

**Universidad Austral**

Facultad de Ingeniería

Maestría en Ciencia de Datos

**Trabajo Práctico Final**

Data Mining Avanzado

Autores:

Alvaro, Giuliana

Chalup, Sarah

Franco, Agustina

Guzman, Mariana

Prof. Volpacchio, Martín

Septiembre 2025

Índice

[Ejercicio 1 – Redes Neuronales 4](#_Toc209194615)

[Perceptron Multicapa 4](#_Toc209194616)

[Modelo Radial Base 5](#_Toc209194617)

[Diferencias entre Redes “Deep” y “Shallow” 5](#_Toc209194618)

[Conceptos de “embedding” y “autoencoder” 6](#_Toc209194619)

[Red de Kohonnen – knet o Mapa Autoorganizativo 8](#_Toc209194620)

[Convolución en redes neuronales 10](#_Toc209194621)

[Algoritmo Encoder-Decoder 12](#_Toc209194622)

[Ejercicio 2 – Support Vector Machine 13](#_Toc209194623)

[Tipos de VSM 13](#_Toc209194624)

[Parámetros del algoritmo en su versión no lineal separable 15](#_Toc209194625)

[Características de VSM vs. Otros algoritmos de Machine Learning 15](#_Toc209194626)

[Adaptación de VSM en clasificación, para poder usarlo como regresión 16](#_Toc209194627)

[Interpretación sobre ventajas de Redes Neuronales sobre VSM 16](#_Toc209194628)

[Ejercicio 3 – Statistical Learning Theory 17](#_Toc209194629)

[Concepto de Statistical Learning Theory 17](#_Toc209194630)

[Características claves que se abordan con la SLT: 17](#_Toc209194631)

[Indicador de complejidad de un dataset usando el algoritmo SVM 17](#_Toc209194632)

[Capacidad de un algoritmo y manera heurística de Vapnik-Chervonenkis para estimarla 18](#_Toc209194633)

[Ejercicio 4 – Algoritmos Genéticos 19](#_Toc209194634)

[Qué son los algoritmos genéticos 19](#_Toc209194635)

[Pseudocódigo de un algoritmo genético 19](#_Toc209194636)

[Diferencias principales entre algoritmos genéticos reales y binarios 20](#_Toc209194637)

[Casos de aplicación 21](#_Toc209194638)

[Implementación open-source 22](#_Toc209194639)

[Ejercicio 5 – Componentes Principales 22](#_Toc209194640)

[Qué es PCA 22](#_Toc209194641)

[Kernel 23](#_Toc209194642)

[Ejercicio 6 – Análisis de Supervivencia 24](#_Toc209194643)

[Qué es el análisis de supervivencia 24](#_Toc209194644)

[Tipos de análisis de supervivencia 25](#_Toc209194645)

[Implementación en Python 25](#_Toc209194646)

[Ejercicio 7 – Clustering 26](#_Toc209194647)

[Análisis de Componentes Principales (PCA) 27](#_Toc209194648)

[Algoritmo t-sne 29](#_Toc209194649)

[Ejemplo de implementación en Python: 29](#_Toc209194650)

[Limitaciones 30](#_Toc209194651)

[Ventajas 31](#_Toc209194652)

[Ejercicio 8 – Adaboost y Gradient Boosting 31](#_Toc209194653)

[Diferencias entre Adaboost y Gradient Boosting 31](#_Toc209194654)

[Ejercicio 9 – Random Forest 32](#_Toc209194655)

[Qué es Random Forest 32](#_Toc209194656)

[Ejercicio 10 – Modelos en Python 35](#_Toc209194657)

[Ejercico 11 – Voted Peceptron 38](#_Toc209194658)

# **Ejercicio 1 – Redes Neuronales**

1. ***Dar la topología (dibujo y conexiones) y las funciones de activación convenientes para entrenar una red que estime las probabilidades de pertenecer a cada una de tres clases en un problema con 4 variables independientes.***

## **Perceptron Multicapa**

Para poder estimar las probabilidades de pertenecer a una de 3 clases dado un problema con 4 variables independientes, lo más adecuado sería utilizar una red neuronal de tipo Perceptrón Multicapa (MLP) donde:

* La capa de entrada está compuesta por 4 neuronas, una por cada variable independiente
* Haya 1 o 2 capas ocultas con un número moderado de neuronas con función de activación sea ReLU o tanh, ya que permiten introducir no linealidad y el aprendizaje de relaciones complejas
* La capa de salida serán una neurona por cada una de las 3 clases con función de activación de tipo softmax, la cual convierte las salidas en probabilidades que suman 1
* La función de pérdida es categorical cross-entropy

A diagram of a network

AI-generated content may be incorrect.De este modo, el esquema quedaría algo así

1. ***¿Qué aporta el modelo Radial Base a las redes Multilayer Perceptron?***

## **Modelo Radial Base**

El modelo de Redes de Base Radial (RBF) aporta a los Perceptrones Multicapa (MLP) una alternativa nueva a la manera en que se representa y procesa la información. En una MLP tradicional, las capas ocultas trabajan de forma tal que realizan transformaciones globales mediante combinaciones lineales de los pesos y funciones de activación sigmoides, ya sean tanh o ReLU. De este modo, logran captar relaciones no lineales complejas, lo que requiere de un mayor mantenimiento y ajustes más finos en el número de capas, tasa de aprendizaje e inicialización de los pesos que lo vuelve bastante costoso.

Por otro lado, en un modelo RBF la capa oculta utiliza funciones de base radial. Es decir, que cada neurona responde fuertemente a patrones de entrada cercanos a su centro, por lo que el modelo sea más local (ya que aprende regiones en el espacio de entrada). Para ello, el aprendizaje se divide en dos etapas:

1. Determinar los centros de las funciones radiales, como puede ser un clustering K-means
2. Ajustar los pesos de la salida, donde se suelen usar modelos lineales

De este modo, los RBF aportan a los MLP:

* Mayor interpretabilidad al modelo ya que cada neurona oculta se asocia a una región del espacio de entrada
* Más rapidez de entrenamiento, ya que separa el ajuste de los centros y los pesos
* En problemas con datos bien distribuidos, las RBF capturan relaciones locales mejor que las MLP de forma que aumenta la interpolación
* Más robustez ya que, al enfocarse en vecindades locales, suelen ser menos sensibles al ruido global

1. ***Explique las diferencias esenciales entre Redes “Deep” y “Shallow”***

## **Diferencias entre Redes “Deep” y “Shallow”**

Las diferencias esenciales entre las Redes Neuronales "Deep" (Profundas) y "Shallow" (Poco Profundas) están principalmente en su arquitectura, lo que a su vez impacta en su complejidad, capacidad de aprendizaje, requisitos de datos y recursos computacionales, y su interpretabilidad.

Las redes shallow presentan una arquitectura simple, compuesta generalmente por una capa de entrada, una única capa oculta, y una capa de salida. Son útiles en tareas básicas como clasificación binaria, regresión o problemas lineales, donde la relación entre variables no es compleja. Por el contrario, las redes deep están formadas por múltiples capas ocultas  intermedias, lo que les permite aprender representaciones más jerárquicas y abstractas de los datos. Este tipo de red constituye la base fundamental del aprendizaje profundo y se utiliza en tareas como reconocimiento de imágenes, procesamiento de lenguaje natural o voz.

En términos de capacidad de aprendizaje, las redes shallow pueden resolver problemas con estructuras de datos simples, pero se ven limitadas cuando deben aprender relaciones complejas. Las redes deep, en cambio, pueden modelar patrones más sofisticados y descubrir estructuras internas ocultas, gracias a la profundidad de sus capas. Esta ventaja, sin embargo, viene acompañada de una mayor complejidad en el entrenamiento y una mayor cantidad de parámetros, lo cual requiere más datos y más poder computacional.

Otra diferencia importante está en el riesgo de sobreajuste. Las redes shallow, al tener menos parámetros, son menos propensas a memorizar el ruido de los datos de entrenamiento. En cambio, las redes deep tienen una gran capacidad de modelado, lo que también aumenta el riesgo de sobreajuste si no se aplican técnicas de regularización, como dropout, early stopping, o L1/L2.

Desde el punto de vista de los requisitos de datos, las redes shallow pueden entrenarse eficazmente con pocos ejemplos, mientras que las redes deep necesitan grandes volúmenes de datos para lograr buenos resultados y evitar el sobreajuste. Asimismo, los modelos deep suelen requerir hardware especializado, como GPUs o TPUs, ya que el proceso de entrenamiento es intensivo en cálculos, mientras que las redes shallow pueden ejecutarse fácilmente en computadoras convencionales.

En cuanto a la interpretabilidad, las redes shallow son más fáciles de analizar, ya que su estructura sencilla permite comprender con mayor claridad cómo se generan las predicciones. Las redes deep, por su parte, son frecuentemente vistas como "cajas negras", ya que entender cómo interactúan sus numerosas capas es un desafío, incluso para expertos.

En conclusión, la elección entre redes "shallow" y "deep" depende de la complejidad de la tarea, la cantidad de datos disponibles y los recursos computacionales. Las redes shallow  son adecuadas para tareas más simples y conjuntos de datos más pequeños, ofreciendo eficiencia y facilidad de interpretación. Las redes deep, en cambio, son mejores para problemas complejos con grandes conjuntos de datos, proporcionando una capacidad de aprendizaje superior a costa de una mayor complejidad y demandas computacionales

Fuente:Amazon Web Services. (s. f.). The difference between Deep Learning and Neural Networks. Recuperado de<https://aws.amazon.com/es/compare/the-difference-between-deep-learning-and-neural-networks/>

Tan, P.-N., Steinbach, M., Karpatne, A., & Kumar, V. (2019). Introduction to Data Mining (Second Edition). Pearson Education, Inc. (o New York, NY: Pearson Education, Inc.)

1. ***Explique el concepto de “embedding” y autoencoder en el contexto de “Deep Learning”. DE ejemplos de uso.***

## **Conceptos de “embedding” y “autoencoder”**

Un embedding es una representación vectorial densa y de baja dimensión de palabras, frases, imágenes, audio, gráficos o cualquier otro tipo de dato en un espacio continuo. Su objetivo principal es transformar datos complejos y no estructurados en información numérica que los algoritmos de machine learning puedan entender y procesar.

Estos vectores capturan relaciones semánticas y contextuales, lo que le permite a los modelos entender las similitudes y diferencias entre las entidades más allá de una simple coincidencia de características. Los embeddings son vectores densos de números reales con dimensiones significativamente menores (típicamente entre 50 y 300 palabras), lo que disminuye la complejidad computacional e incrementa la eficiencia del almacenamiento. En este espacio vectorial, pueden realizarse operaciones aritméticas para identificar relaciones semánticas, como "rey - hombre + mujer" que puede aproximarse a "reina".

Los embeddings se utilizan para diversos tipos de datos, como palabras, texto, imágenes, audio y gráficos. En lo que se refiere a sus principales usos, se puede mencionar el Procesamiento de Lenguaje Natural (NLP) para tareas como la clasificación de texto, el análisis de sentimientos, la traducción automática, los sistemas de preguntas y respuestas (como los que usan BERT), y la búsqueda de documentos semánticamente similares con Doc2Vec. A su vez, son aplicados en sistemas de recomendación (ej. Netflix), visión por computador para reconocimiento y clasificación de imágenes, búsqueda de imágenes y reconocimiento facial (ej. FaceNet), análisis de redes sociales para detección de temas e influencia, y aplicaciones multimodales. Entre sus principales ventajas, los embeddings reducen la dimensionalidad, son capaces de capturar semántica compleja, mejoran significativamente la precisión de los modelos, y son flexibles y adaptables a diferentes idiomas y dominios.

Los autoencoders, por su parte, son un tipo especializado de redes neuronales planteadas para aprender a comprimir datos de entrada en una representación compacta y después reconstruirlos para que se asimilen lo más posible a la entrada original. Operan bajo un paradigma de aprendizaje no supervisado o auto-supervisado, ya que no necesitan datos etiquetados explícitamente, sino que usan la propia entrada original (o una versión modificada) como "verdad-terreno" para medir su rendimiento. Su propósito es revelar variables latentes o características esenciales de los datos, disminuyendo su dimensionalidad y filtrando ruido o redundancias.

La arquitectura de un autoencoder cuenta de tres partes principales:

1. El **Encoder** (Codificador): Disminuye la dimensionalidad de los datos de entrada mediante capas ocultas con un número progresivamente menor de nodos, capturando las características más importantes.
2. El **Bottleneck** (Cuello de Botella o Espacio Latente): Es la capa más pequeña de la red, que contiene la representación más comprimida de la entrada. Fuerza al modelo a priorizar las características más significativas, aprendiendo la estructura subyacente de los datos.
3. El **Decoder** (Decodificador): Toma la representación comprimida del espacio latente y la reconstruye a la forma de datos original, utilizando capas ocultas que aumentan progresivamente el número de nodos hasta la capa de salida.

El entrenamiento de un autoencoder se focaliza en minimizar la pérdida de reconstrucción (la diferencia entre la salida reconstruida y la entrada original) utilizando algoritmos como backpropagation. Su principal beneficio es el poder capturar correlaciones no lineales complejas, generalmente empleando funciones de activación no lineales como la sigmoidal. A diferencia de los modelos generales codificador-decodificador, donde la salida suele ser diferente de la entrada (ej. traducción de idiomas, segmentación de imágenes) y se entrenan de forma supervisada, los autoencoders se entrenan para reconstruir su propia entrada. Para evitar el sobreajuste y fomentar la extracción de características significativas, se utilizan técnicas de regularización como el ajuste del tamaño del código, el número de capas y nodos, y el uso de autoencoders incompletos, dispersos, contractivos o de eliminación de ruido.

Existen varios tipos de autoencoders, como los Denoising Autoencoders (que aprenden a reconstruir datos limpios a partir de entradas ruidosas), Sparse Autoencoders (que fuerzan la activación de pocas neuronas simultáneamente), y los Variational Autoencoders (VAEs), que son modelos generativos que aprenden distribuciones de probabilidad de los datos, permitiendo generar nuevas muestras de datos (imágenes o texto).

Entre los usos prácticos que tienen los autoencoders, se puede mencionar:

* Compresión de datos y reducción de dimensionalidad.
* Eliminación de ruido en imágenes y audio.
* Detección de anomalías y fraudes mediante la identificación de una alta pérdida de reconstrucción para datos inusuales.
* Reconocimiento facial.
* Generación de datos (especialmente con VAEs) para crear imágenes realistas o estructuras moleculares para medicamentos.
* Reconstrucción de imágenes, como rellenar elementos faltantes o colorear imágenes.
* Preentrenamiento no supervisado de modelos de deep learning.

En conclusión, los embeddings permiten una compresión semántica y contextual de los datos a través de representaciones vectoriales densas, por otro lado, los autoencoders facilitan una comprensión semántica y contextual rica de los datos mediante representaciones vectoriales densas, mientras que los autoencoders permiten la compresión, extracción de características y reconstrucción de datos. También sirven como modelos generativos y de detección de anomalías. Las dos técnicas transforman la forma en que las máquinas interactúan y aprenden de datos complejos, mejorando la precisión y eficiencia en una multitud de aplicaciones de inteligencia artificial.

Fuentes: OpenWebinars. (s.f.). Embeddings. Recuperado de <https://openwebinars.net/blog/embeddings/>

IBM. (s.f.). Embedding. Recuperado de https://www.ibm.com/mx-es/think/topics/embedding

1. ***De un tipo de red Neuronal que sirve para obtener clustering. La que elija, explique brevementecómo funciona.***

## **Red de Kohonnen – knet o Mapa Autoorganizativo**

Un ejemplo de tipo neuronal que sirve para obtener Clustering es la Red de Kohonen – knet o Mapa Autoorganizativo-. Esta red está diseñada para organizar un conjunto de datos en grupos distintos sin conocimiento previo de lo que son esos grupos. Los registros dentro de un mismo clúster son similares entre sí, mientras que son distintos de los registros en otros grupos.

En cuanto a su funcionamiento, cuenta con dos capas de neuronas: una capa de entrada y una capa de salida (también llamada "mapa de resultados" o "capa oculta-competitiva"). Todas las neuronas de entrada están conectadas a todas las neuronas de salida, y estas conexiones tienen pesos o fuerzas asociadas. El mapa de resultados es una red de neuronas bidimensional sin conexiones directas entre sus unidades. No obstante, cada neurona en la capa oculta-competitiva está conectada a otras neuronas contiguas de su vecindad, con conexiones activadoras para las neuronas vecinas e inhibitorias para las más distantes.

En lo que refiere al proceso de entrenamiento, es aprendizaje No Supervisado Competitivo, es decir que no utiliza un campo objetivo. Al inicio, todas las ponderaciones son aleatorias. Y los datos de entrada se presentan a la capa de entrada y se propagan a la capa de salida. La neurona de salida con la respuesta más fuerte se considera la "ganadora" para esta entrada. Esta neurona es aquella cuyo vector de ponderación es más cercano al vector del patrón de entrada. Cuando una unidad "gana" un registro, sus ponderaciones (junto con las de sus unidades vecinas en el mapa de resultados) se ajustan para que se parezca mejor al patrón de los valores de entrada de ese registro. La influencia de una neurona sobre otras disminuye con la distancia. Este proceso se repite presentando todos los registros de entrada y actualizando las ponderaciones de forma consecutiva, hasta que las modificaciones sean mínimas. A medida que avanza el entrenamiento, las ponderaciones de las unidades en el mapa se ajustan para formar un "mapa" bidimensional de los clústeres, de ahí el término "mapa autoorganizativo".

Una vez entrenada la red, los registros que son similares se agruparán cerca unos de otros en el mapa de resultados, mientras que los registros muy diferentes aparecerán separados. Las unidades que capturan muchas observaciones (llamadas unidades "fuertes") representan los posibles centros de los clústeres. El análisis del número de observaciones capturadas por cada unidad puede dar una idea del número adecuado de clústeres.

Las redes de Kohonen también se utilizan para la reducción de dimensionalidad, transformando los predictores originales en dos características derivadas que conservan las relaciones de similitud, de manera similar al análisis factorial o PCA.

En síntesis, la Red de Kohonen aprende a organizar los datos de entrada en un mapa bidimensional donde la proximidad en el mapa refleja la similitud de los datos originales, permitiendo así la identificación de clústeres de forma no supervisada.

Fuentes:

Numerentur. (s.f.). Red de Kohonen. Recuperado de https://numerentur.org/red-de-kohonen/

IBM. (s.f.). Nodo Kohonen (Kohonen Node). Recuperado de https://www.ibm.com/docs/es/spss-modeler/18.5.0?topic=models-kohonen-node

1. ***¿Qué es una convolución y como aplica en las redes neuronales? De un diagrama de las distintas capas de una red convolucional y explique.***

## **Convolución en redes neuronales**

Una convolución es una operación matemática que se aplica a una entrada, como una imagen bidimensional, utilizando un filtro o kernel bidimensional. Este filtro es una pequeña matriz con un conjunto de pesos que se ajustan durante el entrenamiento.

Las redes neuronales convolucionales (CNN) son un tipo especializado de redes neuronales, planteadas específicamente para procesar datos que tienen una topología similar a una cuadrícula, como imágenes, series de tiempo o datos de audio.

Básicamente, el funcionamiento es el siguiente: El filtro (kernel) se desliza sobre el ancho y el alto de la entrada (por ejemplo, una imagen), siguiendo un proceso de "ventana deslizante".

En cada posición donde se asienta el filtro, se calcula un producto punto (o suma ponderada) entre los valores de los píxeles de la región de la entrada cubierta por el filtro y los pesos del filtro.

El resultado de este producto punto se registra en una nueva matriz de salida, conocida como mapa de características (feature map), mapa de activación o característica convolucionada.

Los pesos dentro del filtro permanecen fijos a medida que se desliza por toda la entrada, lo que se conoce como intercambio de parámetro.

La selección de los valores del filtro (sus pesos) origina matrices de salida que realzan o suavizan ciertas partes de la entrada. Por ejemplo, en las primeras capas de una red convolucional, los filtros pueden aprender a detectar bordes o manchas de color.

Las redes neuronales convolucionales tienen en cuenta que el procesar imágenes, los píxeles cercanos están más fuertemente correlacionados que los píxeles más distantes.

Las convoluciones permiten a las CNN aprender a extraer características relevantes de los datos de forma automática y jerárquica, manteniendo la información espacial y reduciendo la complejidad del modelo, lo que las hace extremadamente eficaces en tareas de visión por computador y otras donde los datos tienen una topología de cuadrícula.

A diagram of a process of a pooling process

AI-generated content may be incorrect.Diagrama de las distintas capas de una red convolucional:

Fuente: Diagrama de CNN utilizado para mostrar la arquitectura de capas convolucionales y de reducción. Fuente: Diego Calvo (2017), “Red Neuronal Convolucional CNN”. Recuperado de diegocalvo.es.

La estructura entonces, podría ser:

(Imagen de entrada) → (Capa Convolucional 1: extrae bordes) → (Capa de Pooling: Reduce el tamaño, mantiene lo importante) → (Capa Convolucional 2: Extrae formas más complejas) → (Capa de Pooling) → (Capa Flatten: Convierte a vector 1D) → (Capa Fully Connected: Decide la clase final, por ej. "gato" o "perro").

Una red de neuronas convolucionales (CNN), diseñada para procesar datos de cuadrícula como imágenes, organiza sus neuronas en capas tridimensionales (ancho, alto y profundo). Una arquitectura CNN típica generalmente incluye un conjunto de entrada (INPUT) que contiene los valores sin procesar de los píxeles de la imagen (por ejemplo, el más grande x el más alto x 3 para RGB). A esta le sigue una capa convolucional (CONV), el pilar central, donde se aplican múltiples filtros (kernels) aprendibles para extraer características locales (como bordes o manchas de color) de la entrada, generando mapas de activación (feature maps) que se apilan para formar un volumen de salida. Después de cada operación CONV, una capa de activación (RELU) introduce no linealidad aplicando una función como max(0, x) elemento a elemento, sin cambiar las dimensiones del volumen, permitiendo a la red aprender relaciones más complejas.

Posteriormente, una capa de agrupación (POOL) reduce progresivamente el tamaño espacial de la representación (downsampling) mediante operaciones como MAX pooling, disminuyendo la cantidad de parámetros y cálculos, y ayudando a controlar el sobreajuste. Estas capas convolucionales y de agrupación pueden repetirse, volviendo la red más "profunda" y permitiendo aprender características jerárquicas más complejas. Finalmente, una capa de aplanamiento (FLATTEN) transforma el volumen 3D resultante en un vector 1D largo para ser procesado por las capas totalmente conectadas (FC o densas), que realizan la clasificación o regresión final, conectando cada neurona con todos los números del volumen anterior y a menudo utilizando una función de activación softmax para la salida de clasificación.

Autor(es). (s.f.). Título del capítulo o artículo. Recuperado de https://cdr-book.github.io/cap-redes-convol.html

Autor(es). (s.f.). Convoluciones CIFAR‑10 (informativo). Recuperado de https://dcain.etsin.upm.es/~carlos/bookAA/05.7\_RRNN\_Convoluciones\_CIFAR\_10\_INFORMATIVO.html

IBM. (s.f.). Convolutional Neural Networks. Recuperado de https://www.ibm.com/mx-es/think/topics/convolutional-neural-networks

1. ¿Qué es una red neuronal recurrente clásica? ¿Las redes LSTM en que se diferencian de las redes recurrentes clásicas? De un diagrama de su topología y explíquela brevemente. Cantidad de parámetros que utiliza en base a su configuración. Genere un ejemplo completo en Python de aplicación de una red LSTM para predecir una serie de tiempo arbitraria. Pruebe diferentes configuraciones de capas y cantidad de neuronas.

## **Diferencias entre RNN y LSTM**

Las redes **neuronales recurrentes (RNN clásicas)** son un tipo de arquitectura diseñada para procesar secuencias paso a paso, donde cada salida depende no solo del input actual, sino también del estado oculto anterior. El problema de las RNN clásicas es el desvanecimiento o explosión del gradiente, que limita su capacidad de aprender dependencias a largo plazo.

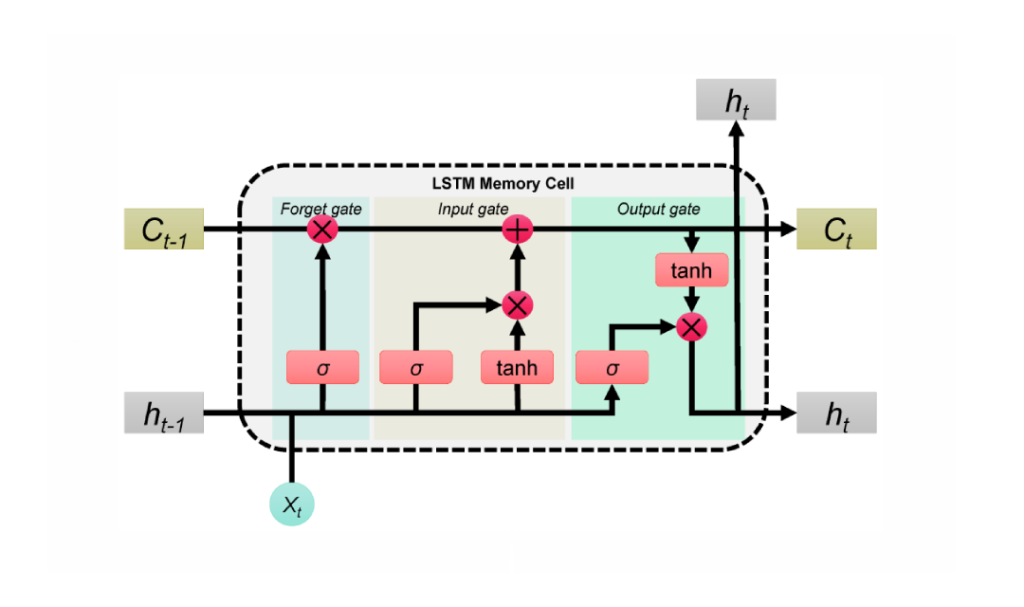
Por su parte, las **redes Long Short-Term Memory (LSTM)** fueron propuestas por Hochreiter y Schmidhuber (1997) para superar estas limitaciones, e introducen una estructura de celdas de memoria con tres compuertas que contronlan qué recordar, qué olvida y qué omitir:

* **Forget gate (ft)**: decide qué información descartar.
* **Input gate (it)**: decide qué nueva información almacenar.
* **Output gate (ot):** controla qué parte de la memoria pasa al siguiente estado.

Gracias a esta arquitectura, las LSTM pueden capturar dependencias de largo plazo en secuencias, resolviendo en gran medida el problema del gradiente, permitiendo moldear las dependencias de modo estable.

## **Topología típica de una LSTM**

La tipología clásica de estas redes se constituye por:

1. Capa de entrada secuencial.
2. Una o varias capas LSTM.
3. Capa densa final para la predicción.

## **Cantidad de parámetros que se utilizan**

La cantidad de parámetros de una capa LSTM con NiN entradas y Nh​ unidades ocultas se calcula como:

Donde el factor 4 corresponde a las cuatro compuertas internas de la celda.

## **Aplicación en Python**

import numpy as np

import tensorflow as tf

from tensorflow.keras import Sequential

from tensorflow.keras.layers import LSTM, Dense

from tensorflow.keras.callbacks import EarlyStopping

import matplotlib.pyplot as plt

# %%

# 0) Reproducibilidad

np.random.seed(42)

tf.random.set\_seed(42)

# %%

# 1) Serie sintética: y = sin(t) + ruido

T = 1600

t = np.linspace(0, 60\*np.pi, T)

y = np.sin(t) + 0.15\*np.random.randn(T) # cambialo por tu serie real si querés

# 2) Ventaneo para armar (X\_seq -> y\_target)

def make\_sequences(series, window=30, horizon=1):

X, Y = [], []

for i in range(len(series) - window - horizon + 1):

X.append(series[i:i+window])

Y.append(series[i+window: i+window+horizon])

X = np.array(X)[:, :, None] # (muestras, pasos, features=1)

Y = np.array(Y)[:, 0] # predecimos 1 paso adelante

return X, Y

WINDOW = 30

X, Y = make\_sequences(y, WINDOW, horizon=1)

# %%

# 3) Split train/val/test

n = len(X)

i\_train = int(n\*0.70)

i\_val = int(n\*0.85)

X\_train, y\_train = X[:i\_train], Y[:i\_train]

X\_val, y\_val = X[i\_train:i\_val], Y[i\_train:i\_val]

X\_test, y\_test = X[i\_val:], Y[i\_val:]

# %%

# 4) Helper para construir modelos con diferentes configuraciones

def build\_lstm(units\_list, input\_shape):

model = Sequential()

for i, u in enumerate(units\_list):

return\_seq = (i < len(units\_list)-1) # True salvo en la última capa

model.add(LSTM(u, return\_sequences=return\_seq,

input\_shape=input\_shape if i==0 else None))

model.add(Dense(1)) # salida escalar (1 paso)

model.compile(optimizer="adam", loss="mse")

return model

# %%

# 5) Configuraciones a comparar

configs = {

"LSTM\_32": [32], # 1 capa, 32 neuronas

"LSTM\_64": [64], # 1 capa, 64 neuronas

"Stacked\_50\_20": [50, 20], # 2 capas (50 -> 20)

"Stacked\_64\_64": [64, 64], # 2 capas (64 -> 64)

}

# %%

# 6) Entrenamiento, evaluación y predicción

es = EarlyStopping(patience=6, restore\_best\_weights=True, monitor="val\_loss")

histories = {}

preds = {}

metrics = {}

for name, units in configs.items():

print(f"\n=== Entrenando {name} ({units}) ===")

model = build\_lstm(units, input\_shape=(WINDOW, 1))

model.summary()

histories[name] = model.fit(

X\_train, y\_train,

validation\_data=(X\_val, y\_val),

epochs=60, batch\_size=32, verbose=0, callbacks=[es]

)

# Predicción one-step en test (desplazada contra y\_test)

yhat = model.predict(X\_test, verbose=0).squeeze()

preds[name] = yhat

rmse = np.sqrt(np.mean((yhat - y\_test)\*\*2))

mae = np.mean(np.abs(yhat - y\_test))

metrics[name] = (rmse, mae)

print(f"{name}: RMSE={rmse:.4f} | MAE={mae:.4f} | Parámetros={model.count\_params()}")

# %%

# 7) Visualización comparada (pred vs real) en TEST

plt.figure(figsize=(12,5))

plt.plot(y\_test, label="Real", linewidth=2)

for name, yhat in preds.items():

plt.plot(yhat, label=name, alpha=0.9)

plt.title("Predicción 1-paso adelante – Conjunto de test")

plt.legend()

plt.tight\_layout()

plt.show()

# %%

# 8) Tabla simple de métricas

print("\n=== Métricas en TEST (menor es mejor) ===")

for name, (rmse, mae) in metrics.items():

print(f"{name:>12} RMSE={rmse:.4f} MAE={mae:.4f}")

1. ***Describa el algoritmo encoder-decoder. Usos habituales.***

## **Algoritmo Encoder-Decoder**

El algoritmo encoder-decoder es un tipo de red neuronal que se utiliza principalmente en tareas donde tanto la entrada como la salida son secuencias, y estas secuencias pueden no tener la misma longitud. Esta arquitectura está planteada para procesar datos de entrada, comprimirlos en una representación compacta, y luego reconstruir una salida a partir de esa representación.

En cuanto a su funcionamiento,  la arquitectura de un modelo encoder-decode cuenta de tres componentes principales que buscan comprimir y luego reconstruir los datos.

1. El Encoder (codificador) procesa los datos de entrada y los convierte para que el modelo pueda entenderlo, para ellos reduce su dimensionalidad y captura las características importantes.

El codificador puede tener varias capas:

* La capa de entrada: donde los datos originales ingresan a la red (por ejemplo, imágenes);
* Las capas ocultas: Transforman los datos de entrada a partir de pesos y funciones de activación que capturan patrones importantes, para reducir progresivamente el tamaño y complejidad de los datos.
* La capa de Autoatención (Self-Attention Layer): Da soporte al codificador para orientarse en diferentes partes de los datos de entrada que son decisivos para comprender el contexto.

1. Bottleneck (Cuello de Botella o Espacio Latente / Vector de Contexto) es la capa más pequeña de la red porque es la versión más comprimida de los datos de entrada-. Se llama “Cuello de botella” porque obliga a la red a priorizar las características más importantes.

Como resultado, el codificador origina un vector comprimido, llamado representación latente, codificación, vector de contexto o espacio latente, que condensa las características más importantes de los dato de entrada, ya que filtra el ruido y las redundancias.

1. El Decoder o decodificador, tiene como función tomar la representación comprimida del espacio latente y reconstruirla de nuevo a la forma de datos original o a la salida deseada. También cuanta con varias capas.

* Las Capas Ocultas, expanden progresivamente el vector latente a un espacio de mayor dimensión, para restaurar la forma y los detalles originales de los datos.
* La *Capa de Autoatención (Self-Attention Layer),* le da lugar al decodificador para enfocarse en diferentes partes de la salida que ha generado.
* La *Capa de Atención Codificador-Decodificador (Encoder-Decoder Attention Layer)* le permite al decodificador concentrarse en partes relevantes de los datos de entrada para generar salidas más precisas.
* La *Red Neuronal de Avance (Feed-Forward Neural Network)* procesa la información para generar la salida final.
* La *Capa de Salida*: Origina la salida reconstruida.

Entre los usos habituales se pueden mencionar:

* Traducción automática: Toman una oración en un idioma como entrada y producen su traducción en otro idioma como salida;
* Resumen de texto: Una secuencia larga de texto se comprime y luego se decodifica en un resumen más corto y coherente;
* Subtítulos de imágenes: Una imagen es codificada en una representación latente, y el decodificador genera una secuencia de palabras que describe el contenido de la imagen.
* Reconocimiento de voz: Convierte el audio a texto;
* Chatbots: Utilizan encoder-decoder para entender preguntas y generar respuestas.
* Detección de anomalías y fraudes: Pueden identificar patrones inusuales o defectos en los datos;
* Reconocimiento facial: Pueden detectar anomalías o confirmar coincidencias auténticas

Fuentes: GeeksforGeeks. (s.f.). *EncoderDecoder Models*. Recuperado de<https://www.geeksforgeeks.org/nlp/encoder-decoder-models/>

Leon, J. (s.f.). *Investigación: Encoder y Decoder*. Scribd. Recuperado de<https://es.scribd.com/document/541668280/Investigacion-ENCODER-Y-DECODER-Jerry-Leon>

# **Ejercicio 2 – Support Vector Machine**

***Mencione y describa los tipos de VSM. Que parámetros externos necesita el algoritmo en su versión no lineal separable. ¿Qué característica especial tiene VSM en relación con el resto de los algoritmos de Machine Learning? ¿Como se adapta VSM clasificación, para poder usarlo como regresión? Interprete la discusión del siguiente link brevemente:*** <http://stackoverflow.com/questions/11632516/what-are-advantages-of-artificial-neural-networks-over-support-vector-machines>

## **Tipos de VSM**

Existen dos tipos de VSM según el tipo de problema: usados como clasificación o como regresión.

**Clasificación**

El objetivo es separar los datos en distintas clases, para esto se desea encontrar un hiperplano que divida los datos de manera que las clases queden lo más separadas posible. Un hiperplano es una frontera que separa los datos en diferentes clases. En 2D el hiperplano es una línea y en 3D es un plano.

El hiperplano se elige de tal forma que debe maximizar el margen entre las dos clases, es decir, la distancia desde el hiperplano hasta los puntos más cercanos.

Las máquinas de soporte vectorial usadas como clasificador también tienen dos tipos principales según el tipo de separación de los datos

1. **Versión Lineal:** Para los datos que son separables por una línea recta en 2D o un plano en más dimensiones. El hiperplano se elige de tal forma que maximice el margen entre la frontera y los puntos más cercanos.

Para conseguir esto se debe minimizar la función sujeta a una serie de restricciones lineales. Esto se puede resolver de dos maneras:

* + Versión Primal
  + Versión Dual

1. **Versión No Lineal:** Esta versión se aplica cuando las fronteras de los datos son no lineales. También existen dos enfoques para este problema dependiendo si los datos son separables o no completamente separables.

* **VSM con datos separables**: Cuando los datos no se pueden separar linealmente, se deben transformar las variables predictoras x hacia un espacio vectorial de más alta dimensión para poder separarlos linealmente.

Se deben llevar los datos de un input space (Espacio donde se ubican en un estado puro) a un feature space (Espacio donde se ubican las transformaciones de x), de tal forma que las operaciones no lineales en el input space sean lineales en el feature space.

Para conseguirlo y aliviar el costo computacional, se utiliza el “Kernel Trick”, que permite calcular el producto interno de las variables antes de llevarlas al feature space.

* + Algunos tipos de funciones kernel son:
  + Lineal
  + Polinomial
  + Radial basis function (RBF)
  + Exponencial RBF
* **VSM con Imperfecta separación**: En este caso, se permite el ruido o la imperfecta separación. Es decir, permite que existan errores en la clasificación, que algunos puntos crucen los hiperplanos, pero agregándoles una penalización, de tal forma que se busca maximizar el margen y minimizar el error.
  + Se agregan variables slack negativas y un término de penalización en la función objetivo.
  + Este es el caso más utilizado en la práctica.

**Regresión (Support Vector Regression SVR)**

El objetivo es predecir un valor numérico continuo. A diferencia de un clasificador, que solo puede asignar una clase discreta, la regresión puede tomar valores reales, estimando la dependencia funcional de una variable dependiente Y con respecto a una variable de entrada X. En lugar de encontrar un hiperplano que separe clases, la SVR busca una función que ajuste los datos dentro de un margen de error permitido. La aproximación se mide mediante un error de predicción, a diferencia del margen que se utiliza en clasificación.

La principal diferencia con la regresión clásica es la introducción de una nueva función de pérdida: la función lineal de pérdida de Vapnik, que incluye una zona de E -insensibilidad. Esta zona define un “tubo” alrededor de la función donde los errores menores a E no se penalizan. Los puntos que quedan fuera de este margen son los que finalmente determinan el modelo, convirtiéndose en los vectores de soporte en el contexto de regresión.

* **SVR Lineal:** Se utiliza cuando los datos presentan una relación lineal entre la variable dependiente Y y las variables de entrada X. Busca un tubo de E-insensibilidad alrededor de una línea recta donde los errores dentro del tubo no se penalizan.
* **SRV no lineal**: Se aplica cuando la relación entre Y y X no es lineal. En este caso, se usa un kernel que proyecta los datos a un espacio de mayor dimensión donde se encuentre un hiperplano lineal en ese espacio transformado. El tubo de E-insensibilidad se aplica al espacio transformado.

## **Parámetros del algoritmo en su versión no lineal separable**

SVM en su versión lineal no separable tiene los siguientes parámetros externos o hiperparámetros que se deben definir antes de entrenar el algoritmo:

* C: Parámetro de penalización que controla el equilibrio entre maximizar el margen y minimizar los errores de clasificación.
  1. Un C bajo da un margen más grande, una mayor generalización y tolerancia a errores.
  2. Un C alto permite menos errores, ajusta más el modelo a los datos y trae riesgo de overfitting.
* Kermel: Es la función que permite transformar los datos de un espacio original a uno de mayor dimensión para hacerlos separables. Algunos tipos de funciones kernel son: lineal, polinomial, Radial Basis Function (RBF) o RBF Exponencial.
* Dependiendo de la función kernel elegida, se deben definir parámetros adicionales
  1. Gamma (Usado en RBF y Polinomial) Determina hasta dónde llega la influencia de cada punto de datos. Un gamma alto da a cada punto una influencia muy local y un gamma alto da una influencia más global.
  2. Degree (Usado en kernel polinomial) Define el grado del polinomio.

## **Características de VSM vs. Otros algoritmos de Machine Learning**

Algunos aspectos que distinguen a VSM de otros algoritmos de Machine Learning son los siguientes.

* La naturaleza de sus parámetros. VSM es un modelo no paramétrico porque el número de parámetros que define el modelo depende de los datos, de la cantidad de vectores soporte que define. La capacidad es qué tan flexible es el modelo para adaptarse a los diferentes patrones de los datos. La capacidad en los VSM está definida por la cantidad de vectores de soporte que seleccionan y sus valores, por lo tanto, los SVM ajustan su capacidad en función de la complejidad de los datos.
* A diferencia de la mayoría de algoritmos donde se supone la normalidad de los datos, las leyes de generalización de SVM podrían alejarse sustancialmente de la normalidad.
* Este algoritmo es específico para problemas Gaussianos. En este contexto, se define problema Gaussiano a aquel cuyos datos tengan clases compactas y más o menos redondeadas, de tal forma que sean separables por una función kernel. En caso de que el problema sea no Gaussianos se debe considerar otros algoritmos.

## **Adaptación de VSM en clasificación, para poder usarlo como regresión**

En un algoritmo de regresión se estima la dependencia funcional de una variable dependiente Y con respecto a variables de entrada X. Y puede tomar valores continuos, valores reales.

Algunas modificaciones que realiza SVM para ser usado como algoritmo de regresión son:

* En clasificación el objetivo es encontrar un hiperplano que separa dos clases con el máximo margen. En regresión se desea ajustar una función que esté lo más cerca posible de los puntos de entrenamiento, con un margen de error tolerado.
* En clasificación, los multiplicadores de Lagrange están asociados a los datos que queden en el margen (vectores de soporte). En regresión, aparecen dos multiplicadores de Lagrange por cada lado del tubo de tolerancia.
* El hessiano, es decir, la matriz cuadrática, tiene el doble de dimensión ya que cuenta con dos multiplicadores por cada punto de entrenamiento.
* Los vectores también tienen el doble de tamaño, por la misma razón.

## **Interpretación sobre ventajas de Redes Neuronales sobre VSM**

El artículo es una discusión sobre cuáles son las ventajas que tienen las redes neuronales por sobre las máquinas de soporte vectorial. Específicamente, se desea comparar en qué casos las redes neuronales de perceptrón multicapa serían más útiles que las SVM.

Primero se mencionan algunas ventajas de las SVM, y es que evitan las dos principales debilidades de las ANN:

* Las ANN por lo general encuentran el mínimo local en lugar del mínimo global, lo que provoca que en muchos casos pierdan de vista el panorama principal o que no logren generalizar de manera correcta.
* Las ANN tienen una tendencia al sobreajuste, es decir, pueden considerar el ruido como parte de los patrones entre los datos.

Por otro lado, algunas ventajas que los ANN presentan por sobre los VSM son:

* Las ANN son modelos paramétricos de tamaño fijo, a diferencia de las SVM (al menos en los modelos no lineales). En ANN se tiene la cantidad de capas dependiendo de la cantidad de atributos, más los parámetros de base. En cambio, en SVM, la cantidad de vectores de soporte se obtiene del set de entrenamiento. El tamaño del modelo crece de manera lineal, lo que constituye un problema especialmente fuerte en el procesamiento de lenguaje natural.
* Otra ventaja es que las ANN pueden tener cualquier cantidad de respuestas, a diferencia de las SVM, que solo pueden tener una. Si se desea tener un clasificador de n clases con SVM se debe construir n algoritmos y entrenarlos todos para reconocer solo una de las clases.
  + En caso de que el objetivo sea que cada clase sea mutuamente excluyente, donde las clases no estén relacionadas entre sí, tiene sentido entrenar varios modelos SVM en lugar de una ANN.
  + En cambio, en el caso donde las clases están interrelacionadas y dependen unas de otras, lo mejor sería utilizar solo un modelo ANN. Por ejemplo, si se desea modelar el balance hormonal de una persona basado en factores fisiológicos.

# **Ejercicio 3 – Statistical Learning Theory**

***Explique brevemente de que trata el campo de Statistical Learning Theory. ¿Qué indicador muestra la complejidad de un data set usando el algoritmo SVM? ¿Qué es la capacidad de un algoritmo y que manera heurística propone Vapnik–Chervonenkis para estimarla, dado un data set de 1000 observaciones con k variables – deje de lado el caso particular de SVM en el cual existe una forma matemática de estimarla?***

## **Concepto de Statistical Learning Theory**

El campo de Statistical Learning Theory (SLT), o Teoría del Aprendizaje Estadístico, construye un marco matemático para comprender y desarrollar el aprendizaje automático. Su foco está principalmente en el aprendizaje supervisado y la clasificación binaria, abordando preguntas sobre la calidad de los clasificadores y cómo garantizar el buen desempeño con la población general.

## **Características claves que se abordan con la SLT:**

* Consistencia de un clasificador: exige una consistencia universal donde el algoritmo no solo funciona para una distribución fija sino para cualquiera de probabilidad p.
* Distinción entre riesgo empírico y riesgo real: asegura que un clasificador con un riesgo empírico bajo (sobre los datos de entrenamiento) también cuente con un riesgo real bajo (sobre la población general).
* Problema de la inconsistencia de la minimización del riesgo empírico (ERM): restringir la capacidad del espacio de funciones donde se toma el clasificador para que no sea excesivamente grande. Como aporte de Vapnik-Chervonenkis surge la convergencia uniforme, la cual describe que para que haya esa consistencia de ERM, la diferencia entre el riesgo real y el riesgo empírico debe ser menor a un valor ε para todas las funciones en el espacio F.

## **Indicador de complejidad de un dataset usando el algoritmo SVM**

La Dimensión VC (Vapnik-Chervonenkis) es el indicador que cuantifica la complejidad de un dataset usando los algoritmos de Support Vector Machine (SVM) en el contexto de SLT. Se trata de una propiedad de un conjunto de funciones que mide la complejidad del espacio de hipótesis. Y no solo se basa en el número de hipótesis distintas, sino además en la máxima cantidad de observaciones de X que pueden ser completamente discriminadas.

Una elevada Dimensión VC presenta una mayor capacidad del modelo a ajustarse a los datos, que controlado no lleva al overfitting. Por lo cual, los SVM exploran un punto de equilibrio entre un riesgo empírico bajo y una capacidad controlada.

## **Capacidad de un algoritmo y manera heurística de Vapnik-Chervonenkis para estimarla**

La capacidad de un algoritmo es la habilidad que tiene para representar diferentes conceptos relacionados con la variable objetivo, es decir, la flexibilidad a ajustarse a un dataset. Si la capacidad es muy alta existe riesgo de overfitting, por el contrario si es muy baja no puede capturar la complejidad subyacente de los datos.

Para estimar la capacidad del algoritmo según una medida heurística, Vapnik-Chervonenkis expone el coeficiente de shattering N (F,n). El mismo cuantifica el número máximo de distintas formas en que el algoritmo puede etiquetar una muestra en dos clases. Esta medida heurística se relaciona con la Dimensión VC h ya que es una cota superior en el logaritmo de N(F,n) y es la cantidad máxima de puntos que una clase de funciones puede shatterear.

A graph with lines and black text

AI-generated content may be incorrect.Por eso para un dataset de 1000 observaciones con k variables, Vapnik-Chervonenkis propone basarse en la Dimensión VC h de la clase de funciones que el algoritmo está utilizando.

La importancia se evidencia en la cota de generalización de Vapnik-Chervonenkis: R(f) <= Remp(f) + sqrt( (h \* (log(2n/h) + 1) - log(eta/4)) / n ), donde h es la Dimensión VC, n es el tamaño de la muestra (1000 en este caso), y eta es la probabilidad de equivocarse. La cota sugiere que si el riesgo empírico y el término de la raíz cuadrada son chicos, entonces el riesgo real tendrá altas probabilidades de ser bajo. A pesar de que calcular h propone ser un desafío, un h reducido más un riesgo empírico bajo, permiten entender que el algoritmo ha aprendido a generalizar.

Fuentes

Vapnik (1998) – Statistical Learning Theory

SVM Universidad Austral.pdf

SLT Universidad Austral1.pdf

# **Ejercicio 4 – Algoritmos Genéticos**

***Escriba el pseudocódigo de algoritmos genéticos. Explique brevemente las principales diferencias entre AG binarios y reales. De 3 casos de aplicación de AG y sus respectivas fuentes (paper, autor, fecha), con breve explicación. Como lo implementaría usando programas disponibles open-source (cita y breve explicación).***

## **Qué son los algoritmos genéticos**

Los algoritmos genéticos (AG) son técnicas de búsqueda y optimización que se inspiran tanto en la teoría de la evolución de Darwin como en los principios de la genética natural. De este modo, la lógica que utilizan para resolver problemas complejos de optimización es similar a la selección natural.

La idea es simple, partiendo de una población de posibles soluciones (o individuos) a un problema que compiten entre sí, aquellos que son mejores son los que se reproducen combinando información. De esta manera, se logran introducir mutaciones paulatinas que mantienen la diversidad de la población pero que, con el tiempo, logra evolucionar hacia soluciones mejores.

Conceptos clave:

* Individuos o cromosomas: representan a cada posible solución al problema, pueden estar codificados de manera binaria (0 y 1) o de manera real (con números continuos)
* Gen: es cada uno de los componentes del cromosoma, por ejemplo, un número
* Población: es el conjunto de individuos en una generación, el tamaño de esta afecta a la diversidad y velocidad de convergencia
* Fitness (o aptitud): es la función que mide qué tan buena es una solución, por lo que es el “criterio de supervivencia” bajo el cual se define quienes tienen más probabilidad de reproducirse
* Operadores genéticos:
  + Selección: elige quiénes se van a reproducir
  + Crossover: combina pares de individuos para generar la descendencia
  + Mutación: introduce los cambios aleatorios en los genes que mantienen la diversidad
* Elitismo: aquellos individuos de una generación que son mejores son copiados a la siguiente garantizando que la calidad no empeore

## **Pseudocódigo de un algoritmo genético**

La dinámica de un algoritmo genético es la siguiente:

1. Se genera aleatoriamente a una población de posibles soluciones
2. Se evalúa la aptitud o fitness de cada uno de los individuos que la componen
3. Se seleccionan a los más aptos para reproducirse
4. Se da la reproducción donde se aplican crossover y mutación para poder crear individuos nuevos
5. Se forma la nueva generación incluyendo élites si aplica

Este ciclo se itera las veces que sean necesarias hasta lograr un criterio de parada que puede ser una cantidad puntual de generaciones o un fitness mínimo, por ejemplo.

De este modo, podemos definir el siguiente pseudocódigo:

Entrada:

* Función de fitness f(x) a maximizar (o minimizar)
* Espacio de búsqueda y codificación de individuos
* Tamaño de población N, prob. de cruce pc, prob. de mutación pm
* Criterio de parada (nº de generaciones G, umbral de fitness, etc.)

Algoritmo:

1. Inicializar población P0 con N individuos al azar
2. Evaluar fitness de cada individuo en Pt
3. (Mientras no se cumpla el criterio de parada)
   1. Selección: elegir padres de Pt según fitness
   2. Reproducción:
      1. Cruce con prob. pc → generar descendencia
      2. Mutación con prob. pm → perturbar genes
   3. Formar población candidata C a partir de descendencia (opcional: elitismo, reemplazo generacional/estacionario)
   4. Evaluar fitness de C
   5. Actualizar Pt+1 ← C (y élites si aplica)
   6. t ← t + 1
4. Devolver el mejor individuo encontrado

Salida:

* Mejor solución y su fitness

## **Diferencias principales entre algoritmos genéticos reales y binarios**

Podemos clasificar a los algoritmos genético según la forma en la que sus cromosomas son representados, siendo las codificaciones binaria y real las más comunes y utilizadas.

Definimos a un algoritmo genético como binario cuando cada cromosoma se representa como una cada de bits que se compone de 0 y 1. Esto resulta muy útil cuando el problema es discreto o combinatorio como puede ser la selección de variables. Los operadores genéticos se aplican de las siguientes maneras:

* El crossover consiste en intercambios de subcadenas entre padres
* La mutación, por su lado, implica la alteración de un bit de 0 a 1 o viceversa

A pesar de que estos algoritmos son bien simples pero robustos, pueden requerir de cadenas muy largas para lograr un alto nivel de precisión en problemas continuos, incrementando su complejidad.

Por otro lado, un algoritmo genético real representa a los cromosomas directamente como vectores de números ente los valores reales. Esto permite trabajar de manera más natural en problemas de optimización continua como lo es el ajuste de parámetros en redes neuronales. En este caso, los operadores genéticos se adaptan de la siguiente manera:

* El crossover puede implementarse como interpolaciones o combinaciones lineales de los valores de los padres
* La mutación se aplica añadiendo pequeñas perturbaciones aleatorias como el ruido gaussiano

En síntesis, la diferencia esencial entre ambos enfoques radica en la naturaleza de la representación del cromosoma y los operadores asociados. Mientras que los AG binarios son más adecuados para problemas discretos, los AG reales ofrecen ventajas significativas en contextos de optimización numérica y continua, al trabajar directamente en el espacio de búsqueda sin necesidad de mapeos intermedios.

## **Casos de aplicación**

**Selección de variables (Feature Selection) en Machine Learning**

Se aplican AG para seleccionar subconjuntos óptimos de variables predictoras para problemas de clasificación. Esto permitió equilibrar la reducción de dimensionalidad con la mejora en la precisión, siendo esta una de las primeras aplicaciones de AG en modelos de machine learning.

Fuente: W. Siedlecki and J. Sklansky. 1989. A note on genetic algorithms for large-scale feature selection. Pattern Recogn. Lett. 10, 5 (November 1989), 335–347. <https://doi.org/10.1016/0167-8655(89)90037-8>

**Optimización de parámetros en redes neuronales**

En este caso, se utiliza un AG para optimizar los pesos de una red neuronal en lugar de aplicar backpropagation. Este método demuestra las ventajas que existen en evitar mínimos locales y mejorar de este modo la robustez en espacios de búsqueda no convexos.

Fuente: D. J. Montana and L. Davis, “Training Feed-Forward Neural Networks Using Genetic Algorithms,” Proceedings of the 11th International Joint Conference on Artificial Intelligence, San Mateo, 20-25 August 1989, pp. 762-767.

**Optimización del tráfico urbano**

Se aplicaron AG como una estrategia de optimización del tráfico definiendo a cada cromosoma como un conjunto de parámetros de temporización de los semáforos. La función fitness en este caso evalúa el rendimiento de esta configuración en términos de indicadores como el tiempo promedio de espera, densidad vehicular y flujo de la red. A través de los operadores genéticos de selección, cruce y mutación, los AG generan configuraciones cada vez más eficientes, explorando un espacio de soluciones extremadamente grande.

Fuente: Halim Ceylan, Michael G.H Bell, Traffic signal timing optimisation based on genetic algorithm approach, including drivers’ routing, Transportation Research Part B: Methodological, Volume 38, Issue 4, 2004, Pages 329-342, ISSN 0191-2615, <https://doi.org/10.1016/S0191-2615(03)00015-8> (<https://www.sciencedirect.com/science/article/pii/S0191261503000158>)

## **Implementación open-source**

* Librería DEAP en Python es una librería muy conocida, se pueden construir AG de manera modular, incluyendo AG reales, binarios y permutaciones. Permite definir funciones de fitness, cromosomas, operadores de selección, cruce y mutación de acuerdo con el problema que se busque resolver.
* PyGAD en Python, otra librería ampliamente utilizada en el área de machine learning y optimización de funciones. Permite la integración con redes neuronales para la optimización de pesos, tiene diferentes tipos de selección y cruce, así como la posibilidad de personalizar la función objetivo.
* GA en R, este paquete permite trabajar con AG binarios, reales y permutaciones también, con opciones de personalización en los operadores genéticos y criterios de convergencia.

# **Ejercicio 5 – Componentes Principales**

***Explique brevemente la utilización de Componente principal en Machine Learning. Investigue la implicancia de utilizar Kernel en Componentes principales (citar fuentes y explique con sus palabras) ¿Qué significa el “Truco del Kernel”. De ejemplos de los kernel más utilizados. Investigue si existen métodos sencillos que se pueden utilizar para seleccionar el Kernel apropiado en un determinado data set. Mencione fuentes y conclusiones brevemente.***

## **Qué es PCA**

Componente principal (PCA) es una técnica que busca encontrar nuevas variables (los componentes) que son combinaciones lineales de las originales y que explican la mayor varianza posible, en orden descendente. PCA se utiliza en Machine Learning primordialmente como un método de preprocesamiento y disminución de la dimensionalidad.

Básicamente, cuando los dataset tienen gran cantidad de variables, algunas o muchas de ellas, pueden estar correlacionadas o aportar poco a la varianza total. De esta manera, PCA busca transformar a las variables originales en un conjunto más reducido de componentes principales, que así como se mencionó anteriormente, son combinaciones lineales de las variables originales y explican la mayor parte de la variabilidad.

Sin embargo, PCA funciona bien siempre y cuando las relaciones de los datos sean esencialmente lineales. Cuando esto no sucede, el truco de Kernel puede ser una opción. El mismo permite proyectar los datos en un espacio de características de dimensión más alta, donde patrones no lineales pueden volverse lineales.

## **Kernel**

Un kernel es una función matemática que permite medir la similitud o cercanía entre dos datos y se obtiene el producto interno entre esos datos en ese espacio de forma implícita. Algunos ejemplos de los más utilizados son:

* **Kernel Lineal**, el cual mide la similitud con el producto interno. Se utiliza cuando los datos son linealmente separables y el número de variables es muy elevado en relación con el total de muestras.
* **Kernel polinómico**, que introduce relaciones no lineales de grado d y se usa cuando hay relación polinómica entre las variables.
* **Kernel RBF (Radial Basis Function o Gaussiano)** el cual mide la similitud decreciente con la distancia euclidiana. Se usa cuando la frontera de decisión es altamente no lineal. Es de gran utilización porque es muy flexible.
* **Kernel Sigmoide**, el cual se basa en las funciones de activación de las redes neuronales. Se utiliza cuando se buscan propiedades similares en redes neuronales.
* **Kernels personalizados** como el String kernel, que mide similitud entre cadenas de texto, el Graph kernel el cual mide similitud entre grafos y el Chi-cuadrado kernel, muy utilizado en histogramas de imágenes.

En cuanto a los métodos sencillos para seleccionar un kernel adecuado para un determinado dataset, se pueden nombrar:

* **Validación Cruzada (Cross-Validation)** que consiste en probar diferentes kernels (lineal, polinómico, RBF) y comparar el rendimiento con validación cruzada. Luego se miden las métricas (accuracy, F1, MSE), y se determina que el mejor kernel es aquel con mejor promedio y menos varianza en CV.
* **Grid Search o Random Search** de hiperparametros el cual implica probar todas las combinaciones posibles de un conjunto de valores predefinidos para los hiperparametros de un modelo. Básicamente se define un espacio de búsqueda discreto para cada hiperparámetro, se define una ¨grid¨ (rejilla) con todas las combinaciones posibles, se entrena y valida el modelo usando validación cruzada y se escoge la combinación con el mejor desempeño.
* **Heurísticas** **iniciales** las cuales son reglas prácticas usadas para determinar valores de partida razonables de hiperparametros antes de hacer otra búsqueda. Por ejemplo, para el kernel RBF, se calcula la mediana de las distancias cuadradas entre todas las muestras.
* **Evaluación de visualización**, para datasets chicos se puede aplicar kernel PCA con distintos kernels y visualizar 2D/3D. La mejor opción será el kernel que muestre una separación más clara de grupos.

A modo de conclusión, se puede mencionar que PCA es una herramienta clave en el reprocesamiento de Machine Learning y su importancia radica en que permite reducir la dimensionalidad de los datos y hacer los modelos más simples, con la ventaja de que mantiene la mayor parte de la variabilidad presente en las variables originales. Por su parte, su punto débil es que no resulta suficiente para cuando la estructura de los datos no es lineal. Cuando esto sucede, se puede emplear “ el truco del Kernel”, que permite proyectar los datos en espacios de mayor dimensión donde las relaciones no lineales se vuelven lineales. Esta extensión amplía las posibilidades del análisis, mejorando la capacidad de los modelos para identificar patrones complejos.

El uso de diferentes funciones kernel como el lineal, polinómico, RBF, sigmoide y otros más especializados, permite adaptar la transformación de los datos a la naturaleza del problema. Sin embargo, elegir el kernel adecuado no es trivial. Métodos como la validación cruzada, búsqueda de hiperparámetros (Grid Search o Random Search), heurísticas iniciales y visualización en espacios reducidos son estrategias prácticas para orientar esta elección de forma sistemática.

De esta manera, combinar PCA con el uso de kernels brinda una poderosa alternativa para el análisis de datos complejos, que permite aprovechar tanto la eficiencia de la disminución de dimensionalidad como la flexibilidad del aprendizaje no lineal.

Fuentes utilizadas:

“Nonlinear component analysis as a kernel eigenvalue problem” de Schölkopf, Smola & Müller (1998)

"Support Vector Machines and Kernel Algorithms" de Schölkopf & Smola (marzo de 2002).

Aprende Estadísticas Fácilmente. (s. f.). ¿Qué es el truco del kernel? Recuperado de https://es.statisticseasily.com/glosario/%C2%BFQu%C3%A9-es-el-truco-del-kernel%3F/

# **Ejercicio 6 – Análisis de Supervivencia**

***Análisis de supervivencia. Explique porque este enfoque es superior a los métodos de regresión para estimar días a un evento. Que tipos existen y que implementaciones en Python/R conoce.***

## **Qué es el análisis de supervivencia**

El enfoque del análisis de supervivencia funciona mejor que uno de regresión para la estimación de tiempo y eventos principalmente porque modela el tiempo como una variable clave. En la regresión logística, solo se puede saber si el evento ocurrió o no. El análisis de supervivencia se centra en el tiempo hasta el evento y utiliza distribuciones que se adaptan mejor a estos datos como exponencial, Weibull entre otras.

También incorpora el concepto de datos censurados, que son individuos donde el evento no ocurre en el periodo de tiempo definido. Los métodos de regresión tradicionales no pueden manejar estos tipos de datos porque cada observación debe tener un valor respuesta, es decir, una estimación de periodos hasta el evento, dato que no siempre se tiene.

Además, este enfoque permite calcular la función de supervivencia (probabilidad de que un individuo no experimente el evento después de cierto tiempo) y la función de riesgo (la tasa instantánea de ocurrencia del evento en un momento dado). Estas funciones son específicas del análisis de supervivencia y proporcionan una información mucho más rica que una simple estimación de días hasta el evento.

Finalmente, el análisis de supervivencia facilita la comparación entre grupos y covariables, lo que permite evaluar cómo distintas características afectan la probabilidad de experimentar el evento. En contraste, los métodos de regresión suelen centrarse únicamente en patrones globales y pierden matices importantes relacionados con el tiempo y la dinámica del riesgo.

## **Tipos de análisis de supervivencia**

**Modelos Paramétricos:** Se hacen suposiciones acerca del tiempo de supervivencia, por lo que se especifica por completo la función de riesgo. Son útiles cuando se cumplen los supuestos sobre la distribución y se desea realizar predicciones más allá del tiempo observado.

Principales distribuciones utilizadas:

* Exponencial: supone un riesgo constante en el tiempo.
* Weibull: permite riesgos crecientes o decrecientes, es muy flexible.
* Log-normal: útil cuando los tiempos siguen una distribución asimétrica.

**Modelos Semi-Paramétricos:** Se utiliza principalmente para explorar las relaciones entre el tiempo de supervivencia y una o más variables exploratorias.

* Se usa el modelo de Riesgos Proporcionales de COX
* No requiere asumir una distribución específica para los tiempos de supervivencia. Se obtienen buenas estimaciones de coeficientes de regresión y de riesgo. Se obtiene también una curva de supervivencia ajustada.

**Modelos no paramétricos:** Estiman la función de supervivencia de los datos observados, sin hacer suposiciones de la distribución de tiempos de supervivencia.

Son útiles cuando se desconoce la forma funcional de la distribución o cuando los supuestos paramétricos no se cumplen.

No se pueden hacer predicciones más allá del tiempo observado.

* Random Forest Suvivasl: En cada nodo se estima la función de riesgo con el estimador de Nelson-Aalen y luego se promedian los resultados de todos los árboles.
* Kaplan-Meier Estimator: estima la función de supervivencia de manera escalonada, muy utilizado para comparar grupos.
* Nelson-Aalen Estimator: estima la función de riesgo acumulada, alternativa a Kaplan-Meier.

## **Implementación en Python**

Kaplan meier estimator

# Importando el dataset

from sksurv.datasets import load\_veterans\_lung\_cancer

data\_x, data\_y = load\_veterans\_lung\_cancer()

# creando el dataframe

import pandas as pd

pd.DataFrame.from\_records(data\_y[[11, 5, 32, 13, 23]], index=range(1, 6))

import matplotlib.pyplot as plt

# function de supervivencia estimador Kaplan meier

from sksurv.nonparametric import kaplan\_meier\_estimator

time, survival\_prob, conf\_int = kaplan\_meier\_estimator(

data\_y["Status"], data\_y["Survival\_in\_days"], conf\_type="log-log"

)

plt.step(time, survival\_prob, where="post")

plt.fill\_between(time, conf\_int[0], conf\_int[1], alpha=0.25, step="post")

plt.ylim(0, 1)

plt.ylabel(r"est. probability of survival $\hat{S}(t)$")

plt.xlabel("time $t$")

# Analisis de supervivencia estratificando por variable

for value in data\_x["Celltype"].unique():

mask = data\_x["Celltype"] == value

time\_cell, survival\_prob\_cell, conf\_int = kaplan\_meier\_estimator(

data\_y["Status"][mask], data\_y["Survival\_in\_days"][mask], conf\_type="log-log"

)

plt.step(time\_cell, survival\_prob\_cell, where="post", label=f"{value} (n = {mask.sum()})")

plt.fill\_between(time\_cell, conf\_int[0], conf\_int[1], alpha=0.25, step="post")

plt.ylim(0, 1)

plt.ylabel(r"est. probability of survival $\hat{S}(t)$")

plt.xlabel("time $t$")

plt.legend(loc="best")

# **Ejercicio 7 – Clustering**

***Mencione las técnicas de pre-proceso que existen en clustering para visualizar la potencial existencia de grupos. Explique brevemente 1 de ellas. ¿Habiendo aplicado alguna técnica de clustering y obteniendo los grupos finales, de qué manera puede estimar que diferencias existen en los perfiles de los grupos obtenidos ¿– Explicar los resultados. Ayuda: utilice el último capítulo de la guía de clustering (Assessment Clustering) En que consiste el algoritmo t-sne. ¿para que se utiliza? De un ejemplo de implementación en Python ¿Qué limitantes tiene? ¿Qué ventajas tiene?***

## **Análisis de Componentes Principales (PCA)**

PCA es una técnica de reducción de dimensionalidad ampliamente utilizada para visualizar datos de alta dimensión. Su objetivo es transformar un conjunto de variables posiblemente correlacionadas en un conjunto de nuevas variables no correlacionadas llamadas componentes principales (PC). Estos componentes se construyen de manera que el primer componente principal explica la mayor cantidad de varianza posible en los datos, el segundo componente explica la mayor cantidad de varianza restante, y así sucesivamente.

El corazón del algoritmo PCA es identificar las direcciones (autovectores) en el espacio de datos donde la variación es máxima, y luego proyectar los datos sobre estas nuevas direcciones. Al conservar solo los primeros componentes principales (por ejemplo, los dos o tres primeros), que explican la mayoría de las variaciones, se puede reducir la dimensionalidad del conjunto de datos y permitir su visualización en un espacio bidimensional o tridimensional. Esta visualización puede revelar patrones de agrupamiento o separación que no son evidentes en el espacio original de alta dimensión. PCA es una proyección lineal, lo que significa que "corta" los datos a través de un plano determinado para visualizarlos.

Formas de estimar las diferencias entre los perfiles obtenidos

Una vez que se han aplicado técnicas de clustering y se han obtenido los grupos finales, el siguiente paso crucial es el "Cluster Profiling" para estimar y entender las diferencias en los perfiles de estos grupos. El perfilado de clústeres nos permite asignar una etiqueta descriptiva a cada grupo, como "inversores jóvenes de alto patrimonio" o "usuarios intensivos de cajeros automáticos".

Las fuentes describen dos variantes principales para perfilar los clústeres:

1. Evaluación de diferencias significativas entre los vectores de medias (centroides) de los clústeres:
   * **Generar los vectores de medias**: Se calculan los centroides de cada clúster para todas las variables de entrada. Por ejemplo, se podría observar que los miembros del clúster 2 tienen un ingreso mensual promedio sustancialmente más alto que los de otros clústeres.
   * **Realizar MANOVA (Análisis Multivariado de Varianza)**: Para determinar si los vectores de medias son significativamente diferentes entre sí en su conjunto, controlando el error de tipo I a nivel del experimento. Si el MANOVA es significativo (p < 0.0001, por ejemplo), indica que existen diferencias significativas entre los centroides de los clústeres.
   * **Realizar ANOVA (Análisis de Varianza) y CDA (Análisis Discriminante Canónico)**: Si el MANOVA es significativo, el ANOVA se usa para identificar cuáles variables individuales son responsables de estas diferencias. Se pueden usar pruebas post-hoc como la de Tukey para hacer comparaciones por pares entre los clústeres para cada variable. Las variables que muestran diferencias significativas pueden ser usadas como etiquetas de clase, mientras que aquellas que no difieren significativamente podrían ser eliminadas del conjunto de variables de entrada.

**Ejemplo de ANOVA**: Si el ingreso mensual (income) es significativamente diferente entre los clústeres 1 y 2, pero no entre 2 y 3, ni entre 1 y 3, entonces el ingreso mensual puede distinguir el clúster 1 del 2, pero no identifica de forma única a ninguno. Sin embargo, si la variable "tiempo de residencia actual" (atres) es significativamente diferente para el clúster 2 en comparación con el 1 y el 3, y los clústeres 1 y 3 no difieren, entonces "atres" puede definir de forma única la pertenencia al clúster 2. Si "número de transacciones de cajero automático" (atmct) muestra diferencias significativas entre todos los clústeres, entonces cada clúster podría etiquetarse de forma única por el nivel de uso del cajero (ej., "usuarios moderados", "usuarios ligeros", "usuarios intensivos").

* + **CDA**: Proporciona información adicional sobre el perfil, mostrando la influencia de cada variable en las funciones discriminantes canónicas. Esto ayuda a entender cómo estas funciones separan los clústeres. Por ejemplo, una función discriminante principal podría separar los clústeres 2 y 3, siendo "atmct" la variable principal para esta separación. Otra función podría separar los clústeres 1 y 3, basándose en una combinación de "atmct", "atres" e "income".

1. Evaluación de los centroides con respecto a una clase externa (dada o derivada):

* Aquí, se compara la solución de clustering con una variable de clase preexistente o una clase creada a partir de los datos (por ejemplo, "inversores pequeños", "medianos", "grandes" creados a partir de la variable 'inversión' usando la función RANK).
* Se genera una matriz de confusión para cuantificar el grado de coincidencia entre las etiquetas de clase originales y las asignaciones de clúster. Un análisis de Chi-cuadrado y el coeficiente V de Cramer se utilizan para evaluar la significancia y la fuerza de la asociación entre la variable de clase y los clústeres derivados. Si el Chi-cuadrado es significativo y el V de Cramer es alto, sugiere una fuerte asociación.
* Interpretación de la matriz de confusión: Muestra la probabilidad de que un número de clúster represente una clase dada. Por ejemplo, si el clúster 1 representa principalmente al grupo de "pequeños inversores", pero también es el que tiene mayor probabilidad de representar al grupo de "medianos inversores", esto sugiere que puede haber problemas para separar bien estas dos clases con el clustering.

Es importante señalar que los resultados de MANOVA y ANOVA deben interpretarse con cautela, ya que el proceso de clustering viola algunas de las suposiciones clave subyacentes a estas pruebas. Sin embargo, combinadas, estas técnicas ofrecen una visión detallada de las características que definen a cada clúster.

## **Algoritmo t-sne**

t-SNE (t-distributed Stochastic Neighbor Embedding) es un método no lineal para visualizar datos de alta dimensión proyectándolos en un espacio de baja dimensión (típicamente 2D o 3D). A diferencia de PCA, que es una proyección lineal y busca preservar la varianza global, t-SNE se enfoca en preservar las relaciones locales entre los puntos, lo que le permite capturar estructuras no lineales complejas.

El algoritmo funciona en dos pasos principales:

1. Creación de una distribución de probabilidad en el espacio de alta dimensión: Para cada punto xi en el espacio original, t-SNE define la probabilidad de elegir otro punto xj como su vecino. Esta probabilidad es proporcional a la densidad de probabilidad de una distribución Gaussiana centrada en xj. La dispersión (σi) de cada Gaussiana se elige de manera que el "perplejidad" (un hiperparámetro que representa el número efectivo de vecinos) sea aproximadamente el mismo para todos los puntos, ajustando σi para que sea pequeño en regiones densas y grande en regiones dispersas. Los puntos cercanos tienen una alta probabilidad de ser vecinos, mientras que los puntos lejanos tienen una probabilidad que decae rápidamente pero nunca llega a cero.
2. Recreación de la distribución de probabilidad en un espacio de baja dimensión: t-SNE busca luego un mapeo de los puntos xi a puntos yi en un espacio de baja dimensión que reproduzca la distribución de probabilidad del paso 1 lo mejor posible. Para esto, utiliza la distribución t de Student con un grado de libertad (también conocida como distribución de Cauchy). La distribución t tiene "colas más largas" que una Gaussiana, lo que ayuda a evitar el problema del "apiñamiento" (crowding problem) en el espacio de baja dimensión. La optimización se realiza mediante descenso de gradiente en la divergencia KL (Kullback-Leibler) entre las distribuciones de probabilidad en alta y baja dimensión. El gradiente representa fuerzas de atracción/repulsión entre los puntos, que los llevan a asentarse en el espacio de baja dimensión.

Una característica importante es que t-SNE no define una función para reducir la dimensionalidad; optimiza directamente las incrustaciones (embeddings). Esto significa que un modelo entrenado con t-SNE no puede usarse para reducir la dimensionalidad de nuevos puntos de datos.

¿Para qué se utiliza?

t-SNE se utiliza principalmente para la visualización y exploración de datos complejos de alta dimensión, especialmente cuando se sospecha la existencia de estructuras no lineales que PCA no podría revelar. Es muy útil para:

* Identificar clústeres naturales o subgrupos en los datos.
* "Desenvolver" estructuras complejas o "manifold" que no son linealmente separables.
* Comprender la estructura global y local de los datos.

## **Ejemplo de implementación en Python:**

# Importar la librería

# from sklearn.manifold import TSNE

# import matplotlib.pyplot as plt

# import numpy as np

# Datos de ejemplo (sustituir con datos reales de alta dimensión)

# X = np.random.rand(100, 50) # 100 muestras, 50 dimensiones

# Inicializar t-SNE

# tsne = TSNE(n\_components=2, random\_state=42, perplexity=30) # n\_components para la dimensión de salida, perplexity es un hiperparámetro clave

# Reducir la dimensionalidad

# X\_2d = tsne.fit\_transform(X)

# Visualizar los resultados

# plt.figure(figsize=(8, 6))

# plt.scatter(X\_2d[:, 0], X\_2d[:, 1])

# plt.title('Visualización de t-SNE')

# plt.xlabel('Componente 1')

# plt.ylabel('Componente 2')

# plt.show()

Este ejemplo ilustra cómo, con muy pocos ajustes, t-SNE puede revelar estructuras subyacentes, como dos clústeres concéntricos, que PCA no podría separar.

## **Limitaciones**

* Optimización no convexa: La función de costo de t-SNE no es convexa, lo que significa que tiene múltiples mínimos locales. Esto hace que sea difícil garantizar la convergencia al óptimo global, y el algoritmo no es determinista; diferentes ejecuciones pueden producir resultados ligeramente diferentes. Se utilizan "trucos" como la "exageración temprana" para mitigar esto, pero no hay métodos infalibles.
* No es una proyección funcional: t-SNE no crea una función de mapeo. Por lo tanto, no se puede aplicar directamente a nuevas observaciones para reducir su dimensionalidad una vez que el modelo ha sido entrenado. Se optimizan directamente los puntos (embeddings).
* Sensibilidad a hiperparámetros: Aunque a menudo funciona bien con valores predeterminados, la calidad de la visualización puede ser sensible a hiperparámetros como la perplejidad y la exageración temprana. Valores extremos de perplejidad pueden dispersar los puntos sin estructura o agruparlos en un solo "blob". Una exageración temprana excesiva puede incluso crear "falsos clústeres".
* Costo computacional: Para conjuntos de datos grandes, la implementación exacta de t-SNE escala como O(n^2), lo que la hace prohibitiva. Existen aproximaciones (como el algoritmo Barnes-Hut) que escalan como O(n log n), pero aún puede ser lento para decenas de miles de puntos, especialmente si se ejecuta varias veces.
* Suposiciones sobre la estructura local: Aunque no es lineal, t-SNE asume que la estructura local de la variedad es lineal, ya que la distancia entre puntos vecinos se mide con la distancia euclidiana. Esto puede ser un problema en variedades intrínsecamente complejas.

## **Ventajas**

* Captura estructuras no lineales: Su principal ventaja sobre métodos como PCA es su capacidad para revelar y visualizar dependencias y agrupamientos no lineales en datos de alta dimensión.
* Manejo del problema de "apiñamiento" (crowding problem): Utiliza la distribución t de Student para "extender" los puntos de distancia media en el espacio de baja dimensión, evitando que todos los puntos se amontonen.
* Enfoque equilibrado entre estructura local y global: Al usar "vecinos estocásticos" (distribuciones de probabilidad entre vecinos), t-SNE considera tanto la estructura local como la global, sin ignorar por completo los puntos distantes.
* Efectividad en la práctica: A menudo produce visualizaciones intuitivas y significativas, siendo muy robusto y efectivo incluso con los hiperparámetros predeterminados en muchos conjuntos de datos.

Fuentes

Van der Maaten, L. J. P., & Hinton, G. E. (2008). Visualizing Data using t-SNE. Journal of Machine Learning Research, 9(Nov), 2579-2605.

Scikit-learn documentation: sklearn.manifold.TSNE.

Introduction to Clustering.pdf

Assessing Clustering Results.pdf:

# **Ejercicio 8 – Adaboost y Gradient Boosting**

***Que diferencia existen entre el algoritmo Adaboost y Gradiente Boosting. Explique brevemente. De una explicación intuitiva de cómo trabaja el Gradiente Boosting.***

## **Diferencias entre Adaboost y Gradient Boosting**

Tanto el Adaboost como el Gradient Boosting son algoritmos que pertenecen a la familia del boosting, en la cual se combinan modelos débiles de manera secuencial para poder obtener un predictor más preciso.

Por su lado, el Adaboost (cuyo nombre proviene de Adaptative Boosting) ajusta el entrenamiento mediante la reponderación de los ejemplos de entrenamiento. Es decir, en cada iteración los casos que son mal clasificados reciben un peso mayor, de forma que el próximo clasificador se concentre en estos. Esto resulta en un modelo compuesto por una combinación ponderada de los clasificadores débiles, donde cada uno tiene un peso proporcional al desempeño.

Por otro lado, el Gradient Boosting interpreta que el boosting es un problema de optimización en el espacio de funciones. Cada modelo nuevo se construye ara poder aproximar el descenso de gradiente de la función de pérdida, corrigiendo los residuos del modelo anterior. Es decir, en lugar de modificar los pesos de las observaciones como en el Adaboost, se añade de manera stagewise nuevas funciones que minimizar la pérdida elegida.

**Explicación de Gradient Boosting**

De acuerdo con Friedman (2001), el Gradient Boosting puede entenderse como un procedimiento de aproximación aditiva guiado por el gradiente:

1. Se comienza con un modelo simple, por ejemplo, una constante que minimiza la pérdida
2. Se calculan los residuos o pseudorespuestas, que representan la dirección del gradiente negativo de la función de pérdida
3. Se ajusta un nuevo modelo débil (como un árbol pequeño) a dichos residuos
4. El modelo global se actualiza sumando una fracción de este nuevo ajuste
5. El proceso se repite, y cada paso constituye un “boost” que aproxima cada vez más a la función objetivo

En términos más simples, el Gradient Boosting actúa como un proceso de corrección sucesiva de errores donde cada nuevo modelo corrige lo que los anteriores no lograron explicar

# **Ejercicio 9 – Random Forest**

***Random Forest. Describir el algoritmo y su diferencia con el resto de los algoritmos de ensamblaje. Describa los principales usos del algoritmo.***

## **Qué es Random Forest**

El Random Forest es un algoritmo de aprendizaje automático de uso común, registrado por Leo Breiman y Adele Cutler, que combina el resultado de múltiples árboles de decisión para llegar a un resultado único (citado de: ¿Qué es el bosque aleatorio?, IBM Think México).

Su funcionamiento se puede explicar de la siguiente manera. Se parte de un dataset original con N observaciones y se generan varios subconjuntos aleatorios de tamaño N mediante muestreo con reemplazo (bootstrap). Cada uno de estos subconjuntos se utiliza para entrenar un árbol distinto. Se hace sin poda y en cada Split (división), no se consideran todas las variables sino que se selecciona un subconjunto aleatorio de M características y se elige la mejor variable de ese subconjunto para hacer la partición.

Cada árbol da como resultado una predicción, en el caso de los Árboles de Clasificación, se vota por una clase y se toma la que sea mayoritaria, y en el de Árboles de regresión, se predice un valor numérico y se calcula el promedio de todas las predicciones.

A la hora de diferenciarlo con otros algoritmos de ensamblaje, se puede comparar primeramente con Bagging, el cual crea múltiples modelos base entrenados con subconjuntos bootstrap de los datos. Un bootstrap es una muestra de un conjunto de datos con reemplazo y cada muestra se genera muestreando uniformemente el conjunto de entrenamiento de tamaño m hasta obtener un nuevo conjunto con m instancias.

El muestreo bootstrap (bagging) consiste en entrenar un conjunto de árboles de decisión no podados sobre diferentes subconjuntos aleatorios de los datos de entrenamiento, obtenidos mediante muestreo con reemplazo. Este procedimiento reduce la varianza que es característica de los árboles de decisión y como combina sus predicciones, se logra un resultado más estable y preciso.

El algoritmo de Random Forest además, incorpora que en cada nodo de un árbol, en lugar de considerar todas las variables, se selecciona un subconjunto aleatorio de características. Esto genera mayor diversidad entre los árboles individuales y, al agregarlos, se obtiene un clasificador o regresor más robusto.

Por otro lado, se encuentra Adaboost, otro algoritmo de conjunto que se basa en la idea de combinar modelos más simples. Se diferencia de Random Forest porque el foco está en mejorar iterativamente el desempeño del modelo al asignar mayores pesos a los datos mal clasificados en cada iteración. En cada iteración, se entrena un nuevo modelo y se ajustan los pesos de los datos para dar más importancia a los casos difíciles.

Una de las principales ventajas de Adaboost es su capacidad para manejar conjuntos de datos desequilibrados, donde las clases varían en sus tamaños. A su vez, es bueno en la selección de características relevantes y puede reducir la varianza del modelo. No obstante, Adaboost puede ser sensible a datos atípicos y ruido en los datos, lo que puede afectar su rendimiento.

De esta manera, se podría decir que Random Forest tiende a funcionar mejor en conjuntos de datos grandes y complejos, donde hay una mayor variabilidad en las características. También es menos sensible a los datos atípicos y ruido en los datos. Por su parte, Adaboost es más conveniente para conjuntos de datos desequilibrados y puede lograr un mejor rendimiento en problemas de clasificación débil.

Otra comparación necesaria es con Gradient Boosting, que es una técnica que combina varios modelos débiles para crear un modelo fuerte. Los modelos se construyen secuencialmente, donde cada nuevo modelo intenta corregir los errores del modelo anterior.

La palabra "boosting" se refiere a la idea de mejorar la precisión combinando varios modelos simples. El término "gradient" hace referencia a que el algoritmo utiliza el gradiente descendente para minimizar los errores en cada etapa.

Gradient Boosting construye árboles secuencialmente, y cada nuevo árbol intenta corregir los errores del anterior optimizando el gradiente de una función de pérdida. Esta estrategia reduce el sesgo y mejora la precisión en tareas complejas, pero a costa de una mayor sensibilidad al ruido y del ajuste fino de hiperparámetros como la tasa de aprendizaje y la profundidad del árbol. En la práctica, Random Forest es más estable y eficiente sin un ajuste exhaustivo, mientras que el Gradient Boosting generalmente ofrece un mejor rendimiento predictivo con un ajuste adecuado.

En cuanto a los principales usos del algoritmo, se pueden nombrar múltiples industrias, como las Finanzas, reduce el tiempo dedicado a la gestión y al preprocesamiento de datos. También se utiliza para evaluar a clientes con alto riesgo crediticio, detectar fraudes y garantizar los problemas de fijación de precios de opciones.

También se utiliza en salud, ya que Random Forest, presente en aplicaciones de biología computacional, permite a los médicos resolver problemas como la clasificación de la expresión génica, el descubrimiento de biomarcadores y la anotación de secuencias. Además, los médicos pueden estimar los efectos de medicamentos específicos.

A su vez, se utiliza en el comercio electrónico a través de motores de recomendación para las últimas novedades en venta cruzada.

Random Forest también se utiliza ampliamente en problemas de aprendizaje automático por su versatilidad y robustez. Sus principales aplicaciones incluyen la clasificación supervisada, donde permite categorizar instancias con alta precisión (por ejemplo, para la detección de spam o diagnósticos médicos), y la regresión supervisada, donde se utiliza para predecir valores continuos como los precios de las viviendas o el consumo de energía. Además, es muy útil para la selección de variables importantes, ya que proporciona métricas para identificar las características que mejor explican el resultado, facilitando así el análisis exploratorio y la reducción de la dimensionalidad. Otro uso clave es el manejo de datos desequilibrados y valores faltantes, ya que es un modelo robusto que tolera escenarios comunes en datos reales e incluso puede usarse para la imputación de valores faltantes.

Fuentes:

Géron, A. (2019). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2ª ed.). O'Reilly Media.

Bishop, C. M. (2006). Pattern Recognition and Machine Learning. Springer.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2ª ed.). Springer.

IBM. (n.d.). What are Decision Trees? IBM Cloud. Recuperado, de https://www.ibm.com/cloud/learn/decision-trees

Scikit-learn. (n.d.). 1.10. Árboles de decisión., recuperado de https://scikit-learn.org/stable/modules/tree.html

Towards Data Science. (n.d.). Publicaciones sobre árboles de decisión. Medium. Recuperado de https://towardsdatascience.com/

# **Ejercicio 10 – Modelos en Python**

***Genere 2 dataset’s obtenidos de internet (una de clasificación y regresión) y genere un código en Python que compare los resultados en Regresión y clasificación para VSM, Random Forest , Gradiente Boosting y multiplayer perceptron, generando un ‘Pipeline’ completo desde el Ajuste de Hiperparámetros (tanto espacio de búsqueda de los parámetros y estrategia de búsqueda) como en la validación (Cross-validation, etc)***

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split, GridSearchCV

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import Pipeline

from sklearn.metrics import mean\_squared\_error, accuracy\_score, classification\_report

# Modelos

from sklearn.svm import SVR, SVC

from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier, GradientBoostingRegressor, GradientBoostingClassifier

from sklearn.neural\_network import MLPRegressor, MLPClassifier

# Dataset de Clasificación: Breast Cancer Wisconsin

from sklearn.datasets import load\_breast\_cancer

cancer = load\_breast\_cancer()

X\_class = pd.DataFrame(cancer.data, columns=cancer.feature\_names)

y\_class = cancer.target

# Dataset de Regresión: California Housing

from sklearn.datasets import fetch\_california\_housing

housing = fetch\_california\_housing()

X\_reg = pd.DataFrame(housing.data, columns=housing.feature\_names)

y\_reg = housing.target

# Separar datos

X\_class\_train, X\_class\_test, y\_class\_train, y\_class\_test = train\_test\_split(X\_class, y\_class, test\_size=0.3, random\_state=42)

X\_reg\_train, X\_reg\_test, y\_reg\_train, y\_reg\_test = train\_test\_split(X\_reg, y\_reg, test\_size=0.3, random\_state=42)

# Modelos y parámetros

models = {

'SVM\_reg': (SVR(), {'model\_\_kernel': ['linear', 'rbf'], 'model\_\_C': [0.1, 1, 10], 'model\_\_gamma': ['scale', 'auto']}),

'RandomForest\_reg': (RandomForestRegressor(), {'model\_\_n\_estimators': [100, 200], 'model\_\_max\_depth': [5, 10]}),

'GradientBoosting\_reg': (GradientBoostingRegressor(), {'model\_\_n\_estimators': [100, 200], 'model\_\_learning\_rate': [0.01, 0.1], 'model\_\_max\_depth': [3, 5]}),

'MLP\_reg': (MLPRegressor(max\_iter=1000), {'model\_\_hidden\_layer\_sizes': [(50,), (100,)], 'model\_\_activation': ['relu', 'tanh'], 'model\_\_alpha': [0.0001, 0.001]}),

'SVM\_class': (SVC(), {'model\_\_kernel': ['linear', 'rbf'], 'model\_\_C': [0.1, 1, 10], 'model\_\_gamma': ['scale', 'auto'], 'model\_\_probability': [True]}),

'RandomForest\_class': (RandomForestClassifier(), {'model\_\_n\_estimators': [100, 200], 'model\_\_max\_depth': [5, 10]}),

'GradientBoosting\_class': (GradientBoostingClassifier(), {'model\_\_n\_estimators': [100, 200], 'model\_\_learning\_rate': [0.01, 0.1], 'model\_\_max\_depth': [3, 5]}),

'MLP\_class': (MLPClassifier(max\_iter=1000), {'model\_\_hidden\_layer\_sizes': [(50,), (100,)], 'model\_\_activation': ['relu', 'tanh'], 'model\_\_alpha': [0.0001, 0.001]})

}

# Evaluación

def evaluate\_model(model\_tuple, X\_train, y\_train, X\_test, y\_test, is\_regression=False):

pipeline = Pipeline([

('scaler', StandardScaler()),

('model', model\_tuple[0])

])

grid = GridSearchCV(pipeline, model\_tuple[1], cv=5,

scoring='neg\_mean\_squared\_error' if is\_regression else 'accuracy',

n\_jobs=-1)

grid.fit(X\_train, y\_train)

print("Mejores parámetros:", grid.best\_params\_)

best\_model = grid.best\_estimator\_

y\_pred = best\_model.predict(X\_test)

if is\_regression:

mse = mean\_squared\_error(y\_test, y\_pred)

print("MSE:", round(mse, 3))

else:

acc = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", round(acc, 3))

print(classification\_report(y\_test, y\_pred))

# Comparación Regresión

print(" Evaluación en Regresión (California Housing):")

for name, model in models.items():

if 'reg' in name:

print(f"\nModelo: {name}")

evaluate\_model(model, X\_reg\_train, y\_reg\_train, X\_reg\_test, y\_reg\_test, is\_regression=True)

# Comparación Clasificación

print("\n Evaluación en Clasificación (Breast Cancer):")

for name, model in models.items():

if 'class' in name:

print(f"\nModelo: {name}")

evaluate\_model(model, X\_class\_train, y\_class\_train, X\_class\_test, y\_class\_test, is\_regression=False)

# **Ejercico 11 – Voted Peceptron**

Muestre el clasificador final del algoritmo voted perceptron y explique sus parámetros. Escriba el pseudocódigo del mencionado algoritmo. Elija un data set y corra VSM No lineal y Voted Perceptron con algún kernel que seleccione. Compare los resultados. s