Asistente inteligente de código fuente basado en técnicas de inteligencia artificial

TRABAJO FIN DE MÁSTER, Máster Universitario en Inteligencia Artificial

Manuel Jesús Jiménez Navarro, Directores: Gualberto Asencio Cortés y Alicia Troncoso Lora

Resumen

El área del procesamiento del lenguaje natural ha obtenido grandes logros en los últimos años. La predicción de texto aplicado a código fuente es un área de gran interés en la actualidad. De hecho, los IDEs ofrecen ayuda a los programadores mediante sugerencias basadas en métodos estadísticos que facilitan las tareas de programación. Mejorar la capacidad de sugerencia de estos IDEs mediante una mayor comprensión del código fuente puede incrementar la productividad de los programadores en gran medida y reducir la escritura de código tediosa. Por lo tanto, la predicción de código fuente es uno de los campos que puede verse beneficiado por los recientes avances en el procesamiento del lenguaje natural. Estos avances han supuesto una mejora en el área del procesamiento del lenguaje natural en general y, en específico, una mejora en la generación de texto. En el presente trabajo, se ha realizado un estudio de la aplicación de las últimas arquitecturas y técnicas, con el objetivo de realizar predicciones de código fuente del lenguaje de programación Python. La generación de código será analizada con el objetivo de conocer si la solución es aplicable a un IDE, conocer si el modelo es capaz de "comprender" el código fuente y estudiar las ventajas e inconvenientes de cada método. Los resultados muestran que las arquitecturas basadas en Transformers mejoran considerablemente otros tipos de arquitecturas en la sugerencia de código fuente.

1. Introducción

La sugerencia de código fuente es de gran importancia para los desarrolladores, especialmente para aquellos que empiezan a aprender a programar o usan algún nuevo lenguaje de programación o framework. Los IDEs actuales son de gran ayuda, sin embargo, se encuentran bastante limitados en cuanto a las sugerencias que pueden llegar a aportar, especialmente en lenguajes no tipados como Python.

Recientemente, se han realizado grandes avances en el campo del procesamiento del lenguaje natural con la llegada de una nueva arquitectura llamada Transformer. Esta arquitectura, a diferencia de las anteriores basadas en redes neuronales, no basa su funcionamiento en capas recurrentes. En su lugar, basan su funcionamiento completamente en el mecanismo de Attention (usada como componente dentro de las redes recurrentes), más específicamente el Self-Attention. Este avance ha hecho posible obtener buenos resultados en la generación de texto, por lo que al ser la sugerencia de código fuente una tarea similar, pero en un contexto diferente al del lenguaje natural, un modelo basado en la arquitectura de Transformer es un buen candidato para obtener buenos resultados en la sugerencia de código fuente.

La sugerencia de código fuente es una tarea que ha cobrado gran interés en el campo del procesamiento de texto. Una de las principales ventajas es la facilidad de acceso a código gracias a repositorios públicos de código y a la gran cantidad de código abierto existente. Dentro del área de la generación de código fuente existen dos grandes retos, el tratamiento de palabras fuera del vocabulario y la capacidad de captar contextos muy alejados. El tratamiento de palabras fuera del vocabulario surge debido a que la programación es una tarea que a veces está sujeta a estándares únicos o características de la/s persona/s que la realizan, este hecho dificulta el proceso de construcción del vocabulario. La capacidad de captar contextos muy alejados es un hecho que se acentúa en la sugerencia de código debido a contextos muy alejados en el "tiempo" como variables o funciones declaradas al principio de un script.

Durante el presente trabajo, se aplicarán arquitecturas basadas en Transformer para poder lidiar con el problema de contextos lejanos en el "tiempo". Por otro lado, diferentes técnicas se han desarrollado en conjunto con la arquitectura Transformer para dar solución a las palabras que se encuentran fuera del vocabulario. Byte Pair Encoding es una técnica basada en la agrupación de caracteres para la generación de los tokens, de esta manera, palabras con prefijos, sufijos o conjunto de caracteres comunes formarán un token, lo que reducirá la cantidad de tokens fuera del vocabulario. Por otro lado, en los últimos años no solo se han desarrollado nuevas arquitecturas y algoritmos que han mejorado el área del procesamiento del lenguaje natural en general, también se han llevado a cabo avances dentro de las técnicas de generación de texto. Beam search fue hasta hace poco el algoritmo estándar para la generación de texto, sin embargo, recientemente se han estudiado técnicas alternativas(Holtzman et al., 2019) que mejoran la generación de texto. Durante el presente trabajo se analizarán estas arquitecturas observando si son aplicables a la sugerencia de código fuente.

En primer lugar, se comentarán trabajos relacionados, anteriores alternativas y aplicaciones similares. Posteriormente, se estudiará la metodología llevada a cabo para llevar a cabo la sugerencia de código fuente. Tras ello, se analizarán los resultados obtenidos con las diferentes alternativas a través de una serie de métricas. Por último, se expondrá el trabajo futuro y las conclusiones generales del trabajo realizado.

2. Trabajo relacionado

El proceso llevado a cabo para la sugerencia de código tiene como objetivo crear un modelo probabilístico tal que la salida sea una distribución de probabilidad $P(S)$; $S = w_1, w_2, ...w_n$, siendo S el conjunto de tokens pasados que conforman la entrada del modelo. Como salida, el modelo asignará una probabilidad a cada uno de los tokens del vocabulario que se corresponderá con el siguiente token sugerido, dependiendo únicamente de los tokens anteriores a este. Los modelos que realizan la tarea de asignar una

probabilidad a los tokens pertenecientes al vocabulario son conocidos como modelos lingüísticos autoregresivos. Los recientes avances han mejorado no solo los modelos linguísticos usados, tambien se han mejorado técnicas de preprocesado y de postprocesado para realizar tareas de generación de texto. Un elemento que tiene el mérito de haber logrado una mejora considerable en los últimos años es una nueva arquitectura de redes neuronales llamada Transformer (Vaswani et al., 2017) que, a diferencia de anteriores arquitecturas basadas en redes neuronales, no usa redes recurrentes. Otra técnica desarrollada, se basa en la aplicación del algoritmo Byte Pair Encoding (BPE) para la segmentación de palabras (Sennrich et al., 2015). Por último, dentro del campo de la generación de texto se han estudiado la aplicación de diferentes técnicas para evitar el problema de la "degeneración" de texto (Holtzman et al., 2019).

2.1. N-Gram

N-Gram ha sido una de las aproximaciones más usadas hasta la llegada de las redes neuronales, este modelo basa su funcionamiento en las cadenas de Markov de orden n, siendo n el número de tokens pasados usados para generar la distribución de probabilidad. Por lo tanto, la distribución de salida será la probilidad condicionada a los n tokens anteriores definida formalmente como:

$$P(S) = \prod P(w_{\mathrm{i}}|w_{\mathrm{i-n+1}}, ..., w_{\mathrm{i-1}})$$

Este modelo ha servido como base para las tareas de sugerencia de código fuente en numerosos estudios, incluso se ha usado conjuntamente con otros modelos usados para la sugerencia de código.
Christine Franks et al. (Franks et al., 2015) usaron el modelo N-Gram en conjunto con las sugerencias aportadas por el IDE Eclipse para mejorar estas. El plugin implementado en Java, usa las sugerencias ofrecidas de forma global a partir de todo el vocabulario o a partir de contextos locales (caché). Aunque este modelo mejoraba considerablemente las recomendaciones aportadas por el IDE Eclipse, la aproximación que se lleva a cabo en este trabajo no solo se limita a aportar sugerencias como nombre de métodos o variables, también aporta sugerencias sobre todos los posibles escenarios de los lenguajes de programación.

Veselin Raychev et al. (Raychev et al., 2014) implementaron el método SLANG que, a partir de un programa con ciertos huecos, usando una combinación de una red recurrente LSTM junto al modelo N-Gram logra detectar y completar los huecos del programa. El método basa su funcionamiento en el uso del análisis del programa para extraer secuencias e indexarlas a un modelo lingüístico que realiza la predicción. En el modelo que se realiza en este trabajo, el objetivo es diferente al de SLANG. El objetivo es lograr un sistema que sea capaz de funcionar en tiempo real al mismo tiempo que el usuario escribe en el programa, más que basar su funcionamiento en un análisis a posteriori.

2.2. Redes neuronales

Las redes neuronales han cobrado gran importancia en numerosas áreas, entre ellas, la sugerencia de código. Las

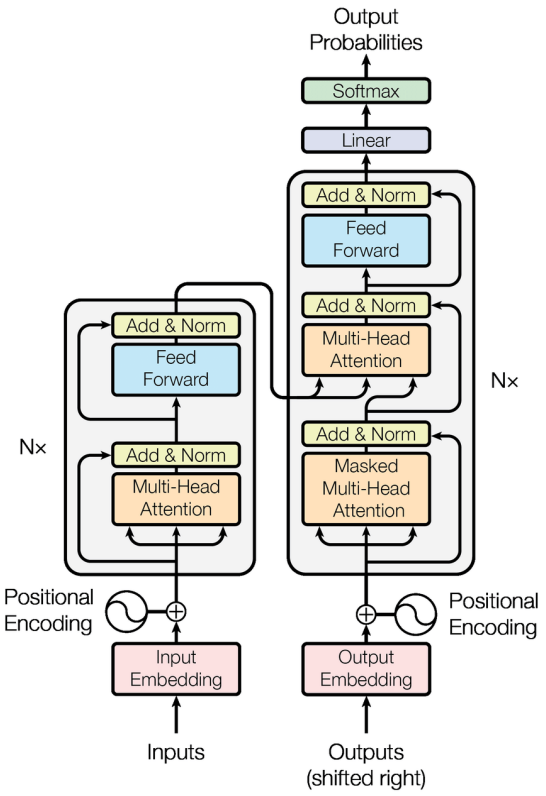

Figura 1: Arquitectura Transformer.

redes neuronales basan su funcionamiento en la construcción de una serie de transformaciones no lineales sobre la entrada, con el objetivo de obtener la distribución de probabilidades del siguiente token o secuencia de tokens. Las arquitecturas consideradas en el estado del arte, son las basadas en redes recurrentes con GRU (Gated Recurrent Unit), una modificación de la LSTM (Long Short Term Memory) y las basadas en la arquitectura Transformer. La principal diferencia entre una aproximación con redes neuronales y la realizada con N-Gram es que el modelado se realiza completamente del lenguaje y no de ciertos aspectos de este, como llamadas a funciones o propiedades de un objeto. Además, al mismo tiempo que se estudiaban nuevas arquitecturas de redes neuronales, también se implementeron otras técnicas complementarias que mejoraban considerablemente las sugerencias aportadas por los modelos.

Avishkar Bhoopchand et al. (Bhoopchand et al., 2016) estudiaron el uso de una "Sparse Pointer Network" para realizar un modelo lingüístico del lenguaje Python. Durante el estudio, los autores hacen hincapié en la importancia de los modelos lingüísticos para ser capaces de captar contextos muy lejanos, para ello, se hizo uso del mecanismo de Attention junto a una "Sparse Pointer Network". Como mecanismo de tokenización, se usó el ofrecido por el propio lenguaje Python, junto a una normalización de los nombres de funciones y variables. En el trabajo que se realiza en el presente documento, se aplican diferentes métodos a los usados en el artículo de Avishkar Bhoopchand et al.. se ha demostrado que los métodos usados en este trabajo presen-

tan mejores resultados gracias a un mejor tratamiento de los valores fuera del vocabulario y a una mejor capacidad de predicción que las redes recurrentes.

Seohyun Kim et al. (Kim et al., 2020) presentan un estudio de diferentes aproximaciones para realizar sugerencias de código mediante el uso de una modificación de la arquitectura Transformer. El modelo propuesto realiza una modificación sobre el mecanismo de Self-Attention añadiéndole información de la estructura del código en forma de un AST (Abstract Sintax Tree). La combinación de la información en forma de secuencia e información estructural del AST hace que el modelo obtenga mejores resultados que solo considerando información secuencial. A pesar de que la aproximación llevada a cabo es similar a la propuesta en este trabajo, las técnicas analizadas son distintas. Mientras que el trabajo de Seohyun Kim et al. se centra en la mejora estructural del propio modelo y su repercusión en los resultados, en este trabajo se mantienen las arquitecturas de los modelos originales con el objetivo de poder aprovechar el conocimiento ya aprendido en la implementación original (Transfer Learning), reduciendo así considerablemente recursos para el entrenamiento. Además, se hace mayor hincapié en las técnicas de preprocesado y postprocesado obteniendo buenos resultados sin necesidad de modificar la estructura del modelo.

Zeyu Sun et al. (Sun et al., 2019) abordan el problema de forma parecida a Seohyun Kim et al. en su trabajo. En este artículo, los autores estudian una modificación de la arquitectura del Transformer original añadiéndole un elemento nombrado como: subcapa convolucional estructural. Esta estructura tiene como objetivo captar la información estructural a partir del AST del código fuente, a esta estructura la llaman TreeGen. Los autores dividen el proceso llevado por el modelo en tres partes: un codificador del código fuente, el lector del AST que usa la subcapa estructural convolucional y el decoder que capta la información para predecir el siguiente token. La diferencia con el trabajo propuesto en este documento reside, al igual que el trabajo de Seohyun Kim et al., en que el presente estudio no se centra en la modificación de la estructura proporcionada por los transformers, adaptándola para la predicción de código. En su lugar, en este estudio se centra en un análisis del código fuente como secuencias aprovechando todo el conocimiento posible del procesamiento del lenguaje natural.

Rafael-Michael Karampatsis et al. (Karampatsis and Sutton, 2019) usan una red recurrente con GRU para construir un modelo lingüístico que mejora a anteriores trabajos realizados con N-Gram o redes neuronales recurrentes. La principal razón de sus buenos resultados, son causa del uso de Byte Pair Encoding para realizar la tokenización del código fuente. Además, los autores proponen un proceso de adaptación dinámica a otros contextos, basado en la actualización del modelo con el código de otro contexto. Esta actualización, limita los cambios en la red para evitar ajustarse demasiado al nuevo contexto aprendido. Esta aproximación aborda el preprocesado de manera similar a la propuesta en el presente trabajo, sin embargo, no aborda el problema de atender a contextos lejanos. Además, para la tarea de predicción basa su funcionamiento en el Beam Search, mientras que en este trabajo se analizan otras aproximaciones para la predicción de código.

3. Arquitecturas

Para realizar el entrenamiento y comparación de resultados, se han usado una serie de arquitecturas basadas en las aproximaciones que aportan mejores resultados en la actualidad. Estas arquitecturas se pueden categorizar en dos grupos: basadas en redes recurrentes y basadas en Transformer. Las distintas arquitecturas serán usadas para realizar el análisis de resultados en la tarea de sugerencia de código fuente.

3.1. GRU

Cho, et al. (Cho et al., 2014) publicaron en 2014 la arquitectura conocida como GRU (Gated Recurrent Unit), esta arquitectura nace posteriormente a la arquitectura LSTM (Long Short Term Memory) como una versión un poco más simple y eficiente de esta. Tanto la arquitectura LSTM como la GRU nace con el objetivo de solucionar el problema de desvanecimiento del gradiente. Este problema afectaba al proceso de entrenamiento de las redes neuronales al reducirse el valor del gradiente como efecto de las continuas actualizaciones realizadas, afectando en gran medida a las redes recurrentes.

El funcionamiento de la arquitectura GRU, se basa en la combinación de la información de entrada junto a la producida anteriormente. Para combinar los datos de entrada junto a la salida producida en el anterior ciclo, se usan una serie de "puertas" que controlan la cantidad de "pasado" (salida t-1) y de "presente" (entrada en t+1) que se usará para producir la salida de la red neuronal (salida t+1). Las "puertas" usadas en la arquitectura son:

- Puerta de actualización: Determina que cantidad de salida producida en t-1 y de entrada en t se debe mantener.
- Puerta de reajuste: Determina que información de salida producida en t-1 se debe olvidar.

La información filtrada mediante el uso de las puertas se combina y se produce la salida en el momento t+1.

Esta arquitectura ha sido usada para numerosas aplicaciones dentro y fuera del área del procesamiento del lenguaje natural, debido a su eficiencia y a sus resultados parecidos a la arquitectura LSTM. Actualmente se usa como modelo base para el tratamiento de datos secuenciales.

3.2. Seq2Seq

Ilya Sutskever et al. (Sutskever et al., 2014) publicaron en 2014 la arquitectura conocida como Seq2Seq, que usa la arquitectura LSTM con la estructura de tipo Encoder-Decoder. Esta arquitectura, surge con el objetivo de dar flexibilidad a la hora de obtener un tamaño de secuencia de salida diferente a la entrada en problemas como traducción automática (Neural Machine Translation). Su funcionamiento se basa en dos componentes principales: el encoder y el decoder. La tarea del encoder es codificar la información de entradas pasadas, para posteriormente

pasar su último estado al decoder. El decoder toma el último estado del encoder, que contiene información de entradas pasadas codificadas junto a las entradas actuales para producir la salida.

A partir de las arquitecturas basadas en redes recurrentes (GRU, LSTM, Seq2Seq), surgieron varias mejoras para obtener mejores resultados. Una de las mejoras que mayor impacto tuvo fue el mecanismo de Attention. Las redes recurrentes proveen de un estado por cada ciclo dentro del bucle interno recurrente, sin embargo, a la hora de computar las salidas solo se tiene en cuenta el último estado de este ciclo, que comprime toda la información de todos los ciclos pasados. Dzmitry Bahdanau et al. (Cho et al., 2014), propuso el mecanismo de Attention con el objetivo de que las redes recurrentes fueran capaces de captar contextos más lejanos que los que actualmente eran capaces a través del uso de todos los estados pasados. Su funcionamiento intenta prestar mayor "atención" a estados pasados que resulten relevantes para la salida en el momento presente.
La arquitectura Seq2Seq junto al mecanismo de Attention, ha sido usada en numerosas aplicaciones del procesamiento del lenguaje natural y, hasta no hace mucho, era considerada el estado del arte en varias tareas dentro de esta área.

3.3. GPT-2

Alec Radford et al. (Radford et al., 2019a) publicaron en 2019 la segunda versión de su modelo lingüístico conocido como GPT-2 (Generated Pre-trained Transformer 2). Este modelo tiene su base en la arquitectura del Transformer, por lo que los componentes principalmente son los mismos. La arquitectura de Transformer posee una estructura Encoder-Decoder y, a diferencia de anteriores aproximaciones, no usa redes recurrentes. En lugar de redes recurrentes, el funcionamiento del Transformer se basa íntegramente en el mecanismo de Attention. Los componentes principales de la arquitectura Transformer son:
Positional Encoding: Este componente se encarga de asignar una posición relativa a cada entrada para poder recordar el orden de entrada. Este componente es necesario debido que, a diferencia de las redes recurrentes, el concepto de orden no existe por lo que es necesario asignar a cada token la posición correspondiente.
Multi-Head Attention: El mecanismo de Attention original, calcula los pesos de la entrada en función de la influencia de cada token sobre los demás dentro de la secuencia. El mecanismo Multi-Head Attention va un paso más allá, este mecanismo se compone de varios módulos de Attention que funcionan independientemente. Esta mejora permite que diferentes módulos (Heads) "atiendan" a diferentes características de la secuencia.

El encoder y el decoder poseen la misma estructura y los mismos componentes, sin embargo, el decoder recibe la salida del encoder a la hora de realizar el cálculo de la salida. En la Figura 1 se puede ver una representación de la arquitectura Transformer, mostrando la interacción de los distintos componentes.

La principal diferencia entre la arquitectura Transformer y la usada en el primer GPT, reside en que su funcionamiento se basa solo en el uso de múltiples decoders (sin recibir información del encoder). Originalmente la arquitectura Encoder-Decoder fue pensada ser usada en tareas de traducción, sin embargo, investigaciones realizadas por los autores de GPT concluyeron que una arquitectura badada solo en decoders podría usarse para mútiples tareas. Posteriormente, se lanzó GPT-2 que entrenaba la arquitectura sobre un conjunto de datos mayor y con un mayor número de decoders.

La arquitectura GPT-2 ha sido entrenada para diferentes tareas, obteniendo un conjunto de textos de la plataforma Reddit en un contexto no supervisado. Entre las múltiples tareas en la que ha sido entrenado, se encuentra la generación de texto. Este modelo es considerado actualmente el estado del arte en la generación de texto obteniendo muy buenos resultados en publicaciones como (Radford et al., 2019b) y (Ziegler et al., 2019). Además, existen numerosas aplicaciones realizadas que demuestran de forma práctica su buen funcionamiento en diversos contextos.

3.4. Sinkhorn

Yi Tay et al. (Tay et al., 2020) publicaron en 2020 una modificación sobre la arquitectura del Transformer para mejorar la eficiencia del mecanismo de Attention. El mecanismo llamado "Sparse Sinkhorn Attention", basa su funcionamiento en el "Meta Sorting Network", con el objetivo de realizar el cálculo de los valores del mecanismo de Attention cuasi global sobre ventanas locales de secuencias. El componente "Meta Sorting Network" basa su funcionamiento en el balance diferenciable de Sinkhorn (Séjourné et al., 2019). Además, los autores proponen la decodificación casual de Sinkhorn para contextos autorregresivos dado que el mecanismo original considera secuencias futuras y pasadas.
La arquitectura propuesta, presenta una mejora considerable en la eficiencia del entrenamiento comparado con las otras arquitecturas basadas en Transformers, manteniendo la eficacia.

4. Metodología

Para realizar la experimentación, se usaron todas las arquitecturas descritas en la sección anterior, de esta manera, se realiza la comparación entre dos aproximaciones basadas en redes recurrentes y dos basadas en la arquitectura Transformer.
A diferencia de las anteriores arquitecturas, la arquitectura GPT-2 se encuentra preentrenada. Aunque el proceso de entrenamiento no se ha realizado exclusivamente sobre código fuente, se ha demostrado experimentalmente que la arquitectura preentrenada posee la capacidad para sugerir código fuente. Además, dado que en el presente trabajo se trata el código fuente en forma secuencial, el conocimiento adquirido para la generación de texto (código fuente o no) puede resultar beneficioso para la sugerencia de código fuente. Por lo tanto, esta arquitectura se dividirá en tres aproximaciones diferentes:

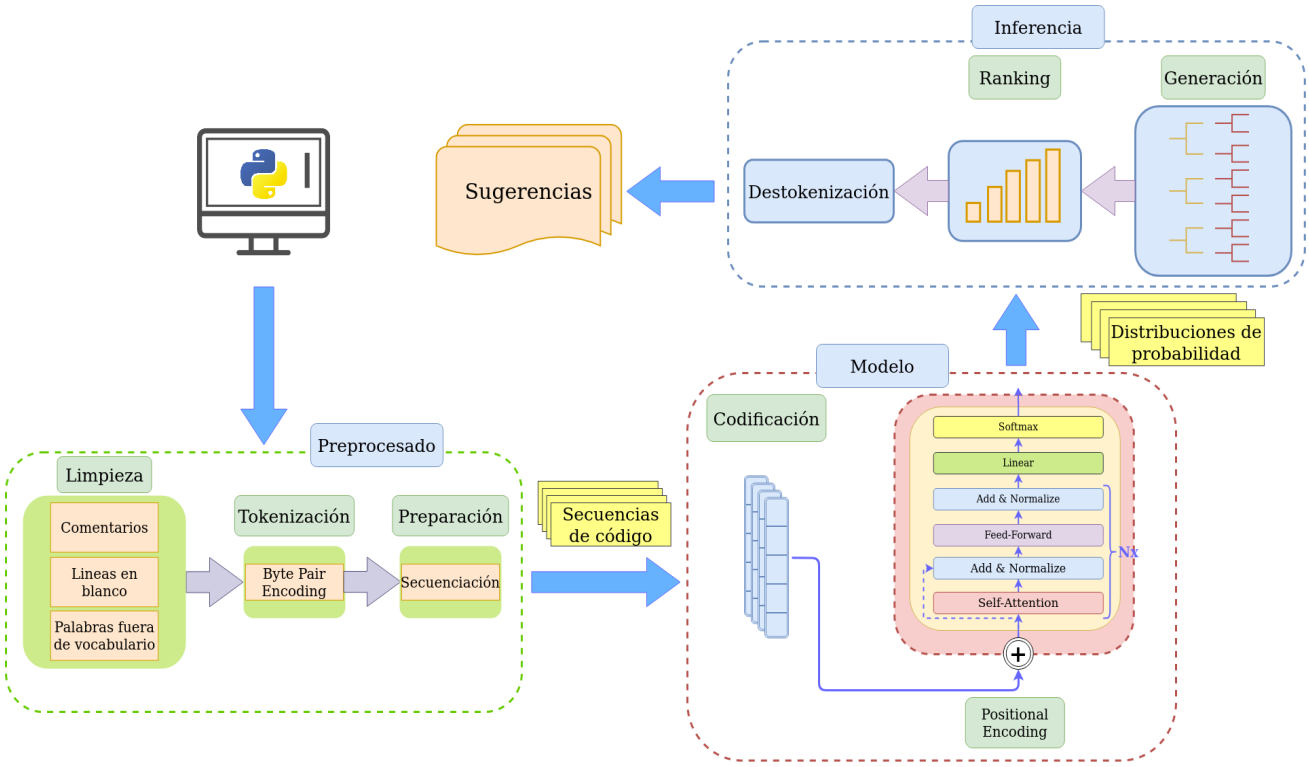

Figura 2: Flujo de trabajo del proceso de sugerencia.

- Arquitectura GPT-2 (GPT-2 Raw) sin pesos preentrenados y entrenada sobre el conjunto de datos obtenido para el presente trabajo.
- Arquitectura GPT-2 (GPT-2 Pretrained) con pesos preentrenados sin entrenar sobre el conjunto de datos obtenido para el presente trabajo.
- Arquitectura GPT-2 (GPT-2 Finetuned) con pesos preentrenados y "afinada" sobre el conjunto de datos para el presente trabajo.

El conjunto de datos escogido, consiste en código fuente escrito en el lenguaje de programación Python. La elección del lenguaje Python se debe a que, al ser un lenguaje de tipado dinámico, la sugerencia de código se dificulta en comparación a otros lenguajes tipados. Los datos fueron extraídos de la plataforma Github usando el mismo conjunto de repositorios que los usados en (Bhoopchand et al., 2016). Se realizó un análisis de las etiquetas más usadas entre los repositorios, entre estas etiquetas se encontraba la librería Flask, por lo que se decidió usar los repositorios de esta librería para formar el conjunto de entrenamiento. Flask es un framework minimalista para la construcción de aplicaciones web en python, se trata de un framework comúnmente usado en la actualidad y usada en numerosos proyectos. El contexto de las aplicaciones web se ajusta perfectamente a los objetivos del presente trabajo, debido a lo monótono que puede resultar su programación a los desarrolladores. Por lo tanto, de todos los repositorios, se escogieron aquellos que tuvieran la etiqueta de Flask o cuyo nombre lo contuviera. Estas librerías, fueron obtenidas realizando un ranking sobre el número de estrellas (stars) y bifurcaciones (forks) del repositorio. Unos valores altos en estrellas y bifurcaciones, pueden considerarse como un indicio de que un repositorio posee una buena calidad de código. Por lo tanto, el modelo tratará de sugerir código que posea una calidad similar a la aprendida, esperando que se sigan las mejores prácticas.

Para eliminar el "ruido", facilitar el entrenamiento de los distintos modelos y preparar los datos para los modelos, se realizó un preprocesado sobre el conjunto de entrenamiento. En primer lugar, se eliminaron todos los comentarios y todas las líneas en blanco. Dado que el objetivo no es generar comentarios y, aunque las líneas en blanco forman parte de las buenas prácticas de formato, los IDEs proveen normalmente de herramientas para formatear el código de forma automática, por lo que no es necesario que el modelo aprenda a seguir dicho formato. En segundo lugar, se dividió el conjunto de datos en tres conjuntos: entrenamiento, validación y test, cada una de estas tres partes corresponderían con el 70 %, 10 % y 20 % del conjunto de datos inicial respectivamente. Por último, se realizó la tokenización usando la técnica Byte Pair Encoding entrenada sobre el conjunto de entrenamiento. Posteriormente la tokenización fue aplicada sobre todo el conjunto de datos, con el objetivo de que el conjunto de validación o test no influyeran sobre los resultados. En el caso de las arquitecturas GPT-2 que usan pesos preentrenados, se aplicará el mismo tipo de tokenización, pero entrenada sobre el conjunto de entrenamiento usado en el artículo original para evitar incompatibilidades.

Para asegurar la mayor igualdad posible entre las distintas arquitecturas para su justa comparación, se usaron los mísmos hiperparámetros y métodos durante el entrenamiento para todas y cada una de las arquitecturas. Como

tamaño del batch y factor de aprendizaje, se escogió un valor de 32 y 5e-5 respectivamente. El tamaño de la secuencia escogida es de 100, por lo que se usarán los 100 tokens pasados para predecir el siguiente token. El algoritmo de optimización escogido es el Adam propuesto en (Loshchilov and Hutter, 2017) que soluciona errores cometidos en varias librerías respecto al uso del parámetro "weight decay". Como función de pérdida se usó el "Categorical Crossentropy loss", función de pérdida comunmente usada para problemas de clasificación. Se llevó a cabo una actualización del factor de aprendizaje mediante el uso de un "learning rate scheduler" lineal que lo reduce con el paso de las épocas. Todas las arquitecturas fueron entrenadas hasta la convergencia con un máximo de 100 épocas.

En cuanto a los parámetros escogidos para las distintas arquitecturas, se ha controlado el número de capas y neuronas en cada capa. Para cada una de las arquitecturas se realizaron pruebas con un rango de capas entre 1 y 12 con un máximo de 1024 neuronas en cada capa. Como tamaño de espacio de codificación de tokens escogido, se realizaron pruebas para un tamaño entre 128 y 768. La configuración de capas, número de neuronas y tamaño de espacio que mejores resultados aportara fue la escogida finalmente. Todas las arquitecturas entrenadas son del tipo muchos a muchos, es decir, se tomarán 100 tokens pasados para predecir los 100 tokens desplazados un paso, añadiendo el siguiente token y excluyendo el primero de la secuencia de entrada.
Las métricas escogidas para evaluar la eficacia de las distintas arquitecturas fueron: precisión, precisión top 5 y perplejidad. La precisión mide el porcentaje de acierto respecto a la predicción con mayor porcentaje y el valor real, mientras que la precisión top 5 mide el acierto con respecto a las 5 predicciones con mayor porcentaje y el valor real de predicción. Por otro lado, la perplejidad se trata de una medida de error que evalúa la capacidad del modelo para predecir la distribución de probabilidad con respecto a los valores reales.
La precisión top 5 se ha escogido ya que normalmente un IDE no da solo una posible sugerencia, en su lugar, aporta varias posibilidades para que el desarrollador pueda escogerla. Por ello, dado que el objetivo es realizar una sugerencia, el hecho de que la sugerencia esté entre los 5 posibles valores más probables se contará como un acierto.

El método de evaluación de las arquitecturas se hará de dos formas: medir toda la secuencia de salida de las arquitecturas y medir solo el/los token/s a predecir durante la generación. Es decir, se evaluarán las métricas de la salida obtenida por las arquitecturas comparadas con la entrada desplazada, además, se realizará la generación de los n tokens siguientes, siendo n como máximo 5, y se compararán con los siguientes n tokens reales. Por lo tanto, se evaluará la capacidad de predecir la secuencia un paso en adelante obteniendo así una estimación de como la arquitectura es capaz de captar la progresión temporal de las secuencias para el siguiente token. Por otro lado, dado que el objetivo del trabajo es predecir los siguientes tokens, se lleva a cabo la evaluación de la capacidad de generación de las arquitecturas.

Modelo	Acc	Acc@5	PP
Gru	0.43	0.62	33.45
Seq2Seq	0.80	0.85	4.87
Sinkhorn	0.93	0.95	1.70
GPT-2 Raw	0.46	0.63	122.27
GPT-2 Pretrained	0.58	0.76	12.71
GPT-2 Finetuned	0.74	0.87	4.25

Tabla 1: Resultados de la entrada desplazada.

Para la generación de sugerencias, se llevó a cabo la comparación de tres métodos usados para la generación de texto en lenguaje natural: Beam Search, Top K y Top P. Beam Search, es el método de generación más usado para la generación del texto, este método surge como mejora del greedy search realizando una búsqueda de los n tokens que en conjunto tienen mayor probabilidad. Top K, es un método de muestreo que, a diferencia de Beam Seach, selecciona los K tokens con mayor probabilidad, eliminando los tokens fuera del contexto, y realiza un muestreo teniendo en cuenta la probabilidad asignada por el modelo a esos tokens. Top P surge como mejora del Top K cuando las distribuciones asignan una alta probabilidad a pocos tokens. Este método toma los tokens que tienen una probabilidad conjunta P solventando el problema del método Top K. EN la figura 2 se observa el flujo de trabajo completo desde la entrada de código fuente hasta la obtención de sugerencias.
La diferencia entre la generación mediante Beam Search y un método basado en muestreo se discute en (Holtzman et al., 2019). En este artículo los autores estudian la forma en la que las personas "generan" palabras en comparación a como lo realiza Beam Search. Un problema conocido en el area del procesamiento del lenguaje natural con Beam Search, es la tendencia a la repetitividad de las palabras generadas conforme aumenta el tamaño n de generación. Los autores argumentan que el problema de la repetitividad de Beam Search se debe a que las personas normalmente escogen palabras que no son las más probables, por lo que la generación de palabras de las personas se ajustan más a un método basado en sampling. Durante los resultados, se estudiará si la generación de código fuente se ajusta más a un método basado en sampling (con mayores variaciones) o no.

5. Resultados

5.1. Predicción de entrada desplazada

La Tabla 1 muestra la precisión, precisión@5 y la perplejidad para el conjunto de test de la librería Flask. Los resultados mostrados para la predicción desplazada por las distintas arquitecturas, muestran que la arquitectura Sinkhorn posee los mejores resultados. Los resultados aportados por la arquitectura Sinkhorn supera con creces en todas las métricas a la arquitectura Seq2Seq y GPT-2 Finetuned.
Por otro lado, aunque la arquitectura Seq2Seq aporta mejores resultados de precisión que GPT2-Finetuned, la precisión@5 es un poco menor. El hecho de que la precisión@5

Beam Search			
Modelo	**NSalidas**	**Acc**	**Acc@5**
Seq2Seq	1	0.06	0.12
	2	0.06	0.11
	3	0.05	0.11
	4	0.05	0.10
	5	0.05	0.10
Sinkhorn	1	0.22	0.39
	2	0.11	0.22
	3	0.09	0.18
	4	0.07	0.15
	5	0.06	0.13
GPT-2 Raw	1	0.33	0.54
	2	0.16	0.28
	3	0.12	0.21
	4	0.09	0.17
	5	0.08	0.15
GPT-2 Pretrained	1	0.67	0.83
	2	0.51	0.65
	3	0.45	0.58
	4	0.40	0.53
	5	0.37	0.49
GPT-2 Finetuned	1	0.78	0.90
	2	0.58	0.68
	3	0.50	0.61
	4	0.44	0.55
	5	0.41	0.51

Tabla 2: Resultados de la generación con la técnica Beam Search.

sea menor, sugiere que la arquitectura Seq2Seq posee una menor capacidad de captar el contexto del problema ya que, a pesar de que su precisión es mayor, la capacidad para predecir una distribución de probabilidades es ligeramente peor que en el caso de la arquitectura GPT-2.

Por último, cabe destacar los resultados de GPT-2 Raw que, a pesar de poseer una arquitectura que a priori esperaba dar mejores resultados, no posee unos resultados destacables. De hecho, los resultados son similares al modelo GRU mostrando una perplejidad extremadamente alta haciéndolos los peores modelos. A pesar de poseer la misma arquitectura tanto GPT-2 Raw como GPT-2 Finetuned, la diferencia tan grande en resultados muestra la necesidad de esta arquitectura de un amplio conjunto de datos. De hecho, la arquitectura GPT-2 Pretrained (one-shot) muestra mejores resultados a pesar de no haber sido entrenada exclusivamente en código, lo cual confirma lo anteriormente explicado.

A pesar de que aparentemente la arquitectura Sinkhorn es claramente superior a las demás, los resultados obtenidos no resultan fiables para evaluar la capacidad de generación de código. En la próxima sección se estudiará la generación de los distintos tipos de arquitecturas que mejores resultados han aportado, dejando fuera la arquitectura GRU.

5.2. Sugerencia

La Tabla 2,3 y 4 muestran la precisión y la precisión@5 para diferente número de salidas, aplicado sobre el conjunto de test de la librería Flask para los métos Beam Search, Top K y Top P respectivamente. Los resultados de la evaluación de la generación muestran unos resultados diferentes que los presentados en el apartado anterior. Los resultados aportados por las arquitecturas GPT-2 muestran unos resultados bastante mejores comparados con las demás arquitecturas. Teniendo en cuenta solo las arquitecturas GPT-2, el orden con respecto a la precisión se ha mantenido. GPT-2 Finetuned presenta los mejores resultados seguido de GPT-2 Pretrained y por último GPT-2 Raw. En el caso de GPT-2 Finetuned, los resultados de la generación resultan mejores comparados con los resultados de todas sus salidas. Este hecho, puede indicar que el modelo ha sido capaz de generalizar de mejor manera la capacidad de generación que la capacidad de establecer la casualidad de los eventos de entrada.

En cuando a la arquitectura Sinkhorn, su funcionamiento parece mostrar que la capacidad de generalización para predecir las entradas desplazadas y establecer la causalidad de los tokens de entrada es mucho mejor que su capacidad de generación.

Por otro lado, las arquitecturas basadas en redes recurrentes muestran unos peores resultados comparado con las arquitecturas basadas en Transformer. La arquitectura Seq2Seq que presentaba los mejores resultados tras la arquitectura Sinkhorn, ha sufrido una reducción de la precisión en la misma medida.

El hecho de que las arquitecturas basadas en redes recurrentes y la arquitectura Sinkhorn hayan tenido una reducción de la precisión tan grande, se puede deber a que el volumen de datos con el que fueron entrenados no es suficiente para generalizar correctamente la generación de texto. Aun así, dentro de las arquitecturas que han usado exclusivamente la información del conjunto de datos, existen algunas arquitecturas que son capaces de extraer un mayor conocimiento para la generación. Las arquitecturas basadas en Transformer (Sinkhorn y GPT-2 Raw) muestran una mayor capacidad de generación a pesar de que el volumen de datos parece no ser totalmente el adecuado. De hecho, la arquitectura GPT-2 Raw parece centrarse más en la capacidad de generación que en la capacidad de predecir la distribución de causalidad de la entrada. Este hecho se observa debido a que la reducción de la precisión es mucho menor comparada con las arquitecturas Sinkhorn y Seq2Seq. Por lo tanto, la arquitectura GPT-2 parece una arquitectura que presenta mejores resultados para tareas de generación en general, mientras que las demás no son arquitecturas que funcionen tan bien en este contexto con este volumen de datos.

En cuanto a la distribución de resultados respecto el número de salidas, el comportamiento en casi todos los casos es similar. Aparentemente, cuantos más tokens de salida genere una arquitectura menor será la precisión. Sin embargo, su evolución no es lineal, la evolución parece asintótica para este número de salidas.

Los resultados muestran como la mayoría de las arquitecturas poseen una precisión menor al 10 % para las 4 o 5 entradas independientemente de la técnica de generación. Este deterioro de la precisión parece afectar en mayor manera a la precisión@5, los resultados muestran como la precisión

Top K			
Modelo	**NSalidas**	**Acc**	**Acc@5**
Seq2Seq	1	0.06	0.08
	2	0.06	0.07
	3	0.05	0.06
	4	0.04	0.06
	5	0.04	0.05
Sinkhorn	1	0.19	0.29
	2	0.10	0.16
	3	0.07	0.13
	4	0.06	0.10
	5	0.05	0.09
GPT-2 Raw	1	0.32	0.37
	2	0.16	0.20
	3	0.12	0.14
	4	0.09	0.11
	5	0.08	0.09
GPT-2 Pretrained	1	0.59	0.74
	2	0.46	0.56
	3	0.41	0.50
	4	0.37	0.45
	5	0.34	0.42
GPT-2 Finetuned	1	0.75	0.83
	2	0.55	0.60
	3	0.48	0.52
	4	0.43	0.47
	5	0.40	0.42

Tabla 3: Resultados de la generación con la técnica Top K.

Top P			
Modelo	**NSalidas**	**Acc**	**Acc@5**
Seq2Seq	1	0.06	0.07
	2	0.05	0.06
	3	0.05	0.06
	4	0.04	0.06
	5	0.04	0.05
Sinkhorn	1	0.19	0.28
	2	0.10	0.15
	3	0.07	0.12
	4	0.06	0.10
	5	0.05	0.09
GPT-2 Raw	1	0.32	0.37
	2	0.16	0.19
	3	0.12	0.14
	4	0.09	0.11
	5	0.08	0.09
GPT-2 Pretrained	1	0.59	0.72
	2	0.45	0.56
	3	0.40	0.49
	4	0.36	0.44
	5	0.34	0.41
GPT-2 Finetuned	1	0.76	0.83
	2	0.56	0.60
	3	0.49	0.52
	4	0.43	0.46
	5	0.40	0.42

Tabla 4: Resultados de la generación con la técnica Top P.

y la precisión@5 son cada vez más cercanas. Este hecho, parece indicar que la generación de un conjunto de predicciones (en este caso 5) converge a la mejor opción cuando el número de salidas crece. El efecto de convergencia de los conjuntos de resultados generados no afecta a todas las técnicas de la misma manera, Beam Search presenta una "velocidad" de deterioro de la precisión@5 menor que las técnicas basadas en muestreo.

La evolución asintótica es una cualidad que resulta de gran importancia dado que, para un numero de salidas muy grande, se puede establecer una aproximación de cómo sería la precisión. Sin embargo, el deterioro de la precisión@5 indica que los conjuntos de tokens generados dejan de tener relevancia cuantas más salidas se obtenga, es decir, para un número de salidas n considerablemente grande no tendría sentido realizar una sugerencia de 5 conjuntos de n tokens.

La Tabla 1 muestra el comportamiento de Beam Search, mientras que la Tabla 2 y 3 muestra el comportamiento de los métodos basados en muestreo para diferente número de salidas. Los dos tipos de métodos de generación usados presentan comportamientos diferentes. Top P y Top K presentan unos resultados muy similares, mientras que Beam Search es ligeramente superior en cuanto a la precisión. La mayor diferencia entre los métodos basados en muestro y Beam Search reside en la precisión@5, los resultados son claramente peores que los aportados por Top P y Top K y además el deterioro es mucho mayor en los métodos basados en muestreo como ya se comentó anteriormente. El hecho de que el deterioro se produzca antes en las técnicas basadas en muestreo hace que la sugerencia de varias opciones pierda sentido mucho antes en estos métodos. En el contexto de este trabajo la precisión@5 resulta de gran interés, por lo que el hecho de que los métodos basados en muestreo produzcan unos peores resultados no los hace una técnica de generación adecuada. Las técnicas basadas en muestreo surgieron para solucionar el problema de la "degeneración" en la generación de texto, el objetivo era hacerlas más similares a la forma en la que las personas "generamos" las palabras. Sin embargo, los resultados muestran que en el caso de la generación de código el comportamiento es más monótono y sin una elección de tokens tan variada.

5.3. Análisis de interpretabilidad

Con el fin de comprender mejor el funcionamiento interno del modelo que mejores resultados ha aportado, se analizarán los pesos aportados por el mecanismo de Attention y algunas sugerencias realizadas.

En la Figura 3 se muestra un ejemplo del funcionamiento interno del modelo GPT-2 Finetune. En este ejemplo se importa un módulo específico de la librería Flask y se instancia la clase "BaseSQL" heredando de una clase del módulo previamente importado ("BaseFilter").

Puede observarse que el conjunto de tokens representado

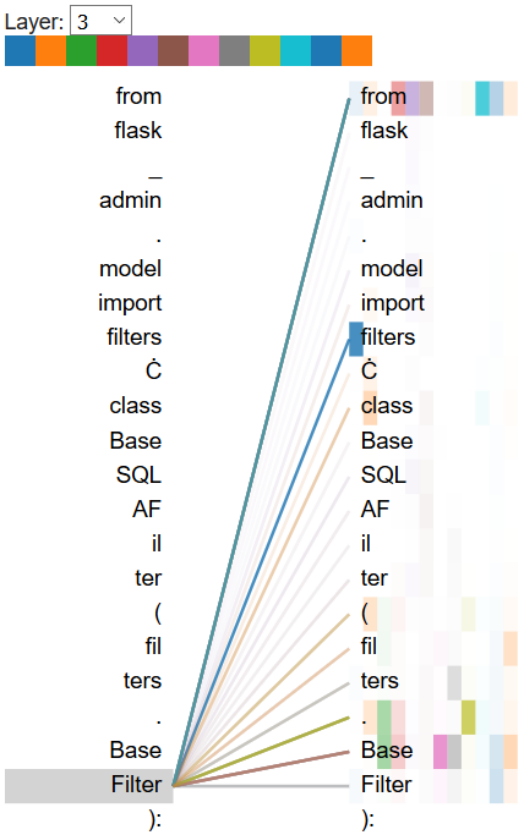

Figura 3: Ejemplo de asignación de pesos para GPT-2 Finetune.

ha realizado una división lógica del código. Esto se debe al uso del Byte Pair Encoding que anteriormente se ha comentado. Además, un detalle curioso aparece en el token "filters". Este token aparece en dos ocasiones en el código, uno a la hora de importarlo y otro al obtener la clase "BaseFilter". Sin embargo, en la primera ocasión aparece como "filters" y la segunda se separa en "fil" y "ter" debido a que el contexto es diferente. Este hecho puede resultar perjudicial para el rendimiento del modelo al tratar de forma diferente el mismo concepto.

Los pesos del mecanismo de Attention son representados como líneas que unen pares de tokens. Cuanto mayor valor tenga el peso asignado, mayor intensidad tendrá la línea. Los distintos colores representan las doce diferentes cabeceras (Heads) del mecanismo Multi-Head Attention. Los pesos asignados representan la importancia que tienen los tokens anteriores sobre el token "Filter" en la tarea de predicción.

La distribución de pesos muestra como varias cabeceras asignan pesos altos a los tokens "." y "Base". Otro conjunto más pequeño de cabeceras asignan pesos altos a los tokens "(", "fil" y "ters".

Las cabeceras que asignan pesos altos al token "Base" parecen centrarse en atender a los tokens que tienen una pertenencia común. Por ejemplo, se podría decir que el token "Base" forma parte del token "Filter" al conformar un mismo nombre. Además, existen dos cabeceras que atienden a los tokens "fil" y "ter" dando a entender que también existe una relación de pertenencia entre el módulo y la clase en cierta manera.

Por otro lado, las cabeceras que asignan pesos altos al token "." parecen centrarse en atender conceptos más estructurales del código. Se puede observar que una misma cabecera también posee unos pesos altos en "class" y "(". Por lo tanto, parece indicar que esa cabecera se centra en elementos comunes (palabras clave) del código.

Los pesos más altos se centran en el módulo, es decir, en el token "filters". Esta cabecera parece indicar que el módulo posee gran importancia dado que se trata de una clase perteneciente a este.

Hay que tener en cuenta que los pesos asignados al primer token carecen de relevancia, dado que cuando el modelo no tiene conocimiento sobre dónde atender por defecto se asignan pesos altos al primer token.

Por lo tanto, aparentemente el modelo ha aprendido eficazmente conceptos de los datos que posteriormente apoyan a las sugerencias de código que realiza.

6. Conclusiones y trabajo futuro

En el presente trabajo se han aplicado diferentes arquitecturas y técnicas de generación de texto a la tarea de sugerencia de código. La evaluación de resultados se ha realizado de dos formas diferentes, una forma evalúa el desempeño de las arquitecturas como modelo multisalida aprendiendo la causalidad de los datos y otro como tarea de sugerencia. Los resultados muestran que las arquitecturas basadas en GPT2 con los datos escogidos, aportan las mejores sugerencias debido a que han sabido adaptarse mejor a las tareas de sugerencia de código de forma más eficiente que otras arquitecturas. La diferencia de resultados entre las dos formas de evaluación, muestran que la primera forma de evaluación no representa adecuadamente la eficacia de las distintas arquitecturas en la tarea de predicción.

La gran diferencia entre las arquitecturas que usan Transfer learning y las que no, muestran que la cantidad y variedad de datos resulta de gran importancia para el aprendizaje de las distintas arquitecturas. Por lo tanto, es necesario invertir una gran cantidad de recursos en el entrenamiento de un modelo que ofrezca buenos resultados. Sin embargo, se ha observado que modelos entrenados en el contexto de la generación del lenguaje natural mayormente pueden adaptarse al contexto de generación de código fuente. Por lo tanto, aprovechando el conocimiento adquirido en modelos entrenados en el contexto del lenguaje natural es posible ahorrar una gran cantidad de recursos.

Las distintas técnicas de generación usadas presentan grandes diferencias. Las técnicas basadas en muestreo (Top K y top P), obtienen resultados similares entre ellas y un poco peores que Beam Search. La diferencia reside en la pérdida de variabilidad existente por parte de las técnicas basadas en muestreo repercutiendo en la precisión@5. Este hecho parece indicar que la generación de código resulta menos impredecible y normalmente sigue una distribución más estable, por lo que la aleatoriedad de los métodos basados en muestreo le perjudica. Por otro lado, las técnicas existentes presentan una evolución asintótica respecto al número de tokens de salida. Sin embargo, la variabilidad de resultados se ve perjudicada en mayor medida, por lo que para

un número de salidas grande, presentar varias alternativas de sugerencia deja de tener sentido.

En el futuro, sería interesante llevar a cabo varias tareas que completen este estudio en varios ámbitos o solucionen problemas detectados. Por ejemplo, recientemente se ha publicado GPT-3 (Brown et al., 2020) que presenta mejores resultados que el estado del arte en varias tareas, entre ellas, el modelado lingüístico en el que se basa el funcionamiento de las sugerencias de código del presente trabajo.

Por otro lado, entrando más en el contexto del desarrollo de código existen otras aplicaciones que resultan de gran utilidad. Por ejemplo, aplicaciones comunes son la traducción de código entre lenguajes de programación (Chen et al., 2018) o desde pseudocódigo (Hasoon, 2014), detección y corrección de errores (Hajipour et al., 2019) (Pradel and Sen, 2018), etc. Estas aplicaciones demuestran que las redes neuronales son una buena aproximación, por lo que sería interesante observar la eficacia de las arquitecturas basadas en Transformer en tareas basadas en comprensión de código fuente en lugar de generación.

El problema de la pérdida de variabilidad en las sugerencias para un número de salidas grande muestra que las técnicas de generación actuales no son las mejores para la tarea de sugerencia de código. Por lo tanto, sería interesante estudiar otro método de generación que se adapte mejor a este tipo de tareas.

Por lo tanto, los resultados respaldan que las redes neuronales basadas en la arquitectura Transformer presentan una buena aproximación para la sugerencia de código fuente.

7. Referencias

Bhoopchand, A., Rocktäschel, T., Barr, E., and Riedel, S. (2016). Learning python code suggestion with a sparse pointer network. 11.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners.

Chen, X., Liu, C., and Song, D. (2018). Tree-to-tree neural networks for program translation.

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation.

Franks, C., Tu, Z., Devanbu, P. T., and Hellendoorn, V. (2015). Cacheca: A cache language model based code suggestion tool. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2:705–708.

Hajipour, H., Bhattacharya, A., and Fritz, M. (2019). Samplefix: Learning to correct programs by sampling diverse fixes.

Hasoon, S. (2014). Automatic pseudocode to source code translation using neural network technique. 05.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. (2019). The curious case of neural text degeneration.

Karampatsis, R.-M. and Sutton, C. (2019). Maybe deep neural networks are the best choice for modeling source code.

Kim, S., Zhao, J., Tian, Y., and Chandra, S. (2020). Code prediction by feeding trees to transformers.

Loshchilov, I. and Hutter, F. (2017). Decoupled weight decay regularization.

Pradel, M. and Sen, K. (2018). *DeepBugs: A Learning Approach to Name-Based Bug Detection*, volume 2. Association for Computing Machinery, New York, NY, USA, October.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019a). Language models are unsupervised multitask learners.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019b). Language models are unsupervised multitask learners.

Raychev, V., Vechev, M., and Yahav, E. (2014). *Code Completion with Statistical Language Models*, volume 49. Association for Computing Machinery, New York, NY, USA, June.

Sennrich, R., Haddow, B., and Birch, A. (2015). Neural machine translation of rare words with subword units.

Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L., and Zhang, L. (2019). Treegen: A tree-based transformer architecture for code generation.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks.

Séjourné, T., Feydy, J., Vialard, F.-X., Trouvé, A., and Peyré, G. (2019). Sinkhorn divergences for unbalanced optimal transport.

Tay, Y., Bahri, D., Yang, L., Metzler, D., and Juan, D.-C. (2020). Sparse sinkhorn attention.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need.

Ziegler, D. M., Stiennon, N., Wu, J., Brown, T. B., Radford, A., Amodei, D., Christiano, P., and Irving, G. (2019). Fine-tuning language models from human preferences.