LoreNexus: Clasificación de nombres derivados en mundos virtuales mediante modelos de PLN

Aingeru García Blas
Universidad del Pais Vasco UPV/EHU
agarcia383@ikasle.ehu.eus

Abstract

LoreNexus es una plataforma centrada en extraer o identificar el universo de origen y las posibles influencias detrás de los nombres utilizados en videojuegos online. Presento dos enfoques para los modelos de lenguaje utilizados, el primero utilizando Flair framework, y el segundo, uno propio implementando desde cero con Pytorch, con estrategias como BiLSTMs y tokenización a nivel de carácter, junto con diversas técnicas de procesamiento y análisis de lenguaje natural. El objetivo principal es clasificar nombres personalizados en función de patrones fonéticos, morfológicos, para intentar detectar referencias que van desde universos ficticios como los de Tolkien o Star Wars, hasta personajes históricos, mitológicos o incluso términos con connotaciones insultantes. Este proyecto ha sido muy enriquecedor y he podido apreciar un proceso de aprendizaje progresivo durante el desarrollo.

1 Introducción

En comunidades de videojuegos online, los nombres personalizados tienden reflejar influencias de universos ficticios y culturales. Los jugadores eligen nombres inspirados en grandes sagas como las de Tolkien, Star Wars o Game of Thrones, personajes históricos o incluso términos con connotaciones humorísticas o insultantes, tan frecuentes en internet. Con lo que, el objetivo de este proyecto se centra en intentar clasificar dichos nombres analizando patrones lingüísticos y variaciones fonéticas, mediante modelos de lenguaje. Algo que considero importante también, es el destacar y exponer en el presente informe el proceso de aprendizaje al que me ha llevado LoreNexus - pienso que ha enriquecido muchísimo mi visión de la temática subyacente y me ha ayudado a mejorar.

1.1 Framework

Con la intención de automatizar procesos y, sobre todo, generalizar para un posible escalado futuro, el desarrollo ha terminado convergiendo en un *mini-framework* modular altamente configurable para evaluar y comparar modelos. He implementado un entorno para probar parámetros mediante grids y *enfrentar* múltiples modelos para generar estadísticas. Parte de este marco es una clase wrapper extensible que nos permite integrar nuevos modelos fácilmente, un pipeline de datos para regenerar el corpus unificado con distintos datasets y configuraciones, logging exhaustivo, gráficas y una app CLI para inferencias manuales.

2 Fuentes de Datos

Al ser datos personalizados y requerir recolección de multitud de fuentes diferentes, se pueden clasificar en dos fuentes principales: Datasets obtenidos a través de APIs públicas (o en algún caso, datasets ya preparados) y datos extraídos de libros en formatos pdf, mediante técnicas como **NER**.

Todos ellos se tratan de manera modular, para luego ser integrados y unificados en un único corpus.

2.1 Datasets

2.1.1 Wikidata

Wikidata ha sido la principal fuente de datos, su amplitud y cantidad inmensa de información, al final resulta en una especie de monopolio en estos contextos y ha dejado obsoletas todas las APIs especializadas que durante la planificación recolecté. En este caso, he realizado consultas **SPARQL** para extraer información de todos los universos ficticios/fantásticos.

Posteriormente, incluí personajes históricos de gran relevancia en las consultas (dataset separado). Los criterios que he seguido son: el hecho de que consten como figuras relevantes, y hayan contribuido en aŕeas como ciencia, arte, literatura, música etc, desde el año 1400. Los considero *de gran relevancia*, si poseen más de 30 *sitelinks* (número de enlaces interwiki. Número 30 ha sido elegido de manera empírica).

2.1.2 Slurs

Una vez cubiertos universos ficticios, nos tenemos que acordar que estamos en Internet y la gente da rienda suelta a su imaginación y creatividad para incluir de manera ocurrente elementos insultantes en los nombres o nicknames. Con lo que, para dentificar términos ofensivos, he utilizado el dataset proporcionado por **Surge AI Profanity**, que contiene una amplia colección de *slurs* y expresiones insultantes.

2.1.3 Mitología

Los datos relacionados con personajes y entidades mitológicas han sido obtenidos a través de la API de **GodChecker**.

2.2 NER (Reconocimiento de entidades nombradas)

El módulo desarrollado para encargarse de las NERs, está pensado para procesar automáticamente todos los archivos PDF en la carpeta `books` que no han sido previamente procesados, así como para construir el corpus ya etiquetado. Está configurado para utilizar el modelo preentrenado `en_core_web_trf` de **spaCy** , basado en el modelo transformer: **RoBERTa**.

Dado que ya dispongo de datos ricos en nombres personajes gracias a Wikidata, el uso de NERs lo he limitado a LOC y GPE para ampliar la cobertura de información contextual dentro de algunos universos. Por ejemplo, incluir ubicaciones como *Rivendell* y *Mordor* (*El Señor de los Anillos*), *Winterfell* (*Game of Thrones*) y *Tatooine* (*Star Wars*). La sinergia entre el módulo implementado y el modelo de Spacy identifica automáticamente las entidades, normaliza y etiqueta (basadándose en el libro del que fueron extraídas).

3 Generación de datos

3.1 Pipeline de procesado y etiquetado

La ejecución del pipeline que se encarga de generar el dataset unificado final está totalmente automatizada y es configurable. Mediante un fichero `config.json`, se pueden especificar parámetros como las rutas, fuentes de datos, estratificación, etiquetas a incluir/excluir, y configuraciones exhaustivas del data augmentation.

1. **Carga de configuración:**
   (a) Se parsea la configuración de `config.json` y con ello establecen los parámetros para todos los procesadores y builders.
2. **Procesamiento por fuentes de datos:**
   (a) **Datasets y NER extraídos:**
      i. Cada uno tiene su procesador propio y se encarga de gestionar las peculiaridades específicas, pero el más notable es el de **Wikidata**:
         A. Dispone de dos builders separados: uno para datos relacionados con personajes históricos, y otro para personajes de universos ficticios.
         B. También se hace una eliminación de la palabra **universe** que está presente en la enorme mayoría de los casos.
   (b) **Etiquetado:**
      i. Cada dataset se etiqueta por su naturaleza: *Historical* para datos históricos, *Mythology* para datos mitológicos, *Offensive* para términos ofensivos...etc
      ii. El dataset de Wikidata contiene cientos de etiquetas diferentes, durante el procesado se etiqueta **todo** y normaliza. El proceso de pipeline se encarga de, en función de la configuración, incluir o no ese universo.
      iii. Las etiquetas siguen el formato de `FastText`:

```
__label__StarWars Tatooine
__label__Historical Einstein
__label__GameOfThrones Ned Stark
```

   (c) **Homogeneización:**
      i. Decidí unificar las etiquetas, ya que en muchos casos tiene variaciones pese a ser el mismo contexto. Ejemplo: `Star Wars animation` y `Star Wars world`, que consolido bajo una única etiqueta `Star Wars`.
3. **Normalización de datos:**
   (a) Necesito que **todos** los nombres y etiquetas sean caracters alfabéitcos:
      i. Normalización Unicode a ASCII.
      ii. Todo contenido entre paréntesis, números y puntuación - eliminados.
      iii. Sustitución de carácteres como guiones, apóstrofes etc, por espacios.
4. **Aumento de datos (si está habilitado):**
   (a) Se aplican técnicas para producir datos sintéticos y hacer que el modelo se esfuerce en generalizar.
5. **Generación de datasets finales:**
   (a) En la estratificación, genero los sets con una distribución de etiquetas equitativa, agrupo primero por etiquetas todo el dataset y de ahí, se produce la divisón.
   (b) Se crean los sets de train, dev y test en función de lo configurado (para la entrega: 80%, 10%, 10%)
6. **Dump de configuración:**
   (a) Al finalizar, los datasets se dejan en la ruta *dataset/output*; con un archivo **data_config.info** a modo de dump que describe la configuración utilizada.

Data Augmentation

Todos los parámetros de este proceso se pueden ajustar de manera granular, desde cantidad de carácteres, probabilidades, intensidad, hasta exclusión de etiquetas específicas o decidir si activar una técnica o no.

La idea en general es la de hacer cambios fonéticos pequeños, simular errores de escritura comunes...etc Al fin y al cabo todo es para poder mostrar al modelo variantes típicas que los jugadores tienden a realizar en los casos en los que el nombre original no está disponible: (`Chewbacca` → `Chewbaka`, `Greyjoy` → `Grejoyy`, `Aragorn` → `Araghornn`).

1. **Operaciones básicas:**
   - **División y combinación de nombres:** Generación de fragmentos a partir de nombres compuestos (`Jon Snow` → `Jon`, `Snow`) y variantes sin espacios (`Jon Snow` → `JonSnow`).
   - **Swap interno:** permutaciones internas controladas - se hace swap de caracteres internos, pero se preservan el primero y el último (`Tyrion` → `Tyorin`).
2. **Técnicas avanzadas (`TextAttack`):**
   - **Sustituciones fonéticas:** Casos del inglés como:`oo` → `u`, ejemplo: `Tatooine` → `Tatuine`.
   - **Swap:** Swap de caracteres vecinos (sin necesidad de preservar el primero/último) (`Cloud` → `Cluod`).
   - **Inserción:** Inserta caracteres aleatorios en posiciones aleatorias (`Cloud` → `Cloudz`).
   - **Eliminación:** Elimina caracteres aleatoriamente (`Cloud` → `Coud`).
   - **Duplicación:** Duplica aleatoriamente (`Cloud` → `Clouud`).
3. **Configuración adaptable:**
   - **Intensidad ajustable:** Si se desea hacer varias iteraciones de aumento de datos sobre el mismo dataset, se puede ajustar el parámetro `intensity`.
   - **Exclusión de etiquetas:** En caso de querer que alguna etiqueta no esté sujeta a ser aumentada.

4 Diseño y construcción de los modelos

4.1 BiLSTM y tokenización a nivel de carácter

Desde un principio se definió que la arquitectura sería un **LSTM bidireccional**, y que los **tokens serían a nivel de carácter**, de manera que estos son convertidos a vectores densos mediante capas de embeddings aprendibles.

Por un lado, la LSTM bidireccional captura las secuencias en ambos sentidos y esto ayuda a contextualizar mejor, y por otro, los embeddings de carácteres sirven para aprender representaciones morfológicas y fonéticas en los nombres - claro, esto es **perfecto** para identificar nombres con variaciones tipográficas, que es justo lo que busca LoreNexus.

4.2 Wrappers del framework

Al ser mi primera vez trabajando con modelos de lenguaje de manera independiente (o redes neuronales, en general), decidí empezar con una librería que facilita mucho el proceso (**Flair Framework**), para ver que lo que tenía en mente *más o menos funcionaba*. Una vez obtuve resultados decentes, me aventuré a programarlo `from scratch`.

4.2.1 LoreNexusFlairModel (Flair)

Flair facilita muchísimo las cosas, como su uso es muy simple y además, el grosso en el que me he centrado es en otro modelo, lo expongo superficialmente:

Modelo: `TextClassifier`

1. Capa de embedding(dim, dim)
2. LSTM(dim, dim) [`num_layers > 1`: se aplica otra LSTM(dim, dim)]
3. Linear(dim, dim) seguido de Dropout

- **ClassificationCorpus:** Se encarga de organizar los datasets en batches etc.
- **Entrenamiento:** Para el training, he utilizado *AdamW*, regularización mediante *dropout* (si más de 1 capa) y *weight decay*.

4.2.2 LoreNexusPytorchModel (Pytorch, `from scratch`)

Este ha sido, en gran medida, el grosso del tiempo invertido en el proyecto, intento resumir brevemente sus componentes aunque se pueden ver en el fichero `/models/pytorch/model.py`, aquí.

- **Vocabulario e indexación:**
  - **Codificación de caracteres:** Mapeo cada carácter a un índice único, en una estructura `CharacterVocab` que además incluye tokens especiales (`<PAD>` y `<UNK>`). Vocabulario dinámico, se construye según va descubriendo nuevos carácteres.
  - **Etiquetas indexadas:** `LabelEncoder` codifica etiquetas a índices y viceversa.
- **Arquitectura del modelo:**
  - **Capa de embeddings**
  - **Capa BiLSTM**
  - **Capa FC** Nota: Concateno los estados ocultos finales de las ambas direcciones.
  - **Inicialización de pesos:** *Xavier Uniform* y *Kaiming Uniform* dependiendo de la capa.
- **Gestión de datos:**
  - **Dataset tokenizado:** Primero codifico los nombres como secuencias de índices - truncados o rellenados con `padding`, para poder devolverlo en `__getitem__` junto con la etiqueta.
  - **DataLoader:** Le nutro con los datasets mencionados.
- **Entrenamiento y optimización:**
  - **Criterio de pérdida:** Entropía cruzada (ponderada, expongo más adelante esto).
  - **Optimizador:** `AdamW` con regularización por *weight decay*.
  - **Scheduler:** Se hace reducción progresiva del `learning rate`, uso `StepLR`, params `step_size=5`, `gamma=0.5`.
  - **Regularización:** `dropout` y `weight decay`
- **Evaluación e inferencia:**
  - **Métricas:** Reportes detallados gracias a `classification report`.
  - **Inferencia:** Devuelvo etiquetas más probables con sus respectivas puntuaciones.

4.2.3 Checkpoints

En cada checkpoint, guardo el modelo como un diccionario de PyTorch con: los pesos, el estado del optimizador etc, pero también `char_vocab`, `label_encoder`, para poder utilizar en el proceso de inferencia (app `CLI`).

4.2.4 Logs

El sistema de logs se centra en documentar exhaustivamente los entrenamientos:

1. **Carpeta de logs:** Creada para cada lanzamiento de entrenamiento.
   (a) **Archivos:**
   - `.log`: Registro detallado en texto con:
     i. Hiperparámetros, métricas por época y la mejor de las épocas.
     ii. Configuración de los datos utilizados (datasets, etiquetas, augmentación, etc.).
   - `.png`: Gráfico de métricas (loss, val) por época, muestra de hyperparams y matriz de confusión.
2. **Salida en consola:** Resumen por época con métricas, ponderaciones de etiquetas, y muchos otros datos.

5 Training Grounds

El **Training Grounds** es una especie de entorno para simular *enfrentamientos entre modelos* y explorar espacios de hiperparámetros - que al fin y al cabo, simplemente se centra en **entrenar** modelos con múltiples grids de parámetros (configurables, tantas como se quiera en `params_grids.json`), para poder producir logs y métricas al detalle en cada run. Toda esta funcionalidad está gestionada por la clase `HyperparameterTuner`.

1. **Configuración:**
   - `param_grids.json`: Todas las grids de hiperparámetros con las que entrenar.
   - `models`: Se pueden inyectar todos los modelos que queramos, por ahora solo dos disponibles: `LoreNexusFlairModel`, `LoreNexusPytorchModel`).
2. **Output:**
   - **Carpeta `arena`:** Contiene logs individuales (`model_name_hyperparameter_tuning.log`) con:
     (a) Grid de hyperparams utilizada.
     (b) Métricas por training.
     (c) Mejores resultados, muestra el ganador, y por ende la **mejor configuración identificada**

5.1 Modelos, a la arena!

Para la evaluación se utilizará **F1-Score (Macro)** al tener clases desbalanceadas. La configuración de los datos es:

```
train_size: 0.8
dev_size: 0.1
test_size: 0.1
Training samples: 567090
Validation samples: 70880
unique_names: True
augmentation:
  enabled: True
  only_basic_augmentation: False
  intensity: 2
  swap_characters: {'enabled': True, 'pct_words_to_swap': 0.5,
  'transformations_per_example': {'min': 1, 'max': 4}}
 insert_characters: {'enabled': True, 'pct_words_to_swap': 0.6,
  'transformations_per_example': {'min': 1, 'max': 4}}
 delete_characters: {'enabled': True, 'pct_words_to_swap': 0.5,
  'transformations_per_example': {'min': 1, 'max': 3}}
 duplicate_characters: {'enabled': True, 'pct_words_to_swap': 0.4,
  'transformations_per_example': 2}
  split_names: {'enabled': True, 'join_parts': True}
  label_exclusion: {'enabled': False, 'excluded_labels': []}
  internal_swap: {'enabled': True, 'swap_probability': 0.5}
labels: ['HarryPotter', 'StarWars', 'Tolkien', 'Warcraft',
'DragonBall', 'Naruto', 'ForgottenRealms', 'FinalFantasy',
'GameofThrones', 'TheWitcher', 'DoctorWho', 'Discworld',
'Mythology', 'Offensive', 'Historical']
```

5.2 Combatiente A: Flair

B	Ep	WD	H	E	L	D	F1
32	5	0.03	64	32	1	0.5	0.3227
32	7	0.03	128	64	2	0.4	0.4905
64	7	0.03	128	64	2	0.3	0.6013
32	10	0.02	192	96	2	0.3	0.6880
32	12	0.02	256	128	3	0.3	0.7214

Table 1: Ver log completo: FlairModel Log

5.3 Combatiente B: Pytorch

B	Ep	WD	H	E	L	D	F1
32	5	0.03	64	32	1	0.5	0.62
32	7	0.03	128	64	2	0.4	0.89
64	7	0.03	128	64	2	0.3	0.90
32	10	0.02	192	96	2	0.3	0.92
32	12	0.02	256	128	3	0.3	0.94

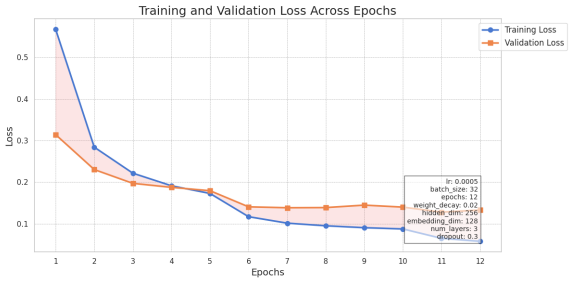
Table 2: Ver log completo: PytorchModel Log

5.4 And the winner is...

Model: LoreNexusPytorchModel (from scratch)
F1-Score 0.94
Hyperparameters: `[lr=0.0005, batch_size=32, [epochs=12, w_decay=0.02, hidden_dim=256, [embedd_dim=128, num_layers=3, dropout=0.3]`

5.4.1 Intentando mejorar al ganador

He tomado los datos del ganador del *torneo* y he hecho que entrene durante más épocas, para ver si lo estaba limitando con tan solo 12. Tras el train, el score es algo peor, que como se puede observar: **0.93**:



B	Ep	WD	H	E	L	D	F1
32	**25**	0.02	256	128	3	0.3	0.93

Table 3: Link: Full report (incluidos gráficos)

6 Conclusiones - Aprendiendo a la par que el modelo

Durante el proceso de desarrollo, considero que he aprendido mucho gracias a las dificultades o baches que me he ido encontrando por el camino. Con lo que he pensado en describir algunos puntos de inflexión que considero lo representan, aunque ha habido más, claro.

Con lo que, me gusta pensar que, *conforme he ido aprendido yo, también así lo ha ido haciendo el modelo*. Para todo lo que mostraré a continuación, he utilizado el modelo de **LoreNexusPytorchModel** (*from scratch*).

6.1 Importancia de una correcta distribución de datos

Al principio, pude comprobar como una distribución no equitativa entre los sets, puede dar lugar a etiquetas que **no** están presentes en alguno de ellos, o cómo unas proporciones desbalanceadas, dan resultados totalmente inconsistentes.

6.2 Sobre los optimizadores

Al comenzar, tenía la idea de jugar con diferentes optimizadores. Sin embargo, tras utilizar `SGD`, `Adam` (con/sin momentum), me percaté de que convenía utilizar `Adamw`, para poder contrarrestar la tendencia al overfit que el problema que intenta resolver LoreNexus, tiene de manera inherente. De ésta forma, puedo hacer una regularización más explítica, gracias al **weight decay**.

6.3 Pesos en CrossEntropy

Al tener un desbalance de etiquetas tan masivo, he probado a intentar balancearlo haciendo que la función de pérdida lo tenga en cuenta: `CrossEntropy(weight=class_weights)`, para que pondere con la inversa de la frecuencia de cada etiqueta. Desafortunadamente, no he notado gran diferencia. Veamos un ejemplo para ilustrar las relaciones:

- **Label indexes:** `[0-14]`
- **LabelStrings:** `['Historical','Tolkien','Warcraft' 'Offensive','StarWars','Naruto','HarryPotter', 'Mythology', 'GameofThrones','FinalFantasy', 'DoctorWho', 'TheWitcher', 'ForgottenRealms', 'DragonBall', 'Discworld']`
- **Label counts:** `{0: 278626, 7: 55868, 1: 35661, 8: 28598, 4: 27008, 3: 22556, 6: 21927, 10: 4984, 5: 4539, 12: 4021, 13: 3384, 2: 3304, 11: 3103, 9: 2280, 14: 1393}`
- **Class weights:** `[1.78, 13.94, 150.5, 22.04, 18.41, 109.55, 22.68, 8.9, 17.39, 218.09, 99.77, 160.25, 123.66, 146.94, 356.96]`

6.4 LoreNexusPytorch big sizes

Veamos unas pruebas donde utilizo tamaños para los estados ocultos y embeddings superiores a los vistos durante *la arena*. Como explicaré después, durante todo el proceso los valores que he utilizado para el desarrollo y pruebas han sido similares a estos. Los datos no han sido regenerados y su augmentation es la misma que la vista anteriormente.

B	Ep	WD	H	E	L	D	Acc	F1
32	10	0.02	256	256	1	0.2	0.9375	0.90
32	10	0.02	756	256	1	0.4	0.9483	0.92
64	15	0	512	128	2	0.3	0.9520	0.93
64	15	0.03	515	128	2	0.5	0.9576	0.94
64	20	0.03	756	128	2	0.5	0.9596	0.94
64	20	0.03	756	128	3	0.5	0.9597	0.94

Table 4: Link: Ver logs

6.5 Eliminando nombres duplicados

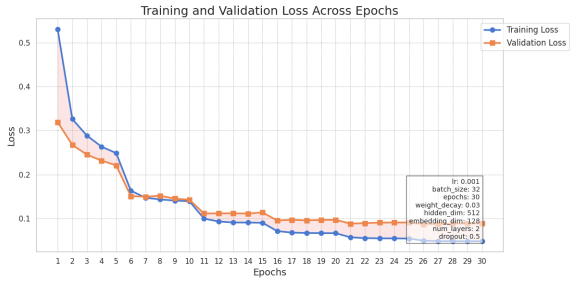
```
Con nombres duplicados:
Training samples: 670,540
Validation samples: 83,812

Nombres únicos:
Training samples: 477,546
Validation samples: 59,689
```

B	Ep	WD	H	E	L	D	T.L	V.L	F1
32	30	0.03	512	128	2	0.5	0.0482	0.0887	0.95
32	30	0.03	512	128	2	0.5	0.0461	0.1195	0.94

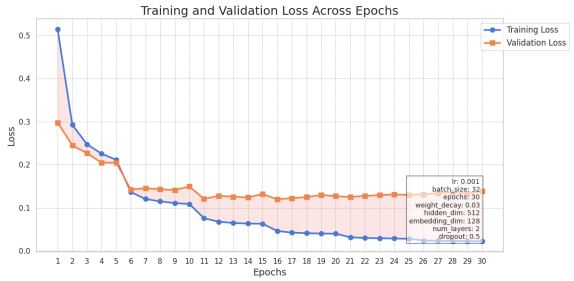
Table 5: Primera fila es con nombres duplicados y segunda sin ellos, únicos. T.L y V.L son los best train y validation loss, respectivamente. La idea es ilustrar **contaminación** al estratificar con duplicados.

Veámoslo gráficamente para ver el cambio en la varianza y ver como efectivamente el modelo estaba sobreajustándose:
graphicx subcaption

Nombres duplicados



Nombres únicos



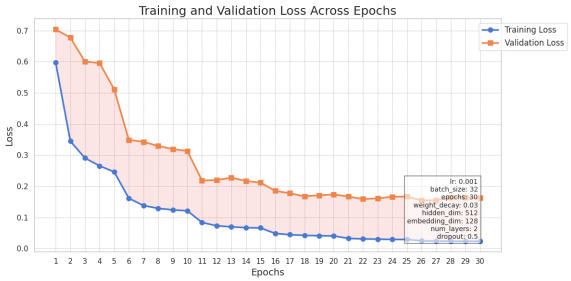
6.6 Train full augmented y dev con basic augment

A modo de prueba, para este experimento generé los datasets de manera separada, el train fué aumentado sin embargo, al dev apliqué el augment básico, que consiste en splitear nombres, quitar espacios separados si son nombres múltiples etc.

```
Training samples: 359,676
Validation samples: 5,571
```

B	Ep	WD	H	E	L	D	T.L	V.L	F1
32	30	0.03	512	128	2	0.5	0.0247	0.1544	0.89

Table 6: Resultados con Train full augmented y dev con basic



6.7 Probando a simplificar el modelo

Reflexionando, consideré la opción de que, al tratar tokens a nivel de carácter, quizá el espacio de dimensiones se podría reducir drásticamente y aún así capturar relaciones complejas, en general, he utilizado 512 y 756 para hidden y 128-256 para embeddings. Al reducir el tamaño de embeddings y hidden states, el train se aceleró muchísimo, y además con resultados decentes, aunque lejos del mejor de los resultados.

```
intensity: 2
Training samples: 402069
Validation samples: 550252
```

B	Ep	WD	H	E	L	D	F1
32	15	0.03	64	32	1	0.5	0.61
32	15	0.03	64	64	1	0.5	0.61
32	15	0.03	128	32	1	0.5	0.77
32	15	0.03	128	64	1	0.5	0.77
64	15	0.03	64	32	2	0.5	0.63
64	15	0.03	64	64	2	0.5	0.64
64	15	0.03	128	32	2	0.5	0.80
64	15	0.03	128	64	2	0.5	0.82
64	15	0.03	128	128	2	0.5	0.82

Table 7: [Full log (con Hyperparameter tuner)]

He apreciado una relación entre el tamaño de hidden, embedding size y la cantidad de datos. Si aumento mucho los datos, parece ser que el modelo se beneficia de mayores tamaños para capturar patrones. Esto tiene sentido, a más complejidad, se requiere más dimensiones.

6.8 Modelo ganador, variaciones

Ya que el modelo ganador es un modelo relativamente ligero y balanceado, vamos a hacer diversas pruebas con él para ver el impacto que tiene el data augmentation.

6.8.1 Modelo ganador, CON data augmentation

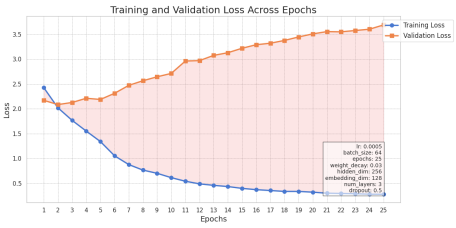
B	Ep	WD	H	E	L	D	F1
64	25	0.03	256	128	3	0.5	0.93

Table 8: Links: [Data aug. info | Full log | Gráficos]

6.8.2 Modelo ganador, SIN data augmentation

B	Ep	WD	H	E	L	D	F1
64	25	0.03	256	128	3	0.5	0.35

Table 9: Links:[Data aug. info | Full log | Gráficos]



Como se puede observar en el gráfico, existe un problema enorme de overfit, el modelo necesita de muchos más datos (regularización está bastante alta ya)

6.8.3 Modelo ganador, solo basic data augmentation

B	Ep	WD	H	E	L	D	F1
64	25	0.03	256	128	3	0.5	0.42

Table 10: Links:[Data aug. info | Full log | Gráficos]

Con tan solo activa augmentación básica - mejor, aún así llega un momento donde el validation loss vuelve a subir (epoch 8), no obstante, es más bajo que sin nada de augment.

6.9 El mejor: `LoreNexusPytorchModel v1.0`

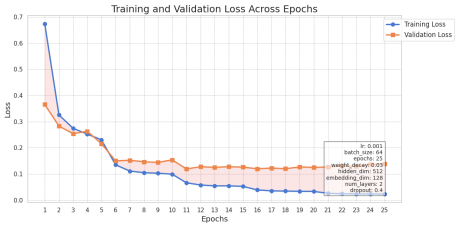
Con todo lo puesto en práctica y, haciendo uso de conocimientos adquiridos en las asignaturas de PLN, AARN y MdD, así como probando diferentes ideas y/o siguiendo mi intuición, el mejor modelo que he conseguido es el siguiente, el cual está disponible en Hugging Face.

```
  intensity: 5
Training samples: 883341
Validation samples: 110414
  swap_characters: {'enabled': True, 'pct_words_to_swap': 0.6,
  'transformations_per_example': {'min': 1, 'max': 3}}
 insert_characters: {'enabled': True, 'pct_words_to_swap': 0.6,
  'transformations_per_example': {'min': 1, 'max': 3}}
 delete_characters: {'enabled': True, 'pct_words_to_swap': 0.6,
  'transformations_per_example': {'min': 1, 'max': 3}}
 duplicate_characters: {'enabled': True, 'pct_words_to_swap': 0.6,
  'transformations_per_example': 2}
  split_names: {'enabled': True, 'join_parts': True}
  label_exclusion: {'enabled': False, 'excluded_labels': []}
  internal_swap: {'enabled': True, 'swap_probability': 0.7}
```

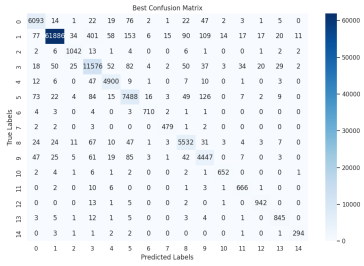
B	Ep	WD	H	E	L	D	F1
64	25	0.03	512	128	2	0.5	0.96

Table 11: Links:[Data aug. info | Full log | Gráficos]

Rendimiento



Confusion Matrix



7 CLI

La aplicación puede hacer uso de cualquier implementación de modelo que se le pase, tan sólo ha de implementar la clase de `LoreNexusWrapper` y, para seleccionar un checkpoint en concreto se puede hacer a través del `config.yaml`. Veamos brevemente la aplicación y unas predicciones hechas con ella:

Nombre derivado	Ground Truth	Etiqueta	Predicción
Lorecraft	Lovecraft	Historical	Historical (0.87), Game of Thrones (0.11)
Harry Potta	Harry Potter	Harry Potter	Harry Potter (1.00)
Tattuinne	Tatooine	Star Wars	Star Wars (0.99)
Bhoromirian	Boromir	Tolkien	Tolkien (0.94), Historical (0.03)
Dragomir	Inventado	Tolkien	Tolkien (1.00)
Gandalfuck	Gandalf + Fuck	Tolkien, Offensive	Tolkien (0.74), Offensive (0.25)
Hitler	Hitler	Historical, Offensive	Historical (0.89), Offensive (0.09)
Hittler	Hitler	Historical, Offensive	Offensive (0.89), Historical (0.10)
Riverssong	River Song	Doctor Who	Doctor Who (1.00)
Ghaladirel	Galadriel	Tolkien	Tolkien (1.00)
Orhootxiimarru	Orochimaru	Naruto	Naruto (1.00)
Futhermucker	-	Offensive	Offensive (1.00)
Utwatxd	-	Offensive	Offensive (1.00)

8 References

SPARQL: https://www.wikidata.org/wiki/Wikidata:SPARQL_tutorial
spaCy Transformer Models: https://spacy.io/models/en
GodChecker API: https://www.godchecker.com/api/
Surge AI Profanity Dataset: https://github.com/surge-ai/profanity
Flair Framework: https://github.com/flairNLP/flair
LoreNexus Repository: https://github.com/geru-scotland/lore-nexus