La investigación reportada en esta tesis es parte de los programas de investigación del CICESE (Centro de Investigación Científica y de Educación Superior de Ensenada, B.C.).

La investigación fue financiada por el CONAHCYT (Consejo Nacional de Humanidades, Ciencias y Tecnologías).

Centro de Investigación Científica y de Educación Superior de Ensenada, Baja California

Maestría en Ciencias
en Ciencias de la Computación

Una propuesta de agregación de nodos basada en la integral discreta de Choquet para la clasificación de grafos utilizando RNGs

Tesis

para cubrir parcialmente los requisitos necesarios para obtener el grado de

Maestro en Ciencias

Presenta:

Hugo Daniel Sarmiento Rodríguez

Ensenada, Baja California, México

2023

Tesis defendida por

Hugo Daniel Sarmiento Rodríguez

y aprobada por el siguiente Comité

Dr. Carlos Alberto Brizuela Rodríguez

Director de tesis

Dr. César Raúl García Jacas

Dr. Hugo Homero Hidalgo Silva

Dra. María Tereza Cavazos Pérez

Dr. Pedro Gilberto López Mariscal

Coordinador del Posgrado en Ciencias de la Computación

Dra. Ana Denise Re Araujo

Directora de Estudios de Posgrado

Resumen de la tesis que presenta Hugo Daniel Sarmiento Rodríguez como requisito parcial para la obtención del grado de Maestro en Ciencias en Ciencias de la Computación.

Una propuesta de agregación de nodos basada en la integral discreta de Choquet para la clasificación de grafos utilizando RNGs

Resumen aprobado por:

Dr. Carlos Alberto Brizuela Rodríguez
Director de tesis

Los grafos son estructuras matemáticas que se utilizan para modelar redes sociales, sistemas de transporte, sistemas biológicos, fármacos, entre otros. Una tarea relevante en estas áreas es poder, por ejemplo, discriminar un fármaco tóxico de otros que no lo son, distinguir un sistema biológico de otro, un grupo de red social de otro, etcétera. Esta tarea de clasificación se extiende de manera natural a un gran número de situaciones del mundo real que pueden ser modelados como grafos. Las redes neuronales en grafos se están consolidando como la primera opción para estos retos de clasificación. Un elemento crucial en la construcción de clasificadores basados en redes neuronales tiene que ver con la forma en la que se agrega la información contenida en cada nodo del grafo. En este contexto, los modelos de redes neuronales en grafos comúnmente adoptan el promedio como mecanismo de agregación. Este enfoque, aunque combina la información de los nodos, no toma en consideración la posible sinergia que puede existir entre las características de los nodos. Además, carece de un sistema que establezca una jerarquía de prioridad durante el proceso de agregación. En el presente trabajo se propone un nuevo método de agregación basado en la integral discreta de Choquet, el cual permite tener en cuenta la sinergia que podría existir entre las características de los nodos al momento de combinar la información y generar una jerarquía de estos. El algoritmo y el código para su implementación práctica se desarrollaron en PyTorch. Para evaluar el desempeño de este nuevo método de agregación se comparó contra el promedio, una forma de agregación ampliamente usada. Para esto se seleccionaron cinco conjuntos de casos de prueba y los modelos conocidos como GCN, GAT y GIN para comparar el rendimiento de ambos métodos. Los hallazgos sugieren que, si bien es posible integrar la integral discreta de Choquet en una red neuronal y facilitar el aprendizaje automático de los pesos en el método de agregación propuesto, no se observaron diferencias en los resultados en comparación con el uso del promedio como método de agregación.

Palabras clave: Clasificación de grafos, redes neuronales, método de agregación, integral discreta de Choquet, aprendizaje profundo

Abstract of the thesis presented by Hugo Daniel Sarmiento Rodríguez as a partial requirement to obtain the Master of Science degree in Computer Science.

A proposal for node aggregation based on Choquet discrete integral for graph classification using GNNs.

Abstract approved by:

Dr. Carlos Alberto Brizuela Rodríguez
Thesis Director

Graphs are mathematical structures used to model social networks, transport systems, biological systems, drugs, and more. A relevant task in these areas is, for example, to distinguish a toxic drug from those that are not, differentiate one biological system from another, one social network group from another, and so forth. This classification task naturally extends to a large number of real-world situations that can be modeled as graphs. Graph neural networks are emerging as the first choice for these classification challenges. A crucial element in building classifiers based on neural networks has to do with how the information contained in each node of the graph is aggregated. In this context, graph neural network models commonly adopt averaging as an aggregation mechanism. This approach, although combining the information from the nodes, does not take into account the potential synergy that may exist among the node features. Moreover, it lacks a system that establishes a hierarchy of priority during the aggregation process. This study proposes a new aggregation method based on the Choquet discrete integral, which allows taking into account the synergy that could exist among the node features when combining the information and generating a hierarchy of them. The algorithm and code for its practical implementation were developed in PyTorch. To evaluate the performance of this new aggregation method, it was compared against averaging. Five test case sets and known models such as GCN, GAT, and GIN were selected to compare the performance of both methods. The findings suggest that, while it is possible to integrate the Choquet discrete integral into a neural network and facilitate the automatic learning of the weights in the proposed aggregation method, no differences were observed in the results compared to using averaging as an aggregation method.

Keywords: Graph classification, neural networks, aggregation method, Choquet discrete integral, deep learning

Dedicatoria

Dedicado al intrépido zorrito que una vez sintió miedo. Aquel que, con audacia, empuñó su espada para combatir a las sombras. Quien, a pesar de sus fracasos, lo intentó una y otra vez. Continúa hoy, persiste en tu camino. Se valiente zorrito.

Agradecimientos

Al Centro de Investigación Científica y de Educación Superior de Ensenada, Baja California por brindarme la oportunidad y los medios para concluir este proyecto de maestría.

Al Consejo Nacional de Humanidades, Ciencia y Tecnología (CONAHCYT) por brindarme el apoyo económico para realizar mis estudios de maestría por medio de la beca No. 1064349.

Al Dr. Brizuela por sus charlas, paciencia y consejo que hicieron que me sienta orgulloso de este trabajo. A mi comité de tesis por su paciencia y apoyo en todo momento del proyecto.

A mi madre por ser incondicional y cuyo amor me hizo grande. A mi padre quién me aceptó por lo que soy y me ama con orgullo. A mi hermano Kevin cómplice de mis aventuras. A mi hermano Brian por sus abrazos y prestarme su calculadora. A mi hermano Axel porque es increíble. A mi hermana Valentina por ser simplemente ella. A Rosalba por su cariño.

A mi abuelo Jaime por inspirarme a ser un hombre mejor. A mi abuela Gelos quien me enseñó a querer. A mis tíos Jako y Jaime, quienes han sido parte importante de mi vida.

A Karla, porque todo lo imposible fue posible gracias a su apoyo y amor. Gracias desde el fondo de mi corazón, gracias. A Coco, mi gatita, por retumbar con furia cuando tiene hambre.

A mis amigos Axel, Emmanuel, Hector y Leo, quienes desde la distancia me provocaron una sonrisa cuando más la necesitaba.

A mis amigas Sofi, Diana, Vianney, quienes siempre me tienen en sus pensamientos.

A toda la gente de Ensenada, compañeros y amigos que me han recibido con los brazos abiertos. Y sin duda, a toda la gente que no menciono explícitamente, pero que con sus acciones me hicieron llegar lejos. Gracias.

Tabla de contenido

Página

Lista de figuras

Figura Página

Lista de tablas

Capítulo 1. Introducción

1.1. Motivación

Los grafos son estructuras que se utilizan en muchas situaciones cotidianas para representar relaciones entre entidades u objetos (Barabasi, 2013). Estas estructuras matemáticas tienen nodos que representan los objetos o entidades y aristas que muestran las conexiones entre ellos (Cormen et al., 1994). Los grafos son comunes en nuestra vida diaria y se pueden encontrar en diferentes situaciones, por ejemplo en las redes sociales, donde los nodos representan personas y las aristas simbolizan relaciones de amistad o seguidores. También se utilizan en las redes de transporte, donde los nodos son estaciones o paradas y las aristas son rutas entre ellas. Además, se utilizan en redes de comunicación, como Internet, donde los nodos son dispositivos y las aristas son conexiones entre ellos. Las redes biológicas también utilizan grafos, donde los nodos representan proteínas o genes y las aristas representan las interacciones entre ellos.

La clasificación de grafos es un proceso por el cual se asigna una etiqueta o categoría a un grafo en función de sus características y propiedades estructurales. En términos simples, la clasificación de grafos busca identificar y diferenciar grafos según ciertos criterios, como la presencia de subestructuras específicas, la densidad de conexiones o propiedades topológicas particulares. Este problema es fundamental en la ciencia de datos, ya que permite identificar y diferenciar grafos según sus características y propiedades. La clasificación de grafos puede ser esencial para una variedad de tareas y aplicaciones en la vida cotidiana, como:

- Análisis de redes sociales: La clasificación de grafos puede ayudar a identificar comunidades o grupos de usuarios con intereses similares, detectar patrones de comportamiento y predecir tendencias en redes sociales (Kipf & Welling, 2016).
- Detección de fraudes y anomalías: En sistemas financieros y de comunicación, la clasificación de grafos puede ser útil para identificar patrones anómalos o actividades fraudulentas, como transacciones bancarias inusuales o ataques cibernéticos (Cheng et al., 2022).
- Estudio de redes biológicas: La clasificación de grafos puede ayudar a identificar patrones funcionales y estructurales en redes de proteínas, genes y metabolitos, lo que puede contribuir al desarrollo de nuevos tratamientos y terapias para enfermedades (Zitnik & Leskovec, 2017).

- Diseño y optimización de redes de transporte: La clasificación de grafos puede facilitar la identificación de rutas eficientes y la asignación de recursos en sistemas de transporte público y logístico (Li et al., 2017).

Debido a la prevalencia y relevancia de los grafos en nuestra vida diaria, es crucial que se desarrollen y mejoren métodos para clasificarlos. Las redes neuronales en grafos son una técnica importante que puede abordar de manera efectiva los desafíos y aplicaciones en varios campos.

1.2. Antecedentes

Antes de la adopción de las redes neuronales en grafos para la clasificación de estos objetos, se utilizaron diversos enfoques basados en características topológicas y estructurales del grafo. Estos enfoques, en general, se centraban en extraer características descriptivas de los grafos y, posteriormente, aplicar algoritmos de aprendizaje supervisado tradicionales, como máquinas de vectores de soporte (SVM) y árboles de decisión, para la clasificación (Borgwardt & Kriegel, 2005).

Uno de los enfoques más comunes en la clasificación de grafos antes de las GNN fue el uso de kernel de grafos. Los kernel de grafos miden la similitud entre pares de grafos basándose en subestructuras comunes, y luego utilizan estas medidas de similitud como entrada para algoritmos de aprendizaje supervisado, como las SVM (Vishwanathan et al., 2008). Algunos de los kernel de grafos más populares incluyen el kernel de caminos más cortos (Borgwardt & Kriegel, 2005), el kernel de subgrafos (Shervashidze et al., 2009) y el kernel de Weisfeiler-Lehman (Shervashidze et al., 2011).

El desarrollo de las redes neuronales en grafos (GNN) se originó a partir de la necesidad de extender las redes neuronales tradicionales para procesar datos estructurados como grafos. En 2009, Scarselli et al. (2009) introdujeron el concepto de GNN, donde propusieron un marco de aprendizaje profundo para grafos que permitía modelar relaciones de nodos y aprender representaciones de los nodos. En este enfoque, las GNN iterativamente actualizan las representaciones de los nodos utilizando información local de sus vecinos y el estado del nodo previamente calculado, hasta que se alcanza un estado de equilibrio.

Después del trabajo inicial de Scarselli et al. (2009), las GNN se volvieron cada vez más populares, y se exploraron varios enfoques y variaciones de la idea original. Uno de estos enfoques fue la creación de Graph Convolutional Networks (GCN) por Kipf & Welling (2016). Inspirados en las redes neuronales convolucionales (CNN), los autores desarrollaron una generalización de la convolución para grafos que

permite el aprendizaje de representaciones de los nodos.

Posteriormente, se propusieron numerosas variantes y extensiones de las GNN para abordar diferentes problemas y desafíos en el procesamiento de datos en grafos. Por ejemplo, Hamilton et al. (2017) presentaron GraphSAGE, un enfoque inductivo para aprender representaciones de los nodos en grafos grandes y dinámicos. GraphSAGE introduce una técnica de muestreo de vecinos y permite aprender representaciones para nodos no vistos durante el entrenamiento.

Otro avance significativo fue la propuesta de Graph Attention Networks (GAT) (Veličković et al., 2018). GAT introduce mecanismos de atención para ponderar la importancia de los vecinos en la actualización de representaciones de vértices, lo que permite a las GNN aprender representaciones más selectivas y expresivas, según sus creadores.

En 2018, Xu et al. presentaron el Graph Isomorphism Network (GIN), que utiliza un mecanismo de agregación para distinguir grafos no isomorfos (Xu et al., 2018). GIN ha demostrado un rendimiento sobresaliente en varias tareas de clasificación de grafos.

1.3. Planteamiento del problema

En esta sección, nuestra atención se dirige al tema de la clasificación de grafos mediante el uso de redes neuronales en grafos (GNN), y se resalta la relevancia de los métodos de agregación para obtener un vector de características global del grafo, lo que resulta esencial para su clasificación.

La tarea de clasificar grafos consiste en etiquetar todo el grafo de acuerdo a sus características y propiedades estructurales. Para emplear GNN para resolver este problema, se necesita un enfoque de dos pasos: (1) aprender las representaciones de los nodos y (2) agrupar estas representaciones en un vector de características del grafo, el cual se utilizará para la clasificación. Nos concentraremos en el segundo paso y examinaremos los enfoques de agregación más populares, así como algunas estrategias innovadoras.

Una vez que las GNN han aprendido las representaciones de vértices, es necesario combinar esta información en un único vector de características global del grafo. Este vector captura información relevante sobre el grafo en su conjunto y es utilizado por un clasificador para asignar la etiqueta correspondiente.

Entre los métodos de agregación más utilizados para obtener el vector de características global del grafo se encuentran:

- **Promedio o suma de representaciones de los nodos:** Este enfoque simple consiste en calcular

el promedio o la suma de las representaciones de todos los nodos del grafo, lo que proporciona un vector de características global que captura información agregada de todos los vértices (Wu et al., 2021).

- **Máximo o mínimo de las representaciones de los nodos:** Estos métodos aplican operaciones de máximo o mínimo a las representaciones de los nodos (Wu et al., 2021).

Sin embargo, también se han propuesto enfoques más novedosos y sofisticados para obtener el vector de características global del grafo:

- **Redes neuronales auxiliares:** Algunos trabajos proponen el uso de redes de memoria a corto/largo plazo (LSTM), como en Set2Set (Vinyals et al., 2015).
- **Agregación jerárquica:** Otro enfoque es la agregación jerárquica de representaciones de los nodos, como en el método de DiffPool (Ying et al., 2018). Este enfoque agrupa vértices en clústeres y aplica métodos de agregación a nivel de clúster, permitiendo una representación más estructurada y compacta del grafo.

En conclusión, se puede notar la relevancia de los métodos de agregación en la obtención de un vector de características globales del grafo, ya que forman una parte crucial en la clasificación de grafos. Tradicionalmente, el promedio se utiliza como método de agregación, pero presenta limitaciones ya que no captura todas las complejidades e interacciones entre los nodos debido a su invarianza al orden. Aquí es donde la integral discreta de Choquet se vuelve interesante. Esta última es una generalización de varios métodos de agregación, incluyendo el promedio, y se puede utilizar para ranquear nodos, determinando cuáles podrían ser más relevantes en la agregación. Es por ello que nuestro objetivo es investigar y desarrollar un método de agregación basado en la integral discreta de Choquet, para así evaluar esta estrategia de agregación y explorar si se puede lograr un mejor desempeño de las GNN en la clasificación de grafos.

1.4. Objetivos

1.4.1. Objetivo general

El objetivo principal de este trabajo es desarrollar y evaluar un nuevo método de agregación para la clasificación de grafos, basado en la integral discreta de Choquet, dentro del contexto de las redes neuronales en grafos.

1.4.2. Objetivos específicos

- Desarrollar un método de agregación de Choquet basado en la integral discreta de Choquet para su aplicación en la clasificación de grafos en redes neuronales basadas en grafos.
- Seleccionar conjuntos de casos de pruebas sobre los cuales se evaluará la propuesta.
- Seleccionar un conjunto de modelos de aprendizaje en donde se evaluará la propuesta.
- Desarrollar una implementación del método de agregación propuesto en PyTorch Geometric.
- Proponer un protocolo de experimentación que otorgue un marco de comparación justa entre modelos de redes neuronales en grafos (GNN).
- Identificar las diferencias en precisión del método de agregación de Choquet en comparación con el método promedio en la clasificación de grafos, mediante la evaluación de su rendimiento en varios conjuntos de datos de prueba.
- Validar la eficacia del método de agregación de Choquet en comparación con el método promedio mediante un análisis descriptivo y estadístico de los resultados obtenidos.

1.5. Organización de la tesis

La tesis se organiza de la siguiente manera: en el Capítulo 2 se presentan los fundamentos teóricos necesarios para la comprensión del trabajo. El Capítulo 3 se centra en el desarrollo del método de agregación de Choquet y su aplicación en PyTorch. En el Capítulo 4 se describe el protocolo experimental, el conjunto de datos utilizado y la implementación de los modelos de aprendizaje automático. Además, se presentan los resultados obtenidos y se realiza una comparación entre la agregación de Choquet y el promedio, junto con un análisis estadístico de los resultados. Finalmente, el Capítulo 5 presenta las conclusiones de la investigación, discute las limitaciones encontradas y sugiere posibles direcciones para trabajos futuros.

Capítulo 2. Marco Teórico

2.1. Redes neuronales en grafos

En esta sección, proporcionaremos una introducción a conceptos básicos sobre grafos y presentaremos las Redes Neuronales en Grafos (GNN), que combina las técnicas de aprendizaje profundo con la teoría de grafos para analizar datos estructurados en forma de grafos. Explicaremos cómo las GNN pueden ser utilizadas tanto para la clasificación de nodos como para la clasificación de grafos. A lo largo de esta sección, también introduciremos algunos modelos particulares de redes neuronales en grafos, como el Graph Convolutional Network (GCN), el Graph Attention Network (GAT) y el Graph Isomorphism Network (GIN). Por último, la Tabla 1 muestra las notaciones empleadas en esta tesis.

Tabla 1. Tabla de notaciones

Notación	Significado
a	Escalar
$\mathbf{v}$	Vector
$\mathbf{X}$	Matriz
V	Conjunto de nodos
E	Conjunto de aristas
$G = (V, E)$	Grafo
$\mathbf{x}_i$	Atributo del nodo i
$\mathbf{A}_{ij}$	Elemento de la matriz de adyacencia
$\mathbf{D}$	Matriz de grados
$\mathbf{L}$	Laplaciano del grafo
$N(i)$	Vecindario del nodo i
d_i	Grado del nodo i

2.1.1. Introducción a grafos

En esta subsección, nos centraremos en proporcionar una breve introducción a la teoría de grafos. El objetivo es introducir los conceptos esenciales para comprender adecuadamente el tema de las redes neuronales en grafos.

2.1.1.1. Definición y elementos básicos

En términos sencillos, un grafo es una representación visual de un conjunto de objetos y las relaciones entre ellos. Los objetos se representan como puntos llamados nodos o vértices, y las relaciones se

representan como líneas que conectan estos puntos, llamadas aristas o enlaces. Los grafos son útiles para modelar y analizar diferentes tipos de relaciones y estructuras en diversas disciplinas como las ciencias sociales, la biología, la informática y muchas otras. En la Figura 1 se muestran ejemplos de grafos utilizados en diversas disciplinas.

Definición 2.1 *Un grafo G es un par ordenado (V, E), donde V es un conjunto finito no vacío de elementos llamados nodos (o vértices) y E es un conjunto de pares no ordenados de nodos llamados aristas (o enlaces). Es decir, $G = (V, E)$ con $V = \{v_1, v_2, \ldots, v_n\}$ y $E = \{(v_i, v_j) \mid v_i, v_j \in V, i \neq j\}$.*

Figura 1. Ejemplos de grafos para modelar relaciones en diferentes disciplinas: molécula de cafeína, red social, mapa.

2.1.1.2. Tipos de grafos

Siguiendo la definición previa de un grafo, podemos clasificar los grafos en dos categorías principales: grafos dirigidos y no dirigidos. La principal diferencia entre estos dos tipos de grafos radica en la naturaleza de sus aristas. En un grafo no dirigido, las aristas son pares no ordenados de nodos, lo que significa que la conexión entre dos nodos no tiene una dirección específica. En cambio, en un grafo dirigido, las aristas son pares ordenados de nodos, lo que implica que la conexión entre dos nodos tiene una dirección asociada. En este caso, la arista (v_i, v_j) indica una conexión que sale del nodo v_i y llega al nodo v_j. La dirección de las aristas en los grafos dirigidos es esencial para modelar relaciones asimétricas entre nodos, mientras que los grafos no dirigidos son adecuados para representar relaciones simétricas o no direccionales.

Un ejemplo de grafo dirigido es una red de transporte público, donde las aristas representan las rutas entre estaciones y las direcciones indican el sentido en que se realiza el recorrido. Por otro lado, un ejemplo de grafo no dirigido es una red de amigos en una plataforma de redes sociales, donde cada arista representa una relación de amistad entre dos usuarios.

En esta tesis, nos enfocaremos únicamente en el estudio de grafos no dirigidos. Estos grafos son relevantes

en diversas aplicaciones, como la representación de redes sociales, sistemas biológicos y en la solución de problemas de optimización, entre otros. A lo largo del texto, cuando hablemos de "grafos", nos estaremos refiriendo a grafos no dirigidos, a menos que se indique lo contrario.

2.1.1.3. Propiedades de los grafos

En esta subsección, abordaremos algunas de las propiedades más importantes y relevantes de los grafos. A continuación, se presentan algunas de ellas junto con sus definiciones formales:

- **Grado de un nodo**: El grado de un nodo v en un grafo no dirigido $G = (V, E)$, denotado como $\deg(v)$, es el número de aristas que inciden en v. Formalmente, $\deg(v) = |\{e \in E \mid v \in e\}|$. En el caso de un grafo dirigido, se diferencian entre grado de entrada y grado de salida. El grado de entrada de un nodo v, denotado como $\deg^-(v)$, es el número de aristas que llegan a v, mientras que el grado de salida, denotado como $\deg^+(v)$, es el número de aristas que parten de v.

- **Caminos y ciclos**: Un camino en un grafo no dirigido $G = (V, E)$ es una secuencia alternada de nodos y aristas de la forma $v_0, e_1, v_1, e_2, \ldots, e_k, v_k$, donde cada arista e_i conecta los nodos v_{i-1} y v_i, y todos los nodos y aristas son distintos. Un ciclo es un camino que comienza y termina en el mismo nodo, es decir, $v_0 = v_k$. En el caso de grafos dirigidos, un camino dirigido es similar, pero las aristas deben respetar su dirección, y un ciclo dirigido es un camino dirigido que comienza y termina en el mismo nodo.

- **Conexión**: Un grafo no dirigido $G = (V, E)$ se dice conexo si existe un camino entre cada par de nodos en V. En el caso de un grafo dirigido, se dice fuertemente conexo si existe un camino dirigido entre cada par de nodos. Un grafo dirigido es débilmente conexo si su versión no dirigida es conexa.

- **Distancia y diámetro**: La distancia entre dos nodos v_i y v_j en un grafo G, denotada como $d(v_i, v_j)$, es la longitud del camino más corto que conecta a ambos nodos. La longitud de un camino se mide en términos del número de aristas que lo componen. El diámetro de un grafo G, denotado como diam(G), es la distancia máxima entre cualquier par de nodos en G, es decir, diam$(G) = \max\{\ d(v_i, v_j) \mid v_i, v_j \in V\}$.

- **Subgrafos**: Un subgrafo H de un grafo $G = (V, E)$ es un grafo que se obtiene al eliminar nodos y/o aristas de G. Formalmente, $H = (V', E')$ es un subgrafo de G si $V' \subseteq V$ y $E' \subseteq E$.

- **Isomorfismo de grafos**: Dos grafos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ son isomorfos si existe una biyección $\phi : V_1 \to V_2$ tal que para todo par de nodos $v_i, v_j \in V_1$, se tiene que $(v_i, v_j) \in E_1$ si y solo si $(\phi(v_i), \phi(v_j)) \in E_2$. Es decir, los grafos isomorfos tienen la misma estructura, pero posiblemente con diferentes etiquetas en los nodos.

Estas propiedades y conceptos son fundamentales para el estudio de grafos y serán útiles a lo largo de esta tesis. Al analizar las redes neuronales en grafos, es importante tener en cuenta estas propiedades, ya que pueden proporcionar información valiosa sobre la estructura y el comportamiento de las redes estudiadas.

2.1.1.4. Representación de grafos

Existen diferentes formas de representar grafos computacionalmente. Dos de las representaciones más comunes son las matrices de adyacencia y el formato COO (Coordinate List). A continuación, se describen ambas representaciones y se muestra un ejemplo de un grafo representado en cada formato.

Definición 2.2 *Una* **matriz de adyacencia** *es una matriz cuadrada A de tamaño $n \times n$, donde n es el número de nodos en el grafo. El elemento a_{ij} de la matriz es 1 si existe una arista entre los nodos v_i y v_j, y 0 en caso contrario. Para grafos dirigidos, el elemento a_{ij} es 1 si existe un arista desde el nodo v_i hacia el nodo v_j. Las matrices de adyacencia de grafos no dirigidos son simétricas.*

Definición 2.3 *El* **formato COO** *es una representación basada en listas que almacena solo las aristas existentes en el grafo. Consiste en dos listas de igual longitud: una lista de nodos de origen y una lista de nodos de destino. Cada elemento en la posición k de ambas listas representa una arista que conecta el nodo de origen con el nodo de destino. Este formato es especialmente útil para grafos dispersos, ya que solo requiere almacenar las conexiones existentes, ahorrando espacio en memoria.*

El formato COO también se puede representar como una única matriz de dos filas y m columnas, donde m es el número de aristas (o arcos) en el grafo. La primera fila contiene los nodos de origen y la segunda fila contiene los nodos de destino. Esta representación es equivalente a las dos listas separadas, pero apiladas en una única matriz.

Ejemplo 2.1 *Considere el siguiente grafo no dirigido de la Figura 2 el cual tiene 5 nodos y 6 aristas:*

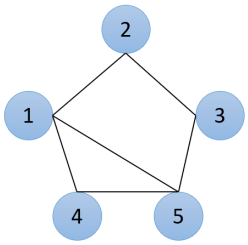

Figura 2. Ejemplo de grafo.

La matriz de adyacencia de este grafo es:

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{1}$$

La representación en formato COO apilado como una matriz de dos filas y 5 columnas es:

$$\begin{bmatrix} 1 & 1 & 1 & 2 & 3 & 4 \\ 2 & 4 & 5 & 3 & 5 & 5 \end{bmatrix} \tag{2}$$

Computacionalmente, el formato COO es más eficiente que las matrices de adyacencia para grafos dispersos, ya que requiere menos espacio en memoria al almacenar solo las conexiones existentes. Además, muchas operaciones sobre grafos pueden realizarse de manera eficiente utilizando el formato COO. Dado que en la mayoría de las aplicaciones prácticas, los grafos suelen ser dispersos, en esta tesis se utilizará el formato COO para representar grafos de manera computacional.

2.1.1.5. Matriz de características asociadas a un grafo

En muchos problemas del mundo real, los nodos de un grafo pueden tener asociadas características adicionales que describen propiedades o atributos de esos nodos. Estas características pueden ser representadas como un vector de características $\mathbf{x}$. Por ejemplo, en un grafo que representa una red social,

los nodos pueden ser personas y el vector de características podría incluir información como la edad, género, intereses, ocupación, entre otros.

Para trabajar con las características de los nodos en un grafo, podemos definir una matriz de características X asociada al grafo, que consiste en apilar todos los vectores de características de los nodos por renglones. De esta manera, si el grafo tiene n nodos y cada nodo tiene un vector de características de dimensión d, entonces la matriz de características X será de tamaño $n \times d$. Es importante observar que el número de nodos en el grafo corresponde al número de renglones en la matriz de características X.

Formalmente, la matriz de características X se define como:

$$X = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{bmatrix} \tag{3}$$

donde $\mathbf{x}_i^T$ representa la transposición del vector de características del nodo i y cada fila de la matriz X corresponde a las características de un nodo en el grafo.

De manera análoga se puede asociar una matriz de características a las aristas de un grafo. Pero esto va mas allá del alcance de este trabajo.

2.1.2. Redes neuronales en grafos

2.1.2.1. Motivación y aplicaciones

Las Redes Neuronales en Grafos (GNN) son una clase de modelos de aprendizaje profundo diseñados específicamente para operar con datos estructurados en forma de grafos. Las GNN han ganado popularidad en los últimos años debido a su capacidad para capturar las dependencias y relaciones complejas presentes en los grafos, lo que las hace adecuadas para una amplia gama de aplicaciones en diversos campos, como la química (Gilmer et al., 2017), la biología (Zitnik & Leskovec, 2017), la teoría de redes sociales (Kipf & Welling, 2016), el análisis de sistemas de transporte (Li et al., 2017), y la clasificación de documentos (Yang et al., 2016), entre otros.

2.1.2.2. Mecanismos de funcionamiento de las GNN

Las GNN procesan grafos a través de capas iterativas que aprenden representaciones vectoriales de los nodos, aristas y posiblemente del grafo completo. Estas representaciones se utilizan en tareas de aprendizaje, tanto supervisadas como no supervisadas, como la clasificación de nodos, predicción de enlaces, y clasificación de grafos (Wu et al., 2021). Un aspecto destacado de las GNN es su invariancia a las permutaciones de nodos, lo que les permite adaptarse a diferentes estructuras de grafos, capturando información contextual local y global (Battaglia et al., 2018).

2.1.2.3. Proceso de paso de mensajes en las GNN

El *paso de mensajes* es un mecanismo crucial en las GNN que describe cómo se propaga la información entre los nodos de un grafo. Este procedimiento actualiza iterativamente la representación de los nodos al intercambiar y combinar información de sus nodos vecinos.

Este proceso, ejecutado a través de las capas de la GNN, busca que cada nodo integre información de su entorno local y finalmente aprenda una representación que refleje sus propias características y las de su vecindario. Las GNN se fundamentan en dos operaciones esenciales: agregación y actualización. En cada iteración, se realizan los siguientes pasos:

- **AGREGACIÓN:** Cada nodo recopila información (mensajes) de sus vecinos. Esta información puede ser la representación actual de los nodos vecinos, que se combina mediante una función de agregación (como suma, promedio, máximo, etc.).
- **ACTUALIZACIÓN:** Cada nodo actualiza su representación utilizando la información agregada de sus vecinos y su propia representación. Esto puede involucrar la aplicación de una función de actualización, que generalmente incluye una transformación lineal (mediante una matriz de pesos aprendida) seguida de una función de activación no lineal.

El punto de partida de este proceso es el vector de características inicial, denotado como $\mathbf{x}_v = \mathbf{h}_v^0$, asociado a un nodo v del grafo G. En cada capa de la red, se llevan a las operaciones de agregación y actualización. Luego, las redes neuronales basadas en grafos se pueden describir matemáticamente a través de las siguientes ecuaciones:

$$
\begin{aligned}
&\text{Inicializamos } \mathbf{h}_v^0 = \mathbf{x}_v \\
&\text{Para cada } k = 1, 2, ..., K, \\
&\quad \mathbf{a}_v^k = \textbf{AGREGAR}^k\{\mathbf{h}_u^{k-1} : u \in N(v)\} \\
&\quad \mathbf{h}_v^k = \textbf{ACTUALIZAR}^k\{\mathbf{h}_v^{k-1}, \mathbf{a}_v^k\}
\end{aligned}
\tag{4}
$$

En las ecuaciones anteriores, $N(v)$ representa la vecindad del nodo v y K indica el número de capas en la GNN. Al final del proceso, la representación $\mathbf{h}_v^K$ es el resultado final para el nodo v, tras su paso por todas las capas de la red.

Es importante observar que, a medida que la información se propaga a través de las sucesivas capas, se va agregando y refinando información de vecindarios más amplios. Sin embargo, una vez que la cantidad de capas de la GNN alcanza el diámetro del grafo, es decir, la distancia máxima más corta entre cualquier par de nodos en el grafo, la información de todos los nodos y sus respectivos vecindarios ya ha sido incorporada en la representación final de cada nodo. En este punto, agregar más capas no aportaría nueva información. Por lo tanto, en general, no es necesario que una GNN tenga más capas que el diámetro del grafo para capturar de manera efectiva las relaciones y dependencias presentes en la estructura del grafo.

En resumen, las GNN aprenden representaciones jerárquicas de los nodos, integrando información local y de vecindad en diferentes escalas. Además, las funciones de agregación y actualización pueden adaptarse a distintos tipos de grafos y tareas de aprendizaje, lo que convierte a las GNN en herramientas versátiles para una amplia gama de problemas y dominios.

2.1.2.4. Modelos de GNN

Existen diversos modelos de GNN que difieren en la forma en que implementan las operaciones de agregación y actualización. Algunos ejemplos de GNN son el Graph Convolutional Network (GCN) (Kipf & Welling, 2016), GraphSAGE (Hamilton et al., 2017), Graph Attention Network (GAT) (Veličković et al., 2018) y el Message Passing Neural Network (MPNN) (Gilmer et al., 2017). Cada uno de estos modelos presenta enfoques diferentes y puede ser más adecuado para distintos tipos de aplicaciones y conjuntos de datos. En las subsecciones siguientes se describen algunos de estos modelos de GNN.

2.1.3. Clasificación de nodos usando GNN

2.1.3.1. Concepto general

Una de las aplicaciones más comunes de las redes neuronales en grafos (GNN) es la clasificación de nodos. Esta tarea consiste en asignar etiquetas a los nodos de un grafo basándose en sus características y las de su vecindad. En el contexto de las GNN, la clasificación de nodos es especialmente relevante, ya que estos modelos son capaces de capturar las dependencias y relaciones complejas presentes en los grafos.

El proceso comienza con una representación inicial de los nodos, que puede ser un vector de características basado en información específica del nodo. A través de las operaciones de agregación y actualización, esta representación se refina iterativamente a lo largo de las capas de la red. Después de K iteraciones, cada nodo v tiene una representación final $\mathbf{h}_v^K$, que encapsula información sobre el nodo y su vecindad.

La representación final $\mathbf{h}_v^K$ se puede utilizar como entrada para un clasificador, que asignará una etiqueta a cada nodo. Este clasificador puede ser, por ejemplo, una capa de salida de tipo softmax en el caso de una clasificación multiclase, o una capa sigmoide para una clasificación binaria. La elección del clasificador dependerá del problema específico que se esté abordando.

Para entrenar el modelo, se necesita un conjunto de nodos etiquetados que se pueden utilizar como datos de entrenamiento. El objetivo es minimizar la diferencia entre las etiquetas predichas por el modelo y las etiquetas reales. Este proceso de aprendizaje supervisado permite a las GNN ajustar sus parámetros para mejorar la precisión de la clasificación.

2.1.3.2. Proceso de clasificación de nodos

Una representación matemática del proceso de clasificación de nodos en una Red Neuronal en Grafos (GNN) puede ser formulada de la siguiente manera:

1. **Inicialización:** Cada nodo v en el grafo G tiene una representación inicial basada en sus atributos, denotada como $\mathbf{h}_v^0 = \mathbf{x}_v$.

2. **Procesamiento en la GNN:** A través de K capas de la GNN, cada nodo v actualiza su representación utilizando las operaciones de agregación y actualización. La representación final del nodo v

después de K capas se denota como $\mathbf{h}_v^K$.

$$\begin{aligned} &\text{Para cada } k = 1, 2, ..., K, \\ &\quad \mathbf{a}_v^k = \textbf{AGREGAR}^k\{\mathbf{h}_u^{k-1} : u \in N(v)\} \\ &\quad \mathbf{h}_v^k = \textbf{ACTUALIZAR}^k\{\mathbf{h}_v^{k-1}, \mathbf{a}_v^k\} \end{aligned} \tag{5}$$

3. **Clasificación de nodos:** La representación final $\mathbf{h}_v^K$ se utiliza como entrada para un clasificador f_θ, que asigna una etiqueta y_v a cada nodo v. En el caso de una clasificación multiclase, el clasificador puede ser una capa de salida de tipo softmax (Wang et al., 2020):

$$y_v = f_\theta(\mathbf{h}_v^K) = \text{softmax}(\mathbf{W}\mathbf{h}_v^K + \mathbf{b}) \tag{6}$$

donde $\mathbf{W}$ y $\mathbf{b}$ son los parámetros del clasificador, y θ representa todos los parámetros del modelo, incluyendo los de la GNN y el clasificador.

4. **Aprendizaje supervisado:** El modelo se entrena minimizando una función de pérdida L, que mide el error entre las etiquetas predichas por el modelo y_v y las etiquetas reales y_v^*:

$$L = \sum_{v \in V} \text{Loss}(y_v, y_v^*) \tag{7}$$

donde $\text{Loss}(y_v, y_v^*)$ es la función de pérdida para el nodo v, que podría ser, por ejemplo, la pérdida de entropía cruzada (Wang et al., 2020) en el caso de una clasificación multiclase.

2.1.4. Clasificación de grafos

2.1.4.1. Concepto general

Otra tarea importante en el análisis de grafos donde las GNN juegan un papel crucial es la clasificación de grafos. A diferencia de la clasificación de nodos, que busca asignar etiquetas a los nodos individuales de un grafo, en la clasificación de grafos se propone asignar una etiqueta a un grafo completo, considerando su estructura global y las interacciones entre sus nodos.

En la clasificación de grafos, cada grafo en el conjunto de datos se considera una sola instancia y se le asigna una etiqueta. La tarea es aprender un modelo que pueda predecir correctamente la etiqueta de

un grafo dado. Al igual que en la clasificación de nodos, las representaciones de los nodos obtenidas a través de las GNN desempeñan un papel crucial en este proceso.

Para clasificar un grafo, se comienza obteniendo representaciones finales para cada uno de sus nodos mediante la aplicación sucesiva de las operaciones de agregación y actualización a lo largo de las capas de la red, tal cual como se hace para la clasificación de nodos. Sin embargo, a diferencia de la clasificación de nodos, se necesita una representación para el grafo completo, no solo para los nodos individuales. Por tanto, se emplea un método de agregación que combine las representaciones de los nodos en una representación única del grafo. Esta operación puede ser, por ejemplo, un promedio o una suma de las representaciones finales de los nodos.

La representación resultante del grafo se puede utilizar luego como entrada para un clasificador, que asignará una etiqueta a cada grafo. Este clasificador puede ser, por ejemplo, una capa de salida de tipo softmax para la clasificación multiclase, o una capa sigmoide para la clasificación binaria.

Para entrenar el modelo, se necesita un conjunto de grafos etiquetados que se pueden utilizar como datos de entrenamiento. Al igual que en la clasificación de nodos, el objetivo es minimizar la diferencia entre las etiquetas predichas por el modelo y las etiquetas reales. Este proceso de aprendizaje supervisado permite a las GNN ajustar sus parámetros para mejorar la precisión de la clasificación.

2.1.4.2. Proceso de clasificación de grafos

El proceso para la clasificación de grafos utilizando las representaciones finales de los nodos en una GNN consta de los siguientes pasos:

1. **Inicialización:** Cada nodo v en el grafo G tiene una representación inicial basada en sus atributos, denotada como $\mathbf{h}_v^0 = \mathbf{x}_v$.

2. **Procesamiento en la GNN:** A través de K capas de la GNN, cada nodo v actualiza su representación utilizando las operaciones de agregación y actualización. La representación final del nodo v después de K capas se denota como $\mathbf{h}_v^K$.

3. **Agregación de las representaciones de nodos:** Para obtener una representación del grafo completo, se realiza una operación de agregación sobre las representaciones finales de todos los nodos en el grafo. La representación del grafo G se denota como $\mathbf{h}_G$, y puede obtenerse mediante diversas

estrategias de agregación, como la suma, el promedio o el máximo de las representaciones de los nodos:

$$\mathbf{h}_G = \textbf{AGREGACIÓN DEL GRAFO}(\{\mathbf{h}_v^K : v \in V\}) \tag{8}$$

4. **Clasificación de grafos:** La representación del grafo $\mathbf{h}_G$ se utiliza como entrada para un clasificador g_ϕ, que asigna una etiqueta y_G al grafo completo. Similar a la clasificación de nodos, en el caso de una clasificación multiclase, el clasificador puede ser una capa de salida de tipo softmax:

$$y_G = g_\phi(\mathbf{h}_G) = \text{softmax}(\mathbf{W}_G\mathbf{h}_G + \mathbf{b}_G) \tag{9}$$

donde $\mathbf{W}_G$ y $\mathbf{b}_G$ son los parámetros del clasificador, y ϕ representa todos los parámetros del modelo, incluyendo los de la GNN y el clasificador.

5. **Aprendizaje supervisado:** El modelo se entrena minimizando una función de pérdida L_G, que mide la discrepancia entre las etiquetas predichas por el modelo y_G y las etiquetas reales y_G^*:

$$L_G = \sum_{G \in \mathcal{G}} \text{Loss}(y_G, y_G^*) \tag{10}$$

donde $\mathcal{G}$ es el conjunto de grafos y $\text{Loss}(y_G, y_G^*)$ es la función de pérdida para el grafo G, que podría ser, por ejemplo, la pérdida de entropía cruzada en el caso de una clasificación multiclase.

Este enfoque permite a las GNN modelar y aprender patrones a nivel de grafo, lo cual es útil para tareas en las que se requiere capturar relaciones y dependencias estructurales a gran escala en los datos. Al mismo tiempo, mantiene las ventajas de las GNN en términos de invariancia a las permutaciones de los nodos y la capacidad de manejar grafos de diferentes tamaños y estructuras.

2.1.5. Modelos específicos de GNN

2.1.5.1. Graph convolutional network

El modelo Graph Convolutional Network (GCN), propuesto por Kipf & Welling (2016), es uno de los modelos más representativos de las GNN. La idea principal de las GCN es aprender una representación

de los nodos que capture tanto sus características individuales como las de su vecindad, mediante la aplicación de convoluciones en el dominio del grafo.

La operación de paso de mensaje en las GCN se realiza de la siguiente manera. Para la agregación, se pondera la representación de cada nodo vecino por el inverso de la raíz cuadrada del producto de los grados de los nodos involucrados. Luego, para la actualización, se aplica una transformación lineal seguida de una función de activación no lineal. Matemáticamente, estas operaciones se pueden representar como sigue:

$$\begin{aligned} &\text{Para cada } k = 1, 2, ..., K, \\ &\mathbf{a}_v^k = \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(k-1)}}{\sqrt{deg(v)deg(u)}} \\ &\mathbf{h}_v^k = \sigma\left(\mathbf{W}^k \mathbf{a}_v^k\right) \end{aligned} \tag{11}$$

En la ecuación, $\mathbf{a}_v^k$ denota la agregación de las representaciones de los vecinos del nodo v, donde el término $\frac{1}{\sqrt{deg(v)deg(u)}}$ actúa como un factor de normalización. La ponderación de las representaciones de los nodos vecinos con este factor de normalización ayuda a mantener la estabilidad numérica y mejora el rendimiento de la GCN.

También es importante mencionar que, $\mathbf{W}^k$ es una matriz de pesos que se aprende durante el entrenamiento, y σ es una función de activación no lineal, como la función ReLU (Nwankpa et al., 2018).

Los GCN no requieren hiperparámetros específicos, salvo la cantidad de capas K que se utilizarán en la red.

2.1.5.2. Graph attention network

El modelo Graph Attention Network (GAT), introducido por Veličković et al. (2018), es un modelo de GNN que introduce mecanismos de atención para ponderar las contribuciones de los vecinos en el paso de mensajes.

La idea central de las GAT es que no todos los vecinos son igualmente importantes para la representación de un nodo. Por lo tanto, las GAT asignan un peso a cada vecino en función de su relevancia para el nodo.

La operación de paso de mensajes en las GAT se realiza de la siguiente manera. Para la agregación, se calcula una suma ponderada de las representaciones de los nodos vecinos, donde los pesos se determinan mediante un mecanismo de atención. Para la actualización, se aplica una transformación lineal a la representación del nodo. Matemáticamente, estas operaciones se pueden representar como sigue:

$$\begin{aligned} &\text{Para cada } k = 1, 2, ..., K, \\ &\mathbf{a}_v^k = \sum_{u \in N(v)} \alpha_{vu}^k \mathbf{h}_u^{k-1} \\ &\mathbf{h}_v^k = \sigma\left(\mathbf{W}^k \mathbf{a}_v^k\right) \end{aligned} \tag{12}$$

Aquí, α_{vu}^k es el peso asignado al vecino u en la capa k, que se calcula mediante un mecanismo de atención basado en una red neuronal hacia adelante:

$$\alpha_{vu}^k = \frac{\exp\left(\text{LeakyReLU}\left(\mathbf{a}^T[\mathbf{W}^k \mathbf{h}_v^{k-1} || \mathbf{W}^k \mathbf{h}_u^{k-1}]\right)\right)}{\sum_{u' \in N(v)} \exp\left(\text{LeakyReLU}\left(\mathbf{a}^T[\mathbf{W}^k \mathbf{h}_v^{k-1} || \mathbf{W}^k \mathbf{h}_{u'}^{k-1}]\right)\right)} \tag{13}$$

En la ecuación, los corchetes con la doble barra $[\cdot||\cdot]$ denotan la concatenación de dos vectores, es decir, se unen los elementos de los dos vectores en un solo vector. Por ejemplo, si $\mathbf{a} = [1, 2]$ y $\mathbf{b} = [3, 4]$, entonces $[\mathbf{a}||\mathbf{b}] = [1, 2, 3, 4]$.

Las GAT tienen varios hiperparámetros que deben ser ajustados, incluyendo el número de capas K, la dimensión de las representaciones de los nodos, y los parámetros del mecanismo de atención, como la dimensión de la capa de atención y la tasa de caída (dropout rate).

2.1.5.3. Graph isomorphism network

El modelo Graph Isomorphism Network (GIN), propuesto por Xu et al. (2018), es un modelo de GNN que fue diseñada para ser capaz de distinguir entre grafos que no son isomorfos. Las GIN logran esto a través de un esquema de paso de mensajes que permite capturar patrones de interacción complejos.

La operación de paso de mensajes en las GIN consiste en dos pasos. Primero, en la etapa de agregación, se suman las representaciones de los nodos vecinos. Segundo, en la etapa de actualización, se aplica una transformación lineal a la suma de la representación del nodo y la representación agregada de los

vecinos. Matemáticamente, estas operaciones se pueden representar de la siguiente manera:

$$\begin{aligned} &\text{Para cada } k = 1, 2, ..., K, \\ &\mathbf{a}_v^k = \sum_{u \in N(v)} \mathbf{h}_u^{k-1} \\ &\mathbf{h}_v^k = \sigma\left(\mathbf{W}^k(\mathbf{h}_v^{k-1} + (1+\epsilon)\mathbf{a}_v^k)\right) \end{aligned} \tag{14}$$

Aquí, ϵ es un hiperparámetro que controla la importancia relativa de la representación del nodo y la representación agregada de los vecinos. Al permitir una interacción no trivial entre la representación del nodo y la representación agregada de los vecinos, las GIN pueden capturar patrones de interacción más complejos que otros modelos de GNN.

Las GIN tienen varios hiperparámetros que deben ser ajustados, incluyendo el número de capas K, la dimensión de las representaciones de los nodos, y el parámetro ϵ.

2.2. Funciones de agregación

Las funciones de agregación juegan un papel fundamental en las redes neuronales en grafos. Estas funciones permiten combinar información de múltiples nodos, generalmente los vecinos inmediatos de un nodo dado, para generar una representación más rica y contextual del nodo (Wu et al., 2021). La elección de la función de agregación puede tener un impacto significativo en la capacidad de la GNN para capturar diferentes tipos de estructuras y patrones en los datos del grafo.

La agregación en las GNN es una operación que suele ser invariante al orden, es decir, los resultados de la agregación no se ven afectados por el orden en que se procesan los nodos (Morris et al., 2019). Esto es importante ya que, en general, los grafos no tienen un orden inherente de sus nodos.

Diferentes funciones de agregación pueden capturar diferentes tipos de relaciones y dependencias entre los nodos. Por ejemplo, la agregación de media aritmética y suma permiten capturar información global de los vecinos de un nodo, mientras que las funciones de agregación basadas en atención, como en GAT, permiten a la red aprender a poner diferentes pesos a diferentes nodos en función de su relevancia para la tarea de aprendizaje (Veličković et al., 2018).

En las siguientes subsecciones, exploraremos en detalle diferentes tipos de funciones de agregación y

discutiremos su aplicabilidad en la clasificación de grafos.

2.2.1. Funciones de agregación básicas

Las funciones de agregación son operaciones que toman un conjunto de valores y producen un solo valor resumen. Las funciones de agregación básicas incluyen operaciones como suma, promedio, mínimo y máximo. Aquí, presentamos formalmente estas funciones básicas.

Definición 2.4 *La función de agregación suma toma un conjunto de valores $a_1, a_2, \ldots, a_n$ y devuelve la suma de estos valores:*

$$\textit{Suma}(a_1, a_2, \ldots, a_n) = a_1 + a_2 + \ldots + a_n. \tag{15}$$

Definición 2.5 *La función de agregación promedio toma un conjunto de valores $a_1, a_2, \ldots, a_n$ y devuelve el promedio de estos valores:*

$$\textit{Promedio}(a_1, a_2, \ldots, a_n) = \frac{a_1 + a_2 + \ldots + a_n}{n}. \tag{16}$$

Definición 2.6 *La función de agregación mínimo toma un conjunto de valores $a_1, a_2, \ldots, a_n$ y devuelve el valor mínimo:*

$$\textit{Mínimo}(a_1, a_2, \ldots, a_n) = \text{mín}(a_1, a_2, \ldots, a_n). \tag{17}$$

Definición 2.7 *La función de agregación máximo toma un conjunto de valores $a_1, a_2, \ldots, a_n$ y devuelve el valor máximo:*

$$\textit{Máximo}(a_1, a_2, \ldots, a_n) = \text{máx}(a_1, a_2, \ldots, a_n). \tag{18}$$

2.2.2. Definición general de función de agregación

Las funciones de agregación básicas que hemos presentado hasta ahora son solo algunos ejemplos comunes y sencillos de este tipo de operaciones. Sin embargo, el concepto de una función de agregación es mucho más general. Para profundizar en su comprensión, es útil establecer una definición formal de lo que constituye una función de agregación.

Definición 2.8 *Una* **función de agregación** *es una función que opera sobre un conjunto de valores y produce un solo valor resumen. Formalmente, si consideramos un intervalo $\mathbb{I} = [a, b]$, una función de agregación f es una función que toma $n > 1$ argumentos de este intervalo y devuelve un valor en el mismo intervalo, es decir, $f : \mathbb{I}^n \to \mathbb{I}$. Una función de agregación tiene las siguientes propiedades:*

1. *Identidad: $f(\underbrace{a, a, ..., a}_{n \text{ veces}}) = a$ y $f(\underbrace{b, b, ..., b}_{n \text{ veces}}) = b$.*
2. *Monotonía: para cada par de vectores $\mathbf{x}, \mathbf{y} \in \mathbb{I}^n$, si $x_i \leq y_i$, para cada $i \in \{1, ..., n\}$, entonces $f(\mathbf{x}) \leq f(\mathbf{y})$.*

La primera propiedad, la identidad, indica que si todos los argumentos son iguales, la función de agregación simplemente devuelve ese valor común. La segunda propiedad, la monotonía, asegura que si cada componente de un vector $\mathbf{x}$ es menor o igual que la componente correspondiente de un vector $\mathbf{y}$, entonces el resultado de aplicar la función de agregación a $\mathbf{x}$ no será mayor que el resultado de aplicarla a $\mathbf{y}$.

2.2.3. Promedio ponderado

Una instancia particular de las funciones de agregación que tiene relevancia tanto en teoría como en aplicaciones prácticas es el *promedio ponderado*. Esta función de agregación toma en cuenta tanto los valores que se están agregando como su respectiva importancia o peso.

Definición 2.9 *El* **promedio ponderado** *de un conjunto de valores $x_1, x_2, ..., x_n$ con pesos correspondientes $w_1, w_2, ..., w_n$ (donde cada $w_i \geq 0$) es definido como:*

$$f(x_1, x_2, ..., x_n) = \frac{w_1 x_1 + w_2 x_2 + \cdots + w_n x_n}{w_1 + w_2 + \cdots + w_n} \tag{19}$$

La condición $\sum_{i=1}^{n} w_i = 1$ es usualmente impuesta para que los pesos sumen a uno.

El promedio ponderado es una forma de calcular un tipo de *centro* o *promedio* de los datos que toma en cuenta cuánto importa cada valor. Los valores con un peso mayor contribuirán más al resultado final.

El promedio ponderado es un caso especial de las *funciones de agregación de tipo OWA* (Ordered Weighted Averaging) (Beliakov et al., 2016). En el caso del promedio ponderado, los pesos están asignados a

priori a cada valor. En cambio, en las funciones OWA, los pesos se asignan a los valores de acuerdo a su orden en la secuencia de entrada. Esto permite modelar situaciones donde el orden de los valores es relevante.

2.2.4. Funciones de agregación de tipo ordenado ponderado (OWA)

Las *funciones de agregación de tipo ordenado ponderado (OWA)* representan una generalización de las funciones de agregación básicas que hemos discutido anteriormente, incluyendo el mínimo, el máximo y el promedio ponderado. Este tipo de funciones de agregación se caracterizan por su capacidad para considerar el orden de los valores de entrada.

Definición 2.10 *Una* **función de agregación de tipo ordenado ponderado (OWA)** *es una función de* n *argumentos,* $f : \mathbb{I}^n \to \mathbb{I}$*, definida como:*

$$f(x_1, x_2, ..., x_n) = \sum_{i=1}^{n} w_i x_{(i)} \tag{20}$$

donde $x_{(1)}, x_{(2)}, ..., x_{(n)}$ *denotan los valores de entrada ordenados en orden no creciente y* $w_1, w_2, ..., w_n$ *son los pesos asignados a las posiciones, con* $w_i \geq 0$ *y* $\sum_{i=1}^{n} w_i = 1$*.*

Las funciones OWA permiten una mayor flexibilidad en la agregación de información, ya que permiten modelar distintos grados de optimismo o pesimismo en la toma de decisiones. Por ejemplo, si los pesos están concentrados en los valores más altos, la función OWA se comportará de manera similar a la función de máximo; si los pesos están concentrados en los valores más bajos, se comportará de manera similar a la función de mínimo. Si los pesos están distribuidos de manera uniforme, la función OWA se comportará de manera similar a la media aritmética.

Como casos especiales de la función OWA, se tienen:

- Si todos los pesos son iguales con $w_i = \frac{1}{n}$, la función *OWA* se comporta como la media aritmética.
- Si $\mathbf{w} = (1, 0, ..., 0)$, entonces $OWA_{\mathbf{w}}(\mathbf{x}) = \max(\mathbf{x})$.
- Si $\mathbf{w} = (0, 0, ..., 1)$, entonces $OWA_{\mathbf{w}}(\mathbf{x}) = \min(\mathbf{x})$.
- La función *Olímpica OWA* (Coroianu et al., 2022) tiene como vector de pesos $\mathbf{w} = (0, \frac{1}{n-2}, ..., \frac{1}{n-2}, 0)$, lo que representa un caso especial de medias truncadas en el que se excluyen los valores extremos.

2.2.5. Funciones de agregación de tipo ordenado ponderado con pesos variables (WOWA)

Las *funciones de agregación de tipo ordenado ponderado con pesos variables (WOWA)*, introducidas por (Yager & Filev, 1999), son una generalización de las funciones OWA que permiten mayor flexibilidad en la agregación de la información. Estas funciones consideran el orden de los valores de entrada y también incorporan un grado de importancia a cada uno de estos valores.

Definición 2.11 *Una* **función de agregación de tipo ordenado ponderado con pesos variables (WOWA)** *es una función de* n *argumentos,* $f : \mathbb{I}^n \to \mathbb{I}$*, definida como:*

$$f(x_1, x_2, ..., x_n) = \sum_{i=1}^{n} w_i x_{(i)} \tag{21}$$

donde $x_{(1)}, x_{(2)}, ..., x_{(n)}$ *denotan los valores de entrada ordenados en orden no creciente, y* $w_1, w_2, ..., w_n$ *son los pesos asignados a las posiciones, calculados como:*

$$w_i = \sum_{j=1}^{i} o_j - \sum_{j=1}^{i-1} o_j \tag{22}$$

con $o_1, o_2, ..., o_n$ *siendo los pesos de orden, con* $o_i \geq 0$ *y* $\sum_{i=1}^{n} o_i = 1$.

Las funciones WOWA permiten una mayor flexibilidad en la agregación de la información, ya que además de considerar el orden de los valores, también toman en cuenta el grado de importancia de cada uno de estos valores.

2.2.6. Métodos de agregación de vectores

Hasta ahora, hemos discutido funciones de agregación que operan en valores escalares. Sin embargo, en muchas situaciones, especialmente en el contexto de las GNN, es necesario operar en vectores. Todas las funciones de agregación que hemos mencionado hasta ahora se pueden extender para trabajar con vectores. En lugar de operar en un conjunto de valores escalares, ahora consideramos un conjunto de vectores.

La extensión de las funciones de agregación a vectores generalmente implica aplicar la función de agregación componente por componente. Dado un conjunto de vectores $\{\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n\}$, donde cada vector $\mathbf{v}_i$ es un vector d-dimensional, es decir, $\mathbf{v}_i = (v_{i1}, v_{i2}, ..., v_{id})$, la función de agregación f se aplica a cada conjunto correspondiente de componentes $(v_{1j}, v_{2j}, ..., v_{nj})$ para $j \in \{1, 2, ..., d\}$. Esto produce un nuevo vector $\mathbf{v} = (f(v_{1j}, v_{2j}, ..., v_{nj}), ..., f(v_{1d}, v_{2d}, ..., v_{nd}))$.

Por ejemplo, en el caso del promedio ponderado, dado un conjunto de vectores y un conjunto de pesos, el promedio ponderado de los vectores se calcula componente por componente utilizando los mismos pesos para cada componente. Del mismo modo, para las funciones OWA y WOWA, los vectores de entrada se ordenan y ponderan componente por componente.

Esta extensión de las funciones de agregación a vectores nos permite manejar información más compleja y rica en nuestras operaciones de agregación. En particular, en el contexto de las GNN, esto nos permite capturar y combinar características multidimensionales de los nodos y sus vecindarios.

2.3. Medidas difusas e integral discreta de Choquet

En esta sección, desarrollaremos los conceptos esenciales para comprender la integral discreta de Choquet, una función que generaliza las funciones WOWA (Lust, 2015). Como punto de partida, introduciremos el concepto de medida difusa, que es un ingrediente clave para definir la integral discreta de Choquet.

2.3.1. Medidas difusas

En el estudio de la incertidumbre y la toma de decisiones bajo condiciones imprecisas, es importante contar con herramientas que nos permitan representar y manejar adecuadamente la información disponible. Uno de estos enfoques es el de las medidas difusas, que extienden y generalizan el concepto de funciones de probabilidad, permitiendo modelar situaciones en las que no es posible asignar probabilidades de manera convencional a los eventos.

A las medidas difusas se les han encontrado diversas aplicaciones en teoría de la decisión (Grabisch & Roubens, 2000), inteligencia artificial (Yager, 1988) y procesamiento de la información, entre otras áreas. Son especialmente útiles para modelar la incertidumbre y la ambiguedad inherentes a muchos sistemas reales (Klir & Yuan, 1995). En el contexto de las funciones de agregación, las medidas difusas permiten introducir una estructura de interacción entre los argumentos. A continuación, se presenta la definición formal de una medida difusa:

Definición 2.12 Medida difusa. *Sea $\mathcal{A}$ una familia de subconjuntos de un conjunto universo Ω, con $\varnothing \in \mathcal{A}$. Una función $\mu : \mathcal{A} \to [0, \infty)$ es llamada* **medida difusa** *si satisface las siguientes propiedades:*

1. $\mu(\varnothing) = 0$.
2. *Para todo $A, B \in \mathcal{A}$ tal que $A \subseteq B$, se tiene que $\mu(A) \leq \mu(B)$.*

Al trío $(\Omega, \mathcal{A}, \mu)$ se le denomina **espacio de medida difusa**.

La definición de medida difusa presentada establece dos propiedades fundamentales que deben cumplir estas funciones para ser consideradas como tal. La primera propiedad establece que la medida difusa de un conjunto vacío $\varnothing$ es igual a cero, es decir, $\mu(\varnothing) = 0$. Esto refleja la idea de que un evento que no tiene elementos no puede tener ninguna medida de incertidumbre asociada. Esta propiedad es análoga al caso de las funciones de probabilidad, donde la probabilidad del evento vacío también es cero.

La segunda propiedad, conocida como monotonía, requiere que si un conjunto A está contenido en otro conjunto B, entonces la medida difusa de A debe ser menor o igual a la medida difusa de B, o formalmente, $\mu(A) \leq \mu(B)$. Esta propiedad asegura que la medida difusa asignada a un conjunto no disminuye al agregar más elementos a dicho conjunto. Esta característica es fundamental para garantizar una interpretación intuitiva y coherente de la medida difusa como una generalización de las funciones de probabilidad.

Dentro de la teoría de las medidas difusas y su aplicación en el análisis y gestión de la incertidumbre, es útil considerar diferentes tipos de medidas difusas que reflejen distintos comportamientos en relación a la agregación de conjuntos. En este sentido, se introducen tres clases de medidas difusas: aditivas, sub-aditivas y super-aditivas, cada una con propiedades específicas que pueden ser útiles en distintos contextos y aplicaciones. A continuación, se presentan las definiciones formales de estas clases de medidas difusas:

Definición 2.13 Medida aditiva. *Una medida difusa μ es* **aditiva** *si para todo par de conjuntos disjuntos $A, B \in \mathcal{A}$, se tiene que $\mu(A \cup B) = \mu(A) + \mu(B)$.*

Definición 2.14 Medida sub-aditiva. *Una medida difusa μ es* **sub-aditiva** *si para todo par de conjuntos $A, B \in \mathcal{A}$, se tiene que $\mu(A \cup B) \leq \mu(A) + \mu(B)$.*

Definición 2.15 Medida super-aditiva. *Una medida difusa μ es* **super-aditiva** *si para todo par de conjuntos $A, B \in \mathcal{A}$, se tiene que $\mu(A \cup B) \geq \mu(A) + \mu(B)$.*

2.3.2. Integral discreta de Choquet

Al tomar decisiones entre un conjunto de opciones, es común promediar los valores de sus características. Ya sea que se haga con el promedio o promedio ponderado, un valor alto sobre una característica específica suele ser determinante en la agregación. Sin embargo, seleccionar una opción solo con base en los valores altos en cada una de sus características no siempre es lo deseado por el tomador de decisiones. En tales casos, el promedio o el promedio ponderado no son útiles para encontrar opciones equilibradas. Este problema se ilustra con el siguiente ejemplo:

Ejemplo 2.2 *Imaginemos que deseamos adquirir un automóvil, para lo cual tenemos a disposición tres modelos diferentes: Modelo 1, Modelo 2 y Modelo 3. Para tomar una decisión, consideraremos dos criterios: confort y calidad. A continuación se presentan los datos correspondientes:*

Tabla 2. Opciones y criterios de selección.

Opción	Confort	Calidad
modelo 1	5	5
modelo 2	7	3
modelo 3	3	7

Supongamos que se desea seleccionar una opción equilibrada mediante un promedio ponderado. Al observar la tabla, se concluye que la opción más equilibrada es el Modelo 1. Para seleccionar este modelo mediante un promedio ponderado, se deben satisfacer las siguientes condiciones para dos ponderaciones w_1 y w_2, tales que $w_1, w_2 \in [0, 1]$:

$$\begin{aligned} 5w_1 + 5w_2 &> 3w_1 + 7w_2 \\ 5w_1 + 5w_2 &> 7w_1 + 3w_2 \end{aligned} \tag{23}$$

desarrollando se obtiene

$$\begin{aligned} 5w_1 &> 3w_1 + 2w_2 \\ 5w_2 &> 2w_1 + 3w_2 \end{aligned} \tag{24}$$

sumando ambas inecuaciones se obtiene que

$$5w_1 + 5w_2 > 5w_1 + 5w_2 \tag{25}$$

Por lo tanto, no existen pesos w_1 y w_2 que hagan al Modelo 1 la opción a seleccionar.

Hemos observado que el uso del promedio ponderado no tiene la suficiente robustez para abarcar todas las posibles preferencias de un tomador de decisiones. Por lo tanto, a continuación se presenta una herramienta más adecuada para tal fin.

2.3.3. La integral discreta de Choquet para la toma decisiones multicriterio

En esta subsección, nos centraremos en la integral discreta de Choquet y observaremos que es muy amplia en cuanto a capacidades en comparación con otras funciones de agregación. La integral discreta de Choquet engloba como casos particulares al promedio, promedio ponderado, la función máximo, la función mínimo, todos los cuantiles estadísticos y los operadores de agregación tipo OWA y WOWA (Lust, 2015).

Antes de presentar la definición formal de la integral discreta de Choquet, es necesario introducir el concepto de permutación no decreciente.

Definición 2.16 Permutación no decreciente. *Denotamos por $\mathbf{x}_{\nearrow}$ al vector obtenido a partir de $\mathbf{x}$ al ordenar sus componentes de forma no decreciente. Es decir, $\mathbf{x}_{\nearrow} = \mathbf{x}_{\boldsymbol{\pi}}$, donde π es una permutación que satisface $x_{\pi(1)} \leq x_{\pi(2)} \leq ... \leq x_{\pi(n)}$.*

Definición 2.17 Integral discreta de Choquet. *Dado un vector $\mathbf{x}$ de tamaño n, la integral discreta*

de Choquet de $\mathbf{x}$ *con respecto a una medida difusa* μ *se define como*

$$C_\mu(\mathbf{x}) = \sum_{i=1}^{n} [x_{(i)_\nearrow} - x_{(i-1)_\nearrow}]\ \mu(H_i) \tag{26}$$

donde $\mathbf{x}_\nearrow = (x_{(1)_\nearrow}, ..., x_{(n)_\nearrow})$ *es una permutación no decreciente de* $\mathbf{x}$*. Además,* $x_{(0)} = 0$ *por convención, y* $H_i = \{(i), ..., (n)\}$ *es el subconjunto de índices de las* $n - i + 1$ *características más grandes de* $\mathbf{x}$

A continuación, se presenta un ejemplo de cómo se calcula la integral discreta de Choquet:

Ejemplo 2.3 *Supongamos que* $\mathbf{x} = (10, 7, 8, 6)$ *es el vector de calificaciones obtenidas por un estudiante en los cursos de matemáticas, música, literatura y biología, respectivamente. De esta forma, podemos ordenar el vector de calificaciones en orden no decreciente, obteniendo así* $\mathbf{x}_\nearrow = (6, 7, 8, 10)$*. Luego, los subconjuntos de índices son los siguientes:*

$$\begin{aligned} H_1 &= \{\textit{biología}, \textit{música}, \textit{literatura}, \textit{matemáticas}\} \\ H_2 &= \{\textit{música}, \textit{literatura}, \textit{matemáticas}\} \\ H_3 &= \{\textit{literatura}, \textit{matemáticas}\} \\ H_4 &= \{\textit{matemáticas}\} \end{aligned} \tag{27}$$

Tenga en cuenta que las asignaturas de H_1 *están ordenadas de menor a mayor según el orden en* $\mathbf{x}_\nearrow$*. A continuación, en* H_2 *se encuentran las tres asignaturas con las calificaciones más altas. Luego, en* H_3 *se incluyen las dos asignaturas con las calificaciones más altas, mientras que en* H_4 *se encuentra la asignatura con la calificación más alta.*

Además, para el ejemplo, supongamos que la función de medida μ *está definida de la siguiente manera para cada* H_i*:*

$$\begin{aligned} \mu(H_1) &= 1 \\ \mu(H_2) &= 0.3 \\ \mu(H_3) &= 0.2 \\ \mu(H_4) &= 0.5 \end{aligned} \tag{28}$$

Por lo que el cálculo de la integral discreta de Choquet de $\mathbf{x}$ *es el siguiente:*

$$
\begin{aligned}
C_\mu(\mathbf{x}) &= [x_{(1)_\nearrow} - x_{(0)_\nearrow}]\mu(H_1) + [x_{(2)_\nearrow} - x_{(1)_\nearrow}]\mu(H_2) + [x_{(3)_\nearrow} - x_{(2)_\nearrow}]\mu(H_3) + [x_{(4)_\nearrow} - x_{(3)_\nearrow}]\mu(H_4) \\
C_\mu(\mathbf{x}) &= [6-0](1) + [7-6](0.3) + [8-7](0.2) + [10-8](0.5) \\
C_\mu(\mathbf{x}) &= 6 + 0.3 + 0.2 + 1 \\
C_\mu(\mathbf{x}) &= 7.5
\end{aligned}
\tag{29}
$$

Ahora veremos cómo la integral discreta de Choquet puede proporcionar una solución balanceada al problema del Ejemplo 2.2. Para ello, definimos μ *como:*

$$
\begin{aligned}
\mu(\{\textit{confort}\}) &= 0.25 \\
\mu(\{\textit{calidad}\}) &= 0.35 \\
\mu(\{\textit{confort}, \textit{calidad}\}) &= 1
\end{aligned}
\tag{30}
$$

De lo anterior, podemos observar que la calidad tiene una prioridad ligeramente superior al confort. Sin embargo, es importante destacar que la combinación de confort y calidad adquiere una prioridad aún mayor que cuando se consideran por separado. Luego, los valores de la integral discreta de Choquet de cada opción se calcularían de la siguiente manera:

$$
\begin{aligned}
C_\mu(\textit{modelo 1}) &= [5-0] \times 1 + [5-5] \times 0.25 = 5 \\
C_\mu(\textit{modelo 2}) &= [3-0] \times 1 + [7-3] \times 0.25 = 4 \\
C_\mu(\textit{modelo 3}) &= [3-0] \times 1 + [7-3] \times 0.35 = 4.4
\end{aligned}
\tag{31}
$$

Es importante destacar que el Modelo 1 es la opción con el valor más alto de la integral discreta de Choquet, lo que la coloca por encima de las demás opciones y soluciona el problema presentado en el Ejemplo 2.2, el cual empleaba el promedio ponderado.

La integral discreta de Choquet tiene la capacidad de combinar la información de modo que no solo se ponderen las características individuales, sino también grupos de características (Beliakov et al., 2016). Como se puede apreciar en el ejemplo anterior, $\mu(\text{confort}, \text{calidad}) = 1$, lo que pondera ambos criterios y tiene mayor peso en comparación con $\mu(\text{confort}) = 0.25$ y $\mu(\text{calidad}) = 0.35$.

Se puede calcular la integral discreta de Choquet de un vector $\mathbf{x}$ con respecto a una medida μ utilizando el Algoritmo 1.

2.3.4. Alcance y limitaciones de la integral discreta de Choquet

Hasta el momento, se ha demostrado que la integral discreta de Choquet puede resolver problemas que el promedio y promedio ponderado no pueden abarcar. Además, como una ventaja adicional, la integral discreta de Choquet puede modelar esas funciones de agregación. Por ejemplo, para modelar la media aritmética, solo es necesario definir una medida μ para cada subconjunto $A \subseteq C$ de características, como $\mu(A) = \frac{|A|}{|C|}$. Sin embargo, en general, el uso práctico de la integral discreta de Choquet se ve limitada por la necesidad de definir una función de medida μ que satisfaga las prioridades del tomador de decisiones (Klir et al., 1997). El problema radica en que, para un conjunto C con m características, se deben definir $2^m - 2$ valores de medida para cada subconjunto $A \subseteq C$. Incluso para valores pequeños de m, este problema no es tratable computacionalmente.

Se han desarrollado diversos métodos para determinar funciones de medida que satisfacen ciertas condiciones, por ejemplo, utilizando técnicas de muestreo y algoritmos genéticos (Chen & Wang, 2001). En nuestra propuesta de tesis, hemos decidido implementar la medida de Sugeno debido a que presenta cualidades prácticas que se ajustan bien en el marco de las redes neuronales. Además, el costo de cálculo de los valores de medida es de orden lineal en el tamaño de C, por lo que vuelve práctico el uso de la integral discreta de Choquet.

Algoritmo 1: Cálculo de la integral discreta de Choquet de un vector $\mathbf{x}$ con respecto a una medida μ

Input : Vector $\mathbf{x}$ de tamaño n; Medida μ.
Output: Valor de la integral discreta de Choquet de $\mathbf{x}$ con respecto a μ.

```
Hacer suma = 0;
Ordenar x de forma no decreciente;
for i = 1, 2, ..., n do
    if i = 1 then
        choquet = x[i] · μ(H_i);
    else
        choquet = (x[i] − x[i − 1]) · μ(H_i);
    suma = suma + choquet;
```

2.3.5. Medida de Sugeno

Como se mencionó anteriormente, determinar las funciones de medida es un problema computacionalmente costoso. Por lo tanto, es recomendable buscar soluciones que reduzcan este costo. En nuestro caso, la medida de Sugeno ofrece facilidades de implementación para el uso práctico de la integral discreta de

Choquet.

Definición 2.18 Medida de Sugeno. *Dado un parámetro $\lambda \in [-1, \infty)$, la medida de Sugeno es una medida difusa μ que para cada $A, B \subseteq \Omega, A \cap B = \varnothing$ satisface*

$$\mu(A \cup B) = \mu(A) + \mu(B) + \lambda\mu(A)\mu(B) \tag{32}$$

Cuando $\lambda \in [-1, 0)$, μ es una medida difusa sub-aditiva, cuando $\lambda = 0$ es aditiva, y cuando $\lambda > 0$ es super-aditiva. La medida de Sugeno también es conocida como una λ-medida difusa.

En la definición de *medida difusa* en general no impone restricciones sobre el valor de medida de Ω, pero es práctico definir a partir de este punto $\mu(\Omega) = 1$ para las medidas subsiguientes. Esta cualidad se considera como un criterio de normalización y resulta útil para el cálculo de la medida de Sugeno.

Dado un conjunto de características $\Omega = \{g_1, ..., g_n\}$, podemos definir valores de medida para cada $\mu(\{g_i\}) = p_i$, donde cada $p_i \in [0, 1]$ representa la medida de importancia asociada a la característica g_i. A partir de la definición de la medida de Sugeno, se puede calcular la medida de cada subconjunto A de Ω mediante la siguiente fórmula:

$$\mu(A) = \frac{\prod_{g_i \in A}(1 + \lambda p_i) - 1}{\lambda} \tag{33}$$

De la fórmula anterior, se observa que solo queda por determinar el valor de λ para obtener $\mu(A)$. A partir de los valores $p_1, ..., p_n$, podemos obtener el valor de λ resolviendo la siguiente ecuación:

$$\lambda + 1 = \prod_{i=1}^{n}(1 + \lambda p_i) \tag{34}$$

La ecuación anterior se obtiene al intercambiar A por Ω en la formula de $\mu(A)$ y aplicando el criterio de normalización mencionado previamente. El valor de λ puede determinarse utilizando un método numérico, el cual será discutido más adelante.

Con respecto a las soluciones de la ecuación 34, es importante destacar que el valor $\lambda = 0$ satisface la ecuación, pero no permite determinar un valor de $\mu(A)$ para cada $A \subset \Omega$ debido a la indeterminación de la fórmula, por lo que se buscan soluciones donde $\lambda \neq 0$. Un detalle importante es que la solución para $\lambda \in [-1, 0) \cup (0, \infty)$ es única (Leszczynski et al., 1985). Este hecho es relevante ya que el comportamiento de μ adquiere propiedades sub-aditivas o super-aditivas en función de dónde se ubique el valor de λ en relación a 0, siendo μ sub-aditiva cuando $\lambda \in [-1, 0)$ y super-aditiva cuando $\lambda \in (0, \infty)$.

Otro aspecto fundamental es determinar en qué casos $\lambda \in [-1, 0)$ o $\lambda \in (0, \infty)$. Siendo el caso cuando $\sum_{i=1}^{n} p_i > 1$ entonces $\lambda \in [-1, 0)$ y cuando $\sum_{i=1}^{n} p_i < 1$ entonces $\lambda \in (0, \infty)$ (Leszczynski et al., 1985). Por lo tanto, se pueden definir los valores p_i para hacer que la medida μ posea propiedades sub-aditivas o super-aditivas.

2.4. Programación de redes neuronales en grafos usando PyTorch Geometric

2.4.1. Pytorch Geometric

PyTorch Geometric (PyG) es una biblioteca desarrollada sobre PyTorch, diseñada para facilitar la escritura y entrenamiento de Redes Neuronales basadas en Grafos (GNNs) que pueden ser utilizadas en una amplia gama de aplicaciones relacionadas con datos estructurados (Fey & Lenssen, 2019). Recordemos que las GNNs son una clase de algoritmos de aprendizaje profundo especializados en procesar y extraer información útil de datos representados como grafos o estructuras irregulares (Battaglia et al., 2018).

El uso de PyTorch Geometric se justifica por varias razones. En primer lugar, su API unificada y fácil de usar permite a los usuarios familiarizados con PyTorch implementar rápidamente modelos de GNN con solo 10-20 líneas de código, tal como se puede observar en los ejemplos suministrados en su repositorio en GitHub [1]. Además, la biblioteca cuenta con una amplia gama de modelos de GNN predefinidos y bien mantenidos, que incluyen las arquitecturas más recientes y avanzadas [2]. PyG también ofrece una gran flexibilidad, permitiendo a los usuarios adaptar fácilmente los modelos existentes o crear nuevas arquitecturas para llevar a cabo investigaciones propias en el campo de las GNNs.

A pesar de sus ventajas, PyTorch Geometric también presenta algunas desventajas. Al estar construida sobre PyTorch, la biblioteca hereda sus limitaciones y puede ser menos eficiente en términos de rendimiento en comparación con otras soluciones especializadas en GNN, como DGL [3] (Wang et al., 2019a).

Dado que la documentación de PyTorch, que constituye el núcleo sobre el cual se construye PyTorch Geometric, es extensa y abarca una amplia gama de recursos, como cursos, libros y foros en línea, existe una cantidad significativa de documentación aplicable también a PyTorch Geometric. Además, PyTorch Geometric proporciona materiales educativos adicionales, tales como videos y cuadernos Jupyter[4], que

[1] https://github.com/pyg-team/pytorch_geometric
[2] https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html
[3] https://www.dgl.ai/
[4] https://pytorch-geometric.readthedocs.io/en/latest/get_started/colabs.html

facilitan la comprensión y aplicación de GNNs en proyectos de investigación. Gracias a estas ventajas, se ha seleccionado PyTorch Geometric como el entorno de trabajo para la presente propuesta de tesis.

2.4.2. Instalación

La instalación de PyTorch Geometric (PyG) se puede realizar a través de diversas vías, incluyendo Anaconda, Pip Wheels y compilación desde el código fuente. PyG es compatible con versiones de Python desde 3.7 hasta 3.10 y está disponible para los sistemas operativos Windows, Mac y Linux. Para obtener información detallada sobre la instalación, se puede consultar la documentación en el sitio web oficial de PyTorch Geometric [5].

2.4.3. Tipos de datos, bases de datos y mini-batches

En esta sección, discutiremos tres puntos clave relacionados con el manejo de datos en grafos, conjuntos de datos de referencia comunes y mini-lotes utilizando la biblioteca PyTorch Geometric.

2.4.3.1. Conjuntos de datos

PyG incluye una amplia gama de conjuntos de datos de referencia (también conocidos como *benchmark datasets*), los cuales son frecuentemente empleados en la literatura científica para evaluar y comparar el rendimiento de diversos modelos y arquitecturas de GNNs en tareas de clasificación de nodos y de grafos. La lista completa de estos conjuntos de datos de referencia está disponible en la página web correspondiente[6].

El procedimiento general para importar un conjunto de datos comienza por importar el paquete correspondiente desde la librería `torch_geometric.datasets`. En este ejemplo, se utiliza la función `TUDataset`, la cual permite acceder a diversas bases de datos químicas. Luego, se especifica la ruta y el nombre específico del conjunto de datos que se va a importar. En este caso, se importa el conjunto de datos llamado `ENZYMES`. La información se guarda en una variable llamada `dataset`. Este proceso puede visualizarse en las siguientes líneas de código.

El procedimiento general para importar un conjunto de datos comienza con la importación del módulo adecuado desde la biblioteca `torch_geometric.datasets`. En este ejemplo, se emplea la clase

[5] https://pytorch-geometric.readthedocs.io/en/latest/install/installation.html#quick-start
[6] https://pytorch-geometric.readthedocs.io/en/latest/modules/datasets.html

TUDataset, que permite acceder a varias bases de datos químicas. A continuación, se especifica la ruta y el nombre específico del conjunto de datos que se desea importar. En este caso, se importa el conjunto de datos denominado ENZYMES. La información se almacena en una variable llamada ENZYMES. Este proceso puede visualizarse en las siguientes líneas de código.

```
from torch_geometric.datasets import TUDataset
dataset = TUDataset(root='/tmp/ENZYMES', name='ENZYMES')
>>> ENZYMES(600)
```

Lo anterior, indica que el conjunto de datos se ha cargado correctamente y contiene 600 grafos.

En algunas situaciones, es necesario obtener información detallada del conjunto de datos importado, como la cantidad de grafos que contiene, el número de clases asociadas a cada grafo o el tamaño del vector de características asociado a cada nodo. Para acceder a esta información, se pueden utilizar los siguientes comandos en el objeto dataset previamente importado:

```
len(dataset) # Cantidad de grafos en el conjunto de datos
>>> 600
dataset.num_classes #Cantidad de clases asociadas al conjunto de datos
>>> 6
dataset.num_node_features #Cantidad de caracteristicas asociadas a cada
    nodo
>>> 3
```

2.4.3.2. Datos

Hemos visto cómo importar conjuntos de datos de referencia comunes. A continuación, examinaremos los atributos que describen una instancia de esos conjuntos de datos, es decir, un grafo. Un grafo individual en PyG es representado por una instancia de clase torch_geometric.data.Data y contiene, por omisión, los siguientes atributos:

- data.x: Matriz de características de los nodos con forma [num_nodos, num_características_nodo].
- data.edge_index: Conectividad del grafo en formato COO con forma [2, num_aristas] y tipo torch.long.
- data.edge_attr: Matriz de características de las aristas con forma [num_aristas, num_características_aristas].

- `data.y`: Objetivo contra el cual entrenar (puede tener una forma arbitraria), por ejemplo, objetivos a nivel de nodo con forma [num_nodos, *] o a nivel de grafo con forma [1, *].
- `data.pos`: Matriz de posición de los nodos con forma [num_nodos, num_dimensiones].

El símbolo * indica que el número de elementos en esa dimensión puede ser arbitrario. Por ejemplo, en el caso de `data.y`, puede tener cualquier número de elementos en la segunda dimensión, ya sea a nivel de nodo o a nivel de grafo. Para obtener ejemplos e información completa sobre la clase `torch_geometric.data.Data` busque en la página correspondiente [7].

Tomando en cuenta la variable `dataset` mencionada anteriormente, que contiene los 600 grafos de la base `ENZYMES`, para acceder al primer grafo de dicha base se puede hacer mediante el siguiente código:

```
data = dataset[0] # Acceder al primer grafo de la base ENZYMES
>>> Data(edge_index=[2, 168], x=[37, 3], y=[1])
```

A partir de la salida, se observan las dimensiones de los atributos previamente mencionados. Notamos que el grafo contiene 168 aristas, 37 nodos y el tamaño del vector de características asociado a cada nodo es de 3 elementos. Además, el grafo tiene asociada una etiqueta.

2.4.3.3. Procesamiento en mini-lotes

El entrenamiento de redes neuronales se lleva a cabo, por lo general, utilizando lotes. PyG consigue paralelizar el proceso sobre un mini-lote al crear matrices de adyacencia diagonales dispersas (definidas por edge_index) y concatenar matrices de características y vectores objetivos. En otras palabras, este enfoque implica combinar todos los grafos para crear un único grafo gigante, que tiene una única matriz de adyacencia que lo representa, además de apilar todas las matrices de características de dichos grafos y crear un solo vector de etiquetas. En lugar de procesar los ejemplos uno a uno, un mini-lote agrupa un conjunto de ejemplos en una representación unificada que permite procesarlos de manera eficiente y paralela.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & & \\ & \ddots & \\ & & \mathbf{A}_n \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_n \end{bmatrix}, \qquad \mathbf{Y} = \begin{bmatrix} \mathbf{Y}_1 \\ \vdots \\ \mathbf{Y}_n \end{bmatrix} \tag{35}$$

[7] https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.data.Data.html

PyG dispone de un cargador de datos específico (DataLoader), torch_geometric.loader.DataLoader, que se encarga de realizar este proceso de concatenación. Tomando en cuenta la base de datos `ENZYMES` previamente cargada en la variable `dataset`, la generación de mini-lotes se realiza de la siguiente forma:

```
from torch_geometric.loader import DataLoader

loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

En el código de ejemplo presentado, se utiliza un tamaño de lote (batch_size) de 32. Este parámetro define cuántos ejemplos se agruparán en un mini-lote para el procesamiento en paralelo. El tamaño del lote puede afectar tanto la eficiencia computacional como la calidad del aprendizaje, por lo que es importante ajustarlo adecuadamente.

El parámetro `shuffle=True` indica que los datos se mezclarán antes de ser divididos en mini-lotes. Esto ayuda a garantizar que cada mini-lote sea aleatorizado, lo que puede mejorar la calidad del aprendizaje. Un observación importante es que `torch_geometric.data.Batch` hereda de `torch_geometric.data.Data` y contiene un atributo adicional llamado `batch`.

El atributo `batch` es un vector columna que asigna cada nodo a su respectivo grafo en el lote:

$$\text{batch} = \begin{bmatrix} 0 & \cdots & 0 & 1 & \cdots & n-2 & n-1 & \cdots & n-1 \end{bmatrix}^\top \tag{36}$$

Para ver métodos avanzados sobre mini-lotes se puede acceder a la pagina destinada para ello [8].

2.4.4. Entrenamiento de una red neuronal en grafos para la clasificación de grafos

Hasta ahora hemos visto cómo cargar conjuntos de datos, los atributos asociados a cada grafo y cómo dividir los datos en mini-lotes para mejorar el rendimiento al entrenar una GNN. A continuación, proporcionaremos un esquema general, basado en un ejemplo detallado, para entrenar una GNN utilizando PyTorch Geometric con el propósito de clasificar grafos:

2.4.4.1. Importar datos y división por mini-lotes

Como se mencionó previamente, el primer paso consiste en cargar el conjunto de datos que se utilizará. En este caso, empleamos el conjunto de datos *PROTEINS* de la colección TUDataset. La siguiente línea

[8] https://pytorch-geometric.readthedocs.io/en/latest/advanced/batching.html

de código ilustra cómo cargar dicho conjunto de datos:

```
dataset = TUDataset(root, name="PROTEINS")
len(dataset)
>>> 1230 #cantidad de grafos en PROTEINS
```

Para facilitar el entrenamiento, dividimos el `dataset` en mini-lotes utilizando la clase `DataLoader`. En este ejemplo, se emplea un tamaño de lote de 256:

```
loader = DataLoader(dataset, batch_size=256)
```

2.4.4.2. Creación del modelo y definición de la arquitectura

El siguiente código define una clase `GNN` que implementa una red neuronal basada en grafos utilizando la biblioteca PyTorch y PyTorch Geometric. La red utiliza la capa `GATConv` (Graph Attention Network) como capas de convolución y la capa `Linear` para la clasificación.

El siguiente código ilustra cómo implementar una GNN utilizando PyTorch Geometric. Además de la definición de la clase, es importante destacar las dos partes que la conforman: el método `init`, en el cual se define la arquitectura de la red, y el método `forward`, que establece cómo fluirán los datos de entrada a través de la red.

```
class GNN(torch.nn.Module):
    def init(self):
        self.conv1 = GATConv(dataset.num_features, 16)
        self.conv2 = GATConv(16, 16)
        self.lin = Linear(16, dataset.num_classes)

    def forward(self, x, edge_index, batch):
        x = self.conv1(x, edge_index).relu()
        x = self.conv2(x, edge_index).relu()
        x = global_mean_pool(x, batch)
        return self.lin(x)
```

A continuación se detallan las componentes del código:

1. La clase `GNN` hereda de `torch.nn.Module`, lo que indica que es una subclase de una red neuronal en PyTorch.

2. El método `init` inicializa las capas de la red neuronal:
    - `self.conv1`: Primera capa `GATConv` que toma como entrada el número de características (atributos) de cada nodo en el grafo y produce 16 características de salida.
    - `self.conv2`: Segunda capa `GATConv` que toma como entrada 16 características y produce 16 características de salida.
    - `self.lin`: Capa lineal (fully connected) que toma como entrada 16 características y produce una salida del tamaño igual al número de clases en el conjunto de datos.
3. El método `forward` define el flujo hacia adelante (forward pass) de la red neuronal:
    - La entrada `x` es el tensor de características de los nodos, `edge_index` tiene la información de las aristas del grafo y `batch` es un vector que indica a qué grafo pertenece cada nodo en caso de que se utilicen mini-lotes.
    - La primera y segunda capas `GATConv` se aplican en secuencia, usando la función de activación ReLU entre ellas.
    - Después de aplicar las dos capas `GATConv`, se utiliza la función `global_mean_pool` para obtener un vector de características global para cada grafo en el lote (`batch`). Esta función calcula el promedio de las características de los nodos para cada grafo.
    - Finalmente, se aplica la capa lineal `self.lin(x)` para obtener la salida de la red, que puede ser utilizada para clasificación.

2.4.4.3. Inicialización del modelo

A continuación, se presenta un código que inicializa y configura nuestra GNN, así como el optimizador y la función de pérdida que serán utilizados durante el entrenamiento y evaluación del modelo:

```
model = GNN()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = torch.nn.CrossEntropyLoss()
```

A continuación se detallan las componentes del código:

1. `model = GNN()`: Esta línea crea una instancia de la clase `GNN` que se definió anteriormente. La instancia `model` representa la red neuronal basada en grafos y será utilizada para entrenar y evaluar el modelo en los datos.

2. `optimizer = torch.optim.Adam(model.parameters(), lr=0.01)`: Esta línea crea una instancia del optimizador Adam (Kingma & Ba, 2014) de la biblioteca PyTorch. El optimizador es responsable de actualizar los parámetros del modelo durante el entrenamiento para minimizar la función de pérdida. La función `model.parameters()` pasa todos los parámetros del modelo (es decir, los pesos y sesgos de las capas) al optimizador, mientras que `lr=0.01` establece la tasa de aprendizaje del optimizador en 0.01.

3. `criterion = torch.nn.CrossEntropyLoss()`: Esta línea crea una instancia de la función de pérdida de entropía cruzada de la biblioteca PyTorch. La entropía cruzada es una medida comúnmente utilizada para calcular la pérdida en problemas de clasificación. La función de pérdida `criterion` se utilizará para calcular la discrepancia entre las predicciones del modelo y las etiquetas reales durante el entrenamiento y la evaluación.

2.4.4.4. Entrenamiento del modelo

Finalmente, entrenamos nuestra GNN utilizando el cargador de mini-lotes creado previamente. En cada iteración, procesamos un lote de datos y calculamos la pérdida entre las predicciones y las etiquetas reales. El siguiente código muestra cómo entrenar el modelo GNN utilizando el optimizador y la función de pérdida establecidos previamente:

```
for epoch in range(1, 200):
    model.train()
    for data in loader:
        y_hat = model(data.x, data.edge_index, data.batch)
        loss = criterion(y_hat, data.y)
        loss.backward
        optimizer.step()
        optimizer.zero_grad()
```

A continuación se detallan las componentes del código:

1. `for epoch in range(1, 200)`: Este bucle se ejecutará 200 veces. En el contexto de entrenamiento de redes neuronales, cada iteración se conoce como una "época". Así que este código implica que el modelo se entrenará durante 200 épocas. Una época se refiere a cuando todo el conjunto de datos de entrenamiento ha sido utilizado una vez para actualizar los pesos de la red neuronal.

2. `model.train()`: Esta línea pone el modelo en modo de entrenamiento, lo que habilita características como el dropout y la normalización por lotes si se utilizan en el modelo.

3. `for data in loader`: Este bucle itera en lotes sobre el conjunto de datos de entrenamiento. El cargador de datos (`loader`) es responsable de dividir el conjunto de datos en lotes y entregarlos al modelo durante el entrenamiento.

4. `y_hat = model(data.x, data.edge_index, data.batch)`: Esta línea realiza una pasada hacia adelante (forward pass) a través del modelo utilizando las características de los nodos (`data.x`), la representación de las aristas del grafo (`data.edge_index`) y el vector de asignación de lotes (`data.batch`). La salida del modelo, `y_hat`, son las predicciones de clases para cada grafo en el lote.

5. `loss = criterion(y_hat, data.y)`: Esta línea calcula la función de pérdida usando las predicciones del modelo (`y_hat`) y las etiquetas reales (`data.y`) con la función de pérdida de entropía cruzada (`criterion`), establecida anteriormente.

6. `loss.backward()`: Esta línea calcula los gradientes de la función de pérdida con respecto a los parámetros del modelo (pesos y sesgos) mediante la propagación hacia atrás (backpropagation).

7. `optimizer.step()`: Esta línea actualiza los parámetros del modelo (pesos y sesgos) utilizando los gradientes calculados y la tasa de aprendizaje del optimizador.

8. `optimizer.zero_grad()`: Esta línea reinicia los gradientes, almacenados como un atributo de los parámetros, para el siguiente paso de entrenamiento. Es importante reiniciar los gradientes después de actualizar los parámetros del modelo al usar `optimizer.step()`, ya que, de lo contrario, se acumularían y afectarían las actualizaciones de los parámetros del modelo.

2.4.4.5. Resumen del proceso de entrenamiento de una GNN

A lo largo de esta sección, hemos abordado el proceso completo para entrenar una Red Neuronal basada en Grafos (GNN) utilizando PyTorch Geometric. Hemos examinado aspectos como la importación y carga de conjuntos de datos, así como la división de datos en mini-lotes para optimizar el rendimiento durante el entrenamiento. Posteriormente, diseñamos e implementamos una GNN personalizada, definiendo su arquitectura y estableciendo cómo los datos fluyen a través de la red. Finalmente, inicializamos el modelo, configuramos el optimizador y la función de pérdida, y llevamos a cabo el entrenamiento del modelo a lo largo de una cantidad considerable de épocas, actualizando los parámetros en cada iteración.

En resumen, el esquema general del proceso para entrenar una GNN con PyTorch Geometric es el siguiente:

1. Importar y cargar el conjunto de datos.
2. Procesar atributos y dividir los datos en mini-lotes.
3. Definir la arquitectura de la GNN y su flujo de datos.
4. Instanciar el modelo, el optimizador y la función de pérdida.
5. Entrenar el modelo a lo largo de varias épocas, actualizando los parámetros en cada iteración.

2.4.5. Aspectos clave en la implementación de nuevas funcionalidades en PyTorch y PyTorch Geometric

En esta sección, abordaremos aspectos clave para tener en cuenta al implementar nuevas funcionalidades en PyTorch y PyTorch Geometric. Primero, analizaremos el concepto de grafo computacional y su importancia en el aprendizaje profundo. Luego, discutiremos el motor de diferenciación automática de PyTorch y cómo funciona en relación con el grafo computacional. Finalmente, examinaremos los errores comunes que pueden ocurrir al utilizar el autograd de PyTorch y cómo evitarlos para garantizar una implementación exitosa de las funcionalidades personalizadas.

2.4.5.1. Concepto de grafo computacional

Antes de adentrarnos en la explicación del autograd en PyTorch, es importante entender el concepto de grafo computacional en sí mismo. Un grafo computacional es una representación gráfica de todas las operaciones matemáticas y sus relaciones en un proceso de cálculo. En el contexto del aprendizaje profundo, estas operaciones incluyen las operaciones realizadas en un modelo de red neuronal durante el entrenamiento.

Un grafo computacional está compuesto por nodos y aristas. Los nodos representan las operaciones matemáticas y las variables involucradas en ellas, mientras que las aristas simbolizan los resultados que fluyen entre dichas operaciones. La Figura 3 muestra el grafo computacional correspondiente a una capa de una red neuronal simple, donde la entrada es x, y los parámetros son w y b. Estas operaciones se

pueden resumir como $z = x * w + b$ y, posteriormente, $loss = CE(z, y)$, donde CE denota la función de entropía cruzada.

Este tipo de representación facilita el proceso de diferenciación automática, que es esencial para optimizar los parámetros de los modelos en entornos como PyTorch.

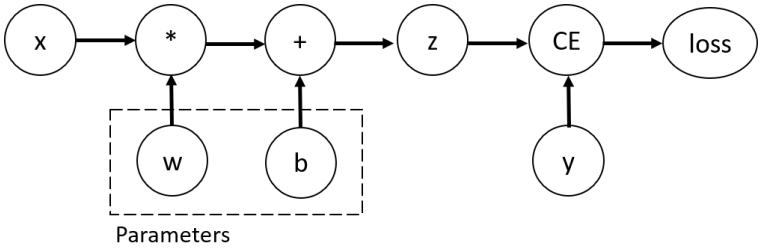

Figura 3. Grafo computacional de la operación $z = x * w + b$ y $loss = CE(z, y)$.

2.4.5.2. Motor de diferenciación automática de PyTorch

Un módulo clave en PyTorch es Torch.autograd, el cual permite realizar cálculos de gradientes automáticos en las operaciones de tensores, lo que facilita la actualización de los parámetros en un modelo de aprendizaje profundo durante el proceso de optimización.

El grafo computacional es una herramienta que permite representar todas las operaciones y cálculos realizados en un modelo de aprendizaje profundo. En PyTorch, este grafo se construye dinámicamente durante el paso hacia adelante (forward pass) del modelo y se emplea para calcular los gradientes de los parámetros con respecto a la función de pérdida en el paso hacia atrás (backward pass).

Al implementar nuevas funcionalidades y modelos no desarrollados en PyTorch o PyTorch Geometric que involucren pesos aprendibles, es esencial tener en cuenta el uso adecuado de autograd. Cuando se emplea PyTorch.autograd, resulta crucial garantizar que todas las operaciones y cálculos asociados con la nueva funcionalidad formen parte del grafo computacional. Si no se hace así, PyTorch no podrá calcular los gradientes de los nuevos pesos durante el paso hacia atrás, lo que ocasionará que estos pesos no se actualicen.

Tomemos en cuenta el ejemplo ilustrado en la Figura 3, que puede ser representado en Python de la siguiente forma:

```
import torch
x = torch.ones(5)
y = torch.zeros(3)
```

```
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

Observemos que los tensores w y b poseen el parámetro `requires_grad=True`. Este parámetro permite que los tensores sean rastreados en el grafo computacional y permite calcular el gradiente de la función de pérdida con respecto a estos tensores. Es crucial destacar que cualquier resultado obtenido al operar tensores con el parámetro `requires_grad=True` también tendrá dicho atributo.

2.4.5.3. Errores comunes al utilizar Autograd de PyTorch

Para finalizar esta sección, es importante destacar algunos errores comunes al utilizar PyTorch.autograd en la implementación de nuevas funcionalidades en PyTorch y PyTorch Geometric. Estos errores deben evitarse para garantizar un uso adecuado del autograd:

- No incluir las operaciones de la nueva funcionalidad en el grafo computacional.
- Utilizar tensores que no requieren gradientes (con el atributo `requires_grad` establecido en `False`) en las operaciones de la nueva funcionalidad.
- No conectar correctamente la nueva funcionalidad a la función de pérdida.

Si alguno de estos errores ocurre, los gradientes de los pesos de la nueva funcionalidad no se calcularán ni actualizarán durante el proceso de entrenamiento. Como resultado, el modelo no aprenderá la configuración óptima de estos pesos, lo que podría afectar negativamente su rendimiento en la tarea de clasificación.

2.4.5.4. Métodos de agregación en PyTorch Geometric

Hemos visto cómo definir y entrenar un modelo de GNN utilizando PyTorch Geometric. Al revisar el código presentado en la sección 2.4.4.2, notamos la utilización de la función `global_mean_pool(x, batch)` en el método `forward`, la cual corresponde al método de agregación de promedio ponderado.

```
class GNN(torch.nn.Module):
    def init(self):
        self.conv1 = GATConv(dataset.num_features, 16)
        self.conv2 = GATConv(16, 16)
        self.lin = Linear(16, dataset.num_classes)

    def forward(self, x, edge_index, batch):
        x = self.conv1(x, edge_index).relu()
        x = self.conv2(x, edge_index).relu()
        x = global_mean_pool(x, batch)
        return self.lin(x)
```

PyTorch Geometric proporciona varios métodos de agregación para la clasificación de grafos. Algunos de ellos incluyen:

- `global_mean_pool`: Agregación de promedio ponderado.
- `global_max_pool`: Agregación de máximo global.
- `global_add_pool`: Agregación de suma global.

Cada uno de estos métodos de agregación puede ser utilizado en función de las necesidades específicas de la tarea de clasificación de grafos en cuestión. Para obtener más información acerca de los métodos de agregación disponibles en PyTorch Geometric, se puede consultar la documentación correspondiente en su página web[9].

[9] https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#pooling-layers

Capítulo 3. Metodología

En este capítulo se presenta un nuevo método de agregación para la clasificación de grafos, basado en la integral discreta de Choquet. Se comenzará con una motivación del método a partir del uso de la integral discreta de Choquet en la toma de decisiones multicriterio. Posteriormente, se analizarán los alcances y limitaciones de la integral discreta de Choquet, y se presentará una solución para su aplicabilidad computacional basada en el uso de la medida de Sugeno. Luego, se definirá el método de agregación de Choquet aplicado a una matriz. Por último, se describirá el método de agregación de Choquet en el marco de las redes neuronales en grafos y su aplicación en la clasificación de grafos.

3.1. Propuesta

Los métodos de agregación tradicionales para la clasificación de grafos carecen de mecanismos que les permitan distinguir la forma en que se combina la información de un grafo. Estos métodos, al basarse en funciones de agregación del tipo conmutativas, tienen la limitación de no poder priorizar ni distinguir un orden de agregación que les permita dar una importancia relativa a la información dependiendo del contexto del problema.

La motivación detrás de esta propuesta es poder establecer una jerarquía en el conjunto de nodos de un grafo para determinar un orden de agregación, donde el nodo que ocupa el primer lugar en la jerarquía tenga una mayor ponderación en la agregación de toda la información del grafo.

La integral discreta de Choquet proporciona los mecanismos para jerarquizar los nodos de un grafo, además de la capacidad de combinar la información teniendo en cuenta la sinergia que existe entre las características de los nodos.

3.2. Método de agregación de Choquet

Anteriormente se ha visto el concepto de la integral discreta de Choquet y cómo el uso de la medida de Sugeno puede ser útil para su implementación práctica. En general, el uso de esta integral implica su aplicación más allá de la generación de un único valor, sino que se aplica a n vectores de igual tamaño. Los valores de la integral correspondientes a cada vector pueden ser útiles para generar un método de agregación que permita obtener un único vector que represente a todos los vectores. Por lo tanto, a continuación se definirá el método de agregación de Choquet aplicado a una matriz genérica.

Consideremos $X \in \mathbb{R}^{n \times m}$ una matriz de características genérica y sean $p_1, p_2, \ldots, p_m \in [0,1]$ los pesos asignados a cada una de las m características de X, respectivamente. Sea λ el valor que convierte μ en una medida de Sugeno para los pesos dados.

El método de agregación de Choquet tiene como objetivo combinar la información de la matriz X generando una jerarquía entre sus n filas. Para ello, se define $C_\mu(\mathbf{x_i})$ como el valor de la integral discreta de Choquet aplicada al i-ésimo renglón de X, donde μ es la medida de Sugeno utilizada para el cálculo.

El método de agregación de Choquet consiste en aplicar esta operación sobre cada uno de los renglones de X, lo que nos permite obtener $C_\mu(\mathbf{x_1}), C_\mu(\mathbf{x_2}), \ldots, C_\mu(\mathbf{x_n})$, que son los valores de la integral discreta de Choquet asociados a cada uno de los renglones de X. De esta manera, la jerarquía de agregación puede ser obtenida al ordenar estos valores de la siguiente manera:

$$C_\mu(\mathbf{x_{(1)}}) \geq C_\mu(\mathbf{x_{(2)}}) \geq ... \geq C_\mu(\mathbf{x_{(n)}}) \tag{37}$$

donde $\{(1), (2), ..., (n)\}$ es el conjunto de índices que ordena los valores de mayor a menor.

El método de agregación de Choquet propone que cada renglón de X se agregue de manera proporcional a su valor de Choquet. Para obtener esta agregación, podemos sumar los n renglones de X de la siguiente manera:

$$C_\mu(X) = \sum_{i=1}^{n} C_\mu(\mathbf{x_{(i)}}) \times \mathbf{x_{(i)}} \tag{38}$$

En la práctica, es común normalizar las características de los datos en tareas de clasificación (Singh & Singh, 2020). Para mantener esta escala en los datos, se propone que $C_\mu(\mathbf{x_{(i)}}) \in [0,1]$ y $\sum_{i=1}^{n} C_\mu(\mathbf{x_{(i)}}) = 1$. Para cumplir con estos requisitos, podemos recurrir a la función *softmax* (Bishop, 2006). Definimos entonces:

$$\hat{C}_\mu(\mathbf{x_{(i)}}) = \frac{e^{C_\mu(\mathbf{x_{(i)}})}}{\sum_{k=1}^{n} e^{C_\mu(\mathbf{x_{(k)}})}} \tag{39}$$

Teniendo en cuenta lo anterior, el método de agregación de Choquet para una matriz de características X se define de la siguiente manera:

$$C_\mu(X) = \sum_{i=1}^{n} \hat{C}_\mu(\mathbf{x_{(i)}}) \times \mathbf{x_{(i)}} \tag{40}$$

Es importante destacar que el método de agregación requiere ordenar la matriz X en cada una de sus filas. Si X tiene un tamaño de $n \times m$, entonces la complejidad del proceso de ordenamiento es de $\mathcal{O}(n \cdot m \cdot \log m)$. Este cálculo se basa en el supuesto de que se utiliza un algoritmo de ordenamiento con una complejidad de $\mathcal{O}(m \cdot \log m)$. Por tanto, el cálculo del método de agregación de Choquet puede ser muy costoso en el caso de que m o n sean muy grandes.

A continuación, se presentará un ejemplo del método de agregación de Choquet aplicado a una matriz de características.

Ejemplo 3.1 *Considerando la siguiente matriz de características X:*

$$X = \begin{bmatrix} 10 & 3 & 2 \\ 5 & 7 & 3 \\ 1 & 2 & 8 \end{bmatrix} \tag{41}$$

Definimos los pesos asociados a cada una de las características de X: (1), (2) y (3) como:

$$p_{(1)} = 0.5, \quad p_{(2)} = 0.6, \quad p_{(3)} = 0.2 \tag{42}$$

Recordando que $p_{(i)} = \mu(\{i\})$, para definir una medida de Sugeno a partir de estos valores, es necesario determinar el valor de λ que satisfaga la ecuación $\lambda + 1 = \prod_{i=1}^{3}(1 + \lambda p_{(i)})$. Para obtener dicho valor se requiere utilizar un método numérico. Una vez obtenido el valor de λ, que en este caso es $\lambda = -0.62149$, se debe asignar un valor de medida a los conjuntos de características restantes: $\{(1),(2)\}$, $\{(2),(3)\}$ y $\{(1),(3)\}$. Estos valores pueden ser obtenidos mediante la expresión $\mu(A) = \frac{\prod_{g_i \in A}(1+\lambda p_i)-1}{\lambda}$. Los valores correspondientes son:

$$\begin{aligned}
\mu(\{(1),(2)\}) &= \frac{(1-0.62149\times 0.5)(1-0.62149\times 0.6)-1}{-0.62149} = 0.91 \\
\mu(\{(2),(3)\}) &= \frac{(1-0.62149\times 0.6)(1-0.62149\times 0.2)-1}{-0.62149} = 0.72 \\
\mu(\{(1),(3)\}) &= \frac{(1-0.62149\times 0.5)(1-0.62149\times 0.2)-1}{-0.62149} = 0.64
\end{aligned} \tag{43}$$

Luego, los valores de la integral discreta de Choquet para cada uno de los renglones de la matriz X son los siguientes:

$$\begin{aligned}
C_\mu(\mathbf{x}_1) &= [2-0]\times 1 + [3-2]\times 0.91 + [10-3]\times 0.5 = 6.41 \\
C_\mu(\mathbf{x}_2) &= [3-0]\times 1 + [5-3]\times 0.64 + [7-5]\times 0.6 = 7.28 \\
C_\mu(\mathbf{x}_3) &= [1-0]\times 1 + [2-1]\times 0.72 + [8-2]\times 0.2 = 3.92
\end{aligned} \tag{44}$$

Al aplicar la función softmax a dicho conjunto de valores, se obtiene lo siguiente:

$$\begin{aligned}
\hat{C}_\mu(\mathbf{x_1}) &= 0.29 \\
\hat{C}_\mu(\mathbf{x_2}) &= 0.69 \\
\hat{C}_\mu(\mathbf{x_3}) &= 0.02
\end{aligned} \tag{45}$$

Ahora, podemos obtener la agregación de la matriz utilizando el método de agregación de Choquet:

$$\begin{aligned}
C_\mu(X) &= 0.29\times[10,3,2] + 0.69\times[5,7,3] + 0.02\times[1,2,8] \\
&= [2.9, 0.87, 0.58] + [3.45, 4.83, 2.07] + [0.02, 0.04, 0.16] \\
&= [6.37, 5.74, 2.81]
\end{aligned} \tag{46}$$

Podemos observar en la ecuación (12) que el método de agregación de Choquet incorpora la información de cada característica de manera proporcional a su respectivo coeficiente de Choquet. Como resultado, la mayor contribución a la agregación final está determinada por aquellas características que ocupan una posición más alta en la jerarquía definida por la integral discreta de Choquet.

3.3. Método de agregación de Choquet para la clasificación de grafos en redes neuronales basadas en grafos

3.3.1. Introducción al método de agregación de Choquet para clasificación de grafos

Como se ha visto, la integral discreta de Choquet es una herramienta flexible en cuanto a capacidades, pero su costo computacional puede ser elevado al tratar de determinar una función de medida adecuada para el problema. En este sentido, la medida de Sugeno resulta una buena opción para hacer más eficiente el cálculo de dicha función.

A pesar de las ventajas computacionales de la medida de Sugeno, surge otro problema a considerar. Supongamos que se tiene un problema con n opciones y m criterios. En este caso, un tomador de decisiones tendría que asignar un peso a cada uno de los m criterios. Sin embargo, a priori no se sabe si esta asignación de pesos cumplirá con sus preferencias, lo que podría llevar a que la jerarquía obtenida a partir de la integral de Choquet no sea la deseada.

Por lo que se propone implementar la integral discreta de Choquet en conjunción con la medida de Sugeno en el ámbito de las redes neuronales. Así, la asignación de pesos en la medida de Sugeno se convierte en un conjunto de parámetros que pueden ser aprendidos dentro de la estructura de una red neuronal, permitiendo que se ajusten automáticamente según las particularidades de cada problema.

Dicho esto, se identifica una oportunidad para aplicar esta combinación como un método de agregación en redes neuronales en grafos. Un método de agregación con estas características podría ser capaz de detectar interacciones entre las características asociadas a los nodos que forman parte del mismo grafo. Estas interacciones podrían ser resultado de la conectividad inherente al grafo, lo que motiva la implementación del método de agregación de Choquet.

3.3.2. Aplicación del método de agregación de Choquet para clasificación de grafos

Hasta ahora, hemos desarrollado una metodología para aplicar el método de agregación de Choquet en términos de su aplicación a una matriz X. En el marco teórico, se mencionaron los métodos de agregación para la clasificación de grafos en función de los vectores de características asociados a los nodos de un grafo. A continuación, se presentará una analogía matricial de dichos métodos para introducir de manera natural el desarrollo del método de agregación de Choquet en una GNN.

Consideremos un grafo $G = (V, E)$ y $X \in \mathbb{R}^{n \times m}$ como la matriz de características asociada a dicho

grafo. Es importante destacar que cada uno de los n renglones de X está asociado exclusivamente a un nodo en G, lo que implica que $|V| = n$. Además, m representa el número de características de X. De manera general, buscamos aprender dicho grafo mediante una *GNN* de K capas. Como resultado del aprendizaje, se obtiene una matriz de características ocultas $H^K \in \mathbb{R}^{n \times r}$, donde r es el número de características de H^K. Obsérvese que la matriz H^K sigue manteniendo n filas, cada una correspondiente a un nodo en el grafo. Esta matriz puede expresarse en función de X de la siguiente manera:

$$GNN^K(X) = H^K \tag{47}$$

Es importante destacar que cada fila i de H^K corresponde a un vector asociado al nodo i-ésimo del grafo G. De esta manera, podemos representar la matriz H^K como sigue:

$$H^K = \begin{bmatrix} -h_{v_1}^K- \\ -h_{v_2}^K- \\ \vdots \\ -h_{v_n}^K- \end{bmatrix} \tag{48}$$

Así, es posible expresar los métodos de agregación convencionales, como la suma, el promedio y máximo, en términos de la matriz de características ocultas resultante del entrenamiento de un grafo G utilizando una *GNN* de K capas de la siguiente manera:

$$\begin{aligned} \text{Suma}(H^K) &= \sum_{i=1}^{n} \mathbf{h}_{v_i}^K \\ \text{Promedio}(H^K) &= \frac{\sum_{i=1}^{n} \mathbf{h}_{v_i}^K}{|G|} \\ \text{Max}(H^K) &= \mathbf{max}\{\mathbf{h}_{v_i}^K : i \in \{1, 2, ..., n\}\} \end{aligned} \tag{49}$$

Por lo tanto, el método de agregación de Choquet puede ser aplicado de la siguiente manera:

$$C_\mu(H^K) = \sum_{i=1}^{n} \hat{C}_\mu(\mathbf{h}_{v_i}^K) \times \mathbf{h}_{v_i}^K \tag{50}$$

Donde μ es una medida de Sugeno con pesos aprendibles por la red neuronal. Es posible inicializar los

pesos de tal forma que su suma se aproxime a 1. Esto permite que, a través del proceso de entrenamiento, los pesos converjan hacia uno de los extremos, detectando así un tipo particular de sinergia que mejore el rendimiento de la red neuronal. Además, es importante destacar que el resultado de la agregación no depende únicamente de H^K, sino también de los pesos $\mathbf{p} = [p_1, p_2, ..., p_r]$ que definen el valor de λ que hace que μ sea una medida de Sugeno. Por lo tanto, se puede escribir $C_\mu(H^K, \mathbf{p})$ para enfatizar que los pesos variarán a medida que se ajusten durante la propagación del error en el proceso de entrenamiento de la *GNN*.

3.3.3. Pseudocódigos para implementación del método de agregación de Choquet en GNNs

En el Algoritmo 2 se ve una implementación general del entrenamiento de una red neuronal de K capas para la clasificación de grafos. Se puede ver en la *línea 5* del algoritmo que hay que determinar el método de agregación que se pretenda utilizar, el cual puede ser cualquiera de los antes mencionados.

Algoritmo 2: Entrenamiento de una red neuronal para clasificación de grafos utilizando el método de agregación de Choquet

Input : Inicializar(GNN)
Output: Modelo de *GNN* entrenado utilizando el método de agregación de Choquet

```
1 for epoch = 1, 2, ..., E do
2     for G_1, ..., G_N do
3         X, y ← preprocess(G_i);
4         H^K ← forward(GNN^K, X);
5         a ← aggregation(H^K);
6         l ← loss(a, y);
7         grad ← backward(l);
8         update(net,grad)
```

Con el fin de destacar las particularidades del método de agregación de Choquet, se presenta a el Algoritmo 3 que muestra explícitamente los componentes necesarios para su implementación.

Es importante destacar algunos elementos del Algoritmo 3:

- En la línea 5, la función $\text{Sugeno}_\mu(\mathbf{p})$ determina numéricamente el valor de λ que convierte μ en una medida de Sugeno. Como la función a aproximar es creciente y tiene una única solución, el método de la secante es una buena opción para calcular el valor de λ (Burden & Faires, 1989).
- La línea 6 muestra el método de agregación de Choquet en función de H^k, $\mathbf{p}$ y λ.

- Es necesario inicializar el vector de pesos $\mathbf{p}$ como parámetros aprendibles en la red. Como se puede observar en la línea 9, además de los parámetros de GNN^k, $\mathbf{p}$ también se actualiza mediante la propagación del error. Esto hace que los valores de $\mathbf{p}$ se ajusten para reducir el error en el aprendizaje.
- La dimensionalidad de H^k debe ser $|V_{G_i}| \times r$, coincidiendo con el número de pesos aprendibles r del vector $\mathbf{p}$.

Algoritmo 3: Entrenamiento de una red neuronal para clasificación de grafos con el método de agregación de Choquet

Input : Inicializar(GNN^K); Inicializar vector de pesos aprendibles $\mathbf{p} = [p_1, p_2, ..., p_r]$ donde $p_k \in [0, 1]$.
Output: Modelo de *GNN* entrenado

```
1 for epoch = 1, 2, ..., E do
2     for G_1, ..., G_N do
3         X, y ← preprocess(G_i);
4         H^K ← forward(GNN^K, X);
5         λ ← Sugeno_μ(p);
6         a_ch ← C_μ(H^K, p, λ) ;
7         l ← loss(a_ch, y);
8         grad ← backward(l);
9         update(GNN^K, p, grad)
```

Durante la presentación del método de agregación de Choquet, se mencionó que su aplicación implica un ordenamiento de la matriz a la cual se aplica, lo que puede generar un costo computacional elevado cuando las dimensiones de la matriz son grandes. En el caso específico de redes neuronales basadas en grafos, no es recomendable reducir la cantidad de nodos del grafo y, por tanto, el tamaño de su matriz de características asociada. Por ello, es preferible reducir la dimensionalidad de la matriz $H^K \in \mathbb{R}^{n \times r}$, obteniendo una matriz H'^K con dimensiones $n \times t$, donde t es considerablemente menor que r, lo cual se puede lograr mediante una matriz $T \in \mathbb{R}^{r \times t}$ haciendo $H'^K = H^K T$. En consecuencia, se propone una modificación en el Algoritmo 4 que incluye esta reducción de la dimensionalidad de la matriz H^K.

Es importante destacar que la matriz T se inicializa con pesos aleatorios, lo que permite su aprendizaje durante el entrenamiento de la red neuronal.

3.4. Método de agregación de Choquet en PyTorch Geometric

Habiendo revisado el fundamento teórico que lleva a la ecuación 50 del método de agregación de Choquet, ahora nos enfocaremos en su implementación en el entorno de PyTorch Geometric. En esta sección,

exploraremos cómo implementar de manera adecuada el método de agregación de Choquet en el entrenamiento de una red neuronal basada en grafos para la clasificación de grafos. Para lograr esto, definiremos un modelo y su arquitectura correspondiente que servirán como ejemplo, permitiéndonos identificar con claridad la ubicación de cada componente y las características fundamentales para el correcto funcionamiento del método de agregación de Choquet.

Algoritmo 4: Entrenamiento de una red neuronal para clasificación de grafos con el método de agregación de Choquet con reducción de H^K

Input : Inicializar(GNN^K); Inicializar matriz de reducción $T \in \mathbb{T}^{r \times t}$; Inicializar vector de pesos aprendibles $\mathbf{p} = [p_1, p_2, ..., p_t]$ donde $p_k \in [0, 1]$.
Output: Modelo de *GNN* entrenado

for $epoch = 1, 2, ..., E$ **do**
  for $G_1, ..., G_N$ **do**
    $X, y \leftarrow$ preprocess(G_i);
    $H^K \leftarrow$ forward(GNN^K, X);
    $H'^K \leftarrow H^K T$;
    $\lambda \leftarrow$ Sugeno$_\mu(\mathbf{p})$;
    $a_{ch} \leftarrow C_\mu(H'^K, \mathbf{p}, \lambda)$;
    $l \leftarrow$ loss(a_{ch}, y);
    grad $\leftarrow$ backward(l);
    update(GNN^K, T, $\mathbf{p}$, grad)

3.4.1. Desarrollo del método de agregación de Choquet a partir de un ejemplo

El siguiente fragmento de código, basado en el ejemplo de la subsección 2.4.4.2, define una red compuesta por dos capas GATConv seguidas de una capa Lineal. No obstante, en esta ocasión, se han incorporado las funciones esenciales para la implementación del método de agregación de Choquet. A lo largo de la explicación, haremos referencia frecuente a este fragmento de código, por lo que resulta conveniente referirnos a él como *código*, lo que permitirá evitar introducir un número elevado de referencias y mejorar la claridad en la redacción.

```
class GNN(torch.nn.Module):
    def init(self):
        self.conv1 = GATConv(dataset.num_features, 16)
        self.conv2 = GATConv(16, 16)
        self.lin = Linear(16, dataset.num_classes)
        self.p  = nn.Parameter(torch.tensor([0.06]).repeat(16))

    def forward(self, x, edge_index, batch):
```

```
        x = self.conv1(x, edge_index).relu()
        x = self.conv2(x, edge_index).relu()
        lamb = root(sugeno_function, self.p)
        x = choquet_aggregation(x, lamb, self.p, batch)
        return self.lin(x)
```

Obsérvese las siguientes partes añadidas en la definición del modelo y su arquitectura, las cuales se explicarán detalladamente en la subsección 3.4.1.2.

- `self.p`: Este elemento corresponde al vector **p** (en este caso, tensor) de pesos que se aprenderán y se utilizarán para determinar una medida de Sugeno óptima en el método de agregación de Choquet.
- `lamb = root(sugeno_function, self.p)`: Este componente representa el cálculo del valor de λ mediante un método numérico `root` que busca la raíz de la ecuación 34 utilizando los elementos de `self.p`.
- `choquet_aggregation(x, lamb, self.p, batch)`: Este elemento hace referencia al método de agregación de Choquet. A continuación, se detallan sus parámetros:
  - Cabe destacar que toma como entrada la salida de la última capa de la red (`self.conv2`), que en este caso es `x`, pero en el contexto de la subsección 3.3.2 correspondería a H^2.
  - Asimismo, recibe `lamb` y `self.p`, los cuales se utilizarán para calcular la función de medida μ.
  - Además, se suministra el `batch` asociado a cada uno de los grafos.

Los componentes mencionados constituyen los elementos fundamentales para la implementación del método de agregación de Choquet. En las siguientes secciones, se profundizará en estos aspectos y se presentará el código correspondiente a su implementación.

3.4.1.1. Grafo Computacional del método de agregación de Choquet

Para llevar a cabo una implementación adecuada del método de agregación de Choquet, es fundamental considerar el grafo computacional involucrado en una red neuronal basada en grafos. En la Figura 4,

se puede apreciar el grafo computacional asociado al *código* discutido en la sección 3.4.1. Este grafo representa el flujo de la matriz de características X en el método `forward` del *código*.

Es importante recordar que, en PyTorch, los únicos parámetros que pueden ser aprendidos son aquellos que corresponden a nodos hoja en el grafo computacional. Esto se debe a que PyTorch no almacena los gradientes que no pertenecen a este tipo de nodos. En el contexto de nuestro método, cabe destacar que $\mathbf{p}$ se incorpora al proceso durante el cálculo del método de agregación de Choquet. Por consiguiente, es susceptible de ser aprendido, ya que no forma parte del grafo computacional hasta dicha etapa y, por lo tanto, se trata de un nodo hoja.

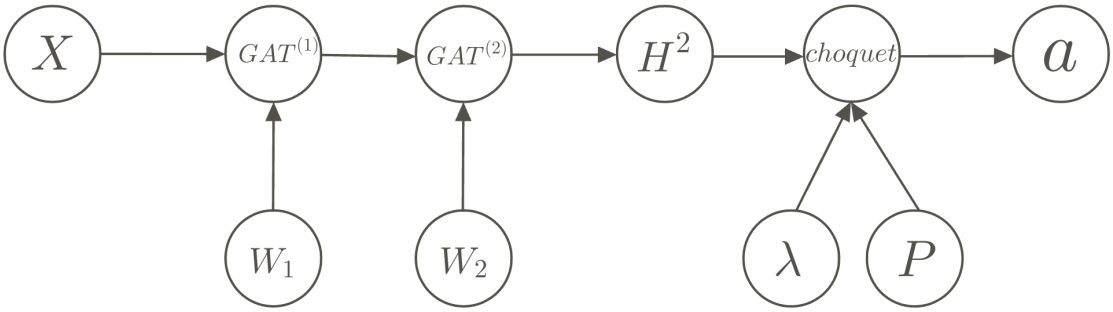

Figura 4. Grafo computacional del método `forward`.

Por otro lado, la Figura 5 muestra el grafo computacional correspondiente al cálculo del valor de λ, en el cual se emplea una función `root` que depende de $\mathbf{p}$. A primera vista, podría parecer apropiado conectar el último nodo de este grafo al nodo λ del grafo computacional presentado en la Figura 4. Sin embargo, esto provocaría que $\mathbf{p}$ se convierta en un nodo hoja en el grafo computacional en dos ocasiones. Como resultado, el cálculo del descenso del gradiente afectaría a $\mathbf{p}$ dos veces, lo cual resulta indeseable. Por lo tanto, es necesario realizar el cálculo de λ fuera del grafo computacional del método `forward`. Esto se puede lograr enviando una copia de $\mathbf{p}$ a la función `root`, en lugar de utilizar la versión empleada para el cálculo del método de agregación de Choquet.

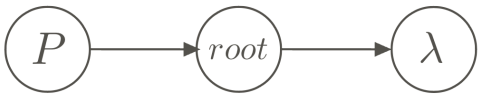

Figura 5. Grafo computacional del cálculo de λ.

3.4.1.2. Pesos aprendibles: `self.p`

Hasta ahora, hemos examinado el grafo computacional del *código* de la implementación del método de agregación de Choquet y hemos determinado que $\mathbf{p}$ es un nodo hoja, lo cual lo hace susceptible de ser aprendido. En esta sección, nos enfocaremos en cómo lograr esto.

Es importante señalar que $\mathbf{p}$ ha sido introducido como un atributo de la red denominado `self.p`. Lo primero que debemos considerar es que `self.p` debe tener la misma cantidad de características que la última capa de la red, la cual, en el *código*, es de 16. Por ahora, no nos centraremos en definir los valores iniciales de `self.p`, ya que esto se abordará en una sección posterior. Para inicializar `self.p`, podríamos sentirnos tentados a crear un tensor mediante el método `torch.tensor()` con su argumento `requires_grad=True`. En teoría, al ser un nodo hoja en el grafo computacional, esto permitiría que se aprenda mediante el método del descenso del gradiente. Sin embargo, PyTorch Geometric ya utiliza una función especial para introducir parámetros aprendibles dentro de la definición del modelo de red. Esto se logra mediante la función `nn.Parameter()`, que toma como argumento un tensor creado a través de la función `torch.tensor()` sin especificar `requires_grad`. No obstante, el resultado obtenido al emplear esta función sí termina siendo un tensor con `requires_grad=True`.

Las ventajas de utilizar este enfoque radican en que, al convertir un tensor en un objeto `nn.Parameter`, automáticamente se registra como un parámetro del módulo al cual pertenece. De esta manera, se garantiza una gestión adecuada de los parámetros aprendibles en el modelo de red, permitiendo un mejor control y seguimiento durante el proceso de aprendizaje.

3.4.1.3. Función de Sugeno: `sugeno_function`

Se ha mencionado previamente que, dado un vector de pesos $\mathbf{p}$ con n elementos, para hallar el valor de λ que convierte a μ en una medida de Sugeno, debemos resolver la siguiente ecuación con respecto a λ:

$$\lambda + 1 = \prod_{i=1}^{n}(1 + \lambda p_i) \tag{51}$$

Resolver esta ecuación es equivalente a encontrar la raíz de la siguiente función que a partir de ahora

definiremos como *función de Sugeno*:

$$f(x) = \prod_{i=1}^{n}(1 + xp_i) - x - 1 \tag{52}$$

En la sección 3.4.1.1, se discutió la importancia de mantener el proceso de obtención del valor de λ fuera del grafo computacional. Por razones que se expondrán en la siguiente sección, se ha optado por implementar la función de Sugeno utilizando la biblioteca `Numpy`. Por consiguiente, es necesario tener en cuenta que los parámetros recibidos serán objetos de tipo `numpy.array`. Habiendo realizado esta aclaración, podemos proceder a implementar la función de Sugeno en Python, utilizando la biblioteca Numpy de la siguiente manera:

```
def sugeno_function(lamb,p):
  product = np.prod(1+lamb*p)
  return (product - lamb - 1)
```

En este código, `np.prod` es una función de la biblioteca `Numpy` que calcula el producto de todos los elementos en un arreglo. La función `sugeno_function` toma como argumentos `lamb`, que representa a λ, y `p`, el vector de pesos $\mathbf{p}$. La función `np.prod` se utiliza para calcular el producto $\prod_{i=1}^{n}(1 + \lambda p_i)$ y, posteriormente, se retorna el resultado de la función de Sugeno: $(product - \lambda - 1)$, como se espera.

3.4.1.4. Método numérico: `root`

El método numérico utilizado para resolver la función de Sugeno es el método de Newton (Burden & Faires, 1989), que presenta una rápida convergencia cuando se elige una aproximación inicial cercana a la raíz de la función aplicada. De acuerdo con la sección 2.3.5, sabemos que, dado $\mathbf{p}$, podemos determinar la ubicación de la raíz de la función de Sugeno en función de si $\sum_{i=1}^{n} p_i > 1$ o $\sum_{i=1}^{n} p_i < 1$. Además, esta solución es única en $[-1, 0) \cup (0, \infty)$, aunque el 0 también es solución, no nos interesa en este caso. Teniendo esto en cuenta, el método de Newton resulta ser una opción atractiva en este contexto debido a sus características de rápida y eficiente convergencia.

A continuación, se muestra el código de la implementación del método de Newton, el cual ha sido desarrollado utilizando funciones de la biblioteca Numpy:

```
def root(f, p, max_iter=100, tol=1e-6):
    if p.sum() > 1:
      x_ini = -1.0
```

```
    elif p.sum() > 0 and p.sum() <= 0.60:
      x_ini = 100.0
    elif p.sum() < 1 and p.sum() > 0.60:
      x_ini = 2.0
    x0 = x_ini
    gradient_func = grad(f, argnum=0)
    for _ in range(max_iter):
        fx0 = np.nan_to_num(f(x0,dt))
        dfx0 = np.nan_to_num(gradient_func(x0, p))
        if dfx0 == 0:
            x0 = 0.6171
            continue
        x1 = x0 - fx0 / dfx0
        if np.abs(x1 - x0) < tol:
            break
        x0 = x1
    return torch.tensor(x0)
```

Es importante señalar que, en la función `root`, se adapta el método de Newton al contexto del método de agregación de Choquet. Por lo tanto, no acepta como parámetro una aproximación inicial a la raíz de la función `f`, dado que se espera que dicha función corresponda a la función de Sugeno. A continuación, se describen detalladamente los componentes que conforman la función `root`:

1. Se determinan las condiciones iniciales `x_ini` en función del valor de `p.sum()`. Estos casos se explicarán en detalle después de la explicación completa de la función `root`.

2. Se asigna el valor de `x_ini` a `x0`, que será la aproximación inicial de la raíz.

3. Se define `gradient_func` como la función que calcula la derivada de `f` respecto a su primer argumento, utilizando la función `grad` de la biblioteca Autograd de Numpy. Inicialmente, se consideró integrar Autograd de PyTorch para calcular la derivada de la función de Sugeno, pero la implementación de dos grafos computacionales en el mismo entorno generaba múltiples conflictos. Por esta razón, se decidió desarrollar el método numérico utilizando objetos de Numpy en lugar de emplear PyTorch.

4. Se ejecuta un bucle `for` que itera hasta un máximo de `max_iter` veces (por omisión, 100).

a) Se evalúa la función `f` en `x0` y se almacena el resultado en `fx0`. Es importante destacar que la función de Sugeno es un polinomio cuyo grado coincide con la cantidad de elementos en `p`. Así, si `p` tiene 128 elementos y `x0` > 1, rápidamente se obtienen valores grandes para `fx0`. En caso de que el resultado sea `Inf`, se convierte `fx0` al valor finito más grande disponible en Numpy utilizando la función `np.nan_to_num`.

b) Se evalúa `gradient_func` en `x0`, para obtener la derivada de `f` en `x0`, y se almacena el resultado en `dfx0`. Al igual que antes, se utiliza la función `np.nan_to_num` para evitar posibles problemas con valores infinitos. Si el resultado es 0, se asigna un valor de 0.6171 a `x0` y se continúa con la siguiente iteración. La justificación de esta asignación será explicada en breve.

c) Se calcula el valor de `x1` como `x0 - fx0 / dfx0`, que es la fórmula de actualización del método de Newton.

d) Se comprueba si la diferencia absoluta entre `x1` y `x0` es menor que la tolerancia `tol` (por omisión, 1e-6). Si se cumple esta condición, se rompe el bucle, ya que se ha alcanzado la convergencia.

e) Si no se ha alcanzado la convergencia, se actualiza el valor de `x0` con `x1` y se continúa con la siguiente iteración.

5. Finalmente, se devuelve el valor de `x0` como un tensor de PyTorch.

Retomando el punto 1 de la lista anterior, existen tres aproximaciones iniciales según el intervalo en el que se encuentre `p.sum()`. A continuación, se explicarán los detalles y razones detrás de la división de estos intervalos:

- Si `p.sum()` $\in (1, \infty)$, esto implica que la solución de la función de Sugeno se encuentra en el intervalo $[-1, 0]$. Dado que 0 también es una solución, se busca estar más cerca de la otra solución distinta de cero. Por lo tanto, -1 representa la aproximación más cercana a la raíz diferente de cero en este caso.

- Si `p.sum()` $\in (0, 0.6]$, esto implica que la solución de la función de Sugeno se encuentra en el intervalo $(0, \infty)$. Sin embargo, cuando `p.sum()` se aproxima a 0, la raíz adopta valores cada vez más grandes. Si hacemos $\sum_{i=1}^{n} p_i = q$, y denotamos a λ como la solución de la función de Sugeno, podemos expresar esta idea matemáticamente de la siguiente manera:

$$\lim_{q \to 0} \lambda = \infty \tag{53}$$

Por lo tanto, se propone una solución inicial para el caso en que `p.sum()` $\in (0, 0.60]$, utilizando `x_ini = 100` como punto de partida.

- Si `p.sum()` $\in [0.6, 1)$, se observa que, a medida que la suma se acerca a 1 por la izquierda, el valor de λ tiende a acercarse a cero. Al igual que en el punto anterior, esto se puede expresar matemáticamente de la siguiente manera:

$$\lim_{q \to 1^-} \lambda = 0 \tag{54}$$

En este caso, la aproximación inicial `x_ini = 2` se encuentra relativamente cerca de la solución distinta de 0. Retomando la observación en 4*b* sobre ajustar `x0` a 0.6171, esto ocurre debido a que, en alguna iteración, se tomó como siguiente aproximación el valor de 0. Por lo tanto, resulta conveniente retroceder en la aproximación para considerar la solución distinta de 0 de nuevo.

3.4.1.5. Método de agregación de Choquet: `choquet_aggregation`

A continuación, se presenta la implementación en PyTorch del método de agregación de Choquet. A diferencia de los ejemplos previamente mencionados, en un entorno como PyTorch se espera que todas las operaciones se realicen sobre un tensor completo, en contraste con los cálculos manuales que requieren enfocarse en cada elemento individualmente. Por esta razón, se optó por explicar el funcionamiento de cada parte del código del método de agregación al mismo tiempo que se muestra cómo estas funciones operarían en un ejemplo específico. De esta manera, se busca aclarar el funcionamiento del método en su totalidad.

El código del método de agregación de Choquet, `choquet_aggregation(x, p, lamb, batch)`, recibe cuatro parámetros. El tensor `x` representa la matriz de características a la que se aplica la agregación, el tensor `p` contiene los pesos asociados a cada una de las características de `x`, el valor de `lamb` corresponde al valor de λ necesario para formar una medida de Sugeno con los pesos `p`, y el tensor `batch` se utiliza para asociar cada grafo a su correspondiente lote (batch) dentro del entrenamiento de la red neuronal basada en grafos.

El código del método de agregación de Choquet es el siguiente:

```
def choquet_aggregation(x, p, lamb, batch):

    choquet = torch.zeros(x.shape[0])
```

```
    ind = torch.argsort(x, descending=True)
    product = 1 + lamb * p
    product_sorted = product[ind]

    aux = x.gather(1, ind).roll(shifts=-1, dims=1)
    aux[:, -1] = 0
    choquet = ((x.gather(1, ind) - aux) * (product_sorted.cumprod(1) - 1)
               /lamb).sum(1) #Esto es continuacion de la linea de arriba

    choquet_softmax = softmax(choquet, batch)

    x_product = choquet_softmax.view(-1, 1) * x
    x_aggregated = scatter_add(x_product, batch, dim = 0)

    return x_aggregated
```

A continuación, explicaremos el código del método de agregación de Choquet junto con un ejemplo para ilustrar cómo se aplican las líneas de código en casos particulares. En este ejemplo, consideraremos dos grafos que pertenecen al mismo lote, ambos con dos nodos cada uno. Recordemos que sus matrices (en este caso, tensores) de características se apilan como se vio en la sección 2.4.3.3. Por lo tanto, el tensor `x` resultante contiene 4 nodos (filas) y 3 características (columnas). Cada una de estas características tendrá un peso asociado, que corresponderá respectivamente con las columnas del tensor `p`. El valor de `lamb` es el valor de λ que nos permite crear una medida de Sugeno con los elementos en `p`; este valor ya se ha calculado utilizando la función `root`. El tensor `batch` indica a qué grafo pertenece cada uno de los nodos del tensor `x`. Los valores de estos tensores son los siguientes:

$$\texttt{x} = \begin{bmatrix} 45 & 50 & 40 \\ 56 & 35 & 50 \\ 39 & 58 & 55 \\ 58 & 38 & 57 \end{bmatrix}, \quad \texttt{p} = \begin{bmatrix} 0.45 & 0.45 & 0.3 \end{bmatrix}, \quad \texttt{lamb} = -0.4492, \quad \texttt{batch} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \tag{55}$$

Ahora procedemos a explicar el código del método de agregación de Choquet utilizando los tensores de ejemplo mencionados previamente.

1. Se crea un tensor de ceros `choquet` con la misma forma que el tensor `x` de entrada.

$$\texttt{choquet} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{56}$$

2. Se ordenan los elementos del tensor de entrada (`x`) en orden descendente, almacenando los índices en `ind`.

$$\texttt{ind} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 2 & 1 \\ 1 & 2 & 0 \\ 0 & 2 & 1 \end{bmatrix} \tag{57}$$

   Note que el tensor de índices `ind` ordena los elementos de cada fila de `x` en orden descendente, haciendo referencia a la posición que cada elemento del tensor `x` debe ocupar para estar ordenado.

3. Se calcula el tensor `product` sumando 1 a cada elemento del tensor `p` multiplicado por el escalar `lamb`.

$$\texttt{product} = 1 + (-0.4492) \begin{bmatrix} 0.45 & 0.45 & 0.3 \end{bmatrix} = \begin{bmatrix} 0.7979 & 0.7979 & 0.8652 \end{bmatrix} \tag{58}$$

   Obsérvese que estos productos corresponden a los términos presentes en la ecuación 52.

4. Se ordena el tensor `product` utilizando los índices en `ind`, almacenando el resultado en `product_sorted`.

$$\texttt{product\_sorted} = \begin{bmatrix} 0.7979 & 0.7979 & 0.8652 \\ 0.7979 & 0.8652 & 0.7979 \\ 0.7979 & 0.8652 & 0.7979 \\ 0.7979 & 0.8652 & 0.7979 \end{bmatrix} \tag{59}$$

   Nótese que el tensor `product` se ha expandido para incluir los pesos asociados a cada característica, de acuerdo con el orden de los valores de las características en cada fila.

5. Se aplica la operación `gather`, que sirve para ordenar a `x` con los índices en `ind`, y se desplaza cíclicamente cada fila una posición hacia la izquierda utilizando la función `roll`. Se asigna 0 a la última columna del tensor resultante `aux`.

$$
\begin{aligned}
\texttt{x.gather(1, ind)} &= \begin{bmatrix} 50 & 45 & 40 \\ 56 & 50 & 35 \\ 58 & 55 & 39 \\ 58 & 57 & 38 \end{bmatrix}, \\
\texttt{aux} &= \begin{bmatrix} 45 & 40 & 50 \\ 50 & 35 & 56 \\ 55 & 39 & 58 \\ 57 & 38 & 58 \end{bmatrix} \\
aux[:, -1] = 0 \rightarrow \texttt{aux} &= \begin{bmatrix} 45 & 40 & 0 \\ 50 & 35 & 0 \\ 55 & 39 & 0 \\ 57 & 38 & 0 \end{bmatrix}
\end{aligned}
\tag{60}
$$

Este procedimiento se lleva a cabo con el propósito de calcular las diferencias entre los valores ordenados presentes en la definición de la integral discreta de Choquet, tal como se muestra en la ecuación 26.

6. La integral discreta de Choquet se calcula y se almacena en el tensor `choquet`. Para lograr esto, se multiplica la diferencia entre el tensor `x` ordenado y `aux` por el resultado de dividir la suma acumulada de `product_sorted` menos 1 entre `lamb` (cabe destacar que esta parte corresponde a la ecuación 52). Posteriormente, se suman los elementos a lo largo del eje 1 (columnas).

$$
\begin{aligned}
\texttt{x.gather(1, ind)-aux} &= \begin{bmatrix} 5 & 5 & 40 \\ 6 & 15 & 35 \\ 3 & 16 & 39 \\ 1 & 19 & 38 \end{bmatrix} \\
\texttt{(product\_sorted.cumprod(1) - 1)/lamb} &= \begin{bmatrix} 0.45 & 0.8090 & 1 \\ 0.45 & 0.6894 & 1 \\ 0.45 & 0.6894 & 1 \\ 0.45 & 0.6894 & 1 \end{bmatrix}
\end{aligned}
\tag{61}
$$

haciendo `pre_choq = (x.gather(1, ind) - aux) * (product_sorted.cumprod(1) - 1)/lamb`

$$\texttt{pre\_choq} = \begin{bmatrix} 2.25 & 4.04 & 40.00 \\ 2.70 & 10.34 & 35.00 \\ 1.35 & 11.03 & 39.00 \\ 0.45 & 13.10 & 38.00 \end{bmatrix} \tag{62}$$

luego

$$\texttt{choquet} = \texttt{pre\_choq.sum(1)} = \begin{bmatrix} 46.29 \\ 48.04 \\ 51.38 \\ 51.54 \end{bmatrix} \tag{63}$$

7. Se aplica la función softmax a los elementos de `choquet` agrupados por el tensor `batch`, almacenando el resultado en `choquet_softmax`.

$$\texttt{softmax(choquet, batch)} = \begin{bmatrix} 0.1487 \\ 0.8513 \\ 0.4581 \\ 0.5419 \end{bmatrix} \tag{64}$$

   Obsérvese que la suma de las filas correspondientes al mismo `batch` es igual a 1.

8. Se multiplica el tensor `choquet_softmax` (cambiado de forma para que tenga la misma forma que `x`) por el tensor de entrada `x`, para obtener `x_product`.

$$\texttt{x\_product} = \begin{bmatrix} 0.1487 \\ 0.8513 \\ 0.4581 \\ 0.5419 \end{bmatrix} \begin{bmatrix} 45 & 50 & 40 \\ 56 & 35 & 50 \\ 39 & 58 & 55 \\ 58 & 38 & 57 \end{bmatrix} = \begin{bmatrix} 6.68 & 7.43 & 5.94 \\ 47.67 & 29.79 & 42.56 \\ 17.86 & 26.56 & 25.19 \\ 31.43 & 20.59 & 30.88 \end{bmatrix} \tag{65}$$

   Es importante destacar que los coeficientes de Choquet, los cuales corresponden a los elementos en `choquet_softmax`, multiplican al tensor de características `x` original. Como resultado, se obtiene un porcentaje de las características originales.

9. A continuación, se suma este producto elemento a elemento de acuerdo con los índices del tensor `batch`, almacenando el resultado en `x_aggregated`. Como resultado de esto, terminamos con dos filas, cada una representando la suma de las características de los nodos en cada grafo.

$$\texttt{x\_aggregated} = \begin{bmatrix} 54.36 & 37.22 & 48.51 \\ 49.29 & 47.16 & 56.08 \end{bmatrix} \tag{66}$$

A partir de esto, se pueden realizar diversas interpretaciones. Nótese que la primera fila de `x_aggregated` corresponde a la agregación de las dos primeras filas de `x_product`. La primera de ellas tenía un coeficiente de Choquet bastante bajo (0.1487) en comparación con la otra (0.8513). Como resultado, la agregación final termina siendo más parecida a las características del segundo nodo. En contraste, la agregación en la segunda fila de `x_aggregated` presenta coeficientes de Choquet más cercanos entre sí (0.4581 y 0.5419), lo que da lugar a una agregación más próxima al promedio entre las características de las últimas filas de `x_aggregated`. Es importante considerar que los coeficientes de Choquet dependen de los valores de `p`, los cuales se ajustarán a lo largo del proceso de entrenamiento de la red.

Capítulo 4. Experimentación y resultados

En este capítulo se describen los experimentos realizados y los resultados obtenidos para evaluar y comparar el método de agregación de Choquet frente al enfoque convencional de emplear el promedio como método de agregación en la clasificación de grafos aplicada a redes neuronales basadas en grafos.

A lo largo de este capítulo, se presentan todos los componentes involucrados en la experimentación con el objetivo de establecer una comparación justa entre los métodos de agregación. Posteriormente, expondremos los resultados obtenidos en cada uno de los experimentos mostrados en la Tabla 3. Además, se presenta un análisis estadístico de los resultados para concluir de manera precisa si existe una diferencia al emplear el método de agregación de Choquet. Posteriormente, se examinará la convergencia final del valor de λ utilizado por la medida de Sugeno en la integral discreta de Choquet, con el fin de determinar dónde se ubica dicho valor y establecer qué tipo de sinergia existe entre las características de los nodos. Finalmente, analizaremos si hay diferencias entre la utilización de distintos modelos de GNN empleando el promedio como método de agregación.

4.1. Objetivo de la experimentación y resumen del proceso experimental

4.1.1. Objetivo de la experimentación

El objetivo principal de nuestro estudio es comparar el método de agregación de Choquet contra el uso del promedio en la agregación de nodos para la clasificación de grafos usando redes neuronales en grafos (GNN). La ventaja distintiva del método de Choquet radica en su capacidad para tener en cuenta las interacciones entre las características de los nodos, lo que posibilita la creación de una jerarquía de los mismos y su utilización para fusionar la información del grafo en cuestión. Este enfoque podría ser útil en problemas de clasificación complejos, donde las características pueden no ser independientes entre sí debido a la conectividad del grafo, situación que la media aritmética no tiene mecanismos para reconocer.

Para llevar a cabo la comparativa, utilizaremos tres modelos de GNN: Graph Convolutional Networks (GCN) (Kipf & Welling, 2016), Graph Isomorphism Networks (GIN) (Xu et al., 2018) y Graph Attention Networks (GAT) (Veličković et al., 2018). Los experimentos consistirán en comparar cada uno de estos modelos utilizando el promedio como método de agregación frente al método de agregación de Choquet. Cada comparativa se evaluará en diferentes conjuntos de datos, analizando las métricas de rendimiento en diversos contextos.

4.1.2. Resumen del proceso experimental

El proceso experimental consistirá en entrenar y evaluar los modelos de GNN (GCN, GAT, GIN) empleando los dos métodos de agregación mencionados, en cinco bases de datos diferentes, lo que nos permitirá evaluar el rendimiento de los métodos de agregación en contextos diferentes. Para garantizar que el efecto de emplear cada método de agregación no se vea influenciado por diferencias en las arquitecturas, se considerará la realización de estas evaluaciones en arquitecturas equivalentes en términos de cantidad de capas y parámetros aprendibles.

A fin de evaluar de manera eficiente el desempeño de los métodos de agregación en cada conjunto de datos, se aplicará el método de 10-fold cross-validation. Este proceso se repetirá 10 veces para obtener resultados que puedan ser analizados estadísticamente. Los resultados de la experimentación se presentarán y analizarán en términos de la precisión (accuracy) para cada método de agregación y en cada conjunto de datos.

4.2. Conjuntos de casos de prueba utilizadas

Utilizamos un conjunto de casos de prueba extraídos de la TUDataset (Morris et al., 2020), una colección ampliamente utilizada de conjuntos de datos de grafos etiquetados. Los casos de prueba abarcan cinco conjuntos de datos principales: AIDS, COX2, ENZYMES, BZR y MUTAG. A continuación, se detallan estos conjuntos de datos, incluyendo sus referencias, la cantidad de grafos y sus características.

4.2.1. Características y propiedades de cada caso de prueba

4.2.1.1. AIDS

El conjunto de datos AIDS (Riesen & Bunke, 2008) proviene del Instituto Nacional del Cáncer y contiene grafos moleculares de compuestos químicos, donde cada nodo representa un átomo y las aristas representan enlaces químicos. Los grafos están etiquetados como activos o inactivos frente al VIH.

- Cantidad de grafos: 2000
- Clases: 2 (activo, inactivo)
- Nodos promedio por grafo: 15.69

- Aristas promedio por grafo: 16.20
- Nodos etiquetados: Sí (tipo de átomo)
- Atributos de los nodos: Sí, 4 atributos: (química, carga, coordenada X, coordenada Y)

4.2.1.2. COX2

El conjunto de datos COX2 (Helma et al., 2004) consta de grafos moleculares de inhibidores de la ciclooxigenasa-2 (COX-2), una enzima clave en el proceso inflamatorio. Al igual que en el conjunto de datos AIDS, los nodos representan átomos y las aristas enlaces químicos.

- Cantidad de grafos: 467
- Clases: 2 (activo, inactivo)
- Nodos promedio por grafo: 41.22
- Aristas promedio por grafo: 43.45
- Nodos etiquetados: Sí (tipo de átomo)
- Atributos de los nodos: Sí, 3 atributos: (coordenada X, Y, Z)

4.2.1.3. ENZYMES

El conjunto de datos ENZYMES (Feragen et al., 2013) representa enzimas como grafos, donde los nodos son las estructuras secundarias de las proteínas (hélices, láminas y giros) y las aristas representan las relaciones espaciales o secuenciales entre estas (Borgwardt et al., 2005). Las proteínas están clasificadas en seis clases en función de la Comisión de Nomenclatura de la Unión Internacional de Bioquímica y Biología Molecular (IUBMB).

- Cantidad de grafos: 600
- Clases: 6 (Número EC, según la Comisión de Nomenclatura de la IUBMB)
- Nodos promedio por grafo: 62.14

- Aristas promedio por grafo: 32.63
- Nodos etiquetados: Sí (tipo de aminoácido)
- Atributos de los nodos: Sí, de tamaño 18 que contienen medidas físicas y químicas, las cuales pueden ser revisadas en la referencia del conjunto de datos.

4.2.1.4. BZR

El conjunto de datos BZR (Kazius et al., 2008) comprende grafos moleculares de ligandos de la familia de receptores benzodiazepínicos. Los nodos representan átomos y las aristas enlaces químicos. Los grafos están etiquetados como activos o inactivos en función de su capacidad para unirse o no a los receptores.

- Cantidad de grafos: 405
- Clases: 2 (activo, inactivo)
- Nodos promedio por grafo: 35.75
- Aristas promedio por grafo: 38.36
- Nodos etiquetados: Sí (tipo de átomo)
- Atributos de los nodos: Sí, 3 atributos: (coordenada X, Y, Z)

4.2.1.5. MUTAG

El conjunto de datos MUTAG (Debnath et al., 1991) consiste en grafos moleculares de compuestos químicos llamados nitroaromáticos, con etiquetas que indican si son mutagénicos (causan mutaciones genéticas) o no. Los nodos representan átomos y las aristas enlaces químicos.

- Cantidad de grafos: 188
- Clases: 2 (mutagénico, no mutagénico)
- Nodos promedio por grafo: 17.93
- Aristas promedio por grafo: 19.79

- Nodos etiquetados: Sí (tipo de átomo)
- Atributos de los nodos: No, sin embargo, se genera un formato one-hot de las etiquetas de los nodos.

4.3. Diseño experimental

En esta sección, se abordan los aspectos relacionados con el diseño experimental destinado a evaluar el rendimiento del método de agregación de Choquet. Se detallan los experimentos a llevar a cabo, la implementación de la estrategia de validación cruzada y cómo serán analizados los resultados.

4.3.1. Experimentos a realizar

La Tabla 3 presenta las combinaciones de experimentos que se ejecutarán con el propósito de examinar la influencia del método de agregación de Choquet en diversos modelos de GNN. Al comparar el rendimiento de los modelos utilizando el promedio y el método de agregación de Choquet, se buscará establecer si existe un impacto significativo al aplicar este último en la clasificación de grafos.

4.3.2. Estrategia de validación cruzada

A continuación, se describirá el proceso de validación cruzada utilizado para la evaluación de los experimentos realizados. Para cada combinación de la Tabla 3 de experimentos proporcionada previamente, se llevaron a cabo 10 repeticiones de validación cruzada de 10 pliegues. En cada una iteración, se calculó el promedio de la precisión de los 10 pliegues, obteniendo así un único valor de precisión para cada iteración.

La división de los datos en los pliegues de la validación cruzada se realizó de manera aleatoria utilizando una semilla fija. Esta semilla garantiza que, para un mismo conjunto de datos, todos los experimentos se ejecutan sobre los mismos pliegues, asegurando una comparación justa y consistente entre ellos. Por ejemplo, los modelos GCN, GAT, GIN, GCN+CH, GAT+CH y GIN+CH, evaluados en el caso de prueba MUTAG, se entrenaron y evaluaron utilizando las mismas divisiones de datos generadas por la semilla fija.

La partición de los datos se realizó utilizando muestreo estratificado (learn developers, 2021), siguiendo la recomendación de Dwivedi et al. (2020), ya que las clases en cada conjunto de datos no siempre están balanceadas. De esta manera, cada subconjunto mantiene el mismo porcentaje de muestras de

cada clase en comparación con el conjunto completo. Esta estrategia asegura que los resultados de las diferentes combinaciones de modelos y métodos de agregación sean comparables y permitan evaluar de manera efectiva el efecto del método de agregación de Choquet.

Tabla 3. Combinaciones de experimentos de GNN y métodos de agregación

Modelo de GNN	Método de agregación	Casos de prueba
GCN	Promedio	MUTAG
		ENZYMES
		COX2
		BZR
		AIDS
	Choquet	MUTAG
		ENZYMES
		COX2
		BZR
		AIDS
GAT	Promedio	MUTAG
		ENZYMES
		COX2
		BZR
		AIDS
	Choquet	MUTAG
		ENZYMES
		COX2
		BZR
		AIDS
GIN	Promedio	MUTAG
		ENZYMES
		COX2
		BZR
		AIDS
	Choquet	MUTAG
		ENZYMES
		COX2
		BZR
		AIDS

4.3.3. Criterio de evaluación

Siguiendo las recomendaciones de Labatut & Cherifi (2012), en este estudio nos enfocaremos en analizar los resultados en función de la precisión (accuracy), ya que no nos centraremos en una clase específica al momento de clasificar en las distintas bases de datos. La elección de la precisión como medida de evaluación se basa en su simplicidad y su capacidad para proporcionar una evaluación general del desempeño del modelo. Además, al no requerir atención específica a una única clase, la precisión resulta

apropiada para comparar el rendimiento de los métodos de agregación en múltiples conjuntos de datos y clasificaciones.

4.3.4. Comparación de resultados

Para el análisis de los resultados, se emplearán medidas estadísticas descriptivas, representaciones gráficas y pruebas estadísticas adecuadas. Si se cumplen los supuestos de homocedasticidad y normalidad, se utilizará la prueba t de Student; en caso contrario, se aplicará la prueba de Wilcoxon o la prueba t de Welch. Estos análisis facilitarán la evaluación y comparación del desempeño de los modelos y métodos de agregación en relación con las distintas combinaciones experimentales.

4.4. Configuración de los modelos GNN

El código de este trabajo, el esquema de entrenamiento y los hiperparámetros de los modelos se basan en el trabajo de Dwivedi et al. (2020) del artículo *Benchmarking Graph Neural Networks*. En general, dicho trabajo consistió en generar un marco de referencia de código abierto (benchmark framework) que facilite el modelado, la implementación y la experimentación con modelos de GNN en diferentes conjuntos de datos de manera sencilla. Su desarrollo se basó principalmente en el paquete de Python llamado Deep Graph Library (DGL) (Wang et al., 2019b). Sin embargo, para este trabajo, se adaptó la misma funcionalidad pero utilizando PyTorch Geometric (Fey & Lenssen, 2019), ya que fue más sencillo modificar su código para implementar el método de agregación de Choquet.

El código fuente e instrucciones del programa de evaluación (benchmark) basado en la biblioteca Deep Graph Library (DGL) están disponibles en el repositorio de GitHub del proyecto[1]. Por otro lado, las referencias al código de la adaptación a PyTorch Geometric se encuentran en el Apéndice A.

4.4.1. Presupuesto de parámetros aprendibles

Para realizar una comparación adecuada entre distintos modelos de GNN, se emplea el enfoque sugerido por Dwivedi et al. (2020), que se basa en asignar un presupuesto similar de parámetros a cada uno de los modelos. De este modo, la selección del número de capas y sus dimensiones se realiza de tal forma que se ajuste al presupuesto establecido. En nuestro caso, dicho presupuesto se establece en aproximadamente 100,000 parámetros aprendibles.

[1] https://github.com/graphdeeplearning/benchmarking-gnns

El objetivo de utilizar este enfoque radica en evitar la necesidad de realizar una optimización exhaustiva de los hiperparámetros para cada modelo, proceso que resulta computacionalmente costoso (Errica et al., 2019). Establecer un presupuesto de parámetros resulta más adecuado dadas nuestras limitaciones computacionales. Por lo tanto, los hiperparámetros asignados para cada modelo de GNN se mantendrán en sus valores predeterminados, según los establecidos en el programa de evaluación por Dwivedi et al. (2020).

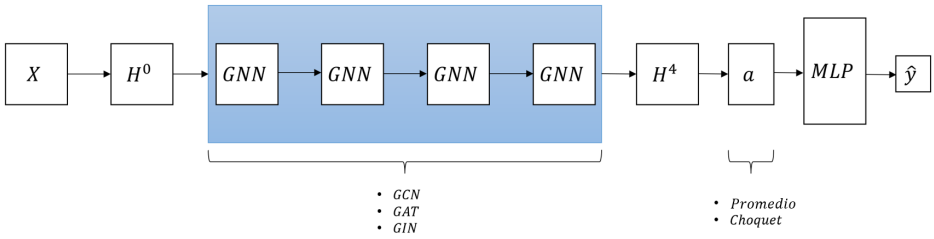

Figura 6. Arquitectura de los modelos de GNN

4.4.2. Arquitectura de cuatro capas y propagación hacia adelante de los modelos

Para ajustarse al presupuesto de 100,000 parámetros aprendibles, se establece una arquitectura equivalente que se aplica a todos los modelos de GNN. Esta arquitectura se especifica a través del método de propagación hacia adelante (forward) en la definición de los modelos. El flujo de la arquitectura es el siguiente y puede observarse en la Figura 6 para una mejor comprensión:

1. Se multiplica la matriz de características X del grafo por una matriz H^0 para que coincida en dimensiones con el tamaño de la capa inicial de GNN.

2. La información fluye a través de cuatro capas del mismo modelo de GNN, que pueden ser GCN, GAT o GIN. Además, entre cada capa del modelo de GNN, se aplica una función de activación de tipo *relu* para introducir no linealidad en el proceso de propagación de información.

3. La salida de la última capa devuelve la matriz H^4.

4. Se aplica el método de agregación (promedio o método de agregación de Choquet) a la matriz H^4, obteniendo como resultado a.

5. Finalmente, se introduce a en un perceptrón multicapa (Multilayer Perceptron) para obtener la predicción del modelo $\hat{y}$.

Es relevante a partir de este momento mostrar el código del método de propagación utilizado en la experimentación.

```
    def forward(self, x, edge_index, batch):
        # 1. H^{0} y las 4 capas de GNN
        x = self.embedding_h(x)
        x = self.layers(x, edge_index)
        # 2. Calculo del valor de lambda
        if self.readout == "choq":
            lamb = root( sugeno_function, self.choquet_weigth.detach().cpu
                ().numpy()) # Esto es continuacion de la linea de arrib
        # 3. Metodo de agregacion
        if self.readout == "mean":
            x = global_mean_pool(x, batch)
        elif self.readout == "choq":
            x = choquet_aggregation(x, self.choquet_weigth, lamb, batch)
        else:
            x = global_mean_pool(x, batch)
        # 4. Prediccion
        return self.MLP_layer(x)
```

En general, sigue el concepto del código presentado en la sección 3.4.1, pero con las siguientes diferencias notables:

- Se introduce la matriz H^0 a través de `self.embedding_h(x)`.
- En este código, se ha inicializado un atributo `self.layers` que contiene las 4 capas del modelo de GNN seleccionado y la función de activación a utilizar. Esto contrasta con el código presentado en la subsección 3.4.1.1 donde las capas se mantienen separadas y se especifica el uso de la función relu.
- Se menciona `self.choquet_weigth`, que es el vector de pesos asociado al método de agregación de Choquet, que previamente aparecía como `self.p`. Además, es importante destacar que se introduce en la función `root` como `self.choquet_weigth.detach().cpu().numpy()`, lo que lo desvincula del grafo computacional, en concordancia con lo mencionado en la subsección 3.4.1.1 sobre mantener separado el cálculo del valor de λ del grafo computacional.
- La presencia de un atributo llamado `self.readout`, que simplemente almacena el método de

agregación que se desea utilizar.

- Al finalizar, se puede apreciar en el `return` el uso del MLP para realizar la predicción $\hat{y}$.

El código completo de los modelos de GNN se encuentra disponible en el Apéndice B.

4.4.3. Esquema de entrenamiento

El esquema de entrenamiento que se utiliza consiste en emplear el optimizador Adam y una estrategia de decaimiento de la tasa de aprendizaje. En concreto, se selecciona una tasa de aprendizaje inicial, por ejemplo, 10^{-2}. Posteriormente, esta tasa se reduce a la mitad si la pérdida de validación no mejora después de un número fijo de épocas. En nuestro caso, este número se mantuvo constante en 25. Además, el número de épocas máximo de entrenamiento se fijó en 1000. No obstante, el entrenamiento se detiene cuando la tasa de aprendizaje alcanza un valor de 10^{-6}. En este contexto, definimos que el modelo ha alcanzado la convergencia cuando no se registran mejoras en la pérdida de validación una vez que la tasa de aprendizaje ha descendido a 10^{-6} y se han completado 25 épocas.

4.4.4. Selección de la tasa de aprendizaje inicial

De acuerdo con lo establecido en el esquema de entrenamiento, se debe elegir una tasa de aprendizaje inicial. Aunque no se tiene previsto ajustar los hiperparámetros de los modelos de GNN y se pretende mantener un presupuesto de parámetros aprendibles, la única excepción será la tasa de aprendizaje inicial. Esto se debe a que el resultado final puede depender en gran medida de la tasa de aprendizaje inicial (Dwivedi et al., 2020). Por lo tanto, se llevó a cabo una experimentación preliminar para seleccionar las tasas de aprendizaje adecuadas.

Esta experimentación consistió en realizar una iteración de validación cruzada de 10 pliegues para cada experimento en la Tabla 3 con tres tasas de aprendizaje iniciales diferentes: 0.005, 0.0005 y 0.00005. Se seleccionó la tasa de aprendizaje que produjo la mayor precisión en cada conjunto de prueba para cada experimento. Los resultados y la selección de las tasas de aprendizaje iniciales se pueden encontrar en el Apéndice C.

4.4.5. Parámetros aprendibles por modelo, conjunto de prueba y método de agregación

En la Tabla 4, se muestra la cantidad de parámetros aprendibles que tiene cada modelo de GNN según el conjunto de casos de prueba y la dimensión del vector de características en cada capa (referido en adelante como .[el] tamaño de las capas"). Cabe destacar que la cantidad de parámetros depende del tamaño de las capas y las características iniciales de los nodos, pero solo se ajusta la dimensión del vector de características en cada capa para alcanzar aproximadamente 100,000 parámetros aprendibles según nuestro presupuesto.

Tabla 4. Cantidad de parámetros aprendibles de los modelos de GNN por caso de prueba y método de agregación

	Modelo	Capas	Tamaño	Parámetros
MUTAG	GCN	4	145	100069
	GAT	4	160	106717
	GIN	4	110	107883
AIDS	GCN	4	142	101069
	GAT	4	152	101804
	GIN	4	105	102074
COX2	GCN	4	142	100501
	GAT	4	152	101196
	GIN	4	105	101654
ENZYM	GCN	4	144	100986
	GAT	4	152	98632
	GIN	4	105	99977
BZR	GCN	4	140	100347
	GAT	4	152	103932
	GIN	4	105	103544

(a) Modelos con media aritmética como método de agregación

	Modelo	Capas	Tamaño	Parámetros
MUTAG	GCN+CH	4	145	100214
	GAT+CH	4	160	106737
	GIN+CH	4	110	107993
AIDS	GCN+CH	4	142	101211
	GAT+CH	4	152	101823
	GIN+CH	4	105	102179
COX2	GCN+CH	4	142	100643
	GAT+CH	4	152	101215
	GIN+CH	4	105	101759
ENZYM	GCN+CH	2	144	101130
	GAT+CH	3	152	98651
	GIN+CH	4	105	100082
BZR	GCN+CH	3	140	100487
	GAT+CH	4	152	103951
	GIN+CH	5	105	103649

(b) Modelos con método de agregación de Choquet

Es importante señalar que la selección del tamaño de las capas en la Tabla 4a es la misma que en la Tabla 4b. Por lo tanto, la introducción del método de agregación de Choquet no agrega muchos parámetros aprendibles adicionales en comparación con el uso de la media aritmética como método de agregación. La única diferencia está en los pesos asociados a las características utilizadas en el método de agregación de Choquet. Por tanto, la única diferencia en la cantidad de parámetros aprendibles entre la Tabla 4a y la Tabla 4b radica en la adición del tamaño de las capas a los parámetros.

4.5. Recursos computacionales

Durante la realización de los experimentos para esta investigación, se contó con recursos computacionales proporcionados a través de una suscripción a Colab Pro+. Los recursos disponibles en la plataforma incluyen una CPU con 2 núcleos, una memoria RAM de 12.68 GB y una GPU Tesla T4 con 15.360 MiB de memoria y controlador versión 525.85.12. El sistema operativo utilizado fue Linux, específicamente la versión #1 SMP Sat Dec 10 16:00:40 UTC 2022.

Es importante mencionar que la ejecución de los experimentos no se llevó a cabo en un entorno de cómputo local, sino en la infraestructura en la nube proporcionada por Google Colab. Esta plataforma permite realizar cálculos intensivos y experimentos con grandes volúmenes de datos de manera más eficiente, aprovechando las ventajas de la computación en la nube y evitando la necesidad de adquirir y mantener hardware especializado.

La utilización de Colab Pro+ proporciona recursos adicionales en comparación con la versión gratuita de Google Colab, lo que permite agilizar los procesos de entrenamiento y evaluación de los modelos y algoritmos implementados en esta investigación. Además, el uso de esta plataforma en la nube facilita la colaboración y la reproducción de los experimentos, ya que los recursos y configuraciones se mantienen consistentes y accesibles para otros investigadores interesados en replicar o expandir este trabajo.

4.6. Procedimiento de experimentación

4.6.1. Semillas para la experimentación

En la subsección 4.3.2, se explicó que la división de los datos en los pliegues de la validación cruzada se realizó de forma aleatoria utilizando una semilla fija. Dado que hay 10 iteraciones de cada experimento, se requieren 10 semillas diferentes. Estas semillas se seleccionaron al azar de entre los números enteros del 1 al 100, y son las siguientes: 53, 36, 74, 94, 55, 88, 2, 40, 76 y 82.

4.6.2. Inicialización de los experimentos

En Colab de Google, los recursos no están garantizados en todo momento, por lo que la experimentación puede interrumpirse por causas ajenas al usuario. Por lo tanto, es necesario tener un método para guardar la experimentación en caso de algún imprevisto. Para ello, se generó un archivo `csv` que guarda el estado de la experimentación.

El archivo csv de estado se genera utilizando el siguiente código:

```
#Generar estatus del experimento
import pandas as pd
bases = ['MUTAG','ENZYMES', 'COX2', 'BZR', 'AIDS']
modelos = ['GCN', 'GAT', 'GIN']
choquet = [True,False]
seed = [53, 36, 74, 94, 55, 88, 2, 40, 76, 82]

data = []
contador = 1
for i in bases:
  for j in modelos:
    for h in choquet:
      for k in seed:
          lr = define_lr(j,i,h)
          data.append([i,j,lr,k,h,
                       contador,False,
                       0.0, 0.0, 0.0, 0.0,
                       0.0, 0.0, 0.0, 0.0])
          contador += 1
df = pd.DataFrame(data, columns=['base', 'model','lr','seed','choquet',
                                 'num','status','test_acc', 'test_sd',
                                 'train_acc', 'train_sd', 'conv_ep_mean',
                                 'conv_ep_sd', 'time', 'epc_time_mean'])
path_status = '/ruta/del/experimento/' + 'nombre_del_archivo.csv'
df.to_csv(path_status, index=False)
```

Tabla 5. csv del estatus del proyecto

base	model	lr	seed	choquet	num	status	test_acc	test_sd	train_acc	train_sd
MUTAG	GCN	0.005	53	True	1	False	0.0	0.0	0.0	0.0
MUTAG	GCN	0.005	36	True	2	False	0.0	0.0	0.0	0.0
MUTAG	GCN	0.005	74	True	3	False	0.0	0.0	0.0	0.0
MUTAG	GCN	0.005	94	True	4	False	0.0	0.0	0.0	0.0
MUTAG	GCN	0.005	55	True	5	False	0.0	0.0	0.0	0.0
MUTAG	GCN	0.005	88	True	6	False	0.0	0.0	0.0	0.0
MUTAG	GCN	0.005	2	True	7	False	0.0	0.0	0.0	0.0
MUTAG	GCN	0.005	40	True	8	False	0.0	0.0	0.0	0.0
MUTAG	GCN	0.005	76	True	9	False	0.0	0.0	0.0	0.0
MUTAG	GCN	0.005	82	True	10	False	0.0	0.0	0.0	0.0

En el código anterior, es importante destacar que al principio se definen listas que se utilizarán en los ciclos `for` anidados para generar las combinaciones de experimentos que se requieren realizar. Dentro del ciclo `for` más interno, se utiliza la función `define_lr` para definir la tasa de aprendizaje inicial según el caso de prueba, el modelo de GNN y su método de agregación. Después, las combinaciones de experimentos y otras variables, que se discutirán en un momento, se incorporan en la lista `data`.

Fuera del ciclo `for`, se genera un objeto `Pandas DataFrame` que contiene las siguientes variables:

- `base`: Nombre del conjunto de prueba utilizado en el experimento.
- `model`: Modelo de GNN utilizado en el experimento.
- `lr`: Tasa de aprendizaje inicial utilizada en el experimento.
- `seed`: Semilla utilizada en el experimento.
- `choquet`: Indicador de uso del método de agregación de Choquet. Es `True` si se utiliza y `False` si se utiliza la media aritmética.
- `num`: Número del experimento.
- `status`: Estado del experimento. Es `True` si el experimento se realizó con éxito y `False` en caso contrario.
- `test_acc`: Promedio de la precisión en el conjunto de prueba de los 10 pliegues del experimento.
- `test_sd`: Desviación estándar de la precisión en el conjunto de prueba de los 10 pliegues del experimento.
- `train_acc`: Promedio de la precisión en el conjunto de entrenamiento de los 10 pliegues del experimento.
- `train_sd`: Desviación estándar de la precisión en el conjunto de entrenamiento de los 10 pliegues del experimento.
- `conv_ep_mean`: Cantidad promedio de épocas necesarias para la convergencia en los 10 pliegues del experimento.
- `conv_ep_sd`: Desviación estándar de la cantidad de épocas necesarias para la convergencia en los 10 pliegues del experimento.

- time: Tiempo total empleado en el experimento.
- epc_time_mean: Tiempo promedio empleado por época.

En la Tabla 5, se presentan las 10 primeras filas del archivo csv destinado a almacenar la información relacionada con el experimento. El resto de las columnas, inicializadas en cero, se han omitido, ya que también se utilizarán para registrar progresivamente los resultados obtenidos en los experimentos.

4.6.3. Ejecución de los experimentos

Finalmente para ejecutar los experimentos se usan las siguientes líneas de código:

```
from time import strftime
path_status = '/ruta/del/experimento/' + 'nombre_del_archivo.csv'
df = pd.read_csv(path_status)
df_copy = df
for index, row in df.iterrows():
  if row['status'] == False:
    dirs, params, net_params = leer_model_base(row['model'],
                                               row['base'],
                                               row['seed'],
                                               row['choquet'],
                                               row['lr'])
    #Obtener el dataset-----------------------------------
    dataset = TUDataset(os.path.join('data', df_copy['base'][index]),
                        df_copy['base'][index],
                        use_node_attr=True)
    #Correr Experimento ----------------------------------
    te_acc, te_sd, tr_acc, tr_sd, conv_m, conv_sd, time_exp, time_epc =
       train_val_pipeline_exp(row['model'], row['base'], params,
       net_params, dirs, dataset)
    #Guardar datos ---------------------------------------
    df_copy.iloc[index,6] = True
    df_copy.iloc[index,7] = te_acc
    df_copy.iloc[index,8] = te_sd
    df_copy.iloc[index,9] = tr_acc
    df_copy.iloc[index,10] = tr_sd
    df_copy.iloc[index,11] = conv_m
```

```
    df_copy.iloc[index,12] = conv_sd
    df_copy.iloc[index,13] = time_exp
    df_copy.iloc[index,14] = time_epc

    df_copy.to_csv(path_status, index=False)
  else:
    pass
```

El código presentado tiene como objetivo leer el archivo de estatus previamente generado e itera sobre cada una de sus filas. La ejecución se llevará a cabo siempre que la variable `status` sea igual a `False`; de lo contrario, pasará al siguiente experimento. Dentro del bloque `if`, la función `leer_model_base` recibe las variables del estatus y se encarga de cargar todos los hiperparámetros del modelo de GNN a emplear, así como los directorios de salida donde se almacenarán los resultados.

Posterior a ello, se invoca la función `TuDataset` para descargar el conjunto de grafos que se desea aprender, almacenándolo en la variable `dataset`.

A continuación, se invoca la función `train_val_pipeline_exp`, que recibe el modelo de GNN, el nombre del conjunto de casos de prueba, los parámetros de aprendizaje, los parámetros del modelo, los directorios y el conjunto de grafos para el aprendizaje. Cabe destacar que esta función devuelve los resultados de las variables que corresponden a los espacios inicializados en cero en la Tabla 5. Finalmente, se almacenan los resultados en las columnas pertinentes y se actualiza el archivo `csv` del estatus.

Para obtener información adicional sobre el acceso de las funciones empleadas en esta sección, consulte el Apéndice A.

4.7. Comparativa de resultados entre la agregación de Choquet y el promedio: Análisis Descriptivo

Los resultados del experimento se presentan en una tabla para cada uno de los casos de prueba, que incluyen: MUTAG, BZR, ENZYMES, COX2 y AIDS. En cada tabla se comparan los modelos correspondientes en un mismo recuadro. Además, se evalúan las siguientes variables:

- Precisión promedio en los datos de prueba $\pm$ desviación estándar, la cual está denotada en la tabla como: Test Acc.$\pm$s.d.

- Precisión promedio en los datos de entrenamiento $\pm$ desviación estándar, la cual está denotada en la tabla como: Train Acc.$\pm$s.d.
- Cantidad promedio de épocas para alcanzar la convergencia $\pm$ desviación estándar, la cual está denotada en la tabla como: Epochs$\pm$s.d.
- Tiempo promedio (en horas) por experimento $\pm$ desviación estándar, la cual está denotada en la tabla como: Time$\pm$s.d.
- Tiempo total (en horas) empleado en cada una de las 10 iteraciones de la validación cruzada de 10 pliegues, la cual está denotada en la tabla como: Total Time.

Recordando, como se explicó en la subsección 4.4.3, consideramos que un modelo ha alcanzado la convergencia cuando no se observan mejoras significativas en la pérdida de validación después de varias épocas consecutivas y la tasa de aprendizaje ha alcanzado el valor de 10^{-6}.

4.7.1. Observación sobre la terminología usada en este capítulo

Es importante establecer la terminología que se utilizará este capítulo. En general, cuando se hable de un modelo de GNN sin especificar el método de agregación, se supone que se está utilizando el método de agregación promedio. En cambio, cuando se haga referencia al modelo de GNN con Choquet, se entenderá que se está utilizando el método de agregación de Choquet, o en su defecto se utilizará la notación GNN+CH.

4.7.2. Precisión y esfuerzo de cómputo en el caso de prueba MUTAG

La Tabla 6 muestra los resultados obtenidos en los casos de prueba del conjunto MUTAG. Aunque no hay diferencias significativas en la precisión promedio de los conjuntos de prueba entre los modelos, es notable que los modelos GCN y GAT con Choquet presentan una precisión ligeramente superior. Analizando los diagramas de cajas en la Figura 7, se puede apreciar que GCN con Choquet exhibe una dispersión de observaciones que apenas se superpone con su versión sin Choquet. En contraste, los resultados de GAT están superpuestos en ambos modelos. En cuanto a GIN, las diferencias en precisión son de 0.08, destacando únicamente la menor varianza en el modelo con Choquet, lo cual también es observable en la Figura 7.

Respecto a la precisión promedio en los datos de los conjuntos de entrenamiento, los modelos GCN y GAT con Choquet exhiben valores superiores, lo que podría explicar su mejor rendimiento en los datos de los conjuntos de prueba. En cambio, GIN muestra un rendimiento casi idéntico tanto en el conjunto de entrenamiento como en el de prueba.

Además, es relevante señalar que los modelos GCN y GAT con Choquet requieren, en promedio, menos épocas para alcanzar la convergencia. No obstante, esta diferencia no se refleja en un menor tiempo total para estos modelos. En cuanto al tiempo total de GIN, ambos modelos, con y sin Choquet, exhiben un comportamiento similar y son los que menos tiempo requieren. El hecho de que los modelos con Choquet tomen más tiempo, a pesar de converger más rapido en algunos casos, puede explicarse por la mayor demanda computacional que supone ordenar la matriz de características de los nodos en la última capa de la red. Estos detalles se explicarón en la subsección 3.3.3.

Tabla 6. Precisión y esfuerzo de cómputo en el caso de prueba MUTAG

	Modelo	Train Acc.±s.d	Test Acc.±s.d	Epochs±s.d	Time±s.d (hr)	Total Time (hr)
MUTAG	GCN	92.98 ± 1.03	81.70 ± 1.81	324.23 ± 25.14	0.08 ± 0.01	0.76
	GCN+CH	98.49 ± 0.24	83.80 ± 1.66	286.97 ± 11.73	0.09 ± 0.01	0.90
	GAT	82.13 ± 7.90	74.90 ± 7.01	431.55 ± 55.37	0.12 ± 0.02	1.25
	GAT+CH	91.00 ± 1.02	76.16 ± 3.03	389.19 ± 25.13	0.14 ± 0.01	1.43
	GIN	99.69 ± 0.29	85.25 ± 1.82	265.41 ± 5.53	0.07 ± 0.00	0.72
	GIN+CH	99.21 ± 0.28	85.17 ± 0.99	272.61 ± 13.62	0.08 ± 0.00	0.84

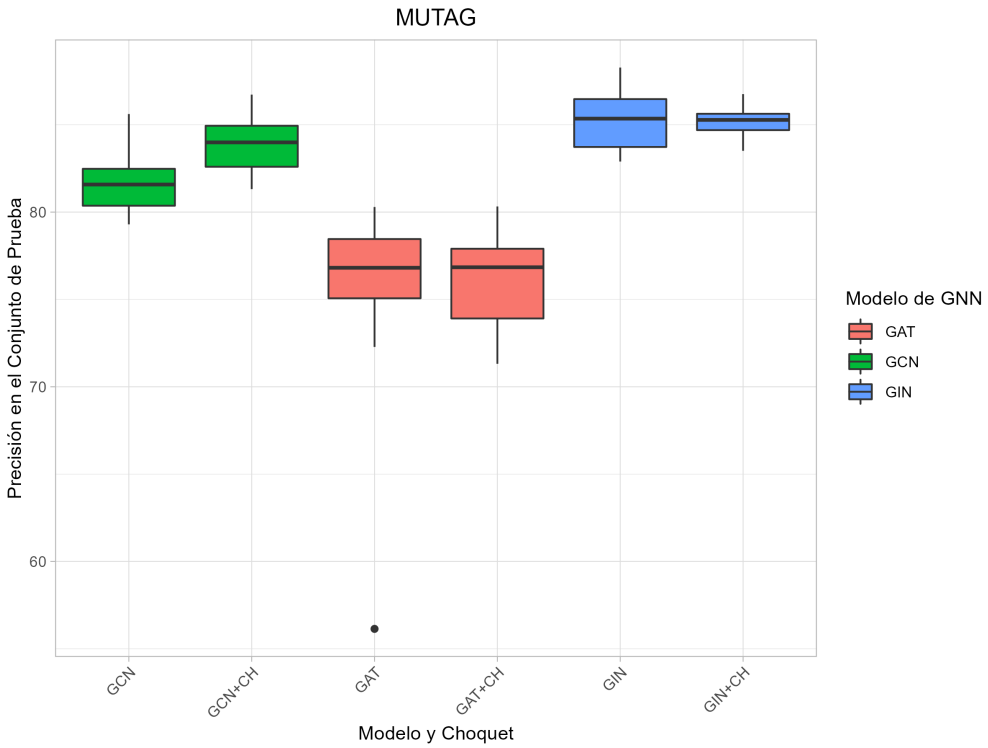

Figura 7. Diagramas de cajas de los resultados en la precisión en los conjuntos de prueba en MUTAG

4.7.3. Precisión y esfuerzo de cómputo en el caso de prueba BZR

La Tabla 7 muestra los resultados obtenidos en los casos de prueba del conjunto BZR. En este caso, no se detectan diferencias significativas en la precisión promedio de los conjuntos de prueba, siendo la mayor variación de 1.29, registrada entre los modelos de GAT. En todos los casos, los modelos con Choquet muestran un promedio menor. Al examinar los diagramas de caja de la Figura 8, se destaca que las dispersiones de GIN apenas se superponen. En contraste, GCN y GIN muestran dispersiones similares. Sin embargo, debido a la escala del eje x, estas diferencias no son sustanciales y, en la práctica, el uso de GIN como el mejor clasificador apenas sería perceptible para este caso de prueba.

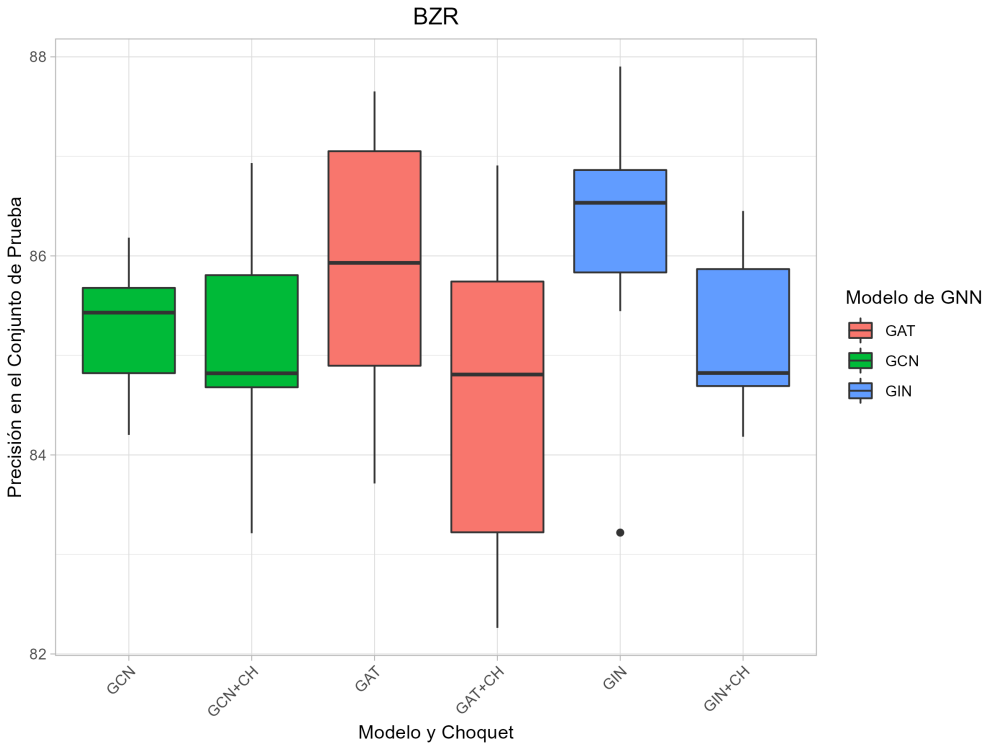

Figura 8. Diagramas de cajas de los resultados en la precisión en los conjuntos de prueba en BZR

En lo que respecta a la precisión promedio en los datos de los conjuntos de entrenamiento, tampoco se aprecian grandes variaciones entre los promedios, lo que podría explicar los resultados semejantes en los conjuntos de prueba. En este contexto, GIN demostró tener el mejor desempeño en ambos conjuntos.

En esta ocasión, los modelos GCN y GAT con Choquet presentan un promedio ligeramente mayor en la cantidad de épocas necesarias para alcanzar la convergencia, en contraste con lo observado en el conjunto de prueba MUTAG. La mayor diferencia en este aspecto se da entre GIN con y sin Choquet. Nuevamente, se observa que el tiempo total es mayor en los modelos que incorporan Choquet. Este hecho

refuerza la idea de que el cálculo de la integral discreta de Choquet es más costoso computacionalmente debido al paso de ordenamiento en la matriz de características de los nodos.

Tabla 7. Precisión y esfuerzo de cómputo en el caso de prueba BZR

	Modelo	Train Acc.±s.d	Test Acc.±s.d	Epochs±s.d	Time±s.d (hr)	Total Time (hr)
BZR	GCN	97.69 ± 0.74	85.31 ± 0.68	249.83 ± 12.70	0.11 ± 0.01	1.14
	GCN+CH	99.92 ± 0.10	85.14 ± 1.07	258.92 ± 6.87	0.16 ± 0.01	1.57
	GAT	98.34 ± 2.86	85.89 ± 1.37	309.48 ± 61.82	0.16 ± 0.03	1.64
	GAT+CH	97.29 ± 2.87	84.60 ± 1.56	313.46 ± 67.91	0.23 ± 0.05	2.28
	GIN	100.00 ± 0.00	86.27 ± 1.30	254.84 ± 5.45	0.12 ± 0.00	1.23
	GIN+CH	99.60 ± 0.17	85.18 ± 0.85	378.35 ± 7.37	0.21 ± 0.04	2.12

El hecho de que diferentes conjuntos de prueba requieran más tiempo está directamente relacionado con la cantidad de grafos que contienen, así como con el promedio de nodos y aristas que estos grafos incorporan. Esto permite contrastar las características de los conjuntos de prueba MUTAG y BZR, detalladas en las subsecciones 4.2.1.5 y 4.2.1.4, respectivamente. En el que se puede apreciar que el conjunto de prueba BZR es más complejo en las características mencionadas, por lo que requiere un mayor esfuerzo computacional.

4.7.4. Precisión y esfuerzo de cómputo en el caso de prueba ENZYMES

La Tabla 8 presenta los resultados obtenidos en el caso de prueba ENZYMES. Este conjunto de prueba es bastante complejo, y se considera que resultados por encima del 60 % en el conjunto de prueba son buenos (Dwivedi et al., 2020). En este caso, se observan diferencias significativas entre los modelos en términos de la precisión promedio en los conjuntos de prueba, que también son apreciables en la Figura 9. Por lo tanto, podríamos clasificar los modelos de mejor a peor, tanto con y sin Choquet, en el siguiente orden: GCN, GIN y GAT. Es importante destacar que en este caso particular, GCN sobresale como el mejor modelo para este conjunto de casos de prueba, a diferencia de los conjuntos de prueba MUTAG y BZR, donde GIN se estableció como el modelo más efectivo.

Tabla 8. Precisión y esfuerzo de cómputo en el caso de prueba ENZYMES

	Modelo	Train Acc.±s.d	Test Acc.±s.d	Epochs±s.d	Time±s.d (hr)	Total Time (hr)
ENZYMES	GCN	99.80 ± 0.12	66.32 ± 1.46	268.66 ± 7.00	0.15 ± 0.01	1.51
	GCN+CH	99.84 ± 0.09	64.92 ± 1.54	264.41 ± 6.79	0.22 ± 0.01	2.24
	GAT	93.61 ± 12.66	58.27 ± 7.20	366.23 ± 9.39	0.26 ± 0.01	2.60
	GAT+CH	94.40 ± 12.82	57.37 ± 7.06	365.10 ± 8.95	0.31 ± 0.01	3.11
	GIN	100.00 ± 0.00	63.20 ± 1.77	260.17 ± 8.54	0.17 ± 0.01	1.65
	GIN+CH	99.81 ± 0.56	60.60 ± 1.53	253.62 ± 6.35	0.21 ± 0.01	2.09

En relación con la precisión promedio en los datos de los conjuntos de entrenamiento, es notable que los modelos GCN y GIN, tanto con Choquet como sin él, alcanzan cerca del 100 % en este valor. Esto contrasta con GAT, que exhibe un promedio más bajo y una desviación estándar considerablemente mayor en comparación con los demás modelos, factores que impactan directamente en los resultados en los conjuntos de prueba. También es observable que en los modelos que emplean GAT se presenta un valor atípico, posiblemente debido a la misma división de datos en la misma iteración de validación cruzada. Esto podría sugerir que el método de agregación de Choquet no proporcionó diferencia sustancial al manejar este caso particular.

La diferencia en la precisión entre los conjuntos de entrenamiento y prueba en este caso de prueba es notablemente más amplia en comparación con los otros casos. Esta brecha sugiere la presencia de un sobre ajuste considerable por parte de los modelos. A partir de los resultados obtenidos en el conjunto de prueba, se infiere que ningún modelo parece capaz de generalizar efectivamente la información contenida en los grafos.

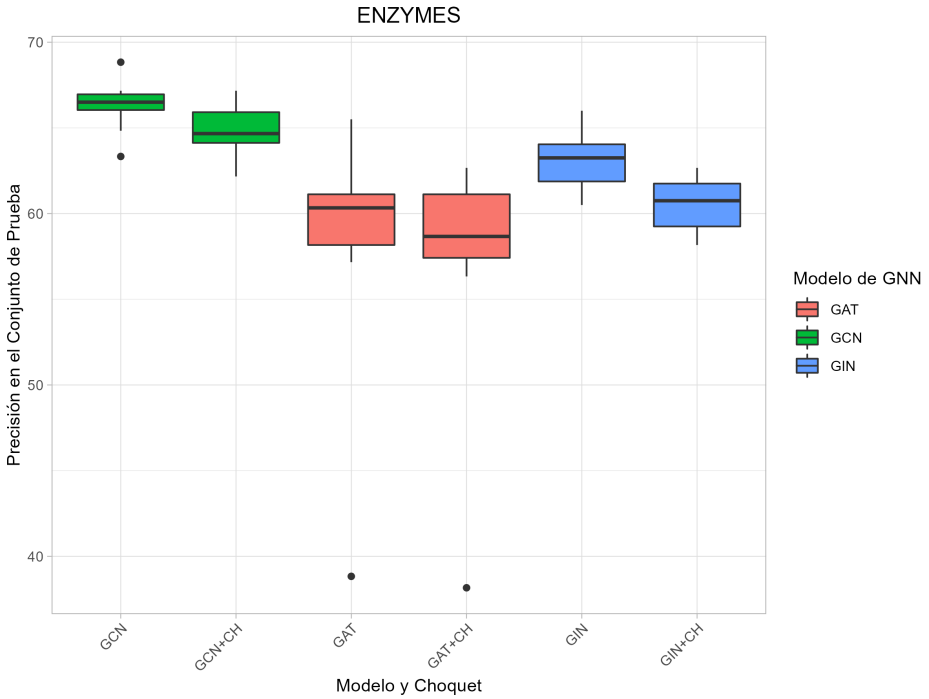

Figura 9. Diagramas de cajas de los resultados en la precisión en los conjuntos de prueba en ENZYMES

En lo que se refiere al número promedio de épocas necesarias para alcanzar la convergencia, las diferencias entre los modelos con y sin Choquet no son significativas. Sin embargo, se evidencia nuevamente que el método de agregación de Choquet incide directamente en el tiempo total, siendo GAT el modelo que requiere más tiempo.

4.7.5. Precisión y esfuerzo de cómputo en el caso de prueba COX2

La Tabla 9 muestra los resultados obtenidos en el caso de prueba COX2. Se puede observar que las diferencias en la precisión de los conjuntos de prueba entre el uso de Choquet y el promedio son similares entre mismos modelos, tanto con y sin Choquet. Destacando en este caso a GCN, a pesar que la diferencia entre el mejor y peor modelo, GCN y GAT+CH, respectivamente, es de 2.84, por lo que tampoco se encuentran demasiado separados los promedios.

Tabla 9. Precisión y esfuerzo de cómputo en el caso de prueba COX2

	Modelo	Train Acc.±s.d	Test Acc.±s.d	Epochs±s.d	Time±s.d (hr)	Total Time (hr)
COX2	GCN	94.59 ± 0.83	83.17 ± 1.01	243.54 ± 8.08	0.12 ± 0.00	1.21
	GCN+CH	95.79 ± 1.20	82.46 ± 0.97	229.87 ± 13.09	0.17 ± 0.02	1.68
	GAT	86.47 ± 4.87	80.88 ± 1.65	355.56 ± 46.90	0.16 ± 0.02	1.63
	GAT+CH	92.07 ± 4.21	80.33 ± 1.39	318.56 ± 90.43	0.20 ± 0.06	2.03
	GIN	100.00 ± 0.01	81.84 ± 0.67	256.23 ± 4.98	0.11 ± 0.00	1.10
	GIN+CH	98.55 ± 0.82	80.92 ± 1.55	384.84 ± 6.27	0.23 ± 0.01	2.33

La Figura 10 ilustra que GIN presenta una dispersión menor en comparación con los demás modelos. Por otro lado, tanto GCN como GAT, ya sea con Choquet o sin él, exhiben diagramas de caja superpuestos y una dispersión comparable.

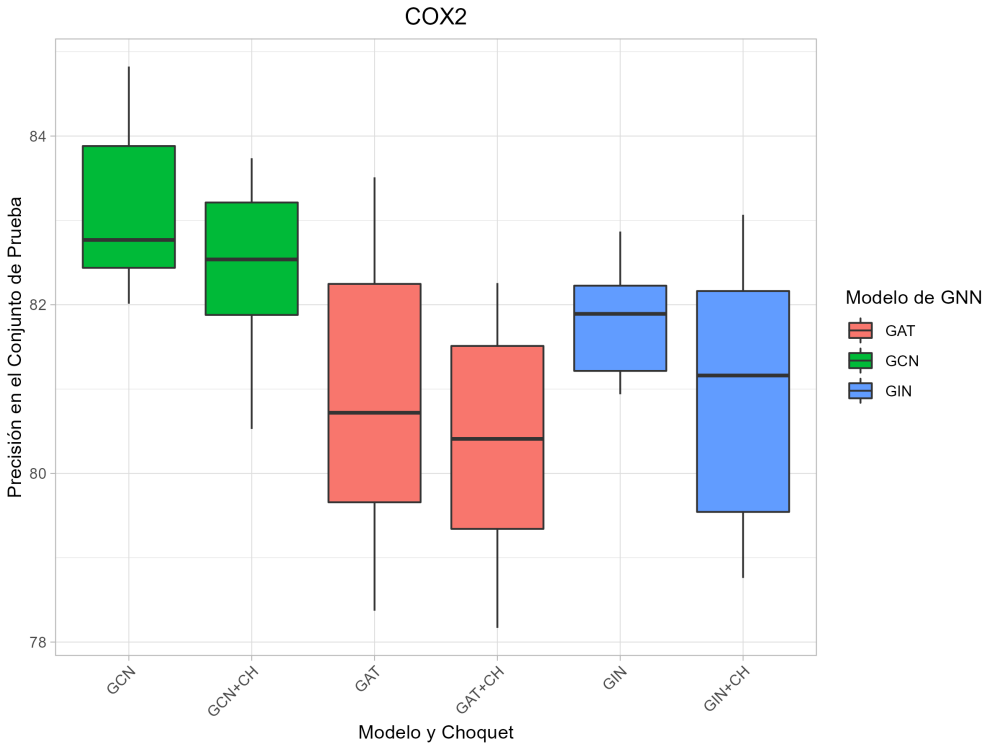

Figura 10. Diagramas de cajas de los resultados en la precisión en los conjuntos de prueba en COX2

En relación con la precisión promedio en los datos de los conjuntos de entrenamiento, en este caso, una mayor precisión en este parámetro no se traduce en un mejor rendimiento en los conjuntos de prueba. Como ejemplo, los modelos que utilizan GIN no reflejan un desempeño superior, en contraste con GCN que, a pesar de tener un rendimiento inferior en el conjunto de entrenamiento, muestra un desempeño superior en el conjunto de prueba.

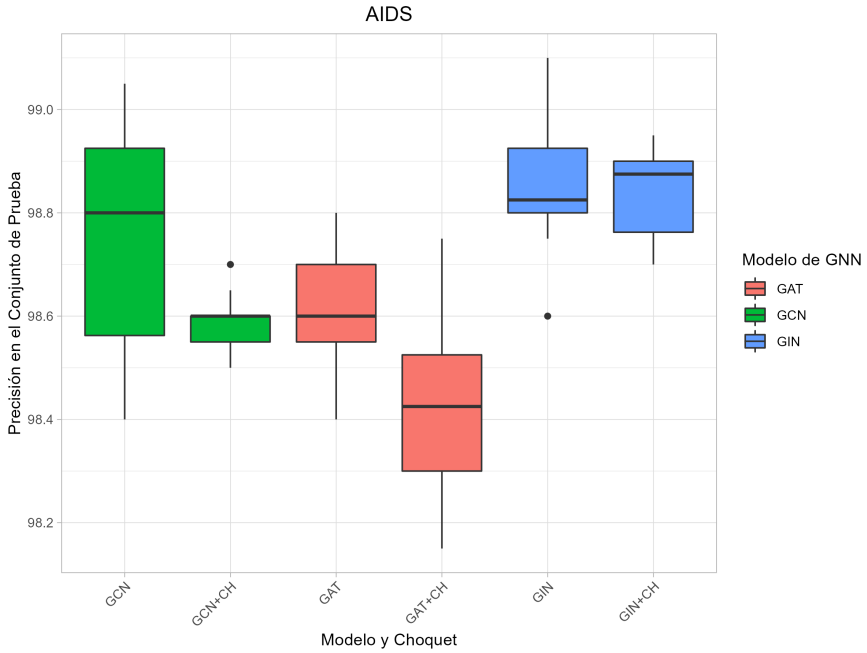

Figura 11. Diagramas de cajas de los resultados en la precisión en los conjuntos de prueba en AIDS

En cuanto al número promedio de épocas necesarias para alcanzar la convergencia, estos resultan comparables para GCN, aunque se observan diferencias en GAT y GIN. Como en observaciones anteriores, los modelos con Choquet requieren un mayor tiempo, siendo más demandantes en este aspecto.

4.7.6. Precisión y esfuerzo de cómputo en el caso de prueba AIDS

La Tabla 10 presenta los resultados obtenidos en el caso de prueba AIDS. En este caso, todos los modelos alcanzan una precisión superior al 98 % en los conjuntos de prueba, lo que podría indicar una segmentación bastante clara de los datos. Por esta razón, tampoco se observan diferencias en la precisión en los conjuntos de entrenamiento. En la Figura 11, debido a la escala, podría parecer que existen modelos mejores; sin embargo, en la práctica, estas diferencias no son notables. Lo más destacable es la menor dispersión que muestran los modelos que emplean GIN en comparación con los demás modelos.

El número de épocas requerido para alcanzar la convergencia es similar entre los modelos con y sin

Choquet, ambos presentando una desviación estándar comparable. Es importante destacar que este conjunto de datos requiere más tiempo para completar el experimento, lo que se puede atribuir a ser el caso de prueba con la mayor cantidad de grafos, específicamente 2000, tal como se observa en la subsección 4.2.1.1. Haciendo notar nuevamente que el método de agregación de Choquet tiene un esfuerzo de cómputo mayor a los modelos que usan solo el promedio.

Tabla 10. Precisión y esfuerzo de cómputo en el caso de prueba AIDS

	Modelo	Train Acc.±s.d	Test Acc.±s.d	Epochs±s.d	Time±s.d (hr)	Total Time (hr)
AIDS	GCN	99.92 ± 0.03	98.74 ± 0.24	298.03 ± 14.66	0.41 ± 0.06	4.13
	GCN+CH	99.87 ± 0.01	98.58 ± 0.06	286.46 ± 11.24	0.47 ± 0.02	4.72
	GAT	99.97 ± 0.02	98.61 ± 0.14	302.51 ± 31.94	0.45 ± 0.05	4.45
	GAT+CH	99.95 ± 0.02	98.41 ± 0.17	304.27 ± 38.24	0.59 ± 0.08	5.93
	GIN	99.96 ± 0.01	98.84 ± 0.13	289.85 ± 12.63	0.37 ± 0.02	3.66
	GIN+CH	99.93 ± 0.01	98.84 ± 0.08	285.22 ± 12.14	0.45 ± 0.02	4.48

4.8. Comparación de resultados entre la agregación de Choquet y el promedio: Análisis Estadístico

A fin de realizar un análisis estadístico de los resultados, empleamos la prueba t de Welch con el propósito de determinar si existen diferencias significativas en las medias de los valores de precisión en los conjuntos de prueba. Para ello, comparamos cada caso de prueba y modelo de GNN utilizando tanto el promedio como método de agregación, así como el método de agregación de Choquet. La elección de esta prueba en lugar de la t de Student se debe a que, al analizar los diagramas de cajas en la sección anterior, resulta evidente que no se puede garantizar el supuesto de homocedasticidad (varianzas iguales) entre los modelos con y sin Choquet, razón por la cual se optó por utilizar esta prueba.

4.8.1. Supuestos de la prueba t de Welch

La prueba t de Welch es una prueba paramétrica (Welch, 1951) que supone ciertos supuestos para su aplicación:

- Normalidad: Se presupone que las distribuciones de las medias de precisión de ambos métodos de agregación siguen una distribución normal.
- Independencia: Las observaciones de los valores de precisión en los conjuntos de prueba para cada método de agregación deben ser independientes entre sí.

- Varianzas desiguales: A diferencia de la prueba t de Student, la prueba t de Welch no supone varianzas iguales entre las poblaciones, lo que permite una mayor flexibilidad al comparar las medias.

Para verificar el supuesto de normalidad, se realizó una prueba de Kolmogorov-Smirnov, cuyos resultados pueden consultarse en el Apéndice D. A partir de estos datos, se concluye que se cumple el supuesto de normalidad para cada conjunto de valores. El supuesto de independencia de los datos se basa en que cada experimento se llevó a cabo de manera independiente y no hay razón para argumentar a favor de que exista interacción entre los modelos al momento de realizar la experimentación. El supuesto de varianzas desiguales se cumple, por lo mencionado al principio de la subsección.

4.8.2. Hipótesis de la prueba

La prueba t de Welch plantea las siguientes hipótesis:

- H_0 (hipótesis nula): Las medias de los valores de precisión en los conjuntos de prueba para ambos métodos de agregación (promedio y Choquet) son iguales.
- H_1 (hipótesis alternativa): Las medias de los valores de precisión en los conjuntos de prueba para ambos métodos de agregación son diferentes.

Para realizar las pruebas estadísticas, utilizamos la función 't.test' del paquete 'stats' en R, con el parámetro 'var.equal' establecido en FALSE. Esto nos permite realizar la prueba t de Welch.

4.8.3. Nivel de significancia y valor p

Para llevar a cabo la prueba t de Welch, se estableció un nivel de significancia $\alpha = 0.05$. Por lo tanto, si $p \leq 0.05$, se rechaza la hipótesis nula en favor de la hipótesis alternativa (las medias son diferentes). En caso contrario, no se rechaza la hipótesis nula (las medias son iguales).

4.8.4. Observaciones sobre la prueba t de Welch

Tras llevar a cabo la prueba t de Welch, se obtuvieron los resultados que se muestran en la Tabla 11. Debido a limitaciones de formato relacionadas con el tamaño de la tabla, téngase en cuenta las siguientes

consideraciones al interpretar las conclusiones de las pruebas t de Welch. Estas aclaraciones tienen el objetivo de garantizar la precisión al describir estadísticamente lo que implican estas conclusiones:

- Cuando se concluye que *Las medias son diferentes*, en realidad debe interpretarse como: Se rechaza H_0 en favor de H_1, por lo tanto, las medias son significativamente diferentes.
- Cuando se concluye que *Las medias son iguales*, en realidad debe interpretarse como: No se rechaza H_0, por lo tanto, no hay evidencia estadística suficiente para afirmar que las medias son diferentes.

Tabla 11. Resultados de las pruebas t de Welch para comparar medias

Caso de Prueba	Modelo	Modelo con Choquet	Valor P	Conclusión
MUTAG	GCN	GCN+CH	0.01	Las medias son diferentes.
MUTAG	GAT	GAT+CH	0.61	Las medias son iguales.
MUTAG	GIN	GIN+CH	0.91	Las medias son iguales.
BZR	GCN	GCN+CH	0.68	Las medias son iguales.
BZR	GAT	GAT+CH	0.06	Las medias son iguales.
BZR	GIN	GIN+CH	0.04	Las medias son diferentes.
ENZYMES	GCN	GCN+CH	0.05	Las medias son iguales.
ENZYMES	GAT	GAT+CH	0.78	Las medias son iguales.
ENZYMES	GIN	GIN+CH	0.01	Las medias son diferentes.
COX2	GCN	GCN+CH	0.13	Las medias son iguales.
COX2	GAT	GAT+CH	0.43	Las medias son iguales.
COX2	GIN	GIN+CH	0.11	Las medias son iguales.
AIDS	GCN	GCN+CH	0.07	Las medias son iguales.
AIDS	GAT	GAT+CH	0.02	Las medias son diferentes.
AIDS	GIN	GIN+CH	0.92	Las medias son iguales.

4.8.5. Análisis de los resultados de la prueba t de Welch

Al examinar la Tabla 11, observamos que en la mayoría de los casos se concluye que, estadísticamente, las medias son iguales. Sin embargo, en algunos casos se determina que las medias son diferentes. Por ejemplo, en el caso de GCN con MUTAG, al revisar la Tabla 6, notamos que GCN+CH tiene un promedio mayor que GCN ($83.80 > 81.70$). En el caso de GIN con BZR, al consultar la Tabla 7, vemos que GIN tiene un promedio mayor que GIN+CH ($86.27 > 85.18$). En el caso de ENZYMES, al revisar la Tabla 8, se observa que GIN tiene un promedio mayor que GIN+CH ($63.20 > 60.60$). Finalmente, en el caso de AIDS, al analizar la Tabla 10, encontramos que GAT tiene un promedio mayor que GAT+CH ($98.61 > 98.41$).

Cabe destacar que en un único caso se puede afirmar estadísticamente que el uso del método de agregación de Choquet tuvo un efecto, siendo este el modelo GCN en MUTAG. En los otros casos en los

que hubo diferencias en las medias, los modelos que empleaban el método de agregación de Choquet resultaron en promedios más bajos que aquellos que utilizaban el promedio como método de agregación. Aún así, estadísticamente no son diferentes en rendimiento en aquellos casos.

4.9. Convergencia del valor de λ de la medida de Sugeno

Como se mencionó en la subsección 2.3.5 sobre la medida de Sugeno y en la subsección 3.4.1.4 acerca de la programación del método numérico para determinar el valor de λ de la medida de Sugeno, el valor de λ se encuentra en un intervalo que depende de si la suma de los pesos asociados a las características de los nodos de la capa final es mayor o menor a 1. En el caso de que $\lambda \in (0, \infty)$, la suma de los pesos cumple con $\sum_{i=1}^{N} p_i = q < 1$, mientras que $\lambda \in [-1, 0)$ cuando $\sum_{i=1}^{N} p_i = q > 1$. Al observar los resultados en la Tabla 12, notamos que la convergencia promedio de los valores de λ se encuentra dentro del intervalo $[-1, 0)$, lo cual se atribuye a que la suma promedio de los pesos de las características siempre es mayor a 1. Por lo tanto, se puede afirmar que los pesos se han adaptado para actuar con una sinergia sub-aditiva entre las características de los nodos en todos los casos. Esto podría sugerir que, en general, existen características ocultas que se contraponen entre sí. Una posible forma de evaluar esto sería modificando el número de características en las capas de la red, aumentándolas o reduciéndolas, para ver si esto influye en alguna manera en el tipo de sinergia.

Por otro lado, es posible determinar la intensidad de esta sinergia en función de qué tan cercano está el valor de λ al -1. La sinergia será más fuerte (en terminos de sub-aditividad) cuanto más se acerque al valor de λ a -1. Sin embargo, no se puede identificar con precisión la razón de estos resultados, ya que al ponderar características ocultas, no necesariamente existe un significado asociado a los casos de prueba. Por lo que no podríamos decir que existe cierta sinergia entre las características de los nodos de los grafos en los casos de prueba.

Viendo los resultados por modelos, se puede apreciar que GIN+CH ha sido el modelo que presenta un valor de λ más cercano a -1, en contraste con lo que se observa con GCN+CH y GAT+CH, que tienen valores más dispersos. Esto también podría ser indicativo del tipo de evolución que las características ocultas experimentan en diferentes modelos de GNN. Sin embargo, no se puede afirmar que GIN tiende a producir sinergias sub-aditivas entre las características, dado que esto está influenciado directamente por el uso del método de agregación de Choquet. Una vez más, sería interesante investigar si esta dinámica puede ser alterada cambiando el tamaño de las capas de la red.

Un aspecto importante a tener en cuenta es la aplicación del método de agregación de Choquet en la

salida de la última capa de la red. Recordemos que la esencia del método de agregación de Choquet reside en adaptar los pesos asociados a las características con el propósito de combinar la información de los nodos del grafo. Por lo que al implementar este método en las características ocultas podría no ser el enfoque adecuado.

Basándonos en los resultados obtenidos de las pruebas t de Welch, se podría inferir que, pese a todo, el método de agregación de Choquet parece haberse adaptado para comportarse de manera similar al promedio.

Tabla 12. Convergencias promedios y desviación estandar de los valores de λ en cada modelo y caso de prueba

Modelo	Conjunto de datos	Convergencia del valor de λ	Suma promedio de los pesos
GCN+CH	AIDS	-0.71±0.08	1.80±0.21
GCN+CH	BZR	-0.89±0.03	2.47±0.23
GCN+CH	COX2	-0.54±0.05	1.47±0.06
GCN+CH	ENZYMES	-0.63±0.16	3.03±0.82
GCN+CH	MUTAG	-0.57±0.11	1.50±0.17
GAT+CH	AIDS	-0.60±0.06	1.51±0.09
GAT+CH	BZR	-0.70±0.11	1.90±0.29
GAT+CH	COX2	-0.70±0.22	2.02±0.35
GAT+CH	ENZYMES	-0.82±0.09	2.13±0.30
GAT+CH	MUTAG	-0.61±0.26	1.61±0.25
GIN+CH	AIDS	-0.82±0.04	2.10±0.17
GIN+CH	BZR	-0.89±0.02	3.66±0.57
GIN+CH	COX2	-0.99±0.01	4.56±0.86
GIN+CH	ENZYMES	-0.80±0.18	2.79±0.48
GIN+CH	MUTAG	-0.73±0.05	1.81± 0.13

4.10. Análisis descriptivo de los resultados considerando solo los modelos de GNN

Para concluir este capítulo, queremos destacar los diagramas de cajas presentados en la Figura 12. En dicha figura, se muestran los resultados de precisión para cada caso de prueba utilizando modelos de GNN, omitiendo aquellos modelos que emplean el método de agregación de Choquet. Al observar cada caso de prueba, se puede apreciar que existe un modelo de GNN que supera a los demás en rendimiento. Sin embargo, no hay un modelo que sea superior en todos los casos de prueba. Esto concuerda con el teorema de "No Free Lunch"(Wolpert & Macready, 1997), que sostiene que no existe un algoritmo de aprendizaje único que sea el mejor en todos los escenarios.

En situaciones particulares, se puede observar que en algunos casos el modelo GIN obtiene los mejores resultados, mientras que en otros es el modelo GCN. Aunque GAT no parece superar a los demás modelos en estos casos de prueba, podría haber situaciones en las que sí lo haga. Por lo tanto, siempre

es importante evaluar diferentes modelos para identificar el que mejor se adapte a las necesidades específicas.

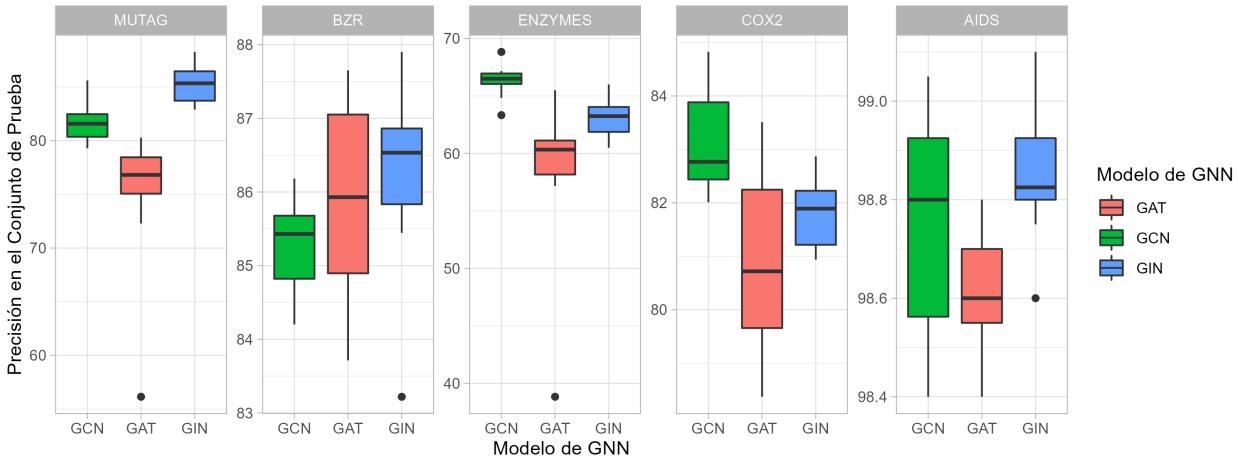

Figura 12. Diagrama de cajas de los resultados en la precisión en los conjuntos de prueba por modelo de GNN

Capítulo 5. Conclusiones

5.1. Sumario

En este trabajo de tesis, se ha desarrollado un nuevo método de agregación, fundamentado en la integral discreta de Choquet. Este método puede aplicarse en la clasificación de grafos utilizando redes neuronales en grafos. Además, hemos diseñado algoritmos y código para su implementación práctica en PyTorch. Para complementar, se realizaron una serie de experimentos con el objetivo de evaluar la efectividad de este método en contraposición al enfoque convencional que se basa en el uso del promedio para la agregación en grafos.

5.2. Conclusiones

Como resultado de los hallazgos en este trabajo de tesis, podemos establecer las siguientes conclusiones. En primer lugar, vemos que es posible integrar los conceptos de la integral discreta de Choquet en una red neuronal para facilitar el aprendizaje automático de los pesos involucrados en el método de agregación basado en esta integral. Esto se logra a través de una formulación apropiada del grafo computacional y su implementación utilizando PyTorch Geometric. Por otro lado, se observó que el método de agregación propuesto en este trabajo, debido al ordenamiento de la matriz de características en el cálculo de la integral discreta de Choquet, requiere un esfuerzo computacional mayor, lo que se traduce en un tiempo de ejecución más largo.

También se concluyó que, desde una perspectiva estadística, en los casos de prueba analizados, generalmente el método de agregación de Choquet no presenta diferencias significativas en comparación con el uso del promedio como método de agregación. Sin embargo, hubo algunos casos en los que se encontraron diferencias estadísticamente significativas, en su mayoría favorables a los modelos que utilizan el promedio. La única excepción fue el caso de prueba MUTAG, donde el método de agregación de Choquet tuvo mejor rendimiento. En los casos donde no se observaron diferencias significativas entre los métodos de agregación, los pesos asociados al método de Choquet podrían haber hecho que su rendimiento se asemejara al del promedio. Esto podría deberse al bajo nivel de sinergia reflejado en los valores de λ obtenidos. Además, en todos los escenarios, los modelos de GNN con Choquet presentaron un valor de λ en el intervalo de $[-1, \infty)$, lo que sugiere que las características finales de los nodos en los diferentes modelos se ajustaron para actuar de manera sub-aditiva.

Para finalizar, hacemos notar que la aplicación del método de agregación de Choquet en características

ocultas no parece ser el enfoque más efectivo para su utilización en los conjuntos de datos analizados. Este argumento se fundamenta en la idea de que el método de Choquet se basa en explotar el significado intrínseco de las características de los nodos.

5.3. Trabajo futuro

En las siguientes líneas, se expondrán algunas reflexiones e ideas que podrían ser útiles para futuras investigaciones:

- Implementar un método de paso de mensajes que incorpore una operación de agregación basada en el método de agregación de Choquet. Esto podría proporcionar beneficios al considerar información local y una menor cantidad de datos en comparación con la agregación completa del grafo. De esta manera cada nodo tendría asociado un vector de pesos asociados a las características de los nodos de su vecindad. Por lo que cada nodo tendría ajustado un método de agregación en función de las características de sus vecinos.

- Adaptar un nuevo método de agregación basado en la medida-q (q-measure) que es conceptualmente similar a la medida de Sugeno. Esta propuesta tiene la ventaja de permitir la definición a priori del valor de λ, en lugar de dejar que los pesos aprendidos definan dicho valor. De esta manera, es el valor de λ el que se aprende. Esto último podría otorgar mayor libertad para que el modelo explore sinergias entre las características. Caso contrario al método de agregación de Choquet que tiene que coordinar la suma de los pesos para definir un tipo sinergia entre las características.

- Podrían considerarse otros puntos de incorporación para el método de agregación de Choquet en una GNN. Por ejemplo, este método podría emplearse en las características iniciales de los nodos para establecer un ranking, y posteriormente, esta información podría ser integrada en la agregación utilizando cualquier otro método de agregación.

- El método de agregación de Choquet puede utilizarse en contextos distintos a la clasificación de grafos. Como se observó en la metodología, la agregación de información utilizando el método de agregación de Choquet puede aplicarse a cualquier matriz.

- El método de agregación podría ser aplicado en contextos distintos al del aprendizaje profundo. En particular, podría explorarse su uso en el ámbito del aprendizaje automático, en el que las características no atraviesan un proceso que despoje por completo su significado intrínseco.

- Para el caso de prueba ENZYMES, se observa un sobre ajuste considerable en los modelos, independientemente del método de agregación empleado. Dado esto, se podrían explorar las causas de esta situación y buscar estrategias que puedan solucionar este problema.

Literatura citada

Barabasi, A. L. (2013). Network science. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371. https://doi.org/ 10.1098/rsta.2012.0375.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V. F., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Çaglar Gülçehre, Song, H. F., Ballard, A. J., Gilmer, J., Dahl, G. E., Vaswani, A., Allen, K. R., Nash, C., Langston, V., Dyer, C., Heess, N. M. O., Wierstra, D., Kohli, P., Botvinick, M. M., Vinyals, O., Li, Y., & Pascanu, R. (2018). Relational inductive biases, deep learning, and graph networks. *ArXiv*, abs/1806.01261. https://doi.org/ 10.48550/arXiv.1806.01261.

Beliakov, G., Bustince, H., & Calvo, T. (2016). A practical guide to averaging functions. In *Studies in Fuzziness and Soft Computing*. https://doi.org/ 10.1007/978-3-319-24753-3.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

Borgwardt, K. M. & Kriegel, H.-P. (2005). Shortest-path kernels on graphs. *Fifth IEEE International Conference on Data Mining (ICDM'05)*, 8 pp.–. https://doi.org/ 10.1109/ICDM.2005.132.

Borgwardt, K. M., Ong, C. S., Schönauer, S., Vishwanathan, S. V. N., Smola, A., & Kriegel, H.-P. (2005). Protein function prediction via graph kernels. *Bioinformatics*, 21 Suppl 1:i47–56. https://doi.org/ 10.1093/bioinformatics/bti1007.

Burden, R. L. & Faires, J. D. (1989). *Numerical Analysis*. The Prindle, Weber and Schmidt Series in Mathematics. PWS-Kent Publishing Company, Boston, (4ta ed.).

Chen, T.-Y. & Wang, J.-C. (2001). Identification of [lambda]-fuzzy measures using sampling design and genetic algorithms. *Fuzzy Sets Syst.*, 123:321–341. https://doi.org/ 10.1016/S0165-0114(97)00174-7.

Cheng, D., Wang, X., Zhang, Y., & Zhang, L. (2022). Graph neural network for fraud detection via spatial-temporal attention. *IEEE Transactions on Knowledge and Data Engineering*, 34(8):3800–3813. https://doi.org/ 10.1109/TKDE.2020.3025588.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1994). *Introduction to algorithms*. MIT press Cambridge, MA, USA, (3a ed.).

Coroianu, L. C., Fullér, R., & Harmati, I. Á. (2022). Best approximation of owa olympic weights under predefined level of orness. *Fuzzy Sets Syst.*, 448:127–144. https://doi.org/ 10.1016/j.fss.2022.07.009.

Debnath, A. K., Lopez de Compadre, R. L., Debnath, G., Shusterman, A. J., & Hansch, C. (1991). Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry*, 34(2):786–797. https://doi.org/ 10.1021/jm00106a046.

Dwivedi, V. P., Joshi, C. K., Laurent, T., Bengio, Y., & Bresson, X. (2020). Benchmarking graph neural networks. *ArXiv*, abs/2003.00982. https://doi.org/ 10.48550/arXiv.2003.00982.

Errica, F., Podda, M., Bacciu, D., & Micheli, A. (2019). A fair comparison of graph neural networks for graph classification. *ArXiv*, abs/1912.09893. https://doi.org/ 10.48550/arXiv.1912.09893.

Feragen, A., Kasenburg, N., Petersen, J., de Bruijne, M., & Borgwardt, K. (2013). Scalable kernels for graphs with continuous attributes. In *Advances in Neural Information Processing Systems*, (pp.216–224).

Fey, M. & Lenssen, J. E. (2019). Fast graph representation learning with pytorch geometric. *ArXiv*, abs/1903.02428. https://doi.org/ 10.48550/arXiv.1903.02428.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*. https://doi.org/ 10.48550/arXiv.1704.01212.

Grabisch, M. & Roubens, M. (2000). Application of the choquet integral in multicriteria decision making.

Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 1024–1034. Curran Associates Inc. https://doi.org/ 10.48550/arXiv.1706.02216.

Helma, C., Cramer, T., Kramer, S., & De Raedt, L. (2004). Data mining and machine learning techniques for the identification of mutagenicity inducing substructures and structure activity relationships of noncongeneric compounds. *Journal of Chemical Information and Computer Sciences*, 44(4):1402–1411. https://doi.org/ 10.1021/ci034254q.

Kazius, J., Nijssen, S., Kok, J., Bäck, T., & Ijzerman, A. (2008). Substructure mining using elaborate chemical representation. *Journal of Chemical Information and Modeling*, 48(2):251–266. https://doi.org/ 10.1021/ci0503715.

Kingma, D. P. & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. https://doi.org/ 10.48550/arXiv.1412.6980.

Kipf, T. & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *ArXiv*, abs/1609.02907. https://doi.org/ 10.48550/arXiv.1609.02907.

Klir, G. J., Wang, Z., & Harmanec, D. (1997). Constructing fuzzy measures in expert systems. *Fuzzy Sets Syst.*, 92:251–264.

Klir, G. J. & Yuan, B. (1995). *Fuzzy sets, uncertainty, and information*. Prentice Hall PTR.

Labatut, V. & Cherifi, H. (2012). Accuracy measures for the comparison of classifiers. *ArXiv*, abs/1207.3790. https://doi.org/ 10.48550/arXiv.1207.3790.

learn developers, S. (2021). Cross-validation: evaluating estimator performance. `https://scikit-learn.org/stable/modules/cross_validation.html#stratification`. Accessed: 2023-03-15.

Leszczynski, K. W., Penczek, P. A., & Grochulski, W. D. (1985). Sugeno's fuzzy measure and fuzzy clustering. *Fuzzy Sets and Systems*, 15:147–158. https://doi.org/ 10.1016/0165-0114(85)90043-0.

Li, Y., Yu, R., Shahabi, C., & Liu, Y. (2017). Graph convolutional recurrent neural network: Data-driven traffic forecasting. *CoRR*, abs/1707.01926.

Lilliefors, H. W. (1967). On the kolmogorov-smirnov test for normality with mean and variance unknown. *Journal of the American Statistical Association*, 62:399–402. https://doi.org/ 10.2307/2283970.

Lust, T. (2015). Choquet integral versus weighted sum in multicriteria decision contexts. In *Algorithmic Decision Theory*. https://doi.org/ 10.1007/978-3-319-23114-3_18.

Morris, C., Kersting, K., & Mutzel, P. (2020). Tudataset: A collection of benchmark datasets for learning with graphs. *arXiv preprint arXiv:2007.08663*. https://doi.org/ 10.48550/arXiv.2007.08663.

Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., & Grohe, M. (2019). Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 4602–4609. https://doi.org/ 10.48550/arXiv.1810.02244.

Nwankpa, C., Ijomah, W. L., Gachagan, A., & Marshall, S. (2018). Activation functions: Comparison of trends in practice and research for deep learning. *ArXiv*, abs/1811.03378. https://doi.org/ 10.48550/arXiv.1811.03378.

Riesen, K. & Bunke, H. (2008). Iam graph database repository for graph based pattern recognition and machine learning. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, 287–297. Springer, Berlin, Heidelberg. https://doi.org/ 10.1007/978-3-540-89689-0_33.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, 20:61–80. https://doi.org/ 10.1109/TNN.2008.2005605.

Shervashidze, N., Schweitzer, P., van Leeuwen, E. J., Mehlhorn, K., & Borgwardt, K. M. (2011). Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561. https://doi.org/ 10.5555/1953048.2078187.

Shervashidze, N., Vishwanathan, S., Petri, T., Mehlhorn, K., & Borgwardt, K. (2009). Efficient graphlet kernels for large graph comparison. In van Dyk, D. & Welling, M., editors, *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, 488–495, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA. PMLR.

Singh, D. & Singh, B. (2020). Investigating the impact of data normalization on classification performance. *Appl. Soft Comput.*, 97:105524. https://doi.org/ 10.1016/j.asoc.2019.105524.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., & Bengio, Y. (2018). Graph attention networks. In *International Conference on Learning Representations (ICLR)*. https://doi.org/ 10.48550/arXiv.1710.10903.

Vinyals, O., Bengio, S., & Kudlur, M. (2015). Order matters: Sequence to sequence for sets. *CoRR*, abs/1511.06391. https://doi.org/ 10.48550/arXiv.1511.06391.

Vishwanathan, S. V. N., Schraudolph, N. N., Kondor, R., & Borgwardt, K. M. (2008). Graph kernels. *ArXiv*, abs/0807.0093. https://doi.org/ 10.5555/1756006.1859891.

Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, M., Zhou, J., Huang, Q., Ma, C., Huang, Z., Guo, Q., Zhang, H., Lin, H., Zhao, J. J., Li, J., Smola, A., & Zhang, Z. (2019a). Deep graph library: Towards efficient and scalable deep learning on graphs. *ArXiv*, abs/1909.01315.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., & Zhang, Z. (2019b). Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*. https://doi.org/ 10.48550/arXiv.1909.01315.

Wang, Q., Ma, Y., Zhao, K., & jie Tian, Y. (2020). A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, 1–26. https://doi.org/ 10.1007/s40745-020-00253-5.

Welch, B. L. (1951). On the comparison of several mean values: An alternative approach. *Biometrika*, 38:330–336. https://doi.org/ 10.2307/2332579.

Wolpert, D. H. & Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Trans. Evol. Comput.*, 1:67–82. https://doi.org/ 10.1109/4235.585893.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2021). A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24. https://doi.org/ 10.1109/TNNLS.2020.2978386.

Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2018). How powerful are graph neural networks? In *Proceedings of the International Conference on Learning Representations (ICLR)*. https://doi.org/10.48550/arXiv.1810.00826.

Yager, R. (1988). On ordered weighted averaging aggregation operators in multicriteria decision-making. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):183–190. https://doi.org/10.1109/21.87068.

Yager, R. & Filev, D. (1999). Induced ordered weighted averaging operators. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 29(2):141–150. https://doi.org/10.1109/3477.752789.

Yang, Z., Cohen, W. W., & Salakhutdinov, R. (2016). Revisiting semi-supervised learning with graph embeddings. In *Proceedings of the 33rd International Conference on Machine Learning*, 40–48. PMLR. https://doi.org/ 10.48550/arXiv.1603.08861.

Ying, R., You, J., Morris, C., Ren, X., Hamilton, W. L., & Leskovec, J. (2018). Hierarchical graph representation learning with differentiable pooling. In *Neural Information Processing Systems*. https://doi.org/ 10.48550/arXiv.1806.08804.

Zitnik, M. & Leskovec, J. (2017). Predicting multicellular function through multi-layer tissue networks. *Bioinformatics*, 33(14):i190–i198. https://doi.org/ 10.1093/bioinformatics/btx252.

Anexos

Anexo A. Acceso al conjunto de funciones y librerías utilizadas

El presente anexo tiene como objetivo proporcionar acceso al conjunto de funciones y librerías empleadas en el desarrollo de esta tesis. Los interesados podrán acceder a dichos recursos a través del siguiente enlace:

https://colab.research.google.com/drive/1C-CPnO2ce__FZ0uRMFnOkLtOYmkBAhG4?usp=share_link

Es importante mencionar que, aunque el contenido de la libreta de código en el enlace proporcionado puede ser copiado para su uso o estudio, no está permitido modificarlo directamente. Cualquier cambio debe realizarse en una copia del archivo original.

En caso de tener alguna duda, comentario o requerir asistencia relacionada con el código presentado, por favor no dude en ponerse en contacto con el autor de la tesis mediante correo electrónico a la siguiente dirección: hsarmiento@cicese.edu.mx

Anexo B. Códigos de los modelos de GNN

En este anexo se presentan los códigos de los modelos de GNN utilizados en esta tesis.

A continuación se muestra el código que define el modelo **GCN** utilizado en los experimentos realizados en esta tesis:

```
class GCNNet(nn.Module):
    def __init__(self, net_params):
        super().__init__()
        in_dim = net_params['in_dim']
        hidden_dim = net_params['hidden_dim']
        out_dim = net_params['out_dim']
        n_classes = net_params['n_classes']
        in_feat_dropout = net_params['in_feat_dropout']
        dropout = net_params['dropout']
        n_layers = net_params['L']
        self.readout = net_params['readout']
        self.batch_norm = net_params['batch_norm']
        self.residual = net_params['residual']
        self.embedding_h = nn.Linear(in_dim, hidden_dim)
        self.in_feat_dropout = nn.Dropout(net_params['in_feat_dropout'])

        #CAPAS DE GNN ...
        self.layers = GCN(              in_channels=hidden_dim,
                                        hidden_channels=hidden_dim,
                                        out_channels =out_dim,
                                        act= "relu", #revisar
                                        dropout = dropout,
                                        norm= 'BatchNorm',
                                        num_layers = n_layers
                                        )
        #...
        #PARA CHOQUET...
        self.lamb = torch.tensor(0.0)
        self.suma = 0.0
        self.choquet = net_params['choquet']

        if net_params['init_w'] == 0.0:
```

```
            self.choquet_weigth = nn.Parameter(torch.rand(net_params['
                choquet_feats']))
        else:
            self.choquet_weigth = nn.Parameter(torch.tensor([net_params['
                init_w']]).repeat(net_params['choquet_feats']))

        if net_params['choquet']:
            self.MLP_layer = MLPReadout( net_params['choquet_feats'],
                n_classes)
        else:
            self.MLP_layer = MLPReadout(out_dim, n_classes)

    def forward(self, x, edge_index, batch):
        # 1. H^{0} y las 4 capas de GNN
        x = self.embedding_h(x)
        x = self.layers(x, edge_index)

        # 2. Calculo del valor de lambda
        if self.choquet:
            root = root(lamb_func_np, self.choquet_weigth.detach().cpu().
                numpy())
            self.lamb = root
            self.suma = self.choquet_weigth.detach().cpu().numpy().sum()

        # 3. Metodo de agregacion
        if self.readout == "sum":
            x = global_add_pool(x, batch) #[batch_size, hidden_channels]
        elif self.readout == "max":
            x = global_max_pool(x, batch) #[batch_size, hidden_channels]
        elif self.readout == "mean":
            x = global_mean_pool(x, batch) #[batch_size, hidden_channels]
        elif self.readout == "choq":
            x = choquet_aggregation(x, self.choquet_weigth, root, batch)
        else:
            x = global_mean_pool(x, batch) #[batch_size, hidden_channels]

        return self.MLP_layer(x)
```

```
    def loss(self, pred, label):
        criterion = nn.CrossEntropyLoss()
        loss = criterion(pred, label)
        return loss
```

A continuación se muestra el código que define el modelo **GAT** utilizado en los experimentos realizados en esta tesis:

```
class GATNet(nn.Module):
    def __init__(self, net_params):
        super().__init__()
        in_dim = net_params['in_dim']
        hidden_dim = net_params['hidden_dim']
        out_dim = net_params['out_dim']
        n_classes = net_params['n_classes']
        num_heads = net_params['n_heads']
        in_feat_dropout = net_params['in_feat_dropout']
        dropout = net_params['dropout']
        n_layers = net_params['L']
        self.readout = net_params['readout']
        self.batch_norm = net_params['batch_norm']
        self.residual = net_params['residual']
        self.dropout = dropout
        self.embedding_h = nn.Linear(in_dim, hidden_dim * num_heads)
        self.in_feat_dropout = nn.Dropout(net_params['in_feat_dropout'])


        #CAPAS DE GNN ...
        self.layers = GAT(        in_channels=  hidden_dim * num_heads ,
                                  hidden_channels=hidden_dim * num_heads,
                                  out_channels =out_dim,
                                  act= "relu", #revisar
                                  dropout = dropout,
                                  norm= 'BatchNorm',
                                  num_layers = n_layers,
                                  heads = num_heads
                                  )
        #...
```

```
        #PARA CHOQUET...
        self.lamb = torch.tensor(0.0)
        self.suma = 0.0
        self.choquet = net_params['choquet']

        if net_params['init_w'] == 0.0:
          self.choquet_weigth = nn.Parameter(torch.rand(net_params['
              choquet_feats']))
        else:
          self.choquet_weigth = nn.Parameter(torch.tensor([net_params['
              init_w']]).repeat(net_params['choquet_feats']))

        if net_params['choquet']:
          self.MLP_layer = MLPReadout( net_params['choquet_feats'],
              n_classes)
        else:
          self.MLP_layer = MLPReadout(out_dim, n_classes)
    def forward(self, x, edge_index, batch):
        # 1. H^{0} y las 4 capas de GNN
        x = self.embedding_h(x)
        x = self.layers(x, edge_index)

        # 2. Calculo del valor de lambda
        if self.choquet:
            root = root(lamb_func_np, self.choquet_weigth.detach().cpu().
                numpy())
            self.lamb = root
            self.suma = self.choquet_weigth.detach().cpu().numpy().sum()

        # 3. Metodo de agregacion
        if self.readout == "sum":
            x = global_add_pool(x, batch) #[batch_size, hidden_channels]
        elif self.readout == "max":
            x = global_max_pool(x, batch) #[batch_size, hidden_channels]
        elif self.readout == "mean":
            x = global_mean_pool(x, batch) #[batch_size, hidden_channels]
        elif self.readout == "choq":
            x = choquet_aggregation(x, self.choquet_weigth, root, batch)
```

```
        else:
            x = global_mean_pool(x, batch) #[batch_size, hidden_channels]

        return self.MLP_layer(x)

    def loss(self, pred, label):
        criterion = nn.CrossEntropyLoss()
        loss = criterion(pred, label)
        return loss
```

A continuación se muestra el código que define el modelo **GIN** utilizado en los experimentos realizados en esta tesis:

```
class GINNet(nn.Module):
    def __init__(self, net_params):
        super().__init__()
        in_dim = net_params['in_dim']
        hidden_dim = net_params['hidden_dim']
        out_dim = net_params['out_dim']
        n_classes = net_params['n_classes']
        dropout = net_params['dropout']
        self.n_layers = net_params['L']
        n_mlp_layers = net_params['n_mlp_GIN']               # GIN
        learn_eps = net_params['learn_eps_GIN']              # GIN
        neighbor_aggr_type = net_params['neighbor_aggr_GIN'] # GIN
        self.readout  = net_params['readout']                # this is
            graph_pooling_type
        batch_norm = net_params['batch_norm']
        residual = net_params['residual']
        self.embedding_h = nn.Linear(in_dim, hidden_dim)
        self.ginlayers = torch.nn.ModuleList()               # List of
            MLPs
        self.in_feat_dropout = nn.Dropout(net_params['in_feat_dropout'])

        #CAPAS DE GNN ...
        self.layers = GIN( in_channels= hidden_dim,
                           hidden_channels = hidden_dim,
                           out_channels= out_dim,
```

```
                    dropout= dropout,
                    norm = 'BatchNorm',
                    num_layers= net_params['L'] ,
                    act='relu'
                    )
        #...
        #PARA CHOQUET...
        self.lamb = torch.tensor(0.0)
        self.suma = 0.0
        self.choquet = net_params['choquet']

        if net_params['init_w'] == 0.0:
          self.choquet_weigth = nn.Parameter(torch.rand(net_params['
              choquet_feats']))
        else:
          self.choquet_weigth = nn.Parameter(torch.tensor([net_params['
              init_w']]).repeat(net_params['choquet_feats']))

        if net_params['choquet']:
          self.MLP_layer = MLPReadout( net_params['choquet_feats'],
              n_classes)
        else:
          self.MLP_layer = MLPReadout(out_dim, n_classes)
    def forward(self, x, edge_index, batch):
        # 1. H^{0} y las 4 capas de GNN
        x = self.embedding_h(x)
        x = self.layers(x, edge_index)

        # 2. Calculo del valor de lambda
        if self.choquet:
            root = root(lamb_func_np, self.choquet_weigth.detach().cpu().
                numpy())
            self.lamb = root
            self.suma = self.choquet_weigth.detach().cpu().numpy().sum()

        # 3. Metodo de agregacion
        if self.readout == "sum":
            x = global_add_pool(x, batch) #[batch_size, hidden_channels]
```

```
        elif self.readout == "max":
            x = global_max_pool(x, batch) #[batch_size, hidden_channels]
        elif self.readout == "mean":
            x = global_mean_pool(x, batch) #[batch_size, hidden_channels]
        elif self.readout == "choq":
            x = choquet_aggregation(x, self.choquet_weigth, root, batch)
        else:
            x = global_mean_pool(x, batch) #[batch_size, hidden_channels]

        return self.MLP_layer(x)

    def loss(self, pred, label):
        criterion = nn.CrossEntropyLoss()
        loss = criterion(pred, label)
        return loss
```

Anexo C. Selección de las tasas de aprendizajes para los experimentos

La Tabla 13 presenta los resultados obtenidos al realizar la sintonización de la tasa de aprendizaje en cada modelo y en todos los casos de prueba. Se utilizaron dos métodos de agregación: el promedio y el método de agregación de Choquet. En la columna **Prueba Prec.±s.d.**, se muestran los resultados empleando el promedio, mientras que en la columna **Prueba Prec.±s.d. Choquet**, se presentan los resultados utilizando el método de agregación de Choquet. Para el experimento, se seleccionó la tasa de aprendizaje que generó la mayor precisión en cada base de datos y en cada experimento, las cuales están resaltadas en negrita.

Tabla 13. Resultados de la sintonización de la tasa de aprendizaje

Caso de Prueba	Modelo	Tasa de Aprendizaje	Prueba Prec.±s.d.	Prueba Prec.±s.d. Choquet
MUTAG	GCN	0.005	82.92±7.52	82.4±7.63
MUTAG	GCN	0.0005	84.53±7.35	84.56±5.58
MUTAG	GCN	0.00005	**86.14±9.84**	**86.14±8.95**
MUTAG	GAT	0.005	77.66±5.19	**81.9±4.27**
MUTAG	GAT	0.0005	80.82±4.42	81.37±5.91
MUTAG	GAT	0.00005	**81.9±4.27**	75.06±8.69
MUTAG	GIN	0.005	85.09±8.15	86.61±9.54
MUTAG	GIN	0.0005	**87.72±5.42**	**87.72±5.96**
MUTAG	GIN	0.00005	85.09±8.15	86.2±8.18
ENZYMES	GCN	0.005	61.67±4.71	66±10.2
ENZYMES	GCN	0.0005	**67.67±6.16**	**66.33±5.72**
ENZYMES	GCN	0.00005	57.33±4.42	57.33±4.42
ENZYMES	GAT	0.005	**60.33±5.72**	**62.5±6.02**
ENZYMES	GAT	0.0005	59±7.57	56.33±9.77
ENZYMES	GAT	0.00005	32.17±8.57	31.17±9.95
ENZYMES	GIN	0.005	55.33±7.7	59.33±4.78
ENZYMES	GIN	0.0005	**63.33±5.32**	**61.67±5.48**
ENZYMES	GIN	0.00005	60.83±5.39	54.83±5.65
COX2	GCN	0.005	81.37±4.25	79.01±5.86
COX2	GCN	0.0005	80.96±5.11	80.54±4.22
COX2	GCN	0.00005	**82.67±4.04**	**81.8±3.82**
COX2	GAT	0.005	80.75±4.49	79.86±5.11
COX2	GAT	0.0005	80.74±4.92	**80.73±5.75**
COX2	GAT	0.00005	**81.82±3.55**	79.68±5.08
COX2	GIN	0.005	78.2±6.67	**81.82±5.11**
COX2	GIN	0.0005	**80.95±4.95**	79.02±3.23
COX2	GIN	0.00005	79.23±4.67	78.82±5.55
BZR	GCN	0.005	82.26±5.41	83.95±3.53
BZR	GCN	0.0005	84.2±3.33	**85.19±3.48**
BZR	GCN	0.00005	**85.18±3.53**	83.44±4.07
BZR	GAT	0.005	84.68±2.92	84.45±4.82
BZR	GAT	0.0005	**85.93±4.45**	**85.17±3.36**
BZR	GAT	0.00005	85.43±4.3	82.98±4.91
BZR	GIN	0.005	84.19±3.59	**84.93±3.42**
BZR	GIN	0.0005	**85.68±3.08**	84.21±3.99
BZR	GIN	0.00005	84.95±4.83	83.23±5.43
AIDS	GCN	0.005	98.45±0.52	98.25±0.87
AIDS	GCN	0.0005	**98.6±0.62**	**98.8±0.56**
AIDS	GCN	0.00005	98.3±0.75	98.3±0.75
AIDS	GAT	0.005	98.2±0.9	98.2±0.9
AIDS	GAT	0.0005	**98.4±0.77**	**98.4±0.77**
AIDS	GAT	0.00005	98.15±1	98.15±1
AIDS	GIN	0.005	98.3±0.71	98.3±0.71
AIDS	GIN	0.0005	**98.45±0.85**	**98.45±0.85**
AIDS	GIN	0.00005	98.35±0.63	98.35±0.63

Anexo D. Prueba de normalidad de Kolmogorov-Smirnov para los resultados de precisión en los conjuntos de prueba de los modelos de GNN

.1. Hipótesis de la prueba de Kolmogorov-Smirnov

La prueba de normalidad de Kolmogorov-Smirnov (Lilliefors, 1967) plantea las siguientes hipótesis para los resultados de precisión en los conjuntos de prueba de las 10 iteraciones de validación cruzada de 10 pliegues de cada uno de los modelos de GNN con y sin Choquet:

- H_0 (hipótesis nula): Los valores de precisión en los conjuntos de prueba siguen una distribución normal.
- H_1 (hipótesis alternativa): Los valores de precisión en los conjuntos de prueba no siguen una distribución normal.

.2. Nivel de significancia y valor p

Para llevar a cabo la prueba de normalidad de Kolmogorov-Smirnov, se estableció un nivel de significancia $\alpha = 0.05$. Por lo tanto, si $p \leq 0.05$, se rechaza la hipótesis nula en favor de la hipótesis alternativa (los valores de precisión no siguen una distribución normal). En caso contrario, no se rechaza la hipótesis nula (los valores de precisión siguen una distribución normal).

Tabla 14. Resultados de las pruebas de Kolmogorov-Smirnov para verificar el supuesto de normalidad

Caso de Prueba	Modelo	Valor P	Conclusión	Modelo con Choquet	Valor P	Conclusión
MUTAG	GCN	0.92	✓	GCN+CH	0.98	✓
MUTAG	GAT	0.27	✓	GAT+CH	0.92	✓
MUTAG	GIN	0.98	✓	GIN+CH	0.97	✓
BZR	GCN	0.89	✓	GCN+CH	0.88	✓
BZR	GAT	0.99	✓	GAT+CH	0.64	✓
BZR	GIN	0.51	✓	GIN+CH	0.75	✓
ENZYMES	GCN	0.67	✓	GCN+CH	1.00	✓
ENZYMES	GAT	0.20	✓	GAT+CH	0.19	✓
ENZYMES	GIN	0.95	✓	GIN+CH	0.98	✓
COX2	GCN	0.65	✓	GCN+CH	0.97	✓
COX2	GAT	0.90	✓	GAT+CH	0.88	✓
COX2	GIN	0.91	✓	GIN+CH	0.92	✓
AIDS	GCN	0.82	✓	GCN+CH	0.79	✓
AIDS	GAT	0.97	✓	GAT+CH	0.98	✓
AIDS	GIN	0.88	✓	GIN+CH	0.50	✓

.3. Resultados de la prueba de Kolmogorov-Smirnov

Después de realizar la prueba de normalidad de Kolmogorov-Smirnov, se presentaron los resultados en la Tabla 14. Se puede observar que tanto para el modelo con y sin Choquet, así como para cada caso de prueba, la hipótesis nula no se rechaza ya que el valor p siempre es mayor que el nivel de significancia α. Como resultado, se cumplen los supuestos de normalidad, lo cual se representa en la conclusión con un símbolo de verificación.