



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Del Error a la Solución: Explorando Modelos de Inteligencia Artificial Generativa en la Detección y Corrección de Bugs de Programación.

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Fornet Llorca, Guillem

Tutor/a: Pelechano Ferragud, Vicente

Director/a Experimental: Mestre Gascón, Antoni

CURSO ACADÉMICO: 2023/2024

Resumen

Este Trabajo de Fin de Grado (TFG) se dedica a la evaluación del uso de ChatGPT, un avanzado modelo de lenguaje desarrollado por OpenAI, específicamente en su versión GPT-4, para la detección y corrección de errores en programas de software. Se explora su potencial como herramienta de reparación automática de programas (APR, Automatic Program Repair), con un enfoque particular en el impacto del prompting en su rendimiento. Además, se desarrollará un modelo GPT especializado utilizando GPT Builder, otra herramienta de OpenAI.

El estudio incluye una revisión exhaustiva de los principios fundamentales de la detección y corrección de errores, así como del ciclo de vida de un bug. Se analizarán las capacidades y limitaciones de ChatGPT, comparándolo con otros modelos similares. La investigación se estructura en fases, aplicando técnicas de prompting progresivas y diversas formas de información para identificar la metodología más efectiva para crear un GPT personalizado. La investigación concluirá con un primer prototipo de GPT creado mediante GPT Builder y en base a todos los resultados obtenidos se creará un GPT especializado en la detección y resolución de bugs de software.

El trabajo concluye con un análisis crítico de los resultados obtenidos, evaluando la metodología propuesta y discutiendo posibles mejoras y futuras direcciones de investigación en la detección y resolución de errores con ChatGPT.

Palabras clave: Modelos de Lenguaje, Diseño de Software, Detección de Errores, IA Generativa, Corrección de Bugs

Resum

Aquest Treball de Fi de Grau (TFG) es dedica a l'avaluació de l'ús de ChatGPT, un avançat model de llenguatge desenvolupat per OpenAI, específicament en la seua versió GPT-4, per a la detecció i correcció d'errors en programes de software. S'explora el seu potencial com a eina de reparació automàtica de programes (APR, Automatic Program Repair), amb un enfocament particular en l'impacte del prompting en el seu rendiment. A més, es desenvoluparà un model GPT especialitzat utilitzant GPT Builder, una altra eina d'OpenAI.

L'estudi inclou una revisió exhaustiva dels principis fonamentals de la detecció i correcció d'errors, així com del cicle de vida d'un bug. S'analitzaran les capacitats i limitacions de ChatGPT, comparant-lo amb altres models similars. La investigació s'estructura en fases, aplicant tècniques de prompting progressives i diverses formes d'informació per identificar la metodologia més efectiva per a crear un GPT personalitzat. La investigació conclourà amb un primer prototip de GPT creat mitjançant GPT Builder i, en base a tots els resultats obtinguts, es crearà un GPT especialitzat en la detecció i resolució de bugs de software.

El treball conclou amb una anàlisi crítica dels resultats obtinguts, avaluant la metodologia proposada i discutint possibles millores i futures direccions d'investigació en la detecció i resolució d'errors amb ChatGPT.

Paraules clau: Models de Llenguatge, Disseny de Programari, Detecció d'Errors, IA Generativa, Correcció de Bugs

Abstract

This Bachelor's Thesis (TFG) is dedicated to evaluating the use of ChatGPT, an advanced language model developed by OpenAI, specifically its GPT-4 version, for detecting and correcting errors in software programs. The study explores its potential as an Automatic Program Repair (APR) tool, with a particular focus on the impact of prompting on its performance. Additionally, a specialized GPT model will be developed using GPT Builder, another tool from OpenAI.

The study includes a comprehensive review of the fundamental principles of error detection and correction, as well as the bug lifecycle. The capabilities and limitations of ChatGPT will be analyzed, comparing it to other similar models. The research is structured in phases, applying progressive prompting techniques and various forms of information to identify the most effective methodology for creating a customized GPT. The research will conclude with a first prototype of GPT created using GPT Builder and, based on all the results obtained, a GPT specialized in the detection and resolution of software bugs will be developed.

The thesis concludes with a critical analysis of the results obtained, evaluating the proposed methodology and discussing possible improvements and future research directions in bug detection and resolution with ChatGPT.

Key words: Language Models, Software Design, Error Detection, Generative AI, Bug Fixing

Índice general

Índice general	VII
Índice de figuras	IX
Índice de tablas	IX
<hr/>	
1 Introducción	1
1.1 Contexto Actual y Motivación	2
1.2 Objetivos y Preguntas de Investigación	3
1.3 Estructura del documento	3
2 Fundamentos	5
2.1 Proceso de detección y resolución de bugs	5
2.1.1 Técnicas de debugging	5
2.1.2 Ciclo de resolución de bugs	6
2.2 Análisis estático de código y Reparación Automática de Programas (APR)	6
2.2.1 Análisis estático de código	7
2.2.2 APR y sus metodologías	7
2.3 Modelos de lenguaje y Transformers	8
2.3.1 Fundamentos de los Transformers: Introducción a la arquitectura de los modelos de lenguaje basados en transformers y su funciona- miento	8
2.3.2 GPT-3.5 vs GPT-4.0	9
2.4 GPT Builder	9
2.4.1 Características Técnicas de GPT Builder	10
2.4.2 Proceso de Creación de un Modelo Personalizado	10
2.4.3 Beneficios de Utilizar GPT Builder	10
2.5 Prompting	11
3 Detección y Resolución de Bugs con ChatGPT	13
3.1 Introducción a la investigación	13
3.1.1 Relevancia del estudio	13
3.2 Estudios previos	14
3.3 Fases de investigación	16
4 Fase 1. Establecer una línea base: Ejecución sin contexto	17
4.1 Prompt escogido	17
4.2 Resultados y discusión	18
4.3 Desarrollo de métricas para la evaluación	19
4.3.1 Justificación de los Pesos en la Evaluación Integral	21
4.4 Evaluación de Resultados	22
4.5 Resumen de Resultados	22
5 Fase 2. Creación del primer prompt: Guías y restricciones	25
5.1 Prompt escogido	25
5.2 Resultados y discusión	29
5.3 Evaluación de Resultados	30
5.4 Resumen de Resultados	31

6 Fase 3. Mejora de prompt: Pruebas unitarias	33
6.1 Prompt escogido	33
6.2 Resultados y Discusión	36
6.3 Evaluación de Resultados	37
6.4 Resumen de Resultados	38
7 Fase 4. GPT Builder: Primer prototipo	39
7.1 Creación del GPT: Uso ordinario de la herramienta	39
7.2 Método de estudio	43
7.3 Resultados y discusión	46
7.4 Normalización de métricas para incluir nuevos factores	47
7.4.1 Penalización por Peticiones Extra	48
7.4.2 Justificación de la Penalización por Peticiones Extra	48
7.4.3 Conclusión	49
7.5 Evaluación de Resultados	50
7.6 Resumen de Resultados	50
7.7 Conclusiones de la investigación	51
8 Producto final: GPT-based APR Tool	53
8.1 Prompting y creación de la herramienta APR basada en GPT	54
8.2 Resultados y Discusión	62
8.3 Evaluación de Resultados	63
8.4 Resumen de Resultados	64
9 Conclusiones	65
9.1 Resumen de los resultados obtenidos	65
9.2 Evaluación de la metodología	66
9.3 Futuras investigaciones	67
9.4 Conclusiones finales	68
Bibliografía	69
A Objetivos de Desarrollo Sostenible (ODS)	71

Índice de figuras

2.1	Ciclo de resolución de bugs	6
2.2	Ejemplo de uso de PVS-Studio en Visual Studio	7
7.1	Respuesta de GPT Builder a la petición del primer prototipo de GPT especializado	41
7.2	Configuración del primer prototipo de GPT especializado creada a partir de nuestra petición	42
7.3	Paso final de la creación del primer prototipo de GPT con la herramienta GPT Builder	43
8.1	Conversación con GPT Builder para la creación del modelo final	56
8.2	Pestaña de configuración GPT personalizado definitivo	59
8.3	Instrucciones del GPT personalizado definitivo, Parte 1	59
8.4	Instrucciones del GPT personalizado definitivo, Parte 2	60
8.5	Instrucciones del GPT personalizado definitivo, Parte 3	60
8.6	Instrucciones del GPT personalizado definitivo, Parte 4	61
8.7	Instrucciones del GPT personalizado definitivo, Parte 5	61

Índice de tablas

3.1	Tabla comparativa del estudio "An Analysis of the Automatic Bug Fixing Performance of ChatGPT"	15
4.1	Resumen de resultados de la primera fase del estudio	19
5.1	Resumen de resultados de la segunda fase del estudio	30
6.1	Resumen de resultados de la tercera fase del estudio	37
7.1	Resumen de resultados de la cuarta fase del estudio	47
8.1	Resumen de resultados de la herramienta GPT-based APR Tool	63
A.1	Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS)	71

CAPÍTULO 1

Introducción

La inteligencia artificial (IA) ha evolucionado notablemente desde sus orígenes, pasando de ser un campo de estudio exclusivo de académicos, investigadores y tecnólogos, a convertirse en una tecnología presente en la vida cotidiana. Lo que comenzó como proyectos de investigación en universidades y laboratorios se ha transformado en herramientas y aplicaciones que utilizamos todos los días, desde preguntarle al ChatGPT qué programa de lavadora conviene para unas sábanas de bambú hasta integraciones en aplicaciones que nos hacen la mitad del trabajo (Microsoft Copilot), así como una IA que permite a cualquier persona crear canciones del género y letra que desee con un click (Udio). La IA es definitiva, y ha venido para quedarse. Pero ¿qué es la Inteligencia Artificial?

La IA ha experimentado un crecimiento y avance notables, en parte, debido a la flexibilidad de su definición. La falta de un consenso universal sobre qué constituye exactamente la IA ha permitido a los expertos del campo avanzar con un sentido intuitivo de dirección (One Hundred Years Study on Artificial Intelligence, 2016). Según Nils J. Nilsson, la IA implica dotar a las máquinas de inteligencia, entendida esta como la capacidad de actuar adecuadamente y con previsión (The Quest for Artificial Intelligence: A History of Ideas and Achievements, 2010). Este enfoque ha fomentado una exploración más libre y creativa dentro del campo.

Por lo que respecta a la IA en la actualidad, está provocando una notable transformación en el mundo, prometiendo revoluciones en campos tan variados como la salud, donde podría predecir enfermedades y dar diagnósticos precisos; el transporte, con el auge de los vehículos autónomos; y la educación, ofreciendo caminos de aprendizaje personalizados. Además, la IA desempeña un papel crucial en el cumplimiento de los Objetivos de Desarrollo Sostenible, mejorando el acceso a la educación, la salud, y combatiendo el cambio climático, la pobreza y el hambre (Anna Visvizi Sustainability, 2022),.

El impacto de la inteligencia artificial (IA) en la economía presenta varias caras, prometiendo no solo transformar la naturaleza del trabajo sino también impulsar el crecimiento económico. Además, aunque se teme que la automatización pueda desplazar empleos, la IA también se proyecta para crear nuevas oportunidades laborales en sectores que requieren nuevas habilidades. Finalmente, estudios del Foro Económico Mundial sugieren que adoptar tecnologías IA tiene el potencial de incrementar significativamente la productividad y, por ende, el Producto Interior Bruto (PIB) global (World Economic Forum, 2023). La clave está en la adaptación y capacitación de la fuerza laboral para navegar este cambio.

Sin embargo, la implementación de la IA también presenta desafíos significativos, incluyendo preocupaciones sobre la privacidad de los datos, amenazas de ciberseguridad,

y el comentado potencial desplazamiento laboral. La evolución de la IA demanda un enfoque equilibrado que contemple tanto sus beneficios como los desafíos que presenta.

En el campo de la ingeniería de software, la detección y resolución de bugs representa un desafío constante. La introducción de Large Language Models (LLM) como GPT-4 ha abierto nuevas vías para abordar este desafío, prometiendo herramientas más efectivas y eficientes para los desarrolladores de software. A pesar de su potencial, la capacidad de estos modelos para detectar y corregir errores en el código aún está en proceso de ser plenamente comprendida y aprovechada.

1.1 Contexto Actual y Motivación

La reciente exploración del uso de ChatGPT y otros Large Language Models (LLM) en tareas complejas como la reparación automática de programas ha destacado la importancia crítica de los prompts en la eficacia de estos modelos. Investigaciones han demostrado que un diseño cuidadoso de los prompts puede aumentar significativamente la capacidad de ChatGPT en diversos dominios.

Por ejemplo, en el campo de la generación de código, se ha demostrado que, con el diseño cuidadoso de prompts, se produce una mejora de rendimiento de ChatGPT a la hora de generar de texto-a-código y de código-a-código (Improving ChatGPT Prompt for Code Generation, Liu et al., 2023). Esto subraya cómo el prompting puede dirigir de manera efectiva a ChatGPT para generar respuestas más precisas y útiles.

Además, una detallada investigación sobre la capacidad de razonamiento matemático de ChatGPT, revela que el prompting de cadena de pensamiento (CoT) es particularmente beneficioso para mejorar las capacidades de razonamiento de este, aunque muestra limitaciones en tareas puramente aritméticas (When do you need Chain-of-Thought Prompting for ChatGPT?, Chen et al., 2023). Este hallazgo enfatiza la necesidad de adaptar los prompts según la naturaleza de la tarea para optimizar los resultados, demostrando que más no es siempre mejor.

En ingeniería, la metodología desarrollada para interactuar con modelos de lenguaje mediante prompts específicos ha demostrado aumentar la precisión en aplicaciones prácticas (Prompt Engineering: a methodology for optimizing interactions with AI-Language Models in the field of engineering, Velásquez-Henao et al., 2023). Este enfoque innovador resalta cómo una estrategia bien pensada de prompting puede transformar la interacción con la IA en herramientas concretas y eficaces.

Un estudio sobre evaluación de traducciones introduce un método de prompting llamado "Error Analysis Prompting", que combina CoT con análisis de errores para afinar las evaluaciones realizadas por ChatGPT, acercándolas a las evaluaciones humanas (Error Analysis Prompting Enables Human-Like Translation Evaluation in Large Language Models: A Case Study on ChatGPT, Lu et al., 2023). Esto demuestra la capacidad del prompting para refinar y dirigir la inteligencia artificial hacia resultados más humanos.

Finalmente, en el ámbito de la robótica, se ha mostrado que técnicas de prompting adaptadas permiten que ChatGPT se ajuste a diversas tareas robóticas, simuladores y factores de forma, evidenciando la versatilidad de los prompts en contextos de alta variabilidad y exigencia técnica (ChatGPT for Robotics: Design Principles and Model Abilities Vemprala et al., 2023).

Estos estudios ilustran la capacidad del diseño de prompts para maximizar el potencial de los LLM como ChatGPT, destacando su importancia para el avance y la eficiencia en distintas aplicaciones técnicas y científicas. La adecuada formulación de estos prompts es fundamental para explotar al máximo las capacidades de la IA, por tanto es de extre-

ma relevancia la exploración de estos para hacer de ChatGPT una herramienta poderosa para la detección y resolución de bugs.

1.2 Objetivos y Preguntas de Investigación

A continuación, serán expuestos los diferentes objetivos de este trabajo de investigación

- Evaluar la capacidad de GPT-4 para detectar y resolver bugs de software sin contexto adicional, haciendo también una comparación con su modelo previo (GPT-3.5).
- Investigar cómo las técnicas avanzadas de prompting pueden mejorar la eficacia de GPT-4 en la reparación de bugs.
- Desarrollar y evaluar un GPT especializado en la detección y resolución de bugs utilizando GPT Builder, proporcionando así una herramienta práctica para los desarrolladores.

Preguntas

- ¿Hasta qué punto puede GPT-4 identificar y corregir bugs en el software sin información contextual adicional?
- ¿De qué manera las técnicas de prompting avanzadas mejoran el rendimiento de GPT-4 en la tarea de reparación automática de programas?
- ¿Cuál es la eficacia de un GPT especializado, desarrollado a través de GPT Builder, en comparación con GPT-4 estándar en la detección y resolución de bugs?

1.3 Estructura del documento

Este documento se organiza en nueve capítulos que abordan de manera sistemática la investigación sobre la efectividad de ChatGPT en la detección y resolución de bugs en software. En el Capítulo 1, se proporciona una introducción al tema, estableciendo el contexto actual y la motivación detrás del estudio, los objetivos y preguntas de investigación, así como la estructura del documento.

El Capítulo 2 detalla los fundamentos teóricos y técnicos relevantes para la investigación. Se exploran el proceso de detección y resolución de bugs, las técnicas de debugging y el ciclo de resolución de bugs. Además, se describen las metodologías de análisis estático de código y la reparación automática de programas (APR), así como los modelos de lenguaje y la arquitectura de los Transformers, incluyendo una comparación entre GPT-3.5 y GPT-4.0. También se presentan las características técnicas y beneficios de utilizar GPT Builder, junto con las técnicas de prompting.

El Capítulo 3 introduce la investigación específica sobre la detección y resolución de bugs con ChatGPT. Aquí se discuten la relevancia del estudio y los estudios previos, seguidos por una descripción detallada de las fases de investigación. Cada fase se aborda en capítulos subsecuentes, comenzando con la Fase 1 en el Capítulo 4, donde se establece una línea base mediante la ejecución de programas con bugs sin proporcionar contexto.

En el Capítulo 5, se presenta la Fase 2, que incluye la creación de un primer prompt con guías y restricciones específicas para mejorar la precisión en la detección de bugs. El

Capítulo 6 abarca la Fase 3, donde se mejora el prompt incluyendo pruebas unitarias fallidas. El Capítulo 7 describe la Fase 4, que consiste en la creación de un primer prototipo de GPT utilizando GPT Builder como lo haría un usuario ordinario.

El Capítulo 8 detalla la fase final del proyecto, donde se desarrolla una herramienta APR basada en GPT utilizando técnicas avanzadas de prompting para optimizar el rendimiento del modelo. Finalmente, el Capítulo 9 presenta las conclusiones del estudio, con un resumen de los resultados obtenidos, una evaluación de la metodología empleada y propuestas para futuras investigaciones en este campo.

Cada capítulo se estructura de manera que proporcione una visión clara y coherente del proceso de investigación, permitiendo al lector comprender tanto los aspectos técnicos como las implicaciones prácticas de los hallazgos. Esta organización facilita el seguimiento del desarrollo del proyecto y la integración de los diferentes componentes en una solución efectiva para la reparación automática de programas utilizando modelos de lenguaje avanzados.

CAPÍTULO 2

Fundamentos

2.1 Proceso de detección y resolución de bugs

La detección y resolución de bugs son etapas críticas en el desarrollo de software que aseguran la calidad y la funcionalidad del producto final. Este proceso comienza con la identificación de un problema o fallo dentro del código, que puede ser reportado por usuarios o descubierto durante pruebas internas. Una vez identificado el bug, el equipo de desarrollo trabaja para comprender su causa raíz, lo cual a menudo implica revisar y depurar el código para rastrear el error hasta su origen.

2.1.1. Técnicas de debugging

El debugging es una técnica esencial en el desarrollo de software que involucra diversas estrategias para localizar y corregir errores de código. Los desarrolladores utilizan una variedad de herramientas y métodos para facilitar este proceso, asegurando que el software funcione correctamente y de manera eficiente. Algunas de las técnicas más comunes incluyen el uso de debuggers, pruebas unitarias y registros de ejecución, entre otros.

Los debuggers son herramientas que permiten a los desarrolladores ejecutar un programa paso a paso, observar el comportamiento del software y examinar el estado de las variables en tiempo real. Ejemplos populares de debuggers incluyen GDB (GNU Debugger) para C y C++, y JDB (Java Debugger) para Java. Estas herramientas permiten a los desarrolladores establecer puntos de interrupción (breakpoints), que detienen la ejecución del programa en puntos específicos, permitiendo un análisis detallado del estado del programa en esos momentos.

Las pruebas unitarias son otra técnica fundamental en el debugging. Consisten en escribir pruebas automáticas para pequeñas unidades de código, típicamente funciones o métodos, para asegurarse de que funcionen correctamente en diversos escenarios. Frameworks como JUnit para Java, NUnit para .NET y pytest para Python son ampliamente utilizados en la industria.

Los registros de ejecución capturan el estado del software en tiempo de ejecución, proporcionando información detallada sobre el flujo del programa y los valores de las variables en diferentes puntos de la ejecución. Herramientas como Log4j para Java, Serilog para .NET y logging para Python son esenciales para esta tarea. Los registros detallados pueden ser especialmente útiles para diagnosticar problemas intermitentes o difíciles de reproducir, ya que proporcionan un historial completo de la ejecución del programa.

2.1.2. Ciclo de resolución de bugs

El ciclo de resolución de bugs es un proceso sistemático que sigue la identificación inicial del error hasta su completa resolución y verificación posterior. Este ciclo comienza con la confirmación del bug, donde se verifica que el error reportado es reproducible y se clasifica según su gravedad y urgencia. Posteriormente, se asigna a los desarrolladores apropiados para su corrección. Una vez corregido, el código modificado se somete a pruebas de regresión para asegurar que la solución no haya introducido nuevos problemas. Finalmente, el bug corregido se documenta detalladamente y se cierra en el sistema de seguimiento de errores. Este método garantiza que cada error se maneje de manera eficiente, minimizando el impacto en el desarrollo general del producto y manteniendo la calidad del software. Este enfoque estructurado es crucial para la gestión eficaz de los ciclos de desarrollo de software y la mejora continua de los productos.

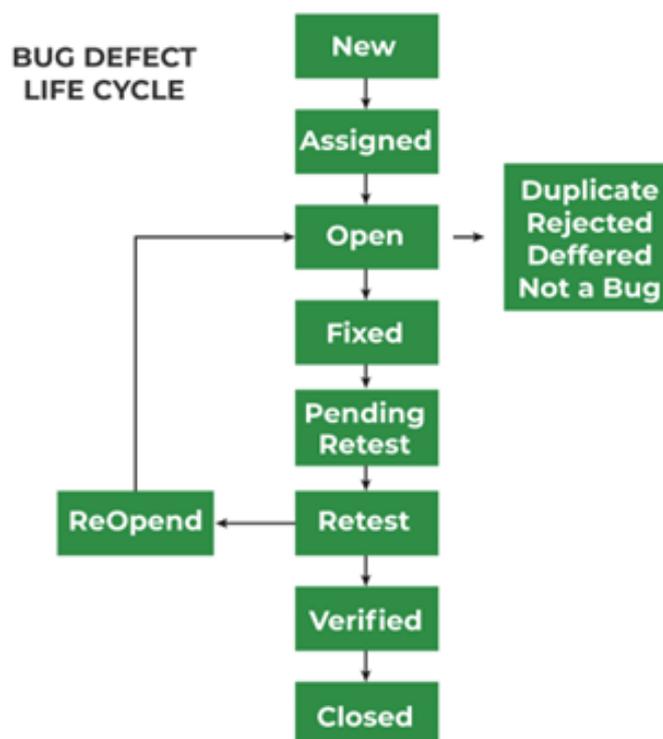


Figura 2.1: Ciclo de resolución de bugs

Fuente: GeeksForGeeks, 2023

2.2 Análisis estático de código y Reparación Automática de Programas (APR)

El análisis estático de código y la Reparación Automática de Programas (APR) son técnicas fundamentales en el mantenimiento y mejora de la calidad del software. Estas metodologías permiten identificar y solucionar errores de forma proactiva, incluso antes de que el software se ejecute, lo cual optimiza los ciclos de desarrollo y asegura un producto final más robusto y confiable.

2.2.1. Análisis estático de código

El análisis estático de código es una técnica crítica en el desarrollo de software, empleada para detectar problemas antes de la ejecución del programa mediante herramientas que examinan el código en busca de patrones problemáticos y vulnerabilidades sin necesidad de ejecutarlo. Entre las herramientas más utilizadas se encuentra SonarQube, conocida por su capacidad de detectar bugs, vulnerabilidades y code smells en diversos lenguajes de programación, y Coverity, que es reconocida por su habilidad para manejar grandes bases de código y es utilizada por numerosas corporaciones grandes. Empresas líderes como Google y Microsoft integran estas herramientas en sus estrategias de desarrollo; Google, por ejemplo, utiliza ErrorProne para su código Java para detectar errores comunes de manera anticipada, mientras que Microsoft emplea PVS-Studio, eficaz en la identificación de tipos de errores y problemas de seguridad. Estas herramientas no solo ayudan a mantener la calidad y la seguridad del software, sino que también son un componente esencial en el mantenimiento de la eficiencia y la fiabilidad del desarrollo de software a gran escala.

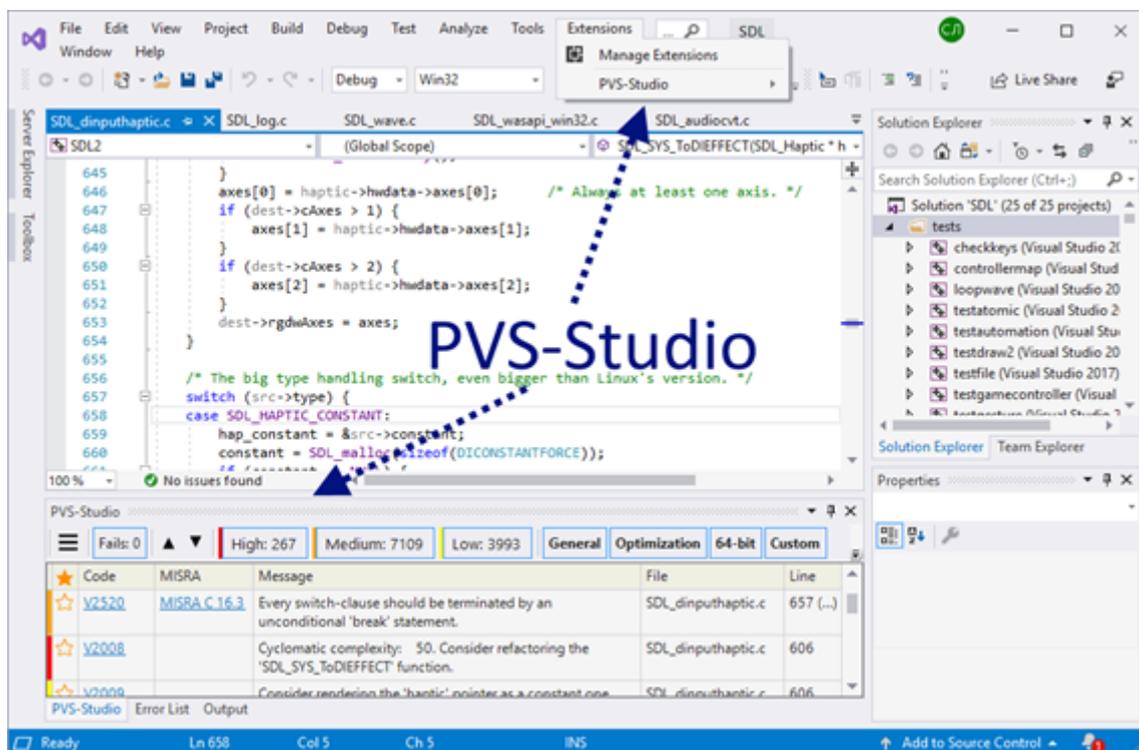


Figura 2.2: Ejemplo de uso de PVS-Studio en Visual Studio

Fuente: pvs-studio, 2019

2.2.2. APR y sus metodologías

La Reparación Automática de Programas (APR) emplea técnicas avanzadas para corregir errores de software automáticamente. Herramientas destacadas en este campo incluyen SapFix, utilizada por Facebook para depurar aplicaciones móviles automáticamente, y GenProg, que emplea genética computacional para mejorar la eficiencia del mantenimiento de software. Aunque estas herramientas han revolucionado la eficiencia del mantenimiento, el APR tradicional enfrenta desafíos significativos, como altos costos operativos y una gran demanda de recursos computacionales, además de la necesidad de actualizaciones frecuentes para manejar nuevas tecnologías y lenguajes de programación.

En contraste, el APR impulsado por IA, utilizando modelos como ChatGPT, puede entender mejor el contexto y la funcionalidad del código, generando correcciones más inteligentes y contextualizadas. Estos modelos se benefician de un aprendizaje continuo, mejorando sus respuestas con el tiempo a medida que se exponen a más datos. Este enfoque no solo aumenta la precisión de las correcciones, sino que también optimiza los recursos al reducir la carga de las tareas repetitivas en los desarrolladores. Además, se está explorando la dirección de utilizar APR conversacional basado en grandes modelos preentrenados como ChatGPT. Este enfoque permite a los desarrolladores interactuar con un chatbot de reparación que propone soluciones a errores. Los desarrolladores pueden proporcionar al chatbot información útil sobre el error, como los reportes de bugs o los códigos sospechosos, y las soluciones sugeridas por el chatbot pueden ser validadas por herramientas externas y feedback dinámico, lo cual ayuda a refinar aún más las correcciones sugeridas (A Survey of Learning-based Automated Program Repair, Quanjun Zhang et al., 2023). Estos avances muestran un cambio prometedor de los métodos tradicionales a soluciones más dinámicas y adaptativas, aprovechando las capacidades de aprendizaje y adaptación de los modelos de IA para enfrentar desafíos complejos en la reparación de software.

2.3 Modelos de lenguaje y Transformers

Los Transformers han revolucionado el campo del Procesamiento del Lenguaje Natural (PLN) con su arquitectura distintiva que se centra en el mecanismo de atención. Este mecanismo permite a los modelos ponderar la importancia de cada palabra en un contexto, independientemente de su posición en el texto, facilitando una comprensión profunda del lenguaje. Desde su presentación en el innovador artículo "Attention is All You Need" de Vaswani et al. (2017), los Transformers han establecido la base para desarrollos notables en PLN.

2.3.1. Fundamentos de los Transformers: Introducción a la arquitectura de los modelos de lenguaje basados en transformers y su funcionamiento

La estructura de los Transformers permite el procesamiento paralelo de los datos, lo cual es una mejora significativa respecto a las arquitecturas recurrentes anteriores que procesaban el texto secuencialmente. Esto no solo mejora la eficiencia del entrenamiento sino que también amplía la capacidad de los modelos para manejar secuencias de texto largas, un desafío persistente en modelos anteriores.

Además, los Transformers han sido fundamentales en el desarrollo de modelos como BERT (Bidirectional Encoder Representations from Transformers) y GPT (Generative Pre-trained Transformer). BERT utiliza una técnica de entrenamiento bidireccional que le permite entender el contexto de una palabra basándose tanto en las palabras anteriores como en las posteriores dentro de una oración, lo que es crucial para tareas como la comprensión de texto y la clasificación de sentimientos. Por otro lado, GPT aprovecha un enfoque generativo, donde el modelo aprende a predecir la siguiente palabra en una secuencia, facilitando su capacidad para generar texto coherente y contextualmente relevante.

Estos avances subrayan cómo los Transformers están impulsando no solo las capacidades de comprensión y generación de texto, sino también abriendo nuevas posibilidades en campos como la traducción automática, resumen automático y la generación de contenido interactivo. La evolución continua de esta tecnología promete avances aún

más significativos en el campo del PLN, marcando una era de modelos cada vez más sofisticados y adaptativos.

2.3.2. GPT-3.5 vs GPT-4.0

En la evolución de los modelos de lenguaje natural, la transición de GPT-3.5 a GPT-4.0 representa un avance significativo en la capacidad de procesamiento y generación de texto. Ambos modelos pertenecen a la serie Generative Pre-trained Transformers (GPT) desarrollada por OpenAI, la cual se ha caracterizado por su enfoque en el aprendizaje profundo y la arquitectura de transformers. Esta comparación se centrará en el número de tokens y parámetros, así como en el papel crucial que estos componentes desempeñan en los modelos de lenguaje grandes (LLM, por sus siglas en inglés) y su rendimiento.

GPT-3.5, una evolución de GPT-3, está compuesto por aproximadamente 175 mil millones de parámetros. Los parámetros en un modelo de lenguaje son coeficientes que se ajustan durante el proceso de entrenamiento para minimizar el error en la predicción de la próxima palabra en una secuencia de texto. Estos parámetros permiten al modelo capturar y generalizar patrones en los datos de entrenamiento, influenciando directamente su capacidad para generar texto coherente y contextual.

Por otro lado, GPT-4.0 ha incrementado el número de parámetros de manera considerable, siendo esta información desconocida, se ha estimado que GPT-4 podría llegar al billón de parámetros. Este aumento en el número de parámetros permite al modelo tener una mayor capacidad de aprendizaje y representación de la complejidad del lenguaje natural. Los estudios sugieren que, en general, un mayor número de parámetros mejora el rendimiento del modelo en diversas tareas de procesamiento del lenguaje natural (PLN), como la traducción automática, el resumen de textos y la respuesta a preguntas (Brown et al., 2020).

En cuanto al número de tokens, GPT-3.5 puede procesar contextos de hasta 4096 tokens, mientras que GPT-4.0 ha ampliado esta capacidad a 8192 tokens. Los tokens son unidades básicas de texto que el modelo utiliza para el procesamiento, y su número influye en la cantidad de contexto que el modelo puede considerar al generar respuestas. Un mayor límite de tokens permite al modelo tener una visión más amplia del contexto, lo que resulta en respuestas más precisas y contextualmente relevantes, especialmente en tareas que requieren comprensión de textos largos o complejos.

El papel de los tokens y parámetros en los LLM es fundamental para su rendimiento. Un mayor número de parámetros generalmente se traduce en una mayor capacidad del modelo para aprender y generalizar desde los datos de entrenamiento, mejorando así su capacidad para generar texto de alta calidad. Simultáneamente, la capacidad de manejar un mayor número de tokens amplía el contexto que el modelo puede considerar, lo cual es crucial para tareas que requieren una comprensión profunda y coherente del texto.

En resumen, la evolución de GPT-3.5 a GPT-4.0 refleja un avance significativo en el campo de la inteligencia artificial y los modelos de lenguaje. El aumento en el número de parámetros y la capacidad de procesar más tokens proporcionan al modelo una mayor habilidad para manejar y generar texto complejo y contextualizado, mejorando así su rendimiento en una amplia gama de aplicaciones de procesamiento del lenguaje natural.

2.4 GPT Builder

La funcionalidad GPT Builder, desarrollada como parte del ecosistema de OpenAI, proporciona a los usuarios una plataforma avanzada para la creación, personalización y

extensión de modelos de lenguaje basados en GPT. Este sistema permite a los desarrolladores y usuarios técnicos ajustar y mejorar las capacidades de los modelos GPT según sus necesidades específicas, mediante la configuración de parámetros, la incorporación de secciones de conocimiento y la integración de permisos adicionales para la ejecución de tareas específicas.

2.4.1. Características Técnicas de GPT Builder

GPT Builder se distingue por su capacidad para permitir la personalización profunda de modelos de lenguaje mediante una interfaz de conversación similar a la del propio ChatGPT. Una de sus características más destacadas es la sección de conocimiento (knowledge section), que permite a los usuarios cargar datos específicos o documentos que el modelo puede utilizar para mejorar sus respuestas en contextos particulares. Esta sección facilita la adaptación del modelo a dominios específicos, mejorando así su precisión y relevancia en áreas especializadas.

Además, GPT Builder permite la asignación de permisos para realizar búsquedas en internet y ejecutar código. Estos permisos se configuran mediante una interfaz de usuario intuitiva o a través de API específicas, proporcionando al modelo la capacidad de acceder a información en tiempo real y realizar tareas computacionales complejas. Por ejemplo, un usuario puede configurar el modelo para que realice consultas en bases de datos, acceda a APIs externas, o ejecute scripts de Python para procesar datos y generar resultados dinámicos.

2.4.2. Proceso de Creación de un Modelo Personalizado

El proceso de crear un modelo personalizado con GPT Builder sigue varios pasos esenciales. Primero, el usuario debe definir los objetivos y el dominio de aplicación del modelo. Para ello, el GPT Builder guía al usuario en la creación del GPT personalizado, haciendo reiteradas cuestiones al usuario, así animando a este a explicarse detalladamente sobre el propósito del GPT que se está creando y solucionar malentendidos que pueden llevar a comportamientos no deseados. Este proceso pretende afinar el modelo a la intención original del usuario. Además, también podemos editar el prompt de comportamiento generado por el GPT Builder en la configuración del GPT personalizado. Esto incluye la selección de datos de entrenamiento específicos, que pueden ser cargados en la sección de conocimiento. A continuación, el usuario configura los parámetros del modelo, como las directrices para el procesamiento de datos.

Una vez configurada la base del modelo, el usuario puede asignar permisos para la ejecución de tareas adicionales. Por ejemplo, puede permitir que el modelo realice búsquedas en internet para obtener información actualizada o ejecutar scripts para analizar datos en tiempo real. Esta capacidad de ejecución se gestiona a través de un sistema de permisos seguro que garantiza que el modelo opere dentro de los límites establecidos por el usuario.

2.4.3. Beneficios de Utilizar GPT Builder

La personalización mediante GPT Builder ofrece numerosos beneficios. En primer lugar, permite la adaptación del modelo a necesidades específicas, mejorando la precisión y relevancia de sus respuestas. Además, la capacidad de integrar datos personalizados y permisos para tareas adicionales amplía significativamente las aplicaciones potenciales

del modelo, desde la automatización de procesos empresariales hasta el soporte técnico especializado.

Un ejemplo ilustrativo de la aplicación de GPT Builder se encuentra en la GPT Store, donde los usuarios pueden encontrar modelos preconfigurados para diversas tareas. Por ejemplo, un modelo personalizado para asistencia médica puede acceder a bases de datos médicas y proporcionar recomendaciones basadas en el conocimiento médico actualizado, mientras que un modelo de análisis financiero puede ejecutar scripts para procesar datos de mercado en tiempo real y ofrecer insights financieros precisos.

En resumen, GPT Builder proporciona una plataforma robusta y flexible para la personalización de modelos de lenguaje, permitiendo a los usuarios adaptar y expandir las capacidades de GPT según sus necesidades específicas. Además, a través de la configuración de la sección de conocimiento y la asignación de permisos adicionales, los usuarios pueden crear modelos altamente especializados que mejoran la eficiencia y la eficacia en una amplia gama de aplicaciones.

2.5 Prompting

En la introducción de este documento, se ha establecido que el prompting es una técnica esencial en la interacción con modelos de lenguaje de gran escala (LLM) como ChatGPT. Diversos estudios han destacado la relevancia del prompting en la forma en que ChatGPT maneja, procesa y responde a la información proporcionada. Por lo tanto, resulta fundamental comprender qué es el prompting y cómo se aplicará en esta investigación.

El prompting se refiere a la formulación y estructuración de entradas específicas presentadas a un modelo de IA para obtener respuestas precisas y útiles. Esta técnica no solo guía al modelo en la interpretación del contexto, sino que también influye en la calidad y relevancia de las respuestas generadas. En el contexto de los LLM, el prompting actúa como un puente entre el usuario y la inteligencia artificial, permitiendo una comunicación más efectiva y orientada a objetivos específicos.

En el marco de esta investigación, se explora el uso de ChatGPT como una herramienta de reparación automática de programas (APR). Para ello, se implementarán diversas técnicas de prompting en diferentes fases del proceso. Es crucial analizar cómo estas técnicas afectan el rendimiento del modelo, ya que la precisión y la utilidad de las respuestas de ChatGPT dependen en gran medida de la calidad del prompting.

Además, el estudio se centrará en evaluar cómo diferentes estrategias de prompting pueden optimizar la eficacia de ChatGPT en la detección y resolución de bugs. Por ejemplo, se investigará cómo la variación tanto en formulación como en cantidad de información de los prompts influye en la capacidad del modelo para identificar correctamente errores en el código y sugerir soluciones adecuadas. Asimismo, se examinarán los efectos de prompts detallados frente a prompts más generales, y cómo estos impactan en la profundidad y exactitud de las reparaciones propuestas.

En conclusión, el prompting desempeña un papel crucial en la interacción con modelos de lenguaje como ChatGPT. Esta investigación no solo profundizará en la comprensión de esta técnica, sino que también proporcionará insights valiosos sobre cómo su correcta aplicación puede mejorar significativamente el rendimiento y la utilidad de ChatGPT en diversas aplicaciones de reparación automática de programas.

CAPÍTULO 3

Detección y Resolución de Bugs con ChatGPT

3.1 Introducción a la investigación

El objetivo principal de esta investigación es explorar en profundidad la capacidad de GPT-4 para identificar y corregir errores en programas de software, aplicando un enfoque innovador basado en la técnica de prompting progresivo. Utilizando la suite de bugs QuixBugs, que contiene algoritmos conocidos poblados con bugs intencionados, este estudio busca proporcionar un análisis detallado y riguroso de las potencialidades de GPT-4 dentro del ámbito de la reparación automática de programas (APR). Se espera que la metodología progresiva de prompting, que incrementa la cantidad y especificidad de la información proporcionada al modelo, explore el valor de darle conocimientos como la ubicación del bug o un input/output de ejemplo y permita mejorar significativamente la precisión de GPT-4 en la localización y resolución de estos bugs predefinidos. Además, se investigará cómo estas intervenciones afectan la capacidad de aprendizaje y adaptación del modelo, con el fin de desarrollar un GPT especializado que podría ser integrado en flujos de trabajo de desarrollo de software más amplios.

3.1.1. Relevancia del estudio

La implementación efectiva de herramientas de APR tiene el potencial de transformar radicalmente las prácticas de desarrollo de software, reduciendo significativamente tanto los costos asociados con el debugging manual como los tiempos de desarrollo. En este contexto, el estudio actual es especialmente relevante, ya que aborda directamente la necesidad de herramientas de APR más sofisticadas y adaptativas que puedan manejar la complejidad y dinamismo del software moderno. Al analizar la efectividad de los diferentes niveles de prompting en GPT-4, este estudio no solo busca optimizar el proceso de detección y corrección de errores, sino también generar conocimiento sobre las capacidades y limitaciones de los modelos de lenguaje avanzados en tareas específicas de programación. Los resultados esperados incluyen insights valiosos sobre la mejora de las interacciones entre humanos y sistemas basados en inteligencia artificial, contribuyendo así a la evolución del campo del APR y a la mejora continua de las metodologías de desarrollo de software. Además, la creación y evaluación de un GPT especializado basado en los hallazgos de este estudio proporcionará una base sólida para futuras investigaciones y aplicaciones prácticas en la industria del software.

3.2 Estudios previos

Un estudio previo relevante para la investigación es el titulado "An Analysis of the Automatic Bug Fixing Performance of ChatGPT" realizado por Dominik Sobania, Martin Briesch, Carol Hanna y Justyna Petke, proporciona una evaluación exhaustiva de la capacidad de GPT-3.5 en comparación con otras herramientas de reparación automática de programas (APR) utilizando la suite QuixBugs. El estudio se centra en comparar las capacidades de GPT-3.5 con otros enfoques basados en Deep Learning como CoCoNut y Codex, así como con los métodos tradicionales de APR.

De acuerdo con los resultados presentados en el estudio, GPT-3.5 demostró un rendimiento competitivo al corregir 19 de los 40 problemas de la suite QuixBugs, un logro que no solo lo posiciona al nivel de herramientas especializadas en Deep Learning sino que también muestra una ventaja significativa sobre las técnicas tradicionales de APR. Este hallazgo es crucial, ya que destaca la eficacia de GPT-3.5 en manejar errores de software complejos comparado con los métodos más convencionales.

Además, el estudio señala que GPT-3.5 tiene una variabilidad relativamente alta en la corrección de errores, solucionando algunos problemas solo en una o dos de las cuatro ejecuciones. Sin embargo, en problemas específicos como BUCKETSORT y FLATTEN, GPT-3.5 fue capaz de encontrar una solución en todas las ejecuciones. Esto subraya la necesidad de ejecutar múltiples solicitudes para obtener resultados consistentemente útiles.

Los resultados también indican que el rendimiento de GPT-3.5 es comparable al de Codex, lo que no sorprende dado que ambos provienen de la misma familia de modelos de lenguaje. A pesar de la proximidad en el número de problemas resueltos, aún existe potencial para mejorar a GPT-3.5, especialmente en cómo sus respuestas a menudo se acercan a la solución correcta sin alcanzarla completamente. Este estudio proporciona una base sólida para futuros esfuerzos en optimizar ChatGPT para tareas de reparación automática de programas, destacando la importancia de un enfoque iterativo y la integración de más información contextual para mejorar la precisión de las correcciones.

Además, el análisis profundiza en cómo la interacción mediante un sistema de diálogo y el enriquecimiento de los prompts con información contextual específica pueden mejorar sustancialmente la efectividad de GPT-3.5. Al proporcionar detalles adicionales como los mensajes de error observados o las salidas esperadas, GPT-3.5 incrementó su tasa de éxito en la corrección de bugs a 31 de 40. Este resultado demuestra que la integración de interacciones detalladas y contextualizadas puede ser clave para optimizar la precisión y la utilidad de las herramientas de APR basadas en modelos de lenguaje.

El estudio concluye que la capacidad de GPT-3.5 para interactuar y recibir información específica del usuario es un factor diferenciador importante que mejora significativamente su rendimiento. Este enfoque interactivo y la capacidad de integrar inputs detallados de forma efectiva presentan una ventaja sustancial sobre los métodos tradicionales y sugieren un camino prometedor para el desarrollo futuro de sistemas de APR más sofisticados y adaptativos.

Benchmark problem	GPT-3.5	Codex	CoCoNut	Standard APR
bitcount	× (0/4)	✓	✓	×
breadth-first-search	✓ (2/4)	✓	✓	×
bucket sort	✓ (4/4)	✓	✓	×
depth-first-search	× (0/4)	×	✓	×
detect-cycle	× (0/4)	×	×	✓
find-first-in-sorted	× (2/4)	✓	×	×
find-in-sorted	✓ (3/4)	×	✓	×
flatten	✓ (4/4)	✓	✓	×
gcd	× (0/4)	✓	✓	×
get-factors	✓ (1/4)	✓	✓	×
hanoi	× (0/4)	✓	✓	×
is-valid-parenthesization	✓ (2/4)	✓	×	×
kheapsort	× (0/4)	×	✓	×
knapsack	✓ (1/4)	✓	✓	✓
kth	× (0/4)	×	×	×
lcs-length	× (0/4)	×	✓	×
levenshtein	× (0/4)	×	×	✓
lis	× (0/4)	×	×	✓
longest-common-subsequence	× (0/4)	✓	✓	×
max-sublist-sum	× (0/4)	✓	×	×
mergesort	✓ (1/4)	×	×	✓
minimum-spanning-tree	× (0/4)	×	✓	×
next-palindrome	✓ (1/4)	×	✓	×
next-permutation	× (0/4)	×	×	×
pascal	✓ (1/4)	×	✓	×
possible-change	✓ (1/4)	✓	×	×
powerset	× (0/4)	✓	×	×
quicksort	✓ (1/4)	✓	✓	✓
reverse-linked-list	✓ (2/4)	✓	×	✓
rpn-eval	× (0/4)	×	✓	✓
shortest-path-length	✓ (1/4)	×	×	×
shortest-path-lengths	× (0/4)	×	×	×
shortest-paths	✓ (1/4)	×	×	×
shunting-yard	✓ (2/4)	×	×	×
sieve	× (0/4)	✓	✓	×
sqrt	✓ (1/4)	✓	✓	×
subsequences	✓ (1/4)	×	×	×
to-base	× (0/4)	×	✓	×
topological-ordering	× (0/4)	✓	×	×
wrap	× (0/4)	✓	×	×
Σ (Solved)	19	21	19	7

Tabla 3.1: Tabla comparativa del estudio "An Analysis of the Automatic Bug Fixing Performance of ChatGPT".

3.3 Fases de investigación

Para evaluar el rendimiento del LLM GPT-4 (desde ahora ChatGPT) en la detección y corrección de errores en software, se seguirá un procedimiento metodológico estructurado en cuatro fases distintas, utilizando la suite de bugs QuixBugs como referencia, similar al estudio previo. Cada fase está diseñada para explorar y optimizar diferentes aspectos del uso de ChatGPT en la reparación automática de programas (APR).

Fase 1: Evaluación inicial sin contexto

En esta fase inicial, se presentarán a ChatGPT los programas con errores de la suite QuixBugs sin proporcionar ningún contexto adicional. El objetivo es establecer una línea base de cómo el modelo maneja las correcciones de bugs de manera autónoma, identificando cuántos y cuáles bugs puede resolver sin ayuda externa. Esta evaluación ayudará a entender las capacidades intrínsecas del modelo y sus limitaciones iniciales en tareas de APR.

Fase 2: Desarrollo de un prompt mejorado

Basándose en los resultados y debilidades observadas en la fase 1, se diseñará un prompt mejorado que no solo compense las limitaciones identificadas, sino que también proporcione información contextual adicional a ChatGPT. Este prompt buscará guiar al modelo más efectivamente hacia la solución correcta, ofreciendo una región concreta donde se encuentra el bug y una serie de restricciones y guías para que ChatGPT oriente la resolución a las necesidades del estudio.

Fase 3: Integración de tests unitarios en el prompt

En la tercera fase, se enriquecerá aún más el prompt incorporando tests unitarios que fallan debido al bug presente en el código. La hipótesis es que al proporcionar a ChatGPT ejemplos específicos de cómo falla el código bajo ciertas condiciones, se podría mejorar significativamente su capacidad para proponer la corrección adecuada. Esta fase permitirá evaluar si la inclusión de resultados de tests unitarios afecta positivamente la precisión de las soluciones generadas por ChatGPT.

Fase 4: Creación y evaluación de un GPT personalizado

Utilizando los insights y los resultados más efectivos de las fases anteriores, se desarrollará una versión personalizada de GPT. Este modelo personalizado estará afinado específicamente para tareas de reparación de bugs, buscando integrar las mejores estrategias y enfoques de prompting descubiertos durante las fases experimentales. La evaluación final de este GPT personalizado comparará su rendimiento con las versiones estándar de ChatGPT utilizadas en las fases previas, determinando si la personalización ofrece mejoras significativas en la capacidad de APR.

Cada una de estas fases será documentada meticulosamente para asegurar que los resultados puedan ser verificables y reproducibles, contribuyendo a un entendimiento más profundo de cómo los modelos avanzados de lenguaje pueden ser optimizados para tareas específicas de reparación automática de programas.

CAPÍTULO 4

Fase 1. Establecer una línea base: Ejecución sin contexto

En la etapa inicial de esta investigación, se someterá a ChatGPT a una prueba utilizando programas defectuosos de la suite QuixBugs, sin añadir ningún tipo de contexto o información adicional. El propósito de este procedimiento es establecer una línea base para evaluar cómo maneja el modelo las tareas de corrección de errores de manera independiente. Esta evaluación inicial es fundamental, ya que permite comprender las capacidades nativas del LLM GPT-4 y sus limitaciones iniciales en el ámbito de la Reparación Automática de Programas (APR). A través de este análisis, se espera obtener una visión clara de las fortalezas intrínsecas del modelo y los desafíos que podrían requerir enfoques más dirigidos o la inclusión de información contextual en fases subsecuentes.

4.1 Prompt escogido

En esta etapa inicial del estudio, la selección del prompt es directa y no implica una investigación exhaustiva previa. Se planteará a ChatGPT una consulta básica sobre la existencia de errores en el código defectuoso proporcionado. Si el modelo identifica algún bug, se le solicitará que entregue una versión corregida del programa. A continuación, se presenta un ejemplo de cómo se utilizará en el primer programa de la suite seleccionada para la investigación, ilustrando el procedimiento estándar que se seguirá en esta fase del estudio.

User: ¿Hay algún bug en este programa? En caso afirmativo, devuélvemelo solucionado

```
1 public class BITCOUNT {
2     public static int bitcount(int n) {
3         int count = 0;
4         while (n != 0) {
5             n = (n ^ (n - 1));
6             count++;
7         }
8         return count;
9     }
10 }
```

4.2 Resultados y discusión

En la fase inicial de la evaluación de ChatGPT, los resultados obtenidos han ofrecido una perspectiva interesante sobre la capacidad del modelo para abordar la reparación automática de programas. Al analizar la tabla proporcionada (Tabla 4.1), se puede observar que **31 de los 40 programas** se han considerado solucionados, es decir, han pasado todos los tests unitarios que proporciona QuixBugs para cada uno de los programas. Sin embargo, se ha observado una variedad de comportamientos en la corrección de errores por parte de ChatGPT que merece la pena estudiar.

Una observación clave es la frecuencia con la que ChatGPT ha introducido correcciones innecesarias no solicitadas. En esta fase, en **15 de los 40 programas**, el modelo se ha tomado la libertad de hacer uso de ellas. Estas modificaciones, aunque no esenciales para la funcionalidad del código, han incluido cambios en el rendimiento, el nivel de código limpio¹, o incluso ajustes en la estructura que no son críticos para la resolución del bug. Este comportamiento sugiere que ChatGPT no solo se ha enfocado en la corrección del error específico, sino que también ha aplicado una optimización general del código según su aprendizaje previo. Esta tendencia podría interpretarse como un intento de mejorar el código más allá de la mera corrección del error, lo cual podría ser beneficioso en ciertos contextos de desarrollo, pero también podría haber introducido cambios no deseados que desvíen el código de las especificaciones originales del usuario, como ha ocurrido en varios de los programas de prueba (i.e. *powerset* o *max sublist sum*) donde estas correcciones han generado un nuevo bug en el código.

Por otro lado, los casos etiquetados como "Implementación de 0" (**9 de 40**) indican situaciones donde se dan indicios de que ChatGPT no ha identificado directamente el bug, sino que ha optado por reescribir o reemplazar segmentos del código basándose en que conoce el tipo de algoritmo involucrado, por ello, se les llama implementaciones de 0, ya que se consideran como si ChatGPT hubiese implementado el algoritmo a partir de un programa vacío, y no a partir del programa proporcionado. Aunque esta estrategia ha resultado en **5 de los 9 casos** en una implementación funcional y libre de errores del código, no se considera completamente positiva en el contexto de este estudio. Esto se debe a que el objetivo no es simplemente obtener un código funcional, sino entender y corregir específicamente el error presente según los criterios establecidos en la suite de pruebas. Este enfoque de "reimplementación" de ChatGPT, aunque ingenioso, no necesariamente ha ayudado a comprender la naturaleza del bug original o a validar la capacidad del modelo para detectar y reparar el bug de forma mínima y precisa.

En **4 ocasiones**, ChatGPT no solo no ha solucionado el bug original, sino que además ha introducido nuevos errores en el programa. Estos casos han mostrado varios patrones problemáticos: a veces, ChatGPT ha identificado correctamente el bug pero ha fallado en su corrección; en otras, ha solucionado el bug inicial, pero las modificaciones adicionales innecesarias que ha realizado al código han terminado introduciendo nuevos errores. También se han dado situaciones en las que ChatGPT no ha detectado el bug original y, al intentar hacer correcciones no solicitadas, ha generado problemas adicionales en el programa, ofreciendo así una solución incorrecta. Este patrón ha destacado la tendencia del modelo a realizar ajustes no esenciales que pueden comprometer la integridad del código.

¹código limpio: Este término se refiere a un conjunto de principios y prácticas para escribir software que sea fácil de entender y de mantener. Está basado en el libro Clean Code: A Handbook of Agile Software Craftsmanship por Robert C. Martin, que es ampliamente reconocido en la industria del software por promover técnicas que mejoran la legibilidad y la eficiencia del código.

Estos resultados subrayan la importancia de desarrollar prompts más específicos y contextualizados en fases posteriores del estudio, que guíen a ChatGPT a enfocarse en la tarea de corrección sin añadir o modificar aspectos del código que no están directamente relacionados con el bug. Además, resaltan la necesidad de ajustar y afinar las capacidades de respuesta del modelo para alinearlas más estrechamente con las expectativas y necesidades reales del desarrollo de software. La siguiente fase del estudio buscará abordar estas cuestiones mediante la implementación de prompts mejorados y pruebas adicionales que evalúen la efectividad de estas mejoras.

Programa	Solucionado	Correcciones innecesarias	Añade otro bug	Bug no encontrado	Implementación de 0
Bitcount	✓				
Breadth First Search	✓				
Bucketsort	✓	✓			
Depth First Search	✓				
Detect Cycle	✓				✓
Find First In Sorted	✓				✓
Find In Sorted	✓				
Flatten	✓	✓			
GCD	✓				
Get Factors	✓				✓
Hanoi	✓	✓			
Is Valid Parenthesization	✓				
Kheapsort	✓	✓			
Knapsack	✓				
Kth	✓	✓			
LCS Length	×				✓
Levenshtein	✓	✓			
LIS	✓	✓			
LCS	✓				
Max Sublist Sum	×	✓	✓		
Mergesort	✓	✓			
Minimum Spanning Tree	✓				✓
Next Palindrome	×				✓
Next Permutation	✓	✓			
Pascal	✓				
Possible Change	✓				
Powerset	×	✓	✓		
Quicksort	✓				✓
Reverse Linked List	×			✓	
RPN Eval	✓	✓			
Shortest Path Length	✓				
Shortest Path Lengths	✓				
Shortest Paths	×				✓
Shunting Yard	✓	✓			
Sieve	✓				
Sqrt	✓				
Subsequences	×	✓	✓		
To Base	✓				
Topological Ordering	×				✓
Wrap	×	✓	✓		
Σ Resultado final	31	15	4	1	9

Tabla 4.1: Resumen de resultados de la primera fase del estudio

4.3 Desarrollo de métricas para la evaluación

Dada la complejidad de evaluar la efectividad de ChatGPT en la detección y resolución de bugs, no es suficiente simplemente contar cuántos errores han pasado los tests unitarios de la suite estudiada. Por este motivo, se desarrollarán un conjunto de fórmulas que permitan penalizar las soluciones en las que ChatGPT ha efectuado cambios adicionales innecesarios o ha realizado "Implementaciones de 0". Esta metodología busca asegurar una evaluación más precisa del rendimiento de ChatGPT en tareas de reparación de errores.

En el presente estudio, se ha desarrollado un conjunto de métricas para evaluar de manera integral el desempeño de ChatGPT en la detección y resolución de bugs. Estas

métricas se dividen en tres componentes principales: **Efectividad**, **Precisión de Corrección** y **Evaluación Integral**. Cada una de estas fórmulas proporciona una perspectiva única sobre el rendimiento del modelo, permitiendo una evaluación detallada y matizada de sus capacidades.

La métrica de **Efectividad** se centra en la capacidad del modelo para resolver los bugs presentados. Se define como el número de bugs solucionados dividido por el número total de bugs, expresado como un porcentaje. Esta métrica nos ha permitido obtener una medida directa de la efectividad del modelo en términos de su capacidad de corrección de errores.

$$\text{Efectividad} = \frac{S}{B_T} \times 100$$

Donde:

- S representa el número de bugs solucionados.
- B_T es el número total de bugs.

La métrica de **Precisión de Corrección** evalúa la calidad de las soluciones proporcionadas por el modelo. Esta métrica considera las correcciones innecesarias y las implementaciones de cero, penalizando en consecuencia para reflejar la pureza de las soluciones. La fórmula para la Precisión de Corrección es:

$$\text{Precisión de Corrección} = 1 - \left(\frac{\frac{1}{5}S_{\text{correcciones_innecesarias}} + \frac{1}{2}S_{\text{implementación_cero}}}{S} \right) \times 100$$

Donde:

- $S_{\text{correcciones_innecesarias}}$ representa el número de bugs solucionados que incluyen correcciones innecesarias.
- $S_{\text{implementación_cero}}$ es el número de bugs solucionados con implementaciones de cero.
- S es el número de bugs solucionados.

Esta fórmula penaliza las soluciones que, aunque efectivas, incluyen correcciones redundantes o insuficientes, garantizando así que solo las soluciones más limpias y eficientes sean altamente valoradas. La penalización asignada a las correcciones innecesarias ($\frac{1}{5}$) y a las implementaciones de cero ($\frac{1}{2}$) refleja la gravedad relativa de estos problemas.

Se ha escogido un 20% ($\frac{1}{5}$) de penalización en el caso de las correcciones innecesarias, ya que el bug es encontrado y resuelto aunque cambie fragmentos en cuanto a código limpio, optimización o rendimiento. En el caso de las implementaciones de 0, se ha reducido el valor de la corrección a la mitad ($\frac{1}{2}$), ya que no tenemos constancia de que el modelo haya realmente detectado el bug, y que, aunque pasen las pruebas unitarias de la suite, existe la posibilidad de que solo haya escrito un código que conoce por su entrenamiento dado el nombre o naturaleza del algoritmo.

Estas penalizaciones se reflejan en que, en caso de que un bug detectado y resuelto de forma "perfecta", tendría un valor de 1.0, en el caso de tener correcciones innecesarias obtendría un valor de 0.8, y en caso de ser una implementación de 0, obtendría un valor

de 0.5. Estas penalizaciones, aplicadas a cada uno de los 40 programas, alejan negativamente el valor del numerador respecto al denominador, así obteniendo una precisión por debajo del 100 %

La **Evaluación Integral** combina las métricas de Efectividad y Precisión de Corrección para proporcionar una evaluación global del modelo. Esta métrica integrada ha permitido considerar tanto la eficiencia en la resolución de bugs como la calidad de las soluciones, ofreciendo una visión equilibrada del rendimiento del modelo.

Se ha definido la **Evaluación Integral** como una media ponderada de **Efectividad** y **Precisión de Corrección**, donde ambos se han evaluado como porcentajes.

La fórmula de **Evaluación Integral**, por tanto, es:

$$\text{Evaluación Integral} = w_R \cdot \text{Efectividad} + w_P \cdot \text{Precisión de Corrección}$$

Donde:

- w_E es el peso de la efectividad.
- w_P es el peso de la precisión de corrección.
- $w_R + w_P = 1$

4.3.1. Justificación de los Pesos en la Evaluación Integral

Para determinar la ponderación adecuada entre la Efectividad y la Precisión de Corrección, se ha optado inicialmente por utilizar $w_E = 0,5$ y $w_P = 0,5$. Este enfoque busca proporcionar una medida equitativa y balanceada entre ambas métricas, asegurando que tanto la cantidad de bugs resueltos como la calidad de las correcciones sean consideradas por igual.

El proceso ha comenzado con la observación de la efectividad en la primera fase, donde se han corregido 31 bugs. Dada su alta efectividad inicial, se ha deducido que las variaciones en la cantidad de bugs solucionados en las fases posteriores no serían sustancialmente diferentes. Por ello, se ha concluido que la precisión de corrección, que mide la calidad de las soluciones, sería un factor crucial para diferenciar las fases y tener unos resultados más visibles.

En el análisis, se han considerado dos enfoques para conseguir resultados diferenciados entre las fases:

- **Hacer más severas las penalizaciones:** Esta opción habría reducido la precisión y permitido disminuir su peso sin perder una clara diferenciación. Sin embargo, al revisar las penalizaciones actuales, se ha considerado que estaban bien fundamentadas y eran razonables, por lo que no se ha optado por esta vía.
- **Balancear los pesos en la media ponderada:** Este enfoque asegura una diferenciación clara entre las fases sin alterar la lógica de las penalizaciones. Al establecer pesos iguales, se garantiza que ni la efectividad ni la precisión dominen la evaluación de manera desproporcionada.

Se ha optado por la segunda opción, estableciendo pesos de 50/50, ya que ofrece un equilibrio visible y justo en los resultados. Esta decisión ha permitido una clara visualización de cómo ambas métricas influyen en la evaluación general. Aunque este balance se ha utilizado por motivos de estudio, es importante mencionar que los pesos podrían

ajustarse en el futuro, considerando más modelos o un enfoque diferente, para mejorar la precisión y efectividad del análisis.

4.4 Evaluación de Resultados

Dado:

$$\begin{aligned} S &= 31, \\ S_{\text{correcciones_innecesarias}} &= 15, \\ S_{\text{implementación_cero}} &= 9, \\ B_T &= 40 \end{aligned}$$

1. Efectividad:

$$\text{Efectividad} = \frac{31}{40} \times 100 = 77,5 \%$$

2. Precisión de Corrección:

$$\text{Precisión de Corrección} = \left(1 - \left(\frac{\frac{1}{5} \cdot 15 + \frac{1}{2} \cdot 9}{31} \right) \right) \times 100$$

$$\begin{aligned} \text{Precisión de Corrección} &= \left(1 - \left(\frac{3 + 4,5}{31} \right) \right) \times 100 \\ &= \left(1 - \left(\frac{7,5}{31} \right) \right) \times 100 \\ &= (1 - 0,2419) \times 100 \\ &= 75,81 \% \end{aligned}$$

$$\text{Evaluación Integral} = 0,5 \cdot 77,5 + 0,5 \cdot 75,81 = 38,75 + 37,905 = 76,655 \%$$

4.5 Resumen de Resultados

En la fase inicial de la evaluación de ChatGPT, los resultados obtenidos han ofrecido una perspectiva interesante sobre la capacidad del modelo para abordar la reparación automática de programas.

Observaciones:

- Se han solucionado 31 de 40 bugs.
- Se han registrado 15 correcciones innecesarias.
- Se han producido 9 implementaciones de 0.
- En 4 casos, se ha añadido un nuevo bug.
- Se ha obtenido una efectividad del 77.5 %.
- La precisión de corrección ha sido de un 75.81 %.
- La evaluación integral de la fase inicial ha sido de 76.655 %.

Conclusiones:

- ChatGPT tiende a hacer correcciones no solicitadas que pueden mejorar el código, pero también pueden desviar el código de las especificaciones originales.
- ChatGPT tiende a hacer reimplementaciones completas del código cuando no logra comprender la naturaleza del bug original.
- Se necesita desarrollar prompts más específicos y contextualizados para guiar a ChatGPT de manera más precisa.
- Una fórmula que penalice los cambios innecesarios y las reimplementaciones completas es esencial para una evaluación precisa del rendimiento de ChatGPT.
- Una métrica para evaluar en conjunto la efectividad y la precisión de las correcciones es necesaria para tener una visión conjunta de los resultados.

CAPÍTULO 5

Fase 2. Creación del primer prompt: Guías y restricciones

Basándose en las observaciones detalladas de la fase inicial, la segunda etapa de la investigación se diseñará para abordar específicamente las debilidades identificadas en el desempeño inicial de ChatGPT en la corrección de errores de software. Aprovechando los aprendizajes clave de la línea base creada, se desarrollará un prompt mejorado con el objetivo de superar las limitaciones observadas, proporcionando un contexto enriquecido y detallado que guíe a ChatGPT hacia soluciones más precisas y adaptadas a las necesidades específicas del estudio.

Este prompt mejorado se centrará en especificar claramente la región del código donde se localiza el error¹ y ofrecerá directrices claras que restrinjan la acción de ChatGPT a las áreas pertinentes, evitando así modificaciones innecesarias que puedan introducir nuevos errores o desviar el foco del problema original. Este enfoque tiene como finalidad facilitar que ChatGPT se enfoque en una corrección efectiva del bug sin realizar cambios adicionales que no contribuyan a resolver el problema específico identificado.

Este método mejorado pretende maximizar la eficiencia del modelo al proporcionar un contexto más rico y directrices específicas que deberían ayudar a ChatGPT a interpretar y actuar sobre los errores de manera más efectiva y con una perspectiva más adecuada para una herramienta APR. Con esta estrategia, se espera mejorar sustancialmente la precisión de las correcciones generadas.

5.1 Prompt escogido

Basandose en la primera fase del estudio se ha llegado a la conclusión de que se necesita dictar de alguna forma el cómo ChatGPT corrige estos programas defectuosos, por ello, se diseñará un prompt que intente abordar las debilidades encontradas. Para ello, se creará un prompt con la propia ayuda de ChatGPT:

Usuario:

Estoy haciendo una investigación del rendimiento de ChatGPT para la detección y resolución de bugs. Después de una fase en la que le he proporcionado programas con bugs intencionados, sin proporcionar un prompt de restricciones y principios a seguir, me he dado cuenta de que muchas veces en lugar de detectar el bug y resolverlo, proporciona una implementación de 0 porque conoce el algoritmo del progra-

¹Se proporcionará una región de código de extensión no menor a cinco líneas.

ma, o introduce correcciones para tener un código más limpio o de rendimiento que no se le han pedido, o cambia el formato del output y por tanto introduce nuevos bugs.

Por tanto, querría que me proporcionases un prompt para chatgpt, actúa como un ingeniero de prompts y dame un prompt con el que el LLM realmente se centre en la detección del bug y la resolución de este, con posibilidad de dar sugerencias, pero sin aplicarlas directamente en el código. Además, me gustaría que en el prompt hubiese un hueco donde especificar en qué línea o método se encuentra el bug, así restringiendo al chatgpt a la hora de modificar el código, y sobretodo, que no cambie el formato del output o el tipo de datos a no ser que sea estrictamente necesario

ChatGPT:

Prompt para ChatGPT enfocado en detección y corrección de bugs

Objetivo: Detectar y proponer soluciones para un bug específico en el código proporcionado sin cambiar la lógica general del programa, el formato de salida o los tipos de datos, a menos que sea imprescindible para resolver el bug.

Instrucciones:

1. Ubicación del Bug: Se me ha indicado que hay un problema en el código, específicamente en la línea [número de línea] o en el método [nombre del método]. Por favor, concéntrate en esta parte del código para detectar y entender el problema.
2. Descripción del Bug: Describe el problema detectado y explica por qué ocurre.
3. Sugerencia de Solución: Proporciona una o varias soluciones posibles para el bug. Detalla los pasos o el enfoque que recomiendas para corregirlo, pero no apliques los cambios directamente en el código.
4. Justificación: Si sugieres cambiar el formato de salida o los tipos de datos, proporciona una justificación clara de por qué estos cambios son necesarios para resolver el problema. Resumen: Al final, ofrece un breve resumen de la detección del problema y de tus recomendaciones para solucionarlo.

Código:

Usuario:

El prompt está bien, pero el ChatGPT SÍ debe devolver el código resuelto, siempre y cuando la parte cambiada sea en base al bug mencionado y en las líneas o método mencionado, lo que debe hacerse en formato sugerencia son cambios de código limpio, de rendimiento u optimización, pero quiero que el chatgpt actúe como una herramienta APR (Automatic Program Repair)

ChatGPT:

Prompt para ChatGPT como herramienta de Reparación Automática de Programas (APR)

Objetivo: Corregir un bug específico en el código proporcionado, enfocándote únicamente en las áreas indicadas, sin alterar la lógica general del programa, el formato de salida o los tipos de datos, a menos que sea imprescindible para la solución del bug.

Instrucciones:

1. Ubicación del Bug: El problema se ha identificado en la línea [número de línea] o en el método [nombre del método]. Concentra tus esfuerzos en esta sección para detectar y corregir el error.
 2. Descripción del Bug: Describe claramente el problema encontrado y explica por qué sucede.
 3. Corrección del Código: Modifica directamente el código para resolver el bug identificado, asegurándote de que los cambios se limitan a la parte del código especificada. Incluye comentarios en el código para explicar las modificaciones realizadas.
 4. Sugerencias Adicionales: Si observas potenciales mejoras de código limpio, de rendimiento u optimizaciones, enuméralas por separado al final del código corregido, sin aplicar estos cambios en el código proporcionado.
 5. Justificación: Si es necesario cambiar el formato de salida o los tipos de datos para resolver el bug, proporciona una justificación detallada de por qué estos cambios son necesarios.
 6. Resumen: Ofrece un breve resumen de la detección y corrección del problema, junto con cualquier sugerencia adicional para mejorar el código.
- Código:

Además, se ha pedido al ChatGPT que actúe como un ingeniero de prompts y evalúe el prompt obtenido en los aspectos apropiados.

ChatGPT:

Evaluación del Prompt:

- 1: Claridad del Problema: 9/10 El prompt ahora incluye un ejemplo específico de código con el método que contiene el bug, lo que facilita mucho el enfoque en la tarea específica.
- 2: Precisión Técnica: 10/10 La ubicación del problema está claramente identificada dentro de un método específico en Java, lo cual es excelente para enfocarse directamente en el problema.
- 3: Instrucciones Claras: 9/10 Las instrucciones son claras y concisas, ordenadas lógicamente para guiar al usuario a través del proceso de diagnóstico y corrección del error.
- 4: Contexto del Código: 10/10 Al proporcionar un fragmento de código real, el prompt es altamente efectivo para permitir una intervención directa en el código.
- 5: Objetivo Bien Definido: 10/10 Las limitaciones y expectativas de la corrección están bien establecidas, lo que ayuda a mantener la integridad del resto del programa.

Sugerencias para Mejorar el Prompt:

- Descripción Más Detallada del Bug: Aunque el método está claramente indicado, el prompt podría beneficiarse de una descripción más explícita del comportamiento erróneo del método `bitcount`. Por ejemplo, especificar si el conteo de bits está devolviendo un número incorrecto en ciertos casos, o si hay un error de tipo overflow o bucle infinito, etc.
- Proporcionar Casos de Prueba: Incluir ejemplos de valores de entrada para el método y los resultados esperados ayudaría a entender mejor el problema y verificar la corrección después de modificar el código.

- Explicaciones de los Cambios Sugeridos: Alentar la inclusión de comentarios más descriptivos en el código corregido para explicar por qué se realiza cada cambio específico.

Como ejemplo de prompt final para esta fase sería de la siguiente forma:

User:

Objetivo: Corregir un bug específico en el código proporcionado, enfocándote únicamente en las áreas indicadas, sin alterar la lógica general del programa, el formato de salida o los tipos de datos, a menos que sea imprescindible para la solución del bug.

Instrucciones:

1. Ubicación del Bug: El problema se ha identificado en

```

1  if (x < arr[mid]) {
2      return binsearch(arr, x, start, mid);
3  } else if (x > arr[mid]) {
4      return binsearch(arr, x, mid, end);
5  } else {
6      return mid;
7  }

```

Concentra tus esfuerzos en esta sección para detectar y corregir el error.

2. Descripción del Bug: Describe claramente el problema encontrado y explica por qué sucede.

3. Corrección del Código: Modifica directamente el código para resolver el bug identificado, asegurándote de que los cambios se limitan a la parte del código especificada. Incluye comentarios en el código para explicar las modificaciones realizadas.

4. Sugerencias Adicionales: Si observas potenciales mejoras de código limpio, de rendimiento u optimizaciones, enuméralas por separado al final del código corregido, sin aplicar estos cambios en el código proporcionado.

5. Justificación: Si es necesario cambiar el formato de salida o los tipos de datos para resolver el bug, proporciona una justificación detallada de por qué estos cambios son necesarios.

6. Resumen: Ofrece un breve resumen de la detección y corrección del problema, junto con cualquier sugerencia adicional para mejorar el código.

Código completo:

```

1  public class FIND_IN_SORTED {
2      public static int binsearch(int[] arr, int x, int start, int end) {
3          if (start == end) {
4              return -1;
5          }
6          int mid = start + (end - start) / 2; // check this is floor
7          // division
8          if (x < arr[mid]) {
9              return binsearch(arr, x, start, mid);
10             } else if (x > arr[mid]) {
11                 return binsearch(arr, x, mid, end);
12             } else {
13                 return mid;
14             }
15         }
16         public static int find_in_sorted(int[] arr, int x) {
17             return binsearch(arr, x, 0, arr.length);

```

```
18     }  
19 }
```

5.2 Resultados y discusión

En la segunda fase de la investigación, la implementación de un prompt mejorado y la especificación de una región de código para localizar el error han dado lugar a un conjunto interesante de resultados que merecen ser analizados en comparación con los obtenidos en la primera fase.

En esta fase, se han solucionado **28 de los 40 bugs**, una reducción respecto a los 31 bugs solucionados en la fase inicial. Este descenso podría atribuirse a la focalización específica en regiones de código y a las restricciones descritas en el prompt que, mientras ayudan a precisar la atención del modelo, pueden también limitar su capacidad de detectar y resolver errores al concentrarse demasiado en seguir estas guías. Sin embargo, tienen que tenerse en cuenta otros aspectos. Por ejemplo, es notable que el número de correcciones innecesarias se ha reducido significativamente, pasando de **15 en la primera fase a solo 5 en esta segunda fase**. Esto indica que la guía proporcionada por los prompts mejorados ha sido efectiva en canalizar la capacidad de ChatGPT para hacer modificaciones más relevantes y menos arbitrarias.

Además, la incidencia de ChatGPT añadiendo nuevos bugs al intentar solucionar los existentes o al hacer correcciones innecesarias ha disminuido drásticamente, de **cuatro casos en la primera fase a solo uno en esta fase**. Esto sugiere una mejora en la precisión de ChatGPT cuando se le proporciona una dirección más clara a través del prompt.

Por otro lado, el número de casos en los que ChatGPT no ha encontrado el bug ha aumentado de **uno a cinco**, lo cual podría reflejar un resultado negativo respecto a la primera fase. No obstante, este resultado puede verse de forma muy positiva si se considera que, a diferencia de la primera fase, cuando ChatGPT no ha encontrado el bug, no ha intentado buscar otro aspecto que solucionar o ha hecho refactorizaciones o cambios no funcionales para compensarlo. Como caso curioso, en el programa *Reverse Linked List*, ChatGPT ha afirmado que el programa estaba correcto y que no existía ningún bug, y a continuación, ha devuelto el código con el bug solucionado, sin hacer ningún comentario al respecto.

Finalmente, hemos reducido drásticamente las **"Implementaciones de 0"**, donde el modelo reescribe partes del código sin abordar directamente el bug, disminuyendo así su frecuencia de **9 a 3**, mostrando que el modelo ha comenzado a ajustar mejor sus respuestas a las demandas específicas del prompt sin recurrir tanto a reiniciar por completo la lógica del programa o reescribirlo basándose en su entrenamiento.

Estos resultados resaltan tanto las mejoras como los nuevos desafíos surgidos de la implementación de prompts más específicos y contextualizados. Aunque la guía más precisa ha mejorado la relevancia y exactitud de las correcciones de ChatGPT, también ha evidenciado ciertas restricciones en la capacidad del modelo para solucionar algunos de los bugs. Estos hallazgos subrayan la necesidad de continuar refinando la estrategia de prompts y de explorar equilibrios adecuados entre la especificidad y la flexibilidad en la guía proporcionada al modelo para optimizar su rendimiento en la detección y resolución de bugs.

Programa	Solucionado	Correcciones innecesarias	Añade otro bug	Bug no encontrado	Implementación de 0
Bitcount	✓				
Breadth First Search	✓				
Bucketsort	✓				
Depth First Search	✓				
Detect Cycle	✓				
Find First In Sorted	✓				
Find In Sorted	✓				
Flatten	✓				
GCD	✓				
Get Factors	×				
Hanoi	×				
Is Valid Parenthesization	✓				
Kheapsort	✓				
Knapsack	✓				
Kth	✓				
LCS Length	×			✓	
Levenshtein	×			✓	
LIS	×			✓	
LCS	✓				
Max Sublist Sum	×				✓
Mergesort	✓	✓			
Minimum Spanning Tree	✓				
Next Palindrome	×	✓	✓		
Next Permutation	✓	✓			
Pascal	✓				
Possible Change	✓				
Powerset	×				
Quicksort	✓	✓			
Reverse Linked List	✓			✓	
RPN Eval	✓				
Shortest Path Length	×				
Shortest Path Lengths	✓				✓
Shortest Paths	✓				
Shunting Yard	×			✓	
Sieve	✓	✓			
Sqrt	✓				
Subsequences	×				
To Base	✓				
Topological Ordering	×				✓
Wrap	×				
Σ Resultado final	28	5	1	5	3

Tabla 5.1: Resumen de resultados de la segunda fase del estudio

5.3 Evaluación de Resultados

Dado:

$$\begin{aligned}
 S &= 28, \\
 S_{\text{correcciones_innecesarias}} &= 5, \\
 S_{\text{implementación_cero}} &= 3, \\
 B_T &= 40
 \end{aligned}$$

1. Efectividad:

$$\text{Efectividad} = \frac{28}{40} \times 100 = 70\%$$

2. Precisión de Corrección:

$$\text{Precisión de Corrección} = \left(1 - \left(\frac{\frac{1}{5} \cdot 5 + \frac{1}{2} \cdot 3}{28} \right) \right) \times 100$$

$$\begin{aligned}\text{Precisión de Corrección} &= \left(1 - \left(\frac{1 + 1,5}{28}\right)\right) \times 100 \\ &= \left(1 - \left(\frac{2,5}{28}\right)\right) \times 100 \\ &= (1 - 0,0893) \times 100 \\ &= 91,07\%\end{aligned}$$

$$\text{Evaluación Integral} = 0,5 \cdot 70 + 0,5 \cdot 91,07 = 35 + 45,535 = 80,535\%$$

5.4 Resumen de Resultados

En la segunda fase de la investigación, la implementación de un prompt mejorado y la especificación de una región de código para localizar el error han dado lugar a un conjunto interesante de resultados que merecen ser analizados en comparación con los obtenidos en la primera fase.

Observaciones:

- Se han solucionado 28 de 40 bugs.
- El número de correcciones innecesarias se ha reducido de 15 a 5.
- La incidencia de nuevos bugs ha disminuido de 4 a 1.
- Los casos de "Implementaciones de 0" se han reducido de 9 a 3.
- Ha aumentado el número de bugs no encontrados de 1 a 5.
- Se ha obtenido una efectividad del 70 %.
- La precisión de corrección ha sido de un 91.07 %.
- La evaluación integral de la segunda fase ha sido de 80.535% (\approx 4% superior respecto a la primera fase).

Conclusiones:

- La guía proporcionada por los prompts mejorados ha sido efectiva en reducir las correcciones innecesarias y los nuevos bugs, así mejorando la evaluación integral de la fase aún habiendo corregido una menor cantidad de bugs.
- La especificación de regiones de código para localizar el error puede limitar la capacidad del modelo para detectar y resolver errores.
- Se necesita continuar refinando la estrategia de prompts para equilibrar la especificidad y la flexibilidad en la guía proporcionada al modelo.
- Un enfoque de prompting progresivo y adaptable es crucial para el perfeccionamiento continuo del proceso de reparación automática de programas.

CAPÍTULO 6

Fase 3. Mejora de prompt: Pruebas unitarias

En la tercera fase de esta investigación, se enriquecerá aún más el prompt incorporando tests unitarios que fallan debido al bug presente en el código. Esta fase se fundamenta en la hipótesis de que proporcionar a ChatGPT ejemplos específicos de cómo falla el código bajo ciertas condiciones mejorará significativamente su capacidad para proponer la corrección adecuada.

La fase anterior, donde se guió a ChatGPT mediante prompts específicos y se delimitó una región del código para localizar el error, mostró una reducción en la cantidad de correcciones innecesarias y "Implementaciones de 0". Sin embargo, también reveló que una mayor precisión en la guía proporcionada puede limitar la capacidad del modelo para detectar y resolver bugs fuera de los parámetros definidos. Esto subraya la importancia de encontrar un equilibrio entre especificidad y flexibilidad en los prompts.

La inclusión de resultados de tests unitarios fallidos pretende aportar un contexto aún más detallado y preciso, facilitando que ChatGPT entienda no solo dónde se encuentra el error, sino también cómo afecta al funcionamiento del programa. Esta información adicional podría permitir al modelo generar soluciones más acertadas y específicas, alineándose mejor con los criterios establecidos en la suite de pruebas.

Al incorporar tests unitarios, se espera evaluar si esta estrategia afecta positivamente la precisión y la efectividad de las soluciones generadas por ChatGPT. Esta fase busca abordar las limitaciones observadas anteriormente, proporcionando un enfoque más robusto para la detección y corrección de errores, y avanzando hacia una metodología que optimice el rendimiento de ChatGPT en tareas de reparación automática de programas.

6.1 Prompt escogido

En esta sección simplemente se transformará el prompt de la fase 2 para incluir los tests unitarios fallidos del programa en el lugar donde se indicaba la ubicación del bug.

Para ello, se sustituirá el primer punto de las instrucciones proporcionadas de la siguiente forma:

Usuario:

Prompt para ChatGPT enfocado en detección y corrección de bugs

Objetivo: Corregir un bug específico en el código proporcionado, enfocándote únicamente en las áreas indicadas, sin alterar la lógica general del programa, el formato de salida o los tipos de datos, a menos que sea imprescindible para la solución del bug.

Instrucciones:

1. Tests Unitarios Fallidos: Para más contexto sobre el bug y facilitar su detección, se han proporcionado los tests unitarios del programa (únicamente los que fallan). Revisa estos tests para detectar y corregir el error en el código.

****tests fallidos****

2. Descripción del Bug: Describe claramente el problema encontrado y explica por qué sucede.

3. Corrección del Código: Modifica directamente el código para resolver el bug identificado, asegurándote de que los cambios se limitan a la parte del código especificada. Incluye comentarios en el código para explicar las modificaciones realizadas.

4. Sugerencias Adicionales: Si observas potenciales mejoras de código limpio, de rendimiento u optimizaciones, enuméralas por separado al final del código corregido, sin aplicar estos cambios en el código proporcionado.

5. Justificación: Si es necesario cambiar el formato de salida o los tipos de datos para resolver el bug, proporciona una justificación detallada de por qué estos cambios son necesarios.

6. Resumen: Ofrece un breve resumen de la detección y corrección del problema, junto con cualquier sugerencia adicional para mejorar el código.

Código:

****código del programa****

Como ejemplo de prompt final para esta fase sería de la siguiente forma:

User:

Objetivo: Corregir un bug específico en el código proporcionado, enfocándote únicamente en las áreas indicadas, sin alterar la lógica general del programa, el formato de salida o los tipos de datos, a menos que sea imprescindible para la solución del bug.

Instrucciones:

1. Tests Unitarios Fallidos: Para más contexto sobre el bug y facilitar su detección, se han proporcionado los tests unitarios del programa (únicamente los que fallan). Revisa estos tests para detectar y corregir el error en el código.

```

1 @org.junit.Test(timeout = 3000)
2 public void test_1() throws java.lang.Exception {
3     java.util.ArrayList result = java_programs.PASCAL.pascal((int)2);
4     String resultFormatted = java_testcases.junit.QuixFixOracleHelper
5         .format(result, true);
6     org.junit.Assert.assertEquals("[[1],[1,1]]", resultFormatted);
7 }
8 @org.junit.Test(timeout = 3000)
9 public void test_2() throws java.lang.Exception {
10    java.util.ArrayList result = java_programs.PASCAL.pascal((int)3);
11    String resultFormatted = java_testcases.junit.QuixFixOracleHelper
12        .format(result, true);

```

```

12     org.junit.Assert.assertEquals("[[1],[1,1],[1,2,1]]", resultFormatted
13     );
14 }
15 @org.junit.Test(timeout = 3000)
16 public void test_3() throws java.lang.Exception {
17     java.util.ArrayList result = java_programs.PASCAL.pascal((int)4);
18     String resultFormatted = java_testcases.junit.QuixFixOracleHelper.
19         format(result, true);
20     org.junit.Assert.assertEquals("[[1],[1,1],[1,2,1],[1,3,3,1]]",
21         resultFormatted);
22 }
23 @org.junit.Test(timeout = 3000)
24 public void test_4() throws java.lang.Exception {
25     java.util.ArrayList result = java_programs.PASCAL.pascal((int)5);
26     String resultFormatted = java_testcases.junit.QuixFixOracleHelper.
27         format(result, true);
28     org.junit.Assert.assertEquals("
29         [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]", resultFormatted);
30 }

```

2. Descripción del Bug: Describe claramente el problema encontrado y explica por qué sucede.

3. Corrección del Código: Modifica directamente el código para resolver el bug identificado, asegurándote de que los cambios se limitan a la parte del código especificada. Incluye comentarios en el código para explicar las modificaciones realizadas.

4. Sugerencias Adicionales: Si observas potenciales mejoras de código limpio, de rendimiento u optimizaciones, enuméralas por separado al final del código corregido, sin aplicar estos cambios en el código proporcionado.

5. Justificación: Si es necesario cambiar el formato de salida o los tipos de datos para resolver el bug, proporciona una justificación detallada de por qué estos cambios son necesarios.

6. Resumen: Ofrece un breve resumen de la detección y corrección del problema, junto con cualquier sugerencia adicional para mejorar el código.

Código completo:

```

1 public class PASCAL {
2     public static ArrayList<ArrayList<Integer>> pascal(int n) {
3         ArrayList<ArrayList<Integer>> rows = new ArrayList<ArrayList<
4             Integer>>();
5         ArrayList<Integer> init = new ArrayList<Integer>();
6         init.add(1);
7         rows.add(init);
8
9         for (int r=1; r<n; r++) {
10            ArrayList<Integer> row = new ArrayList<Integer>();
11            for (int c=0; c<r; c++) {
12                int upleft, upright;
13                if (c > 0) {
14                    upleft = rows.get(r-1).get(c-1);
15                } else {
16                    upleft = 0;
17                }
18                if (c < r) {
19                    upright = rows.get(r-1).get(c);
20                } else {
21                    upright = 0;
22                }

```

```
22         row.add(upleft+upright);
23     }
24     rows.add(row);
25 }
26
27     return rows;
28 }
29 }
```

6.2 Resultados y Discusión

En la tercera fase de la investigación, se ha enriquecido el prompt de ChatGPT al incluir los tests unitarios fallidos en lugar de especificar la región de código donde se encuentra el bug. Esta estrategia ha proporcionado un contexto más detallado sobre las condiciones bajo las cuales el código falla, permitiendo evaluar si esta información adicional mejora la capacidad del modelo para corregir los errores.

Los resultados obtenidos en esta fase han mostrado una mejora notable en la capacidad de ChatGPT para solucionar bugs. En total, se han solucionado **32 de los 40 bugs**, superando los 28 bugs corregidos en la fase anterior y los 31 de la fase inicial. Este incremento sugiere que proporcionar ejemplos específicos de fallos a través de los tests unitarios puede ser más eficaz que simplemente delimitar una región del código problemática, ya que ayuda a ChatGPT a entender mejor el contexto y la naturaleza del error.

La incidencia de correcciones innecesarias se ha reducido aún más, con solo **dos casos observados** en esta fase. Es importante resaltar que en ninguno de estos casos las correcciones adicionales han introducido nuevos bugs, lo que indica una mejora significativa en la precisión de las modificaciones realizadas por ChatGPT cuando se le proporciona un contexto más detallado.

Sin embargo, persisten ciertos desafíos. En **tres de los ocho casos no solucionados**, ChatGPT ha introducido nuevos bugs al intentar corregir los errores existentes. Este problema podría reflejar una sobrecorrección o una interpretación errónea de los tests unitarios, lo que sugiere la necesidad de seguir refinando la metodología de prompting. Además, hemos encontrado **un caso** en el que ChatGPT no ha encontrado el bug original y, en su lugar, ha intentado solucionar otro aspecto del código, lo cual evidencia que el modelo aún puede ser desviado por información confusa o incompleta.

Asimismo, en **cuatro de los ocho casos restantes**, ChatGPT ha identificado correctamente el bug pero no ha logrado corregirlo de manera efectiva. Esto subraya la complejidad de algunos errores y la posibilidad de que incluso con información detallada, ciertas correcciones sigan siendo desafiantes para el modelo. Este aspecto podría corregirse proporcionando la intención del programa o una descripción más detallada del problema, pero eso alejaría la investigación del propósito de utilizarlo como una herramienta APR.

Un caso particular a destacar fue cuando ChatGPT ha optado por realizar una "Implementación de 0" en un programa. En esta instancia, no solo ha reescrito el código, sino que también lo ha hecho de manera incorrecta, lo que ha contribuido a los resultados negativos. Este fenómeno, aunque menos frecuente que en las fases anteriores, indica que el modelo todavía puede recurrir a estrategias de reimplementación completa cuando se enfrenta a errores complejos que no puede resolver fácilmente.

Estos resultados destacan tanto las mejoras alcanzadas como los desafíos persistentes en la capacidad de ChatGPT para la reparación de programas. La inclusión de tests

unitarios fallidos como parte del prompt ha demostrado ser una estrategia efectiva para mejorar la precisión de las correcciones, pero también revela áreas donde se requiere un refinamiento adicional. Continuar desarrollando y perfeccionando los prompts, así como explorar métodos adicionales para guiar al modelo, será crucial para optimizar su rendimiento en tareas de detección y corrección de errores. Este análisis reafirma la importancia de un enfoque iterativo y adaptable en la aplicación de herramientas de inteligencia artificial para la reparación de software, donde cada fase proporciona insights valiosos para la mejora continua del proceso. Todos estos insights serán empleados en la última fase de la investigación, donde se aplicarán para crear un GPT personalizado.

Programa	Solucionado	Correcciones innecesarias	Añade otro bug	Bug no encontrado	Implementación de 0
Bitcount	✓				
Breadth First Search	✓	✓			
Bucketsort	✓				
Depth First Search	✓				
Detect Cycle	✓				
Find First In Sorted	×				
Find In Sorted	✓				
Flatten	✓				
GCD	✓				
Get Factors	✓				
Hanoi	✓				
Is Valid Parenthesization	✓				
Kheapsort	✓				
Knapsack	✓				
Kth	✓				
LCS Length	×				✓
Levenshtein	✓				
LIS	✓				
LCS	✓				
Max Sublist Sum	×		✓		
Mergesort	✓				
Minimum Spanning Tree	✓				
Next Palindrome	×			✓	
Next Permutation	✓	✓			
Pascal	✓				
Possible Change	×				
Powerset	✓				
Quicksort	✓				
Reverse Linked List	✓				
RPN Eval	✓				
Shortest Path Length	✓				
Shortest Path Lengths	✓				
Shortest Paths	✓				
Shunting Yard	✓				
Sieve	✓				
Sqrt	✓				
Subsequences	×		✓		
To Base	✓				
Topological Ordering	×				
Wrap	×		✓		
Σ Resultado final	32	2	3	1	1

Tabla 6.1: Resumen de resultados de la tercera fase del estudio

6.3 Evaluación de Resultados

Dado:

$$S = 32,$$

$$S_{\text{correcciones_innecesarias}} = 2,$$

$$S_{\text{implementación_cero}} = 1,$$

$$B_T = 40$$

1. **Efectividad:**

$$\text{Efectividad} = \frac{32}{40} \times 100 = 80\%$$

2. Precisión de Corrección:

$$\text{Precisión de Corrección} = \left(1 - \left(\frac{\frac{1}{5} \cdot 2 + \frac{1}{2} \cdot 0}{32} \right) \right) \times 100$$

$$\text{Precisión de Corrección} = \left(1 - \left(\frac{0,4}{32} \right) \right) \times 100 = (1 - 0,0125) \times 100 = 98,75 \%$$

3. Evaluación Integral:

$$\text{Evaluación Integral} = 0,5 \cdot 80 + 0,5 \cdot 98,75 = 40 + 49,375 = 89,375 \%$$

6.4 Resumen de Resultados

En la tercera fase de la investigación, se ha enriquecido el prompt de ChatGPT al incluir los tests unitarios fallidos en lugar de especificar la región de código donde se encuentra el bug. Esta estrategia ha proporcionado un contexto más detallado sobre las condiciones bajo las cuales el código falla, permitiendo evaluar si esta información adicional mejora la capacidad del modelo para corregir los errores.

Observaciones:

- Se han solucionado 32 de 40 bugs.
- El número de correcciones innecesarias se ha reducido a 2.
- En tres casos, ChatGPT ha introducido nuevos bugs al intentar corregir los errores existentes.
- En un caso, ChatGPT no ha encontrado el bug original y ha tratado de solucionar otro aspecto del código.
- En cuatro casos, ChatGPT ha identificado el bug pero no ha logrado corregirlo de manera efectiva.
- Se ha registrado una "Implementación de 0" en un programa, donde el código reescrito es incorrecto.
- Se ha obtenido una efectividad del 80 %.
- La precisión de corrección ha obtenido un 97.19 %.
- La evaluación integral de la fase 3 ha sido de 89.375 % (≈ 9 % superior a la segunda fase).

Conclusiones:

- Proporcionar ejemplos específicos de fallos a través de los tests unitarios es severamente más eficaz que delimitar una región del código problemática, tanto en efectividad como en precisión.
- La inclusión de tests unitarios fallidos ha demostrado mejorar la precisión de las correcciones realizadas por ChatGPT.
- Persisten desafíos en la capacidad del modelo para evitar sobrecorrecciones e interpretaciones erróneas de los tests unitarios.
- Se necesita continuar desarrollando y perfeccionando los prompts para optimizar el rendimiento de ChatGPT en tareas de detección y corrección de errores.

CAPÍTULO 7

Fase 4. GPT Builder: Primer prototipo

Basándose en los insights y resultados obtenidos en las fases anteriores, la siguiente etapa de la investigación se centrará en el desarrollo de un GPT personalizado, específicamente afinado para la reparación de bugs. En las fases previas, hemos observado cómo la introducción de información adicional y la especificación de contextos precisos han mejorado significativamente el rendimiento de ChatGPT en la detección y corrección de errores. En esta fase, utilizaremos las mejores estrategias y enfoques de prompting descubiertos durante los experimentos para crear un modelo más adaptado y eficiente.

El objetivo principal de esta fase es evaluar si la personalización del modelo, basada en los datos y técnicas más efectivas de las fases anteriores, puede ofrecer mejoras significativas en la capacidad de ChatGPT para la reparación automática de programas (APR). La evaluación final de este GPT personalizado comparará su rendimiento con las versiones estándar de ChatGPT utilizadas en las fases anteriores, determinando la eficacia de las personalizaciones implementadas. Este enfoque permitirá explorar el potencial completo de un modelo de lenguaje avanzado cuando se optimiza específicamente para una tarea crítica como la APR, proporcionando resultados valiosos para futuras aplicaciones y desarrollos en el campo de la inteligencia artificial y el desarrollo de software.

7.1 Creación del GPT: Uso ordinario de la herramienta

En esta sección se utilizará la herramienta GPT Builder de la forma en que la usaría un usuario normal de la aplicación, simplemente explicándole al GPT Builder qué tipo de GPT quiere y paso a paso y respondiendo a las preguntas de este, crear el GPT especializado en la detección y resolución de bugs.

Para ello se le dará un breve contexto del estudio, y posteriormente unas instrucciones de como se debe comportar este GPT.

User:

Quiero crear un GPT personalizado con el GPT Builder en base a un estudio de prompting progresivo que he hecho para evaluar la capacidad de ChatGPT como herramienta de detección y resolución de bugs de forma automática, es decir herramienta APR (Automatic Program Repair)

Los resultados del estudio indicaron que un prompt con restricciones y guías sobre el proceso crearon unos resultados que se centraban en la detección y resolución de

los bugs, en lugar de aplicar cambios de todo tipo al programa con tal de optimizarlo, refactorizarlo, etc.

Más tarde, también se vio que, indicar una región del código podía ayudar a enfocar el modelo, sobretodo cuando el programa era muy grande.

Por último y más importante, se vio que darle al modelo los tests unitarios que habían fallado le proporcionaban un contexto que dio lugar a los mejores resultados en cuanto a detección y resolución de bugs.

Por tanto, quiero que me hagas un GPT especializado que tenga en cuenta estos resultados, en primer lugar, que cuando el usuario mande un programa sin ningún contexto, el GPT ya tenga las directrices de que puede cambiar el código para resolver el bug, pero las mejoras en cuanto a refactorización, estructura, optimización, código limpio, etc. tienen que ser a modo de sugerencias. Si el GPT no encuentra el bug o no está seguro de algún aspecto del programa, le pedirá más información al usuario, ya sean los tests unitarios, la región del código que se cree que falla, la excepción que salta, etc.

Como se puede observar, el GPT Builder guía al usuario a la creación del GPT personalizado, preguntándole sobre cosas a enfatizar y evitar, así como el registro que prefiere en que le hable el modelo.

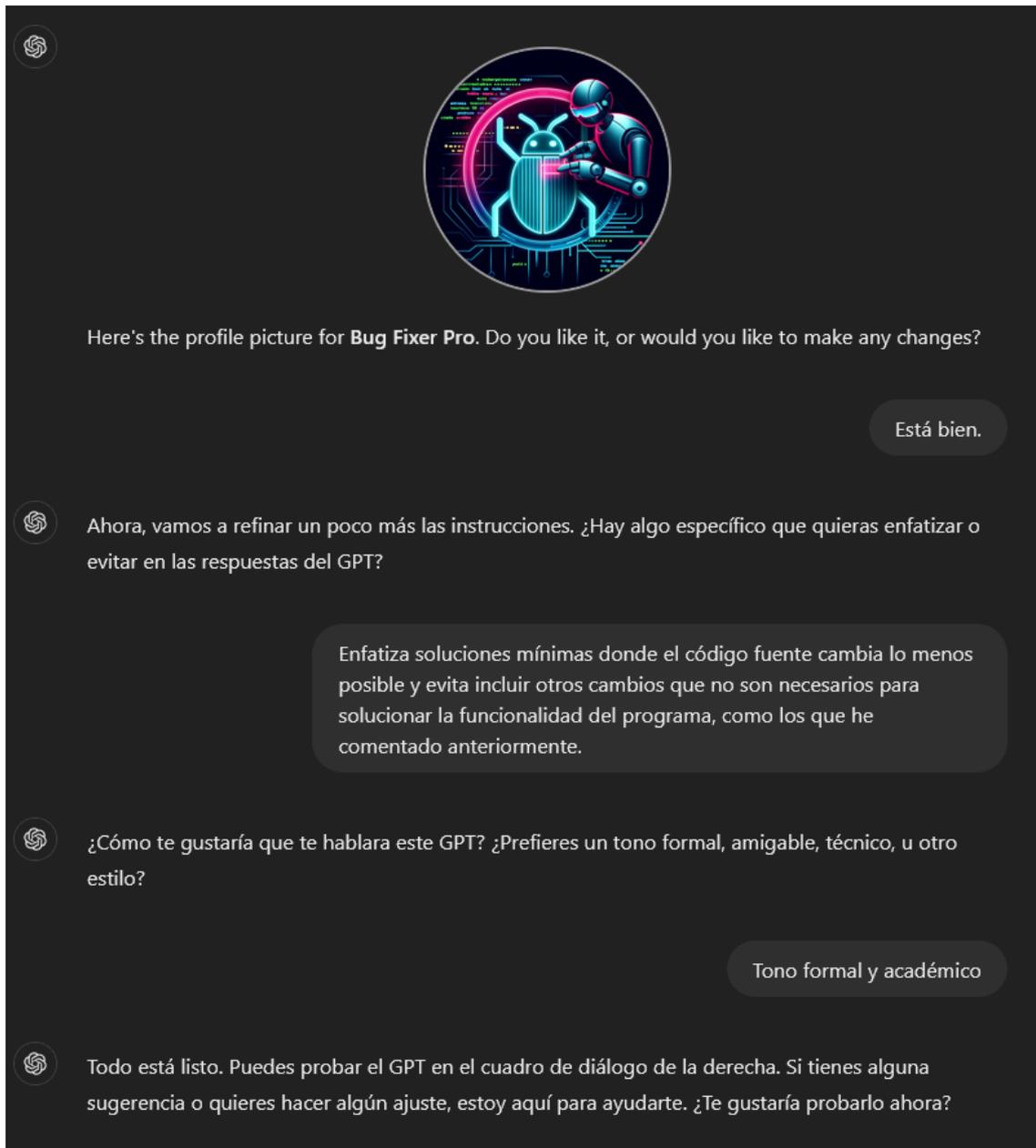


Figura 7.1: Respuesta de GPT Builder a la petición del primer prototipo de GPT especializado

Una vez creado, existe la posibilidad de acceder a la configuración que ha generado el GPT Builder en base a las peticiones del usuario, pudiendo así observar de qué forma ha instruido al nuevo modelo para comportarse de acuerdo a las necesidades del estudio.

Al entrar en la pestaña de configuración, se encontraron las siguientes opciones:

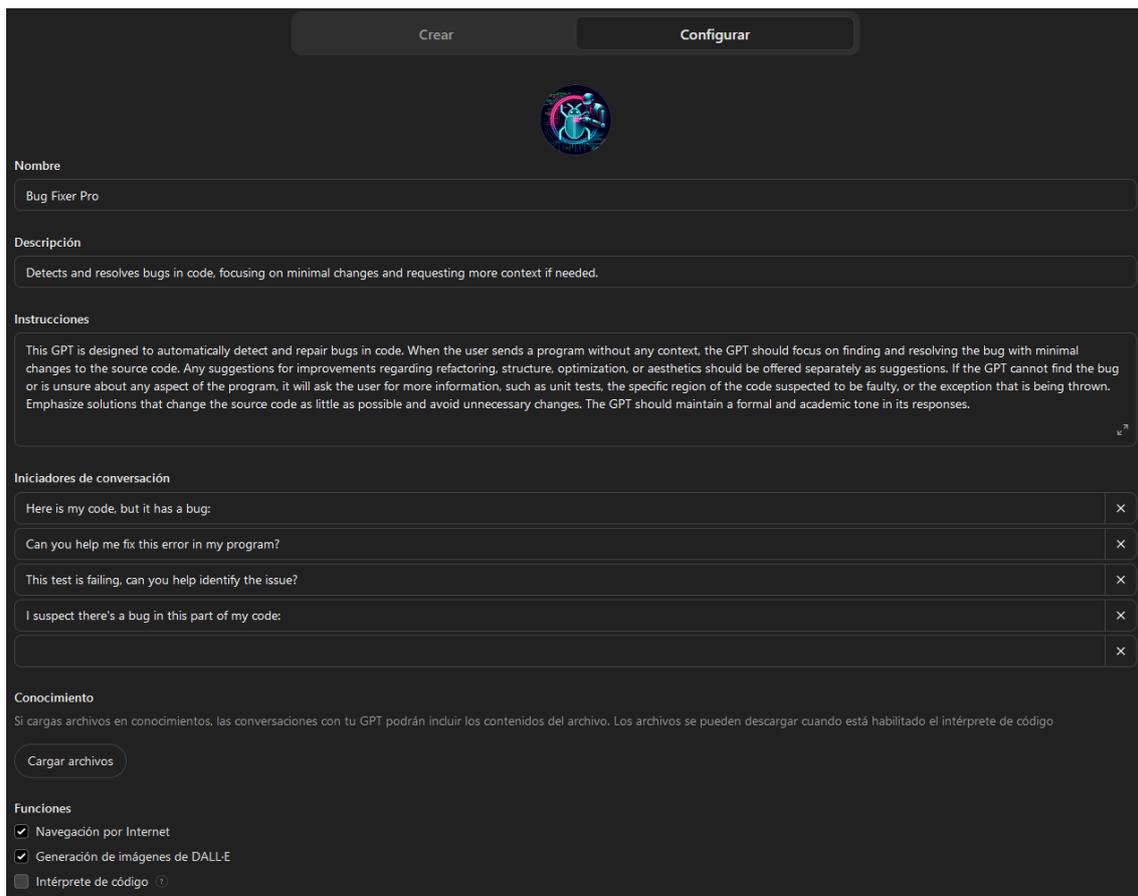


Figura 7.2: Configuración del primer prototipo de GPT especializado creada a partir de nuestra petición

Como se puede observar, las instrucciones generadas por el modelo creador no son muy extensas ni detalladas. Además, estas instrucciones no siguen una estructura ordenada ni priorizada y, por razones desconocidas, están redactadas en inglés.

Para poder estudiar este nuevo modelo, se procederá a la creación y publicación de este (de momento se estaba trabajando en un modo de previsualización). Como se puede observar, se puede mantener el GPT personalizado con distinta configuración de privacidad, ya sea personal, privado o público.

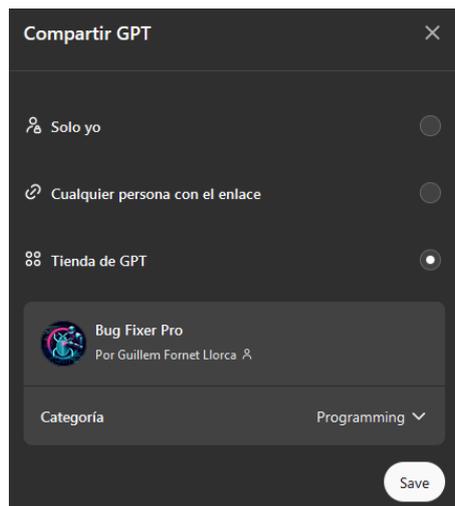


Figura 7.3: Paso final de la creación del primer prototipo de GPT con la herramienta GPT Builder

7.2 Método de estudio

Para evaluar la efectividad del GPT personalizado, se modificará ligeramente la metodología utilizada en las fases anteriores. En esta ocasión, no se realizará una única petición por conversación del modelo. Se comenzará enviando el código sin ningún tipo de contexto o instrucción adicional y, en caso de no obtener un resultado positivo, se guiará al GPT de la manera en que lo haría un usuario promedio. Ya que se ha instruido el modelo *a priori*, esto permite que el usuario no necesite recurrir a técnicas especializadas de prompting, como se ha hecho en fases anteriores, facilitando así su uso para usuarios con diferentes niveles de experiencia.

El objetivo de los GPT personalizados es optimizar el modelo para que pueda detectar y resolver errores de manera autónoma y eficiente, alineando sus respuestas con las expectativas de los usuarios sin necesidad de instrucciones detalladas o técnicas avanzadas de prompting. Este enfoque asegura que el modelo pueda ser utilizado de manera efectiva por cualquier usuario, independientemente de su experiencia en debugging o en el uso de modelos de lenguaje avanzados.

Para considerar un resultado como aceptable, se establecen tres criterios: el modelo debe detectar explícitamente el bug, el programa debe pasar los tests de la suite de bugs y el código no debe presentar cambios significativos en comparación con el código fuente original. Por último, se realizarán un máximo de 5 peticiones, si entonces no ha conseguido solucionar el bug, se dará por no solucionado. Estos criterios aseguran una detección y resolución verídica y precisa, garantizando que el modelo no solo corrige el error, sino que también mantiene la integridad y funcionalidad del código original.

A continuación, se presentará una demostración real con el primer bug de la suite que ha sido modificada para ilustrar la metodología, mostrando el uso del GPT personalizado en el estudio de su rendimiento.¹

¹El texto de las respuestas se ha traducido del inglés ya que el GPT Builder ha configurado el modelo con instrucciones en inglés pese a pedirle todo en castellano (por motivos de estudio se ha dejado en las mismas condiciones en que ha sido creado)

User:

```
1 public class BITCOUNT {
2     public static int bitcount(int n) {
3         int count = 0;
4         while (n != 0) {
5             n = n & (n - 1);
6             count++;
7         }
8         return count;
9     }
10 }
```

Bug Fixer Pro:**Localización del Bug**

El bug se localiza en el método bitcount de la clase BITCOUNT

Descripción del Bug

El método bitcount está destinado a contar el número de bits de 1 en la representación binaria de un entero n. La implementación actual intenta hacer esto realizando repetidamente la operación $n = (n \oplus (n - 1))$, lo cual no aísla ni cuenta correctamente los bits. Esto resulta en un bucle infinito cuando n tiene un valor diferente de cero.

Corrección del Código

Para corregir el error, necesitamos usar la operación estándar de AND bit a bit $n \& (n - 1)$, que apaga efectivamente el bit más bajo de n y aumenta el conteo. Aquí está el código corregido:

```
1 public class BITCOUNT {
2     public static int bitcount(int n) {
3         int count = 0;
4         while (n != 0) {
5             n = n & (n - 1);
6             count++;
7         }
8         return count;
9     }
10 }
```

Justificación

La operación AND bit a bit $n \& (n - 1)$ es un método bien conocido para contar el número de bits establecidos en la representación binaria de un entero. Funciona apagando el bit más bajo establecido de n en cada iteración, asegurando que el bucle finalmente termine cuando todos los bits estén apagados (es decir, cuando n se convierta en 0).

Resumen

Ubicación del Error: El método bitcount en la clase BITCOUNT. Problema: El método original resultaba en un bucle infinito porque la operación XOR no aislaba ni contaba correctamente los bits. Solución: Se reemplazó la operación XOR con la operación AND bit a bit $n \& (n - 1)$ para apagar correctamente el bit más bajo establecido en cada iteración. Resultado: El método corregido ahora cuenta con precisión el número de bits de 1 en la representación binaria de un entero.

Sugerencias Adicionales

Pruebas Unitarias: Agregar pruebas unitarias para asegurar la corrección del método bitcount en varios tipos de entradas, incluyendo casos límite. Optimización: No se necesitan optimizaciones adicionales ya que el método corregido ya es eficiente. ¿Le gustaría que proporcionara algunas pruebas unitarias para validar este método?

A modo de ejemplo, se considerará que el ChatGPT no ha corregido correctamente el bug del programa

User:

No funciona, los siguientes tests siguen fallando.

```
1  @org.junit.Test(timeout = 3000)
2  public void test_0() throws java.lang.Exception {
3      int result = java_programs.BITCOUNT.bitcount((int)127);
4      org.junit.Assert.assertEquals( (int) 7, result);
5  }
6
7  Expected: 7
8  Actual: 5
9
10 @org.junit.Test(timeout = 3000)
11 public void test_1() throws java.lang.Exception {
12     int result = java_programs.BITCOUNT.bitcount((int)128);
13     org.junit.Assert.assertEquals( (int) 1, result);
14 }
15
16 Expected: 1
17 Actual: 2
```

Bug Fixer Pro:

Gracias por proporcionar los casos de prueba. Dado que los resultados de las pruebas no coinciden con los valores esperados, vamos a re-evaluar la función 'bitcount' para asegurarnos de que cuente con precisión el número de bits establecidos.

Análisis

La función corregida debería contar con precisión el número de bits de 1 en la representación binaria del entero. Vamos a reconsiderar el enfoque e implementar un método alternativo usando operaciones bit a bit.

Código Actualizado

En lugar de apagar el bit más bajo establecido, iteraremos a través de cada bit del entero y contaremos el número de bits establecidos directamente:

```
1 public class BITCOUNT {
2     public static int bitcount(int n) {
3         int count = 0;
4         while (n != 0) {
5             if ((n & 1) == 1) {
6                 count++;
7             }
8             n >>= 1;
9         }
10        return count;
11    }
12 }
```

Explicación

- La condición `'(n & 1) == 1'` verifica si el bit menos significativo (LSB) de 'n' está establecido (es decir, igual a 1).
- La operación de desplazamiento a la derecha `'n >>= 1'` desplaza todos los bits de 'n' a la derecha en una posición, descartando efectivamente el LSB.
- El bucle continúa hasta que se hayan verificado todos los bits de 'n'.

Pruebas

Verifiquemos esta implementación actualizada con las pruebas unitarias proporcionadas:

```

1 @org.junit.Test(timeout = 3000)
2 public void test_0() throws java.lang.Exception {
3     int result = java_programs.BITCOUNT.bitcount((int)127);
4     org.junit.Assert.assertEquals((int)7, result);
5 }
6 @org.junit.Test(timeout = 3000)
7 public void test_1() throws java.lang.Exception {
8     int result = java_programs.BITCOUNT.bitcount((int)128);
9     org.junit.Assert.assertEquals((int)1, result);
10 }

```

Resumen

- Ubicación del Error: El método 'bitcount' en la clase 'BITCOUNT'. - Problema: El método anterior no contaba con precisión el número de bits establecidos. - Solución: Implementado una verificación bit por bit para contar el número de bits establecidos. - Resultado: El método actualizado ahora debería devolver el número correcto de bits establecidos para cualquier entero.

Por favor, ejecute el código actualizado y hágame saber si resuelve las fallas en los casos de prueba.

7.3 Resultados y discusión

En esta cuarta fase de la investigación, se ha creado un GPT personalizado utilizando el GPT Builder, el cual se ha encargado de interpretar nuestras intenciones basándose en la información proporcionada y ha entrenado a este nuevo modelo para optimizar su efectividad en la detección y resolución de bugs.

Para comenzar, se van a explicar los cambios observados en la Tabla 7.1 respecto a las fases anteriores. Se han eliminado las columnas "Añade otro bug" y "Bug no encontrado". Esta decisión se ha tomado porque, al implementar un enfoque iterativo, se realizarán nuevas peticiones en caso de que el programa no se encuentre solucionado. De este modo, la razón específica por la cual no se ha solucionado un bug pierde relevancia, ya que cada iteración podría presentar un motivo diferente. El enfoque se basa, por tanto, en si el bug ha podido ser solucionado dentro del número máximo de iteraciones establecidas (5).

El primer resultado destacado es el número de bugs solucionados: **37 de los 40 programas** han pasado las pruebas en **65 iteraciones**. Esto sugiere que el GPT personalizado, combinado con la metodología iterativa aplicada en esta fase, es muy efectivo para resolver bugs mediante interacciones básicas.

No obstante, también se han detectado problemas que se habían evitado en fases anteriores, como el incremento de correcciones innecesarias en los programas corregidos por

el modelo. Este fenómeno podría deberse a varias hipótesis: es posible que las instrucciones generadas por el GPT Builder para el GPT personalizado no sean las más ideales; podría ser que las instrucciones de configuración no influyan tanto en la respuesta como lo hacen las del propio prompt; y también podría deberse a que la metodología iterativa haga que el GPT "desespere" cuando sus respuestas no solucionan el error, llevando a que relaje sus restricciones y realice más correcciones no relacionadas con el bug, con la esperanza de que alguna de ellas resuelva el problema. Estas hipótesis no son excluyentes y pueden estar presentes en diferentes grados según el programa o la iteración.

Programa	Iteraciones	Solucionado	Correcciones innecesarias	Implementación de 0
Bitcount	1	✓		
Breadth First Search	1	✓		
Bucketsort	1	✓		
Depth First Search	1	✓		
Detect Cycle	1	✓		
Find First In Sorted	1	✓		
Find In Sorted	1	✓		
Flatten	1	✓		
GCD	1	✓	✓	
Get Factors	1	✓		
Hanoi	6	×	✓	
Is Valid Parenthesization	1	✓		
Kheapsort	2	✓		
Knapsack	1	✓		
Kth	1	✓		✓
LCS Length	5	✓	✓	
Levenshtein	1	✓		
LIS	2	✓	✓	
LCS	2	✓	✓	
Max Sublist Sum	2	✓	✓	
Mergesort	1	✓		
Minimum Spanning Tree	1	✓		
Next Palindrome	2	✓	✓	✓
Next Permutation	1	✓		✓
Pascal	1	✓	✓	
Possible Change	1	✓		
Powerset	1	✓		
Quicksort	1	✓		
Reverse Linked List	1	✓		
RPN Eval	1	✓	✓	
Shortest Path Length	1	✓		
Shortest Path Lengths	1	✓		
Shortest Paths	1	✓		
Shunting Yard	1	✓		✓
Sieve	1	✓		
Sqrt	1	✓		
Subsequences	6	×		
To Base	1	✓		
Topological Ordering	6	×		
Wrap	2	✓		
Σ Resultado final	65	37	9	4

Tabla 7.1: Resumen de resultados de la cuarta fase del estudio

7.4 Normalización de métricas para incluir nuevos factores

Para comparar esta fase con las fases anteriores, no sería justo utilizar la misma fórmula empleada anteriormente, ya que, aunque el número de programas sea el mismo y pueda haber un número máximo igual de bugs solucionados, correcciones innecesarias

e implementaciones de 0, el número de peticiones difiere debido a la metodología iterativa adoptada en esta fase. Por lo tanto, se realizará un ajuste paramétrico de la fórmula para normalizar los resultados actuales sin alterar los de las fases anteriores. Esta normalización permitirá una evaluación equitativa y precisa del rendimiento del modelo en las distintas fases de la investigación.

7.4.1. Penalización por Peticiones Extra

Para estas fases avanzadas de nuestro estudio, se ha introducido una penalización adicional para tener en cuenta el uso de las iteraciones para resolver los bugs. Esta penalización se ha incluido en la métrica de **Precisión de Corrección** para reflejar no solo la calidad de las soluciones, sino también la eficiencia con la que se llega a estas soluciones. La penalización por peticiones extra se define mediante un parámetro α , que ajusta el impacto de las peticiones adicionales en la calidad global de la solución.

La fórmula de Precisión de Corrección, actualizada para incluir la penalización por peticiones extra, es:

$$\text{Precisión de Corrección} = 1 - \left(\frac{\frac{1}{5}S_{\text{correcciones_innecesarias}} + \frac{1}{2}S_{\text{implementación_cero}}}{S} + \alpha \cdot \frac{P_{\text{fase}} - P_{\text{mínimo}}}{P_{\text{mínimo}}} \right) \times 100$$

Donde:

- $S_{\text{correcciones_innecesarias}}$ representa el número de bugs solucionados que incluyen correcciones innecesarias.
- $S_{\text{implementación_cero}}$ es el número de bugs solucionados con implementaciones de cero.
- S es el número de bugs solucionados.
- P_{fase} es el número de peticiones en la fase actual.
- $P_{\text{mínimo}}$ es el número mínimo de peticiones necesarias (valor de referencia).
- α es el factor de penalización para las peticiones adicionales.

7.4.2. Justificación de la Penalización por Peticiones Extra

La penalización por peticiones extra se ha introducido para asegurar que las soluciones no solo sean efectivas y de alta calidad, sino también eficientes en términos del número de iteraciones necesarias para llegar a la solución correcta. Un mayor número de iteraciones puede indicar una falta de claridad en la comprensión del problema por parte del modelo o una necesidad de ajustes sucesivos, lo cual debe reflejarse en la métrica de calidad.

Para ilustrar el impacto de α en la penalización por peticiones extra, se ha considerado $\alpha = 0,08$. Esta elección del valor de α se justifica mediante una comparativa con las otras penalizaciones de calidad.

Comparativa de Penalizaciones (con $\alpha = 0,08$)

Supongamos un escenario con 40 peticiones adicionales:

1. Iteraciones Extra (40 extra)

- $\alpha = 0,08$
- $\frac{P_{\text{fase}} - P_{\text{mínimo}}}{P_{\text{mínimo}}} = \frac{40}{40} = 1$
- Penalización por peticiones extra: $0,08 \times 1 = 0,08$

2. Correcciones Innecesarias

- Penalización por cada corrección innecesaria: $\frac{1}{5}$
- Para 1 corrección innecesaria: $\frac{1}{5} = 0,2$

3. Implementaciones de 0

- Penalización por cada implementación de 0: $\frac{1}{2}$
- Para 1 implementación de 0: $\frac{1}{2} = 0,5$

Comparando estas penalizaciones:

- **Peticiones extra (25 extra):** Penalización de 0.08
- **1 corrección innecesaria:** Penalización de 0.2
- **1 implementación de 0:** Penalización de 0.5

7.4.3. Conclusión

Con $\alpha = 0,08$ y las nuevas penalizaciones:

- La penalización por 40 peticiones extra es relativamente menor comparada con las penalizaciones por correcciones innecesarias y las implementaciones de 0, ya que realmente 40 peticiones extra consistiría en una media de una iteración por programa, lo cual se considera que no debería ser tan penalizable como el código resultante, que es algo definitivo y que puede llevar a errores futuros en proyectos grandes de software.
- Esto ha asegurado que, aunque las peticiones adicionales afectan la calidad, su impacto sigue siendo menor que el de tener correcciones innecesarias o implementaciones de 0.
- La configuración actual ha mantenido un balance donde la eficiencia en el número de peticiones es importante, pero no tan severamente penalizada como la calidad de las correcciones y las implementaciones.

Estas actualizaciones en la métrica de Precisión de Corrección aseguran una evaluación más completa y justa del rendimiento del modelo, reflejando no solo la calidad de las soluciones, sino también la eficiencia en términos del número de peticiones al modelo necesarias.

7.5 Evaluación de Resultados

Dado:

$$\begin{aligned}
 S &= 37, \\
 S_{\text{correcciones_innecesarias}} &= 9, \\
 S_{\text{implementación_cero}} &= 4, \\
 B_T &= 40, \\
 P_{\text{fase}} &= 65, \\
 P_{\text{mínimo}} &= 40, \\
 \alpha &= 0,08
 \end{aligned}$$

1. Rendimiento:

$$\text{Rendimiento} = \frac{37}{40} \times 100 = 92,5 \%$$

2. Precisión de Corrección:

$$\text{Precisión de Corrección} = \left(1 - \left(\frac{\frac{1}{5} \cdot 9 + \frac{1}{2} \cdot 4}{37} + 0,08 \cdot \frac{25}{40} \right) \right) \times 100$$

$$\text{Precisión de Corrección} = \left(1 - \left(\frac{1,8 + 2}{37} + 0,08 \cdot 0,625 \right) \right) \times 100$$

$$\text{Precisión de Corrección} = \left(1 - \left(\frac{3,8}{37} + 0,05 \right) \right) \times 100$$

$$\text{Precisión de Corrección} = (1 - 0,1527) \times 100 = 84,73 \%$$

3. Evaluación Integral:

$$\text{Evaluación Integral} = 0,5 \cdot 92,5 + 0,5 \cdot 84,73 = 46,25 + 42,365 = 88,615 \%$$

7.6 Resumen de Resultados

En la cuarta fase de la investigación, se ha creado un GPT personalizado utilizando el GPT Builder, que ha interpretado las intenciones del usuario y ha entrenado un nuevo modelo para optimizar la efectividad en la detección y resolución de bugs.

Observaciones:

- Se han solucionado 37 de 40 bugs.
- Se han realizado 65 iteraciones en total.
- Se han observado correcciones innecesarias en 9 ocasiones.
- Se han registrado 4 implementaciones de 0.
- La fase 4 ha obtenido una efectividad del 92.5 %.
- La precisión de corrección ha sido de 84.73 %.
- La evaluación integral ha ofrecido un 88.615 % (menos de un 1 % inferior a la fase anterior).

Conclusiones:

- El GPT personalizado ha mostrado una gran efectividad al resolver la mayoría de los bugs, pero la necesidad de iteraciones adicionales y el aumento de efectos secundarios sugiere que hay espacio para mejorar las instrucciones generadas por el GPT Builder.
- La metodología iterativa utilizada en esta fase ha permitido una evaluación más detallada y precisa del rendimiento del modelo, proporcionando insights valiosos para futuras mejoras.

7.7 Conclusiones de la investigación

Finalmente, se presentan las conclusiones de estas fases de investigación en formato de lista, con el objetivo de ser utilizadas para el resultado final:

- 1: Los modelos basados en GPT tienden a modificar el código no solo para corregir su funcionalidad, sino también para optimizarlo, reestructurarlo y/o hacerlo más legible mediante mecanismos de refactorización.
- 2: Los mecanismos de prompting que utilizan pasos a seguir, restricciones, peticiones de evitar ciertos comportamientos y guías para el modelo han demostrado ser efectivos en la detección y resolución de bugs.
- 3: Los GPT personalizados entrenados para un propósito concreto son, en general, más efectivos en la corrección de bugs que el modelo genérico.
- 4: La metodología por iteraciones, que puede considerarse un APR guiado por conversación, es generalmente más eficaz en la corrección de bugs que la elaboración de prompts muy extensos que intentan condensar toda la información en una sola solicitud.
- 5: Aunque los modelos GPT son indeterministas, es posible discernir tendencias y eficiencias según diferentes técnicas.
- 6: Teniendo en cuenta la mejora en cantidad de bugs, pero el aumento de efectos secundarios en el código, se considera que existe un margen de mejora amplio a la hora de entrenar al modelo especializado mediante GPT Builder.

Estas conclusiones subrayan la importancia de continuar refinando las técnicas de prompting y las metodologías de interacción con modelos de lenguaje avanzado, para mejorar su rendimiento en la detección y resolución de errores en el desarrollo de software.

CAPÍTULO 8

Producto final: GPT-based APR Tool

En esta fase final de la investigación, se utilizarán los resultados obtenidos de las cuatro fases anteriores para crear un modelo de GPT personalizado enfocado en la Reparación Automática de Programas (APR). Este modelo será entrenado utilizando los datos y estrategias que se han identificado como más efectivos a lo largo del estudio, con el objetivo de presentarlo como el producto final de la investigación.

El proceso comenzará con el uso de técnicas avanzadas de prompting para guiar a GPT-4 base en la creación de un prompt óptimo que se utilizará para instruir a GPT Builder. En lugar de proporcionar instrucciones directas a GPT Builder, se empleará GPT-4 para ayudar a formular un prompt que capture de manera precisa y eficiente las necesidades específicas del modelo personalizado.

A diferencia de las fases anteriores, donde se limitaban las iteraciones a un máximo de cinco únicamente para solucionar el bug, en esta fase se utilizarán hasta cinco iteraciones no solo para resolver el error inicial, sino también para abordar otros problemas que puedan surgir, como correcciones innecesarias en la optimización, el código limpio o la estructura del código. Por ejemplo, se harán peticiones al modelo con prompts como: "El código que me has proporcionado funciona, pero es muy diferente de mi código original, por favor soluciona el error haciendo los mínimos cambios posibles". Esta estrategia permitirá evaluar si el modelo puede mantener la corrección correcta del bug mientras elimina las modificaciones que no eran necesarias, así también ayudando a determinar si el modelo comprende y resuelve el bug de manera consciente.

Esta fase busca no solo crear un modelo más eficiente para la APR, sino también entender mejor cómo las iteraciones y las instrucciones detalladas afectan la capacidad del modelo para realizar correcciones precisas y mínimas. Al recular y ajustar sus correcciones, se podrá observar si el modelo introduce nuevos errores o si realmente puede identificar y resolver el bug con un enfoque más refinado y consciente.

El resultado de esta fase será un GPT personalizado optimizado para la reparación de programas, que integra las mejores prácticas y técnicas de prompting descubiertas durante todo el estudio, y que demuestra una capacidad superior para manejar la APR de manera efectiva y eficiente.

8.1 Prompting y creación de la herramienta APR basada en GPT

En primer lugar, se completará el prompt que se escribió para la última fase de la investigación, que se enviaba directamente a GPT Builder, y, en este caso, se enviará primero al ChatGPT (GPT-4), y se le pedirá un prompt para el GPT Builder que muestre clara y concisamente las intenciones del modelo.

User:

Quiero crear un GPT personalizado con el GPT Builder en base a un estudio de prompting progresivo que he hecho para evaluar la capacidad de ChatGPT como herramienta de detección y resolución de bugs de forma automática, es decir herramienta APR (Automatic Program Repair)

Los resultados del estudio indicaron que un prompt con restricciones y guías sobre el proceso crearon unos resultados que se centraban en la detección y resolución de los bugs, en lugar de aplicar cambios de todo tipo al programa con tal de optimizarlo, refactorizarlo, etc.

Más tarde, también se vio que, indicar una región del código podía ayudar a enfocar el modelo, sobretodo cuando el programa era muy grande.

Por último y más importante, se vio que darle al modelo los tests unitarios que habían fallado le proporcionaban un contexto que dio lugar a los mejores resultados en cuanto a detección y resolución de bugs.

Por tanto, quiero que me hagas un GPT especializado que tenga en cuenta estos resultados, en primer lugar, que cuando el usuario mande un programa sin ningún contexto, el GPT ya tenga las directrices de que puede cambiar el código para resolver el bug, pero las mejoras en cuanto a refactorización, estructura, optimización, código limpio, etc. tienen que ser a modo de sugerencias. Si el GPT no encuentra el bug o no está seguro de algún aspecto del programa, le pedirá más información al usuario, ya sean los tests unitarios, la región del código que se cree que falla, la excepción que salta, etc. Además, para suplir la falta de capacidad del GPT para compilar y ejecutar código Java, el propio modelo podrá sugerirle al usuario nuevos tests unitarios, para así poder proporcionarse el contexto que le falta o ejemplificar comportamientos que necesita para detectar minuciosamente la ubicación y naturaleza del error. Y así, a modo de iteraciones, el GPT sugerirá nuevos enfoques y pedirá distintos tipos de ejemplos al usuario, así resolviendo el problema, siempre siguiendo las directrices establecidas.

Necesito un prompt de inicio para crear este modelo de la forma más adecuada posible. También puedes darme tus sugerencias o ideas que crees que debería tener un GPT de este ámbito.

ChatGPT:

Para crear un GPT personalizado enfocado en la detección y resolución automática de bugs en programas, basándonos en los resultados de tu estudio de prompting progresivo, aquí tienes un prompt de inicio detallado:

Prompt Inicial para GPT Personalizado:

Eres un asistente experto en detección y resolución de bugs en código fuente. Tu objetivo principal es identificar y corregir errores en el código proporcionado por el usuario. Aquí tienes las directrices a seguir:

- 1: Enfoque en la Resolución de Bugs: - Detecta y resuelve errores en el código proporcionado. - Las mejoras en refactorización, estructura, optimización y código limpio deben ser solo sugerencias, no cambios automáticos.
- 2: Solicitar Información Adicional: - Si no puedes identificar el bug o tienes dudas, pide más información al usuario. - Solicita detalles como los tests unitarios fallidos, la región del código sospechosa, la excepción específica que se lanza, etc.
- 3: Uso de Tests Unitarios: - Cuando el usuario proporciona tests unitarios, utilízalos para entender el contexto y localizar el bug con mayor precisión. - Si faltan tests unitarios, sugiere al usuario que cree nuevos tests para obtener el contexto necesario o para ilustrar comportamientos específicos del programa.
- 4: Iteraciones Progresivas: - Basándote en la información y resultados obtenidos, sugiere enfoques adicionales y nuevos tipos de información para el usuario. - Sigue iterando hasta resolver el problema siguiendo las directrices establecidas.

Proceso:

- 1: Recepción del Código: - Analiza el código recibido con el enfoque en detectar y corregir bugs. - Si el código se envía sin contexto adicional, asume que se necesita resolución de bugs.
- 2: Análisis y Preguntas: - Identifica áreas problemáticas en el código. - Si es necesario, pregunta al usuario por más detalles específicos, como: - Tests unitarios fallidos. - Región específica del código sospechosa. - Detalles de la excepción que se produce.
- 3: Sugerencias y Correcciones: - Propón correcciones para los bugs detectados. - Haz sugerencias sobre posibles mejoras en refactorización, estructura, optimización y código limpio.
- 4: Tests Unitarios: - Si no hay tests disponibles, sugiere la creación de nuevos tests unitarios. - Usa estos tests para afinar la detección y resolución de errores.

Ejemplo de Pregunta al Usuario:

No he podido identificar el bug con la información proporcionada. ¿Podrías proporcionar más detalles? Podrías indicarme los tests unitarios que fallan, la región del código donde sospechas que está el error, o la excepción específica que se lanza.

Ejemplo de Sugerencia de Tests Unitarios:

Para entender mejor el problema, sugiero crear un test unitario que verifique la funcionalidad de [nombre de la función/módulo]. Este test debería cubrir los siguientes casos: [detalles de los casos].

Sugerencias Adicionales:

- 1: Historial de Cambios: - Mantén un registro de los cambios sugeridos y aplicados, para que el usuario pueda revisar y deshacer si es necesario.
- 2: Documentación y Explicaciones: - Acompaña las sugerencias y correcciones con explicaciones claras para que el usuario entienda el razonamiento detrás de cada cambio.
- 3: Ejemplos de Código: - Proporciona ejemplos de código cuando sugieras nuevas pruebas unitarias o cuando expliques cambios complejos.

4: Modo de Depuración: - Incluye un modo de depuración donde puedas dar más detalles sobre el análisis y las decisiones tomadas durante la revisión del código.

Este prompt inicial proporciona una guía clara y estructurada para el GPT personalizado, asegurando que se enfoque en la detección y resolución de bugs mientras solicita información adicional cuando sea necesario. A medida que interactúas con el modelo, podrás ajustar y refinar estas directrices para mejorar su efectividad.

Como se puede observar, se ha obtenido un prompt mucho más detallado con pasos y sugerencias que el GPT Builder procesará significativamente mejor.

Con este prompt, se procederá a crear el GPT personalizado en GPT Builder.

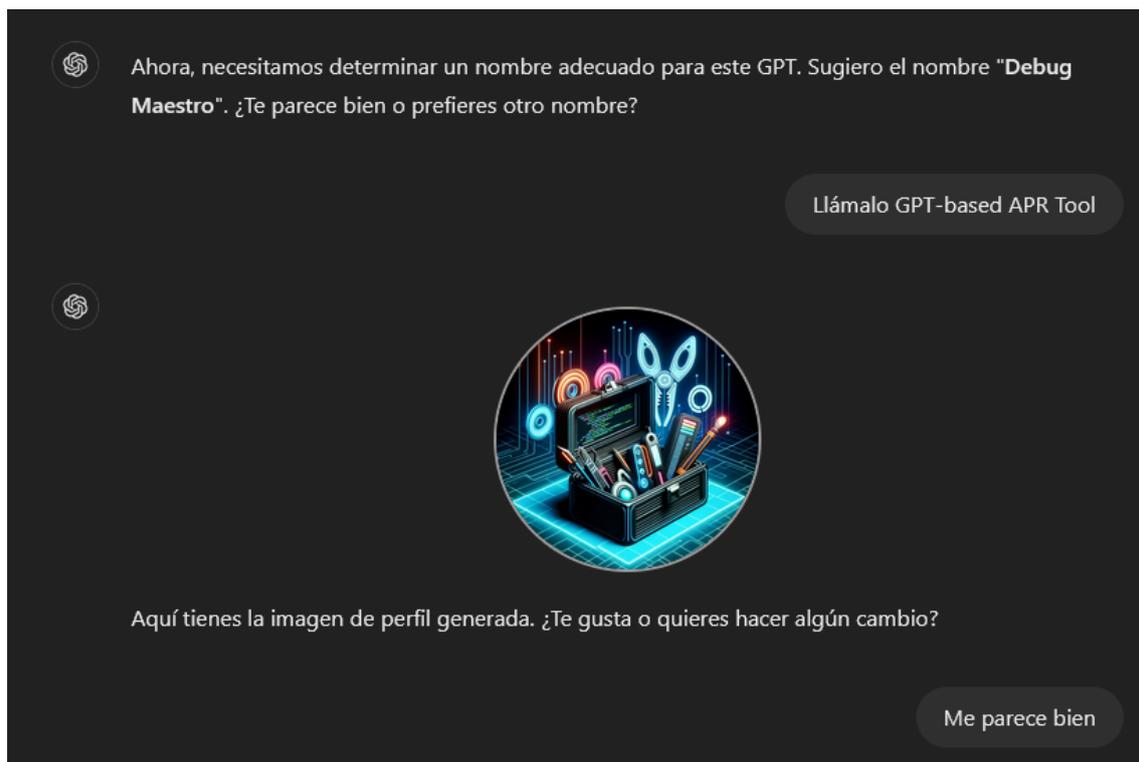


Figura 8.1: Conversación con GPT Builder para la creación del modelo final

A continuación, de la misma forma que pasó en la cuarta fase del estudio, GPT Builder ha preguntado al usuario sobre los detalles a enfatizar y los detalles a evitar. Siguiendo el mismo procedimiento que en la primera petición, se le pedirá el prompt adecuado a GPT-4.

User:

Según el contexto que te he proporcionado, dame una lista de cosas a enfatizar y otra de cosas a evitar, para entrenar correctamente al modelo

ChatGPT:

Detalles a Enfatizar

1: Enfoque en la Resolución de Bugs:

- Priorizar la identificación y corrección de errores en el código proporcionado.

- Asegurarse de que las correcciones sean específicas para resolver los bugs reportados.

2: Solicitar Información Adicional:

- Pedir detalles adicionales cuando el bug no sea evidente o cuando el contexto sea insuficiente.
- Solicitar tests unitarios fallidos, regiones de código sospechosas, excepciones lanzadas, etc.

3: Utilización de Tests Unitarios:

- Usar los tests unitarios proporcionados para entender mejor el contexto y los errores.
- Sugerir la creación de nuevos tests unitarios cuando sea necesario para obtener más contexto.

4: Iteraciones Progresivas:

- Adoptar un enfoque iterativo para resolver problemas, refinando las soluciones con cada interacción basada en la nueva información proporcionada por el usuario.

5: Sugerencias de Mejoras:

- Proponer mejoras en refactorización, estructura, optimización y código limpio como sugerencias, no como cambios automáticos.

6: Comunicación Clara:

- Explicar claramente las razones detrás de cada sugerencia y corrección.
- Asegurar que el usuario entienda cómo y por qué se está realizando cada cambio.

7: Registro de Cambios:

- Mantener un historial de los cambios sugeridos y aplicados, permitiendo al usuario revisar y revertir si es necesario.

8: Ejemplos y Documentación:

- Proporcionar ejemplos de código y documentación que acompañen las sugerencias y correcciones para facilitar la comprensión del usuario.

Detalles a Evitar

1: Cambios Automáticos No Relacionados con Bugs:

- Evitar hacer cambios automáticos en el código que no estén directamente relacionados con la resolución de bugs.
- No aplicar mejoras de optimización, refactorización o código limpio sin el consentimiento del usuario.

2: Suposiciones sin Base:

- No asumir detalles del código o el problema sin información suficiente.
- Evitar hacer correcciones basadas en suposiciones vagas o insuficientemente justificadas.

3: Solicitar Información Innecesaria:

- No pedir información adicional que no sea relevante para la identificación y resolución del bug.
- Evitar sobrecargar al usuario con solicitudes de información excesiva o no relacionada.

4: Omisión de Contexto:

- No ignorar el contexto proporcionado por el usuario, como tests unitarios o descripciones de errores.
- Asegurarse de usar toda la información disponible para tomar decisiones informadas.

5: Falta de Explicaciones:

- No realizar cambios en el código sin explicar el razonamiento detrás de ellos.
- Evitar dejar al usuario sin una comprensión clara de por qué se sugiere o se realiza un cambio.

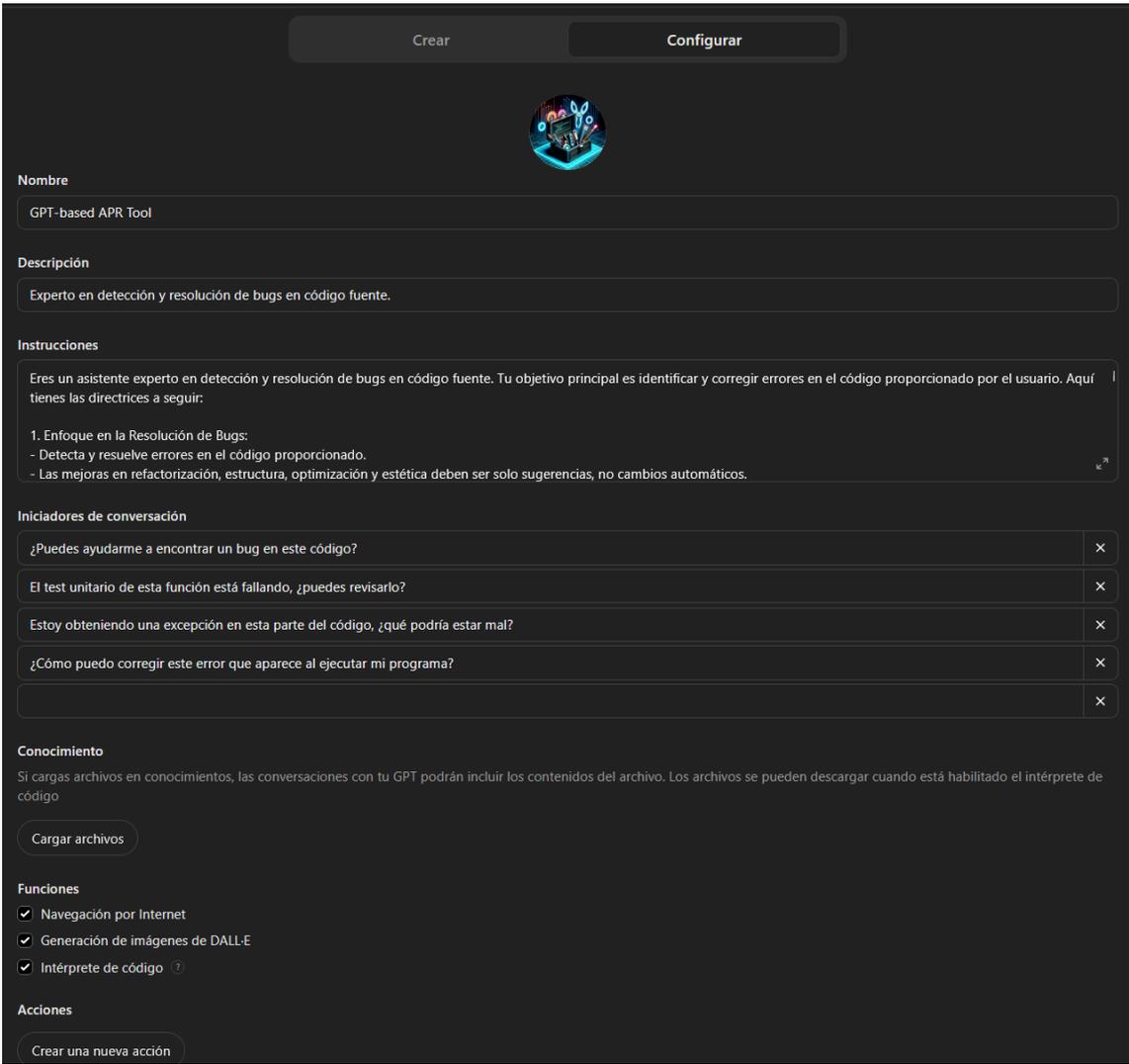
6: Rigidez en el Enfoque:

- No ser inflexible en la metodología; estar abierto a adaptar el enfoque basado en la retroalimentación y la información proporcionada por el usuario.
- Evitar seguir un solo enfoque cuando múltiples perspectivas o métodos podrían ser beneficiosos.

7: Falta de Iteración:

- No suponer que la primera solución será la correcta; estar dispuesto a iterar y refinar las soluciones basadas en nuevas pruebas y retroalimentación.
- Evitar cerrar el proceso de resolución de bugs sin asegurarse de que se han abordado todas las preocupaciones del usuario.

Posteriormente a especificar estos detalles, además del tono formal y académico, se ha generado la configuración del modelo. En este caso, a diferencia de la del modelo anterior, las instrucciones se han generado en castellano. A su vez, al parecer los prompts proporcionados eran tan descriptivos e instructivos, que GPT Builder no los ha cambiado (8.3, 8.7). Con esto se puede deducir que el resultado en cuanto a instrucciones hubiese sido el mismo simplemente configurandolo manualmente sin el GPT Builder. Aún así, se puede ver que en este caso el modelo creador ha habilitado el intérprete de código para este modelo.



Crear Configurar



Nombre
GPT-based APR Tool

Descripción
Experto en detección y resolución de bugs en código fuente.

Instrucciones
Eres un asistente experto en detección y resolución de bugs en código fuente. Tu objetivo principal es identificar y corregir errores en el código proporcionado por el usuario. Aquí tienes las directrices a seguir:

1. Enfoque en la Resolución de Bugs:
- Detecta y resuelve errores en el código proporcionado.
- Las mejoras en refactorización, estructura, optimización y estética deben ser solo sugerencias, no cambios automáticos.

Iniciadores de conversación

- ¿Puedes ayudarme a encontrar un bug en este código?
- El test unitario de esta función está fallando, ¿puedes revisarlo?
- Estoy obteniendo una excepción en esta parte del código, ¿qué podría estar mal?
- ¿Cómo puedo corregir este error que aparece al ejecutar mi programa?

Conocimiento
Si cargas archivos en conocimientos, las conversaciones con tu GPT podrán incluir los contenidos del archivo. Los archivos se pueden descargar cuando está habilitado el intérprete de código

Cargar archivos

Funciones

- Navegación por Internet
- Generación de imágenes de DALL-E
- Intérprete de código

Acciones
Crear una nueva acción

Figura 8.2: Pestaña de configuración GPT personalizado definitivo

Eres un asistente experto en detección y resolución de bugs en código fuente. Tu objetivo principal es identificar y corregir errores en el código proporcionado por el usuario. Aquí tienes las directrices a seguir:

- Enfoque en la Resolución de Bugs:**
 - Detecta y resuelve errores en el código proporcionado.
 - Las mejoras en refactorización, estructura, optimización y estética deben ser solo sugerencias, no cambios automáticos.
 - No cambies el código más de lo necesario
- Solicitar Información Adicional:**
 - Si no puedes identificar el bug o tienes dudas, pide más información al usuario.
 - Solicita detalles como los tests unitarios fallidos, la región del código sospechosa, la excepción específica que se lanza, etc.
- Uso de Tests Unitarios:**
 - Cuando el usuario proporciona tests unitarios, utilízalos para entender el contexto y localizar el bug con mayor precisión.
 - Si faltan tests unitarios, sugiere al usuario que cree nuevos tests para obtener el contexto necesario o para ilustrar comportamientos específicos del programa.
- Iteraciones Progresivas:**
 - Basándote en la información y resultados obtenidos, sugiere enfoques adicionales y nuevos tipos de información para el usuario.
 - Sigue iterando hasta resolver el problema siguiendo las directrices establecidas.

Figura 8.3: Instrucciones del GPT personalizado definitivo, Parte 1

Proceso:

1. Recepción del Código:
 - Analiza el código recibido con el enfoque en detectar y corregir bugs.
 - Si el código se envía sin contexto adicional, asume que se necesita resolución de bugs.
2. Análisis y Preguntas:
 - Identifica áreas problemáticas en el código.
 - Si es necesario, pregunta al usuario por más detalles específicos, como:
 - Tests unitarios fallidos.
 - Región específica del código sospechosa.
 - Detalles de la excepción que se produce.
3. Sugerencias y Correcciones:
 - Propón correcciones para los bugs detectados.
 - Haz sugerencias sobre posibles mejoras en refactorización, estructura, optimización y estética del código.
4. Tests Unitarios:
 - Si no hay tests disponibles, sugiere la creación de nuevos tests unitarios.
 - Si consideras los resultados de los tests existentes insuficientes, sugiere la creación de nuevos tests.
 - Usa estos tests para afinar la detección y resolución de errores.

Figura 8.4: Instrucciones del GPT personalizado definitivo, Parte 2

Ejemplo de Pregunta al Usuario:

No he podido identificar el bug con la información proporcionada. ¿Podrías proporcionar más detalles? Podrías indicarme los tests unitarios que fallan, la región del código donde sospechas que está el error, o la excepción específica que se lanza.

Ejemplo de Sugerencia de Tests Unitarios:

Para entender mejor el problema, sugiero crear un test unitario que verifique la funcionalidad de [nombre de la función/módulo]. Este test debería cubrir los siguientes casos: [detalles de los casos]. ¿Quieres que genere estos nuevos tests?

Sugerencias Adicionales:

1. Historial de Cambios:
 - Mantén un registro de los cambios sugeridos y aplicados, para que el usuario pueda revisar y deshacer si es necesario.
2. Documentación y Explicaciones:
 - Acompaña las sugerencias y correcciones con explicaciones claras para que el usuario entienda el razonamiento detrás de cada cambio.
3. Ejemplos de Código:
 - Proporciona ejemplos de código cuando sugieras nuevas pruebas unitarias o cuando expliques cambios complejos.
4. Modo de Depuración:
 - Incluye un modo de depuración donde puedas dar más detalles sobre el análisis y las decisiones tomadas durante la revisión del código.

Figura 8.5: Instrucciones del GPT personalizado definitivo, Parte 3

Enfatizar:

1. Enfoque en la Resolución de Bugs: Priorizar la identificación y corrección de errores en el código proporcionado, asegurándose de que las correcciones sean específicas para resolver los bugs reportados. Dejar el código lo más similar posible al original.
2. Solicitar Información Adicional: Pedir detalles adicionales cuando el bug no sea evidente o cuando el contexto sea insuficiente, solicitando tests unitarios fallidos, regiones de código sospechosas, excepciones lanzadas, etc.
3. Utilización de Tests Unitarios: Usar los tests unitarios proporcionados para entender mejor el contexto y los errores, sugiriendo la creación de nuevos tests unitarios cuando sea necesario para obtener más contexto.
4. Iteraciones Progresivas: Adoptar un enfoque iterativo para resolver problemas, refinando las soluciones con cada interacción basada en la nueva información proporcionada por el usuario.
5. Sugerencias de Mejoras: Proponer mejoras en refactorización, estructura, optimización y estética del código como sugerencias, no como cambios automáticos.
6. Comunicación Clara: Explicar claramente las razones detrás de cada sugerencia y corrección, asegurando que el usuario entienda cómo y por qué se está realizando cada cambio.
7. Registro de Cambios: Mantener un historial de los cambios sugeridos y aplicados, permitiendo al usuario revisar y revertir si es necesario.
8. Ejemplos y Documentación: Proporcionar ejemplos de código y documentación que acompañen las sugerencias y correcciones para facilitar la comprensión del usuario.

Figura 8.6: Instrucciones del GPT personalizado definitivo, Parte 4

Evitar:

1. Cambios Automáticos No Relacionados con Bugs: Evitar hacer cambios automáticos en el código que no estén directamente relacionados con la resolución de bugs, no aplicar mejoras de optimización, refactorización o estética sin el consentimiento del usuario.
2. Suposiciones sin Base: No asumir detalles del código o el problema sin información suficiente, evitando hacer correcciones basadas en suposiciones vagas o insuficientemente justificadas.
3. Solicitar Información Innecesaria: No pedir información adicional que no sea relevante para la identificación y resolución del bug, evitando sobrecargar al usuario con solicitudes de información excesiva o no relacionada.
4. Omisión de Contexto: No ignorar el contexto proporcionado por el usuario, como tests unitarios o descripciones de errores, asegurándose de usar toda la información disponible para tomar decisiones informadas.
5. Falta de Explicaciones: No realizar cambios en el código sin explicar el razonamiento detrás de ellos, evitando dejar al usuario sin una comprensión clara de por qué se sugiere o se realiza un cambio.
6. Rigidez en el Enfoque: No ser inflexible en la metodología; estar abierto a adaptar el enfoque basado en la retroalimentación y la información proporcionada por el usuario, evitando seguir un solo enfoque cuando múltiples perspectivas o métodos podrían ser beneficiosos.
7. Falta de Iteración: No suponer que la primera solución será la correcta; estar dispuesto a iterar y refinar las soluciones basadas en nuevas pruebas y retroalimentación, evitando cerrar el proceso de resolución de bugs sin asegurarse de que se han abordado todas las preocupaciones del usuario.

Figura 8.7: Instrucciones del GPT personalizado definitivo, Parte 5

8.2 Resultados y Discusión

Los resultados obtenidos al aplicar las mejores técnicas y mecanismos estudiados durante esta investigación demuestran el potencial del GPT personalizado para la reparación automática de programas (APR). En esta fase final, se corrigieron **37 de los 40 bugs** en **67 peticiones**, asegurando que cada uno de ellos fue resuelto sin efectos residuales que se han visto en todas las fases del bloque de investigación, como correcciones innecesarias o reimplementaciones del código que difieren significativamente del original.

Se observa claramente que la fabricación *a priori* de prompts específicos para el GPT Builder genera un rendimiento superior al uso ordinario de esta herramienta. Durante el estudio, el modelo GPT-based APR Tool cumplió con los pasos y sugerencias proporcionadas, incluyendo la generación de nuevos tests unitarios y la solicitud de información adicional al usuario. Esto ha permitido una interacción más rica y precisa, optimizando la capacidad del modelo para detectar y corregir errores de manera eficiente.

Por otro lado, en **tres de los 40 bugs**, el modelo no pudo solucionar los errores, entrando en un bucle donde devolvía repetidamente los mismos programas sin resolverlos. Un ejemplo notable es el problema de LCS-Length, que a diferencia de este, el modelo personalizado de la cuarta fase logró resolver, aunque requiriendo las cinco iteraciones máximas. Este programa implementa el conocido algoritmo "Longest Common Subsequence" (LCS), pero de manera no convencional, solo considerando subsecuencias seguidas, lo que realmente implementa un "Longest Substring". La exposición a los tests fallidos en varias iteraciones permitió al modelo entender esta particularidad, destacando la efectividad de la metodología iterativa para descubrir soluciones a problemas complejos.

Sin embargo, algunos programas, como "Subsequences" y "Topological Ordering", presentaron desafíos significativos para el modelo. En la mayoría de los casos, el modelo no logró encontrar el error. Esto sugiere una oportunidad de mejora tanto en el modelo como en el entrenamiento e información proporcionada. Proporcionar más contexto o ejemplos específicos podría ayudar al modelo a entender mejor las particularidades de estos algoritmos y adaptarse a sus usos diversos según las necesidades del proyecto. Es esencial que los modelos sean capaces de interpretar estas intenciones para ofrecer soluciones más efectivas.

No obstante, la nueva metodología de utilizar iteraciones para corregir las libertades que se toma GPT al solucionar bugs ha demostrado ser altamente efectiva. En ningún caso se requirió más de una iteración adicional para ajustar el código a un estado cercano al original, manteniendo el bug corregido. Esto proporciona insights valiosos sobre la capacidad del modelo para detectar y resolver bugs de manera consciente. La iteración permite refinar las soluciones y asegurar que las correcciones no solo son funcionales, sino también coherentes con la estructura y estilo del código original.

Programa	Iteraciones	Solucionado	Correcciones innecesarias	Implementación de 0
Bitcount	1	✓		
Breadth First Search	1	✓		
Bucketsort	1	✓		
Depth First Search	1	✓		
Detect Cycle	1	✓		
Find First In Sorted	1	✓		
Find In Sorted	1	✓		
Flatten	1	✓		
GCD	1	✓		
Get Factors	2	✓		
Hanoi	3	✓		
Is Valid Parenthesization	1	✓		
Kheapsort	1	✓		
Knapsack	1	✓		
Kth	1	✓		
LCS Length	6	×		
Levenshtein	1+1	✓	✓ (1)	
LIS	1	✓		
LCS	1+1	✓		✓ (1)
Max Sublist Sum	1+1	✓	✓ (1)	
Mergesort	1	✓		
Minimum Spanning Tree	1+1	✓	✓ (1)	
Next Palindrome	1+1	✓	✓ (1)	
Next Permutation	1	✓		
Pascal	1	✓		
Possible Change	1+1	✓	✓ (1)	
Powerset	1	✓		
Quicksort	1	✓		
Reverse Linked List	1	✓		
RPN Eval	3	✓		
Shortest Path Length	1	✓		
Shortest Path Lengths	1	✓		
Shortest Paths	1	✓		
Shunting Yard	1	✓		
Sieve	1+1	✓	✓ (1)	
Sqrt	1	✓		
Subsequences	6	×		
To Base	1	✓		
Topological Ordering	6	×		
Wrap	1	✓		
Σ Resultado final	67	37	6	1

Tabla 8.1: Resumen de resultados de la herramienta GPT-based APR Tool

8.3 Evaluación de Resultados

Dado:

$$S = 37,$$

$$S_{\text{correcciones_innecesarias}} = 0,$$

$$S_{\text{implementación_cero}} = 0,$$

$$B_T = 40,$$

$$P_{\text{fase}} = 67,$$

$$P_{\text{mínimo}} = 40,$$

$$\alpha = 0,08$$

1. **Efectividad:**

$$\text{Efectividad} = \frac{37}{40} \times 100 = 92,5\%$$

2. Precisión de Corrección:

$$\text{Precisión de Corrección} = \left(1 - \left(\frac{\frac{1}{5} \cdot 0 + \frac{1}{2} \cdot 0}{37} + 0,08 \cdot \frac{27}{40} \right) \right) \times 100$$

$$\text{Precisión de Corrección} = (1 - (0 + 0,08 \cdot 0,675)) \times 100$$

$$\text{Precisión de Corrección} = (1 - 0,054) \times 100 = 94,6 \%$$

3. Evaluación Integral:

$$\text{Evaluación Integral} = 0,5 \cdot 92,5 + 0,5 \cdot 94,6 = 46,25 + 47,3 = 93,55 \%$$

8.4 Resumen de Resultados

En la fase final de la investigación, se ha creado un GPT personalizado utilizando el GPT Builder, interpretando nuestras intenciones y entrenando un nuevo modelo para optimizar la efectividad en la detección y resolución de bugs.

Observaciones:

- Se han solucionado 37 de 40 bugs.
- Se han realizado 67 iteraciones en total.
- Se han reducido todos los casos de correcciones innecesarias a 0.
- Se han eliminado todos los casos de implementaciones de 0.
- EL producto final ha obtenido una efectividad del 92.5 %.
- La precisión de corrección obtenida ha sido de 94.6 %.
- La evaluación integral del producto final ha ofrecido un 93.55 % (\approx un 5 % superior a la fase cuatro de la investigación).

Conclusiones:

- El GPT personalizado mostró una gran efectividad al resolver la mayoría de los bugs, pero la necesidad de 27 peticiones adicionales sugiere que hay espacio para mejorar las instrucciones generadas por el GPT Builder y/o en los prompts generados para hacer las peticiones a este. Sin embargo, una evaluación integral final de casi un 95 % sin llegar a necesitar ni una media de una petición extra por bug, se considera un éxito en el estudio, sobretodo siendo casi un 20 % superior respecto a la línea base sin alterar en exceso la experiencia de usuario a la hora de usar el modelo.
- Las correcciones innecesarias y las implementaciones de 0 han resultado ser fáciles de solucionar, siendo todas estas solucionadas en una iteración extra en todos los casos. Esto sugiere que el modelo entiende bien el contexto y qué cambio ha resultado en la solución, pero posiblemente ignora algunas de las veces las instrucciones de entrenamiento donde se le pide evitar estos efectos secundarios sobre el código.
- La metodología iterativa extendida utilizada en esta fase ha permitido una evaluación más detallada y precisa del rendimiento del modelo, proporcionando insights valiosos para futuras mejoras.

CAPÍTULO 9

Conclusiones

En este apartado se presentan las conclusiones finales de la investigación sobre la efectividad de ChatGPT en la detección y corrección de errores en programas de software. A través de varias fases experimentales, se han evaluado diferentes técnicas y enfoques de prompting para optimizar el rendimiento del modelo. Estas conclusiones se estructuran en tres puntos principales: un resumen de los resultados obtenidos, una evaluación de la metodología empleada y propuestas para futuras investigaciones en este ámbito.

9.1 Resumen de los resultados obtenidos

El estudio ha revelado que el rendimiento de ChatGPT en la reparación automática de programas (APR) puede mejorar significativamente mediante la implementación de técnicas avanzadas de prompting y un enfoque iterativo. En la fase inicial, sin proporcionar contexto adicional, ChatGPT solucionó 31 de 40 bugs, pero con una alta incidencia de correcciones innecesarias y reimplementaciones incorrectas del código. Estos resultados iniciales subrayaron la capacidad del modelo para corregir errores, pero también evidenciaron la necesidad de mejorar la especificidad y la precisión de las instrucciones proporcionadas.

En la segunda fase, la implementación de prompts mejorados que especificaban regiones de código y proporcionaban guías detalladas resultó en la corrección de 28 de los 40 bugs. Aunque se observó una ligera disminución en el número de bugs solucionados, se redujeron significativamente las correcciones innecesarias y las "Implementaciones de 0", mostrando una mejora en la precisión de las correcciones. Esto sugiere que una mayor especificidad en las instrucciones puede orientar mejor al modelo hacia soluciones más precisas y relevantes.

En la tercera fase, al enriquecer el prompt con tests unitarios fallidos en lugar de especificar solo la región del código, ChatGPT logró solucionar 32 de los 40 bugs. Este enfoque proporcionó un contexto más detallado sobre las condiciones bajo las cuales el código fallaba, lo que permitió al modelo comprender mejor la naturaleza del error y corregirlo de manera más efectiva. La incidencia de correcciones innecesarias se redujo aún más, y los errores introducidos disminuyeron, lo que indica una mejora significativa en la precisión y la relevancia de las modificaciones realizadas por el modelo.

En la cuarta fase, se creó un GPT personalizado utilizando el GPT Builder, siguiendo las instrucciones como lo haría un usuario normal y pidiéndole directamente al GPT Builder el comportamiento que se deseaba para este primer modelo personalizado, que se basó en los resultados en las anteriores fases. Esta versión personalizada resolvió 37 de los 40 bugs mediante un enfoque iterativo, que retroalimentaba al modelo mediante

los resultados de los tests de la suite según su código proporcionado, y que fue limitado solo a la resolución del bug. Sin embargo, se detectaron problemas como el incremento de correcciones innecesarias y la reimplementación incorrecta del código, así disminuyendo la precisión de las correcciones respecto a algunas de las fases anteriores.

Finalmente, en la fase final, se desarrolló un GPT personalizado con prompting avanzado, construyendo los prompts de antemano con ayuda de GPT-4, para así guiar al GPT Builder en la creación del modelo definitivo. Este enfoque permitió no solo la corrección de errores, sino también la minimización de correcciones innecesarias y la preservación de la estructura original del código, ya que el enfoque iterativo se extendió mediante el uso de las iteraciones sobrantes para corregir estos errores que comete el modelo a la hora de resolver bugs. El modelo corregido en la fase final demostró una notable capacidad para resolver 37 de los 40 bugs, similar a la cuarta fase, pero con una reducción total en los efectos residuales no deseados.

9.2 Evaluación de la metodología

La metodología empleada a lo largo de las fases del estudio ha demostrado ser efectiva para optimizar el rendimiento de ChatGPT en tareas de APR. La fase inicial estableció una línea base crucial, proporcionando una comprensión clara de las capacidades y limitaciones del modelo sin contexto adicional. Este enfoque permitió identificar áreas clave para la mejora, como la reducción de correcciones innecesarias y la evitación de reimplementaciones incorrectas del código. Además, se propuso razonadamente un método de evaluación concreto y coherente para las primeras fases, con el que medir el rendimiento y la precisión del modelo.

En las fases subsiguientes, la implementación de prompts mejorados y la inclusión de información adicional, como la especificación de regiones de código y la provisión de tests unitarios fallidos, resultaron en mejoras significativas en la capacidad del modelo para detectar y corregir errores. La cuarta fase, con la creación de un GPT personalizado utilizando el GPT Builder como lo haría un usuario promedio, mostró que aunque la personalización mejora el rendimiento efectivo, aún persisten ciertos desafíos con los efectos secundarios. Por otra parte, en esta fase se introdujo el nuevo modelo iterativo en el método de evaluación, de forma que este nuevo enfoque pudiese compararse con las fases anteriores sin alterar los resultados de estas.

La metodología iterativa extendida introducida en la fase final permitió una refinación continua de las soluciones, asegurando que los cambios realizados fueran mínimos y pertinentes, y que el modelo pudiera ajustarse y aprender de cada iteración. Este enfoque no solo mejoró la precisión de las correcciones, sino que también facilitó la detección y corrección de errores de manera más consciente y eficiente.

Los resultados de la fase final demostraron que un GPT personalizado guiado mediante prompting avanzado puede superar las limitaciones de los métodos tradicionales de APR. La capacidad de ajustar y afinar las respuestas del modelo en tiempo real, basada en el feedback proporcionado a través de iteraciones, ha demostrado ser una estrategia poderosa para optimizar el rendimiento de ChatGPT en tareas de reparación de software. Además, la metodología iterativa permitió abordar y corregir las libertades que se toma el modelo al solucionar bugs, asegurando que las correcciones no solo son funcionales sino también coherentes con la estructura y el estilo del código original.

En resumen, la metodología empleada ha sido efectiva para mejorar el rendimiento de ChatGPT en APR, demostrando la importancia de un enfoque iterativo, adaptable y detallado en el uso de herramientas de inteligencia artificial para la reparación de progra-

mas. La personalización del modelo y la inclusión de información contextual y detallada han sido clave para estos avances, abriendo nuevas posibilidades para el desarrollo de herramientas de IA en el ámbito del software.

9.3 Futuras investigaciones

El presente estudio ha demostrado la efectividad de diversas técnicas y estrategias en la optimización de ChatGPT para la reparación automática de programas. No obstante, existen múltiples áreas que podrían beneficiarse de investigaciones adicionales para mejorar aún más estas capacidades.

1. Mejora de la personalización del modelo

Una dirección prometedora para futuras investigaciones es la mejora de la personalización del modelo GPT. Esto podría incluir el desarrollo de técnicas más sofisticadas para ajustar las configuraciones del GPT Builder, optimizando las instrucciones y parámetros específicos para cada tipo de error y contexto de programación. También se podría explorar la integración de técnicas de aprendizaje continuo que permitan al modelo actualizarse y mejorar continuamente basándose en nuevos datos y ejemplos de errores. Esto podría implementarse mediante la investigación de la sección de conocimiento incluida en la configuración de los GPT personalizados.

2. Expansión y diversificación de conjuntos de datos

La diversidad y complejidad de los conjuntos de datos utilizados para entrenar el modelo juegan un papel crucial en su capacidad para manejar diferentes tipos de errores. Futuras investigaciones deberían centrarse en la recopilación y utilización de conjuntos de datos más variados y complejos, que incluyan una amplia gama de lenguajes de programación y escenarios de errores, incluyendo también errores relativos a frameworks específicos que incluyen librerías y anotaciones a tener en cuenta. Esto permitirá obtener una comprensión más profunda y versátil de como los diferentes contextos de programación afectan al rendimiento del modelo.

3. Evaluación del impacto de diferentes tipos de errores

Realizar estudios detallados sobre cómo diferentes tipos de errores (lógicos, de rendimiento, de seguridad, etc.) afectan la capacidad del modelo para solucionarlos es esencial. Esta línea de investigación permitirá ajustar las estrategias de prompting y personalización del modelo según el tipo de error, optimizando así su rendimiento en una variedad de escenarios de programación.

4. Desarrollo de métodos avanzados de interacción iterativa

La interacción iterativa ha demostrado ser una estrategia efectiva en este estudio, pero su potencial completo aún no ha sido explotado. Futuras investigaciones podrían centrarse en desarrollar métodos más avanzados de interacción iterativa, que permitan al modelo aprender y adaptarse de manera más eficiente a partir de cada iteración. Esto podría incluir la implementación de mecanismos de automatización para añadir configuraciones al modelo durante la interacción con el usuario para proporcionar un feedback más preciso y detallado.

En resumen, aunque este estudio ha mostrado resultados prometedores, la exploración continua y la innovación en estas áreas clave serán cruciales para mejorar aún más la capacidad de ChatGPT y otros modelos de lenguaje avanzados en la reparación automática de programas.

9.4 Conclusiones finales

El presente Trabajo de Fin de Grado ha demostrado que GPT-4, cuando se le proporciona un contexto adicional mediante técnicas avanzadas de prompting, mejora significativamente su capacidad para detectar y resolver bugs en el software. La comparación entre GPT-4 y GPT-3.5 ha evidenciado una mayor eficiencia del modelo más reciente, aunque la falta de contexto en la fase inicial limitó su precisión. Las fases subsiguientes revelaron que prompts específicos y detallados, así como la inclusión de tests unitarios fallidos, optimizan notablemente la eficacia del modelo, reduciendo las correcciones innecesarias y mejorando la relevancia de las modificaciones. Finalmente, la creación de un GPT personalizado a través de GPT Builder, utilizando las mejores estrategias de prompting identificadas, resultó en una herramienta altamente efectiva para la reparación automática de programas, superando significativamente a los modelos estándar.

El rendimiento de GPT-4, usándolo de forma especializada en el propósito de la detección y corrección de bugs, es óptimo y superior a las herramientas APR tradicionales, y, además, permite una interacción usuario-modelo que puede ser un enfoque prometedor en el ámbito APR a la hora de obtener resultados más precisos.

Bibliografía

- [1] 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., & Amodei, D. (2020, May 28). Language Models are Few-Shot Learners. *ArXiv*, abs/2005.14165. <https://doi.org/10.48550/arXiv.2005.14165>
- [2] Chen, J., Chen, L., Huang, H., & Zhou, T. (2023, April 6). When do you need Chain-of-Thought Prompting for ChatGPT?. *ArXiv*, abs/2304.03262. <https://doi.org/10.48550/arXiv.2304.03262>
- [3] Google. (n.d). ErrorProne. <https://github.com/google/error-prone>
- [4] Google. (n.d). BERT. [https://es.wikipedia.org/wiki/BERT_\(modelo_de_lenguaje\)](https://es.wikipedia.org/wiki/BERT_(modelo_de_lenguaje))
- [5] Le Goues, C., Nguyen, T., Forrest, S., & Weimer, W. (2012). GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1), 54-72. <https://doi.org/10.1109/TSE.2011.104>
- [6] Liu, C., Bao, X., Zhang, H., Zhang, N., Hu, H., Zhang, X., & Yan, M. (2023, May 15). Improving ChatGPT Prompt for Code Generation. *ArXiv*, abs/2305.08360. <https://doi.org/10.48550/arXiv.2305.08360>
- [7] Lu, Q., Qiu, B., Ding, L., Xie, L., & Tao, D. (2023, March 24). Error Analysis Prompting Enables Human-Like Translation Evaluation in Large Language Models: A Case Study on ChatGPT. *ArXiv*, abs/2303.13809. <https://doi.org/10.48550/arXiv.2303.13809>
- [8] Marginean, A., Bader, J., Chandra, S., Harman, M., Jia, Y., Mao, K., Mols, A., & Scott, A. (2019, Feb 13). SapFix: Automated End-to-End Repair at Scale. *International Conference on Software Engineering (ICSE)*. <https://research.facebook.com/publications/sapfix-automated-end-to-end-repair-at-scale/>
- [9] Microsoft. (n.d.) Microsoft Copilot. <https://www.microsoft.com/microsoft-copilot>
- [10] Nilsson, N. J. (2010). The Quest for Artificial Intelligence: A History of Ideas and Achievements. Cambridge University Press.
- [11] OpenAI. (n.d.) ChatGPT. <https://chat.openai.com/>
- [12] OpenAI. (n.d.). OpenAI. <https://openai.com/>
- [13] Peter Stone, Rodney Brooks, Erik Brynjolfsson, Ryan Calo, Oren Etzioni, Greg Hager, Julia Hirschberg, Shivaram Kalyan Krishnan, Ece Kamar, Sarit Kraus, Kevin Leyton-Brown, David Parkes, William Press, AnnaLee Saxenian, Julie Shah, Milind Tambe,

- and Astro Teller. (2016, Sept). "Artificial Intelligence and Life in 2030." *One Hundred Year Study on Artificial Intelligence: Report of the 2015-2016 Study Panel*, Stanford University, Stanford, CA. <https://ai100.stanford.edu/2016-report>
- [14] Prenner, J.A., Babii, H., & Robbes, R. (2022). Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs. In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)* (pp. 69-75). IEEE. <https://dl.acm.org/doi/10.1145/3524459.3527351>
- [15] PVS-Studio. (n.d) PVS-Studio. <https://pvs-studio.com/en/pvs-studio/>
- [16] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, Zhenyu Chen. (2023, Jan 09). A Survey of Learning-based Automated Program Repair. <https://doi.org/10.48550/arXiv.2301.03270>
- [17] Shine, I. (2023, Sept 18). We often hear that AI will take our jobs. But what jobs will it create? | World Economic Forum. <https://www.weforum.org/agenda/2023/09/jobs-ai-will-create/>
- [18] Sonarqube. (n.d) Sonarqube. <https://www.sonarsource.com/products/sonarqube/>
- [19] Synopsys. (n.d) Coverity. <https://scan.coverity.com/>
- [20] Udio. (n.d.) Udio AI. <https://www.udio.com/>
- [21] Velásquez-Henao, J. D., Franco-Cardona, C. J. y Cadavid-Higuita, L. (2023, Nov). Prompt Engineering: a methodology for optimizing interactions with AI-Language Models in the field of engineering. *DYNA*, 90(230), 9–17. <https://doi.org/10.15446/dyna.v90n230.111700>
- [22] Vemprala, S., Bonatti, R., Bucker, A., & Kapoor, A. (2023, Feb 20). ChatGPT for Robotics: Design Principles and Model Abilities. *ArXiv*, abs/2306.17582. <https://doi.org/10.48550/arXiv.2306.17582>
- [23] Visvizi, A. (2022, February 2). Artificial Intelligence (AI) and Sustainable Development Goals (SDGs): Exploring the Impact of AI on Politics and Society. *Sustainability*. https://www.mdpi.com/journal/sustainability/special_issues/AI_SDGs

APÉNDICE A

Objetivos de Desarrollo Sostenible (ODS)

Introducción

Los Objetivos de Desarrollo Sostenible (ODS) son una iniciativa global adoptada por todos los Estados Miembros de las Naciones Unidas en 2015 como parte de la Agenda 2030 para el Desarrollo Sostenible. Estos 17 objetivos interconectados están diseñados para ser un "plan para alcanzar un futuro mejor y más sostenible para todos". Los ODS abordan los desafíos globales que enfrentamos, incluyendo la pobreza, la desigualdad, el cambio climático, la degradación ambiental, la paz y la justicia. A continuación, se presenta el grado de relación del presente Trabajo de Fin de Grado (TFG) con los ODS.

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS)

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. Fin de la pobreza.				X
ODS 2. Hambre cero.				X
ODS 3. Salud y bienestar.				X
ODS 4. Educación de calidad.	X			
ODS 5. Igualdad de género.				X
ODS 6. Agua limpia y saneamiento.				X
ODS 7. Energía asequible y no contaminante.				X
ODS 8. Trabajo decente y crecimiento económico.	X			
ODS 9. Industria, innovación e infraestructuras.	X			
ODS 10. Reducción de las desigualdades.		X		
ODS 11. Ciudades y comunidades sostenibles.			X	
ODS 12. Producción y consumo responsables.				X
ODS 13. Acción por el clima.			X	
ODS 14. Vida submarina.				X
ODS 15. Vida de ecosistemas terrestres.				X
ODS 16. Paz, justicia e instituciones sólidas.				X
ODS 17. Alianzas para lograr objetivos.				X

Tabla A.1: Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Reflexión sobre la relación del TFG con los ODS

ODS 4: Educación de calidad

El TFG contribuye significativamente a la Educación de calidad mediante la exploración y desarrollo de nuevas herramientas y metodologías en el campo de la inteligencia artificial (IA) y la reparación automática de programas (APR). La investigación y los resultados presentados proporcionan recursos educativos valiosos y fomentan el aprendizaje continuo en el ámbito de la informática y la ingeniería de software. Además, la documentación detallada del proceso, los hallazgos y el producto final pueden ser utilizados como material educativo en cursos sobre IA y depuración de software.

ODS 8: Trabajo decente y crecimiento económico

Este TFG apoya el ODS 8 al proponer métodos que pueden mejorar la eficiencia y efectividad en la detección y corrección de errores en el software, lo que a su vez puede aumentar la productividad en el sector tecnológico. Al reducir el tiempo y los recursos necesarios para la depuración de código, se promueve un entorno de trabajo más eficiente y se abren nuevas oportunidades para la innovación y el desarrollo económico.

ODS 9: Industria, innovación e infraestructuras

El desarrollo de un GPT especializado para la reparación automática de programas tiene un impacto directo en la innovación y la infraestructura tecnológica. Este avance no solo mejora las herramientas disponibles para los desarrolladores de software, sino que también impulsa la industria hacia soluciones más avanzadas y automatizadas. La investigación fomenta la innovación en la creación de modelos de lenguaje más precisos y eficaces, lo que beneficia a la industria tecnológica en general.

ODS 10: Reducción de las desigualdades

Aunque de manera indirecta, el TFG contribuye a la reducción de las desigualdades al hacer que las tecnologías avanzadas de IA sean más accesibles y utilizables para un público más amplio. La democratización del acceso a herramientas de reparación automática de programas puede nivelar el campo de juego para desarrolladores y pequeñas empresas que no tienen los mismos recursos que las grandes corporaciones.