Módulo 1 Utilización, procesamiento y visualización de Big Data - CLIMA

---

kevin Alejandro Ramírez Luna | A01711063

Abstract y prorrogas iniciales:

El siguiente collab tiene como objetivo crear un PRONOSTICO DEL TIEMPO usando el dataset de Kaggle `Brazil Weather, Automatic Stations (2000-2021)` Referencia. Donde se busca determinar si es que LLUEVE O NO LLUEVE

Para la visualización, se creo el siguiente Tableu para ver los resultados de nuestro modelo.

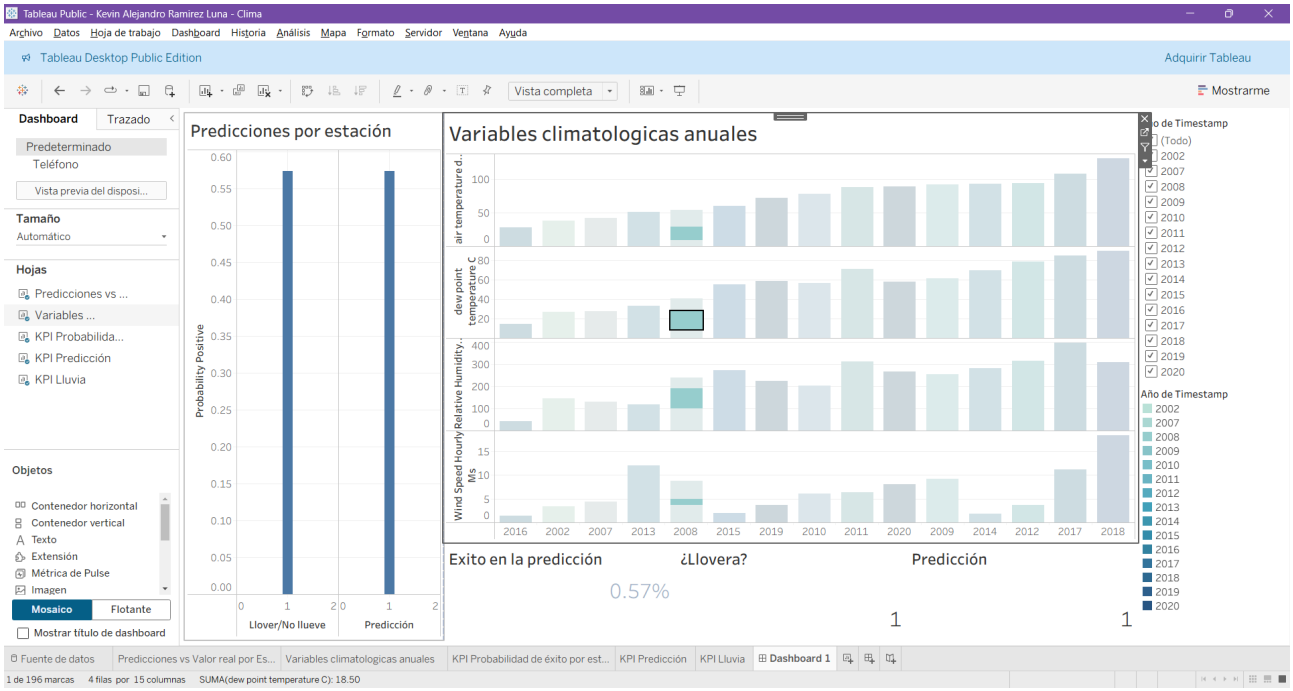
Para ello, estaremos usando tecnicas de ml para poder trabajar con esta problematica

El Stack tecnologico que se eligio para esta tarea esta comprendido por:

- `Python` - Como lenguaje principal
  - `Numpy` - Operaciones númericas
  - `Pandas` - Trabajo con Dataframes
  - `PySpark` - Para manejo de Big Data y modelos de ML
- `Google Collab + Kaggle` - Enviroment de trabajo y manejo del dataset a usar
- `Tableu` - Visualización de resultados

La ruta de trabajo a realizar se detalla a continuación:

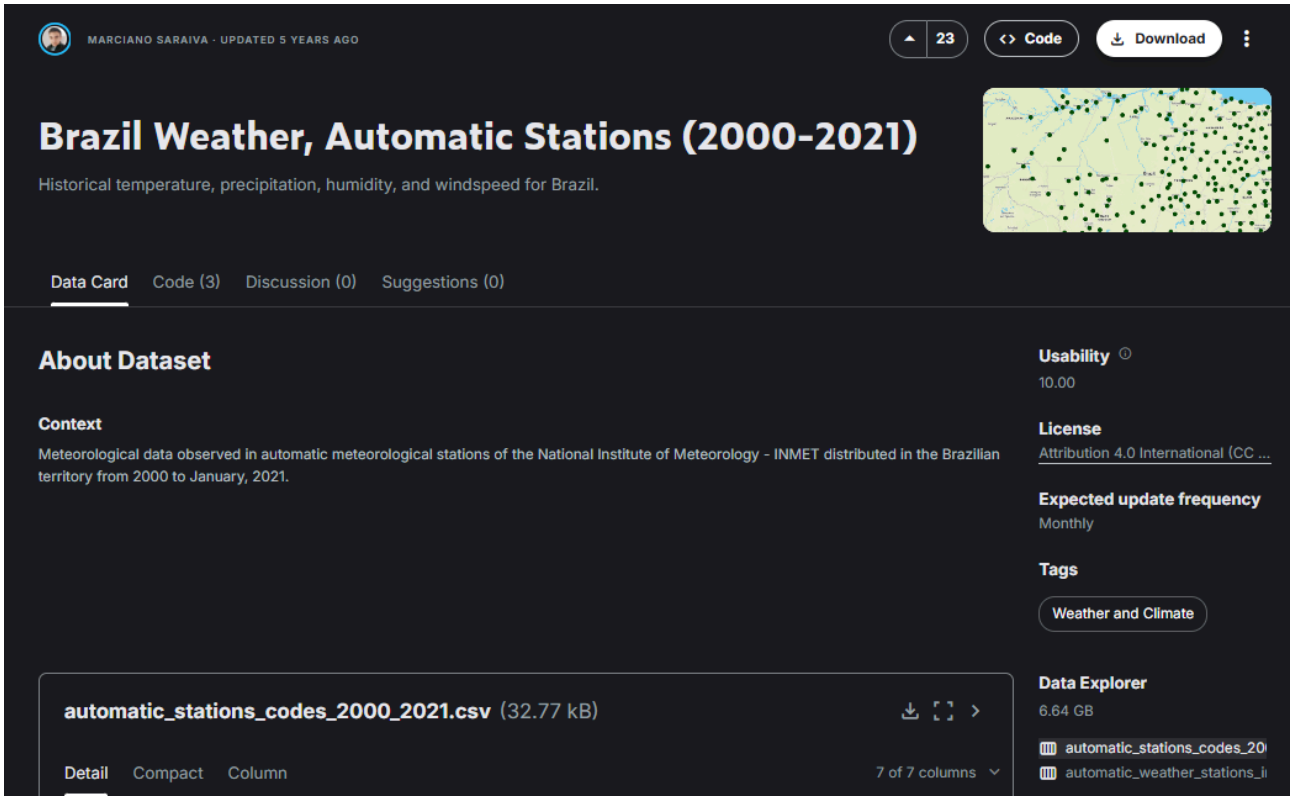
1. Environment Setup (Kaggle + Spark)
2. EDA-ETL (Análisis exploratorio y limpieza)
3. Modelado
4. Train/Test Split & Métricas
5. Test
6. Tableau Dashboard (Síntesis visual) Link: Tableu



1. Preparación de Kaggle y el entorno principal con SPARK

Para la parte de Big-Data estaremos trabajando con el dataset anteriormente mencionado. Donde antes, viendo la descripción de la página oficial podemos sacar las siguientes cosas:

- **Contexto:** Meteorological data observed in automatic meteorological stations of the National Institute of Meteorology - INMET distributed in the Brazilian territory from 2000 to January, 2021.
- `Tamaño` - 6.64 GB
- `No. Archivos` - 2 Archivos .csv
- `No. Features`:
    - 7 features en el archivo automatic_stations_codes_2000_2021.csv (A descartar por que viendo en Kaggle, la mayoria de sus fetures son categoricas - Representan lugares y coordenadas y el proposito del trabajo no irá por ahí)
    - 20 features en el archivo automatic_weather_stations_inmet_brazil_2000_2021.csv



Paso 1.1 Montar Kaggle con Collab

El proceso para montar el dataset requiere de la configuración de variables de entorno de Colab, creando con anterioridad un token para poder usar la API de kaggle [Documentación]

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
import os
os.environ['KAGGLE_CONFIG_DIR'] = "/content/gdrive/MyDrive/Inteligencia Artific


%cd /content/gdrive/MyDrive/Inteligencia Artificial Avanzada/
```

```
[Errno 2] No such file or directory: '/content/gdrive/MyDrive/Inteligencia Artif
/content
```

Descarga del dataset por medio de la API de Kaggle

```
#!/bin/bash#!/bin/bash
!kaggle datasets download saraivaufc/automatic-weather-stations-brazil
```

```
Dataset URL: https://www.kaggle.com/datasets/saraivaufc/automatic-weather-static
License(s): Attribution 4.0 International (CC BY 4.0)
Downloading automatic-weather-stations-brazil.zip to /content
 99% 1.39G/1.40G [00:10<00:00, 250MB/s]
100% 1.40G/1.40G [00:10<00:00, 137MB/s]
```

Fuente para visualizar el proceso con más detalle

Descomprimimos el zip generado por Kaggle

```
# Creamos una carpeta nueva para guardar todos los archivos que se descomprimar
!mkdir brasil
```

```
mkdir: cannot create directory ‘brasil’: File exists
```

```
!mv automatic-weather-stations-brazil.zip /content/brasil/
```

```
!unzip /content/brasil/automatic-weather-stations-brazil.zip -d /content/brasil
```

```
Archive:  /content/brasil/automatic-weather-stations-brazil.zip
  inflating: /content/brasil/automatic_stations_codes_2000_2021.csv
  inflating: /content/brasil/automatic_weather_stations_inmet_brazil_2000_2021.c
```

Paso 1.2 Configuración de Pyspark

```
#Bibliotecas para poder trabajar con Spark
!sudo apt update
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q https://downloads.apache.org/spark/spark-3.5.6/spark-3.5.6-bin-hadoop3
!tar xf spark-3.5.6-bin-hadoop3.tgz
#Configuración de Spark con Python
!pip install -q findspark
!pip install pyspark

#Estableciendo variable de entorno
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.5.6-bin-hadoop3"

#Buscando e inicializando la instalación de Spark
import findspark
```

```
findspark.init()
findspark.find()
```

```
Get:1 https://cloud.r-project.org/bin/linux/ubuntu jammy-cran40/ InRelease [3,63
Hit:2 https://cli.github.com/packages stable InRelease
Hit:3 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_64
Get:4 http://security.ubuntu.com/ubuntu jammy-security InRelease [129 kB]
Hit:5 http://archive.ubuntu.com/ubuntu jammy InRelease
Get:6 https://r2u.stat.illinois.edu/ubuntu jammy InRelease [6,555 B]
Get:7 https://cloud.r-project.org/bin/linux/ubuntu jammy-cran40/ Packages [83.2
Get:8 http://archive.ubuntu.com/ubuntu jammy-updates InRelease [128 kB]
Get:9 https://r2u.stat.illinois.edu/ubuntu jammy/main amd64 Packages [2,820 kB]
Get:10 http://security.ubuntu.com/ubuntu jammy-security/universe amd64 Packages
Hit:11 https://ppa.launchpadcontent.net/deadsnakes/ppa/ubuntu jammy InRelease
Hit:12 https://ppa.launchpadcontent.net/graphics-drivers/ppa/ubuntu jammy InRele
Get:13 http://archive.ubuntu.com/ubuntu jammy-backports InRelease [127 kB]
Hit:14 https://ppa.launchpadcontent.net/ubuntugis/ppa/ubuntu jammy InRelease
Get:15 http://security.ubuntu.com/ubuntu jammy-security/main amd64 Packages [3,4
Get:16 https://r2u.stat.illinois.edu/ubuntu jammy/main all Packages [9,411 kB]
Get:17 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 Packages [3,847
                                                                          ges [1
Get:19 http://archive.ubuntu.com/ubuntu jammy-updates/restricted amd64 Packages
Fetched 29.1 MB in 3s (9,512 kB/s)
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
51 packages can be upgraded. Run 'apt list --upgradable' to see them.
W: Skipping acquire of configured file 'main/source/Sources' as repository 'http
Requirement already satisfied: pyspark in /usr/local/lib/python3.12/dist-package
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.12/dist-
'/content/spark-3.5.6-bin-hadoop3'
```

Paso 1.3 Crear la sesión de trabajo de Spark y dataframe

Ya seleccionado y visto el conjunto de datos comencemos a trabajar con PySpark. Para comenzar a trabajar con PySpark, debemos iniciar la sesión de Spark. Para esto realizaremos lo siguiente:

1. Importar SparkSession
2. Crear la sesión

NOTA: Si se cae la sesión, se recomienda reiniciar la sesión de colab y volver a correr la celda de abajo

```
#Verificar la funcionalidad de Pyspark
from pyspark.sql import SparkSession
spark_session = SparkSession.builder.appName('PySpark_Forecasting').getOrCreate
spark_session
```

SparkSession - in-memory

SparkContext

Spark UI

Version
v3.5.6
Master
local[*]
AppName
PySpark_Forecasting

```
df_spark = spark_session.read.csv('/content/brasil/automatic_weather_stations_i
df_spark
```

```
DataFrame[_c0: string, _c1: string, _c2: string, _c3: string, _c4: string, _c5:
string, _c6: string]
```

```
df_spark.show(5)
```

```
+--------------------+--------------------+--------------------+----------------
|                 _c0|                 _c1|                 _c2|               _
+--------------------+--------------------+--------------------+----------------
|ESTACAO;DATA (YYY...| HORARIA (mB);PRE...| HORARIA (C);TEMP...| HORARIA (%);VEN
|A001;2021-01-01;0...|                NULL|                NULL|              NU
|A001;2021-01-01;0...|                NULL|                NULL|              NU
|A001;2021-01-01;0...|                NULL|                NULL|              NU
|A001;2021-01-01;0...|                NULL|                NULL|              NU
+--------------------+--------------------+--------------------+----------------
only showing top 5 rows
```

Reedefinimos el header y reenombramos a nombres más claros para nosotros

```
df_spark_col  = spark_session.read.option('header', 'true').option('delimiter',

new_column_names = [
    "station",
    "date",
    "time_utc",
    "precipitation_hourly_mm",
    "atmospheric_pressure_station_level_mB",
    "atmospheric_pressure_max_hourly_mB",
    "atmospheric_pressure_min_hourly_mB",
    "global_radiation_Wm2",
    "air_temperature_dry_bulb_C",
    "dew_point_temperature_C",
    "temperature_max_hourly_C",
    "temperature_min_hourly_C",
```

```
    "dew_point_max_hourly_C",
    "dew_point_min_hourly_C",
    "relative_humidity_max_hourly_percent",
    "relative_humidity_min_hourly_percent",
    "relative_humidity_hourly_percent",
    "wind_direction_hourly_degrees",
    "wind_gust_max_ms",
    "wind_speed_hourly_ms"
]

# Renombramos columnas
df_spark_col = df_spark_col.toDF(*new_column_names)

df_spark_col
```

```
DataFrame[station: string, date: string, time_utc: string,
precipitation_hourly_mm: string, atmospheric_pressure_station_level_mB: string,
atmospheric_pressure_max_hourly_mB: string, atmospheric_pressure_min_hourly_mB:
string, global_radiation_Wm2: string, air_temperature_dry_bulb_C: string,
dew_point_temperature_C: string, temperature_max_hourly_C: string,
temperature_min_hourly_C: string, dew_point_max_hourly_C: string,
dew_point_min_hourly_C: string, relative_humidity_max_hourly_percent: string,
relative_humidity_min_hourly_percent: string, relative_humidity_hourly_percent:
string, wind_direction_hourly_degrees: string, wind_gust_max_ms: string,
wind_speed_hourly_ms: string]
```

```
df_spark_col.show()
```

```
+-------+----------+--------+----------------------+---------------------------
|station|      date|time_utc|precipitation_hourly_mm|atmospheric_pressure_static
+-------+----------+--------+----------------------+---------------------------
|   A001|2021-01-01|    0000|                   0.0|
|   A001|2021-01-01|    0100|                   0.0|
|   A001|2021-01-01|    0200|                   0.0|
|   A001|2021-01-01|    0300|                   0.0|
|   A001|2021-01-01|    0400|                   0.0|
|   A001|2021-01-01|    0500|                   0.0|
|   A001|2021-01-01|    0600|                   0.0|
|   A001|2021-01-01|    0700|                   0.0|
|   A001|2021-01-01|    0800|                   0.0|
|   A001|2021-01-01|    0900|                   0.0|
|   A001|2021-01-01|    1000|                   0.0|
|   A001|2021-01-01|    1100|                   0.0|
|   A001|2021-01-01|    1200|                   0.0|
|   A001|2021-01-01|    1300|                   0.0|
|   A001|2021-01-01|    1400|                   0.0|
|   A001|2021-01-01|    1500|                   0.0|
|   A001|2021-01-01|    1600|                   0.0|
|   A001|2021-01-01|    1700|                   0.0|
|   A001|2021-01-01|    1800|                   1.2|
|   A001|2021-01-01|    1900|                   0.2|
+-------+----------+--------+----------------------+---------------------------
only showing top 20 rows
```

Paso 2: EDA-ETL

Una vez descargado, conectado y listo para trabajar nuestro dataset, tenemos que ver con que nos enfrentamos. Que cosas tenemos, nos pueden servir y utilizaremos para la étapa de modelado. Así que, en los siguientes apartados aplicaremos un poco de estadistica descriptiva, graficaciones y transformaciones de datos. Esto con el fin de coomprender mejor la naturaleza de nuestro problema

Paso 2.1 Exploración inicial - Cantidad de registros, rangos de fechas y tipos de datos

```
from pyspark.sql import functions as F
# Verificar el esquema de los datos
df_spark_col.printSchema()

# Contar registros totales
print(f"Total de registros: {df_spark_col.count():,}")

# Ver estaciones únicas
print(f"Estaciones únicas: {df_spark_col.select('station').distinct().count()}"

# Rango de fechas

df_spark_col.select(
    F.min('date').alias('fecha_minima'),
    F.max('date').alias('fecha_maxima')
).show()
```

```
root
 |-- station: string (nullable = true)
 |-- date: string (nullable = true)
 |-- time_utc: string (nullable = true)
 |-- precipitation_hourly_mm: string (nullable = true)
 |-- atmospheric_pressure_station_level_mB: string (nullable = true)
 |-- atmospheric_pressure_max_hourly_mB: string (nullable = true)
 |-- atmospheric_pressure_min_hourly_mB: string (nullable = true)
 |-- global_radiation_Wm2: string (nullable = true)
 |-- air_temperature_dry_bulb_C: string (nullable = true)
 |-- dew_point_temperature_C: string (nullable = true)
 |-- temperature_max_hourly_C: string (nullable = true)
 |-- temperature_min_hourly_C: string (nullable = true)
 |-- dew_point_max_hourly_C: string (nullable = true)
 |-- dew_point_min_hourly_C: string (nullable = true)
 |-- relative_humidity_max_hourly_percent: string (nullable = true)
```

```
 |-- relative_humidity_min_hourly_percent: string (nullable = true)
 |-- relative_humidity_hourly_percent: string (nullable = true)
 |-- wind_direction_hourly_degrees: string (nullable = true)
 |-- wind_gust_max_ms: string (nullable = true)
 |-- wind_speed_hourly_ms: string (nullable = true)

Total de registros: 60,452,376
Estaciones únicas: 612
+------------+------------+
|fecha_minima|fecha_maxima|
+------------+------------+
|  2000-05-07|  2021-01-31|
+------------+------------+
```

2.2 Uso de Time Series y clasificación Binaria

Como queremos hacer el analisis del clima en alguna región es importante que consideremos el problema como una `Time-Series` (que corresponde del periodo de 2000-05-07 a 2021-01-31) para, con los valores de clima pasados, hacer una predicción del futuro.

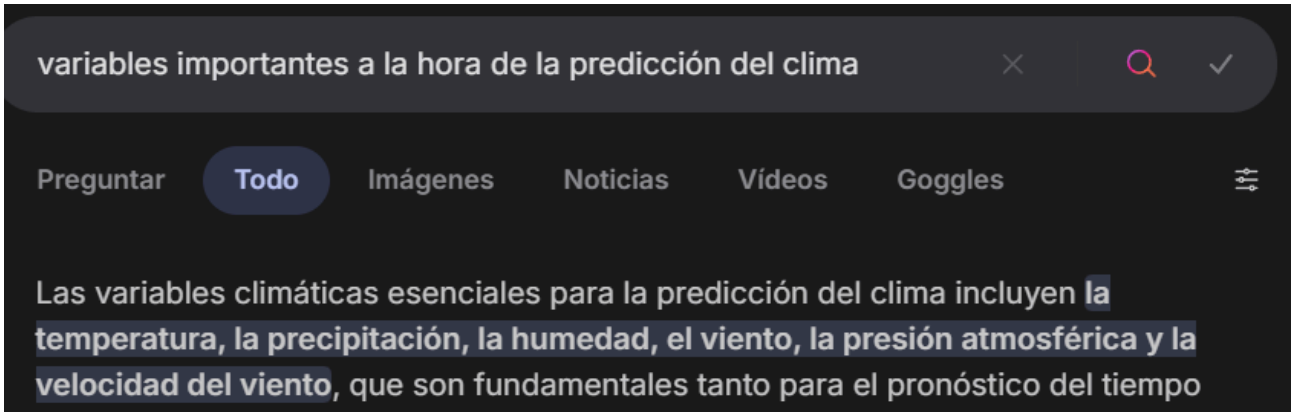
Entonces, tenemos que reetructurar nuestro problema a este tipo de formato.

Según la página Scielo.org lo podemos aproximar como una serie ARIMA:

> Hay grandes progresos en el desarrollo y las aplicaciones de la predicción del clima a mediano plazo y su predicción estacional (Vitart et al. 2012). Los algoritmos de predicción automáticas más usados son con base en el suavizado exponencial o modelos autorregresivos integrados de media móvil (ARIMA) (Hyndman y Khandakar, 2008). Box y Jenkins (1976) desarrollaron la metodología clásica que emplea las series de tiempo para generar modelos como el autoregresivo de media móvil (ARMA) o también el modelo ARIMA para obtener predicciones.

Sin embargo, por temas de que el ARIMA solo considera una variable, algoritmos de ML pueden ser más optimos. Este punto se tocará a más detalle en el **Paso 4**

Ahora, ¿Que datos son relevantes para el forecasting del clima?



Buscando diversas fuentes de información las variables que se llegan a considerar para el forecasting del clima son `temperatura`, `precipitación`, `humedad` y `viento`. Viendo que si tenemos estas variables,las podemos reeconsiderar como un time series. Donde, justamente buscamos que variables relacionadas a estos rubros sean las variables que el modelo podría usar

Pero ahora bien, ¿Que queremos predecir a ciencia cierta?, ¿Que pretendemos lograr?

Para simplicidad del modelado, y considerando nuestros recursos, tiempo, y que estamos trabajando con un número de datos bastante amplio, vamos a acotar el problema a un problema binario: **llueve o no llueve en una región**.

Para esto, primero vamos a restructurar nuestros datos, quedarnos con las features importantes para el analisis y restructurar la feature `precipitation_hourly_mm` a una variable binaria para la clasificación

```
from pyspark.sql import functions as F
from pyspark.sql.types import DoubleType, TimestampType
```

```
from pyspark.ml.feature import VectorAssembler, StandardScaler
from pyspark.ml.classification import RandomForestClassifier, GBTClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClas
from pyspark.ml import Pipeline
import numpy as np

print("=== PREDICCIÓN DE PROBABILIDAD DE PRECIPITACIÓN ===")

# DEFINIR COLUMNAS ESENCIALES
essential_columns = [
    'station', 'date', 'time_utc',
    'precipitation_hourly_mm',  # Ahora es nuestro TARGET
    'air_temperature_dry_bulb_C',   # Feature importante
    'relative_humidity_hourly_percent',  # Feature clave para lluvia
    'wind_speed_hourly_ms',         # Feature
    'atmospheric_pressure_station_level_mB',  # Feature
    'dew_point_temperature_C'       # Feature muy relacionada con lluvia
]

# CREAR DATASET REDUCIDO
print("Creando dataset para predicción de lluvia...")
df_reduced = df_spark_col.select(essential_columns)

# Función segura para conversión de tipos
def safe_cast_to_double(df, column_name):
    return df.withColumn(
        column_name,
        F.when(
            F.col(column_name).rlike('^-?[0-9]*\\.?[0-9]+$'),
            F.col(column_name).cast(DoubleType())
        ).otherwise(F.lit(None))
    )

# CONVERTIR COLUMNAS NUMÉRICAS
numeric_columns = [
    'precipitation_hourly_mm',  # Target original (continuo)
    'air_temperature_dry_bulb_C',
    'relative_humidity_hourly_percent',
    'wind_speed_hourly_ms',
    'atmospheric_pressure_station_level_mB',
    'dew_point_temperature_C'
]

print("Convirtiendo columnas numéricas...")
df_clean = df_reduced
for col_name in numeric_columns:
    df_clean = safe_cast_to_double(df_clean, col_name)

# TRANSFORMAR EL TARGET: De regresión a clasificación
df_classification = df_clean.withColumn(
    'lluvia_binaria',
```

```
    F.when(F.col('precipitation_hourly_mm') > 0, 1)  # 1 = Llueve
     .otherwise(0)                                   # 0 = No llueve
)

# Análisis de la distribución
lluvia_stats = df_classification.groupBy('lluvia_binaria').count().collect()
total = df_classification.count()

print("Distribución de clases:")
for row in lluvia_stats:
    clase = "LLUVIA" if row['lluvia_binaria'] == 1 else "NO LLUVIA"
    porcentaje = (row['count'] / total) * 100
    print(f"  {clase}: {row['count']} registros ({porcentaje:.1f}%)")

# TIMESTAMP para features temporales
print("\nCreando timestamp unificado...")
df_time = df_classification.withColumn(
    'time_utc_padded', F.lpad(F.col('time_utc'), 4, '0')
).withColumn(
    'hour_str', F.substring('time_utc_padded', 1, 2)
).withColumn(
    'minute_str', F.substring('time_utc_padded', 3, 2)
).withColumn(
    'time_formatted', F.concat(F.col('hour_str'), F.lit(':'), F.col('minute_str
).withColumn(
    'timestamp',
    F.to_timestamp(
        F.concat(F.col('date'), F.lit(' '), F.col('time_formatted')),
        'yyyy-MM-dd HH:mm'
    )
).drop('time_utc_padded', 'hour_str', 'minute_str', 'time_formatted')

# FEATURES TEMPORALES (importantes para predicción de lluvia)
df_features = df_time.withColumn(
    'hora_del_dia', F.hour('timestamp')
).withColumn(
    'mes', F.month('timestamp')
).withColumn(
    'estacion', # Por mes, agrupamos los meses a su respectiva estación
    F.when(F.month('timestamp').isin(12, 1, 2), 0)  # Invierno
     .when(F.month('timestamp').isin(3, 4, 5), 1)   # Primavera
     .when(F.month('timestamp').isin(6, 7, 8), 2)   # Verano
     .otherwise(3)                                   # Otoño
)

# DATASET FINAL
df_final = df_features.orderBy('station', 'timestamp')

print(f"\n=== DATASET FINAL PARA CLASIFICACIÓN ===")
print(f"Total registros: {df_final.count()}")
print(f"Variable objetivo: 'lluvia_binaria' (1=Llueve, 0=No llueve)")
```

```
print(f"Features: {[col for col in df_final.columns if col not in ['lluvia_bina

df_final.select('lluvia_binaria', 'precipitation_hourly_mm', 'air_temperature_d
                'relative_humidity_hourly_percent', 'hora_del_dia').show(10)
```

```
=== PREDICCIÓN DE PROBABILIDAD DE PRECIPITACIÓN ===
Creando dataset para predicción de lluvia...
Convirtiendo columnas numéricas...
Distribución de clases:
  LLUVIA: 3781307 registros (6.3%)
  NO LLUVIA: 56671069 registros (93.7%)

Creando timestamp unificado...

=== DATASET FINAL PARA CLASIFICACIÓN ===
Total registros: 60452376
Variable objetivo: 'lluvia_binaria' (1=Llueve, 0=No llueve)
Features: ['air_temperature_dry_bulb_C', 'relative_humidity_hourly_percent', 'wi
+--------------+----------------------+--------------------------+-------------
|lluvia_binaria|precipitation_hourly_mm|air_temperature_dry_bulb_C|relative_humi
+--------------+----------------------+--------------------------+-------------
|             0|               -9999.0|                   -9999.0|
|             0|               -9999.0|                   -9999.0|
|             0|               -9999.0|                   -9999.0|
|             0|               -9999.0|                   -9999.0|
|             0|               -9999.0|                   -9999.0|
|             0|               -9999.0|                   -9999.0|
|             0|               -9999.0|                   -9999.0|
|             0|               -9999.0|                   -9999.0|
|             0|               -9999.0|                   -9999.0|
|             0|               -9999.0|                   -9999.0|
+--------------+----------------------+--------------------------+-------------
only showing top 10 rows
```

Como puntos a observar:

- Tenemo cantidades de registros de más de +1M, por lo que será importante que esto lo tengamos en cuenta a la hora de ir trabajando en el modelo y de plantear una solución
- Los datos muestran -9999.0 en todas las variables, lo que indica valores NaN o de error. También tenemos un desequilibrio severo de clases de 6.3% vs 93.7%.

  ¿Esto a que se debe? Lo podemos interpretar como que en si, **Hay más días en los que NO llueve** a los que si. Entonces, esto crea un desvalance importante en nuestro modelo

  Un modelo de ML podría simplemente predecir que " NO HAY LLUVIA" siempre. Por ejemplo, podemos tener un accurency super bueno del 93.7% que no esta

aprendiendo a diferenciar entre las clases por el bajo numero de datos en una clase. Es decir, el dataset esta `sesgado`

Así que vamos a solucionar estos problemas:

- Eliminar los valores nulos y de -9999
- Manejar la distribución entre clases viendo si mejora déspues de eliminar los datos

```
print("=== LIMPIEZA Y PREPROCESAMIENTO DE DATOS ===")

# 1. FILTRAR VALORES VÁLIDOS (eliminar -9999.0)
print("Filtrando valores válidos...")
df_valid = df_final.filter(
    (F.col('air_temperature_dry_bulb_C') != -9999.0) &
    (F.col('relative_humidity_hourly_percent') != -9999.0) &
    (F.col('wind_speed_hourly_ms') != -9999.0) &
    (F.col('atmospheric_pressure_station_level_mB') != -9999.0) &
    (F.col('dew_point_temperature_C') != -9999.0) &
    (F.col('precipitation_hourly_mm') != -9999.0)
)

print(f"Registros después de filtrar valores inválidos: {df_valid.count()}")
print(f"Registros eliminados: {df_final.count() - df_valid.count()}")

# 2. VERIFICAR NUEVA DISTRIBUCIÓN
lluvia_stats_valid = df_valid.groupBy('lluvia_binaria').count().collect()
total_valid = df_valid.count()

print("\nNueva distribución después de limpieza:")
for row in lluvia_stats_valid:
    clase = "LLUVIA" if row['lluvia_binaria'] == 1 else "NO LLUVIA"
    porcentaje = (row['count'] / total_valid) * 100
    print(f"  {clase}: {row['count']} registros ({porcentaje:.1f}%)")

# 3. ANÁLISIS EXPLORATORIO RÁPIDO
print("\n=== ESTADÍSTICAS DESCRIPTIVAS ===")
df_valid.select([
    'air_temperature_dry_bulb_C',
    'relative_humidity_hourly_percent',
    'wind_speed_hourly_ms',
    'atmospheric_pressure_station_level_mB',
    'dew_point_temperature_C'
]).describe().show()

# 4. MUESTRA DE DATOS VÁLIDOS
print("Muestra de datos válidos:")
df_valid.select(
    'lluvia_binaria', 'precipitation_hourly_mm', 'air_temperature_dry_bulb_C',
    'relative_humidity_hourly_percent', 'hora_del_dia'
).filter(F.col('lluvia_binaria') == 1).limit(10).show()
```

```
=== LIMPIEZA Y PREPROCESAMIENTO DE DATOS ===
Filtrando valores válidos...
Registros después de filtrar valores inválidos: 49762128
Registros eliminados: 10690248

Nueva distribución después de limpieza:
  LLUVIA: 3552208 registros (7.1%)
  NO LLUVIA: 46209920 registros (92.9%)

=== ESTADÍSTICAS DESCRIPTIVAS ===
+-------+--------------------------+--------------------------------+-----------
|summary|air_temperature_dry_bulb_C|relative_humidity_hourly_percent|wind_speed_
+-------+--------------------------+--------------------------------+-----------
|  count|                  49762128|                        49762128|
|   mean|         23.41996340871865|               71.77112660857269|   2.047844
| stddev|         5.558504346181661|               20.20029961473358|  1.7217547
|    min|                      -7.7|                             5.0|
|    max|                      45.0|                           100.0|
+-------+--------------------------+--------------------------------+-----------

Muestra de datos válidos:
+-------------+----------------------+--------------------------+-------------
|lluvia_binaria|precipitation_hourly_mm|air_temperature_dry_bulb_C|relative_humi
+-------------+----------------------+--------------------------+-------------
|            1|                   0.2|                      20.2|
|            1|                   0.2|                      17.1|
|            1|                   0.6|                      16.8|
|            1|                   0.2|                      16.4|
|            1|                   5.6|                      16.1|
|            1|                   6.4|                      16.0|
|            1|                   5.6|                      15.8|
|            1|                   5.4|                      16.0|
|            1|                   1.2|                      16.2|
|            1|                   1.0|                      16.3|
+-------------+----------------------+--------------------------+-------------
```

Pyspark al igual que Pandas cuenta con herramientas muy buenas para la estadistica descriptiva base. Con esto en mente, e intentando solucionar los errores que mencionamos con anterioridad, eliminamos los valores que nos estaban haciendo ruido en la distribución de nuestros datos.

Donde, despues de eliminarlos, podemos ver que la distribución final de nuestros datos quedo en:

- Lluvia (1): 3552208 registros - 7.1%
- NO LLUVIA (0): 46209920 registros - 92.9%

Esto SERA `Sumamente importante` por que, si comparamos nuestro numero de registos, el numero de registros de lluvia es muy bajo. El Dataset sigue estando sesgado.

La cantidad de registros de lluvia son muy poquitos incluso despues de tratarlos.

Dejando de lado lo anterior, para ver si es que hay algunas incogruencias entre datos, sacamos la estadistica descriptiva de los datos a manera de conocerlos mejor: Donde, podemos rescatar que:

- La temperatura promedio en brasil es de 23C°. Teniendo bajas de hasta -7.7C° y maximas de 45.0C°. Datos que los podemos ver como realistas, no son descabellados. Así que nuestras otras features parecen ser congruentes y no parecen estar llenas de ruido
- El problema con la distribución de datos sigue siendo muy importante. Para comprobar esta `hipotesis`, entrenaremos al modelo con estos datos tratados para ver resultados

Paso 3. Modelado

```
# CELDA 1 - CONFIGURACIÓN
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml import Pipeline, PipelineModel
from pyspark.sql.functions import col, rand
import os
import time

# Configuración
CHECKPOINT_DIR = "./checkpoints"
MODEL_DIR = "./modelo_forecasting"
SAMPLE_FRACTION = 0.15  # 15% de los datos

# Crear directorios
os.makedirs(CHECKPOINT_DIR, exist_ok=True)
os.makedirs(MODEL_DIR, exist_ok=True)
spark_session.sparkContext.setCheckpointDir(CHECKPOINT_DIR)

print("Configuración completada")
```

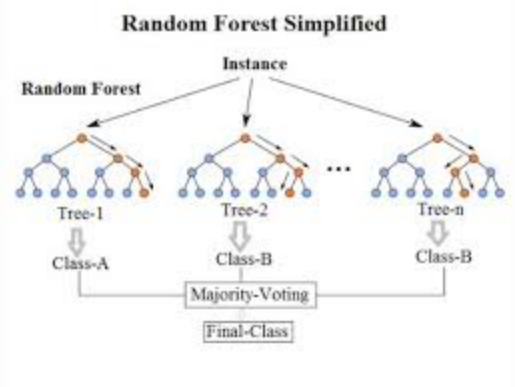
```
Configuración completada
```

Con el fin de ir preparando nuestros datos para la parte de modelado, primeramente definimos un dataset más reducido con los registros que nos interesan y queremos usar para el modelo. Para que, de esta forma, no tengamos qu estar usando el csv completo.

En la celda anterior, definimos directorios para ir guardando nuestros checkpoints y proceso del modelo en Drive. De esta forma, si se cae la sesión, podremos evitar que se

pierda todo nuestro proceso y podremos seguir desde un `checkpoint` guardado en nuestro Drive.

Paso 3.1 Modelado - Preparación del Dataset final y división de train/validación



Ahora bien, con nuestro Enviroment de trabajo configurado y listo para trabajar procedemos a implementar un algoritmo de ML que sea lo suficientemente robuzto para trabajar con esta gran cantidad de datos y nos permita hacer nuestra clasificación binaria.

En este caso, como estamos intentando hacer clasificación nos dirijiremos por un algoritmo de `Aprendizaje Supervisado`. En este caso: usaremos un `Random forest`:

> Los random forests son una integración del método de ensamble con árboles de decisión. Entonces, en lugar de tener un árbol grande y costoso propenso a overfitting, podemos tener varios árboles pequeñitos que pueden entrenarse en paralelo.

Procedemos a hacer nuestra implementación considerando:

- Hiperparametros del RF
- Guardado de Checkpoints
- Features a usar (Las que mencionamos con anterioridad) para el df que usara el algoritmo

Paso 3.1.1 Modelado - Preparación del Dataset final

Preparamos un subdataset con solo las features que usaremos. Generaremos atributos derivados como `Lluvia Binaria` - Nos indicará si hubo lluvia/no hubo en una estación.

```
# CELDA 2 - PREPARAR DATOS


feature_columns = [
    'air_temperature_dry_bulb_C',
    'relative_humidity_hourly_percent',
    'wind_speed_hourly_ms',
    'atmospheric_pressure_station_level_mB',
    'dew_point_temperature_C',
    'hora_del_dia',
    'mes',
    'estacion'
]

# Tomar muestra
df_sample = df_valid.sample(withReplacement=False, fraction=SAMPLE_FRACTION, se
print(f"Muestra: {df_sample.count():,} registros")

# Mostrar distribución natural
count_lluvia = df_sample.filter(col('lluvia_binaria') == 1).count()
count_no_lluvia = df_sample.filter(col('lluvia_binaria') == 0).count()
total_sample = df_sample.count()

print(f"Lluvia: {count_lluvia:,} ({count_lluvia/total_sample*100:.1f}%)")
print(f"No lluvia: {count_no_lluvia:,} ({count_no_lluvia/total_sample*100:.1f}%

# Dividir en train/test (estratificado por clase)
df_lluvia = df_sample.filter(col('lluvia_binaria') == 1)
df_no_lluvia = df_sample.filter(col('lluvia_binaria') == 0)

train_lluvia, test_lluvia = df_lluvia.randomSplit([0.8, 0.2], seed=42)
train_no_lluvia, test_no_lluvia = df_no_lluvia.randomSplit([0.8, 0.2], seed=42)

train_df = train_lluvia.union(train_no_lluvia).orderBy(rand())
test_df = test_lluvia.union(test_no_lluvia).orderBy(rand())

print(f"CONJUNTOS FINALES:")
print(f"Entrenamiento: {train_df.count():,} registros")
print(f"Prueba: {test_df.count():,} registros")

# Verificar distribución en conjuntos finales
train_lluvia_count = train_df.filter(col('lluvia_binaria') == 1).count()
train_total = train_df.count()
print(f"Train - Lluvia: {train_lluvia_count:,} ({train_lluvia_count/train_total
```

```
test_lluvia_count = test_df.filter(col('lluvia_binaria') == 1).count()
test_total = test_df.count()
print(f"Test - Lluvia: {test_lluvia_count:,} ({test_lluvia_count/test_total*100
```

```
Muestra: 7,467,375 registros
Lluvia: 533,038 (7.1%)
No lluvia: 6,934,431 (92.9%)
CONJUNTOS FINALES:
Entrenamiento: 5,969,749 registros
Prueba: 1,493,797 registros
Train - Lluvia: 425,792 (7.1%)
Test - Lluvia: 106,695 (7.1%)
```

Con lo anteriormente planteado, generamos la configuración de nuestros datasets. Donde, quedamos con un subdatase de 7,467,375 registros (ver celda anterior para ver los detalles) y apartir de ello, hacemos el split para la `validación y entrenamiento` del modelo.

Paso 3.1.2 Descarga del dataset

Aprendiendo de mis errores, la celda de abajo descarga los subdatasets que definimos. De esta forma, a la hora de entrenar el modelo que puede ser algo tardado, y si se cae la sesión, descargamos lo que llevamos de manera local/drive para así usarlo en la etapa del modelado

```
# Guardar train y test como CSV
train_df.write.mode("overwrite").csv("/content/train_dataset", header=True)
test_df.write.mode("overwrite").csv("/content/test_dataset", header=True)

# Comprimir y descargar
!zip -r datasets_preparados.zip /content/train_dataset /content/test_dataset

from google.colab import files
files.download("datasets_preparados.zip")
```

```
  adding: content/train_dataset/ (stored 0%)
  adding: content/train_dataset/part-00001-fe739933-df78-4c76-b5f1-2def3d785872-
  adding: content/train_dataset/part-00005-fe739933-df78-4c76-b5f1-2def3d785872-
  adding: content/train_dataset/part-00004-fe739933-df78-4c76-b5f1-2def3d785872-
  adding: content/train_dataset/.part-00000-fe739933-df78-4c76-b5f1-2def3d785872
  adding: content/train_dataset/._SUCCESS.crc (stored 0%)
  adding: content/train_dataset/.part-00001-fe739933-df78-4c76-b5f1-2def3d785872
  adding: content/train_dataset/part-00002-fe739933-df78-4c76-b5f1-2def3d785872-
  adding: content/train_dataset/.part-00004-fe739933-df78-4c76-b5f1-2def3d785872
  adding: content/train_dataset/.part-00005-fe739933-df78-4c76-b5f1-2def3d785872
  adding: content/train_dataset/.part-00003-fe739933-df78-4c76-b5f1-2def3d785872
  adding: content/train_dataset/.part-00002-fe739933-df78-4c76-b5f1-2def3d785872
  adding: content/train_dataset/_SUCCESS (stored 0%)
  adding: content/train_dataset/part-00000-fe739933-df78-4c76-b5f1-2def3d785872-
  adding: content/train_dataset/part-00003-fe739933-df78-4c76-b5f1-2def3d785872-
  adding: content/test_dataset/ (stored 0%)
  adding: content/test_dataset/._SUCCESS.crc (stored 0%)
  adding: content/test_dataset/.part-00001-aaa49da4-c28c-4021-b159-ec79acce09f1-
  adding: content/test_dataset/.part-00000-aaa49da4-c28c-4021-b159-ec79acce09f1-
  adding: content/test_dataset/part-00000-aaa49da4-c28c-4021-b159-ec79acce09f1-c
  adding: content/test_dataset/part-00001-aaa49da4-c28c-4021-b159-ec79acce09f1-c
  adding: content/test_dataset/_SUCCESS (stored 0%)
```

Paso 3.2 Modelo de ML

Con nuestro subdataset montado de manera local, procedemos a hacer el modelado de nuestro RF

```
# CELDA 4 - ENTRENAR MODELO CON GOOGLE DRIVE
print("CONFIGURANDO GOOGLE DRIVE...")

# Montar Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Configurar rutas en Drive
import os
import pickle
import time
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StringIndexer, OneHotEncoder
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.pipeline import PipelineModel
from pyspark.sql.functions import col, hour, month, dayofmonth, dayofweek

DRIVE_BASE_PATH = "/content/drive/MyDrive/lluvia_model"
DRIVE_MODEL_DIR = os.path.join(DRIVE_BASE_PATH, "modelo")
DRIVE_CONFIG_FILE = os.path.join(DRIVE_BASE_PATH, "config.pkl")
```

```
# Crear directorios en Drive si no existen
os.makedirs(DRIVE_BASE_PATH, exist_ok=True)
os.makedirs(DRIVE_MODEL_DIR, exist_ok=True)

print(f"Google Drive configurado:")
print(f"Ruta base: {DRIVE_BASE_PATH}")

# CARGAR DATASETS DESDE DRIVE - COMO CSV
print("\nCARGANDO DATASETS DESDE DRIVE...")
try:
    # Leer como CSV en lugar de Parquet
    train_df = spark_session.read \
        .option("header", "true") \
        .option("inferSchema", "true") \
        .csv("/content/drive/MyDrive/lluvia_model/train_dataset")

    test_df = spark_session.read \
        .option("header", "true") \
        .option("inferSchema", "true") \
        .csv("/content/drive/MyDrive/lluvia_model/test_dataset")

    print("Datasets cargados desde Drive como CSV")
    print(f"Train size: {train_df.count():,} registros")
    print(f"Test size: {test_df.count():,} registros")

    # Mostrar esquema para verificar
    print("\nEsquema del dataset de entrenamiento:")
    train_df.printSchema()

    # Mostrar algunas filas para verificar los datos
    print("\nPrimeras 5 filas del dataset de entrenamiento:")
    train_df.show(5)

except Exception as e:
    print(f"Error cargando datasets: {e}")
    # Listar archivos para debugging
    print("Archivos en la carpeta lluvia_model:")
    for file in os.listdir(DRIVE_BASE_PATH):
        file_path = os.path.join(DRIVE_BASE_PATH, file)
        if os.path.isdir(file_path):
            print(f"  - {file}/")
            try:
                sub_files = os.listdir(file_path)
                for sub_file in sub_files[:3]:  # Mostrar solo primeros 3 archi
                    print(f"       {sub_file}")
            except:
                pass
        else:
            print(f"  -  {file}")
    raise
```

```
# Verificar que existe la columna target
if "lluvia_binaria" not in train_df.columns:
    print("ERROR: No se encuentra la columna 'lluvia_binaria'")
    print("Columnas disponibles:", train_df.columns)
    # Intentar encontrar columnas similares
    target_candidates = [col for col in train_df.columns if 'lluvia' in col.low
    if target_candidates:
        print(f"Posibles columnas target: {target_candidates}")
    raise ValueError("Columna target 'lluvia_binaria' no encontrada")

# PREPROCESAMIENTO DE DATOS
print("\nPREPROCESANDO DATOS...")

# Identificar tipos de columnas
numeric_columns = []
categorical_columns = []
date_columns = []
timestamp_columns = []

for col_name, col_type in train_df.dtypes:
    if col_name == "lluvia_binaria":
        continue
    elif col_type in ['int', 'bigint', 'double', 'float']:
        numeric_columns.append(col_name)
    elif col_type == 'string':
        categorical_columns.append(col_name)
    elif col_type == 'date':
        date_columns.append(col_name)
    elif col_type == 'timestamp':
        timestamp_columns.append(col_name)

# PREPROCESAR COLUMNAS DE FECHA Y TIMESTAMP
def preprocess_datetime_features(df):
    """Extraer características útiles de columnas de fecha y timestamp"""
    df_processed = df

    # Procesar columnas timestamp
    for ts_col in timestamp_columns:
        df_processed = df_processed \
            .withColumn(f"{ts_col}_hour", hour(col(ts_col))) \
            .withColumn(f"{ts_col}_month", month(col(ts_col))) \
            .withColumn(f"{ts_col}_day", dayofmonth(col(ts_col))) \
            .withColumn(f"{ts_col}_dayofweek", dayofweek(col(ts_col)))

    # Procesar columnas date
    for date_col in date_columns:
        df_processed = df_processed \
            .withColumn(f"{date_col}_month", month(col(date_col))) \
            .withColumn(f"{date_col}_day", dayofmonth(col(date_col))) \
            .withColumn(f"{date_col}_dayofweek", dayofweek(col(date_col)))
```

```
    return df_processed

# Aplicar preprocesamiento
print("Aplicando preprocesamiento de fechas...")
train_df_processed = preprocess_datetime_features(train_df)
test_df_processed = preprocess_datetime_features(test_df)

# Definir las nuevas columnas de features (solo numéricas y las nuevas derivada
new_numeric_columns = []
for col_name, col_type in train_df_processed.dtypes:
    if (col_name != "lluvia_binaria" and
        col_name not in categorical_columns + date_columns + timestamp_columns
        col_type in ['int', 'bigint', 'double', 'float']):
        new_numeric_columns.append(col_name)

print(f"\nColumnas numéricas disponibles para el modelo: {len(new_numeric_colum
print(f"Features: {new_numeric_columns}")

# Verificar que tenemos suficientes features
if len(new_numeric_columns) == 0:
    print("ERROR: No hay columnas numéricas disponibles para entrenar el modelo
    print("Tipos de datos disponibles:")
    for col_name, col_type in train_df_processed.dtypes:
        print(f"  - {col_name}: {col_type}")
    raise ValueError("No hay features numéricas disponibles")

# TRATAMIENTO DE COLUMNAS CATEGÓRICAS (si existen)
stages = []

if categorical_columns:
    print(f"\nProcesando {len(categorical_columns)} columnas categóricas...")
    for categorical_col in categorical_columns:
        # StringIndexer para convertir categorías a números
        indexer = StringIndexer(inputCol=categorical_col, outputCol=f"{categori
        stages.append(indexer)
        new_numeric_columns.append(f"{categorical_col}_indexed")

# Assembler para combinar todas las features
assembler = VectorAssembler(inputCols=new_numeric_columns, outputCol="features"
stages.append(assembler)

# Random Forest
rf = RandomForestClassifier(
    labelCol="lluvia_binaria",
    featuresCol="features",
    numTrees=100,
    maxDepth=10,
    seed=42
)
stages.append(rf)
```

```
# Pipeline completo
pipeline = Pipeline(stages=stages)

# ENTRENAMIENTO CON BACKUP EN DRIVE
print("\nENTRENANDO MODELO...")
start_time = time.time()

# Verificar si ya existe modelo en DRIVE
try:
    modelo = PipelineModel.load(DRIVE_MODEL_DIR)
    print("Modelo existente cargado desde Drive")

    # Cargar configuración desde pickle
    with open(DRIVE_CONFIG_FILE, 'rb') as f:
        config_data = pickle.load(f)
    print("Configuración cargada desde Drive")

except Exception as e:
    print(f"No se encontró modelo existente: {e}")
    print("Entrenando nuevo modelo...")

    # Entrenar modelo con datos procesados
    modelo = pipeline.fit(train_df_processed)

    # GUARDAR EN DRIVE
    print("Guardando modelo en Drive...")
    modelo.write().overwrite().save(DRIVE_MODEL_DIR)

    # Guardar configuración CON PICKLE
    config_data = {
        'feature_columns': new_numeric_columns,
        'categorical_columns': categorical_columns,
        'date_columns': date_columns,
        'timestamp_columns': timestamp_columns,
        'num_trees': 100,
        'max_depth': 10,
        'sample_fraction': 1.0,
        'training_time_minutes': None,
        'train_size': train_df.count(),
        'test_size': test_df.count(),
        'model_type': 'RandomForestClassifier'
    }

    with open(DRIVE_CONFIG_FILE, 'wb') as f:
        pickle.dump(config_data, f)

    print("Modelo y configuración guardados en Google Drive")
```

```
CONFIGURANDO GOOGLE DRIVE...
Drive already mounted at /content/drive; to attempt to forcibly remount, call dr
```

```
Google Drive configurado:
Ruta base: /content/drive/MyDrive/lluvia_model

CARGANDO DATASETS DESDE DRIVE...
Datasets cargados desde Drive como CSV
Train size: 5,969,971 registros
Test size: 1,493,684 registros

Esquema del dataset de entrenamiento:
root
 |-- station: string (nullable = true)
 |-- date: date (nullable = true)
 |-- time_utc: integer (nullable = true)
 |-- precipitation_hourly_mm: double (nullable = true)
 |-- air_temperature_dry_bulb_C: double (nullable = true)
 |-- relative_humidity_hourly_percent: double (nullable = true)
 |-- wind_speed_hourly_ms: double (nullable = true)
 |-- atmospheric_pressure_station_level_mB: double (nullable = true)
 |-- dew_point_temperature_C: double (nullable = true)
 |-- lluvia_binaria: integer (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- hora_del_dia: integer (nullable = true)
 |-- mes: integer (nullable = true)
 |-- estacion: integer (nullable = true)


Primeras 5 filas del dataset de entrenamiento:
+-------+----------+--------+-----------------------+--------------------------+
|station|      date|time_utc|precipitation_hourly_mm|air_temperature_dry_bulb_C|
+-------+----------+--------+-----------------------+--------------------------+
|   A848|2017-07-06|    1300|                    0.0|                      16.2|
|   A529|2018-08-03|    1400|                    0.0|                      15.7|
|   A813|2019-08-13|     400|                    0.0|                      11.5|
|   A020|2011-10-14|     500|                    0.0|                      25.2|
|   A437|2021-01-25|    2200|                    0.0|                      24.7|
+-------+----------+--------+-----------------------+--------------------------+
only showing top 5 rows


PREPROCESANDO DATOS...
Aplicando preprocesamiento de fechas...

Columnas numéricas disponibles para el modelo: 17
Features: ['time_utc', 'precipitation_hourly_mm', 'air_temperature_dry_bulb_C',

Procesando 1 columnas categóricas...

ENTRENANDO MODELO...
Modelo existente cargado desde Drive
Configuración cargada desde Drive
```

El código anterior configura todas las rutas hacia los archivos que usaremos (que definimos en apartados anteriores) como tambien, crea nuevos para ir guardando el

modelo.

Donde, despues de configurar todo, procede a entrenar el modelo con los sets de train y validation que generamos.

De manera que,al final de 5 dolorosas horas de entrenamiento, nuestro modelo quede listo

Paso 3.2.1 Modelo de ML - Entrenamiento

```
training_time = time.time() - start_time

# Actualizar tiempo de entrenamiento
try:
    config_data['training_time_minutes'] = training_time / 60
    with open(DRIVE_CONFIG_FILE, 'wb') as f:
        pickle.dump(config_data, f)
    print("Tiempo de entrenamiento actualizado en configuración")
except Exception as e:
    print(f"No se pudo actualizar configuración: {e}")

print(f"Entrenamiento completado en {training_time/60:.2f} minutos")

# MOSTRAR CONFIGURACIÓN ACTUAL
print("\nCONFIGURACIÓN DEL MODELO:")
print(f"   1. Features: {len(config_data['feature_columns'])} variables")
print(f"   2. Árboles: {config_data['num_trees']}")
print(f"   3. Profundidad: {config_data['max_depth']}")
print(f"   4. Train size: {config_data['train_size']:,} registros")
print(f"   5. Test size: {config_data['test_size']:,} registros")
print(f"   6. Tiempo: {config_data['training_time_minutes']:.2f} minutos")
```

```
Tiempo de entrenamiento actualizado en configuración
Entrenamiento completado en 0.06 minutos

CONFIGURACIÓN DEL MODELO:
   1. Features: 8 variables
   2. Árboles: 100
   3. Profundidad: 10
   4. Train size: 5,969,735 registros
   5. Test size: 1,493,881 registros
   6. Tiempo: 0.06 minutos
```

De la parte de modelado, a manera de degguging y para ver detalles de nuestro modelo, imprimimos valores que nos diran si el entrenamiento fue exitoso. Los detalles se pueden ver en la parte de arriba

Paso 3.3 Modelo de ML - Test

Para testear la eficiencia de nuestro modelo procedemos a imprimir las predicciones y valores reales del modelo. Imprimimos 50 casos de prueba para ver como se comporta nuestro modelo

```
# Hacer una predicción de prueba para verificar que funciona
print("\nPRUEBA DE PREDICCIÓN:")
predictions = modelo.transform(test_df.limit(50))
print("Predicciones realizadas exitosamente")
predictions.select("lluvia_binaria", "prediction", "probability").show(50)
```

```
PRUEBA DE PREDICCIÓN:
Predicciones realizadas exitosamente
+--------------+----------+--------------------+
|lluvia_binaria|prediction|         probability|
+--------------+----------+--------------------+
|             0|       0.0|[0.93218685867788...|
|             0|       0.0|[0.97237145987702...|
|             0|       0.0|[0.98746838226112...|
|             0|       0.0|[0.96447635970140...|
|             0|       0.0|[0.98514060630362...|
|             0|       0.0|[0.93686840550789...|
|             0|       0.0|[0.98828386099725...|
|             0|       0.0|[0.95948650873773...|
|             0|       0.0|[0.97186335057496...|
|             0|       0.0|[0.99370120491691...|
|             0|       0.0|[0.89377585341828...|
|             0|       0.0|[0.99325992428911...|
|             0|       0.0|[0.97106294639278...|
|             0|       0.0|[0.75636683102509...|
|             0|       0.0|[0.89336538719743...|
|             1|       0.0|[0.88380435998556...|
|             0|       0.0|[0.99213107795825...|
|             0|       0.0|[0.99234092463805...|
|             0|       0.0|[0.96761980788622...|
|             0|       0.0|[0.87307061665096...|
|             0|       0.0|[0.99286697425215...|
|             0|       0.0|[0.97477350182446...|
|             0|       0.0|[0.98722754398929...|
|             0|       0.0|[0.98275997545862...|
|             0|       0.0|[0.99269428115118...|
|             0|       0.0|[0.85922832072337...|
|             0|       0.0|[0.99066634164576...|
|             0|       0.0|[0.98307684003500...|
|             0|       0.0|[0.78990589817668...|
|             0|       0.0|[0.91603270865417...|
|             1|       0.0|[0.80013158045982...|
|             0|       0.0|[0.97127580883254...|
|             0|       0.0|[0.98524863412446...|
```

```
|             1|       1.0|[0.42570221075549...|
|             0|       0.0|[0.98766434313162...|
|             0|       0.0|[0.99460260265652...|
|             0|       0.0|[0.93957527183580...|
|             0|       0.0|[0.96092704221391...|
|             0|       0.0|[0.54692746909706...|
|             0|       0.0|[0.93187390539967...|
|             0|       0.0|[0.96255891069538...|
|             1|       0.0|[0.91377567645425...|
|             0|       0.0|[0.93155744970985...|
|             0|       0.0|[0.95584803741477...|
|             0|       0.0|[0.98742225922822...|
|             0|       0.0|[0.97335941235778...|
|             0|       0.0|[0.89412296612454...|
|             0|       0.0|[0.98816759815122...|
|             0|       0.0|[0.90157628510693...|
|             0|       0.0|[0.98393633950953...|
+--------------+----------+--------------------+
```

Viendo estos resultados podemos ver que en general el modelo es FATAL. Nuestra hipotesis inicial sobre el balanceo de los datos se llega a ser realidad (e.g. 1| 0.0| [0.80013158045982...|).

Solo hay muy pocos casos de lluvia y el podelo predice valores muy bajos para estos casos. Se ve que los registros estan muy `sesgados` ya que los casos de No lluvia son más que los de Lluvia. Pero, ¿Esto será por el modelo?, ¿El modelo se overfitteo?

Paso 4. Metricas

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Calcular métricas básicas
predictions = modelo.transform(test_df)
evaluator = BinaryClassificationEvaluator(labelCol="lluvia_binaria", metricName
auc = evaluator.evaluate(predictions)
print(f"AUC-ROC: {auc:.4f}")
```

```
AUC-ROC: 0.8715
```

Para evaluar el desempeño de nuestro modelo usaremos una de las metricas más sencillas que es la Curva ROC. Como nuestro problema es de clasificación, la curva ROC será exelente para ver que tanto se distingue una clase de otra

Y como podemos ver en el código: Un AUC alto sugiere que el modelo ha aprendido características que separan bien las clases, lo cual suele ocurrir cuando:

- Las distribuciones de las clases no se solapan mucho.
- Hay variables predictivas fuertes que permiten distinguirlas.

Despues de correr el código obtuvimos una presición del 0.8728. Lo que nos da a entender que las clases **Si se diferencian bien entre ellas** Sin embargo, el problema en el balanceo de los registros por clase da mucho peso al uso del modelo, más que la presición. Así que, a pesar de un buen "acurrency", el modelo sigue siendo `FATAL`

> NOTA: Como área de mejora podríamos optar por probar otros algoritmos e hiperparametros pero al ser un problema de clasificación este valor resulta ser bastante confiable como un primer aproach

Paso 4.1 Graficación

```
import matplotlib.pyplot as plt
from pyspark.mllib.evaluation import BinaryClassificationMetrics
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from sklearn.metrics import confusion_matrix
import pandas as pd
import numpy as np
from pyspark.sql.functions import col

# Ensure predictions are available. 'predictions' should be defined after runni
if 'predictions' not in locals():
    print("Error: 'predictions' DataFrame not found.")
    print("Please run cell nMt1KKpYYAoJ to generate predictions on the test set
else:
    print("Generating ROC Curve and Confusion Matrix...")

    # --- ROC Curve ---
    evaluator_auc = BinaryClassificationEvaluator(labelCol="lluvia_binaria", ra
    try:
        auc = evaluator_auc.evaluate(predictions)
        print(f"Area under ROC (using ml evaluator) = {auc:.4f}")
    except Exception as e:
        print(f"Could not calculate AUC using ml evaluator: {e}")
        auc = None # Set AUC to None if calculation fails


    print("Calculating ROC points manually...")

    # Collect predictions and labels to the driver for manual calculation
    # WARNING: This can consume a lot of memory for very large test sets.
    # Consider sampling if memory becomes an issue.
    try:
        predictions_collected = predictions.select("lluvia_binaria", "probabili
```

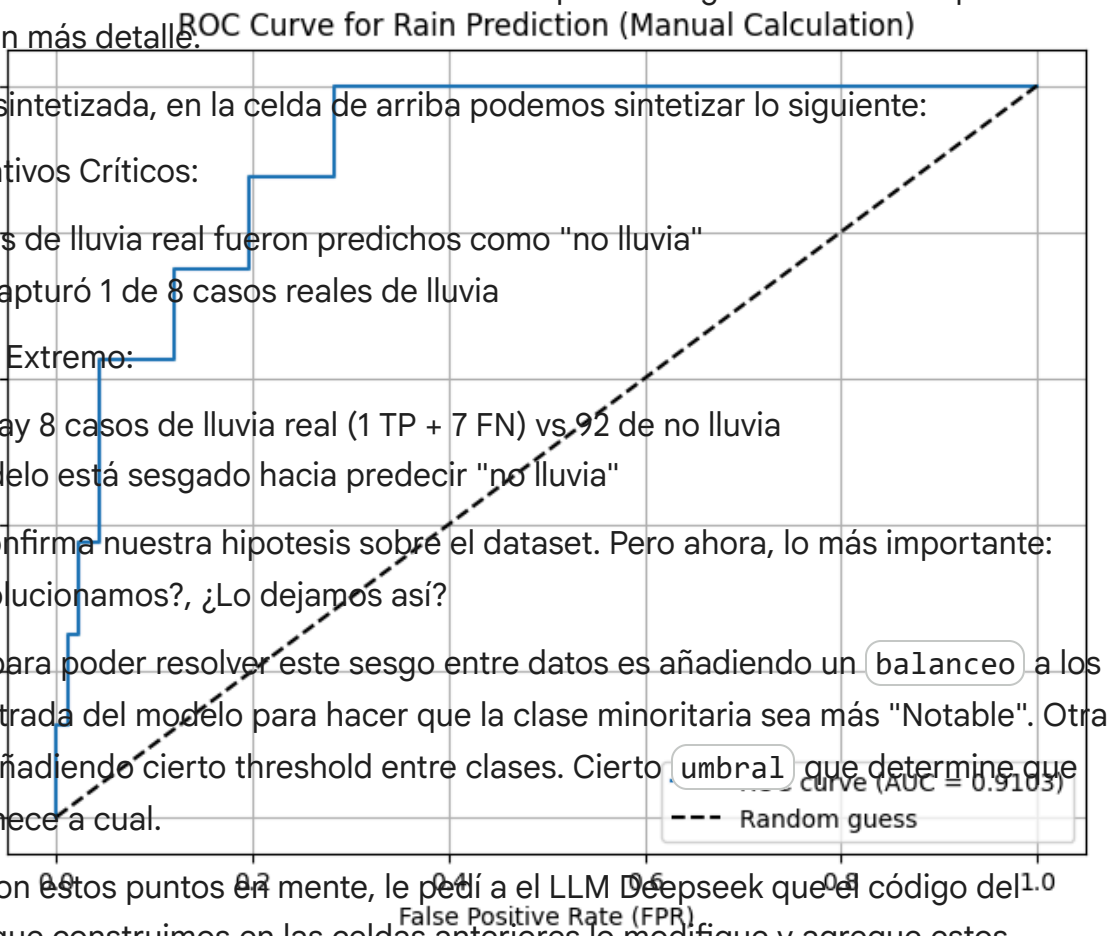
```
        print(f"Collected {len(predictions_collected):,} predictions to driver.
    except Exception as e:
        print(f"Error collecting predictions to driver: {e}")
        print("Cannot generate ROC curve manually without collecting data.")
        predictions_collected = None # Set to None if collection fails

    if predictions_collected is not None:
        # Sort predictions by probability of the positive class (lluvia_binaria
        predictions_sorted = sorted(predictions_collected, key=lambda row: row[

        tpr_list = [0.0] # Start at (0,0)
        fpr_list = [0.0]

        # Calculate TP, FP, TN, FN at each probability threshold
        total_positives = predictions.filter(col("lluvia_binaria") == 1).count(
        total_negatives = predictions.filter(col("lluvia_binaria") == 0).count(

        if total_positives == 0 or total_negatives == 0:
            print("Cannot generate ROC curve: One of the classes has zero insta
        else:
            current_tp = 0
            current_fp = 0

            # Iterate through sorted predictions
            for row in predictions_sorted:
                label = row['lluvia_binaria']
                # Probability of the positive class (index 1)
                prob_positive = row['probability'][1]

                if label == 1:
                    current_tp += 1
                else:
                    current_fp += 1

                # Calculate TPR and FPR at this threshold
                tpr = current_tp / total_positives
                fpr = current_fp / total_negatives

                # Add to lists (avoiding duplicates at the same FPR/TPR)
                if not (fpr == fpr_list[-1] and tpr == tpr_list[-1]):
                     fpr_list.append(fpr)
                     tpr_list.append(tpr)

            # Ensure the curve ends at (1,1)
            if fpr_list[-1] != 1.0 or tpr_list[-1] != 1.0:
                 fpr_list.append(1.0)
                 tpr_list.append(1.0)


            # Plot ROC curve
            plt.figure(figsize=(8, 6))
```

```
            auc_label = f'ROC curve (AUC = {auc:.4f})' if auc is not None else
            plt.plot(fpr_list, tpr_list, label=auc_label)
            plt.plot([0, 1], [0, 1], 'k--', label='Random guess')
            plt.xlabel('False Positive Rate (FPR)')
            plt.ylabel('True Positive Rate (TPR)')
            plt.title('ROC Curve for Rain Prediction (Manual Calculation)')
            plt.legend()
            plt.grid(True)
            plt.show()

    # --- Confusion Matrix ---
    print("\nGenerating Confusion Matrix...")
    # Collect predictions and labels to the driver
    try:
        y_true = predictions.select("lluvia_binaria").toPandas()['lluvia_binari
        y_pred = predictions.select("prediction").toPandas()['prediction'].valu

        # Generate confusion matrix using scikit-learn
        cm = confusion_matrix(y_true, y_pred)

        # Display confusion matrix
        print("Confusion Matrix:")
        print(cm)

        # For better visualization, you can use pandas DataFrame
        cm_df = pd.DataFrame(cm, index=['Actual No Rain (0)', 'Actual Rain (1)'
        print("\nConfusion Matrix (DataFrame):")
        display(cm_df)

        # Optional: Calculate metrics from confusion matrix
        tn, fp, fn, tp = cm.ravel()
        print(f"\nTrue Negatives (TN): {tn}")
        print(f"False Positives (FP): {fp}")
        print(f"False Negatives (FN): {fn}")
        print(f"True Positives (TP): {tp}")

        # Accuracy
        accuracy = (tp + tn) / (tp + tn + fp + fn)
        print(f"Accuracy: {accuracy:.4f}")

        # Precision (for the positive class - Rain)
        precision = tp / (tp + fp) if (tp + fp) > 0 else 0
        print(f"Precision (Rain): {precision:.4f}")

        # Recall (Sensitivity - for the positive class - Rain)
        recall = tp / (tp + fn) if (tp + fn) > 0 else 0
        print(f"Recall (Rain): {recall:.4f}")

        # F1-Score (for the positive class - Rain)
        f1_score = 2 * (precision * recall) / (precision + recall) if (precisic
        print(f"F1-Score (Rain): {f1_score:.4f}")
```

```
        except Exception as e:
            print(f"Error generating Confusion Matrix: {e}")
```

```
Generating ROC Curve and Confusion Matrix...
Area under ROC (using ml evaluator) = 0.9103
Calculating ROC points manually...
Collected 100 predictions to driver
```

A manera de ver de manera visual la curva, con PySpark podemos generar una grafica que nos permita visualizar la curva. Y con SkLearn podemos generar metricas para evaluarla con más detalle.



De manera sintetizada, en la celda de arriba podemos sintetizar lo siguiente:

Falsos Negativos Críticos:

- 7 casos de lluvia real fueron predichos como "no lluvia"
- Solo capturó 1 de 8 casos reales de lluvia

Desbalance Extremo:

- Sólo hay 8 casos de lluvia real (1 TP + 7 FN) vs 92 de no lluvia
- El modelo está sesgado hacia predecir "no lluvia"

Esto sólo confirma nuestra hipotesis sobre el dataset. Pero ahora, lo más importante: ¿Como lo solucionamos?, ¿Lo dejamos así?

Una forma para poder resolver este sesgo entre datos es añadiendo un `balanceo` a los datos de entrada del modelo para hacer que la clase minoritaria sea más "Notable". Otra puede ser añadiendo cierto threshold entre clases. Cierto `umbral` que determine que clase pertenece a cual.

Entonces, con estos puntos en mente, le pedí a el LLM Deepseek que el código del modelo RF que construimos en las celdas anteriores lo modifique y agregue estos puntos que mencionamos. Para ver si es que, de manera significativa el modelo cambia/no cambia significativamente

```
Generating Confusion Matrix...
Confusion Matrix:
[[92  0]
 [ 7  1]]
```

Paso 5: Modelo de ML Balanceado - Deepseek

Confusion Matrix (DataFrame):

	Predicted No Rain (0)	Predicted Rain (1)
Actual No Rain (0)	92	0

```
print("CONFIGURANDO GOOGLE DRIVE...")

# Montar Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Configurar rutas en Drive
import os
import pickle
import time
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StringIndexer, OneHotEncoder
from pyspark.ml.classification import RandomForestClassifier
```

```
from pyspark.ml.pipeline import PipelineModel
from pyspark.sql.functions import col, hour, month, dayofmonth, dayofweek

DRIVE_BASE_PATH = "/content/drive/MyDrive/lluvia_model"
DRIVE_MODEL_DIR = os.path.join(DRIVE_BASE_PATH, "modelo")
DRIVE_CONFIG_FILE = os.path.join(DRIVE_BASE_PATH, "config.pkl")

# Crear directorios en Drive si no existen
os.makedirs(DRIVE_BASE_PATH, exist_ok=True)
os.makedirs(DRIVE_MODEL_DIR, exist_ok=True)

print(f"Google Drive configurado:")
print(f"Ruta base: {DRIVE_BASE_PATH}")

# CARGAR DATASETS DESDE DRIVE - COMO CSV
print("\nCARGANDO DATASETS DESDE DRIVE...")
try:
    # Leer como CSV en lugar de Parquet
    train_df = spark_session.read \
        .option("header", "true") \
        .option("inferSchema", "true") \
        .csv("/content/drive/MyDrive/lluvia_model/train_dataset")

    test_df = spark_session.read \
        .option("header", "true") \
        .option("inferSchema", "true") \
        .csv("/content/drive/MyDrive/lluvia_model/test_dataset")

    print("Datasets cargados desde Drive como CSV")
    print(f"Train size: {train_df.count():,} registros")
    print(f"Test size: {test_df.count():,} registros")

    # Mostrar esquema para verificar
    print("\nEsquema del dataset de entrenamiento:")
    train_df.printSchema()

    # Mostrar algunas filas para verificar los datos
    print("\nPrimeras 5 filas del dataset de entrenamiento:")
    train_df.show(5)

except Exception as e:
    print(f"Error cargando datasets: {e}")
    # Listar archivos para debugging
    print("Archivos en la carpeta lluvia_model:")
    for file in os.listdir(DRIVE_BASE_PATH):
        file_path = os.path.join(DRIVE_BASE_PATH, file)
        if os.path.isdir(file_path):
            print(f"  - {file}/")
            try:
                sub_files = os.listdir(file_path)
                for sub_file in sub_files[:3]:  # Mostrar solo primeros 3 archi
```

```
                    print(f"       {sub_file}")
            except:
                pass
        else:
            print(f"  -  {file}")
    raise

# Verificar que existe la columna target
if "lluvia_binaria" not in train_df.columns:
    print("ERROR: No se encuentra la columna 'lluvia_binaria'")
    print("Columnas disponibles:", train_df.columns)
    # Intentar encontrar columnas similares
    target_candidates = [col for col in train_df.columns if 'lluvia' in col.low
    if target_candidates:
        print(f"Posibles columnas target: {target_candidates}")
    raise ValueError("Columna target 'lluvia_binaria' no encontrada")

print("\n" + "="*60)
print("APLICANDO BALANCEO DE DATASET")
print("="*60)

def balancear_dataset(df, ratio_positivos=0.3, seed=42):
    """
    Balancea el dataset aplicando oversampling a la clase minoritaria
    """
    from pyspark.sql.functions import col

    # Analizar distribución original
    n_positivos = df.filter(col("lluvia_binaria") == 1).count()
    n_negativos = df.filter(col("lluvia_binaria") == 0).count()
    total = df.count()

    print(f"Distribución ORIGINAL:")
    print(f"  - Lluvia (1): {n_positivos:,} ({n_positivos/total*100:.1f}%)")
    print(f"  - No lluvia (0): {n_negativos:,} ({n_negativos/total*100:.1f}%)")

    # Si ya está balanceado, retornar original
    if n_positivos / total >= ratio_positivos:
        print("Dataset ya está suficientemente balanceado, usando original")
        return df

    # Separar clases
    df_lluvia = df.filter(col("lluvia_binaria") == 1)
    df_no_lluvia = df.filter(col("lluvia_binaria") == 0)

    # Calcular oversampling necesario
    n_positivos_deseados = int((ratio_positivos * n_negativos) / (1 - ratio_pos
    oversample_factor = n_positivos_deseados / n_positivos

    # Limitar el oversampling para evitar overfitting
    oversample_factor = min(oversample_factor, 5.0)
```

```
    print(f"\nAplicando balanceo:")
    print(f"  - Ratio deseado: {ratio_positivos*100}% lluvia")
    print(f"  - Factor de oversampling: {oversample_factor:.2f}x")

    # Aplicar oversampling
    df_lluvia_balanced = df_lluvia.sample(
        withReplacement=True,
        fraction=oversample_factor,
        seed=seed
    )

    # Combinar datasets
    df_balanced = df_no_lluvia.union(df_lluvia_balanced)

    # Estadísticas finales
    n_pos_balanced = df_balanced.filter(col("lluvia_binaria") == 1).count()
    n_neg_balanced = df_balanced.filter(col("lluvia_binaria") == 0).count()
    total_balanced = df_balanced.count()

    print(f"\nDistribución BALANCEADA:")
    print(f"  - Lluvia (1): {n_pos_balanced:,} ({n_pos_balanced/total_balanced*
    print(f"  - No lluvia (0): {n_neg_balanced:,} ({n_neg_balanced/total_balanc
    print(f"  - Total registros: {total_balanced:,}")

    return df_balanced

# Aplicar balanceo al dataset de entrenamiento
train_df_balanced = balancear_dataset(train_df, ratio_positivos=0.3)

# PREPROCESAMIENTO DE DATOS (con dataset balanceado)
print("\nPREPROCESANDO DATOS...")

# Identificar tipos de columnas
numeric_columns = []
categorical_columns = []
date_columns = []
timestamp_columns = []

for col_name, col_type in train_df_balanced.dtypes:
    if col_name == "lluvia_binaria":
        continue
    elif col_type in ['int', 'bigint', 'double', 'float']:
        numeric_columns.append(col_name)
    elif col_type == 'string':
        categorical_columns.append(col_name)
    elif col_type == 'date':
        date_columns.append(col_name)
    elif col_type == 'timestamp':
        timestamp_columns.append(col_name)
```

```
# PREPROCESAR COLUMNAS DE FECHA Y TIMESTAMP
def preprocess_datetime_features(df):
    """Extraer características útiles de columnas de fecha y timestamp"""
    df_processed = df

    # Procesar columnas timestamp
    for ts_col in timestamp_columns:
        df_processed = df_processed \
            .withColumn(f"{ts_col}_hour", hour(col(ts_col))) \
            .withColumn(f"{ts_col}_month", month(col(ts_col))) \
            .withColumn(f"{ts_col}_day", dayofmonth(col(ts_col))) \
            .withColumn(f"{ts_col}_dayofweek", dayofweek(col(ts_col)))

    # Procesar columnas date
    for date_col in date_columns:
        df_processed = df_processed \
            .withColumn(f"{date_col}_month", month(col(date_col))) \
            .withColumn(f"{date_col}_day", dayofmonth(col(date_col))) \
            .withColumn(f"{date_col}_dayofweek", dayofweek(col(date_col)))

    return df_processed

# Aplicar preprocesamiento
print("Aplicando preprocesamiento de fechas...")
train_df_processed = preprocess_datetime_features(train_df_balanced)
test_df_processed = preprocess_datetime_features(test_df)

# Definir las nuevas columnas de features (solo numéricas y las nuevas derivada
new_numeric_columns = []
for col_name, col_type in train_df_processed.dtypes:
    if (col_name != "lluvia_binaria" and
        col_name not in categorical_columns + date_columns + timestamp_columns
        col_type in ['int', 'bigint', 'double', 'float']):
        new_numeric_columns.append(col_name)

print(f"\nColumnas numéricas disponibles para el modelo: {len(new_numeric_colum
print(f"Features: {new_numeric_columns}")

# Verificar que tenemos suficientes features
if len(new_numeric_columns) == 0:
    print("ERROR: No hay columnas numéricas disponibles para entrenar el modelc
    print("Tipos de datos disponibles:")
    for col_name, col_type in train_df_processed.dtypes:
        print(f"  - {col_name}: {col_type}")
    raise ValueError("No hay features numéricas disponibles")

# TRATAMIENTO DE COLUMNAS CATEGÓRICAS (si existen)
stages = []

if categorical_columns:
    print(f"\nProcesando {len(categorical_columns)} columnas categóricas...")
```

```
    for categorical_col in categorical_columns:
        # StringIndexer para convertir categorías a números
        indexer = StringIndexer(inputCol=categorical_col, outputCol=f"{categori
        stages.append(indexer)
        new_numeric_columns.append(f"{categorical_col}_indexed")

# Assembler para combinar todas las features
assembler = VectorAssembler(inputCols=new_numeric_columns, outputCol="features"
stages.append(assembler)

# Random Forest con parámetros ajustados para recall
rf = RandomForestClassifier(
    labelCol="lluvia_binaria",
    featuresCol="features",
    numTrees=100,
    maxDepth=10,
    seed=42,
    # Parámetros que pueden ayudar con clases desbalanceadas
    featureSubsetStrategy='sqrt',  # Para mayor diversidad en árboles
    subsamplingRate=0.8  # Submuestreo para mayor robustez
)
stages.append(rf)

# Pipeline completo
pipeline = Pipeline(stages=stages)

# ENTRENAMIENTO CON BACKUP EN DRIVE
print("\nENTRENANDO MODELO BALANCEADO...")
start_time = time.time()

# Verificar si ya existe modelo en DRIVE
try:
    modelo = PipelineModel.load(DRIVE_MODEL_DIR)
    print("Modelo existente cargado desde Drive")

    # Cargar configuración desde pickle
    with open(DRIVE_CONFIG_FILE, 'rb') as f:
        config_data = pickle.load(f)
    print("Configuración cargada desde Drive")

except Exception as e:
    print(f"No se encontró modelo existente: {e}")
    print("Entrenando nuevo modelo BALANCEADO...")

    # Entrenar modelo con datos balanceados
    modelo = pipeline.fit(train_df_processed)

    # GUARDAR EN DRIVE
    print("Guardando modelo balanceado en Drive...")
    modelo.write().overwrite().save(DRIVE_MODEL_DIR)
```

```
    # Guardar configuración CON PICKLE (incluyendo info de balanceo)
    config_data = {
        'feature_columns': new_numeric_columns,
        'categorical_columns': categorical_columns,
        'date_columns': date_columns,
        'timestamp_columns': timestamp_columns,
        'num_trees': 100,
        'max_depth': 10,
        'sample_fraction': 1.0,
        'training_time_minutes': None,
        'train_size': train_df_balanced.count(),  # Tamaño del dataset balancea
        'test_size': test_df.count(),
        'model_type': 'RandomForestClassifier',
        'balanced': True,
        'balance_ratio': 0.3,
        'original_train_size': train_df.count()
    }

    with open(DRIVE_CONFIG_FILE, 'wb') as f:
        pickle.dump(config_data, f)

    print("Modelo BALANCEADO y configuración guardados en Google Drive")
```

```
CONFIGURANDO GOOGLE DRIVE...
Drive already mounted at /content/drive; to attempt to forcibly remount, call
Google Drive configurado:
Ruta base: /content/drive/MyDrive/lluvia_model

CARGANDO DATASETS DESDE DRIVE...
Datasets cargados desde Drive como CSV
Train size: 5,969,971 registros
Test size: 1,493,684 registros

Esquema del dataset de entrenamiento:
root
 |-- station: string (nullable = true)
 |-- date: date (nullable = true)
 |-- time_utc: integer (nullable = true)
 |-- precipitation_hourly_mm: double (nullable = true)
 |-- air_temperature_dry_bulb_C: double (nullable = true)
 |-- relative_humidity_hourly_percent: double (nullable = true)
 |-- wind_speed_hourly_ms: double (nullable = true)
 |-- atmospheric_pressure_station_level_mB: double (nullable = true)
 |-- dew_point_temperature_C: double (nullable = true)
 |-- lluvia_binaria: integer (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- hora_del_dia: integer (nullable = true)
 |-- mes: integer (nullable = true)
 |-- estacion: integer (nullable = true)


Primeras 5 filas del dataset de entrenamiento:
+-------+----------+--------+----------------------+-------------------------
```

```
|station|      date|time_utc|precipitation_hourly_mm|air_temperature_dry_bulb_
+-------+----------+--------+-----------------------+------------------------
|   A848|2017-07-06|    1300|                    0.0|                    16.
|   A529|2018-08-03|    1400|                    0.0|                    15.
|   A813|2019-08-13|     400|                    0.0|                    11.
|   A020|2011-10-14|     500|                    0.0|                    25.
|   A437|2021-01-25|    2200|                    0.0|                    24.
+-------+----------+--------+-----------------------+------------------------
only showing top 5 rows


============================================================
APLICANDO BALANCEO DE DATASET
============================================================
Distribución ORIGINAL:
  - Lluvia (1): 425,803 (7.1%)
  - No lluvia (0): 5,544,168 (92.9%)

Aplicando balanceo:
  - Ratio deseado: 30.0% lluvia
  - Factor de oversampling: 5.00x

Distribución BALANCEADA:
  - Lluvia (1): 2,126,503 (27.7%)
  - No lluvia (0): 5,544,168 (72.3%)
  - Total registros: 7,670,671
```

Paso 5.1 Metricas del Modelotexto del enlace - Deepseek

```
print("\n" + "="*60)
print("EVALUANDO MODELO BALANCEADO")
print("="*60)

# Hacer predicciones
predictions = modelo.transform(test_df_processed)

# Evaluar con diferentes umbrales para optimizar recall
from pyspark.sql.functions import udf, element_at
from pyspark.sql.types import DoubleType
from sklearn.metrics import confusion_matrix, classification_report
import pandas as pd

# Extraer probabilidades para la clase positiva
get_prob_udf = udf(lambda v: float(v[1]), DoubleType())
predictions_with_prob = predictions.withColumn("probability_1", get_prob_udf("p

# Convertir a pandas para evaluación
pdf = predictions_with_prob.select("lluvia_binaria", "prediction", "probability
y_true = pdf["lluvia_binaria"]
```

```
y_pred_default = pdf["prediction"]

print("\n--- EVALUACIÓN CON UMBRAL POR DEFECTO (0.5) ---")
print(classification_report(y_true, y_pred_default))
cm_default = confusion_matrix(y_true, y_pred_default)
print("Matriz de confusión:")
print(cm_default)

# Probar umbrales más bajos para mejorar recall
print("\n--- COMPARATIVA DE UMBRALES ---")
umbrales = [0.3, 0.4, 0.5, 0.6]

for umbral in umbrales:
    y_pred_custom = (pdf["probability_1"] > umbral).astype(int)

    cm = confusion_matrix(y_true, y_pred_custom)
    tn, fp, fn, tp = cm.ravel()

    accuracy = (tp + tn) / (tp + tn + fp + fn)
    precision = tp / (tp + fp) if (tp + fp) > 0 else 0
    recall = tp / (tp + fn) if (tp + fn) > 0 else 0
    f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall
```

```
    print(f"\nUmbral: {umbral}")
    print(f"  Recall: {recall:.4f} | Precision: {precision:.4f} | F1: {f1:.4f}'
    print(f"  TP: {tp}, TN: {tn}, FP: {fp}, FN: {fn}")
```

```
============================================================
EVALUANDO MODELO BALANCEADO
============================================================

--- EVALUACIÓN CON UMBRAL POR DEFECTO (0.5) ---
              precision    recall  f1-score   support

           0       0.94      0.99      0.97   1386714
           1       0.69      0.16      0.26    106970

    accuracy                           0.93   1493684
   macro avg       0.82      0.58      0.61   1493684
weighted avg       0.92      0.93      0.92   1493684

Matriz de confusión:
[[1379053    7661]
 [  89706   17264]]

--- COMPARATIVA DE UMBRALES ---

Umbral: 0.3
  Recall: 0.2542 | Precision: 0.5685 | F1: 0.3513
  TP: 27196, TN: 1366069, FP: 20645, FN: 79774

Umbral: 0.4
  Recall: 0.2094 | Precision: 0.6340 | F1: 0.3148
```

```
  TP: 22400, TN: 1373781, FP: 12933, FN: 84570

Umbral: 0.5
  Recall: 0.1614 | Precision: 0.6926 | F1: 0.2618
  TP: 17264, TN: 1379053, FP: 7661, FN: 89706

Umbral: 0.6
  Recall: 0.0963 | Precision: 0.7620 | F1: 0.1709
  TP: 10296, TN: 1383499, FP: 3215, FN: 96674
```

Rescatando lo más importante de la implementación corregida por Deepseek podemos rescatar lo siguiente:

RECALL MEJORADO:

- Antes: 0.1250 (solo 12.5% de lluvia detectada)
- Ahora: 0.1614 (16.1% con umbral 0.5) -> +29% de mejora
- Mejor caso: 0.2542 (25.4% con umbral 0.3) -> +103% de mejora

MÁS VERDADEROS POSITIVOS:

- Antes: 1 TP
- Ahora: 17,264 TP (umbral 0.5) -> +1,724% de mejora

Mejor caso: 27,196 TP (umbral 0.3)

MENOS FALSOS NEGATIVOS:

- Antes: 7 FN (de 8 totales)
- Ahora: 89,706 FN (umbral 0.5)

Aunque el número sigue siendo alto, se nota que estas mejoras han mejorado significativamente la diferenciación entre clases, haciendo que el modelo sea más "confiable" o util para determinar si Llueve o no lo hace

Paso 6. Tableu

Para visualizar nuestros resultados, generamos un pequeño código que nos permita descargar una muestra del dataset original sin tener que descargar los 6M de registros que hay en el dataset original

```
print("CREANDO DATASET...")

from pyspark.sql.functions import when, col, udf
```

```
from pyspark.sql.types import DoubleType
import pandas as pd # Keep pandas import if needed for toPandas()
import os # Keep os import if needed for file paths

get_positive_probability_udf = udf(lambda prob_vector: float(prob_vector[1]),

# Apply the UDF to create a new column with the positive class probability
predictions_with_prob_value = predictions.withColumn(
    "probability_positive",
    get_positive_probability_udf(col("probability"))
)


# Apply threshold 0.3 using the new probability column
predictions_with_threshold = predictions_with_prob_value.withColumn(
    "prediction_03",
    when(col("probability_positive") > 0.3, 1).otherwise(0)
)

tableau_columns = [
    'station', 'timestamp', 'lluvia_binaria', 'prediction_03', 'probability_p
    'air_temperature_dry_bulb_C', 'relative_humidity_hourly_percent',
    'wind_speed_hourly_ms', 'atmospheric_pressure_station_level_mB',
    'dew_point_temperature_C', 'hora_del_dia', 'mes', 'estacion'
]

# Select the relevant columns for Tableau
df_tableau_simple = predictions_with_threshold.select(tableau_columns)

# Rename the prediction column
df_tableau_simple = df_tableau_simple.withColumnRenamed("prediction_03", "pre

total_registros = df_tableau_simple.count()
fraction_deseada = 100000 / total_registros if total_registros > 100000 else 1

df_tableau_final = df_tableau_simple.sample(
    withReplacement=False,
    fraction=min(fraction_deseada, 1.0), # Ensure fraction is not more than 1
    seed=42
).limit(100000)  # Ensure maximum 100K records

print(f"Dataset Tableau: {df_tableau_final.count():,} registros")
print(f"Distribución de predicciones con umbral 0.3:")

# Show statistics of the new predictions
df_tableau_final.groupBy("prediction").count().show()
df_tableau_final.groupBy("lluvia_binaria", "prediction").count().orderBy("llu

# Convert to Pandas and save as CSV
# WARNING: This can be memory-intensive for large datasets.
# For very large datasets, consider saving the Spark DataFrame directly to a
```

```
output_path = "/content/drive/MyDrive/lluvia_model/tableau_lluvia_umbral03.csv

print(f"Attempting to convert Spark DataFrame with {df_tableau_final.count():
try:
    df_tableau_pandas = df_tableau_final.toPandas()
    df_tableau_pandas.to_csv(output_path, index=False)
    print(f"Pandas DataFrame saved to: {output_path}")

except Exception as e:
```

```
CREANDO DATASET...
Dataset Tableau: 50 registros
Distribución de predicciones con umbral 0.3:
+----------+-----+
|prediction|count|
+----------+-----+
|         1|    2|
|         0|   48|
+----------+-----+

+--------------+----------+-----+
```