Asignación automática de dominios a definiciones

Josu Barrutia
Estudiante / Facultad de Informática EHU
jbarrutia006@ikasle.ehu.eus

1 Introducción

La asignación de dominios a definiciones es un problema de clasificación multiclase de texto. En este caso, se trata de entrenar varios modelos de lenguaje preentrenados para la asignación automática de dominios a conceptos de Wordnet. Concretamente, se utiliza la glosa y los posteriores ejemplos de un synset para clasificarlo en un conjunto de etiquetas llamado BabelDomains (Camacho-Collados and Navigli, 2017). Se llevará a cabo una búsqueda de hiperparámetros para conseguir realizar un fine-tunning de los modelos para lograr el mayor desempeño en la tarea sobre el conjunto de test. Se compararán varios modelos entrenados en esta tarea concreta con un modelo zero-shot Sainz and Rigau (2021) y con un modelo que haya sido entrenado con muy pocas instancias de la tarea en cuestión (few-shot).

2 Datos

Partimos de dos ficheros (wordnet_dataset_gold.txt y babeldomains_wordnet.txt) que contiene un gran número de synsets de wordNet y su respectivo dominio anotado. El fichero wordnet_dataset_gold.txt será usado tan solo para test, mientras que el fichero babeldomains_wordnet.txt será usado tanto para train, como para development, éste último fichero incluye unos índices de fiabilidad en la asignación de dominios a cada synset y, para determinados synsets se han clasificado en varios dominios distintos. Para arreglar estos problemas y limpiar los datos, nos quedaremos únicamente con el primer dominio con el que se clasifica el synset en cuestión y aplicaremos un filtro para quedarnos solo con las tuplas (synset,dominio) que presenten una fiabilidad del 100%.

Tras arreglar unos pequeños fallos en el fichero, podemos aplicar la siguiente expresión de awk para quedarnos únicamente con las instancias que verifiquen las condiciones anteriores:

```
awk –F'\t' '$3 == 1 {print $1 "\t" $2}' \
babeldomains_wordnet.txt > babeldomains_wordnet.txt
```

Tenemos que eliminar los datos del fichero babeldomains_wordnet.txt que se repitan en wordnet_dataset_gold.txt, pues la partición de test tiene que tener instancias nunca vistas por el modelo. Para ello, aplicamos la siguiente expresión de awk:

```
awk 'NR==FNR{a[$1]; next} !($1 in a)' \
wordnet_dataset_gold.txt babeldomains_wordnet.txt \
> babeldomains_wordnet.txt
```

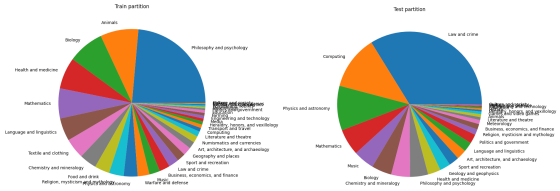
Esto nos deja con 11849 instancias para train y dev, y 1540 para test (wordnet 3.0 tiene 117659) y ya tenemos ambos ficheros limpios. Ahora tenemos que mapear cada synset con su respectiva glosa. Para ello, vamos a utilizar los ficheros de wordnet. En primer lugar, concatenamos el fichero de verbos, adjetivos, adverbios y sustantivos en un único fichero. A continuación, nos quedaremos con el PoS+offset (clave única) y con su respectiva glosa. Los ficheros descargados tiene un Part of Speech que no incluye el fichero babeldomains_wordnet.txt, pues está clasificando algunos adjetivos como *satellite adjetives* con la marca "s", esto hace que algunos synsets no casen. Así pues, vamos a reemplazar todos los synsets "s" por "a".

```
cat data.adj data.adv data.noun data.verb > wordnet.data
awk –v OFS="\t" \
'{print $3$1,substr($0,index($0,"|")+2)}'
wordnet.data > wordnet.data
sed 's/^s/a/' wordnet.data > wordnet.data
```

Y por último mapeamos cada synset con su glosa, para conseguir train.tsv y test.tsv, dos ficheros que contienen las tuplas (glosa, dominio):

```
awk –v OFS="\t" 'NR==FNR{map[$1]= \
substr($0,length($1)+2); next} {$1=($1 in map)? \
map[$1] : $1; print}' wordnet.data \
babeldomains_wordnet.txt > train.tsv
awk –v OFS="\t" 'NR==FNR{map[$1]= \
substr($0,length($1)+2); next} \
{$1=($1 in map)? map[$1] : $1; print}' \
wordnet.data wordnet_dataset_gold.txt > test.tsv
```

Una vez empezamos a cargar los datos con pandas, nos damos cuenta del mal balanceamiento que tienen las clases tanto en train como en test, ver Figura 1



(a) Distribución clases en partición entrenamiento

(b) Distribución clases en partición test

Figure 1: Comparación distribución de etiquetas de los datos

3 Sistema

La mejor opción para realizar esta tarea de clasificación multiclase de texto es la arquitectura transformers. Seleccionaremos tres arquitecturas con sus respectivos checkpoints para el tokenizer y el modelo: distilbert-base-uncased (Sanh et al., 2019), bert-base-uncased (Devlin et al., 2018) y roberta-base (Liu et al., 2019). Los modelos serán entrenados y evaluados con la gráfica NVIDIA GeForce RTX 3060 Mobile / Max-Q.

3.1 Evaluación

Para la evaluación de los modelos usaremos el conjunto gold standard con las métricas de Accuracy, Precision, Recall y F1. Estas métricas vienen definidas por:

$$Accuracy = \frac{Correcto}{Correcto + Incorrecto} \quad (1)$$

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \quad (4)$$

Para el accuracy no tenemos problemas porque los conceptos de instancias clasificadas correctamente e incorrectamente están bien definidos. Sin embargo, para las demás métricas, los conceptos de TP (verdaderos positivos), TN (verdaderos negativos), FP (falsos positivos) y FN (falsos negativos) solamente están definidos para clasificaciones binarias, es decir, para problemas en los que tuvieramos que clasificar algo como positivo o negativo.

Para solventar este problema lo que se hace es computar una matriz de confusión de 3 dimensiones, en la que cada submatriz será una matriz de confusión 2x2 en la que la clase positiva es una de todas las clases posibles y la clase negativa todas las demás, luego habrá tantas submatrices 2x2 como clases tenga nuestro problema multiclase. Esta solución realmente se usa para problemas con varias etiquetas, pero de la misma manera puede ser usado para nuestro problema.

Es importante diferenciar los problemas multiclass (el nuestro), en los que solamente tenemos una clase que puede tomar varios valores, pero cada instancia tiene asignada un único valor; de los multi-label que tienen varias clases y cada instancia puede pertenecer a varias simultáneamente y con una proporción distinta cada vez.

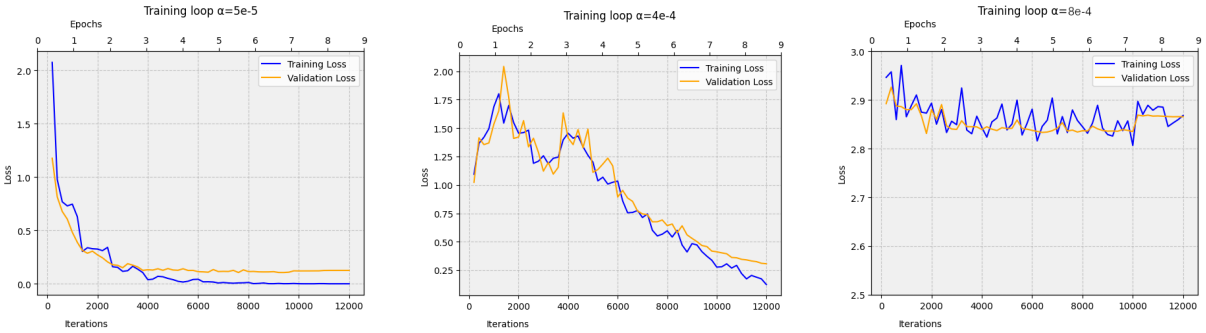
Dada esta solución, lo que se hace es hacer un average sobre esa nueva dimensión siguiendo distintas reglas: micro, macro y weighted. Lo que pasa con macro es que es muy sensible a un dataset desbalanceado y con micro ocurre que recall == precision == accuracy == f1-score. Por lo que, utilizaremos la regla weighted.

4 Experimentación

Para encontrar los hiperparámetros con los que consigamos un mayor acierto en test, tenemos que evaluar el modelo que estamos entrenando en la partición train sobre una partición disjunta que no sea la de test. Para ello, tenemos la partición de validación.

En primer lugar, queremos saber un valor adeacuado para el learning rate. Sun et al. (2019) señala que la tasa de aprendizaje es clave para evitar el "Catastrophic Forgetting" (Olvido Catastrófico), donde el conocimiento sobre el lenguaje del modelo preentrenado se borra durante el aprendizaje de nuevos conocimientos. Tasas de aprendizaje menores que 4e-4 son necesarias para evitar el olvido catastrófico. Con una tasa de aprendizaje demasiado agresiva como 8e-4 el conjunto la pérdida en el conjunto de train no converge (ver Figura 2c. Ésta puede ser la razón por la que el paper original (Devlin et al., 2018) utilizará unas tasas de aprendizaje entre 5e-5, 4e-5, 3e-5 y 2e-5.

El tamaño del batch para el dataloader de entrenamiento adecuado dependerá de la tarea. Devlin et al. (2018) utiliza un tamaño de batch de 32 aunque indican que un tamaño muy grande de las secuencias de entrada (las glosas en este caso)



(a) Entrenamiento tasa adecuada

(b) Entrenamiento tasa alta

(c) Entrenamiento con olvido por tasa alta

Figure 2: Comparación entrenamiento con distintas tasas de aprendizaje

puede hacer que el batch no quepa en la memoria de la GPU. En ese caso, habría que reducirlo a la mitad. En el nuestro, no tendremos problemas de memoria pues nuestras secuencias tienen una longitud bastante pequeña (ver Figura 3)

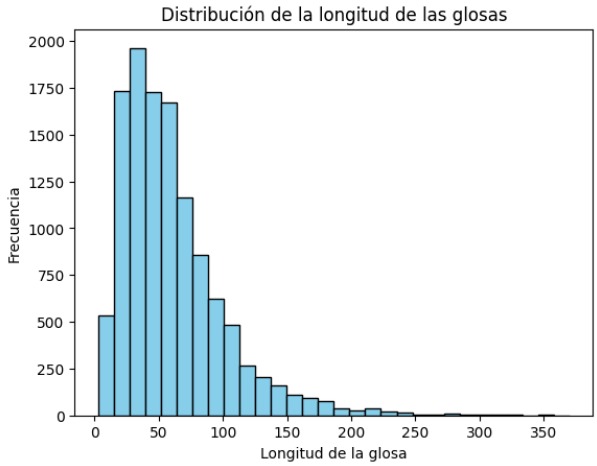


Figure 3: Distribución de la longitud de las secuencias

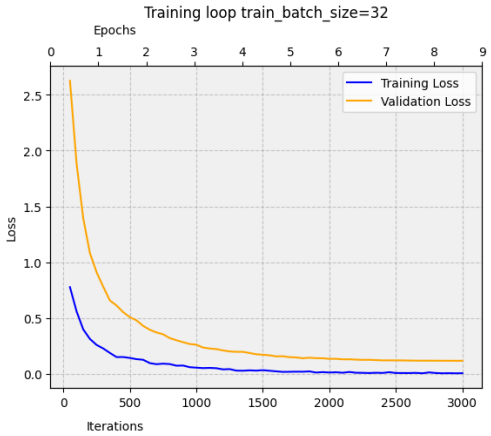
La variación del tamaño del batch no afecta en la convergencia de la pérdida del modelo en los conjuntos de entrenamiento y validación. No obstante, se puede ver como para un tamaño de batch menor la curva de la pérdida no es tan suave como en un tamaño mayor de batch (ver Figura 4). Esto ocurre porque cada iteración el descenso de gradiente y el ajuste de pesos se efectúa para menos instancias, por lo que el ajuste de parámetros es más preciso. Por lo tanto, para el fine-tunning final de los modelos utilizaremos un tamaño de batch del dataloader de entrenamiento de 32 instancias.

El número de épocas que necesita el modelo para aprender la tarea de clasificación es bastante pequeño. Devlin et al. (2018) indica que el número de épocas requeridas era muy pequeño, p.ej 3 épocas para las tareas GLUE. Al analizar la Figura 2a podemos ver como, a partir de la época 4, la pérdida en la partición de validación no disminuye, e incluso aumenta, mientras que la pérdida en entrenamiento mejora levemente. Esto nos sugiere que el modelo está empezando a sobreajustar demasiado sobre los datos de entrenamiento y estamos perdiendo capacidad de generalizar. Por ello, es una buena idea implementar la idea de Early Stopping o entrenar tan solo entre 4 y 6 épocas.

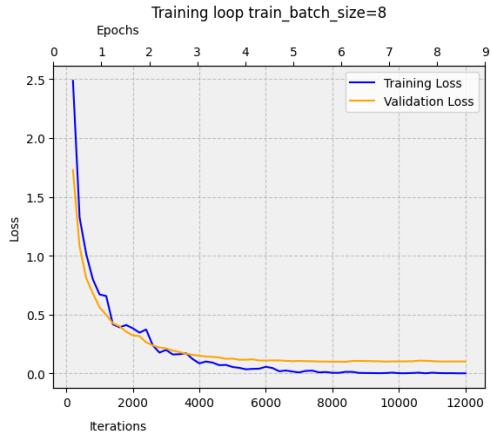
A las arquitecturas de los modelos preentrenados que se utilizan se les añade un feed-forward para convertir los 768 outputs en el número de dominios posibles que tiene nuestra tarea de clasificación. Pueden añadirse más de una capa feed-forward aunque con una es suficiente. Todos los parámetros del modelo BERT serán fine-tuneados, pero se puede congelar el modelo base y añadir más capas de clasificación sobre el modelo base BERT. En nuestro caso añadiremos una feed-forward pre-classifier de 768x768 y otra classifier 768x34, junto con un dropout de 0.2. Esto lo conseguimos con el *AutoModelForSequenceClassification* de HuggingFace.

En cuanto al optimizador, el artículo original (Devlin et al., 2018) también utilizaba Adam con weight decay. Seguir usando el mismo optimizador sería lo más lógico, aunque se pueden probar otros diferentes. Nosotros utilizaremos AdamW (PyTorch).

El planificador de tasa de aprendizaje por defecto es lineal sin warm-up, pero se pueden usar otros como el coseno y se puede establecer un warm-up de alrededor de 100 iteraciones, aunque esto puede suponer la necesidad de aumentar el número de épocas.



(a) Entrenamiento con tamaño 32



(b) Entrenamiento con tamaño 8

Figure 4: Comparación entrenamiento con distintas tamaños de batch

4.1 DistilBERT

DistilBERT (Sanh et al., 2019) es un modelo de lenguaje basado en la arquitectura BERT. La idea detrás de DistilBERT es reducir el tamaño y la complejidad de BERT mientras se mantiene un rendimiento similar en tareas de procesamiento del lenguaje natural.

Para entrenarlo se han probado varios hiperparámetros y hemos medido su rendimiento en la partición de validación.

Tasa Aprendizaje	Planificador	Épocas	Accuracy validación
2e-5	Lineal sin Warm-Up	4	0.9352
2e-5	Lineal sin Warm-Up	10	0.9774
4e-4	Lineal sin Warm-Up	10	0.9335
5e-5	Lineal sin Warm-Up	4	0.9602
5e-5	Lineal sin Warm-Up	6	0.9723
5e-5	Lineal Warm-Up 100	4	0.9628
5e-5	Coseno Warm-Up 100	4	0.9606
5e-5	Coseno Warm-Up 100	6	0.9746

Table 1: Búsqueda hiperparámetros

Tenemos que hacer un balance entre el número de épocas y el accuracy que obtenemos, algo parecido a un EarlyStopping, pues no nos interesa un modelo que haya sido entrenado por muchas épocas por el sobreajuste que pueda tener. Teniendo esto en cuenta, podemos ver como el planificador del coseno con un warm-up de 100 iteraciones ofrece un muy buen rendimiento. Así pues, nos quedaremos con el modelo entrenado durante 6épocas y tasa de aprendizaje 5e-5 con el planificador coseno y lo evaluaremos en la partición de test, obteniendo los siguientes métricas de rendimiento (ver Tabla 2).

También se ha probado a entrenar el modelo de nuevo con esos mismos hiperparámetros, pero ahora sobre el dataset conjunto de entrenamiento y validación para conseguir algo más de rendimiento debido a un mayor número de instancias y variedad para entrenar. Sin embargo, no entiendo por qué, el rendimiento obtenido es menor.

	Accuracy	F1	Precision	Recall
Partición Train	0.8792	0.8765	0.8910	0.8792
Partición Train + Dev	0.8643	0.8608	0.8788	0.8643

Table 2: Rendimiento de distilBERT en Test.

4.2 BERT

BERT (Devlin et al., 2018) es un modelo basado en la arquitectura transformer preentrenado en un gran corpus de datos en inglés de manera autosupervisada. Concretamente, utilizaremos el modelo bert-base-uncased que tiene 110 millones de parámetros.

Para entrenarlo se han probado varios hiperparámetros y hemos medido su rendimiento en la partición de validación.

Tasa Aprendizaje	Planificador	Épocas	Accuracy validación
2e-5	Lineal sin Warm-Up	4	0.9369
2e-5	Lineal sin Warm-Up	10	0.9763
5e-5	Lineal sin Warm-Up	4	0.9634
5e-5	Lineal sin Warm-Up	10	0.9819
3e-5	Lineal Warm-Up 100	4	0.9583
5e-5	Coseno Warm-Up 100	4	0.9769
5e-5	Coseno Warm-Up 100	6	0.9758

Table 3: Búsqueda hiperparámetros

En este caso seleccionamos el modelo con una tasa de aprendizaje de 5e-5 planificador coseno con warmup de 100 iteraciones y 4 épocas, pues es el que mayor accuracy nos ofrece en la partición de

validación y además, al haber sido entrenado por tan solo 4 épocas, estará poco sobreajustado.

Paradójicamente, el rendimiento en la partición de test (ver Figura 4); se puede ver que ha disminuido respecto a distilbert. Esto me parece raro porque el rendimiento en la partición de validación era incluso mayor que la de distilbert, parece que en este caso, con este conjunto de hiperparámetros, nuestro modelo no ha sido capaz de generalizar del todo bien (esto también es una muestra de que a lo mejor el gold standard no sigue la misma distribución que el dataset de entrenamiento).

También se ha probado a entrenar el modelo de nuevo con esos mismos hiperparámetros, pero ahora sobre el dataset conjunto de entrenamiento y validación para conseguir algo más de rendimiento debido a un mayor número de instancias y variedad para entrenar. En este caso el rendimiento sí que ha sido mayor.

	Accuracy	F1	Precision	Recall
Partición Train	0.8526	0.8555	0.8800	0.8525
Partición Train + Dev	0.8662	0.8638	0.8822	0.8662

Table 4: Rendimiento de BERT en Test

4.3 roBERTa

RoBERTa (Liu et al., 2019) es otra variante de BERT (Robustly optimized BERT approach). Éste difiere de bert-base-uncased en su arquitectura, pues elimina la direccionalidad del modelo, y en su cantidad de parámetros, alcanzando los 125 millones, entre otras cosas.

Para entrenarlo se han probado varios hiperparámetros y hemos medido su rendimiento en la partición de validación.

Tasa Aprendizaje	Planificador	Épocas	Accuracy validación
2e-5	Lineal sin Warm-Up	4	0.9583
2e-5	Lineal sin Warm-Up	6	0.9651
5e-5	Lineal sin Warm-Up	4	0.9561
5e-5	Lineal sin Warm-Up	6	0.9645
2e-5	Cosene Warm-Up 100	4	0.9639
2e-5	Coseno Warm-Up 100	6	0.9707
5e-5	Coseno Warm-Up 100	4	0.9696
5e-5	Coseno Warm-Up 100	6	0.9702

Table 5: Búsqueda hiperparámetros

El entrenamiento de roBERTa no ha ido según lo esperado, no hemos sido capaces de ni siquiera sobreajustar el modelo a la partición de entrenamiento, para por lo menos partir de un error en train bajo. Ésto se ha visto reflejado en la validación, donde por muchas combinaciones de hiperparámetros que se probarán no se conseguía alcanzar el rendimiento de distilBERT ni BERT. Hemos seleccionado la combinación de hiperparámetros de tasa de aprendizaje 2e-5 planificador coseno con warmup de 100 iteraciones y 6 épocas.

Como era previsible, su rendimiento en la partición de test es peor que los anteriores modelos (ver Tabla 6)

No entiendo por qué roBERTa, un modelo más grande que BERT y distilBERT, tiene un rendimiento mucho peor en este problema, costándole incluso más reducir la pérdida en train que BERT y distilBERT.

También se ha probado a entrenar el modelo de nuevo con esos mismos hiperparámetros, pero ahora sobre el dataset conjunto de entrenamiento y validación para conseguir algo más de rendimiento debido a un mayor número de instancias y variedad para entrenar. Se ha conseguido aumentar ligeramente el rendimiento del modelo.

	Accuracy	F1	Precision	Recall
Partición Train	0.8467	0.8482	0.8690	0.8467
Partición Train + Dev	0.8597	0.8593	0.8766	0.8597

Table 6: Rendimiento roBERTa en Test

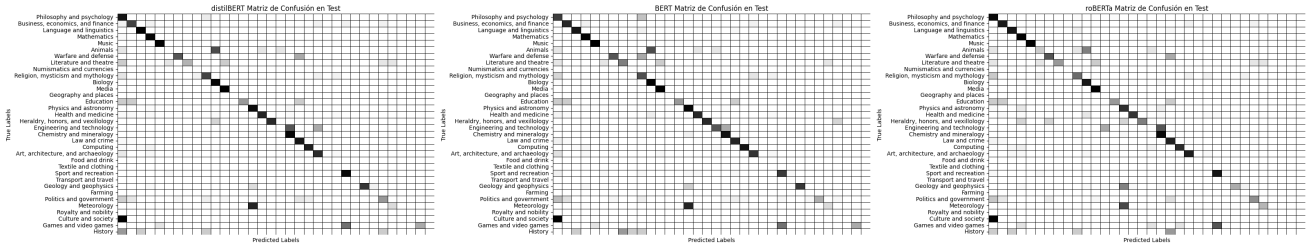
5 Few-Shot

Para simular el aprendizaje few-shot se han creado tres Datasets con 5, 10 y 20 instancias por clase, respectivamente (las clases que, por defecto, tenían menos de 5 instancias se han quedado así).

El entrenamiento de este modelo no ha sido muy costoso porque teníamos muy pocos datos con los que entrenar. Siendo así, se ha reducido el tamaño del batch de entrenamiento y aumentado el número de épocas.

Reduciendo el tamaño del batch se conseguía reducir la pérdida en entrenamiento mucho más rápido y, al parecer, el posible overfitting que esto podría causar no era relevante en test, ya que el rendimiento conseguido era mayor cuanto más pequeño fuera el tamaño del batch.

Aun y todo, al tener tan pocas instancias con las que entrenar, un número pequeño de épocas no era suficiente para conseguir hacer converger la pérdida en train, por lo que se han llegado utilizar hasta 20 épocas. Al medir el rendimiento del modelo en test no se ha apreciado ningún síntoma de overfitting, ya que con 20 épocas el rendimiento ha sido máximo (a partir de las 20 épocas, el rendimiento



(a) distilBERT (b) BERT (c) roBERTa

Figure 5: Matriz de Confusión normalizada por fila

empezaba a decaer). Los demás hiperparámetros se han mantenido como estáticos para no ajustar un modelo más que otro.

Como se puede ver el rendimiento de los modelos aumenta a la vez que aumenta el tamaño del dataset de entrenamiento (menos en el caso de 10 instancias por clase). Con tan solo 20 instancias por clase, es decir, algo menos de 360 instancias en total (pues algunas clases no llegan a tener 20 instancias), conseguimos un resultado bastante bueno casi llegando a los resultados conseguidos con el dataset completo que tiene 11849 instancias.

Número de Instancias/Clase	Accuracy	F1	Precision	Recall
5	0.7766	0.7969	0.8479	0.7766
10	0.7597	0.7756	0.8363	0.7597
20	0.8279	0.8433	0.8786	0.8279

Table 7: Rendimiento modelo Few-Shot en Test

6 Resultados y Análisis

Entre los tres modelos seleccionados distilBERT es el que mejores resultados nos ha proporcionado. Esto puede ser contraintuitivo porque de los tres es el modelo con menos parámetros y es la versión simplificada de BERT.

References

Jose Camacho-Collados and Roberto Navigli. 2017. BabelDomains: Large-scale domain labeling of lexical resources. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 223–228, Valencia, Spain. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

Oscar Sainz and German Rigau. 2021. Ask2Transformers: Zero-shot domain labelling with pretrained language models. In *Proceedings of the 11th Global Wordnet Conference*, pages 44–52, University of South Africa (UNISA). Global Wordnet Association.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *ArXiv*, abs/1910.01108.

Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. 2019. How to fine-tune BERT for text classification? *CoRR*, abs/1905.05583.

A Appendices

Appendices are material that can be read, and include lemmas, formulas, proofs, and tables that are not critical to the reading and understanding of the paper. Appendices should be **uploaded as supplementary material** when submitting the paper for review. Upon acceptance, the appendices come after the references, as shown here.

LaTeX-specific details: Use `\appendix` before any appendix section to switch the section numbering over to letters.

B Supplemental Material

Submissions may include non-readable supplementary material used in the work and described in the paper. Any accompanying software and/or data should include licenses and documentation of research review as appropriate. Supplementary material may report preprocessing decisions, model parameters, and other details necessary for the replication of the experiments reported in the paper.

Seemingly small preprocessing decisions can sometimes make a large difference in performance, so it is crucial to record such decisions to precisely characterize state-of-the-art methods.

Nonetheless, supplementary material should be supplementary (rather than central) to the paper. **Submissions that misuse the supplementary material may be rejected without review.** Supplementary material may include explanations or details of proofs or derivations that do not fit into the paper, lists of features or feature templates, sample inputs and outputs for a system, pseudo-code or source code, and data. (Source code and data should be separate uploads, rather than part of the paper).

The paper should not rely on the supplementary material: while the paper may refer to and cite the supplementary material and the supplementary material will be available to the reviewers, they will not be asked to review the supplementary material.