UNIVERSIDAD EUROPEA DE MADRID

ESCUELA DE ARQUITECTURA, INGENIERÍA Y DISEÑO

MÁSTER UNIVERSITARIO EN ANÁLISIS DE DATOS MASIVOS

TRABAJO FIN DE MÁSTER

Decodificación gramática en arquitecturas Transformers

Rodrigo A. García Casasola

Dirigido por

Andrés Soto Villaverde

CURSO 2023 - 2024

TÍTULO: Decodificación gramática en arquitecturas Transformers

AUTOR: Rodrigo A. García Casasola

TITULACIÓN: MÁSTER UNIVERSITARIO EN ANÁLISIS DE DATOS MASIVOS

DIRECTOR DEL PROYECTO: Andrés Soto Villaverde

FECHA: Octubre de 2025

RESUMEN

Este proyecto propone estudiar y comprender en profundidad las soluciones propuestas a la problemática de dotar de estructura a la generación estocástica de los modelos lingüísticos. Para ello se propone estudiar estas soluciones implementadas (GBD) y usadas extendidamente, integrarlas en un framework industrial de inferencia de modelos (vLLM) y aplicarlas a modelos frontera de código libre (Llama 3 8B y Llama 3 70B).

Se propone establecer una experimentación de estas ténicas en los modelos propuestos, construyendo métricas para su posterior comparación, y monitorizando los recursos hardware usados para entender el problema y el cómo optimizarlo para mejorar los resultados obtenidos. Esta experimentación constará de 3 experimentaciones para 100 muestra por cada modelo, estas 3 experimentaciones serán las técnicas de decodificación gramátical propuestas. Midiendo cómo varía la correctitud sintáctica y semántica de las generaciones obtenidas de cada modelo.

También se incluye un análisis de los recursos hardware usados: el porcentaje de uso de las GPUs y de la memoria total disponible.

Finalmente se discutirán las conclusiones obtenidas a la luz de los resultados y se propondrán futuros trabajos en base al establecido en este proyecto.

Palabras clave: Machine Learning, modelos lingüísticos, decodificación gramatical, Llama 3, vLLM, prompt tunning, FewShots, Hiperparámeter tunning.

ABSTRACT

This project proposes to study and gain a deep understanding of the existing solutions to address the challenge of structuring the stochastic generation of language models. To this end, it suggests examining these implemented solutions (GBD) that are widely used, integrating them into an industrial inference framework (vLLM), and applying them to open-source frontier models (Llama 3 8B and Llama 3 70B).

The project aims to set up experimentation with these techniques on the proposed models, building metrics for subsequent comparison and monitoring hardware resources used to understand the problem and how to optimize it to improve results. This experimentation will consist of three experiments with 100 generations per model, focusing on the proposed grammatical decoding techniques. It will measure how the syntactic and semantic accuracy of each model's generations varies.

Additionally, an analysis of hardware resources used will be included, covering GPU utilization and total memory availability.

Finally, the conclusions will be discussed in light of the results, and future work will be proposed based on the foundations established in this project.

Keywords: Machine Learning, Large Language Models (LLM), Grammar-based-decoding (GBD), Llama 3 family models, prompt tunning, Fewshots tunning, Hyperparameter tunning.

We'll figure it out.

A ti, por todo.

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1. RESUMEN DEL PROYECTO

1.1 Contexto y justificación

Este proyecto[5] busca estudiar en profundidad el problema actual de la decodificación estructurada en los modelos Transformers[24], que se han desarrollado exponencialmente en el ámbito del Machine Learning durante esta pasada década. Se tratan los conceptos teóricos básicos para comprender el contexto en profundidad, se expone la problemática a resolver y se plantea una experimentación usando un framework de inferencia industrial (vLLM)[19] para establecer una comparativa entre las actuales ténicas de decodificación y las ténicas de decodificación gramatical[12], [13], [22], [16] (o estructurada) actualmentes utilizadas en el ámbito industrial en distintos modelos frontera relevantes.

También se busca establecer un punto de inicio acerca de la experimentación con modelos lingüísticos, cómo medirlos, monitorizarlos y establece métricas solidas y fiables para su experimentación.

Se explora la gestión de los recursos disponibles para realizar la experimentación mediante técnicas de monitorización para el uso de la memoria y la GPU que permitirán optimizar el uso de los mismo.

1.2 Planteamiento del problema

El problema a resolver es la decodificación estructurada y determinista para la generación de los modelos lingüísticos, la cual de manera nativa es estocástica y variable. Para ello se estudian las principales técnicas industriales propuestas para resolver este problema, aplicadas a modelos frontera relevantes en el mercado.

También se plantea la monitorización y el análisis de métricas para medir los resultados obtenidos, el planteamiento de una experimentación eficaz y la optimización de los recursos hardware utilizados para afrontar el problema.

1.3 Objetivos del proyecto

El objetivo principal de este proyecto es establecer unas métricas y referencias para conocer qué técnicas actuales son las más eficaces para resolver el problema propuesto. Enfocado a un ámbito real e industrial.

Como objetivo secundarios, se establece el estudio de los modelos Transformers, técnicas de decodificación, planteamiento de una experimentación con métricas y medidas eficaces para poder comparar estas ténicas en distintos modelos. Así como la monitorización y optimización de los recursos disponibles para afrontar la problemática.

1.4 Resultados obtenidos

De manera constrastada, se concluye con que las técnicas actuales de decodificación gramatical mejoran y aseguran una generación acorde a unas reglas establecidas, tanto a nivel sintáctico como semántico. Además se concluye que el uso de modelos con un número de parámetros mayor de manera general contribuye a unos mejores resultados, a costa de perder

velocidad y rapidez en la generación de los mismos.

1.5 Estructura de la memoria

La parte más importante y densa de la memoria se localiza en: Antecedentes teóricos y estado del arte, en donde se establecen las bases teóricas necesarias para entender la problemática además de analizar y estudiar las actuales propuestas de solución para el problema.

Y en el desarrollo, en donde se constituye la experimentación a realizar, se proponen los recursos y métricas a usar, y se desarrollan y explican de manera coherente todas las decisiones tomadas a la hora de realizar la experimentación.

En los objetivos se constatan los principales objetivos planteados para este proyecto los cuales se tratarán de cubrir en el desarrollo del proyecto. Se comentan y enuncian los resultados obtenidos en la parte de discusión, en dónde se encuentran ordenados por modelos utilizados.

En las conclusiones se realiza una retrospectiva en base a los objetivos y lo aprendido en el proyecto dividiendose en conclusiones del proyecto y las personales.

En las futuras líneas de trabajo se plantean mejoras y posibles estudios a plantear en base a lo establecido en el proyecto, como mejoras, nuevos parámetros a estudiar, nuevos enfoques.

Y por último se encuentra la bibliografía y los anexos, donde reside el código utilizado en la experimentación y las gráficas obtenidas durante la misma.

Capítulo 2. ANTECEDENTES / ESTADO DEL ARTE

2.1 Antecedentes teóricos

En este apartado se introducirá todos los conceptos teóricos necesarios para entender el desarrollo del problema y del proyecto.

2.1.1. Modelos lingüisticos

En los últimos años, desde el lanzamiento de *Attention is All You Need* [24] los modelos lingüisticos, en concreto aquellos basados en la arquitectura *Transformer* han ido adquiriendo notoria importancia tanto en la academia como en el ámbito industrial: GPT-3 [4], LLama [9], Gemini [23], Claude [2],etc...

Estos modelos presentan altas capacidades de abstracción y comprensión de la información, llegando a realizar una amplia gama de tareas utilizando distintos modelos de información.

En este proyecto se trabajará con la primera versión unimodal *(text-to-text)* que basa su actividad en la abstracción de la tarea del procesamiento del lenguaje. El cual de manera autoregresiva consigue generar texto de manera coherente.

2.1.2. Arquitectura

La arquitectura que se presentará es la arquitectura actual por antonomasia, la arquitectura *decoder-only*. Existe una amplia variedad en cuanto a la arquitectura del *Transformer*, mezclando bloques como *encoder-decoder*[24], *encoder-only*[8] y optimizaciones de dichas arquitecturas. A continuación se recorrerá de manera general cada una de sus partes:

Un bloque del *Transfomer* inicia con la codificación del texto en tokens, capturando su representación y significado gracias a la *embbeding table*. Esta se trata de una matriz de valores *floats* W_E, que mapea el token t y expande su representación a la de un vector x_0. Este vector captura la información semántica del token y su posición en la secuencia.

$$x_0 = W_E t \tag{2.1}$$

El vector x_0 generado en el paso anterior se propagará a través de lo que se conoce como conexiones residuales[14]. Estas sirven como método de optimización durante las fases del entrenamiento ayudando con problemáticas como el *vanishing gradients* a la vez que permiten el flujo de información entre el resto de bloques.

Estas conexiones residuales permiten que el vector x_0 pase a las distintas capas de atención, formadas a su vez por varias cabezas de atención *(attention heads)*. Cada cabeza de atención, $h_0, h_1, \ldots, h_h$, aplica un mecanismo de atención, que permite que el modelo enfoque

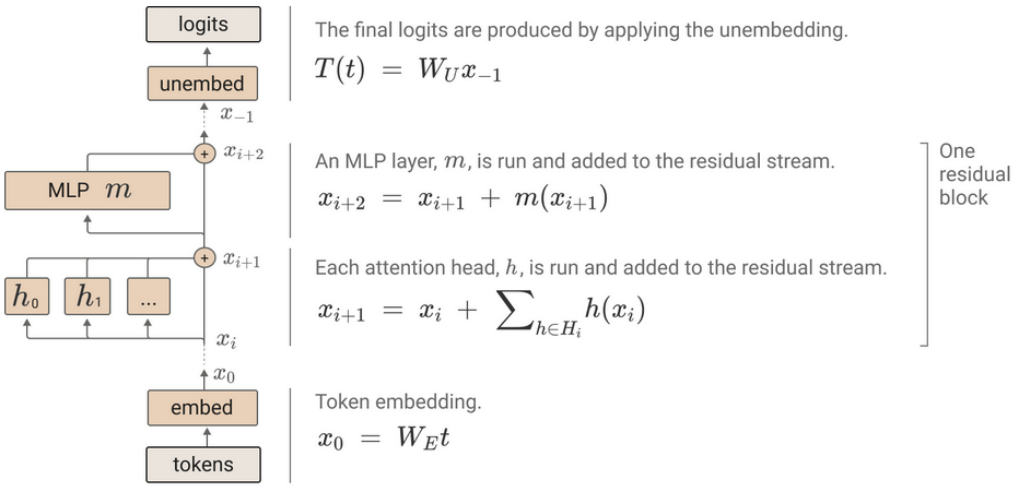

Figura 2.1. Bloque de decodificación del Transformer

diferentes partes de la secuencia de entrada para capturar dependencias a largo plazo entre palabras.

$$x_{i+1} = x_i + \sum_{h \in H_i} h(x_i) \tag{2.2}$$

Aquí, h_i representa el conjunto de todas las cabezas de atención en esa capa, y cada cabeza genera una representación ponderada de los tokens anteriores en función de su relevancia para el token actual. Esto permite que el modelo capture la relación entre el token actual y los tokens anteriores, mejorando la coherencia y el contexto en el texto generado. Como se aprecia, el resultado de cada capa de atención es la suma entre el vector original X_i y el resultado de dicha capa.

El resultado x_{i+1} pasa a continuación a una capa *Multi-Layer Perceptron* (MLP) que le permite capturar patrones complejos en los datos, aportando no linearidad. El resultado de la misma $m(x_{i+1})$ se suma al de la conexión residual, que ayuda a preservar la información a lo largo de las capas del modelo mejorando su capacidad de aprendizaje y generalización.

$$x_{i+2} = x_{i+1} + m(x_{i+1}) \tag{2.3}$$

Tras varias capas de atención con sus correspondientes cabezas de atención y sus capas de MLP, el vector final x_{-1} se proyecta de vuelta al espacio de tokens originales usando una matriz de *un-embeddings* W_U. Esta proyección, también llamada decodificación genera *logits*; que no son más que una representación numérica de las probabilidades para todos los tokens disponibles en el vocabulario. Estos generalmente son normalizados usando una función de tipo softmax de la que se obtiene una distribución de probabilidad acerca de cuales son los tokens siguientes más probables.

$$T(t) = W_U x_{-1} \tag{2.4}$$

Finalmente, el token seleccionado se utiliza como entrada para el siguiente paso en la generación de la secuencia, permitiendo que el proceso continúe hasta completar la secuencia.

Una vez expuesta la arquitectura y flujo de datos a gran escala. Se continuará introduciendo con más detalle cada uno de sus mecanismos más importantes:

2.1.3. Mecanismo de codificación

El proceso comienza con la codificación de la secuencia de entrada (*input*) en tokens. Este proceso se denomina *tokenización*. Para definirlo es necesario:

Establecer un mecanismo de tokenización: La tokenización realizada en la implementación original del Transformer se denomina *byte-pair Encoding*, En ella se comienza por tokenizar el texto en carácteres indiduales para a continuación combinar los pares más comúnes en nuevos tokens.

- Definir un tamaño de vocabulario: Que acotará el número máximo de tokens a utilizar. Este valor puede interpretarse como un hiperparámetro a descubrir mediantes algoritmos de aprendizaje automático.
- Generar el vocabulario: Mediante un *dataset* de entrenamiento para determinar la frecuencia de las palabras, se iterará siguiendo el mecanismo de tokenización propuesto hasta alcanzar el tamaño máximo.
- A cada token se le asigna un ID único que servirá para mapear los inputs con sus representaciones numéricas.

Una vez obtenida la tokenización de los inputs en $t_1, t_2, t_3, ..., t_n$, se procede a codificar los mismos en valores numéricos denominados *embeddings*. Estos capturan la información semántica de cada token. Esta representación se obtiene multiplicando cada token t_n por una matriz de pesos $W_e t$ como muestra la ecuación 2.1.

Debido a su carácter autoregresivo, es necesario añadir información explicita acerca del orden en el que se encuentra el token en la secuencia, para ello se añade un vector de codificación posicional a cada embedding, consiguiendo así capturar también las relaciones basadas en las posiciones de los tokens.

La codificación posicional se define mediante funciones seno y coseno con diferentes frecuencias, de tal manera que cada posición tiene una representación única.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right), PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{2.5}$$

Donde pos es la posición del token en la secuencia, i es la dimensión del vector posición y d_{model} es la dimensión de los embeddings.

Para finalizar se suman ambos *embeddings* dando lugar al vector z_0 que contiene toda la información necesaria codificada.

$$z_0 = x_0 + PE \tag{2.6}$$

2.1.4. Mecanismo de atención

El mecanismo de atención (en concreto, *Scaled-dot product attention*) se articula con tres componentes principales: Q *querys*, K *keys* y V *values* y las siguientes operaciones:

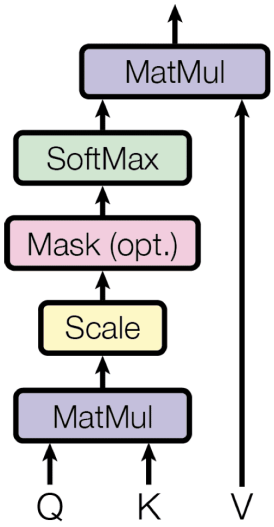

Figura 2.2. Operaciones del Scaled-dot product

Para cada token, se genera su vector Qi y se correlaciona con el resto de vectores Ki generados para cada token del resto de la secuencia. Para calcular esta correlación se utiliza el producto escalar:

$$e_{q_i,k_i} = q_i \cdot k_i \tag{2.7}$$

Al resultado se le aplica una operación softmax para generar los pesos, lo que permitirá descartar correlaciones débiles

$$\alpha_{q_i,k_i} = \text{softmax}(e_{q_i,k_i}) \tag{2.8}$$

El valor de la atención para un token i y el resto de la secuencia se calcula como una suma ponderada del vector V_i por el peso calculado previamente:

$$\text{attention}(q, K, V) = \sum_i V_i \cdot \alpha_{q_i,k_i} \tag{2.9}$$

De manera más general, la ecuación para el *Scaled-dot product attention* sería la siguiente:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}} + b\right)V \tag{2.10}$$

En ella podemos distingir el producto escalar entre Q y K, un factor de normalizacion $\sqrt{d_k}$ que mantiene el proceso de la atención estable al normalizar los valores obtenidos por el pro-

ductos escalar y controlar la agresividad del softmax (Ya que sin ello concentraría la atención en muy pocos tokens) y un bias b que permite ajustar el mecanismo de atención para calibrar mejor las posibles relaciones entre tokens.

Cabe destacar que el conjunto anterior de operaciones se realiza en una única cabeza de atención. Cada capa de atención realiza multiples operaciones similares sobre distintas proyecciones (o subespacios) de la secuencia, permitiendo al modelo descubrir y aprender un mayor número de relaciones diferentes.

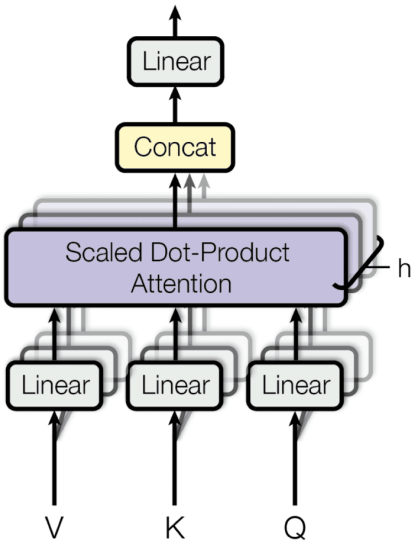

Figura 2.3. Conjunto de cabezas de atención

Nota: El mecanismo de atención fue introducido para resolver la problemática que causaba el usar un tamaño fijo para el vector de codificación, ya que en secuencias largas o complejas, el decodificador no podía acceder a toda la información ya forzaba a usar el mismo tamaño tanto para estas secuencias como para secuencias más simples o cortas.[3]

2.1.5. Mecanismo de decodificación

Se retoma el proceso de generación de tokens hasta el momento en el que el vector inputs procedente de la última capa de atención pasa por la MLP final. (2.3) Ampliando la ecuación anterior con las operaciones realizadas en el MLP obtenemos:

$$m(x) = MLP(x) = ReLU(xW_1 + b_1)W_2 + b_2 \tag{2.11}$$

Donde: W_1 y W_2 son matrices de pesos. b_1 y b_2 son los biases y $ReLU(x)$ es la activación no lineal ReLU.

Esta operación permite al modelo aprender representaciones más ricas y complejas de los datos, mejorando su capacidad para capturar relaciones no lineales entre los tokens y las características que están presentes en la secuencia de entrada.

Después de expandir el espacio de representación, los resultados se vuelven a proyectar a la dimensión original (espacio de tokens originales o espacio del vocabulario). Este paso puede verse como un proceso de compresión o resumido de la información compleja que se aprendió en el espacio de mayor dimensionalidad. El objetivo de esta proyección de regreso es mantener la información más importante que se descubrió en el espacio expandido, pero ahora en una representación más compacta, adecuada para las siguientes capas del modelo.

Este proceso se realiza mediante la matriz de unembedding W_U. Esta matriz mapea los vectores internos de la red de vuelta al espacio del vocabulario, generando logits (valores no normalizados) que representan la probabilidad de cada token en el vocabulario: (2.4)

$$T(t) = W_U x_{-1} \tag{2.12}$$

Los logits obtenidos se pasan a través de una función softmax para convertirlos en probabilidades:

$$P(t) = \text{softmax}(W_U) \tag{2.13}$$

Después de obtener las probabilidades $P(t)$ sobre el vocabulario, se selecciona el token con mayor probabilidad (o se utiliza algún método de muestreo, como top-k sampling o nucleus sampling) para generar el siguiente token en la secuencia.

Este proceso se repite concatenando los tokens generados a la secuencia de entrada (*input*) hasta que se alcanza una condición de parada (como un token de fin de secuencia, o un límite de longitud). Este proceso de unembedding permite que el modelo proyecte las representaciones internas aprendidas de vuelta al espacio original del vocabulario, permitiendo predecir cuál es el siguiente token en la secuencia.

2.2 Estado del arte

Un matiz importante a destacar es que, como se ha visto en el apartado anterior, el proceso de generación de tokens es uno de naturaleza estocástica. El problema principal a tratar es dotar de una estructura formal a la inferencia producida por un modelo lingüistico, convirtiéndola en determinista para una serie de tokens dados. El estado de arte que lo afronta es:

2.2.1. Grammar-Constrained Decoding

La aproximación más implementada es la limitación de los tokens que puede generar un modelo en el proceso de decodificación de los mismos. Esto se consigue definiendo primeramente una gramática formal que describe los estados permitidos.

Una gramática formal se define como un conjunto de símbolos no terminales V, símbolos terminales Σ, reglas de producción P y un símbolo de inicio S. Estos estados permitidos serán el conjunto de símbolos. Durante el proceso de *decoding* el modelo lingüistico emite un conjunto de posibles *tokens* basados en distintas funciones de probabilidad calculadas

durante el entrenamiento del mismo.

$$G = (V, \Sigma, P, S) \tag{2.14}$$

Estas probabilidades para cada tokens (también llamadas *logits*) son interceptadas por un parser incremental que actua como motor de completado; dado una secuencia de carácteres parcial (es decir, que la generación del modelo no ha concluido) el parser devuelve el conjunto de tokens permitidos (estados permitidos) en función de la gramática definida, para continuar esa secuencia.

```
root  ::= (expr "=" ws term "\n")+
expr  ::= term ([-+*/] term)*
term  ::= ident | num | "(" ws expr ")" ws
ident ::= [a-z] [a-z0-9_]* ws
num   ::= [0-9]+ ws
ws    ::= [ \t\n]*
```

Figura 2.4. Ejemplo de gramática aritmética en GBNF.

Grammar-Constrained Decoding[12] (GCD) se realiza durante la fase de la decodificacion, restringiendo la distribución de probabilidad de los tokens generados por el modelo para que solo se incluyan aquelos que cumplen con la grámatica. La ventaja de esta realización es que no modifica el conocimiento aprendido por el modelo al no necesitar un entrenamiento o *fine-tunning* específico.

2.2.2. Grammar-Aligned Decoding

Se introduce el concepto de *Grammar-Aligned Decoding*[22] (GAD), que se enfoca en mejorar la capacidad de los modelos de lenguaje para generar salidas que sean no solo gramaticalmente correctas, sino también que se alineen con las distribuciones de probabilidad aprendidas por el modelo.

El problema surge porque técnicas como la anterior garantizan que las salidas sigan una gramática específica, pero distorsionan la distribución de probabilidad del modelo, lo que puede llevar a la generación de resultados con probabilidades no representativas del modelo original, sesgando las secuencias generadas.

Se define la probabilidad de una secuencia de tokens como un producto de distribuciones condicionales de izquierda a derecha:

$$P(w_1, \ldots, w_n) = \prod_{i=1}^{n} P(w_i | w_1, \ldots, w_{i-1}) \tag{2.15}$$

La clave es calcular una distribución ajustada $Q(w)$ que sea proporcional a la probabilidad

$P(w)$ del modelo, pero restringida a las secuencias aceptadas por la gramática G:

$$Q(w) = \frac{1[w \in L(G)] \cdot P(w)}{\sum_{w' \in L(G)} P(w')} \tag{2.16}$$

Una pieza crucial en el GAD es el cálculo de la gramaticalidad futura esperada (EFG), que mide la probabilidad de que una continuación de la secuencia generada hasta el momento sea gramatical. Esto se formaliza como:

$$QP, G(w_i|w_1 : i-1) = P(w_i|w_1 : i-1) \cdot EFG(w_1 : i) \tag{2.17}$$

Para estimar el valor de EFG se propone un nuevo algoritmo de decodificación llamado Adaptive Sampling with Approximate Expected Futures (ASAp), que permite generar salidas gramaticales mientras mantiene la fidelidad a la distribución de probabilidad original del modelo, ASAp funciona de la siguiente manera:

- Para los tokens generados, recuerda las probabilidades de los tokens enmascarados.
- A continuación revisa las probabilidades futuras a medida que se generan nuevas muestras, ajustando gradualmente las ponderaciones para que se acerquen a la distribución verdadera del modelo.
- El algoritmo va mejorando las estimaciones de las probabilidades futuras gramaticales mediante muestreo repetido y actualización de las probabilidades. A lo largo del tiempo, ASAp converge a una distribución que genera secuencias gramaticales sin sesgar tanto las probabilidades del modelo original.

2.2.3. Automata-based constraints

Se introduce *Automata-based constrained*[18]. Esta metodología soluciona la problemática de que la tokenización usada por los modelos puede ser ambigua y no alineada con las reglas del lenguaje formal.Se propone aplicar teoría de autómatas, permitiendo generar de forma eficiente salidas válidas sin distorsionar las probabilidades del modelo.

Este método utiliza autómatas finitos (*Finite-State Automata*)(FSA) y transductores finitos (*Finite-State transducer*(FST) para definir y aplicar restricciones al proceso de decodificación. Un autómata finito A se define como un conjunto de estados Q, un conjunto de símbolos de entrada Σ , un estado inicial I, un conjunto de estados finales F, y un conjunto de transiciones E:

$$A = (\Sigma, Q, I, F, E) \tag{2.18}$$

Las transiciones en E conectan los estados a través de los símbolos de entrada. Este autómata acepta una cadena $w \in \Sigma^*$ si hay una secuencia de transiciones que conduce desde el estado inicial hasta un estado final. Este autómata puede representar el conjunto de cadenas válidas en un lenguaje formal.

Un transductor finito T extiende los autómatas finitos al generar salidas. Un transductor T se

define de forma similar a un autómata finito, pero las transiciones incluyen tanto un símbolo de entrada como un símbolo de salida:

$$T = (\Sigma, \Delta, Q, I, F, E) \tag{2.19}$$

Donde σ es el conjunto de símbolos de entrada, Δ es el conjunto de símbolos de salida, y cada transición $e \in E$ es un tuple $(es, e\sigma, e\delta, et)$, donde $e\sigma$ es el símbolo de entrada y $e\delta$ es el símbolo de salida. El transductor convierte secuencias de entrada en secuencias de salida, lo que permite modelar el proceso de detokenización en el modelo de lenguaje.

La idea clave es la composición de un transductor finito T con un autómata finito A, lo que permite restringir las secuencias generadas por el modelo de lenguaje. La composición genera un nuevo autómata A' que acepta solo las cadenas que son válidas tanto según la gramática como según las restricciones del proceso de tokenización.

$$A' = A \circ T \tag{2.20}$$

Donde la composición asegura que la salida del transductor T sea aceptada por el autómata A.

Para finalizar, el proceso de detokenización se resuelve como un problema de transducción. Se construye un transductor T_V que convierte las secuencias de tokens en texto, asegurando que el proceso de tokenización del modelo se alinee con la gramática del lenguaje formal.

$$\begin{gathered}
\mathsf{T}_V = (\Sigma_V, \Delta_V, Q_V, I_V, F_V, E_V) \\
\Sigma_V \leftarrow V \\
\Delta_V \leftarrow \{v_i : v \in V, 1 \leq i \leq |v|\} \\
\mathsf{Q}_V \leftarrow \{q_r\}, I_V \leftarrow q_r, F_V \leftarrow \{q_r\} \\
\mathsf{E}_V \leftarrow \{(q_{i-1}, , v_i, q_i) parav \in V\}
\end{gathered}$$

Figura 2.5. Construcción del transductor de detokenización

El proceso de decodificación restringido se basa en aplicar las restricciones del autómata en cada paso del proceso de generación del modelo de lenguaje. Esto se realiza penalizando los logits del modelo para descartar tokens no válidos según la gramática:

$$P_{restric}(w_i|w_1, \ldots, w_{i-1}) = P(w_i|w_1, \ldots, w_{i-1}) \cdot 1[w_i \in L(A)] \tag{2.21}$$

Donde $1[w_i \in L(A)]$ es una función indicadora que vale 1 si el token w_i es válido según el autómata A.

2.3 Contexto y justificación

Como se ha desarrollado, el proceso de generación de texto es totalmente estocástico, ya que implica generar unas probabilidades: logits, para cada posible iteración del modelo. En cada una de estas iteraciones, se asigna logits a cada token del vocabulario, los métodos usados para decidir que tokens escoger se denominan *sampling methods*.

Actualmente el método más extendido es el *Grammar-based decoding*, teniendo implementaciones muy usadas en repositorios como: llama.cpp[13], HuggingFace Guidance[16], vLLM[19], etc...

La principal motivación de este proyecto es experimentar con los métodos actuales de sampling que consiguen asegurar una generación estructurada en torno a un conjunto de reglas formales. Esto es necesario para poder controlar y ajustar la inferencia de modelos a ciertos casos de uso en donde la estructura del lenguaje es crítica.

Como uso práctico, se utilizará una gramática formal que define las reglas aritméticas de las operaciones matemáticas. El objetivo será conseguir generaciones sintacticamente y semánticamente validas mediante la aplicación de estas técnicas de sampling y decoding. También se medirá el rendimiento con diferentes modelos actuales y presentes en la industria de manera muy relevante.

2.4 Planteamiento del problema

El principal problema a tratar es conseguir una generación determinista en función a una gramática formal que describa la sintaxis de una estructura específica.

Para ello se propondrá una experimentación y una métricas que permitan establecer una comparación coherente entre las distintas ténicas de decodificación. También se proponen usar modelos actualmente relevantes en el ámbito industrial para medir las capacidades de estas ténicas a distintas escalas.

Se propone una monitorización de los recursos usados como referencia para poder medir el aprovechamiento de los mismo, y optimizarlos en la medida de lo posible para obtener una mejor generación.

Capítulo 3. OBJETIVOS

En este capítulo se debe incluir la descripción detallada de los objetivos. Es recomendable reutilizar lo indicado en el anteproyecto.

3.1 Objetivos generales

Como objetivos generales, se plantean los siguiente

- Estudio profundo de los modelos lingüísticos; mecanismos, funcionamiento, ideas y conceptos.
- Estudiar el *State of The Art* (SoTA) en cuanto a modelos lingüísticos; métodos de decodificación, ténicas de selección de muestras.
- Profundizar en conceptos de verificación formal y sus aplicaciones directas en la industria.
- Analizar los conceptos matemáticos tras los métodos expuestos.
- Recorrer y desarrollar un método ciéntifico encaminado a la investigación más industrial y práctica.
- Profundizar en conceptos básicos de computación.

3.2 Objetivos específicos

En cuanto a los objetivos más específicos, se desglosarán y ordenarán en función de los anteriores:

- Estudio profundo de los modelos lingüísticos:
  - Arquitecturas Transformers.
  - Mecanismos de codificación.
  - Mecanismos de atención.
  - Mecanismos de decodificación.
  - Mecanismos de sampling.

- *SoTA* técnicas modelos lingüísticos:
  - Grammar-Constrained Decoding.
  - Grammar-Aligned Decoding.
  - Automata-based constrains.

- Definir la experimentación:
  - Artefactos a usar; *baseline* establecido.
  - Modelos a utilizar.
  - Métricas a usar.
  - Presentación de resultados.
- Recorrer la implementación de *GCD* en framework industriales.

3.3 Beneficios del proyecto

Como beneficio principal, se establece una experimentación que permite medir de manera coherente la eficacia y el rendimiento de las actuales ténicas de decodificación gramatical entre distintos modelos relevantes a nivel industrial. Permitiendo conocer las capacidades de estas técnicas en diferentes recursos software.

Se establecen medidas para la monitorización y control de los principales recursos hardware con el fin de aprovecharlos al máximo y entenderlos en profundidad. Se plantea el uso de un framework de inferencia actual con una relevancia industrial potente.

Como beneficio principal individual se obtiene conocimientos y profundidad en un campo tan novel como son los modelos lingüísticos. Este proyecto permite el desarrollo del autor en profundidad en campos de importante relevancia industrial. A la vez que permite afianzar y reforzar conocimientos transversales de igual importancia como conceptos computacionales, conocimiento profundo del funcionamiento del hardware y software.

Capítulo 4. DESARROLLO DEL PROYECTO

4.1 Planificación del proyecto

A modo de resumen acerca de la planificación, se comentará como se afronta el planteamiento inicial para el proyecto, comó se estudia las implementaciones actuales acerca de la codificación basada en una gramática y que conclusiones se obtienen.

Para poder inferir gramaticalmente es necesario primero establecer que frameworks actuales proveen servicio para ello. La selección de estos tiene un fuerte carácter industrial, quiere decir, que han sido seleccionados debido a su actual relevancia en el ecosistema de los modelos lingüísticos.

Se desecha la idea de construir un sistema de inferencia para modelos lingüísticos ya que esto plantearía una problemática completamente distinta. Además de que exiten soluciones ampliamente utilizadas y estandarizadas debido a su eficacia y desempeño real.

4.1.1. Frameworks de inferencia

En primer lugar, se establece cuales son los principales frameworks destinados a optimizar y proporcionar soporte a la inferencia y servicio de modelos lingüísticos. Se destacan:

- **vLLM**: Construido en torno al uso de *PagedAttention*[20], un nuevo algoritmo de atención que destaca por optimizar la gestión de memoria de la KV cache. Inspirado en los conceptos de paginación en sistemas operativos, PagedAttention divide la caché de KV en bloques que pueden almacenarse en ubicaciones de memoria no contiguas. Cada bloque contiene las claves y los valores de un número fijo de tokens. Durante el proceso de atención, en lugar de acceder a un espacio contiguo de memoria, el algoritmo identifica y recupera los bloques relevantes de manera separada.

- **llama.cpp**: Libería escrita en C++ (con la intención de controlar de manera estricta la gestion de la memoria y el multithreading) para el servicio e inferencia de distintos modelos lingüísticos. Destaca por su ligereza y capacidad de proporcionar soporte en una gran variedad de aceleradores, dando cobertura al *edge inference* (Inferencia en pequeños dispositivos localizados en la fuente de la obtención de datos). Basado en el trabajo previo de la librería GGML para algebra tensorial.

- **Text generation Inference (TGI)**: Framework especializado en generación de texto para sistemas de altas prestaciones, su principal aporte reside en el *continous batching* (O *iteration-level scheduling*[25] según el paper original) que permite encadenar lotes de peticiones al modelo lingüístico según van finalizando sus secuencias, en lugar de esperar a que todo el lote complete su generación. Esta técnica permite aprovechar el computo no utilizado en secuencias de longitud variable de un mismo lote.

- **HF Transformers**: No es tan solo un framework de servicio de modelos lingüísticos ya que engloba el groso de actividades, trabajos, algoritmos y optimizaciones realizadas a las arquitecturas transformers. Se ha visto necesario destacar esta librería ya que todas las anteriores beben de ella.

- **Triton Inference Server**: Framework de inferencia desarrollado por NVIDIA optimizado para el despliegue múltiple como servidores clouds, centros de datos, dispositivos *edge inference/computing* y sistemas onchip. Su especialidad reside en la capacidad de gestionar multiples aceleradores utilizando *tensor and pipeline parallelism*. Posee también un analizador de modelos que permite configurar la configuración óptima de hiperparámetros para cada modelo.

- **Torch Serve**: Servicio de inferencia desarrollado por Facebook y AWS, como cabe esperar, destaca por su integración nativa con modelos construidos con la librería Pytorch.

- **Tensorflow Serving**: Similar a Torch Server pero para modelos entrenados mediante la libería de Google de *machine learning*, Tensorflow. Optimizada para aceleradores TPU y para el lenguaje JAX.

- **DeepSpeed-MII**: Desarrollado por Microsoft, incluye optimizaciones como el bloque de la KV caché, *continuous batching*, *SplitFuse*[15] dinámico, paralelismo a nivel de tensor y CUDA kernels personalizados para ofrece un alto rendimiento.

- **Tensor RT LLM**: Desarrollado por NVIDIA, es un framework de inferencia que introduce numerosas optimizaciones, entre ellas, la capacidad de ejecutar modelos en TensorRT, un motor de inferencia de alto nivel introducido también por NVIDIA con optimizaciones tales como: cuantización, fusión de *layers*, *kenerl tiling*, etc...

Una vez investigados y estudiados los anteriores y más actuales frameworks de inferencia es necesario profundizar para escoger aquellos que implementen técnicas de decodificación gramaticales.

4.1.2. Implementaciones de GBD

A continuación se recogen aquellos frameworks que poseen implementaciones para algún mecanismo de decodificación gramatical. Actualmente el más implementado es *Grammar-Based Decoding* para generación de texto basado una gramática, existen simplificaciones de su mecanismo para adaptarlo a generaciones JSON, enumerados de elecciones y mediante patrones *Regex*.

- **vLLM**: Se presenta como el más prometedor ya que incluye técnicas de decodificación basadas en la gramática. Eso se realiza mediante dos proyectos externos. *Outlines* y *lm-format-enforcer*. Este framework ofrece implementaciones para ambas soluciones aunque se ha concluido que Outlines supera con creces debido a la madurez del proyecto.

- **llama.cpp**: Implementa de manera nativa la solución expuesta en el paper original de *Grammar-based decoding* por lo que se destaca su rapidez y rendimiento, un detrimento para este framework es que esta enteramente construido en C++ lo que dificulta su investigaciön a fondo. Para tratar de paliar esta problemática se propone utilizar un *binding* escrito en Python, llama-cpp-python.

- **Text generation Inference (TGI)**: Desarrollo en consonancia con HF Transformers, soporta decodificación gramática gracias a *Outlines*. También soporta JSON y Regex.

- **HF Transformers**: Desarrollo en consonancia con Text Generation Inference.
- **Triton Inference Server**: No implementado.
- **Torch Serve**: No implementado.
- **Tensorflow Serving**: No implementado.
- **DeepSpeed-MII**: No implementado.
- **Tensor RT LLM**: No implementado.

4.2 Descripción de la solución, metodologías y herramientas empleadas

En este apartado se definirá la experimentación a realizar. El objetivo a estudiar es, dada una definición formal gramatical y un modelo lingüístico, poder evaluar la correctitud sintáctica de la generación obtenida. Se estudiará el porcentaje de generaciones sintácticas para cada modelo y para diferentes técnicas de prompting.

4.2.1. vLLM

Se ha escogido vLLM como framework a desarrollar la experimentación debido a su marcado carácter industrial, a su implementación nativa en Python y a las técnicas de optimización de inferencia con las que cuenta. vLLM cuenta con el desarrollo open source de empresas punteras como NVIDIA, AWS, IBM, Cloudflare, BentoML, cuenta con técnicas de optimización como *Paged-Attention*, *Continuous batching* de peticiones, ejecución eficiente de modelos con grafos CUDA/HIP, CUDA kernels optimizados para *FlashAttention*[6] y *FlashInfer*[10], *speculative decoding* [21] y *Chunked prefill*[1].

4.2.2. Metología

Parser

En cuanto a la metodología a aplicar para la experimentación. Se comienza con la gramática escrita en Lark, que define las reglas sintácticas para las expresiones aritméticas. La gramática es declarativa y sigue el estilo de la Backus-Naur Form (BNF), donde se especifican las reglas para las operaciones matemáticas, la precedencia de los operadores, y las operaciones que serán válidas.

Para entender mejor Lark, es necesario explicar las dos partes fundamentales de un parser. El lexer (o analizador léxico) es el primer componente. Su función es tomar la entrada de texto y dividirla en una secuencia de tokens. Un token es una unidad básica de significado (como un número, un operador o un paréntesis). En este caso, el lexer identificará tokens como `NUMBER` (números), operadores aritméticos (+, -, *, /), y paréntesis.

Lark proporciona un lexer incorporado que funciona de forma automática. En la gramática, los tokens se definen implícitamente cuando se utilizan símbolos como `NUMBER`, operadores y paréntesis. La línea `import common.NUMBER` le indica a Lark que use una definición predefinida para el token `NUMBER`. Además, la regla `ignore ' '` le dice al lexer que ignore los espacios en blanco.

EL siguiente componente es el parser per se, o analizador sintáctico. Este toma la secuencia de tokens generada por el lexer y la organiza según las reglas de la gramática. El parser construye un árbol de análisis sintáctico (AST, por sus siglas en inglés), que representa la estructura de la expresión en función de cómo se relacionan los tokens.

El parser se construye automáticamente a partir de la gramática que se proporciona. Lark utiliza estas reglas para crear un parser LALR(1) *(look-ahead, left-to-right, rightmost derivation parser)* o un parser Earley, dependiendo del tipo de gramática. El parser interpreta las reglas de la gramática, como la regla de comparación, la de expresiones o la de términos.

El parser es una parte fundamental en la experimentación ya que permitirá evaluar la correctitud sintáctica de las generaciones obtenidas por los modelos.

Experimentación

Para cada modelo, se realizará una comparativa entre tres escenarios: generación libre y estocástica basada unicamente en las capacidades del modelo *("nogbd")*, generación restringida por la gramática aritmética (*"gbd"*) y generación restriginda por la gramática en conjunción con una técnica de prompting denominada *Few-shots*[4] (*"gbd+fewshots"*). Esta consiste en incluir en el prompt una serie de ejemplos similares a los resultados que se desea obtener para guiar la generación. La idea general será evaluar ambas generaciones mediante el parser y obtener el porcentaje de correctitud sintáticas para cada experimentación. También se propone estudiar la correctitud semántica utilizando simulación de código en Python, en el que se ejecutará las generaciones obtenidas en forma de código Python.

Se propone adiccionalmente un estudio del hiperparámetro del que dependen el número máximo de tokens a generar para evaluar si aumentando la cantidad de tokens generados el rendimiento de los modelos varia.

A continuación se detallan las partes más relevantes de la experientación:

Clase Generation

```python
class Generation:
    def __init__(self, seed, elapsed_time, gen):
        """
        arrival_time: The time when the request arrived.
         first_scheduled_time: The time when the request was first
          ↪  scheduled.
         first_token_time: The time when the first token was generated.
         time_in_queue: The time the request spent in the queue.
         finished_time: The time when the request was finished.
        """
        # Elapsed time its based on vLLM values finished_time - arrival_time
        self.seed = seed
        self.elapsed_time = elapsed_time
        self.gen = gen
```

Esta clase contiene la información necesaria a estudiar para cada generación dado un modelo, se almacena la semilla (*seed*) que permite obtener generaciones similares, el tiempo necesario por el modelo para generar la muestra y la muestra per se.

Clase Result

```python
class Result:
    def __init__(self, seeds: list[int], elapsed_time_gen: list[int],
    ↪  syntax_validation: list[int], semantic_validation: list[int]):

        self.model = MODEL.replace("/","::")
        self.experiment_type = EXPERIMENT_TYPE
        self.samples = SAMPLES
        self.max_tokens= MAX_TOKENS
        self.seeds = [seed for seed in seeds[:SAMPLES]]
        self.elapsed_time_gen = [elapsed for elapsed in
        ↪  elapsed_time_gen[:SAMPLES]]
        self.syntax_validation = syntax_validation
        self.semantic_validation = semantic_validation
    [...]
```

Esta clase continene la información necesaria para los resultados de cada experimentos. Tiene cierto nivel de redundancia con la clase *Generation* a modo de facilitar el acceso de datos. En ella se encuentra información acerca del tipo de experimento (*experiment_type*), el número de muestras (*samples*), el número máximo de tokens (*max_tokens*), el tiempo por generación (*elapsed_time_gen*), y los resultados para los análisis sintáticos (*syntax_validation*) y semántico (*semantic_validation*).

Preprocesamiento

```python
def gen_preproc(generation:str):
    eot_id_comparison = None
    match EXPERIMENT_TYPE:
        case "nogbd":
            # Treatment for returning generation until <|eot_id|>
            eot_id_gen = generation.split("<|eot_id|>")[0]
            # Treatment for replace =(comparision) with == (used in no gbd
            ↪  generation), As using simpify (Python code simulation)
            eot_id_comparison = eot_id_gen.replace("=", "==")

        case "gbd":
            # Treatment for returning generation until <|eot_id|>
            eot_id_comparison = generation.split("<|eot_id|>")[0]

        case "gbd+fewshots":
            # Treatment for returning generation until <|eot_id|>
            eot_id_comparison = generation.split("<|eot_id|>")[0]

    return eot_id_comparison
```

Esta función permite aplicar un preprocesamiento necesario en función del experimento a

realizar, la idea intuitiva es que para todos los experimentos se secciona la generación hasta que el modelo emite el token de final de generación <|eot_id|>. Así se obtiene una única generación por *sample*. En el caso concreto de la experimentación "nogbd", como el modelo tiene total libertad para generar cualquier token, se asegura que la igualdad (=) sea la correcta para el idioma Python (==). Para el resto de experimentaciones como la generación de los tokens está sujeta a aquellos permitidos por la gramática (a saber: números, operadores matemáticos y paréntesis) no hay necesidad de aplicar este preprocesamiento.

Validaciones sintácticas y semánticas

```
from lark import Lark
import sympy as sp
[...]
parser = Lark(grammar, parser='lalr')
[...]
def semantic_test(generation: str):
    sp.sympify(generation)

def syntax_text(generation: str, parser):
    parser.parse(generation).pretty()
```

Estas dos funciones permiten evaluar la generación resultante del modelo. Según lo propuesto en la metodología, esto se conseguirá, en cuanto a la validación sintáctica, *parseando* la generación obtenida. Y en cuanto a la validación semántica, simulando la operación aritmética en código Python y comprobando su correctitud.

La implementación escogida para resolver estas intenciones es la de utilizar la herramienta de Python Lark, Lark es un conjunto de herramientas de *parseo* para gramáticas libres de contexto con especial objetivo en el rendimiento, la modularidad y la facilidad de manejo. Lark permitirá crear un *parser* a partir de la gramática escogida para posteriormente validar las generaciones en función de dicho parser.

Para la validación semántica se ha escogido la herramienta en Python de Simpy. Simpy es un framework de simulación basado en procesos escrito integramente en Python. De manera resumida, simpy permite utilizar el motor aritmético del lenguaje Python e interpretar expresiones válidas como código nativo.

Prompts

```
[...]
 match EXPERIMENT_TYPE:
            case "gbd":
                arithmetic_prompt=f"""
        <|begin_of_text|><|start_header_id|>system<|end_header_id|>

```

```
        You are a helpful AI assistant for creating gramatically and
        ↪  sintactically arithmetic expression<|eot_id|>
        <|start_header_id|>user<|end_header_id|>
        Rewrite 9 * 15 as others equivalents expressions:
        Follow this example:
        (5*5)=(5+5+5+5+5)=(25*1)=(5*3)+(5*2).
        <|eot_id|><|start_header_id|>assistant<|end_header_id|>
        """

            case "nogbd":
                arithmetic_prompt=f"""
        <|begin_of_text|><|start_header_id|>system<|end_header_id|>

        You are a helpful AI assistant for creating gramatically and
        ↪  sintactically arithmetic expression<|eot_id|>
        <|start_header_id|>user<|end_header_id|>
        Rewrite 9 * 15 as other equivalent expression, for the response, do
        ↪  not use text.
        Just only characters available in this grammar:
        ↪  {arithmetic_grammar}<|eot_id|>
        <|start_header_id|>assistant<|end_header_id|>
        """

            case "gbd+fewshots":
                arithmetic_prompt="""
        <|begin_of_text|><|start_header_id|>system<|end_header_id|>

        You are a helpful AI assistant for creating gramatically, equivalent
        ↪  and correct arithmetical expression<|eot_id|>
        <|start_header_id|>user<|end_header_id|>
        Given the following examples:\n
        (5*5)=(5+5+5+5+5)=(25*1)=(5*3)+(5*2).\n
        (3*3)=(3+3+3)=(3+6)=(9*1).\n
        (3*4*5)=3*(2+2)*5=15*4=15*(2+2)=(12*5)=(20*3).\n
        Rewrite 9 * 15 as others equivalents expressions:
        <|eot_id|><|start_header_id|>assistant<|end_header_id|>
        """
[...]
```

Para poder inferir es necesario un prompt del que partir. Se muestran arriba los prompts selecionados para cada tipo de experimentación. Como se ha comentado en la metodología:

Para todos los prompts se ha utlizado como referencia la tarjeta de prompting para el modelo a usar. De manera general, estos muestran una serie de mensajes entre tokens especiales (`<|begin_of_text|><|start_header_id|>`) que guían al modelo en la generación de texto.

Al ser versiones *Instruct* (Esto quiere decir, que han sido refinadas mediante *fine-tuning* para recibir instrucciones y responder como si se tratara de una conversación) esto es mandatorio para no degradar las capacidades del modelo.

El mensaje inicial del sistema (<|begin_of_text|><|start_header_id|>system<|end_header_id|>) indica al modelo como comportarse de manera general. A continuación se incluye el prompt per ser (la instrucción a responder) (<|start_header_id|>user<|end_header_id|> y se indica la cabecera correspondiente que indica al modelo donde comenzar su generación. (<|eot_id|><|start_header_id|>assistant<|end_header_id|>)

Esto es así para los tres tipos de experimentación. Los matices residen dentro del prompt propiamente dicho, en donde para la versión *"nogbd"* se le incluye instrucciones al modelo para que no use texto. En la versión *"gbd"* incluye un ejemplo de respuesta para dar una estructura lógica a la generación restrigida por la gramática y para el *"gbd+fewshots"* se incluye la técnica de *prompting few shots* que cómo se ha indicado previamente permite al modelo utilizar *in context learning*[4] para sintetizar y aprender la respuesta esperada.

4.2.3. Gramáticas

Arimética

Se ha escogido la gramática aritmética como objetivo de la experimentación. El objetivo será obtener generaciones que verifiquen las reglas establecidas en ella.

```
?start: comparison

    ?comparison: expression ("==" expression)* "<|eot_id|>"?

    ?expression: term (("+" | "-") term)*

    ?term: factor (("*" | "/") factor)*

    ?factor: NUMBER
        | "-" factor
        | "(" comparison ")"

    %import common.NUMBER
    %ignore " "  // Ignore spaces

    // Define <|eot_id|> as a terminal
    EOT_ID: "<|eot_id|>"
    """
```

La gramática define la regla `?start: comparison`, lo que indica que la regla inicial del análisis es una comparación. Cabe destacar que estos nombres son contextuales y elegidos para mayor legibilidad de la gramática a formalizar. El símbolo `?` antes del nombre de la regla indica que esta regla puede producir un nodo vacío en el árbol de análisis (es decir, que no tiene que estar necesariamente presente en todas las evaluaciones, dependiendo del contexto). La

regla de comparación servirá como base del lenguaje aritmético, permitiendo comparaciones entre dos expresiones mediante el operador de igualdad.

La regla `comparison` define lo que constituye una comparación. En este caso, una comparación se compone de una expresión (`expression`) seguida opcionalmente por un signo de igualdad (==) y otra expresión. El símbolo * permite encadenar multiples comparaciones. El signo de igualdad es (==) ya que es el signo de comparación nativo de Python, eso es necesario para la comprobación semántica en la que se ejecutará la operación en código Python. Si no hay un operador de igualdad, simplemente se interpreta la primera expresión.

Al final de esta regla se incluye el token especial `<|eot_id|>` que simboliza el fin de una generación por parte del modelo.

La regla `expression` define cómo se estructuran las expresiones aritméticas. Una expresión se compone de un término inicial, seguido opcionalmente de uno o más términos adicionales, precedidos por operadores de suma o resta (”+” | ”-”). El uso de (”*” | ”/”) indica que el parser permite tanto la suma como la resta y repite este patrón cuantas veces sea necesario.

La regla `term` define cómo se estructuran los términos dentro de una expresión. Un término es un `factor` seguido opcionalmente de otros factores precedidos por los operadores de multiplicación o división ("*"| /"). Este patrón permite manejar múltiples operaciones de multiplicación o división entre factores. Este enfoque también respeta la precedencia de los operadores.

La regla `factor` describe cómo se define un factor, que puede ser un número `(NUMBER)`, un número negativo `(’’-’’ factor)`, o una expresión entre paréntesis `(’’(’’ comparison ’’)’’)`. Esta última opción permite modificar la precedencia natural de las operaciones mediante el uso de paréntesis. Los factores son las unidades más simples en la gramática aritmética, ya que representan los elementos que no se pueden descomponer más en términos de esta gramática.

La instrucción `import common.NUMBER` indica que la gramática está importando una definición predefinida del token NUMBER, lo que permite el reconocimiento de números en las expresiones sin necesidad de definir cómo se representan estos números dentro de esta gramática. Esto es una forma conveniente de reutilizar reglas comunes que ya están definidas en Lark.

La instrucción `ignore ’’ ’’` indica al parser que ignore los espacios en blanco en las expresiones, lo que permite que las expresiones contengan espacios sin afectar el análisis. Esto es importante porque los espacios no tienen significado semántico en las operaciones aritméticas, por lo que deben ser ignorados para que el parser se centre solo en los elementos significativos (números, operadores, paréntesis).

4.2.4. Modelos

En cuanto a los modelos a estudiar, se han seleccionado la familia de modelos de Meta Llama[9]: Llama 3 70B Instruct y Llama 3 8B Instruct. Esto es asi debido a que son mode-

los frontera de acceso libre; con capacidades similares a aquellos ofrecidos por servicios como OpenAI, Anthropic y Google. Se ha selecionado una misma familia, con una misma arquitectura, para evaluar como mejora la inferencia en función del aumento del número de parámetros (de 8B a 70B).

La versión *Instruct* indica que el modelo ha recibido *fine-tunning* como agente conversacional, reforzando la acción de responder a preguntas y proporcionar información según lo exigido en el *prompt*. Esta versión se diferencia de la original en que la versión original es el modelo con el resultado de la fase de *training* en donde el modelo unicamente realiza predicciones del texto proporcionado. La versión *Instruct* realiza una comprensión más profunda sobre el texto proporcionado para predecir la respuestas más acorde.

Debido a limitaciones en cuanto a la capacidad de memoria VRAM disponible y tamaño de los modelos. Para Llama3 70B se ha elegido un modelo quantizado en INT8. A continuación se presenta la problemática de la cuantización:

Cuantización de modelos

Debido a las limitaciones de memoria disponibles surge la problemática de alojar grandes modelos de manera óptima sin afectar notoriamente a sus capacidades de conocimiento, para esta problemática se plantea la ténica de cuantización[17].

La técnica de cuantización es una ténica de compresión que convierte los pesos y las activaciones dentro del modelo en un valor de menor precisión que el original. Esto significa cambiar de un tipo de dato con capacidad para almacenar mayor información a otro con una capacidad menor. Esto produce que al almacenar valores menos precisos ocupen menos espacio en memoria. Un ejemplo tipico es convertir datos tipo float de 32-bits a un integer de 8-bits.

Se comprueba[7] que cuantizar hasta valores integer de 4-bits mantienen de manera persistente las capacidades del modelo. Para este proyecto y con el fin de perturbar en lo más mínimo las capacidades originales del modelo, se ha elegido una cuantización en intenger de 8-bits mediante la ténica de cuantización *GPTQ*[11]. Manteniendo el aprovechamiento de la memoria y el computo en lo más óptimo según los recursos proporcionados.

4.3 Recursos requeridos

Para la ejecución de este proyecto se ha contado con los siguientes recursos ordenados por categorías:

4.3.1. Hardware

- CPU: x48 Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz
- GPU: x8 Tesla V100-SXM2-16GB
- RAM: 176GB
- Disco: 1TB

4.3.2. Software

- CUDA Version: 12.1 (Conda)
- Python Version: 3.10
- Driver Version: 550.54.15
- Compute capabilities: 7.0
- torch==2.4.0
- vllm==0.5.5
- transformers==4.43.3

4.4 Presupuesto

Los servicios proporcionados para este proyecto han sido generosamente donados por Datacrunch.io, por lo que su coste es 0€. A continuación se presentará una estimación de estos costes en el supuesto caso de que hubieran costado dinero:

Tipo de coste	Valor	Comentarios
Horas de trabajo en el proyecto	560	Horas realizadas en global para todo el proyecto. (3 meses, 40/h semana)
Costes hardware	7,938.93€	• Instance total: €6,007.208: (€2.721/h) • Storage total: €553.894 (€0.251/h) • Total cost: €6,561.10 • Total cost (incl. VAT): €7,938.93
Costes software	0€	Todo el software utilizado es open source y no tiene ningun coste asociado. El software propietario utilizado es aquel pertinente al hardware ya pagado
Estudios e informes	0€	La investigación y el SoTA ha sido desarrollado por el integrante del proyecto

Tabla 4.1. Costes del proyecto

4.5 Resultados del proyecto

A continuación se presentan los resultados obtenidos en la experimentación de los modelos. Para cada uno de ellos, se adjuntará una tabla comparativa entre las distintas técnicas de decoding para un número máximo de tokens, obteniendo un porcentaje de correctitud sintática y semántica para cada 100 experimentaciones con cada técnica.

4.5.1. Llama 3 8B Instruct

Para el modelo Llama 3 8B, con precisión float 16 *(dytpe=half)* y sin cuantización, los resultados obtenidos son:

Tokens máximos: 50

Experimento	Correctitud sintática (%)	Correctitud semántica (%)
NoGBD	4 %	48 %
GBD	86 %	86 %
GBD+FewShots	55 %	55 %

Tabla 4.2. Llama 3 8B, max_tokens = 50

Experimento	Tiempo total inferencia (segundos)	Tiempo medio (segundos)
NoGBD	9.50s	0.095s
GBD	750.70s	7.50s
GBD+FewShots	933.94s	9.33s

Tabla 4.3. Tiempos Llama 3 8B, max_tokens = 50

Tokens máximos: 100

Experimento	Correctitud sintática (%)	Correctitud semántica (%)
NoGBD	4 %	48 %
GBD	96 %	96 %
GBD+FewShots	82 %	82 %

Tabla 4.4. Llama 3 8B, max_tokens = 100

Experimento	Tiempo total inferencia (segundos)	Tiempo medio (segundos)
NoGBD	9.36s	0.094s
GBD	872.54s	8.72s
GBD+FewShots	1434.81s	14.34s

Tabla 4.5. Tiempos Llama 3 8B, max_tokens = 100

Tokens máximos: 200

Experimento	Correctitud sintática (%)	Correctitud semántica (%)
NoGBD	4 %	48 %
GBD	98 %	98 %
GBD+FewShots	95 %	95 %

Tabla 4.6. Llama 3 8B, max_tokens = 200

Experimento	Tiempo total inferencia (segundos)	Tiempo medio (segundos)
NoGBD	9.10s	0.091s
GBD	1042.36s	10.42s
GBD+FewShots	1738.10s	17.38s

Tabla 4.7. Tiempos Llama 3 8B, max_tokens = 200

4.5.2. Llama 3 70B Instruct

Para el modelo Llama 3 70B, con precisión INT8 y cuantización GTPQ, los resultados obtenidos son:

Tokens máximos: 50

Experimento	Correctitud sintática (%)	Correctitud semántica (%)
NoGBD	73 %	80 %
GBD	84 %	84 %
GBD+FewShots	37 %	37 %

Tabla 4.8. Llama 3 70B, max_tokens = 50

Experimento	Tiempo total inferencia (segundos)	Tiempo medio (segundos)
NoGBD	326.90s	3.26s
GBD	1341.15s	13.41s
GBD+FewShots	1297.17s	12.97s

Tabla 4.9. Tiempos Llama 3 70B, max_tokens = 50

Tokens máximos: 100

Experimento	Correctitud sintática (%)	Correctitud semántica (%)
NoGBD	73 %	80 %
GBD	99 %	99 %
GBD+FewShots	86 %	86 %

Tabla 4.10. Llama 3 70B, max_tokens = 100

Experimento	Tiempo total inferencia (segundos)	Tiempo medio (segundos)
NoGBD	772.53s	7.72s
GBD	2761.27s	27.61s
GBD+FewShots	2056.50s	20.56s

Tabla 4.11. Tiempos Llama 3 70B, max_tokens = 100

Tokens máximos: 200

Experimento	Correctitud sintática (%)	Correctitud semántica (%)
NoGBD	73 %	80 %
GBD	100 %	100 %
GBD+FewShots	100 %	100 %

Tabla 4.12. Llama 3 70B, max_tokens = 200

Experimento	Tiempo total inferencia (segundos)	Tiempo medio (segundos)
NoGBD	1158.16s	11.58s
GBD	2540.49s	25.40s
GBD+FewShots	2217.98s	22.18s

Tabla 4.13. Tiempos Llama 3 70B, max_tokens = 200

Capítulo 5. DISCUSIÓN

5.1 llama 3 8B

Para las distintas variaciones del hiperparámetro max_token se puede observar como la técnica de GBD mejora sustacialmente la generación del modelo a nivel sintáctico y semántico (86 %, 96 %, 98 %), postulandose como la mejor técnica incluso con la técnica de prompting FewShots (55 %, 82 %, 95 %). Podemos ver también como, a pesar de tener unos porcentajes medios en cuanto a validación semántica, la utilización del modelo de manera nativa sin técnicas de decodificación no consigue adaptarse a unas reglas gramaticales formales (correctitud sintáctica).

De manera general, se puede observar que aumentando el contexto disponible en tiempo de inferencia mejora de manera significativa la capacidad del modelo de adaptarse a unas reglas formales y de generar contenido con significado y sentido.

Es notable comentar el aumento del tiempo medio y total para las técnicas de decodificación. Siendo especialmente más rápidas sin ellas. Esto es debido al computo post-inferencia añadido de crear una máquina de estados finitos para recorrer los carácteres de la generación e imponer una gramática con GBD. Este es proporcional al aumentar el contexto disponible en tiempo de inferencia (hiperparámetro max_token).

En cuanto a los tiempos, podemos ver una estabilidad en la media de los mismos cuando se utiliza tan solo la técnica de GBD. (Imagen 8.2, Imagen 8.5 y Imagen 8.8) Siendo mucho más inestables (o con unos valores mas variados) la técnica de GBD+FewShots (Imagen 8.3, Imagen 8.6 y Imagen 8.9) y NoGBD (Imagen 8.1, Imagen 8.4 y Imagen 8.7).

Para el aprovechamiento de recursos se observa una impecable optimización de los mismos por parte de vLLM ya que es capaz de gestionar crecientes cargas de trabajo con un aprovechamiento similar de los recursos. Esto se deduce debido a que las gráficas de recursos para la GPU principal en las 9 experimentaciones para el mismo modelo es idéntica (Imagen 8.19, Imagen 8.20 y Imagen 8.21). También se deduce que aumentar el contexto en tiempo de inferencia como se ha realizado en este proyecto no influye de manera significativa en la carga de trabajo.

5.2 llama 3 70B

Para las distintas variaciones del hiperparámetro max_token se puede observar como la técnica de GBD mejora sustacialmente la generación del modelo a nivel sintáctico y semántico (84 %, 99 %, 100 %), postulandose como la mejor técnica incluso con la técnica de prompting FewShots (37 %, 99 %, 100 %). Podemos apreciar que para la ténica NoGBD los resultados también son bastante aceptables (73 %, 73 %, 73 %) pero estácticos, ya que no escalan aumentando el contexto en tiempo de inferencia.

Para GBD y GBD+FewShots se puede observar que aumentando el contexto disponible en tiempo de inferencia mejora de manera significativa la capacidad del modelo de adaptarse a unas reglas formales y de generar contenido con significado y sentido.

Es notable comentar el aumento del tiempo medio y total para las técnicas de decodificación. Siendo especialmente más rápidas sin ellas. Esto es debido al computo post-inferencia añadido de crear una máquina de estados finitos para recorrer los carácteres de la generación e imponer una gramática con GBD. Este es proporcional al aumentar el contexto disponible en tiempo de inferencia (hiperparámetro max_token).

En cuanto a los tiempos, podemos ver una estabilidad en la media de los mismos cuando se utiliza tan solo la técnica de GBD. (Imagen 8.11, Imagen 8.14 y Imagen 8.17) Siendo mucho más inestables (o con unos valores mas variados) la técnica de GBD+FewShots (Imagen 8.12, Imagen 8.15 y Imagen 8.18) y NoGBD (Imagen 8.10, Imagen 8.13 y Imagen 8.16).

Para el aprovechamiento de recursos se observa una impecable optimización de los mismos por parte de vLLM ya que es capaz de gestionar crecientes cargas de trabajo con un aprovechamiento similar de los recursos. Esto se deduce debido a que las gráficas de recursos para la GPU principal en las 9 experimentaciones para el mismo modelo es idéntica (Imagen 8.22, Imagen 8.23 y Imagen 8.24). También se deduce que aumentar el contexto en tiempo de inferencia como se ha realizado en este proyecto no influye de manera significativa en la carga de trabajo.

5.2.1. Comparativa de modelos

Como se puede intuir, Llama 3 70B obtiene mejores resultados en la experimentación debido a su mayor número de parámetros y por ende, su mayor capacidad de razonamiento y comprension de información más compleja. Se puede apreciar un incremento del tiempo de inferencia proporcional al aumento del contexto en tiempo de inferencia para ambos modelos, siendo estos mayores en Llama 3 70B. Por lo que se concluye que Llama 3 8B es más rápido para cualquier tipo de experimentación propuesta en este proyecto.

Se evidencia que en ambos modelos, el aumento del contexto en tiempo de inferencia mejora sus capacidades para adaptarse a nivel sintáctico y semántico a unas reglas formales. Para ambos, los recursos de uso de GPU y memoria se mantienen estables dependiendo del modelo. Esto puede ser debido a que ambos modelos poseen la misma arquitectura interna, con la única variación del aumento en el número de parámetros.

Capítulo 6. CONCLUSIONES

6.1 Conclusiones del trabajo

Se concluye que las técnicas actuales más implementadas a nivel industrial cumplen con creces la problemática de dotar de una estructura y un determinismo a la generación de los modelos lingüísticos. Se deduce que estas técnicas escalan conforme al número de parámetros del modelo y que también son directamente propocionales al aumento del contexto en tiempo de inferencia.

Se concluye la eficacia y la optimización del framework de inferencia vLLM al aprovechar los recursos al máximo para realizar la experimentación propuesta.

6.2 Conclusiones personales

Se destaca la capacidad personal de plantear, planificar, investigar, estudiar, implementar y realizar en todos los aspectos una investigación con un marcado carácter e interés industrial de manera independiente y autodidacta. Se valora la oportunidad de poder revisar, mejorar y corroborar conocimientos anteriormente adquiridos y añadidos sobre temas de grán interés en la actualidad tecnológica.

Capítulo 7. FUTURAS LÍNEAS DE TRABAJO

Se enumeran a continuación las posibles mejoras o nuevas líneas de investigación basadas en el trabajo propuesto en este proyecto:

- Validación con nuevas gramáticas.
- Comparativa con modelos especializados en generación de código.
- Técnicas de optimización para GBD.
- Implementación de ténicas de decodificación gramatical nóveles no industrializadas.
- Ampliación con ténicas de sampling basadas en la entropía de los *logits* generados.
- Validación semántica de gramáticas más complejas con *Reinforcement Learning* y modelos lingüísticos.
- Implementación de un mayor número de técnicas de prompting y prompt tunning.
- Experimentación con distintos hiperparámetros en correlación con el rendimiento de los modelos.

Bibliografía

[1] Amey Agrawal et al. *SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills*. 2023. arXiv: 2308.16369 [cs.LG]. URL: https://arxiv.org/abs/2308.16369.

[2] Anthropic. *Claude 3 family*. 2024. URL: https://www.anthropic.com/news/claude-3-family.

[3] Dzmitry Bahdanau, Kyunghyun Cho y Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL]. URL: https://arxiv.org/abs/1409.0473.

[4] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[5] Rodrigo A. García Casasola. *Repositorio Github de la experimentación*. 2024. URL: https://github.com/roG0d/gbd_experimentation/tree/main/vllm/tfm.

[6] Tri Dao et al. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. 2022. arXiv: 2205.14135 [cs.LG]. URL: https://arxiv.org/abs/2205.14135.

[7] Tim Dettmers y Luke Zettlemoyer. *The case for 4-bit precision: k-bit Inference Scaling Laws*. 2023. arXiv: 2212.09720 [cs.LG]. URL: https://arxiv.org/abs/2212.09720.

[8] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805.

[9] Abhimanyu Dubey et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[10] FLashInfer. *FlashInfer Github repo*. 2024. URL: https://github.com/flashinfer-ai/flashinfer.

[11] Elias Frantar et al. *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*. 2023. arXiv: 2210.17323 [cs.LG]. URL: https://arxiv.org/abs/2210.17323.

[12] Saibo Geng et al. *Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning*. 2024. arXiv: 2305.13971 [cs.CL]. URL: https://arxiv.org/abs/2305.13971.

[13] Georgi Gerganov. *Llama.cpp github repository*. 2024. URL: https://github.com/ggerganov/llama.cpp.

[14] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[15] Connor Holmes et al. *DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference*. 2024. arXiv: 2401.08671 [cs.PF]. URL: https://arxiv.org/abs/2401.08671.

[16] HuggingFace. *HuggingFace Guidance*. 2024. URL: https://huggingface.co/docs/text-generation-inference/conceptual/guidance.

[17] Benoit Jacob et al. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. 2017. arXiv: 1712.05877 [cs.LG]. URL: https://arxiv.org/abs/1712.05877.

[18] Terry Koo, Frederick Liu y Luheng He. *Automata-based constraints for language model decoding*. 2024. arXiv: 2407.08103 [cs.CL]. URL: https://arxiv.org/abs/2407.08103.

[19] Woosuk Kwon et al. «Efficient Memory Management for Large Language Model Serving with PagedAttention». En: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. 2023. URL: https://github.com/vllm-project/vllm.

[20] Woosuk Kwon et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. 2023. arXiv: 2309.06180 [cs.LG]. URL: https://arxiv.org/abs/2309.06180.

[21] Yaniv Leviathan, Matan Kalman y Yossi Matias. *Fast Inference from Transformers via Speculative Decoding*. 2023. arXiv: 2211.17192 [cs.LG]. URL: https://arxiv.org/abs/2211.17192.

[22] Kanghee Park et al. *Grammar-Aligned Decoding*. 2024. arXiv: 2405.21047 [cs.AI]. URL: https://arxiv.org/abs/2405.21047.

[23] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2024. arXiv: 2312.11805 [cs.CL]. URL: https://arxiv.org/abs/2312.11805.

[24] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[25] Gyeong-In Yu et al. «Orca: A Distributed Serving System for Transformer-Based Generative Models». En: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, jul. de 2022, págs. 521-538. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/yu.

Capítulo 8. ANEXOS

8.1 Código de la experimentación

```
from vllm import LLM, SamplingParams
from dotenv import load_dotenv
from lark import Lark, exceptions
from lark.indenter import Indenter
from random import randint
import time
import sympy as sp
import json
import os

SAMPLES = 100
MODEL = "study-hjt/Meta-Llama-3-70B-Instruct-GPTQ-Int8"


# Auxiliar functions
class Generation:
    def __init__(self, seed, elapsed_time, gen):
        """
        arrival_time: The time when the request arrived.
         first_scheduled_time: The time when the request was first
          ↪  scheduled.
         first_token_time: The time when the first token was generated.
         time_in_queue: The time the request spent in the queue.
         finished_time: The time when the request was finished.
        """
        # Elapsed time its based on vLLM values finished_time - arrival_time
        self.seed = seed
        self.elapsed_time = elapsed_time
        self.gen = gen

class Result:
    def __init__(self, seeds: list[int], elapsed_time_gen: list[int],
    ↪  syntax_validation: list[int], semantic_validation: list[int]):
        self.model = MODEL.replace("/","::")
        self.experiment_type = EXPERIMENT_TYPE
        self.samples = SAMPLES
        self.max_tokens= MAX_TOKENS
        self.seeds = [seed for seed in seeds[:SAMPLES]]
        self.elapsed_time_gen = [elapsed for elapsed in
        ↪  elapsed_time_gen[:SAMPLES]]
        self.syntax_validation = syntax_validation
        self.semantic_validation = semantic_validation

    def save(self):
```

```
42          with open(f"./results/{self.model}/{self.experiment_type}/"
43          "{self.samples}e_{self.max_tokens}t.jsonl", 'w') as file:
44              json_line = json.dumps(self.__dict__)
45              file.write(json_line + "\n")
46
47
48      def load(self):
49          with open(f"./results/{self.model}/{self.experiment_type}/"
50          "{self.samples}e_{self.max_tokens}t.jsonl", "r") as file:
51              for line in file:
52                  results = json.loads(line, object_hook=result_encoder)
53              return results
54
55  def result_encoder(r):
56      return Result(model=r['model'], experiment_type=r['experiment_type'],
        ↪  samples=r['samples'], max_tokens=r['max_tokens'],
57                    seeds=r['seeds'], syntax_validation=
                      ↪  r['syntax_validation'],
                      ↪  semantic_validation=r['semantic_validation'])
58
59  def gen_encoder(g):
60      return Generation(seed=g['seed'], elapsed_time=g['elapsed_time'],
        ↪  gen=g['gen'])
61
62  def fixed_seeds():
63      # Taking the 100 samples seed from the first 100 samples experiment
64      fixed_seeds = []
65      with open(f"./seeds/100.jsonl", "r") as file:
66          for line in file:
67              fixed_seeds = json.loads(line)
68      return fixed_seeds
69
70  def semantic_test(generation: str):
71      sp.sympify(generation)
72
73  def syntax_text(generation: str, parser):
74      parser.parse(generation).pretty()
75
76  def gen_preproc(generation:str):
77      eot_id_comparison = None
78      match EXPERIMENT_TYPE:
79          case "nogbd":
80              # Treatment for returning generation until <|eot_id|>
81              eot_id_gen = generation.split("<|eot_id|>")[0]
82              # Treatment for replace =(comparision) with == (used in no gbd
                ↪  generation), As using simpify (Python code simulation)
83              eot_id_comparison = eot_id_gen.replace("=", "==")
84
85          case "gbd":
```

```
            # Treatment for returning generation until <|eot_id|>
            eot_id_comparison = generation.split("<|eot_id|>")[0]

        case "gbd+fewshots":
            # Treatment for returning generation until <|eot_id|>
            eot_id_comparison = generation.split("<|eot_id|>")[0]

    return eot_id_comparison



# llama-3-70 quantized
llm = LLM(MODEL, gpu_memory_utilization=0.9, tensor_parallel_size=8,
↪  enforce_eager=False, quantization="gptq")
#llm = LLM('meta-llama/Llama-3.2-1B-Instruct', gpu_memory_utilization=0.9,
↪  tensor_parallel_size=8, enforce_eager=False, dtype="half")

tokens = [50, 100, 200]
experiments = ["gbd","nogbd","gbd+fewshots"]
for experiment_type in experiments:

    EXPERIMENT_TYPE = experiment_type

    for token in tokens:

        print(f"Experiment type: {experiment_type}")

        MAX_TOKENS = token

        # Relevant changes: == instead of =, ("==" expression)* instead of
        ↪  ("==" expression)? to allow multiple comparisions
        arithmetic_grammar = """
        ?start: comparison

        ?comparison: expression ("==" expression)* "<|eot_id|>"?

        ?expression: term (("+" | "-") term)*

        ?term: factor (("*" | "/") factor)*

        ?factor: NUMBER
            | "-" factor
            | "(" comparison ")"

        %import common.NUMBER
        %ignore " "  // Ignore spaces

        // Define <|eot_id|> as a terminal
        EOT_ID: "<|eot_id|>"
```

```
        """

        arithmetic_prompt = None

        match EXPERIMENT_TYPE:
            case "gbd":
                arithmetic_prompt=f"""
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

You are a helpful AI assistant for creating gramatically and sintactically arithmetic expression<|eot_id|>
<|start_header_id|>user<|end_header_id|>
Rewrite 9 * 15 as others equivalents expressions:
Follow this example:
(5*5)=(5+5+5+5+5)=(25*1)=(5*3)+(5*2).
<|eot_id|><|start_header_id|>assistant<|end_header_id|>
"""

            case "nogbd":
                arithmetic_prompt=f"""
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

You are a helpful AI assistant for creating gramatically and sintactically arithmetic expression<|eot_id|>
<|start_header_id|>user<|end_header_id|>
Rewrite 9 * 15 as other equivalent expression, for the response, do not use text.
Just only characters available in this grammar: {arithmetic_grammar}<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
"""

            case "gbd+fewshots":
                arithmetic_prompt="""
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

You are a helpful AI assistant for creating gramatically, equivalent and correct arithmetical expression<|eot_id|>
<|start_header_id|>user<|end_header_id|>
Given the following examples:\n
(5*5)=(5+5+5+5+5)=(25*1)=(5*3)+(5*2).\n
(3*3)=(3+3+3)=(3+6)=(9*1).\n
(3*4*5)=3*(2+2)*5=15*4=15*(2+2)=(12*5)=(20*3).\n
Rewrite 9 * 15 as others equivalents expressions:
<|eot_id|><|start_header_id|>assistant<|end_header_id|>
"""

            case "grammar_in_prompt":
                arithmetic_prompt=f"""
```

```
        <|begin_of_text|><|start_header_id|>system<|end_header_id|>

        You are a helpful AI assistant for creating gramatically and
        ↪  sintactically expression given this specific grammar:
        ↪  {arithmetic_grammar}<|eot_id|>
        <|start_header_id|>user<|end_header_id|>
        Rewrite 9 * 15 as others equivalents expressions:
        <|eot_id|><|start_header_id|>assistant<|end_header_id|>
        """

        grammar = arithmetic_grammar



        # Samples generations
        #seeds = [randint(1,SAMPLES*10e9) for i in range(SAMPLES)]
        seeds = fixed_seeds()
        only_generations = []
        elapsed_time_gens = []

        # Iterate experiments to generate completions
        for i in range(SAMPLES):
            seed = seeds[i]

            sampling_params = SamplingParams(
            max_tokens=MAX_TOKENS,
            temperature=1,
            top_p=0.95,
            seed= seed
            )

            start_time = time.perf_counter()

            outputs = None

            match EXPERIMENT_TYPE:
                case "gbd":
                    outputs = llm.generate(
                        prompts=arithmetic_prompt,
                        sampling_params=sampling_params,
                        guided_options_request=dict(guided_grammar=grammar))

                case "nogbd":
                    outputs = llm.generate(
                        prompts=arithmetic_prompt,
                        sampling_params=sampling_params,
                        )

                case "gbd+fewshots":
```

```
                    outputs = llm.generate(
                        prompts=arithmetic_prompt,
                        sampling_params=sampling_params,
                        guided_options_request=dict(guided_grammar=grammar))

            elapsed_time = time.perf_counter() - start_time
            print(f'Elapsed time for generation nº{i}: {elapsed_time} seconds')

            elapsed_time = None
            gen_text = None
            try:
                elapsed_time = outputs[0].metrics.finished_time - outputs[0].metrics.arrival_time
            except:
                elapsed_time = outputs.metrics.finished_time - outputs.metrics.arrival_time

            gen_text = None
            try:
                gen_text = outputs[0].outputs[0].text
            except:
                gen_text = outputs.outputs[0].text

            elapsed_time_gens.append(elapsed_time)
            only_generations.append(Generation(seed=seed, elapsed_time=elapsed_time, gen=gen_text))



        # Save and load results
        # Write the jsonl and serialize the gens
        model_foldername = MODEL.replace("/","::")
        with open(f"./samples/{model_foldername}/{EXPERIMENT_TYPE}/{SAMPLES}"
        "e_{MAX_TOKENS}t.jsonl", 'w') as file:
            for gen in only_generations:
                json_line = json.dumps(gen.__dict__)
                file.write(json_line + "\n")

        # Read the jsonl and deserialize back
        generation_from_file = []
        with open(f"./samples/{model_foldername}/{EXPERIMENT_TYPE}/{SAMPLES}"
        "e_{MAX_TOKENS}t.jsonl", "r") as file:
            for line in file:
                gen = json.loads(line, object_hook=gen_encoder)
                generation_from_file.append(gen)

```

```
266
267         # Checking syntax
268         syntactic_results=[]
269         parser = Lark(grammar, parser='lalr')
270
271         for gen in generation_from_file:
272
273             try:
274                 # Parse a generation
275                 gen_preprocesed = gen_preproc(gen.gen)
276                 syntax_text(generation=gen_preprocesed, parser=parser)
277                 syntactic_results.append(1)
278
279             except:
280                 syntactic_results.append(0)
281
282         print(f"total syntactically valid: {syntactic_results.count(1)}" )
283         print(f"total syntactically invalid: {syntactic_results.count(0)}" )
284
285         print(f"Percentaje syntactically valid:
            ↪  {(syntactic_results.count(1)/SAMPLES) * 100}%" )
286
287
288         # Checking semantic
289         semantic_results=[]
290
291         for gen in generation_from_file:
292
293             try:
294                 # Parse a generation
295                 gen_preprocesed = gen_preproc(gen.gen)
296                 semantic_test(generation=gen_preprocesed)
297                 semantic_results.append(1)
298
299             except:
300                 semantic_results.append(0)
301
302         print(f"total semantically valid: {semantic_results.count(1)}" )
303         print(f"total semantically invalid: {semantic_results.count(0)}" )
304
305         print(f"Percentaje semantically valid:
            ↪  {(semantic_results.count(1)/SAMPLES) * 100}%" )
306
307         res = Result(seeds=seeds, elapsed_time_gen= elapsed_time_gens,
            ↪  syntax_validation= syntactic_results,
            ↪  semantic_validation=semantic_results)
308         res.save()
```

8.2 Gráficas de tiempos

8.2.1. Llama 3 8B Instruct

NoGBD, Max_tokens= 50

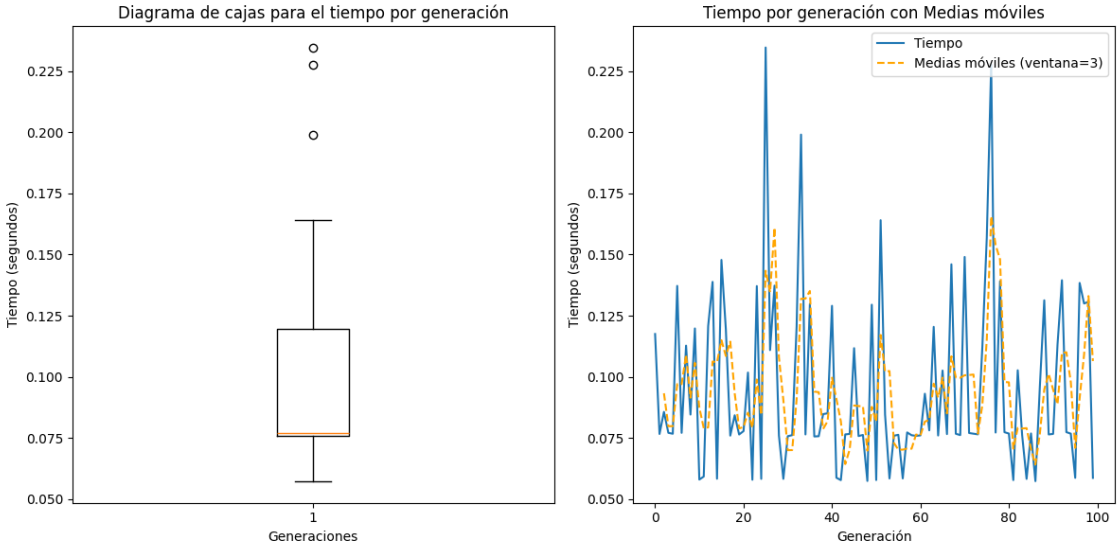

Figura 8.1. Llama 3 8B NoGBD max_tokens 50

GBD, Max_tokens= 50

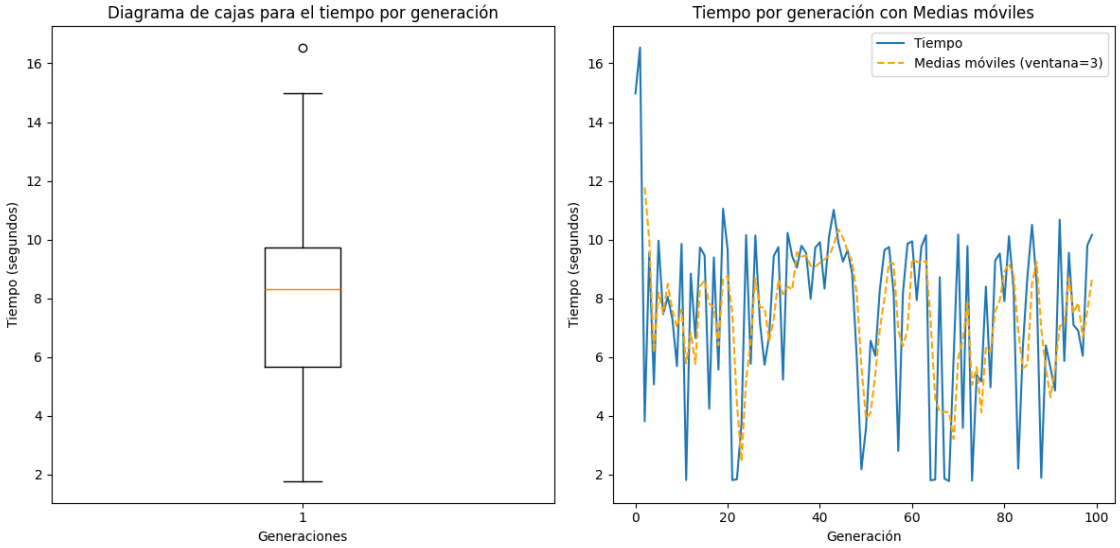

Figura 8.2. Llama 3 8B GBD max_tokens 50

GBD+FewShots, Max_tokens= 50

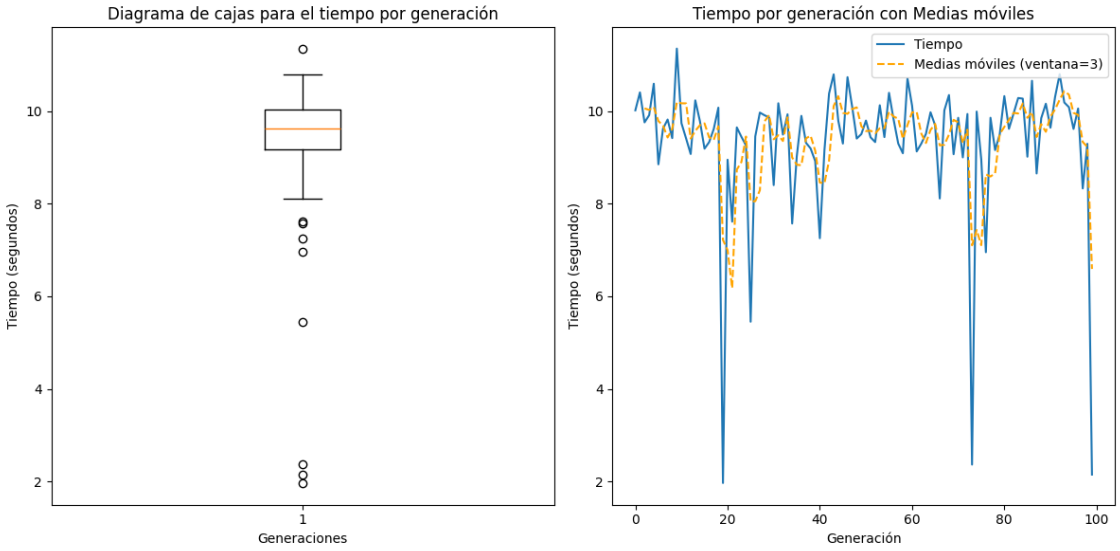

Figura 8.3. Llama 3 8B GBD+FewShots max_tokens 50

NoGBD, Max_tokens= 100

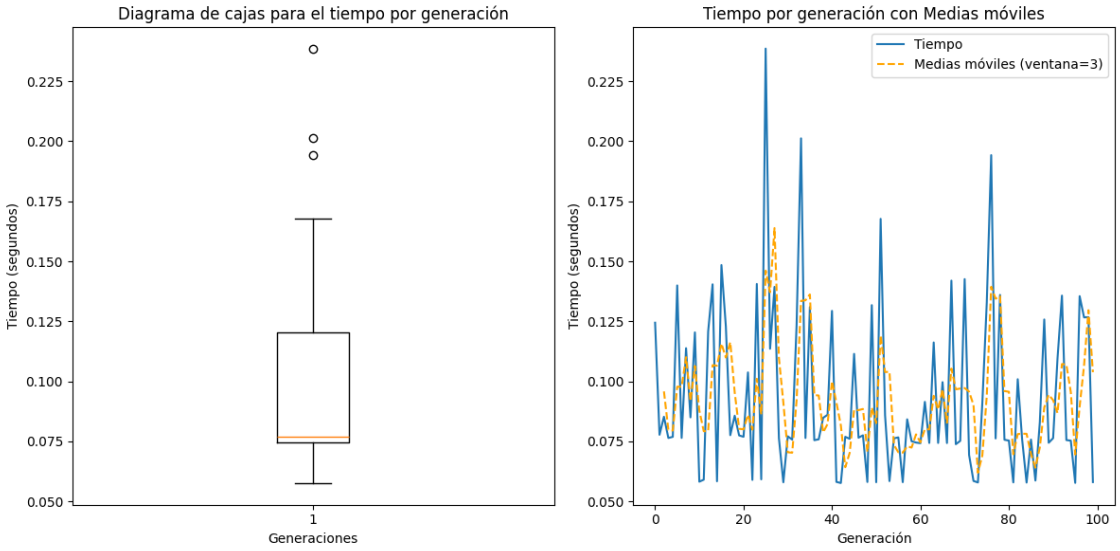

Figura 8.4. Llama 3 8B NoGBD max_tokens 100

GBD, Max_tokens= 100

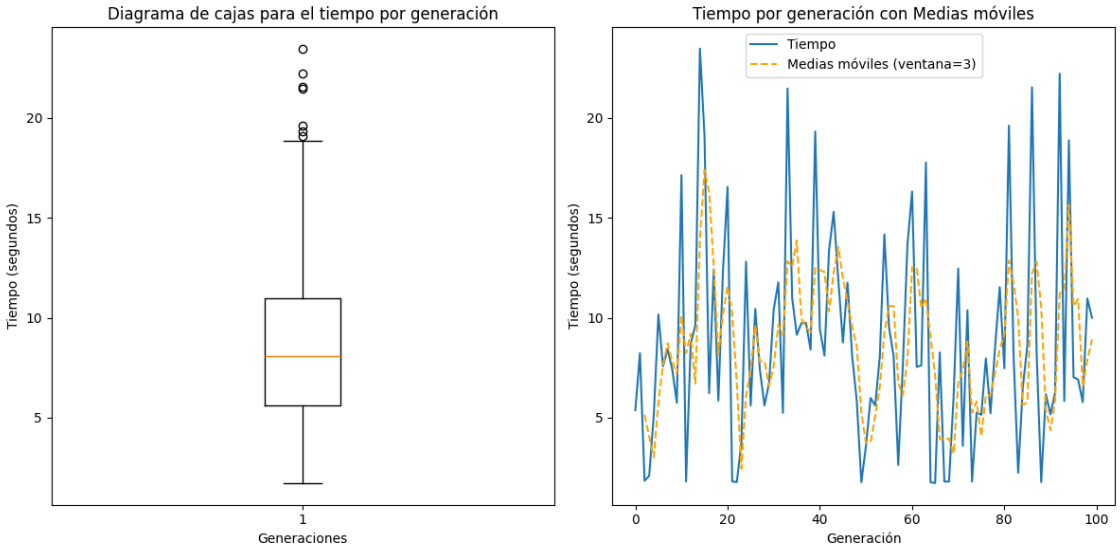

Figura 8.5. Llama 3 8B GBD max_tokens 100

GBD+FewShots, Max_tokens= 100

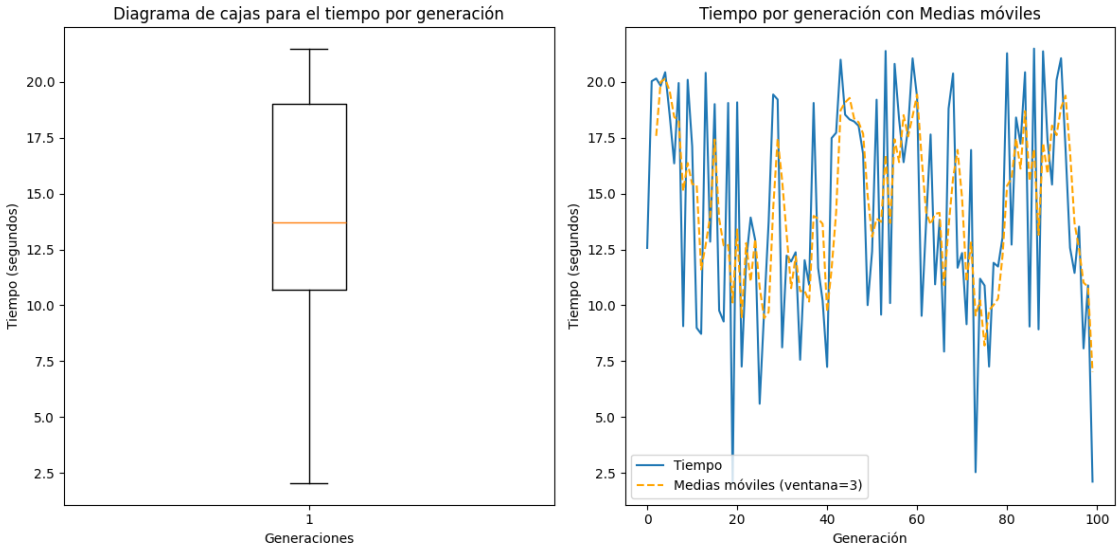

Figura 8.6. Llama 3 8B GBD+FewShots max_tokens 100

NoGBD, Max_tokens= 200

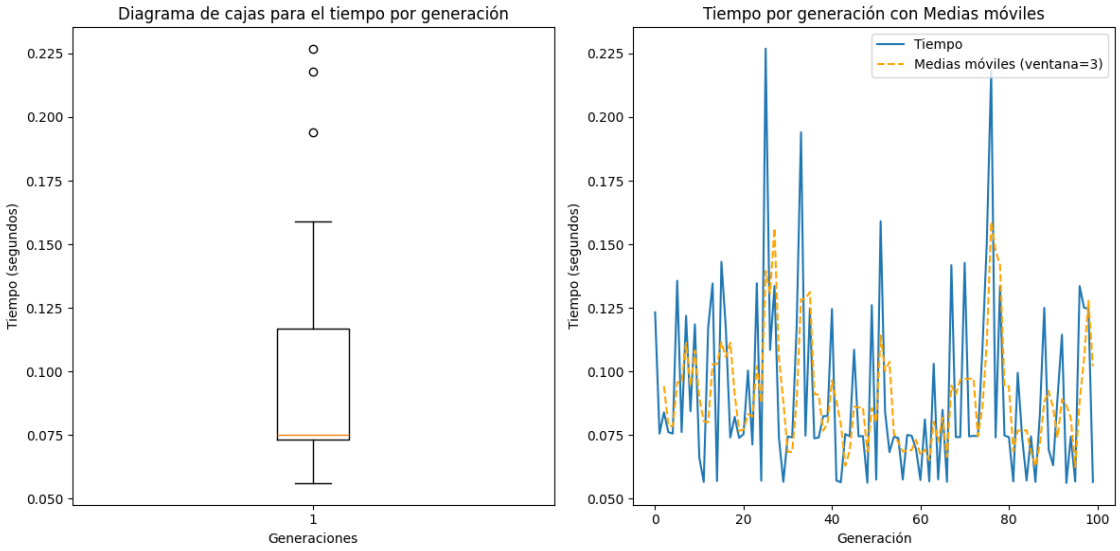

Figura 8.7. Llama 3 8B NoGBD max_tokens 200

GBD, Max_tokens= 200

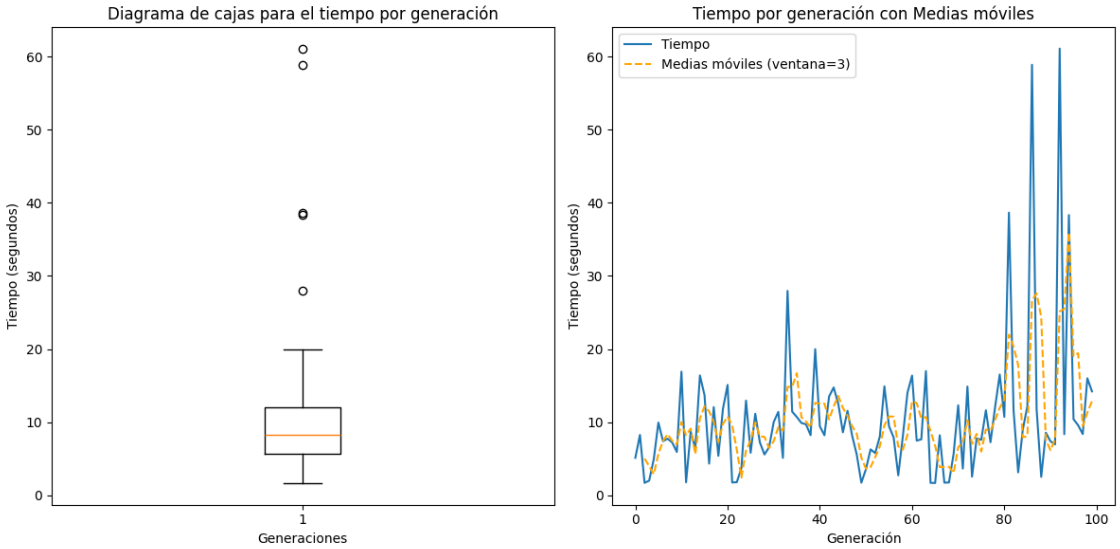

Figura 8.8. Llama 3 8B GBD max_tokens 200

GBD+FewShots, Max_tokens= 200

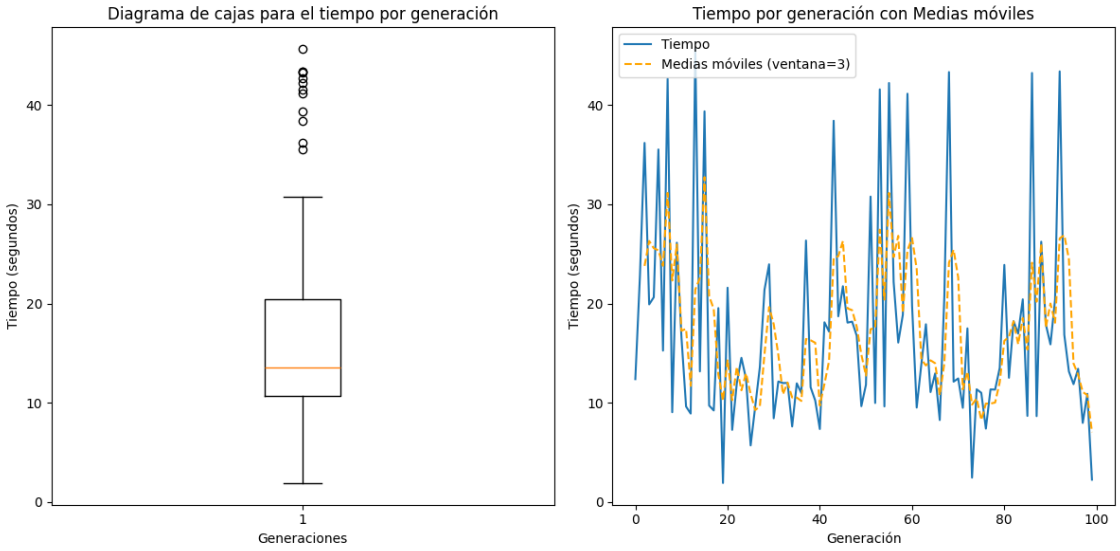

Figura 8.9. Llama 3 8B GBD+FewShots max_tokens 200

8.2.2. Llama 3 70B Instruct

NoGBD, Max_tokens= 50

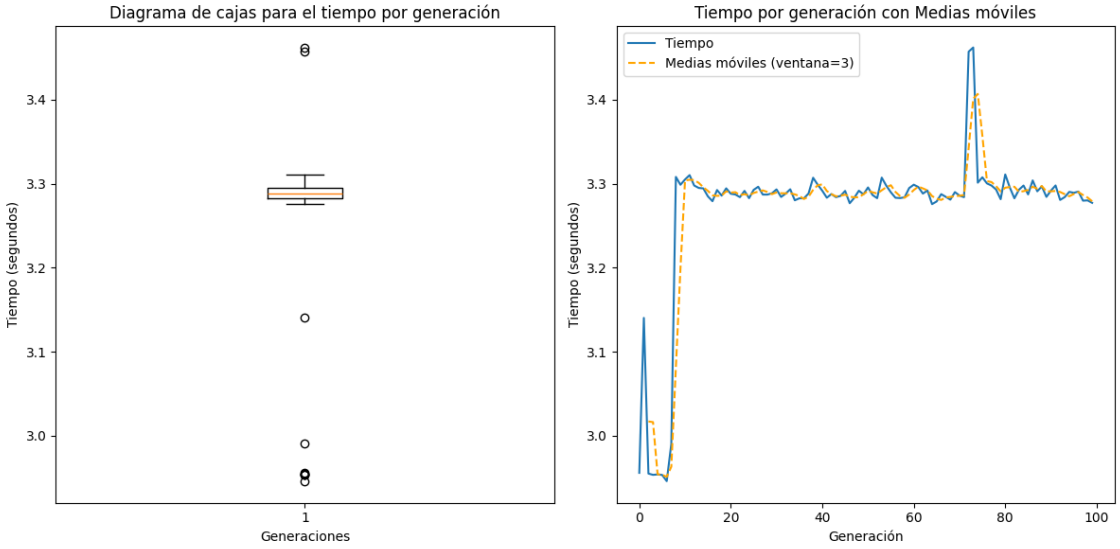

Figura 8.10. Llama 3 70B NoGBD max_tokens 50

GBD, Max_tokens= 50

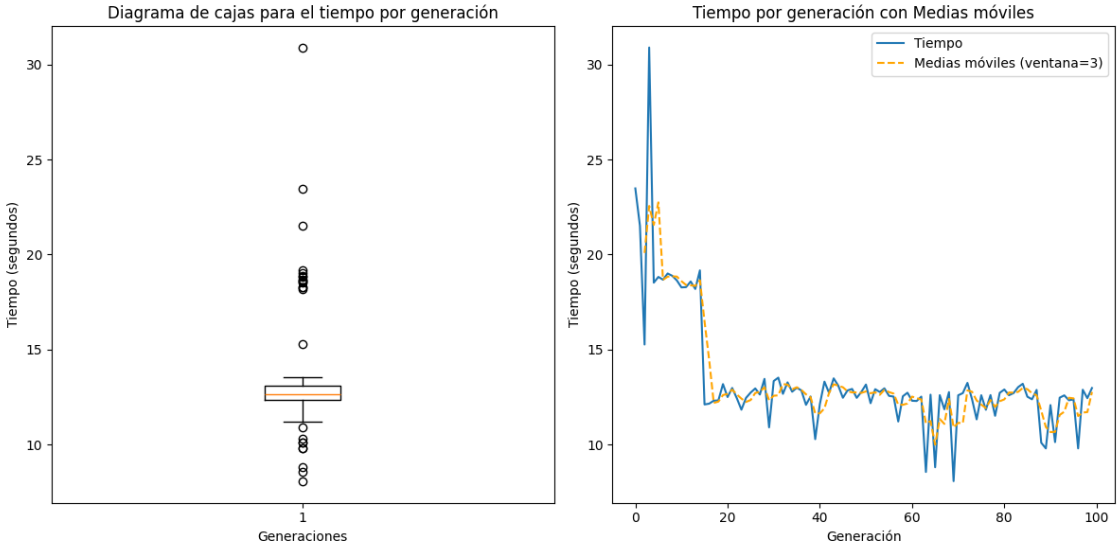

Figura 8.11. Llama 3 70B GBD max_tokens 50

GBD+FewShots, Max_tokens= 50

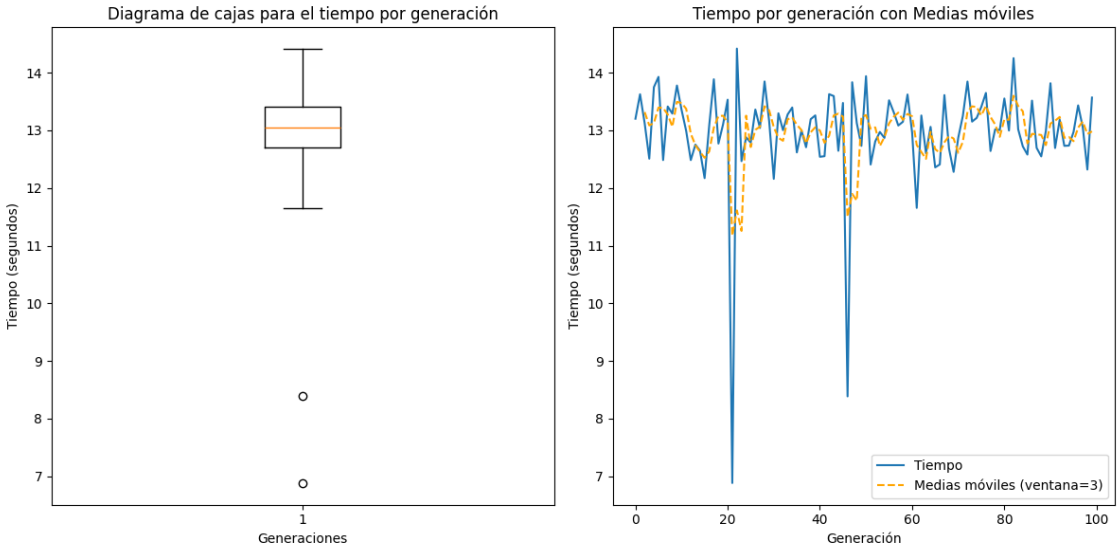

Figura 8.12. Llama 3 70B GBD+FewShots max_tokens 50

NoGBD, Max_tokens= 100

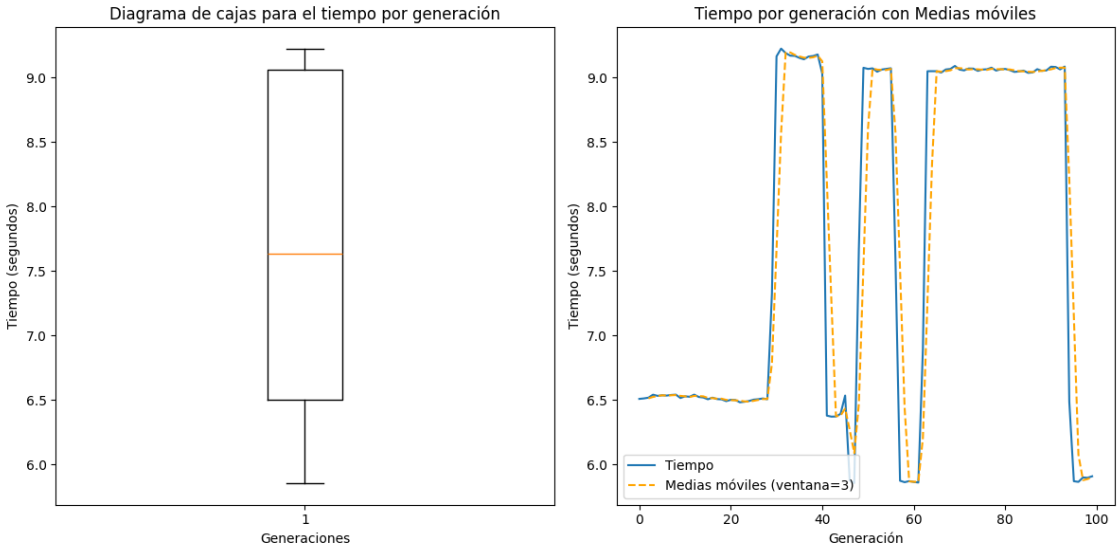

Figura 8.13. Llama 3 70B NoGBD max_tokens 100

GBD, Max_tokens= 100

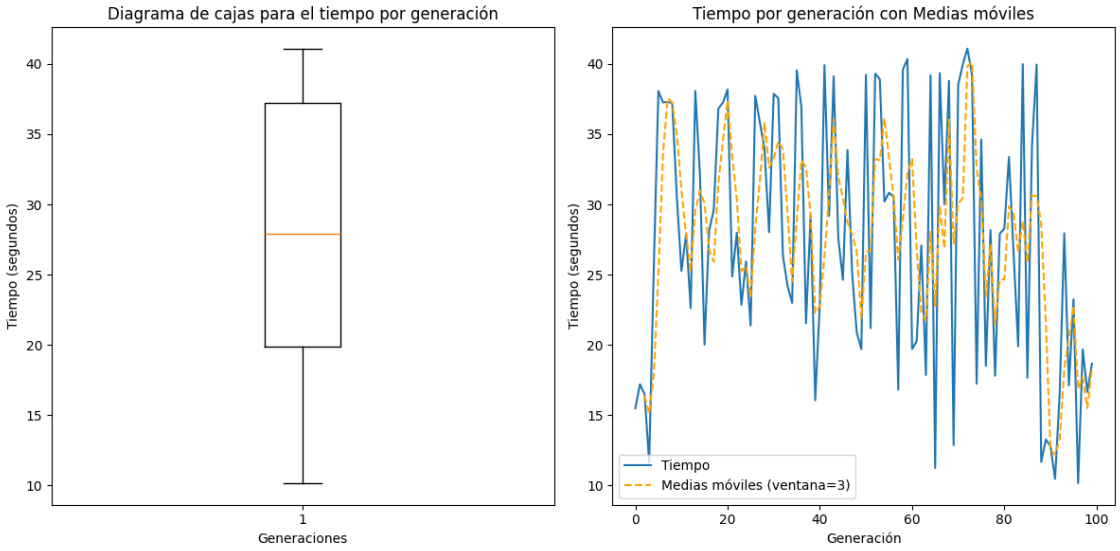

Figura 8.14. Llama 3 70B GBD max_tokens 100

GBD+FewShots, Max_tokens= 100

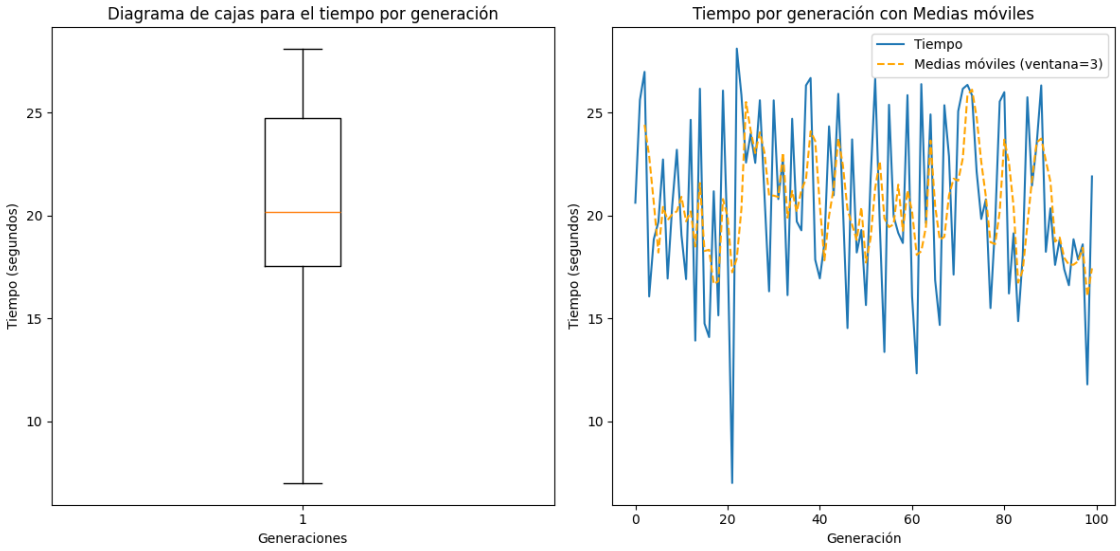

Figura 8.15. Llama 3 70B GBD+FewShots max_tokens 100

NoGBD, Max_tokens= 200

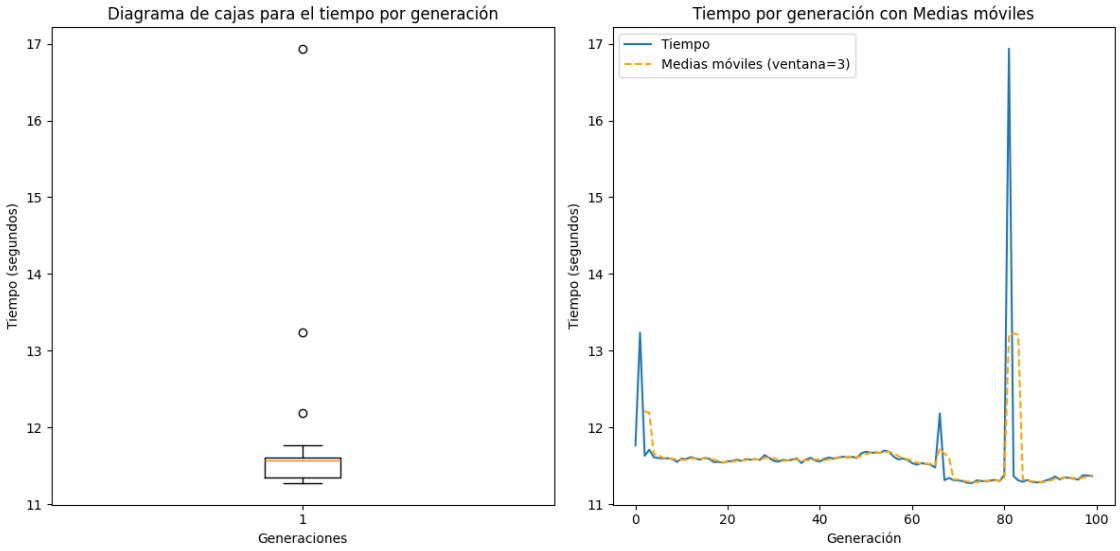

Figura 8.16. Llama 3 70B NoGBD max_tokens 200

GBD, Max_tokens= 200

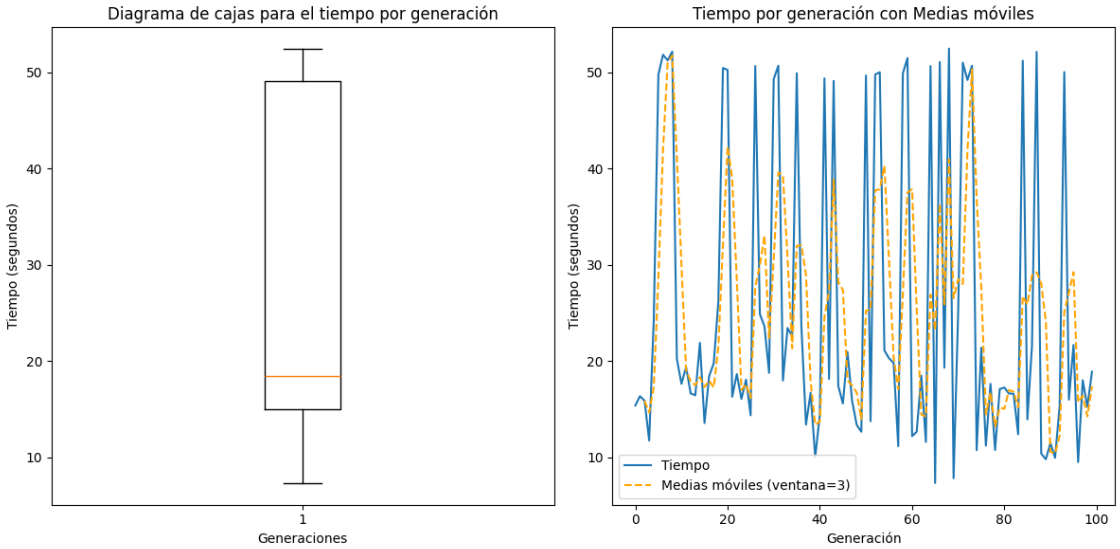

Figura 8.17. Llama 3 70B GBD max_tokens 200

GBD+FewShots, Max_tokens= 200

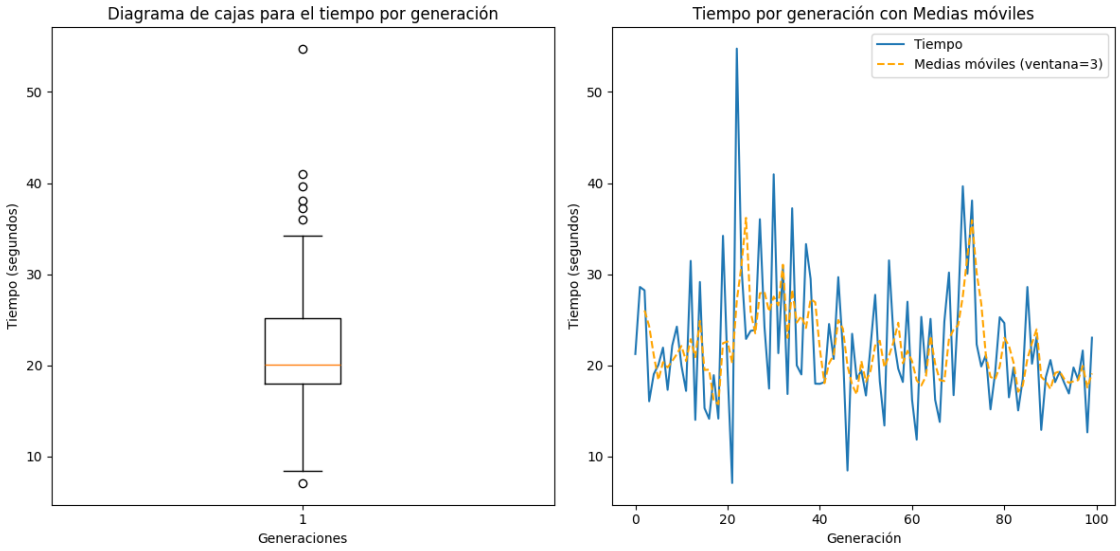

Figura 8.18. Llama 3 70B GBD+FewShots max_tokens 200

8.3 Gráficas de rendimiento

8.3.1. Llama 3 8B

nogbd

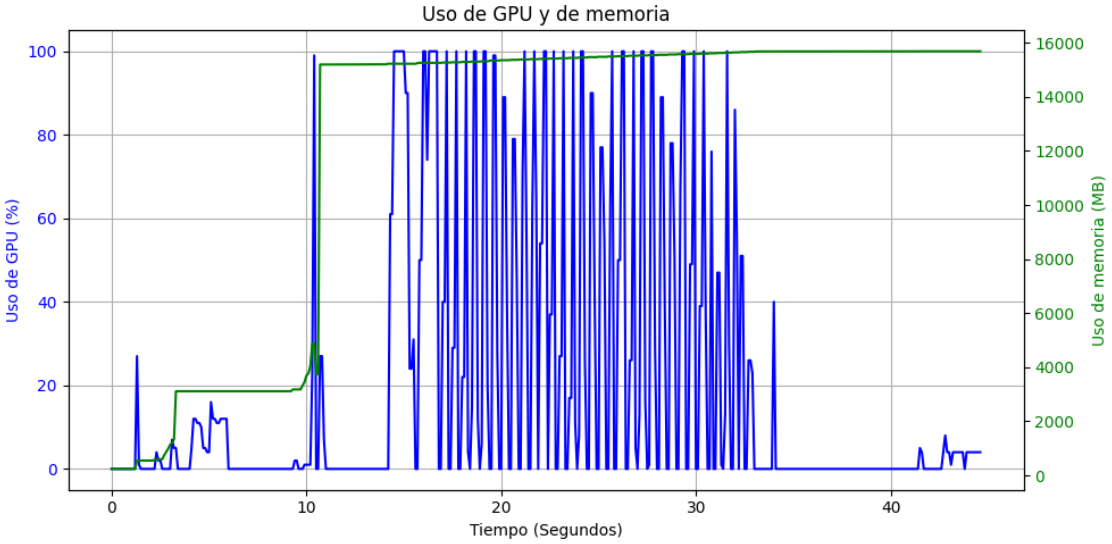

Figura 8.19. Llama 3 8B NoGBD

gbd

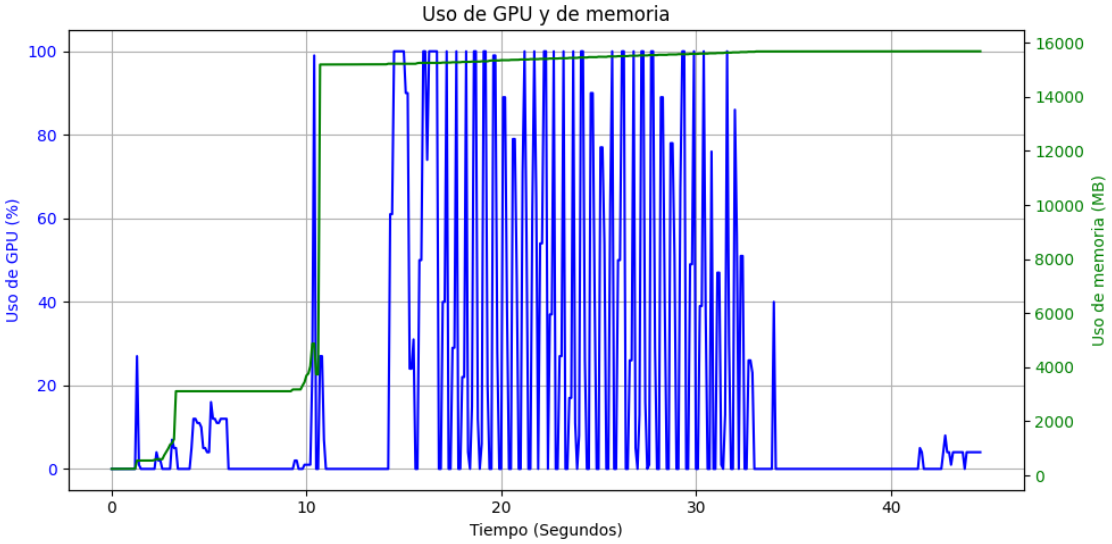

Figura 8.20. Llama 3 8B GBD

gbd+fewshots

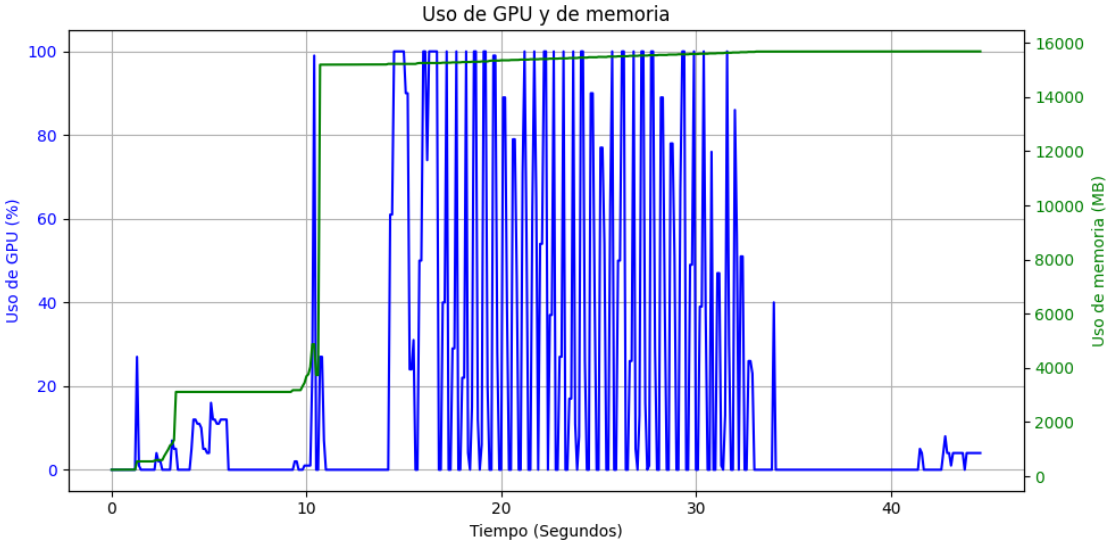

Figura 8.21. Llama 3 8B GBD+FewShots

8.3.2. Llama 3 70B

nogbd

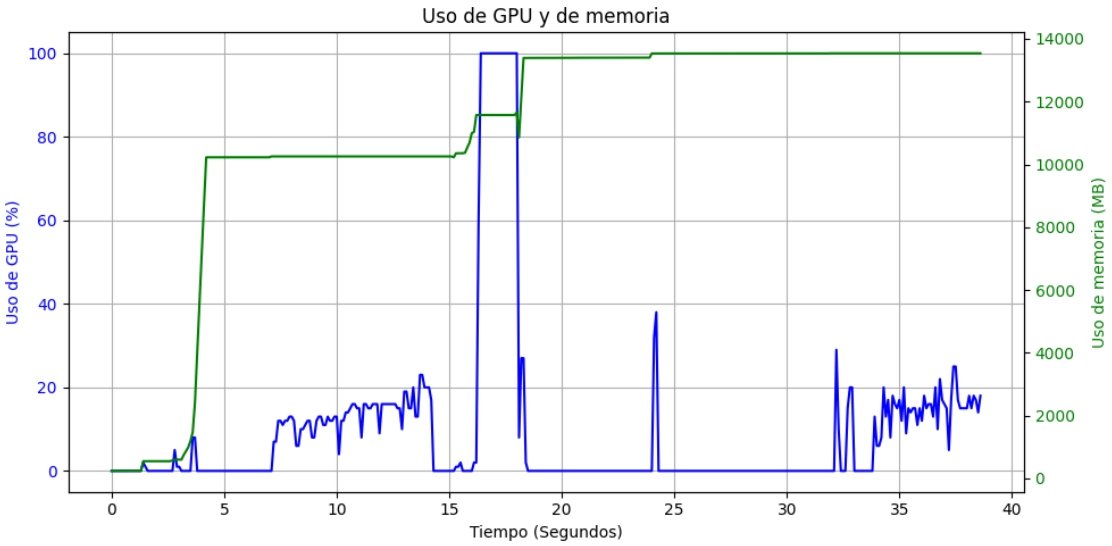

Figura 8.22. Llama 3 70B NoGBD

gbd

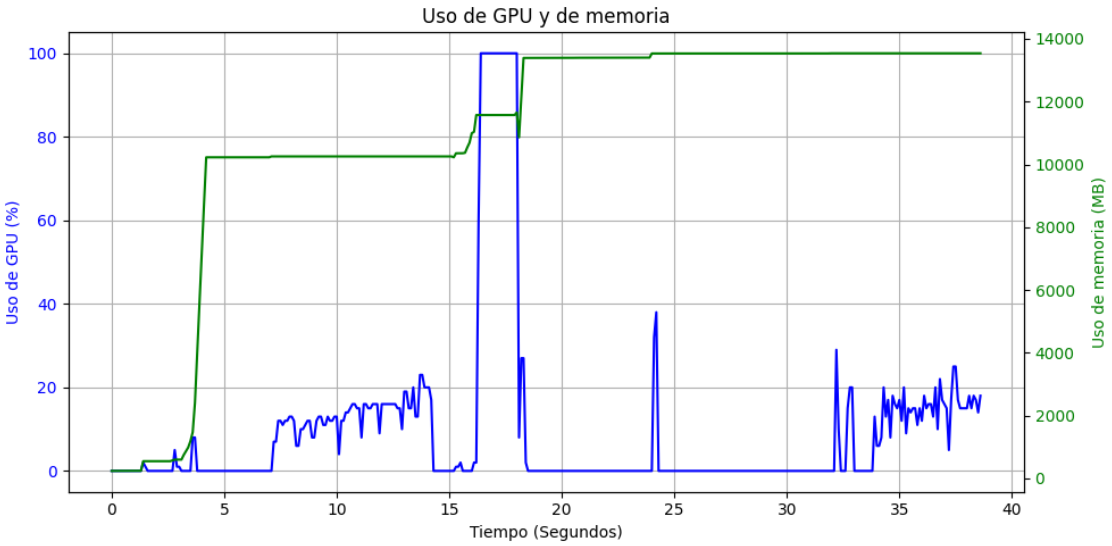

Figura 8.23. Llama 3 70B GBD

gbd+fewshots

Figura 8.24. Llama 3 8B GBD+FewShots