UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Estudio del rendimiento de BERT frente a métodos clásicos de procesamiento de lenguaje natural

Autor: Santiago González- Carvajal Centenera
Tutor: Eduardo C. Garrido Merchán
Ponente: Daniel Hernández Lobato

junio 2020

Santiago González- Carvajal Centenera
Estudio del rendimiento de BERT frente a métodos clásicos de procesamiento de lenguaje natural

Santiago González- Carvajal Centenera

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mi padre, mi madre y mis hermanos.

Resumen

Presentamos la comparación de rendimiento entre el modelo BERT (Bidirectional Encoder Representations from Transformers) y métodos más tradicionales de PLN (Procesamiento del Lenguaje Natural) en distintas tareas de clasificación de PLN. Para representar los métodos tradicionales utilizamos un primer paso de pre-procesamiento, en el que vectorizamos los datos, seguido por un segundo paso de generación del modelo. La primera fase la realizamos, generalmente, utilizando TF-IDF (Term frequency – Inverse document frequency) y la segunda, generalmente, utilizando AutoML (Automated Machine Learning). Queremos contrastar, con distintos experimentos, la hipótesis de que usar BERT como método predeterminado en problemas de PLN es mejor que usar TF-IDF junto con un algoritmo de aprendizaje automático. En total, hemos realizado cuatro experimentos, y, en todos ellos, hemos utilizado el modelo BERT y un método tradicional siguiendo los pasos que acabamos de explicar. No obstante, la lógica interna varía de un experimento a otro.

En el primero, hemos clasificado el conjunto de datos de IMDB (Internet Movie Database) en inglés, que es un problema de análisis de sentimiento. En el segundo, hemos participado en una competición de kaggle, el problema propuesto ha sido un problema de clasificación binaria de tweets en inglés. En el tercero, hemos participado en otra competición de kaggle, la cual ha consistido, en este caso, en clasificar artículos en portugués en nueve categorías distintas. En el cuarto, hemos clasificado el conjunto de datos de Chinese Hotel Reviews en chino peninsular con caracteres simplificados, que es un problema de análisis de sentimiento. Los experimentos demuestran el gran rendimiento de BERT en tareas de clasificación de PLN en diferentes idiomas e incluso con conjuntos de datos pequeños gracias a la transferencia de aprendizaje. De hecho, el rendimiento de BERT ha sido mejor en todos los experimentos que hemos realizado que el de los métodos tradicionales empleados. Y, además, la implementación de BERT ha requerido mucho menos esfuerzo que la implementación de los métodos tradicionales.

Palabras clave

Porcesamiento del lenguaje natural, PLN, BERT, TF-IDF, Machine Learning Automático, AutoML, Aprendizaje automático, ML

Abstract

We present the performance comparison between BERT (Bidirectional Encoder Representations from Transformers) model and more traditional NLP (Natural language processing) approaches in some different NLP classification tasks. In order to represent traditional approaches, we first carry out a preprocessing step, in which we vectorize all the data, followed by a model generation step. The first phase is generally carried out using TF-IDF (Term frequency – Inverse document frequency), and we generally use AutoML (Automated Machine Learning) to carry out the second phase. We want to contrast, through different experiments, the hypothesis that using BERT as a default technique in NLP problems is better than using TF-IDF followed by a machine learning algorithm. We have conducted a total of four experiments, and, in all of them, we have used BERT model and a traditional approach following the steps we have just explained. However the internal logic varies from one experiment to another.

In the first one, we have classified the English IMDB (Internet Movie Database) dataset, which is a sentiment analysis problem. In the second one, we have competed in a kaggle competition, where the problem has been a binary classification problem using tweets written in English. In the third one, we have competed in another kaggle competition, which in this case has been about classifying Portuguese articles into nine different categories. In the fourth one, we have classified the Chinese Hotel Reviews dataset, in which the reviews are written in peninsular Chinese with simplified characters, and it is a sentiment analysis problem. The conducted experiments show BERT's high performance in NLP classification tasks in different languages and even when the problem's dataset is small, thanks to transfer learning. In fact, BERT's performance has been better than the traditional methods' performance in all the experiments we have carried out. Furthermore, the implementation of BERT has taken much less effort than the implementation of the traditional methods.

Keywords

Natural language processing, NLP, BERT, TF-IDF, Automated Machine Learning, AutoML, Machine Learning, ML

Índice

Listas

Lista de figuras

Lista de tablas

1

Introducción

"Natural Language Processing" (NLP) es una rama de la "Artificial Intelligence" (AI) y de la lingüística cuyo objetivo es que los computadores entiendan el lenguaje humano (natural) [19]. Se considera que el NLP comenzó a finales de la década de 1940 con la traducción automática, aunque todavía no existía el término NLP como tal [19]. Sin embargo, de acuerdo a [19], no fue hasta los inicios de la década de 1960 cuando el trabajo influenciado por la comunidad de la AI empezó a colaborar con la comunidad de NLP, concretamente, con los sistemas BASEBALL Q-A [13].

En cuanto a metodologías, podemos diferenciar, principalmente, dos tipos de aproximaciones a los problemas de NLP:

- *Aproximaciones lingüísticas* [7] que, en general, utilizan características del texto que los expertos lingüistas consideran relevantes.
- *Aproximaciones basadas en "Machine Learning" (ML)* [22] que, de manera clásica, se basan en analizar texto etiquetado para inferir qué características del texto son relevantes para la clasificación automática.

Veremos más detalles sobre ambas aproximaciones en el capítulo 2. A continuación, vamos a ver alguno de los problemas que surgen al utilizar métodos tradicionales.

Un problema de los métodos de NLP tradicionales es el multilingüismo [6]. Podemos diseñar un conjunto de reglas para un lenguaje, pero la estructura de las frases, e incluso el alfabeto, pueden cambiar de un lenguaje a otro, conllevando como resultado la necesidad de diseñar nuevas reglas. Algunas aproximaciones como el "Universad Networking Language" [35] tratan de solucionar este problema, pero es difícil construir el recurso multilingüista. Otro problema de este tipo de aproximaciones es, que dado un lenguaje específico, la forma en la que nos expresamos varía dependiendo del documento que estemos escribiendo. Por ejemplo, la manera en la que escribimos en Twitter es muy diferente de la manera en la que escribimos un documento más formal, tal como un artículo de investigación [11].

"Bidirectional Encoder Representations from Transformers" (BERT) [10] es un modelo de NLP que surgió en 2018 capaz de conseguir resultados de estado del arte en muchas tareas de NLP [10]. Este modelo fue diseñado para pre-entrenar sobre un enorme corpus sin etiquetar, y, después, reajustar los parámetros utilizando datos etiquetados para tareas específicas de NLP [10]. Explicaremos este modelo en detalle en el capítulo 3.

En este proyecto comparamos BERT con métodos tradicionales de NLP. Concretamente, para representar los métodos tradicionales, entrenamos modelos de "Automated" ML sobre las características devueltas por el algoritmo "Term Frequency - Inverse Document Frequency" (TF-IDF) [31].

Primero, presentamos la notación utilizada en el documento. Acto seguido, veremos el estado del arte en NLP. Después, daremos la definición del proyecto. A continuación, veremos los diseños funcional y técnico del proyecto, así como la planificación del mismo. Continuaremos con los detalles de la implementación y los experimentos llevados a cabo. Y, para terminar, presentaremos las conclusiones del proyecto y el trabajo futuro.

Notación

En este apartado presentamos un resumen de la notación que utilizaremos a lo largo de el documento.

La notación que vamos a utilizar es la siguiente:

x	x es un escalar.
$\mathbf{x}$	$\mathbf{x}$ es un vector.
X	X es una matriz.
$\mathbf{x}^T$	Vector $\mathbf{x}$ transpuesto.
$\mu(\cdot)$	Media.
$\sigma(\cdot)$	Desviación típica.

Vistas la introducción y la notación que vamos a emplear en el documento, a continuación, vamos a presentar el estado del arte de NLP.

2

Estado del arte

En este capítulo vamos a ver el estado del arte de NLP en cuanto al preprocesamiento de texto, los enfoques lingüísticos y los basados en ML se refiere. Para el preprocesamiento de texto y los enfoques basados en ML seguiremos la referencia [34].

2.1. Preprocesamiento del texto

El preprocesamiento de texto es una tarea importante en NLP que consiste en transformar el texto en algo que los algoritmos de ML puedan utilizar. Esto se consigue, principalmente, mediante la eliminación de las palabras de parada, "stemming" y algoritmos de TF-IDF [37].

Empezaremos centrándonos en uno de los problemas más básicos con los que empezó NLP. Este fue, de acuerdo a [34], la clasificación de documentos. En este tipo de problemas, entraría, por ejemplo, el problema de desarrollar un algoritmo quee determine si un correo es "spam" o no.

Una de las primeras aproximaciones a este problema es utilizar "Bag of Words" (BoW) [41], que consiste en "tokenizar" los documentos, transformándolos en listas de palabras. Este tipo de aproximación, desecha el orden de las palabras y otros factores, y le da importancia, únicamente, a la multiplicidad de cada palabra. Es decir, genera un diccionario en el que a cada palabra le asocia su multiplicidad. Después, aplicaríamos un algoritmo de ML que utilizara los datos generados por BoW para clasificar los documentos. Uno de los problemas de BoW (sin contar que deja de lado las relaciones semánticas, la dimensión de los datos, etc.) son los documentos largos. Esto se debe a que este tipo de documentos elevan la cuenta de algunas palabras, pudiendo influir en la precisión del algoritmo.

Para resolver este problema, podríamos utlizar TF-IDF [31], que normaliza la frecuencia del término en el documento (*TF*) multiplicándola por el logaritmo natural de la inversa de la frecuencia de los documentos en los que aparece el término (*IDF*). Después de aplicar este algoritmo, aplicaríamos un algoritmo de ML que utilizara los datos generados por TF-IDF para clasificar los documentos. Uno de los problemas de TF-IDF (entre otros) es que, como BoW, tampoco tiene en cuenta relaciones semánticas. Veremos solucionas a esto en las próximas secciones.

2.2. Enfoques lingüísticos

Los enfoques lingüísticos se centran, en general, en aplicar una serie de reglas creadas por expertos lingüistas [19]. Este tipo de enfoque tiene la ventaja de que, a diferencia de los enfoques basados en ML, no necesita grandes cantidades de datos [16].

Un ejemplo de estos modelos sería "Valence Aware Dictionary for sEntiment Reasoning" (VADER) [16], que utiliza una combinación de método cualitativos y cuantitativos para construir una lista de características léxicas relacionadas con el análisis de sentimiento [16]. Como podemos ver en [16] este tipo de modelos no requieren un gran coste computacional, pero sí un gran coste humano. Es decir, los expertos tienen que generar el conocimiento fundamental que se le da al modelo (y esto se hace manualmente).

También, comentar, la existencia de herramientas con información lingüística como, por ejemplo, los recursos léxicos ("lexical resources"). Un ejemplo de recurso léxico sería *"WordNet"* [24], que es una gran base de datos léxica de relaciones semánticas entre palabras (está disponible en más de 200 idiomas, pero en su origen era del Inglés) [24]. Este tipo de recursos contiene información semántica que puede ser utilizada por distintos modelos.

No entraremos en más detalles con estos modelos porque los modelos más recientes (que utilizan un enfoque lingüístico) no son modelos lingüísticos puros, sino que se utilizan modelos híbridos (como podemos observar en [21] o en [8]) que son una combinación de métodos basados en ML y lingüísticos. Es decir, los expertos lingüistas proporcionan al modelo conocimientos fundamentales (en forma de reglas o similares), y partiendo de ese punto, se entrena al modelo con algoritmos de ML. Ver [8].

2.3. Enfoques basados en ML

Generalmente, este tipo de enfoques tiene una base estadística (como la inferencia estadística) [19].

Como veíamos, previamente, en 2.1, uno de los problemas de TF-IDF es que no tiene en cuenta relaciones semánticas. Con el objetivo de representar estas relaciones semánticas entre las palabras, podríamos utilizar alguna técnica de *"word embedding"*, que representa cada palabra mediante una lista de valores. Por ejemplo, *"word2vec"* [12]. Este método consiste en, dada una palabra de una frase, fijar un tamaño de ventana y emparejarla con las palabras que se encuentran antes o después a una distancia máxima del tamaño de ventana escogido. Formando de esta manera, para una misma palabra (*palabra objetivo*), varias parejas con diferentes palabras (*palabras de entrada*). Las parejas de palabras, $\langle$*palabra entrada, palabra objetivo*$\rangle$, son dadas a un algoritmo que trata de predecir la palabra objetivo a partir de la palabra de entrada. De esta manera, el algoritmo de ML aprende sobre las relaciones entre palabras. Véase también [23].

En 2014, surge un nuevo algoritmo para generar "word embedding" llamado "Global Vectors for Word Representation" (GloVe) [26]. GloVe agrega una matriz de co-ocurrencias palabra-palabra de un corpus, X. Las entradas de esta matriz de co-ocurrencias, X_{ij}, representan el número de veces que la palabra j aparece en el contexto de la palabra i. El proceso de entrenamiento se realiza sobre los datos de esta matriz (realizando unas cuantas operaciones extra, pero no entraremos en detalles). Para más detalles ver [26]. Finalmente, comentar que la distancia de dos palabras vectorizadas (como distancia entre dos vectores en el espacio vectorial) está relacionada con la relación semántica de ambas palabras. En [26] podemos ver más detalles sobre las operaciones concretas que realiza el modelo, sobre el funcionamiento del mismo y la comparativa con algunos de los modelos de [23].

Como hemos comentado, "word embedding" empezó a conseguir relacionar semánticamente las palabras. Pero, el problema principal es que no tiene en cuenta el orden de las palabras ni el contexto en el que se encuentran. Es decir, es una representación *independiente del contexto*. En ese momento, apareció "Embeddings from Language Models" (ELMo) [27]. ELMo es un "embedding" *consciente del contexto*. Modela características como la sintaxis de la palabra y cómo se utiliza en diferentes contextos. ELMo utiliza un algoritmo "Long Short-Term Memory" (LSTM) [18] bidireccional (en el sentido de que tiene en cuenta las palabras que aparecen antes y las que aparecen después de una palabra dada). LSTM es un tipo de red neuronal recurrente capaz de aprender dependencias a largo plazo [18].

A principios de 2018, ELMo logró un mejor rendimiento que los algoritmos mencionados anteriormente en la mayoría de problemas principales de NLP del momento (análisis de sentimiento, respuesta a preguntas, etc.). Logrando resultados de estado del arte.

Otra manera de manejar las dependencias a largo plazo es utilizar un *"Transformer"* [36], que apareció en 2017. El "Transformer" se basa en la conexión de un codificador y un decodificador mediante un mecanismo de atención, prescindiendo de convoluciones y recurrencias. Dando lugar a un modelo más paralelizable y con un tiempo de entrenamiento menor. Es muy bueno en problemas de secuencia a secuencia ("seq2seq"), como, por ejemplo, tareas de traducción. Este modelo puede hacer frente a problemas de dependencias a largo plazo mejor que el LSTM. Para más información ver [36].

En Junio de 2018, *OpenAI* introdujo el *OpenAI Transformer*, también conocido como *OpenAI* "Generative Pretrained Transformer" (GPT) [28]. El GPT superó el estado del arte en 9 de las 12 tareas que estudiaron.

El GPT es capaz de ajustar el modelo pre-entrenado utilizando el decodificador, que es una buena herramienta de modelado para predecir la palabra siguiente.

Ese mismo año, en Mayo, aparece el "Universal Language Model Fine-tunning for Text Classification" (ULMFiT) [15], que es un método que utiliza transferencia de aprendizaje ("transfer learning"), y que puede ser aplicado a cualquier tarea de NLP. Este método introduce técnicas clave para ajustar un modelo de lenguaje.

La transferencia de aprendizaje consiste en aplicar la idea que se aprende en una tarea específica a otras tareas. Para más detalles ver [25]. El proceso introducido por ULMFiT, es lo que le permitió utilizar la transferencia de apredizaje para ajustarse a distintas tareas de NLP. ULMFiT puede entrenarse sobre un conjunto de texto enorme como Wikipedia para predecir la siguiente palabra. Después, se utiliza la transferencia de aprendizaje para actualizar el modelo para entrenar sobre un conjunto de datos específico (que puede ser muy pequeño). El tener el modelo pre-entrenado (con el conjunto de datos enorme), mejorará la precisión del modelo sobre el conjunto de datos específico.

En Octubre de ese mismo año, apareció BERT [10]. BERT destrozó los récords previos en múltiples conjuntos de datos de NLP, como por ejemplo, en el "Standford Question Answering Dataset". El cual consiste, básicamente, en responder preguntas de acuerdo a un texto de Wikipedia (comprensión lectora).

BERT utiliza la transferencia de aprendizaje. El primer paso es el pre-entrenamiento, que utiliza un enorme corpus (como Wikipedia) para entrenar el modelo del lenguaje. La entrada son los datos pero hay que preprocesarlos antes para generar pares de frases con partes ocultas ("masked"). Es decir, una versión de la frase con ruido. BERT utiliza prácticamente la misma arquitectura para el pre-entrenamiento y para el reajuste. También utiliza la estructura de un "Transformer", concretamente, la parte del codificador del "Transformer".

Para hacerlo bidireccional, BERT no utiliza un modelo "backward" como LSTM o ELMo, sino que añade el *ocultamiento de palabras* ("word masking"). Esta técnica consiste en ocultar un porcentaje de las palabras de la entrada (en [10] este porcentaje es un 15 %) sustituyéndolas por el token "[MASK]". Después, pasan la frase por la arquitectura y tratan de predecir únicamente las palabras ocultas. Esta aproximación es considerada como "deep bidirectional Transformer encoder". BERT es capaz de captar la co-ocurrencia. Daremos más detalles en el capítulo 3.

Con el éxito de BERT aparecieron mejoras como ALBERT, RoBERT, TinyBERT, DistilBERT y SpanBERT. Por ejemplo, SpanBERT [17] introduce el cambio de ocultar palabras consecutivas (entre otros).

En Febrero de 2019, *OpenAI* introdujo GPT-2 [29], que es capaz de generar texto coherente. Este modelo introduce algunas modificaciones respecto a GPT (pero no entraremos en detalles). El objetivo de este modelo es predecir la palabra siguiente dadas todas las palabras previas en un texto.

Un mes antes, en Enero de 2019, había aparecido la arquitectura "Transformer-XL" [9] que resolvía las deficiencias de BERT y de los "Transformers". Las deficiencias de los "Transformers" eran las dependencias a largo plazo y la fragmentación del contexto. Si se desea más información ver [9].

En Junio de 2019, apareció *XLNet* [39], que mejoraba el rendimiento de BERT en 20 tareas de NLP (no entraremos en detalles).

En Marzo de 2019, "Enhanced Representation through Knowledge Integration" (ERNIE) [33], que utiliza un "Transformer" multi-capa como BERT pero añade distintos niveles a la ocultación de la pala-

bras, mejoró el rendimiento de BERT en las 5 tareas de procesamiento del lenguaje en Chino que los autores estudiaron.

En Julio de 2019, ERNIE 2.0 mejoró el rendimiento de BERT y *XLNet* en 16 tareas de NLP (incluyendo varias tareas de lenguaje en Chino).

En Noviembre de 2019, fue presentado el "Compressive Transformer" [30] para tareas de memoria a largo a plazo (no entraremos en más detalles).

En Febrero de 2020, Microsoft anunció "Turing Natural Language Generation" (T-NLG) [32], que tiene más de 17 billones de parámetros (el doble que el *Megatron* de *Nvidia* que tiene 8.3 billones de parámetros y utiliza GPT-2, que era el que más parámetros tenía hasta el momento). Para ver la magnitud del modelo, observar que en 2018, el GPT-2 de *OpenAI* era considerado el que más parámetros tenía con 1.5 billones.

2.4. Conclusiones

En los últimos años, los enfoques basados en ML han ganado mucha fuerza. Estos modelos requieren muchos datos y un gran coste computacional. No obstante, en la actualidad, esto no supone un gran problema ya que, como hemos visto previamente, han surgido técnicas, como la transferencia de aprendizaje, que ayudan a solucionar el problema de los datos. Y, gracias a los continuos avances tecnólogicos, la capacidad computacional es cada vez mayor.

Los modelos basados en ML obtienen resultados de estado del arte para la mayoría de problemas de NLP actuales, mientras que los lingüísticos, en general, parecen no estár a la altura. De hecho, como hemos visto antes, las aproximaciones lingüísticas que se utilizan en la actualidad son aproximaciones híbridas. Además, es probable que los resultados de los modelos basados en ML sigan mejorando conforme se produzcan avances técnológicos.

Con esto en mente, y como explicamos en el próximo capítulo, hemos decidido comparar BERT con enfoques más tradicionales de NLP, con el objetivo de comprobar la evolución de los algoritmos de NLP en los últimos años.

3

Definición del proyecto

En este capítulo vamos a definir el alcance de nuestro proyecto. Para ello, vamos a presentar una serie de listas. La primera, con los objetivos que nos hemos propuesto realizar. La segunda, con las hipótesis que queremos contrastar. La tercera, con las asunciones que hemos realizado. Y, la cuarta, con las restricciones por las que se ha visto afectada la realización del proyecto.

Como hemos dicho en el párrafo anterior, vamos a empezar definiendo una lista de objetivos que queremos realizar.

O-1.– Comparar el rendimiento de BERT con algoritmos más tradicionales de NLP en Inglés.

O-2.– Comparar el rendimiento de BERT con algoritmos más tradicionales de NLP en otros idiomas con incluso distintos alfabetos.

O-3.– Medir el rendimiento de BERT en alguna clase de competición.

O-4.– Demostrar la efectividad de BERT sobre conjuntos de datos pequeños gracias a la transferencia de aprendizaje.

O-5.– Optimización Bayesiana de algunos parámetros de BERT.

Vistos los objetivos principales que queremos realizar, pasamos a presentar una lista con las hipótesis que queremos contrastar de cara a la consecución de los objetivos anteriores.

H-1.– BERT tiene un mejor rendimiento que algoritmos más tradicionales como por ejemplo un algoritmo que utilice TF-IDF seguido de un clasificador.

H-2.– BERT es un modelo capaz de obtener buenos resultados en cualquier idioma.

H-3.– BERT es fácil de implementar comparado con los modelos tradicionales.

H-4.– BERT es capaz de conseguir buenos resultados incluso sobre conjuntos de datos pequeños.

Para cumplir los objetivos anteriores y contrastar las hipótesis mencionadas, hemos realizado una serie de asunciones.

A-1.– El modelo BERT utilizado ha sido pre-entrenado correctamente.

A-2.– El modelo BERT utilizado está implementado siguiendo la lógica explicada en [10].

A-3.– Los paquetes de “Auto Machine Learning” son fiables a la hora de buscar los modelos que mejor se ajustan a los datos de entrenamiento.

Finalmente, y desde un punto de vista técnico, presentamos una lista de las restricciones a las que ha estado sujeto el desarrollo del proyecto.

R-1.– El equipo disponible para realizar los experimentos ha sido un ordenador personal.

R-2.– La cantidad de tiempo disponible ha sido de 232 días.

R-3.– La situación vivida con el COVID-19 ha dificultado todas las labores.

Como hemos comentado, estas listas representan el alcance de nuestro proyecto. A continuación, pasamos a describir el diseño del mismo.

4

Diseño del proyecto

En este capítulo vamos a presentar el diseño del proyecto desde tres perspectivas distintas. Concretamente, vamos a presentar el diseño funcional (o del modelo), el diseño técnico y la planificación del proyecto.

4.1. Diseño del modelo

En esta sección vamos a explicar el funcionamiento de los algoritmos empleados en la realización de los experimentos que veremos en el capítulo 6. Empezaremos explicando TF-IDF, que ha sido el utilizado como representación de los métodos más tradicionales (seguido por un modelo), y terminaremos explicando el funcionamiento del modelo BERT.

4.1.1. Term Frequency - Inverse Document Frequency (TF-IDF)

Vamos a empezar explicando el funcionamiento de TF-IDF. Para ello, seguiremos la explicación de [31].

Para empezar, comentar que, como hemos visto en el capítulo 2, TF-IDF representa "Term Frequency - Inverse Document Frequency". A continuación, vamos a ver en qué consiste exactamente TF-IDF.

Asumiendo que tenememos una colección de N documentos, y que el término t_i aparece en n_i de ellos. Entonces, la frecuencia inversa de documento se puede calcular como:

$$idf(t_i) = log\frac{N}{n_i} \tag{4.1}$$

La medida original era una aproximación entera de esta fórmula, y el logaritmo era en base 2 en vez de en base e. No obstante, la base del logaritmo no es importante en general, y la ecuación (4.1) es la más citada para *IDF*. Para más información ver la fuente original [31].

Por otro lado, dado un término t_i, denotamos por tf_i la frecuencia de t_i en el documento bajo

consideración [31].

Finalmente, TF-IDF se define para un término t_i en un documento dado como el producto de *TF* por *IDF*. Es decir,

$$tfidf(t_i) = tf_i \cdot idf(t_i)$$

Con el factor $idf(t_i)$ conseguimos que los documentos largos no eleven la cuenta de las palabras, y de esa manera puedan influir en la precisión del algoritmo, como bien señalábamos en el capítulo 2. Visto el funcionamiento de TF-IDF, pasamos a explicar el funcionamiento del modelo BERT.

4.1.2. Bidirectional Encoder Representations from Transformers (BERT)

En esta sección vamos a centrarnos en describir de la manera más detallada posible la arquitectura del modelo BERT y daremos algunas nociones básicas de su funcionamiento. Para ello, seguiremos las referencias [10, 36].

A grandes rasgos, y como se explica en [10], BERT tiene dos pasos en su esquema de funcionamiento: *pre-entrenamiento* y *reajuste* ("fine-tuning"). El *pre-entrenamiento* consiste en entrenar al modelo sobre grandes corpus sin etiquetar. El *reajuste* consiste en inicializar el modelo con los parámetros del pre-entrenamiento, y reajustar todos los parámetros utilizando conjuntos de datos etiquetados correspondientes a tareas específicas. Cada una de estas tareas específicas tiene modelos reajustados separados, aunque todos ellos son inicializados con los mismos parámetros del pre-entrenamiento. Una de las características distintivas de BERT es que tiene una arquitectura unificada para las distintas tareas específicas. La diferencia entre la arquitectura de pre-entrenamiento y la de la tarea específica es mínima.

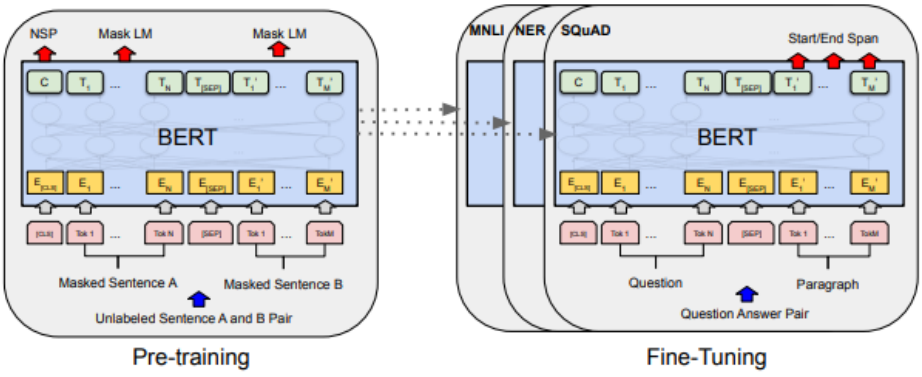

Figura 4.1: Procedimientos de entrenamiento y reajuste. Fuente [10].

Arquitectura del modelo

La arquitectura de BERT es la de un "Transformer" codificador bidireccional multicapa [10]. Y, dado que en [10] refieren a [36] para ver los detalles de la arquitectura del modelo diciendo que su implementación es casi idéntica, vamos a empezar explicando en qué consiste esta arquitectura siguiendo la referencia original [36].

Las arquitecturas codificador-decodificador consisten en que el codificador mapea una secuencia de representaciones simbólicas $\mathbf{x} = (x_1, ..., x_n)$ a una secuencia de representaciones continuas $\mathbf{z} = (z_1, ..., z_n)$. Dado $\mathbf{z}$, el decodificador genera la secuencia salida $\mathbf{y} = (y_1, ..., y_n)$ de símbolos (de uno en uno). En cada paso, el modelo es auto-regresivo, lo que quiere decir que utiliza los símbolos generados en los pasos previos como entrada extra para calcular el símbolo siguiente. Los "Transformer" siguen esta arquitectura general, utilizando capas totalmente conectadas ("fully connected layers") con mecanismos de auto-atención ("self-attention") y aplicados a cada posición ("position-wise"), tanto para el codificador como para el decodificador. Totalmente conectada quiere decir que todas las neuronas de una capa dada están conectadas a todas las neuronas de la capa siguiente.

En este caso, dado que nuestro objetivo es explicar el funcionamiento de BERT, nos centraremos en el codificador y en los mecanismos de auto-atención y posicionales.

Codificador

El codificador está compuesto de $N = 6$ capas idénticas. Cada una de estas capas, tiene 2 sub-capas. La primera es un mecanismo de auto-atención multi-cabezal ("multi-head self-attention"). La segunda es una red neuronal prealimentada totalmente conectada aplicada a cada posición de manera separada pero idéntica ("position-wise fully connected feed-forward network"). Prealimentada quiere decir que las conexiones entre las neuronas no forman ciclos (bucles). Emplea una conexión residual [14] alrededor de cada una de las sub-capas, seguida de una normalización de capa [5]. Es decir, la salida de cada sub-capa es $LayerNorm(x + Sublayer(x))$, donde $Sublayer(x)$ es la función implementada por la sub-capa en cuestión. Para facilitar las conexiones residuales, todas las sub-capas del modelo (y las capas de "embedding") generan salidas de dimensión $d_{model} = 512$.

Veamos ahora en qué consiste exactamente la normalización de capa. Para ello vamos a seguir la referencia [5]. La normalización de capa consiste en normalizar los datos de entrada de la siguiente capa.

Considerando la l-ésima capa oculta de una red neuronal prealimentada, y la i-ésima unidad oculta ("hidden unit"). La normalización de capa se calcula de la siguiente forma:

$$\bar{a}_i^l = \frac{g_i^l}{\sigma^l}(a_i^l - \mu^l),$$

donde

- $\bar{a}_i^l$ es la suma normalizada de las entradas a la i-ésima unidad oculta de la l-ésima capa.
- $a_i^l = \mathbf{w}_i^{l^T}\mathbf{h}^l$ es la suma de las entradas a la i-ésima unidad oculta de la l-ésima capa, donde
  - w_i^l son los pesos que llegan a la i-ésima unidad oculta de la l-ésima capa.
  - $\mathbf{h}^l$ es el vector formado por los $h_i^{l+1} = f(a_i^l + b_i^l)$, donde $f(\cdot)$ es una función no-lineal, y b_i^l es el parámtro escalar del sesgo (“bias”).
- g_i es un parámetro de ganancia. Para más detalles ver [5].
- $\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l$ es una estimación de la media de los a_i^l que tiene el mismo valor para todas las unidades ocultas de la l-ésima capa. H es el número de unidades ocultas en la capa.
- $\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2}$ es una estimación de la desviación típica tiene el mismo valor para todas las unidades ocultas de la l-ésima capa. H es el número de unidades ocultas en la capa.

Por su parte, la conexión residual consiste realizar una conexión saltándose algunas capas. Para más detalles ver [14]. Podemos ver la arquitectura del codificador en la siguiente figura.

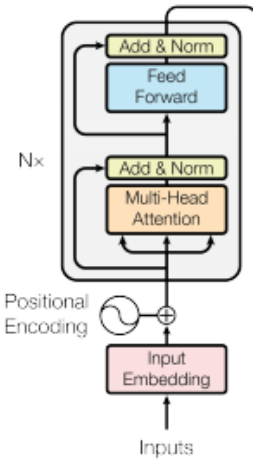

Figura 4.2: Codificador de un “Transformer”. Fuente [36]

Auto-atención multi-cabezal

Como indica [36], una función de atención consiste en mapear una consulta (“query”) y un conjunto de claves-valores (“keys-values”) a una salida, donde todos los componentes anteriores son vectores. La salida es la suma ponderada de los valores, donde el peso asignado a cada valor es calculado por una función de compatibilidad de la consulta con la clave correspondiente [36].

Para definir la atención multi-cabezal, es necesario definir primero la atención “scaled dot-product”. Esta se define como sigue:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

donde,

- $softmax : \mathbb{R}^k \to [0,1]^K$. Definida como $softmax(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{z_k}}$ para $j = 1, ..., K$.
- d_k es la dimensión de las consultas y las claves.
- Q es la matriz de consultas.
- K es la matriz de claves.
- V es la matriz de valores.

A partir de la definición anterior, definimos la atención multi-cabezal ("multi-head") como:

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O$$

donde $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$. Este tipo de atención consiste en proyectar las consultas, claves y valores h veces, con diferentes proyecciones lineales aprendidas, a las dimensiones d_k, d_k y d_v (dimensión de los valores), respectivamente. En cada una de estas proyecciones de las consultas, claves y valores, aplicamos la función de atención en paralelo, obteniendo valores d_v-dimensionales. Finalmente, estos son concatenados y proyectados de nuevo dando lugar a los valores finales [36].

Las proyecciones son matrices de parámetros con las siguientes dimensiones $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_k}$ y $W^O \in \mathbb{R}^{hd_v \times d_{model}}$.

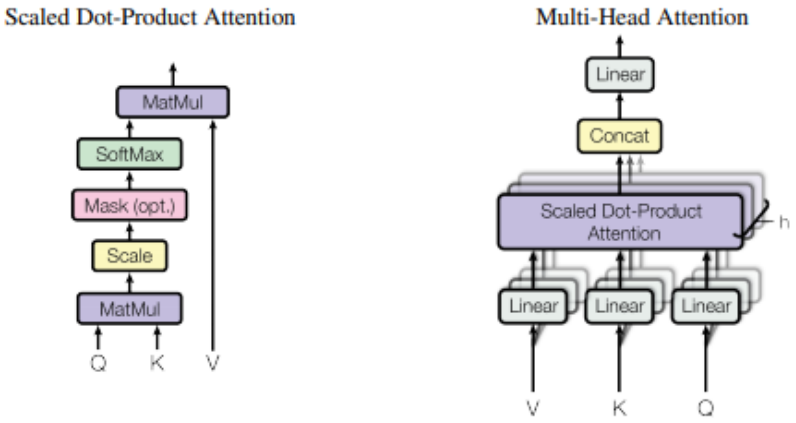

Figura 4.3: Mecanismos de atención de un "Transformer". A la izquierda la atención "scaled dot-product". A la derecha la atención "multi-head". Fuente [36]

La parte de auto de auto-atención, solo indica que las consultas, claves y valores vienen del mismo sitio. En este caso, este sitio es la salida de la capa previa del codificador. Cada posición del codificador puede "atender" a todas las posiciones de la capa previa.

Redes prealimentadas posicionales

Ahora veremos el funcionamiento de las redes prealimentadas posicionales de acuerdo a [36]. El codificador (y el decodificador) contienen una red prealimentada totalmente conectada, que es aplicada

a cada posición de manera separada e idéntica. Esto consiste en dos transformaciones lineales con activación "Rectified Linear Unit" (ReLU) entre ambas [36].

$$FFN(x) = max(0,\ xW_1 + b_1)W_2 + b_2.$$

Las transformaciones lineales son las mismas en las diferentes posiciones, pero los parámetros que utilizan varían de una capa a otra.

Finalmente, si denotamos por L el número de capas (es decir, el número de bloques de "Transformer"), el tamaño oculto como H y el número de cabezales de auto-atención como A, los dos tamaños del modelo BERT utilizados en el estudio original [10] son: **BERT**$_{BASE}$ (L=12, H=768, A=12, parámetros totales=110M) y **BERT**$_{LARGE}$ (L=24, H=1024, A=16, parámetros totales=340M). Con esto, nos podemos hacer una idea del tamaño de este modelo.

Funcionamiento de BERT

A continuación, explicaremos brevemente el funcionamiento de BERT de acuerdo a [10].

BERT representa una sola frase o un pareja de frases (por ejemplo, ⟨pregunta, respuesta⟩) como una secuencia de "tokens" de acuerdo a las siguientes características:

- Utiliza "WordPiece embeddings" [38].
- El primer "token" de la secuencia es "[CLS]".
- Cuando hay dos frases, en la secuencia están separadas por el "token" "[SEP]", y a cada "token" se le añade un "embedding" para indicar a qué frase pertenece.
- Para un "token" dado, su representación para la entrada se computa sumando los correspondientes "embeddings" del "token", segmento y posición.

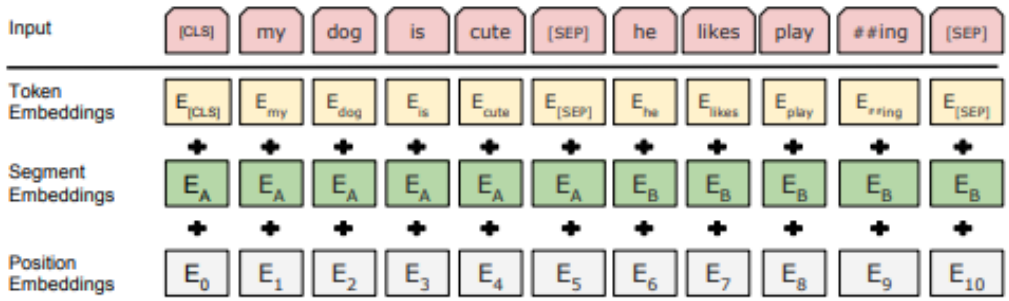

Figura 4.4: Representación de la entrada. Fuente [10].

Pre-entrenamiento

A continuación vamos a explicar en qué consiste el pre-entrenamiento siguiendo [10]. El pre-entrenamiento se divide en dos tareas:

- "*Masked LM*": dado que el modelo es bidireccional, y no queremos que cada palabra sea capaz de "verse a sí misma", lo que se hace es ocultar un porcentaje aleatorio de "tokens" de la entrada para luego tratar de predecirlos. Normalmente, se utiliza el 15 %. Para ello se sustituyen los "tokens" seleccionados: un 80 % de los casos por el "token" "[MASK]", un 10 % por otro "token" aleatorio y un 10 % se deja el "token" que estaba.
- *Predicción de la siguiente frase (NSP)*: dadas dos frases `A` y `B`, para cada ejemplo de pre-entrenamiento, en un 50 % de los casos `B` es la frase que sigue a `A` y se etiqueta como `IsNext`, en el otro 50 % `B` es una frase aleatoria del corpus y se etiqueta como `NotNext`.

Para el pre-entrenamiento del modelo BERT en inglés, se utilizan "BooksCorpus" (800M de palabras) y "Wikipedia" en inglés (2500M de palabras) como conjuntos de datos.

Reajuste

El reajuste ("fine-tuning") es sencillo gracias al mecanismo de auto-atención del "Transformer". Para cada tarea, simplemente se ponen las entradas y salidas correspondientes y se ajustan los parámetros de principio a fin. El coste del reajuste, comparado con el de pre-entrenamiento, es muy bajo. Para más información ver la fuente original [10].

Visto el diseño del modelo, pasamos al diseño técnico en el que describimos el funcionamiento de los experimentos realizados.

4.2. Diseño técnico

En esta sección vamos a utilizar diagramas de flujos de datos (DFDs) para describir el funcionamiento de los experimentos que hemos realizado.

Vamos a empezar analizando el diagrama de flujo de datos de nivel 0 (o diagrama de contexto) que es común a todos los modelos clasificadores utilizados.

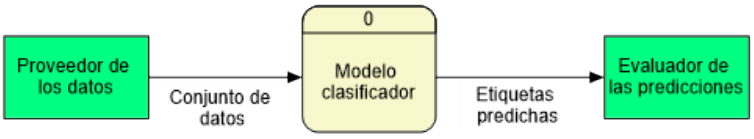

Figura 4.5: Diagrama de flujo de datos (nivel 0).

Como podemos ver, este diagrama tiene dos entidades externas que se corresponden a la que nos proporciona los datos y la que se encarga de evaluar nuestras predicciones. Entre esta dos entidades externas, se encuentra el proceso 0, que corresponde al modelo clasificador que hemos utilizado en el experimento en cuestión, cuya lógica interna varía de un experimento a otro. Este proceso es el encargado de recibir un conjunto de datos y generar predicciones sobre los mismos.

A continuación, vamos a ver con más detalle qué procesos tienen lugar dentro del proceso 0. Para

ello, vamos a analizar el diagrama de flujo de datos de nivel 1 que también es común a todos los modelos clasificadores que hemos utilizado.

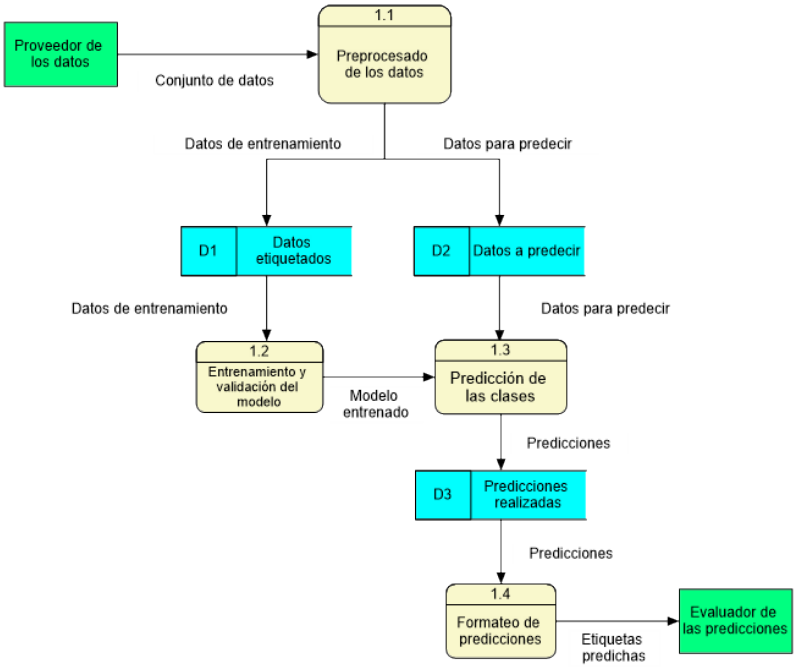

Figura 4.6: Diagrama de flujo de datos (nivel 1).

Como podemos observar, el proceso 0 se divide en 4 procesos principales: proceso 1.1, proceso 1.2, proceso 1.3 y proceso 1.4; además de contar con 3 almacenes de datos.

El proceso 1.1 es el proceso que recibe el conjunto de datos del proveedor de los datos y se encarga del preprocesado de los mismos. Esto puede incluir limpiarlos, "tokenizarlos", etc. En el capítulo 5 5 veremos de qué se encarga este proceso en cada modelo clasificador específico. Después de hacer lo anterior, este proceso genera:

- El conjunto de datos que ya están etiquetados y son guardados en el almacén de datos $D1$.
- El conjunto de datos que no están etiquetados y queremos predecir que son guardados en el almacén de datos $D2$.

El proceso 1.2 es el encargado de entrenar y validar un modelo clasificador a partir del conjunto de datos de entrenamiento almacenados en el almacén de datos $D1$. Este proceso genera un modelo entrenado que depende, lógicamente, del modelo clasificador y que veremos más en profundidad en los correspondientes diagramas de flujo de datos de nivel 2.

El proceso 1.3 recibe el modelo entrenado del proceso 1.2 y lee los datos a predecir del almacén de datos $D1$. Este proceso es el encargado de utilizar el modelo entrenado para predecir las etiquetas de los datos a predecir. Después, guarda las etiquetas predichas en el almacén de datos $D3$.

Por último, el proceso 1.4 es el encargado de leer las predicciones del almacén de datos $D3$ y darles el formato que necesitan para ser enviadas al evaluador de las predicciones.

Finalmente, vamos a ver con más detalle qué procesos tienen lugar dentro del proceso 1.2 para cada modelo clasificador específico. Para ello, vamos a analizar el diagrama de flujo de datos de nivel 2 que, como hemos comentado, depende del modelo clasificador.

Vamos a empezar con el diagrama de flujo de datos de nivel 2 para BERT.

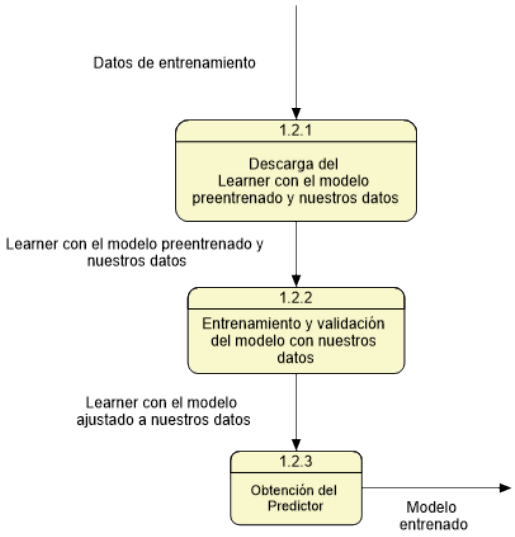

Figura 4.7: Diagrama de flujo de datos (nivel 2) BERT.

Como podemos observar, en este caso el proceso 1.2 se divide en tres procesos: proceso $1.2.1$, proceso $1.2.2$ y proceso $1.2.3$.

El proceso $1.2.1$ recibe los datos de entrenamiento y es el encargado de descargar el "Learner", que es un "wrapper" que contiene el modelo BERT pre-entrenado y nuestros datos.

El proceso $1.2.2$ recibe el "Learner" y es el encargado de re-entrenar el modelo pre-entrenado (este es el momento en el que tiene lugar la transferencia de aprendizaje mencionada en el capítulo 2) con nuestro conjunto de datos. Generando, de esta manera, un modelo ajustado a nuestros datos.

Finalmente, el proceso $1.2.3$ es el encargado de obtener el modelo entrenado del "Learner".

Continuamos con el diagrama de flujo de datos de nivel 2 para el modelo de `auto_ml`.

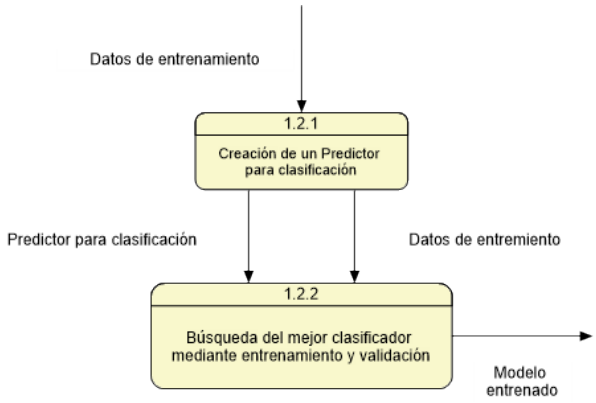

Figura 4.8: Diagrama de flujo de datos (nivel 2) `auto_ml`.

Como podemos observar, en este caso el proceso 1.2 se divide en dos procesos: proceso $1.2.1$ y proceso $1.2.2$.

El proceso $1.2.1$ recibe los datos de entrenamiento y crea un "Predictor" para clasificación.

El proceso $1.2.2$ recibe el "Predictor" para clasificación y los datos de entrenamiento, y se encarga de entrenar y validar el mejor modelo clasificador para nuestros datos. Generando, de esta manera, un modelo clasificador ajustado a nuestros datos de entrenamiento.

A continuación, vamos a analizar el diagrama de flujo de datos de nivel 2 para el `AutoML` de `h2o`.

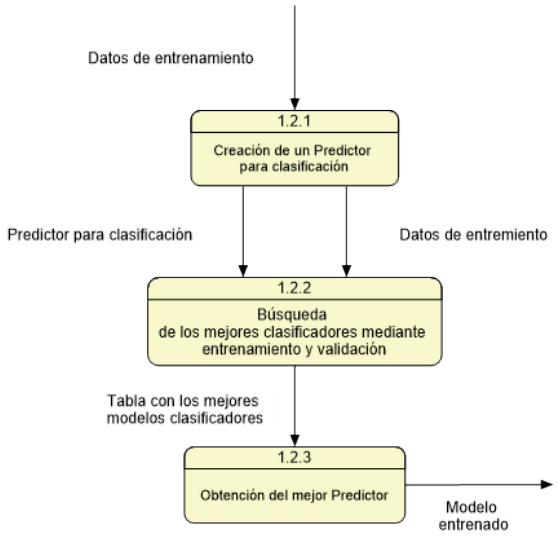

Figura 4.9: Diagrama de flujo de datos (nivel 2) `AutoML` de `h2o`

Como podemos observar, en este caso el proceso 1.2 se divide en tres procesos: proceso $1.2.1$, proceso $1.2.2$ y proceso $1.2.3$.

El proceso $1.2.1$ recibe los datos de entrenamiento y es el encargado de crear un predictor para clasificación.

El proceso 1.2.2 recibe el “Predictor” para clasificación y los datos de entrenamiento, y se encarga de entrenar y validar una serie de modelos clasificadores para nuestros datos. Generando una tabla con los mejores modelos clasificadores.

Finalmente, el proceso 1.2.3 es el encargado de extraer el mejor modelo clasificador para nuestros datos de la tabla con los mejores modelos clasificadores generada por el proceso 1.2.2.

Por último, vamos a analizar el diagrama de flujo de datos de nivel 2 para los modelos clasificadores utilizados en el primer experimento 6.1.

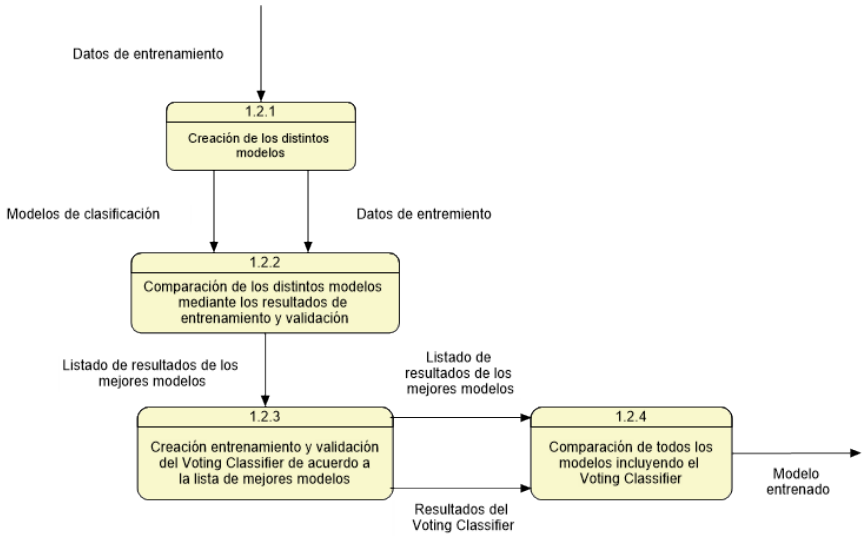

Figura 4.10: Diagrama de flujo de datos (nivel 2) modelos IMDB.

Como podemos observar, en este caso el proceso 1.2 se divide en cuatro procesos: proceso 1.2.1, proceso 1.2.2, proceso 1.2.3 y proceso 1.2.4.

El proceso 1.2.1 recibe los datos de entrenamiento y es el encargado de crear los distintos modelos de clasificación que aparecen listados en la tabla del experimento 6.1.

El proceso 1.2.2 recibe los modelos clasificadores creados por el proceso 1.2.1 y los datos de entrenamiento, y se encarga de comparar los distintos modelos mediante entrenamiento y validación de los mismos. Generando un listado con los resultados de los mejores modelos clasificadores.

El proceso 1.2.3 es el encargado de generar un “Voting Classifier”. Este es un modelo clasificador formado por los mejores clasificadores recibidos del proceso 1.2.2 que predice la etiqueta predicha por la mayoría de los modelos clasificadores que lo forman.

Finalmente, el proceso 1.2.4 recibe los resultados de todos los modelos clasificadores y se queda con el mejor modelo clasificador. Generando de esta manera el modelo entrenado.

Como hemos podido observar en esta sección, hemos tratado de que la lógica que siguen todos los modelos clasificadores sea similar, de manera que sea posible comparar sus rendimientos. Veremos

información más detallada sobre la implementación de cada modelo en el capítulo 5. Antes de eso, vamos a ver la planificación del proyecto.

4.3. Planificación del proyecto

En esta sección vamos a ver la planificación del proyecto desde su asignación hasta la entrega del mismo. Para ello, veremos un diagrama de Gantt con las tareas realizadas y, a continuación, explicaremos cada una de ellas en detalle y su grado de consecución.

El diagrama de Gantt del proyecto es el siguiente:

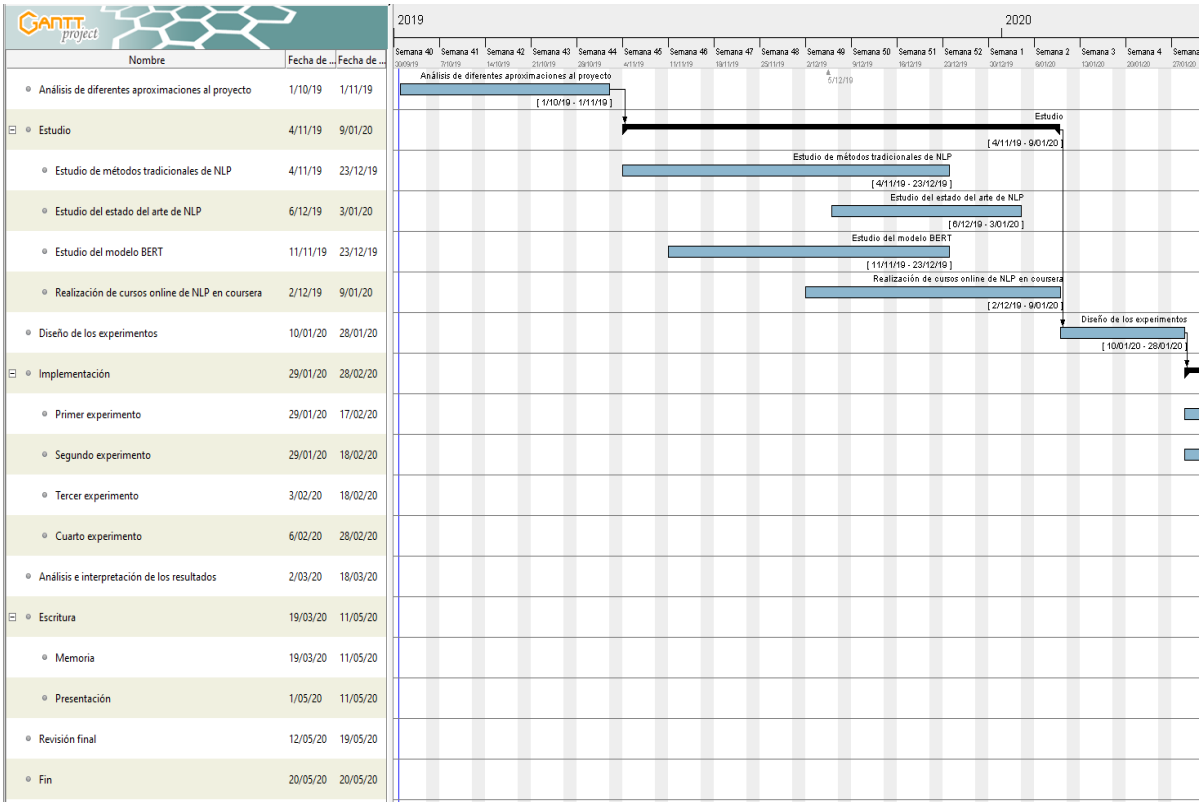

Figura 4.11: Diagrama de Gantt del proyecto 1/2.

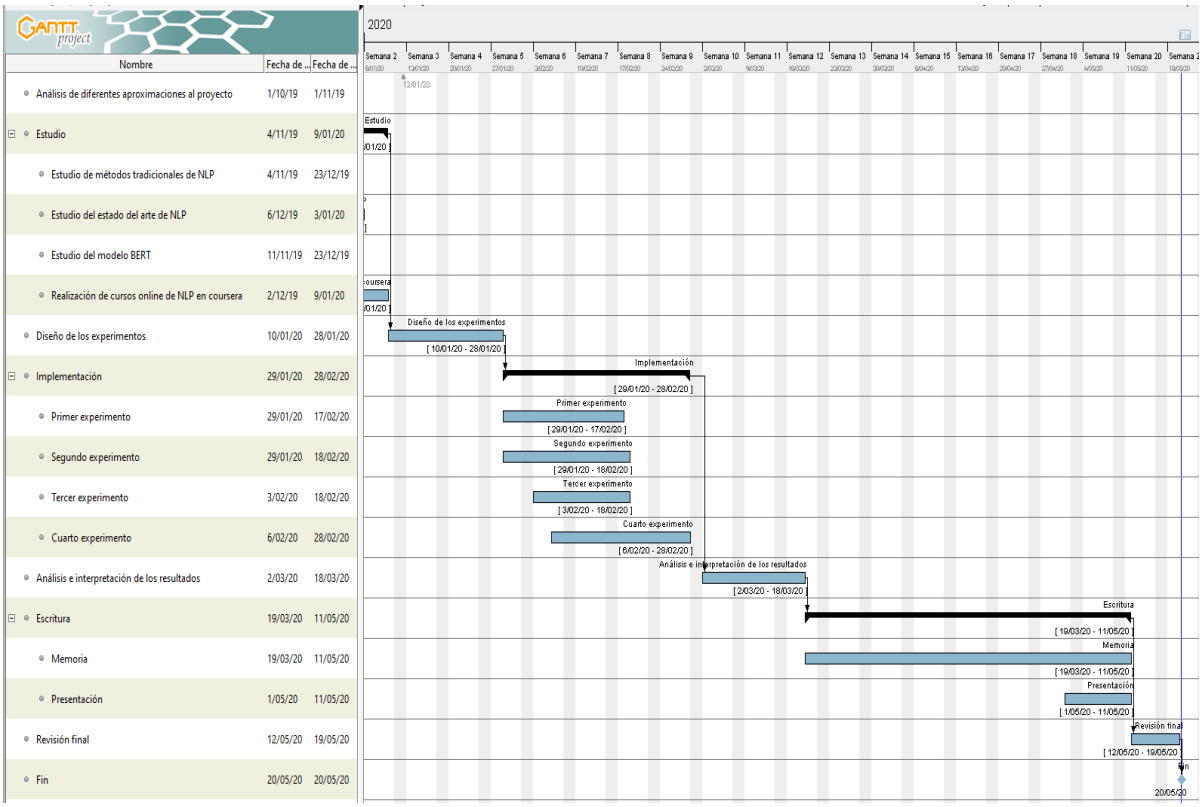

Figura 4.12: Diagrama de Gantt del proyecto 2/2.

La duración del proyecto ha sido de 232 días en total. La media de horas diarias trabajadas ha sido de $1{,}3$ (contando los fines de semana). Por lo tanto, la duración en horas del proyecto ha sido de aproximadamente $301{,}6$ horas. A continuación, vamos a describir en profundidad cada una de las tareas que aparecen en 4.11 y 4.12, y la dificultad (baja, media o alta) que han representado de cara a la realización del proyecto.

- *Análisis de diferentes aproximaciones al proyecto*: esta tarea ha consistido en definir sobre qué iba a tratar el proyecto. Para ello hemos tenido que llevar a cabo varias reuniones periódicas. Aunque ha sido una tarea larga debido a que a medida que pasaba el tiempo íbamos considerando nuevas ideas para el proyecto, no ha supuesto una gran dificultad en cuanto a su realización. Nivel de dificultad: bajo.
- *Estudio*: esta tarea se corresponde al estudio de NLP y la adquisición de los conocimientos necesarios para llevar a cabo el resto del proyecto. La hemos dividido en las siguientes subtareas:
  - *Estudio de métodos tradicionales de NLP*: esta tarea ha consistido en estudiar los métodos más tradicionales de NLP tales como BoW o TF-IDF. Ha supuesto aprender nuevos conceptos y comprender bien el funcionamiento de estos métodos, aunque gracias a la buena base matemática la dificultad se ha visto reducida. Nivel de dificultad: medio.
  - *Estudio del estado del arte de NLP*: esta tarea ha consistido en estudiar el estado del arte del NLP. Ha sido necesario estudiar los modelos mencionados en el capítulo 2, pero sin excederse en la profundidad del estudio. Nivel de dificultad: medio.
  - *Estudio del modelo BERT*: esta tarea ha consistido en estudiar en profundidad el modelo BERT, que es el modelo utilizado para lograr el objetivo principal del proyecto. Por ello, este estudio ha sido todo lo exhaustivo que nos ha sido posible. No obstante, la buena base matemática ha reducido la dificultad

del estudio. Nivel de dificultad: medio.

- *Realización de cursos online de NLP en coursera*: esta tarea ha consistido en realizar dos cursos de NLP: uno más básico y otro más avanzado. Todo ello con el objetivo de sentar una buena base para la realización de los experimentos posteriores. Nivel de dificultad: medio.

- *Diseño de experimentos*: esta tarea ha consistido en diseñar los experimentos a realizar. Ha requerido buscar los mejores datos para cada escenario en el que queríamos probar los modelos. Nivel de dificultad: medio.
- *Implementación*: esta tarea ha consistido en desarrollar los modelos clasificadores que se ajustan a los diagramas de flujo de datos 4.5 y 4.6. Todos los experimentos siguen la lógica mostrada en la figura 4.7 para el modelo BERT. No obstante, la lógica interna de cada modelo clasificador tradicional cambia de un experimento a otro. Por ello, esta tarea la hemos dividido en las siguientes subtareas:
  - *Primer experimento*: esta tarea ha consistido en implementar el primer experimento. Este experimento sigue la lógica mostrada en la figura 4.10 a la hora de entrenar y validar el modelo tradicional. Aunque dentro la parte correspondiente al modelo tradicional ha sido más tediosa que en el resto, su nivel de dificultad no ha sido muy elevado. Nivel de dificultad: medio.
  - *Segundo experimento*: esta tarea ha consistido en implementar el segundo experimento. Este experimento sigue la lógica mostrada en la figura 4.9 a la hora de entrenar y validar el modelo tradicional. Nivel de dificultad: medio.
  - *Tercer experimento*: esta tarea ha consistido en implementar el tercer experimento. Este experimento sigue la lógica mostrada en la figura 4.8 a la hora de entrenar y validar el modelo tradicional. Nivel de dificultad: medio.
  - *Cuarto experimento*: esta tarea ha consistido en implementar el cuarto experimento. Este experimento sigue la lógica mostrada en la figura 4.8 a la hora de entrenar y validar el modelo tradicional. Esta tarea ha sido la de mayor dificultad en el proyecto debido al manejo de ficheros y caracteres chinos. No obstante, el trabajo duro ha hecho que su dificultad no se vea reflejada en su duración en cuanto a días se refiere. Nivel de dificultad: alto.
- *Análisis interpretación de los resultados*: esta tarea ha consistido en analizar los resultados obtenidos en los experimentos e interpretar su significado para decidir si hemos conseguido lograr los objetivos y contrastar las hipótesis del proyecto listadas en el capítulo 3. Nivel de dificultad: medio.
- *Escritura*: esta tarea engloba toda la parte de escritura. Por lo tanto, la hemos dividido en las siguientes subtareas:
  - *Memoria*: esta tarea ha consistido en la escritura de la memoria en LaTeX. Gracias a la buena base de escritura en LaTeXy la disponibilidad de una plantilla para la memoria, la dificultad de esta tarea se ha visto reducida. Nivel de dificultad: medio.
  - *Presentación*: esta tarea ha consistido en confeccionar la presentación y preparar la exposición de la misma ante al tribunal. Nivel de dificultad: medio.
- *Revisión final*: esta tarea ha consistido en revisar la memoria y la presentación de cara a la entrega. Nivel de dificultad: bajo.
- *Fin*: esta tarea se corresponde con el hito final del proyecto. Su dificultad solo ha residido en la parte de defender el proyecto. Nivel dificultad: medio.

Podemos concluir que la dificultad del proyecto es, en general, media. Hemos tenido que aprender muchos nuevos conceptos y sobre el funcionamiento de modelos de NLP. Pero, gracias a una buena base matemática, no hemos tenido grandes problemas a la hora de entender el funcionamiento de estos modelos. Vista la planificación del proyecto, pasamos a la implementación de los experimentos.

5 Implementación

En este capítulo vamos a repasar los detalles de la implementación del proyecto. Concretamente, vamos a ver el sistema operativo utilizado, el lenguaje utilizado, las librerías empleadas y los ficheros que han sido necesarios.

El sistema operativo sobre el que hemos implementado los experimentos ha sido:

- Edición: Windows 10 Pro.
- Versión: 1903.
- Versión del sistema operativo: 18362.778.

En cuanto al lenguaje utilizado, lo primero es mencionar que hemos utilizado Anaconda con las siguientes versiones:

- anaconda: 2019.10.
- anaconda-client: 1.7.2.
- anaconda-navigator: 1.9.7.
- anaconda-project: 0.8.3.

En cuanto al lenguaje, hemos utilizado Python 3.7. La mayor parte del desarrollo se ha hecho sobre notebooks de Jupyter con las siguientes versiones:

- jupyter_client: 5.3.4.
- jupyter_core: 4.6.1.

Todo el proyecto ha sido desarrollado en un entorno virtual con la versión de Python antes mencionada y los paquetes instalados listados en el apéndice A.

También, comentar, que uno de los ficheros ha sido desarrollado en un “notebook” de kaggle (ya que queríamos probar la velocidad a la que se ejecutaba allí). Podemos ver información sobre ese entorno en este enlace (no le hemos instalado ningún paquete extra ni modificado nada en absoluto).

Por último, pasaremos a hablar sobre los archivos principales que componen el proyecto (no incluiremos archivos con pruebas que finalmente no han sido utilizadas en los experimentos) y la funcionalidad desarrollada en cada uno de ellos. Comentaremos, también, los detalles sobre la implementación

que consideramos relevantes. Antes que nada, daremos algunas características generales a algunos ficheros.

La primera es que el modelo BERT pre-entrenado utilizado es el del paquete `ktrain`. El paquete selecciona, automáticamente, la mejor versión de BERT para utilizar, de acuerdo al idioma y características del conjunto de datos. Este modelo necesita que los datos estén en la siguiente estructura de directorios: un directorio, cuyo nombre es irrelevante, que ha de contener dos subdirectorios: uno cuyo nombre sea *train* y otro cuyo nombre sea *test*. Estos subdirectorios, a su vez, contendrán un subdirectorio por cada clase que se quiera clasificar (el nombre de estos subdirectorios tiene que ser el nombre de la clase en cuestión). Finalmente, cada subdirectorio de clase contendrá archivos ".txt", cuyo nombre es irrelevante, cuyo contenido de texto pertenecerá a la clase del subdirectorio en el que se encuentran. Para más información sobre el paquete o la estructura de directorios ver [2]. Todos los ficheros en los que se especifique el uso de BERT, utilizan esta implementación y están basados en [20]. También, comentar que `ktrain` se encarga de preprocesar los datos para el modelo BERT.

La segunda es que en los ficheros que no utilizamos el modelo BERT, utilizaremos TF-IDF salvo en *autoML_chinese.ipynb* (cuyo caso explicaremos en detalle más adelante en este capítulo). Concretamente, utilizaremos el `TfidfVectorizer` de `sklearn` con la palabras de parada correspondientes al idioma del conjunto de datos. Esas palabras de parada, en el caso de portugués las sacamos de `nltk`. También, comentar que, en todos los ficheros con métodos tradicionales, manejamos los datos utilizando o bien `numpy.array`, o bien `DataFrame` de `pandas`. Esto último es debido a que facilita el proceso de llevar a cabo operaciones sobre los mismos. Por último, mencionar que en todos los ficheros, cuando hemos tenido que dividir el conjunto de datos para "train" y "validation" o para "train" y "test" (dependiendo del fichero), hemos utilizado `train_test_split()` de `sklearn`.

Por último, mencionar que todos los ficheros siguen la lógica general especificada en 4.5 y 4.6. Y la lógica particular que especificamos a continuación en cada fichero.

- *bert_basics.py*: en este archivo realizamos la clasificación mediante el modelo BERT del conjunto de datos IMDB. Se puede ver más información en 6.1. Este fichero sigue la lógica especificada en la figura 4.7.
- *BERT_realOrNot_ktrainversion_2epochs.ipynb*: en este notebook de Jupyter realizamos la clasificación mediante el modelo BERT de los datos de la competición Real or Not? NLP with Disaster Tweets de kaggle. Se puede ver más información en 6.2. Los datos son primero preprocesados utilizando el paquete `re` para patrones (veremos los detalles en 6.2). Este fichero sigue la lógica especificada en la figura 4.7.
- *BERT_Portuguese.ipynb*: en este notebook de Jupyter realizamos la clasificación mediante el modelo BERT de los datos de la competición FASAM - NLP Competition - Turma 4 de kaggle. Se puede ver más información en 6.3. Este fichero sigue la lógica especificada en la figura 4.7.
- *BERT_ChineseHotelReviews.ipynb*: en este notebook de Jupyter realizamos la clasificación mediante el modelo BERT del conjunto de datos de las "Chinese hotel reviews". Se puede ver más información en 6.4. Este fichero sigue la lógica especificada en la figura 4.7.
- *TRADITIONAL_imdb_noEffort.ipynb*: en este notebook de Jupyter realizamos la clasificación mediante modelos tradicionales del conjunto de datos IMDB. Se puede ver más información en 6.1. En concreto, en este notebook utilizamos los modelos clasificadores del paquete `sklearn` [3] listados en la tabla 6.1 del primer experimento

6.1. Para cada modelo, comprobamos la precisión sobre el conjunto de validación y el "recall". Aunque finalmente ordenamos los modelos respecto a la precisión sobre el conjunto de validación. Este fichero sigue la lógica especificada en la figura 4.10.

- *AutoML_realOrNot_preprocessing*: en este notebook de kaggle realizamos la clasificación mediante métodos tradicionales de los datos de la competición Real or Not? NLP with Disaster Tweets de kaggle. Se puede ver más información en 6.2. Los datos son primero preprocesados utilizando el paquete `re` para patrones (veremos los detalles en 6.2). En concreto, este notebook utiliza `H2OAutoML` del paquete `h2o` [1]. Este fichero sigue la lógica especificada en la figura 4.9.
- *autoML_portuguese.ipynb*: en este notebook de Jupyter realizamos la clasificación mediante modelos tradicionales de los datos de la competición FASAM - NLP Competition - Turma 4 de kaggle. Se puede ver más información en 6.3. En concreto, este notebook utiliza `Predictor` del paquete `auto_ml` [4]. Este fichero sigue la lógica especificada en la figura 4.8.
- *autoML_chinese.ipynb*: en este notebook de Jupyter realizamos la clasificación mediante modelos tradicionales del conjunto de datos de las "Chinese hotel reviews". Se puede ver más información en 6.4. En concreto, este notebook utiliza `Predictor` del paquete `auto_ml` [4]. Para conseguir implementar este notebook, nos hemos basado en [40]. Primero hemos tenido que leer los archivos en chino utlizando `codecs`, después hemos tenido que convertirlos en caracteres chinos simplificados utilizando el archivo `langconv.py` disponible en la fuente original [40]. A continuación, utilizando `keras`, hemos "tokenizado" los datos para después convertirlos en una secuencia numérica, es decir, los hemos vectorizado. Para más información ver la fuente original [40] en la que nos hemos basado. Este fichero sigue la lógica especificada en la figura 4.8.

Añadir, que cuando hemos tenido que utilizar BERT con los datos descargados de las competiciones de kaggle, lo primero ha sido generar la estructura de directorios antes mencionada.

En general, podemos concluir que la implementación del modelo BERT no ha requerido prácticamente esfuerzo alguno, ya que es prácticamente la misma independientemente del idioma (que es detectado automáticamente) o del conjunto de datos (siempre y cuando este mantenga la estructura de directorios antes mencionada). En cambio, la implementación de los métodos tradicionales ha requerido un esfuerzo considerable, sobretodo en *autoML_chinese.ipynb*, en el que hemos tenido que manejar caracteres en chino que desconocíamos. Pero también en *TRADITIONAL_imdb_noEffort.ipynb*, que hemos tenido que analizar en detalle varios modelos y sus resultados para crear el mejor modelo. En el siguiente capítulo describiremos los experimentos y presentaremos los resultados obtenidos en cada uno.

6

Experimentos

En este capítulo vamos a presentar los experimentos que hemos realizado con el objetivo de comparar BERT con algoritmos más tradicionales de ML, que hemos presentado en el capítulo 3. Primero, expondremos el idioma de los datos utilizados y explicaremos en qué consiste el experimento que hemos realizado, así como su objetivo. Después, presentaremos los resultados obtenidos tras la realización del mismo.

6.1. Primer experimento: IMDB movie reviews

6.1.1. Descripción

Idioma de los datos: Inglés. Este experimento es sobre análisis de sentimiento. Para ello, hemos utilizado el conocido "IMDB dataset" (descargado de este enlace), que contiene 50000 reseñas de cine ("movie reviews"), de las cuáles serán utilizada 25000 para entrenar el modelo ("train") y 25000 para probar el modelo ("test"). Estos datos están clasificados en dos clases:

- *pos*: reseñas de cine positivas.
- *neg*: reseñas de cine negativas.

Nuestro objetivo con este experimento, era ver el alcance de BERT. Es decir, queríamos ver qué tipo de resultados podíamos llegar a esperar de BERT en el resto de experimentos y por cuánto superaría a los métodos más tradicionales.

6.1.2. Resultados

A continuación, vamos a ver los resultados del experimento. Hemos decidido presentar el porcentaje de aciertos sobre el conjunto de "test", que en este experimento son 25000 reseñas de películas:

Modelo	Precisión validación
BERT	**0.9387**
Voting Classifier	0.9007
Logistic Regression	0.8949
Linear SVC	0.8989
Multinomial NB	0.8771
Ridge Classifier	0.8990
Passive Aggresive Classifier	0.8931

Tabla 6.1: Precisión sobre el conjunto de validación de los distintos métodos en el primer experimento.

Como podemos ver, los mejores resultados los hemos obtenido con el modelo BERT (como esperábamos). Además de lo mostrado en la tabla, este modelo ha conseguido: pérdida: 0.2452, precisión: 0.8988 y pérdida sobre el conjunto de validación: 0.1618. Cabe también destacar la gran cantidad de trabajo que ha costado obtener estos resultados con los métodos tradicionales comparado con BERT.

6.2. Segundo experimento: RealOrNot tweets

6.2.1. Descripción

Idioma de los datos: Inglés. Dado que con el experimento anterior 6.1 comparamos BERT con métodos más tradicionales. Decidimos presentarnos a una competeción de kaggle para ver qué resultado éramos capaces de obtener contra los modelos presentados por otros concursantes. El nombre de la competición es Real or Not? NLP with Disaster Tweets. Los datos de la competición se descargan en este enlace. Esta competición consiste en clasificar un conjunto de "tweets" en dos clases:

- *1*: "tweets" sobre un desastre natural real.
- *0*: "tweets" que no son sobre un desastre natural real.

Cabe destacar que en este experimento solo hemos utilizado las columnas "tweet" y "class" disponibles en los datos. Además, hemos preprocesado la columna "tweet" eliminando URLs y haciendo las siguientes sustituciones:

- "#anything" por "hashtag".
- "@anyone" por "entity".

Después de ello hemos generado la estructura de directorios que necesita nuestro modelo BERT (explicada en el capítulo 5), utilizando el 75 % de los datos para entrenar el modelo y el 25 % para

validarlo.

El objetivo de este experimento era comprobar en una competición real cómo de bueno era el rendimiento de los modelos.

6.2.2. Resultados

A continuación, vamos a ver los resultados del experimento. Hemos decidido presentar el porcentaje de aciertos sobre el conjunto de validación, que en este caso es el 25 % de los datos disponible para entrenar el modelo. Además, incluimos el "score" obtenido al clasificar los datos de la competición con el modelo:

Modelo	Precisión validación	"Score" competición
BERT	**0.8361**	**0.83640**
H2OAutoML	0.7875	0.77607

Tabla 6.2: Resultados del segundo experimento.

El modelo ganador con el algoritmo de `AutoML` tiene las siguientes características:

- H2OStackedEnsembleEstimator: Stacked Ensemble.
- Model Key: StackedEnsemble_BestOfFamily_AutoML_20200221_120302.

El resultado en la competición lo podemos ver en este enlace (Santiago González). Nuestro puesto está en torno al top 18 % de unos 2525 participantes.

En cuanto a los resultados, como podemos ver, los mejores han sido obtenidos con el modelo BERT (como esperábamos), que incluso ha conseguido un buen resultado en la clasificación de la competición. Además de lo mostrado en la tabla, este modelo ha conseguido: pérdida: 0.3596, precisión: 0.8529 y pérdida sobre el conjunto de validación: 0.4167. Por otro lado, de nuevo, la implementación BERT ha requerido un esfuerzo notablemente menor.

Luego, podemos concluir, a partir del experiemento anterior 6.1 y de este, que el modelo BERT tiene muy buen rendimiento en inglés y es capaz de conseguir muy buenos resultados incluso en una competición de kaggle.

6.3. Tercer experimento: Portuguese news

6.3.1. Descripción

Idioma de los datos: Portugués. Dado que los experimentos anteriores nos darían una idea del rendimiento de BERT cuando los datos estan en inglés, nos planteamos comprobar el rendimiento del mismo en otros idiomas. Es por ello, que decidimos presentarnos a otra competición de kaggle. El nombre de esta competición es FASAM - NLP Competition - Turma 4. Los datos de la misma se pueden descargar desde este enlace. Esta competición, consiste en clasificar una serie de artículos de un periódico en distintas categorías (una categoría por artículo). Las clases (que se corresponden con las categorías) son: *ambiente*, *equilibrioesaude*, *sobretudo*, *educacao*, *ciencia*, *tec*, *turismo*, *empreendedorsocial* y *comida*.

En este experimento, solo hemos utilizado las columnas “article text” y “class”. Después de quitar el resto de columnas hemos generado la estructura de directorios que necesita nuestro modelo BERT (explicada en el capítulo 5), utilizando el 75 % de los datos para entrenar el modelo y el 25 % para validarlo.

El objetivo de este experimento era comprobar si los modelos (y, en particular, BERT) eran capaces de mantener su rendimiento en otros idiomas distintos al inglés.

6.3.2. Resultados

A continuación, vamos a ver los resultados del experimento. Hemos decidido presentar el porcentaje de aciertos sobre el conjunto de validación, que en este caso vuelve a ser el 25 % de los datos disponible para entrenar el modelo. Además, incluimos el “score” obtenido al clasificar los datos de la competición con el modelo:

Modelo	Precisión validación	“Score” competición
BERT	**0.9093**	**0.91196**
Predictor (auto_ml)	0.8480	0.85047

Tabla 6.3: Resultados del tercer experimento.

El modelo ganador con el algoritmo de `auto_ml` es un *GradientBoostingClassifier*.

El resultado en la competición lo podemos ver en este enlace (Santiago González). Nuestro puesto en la clasificación pública es primeros de 18, y en la clasificación privada (aparece cuando termina una competición; la competición de 6.2 sigue abierta) hemos quedado segundos de 18.

En cuanto a los resultados, como podemos ver, los mejores han sido obtenidos, de nuevo, con el

modelo BERT (como esperábamos), que incluso ha conseguido un muy buen resultado en la clasificación de la competición. Además de lo mostrado en la tabla, este modelo ha conseguido: pérdida: 0.1436, precisión: 0.9515 y pérdida sobre el conjunto de validación: 0.2748. Por otro lado, de nuevo, la implementación BERT ha requerido un esfuerzo mucho menor. Teniendo en cuenta, sobretodo, que estábamos trabajando en un idioma que desconocemos (portugués).

Luego, podemos concluir, a partir de los experimentos 6.1, 6.2 y de este, que el modelo BERT tiene muy buen rendimiento, no solo en inglés, sino también en portugués (que comparte el mismo alfabeto) y es capaz de conseguir muy buenos resultados incluso en una competición de kaggle.

6.4. Cuarto experimento: Chinese hotel reviews

6.4.1. Descripción

Idioma de los datos: Chino peninsular con caracteres simplificados, zh-CN. Por último, decidimos comprobar el rendimiento de BERT en un idioma que ni siquiera comparta el mismo alfabeto que el inglés. Para ello, decidimos utilizar el "Chinese hotel reviews dataset" (descargado de este enlace). Estos datos contienen valoraciones de hoteles ("hotel reviews") clasificadas en dos clases:

- *pos*: valoraciones positivas.
- *neg*: valoraciones negativas.

En este experimento hemos utilizado el 85 % de los datos para entrenar el modelo y el 15 % para probar el modelo. El porcentaje de datos elegido para entrenar el modelo ha subido respecto a los experimentos anteriores por el menor tamaño del conjunto de datos (6000 "hotel reviews" en total).

El objetivo de este experimento era comprobar si los modelos (y, en particular, BERT) eran capaces de mantener su rendimiento en idiomas que no comparte ni siquiera el alfabeto del inglés.

6.4.2. Resultados

A continuación, vamos a ver los resultados del experimento. Hemos decidido presentar el porcentaje de aciertos sobre el conjunto de "test":

Modelo	Precisión validación
BERT	**0.9381**
Predictor (auto_ml)	0.7399

Tabla 6.4: Resultados del cuarto experimento.

El modelo ganador con el algoritmo de `auto_ml` es un *GradientBoostingClassifier*. Del cual, tam-

bién tenemos que mencionar que ha resultado ser un modelo bastante malo para esta tarea. Esto se debe a que la probabilidad de que el modelo clasifique en cada clase es muy cercana. De hecho, su propio método “score” da un valor de aproximadamte -0.17682.

Por consiguiente, y como podemos ver en la tabla 6.4, los mejores resultados los hemos obtenido con el modelo BERT (como esperábamos). Además de lo mostrado en la tabla, este modelo ha conseguido: pérdida: 0.0441, precisión: 0.9855 y pérdida sobre el conjunto de validación: 0.2181. Cabe también destacar la diferencia de trabajo entre BERT y el resto de modelos, ya que en este experimento hemos tenido que manipular caracteres y ficheros en chino y no ha sido trivial.

6.5. Conclusiones

Las conclusiones que podemos sacar de estos experimentos, viendo sus tablas 6.1, 6.2, 6.3 y 6.4, es que BERT supera el rendimiento del resto de los modelos en los casos propuestos. Y, además, el trabajo requerido para conseguir esos resultados con BERT es muchísimo menor que el necesario para conseguir los resultados que hemos obtenido con el resto de modelos. Por último, queremos destacar el impacto que tiene la transferencia de aprendizaje (“transfer learning”), ya que el modelo BERT estaba pre-entrenado y eso nos ha hecho conseguir estos resultados. En el experimento que más se nota este impacto es en el cuarto 6.4, ya que el conjunto de datos a predecir era de un tamaño mucho menor al resto.

7

Conclusiones y trabajo futuro

En este capítulo vamos a comentar las conclusiones del proyecto y algunas proposiciones para líneas de trabajo futuro.

En cuanto a las conclusiones, de acuerdo a los experimentos que hemos realizado, podemos concluir que BERT supera el rendimiento de modelos de NLP tradicionales en tareas de clasificación. No solo eso, sino que facilita la tarea de clasificación llevando al mínimo el esfuerzo que tenemos que realizar para conseguir buenos resultados como los mostrados en los experimentos del capítulo 6.

En cuanto a los objetivos y las hipótesis listados en el capítulo 3, vamos a ver el grado de consecución de cada uno.

O-1.– Comparar el rendimiento de BERT con algoritmos más tradicionales de NLP en Inglés. Este objetivo ha sido logrado mediante los experimentos 6.1 y 6.2.

O-2.– Comparar el rendimiento de BERT con algoritmos más tradicionales de NLP en otros idiomas con incluso distintos alfabetos. Este objetivo ha sido logrado mediante los experimentos 6.3 y 6.4. Concretamente, en el experimento 6.4, el idioma de los datos utilizados tenía un alfabeto distinto al inglés.

O-3.– Medir el rendimiento de BERT en alguna clase de competición. Este objetivo ha sido logrado mediante los experimentos 6.2 y 6.3, que se han correspondido a competiciones de kaggle.

O-4.– Demostrar la efectividad de BERT sobre conjuntos de datos pequeños gracias a la transferencia de aprendizaje. Este objetivo ha sido logrado mediante el experimento 6.4.

O-5.– Optimización Bayesiana de algunos parámetros de BERT. Este objetivo no ha sido logrado, debido a las restricciónes R-1 y R-3. El ordenador personal del que disponíamos no dispone de GPUs ni de mucha memoria RAM, lo cual ha provacado que no puediéramos lograr este objetivo en ese ordenador personal (R-1). Además, no hemos podido disponer de GPUs como las del Centro de Computación Científica de la UAM debido a R-3.

H-1.– BERT tiene un mejor rendimiento que algoritmos más tradicionales como por ejemplo un algoritmo que utilice TF-IDF seguido de un clasificador. Esta hipótesis ha sido contrastada mediante todos los experimentos explicados en el capítulo 6, siempre y cuando consideremos que se cumple la asunción A-3, y que los modelos utilizados en 6.1 son los que mejor resultado pueden obtener.

H-2.– BERT es un modelo capaz de obtener buenos resultados en cualquier idioma. Esta hipótesis ha sido contrastada en inglés, portugués y chino peninsular con caracteres simplificados. Mediante los experimentos explicados en el capítulo 6.

H-3.– BERT es fácil de implementar comparado con los modelos tradicionales. Esta hipótesis ha sido contrastada durante la implementación de todos los experimentos explicados en el capítulo 6 de la manera detallada en el

capítulo 5.

H-4.– BERT es capaz de conseguir buenos resultados incluso sobre conjuntos de datos pequeños. Esta hipótesis ha sido contrastada mediante el experimento 6.4.

Podemos concluir, a partir del grado de consecución de cada uno de los objetivos e hipótesis anteriores, que hemos cumplido prácticamente con todos los objetivos del proyecto, salvo, concretamente, el O-5. Por consiguiente, y a la luz de los resultados presentados en el capítulo 6, proponemos las siguientes líneas de trabajo futuro:

- Optimización Bayesiana de algunos parámetros de BERT. Para ello será necesaria una gran capacidad computacional y tomar otra implementación de BERT como esta.
- Realizar experimentos en más idiomas con distintos alfabetos como, por ejemplo, ruso o árabe.
- Probar más métodos tradicionales como, por ejemplo, GloVe.
- Participar en nuevas competiciones de kaggle para comprobar el rendimiento en otros conjuntos de datos.
- Utilizar BERT para otros problemas de NLP como por ejemplo la contestación a preguntas.

Como podemos ver, los experimentos realizados en este proyecto dan lugar a muchas líneas de trabajo futuro. No obstante, creemos que las mencionadas anteriormente son las que deberían ser consideradas como principales.

Bibliografía

[1] *AutoML: Automatic Machine Learning — H2O 3.30.0.2 documentation*. http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html.

[2] *ktrain: ktrain is a wrapper for TensorFlow Keras that makes deep learning and AI more accessible and easier to apply*. https://pypi.org/project/ktrain/.

[3] *scikit-learn: machine learning in Python — scikit-learn 0.22.2 documentation*. https://scikit-learn.org/stable/.

[4] *Welcome to auto_ml's documentation! — auto_ml 0.1.0 documentation*. https://auto-ml.readthedocs.io/en/latest/.

[5] J. L. Ba, J. R. Kiros, and G. E. Hinton, *Layer normalization*, arXiv preprint arXiv:1607.06450, (2016).

[6] D. Bikel and I. Zitouni, *Multilingual natural language processing applications: from theory to practice*, IBM Press, 2012.

[7] E. Cambria and B. White, *Jumping nlp curves: A review of natural language processing research*, IEEE Computational intelligence magazine, 9 (2014), pp. 48–57.

[8] L. Chen, L. Song, Y. Shao, D. Li, and K. Ding, *Using natural language processing to extract clinically useful information from chinese electronic medical records*, International journal of medical informatics, 124 (2019), pp. 6–12.

[9] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, *Transformer-xl: Attentive language models beyond a fixed-length context*, arXiv preprint arXiv:1901.02860, (2019).

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, arXiv preprint arXiv:1810.04805, (2018).

[11] A. Farzindar and D. Inkpen, *Natural language processing for social media*, Synthesis Lectures on Human Language Technologies, 8 (2015), pp. 1–166.

[12] Y. Goldberg and O. Levy, *word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method*, arXiv preprint arXiv:1402.3722, (2014).

[13] B. F. Green Jr, A. K. Wolf, C. Chomsky, and K. Laughery, *Baseball: an automatic question-answerer*, in Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference, 1961, pp. 219–224.

[14] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[15] J. Howard and S. Ruder, *Universal language model fine-tuning for text classification*, arXiv preprint arXiv:1801.06146, (2018).

[16] C. J. Hutto and E. Gilbert, *Vader: A parsimonious rule-based model for sentiment analysis of social media text*, in Eighth international AAAI conference on weblogs and social media, 2014.

[17] M. Joshi, D. Chen, Y. Liu, D. S. Weld, L. Zettlemoyer, and O. Levy, *Spanbert: Improving pre-training by representing and predicting spans*, Transactions of the Association for Computational Linguistics, 8 (2020), pp. 64–77.

[18] A. Karpathy, J. Johnson, and L. Fei-Fei, *Visualizing and understanding recurrent networks*, arXiv preprint arXiv:1506.02078, (2015).

[19] D. Khurana, A. Koli, K. Khatter, and S. Singh, *Natural language processing: State of the art, current trends and challenges*, arXiv preprint arXiv:1708.05148, (2017).

[20] A. Maiya, *BERT Text Classification in 3 Lines of Code Using Keras*. https://towardsdatascience.com/bert-text-classification-in-3-lines-of-code-using-keras-264db7e7a358, Jan. 2020.

[21] N. Mamede, J. Baptista, C. Diniz, and V. Cabarrão, *String: An hybrid statistical and rule-based natural language processing chain for portuguese*, (2012).

[22] C. D. Manning, C. D. Manning, and H. Schütze, *Foundations of statistical natural language processing*, MIT press, 1999.

[23] T. Mikolov, K. Chen, G. Corrado, and J. Dean, *Efficient estimation of word representations in vector space*, arXiv preprint arXiv:1301.3781, (2013).

[24] G. A. Miller, *Wordnet: a lexical database for english*, Communications of the ACM, 38 (1995), pp. 39–41.

[25] S. J. Pan and Q. Yang, *A survey on transfer learning*, IEEE Transactions on knowledge and data engineering, 22 (2009), pp. 1345–1359.

[26] J. Pennington, R. Socher, and C. D. Manning, *Glove: Global vectors for word representation*, in Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), 2014, pp. 1532–1543.

[27] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, *Deep contextualized word representations*, arXiv preprint arXiv:1802.05365, (2018).

[28] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, *Improving language understanding by generative pre-training*, URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf, (2018).

[29] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, *Language models are unsupervised multitask learners*, OpenAI Blog, 1 (2019), p. 9.

[30] J. W. Rae, A. Potapenko, S. M. Jayakumar, and T. P. Lillicrap, *Compressive transformers for long-range sequence modelling*, arXiv preprint arXiv:1911.05507, (2019).

[31] S. Robertson, *Understanding inverse document frequency: on theoretical arguments for idf*, Journal of documentation, (2004).

[32] C. Rosset, *Turing-NLG: A 17-billion-parameter language model by Microsoft*, Feb. 2020.

[33] Y. Sun, S. Wang, Y. Li, S. Feng, X. Chen, H. Zhang, X. Tian, D. Zhu, H. Tian, and H. Wu, *Ernie: Enhanced representation through knowledge integration*, arXiv preprint arXiv:1904.09223, (2019).

[34] P. Tum, *A Survey of the State-of-the-Art Language Models up to Early 2020*, Mar. 2020.

[35] H. Uchida and M. Zhu, *The universal networking language beyond machine translation*, in International Symposium on Language in Cyberspace, Seoul, 2001, pp. 26–27.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, *Attention is all you need*, in Advances in neural information processing systems, 2017, pp. 5998–6008.

[37] S. Vijayarani, M. J. Ilamathi, and M. Nithya, *Preprocessing techniques for text mining-an overview*, International Journal of Computer Science & Communication Networks, 5 (2015), pp. 7–16.

[38] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al., *Google's neural machine translation system: Bridging the gap between human and machine translation*, arXiv preprint arXiv:1609.08144, (2016).

[39] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, *Xlnet: Generalized autoregressive pretraining for language understanding*, in Advances in neural information processing systems, 2019, pp. 5754–5764.

[40] C. Zhang, *Tony607/Chinese_sentiment_analysis*. https://github.com/Tony607/Chinese_sentiment_analysis, Apr. 2020. original-date: 2017-09-26T03:34:24Z.

[41] Y. Zhang, R. Jin, and Z.-H. Zhou, *Understanding bag-of-words model: a statistical framework*, International Journal of Machine Learning and Cybernetics, 1 (2010), pp. 43–52.

Acrónimos

AI “Artificial Intelligence”.

BERT “Bidirectional Encoder Representations from Transformers”.

BoW “Bag of Words”.

ELMo “Embeddings from Language Models”.

ERNIE “Enhanced Representation through Knowledge Integration”.

GloVe “Global Vectors for Word Representation”.

GPT “Generative Pretrained Transformer”.

LSTM “Long Short-Term Memory”.

ML “Machine Learning”.

NLP “Natural Language Processing”.

ReLU “Rectified Linear Unit”.

TF-IDF “Term Frequency - Inverse Document Frequency”.

T-NLG “Turing Natural Language Generation”.

ULMFiT “Universal Language Model Fine-tunning for Text Classification”.

VADER “Valence Aware Dictionary for sEntiment Reasoning”.

Apéndices

A

Paquetes instalados

En este apéndice, listamos todos los paquetes instalados en el entorno virtual mencionado en el capítulo 5. La lista de paquetes (con sus respectivas versiones) es la siguiente:

- _tflow_select: 2.1.0.
- absl-py: 0.8.1.
- astor: 0.8.0.
- attrs: 19.3.0.
- auto-ml: 2.9.10.
- backcall: 0.1.0.
- bert-tensorflow: 1.0.1.
- blas: 1.0.
- bleach: 3.1.0.
- bokeh: 1.4.0.
- boto3: 1.11.14.
- botocore: 1.14.14.
- bz2file: 0.98.
- ca-certificates: 2020.1.1.
- cachetools: 4.0.0.
- calmap: 0.0.7.
- cchardet: 2.1.5.
- certifi: 2019.11.28.
- cffi: 1.14.0.
- chardet: 3.0.4.
- click: 7.0.
- cloudpickle: 1.2.2.
- colorama: 0.4.3.
- cudatoolkit: 10.0.130.
- cudnn: 7.6.5.
- cycler: 0.10.0.
- deap: 1.3.1.

- decorator: 4.4.1.
- defusedxml: 0.6.0.
- dill: 0.2.9.
- docutils: 0.15.2.
- dopamine-rl: 3.0.1.
- entrypoints: 0.3.
- fastprogress: 0.2.2.
- filelock: 3.0.12.
- flask: 1.1.1.
- freetype: 2.9.1.
- future: 0.18.2.
- gast: 0.2.2.
- gevent: 1.4.0.
- gin-config: 0.3.0.
- google-api-python-client: 1.7.11.
- google-auth: 1.11.2.
- google-auth-httplib2: 0.0.3.
- google-pasta: 0.1.8.
- googleapis-common-protos: 1.51.0.
- greenlet: 0.4.15.
- grpcio: 1.16.1.
- gunicorn: 20.0.4.
- gym: 0.16.0.
- h2o: 3.28.1.2.
- h5py: 2.10.0.
- hdf5: 1.10.4.
- httplib2: 0.17.0.
- icc_rt: 2019.0.0.
- icu: 58.2.
- idna: 2.8.
- importlib_metadata: 1.5.0.
- intel-openmp: 2020.0.
- ipykernel: 5.1.4.
- ipython: 7.12.0.
- ipython_genutils: 0.2.0.
- itsdangerous: 1.1.0.
- jedi: 0.16.0.
- jieba: 0.42.1.
- jinja2: 2.11.1.
- jmespath: 0.9.4.

- joblib: 0.14.1.
- jpeg: 9b.
- jsonschema: 3.2.0.
- jupyter_client: 5.3.4.
- jupyter_core: 4.6.1.
- keras-applications: 1.0.8.
- keras-base: 2.2.4.
- keras-bert: 0.81.0.
- keras-embed-sim: 0.7.0.
- keras-gpu: 2.2.4.
- keras-layer-normalization: 0.14.0.
- keras-multi-head: 0.22.0.
- keras-pos-embd: 0.11.0.
- keras-position-wise-feed-forward: 0.6.0.
- keras-preprocessing: 1.1.0.
- keras-self-attention: 0.41.0.
- keras-transformer: 0.32.0.
- kfac: 0.2.0.
- kiwisolver: 1.1.0.
- ktrain: 0.9.2.
- langdetect: 1.0.7.
- libpng: 1.6.37.
- libprotobuf: 3.11.2.
- libsodium: 1.0.16.
- lightgbm: 2.0.12.
- m2w64-gcc-libgfortran: 5.3.0.
- m2w64-gcc-libs: 5.3.0.
- m2w64-gcc-libs-core: 5.3.0.
- m2w64-gmp: 6.1.0.
- m2w64-libwinpthread-git: 5.0.0.4634.697f757.
- markdown: 3.1.1.
- markupsafe: 1.1.1.
- matplotlib: 3.1.3.
- matplotlib-base: 3.1.3.
- mesh-tensorflow: 0.1.10.
- mistune: 0.8.4.
- mkl: 2019.5.
- mkl-service: 2.3.0.
- mkl_fft: 1.0.15.
- mkl_random: 1.1.0.

- mpmath: 1.1.0.
- msys2-conda-epoch: 20160418.
- multiprocess: 0.70.9.
- nbconvert: 5.6.1.
- nbformat: 5.0.4.
- networkx: 2.3.
- nltk: 3.4.5.
- notebook: 6.0.3.
- numpy: 1.18.1.
- numpy-base: 1.18.1.
- oauth2client: 4.1.3.
- opencv-python: 4.2.0.32.
- openjdk: 11.0.6.
- openssl: 1.1.1.
- opt_einsum: 3.1.0.
- packaging: 20.1.
- pandas: 0.25.3.
- pandoc: 2.2.3.2.
- pandocfilters: 1.4.2.
- parso: 0.6.1.
- pathos: 0.2.5.
- patsy: 0.5.1.
- pickleshare: 0.7.5.
- pillow: 7.0.0.
- pip: 20.0.2.
- plotly: 4.4.1.
- pox: 0.2.7.
- ppft: 1.6.6.1.
- prometheus_client: 0.7.1.
- promise: 2.3.
- prompt_toolkit: 3.0.3.
- protobuf: 3.11.2.
- pyasn1: 0.4.8.
- pyasn1-modules: 0.2.8.
- pycparser: 2.19.
- pyglet: 1.5.0.
- pygments: 2.5.2.
- pyparsing: 2.4.6.
- pypng: 0.0.20.
- pyqt: 5.9.2.

- pyreadline: 2.1.
- pyrsistent: 0.15.7.
- python: 3.7.6.
- python-dateutil: 2.8.1.
- pytz: 2019.3.
- pywin32: 227.
- pywinpty: 0.5.7.
- pyyaml: 5.3.
- pyzmq: 18.1.1.
- qt: 5.9.7.
- regex: 2020.1.8.
- requests: 2.22.0.
- retrying: 1.3.3.
- rsa: 4.0.
- s3transfer: 0.3.3.
- sacremoses: 0.0.38.
- scikit-learn: 0.21.3.
- scipy: 1.4.1.
- seaborn: 0.10.0.
- send2trash: 1.5.0.
- sentencepiece: 0.1.85.
- seqeval: 0.0.12.
- setuptools: 45.2.0.
- sip: 4.19.8.
- six: 1.14.0.
- sklearn-deap2: 0.2.2.
- sqlite: 3.31.1.
- statsmodels: 0.11.0.
- sympy: 1.5.1.
- tabulate: 0.8.7.
- tensor2tensor: 1.15.4.
- tensorboard: 2.0.0.
- tensorflow: 2.0.0.
- tensorflow-base: 2.0.0.
- tensorflow-datasets: 2.0.0.
- tensorflow-estimator: 2.0.0.
- tensorflow-gan: 2.0.0.
- tensorflow-gpu: 2.0.0.
- tensorflow-hub: 0.7.0.
- tensorflow-metadata: 0.21.1.

- tensorflow-probability: 0.7.0.
- termcolor: 1.1.0.
- terminado: 0.8.3.
- testpath: 0.4.4.
- tokenizers: 0.0.11.
- tornado: 6.0.3.
- tqdm: 4.42.1.
- traitlets: 4.3.3.
- transformers: 2.4.1.
- uritemplate: 3.0.1.
- urllib3: 1.25.8.
- vc: 14.1.
- vs2015_runtime: 14.16.27012.
- wcwidth: 0.1.8.
- webencodings: 0.5.1.
- werkzeug: 1.0.0.
- wheel: 0.34.2.
- wincertstore: 0.2.
- winpty: 0.4.3.
- wrapt: 1.11.2.
- yaml: 0.1.7.
- zeromq: 4.3.1.
- zipp: 2.2.0.
- zlib: 1.2.11.