

The background of the entire page is a composite of cosmic images. The top half features a dark blue space filled with numerous small, bright stars. The middle section is a solid teal band where the title and author's name are placed. The bottom half shows a vibrant nebula with swirling clouds of orange, yellow, and green gas, punctuated by several bright stars.

Machine Learning

Fundamentos

Jose M. Saavedra

UNIVERSIDAD DE LOS ANDES, CHILE

[HTTPS://GITHUB.COM/JMSAAVEDRAR/MACHINE_LEARNING](https://github.com/jmsaavedrar/machine_learning)

Jose M. Saavedra 2023

First release, March 2023



Contents

1	Fundamentos de Aprendizaje de Máquinas	7
1.1	Aprendizaje de Máquina (Machine Learning)	9
1.1.1	Modelo General	9
1.1.2	Aprendizaje Supervisado (<i>supervised learning</i>)	11
1.1.3	Aprendizaje Autosupervisado (<i>self-supervised learning</i>)	12
1.1.4	Aprendizaje no Supervisado (<i>unsupervised learning</i>)	12
1.1.5	Aprendizaje por Refuerzo (<i>reinforcement learning</i>)	13
1.2	Conjuntos de Datos	13
1.2.1	Imágenes	14
1.2.2	Textos	14
1.2.3	Videos	14
1.3	Herramientas	14
1.4	Teorema Non-Free Lunch	14
1.5	Algunas Definiciones Importantes	14
1.5.1	Parámetros vs Hiperparámetros	14
1.5.2	Bias vs Varianza	15
1.5.3	Overfitting vs Underfitting	15
2	Modelos Lineales	17
2.1	Regresión Lineal	17
2.1.1	Mínimos Cuadrados (Least Squares)	18
2.1.2	Implementación	19
2.1.3	Ejemplo (Diabetes)	20
2.1.4	Ejemplo (Housing)	22

2.2	Regresión Logística	23
2.2.1	Implementación	26
2.2.2	Ejemplo (Iris dataset)	27
2.3	Modelo Lineal Generalizado GLM	28
2.4	Regresión Multiclase	29
2.4.1	Entrenamiento de Softmax	32
2.4.2	Tarea	36
3	Redes Neuronales	39
3.1	Modelo Biológico	39
3.1.1	Componentes	39
3.2	Perceptron y MLP	40
3.2.1	Interpretación Geométrica	41
3.2.2	Función de Pérdida (<i>Loss</i>)	41
3.2.3	Entrenamiento	42
3.2.4	Implementación	45
3.2.5	Ejemplos	46
3.3	Perceptrón Multicapa (MultiLayer Perceptron)	47
3.3.1	Función de Pérdida	48
3.3.2	Backpropagation	49
3.3.3	Ejemplo	52
3.3.4	Implementación (Keras)	52
3.4	Algunos Términos Nuevos	54
3.5	Modelos Convolucionales	55
3.5.1	Convolución	55
3.5.2	Red Neuronal Convolucional	57
3.5.3	Arquitectura General	58
3.5.4	Capa Convolucional	58
3.5.5	Capa de Pooling (agregación)	59
3.5.6	Capa Densa (MLP)	60
3.5.7	Normalización de Capas	61
3.5.8	Una simple red convolucional	62
3.5.9	Implementación (una simple convnet)	64
3.6	Modelos Atencionales	67
3.7	Modelos Atencionales para Texto	68
3.8	Modelos Atencionales para Imágenes (ViT)	72
3.9	Optimización	74
3.9.1	Gradiente Descendente Estocástico SGD	74
3.9.2	Momentum	74
3.9.3	Momentum Nesterov (NAG)	74
3.9.4	AdaGrad	74
3.9.5	RMSProp	75

3.9.6	Adaptive Momentum (Adam)	75
4	Modelos No-Paramétricos	77
4.1	Espacio de Características	77
4.2	Comparación entre Vectores	78
4.2.1	Funciones de Distancia	78
4.2.2	Funciones de Similitud	79
4.2.3	Relación D_2 y cossim	80
4.2.4	Efecto <i>Bursting</i>	81
4.3	kNN	81
4.3.1	Inferencia	82
4.3.2	Maldición de la Dimensionalidad	82
4.3.3	Ejemplo	84
4.4	Clustering	84
4.4.1	k-Means	85
4.4.2	DBSCAN	87
4.4.3	Evaluación de Modelos de Clustering	91
5	Reducción de la Dimensionalidad	95
5.1	Principal Component Analysis (PCA)	95
5.1.1	Algoritmo PCA	95
5.1.2	Reconstrucción	97
5.1.3	Selección de Características	97
5.1.4	Algoritmo	98
5.1.5	Ejemplo de Aplicación PCA	98
5.2	UMAP	100
5.2.1	Estructura Local	101
5.2.2	Optimización	106
5.2.3	Parámetros UMAP	106



1. Fundamentos de Aprendizaje de Máquinas

Aprendizaje de Máquinas (ML por su nombre en inglés), o también llamado Aprendizaje Automático, es una rama de la **inteligencia artificial** que aprovecha la gran disponibilidad de **datos** existentes para mejorar el **desempeño** de modelos computacionales en una **tarea** determinada. Intencionalmente, he enfatizado los términos *datos*, *desempeño* y *tarea*, pues estos son tres de los componentes claves en un modelo de aprendizaje de máquinas.

Vivimos en medio de una **revolución tecnológica** en el que el desarrollo de modelos computacionales eficientes resulta crítico. Así, las ciencias de la computación (Computer Science) se ha convertido en una disciplina altamente importante en nuestros tiempos, con impacto transversal en casi todas las otras áreas del conocimiento.

Hoy en día, vemos una creciente proliferación de aplicaciones que pueden adaptar, automáticamente, su comportamiento para mejorar su desempeño (*aprendizaje*). El software de detección de rostros de nuestra cámara, el lector de número de patentes de autos en un estacionamiento, el reconocedor de rostros de nuestra computadora, el reconocedor de voz de nuestro celular o el software de recomendación de *Netflix* o *Spotify* son ejemplos comunes que utilizan modelos computacionales basados en aprendizaje. *Es importante observar, que los ejemplos mencionados no parecen sorprender a la generación actual; sin embargo, 20 años atrás era imaginable pensar en el desempeño que los algoritmos involucrados en tales aplicaciones pueden lograr hoy en día.*

Por otro lado, en la última década hemos sido testigos de una transformación digital en casi todos los aspectos de nuestras vidas, salud, negocios, educación, diversión, comunicaciones, entre otros. Lo anterior ha permitido la generación de una inmensa cantidad de datos, que muchos llaman *dataquake*. Esta gran disponibilidad de datos ha disparado el amplio interés en áreas como *machine learning* (ML). Lo anterior, además, ha sido reforzado por significativos avances en el poder computacional, a través del uso de GPUs, TPUs, o la facilidad de acceder a servidores en la nube. En resumen, podemos considerar a los datos como el petróleo de nuestra actualidad, pero necesitamos refinarlos para darles valor, ese es el foco de aprendizaje de máquinas (*machine learning*) y ciencia de datos (*data science*).

Ciencia de datos transforma los datos en valor, mientras que **aprendizaje de máquinas** se enfoca en desarrollar modelos que aprovechen los datos para generar alguna decisión de valor.

Al inicio de esta sección hemos enfatizado lo importante que resulta entender tres conceptos claves en ML: datos, tareas y desempeño. Veamos algo más de detalle al respecto:

- **Datos (dataquake)**: algunos años atrás, los datos eran, simplemente, la entrada a programas para generar alguna salida, eran considerados un recurso pasivo puesto que no participaban en el diseño de los programas (algoritmos). En la actualidad, los datos tienen un comportamiento más activo y guían el diseño de soluciones, a través de los que llamamos aprendizaje de máquinas (*machine learning*).

Según [Alpaydin], los datos han comenzado a guiar las operaciones. Ya no son más los programadores; sino los datos, por sí mismos, los que definen lo qué hay que hacer.

- **Tareas**: existe una diversidad de tareas que pueden ser beneficiadas a través de ML. Estas tareas caen en un rango amplio de disciplinas como: astronomía, medicina, biología, medio ambiente, entre otras. Estas tareas se caracterizan por no poder determinar directamente un algoritmo que las pueda resolver. Por ejemplo, pensemos en la detección de un rostro; nosotros lo hacemos de forma, quizás, inconsciente. Pero, si tratamos de definir un algoritmo para ello será una tarea casi imposible. Sin embargo, podemos coleccionar, en forma relativamente fácil, un conjunto de imágenes de rostros. Estos ejemplos pueden ser aprovechados por un modelo de ML para aprender una función que determine la ocurrencia de un rostro. Un ejemplo de esta aplicación se muestra en la Figura 1.1, en donde, los rostros aparecen rotados o con ciertas oclusiones. Además, el modelo puede reconocer la entidad de las personas que aparecen en la imagen.



Figure 1.1: Ejemplo de una aplicación que usa ML para detectar y reconocer la identidad de los rostros que aparecen en una imagen.

Dado lo anterior, resulta interesante pensar en qué tareas podrían resultar beneficiadas a través de modelos ML. Aquí, algunos ejemplos:

- Entender el sentimiento de comentarios en Twitter o RSS.

- Diagnosticar si cierto paciente tiene o no cierta enfermedad.
- Determinar si un correo electrónico es *spam*.
- Determinar el tipo de música que le gusta a un usuario de Spotify.
- *¿Se te ocurre algunas otras?*
- **Desempeño:** medimos el comportamiento de un modelo en una tarea determinada a través de ciertas métricas de desempeño. La métrica a utilizar dependerá de la tarea particular en la que estamos interesados. En este curso, haremos una revisión de diversas métrica para diversos contextos.

Una importante característica de los datos que tenemos ahora es su naturaleza multimodal. En efecto, los datos pueden venir en forma de texto, imágenes o videos, audio, gráficas, entre otras. Un desafío de ML es combinar y aprovechar apropiadamente tales modalidades para aumentar el desempeño de la tarea subyacente. Por ejemplo, los productos de un eCommerce están representados por textos, imágenes, comentarios, entre otras cosas, que podemos aprovechar para mejorar los motores de búsqueda de productos.

En las última década hemos visto importantes avances en los modelos de ML. Uno de los más significativos avances es el florecimiento de **deep-learning**, tema sobre el que discutiremos más adelante.

1.1 Aprendizaje de Máquina (Machine Learning)

Aprendizaje de Máquinas es un subcampo de **inteligencia artificial** cuyo objetivo es el desarrollo de modelos computacionales capaces de aprender a partir de datos disponibles, para luego realizar predicciones sobre entradas no vistas. Lo que se trata de aprender son patrones, estructuras ocultas o regularidades de los datos, a través de modelos formados por algoritmos y estructuras apropiadas llamados modelos de aprendizaje (*learning models*).

El punto inicial es contar con un **conjunto de datos disponible (experiencia)**. Existen diversos modelos de ML que son aplicados dependiendo del tipo de problema que se trata de resolver. En este curso, revisaremos una diversidad de estos modelos.

La Figura 1.2 muestra cómo se ubica ML en el contexto de inteligencia artificial (AI) y la relación con otras áreas involucradas. Por ejemplo, podemos notar que *deep-learning* está dentro de los modelos de redes neuronales, que caen dentro de aprendizaje de máquinas, y esta a su vez está dentro de inteligencia artificial.

1.1.1 Modelo General

Un modelo ML se define bajo una tarea específica, como aquellas listadas en la sección anterior. Dada la tarea, el primer paso es definir el conjunto de datos (Γ), que representa la experiencia, junto con las métricas de desempeño a utilizar. El siguiente paso es definir un modelo ML que corresponda con la tarea a tratar y los datos a utilizar. En efecto, no es lo mismo trabajar con textos, con audios o con imágenes. Dado lo anterior, procedemos a entrenar (proceso de aprendizaje) el modelo, de modo que este obtenga el desempeño deseado. La Figura 1.3 ilustra un modelo general de aprendizaje de máquinas. Aquí, existe un modelo que recibe los datos de entrenamiento y genera ciertas salidas que son luego evaluadas por un componente de evaluación de desempeño. El resultado de este último será utilizado para ajustar (*fitting*) el modelo. El proceso de ajuste del modelo es conocido como *entrenamiento* del modelo.

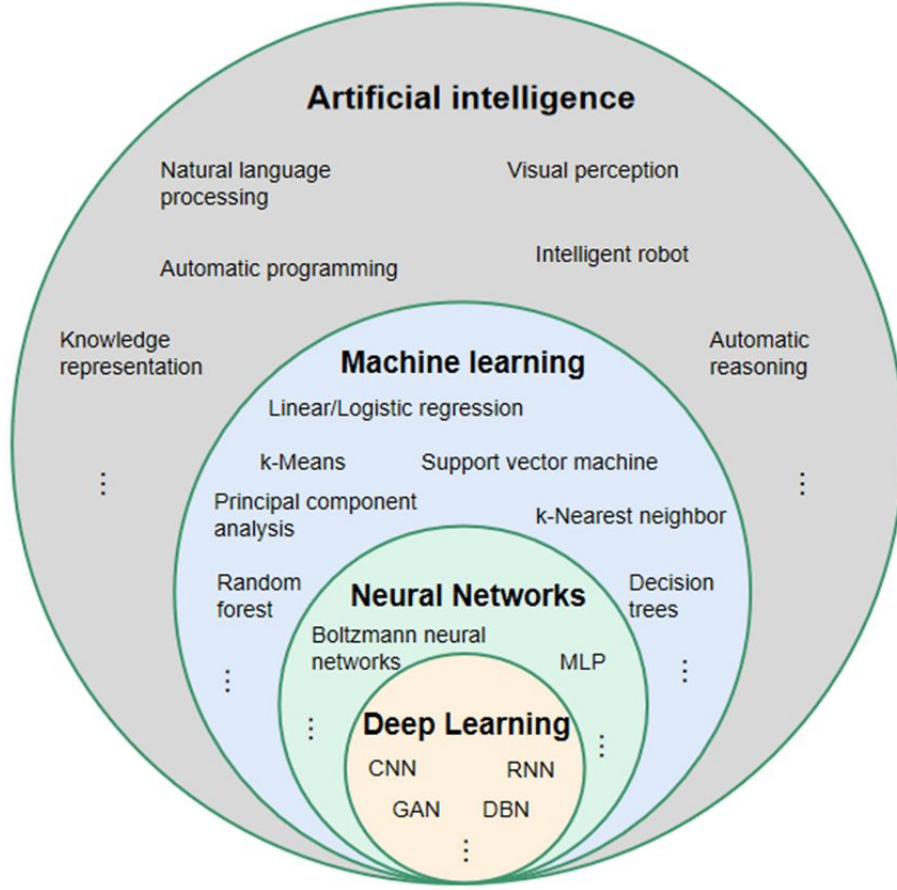


Figure 1.2: Esquema de la relación entre aprendizaje de máquina (*machine learning*) y otras áreas de inteligencia artificial.

En términos formales el objetivo de ML es generar una función $\mathcal{F}_{\Phi} : \mathbb{R}^d \rightarrow \mathbb{R}^m$, de modo que bajo ciertas entradas, en un espacio d -dimensional, este genere salidas deseadas en otro espacio m -dimensional. Esta función puede ser expresada como:

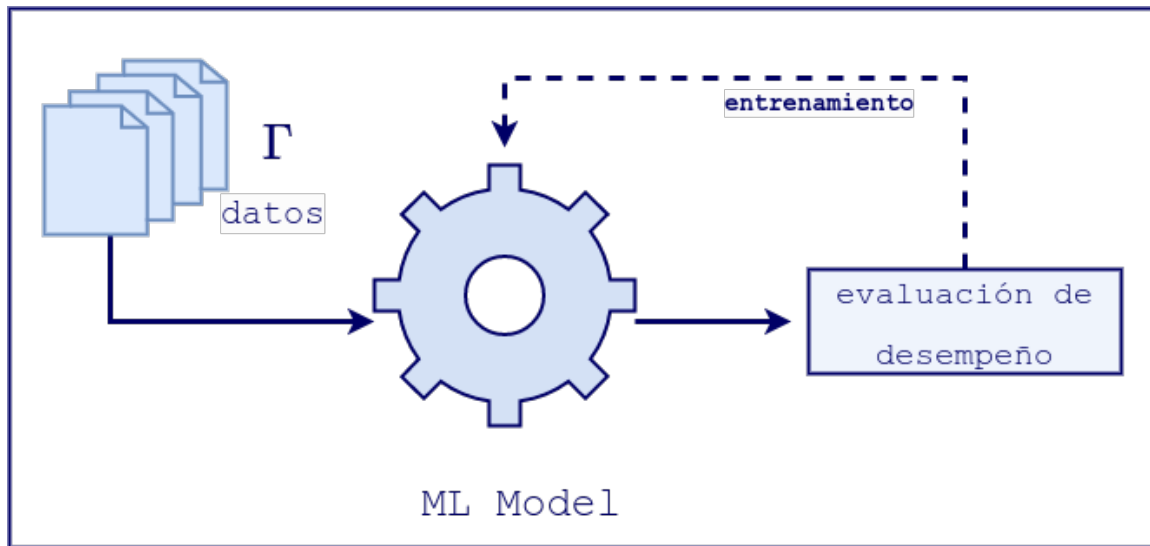
$$\mathcal{F}_{\Phi}(x) = y \quad (1.1)$$

donde $x, y \in \Gamma$ y $x \in \mathbb{R}^d, y \in \mathbb{R}^m$. Además, Φ representa los parámetros de $\mathcal{F}$, que se van ajustando durante el proceso de entrenamiento.

Con los avances en modelos de ML, las entradas y salidas de un modelo de ML van más allá de representaciones vectoriales. Por ejemplo, podemos tener modelos ML que acepten como entrada una imagen y generen un texto. Así, para facilitar nuestra primera aproximación a estos conceptos, extenderemos el concepto de vector (1-D) al de **tensor** que permite representar elementos de mayor dimensionalidad.

Definition 1.1.1 Un **tensor** es un objeto algebraico que describe la relación entre diferentes objetos algebraicos como escalares, vectores u otros tensores.

Un tensor puede ser representado como un arreglo (array), generalmente, multidimensional.



Tarea

Figure 1.3: Esquema general de un modelo de aprendizaje de máquinas (ML). Notemos el proceso cíclico que pasa a través del componente de evaluación de desempeño.

La dimensionalidad del arreglo indica su **grado** (*rank*), por ejemplo (usando Numpy de Python):

- Un escalar $x \in \mathbb{R}$ es un tensor de grado 0.

$x = 16$

- Un vector $v \in \mathbb{R}^d$ es un tensor de grado 1. En este caso, v tiene dimensionalidad igual a 1, con d componentes.

$v = [2, 4, 6, 8]$

- Una matriz $M \in (\mathbb{R}^c)^r$ es un tensor de grado 2. En esta caso M es una matriz de r filas por c columnas.

$v = \begin{bmatrix} [1, 2], \\ [3, 4], \\ [5, 6] \end{bmatrix} \quad (r = 3 \text{ y } c = 2)$

- ¿Podrías dar un ejemplo de tensor de grado 3?

Definition 1.1.2 Aprendizaje es el proceso por el cual adquirimos conocimiento a través de la experiencia para realizar algún arte u oficio.

Desde el punto de vista computacional, **aprendizaje** es el proceso por el cual un modelo aprende a realizar cierta tarea dado un conjunto de datos que utiliza como ejemplos.

1.1.2 Aprendizaje Supervisado (*supervised learning*)

El enfoque de aprendizaje más común en ML es el supervisado. Aquí, definimos el conjunto de datos como $\Gamma = \{(x_i, y_i)\}_{i=1}^N$, donde $x_i \in \mathbb{R}^d$ e $y_i \in \mathbb{R}^m$. Aquí x_i representa un vector de

características de tamaño d e y_i es la salida deseada u objetivo (*target output*). En general la salida $y_i \in \mathbb{R}^m$ y, en particular, $y_i \in \mathbb{R}$.

La definición anterior cubre modelos supervisados como:

- Cuando la entrada es representada por pares o tuplas. Por ejemplo, el modelo de aprendizaje de métricas (*metric learning*).
- Cuando una entrada puede ser etiquetada con varias clases.
- Cuando la salida es un vector multidimensional en el dominio de los reales. Por ejemplo, como cuando necesitamos determinar la ubicación de un objeto a través de un rectángulo definido por cuatro valores (*bounding box*).

Así, cuando los valores de salida no están establecidos en un dominio discreto hablamos de un **modelo de regresión**.

El término supervisado viene del hecho que es necesario tener la respuesta de salida del modelo ML subyacente. De esta manera, el aprendizaje va evaluando el desempeño del modelo y utiliza el error para ajustar los parámetros del modelo. La tarea más común de este tipo de modelos es el de clasificación en el que $y_i \in \{0, \dots, k-1\}$. Así, Dado Γ , un modelo supervisado es entrenado de modo que este se ajuste al conjunto de k clases establecido en los datos de entrenamiento.

Clasificación

En un problema de clasificación, el espacio de salida es un conjunto de k etiquetas que representan k clases diferentes. Aunque es posible que una instancia se asocie a múltiples clases, por ahora, consideraremos al problema de clasificación como un problema de clases excluyentes. Esto es, una entrada tiene asociada una sola clase de las k posibles.

En contextos prácticos, conseguir datos etiquetados puede representar un gran problema que hace prohibitivo contar con este tipo de modelos en diversos contextos. Así, existen enfoques que tratan de reducir la cantidad necesaria de datos etiquetados. En este grupo caen los modelos autosupervisados (*self-supervised*) y no-supervisados (*unsupervised*); e incluso también se habla de modelos débilmente supervisados (*weakly-supervised*).

1.1.3 Aprendizaje Autosupervisado (*self-supervised learning*)

Un modelo autosupervisado es similar a uno supervisado desde el punto de vista de la arquitectura, la diferencia radica en que las etiquetas no necesitan ser definidas a-priori. Para este fin se utilizan estrategias que permiten inferir etiquetas automáticamente.

La gran ventaja de esta estrategia es que podemos generar modelos entrenados que se adaptan a contextos en donde es casi imposible contar con datos etiquetados.

1.1.4 Aprendizaje no Supervisado (*unsupervised learning*)

A diferencia de las dos estrategia anteriores, el enfoque no supervisado no usa etiquetas. Este modelo trata de entender el sentido de los datos sobre una colección de vectores no etiquetados $\Gamma = \{x_i\}_{i=1}^N$. Así, el modelo toma un vector de característica $x \in \mathbb{R}^d$ como entrada para transformarlo en otro vector o en algún valor útil para resolver otro problema. Por ejemplo, en *clustering*, un modelo retorna un *id* del grupo o clúster al cuál pertenece la entrada. Por ejemplo, en el problema de **reducción de la dimensionalidad** un modelo no-supervisado retorna otro vector de menor dimensionalidad.

1.1.5 Aprendizaje por Refuerzo (*reinforcement learning*)

Este enfoque es aplicado en contextos dinámicos, en el que el modelo es capaz de percibir el estado del ambiente en el que este se desenvuelve y generar acciones que le permita cambiar de estado con la intención de maximizar un desempeño. Cada posible acción tiene asociada una recompensa, siendo el objetivo aprender una estrategia (*policy*) que maximice la recompensa total.

El aprendizaje por refuerzo resuelve un tipo de problema en el que la toma de decisiones es secuencial, representada por acciones que modifican el estado y sobre el que hay que volver a tomar acciones. Así una estrategia (*policy*) toma un vector que representa el estado del ambiente y genera una acción “óptima” bajo ese estado. Una acción es óptima si maximiza la recompensa promedio esperada.

Algunos ejemplos de dominio donde el aprendizaje por refuerzo es aplicables son: videojuegos, robótica, administración de recursos y logística.

1.2 Conjuntos de Datos

Los datos representan el insumo más importante en ML. Afortunadamente, nos encontramos en una generación con gran proliferación de ellos. Esto debido a un mundo basado en la digitalización, con uso masivo de Internet, redes sociales y dispositivos móviles.

Por lo anterior, es indispensable que antes de intentar resolver un problema a través de ML, nos aseguremos que podemos contar con los siguientes conjuntos de datos:

- **Conjunto de entrenamiento (training dataset):** corresponde al conjunto de datos que se utiliza para ajustar los parámetros de un modelo.
- **Conjunto de evaluación (testing dataset):** es el conjunto de datos con datos desconocidos, que sirve para evaluar el desempeño del modelo entrenado. En ambientes reales, el equipo de ML no tiene acceso a los datos de evaluación, pues estos datos son resguardados por el cliente.
- **Conjunto de validación (validation dataset):** dado que el equipo ML no tiene acceso a los datos de evaluación, es una buena estrategia considerar un conjunto de datos desconocidos que se aproximen a los datos de evaluación. Este conjunto de datos es conocido como conjunto de validación. Normalmente, este conjunto de datos nos da información para ajustar los *hiperparámetros* del modelo subyacente.

Los avances en ML han dependido directamente de la disponibilidad de datos, en sus diversas modalidades. Así, la comunidad involucrada ha venido liberando diferentes conjuntos de datos que son utilizados en la investigación como en el desarrollo de soluciones industriales. Por ejemplo, conjuntos como **Imagenet**¹ o Coco han favorecido el avances en visión por computadora. Asimismo, repositorio como Kaggle² o HuggingFace <https://huggingface.co/> han favorecido el acceso a una diversidad de datos.

A continuación se listan algunos de los conjuntos de datos más populares en el ámbito de ML.

¹<https://www.image-net.org/>

²<https://www.kaggle.com/>

1.2.1 Imágenes

- MNIST <http://yann.lecun.com/exdb/mnist/>
- CIFAR <https://www.cs.toronto.edu/~kriz/cifar.html>
- ImageNet <https://www.image-net.org/>
- COCO <https://cocodataset.org/>
- OpenImages <https://storage.googleapis.com/openimages/web/index.html>
- Quickdraw <https://github.com/googlecreativelab/quickdraw-dataset>

1.2.2 Textos

- Sentiment140 <http://help.sentiment140.com/for-students/>
- Yelp <https://www.yelp.com/dataset>

1.2.3 Videos

- UCF101 <https://www.crcv.ucf.edu/data/UCF101.php>
- HMDB51 <https://datasets.activeloop.ai/docs/ml/datasets/hmdb51-dataset/>
- Kinetics <https://www.deepmind.com/open-source/kinetics>

1.3 Herramientas

- Python + Numpy
- Scikit-learn
- Tensorflow
- PyTorch

1.4 Teorema Non-Free Lunch

Este teorema indica que no existe **el mejor modelo** para todos los problemas. En general, todos los modelos resultan ser malos para todos los problemas, pero existen modelos útiles para casos particulares. La razón es que el conjunto de suposiciones (*inductive bias*) que funciona bien en un dominio, puede funcionar muy mal en otro.

El objetivo de ML es explotar estas peculiaridades o regularidades de los datos en un dominio particular.

1.5 Algunas Definiciones Importantes

1.5.1 Parámetros vs Hiperparámetros

Los parámetros definen el modelo de aprendizaje (o la función predictora). Los parámetros son modificados directamente por el algoritmo de aprendizaje durante el proceso de entrenamiento. Así, el objetivo del proceso de aprendizaje (entrenamiento) es encontrar los valores de los parámetros que hagan que el modelo funcione correctamente ante los datos de validación, más que antes los datos entrenamiento.

Por otro lado, los hiperparámetros representan propiedades del algoritmo de aprendizaje que se deben definir antes de empezar el entrenamiento. En general, los hiperparámetros suelen estar asociados a un valor numérico, aunque no siempre es el caso. Estos valores afectan el modo en el que el algoritmo de aprendizaje trabaja. Una diferencia importante

con los parámetros, es que los hiperparámetros no son aprendibles, es decir, no modifican su valor durante el entrenamiento.

1.5.2 Bias vs Varianza

Bias es la diferencia entre la predicción promedio de nuestro modelo con respecto al valor correcto. Un modelo con alto **bias** produce alto error en el conjunto de datos de entrenamiento como en el de validación. Un modelo con alto **bias** causa que el algoritmo pierda relaciones relevantes entre las entradas y las salidas. Este tipo de problema genera un **modelo subajustado** (*underfitting*).

Varianza mide la sensibilidad con respecto a pequeñas fluctuaciones en los datos de entrenamiento. Una consecuencia de este problema es la alta diferencia en el desempeño del modelo frente a datos de entrenamiento y validación. Un modelo con alta varianza genera un modelo sobreajustado a los datos de entrenamiento, lo que produce un pobre nivel de generalización a datos no vistos. Así, el modelo se desempeña muy bien en datos de entrenamiento, pero tiene un desempeño pobre en datos de validación o evaluación.

La Figura 1.4 muestra la relación entre bias y varianza en modelos de aprendizaje de máquinas.

1.5.3 Overfitting vs Underfitting

En el desarrollo de modelos de aprendizaje de máquinas es muy importante entender cómo el modelo se va ajustando a los datos. Así, existen dos casos extremos y no deseables: sobreajuste (*overfitting*) y subajuste (*underfitting*).

Un modelo presenta subajuste cuando este produce un alto error en ambos conjuntos de datos, datos de entrenamiento y validación. Por lo general, la causa de este problema es la elección incorrecta del modelo o arquitectura ML, que resulta insuficiente para representar la relación existente entre los datos.

Por otro lado, tenemos el sobreajuste (*overfitting*), uno de los problemas más frecuentes en ML. Aquí, los datos logran alto desempeño en el conjunto de datos de entrenamiento pero un desempeño muy bajo en datos de evaluación o validación. Este problema expresa una **débil capacidad de generalización** del modelo subyacente. La Figura 1.5 ilustra el comportamiento de un modelo sobreajustado y subajustado.

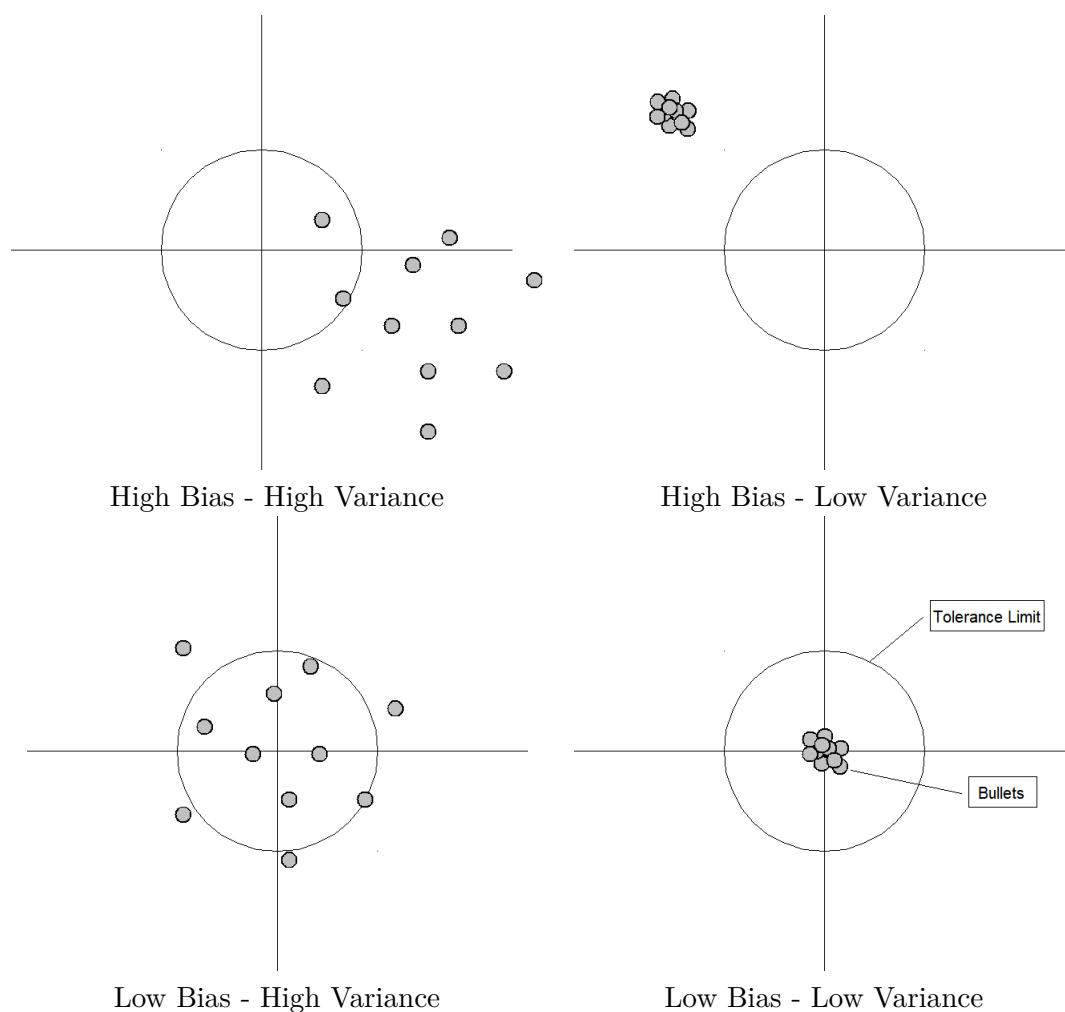


Figure 1.4: Relación entre Bias y Varianza en los modelos de aprendizaje de máquinas.

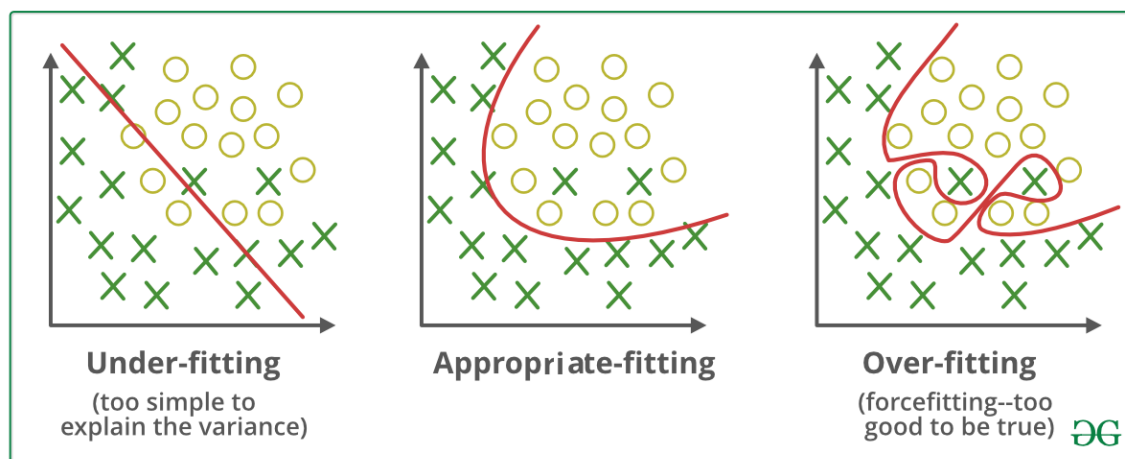


Figure 1.5: Ilustraciones que muestra cuándo un modelo está sobreajustado (overfitting) o subajustado (underfitting).

2. Modelos Lineales

Un modelo es llamado lineal si la relación entre la salida y la entrada es modelada como una función lineal. Es decir, la salida es representada como una combinación lineal de las entradas. Por lo tanto, el problema consiste en determinar, eficientemente, cómo realizar dicha combinación.

2.1 Regresión Lineal

Esta es una herramienta que modela la relación de un conjunto de variables independientes (entrada), definidas por x , y una variable dependiente (salida), definida por y . Así la idea es generar un modelo que infiera una salida real a través de la combinación lineal entre Φ , los parámetros que definen al modelo, y x , la entrada. Ambos son representados por vectores en un espacio d -dimensional.

Formalmente, definimos $\mathcal{F}_\Phi(\cdot)$ (ver Eq. 2.1) al modelo lineal parametrizado por Φ . Este modelo recibe como entrada $x \in \mathbb{R}^d$.

$$\mathcal{F}_\Phi(x) = \phi_0 + x_1\phi_1 + x_2\phi_2 \cdots x_d\phi_d \quad (2.1)$$

$$\mathcal{F}_\Phi(x) = \Phi^T x + \phi_0 \quad (2.2)$$

Sin embargo, es más conveniente incorporar ϕ_0 dentro de Φ y entender x_0 como 1, de modo que Φ y x queden en un espacio de dimensionalidad $d+1$. Así, la función predictora $\mathcal{F}(\cdot)$ queda expresada como:

$$\mathcal{F}_\Phi(x) = \Phi^T x \quad (2.3)$$

donde ϕ_0 es llamado *bias*. De este modo, la función predictora queda representada como el producto punto entre Φ y x

2.1.1 Mínimos Cuadrados (Least Squares)

Una de las formas más simples de medir el error entre las predicciones y los valores deseados es a través de la función **Least Squares** (mínimos cuadrados), definida por:

$$\mathcal{L}(\mathcal{F}_\Phi) = \frac{1}{2} \sum_{i=1}^N (\Phi^T \mathbf{x}_i - y_i)^2 \quad (2.4)$$

El objetivo es determinar el conjunto de parámetros Φ con el menor valor $\mathcal{L}(\Phi)$. Por lo tanto definimos la función objetivo como:

$$\mathbb{J}_\Phi = \operatorname{argmin}_\Phi \mathcal{L}(\Phi) \quad (2.5)$$

Regresión Lineal En regresión lineal, la función de costo $\mathbb{J}_\Phi$ es una **función convexa**, lo que significa que tiene un único mínimo global (no existen mínimos locales). Así, la forma de esta función es la de un *bowl*.

Para resolver el problema anterior, primero calcularemos el gradiente de la función objetivo. Para facilitar este proceso haremos un cambio de variable de modo que $z_i = (\Phi^T \mathbf{x}_i - y_i)$. Así:

$$\mathcal{L}(\mathcal{F}_\Phi) = \frac{1}{2} \sum_{i=1}^N (z_i)^2 \quad (2.6)$$

Ahora calculamos el gradiente $\nabla_\Phi \mathcal{L}(\mathcal{F}_\Phi)$ de la siguiente manera:

$$\nabla_\Phi \frac{1}{2} \sum_{i=1}^N (z_i)^2 = \frac{1}{2} \sum_{i=1}^N \nabla_\Phi z_i^2 \quad (2.7)$$

$$= \frac{1}{2} \sum_{i=1}^N \nabla_\Phi z_i^2 \quad (2.8)$$

$$= \frac{1}{2} \sum_{i=1}^N \nabla_{z_i} z_i^2 \nabla_\Phi z_i \quad (2.9)$$

$$= \frac{2}{2} \sum_{i=1}^N z_i \nabla_\Phi (\Phi^T \mathbf{x}_i - y_i) \quad (2.10)$$

$$= \sum_{i=1}^N z_i \mathbf{x}_i^T \quad (2.11)$$

$$= \sum_{i=1}^N (\Phi^T \mathbf{x}_i - y_i) \mathbf{x}_i^T \quad (2.12)$$

$$= \Phi^T \sum_{i=1}^N \mathbf{x}_i \mathbf{x}_i^T - \sum_{i=1}^N y_i \mathbf{x}_i^T \quad (2.13)$$

La expresión anterior se puede formular en términos matriciales. Para este fin, definimos $\mathbf{X}_{N \times (d+1)}$ como la matriz de datos, de tamaño $N \times (d+1)$, donde N es la cantidad de datos

de entrenamiento y d , la dimensionalidad asociada. Además, definimos $\mathbf{Y}_{N \times 1}$ como un vector de N valores que representa las salidas esperadas para cada fila de $\mathbf{X}$.

$$\mathbf{X}_{N \times (d+1)} = \begin{pmatrix} 1 & x_{1,1} & \cdots & x_{1,d} \\ \vdots & \vdots & \cdots & \vdots \\ 1 & x_{N,1} & \cdots & x_{N,d} \end{pmatrix} \quad \mathbf{Y}_{N \times 1} = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix} \quad (2.14)$$

Ahora, con $\mathbf{X}$ y $\mathbf{Y}$ podemos establecer las siguientes equivalencias:

- $\Phi^T \sum_{i=1}^N \mathbf{x}_i \mathbf{x}_i^T = (\mathbf{X}^T \mathbf{X}) \Phi$
- $\sum_{i=1}^N y_i \mathbf{x}_i^T = \mathbf{X}^T \mathbf{Y}$

Por lo tanto, el gradiente de Eq. 2.7 puede quedar expresado como:

$$\nabla_{\Phi} \frac{1}{2} \sum_{i=1}^N (\Phi^T \mathbf{x}_i - y_i)^2 = (\mathbf{X}^T \mathbf{X}) \Phi - \mathbf{X}^T \mathbf{Y} \quad (2.15)$$

Ahora, igualemos el gradiente a cero para determinar los parámetros Φ óptimos, aquellos que representan el mínimo de la función $\mathbb{J}_{\Phi}$:

$$(\mathbf{X}^T \mathbf{X}) \Phi = \mathbf{X}^T \mathbf{Y} \quad (2.16)$$

que se puede resolver como un sistema lineal de la forma $Ax = b$. En este caso tenemos que $A = \mathbf{X}^T \mathbf{X}$, $x = \Phi$ y $b = \mathbf{X}^T \mathbf{Y}$. Por lo tanto:

$$\Phi = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y} \quad (2.17)$$

2.1.2 Implementación

Crearemos una clase que permita generar modelos de regresión lineal. Esta clase contendrá dos importante métodos. El primero, *fit*, será el responsable del entrenamiento, generando los coeficientes del modelo lineal. El segundo, *predict*, será el encargado de realizar las predicciones. La implementación de esta clase se muestra en el Código 2.1.

Código 2.1: Regresión Lineal

```

1  import numpy as np
2
3  class LinearRegression :
4      def __init__(self):
5          self.coeff = []
6
7      def fit(self, X_train: np.ndarray, Y : np.ndarray )-> np.ndarray :
8          ones = np.ones((X_train.shape[0],1))
9          X = np.concatenate([ones, X_train], axis = 1)
10         tX = np.transpose(X)
11         invX = np.linalg.pinv(np.matmul(tX, X))
12         self.coeff = np.matmul(np.matmul(invX, tX), Y)
13         return self.coeff
14

```

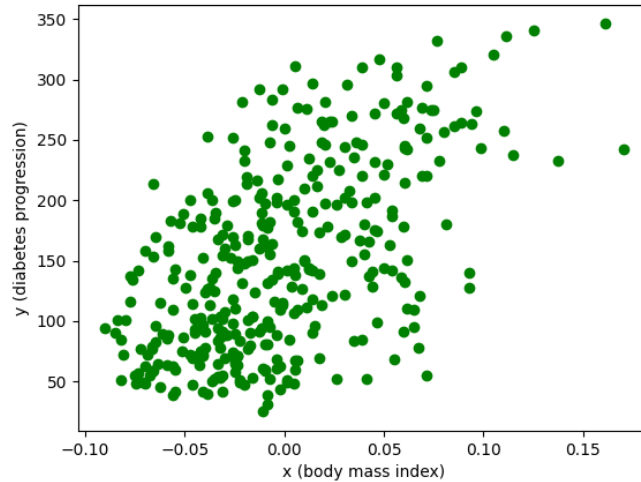



Figure 2.1: Gráfico de dispersión entre índice de masa corporal (bmi) y progresión de la enfermedad.

```

15     def predict(self, X: np.ndarray):
16         ones = np.ones((X.shape[0], 1))
17         X = np.concatenate([ones, X], axis = 1)
18         print(X.shape)
19         print(self.coeff.shape)
20         y_pred = np.matmul(X, self.coeff)
21         return y_pred

```

2.1.3 Ejemplo (Diabetes)

Utilizaremos el conjunto de datos **diabetes** ¹. Este conjunto describe a 442 pacientes con diabetes a través de 10 variables que influyen la progresión de la enfermedad en un año. Estas variables son:

- x_1 : Age
- x_2 : Sex
- x_3 : Body mass index
- x_4 : Average blood pressure
- x_5 : S1(tc), total serum cholesterol
- x_6 : S2(ldl), low-density lipoproteins
- x_7 : S3(hdl), high-density lipoproteins
- x_8 : S4(tch), total cholesterol / HDL
- x_9 : S5(ltg), possibly log of serum triglycerides level
- x_{10} : S6(glu), blood sugar level

Código 2.2: Ejemplo 1

```

1 #diabetes
2 import linear_models.linear as linear

```

¹https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_diabetes.html

```

3 import metrics.metrics as metrics
4 import matplotlib.pyplot as plt
5 import numpy as np
6 import sklearn.datasets as datasets
7
8 # reading data
9 X, y = datasets.load_diabetes(return_X_y=True)
10 # Use only one feature (bmi index = 2)
11 X = X[:,2:3]
12 print(X.shape)
13
14 # Split the data into training/testing sets
15 # training 80% testing 20%
16 n = X.shape[0]
17 n_test = int(np rint(0.2 * n))
18
19 # random sort
20 idx = np.random.permutation(n)
21 X = X[idx]
22 y = y[idx]
23
24 X_train = X[:-n_test]
25 X_test = X[-n_test:]
26 y_train = y[:-n_test]
27 y_test = y[-n_test:]
28
29
30 model = linear.LinearRegression()
31 coeff = model.fit(X_train, y_train)
32 y_pred = model.predict(X_test)
33 v = metrics.r2score(y_test, y_pred)
34 print(coeff)
35 print('v:{'}.format(v))
36 #ploting
37 plt.scatter(X_train, y_train, color="gray")
38 line_x = np.array([[np.min(X)], [np.max(X)]] )
39 line_y = model.predict(line_x)
40 plt.plot(line_x, line_y, marker = 'o', color = 'red' )
41 plt.ylabel('y (diabetes progression)')
42 plt.xlabel('x (body mass index)')
43 plt.show()

```

Coefficiente de Determinación (R^2 -score)

Una de las métricas más comunes en los modelos de regresión lineal es el coeficiente de determinación (R^2 -score). Esta métrica es representada por un escalar que indica qué tan bien los puntos se ajustan a una línea (en el caso lineal).

$$R^2 = 1 - \frac{\sigma_r^2}{\sigma^2} \quad (2.18)$$

donde σ_r^2 es la varianza residual, medida como la suma de los cuadrados de la diferencias entre la salida inferida y la correcta, esto es:

$$\sigma_r^2 = \sum_{i=1}^N (y_i - \hat{y}_i)^2 \quad (2.19)$$

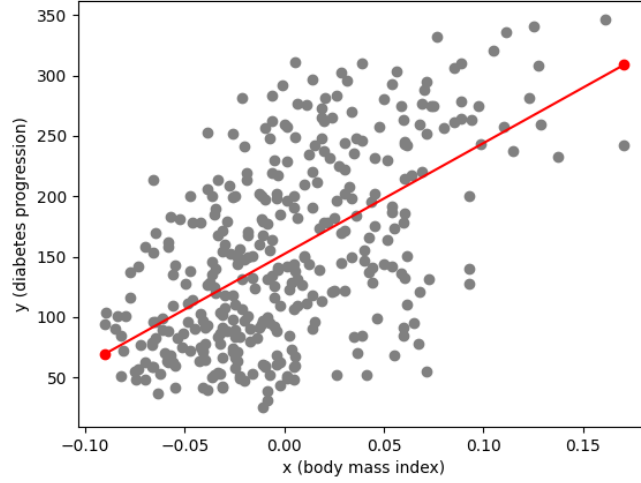


Figure 2.2: Representación gráfica del modelo de regresión lineal aprendido. La línea roja representa el modelo lineal.

y σ^2 es la varianza de las salidas medida como la suma de los cuadrados de la diferencia entre los valores de salida y la media correspondiente. Es decir:

$$\sigma^2 = \sum_{i=1}^N (y_i - \mu_y)^2 \quad (2.20)$$

Es importante notar que en las definiciones anteriores no hemos dividido las diferencias cuadradas por N (que es lo natural en la definición de la varianza), dado que la división de las varianzas la eliminaría.

Así, $R^2 \in [0, 1]$, de modo que un valor 0 indica que el modelo no predice la salida deseada y 1 indica que el modelo predice perfectamente la salida. Un valor entre 0 y 1 indicará que el modelo predice parcialmente las salidas correctas.

R^2 -score utiliza σ^2 como un factor de normalización con respecto a la variación de los datos de salida. Así, sean A y B dos conjunto de datos, donde A presenta mucho mayor varianza con respecto a B . La normalización por σ^2 permite que un error pequeño tenga más impacto en B que en A .

2.1.4 Ejemplo (Housing)

Otro ejemplo interesante es predecir el valor de un predio en base a características como áreas, ubicación, número de habitaciones, entre otras. Un ejemplo de la relación entre área y precio del conjunto de datos de <https://www.kaggle.com/datasets/ashydv/housing-dataset> se muestra en Figura 2.3

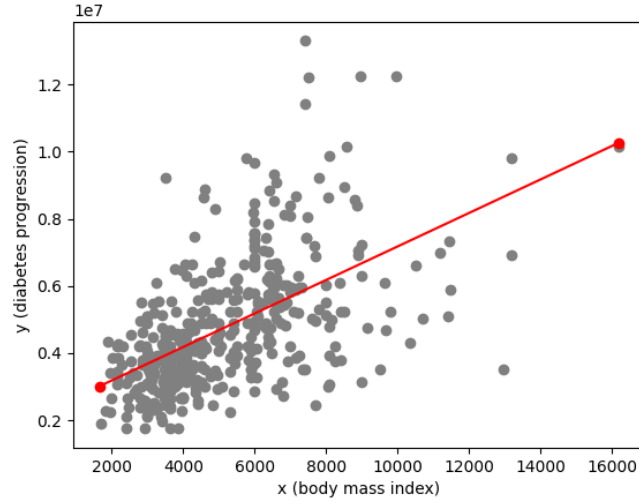


Figure 2.3: Representación gráfica del modelo de regresión lineal aprendido para los datos de *housing*.

2.2 Regresión Logística

Esta es una extensión del modelo de regresión lineal para tratar problemas de clasificación, donde la salida y está asociada a valores discretos. Para facilitar la comprensión, asumiremos un problema de clasificación binaria ($y \in \{0, 1\}$).

Para tratar el problema anterior, aplicamos una función que restringe la salida del modelo $\mathcal{F}_\Phi(x)$ a $[0, 1]$. Esta función es la *función logística* o *función sigmoideal* (ver Figura 2.4) definida por:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (2.21)$$

cuya derivada $f'(x) = f(x)(1 - f(x))$ (es un buen ejercicio deducirla).

Así, la función predictora $\mathcal{F}_\Phi(\cdot)$ queda definida por la Eq. 2.22.

$$\mathcal{F}_\Phi(x) = f(\Phi^T x) \quad (2.22)$$

$$= \frac{1}{1 + e^{-\Phi^T x}} \quad (2.23)$$

Dadas las definiciones anteriores, el siguiente paso es determinar cómo encontrar Φ . Para ello debemos recordar algunos temas elementales de Cálculo y Probabilidades. Así, interpretaremos el modelo de clasificación bajo un enfoque probabilístico, de modo que Φ sea estimado a través de un enfoque conocido como **Maximum Likelihood Estimation**.

Asumimos que:

$$P(y = 1|x; \Phi) = \mathcal{F}_\Phi(x) \quad (2.24)$$

$$P(y = 0|x; \Phi) = 1 - \mathcal{F}_\Phi(x) \quad (2.25)$$

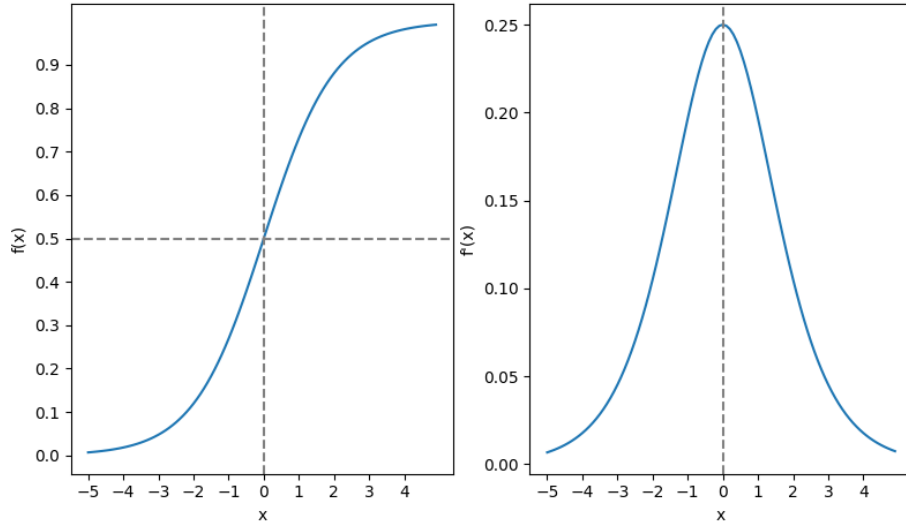


Figure 2.4: Función sigmoidal o logística (a la izquierda) y su correspondiente derivada (a la derecha).

Ambas expresiones anteriores se pueden escribir en forma compacta como:

$$p(y|x; \Phi) = (\mathcal{F}_\Phi(x))^y (1 - \mathcal{F}_\Phi(x))^{1-y} \quad (2.26)$$

Asumiendo que los N datos de entrenamiento fueron generados independientemente, podemos escribir el *likelihood* del modelo $\mathbf{L}(\Phi)$ como:

$$\mathbf{L}(\Phi) = p(\vec{y}|\mathbf{X}; \Phi) \quad (2.27)$$

$$= \prod_{i=1}^N p(y_i|x_i; \Phi) \quad (2.28)$$

$$= \prod_{i=1}^N (\mathcal{F}_\Phi(x_i))^{y_i} (1 - \mathcal{F}_\Phi(x_i))^{1-y_i} \quad (2.29)$$

Definition 2.2.1 Likelihood $\mathbf{L}(\Phi)$ de un modelo paramétrico definido por Φ es la probabilidad condicional de generar las salidas correctas bajo Φ .

Para facilitar los cálculos es recomendable trabajar con el logaritmo del likelihood $\log(\mathcal{F}_\Phi(\cdot))$. Así, para el caso del modelo de regresión logística, tenemos:

$$\log \mathbf{L}(\Phi) = \log \prod_{i=1}^N (\mathcal{F}_\Phi(x_i))^{y_i} (1 - \mathcal{F}_\Phi(x_i))^{1-y_i} \quad (2.30)$$

$$= \sum_{i=1}^N [\log(\mathcal{F}_\Phi(x_i))^{y_i} + \log(1 - \mathcal{F}_\Phi(x_i))^{1-y_i}] \quad (2.31)$$

$$= \sum_{i=1}^N [y_i \log(\mathcal{F}_\Phi(x_i)) + (1 - y_i) \log(1 - \mathcal{F}_\Phi(x_i))] \quad (2.32)$$

Dada la Eq 2.32, el objetivo es fijar Φ maximizando $\log \mathbf{L}(\Phi)$. Esta tarea se conoce como **Maximum Likelihood Estimation (MLE)**.

Una de las formas más eficiente de resolver el problema MLE es a través de un proceso iterativo guiado por el gradiente ascendente. Esto es ir moviendo iterativamente Φ de acuerdo con el gradiente de $\log \mathbf{L}(\Phi)$, bajo la siguiente función de actualización:

$$\Phi = \Phi + \alpha \nabla_{\Phi} \log \mathbf{L}(\Phi) \quad (2.33)$$

Realizaremos un análisis para la entrada (x_i, y_i) . Recordemos que $\nabla_{\Phi}(F) = \left[\frac{\partial F}{\partial \phi_1}, \dots, \frac{\partial F}{\partial \phi_d} \right]$. Así veamos cómo estimar $\frac{\partial}{\partial \phi_j} \log \mathbf{L}(\Phi)$:

$$\begin{aligned} \frac{\partial}{\partial \phi_j} \log \mathbf{L}(\Phi) &= \frac{\partial}{\partial \phi_j} y_i \log(\mathcal{F}_{\Phi}(x_i)) + (1 - y_i) \log(1 - \mathcal{F}_{\Phi}(x_i)) \\ &= \frac{\partial}{\partial \phi_j} y_i \log(\mathcal{F}_{\Phi}(x_i)) + \frac{\partial}{\partial \phi_j} (1 - y_i) \log(1 - \mathcal{F}_{\Phi}(x_i)) \\ &= \frac{\partial}{\partial \phi_j} y_i \log(f(\Phi^T x_i)) + \frac{\partial}{\partial \phi_j} (1 - y_i) \log(1 - f(\Phi^T x_i)) \end{aligned} \quad (2.34)$$

donde, aplicando regla de la cadena, tenemos:

$$\begin{aligned} \frac{\partial}{\partial \phi_j} y_i \log(f(\Phi^T x_i)) &= y_i \frac{1}{f(\Phi^T x_i)} f(\Phi^T x_i)(1 - f(\Phi^T x_i)) x_{i,j} \\ &= y_i x_{i,j} - y_i f(\Phi^T x_i) x_{i,j} \end{aligned} \quad (2.35)$$

$$\begin{aligned} \frac{\partial}{\partial \phi_j} (1 - y_i) \log(1 - f(\Phi^T x_i)) &= (1 - y_i) \frac{1}{1 - f(\Phi^T x_i)} (-f(\Phi^T x_i))(1 - f(\Phi^T x_i)) x_{i,j} \\ &= (y_i - 1) f(\Phi^T x_i) x_{i,j} \\ &= y_i (f(\Phi^T x_i)) x_{i,j} - f(\Phi^T x_i) x_{i,j} \end{aligned} \quad (2.36)$$

Entonces, sumando Eq. 2.35 con Eq. 2.36 tenemos:

$$\begin{aligned} \frac{\partial}{\partial \phi_j} \log \mathbf{L}(\Phi) &= y_i x_{i,j} - f(\Phi^T x_i) x_{i,j} \\ &= (y_i - f(\Phi^T x_i)) x_{i,j} \\ &= (y_i - \mathcal{F}_{\Phi}(x_i)) x_{i,j} \end{aligned} \quad (2.37)$$

Así, el proceso de aprendizaje se realiza a través de la siguiente formulación:

$$\phi_j = \phi_j + \alpha (y_i - \mathcal{F}_{\Phi}(x_i)) x_{i,j} \quad (2.38)$$

donde α es la tasa de aprendizaje. Este factor tiene el objetivo de controlar la velocidad de cambio de los parámetros. Más adelante, profundizaremos en las estrategias de aprendizaje.

Además, podemos extender la Eq. 2.38 de modo que considere una muestra de N ejemplos de entrenamiento. Así tenemos:

$$\phi_j = \phi_j + \alpha \frac{1}{N} \left(\sum_{i=1}^N (y_i - \mathcal{F}_{\Phi}(x_i)) x_{i,j} \right) \quad (2.39)$$

2.2.1 Implementación

En este caso, implementaremos la clase *LogRegression* equipado con las funciones *fit* y *predict* como en el caso de regresión lineal. El código de esta implementación se muestra en el código 2.3

Código 2.3: Regresión Logística

```

1  import numpy as np
2  import linear_models.activations as activations
3  import metrics.metrics as metrics
4
5  class LogRegression :
6      def __init__(self):
7          self.lr = 0.01
8          self.coeff = None
9          self.dim = None
10         self.steps = 100
11         self.min_error = 0.0001
12         self.print_step = 10
13
14     def fit(self, X_train:np.ndarray, y_train : np.ndarray) -> np.ndarray :
15         ones = np.ones((X_train.shape[0],1))
16         X = np.concatenate([ones, X_train], axis = 1)
17         if len(y_train.shape) == 1 :
18             y_train = np.expand_dims(y_train, axis = 1)
19         print(y_train)
20         self.dim = X.shape[1]
21         self.coeff = np.random.normal(loc = 0, scale = 0.1, size = (self.dim
22 ,1))
23
24         for i in range(self.steps) :
25             y_pred = self.predict(X, add_ones = False)
26             if i % self.print_step == 0 :
27                 acc = metrics.accuracy(y_train, y_pred)
28                 print('it {} acc {}'.format(i,acc), flush = True)
29
30             diff = (y_train - y_pred)
31             adjust = diff * X
32             adjust = np.mean(adjust, axis = 0, keepdims = True)
33             adjust = np.transpose(adjust)
34             self.coeff = self.coeff + self.lr * adjust
35
36         return self.coeff
37
38     def predict(self, X: np.ndarray, add_ones = True) -> np.ndarray :
39         if add_ones :
40             ones = np.ones((X.shape[0],1))
41             X = np.concatenate([ones, X], axis = 1)
42         y_pred = np.matmul(X, self.coeff)
43         y_pred = activations.sigmoid(y_pred)
44         y_pred[y_pred < 0.5] = 0
45         y_pred[y_pred >= 0.5] = 1
46         return y_pred

```

2.2.2 Ejemplo (Iris dataset)

Este es uno de los conjuntos de datos más famosos en el contexto de reconocimiento de patrones. Este conjunto contiene características de tres clases de plantas del género Iris. iris setosa, iris versicolor e iris virgínica. La Figura 2.5 ilustra cada una de este tipo de plantas.



Figure 2.5: Ilustración de las clases iris setosa, iris versicolor e iris virgínica, respectivamente.

El conjunto de datos Iris contiene 50 instancias por cada una de las 3 clases. Para cada instancia se han medido los siguientes cuatro atributos:

- x_1 : Largo del sépal (cm)
- x_2 : Ancho del sépal (cm)
- x_3 : Largo del pétalo (cm)
- x_4 : Ancho del pétalo (cm)

La Figura 2.6 muestra una proyección 2D del conjunto de datos Iris, cada color representa una clase diferente. Este gráfico muestra que hay una clase linealmente separable del resto (la clase de color rojo).

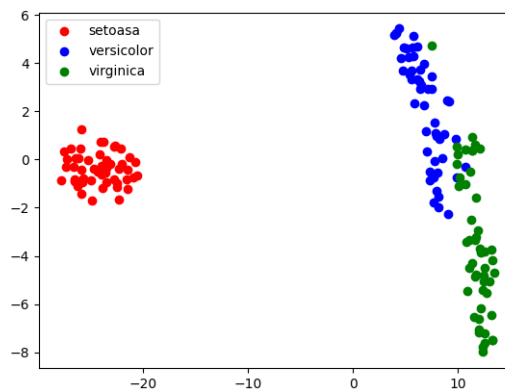


Figure 2.6: Proyección 2D (tSNE) del conjunto de datos Iris.

Para nuestro ejemplo, consideraremos solo dos clases, la clase setosa (color rojo) y la clase versicolor (color azul). El Código 2.4 muestra la implementación de regresión logística sobre el conjunto de datos Iris. Es importante notar que la métrica usada en este caso es

Accuracy, que corresponde al porcentaje de predicciones correctas sobre el total de datos evaluados.

Código 2.4: Ejemplo Regresión Logística

```

1  import numpy as np
2  import linear_models.logreg as logreg
3  import sklearn.datasets as datasets
4  import metrics.metrics as metrics
5  #dataset
6  iris = datasets.load_iris()
7  X = iris.data[:100,:]
8  y = iris.target[:100]
9
10 n = X.shape[0]
11 n_test = int(np rint(0.2 * n))
12
13 # random sort
14 idx = np.random.permutation(n)
15 X = X[idx]
16 y = y[idx]
17
18 X_train = X[:-n_test]
19 X_test = X[-n_test:]
20 y_train = y[:-n_test]
21 y_test = y[-n_test:]
22
23 print(y_test.shape)
24 print(X_test.shape)
25 #Logistic Regression
26 model = logreg.LogRegression()
27
28 print(y_train)
29 coeff = model.fit(X_train, y_train)
30
31 #Evaluation (accuracy x clase)
32 y_pred =model.predict(X_test)
33 acc= metrics.accuracy(np.expand_dims(y_test, axis = 1) , y_pred)
34 print('Acc Test {}'.format(acc))
35 #

```

2.3 Modelo Lineal Generalizado GLM

Hemos vistos dos modelos lineales, uno para el caso de regresión, al que llamamos regresión lineal; y el otro para el caso de clasificación. Este último modelo se conoce como regresión logística. Ambos casos pueden ser interpretados en términos probabilísticos de la siguiente manera:

- **Regresión**

$$y|x; \Phi \sim \mathcal{N}(\mu, \sigma^2) \quad (2.40)$$

- **Clasificación**

$$y|x; \Phi \sim \text{Bernoulli}(\theta) \quad (2.41)$$

Dado lo anterior, es posible definir una **familia de distribuciones** que generalice los modelos lineales antes vistos. Esta familia es conocida como Familia Exponencial y se define como:

$$p(y; \eta) = b(y) \exp(\eta^T T(y) - a(\eta)), \quad (2.42)$$

donde :

- η es el parámetro *natural* asociado a los datos de entrada.
- $T(y)$ es un estadístico sobre y .
- $a(\eta)$ representa un factor de normalización.

Así, fijando T , a y b se define una familia de distribuciones parametrizada por η .

GLM para Clasificación

Como mencionamos anteriormente, el problema de clasificación binaria se puede modelar, linealmente como una distribución de Bernoulli donde $P(y = 1; \theta) = \theta$ y $P(y = 0; \theta) = 1 - \theta$. Así, la distribución queda definida por:

$$p(y; \theta) = \theta^y (1 - \theta)^{1-y} \quad (2.43)$$

$$= \exp(y \log \theta + (1 - y) \log(1 - \theta)) \quad (2.44)$$

$$= \exp\left(y \left[\log \frac{\theta}{1 - \theta} \right] + \log(1 - \theta)\right) \quad (2.45)$$

Dado lo anterior, tenemos que:

- $\eta = \log \frac{\theta}{1 - \theta}$, de donde podemos obtener que $\theta = 1 / (1 + \exp(-\eta))$
- $T(y) = y$
- $a(\eta) = -\log(1 - \theta) = \log(1 + \exp(\eta))$

Inferencia en Clasificación

El valor de salida de un modelo $\mathcal{F}_\Phi(x)$ será el valor esperado $\mathbb{E}(y|x)$, donde $\eta = \Phi^T x$.

Por lo tanto, en el caso clasificación binaria, la salida será el valor esperado de la distribución de Bernoulli(θ), que es igual a θ . Así:

$$\mathcal{F}_\Phi(x) = \theta \quad (2.46)$$

$$= \frac{1}{1 + e^{-\eta}} \quad (2.47)$$

$$= \frac{1}{1 + e^{-\Phi^T x}} \quad (2.48)$$

lo que representa el modelo de regresión logística visto en Sección 2.2.

2.4 Regresión Multiclase

Dado la generalización de un modelo lineal GLM, extenderemos el modelo de regresión logística para el caso inferir múltiples clases.

En un problema de múltiples clases $y \in \{1, 2, \dots, k\}$, donde k es el número de clases a inferir. Por ejemplo, podemos pensar en la clasificación de correos electrónicos, donde un correo puede ser clasificado como *spam*, *no-spam*, *correo de trabajo* o *correo personal*.

Así, extenderemos la definición de clasificación binaria de Eq. 2.26. Para este fin, definimos k valores $\{\theta_1, \dots, \theta_k\}$ que representan la probabilidad de ocurrencia de cada clase con respecto a cierta entrada. Así, la probabilidad de la clase y , dado x que definida por:

$$p(y; \theta) = \theta_1^{\mathbb{1}\{y=1\}} \theta_2^{\mathbb{1}\{y=2\}} \dots \theta_k^{\mathbb{1}\{y=k\}} \quad (2.49)$$

donde, $\mathbb{1}\{\cdot\}$ es una función indicadora que toma el valor de 1 si su argumento es verdadero y 0 en caso contrario. Así, $\mathbb{1}\{3=2\} = 0$ y $\mathbb{1}\{2=2\} = 1$.

Debemos notar que los k valores son redundantes ya que $\sum_{i=1}^{i=k} \theta_i = 1$. Por lo tanto, podemos escribir uno de los parámetros en función de los otros $k-1$. Por ejemplo, $\theta_k = 1 - \sum_{i=1}^{i=k-1} \theta_i$, con lo que la Eq. 2.49 se puede reescribir como:

$$p(y; \theta) = \theta_1^{\mathbb{1}\{y=1\}} \theta_2^{\mathbb{1}\{y=2\}} \dots \theta_k^{1 - \sum_{i=1}^{i=k-1} \mathbb{1}\{y=i\}} \quad (2.50)$$

Además, definiremos $T(y) \in \mathbb{R}^{k-1}$, que representará la probabilidad de ocurrencia de cada una de las clases para una entrada determinada. Así:

$$T(1) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, T(2) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, T(3) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \dots, T(k-1) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}, T(k) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad (2.51)$$

notemos que $T(k) = 0$, pues representa la clase complemento, es decir cuando ninguna de las $T_{1\dots k-1}$ se activa.

La representación anterior es conocida como **one-hot code**, y puede ser expresada como $[T(y)]_i = \mathbb{1}\{y=i\}$, donde $[T(y)]_i$ indica el elemento i -ésimo del vector $T(y)$. Así, con las expresiones indicadas, podemos reescribir la Eq. 2.50 como:

$$p(y; \theta) = \theta_1^{[T(y)]_1} \theta_2^{[T(y)]_2} \dots \theta_k^{1 - \sum_{i=1}^{i=k-1} [T(y)]_i} \quad (2.52)$$

$$= \exp \left([T(y)]_1 \log \theta_1 + [T(y)]_2 \log \theta_2 + \dots + \left(1 - \sum_{i=1}^{k-1} [T(y)]_i\right) \log \theta_k \right) \quad (2.53)$$

$$\begin{aligned} &= \exp \left([T(y)]_1 \log \theta_1 + [T(y)]_2 \log \theta_2 + \dots + \log \theta_k - \log \theta_k \sum_{i=1}^{k-1} [T(y)]_i \right) \\ &= \exp ([T(y)]_1 \log(\theta_1/\theta_k) + [T(y)]_2 \log(\theta_2/\theta_k) + \dots + [T(y)]_{k-1} \log(\theta_{k-1}/\theta_k) + \log \theta_k) \\ &= \exp(\eta^T T(y) - a(\eta)), \end{aligned} \quad (2.54)$$

donde:

$$\eta = \begin{bmatrix} \log(\theta_1/\theta_k) \\ \log(\theta_2/\theta_k) \\ \vdots \\ \log(\theta_{k-1}/\theta_k) \end{bmatrix} \quad (2.55)$$

$$a(\eta) = -\log(\theta_k) \quad (2.56)$$

$$b(y) = 1 \quad (2.57)$$

De la Eq. 2.55 tenemos:

$$\eta_i = \log \frac{\theta_i}{\theta_k} \quad (2.58)$$

de la que podemos deducir:

$$\begin{aligned} e^{\eta_i} &= \frac{\theta_i}{\theta_k} \\ \theta_i &= \theta_k e^{\eta_i} \end{aligned} \quad (2.59)$$

Por otro lado, podemos hacer que $\sum_{i=1}^{k-1} \theta_i = 1$, de modo que se cumplan las siguientes igualdades:

$$\begin{aligned} \sum_{i=1}^{k-1} \theta_i &= 1 \\ \sum_{i=1}^{k-1} \theta_k e^{\eta_i} &= 1 \\ \theta_k \sum_{i=1}^{k-1} e^{\eta_i} &= 1 \\ \theta_k &= \frac{1}{\sum_{i=1}^{k-1} e^{\eta_i}} \end{aligned} \quad (2.60)$$

Por lo que, reescribiendo Eq. 2.59, obtenemos:

$$\theta_i = \frac{e^{\eta_i}}{\sum_{j=1}^{k-1} e^{\eta_j}} \quad (2.61)$$

donde $\theta_i = p(y = i | \mathbf{x}; \Phi)$. Pero ¿en dónde están involucrados $\mathbf{x}$ y Φ ? Estos elementos están relacionados **linealmente** entre sí representados por η según $\eta_i = \Phi_i^T \mathbf{x}$. Así, Φ_i define al predictor de la clase i , por lo que en nuestro modelo tendremos $k - 1$ conjuntos de parámetros ($k - 1$ predictores): $\Phi_1, \Phi_2, \dots, \Phi_{k-1}$.

Dado lo anterior, nuestro modelo asume que la distribución condicional de y dado $\mathbf{x}$ sigue la siguiente expresión:

$$p(y = i | \mathbf{x}; \Phi) = \frac{e^{\Phi_i^T \mathbf{x}}}{\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}}} \quad (2.62)$$

Con la expresión anterior, podemos modelar $\mathcal{F}_\Phi()$ para el problema de regresión multiclase. Este modelo estimará el valor esperado para cada clase dada la entrada $\mathbf{x}$ ($p(y = i | \mathbf{x}; \Phi)$),

bajo el siguiente formulación:

$$\mathcal{F}_\Phi(\mathbf{x}) = \mathbb{E}[T(y)|\mathbf{x}; \Phi] \quad (2.63)$$

$$= \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{k-1} \end{bmatrix} \quad (2.64)$$

$$= \begin{bmatrix} \frac{e^{\Phi_1^T \mathbf{x}}}{\sum_{i=1}^{k-1} e^{\Phi_i^T \mathbf{x}}} \\ \frac{e^{\Phi_2^T \mathbf{x}}}{\sum_{i=1}^{k-1} e^{\Phi_i^T \mathbf{x}}} \\ \vdots \\ \frac{e^{\Phi_{k-1}^T \mathbf{x}}}{\sum_{i=1}^{k-1} e^{\Phi_i^T \mathbf{x}}} \end{bmatrix} \quad (2.65)$$

El modelo deducido es conocido como **regresión Softmax**.

Es importante notar la diferencia ente ϕ y θ .

2.4.1 Entrenamiento de Softmax

Nuestro objetivo es estimar Φ , de modo que maximice $\mathbb{E}[T(y)|\mathbf{x}; \Phi]$. Para este fin, seguiremos la estrategia **Maximum Likelihood Estimation** (MLE) usando gradiente ascendente (ver Eq. 2.33), tal como lo hicimos para el caso de regresión logística (Sección 2.2).

$$\log \mathbf{L}(\Phi) = \sum_{i=1}^N \log p(y_i | \mathbf{x}_i; \Phi) \quad (2.66)$$

$$\log \mathbf{L}(\Phi) = \sum_{i=1}^N \log \theta_1^{\mathbb{1}\{y=1\}} \theta_2^{\mathbb{1}\{y=2\}} \dots \theta_{k-1}^{\mathbb{1}\{y=k-1\}} \quad (2.67)$$

$$\log \mathbf{L}(\Phi) = \sum_{i=1}^N \log \Pi_{l=1}^{k-1} \left(\theta_l^{\mathbb{1}\{y_i=l\}} \right) \quad (2.68)$$

$$\log \mathbf{L}(\Phi) = \sum_{i=1}^N \log \Pi_{l=1}^{k-1} \left(\frac{e^{\Phi_l^T \mathbf{x}_i}}{\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}_i}} \right)^{\mathbb{1}\{y_i=l\}} \quad (2.69)$$

El proceso de optimización requiere estimar $\frac{\partial \log \mathbf{L}(\Phi)}{\partial \phi_{r,s}}$, para cada uno de los parámetros de Φ . En este caso, definimos un parámetro con los índices (r, s) , donde r indica el conjunto de parámetros del predictor Φ_r . Por lo tanto $\phi_{r,s} \in \Phi_r$.

$$\frac{\partial \log \mathbf{L}(\Phi)}{\partial \phi_{r,s}} = \frac{\partial}{\partial \phi_{r,s}} \sum_{i=1}^N \log \Pi_{l=1}^{k-1} \left(\frac{e^{\Phi_l^T \mathbf{x}_i}}{\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}_i}} \right)^{\mathbb{1}\{y_i=l\}} \quad (2.70)$$

donde N es la cantidad de datos de entrenamiento. Para facilitar el análisis supondremos una sola muestra $(\mathbf{x}, y)$, donde $\mathbf{x}$ es la entrada e $y \in \{1, 2, \dots, k\}$ es la clase. Así, reescribimos

la expresión anterior como:

$$\frac{\partial \log \mathbf{L}(\Phi)}{\partial \phi_{r,s}} = \frac{\partial}{\partial \phi_{r,s}} \log \Pi_{l=1}^{k-1} \left(\frac{e^{\Phi_l^T \mathbf{x}}}{\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}}} \right)^{\mathbb{1}\{y=l\}} \quad (2.71)$$

$$= \frac{\partial}{\partial \phi_{r,s}} \log \Pi_{l=1}^{k-1} \left(\frac{e^{\Phi_l^T \mathbf{x}}}{\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}}} \right)^{\mathbb{1}\{y=l\}} \quad (2.72)$$

$$= \frac{\partial}{\partial \phi_{r,s}} \log \left(\frac{e^{\Phi_y^T \mathbf{x}}}{\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}}} \right) \quad (2.73)$$

$$= \frac{\partial}{\partial \phi_{r,s}} \log(e^{\Phi_y^T \mathbf{x}}) - \frac{\partial}{\partial \phi_{r,s}} \log \left(\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}} \right) \quad (2.74)$$

$$= \frac{\partial}{\partial \phi_{r,s}} \Phi_y^T \mathbf{x} - \left(\frac{1}{\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}}} \right) \frac{\partial}{\partial \phi_{r,s}} \left(\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}} \right) \quad (2.75)$$

$$= \frac{\partial}{\partial \phi_{r,s}} \Phi_y^T \mathbf{x} - \left(\frac{1}{\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}}} \right) \frac{\partial}{\partial \phi_{r,s}} e^{\Phi_r^T \mathbf{x}} \quad (2.76)$$

$$= \frac{\partial}{\partial \phi_{r,s}} \Phi_y^T \mathbf{x} - \left(\frac{1}{\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}}} \right) e^{\Phi_r^T \mathbf{x}} \frac{\partial}{\partial \phi_{r,s}} \Phi_r^T \mathbf{x} \quad (2.77)$$

$$= \frac{\partial}{\partial \phi_{r,s}} \Phi_y^T \mathbf{x} - \left(\frac{e^{\Phi_r^T \mathbf{x}}}{\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}}} \right) x_s \quad (2.78)$$

$$(2.79)$$

Nos queda resolver el primer término $\frac{\partial}{\partial \phi_{r,s}} \Phi_y^T \mathbf{x}$. Sin embargo, esto dependerá si $\phi_{r,s}$ es un parámetro del predictor de la clase de $\mathbf{x}$ o no. Así, tendremos dos casos:

- Si $r = y$, tenemos:

$$\frac{\partial}{\partial \phi_{r,s}} \Phi_y^T \mathbf{x} = x_s \quad (2.80)$$

$$\Rightarrow \frac{\partial \log \mathbf{L}(\Phi)}{\partial \phi_{r,s}} = x_s - \left(\frac{e^{\Phi_r^T \mathbf{x}}}{\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}}} \right) x_s \quad (2.81)$$

- Si $r \neq y$, tenemos:

$$\frac{\partial}{\partial \phi_{r,s}} \Phi_y^T \mathbf{x} = 0 \quad (2.82)$$

$$\Rightarrow \frac{\partial \log \mathbf{L}(\Phi)}{\partial \phi_{r,s}} = - \left(\frac{e^{\Phi_r^T \mathbf{x}}}{\sum_{j=1}^{k-1} e^{\Phi_j^T \mathbf{x}}} \right) x_s \quad (2.83)$$

Con los casos anteriores, el parámetro $\phi_{r,s}$ es actualizado de la siguiente manera:

$$\phi_{r,s} = \phi_{r,s} + \alpha \left(\frac{\partial \log \mathbf{L}(\Phi)_{(x,y)}}{\partial \phi_{r,s}} \right) \quad (2.84)$$

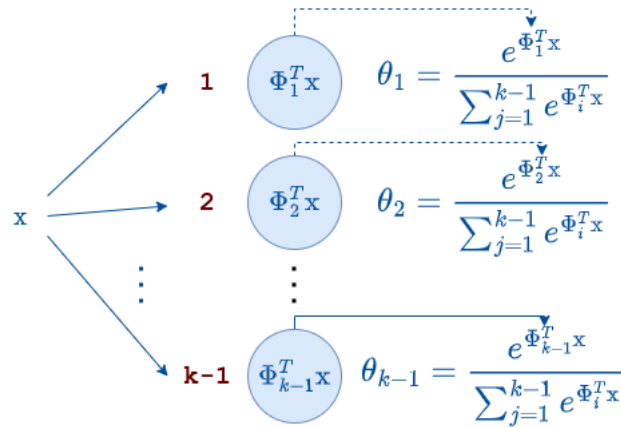


Figure 2.7: Modelo de Regresión Softmax

siendo α el factor o tasa de aprendizaje.

Nuestro estudio anterior ha considerado una sola muestra de entrenamiento (x, y) . Sin embargo, fácilmente podemos calcular las derivadas para un conjunto mayor, dado que la derivadas de una suma de elementos es igual a la suma de las derivadas de cada elemento. Por lo tanto, para N muestras tendríamos:

$$\phi_{r,s} = \phi_{r,s} + \alpha \left(\sum_{i=1}^N \frac{\partial \log \mathbf{L}(\Phi)_{(x_i, y_i)}}{\partial \phi_{r,s}} \right) \quad (2.85)$$

La Figura 2.7 muestra un esquema del modelo de regresión Softmax.

Implementación

Código 2.5: Regresión Softmax

```

1 import numpy as np
2 import linear_models.activations as activations
3 import metrics.metrics as metrics
4
5 class SoftmaxReg :
6     def __init__(self, n_classes):
7         self.lr = 0.01
8         self.n_classes = n_classes
9         self.coeff = None
10        self.dim = None
11        self.steps = 100
12        self.min_error = 0.0001
13        self.print_step = 20
14
15    def fit(self, X_train:np.ndarray, y_train : np.ndarray) -> np.ndarray :
16        n = X_train.shape[0]
17        ones = np.ones((n,1))
18        X = np.concatenate([ones, X_train], axis = 1)
19        if len(y_train.shape) == 1 :
20            y_train = np.expand_dims(y_train, axis = 1)

```

```

21     self.dim = X.shape[1]
22
23     #weights are initialized randomly
24     self.coeff = np.random.normal(loc = 0, scale = 0.1, size = (self.dim
, self.n_classes))
25
26     for i in range(self.steps) :
27         y_pred = self.predict(X, add_ones = False)
28         if i % self.print_step == 0 :
29             acc = metrics.multiclass_accuracy(y_train, y_pred)
30             print('it {} acc {}'.format(i, acc), flush = True)
31
32         #softmax = activations.softmax(np.matmul(X, self.coeff), axis =
1)
33
34         #y_pred es softmax
35         adjust = - np.matmul(np.transpose(X), y_pred)
36         y = np.squeeze(y_train)
37         for cl in np.arange(self.n_classes):
38             x_mean_class = np.sum(X[np.where(y == cl)[0], :], axis = 0)
39             adjust[:, cl] = adjust[:, cl] + x_mean_class
40
41         #taking the mean
42         adjust = adjust / n
43         self.coeff = self.coeff + self.lr * adjust
44     return self.coeff
45
46     def predict(self, X: np.ndarray, add_ones = True) -> np.ndarray :
47         if add_ones :
48             ones = np.ones((X.shape[0], 1))
49             X = np.concatenate([ones, X], axis = 1)
50         y_pred = activations.softmax(np.matmul(X, self.coeff), axis = 1)
51         return y_pred

```

Ejemplo Iris

Código 2.6: Ejemplo de Regresión Softmax

```

1  from sklearn import datasets
2  import linear_models.softmaxreg as softmaxreg
3  import umap
4  import matplotlib.pyplot as plt
5  import numpy as np
6  import metrics.metrics as metrics
7
8  iris = datasets.load_iris()
9  X = iris.data
10 y = iris.target
11
12
13 # Split the data into training/testing sets
14 # training 80% testing 20%
15 n = X.shape[0]
16 n_test = int(np rint(0.2 * n))
17
18 # random sort
19 idx = np.random.permutation(n)
20 X = X[idx, :]

```

```

21 y = y[idx]
22
23 X_train = X[:-n_test, :]
24 X_test = X[-n_test:,:]
25 y_train = y[:-n_test]
26 y_test = y[-n_test:]
27 """ data normalization, improve convergence """
28 mu = np.mean(X_train, axis = 0)
29 dst = np.std(X_train, axis = 0)
30 X_train = (X_train - mu) / dst
31 X_test = (X_test - mu) / dst
32 """-----"""
33
34 SM = softmaxreg.SoftmaxReg(3)
35 coeff = SM.fit(X_train, y_train)
36 y_pred = SM.predict(X_test)
37
38 # show confusion matrix
39 cm = metrics.confusion_matrix(y_test, y_pred, 3)
40 print('confusion matrix')
41 print(cm)
42
43 # Just data visualization
44 # #project 2D
45 # reducer = umap.UMAP()
46 # print('UMAP', flush = True)
47 # reducer.fit(X)
48 # embedding = reducer.transform(X)
49 # print(embedding.shape)
50 # plt.scatter(embedding[:, 0], embedding[:, 1], c=y, cmap='Paired')
51 # plt.show()

```

2.4.2 Tarea

Evaluar el modelo de regresión softmax en el conjunto de datos MNIST (ver Figura 2.8). Para esto, evaluaremos el modelo con dos tipos de entradas:

- Datos crudos provenientes de la propia imagen ($d = 28 \times 28$).
- Una función de extracción de características HOG ($d =$).



Figure 2.8: Una muestra de imágenes de MNIST



3. Redes Neuronales

En los capítulos anteriores, hemos revisado métodos de aprendizaje que asumen una relación lineal entre entradas (x) y las salidas (y). Sin embargo, esta suposición suele estar lejos de la gran diversidad de aplicaciones. Lo anterior se conoce como la “**No Separabilidad Lineal**”.

Una de las formas de resolver el problema anterior es transformando los datos, mediante funciones no-lineales a un espacio que se aproxime a la *suposición lineal*. Lo anterior es realizado, hoy en día, mediante modelos de *deep learning* que usan redes neuronales con un alto número de parámetros, organizados jerárquicamente en capas. En este capítulo haremos una introducción a los modelos neuronales, desde el *Perceptrón* hasta modelos atencionales, pasando por modelos convolucionales.

3.1 Modelo Biológico

Las redes neuronales artificiales (ANN) son parte de la corriente conexionista del aprendizaje de máquinas (revisar el libro - *The Master Algorithm* -). Estos modelos son inspirados, al menos en sus inicios, en la red neuronal biológica, donde **la neurona** representa uno de los componentes principales del sistema nervioso. Esta neurona, tiene la función de recibir, procesar y transmitir información, **en forma de señales electroquímicas**, a través de conexiones llamadas *sinapsis*.

3.1.1 Componentes

Las partes más importantes de una neurona son: el cuerpo celular, las dendritas y el axón (ver Figura 3.1). A continuación, se brinda más detalle de cada una de ellas.

- **Cuerpo Celular (soma):** Es la parte de una neurona que recibe información de otras neuronas, que luego sumariza para transmitir otra señal a través del axón.
- **Dendritas:** Son ramificaciones que se desprenden del soma, cuya función es la de recibir información de otras neuronas y llevarlas al soma.

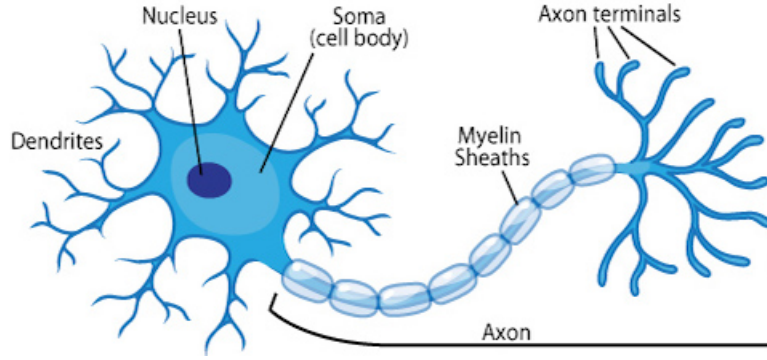


Figure 3.1: La neurona, la unidad de procesamiento mínima

- **Axón:** Es el canal por el cual la neurona envía información hacia otras neuronas. La información viaja desde el soma, pasando por el axón, hasta llegar a los terminales axónicos, que finalmente, se conectan con las dendritas de otras neuronas.

3.2 Perceptron y MLP

El **perceptrón** es el modelo computacional de una neurona biológica, descrito en 1957 por Frank Rosenblatt. Este modelo representa la unidad mínima de procesamiento en una red neuronal artificial. El perceptrón tiene el mismo comportamiento que un modelo de regresión logística (sección 2.2).

Las partes de un perceptrón, o simplemente neurona, se relacionan con las partes de la neurona biológica discutidas anteriormente. La Figura 3.2 ilustra los componentes de un perceptrón.

$$\Phi = [\phi_0, \phi_1, \phi_2, \dots, \phi_d] \quad \mathbf{x} = [x_0, x_1, x_2, \dots, x_d]$$

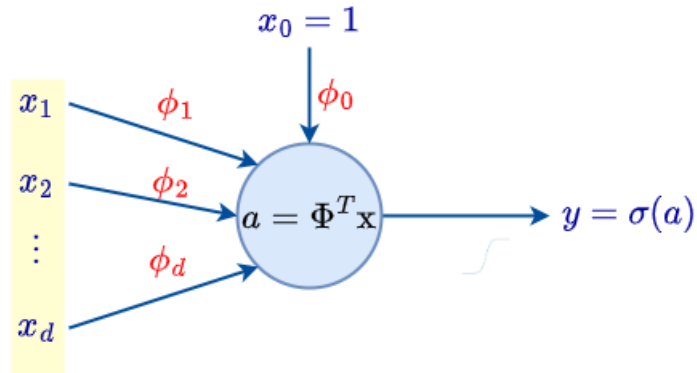


Figure 3.2: Perceptrón, la neurona artificial.

1. **Entradas ($\mathbf{x}$):** La neurona recibe una entrada $\mathbf{x} \in \mathbb{R}^d$, donde $\mathbf{x} = \{x_1, \dots, x_d\}$. Sin embargo, bajo la misma justificación que en el caso de regresión lineal (sección 2.1),

agregaremos un 1 a x de modo que $x = \{x_0 = 1, x_1, \dots, x_d\}$. La entrada x es conocida como el vector de características (*feature vector*).

2. **Pesos (Φ):** La neurona está equipada con un conjunto de parámetros $\Phi \in \mathbb{R}^{d+1}$, donde $\Phi = \{\phi_0, \phi_1, \dots, \phi_d\}$ y ϕ_0 es conocido como **bias**. Estos son los parámetros que la neurona tiene que ajustar.
3. **Valor de agregación (a):** El valor de agregación no es más que la suma agregada del vector de entrada con respecto al vector de parámetros. Esto es:

$$a = \sum_{i=0}^d x_i \phi_i \quad (3.1)$$

Así, la operación de agregación se puede representar por el producto punto (*dot product*) entre el vector de entrada x y el vector de parámetros Φ , $a = \Phi^T x$.

4. **Valor de activación (y):** el valor de activación representa la salida de la neurona. Este valor se calcula a través de una función $f(\cdot)$, generalmente no lineal, aplicada sobre el valor de agregación, es decir $y = f(a)$. En un problema de clasificación binaria, f es la función sigmoidea $f(a) = 1/(1 + e^{-a})$, la misma usada en regresión logística.

3.2.1 Interpretación Geométrica

Un perceptrón representa un hiperplano clasificador, tal como se muestra en la Figura 3.3, que separa el espacio de características en dos partes. En el contexto de clasificación binaria, el hiperplano inferido por una neurona separa el espacio en dos clases. En el ejemplo de la Figura 3.3, vemos que los círculos y los triángulos quedan separados, perfectamente, por el hiperplano azul.

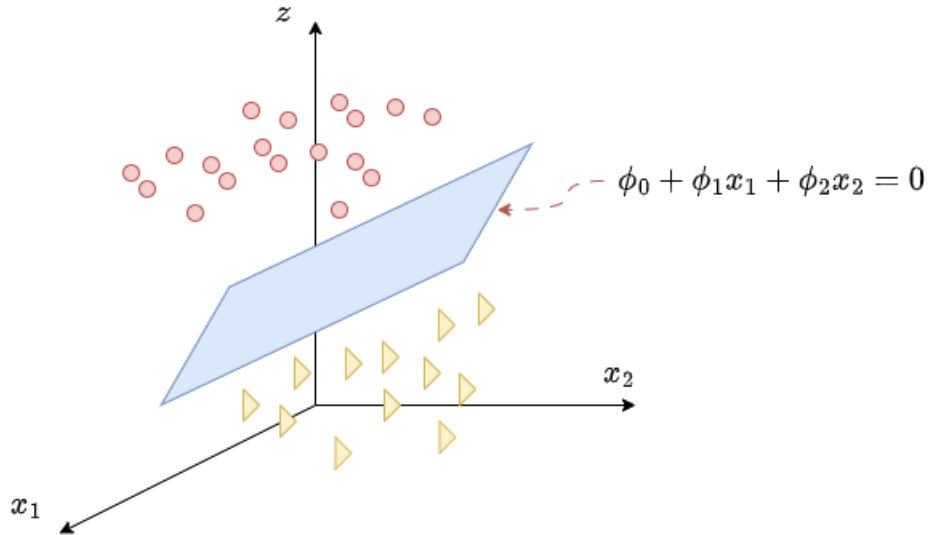


Figure 3.3: Hiperplano separador inferido por una neurona.

3.2.2 Función de Pérdida (Loss)

Una función de pérdida (loss) mide el error entre la predicción ($\hat{y}$) y el valor real (y). Existen diversas funciones de pérdida, las cuales dependen de la tarea en particular que se trata de

resolver. Para el caso del perceptrón, relacionada con clasificación binaria, revisaremos dos funciones de pérdida: error cuadrático medio (MSE) y *binary cross-entropy* (BCE).

1. **MSE**: mide la diferencia entre la predicción y el valor real.

$$\mathcal{L}_i(\Phi, \mathbf{x}_i) = \frac{1}{2}(\hat{y}_i - y_i)^2 \quad (3.2)$$

Notemos que la división por 2 es simplemente una estrategia para facilitar los cálculos posteriores.

2. **BCE**: trata maximizar el valor de salida cuando $y_i = 1$ y minimizarlo cuando $y_i = 0$.

$$\mathcal{L}_i(\Phi, \mathbf{x}_i) = -[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \quad (3.3)$$

En ambos casos anteriores, la pérdida total, $\mathcal{L}(\Phi, \mathbf{X})$, se calcula como:

$$\mathcal{L}(\Phi, \mathbf{X}) = \frac{1}{N} \sum_{i=1}^N \mathcal{L}_i(\Phi, \mathbf{x}_i) \quad (3.4)$$

3.2.3 Entrenamiento

Sea $\Gamma = \langle \mathbf{X}, \mathbf{Y} \rangle$ en conjunto de datos de entrenamiento, compuesto por N muestras, donde $\mathbf{X} = \{x_1, x_2, \dots, x_N\}$ e $\mathbf{Y} = \{y_1, y_2, \dots, y_N\}$. Como estamos en un problema de clasificación binaria, se cumple que $y_i \in \{0, 1\}$. Es importante entender que hay una correspondencia entre los elementos de $\mathbf{X}$ con los elementos de $\mathbf{Y}$, es decir y_i indica la clase de la muestra x_i , donde $x_i \in \mathbb{R}^d$.

Dado Γ , el entrenamiento consiste en ajustar los parámetros de modo que la pérdida se minimice. Para este fin, nos basaremos en la estrategia de **gradiente descendente** que actualiza los pesos en forma iterativa bajo la siguiente expresión:

$$\Phi = \Phi - \alpha \nabla_{\Phi} \mathcal{L}, \quad (3.5)$$

siendo α la tasa de aprendizaje (*learning rate*). [Notemos la diferencia entre el gradiente descendente aquí y el gradiente ascendente usado en el capítulo anterior]

Entonces es hora de calcular el gradiente de la función de pérdida con respecto a los parámetros. Para hacer más fácil la explicación, haremos la deducción con respecto a un parámetro ϕ_j , esto es calcularemos $\frac{\partial}{\partial \phi_j} \mathcal{L}$.

1. **Caso MSE**:

Sea $z = \hat{y}_i - y_i$

$$\frac{\partial}{\partial \phi_j} \frac{1}{2} (\hat{y}_i - y_i)^2 = \frac{\partial}{\partial \phi} \frac{1}{2} (z)^2 \quad (3.6)$$

$$= \frac{\partial}{\partial z} \frac{1}{2} (z)^2 \frac{\partial}{\partial \phi_j} z \quad (3.7)$$

$$= z \frac{\partial}{\partial \phi_j} (\hat{y}_i - y_i) \quad (3.8)$$

$$= z \frac{\partial}{\partial \phi_j} (f(a) - y_i) \quad (3.9)$$

$$= z \frac{\partial}{\partial a} (f(a) - y_i) \frac{\partial}{\partial \phi_j} a \quad (3.10)$$

$$= z f'(a) \frac{\partial}{\partial \phi_j} \Phi^T \mathbf{x}_i \quad (3.11)$$

$$= (\hat{y}_i - y_i) f(a) (1 - f(a)) \mathbf{x}_{i,j} \quad (3.12)$$

$$= (\hat{y}_i - y_i) \hat{y}_i (1 - \hat{y}_i) \mathbf{x}_{i,j} \quad (3.13)$$

$$(3.14)$$

donde, $f(\cdot)$ es la función sigmoideal y a es el valor de agregación $a = \Phi^T \mathbf{x}_i$. Además, $\mathbf{x}_{i,j}$ es el j -ésimo elemento del vector de entrada $\mathbf{x}_i$

2. **Caso BCE** <puede quedar como ejercicio la deducción>

$$\frac{\partial}{\partial \phi_j} (-[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]) = -[y_i \frac{\partial}{\partial \phi_j} \log(\hat{y}_i) \quad (3.15)$$

$$+ (1 - y_i) \frac{\partial}{\partial \phi_j} \log(1 - \hat{y}_i)] \quad (3.16)$$

Necesitaremos definir:

$$\begin{aligned} \Rightarrow \frac{\partial}{\partial \phi_j} \log(\hat{y}_i) &= \frac{\partial}{\partial \phi_j} \log(f(a)) \\ &= \frac{1}{f(a)} \frac{\partial}{\partial a} f(a) \frac{\partial}{\partial \phi_j} a \\ &= \frac{1}{f(a)} f'(a) \mathbf{x}_{i,j} \\ &= \frac{1}{f(a)} f(a) (1 - f(a)) \mathbf{x}_{i,j} \\ &= (1 - f(a)) \mathbf{x}_{i,j} \end{aligned} \quad (3.17)$$

$$\begin{aligned}
\Rightarrow \frac{\partial}{\partial \phi_j} \log(1 - \hat{y}_i) &= \frac{\partial}{\partial \phi_j} \log(1 - f(a)) \\
&= - \frac{1}{1 - f(a)} \frac{\partial}{\partial a} f(a) \frac{\partial}{\partial \phi_j} a \\
&= - \frac{1}{1 - f(a)} f'(a) x_{i,j} \\
&= - \frac{1}{1 - f(a)} f(a)(1 - f(a)) x_{i,j} \\
&= - f(a) x_{i,j}
\end{aligned} \tag{3.18}$$

Finalmente,

$$\begin{aligned}
\frac{\partial}{\partial \phi_j} (-[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]) &= -[y_i(1 - f(a)) x_{i,j} \\
&\quad + (1 - y_i)(-1)f(a) x_{i,j}] \\
&= f(a) x_{i,j} - y_i x_{i,j} \\
&= (f(a) - y_i) x_{i,j} \\
&= (\hat{y}_i - y_i) x_{i,j}
\end{aligned} \tag{3.19}$$

Observen la deducción de incremento de los parámetros de BCE y compárenlo con el modelo de regresión logístico.

Ahora que ya tenemos calculado el incremento de los pesos, podemos entrenar el modelo. Para este fin, los parámetros iniciales pueden quedar inicializados con valores aleatorios, generalmente, bajo una distribución normal $\mathcal{N}(\mu, \sigma)$ (p.ej. $\mu = 0$ y $\sigma = 0.1$). El proceso de entrenamiento es iterativo hasta que se cumpla alguna condición de parada. Por ejemplo:

- Se itera por cierto número de iteraciones o pasos (steps).
- Se llega a un error o valor de pérdida mínimo ($loss < th$).
- No existen mayores cambios en los pesos en los últimos pasos.

El Algoritmo 1 describe cómo entrenar un perceptrón en la tarea de clasificación binaria.

Algorithm 1 (Entrenamiento Perceptrón (*fit*))

Require: $\mathbf{X} = \{x_1, \dots, x_N\}$, donde $x_i \in \mathbb{R}^d$

Require: $\mathbf{Y} = \{y_1, \dots, y_N\}$, donde $y_i \in \{0, 1\}$

Require: α : tasa de aprendizaje

$\Phi_{d+1} \sim \mathcal{N}(0, 0.1)$ (inicialización)

while !converge **do**

$\Phi \leftarrow \Phi - \alpha \nabla_{\Phi} \mathcal{L}(\Phi, \mathbf{X}, \mathbf{Y})$

$loss \leftarrow \text{computeLoss}(\Phi, \mathbf{X}, \mathbf{Y})$

 print $loss$

end while

return Φ

3.2.4 Implementación

Implementaremos la clase *Perpcetron* (ver Código 3.1), con las operaciones *fit* y *predict*. La operación *fit* sigue el Algoritmo 1.

Código 3.1: Perceptrón

```

1  """
2  This class implements the Perceptron
3  """
4  import numpy as np
5  import activations.activations as activations
6  import metrics.metrics as metrics
7  import losses.losses as losses
8
9  class Perceptron :
10     def __init__(self):
11         self.lr = 0.1
12         self.coeff = None
13         self.dim = None
14         self.steps = 1000
15         self.min_error = 0.0001
16         self.print_step = 1
17         self.loss = 'bce'
18
19     def setLoss(self, loss:str):
20         if loss in ['bce', 'mse'] :
21             self.loss = loss
22
23     def setSteps(self, steps:int):
24         self.steps = steps
25
26     def setLearningRate(self, lr:float):
27         self.lr = lr
28
29     def fit(self, X_train:np.ndarray, y_train : np.ndarray) -> np.ndarray :
30         ones = np.ones((X_train.shape[0],1))
31         X = np.concatenate([ones, X_train], axis = 1)
32
33         if len(y_train.shape) == 1 :
34             y_train = np.expand_dims(y_train, axis = 1)
35
36         self.dim = X.shape[1]
37         self.n = X.shape[0]
38         self.coeff = np.random.normal(loc = 0, scale = 0.1, size = (self.dim
39 ,1))
40         print(self.coeff)
41
42         for i in range(self.steps) :
43             y_pred = self.predict(X, add_ones = False)
44
45             if i % self.print_step == 0 :
46                 acc = metrics.accuracy(y_train, y_pred.copy())
47                 if self.loss == 'bce' :
48                     loss = losses.bce_loss(y_train, y_pred)
49                 else :
50                     loss = losses.mse_loss(y_train, y_pred)

```

```

50         print('it {} {} loss {}'.format(i,self.loss, loss), flush =
      True)
51         print('it {} acc {}'.format(i,acc), flush = True)
52
53         diff = (y_pred - y_train)
54
55         if self.loss == 'mse' :
56             df = y_pred * (1.0 - y_pred)
57             diff = diff * df
58
59         adjust = diff * X
60         adjust = np.mean(adjust, axis = 0, keepdims = True)
61         adjust = np.transpose(adjust)
62         print(adjust)
63         self.coeff = self.coeff - self.lr * adjust
64         print(self.coeff)
65     return self.coeff
66
67
68     def predict(self, X: np.ndarray, add_ones = True) -> np.ndarray :
69         if add_ones :
70             ones = np.ones((X.shape[0],1))
71             X = np.concatenate([ones, X], axis = 1)
72         y_pred = np.matmul(X, self.coeff)
73         y_pred = activations.sigmoid(y_pred)
74         return y_pred

```

3.2.5 Ejemplos

Empezaremos por dos ejemplos sencillos, que sirven para ilustrar el comportamiento del perceptrón en forma didáctica. Se trata de las operaciones lógica AND y OR que representan un espacio linealmente separable.

En los siguientes casos, $\alpha = 1.0$.

1. **AND** Aquí definimos $\mathbf{X} = (0,0), (0,1), (1,0), (1,1)$ con sus correspondientes etiquetas $\mathbf{Y} = 0,0,0,1$

$$\begin{aligned}
\Phi &= [0, 0, 0.2] \\
a &= [0, 0.20, 0.2] \\
\hat{y} &= [0.5, 0.549834, 0.5, 0.549834] \\
\hat{y} - y &= [0.5, 0.549834, 0.5, -0.450166] \\
inc &= [0.274917, 0.0124585, 0.024917] \\
\Phi &= \Phi - inc \\
\Phi &= [-0.274917, -0.0124585, 0.175083] \\
\hat{y} &= [0.43170038, 0.47506221, 0.42864651, 0.47195634] \\
&\vdots \\
&\text{(luego de 20 pasos)} \\
\Phi &= [-1.91755057, 0.98649461, 1.05394902] \\
\hat{y} &= [0.12813496, 0.29658743, 0.28271053, 0.53068466]
\end{aligned}$$

2. **OR** Aquí definimos $\mathbf{X} = (0,0), (0,1), (1,0), (1,1)$ con sus correspondientes etiquetas $\mathbf{Y} = 0, 1, 1, 1$

¿Qué pasaría si intentamos que el perceptrón aprenda la operación XOR (OR exclusivo)? XOR tiene como entradas $\mathbf{X} = (0,0), (0,1), (1,0), (1,1)$ y salidas $\mathbf{Y} = 0, 1, 1, 0$

3.3 Perceptrón Multicapa (MultiLayer Perceptron)

MLP extiende, de manera natural, la lógica del perpcetrón, agregando los siguientes cambios:

- Combina perceptrones, organizando las neuronas por capas. Así, tenemos una o más capas ocultas que transforman la entrada para facilitar la clasificación.
- Cada neurona está dotada de una función derivable no lineal. Algunas funciones utilizadas son:
 - Sigmoidal:

$$f(x) = \frac{1}{1 + e^{-x}} \quad f'(x) = f(x)(1 - f(x)) \quad (3.20)$$

- Tangente Hiperbólica:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad f'(x) = 1 - f^2(x) \quad (3.21)$$

- ReLU:

$$f(x) = \max\{0, x\} \quad f'(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{en otro caso} \end{cases} \quad (3.22)$$

La Figura 3.4 muestra un esquema de una MLP compuesta por una capa oculta y una capa de salida.

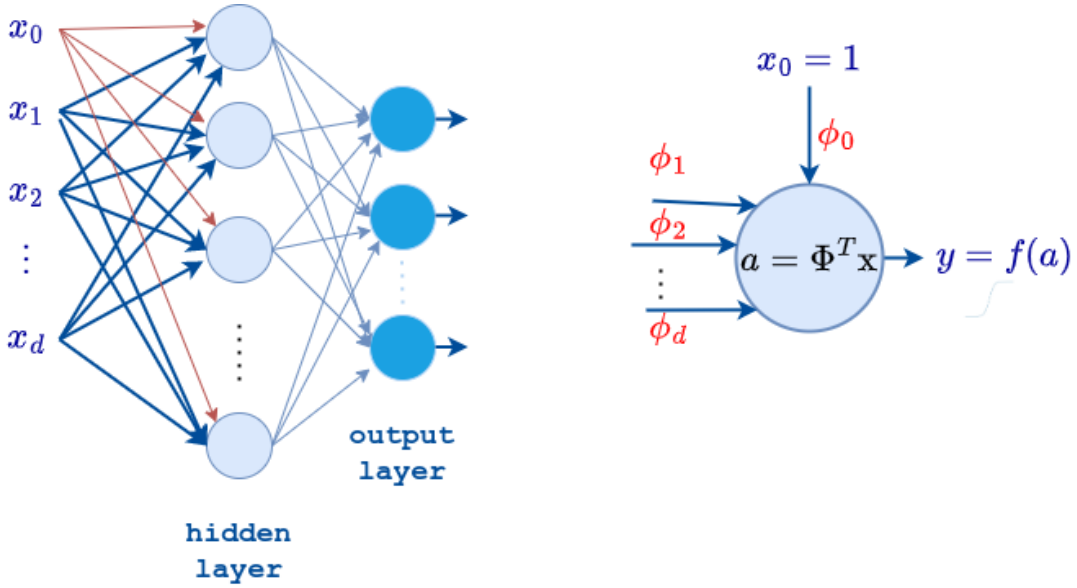


Figure 3.4: Esquema de una MLP

La capas ocultas procesan las entradas capa tras capa, de modo que las salidas resultantes sean fácilmente clasificadas por la capa de salida.

Comúnmente, en un problema de clasificación la cantidad de neuronas en la salida es igual a la cantidad de clases. Hay una excepción para el caso binario, en el que una neurona al final puede ser suficiente.

Cada neurona actúa como un perceptrón independiente, recibiendo entradas y generando salidas. El comportamiento de cada perceptrón está definido por sus parámetros.

¿Cuál es el número de parámetros aprendibles en una MLP con 1 capa oculta de H neuronas y una capa de salida de K neuronas?

3.3.1 Función de Pérdida

Una de las funciones de pérdida más famosas en problemas de clasificación (k-clases) es **Cross-Entropy Loss**, la cual se define de la siguiente manera:

$$CE(y, \hat{y}) = -\log[\text{softmax}(\hat{y}^{i^*})] \quad (3.23)$$

donde, y , $\hat{y}$ son distribuciones de la salida deseada y la inferida por el modelo, respectivamente. En particular, y es un representación *one-hot code*. Además, i^* es el índice de la clase correcta. Pero, ¿de dónde viene esta función de pérdida?, veamos un poco de detalles al respecto.

Le función CE se deduce de la función de **Entropía** utilizada en teoría de información, que popularizó **Claude Shannon** en el siglo pasado.

As3, dada una distribuci3n P , la entrop3a de P , denotada como $\mathbf{H}(P)$, se define como:

$$\mathbf{H}(P) = \sum_{i=1}^{|P|} p_i \log \left(\frac{1}{p_i} \right) \quad (3.24)$$

En el mundo de la codificaci3n de se3ales (como el usado en sistemas compresi3n de datos), Shannon demostr3 que la Entrop3a representa una cota inferior de cualquier sistema de codificaci3n. Por ejemplo, pensemos en la codificaci3n de un texto. Si representamos el tama3o del c3digo de cada s3mbolo de un texto como el logaritmo del inverso de la probabilidad de ocurrencia del s3mbolo, estaremos codificando con c3digos largos a aquellos s3mbolo que tienen poca probabilidad de ocurrir, y con c3digos m3s peque3os a los s3mbolos que ocurren con mayor probabilidad. ¿Tiene sentido?

Ahora, supongamos que nos equivocamos y usamos una distribuci3n diferente a P para calcular los largos de la codificaci3n, digamos que usamos Q . Entonces la Entrop3a queda como:

$$\mathbf{H}(P, Q) = \sum_{i=1}^{|P|} p_i \log \left(\frac{1}{q_i} \right) \quad (3.25)$$

lo anterior es lo que llamamos Entrop3a Cruzada (Cross-Entropy). Seg3n la Teor3a de Shannon, se cumple que:

$$H(P, P) \leq H(P, Q) \quad (3.26)$$

En nuestro caso, P representa la distribuci3n correcta y Q la inferida. En un proceso de aprendizaje, el objetivo es ir acercando Q hacia P . Por lo tanto, tenemos:

$$CE(y, \hat{y}) = \mathbf{H}(y, \hat{y}) = \sum_{i=1}^k y_i \log \left(\frac{1}{\hat{y}_i} \right) \quad (3.27)$$

donde y es una representaci3n **one-hot** de la clase i^* (solamente tenemos un 1 en el c3digo). Por lo tanto, la expresi3n anterior queda:

$$CE(y, \hat{y}) = \mathbf{H}(y, \hat{y}) = \log \left(\frac{1}{\hat{y}_{i^*}} \right) = -\log(\hat{y}_{i^*}) \quad (3.28)$$

Nota: La funci3n softmax es aplicada para garantizar que la salida de la MLP corresponda a una distribuci3n de probabilidades.

3.3.2 Backpropagation

Backpropagation es, tal vez, el algoritmo m3s famoso bajo el cual las redes neuronales son entrenadas. Este algoritmo se basa en el Gradiente Descendente (GD) para actualizar (ajustar) los par3metros aprendibles de una red. La regla general del m3todo GD es:

$$\Phi = \Phi - \alpha \nabla \mathcal{L}(\Phi) \quad (3.29)$$

donde α , la tasa de aprendizaje, controla la velocidad de cambio de los par3metros.

La idea de **backpropagation** es aplicar dos etapas para actualizar los parámetros. La primera es la etapa **forward**, que calcula el error dada las entradas. Esto es, calcular la salida de la red, dada una entrada y estimar el error o *loss*. La segunda etapa es la etapa **backward**, que parte de la capa final y va actualizando el valor de cada parámetro, hasta llegar a la primera capa. Este proceso es conocido como *backpropagation*, dado que el error o pérdida se tiene que propagar hacia atrás para estimar su variación con respecto a cada uno de los parámetros.

La estrategia de partir desde la capa final, es reutilizar las derivadas calculadas en neuronas de capas posteriores, para ser utilizada en el cálculo de derivadas en neuronas anteriores. Entendamos un poco más de este tema, calculado cómo actualizar los valores de dos tipos de parámetros. Para ello observemos la Figura 3.5, donde se muestra parte de la estructura de una MLP de varias capas.

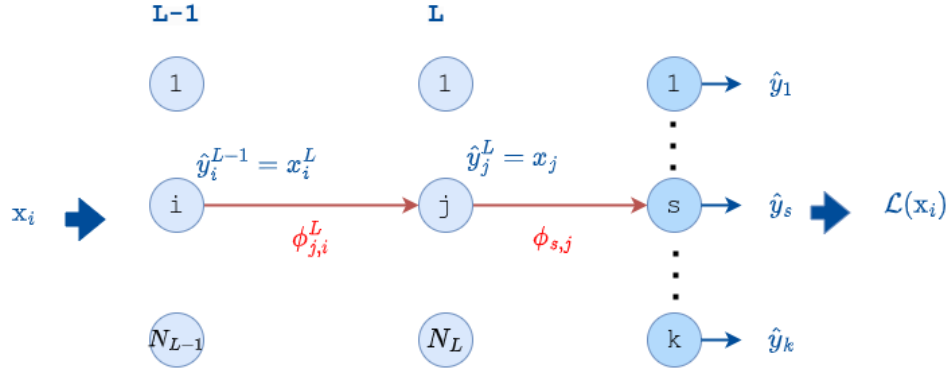


Figure 3.5: Esquema de una MLP con múltiples capas y los parámetros asociados a cada una de ellas.

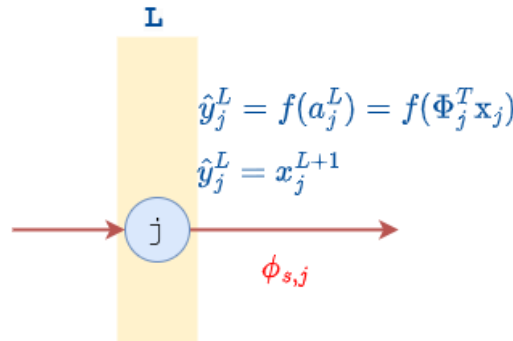


Figure 3.6: Proceso Forward.

- **Caso 1 - Parámetros de capa final** - Este es el caso más simple, pues la capa final o capa de salida afecta directamente a la función de pérdida $\mathcal{L}(\cdot)$. Como ejemplo, calculemos $\frac{\partial \mathcal{L}}{\partial \phi_{s,j}}$. Igual como en casos anteriores, haremos uso de la regla de cadena

para facilitar el cálculo de la derivada, veamos:

$$\frac{\partial \mathcal{L}}{\partial \phi_{s,j}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_s} \frac{\partial \hat{y}_s}{\partial a_s} \frac{\partial a_s}{\partial \phi_{s,j}} \quad (3.30)$$

$$\frac{\partial \mathcal{L}}{\partial \phi_{s,j}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_s} \hat{y}'_s x_{s,j} \quad (3.31)$$

- **Caso 2 - Parámetros de capas intermedias** - Cuando los parámetros sobre los que se deriva se encuentran en capas ocultas, el proceso es un poco más complejo, pero siguiendo la misma idea anterior podemos estimarla. Por ejemplo, calculemos $\frac{\partial \mathcal{L}}{\partial \phi_{j,i}^L}$. Este es un parámetro que se encuentra en la capa L -ésima, justo antes de la capa de salida.

$$\frac{\partial \mathcal{L}}{\partial \phi_{j,i}^L} = \frac{\partial \mathcal{L}}{\partial \hat{y}_j^L} \frac{\partial \hat{y}_j^L}{\partial a_j^L} \frac{\partial a_j^L}{\partial \phi_{j,i}^L} \quad (3.32)$$

La Eq. 3.32 es muy similar a Eq. 3.30, pero hay una importante diferencia. En el caso de Eq. 3.30, $\frac{\partial \mathcal{L}}{\partial \hat{y}_s}$ se puede calcular directamente, dado que $\hat{y}_s$ afecta directamente al error. En el caso de Eq. 3.32, $\frac{\partial \mathcal{L}}{\partial \hat{y}_j^L}$ no es directo, dado que $\partial \hat{y}_j^L$ afecta a las neuronas de las capas siguiente, que consecuentemente afectarán el valor de pérdida. Además debemos notar que la salida de la neurona j de la capa L se convierte en la j -ésima entrada en las neuronas de la capa $L+1$. Por lo tanto se cumple que:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_j^L} = \sum_{i=1}^{N_{L+1}} \frac{\partial \mathcal{L}}{\partial x_{i,j}^{L+1}} \quad (3.33)$$

$$= \sum_{i=1}^{N_{L+1}} \frac{\partial \mathcal{L}}{\partial a_i^{L+1}} \frac{\partial a_i^{L+1}}{\partial x_{i,j}^{L+1}} \quad (3.34)$$

$$= \sum_{i=1}^{N_{L+1}} \frac{\partial \mathcal{L}}{\partial a_i^{L+1}} \phi_{i,j}^{L+1} \quad (3.35)$$

$$= \sum_{i=1}^{N_{L+1}} \frac{\partial \mathcal{L}}{\partial \hat{y}_i^{L+1}} \frac{\partial \hat{y}_i^{L+1}}{\partial a_i^{L+1}} \phi_{i,j}^{L+1} \quad (3.36)$$

$$= \sum_{i=1}^{N_{L+1}} \frac{\partial \mathcal{L}}{\partial \hat{y}_i^{L+1}} \hat{y}'_i^{L+1} \phi_{i,j}^{L+1} \quad (3.37)$$

Si llamamos $\gamma_i^{L+1} = \frac{\partial \mathcal{L}}{\partial \hat{y}_i^{L+1}}$, tenemos la siguiente expresión:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_j^L} = \sum_{i=1}^{N_{L+1}} \gamma_i^{L+1} \hat{y}'_i^{L+1} \phi_{i,j}^{L+1} \quad (3.38)$$

Además, conociendo que $\frac{\partial \mathcal{L}}{\partial \hat{y}_j^L} = \gamma_j^L$, tenemos:

$$\gamma_j^L = \sum_{i=1}^{N_{L+1}} \gamma_i^{L+1} \hat{y}'_i^{L+1} \phi_{i,j}^{L+1} \quad (3.39)$$

La expresión anterior indica que una dependencia factorial de γ^L con respecto a γ^{L+1} . Así, si desarrollamos γ^{L+1} tendríamos:

$$\gamma_j^L = \sum_{i=1}^{N_{L+1}} \left(\sum_{z=1}^{N_{L+2}} \gamma_z^{L+2} \hat{y}_z^{L+2} \phi_{z,i}^{L+1} \right) \hat{y}_i^{L+1} \phi_{i,j}^{L+1} \quad (3.40)$$

Es decir, el ajuste de un parámetro depende multiplicativamente de las derivadas de las funciones de activación de las capas siguientes. **¿Qué ocurre si la derivada de la función de activación es un valor muy pequeño, menor que 1?**

3.3.3 Ejemplo

Recordemos el ejemplo de dígitos manuscritos con el conjunto de datos MNIST. Ahora, crearemos un red MLP para resolver dicha tarea. Nuestra arquitectura podría quedar como la que se observa en la Figura 3.7

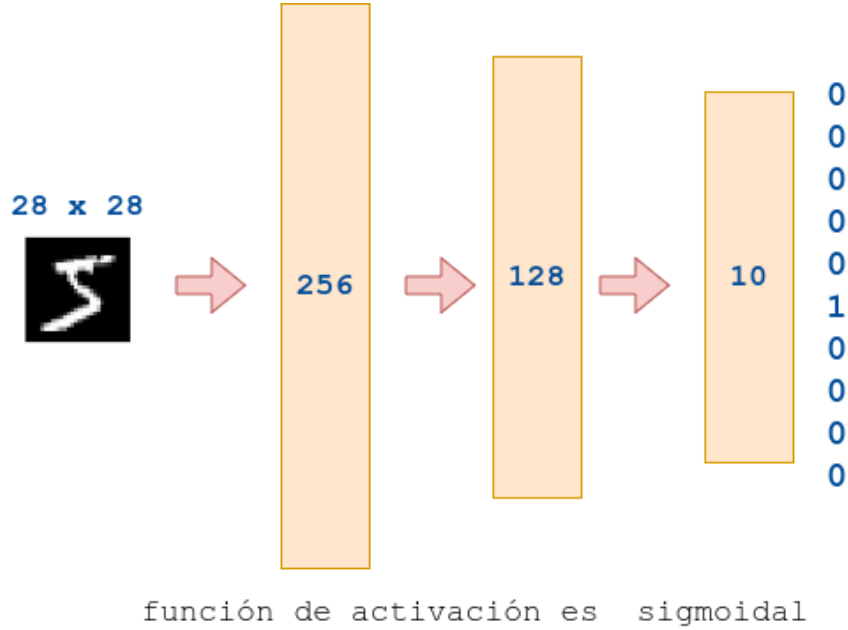


Figure 3.7: Arquitectura de una MLP para MNIST.

3.3.4 Implementación (Keras)

Para implementar la arquitectura definida en Figura 3.7 haremos uso de las librerías *Tensorflow* y *Keras*. Así, el Código 3.2 muestra la implementación general de una MLP. Una instancia particular para el caso descrito se muestra en Código 3.3, en donde se incluye los pasos para:

- Cargar datos
- Crear modelo
- Entrenar modelo
- Predecir salidas

Código 3.2: MLP

```

1  import tensorflow as tf
2
3  class MLP(tf.keras.Model) :
4      #defining components
5      def __init__(self, layer_size, n_classes):
6          super(MLP, self).__init__()
7          self.layer_list = []
8          for lsize in layer_size:
9              self.layer_list.append(tf.keras.layers.Dense(lsize))
10             self.classifier = tf.keras.layers.Dense(n_classes)
11
12     # defining architecture
13     def call(self, inputs):
14         x = inputs
15         for mlp_layer in self.layer_list:
16             x = mlp_layer(x)
17             x = tf.keras.activations.sigmoid(x)
18
19         x = self.classifier(x)
20         x = tf.keras.activations.softmax(x)
21         return x

```

Código 3.3: Instancia de una MLP para MNIST

```

1  """
2  This is an example using MNIST dataset with our own customized MLP
3  """
4
5  import tensorflow as tf
6  import numpy as np
7  import nn.mlp as mlp
8  import metrics.metrics as metrics
9  import tensorflow.keras.datasets as datasets
10
11
12  # loading dataset
13  (x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()
14  print ('{} {}'.format(x_train.shape, x_train.dtype))
15  print ('{} {}'.format(x_test.shape, x_train.dtype))
16  print ('{} {}'.format(y_train.shape, y_train.dtype))
17  print ('{} {}'.format(y_test.shape, y_train.dtype))
18
19  n_train = x_train.shape[0]
20  n_test = x_test.shape[0]
21
22  #reshape the images to represent 1D arrays -- feature vectors --
23
24  x_train = np.reshape(x_train, (n_train, -1))
25  x_test = np.reshape(x_test, (n_test, -1))
26
27  #converting labels to one-hot encoding
28  y_train_one_hot = tf.keras.utils.to_categorical(y_train)#to_one_hot(y_train)
29  y_test_one_hot = tf.keras.utils.to_categorical(y_test)
30
31  x_train = x_train.astype(np.float32)
32  x_test = x_test.astype(np.float32)

```

```

33
34 mu = np.mean(x_train, axis = 0)
35
36 # normalize data, just centering
37 x_train = (x_train - mu)
38 x_test = (x_test - mu)
39
40 # create the model
41 mlp = mlp.MLP([256, 128], 10)
42 input_vector = tf.keras.Input(x_train.shape)
43 mlp(input_vector)
44 mlp.summary()
45
46 # defining optimizer
47 opt = tf.keras.optimizers.SGD()
48
49 # put all together
50 mlp.compile(
51     optimizer=opt,
52     loss='categorical_crossentropy',
53     metrics=['accuracy'])
54
55 # training or fitting
56 mlp.fit(x_train,
57         y_train_one_hot,
58         batch_size=1000,
59         epochs = 100,
60         validation_data = (x_test, y_test_one_hot))
61
62 # prediction using directly the trained model
63 # there is also a function called -- predict -- , you can check it
64 y_pred = mlp(x_test, training = False)
65
66 # computing confusion_matrix
67 mc = metrics.confusion_matrix(y_test, y_pred, 10)
68
69 # print mc
70 print(mc)

```

Para el caso de MNIST, el Código 3.3 arroja el resultado mostrado en Figura 3.8.

3.4 Algunos Términos Nuevos

- **Batch:** Cuando el conjunto de de datos de entrenamiento es muy grande, entrenar usando todos los datos para estimar el cambio de los pesos es ineficiente. Un estrategia para aliviar este problema es dividir el conjunto en *batches* de cierto tamaño, y estimar el cambio de los parámetros usando la información de cada *batch*. Para evitar sesgos, la división del conjunto se realiza en forma aleatoria. Esta estrategia aplicada a GD se conoce como SGD (*Stochastic Gradient Descent*).
- **Iteración** (*step*): Cada vez que pasa un *batch* por la red y se actualizan los parámetros se conoce como iteración o *step*.
- **Época** (*epoch*): Una época es cuando todo el conjunto de datos ha pasado completamente por la red. Así, una época es igual al número de iteraciones necesarias para completar el tamaño del conjunto de entrenamiento. Por ejemplo, si el conjunto de

```

Epoch 91/100
60/60 [=====] - 0s 8ms/step - loss: 0.2821 - accuracy: 0.9238 - val_loss: 0.3169 - val_accuracy: 0.9099
Epoch 92/100
60/60 [=====] - 0s 7ms/step - loss: 0.2808 - accuracy: 0.9241 - val_loss: 0.3160 - val_accuracy: 0.9099
Epoch 93/100
60/60 [=====] - 0s 8ms/step - loss: 0.2795 - accuracy: 0.9244 - val_loss: 0.3150 - val_accuracy: 0.9103
Epoch 94/100
60/60 [=====] - 0s 8ms/step - loss: 0.2782 - accuracy: 0.9246 - val_loss: 0.3141 - val_accuracy: 0.9100
Epoch 95/100
60/60 [=====] - 0s 8ms/step - loss: 0.2769 - accuracy: 0.9250 - val_loss: 0.3132 - val_accuracy: 0.9099
Epoch 96/100
60/60 [=====] - 0s 8ms/step - loss: 0.2757 - accuracy: 0.9252 - val_loss: 0.3123 - val_accuracy: 0.9105
Epoch 97/100
60/60 [=====] - 0s 7ms/step - loss: 0.2744 - accuracy: 0.9255 - val_loss: 0.3115 - val_accuracy: 0.9109
Epoch 98/100
60/60 [=====] - 0s 7ms/step - loss: 0.2732 - accuracy: 0.9255 - val_loss: 0.3106 - val_accuracy: 0.9113
Epoch 99/100
60/60 [=====] - 0s 8ms/step - loss: 0.2721 - accuracy: 0.9258 - val_loss: 0.3098 - val_accuracy: 0.9113
Epoch 100/100
60/60 [=====] - 0s 8ms/step - loss: 0.2709 - accuracy: 0.9261 - val_loss: 0.3090 - val_accuracy: 0.9112
[[ 956  0  2  2  1  8  8  1  2  0]
 [  0 1105  1  4  1  2  3  2 17  0]
 [ 11  6 912 16 17  1 13 13 36  7]
 [  4  0 22 905  1 34  2 19 17  6]
 [  1  3  6  0 916  1 13  2  5 35]
 [ 16  2  6 45  9 754 17 10 24  9]
 [ 16  3  7  0 10 19 899  1  3  0]
 [  1 12 24  7  6  1  1 938  1 37]
 [  6  9 16 27 15 29 12 14 837  9]
 [ 13  6  5 11 45 10  0 24  5 890]]

```

Figure 3.8: Resultado de MLP para MNIST

datos es de tamaño 1000 y el *batch* es de 100, entonces cada época es igual a 10 iteraciones. Es decir, cada 10 iteraciones se cumple una época.

Teorema de Aproximación Universal

El Teorema de Aproximación Universal establece que una red neuronal con una capa oculta puede aproximar cualquier función continua para entradas dentro de un rango específico.

3.5 Modelos Convolucionales

3.5.1 Convolución

La **convolución** es una de las operaciones más comunes en procesamiento de imágenes, cuyo papel es resaltar algunas características de una imagen de entrada. Por ejemplo, usamos la operación de convolución para eliminar ruido en una imagen (resaltando zonas homogéneas) o para detectar bordes.

La convolución es una operación lineal entre un operador llamado *kernel* $\mathbf{W}$ y una imagen de entrada $\mathbf{I}$. Esta operación se define como:

$$\mathbf{G}(y, x) = \mathbf{W} * \mathbf{I} = \sum_{v=-r_y}^{r_y} \sum_{u=-r_x}^{r_x} \mathbf{I}(y+v, x+u) \mathbf{W}(v, u), \quad (3.41)$$

por comodidad las posiciones de $\mathbf{W}$ están centradas, es decir la posición (0,0) está en el centro del *kernel*. Así, las filas varían entre $-r_y$ y r_y , y las columnas entre $-r_x$ y r_x . Lo anterior supone que el número de filas y columnas del *kernel* es un número impar. Aunque lo anterior, no es estrictamente necesario, facilita la operabilidad. Las Figuras ?? muestra un ejemplo del proceso de convolución.

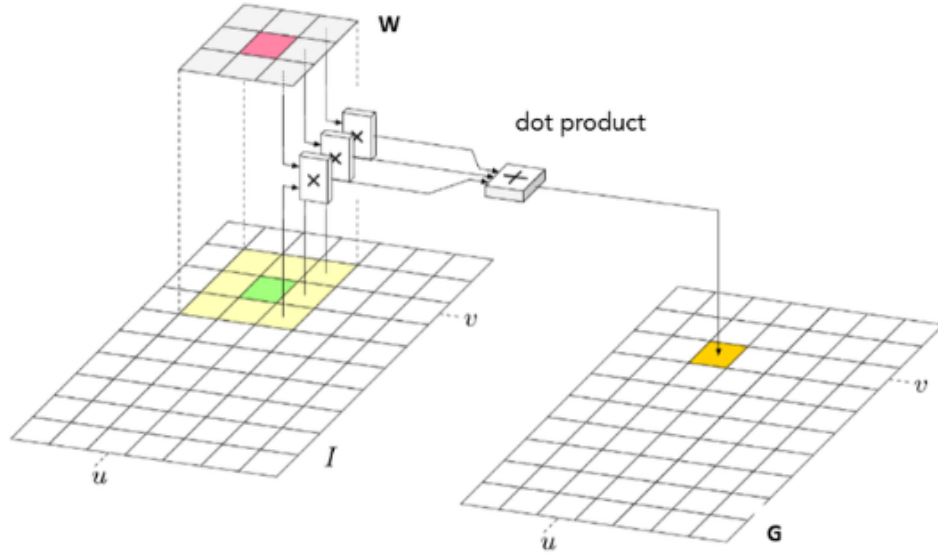


Figure 3.9: Operación de Convolución

	0	1	2	3	4	5			0	1	2	3	4	5
0	1	0	1	1	0	1								
1	0	1	1	0	0	1								
2	1	1	0	1	0	0								
3	0	0	1	1	1	0								
4	1	1	0	0	0	1								
	I													

	-1	0	1
-1	0	1	0
0	1	1	1
1	0	1	0
	$W_{3 \times 3}$		

	0	1	2	3	4	5
0						
1		3	3	3	1	
2		3	4	2	2	
3		3	2	4	2	
4						
	$W * I$					

Figure 3.10: Ejemplo del proceso de convolución.

Por ejemplo, para el caso de eliminación de ruido podemos usar un filtro Gaussiano como el que se muestra en la Eq. 3.42. Para el caso de detección de bordes, usamos filtros que aproximan el gradiente de una imagen, dado que los bordes se definen como posiciones donde ocurren discontinuidades con respecto al color. En este último caso, uno de los filtros más utilizados es el de **Sobel**, cuyas aproximaciones de los gradientes en x e y se muestran en la Eq. 3.43. La Figura 3.11 ilustra el resultado de la aplicación del operador de convolución para eliminar ruido (primera fila) y detectar bordes (segunda fila).

$$\frac{1}{16} \times \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \quad \frac{1}{273} \times \begin{pmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{pmatrix} \quad (3.42)$$

$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \quad (3.43)$$

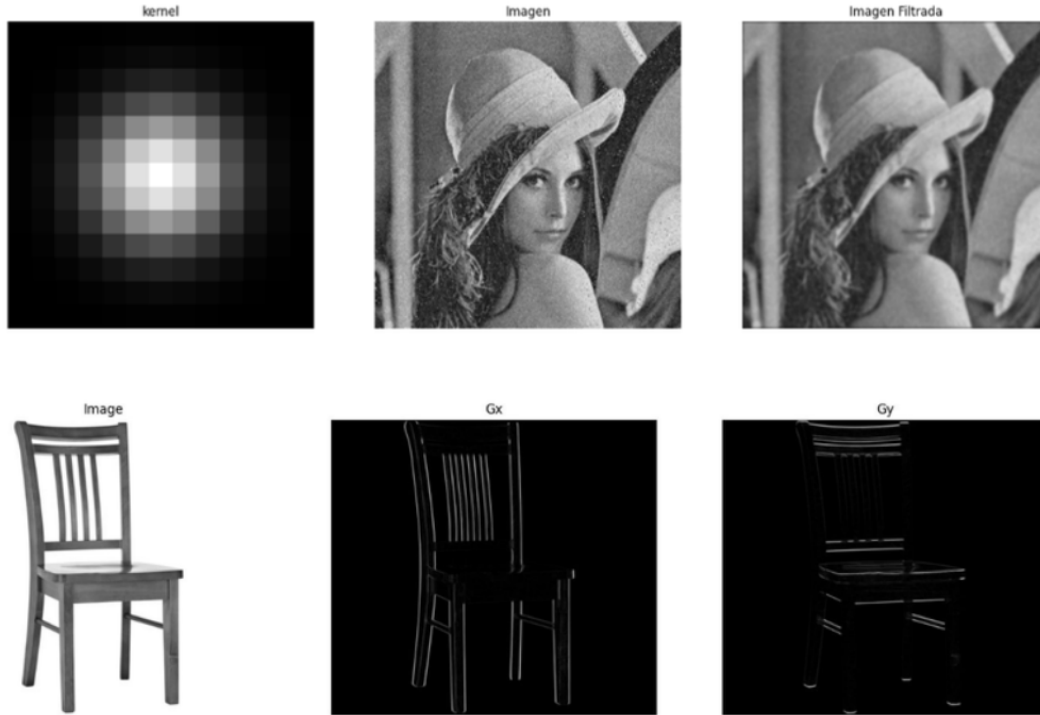


Figure 3.11: Ejemplo de aplicación del operador de convolución. La primera fila muestra una operación de eliminación de ruido, mientras que el segundo muestra una operación de detección de bordes.

En términos generales, debemos entender a la operación de convolución como un operador de **extracción de características**. La característica a extraer depende del **kernel** utilizado.

3.5.2 Red Neuronal Convolutional

Una red neuronal convolucional (CNN) expande la idea de una MLP, de modo que la red esté equipada con componentes de extracción de características, a través de operaciones de convolución. Sin embargo, la capacidad más importante de una CNN es que esta red es capaz de aprender los *kernels* más apropiados para la tarea subyacente. Así, una red convolucional incluye, en su primera parte, un exttactor de características **aprendible**, compuesto por capas convolucionales.

Una CNN está relacionada, de alguna manera, con el modelo biológico, descrito por los científicos Hubel y Wiesel (1961) luego de su investigación en el proceso de percepción

visual. La Figura 3.12 ilustra el comportamiento de una red neuronal biológica, encargada de extraer, a través de sus primeras capas, información de una escena para luego tomar alguna acción de alto nivel como reconocer los objetos presentes en la escena. Según Hubel y Wiesel, este proceso involucra un sistema neuronal jerárquico en el que podemos notar células simples, en las primeras capas, y células más complejas en las capas posteriores.

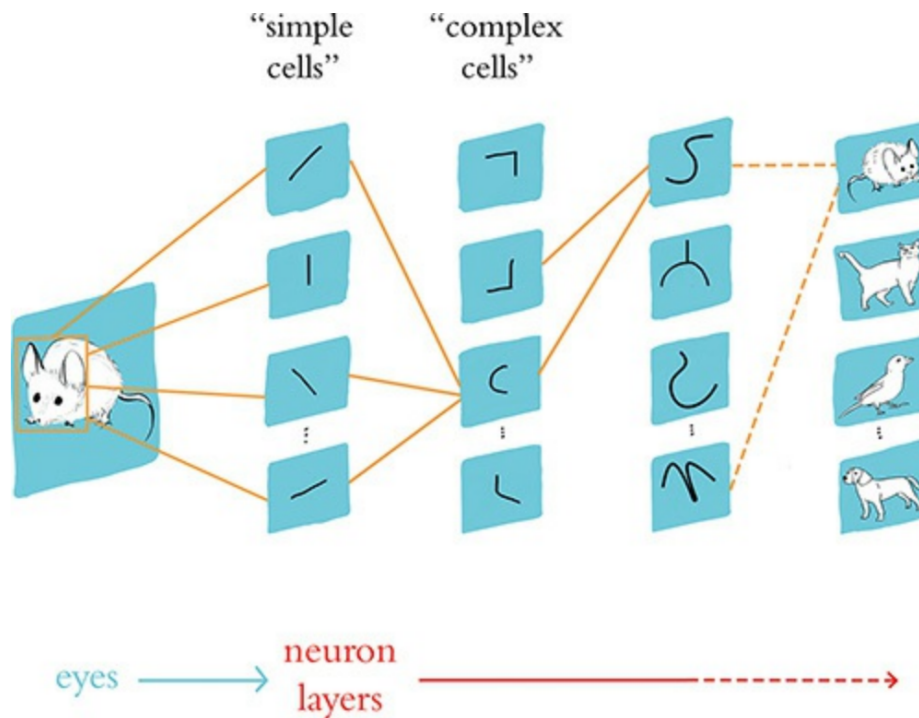


Figure 3.12: CNN y su relación con el modelo biológico

3.5.3 Arquitectura General

La Figura 3.13 muestra el esquema general de una CNN. Aquí se muestra una simple arquitectura de red convolucional (CNN) compuesta por capas convolucionales (al inicio) y capas densas (al final). Recordemos que las capas densas son las que definen a una red MLP (sección 3.3).

En el proceso *forward* cada capa recibe un tensor (generalmente de dimensionalidad 3) y devuelve otro tensor.

Un tensor T de rango 3 es representado por 3 ejes $H \times W \times C$. Cuando procesamos en batch (SGD) añadimos una cuarta dimensión al inicio. Así, el tensor T queda representado como $B \times H \times W \times C$, donde B es el tamaño de batch.

3.5.4 Capa Convolutiva

Es la capa responsable de la extracción de características, a través de un conjunto de *kernels* aprendibles.

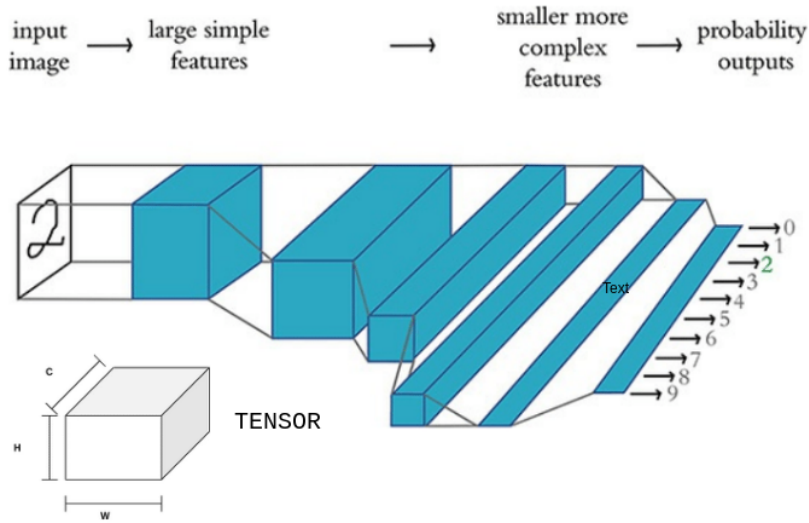


Figure 3.13: Esquema general de una CNN. En la parte inferior se muestra un tensor con 3 dimensiones, cada una representando H : alto, W : ancho y C : número de canales.

La Figura 3.14 muestra la estructura de una capa convolucional. Sin pérdida de generalidad, supondremos que tenemos *kernels* cuadrados de tamaño $k \times k \times n_{in}$, donde n_{in} es la cantidad de canales de entrada. Así, para una tensor de entrada de $h \times w \times n_{in}$, y un *kernel* de $k \times k \times n_{in}$, una capa convolucional genera un canal de salida de tamaño $h \times w$ (este tamaño puede variar ligeramente dependiendo de cómo se procesen los bordes).

Cada *kernel* representa un extractor de alguna característica, definida por los valores del *kernel*. Sin embargo, una sola característica es insuficiente para representar apropiadamente una entrada, necesitamos más características. Por lo tanto, una capa convolucional define n_{out} características, es decir n_{out} *kernels*. Lo anterior implica que el resultado de la capa convolucional es un conjunto de n_{out} canales de tamaño $h \times w$, el cual queda representado por un tensor de salida de tamaño $h \times w \times n_{out}$ (ver Figura 3.15).

Notemos que la operación de convolución procesa cada punto del tensor de entrada en una localidad o vecindad de $k \times k$. Además, se define un parámetro *bias* b por cada canal, tal como se muestra en la Figura 3.14.

Una capa convolucional con *kernel* de tamaño $k \times k$ requiere aprender $(k \times k \times n_{in} + 1) \times n_{out}$ parámetros.

3.5.5 Capa de Pooling (agregación)

En una imagen existe poca variabilidad en un localidad pequeña. Por lo tanto, es eficiente combinar (pooling) tal información y reducir el tamaño espacial de un tensor.

Una capa de *pooling* realiza una operación de agregación sobre un tensor de entrada. En particular, se calcula un valor de agregación (p.ej. suma, máximo o promedio) para cada punto de un canal con respecto a su vecindad y este valor calculado reemplaza el valor del punto subyacente. Esta operación genera valores muy similares entre puntos vecinos, por lo tanto podemos descartar posiciones, por ejemplo saltando una coordenada en cada

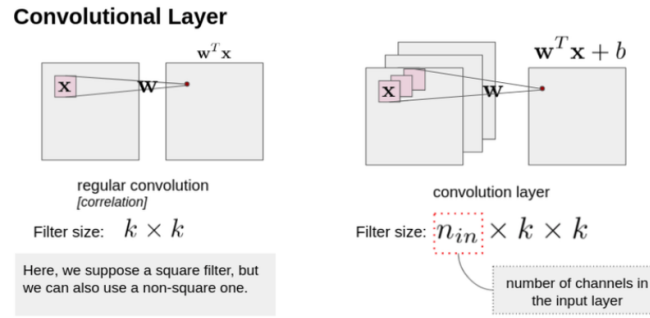


Figure 3.14: Estructura de la capa convolucional. Aquí, se muestra un filtro de tamaño $k \times k$.

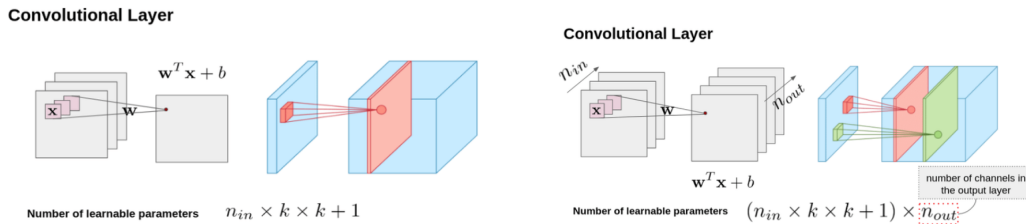


Figure 3.15: Comportamiento de la capa convolucional. El número de canales de salida es el número de *kernels* a aprender.

dimensión. Este salto es parametrizable y se conoce como *stride*. La Figura 3.16 ilustra la estructura de una capa de *pooling*.

Así, si usamos un $stride = 2$, la operación de pooling recorre cada punto saltando 2 posiciones en cada dimensión espacial. Por lo tanto, si un canal de entrada es de tamaño $h \times w$ el resultado de la agregación será un canal de tamaño $h/2 \times w/2$.

La capa de **pooling** no tienen parámetros aprendibles.

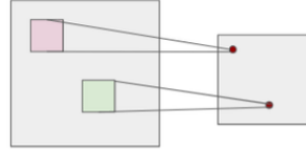
3.5.6 Capa Densa (MLP)

Una capa densa, o también conocida como capa *fully connected*, es la capa utilizada en una MLP. Esta capa no tiene dimensión espacial, solamente número de canales, donde cada canal contiene una sola neurona. La Figura 3.17 ilustra la estructura de una capa densa.

La entrada a esta capa es un tensor sin dimensión espacial, por ejemplo de tamaño n_{in} , el cual representa el número de canales de entrada. La capa densa genera un tensor de tamaño n_{out} , que sería la cantidad de canales de salida.

El número de parámetros aprendibles en una capa densa es $(n_{in} + 1) \times n_{out}$. ¿Por qué sumamos 1 a n_{in} ?

Pooling Layer



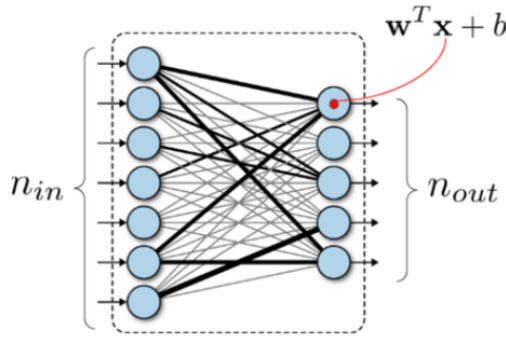
pooling = It locally aggregates the network's responses.

Parameters (Hyperparameters):

- Pooling Type : AVG, MAX
- Regions Size (kernel size) : Size of the local region
- Stride: It will affect the output size. A stride = 2 will produce an output reduced to half (spatially).
- **No learnable parameters**

Figure 3.16: Estructura de una capa de *pooling*.

Fully-Connected Layer (Dense Layer)



Parameters :

- Size : Number of neurons n_{out}

Number of learnable parameters:

$$(n_{in} + 1) \times n_{out}$$

Figure 3.17: Estructura de una capa densa.

3.5.7 Normalización de Capas

Normalizar la salida de una capa es tan importante como normalizar la entrada. Recordemos que normalizar la entrada de un modelo significa hacer que las diferentes características tengan el mismo rango de variación. En una red neuronal, las salidas de una capa también pueden mostrar variaciones que afectan el entrenamiento. Para manejar tales variaciones se han propuesto estrategias que permiten normalizar las salidas de una capa de una red neuronal.

En general, la idea es estimar la media μ y la desviación estándar σ , y normalizar los datos bajo la siguiente expresión:

$$\hat{X} = \frac{X - \mu}{\sigma} \quad (3.44)$$

$$\text{norm}(X) = \gamma \hat{X} + \beta \quad (3.45)$$

$$(3.46)$$

donde γ y β son parámetros aprendibles, y agregan flexibilidad a la normalización.

Batch Normalization

En este caso, los estadísticos μ y σ se calculan en la dirección del *batch*. Así, cada característica (canal) estima sus estadísticas con respecto al *batch* de entrada. La Figura

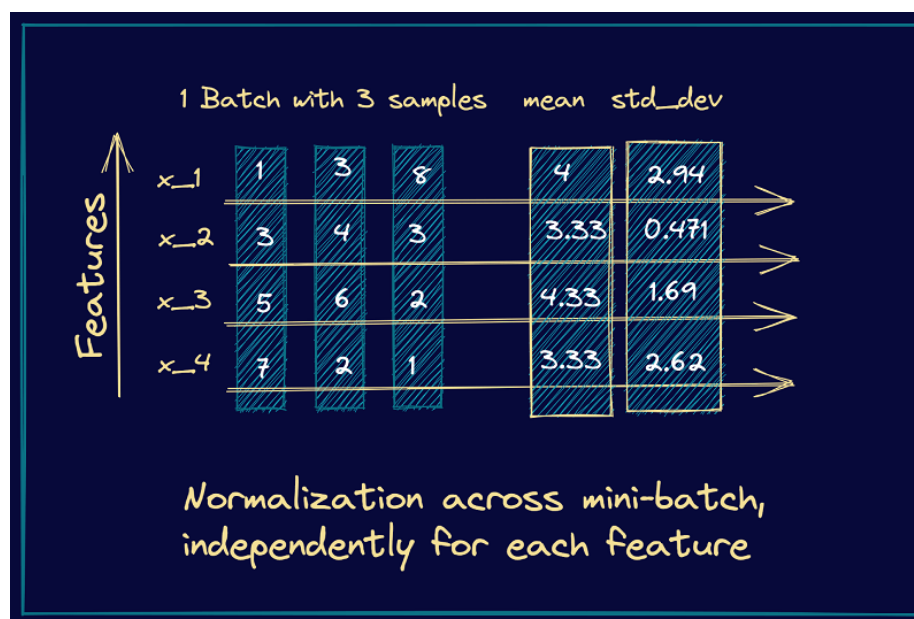


Figure 3.18: Esquema de Batch Normalization (BN)

3.18 ilustra el cálculo de las estadísticas con respecto al *batch*.

BatchNorm requiere 2 parámetros (γ y β) aprendibles por cada canal de la capa subyacente. Además, es necesario mantener los estadísticos μ y σ para la fase de predicción.

Layer Normalization

En este caso, la normalización no depende del *batch*. Los estadísticos se calculan independientemente para cada entrada. Así, μ y σ se estiman en dirección de las características como muestra la Figura 3.19. Además, la Figura 3.20 ilustra al dirección del cálculo de los estadísticos μ y σ para el caso de redes convolucionales.

LayerNorm requiere 2 parámetros (γ y β) aprendibles por cada canal de la capa subyacente. A diferencia de BatchNorm, LayerNorm no necesita mantener los estadísticos μ y σ , ya que estos dependen de la entrada.

3.5.8 Una simple red convolucional

La Figura 3.21 muestra la arquitectura de una red convolucional para la tarea de clasificación de dígitos manuscritos MNIST. Las capas convolucionales se especifican con 3 parámetros, los dos primeros definen el tamaño del *kernel*, y último indica en número de canales de salida. En el caso de la capa de *pooling*, los dos primeros parámetros indican el tamaño de la vecindad utilizada para calcular el valor agregado por cada punto, y el tercer parámetro es el *stride*. Las capas densas se definen por el número de canales que generan.

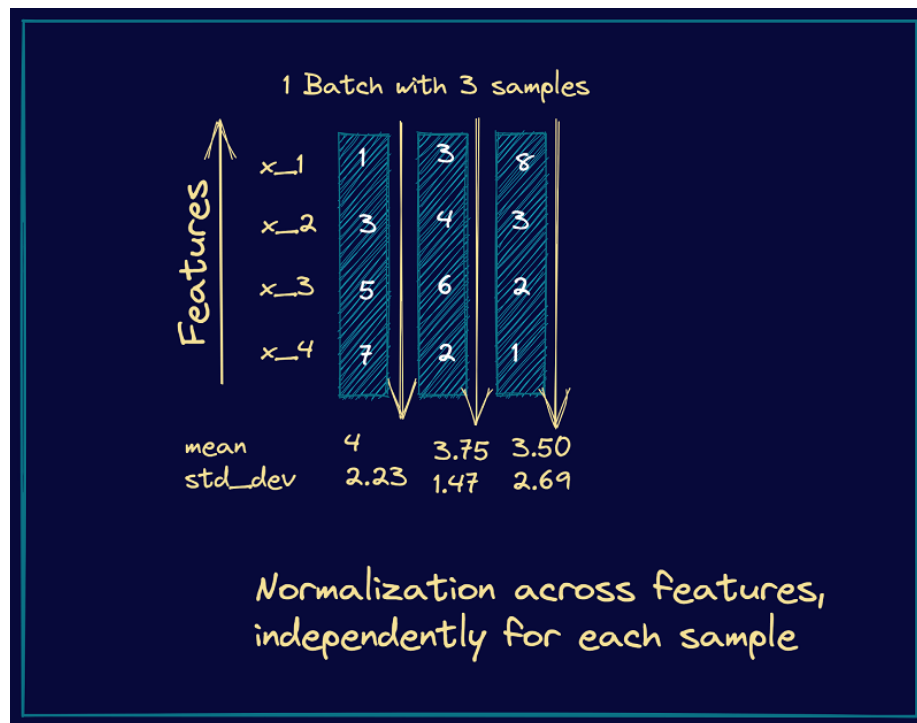


Figure 3.19: Esquema de Layer Normalization (LN)

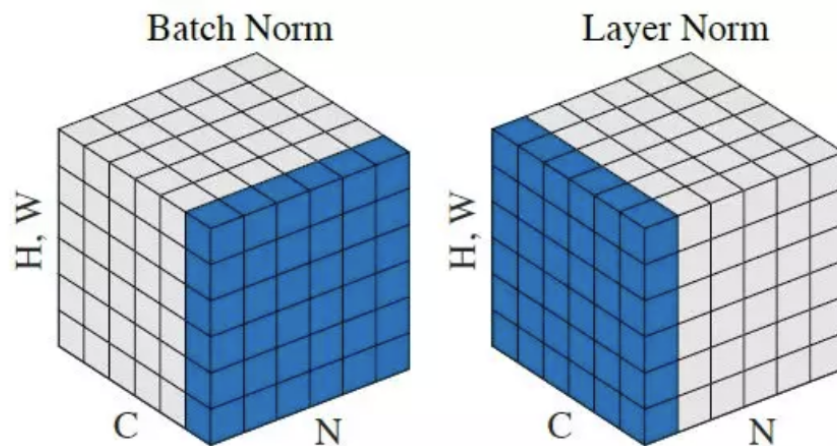


Figure 3.20: Normalización en capas convolucionales.

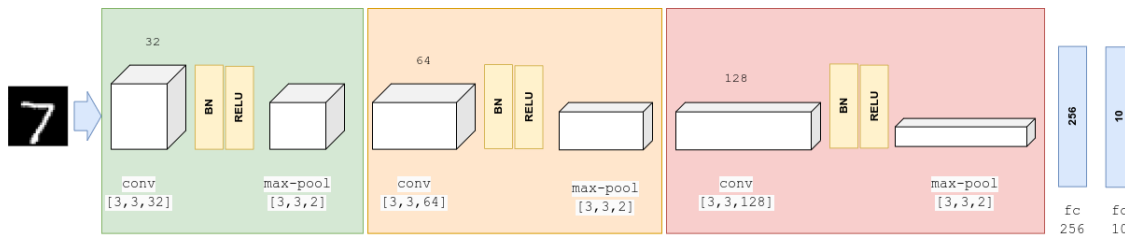


Figure 3.21: Una simple arquitectura convolucional para MNIST.

3.5.9 Implementación (una simple convnet)

El código 3.4 muestra la implementación de la arquitectura CNN de la Figura 3.21. Además, un ejemplo de cómo entrenar el modelo en MNIST se muestra en el Código 3.5.

Código 3.4: Simple ConvNet

```

1  """
2  A simple cnn used for training mnist.
3  """
4  import tensorflow as tf
5
6
7  class SimpleModel(tf.keras.Model):
8      def __init__(self, number_of_classes):
9          super(SimpleModel, self).__init__()
10         self.conv_1 = tf.keras.layers.Conv2D(32, (3,3), padding = 'same',
11         kernel_initializer = 'he_normal')
12         self.max_pool = tf.keras.layers.MaxPool2D((3,3), strides = 2,
13         padding = 'same')
14         self.relu = tf.keras.layers.ReLU();
15         self.bn_conv_1 = tf.keras.layers.BatchNormalization()
16         self.conv_2 = tf.keras.layers.Conv2D(64, (3,3), padding = 'same',
17         kernel_initializer='he_normal')
18         self.bn_conv_2 = tf.keras.layers.BatchNormalization()
19         self.conv_3 = tf.keras.layers.Conv2D(128, (3,3), padding = 'same',
20         kernel_initializer='he_normal')
21         self.bn_conv_3 = tf.keras.layers.BatchNormalization()
22         self.fc1 = tf.keras.layers.Dense(256, kernel_initializer='he_normal'
23         )
24         #self.bn_fc_1 = tf.keras.layers.LayerNormalization()
25         self.bn_fc_1 = tf.keras.layers.BatchNormalization()
26         self.fc2 = tf.keras.layers.Dense(number_of_classes)
27
28     # here, connecting the modules
29     def call(self, inputs):
30         #first block
31         x = self.conv_1(inputs)
32         x = self.bn_conv_1(x)
33         x = self.relu(x)
34         x = self.max_pool(x)    #14 X 14
35         #second block
36         x = self.conv_2(x)
37         x = self.bn_conv_2(x)
38         x = self.relu(x)
39         x = self.conv_3(x)
40         x = self.bn_conv_3(x)
41         x = self.relu(x)
42         x = self.fc1(x)
43         x = self.bn_fc_1(x)
44         x = self.fc2(x)
45         return x

```

```

34         x = self.max_pool(x)    #7X7
35         #third block
36         x = self.conv_3(x)
37         x = self.bn_conv_3(x)
38         x = self.relu(x)
39         x = self.max_pool(x)    #4X4
40         #last block
41         x = tf.keras.layers.Flatten()(x)
42         x = self.fc1(x)
43         x = self.bn_fc_1(x)
44         x = self.relu(x)
45         x = self.fc2(x)
46         x = tf.keras.activations.softmax(x)
47         return x

```

Código 3.5: Ejemplo ConvNet

```

1  """
2  This is an example using MNIST dataset with our own customized MLP
3  """
4
5  import tensorflow as tf
6  import numpy as np
7  import convnet.simple as simple
8  import metrics.metrics as metrics
9  import tensorflow.keras.datasets as datasets
10
11
12  # loading dataset
13  (x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()
14  print ('{} {}'.format(x_train.shape, x_train.dtype))
15  print ('{} {}'.format(x_test.shape, x_test.dtype))
16  print ('{} {}'.format(y_train.shape, y_train.dtype))
17  print ('{} {}'.format(y_test.shape, y_test.dtype))
18
19  n_train = x_train.shape[0]
20  n_test = x_test.shape[0]
21
22  #reshape the images to represent BxHxWxC
23  x_train = x_train.astype(np.float32)
24  x_test = x_test.astype(np.float32)
25
26  x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], x_train.
    shape[2], 1))
27  x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], x_test.shape
    [2], 1))
28
29  #converting labels to one-hot encoding
30  y_train_one_hot = tf.keras.utils.to_categorical(y_train)#to_one_hot(y_train)
31  y_test_one_hot = tf.keras.utils.to_categorical(y_test)
32
33  x_train = x_train.astype(np.float32)
34  x_test = x_test.astype(np.float32)
35
36
37  mu = np.mean(x_train, axis = 0)
38

```

```
39 # normalize data, just centering
40 x_train = (x_train - mu)
41 x_test = (x_test - mu)
42
43 # create the model
44 model = simple.SimpleModel(10)
45 input_vector = tf.keras.Input(x_train.shape[1:])
46 model(input_vector)
47 model.summary()
48
49 # defining optimizer
50 opt = tf.keras.optimizers.SGD()
51
52 # put all together
53 model.compile(
54     optimizer=opt,
55     loss='categorical_crossentropy',
56     metrics=['accuracy'])
57
58 # training or fitting
59 model.fit(x_train,
60         y_train_one_hot,
61         batch_size=256,
62         epochs = 20,
63         validation_data = (x_test, y_test_one_hot))
64
65 # prediction using directly the trained model
66 # there is also a function called -- predict -- , you can check it
67 y_pred = model(x_test, training = False)
68
69 # computing confusion_matrix
70 mc = metrics.confusion_matrix(y_test, y_pred, 10)
71
72 # print mc
73 print(mc)
74 # mc as percentages
75 rmc = mc.astype(np.float32) / np.sum(mc, axis = 1, keepdims = True)
76 rmc = (rmc * 100).astype(np.int32) / 100
77 print(rmc)
```

3.6 Modelos Atencionales

Atención es uno de los mecanismos modernos en *deep learning*, que ha demostrado un impacto positivo en tareas de **procesamiento de lenguaje natural** (NLP) y **reconocimiento visual** (*computer vision*). Pero ¿qué es el mecanismo de atención? y ¿cómo se aplica? En esta sección trataremos de responder ambas interrogantes.

Atención es un mecanismo natural en nuestro sistema biológico, que nos permite tomar la información más relevante de un contexto para realizar cierta tarea. En **percepción visual** existe un fenómeno conocido como *agrupamiento perceptual* (perceptual grouping), por el cual ciertos elementos de una escena se potencian entre sí para percibirlos como un solo objeto. Por ejemplo, veamos la Figura 3.22, en donde se presentan dos casos. Si observamos la primera fila, es más fácil fijarnos en los elementos por separado. Así, podemos notar el círculo, el cuadrado y las elipses. Sin embargo, si observamos la segunda fila, rápidamente notamos rostros. Esto es el efecto de haber aprendido, desde muy temprano, la morfología y estructura de los rostros.

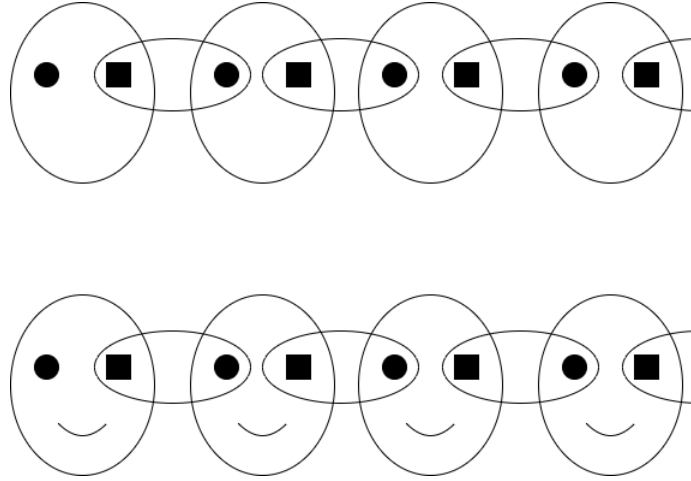


Figure 3.22: Agrupamiento Perceptual (*perceptual grouping*). La segunda fila, rápidamente, nos evoca imágenes de rostros. En la primera fila resulta más fácil ver los objetos por separado.

En términos prácticos, dada una secuencia de representaciones $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$, el mecanismo de atención genera una nueva representación $\mathbf{y}_i$ para cada $\mathbf{x}_i$, relacionándola con el resto de representaciones. Así, $\mathbf{y}_i$ es estimado bajo el siguiente esquema:

$$\mathbf{y}_i = \sum_{j=1}^n f(\Theta(\mathbf{x}_i), \Phi(\mathbf{x}_j)) g(\mathbf{x}_j) \quad (3.47)$$

donde Θ, Φ, g son transformaciones sobre las representaciones de entrada. Además, $f(\cdot, \cdot)$ es una función que mide la relación entre $\mathbf{x}_i$ y $\mathbf{x}_j$.

El modelo de atención más famoso es **Transformer**, inicialmente usado para NLP, y hoy expandido a problemas de visión (visionTransformer-ViT).

3.7 Modelos Atencionales para Texto

Uno de los trabajos más importantes en el área de procesamiento de texto y lenguaje natural es “Attention is all you need” de Vaswani et al. publicado en NeurIPS’17, en donde se describe **Transformer**. Para entender un poco más de este modelo, veamos un ejemplo de aplicación en NLP.

Supongamos que nos encontramos en el contexto de traducción de lenguaje, donde tenemos un texto de entrada (texto origen) y uno de salida (texto traducido). Por ejemplo:

```
<input>: I like machine learning
<output>: Me gusta aprendizaje de máquinas
```

Un modelo de aprendizaje basado en *deep learning*, se compone de dos partes. Un *encoder*, que codifica la entrada, y un *decoder*, que genera el texto de salida, tomando en cuenta la codificación de entrada. En realidad, este esquema es muy común en diversas tareas de aprendizaje, relacionadas con NLP, visión, audio, etc. Un esquema general del modelo *encoder-decoder* se muestra en la Figura 3.24.

Un texto se divide en términos o tokens. Por simplicidad, supondremos que cada token es una palabra. Así, un texto de T tokens se codifica con $E = \{e_1, e_2, \dots, e_T\}$, donde $e_i \in \mathbb{R}^d$. El texto tiene una representación inicial (componentes en azul) a la que se añade **información posicional** (componentes en amarillo). Esta información es la entrada del **encoder**, que generará mejores representaciones (componentes en verde) con respecto a una tarea determinada.

La información posicional se obtiene codificando la posición de cada token con códigos del mismo largo que el largo de los código de los tokens. Una codificación muy trivial sería el **one-hot code** de cada posición. Sin embargo, una codificación que ha funcionado bastante bien es la siguiente:

$$PE(pos, 2i) = \sin(pos/10000^{2i/d}) \quad (3.48)$$

$$PE(pos, 2i + 1) = \cos(pos/10000^{2i/d}) \quad (3.49)$$

La Figura 3.23 ilustra 128 vectores posicionales calculado de 512 dimensiones cada uno.

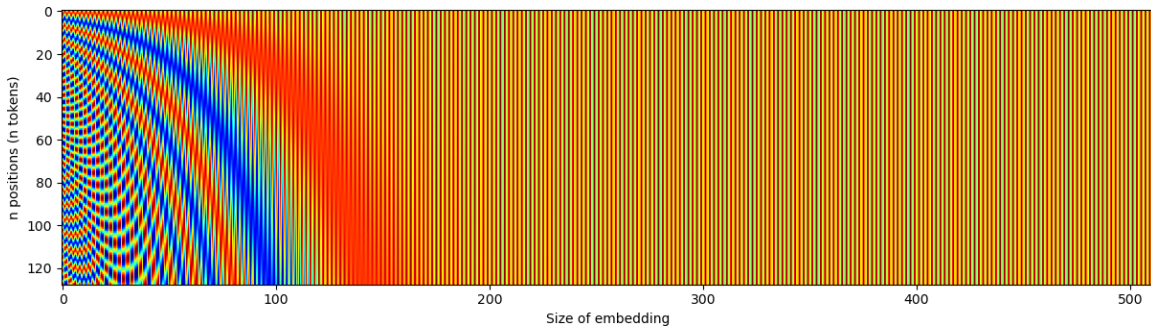


Figure 3.23: Ejemplo de 128 vectores posicionales (positional embeddings) con $d = 512$

Las representaciones generadas por el *encoder* alimentan al *decoder*. Este último irá generando cada uno de las palabras del texto destino. Sin embargo, es crítico que el modelo

también utilice lo que va generando para predecir la siguiente palabra. Así, la representación del texto traducido acompaña a la codificación de la entrada como insumo al *decoder*.

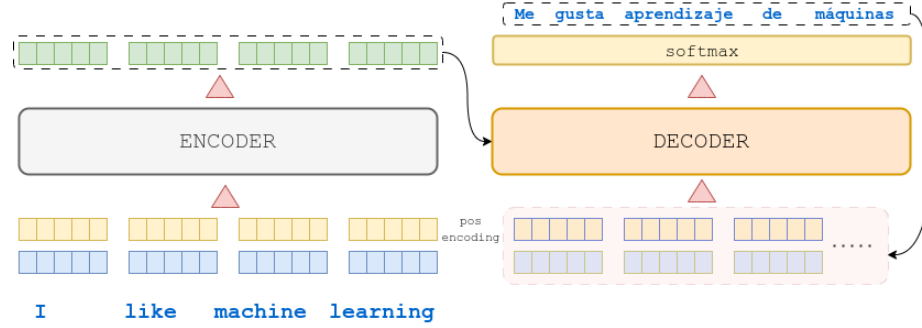


Figure 3.24: Esquema general de un modelo *encoder-decoder* para traducción de texto.

Como hemos mencionado anteriormente, el mecanismo de atención permite generar representaciones en la que cada una toma en cuenta el resto. En el caso del texto, la representación de cada palabra debe considerar las representaciones de las otras palabras presentes en el texto. Sin embargo, el modelo decidirá, a través de un proceso de aprendizaje, cómo impacta una representación en otra. Esto último es el efecto de la atención. Un esquema de lo anterior se muestra en Figura 3.25, que se relaciona con la Eq. 3.47. El término α_i representa el factor de atención.

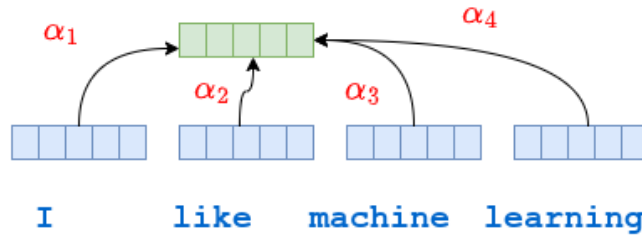


Figure 3.25: Ejemplo del proceso de atención en un texto.

Podemos definir la representación de una secuencia de tokens por una matriz $\mathbf{X}_{n \times d}$, donde n es la cantidad de tokens y d es la dimensionalidad de la representación de cada token. La forma de llevar a la práctica la Eq. 3.47 es a través de la siguiente expresión.

$$Attention(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d}} \right) V, \quad (3.50)$$

donde Q , K y V son transformaciones lineales sobre $\mathbf{X}$. Es decir, $Q = \mathbf{XW}_Q$, $K = \mathbf{XW}_K$ y $V = \mathbf{XW}_V$. Aquí, $\mathbf{W}_Q$, $\mathbf{W}_K$, $\mathbf{W}_V$ son matrices que el modelo debe aprender. En general, estas matrices son de tamaño $d_{in} \times d_{out}$, que permiten transformar $\mathbf{X}$, cambiando su dimensionalidad desde d_{in} a d_{out} . En particular $d_{in} = d_{out} = d$.

En un modelo atencional, los términos Q , K y V vienen de $Q : query$, $K : key$ y $V : value$. Cuando tenemos dos fuentes de entrada, Q debe venir de una de las fuentes y K con V de la otra. El último caso es conocido como atención cruzada (cross attention).

De Eq. 3.50, el producto QK^T relaciona todas las representaciones a través del producto punto, que representa un valor de similitud entre vectores. Esta operación genera una matriz de $n \times n$ indicando la ponderación de cada representación con respecto a las otras. Para transformar las ponderaciones a una distribución de probabilidades se aplica la función *softmax* por cada fila. El término $\sqrt{d}$ es un factor de normalización con respecto a la dimensionalidad de los datos. Finalmente, el resultado de *softmax* es multiplicado por V , generando una matriz de $n \times d$, que será la nueva representación. Un esquema de este proceso se muestra en Figura 3.26.

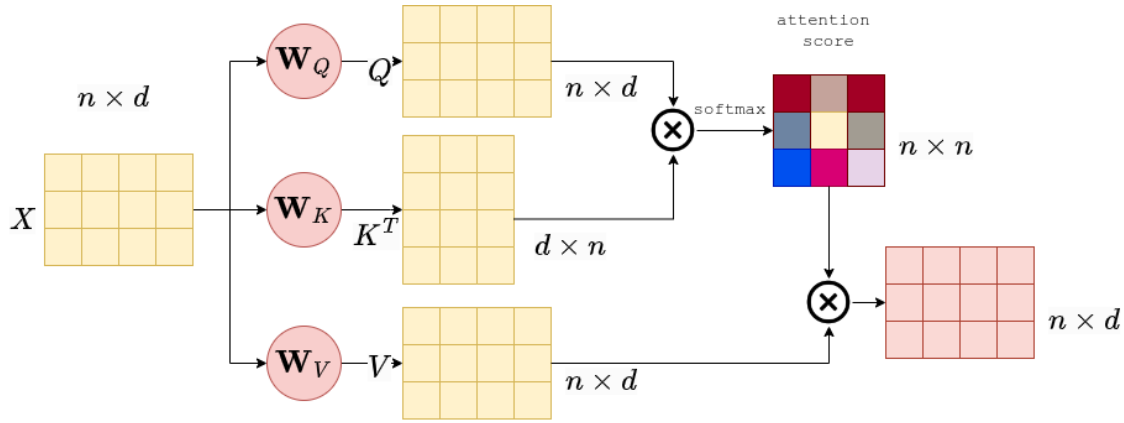


Figure 3.26: Mecanismo de atención entre representaciones.

MultiHead Attention (MHA)

Para dar mayor poder al modelo atencional, podemos realizar diferentes transformaciones sobre una misma entrada y generar diversas formas de atención; esto es conocido como *MultiHead Attention* (MHA). La Figura 3.27 ilustra este proceso. Los resultados de cada atención son concatenados y luego combinados por una capa lineal (e.g. una convolución de 1×1).

Transformer

Con lo anterior, estamos listos para definir el modelo **Transformer** según el esquema de la Figura 3.28. Revisemos la estructura definida para el *encoder* y *decoder* en forma separada:

- **Encoder:** este componente concatena N bloques atencionales. Cada bloque se compone de un módulo MHA, seguido de una MLP (*feed forward*) aplicada a cada representación por separado, pero con los mismos parámetros. Además, el bloque MLP incluye una función no lineal (e.g. ReLU o sus variantes). Las capas MHA y MLP incluyen una capa de normalización, por lo general **LayerNorm**. Además,

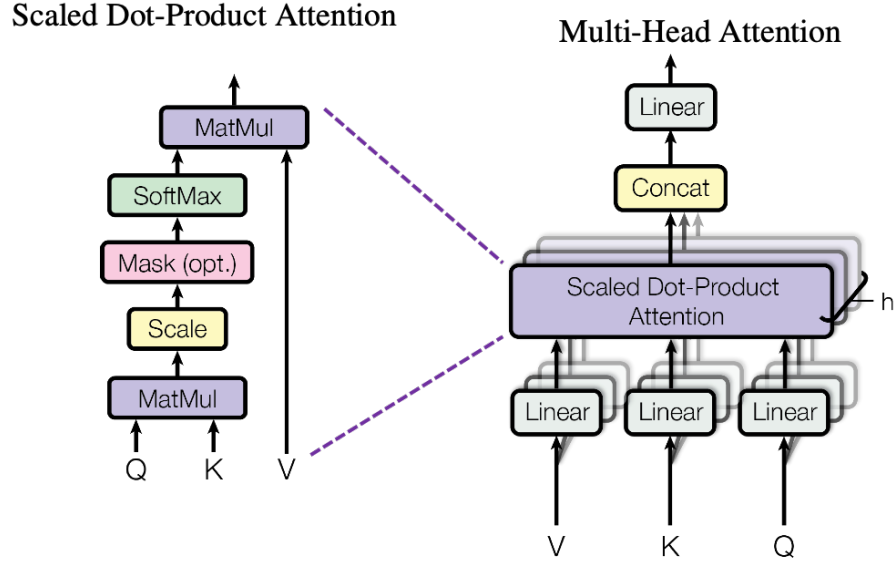


Figure 3.27: MultiHead Attention (MHA).

ambas capas se implementan como capas residuales. Con lo anteriormente descrito, un bloque del *encoder* se define como:

$$Block_E(X) = LayerNorm(Z + MLP(Z)) \quad (3.51)$$

$$Z = LayerNorm(X + MHA(X)) \quad (3.52)$$

Dado lo anterior, podemos formular el algoritmo de atención simple usado en Transformer según Algoritmo 2.

Algorithm 2 Mecanismo de Atención

Require: $\mathbf{X}_{n \times d}$: es la secuencia de tokens con su representación inicial (afectado por el embedding posicional).

$\mathbf{W}_Q \leftarrow$ inicializar matriz $d \times d$

$\mathbf{W}_K \leftarrow$ inicializar matriz $d \times d$

$\mathbf{W}_V \leftarrow$ inicializar matriz $d \times d$

$Q \leftarrow \mathbf{X} \times \mathbf{W}_Q$

$K \leftarrow \mathbf{X} \times \mathbf{W}_K$

$V \leftarrow \mathbf{X} \times \mathbf{W}_V$

$A \leftarrow \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V$

$Z \leftarrow LayerNorm(X + A)$

$Y \leftarrow LayerNorm(Z + MLP(Z))$

return Y

- **Decoder:** este componente es muy similar al *encoder* con N bloques atencionales. La diferencia es que un bloque *decoder* incluye un componente *cross-attention* entre su

MHA y MLP. Así, MHA va codificando el texto generado, que luego es cruzado con las representaciones de la entrada para obtener la probabilidad del siguiente término, en un modelo de generación de texto. En el componente *cross-attention*, K y V provienen del *encoder* y Q del *decoder*.

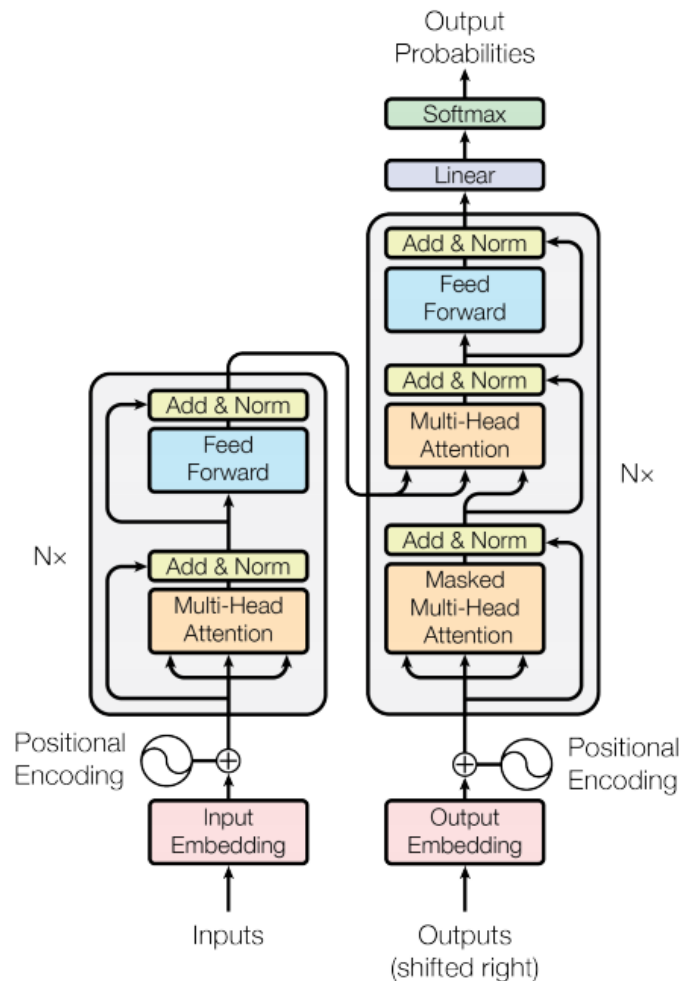


Figure 3.28: Modelo Transformer.

Self-Attention se refiere a generar representaciones atencionales desde una sola fuente. Así Q , K y V provienen de los mismos datos. En el caso de **Cross-Attention**, tenemos dos fuentes de datos, K y V provienen de una y Q de la otra.

3.8 Modelos Atencionales para Imágenes (ViT)

El modelo Transformer ha sido extendido al contexto de visión por computadora. Este modelo visual basado en Transformer es conocido como **Vision Transformer (ViT)**. La idea es casi idéntica al modelo de texto, la única diferencia está en cómo generar tokens

desde una imagen. Para este fin, una idea muy simple ha tenido éxito; se trata de dividir la imagen en pequeños pedazos llamados *patches*. Así, cada *patch* es entendido como un token. Más aún, la codificación inicial de cada *patch* puede ser el patch *per-se*, utilizando sus valores de intensidad. La Figura 3.29 ilustra la estructura de ViT. Seguro, podemos hacer algo más elaborado, pero el enfoque descrito ha demostrado ser suficientemente bueno en diversas tareas de visión.

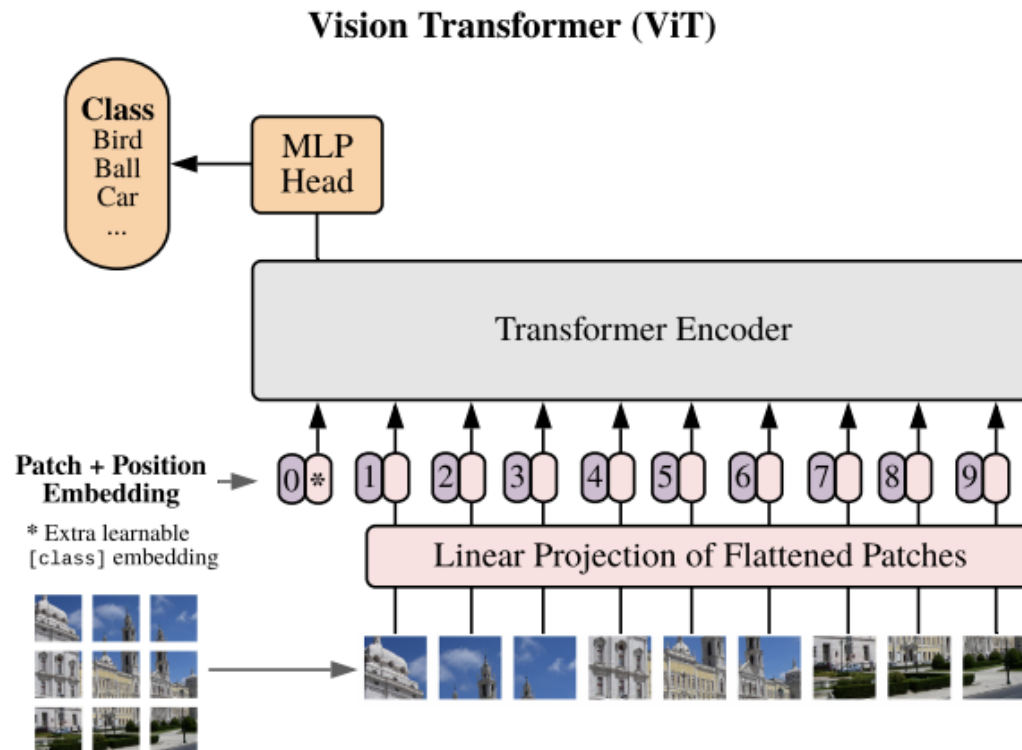


Figure 3.29: Vision Transformer (ViT)

Para generar una representación agregada, se crea una representación *dummy* (MLP head) al inicio, que absorberá la información de las otras representaciones. Así, la representación *dummy*, podrá ser luego utilizada en tareas más específicas.

3.9 Optimización

El optimizador es uno de los componentes críticos en problemas ML, ya que es el encargado del ajuste de los parámetros de un modelo. La elección del optimizador afecta la eficiencia y efectividad. Sin embargo, es importante entender que la mayoría de optimizadores son variaciones del Gradiente Descendente (GD) para facilitar el aprendizaje. Recordemos la regla general de actualización de parámetros bajo el esquema GD:

$$\Phi = \Phi - \alpha \nabla \mathcal{L}_{\Phi} \quad (3.53)$$

3.9.1 Gradiente Descendente Estocástico SGD

En cada iteración del entrenamiento, la actualización de los parámetros se estima usando una porción del conjunto total de entrenamiento, llamado mini-batch. Así, en cada época el conjunto total de datos de entrenamiento se divide en B *batches*, no solapados, cada uno de ellos pasará por la red (forward) y actualizar los pesos de acuerdo con la función de pérdida.

3.9.2 Momentum

En SGD, el momentum acumula un promedio del gradiente pasado, con decaimiento exponencial, y el SGD utiliza este promedio para corregir el gradiente actual.

$$V_t = \rho V_{t-1} + \alpha \nabla \mathcal{L}_{\Theta_{t-1}} \quad (3.54)$$

$$\Theta_t = \Theta_{t-1} - V_t, \quad (3.55)$$

donde $\rho < 1$, generalmente $\rho \in \{0.9, 0.99\}$

3.9.3 Momentum Nesterov (NAG)

NAG es similar a momentum pero el gradiente se calcula con respecto a la posición aproximada futura de los parámetros. Esto es:

$$V_t = \rho V_{t-1} + \alpha \nabla \mathcal{L}_{\Theta_{t-1} + \rho V_{t-1}} \quad (3.56)$$

$$\Theta_t = \Theta_{t-1} - V_t, \quad (3.57)$$

3.9.4 AdaGrad

AdaGrad es un algoritmo adaptativo. Este adapta la tasa de aprendizaje con respecto a cada parámetro. La idea es favorecer parámetros con poca activación, es decir, aquello que ha variado muy poco. Para este fin, mantenemos la suma de los cuadrados de cada parámetro, como lo indica la Eq. 3.58.

$$\mathbf{G}_t = \mathbf{G}_{t-1} + g_{t-1} \odot g_{t-1} \quad (3.58)$$

Así, definimos la actualización de los parámetros como:

$$\Theta_t = \Theta_{t-1} - \frac{\alpha}{\sqrt{\mathbf{G}_t + \epsilon}} \odot g_{t-1}, \quad (3.59)$$

donde $g_t = \nabla \mathcal{L}_{\Theta_t}$ y $\odot$ es el producto elemento a elemento.

3.9.5 RMSProp

Este es una versión mejorada de AdaGrad, el cual usa un promedio con decaimiento exponencial en el cálculo de $\mathbf{G}$. Esto es:

$$\mathbf{G}_t = \rho \mathbf{G}_{t-1} + (1 - \rho) g_{t-1} \odot g_{t-1}, \quad (3.60)$$

donde $\rho \in \{0.9, 0.99\}$

3.9.6 Adaptive Momentum (Adam)

Adam es una combinación de RMSProp con momentum. Este último aplicado a $\mathbf{G}$ como al gradiente. Así, primero obtenemos un promedio del gradiente $\mathbf{S}$ de la siguiente manera:

$$\mathbf{S}_t = \rho_1 \mathbf{S}_{t-1} + (1 - \rho_1) g_{t-1}. \quad (3.61)$$

Luego, estimamos $\mathbf{G}$, en forma idéntica a RMSProp, pero cambiamos el parámetro ρ por ρ_2 . Así, la actualización de pesos queda definida por:

$$\Theta_t = \Theta_{t-1} - \frac{\alpha}{\sqrt{\mathbf{G}_t + \epsilon}} \odot \mathbf{S}_t, \quad (3.62)$$

Finalmente, estimamos la actualización de los parámetros en forma idéntica a AdaGrad. Para reducir el impacto negativo generado por la poca información en etapas tempranas, normalizamos $\mathbf{S}_t$ y $\mathbf{G}_t$ como:

$$\mathbf{S}_t = \frac{\mathbf{S}_t}{1 - \rho_1^t} \quad (3.63)$$

$$\mathbf{G}_t = \frac{\mathbf{G}_t}{1 - \rho_2^t}. \quad (3.64)$$

En este algoritmo los valores sugeridos para ρ_1 y ρ_2 son 0.9 y 0.999, respectivamente.

4. Modelos No-Paramétricos

4.1 Espacio de Características

En los capítulos anteriores hemos resaltado la importancia que tienen las representaciones de los objetos de estudio en los modelos de ML. En efecto, todo problema de aprendizaje se reduce a tener buenas representaciones, comúnmente en forma de vectores reales $v \in \mathbb{R}^d$. Una representación es buena si es efectiva y eficiente para resolver un problema. El término eficiente se refiere a que el vector o la representación debe ser lo más simple posible (baja dimensionalidad) como para facilitar las operaciones subsiguientes. En muchos casos, obtener una buena representación no algo trivial. Por ejemplo, ¿cuál sería una buena representación para la imagen de la Figura 4.1, en el contexto de clasificación de aves?

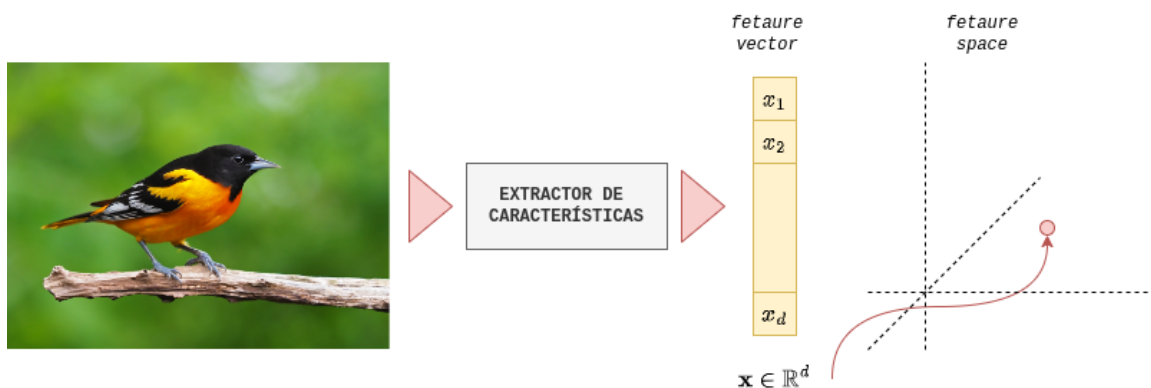


Figure 4.1: Extracción de características en $\mathbb{R}^d$.

El espacio donde caen las representaciones se conoce como espacio de características. Decimos que un espacio es **efectivo** si representa apropiadamente la semántica del problema subyacente. Por ejemplo, la Figura 4.2 muestra un espacio semántico. En este caso, vemos que dos imágenes de perros de la misma raza caen muy cerca en el espacio, mientras que

un perro de otra raza cae un poco más lejos. Además, una imagen de un gato cae más lejos aún.

Una representación efectiva y eficiente es la clave

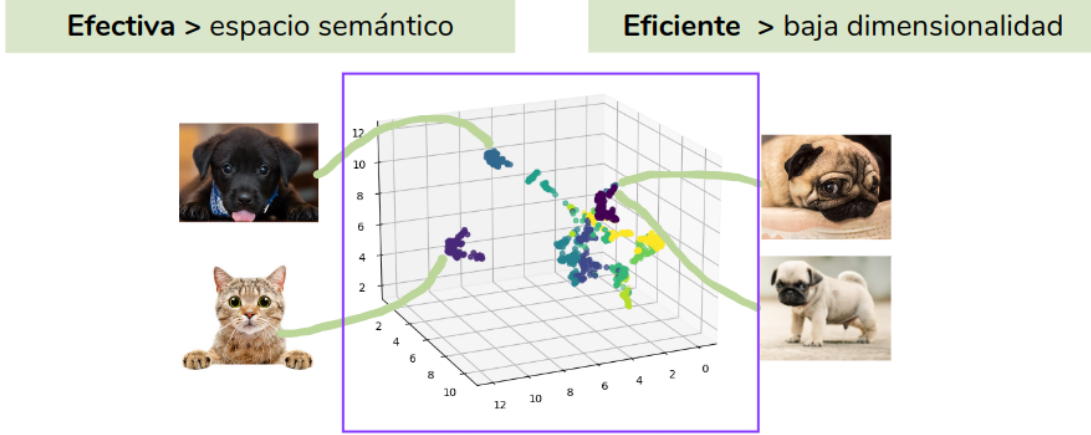


Figure 4.2: Esquema de un espacio de características (*feature space*)

4.2 Comparación entre Vectores

Dado un espacio de características, es crítico definir una función que mida la cercanía entre los vectores. Para este fin, hay dos tipos de funciones. Por un lado tenemos las funciones de distancia que miden qué tan lejos están dos puntos en el espacio y por otro lado, tenemos la funciones de similitud que miden la semejanza entre vectores. En este último caso, la idea es medir la dirección del vector. Así, dos vectores codirigidos se consideran idénticos. La Figura 4.3 ilustra estos dos casos.

4.2.1 Funciones de Distancia

Existe una familia de funciones de distancia muy utilizadas en este dominio, se trata de las distancias de **Minkowski**, definida bajo la siguiente expresión:

$$D(\mathbf{u}, \mathbf{v}) = \left(\sum_{i=1}^d |u_i - v_i|^p \right)^{1/p} \quad (4.1)$$

- Distancia de Manhattan ($p = 1$)

$$D_1(\mathbf{u}, \mathbf{v}) = \sum_{i=1}^d |u_i - v_i| \quad (4.2)$$

- Distancia Euclidiana ($p = 2$)

$$D_2(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^d |u_i - v_i|^2} \quad (4.3)$$

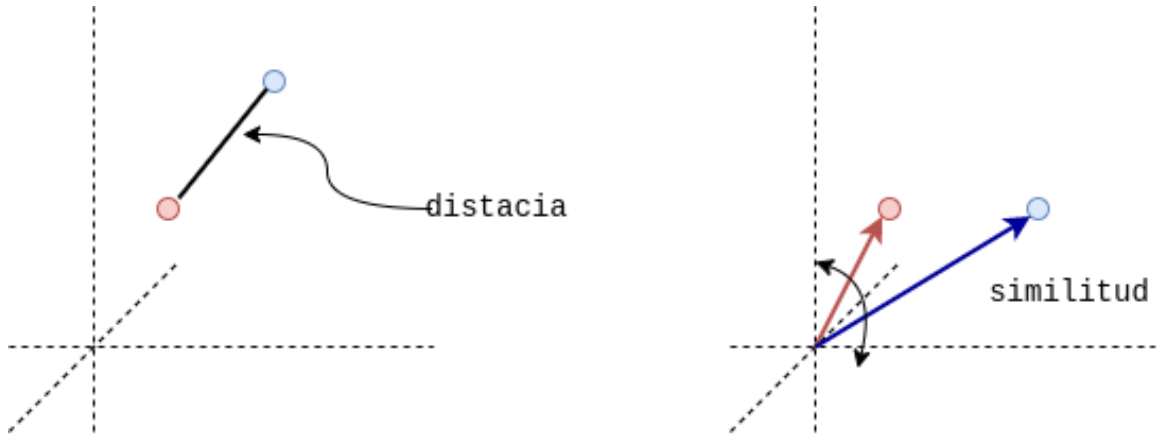


Figure 4.3: Izquierda: distancia entre puntos. Derecha: similitud entre vectores, medido en base al ángulo entre los vectores.

- Distancia Chebyshev ($p = \infty$)

$$D_{\infty}(\mathbf{u}, \mathbf{v}) = \max_i |u_i - v_i| \quad (4.4)$$

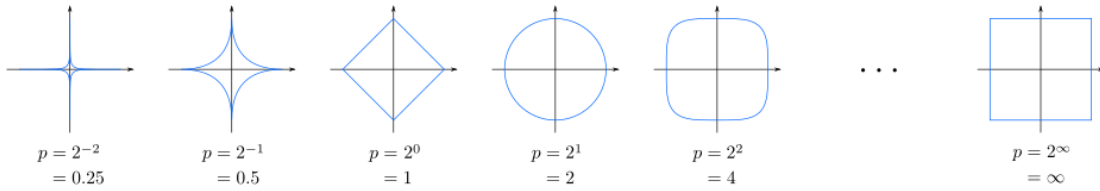


Figure 4.4: Regiones inducidas por diferentes valores de p . Los bordes de las regiones indican, puntos a la misma distancia del centro.

4.2.2 Funciones de Similitud

La función de similitud más usada es la similitud coseno. Este mide la similitud basada en la dirección de los vectores. Así, si ambos tienen la misma dirección y sentido, la similitud es 1. Si tienen sentidos opuestos, la similitud es -1. La forma de implementar esta función es a través de determinar el coseno del ángulo entre los vectores. Si el ángulo es 0, entonces el coseno es 1. Si el ángulo es 90 el coseno es 0, y si es 180 entonces será -1. La forma de calcular esta función está dada por:

$$\text{cossim}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}, \quad (4.5)$$

donde $\|\cdot\|$ define la norma o magnitud del vector. Esto es:

$$\|\mathbf{u}\| = \sqrt{\sum_{i=1}^d u_i^2} \quad (4.6)$$

La idea detrás de la normalización es quitarle importancia a la magnitud del vector. Con lo anterior, ambos $\mathbf{u}$ y $\mathbf{v}$ quedan con la misma magnitud. La Figura 4.5 muestra ejemplo de 3 diferentes casos sobre los que se calcula cossim .

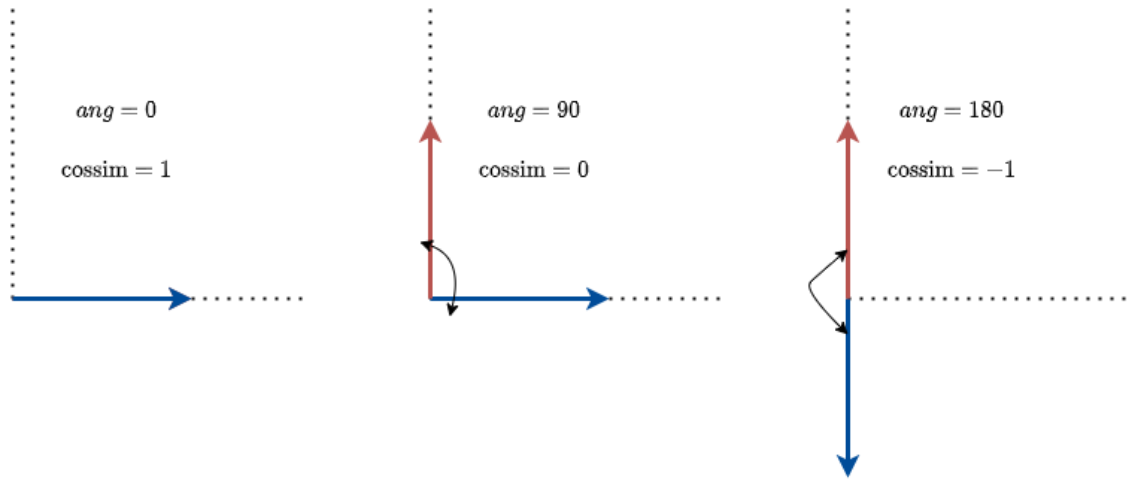


Figure 4.5: Ángulo entre vectores y cossim .

4.2.3 Relación D_2 y cossim

La distancia Euclidiana D_2 y la función cossim son equivalentes, siempre que la norma de los dos vectores subyacentes sea la misma (por ejemplo ambas igual a 1). Observemos la Figura 4.6. Aquí, se muestran dos vectores $\vec{A}$ y $\vec{B}$, que a su vez, definen los puntos A y B , respectivamente. Notemos que mientras el ángulo entre los vectores se reduce, el coseno aumenta aproximándose a 1 (i.e. aumenta la similitud), y la distancia se reduce.

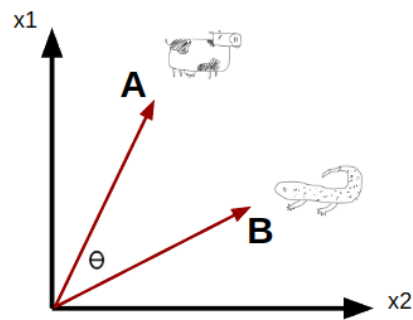


Figure 4.6: Relación distancia D_2 y similitud.

Supongamos que $\|A\| = \|B\| = 1$, entonces tenemos:

$$D_2(A, B) = \sqrt{\sum_{i=1}^d (a_i - b_i)^2} \quad (4.7)$$

$$= \sqrt{\sum_{i=1}^d (a_i^2 - 2a_i b_i + b_i^2)} \quad (4.8)$$

$$= \sqrt{\sum_{i=1}^d a_i^2 - \sum_{i=1}^d 2a_i b_i + \sum_{i=1}^d b_i^2} \quad (4.9)$$

$$= \sqrt{2 - 2 \sum_{i=1}^d a_i b_i} \quad (4.10)$$

$$= \sqrt{2 - 2\cos\theta} \quad (4.11)$$

Es decir, podemos expresar la distancia Euclidiana de dos vectores unitarios en términos del coseno entre dichos vectores.

4.2.4 Efecto *Bursting*

Dado que las funciones de distancia entre vectores se basan en la diferencia entre sus características, es posible que el valor de distancia resultante quede sesgado por grandes diferencias entre pocas características (outliers). Para minimizar este efecto, podemos reducir la magnitud de cada característica a través de una función sublineal como la raíz cuadrada. De esta manera, definimos la función $D_{2<root>}$ como:

$$D_{2<root>}(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^d (\sqrt{u_i} - \sqrt{v_i})^2} \quad (4.12)$$

Pero, hay un problema con el signo, ¿qué pasa si hay valores negativos?. En realidad, lo que nos importa es reducir la magnitud manteniendo el signo. Así reescribimos la función $D_{2<root>}$ como:

$$D_{2<root>}(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^d (\text{sign}(u_i)\sqrt{|u_i|} - \text{sign}(v_i)\sqrt{|v_i|})^2}, \quad (4.13)$$

donde $\text{sign}(\cdot)$ devuelve el signo del argumento.

4.3 kNN

Dado un espacio de características, podemos implementar modelos de clasificación que no requieran entrenamiento. Este es el caso del clasificador kNN (k Nearest Neighbors) que aprovecha representaciones efectivas de objetos conocidos para clasificar objetos nuevos.

Este modelo guarda los vectores de un conjunto de objetos de los cuales sí conocemos sus clases. Así un nuevo objeto será clasificado en base a la clase de sus vecinos más cercanos. Observamos la Figura 4.7

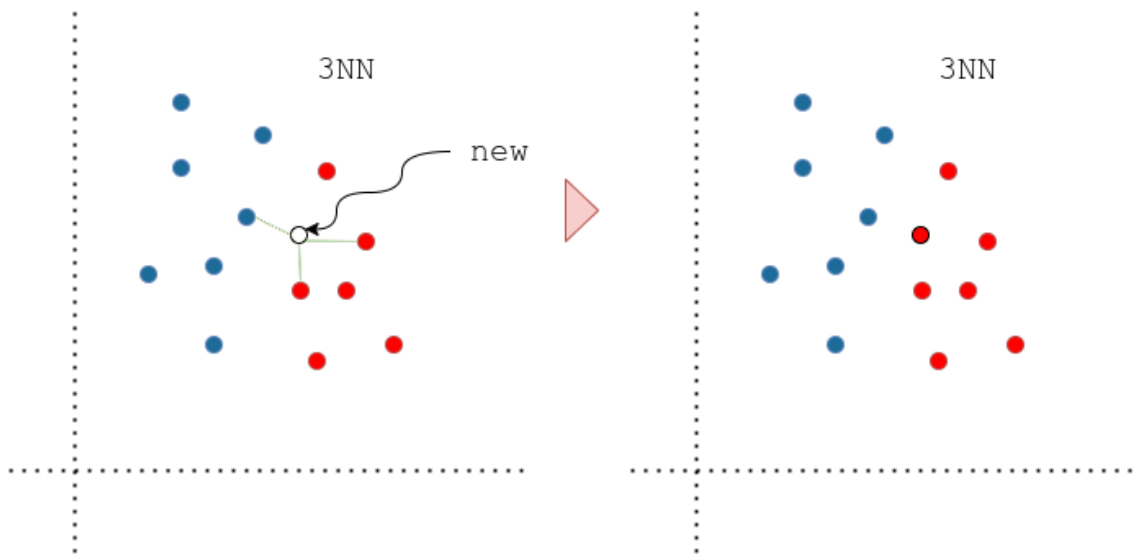


Figure 4.7: Ejemplo de un espacio de características en la clasificación de círculos rojo y azules usando 3NN. Aquí el nuevo elemento será clasificado como rojo, dado que hay más rojo en sus 3 vecinos más cercanos.

4.3.1 Inferencia

Dado un nuevo objeto representado como un vector de características $\mathbf{x} \in \mathcal{R}^d$, el objetivo es clasificarlo de alguna de las clases conocidas. ¿Cómo obtenemos información de los k vecinos más cercanos?

- **Vecino más cercano (NN)**: este es el caso más fácil, pero muy efectivo en muchos problemas. Aquí solo analizamos en vecino más cercano, y tomamos la clase de ese vecino como la clase del nuevo objeto.
- **Votación dura**: en este caso, realizamos una votación de los k vecinos más cercanos. Cada vecino vota por su clase. Finalmente, la clase del nuevo objeto es la clase con mayor votación.
- **Votación suave**: esto es similar al caso anterior, en donde cada vecino vota por su clase. Sin embargo, el voto es ponderado en forma inversa a la distancia con el objeto a clasificar. Así, vecinos más cercanos tienen mayor ponderación que vecinos más lejanos. La Figura 4.8 ilustra este proceso.

4.3.2 Maldición de la Dimensionalidad

Este fenómeno ocurre cuando tratamos de realizar predicciones a partir de datos en alta dimensionalidad. Dos de los factores que afectan las predicciones son **sparsity** y **closeness**.

- **Sparsity**: el volumen del espacio representado crece rápidamente originando muchos espacio vacío sobre el cual no se puede obtener estadísticas. Para cubrir tales espacio requeriríamos un mayor cantidad de datos, que crece exponencialmente con el tamaño del espacio subyacente.
- **Closeness**: en alta dimensionalidad, los puntos están más lejos entre sí. Incluso, los más cercanos se encuentran bien lejos en alta dimensionalidad. Este fenómeno hace que la distancia entre puntos pierda relevancia. Pensemos en la relación entre

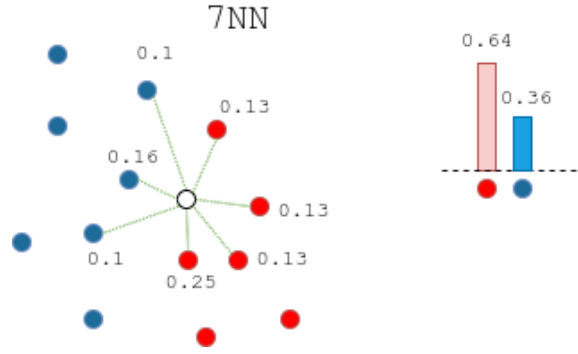


Figure 4.8: Votación suave para inferir la clase de un nuevo objeto mediante kNN.

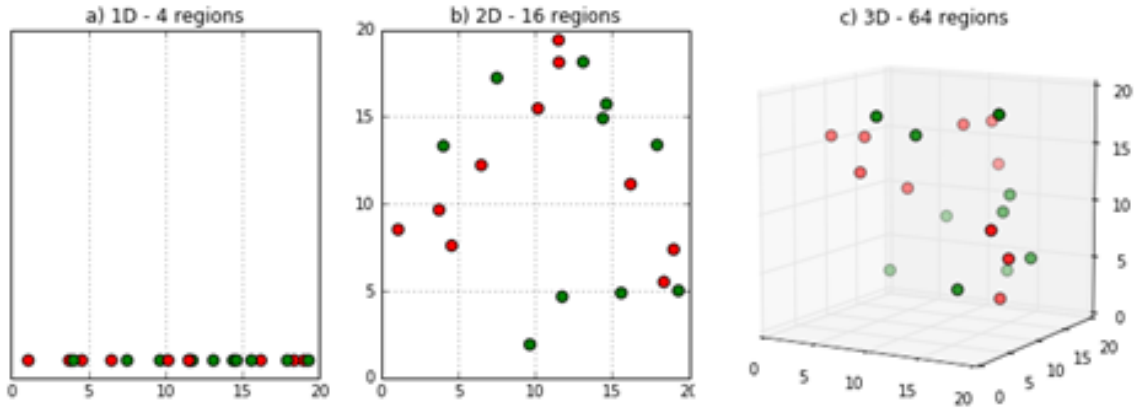


Figure 4.9: Efecto de *sparsity* en relación a la dimensionalidad de los datos.

una hipersfera de radio r en un espacio de D dimensiones y un hipercubo de lado $2r$. Para facilitar la comprensión, pensemos en $r = 0.5$. Además, partamos mirando la relación en $D = 2$ y $D = 3$. En $D = 2$, el área del cuadrado es 1 (lado = $2 \times 0.5 = 1$) y el área del círculo es $\pi r^2 = 0.5^2 \pi = 0.785$. Es decir, el área del círculo corresponde al 78.5% del área del cuadrado. En $D = 3$, medimos volumen. El volumen del cubo es 1, y el volumen de la hipersfera es $4/3 \pi 0.5^3 = 0.52$. Es decir, la hipersfera representa el 52% del hipercubo. Si generalizamos, la relación entre el volumen de una hipersfera con respecto a un hipercubo de radio r tenemos:

$$\frac{V(S)}{V(C)} = \frac{r^D \frac{\pi^{D/2}}{\Gamma(D/2+1)}}{(2r)^D} = \frac{\pi^{D/2}}{2^D \Gamma(D/2+1)} = \left(\frac{\sqrt{\pi}}{2} \right)^D \frac{1}{\Gamma(D/2+1)}, \quad (4.14)$$

donde $\Gamma(x) = (x-1)!$. Cuando D tiende al infinito, la expresión anterior tiende a 0. Es decir, mientras aumentamos la dimensionalidad, el espacio se hace considerablemente más grande con respecto a las regiones que podemos evaluar.

4.3.3 Ejemplo

En este ejemplo, reutilizaremos nuestro modelo convolucional MNIST para clasificar otros símbolos manuscritos. Para este fin utilizaremos el conjunto de datos EMNIST (Extended MNIST)¹. La Figura 4.10 muestra un ejemplo de las letras a clasificar.

Para resolver este problema utilizaremos las características aprendidas en MNIST y analizaremos la capacidad de esas características para resolver un problema diferente pero parecido. La idea es utilizar un clasificador kNN.



Figure 4.10: Ejemplos de las letras manuscritas en EMNIST.

La Figura 4.11 muestra dos ejemplos de recuperación de los 9 vecinos más cercanos en el espacio de MNIST. La primera imagen de cada fila corresponde a la consulta.

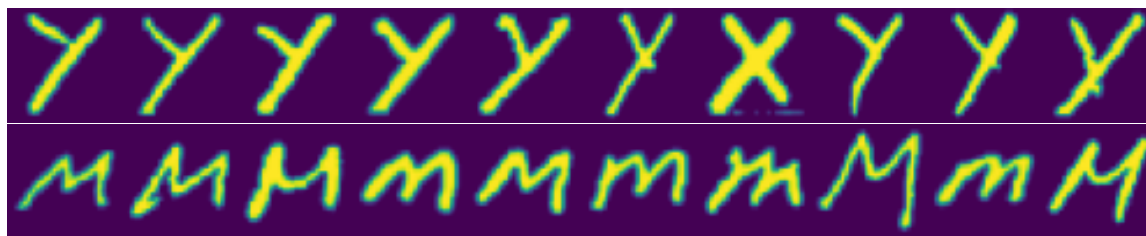


Figure 4.11: Resultado de los 9 vecinos más parecidos a la consulta. La primera imagen de cada fila es la consulta

4.4 Clustering

Clustering (agrupamiento o particionamiento) es una tarea común en aprendizaje de máquinas, generalmente llevada a cabo a través de modelos no-supervisados. Dado un conjunto de n puntos $\mathcal{D} = \{x_1, x_2, \dots, x_n\}$, el objetivo es generar un conjunto de grupos (digamos k grupos) $\mathcal{C} = \{c_1, c_2, \dots, c_k\}$. En general, esperamos que cada punto $x \in \mathbb{R}^d$ quede asignado al su grupo más representativo, por ejemplo, podría ser al grupo más cercano.

¹<https://www.nist.gov/itl/products-and-services/emnist-dataset>

Decimos que *clustering* está asociado a una estrategia no-supervisada dado que los algoritmos no necesitan conocer la clase de los items (puntos). Los modelos simplemente tratan de asignar puntos a grupos. Existen modelos en el que la cantidad de grupos se debe conocer *a-priori* y otros en el que esa cantidad es inferida por el propio modelo. La Figura 4.12 ilustra el proceso la entrada y salida de un modelo de *clustering*.

En esta sección describiremos dos modelos de clustering ampliamente utilizado, K-Means y DBSCAN.

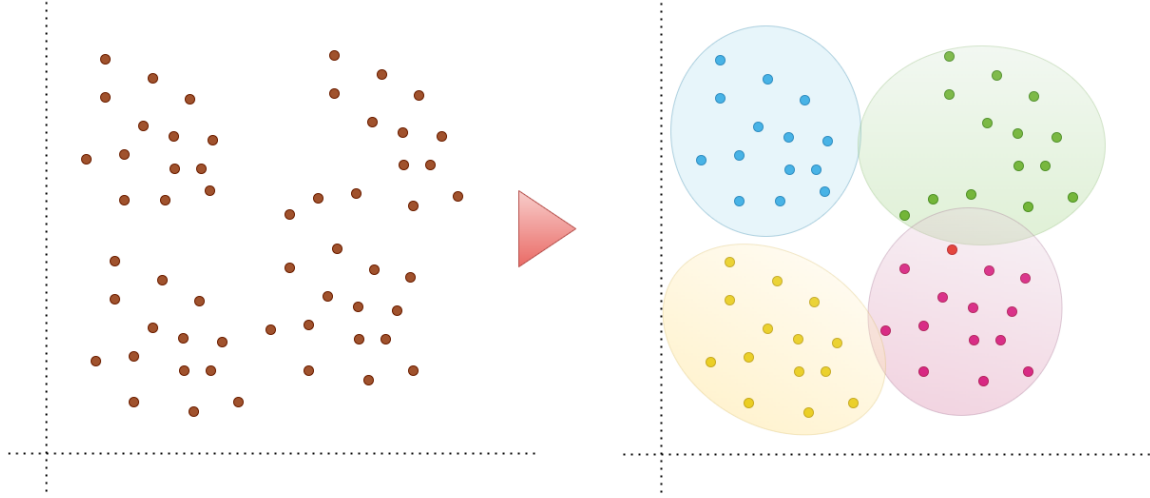


Figure 4.12: Esquema general del proceso de *clustering*.

4.4.1 k-Means

Es uno de los modelos más utilizados dada su simplicidad y efectividad de varios problemas relacionados con aprendizaje de máquinas. Este algoritmo está basado en la similitud entre puntos, calculado a través de la **distancia Euclidiana**. Con la distancia entre puntos, el modelo trata de inferir los mejor k centros para cada punto, donde cada centro representa cada uno de los *clusters* inferidos.

El algoritmo **k-Means** asume que hay k centros $\mu_j \in \mathbb{R}^d$. Inicialmente, estos centros son aleatorios y, luego, el algoritmo iterativamente los va ajustando, de acuerdo a los miembros de cada *cluster*. Entonces, dados los centros iniciales, **k-Means** ejecuta iterativamente los siguientes dos pasos:

1. **El primer paso (asignación)**, consiste en asignar cada $x_i \in \mathbb{R}^d$ al centro más cercano.

$$\text{center}(x) = \underset{j}{\operatorname{argmin}} \|x - \mu_j\|_2^2 \quad (4.15)$$

2. **El segundo paso (reestimación)** consiste en actualizar los centros, dado que cada *cluster* han sido actualizado en el paso anterior.

$$\mu_j = \frac{1}{N_{\text{center}(x)}} \sum_{x \in c_{\text{center}(x)}} x \quad (4.16)$$

El algoritmo se repite por un número determinado de iteraciones o hasta que los centros no se modifiquen más, es decir los grupos no cambian. Este algoritmo minimiza la siguiente función objetivo:

$$\mathcal{J} = \sum_{i=1}^n ||x_i - \mu_{center(x_i)}||^2 \quad (4.17)$$

El algoritmo **k-Means** depende de los centros elegidos inicialmente. Por lo tanto, se recomienda correr el algoritmo varias veces y quedarse con el conjunto de centros que minimice $\mathcal{J}$.

El Algoritmo 3 describe en pseudocódigo los pasos de **k-Means**.

Algorithm 3 k-Means

Require: $\mathcal{D} = \{x_1, x_2, \dots, x_n\}, x_i \in \mathbb{R}^d$

Require: N número máximo de iteraciones

$\mu_1, \dots, \mu_k \leftarrow$ inicializar k centros aleatoriamente $\mu_j \in \mathbb{R}^d$

for $i = 0 \rightarrow k$ **do**

$c_i \leftarrow \emptyset$

end for

for $it \leftarrow 1 \rightarrow N$ **do**

for $i = 0 \rightarrow k$ **do**

$c_i \leftarrow \emptyset$

end for

for $x_i \in \mathcal{D}$ **do**

$z \leftarrow \operatorname{argmin}_j ||x_i - \mu_j||_2^2$

$c_z \leftarrow c_z \cup x_i$

end for

for $i = 0 \rightarrow k$ **do**

$\mu_i = \frac{1}{N_i} \sum_{x \in c_i} x$

end for

end for

return $\mu_1, \dots, \mu_k, c_1, \dots, c_k$

La Figura 4.13 ilustra el proceso iterativo del algoritmo k-Means, partiendo con tres centros aleatorios y repitiendo el proceso de asignación y reestimación de centros por tres veces.

¿Cómo podemos aprovechar k-Means para reducir la cantidad de puntos en el espacio requerido por kNN?

Es importante notar que el algoritmo k-Means esta sesgado a crear clusters circulares, lo que puede no representar ciertos problemas. Por ejemplo, la Figura 4.14 muestra los grupos generados por k-Means en un espacio donde los datos no se agrupan en forma circular. Por

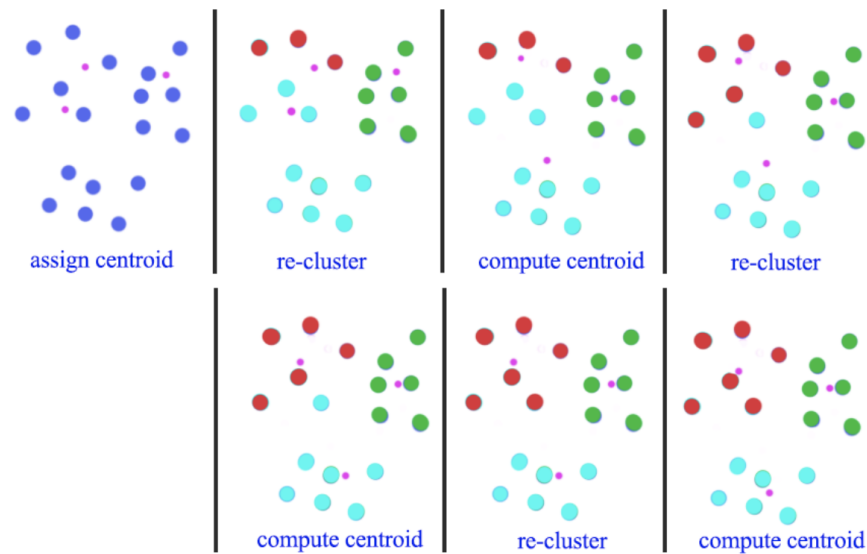


Figure 4.13: Ejemplo del proceso de clustering con k-Means.

lo tanto, es importante estudiar otros tipos de algoritmos. DBSCAN es otro algoritmo de clustering que trata de solucionar este problema.



Figure 4.14: Ejemplo de funcionamiento de k-Means en datos donde los grupos no siguen un esquema circular.

4.4.2 DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) es otro algoritmo de *clustering*, propuesto en 1996. A diferencia de k-Means, DBSCAN se basa en nociones de conectividad sobre vecindades, lo cual permite determinar grupos de diferentes formas siempre que sus elementos mantengan cierta cercanía (densidad) entre sí.

La idea principal de DBSCAN es que un punto pertenece a un *cluster* si este está cerca a muchos puntos de ese *cluster*.

Un *cluster* DBSCAN Q se forma por un conjunto de puntos tal que desde un punto cualesquiera $p \in Q$ se puede llegar a otro $q \in Q$ siguiendo una secuencia de puntos en Q , siempre que la distancia entre los puntos de cada par a, b de la secuencia sea menor o igual a eps .

Un punto B es **alcanzable** por otro A si podemos llegar desde A hasta B a través de una secuencia de puntos, tal que cada par de la secuencia tiene una distancia no mayor a eps .

Parámetros

- **eps**: es el valor de distancia que define a una vecindad. Dos puntos son considerados vecinos si la distancia entre ellos es menor o igual a eps .
- **minPts**: mínimo número de puntos que define a un cluster.

Tipos de Puntos

Basándonos en los parámetros definidos anteriormente, podemos identificar los siguientes tipos de puntos:

- **Core Point**: un punto es **core point** si existe al menos $minPts$ puntos en su vecindad.
- **Border Point**: un punto es **border point** si es alcanzable por un **core point** y existe menos de $minPts$ en su vecindad.
- **Outlier**: un punto es **outlier** si no es **core point** y no es alcanzable por ningún punto.

La Figura 4.15 ilustra la ocurrencia de los tres tipos de puntos descritos anteriormente. Aquí, suponemos que $minPts$ es 4. En este caso, N es un **outlier** ya que no tiene vecinos, A es **core point** dado que su vecindad tiene 4 vecinos. Sin embargo, B y C son **border points**, al tener una cantidad de vecinos menor a $minPts$, pero son alcanzables por A .

Algoritmo

Dadas las definiciones anteriores, estamos listos para describir el algoritmo. El algoritmo requiere definir previamente el valor de $minPts$ y eps , y luego procedemos con los siguientes pasos:

1. Seleccionamos inicialmente un punto de partida y determinamos su vecindad (radio = eps). Si la vecindad tiene menos de $minPts$, el punto es marcado como **outlier**. Si la vecindad tiene al menos $minPts$ puntos entonces el punto es definido como **core point**, creamos un nuevo *cluster* c con ese punto y lo marcamos como visitado. Luego visitamos cada uno de los vecinos y repetimos el proceso, incorporando a todos los visitados como miembros de c .
2. Seleccionamos aleatoriamente otro punto no visitado y repetimos el proceso anterior.
3. El proceso termina cuando todos los puntos han sido visitados. Al final los puntos no alcanzados son considerados *outliers*, es decir no pertenecen a ninguno de los *clusters*.

Algorithm 4 DBSCAN

Require: $\mathcal{D}$: conjunto de puntos.**Require:** eps **Require:** $minPts$ $C \leftarrow \emptyset$ **for** $x \in \mathcal{D}$, not visited **do** $z_x \leftarrow getNeighbors(x, \mathcal{D}, eps)$ **if** $|z_x| \geq minPts$ **then** $c \leftarrow$ new cluster $c \leftarrow DBSCAN_{search}(x, \mathcal{D}, eps, minPts, z_x, c)$ $C \leftarrow C \cup c$ **end if****end for**

Algorithm 5 $DBSCAN_{search}$

Require: x : punto inicial**Require:** $\mathcal{D}$: conjunto de puntos.**Require:** eps **Require:** $minPts$ **Require:** z_q : neighborhood**Require:** c : cluster $c.append(x)$ mark x as visited**if** $|z_x| \geq minPts$ **then****for** $q \in z_x$ **do** $z_q \leftarrow getNeighbors(q, \mathcal{D}, eps)$ $c \leftarrow DBSCAN_{search}(q, \mathcal{D}, eps, minPts, z_q, c)$ **end for****end if****return** c

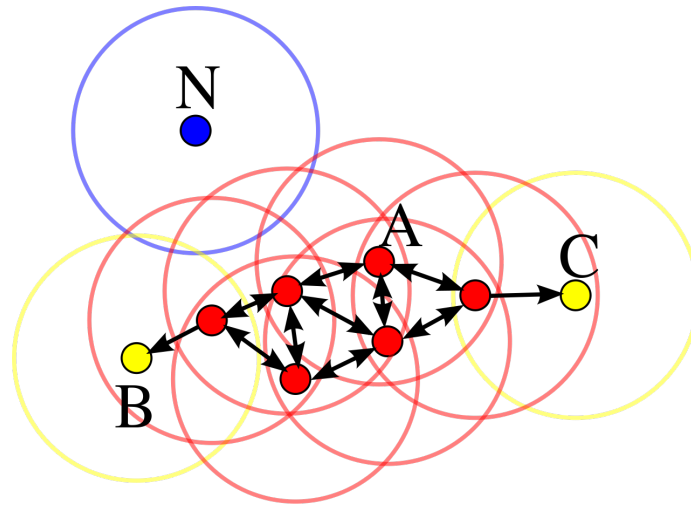


Figure 4.15: Ejemplo de tipos de puntos en DBSCAN con $\text{minPts}=4$. Aquí, N es un **outlier**, A es **core point** y B y C son **border points**.

La Figura 4.16 muestra el comportamiento de los algoritmos de clustering descritos anteriormente para diversas distribuciones de datos.

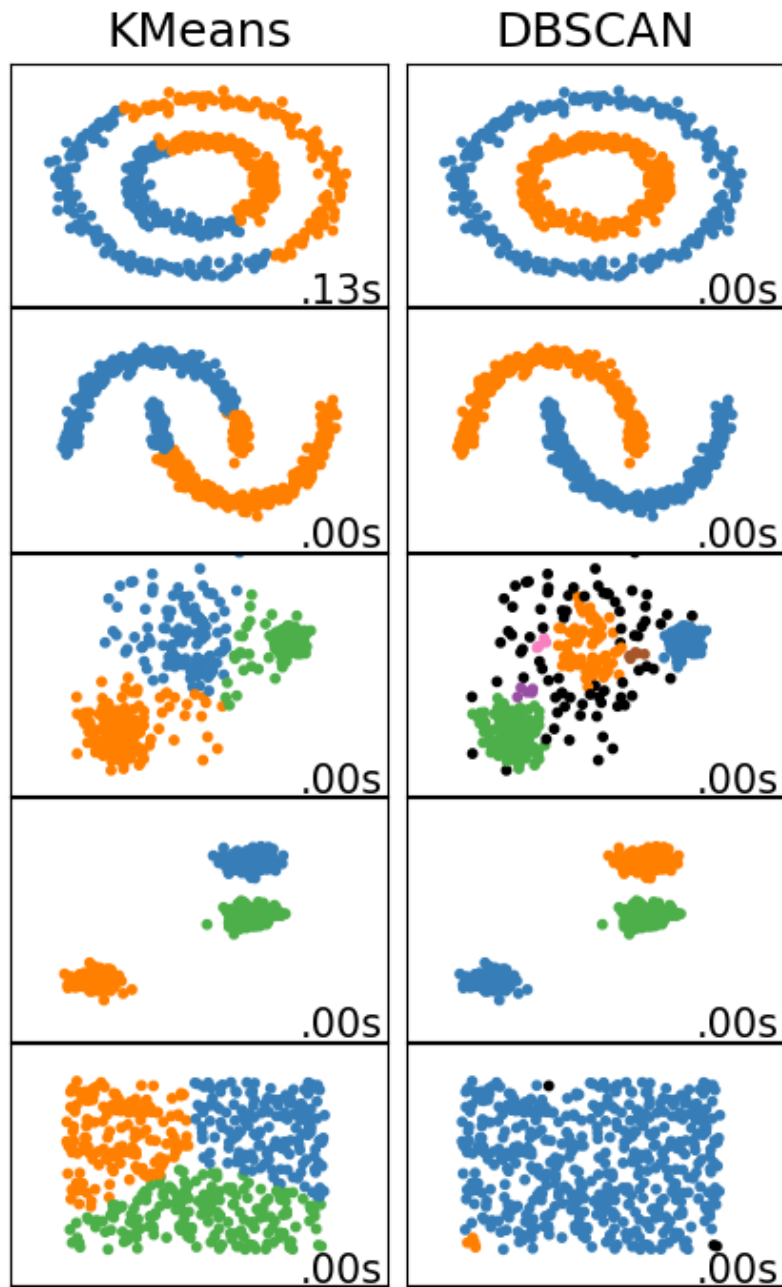


Figure 4.16: k-Means vs DBSCAN para diferentes conjuntos de datos.

4.4.3 Evaluación de Modelos de Clustering

Dada la naturaleza no-supervisada del proceso de *clustering*, resulta difícil evaluar el desempeño de un modelo. Sin embargo, hay algunas propuestas basados en medir la inter-variabilidad y entre-variabilidad de los grupos formados. La idea es que puntos similares formen parte del mismo *cluster* (baja inter-variabilidad) y puntos diferentes caigan en *clusters* diferentes (entre-variabilidad).

Rand Index

Sean dos particiones $U = \{u_1, \dots, u_R\}$ y $V = \{v_1, \dots, v_C\}$, donde U es la partición inferida y V es la partición de referencia (*ground truth*) sobre un conjunto de n puntos.

Definimos los siguientes conceptos:

- **True Positives (TP)**: es el número de pares que están en el mismo cluster en U y V .
- **True Negatives (TN)**: es el número de pares que aparecen en diferentes clusters en ambos U y V .
- **False Positives (FP)**: es el número de pares que aparecen en el mismo cluster en U , pero en diferentes clusters en V .
- **False Negatives (FN)**: es el número de pares que aparecen en diferentes clusters en U , pero en el mismo cluster en V .

Notemos que los conceptos **Positives** y **Negatives** en cómo ocurren los pares en V . Un par **Positivo** es aquel en el que sus elementos aparecen en el mismo *cluster*. Un par **Negativo** es aquel en el que sus elementos aparecen en diferentes *clusters*.

Dadas las definiciones anteriores definimos el estadístico **Rand Index** (R) como la fracción de pares correctamente agrupados. Esto es:

$$R = \frac{TP + TN}{TP + TF + FN + FP} \quad (4.18)$$

Para ejemplificar esta métrica, veamos la Figura 4.17 donde hay 17 puntos etiquetados como tres colores: azul (8), verde (5) y amarillo (4). Además, cierto modelo genera tres clusters. Según el ejemplo ¿Cuál será el valor de la métrica **Rand Index**?

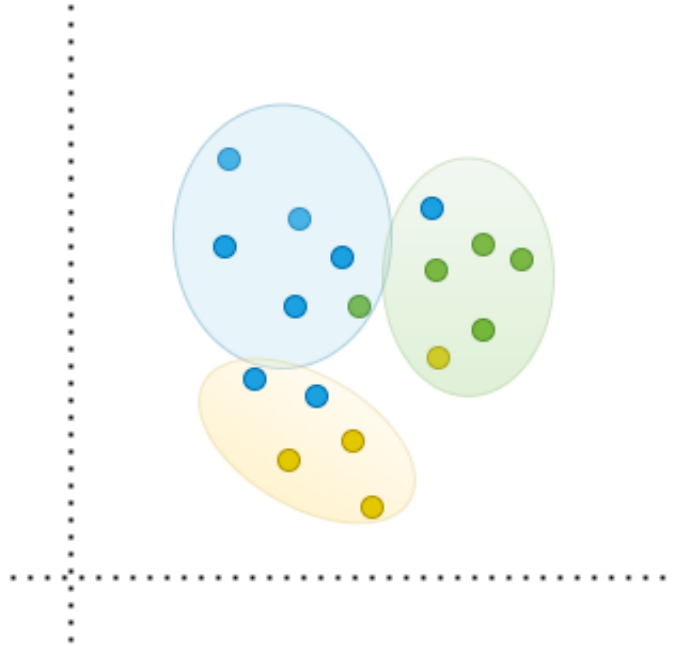


Figure 4.17: Ejemplo de resultado de un modelo de clustering.

Para estimar la métrica, definiremos dos grupos en base a los datos de referencia. El primero es el grupo de pares positivos, es decir pares en el que los elementos forman de un mismo *cluster*. El segundo es el grupo de pares negativos, pares en el que los elementos forman parte de de **clusters** diferentes.

Notemos que el total de pares es igual a TP + FN + TN + FP. Además, el total de positivos en los *clusters* referenciales debe ser igual a TP + FN. Finalmente, el total de negativos referenciales es igual a TN + FP.

Para el ejemplo, partamos por definir el total de pares posibles con los n puntos. Esto es obtenido calculando la combinatoria de n en 2, esto es $\binom{n}{2}$. Para nuestro caso $n = 17$ tenemos que el total de pares es $\binom{n}{2}$:

$$\binom{n}{2} = \frac{17!}{2!15!} = 17 \times 8 = 136. \quad (4.19)$$

Ahora el total de pares positivos se obtiene calculando la cantidad de pares en cada grupo de referencia. Así, el total de pares por cada color es:

$$\text{azules} \rightarrow \binom{8}{2} = \frac{8!}{2!6!} = 28 \quad (4.20)$$

$$\text{verdes} \rightarrow \binom{5}{2} = \frac{5!}{2!3!} = 10 \quad (4.21)$$

$$\text{amarillos} \rightarrow \binom{4}{2} = \frac{4!}{2!2!} = 6 \quad (4.22)$$

Por lo tanto, el total de pares positivos es $28 + 10 + 6 = 44$. El total de pares negativos se calcula como complemento $136 - 44 = 92$.

Ahora, podemos calcular TP en base a calcular los pares del mismo *cluster* referencial que caen en el mismo *cluster* inferido.

$$TP = \binom{5}{2} + \binom{4}{2} + \binom{3}{2} + \binom{2}{2} \quad (4.23)$$

$$= 10 + 6 + 3 + 1 = 20 \quad (4.24)$$

Dado lo anterior, $FN = \text{Positivo} - TP = 44 - 20 = 24$. Para estimar los FP, debemos estimar el número de pares positivos inferidos, para esto tomamos los pares de cada *cluster* inferido PI. Así, $FP = PI - TP$, es decir FP representa el total de pares inferidos positivamente (caen en el mismo *cluster*) que no son correctos.

$$PI = \binom{6}{2} + \binom{6}{2} + \binom{5}{2} \quad (4.25)$$

$$= 15 + 15 + 10 = 40 \quad (4.26)$$

TP	FN	TN	FP
20	24	72	20

Table 4.1: Caption

Por lo anterior, $FP = PI - TP = 40 - 20 = 20$. Ahora los $TN = \text{Negativos} - FP$, es decir $TN = 92 - 20 = 72$. Por lo tanto ya tenemos todos los valores que necesitamos (Tabla 4.1):

Por lo tanto, para el ejercicio propuesto, el valor de **Rand Index** queda como:

$$R = \frac{20 + 72}{20 + 24 + 72 + 20} = 92/136 \sim 0.68 \quad (4.27)$$



5. Reducción de la Dimensionalidad

La tarea de reducción consisten en la transformación de los datos desde un espacio original, posiblemente de alta dimensionalidad, a un espacio con mayor poder discriminativo y menor número de variables. Así, la reducción de dimensionalidad permite:

- Extracción de características relevantes.
- Reducción de redundancia de datos.
- Aumento de la eficiencia en cálculo posteriores.

En esta sección revisaremos dos modelos de *clustering* popularmente utilizados. Primero, uno tradicional, enfocado en un análisis global de los datos; y el segundo, un método más moderno, enfocado en estructuras locales del espacio subyacente.

5.1 Principal Component Analysis (PCA)

PCA es uno de los métodos no supervisados más populares para reducción de la dimensionalidad. El hecho que sea no-supervisado indica que no considera las clases o etiquetas de los datos.

La idea de PCA es encontrar una transformación lineal que logre una mejor representación de los datos. En particular, PCA se basa en el criterio de varianza, buscando los ejes de mayor varianza sobre el que se transforman los datos. Así, ejes con poca varianza pueden ser descartados sin afectar gravemente la semántica de las representaciones. La Figura 5.1 ilustra la transformación generada por el algoritmo de PCA.

5.1.1 Algoritmo PCA

Sea $\mathbf{X}_{n \times d}$, una matriz que representa los datos en el espacio original, donde cada por fila contiene un ejemplar con d dimensiones. Además, supondremos que los datos están centrados, es decir cada variable tiene media 0. Esto se logra obteniendo el vector promedio μ y luego, restándolo de $\mathbf{X}$.

Por otro lado, $\mathbf{Y}_{n \times d}$ representa el conjunto de los datos transformados y $\mathbf{W}_{d \times d}$ es la

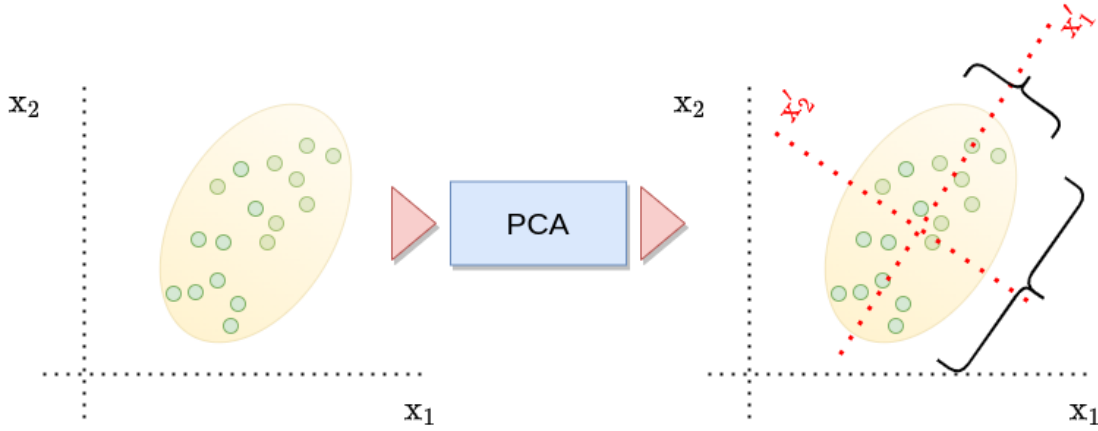


Figure 5.1: Esquema de PCA

matriz de transformación.

$$\mathbf{Y} = \mathbf{X}\mathbf{W} \quad (5.1)$$

Nuestro objetivo es generar representaciones en el que las variables no presenten correlación, es decir sean independientes entre sí. Por lo tanto la matriz de covarianza de $\mathbf{Y}$ se define como:

$$\text{cov}(\mathbf{Y}) = \text{cov}(\mathbf{X}\mathbf{W}) = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_d \end{pmatrix}_{d \times d} \quad (5.2)$$

$$\begin{aligned} \text{cov}(\mathbf{Y}) &= \mathbb{E}[\mathbf{Y}^T \mathbf{Y}] \\ &= \mathbb{E}[(\mathbf{X}\mathbf{W})^T (\mathbf{X}\mathbf{W})] \\ &= \mathbb{E}[\mathbf{W}^T \mathbf{X}^T (\mathbf{X}\mathbf{W})] \\ &= \mathbf{W}^T \mathbb{E}[\mathbf{X}^T \mathbf{X}] \mathbf{W} \\ &= \mathbf{W}^T \text{cov}(\mathbf{X}) \mathbf{W} \end{aligned}$$

multiplicando a ambos lados por $\mathbf{W}$, tenemos:

$$\mathbf{W} \text{cov}(\mathbf{Y}) = \text{cov}(\mathbf{X}) \mathbf{W} \quad (5.3)$$

$$\text{recordemos que } \text{cov}(\mathbf{Y}) \text{ es diagonal} \quad (5.4)$$

$$[\lambda_1 \mathbf{W}_1, \dots, \lambda_d \mathbf{W}_d] = \text{cov}(\mathbf{X}) \mathbf{W} \quad (5.5)$$

Dado, lo anterior podemos inferir que:

$$\text{cov}(\mathbf{X}) \mathbf{W}_i = \lambda_i \mathbf{W}_i, \quad (5.6)$$

que resulta ser la ecuación general de autovectores (*eigenvectors*). Es decir, cada $\mathbf{W}_i$ corresponde a un autovector y λ_i es su correspondiente autovalor. Así, un punto d -dimensional $\mathbf{x}$, se transforma a un punto $\mathbf{y}$, a través de:

$$\mathbf{y}_{1 \times d} = \mathbf{x}_{1 \times d} \mathbf{W}_{d \times d} \quad (5.7)$$

5.1.2 Reconstrucción

Cada columna de $\mathbf{W}$ representa una base de un sistema de bases ortogonales. Así, para reconstruir $\mathbf{x}$ desde $\mathbf{y}$, simplemente debemos multiplicar el valor de cada componente y_i por la base correspondiente. Miremos la siguiente ecuación:

$$\mathbf{x} = \sum_{i=1}^d y_i \mathbf{W}_i \quad (5.8)$$

5.1.3 Selección de Características

Recordemos que en el nuevo sistema, la varianza en la dimensión i está dada por el autovalor λ_i . Por lo tanto, para reducir la dimensionalidad original, es suficiente tomar los k autovectores $\mathbf{W}_i$ con mayor autovalor, descartando los $k - d$ con menor autovalor.

$$\mathbf{Y} = \mathbf{X} \mathbf{W}_{1 \dots k} \quad (5.9)$$

Dado un espacio reducido, no podremos reconstruir completamente los datos originales dado que descartamos ciertas componentes. Sin embargo, las componentes descartadas son aquellas que aportan menos información a los datos. Por lo tanto, la reconstrucción, en cierto sentido, será una reconstrucción aproximada.

$$\hat{\mathbf{x}} = \sum_{i=1}^k y_i \mathbf{W}_i, \quad k < d \quad (5.10)$$

Pero, ¿cuál es la pérdida o error generada por la aproximación?

Primero, definamos el error de reconstrucción como:

$$\begin{aligned} \delta &= \mathbb{E}[\|\mathbf{x} - \hat{\mathbf{x}}\|_2^2] \\ &= \mathbb{E}\left[\left\|\sum_{i=k+1}^d y_i \mathbf{W}_i\right\|_2^2\right] \\ &= \mathbb{E}\left[\sum_{i=k+1}^d (y_i \mathbf{W}_i)^T (\mathbf{W}_i y_i)\right] \\ &= \mathbb{E}\left[\sum_{i=k+1}^d y_i (\mathbf{W}_i^T \mathbf{W}_i) y_i\right] \\ &= \sum_{i=k+1}^d \mathbb{E}[y_i y_i] \\ &= \sum_{i=k+1}^d \lambda_i \end{aligned} \quad (5.11)$$

Notemos que la pérdida está asociada a la suma de los autovalores más pequeños, lo que representa una solución óptima bajo el esquema centrado en la reconstrucción global de los datos.

5.1.4 Algoritmo

El algoritmo 6 muestra el proceso PCA y la Figura 5.2 ilustra un ejemplo de aplicación sobre el conjunto de datos de Iris.

Algorithm 6 PCA

Require: $\mathbf{X}$: conjunto de n datos en un espacio de d -dimensiones.

$\mu \leftarrow$ media de los datos

$\mathbf{X} \leftarrow \mathbf{X} - \mu$

$\mathbf{Z} \leftarrow \text{cov}(\mathbf{X})$

resolver $\mathbf{Z}\mathbf{W} = \lambda\mathbf{W}$

tomar los k autovectores con mayor autovalor $\rightarrow \mathbf{W}_{1\dots k}$

$\mathbf{Y} \leftarrow \mathbf{X}\mathbf{W}_{1\dots k}$

return $\mathbf{Y}, \mathbf{W}_{1\dots k}$

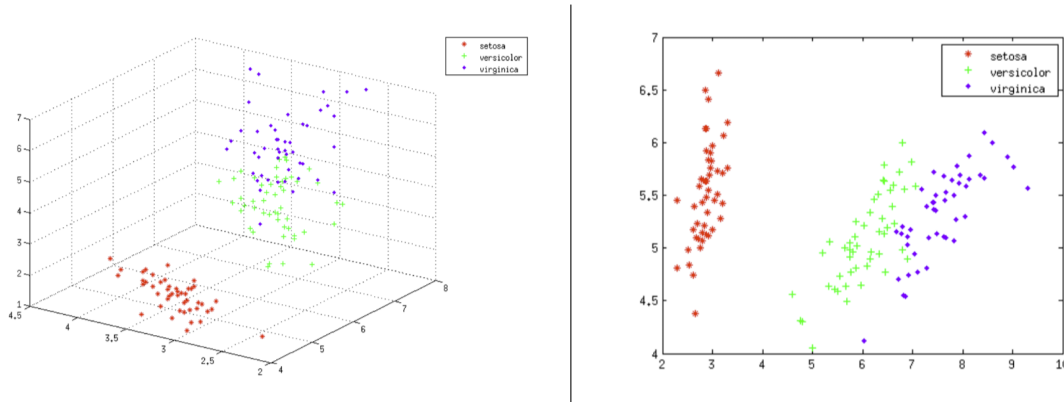


Figure 5.2: Ejemplo de reducción de dimensionalidad con PCA sobre el conjunto de datos de Iris

5.1.5 Ejemplo de Aplicación PCA

El algoritmo PCA permite caracterizar los objetos de cierta naturaleza. Por ejemplo, a inicios de los 90's fue utilizado para caracterizar rostros de personas en el trabajo de Turk y Pentland, titulado "Eigenfaces for Recognition". La idea es llevar los rostros a un espacio discriminativo a través de PCA y aplicar **vecino más cercano** (kNN) sobre ese espacio. Además, podemos utilizar el error de reconstrucción con el vecino más cercano para determinar un valor de certeza.

Para ejemplificar este caso, utilizaremos un famoso conjunto de datos de rostros (Yale database ¹). La Figura 5.3 muestra algunos rostros de Yale.

¹<https://www.kaggle.com/datasets/olgabelitskaya/yale-face-database/code?resource=>



Figure 5.3: Diversos rostros del conjunto Yale.

La Figura 5.4 muestra los 25 eigenfaces más importantes sobre el conjunto de datos Yale. Además, la Figura 5.5 muestra la reconstrucción de una imagen de rostro desde dos espacios reducidos, utilizando la Ecuación 5.10.

Tarea 4: Evaluar reconocimiento de letras con espacios reducidos PCA vs UMAP (k=8,16,32,64,128 + original)



Figure 5.4: Ejemplo de los 25 eigenfaces principales.



Figure 5.5: Ejemplo de reconstrucción de imágenes de rostro desde el espacio reducido con PCA. La primera imagen es la imagen original. La segunda imagen es la reconstrucción desde un espacio reducido a 20 dimensiones. La tercera es el resultado de reconstruir la imagen original desde un espacio de 60 dimensiones.

5.2 UMAP

PCA es un modelo de reducción dimensional global, es decir analiza características de los datos en su conjunto (como es el caso de la varianza). Sin embargo, eso no garantiza que la estructura local (vecindad entre puntos) se mantenga. Mantener la vecindad local es altamente relevante para garantizar un desempeño similar en el espacio reducido con respecto al original. Por ejemplo, para búsqueda por similitud.

Uno de los algoritmos modernos de reducción dimensional que trata de mantener la estructura local del espacio original es UMAP (Unifold Manifold Aproximation and

Projection), algoritmo propuesto por McInnes et al, en 2018. UMAP ofrece importantes ventajas sobre tSNE, otro algoritmo popular, perteneciente a la misma familia de algoritmos que preservan la estructura local. UMAP es más rápido y preserva mejor la estructura global de los datos; además, tiene un fundamento matemático que lo hace el preferido por la comunidad.

5.2.1 Estructura Local

Un **Complejo Simplicial** es un tipo particular de espacio topológico, formado por la unión de complejos simpliciales más simples. Aquí, un punto es un complejo simplicial 0-dimensional, mientras que una línea se dice que es 1-dimensional, y es la unión de dos simpliciales 0-dimensionales. La Figura 5.6 ilustra ejemplos de complejos simpliciales de diversos órdenes.

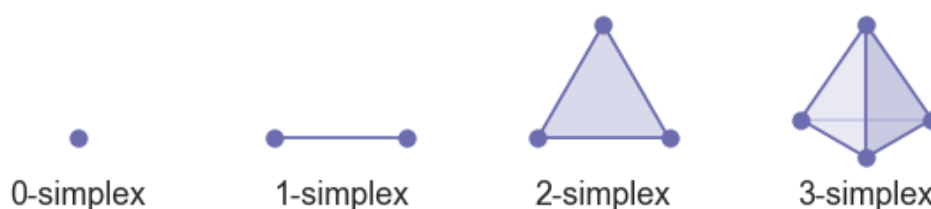


Figure 5.6: Ejemplo de complejos simpliciales de diversos órdenes.

Los complejos simpliciales representan una buena forma de establecer la estructura local del espacio subyacente. Particularmente, en el contexto de simpliciales 0 y 1 dimensionales (un grafo), que hacen más eficiente el análisis.

Un **conjunto cobertor** (Cover), representa la vecindad de cada punto. En nuestro caso cada conjunto cobertor se establece en una vecindad definida por una hipersfera (ball) de cierto radio alrededor de cada punto. Miremos el ejemplo de la Figura 5.7. Dada la vecindad, determinamos los complejos simpliciales como muestra la Figura 5.8

Elegir el radio correcto es complicado. Si elegimos radios muy pequeños, tendremos muchos complejos simpliciales 0-dimensionales. Si elegimos radios muy grandes tendremos vecindades que no representan la estructura local del espacio. Además, dado que los datos no están uniformemente distribuidos, resulta que radios constantes puede generar malas representaciones, por ejemplo, radios pequeños no representarían la estructura de zonas poco densas. Por lo tanto, usar radios variables es una buena solución (Figura 5.9). Además, normalizamos localmente el tamaño de la esfera, de manera que todas sean esferas unitarias. **Esta normalización es crítica para la formulación matemática.**

Una forma de hacer variable la región es a través de la la noción de k -vecinos. Así, el valor de k indicará el nivel de localidad que queremos representar. Además, se extiende la idea de regiones bien delimitadas a regiones difusas (Figura 5.10).

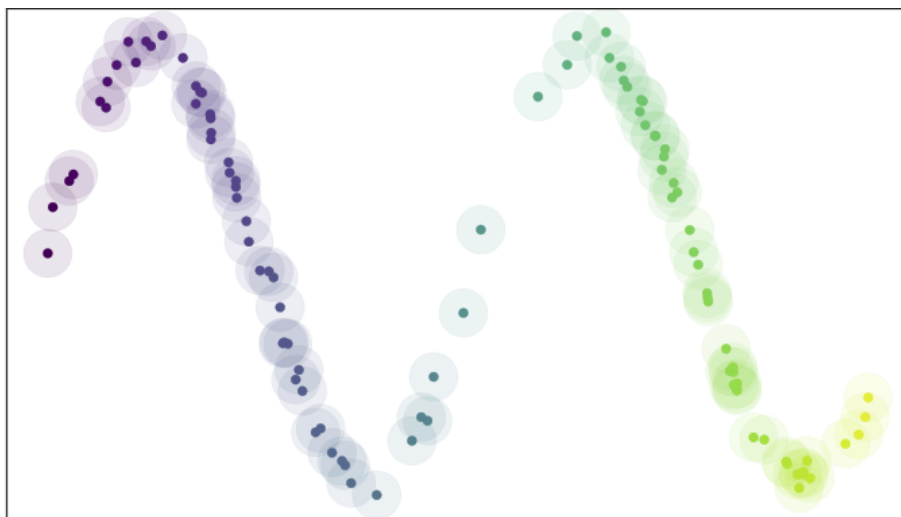


Figure 5.7: Cojuntos cobertores definidos en cada punto.

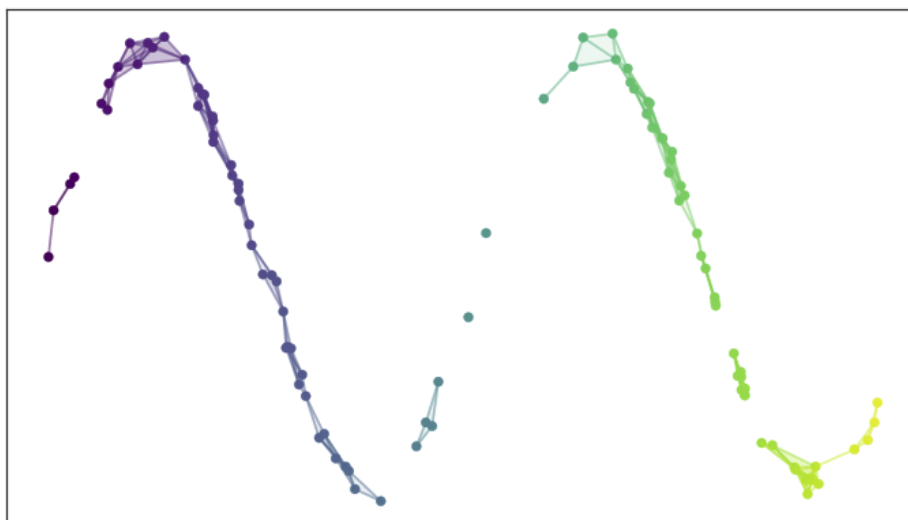


Figure 5.8: Simpliciales inferidos en cada vecindad.

Distribución de distancias

Normalizar las distancias localmente reduce el impacto de la maldición de dimensionalidad, que hace que en altas dimensiones las distancias sean muy parecidas entre todas (ver Figura 5.11, 5.12 y 5.13).

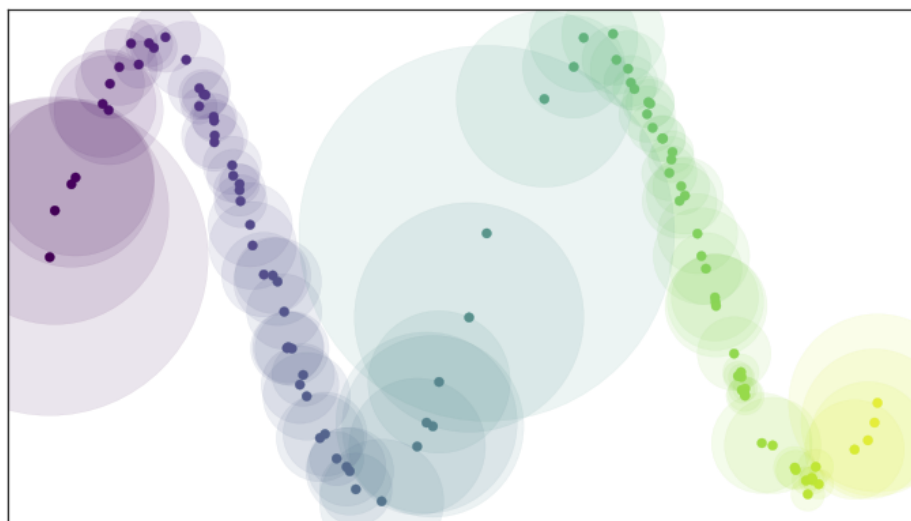


Figure 5.9: Regiones de radio variable.

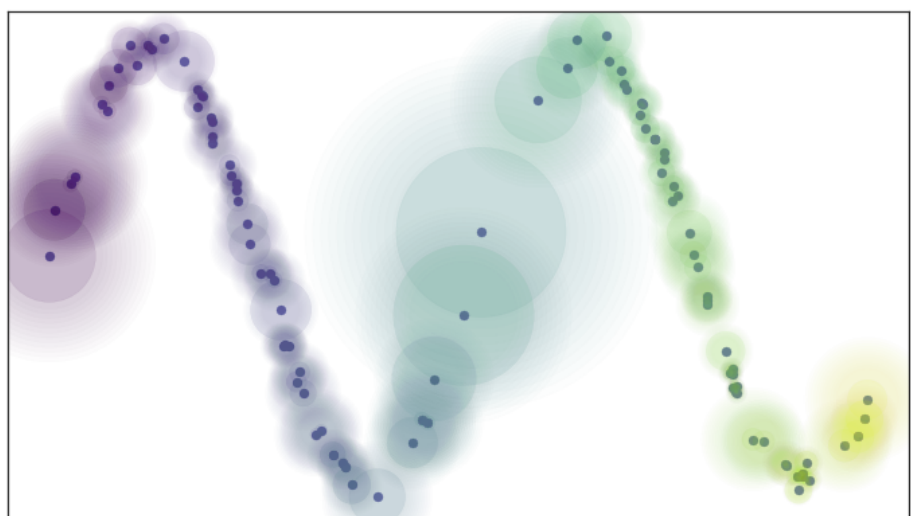
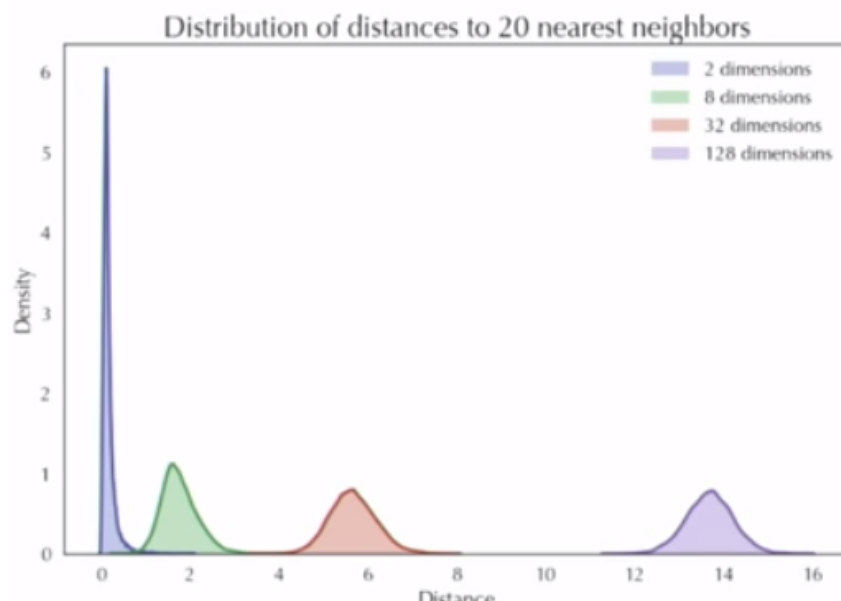


Figure 5.10: Regiones de radio variable con límite difusos.



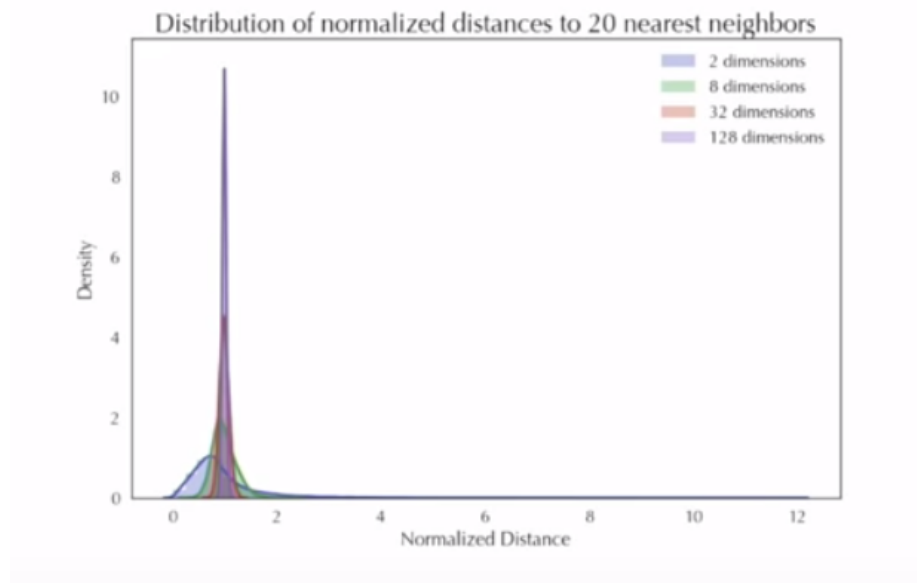


Figure 5.12: Distribución de distancias normalizadas en diferentes dimensionalidades

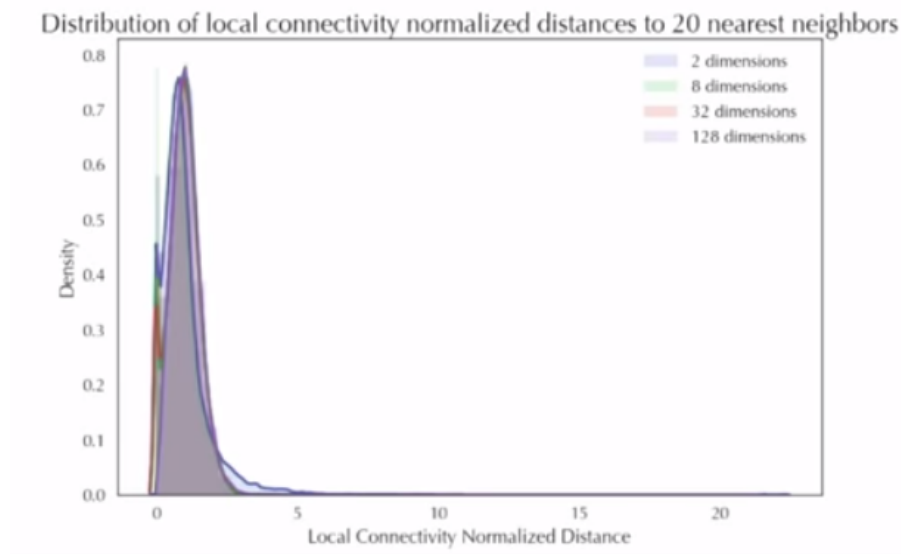


Figure 5.13: Distribución de distancias en diferentes dimensionalidades normalizadas localmente.

Grafo

Dado lo anterior, UMAP representa el espacio original como un grafo, donde los nodos son los puntos y las aristas son simpliciales 1-dimensionales que unen dos puntos y los pesos de las aristas se calculan localmente con respecto a cada punto². Por ejemplo, miremos el esquema de la Figura 5.14

²<https://www.youtube.com/watch?v=eN0wFzBA4Sc&t=219s>

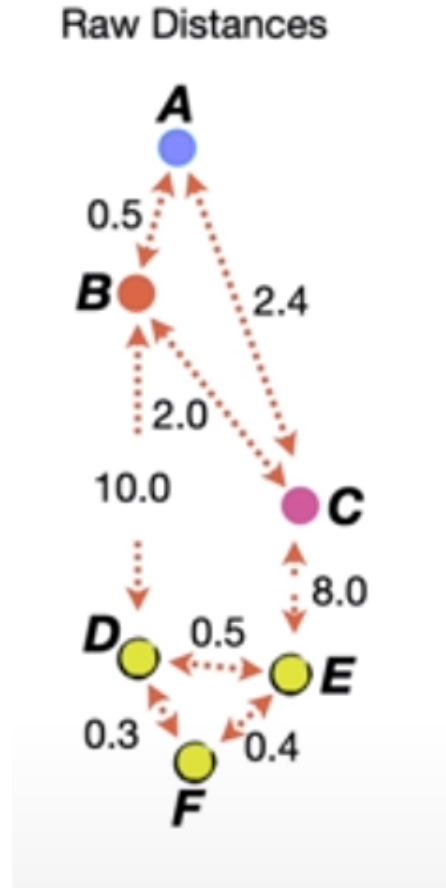


Figure 5.14: Ejemplo de formación de grafo inicial con UMAP

donde. el punto en referencia $\mathbf{x}_i = A$, el peso de cada arista que va desde $\mathbf{x}_i$ a $\mathbf{x}_j$ está dado por:

$$w(\mathbf{x}_i, \mathbf{x}_j) = e^{-\frac{D(\mathbf{x}_i, \mathbf{x}_j) - \rho_i}{\sigma_i}} \quad (5.12)$$

- D : es una función de distancia entre puntos. Aquí asumiremos que es la distancia Euclidiana.
- ρ_i : es la distancia mínima de los vecinos de $\mathbf{x}_i$.
- σ_i : es un parámetro de normalización de modo que se cumpla lo siguiente:

$$\sum_{j=1}^k w(\mathbf{x}_i, \mathbf{x}_j) = \log_2(k) \quad (5.13)$$

es importante notar que los k -vecinos incluye al mismo punto. Además, para garantizar las simetría:

$$w(\mathbf{x}_i, \mathbf{x}_j) = w(\mathbf{x}_i, \mathbf{x}_j) + w(\mathbf{x}_j, \mathbf{x}_i) - w(\mathbf{x}_i, \mathbf{x}_j)w(\mathbf{x}_j, \mathbf{x}_i) \quad (5.14)$$

5.2.2 Optimización

Dado el grafo inicial, que representa el espacio original, donde los nodos se representan en alta dimensionalidad, queremos encontrar un grafo de menor dimensionalidad que mantenga la estructura del grafo. Para ello, definimos la siguiente función de pérdida que debemos minimizar:

$$\sum_{e \in \mathcal{E}} w_h(e) \log \left(\frac{w_h(e)}{w_l(e)} \right) + (1 - w_h(e)) \log \left(\frac{1 - w_h(e)}{1 - w_l(e)} \right), \quad (5.15)$$

donde w_h y w_l son los pesos de las aristas en el grafo de alta y baja dimensionalidad, respectivamente. La optimización se realiza tomando dos pares con respecto a un punto. Uno que corresponde a un vecino y otro a un que está fuera de su alcance. Luego se procede a través del **gradiente descendente**.

La ecuación anterior expresa dos tipo de fuerzas: **fuerza de atracción** que permite que vecinos en el espacio original se mantengan como vecinos en el espacio destino, y **fuerza de repulsión** que mantenga lejos puntos distantes en el espacio.

Para el caso del grafo destino, UMAP utiliza una función de pertenencia que le permite controlar con mayor flexibilidad el espacio destino. Para ello, se define la función $\Phi(\cdot, \cdot)$ de la siguiente manera:

$$\Phi(x, y) = \frac{1}{1 + a(\|x - y\|_2^2)^b}, \quad (5.16)$$

don a y b se parametrizan de acuerdo a la distancia mínima entre puntos esperada (`mind_dist`)

5.2.3 Parámetros UMAP

- **k (n_neighbors)**: controla el balance entre estructura local y global. Un valor recomendado es 15.
- **min_dist**: controla la cercanía entre puntos. Un valor pequeño trata de acercar los puntos, de modo que se generan clusters. Valores altos generan mayor dispersión. Un valor recomendado es 0.1 .
- **n_components**: indica el tamaño del nuevo espacio

Un interesante recurso sobre UMAP lo puedes encontrar en <https://pair-code.github.io/understanding-umap/>

Caso de Estudio

Revisar el efecto de UMAP para mejorar las representaciones en **sketch-based image retrieval**. https://openaccess.thecvf.com/content/CVPR2021W/SketchDL/papers/Torres_Compact_and_Effective_Representations_for_Sketch-Based_Image_Retrieval_CVPRW_2021_paper.pdf