Caso 1: Verificar el archivo de ventas de la tienda Zara.

Fuente de el archivo a Analizar	**Kaggle**
Dirección	*https://www.kaggle.com/datasets/michaelhakim/zara-sales-analysis*
Acerca del archivo	This Zara sales dataset is a comprehensive resource for e-commerce analytics and retail performance analysis. It captures critical sales data over a defined period, offering granular insights into product sales trends within Zara stores.
Columnas Disponibles	['Product ID','Product Position','Promotion', 'Product Category','Seasonal', 'Sales Volume', 'brand', 'url', 'sku', 'name', 'description', 'price', 'currency', 'scraped_at', 'terms', 'section']
Herramientas a utilizar en este ejercicio	1. Google Colab https://colab.research.google.com/ 2. Pydeequ https://pydeequ.readthedocs.io/en/latest/README.html
autor:	Enrique Davila enrique.davila@gmail.com

Objetivos

1. Verificar la calidad del dataset de ventas de Zara utilizando PyDeequ y PySpark, asegurando la integridad, consistencia y validez de los datos para un análisis de ventas confiable.

Ejercicios a desarrollar:

Preparación del ambiente de desarrollo.

1. Abrir la plataforma de colab

2. Subir el archivo de ventas Zara_Sales_Analysis.csv y Zara_Sales_Analysis_missing.csv. Ambos archivos contienen lo mismo, con excepción de que el que contiene la palabra missing le faltan algunos registros.
3. Instalar Pyspark y Pydeequ en Google colab:

```
# @title 1. Instalar Java, PySpark y PyDeequ
# Instalar Java Development Kit 8
!apt-get update -qq > /dev/null
!apt-get install -y openjdk-8-jdk-headless -qq > /dev/null

# Instalar PySpark (la versión que prefieras, compatible con tu Java)
!pip install pyspark==3.4.1 pydeequ

# Establecer las variables de entorno para Java
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["PATH"] += ":/usr/lib/jvm/java-8-openjdk-amd64/bin" # Asegura que java esté en el PATH
```

Autor: Enrique Davila enrique.davila@gmail.com

4. Crear una variable con la versión de spark

```
import os
# indicate your Spark version, here we use Spark 3.5 with pydeequ 1.4.0
os.environ["SPARK_VERSION"] = '3.3'
```

5. Inicializar spark junto con pydeequ

```
# @title 2. Iniciar la Sesión de Spark
import pydeequ
from pyspark.sql import SparkSession
# Iniciar la sesión de Spark, se añade la configuración para cargar las dependencias de Deequ.
# Es importante establecer la versión de Spark que usarás,
# aquí usamos Spark 3.4.1 que es compatible con pydeequ 1.2.0-spark-3.4
spark = (SparkSession
        .builder
        .appName("PyDeequ Colab Example")
        .config("spark.jars.packages", pydeequ.deequ_maven_coord)
        .config("spark.jars.excludes", "net.soreine.java:io.deequ-core") # Evita conflictos si ya está en el classpath
        .config("spark.driver.memory", "4g") # Aumenta la memoria del driver si trabajas con datos grandes
        .config("spark.executor.memory", "4g") # Aumenta la memoria del executor
        .config("spark.sql.warehouse.dir", "file:///tmp/spark-warehouse") # Directorio para el almacén de Spark
        .enableHiveSupport() # Habilita el soporte para Hive, útil para metadatos
        .getOrCreate())

print("Sesión de Spark iniciada correctamente.")
```

6. Verificar Sparkcontext este ejecutándose correctamente.

Autor: Enrique Davila enrique.davila@gmail.com

Ejercicios a desarrollar parte 2.

Cargando el archivo de ventas de Zara en pyspark y ejecutando algunas validaciones básicas.

1. Leer el archivo de ventas con pyspark, el archivo es un tipo csv.
2. Imprimir los primeros 10 registros dentro del archivo.

```
# Read CSV with semicolon (;) delimiter
df = spark.read.csv(
   '/content/Zara_Sales_Analysis_missing.csv',
   header=True,
   inferSchema=True,
   sep=';'  # or delimiter=';'
)
# Show the DataFrame
df.show(10,truncate=False)
```

3. Imprimir el schema(incluido el tipo de dato)

```
#Show the Schema
df.printSchema()
```

4. Imprimir el conteo de registros

5. Imprimir estadísticas de columnas numéricas(Price and Sales Volume)
6. Imprimir valores únicos para la columna Brand

Autor: Enrique Davila enrique.davila@gmail.com

Imprimer el conteo de registros

```
#Imprimir el conteo de registros
df.count()
```

Revisar información estadística con describe

```
#Imprimir estadísticas de columnas numéricas(Price and Sales Volume)
df_numeric_columns = df.select(['Sales Volume','price'])
df_numeric_columns.describe().show()
```

Ver valores unicos de la columna brand

```
#Imprimir valores únicos para la columna Brand
df.select('brand').distinct().show()
```

Autor: Enrique Davila enrique.davila@gmail.com

Usando Pydeequ

Buscando valores nulls en nuestro dataset

```
from pydeequ.checks import *
from pydeequ.verification import *

check = Check(spark, CheckLevel.Warning, "Completeness Check")

# Add completeness checks for all columns to the single check object
for columna in df.columns:
   check = check.hasCompleteness(columna, lambda completeness: completeness >= 1)



verification_result = (
   VerificationSuite(spark)
   .onData(df)
   .addCheck(check) # Add the single check object
   .run()
)

# Muestra resultados (filtrando solo columnas que fallan)
resultados = VerificationResult.checkResultsAsDataFrame(spark, verification_result)
resultados.show(truncate=False)
```

Validando la columna Brand

La columna brand solo debe de tener los valores Zara, otros valores son incorrectos

```
from pydeequ.checks import *
from pydeequ.verification import *

check = Check(spark, CheckLevel.Error, "Brand Validation Check")

# Check if 'brand' column ONLY contains 'Zara'
check = check.satisfies(
   "brand = 'Zara'",  # SQL condition (must evaluate to True)
   "Brand must be Zara",  # Constraint description
   lambda compliance: compliance >= 1.0  # 100% compliance required
)

# Run verification
result = VerificationSuite(spark).onData(df).addCheck(check).run()

# Show results
VerificationResult.checkResultsAsDataFrame(spark, result).show(truncate=False)
```

Autor: Enrique Davila enrique.davila@gmail.com

6 validaciones en un mismo VerificationSuite

---

1. Brand solo puede contener Zara / Levi
2. Precio no puede ser null/Negativo
3. Precio no puede ser igual a cero
4. Precio debe de ser en dólares
5. Product ID debe ser único
6. URL debe tener una dirección válida

```
from pydeequ.checks import *

from pydeequ.verification import *

verification_result = (

   VerificationSuite(spark)

   .onData(df)

   .addCheck(

       Check(spark, CheckLevel.Error, "Data Quality Checks")

       # 1. Brand must be "Zara" or "Levi's" (95% compliance allowed)

       .hasCompleteness("brand", lambda completeness: completeness >= 1)  # 100% completness

       .satisfies(

           "brand IN ('Zara', 'Levi')",
```

Autor: Enrique Davila enrique.davila@gmail.com

```
        "Brand must be Zara/Levi's",
        lambda compliance: compliance >= 1  # 95% of rows must comply
    )
    # 2. Price must be non-null (98%) and non-negative (100%)
    .hasCompleteness("price", lambda completeness: completeness >= 1)  # 100% non-null
    .isNonNegative("price")  # 100% compliance (no tolerance for negative prices)
    .satisfies(
    "price != 0",
    "Price must not be zero",
    lambda compliance: compliance >= 1  # 100% of rows must comply
    )
    # 3. Currency must be "USD" (99% compliance)
    .hasPattern("currency", "^USD$", lambda compliance: compliance >= 1)  # 100% exact matches


      # 4. New check: No duplicates in 'id' (100% uniqueness)
    .hasUniqueness(["Product ID"], lambda uniqueness: uniqueness >= 1)
)
```

Autor: Enrique Davila enrique.davila@gmail.com

```
    .run()

)

# Show only failures

result_df = VerificationResult.checkResultsAsDataFrame(spark, verification_result)

result_df.show(truncate=False)
```

URL Válida

```
from pydeequ.checks import *
from pydeequ.verification import *

check = Check(spark, CheckLevel.Error, "URL Validation Check")

# Regex for basic URL validation (adjust as needed)
url_regex = (
   r"^(https?:\/\/)?"  # http:// or https:// (optional)
   r"([\w\-]+(\.[\w\-]+)+)"  # Domain (e.g., google.com)
   r"([\w\-\._\~\:\/\?\#\[\]\@\!\$\&'\(\)\*\+\,\;\=]*)?"  # Optional path/query
)

check = check.hasPattern(
   "url",  # Replace with your column name
   url_regex,
   lambda compliance: compliance >= 1.0  # 100% compliance (adjust threshold)
)

result = VerificationSuite(spark).onData(df).addCheck(check).run()
VerificationResult.checkResultsAsDataFrame(spark, result).show(truncate=False)
```

Checks Disponibles en PyDeequ

Autor: Enrique Davila enrique.davila@gmail.com

En PyDeequ, los **checks** disponibles están definidos en la clase `Check` y se utilizan para validar la calidad de los datos en un DataFrame de Spark. Estos checks permiten verificar propiedades como completitud, unicidad, valores permitidos, patrones, rangos, entre otros. A continuación, te detallo los principales métodos de validación (checks) disponibles en PyDeequ, basándome en la documentación oficial y el uso común de la librería:

Los checks se aplican a través de la clase `Check` en el módulo `pydeequ.checks`. Cada método define una regla de calidad de datos que se evalúa sobre una columna o un conjunto de columnas en el DataFrame. Aquí están los más importantes:

- **Completitud y Nulidad**:
  - `.hasCompleteness(column, assertion)`: Verifica el porcentaje de valores no nulos en una columna.
    - Ejemplo: `.hasCompleteness("price", lambda x: x >= 0.98)` (98% de los valores deben ser no nulos).
  - `.isComplete(column)`: Verifica que una columna esté 100% completa (sin valores nulos).
    - Ejemplo: `.isComplete("id")`.
- **Unicidad**:
  - `.isUnique(column)`: Verifica que todos los valores en una columna sean únicos (100% de unicidad).
    - Ejemplo: `.isUnique("id")`.
  - `.hasUniqueness(columns, assertion)`: Verifica el porcentaje de filas únicas en una o más columnas.
    - Ejemplo: `.hasUniqueness(["id", "name"], lambda x: x >= 0.95)` (95% de las filas deben ser únicas).
  - `.isPrimaryKey(columns)`: Verifica que una o más columnas formen una clave primaria (valores únicos y no nulos).
    - Ejemplo: `.isPrimaryKey(["id"])`.

- **Valores Permitidos y Restricciones:**
  - `.satisfies(column_condition, constraint_name, assertion)`: Evalúa una condición SQL personalizada sobre una columna.
    - Ejemplo: `.satisfies("price > 0", "Price must be positive", lambda x: x >= 1)` (100% de los valores deben cumplir).
  - `.isContainedIn(column, allowed_values)`: Verifica que los valores de una columna estén en una lista específica.
    - Ejemplo: `.isContainedIn("brand", ["Zara", "Levi's"])` (todos los valores deben ser "Zara" o "Levi's").
  - `.hasPattern(column, pattern, assertion)`: Verifica que los valores de una columna cumplan con un patrón regex.
    - Ejemplo: `.hasPattern("currency", "^USD$", lambda x: x >= 0.99)` (99% deben ser "USD").
- **Restricciones Numéricas:**
  - `.isNonNegative(column)`: Verifica que todos los valores en una columna sean no negativos (≥ 0).
    - Ejemplo: `.isNonNegative("price")`.
  - `.hasMin(column, assertion)`: Verifica que el valor mínimo de una columna cumpla con una condición.
    - Ejemplo: `.hasMin("price", lambda x: x >= 0)` (el valor mínimo debe ser ≥ 0).
  - `.hasMax(column, assertion)`: Verifica que el valor máximo cumpla con una condición.
    - Ejemplo: `.hasMax("price", lambda x: x <= 1000)` (el valor máximo debe ser ≤ 1000).
  - `.hasMean(column, assertion)`: Verifica que el promedio de una columna cumpla con una condición.
    - Ejemplo: `.hasMean("price", lambda x: 50 <= x <= 100)`.
  - `.hasStandardDeviation(column, assertion)`: Verifica la desviación estándar de una columna.
    - Ejemplo: `.hasStandardDeviation("price", lambda x: x <= 10)`.
- **Tamaño del DataFrame:**
  - `.hasSize(assertion)`: Verifica el número total de filas en el DataFrame.
    - Ejemplo: `.hasSize(lambda x: x >= 1000)` (debe haber al menos 1000 filas).
- **Distribución y Estadísticas:**

    - `.hasApproxQuantile(column, quantile, assertion)`: Verifica que un cuantil aproximado cumpla con una condición.
        - Ejemplo: `.hasApproxQuantile("price", 0.5, lambda x: x >= 50)` (la mediana debe ser ≥ 50).
    - `.hasHistogramValues(column, assertion)`: Verifica la distribución de valores en una columna categórica.
        - Ejemplo: `.hasHistogramValues("category", lambda x: x["shirts"] >= 100)` (al menos 100 filas con categoría "shirts").
- **Correlación y Relaciones:**
    - `.hasCorrelation(column1, column2, assertion)`: Verifica la correlación entre dos columnas.
        - Ejemplo: `.hasCorrelation("price", "quantity", lambda x: abs(x) <= 0.5)` (correlación absoluta ≤ 0.5).
- **Conteo Distinto:**
    - `.hasDistinctness(columns, assertion)`: Verifica el porcentaje de valores distintos en una o más columnas.
        - Ejemplo: `.hasDistinctness(["brand"], lambda x: x <= 0.1)` (máximo 10% de valores distintos).
    - `.hasApproxCountDistinct(column, assertion)`: Verifica el conteo aproximado de valores distintos.
        - Ejemplo: `.hasApproxCountDistinct("brand", lambda x: x <= 10)` (máximo 10 valores distintos).
- **Validaciones de Tipo de Datos:**
    - `.hasDataType(column, data_type, assertion)`: Verifica el porcentaje de valores que cumplen con un tipo de datos específico (por ejemplo, `Conformance.IntegerType`).
        - Ejemplo: `.hasDataType("price", Conformance.IntegerType, lambda x: x >= 0.9)` (90% deben ser enteros).
- **Validaciones de Dependencia:**
    - `.hasConditionalDependency(column, dependent_column, assertion)`: Verifica dependencias condicionales entre columnas.
        - Ejemplo: Verificar que si `brand = 'Zara'`, entonces `price > 10`.
- **Validaciones de Entropía:**
    - `.hasEntropy(column, assertion)`: Verifica la entropía de una columna (medida de aleatoriedad).

        - Ejemplo: `.hasEntropy("category", lambda x: x >= 1.0)`.
- **Validaciones de Aproximación:**
    - `.hasApproxCount(column, assertion)`: Verifica un conteo aproximado de filas que cumplen una condición.
        - Ejemplo: `.hasApproxCount("price > 0", lambda x: x >= 100)`.

Notas Importantes:

- **Niveles de Check:** Los checks se definen con un nivel de severidad (`CheckLevel.Error` o `CheckLevel.Warning`). Si un check falla con `Error`, el proceso puede detenerse; con `Warning`, solo se registra.
    - Ejemplo: `Check(spark, CheckLevel.Error, "Data Quality Checks")`.
- **Assertions:** La mayoría de los checks aceptan una función lambda para definir el umbral de cumplimiento (por ejemplo, `lambda x: x >= 0.95` para un 95% de cumplimiento).
- **Ejecución:** Los checks se ejecutan con `.run()` y los resultados se obtienen en un `VerificationResult`, que puede inspeccionarse para ver qué reglas pasaron o fallaron.
- **Múltiples Columnas:** Algunos checks, como `.hasUniqueness` o `.isPrimaryKey`, permiten pasar una lista de columnas para validar combinaciones.
- **Personalización:** Para reglas más complejas, `.satisfies()` permite usar expresiones SQL arbitrarias, lo que da gran flexibilidad.

[1]Ejemplo Completo:

[1]

```
python

from pydeequ.checks import Check, CheckLevel
from pydeequ.verification import VerificationSuite

check = Check(spark, CheckLevel.Error, "Data Quality Checks") \
    .hasCompleteness("price", lambda x: x >= 0.98) \
    .isNonNegative("price") \
    .satisfies("price != 0", "Price must not be zero", lambda x: x >= 1) \
    .isContainedIn("brand", ["Zara", "Levi's"]) \
    .hasPattern("currency", "^USD$", lambda x: x >= 0.99) \
    .hasSize(lambda x: x >= 1000)

verification_result = VerificationSuite(spark) \
    .onData(df) \
    .addCheck(check) \
    .run()

verification_result.checkResultsAsDataFrame(spark).show()
```

Recursos Adicionales:

- **Documentación Oficial**: La documentación de PyDeequ en GitHub (https://github.com/awslabs/python-deequ) detalla todos los métodos disponibles.
- **API de Deequ**: PyDeequ es un wrapper de Deequ (Scala), por lo que los checks disponibles en Deequ también aplican (https://github.com/awslabs/deequ).
- **Ejemplos Prácticos**: Revisa los ejemplos en el repositorio de PyDeequ para casos de uso