Apunts CE_5074 8.1

Iloc:	Institut d'Ensenyaments a Distància de les Illes Balears	Imprès per:	Carlos Sanchez Recio
Curs:	Sistemes de Big Data	Data:	dimarts, 22 d'abril 2025, 21:29
Llibre:	Apunts CE_5074 8.1		

Taula de continguts

1. Introducció

En el lliurament 7 del mòdul de Big Data Aplicat hem introduït el *framework* Apache Spark, que permet un processament distribuït i eficient de grans conjunts de dades. Allà hem introduït també els RDD o *Resilient Distributed Datasets* (conjunts de dades distribuïts resilients), com a estructura de dades bàsica per treballar amb Spark.

En aquest darrer lliurament del mòdul de Sistemes de Big Data integrarem tot el que hem anant treballant en els lliuraments anteriors d'aquest mòdul amb el que hem après sobre Apache Spark a Big Data Aplicat.

Recordem que en el lliurament 6 vàrem introduir els conceptes d'analítica descriptiva (què ha passat o està passant?) i predictiva (què és el més probable que passi en el futur?).

En aquest lliurament, en primer lloc, veurem com fer una analítica descriptiva en un *framework* distribuït com Apache Spark, fent servir el mòdul **Spark SQL**. Ja sabem que l'anàlisi de dades està basat en els conceptes d'estadística que vàrem introduir en el lliurament 2 i que ara tornaran a aparèixer.

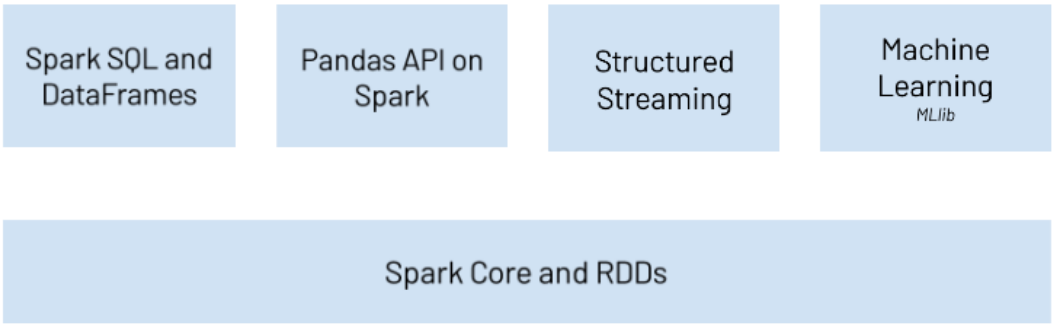
I en segon lloc, veurem com fer una analítica predictiva, treballant amb algorismes de *Machine Learning* sobre Apache Spark, fent servir el mòdul **MLlib**. En aquesta segona part ens trobarem també amb tècniques de *data wrangling* que vam introduir en el darrer lliurament, com per exemple el reescalat de dades.

Abans de tot això, però, veurem quins són els components principals de PySpark, entre els quals trobam Spark SQL i MLlib.

Per simplificar, en aquest lliurament ja no farem feina sobre el clúster que hem definit a Big Data Aplicat, sinó que treballarem amb quaderns Colab que s'executen en els servidors de Google.

2. Components de PySpark

Sobre el nucli de Spark (incloent l'API per treballar amb els RDD) s'han definit 4 grans mòduls, tal i com poden veure en la següent figura:



Imatge: *Components de PySpark. Font: spark.apache.org*

Spark SQL és el mòdul per fer feina amb dades estructurades. Està basant en el concepte de DataFrame, que ofereix un major nivell d'abstracció que els RDD. Un DataFrame és una representació de les dades en forma de taula, amb files i columnes amb nom. El subpaquet de PySpark per a SparkSQL és *pyspark.sql*.

Pandas API on Spark permet escalar els treballs fets amb la llibreria pandas a qualsevol mida, executant-se de manera distribuïda en múltiples nodes Spark. El seu objectiu és fer senzilla la transició de pandas a Spark. No obstant això, si no tenim ja una aplicació feta amb pandas, es recomana utilitzar directament Spark SQL i els DataFrames.

Structured Streaming és un motor per al processament de fluxos de dades (*streams*), escalable i tolerant a fallades, que s'executa sobre Spark. El subpaquet de PySpark corresponent és *pyspark.streaming*.

MLlib és la llibreria per a l'aprenentatge automàtic que s'executa sobre Apache Spark. MLlib incorpora multitud d'algorismes d'aprenentatge automàtic que s'executen de forma distribuïda i eficient sobre els nodes d'un clúster Spark. PySpark té dos subpaquets diferents per a MLlib: *pyspark.mllib*, que treballa sobre l'API de RDDs, i *pyspark.ml*, que permet treballar sobre l'API de DataFrames (Spark SQL).

En aquest lliurament ens centrarem en dos d'aquests mòduls: per una banda, Spark SQL i els DataFrames; i per l'altra MLlib i en concret, el subpaquet *pyspark.ml*.

3. Spark SQL i els DataFrame

En el mòdul de Big Data Aplicat hem vist en detall els RDD de Spark. Amb l'API de RDD podem carregar dades que estiguin en fitxers, però no podem treballar amb l'estructura (columnes i els seus tipus de dades) d'aquestes dades. És per això que necessitam utilitzar Spark SQL.

Spark SQL és un mòdul de Spark per al processament estructurat de dades. A diferència de l'API bàsica de Spark RDD, les interfícies proporcionades per Spark SQL proporcionen a Spark més informació sobre l'estructura de les dades i els càlculs que es realitzen sobre elles. Internament, Spark SQL utilitza aquesta informació per realitzar optimitzacions addicionals.

Hi ha diverses maneres d'interactuar amb Spark SQL. Les més importants són les API de Dataset i DataFrame.

Un **Dataset** és una col·lecció distribuïda de dades. Dataset és una interfície introduïda en Spark 1.6 que proporciona els beneficis dels RDD, a més d'incorporar els beneficis del motor d'execució optimitzat de Spark SQL. Un dataset es pot construir a partir d'objectes Scala o Java i després es pot manipular mitjançant transformacions (*map*, *flatMap*, *filter*, etc.). L'API de Dataset està disponible en Scala i Java, però no en Python ni R.

Un **DataFrame** és un dataset organitzat en columnes amb nom. És conceptualment equivalent a una taula en una base de dades relacional o un dataframe de pandas, però amb optimitzacions per a la computació distribuïda. Els DataFrames es poden construir a partir d'una àmplia varietat de fonts com: arxius de dades estructurats (per exemple, CSV o JSON), taules en Hive, bases de dades externes (relacionals o NoSQL) o RDD ja existents. L'API DataFrame està disponible en Scala, Java, Python i R.

IMPORTANT

Quan es computa un resultat, s'utilitza el mateix motor d'execució, independentment de l'API o llenguatge de programació que s'hagi utilitzat per expressar el càlcul.

Aquesta unificació significa que els desenvolupadors poden canviar fàcilment entre diferents APIs, permetent al programador triar la manera més natural d'expressar una transformació donada.

Spark SQL va aparèixer amb la versió 2.0 de Spark, fonamentalment per resoldre algunes de les principals limitacions de Hive. Recordem que Hive també ens permet interactuar amb dades estructurades mitjançant HiveSQL, un llenguatge pràcticament idèntic a SQL. Però la principal limitació de Hive és que fa servir MapReduce com a motor d'execució, la qual cosa ja sabem que, tot i que permet l'execució distribuïda, introdueix importants retards en el seu rendiment. En conseqüència, Hive no suporta consultes en temps real ni operacions transaccionals.

Així doncs, Spark SQL s'executa en memòria d'una manera molt més ràpida, permetent consultes en temps real. Veurem, a més, que Spark SQL també permet executar queries SQL.

AMPLIACIÓ

Podeu trobar més detalls a l'apartat de Spark SQL de la documentació general de Spark: https://spark.apache.org/docs/latest/sql-programming-guide.html

ALERTA

El codi que anirem explicant en els següents subapartats el podeu trobar també en aquest quadern de Colab:

https://colab.research.google.com/drive/1pjsJVSuoFhrd-cY9KO_AIDgctSF-2voO

3.1. Punt d'entrada de l'API

En el mòdul de Big Data Aplicat vàrem veure que **SparkContext** és el punt d'entrada per treballar amb Spark. Entre d'altres coses, permet crear RDDs. Recordem que en el shell de pyspark tenim disponible directament la variable ***sc***, instància de SparkContext.

Spark SQL proporciona **SQLContext** com a punt d'entrada per a les funcionalitats de SQL i DataFrames. Entre d'altres, es pot emprar per crear nous DataFrames.

Es defineix així, a partir del SparkContext (*sc*):

```
sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

A partir del SQLContext, podem generar un DataFrame, per exemple a partir d'un fitxer JSON:

```
df = sqlContext.read.json("/tmp/persones.json")
```

O d'un CSV:

```
df = sqlContext.read.csv("/tmp/persones.csv")
```

Així doncs, si treballam amb RDDs, hauríem d'utilitzar SparkContext com a punt d'entrada, mentre que si treballam amb Spark SQL, utilitzaríem SQLContext. De la mateixa manera, l'API per a streaming (Spark Streaming) també té el seu punt d'entrada propi, StreamingContext. Per evitar haver de treballar amb múltiples punts d'entrada, Spark proporciona un punt d'entrada únic, que permet interactuar amb totes les funcionalitats de Spark i els seus mòduls, incloent l'API de DataFrame. Aquest és **SparkSession**.

En el shell de pyspark de versions recents, a més de *sc*, també es proporciona la variable ***spark***, instància de SparkSession (no és el cas de la màquina virtual de Cloudera Quickstart). En el cas de codi Python, ho referenciam mitjançant *builder* de la classe pyspark.sql.SparkSession:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder\
        .master("ubicació del màster") \
        .appName("nom de l'aplicació")\
        .config("opció de configuració", "valor")\
        .getOrCreate()
```

Amb *master* especificam quin és el node mestre al qual s'ha de connectar. Això pot ser "local", per executar-se localment, "local[4]", que també s'executaria localment, però especificant que s'emprin 4 cores, o també podem especificar la URL del node, per exemple: “spark://master:7077".

Amb *appName* donam un nom a l'aplicació, que serà el que es veurà en la interfície web de Spark.

Amb *config* especificam una opció de configuració (que es propagarà automàticament a l'objecte SparkConf.

Amb *getOrCreate*, feim que recuperi una SparkSession existent, o que si no existeix, en creï una de nova amb les opcions que especificam amb aquest *builder*.

Aquestes opcions són les més habituals (n'hi ha d'altres), però convé aclarir que no són obligatòries.

Vegem un exemple que emprarem sovint en els nostres quaderns de Google Colab:

```
spark = SparkSession.builder\
        .master("local")\
        .appName("Exemples amb Spark SQL")\
        .config('spark.ui.port', '4050')\
        .getOrCreate()
```

Una vegada tenim ja l'objecte *spark*, podem crear un dataframe, per exemple, a partir d'un arxiu JSON:

```
df = spark.read.json("/tmp/persones.json")
```

O d'un CSV:

```
df = spark.read.csv("/tmp/persones.csv")
```

En veurem més detalls sobre la creació de dataframes a continuació.

3.2. Crear un dataframe

Ja hem vist en el subapartat anterior que amb spark.read podem llegir un fitxer JSON o CSV i generar un dataframe.

A més d'aquests dos tipus de fitxers, Spark SQL permet treballar amb altres tipus de fonts de dades. Anam a veure els més habituals amb més detall.

Fitxers JSON

Suposem que tenim el següent fitxer *persones.json* en el directori *dades* de la nostra unitat de Google Drive:

```
{"id":1, "nom":"Joan", "edat":50}
{"id":2, "nom":"Aina", "edat":30}
{"id":3, "nom":"Pep", "edat":40}
```

Amb el mètode **spark.read.json()**, llegim el fitxer i generam el dataframe:

```
df1 = spark.read.json('/content/drive/MyDrive/dades/persones.json')
```

Vegem el contingut del dataframe:

```
df1.show()
```

Que retorna:

```
+----+---+----+
|edat| id| nom|
+----+---+----+
|  50|  1|Joan|
|  30|  2|Aina|
|  40|  3| Pep|
+----+---+----+
```

Vegem l'esquema que ha generat:

```
df1.printSchema()
```

Que retorna:

```
root
 |-- edat: long (nullable = true)
 |-- id: long (nullable = true)
 |-- nom: string (nullable = true)
```

ALERTA

Les columnes que conformen un DataFrame es mostren per ordre alfabètic. Per això, tot i que l'ordre dins el JSON era id, nom, edat, aquí ho veim amb l'ordre edat, id, nom.

Fitxers CSV

En el cas de fitxers CSV, emprarem **spark.read.csv()**. Podem especificar diversos paràmetres per fer la lectura. Per exemple, suposem que tenim aquest fitxer CSV, on hem utilitzat el tabulador com a separador de camps:

```
id      nom     edat
1       Joan    50
2       Aina    30
3       Pep     40
```

Per fer la lectura del fitxer i generar el dataframe de manera correcta, hem d'especificar algunes propietats: que tenim una primera línia de capçalera, que empram el tabulador com a separador de camps (per defecte n'utilitza la coma) i que volem que inferesqui els tipus de dades a partir dels valors. Aquestes propietats les especificam mitjançant el mètode *option()*, passant-li com a arguments el nom de la propietat i el seu valor. Així doncs, la següent sentència llegeix el fitxer i genera el dataframe:

```
df2 = spark.read\
  .option("header", True)\
  .option("sep", "\t")\
  .option("inferSchema", True)\
  .csv('/content/drive/MyDrive/dades/persones.csv')
```

En lloc de *sep*, podem emprar *delimiter*, són sinònims.

Podem veure que tant el contingut com l'esquema són iguals als que hem obtingut a partir del fitxer JSON. Només canvia l'ordre en què apareixen les columnes:

`df2.show()`

Retorna:

```
+---+----+----+
| id| nom|edat|
+---+----+----+
|  1|Joan|  50|
|  2|Aina|  30|
|  3| Pep|  40|
+---+----+----+
```

`df2.printSchema()`

Retorna:

```
root
 |-- id: integer (nullable = true)
 |-- nom: string (nullable = true)
 |-- edat: integer (nullable = true)
```

I què passa si el fitxer CSV no té capçalera i per tant no podem inferir el nom dels camps? Vegem-ne un exemple, on carregam el fitxer persones_sense_header.csv amb el següent contingut:

```
1       Joan    50
2       Aina    30
3       Pep     40
```

Si executam

```
df2 = spark.read\
  .option("header", False)\
  .option("sep", "\t")\
  .csv('/content/drive/MyDrive/dades/persones_sense_header.csv')
df2.show()
```

Veurem que fa servir *_c0*, *_c1* i *_c2* com a nom de les columnes:

```
+---+----+---+
|_c0| _c1|_c2|
+---+----+---+
|  1|Joan| 50|
|  2|Aina| 30|
|  3| Pep| 40|
+---+----+---+
```

Ara podem assignar-li el nom a les columnes emprant el mètode *toDF()*:

```
df2 = df2.toDF('id', 'nom', 'edat')
df2.show()
```

Retorna:

```
+---+----+----+
| id| nom|edat|
+---+----+----+
|  1|Joan|  50|
|  2|Aina|  30|
|  3| Pep|  40|
+---+----+----+
```

Una altra manera de fer-ho és definint explícitament un esquema, on també podem especificar els tipus de dades de cada columna:

```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
esquema = StructType([
    StructField("id", IntegerType(), True),
    StructField("nom", StringType(), True),
    StructField("edat", IntegerType(), True)])
df2 = spark.read\
  .option("header", False)\
  .option("sep", "\t")\
  .csv('/content/drive/MyDrive/dades/persones_sense_header.csv', schema=esquema)
df2.show()
```

El resultat és el mateix d'abans:

```
+---+----+----+
| id| nom|edat|
+---+----+----+
|  1|Joan|  50|
|  2|Aina|  30|
|  3| Pep|  40|
+---+----+----+
```

Taula d'una base de dades relacional

Fent servir **sqlContext.read.load()** podem fer càrregues de dades a un dataframe de forma genèrica. Això ens permet llegir un fitxer Parquet, una taula Hive, o el que ens interessa ara, una taula d'una base de dades relacional fent servir JDBC. Suposem que volem llegir la taula "taula" de la base de dades "bd", que està en el servidor MySQL "servidor", al qual ens connectam amb usuari "usuari" i contrasenya "contrasenya".

Primer de tot, hem d'incloure el driver JDBC per al nostre gestor de bases de dades (en aquest cas MySQL) en el classpath de Spark:

```
SPARK_CLASSPATH=mysql-connector-java-5.1.35-bin.jar bin/spark-shell
```

I ara ja podem fer la lectura, mitjançant *sqlContext.read.load()*, especificant que el format és JDBC (amb el mètode *format()*) i configurant les propietats de connexió (amb el mètode *option()*), de la següent manera:

```
val df3 = spark.read
    .format("jdbc")
    .option("driver","com.mysql.jdbc.Driver")
    .option("url", "jdbc:mysql://servidor:3306/bd")
    .option("dbtable", "taula")
    .option("user", "usuari")
    .option("password", "contrasenya")
    .load()
```

3.3. Interoperar amb un RDD

Spark SQL pot convertir un RDD en un DataFrame. Per fer-ho, el RDD ha de contenir objectes de tipus Row, que permetran a Spark SQL inferir l'esquema del DataFrame.

Vegem com es fa a partir d'un exemple. Tenim el fitxer persones.txt al directori *dades* de la nostra unitat de Google Drive, amb aquest contingut:

```
1, Joan, 50
2, Aina, 30
3, Pep, 40
```

Generarem un RDD a partir d'aquest fitxer de text, tal i com vàrem veure al mòdul de Big Data Aplicat. Recordem que quan llegim un fitxer de text i generam un RDD, es fa línia a línia, i no té en compte cap estructura interna de columnes.

```
from pyspark import SparkContext
sc = SparkContext.getOrCreate()
rdd = sc.textFile("/content/drive/MyDrive/dades/persones.txt")
rdd.collect()
```

```
Veim que el contingut de rdd és:
```

```
['1, Joan, 50', '2, Aina, 30', '3, Pep, 40']
```

El que volem saber en aquest subapartat és com convertir aquest RDD en un DataFrame. Primer, aplicam una transformació *map* per separar cada una de les files per comes i obtenim un RDD nou al qual anomenam *parts*. A continuació, aplicam una altra transformació *map* per convertir cada una d'aquestes files en un objecte pyspark.sql.Row, on hem d'especificar les tres columnes (*id*, *nom* i *edat*) de la fila, i obtenim un nou RDD al qual anomenam *persones*:

```
from pyspark.sql import Row
parts = rdd.map(lambda a: a.split(","))
persones = parts.map(lambda p: Row(id=int(p[0]), nom=p[1], edat=int(p[2])))
persones.collect()
```

Veim que el contingut de *persones* és:

```
[Row(id=1, nom=' Joan', edat=50),
 Row(id=2, nom=' Aina', edat=30),
 Row(id=3, nom=' Pep', edat=40)]
```

I ara, a partir del RDD *persones*, ja podem crear un nou DataFrame mitjançant el mètode *createDataFrame* de *SQLContext* (també ho podríem fer amb el mateix mètode de l'objecte SparkSession):

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
df = sqlContext.createDataFrame(persones)
df.show()
```

El contingut del dataframe *df* és:

```
+---+-----+----+
| id|  nom|edat|
+---+-----+----+
|  1| Joan|  50|
|  2| Aina|  30|
|  3|  Pep|  40|
+---+-----+----+
```

3.4. Operacions sobre dataframes

L'API de Spark SQL ens permet interactuar amb les dades dels dataframes per fer una anàlisi de dades.

Ja hem vist alguns dels mètodes més habituals i senzills de DataFrame com són *show()* i *printSchema()*. Però en tenim molts més. Podeu trobar la referència completa de la classe DataFrame: https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html

De tots els mètodes, els més emprats són els que ens permeten recuperar dades (*select*), filtrar (*filter*), ordenar (*sort*) i agrupar (*groupBy*). Un altre mètode molt habitual és el que ens permet fer joins entre dos dataframes (*join*).

Vegem com funcionen amb un exemple, on treballarem amb el nostre fitxer JSON de persones, al qual hem afegit algunes files més per poder fer algunes consultes:

```
{"id":1, "nom":"Joan", "edat":50}
{"id":2, "nom":"Aina", "edat":30}
{"id":3, "nom":"Pep", "edat":40}
{"id":4, "nom":"Marga", "edat":35}
{"id":5, "nom":"Cati", "edat":50}
{"id":6, "nom":"Toni", "edat":30}
{"id":7, "nom":"Tomeu", "edat":25}
{"id":8, "nom":"Pau", "edat":35}
{"id":9, "nom":"Maria", "edat":45}
{"id":10, "nom":"Pere", "edat":30}
```

Generam el DataFrame *df* a partir d'aquest fitxer (en el directori *dades* de la nostra unitat de Google Drive).

```
df = spark.read.json('/content/drive/MyDrive/dades/persones2.json')
```

select()

El mètode **select()** retorna un nou DataFrame amb els valors d'una una o diverses columnes del DataFrame original.

Per exemple, *df.select("nom")* recupera el valor de la columna *nom* de totes les files de *df*. Li aplicam després el mètode *show()* per veure el resultat:

```
df.select("nom").show()
```

Que dona com a resultat:

```
+-----+
|  nom|
+-----+
| Joan|
| Aina|
|  Pep|
|Marga|
| Cati|
| Toni|
|Tomeu|
|  Pau|
|Maria|
| Pere|
+-----+
```

Podem seleccionar més d'una columna, per exemple:

```
df.select("nom", "edat").show()
```

```
+-----+----+
|  nom|edat|
+-----+----+
| Joan|  50|
| Aina|  30|
|  Pep|  40|
|Marga|  35|
| Cati|  50|
| Toni|  30|
|Tomeu|  25|
|  Pau|  35|
|Maria|  45|
| Pere|  30|
+-----+----+
```

I també podem emprar * per referenciar a totes les columnes:

```
df.select("*").show()

+----+---+-----+
|edat| id|  nom|
+----+---+-----+
|  50|  1| Joan|
|  30|  2| Aina|
|  40|  3|  Pep|
|  35|  4|Marga|
|  50|  5| Cati|
|  30|  6| Toni|
|  25|  7|Tomeu|
|  35|  8|  Pau|
|  45|  9|Maria|
|  30| 10| Pere|
+----+---+-----+
```

En Spark SQL tenim diferents maneres de fer referència a una columna. Una és directament amb el seu nom, com en l'exemple anterior. I una altra és mitjançant un objecte *Column*. Això ho podem escriure de les següents dues maneres, ambdues donen el mateix resultat que *df.select("nom").show()*:

```
df.select(df['nom']).show()
```

```
df.select(df.nom).show()
```

filter()

El mètode **filter()** retorna un nou DataFrame només amb les files que satisfan la condició que se li especifica com a argument. Aquí tenim dues maneres de referenciar una condició. Una és mitjançant un string. Per exemple:

```
df.filter("edat = 30").show()
```

Retorna les files que tenen una edat de 30:

```
+----+---+----+
|edat| id| nom|
+----+---+----+
|  30|  2|Aina|
|  30|  6|Toni|
|  30| 10|Pere|
+----+---+----+
```

ALERTA

where() es un alias de *filter()*, es poden utilitzar indistintament. Podríem haver escrit:

```
df.where("edat = 30").show()
```

Però també podem emprar objectes *Column*. Seria de qualsevol d'aquestes tres maneres:

```
df.filter(df.edat == 30).show()
df.filter(df['edat'] == 30).show()
```

```
from pyspark.sql.functions import col
df.filter(col("edat") == 30).show()
```

Notau que empram l'operador == per a comparar si és igual que un valor (!= seria per a comparar si és diferent)

També podem emprar operadors lògics (*and*, *or* i *not*) en les condicions. Per exemple, per recuperar les files amb edat 30 o 40 seria:

```
df.filter("edat = 30 or edat=40").show()
```

O en la notació utilitzant objectes Column, amb qualsevol d'aquestes tres sintaxis:

```
df.filter( (df.edat == 30) | (df.edat == 40) ).show()
df.filter( (df["edat"] == 30) | (df["edat"] == 40) ).show()
df.filter( (col("edat") == 30) | (col("edat") == 40) ).show()
```

Notau que amb aquesta sintaxi els operadors lògics s'escriuen diferent: / per *or* i *&* per *and*.

En lloc d'operadors de comparació, també es poden utilitzar altres funcions en les condicions. Per exemple, la següent sentència retorna les files tals que el seu nom comença per la lletra T:

```
df.filter( df.nom.startswith('T')).show()

+----+---+-----+
|edat| id|  nom|
+----+---+-----+
|  30|  6| Toni|
|  25|  7|Tomeu|
+----+---+-----+
```

```

```

---

sort()

El mètode **sort()** retorna un nou DataFrame ordenat per la columna (o columnes) especificada.

Per exemple, podem ordenar el nostre DataFrame per la columna nom:

```
df.sort(df.nom).show()
```

Que retorna:

```
+----+---+-----+
|edat| id|  nom|
+----+---+-----+
|  30|  2| Aina|
|  50|  5| Cati|
|  50|  1| Joan|
|  35|  4|Marga|
|  45|  9|Maria|
|  35|  8|  Pau|
|  40|  3|  Pep|
|  30| 10| Pere|
|  25|  7|Tomeu|
|  30|  6| Toni|
+----+---+-----+
```

ALERTA

orderBy() és un alias de *sort()*, es poden utilitzar indistintament. Podríem haver escrit:

```
df.orderBy(df.nom).show()
```

Si ho volem ordenar en ordre descendent, aplicam el mètode *desc()* a la columna:

```
df.sort(df.nom.desc()).show()
```

Que retorna:

```
+----+---+-----+
|edat| id|  nom|
+----+---+-----+
|  30|  6| Toni|
|  25|  7|Tomeu|
|  30| 10| Pere|
|  40|  3|  Pep|
|  35|  8|  Pau|
|  45|  9|Maria|
|  35|  4|Marga|
|  50|  1| Joan|
|  50|  5| Cati|
|  30|  2| Aina|
+----+---+-----+
```

Podem emprar altres notacions per a les columnes, per exemple amb un string. I també podem especificar que l'ordre sigui descendent mitjançant l'opció *ascending=False*. Aquesta sentència retorna el mateix que l'anterior:

```
df.sort("nom", ascending=False).show()
```

groupBy()

El mètode **groupBy()** agrupa les dades emprant una columna (o múltiples), de manera que després puguem utilitzar funcions d'agregració sobre aquests grups.

Per exemple, anam a agrupar les dades per edat i després comptar quants elements tenim en cada grup:

```
df.groupBy("edat").count().show()
```

El resultat és:

```
+----+-----+
|edat|count|
+----+-----+
|  50|    2|
|  25|    1|
|  35|    2|
|  30|    3|
|  40|    1|
|  45|    1|
+----+-----+
```

Podeu veure les diferents funcions d'agregació disponibles en Spark SQL:
https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.html

join()

El mètode **join()** permet fer un *join* entre dos dataframes. Funciona igual que un *join* entre dues taules d'una base de dades relacional. El mètode té 3 arguments:

- el dataframe amb el qual feim el *join* (el costat dret del *join*)
- la condició de *join* (opcional)
- el mètode de *join* (opcional). Per defecte és *inner*. Altres possibles valors són: *cross, outer, full, fullouter, full_outer, left, leftouter, left_outer, right, rightouter, right_outer, semi, leftsemi, left_semi, anti, leftanti* i *left_anti*.

Vegem un exemple. Tenim els següents dos fitxers JSON d'empleats i departaments en el directori *dades* de la nostra unitat de Google Drive.

empleats.json:

```
{"id":1, "nom":"Joan", "departament":1}
{"id":2, "nom":"Aina", "departament":1}
{"id":3, "nom":"Pep", "departament":2}
```

departaments.json:

```
{"id":1, "nom":"Desenvolupament"}
{"id":2, "nom":"Sistemes"}
```

Generam els dos dataframes:

```
dfemp = spark.read.json('/content/drive/MyDrive/dades/empleats.json')
dfdep = spark.read.json('/content/drive/MyDrive/dades/departament.json')
```

I ara feim el *join* (intern, que és l'opció per defecte):

```
df = dfemp.join(dfdep, dfemp.departament == dfdep.id)
```

```
df.show()
```

Que retorna:

```
+-----------+---+----+---+---------------+
|departament| id| nom| id|            nom|
+-----------+---+----+---+---------------+
|          1|  1|Joan|  1|Desenvolupament|
|          1|  2|Aina|  1|Desenvolupament|
|          2|  3| Pep|  2|       Sistemes|
+-----------+---+----+---+---------------+
```

3.5. Executar sentències SQL

Una de les característiques més importants de Spark SQL és que ens permet tractar un dataframe com si fos una vista d'una base de dades i executar sentències SQL sobre ella.

Tenim el nostre dataframe que hem generat a partir del fitxer persones.csv que ja hem utilitzat anteriorment:

```
df = spark.read\
  .option("header", True)\
  .option("sep", "\t")\
  .option("inferSchema", True)\
  .csv('/content/drive/MyDrive/dades/persones.csv')
```

Per poder emprar el dataframe en sentències SQL hem de crear una vista temporal a partir de les dades del dataframe. A aquesta vista li assignam un nom (*persones* en l'exemple), que podrem emprar posteriorment per escriure i executar queries sobre les dades.

```
df.createOrReplaceTempView("persones")
```

I ara ja podem executar queries SQL, amb el mètode *sql* de l'objecte *SparkSession*. Per exemple, recuperar les persones de més de 35 anys:

```
majors35 = spark.sql("SELECT nom FROM persones WHERE edat > 35")
```

Veim que això retorna un nou dataframe *majors35*. Vegem el contingut:

```
majors35.show()
```

Que retorna:

```
+----+
| nom|
+----+
|Joan|
| Pep|
+----+
```

4. MLlib

MLlib és la llibreria d'Apache Spark per a l'aprenentatge automàtic (*machine learning, ML*) escalable.

En el mòdul de Sistemes d'aprenentatge automàtic es van introduir els principals models de l'aprenentatge no supervisat (principalment clustering o clusterització) i de l'aprenentatge supervisat (classificació i regressió). Tant allà com al mòdul de Programació d'intel·ligència artificial, s'ha vist també com implementar aquests models amb la llibreria scikit-learn. Per què ara veure'n una altra, de llibreria? Perquè MLlib s'executa sobre Apache Spark i permet un processament distribuït eficient, un factor fonamental quan treballam amb dades massives.

Les possibilitats d'aplicar algorismes de ML en un context de dades massives són enormes ja que aquests grans conjunts de dades són la base per a un bon entrenament dels models de ML. Alguns exemples són:

1. En l'àmbit bancari, models de ML s'entrenen per detectar operacions fraudulentes amb targetes de crèdit.
2. En l'àmbit dels vehicles autònoms, s'utilitza ML per a identificar objectes al voltant del vehicle, calcular les distàncies amb aquests objectes, identificar senyals de trànsit, etc.
3. En l'àmbit dels *smartphones*, trobam multitud d'aplicacions de ML: des de l'assistent de veu als algorismes utilitzats per la càmera de fotos per obtenir bones imatges en situacions difícils de llum.
4. En l'àmbit de la salut, un exemple és l'anomenada diagnosi assistida per ordinador (*Computer-Assisted Diagnosis*, CAD), que per exemple permet predir la possibilitat d'un càncer de pit a partir d'una mamografia.

Segons la pròpia documentació de MLlib, aquesta llibreria proporciona:

- Algorismes de ML: algorismes d'aprenentatge comuns com la classificació, regressió, clustering o sistemes de recomació.
- Caracterització: extracció de característiques (*features*), transformació, reducció de dimensionalitat i selecció
- *Pipelines*: eines per construir, avaluar i afinar els motors ML
- Persistència: guardar i carregar algorismes, models i motors
- Utilitats: àlgebra lineal, estadístiques, gestió de dades, etc.

Per altra banda, recordem que Spark té dues API per fer feina amb els conjunts de dades: RDD i DataFrame (Spark SQL). MLlib suporta les dues, tot i que la principal, des d'Apache Spark 2.0, és la de DataFrame. D'aquesta manera, tenim dues API diferents de MLlib:

- *spark.ml* per fer feina amb l'API de DataFrame
- *spark.mllib* per fer feina amb l'API de RDD

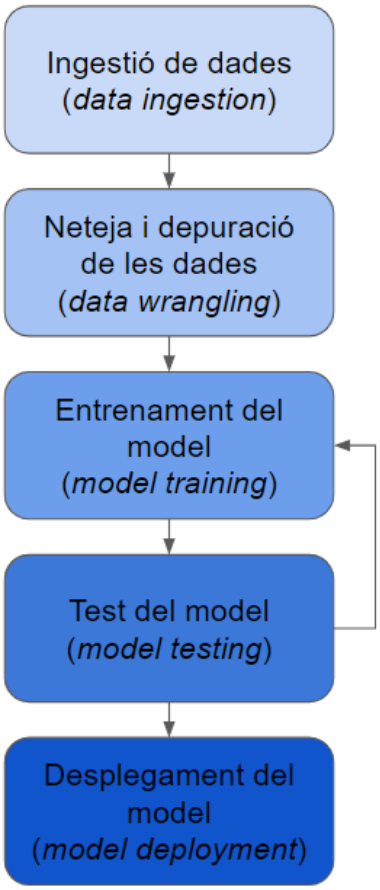
Tot i que no és el seu nom oficial, sovint es xerra de "Spark ML" per referir-se a l'API *spark.ml*, la que fa feina amb DataFrames.

Podeu trobar una guia molt completa sobre MLlib i els diferents algorismes d'aprenentatge que suporta a la documentació oficial:

https://spark.apache.org/docs/latest/ml-guide.html

4.1. Fases del procés de ML

Un procés d'aprenentatge automàtic consta de diverses fases:



Imatge: *Fases d'un procés d'aprenentatge automàtic*

Primer s'obtenen les dades, que poden provenir de diverses fonts tals com fitxers, bases de dades, fluxos en temps real, etc.

En segon lloc es netegen les dades aplicant les tècniques de *data wrangling* que vàrem veure en el lliurament anterior: detecció de valors absents, detecció d'outliers, reescalat de les dades, etc. Un punt important aquí és la caracterització de les dades: els algorismes de ML fan feina amb una única columna que conté totes les característiques que es consideren rellevants per al model. Així que sempre haurem de crear una columna que contengui tots els valors de les columnes que vulguem utilitzar en el model.

I ara entram ja en el que pròpiament és *Machine Learning*: un procés cíclic en què s'entrena un model i s'avalua. Això es pot fer en repetides ocasions fins que obtenim els paràmetres que consideram adequats per a un bon model. Una vegada que tenim el model definitiu, aquest es desplega i es pot fer servir amb noves dades.

4.2. Aprenentatge no supervisat: clustering

Recordem que en l'aprenentatge no supervisat no tenim disponibles les etiquetes de classes (les sortides desitjades) en les dades d'entrenament. Aquest tipus d'aprenentatge s'utilitza fonamentalment per a ***clustering***, és a dir, trobar agrupacions en les dades, de manera que les dades de cada grup o clúster siguin semblants entre sí (i més semblants que amb les dades dels altres clústers).

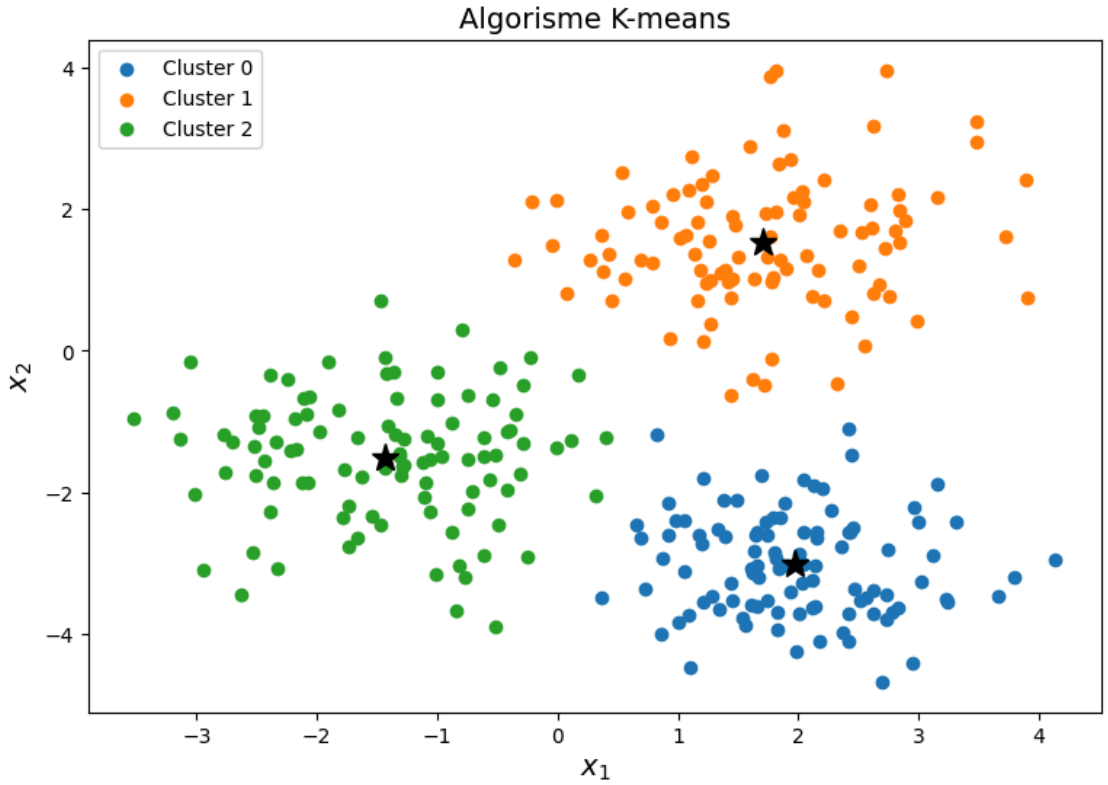
L'algorisme més utilitzat per a *clustering* és el de les K-mitjanes o ***K-means***. Els detalls els podeu trobar en el mòdul de Sistemes d'aprenentatge automàtic. Aquí veurem una aplicació amb un cas real, que podeu trobar en el següent quadern de Colab: **https://colab.research.google.com/drive/16FLDpt6ZmmztT2acxxc708dxW62_veuv**. Podeu descarregar l'arxiu CSV amb les dades que farem servir des de https://raw.githubusercontent.com/tnavarrete-iedib/bigdata-24-25/refs/heads/main/dades_cluster.csv.

Veureu que, una vegada configurat l'entorn de Spark, construïm el vector de característiques (fase de caracterització) i obtenim una nova columna *features*. Seguidament, feim un reescalat de les dades d'aquesta columna i n'obtenim una altra *standardized*, que és la que emprarem per al model.

A continuació ajustam el model amb les nostres dades. Emprarem l'algorisme K-means, que està implementat mitjançant la classe *pyspark.ml.clustering.KMeans*, i en aquest cas farem servir *k=3*, és a dir, volem obtenir 3 clústers. Obtindrem una predicció d'a quin clúster pertany cada fila: tenim un nou DataFrame *predictions* amb una columna *prediction* que val 0, 1 o 2, indicant quin a quin clúster pertany.

Després d'aplicar un algorisme de ML és convenient avaluar els resultats. En aquest cas, emprarem l'anomenat coeficient Silhouette, que ens dona una indicació de com de bé estan distribuïts els clústers.

Per últim, acabarem representant gràficament les dades amb colors diferents per als 3 clústers obtinguts.



Imatge: *Resultat de l'algorisme de clustering amb K-means*

4.3. Aprenentatge supervisat: classificació

Recordem que en l'aprenentatge supervisat tenim l'etiqueta de la classe a la que pertany cada instància (fila de les dades). És a dir, sabem quina és la sortida esperada de l'algorisme per a cada instància, la qual cosa ens permet entrenar el nostre model i, en el *testing*, comparar els resultats obtinguts amb els esperats.

Recordem també que hi ha dos tipus d'aprenentatges supervisats: els problemes de classificació i els de regressió. En els de classificació, que són els que ara ens ocupen, volem classificar cada instància, és a dir, donar-li un valor discret que indiqui a quina classe pertany. Si el conjunt de possibles valors de l'etiqueta és 2, és a dir, només hi ha dues possibles classes, xerram de classificació binomial. En canvi, si el número és major, xerram de classificació multinomial.

Hi ha diversos models de classificació, cadascun amb els seus avantatges i inconvenients. Podeu trobar els detalls dels més importants al mòdul de Sistemes d'aprenentatge automàtic.

Aquí el que veurem és l'aplicació a un problema real per a la detecció de bitllets falsos. Farem feina amb dues classes (classificació binomial): 0 per indicar que no és fals i 1 que sí. Podeu descarregar l'arxiu CSV amb les dades des de https://raw.githubusercontent.com/tnavarrete-iedib/bigdata-24-25/refs/heads/main/banknote_authentication.csv.

Veurem dues implementacions amb dos models diferents:

- Classificació amb Regressió logística: **https://colab.research.google.com/drive/1tIcoHL5NNHpyfsOYb11vzOt-ZNJS7Asm**
- Classificació mitjançant Random forest: **https://colab.research.google.com/drive/1NRXL6YvpF62O_EHoDK4eADPEWmbHCCIT**

Les primeres fases són semblants a les que hem vist en el cas de clustering: preparam l'entorn de Spark, carregam les dades, les caracteritzam i reescalam.

Una vegada fet això, dividim el conjunt de dades en dos: una part l'emprarem per a l'entrenament del model (70%) i l'altra per al *testing* (30%).

En el cas de la regressió logística, empram la classe *pyspark.ml.classification.LogisticRegression* per al nostre model. En el cas de Random forest, emprarem *pyspark.ml.classification.RandomForestClassifier*.

En el segon cas, hem afegit una passa adicional per trobar quins són els millors valors d'alguns hiperparàmetres del model (la profunditat i el número d'arbres). Empram la tècnica de *cross-validation* (classe *pyspark.ml.tuning.CrossValidator*).

Una vegada que hem entrenat el model, obtenim les prediccions: tendrem un nou DataFrame *pred*, amb una columna *prediction* que indica si la instància pertany a la classe 0 o a la 1.

Com sempre, per acabar, avaluarem el model. En aquest cas, ho feim obtenint l'exactitud (*accuracy*) i la matriu de confusió.

4.4. Aprenentatge supervisat: regressió

Recordem que la regressió és un tipus d'aprenentatge supervisat, on el que volem és predir el valor d'una determinada variable de sortida (*target*) per a cada instància. En aquest cas, a diferència de la classificació, aquesta variable no és discreta, sinó que té un domini continu (per exemple, un número real). Com que és supervisat, tenim un conjunt de dades on coneixem quin és el valor real d'aquesta variable. El que volem fer és entrenar el model amb les dades d'entrenament i després, per provar el model, compararem els valors obtinguts amb els reals.

També hi ha diversos models de regressió, cadascun amb els seus avantatges i inconvenients. Podeu trobar els detalls dels més importants al mòdul de Sistemes d'aprenentatge automàtic.

Aquí el que veurem és l'aplicació a un problema real. Tenim unes dades del lloguer de bicicletes a una ciutat i suposam que el número de bicicletes llogades depèn de les condicions meteorològiques de cada dia: els dies amb millor temps, es lloguen més bicis. Així doncs, tenim tres variables d'entrada: temperatura, humitat i velocitat del vent. I la variable de sortida és el número de lloguers. Podeu descarregar l'arxiu CSV des de https://raw.githubusercontent.com/tnavarrete-iedib/bigdata-24-25/refs/heads/main/bikes.csv.

Veurem la implementació fent servir dos models diferents:

- Regressió lineal: **https://colab.research.google.com/drive/1-rmrvCiG1DBIyAjRkjDOujF2XW3aCRfC**
- Regressió amb Random forest: **https://colab.research.google.com/drive/1hshTDja56j1TdrdLDwRsieYctStcDTGO**

Les primeres fases són semblants a les que hem vist en el cas de clustering i classificació: preparam l'entorn de Spark, carregam les dades, les caracteritzam i reescalam.

Una vegada fet això, dividim el conjunt de dades en dos: una part l'emprarem per a l'entrenament del model (les dades del primer any) i l'altra per al testing (segon any).

En el cas de la regressió lineal, empram la classe *pyspark.ml.regression.LinearRegression* per al nostre model. En el cas de Random forest, emprarem *pyspark.ml.regression.RandomForestRegressor*.

Igual que hem fet amb la classificació, per al Random forest, hem afegit una passa adicional per trobar quins són els millors valors d'alguns hiperparàmetres del model (la profunditat i el número d'arbres). Empram la tècnica de cross-validation (classe *pyspark.ml.tuning.CrossValidator*).

Una vegada que hem entrenat el model, obtenim les prediccions: tendrem un nou DataFrame *pred*, amb una columna *prediction* amb el número de bicis que s'estima que es llogaran segons les variables d'entrada.

Com sempre, després hem d'avaluar el model. En aquest cas, emprarem diverses mètriques: error quadràtic mitjà (RMSE), error absolut mitjà (MAE) i coeficient de determinació (R^2).

Acabam representant gràficament els valors reals i els de la predicció que hem obtingut amb el nostre model.