Spark MLlib: Machine Learning Library

Librería de algoritmos paralelos de ML para datos masivos

- Algoritmos clásicos de machine learning: clasificación, regresión, clustering, filtrado colaborativo
- Otros algoritmos: extracción de características, transformación, reducción de dimensionalidad y selección
- Herramientas para construir, evaluar y ajustar pipelines de ML
- Otras utilidades: álgebra lineal, estadística, manejo de datos, etc.

Dos paquetes:

- spark.mllib: API original, basada en RDDs
  - En *mantenimiento*
- spark.ml: API de alto nivel, basada en DataFrames

Documentación y APIs:

- ML
  - Guia: http://spark.apache.org/docs/latest/ml-guide.html
  - API Python: https://spark.apache.org/docs/latest/api/python/pyspark.ml.html
  - API Scala: https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.package
- MLlib
  - Guia: http://spark.apache.org/docs/latest/mllib-guide.html, con APIs Python y Scala

Ejemplo

Usa el algoritmo de clustering KMeans para agrupar datos de vectores dispersos en dos clusters.

In [ ]:
```
from pyspark import SparkContext
from pyspark.sql import SparkSession
import os

# Elegir el máster de Spark dependiendo de si se ha definido la variable de entorno HADOOP_CONF_DIR o YARN_CONF
SPARK_MASTER: str = 'yarn' if 'HADOOP_CONF_DIR' in os.environ or 'YARN_CONF_DIR' in os.environ else 'local[*]'

# Creamos un objeto SparkSession (o lo obtenemos si ya está creado)
spark: SparkSession = SparkSession \
  .builder \
  .appName("Mi aplicacion") \
  .config("spark.rdd.compress", "true") \
  .config("spark.executor.memory", "3g") \
  .config("spark.driver.memory", "3g") \
  .master(SPARK_MASTER) \
  .getOrCreate()

sc: SparkContext = spark.sparkContext
```

In [ ]:
```
from pyspark.ml.linalg import SparseVector, Vectors

# Define un array de 4 vectores dispersos, de 3 elementos cada uno
sparseData: list[SparseVector] = [
     Vectors.sparse(3, {1: 1.2}),
     Vectors.sparse(3, {1: 1.1}),
     Vectors.sparse(3, {0: 0.9, 2: 1.0}),
     Vectors.sparse(3, {0: 1.0, 2: 1.1})
 ]

for i in range(4):
    print(sparseData[i].toArray())
```

In [ ]:
```
# Convierte el array en un DataFrame
from pyspark.sql.dataframe import DataFrame

dfSD: DataFrame = sc.parallelize([
  (1, sparseData[0]),
  (2, sparseData[1]),
  (3, sparseData[2]),
  (4, sparseData[3])
]).toDF(["fila", "características"])

dfSD.show()
```

In [ ]:
```
# Creamos un modelo KMeans sin entrenar, con 2 clusters
# para más opciones ver https://spark.apache.org/docs/latest/ml-clustering.html
# y https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.clustering.KMeans.html
from pyspark.ml.clustering import KMeans
```

```
kmeans: KMeans = KMeans()\
    .setInitMode("k-means||")\
    .setFeaturesCol("características")\
    .setPredictionCol("predicción")\
    .setK(2)\
    .setSeed(1)
```

In [ ]:
```
# Ajustamos el modelo al DataFrame anterior y mostramos los centros de los clusters
from pyspark.ml.clustering import KMeansModel

kmModel: KMeansModel = kmeans.fit(dfSD)
print("Centros de los clusters: {0}.".format(
    kmModel.clusterCenters()))
```

In [ ]:
```
# Vemos cómo el modelo clusteriza los datos del array anterior
predicciones: DataFrame = kmModel.transform(dfSD)
predicciones.show()

# Calcula el coste como la suma de la distancia al cuadrado entre los puntos de entrada
# y los centros de los clusters correspondientes
from pyspark.ml.evaluation import ClusteringEvaluator
evaluator = ClusteringEvaluator(featuresCol="características", predictionCol="predicción")
silhouette = evaluator.evaluate(predicciones)
print("Coste = {0}.".format(silhouette))
```

In [ ]:
```
# Probamos el modelo con otros puntos
dfTest: DataFrame = sc.parallelize([
  (1, Vectors.sparse(3, {0: 0.9, 1:1.0, 2: 1.0})),
  (2, Vectors.sparse(3, {1: 1.5, 2: 0.3}))
]).toDF(["fila", "características"])

predicciones: DataFrame = kmModel.transform(dfTest)
predicciones.show(truncate=False)

# Calcula el coste como la suma de la distancia al cuadrado entre los puntos de entrada
# y los centros de los clusters correspondientes
from pyspark.ml.evaluation import ClusteringEvaluator
evaluator = ClusteringEvaluator(featuresCol="características", predictionCol="predicción")
silhouette = evaluator.evaluate(predicciones)
print("Coste = {0}.".format(silhouette))
```

In [ ]:
```
# Salva el modelo en un directorio
kmModel.save("/tmp/kmModel")
```

In [ ]:
```
# Vuelve a cargar el modelo
sameModel: KMeansModel = KMeansModel.load("/tmp/kmModel")

predicciones: DataFrame = kmModel.transform(dfTest)
predicciones.show(truncate=False)

# Calcula el coste como la suma de la distancia al cuadrado entre los puntos de entrada
# y los centros de los clusters correspondientes
from pyspark.ml.evaluation import ClusteringEvaluator
evaluator = ClusteringEvaluator(featuresCol="características", predictionCol="predicción")
silhouette = evaluator.evaluate(predicciones)
print("Coste = {0}.".format(silhouette))
```

In [ ]:
```
%%sh
ls -l /tmp/kmModel/data
rm -rf /tmp/kmModel
```