Generación Automática de Código Fuente a través de Modelos Preentrenados de Lenguaje, un análisis de la literatura

Adrián Bender, Santiago Nicolet, Pablo Folino, Juan José Lopez y Gustavo Hansen

Facultad de Ingeniería, Universidad del Salvador, Buenos Aires, Argentina
{bender.adrian, santiago.nicolet}@usal.edu.ar

Abstract. Un Transformer es un modelo de Aprendizaje Profundo creado en 2017 con el objetivo de realizar traducciones entre lenguajes naturales. Las innovaciones que introdujo, particularmente la de auto-atención, han permitido construir prototipos que tienen una noción intuitiva del contexto, y comprenden el significado y los patrones subyacentes del lenguaje. En 2020 OpenAI hizo público GPT-3, un modelo preentrenado enfocado hacia la generación de lenguaje, que mostró resultados prometedores, creando textos con una calidad tal que se hace difícil distinguir si fueron escritos por un humano o por una máquina. Podemos afirmar que el código fuente es texto generado en un lenguaje formal, y por lo tanto podría ser generado con herramientas basadas en estos prototipos. Este trabajo presenta un estudio de la evolución y el estado del arte en este campo: la generación automática de código fuente a partir de especificaciones escritas en lenguaje natural. Recorremos diferentes casos, su éxito, las dificultades de encontrar mecanismos de evaluación y su posible implementación en un futuro por las empresas.

Keywords: Generación de Código, Transformers, Modelos Preentrenados, Automatización.

1 Introducción

En el último tiempo se ha comenzado a utilizar Aprendizaje Automático –entre otros tópicos de la Inteligencia Artificial– para automatizar tareas del área del desarrollo de software [1]. Así, por ejemplo, está surgiendo una nueva generación de herramientas impulsadas por IA, que guían y capacitan a los profesionales de software para generar mejores documentos de requisitos, escribir código más confiable y detectar automáticamente errores y vulnerabilidades de seguridad [2].

Muchas de ellas se enfocan en la generación automática de código, como por ejemplo: Kite, TabNine y GitHub Copilot. Son asistentes que permiten completar la escritura gracias al conocimiento aprendido sobre miles de millones de líneas de código público a través de una arquitectura de Aprendizaje Profundo (Deep Learning) llamada Transformers [3].

Si bien son herramientas que fueron lanzadas en forma reciente y que aún no se utilizan ampliamente, las grandes compañías del mercado de las TICs aseguran que gracias a los modelos subyacentes se podrá generar automáticamente código tan bueno como el que producen los humanos [4].

Motivado por el impacto que generaría en la industria del software el contar con herramientas que automaticen la escritura de código, el presente trabajo explica la temática de la generación automática de código fuente, brinda información de los Transformers, de cómo han evolucionado hacia los Modelos Preentrenados y ahonda en la adopción de estos prototipos para la generación de código, detallando los casos de éxito más representativos, destacando los últimos avances, y mencionando las oportunidades que se les presentan a las organizaciones más allá de las grandes compañías del sector.

Este documento se organiza de la siguiente manera: la sección 2 describe los Transformers, su origen y las innovaciones que introdujeron; en la sección 3 se muestra cómo evolucionaron hasta convertirse en Modelos Preentrenados; la sección 4 explica la tecnología utilizada en la generación de código fuente; la sección 5 detalla los hallazgos de la revisión de la literatura realizada y finalmente, en la sección 6, se listan nuestras conclusiones y algunas líneas futuras de investigación.

2 Transformers

Un Transformer es un modelo de Aprendizaje Profundo que viene causando un gran impacto en el campo del Procesamiento del Lenguaje Natural. Fue creado en 2017 con el objetivo de realizar traducciones entre lenguajes naturales (LN) [5].

Su arquitectura original era del tipo Codificador-Decodificador (Encoder-Decoder), la misma que la adoptada por los modelos de lenguaje mayormente utilizados hasta ese momento: las Redes Neuronales Recurrentes (RNN) [6]. Dichas redes capturan el orden de las palabras procesándolas una a la vez, en una secuencia. Esto les dificulta el manejo de textos de gran tamaño, como artículos largos o ensayos: cuando llegan al final de un párrafo extenso ya no tienen visibilidad sobre el comienzo del mismo. Además, las RNNs presentan una gran complejidad en su entrenamiento y, al procesar sus entradas secuencialmente, se dificulta la implementación de paralelismo en su aprendizaje [7].

Un hito en tareas de traducción fue la incorporación en 2015 del mecanismo de Atención [8], que permite que un modelo observe todas las palabras en la sentencia original y sepa a cuales “atender” –a través de un vector de similitudes aprendido durante su preentrenamiento– al momento de decidir cómo generar la traducción. Con este agregado se lograron resolver problemas importantes de la tarea como son los de respetar el género, los plurales y otras reglas gramaticales del lenguaje al que se traduce [9].

El mecanismo de Atención está presente en los Transformers así como otras dos innovaciones propias de estos modelos. Una es la Codificación Posicional, que consiste en agregar información del orden de las palabras a las entradas de la red. Con esta novedad se logró trasladar la carga de la comprensión del orden desde la estructura de la red hacia los datos, y esto trajo otras dos mejoras: la posibilidad de incorporar paralelismo al entrenamiento y la disminución significativa de su complejidad [10].

La segunda innovación, y posiblemente la más impactante, es la capa de Auto-Atención. Con ella, el Transformer aprende a ponderar la relación entre cada elemento de entrada y todos los ítems de la secuencia de salida en base a una función que incorpora

al resto de las palabras del texto de origen, y con esto se logra adquirir una noción intuitiva del contexto en el lenguaje [5].

Al permitir comprender una palabra en el contexto de las otras que la rodean, la Auto-Atención ayudó a las redes neuronales a eliminar la ambigüedad en las tareas de traducción y, a nivel más general, a construir modelos que comprenden el significado y los patrones subyacentes del lenguaje, y que actualmente son utilizados eficazmente en una amplia gama de tareas como por ejemplo: generar resúmenes de textos, entender y responder preguntas, etiquetar partes de discursos y extraer entidades [11].

3 Modelos Preentrenados de Lenguaje basados en Transformers

Desde su introducción los Transformers han ido evolucionando desde una arquitectura de modelo hacia un modelo entrenado en sí mismo. A la vez se han diversificado según su estructura y su forma de procesar la entrada para enfocarse en diferentes metas.

Así, por un lado están los enfocados en la generación del lenguaje, desarrollados para predecir la siguiente palabra en una secuencia, que mantienen solo el decodificador de la estructura original y, al procesar la entrada en una sola dirección solo conocen el texto leído hasta el momento. Por otro lado están los enfocados en el entendimiento, que son entrenados para predecir una palabra en cualquier orden de una secuencia, que mantienen solo la parte del codificador y leen la entrada en dos direcciones. Y finalmente hay Transformers que mantienen tanto el codificador como el decodificador, y los complementan para realizar tareas tanto de generación como de entendimiento [12].

En 2018, fue BERT (Bidirectional Encoder Representations from Transformers) [13] el que incorporó el concepto de entrenamiento bidireccional, lo que permitió generar prototipos con un conocimiento más rico del contexto y del flujo en el lenguaje. También innovó utilizando un corpus masivo de texto extraído mayormente de Wikipedia y Reddit para generar modelos preentrenados que pueden luego ser adaptados con datos específicos a diversos casos de uso.

Esta estrategia forma parte de lo que se conoce como Aprendizaje por Transferencia (Transfer Learning), cuyo objetivo es transferir el conocimiento aprendido de ciertos datos a otros más específicos de un dominio [14]. Así, podemos ver a un modelo de lenguaje preentrenado como un Modelo Base (Foundation Model) que está entrenado sobre un volumen masivo de datos sin etiquetar y que luego puede adaptarse a una amplia gama de tareas a través de Aprendizaje Supervisado sobre un conjunto acotado y específico de datos que recibe el nombre de Ajuste Fino (Fine-Tuning) [15].

A fines 2019 Google lanzó T5 (Text-to-Text-Transfer-Transformers Model) [16] y con él constató la eficacia del aprendizaje a través de modelos preentrenados sobre un gran volumen de datos (lo hizo sobre un corpus más grande que el de BERT) que luego son ajustados a objetivos específicos. Así, T5 puede ser adaptado con una gran calidad a tareas como por ejemplo: traducción entre distintos idiomas, generación de resúmenes de texto y desarrollo de bots conversacionales.

En 2020 OpenAI hizo público GPT-3 [17], la tercer versión de sus modelos GPT (Generative Pre-trained Transformer) enfocados hacia la generación de lenguaje, que mostró resultados prometedores generando textos con una calidad tal que se hace difícil distinguir si fueron escritos por un humano o por una máquina.

Una de las claves de dichos resultados fue crear un nuevo conjunto de datos, de mejor calidad y con mayor diversidad en el estilo de escritura. Otro aspecto que podría explicar su éxito es la gran cantidad de parámetros del modelo que van hasta 175 mil millones, siendo que la versión anterior, GPT-2 [18], tenía 1,5 millones de parámetros.

Pero quizás el aspecto más innovador es que se define como un Modelo de Uso General (Task-Agnostic) [19], y que por ende puede ser utilizado con un mínimo Ajuste Fino para diferentes fines, como por ejemplo para la modalidad de preguntas y respuestas, para la redacción de textos extensos, o para la traducción de lenguaje natural [20].

Un enfoque utilizado para la construcción de Modelos de Uso General es el de Meta-Aprendizaje [21], que combina, durante su entrenamiento, ciclos de Aprendizaje Supervisado para aprender cada tarea con fases de Aprendizaje No Supervisado para detectar qué se pretende realizar. Dado que este tipo de aprendizaje implica absorber conocimiento dentro de los parámetros del modelo, resulta lógico adjudicar gran parte de los resultados logrados por GPT-3 al crecimiento sustancial de su capacidad.

Este modelo fue validado en tres escenarios que se diferencian entre sí por la cantidad de ejemplos –cero, uno y pocos– que se le proporcionan, junto a una descripción en lenguaje natural de la tarea, en el momento de la inferencia. La alta calidad lograda con pocos ejemplos (Few-Shots) permitió concluir que el Fine-Tuning no es necesario en un gran número de tareas [17].

Esto profundiza el interés por el desarrollo de Modelos de Lenguaje de Uso General, que resultan beneficiosos porque evitan la necesidad de contar con una gran cantidad de ejemplos para cada nueva tarea que se pretenda realizar, y sumamente prometedores por el universo de aplicaciones que pueden generar.

4 Generación Automática de Código Fuente

Un sistema de generación automática de software traduce una especificación escrita en lenguaje natural a un programa ejecutable por medio de la definición de su código fuente en un lenguaje de programación (LP) dado.

Un lenguaje de programación es un lenguaje formal, y todo código fuente debe seguir las reglas definidas por su gramática. Así, uno de los desafíos de la automatización es que la salida de un sistema de generación de código debe adaptarse a reglas muy específicas para ser sintácticamente correcto [22].

Además, al tratarse de una pieza ejecutable, sufre de cierta sensibilidad semántica, ya que pequeños cambios en el código pueden modificar drásticamente su significado. En este sentido el lenguaje natural es más robusto porque un texto puede llegar a ser entendido incluso conteniendo errores sintácticos o semánticos [23].

El código fuente, al estar escrito en un lenguaje formal, es visto generalmente como un modelo matemático y suele ser evaluado por el rigor y la claridad de sus definiciones, las abstracciones presentes y las pruebas formales de sus propiedades [24].

Es por eso que la investigación en esta área ha estado dominada por un enfoque formal o lógico-deductivo: se sostenía que, dado que el software se construye sobre un lenguaje formal, las herramientas que busquen automatizar su generación debían concebirse en términos puramente formales. Así, los primeros acercamientos se basaban principalmente en plantillas que permitían mapear sentencias en lenguaje natural a una

representación formal de su significado, generalmente a través de la construcción de su árbol de sintaxis (AST: Abstract Syntax Tree) [25].

Un enfoque más moderno es la Síntesis de Programas (Program Synthesis), que consiste en utilizar técnicas basadas en búsqueda para generar un código que cumpla con una especificación dada. Si bien existen casos de éxito en una gran variedad de dominios como por ejemplo: la manipulación de datos en Excel [26], y la síntesis de expresiones regulares [27], los mismos están acotados a un lenguaje específico de dominio (DSL: Domain Specific Language). Esta limitación impide poder escalar este tipo de abordaje a LPs de propósito general que, en comparación con los DSL, incluyen muchas más funciones de lenguaje y reglas de sintaxis y, por lo tanto, constituyen un espacio de búsqueda mucho más grande para generar un programa.

En los últimos años muchos sistemas ampliamente usados y exitosos de código abierto han ido publicando en la web su código fuente como así también los metadatos relacionados con su autoría, la corrección de errores y los procesos de revisión formando así un cuerpo masivo y creciente de código. De igual manera, muchos sitios web de código abierto (w3school, GitHub, StackOverflow, etc.) proporcionan bibliotecas, API, ejemplos de uso y solución de errores generando una base de datos en la que los desarrolladores de software confían en gran medida para sugerencias de código fuente, completado de código y corrección de errores [25].

La existencia y disponibilidad de este corpus masivo de código dio lugar a un nuevo enfoque, basado en datos, para la generación automática de código que apuesta a que los modelos de Aprendizaje Automático aplicados sobre millones de líneas de código bien escritas puedan descubrir patrones que caractericen parcialmente al software que es confiable, fácil de leer y de mantener, y utilizar dicho conocimiento para, entre otras cosas, generar código en forma automática [28].

La hipótesis de naturalidad del código es una de las razones que facilitó la aplicación de técnicas de Procesamiento del Lenguaje Natural [29]. Dicha teoría sostiene que, debido a que la codificación es un acto de comunicación, uno podría esperar que un corpus masivo de código tenga patrones ricos, como ocurre en el LN, que puedan ser aprendidos por modelos de lenguaje. La naturalidad del código parece tener una fuerte relación con el hecho de que los desarrolladores prefieren escribir código claro y fácilmente legible, lo que les simplifica la comprensión y el mantenimiento del software [30].

Así se han logrado desarrollar modelos probabilísticos de lenguaje, principalmente modelos de N-gramas, que son capaces de predecir el siguiente token de código en la secuencia dados los que lo preceden [31]. Los modelos N-gramas asumen que los tokens se generan secuencialmente, de izquierda a derecha, y que el siguiente token se puede predecir utilizando solo los $n-1$ tokens anteriores [32].

Estos métodos estadísticos permiten que un sistema genere una hipótesis, junto con sus valores de confianza asociados, de lo que un desarrollador quiere hacer a continuación, y demostraron ser una técnica efectiva y práctica para aprender dependencias simples y cercanas en secuencias [33].

Sin embargo, comparado con el texto en LN, el código suele tener dependencias contextuales, sintácticas y estructurales más estrictas. Tratar el código fuente como un texto puede no ser efectivo para capturar estas dependencias más complejas [34].

Además el modelo N-gramas, como el resto de los modelos probabilísticos de lenguaje, trabaja sobre secuencias de tamaño limitado, y esa cota le impide manejar dependencias de largo alcance. Esto constituye un importante problema en la generación

automática de software, en donde un elemento depende frecuentemente de otro lejano en el código, por ejemplo una referencia de variable depende de su definición que puede aparecer muchas líneas antes. Por otro lado, predecir una secuencia grande en un solo paso resulta inviable debido al número exponencial de alternativas posibles [28].

Las limitaciones de los modelos estocásticos y el gran desarrollo de las redes neuronales han dado lugar a avances significativos en la generación automática de código a través de modelos construidos mediante algoritmos de Aprendizaje Profundo. La mayoría de dichas arquitecturas neuronales generan código fuente definiendo la secuencia de tokens que lo componen, y por ende son denominados modelos de secuencia a secuencia (Seq2Seq), pero también hay modelos de secuencia a árbol (Seq2Tree), y modelos de secuencia a grafos semánticos (Seq2Graph) [35-37].

Conociendo los resultados obtenidos en el procesamiento del lenguaje natural en los últimos años y, entendiendo que por su capacidad podrían abordar las limitaciones de los modelos probabilísticos y mejorar lo logrado por los modelos neuronales, resulta interesante conocer el nivel de efectividad de los modelos preentrenados de lenguaje basados en Transformers en la generación automática de código.

5 Generación Automática de Código Fuente a través de Modelos Preentrenados de Lenguaje basados en Transformers

Realizamos una revisión bibliográfica sobre documentos publicados, libros y referencias en foros acerca de la generación automática de código a través de modelos preentrenados de lenguaje hasta Abril de 2022 privilegiando trabajos en base a los siguientes criterios: (1) que se enfoquen en generar código fuente –funciones, módulos o programas– a partir de una especificación en LN, descartando aquellos destinados al autocompletado; (2) que demuestren haber generado código legible o compilable, (3) que hayan brindado detalles acerca del modelo y del entrenamiento, y (4) que hayan reportado los resultados de las evaluaciones de su uso.

Hemos encontrado varios casos de éxito. Algunos mediante prototipos entrenados desde cero para la generación automática de código, mientras que otros por medio de extensiones o ajustes realizados a los modelos de lenguaje existentes para dicha tarea.

Varios modelos son de código abierto y han detallado el corpus de datos sobre el cual se han entrenado y sus parámetros, pero otros no están disponibles públicamente, no permiten su acceso gratuito, ni hacen públicos sus parámetros ni sus fuentes de entrenamiento, lo que dificulta la investigación y el relevamiento en un área que ya de por sí está bastante limitada debido a la cantidad de recursos que se deben poseer para entrenar arquitecturas tan grandes sobre volúmenes tan amplios de datos.

La comparación entre modelos no resulta fácil puesto que tanto en los trabajos en que fueron publicados, así como en aquellos en que fueron evaluados, se utilizaron métricas distintas. Los primeros documentos midieron la generación de código utilizando indicadores que tradicionalmente eran aplicados a los modelos N-gramas para ponderar tareas de lenguajes como por ejemplo la traducción. Estas métricas apuntan a medir el grado de similitud entre el texto generado y uno que se tiene como referencia, y entre ellas se encuentran: BLEU (Bilingual Evaluation Understudy) [38] y ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [39].

Tabla 1. Modelos preentrenados de lenguaje evaluados para la generación de código. En las columnas se lista para cada uno: el año de desarrollo, la cantidad de parámetros, el nombre, si es código abierto, si incluyó código dentro del corpus de entrenamiento y si permite ajuste fino.

Año	Parámetros (en millones)	Nombre	Código Abierto	Código en Entrenamiento	Permite Ajuste Fino
2019	1.500	GPT-2	No	No	Sí
2020	175.000	GPT-3	No	Sí	No
2021	2.700	GPT-Neo	Sí	Sí	Sí
2022	6.000	GPT-J	Sí	Sí	Sí
2021	20.000	GPT-NeoX	Sí	Sí	Sí

En 2020 Ren y otros [40] demostraron que BLEU tiene problemas para capturar características semánticas del código y crearon CodeBLEU, una métrica para medir la calidad del código sintetizado ampliando BLEU para considerar también la corrección sintáctica y semántica basándose en el AST y en la estructura de flujo de datos.

Pero las métricas basadas en coincidencias no pueden dar cuenta del espacio grande y complejo de programas funcionalmente equivalentes a una solución de referencia. Como consecuencia, los trabajos más recientes se han centrado en la corrección funcional (Functional Correctness), donde una muestra de código es considerada correcta si pasa un conjunto de pruebas unitarias.

A continuación detallamos los resultados de la recopilación de casos de éxitos más representativos. En la Tabla 1 listamos los modelos preentrenados de lenguaje que han sido utilizados –luego de extensiones o ajustes– para la generación de código, y en la Tabla 2 aquellos preentrenados específicamente para generación de código.

Los primeros modelos preentrenados basados en Transformers desarrollados específicamente para tareas relacionadas al Software surgieron en 2020 y entre ellos los más destacados fueron CodeBERT [41] y PyMT5 [42]. Los mismos se centraron principalmente en tareas de recuperación, clasificación y reparación de programas.

CodeBERT [41] se inspiró en BERT, es bimodal –LN a LP y LP a LN–, y fue el primer modelo preentrenado con código de múltiples lenguajes de programación: millones de pares de código-descripción de funciones de seis LP distintos extraídos de repositorios públicos de GitHub.

El modelo logró capturar la conexión semántica existente entre el lenguaje natural y los lenguajes de programación, y producir representaciones de propósito general capaces de dar soporte, luego de un proceso de ajuste fino, a tareas específicas vinculadas a al lenguaje. Así, por ejemplo, CodeBERT obtuvo sólidos resultados en la búsqueda de código, mostrando que los Transformers permiten optimizar la recuperación empleada en la síntesis de programas, hipótesis que ya había sido comprobada en trabajos anteriores a través de la utilización de RNNs [43].

Este trabajo planteó la necesidad de incorporar información de los ASTs en el paso previo al entrenamiento para capturar la fuerte sintaxis subyacente de los LPs, algo que ya había sido formulado en desarrollos previos a los modelos preentrenados [43,44].

Tabla 2. Modelos de lenguaje preentrenados específicamente para generación de código. En las columnas se lista para cada uno: el año de desarrollo, la cantidad de parámetros (en millones), el nombre y si es no código es abierto, la arquitectura, los lenguajes de programación en su entrenamiento y la forma de evaluación utilizada para publicar sus resultados.

Año	Parámetros (en millones)	Nombre	Inspirado en / Arquitectura	Lenguajes de Programación Usados en Entrenamiento	Métrica Evaluación
2020	125	CodeBERT	BERT (Codificador)	Go - Java - Javascript - PHP - Python - Ruby	BLEU
2020	374	PyMT5	T5 (Cod./Decod.)	Python	BLEU, ROUGE
2021	229	CodeT5	T5 (Cod./Decod.)	Go - Java - Javascript - PHP - Python - Ruby - C - C++	EM, BLEU y CodeBLEU
2021	406	PLBART	BART (Cod./Decod.)	Java - Python	EM, BLEU y CodeBLEU
2021	124	CodeGPT	GPT (Decodificador)	Java - Python	EM, BLEU y CodeBLEU
2021	Hasta 12.000	Codex (No es código abierto)	GPT (Decodificador)	Python	Corrección Funcional
2021	Hasta 12.000	Codex-S (No es código abierto)	GPT (Decodificador)	Python	Corrección Funcional
2021	Hasta 137.000	Austin-Odena (No es código abierto)	GPT (Decodificador)	No informado	Corrección Funcional
2022	1.500	CodeParrot	GPT (Decodificador)	Python	Corrección Funcional
2022	2.700	Polycoder	GPT (Decodificador)	C - C# - C++ - Go - Java - Javscript - PHP - Python - Ruby - Rust - Scala - Typescript	Corrección Funcional

PyMT5 [42] se planteó objetivos similares a CodeBERT pero se inspiró en T5 como arquitectura para entrenar un sistema multimodal que logre traducir entre subconjuntos no superpuestos de firma-definición-descripción de funciones. Fue entrenado sobre código Python de repositorios GitHub a los cuales se les agregó información del AST de dichos códigos, lo que permitió que más del 90% de los métodos generados sean sintácticamente correctos.

CodeT5 [46] utilizó un preentrenamiento multitarea sobre un corpus con códigos públicos de ocho LPs demostrando que la arquitectura T5 beneficia tanto a las tareas de comprensión como a las de generación de lenguaje. Planteó un objetivo novedoso: agregar información de los tipos de token y, haciendo foco en los identificadores, permitió incorporar al modelo la semántica transmitida por los desarrolladores al asignar un nombre a los mismos. Pudo aprender mejor la semántica del código logrando mejoras significativas en la mayoría de las tareas de CodeXGLUE [47], un punto de referencia usado para modelos de Aprendizaje Automático destinados a generación y entendimiento de código.

PLBART [48] (Program and Language BART) utilizó el modelo de lenguaje BART –del tipo Codificador-Decodificador– con un entrenamiento previo sobre una amplia colección de pares código-descripción de funciones de Java y Python obtenidas de GitHub y StackOverflow. La eficacia de PLBART quedó demostrada a través de su éxito en pruebas realizadas en la reparación de programas y la detección de clones y software vulnerable. En cuanto a la generación, logró retornar código sintáctica y semánticamente válido.

CodeGPT [47] es un modelo preentrenado construido desde cero con la arquitectura GPT-2 sobre el dataset CodeSearchNet [49] formado por pares código-descripción de más de 2,5 millones de funciones de Python y Java. Además, utilizaron el mismo conjunto de datos para extender el preentrenamiento de GPT-2 generando otro prototipo al que llamaron CodeGPT-Adaptado. Hicieron una prueba de rendimiento (benchmark) y, utilizando distintas métricas de calidad, comprobaron que el código generado por CodeGPT-Adaptado era superior al de CodeGPT, y que a su vez ambos superaban al generado por el modelo GPT-2 original al que habían realizado previamente un ajuste fino con código GitHub.

GPT-Neo [50], GPT-J [51] y GPT-NeoX [52] son implementaciones de código abierto similares a GPT-3 preentrenados en The Pile [53], un corpus formado por una amplia variedad de textos, incluidos artículos de noticias, foros, etcétera, y sólo una modesta parte –menor al 10%– de código extraído de GitHub. A pesar de ello, estos modelos de lenguaje pueden ser usados para generar código fuente con una calidad razonable, incluso en algunos trabajos se muestra que superan a los sistemas GPT existentes en evaluaciones cualitativas de programación [54].

En 2021 se publicó APPS [55], un marco de referencia para evaluar la capacidad de los modelos preentrenados para generar código Python a partir de una especificación en LN. Incluye 10 mil problemas de programación de distintos niveles de dificultad cuyo código no necesariamente se corresponde con la definición de una única función, sino con la escritura de un programa que lee una entrada, la procesa y escribe una salida.

En dicho benchmark evaluaron los modelos de lenguaje GPT existentes. Se hizo un ajuste fino previo con código GitHub sobre los prototipos que lo permiten –GPT-2 y GPT-Neo– mientras que el restante –GPT-3– fue evaluado en su versión de pocos ejemplos. Para medir los resultados decidieron utilizar casos de prueba y captura de errores, a diferencia de la mayoría de los trabajos previos donde se utilizaba BLEU.

Descubrieron que el porcentaje de problemas resueltos correctamente a través del código generado disminuye con el nivel de dificultad de los mismos, y mejora a través de ajustes finos y el aumento en el tamaño de los prototipos. Respecto al incremento de los parámetros, también se comprobó que influye notablemente en la disminución del

número de errores sintácticos generados. Aún así, los resultados no fueron muy alentadores, puesto que el modelo que mejores resultados logró –GPT-Neo– solo pudo pasar aproximadamente el 20% de los casos de prueba de los problemas más sencillos. Este nivel tan bajo probablemente haya estado relacionado a que los problemas de competencias de programación –utilizados en el dataset– están generalmente escritos con un estilo que no ayuda a los algoritmos subyacentes para resolverlos.

OpenAI desarrolló Codex [56], un prototipo que posee hasta 12.000 millones de parámetros, ajustando GTP-3 a través de un entrenamiento sobre el código de 54 millones de repositorios GitHub con el objetivo de evaluar la capacidad de los grandes modelos de lenguaje para producir código funcionalmente correcto a partir de una descripción en lenguaje natural.

Para los testeos desarrollaron HumanEval, un conjunto de 164 problemas de programación escritos a mano y destinados a evaluar la comprensión del lenguaje, el razonamiento, la resolución algorítmica y la aplicación de matemáticas básicas. Cada ejemplo del dataset incluye la firma y el cuerpo de una función en Python, la descripción en lenguaje natural, y varias pruebas unitarias.

Evaluaron distintos tamaños del modelo y los compararon con lo que puede realizar GPT-3 sin ajustes. Para medir los resultados obtuvieron múltiples muestras de código generado por los prototipos evaluados, y verificaron cuántos de ellos pasaban las pruebas unitarias. Encontraron que con una sola muestra, Codex con 12.000 millones de parámetros resolvió casi el 29% de los problemas, con 300 millones resolvió el 13%, y GPT sin ajustar el 0%.

Para mejorar el desempeño del modelo ajustaron Codex con un dataset adicional de funciones independientes correctamente implementadas. Llamaron Codex-S al prototipo resultante, y comprobaron que resolvía casi el 38% de los problemas con una sola muestra y que además fue capaz de generar dentro de 100 muestras al menos una función correcta para el 77,5 % de los problemas.

Comprobaron que los resultados de Codex son superiores a los de TabNine [57], que junto a Kite [58] son dos de las herramientas más populares del mercado para autocompletado de código, ambas basadas en GPT-2.

Codex fue utilizado para impulsar GitHub Copilot [59], un asistente de desarrollo que es capaz de generar código en forma automática basándose en el contexto del usuario. Su lanzamiento al mercado fue muy comentado, aunque uno de los primeros análisis de usabilidad evidenció que, si bien ayuda a los programadores proporcionando un punto de partida útil para comenzar las tareas y ahorrando esfuerzo de búsqueda en línea, no necesariamente mejora el tiempo de finalización del trabajo ni su tasa de éxito. Además, algunos encuestados manifestaron que muchas veces les resultó difícil comprender, editar y depurar fragmentos de código generados por Copilot, y que esto disminuyó su efectividad en la resolución de tareas [60].

Austin y Odena [61] desarrollaron dentro de Google Research un modelo de lenguaje basado en Transformers del tipo solo decodificador preentrenado sobre un amplio corpus conformado por documentos de la web, textos, códigos fuente y Wikipedia. Realizaron un benchmark con el objetivo de medir la capacidad de su prototipo para generar pequeños programas en Python a partir de descripciones en lenguaje natural.

Lo hicieron sobre dos datasets: uno con 974 tareas de un nivel de dificultad básico y otro con 23.914 problemas con descripciones más complejas. Cada ejemplo de los dos conjuntos incluyó la documentación del problema, el código de una función Python que

lo resuelve y algunos casos de prueba para verificar la funcionalidad. Experimentaron con distintas cantidades de parámetros –entre 244 millones y 137 mil millones– y midieron la corrección funcional de los códigos generados por los distintos modelos, con ajuste fino y sin él en la versión de pocos ejemplos.

Constataron que se logra mayor porcentaje de resolución a medida que aumenta el número de parámetros y se diseñan mejor los enunciados. En el dataset de nivel básico, el modelo más grande logró resolver correctamente casi el 60% de las tareas sin ajuste fino, y a través de fine-tuning el porcentaje mejoró en promedio 10 puntos porcentuales en la mayoría de los tamaños evaluados. En el conjunto de datos de tareas más complejas, el prototipo ajustado más grande logró resolver el 84% de los problemas correctamente.

Un análisis más profundo reveló que la tasa de resolución es menor cuando los problemas plantean múltiples restricciones o están compuestos por varios subproblemas. Así mismo, explorando la base semántica de lo construido, ajustaron los modelos para predecir los resultados de la ejecución del código generado por ellos mismos y encontraron que no logran predecir la salida a partir de una entrada específica. Esto sugiere una gran brecha entre lo que hacen estos prototipos y lo que alcanzarían a comprender de la tarea que resuelven.

Otro aspecto interesante del trabajo es un estudio realizado acerca de la capacidad de los modelos para mejorar sus resultados incorporando comentarios recibidos a partir de un intercambio con humanos acerca del código. Descubrieron que la retroalimentación en lenguaje natural permite reducir considerablemente la tasa de error existente en comparación con el código generado inicialmente por el prototipo.

A fines de 2021 Tunstal y otros desarrollaron CodeParrot [62], entrenando GPT-2 desde cero con todo el código Python público disponible en GitHub. Uno de los objetivos planteados fue el de desarrollar un modelo preentrenado para la generación de software de código abierto que logre resultados similares a los de los prototipos más fuertes, cuyo código no está publicado. Realizaron un benchmark utilizando el dataset HumanEval y la misma métrica de evaluación funcional utilizada por Codex para poder contrastar los resultados, y encontraron una diferencia significativa en favor de este último en todos los tamaños del modelo.

Polycoder [63] se publicó en 2022 con un objetivo similar al de CodeParrot. Pero en este caso se buscaba además generar un prototipo preentrenado con código de múltiples lenguajes de programación, buscando la mejor generalización que generalmente proporcionan los modelos multilingües. Utilizaron también la arquitectura GPT-2 para entrenar al modelo desde cero sobre un corpus de código de 12 lenguajes de programación extraído de GitHub.

Experimentaron con distintas cantidades de parámetros –hasta 2700 millones– y compararon la corrección funcional sobre el dataset HumanEval contra la de varios modelos de lenguaje: CodeParrot, GPT-Neo, GPT-J y Codex. Comprobaron que ninguno alcanza el nivel de los indicadores obtenidos por Codex, y que el rendimiento de PolyCoder –al igual que el del resto– aumenta con el incremento del tamaño y del tiempo de entrenamiento. Pudieron observar que los resultados de GPT-Neo son mejores que los de PolyCoder en algunos lenguajes, lo que sugiere que la capacitación tanto en código como en texto en lenguaje natural puede beneficiar al modelado de código.

6 Conclusiones y Futuras Líneas de Investigación

Los modelos basados en Transformers están demostrando ser, por el impacto de sus resultados, la tecnología adecuada para procesar largas secuencias en PLN. La hipótesis de naturalidad del código fuente y las dependencias de largo alcance que lo caracterizan propician la investigación en la generación de software a partir de una especificación escrita en lenguaje natural a través de estos prototipos.

Los trabajos actuales en el área –los modelos preentrenados desde cero específicamente para tareas de software y las extensiones y/o los ajustes realizados a los modelos de lenguaje existentes– logran resultados razonables, a pesar de que la generación de código supone un problema más difícil debido, entre otras cosas, a que el mismo suele tener dependencias contextuales, sintácticas y estructurales más estrictas que el texto.

Además se observa una constante y notoria evolución a lo largo de los últimos tres años. Diversos trabajos comprueban que el incremento en la cantidad de parámetros y, la adecuada selección, el curado y la ampliación de los corpus de entrenamiento son algunas de las principales causas de mejora significativa en los resultados.

El progreso en el área también se hace explícito a través de los cambios producidos en la medición de los resultados. Con la publicación de Codex se estipuló que el modelo genere varias muestras de código, y se tome la que mejor funciona –o una de las que funciona– a partir de pruebas unitarias para la evaluación. Todos los trabajos posteriores relevados utilizaron la misma métrica, la que podría convertirse en un estándar, facilitando la comparación entre modelos.

La gran mayoría de los trabajos se planteó como objetivo el escribir un código mediante la definición de una única función, logrando resultados correctos. Algunos otros desarrollos ampliaron el alcance, y evidenciaron inconvenientes en la generación cuando la resolución del problema implicaba un código con más de una función o varias subtareas. Probablemente la naturaleza compositiva del código y las abstracciones en capas del software planteen un desafío aún no resuelto en este objetivo de interconexión entre texto y código.

Así mismo, estudios de usabilidad de GitHub Copilot indicaron ciertas limitaciones de esta herramienta en la ayuda que puede proveer a los programadores, mostrando cierta falta de legibilidad del código generado automáticamente por estas tecnologías.

A las dificultades mencionadas, y en base a las experiencias documentadas de los experimentos de los distintos modelos, nos parece relevante agregar que este código podría mostrar falta de características que faciliten las tareas de testeo y de mantenimiento.

La depuración y la mejora de un software conllevan mucho tiempo. Son dos de las razones por la cual resulta importante mantener baja la complejidad, y una manera de lograrlo es a partir de partes simples conectadas por interfaces bien definidas, de modo que la mayoría de los problemas sean locales. Los modelos no habrían logrado incorporar este concepto.

Cada una de las limitaciones mencionadas constituyen, a nuestro parecer, un tema abierto que debe tenerse en cuenta en el desarrollo de este tipo de sistemas.

Además, considerando que los modelos multilenguaje empleados en tareas de lenguaje natural han demostrado mayor capacidad de generalización, y que en generación de código fuente la gran mayoría de trabajos utilizó fuentes en Python, sería interesante ver lo que pueden lograr entrenándose con código de más lenguajes.

Finalmente los modelos que mejor efectividad logran –Codex y Austin-Odena– no están disponibles para su uso gratuito, lo que limita los avances a lo que puedan lograr muy escasas compañías grandes de tecnología.

Referencias

1. Chandler, S.: How AI Is Making Software Development Easier For Companies And Coders. Recuperado: 2022-02-20. https://www.forbes.com/sites/simonchandler/2020/02/05/how-ai-is-making-software-development-easier-for-companies-and-coders.
2. Schatsky, D., Bumb, S.: AI is helping to make better software. Recuperado: 2022-02-20. https://www2.deloitte.com/us/en/insights/focus/signals-for-strategists/ai-assisted-software-development.html.
3. Singh, T.: Software Ate The World, Now AI Is Eating Software. Recuperado: 2022-02-20. https://www.forbes.com/sites/cognitiveworld/2019/08/29/software-ate-the-world-now-ai-is-eating-software.
4. Vincent, J.: DeepMind says its new AI coding engine is as good as an average human programmer. Recuperado: 2022-02-24. https://www.theverge.com/2022/2/2/22914085/alphacode-ai-coding-program-automatic-deepmind-codeforce.
5. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, L., Polosukhin, I.: Attention is all you need, en Advances in neural information processing systems (2017).
6. Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., Wu, Y.: Exploring the Limits of Language Modeling, arXiv: 1602.02410v2 (2016).
7. Wu, Y., Schuster, M., Chen, Z., Le, Q., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, U., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Dean, J.: Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, arXiv: 1609.08144 (2016).
8. Bahdanau, D., Cho, K., Bengio, Y.: Neural Machine Translation by Jointly Learning to Align and Translate, en Third International Conference on Learning Representations (2015).
9. Luong, M., Pham, H., Manning, C.: Effective Approaches to Attention-based Neural Machine Translation, doi: 10.18653/v1/D15-1166 (2015).
10. Luo, S., Li, S., Cai, T., He, Di., Peng, D., Zheng, S., Ke, G., Wang, L., Liu, T.: Stable, fast and accurate: kernelized attention with relative positional encoding, en Advances in Neural Information Processing Systems (2021).
11. Soydaner, D.: Attention mechanism in neural networks: where it comes and where it goes. Neural Computing and Applications. doi: 10.1007/s00521-022-07366-3 (2022).
12. Dong, L., Yang, N., Wang, W., Wei, F., Liu, X., Wang, Y., Gao, J., Zhou, M., Hon, H.: Unified language model pre-training for natural language understanding and generation, arXiv: 1905.03197 (2019).
13. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: Pretraining of deep bidirectional transformers for language understanding, arXiv: 1810.04805 (2018).
14. McCann, B., Bradbury, J., Xiong, C., Socher, R.: Learned in Translation: Contextualized Word Vectors, arXiv: 1708.00107 (2017).
15. Howard, J. Ruder, S.: Universal language model fine-tuning for text classification, arXiv: 1801.06146 (2018).
16. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.: Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, arXiv: 1910.10683 (2019).
17. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krüger, B., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin,

Mateusz, Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners, arXiv: 2005.14165 (2020).
18. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners, OpenAI blog (2019).
19. Finn, C., Abbeel, P., Levine, S.: Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks, arXiv: 1703.03400 (2017).
20. Reed, S., Chen, Y., Paine, T., Oord, A., Eslami, S., Rezende, D., Vinyals, O., Freitas, N.: Few-shot Autoregressive Density Estimation: Towards Learning to Learn Distributions, arXiv: 1710.10304 (2017).
21. Chen, Y., Hoffman, M., Gomez Colmenarejo, S., Denil, M., Lillicrap, T., de Freitas, N.: Learning to learn for global optimization of black box functions, en ICML (2017).
22. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: using static analysis to find bugs in the real world, Commun. ACM (2010).
23. Gabel, M., Su, Z.: A study of the uniqueness of source code, en Proceedings of the International Symposium on Foundations of Software Engineering (2010).
24. Hunt, A. Thomas, D.: The pragmatic programmer: from journeyman to master, Addison-Wesley Professional (2000).
25. Allamanism M., Barr, E., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness, arXiv: 1709.06182 (2017).
26. Gulwani, S., Marron, M.: NLyze: Interactive programming by natural language for spreadsheet data analysis and manipulation, in Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (2014).
27. Kushman, N., Barzilay, R.: Using Semantic Unification to Generate Regular Expressions from Natural Language, en Proceedings of Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (2013).
28. Bhoopchand, A., Rocktäschel, T., Barr, E., Riedel, S.: Learning Python Code Suggestion with a Sparse Pointer Network, arXiv: 1611.08307 (2016).
29. Hindle, A., Barr, E., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software, Commun. ACM (2016).
30. Hindle, A., Barr, E., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software, en 2012 34th International Conference on Software Engineering (ICSE), páginas 837–847 (2012).
31. Nguyen, T., Nguyen, A., Nguyen, H,, Nguyen, T.: A statistical semantic language model for source code, en ESEC/FSE, páginas 532–542, ACM (2013).
32. Ranjan, N., Mundada, K., Phaltane, K., Ahmad, S.: A survey on techniques in NLP, International Journal of Computer Applications (2016).
33. Cambria, E., White, B.: Jumping NLP curves: A review of natural language processing research, IEEE Computational intelligence magazine (2014).
34. Hussain, Y., Huang, Z., Wang, S., Zhou, Y.: Codegru: Context-aware deep learning with gated recurrent unit for source code modeling, arXiv: 1903.00884 (2019).
35. Sutskever, I., Vinyals, O., Le, Q.: Sequence to sequence learning with neural networks, en NIPS, pág. 3104–3112 (2014).
36. Rabinovich, M., Stern, M., Klein, D.: Abstract Syntax Networks for Code Generation and Semantic Parsing, en ACL, páginas 1139–1149 (2017).
37. Sun, Z., Zhu, Q., Mou, L., Xiong, Y., Li, G., Zhang, L.: A grammar-based structural CNN decoder for code generation, en AAAI, volume 33, páginas 7055–7062 (2019).
38. Papineni, K., Roukos, S., Ward, T., Zhu, W.: BLEU: a method for automatic evaluation of machine translation, en Proceedings of the Annual Meeting of the Association for Computational Linguistics (2002)
39. Lin, C..: ROUGE: A Package for Automatic Evaluation of Summaries, en MarieFrancine Moens SS (ed) Text Summarization Branches Out: Proceedings of the Association for Computational Linguistics, páginas 74–81 (2004).

40. Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S.: Codebleu: a method for automatic evaluation of code synthesis, ar-xuv: 2009.10297 (2020).
41. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: Codebert: A pre-trained model for programming and natural languages, arXiv: 2002.08155 (2020).
42. Clement, C., Drain, D., Timcheck, J., Svyatkovskiy, A., Sundaresan, N.:Pymt5: multi-mode translation of natural language and python code with transformers, arXiv: 2010.03150 (2020).
43. Balog, M., Gaunt, A., Brockschmidt, M., Nowozin, S., Tarlow, D.: DeepCoder: Learning to Write Programs, en Proceedings of the ICLR (2017).
44. Yin, P., Neubig, G.: TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation, en EMNLP (2018).
45. Sun, Z., Zhu, Q., Xiong, S., Sun, Y., Mou, L., Zhang, L.: TreeGen: A Tree-Based Transformer Architecture for Code Generation, arXiv: 1911.09983 (2019).
46. Wang, Y., Wang, W., Joty, S., Hoi, S.: Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, arXiv: 2109.00859 (2021).
47. Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Liu, S.: CodeXGLUE: A machine learning benchmark dataset for code understanding and generation, en Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (2021).
48. Ahmad, W., Chakraborty, S., Ray, B., Chang, K.: Unified pre-training for program understanding and generation, en Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, páginas 2655–2668 (2021).
49. Husain, H., Wu, H., Gazit, T., Allamanis, M., Brockschmidt, M.: Codesearchnet challenge: Evaluating the state of semantic code search, arXiv: 1909.09436 (2019).
50. Black, S., Gao, L., Wang, P., Leahy, C., Biderman, S.: GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, URL https://doi.or g/10.5281/zenodo.5297715 (2021).
51. Wang, B., Komatsuzaki, A.: GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model https://github.com/kingoflolz/mesh-transformer-jax (2021).
52. Black, S., Biderman, S., Hallahan, E., Anthony, Q., Gao, L., Golding, L., He, H., Leahy, C., McDonell, K., Phang, J., Pieler, M., Prashanth, S., Purohit, S., Reynolds, L., Tow, J., Wang, B., Weinbach, S.. GPT-NeoX-20B: An open-source autoregressive language model (2022).
53. Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., Presser, S., Leahy, C.: The Pile: An 800gb dataset of diverse text for language modeling, arXiv: 2101.00027 (2020).
54. Woolf, M.: Fun and dystopia with ai-based code generation using gpt-j-6b, URL https://minimaxir.com/ 2021/06/gpt-j-6b/ (2021).
55. Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., Steinhardt, J.: Measuring coding challenge competence with apps, arXiv: 2105.09938 (2021).
56. Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. arXiv: 2107.03374 (2021).
57. Tabnine. https://www.tabnine. com/. Recuperado: 2022-04-30.
58. Kite. https://www.kite.com/. Recuperado: 2022-04-30.
59. GitHub Copilot. https://copilot.github.com/. Recuperado: 2022-04-30.
60. Vaithilingam, P., Zhang, T., Glassman, E.:. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models, en CHI Conference on

Human Factors in Computing Systems Extended Abstracts, Association for Computing Machinery, New York, NY, USA, Article 332, 1–7 (2022).
61. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., Sutton, C.: Program synthesis with large language models, arXiv:2108.07732 (2021).
62. Tunstall, L., von Werra, L., Wolf, T.: Natural Language Processing with Transformers. O'Reilly Media, Inc. (2022).
63. Xu, F., Alon, U., Neubig, G., Hellendoorn, V.: A Systematic Evaluation of Large Language Models of Code. arXiv: 2202.13169 (2022).