Revelando patrones arquitectónicos implícitos en proyectos de Infraestructura como Código a través de la transferencia de conocimiento de repositorios de código

por

Luis Felipe Díaz Chica

Con la guía de

Luis Daniel Benavides

Wilmer Garzón

Maestría en informática

ESCUELA COLOMBIANA DE INGENIERÍA JULIO GARAVITO

Bogota, Colombia

Agosto 2, 2023

Abstract

La infraestrucutura como código o por sus siglas en inglés IaC (Infrastructure as Code) es una modelo de gestión de recursos en la nube por medio de especificaciones de código. En nuestra investigación buscamos extraer conocimiento implícito de los proyectos de IaC relacionado a los patrones de arquitectura que están siendo utilizados en la comunidad de código libre.

Para esto hemos realizado un análisis del estado del arte en temas relacionados con el análisis estático de código con modelos de lenguaje de gran envergadura también conocidos como Large Language Models en inglés(LLM), para posteriormente aplicar técnicas de transferencia de conocimiento a un conjunto de modelos pre-entrenados y categorizar los patrones de arquitectura encontrados en los proyectos de IaC. La transferencia de conocimiento es aplicada usando refinamiento (fine-tuning) y supervisado débil. Definimos un sistema de reglas que según los componentes de la infraestructura presente en el proyecto categorizamos un posible patrón de arquitectura. Este sistema de reglas es usado para construir un dataset inicial de 13200 archivos en 4 lenguajes de programación con sus respectivas etiquetas en 11 categorías de patrones de arquitectura.

Hemos logrado encontrar una mejora significativa en la categorización de los patrones de arquitectura después de aplicar transferencia de conocimiento a los modelos pre-entrenados en código. UnixCode y CodeBERT lograron alcanzar un F1-score 0.96% de precisión durante entrenamiento. Después de aplicar los modelos a un dataset desconocido encontramos que los patrones más usado son event-driven, serverless, microservicios y object storage dentro de la comunidad open source(Github). También el lenguaje de programación predominante en Cloud Development Kit (CDK) es Typescript seguido por python. Logramos evidenciar un buen rendimiento en la clasificación de los patrones usando seq2seq como la técnica de representación del código y modelos pre-entrenados basados en RoBERTa.

Tabla de contenido

Lista de imágenes

Lista de tablas

Abbreviations

AST	**A**bstract **S**yntax **T**ree
LLM	**L**arge **L**anguage **M**odel
LSTM	**L**ong **S**hort **T**erm **M**emory

Chapter 1

Introducción

1.1 Introducción

Los sistemas de cómputo son cada vez más complejos. Millones de líneas de código son usadas para especificar los componentes de software que son desplegados sobre infraestructuras computacionales distribuidas y heterogéneas. Estas infraestructuras heterogéneas pueden incluir servidores locales desplegados en centros de datos corporativos, infraestructuras virtuales en proveedores de servicios en la nube, dispositivos móviles personales y dispositivos de internet de las cosas (IoT, Internet of Things). Todos interconectados por redes públicas y privadas. Diseñar, implementar, mantener y administrar sistemas de esta envergadura es una tarea difícil para los ingenieros.

Para atacar esta complejidad se han desarrollado diferentes herramientas y técnicas. Considere por ejemplo lo que hoy se denomina Infraestructura cómo Código (IaC, Infrastructure as Code). La IaC es una técnica que permite especificar por medio de un lenguaje de programación el aprovisionamiento, configuración y gestión de infraestructura computacional. Está técnica se ha desarrollado con especial fuerza en

la última década impulsada por el crecimiento y popularización de los proveedores de servicios en la nube. Así, IaC ha permitido que la gestión de infraestructura pase de ser un tema de hardware a un tema de software. Ahora, las abstracciones computacionales como servidores, memoria, redes se gestionan directamente desde el código fuente administrando abstracciones virtuales equivalentes, es decir, servidores virtuales, memoria virtual, e inclusive redes virtuales. Por supuesto, esta especificación en lenguajes de programación y la amplia disponibilidad de recursos computacionales virtuales a bajo costo, ha creado un entorno para el desarrollo de sistemas aún más complejos, que escalan fácilmente para atender millones de usuarios y dispositivos en tiempo real.

En este trabajo estamos interesados en investigar el estado del arte de técnicas y herramientas de inteligencia artificial que nos permitan explorar de manera sistemática grandes cantidades de repositorios de software, identificando qué archivos de código fuente tienen especificaciones de IaC y qué tipo de arquitectura definen estas especificaciones. Sin embargo muchas veces las arquitecturas no están definidas de forma explícita en el código, En muchas ocasiones, las arquitecturas no se encuentran explícitamente en el código, sino que su existencia es implícita y puede derivarse a través de las relaciones entre los componentes definidos en la especificación de la infraestructura como código. Para esto, necesitamos construir una herramienta que no solo nos permita determinar que un código define una IaC, sino que también debe ser capaz de analizar el contenido y determinar si por ejemplo se especifica una infraestructura para una arquitectura de micro-servicios, una arquitectura orientada a eventos (event-driven, por sus sigla en inglés) o si es una arquitectura sin servidor (serverless) u otro tipo de arquitectura. Concretamente, presentamos las siguientes contribuciones:

- Una discusión del concepto de Arquitecturas Implícitas en las especificaciones de IaC.
- Una taxonomía de técnicas de inteligencia artificial usadas para el análisis semántico de código fuente.
- Un proceso definido para la extracción y etiquetado de patrones de arquitectura presentes en archivos de infraestructura como código.
- Entrenamiento y ajuste de 4 modelos de lenguaje en la tarea de clasificación de patrones de arquitectura en proyectos de IaC
- Un experimento de categorización de patrones de arquitectura en especificaciones de IaC con modelos de lenguaje en un dataset desconocido

El proyecto está organizado de la siguiente manera. El capítulo 2 introduce el concepto de Arquitecturas Implícitas. El capítulo 3 presenta una discusión de diferentes técnicas de inteligencia artificial que son útiles para el análisis semántico de especificaciones de IaC. El capítulo 4 presenta el diseño e implementación de un pipeline para la construcción de un conjunto de datos etiquetado usando usando supervizado débil. EL capítulo 5 presenta el proceso de entrenamiendo de los modelos de lenguaje pre-entrenados en código por medio de transferencia de código. Luego el capítulo 6 presenta un experimento de la aplicación de los modelos ajustados para clasificar patrones de arquitectura en un dataset desconocido. Finalmente, las conclusiones se presentan en el capítulo 7.

El código de este proyecto puede encontrarse en github.com/Lufedi/iac-pattern-classification-with-llm y el dataset de entrenamiento junto con los modelos de lenguaje ajustados están disponibles en 10.6084/m9.figshare.23537370

Chapter 2

Arquitecturas Implícitas

En esta sección discutimos el problema del conocimiento implícito en las especificaciones de infraestructura en código. Para esto primero introducimos el concepto de infraestructura como código, luego discutimos el problema del conocimiento implícito, introducimos el concepto de arquitectura implícita y finalmente lo relacionamos con los patrones de arquitectura en especificaciones de código.

2.1 Infraestructura como código

La Infraestructura como Código es un enfoque en la ingeniería de software y la computación en la nube que enfatiza la gestión y aprovisionamiento de recursos de infraestructura utilizando código y automatización. Permite a las organizaciones tratar las configuraciones, implementaciones y gestión de infraestructura como artefactos de software, brindando los beneficios del control de versiones, la repetibilidad y la escalabilidad a la gestión de la infraestructura. Rahman et al. (2019)

La infraestructura como código ha crecido con el tiempo hasta el punto de tener un gran abanico de herramientas disponibles que ayudan a gestionar la infraestructura en la nube Morris (2023).

Al trabajar con infraestructura como código se heredan los mismos beneficios y retos de trabajar con un lenguaje de programación convencional, como lo es el mantenimiento del código, curva de aprendizaje, versionamiento, la posibilidad de inyección de bugs Dalla Palma et al. (2022) y testing Hasan et al. (2020)

Usar código para definir y consolidar la infraestructura de un proyecto de software habilita también los temas relacionados a diseño que encontramos en un lenguaje de programación convencional Guerriero et al. (2019). Podemos hablar de patrones de diseño, mejores prácticas e incluso construcción de librerías o frameworks Dalla Palma et al. (2020).

Nuestra investigación se basó en el procesamiento de proyectos que están usando el framework CloudFormation sobre la librería CDK de Amazon. Este framework representa desde el momento de su lanzamiento en 2019 y al momento de escribir este proyecto, un 20% de las herramientas que usan hoy los desarrolladores y devops para trabajar con proyectos de infraestructura como código. Guerriero et al. (2019)

Podemos decir que el ecosistema que gira alrededor de la infraestructura como código es muy nuevo Guerriero et al. (2019), a pesar de que existen herramientas como Ansible que llegaron en el 2012, el movimiento de la infraestructra como código Morris (2023) no tuvo la gran atención que tiene hoy en día de no ser por el auge de la computación en la nube Guerriero et al. (2019) . Este sistema de rentar un pequeño pedazo de infraestructra a un proovedor y usarla según la demanda, dio el paso a pensar en cómo los usuarios pueden automatizar los procesos para construir,

manipular y destruir estos recursos en la nube de forma eficiente Guerriero et al. (2019).

2.2 Conocimiento Implícito

El conocimiento abstracto almacenado en el código fuente se refiere a la información implícita Land et al. (2001) y experiencia en el dominio incrustados dentro del código base de proyectos de software. Si bien el propósito principal del código fuente es instruir a las computadoras sobre cómo ejecutar tareas, también sirve como un repositorio de conocimiento valioso que los desarrolladores acumulan durante el proceso de desarrollo de software. Este conocimiento abstracto abarca varios aspectos, incluyendo decisiones de diseño, patrones arquitectónicos, convenciones de codificación, optimizaciones de rendimiento y técnicas de solución de problemas.

El framework CDK de Amazon por ejemplo soporta los lenguajes de programación Typescript, Java, Golang, Python y C# Guerriero et al. (2019). Cada archivo escrito usando la librería de CDK en alguno de estos lenguajes, tiene conocimiento implícito que puede ser extraído y analizado. Una de las principales ventajas del conocimiento abstracto almacenado en el código fuente es su capacidad para servir como una forma de documentación. La documentación tradicional, como manuales o especificaciones, a menudo se vuelve obsoleta o no se mantiene de manera efectiva. En contraste, el código fuente actúa como una documentación viva que refleja el estado actual del sistema, que para el caso de la infraestructura como código, refleja el estado actual de los recursos físicos o virtuales sobre los cuales se ejecuta un sistema.

Al analizar el código base, los desarrolladores pueden obtener información sobre la razón detrás de ciertas decisiones de diseño Babar et al. (2005), comprender el flujo

de datos y control, y descubrir suposiciones implícitas sobre el comportamiento del sistema. Además, el conocimiento abstracto almacenado en el código fuente facilita la colaboración y el intercambio de conocimiento entre los miembros del equipo Smite et al. (2019). Cuando los desarrolladores trabajan en un código base compartido, pueden aprovechar la sabiduría colectiva implícita en el código para mejorar su comprensión y toma de decisiones. Los desarrolladores experimentados pueden transferir su conocimiento a otros de forma indirecta a través del código, permitiendo la transferencia de conocimiento y fomentando una cultura de aprendizaje dentro del equipo.

Extraer y utilizar el conocimiento abstracto del código fuente no siempre es sencillo. Requiere que los desarrolladores posean las habilidades y experiencia necesarias para navegar, descifrar de manera efectiva código base complejo y las entender decisiones tomadas en el pasado que justifican lo que existe actualmente. El propio código debe estar bien estructurado, modular y adherirse a las mejores prácticas de codificación para facilitar la extracción de conocimiento.

El conocimiento abstracto almacenado en el código fuente puede contribuir a la evolución y el mantenimiento de los sistemas de software (ver fig 2.1). Si lo relacionamos con la infraestructura como código, el código fuente y su respectivo versionamiento Opdebeeck et al. (2021) es la representación de cómo la infraestructura se ha ido adaptando a los cambios en el tiempo. Estos cambios pueden estar dados por nuevos requerimientos que llegan al sistema, necesidades de escalar una solución o simplemente actualizar componentes con instancias más modernas Kagdi et al. (2007). Los desarrolladores a menudo se enfrentan a la necesidad de modificar o extender la funcionalidad existente. El conocimiento abstracto capturado en el código puede servir como guía para tomar decisiones informadas durante estos cambios. Al comprender los principios de diseño y los patrones utilizados en el código, los

desarrolladores pueden realizar modificaciones más efectivas que se alineen con la arquitectura del sistema y cumplan con los estándares de codificación establecidos.

Reconocer la importancia del conocimiento abstracto en el código fuente y emplear técnicas apropiadas para aprovechar este conocimiento puede mejorar las prácticas de desarrollo de software, mejorar la calidad del código y facilitar el aprendizaje y mejora continua.

2.3 Arquitecturas implícitas

La idea de resolver problemas siguiendo un patrón estructurados fue introducido por Alexander et al. (1977) en 1997. En este estudio se presentan patrones para la construcción de estructuras como pueblos, edificios o cuartos dentro de un edificio. La arquitectura de software es similar, en el sentido que es un diseño a alto nivel de la estructura que uno o varios componentes de software tendrán al momento de implementarse Gamma et al. (2005) Sharma et al. (2015). Esta planificación normalmente se basa en patrones o abstracciones teóricas que ya han sido probadas y han funcionado para resolver ciertos problemas comunes. Un ejemplo de esto son las arquitecturas orientadas a eventos que ayuda implementar proyectos donde existe comunicación asíncrona entre varios sistemas. Los proyectos de infraestructura como código aprovechan los patrones de arquitecturas para implementar soluciones generales y reutilizables para problemas recurrentes respecto a la definición de los componentes de la infraestructura. Un ejemplo de esto son las soluciones basadas en contenedores, que son un patrón muy común para implementar arquitecturas de micro-servicios De Lauretis (2019).

Dentro de los proyectos de infraestructura como código tenemos un conjunto de recursos de infraestructura agrupados y ordenados de tal forma que nos pueden dar

una idea de las decisiones de arquitectura que tomaron los desarrolladores y arquitectos. Derivado de esto hemos definido como arquitecturas implícitas, una posible arquitectura o patrón que siguen estas definiciones para resolver problemas recurrentes que puede ser derivada del código fuente de un proyecto. El conocimiento no implícito, es decir el que reposa en la cabeza de los arquitectos y desarrolladores representa una parte importante en la toma de decisiones de diseño de una arquitectura de software, que según estudios este conocimiento queda en un grupo pequeño del equipo que tiene un gran porcentaje de contribuciones a los diseños de arquitectura Perez. et al. (2021). Con la llegada de la infraestructura como código podemos acercarnos a ese conocimiento tácito que reposa en los equipos de desarrollo de software identificando las arquitecturas implícitas en los proyectos de software.

El conocimiento sobre arquitecturas de software tiene un constante cambio Rahman et al. (2019) y evoluciona con los años Ahmad et al. (2013). La ventaja de tener infraestructura como código es que se puede aprovechar el versionamiento del código para extraer una foto de los recursos de infraestructura Maffort et al. (2013) y recrear la arquitectura implícita en cualquier punto en el tiempo Savidis and Savvaki (2021) .

Existen proyectos que han aprovechado las técnicas de aprendizaje de máquina y los modelos de lenguaje para minar conocimiento implícito en documentos de texto, que dada su naturaleza, comparte ciertas características con el código fuente al ser ambos escritos en un lenguaje con una estructura y reglas "sintácticas". En el trabajo realizado por Becker et al. (2021) se aplican técnica de transferencia de conocimiento a modelos de lenguaje pre-entrenados con base de conocimiento que explica el conocimiento implícito en los documentos de texto. Allí se concluye que las tareas de reconstrucción de conocimiento implícito necesitan una guía muy cuidadosa para obtener resultados coherentes.

Los problemas de la reconstrucción de una arquitectura es que necesitan bastante trabajo manual escaneando los componentes de un sistema de software. Es por esto que necesitamos de herramientas Schmidt et al. (2014) apoyadas en técnicas de inteligencia artificial para extraer las arquitecturas implícitas en proyectos de infraestructura como código.

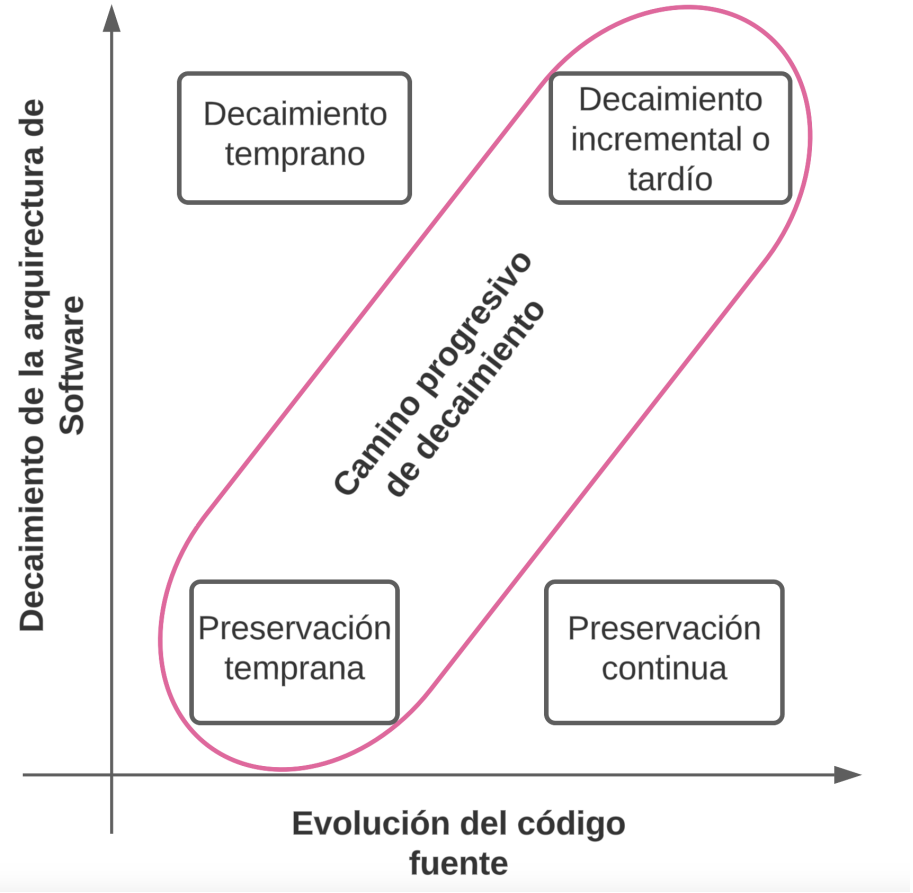


FIGURE 2.1: Evolución del código vs Decaída en la arquitectura de software

2.3.1 Patrones en la nube

Con el surgimiento de la computación en la nube los sistemas empresariales comenzaron a tener una migración a ese ecosistema Wan Mohd Isa et al. (2019). Las

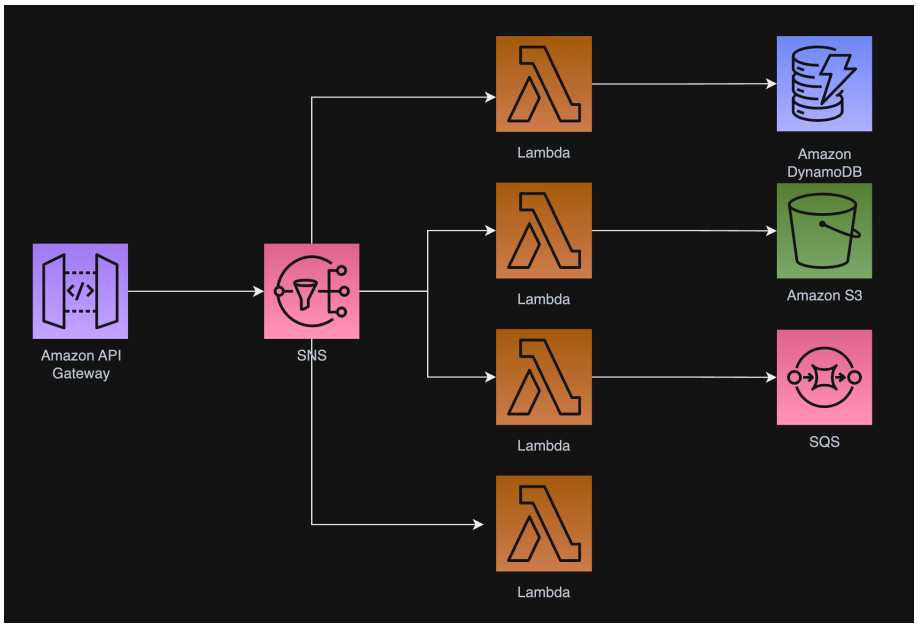


FIGURE 2.2: Ejemplo de arquitectura serverless

empresas buscan aprovechar las ventajas que la nube ofrece tales como escalabilidad, reducción de costos, mantenimiento, seguridad entre otras Keery et al. (2019). En paralelo surgió el concepto de aplicaciones nativas en la nube Linthicum (2017) (Cloud-native applications), que según lo describe el nombre son aplicaciones que están construidas para correr de forma nativa en ambientes cloud. Con la llegada de la computación en la nube y la infraestructura como código como herramienta para desplegar infraestructura de forma automatizada, han surgido diferentes tipo de patrones arquitectónicos Fehling et al. (2014) comúnmente usados al momento de implementar soluciones nativas. Algunos de los patrones que podemos encontrar en la literatura y en la industria son:

- Arquitecturas orientadas a eventos
- Arquitecturas serverless Taibi et al. (2020)
- Arquitecturas big data Mistrik et al. (2017)
- Arquitecturas para IoT Washizaki et al. (2020).

- Arquirecturas de microservicios

Chapter 3

Taxonomía de las redes neuronales y LLM

En esta sección presentamos una taxonomía conceptual para entender el estado del arte en el uso de técnicas computacionales para el análisis e interpretación de código usando aprendizaje de máquina. Para esto, en la primera sección discutimos diferentes formas en la que se puede representar el código para entrenar modelos de inteligencia artificial. Luego, introducimos conceptos de redes neuronales y discutimos diferentes arquitecturas de redes neuronales convolucionales y su uso. Finalmente, presentamos una discusión de trabajos que usan modelos de lenguaje de gran envergadura (Large Language Models, en inglés) para estudiar código fuente y el proceso de refinamiento para ajustar estos modelos a tareas particulares.

3.1 Representaciones de código

La representación de código se compone de las estrategias para transformar código fuente en representaciones que pueden ser usadas para entrenar modelos de inteligencia artificial. En las siguientes secciones exponemos las diferentes técnicas para representar código fuente de forma secuencial o estructurada Siow et al. (2022) en un espacio vectorial de baja dimensión conocido como representaciones vectoriales. Las representaciones vectoriales permiten a los modelos de inteligencia artificial entender los datos de entrenamiento y extraer relaciones entre ellos.

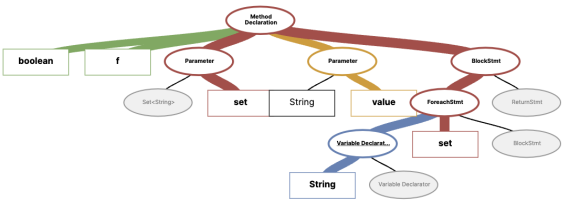
3.2 Árboles de Sintaxis Abstractos

Los árboles de sintaxis abstractos o AST 3.1 son una estructura de datos usada para representar la estructura y semántica de un código fuente. Un AST es una representación jerárquica basada en árboles que captura la estructura sintáctica del código deconstruyéndolo en nodos del árbol. Cada nodo representa un constructor sintáctico en el código, tales como funciones, expresiones, variables, operaciones o asignaciones Zhang et al. (2019).

Los AST se generan en un proceso previo a la compilación del programa ya que son ejecutados como un analizador de código sintáctico. Proveen una representación a alto nivel del código y eliminan detalles como la puntuación y el formato. ASTs capturan las relaciones entre los elementos del código y la relación jerárquica, haciendo fácil el análisis, manipulación y transformación de código de forma programática. Alon et al. (2018b)

```java
boolean f(Set<String> set, String value) {
    for (String entry : set) {
        if (entry.equalsIgnoreCase(value)) {
            return true;
        }
    }
    return false;
}
```

(a) Función en Java.



(b) Representacion AST.

FIGURE 3.1: Representación de la función contains en un árbol abstracto de sintaxis (AST). Generado en code2seq.org

Los AST han comenzado a ser usado como herramienta para crear representaciones vectoriales de código fuente, lo que ha ayudado a solucionar tareas tales como análisis de código estático, refactorización y generación de código Kovalenko et al. (2019). La ventaja de AST es que permiten entender el código por medio de una estructura que preserva la semántica y la intención del código. Las representaciones con AST pueden ser usadas en varios modelos de machine learning para aprender representaciones significativas del código, mucho de esto se ha usado para tareas como detección de bugs, resumen de código, clasificación y generación de código.

3.3 Grafo de flujo

Grafo de flujo es una representación de código que se enfoca en capturar el control de flujo y dependencias de datos dentro de un programa. En una representación basada en graph flow, los constructores del código, tales como funciones o módulos son representados por nodos en un grafo dirigido, y los arcos representan un control de flujo o dependencias entre los constructores (ver fig 3.2).

Los arcos del flujo capturan el orden en el que se ejecuta un programa. Ellos representan el control de flujo entre loops condicionales y llamadas a funciones. Esta

técnica de representación permite a modelos de machine learning entender la lógica del programa y tomar decisiones según los diferentes caminos dentro del grafo.

Los grafos de dependencias de datos capturas las dependencias entre elementos de datos dentro del código. Ellos representan la relación entre las variables, expresiones y computaciones que se efectúan sobre los valores Zeng et al. (2021). Con el modelado de dependencia de datos, las representaciones de grafo de flujo permiten a los modelos de machine learning entender las relaciones entre partes del código y tomar acciones sobre las dependencias entre datos.

La representación con grafo de flujo funciona bastante bien para tareas que requieren entender el comportamiento dinámico del código, tales como tareas de optimización o debugeo. Esta representaciónn permite capturar interacciones y dependencias dentro del código Siow et al. (2022).

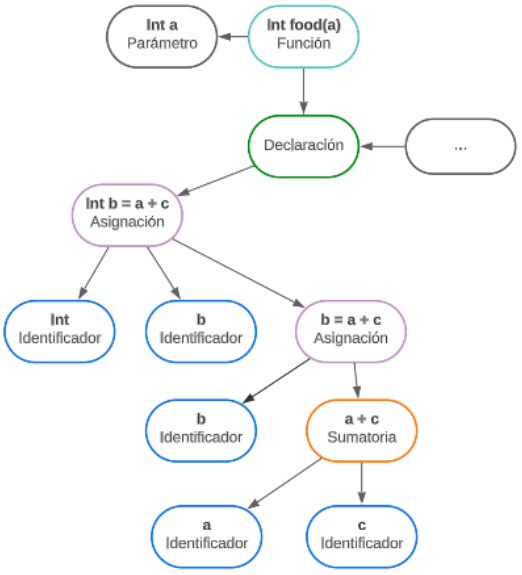


FIGURE 3.2: Representación de una función usando un grado de flujo (GraphFlow)

Los grafos de dependencias de datos capturas las dependencias entre elementos de

datos dentro del código. Ellos representan la relación entre las variables, expresiones y computaciones que se efectúan sobre los valores Zeng et al. (2021). Con el modelado de dependencia de datos, las representaciones de GraphFlow permiten a los modelos de aprendizaje de máquine entender las relaciones entre partes del código y tomar acciones sobre las dependencias entre datos. Grafo de flujo funciona bastante bien para tareas que requieren entender el comportamiento dinámico del código, tales como tareas de optimización o debugging. Esta representación permite capturar interacciones y dependencias dentro del código Siow et al. (2022).

3.4 Sequencia

La representación en secuencia es una forma de representar el código fuente en dónde el código es tratado como una secuencia de tokens o identificadores. En esta representación, el código es tokenizado en elementos individuales (ver fig 3.3) , tales como palabras claves, operadores, variables y valores, y son organizados de forma lineal en una secuencia Niu et al. (2022).

La representación por secuencia captura la secuencia natural de la ejecución del código y el orden en el que las expresiones son ejecutadas. Este provee una forma muy concreta e intuitiva para modela el código aprovechando técnicas como las redes neuronales recurrentes o los modelos transformador. En este tipo de representación cada token o identificador en el código es codificado como un vector de longitud fija. Es utilizado en modelos como LSTM o LSMT bidireccional Schuster and Paliwal (1997), también en modelos con arquitecturas basadas en transformadores Vaswani et al. (2017). Esta representación es enviada a modelos de aprendizaje profundo que procesan secuencias de datos, permitiendo trabajar con tareas tales como generación de código, completar código y resumir código.

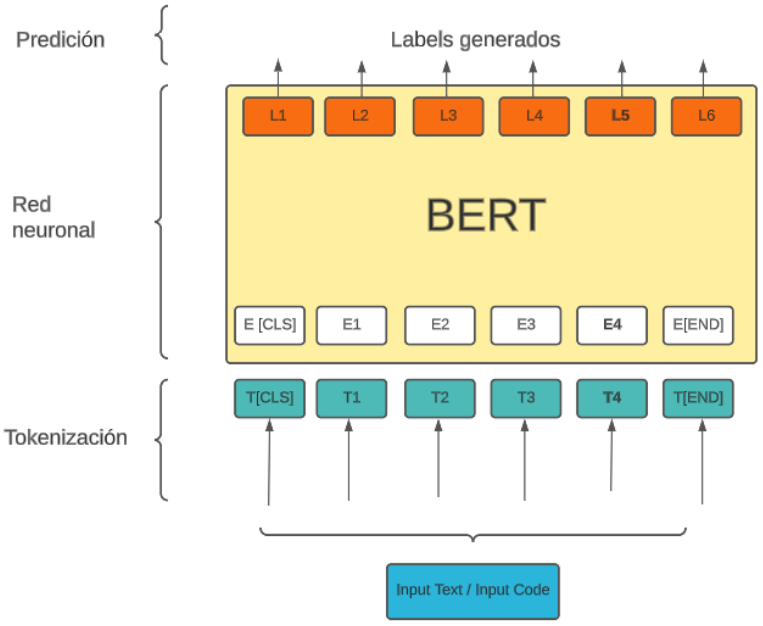


FIGURE 3.3: Tokenizer del modelo BERT para una frase

Las representación por secuencia ha mostrado gran efectividad en capturar dependencias locales y patrones dentro del código, sin embargo pueden llegar a tener problemas capturando dependencias o relaciones complejas en grandes bloques de código, en estos casos representaciones como ASTs o grafos de flujo han demostrado ser mejores.

3.5 Aprendizaje profundo y redes neuronales

Aprendizaje profundo es un subconjunto de aprendizaje de máquine donde se usan redes neuronales para aprender representaciones de datos. Las redes neuronales son el bloque principal sobre el cual están construidos los modelos de aprendizaje profundo Georgousis et al. (2021). El objetivo de los modelos de aprendizaje profundo es permitir a un sistema computacional aprender y tomar decisiones sin la necesidad de una programación explicita. Los algoritmos de utilizados se basan en el poder de

las redes neuronales para aprender automáticamente representaciones de los datos. Una de las ventajas de estos algoritmos es la habilidad de procesar grandes cantidades de datos, lo que las hace resaltar en tareas como detección de imágenes y de voz Fadlullah et al. (2017).

El entrenamiento de una red neuronal normalmente conlleva dos pasos: propagación hacia adelante y propagación hacia atrás. Durante forward propagation, los datos son enviados a través de la red, y las salidas son calculadas capa por capa Du et al. (2016) . En el caso de progragación hacia atrás, se calculan los gradientes de los parámetros del modelo con respecto a una función de perdida, lo que permite actualizar de forma eficiente los parámetros por medio de algoritmos de optimización como el gradiente descendiente estocástico.

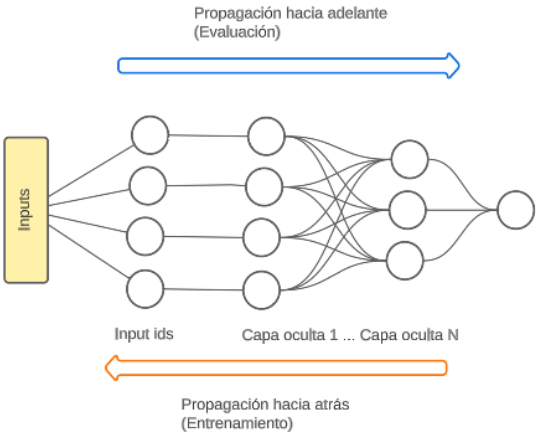


FIGURE 3.4: Propagación hacia adelante y hacia atrás

En el estado del arte encontramos que el aprendizaje profundo ha alcanzado logros significativos que han revolucionado varios dominios. Ejemplos claros incluyen el uso de redes convolucionates(CNN) para reconocimiento de imágenes Hao et al. (2018), redes neuronales recurrentes (RNNs) para el procesamiento de secuencia de datos Sehovac and Grolinger (2020) y modelos transformador para el procesamiento de lenguaje natural Vaswani et al. (2017). Estos avances han liderado el estado del arte

en tareas como detección de objetos, traducción a máquina, análisis de sentimiento y modelos generativos.

Una red neuronal es un algoritmo de machine learning que está estructurado y modelado en base al cerebro humano Shrestha and Mahmood (2019). Se compone de capas que están interconectadas por nodos, o también conocidos como neuronas. Cada neurona procesa y transforma los datos que recibe en probabilidades. La estructura por capas permite que cada neurona reciba como entrada los resultados de la capa anterior y nuevamente se aplican transformaciones matemáticas para producir una salida. Este proceso se repite sobre la secuencia de capas hasta llegar a la capa final donde el resultado es generado.

El bloque principal de una red neuronal son los perceptrones, donde se calcula la suma de los pesos de los datos de entrada y se aplica una función de activación para producir un output. Cada peso determina la importancia de cada input, y la función de activación introduce modelos no lineales Alzubaidi et al. (2021). La combinación de diferentes tipos de perceptrones permiten a las redes neuronales aprender patrones y representaciones cada vez más complejas. Las redes neuronales están compuestas de hasta cientos de capas y cada capa está compuesta por varios perceptrones Goodfellow et al. (2014). Cada capa permite aprender de forma incremental representaciones abstractas de los datos que se reciben como input. La gran ventaja de un modelo de aprendizaje profundo es que puede aprender características de los datos de forma automática.

3.5.1 Arquitecturas de aprendizaje profundo

Una arquitectura de aprendizaje profundo se refiere a la estructura y el diseño de una red neuronal profunda (deep neural network). Define cómo se organizan y conectan

las capas de la red neuronal para procesar la información y realizar tareas específicas de aprendizaje automático Alom et al. (2019).

Una arquitectura de aprendizaje profundo generalmente consta de múltiples capas de neuronas interconectadas, donde cada capa procesa y transforma los datos de entrada para generar una salida final. Cada capa puede contener diferentes tipos de unidades, como neuronas, celdas LSTM o convoluciones, dependiendo de la tarea y el tipo de datos que se están procesando.

3.5.2 Redes neuronales prealimentada

También conocidas como Multi-Layer Perceptrons (MLP), estas arquitecturas constan de una secuencia de capas interconectadas (ver imagen 3.4), donde la información fluye en una dirección, desde la capa de entrada hacia la capa de salida. Las capas ocultas en el medio procesan y transforman los datos de entrada para generar una salida final.

3.5.3 Redes neuronales Convolucionales

Estas arquitecturas se utilizan principalmente para el procesamiento de imágenes y video. Utilizan capas de convolución para extraer características y patrones espaciales de los datos de entrada, lo que las hace efectivas en tareas como clasificación de imágenes, segmentación de objetos y detección de características visuales Gu et al. (2018).

En la clasificación de imágenes las redes neuronales convolucionales vectorizan las imágenes y envían esto como entrada a la red neuronal creando estados ocultos (ver imagen 3.5). Cada capa de la red neuronal selecciona secciones más pequeñas de

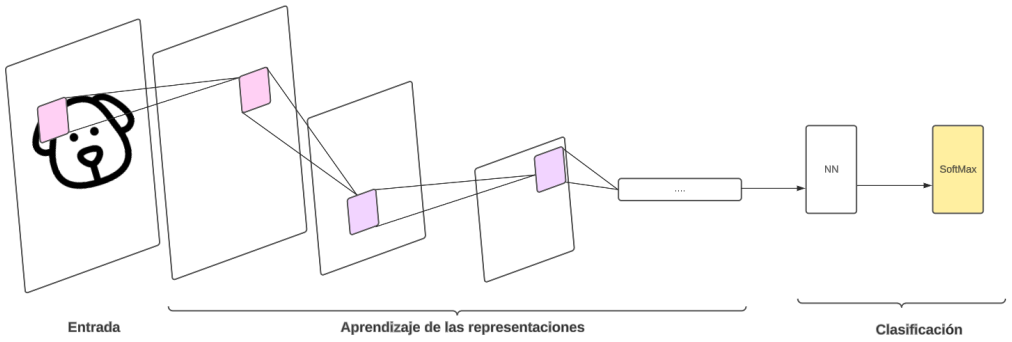


FIGURE 3.5: Arquitectura Redes neuronales convolucionales

la imagen de entrada y aplica un filtro (usando una función de activación) para generar un nuevo vector con nueva información sobre los píxeles de la imagen. Por ejemplo para una imagen de 12x12 pixeles la red neuronal tomara pedazos de 3x3 pixeles y aplicará un producto punto entre el segmento de la imagen y un filtro predeterminado también conocidos como filtros convolucionados, de ahí el nombre de redes convolucionales. Luego se aplican transformaciones o pooling a los resultados de la función de activación para reducir la dimensión de la imagen y encontrar relaciones entre los pixeles (representados por vectores) de la imágen.

3.5.4 Redes neuronales recurrentes

Estas arquitecturas están diseñadas para procesar datos secuenciales, como series de tiempo o texto. Las capas recurrentes tienen conexiones de retroalimentación, lo que les permite mantener una memoria de estados anteriores y capturar dependencias a largo plazo en la secuencia. Son ampliamente utilizadas en tareas como modelado de lenguaje, traducción automática, reconocimiento de voz y generación de texto. Salehinejad et al. (2018)

Las redes neuronales recurrendes calculan y memorizan estados anteriores en el tiempo para calcular el valor actual usando una función de pérdida, en la imagen

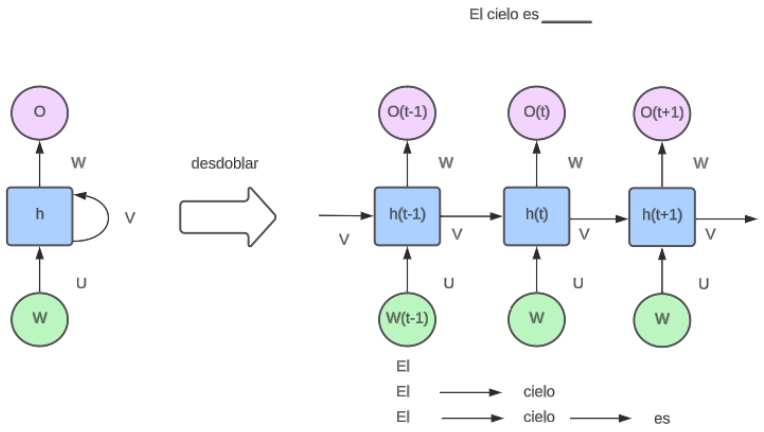


FIGURE 3.6: Arquitectura red neuronal recurrente con estados ocultos

3.6 se muestra cómo una red neuronal recurrente reutiliza los estados ya calculados de una frase antes de precedir la salida de una nueva palabra. Cada estado en el tiempo h es calculado reutilizando los estados ocultos ya calculados. La red neuronal recurrente también hace uso de propagación hacia adelante y en reversa para ajustar los valores de las neuronas.

3.5.5 Redes generativas adversarias

Estas arquitecturas se componen de un generador y un discriminador que compiten entre sí. El objetivo del generador es crear muestras sintéticas que sean indistinguibles de los datos reales Karras et al. (2017), mientras que el discriminador intenta discernir entre las muestras generadas y los datos reales (ver imagen 3.7). El generador construye imágenes aleatorias que pertenecen al dominio de la red neuronal, y el discriminador utiliza una muestra de imágenes como referencia para ajustar su criterio y predecir si una imagen es real o falsa (generada por el generador). Las imágenes de muestra provienen de un conjunto de datos de entrenamiento.

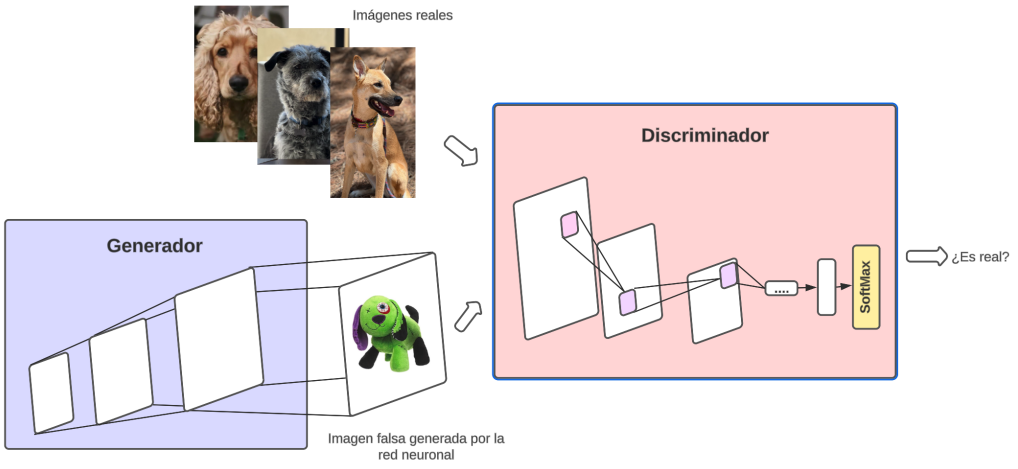


FIGURE 3.7: Arquitectura redes neuronales generativas adversarias

Los modelos generativos se consideran un problema de aprendizaje no supervisado, ya que tanto el generador como el discriminador se entrenan simultáneamente. El generador genera un conjunto de ejemplos, que se combinan con las imágenes reales y se presentan al discriminador para ser clasificados como reales o falsos. Luego, el discriminador actualiza sus parámetros para corregir los errores al identificar las imágenes falsas y reales. Del mismo modo, el generador se actualiza en función de su desempeño en iteraciones anteriores y su capacidad para engañar al discriminador.Una vez que el generador está suficientemente entrenado, el modelo del discriminador se descarta, ya que en este tipo de problemas solo nos interesa el resultado final del modelo generador.

3.5.6 Seq2seq

Seq2seq es una clase de arquitectura de red neuronal diseñadas para mapear una secuencia de entrada a una secuencia de salida. Son particularmente útiles para tareas que generan secuencias tales como traducciones, resumen de texto y sistemas de diálogo (chatbots) Boyanov et al. (2017). Funciona con un codificador que procesa

la entrada transformándola en una representación vectorial con una longitud fija, la cuál es tomada luego por el decodificador para generar la salida Sriram et al. (2017). Un codificador tipicamente está construido usando una arquitectura RNN(ver imagen 3.6), tales como Long Short-term memory (LSTM) Palangi et al. (2016) o Gated Recurrent Unit (GRU) para capturar las dependencias entre los token de las secuencia. cada elemento de la secuencia es enviado a un codificador, donde se genera un estado oculto que captura la información de la secuencia de entrada. A menudo se incorporan mecanismos de atención para abordar la limitación del vector de contexto de longitud fija al permitir que el decodificador se centre en diferentes partes de la secuencia de entrada en cada paso de tiempo. Los mecanismos de atención permiten que el modelo alinee las secuencias de entrada y salida de manera más efectiva.

Además, se han vuelto populares variaciones arquitectónicas avanzadas como los modelos basados en transformador Vaswani et al. (2017) . Los transformadores emplean mecanismos de autoatención para capturar las dependencias entre todas las posiciones de las secuencias de entrada y salida simultáneamente. Han logrado resultados notables en tareas como la traducción automática, alcanzando un rendimiento de vanguardia.

code2vec

Code2Vec es un modelo de aprendizaje profundo diseñado para aprender representaciones distribuidas de código fuente Alon et al. (2018a). Su objetivo es capturar el significado semántico de fragmentos de código al representarlos como vectores de longitud fija. El modelo Code2Vec trata el código fuente como secuencias de identificadores y captura las relaciones entre estos identificadores para aprender representaciones vectoriales que codifican la información contextual del código.

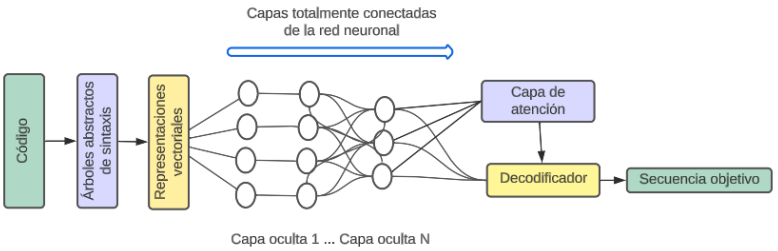


FIGURE 3.8: Arquitectura code2vec

Utiliza una arquitectura de red neuronal que combina CNN y RNN. El modelo toma un fragmento de código como entrada, lo tokeniza en una secuencia de identificadores y representa cada identificador como un vector codificado. Estos vectores de identificadores se pasan luego por una capa convolucional para capturar las dependencias locales dentro del fragmento de código. Los mapas de características resultantes se alimentan en capas recurrentes, como LSTM o GRU, para capturar las relaciones contextuales entre los identificadores.

Durante el proceso de entrenamiento, Code2Vec generalmente se entrena de manera supervisada utilizando un corpus grande de fragmentos de código junto con sus etiquetas o tareas objetivo correspondientes. El modelo aprende a mapear fragmentos de código a sus respectivas representaciones, lo que le permite generalizar y producir incrustaciones significativas para código no visto previamente (ver imagen 3.8).

El modelo Code2Vec se ha aplicado con éxito a varias tareas relacionadas con el código, incluyendo el autocompletado de código, la búsqueda de código, la recomendación de código y detección de bugs Briem et al. (2019). Al representar los fragmentos de código como vectores, Code2Vec facilita la similitud semántica y la comprensión del código, lo que puede mejorar el proceso de desarrollo y ayudar a los programadores a escribir código más eficiente y mantenible.

code2seq code2seq es el modelo sucesor de code2vec Alon et al. (2018a) diseñado

para representación y generación de código fuente. Extiende el concepto del modelo seq2seq al dominio del código fuente. Esto lo logra mapeando código fuente de entrada a código fuente de salida, code2seq logra generar secuencias de código como cuerpos de un método o implementaciones completas de funciones según el contexto o las descripciones dadas como entrada. El modelo code2seq consta de dos componentes principales: un codificador y un decodificador. El codificador toma el fragmento de código de entrada y lo codifica en una representación de longitud fija, similar al modelo Seq2Seq. Sin embargo, en lugar de utilizar un codificador estándar basado en RNN, code2seq utiliza una arquitectura de red neuronal especializada que combina codificadores basados en rutas y basados en AST. Estos codificadores capturan tanto la información estructural del código como las relaciones contextuales entre diferentes partes del código.

El componente decodificador de code2seq genera la secuencia de código de salida en función de la representación codificada del codificador. El decodificador utiliza técnicas como mecanismos de atención y búsqueda de haz para producir la secuencia más probable de tokens de código dados el contexto. El modelo se entrena utilizando aprendizaje supervisado con pares de fragmentos de código de entrada y salida, donde el objetivo es maximizar la probabilidad de generar la salida correcta dada la entrada. Alon et al. (2019).

3.5.7 Modelo transformador:

Los transformadores son arquitecturas basas en atenciónVaswani et al. (2017) propuestas en el paper "Attention is all you need" en 2017. Este paper revolucionó muchas de las tareas que se realizaban sobre procesamiento de lenguaje natural y

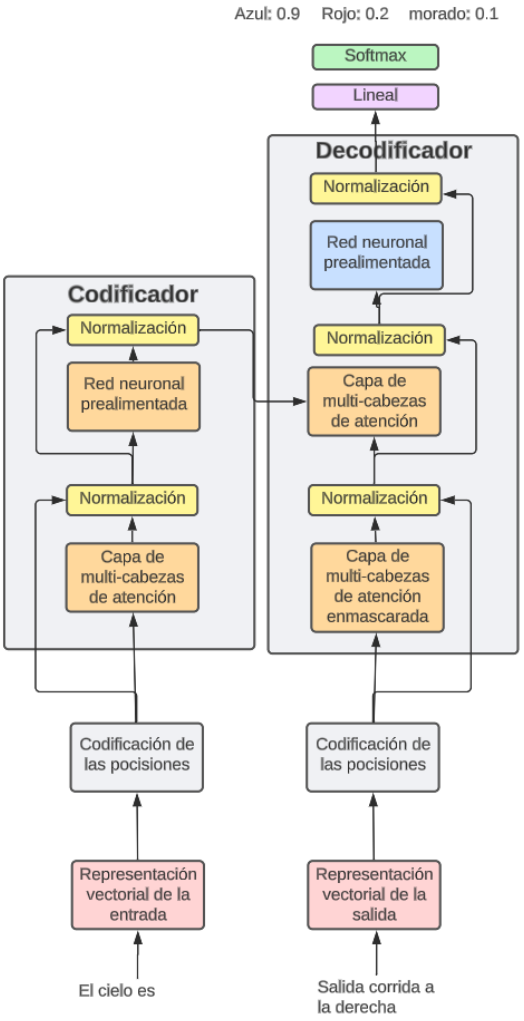


FIGURE 3.9: Arquitectura modelo transformador

ha sido adoptado masivamente en varios dominios relacionados. El modelo transformador es conocido por su capacidad para capturar eficazmente dependencias a largo plazo y manejar datos secuenciales sin depender de redes neuronales recurrentes (RNN).

La innovación clave del modelo transformador radica en su mecanismo de atención y el uso de self-attention(ver imagen 3.9). A diferencia de los modelos de secuencia tradicionales que procesan la entrada de forma secuencial, el transformador utiliza

self-attention para capturar las dependencias entre todas las posiciones de la secuencia de entrada simultáneamente. Esto permite que el modelo pondere la importancia de cada token de entrada en función de su relevancia para otros tokens, lo que resulta en una representación más integral de la entrada.

El modelo transformador consta de un codificador y un decodificador (ver imagen 3.9). Tanto el decodificador como el decorer están compuestos por múltiples capas, y cada capa contiene dos subcapas: un mecanismo de self-attention de múltiples cabezas y una red neuronal de avance(feed-forward) por posición. El mecanismo de self-attention Galassi et al. (2021) de múltiples cabezas permite que el modelo atienda diferentes partes de la secuencia de entrada en paralelo, mientras que la red neuronal aplica una transformación no lineal a cada posición por separado. Esta característica de procesar token en paralelo por separado ha sido uno de los grandes diferenciadones con respecto a las RNN y otros modelos en el estado del arte trabajando sobre tareas de procesamiento de lenguaje natural.

En el codificador, el mecanismo de self-attention permite que el modelo capture las relaciones entre las palabras en la oración de entrada, mientras que la red por posición proporciona transformaciones adicionales. Por otro lado, el decodificador no solo atiende a la secuencia de entrada, sino que también utiliza un mecanismo de atención adicional sobre la salida del codificador. Esto permite que el decodificador considere el contexto de la secuencia de entrada mientras genera la secuencia de salida. El modelo transformador ha logrado un rendimiento de vanguardia en diversas tareas de procesamiento del lenguaje natural, incluyendo la traducción automática, la generación de texto, la respuesta a preguntas y la comprensión del lenguaje Sundararaman et al. (2019). Ha demostrado ser particularmente eficaz en la modelización y generación de secuencias con dependencias a largo plazo, lo

que lo convierte en una arquitectura versátil y poderosa para una amplia gama de aplicaciones basadas en secuencias.

Transformador Generativo Preentrenado: Transformador Generativo Preentrenado (GPT) es un tipo de modelo de aprendizaje profundo que utiliza la arquitectura transformador Vaswani et al. (2017) y se pre-entrena en un corpus grande de datos de texto. GPT fue introducido por OpenAI con sus versiones GPT-2 Radford et al. (2019b), GPT-3Brown et al. (2020) y GPT4 OpenAI (2023), siendo GPT-4 la más reciente. La arquitectura de GPT-3 reutiliza sólo la parte del decoder (ver imagen 3.10) de la arquitectura original del modelo transformador y se puede decir que GPT es una extensión del modelo transformador con un proceso de entrenamiento (ajuste fino) sólamente usando el componente de decodificadores.

Para el caso de GPT-3 se reutiliza la misma arquitectura y modelo de GPT-2 con la diferencia que se incrementa el número de parámetros usados de 125 billones a 175 billones. Durante el pre-entrenamiento el modelo se expone a una gran cantidad de datos de texto no etiquetados, como libros, artículos y páginas web. El objetivo es aprender las propiedades estadísticas y los patrones presentes en los datos. La innovación clave de GPT es que se entrena de manera no supervisada, sin necesidad de etiquetas humanas. La fase de pre-entrenamiento de GPT implica predecir la siguiente palabra en una secuencia dada las palabras precedentes. Esta tarea se conoce como "modelo de lenguaje enmascarado". Al entrenar en esta tarea, el modelo aprende a entender las relaciones contextuales entre las palabras y captura representaciones de lenguaje a un nivel más alto.

Después del pre-entrenamiento, el modelo se ajusta finamente en tareas específicas, como clasificación de texto, análisis de sentimientos o traducción automática. En esta fase, el modelo se entrena con datos etiquetados específicos de la tarea, lo que le permite adaptar sus representaciones previamente aprendidas a la tarea objetivo.

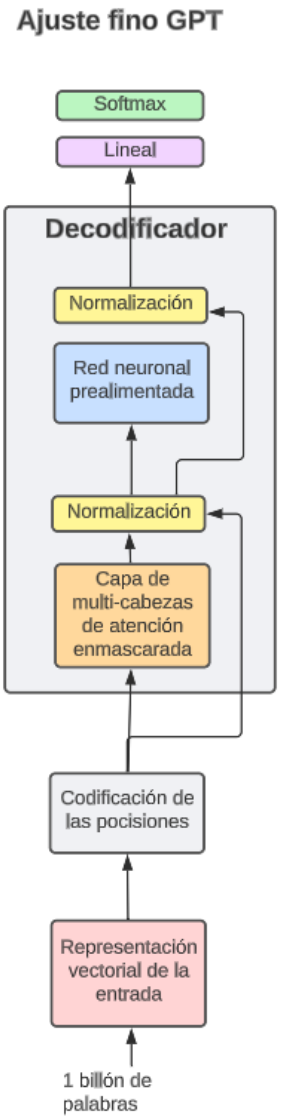


FIGURE 3.10: Arquitectura GPT

Una de las características destacadas de GPT es su capacidad para generar texto coherente y apropiado en contexto. Dado un fragmento o una oración incompleta, el modelo puede generar las siguientes palabras de manera consistente con el estilo y contenido de los datos de pre-entrenamiento. Esto hace que GPT sea útil para aplicaciones como completado de texto, chatbots y generación de contenido.

3.6 Modelos de gran envergadura

Los Modelos de gran enbergadura (LLM) son modelos de aprendizaje profundo que están entrenados en cantidades masivas de datos, normalmente texto, para generar texto entendible por el humano o para llevar a cabo ciertas tareas particulares relacionadas con la generación de texto o secuencias de palabras (tokens). Estos modelos han ganado bastante popularidad gracias a la habilidad que tienen para procesar lenguaje natural.

Los LLM normalmente emplean arquitecturas basadas en transformadores, donde se utilizan mecanismos de atención Vaswani et al. (2017) para capturar las relaciones entre las palabras y crear un contexto. Estos modelos están siendo entrenados en bases de datos gigantescas incluyendo datos corporativos, libros, artículos, sitios webs y cualquier fuente de texto digitalizable para aprender patrones, relaciones semánticas y sintácticas de un lenguaje.

Estos modelos son normalmente entrenados usando aprendizaje no supervisado Radford et al. (2019a), donde aprenden a predecir la siguiente palabra en una secuencia usando como entrada todas las palabras anteriores. Básicamente entienden el contexto de una frase y aprenden de este, lo que ha traído logros muy destacados en tareas de procesamiento de lenguaje natural como la generación de texto, completar texto, análisis de sentimientos y obtención de información. Su gran popularidad ha hecho que se distribuyan en chatbots, asistentes virtuales, generadores de contenido entre muchos más.

Algunos ejemplos a destacar de large language models incluyen OpenAI con sus modelos GPT tales como GPT-3 Brown et al. (2020), GPT-2, GPT-1. Estos modelos han estado en la punta del mundo académico y cada vez más acoplados a la industria gracias a la capacidad de procesar de forma eficientes diferentes tareas de NLP.

3.7 Modelos de lenguaje de gran envergadura Pre-entrenados en código

Los LLM pre-entrenandos también han sido aplicados a dominios de código fuente para mejorar varias tareas relacionadas con código y entender mejor la semántica del código. Estos modelos se apoyan de la inmensa cantidad de código fuente disponible en repositorios open-source, foros, blogs y repositorios privados para identificar patrones, relaciones y conceptos de programación.

Pre-entrenando un modelo en código permite capturar propiedades sintácticas y semánticas del código y aprender a generar fragmentos de código, completar códigos o ejecutar tareas como clasificación o resumen de bloques de código. En la actualidad han tomado bastante popularidad en la comunidad de desarrolladores de software los modelos asistentes que ayudan a generar código, refactorizar o incluso encontrar bugs. Estos modelos pueden usar diferentes tipos de representaciones para entrenar un modelo.

- CodeBERT: es un modelo bimodal (dos tipos de modalidades como entrada) pre-entrenado para lenguaje natural y lenguajes de programación. CODEBERT captura las conexiones semánticas entre lenguaje naturas y lenguajes de programación, y produce representaciones generales que pueden ayudar a resolver tareas de NL-PL (i.e. búsqueda de código con lenguaje natural) y tareas de generación (i.e generación de documentación de código). Fue construido con un transformador de múltiples capas, el cuál es usado en la mayoría de los modelos de lenguaje pre-entrenados. CodeBERT está entrenado en 6 lenguajes de programación usando como conjunto de entrenamiento proyectos de github Feng et al. (2020).

- GraphCodeBERT: un modelo pre-entrenado que incorpora un grafo de flujo de datos para mejorar la representación del código. Considerando dependencias de datos entre los elementos de código GraphCodeBERT logra mejorar el rendimiento del estado del arte en algunas tareas de código como resumen de código y detección de bugs. Guo et al. (2021)

- CodeT5: Es un modelo basado en la arquitectura T5 especialmente para generar código. CodeT5 alcanza el estado del arte en tareas como resumen de código y traducción de código. Wang et al. (2021)

- UnixCode: Es un modelo pre-entrenado que usa matrices de máscaras de atención para modelar el comportamiento del modelo y aprovechar las representaciones como Árboles Abstractos de Sintaxis (AST) y los comentarios dentro del código. UnixCode demuestra buenos resultados en varias tareas de código revelando que los comentarios en el código y la representación con AST pueden mejorar el rendimiento de UIniXcoder. Guo et al. (2022)

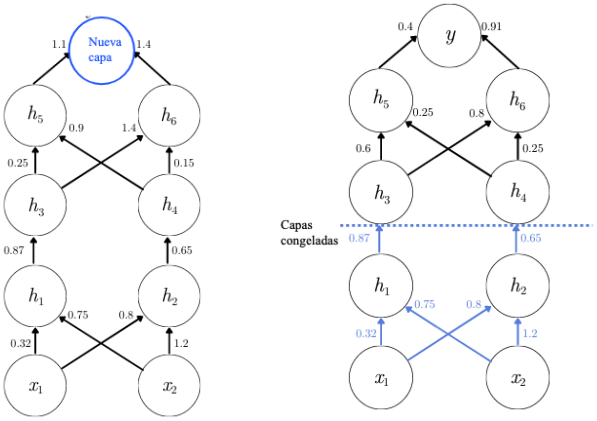
3.7.1 Transferencia de aprendizaje y Refinamiento

Transferencia de conocimiento: es una técnica en la que un modelo que se ha entrenado para una tarea particular se utiliza como punto de partida para un modelo que se usará en una tarea similar Zhuang et al. (2021). Un ejemplo de esto puede ser tomar un modelo que fue entrenado para la identificación de mochilas y reutilizarlo para crear un nuevo modelo que identifica lentes de sol. En este caso se descongelan unas capas del modelo inicial y se dejan aquellas que saben identificar objetos en general, luego se adicionan capas nuevas que se entrenan para detectar lentes de sol.
Refinamiento: también conocido como fine-tunning es una técnica de transferencia

de conocimiento utilizada en el aprendizaje automático para mejorar un modelo pre-entrenado y adaptarlo a una tarea específica Cetinic et al. (2018). Consiste en tomar un modelo ya entrenado en un conjunto de datos masivamente grande y ajustar sus parámetros utilizando un conjunto de datos más pequeño y específico para la tarea que se desea resolver.

3.7.1.1 Congelamiento de capas

Congelar una capa es la estrategia de entrenamiento donde se controla la forma como los pesos son actualizados al momento de ajustar un modelo previamente entrenado. Cuando una capa está congelada los pesos no pueden ser modificados. Esta técnica es normalmente usada cuando se quiere reducir el tiempo computacional durante el entrenamiento sin perder mucha precisión en los resultados Brock et al. (2017). De forma indirecta se congelan capas ocultas de la red neuronal para acelerar el entrenamiento.



(a) Capa truncada (b) Capa congelada

FIGURE 3.11: Técnicas de transferencia de conocimiento sobre redes neuronales de modelos pre-entrenados

Si se considera una red neuronal de 2 capas y se congela la primera capa (ver fig. 3.11) , al ejecutar un entrenamiento con 50 epochs estamos ejecutando una computación

idéntica para la primera capa, es decir, no existe como tan una computación en la primera capa y los pesos no son actualizados. Por defecto los modelos pre-entrenados tienen sus capas congeladas y sólo la última capa puede ser entranada, pero esto puede ser ajustable.

El congelamiento de capas se puede ajustar según el error que estemos dispuestos a tolerar en las predicciones, sacrificando un poco de error por un tiempo menor de entrenamiento Liu et al. (2021). Aunque el refinamiento no es una solución de oro para todos los problemas si hay que tener en cuenta que el dominio de los modelos pre-entrenados para tener mejores resultados.

3.7.1.2 Truncado

Una de las prácticas mas comunes en refinamiento es truncar la última capa del modelo pre-entrenado y reemplazarlo con una nueva cara de tipo softwax específica para el problema Zhang et al. (2022). Si tenemos un modelo que clasifica imágenes en 1000 categorías podemos truncar la última capa y agregar una nueva que reduzca esta categorización a 10 categorías. Después de agregar esta nueva capa se ejecuta el proceso de entrenamiento sobre la red usando los pesos pre-entrenados.

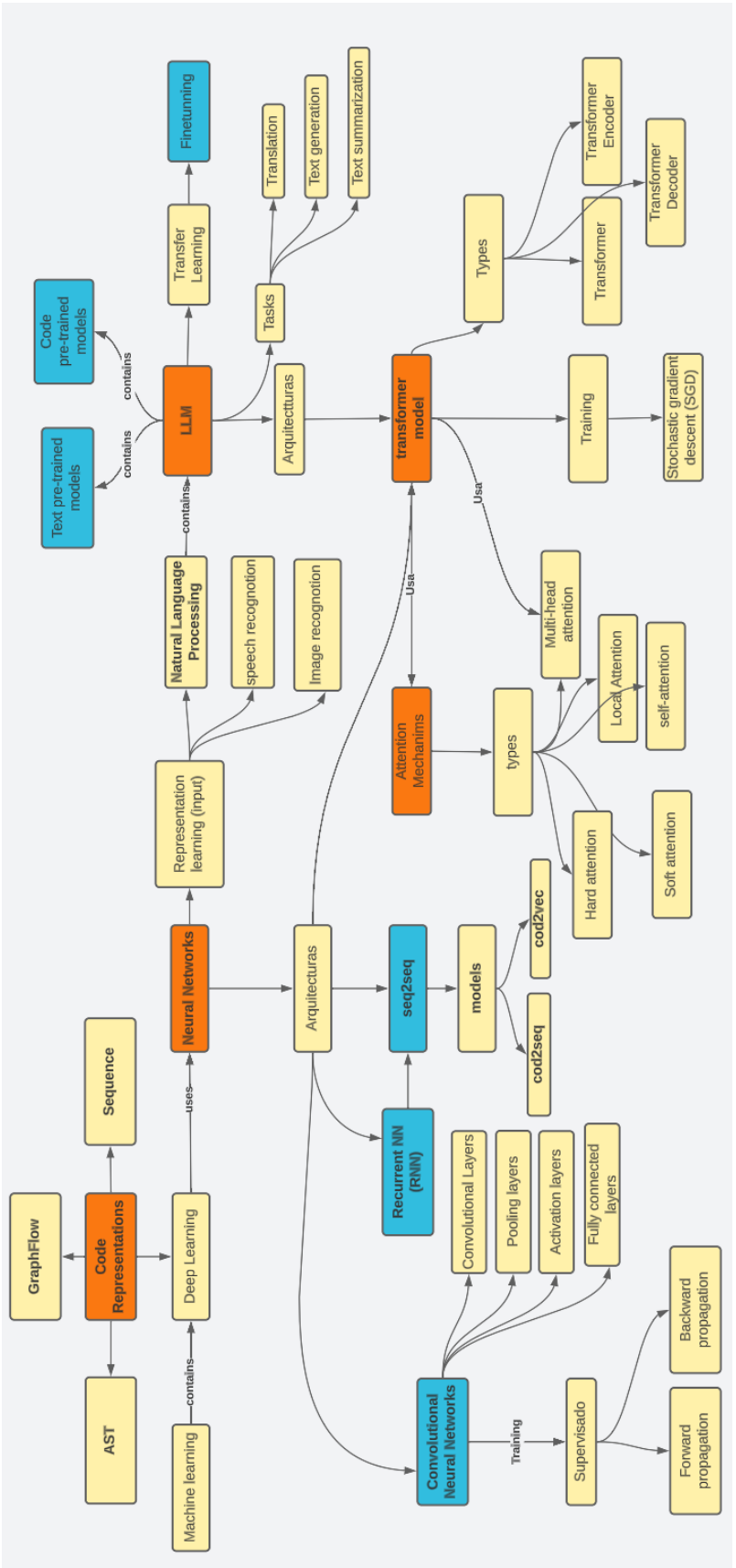


FIGURE 3.12: Taxonomia de las redes neuronales y LLM

Chapter 4

Construcción de un dataset etiquetado con patrones de arquitectura

A continuación, se describe el proceso de minado de archivos que contienen especificaciones de IaC. También se presenta el procedimiento utilizado para mapear y construir un dataset semi-supervizado que ha sido etiquetado con los patrones de arquitectura encontrados en los archivos de IaC.

4.1 Mineria de repositorios

El proceso de minería fue construido en un pipeline (ver 4.1) de extracción de repositorios y refinamiento de modelos pre-entrenados en código para la clasificación de patrones en proyectos IaC. El código de este pipeline puede encontrarse en github y

el conjunto de datos se entrenamiento con los archivos etiquetados están disponibles en figshare

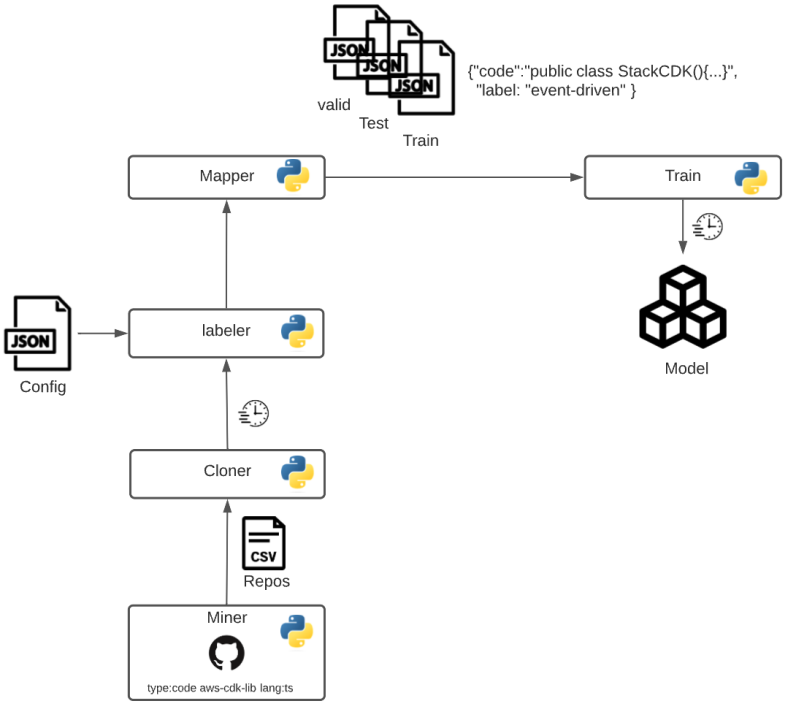


FIGURE 4.1: Pipeline de minería y etiquetado de repositorios

4.1.1 Clonado

Para la extracción del conjunto de datos construimos un pipeline que obtiene repositorios de Github usando su API público. Se utilizaron consultas como *type=code aws-cdk-lib AND language:typescript* para buscar repositorios que hacen referencia CDK (Cloud development Kit) por medio del SDK de AWS. Los lenguajes incluidos en la búsqueda fueron: Java, Python, Go, Typescript. De esto se consiguió un total de 15000 repositorios. 4.1

Lenguaje	**Repositorios**
Typescript	8760
Python	3870
Java	919
Go	401

TABLE 4.1: Distribución de repositorios

4.1.2 Etiquetado

Para el etiquetado se usó supervisión débil. La supervisión débil se refiere a un enfoque de aprendizaje automático donde los datos de entrenamiento son etiquetados o anotados utilizando heurísticas, reglas o fuentes imperfectas de supervisión, en lugar de depender de datos etiquetados manualmente Shin et al. (2021). A diferencia del aprendizaje supervisado tradicional, donde cada punto de datos es etiquetado con precisión por humanos, la supervisión débil utiliza fuentes de supervisión más amplias, menos precisas o más ruidosas.

En la supervisión débil, el proceso de etiquetado generalmente se automatiza o se semi-automática, utilizando técnicas como etiquetado basado en reglas, supervisión a distancia, crowdsourcing o programación de datos. Estos métodos generan etiquetas a una escala más grande y a un costo más bajo en comparación con el etiquetado manual Karamanolakis et al. (2021). Sin embargo, dado que las etiquetas generadas no son completamente precisas o pueden contener ruido, el proceso de aprendizaje debe tener en cuenta estas imperfecciones.

La supervisión débil ha ganado popularidad debido a su potencial para superar las limitaciones de la anotación manual, especialmente cuando se trata de conjuntos de datos grandes o dominios donde obtener etiquetas precisas es desafiante o costoso Rühling Cachay et al. (2021). Permite el uso de cantidades masivas de datos no

etiquetados o parcialmente etiquetados, lo que la convierte en un enfoque atractivo para diversas aplicaciones, incluyendo el procesamiento del lenguaje natural, la visión por computadora y la atención médica, entre otras

Para la tarea de clasificación de patrones de arquitectura cloud escribimos un script en python que implementa un etiquetador para cada lenguaje de programación (ver 4.2), donde escanea cada uno de los archivos de los repositorios clonados y usando un sistema de reglas(explicado más adelante) etiqueta los archivos de cada repositorio en las siguientes categorias: Serverless, Microservices, Big-data, Event-driven, Batch- processing, Internet of things, Streaming, Nosql-storage, Data-warehouse, Data-orch, Object-storage.

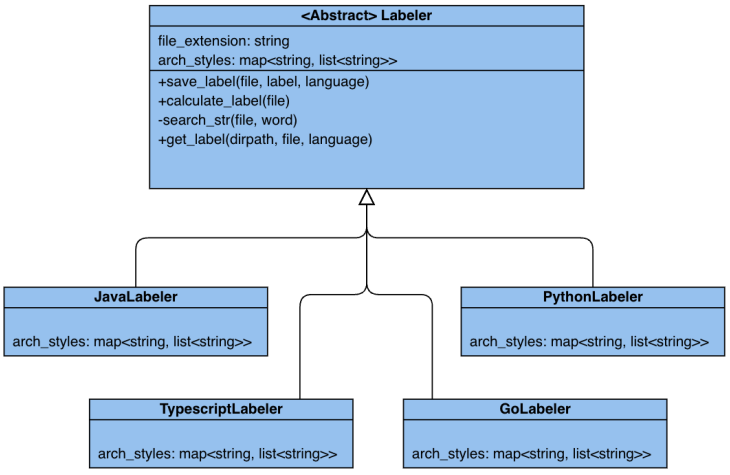


FIGURE 4.2: Clase abstracta Labeler con implementación por lenguaje

Hemos definido un conjunto de reglas usando una relación de uno a muchos entre la el patrón de arquitectura y los componentes de infraestructura cloud que son representativos de esta arquitectura (ver 4.3). Se utilizaron los nombres de los paquetes que contienen los componentes cloud en la libraría de CDK para crear el mapeo. Si alguna de estas librerías es referenciada dentro del archivo de código inspeccionado se etiqueta con el patrón al que pertenece. En caso de que múltiples

paquetes sean referenciados en un mismo archivo se decidió tomar el que mayor número de veces es referenciado dentro del archivo. Para el trabajo realizado no se soporta clasificación multi-label. Sólo una categoría es asignada a cada archivo de código fuente.

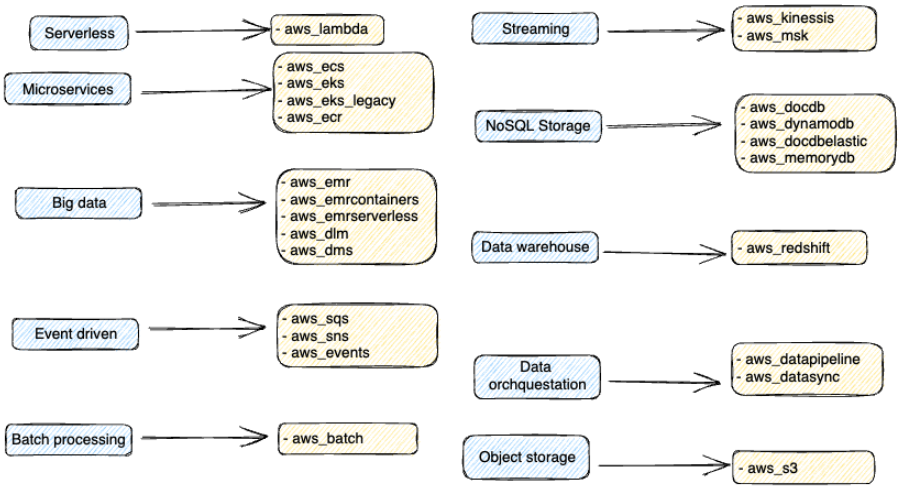


FIGURE 4.3: Reglas usadas en el entrenamiento supervisado débil

En la fase de mapeo se toman los archivos generados por el etiquetado y se copia el contenido de un archivo de código a un objeto JSONL y también se agrega la etiqueta asignada en el paso anterior. Se crean 3 archivos *train.jsonl*, *test.jsonl* y *valid.json*, los cuales contienen una lista de objetos json con las propiedades "code" y "label".

Chapter 5

Entrenamiento de los modelos pre-entrenados en código

5.1 Preparar los modelos pre-entrenados

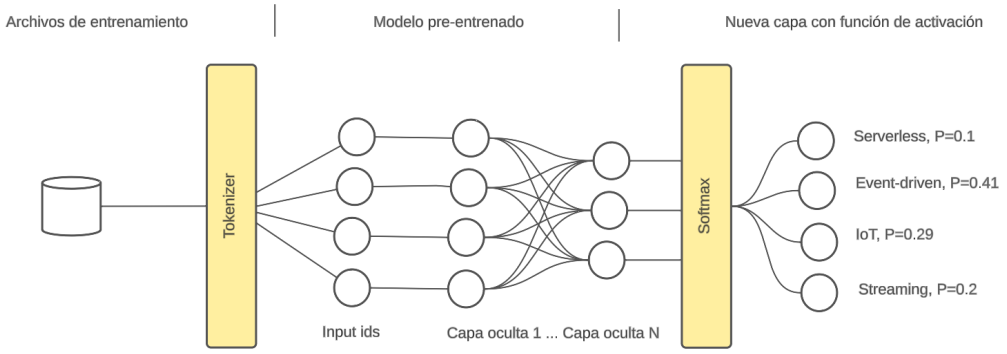


FIGURE 5.1: Estructura de la red neuronal para el ajuste de los modelo preentrenados

Para el entrenamiento, seleccionamos cuatro modelos pre-entrenados en código: CodeBERT, UnixCode, CodeT5 y RoBERTa. Estos modelos utilizan una arquitectura de transformadores multi-cabeza ampliamente utilizada en modelos de gran

escala Kaliyar (2020). Aplicamos la técnica de transferencia de conocimiento con truncado, representada en la imagen 5.1. Agregamos una nueva capa al final de la red para clasificar los archivos en las 11 categorías previamente definidas. Estas categorías se asignan igualmente a números decimales del 0 al 10 y se proporcionan al modelo durante lel entrenamiento más adelante. Para implementar nuestra red neuronal, creamos una clase llamada *Model* que hereda de la clase base para redes neuronales *nn.Module* de PyTorch. Sobrescribimos el método *forward* con una función de activación *SoftMax* Sharma et al. (2017) y utilizamos la función de pérdida *CrossEntropyLoss* (ver código 5.1). La preparación de la red neuronal implica cargar los modelos pre-entrenados en código, junto con su tokenizer correspondiente, luego se crea una instancia de nuestra clase *Model* pasando como parámetros el modelo preentrenado, el tokenizer y un objeto de configuraciones.

```
import torch.nn as nn
class Model(nn.Module):
    def __init__(self, encoder,config,tokenizer,args):
        super(Model, self).__init__()
        self.encoder = encoder
        self.config=config
        self.tokenizer=tokenizer
        self.args=args

    def forward(self, input_ids=None,labels=None):
        logits=self.encoder(input_ids,attention_mask=input_ids.ne(1))[0]
        prob=torch.softmax(logits,-1)
        if labels is not None:
            loss_fct = nn.CrossEntropyLoss(ignore_index=-1)
            loss = loss_fct(logits,labels)
            return loss,prob
        else:
            return prob
// Construir del tokenizer del modelo
tokenizer = RobertaTokenizer.from_pretrained(('microsoft/codebert-base')
// Crear una instancia del modelo usando como punto de inicio el modelo pre-entrenado
model = RobertaForSequenceClassification.from_pretrained('microsoft/codebert-base',config=config)
// Crear una instacia de nuestra clase model pasando como valores el tokenizer y el modelo
```

```
model=Model(model,config,tokenizer='microsoft/codebert-base',{})
```

LISTING 5.1: Creación de una instancia de CodeBERT con una capa adicional de clasificación

5.2 Tokenizando los datos de entrada

Para transformar el conjunto de datos etiquetados en representaciones vectoriales (embeddings), invocamos el tokenizer correspondiente de cada modelo mediante la instrucción `tokenizer.tokenize(code)`. Procedemos a aplicar el tokenizer a cada uno de los archivos de infraestructura como código en el conjunto de datos, lo que genera una secuencia de tokens para cada archivo. Estas secuencias de tokens se leen posteriormente por la red neuronal durante el entrenamiento. Dicha representación del código es una representación *seq2seq* (ver figura 3.3), ya que se trata de una secuencia de tokens derivada de los bloques de código presentes en el conjunto de datos de entrenamiento.

A cada palabra o elemento del código se le asigna un identificador numérico único que forma parte de un vocabulario de tokens. Estos tokens se mapean en un espacio vectorial de n dimensiones según la configuración del tokenizer. En la imagen 5.2 de puede ver que si tokenizamos la palabra *class*, obtenemos el identificador *4423*, que se representa en el espacio vectorial mediante el vector:

$$[0.0069, -0.0037, -0.0042, ...]$$

Además, se agregan dos tokens especiales: uno al inicio (conocido como CLS) y otro al final (conocido como SEP) de la secuencia de tokens, lo cual indica a la red neuronal el comienzo y el fin del código.

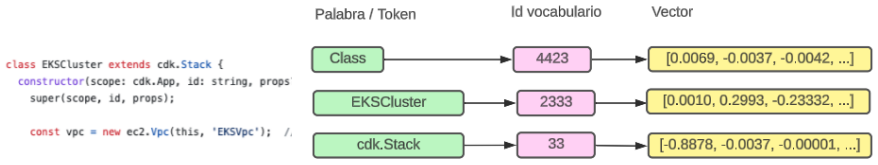


FIGURE 5.2: Reglas usadas en el entrenamiento supervisado débil

Una vez cargado el conjunto de datos de entrenamiento y tokenizado en su totalidad, se crea una instancia del optimizador *AdamW* de la biblioteca transformer. Este optimizador se instancía con una referencia a todos los parámetros expuestos del modelo, lo que permite ajustarlos durante el entrenamiento (ver código 5.2). Durante cada época o ciclo de entrenamiento, el optimizador es responsable de ajustar los parámetros del modelo.

```
optimizer_grouped_parameters = [
    {'params': [p for n, p in model.named_parameters() if not any(nd in n for nd in no_decay)],
        'weight_decay': weight_decay},
    {'params': [p for n, p in model.named_parameters() if any(nd in n for nd in no_decay)],
        'weight_decay': 0.0}
]
optimizer = AdamW(optimizer_grouped_parameters, lr=learning_rate, eps=adam_epsilon)
```

LISTING 5.2: Creación de un optimizer usando los parámetros del modelo

5.3 Entrenamiento

El entrenamiento comienza iterando todo el dataset tokenizado, se toman pequeños grupos del dataset y se envían al modelo invocando la función *train*. Allí se pasan como parametros el modelo precargado, el tokenizer, el optimizados, la función lineal y por último el conjunto de datos que queremos enviar a entrenar (ver imagen 5.3). Después de esto se reciben los valores de pérdida, los cuáles son utilizados por el optimizador para ajustar los parámetros del modelo llamando la función

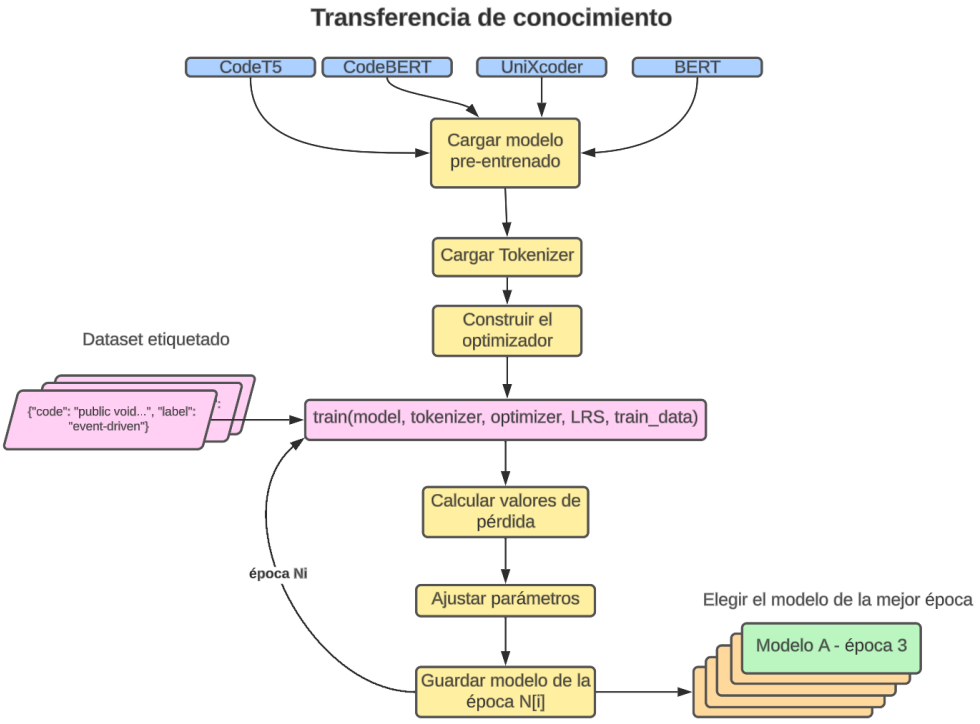


FIGURE 5.3: Entrenamiento del modelo con transferencia de conocimiento

optimizer.step(). Al finalizar una época (epoch) si los valores de pérdida son los más bajos hasta el momento se guarda el modelo en memoria, de esta forma al finalizar todas las épocas se tiene en memoria una copia del modelo con los mejores resultados en el entrenamiento.

```
for idx in range(num_train_epochs):
    bar = tqdm(train_dataloader,total=len(train_dataloader))
    losses=[]
    for step, batch in enumerate(bar):
        inputs = batch[0].to(device)
        labels=batch[1].to(device)
        model.train()
        loss,logits = model(inputs,labels)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm)
        losses.append(loss.item())
        bar.set_description("epoch {} loss {}".format(idx,round(np.mean(losses),3)))
        optimizer.step()
        optimizer.zero_grad()
```

```
        scheduler.step()

    results = evaluate(model, tokenizer, eval_data_file)
    # Save model checkpoint
    if results['eval_acc']>best_acc:
        best_acc=results['eval_acc']
        ...
        torch.save(model_to_save.state_dict(), f'drive/My Drive/models/{save_as_model_name}.bin')
```

Listing 5.3: Entrenamiento y persistencia del mejor modelo ajustado

Todo el entrenamiento de los modelos se lleva a cabo en una GPU con una CPU Intel Xeon a 2.20 GHz, 13 GB de RAM, un acelerador Tesla K80 y 12 GB de VRAM GDDR5 CUDA disponible en Google Colab. Se utilizan los siguientes parámetros:

Parámetro	**Valor**
block size	256
epochs	5
adam epsilon	1e-8
learning rate	2e-5
weight decay	0.1
batch size	2
max gran norm	1.0
eval batch size	16

Table 5.1: Distribución de repositorios

5.4 Evaluación F1-score

Para la evaluación del entrenamiento se utilizó la medida F1 Score, que es comúnmente utilizada en Machine Learning para evaluar el rendimiento de modelos de clasificación binaria y multiclase. Combina las métricas de precisión y reajuste en

un solo valor, proporcionando una medida más completa del desempeño del modelo (ver tabla 5.2). La elección de F1 Score se debe a su capacidad para tener en cuenta tanto los falsos positivos como los falsos negativos, lo que resulta especialmente útil en problemas de clasificación donde se busca un equilibrio entre la precisión y el recall.

TABLE 5.2: F1 score de CodeBERT por catogeoría

Category	**Precisión**	**Recall**	**F1-Score**	**Support**
awsservice	0.98	0.98	0.98	2250
serverless	0.97	0.97	0.97	1738
object-storage	0.94	0.95	0.95	970
microservices	0.96	0.97	0.97	570
event-driven	0.99	0.98	0.99	2009
nosql-storage	0.92	0.96	0.94	389
iot	0.85	0.96	0.90	74
batch-processing	0.90	0.60	0.72	15
streaming	0.80	0.67	0.73	24
data-warehouse	1.00	0.95	0.98	21
big-data	0.65	0.72	0.68	18
data-orch	0.00	0.00	0.00	5
accuracy			0.97	8083
macro avg	0.83	0.81	0.82	8083
weighted avg	0.97	0.97	0.97	8083

Al analizar los resultados del F-score 5.3 obtenidos de entrenar los modelos con una base de 8033 repositorios , es sorprendente observar que el modelo de lenguaje Roberta, entrenado únicamente con lenguaje natural, alcanza un score muy similar al de los demás modelos de lenguaje basados en código. Esto sugiere que, a pesar de las diferencias en su arquitectura o tokenizer, todos los modelos logran capturar características relevantes para la tarea de clasificación en cuestión. Es un indicativo de la capacidad de los modelos de lenguaje preentrenados para adaptarse a diferentes dominios y tareas específicas.

Además, es notable destacar que los porcentajes obtenidos en la evaluación son bastante elevados. Esto indica que el sistema de aprendizaje supervisado débil utilizado en este estudio tiene un impacto significativo en la categorización de los datos. El hecho de que los modelos hayan logrado un alto rendimiento sugiere que el enfoque de supervisión débil aplicado es efectivo y capaz de capturar patrones importantes en los datos de entrada.

Estos resultados respaldan la idea de que los modelos de aprendizaje automático, en particular los basados en modelos de lenguaje pre-entrenados, son herramientas poderosas para abordar tareas de clasificación en diversos dominios. Sin embargo, es importante tener en cuenta que los resultados pueden variar según el conjunto de datos y la tarea específica. Por lo tanto, es necesario realizar evaluaciones exhaustivas y considerar otros factores relevantes al seleccionar y aplicar modelos de aprendizaje automático en contextos reales.

Table 5.3: Resultados obtenidos para los Modelos de lenguaje

Model	**F1 score**
CodeBERT	0.95
UnixCode	0.96
CodeT5	0.93
Roberta	0.96

5.5 Comparación de modelos

Desglosando en F-score para cada uno de los lenguajes de programación (Typescript, Python, Java y Go), podemos obtener las siguientes conclusiones:

CodeBert muestra una alta precisión para Typescript (0.96) y una precisión moderada para Python (0.6). Sin embargo, CodeBert tiene una precisión muy baja para

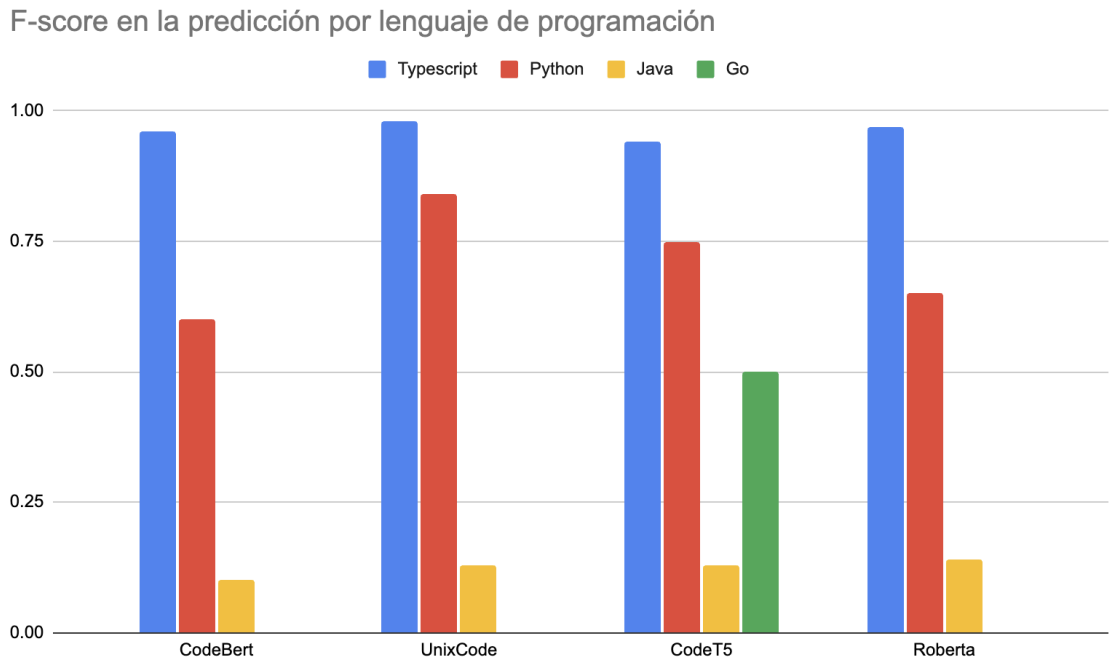


FIGURE 5.4: F-score en la prediccion de los modelos por lenguaje de programación

Java (0.1) y Go (0). Esto sugiere que CodeBert puede tener dificultades para clasificar e identificar con precisión fragmentos de código escritos en Java y Go. Algo muy parecido sucede en UnixCode, CodeT5 y RoBERTA. Lo que demuestra un rendimiento confiable al identificar patrones en Typescript y Python pero muy baja efectividad en Java y Go. Podemos relacionar esta distribución de precisión con la cantidad de datos utilizados en el conjunto de entrenamiento, Java y Go tuvieron la menor representación y por esto los modelos pueden tener dificultades al predecir patrones en las 11 categorías definidas.

Otro punto interesante es Para Java (0.13), CodeT5 muestra una precisión relativamente baja, lo que implica dificultades para clasificar con precisión el código Java. La precisión para Go es 0.5, lo que sugiere que CodeT5 tiene un éxito moderado al identificar instancias de código Go. Roberta:

La gráfica sobre las pérdidas de evaluación (eval_loss) nos puede ayudar a medir la discrepancia entre las predicciones del modelo y los valores reales en el conjunto de

datos de evaluación. Una pérdida más baja indica un mejor rendimiento del modelo. Al analizar los valores de eval_loss en la gráfica 5.5, se pueden destacar varias observaciones. Para el lenguaje Typescript, todos los modelos presentan pérdidas de evaluación relativamente bajas, oscilando entre 0.1 y 0.2. Esto indica un buen rendimiento en la clasificación de patrones para este lenguaje en particular. Todos los modelos muestran resultados comparables en términos de eval_loss para Typescript. En el caso de Python, se observa un aumento en las pérdidas de evaluación para todos los modelos, con valores que varían entre 1.3 y 2.4. Esto sugiere que los modelos tienen dificultades para clasificar patrones en Python en comparación con Typescript. Se observa una diferencia considerable en las pérdidas de evaluación entre los modelos, con CodeBERT y Roberta mostrando pérdidas más altas que Unix y CodeT5.

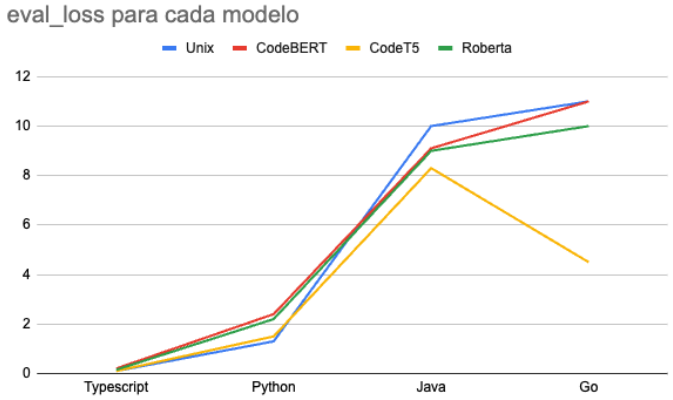


FIGURE 5.5: Función de pérdida para los modelos por lenguaje de programación

Para el lenguaje Java, se encuentran pérdidas de evaluación más altas en general, con valores que oscilan entre 8.3 y 10. Esto indica un rendimiento inferior de los modelos en la clasificación de patrones para Java en comparación con los otros lenguajes analizados. Nuevamente, se observa una diferencia en las pérdidas de evaluación entre los modelos, siendo CodeT5 el modelo con la pérdida más baja.En el caso de Go, todos los modelos muestran pérdidas de evaluación relativamente

altas, con valores que oscilan entre 10 y 11, indicando un rendimiento deficiente en la clasificación de patrones para este lenguaje. Se observa una consistencia en las pérdidas de evaluación entre los modelos, con valores similares en todos ellos.

En general, se puede inferir que los modelos tienen un mejor rendimiento en la clasificación de patrones para Typescript en comparación con Python, Java y Go. El bajo eval_loss en Typescript indica una mejor capacidad para reconocer y clasificar patrones en este lenguaje en particular. Sin embargo, se observa una variabilidad en las pérdidas de evaluación entre los modelos, lo que sugiere diferencias en su rendimiento y efectividad en la tarea de clasificación de patrones.

Estos resultados de eval_loss proporcionan información valiosa sobre el desempeño de los modelos en la clasificación de patrones para diferentes lenguajes de programación. Pueden ser utilizados para identificar áreas de mejora y optimización en el entrenamiento y ajuste de los modelos, así como para guiar futuras investigaciones en la clasificación de patrones en el contexto de lenguajes de programación específicos.

Se relasta la necesidad de investigar y mejorar la capacidad de los modelos para clasificar con precisión el código escrito en lenguajes de programación específicos, especialmente aquellos con puntajes de precisión más bajos. Además, puede ser beneficioso explorar modelos específicos del dominio o estrategias de ajuste fino para mejorar el rendimiento en cada lenguaje de programación.

Chapter 6

Aplicación de modelos ajustados a un conjunto de datos desconocido

Para la aplicación de los modelos ajustados, se realizó la selección de un grupo de proyectos de infraestructura como código que no formaron parte del conjunto de datos de entrenamiento original. Con el fin de obtener este conjunto de datos adicional, se llevó a cabo una búsqueda específica en el sitio web github.com. Se utilizó la siguiente consulta de búsqueda: "cdk created:>2023-05-01", que recupera los repositorios creados recientemente en GitHub que hacen uso de CDK como una biblioteca de desarrollo de infraestructura como código. Además, se aplicó un filtro adicional para incluir proyectos escritos en lenguajes de programación como Java, Python, Typescript y Golang, que son los lenguajes más comunes utilizados con CDK.

Una vez obtenida esta colección de repositorios relevantes, se corrió el mismo pipeline de procesamiento de datos utilizado en el capítulo 3. Esto incluyó las etapas de minado, clonado y mapeo, con el objetivo de obtener los archivos necesarios para

el análisis. El proceso de minado permitió recopilar una cantidad significativa de archivos, los cuales se distribuyeron como se muestra en la table 6.1:

TABLE 6.1: Distribución de archivos para evaluación de modelos

Lenguaje	**Número de archivos**
Typescript	2491
Python	375
Java	75
Go	2

Estos números reflejan la cantidad de archivos recolectados para cada uno de los lenguajes de programación específicos. Es importante destacar que esta muestra adicional de proyectos de infraestructura como código provenientes de GitHub amplía la diversidad y variedad de datos utilizados en la evaluación de los modelos ajustados. Esto proporciona una oportunidad para evaluar la capacidad de generalización de los modelos entrenados y verificar si son capaces de manejar diferentes casos y lenguajes de programación en el ámbito de la infraestructura como código.

6.1 Resultados

El resultado de aplicar los modelos ajustados se muestra en la tabla 6.2. La distribución de la categorización en el conjunto de datos desconocido refleja diferentes patrones. El más visible de todos es un alto número de archivos clasificados en la categoría de *awsservice*, esta categoría representa los archivos que contienen componentes de infraestructura en AWS pero no se pudo identificar un patrón de arquitectura. En promedio esta categoría abarca un 64% de los resultados para cada modelo, lo que significa que, el sistema de reglas usado para aplicar un etiquetado con supervisado débil puede mejorarse para cubrir estos archivos que no han logrado ser categorizados. La categoría de *awsservice* también puede ser aprovechada para

TABLE 6.2: Categorización de patrones aplicando modelos pre-entrenados ajustados con transferencia de conocimiento

Categoría	**CodeBert**	**UnixCode**	**Roberta**	**CodeT5**
awsservice	1764	1903	1850	2100
serverless	311	308	312	162
object-storage	268	228	245	211
microservices	201	158	191	111
event-driven	244	183	181	121
nosql-storage	94	94	96	121
iot	27	33	25	65
batch-processing	10	9	5	14
big-data	1	0	7	7
data-warehouse	14	11	14	14
streaming	7	13	10	10
data-orch	2	3	7	7

explorar qué otros componentes de infraestructura están siendo utilizados en la comunidad de código libre, así se puede robustecer el sistema de reglas propuesto en nuestra investigación para relacionar componentes de infraestructura con patrones de arquitectura. Así mismo, según los componentes encontrados en los archivos de esta categoría se pueden definir nuevos patrones de arquitectura o extender los ya propuestos.

Por otro lado los 4 modelos distribuyen de forma significativa el conjunto de datos entre las siguientes categorias:, *serverless, object-storage, microservice, event-driven y no-sql storage*. Muy parecido a los patrones detectados en el conjunto de entrenamiento, estas 5 categorias son las más predominantes. Podemos evidenciar que entre la comunidad de código libre para los proyectos de CDK estos son los patrones de arquitecturas más comunes.

Dentro de la modernidad de la ingeniería de software, las arquitecturas basadas en microservicios han mantenido su popularidad y poco a poco las arquitecturas asíncronas y orientadas a eventos son cada vez más utilizadas. Sin embargo, las

soluciones basadas en componentes sin-servidor o serverless están tomando bastante fuerza en las soluciones cloud. Se puede ver que el patrón de arquitectura **serverless** siempre tiene la mayor cantidad de proyectos (excluyendo la categoría *awsservice*), esto nos muestra que hoy en día cada vez más soluciones son construidas en estas arquitecturas sin-servidor para aprovechar los beneficios que ofrece Yussupov et al. (2021), la flexibilidad, costos y facilidad de mantenimiento están haciendo de este tipo de arquitecturas unas de las más utilizadas al menos en AWS con CDK.

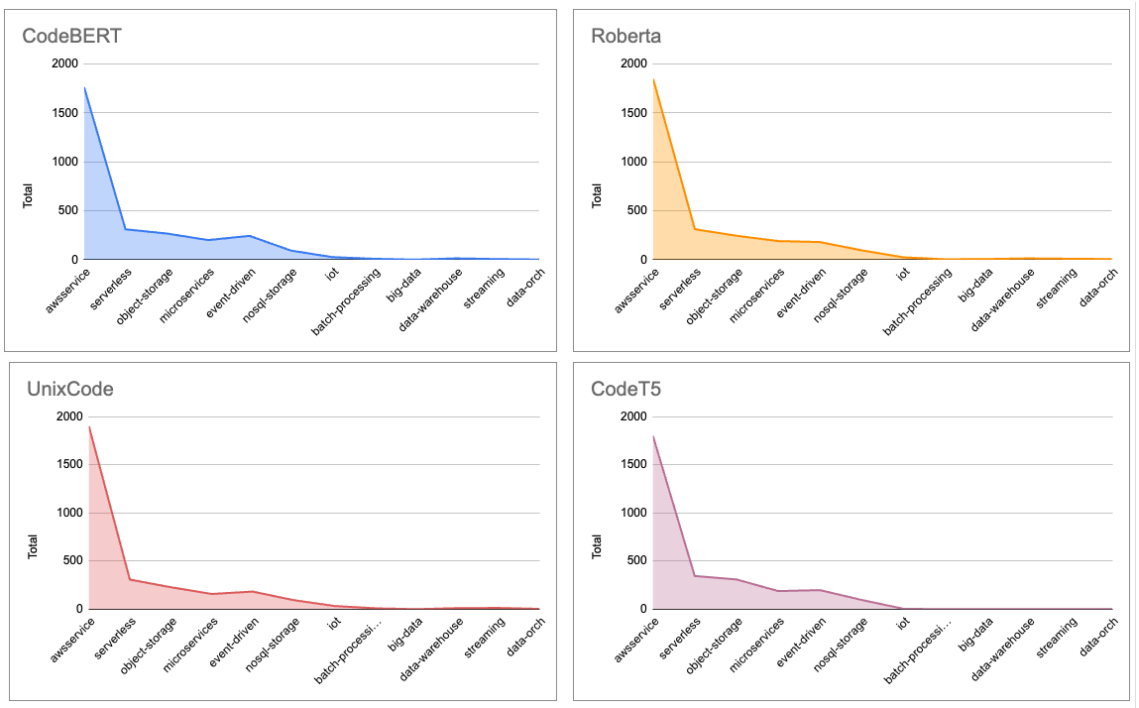


FIGURE 6.1: Categorización de patrones de arquitectuira en un dataset desconocido por modelos ajustados con trasnferencia de conocimiento.

También se puede evidenciar una baja representación de las demás categorías, especialmente la categoría de *big-data* sorprende ya que es un poco contradictorio con el gran movimiento revolucionario de la inteligencia artificial y la ciencia de datos. Esto puede ser también una señal para revisitar el sistema de reglas definido en el capítulo 4 (ver figura 4.3) y evaluar si los componentes asignados a la categoría de *big-data* puede complementarse. Otra conclusión puede ser que los proyectos de IaC

en la comunidad de código abierto no están muy involucrados con soluciones orientadas a ciencia de datos o inteligencia artificial. Esto puede llegar a tener sentido ya que las infraestructuras para este tipo de soluciones no tienen la misma madurez que las arquitecturas convencionales (microservicios o sin-servidor). Muchas de las soluciones ofrecidas por los proveedores cloud en temas de inteligencia artificial o ciencia de datos es expuesto como software como servicio y no como infraestructura como servicio, como es el caso de SageMaker para AWS Joshi (2020).

4 modelos han presentado una distribución muy similar (ver imagen 6.1), se podría decir que casi que idéntica ignorando las diferencias menos relevantes. Roberta al ser un modelo entrenado en sólo fuentes de texto aisladas de código tuvo un desempeño muy similar a los otros modelos entrenados en código, lo que nos da un indicio de que los modelos de lenguaje pueden comportarse de forma muy similar en problemas de categorización de código independientemente del pre-entrenamiento de cada uno. La transferencia de conocimiento logra hacer converger a los 4 modelos en un punto muy similar, el ajuste de los parámetros de cada modelo se ha dirigido en la misma dirección.

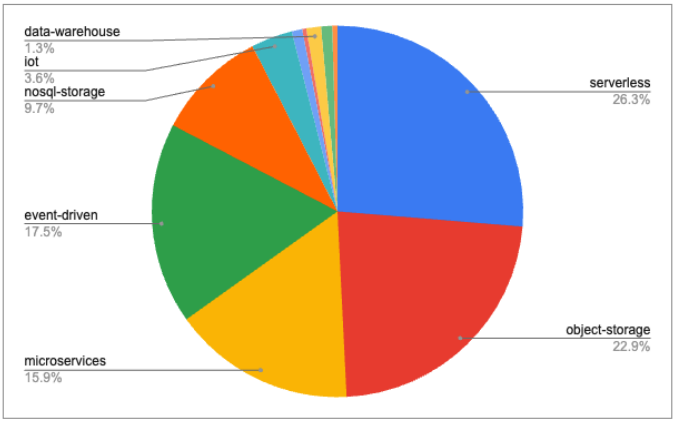


FIGURE 6.2: Distribución porcentual de los patrones de arquitectura excluyendo awsservice

Si excluimos el 64% de los datos que caen la categoría *awsservice*, podemos ver una distribución más realista de los patrones encontrados (ver figura 6.2). Según los porcentajes proporcionados, se observa que "serverless", "object-storage", "microservices" y "event-driven" son las categorías más prominentes en términos de distribución. Por otro lado, "batch-processing", "big-data", "streaming" y "data-orch" tienen una distribución muy baja en comparación con las categorías principales.

6.2 Transferencia de conocimiento

Para validar las mejoras en la clasificación de patrones, llevamos a cabo una comparación exhaustiva entre los modelos antes de la transferencia de conocimiento y las instancias de los modelos ajustados. La tabla 6.3 muestra los resultados de clasificar los patrones de arquitectura antes de aplicar transferencia de conocimiento a los modelos pre-entrenados en código.

TABLE 6.3: Comparison of Results for Categorias, CodeBERT, UnixCode, Roberta, and CodeT5

Categorias	**CodeBERT**	**UnixCode**	**Roberta**	**CodeT5**
awsservice	0	222	0	0
serverless	0	79	0	1639
object-storage	0	470	0	284
microservices	0	139	0	212
event-driven	0	138	0	815
nosql-storage	0	94	0	2034
iot	0	37	0	72
batch-processing	0	39	0	3
big-data	2943	155	2943	0
data-warehouse	0	496	0	0
streaming	0	22	0	0
data-orch	0	1052	0	0

Si se grafica la distribución de las categorías no existe una distribución tan marcada como sucede después de aplicar transferencia de conocimiento (ver imagen 6.3).

Es interesante de igual forma cómo algunas relaciones se presentan en los resultados. Por ejemplo CodeBERT y Roberta no generan ninguna clasificación, todos los archivos son categorizados en la misma categoría, esto puede deberse a que CodeBERT extrae mucha de su implementación de Roberta, lo que puede darnos una idea de que el tokenizer y la arquitectura de la red neurona del CodeBERT es muy similar a la Roberta.

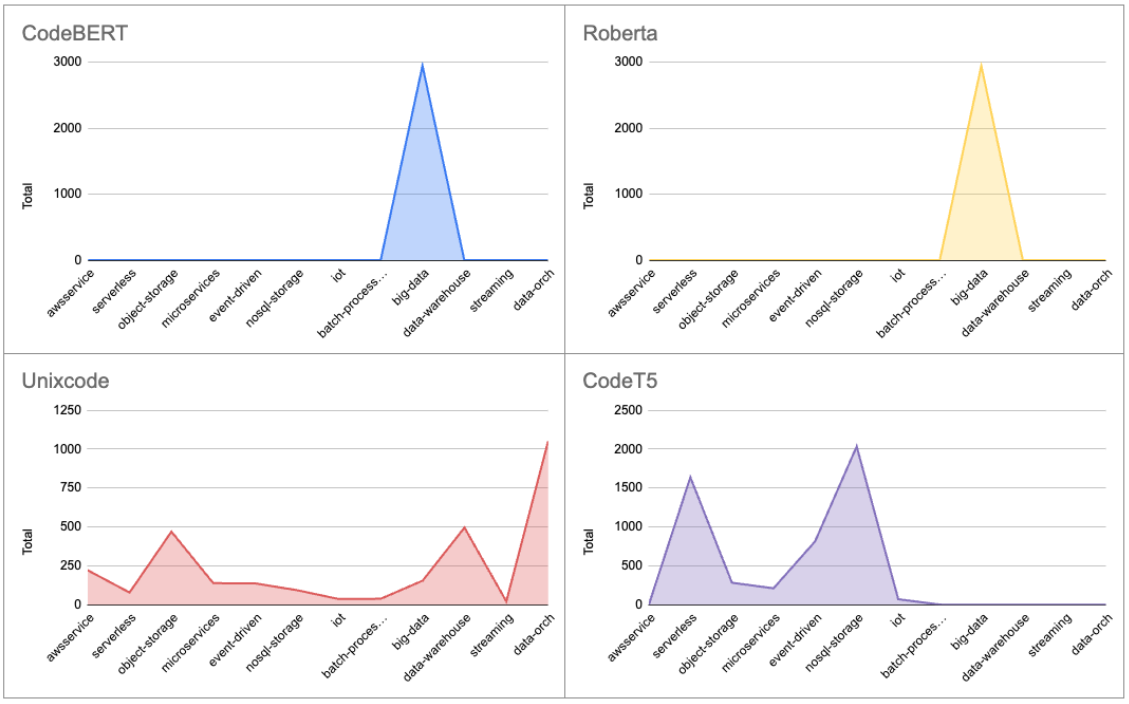


FIGURE 6.3: Categorización de patrones de arquitectuira en un dataset desconocido por modelos pre-entrenados sin ajustar

Para el caso de Unixcode y CodeT5 vemos una categorización más marcada, a pesar de no haber un patrón en común los modelos tienen la capacidad de ofrecer respuestas más diversas. Para el caso de UnixCode, la distribución es muy aleatoria y un poco ladeada a la categoría *data-otch* u orquestación de datos. Por el lado de CodeT5 es interesante ver cómo a pesar de no tener ningún entrenamiento previo, CodeT5 marca una similitud con los modelos ajustados con transferencia de conocimiento. La distribución de CodeT5 está inclinada hacia las categorias más

relevantes encontradas en por los modelos ajustados, lo cual es algo sorprendente, a pesar de que no existe una relación directa, si se logra evidenciar que CodeT5 construye estos clusters sin ningún tipo de entrenamiento previo.

Claramente existen diferencias en el rendimiento de los modelos ajustados en comparación con los modelos solamente pre-entrenados. Los modelos pre-entrenados tienen un desempeño deficiente en la categorización de patrones debido a la falta de reglas o pistas específicas sobre el proceso de clasificación de los patrones. Esto enfatiza la importancia y el impacto crucial de la transferencia de conocimiento en tareas específicas (downstream tasks) al extender los modelos pre-entrenados a tareas particulares.

Los resultados muestran que el ajuste de los modelos pre-entrenados tiene un impacto significativo en su capacidad para clasificar patrones en tareas específicas. Se evidencia una desmejora sustancial en la efectividad de los modelos antes de ser ajsutados, lo que resalta la importancia de la transferencia de conocimiento para mejorar y adaptar los modelos pre-entrenados a tareas particulares en el ámbito de la clasificación de patrones.

6.3 Los lentes que nos permiten ver más allá del código

El proceso de diseccionar y comprender aproximadamente 13,200 archivos de Infrastructure as Code (IaC) ha permitido desvelar las decisiones de arquitectura tomadas por ingenieros, arquitectos y desarrolladores de software. Mediante un sistema de reglas sencillo, conocimiento empírico y herramientas de inteligencia artificial, hemos

logrado construir "lentes" que nos brindan una visión más allá del código estático almacenado en un repositorio Land et al. (2001). Estos "lentes" nos permiten extraer información que anteriormente solo existía en la mente de aquellos que diseñaron e implementaron el código Smite et al. (2019).

Es fascinante observar cómo la comunidad de código libre se ha convertido en un referente para comprender la dirección que está tomando la industria del software. La distribución de patrones identificados en el experimento proporciona una comprensión real del conocimiento tácito presente en el mundo tecnológico actual. Este tipo de experimentos nos permite dar un paso atrás y obtener una visión panorámica. Actualmente, la comunidad se mueve en torno a ciertos patrones arquitectónicos y componentes de IaC, pero en el futuro, podríamos repetir el experimento en unos 20 años, usar los mismos "lentes" para extraer conocimiento implícito y obtener una imagen actualizada de cómo los ingenieros están construyendo software.

Los modelos de inteligencia artificial se han revelado como una herramienta valiosa para explorar el conocimiento tácito presente en los proyectos de software. En nuestro experimento, hemos logrado extraer conocimiento relacionado con las arquitecturas en la nube en proyectos desconocidos utilizando modelos de lenguaje preentrenados y ajustados. La transferencia de conocimiento se convierte en un medio para compartir conocimiento entre diferentes herramientas de inteligencia artificial, permitiéndonos aprovechar el conocimiento existente en un dominio y aplicarlo en otro, lo que nos permite abordar preguntas más complejas y sofisticadas.

Chapter 7

Conclusiones y Trabajo futuro

A través del análisis de 14,000 archivos de Infrastructure as Code (IaC), se ha logrado extraer conocimiento sobre las decisiones de arquitectura tomadas por profesionales del software. Mediante un sistema de reglas, conocimiento empírico y herramientas de inteligencia artificial, se ha construido una metodología que permite comprender el conocimiento implícito en el código y obtener una visión más amplia de los patrones arquitectónicos y componentes utilizados en la comunidad de código libre. Los modelos de inteligencia artificial han demostrado ser útiles para explorar este conocimiento en proyectos desconocidos, utilizando transferencia de conocimiento para resolver cuestiones más complejas y compartir información entre diferentes herramientas.

Tomando los resultados del etiquetado y la aplicación de los modelos a un dataset desconocido, identificamos que los patrones más utilizados en la comunidad open source (Github) son *event-driven*, *serverless*, *microservicios* y *object storage*. Estos patrones son fundamentales en el desarrollo de aplicaciones modernas y han sido ampliamente adoptados por los comunidad de código libre en sus proyectos de infraestructura como código.

Durante la fase de entrenamiento todos los modelos lograron alcanzar un F1-score alto. Esto nos dice que la técnica de transferencia de conocimeinto por truncado es bastante efectiva para entrenar y ajustar los parámetros del modelo para la tarea particular que hemos desarrollado durante la investigación. Esto refuerza la efectividad de la transferencia de conocimiento en los problemas de clasificación usando modelos de lenguaje de gran enbergadura.

Se realizó una comparación exhaustiva entre los modelos antes de la transferencia de conocimiento y los modelos ajustados para validar las mejoras en la clasificación de patrones. Los resultados de clasificación antes de aplicar la transferencia de conocimiento mostraron una distribución menos marcada en las categorías, lo que indica que los modelos pre-entrenados no lograron clasificar de manera efectiva los patrones de arquitectura. Sin embargo, se observaron relaciones interesantes, como la similitud entre CodeBERT y Roberta debido a su implementación compartida. Por otro lado, UnixCode y CodeT5 mostraron una categorización más marcada, con CodeT5 demostrando una distribución inclinada hacia las categorías más relevantes identificadas en los modelos ajustados, a pesar de no haber sido pre-entrenado. Estas diferencias resaltan la importancia del ajuste fino de los modelos pre-entrenados para tareas específicas, ya que se evidencia una mejora significativa en el rendimiento de los modelos ajustados.

Además, observamos que el lenguaje de programación predominante utilizado en CDK es Typescript, seguido de cerca por Python. Esto coincide con la tendencia en la comunidad de desarrollo de código libre redmonk, donde Typescript ha ganado popularidad debido a su tipado estático y su integración con las principales bibliotecas y frameworks. Esta información es relevante, ya que nos permite entender mejor el contexto en el que se aplican los modelos de clasificación de patrones y nos proporciona una visión más precisa de los lenguajes de programación y las tecnologías

más utilizadas en proyectos de infraestructura como código.

En general, los resultados obtenidos sugieren que aplicar transferencia tiene un efecto importante en el rendimiento de los modelos. En todos los casos analizados, los modelos experimentaron mejoras sustanciales después del proceso de ajuste fino. Esto resalta la importancia de ajustar u optimizar los modelos para alcanzar su máximo potencial, especialmente al trabajar en tareas secundarias, como la clasificación de patrones en este caso. Las mejoras significativas observadas indican que el enfoque de entrenamiento supervisado débil utilizado resultó en modelos más efectivos para las tareas y conjuntos de datos específicos analizados.

Este estudio ha demostrado que la combinación de la técnica de representación de código *seq2seq* junto con modelos pre-entrenados basados en RoBERTa ha logrado un buen rendimiento en la clasificación de patrones de arquitectura. Estos hallazgos son prometedores y pueden ser aplicados en diversas áreas donde la clasificación sea una necesidad, proporcionando una base sólida para futuras investigaciones y mejoras en este campo de estudio.

Al ajustar los modelos mediante el truncado de capas, pudimos aprovechar el conocimiento pre-entrenado de los mismos y adaptarlos específicamente a la tarea de clasificación de patrones. Utilizamos la técnica de truncado de la última capa, lo que permitió que los modelos aprendieran características específicas de la tarea y mejoraran de manera significativa su rendimiento en la clasificación de patrones. La comparación entre los resultados antes y después del refinamiento deja en claro la importancia del proceso de calibración y refinamiento para mejorar las capacidades de los modelos en esta tarea.Los modelos pre-entrenados sirven como un punto de partida valioso en el proceso de transferencia de conocimiento. Al contar con un conocimiento general adquirido durante el pre-entrenamiento, los modelos pueden ser ajustados y especializados para adaptarse a dominios o tareas específicas. Esto significa que

los modelos no parten de cero, sino que tienen una base sólida sobre la cual construir conocimiento especializado. Permite un aprendizaje más dirigido y específico al contexto, lo que permite a los modelos adaptarse y destacarse en la clasificación de patrones. Estos resultados y enfoques pueden guiar investigaciones y aplicaciones futuras en el uso de modelos pre-entrenados y técnicas de refinamiento para lograr un mejor rendimiento en diversas tareas de clasificación.

7.1 ¿Qué porcentaje de arquitecturas se lograron evidenciar en los repositorios open source?

Las arquitecturas basadas en eventos tienen la mayor cantidad con 34% de ocurrencias. Esto indica que las arquitecturas y procesos basados en eventos son prevalentes en el conjunto de datos y probablemente desempeñen un papel crucial en la gestión y procesamiento de datos en la nube. Las arquitecturas serverless representan el 29.9% lo que sugiere que la computación serverless está tomando cada vez más fuerza en la computación cloud, que va muy bien de la mano con arquitecturas orientadas por eventos. Así mismo el almacenamiento de objetos representa 16% que muestra una gran tendencia a utilizar AWS como una plataforma de almacenamiento de objetos, algo que también va muy relacionado en general con todas las arquitecturas. Por otro lado podemos ver que las arquitecturas basadas en microservicios o contenedores tienen un porcentaje más bajo, sorprendentemente entre los datos de entrenamiento y evaluación sólo representa un 9.9% ver imagen 7.1.

7.2 ¿Existe alguna relación entre los patrones encontrados y la metadata de los repositorios?

Se encontró que un gran porcentaje de los proyectos minados de github están escritos en typescript, a pesar de que lenguajes como Python y Java ocupan los primeros puestos en algunos rankings globales, como por ejemplo el ranking de redmonk redmonk. Typescript destaca en el framework de CDK (ver tabla 4.1). Esto nos da una idea de cómo la comunidad de Typescript y NodeJs continua adaptando nuevas herramientas y tendencias constantemente, especialmente en proyectos de infraestructura como código.

7.3 ¿Existe alguna relación entre los recursos de infraestructura y los patrones de arquitectura en la nube?

Los resultados nos permiten ver que si existen patrones en el uso de los recursos de la nube y que hay algunos recursos que son más usados que otros, como por ejemplo S3, lambas y SQS son algunos de los recursos que más hemos evidenciado en CDK. Podemos decir que al momento de implementar una arquitectura de software existen muchas posibilidades y decisiones que tomar sobre el tipo de implementación, pero con los datos minados de los 13950 repositorios y el sistema de reglas construido empíricamente nos ayuda comprobar que un patrón de arquitectura puede ser identificable por los componentes de infraestructura que lo componen. Los ejemplos más visibles son el caso de Serverless y event-driven, para el primero las funciones

lambda son el corazón de este patrón, es por esto que puede ser fácilmente identificable a través de este recurso, para el caso de event-driven sucede parecido con los recursos SNS y SQS que permiten el procesamiento de eventos dentro de AWS. Los resultados obtenidos muestran un alto porcentaje de uso de estos recursos, lo que nos da una idea de los patrones cloud que más están siendo adoptados hoy en día por la comunidad open source.

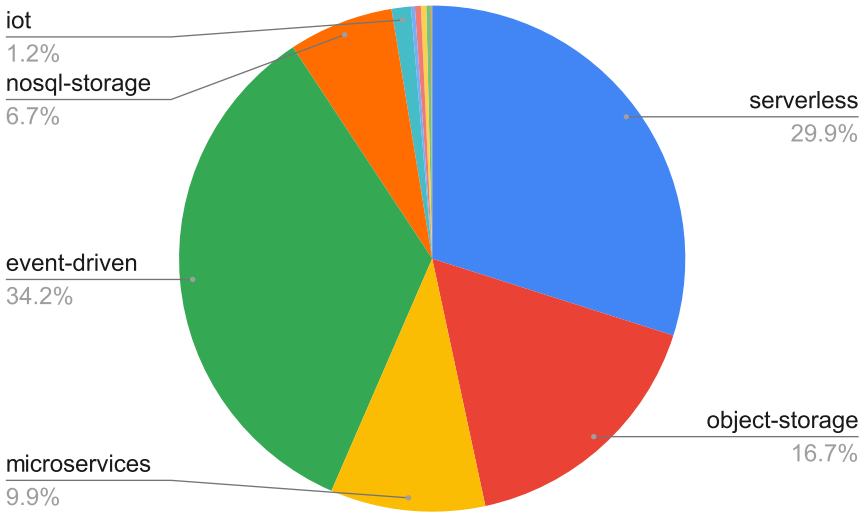


FIGURE 7.1: Porcentaje de patrones en los repositorios

El conocimiento implícito en el código fuente puede ser ilimitado y existe una gran área de oportunidad para explorar otros ángulos o áreas de conocimiento. Se puede tomar como ejemplo la investigación realizada por Feitosa et al. (2023) donde se trata de determinar que can consciente es un desarrollador en los costos de la infraestructura como código por medio de la minería de código fuente. Este estudio busca una relación entre los cambios realizados en repositorios open source y los costos en el tiempo respecto a la infraestructura.

En nuestra investigación trabajamos usando la técnica de representación por secuencia, queda abierta la oportunidad a experimentar con otros tipos de representación de código como AST y GraphFlow, y aplicar estas representaciones en tareas de

clasificación de patrones. También queda abierta la puerta a comparar diferentes tipos de modelos, más allá de modelos basados en RoBERTa se pueden explorar modelos basados en GPT u otras arquitecturas de redes neuronales.

7.4 Riegos de validación

Nuestro análisis está soportado en un entrenamiento basado en supervisión débil, lo que puede causar un descablanceo en los resultados según las reglas que hemos definido según el criterio y experiencia del equipo. Esto sin duda es un factor de riesgo a nuestros resultados.

Otro riesgo es que nuestra fuente de datos está basada solamente en repositorios open source, donde, la calidad de los proyectos puede tener una calidad muy diferente y tener divergencias por ejemplo con código privado. A pesar de que hemos identificado patrones usados en proyectos open source no podemos generalizar este mismo resultado para otro tipo de proyectos en ecosistemas diferentes.

Un análisis adicional podría implicar comparar el costo computacional o el tiempo de entrenamiento requerido para el proceso de refinamiento y evaluar la capacidad de generalización de los modelos en diferentes conjuntos de datos o tareas. Además, las métricas específicas utilizadas para medir el rendimiento (por ejemplo, precisión, puntuación F1, etc.) también serían factores relevantes a considerar en un análisis exhaustivo de la efectividad de los modelos.

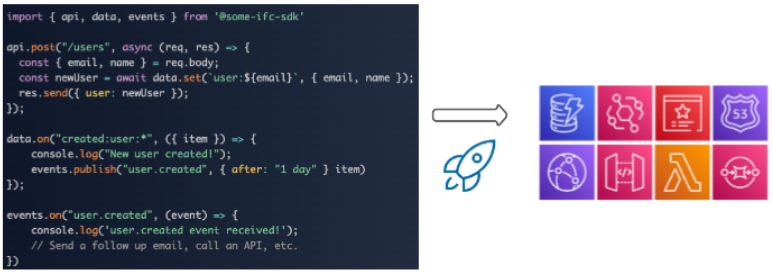


FIGURE 7.2: Proceso de infraestructura desde código

7.5 Infraestructura DESDE código

Hace poco ha comenzado a tener fuerza un nuevo concepto en el mundo de los Devops conocida como **"Infraestructura desde el código"** Aviv et al. (2023), este concepto lleva la infraestructura como código a un punto más abstracto. La infraestructura desde código es la evolución lógica de la nube, no es necesario escribir código de bajo nivel ni instrucciones para desplegar un grupo de recursos de infraestructura, la infraestructura desde código infiere los requerimientos desde la lógica de la aplicación y provisiona de forma óptima los recursos (ver imagen 7.2). Este nuevo concepto busca crear disrupción en la forma como de despliega y mantiene la infraestructura de una aplicación. Ya existen en el mercado algunos SDKs que permite la codificación de servicios y procesamiento de datos ammpt. Este puede ser un punto de partida para más estudios. La infraestructura desde código rompe la barrera que existe entre el código y la infraestructura, los creadores de aplicaciones no necesitarán conocimiento en el dominio de la infraestructura y reducirá la probabilidad de errores manuales Aviv et al. (2023).

En el futuro Se podrían hacer estudios sobre cómo mejorar la inferencia de la infraestructura usando técnicas de machine learning, hacer comparaciones contra las infraestructuras generadas por los SDKs como Nitric o Ampt y construir modelos que busquen mejorar las inferencias. Al ser un problema relacionado con código

puede apoyarse también en modelos de lenguaje basados en código y experimentas con diferentes configuraciones.

7.6 Soluciones LLM existentes

Otro trabajo futuro puede ser la comparación de los modelos que hemos entrenado con transferencia de conocimiento frente a modelos lingüísticos existentes en el mercado. Como Github Copilot de Microsoft, ChatGPT de OpenAI o CodeWhisper de Amazon **?**. El desarrollo de modelos lingüísticos entrenados en código está avanzando bastante rápido y un trabajo futuro puede ser evaluar el rendimiento de nuestros modelos frente a otros modelos que no tengan un entrenamiento especializado en la detección de patrones de arquitectura en la nube.

7.7 Extensión del conocimiento implícito

Otra área de oportunidad es sin duda buscar otras áreas de conocimiento en el mundo de la ingeniería de software que puedan ser exploradas en proyectos de infraestructura como código. En nuestra investigación estábamos particularmente interesados en el dominio de las arquitecturas en la nube, pero quedan temas muy visibles sin explorar como el análisis estático de errores Dalla Palma et al. (2022), la generación de código (pruebas unitarias, plantillas de arquitectura, etc.), anti-patrones Borovits et al. (2020), modelos que sugieren un conjunto de arquitecturas para un conjunto de requisitos funcionales y no funcionales. La lista podría continuar, pero poder escribir infraestructura en términos de código abre muchas puertas que quizá ya se hayan explorado a lo largo de los años, pero no se han centrado en el nicho de la infraestructura como código.

References

Ahmad, A., Jamshidi, P., Pahl, C., 2013. A framework for acquisition and application of software architecture evolution knowledge URL: https://researchrepository.ul.ie/articles/journal_contribution/A_framework_for_acquisition_and_application_of_software_architecture_evolution_knowledge/19854751.

Alexander, C., Ishikawa, S., Silverstein, M., 1977. A Pattern Language: Towns, Buildings, Construction. Center for Environmental Structure Berkeley, Calif.: Center for Environmental Structure series, OUP USA. URL: https://books.google.com.mx/books?id=hwAHmktpk5IC.

Alom, M.Z., Taha, T.M., Yakopcic, C., Westberg, S., Sidike, P., Nasrin, M.S., Hasan, M., Van Essen, B.C., Awwal, A.A.S., Asari, V.K., 2019. A state-of-the-art survey on deep learning theory and architectures. Electronics 8. URL: https://www.mdpi.com/2079-9292/8/3/292, doi:10.3390/electronics8030292.

Alon, U., Brody, S., Levy, O., Yahav, E., 2019. code2seq: Generating sequences from structured representations of code. arXiv:1808.01400.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2018a. code2vec: Learning distributed representations of code. arXiv:1803.09473.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2018b. A general path-based representation for predicting program properties. arXiv:1803.09544.

Alzubaidi, L., Zhang, J., Humaidi, A.J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., J., S., Fadhel, M.A., Al-Amidie, M., Farhan, L., 2021. Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions - journal of big data. URL: https://doi.org/10.1186/s40537-021-00444-8.

ammpt, . The future of cloud development - Ampt — getampt.com. https://www.getampt.com/blog/introducing-ampt/. [Accessed 15-Jun-2023].

Aviv, I., Gafni, R., Sherman, S., Aviv, B., Sterkin, A., Bega, E., 2023. Infrastructure from code: The next generation of cloud lifecycle automation. IEEE Software 40, 42–49. doi:10.1109/MS.2022.3209958.

Babar, M., Gorton, I., Jeffery, R., 2005. Capturing and using software architecture knowledge for architecture-based software development, in: Fifth International Conference on Quality Software (QSIC'05), pp. 169–176. doi:10.1109/QSIC.2005.17.

Becker, M., Liang, S., Frank, A., 2021. Reconstructing implicit knowledge with language models, in: Workshop on Knowledge Extraction and Integration for Deep Learning Architectures; Deep Learning Inside Out.

Borovits, N., Kumara, I., Krishnan, P., Palma, S.D., Di Nucci, D., Palomba, F., Tamburri, D.A., van den Heuvel, W.J., 2020. Deepiac: Deep learning-based linguistic anti-pattern detection in iac, in: Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation, Association for Computing Machinery, New York, NY, USA. p. 7–12. URL: https://doi.org/10.1145/3416505.3423564, doi:10.1145/3416505.3423564.

Boyanov, M., Koychev, I., Nakov, P., Moschitti, A., Martino, G.D.S., 2017. Building chatbots from forum data: Model selection using question answering metrics. arXiv:1710.00689.

Briem, J.A., Smit, J., Sellik, H., Rapoport, P., 2019. Using distributed representation of code for bug detection. arXiv:1911.12863.

Brock, A., Lim, T., Ritchie, J.M., Weston, N., 2017. Freezeout: Accelerate training by progressively freezing layers. arXiv:1706.04983.

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. Language models are few-shot learners. arXiv:2005.14165.

Cetinic, E., Lipic, T., Grgic, S., 2018. Fine-tuning convolutional neural networks for fine art classification. Expert Systems with Applications 114, 107–118. URL: https://www.sciencedirect.com/science/article/pii/S0957417418304421, doi:https://doi.org/10.1016/j.eswa.2018.07.026.

Dalla Palma, S., Di Nucci, D., Palomba, F., Tamburri, D.A., 2022. Within-project defect prediction of infrastructure-as-code using product and process metrics. IEEE Transactions on Software Engineering 48, 2086–2104. doi:10.1109/TSE.2021.3051492.

Dalla Palma, S., Di Nucci, D., Tamburri, D.A., 2020. Ansiblemetrics: A python library for measuring infrastructure-as-code blueprints in ansible. SoftwareX 12, 100633. URL: https://www.sciencedirect.com/science/article/pii/S2352711020303460, doi:https://doi.org/10.1016/j.softx.2020.100633.

De Lauretis, L., 2019. From monolithic architecture to microservices architecture, in: 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 93–96. doi:10.1109/ISSREW.2019.00050.

Du, X., Cai, Y., Wang, S., Zhang, L., 2016. Overview of deep learning, in: 2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC), pp. 159–164. doi:10.1109/YAC.2016.7804882.

Fadlullah, Z.M., Tang, F., Mao, B., Kato, N., Akashi, O., Inoue, T., Mizutani, K., 2017. State-of-the-art deep learning: Evolving machine intelligence toward tomorrow's intelligent network traffic control systems. IEEE Communications Surveys & Tutorials 19, 2432–2455. doi:10.1109/COMST.2017.2707140.

Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P., 2014. Cloud computing patterns. 2014 ed., Springer, Vienna, Austria.

Feitosa, D., Penca, M.T., Berardi, M., Boza, R.D., Andrikopoulos, V., 2023. Mining for cost awareness in the infrastructure as code artifacts of cloud-based applications: an exploratory study. arXiv:2304.07531.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv:2002.08155.

Galassi, A., Lippi, M., Torroni, P., 2021. Attention in natural language processing. IEEE Transactions on Neural Networks and Learning Systems 32, 4291–4308. doi:10.1109/TNNLS.2020.3019893.

Gamma, E., Helm, R., Larman, C., Johnson, R., Vlissides, J., 2005. Valuepack: Design Patterns:Elements of Reusable Object-Oriented Software with Applying UML and Patterns:An Introduction to Object-Oriented Analysis and Design and

Iterative Development. Addison Wesley. URL: https://books.google.com.mx/books?id=VDRePgAACAAJ.

Georgousis, S., Kenning, M.P., Xie, X., 2021. Graph deep learning: State of the art and challenges. IEEE Access 9, 22106–22140. doi:10.1109/ACCESS.2021.3055280.

Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y., 2014. Generative adversarial networks. arXiv:1406.2661.

Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., Liu, T., Wang, X., Wang, G., Cai, J., Chen, T., 2018. Recent advances in convolutional neural networks. Pattern Recognition 77, 354–377. URL: https://www.sciencedirect.com/science/article/pii/S0031320317304120, doi:https://doi.org/10.1016/j.patcog.2017.10.013.

Guerriero, M., Garriga, M., Tamburri, D.A., Palomba, F., 2019. Adoption, support, and challenges of infrastructure-as-code: Insights from industry, in: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 580–589. doi:10.1109/ICSME.2019.00092.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J., 2022. Unixcoder: Unified cross-modal pre-training for code representation. arXiv:2203.03850.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S.K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., Zhou, M., 2021. Graphcodebert: Pre-training code representations with data flow. arXiv:2009.08366.

Hao, W., Bie, R., Guo, J., Meng, X., Wang, S., 2018. Optimized cnn based image recognition through target region selection. Optik 156, 772–777. URL: https://www.sciencedirect.com/science/article/pii/S0030402617315735, doi:https://doi.org/10.1016/j.ijleo.2017.11.153.

Hasan, M.M., Bhuiyan, F.A., Rahman, A., 2020. Testing practices for infrastructure as code, in: Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing, Association for Computing Machinery, New York, NY, USA. p. 7–12. URL: https://doi.org/10.1145/3416504.3424334, doi:10.1145/3416504.3424334.

Joshi, A.V., 2020. Amazon's Machine Learning Toolkit: Sagemaker. Springer International Publishing, Cham. pp. 233–243. URL: https://doi.org/10.1007/978-3-030-26622-6_24, doi:10.1007/978-3-030-26622-6_24.

Kagdi, H., Collard, M.L., Maletic, J.I., 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. Journal of Software Maintenance and Evolution: Research and Practice 19, 77–131. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.344, doi:https://doi.org/10.1002/smr.344, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.344.

Kaliyar, R.K., 2020. A multi-layer bidirectional transformer encoder for pre-trained word embedding: A survey of bert, in: 2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence), pp. 336–340. doi:10.1109/Confluence47617.2020.9058044.

Karamanolakis, G., Mukherjee, S., Zheng, G., Awadallah, A.H., 2021. Self-training with weak supervision. CoRR abs/2104.05514. URL: https://arxiv.org/abs/2104.05514, arXiv:2104.05514.

Karras, T., Aila, T., Laine, S., Lehtinen, J., 2017. Progressive growing of gans for improved quality, stability, and variation. CoRR abs/1710.10196. URL: http://arxiv.org/abs/1710.10196, arXiv:1710.10196.

Keery, S., Harber, C., Young, M., 2019. Implementing Cloud Design Patterns for AWS: Solutions and design ideas for solving system design problems. Packt Publishing, Limited.

Kovalenko, V., Bogomolov, E., Bryksin, T., Bacchelli, A., 2019. Pathminer: A library for mining of path-based representations of code, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 13–17. doi:10.1109/MSR.2019.00013.

Land, L., Aurum, A., Handzic, M., 2001. Capturing implicit software engineering knowledge, in: Proceedings 2001 Australian Software Engineering Conference, pp. 108–114. doi:10.1109/ASWEC.2001.948504.

Linthicum, D.S., 2017. Cloud-native applications and cloud migration: The good, the bad, and the points between. IEEE Cloud Computing 4, 12–14. doi:10.1109/MCC.2017.4250932.

Liu, Y., Agarwal, S., Venkataraman, S., 2021. Autofreeze: Automatically freezing model blocks to accelerate fine-tuning. arXiv:2102.01386.

Maffort, C., Valente, M.T., Bigonha, M., Hora, A., Anquetil, N., Menezes, J., 2013. Mining Architectural Patterns Using Association Rules, in: International Conference on Software Engineering and Knowledge Engineering (SEKE'13), Boston, United States. URL: https://inria.hal.science/hal-00854851.

Mistrik, I., Bahsoon, R., Ali, N., Heisel, M., Maxim, B., 2017. Software architecture for Big Data and the cloud. Morgan Kaufmann, Oxford, England.

Morris, K., 2023. Infrastructure as code. O'Reilly Media, Location.

Niu, C., Li, C., Ng, V., Ge, J., Huang, L., Luo, B., 2022. Spt-code: Sequence-to-sequence pre-training for learning source code representations, in: Proceedings of the 44th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 2006–2018. URL: https://doi.org/10.1145/3510003.3510096, doi:10.1145/3510003.3510096.

Opdebeeck, R., Zerouali, A., Velázquez-Rodríguez, C., De Roover, C., 2021. On the practice of semantic versioning for ansible galaxy roles: An empirical study and a change classification model. Journal of Systems and Software 182, 111059. URL: https://www.sciencedirect.com/science/article/pii/S0164121221001564, doi:https://doi.org/10.1016/j.jss.2021.111059.

OpenAI, 2023. Gpt-4 technical report. arXiv:2303.08774.

Palangi, H., Deng, L., Shen, Y., Gao, J., He, X., Chen, J., Song, X., Ward, R., 2016. Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval. IEEE/ACM Transactions on Audio, Speech, and Language Processing 24, 694–707. doi:10.1109/TASLP.2016.2520371.

Perez., Q., Borgne., A.L., Urtado., C., Vauttier., S., 2021. Towards profiling runtime architecture code contributors in software projects, in: Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE, INSTICC. SciTePress. pp. 429–436. doi:10.5220/0010495804290436.

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

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al., 2019b. Language models are unsupervised multitask learners. OpenAI blog 1, 9.

Rahman, A., Mahdavi-Hezaveh, R., Williams, L., 2019. A systematic mapping study of infrastructure as code research. Information and Software Technology 108, 65–77. URL: https://www.sciencedirect.com/science/article/pii/S0950584918302507, doi:https://doi.org/10.1016/j.infsof.2018.12.004.

redmonk, . The RedMonk Programming Language Rankings: January 2023 — redmonk.com. https://redmonk.com/sogrady/2023/05/16/language-rankings-1-23/. [Accessed 14-Jun-2023].

Rühling Cachay, S., Boecking, B., Dubrawski, A., 2021. End-to-end weak supervision, in: Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., Vaughan, J.W. (Eds.), Advances in Neural Information Processing Systems, Curran Associates, Inc.. pp. 1845–1857. URL: https://proceedings.neurips.cc/paper_files/paper/2021/file/0e674a918ebca3f78bfe02e2f387689d-Paper.pdf.

Salehinejad, H., Sankar, S., Barfett, J., Colak, E., Valaee, S., 2018. Recent advances in recurrent neural networks. arXiv:1801.01078.

Savidis, A., Savvaki, K., 2021. Software architecture mining from source code with dependency graph clustering and visualization. doi:10.5220/0010896800003124.

Schmidt, F., MacDonell, S.G., Connor, A.M., 2014. An automatic architecture reconstruction and refactoring framework, in: International Conference on Software Engineering Research and Applications.

Schuster, M., Paliwal, K., 1997. Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing 45, 2673–2681. doi:10.1109/78.650093.

Sehovac, L., Grolinger, K., 2020. Deep learning for load forecasting: Sequence to sequence recurrent neural networks with attention. IEEE Access 8, 36411–36426. doi:10.1109/ACCESS.2020.2975738.

Sharma, A., Kumar, M., Agarwal, S., 2015. A complete survey on software architectural styles and patterns. Procedia Computer Science 70, 16–28. URL: https://www.sciencedirect.com/science/article/pii/S187705091503183X, doi:https://doi.org/10.1016/j.procs.2015.10.019. proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems.

Sharma, S., Sharma, S., Athaiya, A., 2017. Activation functions in neural networks. Towards Data Sci 6, 310–316.

Shin, C., Li, W., Vishwakarma, H., Roberts, N.C., Sala, F., 2021. Universalizing weak supervision. CoRR abs/2112.03865. URL: https://arxiv.org/abs/2112.03865, arXiv:2112.03865.

Shrestha, A., Mahmood, A., 2019. Review of deep learning algorithms and architectures. IEEE Access 7, 53040–53065. doi:10.1109/ACCESS.2019.2912200.

Siow, J.K., Liu, S., Xie, X., Meng, G., Liu, Y., 2022. Learning program semantics with code representations: An empirical study, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 554–565. doi:10.1109/SANER53432.2022.00073.

Smite, D., Moe, N.B., Levinta, G., Floryan, M., 2019. Spotify guilds: How to succeed with knowledge sharing in large-scale agile organizations. IEEE Software 36, 51–57. doi:10.1109/MS.2018.2886178.

Sriram, A., Jun, H., Satheesh, S., Coates, A., 2017. Cold fusion: Training seq2seq models together with language models. arXiv:1708.06426.

Sundararaman, D., Subramanian, V., Wang, G., Si, S., Shen, D., Wang, D., Carin, L., 2019. Syntax-infused transformer and bert models for machine translation and natural language understanding. arXiv:1911.06156.

Taibi, D., El Ioini, N., Pahl, C., Niederkofler, J.R.S., 2020. Serverless cloud computing (function-as-a-service) patterns: A multivocal literature review, in: Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER'20).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I., 2017. Attention is all you need. arXiv:1706.03762.

Wan Mohd Isa, W.A.R., Suhaimi, A.I.H., Noordin, N., Harun, A., Ismail, J., Teh, R., 2019. Cloud computing adoption reference model. Indonesian Journal of Electrical Engineering and Computer Science 16, 395. doi:10.11591/ijeecs.v16.i1.pp395-400.

Wang, Y., Wang, W., Joty, S., Hoi, S.C.H., 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859.

Washizaki, H., Ogata, S., Hazeyama, A., Okubo, T., Fernandez, E.B., Yoshioka, N., 2020. Landscape of architecture and design patterns for iot systems. IEEE Internet of Things Journal 7, 10091–10101. doi:10.1109/JIOT.2020.3003528.

Yussupov, V., Soldani, J., Breitenbücher, U., Brogi, A., Leymann, F., 2021. From serverful to serverless: A spectrum of patterns for hosting application components, pp. 268–279. doi:10.5220/0010481002680279.

Zeng, C., Yu, Y., Li, S., Xia, X., Wang, Z., Geng, M., Xiao, B., Dong, W., Liao, X., 2021. degraphcs: Embedding variable-based flow graph for neural code search. arXiv:2103.13020.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 783–794. doi:10.1109/ICSE.2019.00086.

Zhang, X., Fan, J., Hei, M., 2022. Compressing bert for binary text classification via adaptive truncation before fine-tuning. Applied Sciences 12. URL: https://www.mdpi.com/2076-3417/12/23/12055, doi:10.3390/app122312055.

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., He, Q., 2021. A comprehensive survey on transfer learning. Proceedings of the IEEE 109, 43–76. doi:10.1109/JPROC.2020.3004555.