset, which is four separate
        files for some reason
77      """
78      assert dataset_name == 'vi_vlsp22' or dataset_name == '
        vi_vlsp23'
79
80      if dataset_name == 'vi_vlsp22':
81          default_subdir = 'VLSP_2022'
82          default_make_test_split = False
83          updated_tagset = False
84      elif dataset_name == 'vi_vlsp23':
85          default_subdir = os.path.join('VLSP_2023', '
        Trainingdataset')
86          default_make_test_split = True
87          updated_tagset = True
88
89      parser = argparse.ArgumentParser()
90      parser.add_argument('--subdir', default=default_subdir, type
        =str, help='Where to find the data - allows for using
        previous versions, if needed')
91      parser.add_argument('--no_convert_brackets', default=True,
        action='store_false', dest='convert_brackets', help="Don't
        convert the VLSP parens RKBT & LKBT to PTB parens")
92      parser.add_argument('--n_splits', default=None, type=int,
        help='Split the data into this many pieces.  Relevant as
        there is no set training/dev split, so this allows for N
        models on N different dev sets')
93      parser.add_argument('--test_split', default=
        default_make_test_split, action='store_true', help='Split
        1/10th of the data as a test split as well.  Useful for
        experimental results.  Less relevant since there is now an
        official test set')
94      parser.add_argument('--no_test_split', dest='test_split',
        action='store_false', help='Split 1/10th of the data as a
        test split as well.  Useful for experimental results.  Less
        relevant since there is now an official test set')
```

```
95      parser.add_argument('--seed', default=1234, type=int, help='
        Random seed to use when splitting')
96      args = parser.parse_args(args=list(*args))
97
98      if os.path.exists(args.subdir):
99          vlsp_dir = args.subdir
100     else:
101         vlsp_dir = os.path.join(paths["CONSTITUENCY_BASE"], "
        vietnamese", args.subdir)
102     if not os.path.exists(vlsp_dir):
103         raise FileNotFoundError("Could not find the {} dataset
        in the expected location of {} - CONSTITUENCY_BASE == {}".
        format(dataset_name, vlsp_dir, paths["CONSTITUENCY_BASE"]))
104     vlsp_files = os.listdir(vlsp_dir)
105     vlsp_test_files = [os.path.join(vlsp_dir, x) for x in
        vlsp_files if x.startswith("private") and not x.endswith(".
        zip")]
106     vlsp_train_files = [os.path.join(vlsp_dir, x) for x in
        vlsp_files if x.startswith("file") and not x.endswith(".zip")
        ]
107     vlsp_train_files.sort()
108     if len(vlsp_train_files) == 0:
109         raise FileNotFoundError("No train files (files starting
        with 'file') found in {}".format(vlsp_dir))
110     if not args.test_split and len(vlsp_test_files) == 0:
111         raise FileNotFoundError("No test files found in {}".
        format(vlsp_dir))
112     print("Loading training files from {}".format(vlsp_dir))
113     print("Procesing training files:\n  {}".format("\n  ".join(
        vlsp_train_files)))
114     with tempfile.TemporaryDirectory() as train_output_path:
115         vtb_convert.convert_files(vlsp_train_files,
        train_output_path, verbose=True, fix_errors=True,
        convert_brackets=args.convert_brackets, updated_tagset=
        updated_tagset)
116         # This produces a 0 length test set, just as a
        placeholder until the actual test set is released
117         if args.n_splits:
118             test_size = 0.1 if args.test_split else 0.0
119             dev_size = (1.0 - test_size) / args.n_splits
120             train_size = 1.0 - test_size - dev_size
121             for rotation in range(args.n_splits):
122                 # there is a shuffle inside the split routine,
123                 # so we need to reset the random seed each time
124                 random.seed(args.seed)
```

```
                rotation_name = "%s-%d-%d" % (dataset_name, rotation, args.n_splits)
                if args.test_split:
                    rotation_name = rotation_name + "t"
                vtb_split.split_files(train_output_path, paths["CONSTITUENCY_DATA_DIR"], rotation_name, train_size=train_size, dev_size=dev_size, rotation=(rotation, args.n_splits))
        else:
            test_size = 0.1 if args.test_split else 0.0
            dev_size = 0.1
            train_size = 1.0 - test_size - dev_size
            if args.test_split:
                dataset_name = dataset_name + "t"
            vtb_split.split_files(train_output_path, paths["CONSTITUENCY_DATA_DIR"], dataset_name, train_size=train_size, dev_size=dev_size)

    if not args.test_split:
        print("Procesing test files:\n  {}".format("\n  ".join(vlsp_test_files)))
        with tempfile.TemporaryDirectory() as test_output_path:
            vtb_convert.convert_files(vlsp_test_files, test_output_path, verbose=True, fix_errors=True, convert_brackets=args.convert_brackets)
            if args.n_splits:
                for rotation in range(args.n_splits):
                    rotation_name = "%s-%d-%d" % (dataset_name, rotation, args.n_splits)
                    vtb_split.split_files(test_output_path, paths["CONSTITUENCY_DATA_DIR"], rotation_name, train_size=0, dev_size=0)
            else:
                vtb_split.split_files(test_output_path, paths["CONSTITUENCY_DATA_DIR"], dataset_name, train_size=0, dev_size=0)

def process_arboretum(paths, dataset_name, *args):
    """
    Processes the Danish dataset, Arboretum
    """
    assert dataset_name == 'da_arboretum'

    arboretum_file = os.path.join(paths["CONSTITUENCY_BASE"], "danish", "arboretum", "arboretum.tiger", "arboretum.tiger")
    if not os.path.exists(arboretum_file):
```

```
156         raise FileNotFoundError("Unable to find input file for Arboretum.  Expected in {}".format(arboretum_file))
157
158     treebank = convert_tiger_treebank(arboretum_file)
159     datasets = utils.split_treebank(treebank, 0.8, 0.1)
160     output_dir = paths["CONSTITUENCY_DATA_DIR"]
161
162     output_filename = os.path.join(output_dir, "%s.mrg" % dataset_name)
163     print("Writing {} trees to {}".format(len(treebank), output_filename))
164     parse_tree.Tree.write_treebank(treebank, output_filename)
165
166     write_dataset(datasets, output_dir, dataset_name)
167
168
169 def process_starlang(paths, dataset_name, *args):
170     """
171     Convert the Turkish Starlang dataset to brackets
172     """
173     assert dataset_name == 'tr_starlang'
174
175     PIECES = ["TurkishAnnotatedTreeBank-15",
176               "TurkishAnnotatedTreeBank2-15",
177               "TurkishAnnotatedTreeBank2-20"]
178
179     output_dir = paths["CONSTITUENCY_DATA_DIR"]
180     chunk_paths = [os.path.join(paths["CONSTITUENCY_BASE"], "turkish", piece) for piece in PIECES]
181     datasets = read_starlang(chunk_paths)
182
183     write_dataset(datasets, output_dir, dataset_name)
184
185 def process_ja_alt(paths, dataset_name, *args):
186     """
187     Convert and split the ALT dataset
188
189     TODO: could theoretically extend this to MY or any other similar dataset from ALT
190     """
191     lang, source = dataset_name.split("_", 1)
192     assert lang == 'ja'
193     assert source == 'alt'
194
```

```
    PIECES = ["Japanese-ALT-Draft.txt", "Japanese-ALT-Reviewed.txt"]
    input_dir = os.path.join(paths["CONSTITUENCY_BASE"], "japanese", "Japanese-ALT-20210218")
    input_files = [os.path.join(input_dir, input_file) for input_file in PIECES]
    split_files = [os.path.join(input_dir, "URL-%s.txt" % shard) for shard in SHARDS]
    output_dir = paths["CONSTITUENCY_DATA_DIR"]
    output_files = [os.path.join(output_dir, "%s_%s.mrg" % (dataset_name, shard)) for shard in SHARDS]
    convert_alt(input_files, split_files, output_files)

def process_pt_cintil(paths, dataset_name, *args):
    """
    Convert and split the PT Cintil dataset
    """
    lang, source = dataset_name.split("_", 1)
    assert lang == 'pt'
    assert source == 'cintil'

    input_file = os.path.join(paths["CONSTITUENCY_BASE"], "portuguese", "CINTIL", "CINTIL-Treebank.xml")
    output_dir = paths["CONSTITUENCY_DATA_DIR"]
    datasets = convert_cintil_treebank(input_file)

    write_dataset(datasets, output_dir, dataset_name)

def process_id_icon(paths, dataset_name, *args):
    lang, source = dataset_name.split("_", 1)
    assert lang == 'id'
    assert source == 'icon'

    input_dir = os.path.join(paths["CONSTITUENCY_BASE"], "seacorenlp", "seacorenlp-data", "id", "constituency")
    input_files = [os.path.join(input_dir, x) for x in ("train.txt", "dev.txt", "test.txt")]
    datasets = []
    for input_file in input_files:
        trees = tree_reader.read_tree_file(input_file)
        trees = [Tree("ROOT", tree) for tree in trees]
        datasets.append(trees)

    output_dir = paths["CONSTITUENCY_DATA_DIR"]
    write_dataset(datasets, output_dir, dataset_name)
```

```
232
233 def process_ctb_51(paths, dataset_name, *args):
234     lang, source = dataset_name.split("_", 1)
235     assert lang == 'zh-hans'
236     assert source == 'ctb-51'
237
238     input_dir = os.path.join(paths["CONSTITUENCY_BASE"], "
    chinese", "LDC2005T01U01_ChineseTreebank5.1", "bracketed")
239     output_dir = paths["CONSTITUENCY_DATA_DIR"]
240     convert_ctb.convert_ctb(input_dir, output_dir, dataset_name,
     convert_ctb.Version.V51)
241
242 def process_ctb_90(paths, dataset_name, *args):
243     lang, source = dataset_name.split("_", 1)
244     assert lang == 'zh-hans'
245     assert source == 'ctb-90'
246
247     input_dir = os.path.join(paths["CONSTITUENCY_BASE"], "
    chinese", "LDC2016T13", "ctb9.0", "data", "bracketed")
248     output_dir = paths["CONSTITUENCY_DATA_DIR"]
249     convert_ctb.convert_ctb(input_dir, output_dir, dataset_name,
     convert_ctb.Version.V90)
250
251
252 def process_ptb3_revised(paths, dataset_name, *args):
253     input_dir = os.path.join(paths["CONSTITUENCY_BASE"], "
    english", "LDC2015T13_eng_news_txt_tbnk-ptb_revised")
254     if not os.path.exists(input_dir):
255         backup_input_dir = os.path.join(paths["CONSTITUENCY_BASE
    "], "english", "LDC2015T13")
256         if not os.path.exists(backup_input_dir):
257             raise FileNotFoundError("Could not find ptb3-revised
     in either %s or %s" % (input_dir, backup_input_dir))
258         input_dir = backup_input_dir
259
260     bracket_dir = os.path.join(input_dir, "data", "penntree")
261     output_dir = paths["CONSTITUENCY_DATA_DIR"]
262
263     # compensate for a weird mislabeling in the original dataset
264     label_map = {"ADJ-PRD": "ADJP-PRD"}
265
266     train_trees = []
267     for i in tqdm(range(2, 22)):
268         new_trees = tree_reader.read_directory(os.path.join(
    bracket_dir, "%02d" % i))
```

```
269         new_trees = [t.remap_constituent_labels(label_map) for t
    in new_trees]
270         train_trees.extend(new_trees)
271
272     move_tregex = "_ROOT_ <1 __=home <2 /^[.]$/=move"
273     move_tsurgeon = "move move >-1 home"
274
275     print("Moving sentence final punctuation if necessary")
276     with tsurgeon.Tsurgeon() as tsurgeon_processor:
277         train_trees = [tsurgeon_processor.process(tree,
    move_tregex, move_tsurgeon)[0] for tree in tqdm(train_trees)]
278
279     dev_trees = tree_reader.read_directory(os.path.join(
    bracket_dir, "22"))
280     dev_trees = [t.remap_constituent_labels(label_map) for t in
    dev_trees]
281
282     test_trees = tree_reader.read_directory(os.path.join(
    bracket_dir, "23"))
283     test_trees = [t.remap_constituent_labels(label_map) for t in
     test_trees]
284     print("Read %d train trees, %d dev trees, and %d test trees"
     % (len(train_trees), len(dev_trees), len(test_trees)))
285     datasets = [train_trees, dev_trees, test_trees]
286     write_dataset(datasets, output_dir, dataset_name)
287
288 def process_en_masc(paths, dataset_name, *args):
289     """
290     Convert the MASC english treebank
291     """
292     lang, source = dataset_name.split("_", 1)
293     assert lang == 'en'
294     assert source == 'masc'
295
296     input_path = os.path.join(paths["CONSTITUENCY_BASE"], "
    english", "MASC")
297     output_dir = paths["CONSTITUENCY_DATA_DIR"]
298     datasets = convert_en_masc(input_path)
299
300     write_dataset(datasets, output_dir, dataset_name)
301
302 DATASET_MAPPING = {
303     'da_arboretum': process_arboretum,
304
305     'en_ptb3-revised': process_ptb3_revised,
```

```
306
307     'id_icon':      process_id_icon,
308
309     'it_turin':     process_it_turin,
310     'it_vit':       process_it_vit,
311
312     'ja_alt':       process_ja_alt,
313
314     'pt_cintil':    process_pt_cintil,
315
316     'tr_starlang':  process_starlang,
317
318     'vi_vlsp09':    process_vlsp09,
319     'vi_vlsp21':    process_vlsp21,
320     'vi_vlsp22':    process_vlsp22,
321     'vi_vlsp23':    process_vlsp22,  # options allow for this
322
323     'zh-hans_ctb-51':   process_ctb_51,
324     'zh-hans_ctb-90':   process_ctb_90,
325
326     'en_masc':   process_en_masc,
327 }
328
329 def main(dataset_name, *args):
330     paths = default_paths.get_default_paths()
331
332     random.seed(1234)
333
334     if dataset_name in DATASET_MAPPING:
335         DATASET_MAPPING[dataset_name](paths, dataset_name, *args
    )
336     else:
337         raise UnknownDatasetError(dataset_name, f"dataset {
    dataset_name} currently not handled by prepare_con_dataset")
338
339 if __name__ == '__main__':
340     main(sys.argv[1], sys.argv[2:])
```

Listing 2: prepare_con_dataset.py

B.3 convert_en_masc.py para parser de constituyentes

A continuación se presenta el script convert_en_masc.py generado para procesar el Treebank *The Manually Annotated Sub-Corpus (MASC)* en el Ejercicio de parsers

de constituyentes.

```
import os

from stanza.models.constituency import tree_reader
from stanza.utils.datasets.constituency import utils


def read_files(input_path):
    # The files are read from the specified directory
    tree_files = [os.path.join(input_path, x) for x in os.listdir(input_path)]
    all_sents = []
    sent = ''
    for file in tree_files:
        with open(file, encoding='utf-8') as fin:
            lines = fin.readlines()
            for line in lines:
                # The files all have the same format, which is that after each sentence there is a line with only the character \n
                # So the sentences will be separated from each other using this character.
                if line != '\n':
                    sent += line.strip()
                else:
                    # The ROOT is added as per Stanza guidelines
                    sent = sent.replace('( (' , '(ROOT (').replace('\n', '')
                    # There is an exception with the Sentences that have the structure '(CODE', so those sentences will be skipped.
                    if '(CODE' not in sent and sent != '':
                        # A basic check is done to ensure that the parenthesis are properly matched.
                        if sent.count('(') == sent.count(')'):
                            all_sents.append(sent)
                    sent = ''
    return all_sents

def convert_en_masc(input_path, train_size=0.8, dev_size=0.1):
    sents = read_files(input_path)
    natural_trees = []

    # The following code is recycled from another language, as the procedure after the processing is similar for all of them
```

```
    .
    for sent in sents:
        trees = tree_reader.read_trees(sent)
        tree = trees[0]
        natural_trees.append(tree)

    print("Read %d natural trees" % len(natural_trees))
    train_trees, dev_trees, test_trees = utils.split_treebank(
    natural_trees, train_size, dev_size)
    print("Split %d trees into %d train %d dev %d test" % (len(
    natural_trees), len(train_trees), len(dev_trees), len(
    test_trees)))
    print("Total lengths %d train %d dev %d test" % (len(
    train_trees), len(dev_trees), len(test_trees)))
    return train_trees, dev_trees, test_trees
```

Listing 3: convert_en_masc.py

B.4 Función wrapper para mostrar métricas Conllu 2018 por pantalla

En este Anexo se adjunta la función generada para mostrar métricas por pantalla para la evaluación de los parsers de dependencias:

```
# Funcion para imprimir por pantalla la evaluacion, como el
    conll18_ud_eval tiene esta implementacion en main,
# se hace un wrapper para emplearlo como funcion fuera del
    script

def print_eval(verbose:bool, counts:bool, evaluation):
    if not verbose and not counts:
    print("LAS F1 Score: {:.2f}".format(100 * evaluation["LAS"].
    f1))
    print("MLAS Score: {:.2f}".format(100 * evaluation["MLAS"].
    f1))
    print("BLEX Score: {:.2f}".format(100 * evaluation["BLEX"].
    f1))
    else:
    if counts:
        print("Metric     | Correct   |      Gold | Predicted |
    Aligned")
    else:
        print("Metric     | Precision |    Recall |  F1 Score |
    AligndAcc")
```

```
    print("
-----------+-----------+-----------+-----------+-----------")
    for metric in["Tokens", "Sentences", "Words", "UPOS", "XPOS"
, "UFeats", "AllTags", "Lemmas", "UAS", "LAS", "CLAS", "MLAS"
, "BLEX"]:
        if counts:
        print("{:11}|{:10} |{:10} |{:10} |{:10}".format(
        metric,
        evaluation[metric].correct,
        evaluation[metric].gold_total,
        evaluation[metric].system_total,
        evaluation[metric].aligned_total or (evaluation[metric].
correct if metric == "Words" else "")
        ))
        else:
        print("{:11}|{:10.2f} |{:10.2f} |{:10.2f} |{}".format(
        metric,
        100 * evaluation[metric].precision,
        100 * evaluation[metric].recall,
        100 * evaluation[metric].f1,
        "{:10.2f}".format(100 * evaluation[metric].
aligned_accuracy) if evaluation[metric].aligned_accuracy is
not None else ""
        ))


```

Listing 4: Wrapper function

C Arboles eliminados en el ejercicio de parsers de Constituyentes

En este anexo se muestran los dos arboles que se eliminan de la partición Train del Treebank *The Manually Annotated Sub-Corpus (MASC)* en Inglés:

(ROOT (S (PP-LOC (IN In) (NP (NP-TTL (JJ Great) (NNS Expectations)) (-LRB- -LRB-) (NP (NN Chapter) (CD VI)) (-RRB- -RRB-))) (, ,) (NP-SBJ (PRP we)) (VP (VBP learn) (SBAR (WHADVP-3 (WRB how)) (“ “) (NFP ...) (ROOT (S (NP-SBJ-1 (NNP Mr.) (NNP Whopsle)) (, ,) (S-ADV (NP-SBJ-2 (-NONE- *PRO*-1)) (VP (VBG being) (VP (VBN knocked) (NP (-NONE- *-2)) (PRT (RP up))))) (, ,) (VP (VBD was) (PP-PRD (IN in) (NP (PDT such) (DT a) (ADJP (RB very) (JJ bad)) (NN temper))) (ADVP-MNR (-NONE- *T*-3)))))) (. .) (” ”)))

(ROOT (SBAR-PRP (IN because) (S (NP-SBJ (PRP they)) (ADVP (RB just)) (VP (VBP say) (SBAR (-NONE- 0) (S (NP-SBJ (EX there)) (VP (VBZ ’s) (CC either) (NP-PRD (NP (DT no) (NN room)) (PP-LOC (PP (IN in) (NP (DT the) (NN system))) (PRN (S (NP-SBJ (PRP you)) (VP (VBP know)))) (PP (IN in) (NP (DT the) (NN jails)))) (PP (IN for) (NP (PRP them)))))))))) (SU /))

D Tablas completas de resultados de evaluación de parsers de Dependencias

En este anexo se incluyen las tablas completas de resultados del script de evaluación empleado en el ejercicio de parsers de Dependencias.

D.1 Inglés

	Precision	Recall	F1 Score	Aligned Acc
Tokens	94.20	97.13	95.64	-
Sentences	71.54	69.72	70.62	-
Words	96.61	98.22	97.41	-
UPOS	90.22	91.72	90.96	93.38
XPOS	88.23	89.70	88.96	91.33
UFeats	90.64	92.14	91.38	93.81
AllTags	86.54	87.98	87.25	89.57
Lemmas	91.63	93.15	92.39	94.84
UAS	79.99	81.31	80.64	82.79
LAS	69.77	70.93	70.34	72.22
CLAS	71.54	62.83	66.90	64.07
MLAS	64.50	56.65	60.32	57.77
BLEX	68.46	60.13	64.03	61.32

Table 13: Métricas en idioma Inglés empleando SpaCy

	Precision	Recall	F1 Score	Aligned Acc
Tokens	99.24	99.18	99.21	-
Sentences	80.97	65.33	72.32	-
Words	99.06	98.96	99.01	-
UPOS	98.51	98.41	98.46	99.44
XPOS	98.44	98.34	98.39	99.38
UFeats	98.63	98.52	98.57	99.56
AllTags	98.26	98.15	98.20	99.19
Lemmas	96.36	96.25	96.31	97.27
UAS	85.87	85.78	85.83	86.69
LAS	83.57	83.48	83.53	84.36
CLAS	79.50	79.13	79.32	80.05
MLAS	78.46	78.10	78.28	79.00
BLEX	76.75	76.39	76.57	77.27

Table 14: Métricas en idioma Inglés empleando Stanza

D.2 Italiano

	Precision	Recall	F1 Score	Aligned Acc
Tokens	99.91	99.85	99.88	-
Sentences	98.35	99.17	98.76	-
Words	92.30	85.72	88.89	-
UPOS	89.46	83.08	86.15	96.92
XPOS	89.18	82.82	85.88	96.62
UFeats	89.40	83.03	86.10	96.86
AllTags	88.08	81.80	84.82	95.43
Lemmas	89.49	83.10	86.18	96.95
UAS	80.17	74.46	77.21	86.86
LAS	73.00	67.79	70.30	79.09
CLAS	78.35	69.73	73.79	71.28
MLAS	62.92	55.99	59.25	57.24
BLEX	75.57	67.25	71.17	68.75

Table 15: Métricas en idioma Italiano empleando SpaCy

	Precision	Recall	F1 Score	Aligned Acc
Tokens	99.84	99.78	99.81	-
Sentences	98.55	98.96	98.76	-
Words	99.46	99.29	99.38	-
UPOS	99.44	99.27	99.36	99.98
XPOS	99.44	99.27	99.36	99.98
UFeats	99.42	99.25	99.34	99.96
AllTags	99.42	99.25	99.34	99.96
Lemmas	97.73	97.56	97.65	98.26
UAS	90.17	90.02	90.09	90.66
LAS	87.59	87.43	87.51	88.06
CLAS	82.44	82.02	82.23	82.91
MLAS	81.55	81.14	81.35	82.02
BLEX	80.22	79.82	80.02	80.68

Table 16: Métricas en idioma Italiano empleando Stanza

D.3 Griego

	Precision	Recall	F1 Score	Aligned Acc
Tokens	99.77	99.86	99.81	-
Sentences	91.34	87.94	89.61	-
Words	97.37	95.17	96.26	-
UPOS	93.32	91.21	92.25	95.84
XPOS	93.16	91.06	92.10	95.68
UFeats	87.12	85.16	86.13	89.48
AllTags	85.47	83.54	84.49	87.77
Lemmas	87.01	85.04	86.02	89.36
UAS	83.01	81.14	82.06	85.25
LAS	75.68	73.97	74.81	77.72
CLAS	74.50	68.29	71.26	68.40
MLAS	60.03	55.03	57.42	55.12
BLEX	63.36	58.09	60.61	58.18

Table 17: Métricas en idioma Griego empleando SpaCy

	Precision	Recall	F1 Score	Aligned Acc
Tokens	99.53	99.74	99.64	-
Sentences	90.27	87.50	88.86	-
Words	99.53	99.73	99.63	-
UPOS	99.51	99.71	99.61	99.98
XPOS	99.51	99.71	99.61	99.98
UFeats	99.44	99.63	99.54	99.91
AllTags	99.44	99.63	99.54	99.91
Lemmas	95.76	95.95	95.86	96.21
UAS	89.19	89.36	89.28	89.61
LAS	86.38	86.55	86.47	86.79
CLAS	80.99	80.95	80.97	81.30
MLAS	79.68	79.64	79.66	79.98
BLEX	76.09	76.05	76.07	76.38

Table 18: Métricas en idioma Griego empleando Stanza

D.4 Chino

	Precision	Recall	F1 Score	Aligned Acc
Tokens	32.06	51.26	39.45	-
Sentences	22.14	48.00	30.30	-
Words	32.06	51.26	39.45	-
UPOS	28.57	45.69	35.16	89.13
XPOS	29.05	46.45	35.75	90.63
UFeats	31.31	50.07	38.53	97.68
AllTags	28.08	44.91	34.56	87.61
Lemmas	32.06	51.26	39.45	100.00
UAS	7.58	12.11	9.32	23.63
LAS	6.75	10.79	8.30	21.05
CLAS	3.15	5.72	4.06	20.18
MLAS	2.58	4.68	3.32	16.51
BLEX	3.15	5.72	4.06	20.18

Table 19: Métricas en idioma Chino empleando SpaCy

D.5 Ruso

	Precision	Recall	F1 Score	Aligned Acc
Tokens	99.46	99.51	99.48	-
Sentences	95.99	95.51	95.75	-
Words	99.46	99.51	99.48	-
UPOS	99.46	99.51	99.48	100.00
XPOS	99.42	99.47	99.45	99.96
UFeats	99.35	99.40	99.38	99.89
AllTags	99.32	99.38	99.35	99.87
Lemmas	93.64	93.68	93.66	94.15
UAS	87.85	87.90	87.87	88.33
LAS	83.92	83.96	83.94	84.38
CLAS	81.55	81.62	81.58	82.05
MLAS	80.51	80.58	80.55	81.01
BLEX	75.37	75.43	75.40	75.84

Table 20: Métricas en idioma Ruso empleando Stanza