FUNDAMENTALS OF BIG DATA

3 V's → Volume, Velocity, Variety

Big Data processing systems → Hadoop/MapReduce { - OPEN SOURCE, - BATCH PROCESSING } / Apache Spark { - OPEN SOURCE, - BATCH AND REAL-TIME PROCESSING }

Features of Spark { - Distributed cluster computing framework, - Efficient in memory computations, - Fast data processing }

Modos de deploy → local Mode / Cluster Mode

Spark está escrito en scala.

SparkContext (sc) { - SparkContext es un entry point a Spark, - Un entry point es una manera de conectarse a un cluster de spark. }

↓ Atributos importantes:

- Version → `sc.version`
- Python Version → `sc.pythonVer`
- Master (es la URL del cluster o un string local si corre en local) → `sc.master`
- Para cargar datos: `sc.parallelize([1,2,3...])` / `sc.textFile('text.txt')` → Se le puede añadir un arg `"minPartitions=numero"` ⇒ Define el número mínimo de particiones

Spark RDD (Resilient distributed dataset)

Para ver el Nº de particiones de un RDD → `numRDD.getNumPartitions()`

Transformaciones (nuevo RDD) {
- map() → crea un nuevo RDD con el resultado ⇒ `rdd = sc.parallelize([...])` / `rdd.map = rdd.map(lambda x: x**2)`
- filter() → `rdd.filter = rdd.filter(lambda x: x>2)`
- flatmap() → retorna multiples elementos por cada elemento del RDD original ⇒ `rdd = sc.pa... ('hola yo', 'tal cosa')` / `rdd.flatmap = rdd.flatMap(lambda x: x.split(' '))` → `['hola','yo','tal','cosa']`
- union() → une 2 rdds → `combinado = rdd.a.union(rdd.2)`
}

Acciones (se edita el RDD) {
- collect() → retorna todos los elementos del dataset como un array → `rdd.collect()`
- take(N) → primeros N elementos como un array. → `rdd.take(2)`
- first() → Retorna solo el 1 elemento → `rdd.first()`
- count() → Retorna el Nº de elementos que tiene el rdd → `rdd.count()`
}

Par RDD `'llave': 'valor'`

Se pueden crear con tuplas como data de entrada → `rdd_par = sc.parallelize([(A,1),(B,2)])`

Tienen las mismas transformaciones y acciones {
- reduceByKey() → combina valores con la misma key ⇒ `rdd_reduce = rdd.reduceByKey(lambda x,y: x+y)` (si tienen misma llave, se suman el valor)
- sortByKey() → ordena por llave
- groupByKey() → agrupa valores con la misma llave ⇒ `rdd_group = rdd.groupByKey()`
- join() → une 2 RDD por el valor de la llave ⇒ `rdd_join = rdd.a.join(rdd.2)`
}

FUNDAMENTALS OF BIG DATA

Advanced RDD actions

- reduce(func) → por ejemplo para sumar todos los valores → `rdd.reduce(lambda x,y: x+y)`
- saveAsTextFile() → guarda el RDD con cada partición como un archivo distinto → `rdd.saveAsTextFile('nombre')`
  - ↳ para guardarlo en 1 único archivo → `rdd.coalesce(1).saveAsTextFile('nombre')`

Para Pair RDD:
- countByKey() → cuenta los elementos por cada llave → lo RETORNA COMO DICT.
- collectAsMap() → retorna un diccionario con los valores {'key': valor}

Deben usarse si hay pocos datos ya que todos se cargan en la memoria.

PySpark DataFrames I

SparkSession es el entry point para interactuar con estos DataFrames → Hace lo que SparkContext para los RDD.

Para crear DataFrames:
- Desde un RDD → `df = spark.createDataFrame(rdd, schema = nombre_columnas)`
- Desde archivos →
  - csv → `df = spark.read.csv('archivo.csv')`
  - json → `df = spark.read.json('archivo.json')`
  - txt → `df = spark.read.text('archivo.txt')`

Pueden especificarse: header=True → si tiene como nombre columnas; inferSchema=True → asigna el tipo de dato a cada columna.

Operaciones:

- Transformaciones:
  - select(col) → extrae las columnas del df.
  - show() → muestra las primeras filas
  - filter(condicion) → filtra los valores.
  - groupby(col) → agrupa por una columna
  - count()
  - orderby(col) → Ordena por una columna
  - dropDuplicates() → Retorna un nuevo df sin duplicados
  - withColumnRenamed('Antiguo nombre', 'nuevo')
  - printSchema() → Muestra los tipos de datos que contiene
  - columns → Retorna las columnas del df.
  - describe() → Retorna las estadísticas de las columnas numéricas.
  - withColumn('nueva', calculo con vieja) → Crea una nueva columna a partir de otra.
  - drop() → Para eliminar una columna.
  - distinct() → Valores únicos
  - startswith() → Comprueba si empieza por un caracter.

Usando consultas SQL:
- Primero debemos crear una tabla temporal → `df.createOrReplaceTempView('tabla')`
- Para ejecutar un comando SQL → `df_ej = spark.sql('SELECT col1, col2 FROM tabla')` → El resultado es como un dataframe.

FUNDAMENTALS OF BIG DATA

Pyspark DataFrames II

Data visualization with df.

- Spark DataFrames no funciona con matplotlib, seaborn... hay 3 opciones → Convertir a pandas → `toPandas()`
  - Librería pyspark_dist_explore
  - HandySpark
- pyspark_dist_explore (mantiene la ejecución distribuida)
  - Para un histograma → `hist(df, bins=20, cols=['...'])`
  - Otras funciones → `distplot()`, `pandas_histogram()`
- HandySpark
  - Convertir a HandyDf → `h_df = df.toHandy()`
  - Crear un histograma → `h_df.hist()`

PySpark MLib ⇒ SOLO SOPORTA RDDS

Collaborative filtering → Produce recommendations — `from pyspark.mllib.recommendation import ALS`

Clustering → agrupa los datos con características similares — `from pyspark.mllib.clustering import KMeans`

Classification → identifica a que categoría pertenece. — `from pyspark.mllib.classification import LogisticRegressionWithLBFGS`

Collaborative filtering

- 2 enfoques: user-user ⇒ busca usuarios similares al usuario objetivo; item-item ⇒ " items " en los items del usuario obj.
- Componentes
  - Rating ⇒ es una clase que procede el RDD donde (user, product, rating)
    `from pyspark.mllib.recommendation import Rating`
    `r = Rating(user=..., product=..., rating=...)`
  - randomSplit() ⇒ para dividir en train y test — `train, test = rdd.randomSplit([0.6, 0.4])`
  - ALS (Alternating Least Squares) → `ratings = sc.parallelize([r1, r2, r3])` (objetos de la clase Rating); `model = ALS.train(ratings, rank=10, iterations=10)` (rank → Nº de características)
  - predictAll() ⇒ aplica los ratings → `unrated = sc.parallelize([(1,1), (1,2)])` (user ID, product ID); `pred = model.predictAll(unrated)`

Classification

- MLLib contiene 2 tipos de datos específicos
  - Vectores
    - Dense Vector ⇒ guarda los entradas en un array de floating point numbers — `dense = Vectors.dense([1.0, 2.0, ...])`
    - Sparse Vector ⇒ solo valores ≠0 y sus índices — `sparse = Vectors.sparse(4, {1:5.0, 3:5.5})` (size, índice:valor)
  - LabeledPoint → ejemplo de clase/etiqueta — `positive = LabeledPoint(1.0, [1.0, 0.0, 3.0])`
- HashingTF() ⇒ mapea las x a valores en el vector de x — `pyspark.mllib.feature import HashingTF`

```
frase = 'hola hola que'
words = frase.split(' ')
tf = HashingTF(10000)
tf.transform(words)
```

(número de features)

- logisticRegression ⇒

```
data = [LabeledData(...), ...]
rdd = sc.parallelize(data)
lrm = LogisticRegressionWithLBFGS.train(rdd)
lrm.predict(...)
```

FUNDAMENTALS OF BIG DATA

PySpark MLib

Clustering

- unsupervised learning task to organize a collection of data into groups

- K-means →

```
model = KMeans.train(rdd, k=2, maxIterations=10)   # Nº de clusters
model.clusterCenters
```

- Evaluación →

```
def error(point):
    center = model.centers[model.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))
error = rdd.map(lambda point: error(point)).reduce(lambda x,y: x+y)
```

- Visualizar el cluster →

```
data_df = spark.createDataFrame(rdd, schema=["col 1", ...]).toPandas()
centers_df = pd.DataFrame(model.clusterCenters, columns=["...", ...])
plt.scatter(data["col 1"], data["col 2"])
plt.scatter(centers["col 1"], centers["col 2"], color='red', marker='x')
```