# Predicción de Aristas para Data Augmentation en Problemas de Clasificación de Nodos con Graph Neural Networks

Gonzalo Carmona López

# Predicción de Aristas para Data Augmentation en Problemas de Clasificación de Nodos con Graph Neural Networks

Gonzalo Carmona López

Memoria presentada como parte de los requisitos para la obtención del título de Máster Universitario en Matemáticas por la Universidad de Sevilla.

Tutorizada por

Prof. Tutora María del Rocío González Díaz
Prof. Tutor Eduardo Paluzo Hidalgo

# Resumen

En este trabajo, repasamos detalladamente el planteamiento de un problema de aprendizaje supervisado en general, sobre datos en forma de grafo en particular, y la construcción de la arquitectura de *Graph Neural Networks* para la resolución del mismo. En concreto, nos centramos en SEAL, una metodología basada en *Graph Neural Networks* para predicción de aristas. Además, explicaremos el método GAug-M para *data augmentation* en problemas de clasificación de nodos, que consiste en añadir o eliminar aristas en base a las probabilidades obtenidas de un modelo predictor de aristas, para enriquecer la información del grafo y aumentar el rendimiento del modelo de clasificación.

En este TFM se propone un nuevo uso de la metodología SEAL como modelo predictor de aristas junto con GAug-M para realizar *data augmentation* en un grafo. Evaluaremos el rendimiento de usar esta combinación, y lo compararemos con el rendimiento obtenido en [37] con otro modelo predictor de aristas, en los mismos *datasets* y con el mismo modelo de clasificación de nodos.

# Abstract

In this work, we explain in detail the formulation of a supervised learning problem in general, over graph data in particular, and the arquitecture of Graph Neural Networks for solving them. In particular, we focus on SEAL, a Graph Neural Network framework for predicting the existence of a link between nodes. We also talk about GAug-M, a method for data augmentation in node classification problems, which consists in adding or removing links based in precalculated probabilities, in order to improve the graph information, and also improve the performance of the node classification model. In this work, we propose a new use of SEAL as the link predictor, together with GAug-M for performing data augmentation in a graph. We evaluate the performance of this combination, and we compare it with the performance obtained in [37] with a different link prediction model, in the same datasets and with the same model for node classification.

Moreover, we explain how to use this arquitecture for performing data augmentation on a graph, whose nodes we want to classify, using the GAug-M method, explained and tested in [37] with other link prediction arquitectures different from SEAL. . We evaluate the performance when using SEAL as the link predictor, and we compare it with the efficiency obtained in [37] by using other predictor models, in the same datasets and with the same node classification model.

# Índice general

**1. Introducción** 6

**2. Grafos** 8

2.1. Teoría de Grafos . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

**3. Aprendizaje automático** 13

3.1. Aprendizaje Supervisado . . . . . . . . . . . . . . . . . . . . . . . . 13

3.1.1. Conjunto de datos de entrenamiento y de *test* . . . . . . . 14

3.2. Funciones de coste . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

3.3. Optimización del modelo . . . . . . . . . . . . . . . . . . . . . . . . 17

3.3.1. Descenso del gradiente y variantes . . . . . . . . . . . . . 17

3.4. Evaluación del modelo . . . . . . . . . . . . . . . . . . . . . . . . . 21

3.4.1. *Overfitting*, *underfitting* y conjunto de validación . . . . . 21

3.4.2. Métricas de evaluación . . . . . . . . . . . . . . . . . . . . 23

**4. Redes Neuronales** 29

4.1. Componentes de una red neuronal . . . . . . . . . . . . . . . . . . 29

4.1.1. Perceptrones . . . . . . . . . . . . . . . . . . . . . . . . . 29

4.1.2. Neuronas y funciones de activación . . . . . . . . . . . . . 31

4.2. El modelo de aprendizaje de redes neuronales . . . . . . . . . . . 33

4.2.1. Redes Neuronales . . . . . . . . . . . . . . . . . . . . . . . 33

4.2.2. Descenso del gradiente en redes neuronales y propagación hacia atrás . . . . . . . . . . . . . . . . . . . . . . . . . 36

**5. Graph Neural Networks 42**

5.1. Definición del modelo de Graph Neural Networks . . . . . . . . . 42

5.2. Graph Neural Network básica . . . . . . . . . . . . . . . . . . . . 45

5.3. Normalización del operador aggregate: Graph Convolutional Network . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

5.4. Deep Graph Convolutional Network . . . . . . . . . . . . . . . . 48

**6. El método SEAL 51**

6.1. Heurísticas de predicción de aristas . . . . . . . . . . . . . . . . . 52

6.1.1. Heurísticas de orden bajo . . . . . . . . . . . . . . . . . . 52

6.1.2. Heurísticas de orden alto . . . . . . . . . . . . . . . . . . 53

6.1.3. $\gamma$-heurísticas . . . . . . . . . . . . . . . . . . . . . . . . . 54

6.2. SEAL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

6.2.1. *Features* explícitas . . . . . . . . . . . . . . . . . . . . . . 57

6.2.2. Etiquetas de estructura . . . . . . . . . . . . . . . . . . . 58

6.2.3. *Node Embeddings* . . . . . . . . . . . . . . . . . . . . . . 59

6.2.4. Definición de SEAL . . . . . . . . . . . . . . . . . . . . . 61

6.3. Elección del modelo y evaluación del rendimiento de SEAL . . . 61

**7. Método GAug-M para *data augmentation* en grafos. 65**

7.1. Posibles estrategias para *data augmentation* en grafos . . . . . . 66

7.2. GAug-M . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

7.2.1. Motivación teórica . . . . . . . . . . . . . . . . . . . . . . 66

7.2.2. Definición de GAug-M . . . . . . . . . . . . . . . . . . . . 67

7.2.3. Evaluación y rendimiento de GAug-M . . . . . . . . . . . 69

**8. Evaluación de GAug-M usando SEAL 72**

8.1. Obtención de la matriz de probabilidades . . . . . . . . . . . . . 73

8.2. Aplicación de GAug-M y evaluación . . . . . . . . . . . . . . . . . 75

**9. Conclusiones y trabajos futuros 77**

# Capítulo 1

# Introducción

En los últimos años, el novedoso campo de estudio de las *Graph Neural Networks* (GNNs) ha avanzado enormemente, desarrollando nuevas y mejores arquitecturas para la resolución de problemas de aprendizaje supervisado sobre datos con estructura de grafo, como pueden ser; clasificar las funciones de cada proteína en un interactoma[1] [9] (clasificación sobre nodos de un grafo), predicción de la toxicidad o la solubilidad de una molécula [6], o encuadrarla en una categoría (clasificación/regresión sobre un grafo), además de predicción de la probabilidad de que dos usuarios sean amigos en una red social (obtención de la probabilidad de la existencia de una arista entre dos nodos [34]). La idea principal de un modelo de *Graph Neural Network* es la de actualizar los estados de los nodos en base a los estados de sus nodos vecinos, cuya información viene codificada por una función de agregación, para utilizarla junto con el estado anterior en una función de actualización.

En el ámbito de *data augmentation* (cuyo objetivo es aumentar o mejorar los datos de entrenamiento, para obtener un mejor rendimiento del modelo), campos de estudio como reconocimiento de imágenes aprovechan la naturaleza euclídea de los datos, para generar nuevos datos de entrenamiento, realizando operaciones como voltear la imagen, reflejarla, aplicar filtros visuales, etc. Sin embargo, en el caso de datos estructurados en grafos, los métodos convencionales de *data augmentation* no funcionan debido a que están definidos en un espacio no euclídeo [4]. Es por ello que se han ideado nuevos métodos que manipulan aristas y nodos para llevar a cabo la tarea de *data augmentation* sobre grafos.

En este trabajo, veremos uno de esos métodos, conocido como GAug-M [37], pensado para mejorar un grafo a la hora de enfrentarnos a un problema de clasificación de sus nodos. El método se basa en alguna estructura de *Graph*

[1] Conjunto de todas las interacciones moleculares dentro de una célula en particular.

*Neural Network* de la que podamos obtener la probabilidad de existencia de cualquier posible arista del grafo, para luego añadir al grafo las aristas con mayor probabilidad, y eliminar (si existían) las de menor probabilidad, de manera que se enriquezca la información del grafo, y el modelo sea capaz de distinguir más fácilmente las clases.

Originalmente, el rendimiento de GAug-M se evalúa utilizando *Variational Graph Auto Encoder* (VGAE) [14] como modelo predictor de aristas. Sin embargo, en este TFM, como novedad, nosotros emplearemos la metodología SEAL para obtener las probabilidades, y llevaremos a cabo experimentos con esta nueva combinación bajo las mismas condiciones que los realizados en [37] (misma GNN para clasificación de nodos, sobre los mismos *datasets*), y compararemos el rendimiento obtenido con el rendimiento original, para concluir si la mejora de SEAL respecto a otros modelos de predicción de aristas (incluido VGAE) se traduce en una mejora de rendimiento de GAug-M. Llevaremos a cabo estos experimentos mediante *scripts* de *python*, que pueden consultarse en el repositorio SEAL-GAugM, creado por mí específicamente para este trabajo.

En el Capítulo 2 recordamos las definiciones básicas de grafos, y mostramos algunos ejemplos de tipos de datos que se pueden estructurar como grafos, mientras que en el Capítulo 3 explicamos en qué consiste un problema de aprendizaje supervisado de forma general, y los conceptos y métodos necesarios para resolverlos. A continuación, explicamos las redes neuronales (Capítulo 4), modelos ampliamente conocidos y usados en aprendizaje automático, comúnmente encuadrados en aprendizaje profundo (*deep learning*). Esto nos permitirá comprender las *Graph Neural Networks*, de las que hablaremos en el Capítulo 5. A continuación, explicaremos y justificaremos la metodología SEAL para predicción de aristas (Capítulo 6), para luego hablar del método de *data augmentation* GAug-M (Capítulo 7), además de comentar el rendimiento de cada uno de ellos. Finalmente, llevaremos a cabo experimentos con la nueva combinación de GAug-M y SEAL (Capítulo 8), para comparar los resultados con los obtenidos en [37] (obtenidos con la combinación GAug-M y VGAE). Terminaremos la memoria con un capítulo dedicado a conclusiones y trabajos futuros (Capítulo 9).

# Capítulo 2

# Grafos

En este capítulo, recordamos brevemente qué es un grafo, su definición y variantes, y algunos conceptos básicos sobre los mismos. Los datos estructurados en grafos constituyen la base del trabajo, ya que el planteamiento de un problema de aprendizaje supervisado sobre ellos nos lleva a la construcción de una *Graph Neural Network*.

## 2.1. Teoría de Grafos

La teoría de grafos es un campo de las matemáticas que surge en el siglo XVIII y estudia una estructura de datos que modela relaciones entre elementos. A continuación, recordamos algunas definiciones básicas sobre grafos. Para una aproximación más detallada, se puede consultar [10, 28].

Denotaremos por $\mathcal{P}(X)$ al conjunto de las partes de un conjunto $X$, y usaremos $|\cdot|$ para el cardinal de un conjunto.

**Definición 2.1.1.** Un grafo simple es un par de conjuntos finitos $(\mathcal{V}, \mathcal{E})$ tales que $\mathcal{V} \neq \emptyset$ y $\mathcal{E} \subseteq \{x \in \mathcal{P}(\mathcal{V}) \mid |x| = 2\}$. Denotaremos por $\mathcal{V}$ al conjunto de vértices o nodos, y como $\mathcal{E}$ al conjunto de aristas. Decimos que dos nodos $v_1$ y $v_2$ son adyacentes si $\{v_1, v_2\} \in \mathcal{E}$.

**Definición 2.1.2.** Sea $G = (\mathcal{V}, \mathcal{E})$ un grafo simple. Un subgrafo de $G$ es un grafo $G' = (\mathcal{V}', \mathcal{E}')$ de manera que $\mathcal{V}' \subseteq \mathcal{V}$, y $\mathcal{E}' \subseteq \{\{u, v\} \in \mathcal{E} \mid u, v \in \mathcal{V}'\}$

Es decir, un subgrafo está formado por un subconjunto de nodos de un grafo, y un subconjunto de aristas entre esos nodos.

**Ejemplo 2.1.3.** Consideremos el grafo simple $G = (\mathcal{V}, \mathcal{E})$, $\mathcal{V} = \{a, b, c, d, e, f\}$ y $\mathcal{E} = \{\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, f\}\}$. Normalmente, representamos gráficamente los nodos como puntos. Los elementos de $\mathcal{E}$ se representan como segmentos con los nodos como puntos finales, ya que sus elementos son conjuntos no ordenados. En la Figura 2.1 se ilustra el grafo $G$.

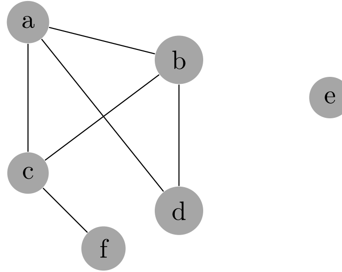


Figura 2.1: Representación del grafo simple $G$ del Ejemplo 2.1.3.

Utilizamos los grafos para representar información correspondiente a situaciones reales en las que tenemos elementos relacionados entre sí. Por ejemplo, un grafo simple como el de la Figura 2.1 puede representar una serie de ciudades conectadas entre sí por redes ferroviarias, de manera que los nodos son las ciudades, que estarán unidos si existe conexión ferroviaria entre las ciudades.

Los grafos simples no son el único tipo de grafo existente. Veamos nuevas definiciones que permiten ampliar los casos y situaciones que se pueden describir mediante un grafo.

**Definición 2.1.4.** Un grafo dirigido simple, o digrafo simple, es un par de conjuntos finitos $(\mathcal{V}, \mathcal{E})$ tales que $\mathcal{V} \neq \emptyset$ y $\mathcal{E} \subseteq \{(a, b) \in \mathcal{V} \times \mathcal{V} \mid a \neq b\}$.

Nótese que la diferencia con la Definición 2.1.1 es que los elementos de $\mathcal{E}$ son pares ordenados y no subconjuntos de orden 2, por lo que ahora el orden de los nodos importa a la hora de definir aristas. Además, exigimos que los pares ordenados sean de elementos distintos de $\mathcal{V}$, lo que impide aristas que partan e incidan sobre el mismo nodo.

**Ejemplo 2.1.5.** Consideremos el grafo dirigido simple $G = (\mathcal{V}, \mathcal{E})$ con el mismo conjunto de nodos que en el Ejemplo 2.1.3, pero con $\mathcal{E} = \{(a, b), (b, a), (c, a), (b, c), (a, d), (c, f)\}$. Representamos la dirección de estas aristas con una flecha:

Este digrafo simple podría representar una red social, donde los nodos representan a los usuarios, y existe una arista dirigida entre, por ejemplo, los nodos $a$ y $b$ si el usuario $a$ es seguidor del usuario $b$, cosa que no nos permitiría representar un grafo simple.

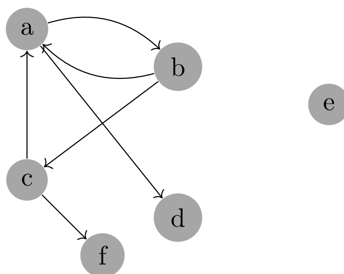


Figura 2.2: Representación del digrafo simple $G$ del Ejemplo 2.1.5.

**Definición 2.1.6.** Un multigrafo (dirigido) es un grafo (dirigido), en el cual permitimos múltiples aristas (con la misma dirección) entre un mismo par de nodos, además de aristas que conecten un nodo consigo mismo (bucles). Formalmente, permitimos en ambos casos que $\mathcal{E}$ sea un multiconjunto, y permitimos que contenga multiconjuntos del tipo $\{a, a\}$ en el caso no dirigido, o pares del tipo $(a, a)$ en el caso dirigido.

El objetivo de este trabajo es el estudio de un modelo de aprendizaje automático, las *Graph Neural Networks*. Aunque este modelo puede adaptarse a distintos tipos de grafos, nos limitaremos a la definición que se basa en grafos simples. Así que, salvo que se diga lo contrario, los grafos que consideraremos son grafos simples.

**Definición 2.1.7.** Sea $G = (\mathcal{V}, \mathcal{E})$ un grafo simple. Definimos el grado de un nodo $v$ como

$$\delta(v) = \big|\{x \in \mathcal{E} \mid v \in x\}\big|,$$

es decir, la cantidad de aristas incidentes en $v$ (o la cantidad de nodos adyacentes a $v$).

**Definición 2.1.8.** Sea $G = (\mathcal{V}, \mathcal{E})$ un grafo simple, cuyos nodos dotamos de un orden $\mathcal{V} = \{v_1, ..., v_n\}$, donde $n = |\mathcal{V}|$. Definimos la **matriz de adyacencia** de $G$ como una matriz $A = (a_{ij})_{1\leq i,j\leq n}$ de tamaño $n \times n$, de manera que

$$a_{ij} = \begin{cases} 1, & \text{si } v_i \text{ y } v_j \text{ son adyacentes} \\ 0, & \text{en caso contrario.} \end{cases}$$

Observemos que la matriz de adyacencia de un grafo simple es simétrica, que los elementos de la diagonal son ceros y que la suma de la columna o fila $i$ es igual a $\delta(v_i)$.

**Definición 2.1.9.** Sea $G = (\mathcal{V}, \mathcal{E})$ un grafo simple. Un camino entre dos nodos $v, w \in \mathcal{V}$ es una sucesión de nodos $C = (v_0, ..., v_n)$ de manera que $v_0 = v$, $v_n = w$ y $\{v_i, v_{i+1}\} \in \mathcal{E}$ para todo $i = 0, ..., n-1$. Decimos que $n$ es la longitud

del camino, y utilizamos la notación $|C|$. Si (no) existe un camino entre dos nodos $v$ y $w$, decimos que $v$ y $w$ (no) pertenecen a la misma componente conexa de $G$. Esto nos define una partición en componentes conexas de cualquier grafo $G = G_1 \cup \cdots \cup G_k$.

**Definición 2.1.10.** Dados dos nodos $v$ y $w$ de un grafo $G$, definimos la distancia entre ellos como

$$d(v,w) = \begin{cases} \text{mín}\{|C| : C \text{ camino entre } v \text{ y } w\}, & \text{si existe camino entre } v \text{ y } w. \\ \infty, & \text{en caso contrario.} \end{cases}$$

Utilizamos también la notación $\mathcal{N}(u)$ para referirnos al conjunto de vecinos de un nodo $u$, es decir, $\mathcal{N}(u) = \{v \in \mathcal{V} \mid \{u,v\} \in \mathcal{E}\}$.

**Ejemplo 2.1.11.** Si tomamos el grafo $G = (\mathcal{V}, \mathcal{E})$ del Ejemplo 2.1, y ordenamos los nodos alfabéticamente, la matriz de adyacencia de $G$ es

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

Cada columna (fila) tiene como suma los grados correspondientes:

$$\delta(a) = \delta(b) = \delta(c) = 3,\ \delta(d) = 2,\ \delta(e) = 0,\ \delta(f) = 1,$$

y el grafo se divide en dos componentes conexas:

$$G = G_1 \cup G_2,\ G_1 = (\{a,b,c,d,f\}, \mathcal{E}),\ G_2 = (\{e\}, \emptyset).$$

El conjunto de vecinos de $a$ es $\mathcal{N}(a) = \{b,c,d\}$.

El siguiente teorema muestra cómo las potencias de la matriz de adyacencia almacenan información importante sobre caminos en un grafo, cosa que nos servirá en algunas definiciones más adelante.

**Teorema 2.1.12** ([5])**.** *Sea $G$ un grafo, y sea $A$ su matriz de adyacencia. Entonces, la entrada $(i,j)$ de la matriz $A^n$, que denotamos $a_{ij}^{(n)}$, indica cuántos caminos de longitud $n$ hay entre los nodos $v_i$ y $v_j$, para todo $n \geq 1$.*

*Demostración.* Lo probamos por inducción en $n$. Si $n = 1$, $A^n = A$, que por definición nos indica si existe o no arista (camino de longitud 1) entre los nodos. Supongamos ahora que se tiene el resultado para $n$, es decir, $a_{ij}^{(n)}$ indica cuántos caminos de longitud $n$ hay entre $v_i$ y $v_j$. El número de caminos de longitud $n+1$

entre $v_i$ y $v_j$ es igual al número de caminos de longitud $n$ entre $v_i$ y cualquier nodo $v$ adyacente a $v_j$. Este número es exactamente el resultado de multiplicar la fila $i$ de $A^n$ por la columna $j$ de $A$, es decir, la entrada $a_{ij}^{(n+1)}$ de la matriz $A^n A = A^{n+1}$. □

Veamos ahora algunas definiciones de subgrafos, importantes para comprender las heurísticas de predicción de aristas, de las que hablaremos en la Sección 6.1.

**Definición 2.1.13.** Sea $G = (\mathcal{V}, \mathcal{E})$ un grafo y sea $\mathcal{U} \subseteq \mathcal{V}$. Definimos el subgrafo inducido por $G$ y el conjunto de nodos $\mathcal{U}$ como $G' = (\mathcal{U}, \mathcal{E}')$, donde

$$\mathcal{E}' = \{\{u, v\} \in \mathcal{E} \mid u, v \in \mathcal{U}\}$$

**Definición 2.1.14.** Dado un grafo $G = (\mathcal{V}, \mathcal{E})$ y un par de nodos cualesquiera $x, y \in \mathcal{V}$, definimos el subgrafo $h$-recubridor del par $(x, y)$ como el subgrafo $G^h_{x,y}$ inducido por $G$ y el conjunto de nodos

$$\{i \in \mathcal{V} \mid d(i, x) \leq h \text{ ó } d(i, y) \leq h\}.$$

**Ejemplo 2.1.15.** En el grafo del Ejemplo 2.1.3, el subgrafo recubridor $G^1_{a,b}$ (ver Figura 2.3) tiene como conjunto de vértices $\{a, b, c, d\}$, y como conjunto de aristas,

$$\{\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}\}.$$

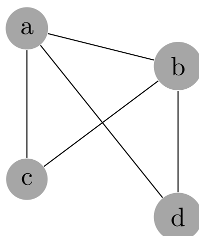


Figura 2.3: Representación del subgrafo recubridor $G^1_{a,b}$ del Ejemplo 2.1.15.

# Capítulo 3

# Aprendizaje automático

El aprendizaje automático (también llamado *Machine Learning* o *Inteligencia Artificial*, según la corriente o conjunto de modelos a los que nos refiramos) surge a finales de los años 50 y años 60, e incluye desde árboles de decisión a redes neuronales, entre otros muchos modelos. En este capítulo, introduciremos conceptos sobre aprendizaje automático supervisado, como por ejemplo, división en conjuntos de entrenamiento y *test*, funciones de coste, optimización y evaluación de un modelo, necesarios para entender el funcionamiento de una *Graph Neural Network*. Para un estudio más en profundiad, se pueden consultar [17], [18] y [8].

## 3.1. Aprendizaje Supervisado

De manera general, un modelo de aprendizaje automático puede resolver problemas de muy distinto tipo, como problemas de regresión o clasificación. Además, según la información que se le aporte al modelo, hablaremos de aprendizaje supervisado o no supervisado. Un problema de aprendizaje supervisado parte de un conjunto de datos para el que se conoce lo que debería devolver el modelo. En el caso de los problemas de clasificación, se conocerán las etiquetas o clases de los datos y, en el caso de problemas de regresión, una magnitud numérica.

### 3.1.1. Conjunto de datos de entrenamiento y de *test*

En aprendizaje supervisado, un modelo se ajusta a un conjunto de datos finito del que conocemos el valor de la variable objetivo (conjunto de entrenamiento) con el fin de poder predecir o clasificar datos nuevos, lo que se conoce como capacidad de generalización. El modelo puede entenderse como una función que depende de un conjunto de parámetros que se ajustan durante un proceso de optimización. Se parte de un conjunto de parámetros iniciales que se suelen llamar pesos y que, a partir de una función objetivo o error, se ajustan durante un proceso de minimización.

El objetivo en aprendizaje automático es la búsqueda de un modelo capaz de predecir o aproximar una función "ideal" $f\colon X \longrightarrow Y$, que asocia a cada vector de datos de entrada el valor correcto (real) de la variable objetivo (p.ej., la clase a la que pertenece, o el valor de la magnitud numérica que queremos predecir). Conocemos el valor de esta función $f$ para los datos de entrenamiento, lo que nos sirve para llevar a cabo un proceso de optimización de los parámetros del modelo. Formalmente, podemos definir un modelo como una función $g_\theta\colon X \longrightarrow Y$, que depende de un conjunto de parámetros $\theta \subseteq \Theta$, donde $\Theta$ es el espacio de posibles parámetros. Estos parámetros se determinan escogiendo unos valores iniciales[1], y optimizándolos usando el conjunto de entrenamiento, junto con una función error que compara $g_\theta$ con $f$, y un algoritmo que minimiza esta función de error, ajustando los parámetros durante el proceso, hasta que se obtenga el rendimiento deseado o se haya repetido el algoritmo una cantidad de veces previamente prefijada (criterios de parada).

**Definición 3.1.1.** Un conjunto de datos de entrenamiento,

$$T = \{(x_1, y_1), ..., (x_N, y_N)\},$$

está formado por pares, donde los $x_i$ son vectores de un cierto conjunto de datos de entrada, $X \subseteq \mathbb{R}^n$ (decimos que $n$ es el número de variables de entrada, o *features*), y los $y_i$ son los valores de la variable objetivo asociados a cada uno de los vectores.

En la práctica, el conjunto de datos de entrenamiento nunca está formado por la totalidad de los datos de los que conocemos la variable objetivo, sino que se realiza una división de los mismos en conjunto de entrenamiento y conjunto de *test*. Este último sirve para, una vez entrenado el modelo, medir el rendimiento del mismo, para tener una idea de lo buenas que serán las predicciones y compararlo con otros modelos. De esta manera, conseguimos simular el rendimiento del modelo con datos reales de los que no conocemos la variable a predecir, ya que en la fase de entrenamiento, el modelo solo aprende de información que proviene del conjunto de entrenamiento, mientras que la del conjunto

[1] Existen distintas formas de tomar los valores iniciales, aunque suelen ser escogidos de manera aleatoria.

de *test* es información nueva. Además de la división en entrenamiento y test, hay otras técnicas de evaluación y entrenamiento como la validación cruzada (*cross-validation*).

Es muy importante evitar que el modelo obtenga información del conjunto de *test*, ya que entonces no simularía correctamente su comportamiento con datos reales. Cuando esto ocurre, diremos que se está produciendo ***data leakage***. Veremos más adelante que hay situaciones en las que tenemos que dividir nuestros datos hasta en 3 partes para evitar este fenómeno.

En el caso particular de las *Graph Neural Networks*, que introduciremos en el Capítulo 5, veremos que no se cumple del todo el requisito de no obtener información de los datos de *test*, debido a su arquitectura, a pesar de que sigue los principios de un problema de aprendizaje supervisado (optimizamos sobre el conjunto de entrenamiento, y evaluamos sobre el de *test*). Es por ello que los consideramos modelos de aprendizaje semi-supervisado.

## 3.2. Funciones de coste

Hemos hablado de funciones que miden el error que comete $g_\theta$ al aproximar $f$. Estas funciones son las funciones de coste, que estudiamos a continuación.

**Definición 3.2.1.** Una función de coste (*loss function*), $\mathcal{L}$, se define sobre el espacio de parámetros $\Theta$, y mide el error que cometen las predicciones del modelo sobre el conjunto de entrenamiento, cosa que podemos hacer ya que conocemos los $y_i$ sobre ese conjunto.

$$\begin{aligned} \mathcal{L}\colon \Theta &\longrightarrow \mathbb{R}_{\geq 0} \\ \theta &\longrightarrow \varepsilon_T \end{aligned}$$

Damos a continuación los ejemplos más importantes de funciones de coste, tanto para:

- **Problemas de regresión**, donde la variable objetivo es un valor numérico continuo, y el modelo calcula el valor de la misma correspondiente a una observación.
- **Problemas de clasificación**, donde la variable objetivo toma un número finito de valores, que identificamos con clases, y el modelo calcula la probabilidad de que una observación pertenezca a cada clase.

**Ejemplo 3.2.2.** La función de **error medio absoluto**, o MAE (*mean absolute error*) se define como:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} |g_\theta(x_i) - y_i|, \tag{3.1}$$

donde $N$ es la cardinalidad de $T$.

**Ejemplo 3.2.3.** La función de **error cuadrático medio**, o MSE (*mean squared error*) se define como:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} (g_\theta(x_i) - y_i)^2. \tag{3.2}$$

Ambas funciones de coste sirven para medir el error en problemas de regresión, pero son muy parecidas entre sí. Sin embargo, la diferencia entre las dos es que la función de error cuadrático medio penaliza más los errores graves del modelo, ya que, en ese caso, el cuadrado de la diferencia es mayor que la diferencia absoluta (generalmente, se tiene $|g_\theta(x_i) - y_i| > 1$ si el error se considera grave). Por otro lado, la función de error medio absoluto penaliza más un modelo que cometa muchos errores pequeños ($|g_\theta(x_i) - y_i| < 1$), ya que elevar al cuadrado estas cantidades las hace más pequeñas.

Por lo tanto, la elección de una función u otra dependerá de cada problema en particular, y del objetivo que busquemos cumplir con el modelo. Si queremos un modelo que, sin ser el más preciso, sea fiable y no cometa errores muy graves, utilizaremos el error cuadrático medio. Sin embargo, si podemos permitirnos pocos errores graves pero buscamos mucha precisión en general, utilizaremos el error medio absoluto.

Veamos ahora una función de coste más adecuada para medir el error en un problema de clasificación.

**Ejemplo 3.2.4.** La función de coste *cross-entropy* se define como:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N}\sum_{k=1}^{M} y_{i,k} \log(g_\theta(x_i)_k), \tag{3.3}$$

donde $M$ es el número de valores que puede tomar la variable objetivo (número de clases), $y_{i,k} = 1$ si la observación $x_i$ pertenece a la clase $k$, e $y_{i,k} = 0$ en caso contrario, y $g_\theta(x_i)_k$ es la probabilidad (calculada por el modelo, y estrictamente positiva) de que la observación $x_i$ pertenezca a la clase $k$.

Parece una definición complicada, pero analicémosla para entender la lógica que hay detrás:

- Los valores $g_\theta(x_i)_k$ se encuentran en el intervalo $(0, 1]$ al ser probabilidades no nulas, por lo que $\log(g_\theta(x_i)_k) \leq 0$, y por ello incluimos un signo menos antes del sumatorio sobre las clases.

- Por otro lado, sólo uno de los $y_{i,k}$ es igual a 1 (el correspondiente a la clase a la que pertenece $x_i$), mientras que el resto es igual a 0. Con lo cual, tenemos las siguientes situaciones:

  - Si el modelo hace una buena predicción, es decir, la probabilidad $g_\theta(x_i)_k$ es alta, donde $k$ es tal que $y_{i,k} = 1$, el valor $-\log(g_\theta(x_i)_k)$ es cercano a cero, por lo que el valor de la función de coste es pequeño.
  - Si, en su lugar, $g_\theta(x_i)_k$ es baja, el valor $-\log(g_\theta(x_i)_k)$ es más grande que en el caso anterior, por lo que la función de coste penaliza esta mala predicción.

## 3.3. Optimización del modelo

Ahora que sabemos el planteamiento de un problema de aprendizaje supervisado, la siguiente pregunta es: ¿cómo lo resolvemos? ¿Cómo actualizamos los parámetros para minimizar la función de coste?

Lo primero en lo que podríamos pensar es en encontrar el mínimo de manera analítica, usando optimización de funciones en varias variables. Sin embargo, esto no va a funcionar en general, ya que lo normal es que el conjunto de parámetros $\theta$ sea grande, y no dos o tres variables, como la mayoría de funciones de las cuales obtenemos los máximos y mínimos en cálculo en varias variables. Por lo tanto, lo normal es que tengamos una expresión que no sea posible optimizar de manera analítica. Por otro lado, queremos que el aprendizaje del modelo sea un proceso automático, e implementable en un ordenador. Es por ello que aplicaremos un algoritmo heurístico para aproximar un mínimo local que no nos proporcionará la solución óptima al problema de minimización, es decir, en general alcanzaremos un mínimo local ya que la función no tiene por qué ser convexa.

### 3.3.1. Descenso del gradiente y variantes

El algoritmo del que hablamos es el método de descenso del gradiente. Este método se basa en la siguiente idea: el gradiente de la función coste, $\nabla\mathcal{L}$ es un vector que indica, en cada punto, la dirección hacia la cual la función $\mathcal{L}$ crece. Basándonos en esta idea, formalizaremos lo siguiente: partimos de un punto cualquiera, calculamos el gradiente de la función en ese punto, y nos movernos un poco en la dirección contraria al gradiente hacia un nuevo punto

(donde esperamos que la función coste tenga un valor inferior al del punto anterior), en el que volvemos a repetir el proceso, hasta que se cumpla cierto criterio de parada que puede ser, entre otros, que la función coste alcance un valor aceptable (tolerancia), o simplemente hacer un número fijo de iteraciones. Podemos imaginar este proceso en dos dimensiones más fácilmente, donde el mínimo local se encuentra en el punto más bajo de un valle de la función, y el movimiento sería el de una pelota que cae hacia el mínimo en cada caso.

Para entender el algoritmo, recordemos algunas definiciones de cálculo en varias variables.

**Definición 3.3.1.** Sea $f\colon \mathbb{R}^n \to \mathbb{R}$. Definimos el gradiente de $f$ como

$$\nabla f = \left( \frac{\partial f}{\partial x_1}, ..., \frac{\partial f}{\partial x_n} \right),$$

donde cada coordenada es la derivada parcial

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, ..., x_i + h, ..., x_n) - f(x_1, ...x_n)}{h}.$$

En nuestro problema, tendríamos

$$\nabla \mathcal{L} = \left( \frac{\partial \mathcal{L}}{\partial \theta_1}, ..., \frac{\partial \mathcal{L}}{\partial \theta_k} \right),$$

donde $\theta = \{\theta_1, ..., \theta_k\}$ es el conjunto de parámteros del modelo.

Utilizaremos una herramienta clave del cálculo en varias variables, que es la aproximación lineal de la función en un punto. Podemos asumir que, dado un punto $\theta_0$ (vector de $k$ componentes), y un entorno suficientemente pequeño del mismo, se tiene

$$\mathcal{L}(\theta) \approx \mathcal{L}(\theta_0) + \nabla \mathcal{L} \cdot (\theta - \theta_0)$$

Y por lo tanto, si llamamos $\Delta\theta = \theta - \theta_0$, tenemos que:

$$\mathcal{L}(\theta) - \mathcal{L}(\theta_0) \approx \nabla \mathcal{L} \cdot \Delta\theta \tag{3.4}$$

Por tanto, esta ecuación nos permite escoger $\Delta\theta$ de manera que $\Delta\mathcal{L}$ sea negativo, reduciendo así el valor de la función objetivo. No podemos realizar cambios muy grandes cada vez, ya que no debemos alejarnos demasiado del punto debido a la aproximación lineal de $\Delta\mathcal{L}$ que estamos utilizando, por lo que restringimos: $||\Delta\theta|| = \varepsilon$, $\varepsilon > 0$.Teniendo en cuenta esta restricción, es lógico pensar que no solo queremos una elección de $\Delta\theta$ que consiga este objetivo, sino la mejor.

**Teorema 3.3.2** ([18]). *La elección de $\nabla\theta$ que minimiza $\Delta\mathcal{L}$ en la ecuación 3.4, sujeta a $||\Delta\theta|| < \varepsilon,\ \varepsilon > 0$ es*

$$\Delta\theta = -\mu\nabla\mathcal{L},\ \mu = \frac{\varepsilon}{||\nabla\mathcal{L}||} \tag{3.5}$$

*Demostración.* Tenemos, por la desigualdad de Cauchy-Bunyakovsky-Schwarz, que

$$|\nabla\mathcal{L}\cdot\Delta\theta| \leq ||\nabla\mathcal{L}||||\Delta\theta|| = ||\nabla\mathcal{L}||\varepsilon$$

y sabemos que la igualdad, que nos daría el mayor cambio que podemos conseguir en el valor absoluto de $\nabla\mathcal{L}\cdot\Delta\theta$, se da cuando los vectores son linealmente dependientes, es decir, $\Delta\theta = \pm\mu\cdot\nabla\mathcal{L}$, $\mu \in \mathbb{R}_{>0}$. Naturalmente, escogemos el signo negativo, ya que queremos minimizar $\nabla\mathcal{L}\cdot\Delta\theta$. Por otra parte, por la restricción, tenemos que

$$||\Delta\theta|| = \varepsilon = \mu||\nabla\mathcal{L}|| \Rightarrow \mu = \frac{\varepsilon}{||\nabla\mathcal{L}||}.$$

□

Llamamos a $\mu$ coeficiente de aprendizaje (o *learning rate*), y normalmente lo utilizaremos como valor por defecto para determinar los cambios $\Delta\theta$ (escogido $\mu$, $\varepsilon$ está unívocamente determinado por la ecuación 3.5). Este valor no debe escogerse muy grande, dado que no podemos hacer cambios muy grandes debido a la aproximación de la ecuación 3.4 que utilizamos para $\Delta\mathcal{L}$. Sin embargo, tampoco debe escogerse demasiado pequeño, ya que eso hará que el algoritmo trabaje lentamente, es decir, que necesite muchas iteraciones hasta satisfacer alguna condición de parada.

En la práctica, se escoge un valor de entre una serie de valores en escala logarítmica, (p. ej., $\mu \in \{0.001, 0.01, 0.1, 1, 10\}$). Para ello, se entrena el modelo con cada una de las posibilidades, y se comprueba el valor de la función coste asociado a cada uno en el conjunto de *test*, para elegir el valor del parámetro que nos de mejor rendimiento. Parámetros del mismo tipo que el *learning rate*, distintos a los parámetros intrínsecos del modelo, $\theta$, que no se optimizan en el proceso de aprendizaje, sino que tenemos que escogerlos previamente, se conocen como **hiperparámetros** del modelo.

Sin embargo, para escoger el valor de un hiperparámetro como hemos descrito, es necesario tener un tercer conjunto de datos etiquetados para evaluar el modelo, que llamamos **conjunto de validación**. Esto se debe a que, al usar el conjunto de *test* para escoger los hiperparámetros del modelo final, estamos obteniendo información del mismo, por lo que evaluar el modelo con el conjunto de *test* daría lugar a *data leakage*. Un tercer conjunto distinto a los otros dos del que no hayamos obtenido ninguna información resuelve este problema. Una práctica común es dividir los datos disponibles de manera que un 50 % sea el

conjunto de entrenamiento, y cada 25 % restante sea cada uno de los otros dos conjuntos.

**Algoritmo 3.3.1** (Descenso del gradiente)**.** Dado un modelo de aprendizaje supervisado, inicializado con parámetros $\theta$, un conjunto de entrenamiento $T$ de tamaño $N$, una función de coste $\mathcal{L}$, un coeficiente de aprendizaje $\mu$ y, o bien un entero $k$ (número de iteraciones) o bien un $\varepsilon > 0$ (tolerancia), hacer:

1. Calcular $\nabla\mathcal{L} = \sum_{x\in T} \nabla\mathcal{L}_x$[2].
2. Actualizamos cada componente del vector de parámetros según la regla

$$\theta_i \longrightarrow \theta_i' = \theta_i - \mu\frac{\partial\mathcal{L}}{\partial\theta_i}$$

3. Si el criterio de parada es tolerancia, comprobar si $\mathcal{L}(\theta^{'}) \leq \varepsilon$. En caso afirmativo, parar y devolver $\theta^{'}$. En caso contrario, repetir 1. Si el criterio de parada es el número de iteraciones, repetir 1., y parar y devolver $\theta^{'}$ en la $k$-ésima iteración.

Esta aproximación inicial a un buen algoritmo presenta un problema: para calcular el gradiente $\nabla\mathcal{L}$, es necesario calcular cada uno de los gradientes $\nabla\mathcal{L}_x$. Normalmente, los conjuntos de entrenamiento son grandes, para que el modelo disponga de suficientes datos para aprender los patrones y extraer suficiente información para ser predictivo. Por lo tanto, este cálculo puede hacer que nuestro algoritmo sea muy ineficiente.

Para solucionarlo, se usa una versión ligeramente distinta de este algoritmo, el descenso del gradiente estocástico. Esta consiste en tomar aleatoriamente un subconjunto $\{X_1, ..., X_m\}$ de $T$, relativamente pequeño en comparación, y entrenar el modelo con el mismo para reducir el número de cálculos. Una vez hecho esto, desechamos estos datos de entrenamiento, y tomamos otro subconjunto de $T$ disjunto del anterior, y volvemos a entrenar el modelo. Cuando ya no podemos escoger otro subconjunto de tamaño $m$ de $T$, decimos que termina una época de aprendizaje. A continuación, podemos volver a aplicar este proceso un número determinado de veces, ya que cada vez los subconjuntos se escogen aleatoriamente, es decir, entrenar el modelo durante varias épocas.

Esta idea mejora considerablemente la eficiencia, pero también tiene sus contras. En concreto, no calculamos el valor real del gradiente $\nabla\mathcal{L}$, sino una aproximación basada en el subconjunto tomado aleatoriamente. Sin embargo, es esperable que sea una aproximación suficientemente buena si escogemos subconjuntos de un tamaño $m$ suficiente.

[2] Asumimos que la función coste se puede escribir como suma de costes asociados a cada elemento del conjunto de entrenamiento. En la práctica esto ocurre prácticamente en la totalidad de los casos, como en los ejemplos de funciones coste que hemos visto.

**Algoritmo 3.3.2** (Descenso del gradiente estocástico)**.** Dado un modelo de aprendizaje supervisado, inicializado con parámetros $\theta$, un conjunto de datos de entrenamiento $T$ de tamaño $N$, una función de coste $\mathcal{L}$, un coeficiente de aprendizaje $\mu$, un entero positivo $m < N$ (tamaño de los subconjuntos) y, o bien un entero $k$ o bien un $\varepsilon > 0$, hacer $E$ veces (épocas de entrenamiento):

1. Generar aleatoriamente $\{X_1, ..., X_m\} \subseteq D$.
2. Aplicar el Algoritmo 3.3.1 con $D = \{X_1, ..., X_m\}$ y con $k$ iteraciones o tolerancia $\varepsilon$ como criterio de parada.
3. Actualizar $D \longrightarrow D' = D - \{X_1, ..., X_m\}$. Si $|D^{'}| > m$, repetir 1., en caso contrario, terminar y devolver los parámetros $\theta^{'}$.

Otra variante, conocida como *on-line leraning*, se da cuando tomamos $m = 1$, es decir, entrenamos el modelo con una única muestra cada vez que llegamos al paso 2. Este método se utiliza cuando se tiene un modelo listo para hacer predicciones sobre datos reales, pero también aparecen nuevos datos de entrenamiento con el paso del tiempo. Con el aprendizaje en línea, el modelo puede mantenerse actualizado cada vez que aparece un nuevo dato de entrenamiento, en lugar de reentrenarlo cada cierto período de tiempo añadiendo los nuevos datos, lo que le permite estar completamente actualizado.

## 3.4. Evaluación del modelo

Una vez que hemos finalizado el entrenamiento del modelo, el siguiente paso es evaluar su rendimiento. Esto se lleva a cabo calculando ciertos valores predefinidos sobre el conjunto de *test*, que llamamos **métricas de evaluación**, y que nos dan una idea del rendimiento de las distintas cualidades del modelo.

### 3.4.1. *Overfitting*, *underfitting* y conjunto de validación

Los cálculos de las métricas de evaluación deben realizarse sobre el conjunto de *test*, ya que si los hiciésemos sobre el conjunto de entrenamiento, estaríamos evaluando únicamente la capacidad del modelo de ajustarse a esos datos, en lugar de su capacidad de generalizar a nuevas observaciones. Cuando un modelo se ajusta de manera excesiva al conjunto de entrenamiento, y no generaliza bien a nuevos datos (su evaluación en el conjunto de entrenamiento es buena, pero en el de *test* es mala), decimos que se produce ***overfitting*** (el modelo se sobreajusta) situación que sugiere reducir la complejidad del modelo. Por el contrario, si el modelo tiene un mal rendimiento en ambos conjuntos, decimos

que se produce ***underfitting***, lo que indica que el modelo no está aprendiendo correctamente de los datos de entrenamiento, y que es necesario aumentar la complejidad del modelo (flexibilidad, número de *features* de las que aprende, etc).

En definitiva, debemos modificar los hiperparámetros, y escoger una combinación adecuada para obtener la versión final del modelo. Sin embargo, si evaluamos los modelos obtenidos con las distintas combinaciones de hiperparámetros en el conjunto de *test* para escoger la combinación óptima, y usamos esa evaluación como medidor final del rendimiento del modelo, no sería del todo fiable, ya que estaríamos evaluando una vez más la capacidad del modelo de ajustarse a los datos de *test*, en lugar de su capacidad de generalizar a los nuevos datos. Es por ello que es necesario utilizar un tercer conjunto de datos (conjunto de validación) para evaluar cada una de las combinaciones de hiperparámetros, y así poder dar la evaluación final usando el conjunto de *test*, para dar una estimación fiable del rendimiento del modelo. Con lo cual, el proceso para entrenar y escoger un modelo podría ser el siguiente:

1. Separar el conjunto de datos en conjuntos de entrenamiento, *test* y validación. Comúnmente, se reservan la mayoría de los datos para el entrenamiento, de manera que el modelo tenga más datos de donde aprender. Un ejemplo de separación sería utilizar el 60 % de los datos para el conjunto de entrenamiento, un 20 % para el de *test* y el 20 % restante para el de validación.

2. Escoger posibles combinaciones de hiperparámetros del modelo, como pueden ser, si se trata por ejemplo de una red neuronal, el número de capas, las conexiones entre ellas, o la probabilidad límite de decisión si se trata de un problema de clasificación. A continuación, entrenar el modelo en el conjunto de entrenamiento con cada una de las combinaciones.

3. Calcular las métricas de evaluación correspondientes para cada combinación de hiperparámetros sobre el conjunto de validación. Escogemos el modelo correspondiente a la combinación que haya obtenido la mejor evaluación.

4. Evaluar el modelo escogido sobre el conjunto de *test*, para obtener la evaluación final.

Este proceso que acabamos de describir no es la única manera de utilizar el conjunto de validación para escoger un modelo. Por ejemplo, cuando se entrenan modelos de aprendizaje profundo (redes neuronales o *Graph Neural Networks*) a lo largo de varias épocas (ver Algoritmo 3.3.2), se suele utilizar el conjunto de validación para evaluar el modelo resultante después de cada una de las épocas, de manera que, cuando finaliza el entrenamiento, podamos escoger el modelo resultante de la época en la que hayamos obtenido la mejor evaluación.

Esto previene el posible *overfitting* resultante de entrenar durante demasiadas iteraciones el modelo. Esta técnica es la que emplearemos en la práctica para entrenar el modelo SEAL, que veremos más adelante en el Capítulo 6. Ahora que tenemos una idea general de cómo se evalúa un modelo, veamos algunas de las métricas más usadas, y en qué casos se utiliza cada una.

### 3.4.2. Métricas de evaluación

En el caso de problemas de clasificación, veremos primero métricas específicas para problemas de clasificación binarios, y después veremos cómo generalizan a problemas con más de dos clases.

**Definición 3.4.1.** Dado un conjunto de pares clase-predicción $\mathcal{P} = \{(c_1, v_1), \ldots, (c_n, v_n)\}$ (clase real de un dato, y predicción obtenida por un modelo), donde $c_i, v_i \in \{P, N\}$ (clase positiva y clase negativa), definimos:

- El número de verdaderos positivos, $TP$ (*True Positives*) como el número de pares tales que $c_i = v_i = P$.
- El número de falsos positivos, $FP$ (*False Positives*) como el número de pares tales que $c_i = N$ y $v_i = P$.
- El número de verdaderos negativos, $TN$ (*True Negatives*) como el número de pares tales que $c_i = v_i = N$.
- El número de falsos negativos, $FN$ (*False Negatives*) como el número de pares tales que $c_i = P$ y $v_i = N$.

Estos valores se suelen representar en una matriz de dimensión $2 \times 2$ llamada **matriz de confusión**, donde cada fila y cada columna se asocia con cada una de las clases reales y predichas respectivamente, obteniendo así los 4 valores en cada una de las entradas de la matriz (Figura 3.1).

Los valores de la definición anterior son las piezas con las que se definen las métricas que vemos a continuación.

**Definición 3.4.2.** Dado un conjunto de pares clase-predicción $\mathcal{P} = \{(c_1, v_1), ..., (c_n, v_n)\}$, donde las predicciones han sido obtenidas mediante un modelo, se definen:

- La *accuracy*[3] del modelo como

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}.$$

[3]Utilizamos el término en ingles debido a la falta de un término distinto a "Precisión" en castellano que nos permita distinguir *accuracy* y precisión

Figura 3.1: Matriz de Confusión. Fuente: *Towards Data Science*.

- La precisión del modelo como

$$\text{Precisión} = \frac{TP}{TP + FP}.$$

- La exhaustividad (o *recall*) del modelo como

$$\text{Exhaustividad} = \frac{TP}{TP + FN}.$$

Es decir, la *accuracy* mide el porcentaje de predicciones acertadas respecto del total, la precisión mide el porcentaje de verdaderos positivos respecto de los datos para los que se ha predicho la clase positiva, y la exhaustividad mide el porcentaje de verdaderos positivos respecto de los datos cuya clase real es la clase positiva.

La *acurracy* es una métrica que sirve para conjuntos de datos balanceados, pero con la que hay que tener cuidado cuando se utiliza para evaluar un modelo cuando el conjunto de datos está desbalanceado. Por ejemplo, si queremos un modelo que debe detectar una enfermedad que sufre el 1 % de la población, y utilizamos un modelo que realice siempre una predicción negativa, la *acurracy* del modelo será de un 99 %, ya que acertará en todos los casos en los que la persona no padezca la enfermedad, cuando lógicamente, el modelo es inútil ya que nunca detectará a las personas que sí padecen la enfermedad. Es en estos casos en los que la precisión y la exhaustividad se vuelven más útiles para medir el rendimiento del modelo. En el caso anterior, tendríamos una exhaustividad de 0, lo que indicaría que el modelo nunca detecta los verdaderos positivos.

Intuitivamente, la precisión mide la capacidad del modelo de acertar con las predicciones positivas, mientras que la exhaustividad mide la capacidad del modelo de detectar los verdaderos positivos. Es por ello que dependiendo del

problema en cuestión, nos interese priorizar una métrica u otra, ya que existe un *trade-off* entre ambas. Esto se debe a que, en la práctica, los modelos que resuelven problemas de clasificación predicen la probabilidad de que la muestra pertenezca a la clase positiva, y se establece un límite de probabilidad (valor entre 0 y 1) de manera que, si la probabilidad se encuentra por encima del límite, la predicción final es la clase positiva, mientras que si la probabilidad se encuentra por debajo del límite, la predicción es la clase negativa. Por esta razón, representar en una gráfica bidimensional cómo varían la exhaustividad y la precisión en función del límite de probabilidad es una técnica comúnmente usada para escoger el valor del límite (que es un hiperparámetro del modelo) para encontrar un equilibrio entre ambas (Figura 3.2)

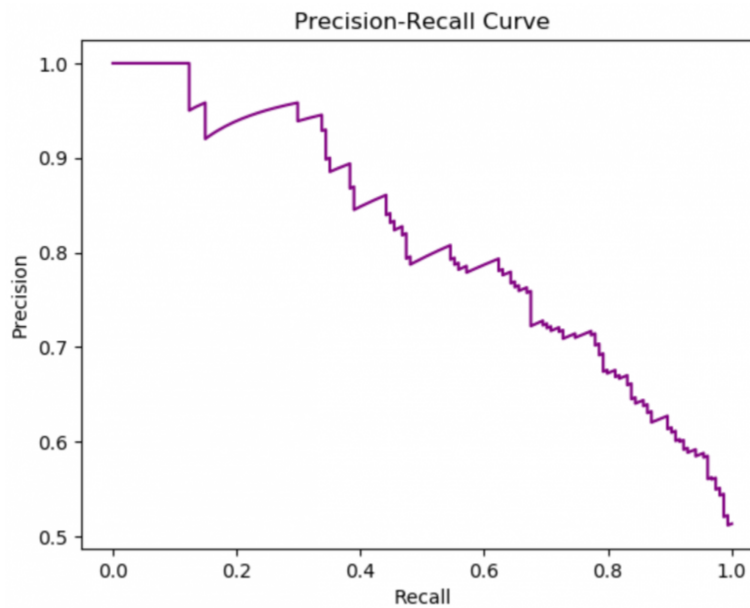


Figura 3.2: Gráfica de precisión vs exhaustividad (*precision vs recall curve*), en función del límite de probabilidad (Fuente).

El ejemplo de la enfermedad comentado anteriormente sería un caso en el que tendría más sentido darle más importancia a la exhaustividad, ya que es más importante detectar la mayoría de casos de la enfermedad, que equivocarse con falsos positivos en algunos. Sin embargo, si queremos clasificar contenido en una red social como una posible recomendación para un usuario, nos interesa más tener una buena precisión, ya que queremos que las recomendaciones sean acertadas, aunque se omitan muchas posibles recomendaciones. En todo caso, por regla general siempre se suele buscar un equilibrio entre las dos métricas. Es por ello que la siguiente métrica, que resulta de una combinación de ambas, también se usa con frecuencia.

**Definición 3.4.3.** Definimos el **Valor-F** como

$$F_\beta = (1+\beta^2)\frac{\text{Precisión} \cdot \text{Exhaustividad}}{(\beta^2 \cdot \text{Precisión}) + \text{Exhaustividad}},$$

donde $\beta$ es un número tal que, si es mayor que uno, se le da más ponderación a la exhaustividad, y si es menor que uno a la precisión.

En la práctica, se suelen escoger valores como $\beta = 2$ si se quiere priorizar la exhaustividad, o $\beta = 0.5$ si se quiere priorizar la precisión. También se usa con frecuencia $\beta = 1$, que mantiene un equilibrio entre ambas, métrica que es conocida como el valor F1:

$$F_1 = 2\frac{\text{Precisión} \cdot \text{Exhaustividad}}{\text{Precisión} + \text{Exhaustividad}}.$$

Otra métrica conocida para problemas de clasificación binarios es el $AUC$, o *Area Under Curve*, que definimos a continuación:

**Definición 3.4.4.** Dado un problema de clasificación binario y un modelo que resuelva el mismo, se define la tasa de falsos positivos (*FPR, False Positive Rate*) como:

$$FPR = \frac{FP}{FP + TN}.$$

Es decir, una métrica análoga a la exhaustividad, pero fijándonos en los falsos positivos, en lugar de los verdaderos positivos. Naturalmente, la exhaustividad es por lo tanto conocida también como la tasa de verdaderos positivos (TPR o *True Positive Rate*).

**Definición 3.4.5.** Definimos la curva ROC (*Receiver Operating Characteristic*) como la gráfica bidimensional de la tasa de falsos positivos vs la tasa de verdaderos positivos en función del límite de probabilidad del modelo (Figura 3.3). Se define la métrica AUC (*Area under curve*) como la integral definida entre 0 y 1 de la curva ROC.

Por lo tanto, es una curva idéntica a la curva precisión vs exhaustividad, pero sustituyendo la precisión por la tasa de verdaderos negativos. Entonces, es posible preguntarse en qué casos se usa una u otra para medir el rendimiento de un modelo. En este trabajo nos ceñiremos al uso de la curva ROC y la métrica AUC para evaluar los experimentos del clasificador SEAL (Capítulo 6), por lo que no entraremos en profundidad en el tema. En general, se utiliza la curva precisión vs exhaustividad cuando el porcentaje de muestras positivas es un dato relevante en el contexto del problema, pero para una discusión más en profundidad, se puede consultar [3].

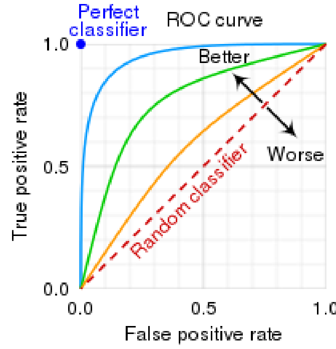


Figura 3.3: Algunas curvas ROC, y una interpretación de las mismas. En la esquina superior izquierda, vemos cómo un clasificador ideal (perfecto) tendría un 0 % de FPR y un 100 % de TPR, independientemente del límite de probabilidad (la gráfica sería el punto $(0, 1)$). (Fuente).

Para el caso de problemas de clasificación en múltiples clases[4], tenemos una serie de definiciones análogas. Para dar una definición análoga a la Definición 3.4.1, debemos ver el problema como uno de clasificación binaria para cada clase, donde la pertenencia a la clase sería un positivo, y la no pertenencia un negativo. De esta manera, podemos contar los verdaderos positivos, falsos positivos, etc. fijándonos en cada clase, y a continuación sumarlos. Así, obtenemos los valores agregados $TP_{sum}$, $TN_{sum}$, $FP_{sum}$ y $FN_{sum}$, con los que podemos establecer las siguientes definiciones.

**Definición 3.4.6.** Dado un problema de clasificación en múltiples clases y un modelo que lo resuelve, definimos:

- La micro-exhaustividad (*micro-recall*) del modelo como

$$\text{Micro-Exhaustividad} = \frac{TP_{sum}}{TP_{sum} + FN_{sum}}.$$

- La micro-precisión (*micro-precision*) del modelo como

$$\text{Micro-Precisión} = \frac{TP_{sum}}{TP_{sum} + FP_{sum}}.$$

- El micro-F-valor (micro-F1 si $\beta = 1$) como:

$$\text{micro-F} = (1 + \beta^2)\frac{\text{Micro-Precisión} \cdot \text{Micro-Exhaustividad}}{(\beta^2 \cdot \text{Micro-Precisión}) + \text{Micro-Exhaustividad}},$$

  donde $\beta \in \mathbb{R}$.

[4]Hay más de dos clases, y un dato puede ser clasificado como perteneciente a varias a la vez.

Utilizaremos la métrica micro-F1 para evaluar el rendimiento de los distintos modelos de GNN a los que aplicaremos GAug-M en el Capítulo 7.

# Capítulo 4

# Redes Neuronales

En este capítulo, adaptaremos lo que conocemos sobre aprendizaje supervisado a un tipo de modelo en concreto, las redes neuronales, en las que se basará la arquitectura de una *Graph Neural Network*. Primero, dedicaremos una sección a definir las neuronas, las piezas que forman una red neuronal para, a continuación, hablar de redes neuronales, los distintos tipos de capas que pueden presentar, y su optimización.

## 4.1. Componentes de una red neuronal

Para entender una red neuronal, debemos conocer las piezas que la conforman que, naturalmente, son las neuronas. En adelante, denotaremos como $\mathbb{B}$ al alfabeto binario $\{0, 1\}$. Explicaremos el concepto básico de perceptrón, para luego hablar de neuronas y funciones de activación.

### 4.1.1. Perceptrones

La idea inicial sobre la que se construye una neurona es la de perceptrón, que definimos a continuación.

**Definición 4.1.1.** Un **perceptrón** es una función $f$ de $\mathbb{B}^n$ en $\mathbb{B}$, de manera que

$$f(x_1, ..., x_n) = \begin{cases} 1, & \text{si } \sum_{i=1}^{n} w_i x_i > c \\ 0, & \text{si } \sum_{i=1}^{n} w_i x_i \leq c \end{cases}$$

donde los $w_i$ son números reales que llamamos pesos, y $c$ es otro número real, comúnmente llamado *bias*.

Intuitivamente, un perceptrón es una máquina que, en función de una serie de valores binarios (que pueden representar, por ejemplo, si se cumplen determinadas condiciones o no) toma una decisión (1 o 0). Nosotros definimos los pesos $w_i$ y el *bias* $c$, de manera que:

- Controlamos si una condición influye positiva o negativamente y en qué medida a la hora de decidir, mediante los pesos $w_i$.
- Controlamos el umbral con el que el perceptrón responde afirmativa o negativamente, aumentando o disminuyendo $c$.

Por ejemplo, supongamos que queremos diseñar un perceptrón que responda si va a llover o no mañana, en función de la predicción de AEMET[1], $x_1$, y la opinión de dos amigos, $x_2$ y $x_3$ (Figura 4.1). Supongamos que nos fiamos más de la predicción de AEMET, y a continuación de la del primer amigo. Una manera de diseñar el perceptrón sería: $w_1 = 5$, $w_2 = 2$, $w_3 = 1$ y $c = 4$. En este caso, podemos observar que el perceptrón predice que va a llover si y solo si $x_1 = 1$. Es decir, nuestro perceptrón ignora la opinión de nuestros amigos y se fía únicamente de la predicción de AEMET.

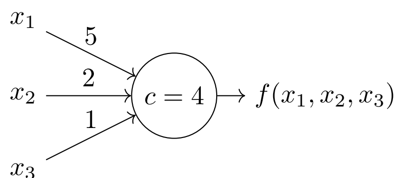


Figura 4.1: Un ejemplo de perceptrón

Sin embargo, podríamos modificarlo, poniendo $c = 6$. Ahora, no basta con $x_1$ para responder que sí. El perceptrón responde "sí" si AEMET y el primer amigo dicen que sí, sin embargo, no le bastaría con el "sí" de AEMET y el segundo amigo. Por lo tanto, en función de la situación o de cómo queremos que se comporte el perceptrón, podemos modificar sus pesos y su límite para cambiar su decisión. Veremos más adelante que la misma idea de modificar los pesos para cambiar la decisión es la clave del aprendizaje de una red neuronal.

Otra manera de escribir la definición de $f$ (que usaremos de ahora en adelante) es usando lo que llamamos el sesgo, que es sencillamente $b = -c$. Además, podemos utilizar la notación vectorial $\sum_{i=1}^{n} w_i x_i = w^T x$ lo que nos permite escribir

[1] Agencia Estatal de Meteorología `https://www.aemet.es/es/`

$f$ como:

$$f(x_1, ..., x_n) = \begin{cases} 1, & \text{si } w^T x + b > 0 \\ 0, & \text{si } w^T x + b \leq 0 \end{cases}$$

que es más compacto y fácil de manejar.

### 4.1.2. Neuronas y funciones de activación

Sin embargo, para poder aplicar el algoritmo de descenso del gradiente buscamos que, mediante pequeños cambios en los pesos y en el sesgo, modifiquemos ligeramente la respuesta de los perceptrones hasta que den la respuesta buscada para ciertos datos de entrada. Pero pequeños cambios en $w_i$ y en $b$ con la definición que hemos dado solo pueden provocar que el perceptrón cambie bruscamente su respuesta de 0 a 1 o viceversa, o bien que el perceptrón responda lo mismo que antes del cambio, con lo cual no podemos saber de qué forma hay que cambiar los $w_i$ y los $b$ de manera que haya un aprendizaje.

Por ello, necesitamos hacer uso de una cierta función continua creciente $\sigma$, que llamamos función de activación, de manera que $\sigma\colon \mathbb{R} \to (0,1)$ y tal que

$$\lim_{z \to \infty} \sigma(z) = 1, \quad \lim_{z \to -\infty} \sigma(z) = 0.$$

De esta manera, si $z = w^T x + b$, la función simula el comportamiento del perceptrón $\theta$, pero además podemos ver el resultado como una probabilidad entre 0 y 1, y pequeños cambios a los parámetros provocarán pequeños cambios al valor de probabilidad $\sigma(z)$. Así, podremos acercar el perceptrón al valor deseado mediante pequeños cambios en los parámetros. En conclusión, modificaremos ligeramente la definición de perceptrón incluyendo esta idea.

**Definición 4.1.2.** Una **neurona**, es una función continua $f\colon [0,1]^n \to (0,1)$, de manera que

$$f(x_1, ..., x_n) = \sigma(z), \ z = w^T x + b,$$

donde $\sigma$ es una función de activación.

Es decir, una neurona puede recibir valores reales entre 0 y 1, lo cual es una ventaja para problemas como, por ejemplo, reconocimiento de imágenes, ya que los datos pueden representar la intensidad de cierto color en cada píxel. En el caso del ejemplo anterior sobre predecir el clima, obtendríamos la probabilidad de que mañana llueva o no, en lugar de un simple "sí" o "no". Sin embargo, también pueden usarse para resolver problemas de decisión al igual que un perceptrón. Para ello, consideramos un límite de probabilidad, a partir del cual damos una respuesta positiva. Por ejemplo, podríamos definir que la respuesta

es "sí" si la neurona responde un valor de probabilidad superior a 0.5, o "no" si es menor que 0.5.

Existen varias funciones de activación conocidas y muy usadas, sin embargo, el ejemplo que damos a continuación es la que usaremos con más frecuencia en el trabajo.

**Ejemplo 4.1.3.** La **función logística** se define como:

$$\sigma(z) = \frac{1}{1+e^{-z}}, \ z \in \mathbb{R} \tag{4.1}$$

Es una función de activación, ya que $\sigma(z) \in (0,1)$, debido a que $1 + \frac{1}{e^z} > 1$ $\forall z \in \mathbb{R}$, y además es fácil comprobar que es creciente y que cumple

$$\lim_{z\to\infty} \sigma(z) = 1, \ \lim_{z\to-\infty} \sigma(z) = 0.$$

La función logística es ideal para representar el resultado de un perceptrón como una probabilidad, ya que se tiene $\sigma(0) = 0.5$, y además, crece muy rápidamente, tanto hacia 0 para valores negativos como hacia 1 para valores positivos (Figura 4.2)

Otros ejemplos importantes de funciones de activación son:

**Ejemplo 4.1.4.** La función ReLu (*Rectified Linear Unit).*

$$\sigma(z) = \max(0, z) = \begin{cases} 0, & \text{si } z < 0 \\ z, & \text{si } z \geq 0 \end{cases}$$

Es utilizada comúnmente en redes neuronales para que ciertas neuronas no se activen (devuelvan 0 como respuesta), lo que consigue reducir los cálculos y hacer que neuronas que no sean relevantes para la decisión no sean tenidas en cuenta.

**Ejemplo 4.1.5.** La función *SoftMax*. A diferencia de las anteriores, recibe como entrada un vector de datos.

$$\sigma\colon \mathbb{R}^n \longrightarrow [0,1]^n$$
$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{n} e^{z_k}}, \ j = 1, ..., n.$$

La función *SoftMax* es comúnmente usada en problemas de clasificación en los que se usan redes neuronales, ya que si se escoge como función de activación de la última capa, se obtiene como resultado una distribución de probabilidades de pertenecer a cada una de las clases.

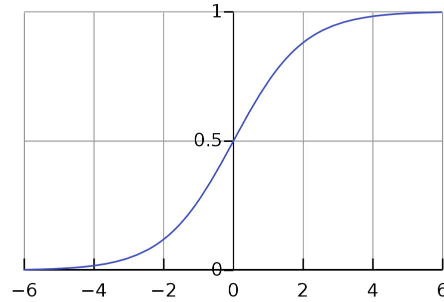


Figura 4.2: Gráfica de la función logística para valores cercanos a 0.

## 4.2. El modelo de aprendizaje de redes neuronales

Ahora que conocemos el funcionamiento de las neuronas, podemos hablar de redes neuronales. Comentaremos algunos tipos de capas comúnmente usadas en redes neuronales, y también explicaremos el algoritmo de propagación hacia atrás para poder optimizarlas mediante descenso del gradiente.

### 4.2.1. Redes Neuronales

Las redes neuronales son uno de los modelos más importantes en *Machine Learning*. Conforman su propia rama, llamada *deep learning*, y son ampliamente usadas en diversos problemas, como por ejemplo, reconocimiento de imágenes, como identificar dígitos escritos a mano [18], entre otros. Además de ver su definición, veremos algunos ejemplos comúnmente usados de tipos de capas de redes neuronales, diseñadas para tareas específicas.

**Definición 4.2.1.** Una **red neuronal** es un modelo de aprendizaje supervisado que está formado por $l$ conjuntos ordenados de neuronas, llamados capas, de manera que:

1. La primera capa, o capa de entrada, recibe valores de entrada $x_1, ..., x_n$ para cada neurona.
2. Las capas intermedias, o capas ocultas, reciben valores arrojados por las neuronas de capas anteriores, pero nunca de neuronas de capas posteriores, es decir, no existen conexiones hacia atrás. Es por ello que también se suelen llamar *feedforward neural networks*, en contraposición a las *recurrent neural networks*, que sí pueden presentar este tipo de conexiones.

La respuesta de una neurona puede ser valor de entrada de más de una neurona en las capas siguientes.

3. La respuesta de la red neuronal es el conjunto de respuestas de las neuronas de la última capa, o capa de salida.

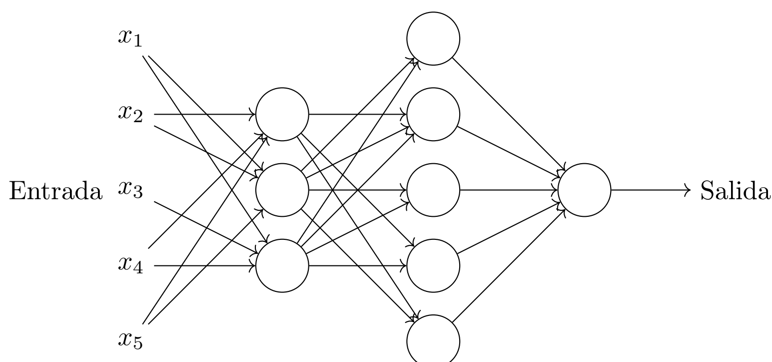


Figura 4.3: Un ejemplo de una red neuronal, en este caso con una capa oculta, 5 valores de entrada y uno de salida. Cada nodo es una neurona con distinto número de entradas, con sus correspondientes pesos y sesgos.

Un detalle a tener en cuenta es que el número de entradas y de neuronas de salida depende del problema en cuestión. Por ejemplo, si tenemos un problema de clasificación con $k$ categorías, y queremos obtener la probabilidad de pertenecer a cada categoría, necesitaremos tener $k$ neuronas en la capa de salida (con una activación *SoftMax* a continuación), mientras que el número de entradas siempre será el número de variables que tendremos en cuenta para hacer la predicción. En contraposición, el número de capas ocultas de la red neuronal y el número de neuronas en cada una de las capas que no sean la salida son, esencialmente, hiperparámetros de nuestro modelo, al igual que el *learning rate*. Esto quiere decir que podríamos comparar distintas arquitecturas de redes neuronales utilizando el conjunto de *test*, para luego medir el rendimiento de la que escojamos utilizando el conjunto de validación.

Veamos algunos ejemplos de tipos de capas de redes neuronales comúnmente usadas.

**Ejemplo 4.2.2.** Decimos que una capa de red neuronal es **completamente conexa** si todos los *outputs* de la capa anterior son *inputs* de todas las neuronas de la capa. Es decir, tendríamos todas las conexiones posibles entre neuronas de la capa anterior y la capa completamente conexa, como en la Figura 4.4

Los siguientes ejemplos están inspirados en las *convolutional neural networks*, pero en este caso, trabajan en una dimensión.

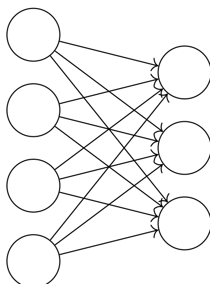

Figura 4.4: Ejemplo de una capa completamente conexa con 3 neuronas.

**Ejemplo 4.2.3.** Una **capa de convolución 1-D**, con tamaño de kernel $k$ y *step* $s$, es una capa de red neuronal cuyas conexiones se definen de la siguiente manera: si el *output* de la capa anterior tiene longitud $n$, de manera que $n = k + ms$ para algún $m \in \mathbb{N}$: colocamos una neurona que recibe las coordenadas $1, ..., k$ del vector, a continuación, la segunda neurona, que recibe las coordenadas $1 + s, ..., k + s$ del vector, etc. Así, hasta la última neurona, que recibe las coordenadas $n - s, ..., n$ del vector. Intuitivamente, podemos imaginar un filtro de tamaño $k$ sobre las coordenadas del vector, del cual tomamos los inputs de cada neurona que se coloca. Una vez colocada la neurona, movemos el filtro $s$ posiciones a la derecha, y colocamos otra, así hasta que no podamos mover más el filtro a la derecha. La condición $n = k + ms$ se impone para que el último filtro coincida con las $k$ coordenadas finales del vector. El esquema de la Figura 4.5 hace más facil la comprensión de una capa de convolución 1-D.

**Ejemplo 4.2.4.** Una capa **MaxPooling** funciona exactamente igual que una capa de convolución 1-D, salvo que, para cada posición del filtro, en lugar de definir una neurona, simplemente tomamos el máximo de las coordenadas del kernel como *output* de esa posición del kernel. Supongamos que, en un situación igual que la de la Figura 4.5, el vector *input* es $v = (2, 5, 6, 5, 9, 3, 9, 0)$. Entonces, una capa *MaxPooling* con la misma arquitectura devolvería el vector $u = (6, 6, 9, 9, 9, 9)$, que son los valores máximos para cada posición del filtro.

De la misma manera que podemos ver una neurona como una función no solo de los valores de entrada, sino también de los pesos;

$$f(x, w, b) = \sigma\big(\sum_{i=1}^{n} w^T x + b\big)$$

podemos ver una red neuronal como una función de la misma manera, que es a su vez composición de $l$ funciones (una por cada capa), introduciendo la siguiente notación:

$$y = \big(f_\Theta(x) = f_{\theta_l}^{(l)} \circ f_{\theta_{l-1}}^{(l-1)} \circ \cdots \circ f_{\theta_2}^{(2)} \circ f_{\theta_1}^{(1)}\big)(x).$$

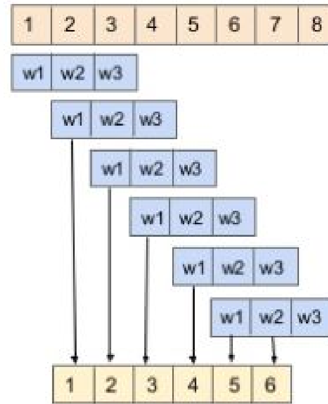


Figura 4.5: Una capa de convolución que recibe un vector de longitud $n = 8$, con tamaño de kernel $k = 3$ y step $s = 1$. Se pueden observar todas las posibles posiciones del kernel, y los pesos $w_1, w_2, w_3$ de las neuronas que cada posición va definiendo. Los pesos pueden escogerse tanto distintos como iguales para cada neurona (Fuente).

Donde cada $\theta_i$ es el conjunto de parámetros de la capa $i$ (vectores de pesos y sesgos de cada neurona de la capa), y $\Theta$ es el conjunto de todos los $\theta_1, ..., \theta_l$. De esta manera, conseguimos una notación compacta que casa con las generalidades sobre aprendizaje supervisado que vimos en la sección anterior. Por lo tanto, estamos preparados para aplicar el algoritmo de descenso del gradiente en el caso particular de redes neuronales.

### 4.2.2. Descenso del gradiente en redes neuronales y propagación hacia atrás

Indaguemos ahora en la optimización de redes neuronales, en concreto, en la aplicación del descenso del gradiente a las mismas. Esto se consigue mediante el algoritmo de propagación hacia atrás, el cual explicamos en esta subsección.

Para formular el descenso del gradiente, separaremos los parámetros de toda la red neuronal $\Theta$ en pesos de todas las neuronas $(w_k)$ por un lado, y en sesgos $(b_l)$ por otro. De esta manera, actualizaríamos los parámetros de una red neuronal en cada paso del descenso del gradiente como sigue:

$$w_k \to w'_k = w_k - \mu \frac{\partial \mathcal{L}}{\partial w_k},$$
$$b_l \to b'_l = b_l - \mu \frac{\partial \mathcal{L}}{\partial b_l},$$

ya que el gradiente en este caso sería

$$\nabla \mathcal{L} = (\frac{\partial \mathcal{L}}{\partial w_1}, ..., \frac{\partial \mathcal{L}}{\partial w_m}, \frac{\partial \mathcal{L}}{\partial b_1}, ..., \frac{\partial \mathcal{L}}{\partial b_n})$$

si tenemos $m$ pesos y $n$ sesgos en total. Pero, ¿cómo calculamos el gradiente $\nabla \mathcal{L}$? Es aquí donde entra en juego el algoritmo de propagación hacia atrás.

Lo primero que tenemos que tener en cuenta es que el algoritmo de propagación hacia atrás calcula el error asociado a cada muestra de entrenamiento, lo que permite calcular el gradiente $\nabla \mathcal{L}$, ya que $\nabla \mathcal{L} = \sum_{x \in D} \nabla \mathcal{L}_x$ pero, para no sobrecargar la notación, en lo que sigue escribiremos $\mathcal{L}$ para referirnos al error asociado únicamente a una muestra del conjunto de entrenamiento. Por otro lado, el algoritmo será mas fácil de formular y entender si utilizamos una notación que concrete la capa y la neurona en la que se encuentran los pesos y los sesgos de los que hablemos en todo momento, que explicamos a continuación.

1. $w^l_{jk}$ denota el peso asociado a la entrada en la neurona $j$ de la capa $l$, que proviene de la neurona $k$ en la capa $l-1$.
2. $b^l_j$ denota el sesgo de la neurona $j$ de la capa $l$.
3. Escribimos el *output* de la neurona $j$ de la capa $l$ (antes de la aplicación de la función de activación) como $z^l_j$.
4. Finalmente, usamos $a^l_j$ para referirnos a la activación de la neurona $j$ de la capa $l$.

Con esta notación, la relación entre la activación de una neurona en la capa $l$ y los *inputs* que provienen de la capa anterior se expresa como:

$$a^l_j = \sigma\left(z^l_j\right) \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right), \tag{4.2}$$

donde la suma es sobre cada neurona $k$ de la capa $l-1$. Podemos escribir esta expresión de manera matricial, para que sea más compacta. Para ello, definimos una matriz de pesos para cada capa $l$, $w^l$, cuyas entradas no son más que los pesos de las neuronas de la capa, de manera que el elemento en la posición $(j, k)$ sea $w^l_{jk}$. Es decir, que cada **fila** de la matriz corresponde a los pesos de cada

neurona de la capa. De manera similar, para cada capa $l$, denotamos el vector de sesgos como $b^l$, y el vector cuyas componentes son las activaciones $a_j^l$ como $a_l$. Así, reescribimos la Ecuación 4.2 como:

$$a^l = \sigma\left(z^l\right) = \sigma\left(w^l a^{l-1} + b^l\right), \tag{4.3}$$

donde la función de activación $\sigma$ se aplica componente a componente. De esta manera, evitamos la notación de la Ecuación 4.2, más confusa debido a los subíndices.

El algoritmo de propagación hacia atrás consiste en operaciones comunes de álgebra lineal en su mayoría, como la suma componente a componente, o el producto de matrices. Sin embargo, necesitamos conocer la siguiente operación, no tan común.

**Definición 4.2.5.** Sean $s$ y $t$ vectores de un cierto $\mathbb{R}^n$. Definimos el **producto de Hadamard** (también conocido como producto de Schur o producto componente a componente) como

$$(s \odot t)_i = s_i t_i,\ i = 1, ..., n.$$

Lo que queremos entender es cómo influye en la función coste cambiar cada peso y sesgo de la red. Pero al cambiar el peso o el sesgo de la neurona $j$ en la capa $l$, también estamos cambiando el *output* de la neurona, $z_j^l$. Este cambio, $\Delta z_j^l$, se propaga a lo largo de las capas siguientes, hasta que provoca un cambio en la función coste:

$$\frac{\partial \mathcal{L}}{\partial z_j^l} \Delta z_j^l.$$

Por lo tanto, cuando $\frac{\partial \mathcal{L}}{\partial z_j^l}$ es muy pequeño, el cambio producido en la función coste es muy bajo cuando realizamos un cambio $\Delta z_j^l$. Intuitivamente, esto quiere decir que es difícil hacer la neurona más eficiente de lo que ya es mediante cambios en los parámetros, lo que significa que, en cierto sentido, $\frac{\partial \mathcal{L}}{\partial z_j^l}$ mide el error cometido por la neurona respecto a su configuración óptima. Es por ello que en el algoritmo de propagación hacia atrás definimos el "error" de la neurona $j$ de la capa $l$ como

$$\delta_j^l = \frac{\partial \mathcal{L}}{\partial z_j^l}.$$

Como viene siendo habitual con nuestra notación, usaremos $\delta^l$ para el vector de errores de la capa $l$. El algoritmo de propagación hacia atrás nos permitirá calcular estos errores, y relacionarlos con las derivadas parciales de la función coste respecto de los parámetros, que son las que realmente nos interesan. Damos a continuación el resultado clave para el funcionamiento del algoritmo.

**Teorema 4.2.6** (Ecuaciones fundamentales del algoritmo de propagación hacia atrás, [18])**.** *Sea $L$ la capa de salida de una red neuronal, $l$ cualquiera de sus capas ocultas, $j$ y $k$ neuronas de las capas $l$ y $l-1$ respectivamente. Se cumplen las siguientes igualdades:*

1. *Propagación hacia atrás de errores.*

$$\delta^L = \nabla_{a^L}\mathcal{L} \odot \sigma'(z^L) \tag{4.4}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{4.5}$$

2. *Derivadas parciales respecto de cualquier parámetro de la red.*

$$\frac{\partial \mathcal{L}}{\partial b^l} = \delta^l \tag{4.6}$$

$$\frac{\partial \mathcal{L}}{\partial w^l_{jk}} = a_k^{l-1}\delta_j^l. \tag{4.7}$$

*Donde $\nabla_{a^L}\mathcal{L}$ es un vector cuyas componentes son las derivadas parciales de $\mathcal{L}$ respecto de cada una de las activaciones de la capa $L$, $a_i^L$, y donde $(w^{l+1})^T$ es la matriz de pesos de la capa $l+1$ traspuesta.*

Con estas ecuaciones, podemos calcular el gradiente de $\mathcal{L}$, ya que primero, podemos calcular las activaciones de la última capa, y en la práctica podremos calcular $\nabla_{a^L}\mathcal{L}$ ya que $\mathcal{L}$ se trata de una expresión sencilla en función de $a^L$, que es el output de toda la red neuronal. Por ejemplo, si $\mathcal{L}$ es la función de error cuadrático medio, tenemos $\nabla_{a^L}\mathcal{L} = \frac{2(a^L-y)}{N}$. Con esto y con la Ecuación 4.4, obtenemos los errores $\delta^L$ que, gracias a la Ecuación 4.5, nos permite calcular los errores de la capa anterior, que podemos volver a utilizar para calcular los de la siguiente, así hasta obtener todos los errores de todas las neuronas de la red. Finalmente, conociendo estos valores, podemos calcular todas las parciales gracias a las Ecuaciones 4.6 y 4.7.

*Demostración.* Demostremos cada una de las igualdades del Teorema 4.2.6:

- Ecuación 4.4: Por definición, $\delta_j^L = \frac{\partial \mathcal{L}}{\partial z_j^L}$ y, si aplicamos la regla de la cadena para varias variables para expresarlo en función de las activaciones de cada neurona $k$ de la capa $L$, obtenemos

$$\delta_j^L = \sum_k \frac{\partial \mathcal{L}}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L}.$$

Pero lógicamente, la activación de la neurona $k$, $a_k^L = \sigma(z_k^L)$, depende solo del output $z_k^L$, por lo que $\frac{\partial a_k^L}{\partial z_j^L} = 0$ si $j \neq k$. Por lo tanto, la ecuación

anterior es en realidad

$$\delta_j^L = \frac{\partial \mathcal{L}}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial \mathcal{L}}{\partial a_j^L} \sigma'(z_j^L),$$

que es la Ecuación 4.4 componente a componente.

- Ecuación 4.5: Usando otra vez la regla de la cadena con el error de la neurona $j$ de la capa $l$, pero esta vez con los *outputs* de la capa $l+1$, obtenemos

$$\delta_j^l = \frac{\partial \mathcal{L}}{\partial z_j^l} = \sum_k \frac{\partial \mathcal{L}}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}. \tag{4.8}$$

Teniendo en cuenta que $z_k^{l+1} = \sum_m w_{km}^{l+1} a_m^l + b_k^{l+1} = \sum_m w_{kj}^{l+1} \sigma(z_m^l) + b_k^{l+1}$ y derivando, obtenemos

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$$

y, sustituyendo en el último término de la Ecuación 4.8, obtenemos

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l),$$

que es la Ecuación 4.5 componente a componente.

- Ecuación 4.6: Por la regla de la cadena,

$$\frac{\partial \mathcal{L}}{\partial b_j^l} = \sum_k \frac{\partial \mathcal{L}}{\partial z_k^l} \frac{z_k^l}{\partial b_j^l}$$

y, dado que el output de la neurona $k$, $z_k^l$, depende de $b_j^l$ solo si $k = j$,

$$\frac{\partial \mathcal{L}}{\partial b_j^l} = \frac{\partial \mathcal{L}}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \frac{\partial z_j^l}{\partial b_j^l}.$$

Por otra parte,

$$z_j^l = \sum_k w_{jk} a_k^{l-1} + b_j^l \Rightarrow \frac{\partial z_j^l}{\partial b_j^l} = 1$$

y, por consiguiente,

$$\frac{\partial \mathcal{L}}{\partial b_j^l} = \delta_j^l.$$

- Ecuación 4.7: De manera similar a los razonamientos anteriores,

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^l} = \sum_n \frac{\partial \mathcal{L}}{\partial z_n^l} \frac{\partial z_n^l}{\partial w_{jk}^l} = \frac{\partial \mathcal{L}}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l \frac{\partial z_j^l}{\partial w_{jk}^l}.$$

Por otra parte, como $z_j^l = \sum_m w_{jm}^l a_m^{l-1} + b_j^l$, tenemos

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1}$$

y por lo tanto,

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}.$$

□

Finalmente, utilizando las ecuaciones fundamentales, podemos formular formalmente el algoritmo de propagación hacia atrás y calcular el gradiente $\nabla\mathcal{L}$.

**Algoritmo 4.2.1** (de propagación hacia atrás)**.** Dada una red neuronal $f$ con $L$ capas y con parámetros $\Theta$ ($\theta_l = (w_{jk}^l, b_j^l)_{j,k}$ para cada capa $l$) hacer, para cada $x \in T$,:

1. Calcular $a^L = f_\Theta(x)$ (por lo que también calculamos $z^l$ y $a^l$ para cada capa $l$) y $\nabla_{a^L}\mathcal{L}_x$.
2. Calcular $\delta^L = \nabla_{a^L}\mathcal{L}_x \odot \sigma'(z^L)$.
3. Para $l = L-1, L-2, ..., 2, 1$, calcular y almacenar:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l).$$

4. Calcular las parciales de $\mathcal{L}_x$ respecto de los parámetros de $\Theta$:

$$\frac{\partial \mathcal{L}_x}{\partial b^l} = \delta^l$$
$$\frac{\partial \mathcal{L}_x}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l.$$

Devolver $\nabla\mathcal{L} = \sum_{x\in T} \nabla\mathcal{L}_x$.

# Capítulo 5

# Graph Neural Networks

En este capítulo definiremos la estructura de una *Graph Neural Network* básica, y daremos variantes comúnmente usadas de la misma. Formularemos las *Graph Neural Networks* en términos de redes neuronales, lo que significa que lo visto en el capítulo anterior, en especial el algoritmo de propagación hacia atrás, es aplicable al entrenamiento de las *Graph Neural Networks*.

## 5.1. Definición del modelo de Graph Neural Networks

El concepto *Graph Neural Network* aparece propuesto por primera vez en 2009 por Scarselli et. al en [24], y tiene como objetivo definir una red neuronal que reciba como input un grafo $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, donde cada vértice tiene asignado un vector de *features*, de manera que nos permita resolver cierto problema de aprendizaje supervisado sobre el grafo. La gran mayoría de GNNs se diseñan para la resolución de uno de estos tres problemas:

- Clasificación de nodos. El objetivo es obtener un *embedding* $x_u \in \mathbb{R}^n$, para cada nodo $u \in \mathcal{V}$, y así clasificarlo en una categoría. Lo más común es aplicar la función SoftMax al embedding, para obtener un vector de probabilidades para cada clase, y utilizar la función de coste *cross-entropy*. Algunas aplicaciones son, por ejemplo, clasificar las funciones de cada proteína en un interactoma [9], o distinguir usuarios reales de bots en una red social (clasificación binaria) [8].
- Clasificación o regresión sobre el grafo, de la misma manera que en el caso anterior, computando un *embedding* (o valor) $x_{\mathcal{G}}$. Por ejemplo, podemos

tener un grafo que representa la estructura de una molécula, y queremos construir un modelo que realice una predicción de la toxicidad o solubilidad de la molécula (regresión) [6]. En el caso de clasificación, podríamos querer detectar si un programa es malicioso o no en base a un grafo que represente su sintaxis y su flujo de información (clasificación) [16].

- Predecir aristas entre nodos. Es bastante común obtener *embeddings* de parejas de nodos para predecir la existencia de una arista entre ellos, y existen funciones de coste específicas para este tipo de problemas ([8], Cap. 3 y 4). Entre otras aplicaciones, tenemos sistemas de recomendación, como en [31], donde se explica un sistema de recomendación en la red social *Pinterest*. El método SEAL [34], que explicaremos más adelante, clasifica un subgrafo del entorno de una pareja de nodos para predecir la probabilidad de que exista una arista entre ellos.

Las *Graph Neural Networks* surgen de la necesidad de un nuevo modelo de aprendizaje supervisado sobre datos estructurados como grafos, de manera que los resultados dependan de esta estructura. Esto es debido a que los modelos existentes en su momento, como por ejemplo, las *convolutional neural networks* (CNN) o las *recurrent neural networks* (RNN), pensadas para datos en forma de cuadrícula (como imágenes) y secuencial (como texto) respectivamente, no servían para tener en cuenta las relaciones entre datos en una estructura de grafo.

Una primera idea ingenua para abordar un problema de este estilo donde queremos clasificar un grafo $\mathcal{G}$ sería utilizar la matriz de adyacencia del grafo, $A$, como input de una red neuronal. Concatenaríamos las filas de la matriz, y entrenamos una red $f_\Theta$ que nos devuelva el *embedding* de $\mathcal{G}$:

$$z_{\mathcal{G}} = f_\Theta(A[1] \oplus A[2] \oplus \cdots \oplus A[|\mathcal{V}|])$$

Sin embargo, el problema de este método es que depende de la numeración/orden de los nodos del grafo. Por ello, buscamos modelos que sean, o bien invariantes respecto a permutaciones(Ecuación 5.1), o bien equivariantes respecto a permutaciones (Ecuación 5.2), si la permutación viene dada por una matriz $P$:

$$f_\Theta(PAP^T) = f_\Theta(A) \tag{5.1}$$

$$f_\Theta(PAP^T) = Pf_\Theta(A) \tag{5.2}$$

Una *Graph Neural Network* es un caso de modelo equivariante.

La idea en la que se basa una *Graph Neural Network* es la de transmisión de información entre nodos vecinos. Si partimos de un grafo $G = (\mathcal{E}, \mathcal{V})$ y de un

vector de *features* $z_u \in \mathbb{R}^n$ para cada nodo $u \in \mathcal{V}$, diseñamos una red neuronal que, en cada capa, actualiza el vector de un nodo $u$, $h_u^k \to h_u^{(k+1)}$ (comenzando por $h_u^0 = z_u$), en función de los vectores de los nodos conectados con $u$ (sus vecinos). Sin embargo, los vectores de estos nodos también se actualizan con la información de sus vecinos, por lo que, en la segunda capa, un nodo $u$ obtiene información de nodos a distancia 2 (vecinos de sus vecinos), y así hasta que lleguemos a la capa final. El número de capas depende de la arquitectura que escojamos, aunque en la mayoría de los problemas no es superior a 2.

Este procedimiento se puede escribir de manera general de la siguiente forma:

$$h_u^{(k+1)} = \text{UPDATE}^k(h_u^k, \text{AGGREGATE}^k(\{h_v^k, \forall v \in \mathcal{N}(u)\})) \tag{5.3}$$

$$= \text{UPDATE}^k(h_u^k, m_{\mathcal{N}(u)}^k), \tag{5.4}$$

donde $h_u^k$ es el vector correspondiente al nodo $u$ en la capa $k$ y $\text{UPDATE}^k$ es la función que se encarga de actualizar $h_u^k$ en base a la información de los vecinos de $u$, que es transmitida mediante una función **invariante** por permutaciones $\text{AGGREGATE}^k$, y cuyo resultado expresamos como $m_{\mathcal{N}(u)}^k$. A pesar de que AGGREGATE se trate de una función invariante por permutaciones (necesario para que el mensaje no cambie en función del orden de los vecinos), el cálculo final es equivariante por permutaciones, ya que una ordenación distinta de los nodos permuta el orden en el que aparecen los resultados, pero no el resultado final del cálculo del *embedding* para cada nodo en particular. Lo mas común es que $\text{UPDATE}^k$ sea calculada por una red neuronal, mientras que $\text{AGGREGATE}^k$ puede ser, o bien tan simple como sumar los vectores $h_v^k$ (como en el primer tipo de GNN que veremos), o bien otra red neuronal.

Solo con esto puede parecer difícil entender qué es realmente una GNN, porque parece una red neuronal, pero que a su vez tiene otras redes neuronales dentro, y no está claro si lo visto en el capítulo anterior aplica en este caso. En realidad, se trata de una red neuronal construida a partir de concatenar redes neuronales más pequeñas, correspondientes a cada nodo y a las funciones descritas en el párrafo anterior (si usamos los *outputs* de una red neuronal como *inputs* de otra red neuronal, lo que estamos haciendo es construir una red neuronal más grande). Para aclararlo, en la Sección 5.2, veremos un ejemplo de GNN básica, y justificaremos que el algoritmo de propagación hacia atrás sirve para optimizar una GNN al igual que hacíamos con una red neuronal.

Esta descripción abstracta de una GNN está bien, pero a la hora de implementarlas, necesitamos dar la forma que tienen las funciones $\text{UPDATE}^k$ y $\text{AGGREGATE}^k$, lo que da lugar a diversas arquitecturas de GNN (que explicaremos en las Secciones 5.3 y 5.4).

## 5.2. Graph Neural Network básica

Definimos la actualización del estado o vector de un nodo en una GNN básica [24] como:

$$h_u^k = \sigma\left(W_{\text{SELF}}^k h_u^{(k-1)} + W_{\text{NEIGH}}^k \sum_{v\in\mathcal{N}(u)} h_v^{(k-1)} + b^k\right), \tag{5.5}$$

donde $W_{\text{SELF}}^k$ y $W_{\text{NEIGH}}^k \in R^{d^{(k-1)}\times d^k}$ (donde $d^{(k-1)}$ es la dimensión del vector recibido de la capa $k$, y $d^k$ es la dimensión del vector que devuelve la capa $k$) son matrices de pesos correspondientes al vector del nodo en la capa anterior y al mensaje de los vecinos de $u$ respectivamente, y $\sigma$ es una función de activación aplicada componente a componente. Alternativamente, expresado en forma matricial tenemos

$$H^k = \sigma\left(AH^{(k-1)}W_{\text{NEIGH}}^k + H^{(k-1)}W_{\text{SELF}} + B^k\right), \tag{5.6}$$

donde $H^k, H^{k-1} \in \mathbb{R}^{|\mathcal{V}|\times d}$ es la matriz cuyas filas son los vectores asociados a cada nodo en una capa, $A$ es la matriz de adyacencia del grafo, y $B^k$ es una matriz cuyas filas son los sesgos asociados a cada uno de los nodos en una capa.

Con esta arquitectura, una capa de la GNN es equivalente a una capa de red neuronal (esto no siempre ocurre si UPDATE y AGGREGATE son a su vez redes neuronales), donde los pesos de cada neurona son las filas de las matrices $W_{\text{SELF}}^k$ y $W_{\text{NEIGH}}^k$, y los sesgos las componentes del vector $b^k$. Aunque especificamos la capa $k$, podríamos escoger utilizar los mismos parámetros para cada capa, o incluso si utilizar los mismos o no para las actualizaciones de cada nodo. Si lo formulamos en términos de las funciones UPDATE y AGGREGATE, tendríamos

$$m_{\mathcal{N}(u)}^{(k-1)} = \sum_{v\in\mathcal{N}(u)} h_v^{(k-1)},$$
$$\text{UPDATE}(h_u^{(k-1)}, m_{\mathcal{N}(u)}^{(k-1)}) = \sigma(W_{\text{SELF}}^k h_u^{(k-1)} + W_{\text{NEIGH}}^k m_{\mathcal{N}(u)}^{(k-1)}).$$

**Ejemplo 5.2.1.** Consideremos el grafo $G = (\mathcal{V}, \mathcal{E})$, representado en la figura 5.1, donde para cada nodo $u \in \mathcal{V}$, tenemos un vector de features inicial $z_u \in \mathbb{R}^3$.

Sobre este grafo, construimos una GNN básica con 2 capas que daría lugar a la red neuronal de la Figura 5.2, de la cual mostramos solo las conexiones y neuronas que actualizan el nodo $a$ para visualizar bien qué ocurre. En la red habría realmente 3 estructuras similares adicionales, que actualizan los vectores de cada nodo en cada capa, de donde provienen los vectores de $d$ y $b$ para calcular AGGREGATE$^1(\{h_d^1, h_b^1\})$. La red se inicializa con $h_u^0 = z_u$ $\forall u \in \mathcal{V}$.

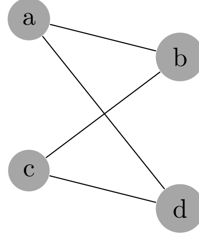


Figura 5.1: Grafo $G = (\mathcal{V}, \mathcal{E})$

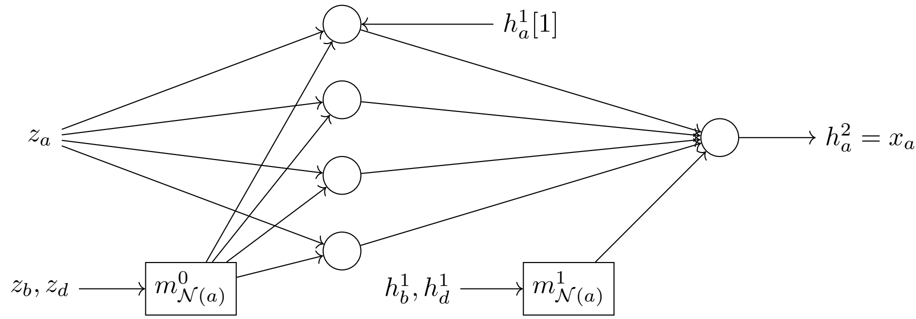


Figura 5.2: GNN básica de 2 capas sobre el grafo de la Figura 5.1. En esta GNN, tendríamos $W^1_{\text{SELF}}, W^1_{\text{NEIGH}} \in \mathbb{R}^{4\times 3}$, $b^1 \in \mathbb{R}^4$, y $W^2_{\text{SELF}}, W^2_{\text{NEIGH}} \in \mathbb{R}^{1\times 4}$, $b^2 \in \mathbb{R}$. El vector $h_a^1 \in \mathbb{R}^4$, cuyas coordenadas provienen de cada una de las 4 neuronas de la primera capa, $h_a^1[i] = \sigma\left(W^1_{\text{SELF}}[i] z_a + W^1_{\text{NEIGH}}[i] m^0_{\mathcal{N}(a)} + b^1_i\right)$, es el *input* de la última neurona, que devuelve el *output* $h_a^2 = \sigma\left(W^2_{\text{SELF}} h_a^1 + W_{\text{NEIGH}} m^1_{\mathcal{N}(a)} + b^2_1\right)$.

Podemos observar que, aunque $z_a$ tiene tres *features*, la primera capa devuelve un vector oculto $h_a^1$ de cuatro coordenadas (al igual que puede ocurrir con las capas ocultas de redes neuronales). La elección del número de neuronas (cuatro en este caso) no tiene relación con la estructura del grafo, y es por ello que es un hiperparámetro del modelo. En este caso, la capa final tiene una neurona, por lo que esta red podría estar diseñada para, por ejemplo, un problema de clasificación binario, donde el *embedding* $x_a$ sería la probabilidad de que el nodo pertenezca a la clase positiva. En la figura también se observa con claridad cómo una capa de la GNN coincide con una capa usual de red neuronal, con la salvedad de que realizamos las operaciones $m^0_{\mathcal{N}(a)}$ y $m^1_{\mathcal{N}(a)}$. Sin embargo, esto último no afecta a la hora de aplicar el algoritmo de propagación hacia atrás, ya que lo que importa es que se cumplan las ecuaciones (que ocurre siempre que la información fluya hacia adelante) y que podamos calcular las activaciones de cada capa.

En contraste con lo anterior, una GNN genérica de dos capas de acuerdo con la Ecuación 5.3, vendría representada por la Figura 5.3, donde $\Theta_i$ y $\Phi_i, i = 1, 2$

son los parámetros de los que dependen las funciones/redes neuronales arbitrarias UPDATE y AGGREGATE.

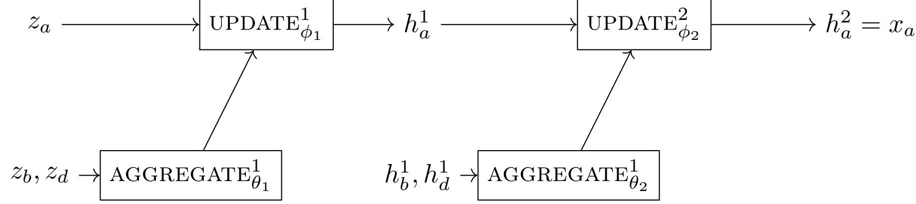


Figura 5.3: GNN genérica sobre el grafo de la figura 5.1

Una manera alternativa de plantear una GNN es omitir la función UPDATE, y en su lugar, añadir bucles en todos los nodos del grafo. De esta manera, incluimos la información del nodo en la capa anterior en la función AGGREGATE junto con la de sus vecinos, y ahorramos la definición de UPDATE.

$$h_u^k = \text{AGGREGATE}\left(\{h_v^{(k-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}\right). \tag{5.7}$$

En el caso de una GNN básica, esto equivale a que las matrices $W_{\text{NEIGH}}$ y $W_{\text{SELF}}$ sean iguales, dando lugar a la siguiente ecuación matricial

$$h_u^k = \sigma\left((A+I)H^{(k-1)}W^{(k)}\right). \tag{5.8}$$

## 5.3. Normalización del operador aggregate: Graph Convolutional Network

Un problema que surge de utilizar el operador AGGREGATE de una GNN básica (i.e., simplemente sumar) es que puede crear grandes diferencias en las magnitudes de las *features* si un nodo tiene un grado bastante alto. En efecto, supongamos que un nodo $u \in \mathcal{V}$ tiene muchos más vecinos que otro nodo $u' \in \mathcal{V}$ (por ejemplo, $\delta(u) > 100\delta(u')$). Entonces, es esperable que $||\sum_{v\in\mathcal{N}(u)} h_v^k|| >> ||\sum_{v'\in\mathcal{N}(u')} h_{v'}^k||$, lo cual daría lugar a problemas en el aprendizaje y la optimización del modelo.

Para solucionar este problema, agregamos un elemento de normalización en AGGREGATE; en primera instancia, sería lógico pensar en

$$m_{\mathcal{N}(u)}^k = \frac{\sum_{v\in\mathcal{N}(u)} h_v^k}{|\mathcal{N}(u)|}. \tag{5.9}$$

Otra alternativa es la llamada normalización simétrica, que también tiene en cuenta el grado de los vecinos de $u$ a la hora de normalizar, con la idea de que un vecino que esté conectado con muchos otros nodos es poco indicativo de las propiedades o la estructura en la que se encuentra $u$.

$$m^k_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \frac{h^k_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}} \tag{5.10}$$

Basada en esta idea de normalización, se definen las *graph convolutional networks*, propuestas por primera vez por Kipf y Welling en 2016 [15], que también utilizan la idea de añadir bucles de la Ecuación 5.7. Nosotros utilizaremos *graph convolutional* networks con la idea de normalización de la Ecuación 5.9 (Ecuaciones 5.11 y 5.12 en forma matricial), aunque también son comunes las que utilizan la normalización simétrica (Ecuaciones 5.13 y 5.14 en forma matricial):

$$h^k_u = \sigma \left( W^k \frac{\sum_{v \in \mathcal{N}(u) \cup \{u\}} h^{(k-1)}_v}{|\mathcal{N}(u)|} \right), \tag{5.11}$$

$$H^k = \sigma \left( \tilde{D}^{-1} \tilde{A} H^{(k-1)} W^{k-1} \right), \tag{5.12}$$

$$h^k_u = \sigma \left( W^k \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{h_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}} \right), \tag{5.13}$$

$$H^k = \sigma \left( \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(k-1)} W^{k-1} \right), \tag{5.14}$$

donde $\tilde{A} = A + I$ y $\tilde{D}$ es la matriz diagonal de grados: $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$.

## 5.4. Deep Graph Convolutional Network

Hasta ahora, hemos visto arquitecturas de GNN que sirven para actualizar la información de los nodos un determinado número de veces, y obtener un output

de la última capa, $x_u$, que nos sirve para clasificar el nodo $u$. Sin embargo, ¿cómo modificamos una GNN para que clasifique el grafo entero que recibe como input? La *Deep graph convolutional network* (DGCN) [35], que explicamos a continuación, resuelve este tipo de problemas, y es la GNN utilizada en el método SEAL.

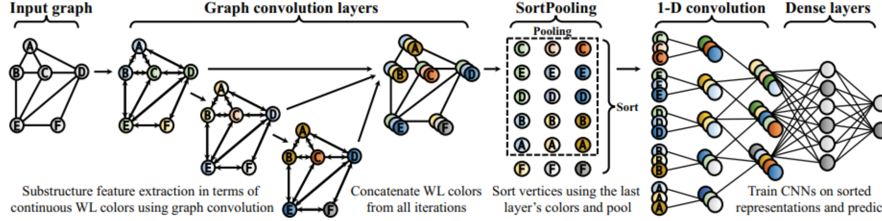


Figura 5.4: Esquema de la arquitectura DGCNN. En el esquema, se refiere a las features obtenidas por cada capa como *colors*, en referencia al kernel *Weisfeiler-Lehman*, en el que está basada la arquitectura de la GCN [35].

La arquitectura de una DGCN consiste en lo siguiente:

1. Primero, aplicamos una GCN de $l$ capas sobre el grafo, de acuerdo con la Ecuación 5.12, para obtener embeddings $h_u^l = x_u$ para cada nodo $u \in \mathcal{V}$, pero en su lugar, almacenamos los vectores $h_u^k$ para cada capa $k$ y damos como *output*, para cada nodo $u \in \mathcal{V}$, los vectores $h_u^k, k = 1, ..., l$ concatenados horizontalmente. Es decir, obtenemos una matriz $H^{1:l} = [H^1, H^2, ..., H^l]$ de manera que $H^{1:l} \in \mathbb{R}^{|\mathcal{V}| \times \sum_{t=1}^{l} c_t}$, donde $c_t$ es la dimensión de los vectores $h_u^t$.

2. A continuación, dotamos de un orden a los vértices del grafo, utilizando la matriz $H^{1:l}$ del paso anterior. El método utilizado para ordenarlos es *Sort-Pooling*, que consiste en, primero, ordenar las filas de la matriz de acuerdo con $H^l$, comparando primero la última *feature* y escogiendo la mayor, y en caso de que sean iguales, comparando la penúltima, etc. En caso de que los $h_u^l$ sean iguales para dos vértices, se compara el vector de la capa anterior $(l-1)$, etc. Por lo tanto, se trata de un orden lexicográfico, pero donde leemos las coordenadas de derecha a izquierda. Una vez ordenados, se escoge un número fijo de filas $k$, y se descartan las que sobren, o se completa la matriz con ceros en caso de que el subgrafo tenga menos de $k$ nodos. Esto se hace precisamente porque no todos los subgrafos recubridores tienen la misma cantidad de nodos, pero es necesario fijar una cantidad de filas para el siguiente paso.

3. Una vez que tenemos el output de *SortPooling*, una matriz $H^{\text{sp}}$ como la anterior, pero esta vez en $\mathbb{R}^{k \times \sum_{t=1}^{l} c_t}$, la reescribimos como un vector de longitud $k(\sum_{t=1}^{l} c_t)$ concatenando horizontalmente las filas, para utilizarlo

como input de una red neuronal, que clasificará el grafo. Esta red neuronal consta de, primero, una capa de convolución 1-D, de tamaño y *step* $\sum_{t=1}^{l} c_t$. A continuación, se añaden algunas capas *MaxPooling*, y algunas capas de convolución 1-D, para acabar con varias capas completamente conexas, y una capa final con una activación *SoftMax*, con tantas neuronas como clases haya en las que clasificar el grafo. En nuestro caso tendríamos dos clases, correspondientes a predecir que (no) existe la arista en la que se centra el grafo recubridor.

# Capítulo 6

# El método SEAL

En este capítulo explicaremos la metodología SEAL para predecir una arista entre dos nodos $x$ e $y$ en un grafo, la cual se basa en usar la información de los llamados subgrafos $h$-recubridores de $x$ e $y$ (Definición 2.1.14). Esta metodología queda justificada por los resultados de la Subsección 6.1.3, donde veremos que estos subgrafos contienen casi toda la información necesaria sobre la existencia de la arista.

Al principio del capítulo, en la Sección 6.1, exploraremos algunas de las heurísticas más importantes para la predicción de aristas entre dos nodos. Además, veremos que estas heurísticas se pueden encuadrar en una clase, las $\gamma$-heurísticas, las cuales tienen una propiedad clave, que es que pueden ser aproximadas con subgrafos $h$-recubridores de $x$ e $y$, donde el error de aproximación decrece de manera exponencial respecto de $h$. Esto nos indica que dichos subgrafos contienen la mayoría de la información necesaria para realizar una predicción de la arista, incluso si el valor de $h$ es bajo, lo que justifica utilizarlos como datos de entrada al modelo que realizará la predicción.

Además de la utilización de subgrafos recubridores, la metodología también se basa en la generación de *features* adicionales a las intrínsecas del problema (Secciones 6.2.2 y 6.2.3). Por una parte, se etiquetan los nodos del subgrafo en función de su rol estructural respecto de los nodos base $x$ e $y$ (etiquetas de estructura), y por otra parte, se utilizan *node embeddings*, codificaciones de los nodos en vectores numéricos que almacenan su información estructural (como por ejemplo, *node2vec*). Los tres tipos de features se concatenan para obtener el vector de *features* de cada nodo; esto, junto con los subgrafos, es por lo que el método es conocido como SEAL; *learning from Subgraphs, Embeddings, and Attributes for Link prediction.* Finalmente, se escoge una arquitectura o modelo, diseñada para realizar una clasificación binaria de los subgrafos (existencia o no de la arista), el cual se entrena con la ayuda de un muestreo de datos de

entrenamiento negativos (parejas de nodos entre los que no existe arista en el grafo).

## 6.1. Heurísticas de predicción de aristas

En esta sección, estudiaremos algunos índices heurísticos para medir la probabilidad de que exista una arista entre un par de nodos cualesquiera $(x, y)$, en base a la estructura del grafo. Veremos una serie de heurísticas que se pueden englobar en una clase, las $\gamma$-heurísticas. Probaremos que cualquier $\gamma$-heurística se aproxima bien con subgrafos $h$-recubridores, cosa que justificará el uso de los mismos como datos de entrenamiento para el modelo SEAL.

En lo que sigue, trabajamos con un grafo simple $G = (\mathcal{V}, \mathcal{E})$, y $x, y$ serán nodos del grafo. Alternativamente, podremos referirnos a los nodos del grafo mediante el conjunto ordenado $\{v_1, ..., v_{|\mathcal{V}|}\}$.

Una heurística $\mathcal{H}(x, y)$ es un cálculo en función de los nodos $x, y \in \mathcal{V}$ e información estructural del grafo, que da una estimación de la probabilidad de que exista la arista entre $x$ e $y$. Definimos el orden de una heurística como el $h$ mínimo de manera que el subgrafo $h$-recubridor contenga toda la información necesaria para el cálculo exacto de la heurística. Sin embargo, también existen heurísticas que necesitan información del grafo entero para ser calculadas de manera exacta. En ese caso, decimos que son heurísticas de orden alto.

### 6.1.1. Heurísticas de orden bajo

Todas las heurísticas de orden bajo que veremos son de primer o segundo orden, y siguen principios muy básicos.

**Definición 6.1.1.** Definimos la heurística de vecinos comunes como

$$\text{CN}(x, y) = |\mathcal{N}(x) \cap \mathcal{N}(y)|. \tag{6.1}$$

**Definición 6.1.2.** El índice de Jaccard, que sirve en general para teoría de conjuntos, se define como

$$\text{Jac}(x, y) = \frac{|\mathcal{N}(x) \cap \mathcal{N}(y)|}{|\mathcal{N}(x) \cup \mathcal{N}(y)|}. \tag{6.2}$$

**Definición 6.1.3.** El índice de conexión preferencial se define como

$$\text{PA}(x, y) = |\mathcal{N}(x)||\mathcal{N}(y)|. \tag{6.3}$$

Como podemos ver, se tratan de heurísticas de primer orden muy simples, que siguen el principio básico de que mientras más vecinos tengan en común

los nodos $x$ e $y$, más posibilidades hay de que exista la arista entre ellos. Claro está que, tanto vecinos comunes como el índice de conexión preferencial no dan un valor de probabilidad como resultado (cosa que sí hace el de Jaccard), pero siguen siendo índices indicadores de la probabilidad de la arista.

Veamos ahora dos heurísticas de segundo orden, similares y algo más complejas que las tres primeras que hemos visto.

**Definición 6.1.4.** Definimos el índice de Adamic-Adar como

$$\text{AA}(x, y) = \sum_{z \in \mathcal{N}(x) \cap \mathcal{N}(y)} \frac{1}{\log |\mathcal{N}(z)|}. \tag{6.4}$$

**Definición 6.1.5.** La heurística de *resource allocation* (asignación de recursos) se define como

$$\text{RA}(x, y) = \sum_{z \in \mathcal{N}(x) \cap \mathcal{N}(y)} \frac{1}{|\mathcal{N}(z)|} \tag{6.5}$$

En ambos casos, se mide cómo de "exclusivos" son los vecinos de $x$ y de $y$, es decir, un nodo que tenga muchos más vecinos además de $x$ e $y$ es menos significativo del parecido entre ellos que un nodo que los conecte exclusivamente a ellos. En el caso de Adamic-Adar, se calcula de manera que los nodos menos significativos tengan un aporte mayor al índice que en *resource allocation*. Ambas heurísticas son de segundo orden, ya que no se necesita información más allá de los segundos vecinos de $x$ e $y$.

### 6.1.2. Heurísticas de orden alto

Las heurísticas de orden alto necesitan información de todo el grafo para ser calculadas de manera exacta. Por lo que, si uno piensa en trabajar con subgrafos $h$-recubridores, puede parecer necesario un $h$ bastante grande para tener una buena aproximación de las mismas. En esta sección, definiremos algunas heurísticas de orden alto, demostraremos que pertenecen a la clase de $\gamma$-heurísticas, y probaremos que no es necesario un $h$ demasiado grande para aproximarlas bien.

**Definición 6.1.6.** Dado un grafo $G = (\mathcal{V}, \mathcal{E})$ y un par de nodos $x, y \in \mathcal{V}$, definimos el **índice de Katz** [13] como

$$\text{Katz}_{x,y} = \sum_{l=1}^{\infty} \beta^l [A^l]_{x,y}, \tag{6.6}$$

donde $[A^l]_{x,y}$ es el número de caminos de longitud $l$ entre $x$ e $y$ (Teorema 2.1.12), y $\beta \in (0, 1)$ es un parámetro con el que damos más peso a los caminos más cortos.

El índice de Katz estima que es más probable que exista una arista entre dos nodos si hay muchos caminos entre esos nodos, dando especial importancia a los de longitud pequeña.

La siguiente heurística calcula la distribución estacionaria de un camino aleatorio comenzando en el nodo $x$, que se mueve cada vez a un vecino aleatorio con probabilidad $\alpha$, o regresa a $x$ con probabilidad $1-\alpha$ (esto es un ejemplo de cadena de Markov, de la que proviene la distribución estacionaria [22]). Sea $\pi_x$ el vector de probabilidades de la distribución estacionaria, donde $[\pi_x]_i$ denota la probabilidad de que el camino se encuentre en el nodo $x$ (coordenada i). Si el camino no tuviese la opción de volver al nodo $x$, $(\alpha = 1)$ la matriz de transición sería $P$, donde $P_{i,j} = \frac{1}{|\mathcal{N}(v_j)|}$ si $(v_i, v_j) \in \mathcal{E}$ y $P_{i,j} = 0$ en caso contrario. Entonces, la distribución estacionaria satisface

$$\pi_x = \alpha P \pi_x + (1-\alpha)e_i,$$

donde $e_i$ es el vector canónico (1 en la coordenada $i$ y 0 en el resto).

**Definición 6.1.7.** Definimos entonces la heurística *PageRank* como

$$\text{PageRank}(x, y) = [\pi_x]_y \tag{6.7}$$

Veamos ahora *SimRank*, una heurística que se define de manera recursiva, y que se basa en la idea de que dos nodos son parecidos si sus entornos son parecidos.

**Definición 6.1.8.** La heurística *SimRank* se define de manera recursiva como $s(x, y) = 1$ si $x = y$, y en caso contrario, como

$$\gamma \frac{\sum_{a\in\mathcal{N}(x)} \sum_{b\in\mathcal{N}(y)} s(a, b)}{|\mathcal{N}(x)||\mathcal{N}(y)|}. \tag{6.8}$$

### 6.1.3. $\gamma$-heurísticas

¿Cómo se define entonces la clase de $\gamma$-heurísticas?

**Definición 6.1.9.** Una $\gamma$-heurística sobre $x, y \in \mathcal{V}$ es de la forma

$$\mathcal{H}(x, y) = \eta \sum_{l=1}^{\infty} \gamma^l f(x, y, l), \tag{6.9}$$

donde $\gamma \in (0, 1)$, $\eta$ es, o bien una constante positiva, o bien una función positiva de $\gamma$ acotada superiormente por una constante, y $f$ es una función no negativa definida sobre el grafo.

La propiedad clave de las $\gamma$-heurísticas es que, bajo ciertas condiciones, pueden ser aproximadas por un subgrafo $h$-recubridor, de manera que el error decrece, al menos, **exponencialmente** respecto de $h$.

**Teorema 6.1.10** ([34]). *Dada una $\gamma$-heurística $\mathcal{H}(x,y) = \eta \sum_{l=1}^{\infty} \gamma^l f(x,y,l)$, si $f(x,y,l)$ satisface las siguientes condiciones:*

1. *$f(x,y,l) \leq \lambda^l$, donde $\lambda < \frac{1}{\gamma}$.*
2. *$f(x,y,l)$ se puede calcular con información de $G^h_{x,y}$ para $l = 1,2,...,g(h)$, donde $g(h) = ah + b$ con $a, b \in \mathbb{N}$ y $a > 0$,*

*entonces $\mathcal{H}(x,y)$ puede aproximarse con información de $G^h_{x,y}$ de manera que el error decrece al menos exponencialmente respecto de $h$.*

*Demostración.* Podemos aproximar la $\gamma$-heurística sumando sobre sus primeros $g(h)$ términos, que nos permiten el cálculo de $f(x,y,l)$ gracias a la condición 2.

$$\tilde{\mathcal{H}}(x,y) = \eta \sum_{l=1}^{g(h)} \gamma^l f(x,y,l). \tag{6.10}$$

El error de aproximación puede ser acotado de la siguiente manera:

$$\begin{aligned} |\mathcal{H}(x,y) - \tilde{\mathcal{H}}(x,y)| &= \eta \sum_{l=g(h)+1}^{\infty} \gamma^l f(x,y,l) \\ &\leq \eta \sum_{l=ah+b+1}^{\infty} \gamma^l \lambda^l = \eta(\gamma\lambda)^{ah+b+1}(1-\gamma\lambda)^{-1}. \end{aligned}$$

□

En el último paso, entra en juego la condición $\lambda < \frac{1}{\gamma}$ para poder realizar la suma de la serie geométrica de razón $\gamma\lambda < 1$. Este resultado sugiere que casi toda la información relevante para la existencia (o no) de la arista se encuentra en un subgrafo $h$-recubridor, sin necesidad de que $h$ sea demasiado grande, y esto constituye el hecho principal en el que se basa la metodología SEAL, como veremos en la Subsección 6.2.4. Ahora, veamos que las heurísticas de orden alto que hemos definido son $\gamma$-heurísticas, y que cumplen las condiciones del Teorema 6.1.10.

**Lema 6.1.11** ([34]). *Cualquier camino entre $x$ e $y$ de longitud $l \leq 2h+1$ está contenido en $G^h_{x,y}$.*

*Demostración.* Dado un camino $w = (x, v_1, ..., v_{l-1}, y)$, con longitud $l$, probaremos que cualquier nodo está en $G^h_{x,y}$. Dado cualquier $v_i$ de $w$, supongamos que $d(v_i, x) \geq h + 1$ y que $d(v_i, y) \geq h + 1$. Entonces $2h + 1 \geq l = |(x, v_1, ..., v_i)| + |(v_i, ..., v_{l-1}, y)| \geq d(v_i, x) + d(v_i, y) = 2h + 2$, lo cual es una contradicción. Por lo tanto, $v_i \in G^h_{x,y}$. □

El índice de Katz (Ecuación 6.6) está definido en forma de $\gamma$-heurística, con $\eta = 1$, $\gamma = \beta$, y $f(x, y, l) = [A^l]_{x,y}$ que, de acuerdo con el Lema 6.1.11, es calculable con $G^h_{x,y}$ si $l \leq 2h + 1$, por lo que se satisface la Condición 2. del Teorema 6.1.10. Veamos que se cumple también la Condición 1.

**Teorema 6.1.12** ([34])**.** *Para dos nodos cualesquiera $i, j \in \mathcal{V}$, $[A^l]_{i,j}$ está acotado por $d^l$, donde $d$ es el grado máximo de los nodos del grafo.*

*Demostración.* Lo probamos por inducción en $l$. Si $l = 1$, es directo que $[A]_{i,j} \leq d$. Ahora, supongamos que $[A^l]_{i,j} \leq d^l$ para cualquier par $(i, j)$. Entonces,

$$[A^{l+1}]_{i,j} = \sum_{k=1}^{|\mathcal{V}|} [A^l]_{i,k} A_{k,j} \leq d^l \sum_{k=1}^{|\mathcal{V}|} A_{k,j} \leq d^l d = d^{l+1}.$$

□

Por lo tanto, si tomamos $\lambda = d$, podemos tomar $\beta$ suficientemente pequeño como para que $d \leq \frac{1}{\beta}$, y así cumpla la propiedad 1. del Teorema 6.1.10. Precisamente esto se hace en la práctica, donde se toman valores como $\beta = 5\text{e}{-4}$, que aseguran que se de la desigualdad anterior.

Para probar que *PageRank* es una $\gamma$-heurística, necesitamos escribir $[\pi_x]_y$ de la siguiente forma [12]:

$$[\pi_x]_y = (1 - \alpha) \sum_{w : x \rightsquigarrow y} P[w] \alpha^{|w|}, \tag{6.11}$$

donde $w : x \rightsquigarrow y$ denota cualquier camino de $x$ a $y$, $|w|$ es su longitud, y $P[w] = \prod_{i=0}^{k-1} \frac{1}{|\mathcal{N}(v_i)|}$, interpretable como la probabilidad de recorrer el camino $w$.

**Teorema 6.1.13** ([34])**.** *La heurística PageRank es una $\gamma$-heurística que cumple las condiciones del Teorema 6.1.10.*

*Demostración.* Podemos escribir la Ecuación 6.11 de la siguiente manera:

$$[\pi_x]_y = (1 - \alpha) \sum_{l=1}^{\infty} \sum_{\substack{w : x \rightsquigarrow y \\ |w| = l}} P[w] \alpha^l, \tag{6.12}$$

y, si definimos $f(x,y,l) = \sum_{\substack{w:x\rightsquigarrow y \\ |w|=l}} P[w]$, la tenemos escrita en forma de $\gamma$-heurística, con $\eta = 1-\alpha$ y $\gamma = \alpha$. Por otro lado, $f(x,y,l)$ es la probabilidad de que un camino aleatorio de longitud $l$ que comienza en $x$ se detenga en $y$, que satisface $\sum_{z\in\mathcal{V}} f(x,z,l) = 1$, por lo que $f(x,y,l) \leq 1 < \frac{1}{\alpha}$ (Condición 1). Por otro lado, el Lema 6.1.11 nos asegura que $f(x,y,l)$ es calculable con $G^h_{x,y}$, si $l \leq 2h+1$ (Condición 2). □

Al igual que con *PageRank*, es necesario escribir *SimRank* de la siguiente manera para probar que es $\gamma$-heurística [11]:

$$s(x,y) = \sum_{w:(x,y)\rightsquigarrow(z,z)} P[w]\gamma^{|w|}, \tag{6.13}$$

donde $w:(x,y)\rightsquigarrow(z,z)$ denota un par de caminos de igual longitud de manera que uno comienza en $x$, el otro comienza en $y$ y que se encuentren por primera vez en cualquier nodo $z \in \mathcal{V}$. De manera similar, $P[w] = \frac{1}{|\mathcal{N}(v_i)||\mathcal{N}(u_i)|}$ describe la probabilidad de recorrer esos caminos.

**Teorema 6.1.14** ([34])**.** *SimRank es una $\gamma$-heurística que cumple las condiciones del Teorema 6.1.10.*

*Demostración.* Reescribimos 6.13:

$$s(x,y) = \sum_{l=1}^{\infty} \sum_{\substack{w:(x,y)\rightsquigarrow(z,z) \\ |w|=l}} P[w]\gamma^{l} \tag{6.14}$$

y definimos $f(x,y,l) = \sum_{\substack{w:(x,y)\rightsquigarrow(z,z) \\ |w|=l}} P[w]$, por lo que *SimRank* es una $\gamma$-heurística con $\eta = 1$, y el mismo coeficiente $\gamma$ de la Definición 6.1.8. Además, $f(x,y,l) \leq 1 < \frac{1}{\gamma}$ (Condición 1) y, por otra parte, es directo que $f(x,y,l)$ es calculable usando $G^h_{x,y}$ para $l \leq h$ (Condición 2). □

## 6.2. SEAL

En esta sección explicamos los distintos tipos de datos con los que trabaja la metodología SEAL para realizar las predicciones y, finalmente, definiremos los pasos que sigue.

### 6.2.1. *Features* explícitas

Dado un problema modelado por un grafo $G = (\mathcal{V}, \mathcal{E})$, tenemos una matriz de *features* $X \in \mathbb{R}^{|\mathcal{V}|\times d}$ (donde cada fila es el vector de *features* de cada nodo). Entonces, llamamos a estas $d$ *features* intrínsecas del problema ***features***

**explícitas**. Por ejemplo, si tratamos con el grafo de una red social como *Facebook* donde los nodos son los usuarios, y queremos predecir la probabilidad de que dos usuarios sean amigos (que exista una arista entre ellos), el vector de *features* explícitas de un usuario podría contener información como su edad, su nacionalidad, cuántas fotografías ha subido, etc.

Por lo tanto, estas *features* solo requieren del procesado previo necesario, dependiendo de en qué formato vengan dadas en el grafo, para codificar la información que aporten de manera numérica. Además de estas *features* intrínsecas del problema, utilizamos *features* adicionales como *input* al modelo, que explicamos a continuación.

### 6.2.2. Etiquetas de estructura

Las etiquetas de estructura se generan para cada nodo del subgrafo $h$-recubridor del par de nodos $(x, y)$, y tienen como objetivo identificar el rol que tiene cada uno de los nodos en el subgrafo, en función de la distancia a los nodos centrales $x$ e $y$. Por ejemplo, los nodos $x$ e $y$ tienen su etiqueta que los identifica como los nodos entre los que hay que predecir la arista, mientras que otros nodos con distintas posiciones relativas al centro (los nodos $x$ e $y$) tienen distinta importancia para la existencia de la arista.

Las etiquetas de estructura deberían resaltar correctamente las diferencias estructurales entre nodos del subgrafo, especialmente entre los nodos $x$ e $y$ y el resto, para que el modelo los identifique como entre los que se encuentra la arista. O por ejemplo, para distinguir un nodo que esté a distancia 1 tanto de $x$ como de $y$ (que es de esperar que aporte mucha información sobre la estructura del subgrafo y sobre la existencia de la arista al ser un nodo muy central) de uno que esté a distancia 2 de $x$ y a distancia 3 de $y$, ya que es un nodo más periférico cuya información quizás no sea tan relevante para la predicción de la arista.

También buscamos una manera de etiquetar los nodos que asigne la misma etiqueta a dos nodos distintos con el mismo rol estructural. Siguiendo el ejemplo anterior, un nodo a distancia 3 de $x$ y a distancia 2 de $y$ debería tener la misma etiqueta que uno a distancia 2 de $x$ y 3 de $y$. Por lo tanto, seguimos los siguientes criterios para etiquetar los nodos:

1. Los nodos centrales $x$ e $y$ siempre tienen la etiqueta 1.
2. Dos nodos $i$ y $j$ tienen la misma etiqueta si $d(i, x) = d(j, x)$ y $d(i, y) = d(j, y)$, o bien si $d(i, x) = d(j, y)$ y $d(i, y) = d(j, x)$.

Esto se debe a que la posición de un nodo respecto a los nodos $x$ e $y$ en el subgrafo recubridor se puede describir en base a su radio respecto a esos nodos,

es decir, $(d(i,x), d(i,y))$. De esta manera, le damos la misma etiqueta a nodos en la misma “órbita’, para que quede reflejada su posición relativa y que cumplen el mismo rol estructural en el subgrafo. Dado un nodo $i$, denotamos su etiqueta estructural como $f_l(i)$.

Basándonos en estos criterios, etiquetamos los nodos de la siguiente manera: primero, etiquetamos a $x$ e $y$ con 1. Después, para cada nodo $i$ tal que $(d(i,x), d(i,y)) = (1,1)$, lo etiquetamos con 2. Los nodos con radios $(1,2)$ o $(2,1)$ reciben la etiqueta 3. Nodos con radios $(1,3)$ o $(3,1)$ reciben la etiqueta 4, con radios $(2,2)$ reciben la etiqueta 5 y con radios $(1,4)$, etc. Es decir, asignamos etiquetas mayores a nodos con suma de los radios mayores, guardando un orden con las combinaciones de radios que suman igual (orden en base al mínimo de ambos radios). Para cada nodo $i$, las etiquetas $f_l(i)$ y los radios $(d(i,x), d(i,y))$ satisfacen:

1. Si $d(i,x) + d(i,y) \neq d(j,x) + d(j,y)$, entonces

$$d(i,x) + d(i,y) < d(j,x) + d(j,y) \Leftrightarrow f_l(i) < f_l(j).$$

2. Si $d(i,x) + d(i,y) = d(j,x) + d(j,y)$, entonces

$$d(i,x)d(i,y) < d(j,x)d(j,y) \Leftrightarrow f_l(i) < f_l(j).$$

Una ventaja de este etiquetado es que se puede expresar como un solo cálculo, lo que facilita la computación de las etiquetas:

$$f_l(i) = 1 + \text{mín}(d_x, d_y) + (d/2)[(d/2) + (d\%2) - 1],$$

donde $d_x = d(i,x)$, $d_y = d(i,y)$, $d = d_x + d_y$, y $d/2$ y $d\%2$ son la división entera y el resto de dividir $d$ entre 2, respectivamente. Finalmente, codificamos las etiquetas de cada nodo como vectores *one-hot*, y obtenemos una matriz $X$ que codifica la información de las etiquetas de cada nodo del subgrafo.

### 6.2.3. *Node Embeddings*

El segundo tipo de información con la que trabaja SEAL son los *node embeddings*. Estos vectores, como su nombre indica, son representaciones o *embeddings* de los nodos basados únicamente en la información estructural del grafo, y normalmente se obtienen mediante métodos que factorizan una matriz específica que provenga del grafo, como la matriz de adyacencia o la matriz Laplaciana. Algunos métodos mediante los cuales se obtienen dichos *embeddings* son *node2vec* [7], *DeepWalk* [20] o VGAE (*Variational Graph Auto-encoder*) [14]. La idea de obtener *node embeddings* es obtener una representación vectorial de los nodos, de manera que dos representaciones sean similares si los nodos cumplen

roles estructurales similares en el grafo. Esto sirve para, por ejemplo, utilizar esas representaciones vectoriales por parejas como input de una red neuronal, y entrenarla para que calcule la probabilidad de la arista entre la pareja de nodos correspondiente. Este es el funcionamiento de los métodos con los que se compara el rendimiento de SEAL en la Sección 6.3, a los que nos referimos directamente con el nombre del método para obtener los *node embeddings*, como pueden ser los ya mencionados VGAE o *node2vec*. Esta es la razón principal por la que utilizaremos los *node embeddings* como una de las fuentes de información de las que dispone SEAL para su aprendizaje.

Sin embargo, ¿qué entendemos como similaridad entre dos nodos de un grafo? Hay dos maneras principales de entenderlo, aunque no entramos en detalle sobre a cuál es más afín cada uno de los métodos que mencionaremos:

1. Homofilia: dos nodos son similares mientras más cercanos sean. Por ejemplo, en un grafo que representa una red social, queremos obtener embeddings parecidos para usuarios que se encuentran en un mismo círculo de amigos.

2. Equivalencia estructural: dos nodos son similares mientras más se parezcan los roles estructurales que cumplen en la red. A diferencia de la homofilia, dos usuarios que sean amigos no tienen por qué tener representaciones similares. Sin embargo, dos usuarios que, aunque estén muy lejos el uno del otro, ambos son puentes entre dos círculos de amigos tendrán una representación similar, al igual que dos usuarios que sean el nodo central de una comunidad (conectados con casi todos los nodos de la comunidad), o usuarios que sean hojas (conectados con sólo una persona de su comunidad).

Hay un detalle a tener en cuenta a la hora de generar los *embeddings* para usarlos como *input* de SEAL, y es que lo hacemos de cierta forma. Supongamos que tenemos el grafo
$G = (\mathcal{V}, \mathcal{E})$, una muestra de aristas de entrenamiento positivas $\mathcal{E}_p \subseteq \mathcal{E}$ y una muestra de aristas de entrenamiento negativas $\mathcal{E}_n$ tal que $\mathcal{E}_n \cap \mathcal{E} = \emptyset$. Si generamos los *embeddings* directamente sobre el grafo $G$, estos codificarán la información de existencia de las aristas del grafo, en particular de las aristas $\mathcal{E}_p$. Se ha comprobado experimentalmente en [34] que las GNNs pueden detectar rápidamente esta información de existencia de aristas, y optimizarse utilizando sólo esa parte de la información que se les proporcione, lo que resulta en problemas de generalización del modelo (buen rendimiento en el conjunto de entrenamiento, pero malo en el de test). El truco para solucionarlo consiste en incluir temporalmente las aristas de $\mathcal{E}_n$ en $\mathcal{E}$, es decir, generar los *embeddings* sobre el grafo $G' = (\mathcal{V}, \mathcal{E} \cup \mathcal{E}_n)$. De esta manera, los *embeddings* codificarán la información de existencia de las aristas de entrenamiento, tanto negativas como positivas, de manera que la GNN no pueda clasificar las aristas usando solo

esta parte de la información. Se ha verificado experimentalmente la mejora de rendimiento asociada a este método, bautizado como *negative injection* [34].

### 6.2.4. Definición de SEAL

Ahora que conocemos en detalle todos los elementos que intervienen, podemos definir la metodología SEAL. Dado un grafo $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ y nodos $x, y \in \mathcal{V}$, el cálculo mediante SEAL de la probabilidad de que exista una arista entre ellos se divide en tres pasos:

1. Se calcula el subgrafo $h$-recubridor, $G^h_{(x,y)}$ para cierto $h > 0$.
2. Se construye la matriz de *features* asociada a ese subgrafo: Para ello, se concatenan para cada nodo los tres tipos de *features*: explícitas, *node embeddings* y la etiqueta estructural (codificada como vector *one-hot*), para obtener el vector de *features* final. Finalmente, se forma la matriz $X$, cuyas filas son los vectores correspondientes a cada uno de los nodos del subgrafo.
3. Se escoge una arquitectura de GNN diseñada para clasificación binaria de grafos, que recibe el *input* $(G^h_{(x,y)}, X)$ y devuelve la probabilidad de pertenencia a la clase positiva, i.e., de que exista la arista entre $x$ e $y$.

Como podemos ver, nos basamos en los resultados teóricos que vimos en la Sección 6.1.3 (en especial, en el Teorema 6.1.10) para diseñar una metodología en la que el aprendizaje y la información para predecir la arista proviene de un subgrafo recubridor entre los nodos correspondientes. En definitiva, se puede decir que estamos aprendiendo automáticamente una nueva heurística mediante el entrenamiento de dicha arquitectura de GNN sobre el grafo (*dataset*) que nos interese.

## 6.3. Elección del modelo y evaluación del rendimiento de SEAL

SEAL es una metodología de elección de features y de elección de los subgrafos recubridores como datos a clasificar, pero es flexible con qué clasificador de grafos utilizar. De ahora en adelante, siempre que nos refiramos al método o modelo SEAL, nos referiremos a la metodología SEAL usada con el clasificador de grafos DGCNN que vimos en la Sección 5.4, ya que se trata también del modelo escogido en [34] para evaluar SEAL, y obtener así los resultados de rendimiento que presentamos en esta sección (obtenidos todos de experimentos realizados en [34]).

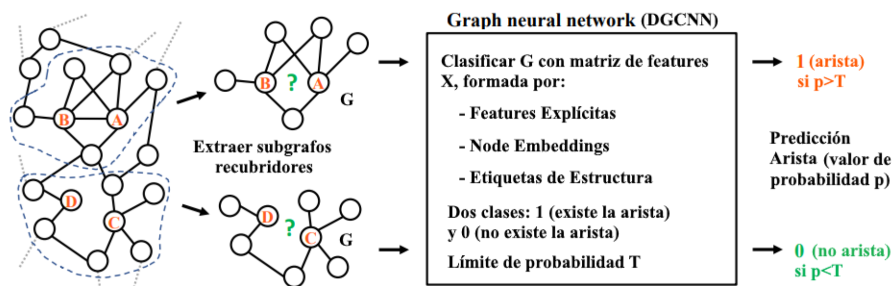


Figura 6.1: Esquema de la metodología SEAL [34]. Se visualizan dos posibles inputs; las parejas de nodos $(A, B)$ (entre los que existe arista) y $(D, C)$ (entre los que no existe arista). Podemos observar que, para ambos casos, se procede extrayendo el subgrafo recubridor, construyendo su matriz de información y utilizando un clasificador de grafos (p.ej., DGCNN) para obtener la predicción.

Para la obtención de esos resultados, se tomaron el 10 % de las aristas del grafo como datos positivos del conjunto de *test*, y se muestrea la misma cantidad de aristas no existentes (pares de nodos entre los cuales no existe arista) como datos de *test* negativos. Se realiza el mismo procedimiento para obtener un 5 % de las aristas como datos del conjunto de validación. El 90 % de aristas restantes se utilizan como datos de entrenamiento positivos y, análogamente, se muestrea la misma cantidad de aristas negativas para usarse como datos de entrenamiento negativos. Como parámetros concretos del modelo DGCNN tenemos:

- Cuatro capas de convolución, las tres primeras con *outputs* de dimensión 32, con una capa final con *output* de dimensión 1.
- Para la capa *SortPooling* se escoge el parámetro $k$ de manera que el 60 % de los grafos tengan menos de $k$ nodos. Es decir, que solo un 40 % de los subgrafos recubridores ven truncado su número de nodos hasta $k$, mientras que el resto, o bien ya tenía exactamente $k$ nodos, o ven su matriz completada con ceros. Los *outputs* son vectores (de dimensión $k(32 \cdot 4+1)$).

A continuación se encuentra la red neuronal final. La arquitectura de la misma consta de:

- Primero, una capa de convolución 1-D, con tamaño de *kernel* y *step* $32 \cdot 4 + 1$, y con 16 *features* de salida, es decir, que hay 16 conjuntos de pesos distintos en esta capa, que dan lugar a 16 *outputs* distintos para cada posición del *kernel*. Por lo tanto, el output de esta capa es una matriz de dimensiones $16 \times k$.
- A continuación, una capa *MaxPooling* de tamaño de *kernel* y *step* 2 para cada una de las 16 *features*.

- La matriz de dimensiones $16 \times \frac{k}{2}$ que obtenemos, la utilizamos a continuación como *input* de otra capa de convolución 1-D, con tamaño de *kernel* de 5 y *step* 1, con 32 *features* de salida. Esto nos deja con una matriz de dimensiones $32 \times (\frac{k}{2} - 4)$.
- Finalmente, aplicamos una capa lineal que toma, precisamente $32 \cdot (\frac{k}{2} - 4)$ valores de entrada, y devuelve 128, a los que aplicamos otra capa lineal, que lo transforma en solo 1 valor, al que finalmente aplicamos una función sigmoide para obtener la probabilidad que buscábamos.

El entrenamiento del modelo se realiza a lo largo de 50 épocas, y se escoge como modelo final el que haya resultado en un valor más bajo de la métrica AUC en el conjunto de validación. Los distintos *datasets* sobre los que se evalúa SEAL son *USAir, NS, PB, Yeast, C.ele, Power, Router y E.Coli* (consultar Apéndice C de [34] para más información sobre los mismos). Es importante determinar el valor $h$, que establece el tamaño de los subgrafos $h$-recubridores; se escoge $h = 1$ o $h = 2$, dependiendo del *dataset*, pero no se escogen valores iguales o mayores que 3, ya que, por un lado, en [34] se comprueba experimentalmente que el rendimiento no aumenta con $h > 3$ y, por otro lado, debido a que $h = 3$ puede acabar generando subgrafos demasiado grandes, sobre todo si un nodo central/muy conectado (*hub node*) se incluye en el grafo. En definitiva, el valor se selecciona de $\{1, 2\}$ en base al siguiente criterio: Si la heurística de segundo orden Adamic-Adar (AA, Definición 6.1.4) consigue un rendimiento superior a la heurística de primer orden de vecinos comunes (Definición 6.1.1), entonces se escoge $h = 2$; de lo contrario, se utiliza $h = 1$.

Primero, se compara el rendimiento de SEAL con las heurísticas que ya hemos visto, además de con otros dos métodos de aprendizaje automático que también aprenden de subgrafos: el kernel de Weisfeiler-Lehman (WLK, [25]), y WLNM [33] (Figura 6.2). Se incluye también *Ensemble* (ENS), que consiste en entrenar un clasificador de regresión logística sobre las ocho heurísticas. Dado que los métodos heurísticos solo obtienen información estructural de los subgrafos, no se han utilizado ni *features* explícitas ni *node embeddings* con SEAL, para que aprenda precisamente sólo de la estructura del subgrafo. Lo primero que podemos observar es que los métodos que utilizan subgrafos recubridores (WLK, WLNM y SEAL) tienen, en general, mejor rendimiento que los métodos heurísticos. Entre ellas, SEAL es la que presenta mejor rendimiento, lo que demuestra la mejora que supone el uso de GNN's respecto de los métodos utilizados por WLNM y WLK (*graph kernels* y redes neuronales completamente conexas). Por otra parte, podemos observar que, sobre los *datasets* *Power* y *Router*, las heurísticas presentan un rendimiento similar a adivinar al azar (i.e., $AUC \simeq 0.5$, ver Figura 3.3), mientras que los métodos de aprendizaje automático los mejoran muchísimo y obtienen un rendimiento muy bueno. Esto sugiere que, mediante el aprendizaje automático, se pueden incluso descubrir nuevas heurísticas para grafos sobre los cuales las heurísticas predefinidas no funcionan.

| **Data** | CN | Jaccard | PA | AA | RA | Katz | PR | SR | ENS | WLK | WLNM | **SEAL** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| USAir | 93.80±1.22 | 89.79±1.61 | 88.84±1.45 | 95.06±1.03 | 95.77±0.92 | 92.88±1.42 | 94.67±1.08 | 78.89±2.31 | 88.96±1.44 | **96.63**±0.73 | 95.95±1.10 | **96.62**±0.72 |
| NS | 94.42±0.95 | 94.43±0.93 | 68.65±2.03 | 94.45±0.93 | 94.45±0.93 | 94.85±1.10 | 94.89±1.08 | 94.79±1.08 | 97.64±0.25 | 98.57±0.51 | 98.61±0.49 | **98.85**±0.47 |
| PB | 92.04±0.35 | 87.41±0.39 | 90.14±0.45 | 92.36±0.34 | 92.46±0.37 | 92.92±0.35 | 93.54±0.41 | 77.08±0.80 | 90.15±0.45 | 93.83±0.59 | 93.49±0.47 | **94.72**±0.46 |
| Yeast | 89.37±0.61 | 89.32±0.60 | 82.20±1.02 | 89.43±0.62 | 89.45±0.62 | 92.24±0.61 | 92.76±0.55 | 91.49±0.57 | 82.36±1.02 | 95.86±0.54 | 95.62±0.52 | **97.91**±0.52 |
| C.ele | 85.13±1.61 | 80.19±1.64 | 74.79±2.04 | 86.95±1.40 | 87.49±1.41 | 86.34±1.89 | **90.32**±1.49 | 77.07±2.00 | 74.94±2.04 | 89.72±1.67 | 86.18±1.72 | **90.30**±1.35 |
| Power | 58.80±0.88 | 58.79±0.88 | 44.33±1.02 | 58.79±0.88 | 58.79±0.88 | 65.39±1.59 | 66.00±1.59 | 76.15±1.06 | 79.52±1.78 | 82.41±3.43 | 84.76±0.98 | **87.61**±1.57 |
| Router | 56.43±0.52 | 56.40±0.52 | 47.58±1.47 | 56.43±0.51 | 56.43±0.51 | 38.62±1.35 | 38.76±1.39 | 37.40±1.27 | 47.58±1.48 | 87.42±2.08 | 94.41±0.88 | **96.38**±1.45 |
| E.coli | 93.71±0.39 | 81.31±0.61 | 91.82±0.58 | 95.36±0.34 | 95.95±0.35 | 93.50±0.44 | 95.57±0.44 | 62.49±1.43 | 91.89±0.58 | 96.94±0.29 | 97.21±0.27 | **97.64**±0.22 |

Figura 6.2: Rendimiento (AUC) de diversas heurísticas en comparación con SEAL en distintos *datasets* ([34]).

A continuación, se compara SEAL con otros métodos de aprendizaje automático; *matrix factorization* (MF), *stochastic block model* (SBM) [1], *node2vec* (N2V) [7], LINE [27], *spectral clustering* (SPC) y *variational graph auto-encoder* (VGAE) [14]. Estos métodos consisten en obtener *node embeddings*, que luego son utilizados como datos de entrada de un clasificador binario que predice la existencia o no de una arista entre nodos. Los resultados se muestran en la Figura 6.3. Podemos observar que SEAL supone una mejora notable en general respecto al resto de métodos. En estos *datasets* en concreto, no disponemos de *features* explícitas, lo que llama la atención ya que SEAL (que, recordemos, utiliza los *embeddings* de *node2vec* como parte del *input*) mejora mucho el rendimiento de sólo *node2vec*, gracias al aprendizaje obtenido de los subgrafos y de las etiquetas de estructura. Una observación interesante es que, en ciertos casos, SEAL sin uso de *node embeddings* (6.2) mejora el rendimiento de SEAL cuando sí se hace uso de ellos, lo que indica que no siempre supone una mejora de rendimiento usarlos como parte del *input*.

| **Data** | MF | SBM | N2V | LINE | SPC | VGAE | **SEAL** |
|---|---|---|---|---|---|---|---|
| USAir | 94.08±0.80 | 94.85±1.14 | 91.44±1.78 | 81.47±10.71 | 74.22±3.11 | 89.28±1.99 | **97.09**±0.70 |
| NS | 74.55±4.34 | 92.30±2.26 | 91.52±1.28 | 80.63±1.90 | 89.94±2.39 | 94.04±1.64 | **97.71**±0.93 |
| PB | 94.30±0.53 | 93.90±0.42 | 85.79±0.78 | 76.95±2.76 | 83.96±0.86 | 90.70±0.53 | **95.01**±0.34 |
| Yeast | 90.28±0.69 | 91.41±0.60 | 93.67±0.46 | 87.45±3.33 | 93.25±0.40 | 93.88±0.21 | **97.20**±0.64 |
| C.ele | 85.90±1.74 | 86.48±2.60 | 84.11±1.27 | 69.21±3.14 | 51.90±2.57 | 81.80±2.18 | **89.54**±2.04 |
| Power | 50.63±1.10 | 66.57±2.05 | 76.22±0.92 | 55.63±1.47 | **91.78**±0.61 | 71.20±1.65 | 84.18±1.82 |
| Router | 78.03±1.63 | 85.65±1.93 | 65.46±0.86 | 67.15±2.10 | 68.79±2.42 | 61.51±1.22 | **95.68**±1.22 |
| E.coli | 93.76±0.56 | 93.82±0.41 | 90.82±1.49 | 82.38±2.19 | 94.92±0.32 | 90.81±0.63 | **97.22**±0.28 |

Figura 6.3: Rendimiento (AUC) de diversos métodos de aprendizaje automático, en comparación con el método SEAL sobre distintos *datasets* [34].

Como conclusión, hemos explorado algo de teoría sobre heurísticas para predicción de aristas y hemos explicado la metodología SEAL para predicción de aristas mediante *Graph Neural Networks* que, basada en el Teorema 6.1.10, utiliza como elementos clave subgrafos recubridores, además de *features* explícitas, *node embeddings* y etiquetas de estructura, junto con una arquitectura de GNN para clasificación de grafos. Posteriormente, hemos explicado los experimentos llevados a cabo en [34] utilizando esta metodología, junto con el clasificador de grafos DGCNN, y hemos explorado los resultados, concluyendo que SEAL supone una mejora respecto de otros muchos métodos de predicción de aristas del momento, justificando así su posterior uso en este trabajo.

# Capítulo 7

# Método GAug-M para *data augmentation* en grafos.

El campo de estudio de *data augmentation* tiene como objetivo aumentar o mejorar los datos de entrenamiento disponibles en un problema de aprendizaje supervisado, de manera que el modelo a entrenar mejore su rendimiento. En otros ámbitos del aprendizaje automático, como puede ser reconocimiento de imágenes, se emplean técnicas como voltear o invertir las imágenes para generar nuevos datos. En problemas de clasificación tradicionales (los datos no son más que vectores de *features*) se emplean técnicas de muestreo (*oversampling* o *undersampling*) y métodos de interpolación. En problemas de NLP (*Natural Language Processing*), un ejemplo de modificación de los textos consiste en sustituir ciertas palabras por sinónimos.

En el contexto de datos estructurados en grafos, se trata de un ámbito menos estudiado, y que requiere de estrategias distintas a las mencionadas anteriormente, ya que estamos ante un tipo de estructura de datos completamente distinto. En realidad, no son más que conjuntos de vértices y aristas, para los que modificar la representación gráfica no modifica la información que codifican, lo que limita mucho las técnicas para crear nuevos datos de entrenamiento. Por ejemplo, las técnicas de voletado de imágenes de *Computer Vision* no nos servirían. Por lo tanto, es necesario desarrollar métodos más específicos para grafos [4].

En este capítulo introducimos GAug-M (*Graph Augmentation Modified*[1], [37]), un método para modificar un grafo que, mediante un modelo de predicción de aristas, agrega al grafo las aristas más probables y elimina las aristas menos probables, para mejorar el rendimiento de un modelo de clasificación de nodos.

[1] En contraposición a *Graph Augmentation Original*, o GAug-O, también desarrollado en [37]

## 7.1. Posibles estrategias para *data augmentation* en grafos

Para abordar el problema de *data augmentation* en datos estructurados en forma de grafos, las ideas inmediatas pasan por añadir o eliminar nodos o aristas. En el caso de los nodos, añadir nodos presenta un problema a la hora de decidir la conectividad que el nuevo nodo tendrá con el resto de nodos ya existentes, así como el problema de dotarlo de un vector de *features*, mientras que eliminando un nodo solo se consigue reducir la información de entrenamiento. Por lo tanto, la estrategia que a priori parece más viable es añadir o eliminar aristas. El problema pasa a ser entonces, ¿qué método empleamos para añadir o eliminar aristas, de manera que la información del grafo se enriquezca y mejore el rendimiento de un modelo de clasificación de nodos?

Existen varios métodos que han sido propuestos recientemente para abordar este problema. DropEdge [21] elimina aleatoriamente una fracción de las aristas del grafo antes de cada época de entrenamiento, para evitar el *overfitting* de una manera parecida a como hace DropOut [26] para redes neuronales. Entre otras maneras más complejas de abordar el problema, más robustas en cuando a eliminación de ruido (datos o información no relevantes a la hora de la predicción) se encuentran AdaEdge [2], que añade (elimina) aristas de manera iterativa entre nodos para los que se ha predicho la misma (distinta) clase con alta confianza, o BGCN [36], que crea múltiples grafos tras eliminar ruido mediante un método estocástico que utiliza predicciones de una GCN. Sin embargo, en este trabajo, nos centraremos en GAug-M, que explicamos en la siguiente sección.

## 7.2. GAug-M

### 7.2.1. Motivación teórica

Un grafo sirve para representar el proceso o situación real de interés sobre la que queremos trabajar, pero no tiene porqué coincidir exactamente con el proceso o situación que pretende modelar (relaciones reales vs. relaciones observadas). Esto es debido a que muchos grafos que modelan situaciones reales son susceptibles al ruido, provocado por varias razones, como pueden ser:

- *Spammers* que contaminan el espacio de observaciones.
- En el preprocesado de los grafos, se pueden añadir o eliminar bucles, eliminar nodos aislados o aristas en base a sus pesos.

- Errores humanos. Por ejemplo, en un grafo que modele artículos de investigación como nodos, y aristas en función de cómo se citen unos a otros (p.ej.: los datasets *Cora* o *CiteSeer*) un paper puede omitir (incluir) la cita a un paper muy importante (irrelevante) por error, lo que provoca la ausencia (presencia) de una arista no deseada en el grafo.

Todas estas situaciones producen una diferencia entre el grafo que observamos, y el "grafo ideal", que puede afectar al rendimiento de nuestra tarea, que en este caso, sería clasificar los nodos en clases. En el mejor de los casos, mediante adición y eliminación de aristas, podríamos producir un grafo con conectividad ideal $\mathcal{G}_i$, donde aristas esperadas (pero inexistentes) sean agregadas, y aristas insignificantes (pero existentes) sean eliminadas. En la Figura 7.1 se puede apreciar un ejemplo sobre el famoso grafo *Zachary's Karate Club*, donde los nodos representan los miembros del club, y las aristas relacionan a miembros que socializan fuera del ámbito del club: añadir aristas entre nodos de la misma clase, y eliminar aristas entre nodos de distintos grupos provoca una mejora esperada en el rendimiento de la clasificación de nodos, siendo el mejor caso en el que añadimos/eliminamos todas las posibles aristas, cosa que lógicamente no podemos hacer, ya que requiere de conocer las clases de los nodos de antemano. El siguiente teorema muestra cómo, con conocimiento total de las clases de los nodos, la tarea de clasificación de nodos con una GNN pasa a ser trivial.

**Teorema 7.2.1** ([37])**.** *Sea $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un grafo simple con matriz de adyacencia $A$ y matriz de features de nodos $X$. Sea $u \in \mathcal{V}$, $f\colon A, X, W \to H$ una capa de GNN con una función de agregación invariante por permutaciones sobre $u \cup \mathcal{N}(u)$, con matriz de parámetros $W$, y sea $H = f(A, X, W)$ el embedding resultante. Supongamos que $\mathcal{G}$ contiene $k$ componentes conexas completas. Entonces, se tiene:*

1. *Para dos nodos $i, j \in \mathcal{V}$ cualesquiera que estén en una misma componente conexa, se tiene $H_i = H_j$.*
2. *Para dos nodos $i, j \in \mathcal{V}$ cualesquiera que estén en distintas componentes conexas, $S_a, S_b \in \mathcal{V}$, se tiene $H_i \neq H_j$ cuando $W$ no es una matriz de ceros y $\sum_{v \in S_a} X_v \neq \varepsilon \sum_{u \in S_b} X_u,\ \forall \varepsilon \in \mathbb{R}$.*

Esto nos motiva a buscar una manera de añadir y eliminar aristas de manera que se de importancia a las aristas intra-clase, y se eliminen aristas entre clases, asumiendo que es más probable que existan las primeras que las últimas (homofilia).

### 7.2.2. Definición de GAug-M

El método GAug-M sigue los siguientes pasos:

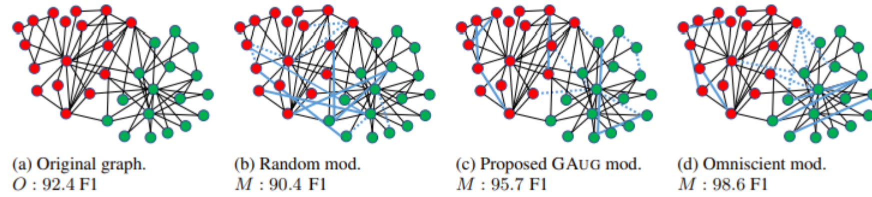


Figura 7.1: Rendimiento de una GCN, medido con el valor $F_1$, en el grafo *Zachary's Karate Club*, y en verisones modificadas del mismo. a) Grafo original, b) Aristas agregadas y eliminadas aleatoriamente, c) Aristas agregadas y eliminadas usando GAug-M, d) Modificaciones realizadas conociendo las clases de los nodos. Fuente: [37]

1. Se utiliza un modelo de predicción de aristas para obtener la matriz $M$ de probabilidades de existencia de todas las posibles aristas del grafo $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

2. Utilizando la matriz $M$, GAug-M añade, de manera determinista, las $i|\mathcal{E}|$ aristas no existentes con mayor probabilidad, y elimina las $j|\mathcal{E}|$ aristas ya existentes con menor probabilidad, para generar un nuevo grafo modificado $\mathcal{G}_m$, donde $i, j \in [0, 1]$ (i.e., porcentaje de aristas a añadir/eliminar).

3. A continuación, procedemos a entrenar y evaluar el modelo de clasificación de nodos, tanto en el grafo $\mathcal{G}$ como en el grafo $\mathcal{G}_i$, y se compara el rendimiento en ambos casos.

En [37], se utiliza VGAE (*Variational Graph Auto-Encoder*, [14]) como modelo predictor de aristas, y como modelos de clasificación de nodos se emplean GCN [15], GSAGE [9], GAT [29] y JK-Net [30].

Como comprobación experimental de que este método consigue el objetivo de promover las aristas entre nodos de la misma clase y desincentivar las aristas entre nodos de clases distintas, se ha aplicado GAug-M al grafo *Cora* utilizando probabilidades obtenidas por *Graph Auto Encoder* [37]. En la Figura 7.2 podemos ver el número total de aristas, tanto intra-clase como entre clases, en función del porcentaje de aristas agregadas y eliminadas ($i$ y $j$), en comparación con un predictor de probabilidades aleatorio, además de la métrica micro-F1 de un clasificador de nodos sobre el grafo evaluada sobre el conjunto de *test*. Podemos observar que al añadir aristas, GAug-M provoca un aumento mayor de las aristas intra-clase que entre clases, además de mejorar el rendimiento del clasificador consistentemente, mientras que añadir aristas de manera aleatoria no tiene un efecto claro sobre aristas intra-clase, sino que aumenta las aristas entre clases (debido a que, de todas las aristas posibles, existen más entre clases que intra-clase), además de empeorar el rendimiento del modelo. Se produce un efecto análogo al eliminar aristas aleatoriamente, solo que esta vez se eliminan

sobre todo aristas intra-clase (debido a que es el tipo más común entre las aristas existentes), mientras que eliminar aristas mediante GAug-M tiene un efecto de mejora de rendimiento con porcentajes entre el 15 % y 25 %, aunque lo empeora para porcentajes superiores, lo cual tiene sentido ya que estaríamos eliminando demasiadas aristas, y por lo tanto demasiada información del grafo, en lugar de solamente eliminar ruido.

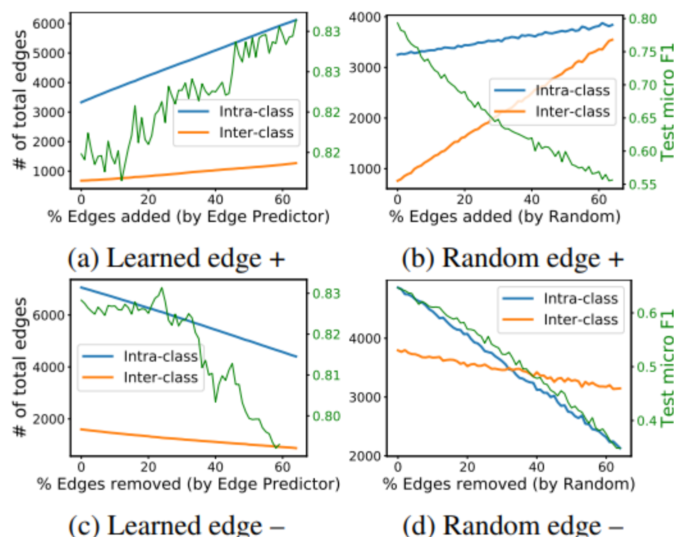


Figura 7.2: Efecto de GAug-M sobre las aristas del grafo *Cora*, y el rendimiento de un clasificador de nodos. Fuente: [37]

### 7.2.3. Evaluación y rendimiento de GAug-M

En esta subsección nos centramos en presentar los resultados de los experimentos de evaluación de GAug-M realizados en [37], que también describimos, así como de qué se puede concluir del rendimiento obtenido en ellos.

Para llevar a cabo la evaluación del rendimiento de GAug-M, primero se calcula la matriz de probabilidades de todas las posibles aristas sobre cada uno de los *datasets* (grafos) con los que evaluaremos los modelos, que son los siguientes

- *Cora* y *CiteSeer*, grafos que modelan conjuntos de artículos científicos, y las citas entre ellos.

| | CORA | CITESEER | PPI | BLOGCATALOG | FLICKR | AIR-USA |
|---|---|---|---|---|---|---|
| # Nodes | 2,708 | 3,327 | 10,076 | 5,196 | 7,575 | 1,190 |
| # Edges | 5,278 | 4,552 | 157,213 | 171,743 | 239,738 | 13,599 |
| # Features | 1,433 | 3,703 | 50 | 8,189 | 12,047 | 238 |
| # Classes | 7 | 6 | 121 | 6 | 9 | 4 |
| # Training nodes | 140 | 120 | 1,007 | 519 | 757 | 119 |
| # Validation nodes | 500 | 500 | 2,015 | 1,039 | 1,515 | 238 |
| # Test nodes | 1,000 | 1,000 | 7,054 | 3,638 | 5,303 | 833 |

Figura 7.3: Características de los distintos *datasets* con los que evaluamos GAUG-M. Fuente: [37]

- *Protein-Protein Interactions* (PPI), grafo que modela proteínas y las interacciones entre ellas.
- *Flickr* y *BlogCatalog*, grafos que modelan redes sociales y las relaciones entre sus usuarios
- *Air-USA*, grafo que modela el tráfico aéreo entre aeropuertos de E.E.U.U.

En la Figura 7.3 se pueden apreciar las características de cada *dataset* (número de nodos, de aristas, de *features*, etc.)

A continuación, se entrenan y evalúan los modelos de clasificación de nodos mencionados anteriormente (GCN [15], GSAGE [9], GAT [29] y JK-NET [30]), sobre los grafos originales $\mathcal{G}$ y sobre los grafos modificados por GAUG-M, $\mathcal{G}_m$, utilizando el predictor de aristas GAE [14]. Se realiza una separación de los nodos en distintos conjuntos, un 10 % para el conjunto de entrenamiento, un 20 % para el conjunto de validación y un 70 % para el conjunto de *test*. Comparamos el rendimiento de GAUG-M con el de otros dos métodos de *data augmentation* similares: ADAEDGE [2] y BGCN [36]. Evaluamos el rendimiento calculando la media y la desviación de la métrica micro-F1 a lo largo de 30 entrenamientos de los modelos (*runs*), al mismo tiempo que se usa la librería de *Python Optuna* para escoger los hiperparámetros de los modelos.

En la Figura 7.4 podemos observar una tabla con los resultados obtenidos en [37] con el predictor de aristas GAE, organizados por modelo de clasificación de nodos, y luego método de *data augmentation* (o grafo original) en las filas, y por *datasets* en las columnas. GAUG-M consigue mejoras de rendimiento en con todas las arquitecturas de GNN, más notablemente sobre GAT. Fijándonos en los *datasets*, GAUG-M consigue mejorar el rendimiento en *Cora* y *CiteSeer*, pero la mejora es especialmente grande sobre los datasets *Flickr*, *BlogCatalog* y *Air-USA*. Es interesante que la mejora sea notable en los *datasets* que modelan redes sociales, ya que son en los que esperamos más ruido, procedente de *spammers*, *bots* u otras causas, lo que apoya a la intuición que comentábamos en la motivación teórica (Subsección 7.2.1). En cuanto a la comparación

con el resto de métodos de *data augmentation*, GAug-M consigue, en general, un rendimiento superior a los otros dos métodos, en especial en los *datasets BlogCatalog*, *Flickr*, y *Air-USA*.

| GNN Arch. | Method | CORA | CITESEER | PPI | BLOGC | FLICKR | AIR-USA |
|---|---|---|---|---|---|---|---|
| GCN | Original | 81.6±0.7 | 71.6±0.4 | 43.4±0.2 | 75.0±0.4 | 61.2±0.4 | 56.0±0.8 |
| | +BGCN | 81.2±0.8 | 72.4±0.5 | – | 72.0±2.3 | 52.7±2.8 | 56.5±0.9 |
| | +ADAEDGE | 81.9±0.7 | 72.8±0.7 | 43.6±0.2 | 75.3±0.3 | 61.2±0.5 | 57.2±0.8 |
| | +GAUG-M | 83.5±0.4 | 72.3±0.4 | 43.5±0.2 | **77.6±0.4** | **68.2±0.7** | 61.2±0.5 |
| GSAGE | Original | 81.3±0.5 | 70.6±0.5 | 40.4±0.9 | 73.4±0.4 | 57.4±0.5 | 57.0±0.7 |
| | +BGCN | 80.5±0.1 | 70.8±0.1 | – | 73.2±0.2 | 58.1±0.3 | 53.5±0.3 |
| | +ADAEDGE | 81.5±0.6 | 71.3±0.8 | 41.6±0.8 | 73.6±0.4 | 57.7±0.7 | 57.1±0.5 |
| | +GAUG-M | **83.2±0.4** | 71.2±0.4 | 41.1±1.0 | **77.0±0.4** | **65.2±0.4** | **60.1±0.5** |
| GAT | Original | 81.3±1.1 | 70.5±0.7 | 41.5±0.7 | 63.8±5.2 | 46.9±1.6 | 52.0±1.3 |
| | +BGCN | 80.8±0.8 | 70.8±0.6 | – | 61.4±4.0 | 46.5±1.9 | 54.1±3.2 |
| | +ADAEDGE | 82.0±0.6 | 71.1±0.8 | 42.6±0.9 | 68.2±2.4 | 48.2±1.0 | 54.5±1.9 |
| | +GAUG-M | 82.1±1.0 | 71.5±0.5 | 42.8±0.9 | 70.8±1.0 | **63.7±0.9** | **59.0±0.6** |
| JK-NET | Original | 78.8±1.5 | 67.6±1.8 | 44.1±0.7 | 70.0±0.4 | 56.7±0.4 | 58.2±1.5 |
| | +BGCN | 80.2±0.7 | 69.1±0.5 | – | 65.7±2.2 | 53.6±1.7 | 55.9±0.8 |
| | +ADAEDGE | 80.4±1.4 | 68.9±1.2 | 44.8±0.9 | 70.7±0.4 | 57.0±0.3 | 59.4±1.0 |
| | +GAUG-M | **81.8±0.9** | 68.2±1.4 | 47.4±0.6 | **71.9±0.5** | **65.7±0.8** | 60.2±0.6 |

Figura 7.4: Valores de la métrica micro-F1 (expresada en porcentaje) para distintas combinaciones de modelo-método de *data augmentation*, sobre cada uno de los distintos *datasets*. Fuente: [37]

# Capítulo 8

# Evaluación de GAug-M usando SEAL

En este capítulo explicamos un procedimiento nuevo que he llevado a cabo en este trabajo, que consiste en evaluar el rendimiento de GAug-M combinado con SEAL, bajo las mismas condiciones sobre las que es llevado a cabo en [37] y con los mismos clasificadores de nodos, para comparar los resultados y comprobar si el uso de SEAL supone una mejora sustancial de rendimiento. La motivación de realizar este experimento surge del hecho de que SEAL mejora en rendimiento a muchos de los métodos *state-of-the-art* para predicción de aristas que utilizan *node embeddings*, con lo cual, parece razonable utilizar el mejor predictor de aristas posible a la hora de realizar *data augmentation* con GAug-M. He llevado a cabo la evaluación de la combinación de ambos métodos sobre los mismos datasets del capítulo anterior: *Cora*, *CiteSeer*, *PPI*, *BlogCatalog*, *Flick* y *Air-USA*, y utilizando las mismas arquitecturas de GNN: GCN, GSAGE, GAT y JK-Net, para poder comparar resultados.

Para realizar estos experimentos, he utilizado *scripts* de *python* de dos repositorios de *GitHub* distintos, uno contiene todo lo correspondiente a SEAL[1] ([34]), mientras que otro todo lo correspondiente a GAug-M [2] ([37]). Sin embargo, he modificado algunos de estos *scripts*, incluso creado algunos desde cero, para adaptarlos y poder usarlos de manera combinada en un mismo repositorio creado por mí: SEAL-GAugM. Con lo cual, en este capítulo, describiremos la realización de los experimentos por medio de este repositorio diseñado específicamente para esta tarea. Todos los *scripts* utilizados para su realización y la obtención de resultados pueden consultarse en el mismo.

[1]`https://github.com/facebookresearch/SEAL_OGB`
[2]`https://github.com/zhao-tong/GAug`

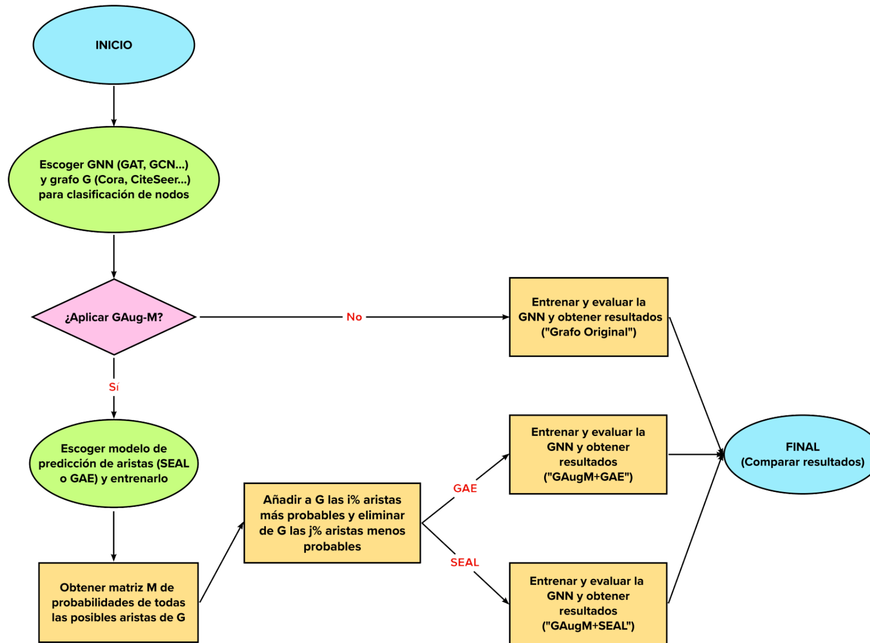


Figura 8.1: *Workflow* de los experimentos cuyos resultados comparamos en la Sección 8.2, de los cuales el correspondiente a la combinación SEAL+GAug-M es nuevo y realizado como parte de este trabajo.

## 8.1. Obtención de la matriz de probabilidades

Para poder evaluar GAug-M sobre un grafo, lo primero que necesitamos es la matriz de probabilidades de todas las posibles aristas. Para obtenerla, utilizamos los siguientes *scripts* de *python*, originalmente del repositorio de SEAL, pero con algunas modificaciones que he realizado, no sólo para entrenar y evaluar el modelo, sino además para realizar los cálculos con todas las posibles parejas de nodos, y para poder ejecutarlo sobre los *datasets* mencionados anteriormente:

1. ***seal_link_pred.py***: *Script* utilizado para entrenar y evaluar el modelo SEAL de acuerdo con los parámetros comentados en la sección 6.3. He modificado ligeramente la casuística inicial de este archivo para que tenga en cuenta la posibilidad de usar algunos *datasets* de los mencionados que no están presentes en [34]. El modelo se entrena a lo largo de 50 épocas, y se escoge el modelo con el AUC más bajo en el conjunto de validación.

2. ***models.py*** e ***utils.py***: El primero de estos scripts contiene las definiciones de las clases correspondientes a los distintos modelos que se pueden entrenar con SEAL, en particular DGCNN, modelo por defecto y que

usaremos en la práctica. Por otro lado, *utils.py* contiene las funciones para tareas específicas que se usan en el resto de *scripts*, como por ejemplo, calcular y almacenar el subgrafo recubridor relativo a una arista, o muestrear las aristas negativas.

Además, incluimos los siguientes *scripts* adicionales, que he creado desde cero durante la realización de este trabajo para la tarea en cuestión:

1. ***custom_ datasets.py***: En este script se incluyen clases correspondientes a los nuevos *datasets*; *Flickr*, *BlogCatalog*, *Air-USA* y *PPI*, de acuerdo con la arquitectura de la librería *PyTorch Geometric*[3], para poder entrenar los modelos sobre ellos usando *seal_ link_ pred.py*.
2. ***inference.py***: Este script se ejecuta una vez que se ha entrenado el modelo, para realizar los cálculos de probabilidad de todas las aristas de un grafo. Algunos de los resultados de *seal_ link_ pred.py* (época del mejor modelo, parámetro $k$, etc.) se utilizan como input manual de este *script*, que calcula todas las probabilidades, las almacena en una matriz de dimensión $n \times n$ (donde n es el número de nodos del grafo), y la guarda en un archivo de formato *pickle*, lista para ser utilizada con el *script* de GAug-M.

Por ejemplo, si queremos entrenar un modelo sobre el grafo *Cora*, con los parámetros por defecto (Sección 6.3), ejecutamos el siguiente comando en la carpeta principal del repositorio:

```
python seal_link_pred.py –dataset Cora –num_hops 2 –use_feature
    –train_node_embedding –save_appendix defaultparams
```

Se puede observar que se proporcionan varios argumentos a la hora de ejecutar el *script*, como por ejemplo, el *dataset* que se va a utilizar o la distancia a tener en cuenta para el subgrafo recubridor (*num_ hops*). Por otra parte, *use_ feature* y *train_ node_ embedding* son necesarios para indicar que el modelo utilice las *features* explícitas y los *node embeddings*, respectivamente. Finalmente, *save_ appendix* indica el nombre de la carpeta en el que se guardarán los resultados. Antes de que comience el entrenamiento, aparece un mensaje informando del valor $k$ que se escoge para la capa *SortPooling*, y del número total de parámetros del modelo:

```
Total number of parameters is 341730
SortPooling k is set to 48
```

[3] `https://pytorch-geometric.readthedocs.io/en/latest/notes/create_dataset.html`

Una vez que el entrenamiento haya completado, el comando devuelve un mensaje similar al siguiente:

Highest Valid: 90.62
Highest Eval Point: 4
Final Test: 91.81

Este mensaje indica, respectivamente, el mejor valor de la métrica AUC en el conjunto de validación de entre los 50 modelos obtenidos después de cada época (*Highest Valid*), la época en la que se ha obtenido (*Highest Eval Point*), y el valor de AUC en el conjunto de *test* para ese modelo. El *script* guarda archivos que almacenan la información de los modelos después de cada época en la carpeta indicada, junto con los mensajes y el comando de input.

A continuación, ejecutamos *inference.py* con el siguiente comando:

python inference.py –dataset Cora –best_run 4 –k 48

Se puede observar que es necesario proporcionar como argumentos el *dataset* sobre el que queremos realizar los cálculos, además del valor de *Highest Eval Point* y el valor $k$, devueltos por *seal_link_pred.py*. Una vez que finalizan los cálculos, el *script* almacena la matriz de probabilidades en la carpeta y formato adecuados para ser utilizada directamente en la siguiente sección sin necesidad de moverla, renombrarla, etc.

## 8.2. Aplicación de GAug-M y evaluación

Una vez que tenemos la matriz de probabilidades, utilizamos los *scripts* originalmente del repositorio *GitHub* de GAug-M para entrenar un modelo de clasificación de nodos sobre el grafo aumentado con las probabilidades. En concreto, utilizamos el *script* ***train_GAugM.py***. Siguiendo el ejemplo anterior, si quisiéramos evaluar el rendimiento sobre *Cora*, ejecutaríamos el siguiente comando:

python train_GaugM.py –dataset Cora –gnn gcn

En este caso, solo es necesario proporcionar como argumentos el *dataset*, y la arquitectura de GNN que se utilizará para la clasificación de nodos. El entrenamiento se produce como se explicó en la Subsección 7.2.3, y una vez que termina la ejecución del *script*, obtenemos el valor medio y la desviación típica de la métrica micro-F1.

En la tabla de la Figura 8.2, mostramos los resultados de los experimentos llevados a cabo de esta manera, y comparamos el rendimiento obtenido mediante

la utilización de SEAL (con los parámetros comentados en la Sección 6.3) con el rendimiento obtenido con *Graph Auto Encoder* en [37]. La tabla se organiza de la misma manera que la de la Figura 7.4; las columnas hacen referencia a distintos *datasets* y las filas dividen primero por arquitectura de GNN, y luego por método de *data augmentation* utilizado (ninguno (grafo original), GAug-M+GAE y GAug-M+SEAL). Omitimos los *datasets Flickr, PPI* y *BlogCatalog* debido a falta de memoria para realizar los cálculos, ya que, al tener demasiadas aristas, los subgrafos recubridores ocupan demasiado espacio.

| **GNN** | **Método** | **Cora** | **CiteSeer** | **Air-USA** |
|---|---|---|---|---|
| GCN | Grafo original | $81.6 \pm 0.7$ | $71.6 \pm 0.4$ | $56.0 \pm 0.8$ |
| | GAug-M+GAE | $\mathbf{83.5 \pm 0.4}$ | $\mathbf{72.3 \pm 0.4}$ | $\mathbf{61.2 \pm 0.5}$ |
| | GAug-M+SEAL | $80.4 \pm 0.7$ | $70.9 \pm 0.2$ | $60.1 \pm 1$ |
| GSage | Grafo original | $81.3 \pm 0.5$ | $70.6 \pm 0.5$ | $57.0 \pm 0.7$ |
| | GAug-M+GAE | $\mathbf{83.2 \pm 0.4}$ | $\mathbf{71.2 \pm 0.4}$ | $60.1 \pm 0.5$ |
| | GAug-M+SEAL | $80.8 \pm 0.5$ | $70.0 \pm 0.5$ | $\mathbf{61.2 \pm 0.7}$ |
| GAT | Grafo original | $81.3 \pm 1.1$ | $70.5 \pm 0.7$ | $52.0 \pm 1.3$ |
| | GAug-M+GAE | $\mathbf{82.1 \pm 1}$ | $\mathbf{71.5 \pm 0.5}$ | $59 \pm 0.6$ |
| | GAug-M+SEAL | $81 \pm 0.7$ | $71.4 \pm 0.8$ | $\mathbf{60.5 \pm 0.8}$ |
| JK-Net | Grafo original | $78.8 \pm 1.5$ | $67.6 \pm 1.8$ | $58.2 \pm 1.5$ |
| | GAug-M+GAE | $\mathbf{81.8 \pm 0.9}$ | $\mathbf{68.2 \pm 1.4}$ | $60.2 \pm 0.6$ |
| | GAug-M+SEAL | $78.9 \pm 1.2$ | $6.2 \pm 4.3$ | $\mathbf{60.4 \pm 1.3}$ |

Figura 8.2: Tabla con los resultados de la métrica micro-F1 de distintas combinaciones de GNN-técnica de *data augmentation* sobre distintos *datasets*

Podemos observar que, a pesar de que SEAL supera en rendimiento a muchos otros métodos de predicción de aristas en grafos [34], su uso con GAug-M no supone un cambio sustancial en el rendimiento, en comparación con el uso de *Graph Auto Encoder* (ya que, en algunos casos, la métrica tiene un valor ligeramente menor, y en otros ligeramente mayor). Esto puede deberse a que, una ligera mejora en el rendimiento del predictor de aristas no se traslada a una mejora en el rendimiento del modelo de predicción de nodos, en comparación con la mejora que supone usar el método de *data augmentation* respecto a no usarlo.

# Capítulo 9

# Conclusiones y trabajos futuros

A lo largo de este trabajo hemos introducido los conceptos básicos sobre grafos y *machine learning*, para, a continuación, estudiar en profundidad las redes neuronales, y en especial, las *Graph Neural Networks*. Una vez explicados los conceptos básicos necesarios, indagamos en el problema concreto de predicción de aristas, y explicamos la metodología SEAL, y mostramos que, empleada junto con una DGCNN, se trata de un modelo *state-of-the-art*, superando en rendimiento a muchos de los otros métodos de predicción de aristas. A continuación, describimos el método GAug-M para realizar *data augmentation* en un grafo a la hora de enfrentarnos a un problema de clasificación de nodos, mostramos el rendimiento obtenido por el mismo en conjunto con un predictor de aristas como VGAE, y observamos cómo supone una mejora sustancial respecto de otros métodos de *data augmentation* aplicables al mismo problema. Finalmente, hemos reproducido los experimentos de [37] para evaluar la combinación de ambos métodos, SEAL y GAug-M, además de crear el repositorio SEAL-GAugM y dar una explicación de su uso para la realización de los experimentos.

Como conclusiones al trabajo, la observación principal es que el rendimiento del predictor de aristas no es el factor más importante a la hora de mejorar el rendimiento de un modelo de clasificación de nodos, sino el método de *data augmentation* en sí, y el hecho de aplicarlo previamente al grafo. Como prueba de ello, observamos que SEAL supera en rendimiento considerablemente a VGAE como predictor de aristas (Figura 6.3), pero que, sin embargo, no hay una diferencia notable entre el rendimiento de las combinaciones GAug-M+SEAL y GAug-M+VGAE (método empleado en [34]) (Figura 8.2).

Entre los posibles trabajos futuros que puedan complementar a estos experimentos, la posibilidad más inmediata es realizar el mismo esquema de experimentos (con SEAL como predictor de aristas), pero con el método de *data augmentation* GAug-O, introducido también en [37]. La idea es comprobar si, en ese caso, el uso de SEAL supone una mejora de rendimiento. Se trata de una variante de GAug-M en la que, en lugar de entrenar el modelo y evaluarlo sobre el grafo modificado $\mathcal{G}_m$ (*Graph Augmentation-Modified*), se modifica el grafo original $\mathcal{G}$ para obtener grafos $\mathcal{G}_i, i = 1, ..., N$, sobre los que se realiza el entrenamiento, para luego evaluar el modelo sobre el grafo original (*Graph Augmentation-Original*). No hemos incluido este posible experimento en el trabajo debido a la complejidad del método GAug-O, que habría supuesto un aumento considerable de la longitud del trabajo.

Otra idea es hacer que SEAL sea más eficiente a la hora de trabajar sobre grafos grandes y a la vez muy densos (que contienen un gran porcentaje de aristas respecto del número máximo posible), ya que extraer subgrafos $h$-recubridores para el entrenamiento puede dar lugar a fallos de tipo OOM (*out of memory*), debido a la cantidad de aristas que contienen, como ocurre por ejemplo en los *datasets Flickr*, *BlogCatalog* o *PPI*, donde surge este problema incluso con $h = 1$. Una idea que se propone en [34] es la de establecer una manera de muestrear cierta cantidad de nodos de los subgrafos recubridores, y considerar el subgrafo inducido por esos nodos, de manera que no se tengan en cuenta todas las aristas del subgrafo recubridor y aliviar un poco la carga de memoria.

# Bibliografía


[1] Edoardo M Airoldi, David M Blei, Stephen E Fienberg, and Eric P Xing. Mixed membership stochastic blockmodels. *Journal of Machine Learning Research*, 9(Sep):1981–2014, 2008.

[2] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34, 2020.

[3] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. URL: `https://www.biostat.wisc.edu/~page/rocpr.pdf`.

[4] Kaize Ding, Huan Liu, Zhe Xu, and Hanghang Tong. Data augmentation for deep graph learning: A survey, 2022. URL: `https://arxiv.org/pdf/2202.08235.pdf`.

[5] Andrew Duncan. Powers of the adjacency matrix and the walk matrix, 2004. URL: `https://www.um.edu.mt/library/oar/bitstream/123456789/24439/1/powers%20of%20the%20adjacency%20matrix.pdf`.

[6] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. *ICML*, 2017.

[7] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. 2016. URL: `https://arxiv.org/pdf/1607.00653.pdf`.

[8] William L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159, 2020.

[9] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *NeurIPS, 2017b*, 2017.

[10] N. Hartsfield and G. Ringel. *Pearls in Graph Theory: A Comprehensive Introduction.* Dover Books on Mathematics. Dover Publications, 2013.

[11] Glen Jeh and Jennifer Widom. Simrank: a measure of structural-context similarity. *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 538–543, 2002.

[12] Glen Jeh and Jennifer Widom. Scaling personalized web search. *Proceedings of the 12th international conference on World Wide Web*, pages 271–279, 2003.

[13] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

[14] Thomas N. Kipf and Max Welling. Variational graph auto-encoders. 2016. URL: `https://dare.uva.nl/personal/search?metis.record.id=547366`.

[15] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *Published as a conference paper at ICLR 2017*, 2017. URL: `https://openreview.net/pdf?id=SJU4ayYgl`.

[16] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. *ICML*, 2019. URL: `https://proceedings.mlr.press/v97/li19d.html`.

[17] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning (Second edition)*. MIT Press, 2018.

[18] Michael Nielsen. *Neural Networks and Deep Learning*. 2019. URL: `http://neuralnetworksanddeeplearning.com/index.html`.

[19] Santiago Pascual. Imagen capa de convolución 1-d. Enlace.

[20] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[21] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification. *ICLR*, 2019.

[22] Sheldon Ross. *Introduction to Probability Models, 11th edition*. Elsevier, 2014.

[23] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[24] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. `doi:10.1109/TNN.2008.2005605`.

[25] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011.

[26] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *JMLR*, 2014.

[27] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. *Proceedings of the 24th International Conference on World Wide Web*, page 1067–1077, 2015.

[28] R.J. Trudeau. *Introduction to Graph Theory*. Dover Books on Mathematics. Dover Pub., 1993.

[29] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. 2017. URL: `https://arxiv.org/pdf/1710.10903.pdf`.

[30] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. 2018. URL: `https://arxiv.org/pdf/1806.03536.pdf`.

[31] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. *KDD 2018*, pages 974–983, 2018.

[32] Manzil Zaheer, Satwik Kottur, Siamak Ravanbhakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J Smola. Deep sets. *NIPS 2017*, 2017. URL: `https://arxiv.org/pdf/1703.06114.pdf`.

[33] Muhan Zhang and Yixin Chen. Weisfeiler-lehman neural machine for link prediction. *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 575–583, 2017.

[34] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*, pages 5165–5175, 2018. URL: `https://proceedings.neurips.cc/paper/2018/file/53f0d7c537d99b3824f0f99d62ea2428-Paper.pdf`.

[35] Muhan Zhang, Zhicheng Hui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. *The Thirty-Second AAAI Conference on Artificial Intelligence*, pages 4438–4445, 2018.

[36] Yingxue Zhang, Soumyasundar Pal, Mark Coates, and Deniz Üstebay. Bayesian graph convolutional neural networks for semi-supervised classification. *The Thirty-Third AAAI Conference on Artificial Intelligence*, 2019.

[37] Tong Zhao, Yozen Liu, Leonardo Neves, Oliver Woodford, Meng Jiang, and Neil Shah. Data augmentation for graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35(12), pages 11015–11023, 2021.