

Map Reduce, Hadoop y Spark

Juan F. Pérez

Departamento MACC
Matemáticas Aplicadas y Ciencias de la Computación
Universidad del Rosario

juanferna.perez@urosario.edu.co

2018

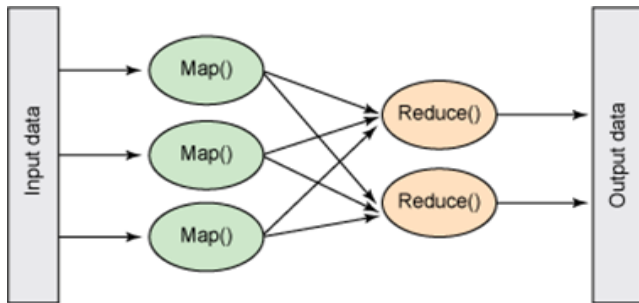
Contenidos

- 1 Map Reduce
- 2 Apache Spark
- 3 Trabajando con Apache Spark
- 4 Databricks
- 5 Trabajando con PySpark
- 6 Transformaciones y Acciones
- 7 Graficando datos
- 8 Un ejemplo: contar palabras
- 9 Ejemplo: regresión lineal
- 10 Un poquito de SQL y gráficos
- 11 Un poquito de Scala y ML
- 12 Ejemplo: streaming estructurado

Map Reduce

Map Reduce

- Google paper (2004)
- Modelo de procesamiento de datos en paralelo
- Ejecución dividida en dos grandes etapas



Map Reduce

¿Por qué Google?

Número de páginas web:

1991: 1 (WWW)

1995: 23,500 (Altavista, Amazon)

1998: 2,410,067 (Google)

2005: 64,780,610 (YouTube)

2010: 206,956,763 (Pinterest)

Hoy: > 1,259,155,000

Ejemplo Contar Palabras

- Contar cuántas veces aparece cada palabra en millones de documentos
- Cada nodo toma unos cuantos documentos y cuenta las palabras en ellos (Map)
- Cada nodo saca el total para un grupo de palabras (Reduce)

De Map Reduce a Hadoop y más allá

- Apache Hadoop 0.1.0 (Abril 2006)



-
- Yahoo corre cluster Hadoop con 1000 máquinas (Octubre 2006)
- Yahoo crea Pig (2007): lenguaje de alto nivel para análisis de datos estilo Map Reduce



-

De Map Reduce a Hadoop y más allá

- Apache Hive (2008):HiveQL: lenguaje estilo SQL para acceder a datos almacenados en Hadoop (HDFS)



■

De Map Reduce a Hadoop y más allá

- Apache Hive (2008):HiveQL: lenguaje estilo SQL para acceder a datos almacenados en Hadoop (HDFS)



- HDFS: Hadoop Distributed File System



■

De Map Reduce a Hadoop y más allá

- Apache Hive (2008):HiveQL: lenguaje estilo SQL para acceder a datos almacenados en Hadoop (HDFS)



- HDFS: Hadoop Distributed File System



- YARN: Yet Another Resource Negotiator - Hadoop 2.0 (2013)



De Map Reduce a Hadoop y más allá



- Apache Spark (2014)
- Desarrollado en Berkeley por Matei Zaharia (2009) en AMPLab
- Donado a la fundación Apache (2013)

De Map Reduce a Hadoop y más allá



- Apache Spark (2014)
- Desarrollado en Berkeley por Matei Zaharia (2009) en AMPLab
- Donado a la fundación Apache (2013)
- Versión 1.6 (2016/01)
- Versión 2.0 (2016/07)
- Versión 2.1 (2016/12)
- Versión 2.2 (2017/07)

De Map Reduce a Hadoop y más allá



- Apache Spark (2014)
- Desarrollado en Berkeley por Matei Zaharia (2009) en AMPLab
- Donado a la fundación Apache (2013)
- Versión 1.6 (2016/01)
- Versión 2.0 (2016/07)
- Versión 2.1 (2016/12)
- Versión 2.2 (2017/07)
- Versión 2.3 (2018/02)
- Versión 2.3.2 (2018/09)

Apache Spark

Apache Spark

Hadoop cuenta con excelentes herramientas para:

- Administración clústers (YARN)
- Manejo de archivos distribuidos (HDFS)

Apache Spark

Hadoop cuenta con excelentes herramientas para:

- Administración clústers (YARN)
- Manejo de archivos distribuidos (HDFS)
- Lenguajes de alto nivel (Pig, Hive, etc)

Apache Spark

Motivaciones

- Hadoop está muy atado al paradigma Map Reduce
- Flujo de datos lineal: Map + Reduce + Algunas etapas adicionales

Apache Spark

Motivaciones

- Hadoop está muy atado al paradigma Map Reduce
- Flujo de datos lineal: Map + Reduce + Algunas etapas adicionales
- Requerimiento de flujos de ejecución más complejos
- Requerimiento de examinar datos repetidamente (algoritmos iterativos)

Apache Spark

Motivaciones

- Hadoop está muy atado al paradigma Map Reduce
- Flujo de datos lineal: Map + Reduce + Algunas etapas adicionales
- Requerimiento de flujos de ejecución más complejos
- Requerimiento de examinar datos repetidamente (algoritmos iterativos)
- En Hadoop resultados de cada etapa almacenados en disco
- Almacenar en memoria mucho más veloz

Apache Spark

Motivaciones

- Hadoop está muy atado al paradigma Map Reduce
- Flujo de datos lineal: Map + Reduce + Algunas etapas adicionales
- Requerimiento de flujos de ejecución más complejos
- Requerimiento de examinar datos repetidamente (algoritmos iterativos)
- En Hadoop resultados de cada etapa almacenados en disco
- Almacenar en memoria mucho más veloz
- Análisis de flujos de datos (además del procesamiento en lotes)

Apache Spark - Librerías

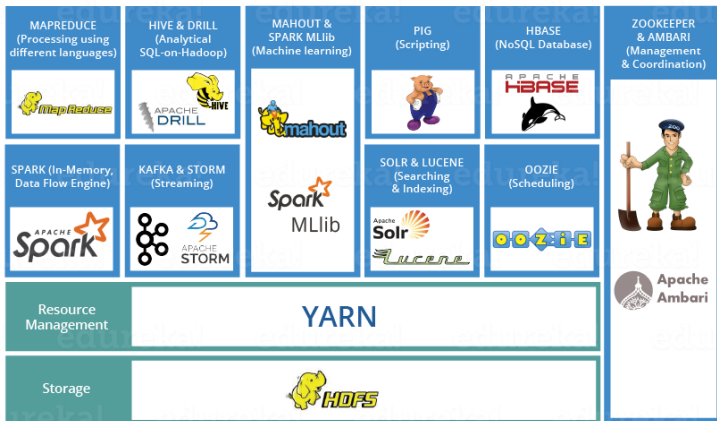
Spark SQL: manejo de datos estructurados

Spark Streaming: manejo de flujos de datos

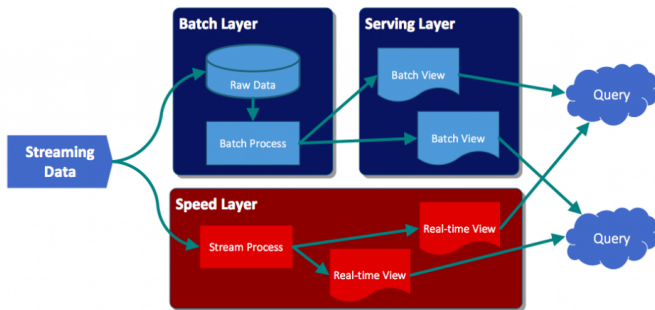
MLlib: algoritmos de aprendizaje de máquina

GraphX: análisis de datos representados como grafos

El ecosistema Hadoop/Spark



Arquitectura Lambda



Trabajando con Apache Spark

Trabajando con Apache Spark

- Spark desarrollado en Scala
- Ofrece APIs para Scala, Java, Python, R
- Python: pyspark
- Módulos:
 - Principal
 - SQL
 - Streaming
 - ML

Pyspark: Módulo principal

SparkContext

- Clase principal del módulo
- Ofrece la funcionalidad de Spark
- Conexión con el clúster de Spark
- Permite crear estructuras de datos (**RDDs**)

Estructuras de Datos en Spark: RDDs

RDD: Resilient Distributed Dataset

- Colección distribuida de objetos
- Inmutable
- Cada partición contiene un subconjunto de objetos
- Creación: cargar un set de datos externos, o distribuir una colección local

Estructuras de Datos en Spark: Dataframes y Datasets

- Introducidos en versiones más recientes de Spark
- Abstracción de más alto nivel (preferibles a trabajar directamente con RDDs)
- Internamente representados como RDDs

Dataframe:

- Colección distribuida de tipos Row
- Datos como filas de una tabla con valores por columna
- Datos sin tipos (simplemente Row)
- Similar a los dataframes en pandas

Estructuras de Datos en Spark: Dataframes y Datasets

Dataset:

- Colección distribuida de tipos de datos fijos
- Tipo de los datos definidos como una clase de caso en Scala o una clase en Java
- Similar a DataFrame pero con tipos de datos fijos
- Desde Apache 2.0, DataFrame es implementado como un Dataset con tipo de datos Row

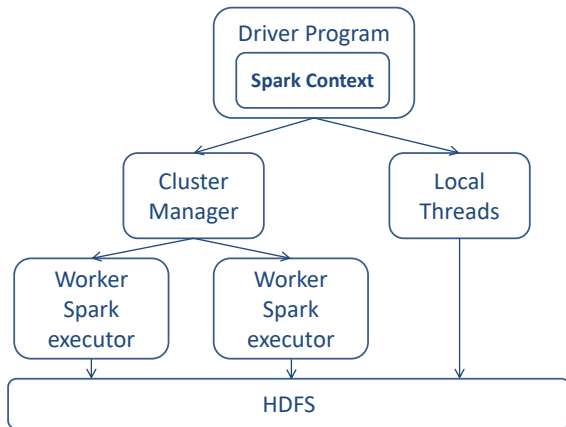
Pyspark: Módulo principal

SparkContext:

- `parallelize(c, numSlices=None)`: distribuye una colección local `c` para construir un RDD con `numSlices` particiones.
- `range(start, end=None, step=1, numSlices=None)`: crea un RDD de enteros de `start` a `end` con incrementos de `step`, almacenado en un número de particiones `numSlices`
- `union(rdds)`: crea la unión de una lista de RDDs
- `textFile(name, minPartitions=None, use_unicode=True)`: lee un archivo de texto (desde HDFS, local, etc) y retorna un RDD de Strings. Puede definir el número mínimo de particiones y si el texto usa unicode o no.

Trabajando con Spark y Pyspark

- Escribimos código en lenguaje Python
- El código se ejecuta en un cluster



Trabajando con Spark y Pyspark

- Setup de cluster es una tarea compleja
- Requiere setup del cluster manager y workers
- Alternativa: *Databricks*

Databricks

Databricks

- Plataforma administrada
- Cluster en la nube (Amazon)
- <https://community.cloud.databricks.com>
- Crear cuenta

Databricks UI

- databricks
- home
- workspace
- recent
- data
- clusters
- jobs
- search

Notebooks (Workspace)

- Workspace, Create, Notebook
- Compuestos de celdas donde escribimos código en: Python, R, Scala, SQL, Markdown
- Celda con lenguaje por defecto, puede reemplazarse con e.g., %python
- Conectar cuaderno a un cluster para ejecutar comandos
- Conexión no es permanente
- Cuadernos se pueden programar como trabajos para correr un pipeline

```
a = range(10)
print(a)
b = sum(a)/len(a)
print(b)
```

Clusters

- `Cluster`, `CreateCluster`
- Revisar versiones de Scala
- Revisar tipo de recurso
- `CreateCluster`, esperar a que esté listo
- Conectar cuaderno a un cluster para ejecutar comandos
- Volver a `Workspace`, `Detached`, `Attach` to
- En la celda `Shift+Enter`

spark

Clusters

- Grupos de computadores donde se ejecuta e código
- La idea es usarlos como si fueran un solo computador
- Ejecutar código de cuadernos sobre datos
- Datos deben estar disponibles en el clúster (HDFS/DFDS, AmazonS3)

Data

- Data, Tables, +
- Upload File, arrastrar el archivo a subir
- Create Table with UI
- Seleccionar Cluster
- Primera fila con encabezados
- Especificar Atributos (tipos)
- Create Table

Trabajando con PySpark

Trabajando con PySpark

```
spark
```

Construyendo DataFrames

```
data = [( 'Juan ' , 20), ( 'Camilo ' , 25), ( 'Sara ' , 32),  
        ( 'Marta ' , 35)]  
df = spark.createDataFrame(data)
```

```
df = spark.createDataFrame(data , [ 'nombre ' , 'edad ' ])
```

Seleccionando Columnas

```
colEdad = df.edad
```

```
df.select('*')
```

```
df2 = df.select('nombre', 'edad')
```

```
df2 = df.select(df.nombre, df.edad)
```

Seleccionando y Operando Columnas

```
df2 = df.select(df.nombre, (df.edad+20).alias('edad'))  
df2.show()
```

Removiendo Columnas

```
df3 = df2.drop(df2.edad)  
df3.show()
```

Funciones Lambda

Función

- Simple
- Sin nombre
- Útil para definir operaciones simples a aplicar a todo un set de datos (e.g., columna de un DataFrame)

Ejemplo:

```
lambda s : s*2
```

Funciones Lambda

```
from pyspark.sql.types import IntegerType
duplicar = udf(lambda s: s*2, IntegerType())
df5 = df.select(df.nombre, duplicar(df.edad)
               .alias('edad') )
df5.show()
```

Filtros

```
df6 = df5.filter(df5.edad > 45)  
df6.show()
```


Eliminando Duplicados

```
data2 = data  
data2.append(('Marta', 35))  
display(data2)
```

```
df7 = spark.createDataFrame(data2, ['nombre', 'edad'])  
df8 = df7.distinct()  
df8.show()
```

Ordenando

```
df9 = df8.sort('edad')  
df9.show()
```

```
df10 = df8.sort('edad', ascending=False)  
df10.show()
```

Ordenando

```
datax = [('Juan', 20), ('Camilo', 25), ('Sara', 32),  
         ('Marta', 35), ('Carlos', 25), ('Camila', 25)]  
dfx = spark.createDataFrame(data, ['nombre', 'edad'])  
dfx = dfx.sort('edad', 'nombre')  
dfx.show()
```

Explode

```
from pyspark.sql import Row
data3 = [Row(a=1, listaEnteros=range(1,6) )]
df11 = spark.createDataFrame(data3)
df11.show()
```

Explode

```
from pyspark.sql.functions import explode
df12 = df11.select( df11.a, explode(df11.listaEnteros)
    .alias('enteros') )
df13 = df12.select(df12.enteros)
df11.show()
df12.show()
df13.show()
```

Agrupando y Contando

```
data4 = [( 'Juan ', 20, 1200), ( 'Camilo ', 25, 10000),  
         ( 'Sara ', 32, 2500),  
         ( 'Marta ', 35, 3000), ( 'Camilo ', 45, 5000)]  
df14 = spark.createDataFrame(data4 ,  
                              [ 'nombre', 'edad', 'Balance' ])  
df14.show()
```

```
df15 = df14.groupBy(df14.nombre)  
df15 = df15.agg({ "*" : "count" } )  
df15.show()
```

Agrupando y Contando

```
df16 = df14.groupBy(df14.nombre).count()  
df16.show()
```

Calculando un promedio

```
df17 = df14.groupBy().avg()  
df17.show()
```

```
df18 = df14.groupBy('nombre').avg('edad', 'balance')  
df18.show()
```


Mostrando parte de los datos

```
df19 = spark.createDataFrame(data, ['nombre', 'edad'])  
df19.collect()
```

```
df19.take(3)
```

```
display(df19.take(3))
```

Calculando Algunas Estadísticas Descriptivas

```
df19.describe().show()
```

Importando Archivos

Después de subir un archivo de texto, e.g., pennylane.txt

```
dataPath = "/FileStore/tables/pennylane.txt"
texto = spark.read.text(dataPath)
texto.show()
print(texto.count())
from pyspark.sql.types import IntegerType
longitud = udf(lambda s: len(s), IntegerType())
textoDf1 = texto.select(longitud(texto.value)
    .alias('long') )
textoDf1.show()
```

Importando Archivos

Después de subir un archivo de texto, e.g., pennylane.txt

```
texto = spark.read.text(dataPath)
from pyspark.sql.types import IntegerType
longitud = udf(lambda s: len(s), IntegerType())
textoDf2 = texto.filter(longitud(texto.value)>80)
textoDf2.show()
print(texto.count(), textoDf2.count())
```

Importando Archivos

Después de subir un archivo de texto, e.g., pennylane.txt

```
texto = spark.read.text(dataPath)
texto.cache()
from pyspark.sql.types import BooleanType
contarPenny =
    udf(lambda s: "pennylane" in s, BooleanType())
textoDf3 = texto.filter(contarPenny(texto.value))
textoDf3.show()
print(texto.count(), textoDf3.count(),
      textoDf3.first())
```

Importando Archivos CSV

Subir un archivo CVS, e.g., `secop.csv`

- Tipos de datos en columnas
- Encabezados
- Tabla

```
dataPath = "/FileStore/tables/SECOP2_corto.csv"
secop = spark.read.format("com.databricks.spark.csv")
    .option("header", "true")
    .option("inferSchema", "true")
    .load(dataPath)
```

Transformaciones y Acciones

Transformaciones

Operaciones que **NO** se ejecutan cuando se ejecuta la celda

- Spark no las ejecuta inmediatamente
- Almacena el plan de ejecución
- Acumula todas las transformaciones que pueda hasta que se llama un acción
- Ejecución Perezosa
- Ejemplos: convertir tipos (int a float), filtrar valores, select, distinct, groupBy, sum, orderBy, filter, limit

Acciones

Operaciones que **SI** se ejecutan cuando se ejecuta la celda

- Spark ejecuta la acción inmediatamente
- Ejecuta también todas las transformaciones almacenadas
- Ejemplos: show, , count, collect, save

Transformaciones y Acciones

Beneficios:

- Spark aprovecha el plan completo para planear mejor la ejecución
- Explora oportunidades de paralelización
- Considera etapas de ejecución
- Grafo con múltiples etapas
- Resultados solo son retornados al nodo maestro al terminar todo el cálculo

Ejemplo

```
data = [('Juan', 20), ('Camilo', 25), ('Sara', 32),  
        ('Marta', 35)]  
df = spark.createDataFrame(data, ['nombre', 'edad'])
```

```
from pyspark.sql.types import IntegerType  
duplicar = udf(lambda s: s*2, IntegerType())  
df5 = df.select(df.nombre, duplicar(df.edad)  
               .alias('edad'))
```

```
df6 = df5.filter(df5.edad > 45)
```

```
df6.show()
```

Transformaciones y Acciones

Exploremos la ejecución de los trabajos

- Spark jobs
- View
- DAG Visualization
- DAG: directed acyclic graph

Graficando datos

Graficando datos (DataFrame): display

```
dataPath = "/databricks-datasets/Rdatasets/data-001/\  
csv/ggplot2/diamonds.csv"\  
diamonds = spark.read\  
. format("com.databricks.spark.csv")\  
. option("header", "true")\  
. option("inferSchema", "true")\  
. load(dataPath)\  
diamonds.show()
```

```
display(diamonds)
```

Un ejemplo: contar palabras

Contar palabras

```
dataPath = "/FileStore/tables/karamazov.txt"  
texto = spark.read.text(dataPath)  
lineas = texto.rdd.map(lambda r: r[0])
```


Contar palabras

```
from operator import add
cuentas = lineas\
    .flatMap(lambda x: x.split(' '))\
    .map(lambda x: (x, 1))\
    .reduceByKey(add)
```

Contar palabras

```
output = cuentas.collect()
outputDF = spark\
.createDataFrame(output, ['palabra', 'cuenta'])
outputDF = outputDF\
.sort(['cuenta', 'palabra'], ascending = False)
outputDF.show()
display(outputDF.take(200))
```

Ejemplo: regresión lineal

Cargar datos de un archivo CSV

```
data = spark.read.format("com.databricks.spark.csv")\
    .option("header", "true")\
    .option("inferSchema", "true")\
    .load("/databricks-datasets/samples/\
        population-vs-price/data_geo.csv")
data.cache()
data.count()
```

```
display(data)
```

Descartar datos nulos

```
data = data.dropna()  
data.count()
```

```
display(data)
```

Seleccionar columnas relevantes

```
data2 = data.select('2014 Population estimate',\
('2015 median sales price'))
display(data2)
```

Seleccionar features para el análisis

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

assembler = VectorAssembler(\
inputCols=["2014 Population estimate"],\
outputCol = "features")
output = assembler.transform(data2)
display(output.select("features",\
    "2015 median sales price"))
output.show()
```

Marcar columna como label

```
output2 = output.selectExpr(\n  " '2015 median sales price ' as label", \n  "features as features")\ndisplay(output2)
```


Definir y ajustar un modelo de regresión lineal

```
from pyspark.ml.regression import LinearRegression  
lr = LinearRegression()  
modelo = lr.fit(output2)
```

Calcular predicciones

```
prediccion = modelo.transform(output2)
display(prediccion)
```

Evaluar el modelo: error cuadrático medio

```
from pyspark.ml.evaluation import RegressionEvaluator
evaluator = RegressionEvaluator(metricName="rmse")
RMSE = evaluator.evaluate(prediccion)
print("Root Mean Squared Error = " + str(RMSE))
```

Usar pandas para preparar datos para graficar

```
from pandas import *  
from ggplot import *  
  
pob = output2.rdd.map(lambda p: (p.features[0]))\  
.collect()  
precio = output2.rdd.map(lambda p: (p.label))\  
.collect()  
pred = prediccion.select("prediction").rdd\  
.map(lambda r: r[0]).collect()  
  
pDF = DataFrame({'poblacion': pob,\  
                'precio': precio, 'prediccion': pred})  
print(pDF)
```

Usar ggplot para graficar

```
from ggplot import *

p = ggplot(pDF, aes('poblacion', 'precio')) \
+ geom_point(color='blue') \
+ geom_line(pDF, aes('poblacion', 'prediccion'), \
color='red') \
+ scale_x_log10() + scale_y_log10()
display(p)
```

Más sobre ggplot

`http://ggplot.yhathq.com/`

Un poquito de SQL y gráficos

Cargar datos de un archivo CSV

```
data = spark.read.format("com.databricks.spark.csv")\  
  .option("header", "true")\  
  .option("inferSchema", "true")\  
  .load("/databricks-datasets/samples/  
        population-vs-price/data-geo.csv")  
data.cache()
```


Limpiar y Registrar Tabla

```
data = data.dropna()  
data.show()  
data.createOrReplaceTempView("data_geo")
```

Seleccionar y graficar

```
%sql select 'State Code',  
'2015 median sales price' from data_geo
```

Gráfico: mapa

Seleccionar y graficar

```
%sql select City ,  
'2014 Population estimate '/1000  
as '2014 Population Estimate (1000s)',  
'2015 median sales price '  
as '2015 Median Sales Price (1000s)'  
from data_geo order by '  
2015 median sales price ' desc limit 10;
```

Gráfico: pie

Seleccionar y graficar

```
%sql select 'State Code',  
'2015 median sales price '  
from data_geo order by  
'2015 median sales price ' desc;
```

Gráfico: histograma

Seleccionar y graficar

```
%sql select 'State Code',  
'2015 median sales price '  
from data_geo where  
'2015 median sales price ' >= 300;
```

Gráfico: cuantiles

Seleccionar y graficar

```
%sql select 'City',  
  'State Code',  
  '2015 median sales price '  
from data_geo where  
  '2015 median sales price ' >=  
300 limit 20;
```

Gráfico: boxplot

Seleccionar y graficar

```
%sql select 'State Code',  
'2015 median sales price '  
from data_geo order by  
'2015 median sales price ' desc;
```

Gráfico: boxplot

Un poquito de Scala y ML

Crear directorio

```
% scala  
val basePath = "/tmp/mlib-persistence-example"  
dbutils.fs.rm(basePath, recurse=true)  
dbutils.fs.mkdirs(basePath)
```

Cargar datos de entrenamiento

```
entrenamiento = sqlContext.read.format("libsvm").\
option("numFeatures", "784").\
load("/databricks-datasets/mnist-digits/data-001/\
mnist-digits-train.txt")

entrenamiento.cache()

print("# de imagenes: %d " % entrenamiento.count())
```

Entrenar un clasificador

```
from pyspark.ml.classification import \
    RandomForestClassifier
rf = RandomForestClassifier(numTrees=20)
modelo = rf.fit(entrenamiento)
```

Guardar modelo

```
basePath = "/tmp/ejemplo-ml-lib-persistencia"  
modelo.save(basePath + "/modelo")
```

Guardar modelo

```
basePath = "/tmp/ejemplo-ml-lib-persistencia"  
modelo.save(basePath + "/modelo")
```

Cargar modelo

```
% scala
import org.apache.spark.ml.classification
  .RandomForestClassificationModel
val model = RandomForestClassificationModel
  .load(basePath + "/modelo")
```

Probar el modelo

```
% scala  
val test = sqlContext.read.format("libsvm")  
.option("numFeatures", "784")  
.load("/databricks-datasets/mnist-digits  
/data-001/mnist-digits-test.txt")
```

Calcular y mostrar predicciones

```
% scala  
val predicciones = model.transform(test)  
display(predicciones.select("label", "prediction"))
```


Ejemplo: streaming estructurado

Cargar datos de archivos JSON

```
% fs ls /databricks-datasets/structured-streaming/  
events/
```

```
% fs head  
/databricks-datasets/structured-streaming/  
events/file -0.json
```

Cargar datos de archivos JSON en DataFrame

```
from pyspark.sql.types import *

inputPath = "/databricks-datasets\
/structured-streaming/events/"

jsonSchema = StructType([\
    StructField("time", TimestampType(), True), \
    StructField("action", StringType(), True) ]])

staticInputDF = spark.read.\
schema(jsonSchema).json(inputPath)

display(staticInputDF)
```

Crear DF estático con ventanas de observación

```
from pyspark.sql.functions import *

staticCuentasDF = staticInputDF.groupBy( \
    staticInputDF.action, \
    window(staticInputDF.time, "1 hour")) \
    .count()

staticCuentasDF.cache()

staticCuentasDF.createOrReplaceTempView("cuentas")
staticCuentasDF.show()
```

Mostrar datos en función del tiempo

```
%sql select action ,  
date_format(window.end, "MMM-dd HH:mm") as time ,  
count from cuentas order by time , action
```

Emular Streaming y crear Dataframe

```
from pyspark.sql.functions import *  
  
streamingInputDF = (  
    spark  
        .readStream  
        .schema(jsonSchema)  
        .option("maxFilesPerTrigger", 1)  
        .json(inputPath)  
)
```

Emular Streaming y crear Dataframe

```
streamingCuentasDF = (  
  streamingInputDF  
    .groupBy(  
      streamingInputDF.action ,  
      window(streamingInputDF.time , "1 hour" ) )  
    .count()  
)
```

```
streamingCuentasDF.isStreaming
```

Definir query dinámica

```
query = (  
    streamingCuentasDF  
        .writeStream  
        .format("memory")           # en memoria  
        .queryName("counts")  
        .outputMode("complete")    # todos los counts en la  
        .start()  
)
```

Explorar counts

Revisar resultados hasta el momento

```
% sql select action ,  
date_format(window.end, "MMM-dd HH:mm") as time ,  
count from counts order by time, action
```