Conocimientos pyspark

Importar sparksession:

from pyspark.sql import SparkSession

Crear sparksession:

#crear sparksession

spark=SparkSession.builder.appName('Dataframe').getOrCreate()

spark

Leer conjunto de datos, ver esquema e imprimir:

#leer conjunto de datos

df\_pyspark=spark.read.option('header','true').csv('test1.csv',inferSchema=True)

#ver esquema

df\_pyspark.printSchema()

#mostrar datos

df\_pyspark.show()

#otra forma para leer

df\_pyspark=spark.read.csv('test1.csv',header=True,inferSchema=True)

#con separador ;

data = SpSession.read.csv('cars.csv', header=True, sep=";")

data.show(5)

Cargar datos definiendo esquema:

# importar tipos

from pyspark.sql.types import StructType, StructField, StringType, IntegerType, DoubleType

# definir esquema

schema = StructType([

StructField(name="id", dataType=IntegerType(), nullable=True),

StructField(name="name", dataType=StringType(), nullable=True),

StructField(name="category", dataType=StringType(), nullable=True),

StructField(name="quantity", dataType=IntegerType(), nullable=True),

StructField(name="price", dataType=DoubleType(), nullable=True)

])

# leer csv

csv\_file\_path = "./data/products.csv"

df = spark.read.csv(csv\_file\_path, header=True, schema=schema)

Leer json:

#json, cada elemento en una linea {}{}

json\_file\_path = "./data/products\_singleline.json"

df = spark.read.json(json\_file\_path)

# ver esquema

df.printSchema()

# ver contenido

df.show(5)

#en array con json [{}]

json\_file\_path = "./data/products\_multiline.json"

df = spark.read.json(json\_file\_path, multiLine=True)

# ver esquema

df.printSchema()

# ver contenido

df.show(5)

Tipo de una variable:

type(df\_pyspark)

Primeros o ultimos elementos:

df\_pyspark.head(3)

df\_pyspark.tail(3)

Seleccionar columnas:

#seleccionar columnas, sin show para guardarlo en otro dataframe

df\_pyspark.select(['Name','Experience']).show()

#seleccionar columnas

finalized\_data=output.select("Independent Features","Salary")

from os import truncate

#seleccionar columna

data.select(data.Car).show(truncate=False)

#otra forma

data.select(data['car']).show(truncate=False)

#otra forma

from pyspark.sql.functions import col

data.select(col('car')).show(truncate=False)

#seleccionar multiples columnas

data.select(data.Car, data.Cylinders).show(truncate=False)

#otra forma

data.select(data['car'], data['cylinders']).show(truncate=False)

#otra forma

from pyspark.sql.functions import col

data.select(col('CAR'), col('CYLINDERS')).show(truncate=False)

# seleccionar columnas

selected\_columns = df.select("id", "name", "price")

print("Selected Columns:")

selected\_columns.show(10)

Ver tipos de columnas:

#ver tipos de las columnas

df\_pyspark.dtypes

Resumen de los datos:

#resumen de los datos

df\_pyspark.describe().show()

print("Análisis algunas variables:")

titanic\_df.select("Age", "Sex", "Survived", "Pclass", "Fare").describe().show()

Añadir columna en base a otra:

#añadir columna en base a otra columna

df\_pyspark=df\_pyspark.withColumn('Experience After 2 year',df\_pyspark['Experience']+2)

# añadir columna calculada

df\_with\_new\_column = df.withColumn("revenue", df.quantity \* df.price)

print("DataFrame with New Column:")

df\_with\_new\_column.show(10)

Eliminar columna:

#eliminar columna

df\_pyspark=df\_pyspark.drop('Experience After 2 year')

# eliminar columnas

dropped\_columns = df.drop("quantity", "category")

print("Dropped Columns:")

dropped\_columns.show(10)

Renombrar columna:

#renombrar columna

df\_pyspark.withColumnRenamed('Name','New Name').show()

SparkSession y SparkContext

SparkContext: Se utiliza para crear RDDs (Resilient Distributed Datasets), que son la estructura de datos fundamental en Spark. SparkContext también administra la comunicación con el cluster Spark, coordinando la ejecución de operaciones en el cluster y administrando la memoria.

SparkSession: Es una interfaz unificada de nivel superior que reemplaza a las antiguas clases SQLContext y HiveContext en versiones anteriores de Spark. SparkSession proporciona una forma conveniente de trabajar con Apache Spark y los datos estructurados, como DataFrames y Datasets, así como con SQL. Además de encapsular SparkContext, SparkSession proporciona funcionalidades adicionales

En resumen, SparkContext es la interfaz principal para interactuar con Spark en un nivel más bajo, mientras que SparkSession es una capa más alta que proporciona una interfaz más fácil de usar para trabajar con datos estructurados y realizar operaciones SQL

Filtras filas:

#personas con salario inferior a 20.000, sin show para guardarlo en otro dataframe

df\_pyspark.filter("Salary<=20000").show()

#solo mostrando columnas necesarias

df\_pyspark.filter("Salary<=20000").select(['Name','age']).show()

#otra forma

df\_pyspark.filter(df\_pyspark['Salary']<=20000).show()

#or con | y and con &

df\_pyspark.filter((df\_pyspark['Salary']<=20000) &

(df\_pyspark['Salary']>=15000)).show()

#opuesto a menores de 20.000

df\_pyspark.filter(~(df\_pyspark['Salary']<=20000)).show()

# filtrar segun condicion

filtered\_data = df.filter(df.quantity > 20)

print("Filtered Data:", filtered\_data.count())

filtered\_data.show()

Agrupar y realizar calculo:

#agrupo por nombre y sumo salarios

df\_pyspark.groupBy('Name').sum().show()

#agrupo por nombre y calculo media

df\_pyspark.groupBy('Name').avg().show()

#agrupo por departamentos y calculo media

df\_pyspark.groupBy('Departments').mean().show()

#agrupo por departamentos y cuento

df\_pyspark.groupBy('Departments').count().show()

# agrupar y aplicar diferentes funciones sobre cada columna

grouped\_data = df.groupBy("category").agg({"quantity": "sum", "price": "avg"})

print("Grouped and Aggregated Data:")

grouped\_data.show()

#agrupa por seso y calcula media sobreviven

print("Tasa de supervivencia por género:")

titanic\_df.groupBy("Sex").agg({"Survived": "mean"}).show()

Join de tablas:

# hacer join

df2 = df.select("id", "category").limit(10)

joined\_data = df.join(df2, "id", "inner")#variable por la que unimos

print("Joined Data:")

joined\_data.show()

Sumar todos los valores de una fila:

#suma de todos los salarios

df\_pyspark.agg({'Salary':'sum'}).show()

Eliminar nulos:

#elimina nulos y muestra

df\_pyspark.na.drop().show()

#elimina la fila si hay algun nulo

#all necesita todos nulos en la fila

df\_pyspark.na.drop(how="any").show()

#si al menos 3 campos completos, mantiene la fila

df\_pyspark.na.drop(how="any",thresh=3).show()

#solo elimina filas si tienen null en esa columna

df\_pyspark.na.drop(how="any",subset=['Age']).show()

#Completar campos vacios de columnas con el valor que queramos

df\_pyspark.na.fill('Missing Values',['Experience','age']).show()

Completar campos vacios con la media:

from pyspark.ml.feature import Imputer

#completa campos vacios con la media de la columna

imputer = Imputer(

inputCols=['age', 'Experience', 'Salary'],

outputCols=["{}\_imputed".format(c) for c in ['age', 'Experience', 'Salary']]

).setStrategy("median")

# aplicar transformacion

imputer.fit(df\_pyspark).transform(df\_pyspark).show()

Regresion lineal:

from pyspark.sql import SparkSession

spark=SparkSession.builder.appName('Missing').getOrCreate()

#cargar datos

training = spark.read.csv('test1.csv',header=True,inferSchema=True)

#ver nombres columnas

training.columns

#age y experience columna utilizadas para la prediccion, se ponen en vector

from pyspark.ml.feature import VectorAssembler

featureassembler=VectorAssembler(inputCols=["age","Experience"],outputCol="Independent Features")

output=featureassembler.transform(training)

output.show()

output.columns

#seleccionar columnas

finalized\_data=output.select("Independent Features","Salary")

#tenemos los atributos en un vector y la clase a predecir

finalized\_data.show()

from pyspark.ml.regression import LinearRegression

#separamos datos

train\_data,test\_data=finalized\_data.randomSplit([0.75,0.25])

#decimos cuales son los atributos y cual la clase

regressor=LinearRegression(featuresCol='Independent Features', labelCol='Salary')

regressor=regressor.fit(train\_data)

#coeficientes

regressor.coefficients

#interceptor

regressor.intercept

#predicion

pred\_results=regressor.evaluate(test\_data)

#ver resultados

pred\_results.predictions.show()

pred\_results.meanAbsoluteError,pred\_results.meanSquaredError

Otro ejemplo de regresion con stringIndexer:

from pyspark.sql import SparkSession

spark=SparkSession.builder.appName('Regr').getOrCreate()

#cargamos datos

df = spark.read.csv('tips.csv',header=True,inferSchema=True)

df.printSchema()

df.columns

df.show()

#para los atributos categoricos

from pyspark.ml.feature import StringIndexer

#sex la convertimos a numeros

indexer=StringIndexer(inputCol="sex",outputCol="sex\_indexed")

df\_r=indexer.fit(df).transform(df)

df\_r.show()

#convertimos el resto de atributos categoricos

indexer=StringIndexer(inputCols=["smoker","day","time"],outputCols=["smoker\_indexed","day\_indexed","time\_index"])

df\_r=indexer.fit(df\_r).transform(df\_r)

df\_r.show()

#creamos vectores con atributos utilizados para predecir

from pyspark.ml.feature import VectorAssembler

featureassembler=VectorAssembler(inputCols=['tip','size','sex\_indexed','smoker\_indexed','day\_indexed', 'time\_index'],outputCol="Independent Features")

output=featureassembler.transform(df\_r)

output.select('Independent Features').show()

output.show()

#datos finales: variables para predecir en un vector y variable a predecir

finalized\_data=output.select("Independent Features","total\_bill")

finalized\_data.show()

from pyspark.ml.regression import LinearRegression

#separamos datos y hacemos regresion

train\_data,test\_data=finalized\_data.randomSplit([0.75,0.25])

regressor=LinearRegression(featuresCol='Independent Features', labelCol='total\_bill')

regressor=regressor.fit(train\_data)

print(regressor.coefficients)

print(regressor.intercept)

#prediciones

pred\_results=regressor.evaluate(test\_data)

#comparacion

pred\_results.predictions.show()

#metricas

pred\_results.r2,pred\_results.meanAbsoluteError,pred\_results.meanSquaredError

Ejecutar en databriks:

#crear cluster y subir archivo

# localizacion databricks

file\_location = "/FileStore/tables/tips.csv"

file\_type = "csv"

# cargar datos

df =spark.read.csv(file\_location,header=True,inferSchema=True)

df.show()

Agrupar y contar:

data.groupBy('Horsepower').count().show(5)

data.groupBy('Origin','Model').count().show()

Filas y columnas e imprimir con formato:

#cuenta filas

num\_fils = data.count()

print("Total de registros:", num\_fils)

#columnas

num\_cols = len(data.columns)

#imprimir con formato

print("El DataFrame tiene {} filas y {} columnas.".format(num\_fils, num\_cols))

Contar con filtro:

#contar con filtro

europa= data.filter(col('Origin')=="US").count()

print("Total de registros en US:", europa)

#contar filtrados

usa= data.filter((col('Origin')=="US")&(col('Horsepower')=="175.0")).count()

print("Total de registros:", usa)

Filtrar filas:

#guardar dataframe

filtrado=data.filter((col('Origin')=="US")&(col('Horsepower')=="175.0")).show(truncate=False)

filtrado

Ordenar:

#ascending se puede quitar

data.orderBy('Cylinders',ascending=False).show(truncate=False)

# ordenamos pr columna

sorted\_data = df.orderBy("price")

print("Sorted Data:")

sorted\_data.show(10)

# orden descendente

from pyspark.sql.functions import col, desc

sorted\_data = df.orderBy(col("price").desc(), col("id").desc())

print("Sorted Data Descending:")

sorted\_data.show(10)

Seleccionar elementos distintos:

# seleccionar elementos distintos

distinct\_rows = df.select("category").distinct()

print("Distinct Product Categories:")

distinct\_rows.show()

Ordenar groupby resultado:

#ordenar group by resultado

data.groupBy('Origin').count().orderBy('count', ascending=False).show()

crear rdd con tuplas y ver sus datos:

# crear rdd de tuplas

data = [("Alice", 25), ("Bob", 30), ("Charlie", 35), ("Alice", 40)]

rdd = spark.sparkContext.parallelize(data)

# ver sus datos

print("All elements of the rdd: ", rdd.collect())

Numero elementos rdd:

# contar elementos

count = rdd.count()

print("The total number of elements in rdd: ", count)

Primer elemento rdd:

# obtener primer elemento

first\_element = rdd.first()

print("The first element of the rdd: ", first\_element)

# coger 2 primeros

taken\_elements = rdd.take(2)

print("The first two elements of the rdd: ", taken\_elements)

foreach con lambda:

# Foreach para cada uno imprimir

rdd.foreach(lambda x: print(x))

Map para aplicar conversion

# convierte nombre a mayusculas

mapped\_rdd = rdd.map(lambda x: (x[0].upper(), x[1]))

#imprimir

result = mapped\_rdd.collect()

print("rdd with uppercease name: ", result)

Filtrar rdd:

# filtra los de edades mayores a 30

filtered\_rdd = rdd.filter(lambda x: x[1] > 30)

print(filtered\_rdd.collect())

Suma de edades reduciendo por key:

# calcula suma de edades por nombre

reduced\_rdd = rdd.reduceByKey(lambda x, y: x + y)

print(reduced\_rdd.collect())

Ordenar descendente

# ordena por edad descendente

sorted\_rdd = rdd.sortBy(lambda x: x[1], ascending=False)

print(sorted\_rdd.collect())

Guardar resultado en txt

# guardar en txt

rdd.saveAsTextFile("output.txt")

# leer de text

rdd\_text = spark.sparkContext.textFile("output.txt")

rdd\_text.collect()

Parar sesion:

#parar sesion

spark.stop()

Contar apariciones palabras:

#separar, inicializar, agrupar y ordenar

rdd = spark.sparkContext.textFile("./data/data.txt")

result\_rdd = rdd.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b).sortBy(lambda x: x[1], ascending=False)

#mostrar 10 mayores

print(result\_rdd.take(10))

# otra forma: cargar texto

df = spark.read.text("./data/data.txt")

#lo mismo, separar palabras, agrupar contando y ordenar

result\_df = df.selectExpr("explode(split(value, ' ')) as word") \

.groupBy("word").count().orderBy(desc("count"))

#mostrar 10 mayores

print(result\_df.take(10))

Crear tabla temporal sql

# crear tabla temporal

df.createOrReplaceTempView("my\_table")

Consulta sql mayores 25

# seleccionar filas con edad superior a 25

result = spark.sql("SELECT \* FROM my\_table WHERE age > 25")

result.show()

Calcular valor medio agrupando

# calculamos salario medio por genero

avg\_salary\_by\_gender = spark.sql("SELECT gender, AVG(salary) as avg\_salary FROM my\_table GROUP BY gender")

avg\_salary\_by\_gender.show()

Comprobar si existe vista y eliminarla:

# comprobamos si existe

view\_exists = spark.catalog.tableExists("people")

view\_exists

# la eliminamos

spark.catalog.dropTempView("people")

# comprobamos si existe

view\_exists = spark.catalog.tableExists("people")

view\_exists

Crear dataframes y vistas:

# creamos dataframes

employee\_data = [

(1, "John"), (2, "Alice"), (3, "Bob"), (4, "Emily"),

(5, "David"), (6, "Sarah"), (7, "Michael"), (8, "Lisa"),

(9, "William")

]

employees = spark.createDataFrame(employee\_data, ["id", "name"])

salary\_data = [

("HR", 1, 60000), ("HR", 2, 55000), ("HR", 3, 58000),

("IT", 4, 70000), ("IT", 5, 72000), ("IT", 6, 68000),

("Sales", 7, 75000), ("Sales", 8, 78000), ("Sales", 9, 77000)

]

salaries = spark.createDataFrame(salary\_data, ["department", "id", "salary"])

employees.show()

salaries.show()

# creamos vistas temporales

employees.createOrReplaceTempView("employees")

salaries.createOrReplaceTempView("salaries")

Operación In:

# buscamos empleados cuyo sueldo esta debajo de la media

result = spark.sql("""

SELECT name

FROM employees

WHERE id IN (

SELECT id

FROM salaries

WHERE salary > (SELECT AVG(salary) FROM salaries)

)

""")

result.show()

Join sql:

from pyspark.sql.window import Window

from pyspark.sql import functions as F

#hacemos join

employee\_salary = spark.sql("""

select salaries.\*, employees.name

from salaries

left join employees on salaries.id = employees.id

""")

employee\_salary.show()

Crear ventana:

# creamos ventana por departamentos

window\_spec = Window.partitionBy("department").orderBy(F.desc("salary"))

# ranking por departamento basandose en el salario

employee\_salary.withColumn("rank", F.rank().over(window\_spec)).show()

Matriz de correlaccion:

#matriz de correlaccion, clase c embarcados en s 17 pasajeros

print("Distribución de la clase de los pasajeros por puerto de embarque:")

class\_by\_embarked = titanic\_df.crosstab("Pclass", "Embarked")

class\_by\_embarked.show()

#similar pero con group by

print("Distribución del puerto de embarque por clase de los pasajeros:")

embarked\_by\_class = titanic\_df.groupBy("Pclass", "Embarked").count().orderBy("Pclass", "Embarked")

embarked\_by\_class.show()

Contar elementos y calcular porcentaje:

#contamos total

total\_passengers = titanic\_df.count()

total\_passengers

from pyspark.sql.functions import count, col

# numero hombres y mujeres y su porcentaje

gender\_distribution = titanic\_df.groupBy("Sex").agg(

count("\*").alias("count"),

((count("\*") / total\_passengers) \* 100).alias("percentage")

)

# Mostrar los resultados

print("Número de hombres y mujeres que embarcaron y su porcentaje sobre el total de pasajeros:")

gender\_distribution.show()

Porcentaje con diferentes combinaciones de atributos:

from pyspark.sql.functions import round

# porcentaje combinacion atributos

survival\_percentage = titanic\_df.groupBy("Embarked", "Sex", "Survived").agg(

(count("\*") / total\_passengers \* 100).alias("percentage")

)

# Redondear

survival\_percentage = survival\_percentage.withColumn("percentage", round("percentage", 2))

# Ordenar

survival\_percentage = survival\_percentage.orderBy("Embarked", "Sex", "Survived")

# tabla

print("Porcentaje de pasajeros para cada combinación de puerto de embarque, género y si sobrevivieron:")

survival\_percentage.show()

Tabla de frecuencia:

from pyspark.sql.functions import expr

# Definir rangos de edades

age\_ranges = [(i, i + 4) for i in range(0, 100, 5)]

# columna con rango de edad

titanic\_df\_with\_age\_range = titanic\_df.withColumn("AgeRange", expr(

"CASE WHEN Age IS NULL THEN 'Unknown' " +

"ELSE CAST(FLOOR(Age / 5) \* 5 AS INT) + 1 || '-' || CAST(FLOOR(Age / 5) \* 5 + 5 AS STRING) END"

))

titanic\_df\_with\_age\_range.show()

# agrupar y ordenar

age\_distribution = titanic\_df\_with\_age\_range.groupBy("AgeRange").count().orderBy("AgeRange")

# Mostrar

print("Tabla de frecuencia de pasajeros por rangos de edades de 5 años:")

age\_distribution.show()

Buscar elemento con valor mas alto:

from pyspark.sql.functions import mean

# agrupar por puerto y calcular media edad

age\_by\_embarked = titanic\_df.groupBy("Embarked").agg(round(mean("Age"),2).alias("AverageAge")).na.drop()

# puerto con edad media mas alta

port\_with\_highest\_average\_age = age\_by\_embarked.orderBy(age\_by\_embarked["AverageAge"].desc()).first()

# resultado

print("El puerto donde la edad media de los pasajeros que embarcaron fue la más alta es:", port\_with\_highest\_average\_age["Embarked"])

print("La edad media en ese puerto fue:", port\_with\_highest\_average\_age["AverageAge"])

Porcentaje cumpliendo condicion:

from pyspark.sql.functions import when, col

# evaluar si tenia hijos

titanic\_df\_with\_children = titanic\_df.withColumn("HadChildren", when(col("Parch") > 0, 1).otherwise(0))

# porcenajte pasajeros sobrevieiro segun numero de hijos

survival\_percentage\_by\_children = titanic\_df\_with\_children.groupBy("HadChildren").agg(

(round((count(when(col("Survived") == 1, True)) / count("\*") \* 100), 2)).alias("SurvivalPercentage")

)

# resultado

print("Porcentaje de pasajeros que sobrevivieron según si tenian hijos:")

survival\_percentage\_by\_children.show()

Valor medio por combinaciones de atributos:

from pyspark.sql.functions import avg

# tarifa media por clase y puerto de embarque

average\_fare\_by\_class\_and\_embarked = titanic\_df.groupBy("Pclass", "Embarked") \

.agg(round(avg("Fare"), 2).alias("AverageFare"))

average\_fare\_by\_class\_and\_embarked = average\_fare\_by\_class\_and\_embarked.orderBy(average\_fare\_by\_class\_and\_embarked["AverageFare"].desc())

# resultado

print("Tarifa media por clase y puerto de embarque:")

average\_fare\_by\_class\_and\_embarked.show()

Distirbucion segun dos atributos:

from pyspark.sql.functions import col, floor, count, when

# intervalos

titanic\_df\_with\_age\_group = titanic\_df.withColumn("AgeGroup", floor(col("Age") / 5) \* 5)

titanic\_df\_with\_age\_group.show()

# Agrupar por intervalo de edad y clase y contar. clase 1 y agegroup=10 pasajeros

age\_class\_counts = titanic\_df\_with\_age\_group.groupBy("AgeGroup", "Pclass").count()

age\_class\_counts.show()

# contar el numero de pasajers en cada grupo de edades. Agegroup 15 de todas las clases =10 pasajeros

total\_in\_age\_group = titanic\_df\_with\_age\_group.groupBy("AgeGroup").agg(count("\*").alias("TotalCount"))

total\_in\_age\_group.show()

#unir dataframes distribucion edades e intervalos. Edad 15 de todas las clases total 10 pasajeros

age\_class\_distribution = age\_class\_counts.join(total\_in\_age\_group, "AgeGroup")

age\_class\_distribution.show()

# calcular porcentaje pasajeros dentro de cada elemento. Edad 15 hay 10 pasajerps. 3 clase 1, 3 clase 2...

age\_class\_distribution = age\_class\_distribution.withColumn(

"Percentage",

round((col("count") / col("TotalCount")) \* 100, 2)

)

age\_class\_distribution.show()

# ordenar

age\_class\_distribution = age\_class\_distribution.orderBy("AgeGroup", "Pclass")

age\_class\_distribution.show()

# resultado

print("Distribución de pasajeros según la edad y la clase por intervalos:")

age\_class\_distribution.show()

Hallar elemento con mayor valor:

from pyspark.sql.functions import col

# Filtrar embarcaron en s y son hombres

filtered\_data = titanic\_df.filter((col("Embarked") == "S") & (col("Sex") == "male"))

filtered\_data.show()

# Agrupar por clase y contar hombre sobrevivieron. Clase 1 sobrevivieron 17.

survived\_men\_by\_class = filtered\_data.groupBy("Pclass").agg(count(when(col("Survived") == 1, True)).alias("SurvivedMen"))

survived\_men\_by\_class.show()

# clase donde mas hombres sobrevivieron en el puerto s

most\_survived\_class = survived\_men\_by\_class.orderBy(survived\_men\_by\_class["SurvivedMen"].desc()).first()

# resultado

print("La clase en la que más hombres sobrevivieron de los que embarcaron en el puerto 'S' fue la clase:", most\_survived\_class["Pclass"])

print("Número de hombres sobrevivientes en esa clase:", most\_survived\_class["SurvivedMen"])

Contar nulos:

#contar filas con nulos

num\_rows\_with\_null = titanic\_df.na.drop().count()

# resultado

print(f"El número de filas con al menos un valor nulo es: {titanic\_df.count() - num\_rows\_with\_null}")

Nulos por columnas:

# contar nulos por columnas

null\_counts = [(column, titanic\_df.where(col(column).isNull()).count()) for column in titanic\_df.columns]

#resultados

for column, count in null\_counts:

print(f"Columna '{column}' tiene {count} valor(es) nulo(s).")

Reemplazar nulos por la media:

# Media de age

mean\_age = titanic\_df.select(mean(titanic\_df['Age'])).collect()[0][0]

# Nulos reemplazamos por la media

titanic\_df = titanic\_df.na.fill(mean\_age, subset=['Age'])

titanic\_df.show()

Eliminar filas con nulo en atributo:

# eliminar filas con embarked null

titanic\_df = titanic\_df.na.drop(subset=["Embarked"])

Stringindexer:

from pyspark.ml.feature import StringIndexer

from pyspark.ml import Pipeline

# indices de salida

indexers = [

StringIndexer(inputCol=col, outputCol=col + "Index")

for col in ["Sex", "Pclass", "Embarked"]

]

# Pipeline

pipeline = Pipeline(stages=indexers)

# Aplicarla

titanic\_df\_indexed = pipeline.fit(titanic\_df).transform(titanic\_df)

# resultado

titanic\_df\_indexed.select("Sex", "SexIndex", "Pclass", "PclassIndex", "Embarked", "EmbarkedIndex").show()

Correlaccion entre atributos:

from pyspark.ml.linalg import Vectors

from pyspark.ml.stat import Correlation

# Indices

index\_columns = ["SexIndex", "PclassIndex", "EmbarkedIndex"]

# columnas a vector

from pyspark.ml.feature import VectorAssembler

assembler = VectorAssembler(inputCols=index\_columns, outputCol="features")

# dataframe con columnas seleccionadas

df\_assembled = assembler.transform(titanic\_df\_indexed).select("features")

#matriz de correlación

correlation\_matrix = Correlation.corr(df\_assembled, "features").collect()[0][0]

# Mostrar la matriz: 1 mucha correlaccion, 0 poca

print("Matriz de correlación entre SexIndex, PclassIndex y EmbarkedIndex:")

print(correlation\_matrix)

#group by de las tres variables

grouped\_df = titanic\_df.groupBy("Sex", "Pclass", "Embarked").count().orderBy("Sex")

grouped\_df.show()

Reemplazar outliers por limite:

#outliers edad

#rango intercuartílico (IQR)

quantiles = titanic\_df.approxQuantile("Age", [0.25, 0.75], 0.05)

Q1 = quantiles[0]

Q3 = quantiles[1]

IQR = Q3 - Q1

# limites

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

# Identificar outliers

outliers = titanic\_df.filter((col("Age") < lower\_bound) | (col("Age") > upper\_bound))

# Mostrarlos

outliers.show()

Normalizar atributos:

#nomralizar

from pyspark.ml.feature import MinMaxScaler

# Combinar las columnas en un vector

assembler = VectorAssembler(inputCols=["Age", "Fare"], outputCol="features")

df\_assembled = assembler.transform(titanic\_df)

#MinMaxScaler

scaler = MinMaxScaler(inputCol="features", outputCol="features\_scaled")

# Ajustar y normalizar

scaler\_model = scaler.fit(df\_assembled)

df\_normalizado = scaler\_model.transform(df\_assembled)

# Mostrar

df\_normalizado.show()

Estandarizar:

#estandarizar

from pyspark.ml.feature import StandardScaler

# Combinar columnas

assembler = VectorAssembler(inputCols=["Age", "Fare"], outputCol="features")

df\_assembled = assembler.transform(titanic\_df\_indexed)

# StandardScaler

scaler = StandardScaler(inputCol="features", outputCol="features\_scaled")

# Ajustar

scaler\_model = scaler.fit(df\_assembled)

df\_estandarizado = scaler\_model.transform(df\_assembled)

# Combinar categoricas convertidas y numericas en un vector

assembler\_total = VectorAssembler(inputCols=["features\_scaled", "SexIndex","EmbarkedIndex","PclassIndex"], outputCol="features\_total")

df\_final = assembler\_total.transform(df\_estandarizado)

df\_final.show(truncate=False)

Arbol de decision para clasificacion:

from pyspark.sql import SparkSession

from pyspark.ml.feature import VectorAssembler

from pyspark.ml.classification import DecisionTreeClassifier

from pyspark.ml.evaluation import BinaryClassificationEvaluator

from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

from pyspark.ml.feature import StringIndexer

from pyspark.ml import Pipeline

spark = SparkSession.builder \

.appName("Titanic Decision Tree Classifier with Cross Validation") \

.getOrCreate()

# Cargar datos

titanic\_df = spark.read.csv("titanic.csv", header=True, inferSchema=True)

titanic\_df = titanic\_df.dropna(subset=['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked'])

# Convertir variables categóricas a numéricas

indexers = [

StringIndexer(inputCol=column, outputCol=column+"\_index").fit(titanic\_df)

for column in ["Sex", "Embarked"]

]

pipeline = Pipeline(stages=indexers)

titanic\_df = pipeline.fit(titanic\_df).transform(titanic\_df)

# Seleccionar las características para el modelo

feature\_columns = ['Pclass', 'Sex\_index', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked\_index']

assembler = VectorAssembler(inputCols=feature\_columns, outputCol="features")

data = assembler.transform(titanic\_df).select("features", "Survived")

# Dividir los datos en 3

train\_data, test\_data, val\_data = data.randomSplit([0.6, 0.2, 0.2], seed=42)

# modelo

dt = DecisionTreeClassifier(featuresCol='features', labelCol='Survived')

# cuadrícula

param\_grid = ParamGridBuilder() \

.addGrid(dt.maxDepth, [3, 5, 7]) \

.addGrid(dt.minInfoGain, [0.0, 0.1, 0.2]) \

.build()

# evaluador

evaluator = BinaryClassificationEvaluator(labelCol='Survived')

# validación cruzada

crossval = CrossValidator(estimator=dt,

estimatorParamMaps=param\_grid,

evaluator=evaluator,

numFolds=5)

# Entrenar el modelo

cv\_model = crossval.fit(train\_data)

# Hacer predicciones

predictions = cv\_model.transform(test\_data)

# Evaluar el rendimiento

accuracy = evaluator.evaluate(predictions)

print("Accuracy:", accuracy)

# Detener la sesión

spark.stop()

Random forest para regresion:

from pyspark.sql import SparkSession

from pyspark.ml.feature import VectorAssembler, StandardScaler

from pyspark.ml.regression import RandomForestRegressor

from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit

from pyspark.sql.functions import col

spark = SparkSession.builder \

.appName("Boston Housing Regression Advanced") \

.getOrCreate()

# Cargar datos

housing\_df = spark.read.csv("HousingData.csv", header=True, inferSchema=True)

columns = housing\_df.columns

# Convertir todas las columnas a tipo Double

for col\_name in columns:

housing\_df = housing\_df.withColumn(col\_name, col(col\_name).cast('double'))

# Eliminar nulos

housing\_df = housing\_df.dropna()

# Seleccionar las características

feature\_columns = housing\_df.columns[:-1]

assembler = VectorAssembler(inputCols=feature\_columns, outputCol="raw\_features")

data = assembler.transform(housing\_df).select("raw\_features", "medv")

# Estandarizar

scaler = StandardScaler(inputCol="raw\_features", outputCol="features", withMean=True, withStd=True)

scaler\_model = scaler.fit(data)

scaled\_data = scaler\_model.transform(data).select("features", "medv")

# Dividir los datos en 3 conjuntos

train\_data, val\_data, test\_data = scaled\_data.randomSplit([0.6, 0.2, 0.2], seed=42)

# Crear modelo

rf = RandomForestRegressor(featuresCol='features', labelCol='medv')

# Definir una cuadrícula

param\_grid = ParamGridBuilder() \

.addGrid(rf.numTrees, [50, 100, 150]) \

.addGrid(rf.maxDepth, [5, 10, 15]) \

.build()

# evaluador

evaluator = RegressionEvaluator(labelCol='medv', predictionCol='prediction', metricName='rmse')

# Configurar la división

tvs = TrainValidationSplit(estimator=rf,

estimatorParamMaps=param\_grid,

evaluator=evaluator,

trainRatio=0.8)

# Entrenar el modelo

tvs\_model = tvs.fit(train\_data)

# Hacer predicciones

predictions = tvs\_model.transform(test\_data)

# Calcular métricas

rmse = evaluator.evaluate(predictions, {evaluator.metricName: "rmse"})

mae = evaluator.evaluate(predictions, {evaluator.metricName: "mae"})

r2 = evaluator.evaluate(predictions, {evaluator.metricName: "r2"})

print("Root Mean Squared Error (RMSE):", rmse)

print("Mean Absolute Error (MAE):", mae)

print("R-squared (R2):", r2)

# Detener la sesión de Spark

spark.stop()

Regresion lineal:

from pyspark.sql import SparkSession

from pyspark.ml.feature import VectorAssembler, StandardScaler

from pyspark.ml.regression import LinearRegression

from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit

from pyspark.sql.functions import col

spark = SparkSession.builder \

.appName("Boston Housing Linear Regression Advanced") \

.getOrCreate()

# Cargar datos

housing\_df = spark.read.csv("HousingData.csv", header=True, inferSchema=True)

columns = housing\_df.columns

# Convertir todas las columnas a tipo Double

for col\_name in columns:

housing\_df = housing\_df.withColumn(col\_name, col(col\_name).cast('double'))

# Eliminar nulos

housing\_df = housing\_df.dropna()

# Dividir los datos segun atributos y clase

feature\_columns = housing\_df.columns[:-1]

assembler = VectorAssembler(inputCols=feature\_columns, outputCol="raw\_features")

data = assembler.transform(housing\_df).select("raw\_features", "medv")

# estandarizar

scaler = StandardScaler(inputCol="raw\_features", outputCol="features", withMean=True, withStd=True)

scaler\_model = scaler.fit(data)

scaled\_data = scaler\_model.transform(data).select("features", "medv")

# Dividir los datos en 3 conjuntos

train\_data, val\_data, test\_data = scaled\_data.randomSplit([0.6, 0.2, 0.2], seed=42)

# Crear modelo

lr = LinearRegression(featuresCol='features', labelCol='medv')

# Definir cuadrícula

param\_grid = ParamGridBuilder() \

.addGrid(lr.regParam, [0.01, 0.1, 1.0]) \

.addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \

.build()

# evaluador

evaluator = RegressionEvaluator(labelCol='medv', predictionCol='prediction', metricName='rmse')

# división de entrenamiento y validación

tvs = TrainValidationSplit(estimator=lr,

estimatorParamMaps=param\_grid,

evaluator=evaluator,

trainRatio=0.8)

# Entrenar el modelo

tvs\_model = tvs.fit(train\_data)

# Hacer predicciones

predictions = tvs\_model.transform(test\_data)

# Calcular métricas

rmse = evaluator.evaluate(predictions, {evaluator.metricName: "rmse"})

mae = evaluator.evaluate(predictions, {evaluator.metricName: "mae"})

r2 = evaluator.evaluate(predictions, {evaluator.metricName: "r2"})

print("Root Mean Squared Error (RMSE):", rmse)

print("Mean Absolute Error (MAE):", mae)

print("R-squared (R2):", r2)

# Detener la sesión de Spark

spark.stop()

Regresion logistica:

from pyspark.sql import SparkSession

from pyspark.ml.feature import VectorAssembler

from pyspark.ml.classification import LogisticRegression

from pyspark.ml.evaluation import BinaryClassificationEvaluator

from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

from pyspark.ml.feature import StringIndexer

from pyspark.ml import Pipeline

spark = SparkSession.builder \

.appName("Titanic Logistic Regression with Cross Validation") \

.getOrCreate()

# Cargar datos

titanic\_df = spark.read.csv("titanic.csv", header=True, inferSchema=True)

#eliminar nulos

titanic\_df = titanic\_df.dropna(subset=['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked'])

# Convertir variables categóricas a numéricas

indexers = [

StringIndexer(inputCol=column, outputCol=column+"\_index").fit(titanic\_df)

for column in ["Sex", "Embarked"]

]

pipeline = Pipeline(stages=indexers)

titanic\_df = pipeline.fit(titanic\_df).transform(titanic\_df)

# Seleccionar las características para el modelo

feature\_columns = ['Pclass', 'Sex\_index', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked\_index']

assembler = VectorAssembler(inputCols=feature\_columns, outputCol="features")

data = assembler.transform(titanic\_df).select("features", "Survived")

# Dividir los datos en 3

train\_data, test\_data, val\_data = data.randomSplit([0.6, 0.2, 0.2], seed=42)

# Crear modelo

lr = LogisticRegression(featuresCol='features', labelCol='Survived')

# Definir cuadrícula

param\_grid = ParamGridBuilder() \

.addGrid(lr.regParam, [0.01, 0.1, 1.0]) \

.addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \

.build()

# evaluador

evaluator = BinaryClassificationEvaluator(labelCol='Survived')

# validación cruzada

crossval = CrossValidator(estimator=lr,

estimatorParamMaps=param\_grid,

evaluator=evaluator,

numFolds=5)

# Entrenar el modelo

cv\_model = crossval.fit(train\_data)

# predicciones

predictions = cv\_model.transform(test\_data)

# Evaluar el rendimiento

accuracy = evaluator.evaluate(predictions)

print("Accuracy:", accuracy)

# Detener la sesión

spark.stop()

SVM clasificacion:

# Importar las bibliotecas necesarias

from pyspark.sql import SparkSession

from pyspark.ml.feature import VectorAssembler, StringIndexer

from pyspark.ml.classification import LinearSVC

from pyspark.ml.evaluation import BinaryClassificationEvaluator

spark = SparkSession.builder \

.appName("Titanic SVM") \

.getOrCreate()

# Cargar datos

titanic\_df = spark.read.csv("titanic.csv", header=True, inferSchema=True)

# Seleccionar las columnas

selected\_columns = ['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked', 'Survived']

titanic\_df = titanic\_df.select(selected\_columns)

# Eliminar filas con nulos

titanic\_df = titanic\_df.dropna()

# Convertir la columna 'Sex' y 'Embarked' a numeros

indexers = [StringIndexer(inputCol=column, outputCol=column+"\_index").fit(titanic\_df) for column in ['Sex', 'Embarked']]

titanic\_indexed = titanic\_df

for indexer in indexers:

titanic\_indexed = indexer.transform(titanic\_indexed)

# ensamblador

assembler = VectorAssembler(inputCols=['Pclass', 'Sex\_index', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked\_index'], outputCol='features')

assembled\_data = assembler.transform(titanic\_indexed)

# Dividir los datos

(training\_data, test\_data) = assembled\_data.randomSplit([0.7, 0.3], seed=123)

# Entrenar

svm = LinearSVC(labelCol='Survived', featuresCol='features')

svm\_model = svm.fit(training\_data)

# Realizar predicciones

predictions = svm\_model.transform(test\_data)

# Evaluar el rendimiento

evaluator = BinaryClassificationEvaluator(labelCol='Survived')

accuracy = evaluator.evaluate(predictions)

# Mostrar la precisión

print("Accuracy:", accuracy)

# Detener la sesión de Spark

spark.stop()

GBT clasificacion:

# Importar las bibliotecas necesarias

from pyspark.sql import SparkSession

from pyspark.ml.feature import VectorAssembler, StringIndexer

from pyspark.ml.classification import GBTClassifier

from pyspark.ml.evaluation import BinaryClassificationEvaluator

from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit

spark = SparkSession.builder \

.appName("GBTClassifier Optimization") \

.getOrCreate()

# Cargar datos

titanic\_df = spark.read.csv("titanic.csv", header=True, inferSchema=True)

# Seleccionar las columnas

selected\_columns = ['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked', 'Survived']

titanic\_df = titanic\_df.select(selected\_columns)

# Eliminar filas con nulos

titanic\_df = titanic\_df.dropna()

# Convertir la columna con textos a numericas

indexers = [StringIndexer(inputCol=column, outputCol=column+"\_index").fit(titanic\_df) for column in ['Sex', 'Embarked']]

titanic\_indexed = titanic\_df

for indexer in indexers:

titanic\_indexed = indexer.transform(titanic\_indexed)

# Crear un ensamblador

assembler = VectorAssembler(inputCols=['Pclass', 'Sex\_index', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked\_index'], outputCol='features')

assembled\_data = assembler.transform(titanic\_indexed)

# Dividir el conjunto de datos en 3

(training\_data, validation\_data, test\_data) = assembled\_data.randomSplit([0.6, 0.2, 0.2], seed=123)

# Crear el modelo

gbt = GBTClassifier(labelCol='Survived', featuresCol='features')

# Definir la cuadrícula

param\_grid = ParamGridBuilder() \

.addGrid(gbt.maxDepth, [3, 5, 7]) \

.addGrid(gbt.maxIter, [10, 20, 30]) \

.build()

# Definir el evaluador

evaluator = BinaryClassificationEvaluator(labelCol='Survived')

# Definir el TrainValidationSplit

tvs = TrainValidationSplit(estimator=gbt,

estimatorParamMaps=param\_grid,

evaluator=evaluator,

trainRatio=0.8)

# Ajustar modelo

gbt\_model = tvs.fit(training\_data)

# Hacer predicciones

validation\_predictions = gbt\_model.transform(validation\_data)

# Calcular la precisión

validation\_accuracy = evaluator.evaluate(validation\_predictions)

print("Validation Accuracy:", validation\_accuracy)

# Hacer predicciones

test\_predictions = gbt\_model.transform(test\_data)

# Calcular la precisión

test\_accuracy = evaluator.evaluate(test\_predictions)

print("Test Accuracy:", test\_accuracy)

# Detener la sesión de Spark

spark.stop()