UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

RESUMEN DE CÓDIGO UTILIZANDO GNN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

JORDI MUXI JIMÉNEZ

PROFESOR GUÍA:
ALEXANDRE BERGEL

MIEMBROS DE LA COMISIÓN:
JORGE PÉREZ ROJAS
HUGO MORA RIQUELME
FEDERICO OLMEDO BERÓN

SANTIAGO DE CHILE
2021

Resumen

Es conocimiento general en el mundo de la programación que un código con buena documentación es más fácil de comprender que otro que no presente las anteriores características.

La dificultad se encuentra sin embargo, en que en un escenario real un programador debe cumplir con determinadas fechas límite que desvían bastante el foco. Debido a esto y otros factores, la buena documentación del código pasa entonces a segundo plano y en la mayoría de los casos queda sin documentar.

Existen a la fecha diversos acercamientos a la tarea de generar resúmenes o explicar segmentos de código de forma automática. El problema común entre ellos es que el lenguaje de programación a diferencia del lenguaje natural posee su propia sintaxis y esquema de relaciones entre enunciados, características que no son explotadas por los trabajos actuales de traducción de código.

Es por ello que este Trabajo de Titulo busca aprovechar tanto los puntos fuertes de las redes neuronales utilizadas en muchos de los trabajos ya mencionados, como las características de preservación de relaciones de los grafos logrando así una implementación basada en redes neuronales de grafos (GNN) de un generador de resúmenes automático o GNN Based Source Code Summarization (GBSCS).

En específico se busca entrenar un modelo de red neuronal capaz de procesar fragmentos de código traduciendo estos a secuencias de nodos y producir como output una secuencia de palabras que logren explicar el funcionamiento principal del fragmento de código.

Para ello se utiliza otra herramienta como punto de partida denominada RENCOS la cual hace uso de redes neuronales recurrentes (RNN) para realizar la confección de resúmenes de código. La métrica de comparación deseable para las herramientas consiste en someter a ambas a distintos métodos evaluadores ampliamente usados en la generación de lenguaje natural y comparar los puntajes obtenidos para cada una de aquellas.

Debido a problemas de implementación si bien fue posible entrenar el modelo, no fue posible producir resúmenes de código resultado debido a un exceso de uso de recursos por parte del modelo. Sin embargo debido al amplio estudio realizado en conjunción al desarrollo de la herramienta se hace la propuesta de posibles cambios que logren habilitar la obtención de un modelo que efectivamente logre producir predicciones y que estas sean posiblemente mejores que las de la herramienta base.

Es así como este trabajo provee de un primer acercamiento a un generador de resumen de código fuente que haga uso de la potencialidad de los grafos.

Tabla de Contenido

1. Introducción 1
1.1. Antecedentes . . . 1
1.2. Motivación . . . 2
1.3. Objetivos . . . 2
1.4. Metodología . . . 3

2. Marco Teórico 4
2.1. Grafo . . . 4
2.1.1. Abstract Syntax Tree (AST) . . . 5
2.2. Red Neuronal . . . 5
2.2.1. Feed Forward Neural Network (FFNN) . . . 6
2.2.2. Recursive Neural Network (RNN) . . . 6
2.2.3. Graph Neural Network (GNN) . . . 8
2.3. Trabajo Relacionado . . . 10
2.3.1. Heramientas Retrieval-based . . . 10
2.3.2. Heramientas basadas en NMT . . . 11
2.3.3. RENCOS . . . 12

3. Diseño de la Solución 17
3.1. Planteamiento del Problema . . . 17
3.2. Alternativas Consideradas . . . 18
3.2.1. Incorporación de un nuevo módulo . . . 18
3.2.2. Reimplementación de un modulo . . . 18
3.2.3. Implementación nueva . . . 18
3.3. Solución Propuesta . . . 19
3.3.1. Selección del *dataset* . . . 19
3.3.2. Implementación de la red . . . 19

4. Desarrollo 21
4.1. Consideraciones Generales . . . 21
4.2. Reproducción de resultados RENCOS . . . 21
4.2.1. Preprocess . . . 21
4.2.2. Train . . . 22
4.2.3. Retrieval . . . 22
4.2.4. Translate . . . 22
4.2.5. Evaluation . . . 22

4.3. Selección y transformación del *dataset* 23
4.4. Implementación de la red 25
4.4.1. Implementación utilizando *Pytorch Geometric* 25
4.4.2. Implementación utilizando ONMT 25

5. Validación y Resultados **26**
5.1. Datos de Prueba 26
5.1.1. RENCOS 26
5.1.2. GBSCS 27
5.2. Calidad del resumen 28
5.2.1. Resultados de las métricas en ambiente local 28
5.2.2. Resultados de las métricas en *Google Colaboratory* 28
5.3. Resultados de entrenamiento del modelo 29
5.3.1. Accuracy 29
5.3.2. Perplexity 32
5.3.3. Cross-entropy 34
5.3.4. Learning rate 37
5.3.5. Elapsed time 39

6. Discusión **42**
6.1. Análisis de Resultados 42
6.1.1. Resúmenes entregados 42
6.1.2. Entrenamiento de los modelos 43
6.2. Propuestas de mejora 43
6.2.1. Investigación Preliminar 43
6.2.2. Implementación y recreación de resultados 43
6.2.3. Segunda fase de investigación y diseño de GNN 44
6.2.4. Implementación GNN 44
6.2.5. Comparación de resultados 44

7. Conclusión **45**

Bibliografía **47**

Índice de Ilustraciones

2.1. Representación de un grafo simple de 6 nodos numerados 4
2.2. Representación de un árbol de sintaxis abstracta o AST 5
2.3. Red neuronal *feedforward* con una capa de input, una escondida y una capa de output 6
2.4. Una unidad LSTM donde $x(t)$ es el input, $h(t)$ es el estado escondido actual, $h(t-1)$ es el estado escondido anterior, $c(t)$ es el vector de contexto actual y $c(t-1)$ es el vector de contexto anterior 7
2.5. Una red convolución de grafos en la cual se denota el *message-passing* como método de traspaso de señales 8
2.6. Pipeline general de una GNN. 10
2.7. Funcionamiento general de la herramienta Rencos 12
2.8. Proceso de obtención de probabilidad de la siguiente palabra del resumen . . 15

5.1. Numero de nodos por AST del *training set* 27
5.2. Evolución de precisión para ambas herramientas durante los primeros 10000 pasos 30
5.3. Evolución de precisión para ambas herramientas durante los primeros 10000 pasos después de realizar una regresión lineal 31
5.4. Evolución de precisión para la herramienta RENCOS durante el transcurso de todo su entrenamiento 32
5.5. Evolución de perplexity para ambas herramientas durante los primeros 10000 pasos 33
5.6. Evolución de perplexity para la herramienta RENCOS durante el transcurso de todo su entrenamiento 34
5.7. Evolución de la entropía cruzada para ambas herramientas durante los primeros 10000 pasos 35
5.8. Evolución de la entropía cruzada para ambas herramientas durante los primeros 10000 pasos después de realizar una regresión lineal 36
5.9. Evolución de la entropía cruzada para la herramienta RENCOS durante el transcurso de todo su entrenamiento 37
5.10. Evolución del factor de aprendizaje para ambas herramientas durante los primeros 10000 pasos 38
5.11. Evolución del factor de aprendizaje para la herramienta RENCOS durante el transcurso de todo su entrenamiento 39
5.12. Evolución del factor de aprendizaje para ambas herramientas durante los primeros 10000 pasos 40

5.13. Evolución del factor de aprendizaje para la herramienta RENCOS durante el transcurso de todo su entrenamiento 41

Índice de Tablas

5.1. Resultados obtenidos en ambiente local versus presentados en paper RENCOS 28
5.2. Resultados obtenidos en plataforma *Google Colaboratory* versus los previamente expuestos 28

Capítulo 1

Introducción

1.1. Antecedentes

En la actualidad las *Artificial Neural Networks* (ANN) son utilizadas en diversas áreas de la computación, ya sea debido a su capacidad de aprender y modelar relaciones complejas; su capacidad de generalización o bien las pocas restricciones que requiere su uso ante otros métodos predictivos, tales como el procesamiento de imágenes, el reconocimiento de caracteres, la construcción de modelos de predicción entre otros.

Las redes neuronales sin embargo pueden diferir bastante en su método de aplicación, captación de input y/o arquitectura. Por ejemplo las capacidades de una *Feedforward Neural Network* (FNN) no serán las mismas que las de una *Recurrent Neural Network* (RNN), aunque su principio sea el mismo. Utilizando otra perspectiva también, la implementación de esta última puede no ser tan simple como la de la primera. Debido a lo anterior existe un amplio rango de implementaciones de redes neuronales.

Entre aquellas formas de implementación existe un modelo relativamente reciente y del cual se estima puede obtenerse un gran valor, este consiste en las *Graph Neural Networks* (GNN)[6]. Las GNN si bien preservan la idea principal de una ANN, tienen un potencial por lo general no explotado por el resto de las redes y se trata de su capacidad de preservar las relaciones entre los datos a la hora de captarlos para su trabajo. Es común que se utilicen datos en forma de grafos para su trabajo en *machine learning*, pero por lo general estos son aplastados o pierden dimensiones cuando llega la hora de trabajar con ellos. En el caso de las GNN se busca preservar por el mayor tiempo posible la dimensionalidad y relaciones presentes entre ellos, permitiendo la perseverancia de información y aprovechamiento de patrones de otra forma escondidos o perdidos durante el proceso de trabajo.

Debido a lo reciente que es el trabajo con GNN, el objetivo de esta memoria es ampliar el conocimiento acerca de este modelo computacional por lo cual se buscaron potenciales áreas de aplicación para aquel modelo.

Después de considerar diversas alternativas se optó por el trabajo en detección de patrones de caracteres, en particular, en la elaboración automática de resúmenes de código fuente.

1.2. Motivación

Es conocimiento general en el mundo de la programación que un código con buena documentación es más fácil de comprender que otro que no presente las anteriores características[7].

Sin embargo la dificultad se encuentra en que en un escenario real un programador debe cumplir con determinadas fechas límite que desvían bastante el foco. En estos casos entonces en vez de privilegiar el buen entendimiento del código fuente, se pone mayor preocupación y recursos en que dicho código cumpla con su objetivo dejando de lado la documentación. Sumado a lo anterior muchas veces una misma sección de código es modificada frecuentemente lo cual puede resultar frustrante a la hora de documentar y resultar en que simplemente no se realice esta acción.

Debido a los diversos factores previamente mencionados, (y varios por mencionar [10]) hoy en día se cuenta con una cantidad enorme de código pendiente de documentar. La idea de documentar dicha magnitud con esfuerzo humano no es realista ni eficiente. El prospecto de automatizar dicho proceso y ahorrar entonces el esfuerzo humano que significaría realizar aquella tarea, suena como el camino a seguir. Sin embargo la realización de la tarea anterior no está desligada de diversas dificultades.

Existen a la fecha diversos acercamientos a la tarea de generar resúmenes o explicar segmentos de código de forma automática. De aquellos se destaca la herramienta RENCOS [11] capaz de utilizar tanto técnicas de recuperación de información como redes neuronales. Esta herramienta posee los mejores resultados en comparación a sus pares a la fecha de realización de esta memoria, razón por la cual se utilizó como base para el trabajo. El problema y posible fuente de mejoras recae en el input recibido, el cual actualmente consta solo de funciones o métodos aislados además de que las relaciones entre las distintas operaciones que realiza el código no son aprovechadas por este modelo.

1.3. Objetivos

Objetivo General

Este trabajo tiene como objetivo explorar la capacidades y potencialidades de las redes neuronales de grafos (GNN's), buscando manifestar a través de ejemplos concretos su diferencia ante otros arquetipos de redes. Además se desea avanzar el trabajo de documentación automática de código por las razones previamente expuestas.

Objetivos Específicos

1. Desarrollar una nueva herramienta de generación automática de comentarios de código fuente.

2. Evaluar la calidad de dichos comentarios en relación a otra herramienta la cual se considera del estado del arte frente a este tópico usando métricas predefinidas.

3. Comparar el rendimiento de la herramienta en cuanto a tiempo de ejecución, rapidez de convergencia y otros con el rendimiento con el de la herramienta existente.

4. Ampliar el espectro de input recibido de la herramienta anterior el cual consiste sólo en métodos aislados a todo el código fuente.

1.4. Metodología

La metodología del trabajo se dividió en las fases 1-5, las cuales se describen en grandes rasgos a continuación.

1. Investigación preliminar: Esta parte del desarrollo del trabajo consistió en el estudio de la herramienta utilizada como base para la memoria. Esto engloba el estudio acerca de redes neuronales recurrentes con unidades de memoria *long-short term memory* (LSTM), el estudio de como se relacionan dichas redes a las redes neuronales de grafos, el estudio de herramientas de *machine learning* y el estudio de las diversas librerías de *python* que utilizaba el software base.

2. Implementación y recreación de resultados: Esta etapa consistió principalmente en el montaje y ejecución correctas de la herramienta base buscando corroborar, a base de resultados obtenidos localmente en comparación a resultados publicados por los autores de la herramienta, su correcto funcionamiento.

3. Segunda fase de investigación y diseño de GNN: En esta fase se deliberó qué aspecto del trabajo presente podría ser mejorado por la implementación de una red neuronal de grafos. Luego de aquello se investigó la mejor manera de transicionar los datos y el trabajo presente a una herramienta que pudiese utilizar las capacidades de una GNN.

4. Implementación GNN: Este paso consistió en la implementación de la nueva herramienta empleando las convenciones adoptadas por la fase anterior.

5. Comparación de resultados: Debido a que la herramienta base ya disponía de una forma de evaluación acorde a métricas reconocidas esta fue reutilizada para poder realizar comparaciones directas de resultados entre ambas herramientas.

Capítulo 2

Marco Teórico

Este capitulo se enfoca en la descripción de los distintos elementos que componen el trabajo de esta memoria.

2.1. Grafo

Un grafo es una estructura de datos que consiste de un conjunto finito de vértices (también llamados puntos o nodos) y un conjunto de pares de aquellos los cuales denotan un carácter de relación o enlace entre los puntos que conforman el par. El conjunto de pares de vértices tiene un orden cuando se trata de un grafo dirigido, en otro caso es opcional. Los grafos pueden tener valores tanto para los elementos que conforman el conjunto de puntos o los que conforman el conjunto de relaciones. Los grafos son ampliamente usados como forma de representar datos dado que preservan el carácter relacional de estos, a menudo sirviendo incluso de apoyo visual durante su estudio.

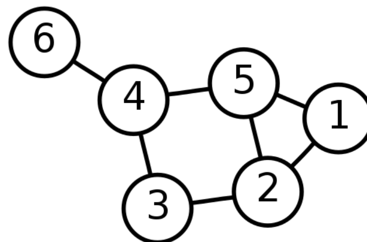

Figura 2.1: Representación de un grafo simple de 6 nodos numerados

2.1.1. Abstract Syntax Tree (AST)

Los AST o árboles de sintaxis abstracta son una representación como árbol (Lo cual es un tipo de grafo) de la sintaxis abstracta de un determinado lenguaje, en este caso nos enfocaremos en lenguajes de programación. El AST consiste entonces en la representación visual de la secuencia de operaciones que realizará el programa guardando en sus nodos dichas operaciones. A nivel fundamental los AST preservan solo la estructura de la funcionalidad de un programa, es decir no consideran los detalles de cómo esta escrito siempre y cuando sea sintácticamente válido. Los AST forman parte fundamental de los compiladores de código dado que sirven de representación intermedia del programa durante las distintas fases de compilación.

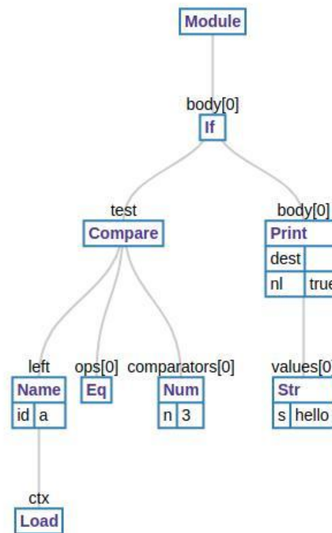

Figura 2.2: Representación de un árbol de sintaxis abstracta o AST

2.2. Red Neuronal

Las redes neuronales son mecanismos de predicción y detección de patrones utilizados en diversas áreas de la computación. Una red neuronal se compone de un elemento básico llamado neurona o nodo el cual de forma similar a una neurona humana, procesa señales pudiendo omitir o propagar mensajes a otras neuronas utilizando criterios de selección intrínsecos a dicha neurona. Estos criterios de selección a su vez hacen uso de dos parámetros fundamentales de una neurona los cuales son el peso de cada una de sus conexiones y el "*bias*" propio de la neurona. Los pesos de las conexiones le indican a esta que tan importante es la activación de una determinada neurona y el *bias* le permite ajustar el valor de la señal que habrá de emitir dicha neurona.

Una red neuronal usualmente tiene múltiples neuronas las cuales por lo general están dispuestas por capas. Las redes neuronales ajustan los criterios de traspaso de señales de sus neuronas mediante mecanismos de aprendizaje. Es en la distribución, estructura y mecanismo de aprendizaje de las neuronas en donde se diferencian los distintos tipos de redes neuronales.

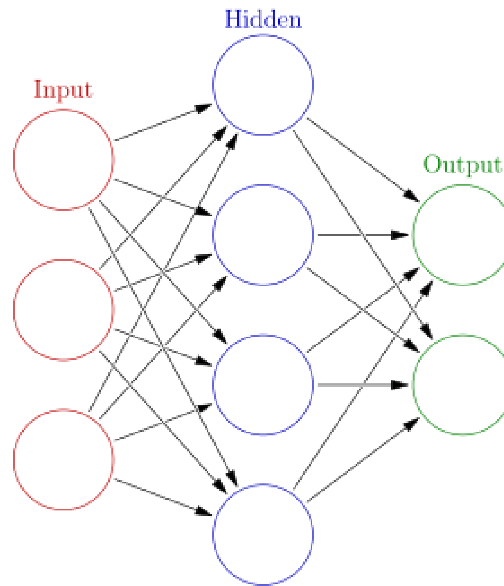

Figura 2.3: Red neuronal *feedforward* con una capa de input, una escondida y una capa de output

2.2.1. Feed Forward Neural Network (FFNN)

Una de las formas más simples de red neuronal usualmente utilizada como mecanismo introductorio a la noción de red neuronal. En este tipo de red la información fluye desde la capa de input a las capas escondidas (si es que tiene) y finalmente a la de output siempre en la misma dirección, es decir no hay ciclos. El mecanismo de aprendizaje más utilizado en las FFNN es *backpropagation*, el cual consiste en la comparación de los valores de output con los valores esperados mediante una función predefinida de error. Dicho error es posteriormente propagado en la dirección opuesta al flujo original de la información para que la red pueda realizar ajustes acorde a este mismo.

Una de las limitaciones de esta implementación es precisamente el hecho de que la información fluye en un solo sentido con lo cual, dado una capa específica, no es posible obtener señales de una capa posterior. Un ejemplo de donde no sería posible obtener buenos resultados al usar una FNN es en el procesamiento de texto dado que por lo general se requiere de información ya procesada para hacer sentido del input.

2.2.2. Recursive Neural Network (RNN)

Las RNN son un derivado de las FNN con la distinción de que sus neuronas poseen estados internos, es decir tienen memoria y son capaces de recordar características del input. Aquí las relaciones entre nodos pueden verse como un grafo dirigido el cual representa una secuencia temporal lo cual le da carácter dinámico a la red. Son utilizadas en múltiples áreas incluyendo el reconocimiento del habla y el trabajo con texto. Un problema latente para las RNN sin embargo, es la desaparición exponencial de la gradiente de la función de error. Esto ocurre cuando la red esta aprendiendo y significa básicamente que los cambios en la gradiente son ínfimos lo que ocasiona que los pesos no se actualicen y en consecuencia la red deje de

aprender. Una manera de prevenir este problema consiste en utilizar la arquitectura LSTM.

Long Short Term Memory (LSTM)

Las redes de LSTM son un tipo de arquitectura de RNN en la cual, a diferencia de las FFNN se cuenta con una conexión *feedback*, es decir una capa puede recibir información desde una capa siguiente. Las LSTM son utilizadas en problemas donde se requiere clasificación, procesamiento o la elaboración de predicciones sobre datos dispuestos en secuencias temporales. Una unidad LSTM generalmente se compone de una célula, un *input gate*, un *forget gate* y un *output gate*. Esta unidad mediante estos 3 portones se encarga de seleccionar que información le sirve preservar y cuando, eliminando la información mediante señales de su *forget gate* cuando estima que ya no le será de uso.

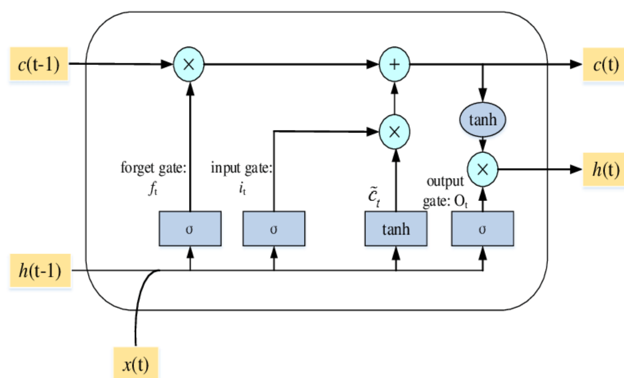

Figura 2.4: Una unidad LSTM donde $x(t)$ es el input, $h(t)$ es el estado escondido actual, $h(t-1)$ es el estado escondido anterior, $c(t)$ es el vector de contexto actual y $c(t-1)$ es el vector de contexto anterior

2.2.3. Graph Neural Network (GNN)

Las GNN son una arquitectura relativamente reciente de redes neuronales y extienden los métodos utilizados por aquellas para que puedan incorporar datos presentes en el dominio de los grafos. Las redes neuronales de grafos operan mediante un mecanismo llamado *message-passing*, el cual se manifiesta mediante señales de output de un determinado nodo el cual considera tanto sus propias *features* como las de sus vecinos cercanos[6].

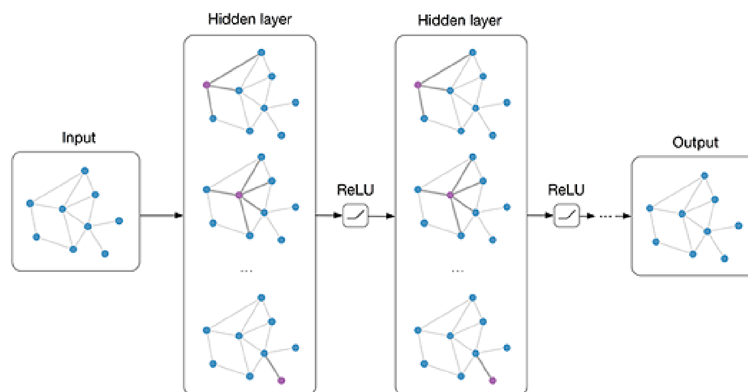

Figura 2.5: Una red convolución de grafos en la cual se denota el *message-passing* como método de traspaso de señales

Una de las características más atractivas de las redes neuronales de grafos es su capacidad de captar *features* del input y preservar relaciones entre aquellas en fases del entrenamiento donde de otra manera, para las redes neuronales clásicas, se perderían al perder la dimensionalidad del grafo para poder entregarlo como input.

Con lo anterior en mente a la hora de construir una red neuronal de grafos se debe tener en cuenta que lo que se busca preservar es el esquema de relaciones que posee la data. Es por ello que le primer paso es definir cómo se obtendrá el input de aquella. Existen dos posibles opciones, la data ya viene en estructura de grafo (por ejemplo los datos se tratan de moléculas, árboles genealógicos, conexiones en redes sociales, etc.) o el grafo está implícito en los datos [13]. En base al grafo mismo de input a su vez se pueden clasificar como:

- Dirigidos/no-dirigidos: En estos grafos se posee una dirección (o no en el caso no-dirigido) en la cual se deben recorrer los arcos grafo.

- Homogéneo/heterogéneo: Para el caso de los grafos homogéneos todo nodo o arco del grafo es del mismo tipo que cualquier otro nodo o arco respectivamente del mismo grafo. En el caso heterogéneo la diferencia entre estos nodos y/o arcos es de alta importancia.

- Estático/dinámico: Si el grafo de input varía con el tiempo estamos en el caso de un grafo dinámico, si se mantiene constante es estático.

Todas las aquellas categorías de grafos son combinables entre sí (por ejemplo se puede tener un grafo no-dirigido, homogéneo y dinámico). Existen aún más categorías no relevantes para el trabajo de esta memoria por lo cual no se listan.

Una vez obtenido el tipo de input a recibir se deberá definir cómo aprenderá la red, es decir se deberá definir el tipo de función de pérdida a utilizar. Por lo general se utilizan 3 enfoques:

- A nivel de nodo: El enfoque del entrenamiento es en los nodos, incluye clasificación de nodos, regresión de nodos y *clustering* entre otros. La clasificación busca categorizar a los nodos en distintas clases, la regresión predice un valor continuo para cada nodo y el *clustering* consiste en dividir los nodos en grupos disjuntos donde los nodos similares entre sí son del mismo grupo.

- A nivel de arco: Para el caso de los arcos existe la clasificación y la predicción donde la clasificación consiste en asignarle un tipo al arco y la predicción en formar nuevos arcos inexistentes previamente entre nodos.

- A nivel de grafo: Para este tipo de enfoque se busca clasificar el tipo de grafo, hacer una regresión del grafo o hacer un *matching* entre grafos.

En el ámbito de la supervisión se tienen las distintas categorías siguientes:

- Entrenamiento supervisado: Los datos vienen etiquetados para el entrenamiento.

- Entrenamiento semi-supervisado: Existe una pequeña cantidad de nodos etiquetados y una gran cantidad de nodos de etiqueta desconocida.

- Entrenamiento no supervisado: No existen etiquetas para los nodos, por lo general se utiliza en *clustering* .

Con tanto el enfoque y tipo de entrenamiento seleccionados es posible diseñar una función de pérdida específica para la tarea a realizar con la red neuronal de grafos.

Finalmente en términos de los pasos de diseño de una red neuronal de grafos se tiene la selección del módulo de cómputo. Los más comunes son:

- Módulo de propagación: Se propaga la información entre los nodos de tal manera que el cúmulo de aquella pueda capturar tanto datos de las *features* como información topológica. En módulos de propagación, el operador de convolución y el operador recurrente usualmente se usan para acumular la información entre los nodos vecinos mientras que la operación *skip-connection* se usa para acumular información de las representaciones históricas de los nodos y mitigar el problema del allanamiento excesivo.

- Módulo de producción de muestras: Cuando el grafo es muy grande, se utiliza un modulo de producción de muestras para realizar la propagación en el grafo. Por lo general se aplica en conjunción con el módulo de propagación.

- Módulo de *pooling*: Cuando se necesita hacer representaciones de alto-nivel de grafos o sub-grafos, se utilizan módulos de *pooling* para extraer información de los nodos.

Usualmente una GNN hace uso de varias combinaciones de estos módulos de cómputo. Por ejemplo una red que hace uso de un operador convolucional, un operador recurrente,

un módulo de producción de muestras (*sampling module*) y una operación *skip-connection* se usan para propagar información entre las capas para finalmente utilizando un módulo de *pooling* extraer información de alto nivel. Dicha red se puede observar en la Figura 2.6.

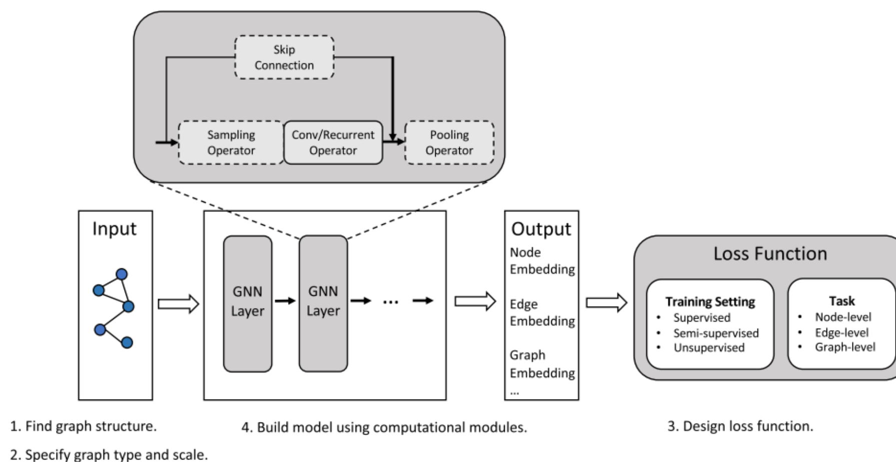

Figura 2.6: Pipeline general de una GNN.

2.3. Trabajo Relacionado

Existen diversas herramientas dedicadas a la generación automática de comentarios[11] las cuales son agrupables acorde al método principal de obtención de dicho comentario. Los métodos previamente mencionados consisten en *Retrieval-based* y *Neural Machine Translator* (NMT).

2.3.1. Heramientas Retrieval-based

Las herramientas *Retrieval-based* hacen uso de la información latente en el texto para producir un resumen de términos basándose en la selección de palabras apropiadas del código original. Además de estos resúmenes basados en términos, se puede generar otro resumen a partir de comentarios de código similar. Dado que el código duplicado es común en repositorios de gran escala por lo general se adoptan técnicas de clonación de código para obtener retazos de código similar desde foros de programación cuyos comentarios después pueden ser adaptados para generar un nuevo resumen. A continuación se listan algunas herramientas cuyo enfoque es *Retrieval-based*.

LSI

Consiste en una técnica de recuperación de texto la cual hace uso del significado o conceptos de un documento.

VSM

Acrónimo de *Vector Space Model* y se encarga de indexar fragmentos de código a vectores de peso basados en la frecuencia de aparición de un termino y a su vez la frecuencia de

aparición de su documento.

NNGen

Algoritmo basado en la búsqueda de vecinos cercanos propuesto para producir mensajes de *commits* sensibles a cambios en el código.

2.3.2. Heramientas basadas en NMT

Esta clase de herramientas basa su funcionalidad en el aprendizaje mediante redes neuronales de características intrínsecas de un fragmento de código que maximizan la probabilidad de aparición de una determinada palabra en el resumen de aquel. Estos enfoques por lo general privilegian palabras comunes ignorando los términos especiales. Es debido a este último punto que por lo general el enfoque NMT se considera el enfoque más débil de los dos. Algunas de las herramientas que hacen uso del enfoque NMT se exponen a continuación.

CODE-NN

Es la primera herramienta documentada que aprende a generar resúmenes a partir de código fuente. Consiste en una red LSTM *encoder-decoder* que codifica los fragmentos de código a vectores de contexto con mecanismos de atención y luego produce dichos resúmenes.

TL-CodeSum

Es un modelo neuronal que codifica secuencias API además de tokens de código y genera resúmenes con conocimiento transferido de la API. Primero entrena un codificador de secuencias de API usando un *dataset* externo y luego usa las representaciones de estas API para apoyar al proceso de generación del resumen.

Hybrid-DRL

Es un aproximamiento que hace uso de representaciones híbridas del código y aprendizaje de refuerzo profundo. Su arquitectura hace uso de *multi-encoders* para aprender información estructural y secuencial de la información al codificar tanto los AST como las tokens del código fuente.

2.3.3. RENCOS

La principal herramienta en la cual se basa el trabajo de esta memoria consistió en el *Retrieval-based Neural Source Code Summarizer* o RENCOS. Esta herramienta puede hacer uso de las ventajas de cada uno de los enfoques vistos anteriormente y su funcionamiento se explicará a continuación.

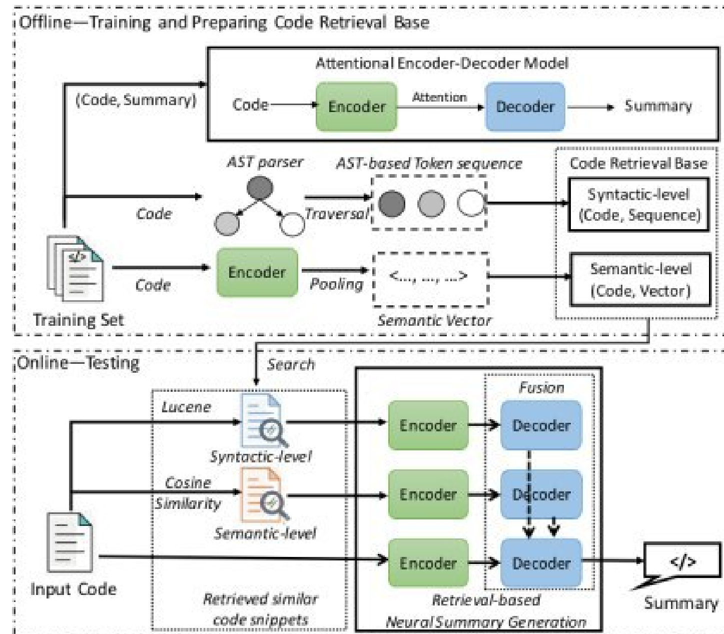

Figura 2.7: Funcionamiento general de la herramienta Rencos

A modo general la herramienta primero reúne una gran cantidad de fragmentos de código y sus respectivos resúmenes para entrenar un modelo atencional del tipo *encoder-decoder*. Una vez entrenado dicho modelo se utiliza la parte del *encoder* para obtener una representación como vector semántico del código la cual se utilizará durante la fase de *testing*. En esta última fase dado un fragmento de código de prueba, se buscan los dos fragmentos de código más parecidos sintáctica y semánticamente de una base de códigos elaborada con anterioridad. El código más parecido sintácticamente es obtenido usando un motor de búsqueda usando la secuencia de tokens del código y el otro es obtenido utilizando la similitud por coseno de la representación como vector semántico del código (es aquí donde se utiliza la representación previamente obtenida del *encoder*). Se obtiene el resultado final dándole al modelo los 3 fragmentos de código y fusionando los resultados.

Para mejor entender el funcionamiento de esta herramienta y la desarrollada en esta memoria se explica cada una de las fases por las que pasa RENCOS en la generación de código automático a continuación.

Modelo atencional encoder-decoder

Como ya se dijo anteriormente y similarmente a otros NMT, se construye y entrena un modelo del tipo *encoder-decoder* del cual se hace uso en 3 lugares: en la codificación de

una representación intermedia del input de *testing* y sus fragmentos más parecidos, en la decodificación de esta representación y para obtener la representación como vector semántico de un fragmento de código.

Para el *encoder* suponiendo c como un fragmento de código consistente de una secuencia de palabras $w_1, \cdots, w_n$ se utiliza una capa de *embbeding* para inicializar estas palabras como vectores de la siguiente manera:

$$x_{\mathrm{i}} = W_{\mathrm{e}}^{\top} w_{\mathrm{i}}, \mathrm{i} \in [1, n] \tag{2.1}$$

Donde n es el largo del fragmento de código c, W_{e} es la matriz de *embbeding*. Luego de esta inicialización se utilizan unidades LSTM para codificar esta secuencia de vectores $x = x_1, \cdots, x_n$ a vectores de estado escondido $h_1, \cdots, h_n$. Por simplificación se denota a la unidad LSTM como:

$$h_t = LSTM(x_t, h_{t-1}) \tag{2.2}$$

Luego se utilizan estas unidades en ambas direcciones (esta última práctica denominada Bi-LSTM y estas direcciones de 1 a n y viceversa), para capturar la semántica tanto adelante como atrás de la posición actual.

A la hora de decodificar esta representación del fragmento de código y generar la i-ésima palabra del resumen, se utiliza un decodificador atencional que computa un vector de contexto v_{i} sobre la secuencia previa de estados escondidos $h_1, \cdots, h_t, \cdots, h_n$ de acuerdo a la siguiente fórmula:

$$v_{\mathrm{i}} = \sum_{j=1}^{n} a_{\mathrm{i}j} h_j \tag{2.3}$$

Donde $a_{\mathrm{i}j}$ es el peso de atención de h_j calculado como:

$$a_{\mathrm{i}j} = \frac{\mathrm{e}xp(\mathrm{e}_{\mathrm{ei}j})}{\sum_{k=1}^{n} \mathrm{e}xp(\mathrm{e}_{\mathrm{i}k})}. \tag{2.4}$$

Con:

$$\mathrm{e}_{\mathrm{i}j} = a(s_{\mathrm{i}-1}, h_j) \tag{2.5}$$

Donde $s_{\mathrm{i}-1}$ corresponde a el último estado escondido del *decoder*. En la ecuación anterior se utiliza una unida de *perceptron* multi-capa (MLP) como modelo de alineamiento a. Durante el paso de tiempo i, el estado escondido s_{i} del *decoder* es actualizado por la ecuación:

$$s_{\mathrm{i}} = LSTM(s_{\mathrm{i}-1}, y_{\mathrm{i}-1}) \tag{2.6}$$

Donde $y_{\mathrm{i}-1}$ es el vector del input anterior. Para considerar información de alineamiento ya procesada, se concatena $v_{\mathrm{i}-1}$ con los inputs $y_{\mathrm{i}-1}$.

$$p(y_{\mathrm{i}} | y_1, \cdots, y_{\mathrm{i}-1}, c) = g(y_{\mathrm{i}-1}, s_{\mathrm{i}}, v_{\mathrm{i}}) \tag{2.7}$$

Donde g es una función generadora, la que aplica una capa MLP junto con un $softmax$. También hace uso del algoritmo "*beam search*", el cual consiste en la búsqueda de las mejores B hipótesis durante cada paso t donde B es el tamaño de dicho *beam*. La función de corrección de este modelo o función de pérdida es:

$$LOSS(\theta) = -\sum_{\mathrm{i}=1}^{N} \sum_{t=1}^{L} logP(y_t^{\mathrm{i}}, y_{<t}^{\mathrm{i}}, c) \tag{2.8}$$

Donde θ son los parámetros entrenables, N son las instancias de entrenamiento y L es la longitud de cada secuencia objetivo. Luego del entrenamiento con los fragmentos de código y sus respectivos resúmenes, se obtiene un *encoder* capaz de representar estos fragmentos y un decodificador capaz de predecir resúmenes palabra a palabra al maximizar las probabilidades condicionales de las palabras siguientes como visto en la ecuación de probabilidades. Si solo se utilizase este modelo para la predicción de resúmenes como los anteriores trabajos descritos con enfoque en NMT, este modelo preferirá solo las palabras con alta frecuencia de aparición e ignorará las palabras menos frecuentes.

Recuperación de códigos similares

Basándose en estudios previos, la experiencia sugiere que para fragmentos de código similar sus comentarios son altamente probables de contener palabras de poca frecuencia presentes en cada uno por separado. Es por eso que para el enriquecimiento del resumen, la herramienta busca la inclusión de la mayor cantidad de palabras de poca frecuencia particulares para un determinado fragmento de código. Para esta tarea utiliza dos técnicas de recuperación de código similar, buscando siempre la eficiencia dado que la base de códigos utilizada es bastante grande. Estas técnicas tienen dos enfoques distintos: una busca código de sintaxis similar y la otra de semántica.

Dado que el código a diferencia del lenguaje natural posee su propia estructura sintáctica, esta estructura es muy importante a la hora de buscar código similar. En trabajos previos a la herramienta se buscó explorar esta estructura basándose en métodos costosos computacionalmente de *matching* de árboles. Debido a esto y otras cuantas razones la herramienta busca la comparación optimizada de una secuencia de tokens derivada de estos AST si requerir el entrenamiento de un modelo adicional. En específico dado un fragmento de código del *set* de *testing* y uno cualquiera del *set* de entrenamiento se convierten a AST y luego se calcula su similitud. Para evitar como ya se dijo estos métodos de comparación de árboles los cuales resultan bastante costosos, estos AST se "aplastan" convirtiéndolos en secuencias de tokens las cuales preservan el orden del árbol en formato "preorden". Se buscan luego estas secuencias utilizando el motor de búsqueda Lucene el cual es bastante eficiente.

Para el ámbito semántico se reutiliza el *encoder* ya entrenado en el paso anterior dado que es capaz de capturar la información secuencial del código y acoplar su semántica a un vector de *hidden state*. Entonces dado c un fragmento de código, se codifica utilizando el *encoder* Bi-LSTM y se obtiene una secuencia de vectores $[h_1, \cdots, h_n] \in \mathbb{R}^{n \times 2k}$ con k la dimensión del vector y n la longitud del código c. Se utiliza luego una operación de *max-pooling* global sobre la secuencia de vectores para obtener la representación semántica $r_c \in \mathbb{R}^{1 \times 2k}$ como sigue:

$$r_c = [max(h_{\mathrm{i}}^1), \cdots, max(h_{\mathrm{i}}^2 k), \mathrm{i} = 1], \cdots, n \tag{2.9}$$

Para un fragmento de código de *testing* c_{test} y cualquiera del *training set* c_{i}, se calcula la similitud por coseno:

$$cos(\overrightarrow{r_{test}}, \overrightarrow{r_{\mathrm{i}}}) = \frac{\overrightarrow{r_{\mathrm{test}}} \cdot \overrightarrow{r_{\mathrm{i}}}}{||\overrightarrow{r_{test}}||||\overrightarrow{r_{\mathrm{i}}}||} \in [-1, 1], \mathrm{i} \in [1, N] \tag{2.10}$$

Donde N es el tamaño del *set* de entrenamiento. Se selecciona el fragmento con la mejor puntuación de esta ecuación dado que ese es el más similar. Ambas bases de representaciones

de códigos es decir los códigos tanto en representación como AST y como vector semántico, son calculadas con anterioridad al *testing*.

Resumen de código generado por proceso neuronal basado en recuperación de información

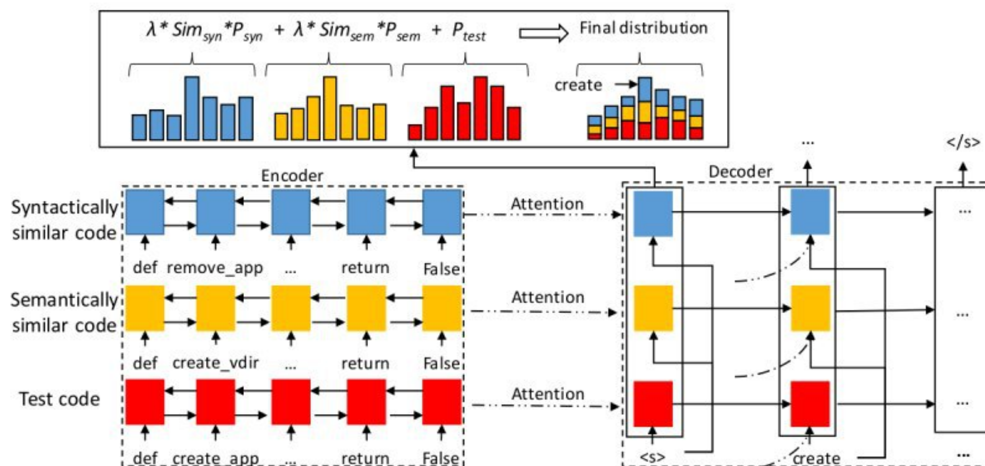

Figura 2.8: Proceso de obtención de probabilidad de la siguiente palabra del resumen

Una vez obtenidos los dos códigos más similares se predicen y generan las palabras del resumen de forma online (es decir durante el testeo) para los códigos del *set* de *testing*. Se utilizan estos dos fragmentos adicionales para el enriquecimiento de la predicción entregada por el modelo, es decir para incluir palabras de baja frecuencia por sobre otras de alta frecuencia.

Basado en el modelo *encoder-decoder* previamente descrito el generador de resumen codifica el fragmento de codigo del *testing set* y sus dos códigos más similares todos aquellos de forma simultanea con lo cual obtiene sus vectores de contexto y luego decodifica estos combinando las probabilidades condicionales de las palabras del resumen y sus similitudes.

En concreto por un fragmento de código del *training set* c_{test}, se busca c_{sin} y c_{sem} (sus fragmentos más parecidos sintácticamente y semánticamente respectivamente) del *training set*. Luego se codifican estos tres por el modelo encoder-decoder previamente entrenado, en paralelo.

Basándose entonces en las 3 secuencias obtenidas de vectores de *hidden state* H_{test}, H_{sin} y H_{sem} para c_{test},c_{sin} y c_{sem} respectivamente, en cada paso t durante el decodificado se pueden computar los pesos atencionales para producir vectores de contexto como en la ecuación 2.3 y luego calcular las probabilidades condicionales para producir la siguiente palabra como en la ecuación 2.7.

A fin de aumentar el desempeño del modelo original, se fusionan estas 3 probabilidades condicionales considerando el valor de similitud entre ellas. Dado que las similitudes entre los fragmentos no son directamente comparables debido a que se enfocan en ámbitos distintos,

se realiza el cálculo de similitud como:

$$Sim(c_{test}, c_{ret}) = 1 - \frac{\mathrm{d}(c_{test}, c_{ret})}{max(|c_{test}|, |c_{ret}|)} \tag{2.11}$$

Donde c_{ret} es el código recuperado de la base. Con estas similitudes normalizadas, finalmente se combinan las probabilidades condicionales como sigue:

$$p_{final}(y_t|y_{<t}) = p_{test}(y_t|y_{<t}) + \lambda \cdot Sim(c_{test}, c_{syn}) p_{syn}(y_t|y_{<t}) + \lambda \cdot Sim(c_{test}, c_{sem}) p_{sem}(y_t|y_{<t}) \tag{2.12}$$

Capítulo 3

Diseño de la Solución

Durante esta etapa del trabajo se consideró de forma exhaustiva cual era realmente el problema que se deseaba atacar, ponderando diversas problemáticas interesantes. Se exponen a continuación los puntos principales considerados durante el planteamiento del problema, junto a las posibles soluciones descartadas preliminarmente.

3.1. Planteamiento del Problema

La herramienta en la cual se basa el trabajo de esta memoria, denominada RENCOS[11], permite la generación automática de comentarios a partir de un fragmento de código. Esto es posible a partir de una red neuronal recurrente la cual hace uso de diversas técnicas de recuperación de información a su vez de un modelo entrenado de predicciones.En específico se utiliza un *dataset* de métodos o funciones en lenguaje de programación y sus correspondientes comentarios para entrenar un modelo atentivo capaz de codificar los fragmentos de código como representación de vectores semánticos para su posterior decodificación como probabilidades de la siguiente palabra del resumen entregado en el output. Una vez entrenado y a la hora de hacer *testing*, para un método de testeo se obtienen los dos fragmentos de código más parecidos sintáctica y semánticamente a este y estos tres son entregados al codificador aumentando la posibilidad de obtención de palabras únicas y enriqueciendo a grandes rasgos la calidad del comentario.

El problema que se buscó atacar en esta memoria consistió en la búsqueda de una implementación que replicase el funcionamiento de esta herramienta buscando preservar sus fortalezas pero privilegiando siempre que incorporase y aprovechase los puntos fuertes de las redes neuronales de grafos. Como puntos fuertes de la herramienta RENCOS se cuenta la aplicación de las técnicas de recuperación de información además de la interpretación intermedia del código como estados escondidos. La idea de expandir el trabajo a una red neuronal de grafos es expandir el rango de input que puede proveerse a la herramienta dado que en la fecha de realización de esta memoria esta herramienta solo era capaz de realizar predicciones de métodos aislados más que de fragmentos completos de código.

El problema de la formulación de dicha implementación recayó principalmente en el grado

de complejidad de la herramienta base sumado también al amplio espectro de posibles mejoras que se pueden realizar sobre ella.

3.2. Alternativas Consideradas

El principal problema que debió atacar la nueva implementación fue seleccionar el punto de partida. Para ello se consideraron 3 alternativas discutidas a continuación:

3.2.1. Incorporación de un nuevo módulo

En esta alternativa se consideraron las principales fortalezas y debilidades de incorporar un modulo adicional de extracción de *features*. Parte de la funcionalidad de la herramienta es seleccionar códigos similares semántica y sintácticamente al código siendo testeado. Es por eso que se estimó factible añadir un tercer criterio que hiciese uso de la estructura del código el cual podría expandir el rango de captación de input de métodos a archivos o proyectos completos. Debido a la evidente relación entre un sistema de archivos y un grafo esta alternativa sonaba bastante atractiva.

Un aspecto a favor de esta alternativa fue que una vez investigada la procedencia del *dataset* utilizado para el desarrollo de la herramienta base, se descubrió que en la elaboración de dicho *dataset* fue preservada la información estructural de método o función. En efecto en el *dataset* original cada uno de los métodos poseía información respecto a su ubicación en el sistema de archivos y en la linea específica del archivo al que pertenecía.

Si bien la idea de poder ampliar el rango de captación de input sonó prometedora, se estimó que requeriría mucho mas tiempo de investigación del cual se disponía. Además se evaluó que debido a que los métodos son por lo general auto contenidos dentro del proyecto sería difícil encontrar la relación proyecto-método por lo cual se descartó esta alternativa.

3.2.2. Reimplementación de un modulo

En esta alternativa se evaluó la posibilidad de re-inventar uno de los mecanismos de extracción de código similar, más específicamente el modulo de extracción de código sintácticamente similar. Debido principalmente a que un árbol de sintaxis abstracta ya consiste en un grafo es natural ver cómo se relaciona con una GNN. Si bien esta fue la alternativa más favorecida, fue sin embargo descartada debido a la complejitud de la implementación de la herramienta. Se estimó que incluso si se implementase dicho acoplamiento no se podría corroborar que efectivamente tuviese el funcionamiento esperado.

3.2.3. Implementación nueva

Esta alternativa consistió en simplemente replantearse la herramienta existente pero que utilice solo GNN. La nueva implementación debería hacer uso del *dataset* y las herramientas de evaluación entregadas por la herramienta base. Entonces a pesar de que se trate de una herramienta nueva desarrollada desde cero debiese ser comparable con los resultados ya obtenidos por la herramienta RENCOS.

3.3. Solución Propuesta

La alternativa escogida a desarrollar luego de considerar los puntos previamente expuestos fue finalmente la tercera, es decir se implementaría un nuevo modelo que hiciese uso de las GNN en su completitud.

La idea fundamental sería que el nuevo modelo utilizase el *dataset* y las herramientas de evaluación ya presentes en la herramienta en la cual se basa. El proceso entonces se puede dividir en 3 partes: selección del *dataset*, diseñar la red neuronal de grafos y finalmente implementar los mecanismos de evaluación. A continuación se describen estas fases en mayor detalle.

3.3.1. Selección del *dataset*

Como primera instancia se debe determinar a partir de qué datos se elaborara un modelo predictivo, es decir, qué tipo de aspectos de aquellos son atractivos para realizar un estudio más profundo y así realizar generalizaciones.

Actualmente el *dataset* con el cual se cuenta consta de secuencias de código fuente, sus AST's tokenizados, sus vectores semánticos y finalmente sus comentarios en lenguaje natural. De estos datos se considera que el único fácilmente extrapolable a grafo consta de las secuencias de AST pero aún así se estima no es una conversión trivial. Realizando un estudio de dicho *dataset* se notó además que no disponía de la información completa respecto a su origen. El *dataset* original entonces consistía además de los fragmentos de código y sus respectivos comentarios, de la ubicación en el repositorio del cual provenían. Se planteó de forma extensiva implementar un modelo que considerase estos aspectos pero por motivos de tiempo se optó por descartar esta última información.

La decisión final tomada respecto a los datos a utilizar, fue que se considerarían solo los fragmentos de código los cuales serían transformados a AST pero en vez de ser tokenizados se convertirían a una estructura compatible con el trabajo con grafos es decir trabajables por una GNN. Entonces la red a implementar en el paso siguiente haría utilización de estos AST para elaborar sus predicciones respecto del resumen.

3.3.2. Implementación de la red

Una vez definido que el *dataset* a utilizar constará de los AST de los fragmentos de código fuente, se realizó la tarea de decir qué tipo de arquitectura de GNN se utilizaría para el desarrollo de la solución. Se consideraron para esto 2 alternativas las cuales se discutirán a continuación.

Graph Atention Network (GAT)

La idea principal de las GAT consiste en que se computan representaciones escondidas de los nodos del grafo al usar mecanismos de *self-attention* en sus nodos vecinos. Una ventaja de los mecanismos de atención es que permiten trabajar con inputs de distintos largos y tamaños enfocandose en las partes más relevantes para la toma de decisiones[**?**]. Por lo general cuando se utiliza un mecanismo de atención para computar una representación de

una sola secuencia se le llama a esto *self-attention*. Se ha demostrado que este mecanismo no solo es capaz de mejorar un método basado en RNN, sino que además solo es necesario este primer mecanismo para generar un modelo lo suficientemente poderoso para obtener rendimiento acorde al estado del arte en tareas de *machine translation*. Dado este último punto es por lo cual se considero utilizar una GAT. Finalmente las principales ventajas de esta arquitectura son:

- Es eficiente, dado que es paralelizable a través de distintos pares de nodos con nodos vecinos.
- Se puede aplicar con nodos de distinto grado añadiendo pesos específicos para los vecinos.
- El modelo es directamente aplicable a problemas de aprendizaje inductivo donde se deben hacer generalizaciones a grafos no vistos.

Gated Graph Neural Network (GGNN)

Este modelo de red neuronal busca extender la funcionalidad del modelo clásico de GNN, para que en vez de producir un output de tipo clasificatorio o predictivo de clase específica de un grafo, produzca una secuencia de *outputs*[5].

La modificación más grande que efectúan las GGNN sobre las GNN's convencionales es que utilizan *Gated Recurrent units* (GRU) y desenrollan la secuencia por un numero fijo de pasos T y utilizan *backpropagation* a través del tiempo para así computar las gradientes. Esto requiere una cantidad considerable de memoria pero remueve la necesidad de acotar parámetros para asegurar la convergencia del modelo. Además son extendidas las representaciones internas y el modelo de *outputs* de la red. Las GGNN's además permiten incorporar *labels* de los nodos como input adicional.

Dado que este tipo de red tiene mayor grado de similitud a la red neuronal implementada en la herramienta RENCOS (ambas utilizan un módulo de propagación con operadores recurrentes de portones o *gates*), se decidió utilizar este arquetipo para el desarrollo de la solución.

Capítulo 4

Desarrollo

En este capítulo se aborda el desarrollo de la solución propuesta en la sección anterior 3.3. En esta sección en contraste a la ya mencionada sección anterior se presentará en detalle las distintas herramientas, consideraciones respecto al software y en general detalles de su implementación.

A diferencia de la sección anterior en la cual se dividió el desarrollo de la solución en 3 partes, esta sección contempla un apartado adicional el cual detalla cómo fueron replicados los resultados de la herramienta RENCOS los cuales sirvieron como base para la formulación de la nueva herramienta.

4.1. Consideraciones Generales

Para comenzar la formulación de la solución lo primero que se hizo fue buscar la replicación de resultados de la herramienta RENCOS dado que esta serviría de base para el trabajo futuro. Esta herramienta estaba escrita en el lenguaje de programación *Python* y utilizaba diversas herramientas enfocadas en *machine learning* las cuales se listan en listan. Entonces se puede dividir esta sección en 4 partes listadas a continuación:

4.2. Reproducción de resultados RENCOS

Dado que el objetivo de esta memoria fue la replicación de una herramienta ya existente, se tuvo como primer paso lograr el entendimiento y posteriormente lograr obtener los resultados de aquella herramienta. Para ello como primera acción se clonó el repositorio original del proyecto desde *GitHub* a un ambiente local. Luego de aquello se estudió a fondo el funcionamiento interno de la herramienta, examinando a fondo el algoritmo empleado para obtener resultados. Las fases por las que pasa este programa pueden ser divididas en 5 fases:

4.2.1. Preprocess

Esta fase como estipula su nombre consiste en el preprocesamiento del *dataset* de la herramienta lo cual consiste principalmente en la creación de las estructuras de las que hará

uso el modelo de aprendizaje por redes neuronales. Es aquí donde se inicializan las matrices de input, se construye el vocabulario de los datos (esto consiste en una representación interna del modelo para las palabras del *dataset*), además se aprovecha de establecer un archivo de logging que guarda registro del proceso.

4.2.2. Train

En esta fase es entrenado el modelo *encoder-decoder* previamente descrito. Esto es logrado utilizando la biblioteca de *machine learning Online Machine Translator*, la cual provee funcionalidad principalmente para el trabajo de traducción de texto. Durante esta fase de la replicación de resultados fue evidente que se carecía del poder computacional suficiente para lograr el entrenamiento exitoso del modelo en el ambiente local original. Es por ello que se decidió reimplementar esta sección en la plataforma *Colaboratory* la cual brindaba una mayor potencia computacional. Una vez entrenado el modelo este fue descargado para resumir el trabajo en el ambiente local original puesto que se estimó que sería más fácil de realizar los pasos posteriores en dicho ambiente.

4.2.3. Retrieval

Es en esta fase donde se construyeron los índices del motor de búsqueda *Lucene* además del calculo de los vectores semánticos de los fragmentos de código. Cabe destacar que el cálculo y tokenización de los AST formaron parte del *dataset*, es decir no se tuvo que realizar este caso en ninguna de las fases de obtención de resultados. Se discutirá más tarde como esto afecto de forma negativa el desarrollo de la nueva herramienta.

4.2.4. Translate

Finalmente una vez realizadas todas las tareas anteriores, la herramienta procede a hacer el *testing* y generar resúmenes de código a partir de nuevos fragmentos. Esta fase entonces constituyó en su totalidad de hacer pruebas y obtener resúmenes de código.

4.2.5. Evaluation

En esta última parte la cual podría considerarse completamente desligada de las cuatro anteriores se realizó la evaluación de los resultados en base a distintas métricas de evaluación de calidad de generalización de texto. Los resultados de esta fase se encuentran presentes en la sección de resultados. Estas métricas se listan a continuación:

Bilingual Evaluation Understudy (BLEU)

Dado un resumen generado X y el resumen objetivo Y, esta métrica evalúa la precisión del n-grama entre X e Y al computar los radios de sobre-posición de dichos n-gramas y aplicando una penalización de brevedad para hipótesis de traducción muy cortas. Se utilizó la evaluación de n-gramas para $n \in [1, 2, 3, 4]$. La fórmula de BLEU es:

$$BLEU - N = BP \sum_{n=1}^{N} w_n log p_n$$

Donde p_n es el *score* de precisión de los aciertos del n-grama entre la frase candidata y la frase de referencia. BP es el *penalty* de brevedad y w_n es el peso uniforme $1/N$.

Metric for Evaluation of Translation with Explicit ORdering (METEOR)

METEOR consiste en que para un par de frases se crea un alineamiento de palabras entre ellas y se calculan sus puntajes de similitud en base a:

$$METEOR = (1 - \gamma \cdot frag^{\beta}) \cdot \frac{P \cdot R}{\alpha \cdot P + (1-\alpha) \cdot R}$$

Donde P y R son la precisión y la exhaustividad del unigrama, $frag$ es la fracción de fragmentación y α, β y γ son 3 factores de penalización con valores 0,9, 3,0 y 0,5 respectivamente.

Recall-Oriented Understudy for Gisting Evaluation (ROUGE-L)

ROUGE-L provee un puntaje F basado en la secuencia común más larga (LCS). Suponiendo para X e Y que sus largos son m y n entonces se tiene:

$$P_{lcs} = \frac{LCS(X,Y)}{m}, R_{lcs} = \frac{LCS(X,Y)}{n}, F_{lcs} = \frac{(1+\beta^2) P_{lcs} R_{lcs}}{R_{lcs} + \beta^2 P_{lcs}}$$

Donde $\beta = P_{lcs}/R_{lcs}$ y F_{lcs} es el valor de ROUGE-L.

Consensus-based Image Description Evaluation (CIDEr)

Por lo general esta métrica se utiliza para medir la calidad de *captions* de imágenes, lo cual considera la frecuencia de n-gramas en las frases de referencia al hacer el cálculo TF-IDF de los pesos para cada n-grama. El puntaje $CIDER_n$ para un n-grama dado es computado usando la similitud por coseno promedio entre la frase candidata y la de referencia. El resultado final se obtiene al combinar los puntajes para 4 distintos tipos de n-gramas, sea esto para $n \in [1, 2, 3, 4]$. Esta última métrica a diferencia de las anteriores cuyos puntajes se encuentran en el rango $[0, 1]$, entrega números reales como resultado.

4.3. Selección y transformación del *dataset*

Para la selección del *dataset* se tuvo en consideración los límites de tiempo y complejidad de trabajo que podría significar el utilizarlos es su estado original. Se determino que la nueva red haría uso exclusivo de los AST's de los métodos del *dataset*, dado que estos ya pueden ser considerados como grafo y que se representarían los comentarios como secuencias de nodos que contuviesen una palabra cada uno. Sin embargo la obtención de dichas representaciones no estuvo exenta de dificultades. El pipeline utilizado con el fin de obtener las correctas interpretaciones del *dataset* para el trabajo con GGNN se detalla a continuación.

Selección de datos

Como primera aproximación se evaluó la posibilidad de transformar la secuencia de tokens de AST's presentes en el *dataset* a su representación original como árbol con nodos definidos. El principal problema de esto es que no era posible distinguir de manera intuitiva donde comenzaba un nodo y terminaba otro, por lo cual esta opción fue descartada. Luego se

optó por transformar nuevamente los métodos originales a su representación como árbol utilizando la clase AST's presente en la biblioteca base de *Python*. El problema de esto es que la clase AST tiene como requerimiento que la secuencia a ser transformada, respete las reglas de indentación de *Python*, lo cual en este caso no resultaba posible pues el método no contenía información respecto a su propio formato, es decir todo el código se encontraba en la misma linea. Este hecho significó un gran problema a la hora de considerar el *dataset* de RENCOS como *dataset* para la nueva herramienta dado que la elaboración de un parser que efectuase dicho formateo basado en las reglas de indentación podría tomar mucho tiempo y no funcionar de manera adecuada. Finalmente se decidió que se buscaría el *dataset* original del cual RENCOS extrajo sus datos de trabajo en primera instancia, dado que este posee los métodos con información respecto a los saltos de linea y tabulaciones.

Transformación de datos

Una vez seleccionado el *dataset* de trabajo se procedió a realizar la transformación de los métodos a AST y de los comentarios a una representación simple como grafo basándose en distintas implementaciones vistas en diversas publicaciones [8] [9] [12] [1].

Para la transformación a AST como primer paso se realizó la transformación de los métodos en su forma original a una compatible con el parser de la clase AST de *Python*. Para ello se realizó el reemplazo de las tokens “DCNL” a “\r” y “DCSP” a “\t” las cuales denotan un salto de línea y una tabulación respectivamente. Una vez verificado que a partir de estas secuencias efectivamente era posible obtener el AST del método, se inició el trabajo de transformación de AST a un formato de grafo compatible con las distintas herramientas disponibles de redes neuronales de grafos. En específico se adaptaron los AST para que fuesen compatibles con la biblioteca *Pytorch Geometric*, la cual extiende la funcionalidad de la conocida herramienta para el trabajo de aprendizaje de redes neuronales *Pytorch* para su uso con GNN's. Para ello se buscó una representación intermedia del AST como diccionario JSON el cual logró preservar la jerarquía de nodos pero facilitaba el trabajo de conversión desde la estructura de datos de *Python* a un arreglo de nodos y otro de aristas. Una vez realizada esta transformación se realizó un proceso similar para los comentarios objetivo de cada uno de los métodos. En este caso sin embargo dicha transformación no presentó mayores dificultades dado que se le dio una representación como grafo secuencial donde cada nodo correspondía a una palabra y sus aristas correspondían tanto a la palabra anterior como a la siguiente. Fueron investigados métodos alternativos de representación como los vistos en [3] sin embargo fueron descartados por simplicidad.

Tokenización y padding

Dado que se tuvo en consideración que se utilizarían tensores de *Pytorch* como representación de los datos, uno de los requerimientos fue transformar los arreglos de nodos los cuales contenían palabras como *features* y *labels*, a arreglos que contuviesen exclusivamente números enteros. Para esto se realizó un recuento de palabras distintas y a cada una de estas palabras le fue asignada un número la cual la distinguiría de las demás. Se utilizaron enteros positivos para palabras comunes y un par de números negativos para denotar caracteres especiales y de error. Además dado que los tensores deben tener dimensionalidades iguales se realizó un *padding* de los datos. Para el caso de las secuencias de AST, se determinó un numero fijo de *features* que serían consideradas. Dichas *features* consistieron en el tipo de nodo y su

nombre (por ejemplo se tiene por tipo atributo de función y como nombre "ctx"). Para los nodos que presentasen un número inferior al seleccionado de *features* se les otorgaría *features* de *padding* hasta alcanzar el numero deseado y para los que presentasen un mayor numero simplemente se les truncaría para que tuviesen la cantidad adecuada. Para el caso de las secuencias de palabras de los comentarios se determinó cual fue la secuencia con la mayor cantidad de nodos y se les entregó a todas las secuencias que tuviesen menos, una cantidad de nodos comodín que les permitiese llegar al número deseado.

El resultado final fue la representación de los nodos como arreglos de la misma cantidad de *features* numéricas para lo AST y como arreglos de la misma cantidad de nodos para los comentarios.

4.4. Implementación de la red

Una vez transformados los datos y así adquirido el *dataset* de trabajo, se procedió a realizar la implementación de la red neuronal con la arquitectura específica de GGNN. Para ello se realizaron pruebas con dos implementaciones distintas de esta arquitectura listadas a continuación.

4.4.1. Implementación utilizando *Pytorch Geometric*

Se tuvo en mente como primer acercamiento a la implementación de red el utilizar la biblioteca *Pytorch Geometric*. La transformación de datos discutida en la sección anterior entonces considera el formato de grafos aceptado por las clases provistas por dicha biblioteca. El problema sin embargo fue que no fue posible encontrar una documentación que detallase en mejor medida cómo debiese utilizarse el operador de convolución de las GGNN o así ejemplos de uso. Es por este motivo que no se logró la implementación exitosa de la red y por consiguiente se descartó la posibilidad de continuar el trabajo utilizando esta herramienta.

4.4.2. Implementación utilizando ONMT

Como alternativa a la biblioteca antes mencionada se tuvo la librería *Open Neural Machine Translator* (ONMT), también utilizada por la herramienta RENCOS. Para poder hacer uso de esta herramienta se debió volver a realizar una transformación del *dataset*, sin embargo dado que ya se disponía de este en un formato amigable con el esquema de grafos dicha transformación no resultó particularmente difícil. Si bien la implementación fue capaz de lograr la compilación correcta del código e iniciar el entrenamiento de un modelo GGNN, las primeras pruebas del código estuvieron lejos de ser ideales dado que no se logró converger a un resultado apropiado.

Se realizó un único entrenamiento de prueba con una muestra pequeña del *dataset* a modo de observar el comportamiento de dicha herramienta.

Capítulo 5

Validación y Resultados

Este capítulo se enfoca en el proceso de validación del trabajo realizado. De acuerdo a lo previamente estipulado, se realizaría la evaluación del rendimiento de la herramienta como una comparación directa a los resultados obtenidos para RENCOS en las distintas métricas de evaluación de calidad de resumen generado previamente descritas así como también en su rapidez de convergencia, tamaño del *dataset* de entrenamiento utilizado, disminución de entropía cruzada entre otros. Sin embargo dado que el modelo no fue correctamente entrenado, este no fue capaz de producir un output utilizando el *set* de *testing*. Al no poseer dicho output no fue posible obtener un puntaje en base a las métricas BLEU, Meteor, ROUGe o Cider. Finalmente sin dicho puntaje no fue posible hacer la evaluación principal de la herramienta, que consistía exactamente en la comparación en los puntajes obtenidos de ambas herramientas, independiente del tamaño del *dataset* o la cantidad de iteraciones de entrenamiento.

Con lo anterior en mente primero se detallan los *dataset* utilizados, los parámetros de entrenamiento utilizados para los distintos entrenamientos de modelo realizados, las especificaciones de hardware y luego los resultados obtenidos en las métricas de evaluación de calidad de resumen generado para dos ejecuciones en ambientes distintos para la herramienta RENCOS. Finalmente se comparan los datos obtenidos durante los entrenamientos de los modelos.

5.1. Datos de Prueba

En esta sección se detallan los datos utilizados y los ambientes de ejecución utilizados.

5.1.1. RENCOS

Para la herramienta RENCOS se utilizaron los mismos datos estipulados en el paper asociado a la herramienta, los cuales constan de 108726 pares de código-comentario utilizados para entrenamiento y evaluación. Se efectuó el entrenamiento utilizando una capa de *embbeding* de tamaño 256 y las dimensiones de estados escondidos de las unidades LSTM equivalentes a 512. El tamaño del *batch* fue de 32 y el numero total de iteraciones fue de 100000. Se utiliza el optimizador Adam con un *learning rate* de $0,001$. El modelo fue entrenado en

dos ambientes diferentes:

- Local: 3.2GHz CPU, 16GB RAM y 2GB GPU.
- *Google Colaboratory*: 2.3GHz CPU, 12GB RAM, 12GB GPU.

5.1.2. GBSCS

Para la herramienta desarrollada para este informe se utilizaron sólo 30,000 AST'S distintos para el entrenamiento dado que un número mayor hubiese ocasionado que este tomase demasiado tiempo. Es importante destacar la cantidad de nodos asociados a estos AST's pues la cantidad de nodos presentes en ellos afecta en gran medida el tiempo de ejecución del entrenamiento.

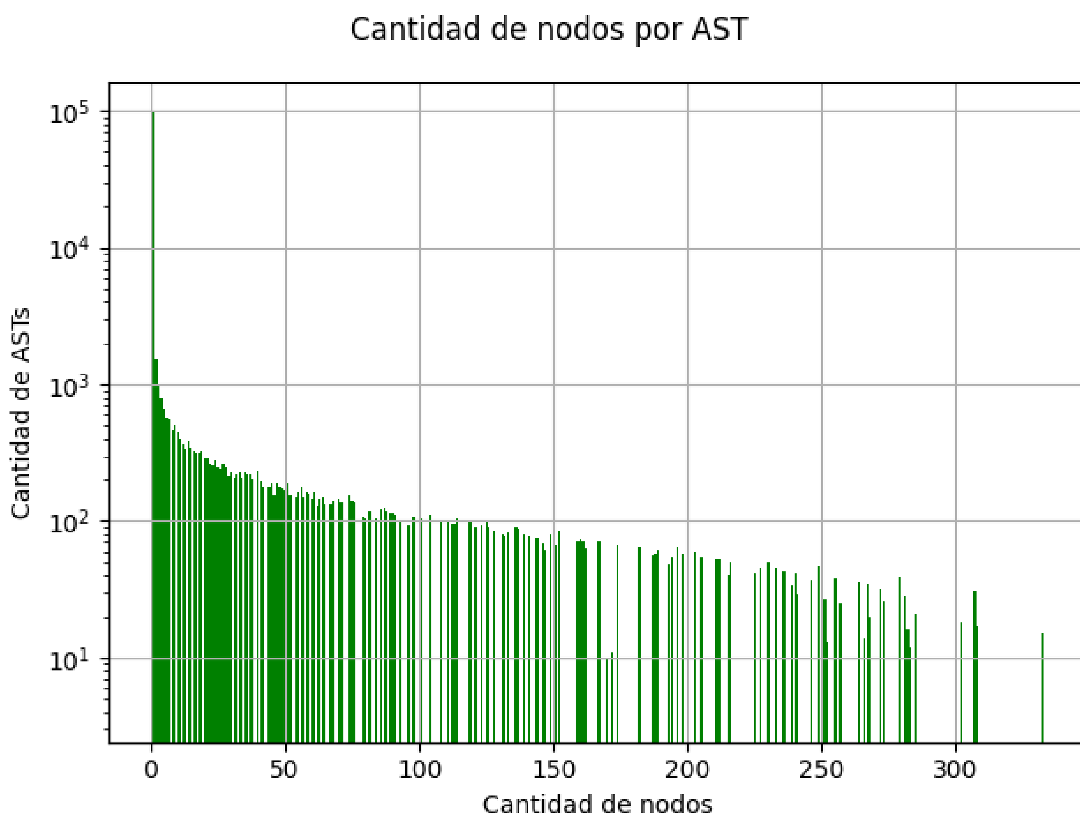

Figura 5.1: Numero de nodos por AST del *training set*

Realizando una inspección rápida del gráfico anterior es posible notar que a pesar de que la mayor concentración de AST's tiene un número inferior a 50 nodos, bastantes de ellos poseen más que esta cifra, lo cual sumado a que la cantidad *default* de *features* consideradas fue de 4 por nodo añaden un grado bastante alto de dimensionalidad y por consiguiente tiempo de cómputo.

Con lo anterior en mente se tuvo para este entrenamiento una capa de *embbeding* de tamaño 128 con una dimensionalidad de estados escondidos de igual tamaño. El tamaño de

batch utilizado fue de 4 y el numero total de iteraciones fue de sólo 10,000 pasos. El *learning rate* fue de $0,002$.

Dado que en este caso también se utilizó la plataforma *Google Colaboratory* como ambiente de ejecución las especificaciones de hardware fueron las mismas que en la sección anterior.

5.2. Calidad del resumen

En un escenario ideal se hubiese realizado la comparación directa de ambas herramientas de acuerdo a sus calificaciones en las distintas métricas de evaluación de calidad de resúmenes. El problema sin embargo es que el modelo entrenado por la herramienta aquí desarrollada presentó complicaciones que no fueron capaces de ser resueltas a la hora de entrega de este informe. Se detallarán las razones de porqué más adelante. Sin embargo si pudieron ser reproducidos resultados para la herramienta base de comparación

5.2.1. Resultados de las métricas en ambiente local

Originalmente esta herramienta fue ejecutada en el denominado ambiente "local" con especificaciones de hardware bastante inferiores a las especificadas por la documentación de aquella. Como resultado de esto, el modelo no pudo ser correctamente entrenado y por consiguiente los resúmenes generados por la herramienta obtuvieron calificaciones muy por debajo del estándar estipulado por la herramienta. Estos resultados pueden verse a continuación.

Ambiente	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Meteor	ROUGe	Cider
Paper RENCOS	0.430	0.295	0.242	0.207	0.211	0.475	2.449
Local	0.304	0.147	0.088	0.060	0.124	0.375	1.071

Tabla 5.1: Resultados obtenidos en ambiente local versus presentados en paper RENCOS

5.2.2. Resultados de las métricas en *Google Colaboratory*

En consideración con lo anterior, se cambió el ambiente de trabajo local a uno remoto en la plataforma *Google Colaboratory*. Las especificaciones de hardware de esta plataforma fueron bastante más poderosas que las del ambiente local con lo que esta vez pudo realizarse el correcto entrenamiento del modelo. Los resúmenes obtenidos esta vez obtuvieron calificaciones apenas inferiores a las presentadas en el Paper de la herramienta con lo cual se estimó que fue en efecto replicado el correcto funcionamiento de la herramienta. Estos resultados son los siguientes.

Ambiente	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Meteor	ROUGe	Cider
Paper RENCOS	0.430	0.295	0.242	0.207	0.211	0.475	2.449
Local	0.304	0.147	0.088	0.060	0.124	0.375	1.071
Google Colaboratory	0.421	0.283	0.233	0.196	0.209	0.455	2.338

Tabla 5.2: Resultados obtenidos en plataforma *Google Colaboratory* versus los previamente expuestos

5.3. Resultados de entrenamiento del modelo

Como métrica auxiliar de evaluación se compararon los resultados obtenidos respecto al entrenamiento del modelo. Dado que se utilizó la misma biblioteca para el desarrollo de ambos modelos y que además ambos modelos fueron entrenados usando el mismo hardware (de esta sección en adelante se utilizan sólo datos de modelos entrenados en *Colaboratory*), estos resultados son altamente contrastables. Las métricas de comparación son las siguientes:

- Accuracy (acc): La métrica de cuantos aciertos tuvo el modelo respeto a la muestra total. Valores en el rango $[0, 1]$.

- Perplexity (ppl): Es una medida que tan buenas son las predicciones realizadas por una distribución de probabilidad o un modelo probabilistico, en este caso la red neuronal. A menor valor mejores resultados.

- Cross-entropy (xent): Es utilizada para medir la diferencia entre dos distribuciones de probabilidad.

- Learning rate (lr): parámetro que determina el tamaño del paso hacia el mínimo de la función de pérdida para cada iteración.

- Elapsed Time: Tiempo transcurrido durante un paso de entrenamiento.

Es necesario destacar sin embargo que el modelo de la herramienta RENCOS fue entrenado utilizando 100000 pasos con validación de los datos cada 10000 pasos, (es decir en total 10 validaciones del modelo), mientras que el modelo de la herramienta en la aquí desarrollada tesis fue entrenado utilizando solo 10000 pasos sin hacer una validación del modelo. Debido a lo anterior cada una de las siguientes secciones realizará una comparación de las métricas de evaluación de forma conjunta para ambas herramientas pero sólo durante los primeros 9500 pasos en el caso de RENCOS. Luego de la misma manera para cada sección se dispondrá la progresión de cada métrica para la totalidad de los pasos de RENCOS es decir todos los 100000 pasos.

En las siguientes secciones se hacen las comparaciones directas de las distintas métricas de evaluación previamente expuestas

5.3.1. Accuracy

Esta métrica como se estipuló en la sección anterior se determina mediante la fórmula:

$$acc = \frac{N_{aciertos}}{N_{total}}$$

Donde $N_{aciertos}$ equivale al número de predicciones acertadas y N_{total} equivale al número de predicciones totales. Dado que $N_aciertos$ es siempre menor o igual a N_{total} el valor de acc siempre estará en el rango $[0, 1]$.

Los valores de esta medición se encuentran en el gráfico de la Figura 5.2. Es posible observar en este gráfico que la curva de precisión (accuracy) para la herramienta RENCOS presenta un crecimiento promedio bastante mayor al de la herramienta GBSCS, la cual adopta un com-

portamiento bastante símil al de una función constante para el rango de pasos considerado.

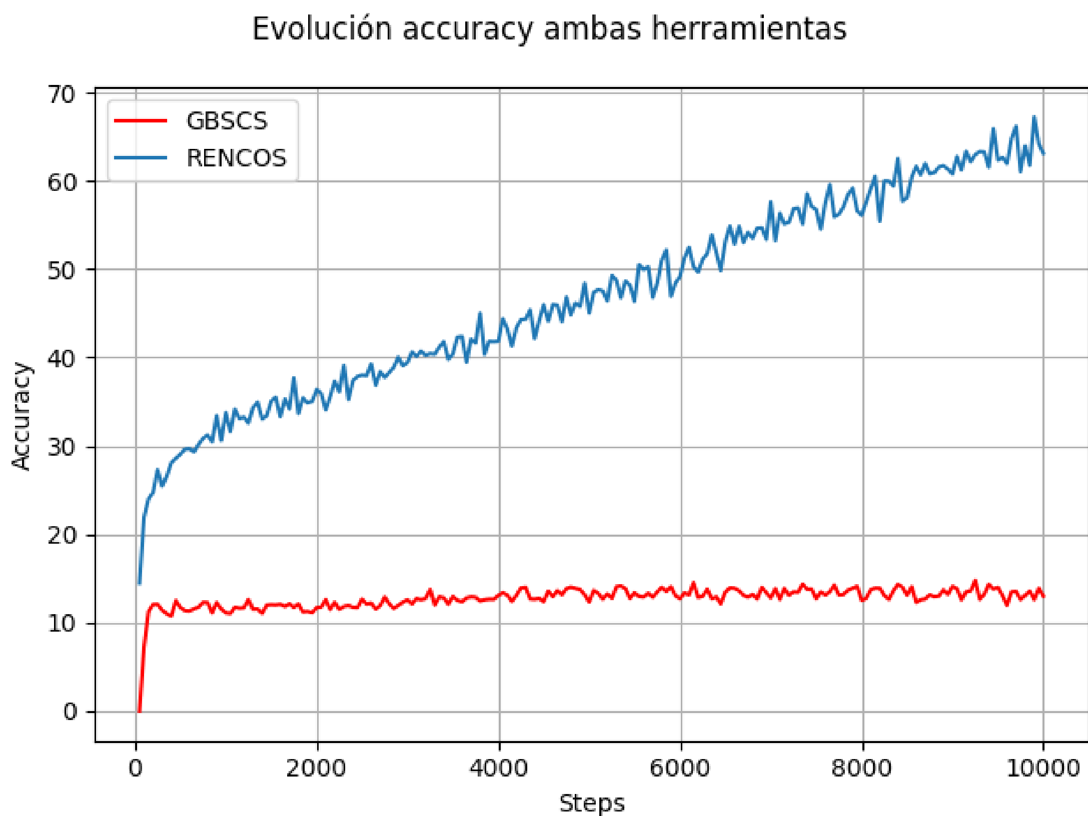

Figura 5.2: Evolución de precisión para ambas herramientas durante los primeros 10000 pasos

A manera de simplificar la visualización del crecimiento de las curvas de precisión de las herramientas se les aplicó una regresión lineal dispuesta en el gráfico de la Figura 5.3. Es posible observar con claridad la diferencia entre las magnitudes de las pendientes de ambas.

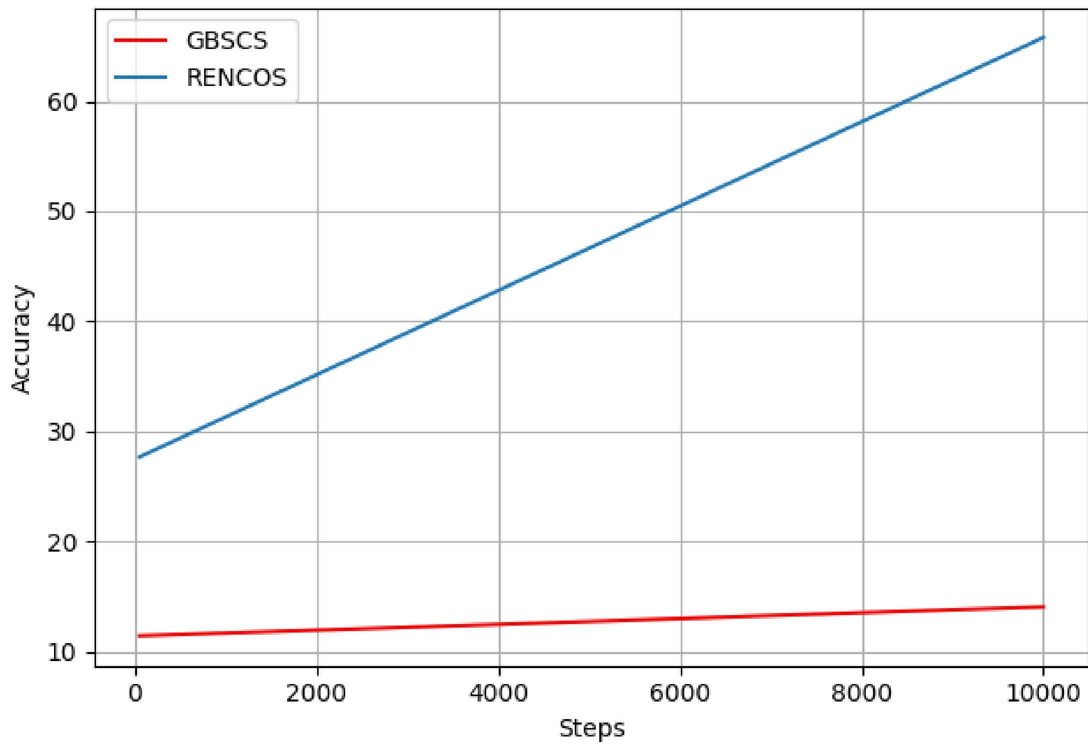

Figura 5.3: Evolución de precisión para ambas herramientas durante los primeros 10000 pasos después de realizar una regresión lineal

Perpetuando el estudio de esta métrica para los pasos restantes de la herramienta RENCOS, queda en evidencia el carácter de crecimiento logarítmico y con tendencia a 1 de la precisión dispuesta gráficamente en la Figura 5.4.

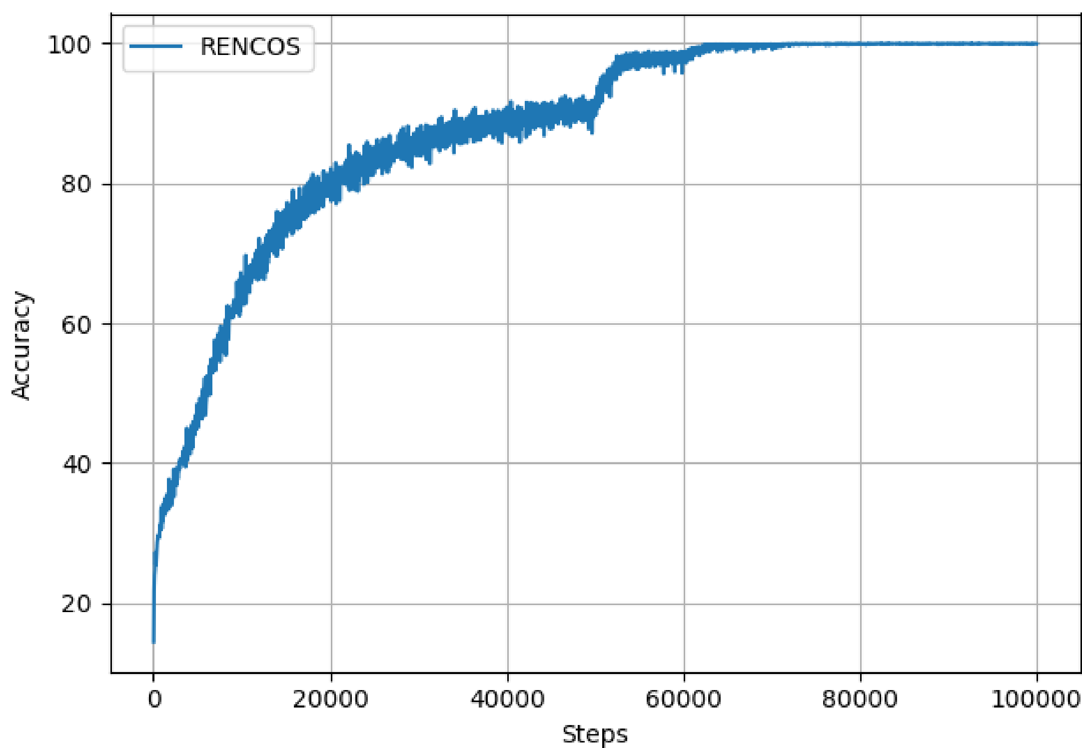

Figura 5.4: Evolución de precisión para la herramienta RENCOS durante el transcurso de todo su entrenamiento

5.3.2. Perplexity

Para esta métrica se tiene que mientras más pequeño sea el valor, mejores serán las predicciones del modelo. Con ello en mente se presentan los valores de esta métrica ajustados logarítmicamente para la dos herramientas de estudio en la Figura 5.5. Aquí se denota claramente la brecha entre ambas herramientas al quedar en evidencia la basta distancia entre los valores obtenidos para cada una de las herramientas. Los valores de GBSCS obtenidos son en todo momento superiores a los de RENCOS en al menos un orden de magnitud, incrementando este valor en dos ordenes al acercarse al paso 8,000. Aquí además es posible observar un decrecimiento logarítmico para ambas herramientas.

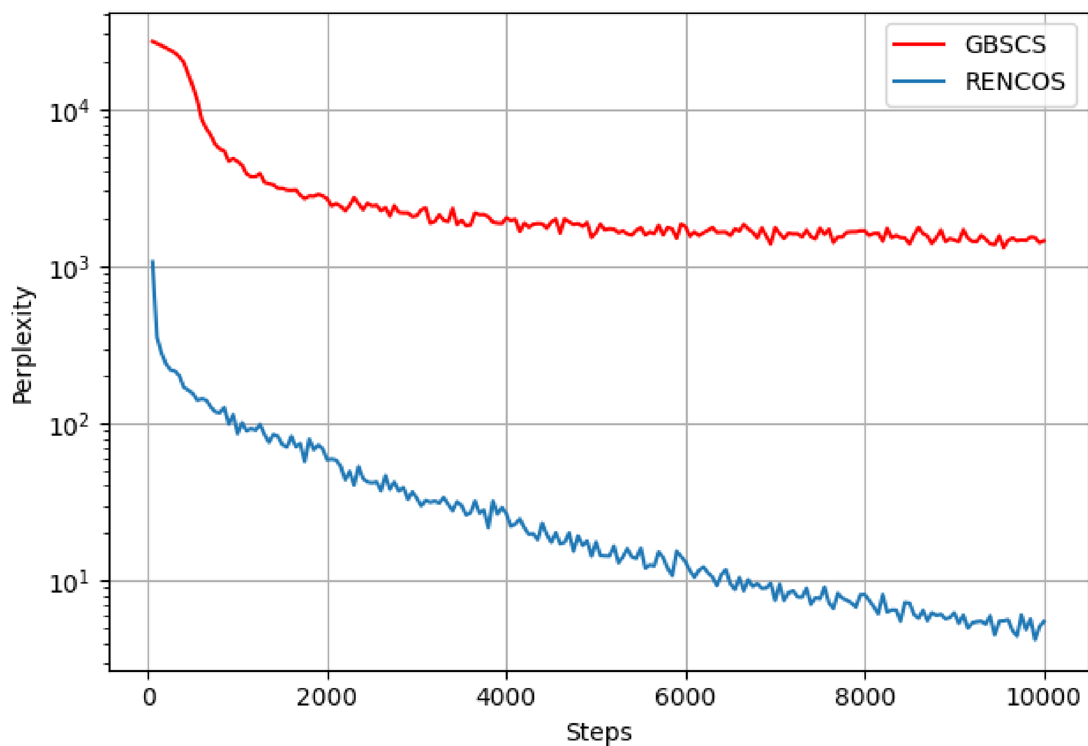

Figura 5.5: Evolución de perplexity para ambas herramientas durante los primeros 10000 pasos

Para los pasos restantes de RENCOS se perpetúa la tendencia de este valor a decrementar su valor acercándose bastante a 0 durante los últimos pasos.

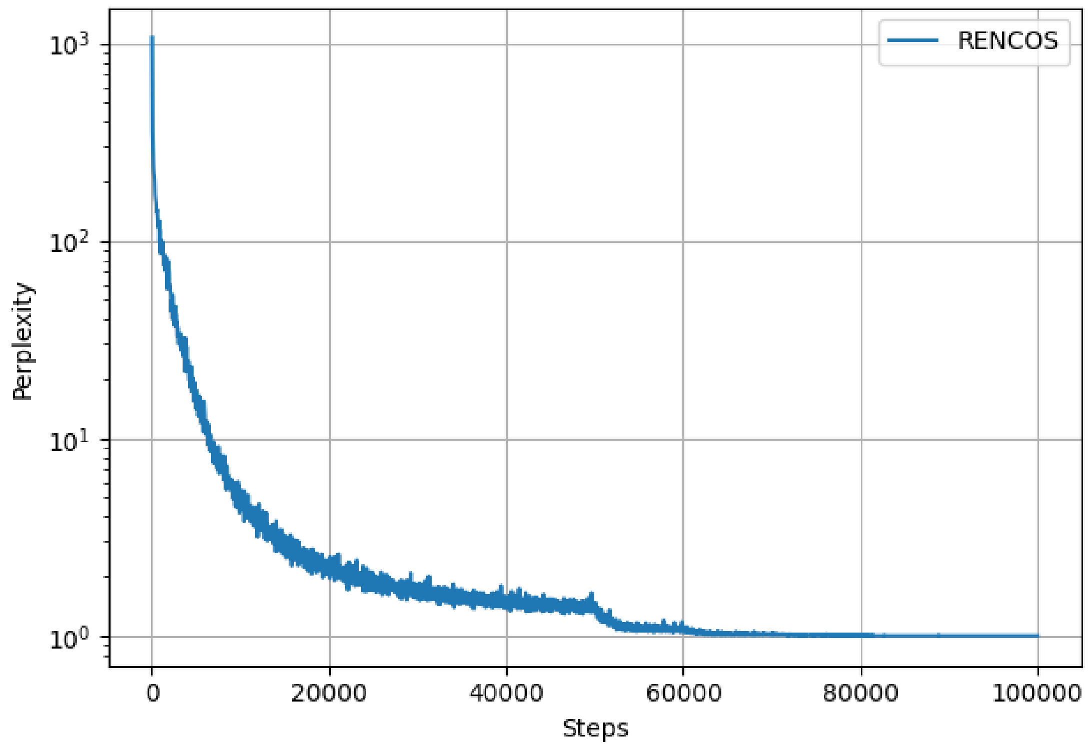

Figura 5.6: Evolución de perplexity para la herramienta RENCOS durante el transcurso de todo su entrenamiento

5.3.3. Cross-entropy

Dado que la métrica de perplexity cuantifica que tan buenas son las predicciones hechas por un modelo probabilístico y que la entropía cruzada (cross-entropy) es utilizada para medir la diferencia entre dos distribuciones de probabilidad es claro notar la estrecha relación de ambas dado que conforme crece la capacidad del modelo de realizar buenas predicciones crece también su similitud al modelo que está intentando predecir con lo cual decrece la diferencia entre ambos. Este carácter queda en evidencia al examinar la evolución de la entropía cruzada dispuesta en la Figura 5.7 para ambas herramientas donde se denota de igual manera que en la sección anterior el carácter de decrecimiento logarítmico de la función de esta métrica.

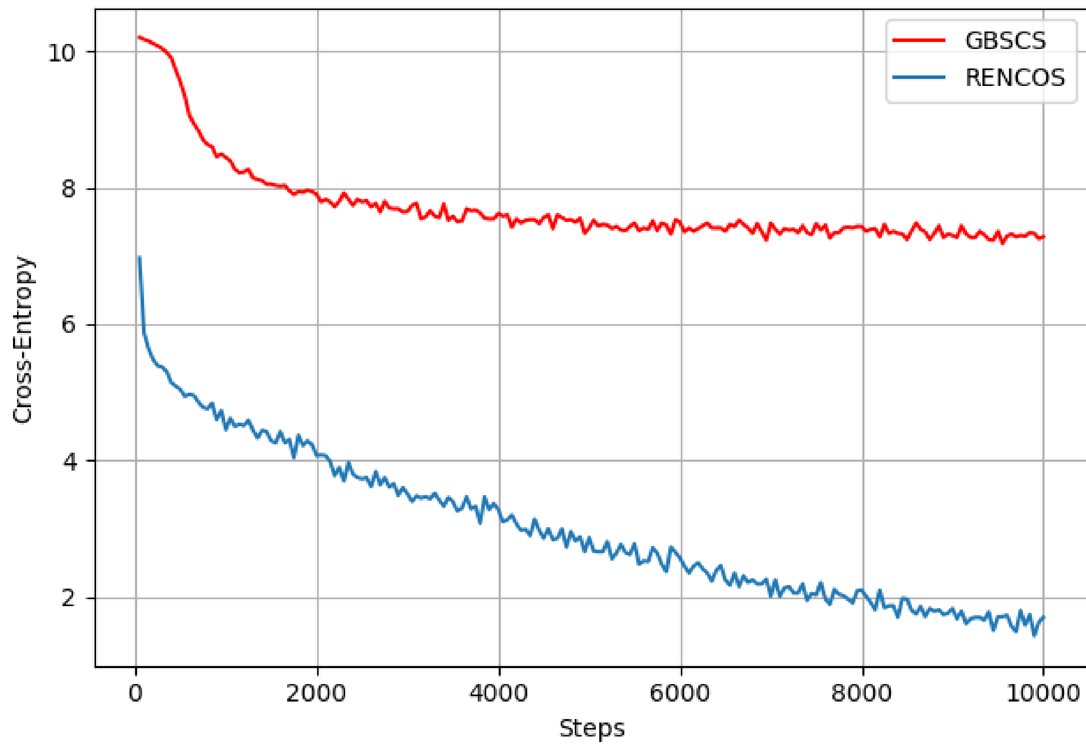

Figura 5.7: Evolución de la entropía cruzada para ambas herramientas durante los primeros 10000 pasos

Al igual que para la función de precisión se realiza una regresión lineal de ambas curvas de entropía las cuales presentan pendientes similares, no bien desplazadas en el eje de las ordenadas. Estas curvas se encuentran en la Figura 5.8.

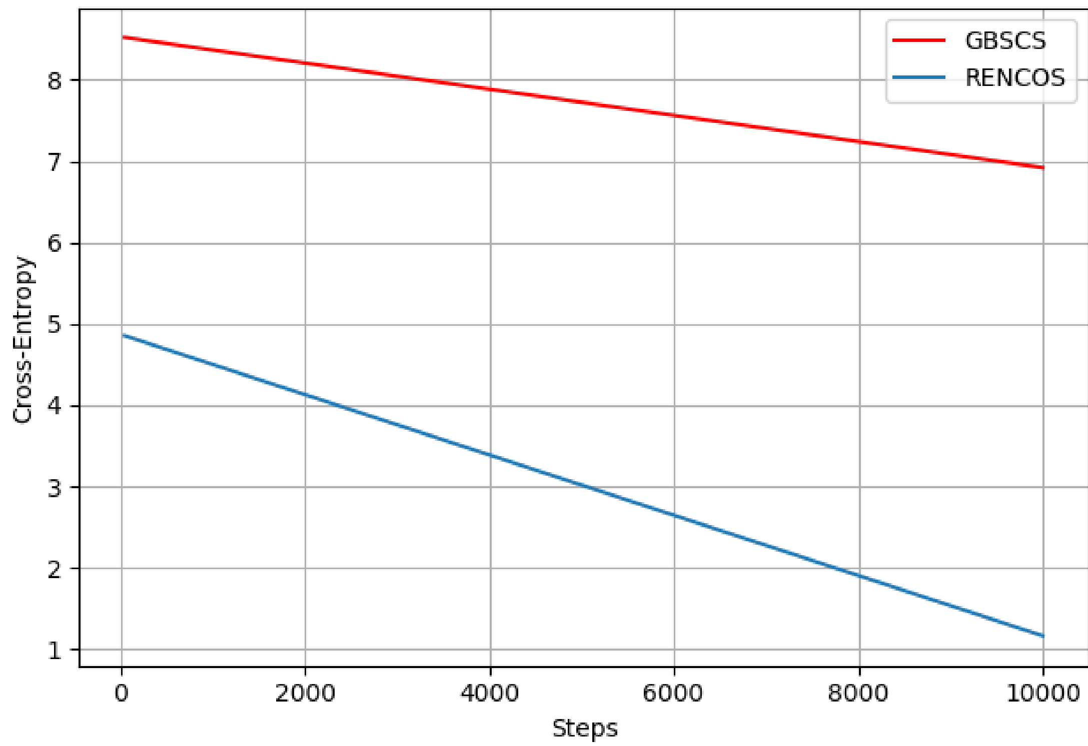

Figura 5.8: Evolución de la entropía cruzada para ambas herramientas durante los primeros 10000 pasos después de realizar una regresión lineal

Al igual que en las secciones anteriores se dispone el resto de la evolución de este valor para los pasos restantes del entrenamiento del modelo RENCOS. Aquí de igual manera que en la curva de perplexity es posible notar la tendencia del valor a 0 con decrecimiento logarítmico.

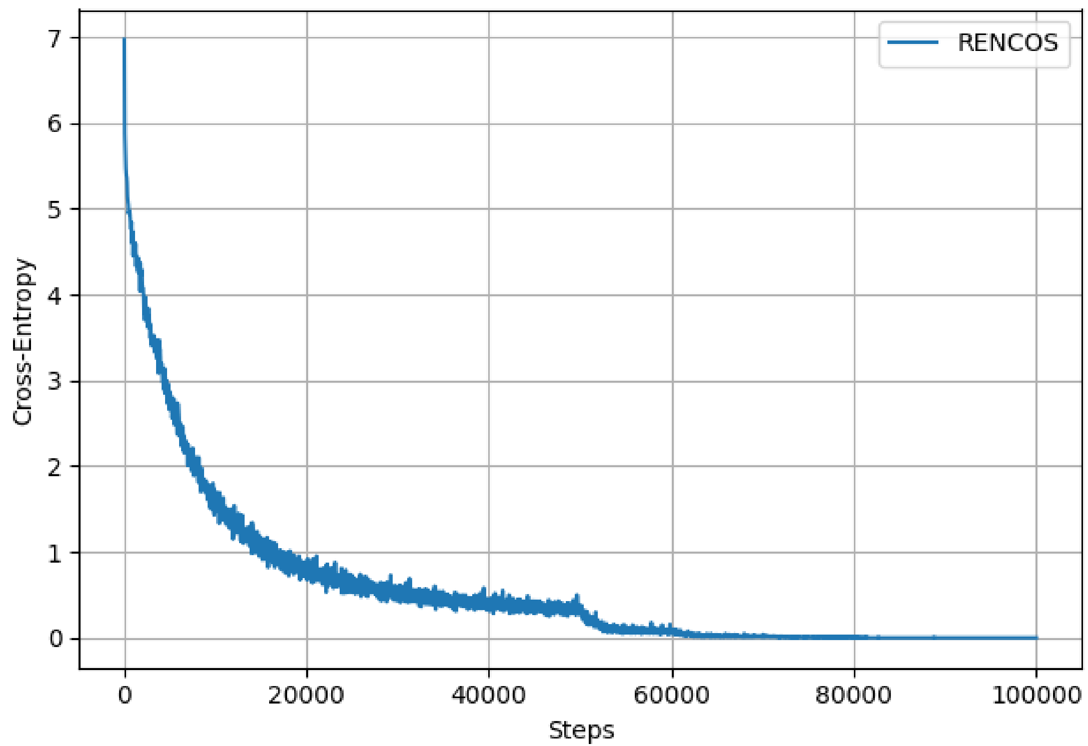

Figura 5.9: Evolución de la entropía cruzada para la herramienta RENCOS durante el transcurso de todo su entrenamiento

5.3.4. Learning rate

Se ilustra la evolución del parámetro de aprendizaje para ambos modelos, el cual es ajustado por la implementación de la red de manera automática basándose en el desempeño mismo de la minimización de la función de pérdida para los modelos. En primera instancia es posible notar que para los primeros 10,000 pasos no hay cambio alguno en este parámetro para la herramienta RENCOS, en cambio para la herramienta desarrollada este valor es decrementado a un valor inferior al 50 % del valor inicial.

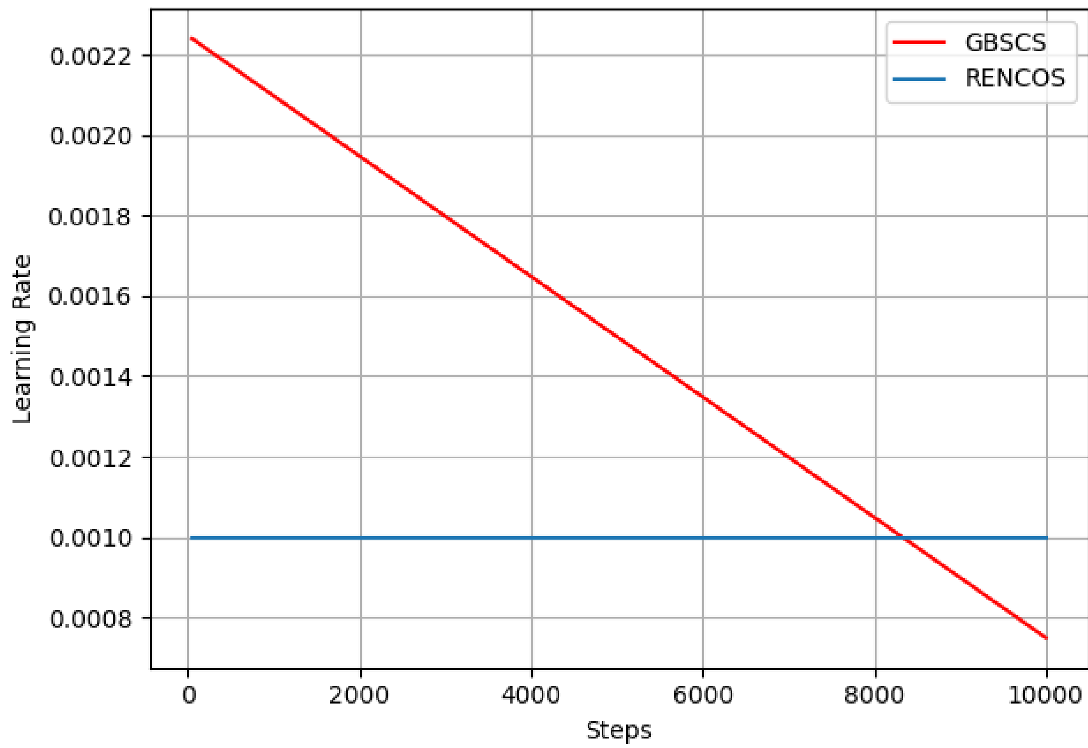

Figura 5.10: Evolución del factor de aprendizaje para ambas herramientas durante los primeros 10000 pasos

Una vez continuado el estudio para RENCOS queda en evidencia el cambio en el valor de este factor el cual decrementa a una fracción inferior al 20 % del valor original.

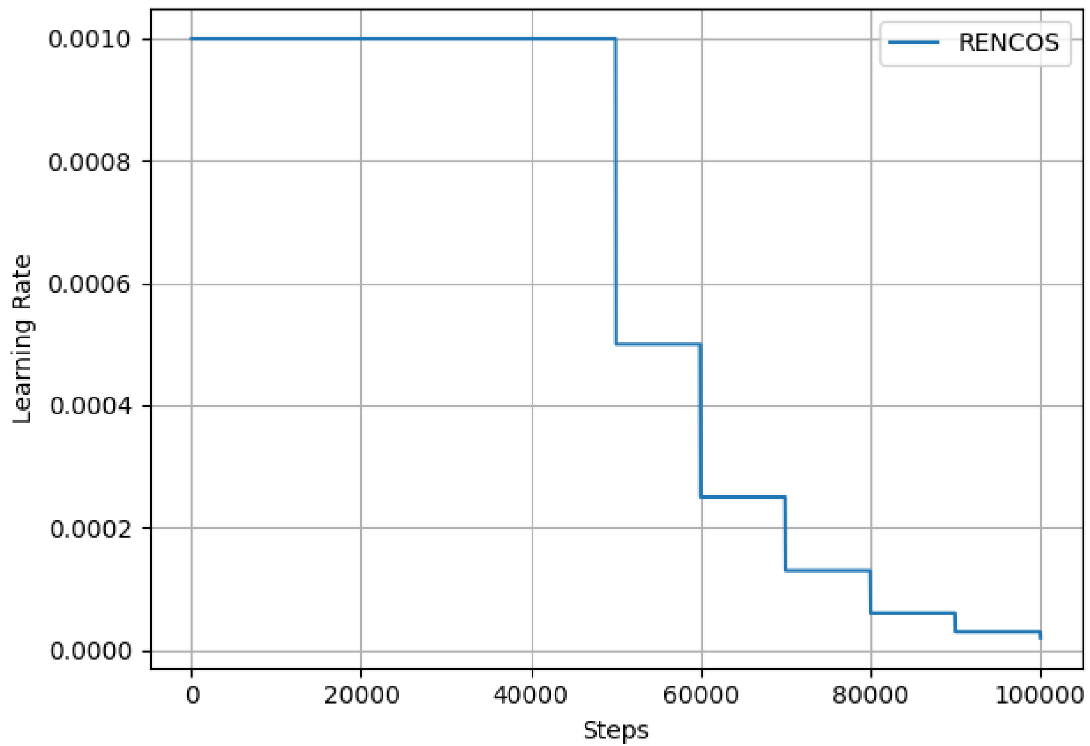

Figura 5.11: Evolución del factor de aprendizaje para la herramienta RENCOS durante el transcurso de todo su entrenamiento

5.3.5. Elapsed time

Como medida final de evaluación de rendimiento se mide el tiempo entre pasos de cada herramienta. Es posible notar a modo general que los valores de esta métrica para ambas herramientas se mantienen dentro de un rango bastante disminuido de valores, pero con una amplia distancia entre ambos siendo el tiempo de ejecución promedio para GBSCS unas 40 veces más grande que el de RENCOS.

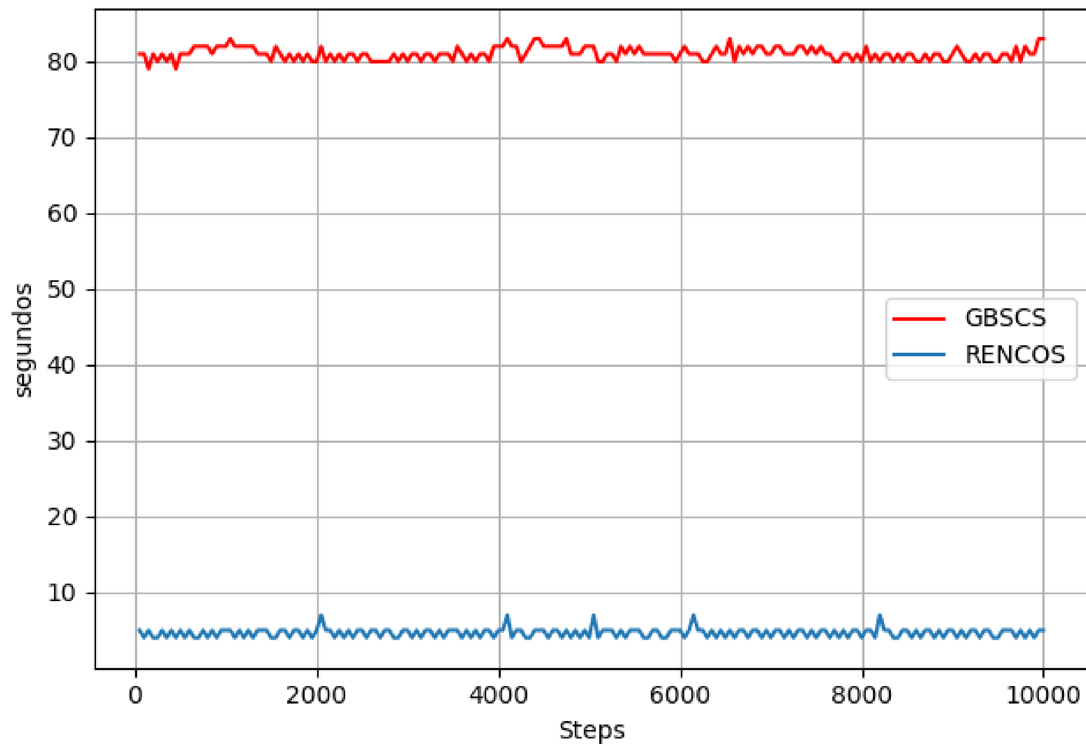

Figura 5.12: Evolución del factor de aprendizaje para ambas herramientas durante los primeros 10000 pasos

Proyectando los pasos restantes para la RENCOS sin embargo, es posible observar crecimientos repentinos pero estructurados de tiempos de paso superiores en gran medida a al valor promedio de esta métrica.

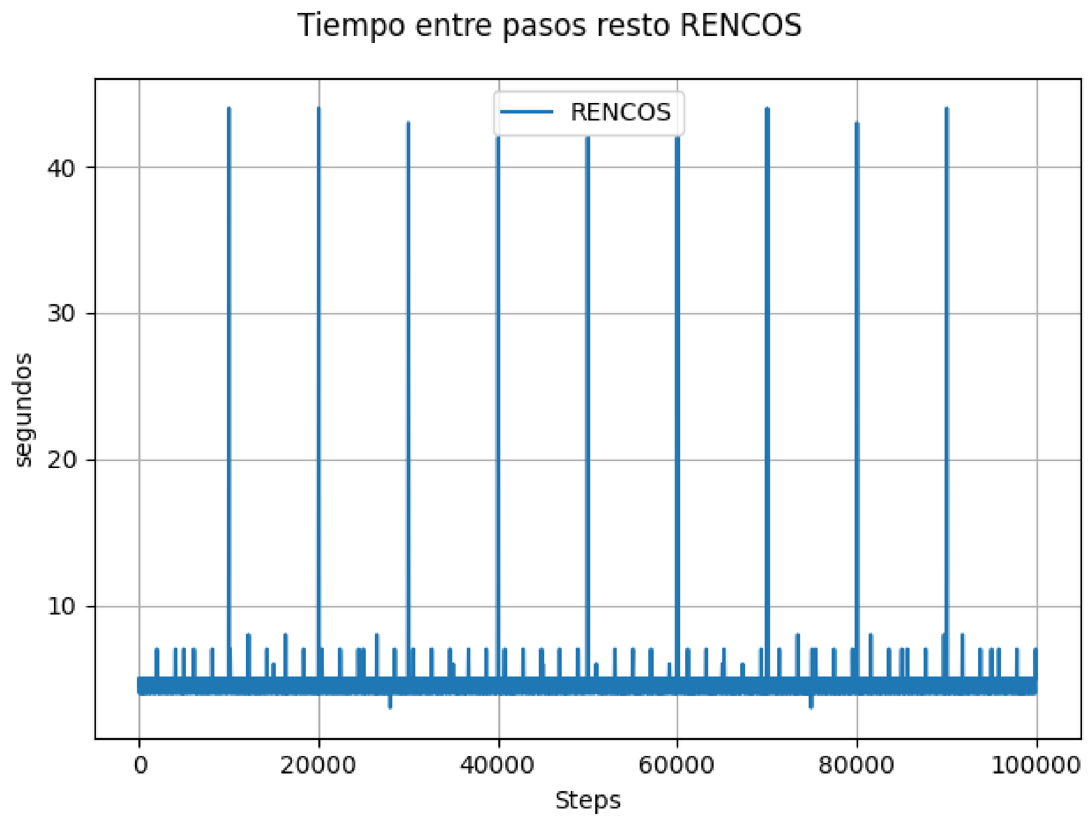

Figura 5.13: Evolución del factor de aprendizaje para la herramienta RENCOS durante el transcurso de todo su entrenamiento

Capítulo 6

Discusión

Este capitulo se reserva para la reflexión ante los resultados obtenidos tras realizar el entrenamiento de la herramienta. Además se plantean posibles mejoras y caminos no explorados durante el desarrollo de este trabajo.

6.1. Análisis de Resultados

6.1.1. Resúmenes entregados

En un caso ideal, el modelo entrenado durante el transcurso de esta memoria sería capaz de producir resúmenes de fragmentos de texto usados durante la fase de *testing*.
Sin embargo se considera que dichos resúmenes no tendrían el grado de calidad alcanzado por la herramienta RENCOS dado que para la herramienta GBSCS no se efectúa un enriquecimiento de las probabilidades condicionales de las palabras del resumen siendo generadas como es en el caso de la anterior herramienta. Con esto en mente se estima que el modelo padecería del problema general de las herramientas de traducción de texto basadas en aprendizaje neuronal el cual es que se omitirían las palabras de baja frecuencia dando privilegio las palabras más comunes a la hora de confeccionar el resumen.
Por otra parte se cree que la pérdida de dimensionalidad realizada con el fin de entrenar el modelo en un tiempo razonable pudo haber afectado de forma bastante adversa al resultado final. Es un problema en su propia medida determinar el grado de riqueza del input con el cual se efectuará el trabajo, dado que muy poco y el modelo no será demasiado distinto a los modelos que usan secuencias de tokens en vez de grafos y se utiliza demasiado el modelo podría tardar demasiado en lograr converger a un buen mínimo de la función de pérdida.
Un ejemplo de esta explosión dimensional como se estipuló en la sección de resultados es que un sin numero de AST's estuvieron conformados por más de 50 nodos cada uno de aquellos con por lo menos 4 *features* con alto grado de dispersión. Dichas *features* corresponden solo al tipo de nodo, su nombre designado por *python*, su nombre asignado por el usuario y la id interna del nodo. Cabe destacar que existieron nodos de por lo menos 14 *features* distintas.
En un inicio se intentó que el modelo considerase todas las *features* de los nodos pero la memoria RAM del ambiente fue rápidamente agotada en consecuencia.

6.1.2. Entrenamiento de los modelos

Una vez observados los distintos gráficos comparativos de ambas herramientas es bastante fácil extrapolar que el modelo no está siendo correctamente entrenado. La función de evolución de precisión del modelo es el principal indicador de progreso respecto del aprendizaje y el hecho de que esta curva tenga una pendiente casi nula no es buen pronóstico para la evolución del modelo. Además considerando que a pesar de que el modelo de RENCOS fue entrenado por una cantidad 10 veces superior de pasos a GBSCS, la primera fue capaz de superar a la segunda en virtualmente todo ámbito durante solo sus primeros 10000 pasos. Lo anterior propicia buscar una alternativa de representación del input u una arquitectura diferente para la red neuronal que sea capaz de lograr generalizaciones más prometedoras.

6.2. Propuestas de mejora

Como propuesta general de mejora a la metodología empleada, sería remover el carácter secuencial que esta presenta. Dicho esto se insta a abordar un modelo metodológico capaz de realizar adelantos en cada una de las fases de manera simultánea en vez de comenzar una nueva fase solo una vez que la fase anterior se encuentre finalizada en su totalidad. Los modelos secuenciales de trabajo son generalmente bastante propensos a los denominados "cuellos de botella" los cuales pueden ralentizar e incluso detener el trabajo por un tiempo considerable. A continuación se listan las mejoras propuestas separadas por la fase de desarrollo de la metodología propuesta.

6.2.1. Investigación Preliminar

El enfoque utilizado para el desarrollo de esta fase en el transcurso de esta memoria fue en su totalidad enfocado en el marco teórico del funcionamiento de las redes neuronales recurrentes.
Se propone como un acercamiento más conveniente enfocar la atención en aplicaciones prácticas de este tipo de redes usando el marco teórico solo como material complementario a la experiencia práctica.
Las redes neuronales constan de muchos componentes distintos, ya sea la arquitectura de sus neuronas, sus optimizadores, uso de funciones de pérdida, métodos de captación de input, etc. La correcta armonía y funcionamiento entre aquellos componentes se estima que es más sencillo de visualizar realizando pruebas prácticas de funcionamiento, alterando sólo un aspecto de los parámetros a la vez y observando así su nuevo comportamiento. Se estima que se debe tener un grado bastante amplio de familiaridad con cada componente para lograr una correcta implementación de cualquier tipo de red neuronal.

6.2.2. Implementación y recreación de resultados

Para esta etapa del trabajo una posible mejora consistiría en la ejecución de múltiples instancias de entrenamiento de la herramienta base. Dado que los resultados obtenidos si bien no difieren demasiado, aún así se clasifican como inferiores a los obtenidos por el equipo de desarrollo de esta herramienta. Para un experimento cualquiera es igual de importante que los resultados de este mismo, el poder reproducir las condiciones que los entregan. Con lo anterior en mente es bastante beneficioso el corroborar que efectivamente la herramienta es capaz

de obtener los resultados que estipula conseguir o cerciorarse que se están implementando correctamente las condiciones para obtener dichos resultados.

6.2.3. Segunda fase de investigación y diseño de GNN

De forma similar a la fase de investigación preliminar se considera más importante conocer cómo implementar la determinada estructura de red neuronal escogida a conocer cada detalle de su funcionamiento de forma teórica. Hechos que corroboran lo anterior corresponden por ejemplo a los primeros intentos realizados para entrenar un modelo de GGNN los cuales se estimaron fallidos por falta de experiencia o falta de familiaridad con la librería o framework de trabajo.
Como mejora específica a esta parte podría considerarse una implementación de la red que solo altere el funcionamiento intermedio de la herramienta base RENCOS, es decir la interfaz entre el input y el output. Debido a que finalmente se utilizó una implementación de GGNN proveniente de la misma biblioteca en la cual se basó la herramienta original se estima bastante plausible lograr extender el software base de RENCOS para que incorpore trabajo con GNN's.

6.2.4. Implementación GNN

La principal dificultad encontrada en esta fase del desarrollo del trabajo fue lograr entender el funcionamiento de las variadas implementaciones de GGNN encontradas. Una solución posible para este problema es realizar una implementación propia de este modelo que sirva como primer acercamiento a otra que ya haya sido estudiada, diseñada y optimizada con mayor grado de preparación dado que la principal dificultad se encuentra en la configuración de los distintos parámetros discutidos en la sección SECCIÓN. Como mejoras propias del modelo de GNN podrían consistir en modularizar las capas de *embbeding* y de representación interna de los nodos de acuerdo a [2][4].

6.2.5. Comparación de resultados

Finalmente para esta fase del trabajo se propone como mejora realizar comparaciones con otras herramientas que hagan uso de GNN's para el trabajo con texto.

Capítulo 7

Conclusión

Las redes neuronales de grafos constan de una herramienta poderosa a la hora de realizar predicciones sobre datos entrelazados por relaciones, ya sean de dependencia, equivalencia, continuidad, orden, etc. El problema recae a la hora de escoger cuales de estas relaciones son realmente relevantes a la hora de construir un modelo predictivo. Como fue explorado en este proyecto el considerar una gamma demasiado amplia de relaciones o características distintivas puede significar una pérdida de generalidad así como también una sobrecarga de cálculos no necesariamente primordiales en la construcción una buena herramienta predictiva. Con el contexto anterior entonces el trabajo realizado durante esta memoria consistió en la proposición de un modelo de red neuronal de grafos capaz de traducir código fuente a comentarios en lenguaje natural, que si bien no logró producir resultados concretos, sirvió como primer acercamiento a una herramienta de calidad que si pueda utilizar la potencialidad de los grafos sin perderse en el océano de posibilidades.

Uno de los problemas colaterales más comunes encontrados en el desarrollo de esta memoria consistió en problemas de compatibilidad entre los distintos componentes de software que comformaban la herramienta RENCOS, más en específico la correcta instalación del motor de búsqueda Lucene.

Uno de los hitos más grandes que tuvo el trabajo en esta memoria fue destacar la importancia de construir más de una iteración de un modelo predictivo. La estrategia original planteada para el proyecto era de realizar el diseño completamente teórico del modelo que se haría de construir sin hacer ensayos o pruebas de concepto previas. Mientras que esto es plausible que produzca resultados en ciertos ámbitos de las ciencias de la computación, en el caso del aprendizaje de redes neuronales son simplemente demasiadas variables a considerar y por consiguiente un modelo desarrollado completamente en la teoría es bastante propenso a fallas.

En retrospectiva se debió haber dedicado más atención a la generación de modelos pequeños y pruebas de concepto rápidas dado que su aplicación práctica consta de un *feedback* inmediato respecto a que variables son las importantes en que entorno. Se debió haber dado foco especial a la correcta compresión de todos los aspectos que involucra una red neuronal dado que en muchas ocasiones no se tuvo claridad suficiente de que es lo que realizaba cada

componente.

Se notó además el amplio espectro de representaciones para los datos, ejemplos claros de esto constan las múltiples formas posibles de elaborar grafos respecto al código, como son los árboles de sintaxis abstracta, los vectores semánticos dispuestos como vectores de posición, las representaciones de grafo dirigido para las secuencias de palabras, etc.

Se cuantifica también la importancia de realizar generalizaciones respecto a las anteriores representaciones de los datos, no desviando el foco de la atención de la generación de un buen modelo predictivo por considerar una mayor cantidad de información la cual no posee garantías de otorgar mejoras a este. Un ejemplo claro de esto fue por ejemplo que a pesar de que el modelo de la herramienta GBSCS hace uso de relaciones implícitas en el código fuente que son de otra manera descartadas por la herramienta RENCOS, esta última superó con creces en todo ámbito a los resultados obtenidos por la red neuronal de grafos a pesar de ser un tipo de red con menos dimensionalidad.

Finalmente como síntesis del trabajo realizado en esta memoria se destaca que si bien se logró explorar las capacidades y potencialidades de las redes neuronales de grafos, no fue posible manifestar mediante ejemplos concretos su diferencia ante otros arquetipos de redes, dados los distintos problemas encontrados durante las fases de implementación como ya se dijo anteriormente. Si bien se considera que la metodología de evaluación era la indicada, la metodología de organización de tareas claramente requiere trabajo.

Como propuestas para trabajos futuros se propone realizar una implementación de la herramienta RENCOS que utilice las implementaciones de GGNN de la biblioteca ONMT pero que además sea capaz de construir representaciones tanto semánticas como sintácticas de los grafos para así poder hacer uso de la arquitectura de la herramienta original y por consiguiente enriquecer el modelo predictivo de palabras de baja frecuencia.

Bibliografía

[1] Allamanis, Miltiadis, Marc Brockschmidt y Mahmoud Khademi: *Learning to Represent Programs with Graphs*, 2018.

[2] Cuba Gyllensten, Amaru y Magnus Sahlgren: *Navigating the Semantic Horizon using Relative Neighborhood Graphs*. En *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, páginas 2451–2460, Lisbon, Portugal, Septiembre 2015. Association for Computational Linguistics. `https://www.aclweb.org/anthology/D15-1292`.

[3] Fatima, A., A. Ghazi y C. Luca: *Semantic graph from free-text*. En *2017 International Conference on Optimization of Electrical and Electronic Equipment (OPTIM) 2017 Intl Aegean Conference on Electrical Machines and Power Electronics (ACEMP)*, páginas 1132–1137, 2017.

[4] Ji, Tao, Yuanbin Wu y Man Lan: *Graph-based Dependency Parsing with Graph Neural Networks*. En *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, páginas 2475–2485, Florence, Italy, Julio 2019. Association for Computational Linguistics. `https://www.aclweb.org/anthology/P19-1237`.

[5] Li, Yujia, Daniel Tarlow, Marc Brockschmidt y Richard Zemel: *Gated Graph Sequence Neural Networks*, 2017.

[6] Scarselli, F., M. Gori, A. C. Tsoi, M. Hagenbuchner y G. Monfardini: *The Graph Neural Network Model*. IEEE Transactions on Neural Networks, 20(1):61–80, 2009.

[7] Steidl, D., B. Hummel y E. Juergens: *Quality analysis of source code comments*. En *2013 21st International Conference on Program Comprehension (ICPC)*, páginas 83–92, 2013.

[8] Wang, Wenhan, Ge Li, Bo Ma, Xin Xia y Zhi Jin: *Detecting Code Clones with Graph Neural Networkand Flow-Augmented Abstract Syntax Tree*, 2020.

[9] Wei, Huihui y Ming Li: *Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code*. En *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, páginas 3034–3040, 2017. `https://doi.org/10.24963/ijcai.2017/423`.

[10] Wei, Jiayi, Maruth Goyal, Greg Durrett y Isil Dillig: *LambdaNet: Probabilistic Type Inference using Graph Neural Networks*, 2020.

[11] Zhang, J., X. Wang, H. Zhang, H. Sun y X. Liu: *Retrieval-based Neural Source Code Summarization*. En *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, páginas 1385–1397, 2020.

[12] Zhang, Jian, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang y Xudong Liu: *A Novel Neural Source Code Representation Based on Abstract Syntax Tree*. En *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, página 783–794. IEEE Press, 2019. `https://doi.org/10.1109/ICSE.2019.00086`.

[13] Zhou, Jie, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li y Maosong Sun: *Graph Neural Networks: A Review of Methods and Applications*, 2021.