Universidad Europea

ESCUELA DE ARQUITECTURA, INGENIERÍA Y DISEÑO

GRADO EN INGENIERÍA INFORMÁTICA

GRANDES VOLÚMENES DE DATOS

PL FINAL

Grupo 3

Destinado a
Rafael Valenzuela Moraleda

CURSO 2024-2025

TÍTULO: PL Final

AUTORES: Alberto García Cambero, Iván García Moriones, Jiale Ji Chen, Pelayo Vázquez Toledo, Raquel Garcia Dominguez y Rubén Rojo Bodega.

TITULACIÓN: Grado en Ingeniería Informática

DIRIGIDO A: Rafael Valenzuela Moraleda

FECHA: Enero de 2025

ÍNDICE

Universidad Europea

Introducción

Esta memoria documenta la solución desarrollada para el problema planteado para el grupo 3 de la asignatura Grandes Volúmenes de Datos.

Este código asegura una correcta gestión de asistencia de alumnos a las asignaturas, mejorando la planificación devolviendo métricas de análisis de asistencia y generando predicciones de los patrones de asistencia futuros.

A continuación, se desarrolla la explicación de las herramientas utilizadas y un desglose del código generado.

Tecnologías utilizadas

- PySpark
- ARIMA
- PMDARIMA
- Pandas
- JSON
- Python

PySpark: PySpark es una interfaz para Apache Spark en Python que sirve para manipular y analizar datos en un entorno distribuido.
ARIMA: ARIMA es un modelo que sirve para pronosticar posibles valores futuros en una serie temporal y significa media móvil integrada autoregresiva.
PMDARIMA: pmdarima es una librería estadística diseñada para rellenar el vacío en las capacidades de análisis de series temporales de Python.
Pandas: Pandas es una librería de Python especializada en la manipulación y el análisis de datos.
JSON: JSON es un lenguaje de texto sencillo que sirve para estructurar datos.
Python: Python es un lenguaje de programación de alto nivel.

Dataframe

El dataframe que hemos creado para la realización de nuestro proyecto incluye los siguientes atributos:

- Timestamp: la hora y fecha del registro de asistencia.
- Asignatura: el nombre de la asignatura.
- Estado Asistencia: muestra el estado de la asistencia (presente, tarde, ausente).
- Alumno: el identificador del alumno.

El dataframe ha sido creado con los siguientes detalles:

- El rango de timestamp es de Septiembre => Junio.
- El tiempo de cada clase serán 2 horas.

- Hay un total de 4 asignaturas (Administración de Sistemas, Grandes Volumenes de Datos, Ingenieria del Software, Empresa y Legislacion)
- Hay un número total de 20 alumnos, cada uno con su identificador único (nombre y apellido).
- El estado de asistencia se registra al comienzo de cada clase, en caso de llegar tarde, la hora se registra en el momento que se ingresa.
- Dejamos algunos datos nulos para comprobar el funcionamiento del filtrado.

Limpiado y filtrado

La función **mostrar_asistencia** está creado para cumplir con los siguientes requisitos:

- Leer el archivo csv, para realizar la limpieza y filtrado.
- Filtrado de los valores nulos..
- Filtrado de los valores de asistencia para su futuro análisis.
- Exportar los csv separados tanto para nulos y tanto el filtrado.

Entrada de la función:

- Se procesa un csv_asistencia con la información de la asistencia, con los campos de timestamp, asignatura, estado_asistencia, alumno.
- csv_nulos: es la ruta en donde se guardará el csv de valores nulos.
- csv_filtrado: es la ruta en donde se guardará el csv después del filtrado.

```
def mostrar_asistencia(csv_asistencia, csv_nulos, csv_filtrado):
```

Dentro de la función, encontramos distintos métodos:

1. Filter: filtramos entre las columnas los valores nulos.

```
df_nulos = df.filter(" OR ".join([f"{col} IS NULL" for col in df.columns]))
```

2. OrderBy: se utiliza para ordenar los valores para realizar un mejor análisis.
3. Write.csv: guarda el df en un csv, sobreescribiendo el csv anterior si hubiese uno.
4. Dropna: elimina los valores nulos.
5. Show: visualización del df.

Métricas

La función **calcular_metricas_y_forecast_json** tiene como propósito:

- Analizar datos de asistencia por asignaturas y alumnos
- Realizar cálculos diarios y acumulados como medias y desviaciones.
- Llamar a la función de forecast para añadir los resultados al json
- Exportar todos los resultados y métricas con un formato específico en un json.

Entrada de la función:

- Se procesará un df de entrada el cual tiene unso campos: timestamp, asignatura…
- Json_output: será la ruta en la que se generará el resultado.

```
def calcular_metricas_y_forecast_json(df, json_output):
```

Dentro de esta función del cálculo de métricas y guardado en json, podemos diferenciar el uso de varios métodos para calcular estas métricas.

1. Distinct y filter: extraen las asignaturas únicas del df y se itera sobre ellas para hacer cálculos independientes por asignatura.
2. Max(date_format) : se utiliza para ver que timestamp es el más reciente para identificar el día actual y saber a partir de qué día deben calcularse las medias.
3. GroupBy: permite agrupar filas de un df en pySpark basándose en varias columnas, para posteriormente aplicar funciones de agregación.
4. Agg: especifica el uso de una funcionde agregacion como puede ser count, mean, stddev…

```
        # Calcular totales del último día por agrupación
                                                totales_estado      =
df_ultimo_dia.groupBy("estado_asistencia").agg(
            count("alumno").alias("total"),
            collect_list("alumno").alias("alumnos")
        ).collect()
```

5. Collect_list: recopila lso valores de una columna en una lista para cada grupo. Ayuda a saber qué alumnos son los que están en cada estado.
6. Collect(): se usa en pySpark para traer los datos del df como una lista de filas.

En este caso no usamos volúmenes de datos enormes por lo que puede utilizarse ya que no compromete la memoria local. Se ha utilizado principalmente para poder manipular esos datos, crear un diccionario compatible con JSON.

Forecast

Para realizar el pronóstico de asistencia de estudiantes se utiliza el modelo ARIMA con series temporales.

Las series temporales se emplean para analizar datos que varían con el tiempo y predecir su comportamiento futuro basándose en patrones pasados. En este caso, utilizamos una frecuencia diaria.

La función encargada del forecast es **generar_forecast_por_asignatura.**
Los pasos que lleva a cabo son los siguientes:

Formatear la fecha

La columna timestamp del CSV de entrada contiene tanto la fecha en formato yyyy-MM-dd como la hora del registro de asistencia. Con esta línea de código, lo que hacemos es truncar la hora para que ARIMA se base solo en la fecha, ya que ARIMA se basa en una frecuencia regular, en este caso diaria, por lo que la hora no es necesaria a la hora del análisis.

```
data     =     data.withColumn("fecha",     date_format(col("timestamp"),
"yyyy-MM-dd"))
```

Calcular totales diarios por estado de asistencia

Se calculan agrupando por fecha y estado de asistencia, y contando cada alumno por estado de asistencia (presente, tarde, ausente).

```
# Calcular totales diarios por estado
                totales_diarios    =    data.groupBy("estado_asistencia",
"fecha").agg(count("alumno").alias("total_diario"))
```

Conversión de Spark a Pandas

Para que la manipulación de los totales diarios sea más fácil convertimos el dataframe a Pandas, nos aseguramos de que la columna fecha tiene el formato correcto y los ordenamos por fecha y estado para asegurarnos de que están en orden cronológico antes de ajustarlos al modelo ARIMA

```
    totales_pandas = totales_diarios.toPandas()
    totales_pandas["fecha"] = pd.to_datetime(totales_pandas["fecha"])
                                                totales_pandas           =
totales_pandas.sort_values(by=["estado_asistencia", "fecha"])
```

Inicialización de un diccionario

```
forecast_resultados = {"presentes": 0, "tarde": 0, "ausentes": 0}
```

Se ha optado por la utilización de un diccionario para almacenar los valores pronosticados,

Iterar el modelo ARIMA para cada estado de asistencia

```
for estado in ["presente", "tarde", "ausente"]:
                                                        grupo_estado       =
totales_pandas[totales_pandas["estado_asistencia"] == estado]
```

Asegurar que tenemos datos suficientes para el entrenamiento

Solo se entrena en caso de tener más de 10 datos para garantizar una mínima calidad a la hora de la predicción.

```
 if len(grupo_estado) > 10
```

Entrenamiento del modelo

```
serie = grupo_estado.set_index("fecha")["total_diario"]
                        modelo_auto = auto_arima(serie, seasonal=False,
stepwise=True, trace=False)
                modelo = ARIMA(serie, order=modelo_auto.order)
                ajuste = modelo.fit()
                prediccion = ajuste.forecast(steps=1)
```

Primero hacemos fecha el índice del dataframe grupo_estado que va a estar compuesto de los totales diarios, autoarima detectará los mejores parámetros para (p,d,q) en la serie temporal y predecimos únicamente el siguiente día

```
                if estado == "presente":
                                        forecast_resultados["presentes"]  =
round(prediccion.iloc[0])
                elif estado == "tarde":
                                            forecast_resultados["tarde"]  =
round(prediccion.iloc[0])
                elif estado == "ausente":
                                         forecast_resultados["ausentes"]  =
round(prediccion.iloc[0])
```

Después dependiendo de en qué estado estemos vamos a añadir la predicción redondeada al diccionario creado anteriormente

Manejo de excepciones

Todo el código del entrenamiento se hace dentro de un try catch para prevenir el fallo del código en caso de una excepción.

```
            except Exception as e:
                        print(f"Error al ajustar ARIMA para {estado}:
{str(e)}")
```

Devolución de la función

Una vez terminada la función devolverá el diccionario con los valores actualizados.

```
return forecast_resultados
```

Tests

Nuestro código consta de las siguientes funciones:

procesar_datos(spark, csv_asistencia, csv_nulos, csv_filtrado)

Aquí leemos el csv de datos, eliminamos valores nulos del mismo y generamos un nuevo documento con los datos sin errores. Ademas devuelve un dataframe con los datos que nos permitirá trabajar con las siguientes funciones.

La ruta es errónea o CSV vacío

Este es un error muy común cuando trabajamos con rutas relativas. Este error no lo maneja el código por lo que se sugirió usar manejadores de excepciones para ver si sucediese esto en que punto se encontraba el error.

Manejo de datos nulos

El manejo de datos nulos se comprueba en los dos documentos que se crean y lo hace de manera correcta.

Si ejecutamos la línea df_filtrado.isnull().sum() nos dará 0.

calcular_metricas_y_forecast_json(df, json_output)

En esta función tomamos como atributos el dataframe que hemos filtrado en la parte anterior y la ruta de salida del json que se va a crear con las métricas y el forecast que queremos.
Primero procesamos el dataframe, luego sacamos el último día de los registros. Una vez tenemos esto agrupamos el df por el estado de la asistencia, donde posteriormente generamos un diccionario para poder calcular las estadísticas acumuladas. Por último genera un forecast y construye un json.

Para probar este código hemos generado un pequeño documento .csv en el que solo tenemos 27 registros de 3 asignaturas pudiendo así controlar con exactitud como esta desenvolviéndose el código.

1. Recorremos la columna de asignaturas del df y vamos haciendo una pequeña comprobación manual de que las tres asignaturas que tenemos el .csv están. Si hubiese o faltase alguna asignatura lo veríamos en el print.

```
asignaturas_df = df.select("asignatura").distinct()
# (Mantenemos la misma lógica, .collect() porque no esperamos datos gigantes)
asignaturas = [row["asignatura"] for row in asignaturas_df.collect()]

# Comprobamos que están las asignaturas del csv reducido que hemos creado para ver las cosas ma
for asig in asignaturas:
    if asig == "Matemáticas":
        print("OK - Se encontró 'Matemáticas'.")
    elif asig == "Historia":
        print("OK - Se encontró 'Historia'.")
    elif asig == "Biología":
        print("OK - Se encontró 'Biología'.")
    else:
        print(f"FAIL - Se encontró una asignatura inesperada: '{asig}'. Revisar el CSV.")
```

```
OK - Se encontró 'Matemáticas'.
OK - Se encontró 'Historia'.
OK - Se encontró 'Biología'.
```

2. Comprobamos que estamos cogiendo el último día como pretende el código. Como vemos lo supera correctamente ya que tenemos registros del día 9 y 10 pero solo obtiene los del 11.

```
# Último día
ultimo_dia = df_asignatura.agg(
    max(date_format(col("timestamp"), "yyyy-MM-dd")).alias("ultimo_dia")
).first()["ultimo_dia"]
print(f"Último día de registros: {ultimo_dia}")
```

```
Asignatura: Matemáticas
Último día de registros: 2025-01-11
```

3. Crea un diccionario de datos y como lo hemos agrupado antes vamos a ver: para la asignatura de matemáticas tenemos ‘presentes x’ , tarde ‘y’ , ausente ‘z’. Si comparamos con el csv vemos que supera la prueba.

```
Totales del día actual:
{'presentes': 1, 'tarde': 2, 'ausentes': 1, 'alumnos_por_estado': {'presentes':
['Alumno9'], 'tarde': ['Alumno10', 'Alumno11'], 'ausentes': ['Alumno12']}}
```

```
# Diccionario de totales del día actual
totales_dia_actual = {
    "presentes": next((row["total"] for row in totales_estado
                       if row["estado_asistencia"] == "presente"), 0),
    "tarde": next((row["total"] for row in totales_estado
                   if row["estado_asistencia"] == "tarde"), 0),
    "ausentes": next((row["total"] for row in totales_estado
                      if row["estado_asistencia"] == "ausente"), 0),
}
```

```
totales_dia_actual["alumnos_por_estado"] = alumnos_por_estado
```

4. Ahora calculamos las estadísticas acumuladas. De forma manual calculamos la media y la desviación para la primera asignatura de matemáticas y vemos que lo hace bien.

```
# Calcular estadísticas acumuladas
conteos_diarios = df_asignatura.groupBy(
    "estado_asistencia",
    date_format(col("timestamp"), "yyyy-MM-dd").alias("fecha")
).agg(count("alumno").alias("total_diario"))

estadisticas_acumuladas = conteos_diarios.groupBy("estado_asistencia").agg(
    mean("total_diario").alias("media"),
    stddev("total_diario").alias("desviacion")
)

print("Estadísticas acumuladas:")
estadisticas_acumuladas.show()
```

```
+-----------------+------------------+------------------+
|estado_asistencia|             media|        desviacion|
+-----------------+------------------+------------------+
|         presente|1.6666666666666667|0.5773502691896258|
|            tarde|1.3333333333333333|0.5773502691896258|
|          ausente|               1.0|               0.0|
+-----------------+------------------+------------------+
```

```
# Matemáticas
# presente
# 2 presentes , 2 presentes  , 1 presente || (2+2+1)/3 = 1.67
    # Distancias al cuadrado:
    # (2 - 1.6667)^2 = 0.1111
    # (2 - 1.6667)^2 = 0.1111
    # (1 - 1.6667)^2 = 0.4444
        # Suma = 0.6666
        # Varianza = 0.6666 / (3 - 1) = 0.3333
        # Desviación = (0.3333)*0.5 ≈ 0.57735

# tarde
# 1 presentes , 1 presentes  , 2 presente || (1+1+2)/3 = 1.33
    # Distancias al cuadrado:
    # (1 - 1.3333)^2 = 0.1111
    # (1 - 1.3333)^2 = 0.1111
    # (2 - 1.3333)^2 = 0.4444
        # Suma = 0.6666
        # Varianza = 0.6666 / (3 - 1) = 0.3333
        # Desviación = (0.3333)*05 ≈ 0.57735

# ausente
# 1 presentes , 1 presentes  , 1 presente || 1
```

Test_forecast_por_asignatura

Lectura y procesamiento del archivo CSV

Se lee un archivo CSV con datos de asistencia por asignatura, eliminando duplicados y agrupando los registros por asignatura y fecha.

```
# Cargo el archivo CSV con los datos
df = spark.read.csv("C:/Users/ruben/Downloads/Bigggg/asistencia_clase_modificada.csv", header=True, inferSchema=True)
```

Eliminación de Duplicados

Se eliminan duplicados en las columnas timestamp, asignatura y alumno para evitar entradas repetidas.

```
# Cargo el archivo CSV con los datos
df = spark.read.csv("C:/Users/ruben/Downloads/Bigggg/asistencia_clase_modificada.csv", header=True, inferSchema=True)
```

Filtrado de Registros Inválidos

Se eliminan registros donde la asignatura es nula o vacía.

```
# Paso 1: Filtrar registros con asignaturas nulas o vacías
print("Eliminando registros con asignaturas nulas o vacías...")
df_filtrado = df.filter((col("asignatura").isNotNull()) & (col("asignatura") != ""))
```

Validación de Registros Válidos

Se valida que el DataFrame filtrado contiene registros.

```
# Paso 2: Obtener asignaturas únicas
asignaturas = [row["asignatura"] for row in df_filtrado.select("asignatura").distinct().collect()]
print(f"Asignaturas únicas detectadas: {asignaturas}")
assert len(asignaturas) > 0, "El archivo debe tener asignaturas válidas."
```

Obtención de Asignaturas Únicas

Se extraen las asignaturas únicas del DataFrame para procesarlas individualmente.

```
# Paso 2: Obtener asignaturas únicas
asignaturas = [row["asignatura"] for row in df_filtrado.select("asignatura").distinct().collect()]
print(f"Asignaturas únicas detectadas: {asignaturas}")
assert len(asignaturas) > 0, "El archivo debe tener asignaturas válidas."
```

Procesamiento por Asignatura

Se recorre cada asignatura para calcular los resultados de asistencia y realizar el forecast

```
# Proceso los datos por asignatura
for asignatura in asignaturas:
    print(f"Procesando asignatura: {asignatura}")

    # Filtro los datos de la asignatura actual
    df_asignatura = df.filter(col("asignatura") == asignatura)

    # Llamo a la función de forecast para estos datos
    resultados = generar_forecast_por_asignatura(df_asignatura)
```

Validación de Registros por Asignatura

Se valida que cada asignatura tenga datos suficientes para realizar el forecast.

```
    # Validar que hay datos suficientes para procesar
    registros_asignatura = df_asignatura.count()
    print(f"Total de registros para {asignatura}: {registros_asignatura}")
    assert registros_asignatura > 0, f"No hay registros para la asignatura {asignatura}."
```

Generación del Forecast

Para cada asignatura, se genera un forecast utilizando la función generar_forecast_por_asignatura.

```
# Ejecutar forecast para esta asignatura
resultados = generar_forecast_por_asignatura(df_asignatura)
```

Validación de Resultados del Forecast

Se verifica que los resultados del forecast sean válidos, incluyendo las claves esperadas y valores no negativos.

```
# Paso 4: Validar los resultados globales
print("Validando los resultados globales...")
for asignatura, resultados in resultados_globales.items():
    assert isinstance(resultados, dict), f"El resultado para {asignatura} debe ser un diccionario"
    assert "presentes" in resultados, f"Debe contener la clave 'presentes' para {asignatura}"
    assert "tarde" in resultados, f"Debe contener la clave 'tarde' para {asignatura}"
    assert "ausentes" in resultados, f"Debe contener la clave 'ausentes' para {asignatura}"
    assert resultados["presentes"] >= 0, f"El valor de 'presentes' no puede ser negativo para {asignatura}"
    assert resultados["tarde"] >= 0, f"El valor de 'tarde' no puede ser negativo para {asignatura}"
    assert resultados["ausentes"] >= 0, f"El valor de 'ausentes' no puede ser negativo para {asignatura}"
```

Resultados Globales

Se imprimen y validan los resultados globales de todas las asignaturas.

```
# Resultados finales
print("Resultados Globales del Forecast:", resultados_globales)
print("Test completado exitosamente.")
```

Resumen

El código principal se encuentra en la rama develop dentro de la clase AsistenciaClase.py, mientras que los tests se encuentran en la rama del mismo nombre.
Las funciones de lectura y filtrado del CSV de entrada, la generación de métricas y el resultado del pronóstico se juntan en el archivo .py, configurándose y ejecutándose gracias a las siguientes líneas:

```
# Configuración y rutas
spark                                                                    =
SparkSession.builder.appName("AsistenciaClaseForecast").getOrCreate()
# Rutas para archivos entrada y salida
csv_input = "asistencia_clase_modificada.csv"
csv_nulos = "nulos.csv"
csv_filtrado = "filtrado.csv"
json_output = "asistencia_forecast.json"

registro_asistencia   =   procesar_datos(spark,   csv_input,   csv_nulos,
csv_filtrado)
calcular_metricas_y_forecast_json(registro_asistencia, json_output)
```

Por lo tanto, la rama develop contiene el código útil final. El resto de ramas han servido como método de trabajo individual, juntando todo en esta rama final para unificar todo el trabajo y verificar el funcionamiento, dejando intacta la rama master. La unificación se ha hecho a través de merge y commits.

Conclusiones

Después de analizar el código planteado, podemos afirmar que se han cumplido los siguientes objetivos:

1. **Automatización del procesamiento de datos**
   Se realiza el filtrado del dataset teniendo en cuenta posibles corrupciones en el registro (valores nulos, estados no válidos etc.) y se separan del dataset principal, asegurando la posterior integridad de las predicciones y métricas.

2. **Cálculo de métricas y estadísticas acumuladas**
   Se han implementado métricas clave como el total diario de asistencia, medias y desviaciones estándar para diferentes estados de asistencia (presente, tarde, ausente) por asignatura. Estas estadísticas proporcionan una visión detallada del comportamiento histórico y facilitan la visualización de datos.

3. **Predicción mediante modelos de series temporales (ARIMA)**
   El uso de ARIMA permite predecir mediante series temporales el número esperado de alumnos para cada estado de asistencia en las diferentes asignaturas.

4. **Salida en formato JSON estructurado**
   Los resultados del procesamiento, análisis y predicción se almacenan en un archivo JSON. Este formato facilita la visualización de los datos y la migración de los mismos en caso de ser necesario.

Limitaciones

Durante el desarrollo del trabajo, se han detectado limitaciones que no han podido llevarse a cabo:

1. **Dependencia de la calidad de los datos**
   El modelo y las métricas dependen en gran medida de la calidad de los datos de entrada. La presencia de errores, datos incompletos o inconsistencias en los registros puede afectar la precisión de las métricas y predicciones.

2. **Simplicidad del modelo ARIMA**
   Aunque ARIMA es un modelo robusto, su capacidad para capturar patrones estacionales o complejidades en los datos es limitada. Esto podría afectar la precisión en contextos donde la asistencia tenga fuertes variaciones estacionales (e.g., exámenes o días especiales).

3. **Contexto limitado en la interpretación del forecast**
   El sistema genera predicciones basadas únicamente en datos históricos sin considerar posibles variables externas (e.g., cambios en horarios, días festivos, clima), lo que podría dar más contexto a las predicciones.

Oportunidades de mejora

Como líneas de mejora futuras, se plantea incluir:

1. **Modo en tiempo real**
   Implementar un flujo continuo con PySpark Streaming para procesar nuevos registros mientras se generan.

2. **Integrar las predicciones**
   Explorar técnicas para realizar las predicciones directamente en PySpark (e.g., integración con MLlib o SparkML) permitiría un mayor paralelismo y escalabilidad.

3. **Incorporación de análisis visuales**
   Generar gráficos o dashboards interactivos que muestren las métricas y predicciones enriquecería el entendimiento y presentación de los resultados.