UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Comportamiento de Graph Convolutional Networks (GCN) ante datos con ruido

María Paz Cuturi Grignola

Facundo Padula Lenna

Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay

Agosto de 2020

UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Comportamiento de Graph Convolutional Networks (GCN) ante datos con ruido

María Paz Cuturi Grignola

Facundo Padula Lenna

Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación.

Director de tesis:
Ph.D. Prof. Pablo Rodríguez Bocca

Montevideo – Uruguay

Agosto de 2020

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

Ph.D. Prof. Adriana Marotta

Ph.D. Prof. Guillermo Moncecchi

Ph.D. Prof. Libertad Tansini

Montevideo – Uruguay

Agosto de 2020

RESUMEN

Los modelos de aprendizaje profundo para grafos han mejorado el estado del arte en muchas tareas. A pesar de su reciente éxito, existen pocas investigaciones acerca de su robustez. En esta tesis se estudia el comportamiento de *Graph Convolutional Networks* (GCNs) ante datos con ruido para la tarea de clasificación de nodos. Se utiliza el cálculo de meta-gradientes para introducir ruido en las aristas y los atributos, esencialmente tratando al grafo como un hiperparámetro a optimizar. Se estudia la cantidad de aristas y atributos que hay que modificar para reducir el *accuracy* de la clasificación en un 5 %, considerando las modificaciones con impacto máximo y mínimo. Nuestros experimentos muestran que el impacto del ruido varía mucho dependiendo de los datos modificados, indicando que no todas las aristas ni todos los atributos inciden de la misma forma en la clasificación de un nodo. En los casos estudiados el impacto al introducir ruido en aristas es mayor que el impacto al introducir ruido en atributos de los nodos. Nuestros resultados pueden servir de guía para estudiar qué determina que una arista o atributo tenga mayor o menor impacto en la clasificación de un nodo, y en general para estimar la robustez de un problema de clasificación frente a ruido en los datos.

Palabras claves:
graph convolutional networks, impacto del ruido, meta-gradientes.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo se introduce la motivación y el contexto de este trabajo, junto a sus objetivos iniciales. Por último se detalla brevemente la organización del documento y el contenido de sus capítulos.

1.1. Motivación

En los últimos años, el aprendizaje profundo ha revolucionado al aprendizaje de máquina, logrando imponerse sobre otras técnicas para resolver problemas muy variados, incluyendo el tratamiento de imágenes, video y audio, procesamiento de lenguaje natural, etc.

Los mejores resultados del aprendizaje profundo se han obtenido para problemas donde los datos son independientes o donde los datos pueden ser representados en espacios euclidianos.

Sin embargo existe una amplia variedad de casos donde los datos están altamente interrelacionados y surgen de espacios no euclidianos. Típicamente en estos problemas los datos se representan con grafos y son resueltos con técnicas particulares desarrolladas en la disciplina que se conoce como "análisis de redes" o "ciencia de redes".

Las tareas más relevantes en esta disciplina son la predicción de enlaces, la predicción de atributos de entidades, y la detección de comunidades. Han sido resueltos exitosamente para instancias que surgen de redes sociales, redes

tecnológicas, datos biológicos, sistemas físicos, etc., donde las relaciones de las entidades con sus vecinos son muy informativas para el problema predictivo, inclusive más que las características de la propia entidad.

Varios estudios han propuesto extensiones al aprendizaje profundo para resolver problemas donde los datos se representan con grafos. Las redes neuronales aplicadas a grafos surgen en 2005 bajo el nombre de *GNN – Graph Neural Networks* [14] y pretenden aprender la representación de un nodo propagando información de sus vecinos de forma iterativa hasta alcanzar un punto fijo estable.

En ocasiones estos modelos han generado mejores resultados que los reportados hasta el momento en *datasets* de referencia, y están siendo aplicados a diversos dominios. Sin embargo, su robustez no ha sido estudiada en profundidad en comparación a otras técnicas de aprendizaje automático.

1.2. Objetivos

El objetivo de este proyecto es estudiar la robustez de las *Graph Convolutional Networks* como un tipo de GNN que extiende el aprendizaje profundo para resolver problemas donde los datos se representan con grafos. En particular nos concentraremos en su robustez al usarse sobre datos con ruido, en la tarea de clasificación de nodos.

Existen investigaciones que estudian el impacto del ruido en diferentes arquitecturas de GNNs pero hasta el momento no se han realizado estudios que analicen el impacto del ruido utilizando específicamente la arquitectura GCN para la clasificación de nodos.

Habitualmente por "ruido" se entiende un error aleatorio espurio. En este trabajo, con un objetivo de generalidad, se utilizará una definición más amplia, que incluye la anterior pero también otro tipo de errores, como sistemáticos al momento de realizar una medición o inclusive ataques intencionados.

En este proyecto abordamos esa problemática, sometiendo a distintos *datasets* a ruido introducido artificialmente. Realizamos perturbaciones en los datos de entrenamiento y testeo, afectando tanto aristas como atributos. Estas distorsiones intentan simular instancias con falta de información o errores en los

datos.

Para alcanzar el objetivo del proyecto se desarrollarán los siguientes objetivos específicos:

- Conocer las técnicas de aprendizaje profundo y comprender las extensiones existentes para aplicarlas a problemas predictivos modelados con grafos.
- Elegir *datasets* emblemáticos de problemas predictivos sobre grafos.
- Elaborar una metodología que permita modificar sus características predictivas fácilmente de forma de evaluar finalmente la robustez de GCN.
- Documentar el conocimiento adquirido, los resultados y conclusiones.

1.3. Resultados alcanzados

Como aportes del proyecto, podemos destacar:

- Una breve revisión bibliográfica acerca de los trabajos existentes sobre el impacto de ruido en GNNs y los ataques adversarios como un antecedente cercano para GCNs.
- Una metodología para introducir ruido de manera paramétrica y controlada en aristas y atributos.
- Un estudio exhaustivo de los resultados, comparando el caso donde el ruido sólo está presente en los atributos, en las aristas o en ambos a la vez.

1.4. Organización del documento

El presente documento está organizado en capítulos. A continuación se describe brevemente cada uno de ellos.

En el Capítulo 2 se presenta el estado del arte. Se incluyen algunas investigaciones que estudian el impacto del ruido en algoritmos automáticos de clasificación y se presenta brevemente el funcionamiento de las *Graph Neural Networks* y en particular de las *Graph Convolutional Networks*. Por último se resumen los trabajos existentes en el estudio del impacto del ruido y en detalle los resultados obtenidos al utilizar ataques adversarios.

El Capítulo 3 detalla la metodología de evaluación desarrollada en este trabajo para estimar el impacto del ruido en la calidad predictiva, incluyendo: la inicialización de valores, la generación del ruido y los criterios de parada en la estimación. Además se presentan los *datasets* sobre los cuales se aplicó esta metodología y algunas de sus características principales.

En el Capítulo 4 se incluyen los principales resultados de aplicar nuestra metodología de introducción de ruido dirigido para cada uno de los casos: ruido en aristas, atributos y en ambos a la vez. Además se presentan algunos resultados de referencia propuestos por otros autores para los *datasets* utilizados, y se presentan resultados obtenidos al introducir ruido de manera aleatoria (técnica que se utilizó a modo exploratorio y como validación de la metodología desarrollada).

El Capítulo 5 resume las principales conclusiones obtenidas a partir del presente proyecto y posibles extensiones a futuro.

Capítulo 2

Estado del arte

Los *datasets* construidos a partir de la realidad (en contraposición con los artificiales) suelen contener ruido [20], entendido como "un error aleatorio o una variación en una variable medida" [17]. Las causas del ruido pueden ser diversas, como ser errores generados por instrumentos que recolectan los datos, errores sistemáticos introducidos por expertos al recabar o procesar los datos, inconsistencias en la definición (convenciones de nomenclatura, códigos, formatos), información incompleta o faltante [17]. Cabe mencionar que, cuando es técnicamente posible, recopilar datos correctos es más costoso y consume más tiempo, lo que hace que exista interés por utilizar *datasets* más grandes pero menos precisos, ya que son más fáciles y económicos de obtener [38] y aplicar técnicas con cierta tolerancia al ruido.

Independientemente del origen del ruido, interesa estudiar su impacto en los diferentes algoritmos para conocer la robustez y confiabilidad de los métodos. Las técnicas de aprendizaje automático son sensibles al ruido, ya que puede afectar la precisión en los modelos generados. "El ruido puede reducir el rendimiento del sistema en cuanto a la precisión de la clasificación, el tiempo de construcción de un clasificador y el tamaño del mismo" [55]. En general los datos ruidosos pueden sesgar el proceso de aprendizaje, haciendo que sea más difícil para los algoritmos formar modelos precisos. Por lo tanto, el desarrollo de técnicas de aprendizaje que aborden de manera eficaz y eficiente este tipo de datos es un aspecto clave en el aprendizaje automático [33].

En las siguientes secciones se analizará esta problemática, partiendo de una

visión general acerca del impacto del ruido en algoritmos automáticos de clasificación. Como se verá más adelante, numerosos trabajos abordan esta temática para datos en espacios euclidianos. Luego se presentará la problemática de clasificación sobre datos en redes, en qué consiste y qué métodos se han utilizado para enfrentarse a ella. Por último, nos enfocaremos en el área de redes neuronales aplicadas a grafos como mecanismo de clasificación de nodos y particularmente en las *Graph Convolutional Networks*. No se han encontrado trabajos sobre el desempeño de GCN sobre datos con ruido, por lo que se presentará un antecedente cercano: ataques adversarios.

2.1. El impacto del ruido en algoritmos automáticos de clasificación

Se han realizado numerosos estudios que analizan el impacto del ruido en los resultados de los algoritmos de aprendizaje automático. En particular, en los problemas de clasificación buscamos identificar clases a partir de atributos. Dado un conjunto de entrenamiento con N pares $(\mathbf{x}_n, t_n)$, con $n = 1, \ldots, N$, donde $\mathbf{x}_n$ son las variables de entrada (o *inputs* o atributos) y t_n las variables de salida (o *outputs*, también llamadas *labels* o clases), el problema consiste en predecir la salida $\mathbf{t}$ para una nueva entrada $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$. La salida $\mathbf{t}$ es un vector de variables discretas que pueden tomar un conjunto finito de valores posibles e indican la clase a la que pertenece cada instancia [40].

La métrica que suele utilizarse para evaluar un modelo de clasificación es el *accuracy*. Se define como el cociente entre la cantidad de predicciones correctas y el número total de predicciones. "Esta medida es multiclase, simétrica, y va desde 0 (clasificación errónea) a 1 (clasificación perfecta). Su popularidad se debe ciertamente a su simplicidad, no sólo en términos de procesamiento pero también de interpretación, ya que corresponde a la proporción observada de casos correctamente clasificados" [24].

En estos problemas de clasificación, la mayoría de los trabajos sobre el impacto del ruido distinguen principalmente dos casos: los atributos y las clases [15]. Brodley et. al. [6] se enfocaron en detectar y remover ruido en las clases. Kubica et. al. [23] estudiaron cómo identificar y eliminar ruido en los atributos, de forma que los casos de entrenamiento restantes puedan ser utilizados para el

modelado. Autores como Zhu et. al. [55] [56] estudian el ruido de acuerdo a qué conjunto de datos afecta: de entrenamiento, de prueba o ambos a la vez. Otros estudios han ahondado en el impacto del ruido en algoritmos puntuales como árboles de decisión, clasificadores bayesianos, *support vector machines*, regresión logística [2], ID3 [37], entre otros.

Los ejemplos expuestos anteriormente se basan en técnicas de aprendizaje automático aplicadas a tareas cuyos datos son tabulares con muestras independientes, lo cual ha sido aplicado exitosamente en la clasificación de imágenes, procesamiento de texto y video. Sin embargo no encontramos antecedentes sobre el estudio del impacto del ruido en problemas de clasificación automática sobre grafos.

2.2. Problemas de clasificación para datos en redes

Redes sociales, estructuras moleculares, redes de interacción entre proteínas, sistemas de recomendación, todos estos dominios y muchos más pueden modelarse fácilmente como grafos que capturan las interacciones (las aristas) entre las unidades individuales (los nodos). Las tareas de aprendizaje automático pueden aplicarse directamente a grafos, por ejemplo para clasificar una proteína en un grafo de interacción biológica, predecir el rol de una persona en una red de colaboración o recomendar nuevos amigos a un usuario en una red social [16]. El aprendizaje sobre grafos pretende resolver tareas como clasificación de nodos, predicción de enlaces y *clustering* [53].

El principal problema de aplicar aprendizaje automático sobre grafos yace en introducir la información estructural en el modelo. Los enfoques tradicionales suelen resolver esto mediante una fase de preprocesamiento que transforma los grafos en una representación más simple, como puede ser un conjunto de vectores. Sin embargo, de esta manera, puede perderse información topológica importante y los resultados obtenidos pueden depender en gran medida de la fase de preprocesamiento.

Métodos como *graph embeddings* aprenden un espacio vectorial continuo para la representación de nodos, aristas o subgrafos, asignando a cada nodo (y/o arista) una posición específica en un espacio vectorial [53]. Un ejemplo de esto

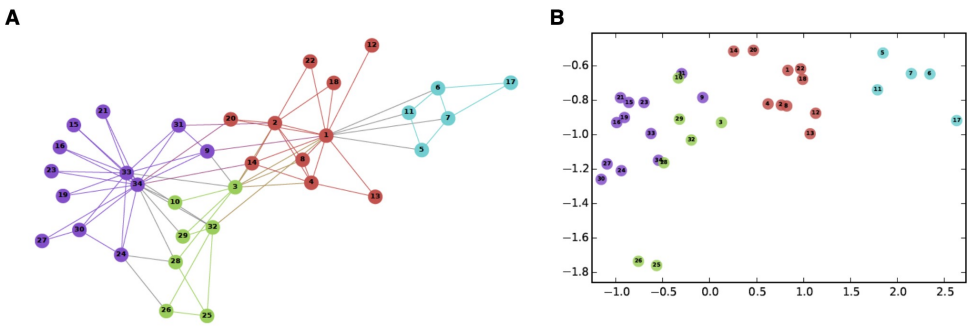

Figura 2.1: Izquierda: Grafo del Club de Karate de Zachary, donde los nodos se conectan si los individuos correspondientes son amigos. Los nodos están coloreados de acuerdo a las diferentes comunidades que existen en la red. **Derecha**: Visualización bidimensional de los *embeddings* de nodos generados a partir de este grafo utilizando el método DeepWalk. De cierta forma se preserva la estructura del grafo, ya que nodos similares están cerca en el espacio. Tomado de Perozzi et. al. [36]

es DeepWalk [36], considerado el primer método de *graph embedding* (ver un ejemplo de aplicación de esta técnica en la Figura 2.1). Se utilizan "caminatas" aleatorias para aprender la estructura y permitir que sea fácilmente explotada por los métodos de clasificación estándar.

Incluso introduciendo esa etapa de preprocesamiento, existe una dificultad adicional al trabajar con grafos. Son estructuras irregulares, ya que el tamaño puede ser variable y cada nodo puede tener una cantidad diferente de vecinos. Además, una de las premisas básicas utilizadas por los algoritmos de aprendizaje automático es que las instancias son independientes entre sí, lo que no ocurre en los grafos porque cada nodo se interrelaciona con otros [52]. "La utilización de grafos plantea desafíos incomparables, en términos de expresividad y complejidad computacional, con respecto al aprendizaje con datos vectoriales" [3].

"A pesar del éxito parcial de algunos de estos métodos de *embedding*, muchos sufren de las limitaciones de los mecanismos de aprendizaje superficiales y podrían no descubrir los patrones más complejos detrás de los grafos" [51].

2.3. Graph Neural Networks

Formalmente un grafo $G = (V, E)$ es una estructura matemática que consiste de un conjunto de vértices V (también llamados "nodos") y un conjunto de

aristas E (también llamado "enlaces"), donde los elementos de E son pares no ordenados $\{u, v\}$ de vértices $u, v \in V$. En este trabajo se consideran grafos no dirigidos.

Los grafos pueden incluir información adicional, tal como un conjunto de pesos en las aristas $\{w_e\},\ e \in E$ indicando por ejemplo la intensidad de la asociación entre los vértices, vectores $\{\mathbf{x}_v\},\ v \in V$ de variables que describen atributos de los vértices y etiquetas [22].

El área de redes neuronales aplicadas a grafos (GNN – *Graph Neural Network*, también conocida como *deep learning* para grafos) tiene una larga y consolidada historia que surge a principios de los noventa con trabajos sobre redes neuronales recursivas (RecNN) en árboles [3].

"La fortaleza de estos métodos yace en el uso de la información relacional de los grafos: en lugar de sólo considerar las instancias individualmente (los nodos y sus atributos), las relaciones entre ellas también se explotan (las aristas). Dicho de otra manera: las instancias no se tratan de forma independiente; utilizamos una cierta forma de datos no i.i.d. donde los efectos de la red como la homofilia apoyan la clasificación" [59].

En aplicaciones de aprendizaje supervisado se busca predecir una salida para un solo nodo, una arista o un grafo completo [3]. Una de las tareas que se aplica con más frecuencia sobre los grafos es la clasificación de los nodos: dado un único grafo (con atributos) y las etiquetas de clase de unos pocos nodos, el objetivo es predecir las etiquetas de los nodos restantes [59]. Además de tareas de inferencia estándar como clasificación de nodos y grafos, los métodos de *deep learning* sobre grafos también se han aplicado a una gran variedad de disciplinas como modelado de influencia social, sistemas de recomendación, química y biología, física, predicción de enfermedades, procesamiento de lenguaje natural, *computer vision* y predicción de tráfico [52]. La Tabla 2.1 resume los tipos de tareas predictivas que se realizan con datos en redes.

Las redes neuronales aplicadas a grafos (GNN) surgen en 2005 [14] y se basan en un sistema de transición de estados similar al de las RecNN (*Recurrent Neural networks*). Puede aplicarse a la mayoría de las topologías de grafos: dirigidos, no dirigidos, etiquetados y cíclicos.

GNN asigna a cada nodo v un vector $\mathbf{x}_v$ llamado estado que recoge una repre-

Tipo	Tarea	
Tareas sobre nodos	Clustering de nodos	
	Clasificación de nodos	Inductiva
		Transductiva
	Reconstrucción de la red	
	Predicción de enlaces	
Tareas sobre grafos	Clasificación de grafos	
	Generación de grafos	Estructura
		Estructura+atributos

Tabla 2.1: Tareas que se suelen utilizar sobre datos en redes, clasificadas según si se aplican sobre nodos o grafos. Tomado de Zhang et. al. [52]

sentación del objeto denotado por el nodo v. Para definir $\mathbf{x}_v$ observamos que los nodos relacionados están conectados por aristas. Así, $\mathbf{x}_v$ se define utilizando la información contenida en el vecindario de v. La Figura 2.2 ejemplifica esta idea calculando el estado de $\mathbf{x}_1$ a partir de la información del propio nodo y la de su vecindario. El estado $\mathbf{x}_v$ es un vector que puede ser utilizado para producir una salida $\mathbf{o}_v$, tal como la etiqueta del nodo. Definimos $\mathbf{w}$ como un conjunto de parámetros y $f_\mathbf{w}$ como una función de transición paramétrica que expresa la dependencia de un nodo con sus vecinos. Sea g la función de salida local que describe cómo se produce la misma. Entonces $\mathbf{x}_v$ y $\mathbf{o}_v$ quedan definidas de la siguiente manera:

$$\mathbf{x}_v = f_\mathbf{w}(\mathbf{l}_v, \mathbf{x}_{\mathbf{ne}[v]}, \mathbf{l}_{\mathbf{ne}[v]}), v \in V \tag{2.1}$$

$$\mathbf{o}_v = g_\mathbf{w}(\mathbf{x}_v, \mathbf{l}_v), v \in V \tag{2.2}$$

donde $\mathbf{l}_v$ es la etiqueta de v y $\mathbf{x}_{\mathbf{ne}[v]}$ y $\mathbf{l}_{\mathbf{ne}[v]}$ son los estados y las etiquetas de los vecinos de v respectivamente [14].

Sean $\mathbf{x}$ y $\mathbf{l}$ los vectores que agrupan todos los estados y todas las etiquetas respectivamente. Entonces las ecuaciones (2.1) y (2.2) pueden reescribirse como:

$$\begin{aligned} \mathbf{x} &= F_\mathbf{w}(\mathbf{x}, \mathbf{l}) \\ \mathbf{o} &= G_\mathbf{w}(\mathbf{x}, \mathbf{l}) \end{aligned} \tag{2.3}$$

donde $F_\mathbf{w}$ y $G_\mathbf{w}$ son la composición de $|V|$ instancias de $f_\mathbf{w}$ y $g_\mathbf{w}$ respectivamente.

Aplicando el teorema del punto fijo de Banach, GNN utiliza el siguiente es-

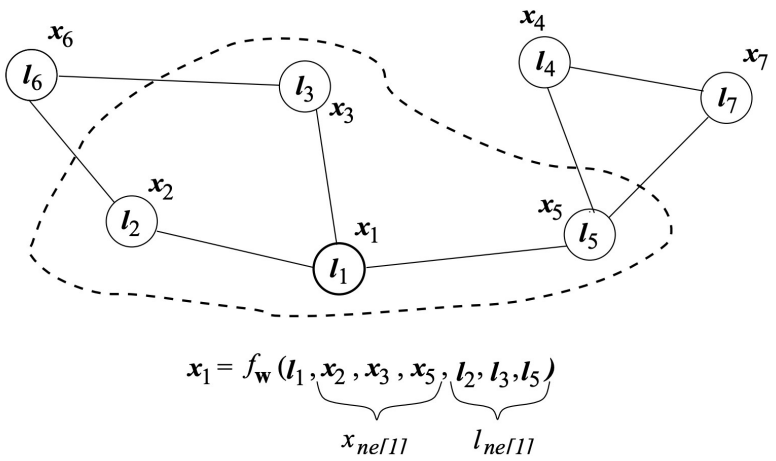

Figura 2.2: El estado $\mathbf{x}_1$ depende de la información de su vecindario. Tomado de Gori et. al. [14].

quema iterativo clásico para computar el estado:

$$\mathbf{x}(t+1) = F_{\mathbf{w}}(\mathbf{x}(t), \mathbf{l}) \tag{2.4}$$

donde $\mathbf{x}(t)$ denota la iteración t de $\mathbf{x}$ y converge exponencialmente a la solución de (2.3) para cualquier estado inicial $\mathbf{x}(0)$. Entonces $\mathbf{x}_v$ y $\mathbf{o}_v$ pueden calcularse de la siguiente manera:

$$\begin{aligned} \mathbf{x}_v(t+1) &= f_{\mathbf{w}}(\mathbf{l}_v, \mathbf{x}_{\mathbf{ne}[v]}(t), \mathbf{l}_{\mathbf{ne}[v]}), v \in V \\ \mathbf{o}_v(t+1) &= g_{\mathbf{w}}(\mathbf{x}_v(t+1), \mathbf{l}_v), v \in V \end{aligned} \tag{2.5}$$

Las ecuaciones (2.5) pueden interpretarse como la representación de una red neuronal llamada *encoding network* que consiste de neuronas que computan $f_{\mathbf{w}}$ y $g_{\mathbf{w}}$. Para construir la red, cada nodo del grafo puede ser reemplazado por una neurona que computa la función $f_{\mathbf{w}}$. Cada neurona almacena el estado actual $\mathbf{x}_v(t)$ del nodo v y cuando es activada, calcula el estado $\mathbf{x}_v(t+1)$ usando las etiquetas y los estados almacenados en sus vecinos. La activación simultánea y repetida de las neuronas produce el comportamiento descrito por la ecuación (2.5). En la red, la salida del nodo v es producida por otra neurona que implementa $g_{\mathbf{w}}$ [14]. En la Figura 2.3 se muestra un grafo con cuatro nodos y su *encoding network* correspondiente.

El algoritmo de aprendizaje se basa en la estrategia de descenso por gradiente

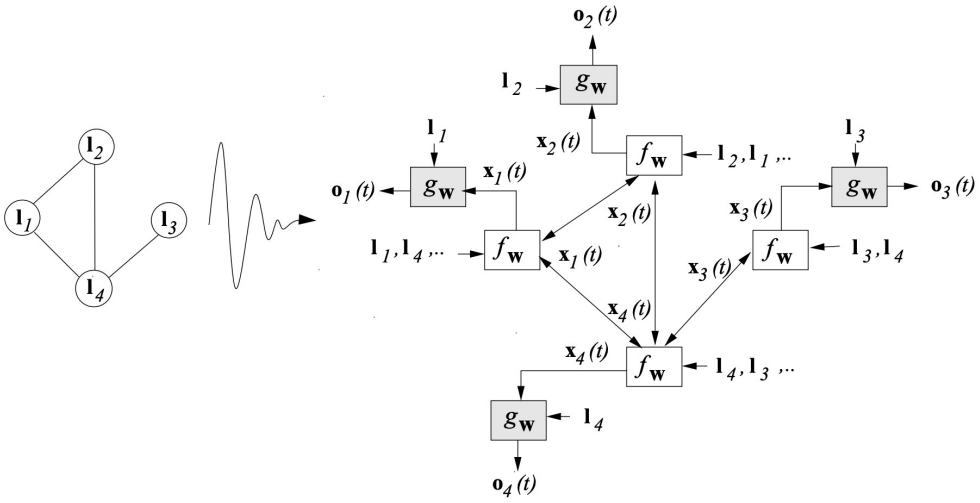

Figura 2.3: Un grafo y su representación como red neuronal (*encoding network*) que computan $f_\mathbf{w}$ y $g_\mathbf{w}$. Tomado de Gori et. al. [14].

y consiste de los siguientes pasos:

1. Los estados $\mathbf{x}_n(t)$ se actualizan iterativamente utilizando la ec. (2.5) hasta que alcanzan un punto estable $\mathbf{x}(T) = \mathbf{x}$ en el tiempo T.
2. El gradiente de los pesos $\mathbf{w}$ es computado a partir de la pérdida.
3. Los pesos $\mathbf{w}$ son actualizados de acuerdo al gradiente calculado en el paso anterior.

En resumen, GNN aprende la representación de un nodo objetivo propagando la información de sus vecinos de manera iterativa hasta que se alcanza un punto fijo estable. Este proceso es computacionalmente costoso, y recientemente ha habido esfuerzos crecientes para superar estos desafíos [48].

2.4. Graph Convolutional Networks

Alentados por el éxito de las *Convolutional Neural Networks* (CNN) en el ámbito de *computer vision*, se desarrollaron en paralelo un gran número de métodos que redefinen la noción de convolución para los grafos [48].

Las *Graph Convolutional Network* (GCN) son consideradas un tipo particular de GNN que usan una agregación convolutiva. También existen otras variantes de GNN basadas en distintos tipos de agregación como *Gated Graph Neural Networks* y *Graph Attention Networks* [53]. "En términos generales, los mo-

delos de *Graph Convolutional Networks* son un tipo de arquitecturas de redes neuronales que pueden aprovechar la estructura del grafo y agregar información de los nodos vecinos de manera convolutiva" [51].

Su nombre se debe a que "toman prestados" conceptos de las CNN. La clave consiste en aprender una función f para generar la representación de un nodo v agregando sus propios atributos $\mathbf{x}_v$ y los atributos de sus vecinos u, donde $u \in N(v)$.

El interés en las GCN se debe principalmente a dos motivos: la cantidad creciente de datos no euclidianos en aplicaciones actuales y el rendimiento limitado que alcanzan las CNN al trabajar sobre tales datos [26]. "*Graph Convolutional Networks* son indiscutiblemente el tema más popular en *deep learning* basado en grafos" [52].

Existen múltiples ejemplos de aplicación de GCN. En el área de procesamiento de lenguaje natural se han utilizado para la clasificación de textos. Al considerar a los documentos como nodos y las citas como aristas entre ellos, se construye una red de citación, donde los atributos de los nodos se modelan mediante *bag of words*. La clase de cada texto es entonces la etiqueta de su nodo. Otros ámbitos de aplicación son la detección de *fake news*, predicción de influencias sociales, clasificación de imágenes y reconocimiento de acciones en videos [51].

Usando la estructura del grafo y la información de los nodos como entrada, las salidas de GCN pueden centrarse en diferentes tareas de análisis de acuerdo a uno de los siguientes mecanismos [48]:

- Las salidas a nivel de nodo se relacionan con las tareas de regresión y clasificación de nodos.
- Las salidas a nivel de arista se relacionan con las tareas de clasificación y predicción de aristas.
- Las salidas a nivel de grafo se relacionan con la tarea de clasificación de grafos.

En particular, dada una red con solo algunos nodos etiquetados, $V_L \subseteq V$, el problema de clasificación de nodos consiste en predecir la clase (etiqueta) de un nodo que no está etiquetado, $v \in V_{\neg L}$ con $V_{\neg L} = V - V_L$, utilizando aquellos que sí lo están. Seguidamente se explicará la arquitectura de una GCN para dicha

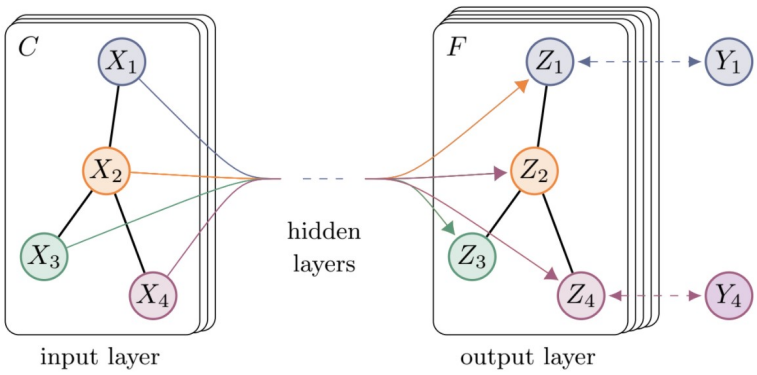

Figura 2.4: Representación esquemática de una GCN de varias capas para el aprendizaje semi-supervisado con C canales de entrada y F mapeos de *features* en la capa de salida. La estructura del grafo (las aristas mostradas como líneas negras) se comparte entre las capas. Las etiquetas se denotan con Y_i. Tomado de Kipf y Welling [21].

tarea. Sea $G = (V, E)$ un grafo e Y el conjunto de clases posibles. Formalmente esto se traduce en aprender una función $f : V \to Y$ tal que asigne una clase $y \in Y$ a cada nodo $v \in V$. Una GCN de este estilo toma como entrada:

- una matriz de atributos $\mathbf{X} \in \mathbb{R}^{|V| \times C}$, donde $|V|$ es el número de nodos y C es la cantidad de atributos de los nodos (o *input channels*).
- una matriz $\mathbf{A} \in \{0, 1\}^{|V| \times |V|}$ representando la estructura del grafo (matriz de adyacencia).

y genera como salida una matriz $\mathbf{Z} \in \mathbb{R}^{|V| \times F}$, donde F es la cantidad de mapas de activación (o *feature maps*).

El modelo combina la estructura del grafo y los atributos de los vértices en la convolución, donde los atributos de los vértices no etiquetados se combinan con aquellos de los vértices cercanos y son propagados por el grafo a través de múltiples capas [27]. "Dependiendo del número de capas de convolución que se utilice, las GCN pueden capturar información sólo sobre los vecinos inmediatos (con una capa de convolución) o cualquier nodo a lo sumo a K saltos de distancia (si K capas están apiladas una encima de la otra)" [28]. La Figura 2.4 muestra una representación completa de una GCN con múltiples capas. La Figura 2.5 muestra una posible arquitectura de GCN para clasificación de nodos.

Análogo al funcionamiento convolutivo de una CNN en una imagen, las convoluciones sobre grafos se definen en base a las relaciones espaciales de un nodo.

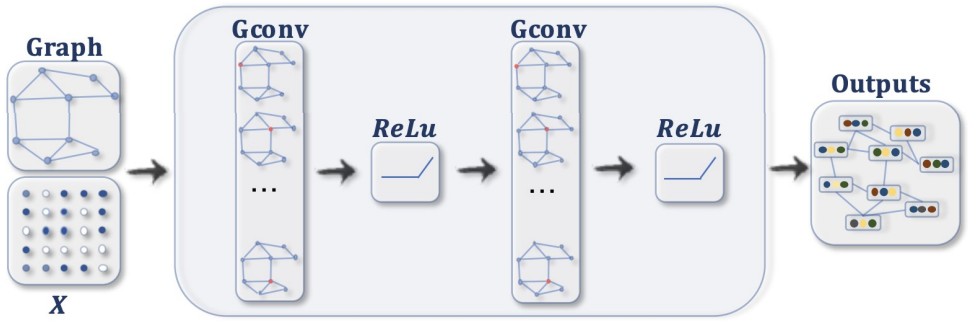

Figura 2.5: Una variante de *Graph Convolutional Networks* con múltiples capas para clasificación de nodos. Una capa GCN encapsula la representación oculta de cada nodo agregando información de atributos de sus vecinos. Después de la agregación de atributos, se aplica una transformación no lineal a las salidas resultantes. Al apilar varias capas, la representación final de cada nodo recibe mensajes de un vecindario más lejano. Tomado de Wu et. al. [48].

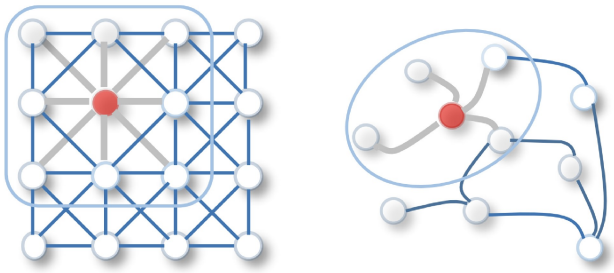

Figura 2.6: Izquierda: Convolución 2D. Análogo a un grafo, cada píxel de una imagen se toma como un nodo donde los vecinos se determinan por el tamaño del filtro (en este caso 3×3). La convolución 2D toma el promedio ponderado de los valores de los píxeles del nodo rojo junto con sus vecinos.
Derecha: Convolución en un grafo. Para obtener una representación del nodo rojo, una solución simple es tomar el valor promedio de los *features* del nodo rojo junto con los de sus vecinos. Tomado de Wu et. al. [48].

Como ilustra la Figura 2.6, las imágenes pueden ser pensadas como un grafo donde cada píxel representa un nodo. Cada píxel está directamente conectado a sus píxeles cercanos.

Cada capa, $\mathbf{H}^{(l)}$, de la red neuronal puede escribirse como:

$$\mathbf{H}^{(0)} = \mathbf{X}, \tag{2.6}$$

$$\mathbf{H}^{(l)} = f(\mathbf{H}^{(l-1)}, \mathbf{A}) \qquad \forall l \in \{1, \ldots, L-1\} \tag{2.7}$$

$$\mathbf{H}^{(L)} = \mathbf{Z} \tag{2.8}$$

siendo L la cantidad de capas. La matriz $\mathbf{H}^{(l)} \in \mathbb{R}^{|V| \times M_l}$ contiene en cada fila una representación de los atributos de un nodo, siendo M_l la dimensión de atributos de cada vértice en la capa l. Los distintos modelos solo difieren en la elección de la función de propagación f.

A modo de ejemplo, supongamos que partimos de la siguiente función de propagación $f(\mathbf{H}^{(l)}, \mathbf{A}) = \sigma(\mathbf{A}\mathbf{H}^{(l)}\mathbf{W}^{(l)})$ donde $\mathbf{W}^{(l)}$ es una matriz de pesos para la capa l de la red neuronal y σ es una función de activación no lineal como $ReLU(\cdot) = max(0, \cdot)$, que básicamente asigna 0 a los valores negativos durante la propagación.

Este modelo cuenta con dos limitaciones:

- Multiplicar por $\mathbf{A}$ significa que para cada nodo estamos sumando los vectores de atributos de sus vecinos pero no del propio nodo. Podemos solucionar esto imponiendo bucles en el grafo sumando la matriz identidad a $\mathbf{A}$.
- Típicamente $\mathbf{A}$ no estará normalizada, por lo que multiplicar por $\mathbf{A}$ cambiará la escala de los vectores de atributos. Para solucionar este problema debemos normalizar $\mathbf{A}$ para que todas sus filas sumen 1, es decir usar $\mathbf{D}^{-1}\mathbf{A}$ donde $\mathbf{D}$ es la matriz diagonal con el grado de cada nodo. Multiplicar por $\mathbf{D}^{-1}$ corresponde a obtener el promedio de los atributos de los nodos vecinos. En la práctica se ha demostrado que es mejor utilizar la normalización simétrica $\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ [21].

Finalmente obtenemos la función de propagación:

$$f(\mathbf{H}^{(l)}, \mathbf{A}) = \sigma(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}) \tag{2.9}$$

donde $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ ($\mathbf{I}$ es la matriz identidad) y $\tilde{\mathbf{D}}$ es la matriz diagonal con el grado de los nodos de $\tilde{\mathbf{A}}$.

Consideremos el ejemplo de utilizar una GCN de dos capas para la tarea semi-supervisada de clasificar nodos de un grafo, cuya matriz de adyacencia es $\mathbf{A}$. Se aplica la función de activación ReLU en la primera capa y la capa de salida es pasada por una función de activación softmax de forma de obtener vectores que representen las probabilidades de pertenencia de cada nodo a la clase. El primer paso consiste en calcular $\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$. El modelo completo,

considerando las dos capas, entonces es:

$$\mathbf{Z} = f(\mathbf{X}, \mathbf{A}) = softmax(\hat{\mathbf{A}} \cdot ReLU(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)}) \cdot \mathbf{W}^{(1)}) \tag{2.10}$$

donde $\mathbf{W}^{(0)} \in \mathbb{R}^{C \times H}$ es una matriz de pesos *input-to-hidden* y $\mathbf{W}^{(1)} \in \mathbb{R}^{H \times F}$ es una matriz de pesos *hidden-to-output*, siendo H la cantidad de mapas de activación (o *feature maps*) de la capa oculta. La función de activación $softmax$ se define como $softmax(x_i) = \frac{1}{Z} exp(x_i)$ con $Z = \sum_i exp(x_i)$ y es aplicada por filas.

Los pesos de la red neuronal $\mathbf{W}^{(0)}$ y $\mathbf{W}^{(1)}$ se entrenan utilizando descenso por gradiente [21]. Los parámetros óptimos $\theta = [\mathbf{W}^{(0)}, \mathbf{W}^{(1)}]$ se aprenden de manera semi-supervisada minimizando la entropía cruzada en la salida de los ejemplos etiquetados V_L, es decir, minimizando

$$\mathcal{L}(\theta; \mathbf{A}, \mathbf{X}) = -\sum_{v \in V_L} ln\mathbf{Z}_{v,c_v}, \tag{2.11}$$

donde $\mathbf{Z} = f_\theta(\mathbf{A}, \mathbf{X})$, c_v es la etiqueta de v del conjunto de entrenamiento. Después del entrenamiento, $\mathbf{Z}$ denota las probabilidades de clase para cada instancia del grafo [59].

2.5. GCN y ruido

Anteriormente se mencionó que el ruido podía presentarse en los atributos y en las clases. Al trabajar con grafos, existe una componente adicional que puede introducir ruido: la definición de aristas. A pesar del interés por las redes neuronales sobre grafos, existen pocos estudios acerca de su robustez.

"Por un lado, los efectos relacionales podrían mejorar la robustez ya que las predicciones no se basan sólo en casos individuales sino en varias instancias conjuntamente. Por otro, la propagación de la información también puede dar lugar a efectos en cascada, en los que la manipulación de un solo caso afecta a muchos otros" [59].

En *clustering* sobre grafos, Chen et. al. [7] midieron los cambios en el resultado al inyectar ruido a un grafo bipartito que representa consultas de DNS. Bojchevski et. al. [5] consideran la existencia de ruido en la estructura del grafo

para mejorar la robustez al realizar *spectral clustering*.

Existen estudios sobre el impacto del ruido en GNNs. Fox et. al. [12] estudiaron el impacto en la clasificación de nodos añadiendo aristas aleatoriamente. Sus resultados muestran que el rendimiento de GNN puede verse drásticamente disminuido. Al añadir un 25 % de ruido respecto a la cantidad de aristas en el grafo original, el *F1-score* decrece hasta un 50 %. Los autores estudiaron esto sobre una variación particular de GNN llamada *Graph Isomorphism Network* (GIN).

NT et. al. [34] entrenaron dos modelos GNN (GIN y GraphSAGE) con datos con ruido introducido artificialmente en las etiquetas (clases) y los testearon con datos no corrompidos.

Las investigaciones mencionadas anteriormente estudian el impacto del ruido en diferentes arquitecturas de GNNs. Hasta el momento no se han realizado estudios que analicen el impacto del ruido utilizando específicamente la arquitectura GCN para la clasificación de nodos. En este proyecto abordamos esa problemática, sometiendo distintos *datasets* a ruido introducido artificialmente. Realizamos perturbaciones en los datos de entrenamiento y testeo, afectando tanto aristas como atributos. Estas distorsiones intentan simular instancias con falta de información o errores en los datos.

Un área de estudio relacionada, muy bien explorada ha sido evaluar el impacto de ataques deliberados en los datos. A continuación, profundizaremos sobre estos trabajos porque serán de relevancia para la solución de nuestro problema.

2.5.1. Adversarial Attacks en GNN

Existen estudios sobre la robustez de GNN en el contexto de ataques intencionados. Los llamados *adversarial attacks* consisten en perturbar los datos introduciendo ruido diseñado especialmente para generar una clasificación errónea. En la Tabla 2.2 se incluyen algunos trabajos sobre ataques adversarios en la tarea de clasificación de nodos y sus referencias. Dentro de los ataques adversarios, se distinguen los *poisoning attacks* que inyectan o modifican datos de entrenamiento o etiquetas para generar un modelo erróneo, y los *evasion attacks* que modifican los datos de prueba para fomentar la clasificación errónea de un modelo entrenado [50].

Ref.	Año	Modelo	Ataque	Métrica	*Dataset*
[11]	2019	GCN	N/A	Accuracy	Cora, NELL, Citeseer
[9]	2019	DeepWalk	N/A	Accuracy, AUC	Cora, Wiki Citeseer, CA-GrQc, CA-HepTh
[42]	2019	GCN	N/A	Accuracy	Cora, Citeseer, Pubmed
[54]	2019	GCN	Nettack, RL-S2V, Random	Accuracy	Cora, Citeseer, Pubmed
[45]	2019	GCN	Nettack, RL-S2V, Random	Classification margin	Cora, Citeseer, PolBlogs
[18]	2019	GCN	N/A	Accuracy, Robustness merit, Attack deterioration	Cora, Pubmed, Citeseer
[47]	2019	GCN	Random, Nettack FGSM, JSMA	Accuracy, Classification margin	Cora, Citeseer, PolBlogs
[49]	2019	GCN	DICE, Meta-self	Accuracy, Misclassification rate	Cora, Citeseer
[10]	2019	GCN	N/A	Accuracy	Citeseer, Cora, Pubmed, NELL
[58]	2019	GCN, GNN	N/A	Accuracy, Average worst-case margin	Cora-ML, Pubmed, Citeseer

Tabla 2.2: Tabla resumen sobre ataques adversarios en la tarea de clasificación de nodos extraída de [43].

"La principal dificultad técnica para generar un *poinsoning attack* es computar los casos envenenados, también llamados *adversarial training examples*. Para eso es necesario resolver un problema de optimización de dos niveles, en el que la optimización externa equivale a maximizar el error de clasificación en un conjunto de validación no contaminado, mientras que la optimización interna corresponde al entrenamiento del algoritmo de aprendizaje sobre los datos envenenados" [32]. La formulación matemática es la siguiente [57]:

$$\max_{\hat{G}} \mathcal{L}_{test}(f_{\theta^*}(\hat{G})) = \min_{\hat{G}} \mathcal{L}_{atk}(f_{\theta^*}(\hat{G})) \quad \text{sujeto a} \quad \theta^* = \arg\min_{\theta} \mathcal{L}_{train}(f_{\theta}(\hat{G})) \tag{2.12}$$

donde f_θ es la GNN con parámetros θ, $\mathcal{L}_{atk}$ es la función de pérdida que el atacante intenta optimizar, $\mathcal{L}_{train}$ es la función de pérdida respecto a los datos de entrenamiento y $\mathcal{L}_{test}$ respecto a los datos de prueba. El atacante busca generar la peor clasificación (aquella que maximiza la función de pérdida respecto a los datos de test) luego de optimizar los parámetros del modelo sobre el grafo $\hat{G}$ modificado (atacado). La optimización sobre G se llama problema *upper-level* y la optimización sobre el modelo θ dado G se llama problema *lower-level* [30].

Zügner et. al. [59] realizaron el primer estudio sobre la tolerancia de las GCN a *adversarial attacks*. La optimización propuesta por los autores busca encontrar los mejores cambios en la estructura de un grafo y los atributos de sus nodos para que el nodo objetivo v_0 sea clasificado erróneamente. Los ataques deben ser "impercetibles", es decir, deben hacer pequeños cambios. Concretamente, los autores propusieron preservar características de los datos, como las distribuciones de grado de los nodos y las co-ocurrencias de los atributos. Se estudiaron dos escenarios de ataque, el ataque directo (que incide directamente en el nodo objetivo v_0) y el ataque por influencia (que sólo ataca a otros nodos) [52].

Dai et. al. [8] estudiaron ataques sobre grafos, enfocándose solo en cambios en la estructura del grafo. En lugar de suponer que el atacante poseía toda la información, los autores consideraron varios escenarios en los que se disponía de diferentes cantidades de información. Los resultados experimentales mostraron que los ataques fueron eficaces para las tareas de clasificación de nodos y grafos [52].

Los dos trabajos mencionados anteriormente son dirigidos, es decir, buscan clasificar erróneamente un nodo objetivo v_0. Los autores de [57] fueron los primeros en estudiar los ataques no dirigidos, que tenían por objeto reducir el rendimiento general del modelo. Los autores invierten el procedimiento de optimización basado en gradientes de los modelos de *deep learning* y tratan los datos de entrada (el grafo) como un hiperparámetro a ser aprendido. Buscan resolver el problema de la ecuación (2.12) utilizando meta-gradientes. La Figura 2.7 indica esquemáticamente las dos optimizaciones realizadas por los autores para resolver el problema. "Los meta-gradientes (gradientes con respecto a hiperparámetros) se obtienen aplicando *backpropagation* durante la etapa de aprendizaje de un modelo diferenciable (típicamente una red neuronal)" [57].

La idea principal detrás de su algoritmo consiste en pensar la estructura del grafo (más específicamente, su matriz de adyacencia) como un hiperparámetro y computar el gradiente de la función de pérdida del atacante con respecto al mismo luego del entrenamiento:

$$\nabla_G^{meta} := \nabla_G \mathcal{L}_{atk}(f_{\theta^*}(G)) \quad \text{sujeto a} \quad \theta^* = \text{opt}_\theta \mathcal{L}_{train}(f_\theta(G))$$

donde $opt(\cdot)$ es un procedimiento de optimización diferenciable (por ejemplo, descenso por gradiente estocástico) y $\mathcal{L}_{train}$ es la función de pérdida durante el entrenamiento.

El atacante utiliza el meta-gradiente calculado para realizar una meta actualización M sobre los datos para minimizar $\mathcal{L}_{atk}$.

$$G^{(k+1)} \leftarrow M(G^{(k)})$$

Estas ideas de utilizar meta-gradientes para optimizar el ataque serán adaptadas y generalizadas al contexto de nuestro trabajo, donde hay datos con ruido, en el Capítulo 3.

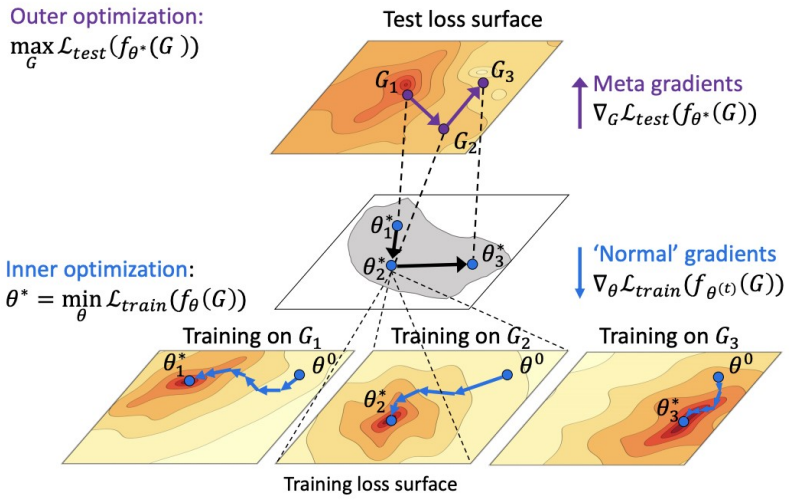

Figura 2.7: Visión esquemática de la idea aplicada por Zügner et. al. [57].

Capítulo 3

Metodología

En este capítulo explicamos el método y los materiales del proyecto. Comenzamos por el método de evaluación para estimar el impacto del ruido en la calidad predictiva, incluyendo: la inicialización de valores, la generación del ruido y los criterios de parada en la estimación. Respecto a los materiales, se describen los conjuntos de datos, *datasets*, que utilizamos para evaluar al método.

3.1. Introducción

La pregunta de investigación detrás del presente trabajo es "¿Cómo se comporta GCN ante datos con ruido?".

Consideramos la tarea de clasificar nodos de un grafo con atributos binarios. Formalmente, sea $G = (\mathbf{A}, \mathbf{X})$ un grafo cuyos nodos V tienen atributos, donde $\mathbf{A} \in \{0,1\}^{|V|\times|V|}$ es la matriz de adyacencia que representa las aristas y $\mathbf{X} \in \{0,1\}^{|V|\times D}$ representa los atributos de los nodos. Denotamos $\mathbf{x}_v \in \{0,1\}$ el vector de atributos de D dimensiones del nodo $v \in V$.

Dado un subconjunto $V_L \subseteq V$ de nodos etiquetados, cuyas clases son $C = \{c_1, c_2, \ldots, c_K\}$, el objetivo es aprender una función $f_\theta : V \to C$ que le asigna a cada nodo $v \in V$ una de las K clases en C.

Esta es una instancia de aprendizaje semi-supervisado, ya que en la etapa de entrenamiento se utiliza una pequeña cantidad de datos etiquetados y una

gran cantidad de datos no etiquetados. “En el aprendizaje semi-supervisado se utilizan datos no etiquetados (es decir, instancias de datos con sólo atributos) junto con los datos etiquetados, en un esfuerzo por mejorar la precisión de los modelos en los datos de entrenamiento, así como por proporcionar un mejor rendimiento en la generalización de datos no vistos” [4].

Además es una instancia de aprendizaje transductivo, ya que todos los casos de test (los nodos sin etiquetas) así como sus atributos y aristas (pero no sus clases) son conocidos y utilizados durante el entrenamiento [57].

Los parámetros θ de la función f_θ se aprenden minimizando una función de pérdida $\mathcal{L}_{train}$ sobre los nodos de entrenamiento etiquetados:

$$\theta = \arg\min_{\theta} \mathcal{L}_{train}(f_\theta(G))$$

Se propone introducir ruido o perturbaciones en los datos (ya sea en las aristas, en los atributos o en ambos) para determinar de qué manera afecta esto a la precisión del modelo construido utilizando GCN.

A diferencia de los *poisoning attacks* donde las distorsiones introducidas artificialmente sólo afectan a los datos de entrenamiento, o a los *evasion attacks* que sólo modifican los datos de prueba, el ruido puede presentarse en cualquier conjunto de datos. Para simular esto, se introducen distorsiones en los datos previo a la etapa de entrenamiento (y consecuentemente de evaluación) del modelo. Dichas distorsiones pueden afectar el conjunto de entrenamiento, el conjunto de prueba o ambos a la vez.

Otra diferencia importante es que los ataques buscan la modificación más pequeña posible que genere una clasificación errónea. Por ejemplo, Zang et. al. [50] aplican un algoritmo llamado IMP (*Iterative Minimum Perturbation*) que traspasa lo que ellos denominan como “frontera de decisión” y clasifica al nodo objetivo i erróneamente. Sin embargo el ruido no tiene esa mala intención y es muy posible que aunque suponga una distorsión grande, no genere un gran impacto.

3.1.1. Evaluación aleatoria

Inicialmente consideramos la opción de introducir ruido de forma aleatoria en las aristas y atributos. Definimos una variable denominada *distortion rate* que establece cuánto ruido se introduce en los datos, definiendo qué porcentaje de los datos (aristas y atributos) se distorsiona, tomando posibles valores entre $0,1$ y $0,9$. La distorsión aplicada es completamente aleatoria. En el caso de los atributos, se opta por asignar 0 en entradas aleatorias de la matriz de atributos (la cantidad correspondiente al *distortion rate* seleccionado). Se aplica algo similar para el caso de aristas, donde se sortean dos matrices aleatorias simétricas: una máscara y una nueva matriz de adyacencia. La nueva matriz de adyacencia se genera de la siguiente manera: en las entradas donde la máscara tiene unos (cantidad correspondiente al *distortion rate*), se sustituye el valor de la matriz original por el de la matriz generada aleatoriamente.

Esta primera aproximación resulta útil para validar la problemática del ruido, pero es poco precisa. Hay que considerar que tal como se define el experimento, la distorsión es a lo sumo tanta como el *distortion rate* pero no es posible asegurar la cantidad exacta de ruido introducido. Por ejemplo, en el caso de los atributos, la matriz suele ser muy dispersa (con un gran número de ceros). Es posible que las entradas a las que se les asigna el valor 0 ya tengan ese valor originalmente, en cuyo caso no se introducirían modificaciones. Por el contrario, es posible que las entradas seleccionadas aleatoriamente sean las pocas cuyo valor original no es 0, por lo que los cambios modifican drásticamente la matriz de atributos original. Lo mismo ocurre con la distorsión de aristas, ya que es posible que la matriz generada aleatoriamente tenga los mismos valores (o similares) a los de la matriz original.

El problema con este enfoque es que sólo se puede ajustar el porcentaje de ruido introducido (cuántas aristas y atributos se afectan), pero no hay forma de parametrizar el ruido introducido de acuerdo a su impacto de forma que sea gradual ni de asegurar que dos distorsiones con el mismo *distortion rate* afectan de manera similar los datos. Hacer esta exploración aleatoria no es concluyente, ya que el espacio de búsqueda es demasiado grande como para poder sacar conclusiones. El crecimiento exponencial del tamaño del espacio de búsqueda en relación a las variables y sus dominios hace que se vuelva imposible aplicar completamente este método, por lo que se opta por utilizar

otra estrategia.

3.1.2. Evaluación dirigida

Como alternativa a la aleatoriedad para introducir ruido, se utiliza la idea de Zügner et. al. [57]: pensar la estructura del grafo como un hiperparámetro y computar el gradiente de la función de pérdida del atacante con respecto al mismo luego del entrenamiento. Si bien ellos lo hacen para simular un ataque, es posible aplicar la misma idea para determinar qué aristas y/o atributos distorsionar para introducir ruido. En ese caso $\mathcal{L}_{atk} = -\mathcal{L}_{train}$ ya que a diferencia de los ataques donde no se disponen de las etiquetas de los nodos de test, en este caso sí:

$$\nabla_G^{meta} := \nabla_G \mathcal{L}_{test}(f_{\theta^*}(G)) \quad \text{sujeto a} \quad \theta^* = \text{opt}_\theta \mathcal{L}_{train}(f_\theta(G))$$

donde $opt(\cdot)$ es un procedimiento de optimización diferenciable (por ejemplo, descenso por gradiente estocástico) y $\mathcal{L}_{train}$ es la función de pérdida durante el entrenamiento.

Si instanciamos *opt* como descenso por gradiente con tasa de aprendizaje α y θ_0 son los pesos iniciales inicializados aleatoriamente, tenemos:

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta_t} \mathcal{L}_{train}(f_{\theta_t}(G))$$

Entonces la función de pérdida luego de entrenar por T iteraciones es $\mathcal{L}_{test}(f_{\theta_T}(G))$ y el cálculo del meta-gradiente se define como la derivada de la función de pérdida con respecto a G:

$$\begin{aligned} \nabla_G^{meta} &= \nabla_G \mathcal{L}_{test}(f_{\theta_T}(G)) \\ &= \frac{\partial \mathcal{L}_{test}(f_{\theta_T}(G))}{\partial G} \end{aligned}$$

Se aplica la regla de la cadena: primero se obtiene la derivada de la función de pérdida con respecto a la red neuronal f, y luego la derivada de f con respecto a G:

$$\begin{aligned} \frac{\partial \mathcal{L}_{test}(f_{\theta_T}(G))}{\partial G} &= \frac{\partial \mathcal{L}_{test}(f_{\theta_T}(G))}{\partial f} \cdot \frac{\partial f_{\theta_T}(G)}{\partial G} \\ &= \nabla f \mathcal{L}_{test}(f_{\theta_T}(G)) \cdot \frac{\partial f_{\theta_T}(G)}{\partial G} \end{aligned} \tag{3.1}$$

Para desarrollar el último término de la ecuación anterior hay que considerar que f depende directamente de G (ya que se utiliza para realizar las predicciones), pero θ_T también depende de G. Podemos pensar en f como una función multivariable $f(\theta_T(G), G)$ y aplicar la regla de la cadena para ese caso:

$$\begin{aligned}\frac{\partial f_{\theta_T}(G)}{\partial G} &= \frac{\partial f(G, \theta_T(G))}{\partial G} \\ &= \frac{\partial f_{\theta_T}(G)}{\partial G} \cdot \frac{\partial G}{\partial G} + \frac{\partial f_{\theta_T}(G)}{\partial \theta_T} \cdot \frac{\partial \theta_T(G)}{\partial G} \\ &= \nabla_G f_{\theta_T}(G) + \nabla_{\theta_T} f_{\theta_T}(G) \cdot \nabla_G \theta_T\end{aligned} \tag{3.2}$$

Combinando las ecuaciones (3.1) y (3.2) obtenemos la fórmula para calcular los meta-gradientes:

$$\nabla_G^{meta} = \nabla_f \mathcal{L}_{test}(f_{\theta_T}(G)) \cdot [\nabla_G f_{\theta_T}(G) + \nabla_{\theta_T} f_{\theta_T}(G) \cdot \nabla_G \theta_T]$$
$$\text{donde } \nabla_G \theta_{t+1} = \nabla_G \theta_t - \alpha \nabla_G \nabla \theta_t \mathcal{L}_{train}(f_{\theta_t}(G))$$

Los datos distorsionados finales, $G^{(\Delta)}$, se obtienen luego de aplicar Δ meta actualizaciones, llamadas M. Una forma sencilla de instanciar M es como (meta) descenso por gradiente con una tasa de aprendizaje β:

$$M(G) = G - \beta \nabla_G \mathcal{L}_{atk}(f_{\theta_T}(G))$$

Pero tal como señalan Zügner et. al. [57], este enfoque no funciona para problemas con datos discretos (como es el caso de los grafos), ya que el gradiente no está definido. Por esto se aplica una estrategia *greedy*: se define una función de puntuación $S : V \times V \to \mathbb{R}$ que le asigna a cada posible acción (arista o atributo a modificar) un valor numérico que indica su impacto (estimado) en la función objetivo del atacante $\mathcal{L}_{atk}$.

$$S(u, v) = \nabla_{a_{u,v}}^{meta}$$

Un gradiente negativo en una entrada (u, v) significa que el error incrementa cuando el valor decrementa. En el caso de la distorsión de aristas, un decremento solo tiene sentido si los nodos u y v están conectados mediante una arista, en cuyo caso el valor de la matriz de adyacencia en la posición (u, v)

era 1 y pasa a ser 0. Para determinar qué arista perturbar, sería necesario considerar si previo a la distorsión existía una arista o no. Habría que elegir el máximo de los gradientes donde no existía arista (ya que se añadiría una) y el mínimo donde sí la había (ya que se eliminaría), y entre esos dos posibles valores elegir aquel cuyo valor absoluto fuera más alto. Una alternativa para evitar eso es invertir el signo de los gradientes en posiciones cuyos nodos están conectados mediante una arista:

$$S(u, v) = \nabla^{meta}_{a_{u,v}} \cdot (-2 \cdot a_{u,v} + 1)$$

donde $a_{u,v}$ corresponde a la entrada (u, v) de la matriz de adyacencia.

Esto permite utilizar el argumento máximo de la matriz de meta-gradientes, sin considerar si antes existía o no una arista.

De acuerdo a la estrategia *greedy* propuesta, en cada iteración se selecciona perturbar la arista e' con el puntaje más alto:

$$e' = \arg \operatorname*{máx}_{e=(u,v)} S(u, v)$$

La idea original de Zügner et. al. [57] busca atacar los datos de la peor forma posible, causando el mayor daño en el *accuracy*. Esto se ve en el hecho de que buscan maximizar $\mathcal{L}_{atk}$, por lo que se elige la arista e' cuyo meta-gradiente es máximo. A esto le llamaremos **distorsión máxima**. En este trabajo también se propone buscar la **distorsión mínima**: aquella que tiene el impacto negativo más pequeño posible. En ese caso:

$$e' = \arg \operatorname*{mín}_{e=(u,v)} S(u, v) \mid S(u, v) > 0$$

La idea es comparar ambos extremos para analizar cómo afecta cada caso a GCN y estimar entonces el impacto de un ruido aleatorio en los datos, sin tener que simular tal cosa (que como mencionamos anteriormente sería prohibitivo dado su tiempo de cómputo en la mayoría de los casos prácticos).

Además de estudiar el caso de ruido en las aristas y en los atributos por separado, se estudia el impacto de ambos a la vez. En el caso de la distorsión máxima, en cada distorsión introducida se elige afectar una arista o un atri-

buto, dependiendo de cuál caso tenga un mayor meta-gradiente. En el caso de la distorsión mínima, el que tenga el menor meta-gradiente.

3.2. Algoritmo para estimar el impacto del ruido

Partiendo del algoritmo propuesto por Zügner et. al. [57] [1], se introducen modificaciones. Dado que los autores estudian ataques intencionados, definen restricciones que se deben cumplir para que los ataques se mantengan imperceptibles, tales como limitar la cantidad de cambios introducidos y exigir que los mismos no alteren drásticamente la distribución de grado de la red original. En nuestro caso el ruido puede ocurrir de cualquier manera, por lo que no se aplican restricciones de este tipo. Una restricción que sí se mantiene es la de evitar eliminar una arista que provoque el surgimiento de nodos aislados o eliminar bucles. Además se verifica que cada arista y atributo se modifique una única vez (a lo sumo), evitando así introducir ruido en elementos ya distorsionados.

La idea principal consiste en introducir ruido en las aristas (añadiendo nuevas o eliminando existentes) y en los atributos de los nodos (modificando su valor binario) hasta alcanzar un ***accuracy* objetivo**. Esta es otra diferencia respecto a Mettack (nombre que le dieron Zügner et. al. [57] a su meta ataque), ya que en ese caso se utiliza un número fijo de cambios que pueden realizarse. Una vez aplicados esos cambios, se vuelve a calcular el *accuracy* y se compara con el original (con datos no distorsionados). En nuestro caso, se busca calcular la cantidad máxima y mínima de aristas y atributos que hay que modificar para alcanzar el *accuracy* objetivo, según se elija en cada paso el elemento (atributo o arista) cuyo meta-gradiente es mayor o menor respectivamente.

El algoritmo se aplica a diferentes *datasets*. Dado que cada uno parte de un *accuracy* original distinto, no tiene sentido utilizar el mismo *accuracy* objetivo en todos los casos. Podemos pensar que un *dataset* que parte de un *accuracy* original de $0,80$ requerirá de menos distorsiones para llegar a un *accuracy* objetivo de $0,75$ que otro cuyo *accuracy* original es $0,87$. Por esta razón se define el *accuracy* objetivo en función del *accuracy* original (concretamente

[1] Disponible en https://github.com/danielzuegner/gnn-meta-attack

cinco puntos porcentuales por debajo).

3.2.1. Definir valores iniciales

El primer paso consiste en determinar el *accuracy* inicial, con los datos originales (sin ruido). Para esto se divide el *dataset* de forma aleatoria en datos de entrenamiento (10 % de los nodos) y datos de test (90 % restante de los nodos). Esto significa que solo se conocen las etiquetas del 10 % de los nodos durante la etapa de entrenamiento.

Se utiliza una GCN de dos capas con 16 neuronas en la capa oculta, tasa de aprendizaje de $0,01$, ReLU como función de activación y 200 épocas. La configuración de esta red es la propuesta por Kipf y Welling [21] y la misma que utilizan Zügner et. al. [57], dado que es el modelo de base más utilizado para una GCN [35].

Ya que los pesos de la red se inicializan aleatoriamente, dos GCNs que entrenen sobre el mismo *dataset* pueden llegar a métricas ligeramente distintas. Por esto es que para definir el *accuracy* original se toma el promedio del *accuracy* obtenido por veinte GCNs definidas de la misma forma y entrenadas sobre los mismos datos. Además de calcular el *accuracy* original para cada *dataset*, también se calcula la desviación estándar que luego se utiliza como parte de la condición de parada.

3.2.2. Introducir ruido

Una vez calculado el *accuracy* objetivo a partir del *accuracy* original, es momento de introducir ruido en las aristas, los atributos o ambos según el caso.

Como analizamos anteriormente, para esto es necesario resolver el problema de optimización (2.12). Se utiliza el mismo principio que Zügner et. al. [59] [57]: una estrategia secuencial donde primero se definen las distorsiones a partir de un modelo "sustituto" (*surrogate*) y una vez que se obtienen, las mismas se aplican sobre el *dataset* final.

La idea es que el modelo sustituto sea similar a una GCN y simule la idea de una convolución sobre grafos. Para eso, se reemplaza la función $\sigma(\cdot)$ de la

ecuación (2.10) con una función de activación lineal, obteniendo así:

$$Z = f_\theta(A, X) = softmax(\hat{A}^2 X W^{(0)} W^{(1)})$$

$W^{(0)}$ y $W^{(1)}$ son los pesos de *input-to-hidden* y *output-to-hidden* respectivamente.

Se propone utilizar un modelo sustituto estático para evitar abordar explícitamente el problema de optimización de dos niveles [57]. "Las soluciones basadas en meta-modelos se utilizan comúnmente para los problemas de optimización, en los que las evaluaciones de las funciones reales son costosas. Un meta-modelo o modelo sustituto es una aproximación del modelo real que es relativamente más rápido de evaluar. Sobre la base de una pequeña muestra del modelo real, se puede entrenar un modelo sustituto y utilizarlo posteriormente para su optimización"[41].

Una vez creado el modelo sustituto, se entrena durante 200 épocas para determinar los parámetros θ utilizando descenso por gradiente. Luego se computan los meta-gradientes para la matriz de adyacencia (distorsión de aristas), la matriz de atributos (distorsión de atributos) o ambos (caso combinado). Dependiendo de si se quiere introducir el mayor ruido posible o el menor, se elige la arista o atributo cuyo meta-gradiente es máximo o mínimo y se introduce ruido invirtiendo su valor original en la matriz de adyacencia o de atributos (0 o 1 según corresponda).

Esto se repite hasta que se alcanza la condición de parada (el *accuracy* objetivo). Notar que los parámetros de la red se reentrenan cada vez que se modifica una arista o atributo, al igual que se repite el cálculo de los meta-gradientes en cada iteración.

A continuación se incluye el algoritmo 1 para la distorsión máxima de aristas:

Algorithm 1 Distorsión máxima de aristas

1: **procedure** DistorsiónAristasMaximal($G = (A, X)$, etiquetas Z, accuracy_objetivo)
2: // *Introducir ruido*
3: **while** true **do**
4: // *Inicializar* θ_0 *aleatoriamente*
5: **for** t in (0..99) **do**
6: // *Actualización con SGD*
7: $\theta_{t+1} = step(\theta_t, \nabla_{\theta_t} \mathcal{L}_{train}(f_{\theta_t}(\hat{A}, X)))$
8: **end for**
9:
10: // *Computar meta-gradientes*
11: $\nabla^{meta}_{\hat{A}} \longleftarrow \nabla_{\hat{A}} \mathcal{L}(f_{\theta_T}(\hat{A}, X))$
12: $S \longleftarrow \nabla^{meta}_{\hat{A}} \odot (-2\hat{A} + 1)$
13: $e' \longleftarrow argmax(S)$ // *Excluyendo aristas ya modificadas*
14: $\hat{A} \longleftarrow$ agregar o eliminar arista e' de $\hat{A}$
15: $\hat{G} \longleftarrow (\hat{A}, X)$
16: cant_aristas += 1
17:
18: accuracy = evaluate($\hat{G}$, Z)
19: **if** accuracy $\leq$ accuracy_objetivo **then**
20: break
21: **end if**
22: **end while**
23:
24: **return** cant_aristas
25:
26: **end procedure**

Esta versión a alto nivel es análoga para el caso en que se busca la distorsión mínima. La única diferencia consiste en cambiar la línea 13 por:

Algorithm 2 Distorsión mínima de aristas

13: $e \longleftarrow argmin(\{s \in S \mid s > 0\})$ // *Excluyendo aristas ya modificadas*

El pseudocódigo del algoritmo completo (incluyendo sus variantes) se especifica en el Apéndice 1 de este documento.

Se introduce una última modificación para el caso minimal. Dado que se selecciona la arista o atributo con el impacto negativo más pequeño sobre el *accuracy*, requiere de un gran número de iteraciones llegar al *accuracy* objetivo.

Inicialmente probamos distorsionar un único atributo o arista por paso (igual que en el caso de distorsión máxima), pero tarda demasiado tiempo alcanzar la condición de parada. Por lo tanto, luego de distintas pruebas se determina que en el caso minimal se distorsionan las 100 aristas o los 100 atributos con menor impacto en cada iteración (previo a evaluar la condición de parada). Esto permite acelerar las ejecuciones y llegar a un resultado aproximado pero cercano al que se busca.

La problemática que puede generar esto es que modificar 100 elementos (aristas o atributos) tenga un impacto demasiado grande y se obtenga un *accuracy* incluso menor al deseado. En ese escenario, no se prevé la posibilidad de revertir distorsiones para ajustar ese número iterativamente. Para descartar esto, se ejecuta el experimento sobre los distintos *datasets* y se puede comprobar que no es el caso. Como se busca modificar la arista o el atributo con menor impacto, hacerlo para 100 a la vez no genera un cambio importante en el *accuracy*. En todas las situaciones que se alcanza la condición de parada, se obtiene un *accuracy* final muy próximo al buscado, lo que asegura que tomar 100 por vez no genera un impacto radical. Si se aplicara a otro conjunto de datos y esto no sucediese, simplemente debería reducirse el valor de este parámetro a un valor adecuado menor a 100.

Para el caso en que se introduce ruido de forma combinada, en cada paso se distorsionan los 100 elementos con menor impacto, sin importar si todos ellos son aristas, atributos o una combinación de ambos.

3.2.3. Evaluar condición de parada

Una vez que se introduce ruido en una arista o atributo, es necesario determinar si se llegó al *accuracy* objetivo. Para esto se instancia un nuevo modelo, que es evaluado y entrenado con la matriz de adyacencia y la matriz de atributos distorsionada. Utilizando la misma división de los datos que se aplica originalmente, se entrena con el 10 % de los nodos etiquetados y se evalúa sobre el 90 % restante. Dado que los meta-gradientes se calculan sobre toda la red, es posible que el ruido haya afectado a los datos de entrenamiento, los datos de test o ambos conjuntos, tal como se busca.

Debido a que el interés radica en comprender el comportamiento de GCN ante datos con ruido, el modelo utilizado es justamente GCN. Por lo tanto el

modelo sustituto solo actúa al momento de determinar dónde introducir ruido, pero el impacto del mismo se mide sobre una GCN tradicional, con las mismas características que la que se utiliza para determinar el *accuracy* y la desviación estándar original.

La GCN instanciada trabaja con la matriz de adyacencia y atributos con ruido para entrenar y luego se calcula el *accuracy* sobre el conjunto de test. Si el *accuracy* obtenido está suficientemente cerca del *accuracy* objetivo, se instancian diecinueve GCNs más con las mismas características, que entrenan y evalúan sobre los mismos datos con ruido que la anterior. Se calcula el *accuracy* para cada red y se promedian los veinte valores. Si el *accuracy* promedio es menor o igual al objetivo, finaliza el experimento, obteniendo la cantidad de aristas y/o atributos que fueron modificados para alcanzar la condición de parada. Si no, se introduce una nueva modificación y se repite el proceso.

Resta explicar qué se considera "suficientemente cerca" y aquí entra en juego la desviación estándar original calculada sobre los datos limpios. Se considera que el *accuracy* está lo suficientemente cerca del objetivo si es menor o igual al *accuracy* objetivo menos la desviación estándar original.

3.3. *Datasets* utilizados

En las últimas décadas las bases de datos bibliográficas han cambiado radicalmente en términos de accesibilidad. La mayoría de estas bases de datos están disponible en línea y pueden ser consultadas con simples búsquedas web.

A partir de los datos que se pueden recolectar de las bases de datos descritas anteriormente es posible construir distintos tipos de grafos de citación utilizando la lista de referencias de cada artículo. Dependiendo del nivel de agregación deseado, cada nodo puede ser un artículo (caso más simple), un autor, una institución, etc. En todos los casos el enlace del grafo queda determinado por la cita bibliográfica, por lo que en un principio el grafo resultante es dirigido, aunque puede ser tratado como no dirigido para atacar problemas de clasificación de nodos [48] (como es nuestro caso). Las aristas no tendrán pesos, puesto que todas las citas tienen la misma importancia.

A modo de resumen, se trabaja con grafos no dirigidos (su matriz de adyacencia es simétrica), sin peso en las aristas, con atributos binarios en sus nodos y sin

Dataset	Nodos	Aristas	Nodos_{LCC}	Aristas_{LCC}	Atributos	Clases
Cora	2.708	5.429	2.485	5.069	1.433	7
CiteSeer	3.312	4.732	2.110	3.757	3.703	6
Polblogs	1.490	16.725	1.222	16.714	-	2

Tabla 3.1: Datos principales de las redes utilizadas en el presente trabajo.

nodos aislados.

Para el presente trabajo se utilizan los siguientes grafos de citación: Cora [29], CiteSeer [13] y Polblogs [1]. En cada caso trabajamos con la componente conexa maximal (también llamada *largest connected component* o LCC). En la Tabla 3.1 se resumen algunas características principales de los *datasets*: cantidad de nodos, cantidad de aristas, cantidad de nodos de la LCC, cantidad de aristas de la LCC, cantidad de atributos y cantidad de clases posibles.

3.3.1. Cora

El *dataset* de Cora contiene artículos científicos de *Machine Learning* divididos en siete clases: *Case Based*, *Genetic Algorithms*, *Neural Networks*, *Probabilistic Methods*, *Reinforcement Learning*, *Rule Learning* y *Theory*. Para la construcción de la red de citación correspondiente a dicho *dataset* se aplicó un proceso de lematización, eliminación de las *stop words* y eliminación de las palabras con una frecuencia menor que 10 [39]. El resultado final es un corpus de 2.708 documentos, 1.433 palabras distintas (correspondientes a los atributos de un vector de *bag of words*) y 5.429 aristas. Los atributos de los nodos contienen ceros y unos indicando la presencia o no de las palabras en el documento. Los *papers* se seleccionaron de forma tal que en el corpus final cada uno cita o es citado por al menos otro de ellos, lo que significa que no existen nodos aislados. La Tabla 3.2 resume cuáles son las siete posibles clases y cuántas instancias de la LCC hay en cada una de ellas.

3.3.2. CiteSeer

Los *papers* de CiteSeer corresponden a seis categorías: *Agents*, *Artificial Intelligence*, *Database*, *Human Computer Interaction*, *Machine Learning* e *Information Retrieval*. Al igual que con Cora, se aplicó un proceso de lematización, eliminación de las *stop words* y de palabras con una frecuencia menor que 10 [39]. Los atributos de los nodos contienen ceros y unos indicando la presen-

Clase	# de instancias
Neural Networks	726
Case Based	285
Reinforcement Learning	214
Probabilistic Methods	379
Genetic Algorithms	406
Rule Learning	131
Theory	344

Tabla 3.2: Distribución de clase en la LCC de Cora.

Clase	# de instancias
Human Computer Interaction	304
Information Retrieval	532
Agents	463
Artificial Intelligence	115
Machine Learning	308
Database	388

Tabla 3.3: Distribución de clase en la LCC de CiteSeer.

cia o no de las palabras en el documento. La Tabla 3.3 resume cuáles son las seis posibles clases y cuántas instancias de la LCC hay en cada una de ellas.

3.3.3. Polblogs

Red formada a partir de hipervínculos entre blogs políticos de Estados Unidos. Las dos posibles clases son liberal y conservador. Algunos blogs fueron etiquetados manualmente, basándose en los hipervínculos entrantes y salientes y en las entradas del blog en el momento de las elecciones presidenciales de 2004. Los enlaces entre los blogs se extrajeron automáticamente aplicando *crawling* sobre la página principal del blog. Todas las aristas (originalmente dirigidas) se definieron sin dirección. Dado que los nodos no cuentan con atributos, en este caso se define $\mathbf{X} = \mathbf{I}_N$, donde $\mathbf{I}_N$ es la matriz identidad $N \times N$ y N es la cantidad de nodos del grafo. La Tabla 3.4 resume cuáles son las dos posibles clases y cuántas instancias de la LCC hay en cada una de ellas.

Clase	# de instancias
Liberal	586
Conservador	636

Tabla 3.4: Distribución de clase en la LCC de Polblogs.

3.3.4. Métricas descriptivas

Es de interés estudiar los *datasets* anteriores en mayor profundidad, particularmente alguna de sus propiedades estructurales.

Una de ellas es la asortatividad. Se define como "la vinculación selectiva entre vértices, según una o varias características determinadas" [22]. Puede ser utilizada para estimar el nivel global de homofilia dentro de una red con respecto a una característica específica (por ejemplo, el grado de sus nodos). Para su cálculo se utiliza el coeficiente de correlación de Pearson cuyo valor está entre -1 y 1: un valor positivo indica una correlación entre nodos con cierta característica similar, mientras que un valor negativo denota relaciones entre nodos con cierta característica opuesta. Cuando la correlación es igual a 1, la red tiene una mezcla perfecta de patrones de asortatividad, cuando es igual a 0 es no asortativa, mientras que cuando es igual a -1 la red es completamente desasortativa [31].

La asortatividad siempre se refiere a una propiedad o variable específica de los nodos de una red. Si los nodos tienen múltiples propiedades, es posible que los nodos se mezclen de manera asortativa para una propiedad pero de manera desasortativa para otra y no asortativa en una tercera propiedad de los nodos.

En particular interesa estudiar el coeficiente de asortatividad en relación a las clases de los nodos, determinando qué tendencia tienen los nodos de conectarse a otros con clases similares.

Si suponemos que cada vértice de un grafo G puede ser etiquetado con una de M clases (como es el caso del problema del presente trabajo), el coeficiente de asortatividad se define de la siguiente manera [22]. Sea $\mathbf{F}$ una matriz $\in \mathbb{R}^{M \times M}$, con f_{ij} la cantidad de aristas en G que unen un nodo de clase i a un nodo de clase j. Sea f_{i+} la suma de la fila i y f_{+i} la suma de la columna i. El coeficiente de asortatividad se define como:

$$r_a = \frac{\sum_i f_{ii} - \sum_i f_{i+} f_{+i}}{1 - \sum_i f_{i+} f_{+i}}$$

La Tabla 3.5 contiene los coeficientes de asortatividad de la componente conexa maximal de cada *dataset*, donde se observa una fuerte tendencia a relacionarse con nodos de igual clase.

Dataset	Asortatividad de clases
Cora	0,76
CiteSeer	0,66
Polblogs	0,81

Tabla 3.5: Coeficiente de asortatividad de la componente conexa maximal de los *datasets* utilizados.

Dataset	k_{mean}	k_{min}	k_{max}
Cora	4	1	169
CiteSeer	4	1	100
Polblogs	31	1	467

Tabla 3.6: Grado promedio (k_{mean}), grado mínimo (k_{min}) y grado máximo (k_{max}) de los vértices para cada *dataset*.

Otra métrica interesante consiste en analizar los grados de los nodos. Esto proporciona una cuantificación básica del grado de conexión de un nodo v con otros vértices del grafo. La Tabla 3.6 resume esta información mediante tres valores para cada *dataset*: el grado promedio de los vértices (k_{mean}), el grado mínimo en la red (k_{min}) y el grado máximo en la red (k_{max}) [25].

Capítulo 4

Presentación de los resultados, Análisis, y Discusión

En este capítulo se describe el entorno de ejecución de los experimentos realizados y las métricas analizadas. Se incluyen resultados de referencia para los mismos *datasets* utilizados y se presentan los principales resultados obtenidos al aplicar distorsión aleatoria y dirigida.

4.1. Entorno de ejecución

Ejecutamos los experimentos de forma remota utilizando el Centro Nacional de Supercomputación ClusterUY [1], infraestructura computacional que tiene un poder superior a 10.000 computadores tradicionales. Inicialmente probamos utilizar CPU pero luego comprobamos que era más conveniente utilizar GPU, excepto en un caso puntual donde esto no es posible ya que se excede la memoria disponible. Cabe destacar que los núcleos de cómputo CPU disponen de 3,5 TB de memoria RAM y los núcleos de cómputo GPU de 12Gb de memoria. Otras herramientas utilizadas fueron Python 2.7.5, TensorFlow 1.12, Jupyter Notebook, NumPy y SciPy.

La métrica utilizada para analizar los resultados es *accuracy*: la proporción de predicciones correctas con respecto al número total de predicciones realizadas. Se elige esta alternativa por ser la más utilizada en trabajos similares. Ejem-

[1] https://www.cluster.uy/

Ref.	Método	Cora	Citeseer	Polblogs
[59]	**GCN**	**0,90**	**0,88**	**0,93**
[59]	CLN	0,84	0,76	0,92
[59]	DW	0,82	0,71	0,63
[45]	GS	0,85	0,83	0,86

Tabla 4.1: Resultados obtenidos en la tarea de clasificación de nodos según el modelo y *dataset* utilizado. Los *datasets* se corresponden a los presentados en el capítulo 3 y los métodos son *Graph Convolutional Network*, *Column Networks*, *DeepWalk* y *GraphSage*. La métrica utilizada es *accuracy*.

plos recabados en la Tabla 2.2 también utilizan en su mayoría *accuracy* como métrica.

Esta métrica podría ser problemática en casos en los que la cantidad de instancias de cada clase está desequilibrada, lo que se conoce como *accuracy paradox* [44]. Por ejemplo, si consideramos un *dataset* donde el 98 % de los casos pertenecen a la clase A y solo 2 % a la clase B, entonces un modelo que siempre prediga la clase A tendrá un *accuracy* de 98 %. Pero eso no es problemático en el caso de los *datasets* elegidos porque como se menciona en el capítulo anterior, la distribución de ejemplos por clase es balanceada (ver Tablas 3.2, 3.3 y 3.4).

4.2. Resultados de referencia

Los tres *datasets* presentados en el capítulo anterior han sido utilizados en múltiples trabajos de investigación para evaluar el rendimiento de diversos métodos ante el problema de clasificación de nodos. La Tabla 2.2 recaba numerosos ejemplos que los han utilizado. La Tabla 4.1 presenta algunos resultados de distintos modelos a modo de referencia en donde se observa un desempeño superior de GCN sobre los otros modelos.

Estos mismos *datasets* son utilizados por Ji. et. al. [19] para comparar distintos ataques no dirigidos sobre grafos. Particularmente comparan Mettatack [57], DICE [46] y *Topology attack* [49]. A diferencia de Mettatack, DICE es un ataque de caja blanca que aleatoriamente conecta nodos que pertenecen a diferentes clases o elimina aristas entre nodos que comparten clase [19]. *Topology attack* es un ataque basado en distorsión de aristas que vence la dificultad de atacar la estructura discreta de un grafo desde una perspectiva de optimización

Método	Cora	Citeseer	Polblogs
Clean	**0,83**	**0,75**	**0,96**
Mettatack	0,77	0,73	0,74
DICE	0,82	0,74	0,93
Topology attack	0,72	0,79	0,72

Tabla 4.2: Comparación de *accuracies* obtenidos mediante GCN en los *datasets* limpios (sin ruido) y luego de aplicar distintos ataques que alteran la red original en un 5 %. Resultados tomados de [19].

de primer orden [49].

Para cada *dataset* se selecciona aleatoriamente un 10 % de nodos para entrenamiento, 10 % para validación y el 80 % restante para testeo. El experimento se ejecuta cinco veces y los resultados promedios se pueden ver en la Tabla 4.2. Los números presentados corresponden al *accuracy* logrado por GCN luego de aplicar una distorsión del 5 %.

4.3. Resultados de la evaluación aleatoria

Como se menciona anteriormente, la primera alternativa pensada para introducir ruido consiste en hacerlo aleatoriamente. Se propone modificar las entradas de la matriz de adyacencia y la matriz de atributos a partir de un parámetro *distortion rate* que determina, en porcentaje, cuánto ruido se introduce. Dado que esta es una idea inicial para explorar la problemática se trabaja solo con Cora.

Este experimento se realiza variando el porcentaje de nodos que la red conoce (*label rate*). Se toman valores entre $0,1$ y $0,9$ pero no hay diferencia significativa entre estos para ninguno de los dos casos (aristas y atributos). Una primera conclusión que cabe destacar es que GCN obtiene muy buenos resultados a pesar de conocer muy poco de la red. Esta es una cualidad muy positiva ya que con muy pocos datos de entrenamiento se puede predecir de forma correcta. La Figura 4.1 demuestra que el *accuracy* varía muy poco a medida que el *label rate* aumenta. La diferencia entre conocer el 10 % y el 90 % de los nodos de la red es tan solo un 5 % de incremento en el *accuracy*. Es por esto que a modo de simplificación, todos los resultados presentados se corresponden a un *label rate* $= 0,1$ (10 %).

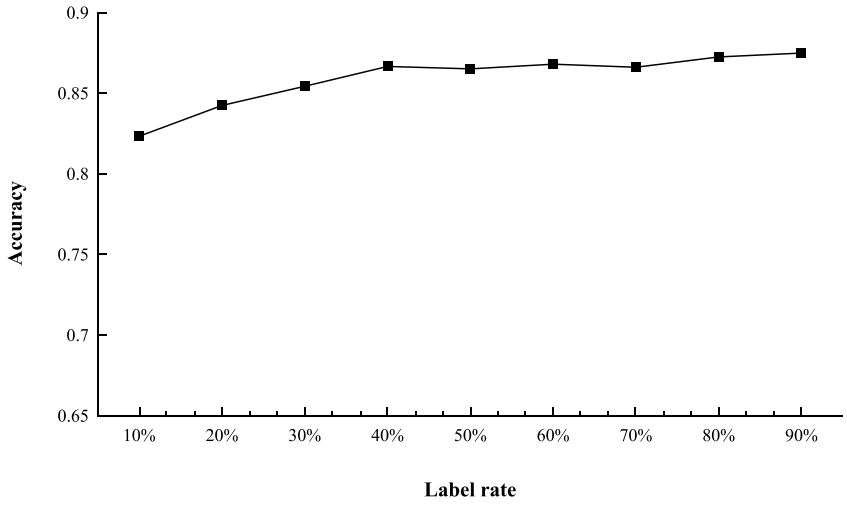

Figura 4.1: *Accuracy* obtenido por GCN para Cora (sin distorsión) de acuerdo a los diferentes valores de *label rate* utilizados.

En todos los casos se realizan diez ejecuciones del experimento dividiendo aleatoriamente el conjunto de datos en un conjunto de entrenamiento y otro de prueba según el *label rate*. Los tiempos reportados se encuentran expresados en segundos y comprenden tanto el entrenamiento del modelo como su correspondiente evaluación (no incluye el tiempo de distorsión). Los resultados expuestos corresponden al promedio de todas las ejecuciones.

En la Tabla 4.3 y en la Figura 4.2 correspondiente se encuentran los resultados obtenidos al distorsionar aleatoriamente la matriz de atributos variando equitativamente el *distortion rate* entre $0,1$ y $0,9$.

Se puede observar que se necesita distorsionar un gran porcentaje de entradas de la matriz de atributos para lograr una reducción considerable del *accuracy*. Como se menciona en el capítulo anterior, esto puede deberse a que la matriz de atributos de Cora es dispersa. Con un *distortion rate* de $0,1$ se fuerza que el $10\,\%$ de las entradas de la matriz de atributos sea 0 pero no es posible saber cuántas entradas efectivamente se distorsionan (cuántas eran 1 originalmente).

La Tabla 4.4 y la Figura 4.3 resumen los resultados de la distorsión aleatoria de aristas. La escala tomada para los valores del *distortion rate* es distinta a la utilizada para la distorsión de atributos ya que para valores de *distortion rate* superiores a $0,02$ no se observa una variación importante del *accuracy*.

Distortion rate	***Accuracy***	**Tiempo (s)**
0,1	0,81	0,04
0,2	0,81	0,05
0,3	0,80	0,06
0,4	0,79	0,09
0,5	0,78	0,11
0,6	0,76	0,16
0,7	0,74	0,18
0,8	0,69	0,22
0,9	0,64	0,26

Tabla 4.3: Resultados de distorsionar aleatoriamente los atributos de la red de Cora. Los números expresados corresponden al promedio de 10 ejecuciones.

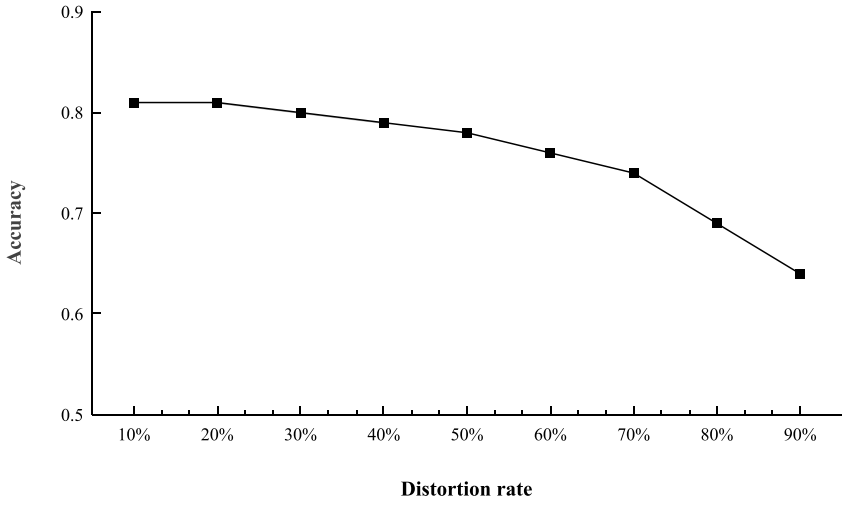

Figura 4.2: *Accuracy* obtenido por GCN luego de distorsionar aleatoriamente distintas cantidades de atributos de los nodos de Cora. Los números expresados corresponden al promedio de 10 ejecuciones.

Distortion rate	***Accuracy***	**Tiempo (s)**
0,001	0,74	0,04
0,01	0,40	0,06
0,02	0,31	0,07
0,05	0,30	0,10
0,1	0,30	0,13
0,2	0,30	0,18
0,5	0,30	0,23

Tabla 4.4: Resultados de distorsionar aleatoriamente las aristas de la red de Cora. Los números expresados corresponden al promedio de 10 ejecuciones.

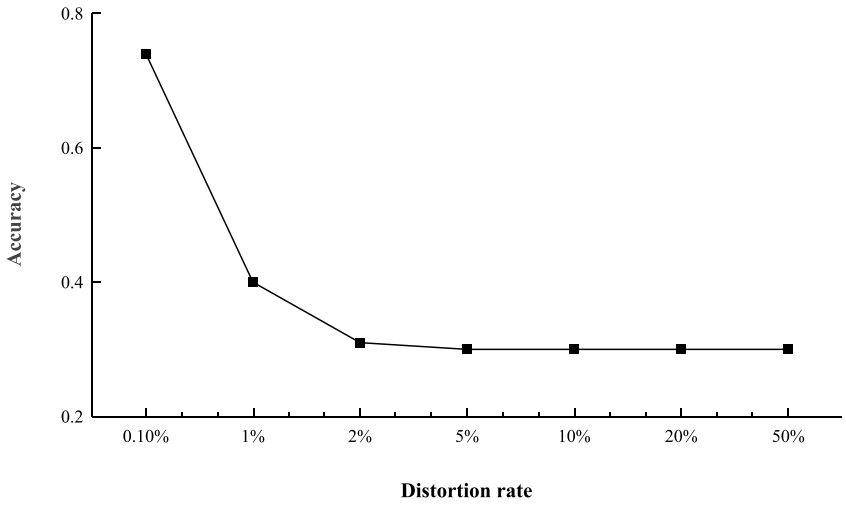

Figura 4.3: *Accuracy* obtenido por GCN luego de distorsionar aleatoriamente distintas cantidades de aristas de la red de Cora. Los números expresados corresponden al promedio de 10 ejecuciones.

A diferencia del caso de los atributos, una pequeña distorsión en las aristas tiene un gran impacto en el *accuracy* final, por lo que fue de interés explorar valores más pequeños de *distortion rate* para analizar el comportamiento en mayor detalle.

4.4. Resultados del método principal: la evaluación dirigida

Se presentan a continuación los resultados de introducir ruido en la red mediante el cálculo de meta-gradientes. Interesa comparar las diferencias entre

aplicar distorsión máxima y mínima para cada *dataset* y variante: ruido en aristas, en atributos y en ambos a la vez.

Los resultados corresponden al promedio de 10 ejecuciones, donde en cada una de ellas se utiliza una división diferente de los datos en conjunto de entrenamiento y test, siempre manteniendo la misma relación (10 % para entrenamiento y 90 % para test). Además del promedio, se incluye el valor mínimo y máximo obtenido para cada caso. Todos los valores corresponden a la cantidad de elementos que hay que modificar para reducir el *accuracy* original en un 5 %.

4.4.1. Ruido en aristas

La Tabla 4.5 reúne la cantidad de aristas que hay que distorsionar para alcanzar al *accuracy* objetivo. En base a estos datos, considerando el *dataset* Citeseer es necesario modificar 115 aristas en el caso de distorsión máxima (en promedio). Si la componente conexa maximal de Citeseer fuera una red en malla (donde cada nodo se conecta a todos los demás), esto equivaldría al 0,005 % de las aristas totales. En la realidad la LCC de Citeseer tiene 3.757 aristas, resultando en un 3 % de aristas existentes. Analicemos ahora el caso de distorsión mínima para Citeseer. En promedio se requiere distorsionar 2.300 aristas: 0,10 % de aristas posibles totales y 61 % de aristas existentes.

Para el caso de Cora, al aplicar distorsión máxima en promedio es necesario modificar 100 aristas para llegar al *accuracy* objetivo. Esto representa aproximadamente un 0,003 % de aristas si la red fuera en malla. Considerando que la LCC de Cora tiene 5.069 aristas, 100 aristas equivale aproximadamente al 2 %. Para la distorsión mínima, se modifican en promedio 4.600 aristas, lo que equivale a un 0,15 % de aristas si el grafo fuera completo y 91 % de las aristas existentes.

Por último, Polblogs genera distorsiones en 78 aristas (en promedio) para el caso máximo: 0,010 % de aristas totales, 5 % de aristas existentes. Para el caso de distorsión mínima, se modifican en promedio 374.700 aristas: 50 % de aristas totales (si el grafo de la LCC fuera completa). Si solo consideramos la cantidad de aristas existentes (16.714) vemos que la cantidad de aristas que hay que distorsionar es 20 veces superior.

Como se puede observar, existe una gran diferencia entre Polblogs y los otros dos *datasets* para la distorsión mínima. Aunque la matriz de adyacencia de Polblogs tiene un tamaño que es aproximadamente la mitad de las otras dos, la diferencia se puede atribuir a que el grado promedio de los nodos de Polblogs es mucho mayor (ver Tabla 3.6), por lo cual la clasificación de un nodo podría depender de más aristas que en el caso de Cora o Citeseer, por lo que modificar una no tendría un impacto tan grande en el *accuracy*. Para la distorsión máxima, sin embargo, no hay tal diferencia. El algoritmo muestra ser efectivo al momento de elegir qué arista atacar y consigue reducir el *accuracy* con menos de un centenar de modificaciones.

Para los tres *datasets* hay una diferencia considerable entre el mínimo y el máximo. La Figura 4.4 muestra la proporción entre la cantidad de aristas modificada en el caso de distorsión máxima y mínima. Esto indica que no todas las aristas tienen el mismo peso a la hora de clasificar nuevos nodos y que si se eligen adecuadamente, la presencia de ruido en pocas aristas puede tener un impacto muy grande. Esta diferencia confirma que aplicar una distorsión aleatoria (como exploramos inicialmente) no es efectivo, ya que en ese caso se considera que todas las aristas tienen la misma relevancia para calcular la clase de un nodo. Estos resultados muestran claramente que eso no es así. Y obtener estos resultados utilizando el método aleatorio requeriría evaluar una cantidad demasiado grande de casos de distorsiones.

Por otro lado, la Tabla 4.6 muestra los tiempos necesarios (en segundos) para la distorsión de aristas para las distintas combinaciones. Si bien la distorsión mínima modifica muchos más elementos que la distorsión máxima, aquí se observa que el tiempo necesario para hacerlo puede ser menor puesto que se distorsionan 100 aristas por iteración, lo cual implica un solo cálculo de meta-gradientes. También se nota un aumento considerable del tiempo requerido en comparación con la distorsión aleatoria presentada anteriormente. Esto puede explicarse porque la distorsión aleatoria elige el elemento a modificar al azar sin necesidad de realizar cálculos adicionales. En cambio calcular meta-gradientes necesariamente requiere de más tiempo.

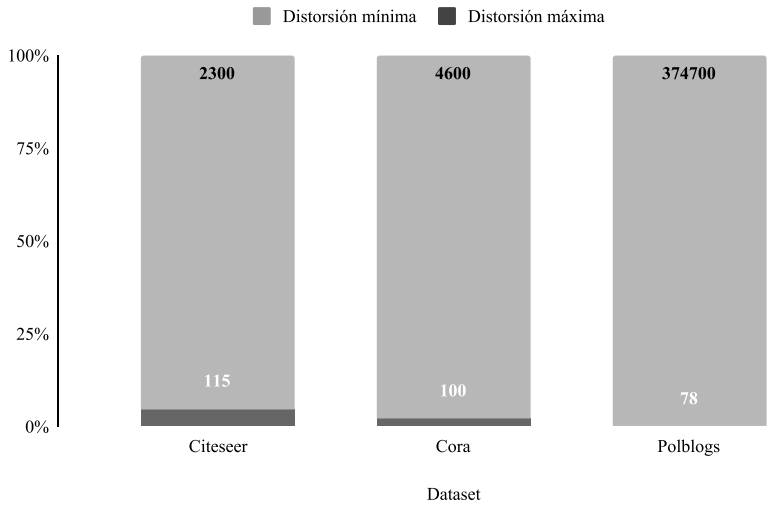

Figura 4.4: Comparación entre la cantidad de aristas que deben modificarse en el caso de distorsión máxima y mínima para alcanzar el *accuracy* objetivo.

Dataset	**Distorsión mínima**			**Distorsión máxima**		
	Mínimo	Promedio	Máximo	Mínimo	Promedio	Máximo
Citeseer	1.800	**2.300**	2.500	83	**115**	165
Cora	3.500	**4.600**	5.500	71	**100**	117
Polblogs	328.100	**374.700**	419.500	64	**78**	91

Tabla 4.5: Cantidad de aristas que deben distorsionarse para llegar al *accuracy* objetivo. Los números presentados corresponden a 10 ejecuciones.

Dataset	**Distorsión mínima**			**Distorsión máxima**		
	Mínimo	Promedio	Máximo	Mínimo	Promedio	Máximo
Citeseer	105	**133**	153	314	**435**	633
Cora	117	**158**	206	208	**314**	471
Polblogs	8177	**9199**	11003	130	**177**	248

Tabla 4.6: Tiempo (en segundos) correspondiente a la distorsión de aristas necesaria para llegar al *accuracy* objetivo. Los números presentados corresponden a 10 ejecuciones.

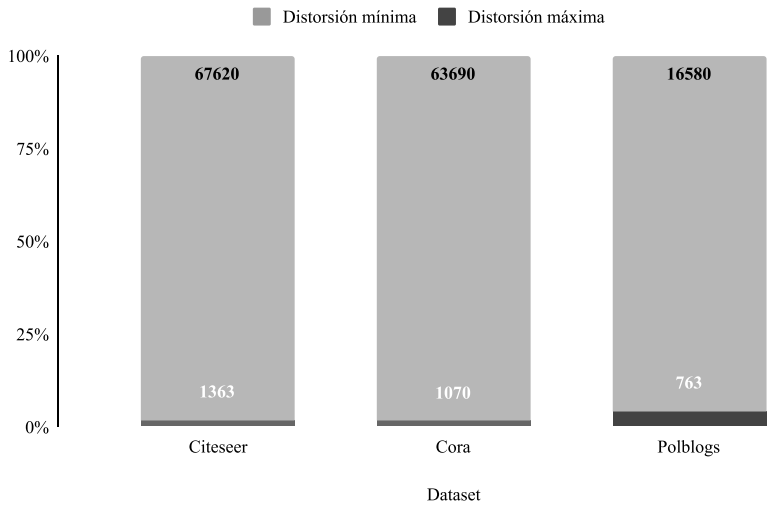

Figura 4.5: Comparación entre la cantidad de atributos que deben modificarse en el caso de distorsión máxima y mínima para alcanzar el *accuracy* objetivo.

4.4.2. Ruido en atributos

Las Tablas 4.7 y 4.8 resumen los resultados obtenidos al introducir ruido en los atributos para cada *dataset*, en el caso de distorsión mínima y máxima.

Se observa que ocurre algo similar al caso de aristas. La cantidad de atributos modificados en el caso de distorsión mínima es muchísimo mayor que la cantidad en el caso de distorsión máxima para todos los *datasets*. La Figura 4.5 muestra la proporción para cada caso. En este caso no hay una diferencia tan grande entre Polblogs y los otros dos *datasets*.

Los resultados de distorsión aleatoria presentan una enorme diferencia entre la distorsión de aristas y atributos. Incluso, como se menciona, para el caso de aristas el impacto es tan grande que se utiliza un *distortion rate* inferior (una escala más reducida de valores). Sin embargo al introducir ruido a partir del cálculo de meta-gradientes (en contraste con modificaciones aleatorias) vemos que la diferencia no es tan notoria.

Pero sí se cumple lo siguiente: en todos los casos la cantidad de aristas que hay que distorsionar para alcanzar el *accuracy* objetivo en el caso de distorsión máxima es inferior al número de atributos que hay que modificar para el mismo caso. Algo similar ocurre con la cantidad de aristas y atributos en el caso de

Dataset	Distorsión mínima			Distorsión máxima		
	Mínimo	Promedio	Máximo	Mínimo	Promedio	Máximo
Citeseer	54.400	**67.620**	74.300	800	**1.363**	2.300
Cora	53.800	**63.690**	83.700	497	**1.070**	1.605
Polblogs	6.800	**16.580**	40.300	327	**763**	1.484

Tabla 4.7: Cantidad de atributos que deben distorsionarse para llegar al *accuracy* objetivo. Los números presentados corresponden a 10 ejecuciones.

Dataset	Distorsión mínima			Distorsión máxima		
	Mínimo	Promedio	Máximo	Mínimo	Promedio	Máximo
Citeseer	2263	**2728**	3451	2737	**4255**	6814
Cora	1289	**1919**	2817	1516	**2947**	4960
Polblogs	141	**295**	855	478	**1104**	2737

Tabla 4.8: Tiempo (en segundos) correspondiente a la distorsión de atributos para llegar al *accuracy* objetivo. Los números presentados corresponden a 10 ejecuciones.

distorsión mínima, excepto para el caso de Polblogs donde se modifican 374.400 aristas y 16.580 atributos. Tiene sentido dejar este caso de lado, ya que Polblogs no cuenta con atributos propios (en su lugar se utiliza la matriz identidad) y por tanto parece ser un caso particular.

De acuerdo a estos resultados podemos afirmar que, al menos para estos *datasets*, las aristas parecen tener más peso en la correcta clasificación de los nodos en comparación con sus atributos.

4.4.3. Ruido en aristas y atributos combinados

Las Tablas 4.9 y 4.10 reúnen los resultados obtenidos en el caso de la distorsión de atributos y aristas combinados. Para este caso, en cada iteración se elige si modificar una arista o un atributo de acuerdo al impacto que se busca (el mayor posible en la distorsión máxima, el menor posible en la distorsión mínima).

Nuevamente la diferencia entre la cantidad de elementos modificados entre la distorsión máxima y mínima es muy grande (lo que es consistente con los casos anteriores). La Figura 4.6 muestra la proporción entre la cantidad de elementos distorsionados (sin importar si son aristas o atributos) en ambos casos.

Es interesante destacar que el comportamiento no es el mismo para los tres *datasets*. La Figura 4.7 indica que para el caso de distorsión máxima siempre se introduce ruido en aristas en el caso de Citeseer y Cora. Sin embargo en el

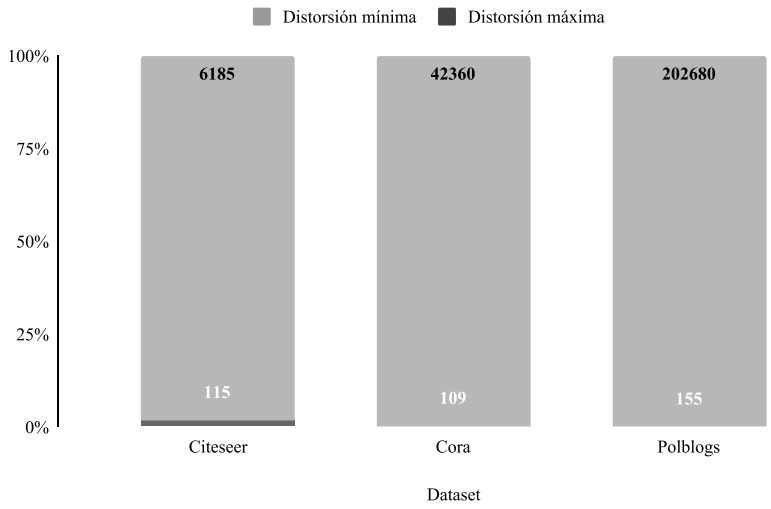

Figura 4.6: Comparación entre la cantidad de elementos (aristas o atributos) que deben modificarse en el caso de distorsión máxima y mínima para alcanzar el *accuracy* objetivo.

Dataset	**Distorsión mínima**			**Distorsión máxima**		
	Mínimo	Promedio	Máximo	Mínimo	Promedio	Máximo
Citeseet	52.500	**61.850**	80.700	61	**115**	171
Cora	25.600	**42.360**	52.900	69	**109**	164
Polblogs	73.700	**202.680**	615.600	110	**155**	267

Tabla 4.9: Cantidad de distorsiones aplicadas (sobre aristas y atributos) para llegar al *accuracy* objetivo (promedio de 10 ejecuciones).

caso de Polblogs, la mayoría de las veces se introduce ruido en atributos.

La Figura 4.8 muestra que para el caso de distorsión mínima, el comportamiento es el inverso: Polblogs introduce más ruido en aristas que en atributos y lo opuesto ocurre para Citeseer y Cora.

Luego de presentados los resultados, resta destacar algunas observaciones.

En primer lugar, identificamos que no todas las aristas ni todos los atributos tienen el mismo peso en la tarea de clasificación de nodos. Al introducir ruido en ciertos elementos que parecen ser fundamentales, el impacto es mucho mayor que en otros. En todos los casos (distorsión exclusiva de aristas, de atributos o ambos) y en los tres *datasets* se cumple que la cantidad de elementos que deben modificarse es sustancialmente menor si se eligen aquellos con mayor impacto que los que tienen menor impacto. Si todas las aristas y todos los

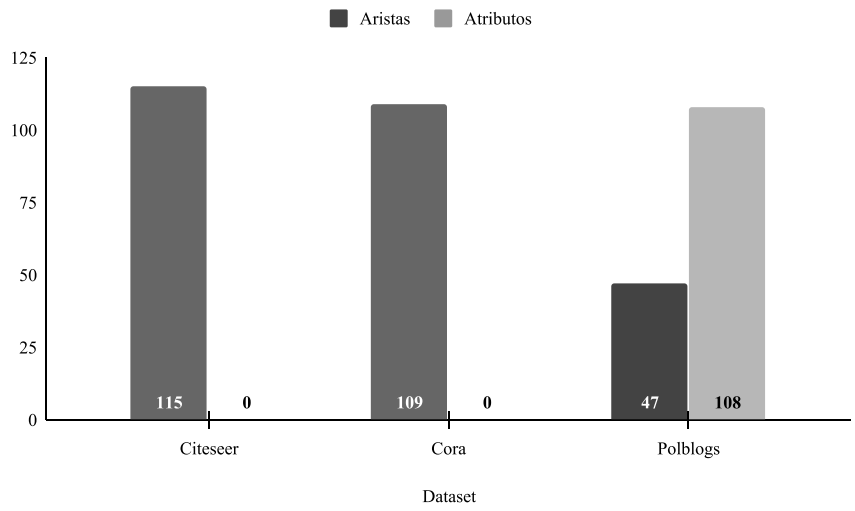

Figura 4.7: Cantidad de aristas y atributos que deben modificarse en el caso de distorsión máxima para alcanzar el *accuracy* objetivo.

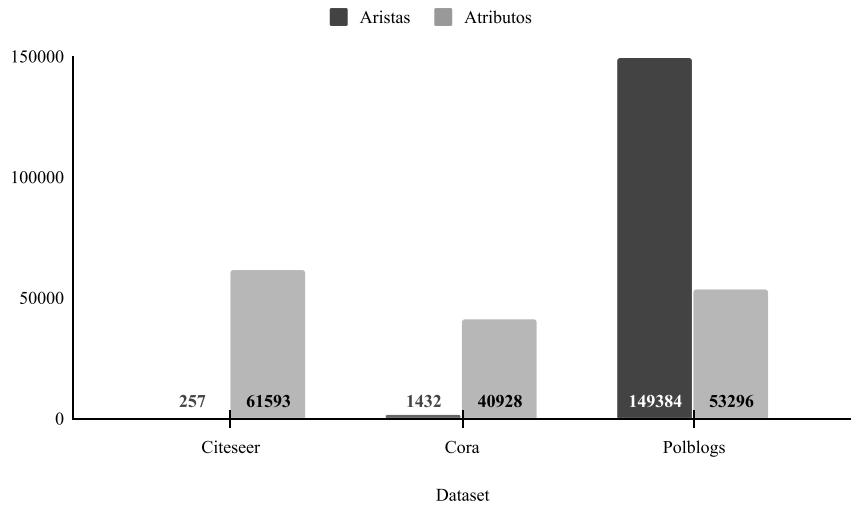

Figura 4.8: Cantidad de aristas y atributos que deben modificarse en el caso de distorsión mínima para alcanzar el *accuracy* objetivo.

Dataset	**Distorsión mínima**			**Distorsión máxima**		
	Mínimo	Promedio	Máximo	Mínimo	Promedio	Máximo
Citeseer	2130	**2818**	4028	290	**614**	1031
Cora[1]	11806	**19499**	23431	3480	**5170**	7424
Polblogs	1622	**6202**	21829	244	**310**	476

[1]Ejecución en CPU en lugar de GPU debido a una limitante de memoria.

Tabla 4.10: Tiempo (en segundos) correspondiente a la distorsión de aristas y atributos para llegar al *accuracy* objetivo. Los números presentados corresponden a 10 ejecuciones.

atributos tuvieran la misma incidencia, los números obtenidos en la distorsión máxima y mínima deberían de haber sido similares.

En segundo lugar, para estos *datasets* en particular las aristas parecen jugar un papel más importante para lograr una clasificación correcta que los atributos. Dejando de lado el caso de Polblogs (ya que los nodos no cuentan con atributos), Cora y Citeseer priorizan introducir ruido en aristas en lugar de atributos para generar un impacto grande en el *accuracy* (distorsión máxima). Por otro lado, también priorizan la elección de atributos ante aristas más frecuentemente al buscar el menor impacto (distorsión mínima). A modo de ejemplo, en Citeseer se elige por introducir ruido en 61.593 atributos y solo en 257 aristas.

Algo similar destacan [57] en su trabajo. A modo experimental, evalúan el comportamiento de Citeseer al atacar atributos junto a aristas. En lugar de introducir perturbaciones hasta alcanzar cierto *accuracy*, los autores proponen introducir cierto número fijo de perturbaciones Δ. Sus resultados muestran que el impacto de atacar aristas y atributos a la vez es menor al impacto de sólo atacar aristas. Atribuyen esto al hecho de que asignan el mismo costo a los cambios de estructura y de atributos, pero sostienen que una perturbación de la estructura tiene un efecto más fuerte sobre el *accuracy* que una perturbación en los atributos. Sus resultados en este ámbito son consistentes con los obtenidos en este trabajo.

Por último, parecería razonable esperar que al buscar el impacto mayor (distorsión máxima), la cantidad de elementos distorsionados al considerar tanto aristas como atributos sea menor a la cantidad al considerar exclusivamente aristas o exclusivamente atributos. Esto es porque si no hay una limitante respecto a si elegir atributos o aristas, es posible elegir el elemento que causa

mayor impacto. En cambio cuando sólo se busca distorsionar aristas o atributos, las opciones son más restringidas. Por ejemplo, al introducir ruido en aristas es posible que exista un atributo que causa más daño pero como no se puede utilizar, hay que seleccionar en cambio la arista con mayor impacto. Esto no es cierto para ningún caso. Tampoco se cumple que la cantidad de aristas y atributos que se debe distorsionar en el caso mínimo sea mayor que solo la cantidad de aristas o solo la cantidad de atributos (nuevamente porque al no haber limitantes, sería posible elegir el elemento que causa menor impacto). Entendemos que esto se debe a la estrategia *greedy* utilizada, ya que el máximo (mínimo) local no necesariamente coincide con el elemento que tiene más (menos) impacto en el *accuracy*.

Capítulo 5

Conclusiones y trabajo futuro

En este capítulo se sintetiza el trabajo llevado a cabo, se exponen las principales conclusiones a las que se llega y se plantean ciertos aspectos que resultaría interesante explorar a futuro.

5.1. Conclusiones

El presente trabajo estudia las *Graph Convolutional Networks*, una extensión al aprendizaje profundo para trabajar con problemas donde los datos son mejor representados mediante un grafo. En particular se centra en analizar qué tan robusto es este tipo de *Graph Neural Network* ante datos con ruido, entendido no solo como errores aleatorios en los datos de entrada, sino también como posibles errores sistemáticos.

El impacto de los datos con ruido se encuentra bien documentado para distintos métodos de aprendizaje profundo, incluidas algunas arquitecturas de GNNs, pero no se encontraron trabajos que analicen el impacto del ruido específicamente para GCNs. Sin embargo, sí hay antecedentes de investigaciones sobre la robustez de GCN (y otras GNNs) ante ataques intencionados, los cuales se encuentran contemplados en nuestra definición de ruido y sirven como punto de partida para sistematizar la introducción de ruido en los *datasets*.

Se evalúa el comportamiento de GCN frente al problema de clasificar los nodos de un grafo en clases. Para dicha evaluación se consideran tres redes de citación emblemáticas: Citeseer, Cora y Polblogs. Como vimos anteriormente, estas

redes son ampliamente utilizadas por distintos autores y suelen ser tomadas para elaborar puntos de referencia de distintos algoritmos y arquitecturas que trabajan con grafos.

Una vez especificada la pregunta de investigación ("¿Cómo se comporta GCN ante datos con ruido?") y los datos con los que trabajar, se busca definir una metodología para introducir y evaluar el impacto de errores en los datos de entrada para GCN. Inicialmente introducimos ruido aleatoriamente como forma de validación del problema, pero finalmente abordamos la problemática a partir de lo que proponen Zügner et. al. [57] para atacar el grafo. Los autores invierten el proceso de optimización basado en gradientes y tratan al grafo como un hiperparámatro a ser aprendido.

Se plantea introducir ruido en las aristas, atributos y una combinación de ambos, buscando la cantidad de elementos que hay que modificar para reducir el *accuracy* de la clasificación en un 5 %. Se consideran dos escenarios: en el primero se busca causar el mayor daño posible con cada elemento distorsionado (distorsión máxima) mientras que en el segundo se intenta lograr lo opuesto (distorsión mínima). De esta manera la idea original es modificada para obtener un vector de meta-gradientes que indique qué arista o atributo distorsionar si queremos causar más o menos daño.

Los resultados obtenidos muestran que, en el caso de las redes consideradas, no todas las aristas ni todos los atributos influyen de la misma manera en la clasificación de un nodo, por lo que el impacto del ruido depende de qué elementos son modificados. Para todas las configuraciones del experimento se verifica que siempre que se modifica el elemento con mayor impacto es necesario introducir una cantidad considerablemente menor de ruido en comparación con el caso en el que se siempre se modifica el elemento con menor impacto.

Si consideramos solo las dos redes que tienen atributos, se cumple que el ruido introducido en las aristas tiene un mayor impacto en el rendimiento de GCN que el ruido en los atributos, por lo que para estos casos las aristas parecen tener mayor importancia al momento de clasificar un nodo. Además se observa que en el caso opuesto donde se busca el menor impacto en cada distorsión, se priorizan los atributos sobre las aristas.

Por otro lado, se esperaba poder encontrar una relación entre el caso que

distorsiona tanto aristas y atributos y los casos que consideran exclusivamente aristas o exclusivamente atributos. En un principio parecía razonable pensar que cuando se busca el mayor impacto en cada caso, la opción mixta es menos restrictiva por lo que necesitaría modificar menos elementos que la opción que elige aristas o atributos por separado. Sin embargo, debido a la naturaleza *greedy* del algoritmo esto no es cierto. Entendemos que esto se debe a que los elementos distorsionados son óptimos locamente pero no necesariamente son los que tienen mayor impacto sobre el rendimiento global.

5.2. Trabajo futuro

Si bien este trabajo comienza a explorar el comportamiento GCN ante datos con ruido, el mismo puede ser extendido en diversos aspectos. En este caso se estudia la tarea de clasificar nodos, pero también sería de interés analizar la robustez de GCN ante otras tareas con grafos, como predicción de enlaces o detección de comunidades.

Por otra parte, sería de interés estudiar la posibilidad de trasladar esta metodología a otras arquitecturas GNN e investigar si se obtienen resultados similares. Tal vez sea necesario introducir cambios, por ejemplo en la definición del *surrogate model*.

En cuanto a lo que refiere a las redes, los experimentos son realizados con solo tres, que si bien son emblemáticas de la literatura, no son suficientes como para generalizar ningún resultado obtenido. Se podría repetir el experimento utilizando redes distintas y analizar si se llega a las mismas conclusiones. En el mismo espíritu, también se deberían considerar *datasets* que presenten distintas características de forma de trabajar con un conjunto heterogéneo de redes. Un aspecto en el cual las tres redes utilizadas son muy similares es la asortatividad, puesto que los nodos de todas las redes presentan una alta tendencia a relacionarse con otros de igual clase, por lo que valdría la pena analizar el caso para redes más dispares en cuanto a su asortatividad.

Además, si bien los tres *datasets* presentados tienen atributos binarios se puede modificar el algoritmo para contemplar atributos que pertenezcan a los números reales. Este caso es más complejo ya que al introducir ruido no solo se debe contemplar cuántos atributos modificar sino además qué tanto modificar

cada uno. Esto requiere encontrar una forma de asegurar que el nivel de ruido introducido en una matriz de atributos reales sea similar al introducido en una binaria. Una primera solución a este problema podría ser sortear un número real (basado en la distribución de los valores existentes) para la matriz real e invertir el valor de los atributos para la matriz binaria. Para asegurar que el ruido es equivalente en los dos tipos de matrices se puede calcular y comparar las normas de la diferencia de las matrices antes y después de aplicar la distorsión.

Por último, una posible continuación de la investigación podría ser explorar qué determina que una arista o atributo influya más o menos en la clasificación de un nodo. Por ejemplo, en el caso de las aristas, se podría evaluar su relación con las métricas de centralidad de arista.

Referencias bibliográficas

[1] L. A. Adamic and N. Glance. The political blogosphere and the 2004 u.s. election: Divided they blog. In *Proceedings of the 3rd International Workshop on Link Discovery*, LinkKDD '05, page 36–43, New York, NY, USA, 2005. Association for Computing Machinery.

[2] A. Atla, R. Tada, V. Sheng, and N. Singireddy. Sensitivity of different machine learning algorithms to noise. *J. Comput. Sci. Coll.*, 26(5):96–103, May 2011.

[3] D. Bacciu, F. Errica, A. Micheli, and M. Podda. A gentle introduction to deep learning for graphs. *arXiv preprint arXiv:1912.12693*, 2019.

[4] S. Basu. Semi-supervised learning. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 2613–2615. Springer US, Boston, MA, 2016.

[5] A. Bojchevski, Y. Matkovic, and S. Günnemann. Robust spectral clustering for noisy data: Modeling sparse corruptions improves latent embeddings. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 737–746, New York, NY, USA, 2017. Association for Computing Machinery.

[6] C. E. Brodley and M. A. Friedl. Identifying and eliminating mislabeled training instances. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*, AAAI'96, page 799–805. AAAI Press, 1996.

[7] Y. Chen, Y. Nadji, A. Kountouras, F. Monrose, R. Perdisci, M. Antonakakis, and N. Vasiloglou. Practical attacks against graph-based clustering. *CoRR*, abs/1708.09056, 2017.

[8] H. Dai, H. Li, T. Tian, X. Huang, L. Wang, J. Zhu, and L. Song. Adversarial attack on graph structured data. *CoRR*, abs/1806.02371, 2018.

[9] Q. Dai, X. Shen, L. Zhang, Q. Li, and D. Wang. Adversarial training methods for network embedding. In *The World Wide Web Conference*, WWW '19, page 329–339, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Z. Deng, Y. Dong, and J. Zhu. Batch virtual adversarial training for graph convolutional networks. *CoRR*, abs/1902.09192, 2019.

[11] F. Feng, X. He, J. Tang, and T. Chua. Graph adversarial training: Dynamically regularizing based on graph structure. *CoRR*, abs/1902.08226, 2019.

[12] J. Fox and S. Rajamanickam. How robust are graph neural networks to structural noise? *ArXiv*, abs/1912.10206, 2019.

[13] C. Giles, K. Bollacker, and S. Lawrence. Citeseer: An automatic citation indexing system. *Proceedings of 3rd ACM Conference on Digital Libraries*, 04 2000.

[14] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 729–734 vol. 2, 2005.

[15] S. Gupta and A. Gupta. Dealing with noise problem in machine learning data-sets: A systematic review. *Procedia Computer Science*, 161:466–474, 01 2019.

[16] W. L. Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: Methods and applications. *CoRR*, abs/1709.05584, 2017.

[17] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, Waltham, MA 02451, USA, 2011.

[18] M. Jin, H. Chang, W. Zhu, and S. Sojoudi. Power up! robust graph convolutional network against evasion attacks based on graph powering. *CoRR*, abs/1905.10029, 2019.

[19] W. Jin, Y. Li, H. Xu, Y. Wang, and J. Tang. Adversarial attacks and defenses on graphs: A review and empirical study. submitted, 03 2020.

[20] E. Kalapanidas, E. Kalapanidas, N. Avouris, M. Craciun, and D. Neagu. Machine learning algorithms: a study on noise. Technical report, in 1st Balcan Conference in Informatics, 2003.

[21] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[22] E. D. Kolaczyk. *Statistical Analysis of Network Data: Methods and Models*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[23] J. Kubica and A. Moore. Probabilistic noise identification and data cleaning. In *Third IEEE International Conference on Data Mining*, pages 131–138, 2003.

[24] V. Labatut and H. Cherifi. Accuracy measures for the comparison of classifiers. *CoRR*, abs/1207.3790, 2012.

[25] V. Levorato. Core decomposition in Directed Networks: Kernelization and Strong Connectivity. In S. I. Publishing, editor, *Complex Networks*, volume 549, pages 129–140, Bologne, Italy, Mar. 2014.

[26] G. Li, M. Müller, A. K. Thabet, and B. Ghanem. Can gcns go as deep as cnns? *CoRR*, abs/1904.03751, 2019.

[27] Q. Li, Z. Han, and X. Wu. Deeper insights into graph convolutional networks for semi-supervised learning. *CoRR*, abs/1801.07606, 2018.

[28] D. Lohani, B. Abdel, and Y. Belaïd. An Invoice Reading System using a Graph Convolutional Network. In *International Workshop on Robust Reading*, PERTH, Australia, Dec. 2018.

[29] A. Mccallum, K. Nigam, and J. Rennie. Automating the construction of internet portals. *Inf. Retr.*, 03 2000.

[30] S. Mei and X. Zhu. Using machine teaching to identify optimal training-set attacks on machine learners. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, page 2871–2877. AAAI Press, 2015.

[31] L. Milli, A. Monreale, G. Rossetti, D. Pedreschi, F. Giannotti, and F. Sebastiani. Quantification in social networks. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–10, 2015.

[32] L. Muñoz-González, B. Biggio, A. Demontis, A. Paudice, V. Wongrassamee, E. C. Lupu, and F. Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. *CoRR*, abs/1708.08689, 2017.

[33] D. Nettleton, A. Orriols-Puig, and A. F. Herrera. A study of the effect of different types of noise on the precision of supervised learning techniques. *Artif. Intell. Rev.*, 33:275–306, 04 2010.

[34] H. NT, C. J. Jin, and T. Murata. Learning graph neural networks with noisy labels. *CoRR*, abs/1905.01591, 2019.

[35] H. NT and T. Maehara. Revisiting graph neural networks: All we have is low-pass filters. submitted, 05 2019.

[36] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652, 2014.

[37] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, Mar. 1986.

[38] D. Rolnick, A. Veit, S. J. Belongie, and N. Shavit. Deep learning is robust to massive label noise. *ArXiv*, abs/1705.10694, 2017.

[39] P. Sen, G. N. Mark, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93–106, 2008.

[40] O. Simeone. A very brief introduction to machine learning with applications to communication systems. *CoRR*, abs/1808.02342, 2018.

[41] A. Sinha, P. Malo, and K. Deb. A review on bilevel optimization: From classical to evolutionary approaches and applications. *IEEE Transactions on Evolutionary Computation*, 22(2):276–295, 2018.

[42] K. Sun, H. Guo, Z. Zhu, and Z. Lin. Virtual adversarial training on graph convolutional networks in node classification. *CoRR*, abs/1902.11045, 2019.

[43] L. Sun, J. Wang, P. S. Yu, and B. Li. Adversarial attack and defense on graph data: A survey. *CoRR*, abs/1812.10528, 2018.

[44] F. J. Valverde-Albacete and C. Peláez-Moreno. 100 % classification accuracy considered harmful: The normalized information transfer factor explains the accuracy paradox. *PloS one*, 9:e84217, 01 2014.

[45] S. Wang, Z. Chen, J. Ni, X. Yu, Z. Li, H. Chen, and P. S. Yu. Adversarial defense framework for graph neural network. *CoRR*, abs/1905.03679, 2019.

[46] M. Waniek, T. Michalak, T. Rahwan, and M. Wooldridge. Hiding individuals and communities in a social network. *Nature Human Behaviour*, 2, 02 2018.

[47] H. Wu, C. Wang, Y. Tyshetskiy, A. Docherty, K. Lu, and L. Zhu. The vulnerabilities of graph convolutional networks: Stronger attacks and defensive techniques. *CoRR*, abs/1903.01610, 2019.

[48] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *CoRR*, abs/1901.00596, 2019.

[49] K. Xu, H. Chen, S. Liu, P. Chen, T. Weng, M. Hong, and X. Lin. Topology attack and defense for graph neural networks: An optimization perspective. *CoRR*, abs/1906.04214, 2019.

[50] X. Zang, Y. Xie, J. Chen, and B. Yuan. Graph universal adversarial attacks: A few bad actors ruin graph learning models. *CoRR*, abs/2002.04784, 2020.

[51] S. Zhang, H. Tong, J. Xu, and R. Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6:1–23, 2019.

[52] Z. Zhang, P. Cui, and W. Zhu. Deep learning on graphs: A survey. *CoRR*, abs/1812.04202, 2018.

[53] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018.

[54] D. Zhu, Z. Zhang, P. Cui, and W. Zhu. Robust graph convolutional networks against adversarial attacks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 1399–1407, New York, NY, USA, 2019. Association for Computing Machinery.

[55] X. Zhu and X. Wu. Class noise vs. attribute noise: A quantitative study of their impacts. *Artificial Intelligence Review*, 22:177–210, 2004.

[56] X. Zhu, X. Wu, and Q. Chen. Eliminating class noise in large datasets. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, page 920–927. AAAI Press, 2003.

[57] D. Zügner and S. Günnemann. Adversarial attacks on graph neural networks via meta learning. *CoRR*, abs/1902.08412, 2019.

[58] D. Zügner and S. Günnemann. Certifiable robustness and robust training for graph convolutional networks. *CoRR*, abs/1906.12269, 2019.

[59] D. Zügner, A. Akbarnejad, and S. Günnemann. Adversarial attacks on neural networks for graph data. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2847–2856, 07 2018.

APÉNDICES

Apéndice 1

Algoritmos de los experimentos realizados

Algorithm 3 Distorsión máxima de aristas

```
1: procedure AristasMaximal(G = (A, X), etiquetas Z, δ = 0,05)
2:     train, test = split(G)
3:
4:     // Definición de valores iniciales
5:     accuracy_original = [ ]
6:     for i in (0..19) do
7:         gcn = GCN.new
8:         gcn.train(G[train])
9:         accuracy = gcn.eval(G[test], Z)
10:        insert(accuracy_original, accuracy)
11:    end for
12:
13:    stdv_original = stdv(accuracy_original)
14:    accuracy_original = average(accuracy_original)
15:
16:    Ĝ ⟵ G
17:    Â ⟵ A
18:    cant_aristas = 0
```

Algorithm 4 Distorsión máxima de aristas (cont.)

```
19:     // Introducir ruido
20:     while true do
21:         // Inicializar θ_0 aleatoriamente
22:         for t in (0..99) do
23:             // Actualización con SGD
24:             θ_{t+1} = step(θ_t, ∇_{θ_t} L_train(f_{θ_t}(Â, X)))
25:         end for
26:
27:         // Computar meta-gradientes
28:         ∇^{meta}_{Â} ⟵ ∇_{Â} L(f_{θ_T}(Â, X))
29:         S ⟵ ∇^{meta}_{Â} ⊙ (−2Â + 1)
30:         e ⟵ argmax(S) // Excluyendo aristas modificadas
31:         Â ⟵ agregar o eliminar arista e de Â
32:         Ĝ ⟵ (Â, X)
33:         cant_aristas += 1
34:
35:         // Evaluar condición de parada
36:         gcn = GCN.new
37:         gcn.train(Ĝ[train])
38:         accuracy = gcn.eval(Ĝ[test], Z)
39:         if accuracy - (accuracy_original - stdv_original) ≤ δ then
40:             avrg_accuracy = [ ]
41:             insert(avrg_accuracy, accuracy)
42:             for i in (0..19) do
43:                 gcn = GCN.new
44:                 gcn.train(Ĝ[train])
45:                 accuracy = gcn.eval(Ĝ[test], Z)
46:                 insert(avrg_accuracy, accuracy)
47:             end for
48:             accuracy = average(avrg_accuracy)
49:         end if
50:         if accuracy_original −δ ≤ accuracy then
51:             break
52:         end if
53:     end while
54:
55:     return cant_aristas
56:
57: end procedure
```

Algorithm 5 Distorsión mínima de aristas

procedure ARISTASMINIMAL($G = (A, X)$, etiquetas Z, $\delta = 0{,}05$)
    train, test = split(G)

    // Definición de valores iniciales
    accuracy_original = [ ]
    for i in (0..19) **do**
        gcn = GCN.new
        gcn.train(G[train])
        accuracy = gcn.eval(G[test], Z)
        insert(accuracy_original, accuracy)
    end for

    stdv_original = stdv(accuracy_original)
    accuracy_original = average(accuracy_original)

    $\hat{G} \longleftarrow G$
    $\hat{A} \longleftarrow A$
    cant_aristas = 0

    // Introducir ruido
    while true **do**
        // Inicializar θ_0 aleatoriamente
        for t in (0..99) **do**
            // Actualización con SGD
            $\theta_{t+1} = step(\theta_t, \nabla_{\theta_t} \mathcal{L}_{train}(f_{\theta_t}(\hat{A}, X)))$
        end for

        // Computar meta-gradientes
        $\nabla^{meta}_{\hat{A}} \longleftarrow \nabla_{\hat{A}} \mathcal{L}(f_{\theta_T}(\hat{A}, X))$
        $S \longleftarrow \nabla^{meta}_{\hat{A}} \odot (-2\hat{A} + 1)$ *// Excluyendo aristas modificadas*
        for e in $bottom_{100}(S)$ **do**
            $\hat{A} \longleftarrow$ agregar o eliminar arista e de $\hat{A}$
        end for
        $\hat{G} \longleftarrow (\hat{A}, X)$
        cant_aristas += 100

Algorithm 6 Distorsión mínima de aristas (cont.)

```
36:         // Evaluar condición de parada
37:         gcn = GCN.new
38:         gcn.train(Ĝ[train])
39:         accuracy = gcn.eval(Ĝ[test], Z)
40:         if accuracy - (accuracy_original - stdv_original) ≤ δ then
41:             avrg_accuracy = [ ]
42:             insert(avrg_accuracy, accuracy)
43:             for i in (0..19) do
44:                 gcn = GCN.new
45:                 gcn.train(Ĝ[train])
46:                 accuracy = gcn.eval(Ĝ[test], Z)
47:                 insert(avrg_accuracy, accuracy)
48:             end for
49:             accuracy = average(avrg_accuracy)
50:         end if
51:         if accuracy_original −δ ≤ accuracy then
52:             break
53:         end if
54:     end while
55:
56:     return cant_aristas
57:
58: end procedure
```

Algorithm 7 Distorsión máxima de atributos

procedure AtributosMaximal($G = (A, X)$, etiquetas Z, $\delta = 0{,}05$)
    train, test = split(G)
    // Definición de valores iniciales
    accuracy_original = [ ]
    for i in (0..19) **do**
        gcn = GCN.new
        gcn.train(G[train])
        accuracy = gcn.eval(G[test], Z)
        insert(accuracy_original, accuracy)
    end for

    stdv_original = stdv(accuracy_original)
    accuracy_original = average(accuracy_original)

    $\hat{G} \longleftarrow G$
    $\hat{X} \longleftarrow X$
    cant_atributos = 0

    // Introducir ruido
    while true **do**
        // Inicializar θ_0 aleatoriamente
        for t in (0..99) **do**
            // Actualización con SGD
            $\theta_{t+1} = step(\theta_t, \nabla_{\theta_t} \mathcal{L}_{train}(f_{\theta_t}(A, \hat{X})))$
        end for

        // Computar meta-gradientes
        $\nabla^{meta}_{\hat{X}} \longleftarrow \nabla_{\hat{X}} \mathcal{L}(f_{\theta_T}(A, \hat{X}))$
        $S \longleftarrow \nabla^{meta}_{\hat{X}} \odot (-2\hat{X} + 1)$
        $f \longleftarrow argmax(S)$ *// Excluyendo atributos modificados*
        $\hat{X} \longleftarrow$ modificar atributo f de $\hat{X}$
        $\hat{G} \longleftarrow (A, \hat{X})$
        cant_atributos += 1

Algorithm 8 Distorsión máxima de atributos (cont.)

```
34:         // Evaluar condición de parada
35:         gcn = GCN.new
36:         gcn.train(Ĝ[train])
37:         accuracy = gcn.eval(Ĝ[test], Z)
38:         if accuracy - (accuracy_original - stdv_original) ≤ δ then
39:             avrg_accuracy = [ ]
40:             insert(avrg_accuracy, accuracy)
41:             for i in (0..19) do
42:                 gcn = GCN.new
43:                 gcn.train(Ĝ[train])
44:                 accuracy = gcn.eval(Ĝ[test], Z)
45:                 insert(avrg_accuracy, accuracy)
46:             end for
47:             accuracy = average(avrg_accuracy)
48:         end if
49:         if accuracy_original −δ ≤ accuracy then
50:             break
51:         end if
52:     end while
53:
54:     return cant_atributos
55:
56: end procedure
```

Algorithm 9 Distorsión mínima de atributos

procedure AtributosMinimal($G = (A, X)$, etiquetas Z, $\delta = 0{,}05$)
    train, test = split(G)

    // Definición de valores iniciales
    accuracy_original = [ ]
    for i in (0..19) **do**
        gcn = GCN.new
        gcn.train(G[train])
        accuracy = gcn.eval(G[test], Z)
        insert(accuracy_original, accuracy)
    end for

    stdv_original = stdv(accuracy_original)
    accuracy_original = average(accuracy_original)

    $\hat{G} \longleftarrow G$
    $\hat{X} \longleftarrow X$
    cant_atributos = 0
    // Introducir ruido
    while true **do**
        // Inicializar θ_0 aleatoriamente
        for t in (0..99) **do**
            // Actualización con SGD
            $\theta_{t+1} = step(\theta_t, \nabla_{\theta_t} \mathcal{L}_{train}(f_{\theta_t}(A, \hat{X})))$
        end for

        // Computar meta-gradientes
        $\nabla^{meta}_{\hat{X}} \longleftarrow \nabla_{\hat{X}} \mathcal{L}(f_{\theta_T}(A, \hat{X}))$
        $S \longleftarrow \nabla^{meta}_{\hat{X}} \odot (-2\hat{X} + 1)$ *// Excluyendo atributos modificados*
        for f in $bottom_{100}(S)$ **do**
            $\hat{X} \longleftarrow$ modificar atributo f de $\hat{X}$
        end for

        $\hat{G} \longleftarrow (A, \hat{X})$
        cant_atributos += 100

Algorithm 10 Distorsión mínima de atributos (cont.)

```
36:         // Evaluar condición de parada
37:         gcn = GCN.new
38:         gcn.train(Ĝ[train])
39:         accuracy = gcn.eval(Ĝ[test], Z)
40:         if accuracy - (accuracy_original - stdv_original) ≤ δ then
41:             avrg_accuracy = [ ]
42:             insert(avrg_accuracy, accuracy)
43:             for i in (0..19) do
44:                 gcn = GCN.new
45:                 gcn.train(Ĝ[train])
46:                 accuracy = gcn.eval(Ĝ[test], Z)
47:                 insert(avrg_accuracy, accuracy)
48:             end for
49:             accuracy = average(avrg_accuracy)
50:         end if
51:         if accuracy_original −δ ≤ accuracy then
52:             break
53:         end if
54:     end while
55:
56:     return cant_atributos
57:
58: end procedure
```

Algorithm 11 Distorsión máxima de aristas y atributos

procedure AristasYAtributosMaximal($G = (A, X)$, etiquetas Z, δ = 0,05)
    train, test = split(G)

    // Definición de valores iniciales
    accuracy_original = [ ]
    for i in (0..19) **do**
        gcn = GCN.new
        gcn.train(G[train])
        accuracy = gcn.eval(G[test], Z)
        insert(accuracy_original, accuracy)
    end for

    stdv_original = stdv(accuracy_original)
    accuracy_original = average(accuracy_original)

    $\hat{G} \longleftarrow G$
    $\hat{A} \longleftarrow A$
    $\hat{X} \longleftarrow X$
    cant_aristas = 0
    cant_atributos = 0
    // Introducir ruido
    while true **do**
        // Inicializar θ_0 aleatoriamente
        for t in (0..99) **do**
            // Actualización con SGD
            $\theta_{t+1} = step(\theta_t, \nabla_{\theta_t} \mathcal{L}_{train}(f_{\theta_t}(\hat{A}, \hat{X})))$
        end for

        // Computar meta-gradientes
        $\nabla^{meta}_{\hat{A}} \longleftarrow \nabla_{\hat{A}} \mathcal{L}(f_{\theta_T}(\hat{A}, \hat{X}))$
        $\nabla^{meta}_{\hat{X}} \longleftarrow \nabla_{\hat{X}} \mathcal{L}(f_{\theta_T}(\hat{A}, \hat{X}))$
        $S \longleftarrow \nabla^{meta}_{\hat{A}} \odot (-2\hat{A} + 1) \bigcup \nabla^{meta}_{\hat{X}} \odot (-2\hat{X} + 1)$ *// Excluyendo aristas y atributos modificados*

        elem $\longleftarrow argmax(S)$
        if *esArista*(*elem*) **then**
            $\hat{A} \longleftarrow$ agregar o eliminar arista *elem* de $\hat{A}$
            cant_aristas += 1
        else
            $\hat{X} \longleftarrow$ modificar atributo *elem* de $\hat{X}$
            cant_atributos += 1
        end if

Algorithm 12 Distorsión máxima de aristas y atributos (cont.)

```
        Ĝ ⟵ (Â, X̂)

        // Evaluar condición de parada
        gcn = GCN.new
        gcn.train(Ĝ[train])
        accuracy = gcn.eval(Ĝ[test], Z)
        if accuracy - (accuracy_original - stdv_original) ≤ δ then
            avrg_accuracy = [ ]
            insert(avrg_accuracy, accuracy)
            for i in (0..19) do
                gcn = GCN.new
                gcn.train(Ĝ[train])
                accuracy = gcn.eval(Ĝ[test], Z)
                insert(avrg_accuracy, accuracy)
            end for
            accuracy = average(avrg_accuracy)
        end if
        if accuracy_original −δ ≤ accuracy then
            break
        end if
    end while

    return cant_aristas, cant_atributos

end procedure
```

Algorithm 13 Distorsión mínima de aristas y atributos

1: **procedure** AristasYAtributosMinimal($G = (A, X)$, etiquetas Z, δ = 0,05)
2: train, test = split(G)
3:
4: *// Definición de valores iniciales*
5: accuracy_original = [ ]
6: **for** i in (0..19) **do**
7: gcn = GCN.new
8: gcn.train(G[train])
9: accuracy = gcn.eval(G[test], Z)
10: insert(accuracy_original, accuracy)
11: **end for**
12:
13: stdv_original = stdv(accuracy_original)
14: accuracy_original = average(accuracy_original)
15:
16: $\hat{G} \longleftarrow G$
17: $\hat{A} \longleftarrow A$
18: $\hat{X} \longleftarrow X$
19: cant_aristas = 0
20: cant_atributos = 0
21: *// Introducir ruido*
22: **while** true **do**
23: *// Inicializar θ_0 aleatoriamente*
24: **for** t in (0..99) **do**
25: *// Actualización con SGD*
26: $\theta_{t+1} = step(\theta_t, \nabla_{\theta_t} \mathcal{L}_{train}(f_{\theta_t}(\hat{A}, \hat{X})))$
27: **end for**
28:
29: *// Computar meta-gradientes*
30: $\nabla^{meta}_{\hat{A}} \longleftarrow \nabla_{\hat{A}} \mathcal{L}(f_{\theta_T}(\hat{A}, \hat{X}))$
31: $\nabla^{meta}_{\hat{X}} \longleftarrow \nabla_{\hat{X}} \mathcal{L}(f_{\theta_T}(\hat{A}, \hat{X}))$
32: $S \longleftarrow \nabla^{meta}_{\hat{A}} \odot (-2\hat{A} + 1) \bigcup \nabla^{meta}_{\hat{X}} \odot (-2\hat{X} + 1)$ *// Excluyendo aristas y atributos modificados*
33:
34: **for** elem in $bottom_{100}(S)$ **do**
35: **if** $esArista(elem)$ **then**
36: $\hat{A} \longleftarrow$ agregar o eliminar arista *elem* de $\hat{A}$
37: cant_aristas += 1
38: **else**
39: $\hat{X} \longleftarrow$ modificar atributo *elem* de $\hat{X}$
40: cant_atributos += 1
41: **end if**
42: **end for**

Algorithm 14 Distorsión mínima de aristas y atributos (cont.)

```
        Ĝ ⟵ (Â, X̂)

        // Evaluar condición de parada
        gcn = GCN.new
        gcn.train(Ĝ[train])
        accuracy = gcn.eval(Ĝ[test], Z)
        if accuracy - (accuracy_original - stdv_original) ≤ δ then
            avrg_accuracy = [ ]
            insert(avrg_accuracy, accuracy)
            for i in (0..19) do
                gcn = GCN.new
                gcn.train(Ĝ[train])
                accuracy = gcn.eval(Ĝ[test], Z)
                insert(avrg_accuracy, accuracy)
            end for
            accuracy = average(avrg_accuracy)
        end if
        if accuracy_original −δ ≤ accuracy then
            break
        end if
    end while

    return cant_aristas, cant_atributos

end procedure
```