Práctica 0: Creación de una base de datos en SQL y operaciones básicas con Spark

DEDICACIÓN APROXIMADA: 12 horas

PUNTUACIÓN: 4 puntos

OBJETIVOS

- Realizar la instalación y configuración de una base de datos SQL en un entorno local.
- Instalar y configurar las APIs necesarias para la integración con Python.
- Utilizar PySpark junto con la biblioteca JDBC para establecer una conexión con la base de datos SQL.
- Cargar datos desde archivos CSV hacia el proyecto de python.
- Ejecutar consultas básicas en Spark, como filtros, agregaciones y ordenamientos sobre los datos.
- Familiarizarse con los principales comandos y técnicas de limpieza de datos.

ALGUNAS INDICACIONES SOBRE SPARK:

Apache Spark es un motor de procesamiento distribuido en memoria, diseñado para trabajar en clústeres mediante un modelo de arquitectura maestro-esclavo. En este esquema, el nodo maestro se conoce como nodo controlador, mientras que el resto de los equipos que forman parte del clúster actúan como nodos trabajadores (workers).

El nodo controlador ejecuta una aplicación que se encarga de coordinar el procesamiento de los datos. Cada aplicación Spark lanza uno o varios "executors", los cuales gestionan múltiples unidades de paralelismo, conocidas como "slots". Estos slots pueden entenderse como núcleos de CPU o hilos de procesamiento. A mayor número de slots disponibles, mayor será la capacidad para ejecutar tareas en paralelo.

Sin embargo, este paralelismo debe ser cuidadosamente balanceado, ya que un mayor número de recursos implica también un aumento en los costos asociados al procesamiento

En Spark, la aplicación driver es la que envía trabajos al clúster para que se procesen. Pero esos trabajos no se ejecutan de inmediato.

Spark usa algo llamado evaluación diferida. Esto significa que, cuando aplicas transformaciones a los datos (como filtros o mapas), Spark no hace nada en ese momento. Solo guarda una lista de lo que tiene que hacer. La ejecución real no ocurre hasta que se pide una acción, como guardar un archivo, contar registros o enviar datos a otro sistema como Kafka.

Cuando se pide esa acción, Spark crea un job (trabajo), que es el conjunto de pasos necesarios para completar esa tarea. El driver se encarga de coordinar ese trabajo.

Cada trabajo se divide en etapas (stages). Dentro de una etapa, las tareas se pueden hacer en paralelo si no hace falta mover datos entre nodos. Si los datos tienen que ir de un

nodo a otro, ocurre lo que se llama un shuffle. Este paso es muy costoso en tiempo y recursos, así que Spark intenta evitarlo todo lo que puede.

PREPARACIÓN DEL ENTORNO:

Creación del entorno

Se recomienda generar un entorno en Anaconda para configurar el entorno. Desde el CMD de Anaconda:

```
conda create --name mineriadedatos2025
```

Y se activa el entorno

```
conda activate mineriadedatos2025
```

Entorno con python

Para completar esta práctica, es necesario tener instalado Python en su ordenador. Puede descargar e instalar la versión más reciente de Python desde el sitio oficial: python.org. Siga las instrucciones de instalación proporcionadas en la página para su sistema operativo o ejecute.

```
conda install python=3.10
```

Si ya lo tiene instalado, abra una terminal o línea de comandos y escriba lo siguiente (puede ser en el cmd de Anaconda:

```
python --version
```

Herramientas

Graphviz es una herramienta que se utiliza para la visualización de grafos. Es muy útil para generar gráficos de manera programática a partir de descripciones en texto simple.

```
conda install -c anaconda graphviz
```

Verifica que Graphviz se haya instalado correctamente ejecutando:

```
dot -V
```

Apache Spark es un motor de procesamiento de datos de código abierto y distribuido que permite realizar análisis de datos a gran escala. PySpark permite a los usuarios aprovechar la potencia de Spark utilizando el lenguaje de programación Python. Para instalarlo hay que ejecutar:

```
conda install -c conda-forge pyspark=3.3
```

Java JRE es un conjunto de herramientas y bibliotecas que permite la ejecución de aplicaciones Java en un sistema.

```
conda install cyclus::java-jre
```

Ahora comprobamos la versión de pyspark que hemos instalado:

```
pyspark --version
```

También es necesario instalar las herramientas para leer los csv con los que se va a trabajar.

```
conda install -c anaconda openpyxl=3.1.5
```

Es necesario verificar que las herramientas se encuentran instaladas en el entorno.

```
conda list
```

Posteriormente se REINICIA.

CADA VEZ QUE SE QUIERA ENTRAR AL ENTORNO DONDE SE HA INSTALADO TODO HAY QUE USAR

```
conda activate mineriadedatos2025
```

A continuación, es necesario crear un nuevo archivo de script de Python usando un editor de texto.

Por ejemplo, para VSCode:

```
code .
```

Si existen errores en el path, tal vez sea necesario modificar la ruta del archivo de Python.exe.

setx PYSPARK_PYTHON "C:\ruta\python.exe"

setx PYSPARK_DRIVER_PYTHON "C:\ruta\python.exe"

Para localizar la ruta se escribe en el cmd *where Python*

CREAR Y SICRONIZAR EL PROYECTO CON GITHUB:

Antes de continuar, es necesario crear un nuevo proyecto, sincronizado con un repositorio en GitHub, público o privado. El proyecto debe seguir el patrón MVC. El objetivo es que la lógica de la aplicación sea fácil de entender, mantener y escalar sin que se vuelva confuso.

Por ejemplo, la organización sigue una separación clara por responsabilidades:

- Inicialización de Spark

  En un módulo aparte (spark_session.py) se define una función que crea la SparkSession.

Esto centraliza la configuración de Spark y permite cambiar parámetros (por ejemplo, nombre de la aplicación, número de cores) sin tocar el resto del código.

- Conexión base de datos

En otro módulo aparte, la aplicación incluye un módulo dedicado para manejar la conexión con la base de datos utilizando JDBC (Java Database Connectivity), que es el mecanismo estándar para que Spark se comunique con sistemas de bases de datos relacionales como PostgreSQL, MySQL o SQL Server.

TECNICAS DE APACHE SPARK PARA TRATAMIENTO DE LOS DATOS:

Show

Muestra el contenido del DataFrame resultante. El ejemplo de a continuación mostrará el DataFrame resultante con truncamiento en columnas que excedan 20 caracteres.

```
data_frame.show()
```

Sin embargo, con el siguiente ejemplo se mostrará el DataFrame sin ningún truncamiento.

```
data_frame.show(truncate=False)
```

Head y tail

Tail es el método más directo para obtener la última fila de un DataFrame. Y head es el método más directo para obtener la priemra fila de un DataFrame. Ambos devuelven siempre una Lista, aunque sea de un elemento. Para acceder a ese elemento es necesario usar [0]

```
Primer_elemnto=data_frame.head(1)[0]
Segundo_elemento=data_frame.tail(1)[0]
```

Columnas

El atributo “columns” de un DataFrame de PySpark devuelve una lista de cadenas con los nombres de todas las columnas del DataFrame, en el mismo orden en que aparecen. Es muy útil cuando se quiere recorrer las columnas de forma dinámica.

```
data_frame.columns
```

Distinct

Se utiliza para eliminar filas duplicadas de un DataFrame. Devuelve un nuevo DataFrame que contiene solo filas únicas, eliminando cualquier fila duplicada que esté presente en el DataFrame original.

```
data_frame.distinct()
```

Eliminar duplicados

Elimina filas duplicadas de un DataFrame. Si se aplica sin parámetros, elimina las filas que son idénticas en todas las columnas, similar a distinct.

```
data_frame.dropDuplicates().show(truncate=False)
```

Si se especifica un conjunto de columnas, elimina las filas duplicadas considerando solo esas columnas, manteniendo una única fila para cada combinación de valores duplicados en las columnas seleccionadas. Es útil para asegurar que los datos sean únicos según los criterios especificados.

```
data_frame.dropDuplicates(["COL1"]).show(truncate=False)
```

Cuando son varias columnas, recibe una sola lista con todas las columnas que se quiere usar para eliminar duplicados. Se revisan todas las filas y se escogen la primera ocurrencia de cada combinación única de las columnas seleccionadas.

```
data_frame.dropDuplicates(["COL1", “COL2”]).show(truncate=False)
```

Dropna

Se utiliza para eliminar filas que contienen valores nulos (null) en un DataFrame. Por defecto, borra cualquier fila que tenga al menos un valor nulo en cualquier columna, pero se puede personalizar con parámetros. Por ejemplo, df.dropna(how="all") elimina solo las filas donde todas las columnas son nulas, mientras que df.dropna(how="any") (por defecto) elimina filas con al menos un valor nulo. También se puede usar el parámetro subset para especificar columnas concretas: df.dropna(subset=["col1", "col2"]) elimina las filas donde col1 o col2 son nulas. Este método es útil para limpiar datos antes de análisis o modelado, evitando que los valores nulos interfieran en los cálculos.

```
data_frame.dropna(subset=['ID'])
```

Estimación de cuartiles

Permite calcular los cuartiles (o percentiles) de una columna numérica utilizando la función approxQuantile. El valor de 0.01 es el parámetro de error relativo.

```
cuartiles = data_frame.approxQuantile("Name Column", [0.25, 0.5, 0.75], 0.01)
```

Select

Se utiliza para seleccionar columnas específicas de un DataFrame y crear un nuevo DataFrame que solo contiene esas columnas. Es una operación común cuando se necesita filtrar o transformar columnas antes de realizar análisis adicionales o aplicar transformaciones en los datos. Muy útil también para mostrar columnas concretas.

```
data_frame.select(["Nombre"]).show()
```

Si se quieren aplicar transformaciones como operaciones sobre los valores o renombrar las columnas, es necesario usar “col” (*from pyspark.sql.functions import col*). El siguiente ejemplo selecciona las columnas Nombre y Edad. En la columna edad añade 1 año a todos.

```
data_frame.select(col("Nombre"), (col("Edad") + 1).show()
```

Alias

Se utiliza para renombrar temporalmente una columna (no modifica el dataframe original). Esto es útil para darle un nombre más comprensible a una columna, o cuando estás trabajando con DataFrames grandes y necesitas referenciar ciertas columnas con nombres más cortos o descriptivos.

```
data_frame.select(col("nombre_original").alias("nombre_nuevo"))
```

Count

Este método devuelve un número entero con el total de registros (filas) del DataFrame, teniendo en cuenta todos los filtros o transformaciones que se hayan aplicado previamente.

```
data_frame.count()
```

CountDistinct

Permite contar cuántos valores únicos hay en una o más columnas de un DataFrame.

```
data_frame.select(countDistinct("namecol"))
```

Filter

Permite seleccionar filas que cumplen ciertas condiciones.

```
dataframe = data_frame.filter(data_frame.namecol > 30)

dataframe.show()
```

Nulls

En minería de datos, el tratamiento de valores nulos es un paso fundamental del preprocesamiento de datos, ya que la presencia de valores faltantes puede afectar la calidad del análisis y la precisión de los modelos. Los datos nulos pueden deberse a múltiples causas, como errores en la captura de información, campos opcionales no completados o problemas de integración entre diferentes fuentes.

Antes de aplicar técnicas de análisis, es importante detectar y cuantificar la cantidad de valores nulos en cada variable, para luego decidir la estrategia más adecuada de tratamiento. Entre las técnicas más comunes se encuentran la eliminación de registros incompletos (cuando la proporción de nulos es baja), la imputación mediante valores

estadísticos (media, mediana, moda) o modelos predictivos, y en algunos casos la creación de categorías especiales que representen la ausencia de dato.

Un manejo adecuado de los nulos garantiza que el conocimiento extraído de los datos sea más representativo y confiable.

Las funciones isNull e isNotNull se utilizan para manejar valores nulos en los DataFrames. Estas funciones son especialmente útiles cuando se necesita filtrar filas en base a la presencia o ausencia de valores nulos en una columna específica.

Por ejemplo, para un dataFrame con una columna llamada Name, y se quieren obtener todas las filas donde Name es null:

```
df_nulls = df.filter(df.Name.isNull())
df_nulls.show()
```

Sin embargo, para verificar los valores en una columna "Colum" no son nulo se usa isNotNull. Devuelve True si el valor no es nulo y False si es nulo.

```
df_not_nulls = df.filter(df.Colum.isNotNull())
df_not_nulls.show()
```

Join

Permite combinar dos DataFrames basados en una columna común.

```
joined_df = data_frame1.join(data_frame2, on="Name", how="inner")
joined_df.show()
```

- Inner: Devuelve solo las filas que tienen coincidencia en ambas tablas.
- Left: Devuelve todas las filas de la tabla izquierda (antes del join) aunque no tengan coincidencia en la derecha.
- Right: Devuelve todas las filas de la tabla derecha (dentro del join) aunque no tengan coincidencia en la derecha.
- Full: Devuelve todas las filas de ambas tablas. Donde no hay coincidencia, rellena con null.

When

Se utiliza para aplicar condiciones lógicas a las columnas de un DataFrame y generar nuevas columnas basadas en esas condiciones. "Value" sería el valor que tendría la columna "when(condition, value)"

Otherwise

Se utiliza en combinación con la función when para especificar el valor que se debe asignar a una columna en caso de que la condición establecida en when no se cumpla. Es similar al if-else de la programación tradicional.

```
data_frame.withColumn("Category",when(col("Age")>=18,"Adult").otherwise("Minor")).show()
```

Renombrar Columnas

Permite cambiar el nombre de una columna en un DataFrame.

```
data_frame.withColumnRenamed(columna_existente, nueva_columna)
```

GroupBy y agg

Permite agrupar datos por una columna y realizar agregaciones.

GroupBy: No se utiliza solo, normalmente se usa con agg o con otra función. Permite seleccionar todas las filas del DataFrame y las organiza en grupos que tienen el mismo valor en la(s) columna(s) que se indiquen. Por lo tanto, con groupBy(), Spark divide el DataFrame en grupos según el valor de una o más columnas pero después hay que decidir qué hacer con cada grupo. Ahí entran las funciones de agregación: son operaciones que toman todas las filas de un grupo y las convierten en un solo valor (por grupo).

Agg: Coje los valores de cara grupo generado anteriormente y aplica una fórmula para obtener un solo número o resultado por grupo. Operaciones básicas:

- Count(): Número de filas en el grupo.
    - "*": Cuenta cuántas filas tiene cada grupo, no importa si hay valores nulos.
    - "nombre_columna": Cuenta cuántas filas tiene cada grupo, pero no cuenta aquellas donde el valor de la columna "nombre_columna" sea nulo.
- CountDistict(): Cuenta valores únicos de una columna.
- Sum(): Suma todos los valores numéricos.
- Avg(): Calcula el promedio.
- Min(): Devuelve el valor mínimo del grupo.
- Max(): Devuelve el valor máximo del grupo.

```
# Agrupar por la columna "namecol" y calcular la edad media de la col Age

data_frame.groupBy("namecol").agg({"Age": "avg"}) grouped_df.show()
```

También se pueden emplear: variance, stddev…

UDF

Permite aplicar una función personalizada a una columna en un DataFrame. **Los UDFs son menos eficientes que las funciones nativas de Spark, porque ejecutan código Python en cada fila.**

Por eso solo se recomiendan cuando no existe una función nativa equivalente.

```
# Primero es necesario crear una función. El valor que se pasa, es el de la columna
def age_category(age):
  if age < 30:
```

```
    return "Young"
  else:
    return "Senior"

# Luego hay que registrar la función UDF
age_category_udf = udf(age_category, StringType())
# Por último se aplica la funcion
newdf=data_frame.withColumn("AgeCategory",age_category_udf(data_frame.namecol))
```

Replace

Se utiliza para sustituir partes de una cadena por otra. y devuelve una nueva cadena en la que todas las apariciones de texto_a_buscar se reemplazan por texto_nuevo. Por ejemplo, "hola.mundo".replace(".", "-") devuelve "hola-mundo". Es importante recordar que replace no modifica la cadena original (las cadenas en Python son inmutables), sino que devuelve una copia modificada.

```
cadena.replace("texto_a_buscar", "texto_nuevo")
```

RegExp Replace

Permite reemplazar partes de una cadena de texto que coincidan con una expresión regular (regexp). Su principal objetivo es buscar coincidencias de patrones dentro de una cadena y reemplazarlas por otra cadena definida por el usuario. “regexp_replace(columna/cadena, patrón, reemplazo)”. Es decir, se utiliza para reemplazar cadenas de texto en los VALORES de una columna usando expresiones regulares.

```
data_frame.select(col("*"),regexp_replace('name', '^Mi', 'mi').alias("new_name")).show()
```

o

```
data_frame.withColumn("Fecha", regexp_replace(col("Fecha"), "/", "-"))
```

Además. permite realizar transformaciones complejas sobre los datos buscando patrones. Por ejemplo, se puede o reordenar “dd/MM/yyyy” a “yyyy-MM-dd”. La primera parte, d{2} indica que se buscan dos dígitos, y spark le da el grupo 1. El siguiente d{2} también dos dígitos y le da el grupo 2. Ambos grupos están separados por “-“. Para poder indicar como queremos reordenar se usa “ $ “ y luego el numero del grupo que corresponde.

```
data_frame.df.withColumn(
        "Fecha",
        regexp_replace(col("Fecha"), r"(\d{2})-(\d{2})-(\d{4})", r"$3-$2-$1")
)
```

Es importante tener en cuenta:

- Opera sobre filas, no sobre nombres de columnas.
- Se usa junto con .withColumn() para sobrescribir la columna con el resultado.
- No sirve para renombrar columnas, para eso se usa withColumnRenamed().

Identificación de datos atípicos

Se pueden identificar valores que están fuera de un rango específico (outliers) utilizando la función when junto con col().

```
data_frame.select(
   col("name"),
   when(col("age") > 30, "yes").otherwise("no").alias("outlier")
).show()
```

Cambiar los tipos de las columnas

Un casting de columnas (o conversión de tipos) en un DataFrame es el proceso de cambiar el tipo de datos de una columna a otro tipo de datos. Conversiones más típicas:

- Cadenas de texto

```
data_frame = data_frame.withColumn("ID", col("ID").cast("string"))
```

- Enteros

```
data_frame = data_frame.withColumn("ID", col("ID").cast("int"))
```

- Decimal

```
data_frame = data_frame.withColumn("price", col("price").cast("decimal(10, 2)"))
```

- Fecha. Tiene que tener formato yyyy-MM-dd.

```
data_frame = data_frame.withColumn("start_date", col("start_date").cast("date"))
```

- Boolean

```
data_frame = data_frame.withColumn("is_active", col("is_active").cast("boolean"))
```

Explode

Se utiliza para expandir una columna de tipo array o mapa en varias filas, creando una nueva fila para cada elemento individual dentro del array o del mapa. Esta operación es útil cuando se tiene una columna con listas o mapas y se requiere "desempaquetarlos" para trabajar con cada valor de forma independiente. En los arrays toma cada elemento dentro del array y lo coloca en su propia fila.

```
data_frame.select(explode(col("columna_array"))).show()
```

Flatten

Se utiliza para aplanar arrays anidados (arrays dentro de otros arrays) en un solo array de nivel superior. Si hay una columna con múltiples niveles de arrays, flatten() fusiona todos los elementos en un solo array.

```
data_frame.select("name", flatten("multiple_arrays").show()
```

Structype

Se define la estructura de un esquema en un DataFrame, permitiendo especificar el tipo de cada columna. Un esquema es como el "esqueleto" de un DataFrame, y es útil cuando se quiere definir explícitamente cómo deben ser interpretados los datos (tipos de columnas, si son nulos o no, etc.). “nombre, tipo, nulos/nonulos”**. Es decir: Esto evita que Spark intente inferir el tipo de datos (lo que puede ser más lento en archivos grandes).**

```
schema_ = StructType([
   StructField("name", StringType(), True),
   StructField("age", IntegerType(), True),
   StructField("hobbies", ArrayType(StringType()), True),  # Array de strings
   StructField("address", StructType([               # Estructura anidada
      StructField("street", StringType(), True),
      StructField("city", StringType(), True)
   ]), True)
])

data = [ ("Maria", 25, ["leer", "bailar"], ("Calle 1", "Ciudad A")), ("Juan", 30, ["futbol"], ("Calle 2", "Ciudad B")), ]

data_frame = spark.createDataFrame(data, schema=schema_)
```

Hay que tener en cuenta que si se escribes col("address.street "), Spark lo interpreta como: Toma la columna address y accede a su campo Street.

Si se lee un string que contiene un punto como “K.P” pero no es un struct da un AnalysisException. Para poder usar las columnas que tengan un “.” hay que escapar los caracteres CON “ ` ”:

```
df.select(col("`K.P`")).show()
```

Por otro lado, cuando se quiere cargar directamente de un csv, hay que tener en cuenta que la opción “header” define si hay o no comprobación de cabecera, es decir, con (header”, False) Spark NO interpreta la primera fila como nombres de columnas. Si es True, buscar los nombres que hemos dado a los StructField en esa primera fila:

```
spark.read .option("header", False) .option("sep", sep) .csv(path)
```

Otros comandos como “contains” se pueden consultar en el siguiente enlace: https://spark.apache.org/docs/latest/sql-ref-syntax.html

Lag

En PySpark hay una función nativa llamada lag() que sirve justo para traer el valor de la fila anterior en una columna.

- Se define una ventana para decirle a Spark en qué orden mirar las filas.

```
wind = Window.orderBy("NombreColumnaparaOrden")
```

- Se crear nueva columna con el valor del día anterior

```
data_frame_anterior = data_frame.withColumn("NombreNuevoParaCol", lag("NombreCol").over(wind)
```

- Se puede calcular cualquier operación respecto al día anterior

```
df = data_frame_anterior .withColumn("ColOperacion", col("NombreCol ") + 2* col("NombreNuevoParaCol "))
```

PRÁCTICA:

Yahoo Finance recopila y pone a disposición pública datos financieros y bursátiles de las empresas que cotizan en el IBEX-35. Esta información se utiliza para múltiples fines, tales como el análisis de la salud financiera y operativa de las compañías, la valoración para inversión, el seguimiento del rendimiento del mercado y la comparación sectorial entre empresas. Los datos permiten observar tendencias temporales, puntos fuertes y débiles, así como riesgos potenciales asociados a cada empresa.

En concreto, la información que se va a analizar sobre los perfiles de empresa del IBEX-35 en Yahoo Finance puede incluir la siguiente información:

- Nombre de la empresa: Denominación oficial de la firma cotizada.
- Ticker / símbolo bursátil: Código que identifica la acción en el mercado.
- Historial de precios: Valores pasados de la acción.
- Fecha: Indica el día que se registró un valor de precio determinado.

El objetivo es desarrollar la capacidad de realizar un análisis inicial de los datos que se nos proporcionan, identificando tendencias y posibles áreas de interés para una interpretación posterior más profunda.

Para poder empezar a trabajar con spark, es necesario definir una sesión

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
from pyspark.sql.window import *

# Crear una SparkSession
spark_session = (SparkSession.builder
      .appName("IBEX35")
      .getOrCreate())
```

Además, se deben cargar los datos del archivo CSV utilizando “spark.read.option”. Si se establece la opción “header” en “True”, Spark interpretará la primera fila del archivo como los nombres de las columnas del DataFrame.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
from pyspark.sql.window import *

df = spark.read.option("header", True).option("sep", “;”).option("dateFormat", "dd/MM/yyyy").csv(path/from/csv)
```

Si “header” se establece en “False” spark no interpretará la primera fila como nombres de columnas, sino que la tratará como una fila más de datos. En ese caso, Spark asignará nombres genéricos a las columnas.

Además de ".option()" y ."csv", también se puede añadir ":schema" para inferir una estructura a los datos.

```
.schema(schema)
```

A continuación, realiza las siguientes **modificaciones/cálculos con las funciones VISTAS EN CLASE**. Indica claramente a que ejercicio pertenece cada sección del código (revisar formato de entrega al final del documento):

- Ej1-a: Carga los datos del archivo CSV en un DataFrame de PySpark, muestra el esquema para comprobar el tipo de cada columna y convierte la columna "Fecha" de tipo "string" a tipo "date" (y cualquier otra columna si fuera necesario). Finalmente, verifica que la conversión se haya realizado correctamente mostrando de nuevo el esquema y 6 filas del DataFrame.

- Ej1-b: Elimina el sufijo .MC de los nombres de todas las columnas. Este sufijo solo indica que las acciones cotizan en la Bolsa de Madrid (Mercado Continuo). Muestra los 6 primeros valores del DataFrame.

- Ej1-c (opcional): Define un StructType que permita cargar los datos directamente con el tipo correcto para cada columna y sustituya los nombres de los tickers por el nombre completo/siglas de la empresa en el esquema. Muestra dicha estructura y las 6 primeras filas del DataFrame.

- Ej2-a: Elimina las filas o columnas duplicadas o aquellas que no aporten información. Muestra CUANTAS filas se han eliminado. ¿De cuántas empresas hay información disponible en el DataFrame resultante? Muéstralo.

- Ej2-b: Determina cuál es el periodo temporal que cubren los datos (fecha inicial y fecha final) y muestra de cuantos días hay información disponible. Comenta si el resultado es coherente con lo esperado y comenta si consideras necesario buscar datos adicionales para completar el periodo y porqué (2 líneas máximo).

- Ej3: Estudia los datos y modifica de acuerdo con lo siguiente:

    - Renombra la columna "Fecha". a "Dia". Muestra 10 filas del DataFrame.
    - Calcula y muestra para cada empresa la media anual (Media anual), el valor máximo anual (Max anual) y el valor mínimo anual (Min anual) dentro del periodo de datos disponible
    - En la bolsa de EE. UU. existen requisitos mínimos de precio para una acción. Si cierra por debajo de 1 USD durante 30 días consecutivos, la empresa recibe un "Deficiency Notice". Con el objetivo de hacer un seguimiento de este suceso, crea una nueva columna booleana llamada "Deficiency Notice UNI" que sea "True" cuando el precio de cierre de la acción de la empresa UNI.MC esté por debajo de 1 €, y "False" en caso contrario, evaluando esta condición para cada día de forma independiente. Muestra 100 filas del DataFrame.

- Ej3-b (opcional): Investiga sobre qué tienen en común las empresas incluidas en el dataset (por qué se han escogido estas y no otras) e Investiga el origen de los valores

NULL presentes en los datos (1 línea máximo) (Añade la referencia de donde obtienes la información). Comenta si estos valores afectan a los cálculos realizados en los apartados anteriores y justifica tu respuesta (1 línea máximo).

- Ej4: Calcula la "Variación Anual" ((diferencia / inicial) * 100) que ha sufrido cada empresa, de forma que las clasifique en "Bajada Fuerte", "Bajada", "Neutra", "Subida" y "Subida Fuerte" en función del valor que tenían al comienzo del periodo y el que conservan al final (de todas las que puedas). Una variación fuerte es de igual o más de un 15% y una Neutra es sobre un 1%. Muestra el resultado.

- Ej5: Con el objetivo de hacer un seguimiento el año que viene, vamos a calcular para cada empresa, cuál es su distribución en ese periodo. Calcula la distribución de precios de cada empresa en el periodo disponible y, para cada sesión, añade una columna llamada "NombreEmpresaCuartil" que indique si el valor de ese día se encuentra en el q1, q2, q3 o q4. Muestra la primera fila del dataframe y Muestra las columnas completas correspondientes a las empresas AENA y BBVA del dataframe.

- Ej6 (opcional): Añade una nueva columna denominada "NombreEmpresaCambioSignificativo" que calcule la variación de cada acción de un día a otro. Si la variación absoluta es mayor al 8 %, la columna debe mostrar el valor de la variación; en caso contrario, debe mostrar un guion ("-"). Muestra la fila 15 del dataframe. Además, investiga a qué se deben esas variaciones tan significativas e interpreta el resultado (Añade la referencia/s de donde obtienes la información).

Una vez modificados los datos, se va a almacenar en una base de datos SQL. Para ello es necesario crear un Proyecto (Ej: practica 0), una base de datos denominada "IBEX35" y usar una conexión JDBC (se recomienda crear un archivo para la sesión de spark y otro archivo para la conexión jdbc, similar a otros proyectos que se han realizado).

Para poder conectar con la base de datos, es necesario añadir la siguiente configuración en la sesión spark. El "path" se corresponde con la ruta al .jar de mysql connector.

```
        .config('spark.driver.extraClassPath', self.path) \
```

Para poder almacenar un data frame como tabla en la base de datos se emplea WRITE JDBC. Se utiliza para escribir un DataFrame en una base de datos a través de JDBC (Java Database Connectivity). Esto permite exportar datos desde PySpark a una base de datos relacional como MySQL, PostgreSQL, SQL Server, Oracle, etc.

```
data_frame.write.jdbc(url, table, mode, properties)
```

Teniendo en cuenta los siguientes parámetros:

- URL: La URL de la base de datos JDBC a la cual se tiene que conectar. Formato: jdbc:mysql://servidor:puerto/base_de_datos

- TABLE: Especifica el nombre de la tabla a la cual se van a escribir los datos en la base de datos. Esta tabla puede ya existir o ser creada automáticamente (dependiendo del valor del parámetro mode).
- MODE: Define cómo manejar los datos en caso de que la tabla ya exista en la base de datos. Los valores más comunes son: ‘append', 'overwrite', 'ignore', 'error' o 'errorifexists.
- PROPERTIES: Contiene los detalles de conexión como el usuario, la contraseña y cualquier otra propiedad específica del conector JDBC. Ejemplo:

```
propiedades = {
  "driver": "com.mysql.cj.jdbc.Driver",
  "user": "root",
  "password": "****"
}
```

Acciones para realizar:

- En primer lugar, trata de almacenar los datos completos leídos del csv como una tabla denominada “Datos2024”.
- Posteriormente, almacena los datos que han sido tratados anteriormente (sin nuevas columnas) en una nueva tabla denominada “Datos2024”.

Otros comandos de conexión con la base de datos se pueden consultar en el siguiente enlace: https://spark.apache.org/docs/latest/sql-ref-syntax-aux-show.html

INSTRUCCIONES DE ENTREGA DEL DOCUMENTO

El incumplimiento de cualquiera de las siguientes instrucciones conllevará una calificación de “cero” en la práctica.

- La Práctica 0 debe entregarse en un archivo comprimido en formato .zip, con el siguiente nombre: NombreDeUsuarioUle_practica0.zip (por ejemplo: pverg_practica0.zip).

- El archivo .zip debe contener el PROYECTO COMPLETO, de manera que pueda ejecutarse en cualquier ordenador. Esto implica que no deben utilizarse rutas absolutas y deben incluirse todos los archivos necesarios para su funcionamiento, incluyendo los archivos .csv u otros recursos utilizados. En caso de no poderse ejecutar, la calificación es un “cero”.

- Todos los archivos que contengan código deben estar debidamente guardados con la extensión .py

- No es necesario incluir la base de datos SQL, pero sí debe incluirse el código necesario para conectarse a ella y utilizarla correctamente.

- El proyecto debe contar con una estructura clara y organizada, donde el código esté bien distribuido siguiendo el principio de responsabilidad única.

- Para responder a cada pregunta o ejercicio, se debe incluir un comentario indicando a cuál se refiere el código y un print() con el número del ejercicio. Por ejemplo:

```
 # Ej1-a
print(“Ej1-a”)
TODO
```

- En los ejercicios donde se indique “muestra”, es obligatorio imprimir por consola únicamente la información solicitada. No se debe mostrar información adicional.
    - Por ejemplo, en el Ej1-a, se solicita mostrar inicialmente la estructura con la que se cargan los datos. A continuación, se deben realizar una serie de operaciones sobre dichos datos y, posteriormente, volver a mostrarlos, incluyendo el contenido de algunas filas.

```
 # Ej1-a
print(“Ej1-a”)
TODO
df.printSchema()
TODO
df_new.printSchema()
df_new.show(5)
```

    - Por ejemplo, en el ejercicio Ej2-b, se solicita, comentar/investigar sobre un tema, para ello, se imprimirá un mensaje.

```
 # Ej2-b
```

```
print(“Ej2-b”)
print(“Comentario/Reflexion/Investigación”)
print(“Referencia/s”)
```

- Para la Práctica 0, se recomienda no utilizar herramientas de inteligencia artificial (IA), por los siguientes motivos:
    - Esta práctica se basa en conocimientos básicos fundamentales sobre los que se construirá el resto de la asignatura. No comprender estos conceptos dificultará considerablemente la realización de las siguientes prácticas.
    - En la evaluación final, se plantearán ejercicios similares a los de esta práctica, y no estará permitido el uso de herramientas de IA. Si no se dominan estos contenidos básicos, será imposible avanzar y resolver correctamente los ejercicios posteriores, que se plantearán de forma secuencial.

  Aun así, en caso de utilizar alguna herramienta de IA, es obligatorio indicar explícitamente qué herramienta se ha usado y en qué ejercicio (en el comentario donde se índique el número de ejercicio). El uso de herramientas de IA sin declarar se considerará como una infracción y la práctica será calificada con un “cero”.