

FACULTAD DE CIENCIAS
GRADO EN MATEMÁTICAS
TRABAJO FIN DE GRADO
CURSO ACADÉMICO 2024-2025

TÍTULO:

**ESTIMACIÓN DE COMPLEJIDAD COMPUTACIONAL CON
GRANDES MODELOS DE LENGUA**

AUTOR:

DANIEL SÁNCHEZ PAGÁN

Resumen

Abstract

Índice

1. Introducción	5
2. Antecedentes	5
3. Grandes modelos de lengua	7
3.1. Transformers	8
3.2. Generación de texto	12
3.3. Entrenamiento	12
3.4. Evaluación	13
3.5. Ajuste fino	14
3.6.	14
4. Experimentos	14
4.1. Zero-shot	15
4.2. Few-shot	17
4.3. Modelo ajustado	19
5. Conclusiones	20
Referencias	20
A. Detalles del desarrollo del trabajo	22

1. Introducción

Para estimar automáticamente la complejidad computacional [1] de un algoritmo, normalmente es necesario ejecutarlo con diferentes tallas de entrada. Otra opción es realizar un análisis manual de la misma mediante cuenta de pasos de programa.

En este trabajo, se pretende investigar el uso de aprendizaje profundo para estimar automáticamente la complejidad computacional de un algoritmo sin que sea necesario ejecutarlo, lo que puede servir de apoyo a los programadores de cara a escribir código más eficiente. Se explorará la capacidad de los grandes modelos de lengua [8] para llevar a cabo esta tarea.

2. Antecedentes

Definición 2.1 (Algoritmo) *Un algoritmo es un conjunto de reglas que permite la resolución de algún tipo de problema.*

Los algoritmos son fundamentales para dar la estructura de una solución sistemática a un determinado problema, para que esta pueda ser replicada y comprendida por otros.

Existen muchos algoritmos capaces de resolver el mismo problema, por lo que para escoger uno específico necesitaremos basarnos en un criterio adecuado.

Definición 2.2 (Complejidad computacional) *La complejidad computacional de un algoritmo es el tiempo empleado por este para ejecutarse y dar un resultado a partir de los datos de entrada.*

El tiempo es un recurso fundamental, por lo que priorizaremos usar un algoritmo que tenga menor complejidad computacional.

Esta definición de complejidad computacional tiene algún problema. Por ejemplo, la complejidad de un algoritmo no es algo constante, sino que, dependiendo de los datos de entrada, esta complejidad podrá variar. Por tanto, no siempre podremos dar un mejor algoritmo para todos los casos posibles de un mismo problema.

En particular, hay algoritmos que varían su complejidad según el tamaño de sus datos de entrada.

Definición 2.3 (Talla) *La talla de un problema corresponde al valor o conjunto de valores asociados a la entrada del problema que representa, normalmente, una medida de su tamaño respecto a otras entradas posibles.*

Algoritmo 1: Sumar elementos de un vector

Entrada: Un vector V de tamaño n **Salida:** Suma de elementos de V

```

1  $suma \leftarrow V[1]$ 
2 para  $i \leftarrow 2$  a  $n$  hacer
3    $suma \leftarrow suma + V[i]$ 
4 fin
5 devolver  $suma$ 

```

Mediremos la complejidad de un algoritmo con una función $T(n)$, que dada una talla $n \in \mathbb{N}$ devolverá el número de pasos que realiza el algoritmo. Para el Algoritmo 1, la talla del problema es el tamaño n del vector V y su complejidad es $T(n) = 5n$.

Aun trabajando con una misma talla, podemos tener un algoritmo que tenga distintas complejidades computacionales según cómo estén configurados sus datos de entrada.

Definición 2.4 (Instancia) *Una instancia de un problema corresponde a todas las configuraciones diferentes de la entrada, de una talla determinada, que dan lugar al mismo comportamiento del algoritmo.*

Algoritmo 2: Contiene cero

Entrada: Un vector V de tamaño n **Salida:** `true` si $0 \in V$, `false` en caso contrario

```

1 para  $i \leftarrow 1$  a  $n$  hacer
2   si  $V[i] = 0$  entonces
3     devolver true
4   fin
5 fin
6 devolver false

```

De esta manera, se hace imposible dar un único valor a la complejidad computacional, por lo que comúnmente se habla de los casos peor, mejor y promedio.

Con el Algoritmo 2 podemos ver que el caso mejor sería encontrar un cero en el primer elemento del vector, lo que conlleva una complejidad de $T(n) = 5$; mientras que en el peor caso no lo encontraría, con una complejidad de $T(n) = 4n + 3$.

A partir de ahora nos vamos a referir siempre al caso peor para el análisis de la complejidad.

Definición 2.5 (Notación O-grande) Sea $f : \mathbb{N} \rightarrow \mathbb{R}^+$. El conjunto de las funciones del orden de $f(n)$, llamado $\mathcal{O}(f(n))$ se define como

$$\mathcal{O}(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \text{ tales que } g(n) \leq c \cdot f(n) \ \forall n \geq n_0\}$$

La notación O-grande, también escrita como $\mathcal{O}(f(n))$, proporciona una cota asintótica superior del crecimiento de una función. La utilizaremos para expresar cómo escala el número de pasos de un algoritmo conforme aumenta la talla del problema, ignorando factores constantes o de menor orden.

Por ejemplo, en el Algoritmo 1, aunque la complejidad exacta sea $T(n) = 5n$, al usar notación O-grande escribimos simplemente que $T(n) \in \mathcal{O}(n)$, ya que existe una constante $c = 5$ tal que $T(n) \leq c \cdot n$ para todo $n \geq 1$.

Algunos órdenes de complejidad comunes, y que vamos a usar más adelante, son:

- $\mathcal{O}(1)$: constante
- $\mathcal{O}(\log n)$: logarítmico
- $\mathcal{O}(n)$: lineal
- $\mathcal{O}(n \log n)$: cuasi lineal
- $\mathcal{O}(n^2)$: cuadrático
- $\mathcal{O}(n^3)$: cúbico
- $\mathcal{O}(a^n)$ con $a \in \mathbb{R}^{++}$: exponencial

3. Grandes modelos de lengua

Los **grandes modelos de lengua** (*Large Language Models*, LLM) [8] se han consolidado como una de las tecnologías más influyentes dentro del campo de la inteligencia artificial. Estos modelos están basados en técnicas de **aprendizaje automático** (*Machine Learning*, ML), concretamente en redes neuronales profundas, y se entrenan sobre enormes volúmenes de datos textuales con el objetivo de modelar el lenguaje humano.

Lo primero que vamos a necesitar es generar un vocabulario V con todas las posibles palabras o símbolos; a cada elemento de este vocabulario lo llamamos **token**.

3.1. Transformers

Una de las claves del éxito de los LLM ha sido la adopción de la arquitectura **Transformer** [13], introducida por Vaswani et al. en 2017. Esta arquitectura permite modelar dependencias a largo plazo en secuencias de texto de manera eficiente mediante mecanismos de atención, superando las limitaciones de modelos anteriores como las redes recurrentes (RNN) o las redes LSTM [4]. Gracias a esta innovación, los Transformers han permitido construir modelos con miles de millones de parámetros, capaces de generar texto coherente, traducir idiomas, resumir información o incluso razonar de forma básica.

Para trabajar con los tokens de entrada necesitamos una representación vectorial de estos, la cual obtendremos a partir de las matrices de incrustación $\mathbf{E} \in \mathbb{R}^{|V| \times d}$, que tiene como fila i la representación vectorial del token i del vocabulario V ; y $\mathbf{E}_{\text{pos}} \in \mathbb{R}^{N \times d}$ que codifica la posición en el contexto de cada token. Esta última matriz \mathbf{E}_{pos} puede generarse con funciones fijas o ser aprendida durante el entrenamiento. A d se le conoce como dimensionalidad del modelo. A partir de los N tokens de entrada, conocidos como contexto, construimos una matriz \mathbf{X} de tamaño $N \times d$ como $\mathbf{X} = \mathbf{E}[\mathbf{i}_1, \mathbf{i}_2, \dots, \mathbf{i}_N] + \mathbf{E}_{\text{pos}}$ siendo $\mathbf{E}[\mathbf{i}_1, \mathbf{i}_2, \dots, \mathbf{i}_N]$ la matriz resultante de tomar las filas i_1, i_2, \dots, i_N de \mathbf{E} que corresponden con las representaciones vectoriales de los tokens de entrada.

La atención es un mecanismo que permite a un modelo enfocar su “atención” en partes relevantes de la entrada al procesar un elemento, en lugar de tratar todos los elementos por igual. Esta atención la mediremos mediante una **cabeza de atención** (*attention head*). Para ello, asignamos tres distintos roles a los tokens de entrada:

- **Consulta** (*Query*, \mathbf{Q}): El elemento actual que está siendo comparado con los anteriores.
- **Clave** (*Key*, \mathbf{K}): Los anteriores elementos que están siendo comparados con el actual para hallar el peso de similitud.
- **Valor** (*Value*, \mathbf{V}): Suma ponderada de los pesos de similitud para el elemento actual.

Estos tres roles los representamos con las matrices de pesos $\mathbf{W}^{\mathbf{Q}}, \mathbf{W}^{\mathbf{K}}$ de tamaño $d \times d_k$ y $\mathbf{W}^{\mathbf{V}}$ de tamaño $d \times d_v$. Estas matrices de pesos dan una proyección de los vectores a los roles consulta, clave y valor:

$$\mathbf{Q} = \mathbf{XW}^{\mathbf{Q}}, \quad \mathbf{K} = \mathbf{XW}^{\mathbf{K}}, \quad \mathbf{V} = \mathbf{XW}^{\mathbf{V}} \quad (1)$$

de tamaños $N \times d_k$ para \mathbf{Q} y \mathbf{K} y $N \times d_v$ para \mathbf{V} . Para la comparación consulta-clave calcularemos la matriz \mathbf{QK}^\top de tamaño $N \times N$, la cual multiplicaremos por $\frac{1}{\sqrt{d_k}}$ para evitar que los coeficientes de la matriz escalen indefinidamente. De esta manera, calculamos:

$$\mathbf{head} = \text{softmax} \left(\text{mask} \left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}} \right) \right) \mathbf{V} \quad (2)$$

$$\mathbf{A} = \mathbf{head} \mathbf{W}^O \quad (3)$$

En el cálculo de \mathbf{head} podemos encontrar una función *mask*, esta es necesaria pues al calcular la matriz \mathbf{QK}^\top estamos comparando keys posteriores al query, lo cual carece de sentido porque no debería poder acceder a tokens futuros. Para evitarlo, introducimos dicha función *mask* que asigna a estas posiciones $-\infty$, de manera que la función *softmax* las transforme en 0. En la práctica, la función *mask* consiste en sumar una matriz $M \in \mathbb{R}^{N \times N}$ triangular superior donde $M_{ij} = -\infty \forall j > i$ y el resto son 0. Además, en el cálculo de \mathbf{A} aparece la matriz $\mathbf{W}^O \in \mathbb{R}^{d_v \times d}$ que usamos para redimensionar \mathbf{head} a un tamaño de $N \times d$.

En la realidad, los Transformers usan múltiples cabezas de atención, cada una de las cuales puede especializarse en distintas tareas mediante la configuración de sus propios parámetros. A esto se le conoce como **atención multi-cabeza** (*multi-head attention*) y sigue un proceso muy similar al cálculo de cada cabeza de atención, pues cada una de estas cabezas se calcula paralelamente al resto. Tomemos un número A de cabezas, para calcular cada \mathbf{head}_i seguiremos el mismo proceso que antes pero para calcular la matriz \mathbf{A} se concatenarán las A cabezas, que indicaremos con el símbolo \oplus ; y, de igual manera que antes, tomaremos la matriz $\mathbf{W}^O \in \mathbb{R}^{Ad_v \times d}$ para ajustar el tamaño de la matriz de salida:

$$\mathbf{Q}_i = \mathbf{XW}_i^K, \quad \mathbf{K}_i = \mathbf{XW}_i^K, \quad \mathbf{V}_i = \mathbf{XW}_i^V \quad (4)$$

$$\mathbf{head}_i = \text{SelfAttention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) = \text{softmax} \left(\text{mask} \left(\frac{\mathbf{Q}_i \mathbf{K}_i^\top}{\sqrt{d_k}} \right) \right) \mathbf{V}_i \quad (5)$$

$$\mathbf{A} = \text{MultiHeadAttention}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \oplus \dots \oplus \mathbf{head}_A) \mathbf{W}^O \quad (6)$$

El cálculo de \mathbf{A} consiste en una de las capas que forman el bloque Transformer; antes y después de esta **capa de atención** (*attention layer*), realizamos una **normalización por**

capas (*layer normalization*) [2]. Para ello, dado un vector \mathbf{x} de dimensión d , se definen la media μ , desviación típica σ y vector normalizado $\hat{\mathbf{x}}$ como

$$\mu = \frac{1}{d} \sum_{i=1}^d x_i \quad (7)$$

$$\sigma = \sqrt{\frac{1}{d} \sum_{i=1}^d (x_i - \mu)^2} \quad (8)$$

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu}{\sigma} \quad (9)$$

de forma que se resta μ y se divide por σ a cada componente del vector \mathbf{x} . Finalmente, en la implementación de la normalización por capas, se introducen dos parámetros aprendibles durante el entrenamiento, γ y β , que representan los valores de escala y desplazamiento, respectivamente.

$$\text{LayerNorm}(\mathbf{x}) = \gamma \frac{\mathbf{x} - \mu}{\sigma} + \beta \quad (10)$$

En nuestro caso, aplicaremos LayerNorm a \mathbf{X} , recordamos que tiene tamaño $N \times d$, por lo que actuará independientemente sobre cada fila de \mathbf{X} .

A otra de las capas se le conoce como **capa de propagación directa** (*feedforward layer*, FFL). Esta capa consiste fundamentalmente en una red neuronal completamente conectada (*fully connected*) con una capa oculta de dimensionalidad d_{ff} y dos matrices de peso $\mathbf{W}_1 \in \mathbb{R}^{d \times d_{ff}}$ y $\mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d}$. Aplicaremos la función

$$\text{FFN}(\mathbf{x}) = \text{ReLU}(\mathbf{x}\mathbf{W}_1 + b_1)\mathbf{W}_2 + b_2 \quad (11)$$

donde ReLU es la función de activación $\text{ReLU}(x) = \max(0, x)$ que se aplica a cada componente del vector \mathbf{x} y b_1 y b_2 son vectores de sesgo de dimensiones d_{ff} y d , respectivamente. De nuevo, esta función se aplica a \mathbf{X} , por lo que la función FFN se aplicará fila a fila.

Tras aplicar cada capa, se suma la entrada que se le proporcionó a la capa con la salida de dicha capa. De esta manera, podemos compactar todos estos procesos en dos expresiones:

$$\mathbf{O} = \mathbf{X} + \text{MultiHeadAttention}(\text{LayerNorm}(\mathbf{X})) \quad (12)$$

$$\mathbf{H} = \mathbf{O} + \text{FFN}(\text{LayerNorm}(\mathbf{O})) \quad (13)$$

También podemos descomponer los cálculos en cinco componentes que llamaremos \mathbf{T}_i , de tamaño $N \times d$, de manera que se pueda ver de forma secuencial:

$$\mathbf{T}_1 = \text{LayerNorm}(\mathbf{X}) \quad (14)$$

$$\mathbf{T}_2 = \text{MultiHeadAttention}(\mathbf{T}_1) \quad (15)$$

$$\mathbf{T}_3 = \mathbf{T}_2 + \mathbf{X} \quad (16)$$

$$\mathbf{T}_4 = \text{LayerNorm}(\mathbf{T}_3) \quad (17)$$

$$\mathbf{T}_5 = \text{FFN}(\mathbf{T}_4) \quad (18)$$

$$\mathbf{H} = \mathbf{T}_5 + \mathbf{T}_3 \quad (19)$$

En un modelo Transformer, varios bloques se apilan en profundidad. La salida de un bloque se convierte en la entrada del siguiente, de manera que el bloque k tiene como entrada \mathbf{H}^{k-1} .

Por último, el elemento necesario para poder procesar el lenguaje natural es la **cabeza de modelado del lenguaje** (*language modeling head*). En esencia, los LLM son predictores de palabras. Por tanto, dado un contexto, podemos calcular la probabilidad de cada palabra del vocabulario de ser la siguiente palabra. Por ejemplo:

$$P(\text{Madrid} | \text{La capital de España es})$$

calcula la probabilidad de que el modelo afirme que Madrid es la capital de España. De esta manera podemos dar una distribución de probabilidad para todos los tokens del vocabulario V .

$$P(w | \text{La capital de España es})$$

De calcular estas probabilidades se encarga la cabeza de modelado del lenguaje, las cuales calcula a partir de la última fila de la salida del último bloque, es decir, de \mathbf{H}_N^L . Necesitaremos una matriz de desincrustación \mathbf{U} , que podemos aprender durante el entrenamiento o tomar como $\mathbf{U} = \mathbf{E}^\top$. De esta manera calculamos:

$$\mathbf{u} = \mathbf{H}_N^L \mathbf{E}^\top \quad (20)$$

$$\mathbf{y} = \text{softmax}(\mathbf{u}) \quad (21)$$

La función softmax es la que convierte los valores de \mathbf{u} en probabilidades en \mathbf{y} . Así, podemos empezar a generar texto.

3.2. Generación de texto

Una vez tenemos las probabilidades de cada token, debemos escoger un criterio para elegir el token que sucederá al contexto. Hay varias estrategias que se pueden seguir, pero la más sencilla es el *greedy decoding*. Esta técnica consiste en elegir siempre el token con mayor probabilidad.

$$\hat{w}_i = \text{argmax}_{w \in V} P(w|w_{<i}) \quad (22)$$

Si bien esta sugiere ser la solución localmente óptima, puede generar textos más genéricos y predecibles, además de ser una generación determinista y repetitiva. Esto lo podemos evitar con otra técnica conocida como **muestreo top- p** (*top- p sampling*), la cual consiste en truncar la distribución de probabilidad para quedarnos solamente con el percentil p de tokens más probables. Es decir, ordenamos todos los tokens de mayor a menor probabilidad según la distribución $P(w|w_{<i})$ y así obtenemos el vocabulario top- p $V^{(p)}$, que es el menor conjunto de estas palabras tal que

$$\sum_{w \in V^{(p)}} P(w|w_{<i}) \geq p \quad (23)$$

Una vez calculado $V^{(p)}$ elegiremos aleatoriamente el token según la distribución de probabilidad resultante. Así aportaremos variabilidad a la generación de texto y obtendremos resultados más orgánicos. Existen otros métodos de muestreo más sofisticados, orientados a tareas específicas; sin embargo, no existe un método de muestreo perfecto y la elección dependerá de la tarea a realizar.

3.3. Entrenamiento

El entrenamiento de los modelos LLM consiste en ajustar parámetros como las matrices de pesos que hemos estado definiendo anteriormente. Para esto necesitamos un corpus masivo de texto, que puede incluir libros, artículos, conversaciones en línea, páginas web,

entre otros. Para hacerlo primero necesitamos una función de pérdida, vamos a usar la función de **pérdida de entropía cruzada** (*cross-entropy loss*):

$$L_{CE} = - \sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w] \quad (24)$$

donde $\mathbf{y}_t[w]$ representa un vector de ceros (con dimensión $|V|$) exceptuando un 1 en la posición de la siguiente palabra correcta, mientras que $\hat{\mathbf{y}}_t[w]$ representa el vector de probabilidades de ser la siguiente palabra para cada token del vocabulario V . De esta manera, se puede simplificar como

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = - \log \hat{\mathbf{y}}_t[w_{t+1}] \quad (25)$$

El objetivo es minimizar esta función de pérdida, por lo que usaremos el método del descenso del gradiente. El problema de esto es calcular el gradiente de la función de pérdida, pues al tener una estructura con tantas capas dificulta el cálculo del gradiente

$$\frac{\partial L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t)}{\partial \theta} \quad (26)$$

en función de los parámetros θ del modelo. Para calcular el gradiente se usan técnicas de **retropropagación** [12], que mediante el uso de la regla de la cadena se propaga hasta el inicio del Transformer. Otros algoritmos más eficientes que el descenso del gradiente para actualizar los parámetros de modelos basados en Transformers son Adam [6] y AdamW [7]. A este entrenamiento se le conoce como **aprendizaje autosupervisado** (*self-supervised learning*, SSL).

Comúnmente, los conjuntos de texto usados para entrenar los modelos son recogidos mediante *web scraping*, es decir, se usan *bots* automatizados para acceder a páginas web y recopilar la información que encuentren en ellas; o proporcionados por terceros, como es el caso de Common Crawl (<https://commoncrawl.org/>), un repositorio gratuito que recoge información de miles de millones de páginas web desde 2008.

3.4. Evaluación

Existen numerosas formas de evaluar un modelo LLM, algunas basadas en sus fundamentos y otras basadas en sus aplicaciones. Una de estas métricas es la **perplejidad** (*perplexity*), que mide la capacidad del modelo para predecir una secuencia de texto. Cuanto menor sea su valor, mayor es el rendimiento del modelo.

$$\text{Perplexity}_\theta(w_{1:n}) = P_\theta(w_{1:n})^{-\frac{1}{n}} = \sqrt[n]{\prod_{i=1}^n \frac{1}{P_\theta(w_i|w_{<i})}} \quad (27)$$

Para tareas específicas como la traducción automática se utiliza BLEU (Bilingual Evaluation Understudy) [10], para resumen de textos se usa ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [9] y para tareas de clasificación o pregunta-respuesta se suelen utilizar la accuracy y F1 [11].

3.5. Ajuste fino

El **ajuste fino** (*fine-tuning*) es un proceso de entrenamiento adicional que consiste en adaptar un modelo de lenguaje previamente preentrenado a una tarea o dominio específico mediante la actualización de sus parámetros, optimizando el rendimiento sin necesidad de entrenar un modelo desde cero [5].

3.6.

El **aprendizaje en contexto** [3] es una capacidad emergente de grandes modelos de lengua mediante la cual el modelo aprende a realizar tareas nuevas sin necesidad de actualizar sus parámetros, simplemente a partir de ejemplos proporcionados en la propia entrada (*prompt*). Es decir, el modelo generaliza el patrón de una tarea dada observando solo unos pocos ejemplos en el contexto del *prompt*, sin entrenamiento adicional. A la estrategia de incluir ejemplos en el *prompt* se le conoce como *few-shot*.

A diferencia del ajuste fino, donde se modifican explícitamente los pesos del modelo, en el aprendizaje en contexto la “adaptación” se realiza implícitamente a través del mecanismo de atención del Transformer, utilizando los ejemplos previos en la entrada para ajustar su comportamiento temporalmente.

4. Experimentos

El objetivo de los experimentos es evaluar el rendimiento de los grandes modelos de lengua a la hora de clasificar la complejidad computacional de un algoritmo. Para ello, vamos a trabajar con el modelo open-source Llama 3.1 8B Instruct, el cual cuenta con 8 mil millones de parámetros y está preentrenado para seguir instrucciones. La genera-

ción de texto será por medio de *greedy decoding* para asegurar la reproducibilidad de los experimentos.

Para evaluar el modelo usaremos el conjunto de datos de CodeComplex, formado por 4769 fragmentos de código de Python. Estos fragmentos de código tienen complejidades computacionales entre: constante, logarítmica, lineal, cuasi lineal, cuadrática, cúbica y exponencial. Las métricas de evaluación serán la accuracy y F1.

El código usado para llevar a cabo los experimentos se encuentra en el repositorio de GitHub: <https://github.com/DSPagan/llms-computational-complexity>

Estudiaremos cómo afecta el prompt a las predicciones del modelo en las siguientes fases:

- **Zero-shot:** El modelo recibe el código a analizar y distintas instrucciones para mejorar su desempeño.
- **Few-shot:** Además del código y las instrucciones, el modelo recibe un ejemplo de cada una de las clases de complejidad que hay en el conjunto de datos.
- **Modelo ajustado:** El modelo se somete a un proceso de ajuste fino, tras el que recibe el código y las instrucciones.

Para ello, se dividirá el conjunto de datos en dos conjuntos: *train* (90 %) y *test* (10 %). Para hacer esta división, se han considerado dichos porcentajes en cada una de las distintas complejidades del conjunto de datos, de manera que las clases queden equilibradas en ambos conjuntos. El conjunto *train* lo usaremos para dar los ejemplos en *few-shot* y para hacer ajuste fino, mientras que los códigos del conjunto *test* serán en los que evaluaremos el modelo en las tres fases.

4.1. Zero-shot

Se han probado distintos *prompts* con el objetivo de estudiar cómo este afecta al rendimiento del modelo. Indicaremos por [CÓDIGO] el lugar donde se incluye el código a analizar por el modelo.

Prompt 1 zero-shot

Give the time complexity of the code:
[CÓDIGO]

Prompt 2 zero-shot

Analyze the time complexity of the following code.

Choose exactly one of the following options: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ or exponential ($O(2^n)$, $O(3^n)$, etc.).

Give the time complexity of the code:

[CÓDIGO]

Prompt 3 zero-shot

You are an expert in algorithm analysis and time complexity.

Your task is to estimate the time complexity of the given code snippet enclosed by '///'.

Analyze the time complexity of the following code.

Choose exactly one of the following options: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ or exponential ($O(2^n)$, $O(3^n)$, etc.).

Give the time complexity of the code:

///

[CÓDIGO]

///

Prompt	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	exponencial	Accuracy	F1
Prompt 1	0.419	0.250	0.650	0.480	0.587	0.241	0.128	0.418	0.407
Prompt 2	0.546	0.318	0.750	0.494	0.290	0.364	0.200	0.450	0.468
Prompt 3	0.546	0.415	0.750	0.468	0.238	0.259	0.167	0.436	0.445

Tabla 1: Accuracy de la predicción de las clases de complejidad con zero-shot

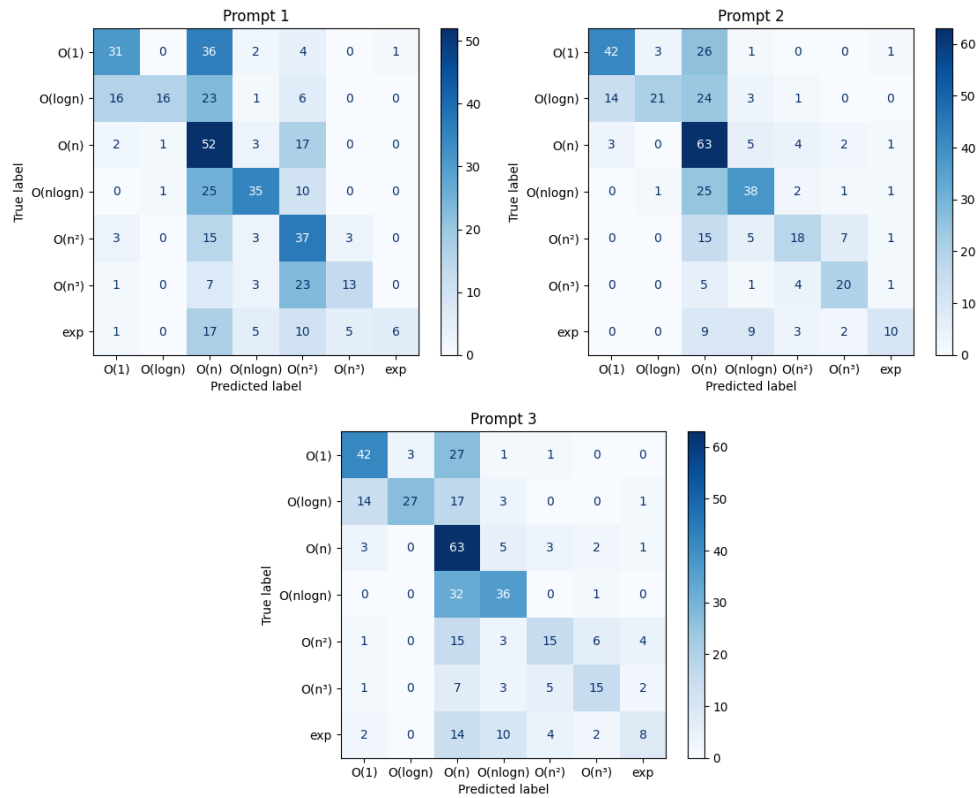


Figura 1: Matriz de confusión haciendo zero-shot con distintos prompts

4.2. Few-shot

Indicaremos por [CÓDIGO] el lugar donde se incluye el código a analizar por el modelo y por [EJEMPLO] el lugar donde se incluyen los distintos códigos de ejemplo.

Prompt few-shot

Analyze the time complexity of the following code.

Choose exactly one of the following options: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ or exponential ($O(2^n)$, $O(3^n)$, etc.).

Here are some examples:

Example 1:

Code:

[EJEMPLO]

Complexity: $O(1)$

Example 2:

Code:

[EJEMPLO]

Complexity: $O(\log n)$

Example 3:

Code:

[EJEMPLO]

Complexity: $O(n)$

Example 4:

Code:

[EJEMPLO]

Complexity: $O(n \log n)$

Example 5:

Code:

[EJEMPLO]

Complexity: $O(n^2)$

Example 6:

Code:

[EJEMPLO]

Complexity: $O(n^3)$

Example 7:

Code:

[EJEMPLO]

Complexity: exponential

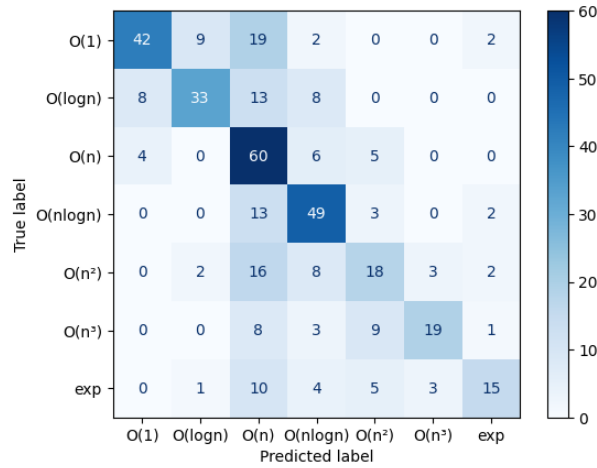
Now analyze this code:

Code:

[CÓDIGO]

Complexity:

$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	exponencial	Accuracy	F1
0.553	0.500	0.714	0.628	0.277	0.333	0.306	0.497	0.518

Figura 2: Accuracy de la predicción de las clases de complejidad con *few-shot***Figura 3:** Matriz de confusión haciendo *few-shot*

4.3. Modelo ajustado

Para el modelo ajustado se ha escogido el *prompt* 2 usado en *zero-shot* pues presenta el mejor rendimiento y ya no son necesarios los ejemplos de *few-shot*. El modelo ha sido ajustado tres veces con distintos números de pasos: 60 pasos, 1 época (537 pasos) y 2 épocas (1074 pasos).

Entrenamiento	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	exponencial	Accuracy	F1
60 pasos	0.714	0.697	0.702	0.551	0.477	0.807	0.714	0.662	0.665
1 época	0.883	0.985	0.833	0.872	0.862	0.965	0.959	0.901	0.906
2 épocas	0.857	0.939	0.917	0.885	0.892	1.000	0.918	0.912	0.918

Figura 4: Accuracy de la predicción de las clases de complejidad haciendo ajuste fino

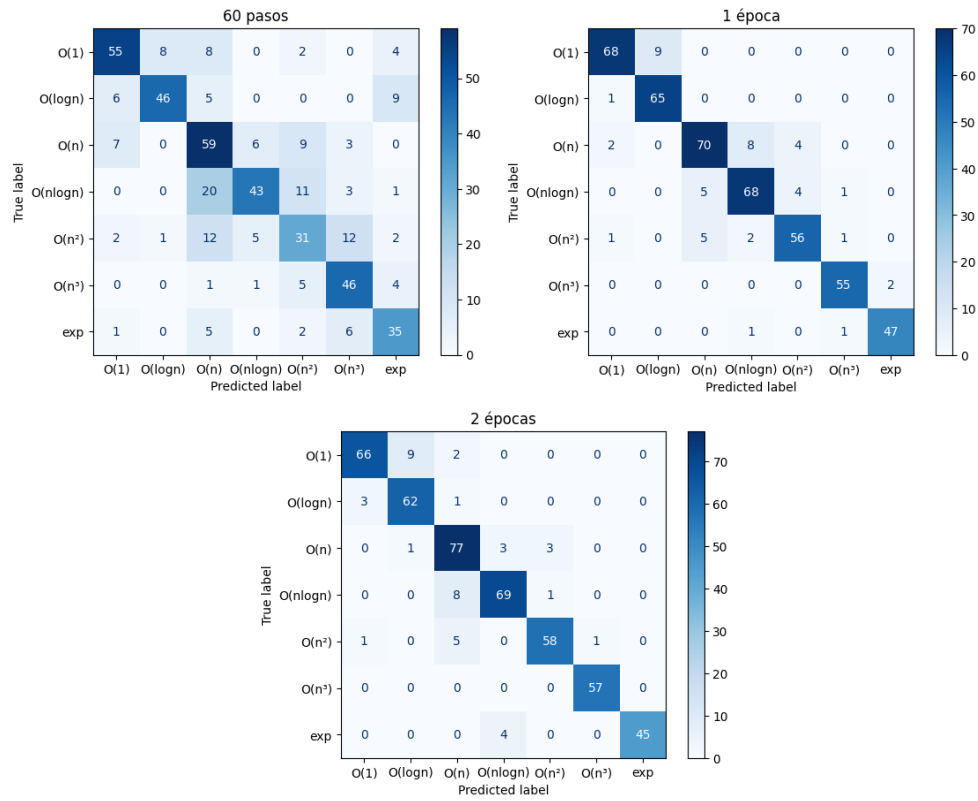


Figura 5: Matrices de confusión haciendo ajuste fino

5. Conclusiones

Referencias

- [1] Albert, J. V., Rabasa, F. J. F., & Quetglás, G. M. (1998). *Introducció a l'anàlisi i disseny d'algorismes*. Universitat de València.
- [2] Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). *Layer normalization*. arXiv. <https://arxiv.org/abs/1607.06450>
- [3] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33, 1877–1901. <https://arxiv.org/abs/2005.14165>

- [4] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780. <https://doi.org/10.1162/neco.1997.9.8.1735>
- [5] Howard, J., & Ruder, S. (2018). Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)* (pp. 328–339). <https://doi.org/10.18653/v1/P18-1031>
- [6] Kingma, D. P., & Ba, J. (2014). *Adam: A method for stochastic optimization*. arXiv. <https://arxiv.org/abs/1412.6980>
- [7] Loshchilov, I., & Hutter, F. (2019). *Decoupled weight decay regularization*. arXiv. <https://arxiv.org/abs/1711.05101>
- [8] Jurafsky, D., & Martin, J. H. (2025). *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition with language models* (3rd ed.). Manuscript in preparation. <https://web.stanford.edu/~jurafsky/slp3>
- [9] Lin, C.-Y. (2004). ROUGE: A package for automatic evaluation of summaries. In *Proceedings of the Workshop on Text Summarization Branches Out (WAS 2004)* (pp. 74–81). <https://aclanthology.org/W04-1013/>
- [10] Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)* (pp. 311–318). <https://doi.org/10.3115/1073083.1073135>
- [11] Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016). SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 2383–2392). <https://aclanthology.org/D16-1264/>
- [12] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533–536. <https://doi.org/10.1038/323533a0>
- [13] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems* (Vol. 30). https://papers.nips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

A. Detalles del desarrollo del trabajo