

BIG DATA APLICADO



UT03: PROCESAMIENTO PARALELO
EN APACHE HADOOP: SPARK

ÍNDICE

- 1.- Fundamentos de Apache Spark**
- 2.- RDDs (Resilient Distributed Datasets)**
- 3.- Manipulación de datos con SparkSQL y DataFrames**
- 4.- Spark en IA y Machine Learning**



Ejemplos Prácticos Recomendados

Para un curso práctico, es vital que implementen ejemplos de principio a fin.

- **Ejemplo 1:** ETL (Extract, Transform, Load) de grandes volúmenes de datos.
- **Ejemplo 2:** Construcción de un Sistema de Recomendación simple con MLlib.
- **Ejemplo 3:** Análisis de Sentimiento en tiempo real usando Structured Streaming y datos de ejemplo simulados (e.g., de Twitter/Kafka).

1

FUNDAMENTOS DE APACHE SPARK



1

FUNDAMENTOS DE APACHE SPARK

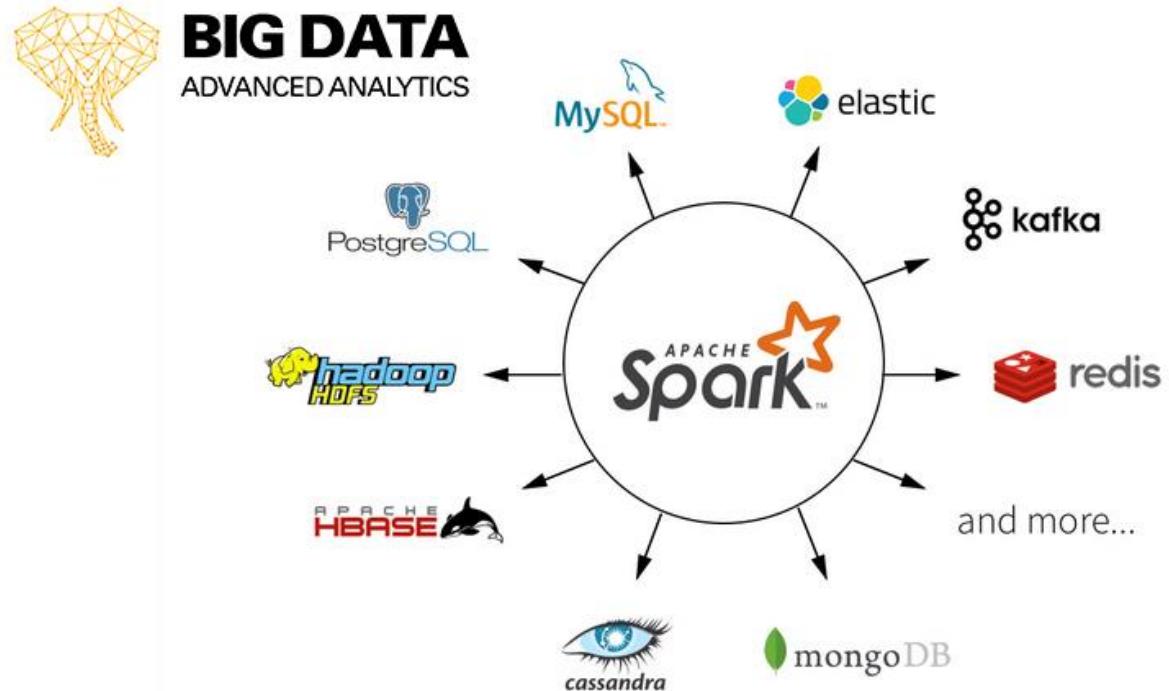
1.1

¿QUÉ ES APACHE
SPARK?

Apache Spark es un **motor de análisis y procesamiento de datos distribuidos y de código abierto** diseñado para manejar grandes volúmenes de datos de manera eficiente y rápida.

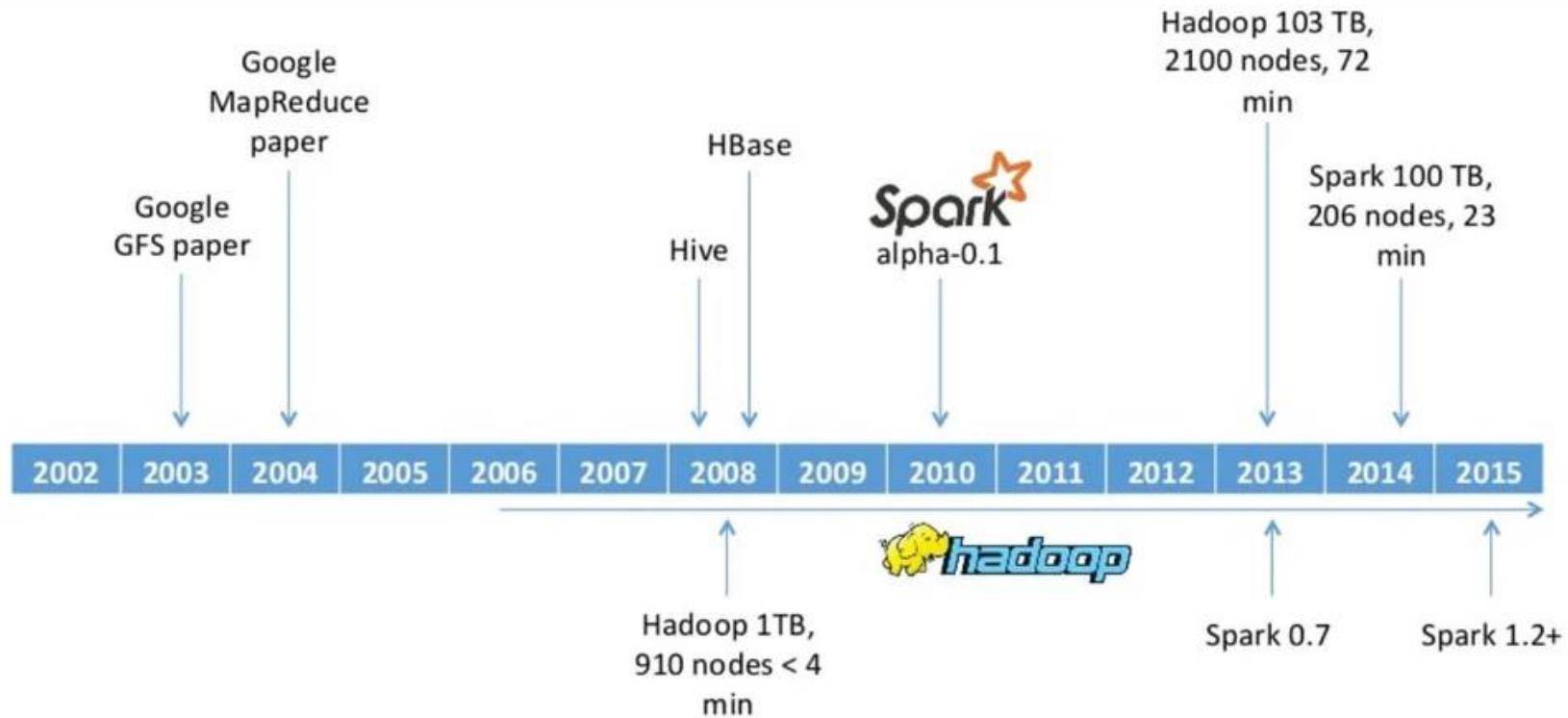
Su principal fortaleza es la capacidad de ejecutar tareas de **procesamiento en memoria**, lo que le hace extremadamente rápido.

Apache Spark puede leer datos de diversas fuentes (HDFS, S3, Cassandra, Kafka,...) y aplicar transformaciones complejas en paralelo a través de un clúster de máquinas.



HISTORIA DE APACHE SPARK

- **2009:** nace como proyecto de investigación en el AMPLab de la Universidad de Berkeley, liderado por Matei Zaharia.
- **2010:** el código se publica como Open Source
- **2013:** se dona a la Apache Software Foundation
- **2014:** se convierte en un proyecto de nivel superior de Apache y rápidamente gana popularidad, superando a MapReduce como estándar de facto para el procesamiento de Big Data



ARQUITECTURA DE APACHE SPARK

La arquitectura de Spark está diseñada en torno al concepto de procesamiento en un clúster de nodos.

Driver Program (Programa Controlador)

Es el proceso que ejecuta el programa principal (p.e. un script de Python o Scala).

Contiene la **Spark Session** (o el antiguo **Spark Context**), que es el punto de entrada a toda la funcionalidad de Spark.

Analiza, optimiza y distribuye la ejecución del trabajo a los *Executors* en el clúster.

Cluster Manager(Gestor del clúster)

Es un servicio externo (como **YARN**, **Mesos** o el propio **Spark Standalone**) que se encarga de adquirir recursos (cores y memoria) en el clúster.

Lanza los procesos *Executor* en los *Worker Nodes*.

Worker Nodes (Nodos trabajadores)

Las máquinas físicas o virtuales **donde se ejecutan las tareas**.

Cada *Worker Node* aloja uno o más procesos *Executor*.

Executors (Ejecutores)

Los procesos que realmente ejecutan las tareas (tasks) de cómputo en los Worker Nodes.

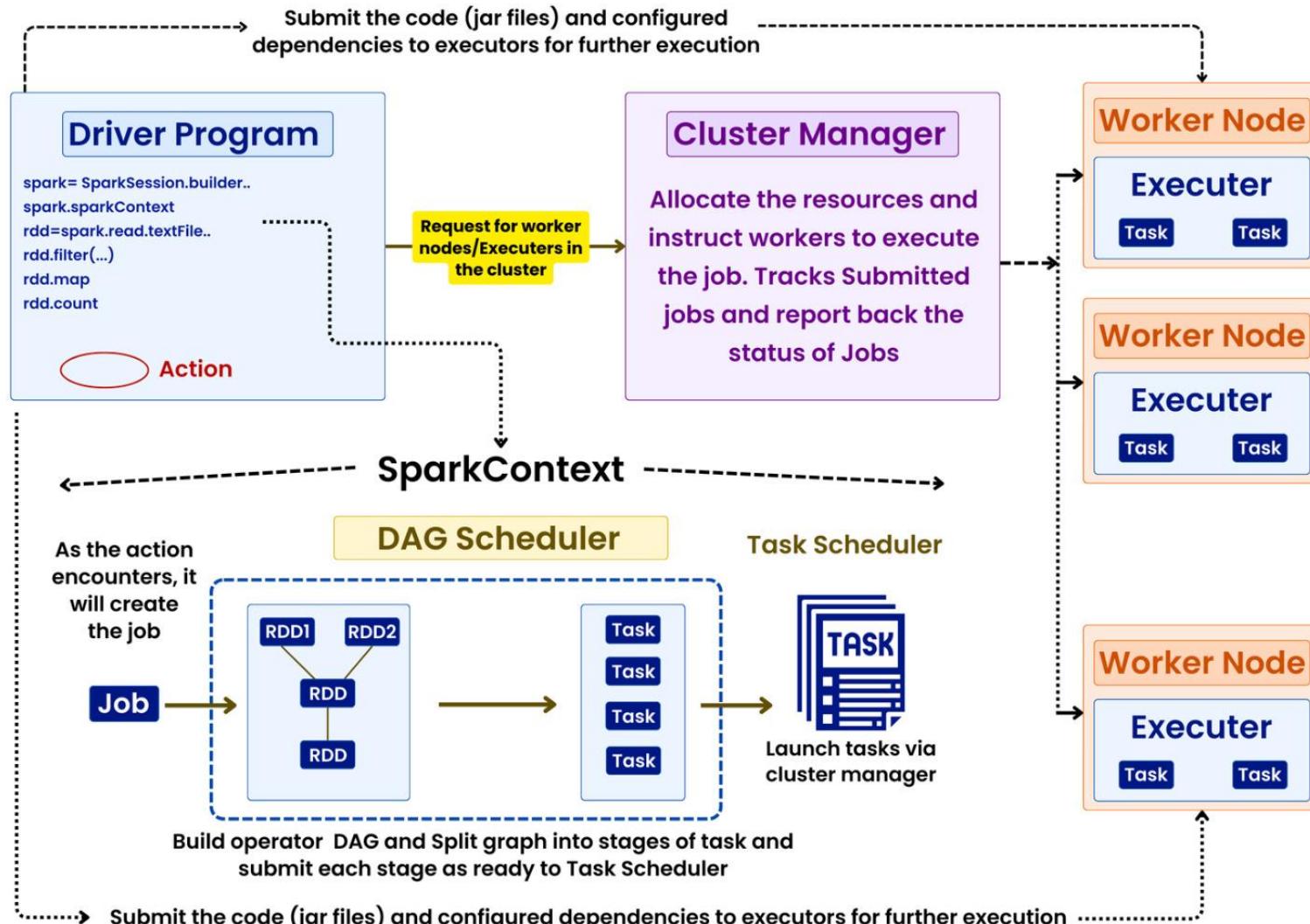
Almacenan los datos en caché de forma persistente.

Reportan el estado y los resultados de vuelta al *Driver Program*.

Internals of Job Execution in Spark



Created by:
Rocky Bhatia [in](#)



RDD (Resilient Distributed Dataset)

Es una **colección inmutable, distribuida y tolerante a fallos de objetos** en el clúster de Spark, que se puede operar en paralelo.

Es la **abstracción** central de Spark Core

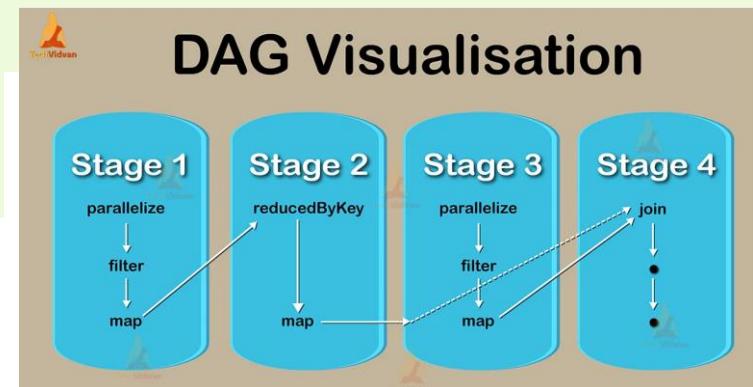
Sus principales características son:

Resiliente (tolerante a fallos)

Un RDD es una **colección de elementos tolerante a fallos que se pueden operar en paralelo**.

Si un nodo falla, Spark puede reconstruir las particiones de datos perdidas desde el origen, sin perder el estado del cálculo.

Para ello utiliza los **Grafos Acíclicos Dirigidos (DAG)**



Distribuido

Un RDD divide automáticamente los datos en varias piezas lógicas llamadas **particiones**.

Cada partición reside en un *worker* diferente del clúster y **se procesa en paralelo**.

El usuario escribe una operación (p.e. map) y Spark se encarga de dividir y orquestar miles de tareas en los diferentes nodos.

Dataset (conjunto de datos)

El *dataset* describe el contenido del RDD

Los RDDs son colecciones de objetos (como una lista o array) que se encuentran dispersos en memoria o disco de las máquinas del clúster.

Los RDDs son **inmutables**. Una vez que se crea un RDD no se puede cambiar. Esto es esencial para la coherencia y la recuperación.

Evaluación perezosa

Las operaciones con RDDs se dividen en dos categorías:

- **Transformaciones:**
 - Crean un nuevo RDD a partir del existente (map, reduce, ..)
 - No ejecutan nada inmediatamente, simplemente registran la operación en el DAG.
- **Acciones**
 - Son operaciones que activan el cómputo (collect, count, save, ...)
 - Cuando se llama a una acción, Spark mira el DAG y planifica la ejecución optimizada de todas las transformaciones

Las transformaciones definen la eficiencia del RDD.

Hay dos tipos:

- **Transformaciones estrechas:**

- Cada partición del RDD padre contribuye a una única partición en el RDD resultante (p.e. map, filter)
- Esto es rápido porque no requiere mover datos entre nodos

- **Transformaciones anchas:**

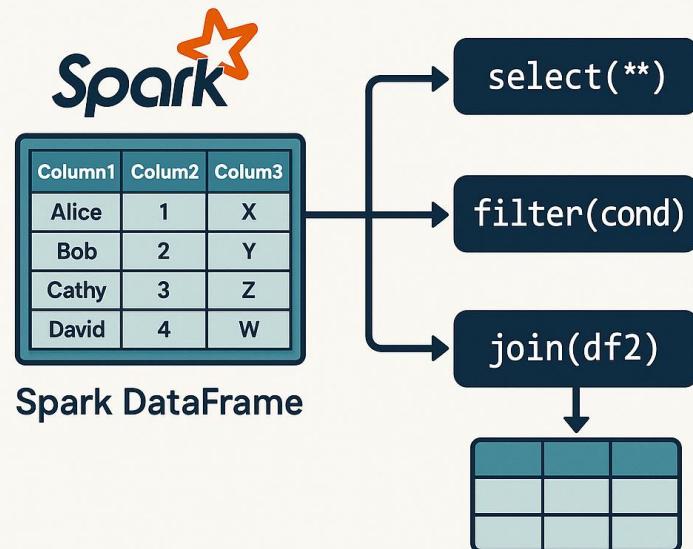
- Una partición del RDD padre puede contribuir a múltiples particiones en el RDD resultante (p.e. groupByKey, reduceByKey)
- Esto requiere un **shuffling**, que es la transferencia costosa de datos a través de la red del clúster

Dataframes

Un dataframe es una **colección distribuida de datos organizados en columnas con nombre**

Conceptualmente es equivalente a una tabla de una base de datos relacional o a un *dataframe* de Pandas, pero con **optimizaciones masivas para ejecutarse en un clúster** de computadoras.

Introduction to Spark DataFrames Basic DataFrame Operations



Sus principales características son:

- **Estructurados:** los dataframes tienen un esquema definido (frente a los RDDs que son colecciones de objetos arbitrarios)
- **Distribuidos:** los datos no residen en una sola máquina.
- **Inmutables:** una vez creado, un dataframe no se puede modificar, si se quiere cambiar algo hay que crear un nuevo dataframe a través de una transformación.
- **Evaluación perezosa:** al igual que pasa con los RDDs, Spark no ejecuta nada inmediatamente, sino que construye un plan lógico y solo procesa los datos cuando se llama a una acción (`show()`, `count()`, `write()`)
- **API unificada:** pueden ser manipulados desde diferentes lenguajes (Scala, Java, Python, R). También permiten ejecutar consultas SQL directamente sobre ellos creando vistas temporales.

2

DATAFRAMES: INGESTA Y TRANSFORMACIÓN



2.1

**INTRODUCCIÓN A
PYSPARK**

2

**DATAFRAMES:
INGESTA Y
TRANSFORMACIÓN**

Vamos a lanzar Spark en un entorno contenerizado

(https://vgonzalez165.github.io/docker_resources/compose/spark/) compuesto por un master, un worker y un tercer contenedor con Jupyter que nos servirá para programar con PySpark

●	<u>spark</u>	-	-	-
●	jupyter-spark-driver	e845bbfd37a2	spark-jupyter-driver	8899:8888 ↗
●	worker-1	62d4a6bb9690	spark:3.5.7-scala2.12-java11-pythc	8081:8081 ↗
●	master	0ee6e345dc6	spark:3.5.7-scala2.12-java11-pythc	7077:7077 ↗ 8080:8080 ↗

Conexión a Spark

Lo primero que tenemos que hacer es conectarnos a Spark

```
from pyspark.sql import SparkSession  
  
try:  
    spark = ( SparkSession.builder  
              .appName("Haciendo pruebas")  
              .master("spark://spark-master:7077")  
              .getOrCreate()  
    )  
  
    print("SparkSession iniciada correctamente.")  
except Exception as e:  
    print("Error en la conexión")  
    print(e)  
  
sc = spark.sparkContext
```

Comenzamos creando una sesión de Spark

Conectamos al contenedor master de Spark a través del puerto 7077

Carga de datos en Spark

En este primer programa simularemos datos de prueba

```
data = [  
    ("Ana", 15),  
    ("Carlos", 22),  
    ("Luis", 10),  
    ("Marta", 35)  
]  
  
columns = ["nombre", "edad"]
```

Los dataframes tienen **estructura tabular**. En este caso crearemos una tabla de 2 columnas y 4 registros

Las columnas de las tablas tienen que tener un nombre

```
df = spark.createDataFrame(data, columns)
```

Creamos el dataframe

```
df.show()
```

```
+----+----+  
|nombre|edad|  
+----+----+  
| Ana | 15 |  
| Carlos | 22 |  
| Luis | 10 |  
| Marta | 35 |  
+----+----+
```

Con la función show() podremos mostrar el contenido de un dataframe

Aplicando transformaciones

Vamos a aplicar nuestra primera **transformación** a los datos. Es importante entender que aquí Spark **no hace nada**, solo toma nota de lo que hay que hacer.

```
df_mayores = df.filter("edad >= 18")
df_mayores.explain()
== Physical Plan ==
*(1) Filter (isnotnull(edad#1L) AND (edad#1L >= 18))
+- *(1) Scan ExistingRDD[nombre#0,edad#1L]
```

Usamos **filter()** para filtrar solo los registros cuya edad sea mayor de 18 años

Con **explain()** podemos saber cuál es el plan de trabajo

Internamente Spark trabaja con **RDDs**.

Aquí vemos los pasos que Spark realizará para realizar la operación solicitada. A medida que vayamos añadiendo transformaciones realizará **optimizaciones** sobre este plan

Acciones: ejecutando el plan de trabajo

Cuando realizamos una **acción** es cuando pone en marcha el plan de trabajo. En este caso vamos a usar la acción **show()**, que muestra los resultados

```
df_mayores.show()
```

```
+----+----+
|nombre|edad|
+----+----+
|Carlos| 22|
|Marta| 35|
+----+----+
```

2

DATAFRAMES: INGESTA Y TRANSFORMACIÓN

2.2

INGESTA DE DATOS

En el ejemplo anterior creamos los datos directamente en el código, pero lo habitual es obtenerlos de alguna fuente externa.

Spark puede recoger los datos desde diferentes fuentes como CSV, JSON o Parquet.

Esquema

Algo importante antes de leer los datos es el **esquema**, que básicamente se puede definir como **el contrato que define la estructura de los datos**.

El esquema le dice tres cosas al motor sobre cada columna:

- **Nombre** (cómo se llama)
- **Tipo de datos** (qué es: DateType, IntegerType, ArrayType, ...)
- **Nullability** (¿puede estar vacío?)

Si no indicamos el esquema, Spark lo inferirá a partir de los datos, pero indicarlo explícitamente tiene una serie de ventajas:

- **Rendimiento:** si no lo definimos, Spark tendrá que leer el archivo completo antes de cargar los datos para intentar averiguar qué tipo de datos son
- **Seguridad de tipos:** evitar que una columna se cargue con un tipo erróneo (p.e. si en una columna de precios hay un registro con valor 10.5€ en lugar de 10.5)
- **Optimización:** Spark reserva memoria de forma más eficiente si sabe, por ejemplo, que una columna es Byte (1 byte) y no un Long (8 bytes)

Para definir el esquema, usaremos los siguientes objetos del módulo `pyspark.sql.types`:

- **StructType**: representa la **fila** completa. Es una colección (lista) de campos
- **StructField**: representa una **columna** individual

Los tipos de datos que hay son:

Equivalente en Python	Tipo PySpark	Descripción
<code>str</code>	<code>StringType()</code>	Texto libre
<code>int</code>	<code>IntegerType()</code>	Entero estándar (32 bits)
<code>int</code> (grande)	<code>LongType()</code>	Entero grande (64 bits)
<code>float</code>	<code>DoubleType()</code>	Decimales (por defecto en Spark)
<code>decimal</code>	<code>DecimalType()</code>	Finanzas (precisión exacta)
<code>datetime</code>	<code>TimestampType()</code>	Fecha y hora
<code>list</code>	<code>ArrayType(T)</code>	Lista de elementos de tipo T
<code>dict</code>	<code>MapType(K, V)</code>	Clave-valor

Veamos cómo sería el código para definir el esquema:

```
from pyspark.sql.types import StructType, StructField  
from pyspark.sql.types import StringType, IntegerType, BooleanType  
  
schema = StructType([  
    StructField("id", IntegerType(), False),  
    StructField("nombre", StringType(), True),  
    StructField("es_cliente", BooleanType(), True)  
])  
  
schema.fields  
  
[StructField('id', IntegerType(), False),  
 StructField('nombre', StringType(), True),  
 StructField('es_cliente', BooleanType(), True)]
```

Tenemos que importar los tipos de datos que usaremos

Instanciamos el objeto StructType.

Para cada campo definimos el nombre, el tipo de datos que contendrá y si puede ser nulo (True) o no (False)

Podemos usar la propiedad Fields para ver la estructura del esquema

Aunque el ejemplo anterior era una tabla plana, el esquema puede ser **anidado** conteniendo datos complejos (tipo JSON o NoSQL como MongoDB)

```
direccion_schema = StructType([
    StructField("calle", StringType(), True),
    StructField("cp", StringType(), True)
])
```

Creamos primero el esquema del tipo complejo

```
main_schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("nombre", StringType(), True),
    StructField("direccion", direccion_schema, True)
])
```

```
root
|-- id: integer (nullable = true)
|-- nombre: string (nullable = true)
|-- direccion: struct (nullable = true)
|   |-- calle: string (nullable = true)
|   |-- cp: string (nullable = true)
```

Y luego lo indicamos como tipo de datos del tipo principal

Para cargar datos directamente en el *dataframe* debemos utilizar la función **createDataFrame()**, a la que opcionalmente se le puede pasar el esquema.

```
data = [
    (1, "Juan Perez", ("Calle Gran Vía 25", "28013")),
    (2, "Maria Lopez", ("Av. Diagonal 100", "08018")),
    (3, "Carlos Ruiz", None)
]

df = spark.createDataFrame(data, schema=main_schema)
```

Usamos el parámetro `schema` para indicar el esquema al crear el *dataframe*

```
df.printSchema()
root
|-- id: integer (nullable = true)
|-- nombre: string (nullable = true)
|-- direccion: struct (nullable = true)
|   |-- calle: string (nullable = true)
|   |-- cp: string (nullable = true)
```

Con la función `printSchema()` podremos visualizar el esquema del *dataframe*

Si tengo campos anidados, podré acceder a ellos utilizando la notación del punto

La transformación
select() filtra por
columnas

```
df.select("nombre", "direccion.calle").show()
```

nombre	calle
Juan Perez	Calle Gran Vía 25
Maria Lopez	Av. Diagonal 100
Carlos Ruiz	NULL

La acción show()
muestra el dataframe

Para cargar datos desde un fichero debemos utilizar el objeto **spark.read**, que nos da acceso a la API **DataFrameReader**.

La sintaxis general para leer un fichero es:

```
df = ( spark.read  
    .format("tipo_archivo")  
    .option("clave", "valor")  
    .schema(mi_esquema)  
    .load("ruta/al/archivo")  
)
```

Formato de la fuente de datos: csv, json, parquet, orc, text, jdbc

Configuraciones específicas del formato (p.e. separador para CSV o multiline para JSON)

Si hemos creado manualmente el esquema lo indicamos aquí

Ruta al fichero, puede ser local (`file://`) o distribuida (`hdfs://`, `s3a://`)

Carga de datos de CSV.

Vamos a leer el fichero worldcities.csv

Name	Modified
recursos	6 min. ago
PR01_No_evaluable.ipynb	24 days ago
spam.data	24 days ago
Untitled.ipynb	24 days ago
Untitled1.ipynb	24 days ago
worldcities.csv	8 min. ago

	city	city_ascii	lat	lng	country	iso2	iso3	admin_name	capital	population	id
1	Tokyo	Tokyo	35.6870	139.7495	Japan	JP	JPN	Tōkyō	primary	37785000	1392685764
2	Jakarta	Jakarta	-6.1750	106.8275	Indonesia	ID	IDN	Jakarta	primary	33756000	1360771077
3	Delhi	Delhi	28.6100	77.2300	India	IN	IND	Delhi	admin	32260000	1356872604
4	Guangzhou	Guangzhou	23.1300	113.2600	China	CN	CHN	Guangdong	admin	26940000	1156237133
5	Mumbai	Mumbai	19.0761	72.8775	India	IN	IND	Mahārāshtra	admin	24973000	1356226629
6	Manila	Manila	14.5958	120.9772	Philippines	PH	PHL	Manila	primary	24922000	1608618140
7	Shanghai	Shanghai	31.2286	121.4747	China	CN	CHN	Shanghai	admin	24073000	1156073548
8	São Paulo	Sao Paulo	-23.5504	-46.6339	Brazil	BR	BRA	São Paulo	admin	23086000	1076532519

```
!head -n2 ./worldcities.csv
```

```
"city","city_ascii","lat","lng","country","iso2","iso3","admin_name","capital","population","id"  
"Tokyo","Tokyo","35.6870","139.7495","Japan","JP","JPN","Tōkyō","primary","37785000","1392685764"
```

Comenzamos analizando la estructura de los datos para crear manualmente el esquema.

World cities database

Accurate and up-to-date database of the world's cities and towns



<https://www.kaggle.com/datasets/juanmah/world-cities>

Una vez comprendidos los datos, creamos el esquema:

```
from pyspark.sql.types import StructType, StructField, StringType, DoubleType, LongType

schema_worldcities = StructType([
    StructField("city", StringType(), True),
    StructField("city_ascii", StringType(), True),
    StructField("lat", DoubleType(), True),
    StructField("lng", DoubleType(), True),
    StructField("country", StringType(), True),
    StructField("iso2", StringType(), True),
    StructField("iso3", StringType(), True),
    StructField("admin_name", StringType(), True),
    StructField("capital", StringType(), True),
    StructField("population", LongType(), True),
    StructField("id", LongType(), True)
])
```

Y procedemos a cargar los datos:

```
df_cities = ( spark.read  
              .format("csv")  
              .schema(schema_worldcities)  
              .option("header", "true")  
              .option("quote", "\"")  
              .load("./worldcities.csv")  
 )  
  
df_cities.printSchema()  
df_cities.show(5)
```

Como el fichero tiene encabezado se lo indicamos

En el fichero todos los campos están rodeados de comillas dobles. Con quote se lo indicamos de forma que las omita y procese únicamente lo que hay entre comillas.

```
"city","city_ascii","lat",  
"Tokyo","Tokyo","35.6870",
```

Con show(5) muestro las 5 primeras filas del *dataframe*

Comprobamos que el esquema del *dataframe* se corresponde con lo que le indicamos

```
root
|-- city: string (nullable = true)
|-- city_ascii: string (nullable = true)
|-- lat: double (nullable = true)
|-- lng: double (nullable = true)
|-- country: string (nullable = true)
|-- iso2: string (nullable = true)
|-- iso3: string (nullable = true)
|-- admin_name: string (nullable = true)
|-- capital: string (nullable = true)
|-- population: long (nullable = true)
|-- id: long (nullable = true)
```

Y podemos ver los datos ya cargados

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| city|city_ascii|  lat|   lng| country|iso2|iso3| admin_name|capital|population|      id|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Tokyo|    Tokyo| 35.687|139.7495|   Japan| JP| JPN|    Tōkyō|primary| 37785000|1392685764|
| Jakarta|  Jakarta| -6.175|106.8275|Indonesia| ID| IDN|    Jakarta|primary| 33756000|1360771077|
| Delhi|    Delhi|  28.61|   77.23|    India| IN| IND|    Delhi| admin| 32226000|1356872604|
| Guangzhou| Guangzhou| 23.13| 113.26|    China| CN| CHN| Guangdong| admin| 26940000|1156237133|
| Mumbai|   Mumbai|19.0761| 72.8775|    India| IN| IND|Mahārāshtra| admin| 24973000|1356226629|
+-----+-----+-----+-----+-----+-----+-----+-----+
only showing top 5 rows
```

Carga de datos JSON

Vamos a ver cómo cargar datos de un JSON, en esta ocasión del *dataset* del Titanic



ABDUL BASIT AI · UPDATED 3 MONTHS AGO

Titanic-json-format

Titanic dataset in JSON format — ready to use with Pandas and APIs.

1 Code Download



<https://www.kaggle.com/datasets/engrbasit62/titanic-json-format>

```
titanic.json X dataframes_sample  
root 891 items  
0  
PassengerId "1"  
Survived "0"  
Pclass "3"  
Name ""Braund, Mr. Owen Harris""  
Sex "male"  
Age "22"  
SibSp "1"  
Parch "0"  
Ticket "A/5 21171"  
Fare "7.25"  
Cabin ""  
Embarked "S"
```

Comenzamos creando el esquema

```
# Definición del esquema para el dataset Titanic
schema_titanic = StructType([
    StructField("PassengerId", IntegerType(), True),
    StructField("Survived", IntegerType(), True),      # 0 o 1
    StructField("Pclass", IntegerType(), True),        # 1, 2 o 3
    StructField("Name", StringType(), True),
    StructField("Sex", StringType(), True),
    StructField("Age", DoubleType(), True),             # Double porque hay edades fraccionarias (ej: 0.42)
    StructField("SibSp", IntegerType(), True),          # Esposos/Hermanos a bordo
    StructField("Parch", IntegerType(), True),           # Padres/Hijos a bordo
    StructField("Ticket", StringType(), True),
    StructField("Fare", DoubleType(), True),             # Tarifa (dinero)
    StructField("Cabin", StringType(), True),
    StructField("Embarked", StringType(), True)          # Puerto de embarque (C, Q, S)
])
```

Y cargamos los datos

```
df_titanic = ( spark.read  
                .format("json")  
                .schema(schema_titanic)  
                .option("multiline", "true")  
                .load("titanic.json")  
            )
```

```
df_titanic.printSchema()  
df_titanic.show(5)
```

Le tenemos que avisar a Spark de que el fichero no es JSONL y tiene varias líneas por registro

Por defecto, Spark espera el formato **JSONL** (JSON Lines) donde:

- Cada línea es un objeto JSON completo y válido
- No hay corchetes envolviendo todo el archivo
- No hay comas al final separando los objetos

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
NULL	NULL	NULL	"Braund, Mr. Owen...	male	NULL	NULL	NULL	A/5 21171	NULL		S
NULL	NULL	NULL	"Cumings, Mrs. Jo...	female	NULL	NULL	NULL	PC 17599	NULL	C85	C
NULL	NULL	NULL	"Heikkinen, Miss....	female	NULL	NULL	NULL	STON/O2. 3101282	NULL		S
NULL	NULL	NULL	"Futrelle, Mrs. J....	female	NULL	NULL	NULL	113803	NULL	C123	S
NULL	NULL	NULL	"Allen, Mr. Willi...	male	NULL	NULL	NULL	373450	NULL		S

Cuando leemos ficheros puede que encontremos **datos corruptos o malformados**.

Debemos indicarle a Spark como actuar ante estos datos mediante
`.option("mode", "mode_name")`

Los modos que hay son:

- **PERMISSIVE:**

- Es el modo por defecto
- Si hay un dato que no encaja con el tipo esperado **no falla ni elimina la fila**
- Convierte el valor corrupto a `null` y carga el resto de la fila
- El problema es que se puede llenar el DataFrame de nulos silenciosamente y estropear cálculos sin darnos cuenta.
- En este modo podemos configurar una columna especial llamada `_corrupt_record` para que Spark guarde ahí el dato original sucio y se pueda revisar luego.

Ejemplo de cómo guardar los datos corruptos en una columna aparte

```
schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("nombre", StringType(), True),
    StructField("edad", IntegerType(), True),
    StructField("_corrupt_record", StringType(), True)
])

df = ( spark.read
        .format("csv")
        .option("header", "true")
        .schema(schema)
        .option("mode", "PERMISSIVE")
        .option("columnNameOfCorruptRecord", "_corrupt_record")
        .load("datos_sucios.csv")
)
```

En esta columna guardaremos los datos corruptos

Debemos usar la opción columnNameOfCorrruptRecord

- **DROPMALFORMED:**

- Si una fila tiene datos corruptos **la ignora por completo**
- Con esto aseguramos que los datos estén limpios, pero en un *dataset* con muchos datos mal formados perderemos muchas líneas y Spark no nos avisará explícitamente.

- **FAILFAST:**

- Al primer error que encuentre, Spark **detiene todo el proceso** inmediatamente y lanza una excepción
- Garantiza al 100% la calidad de los datos
- Se usa en sistemas críticos (p.e. transacciones bancarias) donde no se puede permitir ni un error o dato sucio.

2

2.3

DATAFRAMES: INGESTA Y TRANSFORMACIÓN

OPERACIONES BÁSICAS

Una vez cargados los datos en el DataFrame, lo habitual es prepararlos antes del análisis seleccionando columnas, filtrando filas, corrigiendo valores y creando nuevas métricas.

Es importante recordar que los DataFrames son **inmutables**, es decir, las operaciones no modifican el original, sino que devuelven un nuevo DataFrame con los cambios aplicados.

SELECCIÓN: SELECT

Equivale al SELECT de SQL y sirve para **seleccionar una serie de columnas** y descartar el resto.

```
from pyspark.sql import functions as F
from pyspark.sql.functions import col, lit

df_cities.select("city", "population").show(3)

+-----+-----+
| city|population|
+-----+-----+
| Tokyo| 37785000|
| Jakarta| 33756000|
| Delhi| 32226000|
+-----+-----+
only showing top 3 rows
```

Por convención, las funciones de PySpark se importan como F

Indico las columnas con las que me quiero separar separadas por comas

En lugar de indicar el nombre de la columna, puedo usar la función `col()` que añade más funcionalidades

```
df_cities.select(col("city"), col("population")/1000000).show(3)
```

```
+-----+  
| city|(population / 1000000)|  
+-----+  
| Tokyo| 37.785 |  
| Jakarta| 33.756 |  
| Delhi| 32.226 |  
+-----+  
only showing top 3 rows
```

Puedo realizar **operaciones** con todos los elementos de la columna. En este ejemplo, mostrar los habitantes por millones

```
df_cities.select(col("city"), col("population").alias("población")).show(3)
```

```
+-----+  
| city|población|  
+-----+  
| Tokyo| 37785000 |  
| Jakarta| 33756000 |  
| Delhi| 32226000 |  
+-----+  
only showing top 3 rows
```

Puedo aplicar la función **alias()** para **cambiar el nombre** de la columna

```
df_cities.select(col("city"), col("population").cast("float")).show(3)
```

```
+-----+  
| city|population|  
+-----+  
| Tokyo| 3.7785E7|  
| Jakarta| 3.3756E7|  
| Delhi| 3.2226E7|  
+-----+  
only showing top 3 rows
```

Otra opción es usar la función `cast()` para **cambiar el tipo de datos** de la columna

```
df_cities.select(col("city"), col("population")>10000000).show(3)
```

```
+-----+  
| city|(population > 10000000)|  
+-----+  
| Tokyo| true|  
| Jakarta| true|  
| Delhi| true|  
+-----+  
only showing top 3 rows
```

También podemos **generar una columna True/False** según el valor correspondiente cumpla una condición

RENOMBRADO: `withColumnRenamed()`

Renombra una columna del DataFrame

```
df_cities.withColumnRenamed("Population", "Población").show(3)
```

```
+-----+-----+-----+-----+-----+-----+-----+
| city|city_ascii|  lat|    lng| country|iso2|iso3|admin_name|capital|Población|
+-----+-----+-----+-----+-----+-----+-----+
| Tokyo|    Tokyo|35.687|139.7495|   Japan| JP| JPN|      Tōkyō|primary| 37785000|13926|
| Jakarta|  Jakarta|-6.175|106.8275|Indonesia| ID| IDN|     Jakarta|primary| 33756000|13607|
| Delhi|    Delhi| 28.61|   77.23|   India| IN| IND|      Delhi| admin| 32226000|13568|
+-----+-----+-----+-----+-----+-----+-----+
only showing top 3 rows
```

FILTRADO DE FILAS: `where()` y `filter()`

Ambas funciones son sinónimos exactos y se pueden usar indistintamente.

Filtran los registros del DataFrame según un criterio.

Los operadores Python no funcionan, sino que se deben usar los operadores `&` (AND), `|` (OR) y `~` (NOT)

Cuando hay condiciones compuestas es obligatorio el uso de paréntesis

```
df_cities.filter( col("country") == "Spain" ).show(3)
```

```
+-----+-----+-----+-----+-----+-----+-----+
|    city|city_ascii|      lat|      long|country|iso2|iso3|admin_name|capital|population|
+-----+-----+-----+-----+-----+-----+-----+
| Madrid|    Madrid|40.4169|-3.7033| Spain|   ES| ESP|    Madrid|primary| 6211000|17246
| Barcelona| Barcelona|41.3833| 2.1833| Spain|   ES| ESP| Catalonia| admin| 4800000|17245
| Valencia| Valencia| 39.47|-0.3764| Spain|   ES| ESP|  Valencia| admin| 1595000|17245
+-----+-----+-----+-----+-----+-----+-----+
only showing top 3 rows
```

Ejemplo de condición compuesta

```
(df_cities
    .filter( col("country") == "Spain" & col("population") > 5000000 )
    .select( "city", "country", "population" )
    .show(3)
)
```

```
+---+---+-----+
| city|country|population|
+---+---+-----+
|Madrid| Spain| 6211000|
+---+---+-----+
```

CREACIÓN DE COLUMNAS: `withColumn()`

Este método sirve para dos cosas:

- Crear una columna nueva (si el nombre no existe)
- Sobrescribir una columna existente (si el nombre existe)

```
(df_cities
    .withColumn("million_pop", col("population")/1000000)
    .select("city", "population", "million_pop")
    .show(3)
)
```

```
+-----+-----+
| city|population|million_pop|
+-----+-----+
| Tokyo| 37785000|      37.785|
| Jakarta| 33756000|      33.756|
| Delhi| 32226000|      32.226|
+-----+
only showing top 3 rows
```

Si queremos que todas las columnas tengan un valor constante tenemos que utilizar la función lit()

```
(df_cities
    .withColumn("year", lit(2025))
    .select("city", "population", "year")
    .show(3)
)
```

```
+-----+-----+-----+
| city|population|year|
+-----+-----+-----+
| Tokyo| 37785000|2025|
| Jakarta| 33756000|2025|
| Delhi| 32226000|2025|
+-----+-----+-----+
only showing top 3 rows
```

La función lit la importamos de
pyspark.sql.functions

Podemos realizar diferentes tipos de operaciones en `withColumn()`

1. Matemáticas y redondeo

Para columnas numéricas la clase F dispone de funciones para:

- **Redondeo:** `ceil` (techo), `floor` (suelo), `round` (redondeo), `bround` (redondeo bancario al par más cercano para reducir el sesgo en sumas grandes).
- **Logaritmos y exponentiales:** `log` (logaritmo natural), `log10` (logaritmo base 10), `exp` (exponencial e^x), `factorial` (factorial de un entero).
- **Comparación entre columnas:** `max` y `min` (máximo / mínimo de una columna verticalmente), `greatest(col1, col2, ...)` y `least (col1, col2, ...)` (máximo / mínimo entre varias columnas de una misma fila)
- **Trigonometría:** `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `degrees` (radianes a grados), `radians` (grados a radianes)
- **Aleatoriedad:** `rand` (número entre 0 y 1), `randn` (número aleatorio de distribución normal o gausiana)

Ejemplo: redondear latitud y longitud a 2 decimales

Country	City	AccentCity	Region	Population	Latitude	Longitude
ad andorra la vella	Andorra la Vella	07	20430.0	42.5	1.5166667	
ad canillo	Canillo	02	3292.0	42.5666667	1.6	
ad encamp	Encamp	03	11224.0	42.533333299999995	1.5833333	
ad la massana	La Massana	04	7211.0	42.55	1.5166667	
ad les escaldes	Les Escaldes	08	15854.0	42.5	1.5333333	

```
from pyspark.sql.functions import round, col

df_redondeado = ( df_cities.withColumn("Latitude", round(col("Latitude"), 2))
                  .withColumn("Longitude", round(col("Longitude"), 2))
                )

df_redondeado.show(5)
```

Country	City	AccentCity	Region	Population	Latitude	Longitude
ad andorra la vella	Andorra la Vella	07	20430.0	42.5	1.52	
ad canillo	Canillo	02	3292.0	42.57	1.6	
ad encamp	Encamp	03	11224.0	42.53	1.58	

2. Manipulación de fechas y tiempo

Tenemos las siguientes funciones para fechas:

- `to_date / to_timestamp`: convertir strings a fechas
- `datediff`: restar fechas (devuelve número de días)
- `date_add / date_sub`: sumar/restar días a una fecha
- `year, month, dayofmonth`: extraer partes de una fecha

3. Manipulación de cadenas

Para el trabajo con cadenas hay un montón de funciones:

- **Limpieza y formato:**
 - **trim()**: elimina espacios blancos a ambos lados
 - **ltrim() / rtrim()**: elimina espacios solo a izquierda o derecha
 - **upper() / lower()**: convierte a mayúsculas/minúsculas
 - **initcap()**: tipo título
- **Unión y división:**
 - **concat(col1, col2, ...)**: une columnas sin separador. Si alguna columna es null, el resultado será null.
 - **concat_ws(sep, col1, col2, ...)**: concatena columnas usando un separador. Si hay nulos se los salta.
 - **split(col, sep)**: divide un string en una lista

Ejemplo: añadir una columna eliminando espacios en blanco antes y después

```
from pyspark.sql.functions import col, trim

df_cities = df_cities.withColumn("City_Clean", trim(col("City")))
df_cities.show(3)
```

```
+-----+-----+-----+-----+-----+-----+
|Country|      City| AccentCity|Region|Population|      Latitude|Longitude|    City_Clean|
+-----+-----+-----+-----+-----+-----+
|  ad|andorra la vella|Andorra la Vella|    07|   20430.0|        42.5|1.51666667|andorra la vella|
|  ad|       canillo|      Canillo|    02|    3292.0|        42.5666667|      1.6|       canillo|
|  ad|       encamp|      Encamp|    03|  11224.0|42.533333299999995|1.5833333|       encamp|
+-----+-----+-----+-----+-----+-----+
only showing top 3 rows
```

Ejemplo: crear una nueva columna que será la concatenación de latitud y longitud

```
from pyspark.sql.functions import col, concat, lit

df_cities = df_cities.withColumn("Coordinates",
    concat(
        round( col("Latitude"), 2 ).cast("string"),
        lit(", "),
        round( col("Longitude"), 2).cast("string")
    )
)

df_cities.select("City", "Latitude", "Longitude", "Coordinates").show(5)
```

City	Latitude	Longitude	Coordinates
andorra la vella	42.5	1.5166667	42.5, 1.52
canillo	42.5666667	1.6	42.57, 1.6
encamp	42.533333299999995	1.5833333	42.53, 1.58

- **Extracción y longitud:**

- **length()**: devuelve el número de caracteres
- **substring(col, pos, len)**: extrae una parte del texto. *Ojo: los índices comienzan por 1.*
- **instr(col, sub)**: busca la posición de una subcadena (0 si no la encuentra).

- **Reemplazo y expresiones regulares:**

- **translate(col, "ABC", "123")**: reemplazo carácter a carácter (A->1, B->2, ..) Muy rápido para limpiezas simples
- **regexp_replace(col, pattern, replacement)**: busca un patrón regexp y lo reemplaza
- **regexp_extract(col, pattern, group_idx)**: extrae algo específico que cumpla un patrón (p.e. el dominio de un email)

- **Relleno:**

- **lpad / rpad(col, len, pad)**: rellena por la izquierda o la derecha

Ejemplo: queremos detectar países cuyo código excede de 2 caracteres

```
from pyspark.sql.functions import col, length

df_errores = df_cities.filter(length(col("Country")) != 2)

print("Registros con códigos de país sospechosos:")
df_errores.select("Country", "City").show()
```

Registros con códigos de país sospechosos:

```
[Stage 11:>
+----+----+
|Country|City|
+----+----+
+----+----+
```

4. Manejo de nulos

Para el manejo de nulos Spark dispone de la función **coalesce()**.

Esta función se puede ver como una cascada de opciones que recibe varias columnas o valores y **devuelve el primero que no sea nulo**.

Permite fusionar información dispersa en varias columnas para crear una columna maestra

```
df = df.withColumn("email_final",
    coalesce(
        col("email_personal"),
        col("email_trabajo"),
        lit("No disponible")
    )
)
```

Ejemplo: quiero hacer una campaña de envío de correos, pero de algunos usuarios tengo solo el personal, de otros solo el del trabajo y de otros ninguno.

3. Tipos complejos

PySpark puede manejar **estructuras de datos** dentro de una sola celda.

Para convertir datos planos en lisas tenemos las siguientes funciones:

- **split(col, sep)**: convierte una cadena de texto en un array cortando por el delimitador.
- **array(col1, col2, ...)**: junta los valores de varias columnas del mismo registro en una sola lista.

```
( df_titanic
    .withColumn('sep_name', F.split(col("name"), " "))
    .select("sep_name", "Name")
    .show(3, truncate=False)
)
```

sep_name	Name
["Braund,, Mr., Owen, Harris"]	"Braund, Mr. Owen Harris"
["Cumings,, Mrs., John, Bradley, (Florence, Briggs, Thayer)"]	"Cumings, Mrs. John Bradley"
["Heikkinen,, Miss., Laina"]	"Heikkinen, Miss. Laina"

Una vez que tenemos una lista o array, la pregunta es ¿qué podemos hacer con él?

- **size(col)**: devuelve el número de elementos (longitud)
- **array_contains(col, value)**: devuelve True si el valor está en la lista
- **col[i]**: permite acceder a un elemento específico (ojo, aquí el primer elemento es el índice 0)

```
# 1. Filtrar productos que tengan la etiqueta "urgente"
df.filter(F.array_contains(col("array_etiquetas"), "urgente"))

# 2. Crear columna con el primer elemento de la lista
df = df.withColumn("etiqueta_principal", col("array_etiquetas")[0])

# 3. Filtrar listas vacías
df.filter(F.size(col("array_etiquetas")) > 0)
```

La otra opción es **descomponer el array** con la función **explode()**.

Esta función toma un array en una fila y **genera una nueva fila por cada elemento** del array.

```
from pyspark.sql.functions import explode

# Supongamos df:
# +-----+
# | id/ array_etiquetas|
# +-----+
# | 1/ ["cine", "cena"] |
# +-----+  
  
df_explotado = df.withColumn("etiqueta_individual", explode(col("array_etiquetas")))

# Resultado df_explotado:
# +-----+-----+
# | id/ array_etiquetas/etiqueta_individual|  
# +-----+-----+-----+-----+-----+  
# | 1/ ["cine", "cena"] /          cine/|  
# | 1/ ["cine", "cena"] /          cena/|  
# +-----+-----+-----+-----+
```

Ojo, si tenemos listas con muchos elementos el DataFrame crecerá exponencialmente

Si tenemos varias columnas con arrays, podremos realizar operaciones entre ellos tratándolos como conjuntos:

- **array_intersect**(col1, col2): elementos comunes
- **array_union**(col1, col2): unión sin duplicados
- **array_except**(col1, col2): resta de conjuntos (elementos que están en col1 y no en col2)
- **array_distinct**(col): elimina duplicados dentro de un mismo array

```
# Escenario: Comparar respuestas correctas vs respuestas del alumno
# col("clave"): ["A", "B", "C"]
# col("alumno"): ["A", "C", "D"]

# ¿Cuáles acertó?
df = df.withColumn("aciertos", F.array_intersect(col("clave"), col("alumno")))
# Resultado: ["A", "C"]
```

Por último, también podemos convertir el array en una cadena simple, por ejemplo, si queremos exportar a CSV el DataFrame.

- **array_join(col, delimiter)**: convierte lista a texto

```
df = df.withColumn("texto_final", F.array_join(col("array_etiquetas"), " - "))
```

3

DATAFRAMES: TRANSFORMACIONES AVANZADAS



3

DATAFRAMES: TRANSFORMACIONES AVANZADAS

3.1

AGREGACIONES Y AGRUPAMIENTOS

Una parte muy importante cuando trabajamos con datos son las agregaciones, donde agrupamos una serie de registros según un campo para realizar algún tipo de operación.

Ejemplo: Calcular el promedio de ventas por provincia

AGRUPAMIENTO BÁSICO (groupBy)

La forma más sencilla de agrupar es mediante `groupBy`.

Devuelve un objeto de tipo `RelationalGroupedDataset`, sobre el cual debemos llamar a una función de agregación

```
df.groupBy("categoria").count().show()
```

AGREGACIONES MÚLTIPLES (agg)

Para realizar múltiples agregaciones, personalizar nombres o usar funciones complejas, **siempre** debemos usar .agg.

```
resultado = df.groupBy("tienda").agg(  
    F.sum("ingresos").alias("total_ventas"), # Suma total  
    F.avg("ingresos").alias("ticket_promedio"), # Media  
    F.max("unidades").alias("max_unidades_vendidas"), # Máximo  
    F.count("producto").alias("num_transacciones") # Conteo )  
resultado.show()
```

Es importante usar .alias ya que, en caso contrario, Spark pondrá los nombres a las columnas

AGREGACIONES MÚLTIPLES (agg)

Para realizar múltiples agregaciones, personalizar nombres o usar funciones complejas, **siempre** debemos usar .agg.

```
resultado = ( df.groupBy("tienda")
    .agg( F.sum("ingresos").alias("total_ventas"),
          F.avg("ingresos").alias("ticket_promedio"),
          F.max("unidades").alias("max_unidades_vendidas"),
          F.count("producto").alias("num_transacciones"))
    )
resultado.show()
```

Es importante usar .alias ya que, en caso contrario, Spark pondrá los nombres a las columnas

¿Qué tipo de **funciones de agregación** tenemos disponibles? Hay múltiples funciones que podemos clasificar en varias categorías:

FUNCIONES BÁSICAS

`count(col)`

Cuenta filas **no nulas** de la columna. Si usas `count("*")` o `count(1)`, cuenta **todas** las filas (incluyendo nulos) y es más rápido.

```
( df.groupBy("Crop")
    .agg(
        F.count("*").alias("num_cultivos")
    )
).show()
```

```
+-----+
| Crop|num_cultivos|
+-----+
| ... |          |
| ... |          |
| ... |          |
| ... |          |
| ... |          |
+-----+
```

countDistinct(col)

Cuenta valores únicos exactos. Es un procedimiento costoso (shuffle)

```
( df.groupBy("Crop")
    .agg(
        F.countDistinct("Region").alias("Regiones diferentes")
    )
).show()
```

Crop	Regiones diferentes
Maize	4
Wheat	4
Rice	4
Barley	4

```
sum(col)
```

Suma valores numéricos. Ignora nulos.

```
( df.groupBy("Region")
    .agg(
        F.countDistinct("Rainfall_mm").alias("Total lluvia")
    )
).show()
```

Region	Total lluvia
Region_D	2276
Region_C	2222
Region_A	2321
Region_B	2282

```
avg(col) / mean(col)
```

Calcula el promedio aritmético. Ignora nulos

```
( df.groupBy("Region")
    .agg(
        F.mean("Temperature_C").alias("Temp. media")
    )
).show()
```

```
+-----+-----+
| Region|      Temp. media|
+-----+-----+
|Region_D| 24.87680577849118|
|Region_C| 25.106295993458712|
|Region_A| 24.86243654822333|
|Region_B| 25.06269492203117|
+-----+-----+
```

```
min(col) / max(col)
```

Valores máximo y mínimo. Funciona con números, textos y fechas

```
( df.groupBy("Region")
    .agg(
        F.max("Yield_ton_per_ha").alias("Rendimiento máximo"),
        F.min("Yield_ton_per_ha").alias("Rendimiento mínimo")
    )
).show()
```

Region	Rendimiento máximo	Rendimiento mínimo
Region_D	207.21	31.17
Region_C	205.19	32.88
Region_A	203.29	30.14
Region_B	205.23	28.45

FUNCIONES ESTADÍSTICAS PARA IA Y DATA SCIENCE

Estas funciones son críticas para **entender la distribución** de los datos antes de entrenar modelos (detección de outliers, normalización)

`stddev(col)`

La **desviación estándar** es una función fundamental para entender la **estabilidad y la distribución** de los datos.

Se representa como σ y básicamente mide la **dispersión** de los datos respecto a la media:

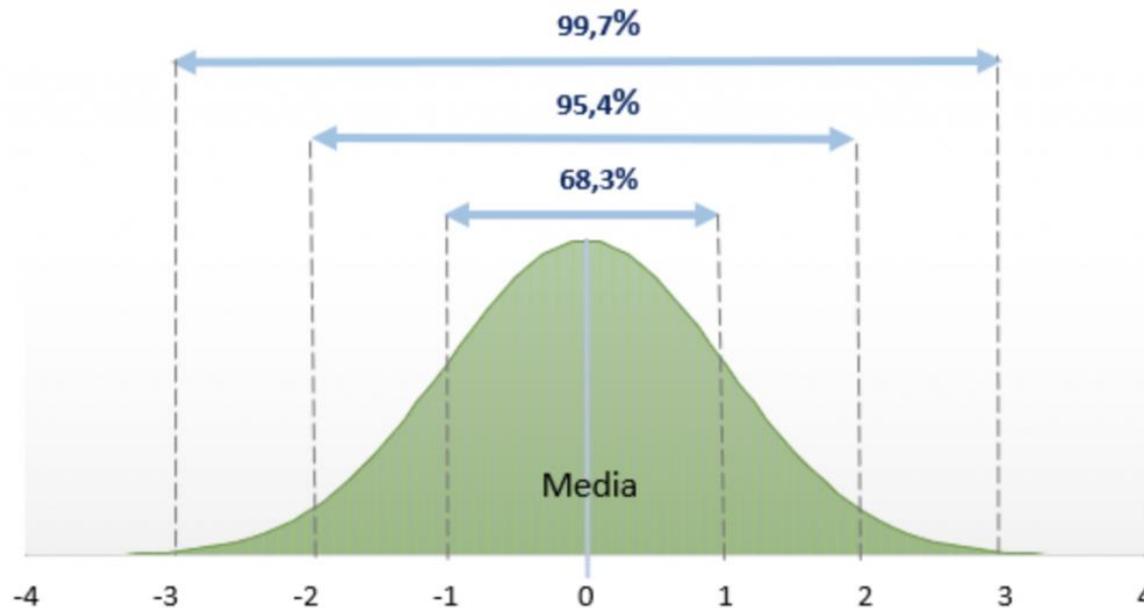
- Una **desviación baja** indica que los datos están muy agrupados cerca del promedio -> comportamiento predecible
- Una **desviación alta** indica que los datos están muy dispersos, hay mucha variabilidad o ruido -> comportamiento volátil.

La desviación estándar se utiliza habitualmente en la fase de **Feature Engineering**:

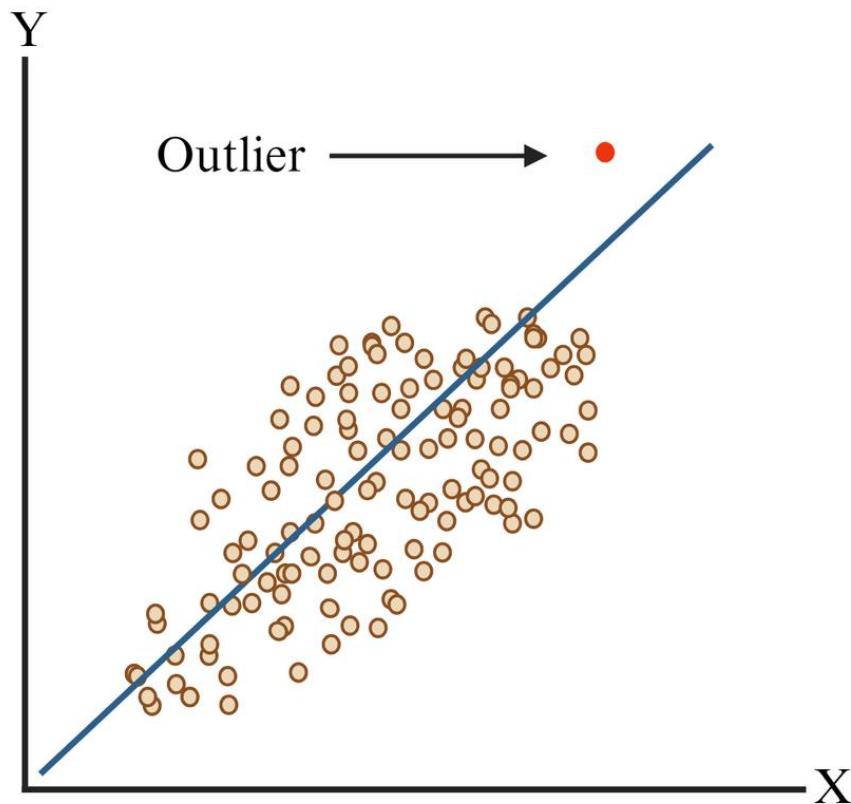
Detección de outliers (Regla de las 3 sigmas)

En una distribución normal (Campana de Gauss):

- El **68%** de los datos está a ± 1 desviación estándar de la media
- El **95%** de los datos está a ± 2
- El **99.7%** de los datos está a ± 3



Si un dato está a más de 3 veces la desviación estándar de distancia de la media, es un **outlier (anomalía)** y probablemente deba ser eliminado o investigado antes de entrenar el modelo.



Estandarización (Z-Score Scaling)

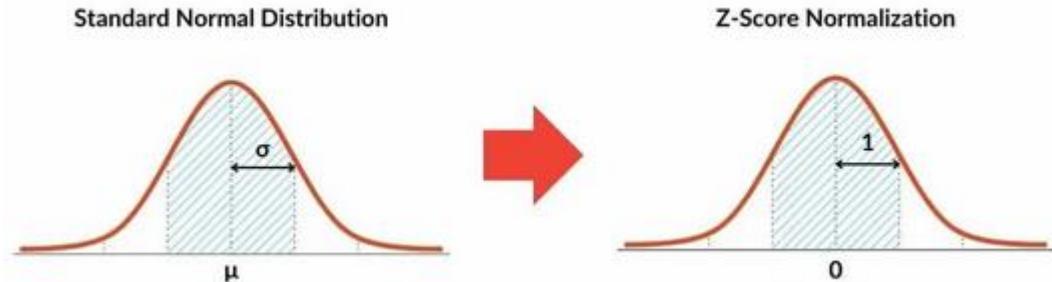
Muchos algoritmos de IA (redes neuronales, K-Means, SVM) fallan si las variables tienen escalas distintas (p.e. edad de 0-100 y salario de 0-1000000)

Para solucionarlo, se deben transformar los datos a una escala común donde **la media es 0 y la desviación estándar es 1**.

La fórmula es:

$$Z = \frac{x - \mu}{\sigma}$$

Score Mean
 ↓
 ↓
 σ SD



```
stats = ( df.groupBy("Crop")
            .agg(
                F.mean("Temperature_C").alias("media"),
                F.stddev("Temperature_C").alias("desviacion")
            )
        )

# Unimos Las estadísticas con Los datos originales
df_con_stats = df.join(stats, "Crop") ← Hago un join para que estos
# 3. Calculamos el Z-Score para detectar anomalías
# Z-Score = (Valor - Media) / Desviación
df_analisis = df_con_stats.withColumn(
    "z_score",
    (F.col("Temperature_C") - F.col("media")) / F.col("desviacion")
)

df_analisis.select("Crop", "Temperature_C", "media", "desviacion", "z_score")
```

Calculo la media y la desviación estándar para cada tipo de cultivo

Hago un join para que estos datos estén disponibles en el dataframe

Crop	Temperature_C	media	desviacion	z_score
Maize	30.0	24.91130399677028	5.809848453283212	0.8758741375352124
Maize	15.5	24.91130399677028	5.809848453283212	-1.6198880353672294
Maize	28.1	24.91130399677028	5.809848453283212	0.5488432321204099
Maize	32.9	24.91130399677028	5.809848453283212	1.3750265721157005
Maize	20.6	24.91130399677028	5.809848453283212	-0.7420682366222323
Maize	31.4	24.91130399677028	5.809848453283212	1.116844278367172
Maize	32.2	24.91130399677028	5.809848453283212	1.2545415016997212
Maize	32.0	24.91130399677028	5.809848453283212	1.2201171958665835

Este sería el valor que debería reemplazar a la temperatura si fuéramos a usar estos datos en un algoritmo de ML

```
stats2 = ( df_analisis.groupBy("Crop")
            .agg(
                F.mean("z_score").alias("media"),
                F.stddev("z_score").alias("desviacion")
            )
)
stats2.show(truncate=False)
```

Crop	media	desviacion
Maize	1.3748478424445721E-15	0.9999999999999991
Wheat	4.6878564318142675E-15	0.9999999999999989
Rice	-7.48306078861112E-15	1.0000000000000002
Barley	6.627201342235676E-15	0.9999999999999981

Podemos comprobar que, efectivamente, la media de los nuevos datos es 0 y la desviación estándar 1

```
variance(col)
```

La **varianza** (σ^2) es el promedio de los cuadrados de las diferencias con la media.

$$\text{Varianza} = \sum_{I=1}^n \frac{(x_i - \bar{x})^2}{n}$$

Es una medida difícil de interpretar para humanos (estoy elevando al cuadrado, ¿qué significan, por ejemplo, grados cuadrados?) pero tiene mucho significado para las máquinas. Sus ventajas son:

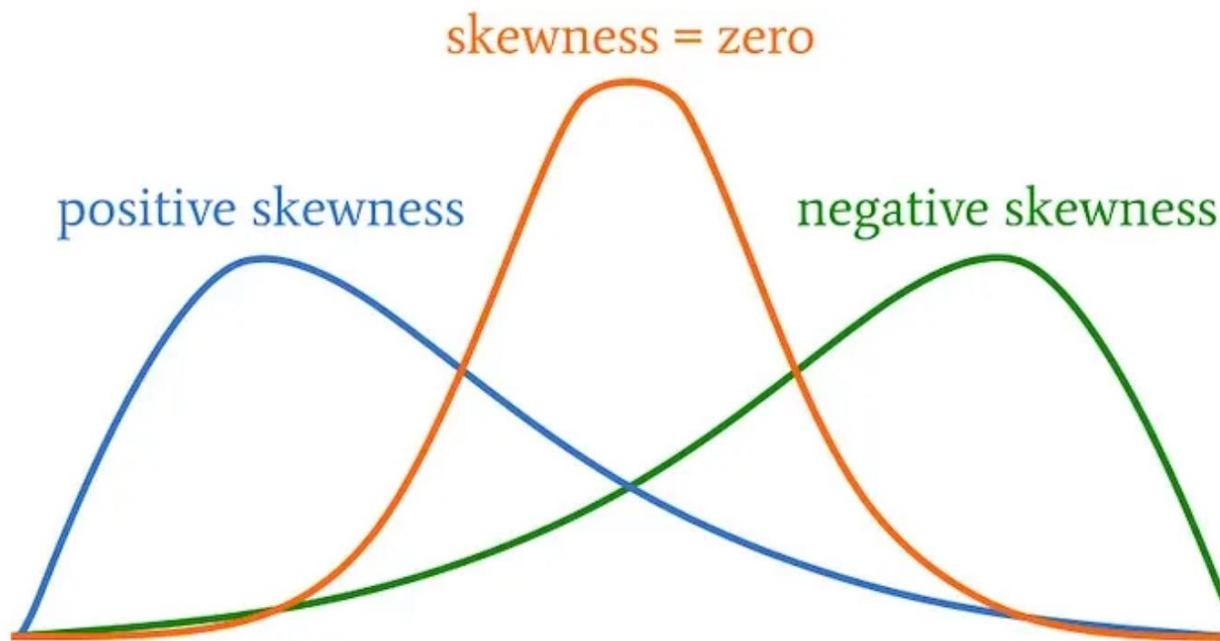
- **Eliminar negativos**: al elevar al cuadrado todos los valores son positivos
- **Castigar a los *outliers***, que destacan mucho más cuando se elevan al cuadrado.

La varianza se utiliza para:

- **Evaluación de los modelos:** si un modelo tiene varianza alta significa que ha memorizado el ruido de los datos de entrenamiento y que es inestable: si cambias un poco los datos de entrada, la predicción cambia radicalmente. Lo ideal es buscar **modelos con baja varianza (estables) y bajo sesgo (precisos)**
- **Selección de características:** si una columna tiene varianza 0 significa que todos los valores son iguales, por lo que no aporta información. Lo ideal es eliminar todas las columnas con varianzas cercanas a 0
- **PCA (Análisis de Componentes Principales):** cuando se quieren reducir dimensiones se usa PCA, que elige las columnas donde hay máxima varianza (la varianza es sinónimo de información).

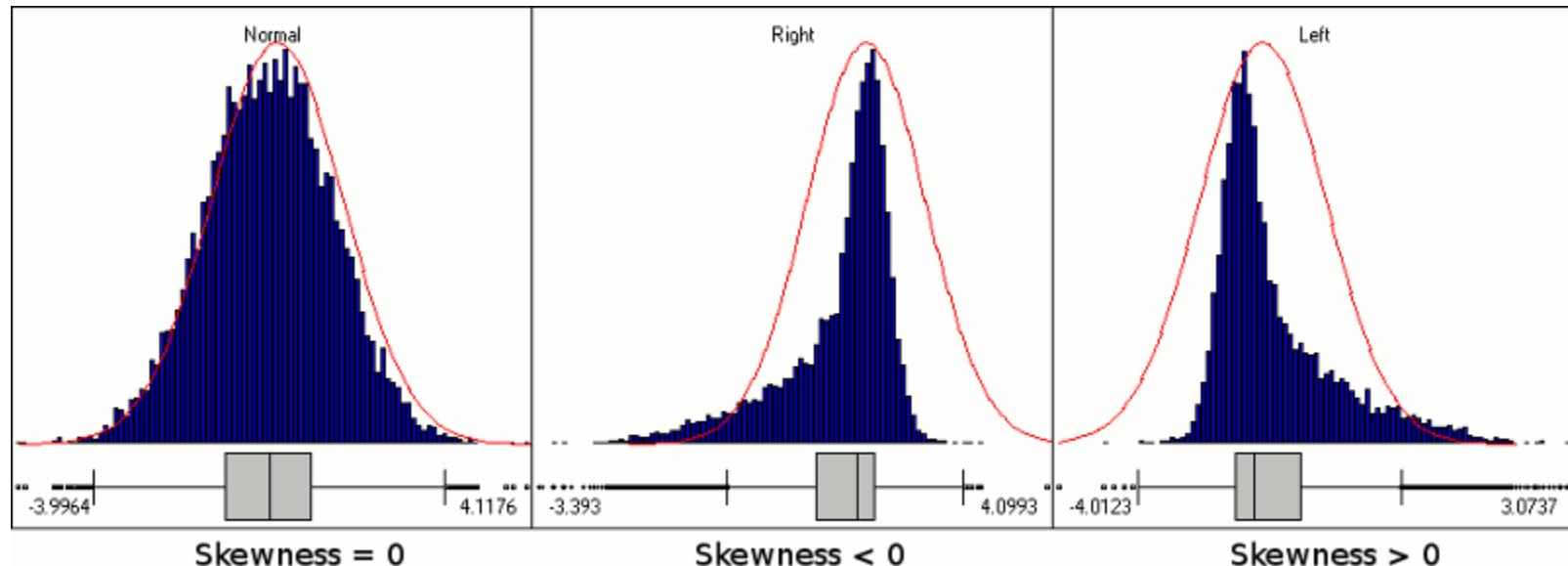
skewness(col)

Es la **asimetría** de la distribución. Si es >1 o <-1 la distribución de datos no es normal y se necesita realizar algún tipo de transformación



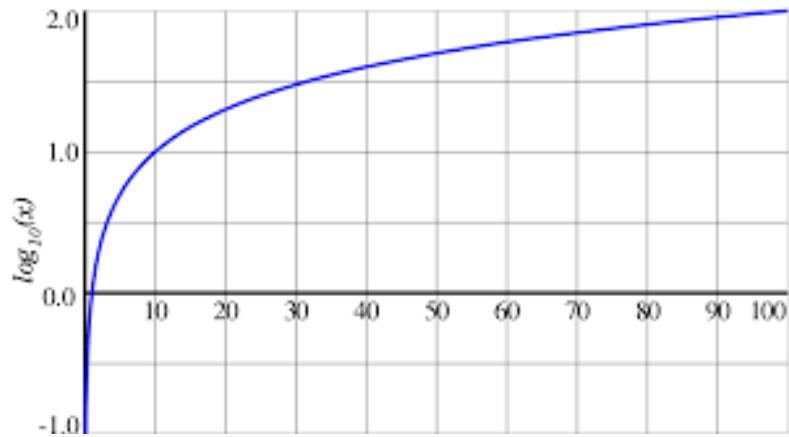
La mayoría de los modelos de ML asumen que los datos tienen forma de **campana perfecta** (distribución normal). Si la *skewness* es alta, el modelo aprenderá **sesgado**.

Lo ideal es que tenga un valor próximo a 0, donde la media y la mediana (*el valor que divide el dataset en dos mitades*) son iguales.



La forma de solucionar el *skewness* de los datos suele ser mediante la aplicación del **logaritmo** base 10 (recuerda que le logaritmo es la operación matemática inversa de la exponenciación).

El logaritmo reduce las distancias de forma **no lineal**. Cuanto más grande es el número, más fuerte lo castiga.



```
( df.groupBy("Crop")
    .agg(
        F.skewness("Temperature_C").alias("sesgo"),
    )
).show(truncate=False)
```

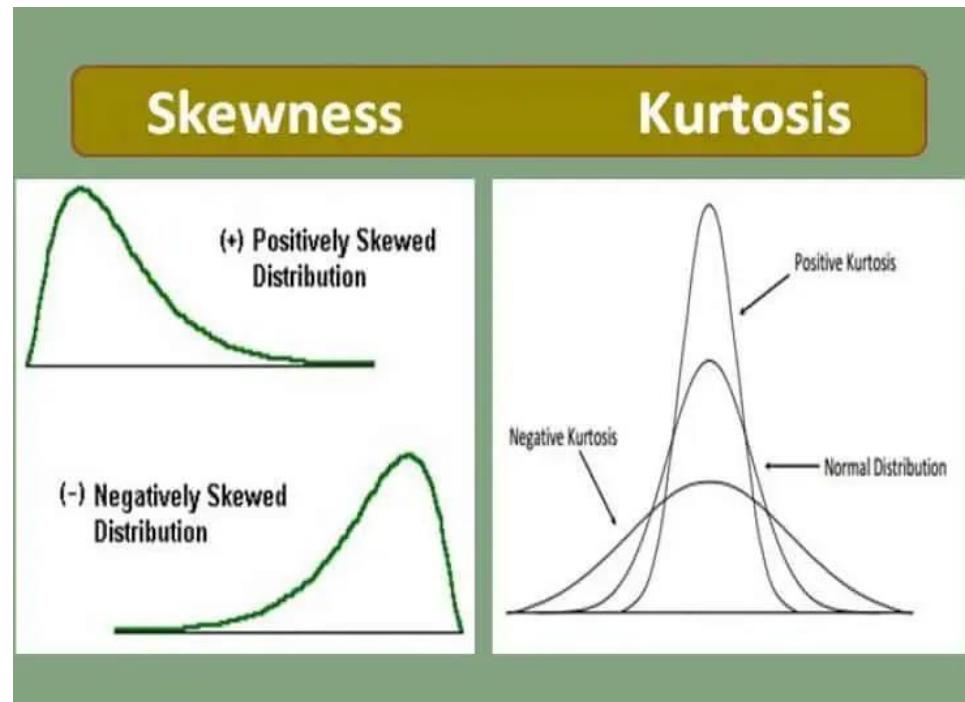
[Stage 87:>

Crop	sesgo
Maize	0.04038400226515987
Wheat	-0.023524078832991815
Rice	0.012614452271872583
Barley	0.009337968579507578

kurtosis(col)

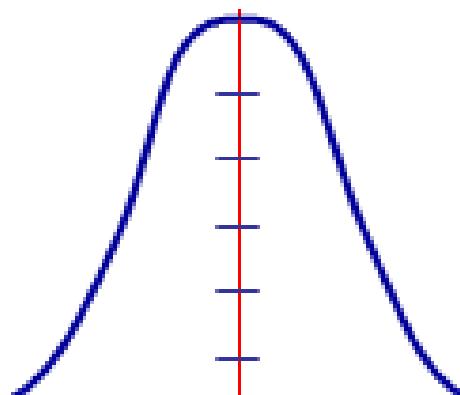
La **Curtosis** es el otro elemento (junto a la media, la desviación y el skewness) que debemos mirar.

Si el skewness nos dice hacia donde se inclina la montaña, la **curtosis** nos dice cuán gordas con las colas de esa montaña, ya que mide el *apuntamiento* y el grosor de las colas respecto a una distribución normal.

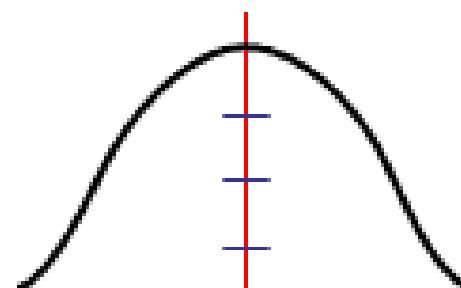


Los valores que podemos encontrar son:

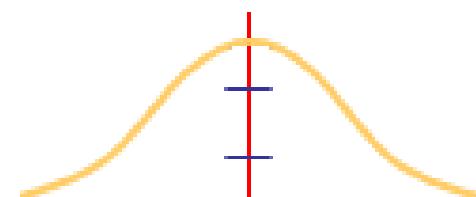
- **Valor próximo a 0 (mesocúrtica):**
 - Es el valor normal.
 - Los eventos ocurren con la frecuencia esperada. Ni muchas sorpresas ni pocas.



Leptocúrtica



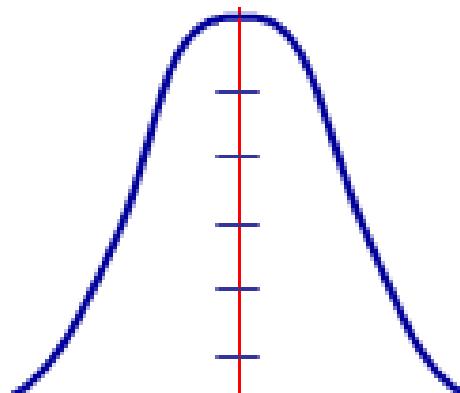
Mesocúrtica



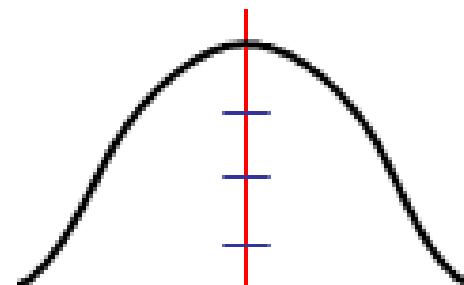
Platicúrtica

- **Valor superior a 0 (leptocúrtica):**

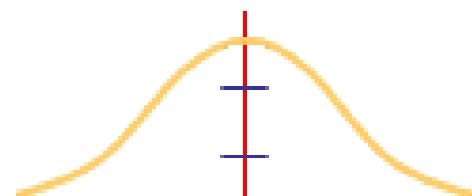
- Pico alto y agudo en el centro, y colas largas y gordas en los extremos.
- La mayoría de los datos están concentrados en el promedio. Pero hay probabilidad anormalmente alta de encontrar valores muy extremos
Ejemplo: criptomonedas, la mayoría de los días no pasa nada, pero un día cae un 40%
- Es necesario limpiar outliers agresivamente



Leptocúrtica



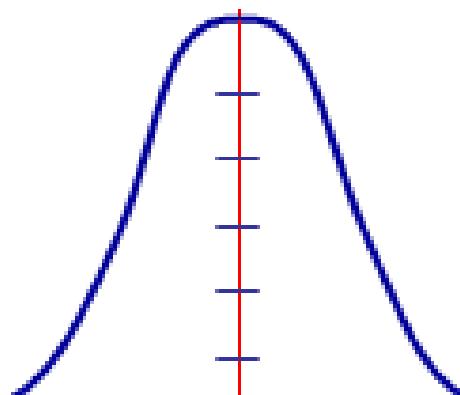
Mesocúrtica



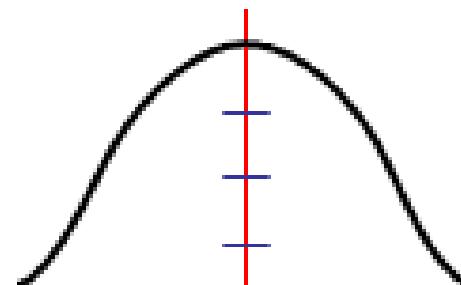
Platicúrtica

- **Valor inferior a 0 (plasticúrtica):**

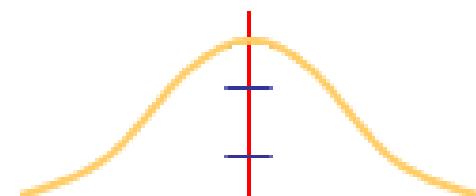
- La cima es plana y ancha, y las colas muy delgadas
- Los datos están muy dispersos alrededor de la media
- No hay valores extremos. Todo es zona gris
- Los datos no tienen patrón claro. A los algoritmos de clustering (como K-means) les cuesta encontrar el centro



Leptocúrtica



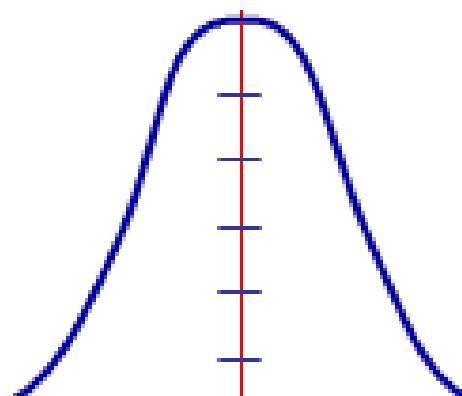
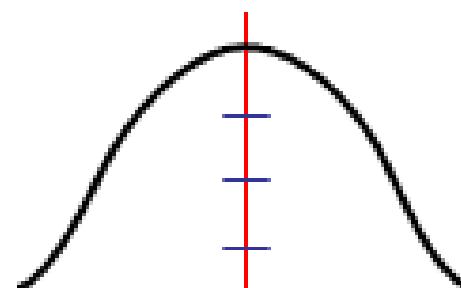
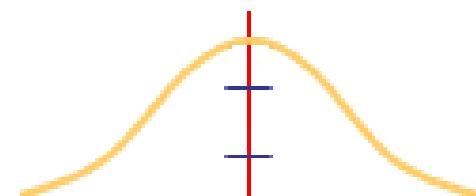
Mesocúrtica



Platicúrtica

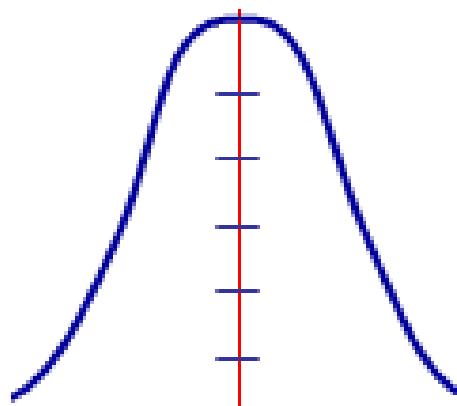
Interpretación de los valores positivos de curtosis:

- **0 a 1 (zona segura)**. La distribución es casi normal.
- **1 a 3 (Precaución)** Es claramente leptocúrtica. Hay más outliers de los que predice la campana de Gauss, pero suelen ser manejables
- **>3 (Curtosis alta)** La curtosis es muy alta. La probabilidad de eventos extremos (5 ó 6 desviaciones estándar) es muy alta
- **>10 (Curtosis extrema)**: suele indicar errores de medición

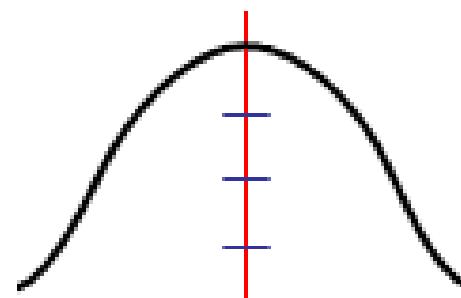
**Leptocúrtica****Mesocúrtica****Platicúrtica**

Interpretación de los valores negativos de curtosis:

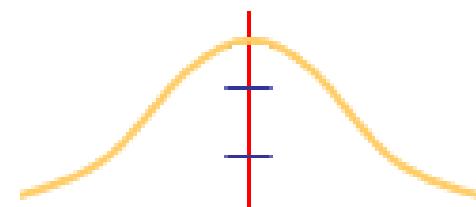
- **0 a -0.5 (casi normal).** Es una campana de Gauss un poco aplastada, pero sigue comportándose bien.
- **-0.5 a -1.0 (moderado).** La cima desaparece y empieza a parecer una colina suave. Los datos están dispersos, pero no extremos.
- **-1.2 (meseta).** Valor de referencia clásico que refleja una distribución uniforme.
- **-2.0 (binario).** Valor mínimo posible. Distribución de Bernoulli con $p=0.5$ (p.e. lanzar moneda: 50% caras y 50% cruces)



Leptocúrtica



Mesocúrtica



Platicúrtica

```
( df.groupBy("Crop")
    .agg(
        F.kurtosis("Temperature_C").alias("curtosis"),
    )
).show(truncate=False)
```

Crop	curtosis
Maize	-1.20149109209531
Wheat	-1.1794781330751978
Rice	-1.2109153977374925
Barley	-1.2315894519670176

Observa que aquí los valores son todos próximos a 1.2, lo que denota una distribución uniforme. Casi seguro que debido a que este se trata de un **dataset sintético**.

FUNCIONES DE APROXIMACIONES

Estas funciones son relevantes cuando la velocidad es más importante que la precisión exacta

`approx_count_distinct(col, rsd)`

Realiza un conteo único estimado utilizando HyperLogLog.

Mucho más rápido que `countDistinct(col)`. `rsd` es el error permitido (por defecto 0.05)

```
approx_percentile(col, rsd)
```

Calcula percentiles (mediana, Q1, Q3)

Calcular la mediana es muy costoso en distribuido. Esto lo hace viable.

FUNCIONES DE SELECCIÓN POSICIONAL

Estas funciones son relevantes cuando la velocidad es más importante que la precisión exacta

```
first(col, ignorenulls=False)
```

3

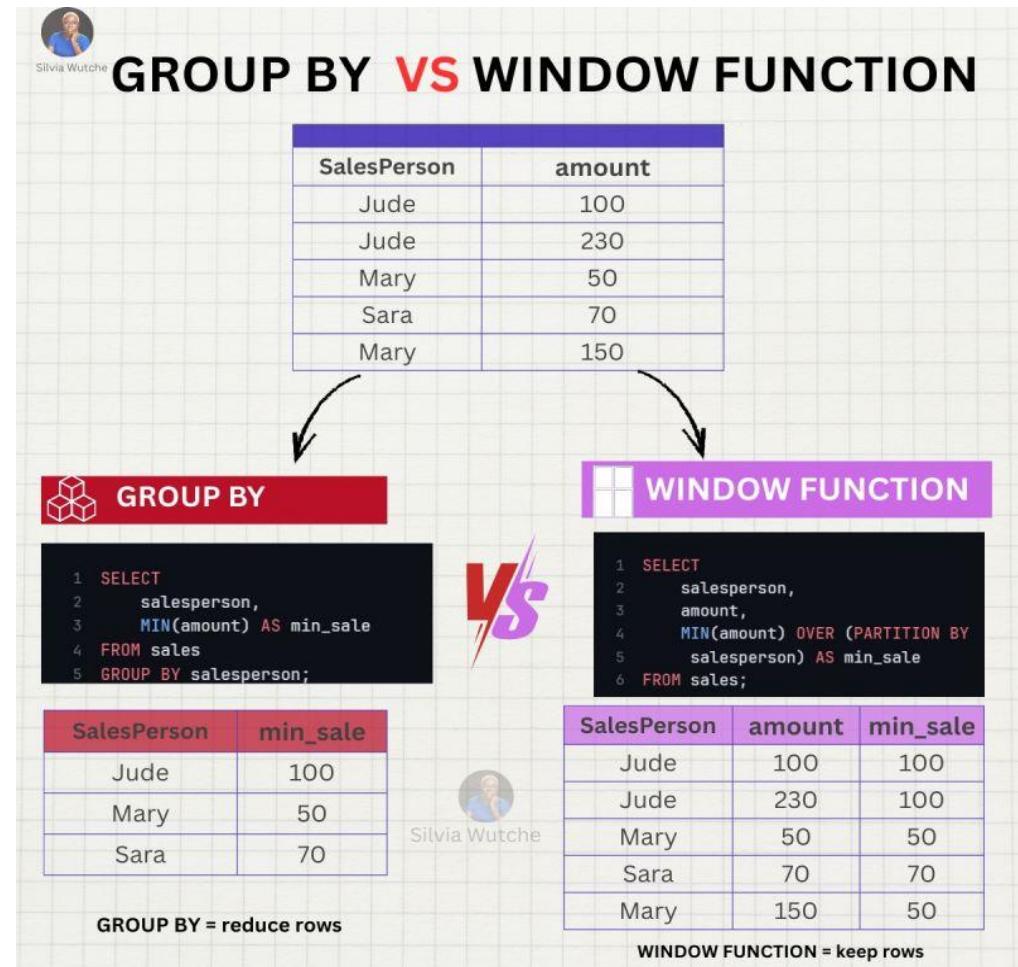
DATAFRAMES: TRANSFORMACIONES AVANZADAS

3.2

FUNCIONES DE VENTANA

Las **funciones de ventana** son una herramienta imprescindible cuando estamos trabajando con **series temporales y datos secuenciales**.

Mientras que con `groupBy` colapsamos las filas, perdiendo el detalle de cada registro individual, las funciones de ventana **calculan un valor basado en un grupo de filas** (ventana) y añade el resultado como una **nueva columna** en cada fila.



Ejemplo: ranking de cultivos dentro de una misma región

```
from pyspark.sql.functions import rank, desc  
  
ventanaRegional = ( Window  
    .partitionBy("Region")  
    .orderBy(desc("Yield_ton_per_ha"))  
)  
  
df.select(  
    "Region",  
    "Crop",  
    "Yield_ton_per_ha",  
    rank().over(ventanaRegional).alias("Ranking_Regional")  
).show()
```

Dividimos los datos por regiones. Cada región tendrá su propia ventana

En **cada ventana** ordenamos los datos de forma descendente según Yield_ton_per_ha

La función **rank** asigna números de posición sobre los registros de la ventana

Es necesario usar la función **over()** para indicar sobre qué ventanas vamos a aplicar la función rank()

Este sería el resultado, donde se ha añadido una columna nueva con el ranking de cada registro del DataFrame dentro de la región

Region	Crop	Yield_ton_per_ha	Ranking_Regional
Region_A	Maize	203.29	1
Region_A	Wheat	203.1	2
Region_A	Rice	202.99	3
Region_A	Rice	201.96	4
Region_A	Rice	200.59	5
Region_A	Wheat	200.49	6
Region_A	Rice	199.64	7
Region_A	Rice	198.81	8
Region_A	Barley	198.56	9
Region_A	Barley	195.98	10

En el ejemplo anterior hemos visto la función **partitionBy()**, que indica la columna sobre la que se harán las ventanas, y **orderBy()** para ordenar todos los elementos dentro de una ventana.

Falta una tercera función, que es **rowsBetween(inicio, fin)** que define el **marco de la ventana**. Esto le dice a Spark: “*Para calcular el dato de esta fila, ¿cuántas filas hacia atrás o hacia adelante tengo que incluir?*”

Los posibles valores de inicio y fin pueden ser:

- **Window.unboundedPreceding**: desde la primera fila del grupo
- **Window.CurrentRow**: la fila donde estoy ahora mismo
- **Window.unboundedFollowing**: hasta la última fila del grupo
- **Números enteros**: -1 (*una atrás*), 1 (*una adelante*), ...

Si no se pone ningún valor el valor por defecto es desde el principio hasta el registro actual.

Una vez que hemos creado las ventanas, el siguiente paso es aplicar algún tipo de función sobre dicha ventana.

Algunas funciones disponibles son:

Window Functions

Aggregate

- AVG()
- MAX()
- MIN()
- COUNT()

Ranking

- RANK()
- DENSE_RANK()
- ROW_NUMBER()
- PERCENT_RANK()
- NTILE()

Value

- LAG()
- LEAD()
- CUME_DIST()

FUNCIÓN avg()

La función **avg()** calcula la **media aritmética** de una columna numérica dentro del marco de filas que tú definas.

El comportamiento cambia radicalmente según cómo esté definida la ventana:

- Si solo usas `partitionBy()` calcula el promedio total del grupo y repite ese mismo número en todas las filas del grupo (es un valor **estático**)
- Si usas `orderBy()` (sin `rowsBetween()`) calcula el **promedio acumulado** desde el inicio del grupo hasta la fila actual (el valor cambia en cada fila)
- Si usas `rowsBetween()` calcula el **promedio móvil** solo de las filas vecinas que especifiques

Vamos a verlo con un ejemplo:

```
from pyspark.sql import SparkSession
from pyspark.sql.window import Window
from pyspark.sql.functions import avg, col

data = [
    ("A", 1, 10.0),
    ("A", 2, 20.0),
    ("A", 3, 30.0),
    ("A", 4, 40.0),
    ("A", 5, 50.0),
    ("B", 1, 100.0),
    ("B", 2, 200.0),
    ("B", 3, 400.0)
]

columns = ["Region", "Dia", "Valor"]

df = spark.createDataFrame(data, columns)

df.show()
```

Region	Dia	Valor
A	1	10.0
A	2	20.0
A	3	30.0
A	4	40.0
A	5	50.0
B	1	100.0
B	2	200.0
B	3	400.0

Caso 1: solo `partitionBy()`. Valor estático para el grupo.

```
ventana_fija = Window.partitionBy("Region")  
  
df_fijo = ( df.withColumn("Promedio_Total_Region", avg("Valor")  
                         .over(ventana_fija))  
            )  
  
df_fijo.show()
```

Region	Dia	Valor	Promedio_Total_Region
A	1	10.0	30.0
A	2	20.0	30.0
A	3	30.0	30.0
A	4	40.0	30.0
A	5	50.0	30.0
B	1	100.0	233.333333333334
B	2	200.0	233.333333333334
B	3	400.0	233.333333333334

Mismo valor en todos los registros de cada grupo

Caso 2: solo `partitionBy()` y `orderBy()`. Promedio acumulado

```
# Ventana por defecto
ventana_acumulada = Window.partitionBy("Region").orderBy("Dia")

df_acumulado = ( df.withColumn("Promedio_Hasta_Ahora", avg("Valor")
                           .over(ventana_acumulada))
                  )

df_acumulado.show()
```

Region	Dia	Valor	Promedio_Hasta_Ahora
A	1	10.0	10.0
A	2	20.0	15.0
A	3	30.0	20.0
A	4	40.0	25.0
A	5	50.0	30.0
B	1	100.0	100.0
B	2	200.0	150.0
B	3	400.0	233.3333333333334

Cada promedio se calcula a partir de los valores acumulados desde el principio hasta él

Caso 3: con rowsBetween(). Promedio móvil

```
# Ventana: Por región, ordenado por día, mirando 1 atrás y la actual
ventana_movil = ( Window.partitionBy("Region")
                  .orderBy("Dia")
                  .rowsBetween(-1, 0)
                )

df_movil = ( df.withColumn("Media_Movil_2dias", avg("Valor")
                           .over(ventana_movil))
              )

df_movil.show()
```

Region	Dia	Valor	Media_Movil_2dias
A	1	10.0	10.0
A	2	20.0	15.0
A	3	30.0	25.0
A	4	40.0	35.0
A	5	50.0	45.0
B	1	100.0	100.0
B	2	200.0	150.0
B	3	400.0	300.0

FUNCIÓNES max() y min()

Añaden una nueva columna con el valor **máximo o mínimo**, pudiendo ser del total del grupo (si no usas orderBy()) o desde el comienzo de la ventana si se han ordenado los datos.

```
ventana = ( Window
            .partitionBy("departamento")
            #.orderBy(col("salario").desc())
          )
(df.withColumn(
  "max",
  max("salario").over(ventana)
)
.show()
)
```

departamento	empleado	salario	max
IT	Carlos	5000	5000
IT	Laura	5000	5000
IT	Juan	4000	5000
Ventas	Ana	3000	3000
Ventas	Luis	3000	3000
Ventas	Maria	2500	3000
Ventas	Pedro	2000	3000

En cada grupo, todos tienen el mismo valor

```
ventana = ( Window
            .partitionBy("departamento")
            .orderBy(col("salario").desc())
        )
(df.withColumn(
    "min",
    min("salario").over(ventana)
)
.show()
)
```

departamento	empleado	salario	min
IT	Carlos	5000	5000
IT	Laura	5000	5000
IT	Juan	4000	4000
Ventas	Ana	3000	3000
Ventas	Luis	3000	3000
Ventas	María	2500	2500
Ventas	Pedro	2000	2000

En este caso se va arrastrando el valor mínimo según se avance

FUNCIÓN count()

La función count() **cuenta el número de registros**. Nuevamente, su comportamiento es diferente en función de si ordenamos los datos o no:

- Si no los ordenamos pondrá en todas las filas el número total de registros de ese grupo.
- Si están ordenadas, calcula un conteo acumulado dentro del grupo

```
ventana = ( Window
            .partitionBy("departamento")
            #.orderBy(col("salario").desc())
          )
(df.withColumn(
  "count",
  count("salario").over(ventana)
)
.show()
)
```

departamento	empleado	salario	count
IT	Carlos	5000	3
IT	Laura	5000	3
IT	Juan	4000	3
Ventas	Ana	3000	4
Ventas	Luis	3000	4
Ventas	Maria	2500	4
Ventas	Pedro	2000	4

Todas las filas contienen el mismo valor, que es el número de registros del grupo

```
ventana = ( Window
              .partitionBy("departamento")
              .orderBy(col("salario").desc())
            )
(df.withColumn(
    "count",
    count("salario").over(ventana)
  )
.show()
)
```

departamento	empleado	salario	count
IT	Carlos	5000	2
IT	Laura	5000	2
IT	Juan	4000	3
Ventas	Ana	3000	2
Ventas	Luis	3000	2
Ventas	María	2500	3
Ventas	Pedro	2000	4

??

FUNCIÓN rank()

Esta función **requiere** que haya una ordenación.

Establece un ranking en función de la posición de cada registro con respecto a la ordenación.

Algo relevante es la gestión de valores repetidos: en ese caso, asignaría la misma posición a ambos valores, pero saltaría el siguiente.

```
ventana = ( Window
            .partitionBy("departamento")
            .orderBy(col("salario").desc())
        )
(df.withColumn(
    "rank",
    rank().over(ventana)
)
.show()
)
```

departamento	empleado	salario	rank
IT	Carlos	5000	1
IT	Laura	5000	1
IT	Juan	4000	3
Ventas	Ana	3000	1
Ventas	Luis	3000	1
Ventas	María	2500	3
Ventas	Pedro	2000	4

Aquí hay dos registros con el mismo valor, así que les asigna el mismo número y se salta el siguiente

FUNCIÓN **dense_rank()**

Esta función es muy similar a rank(), diferenciándose en la gestión de los valores repetidos: si hay valores repetidos asigna la misma posición, pero no se salta luego ningún valor.

Ejemplo: empate en la primera posición.

- **rank()**: la secuencia sería 1, 1, 3, 4
- **dense_rank()**: en este caso sería 1, 1, 2, 3

```
ventana = ( Window
            .partitionBy("departamento")
            .orderBy(col("salario").desc())
        )
(df.withColumn(
    "dense_rank",
    dense_rank().over(ventana)
)
.show()
)
```

departamento	empleado	salario	dense_rank
IT	Carlos	5000	1
IT	Laura	5000	1
IT	Juan	4000	2
Ventas	Ana	3000	1
Ventas	Luis	3000	1
Ventas	Maria	2500	2
Ventas	Pedro	2000	3

Asigna el mismo valor a los que están empuestos, pero no se salta luego ninguno.

FUNCIÓN row_number()

El propósito principal de esta función es asignar un número entero, único y secuencial a cada fila dentro de una partición, comenzando siempre desde el 1. Al igual que rank() y dense_rank(), requiere obligatoriamente el uso de orderBy

```
ventana = ( Window
            .partitionBy("departamento")
            .orderBy(col("salario").desc())
          )
( df.withColumn(
    "row_number",
    row_number().over(ventana)
  )
.show()
)
```

	departamento	empleado	salario	row_number
IT	IT	Carlos	5000	1
	IT	Laura	5000	2
	IT	Juan	4000	3
Ventas	Ventas	Ana	3000	1
Ventas	Ventas	Luis	3000	2
Ventas	Ventas	Maria	2500	3
Ventas	Ventas	Pedro	2000	4

Numera de forma única
los registros de cada
grupo

FUNCIÓN percent_rank()

La función percent_rank() evalúa la **posición relativa de una fila dentro de su grupo**, devolviendo un número decimal que va exactamente **desde 0.0 hasta 1.0**.

Usa la fórmula: rango -1 / número registros -1

```
ventana = ( Window
            .partitionBy("departamento")
            .orderBy(col("salario").desc())
        )
(df.withColumn(
    "percent_rank",
    percent_rank().over(ventana)
)
.show()
```

departamento	empleado	salario	percent_rank
IT	Carlos	5000	0.0
IT	Laura	5000	0.0
IT	Juan	4000	1.0
Ventas	Ana	3000	0.0
Ventas	Luis	3000	0.0
Ventas	Maria	2500	0.6666666666666666
Ventas	Pedro	2000	1.0

No hay valores superiores a 5000, luego es 0%

Hay otros 3 valores, de los que 2 están por debajo

FUNCIÓN ntile()

Divide las filas de cada grupo en un **número específico de partes iguales** basándose en el **orden**.

Permite dividir los grupos en cuartiles, deciles o percentiles.

```
ventana = ( Window
            .partitionBy("departamento")
            .orderBy(col("salario").desc())
            )
( df.withColumn(
    "ntile",
    ntile(2).over(ventana)
    )
.show()
)
```

departamento	empleado	salario	ntile
IT	Carlos	5000	1
IT	Laura	5000	1
IT	Juan	4000	2
Ventas	Ana	3000	1
Ventas	Luis	3000	1
Ventas	Maria	2500	1
Ventas	Victor	2200	2
Ventas	Pedro	2000	2

FUNCIÓN lag()

Esta función permite mirar hacia atrás y obtener el valor de una fila anterior dentro del grupo sin tener que hacer cruces o joins complejos de tablas.

Acepta tres parámetros:

- **Valor** (columna) que quieras traer de la fila anterior
- **Salto** o cuántas filas hacia atrás quieras mirar
- **Valor por defecto** que pasa cuando no hay nada (primeras filas). Por defecto es null

```
data = [
    ("Norte", "2024-01-01", 100),
    ("Norte", "2024-01-02", 150), # Subió 50
    ("Norte", "2024-01-03", 120), # Bajó 30
    ("Norte", "2024-01-04", 200) # Subió 80
]
columns = ["region", "fecha", "ventas"]
df = spark.createDataFrame(data, columns)
```

```
ventana_tiempo = ( Window.partitionBy("region")
                    .orderBy("fecha")
                  )
# Aplicamos Lag para traer la venta de ayer
df_resultado = df.withColumn(
    "venta_ayer", lag("ventas", 1).over(ventana_tiempo)
).withColumn(
    "crecimiento_diario", col("ventas") - col("venta_ayer")
)
df_resultado.show()
```

region	fecha	ventas	venta_ayer	crecimiento_diario
Norte	2024-01-01	100	NULL	NULL
Norte	2024-01-02	150	100	50
Norte	2024-01-03	120	150	-30
Norte	2024-01-04	200	120	80

FUNCIÓN lead()

Esta función es análoga a lag(), pero en lugar de mirar filas anteriores mira en las siguientes filas.

FUNCIÓN cume_dist()

Indica qué porcentaje del grupo tiene un **valor igual o menor al de la fila actual**.

Devuelve un valor entre 0 y 1 y requiere un que los grupos estén ordenados.

La fórmula que usa es:

num. registros con valor menor al actual / total registros

```
data = [
    ("Matemáticas", "Juan", 60),
    ("Matemáticas", "Pedro", 70),
    ("Matemáticas", "María", 80),
    ("Matemáticas", "Luis", 80),
    ("Matemáticas", "Ana", 90)
]
columns = ["materia", "alumno", "nota"]
df = spark.createDataFrame(data, columns)
```

```
ventana_distribucion = Window.partitionBy("materia").orderBy("nota")
df_resultado = (df
    .withColumn(
        "cume_dist",
        cume_dist().over(ventana_distribucion)
    )
    .withColumn(
        "percent_rank",
        percent_rank().over(ventana_distribucion)
    )
)
df_resultado.show()
```

materia	alumno	nota	cume_dist	percent_rank
Matemáticas	Juan	60	0.2	0.0
Matemáticas	Pedro	70	0.4	0.25
Matemáticas	María	80	0.8	0.5
Matemáticas	Luis	80	0.8	0.5
Matemáticas	Ana	90	1.0	1.0

4

¿CÓMO CONSTRUIR UN SISTEMA DE RECOMENDACIÓN?



4

¿CÓMO CONSTRUIR
UN SISTEMA DE
RECOMENDACIÓN?

4.1

SISTEMAS DE
RECOMENDACIÓN

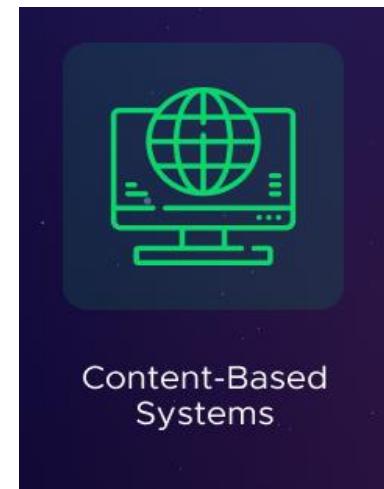
Para finalizar con Spark, vamos a ver cómo crear un **motor de recomendación** (películas, música, productos, ...)

Un **motor de recomendación** es un sistema de inteligencia artificial y *machine learning* que **analiza datos** de comportamiento de usuario para **sugerir productos**, servicios o contenidos personalizados y relevantes.



Hay diferentes tipos de motores de recomendación:

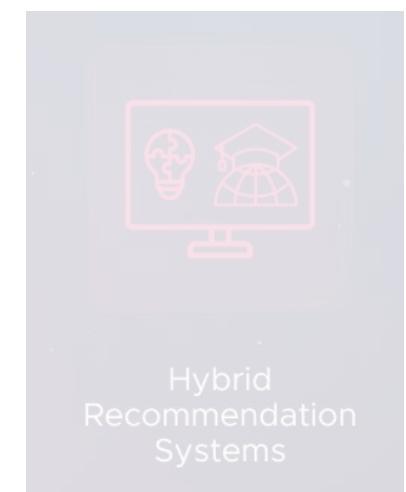
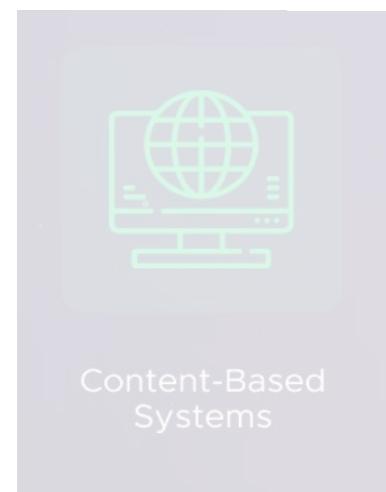
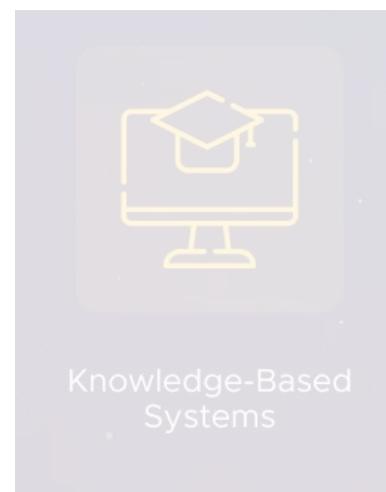
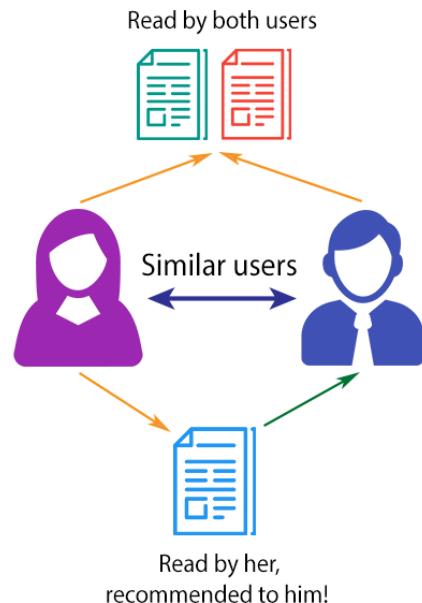
- Sistemas de filtrado colaborativo
- Sistemas basados en el contenido
- Sistemas basados en el conocimiento
- Sistemas híbridos



FILTRADO COLABORATIVO

Sugiere elementos basándose en las preferencias de usuarios similares.

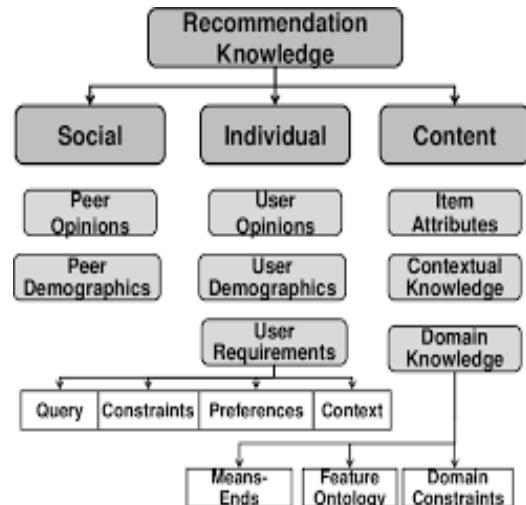
Si el usuario A y B tienen gustos parecidos, recomienda a A lo que le gustó a B.



BASADO EN EL CONOCIMIENTO

Recomiendan productos complejos basándose en **reglas lógicas** y el **conocimiento específico** de expertos sobre el dominio.

En lugar de usar históricos previos, analizan si las características técnicas de un producto satisfacen las necesidades expresadas por el usuario.



Collaborative Filtering Systems



Knowledge-Based Systems



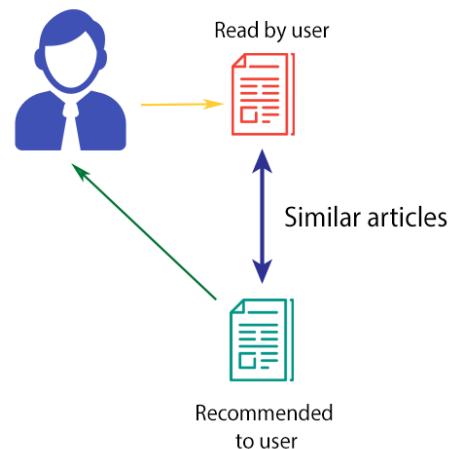
Content-Based Systems



Hybrid Recommendation Systems

FILTRADO BASADO EN CONTENIDO

Recomienda productos similares a los que el usuario ya ha mostrado interés o comprado, analizando los atributos del artículo.



Collaborative Filtering Systems



Knowledge-Based Systems



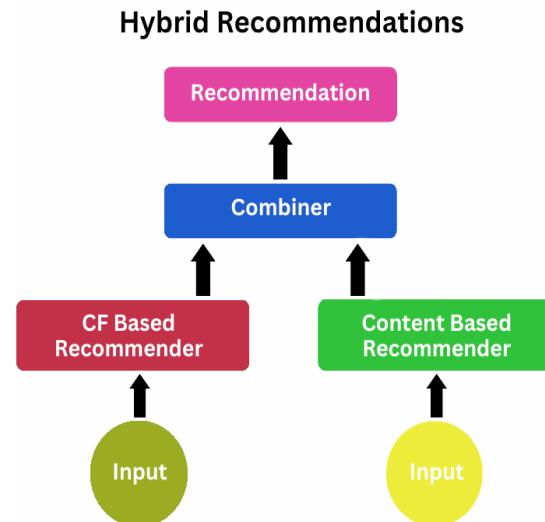
Content-Based Systems



Hybrid Recommendation Systems

SISTEMAS HÍBRIDOS

Combinan varios métodos para mejorar la precisión y evitar limitaciones como recomendar contenido nuevo sin historial de interacción.



Collaborative Filtering Systems



Knowledge-Based Systems



Content-Based Systems



Hybrid Recommendation Systems

4

¿CÓMO CONSTRUIR
UN SISTEMA DE
RECOMENDACIÓN?

4.2

FACTORIZACIÓN
DE MATRICES

La técnica que vamos a implementar pertenece a la categoría de **filtrado colaborativo** y se basa en la **factorización de matrices**.

Este algoritmo se basa en una tabla donde las filas son los usuarios y las columnas con el producto (por ejemplo, películas). Y las celdas las valoraciones.

Ana	5	1		2	5
Elena	2	5	3	5	
Carlos		4	4		3
David	4		5	1	
Sofía	1	4		5	2

Las valoraciones pueden ser de dos tipos:

- **Explícitas:** me gusta/no me gusta, 1 a 5, ...
- **Implícitas:** por ejemplo, extraídas en base al número de películas que has visto de esa categoría.



El objetivo del algoritmo será **rellenar los huecos** con previsión de las valoraciones que los usuarios aún no han visto.

Ana	5	1	?	2	5
Elena	2	5	3	5	?
Carlos	?	4	4	?	3
David	4	?	5	1	?
Sofía	1	4	?	5	2

Factores latentes

La técnica asume que detrás de una valoración hay **características ocultas** (llamadas factores latentes) que comparten usuarios y productos.

- **Matriz de usuarios:** define cuánto le gusta a cada usuario ciertos géneros o estilos (p.e. cuánto le gusta la acción o un actor específico)
- **Matriz de Ítems:** define cuánto tiene cada película de esos mismos géneros o estilos.

La idea es llenar las dos matrices de factores latentes de forma que el resultado encaje en los valores conocidos, y así poder predecir las valoraciones faltantes.

	FL 1	FL 2	FL 3
Ana			
Elena			
Carlos			
David			
Sofía			



FL 1					
FL 2					
FL 3					



Ana	5	1		2	5
Elena	2	5	3	5	
Carlos		4	4		3
David	4		5	1	
Sofía	1	4		5	2

"Producto escalar"

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix} \checkmark$$

<https://www.disfrutalasmatematicas.com/algebra/matrices-multiplicacion.html>

¿Cómo podemos interpretar un factor latente?

Los factores latentes solo tienen sentido para la máquina, es decir, simplemente decide buscar un número K de *conceptos matemáticos ocultos* que expliquen esas notas.

Pero para darles sentido, nosotros podemos suponer:

- El factor 1 mide el nivel de acción / ciencia ficción
- El factor 2 mide el nivel de Romance

Mad Max tiene mucho de acción y poco de romance

	FL 1	FL 2
Ana	4.35	2.01
Elena	1.70	4.60
Carlos	3.85	4.01
David	3.70	1.99
Sofía	0.78	4.50

FL 1	4.77	1.24	3.87	0.25	4.99
FL 2	1.50	4.88	0.87	4.56	0.07

A Sofía le gustan muy poco las películas de acción y bastante las de romance

NOTA: los números son **inventados** y por eso no cuadra el resultado

	FL 1	FL 2
Ana	4.35	2.01
Elena	1.70	4.60
Carlos	3.85	4.01
David	3.70	1.99
Sofía	0.78	4.50



FL 1	4.77	1.24	3.87	0.25	4.99
FL 2	1.50	4.88	0.87	4.56	0.07



Ana	5	1		2	5
Elena	2	5	3	5	
Carlos		4	4		3
David	4		5	1	
Sofía	1	4		5	2

El valor de la puntuación de Ana en la película Matrix es el producto de la fila y la columna marcados en las tablas de arriba

$$4.35 * 4.77 + 2.01 * 1.50$$

NOTA: los números son **inventados** y por eso no cuadra el resultado

	FL 1	FL 2
Ana	4.35	2.01
Elena	1.70	4.60
Carlos	3.85	4.01
David	3.70	1.99
Sofía	0.78	4.50



	MATRIX	TITANIC	INCEPTION	La Fabulosa Aventura del Álbum	MAD MAX
FL 1	4.77	1.24	3.87	0.25	4.99
FL 2	1.50	4.88	0.87	4.56	0.07



Ana	5	1	2	5
Elena	2	5	3	5
Carlos		4	4	3
David	4		5	1
Sofía	1	4		5
				2

Observa que estamos multiplicando **cuánto le gustan a Ana las películas de acción por cuánto tiene Matrix de acción y cuánto le gustan las de romance por cuánto tiene de romance**

NOTA: los números son **inventados** y por eso no cuadra el resultado

	FL 1	FL 2
Ana	4.35	2.01
Elena	1.70	4.60
Carlos	3.85	4.01
David	3.70	1.99
Sofía	0.78	4.50




	MATRIX	TITANIC	INCEPTION	La Fabulosa Album	MAD MAX
FL 1	4.77	1.24	3.87	0.25	4.99
FL 2	1.50	4.88	0.87	4.56	0.07




	MATRIX	TITANIC	INCEPTION	La Fabulosa Album	MAD MAX
Ana	5	1	?	2	5
Elena	2	5	3	5	
Carlos		4	4		3
David	4		5	1	
Sofía	1	4		5	2

Para predecir si a Ana le gustará Inception solo tendríamos que multiplicar la fila y columna correspondiente en las matrices de factores latentes

NOTA: los números son **inventados** y por eso no cuadra el resultado

	FL 1	FL 2
Ana	4.1	
Elena	1.70	0.0
Carlos	3.85	4.01
David	3.70	1.99
Sofía	0.78	4.50



	MATRIX	TITANIC	INCISION	La Fabulosa Album	MAD MAX
FL 1	4.77	1.24		0.25	4.99
FL 2	1.50	4.88	0.87	4.56	0.07



	MATRIX	TITANIC	INCISION	La Fabulosa Album	MAD MAX
Ana	5	1		2	5
Elena	2	5	3	5	
Carlos		4	4		3
David	4		5	1	
Sofía	1	4		5	2

El problema que tenemos es
que **no conocemos las**
matrices de factores latentes.

NOTA: los números son **inventados** y por eso no cuadra el resultado

	FL 1	FL 2
Ana	4	1
Elena	1.70	0
Carlos	3.85	4.01
David	3.70	1.99
Sofía	0.78	4.50



FL 1	4.77	1.24	7	0.25	4.99
FL 2	1.50	4.88	0.87	4.56	0.07



Ana	5	1		2	5
Elena	2	5	3	5	
Carlos		4	4		3
David	4		5	1	
Sofía	1	4		5	2

Así que nuestro objetivo será rellenar esas matrices con valores que, al multiplicarlas, se aproximen lo más posible a las valoraciones que conocemos.

¿Cómo lo haríamos?

Paso 1: defino cuántos factores latentes voy a usar. En este ejemplo voy a definir 2 factores latentes. $K = 2$

Paso 2: el algoritmo crea las dos matrices pequeñas llenándolas inicialmente con **números aleatorios**.

Paso 3: se aplica el algoritmo llamado **descenso de gradiente**:

- Multiplica las matrices con sus valores aleatorios
- Compara el resultado con la tabla original
- Ajusta ligeramente los números en las matrices para que en el próximo intento el error sea menor.
- Repite esto miles de veces hasta que el error para todas las notas sea mínimo

Paso 5: al terminar el entrenamiento las matrices ya no tienen números aleatorios, sino perfiles precisos con los que podríamos llenar la matriz original.

Una cuestión importante es: **¿cómo escojo el valor k?**

Estos valores que en machine learning no lo aprende el sistema, sino que se debe configurar mediante pruebas se les llama **hiperparámetro**.

El dilema de este valor es:

- **Si k es muy pequeño (subajuste)**: el modelo será muy simple y no podrá abarcar todos los gustos cinematográficos.
- **Si k es muy grande (sobreajuste)**: el modelo tiene tantas variables que empieza a memorizar la tabla exacta de valoraciones, incluyendo el ruido o los votos al azar, en lugar de aprender el patrón general. Al margen de la enorme potencia de cálculo requerida.

Para calcularlo utilizaremos una técnica llamada **Validación Cruzada**, que consiste en realizar simulaciones con diferentes valores hasta encontrar el punto en que menos se equivoca.

4

¿CÓMO CONSTRUIR UN SISTEMA DE RECOMENDACIÓN?

4.3

IMPLEMENTACIÓN EN PYSPARK

Vamos a ver cómo implementaríamos en PySpark un sistema de recomendación de películas con un *dataset* sintético.

```
from pyspark.sql import SparkSession
from pyspark.ml.recommendation import ALS
from pyspark.ml.evaluation import RegressionEvaluator

spark = ( SparkSession.builder
            .appName("MiniNetflix_ALS")
            .getOrCreate()
        )
```

Creamos datos sintéticos y los agregamos en el dataframe de valoraciones.

```
# (Usuario, Película, Nota)
# Usuarios : 1: Ana | 2: Carlos | 3: Elena | 4: David | 5: Sofía
# Películas: 101: Matrix | 102: Titanic | 103: Inception | 104: Mad Max | 105: Amelie
datos_sinteticos = [
    (1, 101, 5.0), (1, 103, 5.0), (1, 104, 4.0), (1, 105, 1.0),
    (2, 102, 5.0), (2, 105, 5.0), (2, 101, 1.0), (2, 104, 2.0),
    (3, 101, 4.0), (3, 103, 4.0), (3, 102, 2.0),
    (4, 101, 5.0), (4, 104, 5.0), (4, 102, 1.0),
    (5, 102, 4.0), (5, 105, 5.0), (5, 103, 1.0)
]

columnas = ["usuario_id", "pelicula_id", "valoracion"]
df_valoraciones = spark.createDataFrame(datos_sinteticos, columnas)
```

```
df_valoraciones.show()
```

```
+-----+-----+-----+
|usuario_id|pelicula_id|valoracion|
+-----+-----+-----+
|      1|     101|      5.0|
|      1|     103|      5.0|
|      1|     104|      4.0|
|      1|     105|      1.0|
|      2|     102|      5.0|
|      2|     105|      5.0|
|      2|     101|      1.0|
|      2|     104|      2.0|
|      3|     101|      4.0|
|      3|     103|      4.0|
|      3|     102|      2.0|
|      4|     101|      5.0|
|      4|     104|      5.0|
|      4|     102|      1.0|
|      5|     102|      4.0|
|      5|     105|      5.0|
|      5|     103|      1.0|
+-----+-----+-----+
```

Para verlo más claro, estas serían las valoraciones que estamos introduciendo en el sistema:

	101	102	103	104	105
1 - Ana	5		5	4	1
2- Carlos	1	5		2	5
3 – Elena	4	2	4		
4 – David	5	1		5	
5 - Sofía		4	1		5

Para **evaluar el rendimiento del algoritmo**, dividimos los datos en entrenamiento (80%) y test (20%)

```
# Separamos aleatoriamente: 80% para entrenar, 20% para test
datos_entrenamiento, datos_prueba = ( df_valoraciones
                                         .randomSplit([0.8, 0.2], seed=42)
                                         )
print(f"Datos para entrenar: {datos_entrenamiento.count()}")
print(f"Datos para evaluar: {datos_prueba.count()}")
```

Datos para entrenar: 14
Datos para evaluar: 3

Tengo 17 valoraciones.
Uso 14 para entrenar el
modelo y 3 para
evaluar su rendimiento

La función randomSplit() divide el
DataFrame en 2 (o más)
subconjuntos aleatorios

La **semilla** me sirve para reproducir
la misma división

```
als = ALS(  
    maxIter=10,  
    regParam=1.5,  
    rank=2,  
    userCol="usuario_id",  
    itemCol="pelicula_id",  
    ratingCol="valoracion",  
    coldStartStrategy="drop"  
)
```

Establecemos los parámetros del algoritmo ALS.

Estos parámetros son:

- **userCol**: nombre de la columna con los IDs numéricos de los usuarios
- **itemCol**: nombre de la columna con los IDs de las películas, canciones, productos, ...
- **ratingCol**: nombre de la columna con las valoraciones

```
als = ALS(  
    maxIter=10,  
    regParam=1.5,  
    rank=2,  
    userCol="usuario_id",  
    itemCol="pelicula_id",  
    ratingCol="valoracion",  
    coldStartStrategy="drop"  
)
```

- **rank:**

- Es el valor k. Define cuántos *factores latentes* quieras que el algoritmo busque.
- Si es muy bajo hay subajuste: el modelo es muy simple y agrupa todo en categorías muy anchas.
- Si es muy alto hay sobreajuste: el modelo corre el riesgo de memorizar la tabla en lugar de aprender patrones

```
als = ALS(  
    maxIter=10,  
    regParam=1.5,  
    rank=2,  
    userCol="usuario_id",  
    itemCol="pelicula_id",  
    ratingCol="valoracion",  
    coldStartStrategy="drop"  
)
```

- **maxIter:**
 - El algoritmo ALS funciona congelando una matriz y modificando ligeramente la otra, cambiando alternativamente entre una matriz y otra. Este parámetro define **cuántas iteraciones** se realizarán.
 - Un valor típico suele estar entre 10 y 20

```
als = ALS(  
    maxIter=10,  
    regParam=1.5,  
    rank=2,  
    userCol="usuario_id",  
    itemCol="pelicula_id",  
    ratingCol="valoracion",  
    coldStartStrategy="drop"  
)
```

- **regParam:**

- Este parámetro sirve para penalizar números extremos en las matrices para intentar alcanzar el valor exacto.
- Si es muy bajo (0.001) el modelo memorizará los datos de entrenamiento (sobreajuste), pero predecirá muy mal el futuro
- Si es muy alto (p.e. 10.0) tendrá tanto miedo a usar números grandes que acabará poniendo todo a cero o a la nota media global
- Un valor ideal suele ser entre 0.05 y 0.1

```
als = ALS(  
    maxIter=10,  
    regParam=1.5,  
    rank=2,  
    userCol="usuario_id",  
    itemCol="pelicula_id",  
    ratingCol="valoracion",  
    coldStartStrategy="drop"  
)
```

- **coldStartStrategy:**

- Define el comportamiento cuando se le pide predecir la nota para un usuario o película que nunca ha visto.
- Los valores son:
 - “nan”: devuelve error matemático en este caso
 - “drop”: si no conoce a un usuario o película ignora la fila y no rompe el programa. Obligatorio durante la fase de evaluación.

Ahora solo nos queda entrenar el modelo, para lo que usamos la función **fit()**

```
modelo = als.fit(datos_entrenamiento)  
print("¡El modelo ha sido entrenado!")
```

¡El modelo ha sido entrenado!

Esta función devolverá un **modelo entrenado**

La función **fit()** pone en marcha el algoritmo de entrenamiento sobre los datos utilizando los parámetros que definimos en el punto anterior.

Una vez entrenado el modelo es momento de **evaluar su rendimiento**.

1. El modelo hace sus predicciones sobre los datos de prueba

```
predicciones = modelo.transform(datos_prueba)
```

2. Configuramos el Evaluador

```
evaluador = RegressionEvaluator(  
    metricName="rmse",  
    labelCol="valoracion",  
    predictionCol="prediction"  
)
```

3. Calculamos el margen de error

```
error_rmse = evaluador.evaluate(predicciones)  
print(f"Margen de error (RMSE): {error_rmse}")
```

Margen de error (RMSE): 0.9720901537752907

La función transform() aplica el modelo sobre los datos que se le pasen como parámetro. En este caso sobre los datos de prueba.

En PySpark (y en ML en general) los modelos entrenados se llaman **transformers**.

Si visualizáramos la variable predicciones veremos los registros que se dejaron para test y la predicción en base al modelo creado

```
predicciones.show()
```

usuario_id	pelicula_id	valoracion	prediction
1	103	5.0	3.9214566
1	105	1.0	1.5131731
5	103	1.0	2.1867077

Una vez entrenado el modelo es momento de **evaluar su rendimiento**.

```
# 1. El modelo hace sus predicciones sobre los datos
predicciones = modelo.transform(datos_prueba)

# 2. Configuramos el Evaluador
evaluador = RegressionEvaluator(
    metricName="rmse",
    labelCol="valoracion",
    predictionCol="prediction"
)

# 3. Calculamos el margen de error
error_rmse = evaluador.evaluate(predicciones)
print(f"Margen de error (RMSE): {error_rmse}")

Margen de error (RMSE): 0.972090153775296
```

Vamos a usar `RegressionEvaluator()` para evaluar el rendimiento del algoritmo ya que estamos ante un problema de **regresión** (intentamos adivinar un valor numérico)

Los parámetros que le pasamos son:

- **labelCol**: columna que tiene las valoraciones reales
- **predictionCol**: columna donde el algoritmo guardó su predicción (por defecto, `transform` lo hace en una columna llamada `prediction`)
- **metricName**: fórmula matemática que queremos utilizar. La más común en sistemas de recomendación es **rmse** (*Raíz del Error Cuadrático Medio*)

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

Una vez entrenado el modelo es momento de **evaluar su rendimiento**.

```
# 1. El modelo hace sus predicciones sobre los datos de prueba  
predicciones = modelo.transform(datos_prueba)
```

```
# 2. Configuramos el Evaluador  
evaluador = RegressionEvaluator(  
    metricName="rmse",  
    labelCol="valoracion",  
    predictionCol="prediction"  
)
```

Para aplicar el evaluador utilizamos la función evaluate()

```
# 3. Calculamos el margen de error  
error_rmse = evaluador.evaluate(predicciones)  
print(f"Margen de error (RMSE): {error_rmse}")
```

Margen de error (RMSE): 0.9720901537752907

En este caso nos ha dado un valor RMSE de 0.97, lo cual quiere decir que el error medio en cuanto a las predicciones de aproximadamente un punto.

En este punto habría que decidir si es suficiente o si hay que probar con otros hiperparámetros.

Podemos usar la función `recommendForAllUsers()` para que nos muestre las n películas con mayor valoración para cada usuario.

```
# Recomendar las 3 mejores películas para cada usuario
recomendaciones_finales = modelo.recommendForAllUsers(3)

# Mostrar resultados
recomendaciones_finales.show(truncate=False)
```

```
+-----+-----+
|usuario_id|recommendations
+-----+-----+
|1        |[{{101, 4.163322}, {103, 3.9214566}, {104, 3.8119254}] |
|2        |[{{105, 4.4268155}, {102, 3.599575}, {103, 2.1950958}] |
|3        |[{{103, 3.4295301}, {101, 3.3716993}, {104, 3.2409165}]|
|4        |[{{101, 4.3320847}, {103, 4.056405}, {104, 3.9518893}] |
|5        |[{{105, 4.437932}, {102, 3.6087036}, {103, 2.1867077}] |
+-----+-----+
```

Observa que esta función nos devuelve la predicción para todos los campos, incluso de aquellos que ya teníamos valoración

```
datos_sinteticos = [
    (1, 101, 5.0), (1, 103, 5.0), (1, 104, 4.0), (1, 105, 1.0),
    (2, 102, 5.0), (2, 105, 5.0), (2, 101, 1.0), (2, 104, 2.0),
    (3, 101, 4.0), (3, 103, 4.0), (3, 102, 2.0),
    (4, 101, 5.0), (4, 104, 5.0), (4, 102, 1.0),
    (5, 102, 4.0), (5, 105, 5.0), (5, 103, 1.0)
]
```

4

¿CÓMO CONSTRUIR
UN SISTEMA DE
RECOMENDACIÓN?

4.4

CÁLCULO DE
HIPERPARÁMETROS

El problema del método anterior es que si el valor del RMSE no nos convence habría que ir probando diferentes valores a mano, lo cual puede ser demasiado laborioso.

Para automatizar este proceso disponemos de **Grid Search**, que básicamente es un método que podemos programar para probar todas las combinaciones posibles de forma automática hasta encontrar el que menor RMSE obtenga.

Comenzamos con las importaciones de herramientas:

```
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator  
from pyspark.ml.evaluation import RegressionEvaluator  
from pyspark.ml.recommendation import ALS
```

ParamGridBuilder: será el constructor de la cuadrícula con los valores que queremos probar de cada hiperparámetro.

CrossValidator: el orquestador que realizará las pruebas

Definimos el **modelo base** y el **evaluador** que calculará el RMSE de forma análoga a como hicimos antes

```
als = ALS(  
    userCol="usuario_id",  
    itemCol="pelicula_id",  
    ratingCol="valoracion",  
    coldStartStrategy="drop"  
)  
  
evaluador = RegressionEvaluator(  
    metricName="rmse",  
    labelCol="valoracion",  
    predictionCol="prediction"  
)
```

Usamos **ParamGridBuilder()** para indicar qué valores queremos que pruebe para cada uno de los hiperparámetros.

Observa que indicamos cada uno de los hiperparámetros y qué valores debe probar.

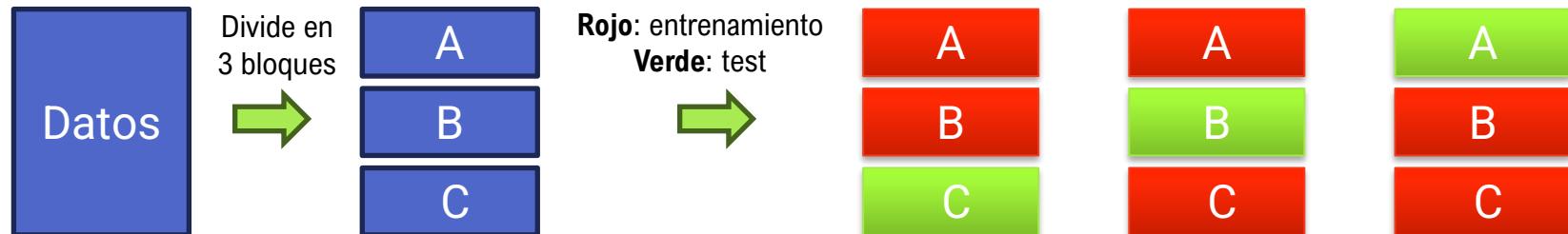
```
grid_params = ( ParamGridBuilder()
    .addGrid(als.rank, [5, 10, 20])
    .addGrid(als.regParam, [0.01, 0.1, 0.5])
    .addGrid(als.maxIter, [10, 20])
    .build()
)
```

En este ejemplo probará un total de 18 combinaciones ($3 \times 3 \times 2$)

El motor que une el modelo, el evaluador y la cuadrícula es **CrossValidator**. Le tenemos que pasar el modelo base, el evaluador y la rejilla de parámetros que hemos definido.

```
validador_cruzado = CrossValidator(  
    estimator=als,  
    estimatorParamMaps=grid_params,  
    evaluator=evaluador,  
    numFolds=3  
)
```

El parámetro **numFolds** sirve para indicar el número de pruebas que realizará para cada combinación de hiperparámetros. En cada prueba utilizará diferentes conjuntos de datos para entrenamiento y para test



Ya solo queda usar la función **fit()** para entrenar el modelo.

Ten en cuenta que, según el tamaño del *dataset* y el número de hiperparámetros escogidos puede tardar horas o incluso días.

```
modelo_optimizado = validador_cruzado.fit(df_valoraciones)
print("Modelo entrenado")
```

Modelo entrenado

Ahora ya tenemos un modelo entrenado con la mejor combinación de parámetros de entre todos los indicados.

Si quisiéramos saber todas las pruebas que ha realizado el CrossValidator y el valor RMSE obtenido en cada una de ellas, podemos hacerlo con este código:

```
# Obtenemos la lista de combinaciones probadas
combinaciones = modelo_optimizado.getEstimatorParamMaps()

# Obtenemos el RMSE de cada combinación
notas_rmse = modelo_optimizado.avgMetrics

# Combinamos todo y ordenamos por RMSE
ranking = sorted(zip(notas_rmse, combinaciones), key=lambda x: x[0])

print("--- RANKING DE TODAS LAS PRUEBAS ---")
for nota, parametros in ranking:
    print(f"\nRMSE Obtenido: {nota}")

    for param, valor in parametros.items():
        print(f"  {param.name}: {valor}")

--- RANKING DE TODAS LAS PRUEBAS ---

RMSE Obtenido: 2.9423635537012136
  rank: 5
  regParam: 0.01
  maxIter: 10
```



Si quisiéramos saber todas las pruebas que ha realizado el CrossValidator y el valor RMSE obtenido en cada una de ellas, podemos hacerlo con este código:

```
# Obtenemos la lista de combinaciones probadas
combinaciones = modelo_optimizado.getEstimatorParamMaps()

# Obtenemos el RMSE de cada combinación
notas_rmse = modelo_optimizado.avgMetrics

# Combinamos todo y creamos un ranking
ranking = sorted(zip(notas_rmse, combinaciones))

print("--- RANKING DE TODAS LAS PRUEBAS ---")
for nota, parametro in ranking:
    print(f"\nRMSE: {nota}\n{parametro}\n")

    for param, valor in parametro.items():
        print(f"  {param}: {valor}\n")
```

Con `getEstimatorParamMaps()` Obtenemos todos los parámetros que se han probado

```
[{Param(parent='ALS_607e8cf7ce33', name='rank', doc='rank of the factorization'): 5,
  Param(parent='ALS_607e8cf7ce33', name='regParam', doc='regularization parameter (>= 0.): 0.01,
  Param(parent='ALS_607e8cf7ce33', name='maxIter', doc='max number of iterations (>= 0.): 10),
  {Param(parent='ALS_607e8cf7ce33', name='rank', doc='rank of the factorization'): 5,
  Param(parent='ALS_607e8cf7ce33', name='regParam', doc='regularization parameter (>= 0.): 0.01,
  Param(parent='ALS_607e8cf7ce33', name='maxIter', doc='max number of iterations (>= 0.): 20),
  {Param(parent='ALS_607e8cf7ce33', name='rank', doc='rank of the factorization'): 5,
```

--- RANKING DE TODAS LAS PRUEBAS ---

RMSE Obtenido: 2.9423635537012136

rank: 5

regParam: 0.01

maxIter: 10

Si quisiéramos saber todas las pruebas que ha realizado el CrossValidator y el valor RMSE obtenido en cada una de ellas, podemos hacerlo con este código:

```
# Obtenemos la lista de combinaciones probadas
combinaciones = modelo_optimizado.getEstimatorParamMaps()

# Obtenemos el RMSE de cada combinación
notas_rmse = modelo_optimizado.avgMetrics

# Combinamos todo y ordenamos por RMSE
ranking = sorted(zip(notas_rmse, combinaciones), key=lambda x: x[0])

print("--- RANKING DE TODAS LAS PRUEBAS ---")
for nota, parametros in ranking:
    print(f"\nRMSE Obtenido: {nota}")

    for param, valor in parametros.items():
        print(f"  {param.name}: {valor}")

--- RANKING DE TODAS LAS PRUEBAS ---

RMSE Obtenido: 2.9423635537012136
rank: 5
regParam: 0.01
maxIter: 10
```

Métricas promedio de validación cruzada para cada *paramMap* en *CrossValidator.estimatorParamMaps*, en el orden correspondiente

modelo_optimizado.avgMetrics

```
[np.float64(2.702995026112658),
 np.float64(2.4161499344761452),
 np.float64(2.2172312281591036),
 np.float64(2.2118749910358866),
 np.float64(2.356879628859447),
 np.float64(2.357357063460524),
 np.float64(2.1481656426751825),
 np.float64(1.8848619604268497),
 np.float64(1.8448468110721177),
 np.float64(1.842286762044288)]
```

Si quisiéramos saber todas las pruebas que ha realizado el CrossValidator y el valor RMSE obtenido en cada una de ellas, podemos hacerlo con este código:

```
# Obtenemos la lista de combinaciones probadas
combinaciones = modelo_optimizado.getEstimatorParamMaps()

# Obtenemos el RMSE de cada combinación
notas_rmse = modelo_optimizado.avgMetrics

# Combinamos todo y ordenamos por RMSE
ranking = sorted(zip(notas_rmse, combinaciones), key=lambda x: x[0])
```

Con el `zip()` combinamos ambas listas donde cada elemento es una tupla con dos valores: el valor y el diccionario que ha generado dicho valor.

```
list(zip(notas_rmse, combinaciones))

[(np.float64(2.702995026112658),
 {Param(parent='ALS_607e8cf7ce33', name='rank', doc='rank of the factorization'): 5,
  Param(parent='ALS_607e8cf7ce33', name='regParam', doc='regularization parameter (>= 0).'): 0.01,
  Param(parent='ALS_607e8cf7ce33', name='maxIter', doc='max number of iterations (>= 0).'): 10}),
 (np.float64(2.4161499344761452),
 {Param(parent='ALS_607e8cf7ce33', name='rank', doc='rank of the factorization'): 5,
  Param(parent='ALS_607e8cf7ce33', name='regParam', doc='regularization parameter (>= 0).'): 0.01,
  Param(parent='ALS_607e8cf7ce33', name='maxIter', doc='max number of iterations (>= 0).'): 20}),
```

Si quisiéramos saber todas las pruebas que ha realizado el CrossValidator y el valor RMSE obtenido en cada una de ellas, podemos hacerlo con este código:

```
# Obtenemos la lista de combinaciones probadas
combinaciones = modelo_optimizado.getEstimatorParamMaps()

# Obtenemos el RMSE de cada combinación
notas_rmse = modelo_optimizado.avgMetrics

# Combinamos todo y ordenamos por RMSE
ranking = sorted(zip(notas_rmse, combinaciones), key=lambda x: x[0])
```

Los ordeno, pero como los elementos son tuplas tengo que implementar una función personalizada de ordenación.

```
list(zip(notas_rmse, combinaciones))
```

x[0]

```
(np.float64(2.702995026112658),
```

```
{Param(parent='ALS_607e8cf7ce33', name='rank', doc='rank of the factorization'): 5,
 Param(parent='ALS_607e8cf7ce33', name='regParam', doc='regularization parameter (>= 0.)'): 0.01,
 Param(parent='ALS_607e8cf7ce33', name='maxIter', doc='max number of iterations (>= 0.)'): 10}),
```

```
(np.float64(2.4161499344761452),
```

```
{Param(parent='ALS_607e8cf7ce33', name='rank', doc='rank of the factorization'): 5,
 Param(parent='ALS_607e8cf7ce33', name='regParam', doc='regularization parameter (>= 0.)'): 0.01,
 Param(parent='ALS_607e8cf7ce33', name='maxIter', doc='max number of iterations (>= 0.)'): 20}),
```

```
maxIter: 10
```

Si quisiéramos saber todas las pruebas que ha realizado el CrossValidator y el valor RMSE obtenido en cada una de ellas, podemos hacerlo con este código:

```
# Obtenemos la lista de combinaciones probadas
combinaciones = modelo_optimizado.getEstimatorParamMaps()

# Obtenemos el RMSE de cada combinación
notas_rmse = modelo_optimizado.avgMetrics

# Combinamos todo y ordenamos por RMSE
ranking = sorted(zip(notas_rmse, combinaciones), key=lambda x: x[0])

print("--- RANKING DE TODAS LAS PRUEBAS ---")
for nota, parametros in ranking:
    print(f"\nRMSE Obtenido: {nota}")

    for param, valor in parametros.items():
        print(f"  {param.name}: {valor}")

--- RANKING DE TODAS LAS PRUEBAS ---
```

```
RMSE Obtenido: 2.9423635537012136
rank: 5
regParam: 0.01
maxIter: 10
```