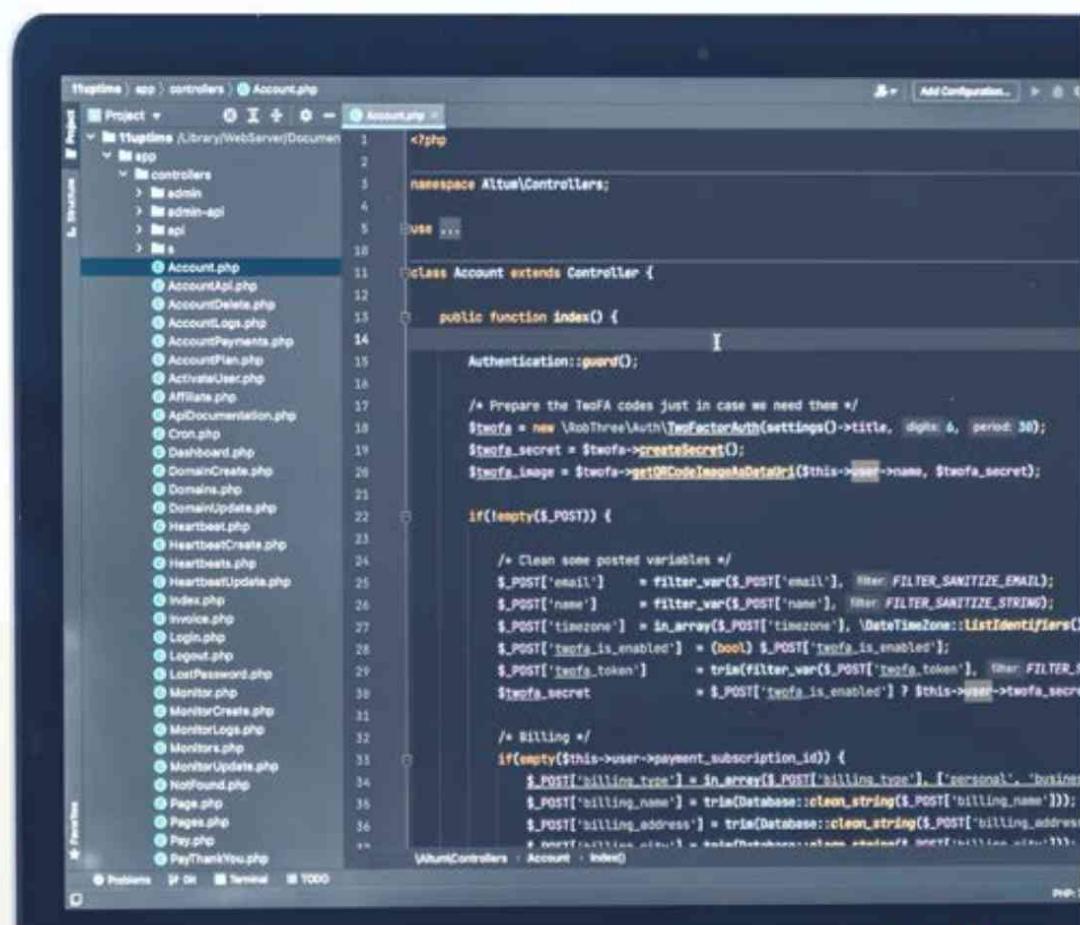


APRENDIZAJE AUTOMÁTICO Y PROFUNDO EN PYTHON

Una mirada hacia la inteligencia artificial



Carlos M. Pineda Pertuz

eduJ

<https://dogramcode.com/bloglibros>



Informática

APRENDIZAJE AUTOMÁTICO Y PROFUNDO EN PYTHON

Una mirada hacia la inteligencia artificial

Carlos M. Pineda Pertuz



Pineda Pertuz, Carlos M.

Aprendizaje automático y profundo en Python/ Carlos M. Pineda Pertuz
-- 1a. edición. Bogotá: Ediciones de la U, 2021
342 p.; 24 cm.
ISBN 978-958-792-316-2 e-ISBN 978-958-792-315-5
1. Informática 2. Programación 3. Aprendizaje automático 4. Python I. Tít.
621.39 cd 24 ed.

Área: Informática

Primera edición: Bogotá, Colombia, octubre de 2021

ISBN. 978-958-792-316-2

© Carlos M. Pineda Pertuz

Email: cmpinedasoft@gmail.com

© Ediciones de la U - Carrera 27 # 27-43 - Tel. (+57-1) 3203510 - 3203499

www.edicionesdelau.com - E-mail: editor@edicionesdelau.com

Bogotá, Colombia

Ediciones de la U es una empresa editorial que, con una visión moderna y estratégica de las tecnologías, desarrolla, promueve, distribuye y comercializa contenidos, herramientas de formación, libros técnicos y profesionales, e-books, e-learning o aprendizaje en línea, realizados por autores con amplia experiencia en las diferentes áreas profesionales e investigativas, para brindar a nuestros usuarios soluciones útiles y prácticas que contribuyan al dominio de sus campos de trabajo y a su mejor desempeño en un mundo global, cambiante y cada vez más competitivo.

Coordinación editorial: Adriana Gutiérrez M.

Carátula: Ediciones de la U

Impresión: DGP Editores SAS

Calle 63 No. 70 D - 34, Pbx. (571) 7217756

Impreso y hecho en Colombia

Printed and made in Colombia

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro y otros medios, sin el permiso previo y por escrito de los titulares del Copyright.

Agradecimientos

Por su cariño y apoyo incondicional:

A mis queridos padres Demetrio y Ruby
A mi esposa Katerine, a mi hija Luciana
y a mis hermanas: Ruby, Diana y Maira

<https://dogramcode.com/bloglibros>



Contenido

Prólogo.....	11
Introducción.....	13
CAPÍTULO 1. Conceptos básicos de programación en Python 3.9.....	15
1.1 Entorno de desarrollo y primeros pasos	16
1.2 Variables, Tipos de datos y operadores	20
1.3 Estructuras de datos: Tuplas, listas y diccionarios	24
1.4 Estructuras selectivas.....	27
1.5 Estructuras repetitivas.....	30
1.6 Funciones.....	32
1.7 Clases y objetos.....	33
CAPÍTULO 2. Introducción al Aprendizaje Automático	35
2.1 ¿Qué es aprendizaje automático?	35
2.2 Conceptos de aprendizaje automático	38
2.3 Tipos de aprendizaje automático	40
2.4 Problemas típicos en aprendizaje automático	43
2.5 Metodología CRISP-DM.....	45
CAPÍTULO 3. Herramientas para el aprendizaje automático	47
3.1 Manejo básico de datos con PANDAS.....	48
3.2 Manejo de arreglos con Numpy.....	53
3.2.1. Creación de arreglos.....	53
3.2.2. Acceso a elementos	55
3.2.3. Redimensionamiento.....	57
3.2.4. Operaciones matemáticas.....	58
3.3 Creando gráficos con Matplotlib	62
3.3.1 Gráficos de líneas.....	62
3.3.2 Gráficos de barras	64
3.3.3 Diagramas de dispersión	65
3.3.4. Histogramas	68
3.3.5. Diagrama de caja y bigotes	69
3.4 Breve Introducción a Scikit-Learn.....	70

CAPÍTULO 4. Preprocesado de datos 73

4.1 ¿Que es el preprocesado de datos?	73
4.2 Creación de conjunto de entrenamiento y pruebas	74
4.3 Manejo de datos ausentes	75
4.4 Manejo de datos categóricos.....	77
4.5 Escalamiento de características	80

CAPÍTULO 5. Modelos de regresión 83

5.1 Visualización de la relación entre características del conjunto de datos	85
5.2 Solución mediante el enfoque de mínimos cuadrados	89
5.3 Descenso del gradiente	92
5.4 Regresión lineal mediante scikit-learn	97
5.4.1. Regresión Simple	98
5.4.2 Regresión múltiple	88
5.5 Regresión con random sample consensus (RAMSAC)	100
5.6 Regresión lineal polinómica.....	102
5.7 Regresión lineal múltiple en notación matricial	105
5.8 Modelos no lineales.....	110
5.8.1 Funciones no lineales.....	110
5.8.2 Ejemplo suscripciones de telefonía	114

CAPÍTULO 6. Regularización, métricas de evaluación y ajuste de hiperparámetros 119

6.1. Regularización	120
6.1.1 Regresión Rígida	120
6.1.2 Regresión Lasso.....	122
6.1.3 Red elástica	122
6.2. Métricas y técnicas de validación de modelos de regresión	123
6.2.1 Error absoluto medio (MAE)	124
6.2.2 Error cuadrático medio (MSE)	124
6.2.3 Coeficiente de determinación (R^2)	124
6.2.4 Validación cruzada por k iteraciones	125
6.3. Curvas de aprendizaje y validación	127
6.4. Técnica de búsqueda de cuadrículas para el ajuste de hiperparámetros..	134

CAPÍTULO 7. Modelos de Clasificación I..... 137

7.1 Perceptrón simple	138
7.2 Neurona lineal adaptativa (ADALINE)	143
7.3 Regresión logística.....	148

CONTENIDO

7.3.1 Regresión logística con scikit-learn.....	154
7.3.2 Regresión logística con el descenso del gradiente estocástico.....	159
7.3.3 Regresión logística con regularización.....	162
7.4. Métricas de evaluación	164
7.4.1. Matriz de confusión	164
7.4.2. Exactitud (Accuracy)	165
7.4.3. Precisión (Precision).....	166
7.4.4. Recall, Sensibilidad o TPR (Tasa de verdadero positivo).....	166
7.4.5. F1	166
7.4.6. Tasa de falsos positivos.....	167
7.4.7. Curvas ROC (receiver operating characteristics)	168
7.5 Máquinas de vectores de soporte (SVM)	169
7.5.1. Clasificación Multiclas con SVM lineal.....	174
7.5.2. Kernels para separar datos no lineales.....	177
 CAPÍTULO 8. Modelos de Clasificación II.....	 183
8.1 Árboles de decisión	184
8.1.1. Métricas para medir la separación	187
8.1.2. Crear y visualizar árboles de decisión con Scikit-Learn	191
8.1.3. Identificación de características importantes	193
8.2 Bosques aleatorios (<i>Random Forest</i>)	196
8.3 Adaboost (<i>Adaptive boosting</i>)	198
8.4 <i>Gradient boosting</i>	200
8.5 <i>Naive bayes</i>	202
8.6 K Vecinos mas cercanos (KNN).....	206
8.7 Sistemas de recomendación	210
8.7.1. Sistemas de recomendación basados en contenido	210
8.7.2. Sistema de recomendación basado en filtro colaborativo	220
8.8 Entrenamiento mediante aprendizaje en línea.....	221
 CAPÍTULO 9. <i>Clustering</i>	 223
9.1 K Medias	223
9.2 <i>Clustering jerárquico</i>	231
9.3 Dbscan (Density Based Spatial Clustering of Applications with Noise).....	233
 CAPÍTULO 10. Reducción de la dimensionalidad	 239
10.1 Análisis de componentes principales (PCA).....	240
10.2 Análisis discriminante lineal (ADL)	250
 CAPÍTULO 11. Introducción a las redes neuronales	 253

11.1 Conceptos básicos sobre redes neuronales	254
11.1.1. Neurona artificial	254
11.1.2. Red neuronal	255
11.1.3. Pesos.....	256
11.1.4. Sesgos (Bias)	256
11.1.5. Funciones de activación determina la salida de la neurona.....	257
11.2 Entrenamiento de una red neuronal	261
11.3 Red neuronal para clasificación binaria	271
11.4 Red neuronal para clasificación múltiple	276
CAPÍTULO 12. Redes neuronales convolucionales.....	283
12.1 Introducción a las redes neuronales convolucionales	283
12.1.1. Convolución	285
12.1.2. Agrupación	288
12.2 CNNs en Keras	289
12.3 Regularización y dropout.....	295
CAPÍTULO 13. Aumento de datos y transferencia de aprendizaje	299
13.1 Generador de datos de imágenes	299
13.2 Aumento de datos (<i>data augmentation</i>)	305
13.3 Transferencia de aprendizaje (<i>transfer learning</i>)	308
13.3.1. Extracción de características	310
13.3.2. Ajuste fino (<i>Fine tuning</i>)	312
CAPÍTULO 14. Introducción al Procesamiento del Lenguaje Natural (PNL) .	315
14.1 Palabras embebidas (<i>Word embedding</i>).....	315
14.2 Introducción a Word2Vec.....	316
14.3 Word2vec con librería Gensim	318
CAPÍTULO 15. Redes neuronales recurrentes (RNN)	323
15.1 Introducción a las redes neuronales recurrentes	323
15.2 Propagación a través del tiempo (BPTT)	326
15.3 LSTM (Memoria larga a corto plazo)	327
15.3.1. Ejemplo sobre análisis de sentimiento	330
15.3.2. Ejemplo sobre generación de texto	332
Referencias bibliográficas.....	337
Índice analítico	339

Prólogo

He decidido crear esta obra como una guía para que estudiantes de ingeniería o cualquier persona interesada en aprender sobre aprendizaje automático y profundo pueda adquirir las bases necesarias sobre este tema tan apasionante. Será un camino lleno de retos en donde exploraremos las técnicas y algoritmos más representativos, lo cual le permitirá al lector forjar conocimientos sólidos en la materia y ser capaz de crear sus propios modelos de aprendizaje para resolver una amplia gama de problemas presentes en el mundo real.

Este libro comenzará dando una introducción al lenguaje Python, siendo esta la herramienta de programación escogida para desarrollar los ejemplos, luego se irán explicando los conceptos teóricos y consecuentemente se abordará de manera práctica el funcionamiento de los diferentes métodos y técnicas usadas en el campo del aprendizaje automático. El lenguaje de programación Python es de los que más ha ganado fuerza en este ámbito, fundamentalmente por su facilidad y por la gran cantidad de librerías que pone a disposición de los desarrolladores y de quienes trabajan en el mundo de la ciencia de datos.

Aprendizaje automático y profundo en Python le será un libro muy ameno con suficientes ejemplos y ejercicios para poner en práctica, y que le permitirá reforzar su aprendizaje mediante el estudio de código fuente que el autor pondrá a su alcance mediante cuadernos de Jupyter Notebook. Además, cada tema será explicado de manera sencilla y exemplificada de manera que el lector pueda ir probando cada aspecto teórico con las herramientas y el software sugeridos por el autor. Esto y más hacen que esta obra se diferencie de otras de su tipo y sea suficientemente útil para toda aquella persona que quiera sumergirse en el mundo del aprendizaje automático de una manera organizada y fácil, pero sin dejar a un lado el fundamento teórico que está detrás de los métodos usados en esta gran disciplina.

Como complemento web al libro en www.edicionesdelau.com encontrará la carpeta Fuentes que contiene el desarrollo de ejemplos y ejercicios, para su mejor comprensión.



<https://dogramcode.com/bloglibros>



Introducción

Esta obra pretende ser una herramienta de apoyo y de consulta para toda aquella persona interesada en dominar los fundamentos del aprendizaje automático y profundo, a tal punto que le permita aprender lo necesario para desarrollar sus propios modelos de aprendizaje aptos para realizar predicciones con base en los datos, para ello el autor hará uso en la mayoría de los casos de explicaciones teóricas y prácticas, que permitan al lector afianzar sus ideas y fortalecer su aprendizaje. El libro arranca exponiendo algunos temas básicos y, poco a poco, irá apor-tando otros temas un poco más complejos necesarios para forjar un conocimiento integral y mucho más sólido sobre aprendizaje automático con la convicción de dar al lector la orientación necesaria para que sea capaz de abordar sus propios proyectos basados en las técnicas descritas.

Para completar cabalmente la lectura de este libro se requieren de unos conocimientos mínimos en cálculo, estadística y de programación, debido a que cada tema, generalmente, se aborda inicialmente con una explicación de diversos conceptos, tomando para ello aspectos del cálculo, la estadística y el álgebra lineal. Posteriormente, los algoritmos se implementarán en su mayoría desde cero y en otros con el apoyo de varias librerías o APIs para facilitar su desarrollo dentro del entorno de programación.

Aprendizaje automático y profundo tiene una estructura en capítulos que inicia con explicaciones sobre el lenguaje Python, para luego abarcar los algoritmos más destacados dentro del aprendizaje de máquina. El libro se encuentra dividido en dos partes: la primera enfocada en el *machine learning* y sus diferentes algoritmos de regresión y clasificación, *clustering*, entre otros. La segunda parte comprende varias técnicas de *deep learning* donde estudiaremos diferentes arquitecturas de redes neuronales como: redes densamente conectadas, redes convolucionales y redes recurrentes.

Espero que esta obra sea del agrado y el disfrute del lector, así como lo fue para quien escribió estas palabras, que sin duda les puede decir que los temas de este libro son de los que más pasión le han conferido a lo largo de su carrera. Bienvenidos a este apasionante mundo.

<https://dogramcode.com/bloglibros>



CAPÍTULO 1

Conceptos básicos de programación en Python 3.9

Temas

- 1.1 Entorno de desarrollo y primeros pasos
- 1.2 Variables, tipos de datos y operadores
- 1.3 Estructuras de datos: Tuplas, listas y diccionarios
- 1.4 Estructuras selectivas
- 1.5 Estructuras repetitivas
- 1.6 Funciones
- 1.7 Clases y objetos

En este capítulo se abordarán algunos fundamentos de la programación en Python que resultarán de utilidad para que aquellos lectores que nunca han manejado este lenguaje puedan entender más fácilmente los algoritmos que se ofrecen en el libro en los posteriores capítulos.

Python es un lenguaje de programación de código abierto e interpretado, esto último significa que cada línea de código que escribamos es leída por un intérprete que las va ejecutando. Esta condición le otorga más rapidez en la ejecución de los programas, puesto que estos no tienen que compilarse previamente ahorrándose el tiempo de compilación. Además, Python es un lenguaje multiplataforma, lo cual hace que cualquier programa que hagamos pueda correr sin ningún problema en diferentes plataformas o sistemas operativos. Así mismo este software de desarrollo dispone de una sintaxis muy simple lo que hace que su curva de aprendizaje no sea tan pronunciada en comparación con sus similares como Java. Y por si fuera poco cuenta con una basta cantidad de librerías actualizadas permanentemente por su gran comunidad de desarrolladores, las cuales permiten programar diver-

sas funcionalidades con un mínimo esfuerzo. Estas características y más hacen de Python el lenguaje de programación preferido por muchos programadores dedicados al desarrollo de sistemas basados en aprendizaje automático.

1.1 Entorno de desarrollo y primeros pasos

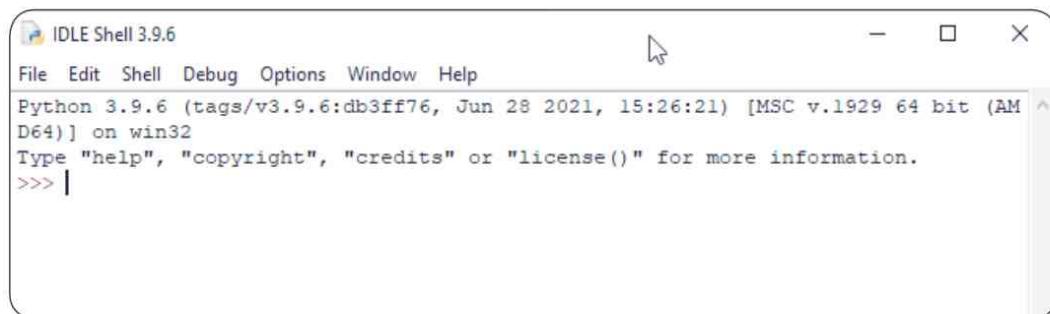
Para comenzar a programar en Python se requiere descargar el instalador de la página oficial en internet: <https://www.python.org/>

Dentro de esta página se puede escoger para qué sistema operativo lo vamos a instalar y la versión del lenguaje Python. Para esto último se puede optar por la última versión, que al momento de escribir este libro es la 3.9.6.

La instalación es como la de cualquier otro software, donde a menos que se desee establecer una configuración personalizada se resumiría solo a pulsar el botón "Siguiente" para avanzar a través de las ventanas del asistente de instalación.

Una vez instalado, ya podemos probar la herramienta y escribir nuestros primeros programas, para ello necesitamos un editor donde coloquemos las líneas de código, en ese sentido Python pone en nuestras manos un programa llamado IDLE que viene junto con la instalación realizada en el punto anterior, y aunque no es el más sofisticado para desarrollar aplicaciones, si puede ser de utilidad para realizar algunas pruebas o ejemplos de poca envergadura.

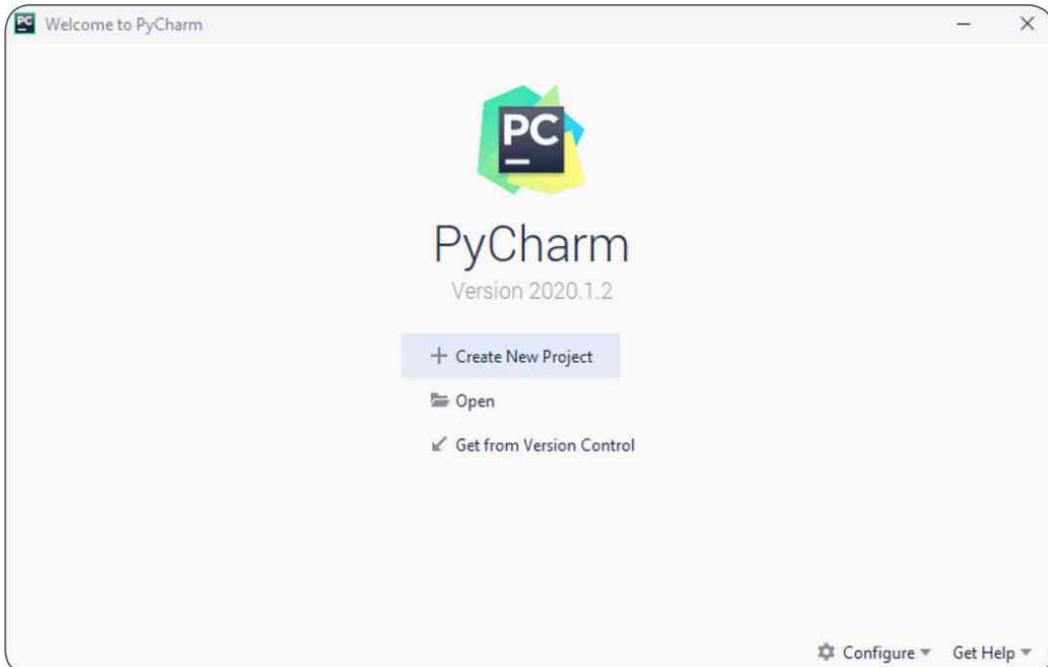
Al abrir IDLE observamos una interfaz como se muestra en la siguiente imagen:



Sin embargo, es de anotar que existen IDEs o entornos de desarrollo integrados: como PyCharm, VIM, Wing, entre otros que ofrecen un entorno más completo y mucho más adecuado para el desarrollo de aplicaciones. En nuestro caso usaremos PyCharm por ser a consideración del autor muy liviano, intuitivo y con una gran cantidad de opciones en su versión community (open source). Este software es desarrollado por la empresa Jetbrains y lo podemos encontrar en la siguiente dirección de internet: <https://www.jetbrains.com/es-es/pycharm/>

CAP. 1 - CONCEPTOS BÁSICOS DE PROGRAMACIÓN EN PYTHON 3.9

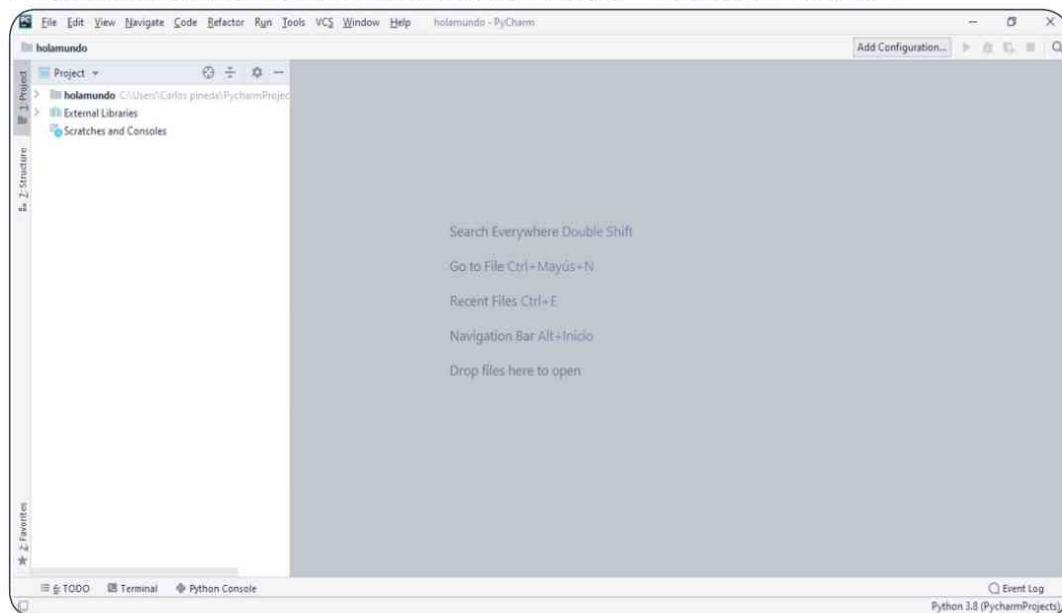
La siguiente figura muestra la interfaz principal de PyCharm, donde básicamente nos indica que podemos crear un nuevo proyecto (Create New Project), abrir uno existente (Open) u obtener un proyecto de un repositorio de control de versiones (Get from Version Control).



Al escoger la opción “Create New Project” seguidamente especificamos un nombre para el proyecto, por ejemplo, llamémosle “holamundo” y una ruta para el mismo, pero podemos dejar esta última y las demás opciones por defecto. Tenga presente el lector que solo haremos una breve introducción a la herramienta, así que se omitirán los detalles referentes a las diferentes configuraciones.

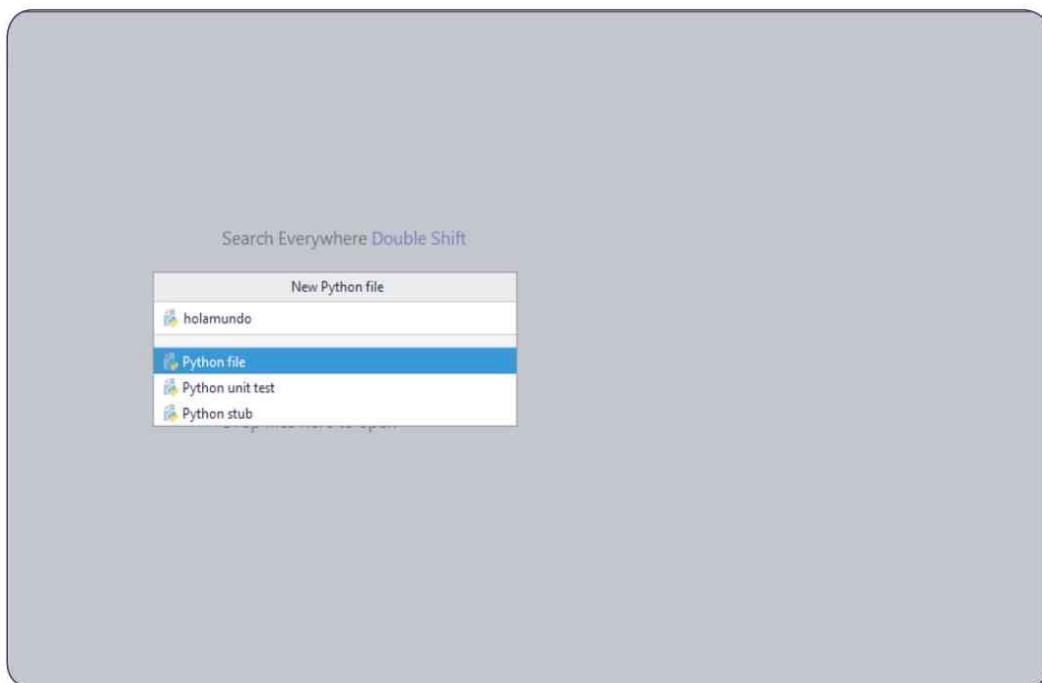
En la siguiente imagen vemos que se creó el proyecto llamado “HolaMundo”.

APRENDIZAJE AUTOMÁTICO Y PROFUNDO EN PYTHON - CARLOS M. PINEDA P.



Ahora dentro del proyecto creado procedemos a adicionar un primer programa de prueba. Para hacer esto, hacemos clic derecho sobre la carpeta “holamundo” y en el menú contextual que aparece escogemos la opción new Python File, mediante la cual creamos un archivo cuyo nombre en nuestro caso será nuevamente “holamundo”, sin necesidad de preocuparnos por la extensión ya que el IDE le agrega la extensión .py automáticamente, siendo esta la utilizada para los scripts o programas escritos en Python.

Luego presionamos la tecla ENTER y finalmente se habrá creado el archivo deseado.



CAP. 1 - CONCEPTOS BÁSICOS DE PROGRAMACIÓN EN PYTHON 3.9

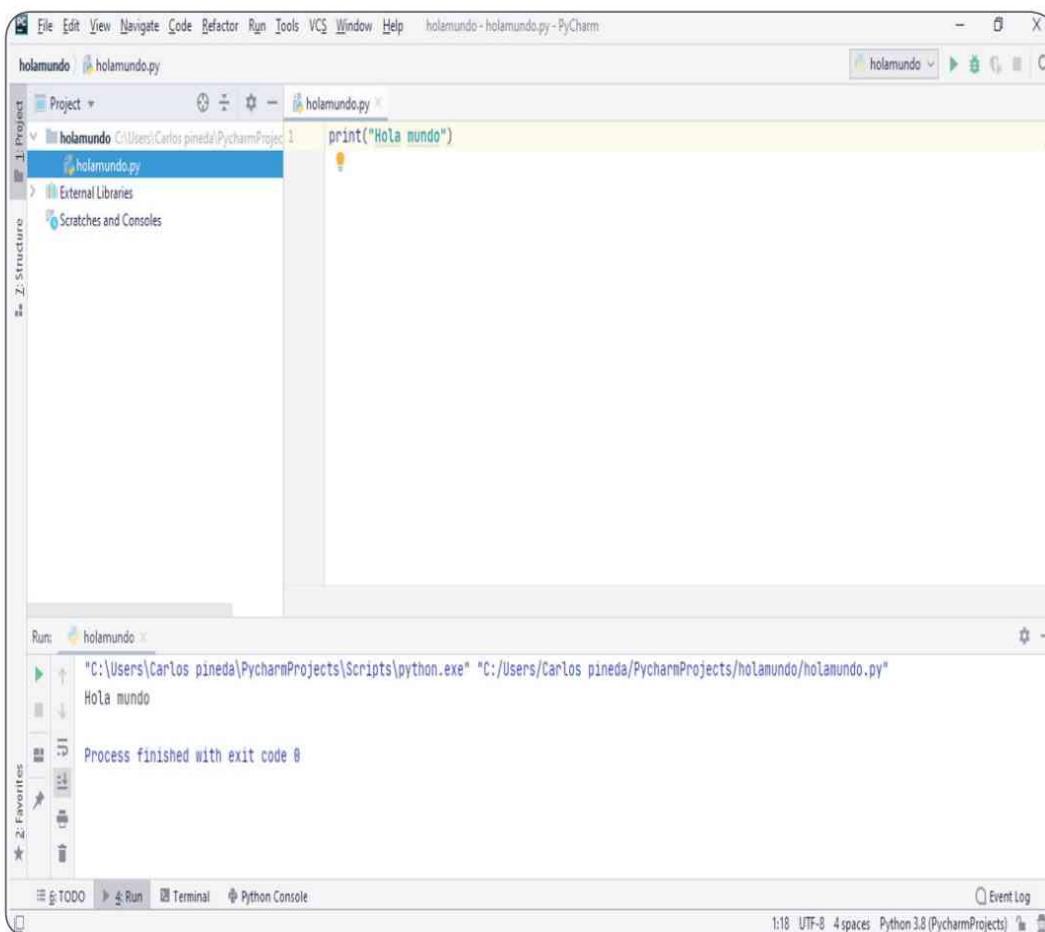
Una vez realizado los pasos anteriores y teniendo desplegado nuestro archivo "holamundo.py" en el editor de código de PyCharm, escribimos la siguiente línea:

```
print("Hola mundo")
```

Esta simple instrucción imprime un mensaje en la consola, en ella actúa la función print que se encarga de mostrar en la salida estándar lo que especifiquemos dentro de los paréntesis, siendo en esta oportunidad la cadena "Hola Mundo" lo que se mostraría.

Para ejecutar el programa simplemente nos dirigimos al menú run y de allí seleccionamos la opción run. Si es la primera vez que ejecutamos nuestro programa se nos mostrará una ventana donde nos pide seleccionar el archivo (holamundo.py). Si no hay errores de sintaxis la ejecución nos mostrará el texto "Hola mundo" y un mensaje al final, que indica que el proceso finalizó con éxito:

Process finished with exit code 0



Hasta aquí hemos creado nuestro primer “hola mundo” en Python y antes de entrar a revisar otros elementos de la programación en este lenguaje es necesario mencionar la importancia que para este significa la indentación. La indentación consiste básicamente en aplicar sangrías a las instrucciones según el nivel que tengan dentro de un bloque de código. Las declaraciones que requieren un primer nivel de sangría terminan con dos puntos como en el caso de las definiciones de clase, funciones o sentencias control de flujo y las que están dentro del bloque se les aumenta la sangría y se colocan alineadas. Por ejemplo, el fragmento de código siguiente tiene un condicional *if* en un primer nivel y más abajo aparecen dos sentencias con la función *print()* con sangría o desplazadas hacia la derecha demostrando que están en un nivel inferior y por tanto dependen de dicho condicional.

```
if a > 0:  
    print("El número es positivo")  
    print("Se trata del número ", a)
```

Por supuesto dentro de un script más grande puede haber muchos más niveles o mucha mayor anidación, pero las sentencias que hacen parte de un mismo nivel como en el ejemplo mostrado deben estar alineadas, si esto no se tiene en cuenta se corre el riesgo de recibir el error:

IndentationError: unindent does not match any outer indentation level

1.2 Variables, Tipos de datos y operadores

Variables

Una variable es un espacio en memoria destinado para el almacenamiento de un valor el cual va a ser usado en un programa. Siendo posible modificar el dato que en algún momento almacena por otro en cualquier otra parte del programa, de esta característica se deriva el nombre de variable.

En cuanto a la declaración de variables, en Python no es necesario especificar los tipos de datos ya que estos son definidos y asumidos por el lenguaje a partir del valor que se le asigne a la variable. Por esta razón, decimos que es un lenguaje de tipado dinámico. Este modo de tratar los tipos de datos permite que a una variable *n* que se le asigne por ejemplo el 10, siendo en ese momento de tipo *int* (entero), se le pueda posteriormente asignar una cadena como “Hola” pasando a ser de tipo *string* (cadena de caracteres).

Tipos de datos

Los tipos de datos manejados en Python son básicamente: números (enteros, reales y complejos), strings(cadenas) y booleanos. Los enteros o *int* manejan un rango infinito de números incluyendo los números positivos, negativos y el cero. Ejemplos: -5, 0, 10, por su parte los reales o *float* también manejan un rango infinito de números y son aquellos que tienen decimales. Ejemplos: -5.0, 3.5, 0.9.

Algunas sentencias de declaración de variables con todos estos tipos de datos serían:

```
n1 = -5
n2 = 0
n3 = 10
n4 = -5
n5 = 3.5
n6 = 0.9
n1 = 0.5
```

Note el lector que a la variable *n1* que inicialmente tenía el valor -5, en la última instrucción se le ha asignado el número 0.5, lo cual permite cambiar el tipo de dato de la variable de entero (*int*) a real (*float*), lo cual reafirma el concepto de tipado dinámico explicado anteriormente.

Después de ejecutar el código anterior el lector puede emplear la función *type(variable)* para conocer el tipo de dato de la variable pasada como parámetro:

```
print(type(n1))
```

En cuyo caso el resultado obtenido será <class 'float'>, lo cual quiere decir que el tipo de dato de la variable *n1* es número de coma flotante.

Por otro lado, existe otro tipo de dato conocido como *boolean*, que solo puede tomar dos valores posibles: True (verdadero) o False (falso).

```
es_adulto = False
esta_trabajando = True
```

Y por último se encuentran los *str* o cadenas de caracteres, que son conjuntos de caracteres, es decir, combinaciones de números, letras y caracteres especiales, como por ejemplo la dirección de una residencia:

```
direccion = "cll 27a # 9i 10"
```

Como consideración importante, se deben colocar comillas simples o dobles cuando se trabaja con cadenas de caracteres. De igual manera es importante resaltar la concatenación, operación que consiste en unir dos o más cadenas por medio del operador (+). En este nuevo ejemplo, unimos la palabra "Hola", más una cadena con un espacio (" ") y la cadena "Mundo":

```
cad1 = "Hola"  
cad2 = "Mundo"  
cad3 = cad1 + " " + cad2  
print(cad3)
```

Otro elemento importante en programación son los comentarios, usados principalmente para hacer explicaciones en el código lo cual facilita enormemente las labores de mantenimiento de los programas, permitiendo que otras personas pueden entender lo que hace cada instrucción con una mayor facilidad. En Python los comentarios de una línea se crean anteponiendo el símbolo # antes del escrito, mientras que en los de varias líneas se colocan tres comillas dobles al principio y al final del texto. Miremos como se pueden aplicar con un ejemplo:

```
#Esto es un comentario de una linea  
""" Esto es un comentario de  
    varias lineas  
"""
```

Operadores

Los operadores son un conjunto de símbolos usados para realizar operaciones sobre números, variables, etc. La ejecución de dichas operaciones trae consigo la obtención de un valor determinado.

CAP. 1 - CONCEPTOS BÁSICOS DE PROGRAMACIÓN EN PYTHON 3.9

En Python se usa el operador igual (`=`) para realizar asignaciones, esto es, colocar datos o expresiones dentro de variables. Además, entre los operadores más usados encontramos principalmente: aritméticos, relacionales y lógicos.

Operadores aritméticos

Son aquellos que operan sobre números, llevando a cabo operaciones aritméticas básicas y devuelven un valor numérico. El resultado exacto depende de los tipos de operandos involucrados.

Operador	Descripción	Ejemplo (a = 5)
<code>+</code>	Suma	<code>b = a + 5</code>
<code>-</code>	Resta	<code>b = a - 5</code>
<code>*</code>	Multiplicación	<code>b = a * 2</code>
<code>/</code>	División real	<code>c = b / a</code>
<code>//</code>	División entera	<code>c = b // a</code>
<code>%</code>	Residuo	<code>c = b % 2</code>
<code>**</code>	Exponenciación	<code>c = b**2</code>

Operadores relationales

Realizan comparaciones entre datos que sean de tipo numérico, carácter y boolean, devolviendo siempre un resultado booleano.

Operador	Descripción
<code>==</code>	Igual que
<code>!=</code>	Diferente que
<code><</code>	Menor que
<code>></code>	Mayor que
<code><=</code>	Menor o igual que
<code>>=</code>	Mayor o igual que

Operadores lógicos

Permiten conectar expresiones a las cuales se les aplican las tablas de verdad para obtener un valor booleano (True o False).

Operador	Descripción
<code>not</code>	Negación
<code>and (y)</code>	Y lógico
<code>or (o)</code>	O lógico

Un aspecto a considerar cuando se está programando es que las expresiones son evaluadas de la misma manera como se evalúan las expresiones aritméticas. Esto quiere decir, que se pueden utilizar las mismas reglas matemáticas para evaluar una expresión en Python. Las expresiones se evaluarán de izquierda a derecha teniendo en cuenta la prioridad de los operadores. La siguiente tabla muestra el orden de evaluación de algunos operadores, de mayor a menor nivel de prioridad.

Operador	Descripción
()	Paréntesis
**	Exponenciación
*	Multiplicación
/	División
Suma	+
Resta	-

1.3 Estructuras de datos: Tuplas, listas y diccionarios

Tuplas

Son estructuras que permiten almacenar diferentes tipos de datos, además se consideran inmutables, es decir, no permiten modificar o agregar elementos una vez se haya definido la tupla. Las tuplas usan paréntesis para especificar los datos. Ejemplo:

```
tupla = (1, "holá", 3.5)
# mostramos todos los datos de la tupla
print(tupla)
# mostramos el primer elemento
print(tupla[0])
# Esta operación no está permitida
tupla[0] = 2
```

Del anterior ejemplo, se tiene que, para mostrar un elemento en concreto de la tupla se utilizan corchetes y dentro de estos un índice que denota la posición del dato. Los índices en Python comienzan desde 0. Por ejemplo, si deseamos mostrar el primer elemento el índice es 0, si deseamos mostrar el segundo elemento el índice sería 1 y así sucesivamente. Además, observe como la última asignación genera error por lo ya comentado acerca de la inmutabilidad de las tuplas.

Listas

Así como las tuplas, las listas se usan para almacenar diferentes tipos de datos, pero con la diferencia que, a estas, si se les pueden modificar sus elementos o agregar elementos nuevos. Se definen con corchetes y al igual que con las tuplas los elementos se acceden mediante un índice entero que inicia desde cero. Veamos un ejemplo donde creamos dos listas y mostramos su contenido:

```
# creamos una lista vacía
lista1 = []
# adicionamos un elemento tipo String
lista1.append("Aprendizaje automático")
print(lista1)
# Creamos otra lista con algunos elementos
lista2 = ["Python", "Java", "C++"]
# mostramos el primer elemento (Python)
print(lista2[0])
```

Observemos en el código el uso de la función `append()` para adicionar un elemento nuevo a la lista `list1`, también se puede ver como para acceder a un valor de una lista se usa un índice entero dentro de corchetes. Sin embargo, Python ofrece una manera fácil y a la vez muy útil de acceder a varios elementos de una lista, mediante el operador `(:)`, observemos cómo los podemos usar:

```
edades = [18, 22, 33, 15, 25, 26, 35, 40, 13, 20]
# muestra todas las edades
print("Todas:", edades[:])
# Muestra de la 1 hasta el último
print("1-ultimo:", edades[1:])
# Muestra de la 1 a la 3 sin incluir la 3
print("1-3:", edades[1:3])
# Muestra del 0 hasta el penúltimo
print("0-penúltimo:", edades[0:-1])
```

Por otro lado, las listas cuentan con una serie de funciones útiles que el lector debe conocer, como, por ejemplo:

- `len(lista)`: Permite obtener la longitud de una lista pasada como parámetro.
- `append(valor)`: Adiciona un nuevo valor a la lista.
- `insert(posición, valor)`: Permite agregar el elemento `valor` en la posición `posición`.

- `del(elemento)`: Permite eliminar un elemento de la lista pasado como parámetro.
- `remove(elemento)`: Al igual que la anterior función, elimina de la lista el elemento pasado como parámetro.
- `sort([reverse=True])`: Permite ordenar una lista en orden ascendente, este es el comportamiento por defecto, si se indica el parámetro `reverse=True` el ordenamiento se realizará descendente.

Probemos el siguiente ejemplo para observar el trabajo de las anteriores funciones:

```
# mostramos la longitud de lista2
print("Longitud:", len(lista2))
# Añadimos un elemento en la posición 3
lista2.insert(3, "PHP")
print(lista2)
# Eliminamos el elemento C++
del(lista2[2])
# Eliminamos el elemento PHP
lista2.remove("PHP")
print(lista2)
# Ordenamos la lista en orden ascendente
lista2.sort()
print(lista2)
```

```
Longitud: 3
['Python', 'Java', 'C++', 'PHP']
['Python', 'Java']
['Java', 'Python']
```

Diccionarios

Son una colección de datos mutable muy similar a las listas en cuanto a su función de almacenamiento, pero con una diferencia en el modo de acceso a los elementos, realizado en este caso por medio del uso de una clave (key) la cual debe expresarse entre corchetes y puede ser de tipo entero o cadena. A los diccionarios también se les conoce como arrays asociativos y se construyen empleando conjuntos de pares clave-valor.

```

registro = {"ID": 12345 , "Nombre": "Carlos", "Apellido": "Pineda"}
# Imprimimos el diccionario
print(registro)
# Accedemos al valor "Carlos" a través de la clave "Nombre"
print(registro["Nombre"])
# Agregamos un nuevo elemento
registro["Profesion"] = "Profesor"
print(registro)

```

```

{'ID': 12345, 'Nombre': 'Carlos', 'Apellido': 'Pineda'}
Carlos
{'ID': 12345, 'Nombre': 'Carlos', 'Apellido': 'Pineda', 'Profesion': 'Profesor'}

```

El anterior diccionario contiene tres claves “ID”, “Nombre” y “Apellido” y tres valores 12345, “Carlos” y “Pineda”. El acceso a un valor se hace escribiendo dentro del corchete la clave asociada al mismo.

1.4 Estructuras selectivas

Las estructuras de selección como el nombre lo indica permite escoger o seleccionar que parte del código de un programa deberá ejecutarse. Esto se logra a partir de la evaluación de una condición. Esta última es una expresión de tipo lógica que al evaluarse tomará como posibles valores **True** o **False**. Por esto también se conocen como estructuras de control de flujo, porque permiten controlar o modificar el flujo (secuencia) de ejecución de un programa. En Python como en otros lenguajes se admiten básicamente tres tipos de estructuras selectivas: Simple, doble y múltiple.

Simple

Esta sentencia permite ejecutar una acción si la condición es verdadera o también podríamos decir que evita la acción si la condición es falsa. La sintaxis es como sigue:

```

if(expresión_condicional):
    sentencias

```

En la anterior definición notamos que la sentencia if debe ser escrita siempre en minúsculas, seguida de una *expresión_condicional* que es una condición evaluada de forma lógica y va seguida del operador (:). Si esta condición se evalúa como *True* se ejecutan las sentencias localizadas posteriormente. En el ejemplo siguiente se

cumple la condición $num > 0$, dando con ello paso a la sentencia de la función `print()`:

```
num = 5
if(num > 0):
    print("El número ", num , "Es positivo")
```

El número 5 es positivo

Tenga presente que la mencionada condición no se restringe a una sola subexpresión, es posible definir varias dependiendo de lo que se quiera verificar o controlar, pero eso sí cumpliendo con el requisito que cada una deba ser evaluada lógicamente. Consideremos el siguiente pseudocódigo de dos estructuras *if* con expresiones A y B conectadas con los operadores lógicos *and* y *or*.

```
if (A and B){
    sentencias
}
if (A or B){
    sentencias
}
```

En situaciones como estas la evaluación se llevará a cabo teniendo en cuenta las tablas de verdad. Observe como en la siguiente tabla se evalúan las expresiones booleanas: A y B.

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Doble

Realiza una acción si la condición es verdadera, o realiza otra acción distinta si la condición es falsa. La sintaxis es como sigue:

```
if(expresión_condicional):
    sentencias1
else:
    sentencias2
```

Ampliemos el ejemplo anterior ahora con una estructura selectiva doble:

```
num = 5
if(num > 0):
    print("El número ", num , "Es positivo")
else:
    print("El número ", num , "Es negativo")
```

El número 5 es positivo

Múltiple

Esta estructura consiste en definir varias opciones o alternativas de las cuales solo una será ejecutada. O dicho de otra manera permite establecer diferentes caminos por donde puede dirigirse el flujo del programa, pero que solo se puede tomar uno en un momento determinado. La sintaxis sería la siguiente:

```
if(expresión_condicional):
    sentencias
elif:
    sentencias
...
else:
    sentencias
```

Miremos un ejemplo de estructura de selección múltiple donde a partir de una nota numérica mostraremos su representación cualitativa equivalente:

```
nota = 3
if (nota > 0 and nota <= 1):
    print("Nota Insuficiente")
elif (nota > 1 and nota <= 2):
    print("Nota Regular")
elif (nota > 2 and nota <= 3):
    print("Nota Buena")
elif (nota > 3 and nota <= 4):
    print("Nota Sobresaliente")
elif (nota > 4 and nota <=5 ):
    print("Nota Excelente")
else:
    print("Nota no valida")
```

Como se aprecia cada una de las condiciones es verificada, si una de ellas se cumple se ejecuta las sentencias de ese bloque, o sea aquellas asociadas a esa condición, si ninguna de las condiciones tras su comprobación resulta verdadera se ejecutará lo establecido en el *else*, que bien podría considerarse la opción por defecto.

1.5 Estructuras repetitivas

Las estructuras repetitivas son usadas para ordenarle a un programa repetir un conjunto de sentencias un número determinado de veces.

Suponga que necesita repetir el mensaje “Bienvenido a Python” 100 veces. Puede por supuesto resultar tedioso escribir dichas sentencias en 100 ocasiones una detrás de la otra, y más aún si son 1.000 o 1.000.000 las veces que debe hacerse dicha repetición. Para evitar tener que hacer este trabajo desgastante, en programación existe lo que se conoce como estructuras repetitivas, ciclos o bucles, las cuales nos facilitan las cosas en este sentido.

Las sentencias provistas por Python para apoyarnos en esta labor son: *for* y *while*. Pero, además, es bastante común cuando se trabaja con ciclos emplear dos tipos especiales de variables: **contadores** y **acumuladores**. Un contador es una variable cuyo valor se aumenta o disminuye en un valor fijo. Como su nombre lo indica son usados para contar. Mientras que un acumulador es una variable cuyo valor aumenta o disminuye en un valor que puede variar. Es usado típicamente para ir sumando cantidades dentro un ciclo. Las líneas que siguen representan dos contadores y un acumulador:

```
# estos son contadores
contador = contador + 1
i = i + 2
# esto es un acumulador
suma = suma + edad
```

En este ejemplo, las variables *i* y *contador* siempre se incrementaría en 1 y en 2 unidades respectivamente, por su parte *suma* iría adicionando diferentes valores de la variable *edad* por cada iteración dentro de un ciclo.

El lenguaje Python maneja dos ciclos fundamentalmente que son el *for* y el *while*.

for

El ciclo *for* es una estructura de control que en Python se usa mayoritariamente para recorrer los elementos de una lista.

La sintaxis es:

```
for i in lista:  
    sentencias
```

En donde *i* es cada elemento que se obtiene del recorrido realizado sobre la lista. Echemos un vistazo a un ejemplo donde se recorre una lista llamada *lista* y luego se imprimen sus elementos (números del 1 al 5).

```
lista = [1,2,3,4,5]  
for i in lista:  
    print(i)
```

while

El ciclo *while* permite también ejecutar una o varias sentencias de forma iterativa o repetitiva. En este, el control sobre las repeticiones se realiza por medio de una condición que determina la continuidad o no del ciclo. Esta condición si es evaluada como *True* permite que se ejecuten las sentencias del cuerpo del ciclo, en caso contrario (evaluada como *False*), el ciclo no hará nada y el programa saltará a la siguiente línea de código. Su sintaxis es:

```
while (condicion):  
    //sentencia(s)
```

El siguiente ejercicio muestra los números del 1 al 10. Vemos que se usa una condición *a<=10*, la cual mientras se cumpla, es decir, sea evaluada como *True* mostrará el valor de la variable *a* y acto seguido la incrementará en 1. Por supuesto, el no cumplimiento de dicha condición marcará el final del ciclo.

```
a = 1  
while(a<=10):  
    print(a)  
    a=a+1
```

Ya para terminar esta sección acerca de estructuras repetitivas, merece la pena mencionar que en Python como en muchos otros lenguajes de programación existen las ordenes: *break* y *continue*, las cuales son usadas para proveer controles adicionales sobre los ciclos repetitivos. La palabra reservada *break* es usada en los

ciclos para terminar un flujo de ejecución mientras que por su parte la palabra clave *continue*, lo que hace es terminar la iteración actual y pasar el control a la siguiente, más no termina el ciclo definitivamente como sucede con la orden *break*.

1.6 Funciones

Una función también llamado método, es un conjunto o bloque de instrucciones encargadas de realizar una determinada tarea dentro de un programa. Su uso más importante radica en agrupar líneas de código que se puedan utilizar en diferentes partes, simplemente haciendo un llamado o invocación a la función en una sola instrucción por medio de su nombre.

La sintaxis para crear una función es como sigue:

```
def nombre_función(lista_parámetros):
    sentencia(s)
    [return]
```

Como se puede apreciar en la sintaxis, para crear funciones se usa en un principio la palabra clave *def* seguida del nombre de la función y de una lista de parámetros opcionales. Luego el cuerpo del método lo conforman las sentencias que harán parte de la tarea concreta que este realizará, así mismo si las circunstancias lo ameritan se especifica un *return* seguido de un valor de retorno, esto en el caso que quisieramos que la función devuelva o retorne explícitamente ciertos datos.

El siguiente método llamado *sumar()* hace la suma de dos variables *a* y *b* y retorna el resultado de dicha operación, para invocarlo usamos la palabra *sumar* y le pasamos como parámetros los números 5 y 6 que serán los valores de las variables *a* y *b*, entonces la operación suma se realizará y el resultado se almacenará en la variable *s*, la cual se imprime posteriormente.

```
# definimos el método con dos parámetros
def sumar(a , b):
    suma = a + b
    return suma

# invitamos el método
s = sumar(5, 6)
print("La suma es:" , s)
```

Es de anotar que en Python los métodos pueden retornar o no retornar ningún valor. Y también pueden tener parámetros por defecto cuyo valor se asigna al momento de su definición. En el siguiente ejemplo aplicamos este concepto:

```
def saludar(nombre = "Carlos"):
    print("Hola", nombre)

#invocamos el metodo saludar pasandole un parametro
saludar("Mario")
#invocamos el metodo sin pasarle parametro alguno
saludar()
```

Hola Mario
Hola Carlos

Notamos que si se le pasa un valor al parámetro *nombre* dicho valor es utilizado en las instrucciones a ejecutar dentro del método, mientras que, si no se le pasa valor alguno, el valor tomado es el asignado por defecto (la cadena "Carlos").

1.7 Clases y objetos

Python soporta el paradigma orientado a objetos por lo que es común crear programas donde se agrupan funcionalidades en clases y a partir de estas últimas crear objetos o instancias que permiten que se "activen" tales funcionalidades. Una clase en Python puede tener un aspecto como este que vemos a continuación:

```
# se define la clase
class Estudiante():
    # constructor
    def __init__(self, nombres, semestre):
        self.nombres = nombres
        self.semestre = semestre

    def saludar(self):
        print("Soy el estudiante: ", self.nombres, "y curso el semestre: ", self.semestre)

    def cambiarSemestre(self, semestre):
        self.semestre = semester
```

La clase Estudiante tiene dos variables de instancia: *nombres* y *semestre*. Así mismo tiene tres métodos: Un constructor paramétrico definido por la palabra clave `__init__()`, comúnmente usado para inicializar las variables definidas en la clase con los valores pasados como parámetros; Un método llamado `saludar()` que imprime una cadena que contiene el nombre y el semestre del estudiante (objeto creado); y un método `cambiarSemestre()` que establece o modifica el semestre del objeto. Procedemos a crear dos instancias llamadas *maria* y *juan* que llaman a los tres métodos definidos en la clase. Cabe anotar que el método constructor se llama automáticamente cuando creamos un objeto de la clase, mientras que los otros dos si deben ser llamados explícitamente.

```
# se crea un objeto
maria = Estudiante("María", "I")
maria.saludar()
juan = Estudiante("Juan", "II")
juan.saludar()
# Juan cambia de semestre
juan.cambiarSemestre("IV")
juan.saludar()
```

Soy el estudiante: María y curso el semestre: I
Soy el estudiante: Juan y curso el semestre: II
Soy el estudiante: Juan y curso el semestre: IV

CAPÍTULO 2

Introducción al Aprendizaje Automático

Temas

- 2.1 ¿Qué es aprendizaje automático?
- 2.2 Conceptos de aprendizaje automático
- 2.3 Tipos de aprendizaje automático
- 2.4 Problemas típicos en aprendizaje automático
- 2.5 Metodología CRISP-DM

Con este capítulo entramos en materia, podremos introducirnos en el aprendizaje automático gracias a la explicación de diversos conceptos fundamentales dentro de esta disciplina.

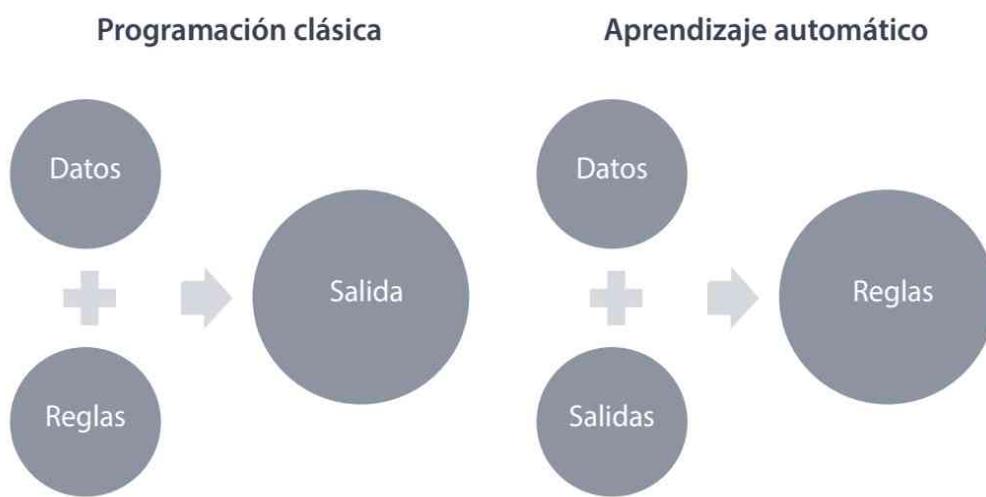
2.1 ¿Qué es aprendizaje automático?

Han sido varias las definiciones que han venido surgiendo a lo largo de la historia sobre aprendizaje automático. Entre las más reconocidas se encuentra la de Arthur Samuel en 1959, quien planteó que el *"Aprendizaje automático es el campo de estudio que da al computador la habilidad de aprender sin haber sido explícitamente programado para ello"*.

En un tiempo más reciente Tom Mitchell presentó la siguiente definición: "*Se dice que un programa de computadora aprende de la experiencia E con respecto a alguna tarea T y alguna medida de rendimiento P, si su rendimiento en T, medido por P, mejora con la experiencia E.*"

Podemos encontrar más definiciones como estas, pero simplemente el aprendizaje automático es una rama de la inteligencia artificial que busca que un programa de computador aprenda de un conjunto de datos con los cuales se entrena, y buscará indentificar un patrón con el que puede realizar predicciones sobre nuevos datos. A diferencia de la programación tradicional donde en primera instancia se procesan un conjunto de datos de entrada y por medio de un serie de reglas se genera una salida, en aprendizaje automático o *machine learning*, los datos y las salidas son los datos iniciales que mediante un proceso de entrenamiento producen las reglas, las cuales comúnmente reciben el nombre de modelo, siendo dicho modelo el resultado de detectar en los datos patrones o tendencias que se pueden usar para hacer predicciones sobre datos nunca vistos. Este proceso en mención es realizado por algoritmos, que debidamente ajustados, al final son los que van a permitir que puedan lograrse buenos resultados predictivos. Tales algoritmos se clasifican dependiendo el tipo de problema de aprendizaje que se esté manejando, pudiéndose usar uno para cada problema específico que se esté abordando, en donde deben afinarse sus parámetros, en la mayoría de los casos mediante prueba y error, a fin que se consiga el mejor desempeño del modelo de aprendizaje automático. Por esta razón, es que muy difícilmente una misma solución se puede replicar en diferentes escenarios, puesto que cada proceso está supeditado tanto a la naturaleza de los datos que utiliza como insumo, como a los ajustes que en materia de parámetros se realizan sobre el algoritmo seleccionado para una determinada situación.

La siguiente gráfica muestra el esquema de la programación clásica vs el aprendizaje automático.



Ahora bien, el aprendizaje automático implica la realización de una serie de etapas que se muestran en el siguiente esquema y se describen brevemente más abajo:



Preprocesamiento de datos: Es una de las etapas más importantes, dado que en ella se realizan tareas como limpieza y transformación de los datos para que queden en una forma adecuada y puedan ser usados por el algoritmo de aprendizaje automático. Es una etapa crucial, toda vez que estos algoritmos trabajan solamente con datos numéricos y en lo posible en una misma escala. Adicionalmente, a los datos faltantes o *nulos* se les debe también prestar atención ya que pueden provocar que el algoritmo opere de manera deficiente. El preprocesado implica estas y otras cuestiones, que serán estudiadas mucho más a fondo en el capítulo 4.

Separación en conjunto de entrenamiento y pruebas: En aprendizaje automático el conjunto de datos (dataset, en inglés) suele separarse en dos subconjuntos llamados entrenamiento y prueba. El primero se destina para entrenar y estimar los parámetros del modelo. Por otro lado, el segundo se usa para hacer predicciones y probar el modelo con datos diferentes a los de entrenamiento para ver si arroja los resultados esperados. En general, para conjuntos de datos de mediano tamaño las divisiones que más se usan son: 70%-30% u 80%-20% para entrenamiento y prueba respectivamente.

Configuración del algoritmo: En esta etapa, básicamente se crea una instancia del algoritmo a utilizar y se definen los llamados hiperparámetros para ese algoritmo con valores apropiados que el científico de datos o programador debe ir ajustando comedidamente. Los hiperparámetros se diferencian de los parámetros en que estos últimos los define el modelo internamente durante el entrenamiento, no teniendo el usuario ninguna participación directa en la generación de sus valores. Algunos hiperparámetros empleados con regularidad en algoritmos de machine learning son el *número de épocas* y *la tasa de aprendizaje*.

Entrenamiento del modelo: Consiste en proporcionarle al objeto o instancia del algoritmo de aprendizaje automático un conjunto de datos de entrenamiento para que pueda aprender, logrando así estimar los parámetros del modelo de aprendizaje.

Predicción: Una vez generado el modelo se puede probar su nivel de predicción pasándole muestras del conjunto de pruebas. El resultado de la predicción sobre una muestra es un valor continuo o discreto que debe ser los más cercano posible al valor esperado.

Evaluación: La evaluación es el proceso de determinar numéricamente que tan efectivo fue nuestro modelo de aprendizaje automático. Esta efectividad en el rendimiento parte del supuesto de que a menor diferencia entre la salida esperada y la salida predicha mejor es la evaluación.

Para evaluar un modelo existen diferentes métricas que se pueden usar dependiendo del tipo de problema de aprendizaje automático, las cuales se estarán estudiando más adelante en el libro.

Hay una última tarea no incluida en el esquema de arriba llamada **exportación**, y consiste en que, una vez el modelo se encuentre en una fase final, es decir, con sus hiperparámetros debidamente afinados y con un valor adecuado de la métrica de evaluación, es posible proceder con la exportación del modelo a un archivo en memoria secundaria, con el fin de usarlo más adelante sin necesidad de repetir todos los pasos previos, hecho que sin duda nos ahorra tiempo y nos hace más productivos.

2.2 Conceptos de aprendizaje automático

Existe una terminología bastante amplia relacionada con el aprendizaje automático, en esta sección solo se mencionarán algunos de los conceptos que a consideración del autor son los más significativos, dado que se estarán usando muy a menudo en el transcurso del libro:

- **Conjunto de datos (dataset):** Los conjuntos de datos no son más que los datos almacenados en un formato específico, recibidos por el algoritmo de aprendizaje para ser entrenado y ser probado. Los *datasets* normalmente son de tipo estructurado y no estructurado. Los primeros se caracterizan por manejar una estructura predefinida, como puede ser una tabla de una base de datos, mientras que los segundos no tienen un formato específico como por ejemplo el cuerpo del mensaje de un correo electrónico.

En este libro se usarán muy a menudo conjuntos de datos estructurados almacenados en archivos en formato csv (*comma separated values*), en ellos la información se representa en forma de tabla, donde las filas reciben el nombre de instancias, ejemplos o muestras y las columnas el nombre de características o

variables. Como ejemplo, considere un conjunto de datos con información sobre casas en venta, este tendría por ejemplo 10.000 objetos definidos uno por cada fila y 5 características definidas una para cada columna: *dirección, num_habitaciones, área_m2, estrato y precio*.

- **Variables descriptivas:** Son los elementos que describen las instancias almacenadas en el conjunto de datos. Estas variables o atributos tienen un tipo de dato asociado. Por ejemplo, en un conjunto de datos con información de casas, algunas variables descriptivas serían: *num_habitaciones, área_m2, estrato*, etc.
- **Variable destino o etiqueta:** Característica o conjunto de ellas usadas para etiquetar las instancias, aspecto necesario en algunos tipos de algoritmos para realizar la tarea de entrenamiento del modelo. También son designadas para representar aquellos atributos que se quieran predecir, Por ejemplo, si consideramos el mismo conjunto de datos de información de casas en venta, el *precio*, sería la variable destino (variable a predecir).
- **Tipos de datos:** Cada variable o atributo tiene un tipo que define que valores puede tomar. Los tipos de datos básicamente son:
 - Numéricos: Representan cantidades numéricas que pertenecen al conjunto de los números reales. Ej. Estatura: 1,85.
 - Categóricos: Representan valores de tipo cadena de caracteres pertenecientes a un rango limitado y hacen referencia a cualidades o categorías: Ej: Color: Azul, Verde, Rojo.
 - Ordinales: Al igual que los categóricos son cadenas, con la diferencia de que se puede establecer un orden entre ellos: Ej. Talla: L, M, S. En este caso: L>M>S.

La siguiente tabla muestra un conjunto de datos simplificado para determinar si una persona tiene derecho o no a un crédito bancario. Como se puede apreciar cuenta con seis atributos y cuatro muestras. Además, atributos como el salario y la edad son de tipo numérico, la historia crediticia es categórico o nominal y la ocupación es ordinal.

Atributos descriptivos					Etiqueta
Id	Salario	Historia crediticia	Ocupación	Edad	Préstamo
1	1.800.000	Bien	Profesional	24	SI
2	700.000	-	Técnico	18	NO
3	2.000.000	Bien	Profesional	20	SI
4	1.000.000	Mal	Tecnólogo	55	NO

Entiéndase que, en la mayoría de algoritmos de aprendizaje de máquina, los valores de los atributos deben transformarse a un tipo numérico para que se realice eficientemente el proceso de entrenamiento y con ello conseguir buenos resultados en materia de predicción.

- **Modelo:** Son el resultado del proceso de aprendizaje automático, y típicamente es producido por un algoritmo a partir del entrenamiento con una parte de los datos. El modelo constituye un conocimiento base (patrón o tendencia) con el cual es posible realizar predicciones al aplicarlo sobre nuevos datos de entrada. Además, en general los modelos están asociados con la determinación de unos coeficientes llamados pesos.

Es claro que la terminología mostrada no es rigurosa ni mucho menos extensa, pero se puede considerar valida como punto de partida para comprender mejor los siguientes temas del libro. Sin embargo, tenga presente el lector que a medida que avancemos se irán explicando otros términos y conceptos imprescindibles dentro del ámbito del aprendizaje automático.

2.3 Tipos de aprendizaje automático

Existen diferentes tipos de aprendizaje automático, entre los más destacados se encuentran: supervisado, no supervisado y por refuerzo.

- **Aprendizaje supervisado:** En este tipo de aprendizaje todas las muestras del conjunto de datos son etiquetadas con un resultado (valor predefinido) o una categoría utilizando una variable llamada destino u objetivo. En el ejemplo anterior las filas 1 y 3 tienen en la variable *préstamo* el valor SI, mientras que las filas 2 y 4 tienen el valor NO. Estos datos etiquetados ayudan a que el modelo internamente construya un patrón y sea capaz de predecir si una muestra nueva (cliente bancario), SI aprueba o NO aprueba un préstamo.

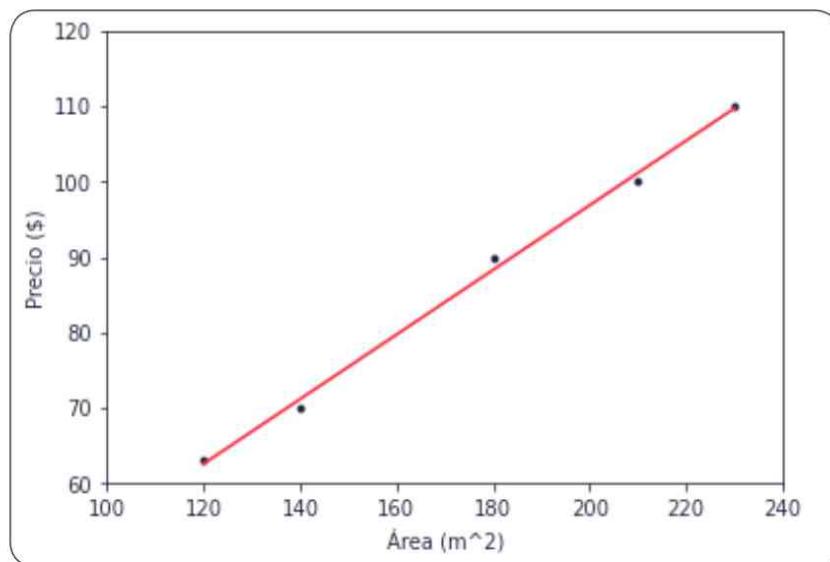
Se distinguen dos tipos de aprendizaje supervisado:

- Regresión: La regresión permite predecir un valor continuo, tomando en consideración varias variables de entrada. O en otras palabras busca encontrar la relación entre una variable dependiente con algunas variables independientes.

Algunos problemas por ejemplo que se pueden abordar con la técnica de regresión podrían ser:

- Predecir el precio de una casa a partir del área en metros cuadrados.
- Predecir la temperatura en el próximo mes.

- Predecir el tiempo de vida de un dispositivo electrónico.



La gráfica anterior es un ejemplo de una regresión lineal, que muestra la relación directa entre el área en metros cuadrados y el precio de una casa. Este es un tipo de regresión lineal simple, debido a que solo tiene una variable independiente (área) usada para predecir el *precio*, pero podemos encontrar regresión lineal múltiple, la cual maneja varias variables independientes y también regresión no lineal o polinómica como veremos más adelante en el libro.

- o Clasificación: Tipo de algoritmos de aprendizaje en los que el valor a predecir es de tipo discreto, lo que quiere decir que asocia una instancia a una categoría o clase particular a la cual pertenece. Por ejemplo, un modelo detector de Spam como el esquematizado a continuación, es capaz de determinar si un correo electrónico pertenece a la categoría SPAM o no SPAM dependiendo del texto del mismo.



Otros ejemplos de problemas de clasificación pueden ser:

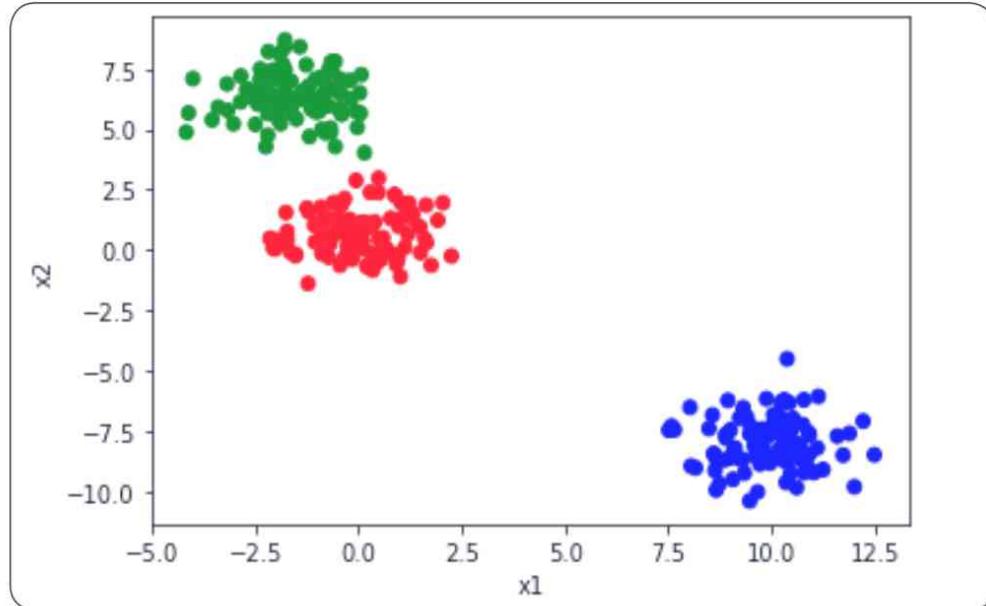
- Clasificar flores iris en las clases: setosa, virginica y versicolor.
- Determinar si un tumor es maligno o benigno.
- Predecir el ganador de las próximas elecciones presidenciales entre dos candidatos A y B.
- Aprendizaje no supervisado: En esta forma de aprendizaje no contamos con etiquetas para las muestras de entrenamiento del modelo. Es decir, el modelo actúa directamente sobre los datos de entrada y busca hallar relaciones entre ellos basándose en características en común.

Los dos problemas que el aprendizaje automático no supervisado trata de resolver principalmente son el agrupamiento o clustering y la reducción de la dimensionalidad. En cuanto al primero, el algoritmo más destacado es el algoritmo k-medias (k-means, en inglés), que consiste en crear grupos entre los datos de acuerdo a algún criterio que típicamente es distancia o similitud. En cambio, en relación al segundo, son algoritmos enmarcados dentro de la técnica de procesamiento, siendo aplicables a tareas de aprendizaje supervisado y no supervisado y cuya meta es disminuir o comprimir el número de variables o características, para luego proyectarlas sobre un nuevo espacio multidimensional donde la información esté mucho más condensada y lograr con esto producir resultados más precisos, lo que a su vez permite también reducir el costo computacional que implica trabajar con un número amplio de características.

El aprendizaje no supervisado es muy usado por ejemplo en el campo de la medicina para el diagnóstico de enfermedades, así como en el área del marketing más específicamente en tareas de segmentación de clientes en donde son agrupados según sus preferencias para así ofrecerles productos y servicios de una manera más personalizada.

Sin embargo, hay otros algoritmos de agrupamiento como DBSCAN que también serán tratados más adelante en el libro, con gran aplicación en la solución de diferentes problemas como la detección de anomalías, sistemas de recomendación, entre otros.

- Aprendizaje semisupervisado: Utiliza datos de entrenamiento tanto etiquetados como no etiquetados. Si juntamos datos etiquetados con datos no etiquetados en ciertas situaciones se consiguen modelos más exactos.



- o Aprendizaje por refuerzo: En este tipo de modelos el aprendizaje se consigue a partir de ensayo y error, teniendo en cuenta que cada vez que se da un acierto en una predicción se establece una recompensa y en caso contrario una penalización. Esto es en esencia, la aplicación de una retroalimentación (feedback) en el proceso de aprendizaje. Lo podemos comparar a la forma como, por ejemplo, aprende un niño en sus primeros años de vida, en donde cada vez que acierta en la identificación de un objeto de su entorno recibe una recompensa por parte de sus padres, mientras que por el contrario cuando falla es corregido por sus progenitores para que en el futuro no vuelva a equivocarse. Estos algoritmos de aprendizaje llamados *agentes*, en definitiva, buscan aprender estrategias para minimizar estas penalizaciones y maximizar las recompensas, con el objetivo de encontrar el mejor ajuste que potencie su capacidad predictiva.

2.4 Problemas típicos en aprendizaje automático

Los problemas más frecuentes que deben afrontar quienes trabajan en el campo del aprendizaje automático son básicamente:

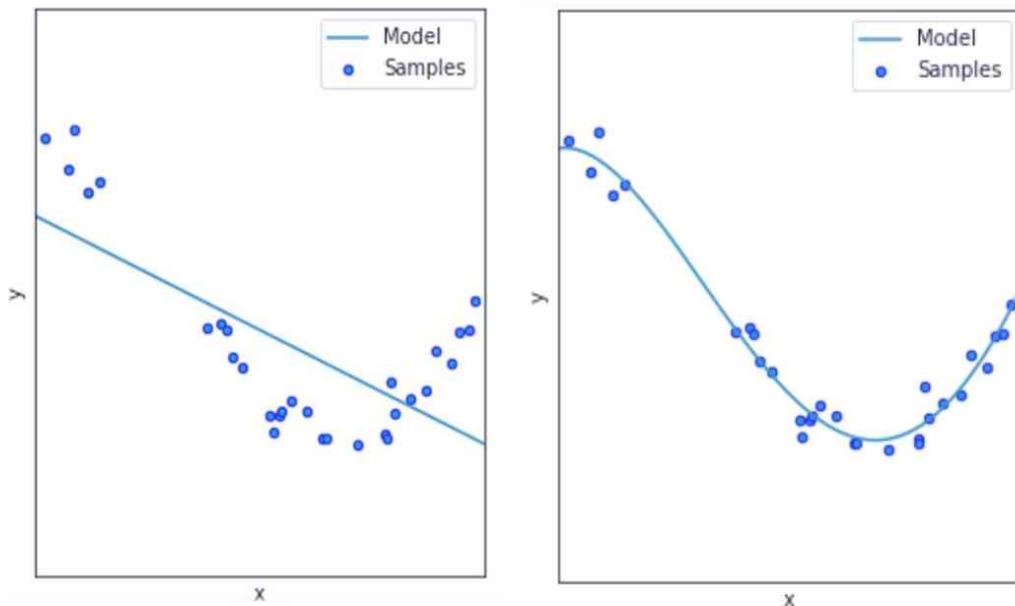
- Calidad de los datos: Algunos de los problemas relacionados con los datos son los valores faltantes, duplicados o incorrectos que pueden provocar resultados no deseados o modelos imprecisos.

- **Maldición de la dimensionalidad:** Los modelos trabajan bien con un conjunto específico de características o atributos, si esto no es manejado correctamente podríamos conseguir un modelo complejo con una gran cantidad de variables, algunas sin aportar verdadero valor, provocando que el rendimiento final no sea el esperado y se pierda exactitud a la hora de predecir.
- **Subajuste (*Underfitting*):** Este error ocurre cuando el modelo es tan simple y no se ha entrenado con los suficientes datos, por tanto, no es capaz de representar adecuadamente la relación entre las variables descriptivas y la variable destino.

Como ejemplo, suponga que ha entrenado un modelo para clasificar perros y gatos, y se tienen como atributos descriptivos: *forma de las orejas*, *tamaño de la nariz*, *especie* y como variable destino: *tipo*. Se quiere predecir si una instancia nueva corresponde a un perro o a un gato. Suponga que se han tomado por ejemplo diez instancias de perros para entrenar el modelo, está claro que la información suministrada no es de ninguna manera suficiente para que el modelo aprenda a encontrar un patrón que ayude efectivamente a diferenciar a estos animales, dado que existe un gran número de especies con diferencias de todo tipo en tamaño, pelaje, forma de las orejas, entre otras características, que suscitan una multitud de detalles no considerados durante el entrenamiento y que son imprescindibles para hacer la debida clasificación.

- **Sobreajuste (*Overfitting*):** A diferencia del anterior, el sobreajuste está asociado a modelos tan complejos que se ajustan a todos los datos de entrenamiento, pero no generalizan bien y por tanto no son capaces de predecir correctamente instancias nuevas. Como ejemplo, tomando la misma situación anterior, pero en este caso el modelo ha sido entrenado con datos de diferentes tipos de perros, y aprendió tanto de ellos que no es capaz de predecir adecuadamente un perro con atributos diferentes a los usados en el entrenamiento, esto significa que no está lo suficientemente generalizado para reconocer este tipo de instancias particulares. En la siguiente figura se observa como el modelo de la izquierda con una función lineal no ajusta bien los datos (subajuste), por otro lado, el modelo de la derecha con función polinómica ajusta los datos de manera correcta, tratándose de un modelo más apropiado, puesto que no aprendió al detalle todos los puntos de entrenamiento.

En *machine learning*, cuando se tiene un modelo que ajusta insuficientemente los datos, es decir, como el de la línea que no pasa por todos los puntos se dice que tiene un sesgo elevado. En cambio, un modelo complejo donde la línea pasa por todos los puntos, aunque parecería bueno a simple vista, no lo es, sería un modelo sobreajustado capaz de cubrir una gran variedad de información no deseada incluyendo ruido y valores atípicos, en cuyo caso dispone de una alta varianza.



En un próximo capítulo abordaremos algunas técnicas que se usan para combatir el sobreajuste como la validación cruzada, la regularización, aumento de datos, etc. Donde las dos primeras han demostrado ser más aplicables en la vida real, dado que, el aumentar los datos si bien esto haría que el modelo no pueda ajustar todos los datos (evitando el sobreentrenamiento), en la práctica no todas las veces es posible obtener más datos de los que ya se tienen y que a su vez sean significativos. De igual forma, podría ser una solución usar un modelo más simple, esto es, con menos datos y/o variables, pero quizás al final esto no resulte adecuado para tareas determinadas, debido a la posibilidad de incurrir en un problema de subajuste o subentrenamiento.

2.5 Metodología CRISP-DM

CRISP-DM (Cross Industry Standard Process for Data Mining) por sus siglas en inglés es el estándar abierto más usado para los proyectos de aprendizaje automático y minería de datos. Es un modelo genérico, flexible y cíclico diseñado para ser adaptado por empresas o individuos para cubrir sus necesidades en procesos de minería de datos, es decir, aquellos en donde es primordial descubrir patrones o información útil que se encuentra oculta en los datos.

CRISP-DM está conformado por las siguientes fases:

1. Conocimiento de negocio: La primera fase consiste en entender el negocio y organización a la cual se va a ayudar a suplir alguna necesidad por medio de soluciones de aprendizaje automático.

2. Conocimiento de los datos: En esta fase el analista debe tener conocimiento de las diferentes fuentes de datos con las cuales cuenta la organización y los tipos de datos disponibles.
3. Preparación de los datos: Consiste en convertir los datos en algún formato específico, de tal manera que sean lo suficientemente aptos para poder ser usados por un algoritmo de aprendizaje automático, el cual generará un modelo con el que se puedan hacer predicciones sobre datos nuevos.
4. Modelado: Se pueden probar diferentes algoritmos para crear varios modelos, pero solo se escogerá el que permita obtener la salida deseada.
5. Evaluación: El modelo es sometido a diferentes tareas de evaluación o desempeño a fin de verificar que sea capaz de hacer predicciones con una buena exactitud.
6. Despliegue: Esta fase cubre todo el proceso de integrar un modelo de aprendizaje automático dentro de un sistema de información organizacional, permitiendo que pueda ser operado por los usuarios finales.

En el siguiente capítulo exploraremos las herramientas que nos van a permitir trabajar con nuestros datos y poder crear modelos predictivos para diferentes tipos de problemas tomando como insumo dichos datos.

CAPÍTULO 3

Herramientas para el aprendizaje automático

Temas

- 3.1 Manejo básico de datos con Pandas
- 3.2 Manejo de arreglos con Numpy
- 3.3 Creando gráficos con Matplotlib
- 3.4 Breve Introducción a Scikit-learn

Las herramientas que se usarán en este libro para explicar los diferentes métodos y técnicas de aprendizaje automático son librerías basadas en el lenguaje de programación Python, que como sabemos coloca al alcance del programador una gran cantidad de utilidades como paquetes, módulos, etc; los cuales son contenedores de una amplia gama de algoritmos muy potentes y optimizados para el trabajo con aprendizaje automático y ciencia de datos en general.

Por tanto, para seguir los ejemplos es necesario además de tener instalado Python, instalar las siguientes librerías:

- NumPy
- Scikit-Learn
- Pandas
- Matplotlib

Para mayor comodidad se recomienda instalar la distribución Anaconda, la cual dispone del núcleo del lenguaje de programación Python y de varias librerías y paquetes como los mencionados más arriba, además viene equipada con un interesante IDE (Entorno de desarrollo integrado) llamado Spyder, ideal para poner en marcha los códigos del libro y poder analizar su funcionamiento. Anaconda trae consigo también un gestor de paquetes llamado conda con el que se pueden

hacer actualizaciones y un editor web bastante interactivo utilizado para escribir y compartir código fuente, denominado Jupyter Notebook. Este último fue el usado en los ejemplos de programación o material complementario que acompañan la presente obra.

El enlace de descarga de los instaladores de Anaconda para los sistemas operativos Windows, MacOs y Linux se muestra a continuación:

<https://www.anaconda.com/products/individual>



3.1 Manejo básico de datos con PANDAS

Todo proceso de aprendizaje automático requiere de un conjunto de datos. En algunos de nuestros ejercicios usaremos uno bastante empleado sobretodo en tareas de regresión, disponible en el sitio web de Kaggle y enfocado en el problema de la predicción de precios de casas. La información de este dataset corresponde a datos de viviendas de las ciudades de Sydney y Melbourn y cuenta con 4.600 instancias (filas) y 18 características (columnas).

El lector lo podrá encontrar junto con los archivos anexos de este libro o también lo puede descargar de la siguiente url:

<https://www.kaggle.com/shree1992/housedata#data.csv>

El conjunto de datos en mención tiene 18 características, algunas de las cuales se describen brevemente a continuación:

Bad_rooms	Número de habitaciones
Bath_rooms	Número de baños
Sqft_living	Área de la sala en pies cuadrados
Sqft_lot	Área del terreno en metros cuadrados
Price	Precio de la casa

Supongamos que ya disponemos de nuestro archivo con el conjunto de datos anterior en algún lugar del disco duro de nuestro computador, miremos como podemos cargarlo y realizar algunas operaciones básicas sobre él, antes de meternos de lleno en tareas propias de aprendizaje automático.

Para realizar este paso vamos a usar la librería llamada Pandas con la cual podemos cargar archivos en diferentes formatos. En nuestro caso el archivo con los datos se encuentra en formato csv, que como vimos es uno de los más usados y más sencillos para representar datos en forma de tabla. En este, las columnas se separan por comas y las filas por un salto de línea.

Lo primero que se debe hacer para cargar un dataset en python es importar la biblioteca Pandas y utilizar un alias que por convención suele ser pd. Luego se carga el archivo con extensión .csv mediante la función read_csv(), pasándole como parámetro la ruta del fichero, esta última acción devuelve un objeto dataframe, estructura de datos parecida a una matriz donde las columnas representan las características y las filas las instancias del conjunto de datos.

```
import pandas as pd
df = pd.read_csv("precios_casas.csv")
```

De esta manera hemos logrado leer el archivo y cargarlo en un dataframe con los parámetros por defecto que ofrece la función read_csv(), la cual consta de muchos otros argumentos para hacer una lectura mucho más precisa de los datos, y que el usuario puede consultar en la documentación oficial de la librería: <https://pandas.pydata.org/docs/reference/index.html#api>.

Con el dataframe anterior podemos realizar algunas acciones como por ejemplo visualizar su contenido con el método print(). Si el dataframe es muy grande se mostrará solamente las 5 primeras y las 5 últimas filas.

```
print(df)
```

```

      date      price bedrooms ... city statezip country
0 2014-05-02 00:00:00 3.130000e+05    3.0 ... Shoreline WA 98133 USA
1 2014-05-02 00:00:00 2.384000e+06    5.0 ... Seattle WA 98119 USA
2 2014-05-02 00:00:00 3.420000e+05    3.0 ... Kent WA 98042 USA
3 2014-05-02 00:00:00 4.200000e+05    3.0 ... Bellevue WA 98008 USA
4 2014-05-02 00:00:00 5.500000e+05    4.0 ... Redmond WA 98052 USA
...
4595 ... ... ... ... ... ...
4596 2014-07-09 00:00:00 3.081667e+05    3.0 ... Seattle WA 98133 USA
4597 2014-07-09 00:00:00 5.343333e+05    3.0 ... Bellevue WA 98007 USA
4598 2014-07-10 00:00:00 4.169042e+05    3.0 ... Renton WA 98059 USA
4599 2014-07-10 00:00:00 2.034000e+05    4.0 ... Seattle WA 98178 USA
[4600 rows x 18 columns]

```

En muchas ocasiones no queremos mostrar todos los datos sino un puñado de ellos, para hacer esta labor existen funciones que se pueden llamar con el mismo objeto df y son:

- `head()`: Devuelve las primeras 5 instancias.
- `tail()`: Devuelve las 5 últimas instancias.
- `sample(tamaño)`: Devuelve un subconjunto aleatorio de instancias de tamaño igual al número pasado como parámetro a la función. Si, por ejemplo, le pasamos 100 nos devuelve igual número de filas.

Por otra parte, dos propiedades de uso muy común usadas sobre los dataframe son `columns` y `shape`. La primera nos devuelve un índice de Pandas con los nombres de las columnas, mientras que la segunda nos devuelve una tupla con la cantidad de filas y de columnas.

```
df.columns
```

```

Index(['date', 'price', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot',
       'floors', 'waterfront', 'view', 'condition', 'sqft_above',
       'sqft_basement', 'yr_built', 'yr_renovated', 'street', 'city',
       'statezip', 'country'],
      dtype='object')

```

```
df.shape
```

(4600, 18)

CAP. 3 - HERRAMIENTAS PARA EL APRENDIZAJE AUTOMÁTICO

Los dataframes de Pandas poseen dos atributos muy usados para acceder a su contenido, se trata de *iloc* y *loc*. *iloc* recibe como parámetros entre corchetes dos números que representan fila y columna respectivamente, con los que es posible localizar un elemento, tal y como ocurre cuando trabajamos con matrices. Sin embargo, si se pasa un solo parámetro se considerará como índice de la(s) fila(s) a mostrar. Miremos algunos ejemplos:

df.iloc[9]	Obtiene la fila en la posición 10, ya que toma 0 como primer índice.
df.iloc[:10]	Obtiene las primeras 10 filas, sin incluir la decima.
df.iloc[0,1]	Obtiene la celda (0,1), Intercepción entre fila 0 y columna 1.
df.iloc[[0,1,2], 0:5]	Obtiene las primeras 5 columnas de las filas 0,1 y 2.

Por otro lado, el atributo *loc*, aunque sirve para lo mismo que *iloc*, tiene un manejo un tanto diferente, ya que las columnas se indican mediante su nombre, pero las filas si se siguen usando con valores numéricos. El hecho de que se use el nombre para identificar una columna y no su posición es muy beneficioso porque si eliminamos alguna columna del dataframe no tendríamos que recalcular sus posiciones nuevamente.

df.loc[0,'price']	Obtiene el valor de la columna precio de la primera fila
df.loc[:3,'price':'bathrooms']	Obtiene las primeras filas desde la 0 hasta la 3, con las columnas desde price a bathrooms

Así mismo existen otras funciones que vale la pena mencionar como *describe()* con la cual se puede obtener una descripción detallada de los datos de un dataframe. Esta descripción incluye entre otras cosas la cantidad de valores no vacíos (*count*), el valor medio (*mean*), la desviación estándar (*std*), el valor mínimo y el valor máximo de dichas columnas.

```
df.describe()
```

```

      price    bedrooms    ...    yr_built    yr_renovated
count  4.600000e+03  4600.000000  ...  4600.000000  4600.000000
mean   5.519630e+05  3.400870  ...  1970.786304  808.608261
std    5.638347e+05  0.908848  ...  29.731848  979.414536
min    0.000000e+00  0.000000  ...  1900.000000  0.000000
25%   3.228750e+05  3.000000  ...  1951.000000  0.000000
50%   4.609435e+05  3.000000  ...  1976.000000  0.000000
75%   6.549625e+05  4.000000  ...  1997.000000  1999.000000
max   2.659000e+07  9.000000  ...  2014.000000  2014.000000

```

Ahora bien, el método `describe()` sin argumentos obtiene información de los datos numéricos, pero si desea conseguir información estadística de todas las columnas incluyendo las no numéricas, se adiciona el parámetro `include='all'` a la función anterior, logrando con esto que se amplíe la información del resumen generado:

```
df.describe(include='all')
```

```

      date        price    ...    statezip    country
count  4600  4.600000e+03  ...  4600  4600
unique  70      NaN  ...     77      1
top    2014-06-23 00:00:00      NaN  ...  WA 98103  USA
freq    142      NaN  ...     148  4600
mean    NaN  5.519630e+05  ...      NaN  NaN
std     NaN  5.638347e+05  ...      NaN  NaN
min    NaN  0.000000e+00  ...      NaN  NaN
25%   NaN  3.228750e+05  ...      NaN  NaN
50%   NaN  4.609435e+05  ...      NaN  NaN
75%   NaN  6.549625e+05  ...      NaN  NaN
max   NaN  2.659000e+07  ...      NaN  NaN

```

Por otro lado, es posible también eliminar alguna columna o atributo del conjunto de datos que consideremos no muy relevante para nuestras pretensiones, mediante la función `drop()`. Por medio del parámetro `columns` se pueden especificar los nombres de las columnas que se desean borrar utilizando una lista. A manera de ejemplo vamos a eliminar la columna `view` de nuestro dataframe. El código sería el siguiente:

```
df.drop(columns=['view'])
```

Ahora, si quisieramos guardar los cambios realizados sobre nuestro conjunto de datos en un archivo `.csv` en el disco duro para poder reutilizarlo más adelante,

bastaría con utilizar la función `to_csv()`. A esta función le pasaremos como un primer parámetro la ruta donde se ha de guardar el archivo junto con el parámetro `index=False`. Este último argumento hace que no se incluya una primera columna como índice para cada instancia en el archivo generado.

```
df.to_csv('precios_casas2.csv', index=False)
```

Como es de esperarse existen muchas otras operaciones aplicadas a los dataframes no incluidas en esta sección del libro por tratarse de un apartado meramente introductorio. Sin embargo, si el lector desea profundizar más sobre la biblioteca Pandas puede consultar la documentación oficial de la librería.

3.2 Manejo de arreglos con Numpy

La librería Numpy adiciona soporte para el manejo de arreglos multidimensionales, colocando al alcance del usuario un gran número de funciones para trabajar con estas estructuras. En Numpy un arreglo es del tipo `ndarray` (arreglo de n dimensiones) y a diferencia de otro tipo de colecciones como las listas de Python, todos sus elementos deben ser del mismo tipo.

Antes de comenzar a ver cómo podemos crear arreglos y aplicar operaciones sobre ellos, lo primero que debes hacer es importar la librería, esto lo conseguimos mediante la siguiente instrucción, donde `np` es un alias:

```
import numpy as np
```

3.2.1. Creación de arreglos

La creación de arreglos se puede hacer de varias maneras, una de ellas es a través de la función `array()`. Veamos el siguiente ejemplo donde creamos un vector de 5 elementos:

Note el lector que para saber las dimensiones del arreglo `vec1` se usa la función `shape()`, que nos arroja `(5,)` esto indica que se trata de un vector o arreglo unidimensional de 5 elementos.

```
import numpy as np
# creamos un vector con 5 elementos
vec1 = np.array([1, 2, 3, 4, 5])
# mostamos los elementos
print(vec1) # [1 2 3 4 5]
# mostramos las dimensiones
print(vec1.shape) # (5,)
```

Ahora bien, para crear una matriz de 3 filas y 3 columnas simplemente agregamos un par de corchetes y dentro tres vectores que representaran las filas de la matriz separados por comas:

```
matriz = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(matriz)
print(matriz.shape) # (3, 3)
```

Vemos que al imprimir las dimensiones de la matriz con la propiedad *shape* el resultado es (3,3), indicando que se trata de un arreglo de dos dimensiones (matriz) de 3 filas y 3 columnas.

Por otra parte, también se puede dar el caso de querer crear un vector o matriz cuyos elementos sean ceros. Esto lo podemos lograr con la función zeros().

```
# creamos un vector de ceros
vec2 = np.zeros(4);
print(vec2) # [0. 0. 0. 0.]
# creamos una matriz 2x3 de ceros
matriz2 = np.zeros((2,3));
print(matriz2)
"""
[0. 0. 0. 0.]
[[0. 0. 0.]
 [0. 0. 0.]] """
```

Existe una opción útil para crear arreglos de números aleatorios mediante la función random() del submódulo numpy.random. El siguiente código crea un matriz de 2x4 y la llena con números aleatorios.

```
matriz3 = np.random.random((2, 4))
print(matriz3)
"""
[[0.06933646 0.65641395 0.02347873 0.21154404]
 [0.22189871 0.83847063 0.20327317 0.52964326]]
"""
```

Los números generados son de tipo float64 y se encuentran en un rango entre 0.0 y 1.0. Así mismo, definir un arreglo a partir de una lista de Python resulta sencillo, siendo la función array() la encargada de hacer esta tarea:

```
listax = [5, 6, 7, 8, 9]
vec3 = np.array(listax)
print(vec3) # [5 6 7 8 9]
```

Otra función útil es arange() que permite crear un arreglo de números secuenciales.

```
nums = np.arange(10);
print(nums) #[0 1 2 3 4 5 6 7 8 9]
```

Este otro ejemplo con la función arange() muestra los números comprendidos entre 1 y 5 con incremento de 0.5

```
a = np.arange(1, 5, 0.5) #[1. 1.5 2. 2.5 3. 3.5 4. 4.5]
```

3.2.2. Acceso a elementos

El acceso a los elementos de un arreglo es similar al de las listas Python y se realiza mediante un índice (posición) que va desde cero hasta la longitud del arreglo – 1. Suponga que deseamos mostrar el primer y tercer elemento del vector vec1 creado anteriormente.

```
print(vec1[0]) # 1
print(vec1[2]) # 3
```

Accedamos al número 1 y 6 del arreglo que hace un momento llamamos *matriz*. En este caso es preciso indicar la fila y la columna del elemento que queremos acceder.

```
print(matriz[0][0]) # 1  
print(matriz[1][2]) # 6
```

Ahora bien, existe otra forma de realizar esta acción utilizando una expresión condicional en vez de un índice numérico. El siguiente ejemplo crea un vector de números del 0 al 9 y obtiene aquellos que son pares mediante el empleo de un índice booleano.

```
nums = np.arange(10);  
print(nums) #[0 1 2 3 4 5 6 7 8 9]  
datos = nums[nums % 2 == 0]  
print(datos) # [0 2 4 6 8]
```

Por otra parte, cuando se trabajan arreglos con Numpy en muchas situaciones resulta necesario acceder no solo a un elemento si no a ciertas porciones de la matriz, esta operación usada también en las listas recibe el nombre de *slicing*. En los siguientes ejemplos veremos cómo manejar esta característica de suma importancia cuando se manejan conjuntos de datos:

```
matriz4 = np.array([[3, 4, 5],  
                   [6, 7, 8],  
                   [0, 1, 2]])  
print(matriz4)  
[[3 4 5]  
 [6 7 8]  
 [0 1 2]]
```

La sintaxis para hacer *slicing* en la matriz es: [inicio:fin , inicio:fin], donde la primera pareja de números inicio:fin hace referencia a la fila y la segunda pareja a la columna.

Como ejemplo, vemos cómo podemos acceder los elementos de la *matriz4* desde la fila uno hasta la tres sin incluir la tres, y todas las columnas hasta las dos sin incluir la dos. Como podemos observar el resultado es un arreglo de 2 dimensiones:

```
matriz_n = matriz4[1:3 , :2]
print(matriz_n)
#####
[[6 7]
 [0 1]]
#####
```

De igual forma podemos acceder a todas las filas, pero apuntando a la primera columna de *matriz4* donde el resultado es un vector de tres elementos.

```
matriz_n = matriz4[:, 1]
print(matriz_n) # [4 7 1]
print(matriz_n.shape) # (3,)
```

3.2.3. Redimensionamiento

La función `reshape()` se usa para redimensionar un arreglo. Por ejemplo, mediante el siguiente código pasamos la *matriz4* que es inicialmente de 3x3 a una matriz de 1 fila y 9 columnas.

```
matriz_n = matriz4.reshape(1, -1)
print(matriz_n) #[[3 4 5 6 7 8 0 1 2]]
print(matriz_n.shape) # (1, 9)
```

En este caso el primer parámetro (1) de la función indica que la matriz se va a redimensionar a 1 fila, y el segundo parámetro el (-1) hace que sea la misma función quien decida cuantas columnas deberá tener la matriz de salida.

Para terminar la explicación de `reshape()` veamos cómo redimensionar un vector de 4 elementos a una matriz de 2x2:

```
vec3 = np.array([1,2,3,4])
# redimensionamos el vector vec3 a una matriz de 2x2
matriz5 = vec3.reshape(2,2)
print(matriz5)
"""
[[1 2]
 [3 4]]
"""

print(matriz5.shape) # (2, 2)
```

Tenga presente que, si la cantidad de elementos del vector fuera de 5 este no pudiera ser transformado a una matriz con las dimensiones especificadas puesto que no coincide con el tamaño de una matriz de 2x2, que por supuesto viene siendo 4. Al tratar de usar reshape() en un caso como este obtendríamos un error como el que se muestra a continuación:

```
ValueError: cannot reshape array of size 5 into shape (2,2)
```

3.2.4. Operaciones matemáticas

Numpy es una herramienta que ofrece un conjunto de operaciones matemáticas para trabajar con arreglos. En este subapartado explicaremos algunas operaciones básicas como: suma, resta, multiplicación, división y producto punto.

Suma

Esta operación la podemos hacer con el operador (+) o con la función add(), considerando de antemano que dicha operación toma como entrada dos o más matrices de igual dimensión y devuelve otra matriz como resultado:

```

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = A + B
print(C)
"""
[[ 6  8]
 [10 12]]
"""

D = np.add(A, B)
print(D)
"""
[[ 6  8]
 [10 12]]
"""

```

Resta

Se puede realizar la operación resta con el operador (-) o la función subtract(). Tomemos las dos matrices A y B anteriores para aplicar la diferencia entre sus elementos:

```

C = A - B
print(C)
"""
[[ -4 -4]
 [-4 -4]]
"""

D = np.subtract(A, B)
print(D)
"""
[[ -4 -4]
 [-4 -4]]
"""

```

Multiplicación

El producto de una matriz A y una matriz B, produce otra matriz C, siempre y cuando la cantidad de columnas de A sea igual a la cantidad de filas de B. Cada elemento de C en la posición fila i y columna j se obtiene multiplicando cada vector fila i de A con cada columna j de B.

$$C_{ij} = \sum_{k=1}^n a_{ik} * b_{kj}$$

La multiplicación o producto se realiza con el operador (*) o con la función multiply().

```
C = A * B
print(C)
"""
[[ 5 12]
 [21 32]]
"""

D = np.multiply(A, B)
print(D)
"""
[[ 5 12]
 [21 32]]
"""
```

División

Para aplicar una división de arreglos usamos el operador (/) o la función divide().

```
C = A / B
print(C)
"""
[[0.2    0.33333333]
 [0.42857143 0.5    ]]
"""

D = np.divide(A, B)
print(D)
"""
[[0.2    0.33333333]
 [0.42857143 0.5    ]]
"""
```

Producto punto (dot)

El producto punto es una operación algebraica que toma dos arreglos de igual tamaño y retorna un escalar (número simple). Básicamente consiste en la suma de las multiplicaciones de los elementos de los arreglos involucrados.

Supongamos dos vectores, $x = [x_1, x_2, \dots, x_n]$ e $y = [y_1, y_2, \dots, y_n]$ entonces el producto punto de estos vectores sería: $x_1 y_1 + x_2 y_2 + \dots + x_n y_n$

```
x = np.array([1,2])
y = np.array([3,4])
r = np.dot(x,y) # 1x3 + 2x4 = 11
```

Miremos ahora como sería el producto punto de dos matrices I y J considerando el siguiente ejemplo:

$$\begin{array}{c}
 \text{I} \qquad \qquad \text{J} \\
 \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline \end{array} \qquad \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline 5 & 6 \\ \hline \end{array} \\
 R = I \cdot J = \qquad \begin{array}{|c|c|} \hline 22 & 28 \\ \hline 49 & 64 \\ \hline \end{array}
 \end{array}$$

En este ejemplo se suman los productos entre los elementos de la primera fila de la matriz I (1,2,3) con los de la primera columna (1,3,5) de la matriz J y ese valor (22) sería el primer elemento de la matriz R. El segundo valor el 28 se obtiene sumando las multiplicaciones de los elementos de la primera fila de I (1,2,3) con los de la segunda columna de J (2,4,6). El número 49 de la matriz R se obtiene sumando las multiplicaciones de los elementos de la segunda fila de I (4,5,6) con los de la primera columna de J (1,3,5). Y por último el 64 se obtiene sumando las multiplicaciones de los elementos de la segunda fila de I (4,5,6) con los de la segunda columna de J (2,4,6).

Con el anterior ejercicio concluimos esta breve introducción acerca de Numpy, no sin antes recordarle al lector que si desea ampliar mucho más la información aquí expuesta puede consultar la documentación oficial de la librería disponible a través del siguiente enlace: <https://numpy.org/doc/1.21/reference/index.html>

3.3 Creando gráficos con Matplotlib

En el trabajo con aprendizaje automático es bastante común crear gráficos para visualizar el comportamiento de los datos, las relaciones existentes entre ciertas variables o características, y también se pueden usar para saber qué tan bueno es el rendimiento de un modelo, ya que por ejemplo mediante la simple observación de una gráfica podemos tener una idea aproximada de que tan sobreentrenado se encuentra un modelo. De tal manera que, por ser una herramienta de análisis tan importante, dedicaremos algunas líneas a la creación de gráficos usando la librería Matplotlib. Hacemos referencia a una de las librerías más empleadas en la creación de diferentes tipos de gráficos de una manera muy fácil. Sin embargo, por ser este una sección introductoria la información aquí mostrada no abordará todas las potencialidades que ofrece la biblioteca, sino los aspectos que a consideración del autor son los más relevante a fin de lograr que el lector pueda entender de la mejor forma posible los programas de ejemplo que se expondrán en futuras sesiones.

Así como con las otras bibliotecas ya estudiadas, en un principio es necesario hacer una importación, en el caso de Matplotlib este paso se lleva a cabo con una sentencia como la siguiente:

```
import matplotlib.pyplot as plt
```

Veamos algunos gráficos que se pueden crear con Matplotlib.

3.3.1 Gráficos de líneas

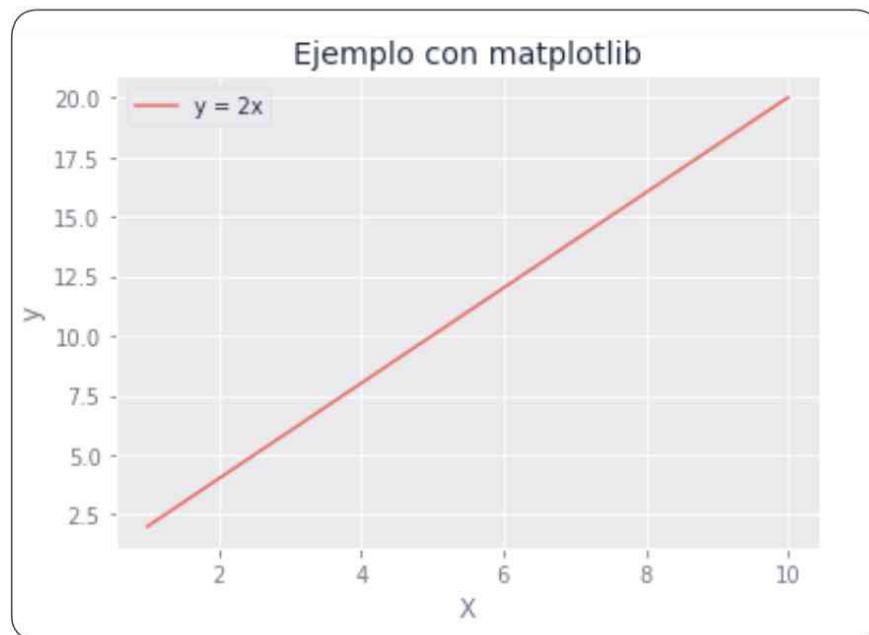
Mediante la función `plot(lista_x, lista_y, label)` podemos crear un gráfico con una línea recta a partir de dos listas pasadas como parámetros que representan las coordenadas *x* e *y* de los puntos de la línea, además un parámetro *label* se aplica para colocarle una etiqueta o leyenda. Consideremos un ejemplo:

```

from matplotlib import style
style.use("ggplot") # se aplica un estilo, puede ver otros con style.available
import matplotlib.pyplot as plt
plt.plot(
    [1,2,3,4,5,6,7,8,9,10],
    [2,4,6,8,10,12,14,16,18,20],
    label="y = 2x"
)
plt.title("Ejemplo con matplotlib") # titulo de la gráfica
plt.xlabel("X") # etiqueta para el eje x
plt.ylabel("y") # etiqueta para el eje y
plt.legend() # permite que se muestre una leyenda

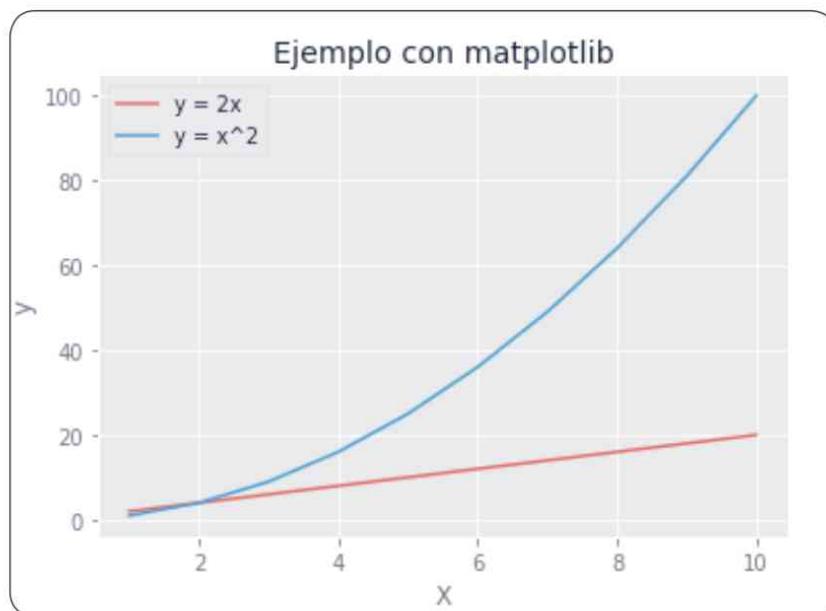
```

El resultado de ejecutar el código anterior será el que se aprecia en la siguiente imagen:



Como es de esperarse con Matplotlib se pueden crear tantos gráficos como se necesiten simplemente agregando funciones `plot()` adicionales. Siguiendo el mismo ejemplo, si queremos dibujar una ecuación cuadrática podríamos colocar un código como el que aparece en negrita (debajo del que ya teníamos), quedando así:

```
plt.plot(  
    [1,2,3,4,5,6,7,8,9,10],  
    [2,4,6,8,10,12,14,16,18,20],  
    label="y = 2x"  
)  
  
plt.plot(  
    [1,2,3,4,5,6,7,8,9,10],  
    [1,4,9,16,25,36,49,64,81,100],  
    label="y = x^2"  
)
```



Para guardar el gráfico anterior recurrimos a la función `savefig()`, la cual admite varios parámetros uno de ellos es la ruta del archivo final. Tal y como se observa en el siguiente fragmento de código:

```
plt.savefig("ejemplo_grafico.png")
```

3.3.2 Gráficos de barras

Matplotlib permite también dibujar gráficos de barras, los cuales son usados normalmente para hacer comparaciones con los datos. La función para esto se llama `bar()` y la podemos usar de la siguiente manera:

```

plt.bar(
    [1,2,3,4,5,6,7,8,9,10],
    [20,40,19,24,35,22,15,27,32,22],
    label = "Datos",
    color = "orange",
    align = "center"
)
plt.title("Resultados")
plt.xlabel("Cursos")
plt.ylabel("Estudiantes")
plt.legend()
plt.grid(True, color="y")

```

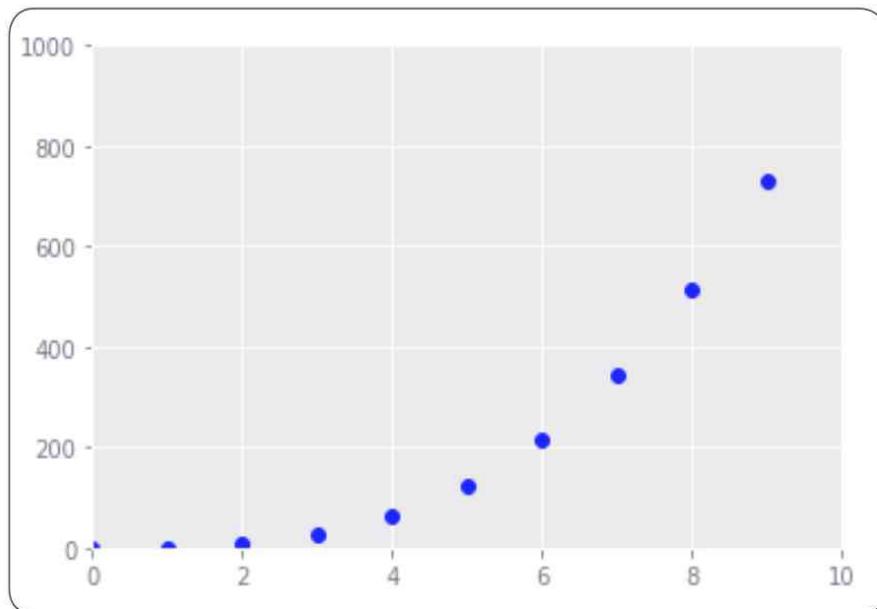


3.3.3 Diagramas de dispersión

Los gráficos de dispersión usan puntos y otras figuras para representar valores de dos variables que se desean analizar. Estos gráficos muestran típicamente la afectación de una variable a causa de un cambio realizado en otra.

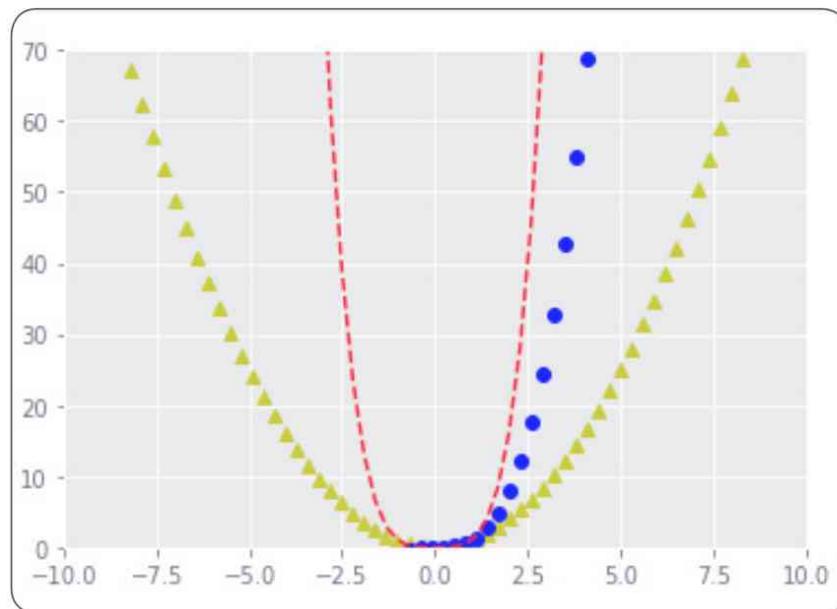
Consideremos este código que crea un sencillo diagrama de dispersión:

```
import numpy as np  
plt.plot(np.arange(0,10),  
         np.arange(0,10)**3,  
         'bo') # marca de circulo azul  
plt.axis([0, 10, 0, 1000]) # xmin, xmax, ymin, ymax  
plt.show()
```



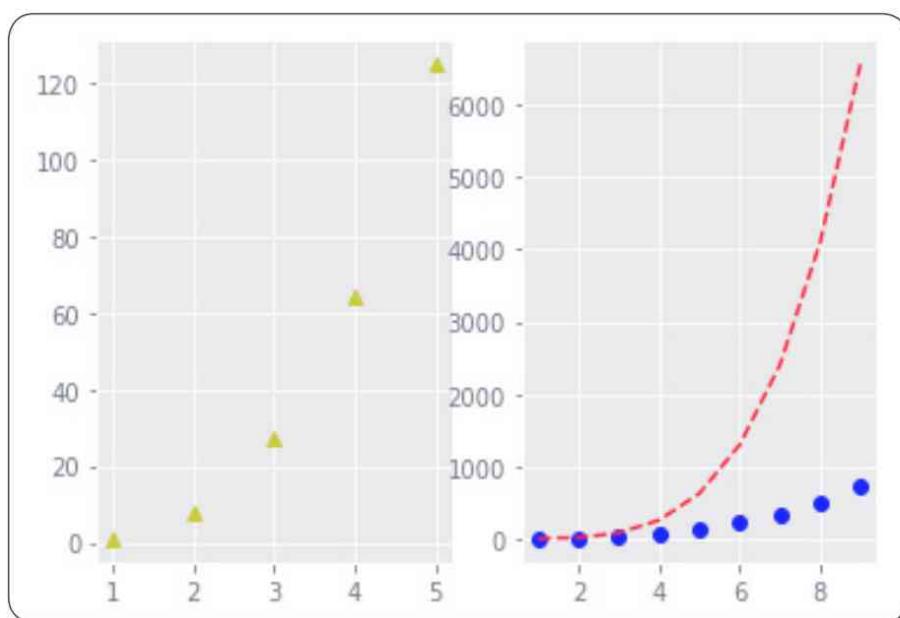
Al igual a como lo hicimos en un ejercicio anterior, podemos crear varias líneas en un mismo gráfico. En el ejemplo que sigue dibujamos tres curvas con puntos de diferente forma y color:

```
import numpy as np  
a = np.arange(-10,10,0.3)  
plt.plot(a, a**2,'y^', # marca triangular  
         a, a**3,'bo', # circulo azul  
         a, a**4,'r--') # linea roja punteada  
plt.axis([-10, 10, 0, 70]) # xmin, xmax, ymin, ymax  
plt.show()
```



Así mismo, la librería nos deja juntar varios diagramas de dispersión en un mismo gráfico con la función subplot(), lo cual es útil en muchos casos principalmente cuando queremos comparar un diagrama con otro, como parte de un análisis sobre el comportamiento de un conjunto de datos.

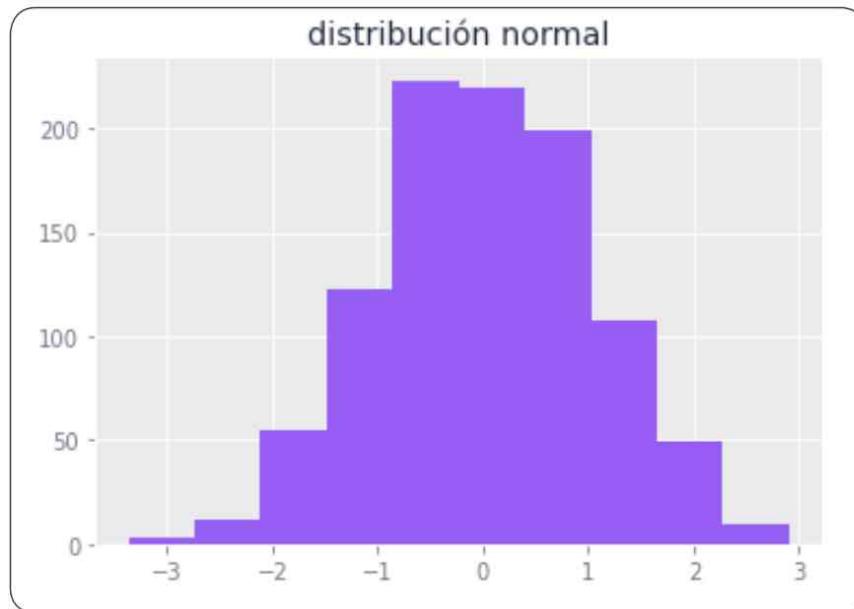
```
import numpy as np
a = np.arange(1,10,1)
plt.subplot(1,2,1) # 1 fila, 2 columnas, primer gráfico
plt.plot([1,2,3,4,5],
          [1,8,27,64,125],
          'y^')
plt.subplot(1,2,2) # 1 fila, 2 columnas, segundo gráfico
plt.plot(a, a**3,'bo',
          a, a**4,'r--')
```



3.3.4. Histogramas

Un histograma se usa básicamente para hacer un resumen de la distribución de un conjunto de datos, donde el eje x representa el o los intervalos en los que se puede encontrar una variable, y el eje y representa la frecuencia de aparición de los valores para cada intervalo de esa variable. Como ejemplo, dibujemos un histograma de una distribución normal con números aleatorios creada con Numpy por medio del método `normal()` del submódulo `random`. Esta función devuelve un ndarray con 1.000 datos con una media de 0 y una desviación estándar de 1. El histograma es creado con la función `hist()` que recibe en su forma básica la lista de valores y la cantidad de intervalos por medio del parámetro `bins`, 10 en este caso. El resultado nos muestra para cada intervalo en el eje x la cantidad de datos coincidentes.

```
import numpy as np
datos = np.random.normal(0,1,1000)
plt.hist(datos, color='#7F38EC', bins=10)
plt.title("distribución normal")
plt.show()
```

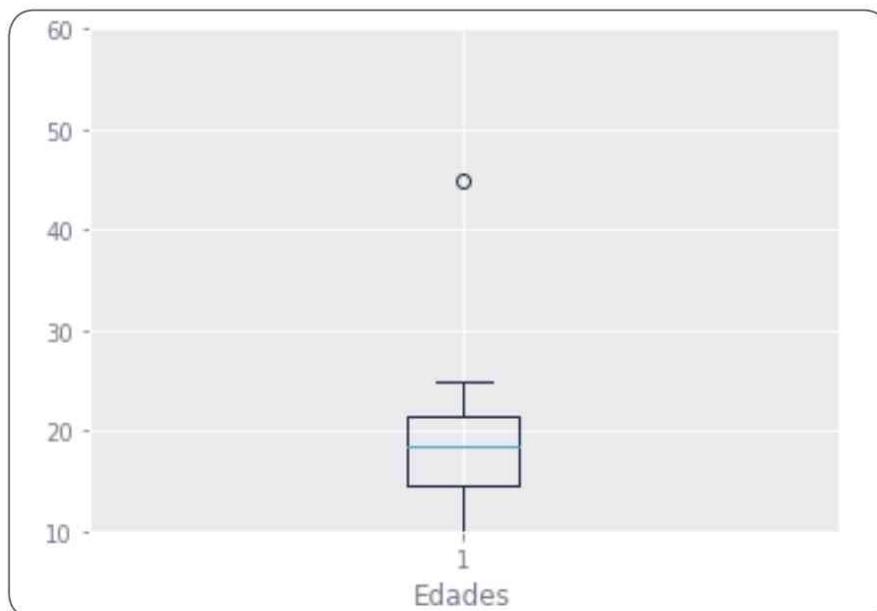


En cuanto al aprendizaje automático la distribución normal goza de una gran popularidad, varios son los algoritmos que necesitan valores normalizados para las características o columnas, como es el caso de las redes neuronales y el algoritmo KNN (K vecinos más cercanos) que estudiaremos en detalle mas adelante. En el primer algoritmo, el hecho de normalizar los datos de entrada a números con media 0 y desviación estándar 1, hace más sencillo el proceso de aprendizaje de los pesos óptimos del modelo, mientras que, en el segundo algoritmo, hace que las variables con valores más altos no dominen a las de valores más pequeños.

3.3.5. Diagrama de caja y bigotes

Por último, miremos este último diagrama usado también cuando se quiere resumir la distribución de una muestra de datos. Nos permiten conocer, por ejemplo: cuartiles, percentiles, valores atípicos, etc.

```
edades = [18, 14, 12, 10, 22, 19, 25, 20, 16, 45]
plt.boxplot(edades)
plt.ylim(10, 60)
plt.xlabel("Edades")
plt.show()
```



El gráfico nos muestra donde se encuentra la mediana de los datos (línea azul) y un valor atípico que corresponde con la edad 45. Este tipo de valores llamados también outliers y otros como los faltantes se dan en muchos casos por un error en la transcripción o una escasa validación de los datos en el diligenciamiento de una encuesta, etc. Por su naturaleza requieren de un tratamiento adecuado para que los modelos de machine learning puedan aprender bien de los datos y logren alcanzar un buen rendimiento, por ello en el próximo capítulo sobre preprocesamiento abordaremos varias técnicas para lidiar con este tipo de situaciones.

3.4 Breve Introducción a Scikit-Learn

Scikit-learn es una biblioteca open source para aprendizaje automático, contiene una gran cantidad de algoritmos de clasificación, regresión, agrupamiento, reducción de la dimensionalidad, selección de modelos y preprocesamiento. La mayoría de estos algoritmos y funciones están agrupados en submódulos dentro del módulo `sklearn`. Así mismo, la librería nos ofrece un conjunto de *datasets* predefinidos que se encuentran dentro del submódulo `dataset`, como también funciones para que nosotros mismos generemos nuestros propios datos, lo cual es muy útil cuando queremos hacer pruebas con los algoritmos antes de ponerlos en marcha con datos reales.

Para ilustrar el uso de Scikit-learn vamos a entrenar un sencillo algoritmo de clasificación utilizando regresión logística sobre el dataset de cáncer de pecho que se puede cargar desde el paquete `datasets` de Scikit-learn. Tenga presente el lector

CAP. 3 - HERRAMIENTAS PARA EL APRENDIZAJE AUTOMÁTICO

que otros aspectos acerca de la regresión y la clasificación serán tratados con más detalle en lo capítulo 5 y 7 respectivamente.

El dataset de tumores de pecho tiene 569 filas y 30 columnas. La variable destino presenta dos posibles valores: benigno o maligno.

El código que sigue comienza principalmente importando y cargando el conjunto de datos por medio de la función `load_breast_cancer()`, luego configura un dataframe con sus respectivas columnas de características y una etiqueta de clase. En consecuencia, obtiene los valores para X y para Y, a partir de los cuales divide los datos con la función `train_test_split()` en dos subconjuntos: entrenamiento y pruebas. Este último se utiliza fundamentalmente para testear el modelo, puesto que una regla de oro en aprendizaje automático es realizar las pruebas con datos diferentes a los de entrenamiento. Más adelante se crea una instancia de la clase `LogisticRegression`, dicha instancia llama al método `fit()`, usado para ajustar el modelo con los datos de entrenamiento y estimar algunos parámetros para el mismo. Por último, se hace la evaluación con la métrica `accuracy_score`, localizada dentro del submódulo `metrics`, la cual toma como parámetros un vector con etiquetas de prueba(y) y otro con las etiquetas predichas por el modelo a través de la función `predict()`. La mayoría de métricas se basan en este par de vectores para medir la calidad de un modelo de machine learning, cuyo valor generado se puede encontrar en el rango entre 0 y 1. En este sentido, entre más cercano a 1 mejor rendimiento tendría nuestro modelo.

El proceso ilustrado en el ejemplo anterior es el que se suele hacer de forma casi que rutinaria en machine learning e involucra las fases esenciales ya mencionadas en el capítulo 2: Preprocesado, separación del dataset en conjuntos de entrenamiento y pruebas, configuración del algoritmo, entrenamiento, predicción y evaluación.

```
import pandas as pd
# se importa y se carga el dataset
from sklearn.datasets import load_breast_cancer
dataset = load_breast_cancer()
# se convierte a un dataframe de pandas
df = pd.DataFrame(dataset.data, columns = dataset.feature_names)
df['tipo'] = dataset.target[df.index]
# se obtienen los valores para las variables X e y
X = df.iloc[:, :-1]
y = df['tipo'].values

# se separan los datos en conjuntos de prueba y entrenamiento
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
# se crea una instancia de la clase LogisticRegression
from sklearn.linear_model import LogisticRegression
reg = LogisticRegression(max_iter=10000)
# se entrena el modelo con los datos de entrenamiento
reg.fit(X_train,y_train)

from sklearn.metrics import accuracy_score
print("Exactitud {:.2f}%".format(accuracy_score(y_test, reg.predict(X_test))))
```

Exactitud 0.96

De todas formas, si este ejemplo a resultado ambiguo para el lector en los próximos capítulos se irán abordando todos estos aspectos de manera más pausada y con más nivel de detalle a fin de brindar más claridad sobre conceptos, técnicas, algoritmos y funciones de aprendizaje automático implementados con la librería Scikit-learn.

Con este ejemplo damos por terminado nuestra introducción a las herramientas necesarias para iniciar nuestro estudio de aprendizaje automático, que, si bien no son todas las que usaremos, si son las que deben conocerse tan si quiera de manera básica para poder comprender sin mayores complicaciones los temas a explicarse más adelante.

CAPÍTULO 4

Preprocesado de datos

Temas

- 4.1 ¿Qué es preprocesado de datos?
- 4.2 Creación de conjunto de entrenamiento y pruebas
- 4.3 Manejo de datos ausentes
- 4.4 Manejo de datos categóricos
- 4.5 Escalamiento de características

4.1 ¿Qué es el preprocesado de datos?

En muchos casos los conjuntos de datos que recopilamos por ejemplo a través de una encuesta en línea, presentan ausencia o inconsistencia de algunos datos siendo las causas de esto muy diversas: Pueden ir desde incompletitud al momento de suministrar la información, en el caso de una encuesta, o hasta por un error al momento de la captura de los datos. Este hecho si no se atiende debidamente puede ocasionar errores o resultados no deseados en los modelos de aprendizaje automático, de allí la importancia de esta disciplina en los procesos de aprendizaje automático.

En este apartado estudiaremos justamente algunos aspectos y técnicas relevantes relacionadas con el preprocesamiento de datos. Pero antes de hacerlo, haremos mención de las funciones `fit()` y `transform()` que comparten muchas clases que implementan algoritmos de *machine learning* dentro de la biblioteca Scikit-learn.

En todos los algoritmos que vamos a comenzar a estudiar desde ahora, el modo de operar después de efectivamente haber cargado el dataset, es el siguiente: 1) instanciamos un objeto de alguna clase de interés con los argumentos respectivos; 2) con el objeto instanciado se invoca al método `fit()`, el cual se ajusta con los datos del subconjunto de entrenamiento y estima valores para los parámetros internos

del modelo; 3) luego se llama al método transform() con el cual se llevan a cabo las transformaciones requeridas sobre los datos de entrada. Estas transformaciones pueden ser por ejemplo escalado de características, imputación de valores ausentes, etc. Scikit-learn nos ofrece un método fit_transform() para efectuar estas dos tareas de ajustar y transformar el objeto en un solo paso.

Cabe anotar, que el método fit() además de usarse con algoritmos de preprocessamiento también se puede emplear con los algoritmos típicos de machine learning, como el de Regresión Logística (LogisticRegression) expuesto en el subpartido sobre introducción a Scikit-learn del capítulo 3. Más adelante en el libro notaremos que para algoritmos supervisados a fit() le pasamos como parámetros los conjuntos X y Y de entrenamiento, mientras que para algoritmos no supervisados solo le pasamos X. Siendo X una matriz con los atributos descriptivos y Y un arreglo con las etiquetas de clase.

4.2 Creación de conjunto de entrenamiento y pruebas

Cuando un conjunto de datos es lo suficientemente grande se puede separar en un conjunto de entrenamiento usado para entrenar el modelo de aprendizaje, y un conjunto de prueba para validar la calidad de dicho modelo. La separación de las instancias generalmente es realizada de manera aleatoria, considerando un 70% u 80% del conjunto original para conformar el conjunto de entrenamiento y el resto para el conjunto de pruebas.

Con la biblioteca Scikit-learn esto puede ser logrado a través de la función train_test_split() del submódulo model_selection.

```
import pandas as pd
from sklearn.model_selection import train_test_split
df = pd.read_csv("precios_casas.csv")
X = df.iloc[:,1:].values
y = df.iloc[:,0].values
print(X)
entX, pruX, enty, pruy = train_test_split(X, y, test_size=0.2, random_state=100)
```

En este código básicamente se recuperan los valores de los atributos descriptivos y se almacenan en X. Mientras que los valores de la etiqueta de clase se almacenan en y. Ambos tanto X como y son arreglos de numpy. El primero es una matriz de dos dimensiones con 4.600 filas y 16 columnas y el segundo es un vector de 4.600 elementos.

La función `train_test_split()` recibe como parámetros los conjuntos X e y junto con el tamaño del conjunto de prueba (`test_size=0.2`), es decir 20%, devolviendo los siguientes 4 arreglos de numpy:

- `entX`: Conjunto de entrenamiento con atributos descriptivos: 3.680 filas y 16 columnas.
- `enty`: Conjunto de entrenamiento con la etiqueta de clase: 3.680 etiquetas.
- `pruX`: Conjunto de prueba con los atributos descriptivos: 920 filas y 16 columnas.
- `pruY`: Conjunto de prueba con la etiqueta de clase: 920 etiquetas.

El parámetro `random_state=100`, se usa como semilla para inicializar un objeto generador de números aleatorios interno, utilizado por numpy para realizar la separación, además con esta configuración garantizamos que aún cuando se ejecute el programa varias veces se producirán los mismos conjuntos de puntos para *train* y *test*. Esto es útil en ciertos casos, por ejemplo, cuando se tiene un error y se quiere hacer una depuración (debug), entonces se necesita reproducir varias veces el problema con los mismos datos generados en un principio, con el fin de verificar si la solución propuesta es la adecuada.

Finalmente después de llamar a `train_test_split()`, ya tendríamos nuestros datos debidamente separados (80% para entrenamiento y 20% para pruebas) y así poder aplicar las diferentes técnicas de preprocesamiento. Como ya se ha mencionado en el libro, se puede usar un valor diferente para hacer la separación como 90%-10%, pero las más usadas son 80%-20% y 70%-30%, en todo caso, se debe tener en cuenta siempre la cantidad de datos disponibles, dado que por ejemplo en un dataset de millones de registros quizás un 99%-1% podría ser una buena medida de separación.

4.3 Manejo de datos ausentes

Es muy común que después de recopilar datos para realizar una actividad de aprendizaje automático nos demos cuenta que faltan algunos de ellos. Esta es una situación a la cual hay que prestarle mucha atención debido a que la ausencia de datos puede provocar problemas a los algoritmos utilizados, dado que estos últimos son muy susceptibles a los faltantes a tal punto que les hacen producir resultados inesperados. La ausencia de un valor en Python se representa con un `NaN` (Not a Number).

Para observar esta situación vamos a crear un conjunto de datos nuevo, ya que el de los precios de las casas viene por defecto debidamente preprocesado y no tiene valores faltantes. Esto lo puede comprobar el lector escribiendo la siguiente instrucción, esta nos devuelve la suma de valores ausentes por cada columna:

```
df.isnull().sum()
```

Siguiendo con el ejemplo, el dataframe que usaremos será el que sigue:

```
df = pd.DataFrame([
    ['1', 1, 30],
    ['2', 1, 32],
    ['3', 0]],
)
df.columns = ['codigo','credito','edad']
```

Como se puede apreciar falta un valor en la columna edad de la última fila. Para paliar esta situación existen dos maneras posibles: La primera es eliminar del conjunto aquellas filas o columnas donde se encuentren los valores ausentes, mediante la función dropna(), que en su forma básica permite:

- Eliminar las filas donde hay valores ausentes, así:
df.dropna(axis=0), esta es la opción por defecto.
- Eliminar las columnas donde hay valores ausentes en alguna fila, así:
df.dropna(axis=1)

Aquí, axis=0 se refiere a filas y axis=1 a columnas.

La función dropna() maneja otros parámetros interesantes que permiten establecer cierto control sobre el borrado de filas o columnas: *how={any, all}*, con *any* borra si hay al menos un valor NaN, en cambio con *all* hace el borrado si todos los valores son NaN; *thresh=valor*, donde *valor* es un número que funciona como umbral con el cual se establece el número máximo de NaN que debe superarse para realizar el borrado; *subset=[lista_nombres_columnas]* que permite borrar las filas siempre y cuando las columnas cuyos nombres se indican en la lista tengan NaN.

Sin embargo, este enfoque no se recomienda demasiado puesto que, se pueden perder datos en demasía o datos que podrían resultar útiles para el entrenamiento del algoritmo de aprendizaje automático. Por lo tanto, el mejor método en este sentido es el conocido como imputación de valores ausentes, encontrándose accesible su implementación en *Scikit-learn* a través de la clase *SimpleImputer* del submódulo *sklearn.imputer*. Al instanciar un objeto, en el constructor se puede especificar una estrategia, como la de imputación por medias (*strategy='mean'*), la cual consiste en reemplazar el valor ausente con la media aritmética de toda

la columna. Otras opciones para *strategy* son ‘median’ para sustituir el valor que falta por la mediana, ‘most-frequent’ para reemplazar por el valor más frecuente (la moda de los datos) y ‘constant’ para cambiar por un valor fijo o constante.

Otro punto importante a considerar, es que hay filas que tienen columnas con algunos caracteres especiales como por ejemplo símbolos de interrogación (?), entonces en estos casos antes de realizar la tarea de imputación se recomienda primero reemplazar el simbolo por un NaN mediante la función `replace()`.

```
df.replace('?', np.nan, inplace=True)
```

Una vez se haya realizado un reemplazo como el anterior se realiza la imputación utilizando el siguiente fragmento de código:

```
from sklearn.impute import SimpleImputer
imp = SimpleImputer(missing_values=np.nan, strategy='mean')
imp = imp.fit(df.values)
imp_dados = imp.transform(df.values)
```

```
[[ 1.  1. 30.]
 [ 2.  1. 32.]
 [ 3.  0. 31.]]
```

Si imprimimos la variable *imp_dados* veremos como el valor faltante se ha reemplazado por el número 31, que es la media aritmética de los datos de la tercera columna: $(30 + 32) / 2 = 31$.

4.4 Manejo de datos categóricos

Los algoritmos de aprendizaje automático están diseñados para operar con datos numéricos, por lo cual se recomienda convertir a números todos aquellos datos categóricos. En este sentido, algo estrictamente necesario es codificar la etiqueta de clase del conjunto de datos a un valor numérico. Esto se consigue con la clase `LabelEncoder`. Miremos como hacer esto tomando en consideración el siguiente dataframe:

```
df = pd.DataFrame([
    ['M', 30, 'Amarillo', 'Clase 1'],
    ['P', 28, 'Azul', 'Clase 2'],
    ['J', 21, 'Rojo', 'Clase 1']],
)
df.columns = ['nombre', 'edad', 'color', 'etiqueta']
dfAux = df
```

Convertiremos ahora los valores de la columna *etiqueta* a su representación numérica:

```
from sklearn.preprocessing import LabelEncoder
le_clase = LabelEncoder()
y = le_clase.fit_transform(df.etiqueta)
print(y)
```

Donde *y* sería un vector con los valores [0 1 0], que significa que "Clase 1" se codificó como 0 mientras que "Clase 2" como 1 resultado del ajuste y la transformación realizada con el método *fit_transform()*. Desde luego, es posible obtener la representación original ['Clase 1' 'Clase 2' 'Clase 1'], mediante el método *inverse_transform()*:

```
clase_inv = le_clase.inverse_transform(y)
print(clase_inv)
```

No obstante, la técnica anterior no es la más adecuada, ya que, si por ejemplo tenemos una variable llamada *profesión* con los valores ingeniero, arquitecto y contador, codificada como 1,2 y 3 respectivamente, un algoritmo de aprendizaje podría inferir que contador es mayor a arquitecto o que arquitecto es mayor que ingeniero, lo cual no es correcto puesto que no se trata de una variable categórica ordinal.

Para mejorar esta situación hay otra técnica muy utilizada denominada **one hot encoding** (Una codificación en caliente), con ella un atributo categórico nominal de *n* valores posibles generará *n* columnas binarias, donde se coloca un 1 en la columna coincidente con el valor que tiene la instancia para ese atributo y el resto de columnas referentes a ese mismo atributo se colocan en 0. Por ejemplo, supongamos un conjunto de datos de vehículos con un atributo *color*, con tres posibles valores: amarillo, azul y rojo. Entonces el atributo *color=azul* de una instancia cualquiera se representaría como: 0,1,0.

Para que la codificación en caliente funcione la entrada recibida debe ser numérica, por lo que usaremos la clase `LabelEncoder` vista hace un momento para realizar esta transformación. Creamos un objeto de la mencionada clase y codificamos la columna `color` a un consecutivo comenzando desde 0, cuyos valores los guardaremos en una nueva columna del dataframe llamada `color_cod`.

```
from sklearn.preprocessing import OneHotEncoder
le_color = LabelEncoder()
ohe_color = OneHotEncoder(categories='auto')
df['color_cod'] = le_color.fit_transform(df.color)
print(df)
```

	nombre	edad	color	etiqueta	color_cod
0	M	30	Amarillo	Clase 1	0
1	P	28	Azul	Clase 2	1
2	J	21	Rojo	Clase 1	2

Posteriormente transformamos la característica `color` mediante el objeto `ohe_color`, cuyo método `fit_transform()` espera un arreglo 2D (bidimensional), por lo que debemos redimensionarlo de 1D a 2D. Además este método devuelve una matriz dispersa, entonces usamos el método `toArray()` para convertirla a una matriz de Numpy. Seguidamente generamos otro dataframe con las nuevas columnas generadas y las concatenamos con nuestro dataframe inicial mediante la función `concat()`.

```
datos_ohe      =      ohe_color.fit_transform(df.color_cod.values.reshape(-1,1)).toarray()
dfOneHot = pd.DataFrame(datos_ohe, columns = ["Color_"+str(int(i)) for i in range(len(df.color))])
df = pd.concat([df, dfOneHot], axis=1)
print(df)
```

	nombre	edad	color	etiqueta	color_cod	Color_0	Color_1	Color_2
0	M	30	Amarillo	Clase 1	0	1.0	0.0	0.0
1	P	28	Azul	Clase 2	1	0.0	1.0	0.0
2	J	21	Rojo	Clase 1	2	0.0	0.0	1.0

Finalmente podemos eliminar las columnas `color` y `color_cod` del dataframe porque sencillamente ya no las necesitamos, mediante la siguiente sentencia:

```
df= df.drop(['color','color_cod'], axis=1)
```

Existe otra función llamada `get_dummies()` de la librería Pandas que también convierte variables categóricas a variables indicadoras (o dummies) y es mucho más apropiada cuando tienen una gran cantidad de valores.

En la siguiente sentencia invocamos a `get_dummies()` con los parámetros: `data`, `columns` y `drop_first`, indicando en ese mismo orden: el dataframe, el nombre de las columnas a ser codificadas y la opción de borrar la(s) columna(s) objetivo de la codificación (`color`) y la primera columna (`color_amarillo`). Esta operación de ninguna manera significa perdida de información ya que, si tenemos para una muestra cualquiera, por ejemplo, en la columna `color_azul` un 0 y en la columna `color_rojo` otro 0, significa que el color para esa muestra es el amarillo. Por otra parte esta característica de la función `get_dummies()` hace que los dataframes no crezcan tanto en situaciones donde hay características que tienen muchos valores, hecho muy común cuando se usa one hot encoding, siendo esta sin duda su principal desventaja.

```
pd.get_dummies(data=dfAux, columns=["color"], drop_first=True)
df= df.drop(['color_cod'], axis=1)
```

	nombre	edad	color	etiqueta	color_cod
0	M	30	Amarillo	Clase 1	0
1	P	28	Azul	Clase 2	1
2	J	21	Rojo	Clase 1	2

4.5 Escalamiento de características

El escalamiento es otra de las tareas importantes dentro del preprocesado de datos, debido a que la gran mayoría de los algoritmos de aprendizaje automático funcionan mejor si se tienen las características de un conjunto de datos bajo la misma escala. Adicionalmente, es necesario usar la misma escala en el conjunto de entrenamiento y de prueba a fin de no terminar provocando un sesgo aleatorio en los datos.

Como ejemplo escalaremos el atributo `precio` del dataframe de los precios de las casas en un rango entre 0 y 1, lo haremos usando la técnica de la **normalización** que le aplicaremos a la variable por medio de la siguiente función:

$$x_{normalizada} = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

Donde, x_i corresponde a los precios de la columna *price*, x_{min} es el valor más pequeño y x_{max} es el valor más grande de la misma. La anterior ecuación la podemos programar en Python de forma sencilla aprovechando la vectorización, característica para mejorar el rendimiento computacional evitando en la medida de lo posible el uso de ciclos en la realización de operaciones con vectores:

```
def normalizar(columna):
    x_norma = (columna - columna.min()) / (columna.max() - columna.min())
    return x_norma
```

Sin embargo, no tendremos que preocuparnos por implementar la función anterior puesto que ya está implementada en la clase *MinMaxScaler* de Scikit-learn.

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
df = pd.read_csv("precios_casas.csv")
esc = MinMaxScaler()
X_ent = esc.fit_transform(df.price.values.reshape(-1,1))
print(X_ent)
```

El resultado es un array de numpy con los precios en una escala entre 0 y 1.

```
[[0.01177134]
 [0.08965777]
 [0.01286198]
 ...
 [0.01567898]
 [0.00764949]
 [0.00829635]]
```

Otra técnica usada también para el escalado es la **estandarización**. La ecuación para aplicar este procedimiento es:

$$x_{estandarizada} = \frac{x_i - \mu}{\sigma}$$

Siendo, μ la media de los valores de la columna y σ la desviación estándar de la misma. Podríamos expresar la ecuación de la estandarización en Python así:

```
def estandarizar(columna):
    x_estandar = (columna - columna.mean()) / columna.std()
    return x_estandar
```

Pero, de igual forma como sucede con la normalización, Scikit-learn dispone de una clase específica para estandarizar características, y es la clase StandardScaler. Retomemos el ejemplo anterior para estandarizar la variable *precio*.

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
df = pd.read_csv("precios_casas.csv")
est = StandardScaler()
X_ent = est.fit_transform(df.price.values.reshape(-1,1))
print(X_ent)
```

```
[[ -0.42386353]
 [ 3.2495981 ]
 [-0.37242442]
 ...
 [-0.23956224]
 [-0.61826787]
 [-0.58775916]]
```

Cabe mencionar que las técnicas de procesamiento explicadas anteriormente, tal y como se expresó al principio necesitan recibir como entrada datos numéricos para operar.

Lo explicado en esta sección pretende servir de introducción a la temática en cuestión, por tanto, se recomienda al lector profundizar más sobre la fase de preprocesamiento de datos, debido a que es una de etapas que más importancia reviste en cualquier actividad de aprendizaje automático. En los próximos capítulos comenzaremos a estudiar los principales algoritmos de aprendizaje automático supervisado, comenzando con el análisis de regresión.

CAPÍTULO 5

Modelos de regresión

Temas

- 5.1 Visualización de la relación entre características
- 5.2 Solución mediante el enfoque de mínimos cuadrados
- 5.3 Descenso del gradiente
- 5.4 Regresión lineal mediante Scikit-learn
- 5.5 Regresión con Random Sample Consensus (RAMSAC)
- 5.6 Regresión lineal polinómica
- 5.7 Regresión lineal múltiple en notación matricial
- 5.8 Modelos no lineales

La regresión es un subcampo del aprendizaje automático cuyo propósito es predecir valores continuos a partir de ciertas entradas o variables descriptivas. Existen dos tipos de regresión: Lineal pudiendo ser simple y múltiple y No lineal. En este primer apartado centraremos nuestra atención en la regresión lineal, la cual permite representar los datos mediante una línea recta, y dejaremos para el final la no lineal que como el lector podrá intuir emplea una línea de tendencia polinómica para ajustar los datos.

La regresión es usada para resolver muchos problemas de la vida real, por ejemplo, predecir el precio de una casa a partir de su área en metros cuadrados, número de cuartos, etc; predecir el número de ventas de un producto basado en la publicidad gastada o en el número de consultas realizadas por los clientes sobre ese producto. Como se podrá dar cuenta el lector los ejemplos anteriores buscan determinar un valor continuo, a diferencia de la técnica de clasificación que se estudiará en el capítulo 7, cuya variable de destino es discreta.

La regresión lineal es aplicable si los datos disponibles se pueden representar mediante una línea, es decir, al graficarlos tienen un comportamiento lineal. Si contamos con una sola variable descriptiva estamos ante una regresión lineal simple. Nuestros conocimientos en matemáticas nos permiten deducir que la ecuación adecuada para este caso es:

$$y = w_0 + w_1 x$$

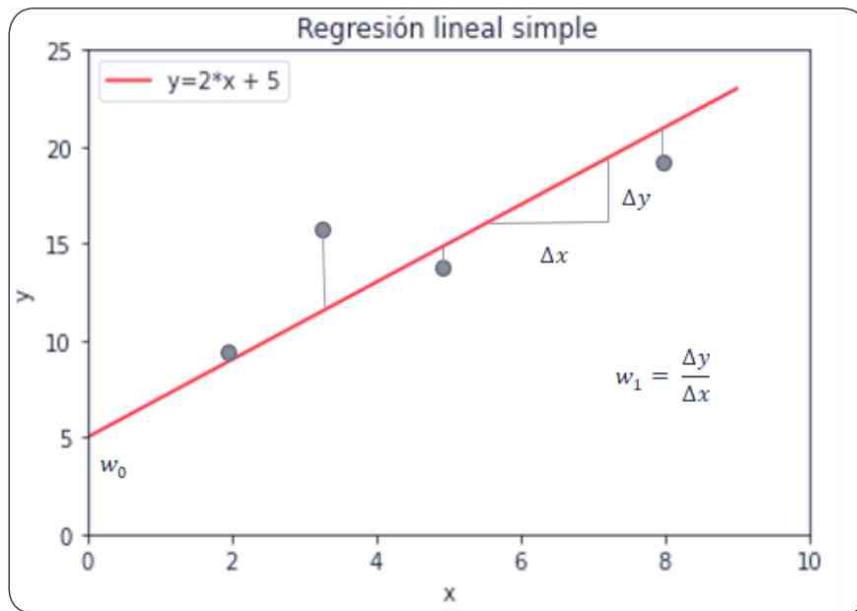
Esta es la ecuación de la línea recta, donde x es la variable independiente e y la variable dependiente, con w_0 y w_1 , coeficientes o parámetros que debemos encontrar. La constante w_0 se suele denominar también *intercepción* con el eje y , y la constante w_1 se denomina *gradiente* o *pendiente de la línea*. Las ecuaciones como esta usadas para definir un modelo de aprendizaje automático comunmente reciben el nombre de hipótesis.

El resultado de entrenar un modelo de regresión lineal con una hipótesis como la anterior, consiste en hallar los valores adecuados de w_0 y w_1 , de tal manera que se consiga la línea que más cerca pase por los puntos de datos. Los valores óptimos de w_0 y w_1 se logran después de realizado un proceso iterativo de entrenamiento, el cual implica resolver un problema de optimización por medio del que se busca minimizar el error en una función de coste. En cada iteración se van calculando nuevos valores para w_0 y w_1 . Estos últimos valores multiplicados por los puntos de datos (x) permiten obtener los correspondientes valores de y predicha (\hat{y}). El error o residuo es el valor absoluto de la diferencia entre la y del conjunto de datos (y real) y la y predicha. El algoritmo de optimización debe ir ajustando los valores w_0 y w_1 , a tal punto de conseguir que dicho error sea lo más cercano a 0 para poder considerar bueno nuestro modelo de regresión.

La gráfica que aparece a continuación ilustra la regresión lineal simple y en ella se puede ver que los valores de w_0 y w_1 , son 5 y 2 respectivamente. Si consideramos por ejemplo que x es la variable "número de habitaciones" y que y es la variable "precio", asumiendo que estamos trabajando un modelo para predecir el precio de una vivienda a partir del número de cuartos, tendríamos la ecuación: $y = 2x + 5$, para predecir cuánto cuesta una casa con 5 habitaciones, entonces calculamos y predicha (\hat{y}), $\hat{y} = 2(5) + 5 = 15$.

Una vez comprendida la regresión simple, ya podemos ver el caso más general, la regresión múltiple o multivariable, la cual se expresa matemáticamente mediante la siguiente ecuación:

$$y = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n$$



El objetivo en la regresión múltiple es hallar los números óptimos de $w_0, w_1, w_2, \dots, w_n$, que al ser multiplicados por $1, x_1, x_2, \dots, x_n$ permiten obtener un valor para la variable y predicha (\hat{y}). Tal y como se mencionó, los valores óptimos de $w_0, w_1, w_2, \dots, w_n$ son aquellos que minimizan la diferencia de la variable \hat{y} y la variable y del conjunto de datos, dicha resta se conoce como residuo. Por consiguiente, podemos deducir, que, hallando los anteriores pesos, estaremos encontrando la recta que mejor se ajusta a los datos de entrada.

Para entender un poco más lo expresado hasta aquí, vamos a analizar un ejemplo utilizando regresión lineal simple a partir del conjunto de datos Boston housing visto en el capítulo 3. Más adelante extenderemos el ejemplo usando regresión lineal múltiple que implementaremos con la librería Scikit-learn. Tenga presente que los métodos y técnicas que estudiaremos en los siguientes subapartados son aplicados en un ejemplo de regresión lineal simple o univariable, sin embargo, los mismos conceptos pueden usarse sin ningún inconveniente en una regresión lineal con múltiples variables descriptivas.

5.1 Visualización de la relación entre características del conjunto de datos

Una buena estrategia cuando se trabaja con la regresión es hallar la relación existente entre las variables descriptivas y la de destino, a fin de conocer qué características muestran un comportamiento lineal con respecto a la variable

objetivo. En este subapartado miraremos si hay relación entre 5 características del conjunto de datos Boston housing y luego mediante una matriz de dispersión observaremos la relación existente entre parejas de tales variables a partir de la distribución de sus datos. Luego, tras la inspección hecha sobre esta matriz seleccionaremos aquellas variables que tienen una representación lineal.

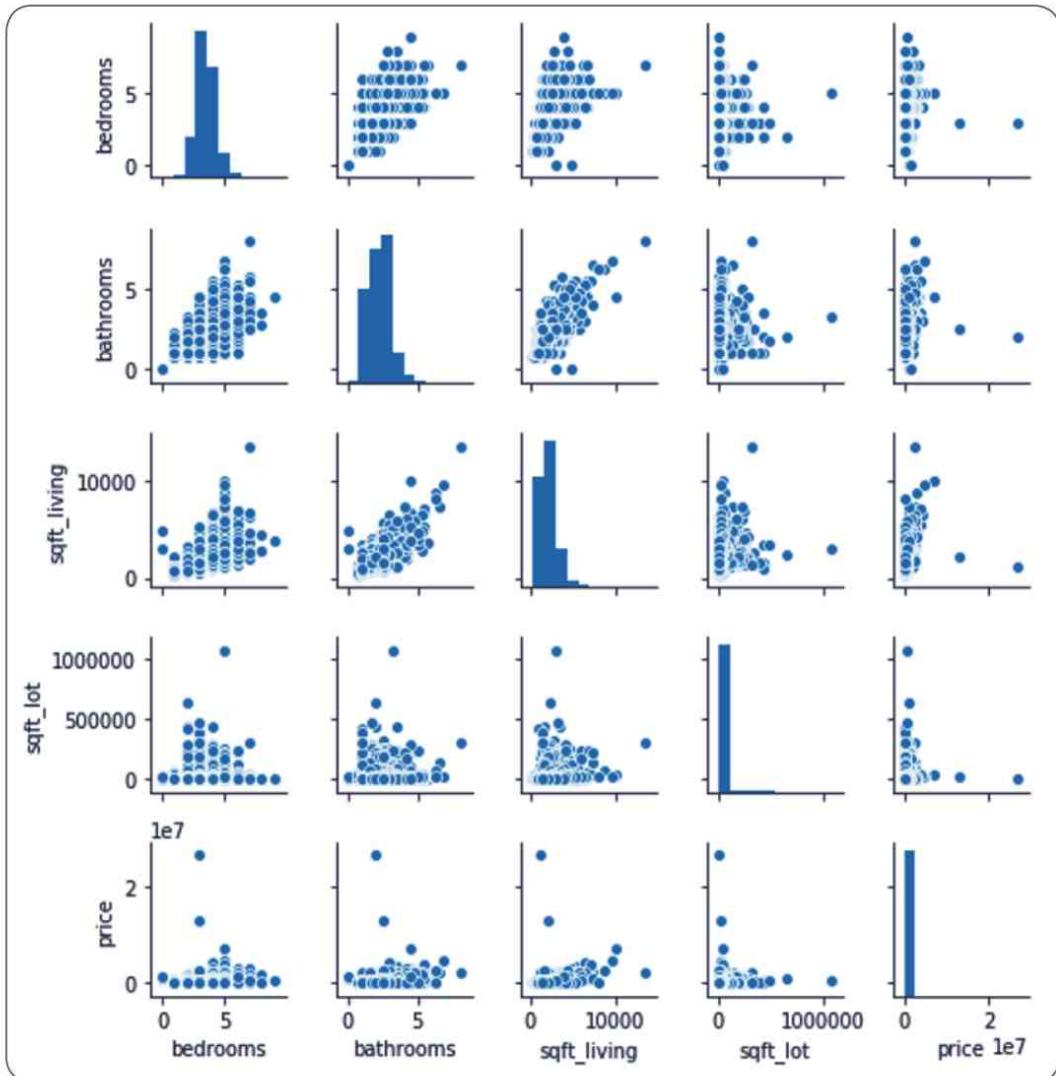
Para crear la matriz de dispersión usaremos una librería llamada Seaborn de Python. La cual se puede instalar mediante la orden: *pip install seaborn*.

Con el siguiente código podemos dibujar la matriz en cuestión:

```
import pandas as pd
import seaborn as sb
import matplotlib.pyplot as pl
df = pd.read_csv("precios_casas.csv")
cols = ['bedrooms','bathrooms','sqft_living','sqft_lot','price']
sb.pairplot(df[cols] , size=2.5)
pl.tight_layout()
pl.show()
```

El resultado nos muestra las relaciones entre las columnas *bedrooms*, *bathrooms*, *sqft_living*, *sqft_lot* y *price* que aparecen dispuestas a lo largo de los ejes x e y, adicionalmente podemos notar que hay una relación lineal entre las características *sqtf_living* y *price*, así como también entre *bathrooms* y *price*. Esta relación lineal de *sqtf_living* y *price* (fila 5 columna 3) tiene mucho sentido, puesto que en la vida real conforme aumenta el valor del área de la sala de una casa mayor también es el precio de la misma. En situaciones como esta, en donde si x crece y también lo hace, decimos que hay una correlación positiva.

La siguiente gráfica es una de las herramientas más usadas para el análisis de relación entre variables, y está basada en dos ejes: El eje x representa la característica descriptiva, mientras el eje y representa la característica destino. Cada instancia del conjunto de datos es representada mediante un punto en la gráfica. Sin embargo, es común encontrar en los conjuntos de datos, lo que se conoce como outliers o valores extremos, que en el presente ejemplo son los puntos que se encuentran muy alejados de la posible línea recta de mejor ajuste de los datos.



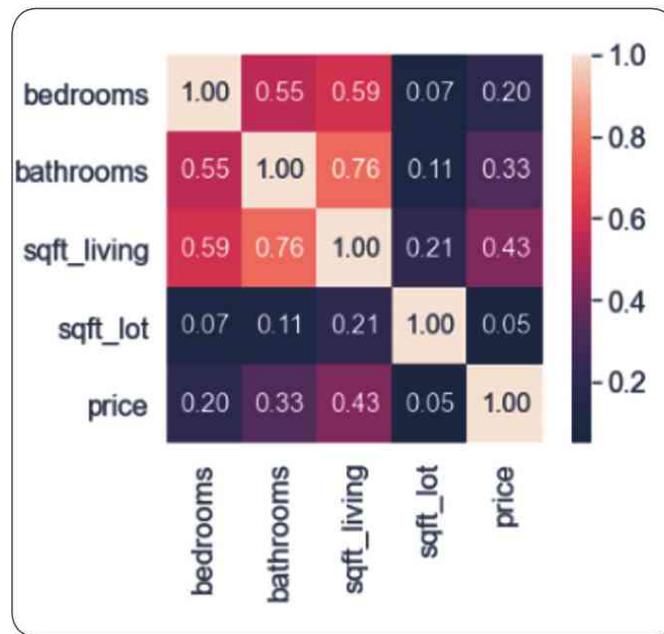
Otra técnica de análisis muy usada también, es la matriz de correlación, con la que podemos obtener el coeficiente de **correlación de Pearson**. Por medio del cual es posible medir numéricamente la relación lineal existente entre pares de características. Un coeficiente cercano a 1, indica una buena correlación, mientras que 0 refleja la ausencia de correlación entre las variables.

La librería NumPy tiene una función llamada `corrcoef()` que usaremos para encontrar el coeficiente de correlación mencionado, así mismo usaremos la función `heatmap()` de Seaborn para dibujar la matriz de correlación mediante un mapa de calor.

```

df = pd.read_csv("precios_casas.csv")
cols = ['bedrooms','bathrooms','sqft_living','sqft_lot','price']
cm = np.corrcoef(df[cols].values.T)
sb.set(font_scale=1.5)
hm = sb.heatmap(cm, cbar=True,
                 annot=True,
                 square=True,
                 fmt='2f',
                 annot_kws={'size':15},
                 yticklabels=cols,
                 xticklabels=cols
                )
pl.show()

```



De acuerdo a la siguiente gráfica notamos que la variable *sqft_living* presenta una correlación con la variable *price* de 0.43, lo cual confirma la buena relación lineal entre estas variables, aspecto ya descubierto en el análisis de la matriz de dispersión. Este resultado sugiere que estas variables se pueden usar en un modelo de regresión lineal simple, expresado matemáticamente así:

$$\text{precio} = w_0 + w_1 * \text{sqft_living}$$

5.2 Solución mediante el enfoque de mínimos cuadrados

Habiendo ya escogido las variables descriptivas (X) que mejor se relacionan con la variable dependiente (y) y conociendo también como es la distribución de los datos en un plano cartesiano bidimensional, el siguiente paso es encontrar una recta tal, que permita representar la mayor cantidad posible de puntos de datos. Esta recta de mejor ajuste definida por la ecuación de la recta, es aquella que pase lo mas cerca posible a dichos puntos. Para ello, como ya se dijo es necesario encontrar los valores óptimos de w_0 y w_1 . Las dos técnicas que usaremos para encontrar w_0 y w_1 son: Mínimos cuadrados ordinarios y el descenso del gradiente.

El enfoque de mínimos cuadrados consiste en hacer la diferencia entre las predicciones hechas por el modelo para cada valor de x del conjunto de entrenamiento y los valores de y actuales del mismo conjunto. Como el error o residuo puede resultar con valores negativos se eleva la diferencia al cuadrado, luego se realiza la sumatoria de todos los errores y finalmente se divide la suma obtenida entre el total de muestras. A este valor se le denomina error cuadrático medio MSE por sus siglas en inglés, y entre más pequeño sea este término mejor ajustado estará el modelo.

A manera de ejemplo, supongamos que $w_0 = 2$ y $w_1 = 1$, quedando $y = x + 2$. Tomemos las 10 primeras muestras de nuestro conjunto de entrenamiento, las cuales han sido normalizadas junto con la variable objetivo. Calculemos entonces el valor de \hat{y} , el residuo y el cuadrado del residuo para cada muestra escogida del conjunto de datos, obteniendo un error cuadrático medio (MSE) de 5,04, como se aprecia en la siguiente tabla.

ID	Sqrt_living	Price (y)	Predicción $\hat{y} = w_0x + w_1$	Error (Residuo) $ \hat{y}-y $	Error ²
1	0.16	0.0	2,16	-2,16	4,6656
2	0.99	1.0	2,99	-1,99	3,9601
3	0.38	0.01	2,38	-2,37	5,6169
4	0.40	0.05	2,4	-2,35	5,5225
5	0.38	0.11	2,38	-2,27	5,1529
6	0.0	0.08	2	-1,92	3,6864
7	0.17	0.01	2,17	-2,16	4,6656
8	0.66	0.08	2,66	-2,58	6,6564
9	0.56	0.07	2,56	-2,49	6,2001
10	0.23	0.16	2,23	-2,07	4,2849
				Suma	50,4114
				Suma/10	5,04114

El error cuadrático medio se puede expresar matemáticamente de la siguiente manera:

$$MSE = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y})^2$$

El MSE permite saber cuál es el margen de error de nuestro modelo, y por lo tanto como ya se mencionó, nuestro objetivo es minimizarlo, buscando que sea lo más cercano a 0, lo cual significa que no haya diferencia entre los valores reales del dataset y los valores predichos. Una vez alcanzado el error mas pequeño en la función anterior se descubren los valores de w_0 y w_1 , que hicieron posible llegar a ese mínimo.

Si aplicaramos la misma función con otros valores para w_0 y w_1 , como 5 y 2 respectivamente obtendríamos un MSE de 31,83, lo que significa un aumento significativo en el error. Esto se va a seguir presentando si seguimos dando valores cada vez mas altos a estas variables. O en otras palabras a medida que aumentamos los coeficientes nos alejamos de un error mínimo. Por tanto, si probamos valores pequeños como 0,1 y 0,05 obtendremos un mejor ajuste. De cualquier manera este procedimiento evidentemente es muy poco práctico, por lo que mas adelante en este capítulo veremos algoritmos en Python que nos ayudarán a cumplir con esta tarea de una forma más rápida. Por ahora, intentemos dibujar los puntos de datos de las 10 primeras instancias, y luego graficaremos las tres hipótesis siguientes a partir de los pesos w_0 y w_1 , para confirmar el resultado de nuestro análisis:

$\hat{y}_1 = x + 2$	$w_0 = 2$ y $w_1 = 1$	$MSE_1 = 5,04$
$\hat{y}_2 = 2x + 5$	$w_0 = 5$ y $w_1 = 2$	$MSE_2 = 31,83$
$\hat{y}_3 = 0.1x + 0.05$	$w_0 = 0.05$ y $w_1 = 0.1$	$MSE_3 = 0,075$

El siguiente fragmento de código establece los valores para X e y de nuestro ejemplo, luego normaliza sus valores y por último dibuja las 3 rectas anteriores, como notaremos la última línea es la que otorga una mejor aproximación del resultado esperado.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
df = pd.read_csv("precios_casas.csv")
X = df.iloc[0:10,3].values
y = df.iloc[0:10,0].values
```

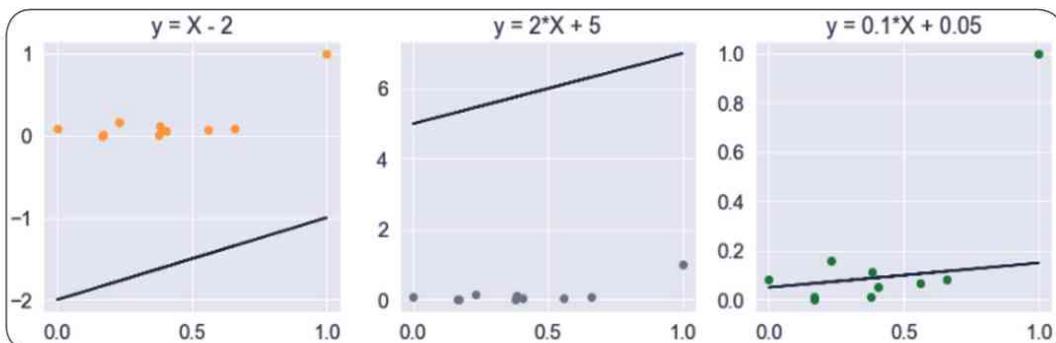
```

mms_x = MinMaxScaler()
mms_y = MinMaxScaler()
X_std = mms_x.fit_transform(X.reshape(-1,1))
y_std = mms_y.fit_transform(y.reshape(-1,1))
y_p1 = X_std - 2
y_p2 = 2*X_std + 5
y_p3 = 0.1*X_std + 0.05
fig, axs = plt.subplots(1,3, figsize=(15,4))
axs[0].scatter(X_std, y_std, color='orange')
axs[0].plot(X_std, y_p1, color='black')
axs[0].set_title('y = X - 2')

axs[1].scatter(X_std, y_std, color='gray')
axs[1].plot(X_std, y_p2, color='black')
axs[1].set_title('y = 2*X + 5')

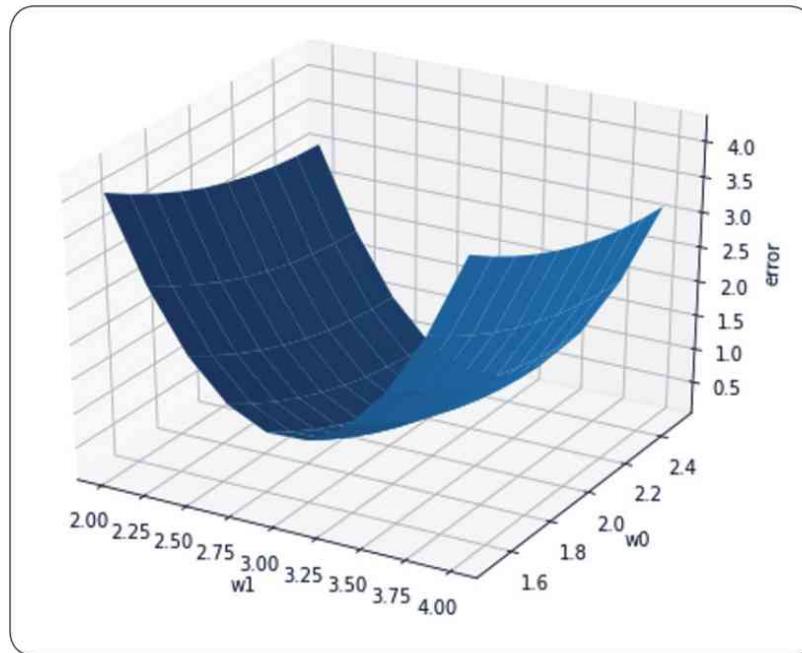
axs[2].scatter(X_std, y_std, color='green')
axs[2].plot(X_std, y_p3, color='black')
axs[2].set_title('y = 0.1*X + 0.05')
plt.show()

```



Como pudimos conocer, para cada combinación de valores de w_0 y w_1 hay un valor del error cuadrático medio, con los cuales podemos construir una grafica de *superficie de error*, como la de la figura que sigue. En ella cada par de valores de los pesos representan un punto en el plano x-y (*espacio de pesos*), y la suma del error cuadrático medio determina la altura de la superficie de error encima del plano x-y. Esta gráfica se caracteriza por ser convexa, lo que permite la utilización de un algoritmo conocido como descenso del gradiente y que es mucho más eficiente para encontrar los pesos óptimos.

Por consiguiente el modelo que mejor se ajuste a los datos de entrenamiento es aquel que tenga el menor punto en la superficie de error, conocido como **minimo global**.

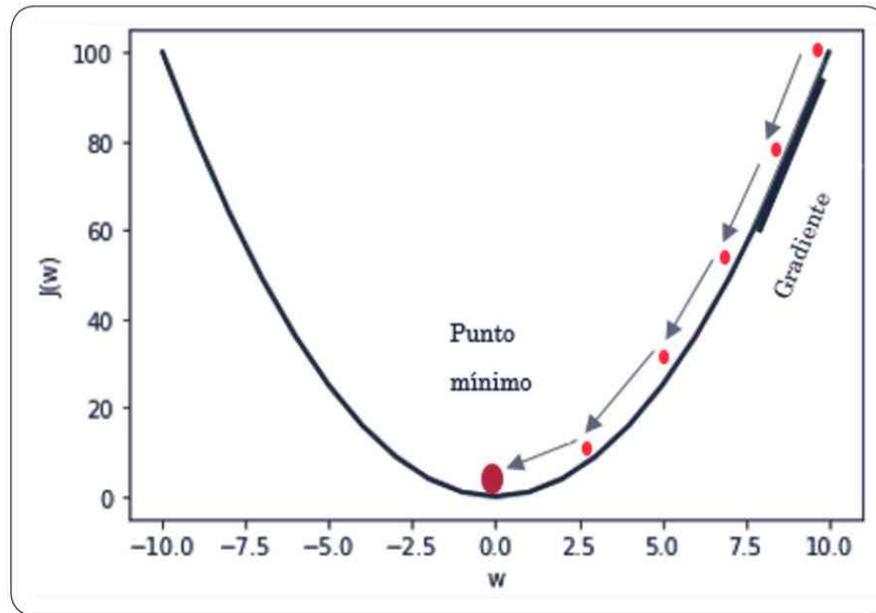


5.3 Descenso del gradiente

La técnica del descenso del gradiente es un algoritmo iterativo usado para encontrar la combinación de pesos que minimicen una función de costo. En cada iteración este algoritmo realiza una actualización sobre los pesos haciendo un paso en la dirección opuesta del gradiente de la función de coste en cada punto ($J(w)$).

$$J(w) = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$$

Esta técnica es fácil de entender si suponemos que nos encontramos en la cima de una montaña y queremos descender hacia el punto más bajo de la misma, pero queremos hacerlo de la manera más adecuada posible. Cuando estamos en lo más alto de la montaña, donde la pendiente es alta, descenderemos más rápido y faremos recorridos o pasos más grandes, y por supuesto en puntos donde la pendiente es menor, daremos pasos más pequeños para alcanzar el valle.



Suponga que el vector w contiene los pesos w_0 y w_1 , que se desean encontrar iterativamente hasta que haya una convergencia de la función de costo o se llegue a un umbral determinado. Por tanto, el gradiente lo podemos expresar mediante el siguiente pseudocódigo:

```
repetir_hasta_converger {
```

$$w_j := w_j - \alpha \frac{\partial J(w)}{\partial w_j}$$

```
}
```

De lo anterior encontramos:

6.1.1. α es la tasa de aprendizaje. Este término determina el tamaño del paso realizado en cada iteración durante el descenso. Este valor controla cuánto se avanza y típicamente se le asigna un valor en el rango de 0.00001 a 10. Este valor se asigna mediante prueba y error teniendo en cuenta que un valor muy grande durante la ejecución del gradiente puede causar que se produzcan saltos bruscos de un lado a otro en la superficie de error, dicho efecto causa que la suma de los errores cuadráticos incremente y decremente subitamente haciendo que el algoritmo nunca alcance la convergencia.

6.1.2. Se calcula la derivada parcial de la función de costo J con respecto a cada peso w_j . Este paso nos genera las siguientes ecuaciones para hallar los pesos w_0 y w_1 , respectivamente:

$$J(w) = \frac{1}{m} \sum_{i=1}^m (y_i - w_0 - w_1 x_i)^2$$

$$\frac{\partial J}{\partial w_0} = -2 \frac{1}{m} \sum_{i=1}^m (y_i - w_0 - w_1 x_i)$$

$$\frac{\partial J}{\partial w_1} = -2 \frac{1}{m} \sum_{i=1}^m (y_i - w_0 - w_1 x_i) x_i$$

Luego,

$$w_0 = w_0 + \alpha * 2 \frac{1}{m} \sum_{i=1}^m (y_i - w_0 - w_1 x_i)$$

$$w_1 = w_1 + \alpha * 2 \frac{1}{m} \sum_{i=1}^m (y_i - w_0 - w_1 x_i) x_i$$

De manera general la ecuación para encontrar la actualización de los pesos sería:

$$w_j = w_j + \alpha * 2 \frac{1}{m} \sum_{i=1}^m (y_i - w_0 - w_1 x_i) x_i$$

6.1.3. Se repite el proceso anterior de manera repetitiva hasta que la función de costo tenga un valor muy pequeño cercano a 0. (0 en el error significa un 100 % de exactitud). Por ser algo muy poco probable de alcanzar en la práctica, frecuentemente el algoritmo se controla por medio de umbrales como el número de épocas (iteraciones), hiperparámetro fijado por el programador a partir de la experimentación, proceso que realiza insistentemente hasta encontrar el que determine el mejor resultado.

Tratemos de implementar este algoritmo en Python, para ello vamos a seguir los siguientes pasos:

1. Obtener las primeras 10 muestras una vez cargado el dataframe con los datos y establecemos el número máximo de épocas.

```
import pandas as pd
df = pd.read_csv("precios_casas.csv")
X = df[['sqft_living']].values[:10]
y = df['price'].values[:10]
num_epocas = 100
```

2. Estandarizar los valores de X e y para que manejen una misma escala, lo cual es imprescindible si queremos que la función de costo converja más rápidamente.

```
import numpy as np
from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
sc_y = StandardScaler()
X = sc_x.fit_transform(X)
y = sc_y.fit_transform(y[:, np.newaxis]).flatten()
```

3. Escribimos los métodos necesarios para implementar el algoritmo del descenso del gradiente:

```
# Este método calcula un valor para y basado en los pesos
def calcular_prediccion(x, w):
    prediccion = np.dot(x, w[1:]) + w[0]
    return prediccion

# Este método permite calcular los valores de J
def calcular_costo(X, y):
    m = len(X)
    error = 0.0
    prediccion = calcular_prediccion(X, w)
    error = (y - prediccion)
    costo = (error ** 2).sum() / m
    return costo

# Este método implementa el algoritmo del descenso del gradiente
def entrenar(X, y, w, max_iter, tasa_aprendizaje):
    costos = []
    for i in range(max_iter):
        prediccion = calcular_prediccion(X, w)
        error = (y - prediccion)
        w[0] += 2 * tasa_aprendizaje * error.sum()
        w[1:] += 2 * tasa_aprendizaje * np.dot(X.T, error)
        costo = calcular_costo(X, y)
        costos.append(costo)
    return w, costos
```

4. Invocamos la función entrenar para 100 iteraciones con una tasa de aprendizaje de 0.001:

```
w = [0.1 ,0.5]
w, costos = entrenar(X, y, w, max_iter=num_epocas, tasa_aprendizaje=0.001)
print('w0: {:.3f}'.format(w[0]))
print('w1: {:.3f}'.format(w[1]))
```

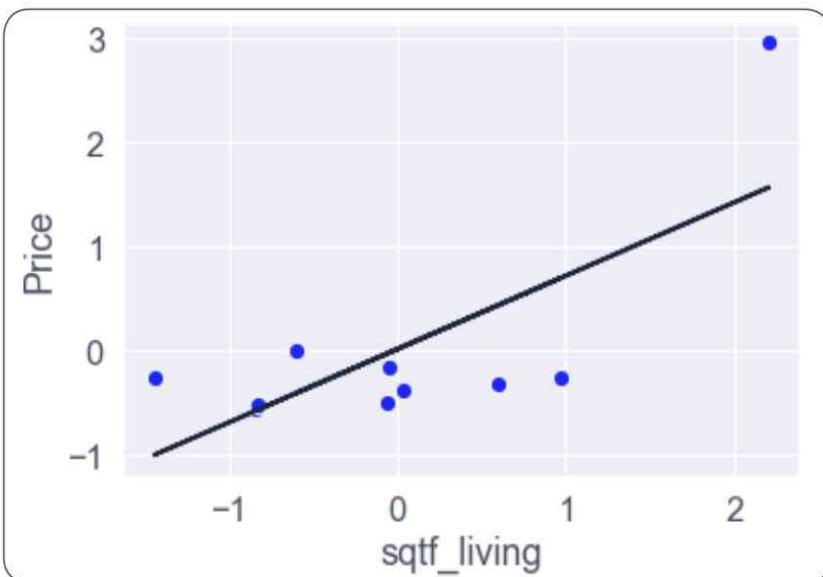
w0: 0.013

w1: 0.706

Siendo los pesos anteriores los que lógicamente configuran la recta que mejor ajusta los puntos de datos, quedando el modelo así:

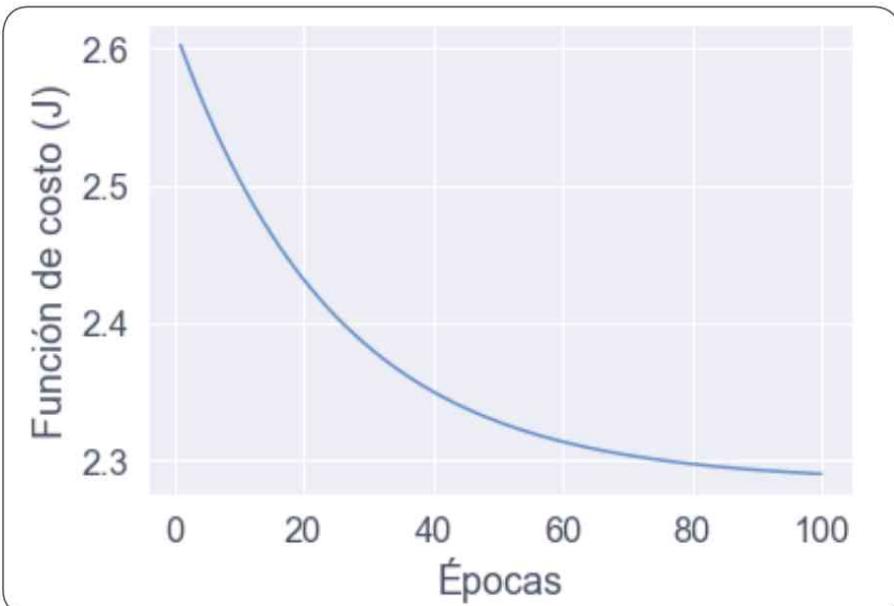
5. Podemos dibujar una gráfica de dispersión de los puntos de datos y la recta con los pesos obtenidos en el paso anterior. Esto lo conseguimos con el siguiente fragmento de código:

```
import matplotlib.pyplot as pl
pl.scatter(X, y, c='blue', edgecolor='white', s=70)
pl.plot(X, calcular_prediccion(X,w), color='black', lw=2)
pl.xlabel('sqtf_living')
pl.ylabel('price')
pl.show()
```



- Así mismo, podemos hacer un gráfico donde se evidencie por cada iteración como fue el comportamiento de la función de costo J hasta alcanzar la convergencia.

```
pl.plot(range(1, num_epocas + 1), costos)
pl.ylabel('Función de costo (J)')
pl.xlabel('Epocas')
pl.show()
```



5.4 Regresión lineal mediante scikit-learn

Ahora trataremos de hacer el mismo proceso anterior, pero haciendo uso de la librería Scikit-Learn, mediante la clase `LinearRegression` del módulo `sklearn`, que además cuenta con la implementación de otros algoritmos de aprendizaje automático de tipo supervisado y no supervisado muy fáciles de usar, principalmente porque siguen el mismo esquema de implementación. Los pasos necesarios para utilizarlos son básicamente:

- Crear una instancia de la clase que implementa el algoritmo específico que queremos utilizar.
- Con el objeto anterior invocar el método `fit()`, que usado en algoritmos de regresión y clasificación lleva como parámetros una matriz con los valores de los atributos descriptivos (X) y un vector con valores para la variable de destino (y).

3. Con el objeto entrenado, se puede llamar al método predict() con un conjunto de instancias X sin etiquetar, devolviendo una secuencia con las etiquetas predichas.

5.4.1. Regresión Simple

La implementación del modelo de regresión lineal simple con las 10 primeras instancias del dataset y con las variables *sqft_living* y *price* debidamente escaladas sería:

```
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
# se obtienen las primeras 10 instancias
X = df[['sqft_living']].values[:10]
y = df['price'].values[:10]
# se estandarizan las variables
from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
sc_y = StandardScaler()
X_esc = sc_x.fit_transform(X)
y_esc = sc_y.fit_transform(y[:, np.newaxis]).flatten()
reg.fit(X_esc, y_esc)
print('w0: {:.3f}'.format(reg.intercept_))
print('w1: {:.3f}'.format(reg.coef_[0]))
```

w0: 0.000

w1: 0.737

Como vemos, este resultado se parece bastante al logrado con nuestra implementación en Python del descenso del gradiente, note el lector además, como los valores de los pesos w_0 y w_1 se han redondeado a 3 decimales y que w_0 es un número muy cercano a 0, por tanto no se ha incluido en la ecuación final:

$$y = 0.737 * x$$

En caso de querer saber por ejemplo que precio tendría una casa con un *sqft_living* de 10.000 llamaríamos al método predict() con el objeto reg, prediciendo un precio de \$7.794.

```
sqft_living = sc_x.transform(np.array([[10000]]))
precio = reg.predict(sqft_living)
print('Precio predicho: %.3f' % precio )
```

Si probamos ahora con nuestro modelo basado en el descenso del gradiente vemos que obtenemos un resultado bastante parecido:

$$y = 0.706 * 10000 + 0,013 = 7.060$$

Sin embargo, esta implementación que hicimos con la clase LinearRegression no usa el método del descenso del gradiente, pero si podemos implementar una regresión lineal usando SGD (Descenso del gradiente estocástico) por medio de la clase SGDRegressor, que por defecto minimiza la función de costo MSE. Concretamente, el SGD selecciona una instancia aleatoria (de ahí el término estocástico) del conjunto de entrenamiento en cada paso y calcula el gradiente basado solo en esa instancia, esta característica permite que el algoritmo sea más rápido, ya que tiene que manejar muy pocos datos en cada iteración.

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, eta0=0.001, random_
state=1)
sgd_reg.fit(X, y)
w0, w1 = sgd_reg.intercept_, sgd_reg.coef_
print("w0:", w0)
print("w1:", w1)
```

w0: [0.00017709]
w1: [0.08539101]

5.4.2 Regresión múltiple

Hagamos la implementación de un modelo de regresión con múltiples variables, usaremos en este ejemplo las columnas: *bedrooms*, *bathrooms*, *sqft_living* y *sqft_lot*, y todas las 4.600 instancias del dataset. Se separarán los datos en 70% para entrenamiento y 30% para pruebas, además al final se calcula la exactitud del modelo por medio de la función *score()*, disponible en los algoritmos de la librería y que internamente aplica una métrica para determinar el exactitud del modelo, para modelos de regresión, la métrica por defecto es el coeficiente de determinación R2, la cual maneja un valor mínimo de 0 y un valor máximo de 1, en nuestro ejercicio un valor de 0.4 indica un resultado aceptable. Las métricas de evaluación serán tratadas con más detalle en un próximo capítulo.

```

from sklearn.linear_model import LinearRegression
reg2 = LinearRegression()
# se obtienen los arreglos X e y
X = df[['bedrooms','bathrooms','sqft_living','sqft_lot']].values
y = df['price'].values
# se estandarizan las variables
from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
sc_y = StandardScaler()
X = sc_x.fit_transform(X)
y = sc_y.fit_transform(y[:, np.newaxis]).flatten()
# se separa el dataset en conjunto de entrenamiento y prueba
from sklearn.model_selection import train_test_split
X_ent, X_pru, y_ent, y_pru = train_test_split(X, y, test_size=0.3, random_
state=1)
# ajustamos el modelo con los datos de entrenamiento
reg2.fit(X_ent, y_ent)
print('Intercepto: {:.3f}'.format(reg2.intercept_))
print('Coeficientes: ', reg2.coef_)
# calculamos la exactitud del modelo
print('Exactitud: {:.3f}'.format(reg2.score(X_pru, y_pru)))

```

Intercepto: 0.009

Coeficientes: [-0.09001078 0.00709681 0.49100858 -0.05325143]

Exactitud: 0.428

5.5 Regresión con Random Sample Consensus (RAMSAC)

Los valores extremos (*outliers*) son puntos de datos numéricamente distantes del resto de datos a los que toca prestarles mucha atención debido a que la regresión lineal es muy sensible a ellos, ya que pueden provocar un sesgo de la línea hacia esos valores y se pierda con ello capacidad predictiva.

La técnica llamada RAMSAC (Consenso de muestra aleatoria) plantea descartar los *outliers* y trabajar solo con los denominados *inliers* o puntos interiores o próximos de un modelo de regresión. Básicamente es un algoritmo iterativo que separa el conjunto de datos en *outliers* y *inliers*. El modelo es entrenado solo con muestras validas que son evaluadas internamente para verificar si son *inliers*. El

proceso termina cuando se ha alcanzado un número máximo de iteraciones o se ha alcanzado un umbral establecido.

En las siguientes líneas de código implementaremos el algoritmo RAMSAC sobre nuestro modelo de regresión creado en el subapartado anterior.

```
from sklearn.linear_model import RANSACRegressor
ransac = RANSACRegressor(reg)
ransac.fit(X_esc,y_esc)
print("w0: %.3f" % ransac.estimator_.intercept_)
print("w1: %.3f" % ransac.estimator_.coef_)
```

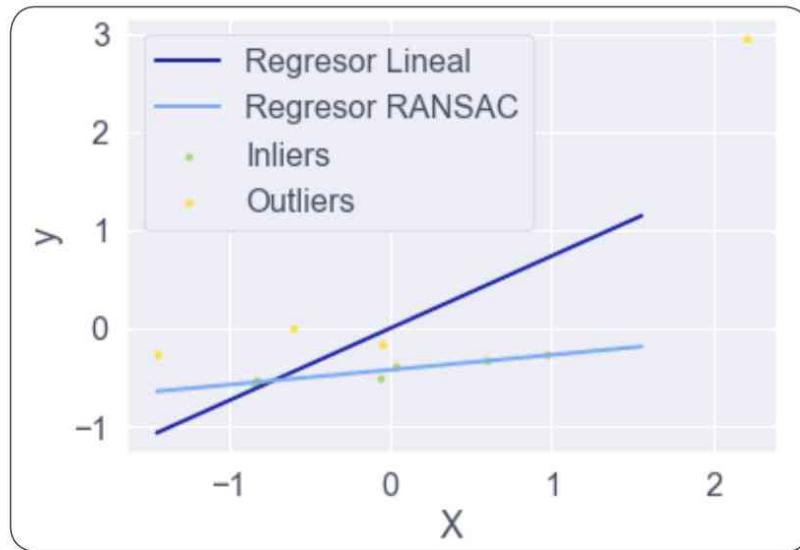
w0: -0.422

w1: 0.152

Como vemos el coeficiente (w1), es la pendiente de la recta y disminuye con RANSAC, esto se debe a que el outlier ubicado en el extremo derecho no es tenido en cuenta por el algoritmo para ajustar el modelo. Lo anterior lo podemos corroborar si graficamos los dos modelos generados: el de regresión lineal pasado y el de RANSAC con el fin de compararlos y evidenciar el ajuste proporcionado por uno y otro.

```
inlier_mask = ransac.inlier_mask_
outlier_mask = np.logical_not(inlier_mask)
line_X = np.arange(X.min(), X.max())[:, np.newaxis]
line_y = reg.predict(line_X)
line_y_ransac = ransac.predict(line_X)

lw = 2
pl.scatter(X[inlier_mask], y[inlier_mask], color='yellowgreen', marker="",
           label='Inliers')
pl.scatter(X[outlier_mask], y[outlier_mask], color='gold', marker="",
           label='Outliers')
pl.plot(line_X, line_y, color='navy', linewidth=lw, label='Regresor Lineal')
pl.plot(line_X, line_y_ransac, color='cornflowerblue', linewidth=lw,
        label='Regresor RANSAC')
pl.legend(loc='upper left')
pl.xlabel("Entrada")
pl.ylabel("Respuesta")
pl.show()
```



5.6 Regresión lineal polinómica

La regresión polinómica surge debido a que hay conjuntos de datos que difícilmente se pueden modelar mediante una regresión lineal, puesto que la tendencia de los datos es a describir un comportamiento curvo. Intuitivamente si a un modelo lineal le adicionamos términos polinómicos queda definido bajo la siguiente ecuación general:

$$Y = w_0 + w_1x + w_2x^2 + \dots + w_kx^k = \sum_{i=0}^k W_i * X^i \quad \text{con} \quad x^0 = 1$$

Además, el conjunto de datos X se convertirá en una matriz de k columnas, mientras que Y y W siguen siendo arreglos de 1 dimensión:

$$X = \begin{bmatrix} x_1^0 & x_1^1 & x_1^2 & \dots & x_1^k \\ x_2^0 & x_2^1 & x_2^2 & \dots & x_2^k \\ \dots & \dots & \dots & \dots & \dots \\ x_n^0 & x_n^1 & x_n^2 & \dots & x_n^k \end{bmatrix}, \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix}, \quad W = \begin{bmatrix} w_0 \\ w_1 \\ \dots \\ w_n \end{bmatrix}$$

En todo caso, hay una forma que ayuda a simplificar las cosas, y parte de convertir una regresión polinómica a una regresión lineal múltiple, asumiendo que por ejemplo: $x_1 = x$; $x_2 = x^2$; $x_3 = x^3$ y así sucesivamente. Esto nos permitirá tratar el problema como si de una regresión múltiple se tratara:

$$y = w_0x_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n = \sum_{i=0}^n W_i * X_i$$

Para realizar este proceso de una manera más amena encontramos en Scikit-learn una clase transformadora llamada `PolynomialFeatures` con la cual se puede convertir en polinomio un simple problema de regresión. Donde el parámetro `degree` representa el grado del polinomio. Esta clase permite generar una matriz compuesta de todas las combinaciones polinomiales de las características de grado menor o igual a `degree`.

Un modelo de regresión polinómico se puede usar con el fin de modelar problemas no lineales. Miremos como podemos realizar esta acción en el ejemplo que hemos estado estudiando a lo largo del capítulo utilizando un polinomio de grado 2 (`degree=2`).

$$Y = w_0 + w_1 x + w_2 x^2$$

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
X_ent, X_pru, y_ent, y_pru = train_test_split(X_esc, y_esc, test_size=0.3 ,
random_state=50)
```

```
poly = PolynomialFeatures(degree=2)
X_ent_poly = poly.fit_transform(X_ent)
X_pru_poly = poly.fit_transform(X_pru)
X_ent_poly
```

```
array([[ 1.0000000e+00, -5.93099559e-02,  3.51767087e-03],
       [ 1.0000000e+00, -4.61299657e-02,  2.12797373e-03],
       [ 1.0000000e+00,  2.20764836e+00,  4.87371127e+00],
       [ 1.0000000e+00, -1.44320893e+00,  2.08285201e+00],
       [ 1.0000000e+00,  3.29499755e-02,  1.08570089e-03],
       [ 1.0000000e+00, -5.99689554e-01,  3.59627561e-01],
       [ 1.0000000e+00, -8.36929378e-01,  7.00450783e-01]])
```

El método `fit_transform()` toma los valores de X y los eleva a potencias desde 0 hasta 2.

$$\begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \rightarrow \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \dots & \dots & \dots \\ 1 & x_n & x_n^2 \end{bmatrix}$$

Este enfoque permite tratar la regresión polinómica como una regresión múltiple, por lo que podemos realizar las mismas acciones que hemos estado usando a lo largo de este capítulo, como se aprecia en el siguiente código.

```
reg = LinearRegression()
reg.fit(X_ent_poly, y_ent)
# Coeficientes
print ('Coeficientes:', reg.coef_)
print ('Intercepto:', reg.intercept_)
```

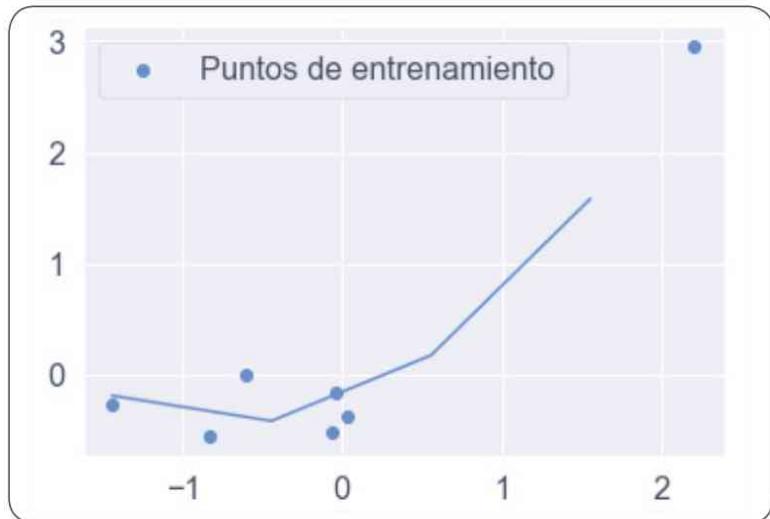
Coeficientes: [0. 0.54382876 0.40900654]

Intercepto: -0.2525610327311758

En este resultado observamos los coeficientes y el intercepto del polinomio. Seguidamente podemos dibujar la curva polinómica y calcularemos e imprimiremos el coeficiente de determinación con la función score().

```
x_fit = np.arange(X_ent.min(), X_ent.max(), 1)[:, np.newaxis]
y_p = reg.predict(poly.fit_transform(x_fit))
pl.scatter(X_ent, y_ent, label='Puntos de entrenamiento')
pl.plot(x_fit, y_p)
pl.legend(loc='upper left')
pl.show()
print('Precisión del modelo: {:.2f}'.format(reg.score(X_ent_poly, y_ent)))
```

Precisión del modelo: 0.97



Nótese que el coeficiente R^2 es 0.97, mucho mayor que el valor obtenido con el ejemplo de regresión anterior, lo cual nos enseña que un modelo polinómico, aparentemente otorga una mejoría, aunque puede no ser siempre la mejor

solución porque puede producir un modelo subajustado. Sin embargo, un poco de experimentación en la asignación de otros valores para los hiperparámetros como *degre* nos puede venir bien para encontrar la solución que mejor ajuste los datos.

5.7 Regresión lineal múltiple en notación matricial

En este subapartado mostraremos una técnica para hallar los términos de un modelo lineal multivariable mediante una notación basada en vectores y matrices. Recordemos la forma de la regresión múltiple.

$$Y = w_0 + w_1 x_1 + w_2 x_2 + \cdots w_d x_d$$

Muchos modelos de aprendizaje pueden mejorar si se agregan más variables descriptivas, situación bastante realista, ya que en la práctica comúnmente vamos a encontrarnos con variables destino dependientes de muchas variables. Tal es el caso de nuestro ejemplo de predicción del precio de casas, si adicionamos otros atributos como *bedrooms* y *bathrooms* como en la tabla siguiente es posible obtener mejores resultados.

Sqft_living	bedrooms	bathrooms	price
1340	3	1.50	3.130000e+05
3650	5	2.50	2.384000e+06
1930	3	2.00	3.420000e+05
2000	3	2.25	4.200000e+05
...

A pesar de que el método a presentar es fácil e intuitivo puede ser computacionalmente muy costoso lo cual limita bastante su uso, haciéndolo no adecuado para proyectos reales. Antes de ver el procedimiento en sí, haremos un repaso de algunos conceptos de álgebra lineal que se estarán mencionando durante su explicación.

Vector: Se puede definir como una secuencia de elementos, donde cada elemento se obtiene a partir de un índice o posición que puede ser 0 o 1. Se representa de la siguiente manera.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 1 \\ 5 \\ 4 \\ 3 \end{bmatrix},$$

Vemos que los vectores x e y tienen tamaños de 3 y 4 respectivamente. Recordemos que para crear un array como el anterior mediante la librería NumPy escribimos:

```
import numpy as np
v = np.array([1, 4, 5, 3])
```

Transpuesta de un vector: La transpuesta de un vector v , se denota como v^T . Consiste en girar el vector de manera tal que la columna se convierta en fila. Por ejemplo:

$$v = \begin{bmatrix} 1 \\ 5 \\ 4 \\ 3 \end{bmatrix}, v^T = [1 \ 5 \ 4 \ 3]$$

La transpuesta del vector v se consigue mediante la función T. así:

```
v1.T
```

Recuerde que en la ecuación de la recta con la cual estudiamos la regresión lineal simple, si los pesos w_0 y w_1 los colocamos en un vector w y en un vector x colocamos los coeficientes de x más la adición del número 1, tendremos que dicha ecuación será igual al producto escalar o punto entre la transpuesta del vector w y el vector x .

$$y = w_0 + w_1 * x = w^T * x$$

$$w = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}, x = \begin{bmatrix} 1 \\ x_1 \end{bmatrix}, \text{ luego}$$

$$y = [w_0 \ w_1] * \begin{bmatrix} 1 \\ x_1 \end{bmatrix} = w_0 * 1 + w_1 * x_1$$

Consideremos los siguientes vectores:

$$w = \begin{bmatrix} 1 \\ 3 \end{bmatrix}, x = \begin{bmatrix} 1 \\ 10 \end{bmatrix}, \text{ luego si } y = w^T * x, \text{ entonces}$$

$$y = [1 \ 3] * \begin{bmatrix} 1 \\ 10 \end{bmatrix} = [1 * 1 + 3 * 10] = 31$$

En la biblioteca NumPy hay una función llamada dot() para realizar el producto punto de vectores y matrices.

```
w = np.array([1, 3])
x = np.array([1, 10])
y = w.T.dot(x)
print(y)
```

Matriz: A diferencia del vector una matriz está conformada por filas y columnas. Y para acceder a algún elemento contenido en ella se usan dos índices uno para referirse a la fila y el otro a la columna. Por ejemplo, considere la siguiente matriz llamada M de 3x2 (3 filas y dos columnas).

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \\ m_{31} & m_{32} \end{bmatrix}, M1 = \begin{bmatrix} 1 & 3 \\ 5 & 7 \\ 9 & 11 \end{bmatrix}$$

El elemento m_{22} se encuentra en la fila 2 y columna 2 de la matriz. Como es de conocimiento del lector una matriz como la anterior se puede crear en NumPy con el siguiente código:

```
matriz = np.mat([[1, 3],
                 [5, 7],
                 [9, 11]])
```

Algunas operaciones que podemos hacer sobre la matriz anterior son:

```
print("Elemento de la matriz en la posición 0 y 1:" , matriz[0,1]) # Resultado 3
print("Dimensiones matriz" , matriz.shape) # Resultado: (3, 2)
print("Maximo matriz" , np.max(matriz)) # Resultado: 11
print("Media aritmética matriz" , np.mean(matriz)) # Resultado: 6.0
print("Varianza matriz" , np.var(matriz)) # Resultado: 11.666
print("Desviación estandar matriz" , np.std(matriz)) # Resultado: 3.415
```

Transpuesta de una matriz: Para una matriz M, la transpuesta M^T , se consigue convirtiendo las filas en columnas y las columnas en filas.

$$M = \begin{bmatrix} 1 & 3 \\ 4 & 5 \\ 9 & 6 \end{bmatrix}, \quad M^T = \begin{bmatrix} 1 & 4 & 9 \\ 3 & 5 & 6 \end{bmatrix}$$

Con la función T obtenemos la transpuesta de la matriz M, así:

```
M = np.mat([[1, 3],
[4, 5],
[9, 6]])

print(M.T)
```

Matriz idéntica: Una matriz I es idéntica si presenta unos en la diagonal principal y ceros en el resto de la matriz.

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Una característica importante de una matriz idéntica I es que, si la multiplicamos por una matriz A, el resultado es la matriz A.

$$A \cdot I = I \cdot A = A$$

Matriz inversa: La inversa de la matriz A es definida como la matriz A^{-1} que satisface $A^{-1} \cdot A = I$:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad A^{-1} = \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Suponga la siguiente matriz:

$$MA = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad MA^{-1} = \frac{1}{1*4-2*3} \begin{bmatrix} 4 & -2 \\ -3 & 1 \end{bmatrix} = \frac{1}{-2} \begin{bmatrix} 4 & -2 \\ -3 & 1 \end{bmatrix}$$

$$MA^{-1} = \begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix}$$

En NumPy esta operación se realiza con la función `inv()` del submódulo `linalg` consiguiendo en este caso el mismo resultado que se obtuvo analíticamente:

```
inv = np.linalg.inv(ma)
print(inv)
```

En este sentido se debe cumplir que:

$$A \cdot A^{-1} = A^{-1} \cdot A = I$$

Teniendo en cuenta el repaso anterior miremos ahora la técnica enunciada al principio de este subapartado, efectiva por supuesto para obtener los pesos de un modelo de regresión lineal:

$$W = (X^T \cdot X)^{-1} \cdot X^T \cdot Y$$

La anterior se llama ecuación normal y expresada en Python sería así:

```
import pandas as pd
import numpy as np
df = pd.read_csv("precios_casas.csv")
X = df[['sqft_living']].values
y = df['price'].values
w = np.zeros(X.shape[1])
from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
sc_y = StandardScaler()
X = sc_x.fit_transform(X)
y = sc_y.fit_transform(y[:, np.newaxis]).flatten()
# adicionamos una columna de unos
X2 = np.hstack((np.ones((X.shape[0], 1)), X))
t1 = np.linalg.inv(np.dot(X2.T, X2))
t2 = np.dot(X2.T, y)
w = np.dot(t1, t2)
print("w0 : {:.3f}".format(w[0]))
print("w1 : {:.3f}".format(w[1]))
```

w0 : 0.000

w1 : 0.430

Como el lector sabrá existen matrices que no se pueden invertir, las llamadas matrices singulares o degeneradas, esta circunstancia provoca una limitación en el método basado en la ecuación normal, pero que se puede mitigar en cierta medida eliminando

algunas características del modelo o usando una técnica especial denominada regularización, que abordaremos más adelante cuando estudiemos el capítulo 6.

5.8 Modelos no lineales

Como podrá intuir el lector si los datos no tienen una tendencia lineal entre las características y la variable objetivo, entonces debemos acudir a funciones no lineales para la construcción del modelo. A continuación, expondremos funciones que pueden ser útiles en este aspecto y finalmente un ejemplo con un dataset de suscripciones de telefonía celular. En algunas de tales funciones se omite la explicación ya que el autor asume la existencia de conocimientos por parte del lector en el área de las matemáticas.

De forma general una función no lineal puede ser representada como una función polinómica de grado d .

$$y = ax^3 + bx^2 + cx + d$$

Pero también puede ir acompañada de otros elementos como logaritmos, exponentes, fracciones, etc. Por ejemplo:

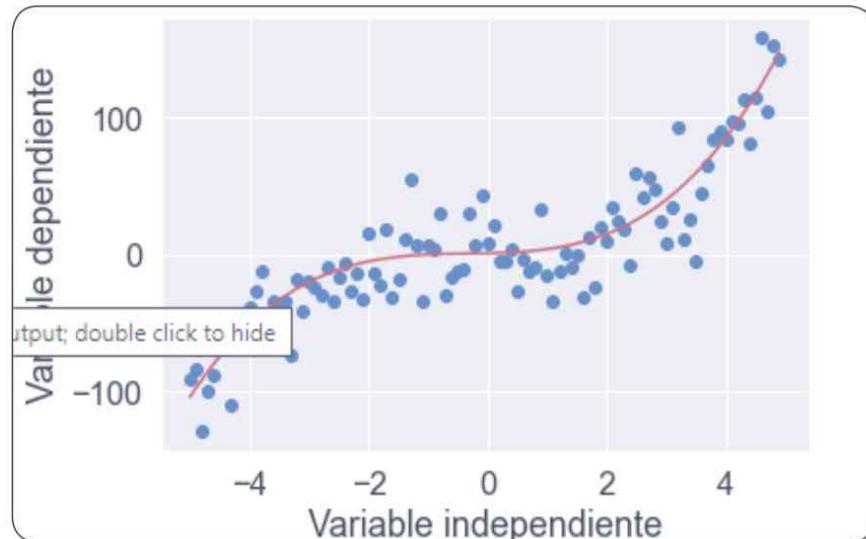
$$y = \log(ax^3 + bx^2 + cx + d)$$

5.8.1 Funciones no lineales

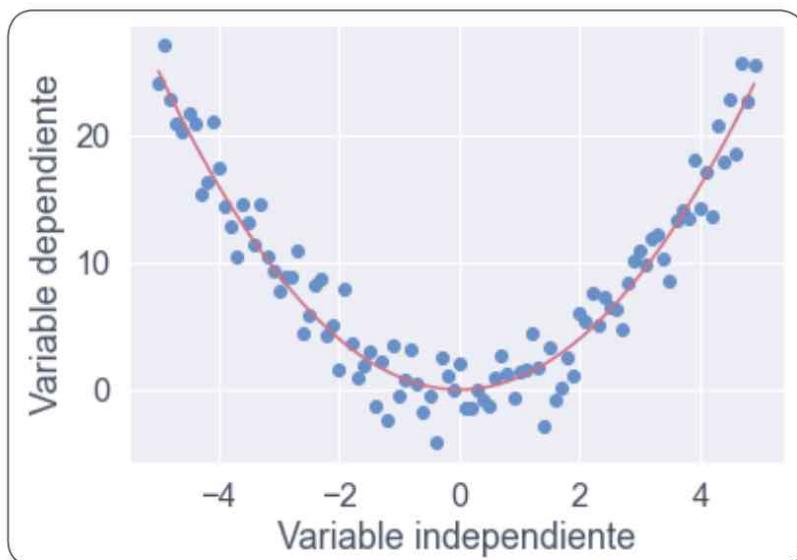
Miremos algunas funciones matemáticas y su visualización mediante Python:

Función cúbica ($y = x^3$)

```
x = np.arange(-5.0, 5.0, 0.1)
y = 1*(x**3) + 1*(x**2) + 1*x + 1
y_noise = 20 * np.random.normal(size=x.size)
ydata = y + y_noise
pl.plot(x, ydata, 'bo')
pl.plot(x,y,'r')
pl.ylabel('Variable dependiente')
pl.xlabel('Variable independiente')
pl.show()
```

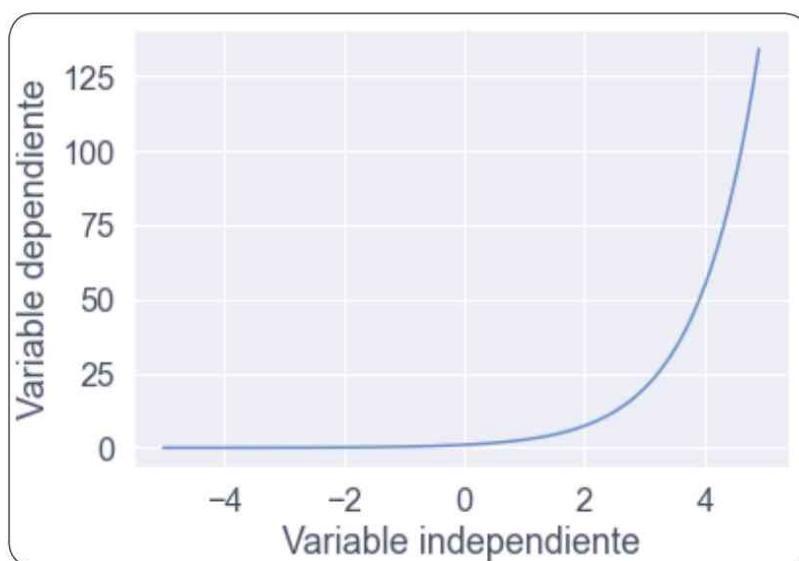
**Función cuadrática ($y = x^2$)**

```
x = np.arange(-5.0, 5.0, 0.1)
y = np.power(x,2)
y_noise = 2 * np.random.normal(size=x.size)
ydata = y + y_noise
pl.plot(x, ydata, 'bo')
pl.plot(x,y, 'r')
pl.ylabel('Variable dependiente')
pl.xlabel('Variable independiente')
pl.show()
```



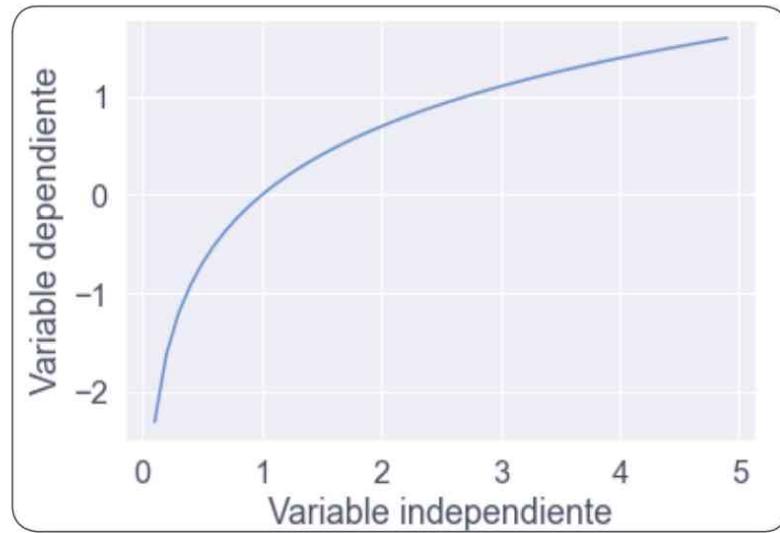
Función exponencial ($y = a + bc^x$)

```
X = np.arange(-5.0, 5.0, 0.1)
Y= np.exp(X)
pl.plot(X,Y)
pl.ylabel('Variable dependiente')
pl.xlabel('Variable independiente')
pl.show()
```



Función logarítmica ($y = \log(x)$)

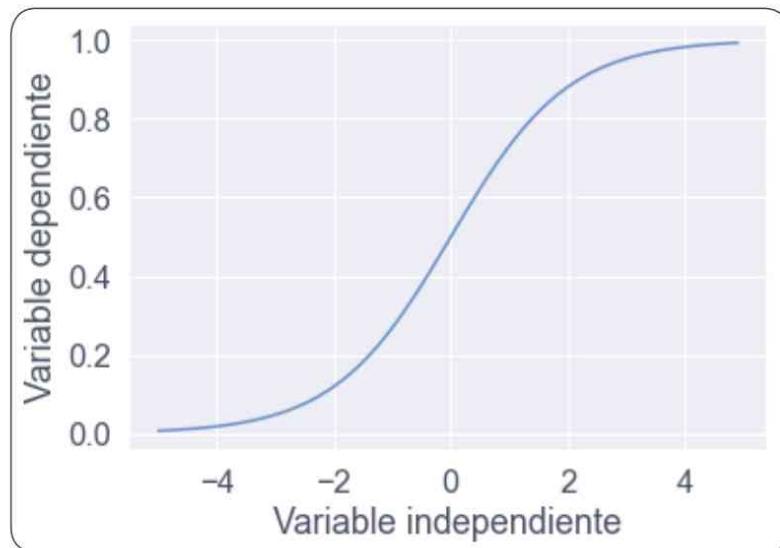
```
X = np.arange(-5.0, 5.0, 0.1)
Y = np.log(X)
pl.plot(X,Y)
pl.ylabel('Variable dependiente')
pl.xlabel('Variable independiente')
pl.show()
```



Función logística o sigmoide:

$$y = \alpha + \frac{b}{1 + c^{(x-d)}}$$

```
X = np.arange(-5.0, 5.0, 0.1)
Y = 1.0 / (1.0 + np.exp(-X))
pl.plot(X,Y)
pl.ylabel('Variable dependiente')
pl.xlabel('Variable independiente')
pl.show()
```



5.8.2 Ejemplo suscripciones de telefonía

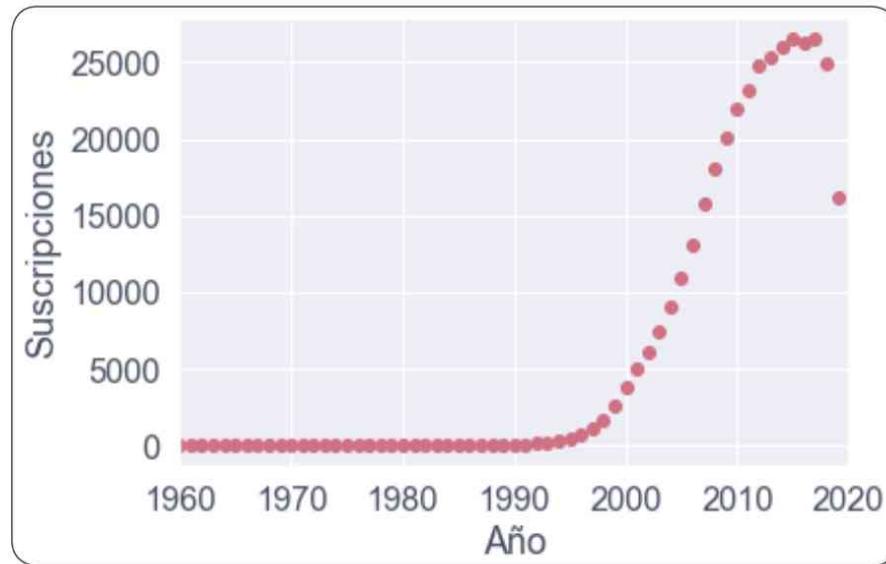
Ahora vamos a mirar cómo podemos representar mediante un modelo no lineal un dataset acerca de las suscripciones a telefonía celular realizadas a nivel mundial desde 1960 hasta 2020, el cual lo puede encontrar dentro del material de apoyo del libro. Primeramente, cargamos el dataset y mostramos 5 instancias aleatorias:

```
df2 = pd.read_csv("datos_suscripciones.csv")
df2.sample(5)
```

	año	cantidad
12	1972.0	0.000000
9	1969.0	0.000000
59	2019.0	16209.380705
38	1998.0	1637.138619
35	1995.0	474.552050

Visualizamos el conjunto de datos con Matplotlib y apreciamos como la cantidad de suscripciones aumenta conforme pasan los años, pero decrece en 2019 y 2020, pero más allá de esto podemos deducir que el comportamiento de los datos se parece al de una función sigmoide.

```
plt.scatter(df2['año'] , df2['cantidad'], color='r')
plt.xlabel("Año")
plt.xlim(1960, 2020)
plt.ylabel("Cantidad")
plt.show()
```



De la función sigmoide o logística tenemos:

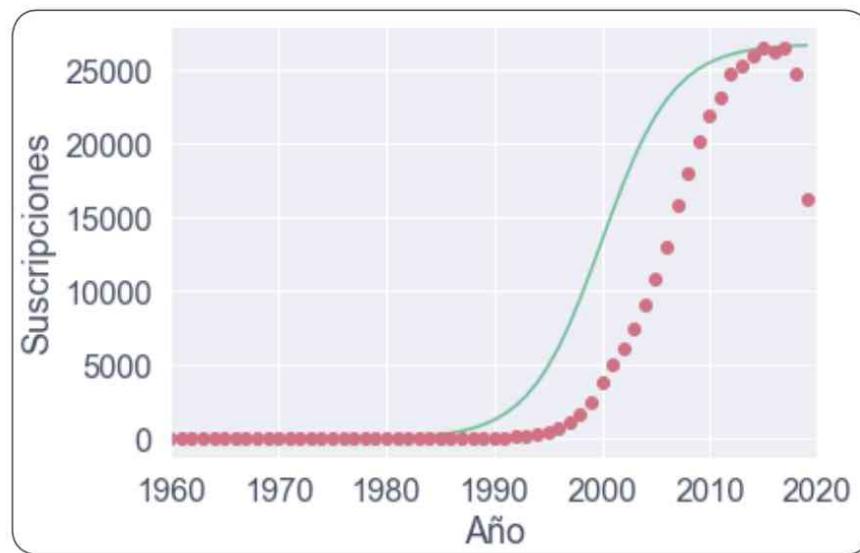
$$\hat{y} = a + \frac{1}{1 + e^{\beta_1(x-\beta_2)}}$$

Donde β_1 controla la inclinación de la curva y β_2 proyecta la curva en x.

Definamos una función en Python para la función sigmoide y hagamos una visualización preliminar de los datos junto con dicha función:

```
def sigmoid(x, beta1, beta2):
    y = 1 / (1 + np.exp(-beta1*(x-beta2)))
    return y
```

```
beta_1 = 0.3
beta_2 = 2000.0
Y_pred = sigmoid(df2['año'], beta_1, beta_2)
pl.plot(df2['año'], Y_pred*26800, 'g')
pl.plot(df2['año'], df2['cantidad'], 'ro')
pl.xlim(1960, 2020)
```



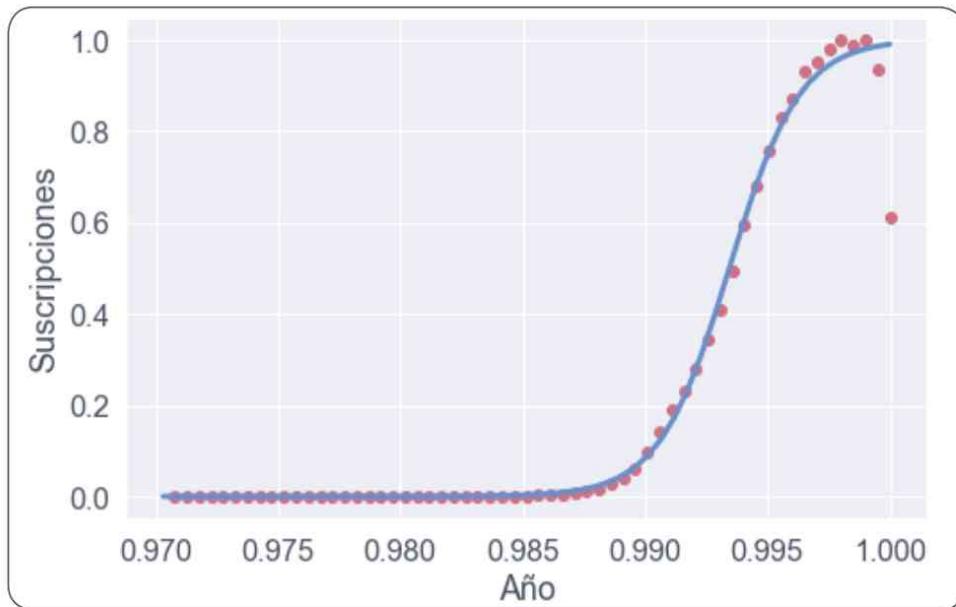
Ahora la meta es encontrar los parámetros óptimos que otorguen el mejor ajuste a la curva con respecto a los datos, para ello sin entrar en mayores detalles emplearemos la función `curve_fit()` de Scipy que usa un enfoque de mínimos cuadrados para ir ajustando los parámetros de manera iterativa por medio de la minimización de la suma de los cuadrados de los errores.

```
# normalicemos los datos
xdata = df2['año']/max(df2['año'])
ydata = df2['cantidad']/max(df2['cantidad'])

from scipy.optimize import curve_fit
popt, pcov = curve_fit(sigmoid, xdata, ydata)
print(" beta1: {:.2f}, beta2: {:.2f}".format(popt[0], popt[1]))
```

beta1: 696.54, beta2: 0.99

```
x = np.linspace(1960, 2020, 55)
x = x/max(x)
pl.figure(figsize=(8,5))
y = sigmoid(x, *popt)
pl.plot(xdata, ydata, 'ro', label='datos')
pl.plot(x,y, linewidth=3.0, label='ajuste')
pl.ylabel('Suscripciones')
pl.xlabel('Año')
pl.show()
```



Finalmente evaluamos el modelo creado con la métrica del coeficiente de determinación (`r2_score`) del submódulo `metrics` cuyo resultado es 1, lo cual no debería alegrarnos en lo absoluto porque evidentemente significa un alto nivel de sobreajuste.

```
X=xdata
y=ydata
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(xdata,ydata, test_size=0.2,
random_state=1)
# crear el modelo
popt, pcov = curve_fit(sigmoid, x_train, y_train)
# predicciones sobre el dataset de test
y_hat = sigmoid(x_test, *popt)
# Evaluacion
from sklearn.metrics import r2_score
print("R2: {:.2f}".format(r2_score(y_hat , y_test) ))
```

R2: 1.00

<https://dogramcode.com/bloglibros>



CAPÍTULO 6

Regularización, métricas de evaluación y ajuste de hiperparámetros

Temas

- 6.1 Regularización
- 6.2 Métricas y técnicas de validación de modelos de regresión
- 6.3 Curvas de aprendizaje y validación
- 6.4 Técnicas de búsqueda de cuadrículas para el ajuste de hiperparámetros

Las técnicas y algoritmos que se describen a continuación están enfocados en buscar simplificar los modelos de aprendizaje automático, comprimiendo o disminuyendo algunas de sus características para evitar caer en el problema de la maldición de la dimensionalidad, también aprenderemos a escoger los mejores valores para los hiperparámetros mediante la búsqueda de cuadrículas y exploraremos algunos métodos para validar el rendimiento en modelos de regresión.

6.1. Regularización

La regularización es un método para tratar el problema del sobreajuste, induciendo una minimización de la suma cuadrática de los errores o residuos (SSE), mediante una penalización a los valores de los coeficientes del modelo, haciendo que se reduzcan a valores cercanos a 0, lo cual ayuda a que el modelo sea menos complejo. Los métodos de regularización más usados para la regresión son Regresión Rígida (Ridge), Lasso y Red Elástica (Elastic Net).

6.1.1 Regresión Rígida

La Regresión Rígida consiste en imponer una penalización de tipo L2 a los coeficientes o pesos calculados para modificar el error de los mínimos cuadrados, adicionándole el producto entre un parámetro alfa y la suma de los cuadrados de los pesos.

$$SSE_R = \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \alpha \sum_{j=1}^p (w_j)^2$$

La implementación de la Regresión Rígida se puede realizar en Scikit-learn con la clase Ridge. Creando por supuesto una instancia y usando el parámetro *alfa*(α) que es quien controla la regularización, entre más grande sea este parámetro más fuerte será la regularización, lo que implica conseguir un modelo más simple con pesos cercanos a 0. El valor por defecto para *alfa* es 1.

Adicionalmente, usaremos en los siguientes ejemplos el conjunto de entrenamiento de Boston Housing utilizado en los ejercicios sobre regresión lineal.

```
from sklearn.linear_model import Ridge
rig = Ridge(alpha=10)

X = df[['bedrooms','bathrooms','sqft_living','sqft_lot']].values
y = df['price'].values
# se estandarizan las variables
from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
sc_y = StandardScaler()
X = sc_x.fit_transform(X)
y = sc_y.fit_transform(y[:, np.newaxis]).flatten()

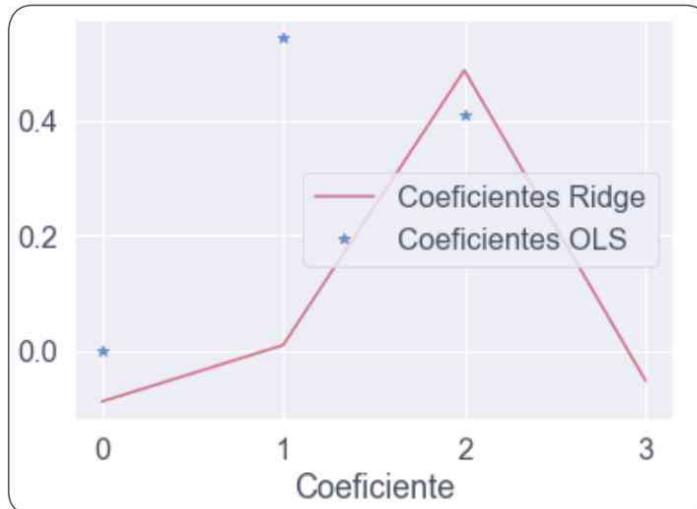
X_ent,X_pru,y_ent,y_pru=train_test_split(X,y,test_size=0.3,random_state=1)
```

```
modelo = rig.fit(X_ent , y_ent)
print("Coeficientes:" , modelo.coef_)
print('R2: {:.3f}'.format(modelo.score(X_pru, y_pru)))
```

Coeficientes: [-0.08844316 0.00950749 0.48659172 -0.05238501]
R2: 0.428

Para este caso, tras aplicar la regresión rígida vemos que no hubo mucha variación en la precisión si lo comparamos con el modelo de regresión que entrenamos con LinearRegression.

```
plt.plot(rig.coef_, 'r-' , label='Coeficientes Ridge')
plt.plot(reg.coef_,'b*' ,label='Coeficientes OLS')
plt.legend()
plt.xlabel('Coeficiente')
plt.show()
```



La biblioteca Scikit-learn ofrece la clase RidgeCV, la cual nos permite escoger el valor ideal para α .

```
from sklearn.linear_model import RidgeCV
reg = RidgeCV(alphas=[0.01, 0.1 , 1.0 , 10.0, 20.0, 50.0 , 100.0])
modelo = reg.fit(X , y)
print("Coeficientes" , modelo.coef_)
print("Valor alfa" , modelo.alpha_)
```

Coeficientes: [-0.09009378 0.01665267 0.47888413 -0.04576882]

Valor alfa: 20.0

6.1.2 Regresión Lasso

La regresión Lasso es similar a Ridge, salvo que el hiperparámetro α aplica una penalización de regularización L1 sobre la función de error SSE, dada por el valor absoluto de los coeficientes. Esta penalización consigue que un grupo de coeficientes sean reducidos a 0 y se dejan solo los de las características más importantes, esto hace que actúe como una especie de algoritmo de selección de características que reduce la complejidad de un modelo.

$$SSE_L = \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \alpha \sum_{j=1}^p |w_j|$$

```
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=0.1)
modelo = lasso.fit(X_ent, y_ent)
print("Coeficientes: ", modelo.coef_)
print('R2 Lasso: {:.3f}'.format(modelo.score(X_pru, y_pru)))
```

Coeficientes: [0. 0. 0.33414394 -0.]

R2 Lasso: 0.391

Notemos como algunos pesos se han reducido a 0, solo ha quedado el de la variable *sqft_living* que es la más importante dentro de nuestro estudio.

6.1.3 Red elástica

Este método incluye las dos penalizaciones de Ridge y Lasso, por lo que se usa con el objetivo de lograr una acción de equilibrio entre ambos, teniendo en cuenta que con $\rho = 0$ aplicamos Ridge y con $\rho = 1$ aplicamos Lasso, la penalización se controla de igual forma con el parámetro α .

$$SSE_{RE} = \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \alpha \rho \sum_{j=1}^p |w_j| + \frac{\alpha(1-\rho)}{2} \sum_{j=1}^p w_j^2$$

En Scikit podemos implementar esta técnica con la clase ElasticNet que maneja dos hiperparámetros alpha (α) y l1_ratio (ρ), si queremos entrenar por ejemplo un regresor Lasso con un alfa de 0.1 como en el ejercicio anterior tendríamos que escribir lo siguiente:

```
from sklearn.linear_model import ElasticNet
modelo = ElasticNet(alpha=0.1, l1_ratio=1)
modelo = reg.fit(X_ent, y_ent)
print("Coeficientes: ", modelo.coef_)
print('R2 Lasso: {:.3f}'.format(lasso.score(X_pru, y_pru)))
```

Coeficientes: [0. 0. 0.33414394 -0.]
R2 Lasso: 0.391

6.2. Métricas y técnicas de validación de modelos de regresión

Existen diferentes métricas que se pueden usar para validar un modelo de regresión, cada una tiene una función o método correspondiente al cual tenemos que pasárselas como parámetros los valores reales de y así como los valores predichos (\hat{y}).

Un método de evaluación bastante usado en tareas de *machine learning* es el que consiste en separar un conjunto de datos en conjunto de entrenamiento y prueba. En Scikit-learn lo podemos aplicar a través de la función `train_test_split()` que ya vimos en el capítulo 4. El primer conjunto se usa para que un modelo aprenda un patrón, estime unos parámetros y con base a ese conocimiento adquirido pueda hacer predicciones sobre información nueva. Por otra parte, el segundo conjunto es empleado para verificar el rendimiento del modelo y determinar su exactitud.

```
from sklearn.model_selection import train_test_split
X_ent, X_pru, y_ent, y_pru = train_test_split(X, y, test_size=0.3)
y_pred = reg.predict(X_pru)
```

El resultado de aplicar la función `train_test_split()` son 4 arreglos de Numpy, donde X_{ent} y y_{ent} constituyen el 70% de los datos a usarse para el entrenamiento y X_{pru} y y_{pru} el otro 30% para las pruebas.

En cuanto a las métricas algunas de las más usadas en tareas de regresión son el error absoluto medio (MAE), el error cuadrático medio (MSE) , el coeficiente de determinación(R^2), entre otras. Veamos cómo es su forma matemática y como se implementan en Scikit-learn:

6.2.1 Error absoluto medio (MAE)

Es el promedio de los errores absolutos de todos los puntos en el conjunto de datos:

$$MAE = \frac{1}{n} \sum_{i=1}^n |y^{(i)} - \hat{y}^{(i)}|$$

```
from sklearn.metrics import mean_absolute_error
mae_pru = mean_absolute_error(y_pru,y_pred)
print("Error absoluto medio {:.2f}".format(mae_pru))
```

Error absoluto medio 0.32

6.2.2 Error cuadrático medio (MSE)

Este es el promedio de los cuadrados de los errores de todos los puntos en el conjunto.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

```
from sklearn.metrics import mean_squared_error
mse_pru = mean_squared_error(y_pru,y_pred)
print("Error cuadrático medio: {:.2f}".format(mse_pru))
```

Error cuadrático medio: 0.29

6.2.3 Coeficiente de determinación (R^2)

Mide la cantidad de varianza de la predicción obtenida por el modelo. Su valor se encuentra dentro del rango 0 y 1, donde un valor cercano a 1 indica una buena calidad del modelo, mientras que un valor cercano a 0 es indicio de que algo anda mal. La fórmula para el coeficiente de determinación es:

$$R^2 = 1 - \frac{\sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^n (y^{(i)} - \mu_y)^2}$$

```
from sklearn.metrics import r2_score
r2_pru = r2_score(y_pru,y_pred)
print("Coeficiente R2: {:.2f}".format(r2_pru))
```

Coeficiente R2: 0.46

Como se puede observar el valor de la métrica del error cuadrático medio estuvo alrededor de 0.29, el coeficiente de determinación fue de aproximadamente 0.46, lo cual nos indica que para ver sido nuestro primer modelo de aprendizaje automático hemos hecho un buen trabajo. De todas maneras, más adelante en el libro exploraremos otras técnicas que nos servirán para mejorar aún más la calidad de nuestros modelos de aprendizaje, como el ajuste de hiperparámetros, la selección de características, reducción de la dimensionalidad, entre otros.

6.2.4 Validación cruzada por k iteraciones

Por último, cabe mencionar otra técnica usada ampliamente para la evaluación de modelos, llamada validación cruzada por k iteraciones (en inglés, *k cross validation*), la cual consiste en dividir aleatoriamente un conjunto de datos en k porciones o pliegues (en inglés, *fold*), de ellos k-1 son usados para entrenar el modelo y el restante para realizar su testeo. Esta acción se ejecuta de manera repetitiva hasta alcanzar un número k de iteraciones o un umbral de rendimiento estimado previamente definido. En este método el número de pliegues generados será igual a la cantidad de iteraciones establecidas. Considere el siguiente ejemplo con un valor de k=5:

Datos	Modelo 1	Modelo 2	Modelo 3	Modelo 4	Modelo 5
P 1	Prueba	Entren.	Entren.	Entren.	Entren.
P 2	Entren.	Prueba	Entren.	Entren.	Entren.
P 3	Entren.	Entren.	Prueba	Entren.	Entren.
P 4	Entren.	Entren.	Entren.	Prueba	Entren.
P 5	Entren.	Entren.	Entren.	Entren.	Prueba

Observamos que se entrena k modelos. El primer modelo se entrena con los conjuntos de entrenamiento del segundo al quinto *pliegue* y se prueba con el conjunto de prueba del pliegue 1. El segundo modelo se entrena con los conjuntos de entrenamiento de los pliegues 1,3,4,5 y se prueba con el conjunto de prueba del pliegue 2, y así sucesivamente. A cada modelo se le aplica su evaluación usando la métrica R^2 , cuyos valores al final son promediados.

La ventaja de la validación cruzada por k iteraciones se debe al hecho de usar todos los puntos de datos tanto para entrenar como para validar, lo cual supone una mejora en comparación a otros métodos como `train_test_split()` que separa la data original en dos subconjuntos claramente definidos, lo cual evidentemente conlleva a entrenar y probar con menos datos de los que se tienen en un principio, esto hace a la validación cruzada sobresaliente porque ayuda a que el modelo no resulte altamente dependiente a algunas muestras del conjunto de entrenamiento, evitando así el sobreajuste. Como consideración adicional tenga en cuenta que para conjuntos de datos pequeños comúnmente se usa un valor de k igual a 10, mientras que, para conjuntos de datos mucho más grandes, 5 podría ser un valor adecuado para este hiperparámetro.

En Scikit-learn el valor por defecto de k es 3, miremos como podemos aplicar la validación cruzada con todas las muestras de nuestro conjunto de datos Boston Housing con un valor de $k = 5$ ($cv=5$):

```
from sklearn.model_selection import cross_val_score

X = df[['sqft_living']].values
y = df['price'].values

vals = cross_val_score(reg, # modelo
                      X, # atributos descriptivos
                      y, # atributo destino
                      cv=5 # k (cantidad de subconjuntos)
                      )
print("Valores validación cruzada: %s" % vals)
print("Media:{:.3f} Desviación estándar:{:.3f}".format(np.mean(vals), np.std(vals))
))
```

Valores validación cruzada: [0.49431685 0.49246653 0.51222568 0.50080369
0.01756058]
Media: 0.403 Desviación estándar: 0.193

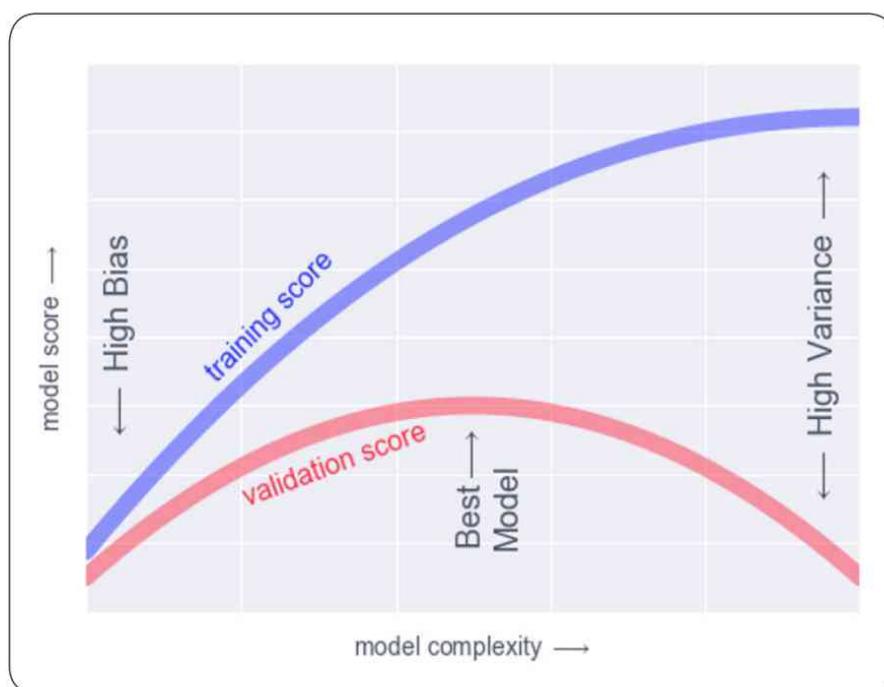
La función `cross_val_score()` devuelve un array con los coeficientes de determinación R^2 aplicados en cada iteración. La calidad del modelo se obtiene calculando la media de estos valores, en nuestro caso resultó ser 0.403; si bien es un resultado regular, no es del todo malo por tratarse de nuestro primer acercamiento a las técnicas de aprendizaje automático.

Existe un tipo especial de validación cruzada que puede consultar el lector si desea, la llamada validación cruzada dejando uno afuera, donde se establece el

valor de los subconjuntos igual a la cantidad de muestras de entrenamiento, permitiendo que una muestra de este tipo sea usada para probar el modelo en cada iteración realizada.

6.3. Curvas de aprendizaje y validación

Una de las claves en machine learning es encontrar un punto óptimo en el equilibrio entre el sesgo y la varianza. Sabemos que un sesgo alto significa un problema de subajuste que se traduce en la incapacidad del modelo para ajustar los datos, por otro lado, una varianza alta es sinónimo de sobreajuste lo cual implica tener un modelo complejo, tan ajustado a los datos de entrenamiento que no es capaz de generalizar con datos nuevos. La siguiente gráfica nos muestra cómo se comportan la exactitud de entrenamiento y de validación en relación a la complejidad del modelo, vemos que un modelo complejo o sobreajustado tiene una alta exactitud cuando se determina con los datos de entrenamiento, pero esta disminuye con los de validación, además señala también que en el punto más alto de la curva de validación es donde se encuentra el mejor modelo posible.



Fuente: Puntaje de validación vs puntaje de entrenamiento¹

¹ <https://jakevdp.github.io/PythonDataScienceHandbook/05.03-hyperparameters-and-model-validation.html>

En este apartado estudiaremos dos técnicas muy populares: Las curvas de validación y de aprendizaje que nos ayudarán a saber cómo es el comportamiento de un modelo en este sentido.

Curvas de validación

Las curvas de validación permiten conocer el efecto que tiene un parámetro en la exactitud de validación.

Consideremos el siguiente ejemplo sobre un conjunto de datos de emisiones de gas metano en el período comprendido entre 1.970 y 2.008. Si dibujamos estos datos veríamos que siguen una tendencia a simple vista polinómica.

```
df3 = pd.read_csv("emisiones2.csv")
pl.scatter(df3['anio'], df3['cantidad'], color='r')
pl.xlabel("año")
pl.xlim(1965, 2010)
pl.ylabel("emisión")
pl.show()
```



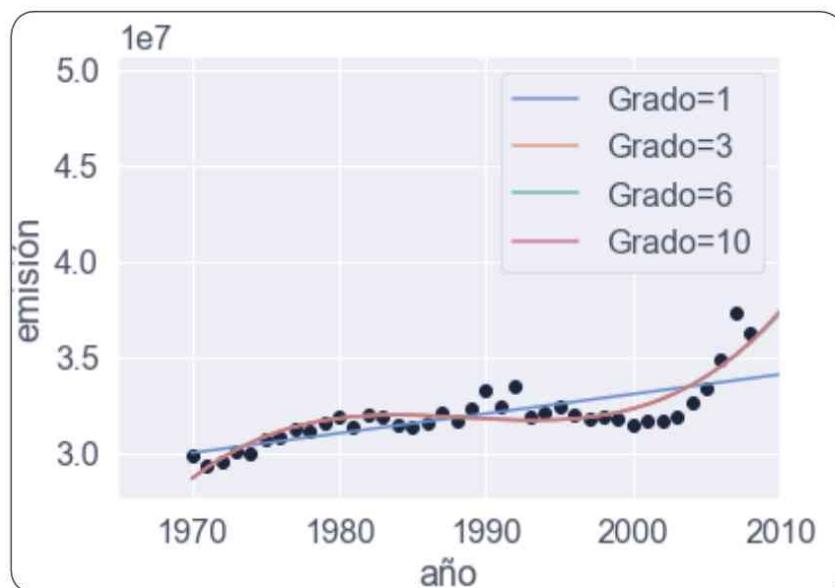
Ahora bien, podemos intentar con polinomios de varios grados para ver cual representa un mejor ajuste.

```

from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
X_test = np.arange(1970, 2020, 1)
X = df3['anio'].values.reshape(-1,1)
y = df3['cantidad'].values.reshape(-1,1)
pl.scatter(X, y, color='black')

for grado in [1, 3, 6, 10]:
    poli = PolynomialFeatures(degree = grado )
    X_p = poli.fit_transform(X)
    modelo = LinearRegression().fit(X_p, y)
    to_predict = poli.fit_transform(X_test.reshape(-1,1))
    y_pred = modelo.predict(to_predict).reshape(1,50)[0]
    pl.plot(X_test, y_pred, label='Grado={0}'.format(grado))
pl.legend(loc='best');
pl.xlabel("año")
pl.xlim(1965, 2010)
pl.ylabel("emisión")

```



Podemos notar que la complejidad de este modelo está determinada por el grado del polinomio, por tanto, el objetivo es encontrar cual es el grado que brinde un mejor equilibrio entre el sesgo y la varianza. Tal situación la podemos solventar aplicando una curva de validación que nos ayude a determinar cuál es el grado que representa el mejor puntaje de validación para el modelo. En Scikit podemos hacer

esta tarea con la función `validation_curve()`. Este método recibe básicamente: Un estimador (objeto predictor); los datos de entrenamiento X e y; el hiperparámetro a ir modificando (grado); un rango de valores para el hiperparámetro; cantidad de iteraciones de validación cruzada(`cv=10`). La ejecución devolverá dos arreglos con los puntajes de exactitud de entrenamiento y validación respectivamente en cada una de las 10 iteraciones realizadas para cada grado del polinomio.

Notará que se ha usado un *pipeline* para simplificar un poco el trabajo. Un pipeline o tubería es una secuencia de objetos transformadores que finaliza en un objeto estimador, en nuestro caso tenemos `StandardScaler`, `PolynomialFeatures` y un `LinearRegression`, tenga en cuenta que al entrenar una tubería con el método `fit()` se invocará en secuencia a los métodos `fit_transform()` de los objetos que hacen parte de ella salvo el último.

```
X_ent, X_pru, y_ent, y_pru = train_test_split(X, y, test_size=0.2, random_
state=1)

grados = [1, 2, 3, 6, 8, 10, 20]
# creamos un pipeline para encadenar el trabajo con PolynomialFeatures y
LinearRegression
pipe = Pipeline([('scaler', StandardScaler()), ('poly',
PolynomialFeatures(degree=2)),
('regr', LinearRegression())])
train_scores, test_scores = validation_curve(pipe, X_ent, y_ent,
'poly_degree', grados, cv=10)
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

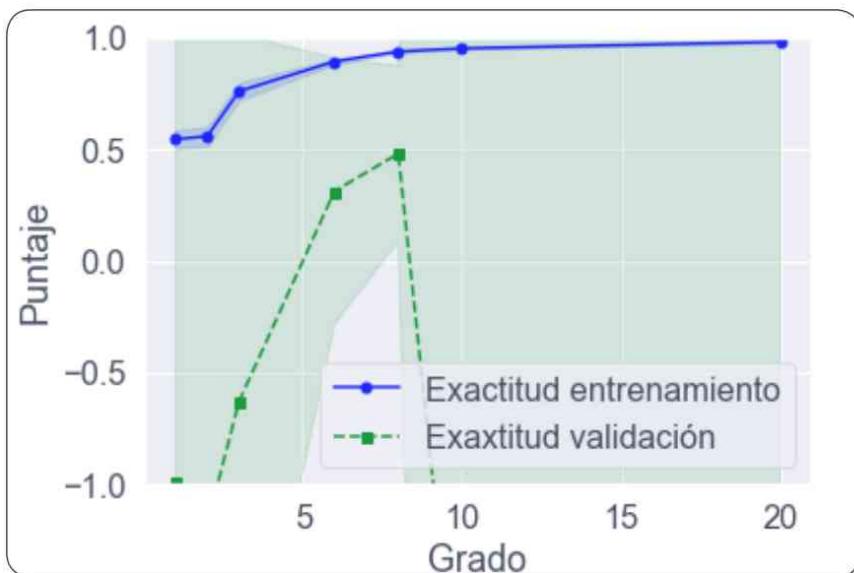
pl.plot(grados, train_mean,
color='blue', marker='o',
markersize=5, label='Exactitud entrenamiento')

pl.fill_between(grados, train_mean + train_std,
train_mean - train_std, alpha=0.15,
color='blue')
pl.plot(grados, test_mean,
color='green', linestyle='--',
marker='s', markersize=5,
label='Exactitud validación')
```

```

pl.fill_between(grados,
                test_mean + test_std,
                test_mean - test_std,
                alpha=0.15, color='green')
pl.legend(loc='best')
pl.ylim(0, 1)
pl.xlabel('Grado')
pl.ylabel('Puntaje')

```

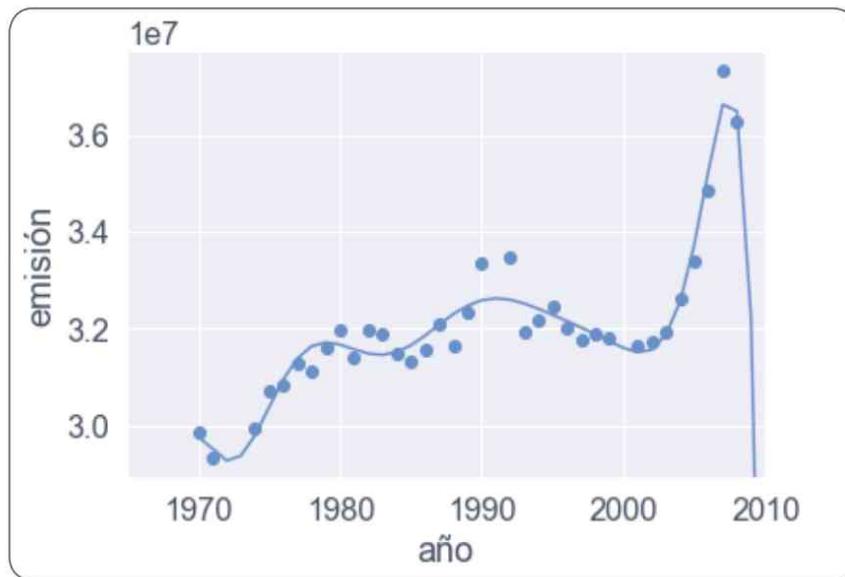


La gráfica nos muestra que nuestro modelo está sobreajustado, probemos con un polinomio de grado 8 para ver cómo se ajusta a los datos. En el siguiente subapartado veremos un algoritmo llamado búsqueda de cuadrículas que nos ayudará a saber así como en casos como este cuál es el hiperparámetro óptimo.

```

pl.scatter(X_ent.ravel(), y_ent)
lim = pl.axis()
pipe = Pipeline([('scaler', StandardScaler()), ('poly', PolynomialFeatures(degree=8)),
                 ('regl', LinearRegression())])
y_test = pipe.fit(X_ent, y_ent).predict(X_test.reshape(-1,1))
pl.plot(X_test.ravel(), y_test);
pl.axis(lim);
pl.xlabel("año")
pl.xlim(1965, 2010)
pl.ylabel("emisión")

```



Curvas de aprendizaje

Las curvas de aprendizaje son otra herramienta útil que muestran como es el rendimiento del aprendizaje de un modelo en función del incremento de muestras del conjunto de entrenamiento. La función que ofrece Scikit para este propósito es `learning_curve()` que recibe en general los mismos parámetros que la función `validation_curve()`.

```
from sklearn.model_selection import learning_curve
train_sizes, train_scores, test_scores = learning_curve(estimator=pipe,
                                                       X=X_ent,
                                                       y=y_ent,
                                                       train_sizes=np.linspace(0.1, 1.0, 10),
                                                       cv=10)
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)
pl.plot(train_sizes, train_mean,
        color='blue', marker='o',
        markersize=5, label='Exactitud entrenamiento')
pl.fill_between(train_sizes,
                train_mean + train_std,
                train_mean - train_std,
                alpha=0.15, color='blue')
```

```
pl.plot(train_sizes, test_mean,
        color='green', linestyle='--',
        marker='s', markersize=5,
        label='Exactitud validación')

pl.fill_between(train_sizes,
                test_mean + test_std,
                test_mean - test_std,
                alpha=0.15, color='green')

pl.grid()
pl.xlabel('Número de muestras de entrenamiento')
pl.ylabel('Exactitud')
pl.legend(loc='lower right')
pl.ylim([-0.5, 1.1])
pl.tight_layout()
pl.show()
```



6.4. Técnica de búsqueda de cuadrículas para el ajuste de hiperparámetros

Un aspecto crítico en aprendizaje automático es encontrar los valores de los hiperparámetros que brinden al modelo una óptima compensación entre el sesgo y la varianza. Cuando estudiamos las curvas de validación lo que tratamos de hacer fue ir probando un rango de hiperparámetros al tiempo que íbamos aplicando una métrica para saber cuál era el que mejor rendimiento ofrecía. Sin embargo, con el algoritmo de búsqueda de cuadrículas podemos tener un mejor conocimiento de nuestro espacio de hiperparámetros y saber cuál es el hiperparámetro o la combinación de ellos que aporten mejor rendimiento o exactitud al modelo. Scikit ofrece dos algoritmos en este sentido: GridSearchCV y RandomizedSearchCV, el primero hace un barrido completo de las combinaciones de los hiperparámetros, mientras que el segundo lo hace por medio de la selección de una combinación aleatoria de los mismos. Se mostrará un ejemplo del uso de GridSearchCV dejando al lector la tarea de indagar sobre la clase RandomizedSearchCV, disponible por supuesto en la documentación oficial de la librería.

```
from sklearn.model_selection import GridSearchCV
model = Pipeline([('scaler', StandardScaler()), ('poly', PolynomialFeatures()),
                  ('regl', LinearRegression())])

parms = {'poly_degree': np.arange(2, 20)}

gscv = GridSearchCV(model, parms, cv=10, scoring='neg_mean_squared_error')
gscv.fit(X_ent.reshape(-1,1),y_ent)

space = np.linspace(1970,2008,50).reshape(-1,1)

est_deg= gscv.best_params_['poly_degree']

pl.scatter(X_ent.ravel(), y_ent)
pl.plot(space, gscv.predict(space), color ='red')
pl.title(f'Grado estimado:{est_deg}')
pl.xlabel("año")
pl.xlim(1965, 2010)
pl.ylabel("emisión")
```



La función GridSearchCV recibe un estimador, en nuestro caso un objeto pipeline con los objetos transformadores ya conocidos, un diccionario con el hiperparámetro y sus posibles valores, número de iteraciones de la estrategia de validación cruzada y la métrica de evaluación. Note que el método best_params_() nos proporciona el mejor valor estimado para nuestro hiperparámetro, que viene siendo grado = 4 en nuestro ejemplo.

<https://dogramcode.com/bloglibros>



CAPÍTULO 7

Modelos de Clasificación I

Temas

- 7.1 Perceptrón
- 7.2 Neurona lineal adaptativa (Adaline)
- 7.3 Regresión logística
- 7.4 Métricas de evaluación
- 7.5 Máquinas de vectores de soporte (SVM)

La clasificación es otra subárea del aprendizaje automático enfocada en la creación de modelos a partir de un conjunto de muestras etiquetadas y permite predecir a qué clase pertenece una muestra nueva. La clase es una variable discreta recomendablemente de tipo entero comprendida en un rango finito de valores. En este sentido, se considera clasificación binaria si tiene dos valores posibles, ternaria si tiene tres valores, y así sucesivamente hasta convertirse en clasificación multiclas.

En este capítulo veremos diferentes algoritmos de clasificación usados en diferentes problemas de la vida real. El primero con el que iniciaremos este viaje será el Perceptrón, predecesor de las neuronas artificiales, las cuales se utilizan ampliamente en el campo del aprendizaje profundo y sobre las que profundizaremos en secciones posteriores del libro. Más adelante también abordaremos otros algoritmos muchos más robustos que el Perceptrón, ampliamente utilizados en el campo investigativo y de la industria.

7.1 Perceptrón simple

El concepto de neurona artificial está basado en el concepto de neurona biológica, consiste en un sistema que necesita un estímulo por parte del exterior típicamente de otra neurona, realiza una acción con la entrada y produce una señal de salida con la cual activa a la neurona adyacente.

El Perceptrón es el modelo que tiene la estructura más simple en el ámbito de las redes neuronales. Este algoritmo parte de la multiplicación entre una matriz de entradas X , y un vector de pesos w , para luego hacer una suma de todos esos productos, este resultado será una variable que llamaremos z (función de entrada), la cual pasará a una función de activación como puede ser una función escalón donde se compara con un umbral θ para determinar si se activa o no la neurona siguiente.

En este sentido la función de entrada la podemos representar matemáticamente como:

$$z = \sum_{i=1}^m w_i * x_i$$

En este tipo de modelos los pesos juegan un papel muy importante porque mediante ellos se consigue amplificar o atenuar la señal de salida.

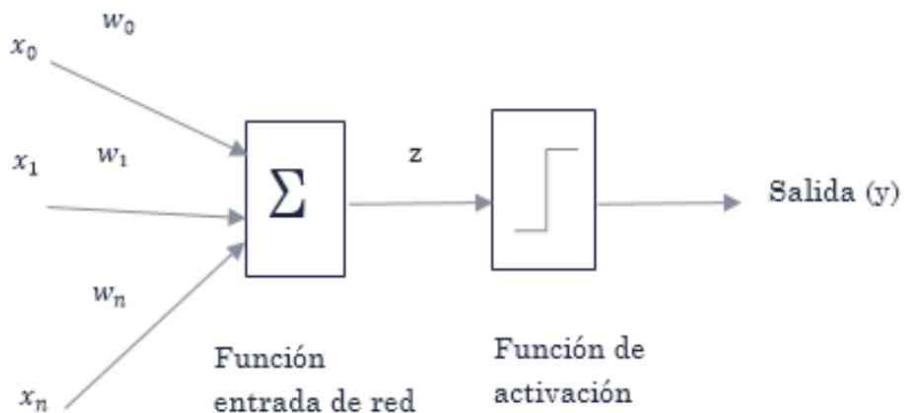
En presencia de un problema de clasificación binaria, una vez se tiene el valor de z , este pasa por una función de activación $h(z)$ que lo compara con un umbral. Si z es mayor o igual a 0 la función devuelve 1, lo cual significa que la muestra pertenece a una de las clases. Mientras que, en cualquier otro caso ($z < 0$), la función de activación devuelve -1, indicando que la muestra pertenece a la otra clase posible. Durante el entrenamiento este valor retornado debe compararse con los valores reales para calcular el error de la predicción y actualizar los pesos.

$$h(z) = \begin{cases} 1, & z \geq 0 \\ -1, & \text{otro caso} \end{cases}$$

La función de entrada de red es expresada de la siguiente manera:

$$z = w_0 x_0 + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m = \sum_{i=1}^m w_i * x_i = w^T X$$

Y el Perceptrón se puede representar graficamente así:



Para que la tarea de clasificación se pueda llevar a cabo durante el entrenamiento se deben encontrar los pesos que minimicen el error, esto se puede lograr mediante un método como el descenso del gradiente visto en el capítulo 5, por medio del cual se actualicen los pesos de manera iterativa. La siguiente fórmula se podría emplear para tal fin.

$$w_j = w_j - \eta(y_j - \hat{y}_j) \cdot x_i$$

De lo anterior, η es la tasa de aprendizaje, un parámetro usado con mucha frecuencia en algoritmos de optimización para ajustar la velocidad del aprendizaje. Hay que ser muy precavidos con este hiperparámetro, porque en general, un valor muy pequeño puede hacer que el algoritmo tarde mucho tiempo para llegar a alcanzar un mínimo local, mientras que, un valor muy grande puede provocar que el modelo no converja porque los pasos efectuados son tan grandes que inducen a saltarse o pasarse el mínimo, causando fluctuaciones que impiden alcanzar el punto óptimo.

El algoritmo anterior se puede resumir de la siguiente forma:

1. Inicialización de los pesos w_j con números aleatorios muy pequeños positivos o negativos.
2. Realizar un entrenamiento mediante un recorrido con k iteraciones o hasta que todas las salidas sean correctas
 - Recorremos cada vector de entrada calculando la activación para cada neurona j usando la función de activación h :

$$z = \sum_{i=1}^m w_i * x_i , \quad h(z) = \begin{cases} 1, & z \geq 0 \\ -1, & \text{otro caso} \end{cases}$$

Luego actualizar cada peso individualmente, mediante:

$$w_j = w_j - \eta(y_j - \hat{y}_j) \cdot x_i$$

Miremos como podemos implementar este algoritmo en Python para que aprenda la función lógica or:

x_1	x_2	y
1	1	1
0	1	1
1	0	1
0	0	0

Primero definimos los datos de entrenamiento e inicializamos un array para los pesos a 0.

```
import numpy as np
from matplotlib import pyplot as plt
errors = []

X = np.array( [[1, 1],
               [0, 1],
               [1, 0],
               [0, 0]])
y = np.array([1, 1, 1, 0])

#inicializamos los pesos
pesos = np.zeros(X.shape[1] + 1)
```

Definimos las otras funciones necesarias para realizar el entrenamiento:

```
# Este método calcula un valor para y basado en los pesos
def obtener_entrada_red(x, pesos):
    prediccion = x.dot(pesos[1:]) + pesos[0]
    return prediccion

# Este método permite predecir la clase de una instancia
def calcular_prediccion(X, pesos):
    z = obtener_entrada_red(X, pesos)
    if (z>=0.0):
```

```

activacion = 1
else:
    activacion = 0
return activacion

# Este método implementa el algoritmo del descenso del gradiente
def entrenar(X, y, pesos, max_iter, tasa_aprendizaje):
    for i in range(max_iter):
        error = 0
        for xi, yi in zip(X, y):
            prediccion = calcular_prediccion(xi, pesos)
            error = yi - prediccion
            actualizacion = tasa_aprendizaje * error
            pesos[1:] += actualizacion * xi
            pesos[0] += actualizacion
        errors.append(error)
    return pesos

```

Llamamos la función entrenar y mostramos los pesos obtenidos:

```

pesos = entrenar(X, y, pesos, max_iter=10, tasa_aprendizaje=0.01)
print(pesos)

```

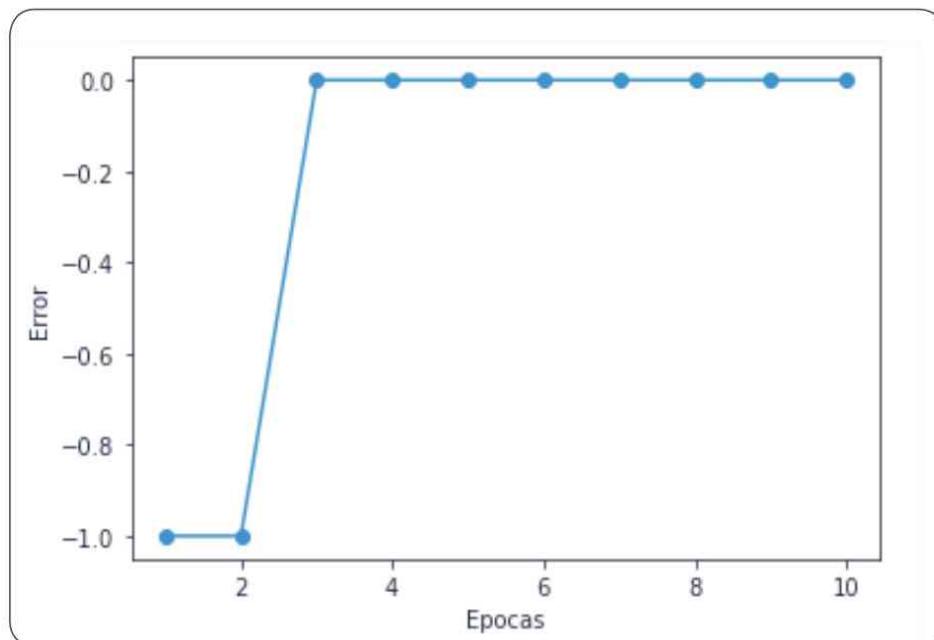
`[-0.01 0.01 0.01]`

Con una tasa de aprendizaje de 0.01 y tan solo 10 iteraciones nuestro Perceptrón alcanza la convergencia, esto lo podemos verificar si escribimos el siguiente fragmento de código con el cual creamos una gráfica del error vs el número de épocas:

```

plt.plot(range(1, len(errors) + 1), errors, marker='o')
plt.xlabel('Epocas')
plt.ylabel('Error')
plt.show()

```

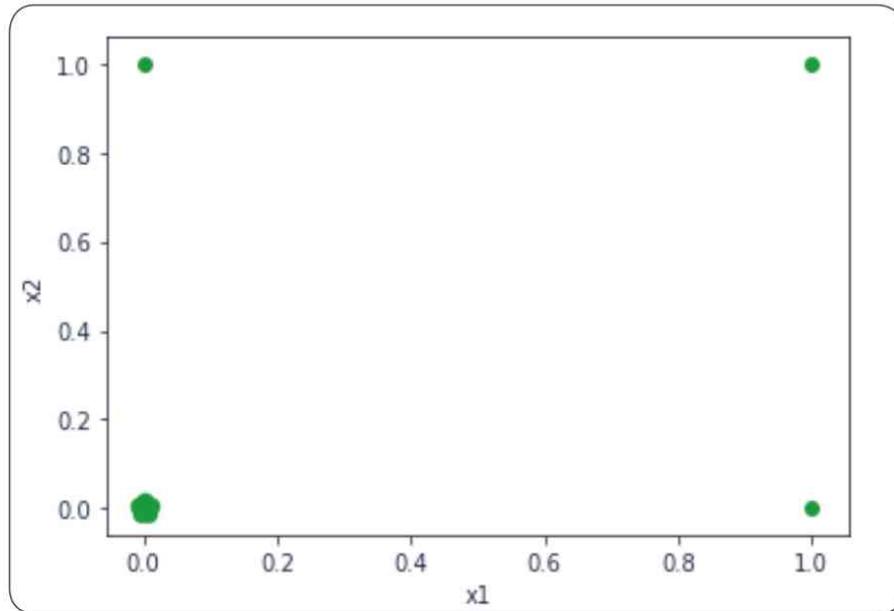


Finalmente probamos nuestro Perceptrón con dos muestras de prueba, verificando que la respuesta ofrecida coincide con el resultado de aplicar la función lógica or:

```
muestra1 = np.array([1,1])
p1 = calcular_prediccion(muestra1, pesos)
print("Predicción {:.2f}".format(p1)) # resultado: 1

muestra2 = np.array([0,0])
p2 = calcular_prediccion(muestra2, pesos)
print("Predicción {:.2f}".format(p2)) # resultado: 0
```

Es necesario remarcar que para un buen desempeño del Perceptrón los puntos de datos etiquetados con una clase deben poder separarse linealmente de los puntos de la otra clase. Considere el siguiente ejemplo donde se puede evidenciar este aspecto en la función lógica *or*. La línea que separa las instancias de una clase de otra se denomina **límite de decisión**.



La buena noticia es que Scikit-Learn dispone de una clase que implementa este algoritmo y se llama precisamente Perceptron. Miremos como podemos entrenar un clasificador con esta clase utilizando los mismos parámetros que hemos manejado en el ejercicio pasado.

```
from sklearn.linear_model import Perceptron
clf = Perceptron(random_state=1, alpha=0.01, max_iter=10)
clf.fit(X, y)
```

Y ahora calculemos la exactitud del clasificador y vamos a predecir la respuesta para una muestra de prueba, con lo cual notaremos que el modelo está trabajando satisfactoriamente.

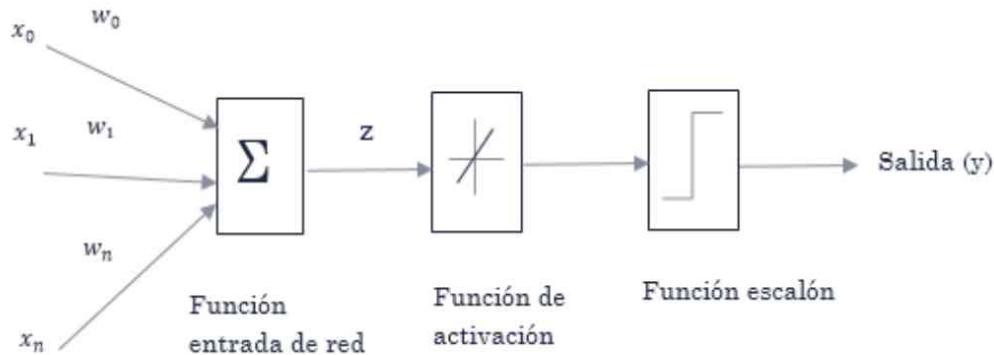
```
print("Exactitud: {:.2f}".format(clf.score(X, y))) # Exactitud: 1.00
muestra1 = np.array([[0,0]])
print("Predicción: {:.2f}".format(clf.predict(muestra1)[0])) # Predicción: 0.00
```

7.2 Neurona lineal adaptativa (ADALINE)

Ahora veremos otro método de red neuronal de una sola capa considerado una mejora al anterior, y es la neurona lineal adaptativa (ADALINE) que a diferencia del Perceptrón, donde la función de activación es una función escalón, en Adaline

es una función lineal, cuya salida se resta a la etiqueta de clase verdadera para calcular el error e ir actualizando los pesos.

La siguiente gráfica muestra el modo de operación de Adaline:



Otra diferencia importante entre el Perceptrón y Adaline es que con el segundo se actualizan todos los pesos de las muestras de entrenamiento al mismo tiempo en cada iteración. Además, esta función lineal que ciertamente podemos considerarla como una función de costo, se puede minimizar mediante la aplicación del algoritmo del descenso del gradiente para encontrar los pesos óptimos.

La función de costo recordemos es:

$$J(w) = \frac{1}{2} \sum_{i=1}^n (y_i - \hat{y})^2, \text{ donde } \hat{y} = w_0 + w_i x_i$$

Entonces para minimizarla usaremos el descenso del gradiente, que recordemos representa la derivada de J respecto a w:

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \left(\frac{1}{2} \sum_{i=1}^n (y_i - \hat{y})^2 \right)$$

$$\frac{\partial J}{\partial w_j} = - \sum_{i=1}^n (y_i - \hat{y}) x_i$$

La actualización de los pesos se consigue multiplicando la tasa de aprendizaje (η) con el gradiente ($\nabla J(w)$):

$$\Delta_{w_n} = -\eta \cdot \nabla J(w) = \eta \cdot \sum_{i=1}^n (y_i - \hat{y}) x_i$$

En Python podemos implementar fácilmente nuestra versión de Adaline. Siendo el proceso parecido al del Perceptrón, Solo debemos seguir los siguientes pasos:

1. Inicializar los pesos con valores cercanos a 0 (positivos o negativos).
2. Realizar un entrenamiento mediante un recorrido con k iteraciones donde vamos calculando el valor de la función entrada de red (z), calculamos los errores, diferencia entre el resultado de la entrada de red y la salida esperada
3. Actualizamos los pesos con la fórmula:

$$w_j = w_j - \eta(y_j - \hat{y}_j) \cdot x_i$$

Un detalle muy importante que ayuda mucho para que el descenso del gradiente minimice la función costo y se logre la mencionada convergencia consiste en escalar los atributos descriptivos del conjunto de datos. En este sentido la normalización puede resultar una buena opción.

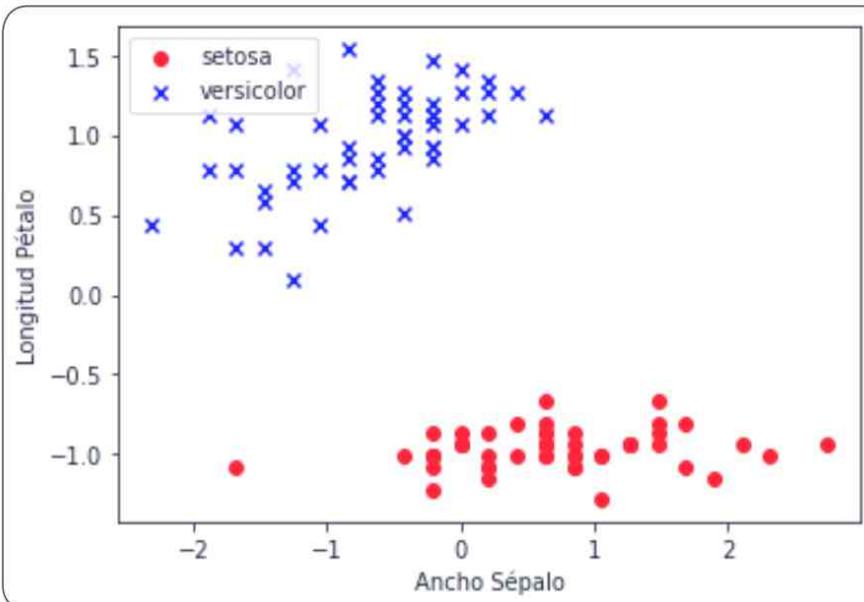
Teniendo en cuenta lo anterior, es hora de implementar un clasificador basado en el algoritmo Adaline. Tomaremos el conjunto de datos llamado Iris incluido dentro del submódulo datasets de Scikit-Learn. Este dataset cuenta con 150 muestras de tres clases de especies de flores: Iris-setosa, Iris-versicolor e Iris-virginica. Para este ejercicio nada más tomaremos las primeras 100 instancias que corresponden con las dos primeras clases. Ya que nuestro objetivo es entrenar un clasificador binario que determine a que especie de flor (Iris-setosa o Iris-versicolor) pertenece una muestra particular.

Los atributos del conjunto de datos iris son los siguientes:

SepalLengthCm (Numérico)	Longitud del sépalo
SepalWidthCm (Numérico)	Ancho del sépalo
PetalLengthCm (Numérico)	Longitud del pétalo
PetalWidthCm (Numérico)	Ancho del pétalo
Species (Texto)	Iris-setosa, Iris-versicolor e Iris-virginica

Antes de entrenar nuestro Adaline con estos datos de entrenamiento y mirar si es posible clasificar nuevas instancias. Miremos como podemos cargar y visualizar en una gráfica de dispersión las primeras 100 instancias, considerando solo las dos primeras columnas (ancho del sépalo y longitud del pétalo). Notará el lector que se han tomado solo estas dos columnas por simplicidad en la visualización de los datos.

```
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target
X = X[0:100 , [1,2]]
y = y[0:100]
# estandarizamos los datos
X[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
plt.scatter(X[:50, 0], X[:50, 1], color='red', marker='o', label='setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],color='blue', marker='x', label='versicolor')
plt.xlabel('Ancho Sépalo')
plt.ylabel('Longitud Pétalo')
plt.legend(loc = 'upper left')
plt.show()
```



Como se puede apreciar en la imagen anterior los puntos de datos pueden ser separados linealmente por un hiperplano lo cual nos garantiza un adecuado funcionamiento del Adaline.

```

#inicializamos los pesos
rgen = np.random.RandomState(1)
pesos = rgen.normal(loc=0.0, scale=0.01,size=1 + X.shape[1])
errors = []
costos = []

# Este método calcula un valor para y basado en los pesos
def obtener_entrada_red(x, pesos):
    z = x.dot(pesos[1:]) + pesos[0]
    return z

def activacion(X,pesos):
    return obtener_entrada_red(X, pesos)

# Este método permite predecir la clase de una instancia
def calcular_prediccion(X, pesos):
    a = activacion(X,pesos)
    if (a>=0.0):
        res = 1
    else:
        res = 0
    return res

# Este método implementa el algoritmo del descenso del gradiente
def entrenar(X, y, pesos, max_iter, tasa_aprendizaje):
    for i in range(max_iter):
        z = obtener_entrada_red(X, pesos)
        error = y - z
        pesos[1:] += tasa_aprendizaje * X.T.dot(error)
        pesos[0] += tasa_aprendizaje * error.sum()
        costo = (error**2).sum() / 2
        costos.append(costo)
    return pesos

```

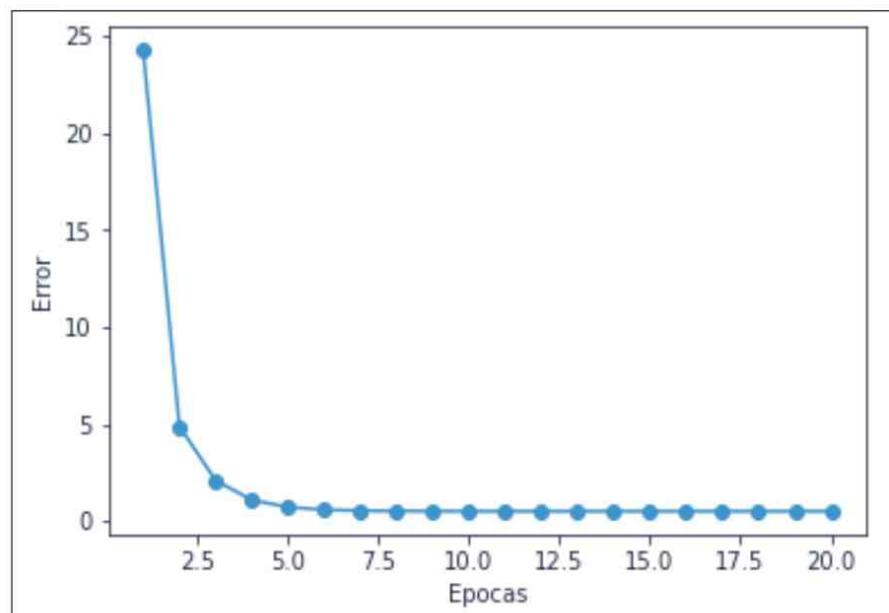
Ahora si llamamos nuestro método `entrenar()` con una tasa de aprendizaje de 0.01 y 10 iteraciones obtendremos los pesos óptimos. Lo cual es fácil de verificar si dibujamos una gráfica de dispersión de los costos calculados en cada iteración.

```

pesos = entrenar(X, y, pesos, max_iter=10, tasa_aprendizaje=0.01)
plt.plot(range(1, len(costos) + 1), costos, marker='o')
plt.xlabel('Epocas')
plt.ylabel('Error')
plt.show()

```

La gráfica muestra cómo la función de costo converge rápidamente con tan solo 10 iteraciones.



Desafortunadamente Scikit-Learn no cuenta con una implementación del algoritmo Adaline, pero sí de otros potentes clasificadores que estudiaremos en los siguientes apartados. Adicionalmente estudiaremos como validarlos mediante diferentes técnicas como la matriz de confusión, curvas ROC, entre otros.

7.3 Regresión logística

La regresión logística es uno de los clasificadores probabilísticos con mayor popularidad en el mundo del aprendizaje automático, ya que por ejemplo además de ser fácil de entender, se puede usar para el trabajo con redes neuronales.

Este clasificador recibe una entrada z que es un modelo lineal ($w_0X_0 + w_1X_1 + w_2X_2 + \dots + w_mX_m$) y devuelve una salida comprendida en el rango $[0, 1]$ que viene siendo la probabilidad de una muestra de pertenecer a una clase o a otra. Por ejemplo,

si esta salida es mayor o igual a 0.5 la muestra se puede clasificar como A y si es menor a este umbral se clasifica como B. Este valor de la probabilidad se consigue mediante la función *sigmoide* o función logistica. La cual como ya sabemos se define matemáticamente así:

$$y = \frac{1}{1 + e^{-z}}$$

Lo dicho anteriormente se puede expresar como:

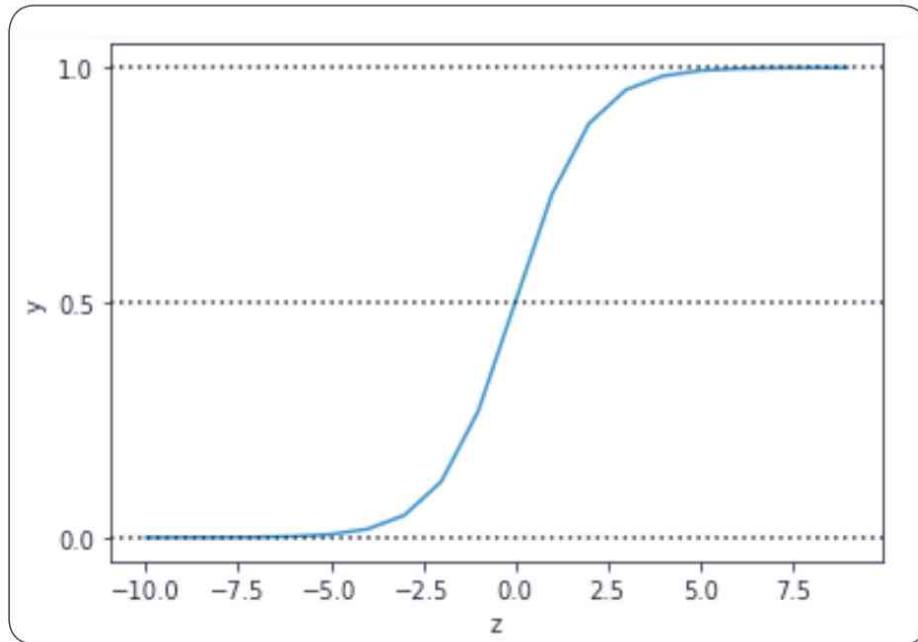
$$\hat{y} = P(y = 1 | x) = \frac{1}{1 + e^{-z}}$$

Recordemos primeramente como es la función sigmoide ya mostrada en un capítulo anterior.

```
import numpy as np
import matplotlib.pyplot as plt

def sigmoide(x):
    return 1. / (1. + np.exp(-x))

z = np.arange(-10, 10, 1)
y = sigmoide(z)
plt.plot(z, y)
plt.axhline(y=0, ls='dotted', color='k')
plt.axhline(y=0.5, ls='dotted', color='k')
plt.axhline(y=1, ls='dotted', color='k')
plt.yticks([0.0, 0.5, 1.0])
plt.xlabel("z")
plt.ylabel("y")
plt.show()
```



Acerca de la función anterior podemos deducir que si los valores de x son muy grandes ($+\infty$) la salida se aproximará a 1, así mismo si los valores de x son muy pequeños ($-\infty$) la salida se acercará a 0, y si x es igual a 0 la respuesta será 0.5.

Así como hacíamos con otras técnicas ya vistas debemos encontrar los pesos óptimos de nuestro modelo mediante la minimización de una función de costo. Por comodidad se usará la siguiente función:

$$J(w) = \frac{1}{m} \sum_{i=1}^m - [y^{(i)} \log(\hat{y}(x^{(i)})) + (1 - y^{(i)}) \log(1 - \hat{y}(x^{(i)}))]$$

Luego para una muestra cualquiera tenemos:

$$J(w) = [y^{(i)} \log(\hat{y}(x^{(i)})) - (1 - y^{(i)}) \log(1 - \hat{y}(x^{(i)}))]$$

Si $y^{(i)} = 1$, se cancela el segundo término y nos queda: $-\log(\hat{y}(x^{(i)}))$.

Si $y^{(i)} = 0$, se cancela el primer término y nos queda: $-\log(1 - \hat{y}(x^{(i)}))$.

Observemos como podemos programar esto en Python utilizando el algoritmo descenso del gradiente sobre un nuevo conjunto de datos llamado Breast Cancer Wisconsin, el cual contiene 30 características extraídas de 569 imágenes digitalizadas de biopsias por aspiración de aguja fina (FNA) de masas mamarias.

Este dataset describe características de los núcleos celulares presentes en la imagen y es usado para clasificación binaria por lo que tiene una etiqueta destino con valores 1: benigno y 0: maligno.

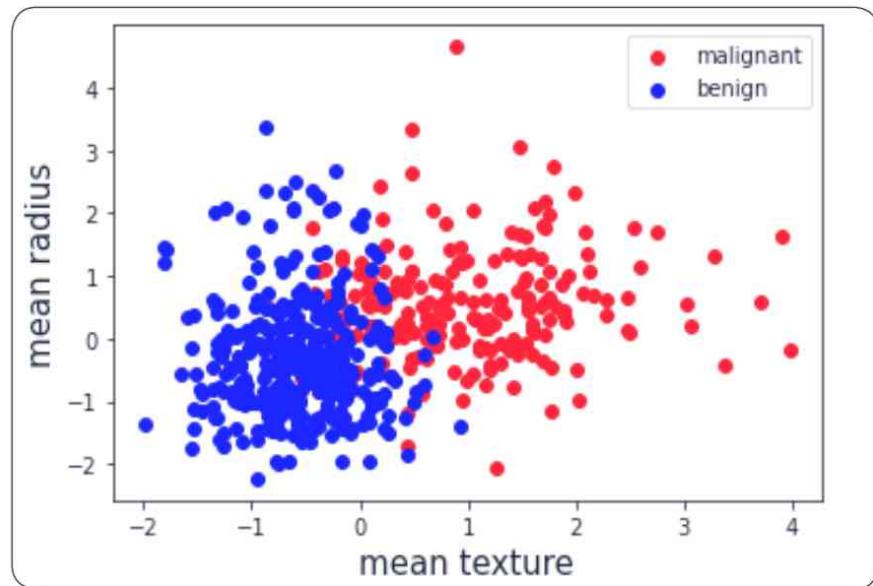
Como siempre lo primero es cargar el dataset localizado en el submódulo datasets:

```
from sklearn import datasets
dataset = datasets.load_breast_cancer()
```

Seguidamente vamos a graficar las variables *mean radius* y *mean texture*:

```
X = dataset.data
y = dataset.target
X = X[:, [1,2]] # tomamos mean radius y mean texture
y = y[:,]
# estandarizamos los datos
X[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()

BENIGNO = 0
MALIGNO = 1
for target,color in zip(range(2),['r','b']):
    plt.scatter(X[[y==target, MALIGNO]], X[[y==target,BENIGNO]],
                color=color, label=dataset.target_names[target])
plt.xlabel(dataset.feature_names[MALIGNO], fontsize=15)
plt.ylabel(dataset.feature_names[BENIGNO], fontsize=15)
plt.legend(prop={'size':10})
plt.show()
```



La gráfica nos dice que si el tumor crece en textura significa mayor posibilidad de ser maligno, esto se puede intuir porque los puntos etiquetados como malignos son los que están ubicados en la parte derecha de la figura.

```

import numpy as np
#inicializamos los pesos
rgen = np.random.RandomState(1)
pesos = []
def sigmoide(x):
    return 1. / (1. + np.exp(-x))
# Este método calcula un valor para y basado en los pesos
def calcular_prediccion(x, pesos):
    z = x.dot(pesos[1:]) + pesos[0]
    prediccion = sigmoide(z)
    return prediccion
# Este método permite calcular los valores de J
def calcular_costo(X, y, pesos):
    prediccion = calcular_prediccion(X, pesos)
    costo = np.mean(-y * np.log(prediccion) - (1 - y) * np.log(1 - prediccion) )
    return costo
# Este método implementa el algoritmo del descenso del gradiente
def entrenar(X, y, pesos, max_iter, tasa_aprendizaje):
    pesos = rgen.normal(loc=0.0, scale=0.01,size=1 + X.shape[1])
    for i in range(max_iter):
        prediccion = calcular_prediccion(X, pesos)

```

```

error = y - predicción
pesos[1:] += tasa_aprendizaje * X.T.dot(error)
pesos[0] += tasa_aprendizaje * error.sum()
# imprimimos el costo cada 10 iteraciones para apreciar como disminuye
if i % 10 == 0:
    print("Costo: %.2f" % calcular_costo(X, y, pesos))
return pesos

```

Luego invocamos el método entrenar pasándole el vector con los pesos generados aleatoriamente, y definimos 200 iteraciones con una tasa de aprendizaje de 0.01.

```

pesos = entrenar(X, y, pesos, 200, 0.01)
print("Pesos:", pesos)

```

```

Costo: 0.28
Costo: 0.24
Costo: 0.24
Costo: 0.24
Costo: 0.24
...
Costo: 0.24
Costo: 0.24
Costo: 0.24
Costo: 0.24
Pesos: [ 0.69917406 -0.95580259 -4.07229889]

```

Probamos ahora la capacidad predictiva del modelo con una muestra de prueba:

```

prueba = np.array([1, 1])
predicción = calcular_predicción(prueba, pesos)
print("Predicción: %.2f" % predicción) # 0.01

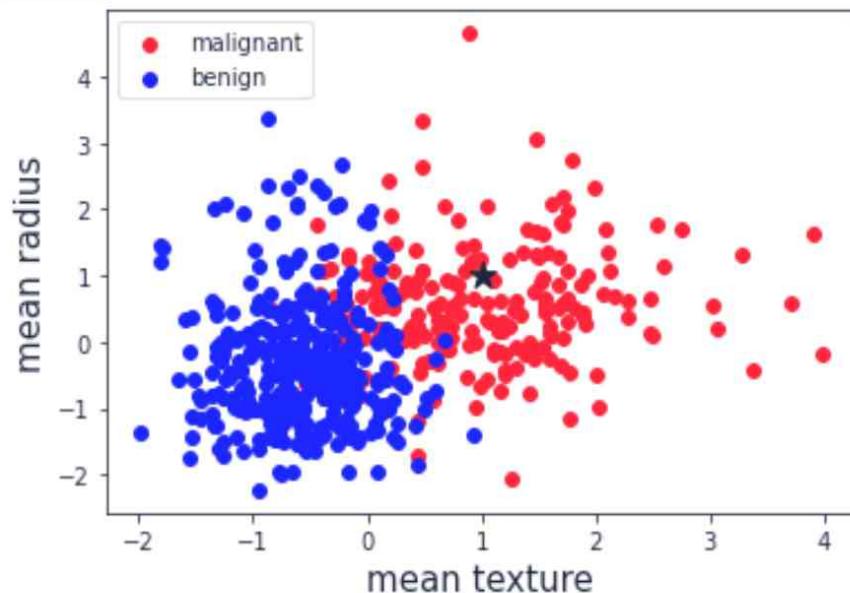
```

Predicción: 0.01

Del ejemplo anterior podemos notar como disminuye el costo a medida que aumenta el número de épocas, además la muestra [1, 1] se ha clasificado como 0 (tumor maligno). Además, si dibujamos una gráfica de dispersión observaremos que los puntos ubicados por encima del límite de decisión pertenecerán a la clase 0 (maligno) y lo que se encuentra por debajo a la clase 1 (benigno).

Luego con el siguiente código dibujamos los datos de entrenamiento en un gráfico de dispersión, así como nuestra muestra de prueba.

```
for target,color in zip(range(2),['r','b']):  
    plt.scatter(X[[y==target, MALIGNO]], X[[y==target,BENIGNO]],  
                color=color, label=dataset.target_names[target])  
plt.xlabel(dataset.feature_names[MALIGNO], fontsize=15)  
plt.ylabel(dataset.feature_names[BENIGNO], fontsize=15)  
plt.legend(loc = 'upper left')  
if predicción >= 0.5:  
    c='green'  
else:  
    c='black'  
plt.scatter(prueba[0], prueba[1], marker='*', color=c, s=150)  
plt.show()
```



7.3.1 Regresión logística con scikit-learn

Scikit-Learn dispone de una clase llamada LogisticRegression con una implementación optimizada del algoritmo de Regresión Logística. Miremos como podemos emplear esta clase en el conjunto de datos Breast Cáncer Wisconsin. Veremos que con esta implementación conseguimos una exactitud de nuestro

modelo de 0,90 para el conjunto de entrenamiento y 0,85 para el conjunto de prueba.

```

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from matplotlib.colors import ListedColormap
X_ent, X_pru, y_ent, y_pru = train_test_split(X, y, random_state = 1)

rloc = LogisticRegression()
rloc.fit(X_ent, y_ent)

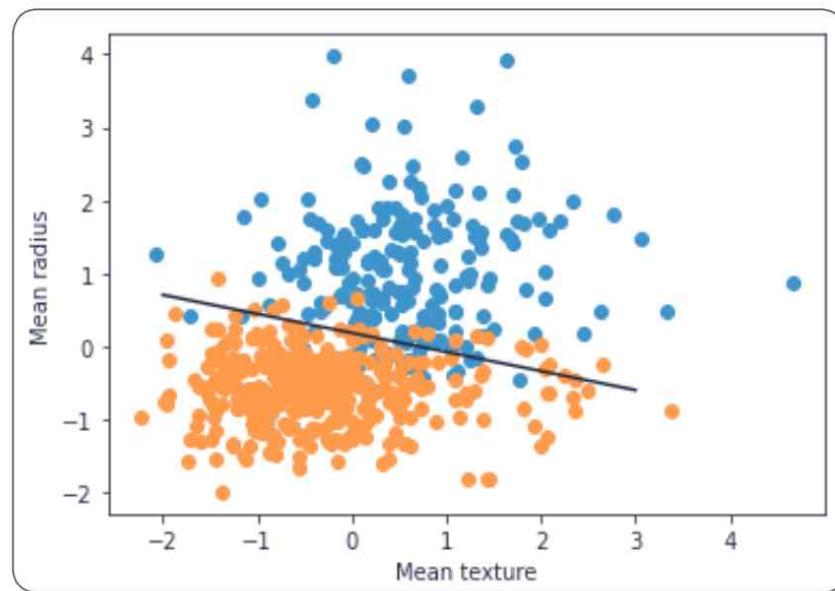
print('Exactitud con conjunto de entrenamiento {:.2f}'.format(rloc.score(X_ent,
y_ent)))
print('Exactitud con conjunto de pruebas: {:.2f}'.format(rloc.score(X_pru, y_
pru)))

colores = ['#FFFFAA', '#EFEFEF']
cmap_light = ListedColormap(colores[0:2])

# dibujamos el gráfico con el límite de decisión
min_x = int(min(X_ent[:,0]))
max_x = int(max(X_ent[:,0]))
xx = range(min_x,max_x)
yy = -(xx*rloc.coef_[0][0] + rloc.intercept_[0])/rloc.coef_[0][1]
plt.plot(xx,yy, color = "black")
for class_value in range(2):
    row_ix = np.where(y == class_value)
    plt.scatter(X[row_ix, 0], X[row_ix, 1])
plt.xlabel("Mean texture")
plt.ylabel("Mean radius")

```

Exactitud con conjunto de entrenamiento 0.90
 Exactitud con conjunto de pruebas: 0.85



Ahora bien, existe un método llamado `predict_proba()` que podemos llamar con nuestro objeto estimador `rloc` para obtener las probabilidades de que la muestra (prueba) pertenezca a la clase 0 (maligno) y pertenezca a la clase 1 (benigno). Vemos que hay un 98% de probabilidad de ser clase 0 y aproximadamente el 2% restante de ser clase 1. También este resultado lo corroboramos con la predicción cuyo valor ha sido 0, dando cuenta de la malignidad asociada a un tumor con una textura media y un radio medio igual a 1.

```
prueba = np.array([[1 , 1]])
prediccion = rloc.predict(prueba)
print("Predicción: %.2f" %prediccion)
print(rloc.predict_proba(prueba))
```

Predicción: 0.00
[[0.98094873 0.01905127]]

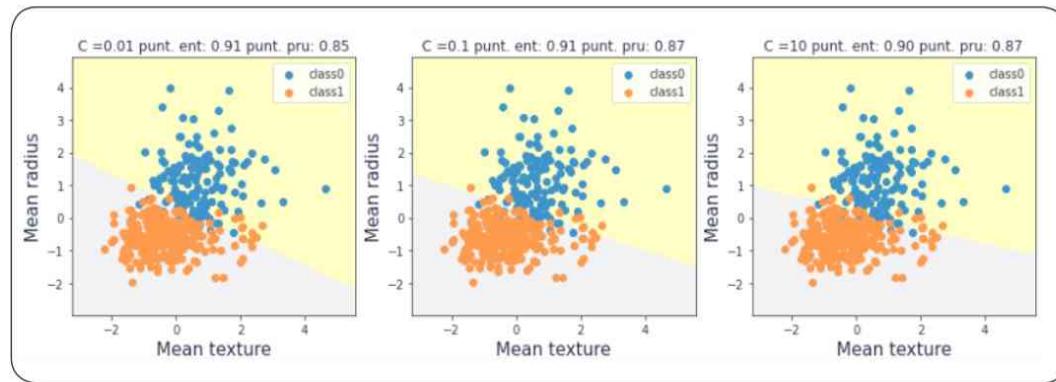
En tanto que el modelo aparentemente sufre algo de sobreajuste la clase `LogisticRegression` maneja un parámetro `C` con el que podemos controlar la regularización (por defecto L2), donde valores pequeños de este hiperparámetro producen un alto nivel de regularización. En el siguiente ejemplo vamos a probar el entrenamiento de algunos modelos con 3 valores diferentes para `C` (0.01, 0.1 y 10); y una solución (`solver = liblinear`) con la que se configura una regresión logística lineal. Sin embargo, tenga presente que existen otros valores para este último parámetro como '`newton-cg`', '`lbfgs`' (valor por defecto), '`sag`' y '`saga`', cuyo estudio se escapa del alcance de este libro, pero se encuentran disponibles para consulta en la documentación oficial de Scikit-learn.

Primero definimos el método `limite_decision()`, en este caso la línea separadora será un contorno relleno creado con la función `contourf()`.

```
def limite_decision(clf, X, subplot):
    min1, max1 = X[:, 0].min()-1, X[:, 0].max()+1
    min2, max2 = X[:, 1].min()-1, X[:, 1].max()+1
    x1grid = np.arange(min1, max1, 0.1)
    x2grid = np.arange(min2, max2, 0.1)
    xx, yy = np.meshgrid(x1grid, x2grid)
    r1, r2 = xx.flatten(), yy.flatten()
    r1, r2 = r1.reshape((len(r1), 1)), r2.reshape((len(r2), 1))
    grid = np.hstack((r1,r2))
    yhat = clf.predict(grid)
    zz = yhat.reshape(xx.shape)
    subplot.contourf(xx, yy, zz, cmap=cmap_light )
```

Ahora creamos varios modelos de regresión logística con valores para C iguales a 0.01, 0.1, y 10.

```
fig, subaxes = plt.subplots(1, 3, figsize=(15, 4))
etiquetas = 2
for c, subplot in zip([0.01, 0.1, 10], subaxes):
    clf = LogisticRegression(C=c, solver = 'liblinear').fit(X_ent, y_ent)
    p_ent = clf.score(X_ent, y_ent)
    p_pru = clf.score(X_pru, y_pru)
    limite_decision(clf, X_ent, subplot)
    for y in range(etiquetas):
        title ='Regresión logistica (mean texture vs mean radius), C = {:.3f}'.format(c)
        subplot.scatter(X_ent[y_ent==y,0],X_ent[y_ent==y,1],label='class'+format(y))
        subplot.set_xlabel('Mean texture', fontsize=15);
        subplot.set_ylabel('Mean radius', fontsize=15)
        subplot.set_title('C =' +str(c) + ' punt. ent: {:.2f}'.format(p_ent) + ' punt. pru: {:.2f}'.format(p_pru))
        subplot.legend()
plt.show()
```



El resultado anterior no nos deja ver el verdadero poder de la regularización, tal vez porque no tenemos un espacio de características dimensionalmente alto, sin embargo, si nos hace posible deducir a simple vista que con $C=10$ conseguimos un mejor equilibrio entre la exactitud del conjunto de entrenamiento y el de prueba. Ahora bien, tal y como hicimos en el capítulo pasado podemos usar la técnica de búsqueda de cuadrículas (GridSearchCV) para encontrar la mejor combinación de hiperparámetros para nuestro modelo:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import GridSearchCV
parametros = [
{
    'C': [ 0.5, 1.0, 1.5, 1.8, 2.0, 2.5],
    'solver': ['newton-cg','lbfgs','sag','saga']
}
]
gs = GridSearchCV(estimator=LogisticRegression(multi_class='auto'),
                  param_grid=parametros,scoring='accuracy', cv=10, n_jobs=-1)
gs.fit(X_ent, y_ent)
```

Una vez ajustado el objeto `gs` con los datos de entrenamiento podemos conocer cuáles son los mejores hiperparámetros por medio de la sentencia `gs.best_estimator_`, mostrando básicamente que un `solver=newton-cg` y un $C=1.0$ le viene bien a nuestro modelo otorgadole un buen desempeño. Entonces, si hacemos una validación cruzada de 10 folds con estos argumentos alcanzamos en promedio un 90% de exactitud, lo cual, aunque es un resultado que no difiere mucho del anterior permite comprender como GridSearchCV es también aplicable a problemas de clasificación, así mismo el ejemplo podrá inducir al lector a hacer pruebas con diferentes valores para los hiperparámetros hasta encontrar los más adecuados en el ámbito de un modelo en particular.

```
from sklearn.model_selection import cross_val_score
print("Exactitud CV: {:.2f}".format(cross_val_score(gs.best_estimator_, X_ent,
y_ent, scoring='accuracy', cv=10).mean()))
clf = LogisticRegression(C=1.0, solver='newton-cg').fit(X_ent, y_ent)
punt_pru = clf.score(X_pru, y_pru)
print("Exactitud {:.2f}".format(punt_pru))
```

7.3.2 Regresión logística con el descenso del gradiente estocástico

La técnica del descenso del gradiente visto hasta ahora realiza una actualización de los pesos tomando como base todas las muestras de entrenamiento por cada iteración del algoritmo, por ello también recibe el nombre de descenso del gradiente en lote. Este hecho puede resultar muy costoso computacionalmente y demorar mucho tiempo cuando tenemos un conjunto de datos muy grande, además algo que puede resultar peor, es que puede producir que el gradiente se quede atascado en un mínimo local en vez de un mínimo global, situación que puede provocar que nuestra función de costo no se optimice y por tanto no se logren producir resultados correctos.

Para mitigar un poco esta circunstancia existe el descenso del gradiente estocástico. En el cual se realiza la actualización de los pesos usando una muestra de entrenamiento por cada iteración. El término estocástico (sinónimo de aleatorio) significa que cada instancia es extraída al azar del conjunto.

A manera de pseudocódigo el SDG por sus siglas en inglés se puede expresar de la siguiente manera:

$$\begin{aligned} \text{for } j \text{ in range(len}(X)\text{):} \\ w = w + \eta (y^{(j)} - \hat{y}^{(j)}) \cdot x^j \end{aligned}$$

Para poder implementar el descenso del gradiente estocástico en el mismo problema de Regresión Logística anterior solo debemos cambiar el método `entrenar()`:

```
# Este método implementa el algoritmo del descenso del gradiente estocástico
def entrenar(X, y, pesos, max_iter, tasa_aprendizaje):
    for i in range(max_iter):
        error = 0.0
        for xi, yi in zip(X, y):
```

```

prediccion = calcular_prediccion(xi, pesos)
error = yi - prediccion
pesos[1:] += tasa_aprendizaje * np.dot(xi.T, error)
pesos[0] += tasa_aprendizaje * error.sum()
    # imprimimos el costo cada 10 iteraciones para apreciar como va
disminuyendo
if i % 10 == 0:
    print("Costo: %.2f" % calcular_costo(X, y, pesos))
return pesos

```

Al llamar a `entrenar()` con los mismos parámetros del ejercicio anterior notaremos que conseguimos resultados muy similares. Note el lector que hemos agregado código adicional para calcular el tiempo que dura el entrenamiento, el cual es de 0,6 segundos.

```

import timeit
timepo_inicial = timeit.default_timer()
pesos = entrenar(X_pru, y_pru, pesos, 200, 0.01)
print("Segundos: %0.3fs" % (timeit.default_timer() - timepo_inicial))

```

El lector si desea puede calcular el tiempo estimado del entrenamiento en el ejercicio anterior y se dará cuenta que es inferior al que acabamos de mostrar, sin embargo, observará que este último necesita menos iteraciones para poder converger, lo cual consiste en una de sus principales fortalezas.

Si tratamos de calcular la predicción de una muestra de prueba [0.5, 0.5], obtendremos como resultado 0.2, lo que sugiere que pertenece a la clase 0 (tumor maligno).

```

prueba = np.array([0.5, 0.5])
prediccion = calcular_prediccion(prueba, pesos)
print("Predicción: %.2f" % prediccion)

```

Para cerrar este tema miremos cómo podemos confirmar nuestros resultados mediante un gráfico de dispersión separándolos mediante un límite de decisión. En él se evidenciará cómo el punto de prueba con forma de estrella se ubica por encima del hiperplano separador.

<https://dogramcode.com/bloglibros>

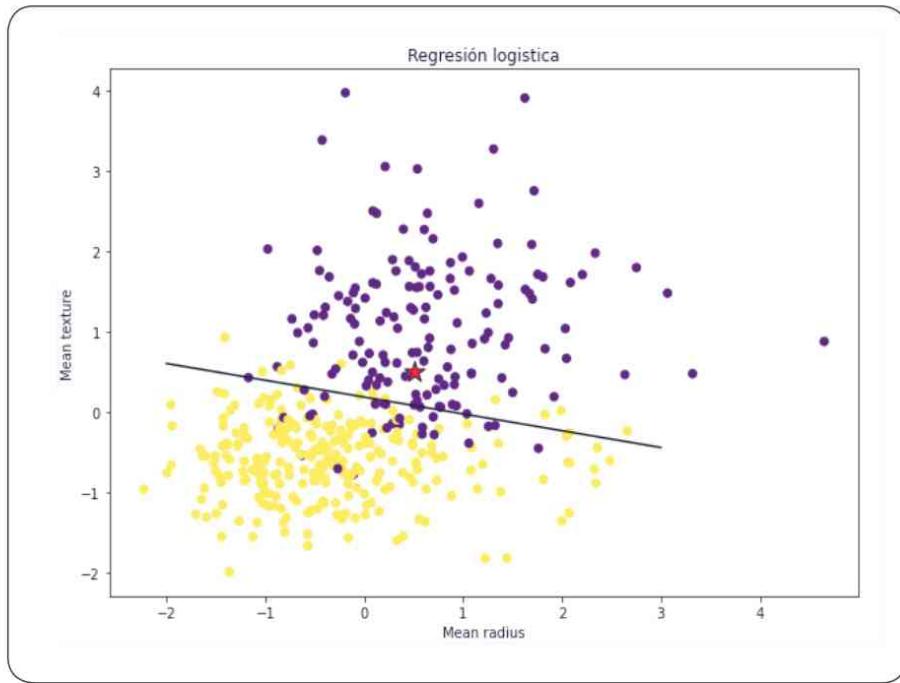


```

min_x = int(min(X[:,0]))
max_x = int(max(X[:,0]))
x = range(min_x,max_x)
y = -(x*pesos[1] + pesos[0])/pesos[2]

plt.figure(figsize =(10, 7))
plt.plot(x,y, color ="black")
plt.scatter(X_ent[:, 0], X_ent[:, 1], c = y_ent.ravel(), alpha = 1)
plt.scatter(prueba[0], prueba[1], marker='*', color='red', edgecolor='black',
s=250)
plt.xlabel('Mean radius')
plt.ylabel('Mean texture')
plt.title('Regresión logistica')

```



En Scikit-Learn existe una versión optimizada de SGD mediante la clase SGDClassifier, cuyo parámetro loss='log' indica que nuestro modelo usará la función logística como función de costo. Corroboremos nuestros resultados anteriores mediante el siguiente fragmento de código donde notaremos que la muestra se clasifica como clase 0 con una probabilidad de 0.99.

```
from sklearn.linear_model import SGDClassifier
prueba = np.array([[0.5 , 0.5]])
clf = SGDClassifier(loss='log')
clf.fit(X_ent, y_ent)
prediccion = clf.predict(prueba)
print("Predicción: %.2f" %prediccion)
print("Probabilidades:", clf.predict_proba(prueba))
```

Predicción: 0.00
 Probabilidades: [[0.99393007 0.00606993]]

7.3.3 Regresión logística con regularización

La regularización tal y como se mencionó en el capítulo 6 consiste en establecer una penalización sobre los parámetros de un modelo, más específicamente una contracción en los pesos a fin de evadir el sobreajuste, el cual como recordará el lector, hace que un modelo aprenda bien del conjunto de datos de entrenamiento, pero presente dificultades al momento de predecir con datos no vistos, debido a una pobre generalización. Cómo ya se explicó existen tres tipos de métodos de regularización: Lasso (L1), Ridge (L2) o ElasticNet. En cualquiera de los casos la regularización agrega un término adicional a la función de costo.

$$J(w) = \frac{1}{m} \sum_{i=1}^m - \left[y^{(i)} \log(\hat{y}(x^{(i)})) + (1 - y^{(i)}) \log(1 - \hat{y}(x^{(i)})) \right] + \alpha \|w\|^q$$

Donde α es la constante que multiplica el término de regularización, y q puede ser 1 o 2, para indicar Regularización L1 o L2 respectivamente.

$$\|w\|^1 = \sum_{j=1}^m |w_j|$$

$$\|w\|^2 = \sum_{j=1}^m w_j^2$$

En regularización la escogencia del parámetro α es crucial, ya que si se escoge un valor muy pequeño el modelo puede sufrir de alta varianza o sobreajuste, mientras que si es muy grande el modelo será muy simple y no será capaz de ajustar correctamente los datos de entrenamiento.

La regularización como ya se mencionó también puede usarse efectivamente para el proceso de selección de características, este como su nombre lo indica consiste en escoger un puñado de características del conjunto total, dado que en la práctica existen algunas que son irrelevantes o simplemente resultan redundantes dentro del ámbito del problema a tratar. La selección de características surte efecto porque como algunos pesos se contraen con la regularización, al punto de acercarse o llegar a 0, esto hace que las características asociadas a dichos pesos se anulen y por ende sean descartadas.

En Scikit-Learn la regularización la podemos agregar a un clasificador como el SGD especificando el parámetro *penalty* el cual admite los valores: none (sin regularización), l1, l2, elasticnet y el parámetro *alpha* que representa la constante de regularización α .

Con el código siguiente podemos entrenar un modelo de regresión logística con el conjunto de datos que hemos venido usando tomando todas sus instancias, definiendo un objeto SGDClassifier con una función de perdida de tipo logística, una penalización l1 para comprimir los pesos y un valor para alpha de 0.01. Al final también mostraremos cuales son los atributos menos importantes.

```
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score, ShuffleSplit

X = dataset["data"]
y = dataset["target"]
nombres = dataset["feature_names"]

sgd = SGDClassifier(loss='log', penalty='l1', alpha=0.01, max_iter=1000,
tol=1e-3)
puntajes = []

for i in range(X.shape[1]):
    punt = cross_val_score(sgd, X[:, :i+1], y, scoring="r2",
                           cv=ShuffleSplit(len(X), 3, .3))
    puntajes.append((round(np.mean(punt), 3), nombres[i]))
caract = sorted(puntajes, reverse=True)
print(caract)
```

[(0.539, 'worst radius'), (0.41, 'area error'), (0.38, 'mean radius'), (0.278, 'worst concavity'), (0.231, 'radius error'), (0.183, 'perimeter error'), (0.113, 'worst compactness'), (-0.025, 'mean concavity'), (-0.096, 'worst concave points'),

(-0.311, 'mean texture'), (-0.356, 'worst texture'), (-0.36, 'mean perimeter'), (-0.421, 'compactness error'), (-0.429, 'mean symmetry'), (-0.434, 'worst smoothness'), (-0.449, 'symmetry error'), (-0.456, 'concavity error'), (-0.464, 'mean compactness'), (-0.466, 'worst symmetry'), (-0.466, 'smoothness error'), (-0.473, 'mean fractal dimension'), (-0.483, 'worst perimeter'), (-0.497, 'texture error'), (-0.506, 'mean concave points'), (-0.525, 'fractal dimension error'), (-0.527, 'mean smoothness'), (-0.54, 'worst fractal dimension'), (-0.542, 'concave points error'), (-0.719, 'mean area'), (-0.728, 'worst area')].

Según este resultado las 3 características menos importantes en nuestro modelo son: *concave points error*, *mean area* y *worst area*. Mientras que las 3 más importantes son: *worst radius*, *area error* y *mean radius*.

7.4. Métricas de evaluación

En este apartado estudiaremos algunas de las técnicas más usadas para evaluar modelos de clasificación, ya que como es de conocimiento del lector la evaluación es un paso fundamental dentro del aprendizaje automático porque permite saber qué tan bien o mal ha sido el trabajo de un modelo. Por tanto, exploraremos las métricas de evaluación de uso más común, tal y como hicimos en el capítulo dedicado a Regresión.

7.4.1. Matriz de confusión

Una matriz de confusión es una especie de cuadrícula usada para visualizar el desempeño de un modelo de aprendizaje supervisado, permite describir la distribución de los valores verdaderos y las predicciones, mostrando así la cantidad de errores y aciertos de nuestro algoritmo de clasificación. Concretamente, en un clasificador binario donde 0 corresponde a la clase negativa y 1 a la positiva, la matriz de confusión tendría una representación como la siguiente:

		Actual	
		0	1
Predicción	0	VN	FN
	1	FP	VP

Siendo,

- VP (Verdadero positivo): El modelo predijo correctamente la salida como positiva.
- FN (Falso negativo): El modelo predijo incorrectamente la salida como negativa.
- FP (Falso positivo): El modelo predijo incorrectamente la salida como positiva.
- VN (Verdadero negativo): El modelo predijo correctamente la salida como negativa

Ahora podemos crear la matriz de confusión de nuestro clasificador por medio de la función `confusion_matrix()` que recibe como parámetros las etiquetas actuales o verdaderas y las etiquetas predichas por el modelo:

```
from sklearn import metrics

y_pred = rloc.predict(X_pru)
print(metrics.confusion_matrix(y_true = y_pru, # etiquetas actuales
                                y_pred = y_pred)) #etiquetas predichas
```

[[44 11]
 [8 80]]

7.4.2. Exactitud (Accuracy)

La exactitud mide el porcentaje de casos que el modelo ha acertado. Consiste en la proporción entre las predicciones correctas y el número total de predicciones. Además, es la métrica por defecto de los algoritmos en Scikit-learn aplicable por medio de la función `score()`, disponible también en la función `accuracy_score()` del submódulo `metrics`.

```
from sklearn import metrics

print("Exactitud: {:.2f}".format( metrics.accuracy_score(y_true = y_pru,
                                                        y_pred = y_pred)))
```

$$\text{Accuracy} = \frac{VP + VN}{VP + VN + FP + FN}$$

Exactitud: 0.87

7.4.3. Precisión (Precision)

La precisión es la relación entre las predicciones clasificadas correctamente como positivas y el número total de predicciones correctas realizadas. (tanto positivas como negativas).

$$\text{Precision} = \frac{VP}{VP + VN}$$

```
from sklearn import metrics

print("Precisión: {:.2f}".format( metrics.precision_score(y_true = y_pru,
    y_pred = y_pred)))
```

Precisión: 0.88

7.4.4. Recall, Sensibilidad o TPR (Tasa de verdadero positivo)

Corresponde con el número de elementos identificados correctamente como positivos del total de positivos verdaderos.

$$\text{Recall} = \frac{VP}{VP + FN}$$

```
from sklearn import metrics

print("Recall: {:.2f}".format( metrics.recall_score(y_true = y_pru,
    y_pred = y_pred)))
```

Recall: 0.91

7.4.5. F1

Esta métrica permite combinar las medidas de precisión y sensibilidad en un solo valor. Lo cual puede ser útil en casos donde se desee comparar el rendimiento combinado de dichas métricas.

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

```
from sklearn import metrics

print("F1: {:.2f}".format(metrics.f1_score(y_true = y_pru,
y_pred = y_pred)))
```

F1: 0.89

7.4.6. Tasa de falsos positivos

Esta métrica corresponde a la proporción entre los puntos de datos negativos que fueron erróneamente considerados como positivos, con respecto a todos los puntos de datos negativos.

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

Entre más alto el FPR significa que más muestras negativas fueron clasificadas erróneamente como positivas.

Scikit-learn ofrece una función muy interesante llamada `classification_report()` que muestra un reporte de las métricas precisión, recall, y F1.

```
print(metrics.classification_report(y_true = y_pru,y_pred = y_pred))
```

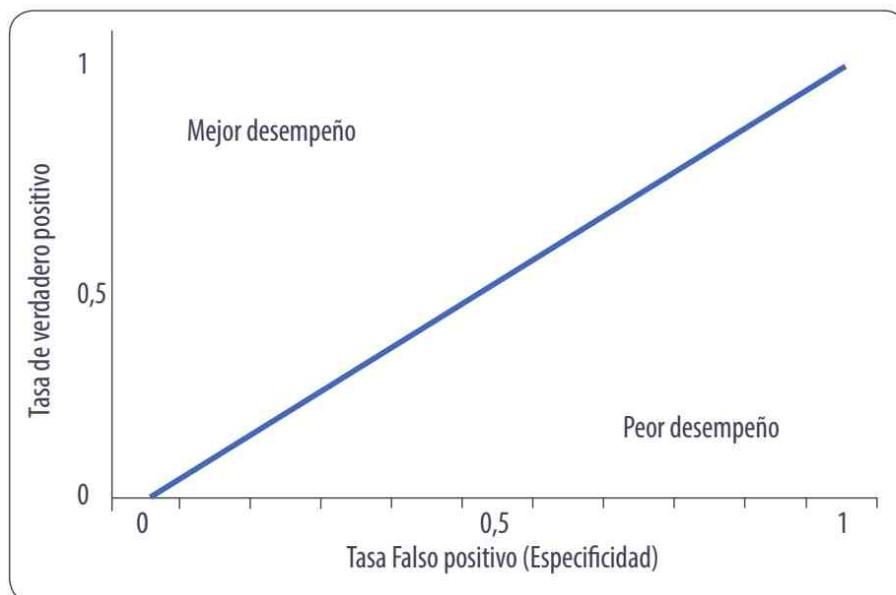
	precision	recall	f1-score	support
0	0.85	0.80	0.82	55
1	0.88	0.91	0.89	88
micro avg	0.87	0.87	0.87	143
macro avg	0.86	0.85	0.86	143
weighted avg	0.87	0.87	0.87	143

<https://dogramcode.com/bloglibros>



7.4.7. Curvas ROC (*receiver operating characteristics*)

Las curvas características operativas del receptor son una herramienta válida para comparar diferentes clasificadores a partir de los puntajes de sus predicciones. En general, este puntaje puede ser interpretado como una probabilidad entre 0 y 1. La estructura es la siguiente:



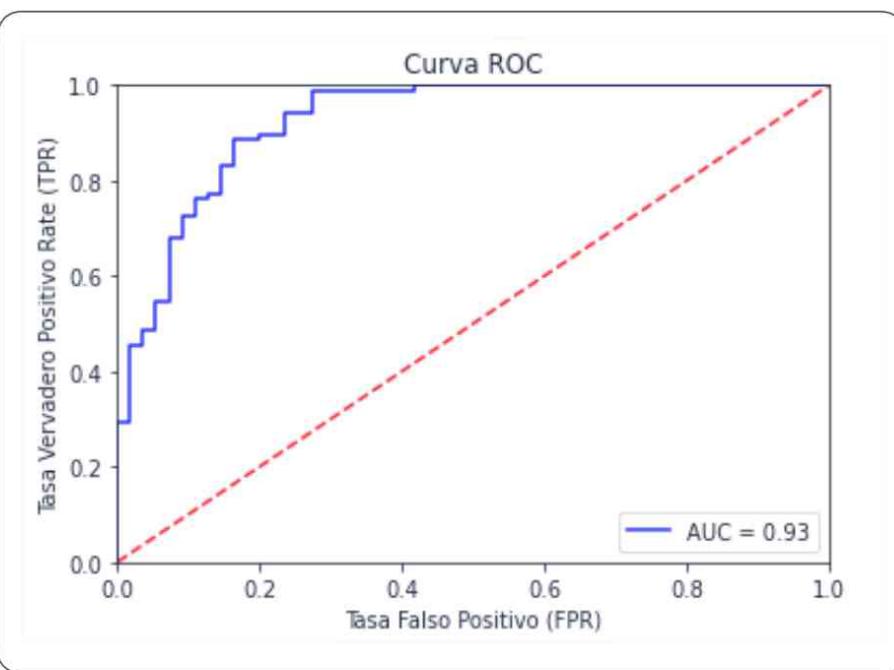
El eje x representa el incremento de la tasa de falsos positivos (conocida como especificidad), mientras el eje y representa la tasa de verdaderos positivos (conocida como sensibilidad). La línea representa un clasificador, entonces todas las curvas debajo de este umbral tienen un bajo desempeño, mientras que las que estén por encima tienen un mejor desempeño. El objetivo es crear modelos cuyo rendimiento se encuentre dentro de los segmentos [0,0] – [0,1] y [0,1] – [1,1].

A continuación, vamos a evaluar nuestro ejemplo de clasificación utilizando la curva ROC, para ello emplearemos la función `roc_curve()` que toma las etiquetas de prueba junto con los valores predichos, devolviendo la tasa de falsos positivos (FPR), la tasa de verdaderos positivos (TPR) y los umbrales o límites de adecuado desempeño.

```
from sklearn.metrics import roc_curve, auc
#hallamos la predicción de probabilidad usando el conjunto de prueba
probs = clf.predict_proba(X_pru)
preds = probs[:,1]
#encontramos el FPR, TPR y los umbrales
fpr, tpr, threshold = roc_curve(y_pru, preds)
```

Luego, con la función `auc()` podemos encontrar el área bajo la curva, la cual tomamos como insumo para dibujar la curva ROC utilizando Matplotlib.

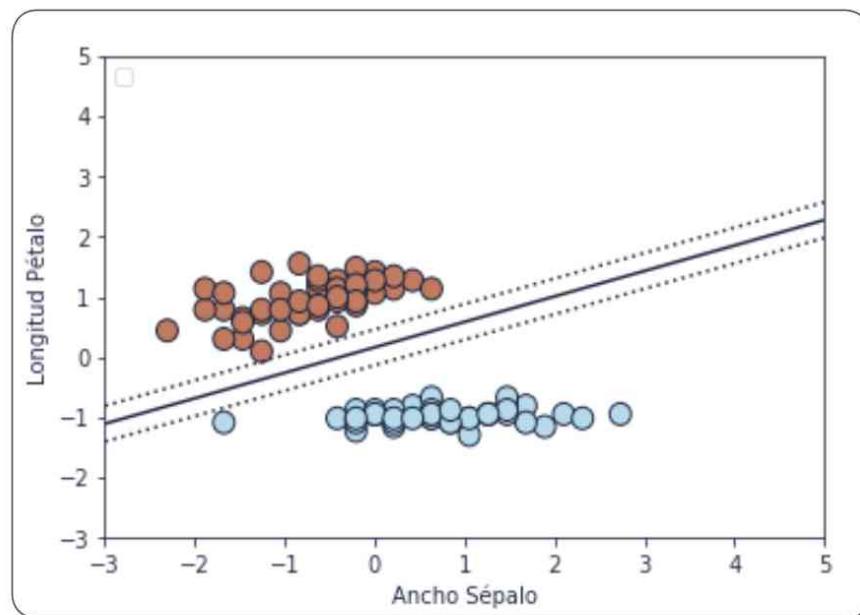
```
# encontramos el área bajo la curva
roc_auc = auc(fpr, tpr)
import matplotlib.pyplot as plt
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc)
plt.plot([0, 1], [0, 1], 'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('Tasa Verdadero Positivo Rate (TPR)')
plt.xlabel('Tasa Falso Positivo (FPR)')
plt.title('Curva ROC')
plt.legend(loc = 'lower right')
plt.show()
```



7.5 Máquinas de vectores de soporte (SVM)

Las máquinas de vectores de soporte o SVM por sus siglas en inglés, son un método muy potente usado para resolver problemas de clasificación, es altamente eficiente en conjuntos de datos de múltiples dimensiones. Consiste básicamente

en definir un hiperplano o límite de decisión que separe las muestras en dos grupos, donde son clasificadas como positivas las que se ubiquen por encima del límite de decisión y como negativas las que se encuentren por debajo de él. El objetivo fundamental es maximizar el margen M o distancia existente entre las muestras vecinas llamadas vectores de soporte y el hiperplano separador. En otras palabras, lo que se busca es que este último se encuentre lo más lejos posible de ambas clases. En la siguiente grafica podemos ver un límite decisión con margen M que separa muestras del conjunto de datos Iris tomando las variables ancho del sépalo (x_1) y longitud del pétalo (x_2).



De lo anterior se deduce que, algún punto x se ubicará por encima del hiperplano, si $wx_2 + b > 0$. Así mismo, se ubicará por debajo si $wx_1 + b < 0$. Y en caso de que $wx_1 + b = 0$ se ubicará en el mismo límite de decisión, considerándose un vector de soporte.

No obstante, cuando se trabaja con SVM es usual considerar a las clases negativas como -1 y a las positivas como +1.

$$\begin{aligned} \text{si } wx_2 + b \geq 0 \text{ entonces } y^{(i)} &= +1 \\ \text{si } wx_1 + b < 0 \text{ entonces } y^{(i)} &= -1 \end{aligned}$$

Ahora si restamos estas dos expresiones tenemos:

$$\begin{aligned} (wx_2 + b) - (wx_1 + b) &= 1 - (-1) \\ w(x_2 - x_1) &= 2 \end{aligned}$$

$$\text{Margen} = x_2 - x_1 = \frac{2}{\|w\|}$$

Luego el objetivo es maximizar el margen ($\frac{2}{\|w\|}$) , en donde $\|w\|$ es la norma euclíadiana, la cual se puede calcular así: $\|w\| = \sqrt{w_0^2 + w_1^2 \dots w_n^2}$

Pero en vez de maximizar el recíproco de la norma, resulta más conveniente minimizar el cuadrado de la misma, multiplicándola por $\frac{1}{2}$ con el fin de lograr que sea derivable y permitir la aplicación de algoritmos como el descenso del gradiente. Por tanto, la meta cuando entrenamos un modelo de máquinas de vectores de soporte es minimizar la función $J = \frac{1}{2} \|w\|^2$, sujetos a la restricción $y_i(wx_i + b) \geq 1$, con la cual se consigue la máxima separación entre las clases, que una vez obtenida nos abre el camino para deducir los valores de w_0 , w_1 y b . El anterior es un problema de optimización con restricciones que se puede resolver usando la técnica de multiplicadores de Lagrange cuya explicación se sale del alcance de este libro.

Minimizar esta función de coste bajo esta restricción se conoce como (SVM) de margen duro y no admite muestras que no estén clasificadas, además resulta computacionalmente muy costoso si se realiza aplicando un algoritmo iterativo como el descenso del gradiente, por lo tanto, optaremos por el uso de la librería Scikit-learn que provee un algoritmo que cumple este propósito y es muy sencillo de implementar.

Es importante que el lector tenga en cuenta que lo anterior funciona para conjuntos de datos linealmente separables, lo cual no es siempre el caso. Para problemas no lineales donde no se puede garantizar para todos los puntos de datos la restricción anterior, la solución es agregar variables flexibles $\xi > 0$ a cada pareja etiqueta-atributos descriptivos (x_i, y_i) . La variable ξ mide la distancia de un punto mal clasificado (x_i) al margen del hiperplano positivo cuando dicho punto está sobre el lado negativo. Para puntos bien clasificados $\xi = 0$. Luego la función objetivo ahora le adicionamos un nuevo término quedando de la siguiente manera:

$$J = \frac{1}{2} \|w\|^2 + C \sum_{i=1}^N \xi_i \text{ sujeto a la restricción } y_i(wx_i + b) \geq 1 - \xi_i \text{ con } \xi_i \geq 0$$

Esta forma se conoce como SVM de margen blando. En esta ecuación, C establece un balance entre el tamaño del margen y las clasificaciones erróneas y se conoce como variable de regularización. Un pequeño valor para C (baja penalización) significa emplear un margen grande permitiendo aceptar más errores de clasificación, mientras que un C grande (alta penalización) significa aplicar un margen más pequeño y por ende el SVM intenta minimizar el número de ejemplos mal clasificados.

Miremos como mediante Scikit-learn podemos implementar un clasificador binario SVM lineal para nuestro conjunto de datos Iris que clasifique las especies Setosa (clase 0) y Versicolor (clase 1), por tal motivo solo se toman las 100 primeras muestras del conjunto de datos original.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

iris = datasets.load_iris()
X = iris.data
y = iris.target
X = X[0:100, [1,2]]
y = y[0:100]

X_ent, X_pru, y_ent, y_pru = train_test_split(X, y, test_size=0.2, random_
state=1)
svc = SVC(kernel='linear', C=1.0)
svc.fit(X_ent, y_ent)
y_pred = svc.predict(X_pru)
print("Exactitud:" ,format(accuracy_score(y_pru, y_pred) * 100))
```

Exactitud: 100.0

En este ejemplo se ha entrenado un clasificador de máquinas de soporte vectorial lineal, con un valor para la constante de regularización C de 1, consiguiendo una exactitud máxima del 100%.

También podemos obtener algunas características importantes del modelo creado utilizando las siguientes sentencias:

```
print('Vector de pesos (w) =',svc.coef_[0])
print('b =',svc.intercept_[0])
print('Indices de vectores de soporte =', svc.support_)
print('Vectores de soporte =', svc.support_vectors_)
print('Número de vectores de soporte por cada clase =', svc.n_support_)
print('Coeficientes del vector de soporte en la función de decisión =',np.abs(svc.
dual_coef_))
```

Vector de pesos (w) = [-0.6687899 1.2261146]
b = -1.055731845509071
Indices de vectores de soporte = [25 32 29]
Vectores de soporte = [[2.3 1.3]
[3.4 1.9]
[2.5 3.]]
Número de vectores de soporte por cada clase = [2 1]
Coeficientes del vector de soporte en la función de decisión = [[0.210191 0.789809
1.]]

Con la información anterior y un poco de nuestros conocimientos matemáticos podemos dibujar el hiperplano y los márgenes que separan las 100 muestras de acuerdo a las clases Setosa y Versicolor.

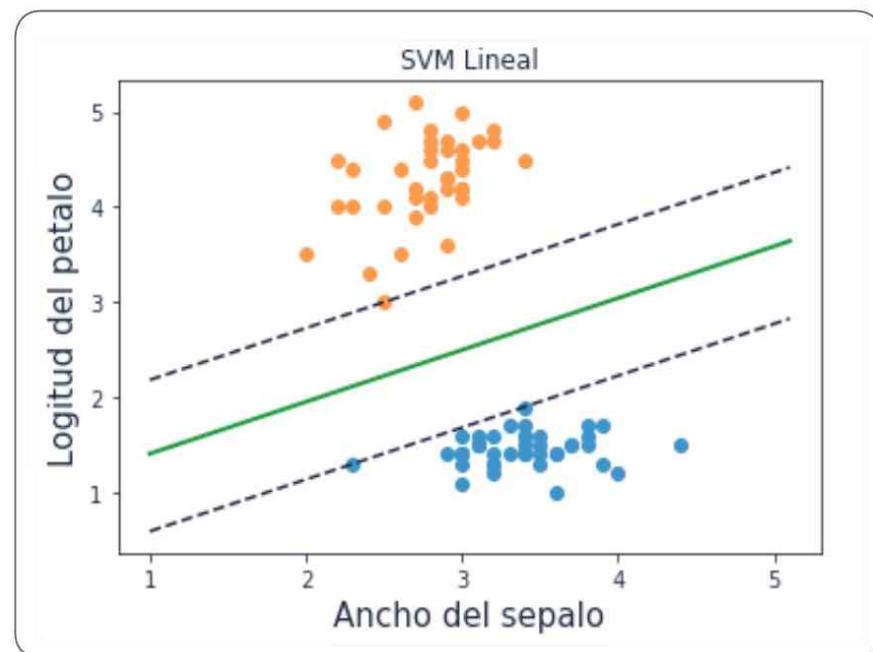
```
import pandas as pd
import numpy as np

df = pd.DataFrame(X_ent, columns = ['Ancho del sepalo','Logitud del petalo'])
df['clase'] = y_ent

w = svc.coef_[0]
# pendiente del hiperplano
pendiente = -w[0] / w[1]
b = svc.intercept_[0]
# coordenadas del hiperplano
xx = np.linspace(1, 5.1)
yy = pendiente * xx - (b / w[1])
# margenes
s = svc.support_vectors_[0] # primer vector de soporte
yy_abajo = pendiente * xx + (s[1] - pendiente * s[0])
s = svc.support_vectors_[2] # último vector de soporte
yy_arriba = pendiente * xx + (s[1] - pendiente * s[0])

for y in range(2):
    plt.scatter(X_ent[y_ent==y,0], X_ent[y_ent==y,1], label='clase'+format(y))

# dibujos del hiperplano
plt.plot(xx, yy, linewidth=2, color='green');
# dibujamos los márgenes
plt.plot(xx, yy_abajo, 'k--')
plt.plot(xx, yy_arriba, 'k--')
plt.xlabel('Ancho del sepalo', fontsize=15);
plt.ylabel('Logitud del petalo', fontsize=15)
plt.title("SVM Lineal")
```

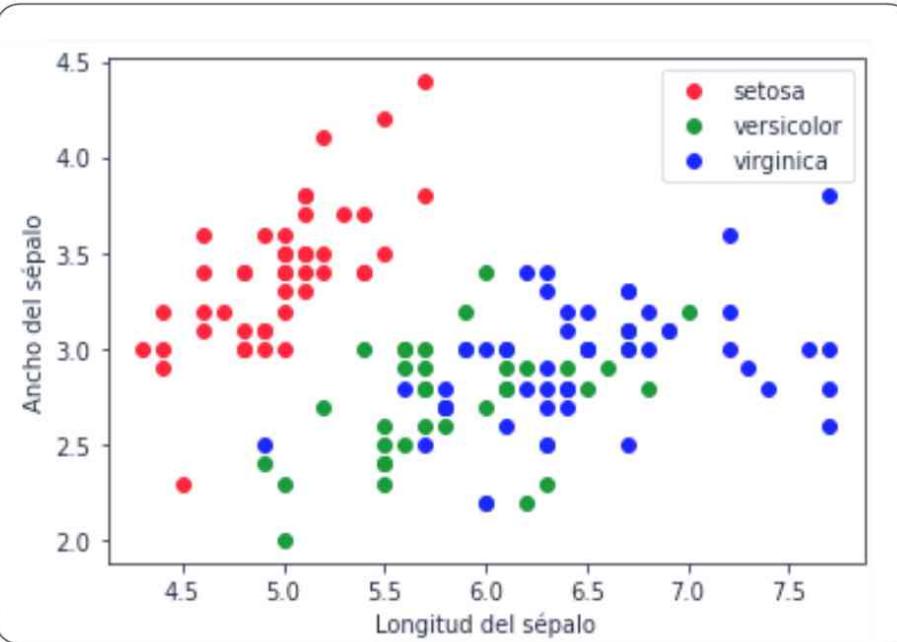


7.5.1. Clasificación Multiclas con SVM lineal

En este subapartado estudiaremos cómo podemos usar un modelo de máquinas de soporte vectorial con kernel lineal para clasificar los 3 tipos de flores del dataset Iris tomando las columnas longitud del sépalo y ancho del sépalo.

```
import numpy as np
from sklearn import datasets
iris = datasets.load_iris()

X = iris.data[:, [0,1]]
y = iris.target
X_ent, X_pru, y_ent, y_pru = train_test_split(X, y, test_size=0.1, random_
state=1)
colors = ['red', 'green', 'blue']
for color, i, target in zip(colors, [0, 1, 2], iris.target_names):
    plt.scatter(X_ent[y_ent==i, 0], X_ent[y_ent==i, 1], color=color, label=target)
plt.xlabel('Longitud del sépalo')
plt.ylabel('Ancho del sépalo')
plt.legend(loc='best')
plt.show()
```

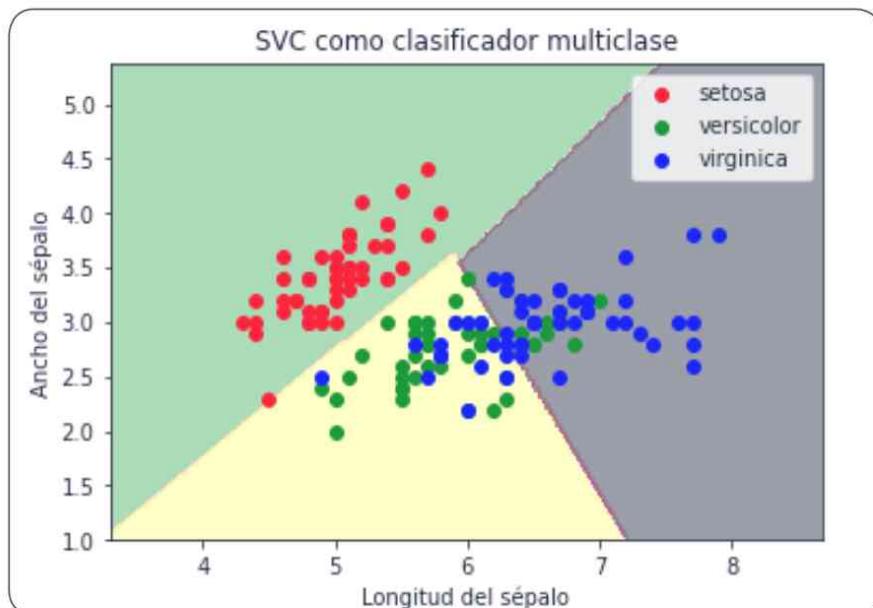


Acto seguido, entrenaremos un clasificador SVM y dibujamos las líneas de separación de cada clase del conjunto de datos.

```
C = 1
clf = SVC(kernel='linear', C=C).fit(X, y)
titulo = 'SVC como clasificador multiclase'
x_min, x_max = X_ent[:, 0].min() - 1, X_ent[:, 0].max() + 1
y_min, y_max = X_ent[:, 1].min() - 1, X_ent[:, 1].max() + 1
h = (x_max / x_min)/100
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.Accent, alpha=0.8)

colors = ['red', 'green', 'blue']
for color, i, target in zip(colors, [0, 1, 2], iris.target_names):
    plt.scatter(X[y==i, 0], X[y==i, 1], color=color, label=target)

plt.xlabel('Longitud del sépalo')
plt.ylabel('Ancho del sépalo')
plt.title(titulo)
plt.legend(loc='best', shadow=False, scatterpoints=1)
```



Nuestros resultados son bastante prometedores y para revalidarlos haremos una predicción con una muestra de prueba.

```
prueba = [[5 ,4]]
pred = clf.predict(prueba)
print("Predicción:", pred[0]) # 0 = setosa

exact = clf.score(X_pru, y_pru)
print("Exactitud:",exact)
```

Predicción: 0

Exactitud: 0.8

También, podemos probar cambiando el valor del hiperparámetro C para ver cómo se comporta el margen, observaremos como este último disminuye a medida que vamos aumentando C, enfocándose más en la clasificación correcta de los puntos.

```
fig, subjes = plt.subplots(1, 3, figsize=(18, 4))
for c, subje in zip([0.1, 1, 100], subjes):
    clf = SVC(kernel='linear', C = c, random_state = 67).fit(X_ent, y_ent)
    titulo = 'SVC Lineal , C = {:.2f}'.format(c)
    colors = ['red','green','blue']
    for color, i, target in zip(colors, [0, 1, 2], iris.target_names):
```

```

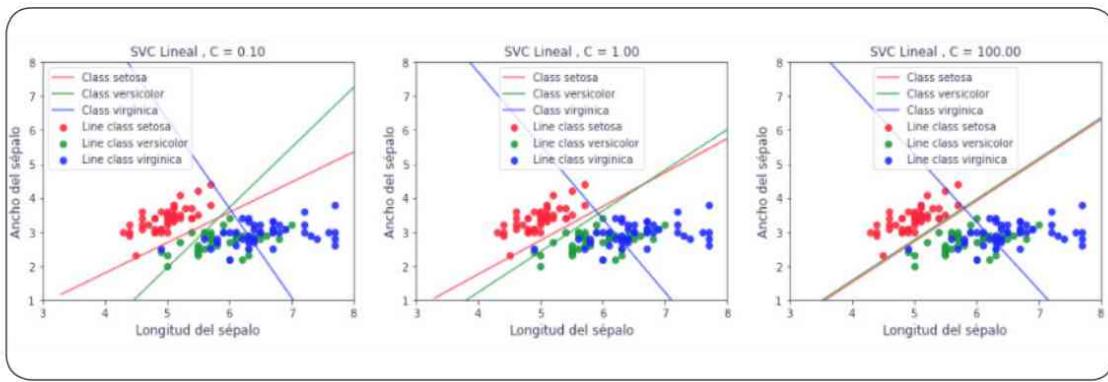
subeje.scatter(X_ent[y_ent==i, 0], X_ent[y_ent==i, 1], color=color, label=target,
alpha=.9)

subeje.set_xlabel('Longitud del sépalo', fontsize=12);
subeje.set_ylabel('Ancho del sépalo', fontsize=12)
subeje.set_title(titulo)

for w, b, color in zip(clf.coef_, clf.intercept_, ['red','green','blue']):
    x_min, x_max = X_ent[:, 0].min() - 1, X_ent[:, 0].max() + 1
    h = (x_max / x_min)/100
    xx = np.arange(x_min, x_max, h)
    pendiente = -w[0] / w[1]
    yy = pendiente * xx - (b / w[1])
    subeje.plot(xx, yy , c=color, alpha=0.8)

subeje.legend(['Class setosa', 'Class versicolor', 'Class virginica', 'Line class
setosa', 'Line class versicolor","Line class virginica"],loc='best')
subeje.set_xlim(3, 8)
subeje.set_ylim(1, 8)

```

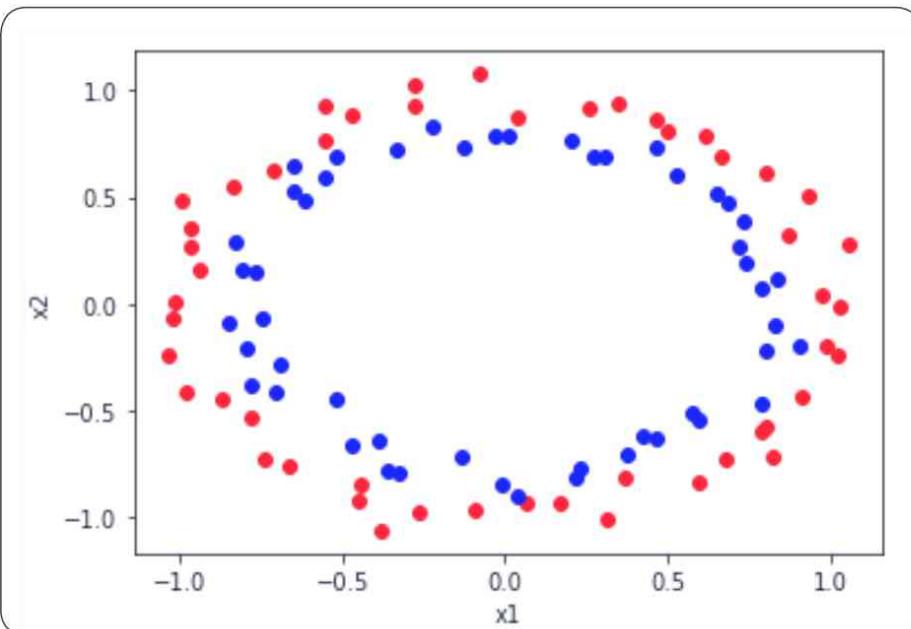


7.5.2. Kernels para separar datos no lineales

Algunas veces los puntos de datos no son linealmente separables como los que aparecen en la figura de abajo. En ella aparecen 50 puntos generados con la función `make_circles()`, la cual se encuentra dentro del submódulo `datasets`.

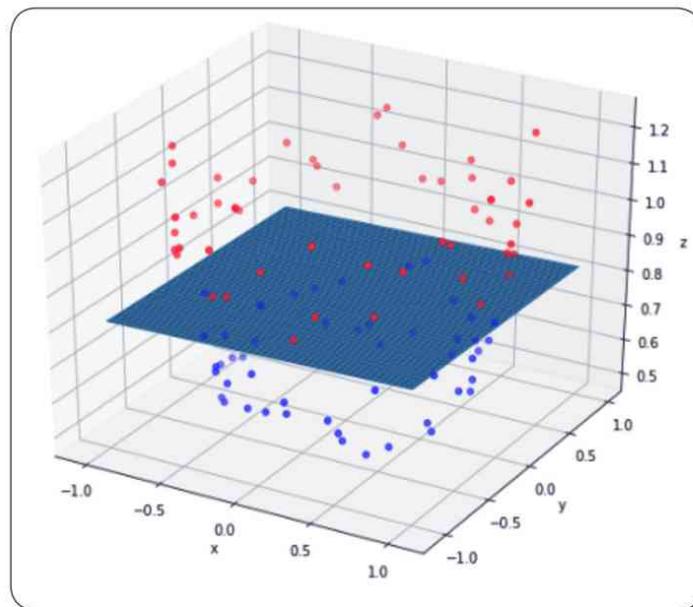
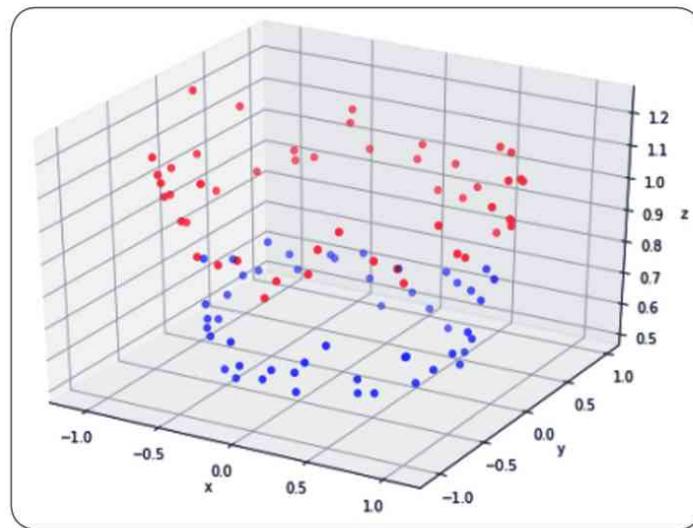
```
from sklearn.datasets import make_circles

X, y = make_circles(n_samples=100, noise=0.05)
rgb = np.array(['r', 'b'])
for i in range(2):
    plt.scatter(X[y==i,0], X[y==i,1], label='clase'+format(i), color=rgb[i])
plt.xlabel("x1")
plt.ylabel("x2")
```



La solución sin embargo es agregar una nueva característica y mapear o transformar el espacio de dos dimensiones a uno de tres dimensiones, en el nuevo espacio proyectado, nosotros podemos encontrar una superficie de separación de los datos. Ahora nuestros datos serán linealmente separables en dicho espacio dimensionalmente mayor (x_1, x_2, z).

Tal transformación se logra mediante una función φ , de manera que el conjunto de datos transformado $\varphi(x^{(i)})$ es linealmente separable por una hiperplano lineal. Las siguientes gráficas ilustran este proceso:



Como vemos nuestro vector de características es ahora filtrado por una función no lineal que hace más compleja la restricción y la hace más costosa computacionalmente, ya que por cada par de vectores toca multiplicar $\varphi(x_i)\varphi(x_j)$, algo que resulta insostenible en grandes conjuntos de datos. Surge entonces el truco del kernel para mitigar esta situación.

La función Kernel presenta la siguiente propiedad:

$$K(x_i, x_j) = \varphi(x_i)\varphi(x_j)$$

Existen varios tipos de kernel que se pueden usar en Scikit-learn:

- Kernel Lineal: Es el que se acaba de explicar recientemente:

$$K(x_i, x_j) = \varphi(x_i) \varphi(x_j)$$

- Kernel con base radial o gausiano: Es la función kernel por defecto y tiene la siguiente definición:

$$K(x_i, x_j) = e^{-\gamma \|x_i - x_j\|^2}$$

- Kernel polinómico:

$$K(x_i, x_j) = (a * x_i * x_j + b)^p, \text{ donde } p \text{ es el grado del polinomio.}$$

- Kernel sigmoide:

$$K(x_i, x_j) = \tanh z = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Básicamente la función K determina qué tipo de hiperplano utilizar para separar las clases. La idea es entonces, probar con varios tipos de kernel e ir variando el valor de la variable C a fin de encontrar la mejor separación de los datos y por supuesto una mejor exactitud en la clasificación. Tenga en cuenta que un C muy grande puede implicar una mayor complejidad y alto sobreajuste, mientras que uno C pequeño una menor complejidad evidenciada en un alto subajuste.

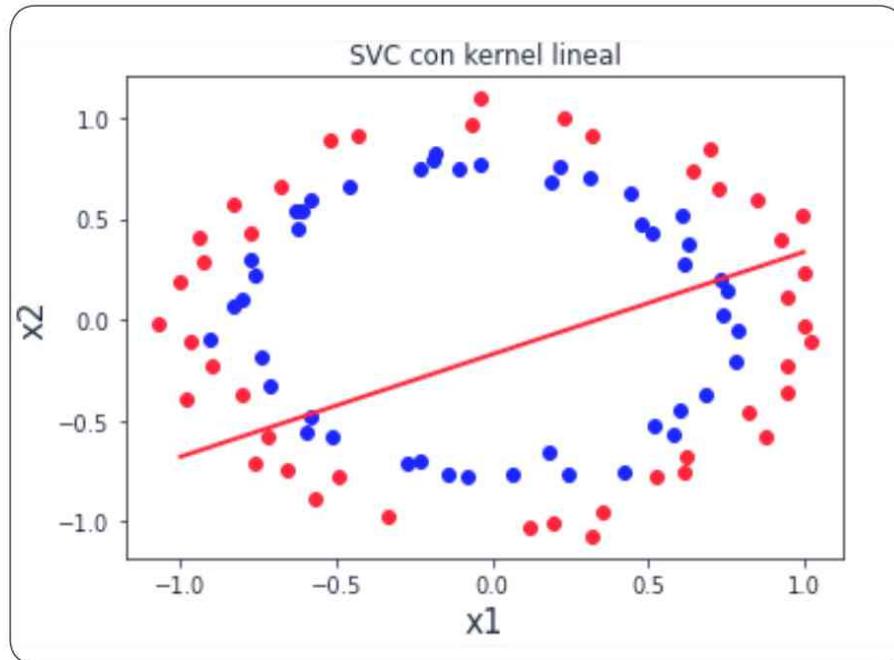
Considere los datos de la gráfica de arriba si intentamos utilizar un kernel lineal obtendríamos una exactitud de 40% en la clasificación, lo cual nos demuestra que este kernel no es el adecuado para este caso, podemos comprobar lo dicho si ejecutamos el siguiente código:

```
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

X_ent, X_pru, y_ent, y_pru = train_test_split(X, y, test_size=0.1, random_state=1)
svc = SVC(kernel='linear', C=1, random_state=1)
svc.fit(X_ent, y_ent)
y_pred = svc.predict(X_pru)
print("Exactitud: ", format(accuracy_score(y_pru, y_pred) * 100))
```

Exactitud: 40.0

La gráfica con el límite de decisión conseguido también da cuenta de una clasificación correcta de aproximadamente un 40% de las muestras de cada clase debido a que no se han logrado separar convenientemente.



Ahora bien, si intentamos probar con la función kernel con base radial (rbf) y un valor de $C=2$, veremos que se obtendrá una mejor separación y por supuesto mejor exactitud del modelo.

```
svc = SVC(kernel='rbf', C=2, random_state=1, degree=3, gamma='auto')
svc.fit(X_ent, y_ent)

y_pred = svc.predict(X_pru)
print("Exactitud:" , format(accuracy_score(y_pru, y_pred) * 100))
```

Exactitud: 80.0

Además del hiperparámetro C existe otro llamado γ que define la influencia de los puntos de entrenamiento. Un bajo valor indica que los puntos tienen la misma influencia provocando un modelo más generalizado, mientras que uno alto significa que los puntos más cercanos a dicho límite son los que tienen mayor influencia, lo cual repercute en un mayor sobreajuste.

El siguiente fragmento de código muestra variaciones en la combinación de estos dos hiperparámetros.

```

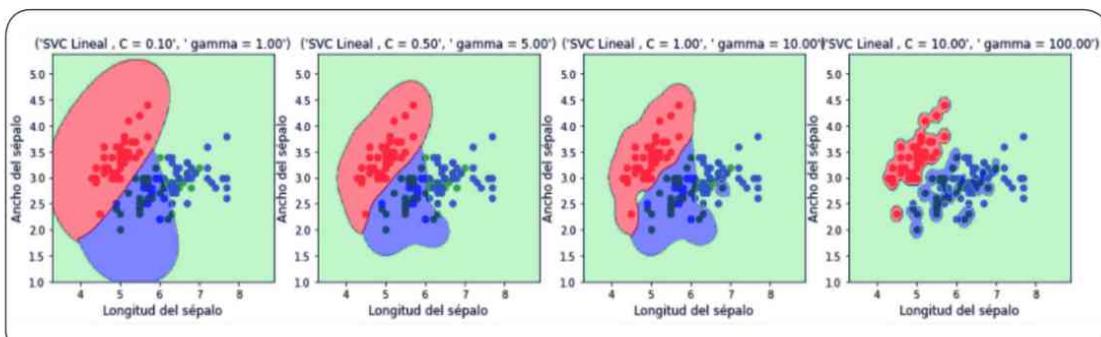
fig, subejes = plt.subplots(1, 4, figsize=(18, 4))
for subeje, c, g in zip(subejes, [0.1, 0.5, 1, 10],[1,5,10,100]):
    clf = SVC(kernel='rbf', C = c, gamma = g, random_state = 67).fit(X_ent, y_ent)
    titulo = 'SVC Lineal , C = {:.2f}'.format(c) , ' gamma = {:.2f}'.format(g)
    colors = ['red','green','blue']

    for color, i, target in zip(colors, [0, 1, 2], iris.target_names):
        subeje.scatter(X_ent[y_ent==i, 0], X_ent[y_ent==i, 1], color=color,
label=target, alpha=.9)

    subeje.set_xlabel('Longitud del sépalo', fontsize=12)
    subeje.set_ylabel('Ancho del sépalo', fontsize=12)
    subeje.set_title(titulo)

    colors = ('red', 'blue', 'lightgreen')
    cmap = ListedColormap(colors[:len(np.unique(y))])
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, 0.02),
                           np.arange(x2_min, x2_max, 0.02))
    Z = clf.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    subeje.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    subeje.set_xlim(xx1.min(), xx1.max())
    subeje.set_ylim(xx2.min(), xx2.max())

```



CAPÍTULO 8

Modelos de Clasificación II

Temas

- 8.1 Árboles de decisión
- 8.2 Bosques aleatorios (Random forest)
- 8.3 Adaboost (Adaptive boosting)
- 8.4 Gradient boosting
- 8.5 Naive Bayes
- 8.6 K vecinos más cercanos (KNN)
- 8.7 Sistemas de recomendación
- 8.8 Entrenamiento mediante aprendizaje en línea

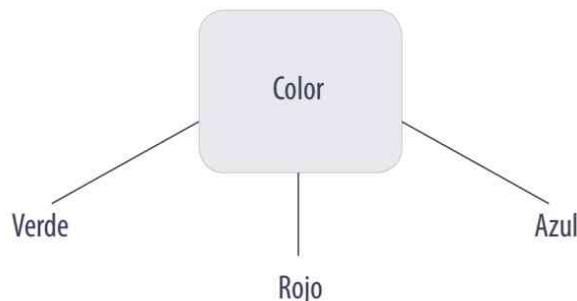
8.1 Árboles de decisión

Un árbol de decisión es un algoritmo de clasificación que también se puede usar para regresión y sirve entre otras cosas para conocer cuáles son las características más influyentes de un conjunto de datos. Así mismo, los árboles de decisión son adecuados cuando se desea mostrar cómo fluye un proceso basado en decisiones. Son muy populares debido a que su representación en forma de árbol es intuitiva y fácil de interpretar, además permiten manipular datos multidimensionales con una buena precisión. Y, por si fuera poco, pueden usarse sobre datos sin normalizar puesto que cada atributo es manejado de forma independiente y no hay una sujeción con respecto a la escala del mismo. Los árboles de decisión se componen de una serie de nodos que manejan unas reglas con las cuales toman decisiones a partir de la evaluación de los valores de los atributos de las muestras, dependiendo de si dichos valores están por encima o debajo de un umbral las muestras son separadas en subconjuntos o ramas. Típicamente el árbol se construye partiendo el conjunto de datos original en subconjuntos, repitiendo este particionamiento de manera recursiva en cada subconjunto separado, hasta que todas las muestras en cada subconjunto comparten la misma etiqueta, es decir, sean homogéneas. Cuando se han alcanzado los nodos hojas, entonces el árbol está listo para clasificar nuevas muestras.

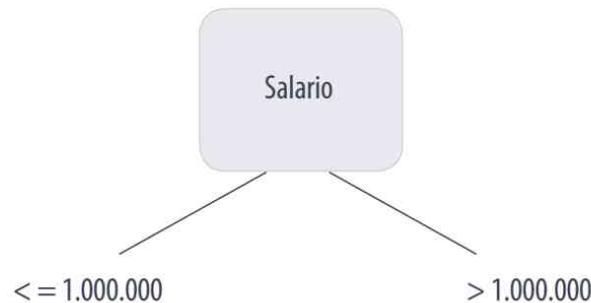
En resumen, a cada nodo interno le atañe una evaluación, a las ramas el resultado de una evaluación y a cada nodo hoja le corresponde definir la clasificación de una muestra.

El particionamiento de un árbol de decisión en subconjuntos se realiza con base a lo que se conoce como criterios de separación y este depende del tipo de dato del atributo, pudiendo ser:

1. Discreto: Se hace una separación para cada posible valor que tiene el atributo:
Ejemplo:



2. Continuo: El atributo A se compara con el valor medio de las dos instancias adyacentes, generando dos salidas correspondientes a las condiciones: $A \leq valor_medio$ y $A > valor_medio$. Ejemplo:



3. Discreto y binario: Se separan en los dos posibles valores del atributo A. Ejemplo:



En los árboles de decisión cuando se usan para clasificación la variable destino es de tipo categórica, tal como: si/no, bueno/regular/malo, etc. En cambio, en regresión la variable destino es una variable continua. Siendo necesario para construirlos llevar a cabo los siguientes pasos de forma recursiva: 1. Escoger un atributo del conjunto de datos; 2. Calcular el atributo más significativo para la separación de los datos, siendo el que logre la mayor pureza (mayor cantidad de muestras pertenecientes a una misma clase) en los nodos subsiguientes. Un nodo es puro cuando todas sus muestras pertenecen a una misma clase; 3. Separar los datos basados en el valor del mejor atributo. 4. Ir al paso 1.

Miremos un ejemplo donde iremos mostrando los pasos para crear un árbol de decisión que permita saber si a una persona le es aprobado o no un préstamo, con base a las variables explicativas *tiene_casa* e *ingreso_anual* del siguiente conjunto de datos:

Id Cliente	Tiene_Casa	Ingreso_anual	Aprobado
1	No	Bajo	No
2	Si	Medio	SI
3	Si	Alto	SI
4	No	Alto	SI
5	Si	Bajo	Si

Id Cliente	Tiene_Casa	Ingreso_anual	Aprobado
6	No	Bajo	No
7	No	Medio	No
8	Si	Alto	Si
9	No	Alto	Si
10	No	Bajo	No

Como se había mencionado previamente, en cada nodo se construyen reglas a partir de preguntas que se van haciendo con los valores de las características de las muestras (tuplas). Dichas preguntas se constituyen a base de condiciones del tipo si... si...sino, etc. Lo cual sirve para determinar cómo las tuplas de un nodo dado serán separadas. Estas reglas se crean con el entrenamiento del algoritmo y se pueden usar para tomar decisiones sobre futuras muestras.

Observando las instancias del dataset anterior podemos deducir a modo de ejemplo las siguientes reglas:

- si (tiene_casa) entonces aprobado='SI'
- si (no tiene_casa y ingreso_Anual='Bajo') entonces aprobado='No'

Considere el siguiente esquema de árbol de decisión basado en el concepto de reglas anteriormente explicado:



Ahora veamos qué pasa con dos muestras interesadas en acceder a un préstamo.

Tiene_Casa	Ingreso_anual	Aprobado
No	Bajo	?
Si	Medio	?

En el anterior árbol de decisión como se podrá dar cuenta el lector solo la segunda muestra tendrá una decisión final favorable, es decir aprobado "SI", pues cumple, con la condición establecida en la primera regla (*si(tiene_casa)* entonces *aprobado='SI'*).

Otro aspecto importante de este método de aprendizaje automático es el de seleccionar los atributos que otorgan una mejor separación de los conjuntos de datos en particiones. Si todas las tuplas que caigan en una partición dada pertenecen a la misma clase se dice que la partición es pura. Consideremos el siguiente ejemplo:



Como apreciamos, cuando el atributo *tiene_casa* tiene el valor "si", 4 muestras cumplen con esta condición, perteneciendo todas ellas a la misma clase (*préstamo='si'*), conformando una partición pura. Vemos que *tiene_casa='si'* produce resultados homogéneos, cosa que no ocurre con *tiene_casa='no'* que produce resultados mezclados, ya que 4 tuplas pertenecen a la clase 'no' y 2 a la clase 'si'. ¿Pero cómo podemos medir si un atributo brinda una mejor homogeneidad en los resultados que otro? Para eso existen unos enfoques matemáticos conocidos como métricas de medición de la separación.

Es importante mencionar que el proceso de hallar las reglas de decisión se realiza recursivamente hasta encontrar que todos los nodos hojas sean puros o se alcance un umbral definido por el programador o el científico de datos.

8.1.1. Métricas para medir la separación

Las métricas usadas para separar un árbol de decisión son básicamente: Impureza de Gini y ganancia de información:

- La impureza de Gini: Mide la tasa de impureza de la distribución de clases de los puntos de datos. La impureza de Gini en un nodo *n*, se calcula con la siguiente ecuación:

$$G(n) = 1 - \sum_{i=1}^c p_i^2$$

donde p_i es la probabilidad de que una muestra en el nodo n pertenezca a una clase c . Cuando se maneja partición binaria se suman el cálculo de la impureza de cada partición. Por ejemplo, si una partición D la separamos en D_1 y D_2 , entonces la impureza de Gini en D se calcularía así:

$$G(D) = \frac{[D_1]}{D} G(D_1) + \frac{[D_2]}{D} G(D_2)$$

El subconjunto que posea la mínima impureza de Gini para ese atributo es seleccionado como subconjunto de separación del atributo.

Teniendo en cuenta el conjunto de datos anterior, tenemos aprobados = 6 y no aprobados = 4. Luego las probabilidades de que las muestras pertenezcan a las clases 'SI' y 'NO' respectivamente son: $P(\text{aprobado}=\text{'SI'})=6/10$ y $P(\text{aprobado}=\text{'NO'})=4/10$, luego la impureza de Gini en este caso será:

$$G(n) = 1 - \sum_{i=1}^c p_i^2 = 1 - \left[\left(\frac{6}{10} \right)^2 + \left(\frac{4}{10} \right)^2 \right] = 0.48$$

Para encontrar el atributo más significativo para la separación de los datos es necesario obtener la impureza de Gini para cada uno de los atributos. Este método considera una separación binaria para cada atributo, y el que tenga la menor impureza será el elegido.

Con estos conocimientos, miremos ahora como podemos crear un árbol de decisión paso a paso, vamos a partir de aplicar la impureza de gini sobre los atributos para conocer cuál será el nodo raíz, y luego lo subdividiremos de manera recursiva hasta alcanzar los nodos inferiores. Comencemos con el atributo *tiene_casa*, el cual como sabemos tiene dos posibles valores "si" y "no", así que se calcula la impureza de Gini para cada uno de estos subconjuntos y para el nodo padre hallamos el promedio ponderado.

$$G_{\text{tiene_casa} \in \{\text{si}\}}(D) = \left[1 - \left(\frac{4^2}{4} + \frac{0^2}{4} \right) \right] = 0$$

$$G_{\text{tiene_casa} \in \{\text{no}\}}(D) = \left[1 - \left(\frac{2^2}{6} + \frac{4^2}{6} \right) \right] = 0.44$$

$$G_{\text{tiene_casa} \in \{\text{si}, \text{no}\}}(D) = \frac{[D_1]}{D} G(D_1) + \frac{[D_2]}{D} G(D_2) = \frac{4}{10} * 0 + \frac{6}{10} * 0.44 = 0.27$$

Miremos ahora cual es la impureza para los subconjuntos del atributo *ingreso_anual*:

$$G_{\{ingreso_anual\} \in \{bajo\}}(D) = 1 - \left[\left(\frac{1}{4}\right)^2 + \left(\frac{3}{4}\right)^2 \right] = 1 - 0.06 + 0.56 = 0,375$$

$$G_{ingreso_anual \in \{medio\}}(D) = 1 - \left[\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2 \right] = 1 - 0.5 = 0,5$$

$$G_{\{ingreso_anual\} \in \{alto\}}(D) = 1 - \left[\left(\frac{4}{4}\right)^2 + \left(\frac{0}{4}\right)^2 \right] = 1 - 1 + 0 = 0$$

$$G_{ingreso_anual \in \{bajo, medio, alto\}}(D) = \frac{4}{10} * 0,375 + \frac{2}{10} * 0,5 + \frac{4}{10} * 0 = 0,25$$

Como se aprecia el atributo *ingreso_anual* tiene una impureza menor en comparación con el atributo *tiene_casa*, de igual manera el subconjunto 'alto' en este mismo criterio tiene un valor de 0, en tanto, se escoge como atributo separador, dando lugar a la definición de la regla *ingreso_anual* = 'Alto' para este primer nodo.



Como vemos en la rama de la izquierda todas las instancias pertenecen a la misma clase aprobado="si" por lo que la pureza es máxima y no debe realizarse más separación en la misma.

Sin embargo, la rama de la derecha no es homogénea, por ello toca sacarle la impureza de Gini a estas 6 instancias considerando como atributo separador *tiene_casa* similar a como hicimos en el paso anterior, este es un proceso que como ya se mencionó se hace de manera recursiva hasta alcanzar que todas las instancias de un nodo pertenezcan a una misma clase o se llegue a un umbral determinado. Este paso lo realizamos así:

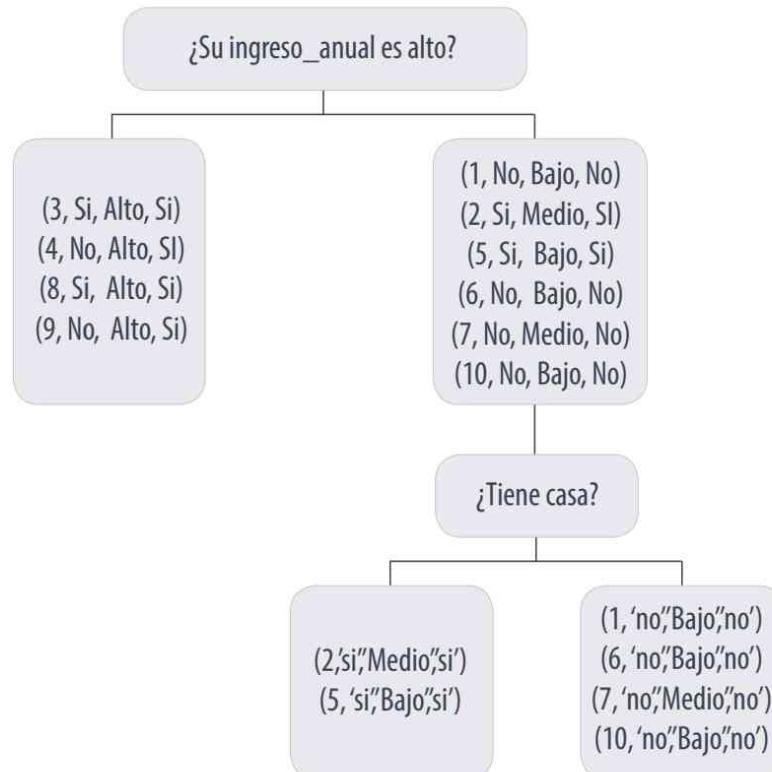
$$G_{tiene_casa \in \{si\}}(D) = \left[1 - \left(\frac{2^2}{2} + \frac{0^2}{2} \right) \right] = 0$$

$$G_{tiene_casa \in \{no\}}(D) = \left[1 - \left(\frac{0^2}{4} + \frac{4^2}{4} \right) \right] = 0$$

$$G_{tiene_casa \in \{si,no\}}(D) = \frac{[D_1]}{D} G(D_1) + \frac{[D_2]}{D} G(D_2) = \frac{2}{6} * 0 + \frac{4}{6} * 0 = 0$$

Al hacer los cálculos nos damos cuenta que este resultado 0 nos indica que el nodo es homogéneo, lo cual se puede confirmar si analizamos los datos de la rama, ya que todas las muestras con `tiene_casa = 'no'` se clasifican como 'no', mientras que las restantes con `tiene_casa='si'` se clasifican como 'si'.

Luego de haber realizado este último paso ya hemos concluido la construcción de nuestro árbol de decisión, puesto que ya no se pueden hacer más subdivisiones.



- Entropía: La entropía básicamente mide la mezcla o impureza en un conjunto de muestras. Se calcula con la siguiente fórmula:

$$E(x) = -\sum_{i=1}^n P(x_i) * \log_2 P(x_i)$$

donde $P(x_i)$ es la probabilidad de que una tupla x_i pertenezca a una clase c .

Si hay un conjunto de muestras donde todas pertenecen a la misma clase entonces se dice que el conjunto es completamente puro. En caso ideal se habla de una entropía de 0, ya que $-1 * \log_2 1 = 0$. En caso de que por ejemplo se tenga un 50 % de impureza la entropía es de 1. ($-0.5 * \log_2 0.5 = 1$).

Consideremos la partición inicial E donde hay 6 aprobados="Si" y 4 aprobados = 'No', la entropía será:

$$E = -\sum_{i=1}^n P(x_i) * \log_2 P(x_i) = -\left(\frac{6}{10} \log_2 \left(\frac{6}{10}\right) + \frac{4}{10} \log_2 \left(\frac{4}{10}\right)\right) = 0.971$$

Por lo anterior, el árbol de decisión hará una separación de los datos en subconjuntos donde la entropía sea reducida. Reducir la entropía basándose en un criterio de separación o regla se conoce como ganancia de información. Cuando se crea un nodo la meta mientras se realiza la separación es la de obtener la mayor ganancia de información.

Calculemos la entropía para el atributo *tiene_casa* e *ingreso_anual* respectivamente, solo con fines demostrativos ya que en la práctica no tiene mucho sentido aplicar más de una métrica para la creación de un árbol de decisión:

$$E_{tiene_casa}(D) = \frac{4}{10} \left[-\left(\frac{4}{4} \log_2 \frac{4}{4} + \frac{0}{4} \log_2 \left(\frac{0}{4}\right) \right) \right] + \frac{6}{10} \left[-\frac{2}{6} \left(\log_2 \left(\frac{2}{6}\right) + \frac{4}{6} \log_2 \left(\frac{4}{6}\right) \right) \right] = 0.083$$

$$\begin{aligned} E_{ingreso_anual}(D) &= \frac{4}{10} \left[-\left(\frac{1}{4} \log_2 \frac{1}{4} + \frac{3}{4} \log_2 \left(\frac{3}{4}\right) \right) \right] + \frac{2}{10} \left[-\frac{1}{2} \left(\log_2 \left(\frac{1}{2}\right) + \frac{1}{2} \log_2 \left(\frac{1}{2}\right) \right) \right] + \\ &\quad \frac{4}{10} \left[-\frac{4}{4} \left(\log_2 \left(\frac{4}{4}\right) + \frac{0}{4} \log_2 \left(\frac{0}{4}\right) \right) \right] = 0.075 \end{aligned}$$

Ahora que tenemos claro qué es la entropía miremos como podemos hallar la ganancia de información. Esta se puede hallar así:

$$\text{Ganancia1} = E(\text{Padre}) - E(\text{Hijo1}) = 0,971 - 0,08 = 0,89$$

$$\text{Ganancia2} = E(\text{Padre}) - E(\text{Hijo1}) = 0,971 - 0,075 = 0,896$$

Este último resultado advierte tal y como habíamos deducido con la métrica anterior, que el atributo *ingreso_anual* es preferible para separar el conjunto de datos pues en este caso representa la mayor ganancia de información. Así pues, ya sea que se use la impureza de Gini o la información de ganancia ambos miden la impureza ponderada de los nodos hijos después de una separación.

8.1.2. Crear y visualizar árboles de decisión con Scikit-learn

Ha llegado el momento de crear un árbol de decisión con la clase `DecisionTreeClassifier` de Scikit-learn con nuestro conjunto de datos sobre aprobación de préstamos, configurando como criterio la ganancia de Gini y una profundidad de 3 (`max_depth=3`), además hemos tenido que codificar los valores de las características a números, ya que como recordará el lector los algoritmos Scikit trabajan con variables numéricas.

```
import numpy as np
from sklearn.tree import DecisionTreeClassifier

X = np.array([['No', 'Bajo'],
              ['Si', 'Medio'],
              ['Si', 'Alto'],
              ['No', 'Alto'],
              ['Si', 'Bajo'],
              ['No', 'Bajo'],
              ['No', 'Medio'],
              ['Si', 'Alto'],
              ['No', 'Alto'],
              ['No', 'Bajo']
             ])
y = [0, 1, 1, 1, 1, 0, 0, 1, 1, 0]

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
X[:,0] = le.fit_transform(X[:,0])
X[:,1] = le.fit_transform(X[:,1])
arbol = DecisionTreeClassifier(criterion='gini', max_depth=3, random_state=1)
arbol.fit(X, y)
```

Ahora la predicción para alguien con tiene_casa='Si' y tiene_ingreso='Bajo' es de 1, como se observa si ejecutamos las siguientes sentencias:

```
prueba = np.array(['Si','Bajo'])
prueba_cod = le.fit_transform(prueba)
prediccion = arbol.predict(prueba_cod.reshape(1 , -1))
print("Predicción:", prediccion[0]) #Predicción: 1
```

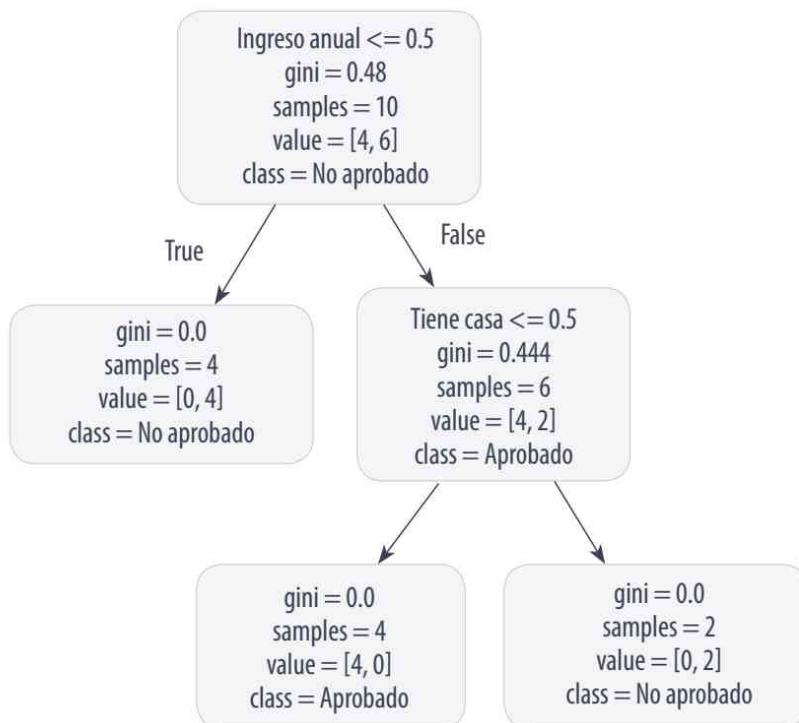
Podemos además mediante el método `export_graphviz()` de la biblioteca `tree` de Scikit-learn crear una representación gráfica de nuestro árbol de decisión desarrollado en el paso anterior.

```
from sklearn import tree
dot = tree.export_graphviz(arbol, feature_names=['Tiene_casa','Ingreso anual'],
                           class_names=['Aprobado','No aprobado'], out_file='arbol.dot')
```

Esto permite crear un archivo .dot el cual podemos pasar a imagen .png mediante el programa graphviz disponible en <https://www.graphviz.org/download/>. Para ello nos vamos a una ventana de línea de comandos y nos ubicamos en la carpeta donde se generó nuestro archivo arbol.dot y tecleamos el siguiente comando:

```
dot -Tpng arbol.dot -o arbol.png
```

Esta acción generará un archivo .png que contiene el árbol de decisión en cuestión:



Observemos en este último gráfico como hemos obtenido un árbol de decisión con la misma estructura del realizado manualmente, cada nodo padre tiene una regla, notamos que se diferencia en este caso en que el atributo se compara con un valor numérico (por ej: tiene_casa <= 0.5), el cual es un promedio de los valores de este atributo. Además, también se aprecia el calculó de la impureza de gini (0.0 para los nodos hoja), de igual manera aparecen la cantidad de muestras (samples), la cantidad de muestras por cada subconjunto (value) y la clase (class) con la que se podría clasificar una nueva instancia.

8.1.3. Identificación de características importantes

En este subapartado vamos a entrenar un algoritmo de árbol de decisión con el conjunto de datos Iris de la misma forma como lo hicimos con el ejemplo introductorio de este capítulo, y finalmente mostraremos cómo podemos identificar las características más importantes para nuestro modelo de aprendizaje.

```

from sklearn import datasets
iris = datasets.load_iris()

X = iris.data
y = iris.target
X_ent, X_pru, y_ent, y_pru = train_test_split(X, y, test_size=0.2, random_
state=1)

arbol = DecisionTreeClassifier(criterion ='gini', max_depth = 3)
arbol.fit(X_ent,y_ent)

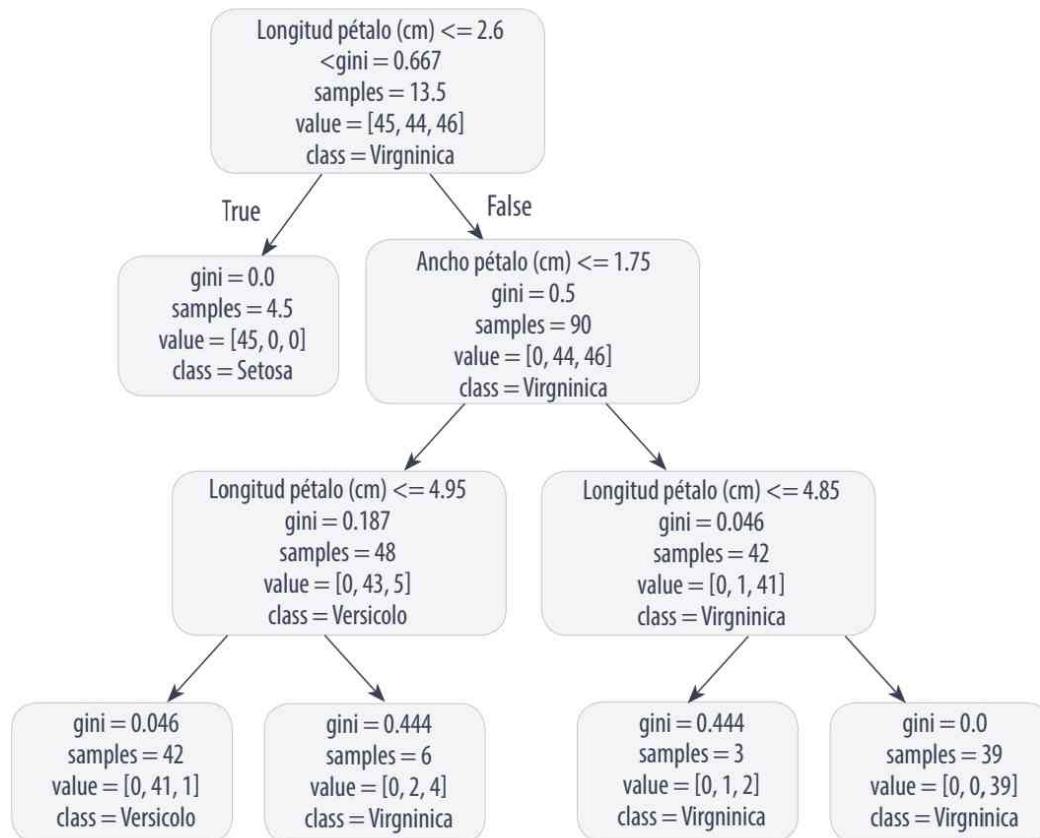
print("Exactitud del conjunto de entrenamiento: {:.2f}".format(arbol.score(X_
ent, y_ent)))
print("Exactitud del conjunto de prueba: {:.2f}".format(arbol.score(X_pru, y_
pru)))

```

Exactitud del conjunto de entrenamiento: 0.98

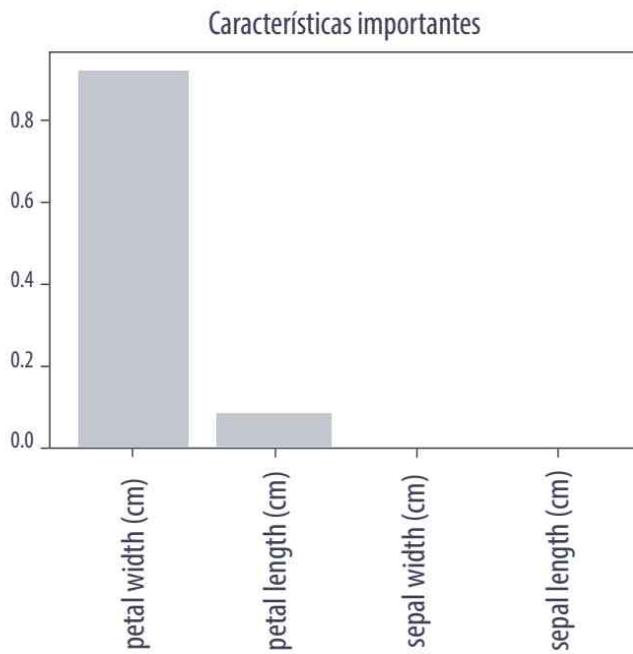
Exactitud del conjunto de prueba: 0.97

Hemos conseguido una exactitud de 0.97 con el conjunto de prueba, ahora si visualizamos nuestro árbol usando el mismo código del subapartado anterior tendríamos lo siguiente:



Intuitivamente una vez observada la imagen anterior podríamos decir que las características más importantes son aquellas que aparecen en las reglas de decisión y que son usadas por el modelo para separar las observaciones. Este análisis lo podemos confirmar por medio de la propiedad `feature_importances_` de nuestro objeto clasificador que nos devuelve un array con 4 valores numéricos que corresponde a las 4 variables del dataset y en donde entre más grande sea el número, mayor importancia tiene la característica asociada.

```
importantes = arbol.feature_importances_
# Ordenamos las características en orden descendente
indices = np.argsort(importantes)[::-1]
# Se reorganizan los nombres de las características
nombres = [iris.feature_names[i] for i in indices]
plt.title("Características Importantes")
plt.bar(range(X.shape[1]), importantes[indices])
plt.xticks(range(X.shape[1]), nombres, rotation=90)
plt.show()
```



8.2 Bosques aleatorios (*Random Forest*)

Los bosques aleatorios son una extensión de los árboles de decisión, en los cuales son incluidos un número mayor de árboles de decisión con el fin de aumentar la exactitud en un modelo de clasificación. Reciben este nombre debido a que cada árbol es entrenado a partir de un subconjunto de muestras de datos seleccionadas aleatoriamente y cada nodo toma un subconjunto aleatorio de atributos al determinar la mejor separación. Los bosques aleatorios hacen parte de un método muy conocido en aprendizaje automático conocido como *ensemble* (conjunto) que busca combinar varios algoritmos de aprendizaje entrenados con un subconjunto distinto de los datos de entrenamiento, y para obtener la salida final todos los modelos participan aportando su predicción individual. Si es una aplicación de regresión la predicción final se toma a partir de la media de todas las predicciones, y si es una aplicación de clasificación se toma la clase más frecuente. Esta técnica de ajuste de modelos del que hacen parte los bosques aleatorios se conoce como *bagging (empaquetado)*. Concretamente, los bosques aleatorios con relación a los árboles de decisión individual brindan una mejor generalización de los datos y tienen menor afectación por el sobreajuste.

Veamos cómo podemos implementar un bosque aleatorio mediante la clase RandomForestClassifier del paquete sklearn.ensemble de Scikit-learn sobre nuestro conjunto de datos Iris, tomando solo las columnas longitud del sépalo y ancho del sépalo para poder visualizarlo en un plano bidimensional. Pero antes vamos a entrenar un árbol de decisión simple.

```
X = iris.data
y = iris.target
X = X[:, [0,1]]
X_ent, X_pru, y_ent, y_pru = train_test_split(X, y, test_size=0.2, random_
state=1)

arbol = DecisionTreeClassifier(criterion = 'gini', max_depth = 3)
arbol.fit(X_ent,y_ent)

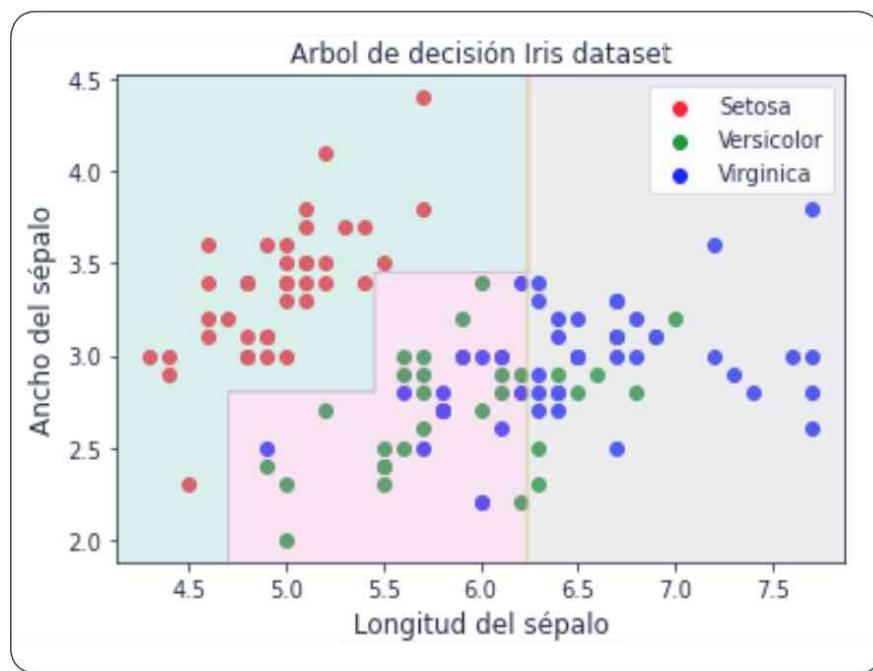
print("Exactitud del conjunto de entrenamiento: {:.2f}".format(arbol.score(X_
ent, y_ent)))
print("Exactitud del conjunto de prueba: {:.2f}".format(arbol.score(X_pru, y_
pru)))
```

Exactitud del conjunto de entrenamiento: 0.81

Exactitud del conjunto de prueba: 0.83

Dibujamos los puntos de datos y los límites de decisión:

```
plt.figure()
colors = ['red', 'green', 'blue']
for color, i, target in zip(colors, [0, 1, 2], iris.target_names):
    ax = plt.scatter(X_ent[y_ent==i, 0], X_ent[y_ent==i, 1], color=color,
                      label=target)
    xlim = plt.xlim()
    ylim = plt.ylim()
    xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                         np.linspace(*ylim, num=200))
    Z = modelo.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)
    contours = plt.contourf(xx, yy, Z, alpha=0.3, cmap='Set2')
    plt.legend(['Setosa', 'Versicolor', 'Virginica'], loc='best')
    plt.xlabel('Longitud del sépalo', fontsize=12)
    plt.ylabel('Ancho del sépalo', fontsize=12)
    plt.title("Arbol de decisión Iris dataset")
plt.show()
```



Ahora si llegó el momento de entrenar nuestro bosque aleatorio:

```
from sklearn.ensemble import RandomForestClassifier
bosque = RandomForestClassifier(random_state=0, n_jobs=-1, criterion='gini',
n_estimators=15)
bosque.fit(X_ent, y_ent)
print("Exactitud del conjunto de entrenamiento: {:.2f}".format(bosque.score(X_
ent, y_ent)))
print("Exactitud del conjunto de prueba: {:.2f}".format(bosque.score(X_pru, y_
pru)))
```

Exactitud del conjunto de entrenamiento: 0.93

Exactitud del conjunto de prueba: 0.83

En este ejemplo hemos entrenado un bosque aleatorio con 15 árboles de decisión, el criterio de separación ha sido la impureza de Gini, y establecimos un `n_jobs` de -1, entendiendo que vamos a parallelizar el entrenamiento entre varios núcleos de procesamiento para agilizar el rendimiento del bosque aleatorio, sin embargo, notamos que nuestro modelo ha quedado sobreajustado, esto ha quedado en evidencia al haber obtenido una mayor exactitud con los datos de entrenamiento que con los de prueba.

8.3 Adaboost (*Adaptative boosting*)

El boosting adaptativo es una alternativa a los árboles de decisión y a los bosques aleatorios, otorgan con frecuencia una mejor exactitud en los resultados. Los algoritmos *boosting* como Adaboost y Gradient boosting son también partidarios del método *ensemble*.

Especificamente, Adaboost consiste en entrenar un conjunto de modelos débiles (generalmente árboles de decisión de poca profundidad) de forma repetitiva, de tal manera que en cada iteración el modelo actual se enfoca en las muestras que el modelo anterior predijo incorrectamente:

Básicamente lo que hace AdaBoost es:

1. Asignar a cada muestra x_i un peso inicial de valor $w_i = \frac{1}{n}$, donde n es el numero total de observaciones.
2. Entrenar un modelo débil con los datos y estimar su tasa de error.
3. Para cada observación:
 - a. Si el modelo predice x_i incorrectamente se le asigna el mayor peso (w_i mayor)

- b. Si el modelo predice x_i correctamente se le asigna el menor peso (w_i menor)
- 4. Entrena un nuevo modelo débil donde las observaciones con w_i más altos se les da una mayor prioridad, es decir, tendrán una alta probabilidad de clasificación.
- 5. Se repiten los pasos 4 y 5 hasta que los datos son perfectamente clasificados o se alcance un umbral específico.

En Scikit-learn se puede usar este algoritmo por medio de la clase AdaBoostClassifier que recibe varios parámetros entre ellos: base_estimator, que es el algoritmo de aprendizaje a usar para entrenar los modelos (por defecto árbol de decisión); n_estimators, corresponde al número de modelos que se entrenarán; learning_rate, es la tasa de aprendizaje que determina la contribución de cada modelo (por defecto 1).

Miremos como funciona AdaBoostClassifier con el conjunto de datos Iris:

```
from sklearn.ensemble import AdaBoostClassifier

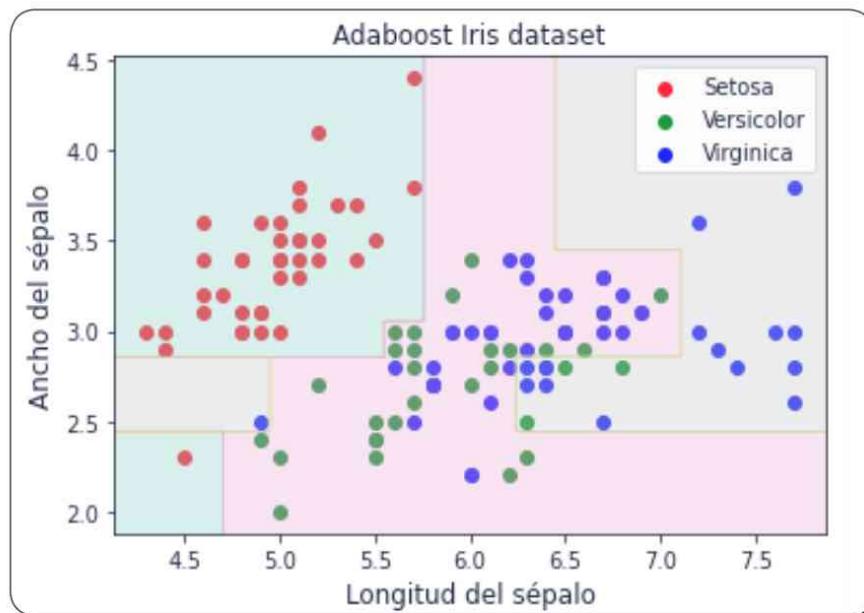
adaboost = AdaBoostClassifier(DecisionTreeClassifier(max_depth = 2),
                             n_estimators = 25,
                             learning_rate = 0.1,
                             random_state = 0)
model = adaboost.fit(X_ent, y_ent)

print("Exactitud del conjunto de entrenamiento: {:.2f}".format(model.score(X_ent, y_ent)))
print("Exactitud del conjunto de prueba: {:.2f}".format(model.score(X_pru, y_pru)))
```

Exactitud del conjunto de entrenamiento: 0.74

Exactitud del conjunto de prueba: 0.83

Podemos notar tal vez un mejor modelo con respecto al logrado con el bosque aleatorio y aunque la exactitud con el conjunto de prueba ha sido igual (0.83), el ejercicio nos ha permitido comparar dos técnicas de clasificación, algo bastante significativo, toda vez que en el mundo del machine learning es una práctica muy común comparar varios algoritmos y escoger el que ofrece un mejor rendimiento en materia de exactitud, eficiencia computacional, etc.



8.4 Gradient boosting

Del Gradient Boosting o Potenciación del Gradiente se puede decir que es una generalización del anterior y de igual manera está basado en un conjunto de árboles de decisión débiles (aprendices) entrenados secuencialmente, centrados en los errores que van dejando los árboles precedentes. En contraposición con el AdaBoost donde los modelos débiles se ajustan con los datos de entrenamiento, con *gradient boosting* dichos modelos son entrenados con los errores residuales (diferencia entre el valor de la variable objetivo y el valor predicho) cometidos por el aprendiz anterior. La predicción de una nueva muestra se obtiene utilizando las predicciones de cada árbol individual.

La clase `GradientBoostingClassifier` de Scikit es encarga de implementar este algoritmo. Sin entrar en más detalles, considere el siguiente fragmento de código donde hallamos la exactitud promedio al aplicar validación cruzada con 10 iteraciones a 50 objetos `GradientBoostingClassifier`.

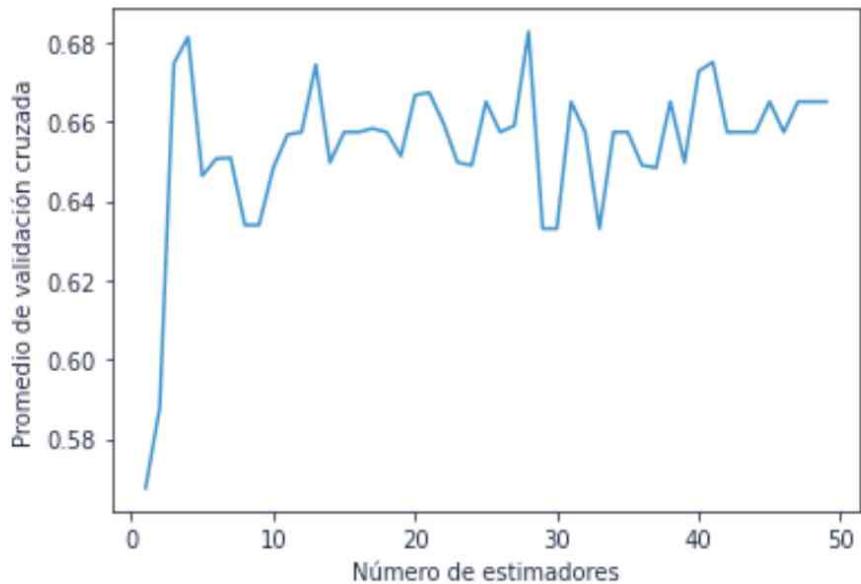
```

from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import cross_val_score

gradboost = GradientBoostingClassifier(n_estimators=10, learning_
rate=1.0,max_depth=1, random_state=0)
model = gradboost.fit(X_ent, y_ent)

puntajesvc = []
max_estimadores = 50
for i in range(1, max_estimadores):
    punt = cross_val_score(GradientBoostingClassifier(n_estimators=i,
    learning_rate=10.0/float(i)), X_ent, y_ent, cv=10, scoring='accuracy').mean()
    puntajesvc.append(punt)
plt.plot(range(1, max_estimadores) , puntajesvc)
plt.xlabel("Número de estimadores")
plt.ylabel("Promedio de validación cruzada")
plt.show()

```



La curva anterior, aunque muestra ciertos picos en la exactitud con cierto número de estimadores, permite evidenciar que el algoritmo de Potenciación del Gradiente no ha aportado una mejora significativa en la exactitud de nuestro modelo, pero sirva esta breve introducción para ampliarnos un poco más el panorama en cuanto a los algoritmos de *ensemble*, e instamos al lector a probar este y los otros métodos vistos, en futuras actividades de aprendizaje automático de las que haga parte.

8.5 Naive bayes

Otro algoritmo probabilístico bastante interesante y usado con frecuencia en tareas de aprendizaje automático, como la clasificación de correos no deseados (spam), análisis de sentimiento, clasificación de documentos, entre otros, es el algoritmo de Naive Bayes basado en el teorema de Bayes.

Este algoritmo tiene varias ventajas: una de ellas es que es fácil y rápido de entrenar, pudiendo ser entrenado con pequeños conjuntos de datos, situación que permite ser utilizado en aplicaciones de predicción en tiempo real. Otra característica es la posibilidad de usarse para tareas de clasificación tanto binaria como multiclase. Sin embargo, presenta también desventajas: una es que asume que las características en el conjunto de datos son completamente independientes entre sí, tal consideración no aplica en todos los casos, puesto que muchas veces se encuentran atributos con una alta relación o dependencia, lo cual es necesario considerar si se quieren rendimientos favorables en las predicciones. Por suponer esta independencia entre atributos, al algoritmo de Naive Bayes se le considera un clasificador inocente o ingenuo.

Antes de mirar el teorema de Bayes sobre el cual se basa el algoritmo en estudio, vamos a explicar el concepto de probabilidad condicional.

La probabilidad condicional consiste en determinar la probabilidad de ocurrencia de un evento A teniendo en cuenta la ocurrencia de un evento B. Esta probabilidad se calcula partiendo de dos sucesos A y B y se representa como $P(A / B)$, y se lee como "probabilidad de A dado B". No existiendo ninguna relación temporal necesariamente entre uno y otro. La probabilidad condicional se expresa de la siguiente manera:

$$P(A / B) = \frac{P(A \cap B)}{P(B)}, \text{ donde } P(A \cap B) \text{ es la probabilidad de que suceda el evento A y el B.}$$

Como ejemplo, considere una urna que tiene 4 bolas rojas y 6 azules. De las 4 bolas rojas 2 son lisas y 2 son rugosas, y de las 6 bolas azules 4 son lisas y los dos restantes son rugosas. Suponga que sin mirar se extrae una bola de la urna. ¿Si alguien nos dice que la bola es roja cual es la probabilidad de que esta sea lisa?

Sean los eventos A: "Bola lisa" y B: "Bola roja". Luego $P(A) = 6/10 = 3/5$ y $P(B) = 4/10$.

Como sabemos que la bola es roja, la probabilidad de que sea también lisa es: 1/2 ya que de las rojas la mitad es lisa. $P(A \cap B) = 1/2$.

$$P(A / B) = \frac{P(A \cap B)}{P(B)} = \frac{2/10}{4/10} = 20/40 = 1/2 = 0,5 = 50\%$$

Según esto la probabilidad de sacar una bola roja que también sea lisa es del 50%.

Con respecto al teorema de Bayes, se fundamenta en el aprendizaje basado en la experiencia. Esto es el cálculo de una probabilidad posterior de un evento A basado en probabilidades de eventos anteriores que llevaron al advenimiento de dicho evento.

Con la probabilidad condicional conocemos la probabilidad de los efectos a partir de las causas, mientras que con el teorema de Bayes conseguimos la probabilidad de las causas a partir de los efectos.

La fórmula de este teorema es la siguiente:

$$P(A | B) = \frac{P(B|A)P(A)}{P(B)}$$

Donde, $P(B|A)$ es la probabilidad del evento B dada la ocurrencia del evento A, $P(A|B)$, es la probabilidad del evento A dado que ocurrió el evento B, $P(A)$ y $P(B)$ representa la probabilidad de ocurrencia de los eventos A y B respectivamente.

Vamos a ilustrar el teorema de Bayes con el mismo ejemplo analizado anteriormente: Pero ahora hallando la probabilidad de sacar una bola roja de la urna dado que fue lisa.

Nuevamente sean los eventos A: "Bola lisa" y B: "Bola roja", entonces:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

$$P(A|B) = 0,5$$

$$P(B) = 4/10$$

$$P(A) = 6/10$$

$$\text{Luego, } P(B|A) = \frac{P(A|B)P(B)}{P(A)} = \frac{0,5 \cdot 0,4}{0,6} = 0,33$$

Ahora que tenemos una idea acerca del teorema de Bayes, miremos cómo podemos implementar un clasificador para decidir si un correo es spam o no, usando la biblioteca Scikit-learn. Para este ejercicio vamos a usar un dataset de juguete con dos columnas: la primera llamada *mensaje* con los mensajes de correo y la segunda llamada *etiqueta* con los valores 0 (No spam) y 1 (Spam). El lector puede conseguir el dataset junto con el material complementario del libro dentro de un archivo con el nombre de "spam_no_spam.csv". El conjunto de datos contiene la siguiente información:

mensaje, etiqueta
Hola ¿cómo estás?,0
Gana dinero sin trabajar,1
Hola Contáctame ahora,0
gana millones de dólares facilmente,1
quieres dinero gratis, 1
convírtete en millonario, 1
gana mucho dinero fácil, 1
necesitas dinero fácil, 1
hola hay reunión hoy, 0
reunión importante, 0

Ahora comenzamos como siempre cargando el conjunto de datos y mostramos las primeras 5 filas con la función head().

```
import pandas as pd
df = pd.read_csv('spam_no_spam.csv')
print(df.head())
```

	mensaje	etiqueta
0	Hola ¿cómo estás?	0
1	Gana dinero sin trabajar	1
2	Hola Contáctame ahora	0
3	gana millones de dolares facilmente	1
4	quieres dinero gratis	1

El siguiente paso es convertir los mensajes de texto a un formato numérico. Para ello usaremos la técnica conocida como bolsa de palabras (Bag of words), la cual consiste en tomar cada fragmento de texto y contar la frecuencia de palabras en el mismo. Al final obtendremos una matriz de frecuencia donde las filas corresponden a mensajes de correo y las columnas a las palabras que estos contienen, luego la intersección entre filas y columnas sería la cantidad de veces que aparece cada palabra en cada uno de los mensajes.

A manera de ejemplo supongamos los siguientes mensajes a los cuales determinaremos su matriz de frecuencia.

[‘Hola, ¿cómo estás?’,’Gana dinero sin trabajar,’,’Hola, Contáctame ahora’]

ahora	cómo	contáctame	dinero	estás	gana	hola	sin	trabajar
0	1	0	0	1	0	1	0	0
0	0	0	1	0	1	0	1	1
1	0	1	0	0	0	1	0	0

Para hacer este trabajo emplearemos el contador de vectorización (CountVectorizer) de Scikit-learn:

```
from sklearn.feature_extraction.text import CountVectorizer
mensajes = ['Hola, ¿cómo estás?', 'Gana dinero sin trabajar', 'Hola, Contáctame ahora']
cont_vect = CountVectorizer()
cont_vect.fit(mensajes)
nombres = cont_vect.get_feature_names()
print(nombres)
```

Resultado:

['ahora', 'contáctame', 'cómo', 'dinero', 'estás', 'gana', 'hola', 'sin', 'trabajar']

Seguidamente mediante el método transform() del objeto anterior obtenemos la matriz de frecuencias, así:

```
matriz = cont_vect.transform(mensajes).toarray()
print(matriz)
```

```
[[0 0 1 0 1 0 1 0 0]
 [0 0 0 1 0 1 0 1 1]
 [1 1 0 0 0 0 1 0 0]]
```

Al usar este tipo de mecanismos donde se realiza tratamiento de textos es muy recomendable excluir de la matriz las denominadas palabras vacías (*stop words*), es decir los artículos, pronombres, entre otros. Ejemplos: el, la, a, etc, pero no lo haremos por ser nuestro ejercicio muy básico, de todas formas, el lector puede conseguir en la web abundante información sobre cómo filtrar las mencionadas palabras como parte del procesamiento de datos.

Siguiendo con el ejemplo, el código siguiente permite cargar los datos en un dataframe de Pandas, pero ahora sin encabezados, conseguimos los datos para X e y, convirtiendo los datos de X en una matriz de frecuencia, y con ellos ajustamos un clasificador bayesiano con la clase MultinomialNB, la cual es la más adecuada para trabajar con datos categóricos:

```

import pandas as pd
pf = pd.read_csv('spam_no_spam.csv', header=0)
X = pf.iloc[:,0]
y = pf.iloc[:,1]

cont_vect = CountVectorizer()
X_t = cont_vect.fit_transform(X)
from sklearn.naive_bayes import MultinomialNB
nb = MultinomialNB()
nb.fit(X_t, y)

```

Posterior al entrenamiento nos disponemos a probar la predicción arrojada por nuestro clasificador con la cadena de prueba “Hola amigo hay reunión importante”, obteniendo como resultado 0, lo cual indica que nuestro mensaje de correo no es spam.

```

mensaje_prueba = np.array(['Hola amigo hay reunión importante'])
datos_pru = cont_vect.transform(mensaje_prueba)

prediccion = nb.predict(datos_pru)
print("Predicción: %.0f" %prediccion)

```

Predicción: 0

8.6 K Vecinos más cercanos (KNN)

El algoritmo K vecinos más cercanos es otro algoritmo ampliamente usado para realizar tareas tanto de clasificación como de regresión y es muy sencillo de entender. En la literatura del aprendizaje automático es considerado un algoritmo “flojo”, básicamente porque no realiza el entrenamiento de un modelo propiamente mediante iteraciones, sino que la clase a la que pertenece una muestra se establece a partir de la obtención de las clases de sus k vecinos más cercanos, escogida por mayoría de votos. Por ejemplo, si una muestra de clase desconocida está cerca de 5 observaciones de clase 1 y cerca de 3 de clase 2, entonces dicha muestra es clasificada como de clase 1. Este método es usado bastante en sistemas de recomendación, detección de fraudes, entre otros. Y entre sus principales desventajas está, que como toma todo el conjunto de datos para ajustar un modelo resulta muy costoso computacionalmente, por lo que se le puede sacar un mejor provecho con datasets pequeños y con un número no tan elevado de características.

Los pasos de este algoritmo se pueden resumir así:

1. Calcular la distancia entre la muestra a clasificar y el resto de muestras del conjunto de datos.
2. Seleccionar los k vecinos más cercanos con respecto a la muestra, utilizando alguna función para hallar la distancia, la más común es la distancia euclídea cuya fórmula es la que sigue:

$$D_{euclídea} = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

Pero existen otras como la distancia Manhattan cuya fórmula es:

$$D_{manhattan} = \sum_{i=1}^n |x_i - y_i|$$

También existe otra métrica llamada distancia de Minkowski:

$$D_{minkowski} (\sum_{i=1}^n |x_i - y_i|^p)^{1/p}$$

En todas estas ecuaciones x_i e y_i son las muestras a las que se le va a calcular la distancia. Además, en la ecuación de Minkowski el parámetro p es un hiperparámetro que si le asignamos 1 aplica la distancia de Manhattan y si le asignamos 2 aplica la distancia euclídea.

3. Mediante voto mayoritario entre los k puntos que estén más cerca de la muestra se escoge la clase a la que esta última pertenece. En otras palabras, clasificamos el nuevo punto de acuerdo a la clase que tienen la mayoría de sus k vecinos.

Observemos como podemos implementar este algoritmo con la biblioteca Scikit-learn, la cual coloca a nuestra disposición la clase KNeighborsClassifier. Usaremos una vez más el conjunto de datos Iris.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

iris = datasets.load_iris()
X = iris.data
y = iris.target

X = X[0:100 , [1,2]]
y = y[0:100]

X_ent, X_pru, y_ent, y_pru = train_test_split(X, y, test_size=0.2, random_state=1)

sc = MinMaxScaler()
X_ent = sc.fit_transform(X_ent)
X_pru = sc.transform(X_pru)

knn = KNeighborsClassifier(n_neighbors=5 , metric='euclidean')
knn.fit(X_ent, y_ent)
print("Exactitud: %.2f" %knn.score(X_pru, y_pru))
```

Con esta configuración del clasificador utilizando 5 vecinos (`n_neighbors=5`), y la métrica de distancia euclidiana (`metric='euclidean'`) obtenemos una exactitud de 1.

Podemos encontrar la predicción para una muestra de prueba, clasificándola correctamente en la clase 1 (`versicolor`):

```
import numpy as np
prueba = np.array([1 , 1])
prediccion = knn.predict(prueba.reshape(1, -1))
print("Predicción %.2f" %prediccion)
```

Si deseamos saber cuáles son los vecinos más cercanos a nuestra muestra de prueba podemos escribir el siguiente código:

```
distancias, indices = knn.kneighbors(prueba.reshape(1,-1))
print("5 Vecinos más cercanos:")
for i, j in enumerate(indices[0][:5]):
    print (X_ent[j])
```

5 Vecinos más cercanos:

```
[0.58333333 0.85365854]
[0.5     0.92682927]
[0.5     0.90243902]
[0.45833333 0.90243902]
[0.41666667 0.97560976]
```

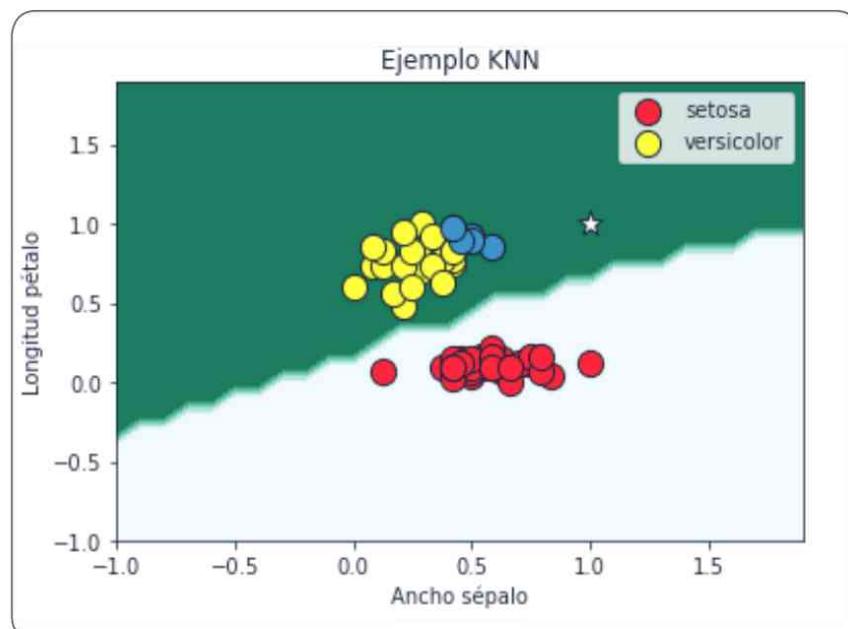
Intentemos con el siguiente código graficar los puntos de entrenamiento, el punto de muestra y los puntos vecinos más cercanos a él.

```
from matplotlib.colors import ListedColormap
X1, X2 = np.meshgrid(np.arange(start = X_ent[:, 0].min() - 1, stop = X_ent[:, 0].max() + 1, step = 0.1),
                     np.arange(start = X_ent[:, 1].min() - 1, stop = X_ent[:, 1].max() + 1,
                               step = 0.1))
plt.contourf(X1, X2, knn.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
              cmap = plt.cm.BuGn)
colors = ['red','yellow']
for color, i, target in zip(colors, [0, 1], iris.target_names):
    ax = plt.scatter(X_ent[y_ent==i, 0], X_ent[y_ent==i, 1], color=color,
                     label=target, s=150, edgecolor='black')

for i in indices:
    plt.scatter(X_ent[i, 0], X_ent[i, 1], marker='o', s=150, edgecolor='black')

plt.scatter(1, 1, marker='*', s=150, color='white', edgecolor='black')

plt.title('Ejemplo KNN')
plt.xlabel('Ancho sépalo')
plt.ylabel('Longitud pétalo')
plt.legend()
plt.show()
```



En el siguiente subapartado trataremos de crear un pequeño sistema de recomendación, aprenderemos conceptos sobre esta clase de programas muy utilizados por las empresas actualmente y de paso también nos servirá para poner en práctica algunos aspectos estudiados sobre el algoritmo de k vecinos más cercanos.

8.7 Sistemas de recomendación

Un sistema de recomendación como su nombre sugiere es un sistema de software que recomienda elementos informáticos o temas a los usuarios finales, dichos elementos recomendados pueden ser, por ejemplo: películas, artículos, noticias entre otros. Un ejemplo de la vida real es el que encontramos en el portal de películas en streaming Netflix que cuenta con un motor de recomendación para sugerir a los usuarios las películas que más se asemejan a sus gustos particulares. En esta aplicación el usuario inicialmente escoge algunas producciones de su gusto personal y el sistema le sugiere otras que guardan relación con las anteriores.

Hay dos tipos de sistemas de recomendación: sistema de recomendación basados en contenido y sistema de recomendación colaborativo.

8.7.1. Sistemas de recomendación basados en contenido

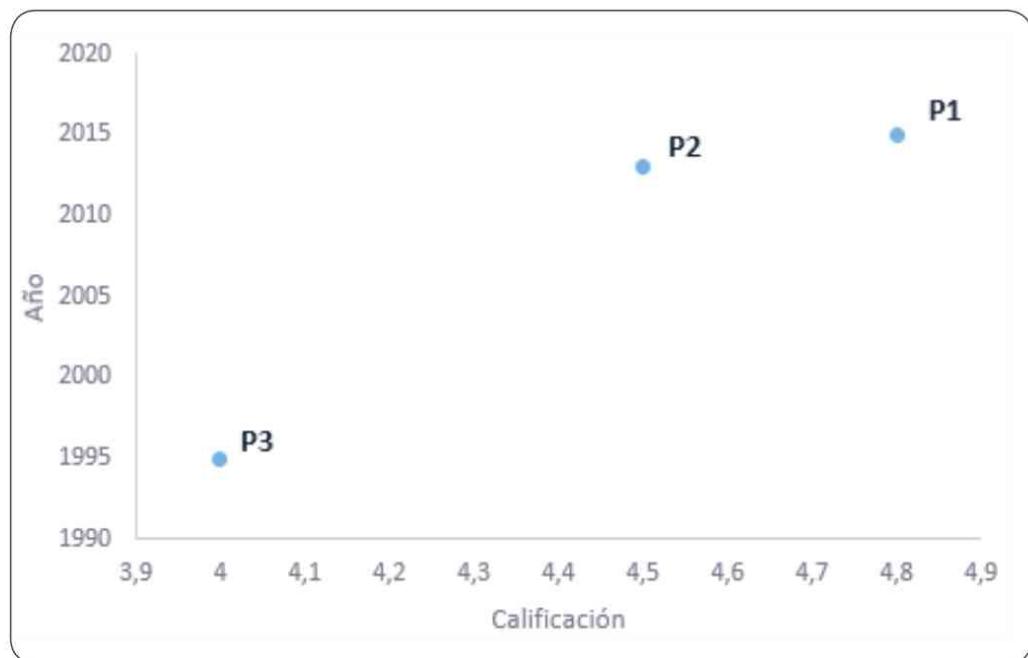
En los sistemas de recomendación basados en contenido, el sistema busca similitudes en los elementos basándose en los atributos o en el contenido de estos. Por ejemplo, en un sistema de recomendación de películas se tomarían en cuenta

atributos como nombre del director, calificación, genero, entre otros. Para hallar la similitud entre los atributos se suelen usar diferentes técnicas ya conocidas, tales como la distancia euclíadiana y el coeficiente de Pearson.

Para ilustrar estas técnicas vamos a considerar las siguientes muestras de prueba, suponiendo que desarrollaremos un sistema de recomendación de películas:

Película	Atributos
Guerra de las galaxias	Calificación: 4,8, Año=2015
Rápido y furioso	Calificación: 4,5, Año=2013
Toy story	Calificación: 4, Año: 1995

Si colocamos los anteriores atributos en un plano de dos dimensiones, donde el eje x será la *calificación* y el eje y el atributo *año* tendríamos una gráfica como la siguiente:



A simple vista se puede apreciar que hay una cercanía entre los puntos P1 y P2 lo cual significa una mayor similitud entre las películas Guerra de las galaxias (P1) y Rápido y furioso (P2).

Existen dos métodos para encontrar tal similitud entre dos puntos. Ellos son la distancia euclíadiana y el coeficiente de Pearson.

- **Distancia euclíadiana:** Está basado en el teorema de Pitágoras, y consiste en hallar la longitud de la hipotenusa del triángulo rectángulo conformado por

los puntos y los lados proyectados sobre los ejes x e y a nivel de la hipotenusa. Utilizando la fórmula de la distancia euclíadiana podemos encontrar la distancia entre los tres puntos expuestos:

$$D_{euclíadiana} = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

Calculemos la distancia entre la película P1 y la P2, tendríamos:

$$D_{(P1,P2)} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} = \sqrt{(4.8 - 4.5)^2 + (2015 - 2013)^2} = 2.02$$

Y ahora la distancia entre la película P1 y P3:

$$D_{(P1,P3)} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} = \sqrt{(4.8 - 4)^2 + (2015 - 1995)^2} = 20,01$$

Como se puede dar cuenta el lector la distancia entre los puntos P1 y P2 es mucho menor a la distancia que hay entre P1 y P3, demostrando la similitud entre los atributos *calificación* y *año* de las películas "Guerra de las galaxias y Rápido y furioso".

Este algoritmo implementado en Python sería como sigue:

```
import numpy as np
import json

def distancia_euclíadiana(X, p1, p2):
    dif_cuadrados = []

    for i in X[p1]:
        if i in X[p2]:
            dif_cuadrados.append(np.square(X[p1][i] - X[p2][i]))

    #return 1 / (1 + np.sqrt(np.sum(dif_cuadrados)))
    return np.sqrt(np.sum(dif_cuadrados))

datos = {"Guerra de las galaxias": {"Calificación":4.8 , "Año":2015} ,
         "Rapido y furioso": {"Calificación":4.5 , "Año":2013},
         "Toy story 4": {"Calificación":4 , "Año":1995}
         }

datos_cod = json.dumps(datos)
decod = json.loads(datos_cod)
```

Si llamamos la función anterior con dos películas de prueba: "Guerra de las galaxias" y "Rápido y furioso" obtendremos una distancia euclíadiana de 2.02 que es el mismo resultado obtenido matemáticamente.

```

pel1 = 'Guerra de las galaxias'
pel2 = 'Rápido y furioso'

d = distancia_euclidiana(datos, peli1, peli2)
print("Distancia euclíadiana: {:.2f}".format(d))

```

Distancia euclíadiana: 2.02

- **Coeficiente de Pearson:** Es otra técnica usada para encontrar la similitud entre puntos de datos a partir de la búsqueda de una relación lineal entre los mismos. Básicamente busca determinar la línea que mejor ajusta o separa los puntos de datos tomados en consideración.

Para hallar el coeficiente de correlación de Pearson se usa la siguiente fórmula:

$$r = \frac{n * \sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n \sum x_i^2 - (\sum x_i)^2} \sqrt{n \sum y_i^2 - (\sum y_i)^2}}$$

En esta fórmula el numerador se le conoce como covarianza S_{xy} y el denominador es el producto de las desviaciones típicas de X (S_x) e Y (S_y). Además, r es el coeficiente de correlación de Pearson, x es un conjunto de valores (x_1, x_2, x_3, \dots) que en nuestro ejemplo son (4.8, 4.5, 4), e y es el conjunto de valores (y_1, y_2, y_3, \dots) en el ejemplo (2015, 2013, 1995).

Como ya habíamos mencionado en el capítulo 5, el coeficiente de Pearson permite medir la relación entre dos variables, pudiendo tomar un valor comprendido entre -1 y 1. Si este es igual a 1 quiere decir que existe una relación lineal directa entre las variables, es decir que si una de ellas aumenta la otra también. Si el valor es de -1 la relación es inversa lo que significa que si una variable aumenta la otra disminuye, en cambio, un coeficiente 0 refleja no asociatividad entre dichas variables.

Miremos la fórmula del coeficiente de Pearson paso a paso con nuestros datos de prueba, consideraremos nuevamente las dos primeras películas:

	X	Y	XY	X ²	Y ²
	4,8	2015	9.672	23,04	4.060.225
	4,5	2013	9.058,5	20,25	4.052.169
Σ	9,3	4.028	18.730,5	43,29	8.112.394

Luego,

$$r = \frac{n * \sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n \sum x_i^2 - (\sum x_i)^2} \sqrt{n \sum y_i^2 - (\sum y_i)^2}} = \frac{2 * 18.730,5 - 9,3 * 4.028}{\sqrt{2 * 43,29 - 86,49} \sqrt{2 * 8.112.394 - 16.224.784}} = \\ \frac{37461 - 37.460,4}{0,3 * 2} = \frac{0,6}{0,6} = 1$$

Tratemos de implementar una función en Python que calcule el coeficiente de Pearson y notaremos que, al llamarla con los datos de prueba respectivos, obtenemos el mismo resultado logrado matemáticamente:

```
def coef_corr(X, pelis):
    n = len(pelis)
    p1 = pelis[0]
    p2 = pelis[1]

    for j in X[p1]:
        datos[j] = ( [X[p1][j], X[p2][j]] )

    sum_x = np.sum( datos['Calificacion'])
    sum_y = np.sum( datos['Año'])
    sum_cua_x = np.sum( np.square( datos['Calificacion']))
    sum_cua_y = np.sum( np.square( datos['Año']))
    suma_prod = np.dot( datos['Calificacion'], datos['Año'])

    Sxy = n * suma_prod - (sum_x * sum_y)
    Sxx = n * sum_cua_x - np.square(sum_x)
    Syy = n * sum_cua_y - np.square(sum_y)

    if Sxx * Syy == 0:
        return 0
    return Sxy / np.sqrt(Sxx * Syy)
pelis = np.array(['Guerra de las galaxias', "Rapido y furioso'])
```

```
r = coef_corr(datos, pelis)
print("Coeficiente de correlación: %.2f" %r)
```

Coeficiente de correlación: 1.00

De igual manera este resultado lo podemos comprobar mediante la función de numpy *corrcoef*, ya utilizada en el capítulo 5:

```
c = np.corrcoef([4.8, 2015], [4.8 , 2013])[0, 1]
print("Coeficiente de correlación {:.2f}".format(c))
```

Coeficiente de correlación 1.00

Con estos conocimientos básicos sobre las técnicas más usadas para encontrar la similitud entre puntos, miremos como podemos aplicarlas en un buscador de películas similares. Para esto usaremos un dataset real extraído del proyecto GroupLens, disponible en <http://movielens.org>. Con el fin de hacer los más simple el ejercicio se tomarán solo 5 muestras de películas y las calificaciones dadas por los usuarios sobre ellas. Esta información se encuentra en dos archivos películas.csv y puntajes.csv incluidos dentro del material complementario del libro.

El conjunto de datos de películas tiene las columnas *movie_id*, *title*, *genres*, pero solo utilizaremos *movie_id*, *title*.

```
columnas = ['movie_id','title']
peliculas = pd.read_csv('dataset_películas/películas.csv',
                       names=columnas, usecols=range(2))
peliculas
```

	movie_id	title
0	1	Toy Story (1995)
1	122886	Star Wars: Episode VII - The Force Awakens (2015)
2	102716	Fast & Furious 6 (Fast and the Furious 6, The)...
3	95510	Amazing Spider-Man, The (2012)
4	120799	Terminator Genisys (2015)

Ahora quitamos el año del título de las películas y lo colocamos en otra columna:

```
peliculas['year'] = peliculas.title.str.extract('(\d\d\d\d)',expand=False)
#Removemos los parentesis
peliculas['year'] = peliculas.year.str.extract('(\d\d\d\d)',expand=False)
#Removemos los años del titulo
peliculas['title'] = peliculas.title.str.replace('(\d\d\d\d)', '')
peliculas['title'] = peliculas['title'].apply(lambda x: x.strip())
print(peliculas.head())
```

	movie_id	title	year
0	1	Toy Story	1995
1	122886	Star Wars: Episode VII - The Force Awakens	2015
2	102716	Fast & Furious 6	2013
3	95510	Amazing Spider-Man	2012
4	120799	Terminator Genisys	2015

Por su parte, el conjunto de datos de puntajes tiene las columnas userId, movieId, rating, timestamp, pero solo utilizaremos las tres primeras.

```
columnas = ['user_id','movie_id','rating']
puntajes = pd.read_csv('dataset_peliculas/puntajes.csv',
                      names=columnas, usecols=range(3))
puntajes.sample(5)
```

	user_id	movie_id	rating
2718	9929	122886	4.5
324	1160	102716	3.5
2900	10592	1	4.0
1556	5620	1	4.0
369	1300	1	2.0

Mezclamos la información de los dataframes películas y puntajes por medio de la función merge(). Con lo cual podemos ver principalmente la puntuación dada por cada usuario a las películas vistas por él.

```
puntajes = pd.merge(peliculas, puntajes)
puntajes.sample(5)
```

	movie_id		title	year	user_id	rating
2998	95510	Amazing Spider-Man	2012	8303	2.0	
1263	1	Toy Story	1995	5040	4.0	
1742	1	Toy Story	1995	7174	4.0	
2966	95510	Amazing Spider-Man	2012	5630	2.0	
801	1	Toy Story	1995	3220	4.0	

Ahora necesitamos conocer la relación existente entre película y usuario junto con las puntuaciones que este último les ha conferido. Para ello Pandas ofrece la función pivot_table():

```
puntajePeliculas = puntajes.pivot_table(index=['user_id'],
                                         columns=['title'],values='rating')
puntajePeliculas.head()
```

user_id	title	Amazing Spider-Man	Fast & Furious 6	Star Wars: Episode VII - The Force Awakens	Terminator Genisys	Toy Story
7		NaN	NaN		5.0	NaN
15		NaN	NaN		NaN	NaN
17		NaN	NaN		NaN	5.0
23		5.0	NaN		NaN	5.0
28		NaN	NaN		NaN	4.0

Note, que algunas películas aparecen con un puntaje NaN indicando que el usuario identificado con *user_id*, no realizó puntuación sobre la película cuyo título aparece como nombre en una de las columnas.

Con el dataframe puntajePeliculas podemos obtener fácilmente las puntuaciones dadas por los usuarios sobre cualquier película.

```
puntajeToyStory = puntajePeliculas['Toy Story']
puntajeToyStory.head()
```

```

user_id
7      NaN
15     4.0
17     5.0
23     5.0
28     4.0
Name: Toy Story, dtype: float64

```

Finalmente, obtenemos las películas similares a Toy Story con la función corrwith().

```

peliculasSimilares = puntajePeliculas.corrwith(puntajeToyStory)
peliculasSimilares = peliculasSimilares.dropna().sort_values(ascending=False)
peliculasSimilares.head(3)

```

```

title
Toy Story          1.000000
Fast & Furious 6  0.379573
Amazing Spider-Man 0.365757
dtype: float64

```

8.7.2. Sistema de recomendación basado en filtro colaborativo

Como su nombre lo indica este tipo de sistema de recomendación está basado en el aporte de los usuarios sobre determinados ítems de información como productos o películas, dada su influencia en aspectos como, por ejemplo: la cantidad de búsquedas realizadas, clicks, calificaciones, etc, lo cual se usa para recomendar elementos a un usuario de entrada. Este modelo trata de encontrar preferencias similares a las del usuario de entrada y a continuación le recomienda posibles elementos de su gusto o interés.

A modo de ejemplo, suponga que un usuario A vio las películas “Guerra de las galaxias”, “Rápido y furioso”. Y un usuario B solo vio la primera de estas películas. Si ambos le dieron una calificación alta a la película en común, estos usuarios serán considerados similares y el sistema le recomendará al usuario B la película “Rápido y furioso”, basado en el hecho de que otro usuario con preferencias parecidas (Usuario A) vio también esta producción.

Vemos un ejemplo de un sistema de filtro colaborativo utilizando el mismo dataset del ejercicio anterior. Para ello primeramente definimos algunas funciones utilitarias, siendo `correlacion_pearson()` la más destacada ya que permite encontrar el coeficiente de correlación entre dos usuarios.

```

def obtener_puntaje(id_usu,id_peli):
    return (puntajes.loc[(puntajes.user_id==id_usu) & (puntajes.movie_id == id_peli)]['rating'].iloc[0])

def obtener_idPelis(userid):
    return (puntajes.loc[(puntajes.user_id==userid)]['movie_id'].tolist())

def obtener_titPeli(movieid):
    return (peliculas.loc[(peliculas.movie_id == movieid)]['title'].iloc[0])

def correlacion_pearson(usuario1,usuario2):
    cont = []
    # Películas vistas por ambos usuarios
    for e in puntajes.loc[puntajes.user_id==usuario1]['movie_id'].tolist():
        if e in puntajes.loc[puntajes.user_id==usuario2]['movie_id'].tolist():
            cont.append(e)

    if len(cont) == 0:
        return 0

    # Calculando covarianzas
    sumPuntajeUsuario1 = sum([obtener_puntaje(usuario1,e) for e in cont])
    sumPuntajeUsuario2 = sum([obtener_puntaje(usuario2,e) for e in cont])
    cuadPuntajeUsuario1 = sum([pow(obtener_puntaje(usuario1,e),2) for e in cont])
    cuadPuntajeUsuario2 = sum([pow(obtener_puntaje(usuario2,e),2) for e in cont])
    prodSumPuntajes = sum([obtener_puntaje(usuario1,e) * obtener_puntaje(usuario2,e) for e in cont])

    n = len(cont)

    # Correlación de Pearson
    numerador = n * prodSumPuntajes - (sumPuntajeUsuario1 * sumPuntajeUsuario2)

```

```
denominador=np.sqrt(n*cuadPuntajeUsuario1-pow(sumPuntajeUsuario1,2))  
* np.sqrt(n * cuadPuntajeUsuario2 - pow(sumPuntajeUsuario2,2) )  
  
if denominador == 0:  
    return 0  
return numerador/denominador
```

Probemos la función anterior con los usuarios 23 y 34, consiguiendo como resultado -1.

```
print('Correlación de Pearson entre usuarios con ids 23 y 34:{}'.format(correlacion_pearson(23,34)))
```

Finalmente, con la función obtener_recomendaciones() obtenemos una lista de películas recomendadas para el usuario 2921.

```
def obtener_recomendaciones(idUsuario):  
    ids_usuarios = puntajes.user_id.unique().tolist()  
  
    total = {}  
    suma_sim = {}  
  
    # Iteración a través de un subconjunto de IDs de usuarios  
    for usuario in ids_usuarios[100]:  
  
        # No se tiene en cuenta el mismo usuario  
        if usuario == idUsuario:  
            continue  
  
        # Obteniendo similitud entre usuarios  
        coef = correlacion_pearson(idUsuario,usuario)  
  
        #if coef <= 0:  
        #continue  
  
        # Obteniendo calificación de similitud ponderada y suma de las similituds  
        # entre ambos usuarios.  
        for idPeli in obtener_idPelis(usuario):  
            # Se tienen en cuenta solo las películas no vistas o calificadas
```

```

if idPeli not in obtener_idPelis(idUsuario) or obtener_puntaje(idUsuario,i
dPeli) == 0:
    total[idPeli] = 0

    total[idPeli] += obtener_puntaje(usuario,idPeli) * coef
    suma_sim[idPeli] = 0
    suma_sim[idPeli] += coef

# Normalizamos los puntajes
clasif = [(tot/suma_sim[idPeli],idPeli) for idPeli,tot in total.items()]
clasif.sort()
clasif.reverse()

# Se obtienen los titulos de las películas
recomendaciones = [obtener_titPeli(idPeli) for coef,idPeli in clasif]
return recomendaciones[:5]
print(obtener_recomendaciones(2921))

```

[‘Star Wars: Episode VII - The Force Awakens’, ‘Amazing Spider-Man’, ‘Toy Story’]

Siendo la anterior predicción bastante aceptable, pero como siempre es recomendable trabajar con muchos más datos para obtener resultados mas convincentes.

8.8 Entrenamiento mediante aprendizaje en línea

El entrenamiento en línea es aquel que se realiza mediante el paso de los datos al modelo en tiempo real. Donde una porción pequeña de los datos son los que se usan para el entrenamiento inicial, lo cual repercute en un ahorro significativo de la memoria del computador. Posterior a este entrenamiento previo el modelo es actualizado con los conjuntos de datos más recientes.

Veamos un ejemplo donde apliquemos este nuevo concepto, en el cual inicialmente crearemos un conjunto de datos para clasificación con la clase make_classification con 1.100.000 puntos de datos, luego creamos un clasificador SGD con un número máximo de iteraciones (n_iter) a 1 para entrenarlo con una pequeña porción de los datos ya que estamos usando regresión en línea, y por último luego con un ciclo vamos ajustando el modelo parcialmente con 100.000 muestras por cada iteración:

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import accuracy_score

filas = 1100000
X, y = make_classification(n_samples=filas,
n_features=2, n_redundant=0, n_informative=1,
n_clusters_per_class=1)

X_ent, X_pru, y_ent, y_pru = train_test_split(X, y, test_size=0.1, random_
state=1)

enc = OneHotEncoder(handle_unknown='ignore')
enc.fit_transform(X)

sgd_rl = SGDClassifier(alpha=0.01, loss='log', max_iter=1, learning_
rate='constant', eta0=0.01)

for i in range(10):
    x_ent = X_ent[i*100000:(i+1)*100000]
    y_train = y_ent[i*100000:(i+1)*100000]
    x_train_enc = enc.transform(x_ent)
    sgd_rl.partial_fit(x_train_enc, y_train, classes=[0, 1])

x_test_enc = enc.transform(X_pru)
pred = sgd_rl.predict(x_test_enc)
print("Exactitud %.3f" %accuracy_score(y_pru, pred))
```

Finalmente, al probar el modelo con los datos de prueba, obtendremos una exactitud de 0.50 aproximadamente.

CAPÍTULO 9

Clustering

Temas

- 9.1 K Medias
- 9.2 Clustering jerárquico
- 9.3 DBSCAN

El *clustering* o agrupamiento pertenece al grupo de técnicas de aprendizaje no supervisado y lo conforman un conjunto de algoritmos que buscan básicamente encontrar patrones o tendencias en los datos para luego crear grupos basándose en las características en común de dichos datos. El agrupamiento tiene muchas aplicaciones en la vida real como por ejemplo segmentación de clientes, sistemas de recomendación, etc.

Son varios los algoritmos que hacen parte de esta categoría del aprendizaje no supervisado, entre ellos tenemos el k-medias (en inglés, k-means), el agrupamiento jerárquico y el dbscan.

9.1 K Medias

El algoritmo k means pertenece al ámbito de aprendizaje no supervisado, esto básicamente significa que no es necesario etiquetar nuestros datos de entrenamiento, sino que el algoritmo se encarga de hallar las características en común que estos tienen y crea k grupos donde va separando las muestras que comparten esos rasgos similares.

Los pasos del algoritmo son básicamente los siguientes:

1. Especificar la cantidad k de grupos o cluster a generar.
2. Cada grupo tendrá un punto central llamado centroide, los cuales inicialmente

se seleccionan aleatoriamente.

3. Una vez se tengan k centroides, se agrupan en un cluster las muestras del conjunto de datos que comparten el centroide más cercano formando un grupo. Dicha cercanía es medida usualmente mediante la distancia euclídea.

No obstante existen otras medidas como la similitud de coseno, distancia promedio, entre otros.

4. Mover los centroides hacia el centro de cada cluster actualizando su posición a partir de las medias de todas las muestras del cluster al que pertenecen. Por esta razón a este algoritmo se le llama k-means.
5. Se repiten los pasos 3 y 4 hasta que el modelo converja, que los centroides alcancen un movimiento mínimo o se alcance un número máximo de iteraciones establecido por el programador.

Con todo esto lo que busca el algoritmo es minimizar las distancias de las muestras dentro del cluster y maximizar la distancia entre clústers. El kmeans es muy fácil de aplicar en Scikit-learn puesto que, no es necesario pasarle el arreglo de etiquetas al momento de realizar el entrenamiento, así que tampoco es necesario separar el conjunto de datos en subconjuntos de entrenamiento y pruebas.

Miremos ahora un ejemplo donde pongamos a prueba el algoritmo de agrupamiento k-means, para lo cual vamos a tomar el conjunto de datos llamado Breast Cancer Wisconsin tomado de los datasets proporcionados por Scikit-learn.

```
# se importan las librerías necesarias
import pandas as pd
import numpy as np
from sklearn.datasets import load_breast_cancer
# se cargar el dataset
datos = load_breast_cancer()
```

Para nuestro ejercicio, lo primero por hacer es importar las librerías requeridas y el dataset en cuestión, el cual es de tipo Bunch, algo muy similar a un diccionario.

Las características de este dataset las podemos ver por medio de la clave *feature_names*, así:

```
datos.feature_names
```

```
array(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
       'mean smoothness', 'mean compactness', 'mean concavity',
       'mean concave points', 'mean symmetry', 'mean fractal dimension',
       'radius error', 'texture error', 'perimeter error', 'area error',
       'smoothness error', 'compactness error', 'concavity error',
       'concave points error', 'symmetry error',
       'fractal dimension error', 'worst radius', 'worst texture',
       'worst perimeter', 'worst area', 'worst smoothness',
       'worst compactness', 'worst concavity', 'worst concave points',
       'worst symmetry', 'worst fractal dimension'], dtype='<U23')
```

El siguiente paso es convertir el dataset en un dataframe de Pandas para un mejor análisis y manejo de los datos.

```
df = pd.DataFrame(datos.data, columns=datos.feature_names)
df["target"] = datos.target
```

De estas 31 características se tomarán para hacer el ejemplo más sencillo, *mean radius* (radio medio) y *mean perimeter* (perímetro medio):

```
X = np.array(list(zip(df['mean radius'], df['mean perimeter'])))
```

En este línea de código la función `zip()` crea tuplas con los valores de *mean_radius* y *mean_perimeter* y la función `list()` crea una lista con estas tuplas, para finalmente obtener un array de Numpy.

Ahora si podemos crear un objeto de la clase KMeans y lo entrenamos con el array anterior. Observe que por medio del parámetro *n_clusters* establecemos el valor de k (número de clusters). Por otro lado, vemos también que no es necesario pasar el array con las etiquetas de clase esperadas como sucede con los algoritmos de aprendizaje supervisado.

```
from sklearn.cluster import KMeans
# creamos un objeto clasificador
knn = KMeans(n_clusters=2)
#entrenamos el modelo
knn.fit(X)
# obtenemos y mostramos los 2 centróides donde cada uno es un array de 31
valores
centroides = knn.cluster_centers_
print(centroides)
# obtenemos y mostramos el cluster al que pertenece cada punto
etiquetas = knn.labels_
print(etiquetas)
```

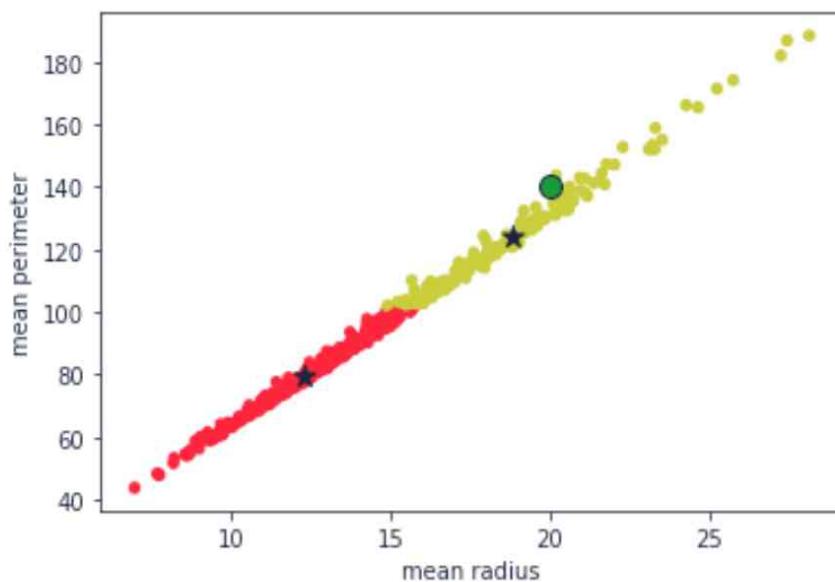
Si intentamos hacer una predicción con una muestra con radio medio=20 y perímetro medio = 140, obtendriamos que pertenece al cluster 1.

```
muestra = [[20,140]]
pred = knn.predict(muestra)
print("Muestra", muestra, "se encuentra en el clúster:", pred)
```

Muestra [[20, 140]] se encuentra en el clúster: [1]

Podemos ahora hacer una gráfica de dispersión para mirar como es la distribución de nuestros datos, los dos centróides encontrados y la muestra de prueba, así:

```
# hacemos una gráfica con los datos
import matplotlib.pyplot as plt
c = ['r','y']
colors = [c[i] for i in etiquetas]
plt.scatter(dff['mean radius'],dff['mean perimeter'], c=colors, s=20)
plt.scatter(centroides[:, 0], centroides[:, 1], marker='*', s=100, c='black')
plt.scatter(muestra[0][0], muestra[0][1], marker='o', s=100, c='green',
edgecolor='black')
plt.xlabel("mean radius")
plt.ylabel("mean perimeter")
```



A continuación estudiaremos dos métodos más efectivos para estimar el valor óptimo del número de clusters k : El método del codo y el coeficiente de silueta.

Método del codo (Elbow)

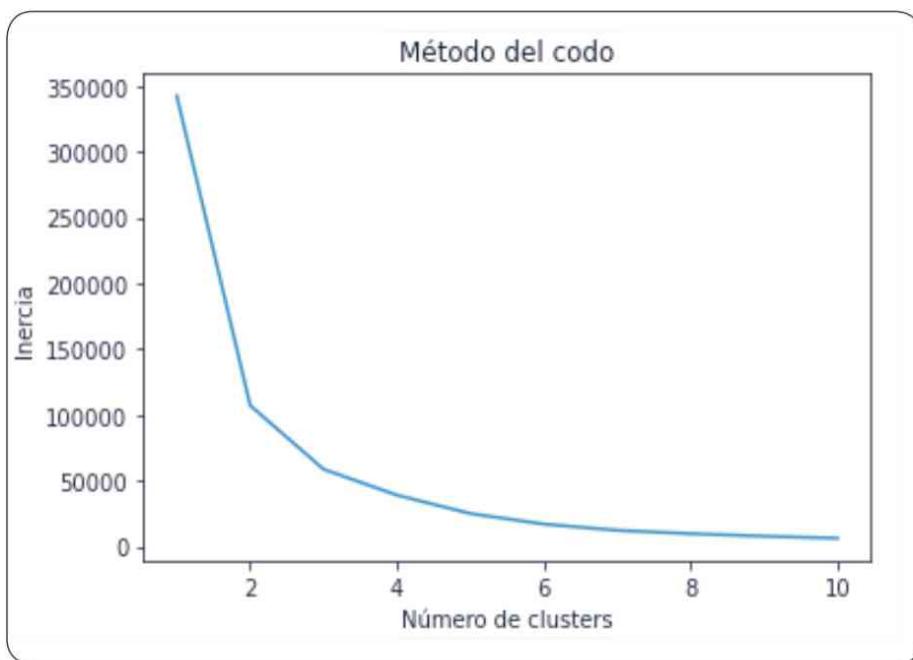
Quizás el aspecto más crítico de usar el algoritmo k-means es encontrar el valor óptimo para el hiperparámetro k , ya que en la práctica nos podemos encontrar con conjuntos de datos dimensionalmente altos en contraposición al que hemos mostrado en el anterior subapartado, resultando casi imposible poder visualizarlo graficamente y más aún determinar por simple inspección el mejor valor de k .

El método del codo usa los valores de la inercia una vez se ha aplicado k-means con un número n de clústers. La inercia es la suma de las distancias al cuadrado de cada punto del cluster a su centroide. Matemáticamente se expresa de la siguiente forma:

$$\text{Inercia} = \sum_{i=1}^n \|x_i - \mu\|^2$$

Los valores de la Inercia los podemos obtener mediante el atributo `inertia_` del objeto `kmean`, luego si la graficamos con respecto al número de clusters la apariencia del gráfico lineal se asemejaría a un brazo, entonces el punto donde notemos una atenuación (el codo del brazo) en la inercia representará el número óptimo de clusters. Observemos como podemos aplicar este concepto.

```
from sklearn.cluster import KMeans
inercias = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)
    kmeans.fit(X)
    inercias.append(kmeans.inertia_)
# Grafica de la suma de las distancias
plt.plot(range(1, 11), inercias)
plt.title('Método del codo')
plt.xlabel('Número de clusters')
plt.ylabel('Inercia')
plt.show()
```



En este caso podríamos pensar que el “codo” se consigue cuando el número de clusters se encuentra entre 2 y 3. Notemos que este método nos da un rango con los posibles valores óptimos de k , sin embargo, el método del coeficiente de silueta, que veremos a continuación es mucho más preciso con respecto a la determinación de este hiperparámetro.

Coeficiente de silueta

El coeficiente de silueta es una medida de la calidad de los cluster o grupos resultantes en un modelo. Mide la cohesión de los grupos, el cual es el espacio entre ellos. El coeficiente está dado por:

$$S = \frac{b - a}{\max(a, b)}$$

Donde a , es la distancia promedio de un punto a todos los demás puntos del mismo cluster, b es la distancia promedio de un punto a todos los demás puntos en el cluster más cercano. Además, S puede tomar valores entre -1 y 1: donde 1 (valor ideal) significa que la muestra está más cerca de su propio cluster y lejos de los demás clústeres, por su parte -1 que una muestra ha sido asignada al cluster equivocado y 0 indica una superposición de grupos, es decir que la diferencia entre clusters es casi nula. El valor de k con el cual se alcanza el coeficiente de silueta más alto se considera el k óptimo. Como calcular manualmente el valor de S para cada punto puede resultar bastante tedioso y computacionalmente muy costoso, optaremos en el siguiente ejemplo por usar el método `silhouette_score()` del módulo `metrics` de Scikit-learn. Recuerde que con esta métrica cuanto más cercano a 1 sea su valor mucho mejor será nuestro modelo:

```
from sklearn import metrics
exactitud = metrics.silhouette_score(X, etiquetas)
print("Coeficiente de silueta: {:.2f}".format(exactitud))
```

Coeficiente de silueta: 0.64

Este valor nos muestra que nuestro modelo es aceptable. En todo caso se puede intentar afinar aún más probando con otros valores para k y con otros hiperparámetros para encontrar un mejor ajuste. Sin embargo, para este ejemplo con $k=2$ se obtiene una mejor precisión del modelo. Miremos cómo:

```
# podemos obtener el valor óptimo de k entre 2 y 10
proms_silueta = []
k_ini = 2
for k in range(k_ini, 10):
    kmean = KMeans(n_clusters=k).fit(X)
    exactitud = metrics.silhouette_score(X, kmean.labels_)
    print("Coeficiente de silueta para k =", k, "es {:.2f}".format(exactitud))
    proms_silueta.append(exactitud)
# el valor de k óptimo es el que tiene el promedio más alto
k_optimo = proms_silueta.index(max(proms_silueta)) + k_ini
print("K óptimo:", k_optimo)
```

Coeficiente de silueta para $k = 2$ es 0.64

Coeficiente de silueta para $k = 3$ es 0.54

Coeficiente de silueta para $k = 4$ es 0.52

Coeficiente de silueta para $k = 5$ es 0.53

Coeficiente de silueta para $k = 6$ es 0.54

Coeficiente de silueta para $k = 7$ es 0.56

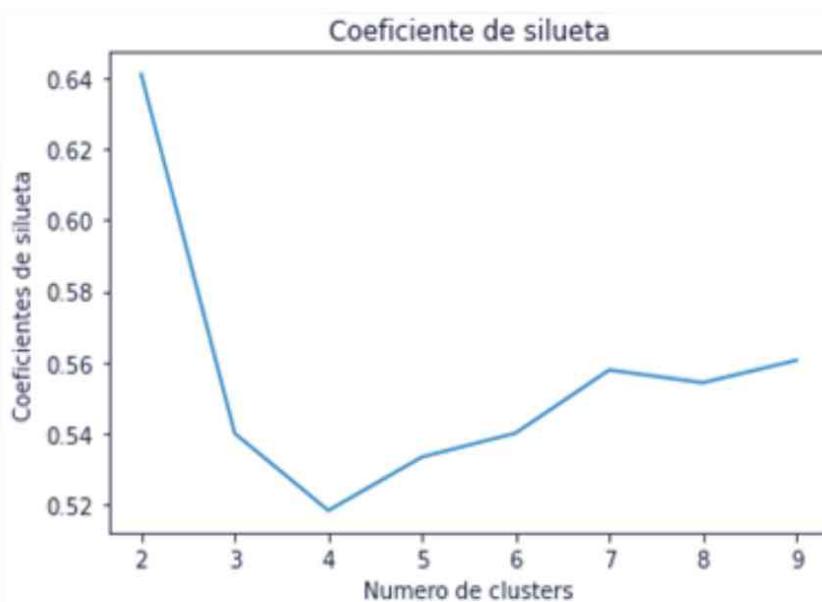
Coeficiente de silueta para $k = 8$ es 0.56

Coeficiente de silueta para $k = 9$ es 0.56

K óptimo: 2

Si dibujamos los valores de k en el rango entre 2 y 10 y los coeficientes de la variable *proms_silueta* tenemos una gráfica que de igual manera nos confirma que con $k=2$ se alcanza el mas alto coeficiente de silueta.

```
plt.plot(range(k_ini, 10), proms_silueta)
plt.title("Coeficiente de silueta")
plt.xlabel("Numero de clusteres")
plt.ylabel("Coeficientes de silueta")
```



9.2 Clustering jerárquico

El *clustering* jerárquico es otro método de agrupamiento de datos en *clusters* basándose en la distancia y busca que los datos de un grupo sean lo más similares entre sí. Su interpretación es sencilla ya que la visualización de los elementos se corresponde con la de un árbol. Tiene entre sus ventajas principales que no es necesario especificar por anticipado el número de grupos. Existen dos tipos de *clustering* jerárquico según la dirección en la que se ejecute el agrupamiento: divisible y aglomerativo. En el primero, partimos de un *cluster* que contiene todos los datos y vamos dividiendo en *clusters* más pequeños. En el segundo, cada punto se encuentra en un *cluster* separado, luego los *clusters* más cercanos se fusionan de forma sucesiva, quedando al final uno solo que recoge a todos los elementos. Este último enfoque es el que se abordará en este libro.

Pero antes es preciso mencionar que además de la dirección también se debe definir cómo se medirá la distancia entre los grupos, para ello tenemos las siguientes medidas:

- Conexión completa: La distancia se mide teniendo en cuenta los puntos más lejanos de cada cluster.
- Conexión simple: En oposición a la anterior, la distancia entre clústeres se calcula tomando la distancia mínima entre dos puntos de cada cluster.
- Distancia entre medias: La distancia entre dos clústeres se calcula a partir de la distancia entre las medias de cada uno.
- La distancia promedio entre pares: Es el promedio de distancias que podemos obtener entre todos los pares de puntos.

Clustering aglomerativo:

Este algoritmo inicia declarando cada punto en su propio cluster, luego fusiona los dos grupos más similares y esto se va repitiendo recursivamente hasta que se cumpla algún criterio de parada, normalmente un número de *clusters* prefijado. En Scikit-learn se manejan dos parámetros: *n_clusters*, con el cual especificamos la cantidad de grupos a formar, así como la cantidad de centroides y *linkage* (vinculación) para determinar cual estrategia de distancia usar para fusionar los conjuntos de observaciones.

El siguiente código presenta una implementación de *clustering* aglomerativo:

```
from sklearn.cluster import AgglomerativeClustering
agl = AgglomerativeClustering(n_clusters = 4, linkage = 'complete')
agl.fit(X)
```

Una vez realizado el ajuste con los datos de entrenamiento procedemos a crear un dendograma (representación gráfica en forma de árbol) con la librería Scipy para lo cual se necesita hallar primero la matriz de distancias que contiene la distancia entre cada uno de los puntos del conjunto de datos. Esto se hará empleando la función `distance_matrix()`, así:

```
from scipy.spatial import distance_matrix
matriz_distancias = distance_matrix(X,X)
print(matriz_distancias)
```

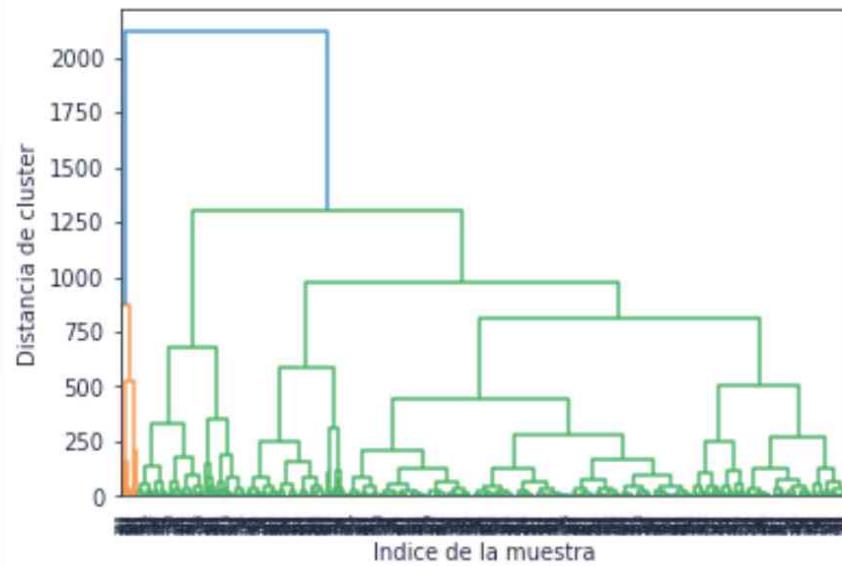
En este caso tendremos una matriz de 569x569 con la diagonal principal en ceros.

```
[[ 0.          10.42431772  7.3979727 ... 14.56647178 17.49577378
  75.57557343]
 [10.42431772  0.          3.0305775 ... 24.91828445  7.2000625
  85.94007505]
 [ 7.3979727   3.0305775   0.          ... 21.91889824 10.14091219
  82.94245776]
 ...
 [14.56647178 24.91828445 21.91889824 ...  0.          32.05058502
 61.02368393]
 [17.49577378  7.2000625 10.14091219 ... 32.05058502  0.
 93.06996293]
 [75.57557343 85.94007505 82.94245776 ... 61.02368393 93.06996293
 0.        ]]
```

```
from scipy.cluster import hierarchy
Z = hierarchy.linkage(matriz_distancias,'complete')
dendro = hierarchy.dendrogram(Z)
plt.xlabel("Indice de la muestra")
plt.ylabel("Distancia de cluster")
```

El dendograma muestra los puntos de datos en el eje de las x, que un principio corresponden a igual número de *clusters*, estas serían las hojas del árbol. Luego por cada fusión de dos *cluster* se va creando un nodo padre.

Desafortunadamente, el *clustering* aglomerativo falla cuando los datos describen formas complejas como el dataset de media luna, así que a continuación estudiaremos el algoritmo DBSCAN que ayuda a remediar esta situación.



9.3 Dbscan (Density Based Spatial Clustering of Applications with Noise)

El DBSCAN (agrupamiento espacial basado en densidad de aplicaciones con ruido) es un poderoso algoritmo que puede resolver problemas en los cuales k-means suele fallar, funciona muy bien con formas complejas y que no necesariamente tienen que ser esféricas como sucede con el kmeans, además, hace una identificación de los atípicos, a diferencia del k medias que asigna todos los puntos a un *cluster* aún cuando ellos no guarden relación con alguno. En contraste el *clustering* basado en densidad localiza regiones de alta densidad que son separadas por regiones poco densas o relativamente vacías. Densidad de un punto es este contexto, tiene que ver con el número de puntos, incluido el que se encuentra dentro de un radio específico (*eps*). Dependiendo de este número cada punto es clasificado como central (*core*), frontera (*border*) o ruido (*noise*).

El algoritmo DBSCAN trabaja con dos parámetros:

- R (radio de la vecindad) o *eps*: Define un radio específico que si incluye suficientes puntos se puede llamar área densa. Entre más pequeño sea este valor se crearán mas *clusters* y mientras más grande se podrán adicionar más puntos.

- MinPts: Mínimo número de puntos de datos.

Punto central: Un punto es central si el número de puntos a una distancia no mayor a *eps* de él, supera el valor mínimo *minpts*.

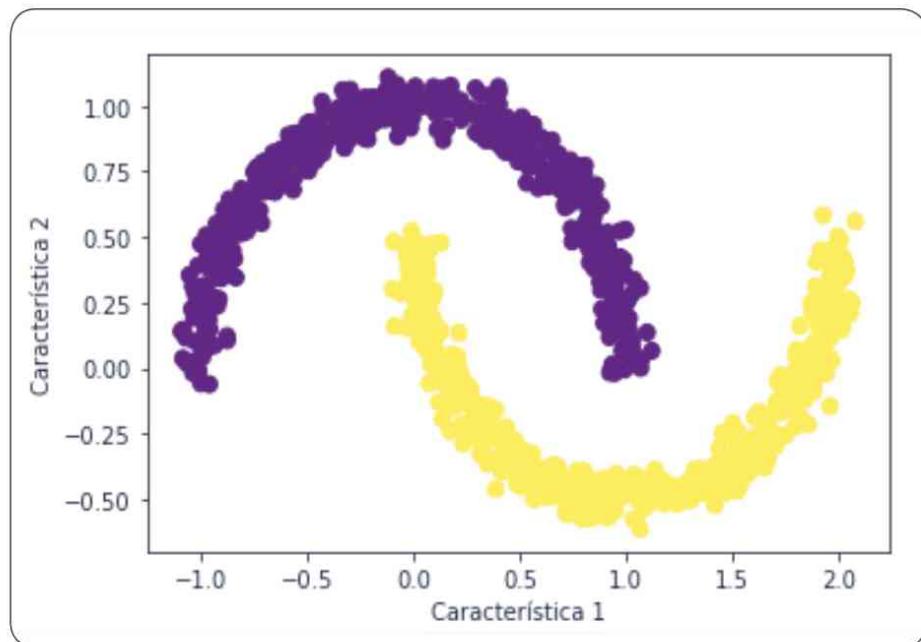
Punto fronterizo: Un punto es fronterizo si no es central, pero hay al menos un punto central a una distancia no mayor a *eps*.

Punto ruidoso: Un punto es ruido o outlier si no es central ni frontera.

Básicamente lo que hace el algoritmo es visitar cada punto, encontrar su tipo y luego agruparlos en *clústers* en función de ese tipo. El algoritmo inicia escogiendo un punto al azar, entonces busca puntos con distancia menor o igual a *eps* desde ese punto. Si hay menos de *min_samples* puntos dentro de la distancia *eps* del punto de partida, este punto se etiqueta como ruidoso, lo que significa que no pertenece a ningún *clúster*. En cambio, si hay más de *min_samples* puntos dentro una distancia de *eps*, el punto se etiqueta como muestra central y se le asigna una nueva etiqueta de grupo, luego, se visitan todos los vecinos (dentro de *eps*) del punto. Si no han sido asignado a un ningún *clúster*, se les asigna la nueva etiqueta de *clúster* que se acaba de crear. Si hay muestras centrales cercanas a alguna otra (dentro de *eps*) son colocadas dentro del mismo cluster. Esto va creciendo hasta que no hay más muestras centrales dentro de la distancia *eps* del *clúster*. Luego se elige otro punto que aún no se ha visitado, y se realiza el mismo procedimiento recursivamente.

Para probar el algoritmo DBSCAN vamos a crear un dataset de media luna con 1.000 puntos (500 para cada media luna) y vamos a intentar demostrar que el K-means falla al momento de capturar los grupos en este tipo de problemas. El código que crea y dibuja los puntos en un plano bidimensional es como sigue:

```
from sklearn.datasets import make_moons
num_muestras = 1000
X, Y = make_moons(n_samples=num_muestras, noise=0.05)
plt.scatter(X[:, 0], X[:, 1], c=Y, cmap='viridis', s=50)
plt.xlabel("Característica 1")
plt.ylabel("Característica 2")
```

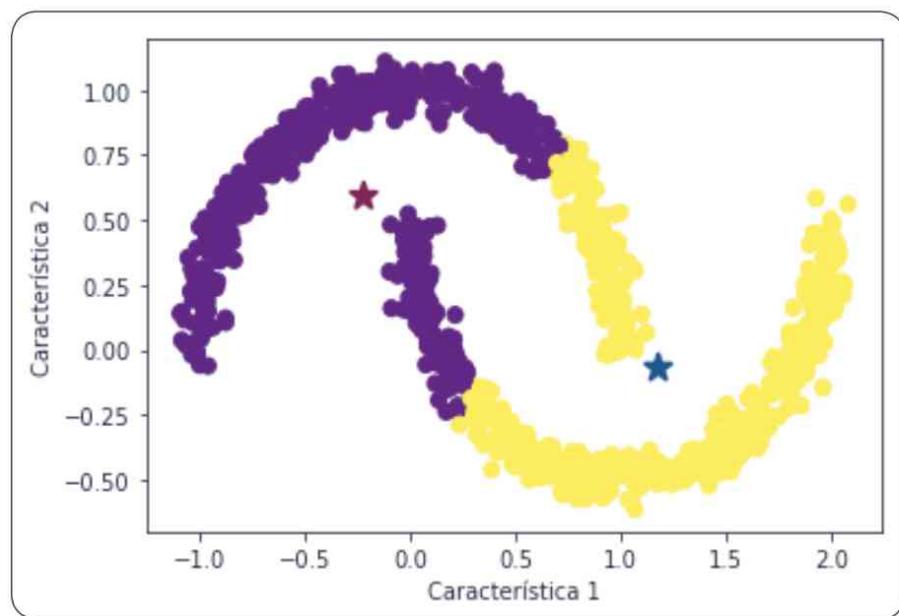


Vamos a crear un objeto kmeans con 2 cluster para tratar de agrupar estos datos:

```
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
y_pred = kmeans.predict(X)

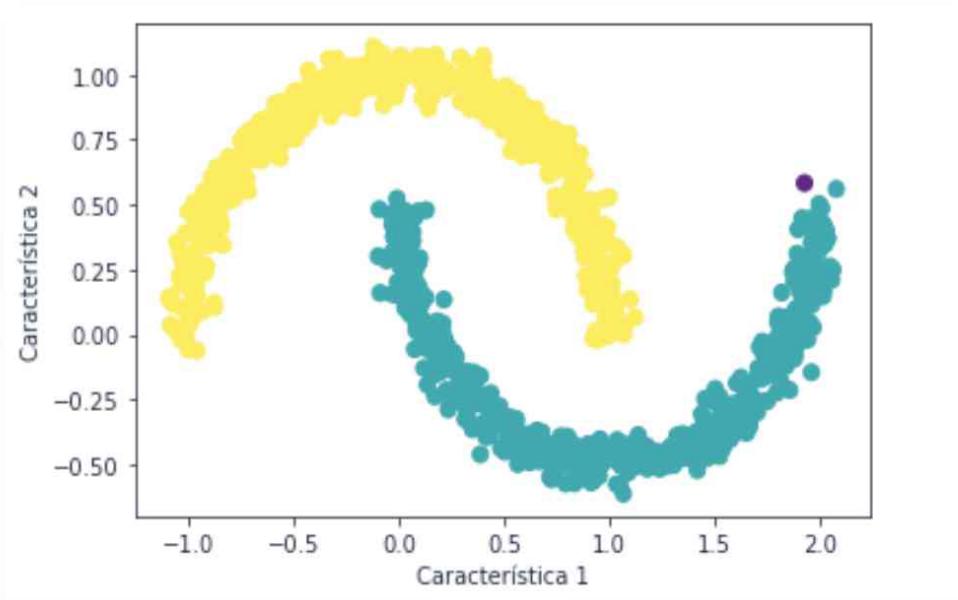
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='viridis', s=50)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
           marker='*', c=np.unique(y_pred), cmap='RdBu', s=150, linewidth=2)
plt.xlabel("Característica 1")
```

La gráfica nos muestra que el algoritmo no ha sido capaz de agrupar adecuadamente las puntas de luna en un cluster aparte.



Ahora intentemos crear un objeto DBSCAN con un valor para `eps` de 0.1 y notemos cómo este, si agrupa las medias lunas, además ha identificado los datos atípicos (ruido), los cuales podrá apreciar de un color diferente al resto de puntos si ejecuta el código en un notebook.

```
from sklearn.cluster import DBSCAN
dbs = DBSCAN(eps=0.1)
Y = dbs.fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=Y, cmap='viridis', s=50)
plt.xlabel("Característica 1")
plt.ylabel("Característica 2")
```



<https://dogramcode.com/bloglibros>



CAPÍTULO 10

Reducción de la dimensionalidad

Temas

- 10.1 Análisis de componentes principales (PCA)
- 10.2 Análisis discriminante lineal (ADL)

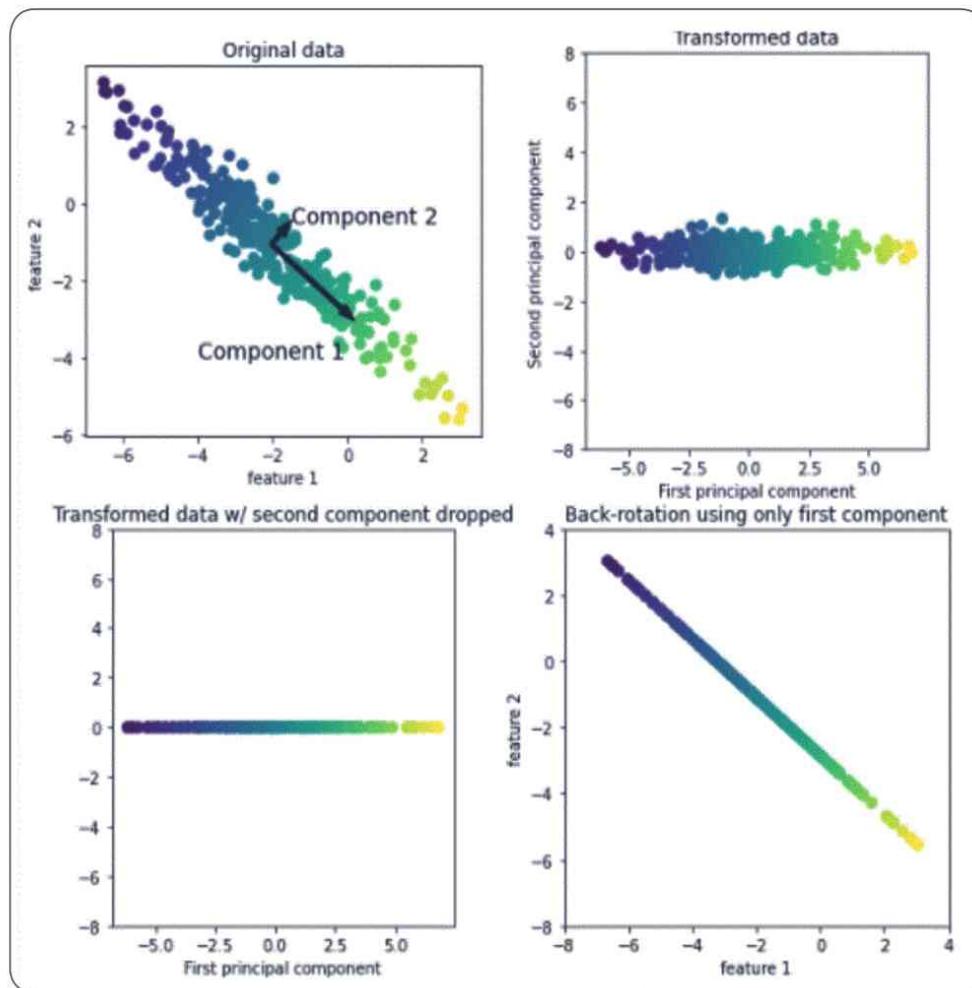
La reducción de la dimensionalidad es una técnica muy importante en *machine learning* y consiste básicamente, como su nombre lo indica, en disminuir el número de características de un conjunto de datos con el fin de mejorar u optimizar el rendimiento de los algoritmos de aprendizaje automático, permitiendo con esto también que puedan ejecutarse de manera más rápida. El objetivo es combatir el problema de la maldición de la dimensionalidad transformando los datos en un nuevo conjunto de dimensiones partiendo de las características iniciales. Cada dimensión creada tiene codificada un subconjunto de las características originales que guardan una alta correlación entre sí. Por ejemplo, suponga un conjunto de datos con las siguientes características descriptivas *nombre_libro*, *precio_libro*, *categoria_libro*, *nombre_autor*, *reputación_autor* y como variable objetivo *existencias_libro*. Se podría crear otro dataset con solo dos características c1 para *nombre_libro*, *precio_libro*, *categoria_libro* y c2 para *nombre_autor*, *reputación_autor*.

En este capítulo exploraremos dos algoritmos de reducción de la dimensionalidad: PCA (Análisis de componentes principales) y ADL (Análisis del discriminante lineal).

10.1 Análisis de componentes principales (PCA)

El análisis de componentes principales es un método de aprendizaje no supervisado que opera sobre variables correlacionadas, realizando una rotación del conjunto de datos de tal manera que las características rotadas son estadísticamente no correlacionadas linealmente. A esta rotación frecuentemente le sigue la selección de un subconjunto de características de acuerdo con su importancia.

Considere la siguiente figura donde se muestra cómo es el proceso de transformación mencionado en el párrafo anterior.



Las siguientes ilustraciones muestran como de una data de dos dimensiones se realiza una proyección en 1 dimensión preservando la mayor cantidad de variabilidad de los datos. La imagen superior izquierda muestra el conjunto de datos original, en ella se evidencia como el algoritmo PCA comienza buscando la dirección de máxima varianza o la dirección en la que se encuentre la mayoría de la información y donde hay mayor correlación entre las características, este es

el Component 1, luego el algoritmo busca otro vector que contenga el resto de la información en sentido ortogonal al anterior, es decir, que esté ubicado de tal manera que forme un ángulo recto, siendo este el component 2. Estos dos vectores son los llamados componentes principales que representan las principales direcciones de variación de los datos. En general, hay tantos componentes principales como características originales existan en el conjunto de datos.

La segunda imagen muestra los mismos datos, pero rotados con el primer componente principal alineado con el eje x y el segundo componente principal sobre el eje y. Adicionalmente a los datos se les resta la media centrándose en torno a 0. (En este punto no hay correlación entre los ejes).

En la tercera imagen es evidente la reducción de la dimensionalidad al desaparecer uno de los componentes, en este caso ha quedado solo el primer componente principal. Pasando el dataset de ser bidimensional a ser de una dimensión.

En la última imagen se observa que se deshace la rotación y se devuelve la media a los datos. Los puntos están ahora en el espacio de características original, pero manteniendo solo la información contenida en el primer componente principal.

Normalmente el algoritmo PCA para hacer la transformación en los datos y lograr reducir la dimensionalidad de los mismos, presenta el siguiente algoritmo mostrado a continuación. Tenga en cuenta que el dataset a usar es el Breast Cancer y a cada paso descrito se le colocará el código Python que lo implementa:

1. Centrar los datos restando la media de cada variable. Note que X_escalado representa las variables explicativas debidamente escaladas con la clase StandardScaler.

```
X_media = np.mean(X_escalado, axis=0)
X = X_escalado - X_media
```

2. Construir una matriz de covarianza. Esta matriz almacena la varianza y la covarianza de cada par de características. La varianza es una medida de dispersión que mide la variabilidad de un conjunto de datos con respecto a su media y la covarianza mide el grado de correlación entre cada par de variables. La varianza y la covarianza manejan las siguientes ecuaciones respectivamente:

$$\sigma_{xx}^2 = \frac{\sum_{i=1}^n (x_i - \bar{x})(x_i - \bar{x})}{n-1}$$

$$\sigma_{xy}^2 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{n-1}$$

Una matriz de covarianza es una matriz cuadrada que muestra en su diagonal principal la varianza de cada característica y en las demás posiciones las covarianzas entre todos los posibles pares de variables del conjunto de datos. A modo de ejemplo para tres características x_1, x_2, x_3 , y tendría la siguiente estructura:

$$C = \begin{bmatrix} cov(x_1, x_1) & cov(x_1, x_2) & cov(x_1, x_3) \\ cov(x_2, x_1) & cov(x_2, x_2) & cov(x_2, x_3) \\ cov(x_3, x_1) & cov(x_3, x_2) & cov(x_3, x_3) \end{bmatrix}$$

Con la función cov() de Numpy se obtiene la matriz de covarianza:

```
matriz_cov = np.cov(X, rowvar=False)
```

3. Luego el algoritmo PCA descompone la matriz de covarianza en sus vectores propios y valores propios. Los primeros corresponden a los componentes principales, es decir, tienen que ver con la dirección donde hay mayor varianza o información en los datos. Y los segundos son las magnitudes de los vectores propios. En este punto, se debe satisfacer la siguiente ecuación:

$$Cov * \vec{V} = \lambda * \vec{V}$$

Numpy permite obtener los vectores y valores propios de una matriz de covarianza por medio de la función eig() del submodule linalg. Estos son devueltos en forma de arreglos unidimensionales y cada uno tiene un tamaño igual al número de características del dataset.

```
valores_propios, vectores_propios = np.linalg.eig(matriz_cov)
vectores_propios = vectores_propios.T
```

4. Los componentes principales se seleccionan de los vectores propios de la matriz de covarianza, por tanto, para identificar aquellos de mayor varianza se procede a ordenar de mayor a menor sus respectivos valores propios. Los k vectores propios seleccionados representarán la dimensionalidad del nuevo espacio de características.

```
componentes_ordenados = np.argsort(valores_propios)[::-1]
```

5. Consecuentemente se construye una matriz de proyección donde cada columna de la matriz es un vector propio correspondiente a un componente principal

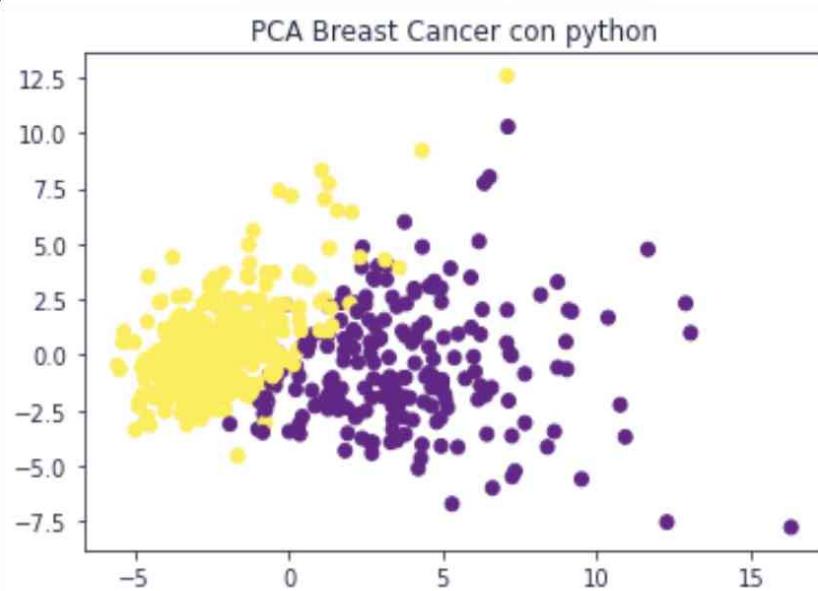
```
matriz_proyeccion = vectores_propios[componentes_ordenados[:2]]
```

6. Finalmente se hace la transformación del conjunto de datos original de una dimensión mayor al nuevo espacio de características de menor dimensión.

```
X_t = np.dot(X, matriz_proyeccion.T)
```

Aquí, X_t es una matriz de igual cantidad de filas que el dataset original pero ahora tiene solo 2 columnas. Podemos crear ahora un gráfico de dispersión con estos datos:

```
import matplotlib.pyplot as plt
plt.figure()
plt.scatter(X_t[:, 0], X_t[:, 1], c=df['y'].values)
```



En Scikit-learn podemos implementar este mismo algoritmo sobre el mismo dataset Breast Cancer mediante la clase PCA localizada dentro del paquete sklearn.decomposition. Debemos crear una instancia de la clase PCA con dos componentes principales (n_components=2), luego con el método fit() encontramos los componentes principales y con transform() aplicamos la transformación y la reducción de la dimensionalidad.

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(X_escalado)
X_pca = pca.transform(X_escalado)
print("Forma Original: {}".format(str(X_escalado.shape)))
print("Forma Reducida: {}".format(str(X_pca.shape)))
```

Forma Original: (569, 30)

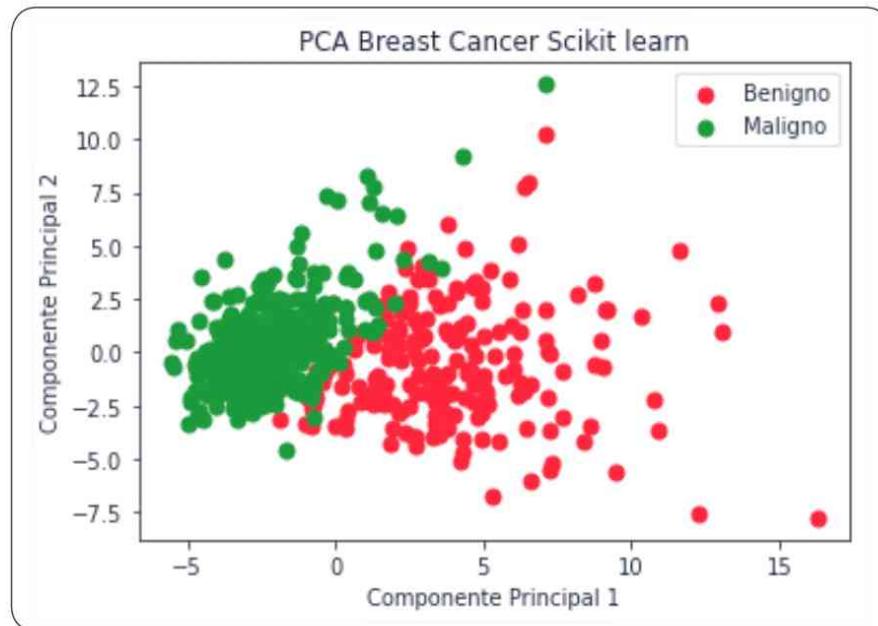
Forma Reducida: (569, 2)

Vemos como se ha reducido a dos el número de características de igual manera que con nuestra implementación en Python. Estos datos transformados que aparecen en X_pca se podrían pasar por ejemplo a un algoritmo de clasificación como veremos en el siguiente subapartado y lograr una mejora significativa en la eficiencia de la predicción, así como en la rapidez en el proceso de entrenamiento. A continuación, creamos un dataframe con los datos transformados.

```
df_pca = pd.DataFrame(data = X_pca
, columns = ['CP1','CP2'])
```

Finalmente dibujamos un gráfico de dispersión usando nuestros dos componentes principales:

```
plt.xlabel('Componente Principal 1')
plt.ylabel('Componente Principal 2')
plt.title("PCA Breast Cancer Scikit learn")
colores = ['r', 'g']
destino = [0, 1]
for vd, cl in zip(destino,colores):
    indice = df['y'] == vd
    plt.scatter(df_pca.loc[indice, 'CP1']
                , df_pca.loc[indice, 'CP2'], c = cl, s = 50)
plt.legend(['Benigno', 'Maligno'])
```



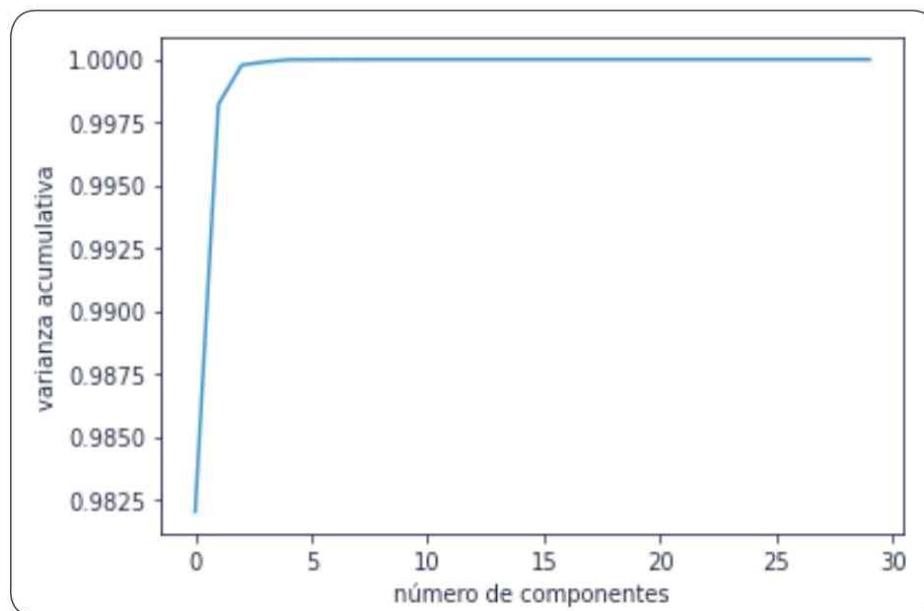
Como cada componente codifica una combinación de variables en muchos casos la interpretación en este tipo de diagramas de dispersión no resulta del todo fácil, sin embargo, acerca de nuestro gráfico podemos afirmar que un clasificador lineal puede sin problemas separar los puntos de datos. Estos componentes principales se almacenan en el atributo *components_* en este caso una matriz de 2 filas y 30 columnas, donde cada fila en orden de importancia corresponde a un componente principal y cada columna corresponde a un atributo original del PCA.

```
print("Componentes PCA:\n{}".format(pca.components_))
```

```
Componentes PCA:
[[ 0.21890244  0.10372458  0.22753729  0.22099499  0.14258969  0.23928535
  0.25840048  0.26085376  0.13816696  0.06436335  0.20597878  0.01742803
  0.21132592  0.20286964  0.01453145  0.17039345  0.15358979  0.1834174
  0.04249842  0.10256832  0.22799663  0.10446933  0.23663968  0.22487053
  0.12795256  0.21009588  0.22876753  0.25088597  0.12290456  0.13178394]
[-0.23385713 -0.05970609 -0.21518136 -0.23107671  0.18611302  0.15189161
  0.06016536 -0.0347675  0.19034877  0.36657547 -0.10555215  0.08997968
 -0.08945723 -0.15229263  0.20443045  0.2327159  0.19720728  0.13032156
  0.183848   0.28009203 -0.21986638 -0.0454673 -0.19987843 -0.21935186
  0.17230435  0.14359317  0.09796411 -0.00825724  0.14188335  0.27533947]]
```

Por otro lado, es posible acceder a la proporción de varianza explicativa a través de la variable de instancia *explained_variance_ratio_*, esta muestra que parte de la varianza es llevada por cada componente individual.

```
pca = PCA().fit(cancer.data)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('número de componentes')
plt.ylabel('varianza acumulativa')
```



La curva cuantifica cuánto de la varianza total está contenida en los primeros componentes. Observamos que, en este caso, los dos primeros componentes contienen alrededor del 99% de la varianza, y se necesitan alrededor de tres componentes para describir cerca del 100% de la varianza.

A continuación, proponemos un ejemplo sobre el algoritmo del análisis de componentes principales utilizando el dataset de dígitos manuscritos.

Clasificador dígitos escritos a mano con SVM usando PCA

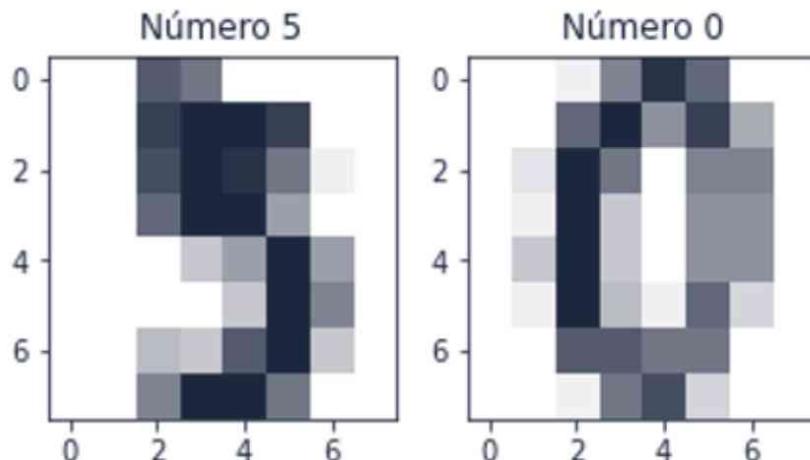
En este nuevo ejemplo trataremos de crear un clasificador de dígitos manuscritos usando el algoritmo de máquinas de vectores de soporte aplicando una reducción en la dimensionalidad de los datos con PCA. El dataset a usar lo vamos a tomar del paquete de datasets de Scikit-learn y se llama Digit. Este dataset tiene información de 1.797 imágenes de números en escala de grises de 8x8 pixeles, las cuales están codificadas numéricamente en el rango entre 0 y 255. Cada imagen corresponde con una fila del dataset y tiene 64 características (pixeles) más una etiqueta de clase con valores entre 0 y 9.

Iniciamos cargando el conjunto de datos Digit:

```
from sklearn.datasets import load_digits
digitos = load_digits()
X, y = digitos.data, digitos.target
```

Mostremos las instancias 5 y 10 del dataset usando la función imshow() que corresponden con los dígitos 5 y 0:

```
import matplotlib.pyplot as plt
plt.figure(figsize=[5,5])
plt.subplot(121)
plt.imshow(digitos.images[5], plt.cm.gray_r)
plt.title("Número " + str(digitos.target[5]) )
plt.subplot(122)
plt.imshow(digitos.images[10], plt.cm.gray_r)
plt.title("Número " + str(digitos.target[10]) )
```



Entrenemos una máquina de vectores de soporte con el conjunto de datos original y calculemos el puntaje con los datos de prueba:

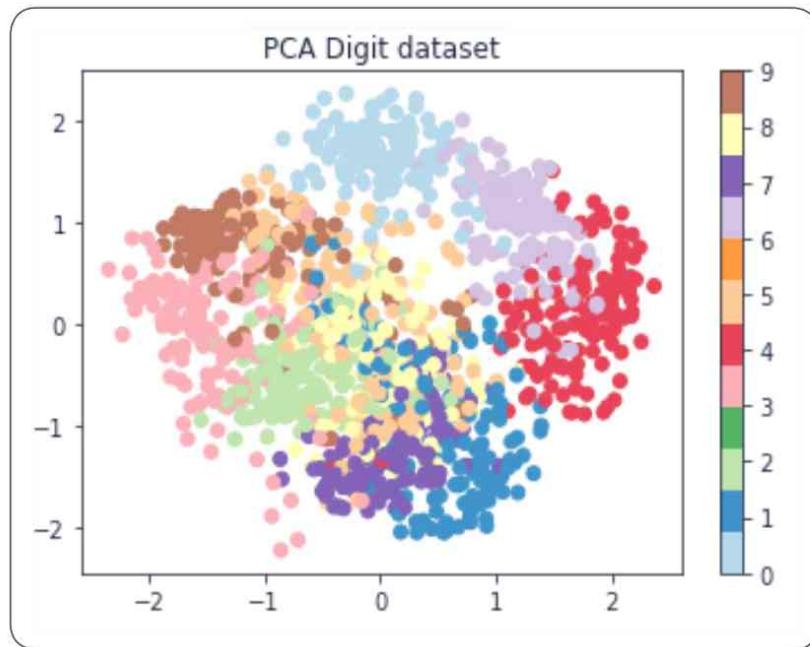
```
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
X_ent, X_pru,y_ent, y_pru = train_test_split(X,y, test_size = 0.2, random_
state=43)
X_ent = X_ent / 255
X_pru = X_pru / 255
clf = SVC(C=1 , gamma=1)
clf.fit(X_ent, y_ent)
print("Exactitud conjunto de entrenamiento {:.2f}".format(clf.score(X_ent, y_
ent)))
y_pred = clf.predict(X_pru)
print("Exactitud conjunto de prueba {:.2f}".format(accuracy_score(y_pru, y_
pred)))
```

Exactitud conjunto de entrenamiento 0.94

Exactitud conjunto de prueba 0.93

Tratemos ahora de mejorar este resultado implementando reducción de dimensionalidad a través de PCA con 10 componentes principales:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=10 , whiten=True)
X_pca = pca.fit_transform(X_ent)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y_ent, cmap="Paired")
plt.title("PCA Digit dataset")
plt.colorbar()
```



Acto seguido, entrenemos nuevamente un clasificador de máquina de soporte vectorial, pero esta vez pasándole los datos transformados por PCA, lo cual mejorará la salida del modelo, pero incluyendo un poco de sobreajuste como veremos a continuación:

```
from sklearn.metrics import accuracy_score
X_entr, X_prue, y_entr, y_prue = train_test_split(X_pca, y_ent, test_size=0.2,
random_state=1)
clf = SVC(C=1, gamma=1)
clf.fit(X_entr, y_entr)
print("Exactitud conjunto de entrenamiento {:.2f}".format(clf.score(X_entr, y_entr)))
y_pred = clf.predict(X_prue)
print("Exactitud conjunto de prueba {:.2f}".format(accuracy_score(y_prue, y_pred)))
```

Exactitud conjunto de entrenamiento 1.00

Exactitud conjunto de prueba 0.95

<https://dogramcode.com/bloglibros>



10.2 Análisis discriminante lineal (ADL)

El análisis discriminante lineal es otro método de extracción de características usado también para reducción de dimensionalidad y muy parecido en su forma de trabajar al algoritmo PCA, pero presenta una diferencia importante y es que al ser un algoritmo de aprendizaje supervisado requiere de las etiquetas de clase para poder operar, además trabaja mejor con datos distribuidos uniformemente.

Los pasos básicos para el análisis discriminante lineal son los siguientes:

1. Calcular los vectores medios de dimensión d para las diferentes clases del conjunto de datos.
2. Calcular las matrices de dispersión entre las clases y dentro de las clases.
3. Calcular los vectores propios y los valores propios para las matrices de dispersión.
4. Ordenar los vectores propios de mayor a menor por medio de los valores propios y seleccionar los vectores propios con los valores propios más grandes.
5. Se construye una matriz de proyección donde cada columna de la matriz es un vector propio correspondiente a un componente principal.
6. Utilizar la matriz de proyección para transformar las muestras en el nuevo subespacio.

Si el lector desea conocer más sobre el ADL y el código en python de los anteriores pasos, puede consultar este excelente tutorial de Sebastian Raschka, disponible en: https://sebastianraschka.com/Articles/2014_python_lda.html.

Sin embargo, en Scikit-learn contamos con una implementación del algoritmo ADL en la clase LinearDiscriminantAnalysis del submódulo discriminant_analysis, con la cual crearemos un objeto con cinco componentes (`n_components=5`) para transformar el dataset Digit.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
lda = LDA(n_components=5)
ldaX_ent = lda.fit_transform(X_ent, y_ent)
ldaX_pru = lda.transform(X_pru)
```

Note, que por tratarse de un algoritmo de aprendizaje supervisado el método `fit_transform()` debe recibir también el array con las etiquetas de clase (`y_ent`). Seguidamente vamos a entrenar nuestro clasificador de igual forma a como hicimos en el ejemplo de la sección anterior, pero observando una reducción en el sobreajuste.

```
X_entr, X_prue, y_entr, y_prue = train_test_split(ldaX_ent, y_ent, test_size=0.1,  
random_state=1)  
clf = SVC(C=1 , gamma=1)  
clf.fit(X_entr, y_entr)  
print("Exactitud conjunto de entrenamiento {:.2f}".format(clf.score(X_entr, y_entr)))  
y_pred = clf.predict(X_prue)  
print("Exactitud conjunto de prueba {:.2f}".format(accuracy_score(y_prue, y_pred)))
```

Exactitud conjunto de entrenamiento 0.99

Exactitud conjunto de prueba 0.90

Con este ejercicio terminamos nuestro aprendizaje sobre los algoritmos y métodos más representativos del aprendizaje automático y a partir de ahora iniciaremos una aventura en el apasionante mundo del aprendizaje profundo, una subárea de la inteligencia artificial que ha tomado mucho auge en los últimos años, generando un gran interés en el campo académico y de la industria.

<https://dogramcode.com/bloglibros>



CAPÍTULO 11

Introducción a las redes neuronales

Temas

- 11.1 Conceptos básicos sobre redes neuronales
- 11.2 Entrenamiento de una red neuronal
- 11.3 Red neuronal para clasificación binaria
- 11.4 Red neuronal para clasificación múltiple

En esta segunda parte del libro trataremos algunas de las técnicas más extendidas en el contexto del aprendizaje profundo o *deep learning*; subárea del aprendizaje automático que ha tenido mucho auge en los últimos años fundamentalmente por la evolución que ha tenido el hardware, evidenciado en el aumento en la capacidad de almacenamiento y procesamiento de dispositivos como la memoria y el procesador. El *deep learning* en definitiva comprende una serie de técnicas y métodos con los cuales se pueden abordar un gran abanico de problemas como: clasificación de imágenes, generación de textos y traducción de textos, etc. Si bien, el aprendizaje profundo no es una disciplina nueva, toda vez algunos de los primeros trabajos en este campo se dieron en la década de los ochenta, ha venido creciendo en popularidad en los últimos años con la aparición de mejores CPUs y de otros chips capaces de realizar muchas más operaciones por segundo conocidos como GPUs (Unidades de Procesamiento Gráfico), que si bien no fueron diseñados en un principio para tareas de aprendizaje profundo, sino para acelerar videojuegos, sin embargo, un inesperado descubrimiento suscitó su utilización en trabajos en este subcampo del aprendizaje de máquina, pues permiten acelerar también el rendimiento de los algoritmos predictivos.

En las siguientes secciones profundizaremos en los diferentes conceptos relacionados con el *deep learning*: neuronas, optimizadores, funciones de activación, etc y posteriormente abordaremos varios tipos de redes neuronales: clásicas, convolucionales y recurrentes.

Los anteriores temas en su mayoría serán tratados primeramente desde un punto de vista teórico y luego de manera práctica por medio de ejemplos de programación. Tanto así que, son requeridas algunas nociones matemáticas y dominio del lenguaje Python.

En el capítulo siguiente comenzamos exponiendo algunos fundamentos relacionados con las redes neuronales, se trata de conceptos importantes porque ayudarán a desentrañar de una mejor manera temas un poco más complejos propios de este apasionante mundo del aprendizaje profundo.

11.1 Conceptos básicos sobre redes neuronales

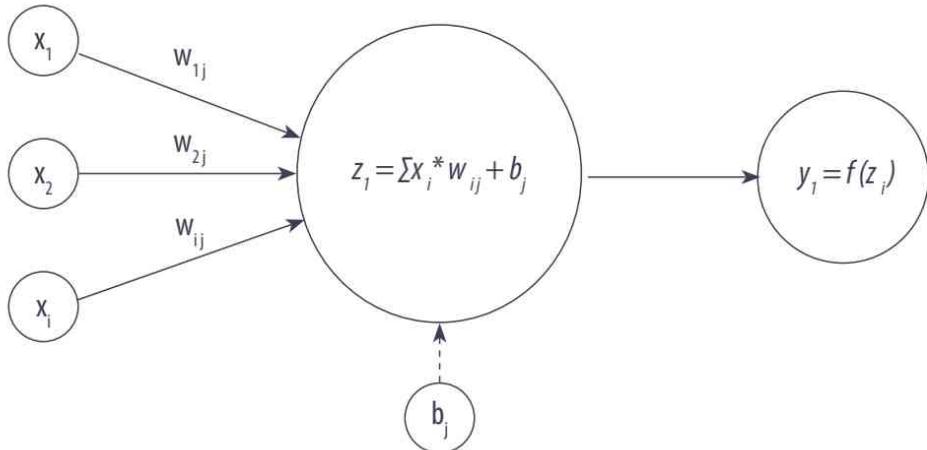
Tal y como se mencionó en la introducción del capítulo, en esta sección iniciaremos nuestro estudio sobre *deep learning* con una conceptualización de los principales elementos teóricos que hacen parte de este campo, y que sin duda marcarán la pauta para una mejor asimilación de los subsiguientes temas.

11.1.1. Neurona artificial

Presentan una estructura y funcionamiento inspirado en la neurona biológica. Las neuronas artificiales al igual a como sucede a nivel biológico reciben un estímulo del exterior, realizan un procesamiento sobre la señal recibida para luego emitir una salida que transmiten al exterior o a otra neurona. Los elementos que constituyen una neurona artificial son:

- x_i representa la i-esima entrada
- w_{ij} , representa unos parámetros llamados pesos que controlan el nivel de inhibición o excitación de las neuronas.
- b_j representa otros parámetros llamados sesgos (en inglés, *bias*)
- z , es la suma de los estímulos recibidos por la neurona
- $f(z)$ es la función de activación o transferencia.
- y_i corresponde con la salida generada por la neurona

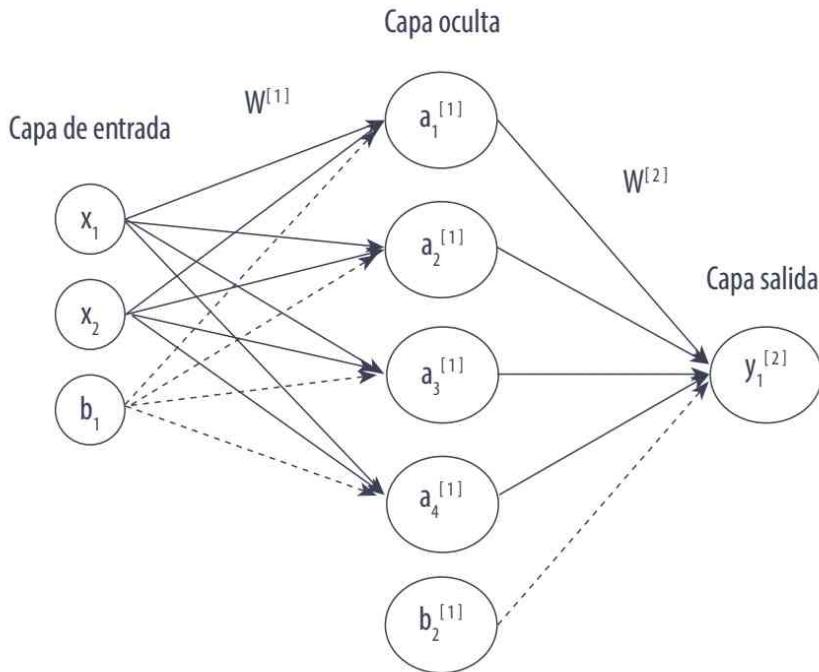
En la siguiente figura se aprecian todos los elementos constituyentes de una neurona artificial:



11.1.2. Red neuronal

Es un modelo computacional basado, de una manera aproximada, en el comportamiento biológico de nuestras neuronas cerebrales. Una red neuronal está conformada por un conjunto de unidades o nodos, que simulan a las neuronas biológicas y normalmente son agrupadas en capas. La capa de entrada corresponde a las características de entrada (x) y cada nodo es una de estas características. La capa de salida representa a la variable de destino, si hablamos de clasificación binaria se maneja un solo nodo, el cual representa la probabilidad de que la predicción pertenezca a una de las dos posibles clases; de igual manera ocurre si es un caso de regresión donde dicho nodo representa un único valor continuo; ahora bien, si estamos ante un clasificación multiclaase la capa de salida tendrá n nodos, donde n representa el número de clases posibles y el valor de cada nodo es la probabilidad de predicción de la clase asociada.

En el campo de las redes neuronales, uno de los primeros avances realizados fue el perceptrón de Frank Rosenblatt, estudiado en el capítulo 7 y que como aprendimos funciona como clasificador de datos separables linealmente. Este perceptrón simple, dispone de una arquitectura basada en una sola neurona de procesamiento y una única salida, pero sí admite un número arbitrario de entradas. Sin embargo, puede utilizarse junto con otros perceptrones o neuronas artificiales para conformar una red neuronal más compleja útil para abordar problemas más avanzados con datos que no son linealmente separables. De allí surge el concepto de perceptrón multicapa, que es un tipo de red neuronal compuesto por un conjunto de neuronas conectadas todas entre sí. La siguiente imagen representa un modelo de este tipo, con 1 capa oculta con 4 neuronas y 1 capa de salida con una neurona. Note que el superíndice en los nodos se utiliza para indicar la capa a la que pertenecen las neuronas. Además, observe la existencia de un parámetro de sesgo para cada capa de entrada y oculta.



No obstante, puede haber redes neuronales con varias capas ocultas en cuyo caso se considera red neuronal profunda. Agregar más capas a una red permite que la red pueda capturar patrones más complejos y aprender más acerca de las relaciones entre los datos, eso sí, a expensas de un incremento en los parámetros del modelo y por ende de una mayor exigencia computacional, situación que hoy en día es solventada sin duda alguna por los equipos de computo con los que contamos en la actualidad, que hacen posible una ejecución más rápida de los algoritmos y técnicas especializadas del aprendizaje profundo.

11.1.3. Pesos

Representan las conexiones entre las neuronas y tienen la capacidad de amplificar o atenuar las señales de las mismas. Los pesos son elementos internos de la red neuronal e influyen en sus resultados, por lo tanto, durante el entrenamiento se deben encontrar sus valores óptimos para que la red pueda tener un desempeño adecuado. En la imagen $W^{[1]}$ representa los pesos que conectan la capa de entrada con la capa de oculta y $W^{[2]}$ son los que conectan la capa oculta con la capa de salida.

11.1.4. Sesgos (Bias)

Los sesgos adicionan una señal extra a la función de activación e igual que las entradas tienen un peso asociado. Normalmente se agrega un término de *bias* para la capa de entrada y uno por cada capa oculta con el fin de mitigar la linealidad provocada por entradas iguales a 0.

Con respecto a la red anterior tenemos que:

- En cuanto a los pesos hay $2 \times 4 = 8$ (primera capa) y $4 \times 1 = 4$ (segunda capa) para un total de 12 pesos. (Líneas negras)
- Con respecto a los sesgos 4 (primera capa) + 1 (segunda capa) = 5 . (Líneas punteadas rojas). Lo anterior provee un total de 17 parámetros entrenables.

Estos componentes se expresan en forma de arreglos, así: \vec{W} matriz de pesos, \vec{b} vector con los sesgos y \vec{a} vector con los datos de la activación. Para la red anterior tendríamos los siguientes arreglos:

Capa de entrada – Capa oculta:

$$W_{ij}^{[1]} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & W_{14} \\ W_{21} & W_{22} & W_{23} & W_{24} \end{bmatrix}, \quad b_i^{[1]} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}, \quad a_i^{[1]} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix}$$

Capa oculta – Capa de salida:

$$W_{ij}^{[2]} = \begin{bmatrix} W_{11} \\ W_{21} \\ W_{31} \\ W_{41} \end{bmatrix}, \quad b_i^{[2]} = [b_2], \quad a_i^{[2]} = [y_1]$$

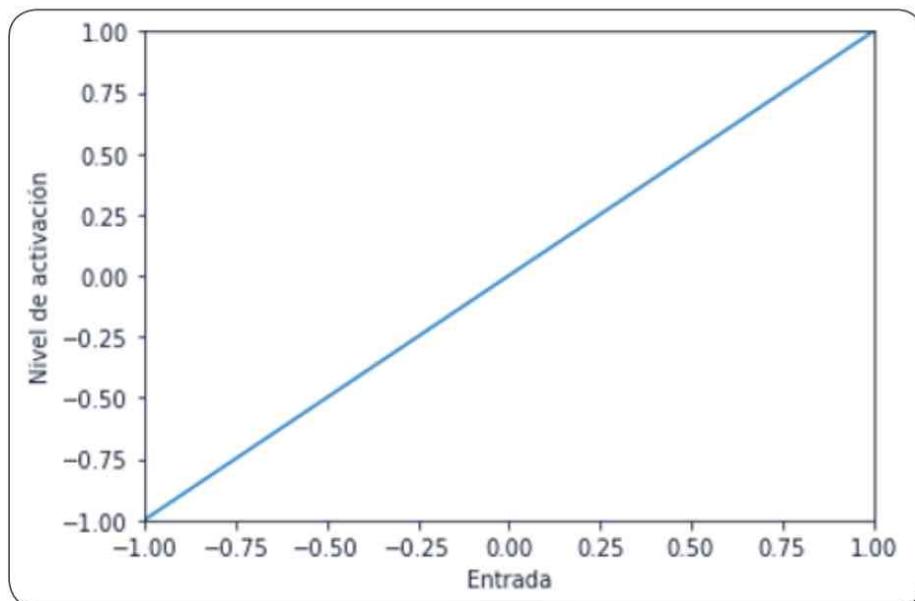
Note que en la representación anterior se han suprimido los superíndices de los valores dentro de los corchetes para facilitar la interpretación. Por otro lado, en el subapartado de *forward propagation* miraremos las matemáticas aplicables sobre estos arreglos en el proceso de propagación de información entre las capas de una red neuronal.

11.1.5. Funciones de activación

La salida de una neurona se da gracias a las funciones de activación, las cuales pueden agregar procesamiento no lineal a las redes neuronales. Como se ha comentado, en las redes neuronales artificiales las entradas son combinadas con los pesos, a esta suma ponderada o entrada neta, por medio de la cual la neurona recoge el grado de estímulo del exterior se le aplica una función de activación que determina la salida de la neurona.

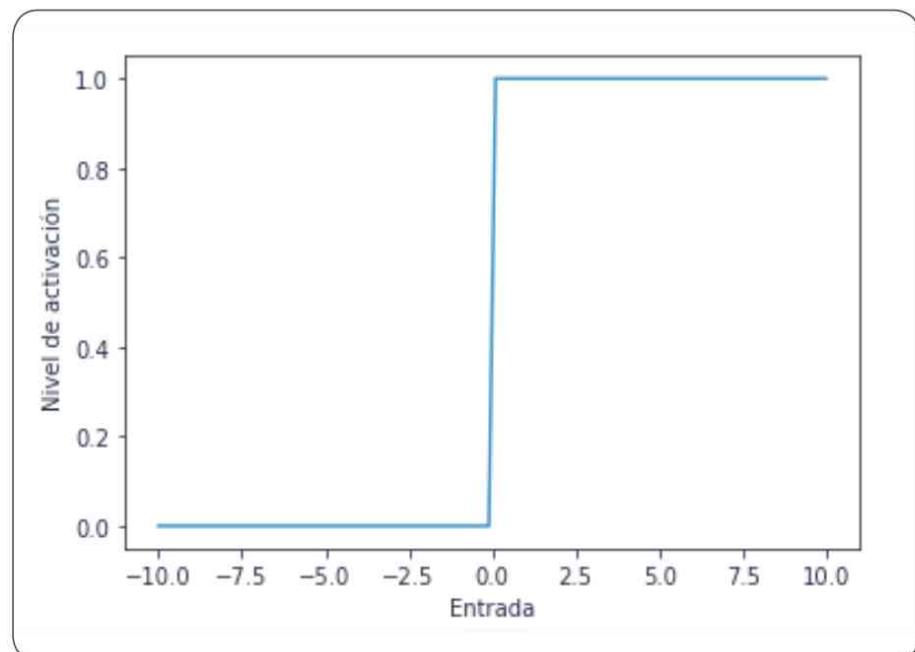
Las funciones de activación más usadas son: lineal, escalón, sigmoide, tangente hiperbólica y unidad rectificadora lineal (RELU).

- Lineal: Es la función más elemental y se encuentra definida por la función identidad. ($\Phi(z) = z$).



- Escalón: Ya la estudiamos en el capítulo dedicado a clasificación y es una función de activación no lineal y devuelve una salida binaria.

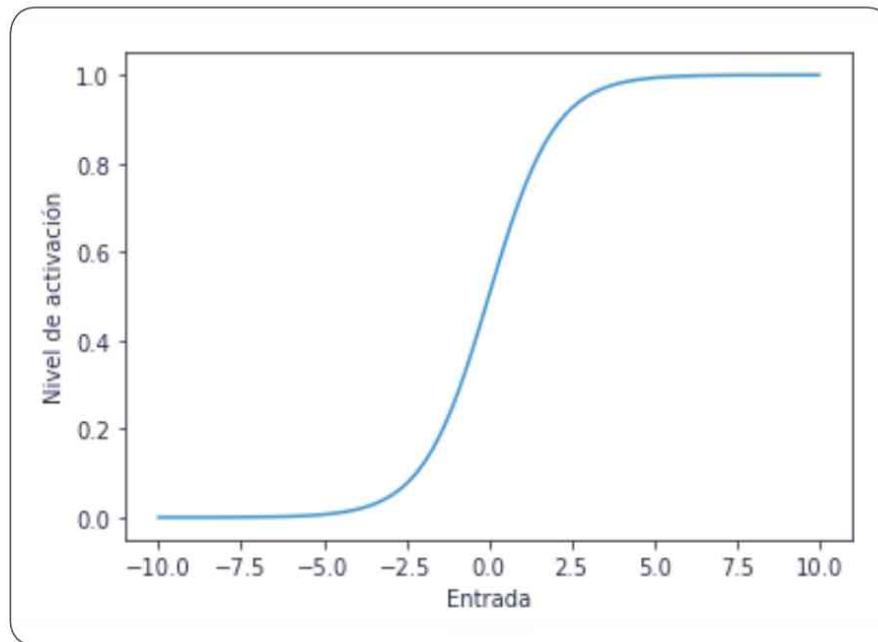
$$\Phi(z) = \begin{cases} 1, & z \geq 0 \\ -1, & z < 0 \end{cases}$$



- Sigmoide: Además de no lineal es una función continua y derivable, lo cual es fundamental si se utiliza un algoritmo de optimización como el descenso del gradiente para hallar los pesos óptimos o si también se va a entrenar la red empleando el algoritmo de propagación hacia atrás o *backpropagation*. Backpropagation es un algoritmo que inicia en la capa de salida en donde una vez se ha calculado el error –resultado de comparar el valor de salida para una entrada con el valor predicho por la red– y teniendo en cuenta que no podemos cambiar los valores de las entradas de la red, pero si modificar los valores de los pesos y los bias. El algoritmo procede buscando en las capas anteriores cuál fue la influencia que tuvieron los pesos y los bias en el error calculado, para luego ir ajustando estos parámetros con el fin de minimizar el error. Dicho ajuste requiere el uso de la derivada de la función de activación de las neuronas, de allí que sea un requisito que dicha función sea derivable.

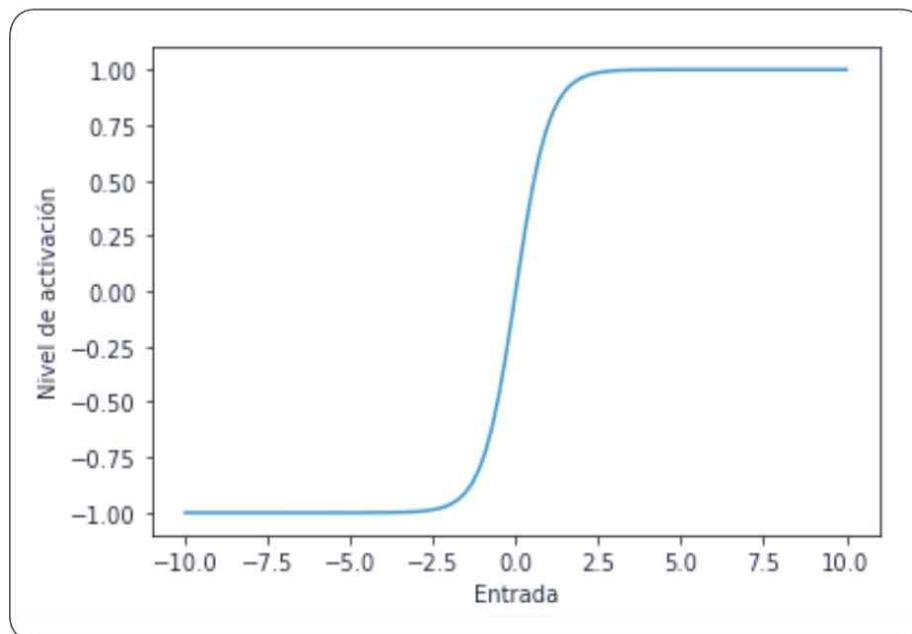
Recordemos la definición de la función sigmoide y su respectiva gráfica:

$$\Phi(z) = \frac{1}{1 + e^{-z}}$$



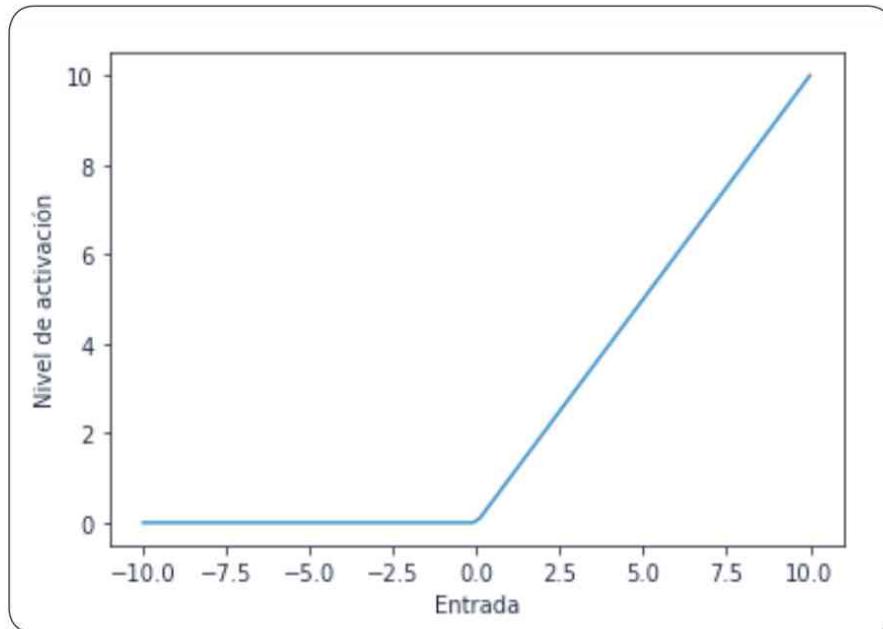
- Tangente hiperbólica (\tanh): Es una extensión de la función anterior y se suele usar mucho en las capas ocultas de las redes neuronales, por ofrecer un rango de salida más amplio (-1, 1) que ayuda mucho a la convergencia del algoritmo de propagación hacia atrás.

$$\Phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



- Unidad lineal rectificada (ReLU): Es una de las más usadas en el ámbito de las redes neuronales profundas, puesto que ayuda a evitar el problema de la desaparición del gradiente. Dificultad en la que los gradientes se encuentran muy próximos a 0, provocando la ineficiente actualización de los pesos. Esta situación causa la no convergencia o en el peor de los casos impide el entrenamiento de una red neuronal artificial entrenada mediante descenso del gradiente y retropropagación.

$$\Phi(z) = \begin{cases} z, & z \leq 0 \\ 0, & z < 0 \end{cases} \quad \text{o también} \quad \Phi(z) = \max(0, z)$$



- Softmax: Esta función es usada sobretodo en clasificación múltiple y devuelve la probabilidad de cada clase, siendo la probabilidad más alta la que se escoge como clase a la que pertenece la muestra a clasificar.

$$\Phi(z_i) = \frac{e^{z_i}}{\sum_{j=0}^k e^{z_j}}$$

Como regla general, se tiene que la función de activación de las capas ocultas puede ser una función ReLU y en la capa de salida dependerá del tipo de problema que se pretende resolver, así:

- Clasificación binaria: Función de activación sigmoide
- Clasificación multclase: Función de activación softmax
- Regresión: Función de activación lineal.

11.2 Entrenamiento de una red neuronal

El entrenamiento de una red comprende dos procesos: propagación hacia adelante (*forward propagation*) y propagación hacia atrás (*back propagation*) que son llevados a cabo de manera repetitiva hasta alcanzar un umbral específico o un número n de iteraciones.

En *forward propagation* o *feed forward* las muestras de datos atraviesan la red neuronal desde las capas inferiores hasta las capas superiores, en donde las neuronas de dichas capas van recibiendo transformaciones por medio de las funciones de activación. La información de salida de las neuronas de una capa alimenta a las de las otras capas de manera progresiva hasta llegar a la última, en la cual se calcula la predicción final.

Con relación a la red con la que iniciamos este capítulo tenemos que:

- La variable de salida de la capa oculta (también llamada activación) puede ser expresada matemáticamente como sigue:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$\alpha^{[1]} = f(z^{[1]})$$

- La salida de la última capa se expresaría así:

$$z^{[2]} = W^{[2]}\alpha^{[1]} + b^{[2]}$$

$$\alpha^{[2]} = f(z^{[2]})$$

Aquí la función $f(z)$ es una función de activación como las ya estudiadas, que sirven básicamente para determinar cómo activar las neuronas de la red.

Ahora, ¿Cómo determinamos si la anterior salida es pertinente, es decir, cómo sabemos qué tan bien lo hizo nuestra red? Para responder esta pregunta suponga que pasamos por la red una muestra x y que además estamos ante un problema de aprendizaje supervisado, debe calcularse la diferencia entre la salida de la red y la etiqueta actual de la muestra, este es el error que como ya hemos indicado se puede determinar en una función de costo. La escogencia de la función de costo depende de la tarea de aprendizaje automático, por ejemplo, para regresión se puede utilizar el error cuadrático medio:

$$MSE = J(W) = \sum_{i=1}^n \frac{1}{2} (y - \hat{y})^2$$

Mientras que para clasificación una función de costo muy utilizada es la pérdida de entropía cruzada:

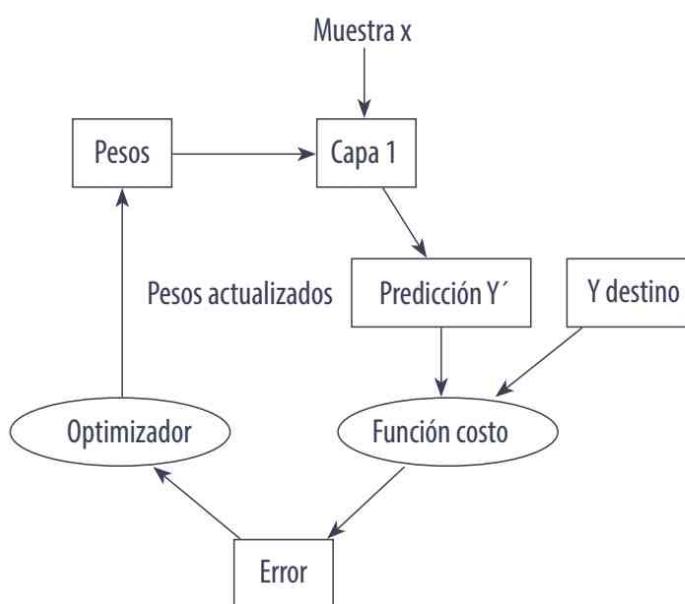
$$CE = - \sum_{i=1}^c y_i * \log \hat{y}$$

donde c es el número de clases, y es la etiqueta actual e \hat{y} es el valor de la predicción. En una tarea de clasificación binaria esta función se puede reescribir de la siguiente manera:

$$CE = - \sum_{i=1}^2 y_i * \log \hat{y} = - y_1 * \log \hat{y}_1 - (1 - y_1) \log(1 - \hat{y}_1)$$

Al principio como es de esperarse habrá valores altos en el error, en gran medida debido a que los parámetros son inicializados al azar, pero una vez se ha repetido el mismo proceso un número óptimo de veces, la red ajustará estos parámetros y el error irá disminuyendo en capa época o iteración. Al final la meta es encontrar un conjunto de valores para los pesos de todas las capas de la red, de manera tal que le permitan a esta mapear acertadamente las entradas recibidas con la clase correspondiente. Esta tarea de optimización de los pesos del modelo es propia de algoritmos como el descenso del gradiente que nos da un vector con la dirección del mínimo local. En redes neuronales los gradientes son calculados por medio de otro algoritmo muy potente llamado *backpropagation* que estudiaremos en la siguiente lección.

La próxima imagen muestra el proceso general de entrenamiento de una red neuronal.



Backpropagation:

El algoritmo de propagación hacia atrás o retropropagación como también se le conoce, encuentra cuál es la contribución que hizo cada parámetro de la red en la obtención del error, para ello, hace uso de la conocida regla de la cadena para determinar tal error con respecto a los pesos en la capa de salida y en las capas ocultas. Esto aplicado a la red neuronal descrita al comienzo, la cual dispone de una sola capa de salida y una sola capa oculta nos dará las siguientes derivadas parciales:

$\frac{\partial J(W)}{\partial w^{[2]}}$ y $\frac{\partial J(W)}{\partial w^{[1]}}$, estas ecuaciones equivalen a:

$$\frac{\partial J(W)}{\partial w^{[2]}} = \frac{\partial J(W)}{\partial a^{[2]}} * \frac{\partial a^{[2]}}{\partial z^2} * \frac{\partial z^{[2]}}{\partial w^{[2]}} \quad (1)$$

$$\frac{\partial J(W)}{\partial w^{[1]}} = \frac{\partial J(W)}{\partial a^{[2]}} * \frac{\partial a^{[2]}}{\partial z^2} * \frac{\partial z^{[2]}}{\partial a^{[2]}} * \frac{\partial a^{[1]}}{\partial z^1} * \frac{\partial z^{[1]}}{\partial x^{[1]}} * \frac{\partial x^{[1]}}{\partial w^{[1]}} \quad (2)$$

Resolvamos la ecuación 1 por separado para una mayor simplicidad:

$$\begin{aligned} \frac{\partial J(W)}{\partial a^{[2]}} &= \frac{\partial (\frac{1}{2}(y - \hat{y})^2)}{\partial \hat{y}}, \text{ recuerde que } a^{[2]} = \hat{y} \\ &= 2 * \frac{1}{2} (y - \hat{y})^{2-1} * (0 - 1) \\ &= (y - \hat{y}) * -1 \\ &= \hat{y} - y \end{aligned}$$

$$\begin{aligned} \frac{\partial a^{[2]}}{\partial z^2} &= \frac{f(z^{[2]})}{\partial z^2} \\ &= \frac{\partial}{\partial z^2} \left(\frac{1}{1+e^{-z}} \right) \\ &= \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}} \right), (\text{o también } f'(z^{[2]})) \end{aligned}$$

$$\frac{\partial z^{[2]}}{\partial w^{[2]}} = \frac{\partial \sum a^{[2]} w^{[2]}}{\partial w^{[2]}} = a^{[2]}$$

Luego,

$$\begin{aligned}\frac{\partial J(W)}{\partial w^{[2]}} &= \frac{\partial J(W)}{\partial a^{[2]}} * \frac{\partial a^{[2]}}{\partial z^2} * \frac{\partial z^{[2]}}{\partial w^{[2]}} \\ &= (\hat{y} - y)f'(z^{[2]}) a^{[2]}\end{aligned}$$

Resolvamos ahora la ecuación 2 tal y como hicimos con la primera:

- $\frac{\partial a^{[1]}}{\partial z^{[1]}} = f'(z^{[1]})$
- $\frac{\partial z^{[1]}}{\partial x_i} = \frac{\partial \sum_{i=1}^n x_i * w^{[1]}}{\partial x_i} = w^{[1]}$
- $\frac{\partial z^{[1]}}{\partial w^{[1]}} = \frac{\partial \sum x_i * w^{[1]}}{\partial w^{[1]}} = x_i$
- $\frac{\partial z^{[2]}}{\partial a^{[2]}} = \frac{\partial \sum a^{[2]} * w^{[2]}}{\partial a^{[2]}} = w^{[2]}$

Entonces tenemos:

$$\begin{aligned}\frac{\partial J(W)}{\partial w^{[1]}} &= \frac{\partial J(W)}{\partial a^{[2]}} * \frac{\partial a^{[2]}}{\partial z^2} * \frac{\partial z^{[2]}}{\partial a^{[2]}} * \frac{\partial a^{[1]}}{\partial z^1} * \frac{\partial z^{[1]}}{\partial x^{[1]}} * \frac{\partial z^{[1]}}{\partial w^{[1]}} \\ &= (\hat{y} - y)f'(z^{[2]}) f'(z^{[1]}) w^{[2]} w^{[1]} x_i \\ &= (\hat{y} - y)f'(z^{[2]}) f'(z^{[1]}) w^{[2]} a^{[1]}\end{aligned}$$

El algoritmo de retropropagación sigue los pasos a continuación:

1. Propagar la información a través de toda la red partiendo de la capa de entrada y calcular las salidas de las funciones de activación en las capas ocultas hasta obtener una predicción \hat{y} ($a^{[2]}$ en nuestro ejemplo.) en la capa de salida. Este paso es la misma propagación hacia adelante (*forward propagation*).
2. En la última capa, calculamos la derivada de la función de costo respecto a los pesos y a los sesgos de la capa de salida.

$$\delta^{(2)} = \frac{\partial J(W)}{\partial w^{[2]}} = (\hat{y} - y)f'(z^{[2]}) a^{[2]}$$

3. Para la capa oculta, calculamos la derivada de la función de costo con respecto a los pesos y a los sesgos de la capa oculta.

$$\delta^{(1)} = \frac{\partial J(W)}{\partial w^{[1]}} = (\hat{y} - y)f'(z^{[2]}) f'(z^{[1]}) w^{[2]} a^{[1]}$$

4. Se actualizan los pesos con los gradientes obtenidos en el paso anterior aplicando la tasa de aprendizaje α .
5. Se repite el proceso anterior hasta que la función de costo converja o se haya alcanzado un número fijado de iteraciones.

$$W^{[1]} := W^{[1]} - \frac{1}{m} \alpha \Delta W^{[1]}$$

$$W^{[2]} := W^{[2]} - \frac{1}{m} \alpha \Delta W^{[2]}$$

Es momento de intentar programar el algoritmo de propagación hacia atrás en Python empleando el dataset Breast Cancer, tomando las columnas: *mean radius*, *mean texture*, *mean perimeter*.

Como siempre cargamos los datos y normalizamos las 3 variables anteriores, esto último es fundamental para que la función de costo converja mucho más rápidamente. Luego sepáramos un 20% para prueba y el resto para entrenar el modelo.

```
from sklearn import datasets
dataset = datasets.load_breast_cancer()
X = dataset.data
y = dataset.target
X = X[:, [1,2,3]] # tomamos mean radius, mean texture y mean perimeter
y = y
# estandarizamos los datos
X[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
X[:,2] = (X[:,2] - X[:,2].mean()) / X[:,2].std()

from sklearn.model_selection import train_test_split
X_ent, X_pru, y_ent, y_pru = train_test_split(X, y, test_size=0.2, random_
state=2)
#número de muestras de entrenamiento
m = X_ent.shape[0]
y_ent = y_ent[:,np.newaxis]
y_pru = y_pru[:,np.newaxis]
```

Colocamos algunas funciones utilitarias que permitirán dar forma a nuestra implementación de red neuronal:

```

def calcular_costo(a2, y):
    costo=-1/m * (np.sum(np.multiply(np.log(a2), y) + np.multiply((1 - y), np.log(1 - a2))))
    return costo

def sigmoide(t):
    return 1/(1+np.exp(-t))

def derivada_sigmoide(p):
    return sigmoide(p) * sigmoide(1 - p)

def relu(Z):
    return np.maximum(0, Z)
def derivada_Relu(x):
    x[x<=0] = 0
    x[x>0] = 1
    return x

def exactitud(y, yhat):
    acc = int(sum(y == yhat) / len(y) * 100)
    return acc

def predecir(x):
    w1 = modelo["w1"]
    b1 = modelo["b1"]
    w2 = modelo["w2"]
    b2 = modelo["b2"]
    z1 = np.dot(x, w1) + b1
    a1 = relu(z1)
    z2 = np.dot(a1, w2) + b2
    a2 = sigmoide(z2)
    p = np.round(a2)
    return p

```

Definimos la función *forward()* que realiza la propagación hacia adelante, para la primera capa aplica la función de activación Relu, mientras que para la segunda la función de activación sigmoide. También definimos la función *backward()* que lleva a cabo la propagación hacia atrás:

```

def forward(X, w1, w2, b1, b2):
    z1 = np.dot(X,w1) + b1
    a1 = relu(z1)
    z2 = np.dot(a1, w2) + b2
    a2 = sigmoide(z2)

    # cache guarda los datos que serán usados en backpropagation
    cache = {"z1":z1,
              "a1":a1,
              "z2":z2,
              "a2":a2}
    return a2, cache

def backward(w1, b1, w2, b2, cache, alfa):
    a1 = cache['a1']
    a2 = cache['a2']
    z1 = cache['z1']
    z2 = cache['z2']

    dz2 = (a2 - y_ent) * derivada_sigmoide(z2)
    dw2 = np.dot(a2.T ,dz2)
    db2 = np.mean(dz2, axis = 0)

    dz1 = np.dot(dz2, w2.T) * derivada_Relu(z1)
    dw1 = np.dot(X_ent.T, dz1)
    db1 = np.mean(dz1, axis = 0)

    w2 = w2 - alfa * dw2 / m
    b2 = b2 - alfa * db2 / m
    w1 = w1 - alfa * dw1 / m
    b1 = b1 - alfa * db1 / m
    return w1, w2, b1, b2

```

La siguiente función sirve para entrenar a nuestra red neuronal densamente conectada, o bien sea, a nuestro perceptrón multicapa como también se le conoce a este tipo de redes. Tendrá 3 unidades en la entrada, 10 unidades en la capa oculta y una unidad en la capa de salida. En cuanto a la primera capa hay $3 \times 10 = 30$ pesos y 10 unidades de sesgo y para la segunda capa habrá $10 \times 1 = 10$ pesos y 1 unidad de sesgo, para un total de 51 parámetros entrenables.

```

def entrenar(X, y, alfa, num_iter):
    global modelo
    global errores
    errores = []
    unid_entrada = X.shape[1]
    unid_oculta = 10
    unid_salida = 1

    np.random.seed(1)
    w1 = np.random.randn(unid_entrada, unid_oculta) # pesos capa 1
    b1 = np.random.randn(1, unid_oculta) # sesgo capa 1
    w2 = np.random.randn(unid_oculta, unid_salida) # pesos capa 2
    b2 = np.random.randn(1, unid_salida) # sesgo capa 2

    for i in range(0, num_iter):
        a2, cache = forward(X, w1, w2, b1, b2)
        costo = calcular_costo(a2, y)
        errores.append(costo)
        w1, w2, b1, b2 = backward(w1, b1, w2, b2, cache, alfa)
        # Imprimimos el costo cada 100 iteraciones
        if i % 100 == 0:
            print ("Costo después de % i: % f" % (i, costo))

    modelo = {"w1": w1,"b1": b1,"w2": w2,"b2": b2}

```

Acto seguido, invocamos el método `entrenar()` con los datos de entrenamiento, una tasa de aprendizaje de 0.01 y 10.000 épocas, veremos como el modelo empieza a converger aproximándose a 0.

```
entrenar(X_ent, y_ent, alfa = 0.01, num_iter = 10000)
```

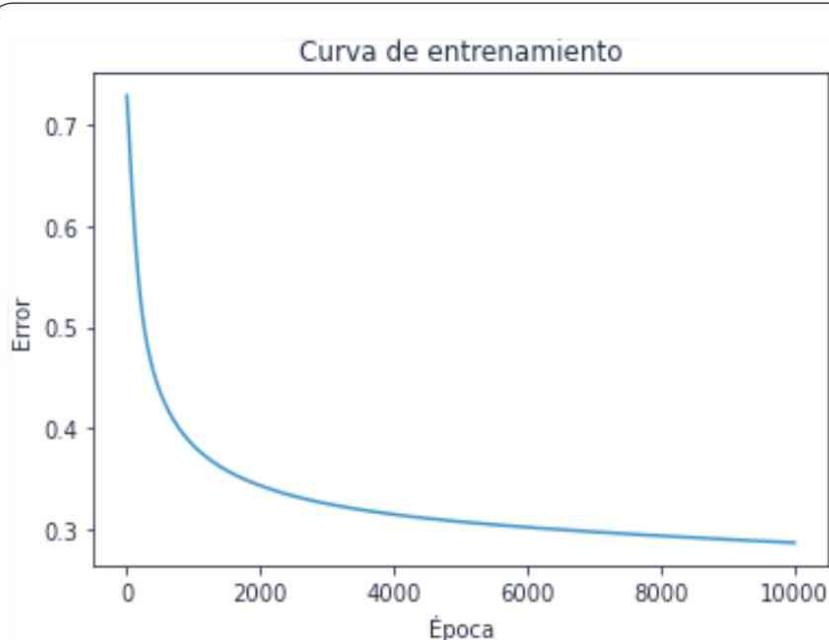
```

Costo después de 0: 0.729519
Costo después de 100: 0.611060
Costo después de 200: 0.529316
Costo después de 300: 0.483042
Costo después de 400: 0.454370
...
Costo después de 9500: 0.273689
Costo después de 9600: 0.273233
Costo después de 9700: 0.272783

```

Costo después de 9800: 0.272337
Costo después de 9900: 0.271896

```
import matplotlib.pyplot as plt
plt.plot(errores)
plt.xlabel("Época")
plt.ylabel("Error")
plt.title("Curva de entrenamiento")
plt.show()
```



Finalmente calculamos la exactitud con los datos de entrenamiento y con los de prueba.

```
pred_ent = predecir(X_ent)
pred_pru = predecir(X_pru)
print("Exactitud de entrenamiento: {}".format(exactitud(y_ent, pred_ent)))
print("Exactitud de prueba: {}".format(exactitud(y_pru, pred_pru)))
```

Exactitud de entrenamiento: 88
Exactitud de prueba: 86

Si bien los resultados anteriores parecen buenos es posible mejorarlos un poco más ajustando los parámetros, no obstante, no se recomienda usar este tipo modelos creados manualmente en contextos reales, en tales escenarios es más apropiado usar librerías como Scikit Learn o Keras, esta última la estaremos estudiando en el próximo capítulo. Estas librerías ofrecen algoritmos mucho más optimizados para abordar muchos tipos de problemas de manera más eficiente y sencilla. Por el momento hagamos una prueba con Scikit-learn que tiene la clase MLPClassifier con la cual podemos crear un perceptrón de una manera mucho más apacible y con muchas menos líneas de código. La implementación con Scikit sería la siguiente:

```
from sklearn.neural_network import MLPClassifier
clf = MLPClassifier(max_iter=10000, # iteraciones
                     learning_rate_init=0.01, # tasa de aprendizaje
                     hidden_layer_sizes=10, # número de unidades en la capa oculta
                     activation='relu', # función de activación en la capa oculta
                     solver='sgd') # optimizador descenso del gradiente estocastico
clf.fit(X_ent, y_ent.ravel())
clf.predict_proba(X_pru[:1])
exac = clf.score(X_pru, y_pru)
print("Exactitud {:.2f}".format(exac))
```

Exactitud 0.89

Vemos que la exactitud con los datos de prueba es muy parecida a la que conseguimos con la implementación manual.

Con esto damos por terminado esta sección, pero en la siguiente exploraremos el API Keras que es una de las librerías más usadas en aprendizaje profundo y que se ha ganado una gran popularidad por ser bastante robusta y sencilla de emplear.

11.3 Red neuronal para clasificación binaria

Keras es una librería de código abierto de alto nivel usada para crear redes neuronales, además se ejecuta sobre frameworks como por ejemplo Tensorflow y goza de una gran popularidad en el campo del aprendizaje profundo. Con esta API se pueden crear redes neuronales muy potentes con pocas líneas de código.

En esta sección crearemos una red densamente conectada como la del subapartado anterior usando keras por lo que primeramente debemos instalarlo, para ello

basta con ejecutar el comando: *pip install keras* dentro del entorno de línea de comandos de Anaconda.

Como siempre el paso inicial antes de implementar algún algoritmo de estas librerías es hacer la importación de las clases a utilizar. Comenzamos importando tensorflow y keras, después la clase Sequential del submódulo keras.models, dado que nuestro ejercicio consistirá en un modelo de red neuronal organizado en bloques (secuencial), y finalmente con la clase Dense del submódulo keras.layers puntualizamos el tipo de capas de nuestra red, en este caso serán densas por motivo de tratarse de una red con capas densamente conectadas.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

Seguidamente se crea el objeto de la clase Sequential y las capas que harán parte del modelo. La primera capa oculta retorna 10 unidades, utiliza la función de activación ReLU y por último la forma de los datos de entrada, en este ejemplo un array unidimensional de 3 elementos.

```
modelo = Sequential()
modelo.add(Dense(10, activation='relu', input_shape=(X.shape[1])))
modelo.add(Dense(5, activation='relu'))
modelo.add(Dense(1, activation='sigmoid'))
```

Podrá notar, que a diferencia de los ejemplos planteados en otros subapartados se ha agregado una capa oculta adicional para mostrar que es posible agregar tantas capas ocultas como se desee a fin de intentar conseguir un modelo más eficaz, no obstante, esta decisión implica incurrir en un mayor consumo de recursos de máquina, porque implica el surgimiento de más parámetros para el modelo. Para conocer cuántos parámetros posee una red se puede usar la función *summary()*.

```
modelo.summary()
```

```
Model: "sequential_9"
```

Layer (type)	Output Shape	Param #
<hr/>		
dense_20 (Dense)	(None, 10)	40
dense_21 (Dense)	(None, 5)	55
dense_22 (Dense)	(None, 1)	6
<hr/>		
Total params:	101	
Trainable params:	101	
Non-trainable params:	0	

Observe que con la capa oculta adicional el número de parámetros entrenables ha pasado de 51 a 101.

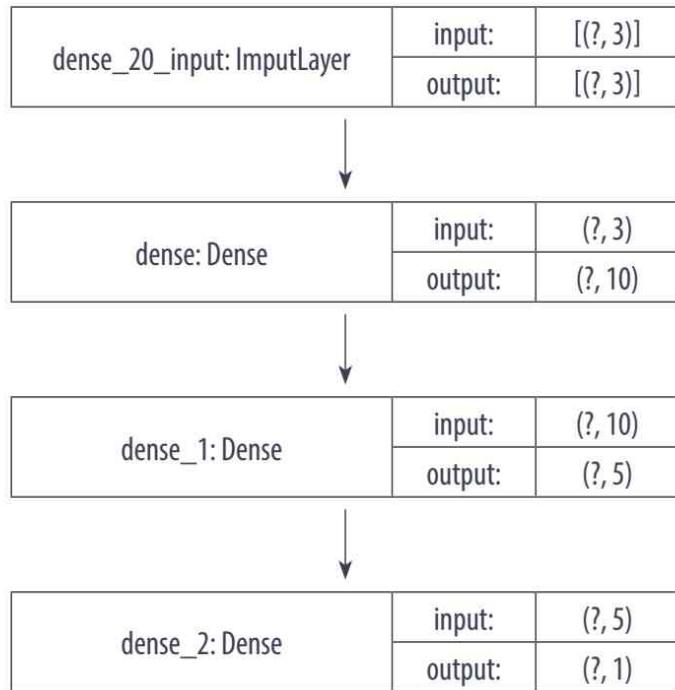
El siguiente paso es definir el optimizador gradiente de descenso estocástico (optimizer='sgd') con el que se va a buscar minimizar la función de coste: entropía cruzada binaria (loss='binary_crossentropy'), con estos argumentos configurados podemos compilar el modelo con la función compile().

```
from keras import optimizers
sgd = optimizers.SGD(lr=0.01)

modelo.compile(loss='binary_crossentropy', optimizer='sgd',
                metrics=['accuracy'])
```

Ahora con el método plot_model() del submódulo keras.utils, podemos ver la estructura de la red neuronal diseñada:

```
keras.utils.plot_model(modelo, show_shapes=True)
```



Ha llegado el momento de entrenar la red con el método `fit()`, le pasamos los datos de entrenamiento `X` e `y`, la cantidad de iteraciones (`epochs=1000`), con `batch_size=64` establecemos un número de entradas para entrenar la red en cada época (lote), esta asignación significa que se empleará el gradiente de descenso por minilotes que busca un equilibrio entre el gradiente de descenso estocástico, que usa una sola muestra escogida aleatoriamente por cada época y el gradiente de descenso de lote completo, el cual como su nombre lo indica usa todo el conjunto de datos en cada iteración. Se ha demostrado que el SGD de minilotes actúa de mejor manera que las otras dos formas, por eso en la práctica suele usarse con mayor frecuencia. Como parámetro final del método `fit()` se establece un 20% de los datos de entrenamiento para validar el modelo. (`validation_split=0.2`).

```
historial = modelo.fit(X_ent, y_ent ,epochs=1000 , batch_size=64, validation_
split=0.2)
```

```
Epoch 1/1000
6/6 [=====] - 0s 22ms/step - loss: 0.6090 - accuracy: 0.6071 - val_loss: 0.6116 - val_accuracy: 0.6154
Epoch 2/1000
6/6 [=====] - 0s 7ms/step - loss: 0.5998 - accuracy: 0.6071 - val_loss: 0.6038 - val_accuracy: 0.6264
Epoch 3/1000
```

...

Epoch 998/1000
 6/6 [=====] - 0s 6ms/step - loss: 0.2165 -
 accuracy: 0.9038 - val_loss: 0.2313 - val_accuracy: 0.9121

Epoch 999/1000
 6/6 [=====] - 0s 5ms/step - loss: 0.2164 -
 accuracy: 0.9038 - val_loss: 0.2313 - val_accuracy: 0.9121

Epoch 1000/1000
 6/6 [=====] - 0s 6ms/step - loss: 0.2166 -
 accuracy: 0.9038 - val_loss: 0.2312 - val_accuracy: 0.9121

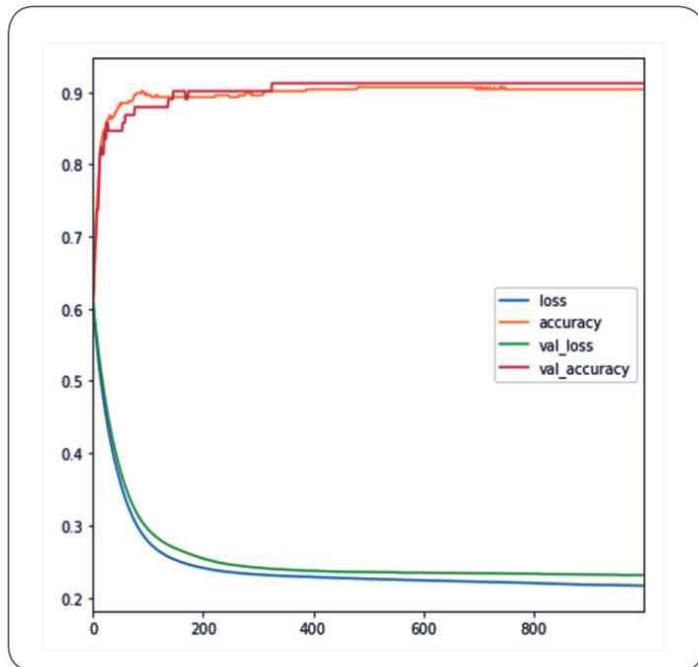
Una vez realizado el entrenamiento obtenemos la exactitud del modelo, siendo esta igual a la conseguida en el ejemplo del Perceptrón implementado con Scikit-learn:

```
perdida, exactitud = modelo.evaluate(X_pru, y_pru)
print("La exactitud es: {:.2f}".format(exactitud))
```

4/4 [=====] - 0s 999us/step - loss: 0.2689 -
 accuracy: 0.8860
 La exactitud es: 0.89

Y por último, mostramos una gráfica a partir del historial de entrenamiento, la cual nos permite ver cómo se comportan las curvas de la pérdida (*loss*) con los datos de entrenamiento y validación, evidenciando una disminución a medida que van aumentando las iteraciones. Así mismo, se aprecia como la exactitud con los datos de entrenamiento y validación por el contrario experimentan un aumento significativo antes de la época 200 y que se mantiene estable hasta el final:

```
import pandas as pd
pd.DataFrame(historial.history).plot(figsize=(7,7))
```



11.4 Red neuronal para clasificación múltiple

En esta sección crearemos una red neuronal de clasificación multclase y usaremos el dataset Fashion Mnist disponible en Keras. Este conjunto de datos contiene imágenes sobre ropa de 28x28 pixeles con 60.000 muestras para entrenamiento y 10.000 para prueba, dispuestas en escala de grises y que pertenecen a una de las siguientes categorías: T-shirt/top (blusa), Trouser (pantalón), Pullover (chaqueta), Dress (vestido), Coat (corbata), Sandal (sandalia), Shirt (sueter), Sneaker (zapatilla), Bag (bolso), Ankle boot (bota). Tal y como hicimos con el dataset anterior debemos importarlo.

```
from keras.datasets import fashion_mnist
(X_ent, y_ent), (X_pru, y_pru) = fashion_mnist.load_data()
```

Al cargarlo obtenemos 4 arrays de Numpy, donde X_ent, X_pru contiene la información numérica de las imágenes con la forma (60000, 28, 28), mientras que y_ent, y_pru contienen las etiquetas (enteros en el rango entre 0 y 9) con la forma (10000,).

```
print("Forma X entrenamiento:",str(X_ent.shape))
print("Forma X prueba:",str(X_pru.shape))
print("Forma y entrenamiento",str(y_ent.shape))
print("Forma y prueba",str(y_pru.shape))
```

Forma X entrenamiento: (60000, 28, 28)

Forma X prueba: (10000, 28, 28)

Forma y entrenamiento (60000,)

Forma y prueba (10000,)

Mostramos algunas imágenes del conjunto de entrenamiento con la función imshow() de Matplotlib:

```
# creamos una lista con los nombres de las etiquetas
columnas = ["Blusa", "Pantalón", "Chaqueta", "Vestido", "Saco", "Sandalia",
            "Sueter", "Zapatilla", "Bolso", "Bota"]
plt.figure(figsize=(12,8))
for i in range(15):
    plt.subplot(3,5,i + 1)
    plt.imshow(X_ent[i])
    plt.title(columnas[y_ent[i]])
    plt.axis('off')
plt.show()
```



Ahora que ya sabemos un poco de los datos del Fashion MNIST el paso que sigue es hacer un trabajo de preprocesamiento sobre los valores de X y de y. X se escala en un rango entre 0 y 1 simplemente dividiéndolo entre 255 e y se le aplica una

codificación de tipo `one_hot_encoding`, que recordemos es una codificación binaria que utiliza un vector de tamaño igual al total de etiquetas de clase y coloca un 1 en la posición cuyo índice coincide con la salida esperada y ceros en las demás posiciones. Este trabajo se puede hacer por medio del método `to_categorical()` del submódulo `utils`.

```
# preprocessado
X_ent = X_ent.astype('float32')
X_pru = X_pru.astype('float32')
X_ent = X_ent / 255
X_pru = X_pru / 255
from keras.utils import to_categorical
y_ent = to_categorical(y_ent, num_classes=10)
y_pru = to_categorical(y_pru, num_classes=10)
```

Por ejemplo, si imprimimos la etiqueta de la primera muestra obtenemos un arreglo binario con un 1 en la última posición, correspondiente a la clase bota que es la última etiqueta del dataset.

```
print(y_ent[0])
```

[0. 0. 0. 0. 0. 0. 0. 0. 1.]

Vamos ahora a entrenar la red con dos capas ocultas con función de activación ReLU y softmax, ambas con 10 neuronas. La primera capa le especificamos que procesará entradas (imágenes) de 28x28 por medio del argumento `input_shape=(28,28)`. Esta capa tendrá una neurona por cada pixel, en total 784. Además, tiene una capa `Flatten` que agrega una dimensión extra al arreglo de entrada quedando así: 28,28,1. Este último valor es llamado canal o profundidad y se establece en 1 cuando las imágenes son en escala de grises como en este caso y a 3 cuando son a color.

```
# crear el modelo
modelo = keras.Sequential()
modelo.add(keras.layers.Flatten(input_shape=(28,28)))
modelo.add(keras.layers.Dense(10, activation='sigmoid'))
modelo.add(keras.layers.Dense(10, activation='softmax'))
```

Podemos hacer un resumen de nuestro modelo y ver que dispone de 7.960 parámetros entrenables utilizando el ya conocido método `summary()`:

```
print(modelo.summary())
```

```
Model: "sequential_2"
Layer (type)          Output Shape       Param #
=====
flatten_2 (Flatten)    (None, 784)        0
dense_3 (Dense)        (None, 10)         7850
dense_4 (Dense)        (None, 10)         110
=====
Total params: 7,960
Trainable params: 7,960
Non-trainable params: 0
=====
None
```

La tarea ahora es compilar el modelo, en esta oportunidad la `loss` será `categorical_crossentropy` puesto que se tratará de un problema de clasificación multiclase. Recuerde que la función de pérdida depende del problema de machine learning que se esté abordando. Por su parte usaremos también el optimizador Adam que tiene muy buen rendimiento en comparación con otros de su tipo.

Tipo de problema	Función de activación	Función de pérdida
Clasificación multiclase con one hot	softmax	categorical_crossentropy
Clasificación multiclase con índice categórico	softmax	sparse_categorical_crossentropy
Regresión	ninguna	mse
Clasificación binaria	sigmoid	binary_crossentropy

Seguidamente compilamos y entrenamos el modelo con un total de 10 épocas.

```
#configurar el modelo  
modelo.compile(loss='categorical_crossentropy', optimizer='adam',  
                metrics=['accuracy'])  
# entremos  
modelo.fit(X_ent, y_ent , epochs=10 , verbose=1)
```

Epoch 1/10
1875/1875 [=====] - 4s 2ms/step - loss:
0.6918 - accuracy: 0.7493
Epoch 2/10
1875/1875 [=====] - 4s 2ms/step - loss:
0.4657 - accuracy: 0.8380
Epoch 3/10
1875/1875 [=====] - 4s 2ms/step - loss:
0.4296 - accuracy: 0.8511
Epoch 4/10
1875/1875 [=====] - 4s 2ms/step - loss:
0.4110 - accuracy: 0.8569
Epoch 5/10
1875/1875 [=====] - 4s 2ms/step - loss:
0.4003 - accuracy: 0.8604
Epoch 6/10
1875/1875 [=====] - ETA: 0s - loss: 0.3915 -
accuracy: 0.86 - 4s 2ms/step - loss: 0.3922 - accuracy: 0.8626
Epoch 7/10
1875/1875 [=====] - 4s 2ms/step - loss:
0.3866 - accuracy: 0.8640
Epoch 8/10
1875/1875 [=====] - 4s 2ms/step - loss:
0.3812 - accuracy: 0.8646
Epoch 9/10
1875/1875 [=====] - 4s 2ms/step - loss:
0.3749 - accuracy: 0.8668
Epoch 10/10
1875/1875 [=====] - 4s 2ms/step - loss:
0.3717 - accuracy: 0.8690

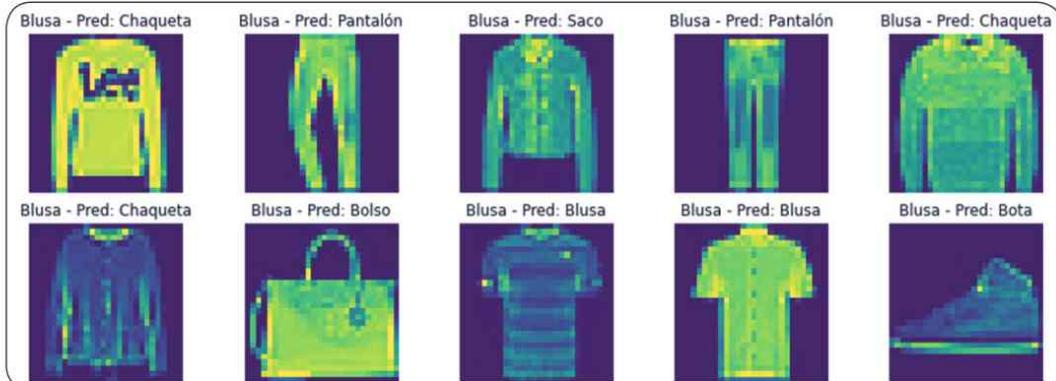
La evaluación nos arroja un 0.84 que se puede mejorar probando con otros valores para los hiperparámetros y con otras técnicas que estaremos viendo en los próximos capítulos.

```
# evaluamos
loss, exac = modelo.evaluate(X_pru, y_pru)
print("La exactitud es: {:.2f}".format(exac))
```

Finalmente, con el siguiente código mostraremos algunas predicciones de nuestro modelo, que nos permiten evidenciar que puede fallar como en el caso de la sexta predicción donde la clase esperada era saco y predijo chaqueta.

```
from keras.preprocessing import image
import numpy as np

plt.figure(figsize=(15,5))
x=0
for i in [1, 5, 10, 15, 20, 25, 30, 35, 40, 45]:
    plt.subplot(2,5,x + 1)
    img = X_pru[i]
    plt.imshow(img)
    img = (np.expand_dims(img,0))
    res = modelo.predict(img)
    plt.title(columnas[np.argmax(y_pru[i])] + " - Pred: " + columnas[np.argmax(res)])
    plt.axis('off')
    x=x+1
plt.show()
```



Con este ejercicio de clasificación múltiple con una red neuronal profunda en keras terminamos el capítulo y nos aprestamos a conocer otra arquitectura muy utilizada en deep learning principalmente en problemas de visión por computador, las redes neuronales convolucionales.

<https://dogramcode.com/bloglibros>



CAPÍTULO 12

Redes neuronales convolucionales

Temas

- 12.1 Introducción a las redes neuronales convolucionales
- 12.2 CNNs en Keras
- 12.3 Regularización y dropout

Las redes neuronales convolucionales significaron un gran avance en el área de la visión por computador, ya que permiten de una manera eficaz resolver problemas de reconocimiento de imágenes, lo cual ha motivado a muchos investigadores y académicos a centrar su atención en ellas y convertirlas en objeto de estudio en sus trabajos investigativos.

12.1 Introducción a las redes neuronales convolucionales

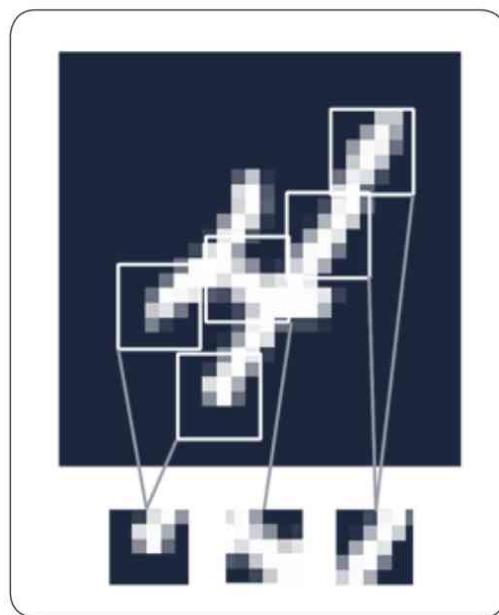
Las redes neuronales convolucionales (convnets o CNNs) son una potente herramienta de clasificación de imágenes, inspirada en la forma como trabaja la corteza visual de nuestro cerebro, la cual se encuentra conformada por múltiples niveles, donde a medida que se avanza a través de ellos es reconocida más y más información.

Por lo general, una red neuronal convolucional parte de identificar simples píxeles, pasando por el reconocimiento de elementales formas geométricas, hasta llegar a identificar elementos mucho más sofisticados como objetos, animales, cuerpos, caras, etc.

Si bien las CNNs surgieron en la década de los ochenta han tenido en los últimos años gran auge gracias al aumento exponencial de la computación, en la cantidad

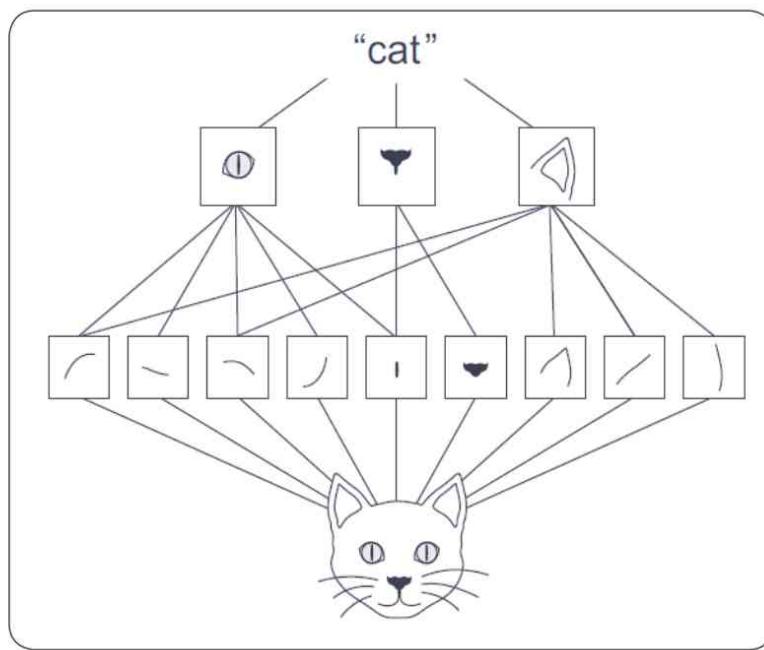
de datos masivos y de algoritmos especiales que han permitido entrenar este tipo de redes, no solo en problemas de visión artificial sino también en otros como el reconocimiento de voz o el procesamiento de lenguaje natural (PLN).

A diferencia de las redes neuronales completamente conectadas vistas en el capítulo anterior las cuales aprenden patrones globales del espacio de características de entrada, por ejemplo, en el caso de imágenes, patrones que resultan de considerar todos los pixeles en cada iteración, las capas de convolución aprenden patrones locales, es decir, patrones encontrados en pequeñas franjas o ventanas de las características de entrada.



Las capas de una red convolucional aprenden patrones invariantes de traslación. Esto es, después de aprender un cierto patrón en un lugar específico, es capaz de reconocerlo en cualquier otro sitio, a diferencia de las redes completamente conectadas que tendrían que aprender el patrón nuevamente si apareciera en una nueva ubicación.

Además, las CNNs pueden aprender jerarquías espaciales de patrones: una primera capa aprenderá patrones simples como líneas, bordes, texturas. Una segunda capa tomará lo aprendido por la primera capa y aprenderá otros patrones más significativos como por ejemplo formas, que de tratarse de una cara serían los ojos, la nariz, etc. Esto se hace sucesivamente hasta llegar a las últimas capas en donde se aprenden elementos visuales más complejos y abstractos (caras, o cualquier objeto), para posteriormente dar con la clasificación de la imagen pertinente.



En general, la arquitectura de las CNNs la conforman básicamente una serie de capas convolucionales y capas de agrupación que se van alternando, las cuales trabajan con dos operaciones que son la convolución y el agrupamiento respectivamente y que describiremos a continuación.

12.1.1. Convolución

Matemáticamente hablando, una convolución es una correlación cruzada, es decir, una medida de la similitud entre dos señales, las cuales en el caso de las CNNs son multidimensionales. Si hablamos de una imagen de dos dimensiones a blanco y negro la ecuación que define una convolución sería:

$$S(i, j) = (X * K)(i, j) = \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{\infty} X(i - m, j - n) * K(m, n)$$

Donde K es conocido como kernel (núcleo) y X corresponde con la imagen de entrada.

La operación de convolución se puede entender como una ventana deslizante K (filtro) de tamaño $m_1 * m_2$ que hace un barrido a través de una imagen X de tamaño $n_1 * n_2$, con $m_1 \leq n_1$ y $n_2 \leq m_2$, con el fin de resaltar o disminuir ciertas características de la imagen de entrada. La convolución aplica una multiplicación con X y K para obtener una matriz $S = X * K$, denominada mapa de características.

Un filtro se enfoca en una característica particular de una imagen, sencillamente es una matriz de pesos cuyo tamaño suele ser de 3x3 o 5x5 más una unidad de sesgo, aprendidos durante el entrenamiento de la red neuronal basándose en la minimización de una función de error. No se acostumbra a usar tamaños más grandes para el filtro debido a que se pueden pasar desapercibidas ciertas características pequeñas pero significativas dentro de un contexto determinado.

Si tomamos el ejemplo intuitivo considerando que los filtros actúan como ventanas deslizantes que se mueven a través de la entrada, inicialmente la ventana se ubicaría en la parte superior izquierda de la imagen, entonces sus pesos se multiplican no con todas las neuronas de la capa de entrada como sucede con las redes densamente conectadas, sino con una franja o porción de esta, por ejemplo, de 5x5 neuronas de la capa de entrada. El resultado de la convolución alimenta a una neurona de la capa oculta siguiente. Luego si la ventana se desplaza un pixel hacia la derecha el resultado de la convolución conectaría a la siguiente neurona de la capa oculta, y así sucesivamente. Este proceso se repetirá de izquierda a derecha y de arriba hacia abajo hasta que la ventana alcance la parte inferior derecha de la imagen y por lo tanto no pueda hacer más desplazamientos. Es importante destacar que se usa el mismo filtro para hacer la operación de todas las neuronas de la capa oculta, este aspecto disminuye notoriamente el número de parámetros de la red neuronal.

$$\begin{array}{c}
 X \\
 \begin{array}{|c|c|c|c|c|} \hline 1 & 1 & 1 & 0 & 0 \\ \hline 0 & 1 & 1 & 1 & 0 \\ \hline 0 & 0 & 1 & 1 & 1 \\ \hline 0 & 0 & 1 & 1 & 0 \\ \hline 0 & 1 & 1 & 0 & 0 \\ \hline \end{array}
 \end{array}
 \quad *
 \quad
 \begin{array}{c}
 K \\
 \begin{array}{|c|c|c|} \hline 1 & 0 & 1 \\ \hline 0 & 1 & 0 \\ \hline 1 & 0 & 1 \\ \hline \end{array}
 \end{array}
 \quad
 = 1+0+1+0+1+0+0+0+1=4
 \quad
 \begin{array}{c}
 S \\
 \begin{array}{|c|c|c|} \hline 4 & & \\ \hline & & \\ \hline & & \\ \hline \end{array}
 \end{array}$$

$$\begin{array}{c}
 X \\
 \begin{array}{|c|c|c|c|c|} \hline 1 & 1 & 1 & 0 & 0 \\ \hline 0 & 1 & 1 & 1 & 0 \\ \hline 0 & 0 & 1 & 1 & 1 \\ \hline 0 & 0 & 1 & 1 & 0 \\ \hline 0 & 1 & 1 & 0 & 0 \\ \hline \end{array}
 \end{array}
 \quad *
 \quad
 \begin{array}{c}
 K \\
 \begin{array}{|c|c|c|} \hline 1 & 0 & 1 \\ \hline 0 & 1 & 0 \\ \hline 1 & 0 & 1 \\ \hline \end{array}
 \end{array}
 \quad
 = 1+0+0+0+1+0+0+0+1=3
 \quad
 \begin{array}{c}
 S \\
 \begin{array}{|c|c|c|} \hline 4 & 3 & \\ \hline & & \\ \hline & & \\ \hline \end{array}
 \end{array}$$

$$\begin{array}{c}
 X \\
 \begin{array}{|c|c|c|c|c|} \hline 1 & 1 & 1 & 0 & 0 \\ \hline 0 & 1 & 1 & 1 & 0 \\ \hline 0 & 0 & 1 & 1 & 1 \\ \hline 0 & 0 & 1 & 1 & 0 \\ \hline 0 & 1 & 1 & 0 & 0 \\ \hline \end{array}
 \end{array}
 \quad *
 \quad
 \begin{array}{c}
 K \\
 \begin{array}{|c|c|c|} \hline 1 & 0 & 1 \\ \hline 0 & 1 & 0 \\ \hline 1 & 0 & 1 \\ \hline \end{array}
 \end{array}
 \quad
 = 1+0+0+0+1+0+1+0+1=4
 \quad
 \begin{array}{c}
 S \\
 \begin{array}{|c|c|c|} \hline 4 & 3 & 4 \\ \hline & & \\ \hline & & \\ \hline \end{array}
 \end{array}$$

Ahora observemos el resultado de la operación de convolución completa por medio de la función `convolve2d()` del submódulo signal de la librería Scipy:

```
X = np.array([[1,1,1,0,0], [0,1,1,1,0], [0,0,1,1,1], [0,0,1,1,0], [0,1,1,0,0]])
K = np.array([[1,0,1],[0,1,0],[1,0,1]])
from scipy.signal import convolve2d
convolve2d(I,K, mode='valid')
```

```
array([[4, 3, 4],
       [2, 4, 3],
       [2, 3, 4]])
```

Este tipo de convolución es conocida como *valid* y su objetivo es capturar algunas de las características de la imagen similares a la del kernel. Un kernel resaltará las características más importantes de una imagen, y por lo general se acostumbran a definir en una capa de convolución un total de 32 filtros e ir aumentándolo por ejemplo a 64 en la siguiente capa y así sucesivamente, estableciendo números cada vez más grandes para lograr aprender rasgos más complejos del mapa de características dejado por la capa anterior.

Siguiendo con el ejemplo anterior, vimos cómo el kernel se fue desplazando de a un pixel (*stride=1*). Este desplazamiento llamado también *paso* o *zancada* es un hiperparámetro que se puede ajustar con valores más grandes. Sin embargo, un problema que puede presentarse en estas circunstancias es el de hacer coincidir o encajar las dimensiones de la entrada con la dimensión del kernel. Una solución a esta problemática es la operación conocida como *zero padding* (rellenar con ceros), que consiste en llenar la entrada X con un número apropiado de filas y columnas de ceros a cada lado de la imagen a fin de lograr que la imagen de salida tenga la misma dimensión de la imagen de entrada. La convolución que consta de padding es conocida como “*same*” y se puede aplicar en Scipy simplemente ajustando el parámetro `mode='same'`.

Para entender un poco más el concepto de padding considere lo siguiente: suponga que tiene una imagen de entrada de 5x5 y un kernel de 3x3 y desea que la imagen de salida tenga el mismo tamaño que la de entrada. Para solventar este problema basta con agregar una columna a la derecha, una columna a la izquierda, una fila en la parte superior y otra en la parte inferior.

<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>0</td></tr><tr><td>0</td><td>4</td><td>5</td><td>6</td><td>0</td></tr><tr><td>0</td><td>7</td><td>8</td><td>9</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	1	2	3	0	0	4	5	6	0	0	7	8	9	0	0	0	0	0	0	<table border="1"><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>0</td><td>4</td><td>5</td></tr></table> (a)	0	0	0	0	1	2	0	4	5	<table border="1"><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr></table> (b)	0	0	0	1	2	3	4	5	6	<table border="1"><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>2</td><td>3</td><td>0</td></tr><tr><td>5</td><td>6</td><td>0</td></tr></table> (c)	0	0	0	2	3	0	5	6	0	<table border="1"><tr><td>a</td><td>b</td><td>c</td></tr><tr><td>d</td><td>e</td><td>f</td></tr><tr><td>g</td><td>h</td><td>i</td></tr></table>	a	b	c	d	e	f	g	h	i
0	0	0	0	0																																																													
0	1	2	3	0																																																													
0	4	5	6	0																																																													
0	7	8	9	0																																																													
0	0	0	0	0																																																													
0	0	0																																																															
0	1	2																																																															
0	4	5																																																															
0	0	0																																																															
1	2	3																																																															
4	5	6																																																															
0	0	0																																																															
2	3	0																																																															
5	6	0																																																															
a	b	c																																																															
d	e	f																																																															
g	h	i																																																															
	<table border="1"><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>0</td><td>4</td><td>5</td></tr><tr><td>0</td><td>7</td><td>8</td></tr></table> (d)	0	1	2	0	4	5	0	7	8	<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td>9</td></tr></table> (e)	1	2	3	4	5	6	7	8	9	<table border="1"><tr><td>2</td><td>3</td><td>0</td></tr><tr><td>5</td><td>6</td><td>0</td></tr><tr><td>8</td><td>9</td><td>0</td></tr></table> (f)	2	3	0	5	6	0	8	9	0																																			
0	1	2																																																															
0	4	5																																																															
0	7	8																																																															
1	2	3																																																															
4	5	6																																																															
7	8	9																																																															
2	3	0																																																															
5	6	0																																																															
8	9	0																																																															
	<table border="1"><tr><td>0</td><td>4</td><td>5</td></tr><tr><td>0</td><td>7</td><td>8</td></tr><tr><td>0</td><td>0</td><td>0</td></tr></table> (g)	0	4	5	0	7	8	0	0	0	<table border="1"><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td>9</td></tr><tr><td>0</td><td>0</td><td>0</td></tr></table> (h)	4	5	6	7	8	9	0	0	0	<table border="1"><tr><td>5</td><td>6</td><td>0</td></tr><tr><td>8</td><td>9</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr></table> (i)	5	6	0	8	9	0	0	0	0																																			
0	4	5																																																															
0	7	8																																																															
0	0	0																																																															
4	5	6																																																															
7	8	9																																																															
0	0	0																																																															
5	6	0																																																															
8	9	0																																																															
0	0	0																																																															

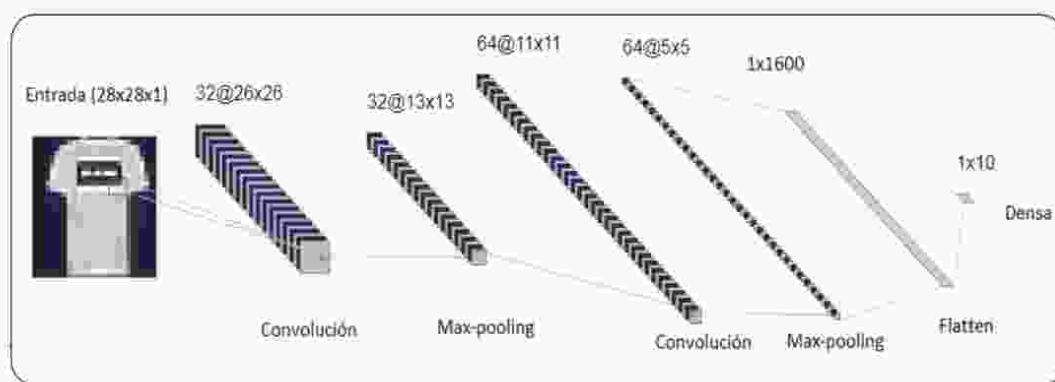
12.1.2. Agrupación

La agrupación o *pooling* como también se le conoce, busca básicamente una reducción en el volumen de los datos, esto trae consigo menor carga computacional, menor uso de memoria y por supuesto menor cantidad de parámetros por entrenar. La capa de *pooling* funciona de manera parecida a la capa convolucional en la medida que hace un recorrido a la imagen aplicando una operación a cada sección de la misma. Su objetivo es agrupar las características más relevantes, para ello hace uso de dos funciones: *max-pooling* y *average-pooling*. La primera obtiene el valor máximo de la sección analizada, mientras que la segunda el valor promedio de la misma. Esta operación frecuentemente se utiliza con un tamaño de 2x2 y un stride=2, como se puede apreciar en el siguiente ejemplo donde se aplica max-pooling y average-pooling sobre una entrada de 4x4.

Entrada				Max-pooling	Average-pooling
12	20	30	0		
8	12	2	2		
34	70	37	4		
112	100	25	12	20	8
				112	37
				79	20

Como ya habíamos comentado, las CNNs se basan en una secuencia de capas de neuronas. Parten de una entrada (imagen) en forma de tensor en 3D (matriz tridimensional) con la forma (ancho, alto, profundidad). Si, por ejemplo, tenemos una imagen como la de dígitos manuscritos este tensor sería de 28x28x1, donde 28x28 son las dimensiones de la imagen (ancho y alto) y 1 corresponde a la profundidad (canal de color), el 1 se emplea para imágenes en escala de grises,

mientras que el 3 para imágenes en RGB. Luego como vemos en la imagen hay una alternancia entre capas convolucionales y capas de agrupación. Cada capa de convolución recibe un mapa de características como entrada y devuelve como salida otro mapa de características de forma (ancho, alto, profundidad), determinado por la dimensión del filtro que en el ejemplo de la imagen es de 3x3. Por esto vemos, que la primera capa de convolución devuelve un tensor 26x26x32, donde 32 es el número de filtros que se aplican, y 26x26 son la cantidad de píxeles de la primera capa oculta definidos después de desplazar la ventana de 3x3 sobre el mapa de características de entrada, antes de chocar con la parte derecha e inferior de la misma. Seguidamente la operación de max-pooling por su parte, selecciona la información más significativa y descarta la menos relevante de la entrada recibida, por ello vemos que tras esta operación el mapa de características recibido como entrada se reduce, el nivel de la reducción dependerá de las dimensiones del max-pooling, en este caso se redujo a la mitad por tratarse un max-pooling de tamaño 2x2 y con strides de 2 en 2 píxeles. Este max-pooling nos arroja un tensor de 13x13x32 y además note que este no afecta la cantidad de filtros de la capa. La siguiente capa de convolución devuelve un tensor de 11x11x64. Siendo el 64 la cantidad de filtros aplicados y 11x11 son la cantidad de neuronas de esta nueva capa oculta obtenidas de pasar la ventana de 3x3 sobre el mapa de características de 13x13, antes de chocar con su borde derecho e inferior. La otra capa de Max-pooling arroja un tensor de 5x5x64. Después de la alternancia entre convolución y max-pooling la red dispone una capa Flatten que aplana el tensor en 3D recibido como entrada de la capa anterior a uno de 1D (array de una dimensión) de 1x1600. Por último, se aplica una capa densa de 1x10 para realizar la predicción basada en una distribución de probabilidades por medio de una función de activación softmax ideal para clasificación multiclase.



12.2 CNNs en Keras

En este ejemplo crearemos una red neuronal convolucional paso a paso sobre el conjunto de datos Fashion Mnist usando el API Keras y al final observaremos si dicha red ayuda a mejorar nuestros resultados de clasificación obtenidos en la sección pasada.

Como siempre, primeramente, invocamos las utilidades necesarias y cargamos el conjunto de datos.

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
from keras.datasets import fashion_mnist
(X_ent, y_ent), (X_pru, y_pru) = fashion_mnist.load_data()
```

A continuación, realizamos el preprocesamiento de los datos. Aquí lo nuevo es el uso de la función reshape() para agregarle la dimensión canal de color a la entrada, ya que como habíamos adelantado en el subapartado anterior se debe pasar a la primera capa de la red un tensor en 3D con la forma (ancho, alto, canal de color):

```
# reshape
X_ent = X_ent.reshape((60000, 28, 28, 1))
X_pru = X_pru.reshape((10000, 28, 28, 1))
X_ent = X_ent.astype('float32')
X_pru = X_pru.astype('float32')
# normalizamos
X_ent, X_pru = X_ent / 255.0, X_pru / 255.0
y_ent = tf.keras.utils.to_categorical(y_ent, 10)
y_pru = tf.keras.utils.to_categorical(y_pru, 10)
```

Ahora creamos la CNN con dos capas convolucionales (Conv2D) con filtros de 3x3, función de activación RELU (activation='relu') y con 32 y 64 filtros cada una. La primera capa recibe el tensor de 28x28x1 (input_shape=(28, 28, 1)), correspondiente a la forma de la entrada que recibirá la red. Además por cada capa de convolución tenemos una capa de MaxPooling2D de tamaño 2x2 (pool_size=(2, 2)) y salto de 2 en 2 (strides=(2, 2)). Más adelante hay una capa de Flatten que pasa el tensor recibido de la capa anterior de 3D a 1D y otra densamente conectada (Dense) con 10 unidades y una función de activación softmax.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(10, activation='softmax'))
```

Mostremos un resumen de nuestra red con el método `summary()` que al final nos dice que se entrenará un total de 34.826 parámetros:

```
print(model.summary())
```

```
Model: "sequential"
-----  

Layer (type)          Output Shape       Param #
conv2d (Conv2D)        (None, 26, 26, 32)    320
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)    0
conv2d_1 (Conv2D)       (None, 11, 11, 64)     18496
max_pooling2d_1 (MaxPooling2D) (None, 5, 5, 64)     0
flatten (Flatten)      (None, 1600)          0
dense (Dense)          (None, 10)            16010
-----  

Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
-----  

None
```

En este punto es preciso hablar un poco de los parámetros de cada una de las capas de nuestra red:

- Capa 1 (conv2d): $(3 * 3 + 1) * 32 = 320$, donde 3*3 es el tamaño del filtro, 1 es el sesgo y 32 es la cantidad de filtros.
- Capa 2(max_pooling2d): No genera parámetros, puesto que es una operación matemática que obtiene valores máximos de una matriz de entrada.
- Capa 3(conv2d_1): $((3 * 3 * 32) + 1) * 64 = 18.496$.
- Capa 4(max_pooling2d_1): No genera parámetros
- Capa 5 (flatten): No genera parámetros. Solo cambia la forma de tensor de entrada.
- Capa 6 (dense): $1.600 * 10 + 10 = 16.010$, donde 1.600 * 10 son los pesos que resultan de multiplicar la cantidad de datos de entrada por la cantidad de unidades (10) más 10 unidades de sesgo.

Compilamos el modelo creado y lo entrenamos de igual forma a como hicimos con la red densamente conectada, pero en este caso evidenciaremos una mayor demora en del proceso de entrenamiento:

```
# compilamos el modelo
model.compile(loss="categorical_crossentropy", optimizer='adam',
metrics=["accuracy"])
history = model.fit(X_ent, y_ent, batch_size=64, epochs=10, verbose=1,
validation_split=0.1)
```

```
Epoch 1/10
844/844 [=====] - 29s 35ms/step - loss: 0.5469 - accuracy: 0.8052 - val_loss: 0.3931 - val_accuracy: 0.8625
Epoch 2/10
844/844 [=====] - 29s 35ms/step - loss: 0.3583 - accuracy: 0.8735 - val_loss: 0.3582 - val_accuracy: 0.8707
Epoch 3/10
844/844 [=====] - 30s 36ms/step - loss: 0.3161 - accuracy: 0.8861 - val_loss: 0.3159 - val_accuracy: 0.8838
Epoch 4/10
844/844 [=====] - 29s 35ms/step - loss: 0.2861 - accuracy: 0.8974 - val_loss: 0.2889 - val_accuracy: 0.8947
Epoch 5/10
844/844 [=====] - 30s 35ms/step - loss: 0.2656 - accuracy: 0.9044 - val_loss: 0.2808 - val_accuracy: 0.8972
Epoch 6/10
844/844 [=====] - 29s 35ms/step - loss: 0.2467 - accuracy: 0.9107 - val_loss: 0.2646 - val_accuracy: 0.9047
Epoch 7/10
844/844 [=====] - 30s 35ms/step - loss: 0.2321 - accuracy: 0.9163 - val_loss: 0.2664 - val_accuracy: 0.9057
Epoch 8/10
844/844 [=====] - 30s 36ms/step - loss: 0.2188 - accuracy: 0.9217 - val_loss: 0.2633 - val_accuracy: 0.9053
Epoch 9/10
844/844 [=====] - 30s 36ms/step - loss: 0.2059 - accuracy: 0.9248 - val_loss: 0.2592 - val_accuracy: 0.9092
Epoch 10/10
844/844 [=====] - 30s 36ms/step - loss: 0.1973 - accuracy: 0.9284 - val_loss: 0.2491 - val_accuracy: 0.9113
```

Obtenemos las puntuaciones de pérdida y exactitud por medio de la función evaluate() y notamos que nuestro modelo ha tenido una mejora, aunque mínima con respecto a la red densamente conectada.

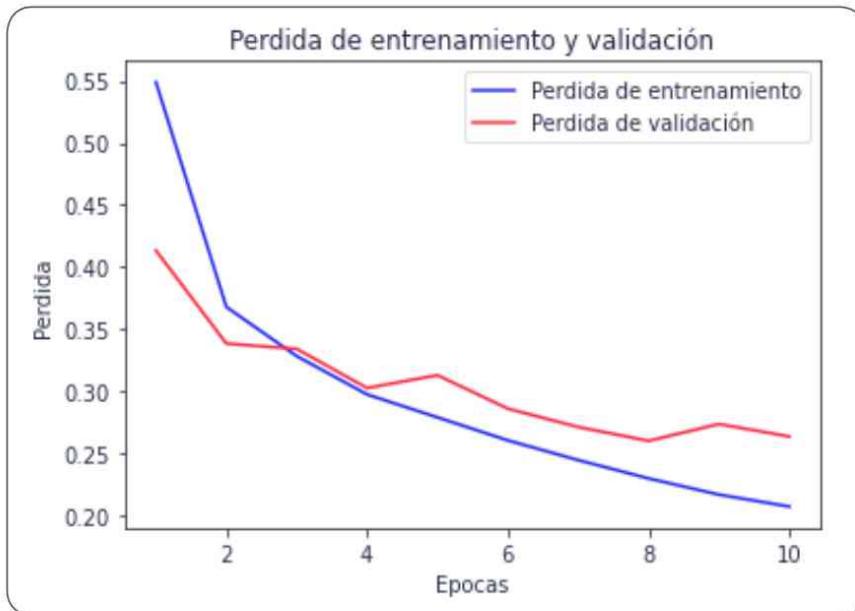
```
puntuaciones = model.evaluate(X_pru, y_pru, verbose=1)
print("\nPérdida: {:.2f}".format(puntuaciones[0]))
print("Exactitud: {:.2f}".format(puntuaciones[1]))
```

Pérdida: 0.28

Exactitud: 0.90

Como ejercicio final podemos dibujar gráficas para observar el comportamiento de los valores de pérdida para el entrenamiento y la validación a medida que pasan las épocas.

```
# graficando la perdida de entrenamiento y validación
import matplotlib.pyplot as plt
history_dict = history.history
loss_values = history_dict['loss']
acc = history_dict['accuracy']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, loss_values, 'b', label='Perdida de entrenamiento')
plt.plot(epochs, val_loss_values, 'r', label='Perdida de validación')
plt.title('Perdida de entrenamiento y validación')
plt.xlabel('Epocas')
plt.ylabel('Perdida')
plt.legend()
plt.show()
```

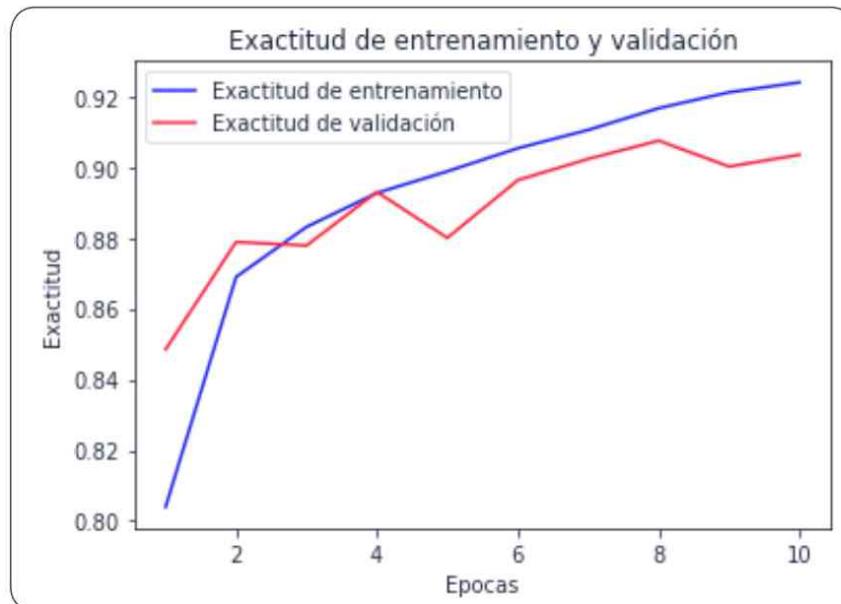


También graficamos la exactitud en el entrenamiento y la validación.

```
plt.clf()
acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']
plt.plot(epochs, acc, 'b', label='Exactitud de entrenamiento')
plt.plot(epochs, val_acc, 'r', label='Exactitud de validación')
plt.title('Exactitud de entrenamiento y validación')
plt.xlabel('Epochas')
plt.ylabel('Exactitud')
plt.legend()
plt.show()
```

Vemos que la exactitud con los datos de entrenamiento y validación aumenta con cada época, pero aproximadamente desde la época 3 la curva de entrenamiento tiende a crecer mientras que la de validación decrece esto es un síntoma de sobreajuste del modelo.

Para afrontar el problema del sobreajuste existen diferentes técnicas como la regularización, estudiada en la primera parte del libro, pero también existen otras como el dropout que estudiaremos en la siguiente sección.



12.3 Regularización y dropout

Regularización

Con respecto a la regularización existen dos métodos que deben resultar familiares, y son: l1 y l2 los cuales básicamente buscan penalizar los pesos sin incluir los sesgos del modelo, gracias a la adición de un término a la función de costo. Esta penalización sobre los pesos hace que estos se restrinjan a valores muy pequeños haciendo pequeños también los valores de la entrada de red (Z), lo cual repercute en el logro de un modelo menos complejo.

Adicionar regularización en Keras es sencillo, basta solo con agregar el parámetro `kernel_regularizer` a la capa donde queremos que se aplique. Los valores posibles son: `keras.regularizers.l1(λ)`, `keras.regularizers.l2(λ)`, `keras.regularizers.l1_l2(λ_1, λ_2)`. Este último aplica ambas regularizaciones l1 y l2. En donde λ es el término de penalización que debe ajustarse debidamente, porque en caso de escogerse mal puede provocar modelos subajustados.

Considere el siguiente experimento donde aplicaremos la regularización l2 con un término λ de 0.001.

```

from keras import regularizers
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1),
kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu', kernel_
regularizer=regularizers.l2(0.001)))
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(10, activation='softmax'))

model.compile(loss="categorical_crossentropy", optimizer='adam',
metrics=["accuracy"])
history = model.fit(X_ent, y_ent, batch_size=64, epochs=10, verbose=1,
validation_split=0.1)

puntuaciones = model.evaluate(X_pru, y_pru, verbose=1)
print("\nPerdida: {:.2f}".format(puntuaciones[0]))
print("Exactitud: {:.2f}".format(puntuaciones[1]))

```

Perdida: 0.35

Exactitud: 0.89

Al parecer el resultado logrado tras aplicar la regularización l2 no fue el mejor, así que probaremos con otra técnica ampliamente usada llamada dropout.

Dropout

El dropout es uno de los métodos insignia en cuanto a regularización de redes neuronales se refiere, y consiste en anular algunas neuronas en cada etapa del entrenamiento con lo cual se reduce la complejidad del modelo, sin embargo, estas neuronas vuelven a ser activadas para ser usadas en la validación. Las neuronas a quitar se controlan con el hiperparámetro p (tasa de dropout) que recibe típicamente un valor entre 0.2 y 0.5. Por ejemplo, si tenemos 10 neuronas y definimos $p=0.2$, se quitarán 2 unidades neuronales.

Considere el siguiente fragmento de código:

```

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(layers.Dropout(0.2))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(layers.Dropout(0.2))
model.add(layers.Flatten())
model.add(layers.Dense(10, activation='softmax'))

model.compile(loss="categorical_crossentropy", optimizer='adam',
metrics=["accuracy"])
history = model.fit(X_ent, y_ent, batch_size=128, epochs=10, verbose=1,
validation_split=0.1)

puntuaciones = model.evaluate(X_pru, y_pru, verbose=1)
print("\nPerdida: {:.2f}".format(puntuaciones[0]))
print("Exactitud: {:.2f}".format(puntuaciones[1]))

```

Perdida: 0.29

Exactitud: 0.89

En este ejemplo además de dropout hemos usado normalización por lotes (BatchNormalization), técnica usada para normalizar los datos de entrada de una capa para que su salida de activación tenga media 0 y desviación estándar 1, esto ayuda a aumentar la rapidez de aprendizaje de las neuronas y mejorar su precisión. Sin embargo, observaremos que nuestros resultados siguen sin mejorar. En todo caso, una buena alternativa ante esta situación, es agregar más capas a la red y modificar los hiperparámetros para tratar de elevar aún más la exactitud del modelo, hecho que sin duda acarrearía un mayor gasto en materia de recursos de máquina y tiempo. Lo anterior lo podemos comprobar si ponemos en marcha el siguiente código:

```

model = models.Sequential()

model.add(layers.Conv2D(32, (3,3), padding='same',
input_shape=X_ent.shape[1:], activation='relu'))
model.add(layers.BatchNormalization())

```

```
model.add(layers.Conv2D(32, (3,3), padding='same', activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D(pool_size=(2,2)))
model.add(layers.Dropout(0.2))

model.add(layers.Conv2D(64, (3,3), padding='same', activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Conv2D(64, (3,3), padding='same', activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D(pool_size=(2,2)))
model.add(layers.Dropout(0.3))

model.add(layers.Conv2D(128, (3,3), padding='same', activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Conv2D(128, (3,3), padding='same', activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D(pool_size=(2,2)))
model.add(layers.Dropout(0.4))

model.add(layers.Flatten())
model.add(layers.Dense(10, activation='softmax'))

model.compile(loss="categorical_crossentropy", optimizer='adam',
metrics=['accuracy'])
history = model.fit(X_ent, y_ent, batch_size=128, epochs=10, verbose=1,
validation_split=0.1)

puntuaciones = model.evaluate(X_pru, y_pru, verbose=1)
print("\nPerdida: {:.2f}".format(puntuaciones[0]))
print("Exactitud: {:.2f}".format(puntuaciones[1]))
```

Perdida: 0.21

Exactitud: 0.93

De cualquier manera, este resultado nos demuestra que las CNNs son mucho más potentes que las redes neuronales densamente conectadas en lo que concierne a la clasificación de imágenes.

En el próximo capítulo exploraremos otras técnicas interesantes con las que buscaremos alcanzar mayor calidad en nuestros modelos de redes neuronales.

CAPÍTULO 13

Aumento de datos y transferencia de aprendizaje

Temas

- 13.1 Generador de datos de imágenes
- 13.2 Aumento de datos (data augmentation)
- 13.3 Transferencia de aprendizaje (*transfer learning*)

En este capítulo exploraremos una técnica muy popular enfocada en generar más datos para nuestras redes neuronales a partir de los existentes, muy útil sobre todo para aquellos casos en los que contemos con pocos datos para entrenar nuestros algoritmos, lo cual en cierta medida nos ayudará también a combatir el sobreajuste.

13.1 Generador de datos de imágenes

Elejemploquemostaremosenestaocasiónharáuso de imágenes reales de 300x300 pixeles relacionadas con un dataset de caballos y personas con el cual crearemos un clasificador binario apoyado en una CNN. Las imágenes las podrá encontrar abriendo este enlace: https://drive.google.com/file/d/1ImBD5jmToC8Aix_suMb_ autmxGHdOMXN/view?usp=sharing



Para poder pasar imágenes a nuestra red debemos hacerles un trabajo de preprocesado, para ello existe una clase llamada `ImageDataGenerator` perteneciente al submódulo `keras.preprocessing.image`. Esta clase permite convertir de manera eficiente imágenes del disco en lotes de tensores preprocesados.

De vuelta con el ejemplo, observemos la primera parte del código. En esta importamos las librerías, creamos la carpeta entrenamiento donde colocaremos el 80% del total de imágenes y la carpeta validación donde copiaremos el 20% restante.

```
import os
import matplotlib.pyplot as plt
import tensorflow as tf
from keras.preprocessing import image
import numpy as np
import shutil

carpeta_base = "horse_human"
dir_ent = os.path.join(carpeta_base, 'entrenamiento')
os.mkdir(dir_ent)
dir_val = os.path.join(carpeta_base, 'validacion')
os.mkdir(dir_val)
dir_ent_cab = os.path.join(dir_ent, 'caballos')
os.mkdir(dir_ent_cab)
dir_ent_hum = os.path.join(dir_ent, 'humanos')
os.mkdir(dir_ent_hum)
dir_val_cab = os.path.join(dir_val, 'caballos')
os.mkdir(dir_val_cab)
dir_val_hum = os.path.join(dir_val, 'humanos')
os.mkdir(dir_val_hum)

num_ima_cab = 500
num_ima_hum = 527
porc_separacion = 0.2
# caballos
num_ima_pru_cab = np.round(porc_separacion * num_ima_cab)
num_ima_ent_cab = np.round(num_ima_cab - num_ima_pru_cab)
# humanos
num_ima_pru_hum = np.round(porc_separacion * num_ima_hum)
num_ima_ent_hum = np.round(num_ima_hum - num_ima_pru_hum)

# img caballos
fnames = ['horse {}'.format(i+1) for i in range(int(num_ima_ent_cab))]
for fname in fnames:
    src = os.path.join(carpeta_base + "/horses", fname)
    dst = os.path.join(dir_ent_cab, fname)
    shutil.copyfile(src, dst)
```

```

fnames = ['horse {}'.format(i+1) for i in range(int(num_ima_ent_cab),
num_ima_cab)]
for fname in fnames:
    src = os.path.join(carpeta_base+"/horses", fname)
    dst = os.path.join(dir_val_cab, fname)
    shutil.copyfile(src, dst)

# img humanos
fnames = ['human {}'.format(i+1) for i in range(int(num_ima_ent_hum))]
for fname in fnames:
    src = os.path.join(carpeta_base+"/humans", fname)
    dst = os.path.join(dir_ent_hum, fname)
    shutil.copyfile(src, dst)

fnames = ['human {}'.format(i+1) for i in range(int(num_ima_ent_hum),
num_ima_hum)]
for fname in fnames:
    src = os.path.join(carpeta_base+"/humans", fname)
    dst = os.path.join(dir_val_hum, fname)
    shutil.copyfile(src, dst)

```

Ahora mostramos por medio de las siguientes instrucciones como han quedado conformadas las carpetas:

```

print('Total muestras de entrenamiento de caballos:', len(os.listdir(dir_ent_cab)))
print('Total muestras de entrenamiento de humanos:', len(os.listdir(dir_ent_hum)))
print('Total muestras de validación de caballos:', len(os.listdir(dir_val_cab)))
print('Total muestras de validación de humanos:', len(os.listdir(dir_val_hum)))

```

Total muestras de entrenamiento de caballos: 400

Total muestras de entrenamiento de humanos: 422

Total muestras de validación de caballos: 100

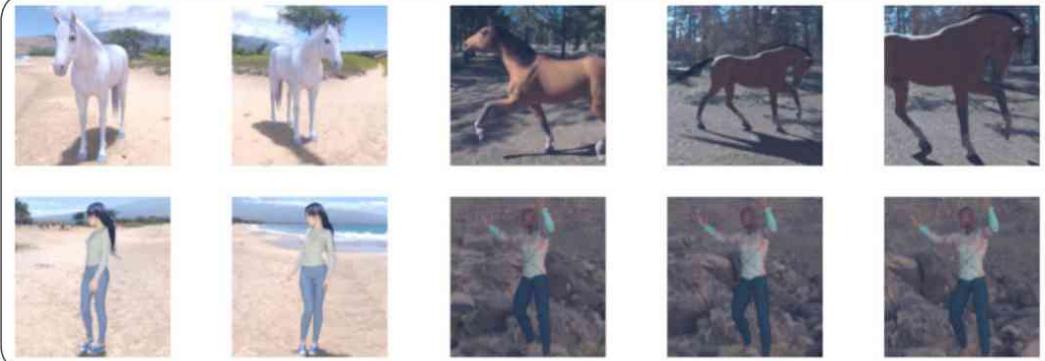
Total muestras de validación de humanos: 105

Mostramos 10 imágenes para analizar su aspecto:

```
# se muestran algunas imágenes
nom_cab_ent = os.listdir(dir_ent_cab)
nom_hum_ent = os.listdir(dir_ent_hum)

plt.figure(figsize=(15,5))
img_caballos=[os.path.join(dir_ent_cab, fname) for fname in nom_cab_ent[:5]]
img_humanos=[os.path.join(dir_ent_hum, fname) for fname in nom_hum_ent[:5]]

for i, img_path in enumerate(img_caballos + img_humanos):
    sp = plt.subplot(2, 5, i + 1)
    sp.axis('Off') # no se muestran los ejes
    img = mpimg.imread(img_path)
    plt.imshow(img)
plt.show()
```



Definimos la arquitectura de la red neuronal de manera similar a como lo hicimos en el apartado anterior y hacemos la compilación ahora usando el optimizador RMSprop con una tasa de aprendizaje de 1e-4 (lr=1e-4):

```
from tensorflow.keras import layers
from tensorflow.keras import models
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(300, 300, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
```

```

model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

from tensorflow.keras import optimizers
model.compile(loss='binary_crossentropy',
    optimizer=optimizers.RMSprop(lr=1e-4),
    metrics=['acc'])

```

El siguiente paso antes de entrenar la red es hacer el preprocessado de las imágenes por medio de la instanciación de dos objetos de la clase `ImageDataGenerator` (Uno para el conjunto de entrenamiento y otro para el de prueba), en este caso le pasamos solo un parámetro al constructor con el fin de pedirle simplemente que reescalaje las imágenes en el rango entre 0 y 1. Además estos objetos llaman al método `flow_from_directory()`, el cual recibe la ruta de la carpeta donde se encuentran las imágenes y devuelve iteradores de directorios (`DirectoryIterator`).

```

# preprocessamiento de datos
from tensorflow.keras.preprocessing.image import ImageDataGenerator
gen_datos_ent = ImageDataGenerator(rescale=1./255)

# entrenamiento
gen_ent = gen_datos_ent.flow_from_directory(
    dir_ent,
    target_size=(300, 300),
    batch_size=32,
    class_mode='binary')
# prueba
gen_datos_pru = ImageDataGenerator(rescale=1./255)
gen_val = gen_datos_pru.flow_from_directory(
    dir_val,
    target_size=(300, 300),
    batch_size=32,
    class_mode='binary')

```

Ahora sí ya podemos entrenar nuestra red, lo cual puede tardar algunos minutos:

```
history=model.fit(gen_ent,steps_per_epoch=10,pochs=10,validation_
data=gen_val,
validation_steps=50,verbose=1)
```

Procedemos a obtenemos la exactitud con los datos de validación:

```
print("Exactitud datos validación: {:.2f}".format(history.history['val_acc'][-1]))
```

Exactitud datos validación: 0.78

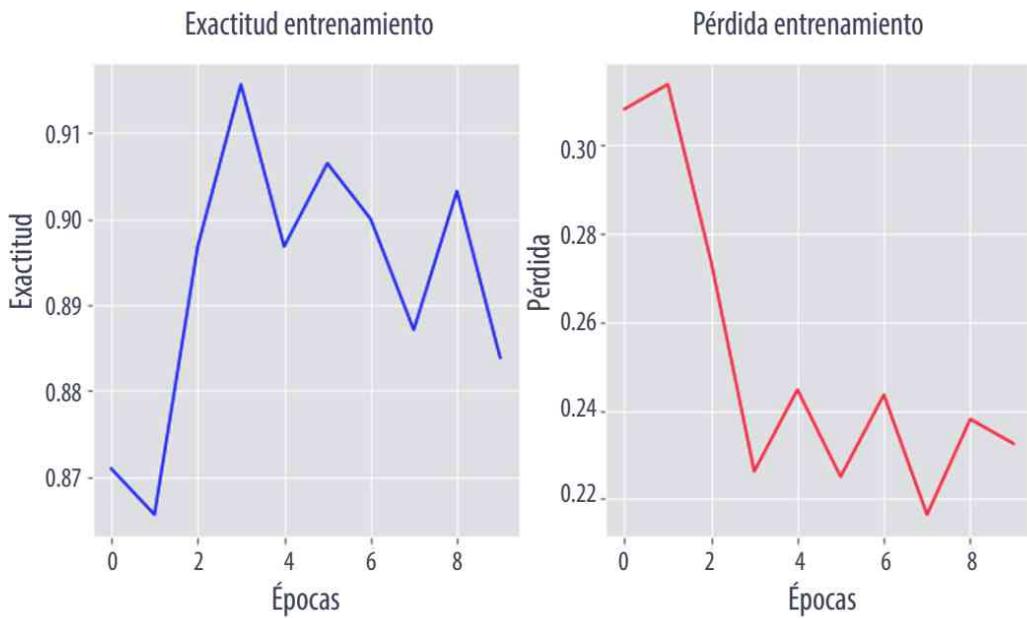
El siguiente método crea un par de gráficas para destacar como fue la exactitud y la perdida durante el entrenamiento.

```
def dibujarGraficas():
    plt.figure(figsize=(10,5))
    n = np.arange(0, 10)
    plt.subplot(121)
    plt.title('Exactitud entrenamiento')
    plt.plot(n, history.history['acc'],'b')
    plt.xlabel("Epochas")
    plt.ylabel("Exactitud")

    plt.subplot(122)
    plt.title('Perdida entrenamiento')
    plt.plot(n, history.history['loss'],'r')
    plt.xlabel("Epochas")
    plt.ylabel("Perdida")
    plt.show()
dibujarGraficas()
```

Lo presentado a continuación nos permite deducir que el modelo está sobreajustado, es decir, presenta *overfitting* dado que la exactitud con datos de entrenamiento supera ampliamente a la obtenida con los datos de validación que fue solo de 0,78.

Para intentar mitigar el sobreajuste del modelo proponemos otra técnica llamada aumento de datos o (en inglés, data augmentation).



13.2 Aumento de datos (*data augmentation*)

Como se comentó en líneas pasadas *data augmentation* es al igual que la regularización y el dropout un arma para combatir el sobreajuste mediante la generación de información nueva, tomando como base la ya existente. El hecho de que el modelo disponga de más datos lo obliga a tener que aprender de un repertorio más amplio lo cual puede repercutir en una mayor generalización.

Para aplicar esta técnica vamos a recurrir a la conocida clase `ImageDataGenerator` pero adicionándole al menos los siguientes parámetros:

- `rotation_range`: Valor en grados (0-180), es un rango para rotaciones aleatorias.
- `width_shift_range`: Fracción del ancho total para trasladar en sentido horizontal.
- `height_shift_range`: Fracción del alto total para trasladar en sentido vertical.
- `shear_range`: Ángulo de corte en sentido anti horario en grados.
- `zoom_range`: Rango para hacer zoom dentro de las imágenes.
- `horizontal_flip` (True/False): Voltea las imágenes de forma aleatoria horizontalmente.
- `fill_mode`: Los puntos por fuera de los límites de la entrada se llenan con el modo pasado como valor.

Nuestro ejemplo con aumento de datos lo definimos de la siguiente manera, teniendo en cuenta de antemano que esta técnica solo se aplica a los datos de entrenamiento.

```
# preprocesamiento de datos
gen_datos_ent = ImageDataGenerator(rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# entrenamiento
gen_ent = gen_datos_ent.flow_from_directory(
    dir_ent,
    target_size=(300, 300),
    batch_size=32,
    class_mode='binary')
```

Mostremos una imagen con algunas transformaciones (rotación, traslación, etc):

```
img_path = img_caballos[1]
img = image.load_img(img_path, target_size=(300, 300))

x = image.img_to_array(img)
x = x.reshape((1,) + x.shape)
gen = gen_datos_ent.flow(x, batch_size=1)

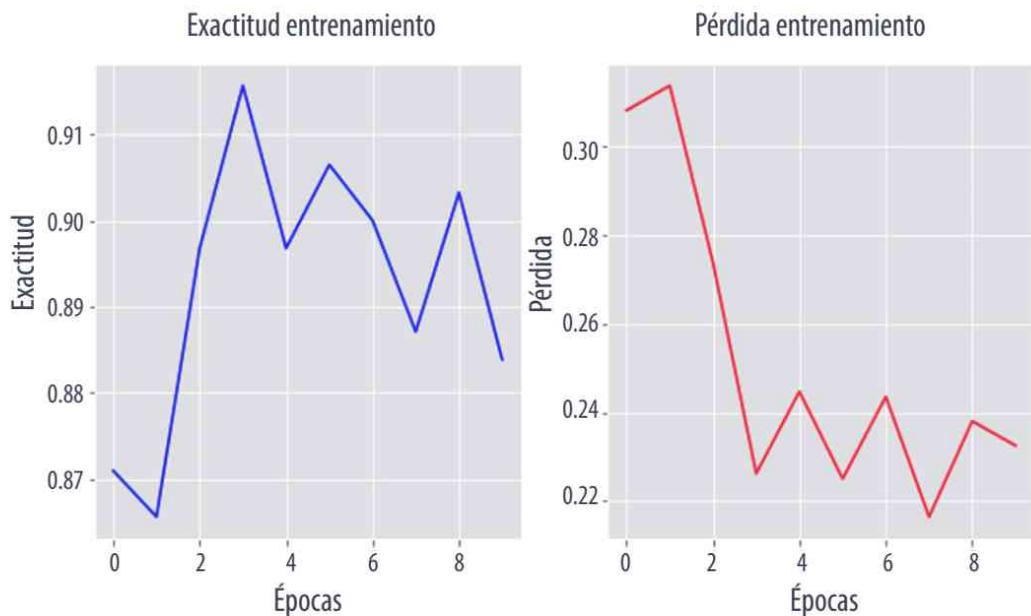
plt.figure(figsize=(20,10))
for i in range(1, 6):
    plt.subplot(1,5,i)
    plt.axis("off")
    plt.imshow(gen.next().squeeze())
    plt.plot()
plt.show()
```



CAP. 13 - AUMENTO DE DATOS Y TRANSFERENCIA DE APRENDIZAJE

Lo demás pasos los dejamos sin modificaciones:

```
history = model.fit(gen_ent, steps_per_epoch=10, epochs=10, validation_data=gen_val,  
                     validation_steps=50, verbose=1)  
  
dibujarGraficas()
```



```
print("Exactitud datos validación: {:.2f}".format(history.history['val_acc'][-1]))
```

Exactitud datos validación: 0.93

Probamos nuestro modelo pasándole la imagen de una persona (prueba_persona.png) para confirmar la exactitud evidenciada numéricamente:

```
path_img = carpeta_base+'/prueba_persona.png'  
img = image.load_img(path_img, target_size=(300, 300))  
x = image.img_to_array(img)  
x = np.expand_dims(x, axis=0)  
  
images = np.vstack([x])  
prediccion = model.predict(images)
```

```
if predicción[0]>0.5:  
    print("Es humano")  
else:  
    print("Es un caballo")
```

Es humano

Hemos conseguido una significativa mejoría, lo cual pone de manifiesto la potencia de esta técnica, que si bien nos ha dado resultado no se debe emplear como única opción ante problemas de sobreajuste, siempre es conveniente explorar otros métodos y estrategias, hasta evidenciar un resultado consistente para que se pueda considerar adecuado y la vez definitivo.

13.3 Transferencia de aprendizaje (*transfer learning*)

La transferencia de aprendizaje es un método que permite que reutilicemos redes preentrenadas y que han surgido en mayor medida de competencias de aprendizaje profundo. Estas redes están conformadas por muchos bloques de capas por lo que requieren del aprendizaje de un gran número de parámetros. Para comodidad de nosotros estas arquitecturas de redes neuronales están disponibles en Keras y las podemos usar en nuestros trabajos simplemente importando sus pesos entrenados, de esta manera ya no tendríamos que entrenar pesos inicializados aleatoriamente, lo cual nos ahorra la fase de entrenamiento, aspecto evidentemente costoso computacionalmente hablando en situaciones reales.

El éxito de la transferencia de aprendizaje parte del hecho de que en las redes neuronales los filtros de las primeras capas aprenden características generales de la entrada como bordes, gotas de color, etc y a medida que la red se hace más profunda otros filtros aprenden aspectos más abstractos en el caso de un avión, sería, por ejemplo, las alas, la cabina hasta llegar a la capa final donde es capaz de identificar el objeto completo. Entonces, la clave es que en caso de tener un problema similar poder utilizar esos parámetros previamente entrenados para no tener que hacer el trabajo desde cero, esto evidentemente permite que podamos ser más ágiles y productivos.

Los modelos preentrenados los podemos usar básicamente para lo siguiente:

- Usarlo completamente para la tarea que deseamos (por ejemplo, usar la arquitectura VGG16 como un clasificador de imágenes con 1.000 clases)

- Usar sus pesos (o parte de ellos) previo al uso de nuestra propia red neuronal. (Los pesos del modelo preentrenado no se van a actualizar en posteriores etapas de entrenamiento (Extracción de características)
- Usar los pesos (o parte de ellos) como parte de nuestra red neuronal, pudiendo actualizar o no alguna de las capas en posteriores etapas de entrenamiento. (Ajuste fino).

Hay varias redes preentrenadas disponibles en keras, entre las más destacadas están: VGG16, VGG19, Xception, RestNet50. Se escapa del alcance del libro explicarlas todas, por lo que solo estudiaremos la VGG16, pero si el lector desea ampliar sus conocimientos en estas arquitecturas de redes neuronales le recomiendo la página oficial de la librería (<https://keras.io/api/applications/>), donde puede consultar todos los modelos disponibles junto con algunos ejemplos.

Para ir entendiendo más las cosas no hay nada mejor que utilizar un ejemplo donde vemos como podemos utilizar la arquitectura VGG16:

```
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input, decode_
predictions

model = VGG16(weights='imagenet', include_top=True)
model.summary()
```

El modelo VGG16 es un modelo bastante profundo en comparación a los que hemos expuesto en los apartados pasados, cuenta con 138.357.544 parámetros entrenables. Para este ejercicio vamos a usar los pesos de Imagenet (weights='imagenet'). Imagenet es una base de datos de imágenes disponible en el siguiente enlace <http://www.image-net.org/>. Siguiendo con el ejemplo, incluiremos también las capas superiores de la red (include_top=True).

Trataremos de probar el modelo creado con la siguiente imagen de un avión.

```
imp_p = 'prueba_avion.jpg'
path = 'horse_human/' + imp_p
img = mpimg.imread(path)
implot = plt.imshow(img)
plt.grid(False)
plt.axis('off')
```



```
img = image.load_img(path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)
preds = model.predict(x)
salida = np.argmax(preds)
print("Índice de la imagen:" + str(salida))
clase = decode_predictions(preds)
clase = clase[0][0]
print("%s: %.2f%%' % (clase[1], clase[2]*100))
```

Índice de la imagen:404

airliner: 76.73%

El modelo nos arroja que hay un 76.73% que se trate de un avión comercial (airliner), el otro 23.27 se divide en las 999 clases restantes.

13.3.1. Extracción de características

La extracción de características (en inglés, *feature extraction*) consiste básicamente en reutilizar la base convolucional, (formada por las capas convolucionales y de pooling) y entrenar un clasificador propio en la parte final que se adapte a la tarea concreta que estemos afrontando. La base convolucional de la red preentrenada ha aprendido patrones genéricos de imágenes que podemos usar en problemas similares, así que supone no tener que volver a entrenar esas capas, por lo cual se hace necesario congelar sus pesos. Esto se hace colocándole a esas capas la propiedad trainable=False.

Pongamos en práctica la extracción de características tomando como datos de entrada el mismo dataset que hemos estado usando en los anteriores ejemplos de este capítulo, pero anticipando que los resultados no serán los mejores ya que ImageNet no tiene un número significativo de figuras de caballos ni de personas. Por lo que lo haremos con los mismos datos solo con la intención nada más de ilustrar la técnica.

```
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras import layers
from tensorflow.keras import models

modelo_base = VGG16(include_top=False, input_shape=(300, 300, 3))
# todas las capas no se volverán a entrenar
for capa in modelo_base.layers:
    capa.trainable = False
```

Creamos un modelo secuencial (Sequential) al cual le adicionamos el modelo preentrenado (model_base) como si de una capa se tratara, además le agregamos las capas finales que hacen parte de nuestro clasificador binario.

```
modeloEC = models.Sequential()
modeloEC.add(modelo_base)
modeloEC.add(layers.Flatten())
modeloEC.add(layers.Dense(512, activation='relu'))
modeloEC.add(layers.Dense(1, activation='softmax'))
modeloEC.summary()
```

Finalmente compilamos, entrenamos y evaluamos nuestro modelo:

```
from tensorflow.keras.optimizers import SGD, Adam
adam = Adam(lr=0.00001)
model.compile(adam, loss='binary_crossentropy', metrics=['accuracy'])

history=model.fit(gen_ent,steps_per_epoch=10,          epochs=10, validation_
data=gen_val,
                    validation_steps=50, verbose=1)
```

```
perdida, exactitud = model.evaluate(gen_val)
print("Exactitud: {:.2f}".format(exactitud))
```

Exactitud: 0.47

13.3.2. Ajuste fino (*Fine tuning*)

La estrategia de ajuste fino por su lado, consiste en que además de entrenar el clasificador final también se puedan entrenar algunas capas de la base convolucional que hasta el momento se mantenían congeladas. Este es un ajuste fino en las capas que representan las características más abstractas del modelo tomado como base.

Siguiendo con el mismo ejemplo, trataremos de afinar algunas capas de nuestro modelo. En este caso experimentaremos congelando todas las capas excepto las últimas 10.

```
modelo_base = VGG16(include_top=False, input_shape=(300, 300, 3))

modelo_base.trainable = True
#Congelar las capas convolucionales excepto las últimas 10
for layer in modelo_base.layers[:-10]:
    layer.trainable = False

for i, layer in enumerate(modelo_base.layers):
    print(i, layer.name, layer.trainable)
```

```
0 input_16 False
1 block1_conv1 False
2 block1_conv2 False
3 block1_pool False
4 block2_conv1 False
5 block2_conv2 False
6 block2_pool False
7 block3_conv1 False
8 block3_conv2 False
9 block3_conv3 True
10 block3_pool True
11 block4_conv1 True
12 block4_conv2 True
```

```

13 block4_conv3 True
14 block4_pool True
15 block5_conv1 True
16 block5_conv2 True
17 block5_conv3 True
18 block5_pool True

```

Definimos un modelo secuencial sin cambios con respecto al mostrado en el subapartado referente a extracción de características. Notando que ahora hay 1.145.408 parámetros no entrenables que corresponden a las capas que permanecieron congeladas.

```

modeloAF = models.Sequential()
modeloAF.add(modelo_base)
modeloAF.add(layers.Flatten())
modeloAF.add(layers.Dense(512, activation='relu'))
modeloAF.add(layers.Dense(1, activation='sigmoid'))
modeloAF.summary()

```

```

Model: "sequential_10"
-----  

Layer (type)          Output Shape       Param #
-----  

vgg16 (Functional)    (None, 9, 9, 512)   14714688  

flatten_17 (Flatten)  (None, 41472)      0  

dense_29 (Dense)      (None, 512)        21234176  

dense_30 (Dense)      (None, 1)          513  

-----  

Total params: 35,949,377  

Trainable params: 34,803,969  

Non-trainable params: 1,145,408

```

Una vez más compilamos, entrenamos y luego obtenemos la exactitud:

```

model.compile(adam, loss='binary_crossentropy', metrics=['accuracy'])
history = model.fit(gen_ent, steps_per_epoch=10, epochs=10, validation_data=gen_val,
                     validation_steps=50, verbose=1)
perdida, exactitud = model.evaluate(gen_val)
print("Exactitud: {:.2f}%".format(exactitud))

```

Exactitud: 0.47

Si bien nuestro modelo ha alcanzado una exactitud insuficiente por las razones ya expuestas, considere este ejemplo como un punto de partida para seguir indagando más acerca de todas estas arquitecturas de redes neuronales preentrenadas disponibles en Keras, porque estamos convencidos que con un conjunto de datos apropiado y una adecuada escogencia de los hiperparámetros y sobre todo de las capas entrenables (no congeladas) es posible alcanzar muy buenos resultados con las técnicas explicadas.

CAPÍTULO 14

Introducción al Procesamiento del lenguaje natural (PLN)

Temas

- 14.1 Palabras embebidas (Word embeddings)
- 14.2 Introducción a Word2vec
- 14.3 Word2vec con librería Gensim

El Procesamiento del Lenguaje Natural (PLN) es una de las ramas de la IA que más ha resonado en los últimos tiempos, como tal trata de una serie de técnicas y modelos que buscan hacer más efectiva y eficiente la comunicación entre los computadores y los seres humanos a partir del lenguaje que estos últimos utilizan para comunicarse.

El procesamiento del lenguaje natural busca ayudar a los computadores a comprender y manipular el lenguaje humano con el fin de cerrar la brecha entre la comunicación humana y el entendimiento de estas máquinas.

14.1 Palabras embebidas (*Word embedding*)

En este capítulo, abordaremos uno de los algoritmos más representativos del PLN, se trata de word2vec, no sin antes hablar de *word embedding*. Cabe resaltar que el modelo word2vec es un tipo de *word embedding* (palabras embebidas) que es una técnica consistente en mapear palabras o frases a vectores de números reales para luego ser analizadas con algoritmos de aprendizaje automático.

Otra forma muy común de embedding es la técnica one hot encoding, vista en un capítulo precedente, con la cual codificamos o representamos una palabra del texto por un vector del tamaño del vocabulario, donde la entrada correspondiente a la palabra es 1 mientras que el resto de las otras entradas es 0. Sin embargo, uno de los inconvenientes de dicha técnica es que puede generar vectores muy gran-

des o dispersos sobre todo cuando el vocabulario de entrada o corpus es amplio, lo cual puede suscitar una pérdida de rendimiento considerable. De igual manera one hot encoding no tiene en cuenta la similitud entre las palabras, aspecto que en algunos casos podría resultar de interés para nosotros. Por ejemplo, mantener una relación entre palabras como perro y gato o mamá y papá, entre otros.

El algoritmo word2vec que estudiaremos en breve, logra hacer esta tarea de mapear palabras a una representación numérica (vector) de una manera más eficiente y considerando que los vectores menos distantes entre sí, representan palabras relacionadas o con un significado parecido. Esto y más convierte a word2vec en uno de los métodos más ampliamente adoptado por quienes trabajan en el área del PLN.

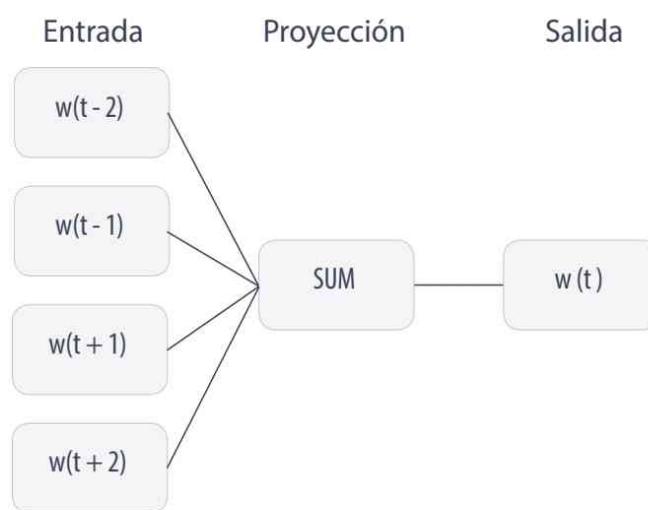
14.2 Introducción a Word2Vec

Word2vec es una técnica para el procesamiento de lenguaje natural desarrollada por investigadores de Google en 2013, básicamente consiste en una red neuronal de dos capas que busca aprender las asociaciones que hay entre las palabras de un gran corpus de texto, para ello convierte cada palabra en un vector en el espacio. Este espacio vectorial es dimensionalmente menor que el generado por técnicas como *one hot encoding*. Además, el espacio de palabras incrustadas (*embeddings*) es también más denso comparado con el de one hot encoding que es altamente esparcido.

Las dos arquitecturas manejadas con word2vec son:

- Bolsa de palabras continuas (CBOW):

Con CBOW (continuous bag of words) el modelo predice la palabra central a partir de un conjunto de palabras circundantes (contexto).



Consideremos la siguiente oración: "Me gusta estudiar aprendizaje automático". Con un tamaño de ventana tam_ven de 3 la sentencia anterior la podemos descomponer en los siguientes conjuntos de pares (contexto, palabra):

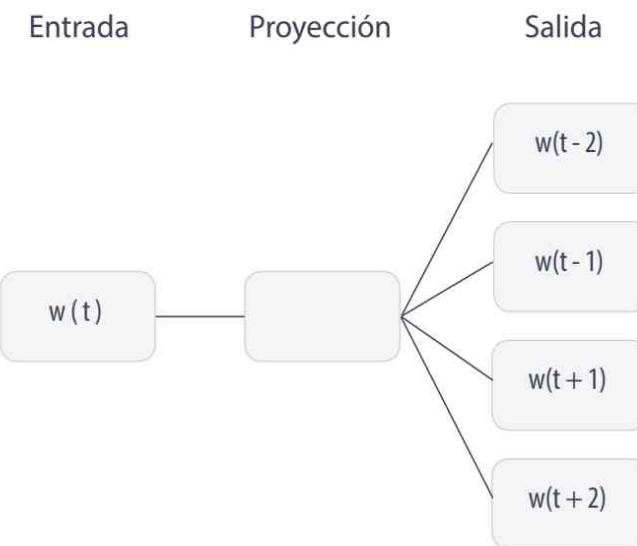
([Me, estudiar], gusta)
 ([gusta, aprendizaje], estudiar)
 ([estudiar, automático], aprendizaje)
 ...

Un modelo definido bajo este esquema deberá ser capaz de predecir la palabra "gusta" a partir del conjunto de palabras ("me", "estudiar") y así sucesivamente.

En general, en CBOW las entradas del modelo son los identificadores de las palabras del contexto. Estos ids alimentan una capa de incrustación (*embedding*) que es inicializada con pesos aleatorios pequeños y que transforman cada id en un vector de tamaño tam_inc . Cada fila del contexto de entrada es transformada en una matriz de tamaño $(2 * tam_ven * N)$ para esa capa. Lo anterior es pasado a una siguiente capa, la cual calcula el promedio de todas las incrustaciones. Este promedio alimenta una capa densa, la cual crea un vector denso de tamaño tam_vec para cada fila. En esta capa, también actúa una función *softmax* que devuelve el valor máximo del vector de salida como una probabilidad, y donde el id con la máxima probabilidad corresponde a la palabra destino.

- Skip-gram:

En esta arquitectura el modelo predice un conjunto de palabras circundantes (contexto) a partir de una palabra central.



Así mismo, Skip-gram se puede emplear para saber si un par de palabras se encuentran relacionadas, lo cual se determina a partir de si estas se encuentran en el mismo contexto.

A modo de ejemplo consideremos nuevamente la oración: "Me gusta estudiar aprendizaje automático". Generando los siguientes pares de entradas positivas (palabras que aparecen juntas) y negativas (palabras que no aparecen juntas y son generadas aleatoriamente).

```
([me, gusta], 1)  
([me, estudiar], 1)  
([gusta, me], 1)  
...
```

```
([gusta, ax], 0)  
([gusta, bga], 0)  
([me, xy], 0)  
...
```

Un modelo entrenado con todas estas entradas es llamado Skip-gram con muestreo negativo (SGNS, por sus siglas en inglés).

Por último, cabe resaltar que la red neuronal del algoritmo en mención se entrena para predecir un 1 si el par de palabras se encuentran relacionadas y 0 en caso contrario.

14.3 Word2vec con librería Gensim

Para poner en práctica nuestros conocimientos acerca de word2vec utilizaremos la librería Gensim. Esta es un kit de herramientas de código abierto bastante utilizada en tareas de programación de lenguaje natural, ideal para problemas como: modelado de temas, indexación de documentos, obtención de similitudes entre palabras, y que además funciona muy bien con grandes colecciones de ejemplos y datos de entrenamiento (corpus de texto).

Vamos a entrenar un modelo utilizando word2vec por medio de la librería gensim, para instalarla usamos el comando: `pip install gensim`. Para el ejemplo tomaremos un archivo con títulos de 100 proyectos extraídos de la web que serán nuestro corpus, pero cabe resaltar que no haremos ningún tipo de trabajo de procesamiento sobre las palabras de entrada.

Inicialmente como de costumbre cargamos las librerías o módulos que vamos a utilizar junto con el método que cargará los datos del archivo en una estructura de datos adecuada:

```
from nltk.tokenize import word_tokenize
from gensim.models import Word2Vec

def cargar_datos(text_file):
    lista = []
    oraciones = []
    with open(text_file, 'r') as f:
        for i, line in enumerate(f):
            lista.append(line)
    sentencias = list([word_tokenize(frase) for frase in lista])
    return sentencias

sentencias = cargar_datos("datos.txt")
```

La lista sentencias ya tiene cargados 100 titulos de proyectos extraídos del archivo datos.txt. Mostremos el elemento ubicado en la posición 1:

```
# mostramos el titulo del primer proyecto
sentencias[1]
```

```
['Desarrollar',
'un',
'sistema',
'que',
'integre',
'distintos',
'algoritmos',
'de',
'procesamiento',
'de',
'imágenes',
'médicas',
'que',
'brinde',
'información',
'relevante',
```

```
'para',
'la',
'toma',
'de',
'decisiones',
'en',
'los',
'diagnósticos',
'y',
'tratamiento',
'del',
'SDRA']
```

Definimos el modelo pasándole entre otros parámetros: sentencias (lista de títulos), tamaño de la ventana=10 y sg=1, este último para indicar que nuestra implementación recurrirá al algoritmo skip-gram, con el fin de obtener un conjunto de palabras circundantes relacionadas a una palabra central:

```
modelo = Word2Vec(
    sentences=sentencias, # datos
    size=300, # tamaño del vector para representar cada palabra
    window=10, # distancia máxima entre la palabra objetivo y las otras
    min_count=5, # mínimo número frecuencia de palabras
    workers=8,# número de hilos de ejecución
    sg=1,# 1 = SKIP , 0 = CBOW
    hs=1,
    negative=0)
modelo.train(sentencias, total_examples=len(sentencias), epochs=20)      # 
entrenamos el modelo con 20 épocas

palabras_incrustadas = modelo[modelo.wv.vocab] # obtenemos los vectores
de palabras
lista_palabras = list(modelo.wv.vocab) # lista con las palabras
```

Una vez realizado el entrenamiento en nuestro caso con 20 épocas ya podemos hallar la similitud entre los vectores (palabras). Para encontrar esta similitud normalmente se utiliza la función *similitud coseno*, matemáticamente expresada como:

$$\cos \theta = \frac{A \cdot B}{\|A\| \cdot \|B\|} = \frac{\sum_{i=1}^n A_i * B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}}$$

Con esto ya podemos probar nuestro modelo para que prediga la similitud entre las palabras "sistema" y "algoritmos", lo cual nos arroja una similitud del 84%:

```
def norma_vectorial(vector):
    return np.sqrt(np.sum([v**2 for v in vector]))

def similitud_coseno(v1, v2):
    return np.dot(v1, v2)/float(norma_vectorial(v1)*norma_vectorial(v2))

print("Similitud: {:.2f} ".format(similitud_coseno(palabras_incrustadas[[lista_
palabras.index("sistema"), :],
palabras_incrustadas[[lista_palabras.index("algoritmos"), :]]]))
```

Similitud: 0.84

De igual manera gensim nos ofrece el método `similarity()` para encontrar la similitud entre palabras:

```
s = modelo.wv.similarity(w1="sistema", w2="algoritmos")
print("similitud {:.2f} ".format(s))
```

Así como otro método llamado `most_similar()` para obtener las 10 palabras más relacionadas con un palabra dada, de mayor a menor probabilidad:

```
palabra = "sistema"
print("Más similar a {0}: ".format(palabra), modelo.wv.most_
similar(positive=palabra))
```

Más similar a sistema: [('de', 0.9480375051498413), ('que', 0.9435662031173706), ('el', 0.9382226467132568), ('', 0.9374876022338867), ('la', 0.9335026741027832), ('un', 0.930368185043335), ('los', 0.9279155731201172), ('y', 0.9270495772361755), ('Desarrollar', 0.9234944581985474), ('en', 0.9188223481178284)]

Con este ejemplo sobre palabras incrustadas con word2vec terminamos este capítulo introductorio al procesamiento del lenguaje natural e iniciaremos el estudio de las redes neuronales recurrentes, no sin antes invitar al lector a seguir explorando la librería Gensim para lograr explotarla mucho más.

<https://dogramcode.com/bloglibros>



CAPÍTULO 15

Redes neuronales recurrentes (RNN)

Temas

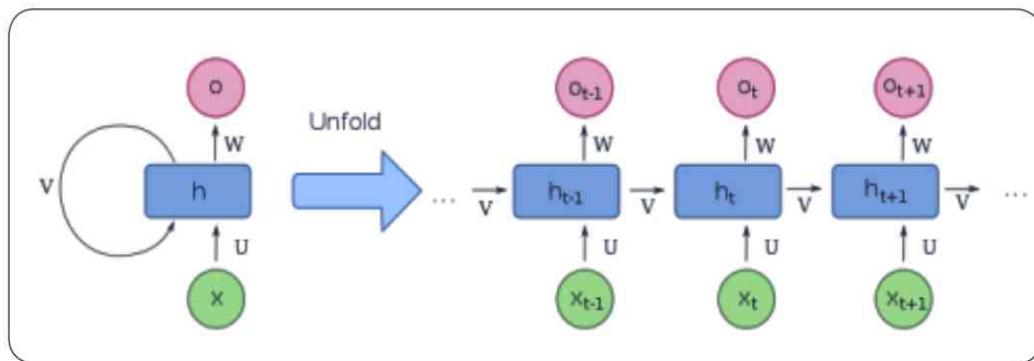
- 15.1 Introducción a las redes neuronales recurrentes
- 15.2 Propagación hacia atrás en el tiempo (BPTT)
- 15.3 LSTM (Memoria larga a corto plazo)

En este nuevo capítulo descubriremos un tipo especial de redes, las llamadas redes neuronales recurrentes (RNN), caracterizadas por incluir realimentación entre sus neuronas, pudiendo recordar las salidas de momentos anteriores y utilizarlas como insumo para alimentar nuevas entradas, esto le otorga a la neurona el factor de temporalidad ausente en los tipos de redes neuronales vistos anteriormente, permitiéndoles tener una especie de memoria para almacenar los datos que recibe.

15.1 Introducción a las redes neuronales recurrentes

El hecho de incluir la dimensión de tiempo hace a las redes neuronales recurrentes muy potentes, permitiéndoles ser utilizadas en la solución de diversos problemas como las series temporales, reconocimiento de voz, traducción de lenguajes, análisis de sentimientos, entre otras situaciones, donde el elemento que se desea predecir depende de elementos anteriores los cuales mantienen una secuencia o son dependientes entre sí.

En la siguiente imagen podemos ver los elementos que hacen parte de una simple RNN. En la izquierda se observa que la neurona recibe una entrada (X), genera una salida (o) y recibe la acción de otro elemento V .



La parte derecha de la imagen anterior nos muestra la misma neurona, pero desplegada en el tiempo, donde se ve cómo va recibiendo en diferentes pasos de tiempo: una entrada y una señal del instante de tiempo anterior, lo cual le ayuda a producir la salida para ese momento dado.

Celdas de memoria

Para poder recordar cierto estado a lo largo de los distintos pasos de tiempo las redes neuronales recurrentes incorporan una memoria interna denominada comúnmente celda de memoria o simplemente celda. El valor del estado en algún punto en el tiempo se determina en función de los valores de los estados ocultos en los instantes previos y el valor de la entrada en el instante de tiempo actual.

$$h_t = \Phi(h_{t-1}, X_t)$$

En donde, en términos castizos h_t sería el estado nuevo o actual, h_{t-1} el estado anterior y X_t un vector con las entradas en un tiempo específico t y Φ es una función de activación.

Tal como sucede con los otros tipos de redes neuronales los parámetros de una RNN son almacenados como matrices de pesos. Siendo en el caso que nos ocupa tres matrices U , V , W , correspondientes a entrada, salida y estados ocultos respectivamente. Notamos de la gráfica, que estas matrices son las mismas que se utilizan cuando la neurona es desplegada en el tiempo, debido fundamentalmente a que las neuronas aplican la misma operación sobre las diferentes entradas que reciben en los diferentes pasos de tiempo. La ecuación anterior expresada en términos de las matrices ponderadas U , V y W multiplicadas por sus correspondientes vectores más la unidad de sesgo quedaría así:

$$h_t = \Phi(U.X_t + V.h_{t-1} + b)$$

La función de activación será tanh (tangente hiperbólica) que se ha demostrado es mucho más estable que otras como ReLu y ayuda a evitar un problema de estas redes llamado desvanecimiento del gradiente.

$$h_t = \tanh (U.X_t + V.h_{t-1} + b)$$

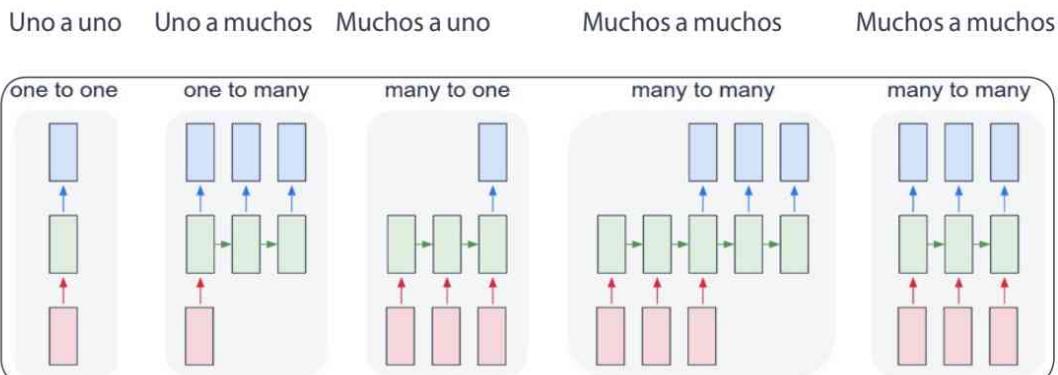
El vector de salida o_t en un tiempo t es el producto de la matriz de pesos W y el estado oculto h_t , con la aplicación de una función *softmax* sobre este producto para dar como resultado un conjunto de probabilidades.

$$o_t = \text{softmax}(W.h_t)$$

Secuencias

Otra diferencia importante entre las RNNs y las redes estudiadas en capítulos anteriores, es que estas últimas solo aceptan un vector de tamaño fijo como entrada (por ejemplo, una imagen) y producen un vector de tamaño fijo como salida (por ejemplo, distribuciones de probabilidad para las clases), mientras que las primeras operan sobre secuencias de vectores. Las topologías manejadas con las RNNs en este sentido son:

- **Uno a uno:** Recibe una entrada de tamaño fijo y entrega una salida de tamaño fijo. Por ejemplo: clasificación de imágenes.
- **Uno a muchos:** De un vector de tamaño fijo se obtiene una secuencia. Por ejemplo: subtítulo una imagen con una oración o palabra.
- **Muchos a uno:** La red recibe una secuencia y devuelve un vector de tamaño fijo. Por ejemplo, de la reseña de una película generar una calificación.
- **Muchos a muchos:** Toma como entrada una secuencia para obtener como salida otra secuencia. Por ejemplo, traducción de una oración en inglés a español.



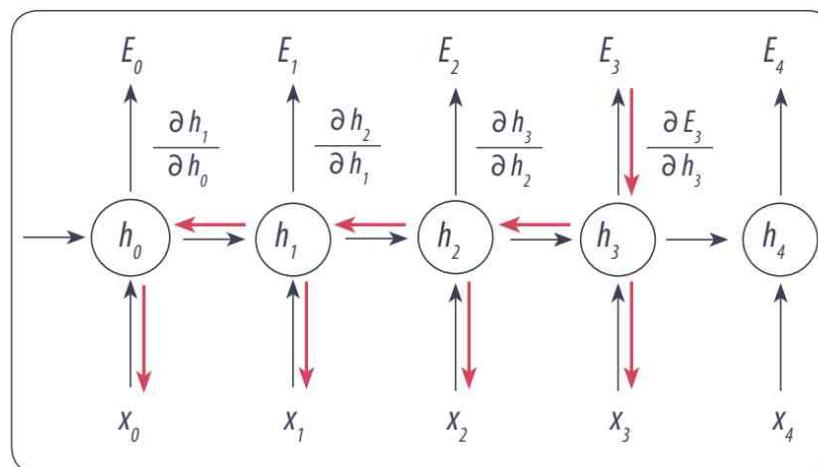
Una red neuronal con estas características puede por ejemplo entrenarse para ser capaz de identificar la palabra siguiente en la frase “El carro pudo...”, que en este caso podría ser *arrancar* y no *avión*, esto es posible dado que las RNN están sometidas a una retroalimentación para aprender elementos secuenciales vistos en momentos previos.

Para entrenar una red de este tipo se debe desplegar o desenrollar a través del tiempo y luego aplicarle retropropagación. Esta técnica se conoce como retropropagación a través del tiempo (Backpropagation through Time - BPTT).

15.2 Propagación a través del tiempo (BPTT)

Al igual que en la propagación hacia atrás explicada al comienzo de esta segunda parte del libro, en las RNN hay una primera propagación hacia adelante en la red desenrollada que produce predicciones \hat{y}_t en un tiempo t que son comparadas con la etiqueta y_t . Esta salida se evalúa utilizando una función de costo C que devuelve un error E . Dentro de esta función de costo se puede ignorar algunas salidas, por ejemplo, en una RNN de secuencia a vector, todas las salidas se ignoran excepto la última.

En la imagen, durante la propagación hacia atrás (mostrada con las líneas con dirección hacia abajo) los gradientes del error con respecto a los pesos U , V y W son calculados en cada paso de tiempo. Finalmente, los parámetros del modelo se actualizan utilizando dichos gradientes calculados. Cabe destacar que estos gradientes fluyen hacia atrás a través de todas las salidas utilizadas por la función de costo, no solo a través de la salida final.



Sin embargo, el BPTT es susceptible a dos problemas propios no solo en RNNs sino también de otros tipos de redes que usan en su entrenamiento gradiente descendente y retropropagación. Recordemos que el gradiente descendente es un método que ayuda a encontrar los valores para los pesos que minimizan una función de costo. El algoritmo en cada iteración de entrenamiento actualiza los pesos de la red neuronal siendo esta proporcional a la derivada parcial de la función de costo con respecto al peso actual.

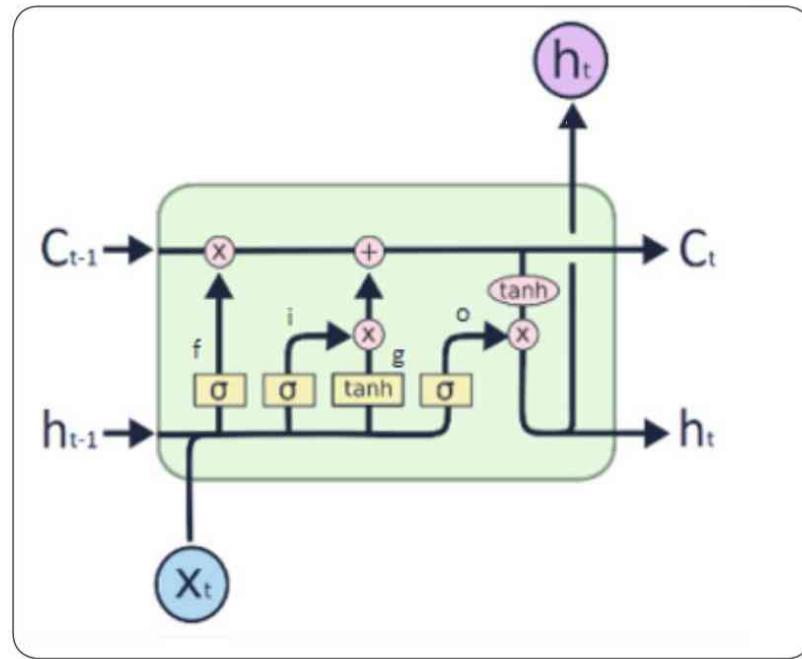
Los problemas a los que nos referimos son: desvanecimiento del gradiente (*vanishing gradient*) y la explosión del gradiente (*exploding gradient*). No entraremos en detalle sobre lo que entraña cada uno de ellos, pero si destacar que el primero se presenta cuando el gradiente se va desvaneciendo con valores muy pequeños impidiendo al peso actualizar su valor, situación que puede provocar que la red deje de entrenarse. Este problema lo puede provocar el uso de funciones de activación como la tangente hiperbólica (\tanh) que entrega gradientes en el rango $(0, 1)$. Entonces con la propagación hacia atrás utilizando la ya conocida regla de la cadena, los productos de los gradientes se hacen cada vez más pequeños a través de múltiples pasos de tiempo. Por el contrario, el segundo problema (*exploding gradient*) se produce cuando los gradientes son mayores a 1 y por tanto los productos también serán grandes.

Para resolver estos dos problemas existen diferentes métodos, nosotros miraremos una arquitectura muy utilizada y es LSTM de la cual hablaremos seguidamente.

15.3 LSTM (Memoria larga a corto plazo)

LSTM (Long short time memory) se refiere a redes de neuronas que cuentan con celdas donde pueden almacenar información tanto a corto como a largo plazo, a diferencia de las celdas de las RNN clásicas donde se puede almacenar información solo por un pequeño período de tiempo. En este sentido podríamos decir, que las LSTM extienden las capacidades de las neuronas recurrentes tradicionales permitiendo que se pueda guardar información por mucho más tiempo.

Las celdas LSTM funcionan como la memoria de un computador, en el sentido que en ellas se puede almacenar, leer y borrar información, esta última operación se da dependiendo de la importancia que tengan los datos, es decir, los datos que no son relevantes o significativos son candidatos a ser borrados. La relevancia o no de un dato la determinan unos pesos y las operaciones sobre la celda son controladas por las puertas de entrada, de salida y de olvido. Una unidad LSTM en un instante de tiempo t tendría las transformaciones que se muestran en la siguiente figura:



Una celda LSTM es parecida a una celda normal, salvo en que el estado de la LSTM tiene dos vectores $h(t)$ y $c(t-1)$. El primero representaría el estado a corto plazo donde se almacenaría la información más inmediata y el segundo el estado a largo plazo para almacenar la información más relevante de la secuencia en un paso de tiempo. Además, recordemos que x_t sería la entrada con la cual alimentamos la unidad neuronal en un tiempo t . Los que se encuentran en los rectángulos son funciones de activación sigmoide (σ) y tangente hiperbólica (\tanh). Así mismo los símbolos \otimes y \oplus representan las operaciones de producto de elementos y suma de elementos respectivamente.

En las tres compuertas encontramos las funciones $f(t)$, $i(t)$, $g(t)$, $o(t)$, que son el mecanismo por medio del cual la LSTM busca abordar el problema de la desaparición del gradiente. Es importante mencionar que durante el entrenamiento la red aprende los parámetros (pesos y sesgos) para estas puertas.

Las ecuaciones para estas funciones son como siguen:

Ecuación puerta de olvido:

$$f_{(t)} = \sigma(W_{(f)} \cdot X_{(t)} + W_{(f)} \cdot h_{(t-1)} + b_{(f)})$$

Ecuación puerta de entrada:

$$i_{(t)} = \sigma(W_{(i)} \cdot X_{(t)} + W_{(i)} \cdot h_{(t-1)} + b_{(i)})$$

$$g_{(t)} = \tanh(W_{(g)} \cdot X_{(t)} + W_{(g)} \cdot h_{(t-1)} + b_{(g)})$$

Ecuación puerta de salida:

$$o_{(t)} = \sigma(W_{(o)} \cdot X_{(t)} + W_{(o)} \cdot h_{(t-1)} + b_{(o)})$$

Este último cálculo nos dará las dos salidas, la del término a corto plazo y la del término a largo plazo:

$$y_{(t)} = h_{(t)} = o_{(t)} \otimes \tanh(C_{(t)})$$

$$C_{(t)} = f_{(t)} \otimes C_{(t-1)} + i_{(t)} \otimes g_{(t)}$$

Estas ecuaciones pueden resultar al principio algo complejas, pero para beneficio de todos ya se encuentran debidamente implementadas en tensorflow y en el API Keras como veremos en los próximos ejemplos. Sin embargo, antes de abarcar dichos ejemplos es conveniente mencionar que existen otras arquitecturas de RNNs como la llamada GRU (*Gated Recurrent Unit*) o unidad recurrente cerrada que surgió tiempo después que las LSTM en el año 2014 y es mucho más simplificada y eficiente en el sentido que consume menos recursos computacionales.

Ha llegado el momento de ver la parte práctica, no sin antes comentar que las redes que programemos como parte de la tarea de preprocesado crean secuencias de igual longitud con los índices de cada una de las palabras del vocabulario. Utilizaremos en nuestras redes una capa *Embedding* para mapear cada palabra en un arreglo de tamaño fijo que contenga números reales en un intervalo infinito. Esta técnica es mucho más elegante y menos propensa a generar vectores dispersos como ocurre al aplicar técnicas como *one hot encoding*. Con *Embedding* la posición de una palabra dentro del espacio vectorial (incrustación) se aprende con el entrenamiento y se determina en función de las palabras que rodean a dicha palabra cuando se usa.

En keras la capa de embedding se define de la siguiente manera:

`keras.layers.Embedding(input_dim , output_dim, input_length)`, donde *input_dim* es el tamaño del vocabulario (número de palabras en el texto), *output_dim* representa el tamaño del arreglo en el que las palabras serán incrustadas. Y *input_length* es la longitud de las secuencias de entrada.

Los dos ejercicios a desarrollar consisten en dos problemas típicos en machine learning: análisis de sentimiento y generación de texto.

15.3.1. Ejemplo sobre análisis de sentimiento

Este ejemplo consiste en un análisis de sentimiento basado en el dataset IMDB (*Internet movie database*) el cual almacena 50.000 registros con información referente a comentarios de películas. La etiqueta tiene los valores 1 (comentario positivo) y 0 (comentario negativo). Podemos disponer de este dataset debidamente preprocesado desde el módulo *datasets* de Keras.

Importamos las librerías necesarias, cargamos el conjunto de datos IMDB y ajustamos la longitud de las entradas a un tamaño fijo, dado que los comentarios suelen tener longitud variable.

```
import numpy as np
from tensorflow import keras
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

(X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data(num_
words=10000)

# hacemos que todas las entradas tengan la misma longitud = 150
X_train = pad_sequences(X_train, maxlen=150)
X_test = pad_sequences(X_test, maxlen=150)
```

Mostramos una instancia del conjunto de entrenamiento:

```
palabra_a_id = keras.datasets.imdb.get_word_index()
id_a_palabra = {i: palabra for palabra, i in palabra_a_id.items()}
print('Comentario: ' + str( [id_a_palabra.get(i, '') for i in X_train[6]]) )
print('Etiqueta: ' + str(y_train[6]))
```

Comentario: [',', '/', 'of', 'props', 'this', 'and', 'concentrates', 'concept', 'issue', 'skeptical', 'to', "god's", 'he', 'is', 'and', 'unfolds', 'movie', 'women', 'like', "isn't", 'surely', "i'm", 'and', 'to', 'toward', 'in', "here's", 'for', 'from', 'did', 'having', 'because', 'very', 'quality', 'it', 'is', 'and', 'starship', 'really', 'book', 'is', 'both', 'too', 'worked', 'carl', 'of', 'and', 'br', 'of', 'reviewer', 'closer', 'figure', 'really', 'there', 'will', 'originals', 'things', 'is', 'far', 'this', 'make', 'mistakes', 'and', 'was', "couldn't", 'of', 'few', 'br', 'of', 'you', 'to', "don't", 'female', 'than', 'place', 'she', 'to', 'was', 'between', 'that', 'nothing', 'dose', 'movies', 'get', 'are', 'and', 'br', 'yes', 'female', 'just', 'its', 'because', 'many', 'br', 'of', 'overly', 'to', 'descent', 'people', 'time', 'very', 'bland']

Etiqueta: 1

Creamos nuestro modelo de red neuronal con una capa Embedding y dos capas LSTM definidas con 32 unidades y disponiendo de regularización con dropout. La última capa es densa y con activación sigmoide puesto que se trata de un problema de clasificación binario.

```
model = keras.models.Sequential([
    keras.layers.Embedding(input_dim=10000 , output_dim=32, input_length=150),
    keras.layers.LSTM(32, dropout=0.1 ,recurrent_dropout=0.2, return_sequences=True),
    keras.layers.LSTM(32, dropout=0.1 ,recurrent_dropout=0.2),
    keras.layers.Dense(1, activation='sigmoid')
])
model.summary()
```

```
Model: "sequential_4"
-----  

Layer (type)          Output Shape         Param #  

-----  

embedding_5 (Embedding)    (None, 150, 32)      320000  

-----  

lstm_9 (LSTM)           (None, 150, 32)      8320  

-----  

lstm_10 (LSTM)          (None, 32)            8320  

-----  

dense_5 (Dense)         (None, 1)              33  

-----  

Total params: 336,673  

Trainable params: 336,673  

Non-trainable params: 0
```

Compilamos y entrenamos nuestro modelo de red neuronal recurrente:

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['acc'])
model.fit(X_train,y_train, epochs=5, batch_size=64, verbose=1, validation_data=(X_test,y_test))
```

Y después de unos minutos podemos calcular la exactitud del modelo con los datos de prueba:

```
scores = model.evaluate(X_test, y_test, verbose=0)
print('Exactitud: {:.2f}'.format(scores[1]))
```

Exactitud: 0.83

Finalmente intentemos probar el rendimiento con una muestra positiva y una muestra negativa del conjunto de prueba:

```
texto_neg = X_test[9]
texto_pos = X_test[13]
texts = (texto_neg, texto_pos)
textos = pad_sequences(texts, maxlen=300, value = 0.0)
preds = model.predict(textos)
print("predicciones:", preds)
```

Predicciones: [[0.97783935]
[0.00336328]]

15.3.2. Ejemplo sobre generación de texto

Este nuevo ejercicio como su nombre lo indica consiste en generar texto por medio de la utilización repetida del método de modelado de lenguaje a nivel de carácter. Este último consiste en darle a la red neuronal una palabra para que genere una distribución de probabilidad para el carácter que debería seguir a la cadena o secuencia inicial. Bajo este principio si llamamos a este modelo repetidamente conseguiremos generar textos más largos.

Hemos decidido emplear para el entrenamiento un documento con las fabulas de Feliz Samaniego, disponibles en texto plano como parte del proyecto Gutenberg. El texto lo encontramos en la siguiente dirección de internet: <https://www.gutenberg.org/files/55206/55206-0.txt>.

Comenzamos importando las clases requeridas para luego proceder con la descarga del archivo y mostrar una porción del texto del mismo:

```
import numpy as np
import pandas as pd
import string
```

```

import matplotlib.pyplot as plt
import tensorflow as tf

path = tf.keras.utils.get_file('Shakespear.txt', 'https://www.gutenberg.org/
files/55206/55206-0.txt')
text = open(path, 'rb').read().decode(encoding='utf-8')
print(text[:500])

```

Obtenemos la cantidad de caracteres del texto y cuantos caracteres únicos hay en este, así:

```

print('Longitud del texto: {} caracteres'.format(len(text)))
vocab = sorted(set(text))
print ('El texto está compuesto de: {} caracteres'.format(len(vocab)))

```

Longitud del texto: 326632 caracteres
 El texto está compuesto de: 115 caracteres

Creamos un índice para cada carácter del texto y luego creamos secuencias de tamaño 100 con estos índices:

```

char_a_ind = {char:i for i, char in enumerate(vocab)}
ind_a_char = np.array(vocab)
codif_text = np.array([char_a_ind[c] for c in text])

long_seq = 100
dataset_caract = tf.data.Dataset.from_tensor_slices(codif_text)
secuencias = dataset_caract.batch(long_seq+1, drop_remainder=True)
for i in secuencias.take(2):
    print(ind_a_char[i.numpy()])

```

Configuramos nuestro dataset y le damos la forma adecuada para el entrenamiento de nuestra red neuronal.

```
def separa_entrada_etiqueta(p):
    texto_entrada = p[:-1]
    text_destino = p[1:]
    return texto_entrada, text_destino

dataset = secuencias.map(separa_entrada_etiqueta)

TAM_LOTE = 64
TAM_BUFFER = 10000

dataset      =      dataset.shuffle(TAM_BUFFER).batch(TAM_LOTE,      drop_
remainder=True)
```

Definimos algunos parámetros, creamos nuestro modelo de red neuronal recurrente y lo entrenamos:

```
tam_vocab = len(vocab)
tam_embedding = 256
unidades_rnn = 1024

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense

def crear_modelo(tam_vocab, tam_embedding, unidades_rnn, tam_lote):
    model = Sequential()
    model.add(Embedding(input_dim=tam_vocab,
                        output_dim=tam_embedding,
                        batch_input_shape=[tam_lote, None]))
    model.add(LSTM(unidades_rnn,
                  return_sequences=True,
                  stateful=True,
                  recurrent_initializer='glorot_uniform'))
    model.add(Dense(tam_vocab))
    return model

model = crear_modelo(
    tam_vocab = len(vocab),
    tam_embedding=tam_embedding,
    unidades_rnn=unidades_rnn,
    tam_lote=TAM_LOTE)
model.summary()
```

```

def perdida(etiquetas, preds):
    return tf.keras.losses.sparse_categorical_crossentropy(etiquetas, preds, from_
logits=True)

model.compile(optimizer='adam', loss=perdida)
epocas = 5
history = model.fit(dataset, epochs=epocas, verbose=1)

```

Después de unos cuantos minutos de entrenamiento exponemos la última parte del código encargado de hacer posible la generación de texto:

```

from tensorflow.keras.models import load_model
model.save('modelo.h5')

model = crear_modelo(tam_vocab, tam_embedding, unidades_rnn, tam_
lote=1)
model.load_weights('modelo.h5')
model.build(tf.TensorShape([1, None]))

def generar_texto(model, cad_inicio):
    total_gen = 1000
    ent_e = [char_a_ind[s] for s in cad_inicio]
    ent_e = tf.expand_dims(ent_e, 0)
    texto_generado = []
    temperature = 0.5
    model.reset_states()
    for i in range(total_gen):
        predicciones = model(ent_e)
        predicciones = tf.squeeze(predicciones, 0)
        predicciones = predicciones / temperature
        id_prediccion = tf.random.categorical(predicciones, num_samples=1)[-1,0]._
numpy()
        ent_e = tf.expand_dims([id_prediccion], 0)
        texto_generado.append(ind_a_char[id_prediccion])
    return (cad_inicio + ''.join(texto_generado))

```

Para probar el modelo ejecutamos el método `generar_texto()` con la palabra inicial "libro". Vemos que, aunque el texto predicho carece de sentido, si nos permite conocer la potencia de estas redes en la generación de secuencias de texto. Se recomienda al lector como siempre probar modificando la arquitectura de la red

(agregando capas) junto con otra combinación de hiperparámetros con el fin de tratar de alcanzar mejores resultados.

```
print(generar_texto(model, cad_inicio=u"libro"))
```

Con este par de ejemplos damos por terminado nuestro estudio sobre las redes neuronales recurrentes, cabe resaltar que el tema es denso, pero con esta introducción el lector dispondrá de un punto de partida para comprender algunos aspectos básicos que le marquen el camino hacia la profundización de los conceptos aquí expuestos y el aprendizaje de otros nuevos, con el fin de explotar más fondo la potencialidad que ofrecen estas redes.

Así mismo, damos por terminado la temática del libro, espero que los tópicos aquí expuestos hayan sido del agrado del lector y hayan sido lo suficientemente claros. Espero que el material y la información presentada marque el punto de partida para que el lector siga profundizando y actualizando los conocimientos adquiridos sobre las técnicas de aprendizaje automático y profundo.

Referencias bibliográficas

- Bobadilla, Jesús (2020). *Machine Learning y Deep Learning*. Bogotá: Ediciones de la U.
- Borges, Pedro. *Deep Learning: Recurrent Neural Networks*. Consultado el 11 de noviembre de 2020. Disponible en: <https://medium.com/deeplearningbrasilia/deep-learning-recurrent-neural-networks-f9482a24d010>
- Caballero Rafael; Martin, Enrique y Riesco, Adrián (2018). *Big data con Python*. Madrid: RC Libros.
- Chollet, Francois (2018). *Deep learning with Python*. Shelter Island: Manning.
- Dimas, José (2019). *Aprende a programar con Python*. Madrid: RC Libros.
- Documentación de la biblioteca Scikit-Learn. Disponible en: https://scikit-learn.org/stable/user_guide.html
- Documentación de la biblioteca Keras. Disponible en: https://keras.io/getting_started/
- Machine Learning Notebook. *Recurrent Neural Network*. Consultado el 10 de diciembre de 2020. Disponible en: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks
- Raschka, Sebastian y Mirjalili, Vahid (2017). *Python Machine Learning*. Packt Publishing.
- Torres, Jordi (2020). *Python Deep Learning*. Madrid: Marcombo.
- VanderPlas, Jake. Python Data Science Handbook. *Hyperparameters and Model Validation*. Consultado el 06 de junio de 2020. Disponible en: <https://jakevdp.github.io/PythonDataScienceHandbook/05.03-hyperparameters-and-model-validation.html>

<https://dogramcode.com/bloglibros>



Índice analítico

A

- Adaboost, 198
- ADALINE, 143
- Agrupación, 288
- Análisis de componentes principales, 240
- Análisis discriminante lineal, 250
- Aprendizaje automático, 35
- Aprendizaje no supervisado, 42
- Aprendizaje por refuerzo, 43
- Aprendizaje semisupervisado, 42
- Aprendizaje supervisado, 40
- Arboles de decisión, 184
- Aumento de datos, 305

B

- Backpropagation, 263
- Bolsa de palabras continuas, 316
- Bosques aleatorios, 196
- BPTT, 326
- Búsqueda de cuadrículas, 131

C

- CBOW, 316
- Celdas de memoria, 324
- Clasificación, 41
- Clustering, 223
- Clustering aglomerativo, 231
- Clustering Jerárquico, 231
- Coeficiente de determinación, 124
- Coeficiente de Pearson, 213
- Conjunto de datos, 37
- Conjunto de entrenamiento y pruebas, 74

C

- Convolución, 285
- Curvas de aprendizaje, 127
- Curvas de validación, 127
- Curvas ROC, 168

D

- DBSCAN, 233
- desaparición del gradiente, 260
- Descenso del gradiente, 92
- descenso del gradiente estocástico, 159
- Distancia euclídea, 211
- Dropout, 295

E

- Entrenamiento del modelo, 39
- Entropía, 190
- Error absoluto medio, 124
- Error cuadrático medio, 89, 123
- Escalamiento de características, 80
- Escalón, 257
- Estandarización, 81
- Evaluación, 38
- Exactitud (Accuracy), 165
- Extracción de características, 310

F

- F1, 166
- FPR, 167
- Funciones de activación, 253

G

- Gensim, 318
- Gradient boosting, 200

- H**
Hipérparámetros, 37
- I**
imputación, 76
- K**
K Means, 223
K Vecinos mas cercanos, 69, 206
Keras, 271
Kernel, 285
Kernel con base radial, 180
Kernel Lineal, 180
- L**
La impureza de Gini, 187
límite de decisión, 142
LSTM, 327
- M**
Maldición de la dimensionalidad, 44
máquinas de vectores de soporte, 169
matplotlib, 62
Matriz de confusión, 148, 164
Matriz de correlación, 87
Matriz de dispersión, 88
Método del codo, 227
Métricas de evaluación, 164
mínimos cuadrados, 89
Modelo, 36
- N**
Naive bayes, 202
Neurona lineal adaptativa, 143
Normalización, 82
Numpy, 53
- O**
Overfitting, 44
- P**
Pandas, 48
Parámetros, 27
- PCA, 240
Perceptrón, 138
Pesos, 256
PLN, 284, 315
Precisión (Precision), 166
Predicción, 38
Preprocesado de datos, 73
Preprocesamiento de datos, 37
Probabilidad condicional, 202
Procesamiento del lenguaje natural, 315
Propagación a través del tiempo (BPTT), 326
Python, 11
- R**
Recall, 166
Red elástica, 122
Red neuronal, 255
redes neuronales, 253
Redes neuronales convolucionales, 283
Redes neuronales recurrentes, 323
Reducción de la dimensionalidad, 239
Regresión, 40, 83
Regresión Lasso, 122
Regresión lineal polinómica, 102
regresión lineal simple, 84
Regresión logística, 148
regresión múltiple, 85
Regresión Rígida, 120
Regularización, 120, 305
RELU, 257
- S**
Sesgo, 134
Sesgos, 256
Sigmoid, 259
Sistema de recomendación colaborativo, 210
Sistemas de recomendación, 210
Sistemas de recomendación basados en contenido, 210
Skip-gram, 317

ÍNDICE ANALÍTICO

Sobreajuste, 44

Softmax, 262

Subajuste, 44

SVM, 169

T

Tangente hiperbólica, 257

tasa de aprendizaje, 199

Teorema de Bayes, 202

Tipos de datos, 20

Transferencia de aprendizaje, 308

U

Underfitting, 44

V

Validación cruzada, 125

Variable destino o etiqueta, 39

Variables descriptivas, 39

Varianza, 127

W

Word2vec, 316, 318

APRENDIZAJE AUTOMÁTICO Y PROFUNDO EN PYTHON

Es una herramienta de apoyo y de consulta para toda aquella persona interesada en dominar los fundamentos del aprendizaje automático y profundo, a tal punto que le permita aprender lo necesario para desarrollar sus propios modelos de aprendizaje aptos para realizar predicciones con base en los datos, para ello el autor hará uso en la mayoría de los casos de explicaciones teóricas y prácticas, que permitan al lector afianzar sus ideas y fortalecer su aprendizaje.

El libro se encuentra dividido en dos partes la primera enfocada en el *machine learning* y sus diferentes algoritmos de regresión, clasificación, *clustering*, entre otros. La segunda parte comprende varias técnicas de *deep learning* donde estudiaremos diferentes arquitecturas de redes neuronales como: redes densamente conectadas, redes convolucionales y redes recurrentes.

✓ Introducción al lenguaje Python

✓ Funcionamiento de diferentes métodos y técnicas usadas en el aprendizaje automático

✓ Ejemplos y ejercicios prácticos



Carlos M. Pineda Pertuz

Ingeniero de sistemas, especialista en informática y telemática y magíster en ingeniería de software y sistemas informáticos. Soy amante de la lectura, la investigación, las nuevas tecnologías y todo lo relacionado con la programación.

Soy docente universitario y he participado en varios proyectos de desarrollo de sistemas informáticos en diversos lenguajes de programación.

ISBN: 978-958-792-316-2



9 789587 923162

edij