Spark高级编程 (I)

摘要

- Spark SQL
- Spark MLlib

摘要

- Spark SQL
- Spark MLlib

Spark SQL

Spark SQL is Apache Spark's module for working with structured data.

Integrated

Seamlessly mix SQL queries with Spark programs.

Spark SQL lets you query structured data inside Spark programs, using either SQL or a familiar DataFrame API. Usable in Java, Scala, Python and R.

```
results = spark.sql(
  "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

Apply functions to results of SQL queries.

Uniform Data Access

Connect to any data source the same way.

DataFrames and SQL provide a common way to access a variety of data sources, including Hive, Avro, Parquet, ORC, JSON, and JDBC. You can even join data across these sources.

```
spark.read.json("s3n://...")
  .registerTempTable("json")
results = spark.sql(
  """SELECT *
     FROM people
     JOIN json ...""")
```

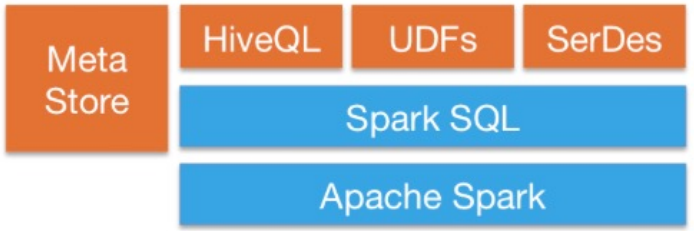
Query and join different data sources.

Spark SQL

Hive Integration

Run SQL or HiveQL queries on existing warehouses.

Spark SQL supports the HiveQL syntax as well as Hive SerDes and UDFs, allowing you to access existing Hive warehouses.



Spark SQL can use existing Hive metastores, SerDes, and UDFs.

Standard Connectivity

Connect through JDBC or ODBC.

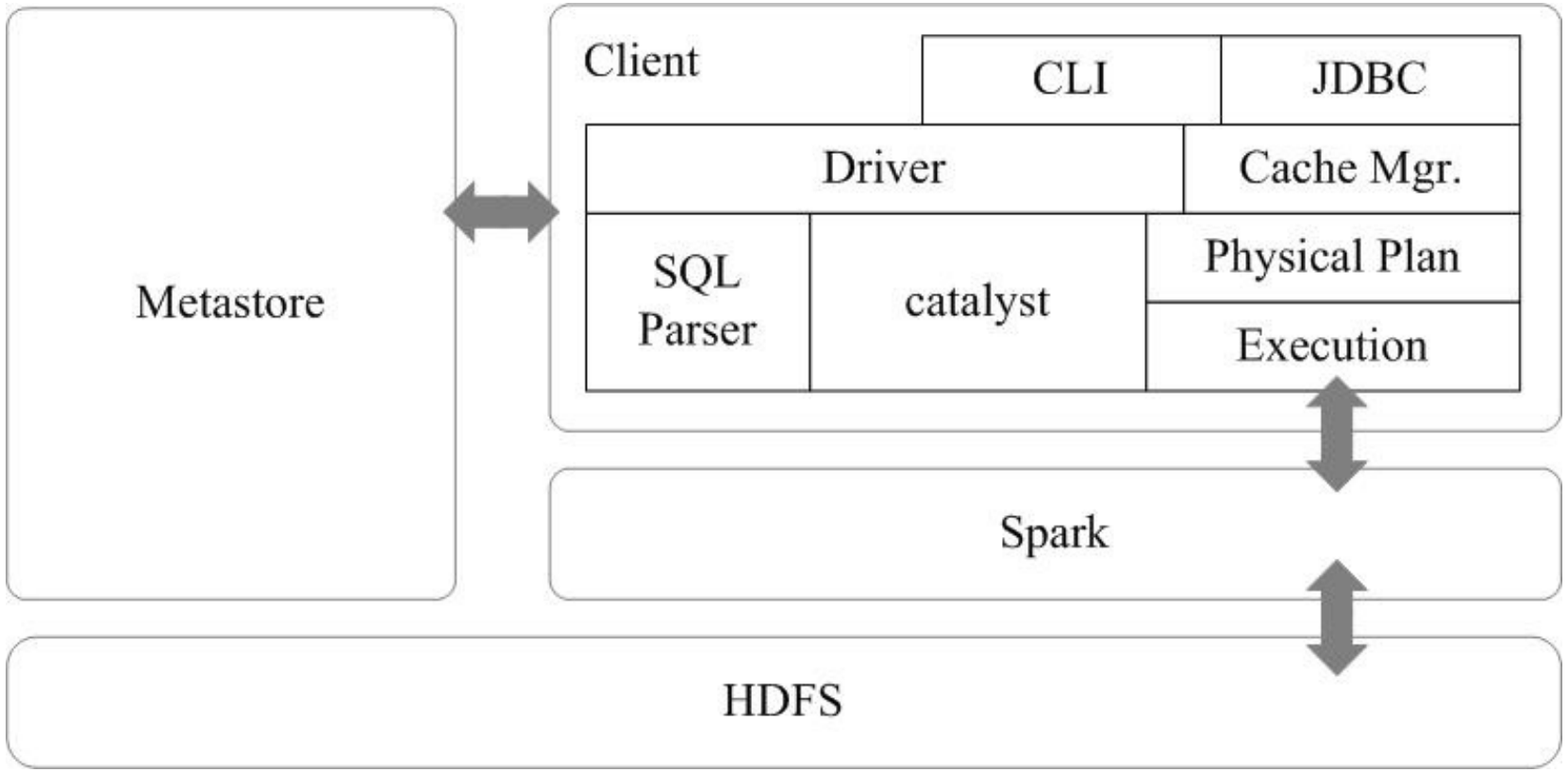
A server mode provides industry standard JDBC and ODBC connectivity for business intelligence tools.

Use your existing BI tools to query big data.

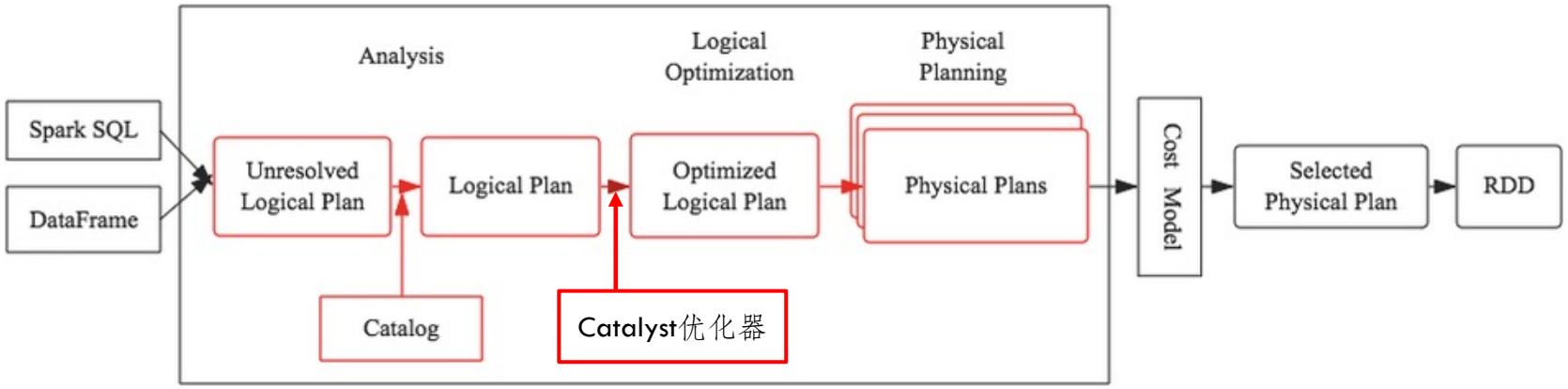
Spark SQL

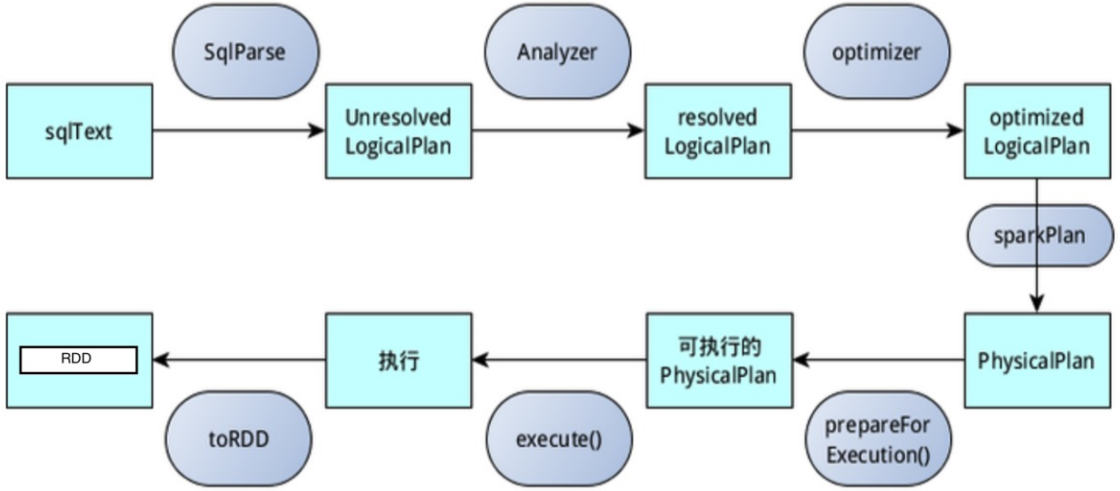
- Spark SQL：用来操作结构化和半结构化数据
  - 可以从各种结构化数据源中（例如JSON、Hive、Parquet等）读取数据；
  - 不仅支持在Spark程序内使用SQL语句进行数据查询，也支持从外部工具中通过JDBC/ODBC连接Spark SQL进行查询；
  - 支持SQL与常规的Python/Java/Scala代码高度整合，包括连接RDD与SQL表、公开的自定义SQL函数接口等。
- SchemaRDD → DataFrame/Dataset

Spark SQL架构

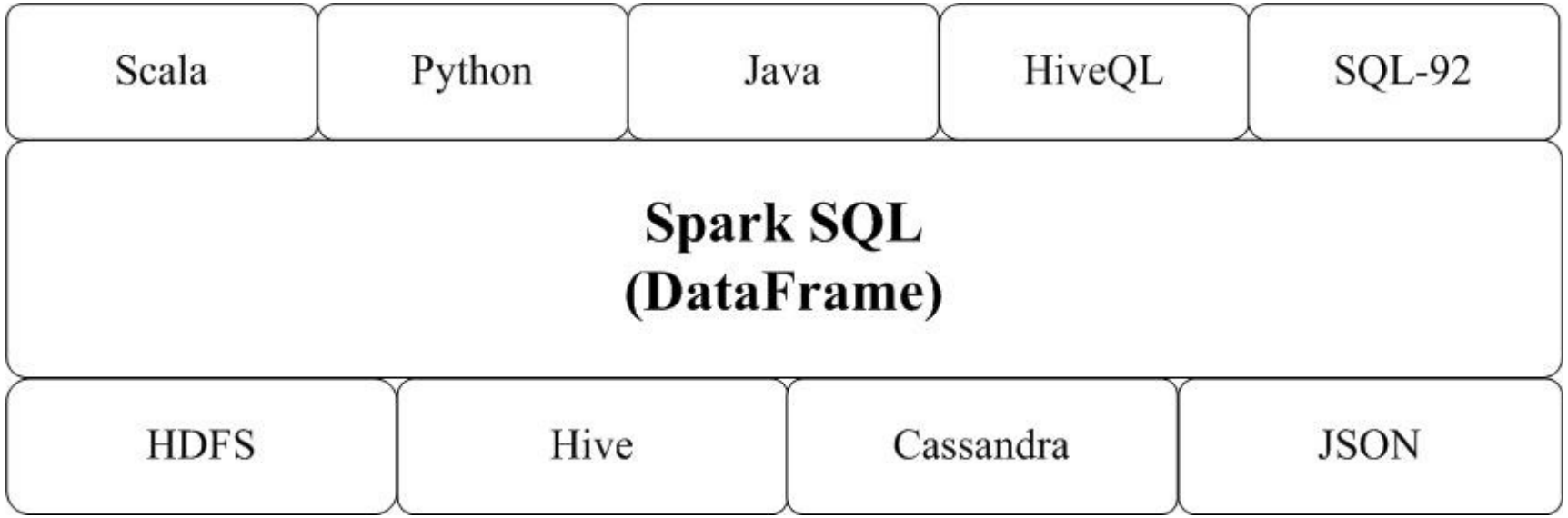

Metastore
Client
CLI
JDBC
Driver
Cache Mgr.
SQL Parser
catalyst
Physical Plan
Execution
Spark
HDFS

Spark SQL执行流程





Spark SQL支持的数据格式和编程语言



Spark SQL特点

- 数据兼容：兼容Hive，还可以从RDD、Parquet文件、JSON文件中获取数据，可以在Scala代码里访问Hive元数据，执行Hive语句，并且把结果取回作为RDD使用。支持Parquet文件读写。
- 组件扩展：语法解析器、分析器、优化器
- 性能优化：内存列存储、动态字节码生成、内存缓存数据
- 支持多种语言：Scala、Java、Python、R，还可以在Scala代码里写SQL，支持简单的SQL语法检查，能把RDD转化为DataFrame存储起来。

RDD

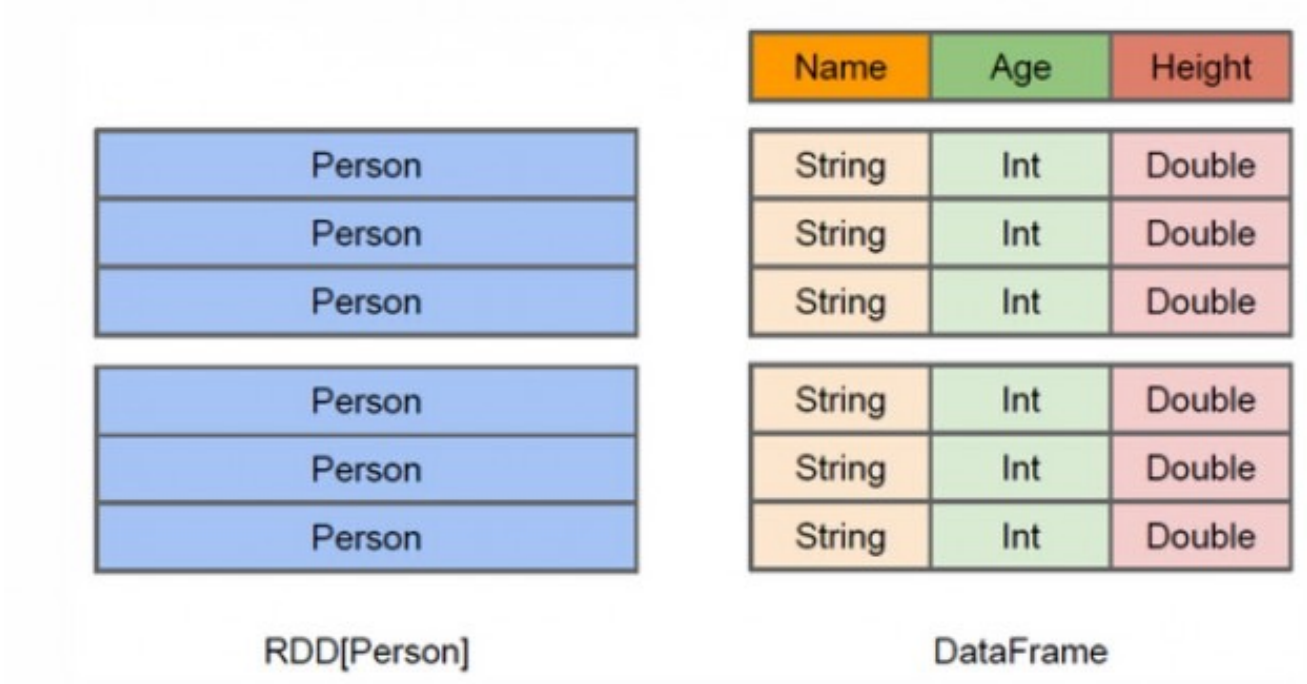
- 弹性
  - 数据可完全放内存或完全放磁盘，也可部分存放在内存，部分存放在磁盘，并可以自动切换
  - RDD出错后可自动重新计算（通过血缘自动容错）
  - 可checkpoint（设置检查点，用于容错），可persist或cache（缓存）
  - 里面的数据是分片的（也叫分区，partition），分片的大小可自由设置和细粒度调整
- 分布式
- 数据集

DataFrame

- DataFrame is a Dataset organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.
- The DataFrame API is available in Scala, Java, Python, and R. In Scala and Java, a DataFrame is represented by a Dataset of Rows. In the Scala API, DataFrame is simply a type alias of Dataset[Row]. While, in Java API, users need to use Dataset<Row> to represent a DataFrame.

DataFrame vs. RDD

- DataFrame的推出，让Spark具备了处理大规模结构化数据的能力，不仅比原有的RDD转化方式更加简单易用，而且获得了更高的计算性能。Spark能够轻松实现从MySQL到DataFrame的转化，并且支持SQL查询。



Dataset

- A Dataset is a distributed collection of data. Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine. A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.). The Dataset API is available in Scala and Java.

Dataset vs. RDD

- 相对于RDD，Dataset提供了强类型支持，也是在RDD的每行数据加了类型约束。

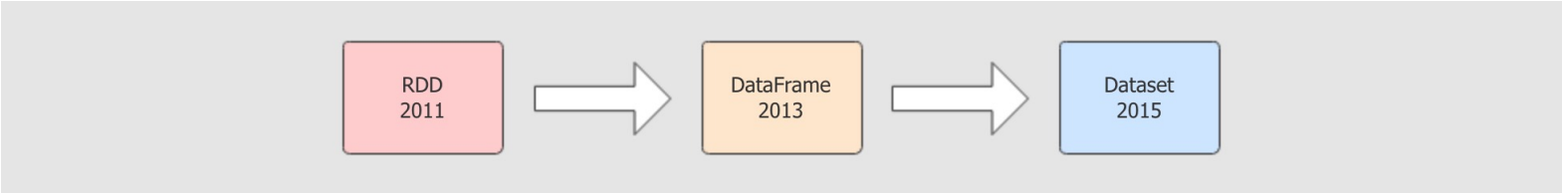
RDD

Dataset

Dataset：每行数据是一个Object

DataFrame vs. Dataset

- 相比DataFrame，Dataset提供了编译时类型检查
- RDD转换DataFrame后不可逆，但RDD转换Dataset是可逆的。
- Dataset包含了DataFrame的功能，在Spark 2.0中两者统一，DataFrame表示为DataSet[Row]，即Dataset的子集。
- 使用API尽量使用Dataset，不行再选用DataFrame，其次选择RDD。



DataFrame vs. Dataset

- 编译时类型检查

```scala
val df1 = spark.read.json("/tmp/people.json")
//json文件中没有score字段，但是能编译通过
val df2 = df1.filter("score > 60")
df2.show()
```

```scala
val ds1 = spark.read.json("/tmp/people.json").as[People]

//使用dataSet这样写，在IDE中就能发现错误
val ds2 = ds1.filter(_.score < 60)
val ds3 = ds1.filter(_.age < 18)
ds3.show()
```

DataFrame vs. Dataset

- 可逆 vs. 不可逆

```
scala> case class People(id: Long, name: String)
defined class People

scala> val peopleRDD = sc.makeRDD(Seq(People(1,"zhangsan"),People(2,"lisi")))
peopleRDD: org.apache.spark.rdd.RDD[People] = ParallelCollectionRDD[0] at makeRDD at <console>:26
```

```
scala> val peopleDf = peopleRDD.toDF
peopleDf: org.apache.spark.sql.DataFrame = [id: bigint, name: string]

scala> peopleDf.rdd
res0: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[4] at rdd at <console>:31
```

```
scala> val peopleDs = peopleRDD.toDS
peopleDs: org.apache.spark.sql.Dataset[People] = [id: bigint, name: string]

scala> peopleDs.rdd
res1: org.apache.spark.rdd.RDD[People] = MapPartitionsRDD[6] at rdd at <console>:31
```

DataFrame

- DataFrame初始化

```scala
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

```scala
val df = spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

DataFrame

- Untyped Dataset Operations (aka DataFrame Operations)

```scala
// This import is needed to use the $-notation
import spark.implicits._
// Print the schema in a tree format
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+

// Select everybody, but increment the age by 1
df.select($"name", $"age" + 1).show()
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+
```

DataFrame

- 常用的DataFrame操作
  - df.printSchema()
  - df.select(df("name"),df("age")+1).show()
  - df.filter(df("age") > 20).show()
  - df.groupBy("age").count().show()
  - df.sort(df("age").desc).show()
  - df.sort(df("age").desc, df("name").asc).show()
  - df.select(df("name").as("username"),df("age")).show()

DataFrame

- 运行SQL

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

DataFrame

- 全局临时视图

```
// Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Dataset

- Dataset初始化

```scala
case class Person(name: String, age: Long)

// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS.show()
// +----+---+
// |name|age|
// +----+---+
// |Andy| 32|
// +----+---+

// Encoders for most common types are automatically provided by importing spark.implicits._
val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)

// DataFrames can be converted to a Dataset by providing a class. Mapping will be done by name
val path = "examples/src/main/resources/people.json"
val peopleDS = spark.read.json(path).as[Person]
peopleDS.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

RDD ←→ DataFrame

- 利用反射推断模式

```
scala> import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder

scala> import org.apache.spark.sql.Encoder
import org.apache.spark.sql.Encoder

scala> import spark.implicits._  //导入包，支持把一个RDD隐式转换为一个DataFrame
import spark.implicits._

scala> case class Person(name: String, age: Long)  //定义一个case class
defined class Person

scala> val peopleDF = spark.sparkContext.textFile("file:///usr/local/spark/examples/src
/main/resources/people.txt").map(_.split(",")).map(attributes => Person(attributes(0),
attributes(1).trim.toInt)).toDF()
peopleDF: org.apache.spark.sql.DataFrame = [name: string, age: bigint]

scala> peopleDF.createOrReplaceTempView("people")  //必须注册为临时表才能供下面的查询使用

scala> val personsRDD = spark.sql("select name,age from people where age > 20")
//最终生成一个DataFrame
personsRDD: org.apache.spark.sql.DataFrame = [name: string, age: bigint]
scala> personsRDD.map(t => "Name:"+t(0)+","+"Age:"+t(1)).show()  //DataFrame中的每个元素
都是一行记录，包含name和age两个字段，分别用t(0)和t(1)来获取值
```

RDD ←→ DataFrame

- 编程指定模式

```
1.  scala> import org.apache.spark.sql.types._
2.  import org.apache.spark.sql.types._
3.
4.  scala> import org.apache.spark.sql.Row
5.  import org.apache.spark.sql.Row
6.
7.  //生成 RDD
8.  scala> val peopleRDD = spark.sparkContext.textFile("file:///usr/local/spark/examples/src/main/resources/people.txt")
9.  peopleRDD: org.apache.spark.rdd.RDD[String] = file:///usr/local/spark/examples/src/main/resources/people.txt MapPartitionsRDD[1] at textFile at <console>:26
10.
11. //定义一个模式字符串
12. scala> val schemaString = "name age"
13. schemaString: String = name age
14.
15. //根据模式字符串生成模式
16. scala> val fields = schemaString.split(" ").map(fieldName => StructField(fieldName, StringType, nullable = true))
17. fields: Array[org.apache.spark.sql.types.StructField] = Array(StructField(name,StringType,true), StructField(age,StringType,true))
18.
19. scala> val schema = StructType(fields)
20. schema: org.apache.spark.sql.types.StructType = StructType(StructField(name,StringType,true), StructField(age,StringType,true))
21. //从上面信息可以看出，schema描述了模式信息，模式中包含name和age两个字段
```

1.从原来的RDD创建一个行的RDD；

2.创建由一个 StructType 表示的模式与第一步创建的RDD的行结构相匹配；

3.在行RDD上通过createDataFrame应用Schema

RDD ←→ DataFrame

- 编程指定模式

```
23. //对peopleRDD 这个RDD中的每一行元素都进行解析val peopleDF = spark.read.format("json").load
    ("examples/src/main/resources/people.json")
24.
25.
26. scala> val rowRDD = peopleRDD.map(_.split(",")).map(attributes => Row(attributes(0),
    attributes(1).trim))
27. rowRDD: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[3] at map
    at <console>:29
28.
29. scala> val peopleDF = spark.createDataFrame(rowRDD, schema)
30. peopleDF: org.apache.spark.sql.DataFrame = [name: string, age: string]
31.
32. //必须注册为临时表才能供下面查询使用
33. scala> peopleDF.createOrReplaceTempView("people")
34.
35. scala> val results = spark.sql("SELECT name,age FROM people")
36. results: org.apache.spark.sql.DataFrame = [name: string, age: string]
37.
38. scala> results.map(attributes => "name: " + attributes(0)+","+"age:"+attributes(1)).sho
    w()
```

对比

编程模型	描述	优点	缺点
RDD	• 针对自定义数据对象进行处理，可以处理任意类型的对象； • 但是无法感知到数据的结构，**无法针对数据结构进行编程**	• 面向对象的操作方式 • 可以处理任何类型的数据	• 运行较慢，执行过程没有优化 • API僵硬，对结构化数据的访问和操作没有优化
DataFrame	• 保留数据的元数据，针对数据的结构进行处理（例如可以针对具体某一列进行处理） • 执行的时候会经过Catalyst进行优化，并且序列化更加高效，性能更好 • 只能处理结构化的数据，无法处理非结构化的数据，因为其**内部都使用Row对象保存数据**	• 针对结构化数据高度优化，通过列名访问和转换数据 • 增加Catalyst优化器，优化执行过程	• 只能操作结构化数据 • 只有无类型的API，API依然僵硬
DataSet	• 结合了RDD的优点(强类型，能够使用强大的lambda函数)和Spark SQL优化执行引擎的优点	• 可以处理结构化数据和非结构化数据 • 可以进行优化	

Spark SQL数据源

- DataFrame提供统一接口加载和保存数据源中的数据，包括：结构化数据、Parquet文件(默认)、JSON文件、Hive表，以及通过JDBC连接外部数据源。

加载


```scala
val usersDF = spark.read.load("examples/src/main/resources/users.parquet")
usersDF.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

```scala
val peopleDF = spark.read.format("json").load("examples/src/main/resources/people.json")
peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```

```scala
val peopleDFCsv = spark.read.format("csv")
  .option("sep", ";")
  .option("inferSchema", "true")
  .option("header", "true")
  .load("examples/src/main/resources/people.csv")
```

```scala
usersDF.write.format("orc")
  .option("orc.bloom.filter.columns", "favorite_color")
  .option("orc.dictionary.key.threshold", "1.0")
  .save("users_with_options.orc")
```

保存

- 保存模式SaveMode

Scala/Java	Any Language	Meaning
SaveMode.ErrorIfExists(default)	"error" or "errorifexists"(default)	如果保存数据已经存在，抛出异常
SaveMode.Append	"append"	如果保存数据已经存在，追加DataFrame数据
SaveMode.Overwrite	"overwrite"	如果保存数据已经存在，重写DataFrame数据
SaveMode.Ignore	"ignore"	如果保存数据已经存在，忽略DataFrame数据

Parquet

- Parquet是一种支持多种数据处理系统的存储格式，Spark SQL提供了读写Parquet文件，并且自动保存原始数据的模式，优点：
  - 高效，Parquet采取列式存储避免读入不需要的数据
  - 方便的压缩和解压缩
  - 可以直接固化为Parquet文件，也可以直接读取Parquet文件，具有比磁盘更好的缓存效果

JSON

- Spark SQL可以自动推断出一个JSON数据集的Schema并作为一个DataFrame加载，通过SQLContext.read.json()方法使用JSON文件创建DataFrame，或者通过转换一个JSON对象的RDD[String]创建DataFrame。

Hive

- 若要把Spark SQL连接到一个部署好的Hive上，必须把hive-site.xml复制到Spark的配置文件目录中(conf/)。
- 如果没有部署好Hive，Spark SQL会在当前的工作目录中创建出自己的Hive元数据仓库，叫做metastore_db。
- 配置项 spark.sql.warehouse.dir，默认的数据仓库地址。

Hive

- Spark SQL支持任何Hive支持的数据格式

```
import java.io.File

import org.apache.spark.sql.{Row, SaveMode, SparkSession}

case class Record(key: Int, value: String)

// warehouseLocation points to the default location for managed databases and tables
val warehouseLocation = new File("spark-warehouse").getAbsolutePath

val spark = SparkSession
  .builder()
  .appName("Spark Hive Example")
  .config("spark.sql.warehouse.dir", warehouseLocation)
  .enableHiveSupport()
  .getOrCreate()

import spark.implicits._
import spark.sql

sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")
sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

// Queries are expressed in HiveQL
sql("SELECT * FROM src").show()
```

连接数据库

- JDBC/ODBC服务器作为一个独立的Spark驱动程序运行，可以在多用户之间共享。任意一个客户端都可以在内存中缓存数据表，对表进行查询。集群的资源以及缓存数据都在所有用户之间共享。
  - 启动Thriftserver
    - >sbin/start-thriftserver.sh -master sparkMaster
  - 连接JDBC服务器
    - >bin/beeline -u jdbc:hive2://localhost:10000

性能调优

- 缓存数据

属性名称	默认值	含义
spark.sql.inMemoryColumnarStorage.compressed	true	当设置为true，Spark SQL将基于数据统计为每列自动选择压缩编码
spark.sql.inMemoryColumnarStorage.batchSize	10000	控制列式缓存的批处理尺寸，大批量可以提升内存的使用率和压缩率，但是缓存数据时会有内存溢出的风险

- 调优参数
  - spark.sql.autoBroadcastJoinThreshold; spark.sql.tungsten.enabled; spark.sql.shuffle.partitions; spark.sql.planner.externalSort; …
- 增加并行度

数据类型

- org.apache.spark.sql.types
- 数值类型
  - 字节，短整型，整型，长整型，浮点型，双精度型，数值型
- 字符串类型
- 二进制类型
- 布尔类型
- 时间类型
  - 时间戳类型，日期类型
- 复杂类型
  - 数组类型，Map类型，StructType，StructField

摘要

- Spark SQL
- Spark MLlib

Spark MLlib

MLlib is Apache Spark's scalable machine learning library.

Ease of Use

Usable in Java, Scala, Python, and R.

MLlib fits into Spark's APIs and interoperates with NumPy in Python (as of Spark 0.9) and R libraries (as of Spark 1.5). You can use any Hadoop data source (e.g. HDFS, HBase, or local files), making it easy to plug into Hadoop workflows.

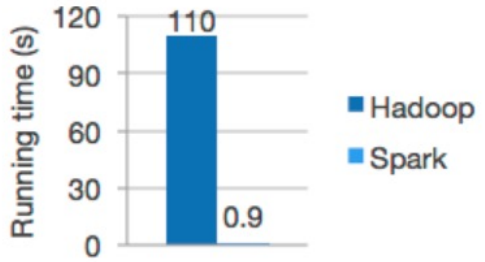
```
data = spark.read.format("libsvm")\
  .load("hdfs://...")

model = KMeans(k=10).fit(data)
```

Calling MLlib in Python

Performance

High-quality algorithms, 100x faster than MapReduce.

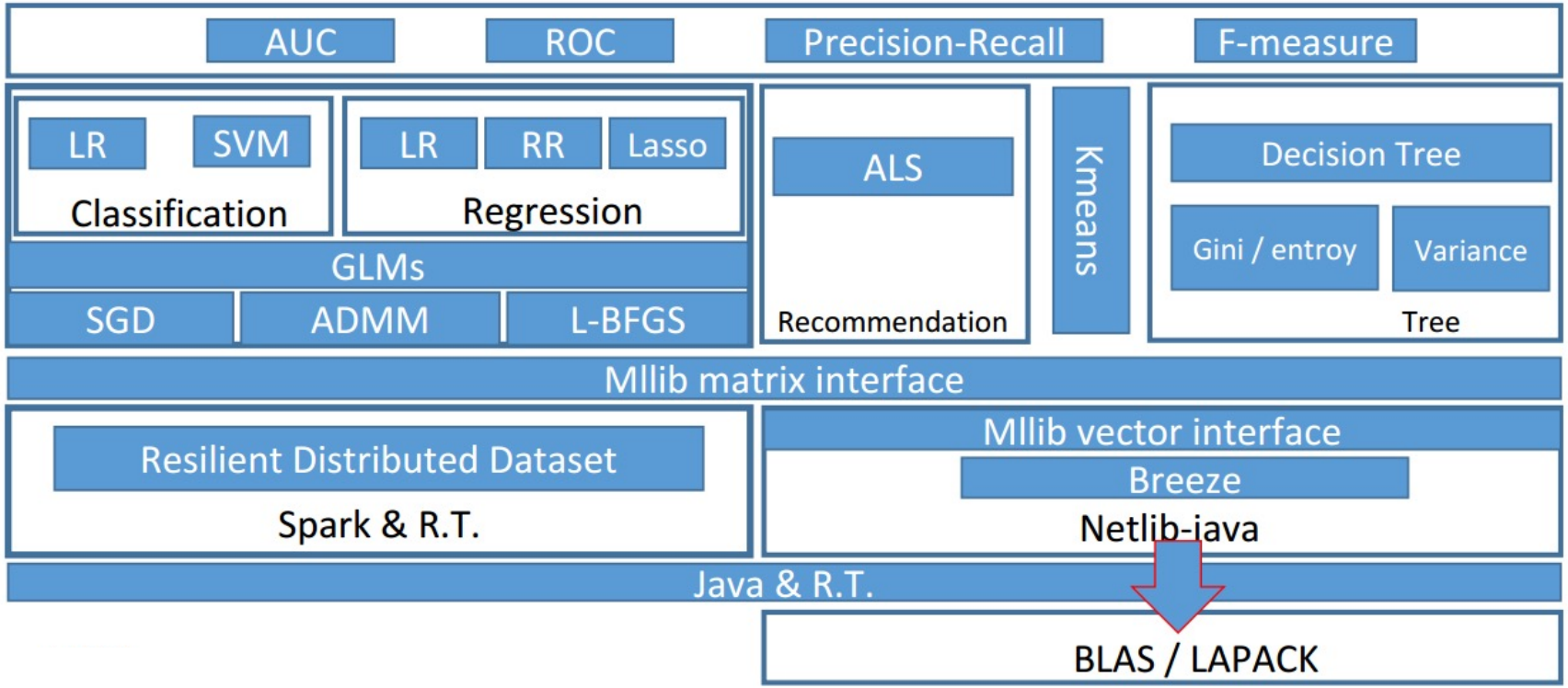
Spark excels at iterative computation, enabling MLlib to run fast. At the same time, we care about algorithmic performance: MLlib contains high-quality algorithms that leverage iteration, and can yield better results than the one-pass approximations sometimes used on MapReduce.



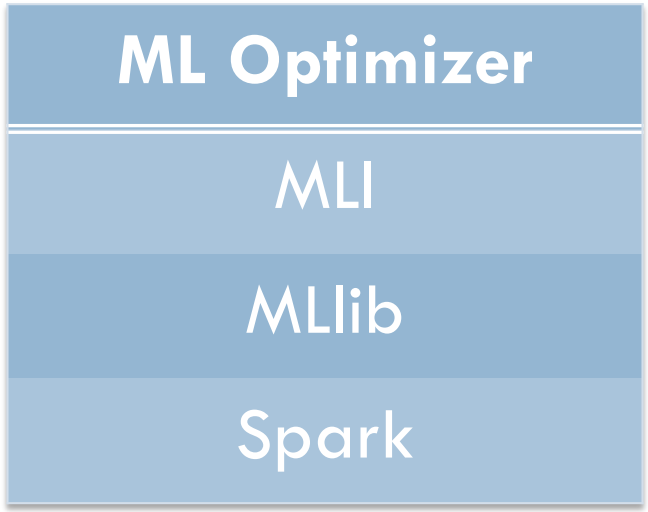
Logistic regression in Hadoop and Spark

架构

- spark.mllib.* // The MLlib RDD-based API is now in maintenance mode.
- spark.ml.* // MLlib DataFrame-based API



架构



MLBase的分层结构

- MLlib是常用机器学习算法的实现库
- MLI是进行特征抽取和高级ML编程抽象的算法实现的API
- ML Optimizer优化器会选择最合适的，已经实现好了的机器学习算法和相关参数

例子

训练分类器

```
//构造一个10行10列的数组
val data = Array.ofDim[Int](10,10)
for (i <- 0 until 10){
    for ( j <- 0 until 10){
        //给数组赋值随机数
        data(i)(j) = scala.util.Random.nextInt(100)
    }
    //取第2～10列数据（训练集的样本特征空间）
    x = data[, 2 to 10]
   //取第1列数据（样本相应的分类标签）
    y = data[, 1]
   //调用分类算法进行分类（MLBase自动选择优化方案）
    model = do_classify(y,x)
```

设计理念

- MLlib：把数据以RDD的形式表示，然后在分布式数据集上调用各种算法。引入一些数据类型（比如点和向量），给出一系列可供调用的函数的集合。
- MLlib只包含能够在集群上运行良好的并行算法
  - 特征提取，例如TF-IDF
  - 统计
  - 分类与回归：线性回归，逻辑回归，SVM，朴素贝叶斯，决策树与随机森林
  - 聚类
  - 协同过滤与推荐
  - 降维
  - 模型评估

MLlib数据类型

- 本地向量
- 标记点
- 本地矩阵
- 分布式矩阵
- 行矩阵
- 索引矩阵
- 三元组矩阵

本地向量

- 本地向量存储在单机上，由从0开始的Int型的索引和Double型的值组成，存储在单机上。
- MLlib支持两种类型的本地向量：密集向量和稀疏向量。密集向量的值由Double型的数据表示，而稀疏向量由两个并列的索引和值表示。

```
//导入MLlib
import org.apache.spark.mllib.linalg.{Vector, Vectors}
//创建（1.0, 0.0, 3.0）的密集向量
val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)
//通过指定非零向量的索引和值，创建(1.0, 0.0, 3.0)的数组类型的稀疏向量
val sv1: Vector = Vectors.sparse(3, Array(0,2), Array(1.0, 3.0))
//通过指定非零向量的索引和值，创建(1.0, 0.0, 3.0)的序列化的稀疏向量
val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))
```

标记点

- 标记点是由一个本地向量（密集或稀疏）和一个标签（Int型或Double型）组成。在MLlib中，标记点主要被应用于回归和分类这样的监督学习算法中。标签通常采用Int型或Double型的数据存储格式。

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
//通过一个正相关的标签和一个密集的特征向量创建一个标记点
val  pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))
//通过一个负向标签和一个稀疏特征向量创建一个标记点
val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0,2), Array(1.0, 3.0)))
```

稀疏数据

- MLlib可以读取存储为LIBSVM格式的数据，其每一行代表一个带有标签的稀疏特征向量。格式如下：

  label index1:value1 index2:value2 …

- 其中label是标签值，index是索引，其值从1开始递增。加载完成后，索引被转换为从0开始。
- 接口： MLUtils.loadLibSVMFile

  val examples: RDD[LabeledPoint] = MLUtils.loadLibSVMFile(sc, “data/MLlib/sample_libsvm_data.txt”)

本地矩阵

- 本地矩阵是由（Int类型行索引，Int类型列索引，Double类型值）组成，存放在单机中。Mllib支持密集矩阵，密集矩阵的值以列优先方式存储在一个Double类型的数组中，矩阵如下：

$$\begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \\ 5.0 & 6.0 \end{bmatrix} \qquad \begin{bmatrix} 9.0 & 0.0 \\ 0.0 & 8.0 \\ 0.0 & 6.0 \end{bmatrix}$$

- 这个3行2列的矩阵存储在一个一维数组[1.0, 3.0, 5.0, 2.0, 4.0, 6.0]中。
- MLlib实现： DenseMatrix

```
val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))
val sm: Matrix = Matrices.sparse(3, 2, Array(0, 1, 3), Array(0, 2, 1), Array(9, 6, 8))
```

分布式矩阵

- 分布式矩阵由（Long类型行索引，Long类型列索引，Double类型值）组成，分布存储在一个或多个RDD中。因为要缓存矩阵的大小，所以分布式矩阵底层的RDD必须是确定的，选择正确的格式来存储巨大的分布式矩阵是非常重要的，否则会导致错误的出现。MLlib已实现了四种分布式矩阵：
  - 行矩阵 RowMatrix
  - 行索引矩阵 IndexedRowMatrix
  - 三元组矩阵 CoordinateMatrix
  - 块矩阵 BlockMatrix

MLlib的算法库

- 基本统计
  - 汇总统计，相关性统计，分层抽样，假设检验，随机数据生成，核密度估计
- 分类和回归
  - 线性模型（支持向量机SVM、逻辑回归、线性回归）
  - 朴素贝叶斯
  - 决策树，随机森林和梯度提升决策树（GBT）
- 协同过滤
  - 交替最小二乘法（ALS）
- 聚类
  - K-means，高斯混合，快速迭代聚类，三层贝叶斯概率模型，流式K-means

MLlib的算法库

- 降维
  - 奇异值分解（SVD）
  - 主成分分析（PCA）
- 频繁模式挖掘
  - FP-growth，关联规则，PrefixSpan
- 优化器
  - 随机梯度下降
  - 限制内存BFGS（L-BFGS）
- 特征值提取和转换，评价指标，PMML模型输出等算法实现

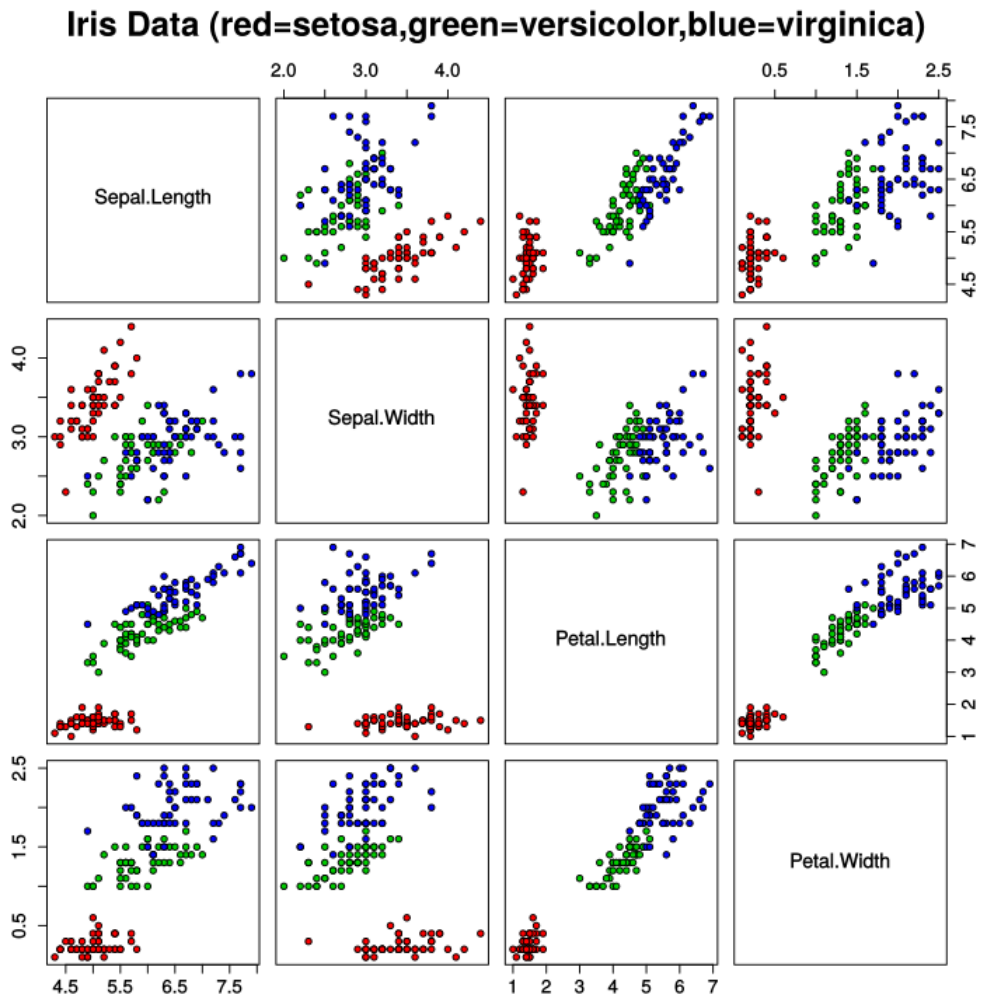
常见步骤

- 例如，如果要用MLlib来完成文本分类的任务，只需如下操作：
  - 首先用字符串RDD来表示你的消息
  - 运行MLlib的一个特征提取算法来把文本数据转换为数值特征，该操作会返回一个向量RDD
  - 对向量RDD调用分类算法（比如逻辑回归），这步会返回一个模型对象，可以使用该对象对新的数据点进行分类
  - 使用MLlib的评估函数在测试数据集上评估模型

再看K-Means


```
import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}
import org.apache.spark.mllib.linalg.Vectors

val data = sc.textFile("data/mllib/kmeans_data.txt")
val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble))).cache()
// Cluster the data into two classes using KMeans
val numClusters = 2
val numIterations = 20
val clusters = KMeans.train(parsedData, numClusters, numIterations)
// Evaluate clustering by computing Within Set Sum of Squared Errors
val WSSSE = clusters.computeCost(parsedData)
println("Within Set Sum of Squared Errors = " + WSSSE)
// Save and load model
clusters.save(sc, "target/org/apache/spark/KMeansExample/KMeansModel")
val sameModel = KMeansModel.load(sc, "target/org/apache/spark/KMeansExample/KMeansModel")
```

Iris数据集分类



Iris数据集是常用的分类实验数据集，由Fisher, 1936收集整理。Iris也称鸢尾花卉数据集，是一类多重变量分析的数据集。数据集包含150个数据，分为3类，每类50个数据，每个数据包含4个属性。可通过花萼长度，花萼宽度，花瓣长度，花瓣宽度4个属性预测鸢尾花卉属于（Setosa，Versicolour，Virginica）三个种类中的哪一类。

实验步骤：数据处理

- 首先需要将Iris-setosa，Iris-versicolour，Iris-virginica转化成0，1，2来表示。生成LabeledPoint类型RDD
  - 利用*loadLibSVMFile*接口从LibSVM格式的文件读取数据。当然首先需要把原始的数据文件转换成LibSVM格式，然后调用*loadLibSVMFile*接口就可以生成LabeledPoint类型的RDD。
  - 先用textFile 读取数据，然后对string类型的RDD调用map操作，转换成LabeledPoint类型的RDD。

实验步骤：数据处理


```
# 读取数据
val rdd: RDD[String] = sc.textFile(path)
# 转换得到LabeledPoint
var rddLp: RDD[LabeledPoint] = rdd.map( x => { val strings: Array[String] = x.split(",")
regression.LabeledPoint( strings(4) match { case "Iris-setosa" => 0.0 case "Iris-versicolor" => 1.0 case "Iris-virginica"
=> 2.0 } , Vectors.dense( strings(0).toDouble, strings(1).toDouble, strings(2).toDouble, strings(3).toDouble)) } )
# 分割数据集为训练集和测试集
val Array(trainData,testData): Array[RDD[LabeledPoint]] = rddLp.randomSplit(Array(0.8,0.2))
```

实验步骤：训练模型及模型评估

- 选取朴素贝叶斯，决策树，随机森林，支持向量机，以及logistics回归共5种分类算法。采用留出法对建模结果评估，留出30%数据作为测试集，评估标准采用精度accuracy。
- 支持向量机（SVM），logistics回归是二分类的算法，由于本数据集有多个类别，所以可以利用多个二分类分类器来实现多分类目标。

参考代码：决策树


```
#构建模型
val decisonModel: DecisionTreeModel = DecisionTree.trainClassifier(trainData,3, Map[Int, Int](),"gini",8,16)
# 得到测试集预测的结果
val result: RDD[(Double, Double)] = testData.map( x=> { val pre: Double = decisonModel.predict(x.features) (x.label,pre) } )
val acc: Double = result.filter(x=>x._1==x._2).count().toDouble /result.count()
```

Scala

参考代码：朴素贝叶斯

Python

```
# 分割数据集为训练集和测试集
traindata,testdata = data.randomSplit([0.7,0.3])
# 朴素贝叶斯训练并评估
Bayesmodel = NaiveBayes.train(traindata,1.0)
predictionAndLabel_Bayes = testdata.map(lambda    p :(Bayesmodel.predict(p.features),p.label))
accuracy= 1.0*predictionAndLabel_Bayes.filter(lambda p1:    p1[0]==p1[1]).count()/testdata.count()
```

参考代码：SVM

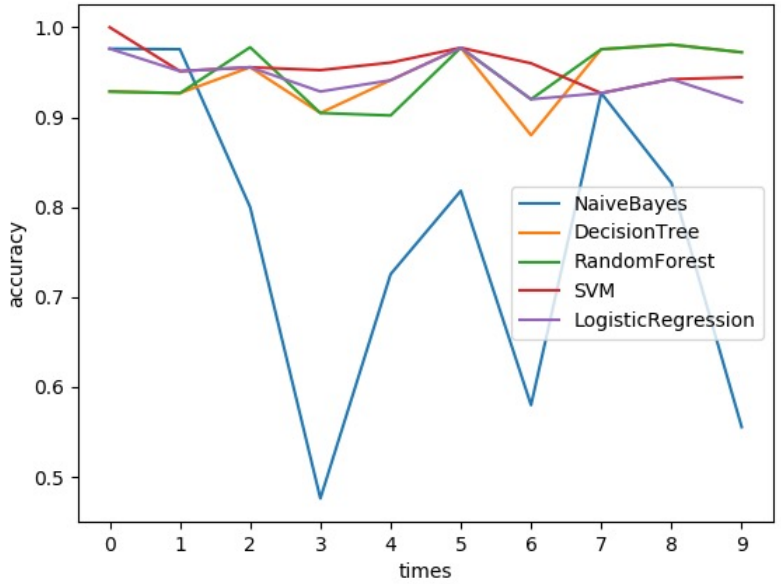
Python

```
#用多个SVM分类器实现多分类
model1 = SVMWithSGD.train(train0_1, iterations=1000)
model2 =SVMWithSGD.train(train0_2,iterations=1000)
model3 = SVMWithSGD.train(train1_2,iterations=1000)
predictions1 = model1.predict(testdata.map(lambda x :x.features))
predictions2 = model2.predict(testdata.map(lambda x :x.features))
predictions3 = model3.predict(testdata.map(lambda x :x.features))
true_label = testdata.map(lambda x :x.label).collect()
label_list1=predictions1.collect() ;
label_list2=predictions2.collect();
label_list3=predictions3.collect()
#投票产生结果
predict_label =[]
account =0
```

参考代码：SVM


```
for index in range(len(true_label)):
    dictionary ={0.0:0,1.0:0,2.0:0}
    if label_list1[index] ==0:
        dictionary[0.0] +=1
    else: dictionary[1.0]+=1
    if label_list2[index] ==0:
        dictionary[0.0]+=1
    else: dictionary[2.0] +=1
    if label_list3[index] ==0:
        dictionary[2.0]+=1
    else: dictionary[1.0]+=1
    maxlabel = 0.0
```

```
for item in dictionary.keys():
    if dictionary[item]>dictionary[maxlabel]:
        maxlabel =item
    if maxlabel==true_label[index]:
        account+=1
    predict_label.append(maxlabel)
accuracy_SVM =1.0*account/len(true_label)
```



ML库

- Spark的ML库基于DataFrame提供高性能的API，帮助用户创建和优化实用的机器学习流水线（Pipeline），包括特征转换独有的Pipelines API。相比较Mllib，变化主要体现在：
  - 从机器学习的library开始转向构建一个机器学习工作流的系统。ML把整个机器学习的过程抽象成Pipeline，一个Pipeline由多个Stage组成，每个Stage由Transformer或者Estimator组成。
  - ML框架下所有的数据源都基于DataFrame，所有模型都基于Spark的数据类型表示，ML的API操作也从RDD向DataFrame全面转变。

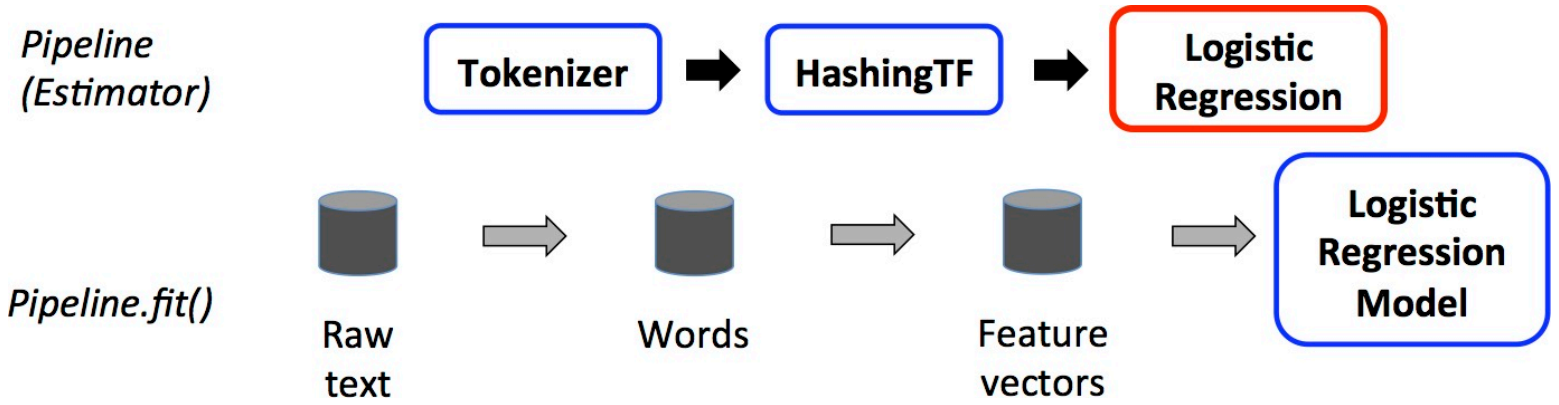
ML主要概念

- DataFrame：将Spark SQL的DataFrame作为一个ML数据集使用，支持多种数据类型。一个DataFrame可以有不同的列存储文本、特征向量、真实标签和预测。
- Transformer：实现一个DataFrame转换成另一个DataFrame的算法。实现transform()方法。
- Estimator：适配一个DataFrame，产生另一个Transformer的算法。实现fit()方法。
- Pipeline：指定连接多个Transformers和Estimators的ML工作流。
- Parameter：全部的Transformers和Estimators共享一个指定Parameter的通用API。

Pipeline

- 机器学习的流水线通常指运行一系列算法的过程，并从数据中学习。例如，一个简单的文本文档处理工作流程可能包括以下几个阶段：
  - 将每个文档的文本切分成单词；
  - 将每个文档单词转换成一个数值特征向量；
  - 使用特征向量和标签，学习一个预测模型。
- Spark ML代表一个作为流水线的工作流，由一系列流水线阶段组成，并以一个特定的顺序运行。
- 一个流水线被指定为一系列由Transformer或Estimator组成的阶段（Stage）。这些阶段按照顺序运行，输入的DataFrame在运行的每个阶段进行转换。

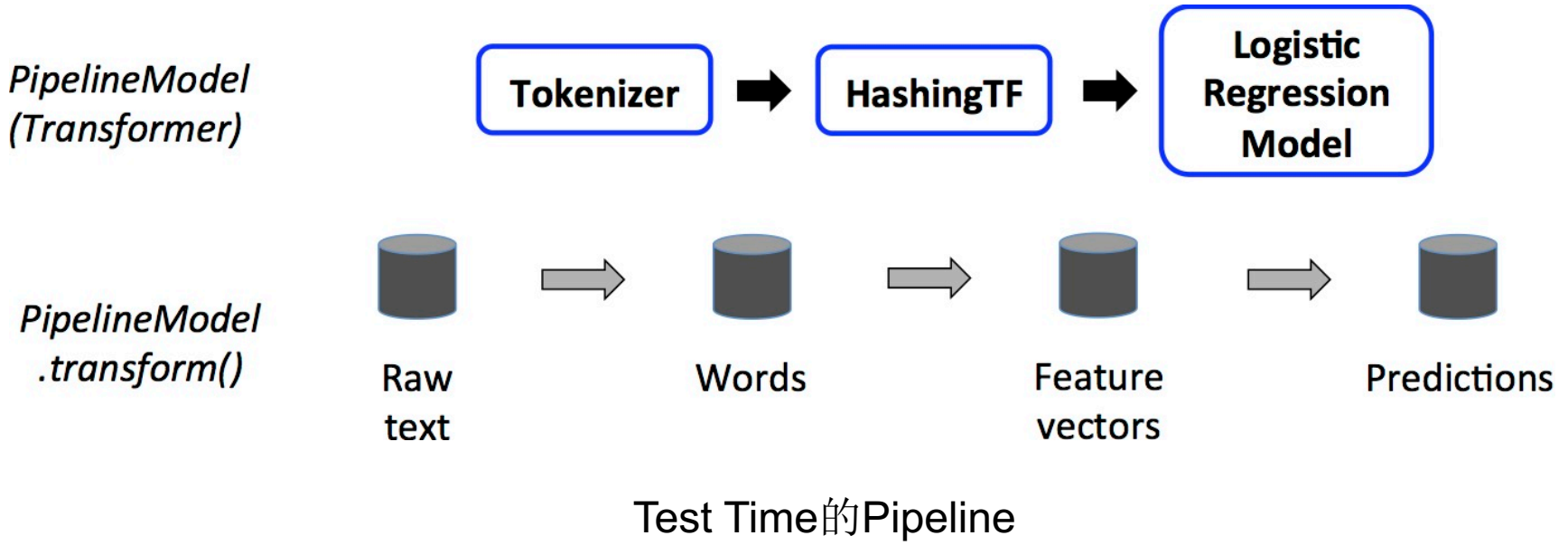
Pipeline



Training Time的Pipeline

流水线是一个Estimator，因此，在一个流水线的fit()方法运行之后，生成一个PipelineModel，该模型是一个Transformer。

Pipeline



Test Time的Pipeline

Pipeline和PipelineModel在实际运行Pipeline之前，使用DataFrame模式（schema）进行类型检查，该模式描述DataFrame中列的数据类型。

再看K-Means


```
import org.apache.spark.ml.clustering.Kmeans
val dataset = spark.read.format("libsvm").load("data/mllib/sample_kmeans_data.txt")
// Trains a k-means model.
val kmeans = new KMeans().setK(2).setSeed(1L)
val model = kmeans.fit(dataset)
// Make predictions
val predictions = model.transform(dataset)
// Evaluate clustering by computing Silhouette score
val evaluator = new ClusteringEvaluator()
val silhouette = evaluator.evaluate(predictions)
// Shows the result.
println("Cluster Centers: ")
model.clusterCenters.foreach(println)
```

再看鸢尾花


```
val df: DataFrame = sparkSession.read.format("csv").option("inferSchema",
"true").option("header","true").option("sep",",").load(path)
//特征工程
//将4个特征整合为一个特征向量
val assembler: VectorAssembler = new VectorAssembler().setInputCols(Array
(“sepal_length”,“sepal_width”,“petal_length”,“petal_width”)).setOutputCol(“features”)
val assmblerDf: DataFrame = assembler.transform(df)
//将类别型class转变为数值型
val stringIndex: StringIndexer = new StringIndexer().setInputCol(“class”). setOutputCol("label")
val stingIndexModel: StringIndexerModel = stringIndex.fit(assmblerDf)
val indexDf: DataFrame = stingIndexModel.transform(assmblerDf)
//将数据切分成两部分，分别为训练数据集和测试数据集
val Array(trainData,testData): Array[Dataset[Row]] = indexDf.randomSplit (Array(0.8,0.2))
```

再看鸢尾花


```
// 准备计算，设置特征列和标签列
val classifier: DecisionTreeClassifier = new DecisionTreeClassifier().setFeaturesCol
("features").setMaxBins(16).setImpurity("gini").setSeed(10)
val dtcModel: DecisionTreeClassificationModel = classifier.fit(trainData)
// 完成建模分析
val trainPre: DataFrame = dtcModel.transform(trainData)
// 预测分析
val testPre: DataFrame = dtcModel.transform(testData)
// 评估
val acc: Double = new MulticlassClassificationEvaluator().setMetricName ("accuracy").evaluate(testPre)
```

一般步骤

- Spark MLlib：
  - 加载数据
  - 把数据转换成所需的格式
  - 设置算法参数
  - 调用算法模型训练
  - 预测
  - 模型评估
- Spark ML：
  - 把整个机器学习过程抽象成Pipeline
  - 通过Transformer和Estimator构成的多个Stage完成Pipeline过程。

预测问题

- 1. 导入需要的包
- 2. 读取训练数据
- 3. 构建模型
- 4. 评估模型

THANK YOU

南京大学
NANJING UNIVERSITY

南京大学计算机软件研究所
Institute of Computer Software, Nanjing University