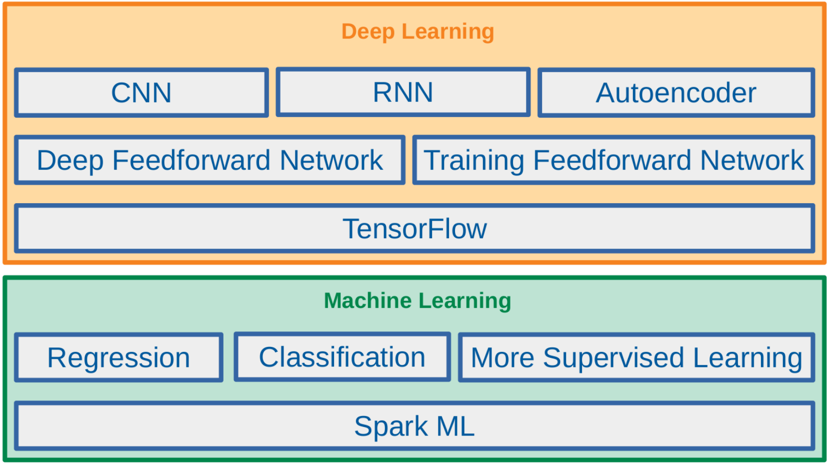
Machine Learning with Spark

Amir H. Payberah
payberah@kth.se
02/11/2018

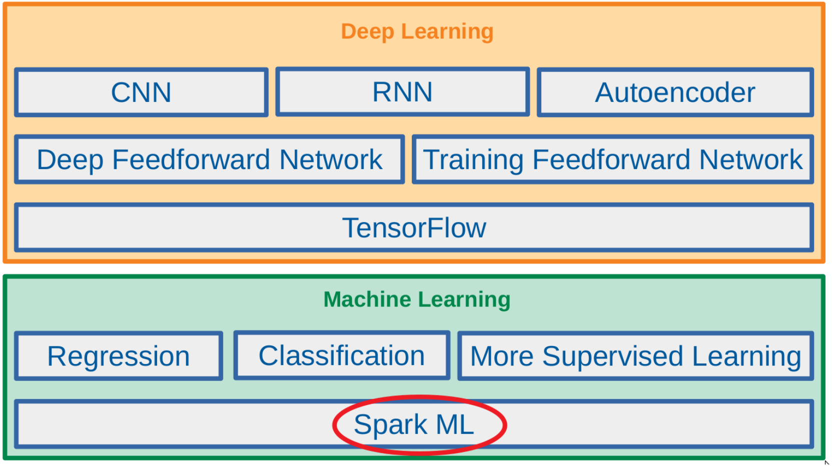
The Course Web Page

https://id2223kth.github.io

Where Are We?


Deep Learning
CNN
RNN
Autoencoder
Deep Feedforward Network
Training Feedforward Network
TensorFlow
Machine Learning
Regression
Classification
More Supervised Learning
Spark ML

Where Are We?



Deep Learning
CNN
RNN
Autoencoder
Deep Feedforward Network
Training Feedforward Network
TensorFlow
Machine Learning
Regression
Classification
More Supervised Learning
Spark ML

Big Data

Problem

- Traditional platforms fail to show the expected performance.
- Need new systems to store and process large-scale data

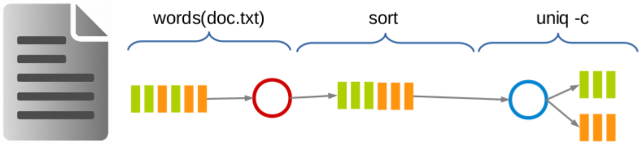
Scale Up vs. Scale Out

- Scale up or scale vertically
- Scale out or scale horizontally

Spark

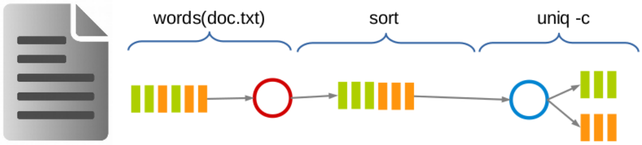
Word Count

- Count the number of times each distinct word appears in the file
- If the file fits in memory: `words(doc.txt) | sort | uniq -c`



Word Count

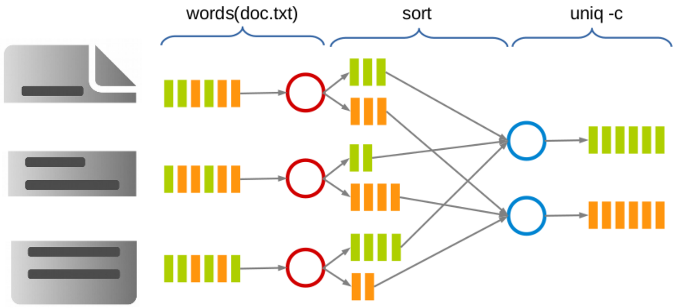
- Count the number of times each distinct word appears in the file
- If the file fits in memory: `words(doc.txt) | sort | uniq -c`



- If not?

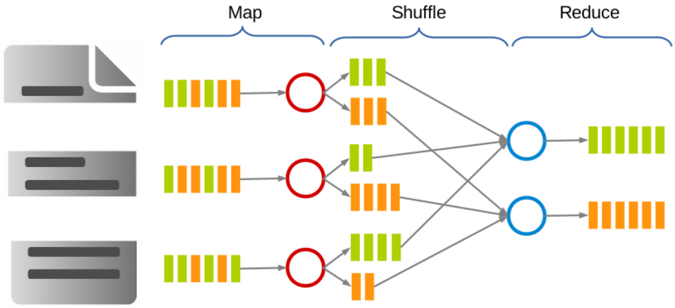
Data-Parallel Processing (1/4)

- Parallelize the data and process.



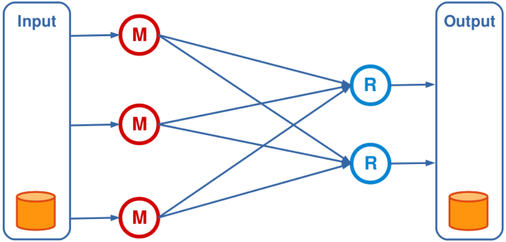
Data-Parallel Processing (2/4)

- MapReduce

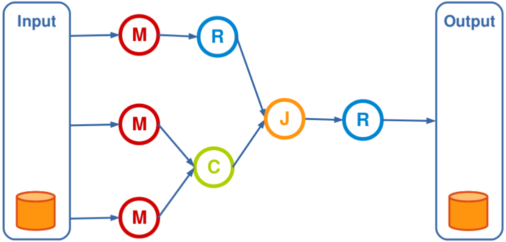


Data-Parallel Processing (3/4)

- Data flow programming model.
- Acyclic Directed Graph (DAG) of data flow from stable storage to stable storage.

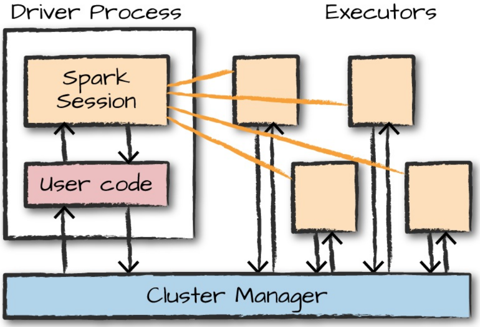


Data-Parallel Processing (4/4)



Spark Execution Model (1/3)

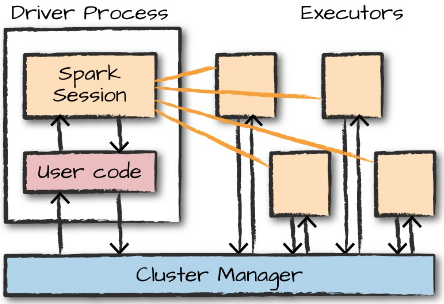
- Spark applications consist of
  - A driver process
  - A set of executor processes



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

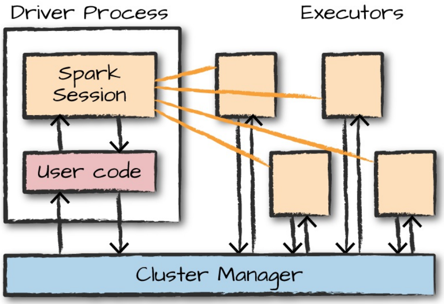
Spark Execution Model (2/3)

- The driver process is the heart of a Spark application
- Sits on a node in the cluster
- Runs the `main()` function



Spark Execution Model (3/3)

- Executors execute codes assigned to them by the driver.



Resilient Distributed Datasets (RDD) (1/2)

- A distributed memory abstraction.

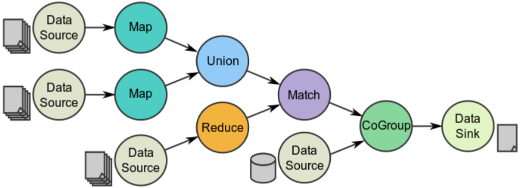
- Immutable collections of objects spread across a cluster.
  - Like a `LinkedList <MyObjects>`

Resilient Distributed Datasets (RDD) (2/2)

- An RDD is divided into a number of partitions, which are atomic pieces of information.
- Partitions of an RDD can be stored on different nodes of a cluster.

Spark Programming Model

- Job description based on directed acyclic graphs (DAG).
- There are two types of RDD operators: transformations and actions.

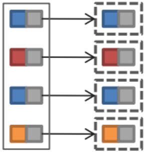


RDD Operations

- Transformations: lazy operators that create new RDDs.
- Actions: lunch a computation and return a value to the program or write data to the external storage.

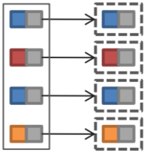
RDD Transformations (1/3)

- Map: all pairs are independently processed.



RDD Transformations (1/3)

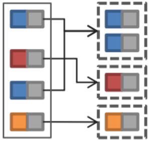
- Map: all pairs are independently processed.



```scala
// passing each element through a function.
val nums = sc.parallelize(Array(1, 2, 3))
val squares = nums.map(x => x * x) // {1, 4, 9}

// selecting those elements that func returns true.
val even = squares.filter(x => x % 2 == 0) // {4}

// mapping each element to zero or more others.
nums.flatMap(x => Range(0, x)) // {0, 0, 1, 0, 1, 2}
```

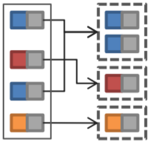
RDD Transformations (2/3)

- Group by: pairs with identical key are grouped.
- Groups are independently processed.



RDD Transformations (2/3)

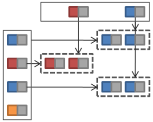
- Group by: pairs with identical key are grouped.
- Groups are independently processed.



```
val pets = sc.parallelize(Seq(("kth", 1), ("rise", 1), ("kth", 2)))

pets.groupByKey()
// {(kth, (1, 2)), (rise, (1))}

pets.reduceByKey((x, y) => x + y)
// {(kth, 3), (rise, 1)}
```

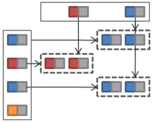
RDD Transformations (3/3)

- Join: performs an equi-join on the key.
- Join candidates are independently processed.



RDD Transformations (3/3)

- **Join**: performs an **equi-join** on the key.
- Join candidates are independently processed.



```
val rdd1 = sc.parallelize(Seq(("a", 1), ("b", 2), ("a", 3)))

val rdd2 = sc.parallelize(Seq(("a", "g1"), ("b", "g2")))

rdd1.join(rdd2)
// ("a", (1, "g1"))
// ("a", (3, "g1"))
// ("b", (2, "g2"))
```

Basic RDD Actions (1/2)

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

Basic RDD Actions (1/2)

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

Basic RDD Actions (1/2)

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

- Return the number of elements in the RDD.

```
nums.count() // 3
```

Basic RDD Actions (2/2)

- Aggregate the elements of the RDD using the given function.

```scala
nums.reduce((x, y) => x + y)
// or
nums.reduce(_ + _) // 6
```

Basic RDD Actions (2/2)

- Aggregate the elements of the RDD using the given function.

```
nums.reduce((x, y) => x + y)
// or
nums.reduce(_ + _) // 6
```

- Write the elements of the RDD as a text file.

```
nums.saveAsTextFile("hdfs://file.txt")
```

Creating RDDs

- Turn a collection into an RDD.

```
val a = sc.parallelize(Array(1, 2, 3))
```

Creating RDDs

- Turn a collection into an RDD.

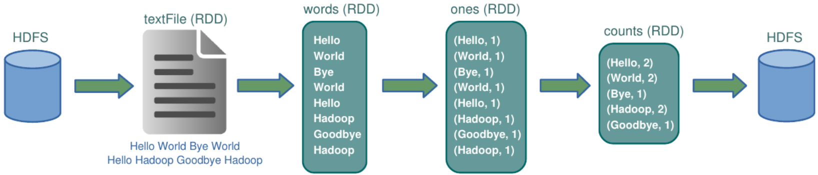
```
val a = sc.parallelize(Array(1, 2, 3))
```

- Load text file from local FS, HDFS, or S3.

```
val a = sc.textFile("file.txt")
val b = sc.textFile("directory/*.txt")
val c = sc.textFile("hdfs://namenode:9000/path/file")
```

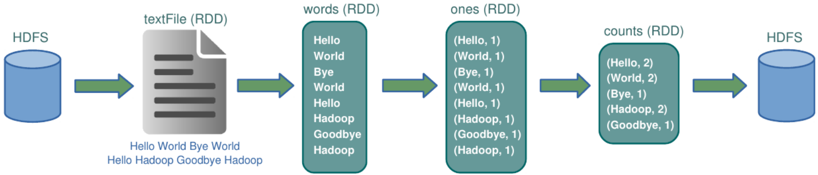
Example

```scala
val textFile = sc.textFile("hdfs://...")

val words = textFile.flatMap(line => line.split(" "))
val ones = words.map(word => (word, 1))
val counts = ones.reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```



Example

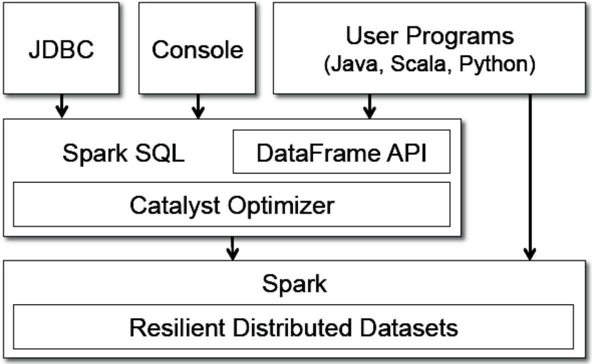
```
val textFile = sc.textFile("hdfs://...")

val words = textFile.flatMap(line => line.split(" "))
val ones = words.map(word => (word, 1))
val counts = ones.reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```



- Lineage: transformations used to build an RDD.

Spark SQL

Spark and Spark SQL

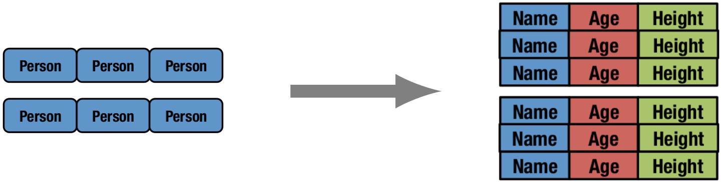


DataFrame

- A DataFrame is a distributed collection of rows with a homogeneous schema.
- It is equivalent to a table in a relational database.
- It can also be manipulated in similar ways to RDDs.

Adding Schema to RDDs

- Spark + RDD: functional transformations on partitioned collections of opaque objects.
- SQL + DataFrame: declarative transformations on partitioned collections of tuples.



Creating a DataFrame - From an RDD

- You can use `toDF` to convert an RDD to DataFrame.

```
val tupleRDD = sc.parallelize(Array(("seif", 65, 0), ("amir", 40, 1)))
val tupleDF = tupleRDD.toDF("name", "age", "id")
```

Creating a DataFrame - From an RDD

- You can use toDF to convert an RDD to DataFrame.

```
val tupleRDD = sc.parallelize(Array(("seif", 65, 0), ("amir", 40, 1)))
val tupleDF = tupleRDD.toDF("name", "age", "id")
```

- If RDD contains case class instances, Spark infers the attributes from it.

```
case class Person(name: String, age: Int, id: Int)

val peopleRDD = sc.parallelize(Array(Person("seif", 65, 0), Person("amir", 40, 1)))
val peopleDF = peopleRDD.toDF
```

Creating a DataFrame - From Data Source

- Data sources supported by Spark.
  - CSV, JSON, Parquet, ORC, JDBC/ODBC connections, Plain-text files
  - Cassandra, HBase, MongoDB, AWS Redshift, XML, etc.

```
val peopleJson = spark.read.format("json").load("people.json")

val peopleCsv = spark.read.format("csv")
  .option("sep", ";")
  .option("inferSchema", "true")
  .option("header", "true")
  .load("people.csv")
```

Column

- Different ways to refer to a column.

```scala
val people = spark.read.format("json").load("people.json")

people.col("name")

col("name")

column("name")

'name

$"name"

expr("name")
```

Row

- A row is a record of data.
- They are of type `Row`.
- Rows do not have schemas.
- To access data in rows, you need to specify the position that you would like.

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

val myRow = Row("Seif", 65, 0)

myRow(0) // type Any
myRow(0).asInstanceOf[String] // String
myRow.getString(0) // String
myRow.getInt(1) // Int
```

DataFrame Transformations (1/6)

- `select` allows to do the DataFrame equivalent of SQL queries on a table of data.

```
people.select("name", "age", "id").show(2)
people.select(col("name"), expr("age + 3")).show()
people.select(expr("name AS username")).show(2)
```

DataFrame Transformations (1/6)

- `select` allows to do the DataFrame equivalent of SQL queries on a table of data.

```
people.select("name", "age", "id").show(2)
people.select(col("name"), expr("age + 3")).show()
people.select(expr("name AS username")).show(2)
```

- `filter` and `where` both filter rows.

```
people.filter(col("age") < 20).show()

people.where("age < 20").show()
```

DataFrame Transformations (1/6)

- `select` allows to do the DataFrame equivalent of SQL queries on a table of data.

```
people.select("name", "age", "id").show(2)
people.select(col("name"), expr("age + 3")).show()
people.select(expr("name AS username")).show(2)
```

- `filter` and `where` both filter rows.

```
people.filter(col("age") < 20).show()

people.where("age < 20").show()
```

- `distinct` can be used to extract unique rows.

```
people.select("name").distinct().count()
```

DataFrame Transformations (2/6)

- `withColumn` adds a new column to a DataFrame.

```
people.withColumn("teenager", expr("age < 20")).show()
```

DataFrame Transformations (2/6)

- `withColumn` adds a new column to a DataFrame.

```
people.withColumn("teenager", expr("age < 20")).show()
```

- `withColumnRenamed` renames a column.

```
people.withColumnRenamed("name", "username").columns
```

DataFrame Transformations (2/6)

- `withColumn` adds a new column to a DataFrame.

```
people.withColumn("teenager", expr("age < 20")).show()
```

- `withColumnRenamed` renames a column.

```
people.withColumnRenamed("name", "username").columns
```

- `drop` removes a column.

```
people.drop("name").columns
```

DataFrame Transformations (3/6)

- `count` returns the total number of values.

```
people.select(count("age")).show()
```

DataFrame Transformations (3/6)

- `count` returns the total number of values.

```
people.select(count("age")).show()
```

- `countDistinct` returns the number of unique groups.

```
people.select(countDistinct("name")).show()
```

DataFrame Transformations (3/6)

- `count` returns the total number of values.

```
people.select(count("age")).show()
```

- `countDistinct` returns the number of unique groups.

```
people.select(countDistinct("name")).show()
```

- `first` and `last` return the first and last value of a DataFrame.

```
people.select(first("name"), last("age")).show()
```

DataFrame Transformations (4/6)

- `min` and `max` extract the minimum and maximum values from a DataFrame.

```
people.select(min("name"), max("age"), max("id")).show()
```

DataFrame Transformations (4/6)

- `min` and `max` extract the minimum and maximum values from a DataFrame.

```
people.select(min("name"), max("age"), max("id")).show()
```

- `sum` adds all the values in a column.

```
people.select(sum("age")).show()
```

DataFrame Transformations (4/6)

- min and max extract the minimum and maximum values from a DataFrame.

```
people.select(min("name"), max("age"), max("id")).show()
```

- sum adds all the values in a column.

```
people.select(sum("age")).show()
```

- avg calculates the average.

```
people.select(avg("age")).show()
```

DataFrame Transformations (5/6)

- groupBy and agg together perform aggregations on groups.

```
people.groupBy("name").agg(count("age")).show()
```

DataFrame Transformations (5/6)

- groupBy and agg together perform aggregations on groups.

```
people.groupBy("name").agg(count("age")).show()
```

- join performs the join operation between two tables.

```
val t1 = Seq((0, "a", 0), (1, "b", 1), (2, "c", 1)).toDF("num", "name", "id")
val t2 = Seq((0, "x"), (1, "y"), (2, "z")).toDF("id", "group")

val joinExpression = t1.col("id") === t2.col("id")
var joinType = "inner"

t1.join(t2, joinExpression, joinType).show()
```

DataFrame Transformations (6/6)

- You can use udf to define new column-based functions.

```
import org.apache.spark.sql.functions.udf

val df = Seq((0, "hello"), (1, "world")).toDF("id", "text")

val upper: String => String = _.toUpperCase

val upperUDF = udf(upper)

df.withColumn("upper", upperUDF(col("text"))).show
```

DataFrame Actions

- Like RDDs, DataFrames also have their own set of actions.
- `collect`: returns an array that contains all the rows in this DataFrame.
- `count`: returns the number of rows in this DataFrame.
- `first` and `head`: returns the first row of the DataFrame.
- `show`: displays the top 20 rows of the DataFrame in a tabular form.
- `take`: returns the first n rows of the DataFrame.

Machine Learning

Machine Learning with Spark

- Spark provides support for statistics and machine learning.
  - Supervised learning
  - Unsupervised engines
  - Deep learning

Supervised Learning

- Using labeled historical data and training a model to predict the values of those labels based on various features of the data points.

- Classification (categorical values)
  - E.g., predicting disease, classifying images, ...

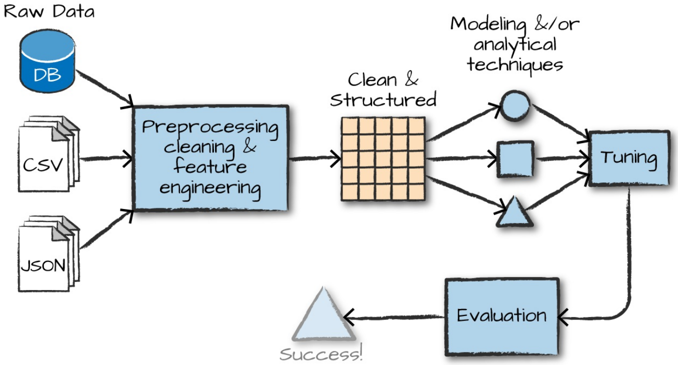
- Regression (continuous values)
  - E.g., predicting sales, predicting height, ...

Unsupervised Learning

- No label to predict.
- Trying to find patterns or discover the underlying structure in a given set of data.
  - Clustering, anomaly detection, ...

The Advanced Analytic Process

- Data collection
- Data cleaning
- Feature engineering
- Training models
- Model tuning and evaluation



What is MLlib? (1/2)

- MLlib is a package built on Spark.
- It provides interfaces for:
  - Gathering and cleaning data
  - Feature engineering and feature selection
  - Training and tuning large-scale supervised and unsupervised machine learning models
  - Using those models in production

What is MLlib? (2/2)

- MLlib consists of two packages.

What is MLlib? (2/2)

- MLlib consists of two packages.

- `org.apache.spark.mllib`
  - Uses RDDs
  - It is in maintenance mode (only receives bug fixes, not new features)

What is MLlib? (2/2)

- MLlib consists of two packages.
- `org.apache.spark.mllib`
  - Uses RDDs
  - It is in maintenance mode (only receives bug fixes, not new features)
- `org.apache.spark.ml`
  - Uses DataFrames
  - Offers a high-level interface for building machine learning pipelines

When Should We Use MLlib?

- Many tools for performing machine learning on a single machine, e.g., scikit-learn and TensorFlow.

When Should We Use MLlib?

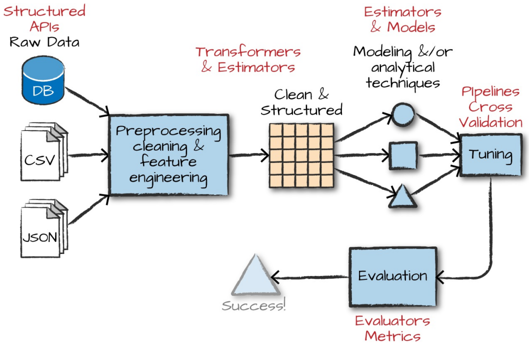
- Many tools for performing machine learning on a single machine, e.g., scikit-learn and TensorFlow.
- These single-machine tools are usually complementary to MLlib.

When Should We Use MLlib?

- Many tools for performing machine learning on a single machine, e.g., scikit-learn and TensorFlow.
- These single-machine tools are usually complementary to MLlib.
- Take advantage of Spark, when you hit scalability issues.
    - Use Spark for preprocessing and feature generation, before giving data to single-machine learning libraries.
    - Use Spark, when input data or model size become too difficult to put on one machine.

High-Level MLlib Concepts

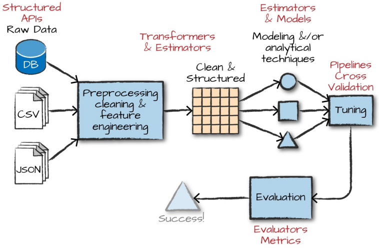
- ML Pipelines (`spark.ml`) provide a uniform set of high-level APIs built on top of DataFrames to create machine learning pipelines.
- Main pipeline components: transformers and estimators

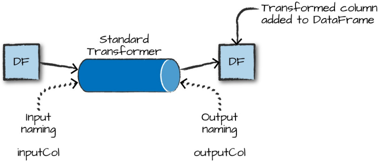


Transformers

- Transformers take a DataFrame as input and produce a new DataFrame as output.
- The class `Transformer` implements a method `transform()` that converts one DataFrame into another.

```
// transformer: DataFrame =[transform]=> DataFrame

transform(dataset: DataFrame): DataFrame
```

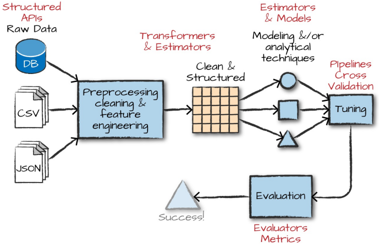




Estimators

- Estimator is an abstraction of a learning algorithm that fits a model on a dataset.
- The class `Estimator` implements a method `fit()`, which accepts a DataFrame and produces a Model (`Transformer`).

```
// estimator: DataFrame =[fit]=> Model

fit(dataset: DataFrame): M
```



Pipeline

- Pipeline is a sequence of algorithms to process and learn from data.

Pipeline

- Pipeline is a sequence of algorithms to process and learn from data.

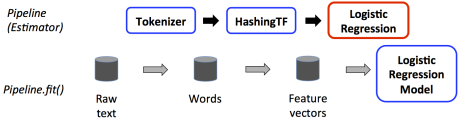
- E.g., a text document processing workflow might include several stages:
  - Split each document's text into words.
  - Convert each document's words into a numerical feature vector.
  - Learn a prediction model using the feature vectors and labels.

Pipeline

- Pipeline is a sequence of algorithms to process and learn from data.
- E.g., a text document processing workflow might include several stages:
  - Split each document's text into words.
  - Convert each document's words into a numerical feature vector.
  - Learn a prediction model using the feature vectors and labels.
- MLlib represents such a workflow as a `Pipeline`, which is a sequence of stages.
  - Each stage is either a `Transformer` or an `Estimator`.

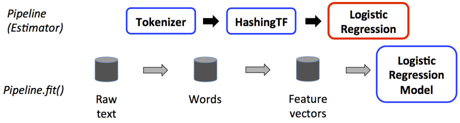
How Does Pipeline Work? (1/3)

- Stages of a pipeline run in order.
- The input DataFrame is transformed as it passes through each stage.
  - Calls `transform()` for `Transformer` stages
  - Calls `fit()` for `Estimator` stages



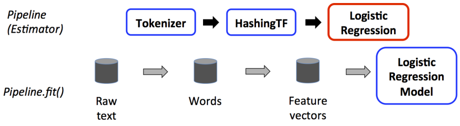
How Does Pipeline Work? (1/3)

- Stages of a pipeline run in order.
- The input DataFrame is transformed as it passes through each stage.
  - Calls `transform()` for `Transformer` stages
  - Calls `fit()` for `Estimator` stages
- E.g., a Pipeline with three stages: `Tokenizer` and `HashingTF` are Transformers, and `LogisticRegression` is an Estimator.



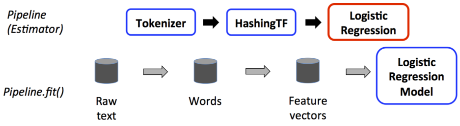
How Does Pipeline Work? (2/3)

- `Pipeline.fit()`: is called on the original DataFrame
  - DataFrame with raw text documents and labels



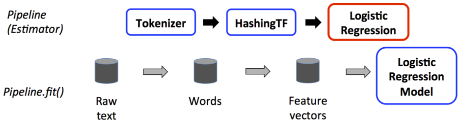
How Does Pipeline Work? (2/3)

- `Pipeline.fit()`: is called on the original DataFrame
  - DataFrame with raw text documents and labels
- `Tokenizer.transform()`: splits the raw text documents into words
  - Adds a new column with words to the DataFrame



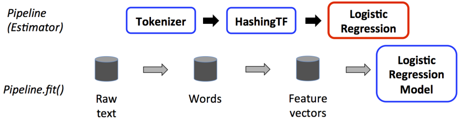
How Does Pipeline Work? (2/3)

- `Pipeline.fit()`: is called on the original DataFrame
  - DataFrame with raw text documents and labels
- `Tokenizer.transform()`: splits the raw text documents into words
  - Adds a new column with words to the DataFrame
- `HashingTF.transform()`: converts the words column into feature vectors
  - Adds new column with those vectors to the DataFrame



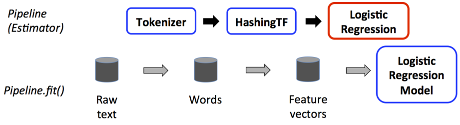
How Does Pipeline Work? (2/3)

- `Pipeline.fit()`: is called on the original DataFrame
  - DataFrame with raw text documents and labels
- `Tokenizer.transform()`: splits the raw text documents into words
  - Adds a new column with words to the DataFrame
- `HashingTF.transform()`: converts the words column into feature vectors
  - Adds new column with those vectors to the DataFrame
- `LogisticRegression.fit()`: produces a model (`LogisticRegressionModel`).

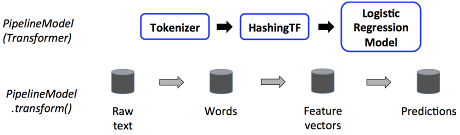


How Does Pipeline Work? (3/3)

- A `Pipeline` is an `Estimator` (`DataFrame =[fit]=> Model`).



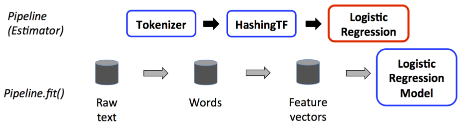
`Pipeline`



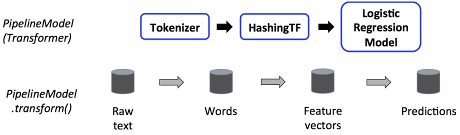
`PipelineModel`

How Does Pipeline Work? (3/3)

- A `Pipeline` is an `Estimator` (`DataFrame =[fit]=> Model`).
- After a `Pipeline`'s `fit()` runs, it produces a `PipelineModel`.



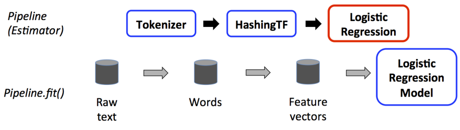
`Pipeline`



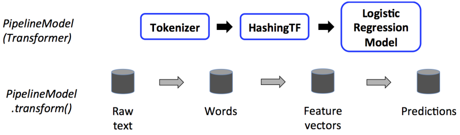
`PipelineModel`

How Does Pipeline Work? (3/3)

- A `Pipeline` is an `Estimator` (`DataFrame =[fit]=> Model`).
- After a `Pipeline`'s `fit()` runs, it produces a `PipelineModel`.
- `PipelineModel` is a `Transformer` (`DataFrame =[transform]=> DataFrame`).



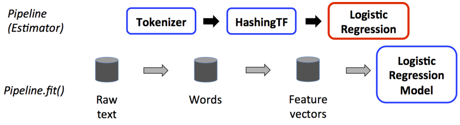
`Pipeline`



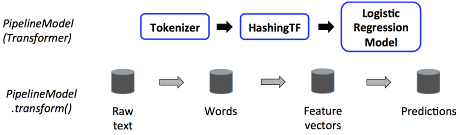
`PipelineModel`

How Does Pipeline Work? (3/3)

- A `Pipeline` is an `Estimator` (`DataFrame =[fit]=> Model`).
- After a `Pipeline`'s `fit()` runs, it produces a `PipelineModel`.
- `PipelineModel` is a `Transformer` (`DataFrame =[transform]=> DataFrame`).
- The `PipelineModel` is used at test time.



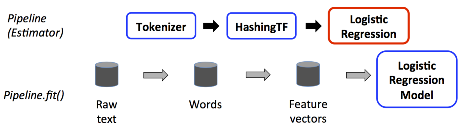
`Pipeline`



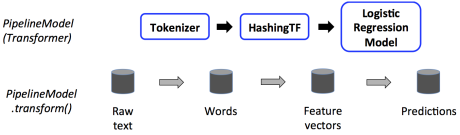
`PipelineModel`

How Does Pipeline Work? (3/3)

- A `Pipeline` is an `Estimator` (`DataFrame =[fit]=> Model`).
- After a `Pipeline`'s `fit()` runs, it produces a `PipelineModel`.
- `PipelineModel` is a `Transformer` (`DataFrame =[transform]=> DataFrame`).
- The `PipelineModel` is used at test time.
- The `PipelineModel` has the same number of stages as the original `Pipeline`.
  - All `Estimators` in the original `Pipeline` have become `Transformers`.



`Pipeline`



`PipelineModel`

Parameters

- MLlib `Estimator`s and `Transformer`s use a uniform API for specifying parameters.

Parameters

- MLlib `Estimators` and `Transformers` use a uniform API for specifying parameters.
- `Param`: a named parameter
- `ParamMap`: a set of (parameter, value) pairs

Parameters

- MLlib `Estimators` and `Transformers` use a uniform API for specifying parameters.
- `Param`: a named parameter
- `ParamMap`: a set of (parameter, value) pairs
- Two ways to pass parameters to an algorithm:
  1. Set parameters for an instance, e.g., `ls.setMaxIter(10)`
  2. Pass a `ParamMap` to `fit()` or `transform()`.

Example - Input DataFrame (1/2)

- Make a DataFrame of the type `Article`.

```
import spark.implicits._

case class Article(id: Long, topic: String, text: String)

val articles = Seq(Article(0, "sci.math", "Hello, Math!"),
                   Article(1, "alt.religion", "Hello, Religion!"),
                   Article(2, "sci.physics", "Hello, Physics!"),
                   Article(3, "sci.math", "Hello, Math Revised!"),
                   Article(4, "sci.math", "Better Math"),
                   Article(5, "alt.religion", "TGIF")).toDF

articles.show
```

Example - Input DataFrame (2/2)

- Add a new column `label` to the DataFrame.
- `udf` is a feature of Spark SQL to define new Column-based functions.

```
val topic2Label: Boolean => Double = x => if (x) 1 else 0

val toLabel = udf(topic2Label)

val labelled = articles.withColumn("label", toLabel($"topic".like("sci%"))).cache
```

Example - Transformers (1/2)

- Break each sentence into individual terms (words).

```
import org.apache.spark.ml.feature.Tokenizer
import org.apache.spark.ml.feature.RegexTokenizer

val tokenizer = new RegexTokenizer().setInputCol("text").setOutputCol("words")

val tokenized = tokenizer.transform(labelled)

tokenized.show
```

Example - Transformers (2/2)

- Takes a set of words and converts them into fixed-length feature vector.
    - 5000 in our example
- Uses a hash function to map each word into an index in the feature vector.
- Then computes the term frequencies based on the mapped indices.

```
import org.apache.spark.ml.feature.HashingTF

val hashingTF = new HashingTF().setInputCol(tokenizer.getOutputCol)
                               .setOutputCol("features")
                               .setNumFeatures(5000)

val hashed = hashingTF.transform(tokenized)

hashed.show
```

Example - Estimator

```
val Array(trainDF, testDF) = hashed.randomSplit(Array(0.8, 0.2))

trainDF.show

testDF.show
```

```
import org.apache.spark.ml.classification.LogisticRegression

val lr = new LogisticRegression().setMaxIter(20).setRegParam(0.01)

val model = lr.fit(trainDF)

val pred = model.transform(testDF).select("topic", "label", "prediction")

pred.show
```

Example - Pipeline

```
val Array(trainDF2, testDF2) = labelled.randomSplit(Array(0.8, 0.2))

trainDF2.show

testDF2.show
```

```
import org.apache.spark.ml.{Pipeline, PipelineModel}

val pipeline = new Pipeline().setStages(Array(tokenizer, hashingTF, lr))

val model2 = pipeline.fit(trainDF2)

val pred = model2.transform(testDF2).select("topic", "label", "prediction")

pred.show
```

Example - ParamMap

```scala
// set parameters using setter methods.
val lr = new LogisticRegression()

lr.setMaxIter(10).setRegParam(0.01)
```

```scala
// specify parameters using a ParamMap
val lr = new LogisticRegression()

val paramMap = ParamMap(lr.maxIter -> 20)
  .put(lr.maxIter, 30)  // specify one Param
  .put(lr.regParam -> 0.1, lr.threshold -> 0.55)  // specify multiple Params

val model = lr.fit(training, paramMap)
```

Low-Level Data Types - Local Vector

- Stored on a single machine
- Dense and sparse
  - Dense (1.0, 0.0, 3.0): [1.0, 0.0, 3.0]
  - Sparse (1.0, 0.0, 3.0): (3, [0, 2], [1.0, 3.0])

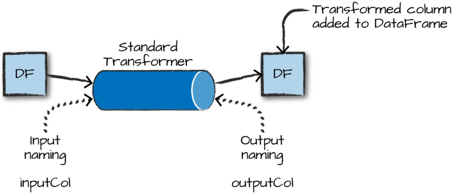
```scala
import org.apache.spark.mllib.linalg.{Vector, Vectors}

val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)

val sv1: Vector = Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0))
val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))
```

Preprocessing and Feature Engineering

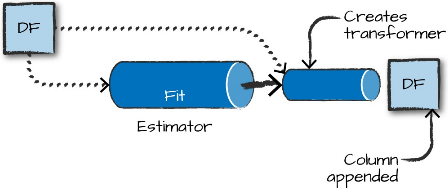
Formatting Models

- In most of classification and regression algorithms, we want to get the data.
  - A column to represent the label (`Double`).
  - A column to represent the features (`Vector`)

Transformers and Estimators



Transformer



Estimator

Transformer Properties

- All transformers require you to specify the input and output columns.
- We can set these with `setInputCol` and `setOutputCol`.

```
val tokenizer = new RegexTokenizer().setInputCol("text").setOutputCol("words")
```

Vector Assembler

- Concatenate all your features into one vector.

```scala
import org.apache.spark.ml.feature.VectorAssembler

case class Nums(val1: Long, val2: Long, val3: Long)

val numsDF = Seq(Nums(1, 2, 3), Nums(4, 5, 6), Nums(7, 8, 9)).toDF

val va = new VectorAssembler().setInputCols(Array("val1", "val2", "val3"))
                              .setOutputCol("features")

va.transform(numsDF).show()
```

MLlib Transformers

- Continuous features
- Categorical features
- Text data

MLlib Transformers

- Continuous features
- Categorical features
- Text data

Continuous Features - Bucketing

- Convert continuous features into categorical features.

```
import org.apache.spark.ml.feature.Bucketizer

val contDF = spark.range(20).selectExpr("cast(id as double)")
val bucketBorders = Array(-1.0, 5.0, 10.0, 15.0, 20.0)

val bucketer = new Bucketizer().setSplits(bucketBorders).setInputCol("id")

bucketer.transform(contDF).show()
```

Continuous Features - Scaling and Normalization

- To scale and normalize continuous data.

```scala
import org.apache.spark.ml.feature.VectorAssembler

case class Nums(val1: Long, val2: Long, val3: Long)
val numsDF = Seq(Nums(1, 2, 3), Nums(4, 5, 6), Nums(7, 8, 9)).toDF
val va = new VectorAssembler().setInputCols(Array("val1", "val2", "val3"))
                              .setOutputCol("features")
val nums = va.transform(numsDF)
```

```scala
import org.apache.spark.ml.feature.StandardScaler

val scaler = new StandardScaler().setInputCol("features").setOutputCol("scaled")
scaler.fit(nums).transform(nums).show()
```

Continuous Features - Maximum Absolute Scaler

- Scales the data by dividing each feature by the maximum absolute value in this feature (column).

```
import org.apache.spark.ml.feature.VectorAssembler

case class Nums(val1: Long, val2: Long, val3: Long)
val numsDF = Seq(Nums(1, 2, 3), Nums(4, 5, 6), Nums(7, 8, 9)).toDF
val va = new VectorAssembler().setInputCols(Array("val1", "val2", "val3"))
                              .setOutputCol("features")
val nums = va.transform(numsDF)
```

```
import org.apache.spark.ml.feature.MaxAbsScaler

val maScaler = new MaxAbsScaler().setInputCol("features").setOutputCol("mas")
maScaler.fit(nums).transform(nums).show()
```

MLlib Transformers

- Continuous features
- Categorical features
- Text data

Categorical Features - String Indexer

- Maps strings to different numerical IDs.

```
val simpleDF = spark.read.json("simple-ml.json")
```

```
import org.apache.spark.ml.feature.StringIndexer

val lblIndxr = new StringIndexer().setInputCol("lab").setOutputCol("labelInd")
val idxRes = lblIndxr.fit(simpleDF).transform(simpleDF)

idxRes.show()
```

Categorical Features - Converting Indexed Values Back to Text

- Maps back to the original values.

```
import org.apache.spark.ml.feature.IndexToString

val labelReverse = new IndexToString().setInputCol("labelInd").setOutputCol("original")

labelReverse.transform(idxRes).show()
```

Categorical Features - One-Hot Encoding

- Converts each distinct value to a boolean flag as a component in a vector.

```
val simpleDF = spark.read.json("simple-ml.json")
```

```
import org.apache.spark.ml.feature.OneHotEncoder

val lblIndxr = new StringIndexer().setInputCol("color").setOutputCol("colorInd")
val colorLab = lblIndxr.fit(simpleDF).transform(simpleDF.select("color"))
val ohe = new OneHotEncoder().setInputCol("colorInd").setOutputCol("one-hot")
ohe.transform(colorLab).show()

// Since there are three values, the vector is of length 2 and the mapping is as follows:
// 0  -> 10, (2,[0],[1.0])
// 1  -> 01, (2,[1],[1.0])
// 2  -> 00, (2,[],[])
// (2,[0],[1.0]) means a vector of length 2 with 1.0 at position 0 and 0 elsewhere.
```

MLlib Transformers

- Continuous features
- Categorical features
- Text data

Text Data - Tokenizing Text

- Converting free-form text into a list of tokens or individual words.

```
val sales = spark.read.format("csv").option("header", "true").load("sales.csv")
    .where("Description IS NOT NULL")

sales.show(false)
```

```
import org.apache.spark.ml.feature.Tokenizer

val tkn = new Tokenizer().setInputCol("Description").setOutputCol("DescOut")
val tokenized = tkn.transform(sales.select("Description"))
tokenized.show(false)
```

Text Data - Removing Common Words

- Filters stop words, such as "the", "and", and "but".

```
import org.apache.spark.ml.feature.StopWordsRemover

val df = spark.createDataFrame(Seq((0, Seq("I", "saw", "the", "red", "balloon")),
    (1, Seq("Mary", "had", "a", "little", "lamb")))).toDF("id", "raw")

val englishStopWords = StopWordsRemover.loadDefaultStopWords("english")

val stops = new StopWordsRemover().setStopWords(englishStopWords)
    .setInputCol("raw").setOutputCol("WithoutStops")

stops.transform(df).show(false)
```

Text Data - Converting Words into Numerical Representations

- Counts instances of words in word features.
- Treats every row as a document, every word as a term, and the total collection of all terms as the vocabulary.

```
import org.apache.spark.ml.feature.CountVectorizer

val df = spark.createDataFrame(Seq((0, Array("a", "b", "c")),
    (1, Array("a", "b", "b", "c", "a")))).toDF("id", "words")

val cvModel = new CountVectorizer().setInputCol("words").setOutputCol("features")
    .setVocabSize(3).setMinDF(2)

val fittedCV = cvModel.fit(df)

fittedCV.transform(df).show(false)
```

Text Data - TF-IDF (1/2)

- TF-IDF: term frequency - inverse document frequency
- Measures how often a word occurs in each document, weighted according to how many documents that word occurs in.
- Words that occur in a few documents are given more weight than words that occur in many documents.

Text Data - TF-IDF (2/2)

```
import org.apache.spark.ml.feature.{HashingTF, IDF, Tokenizer}

val sentenceData = spark.createDataFrame(Seq((0.0, "Hi I heard about Spark"),
    (0.0, "I wish Java could use case classes"),
    (1.0, "Logistic regression models are neat")))
    .toDF("label", "sentence")

val tokenizer = new Tokenizer().setInputCol("sentence").setOutputCol("words")
val wordsData = tokenizer.transform(sentenceData)

val hashingTF = new HashingTF().setInputCol("words").setOutputCol("rawFeatures")
    .setNumFeatures(20)
val featurizedData = hashingTF.transform(wordsData)

val idf = new IDF().setInputCol("rawFeatures").setOutputCol("features")
val idfModel = idf.fit(featurizedData)

val rescaledData = idfModel.transform(featurizedData)
rescaledData.select("label", "features").show(false)
```

Summary

Summary

- Spark: RDD
- Spark SQL: DataFrame
- MLlib
  - Transformers and Estimators
  - Pipeline
  - Feature engineering

References

- Matei Zaharia et al., Spark - The Definitive Guide, (Ch. 24 and 25)

Questions?