Spark

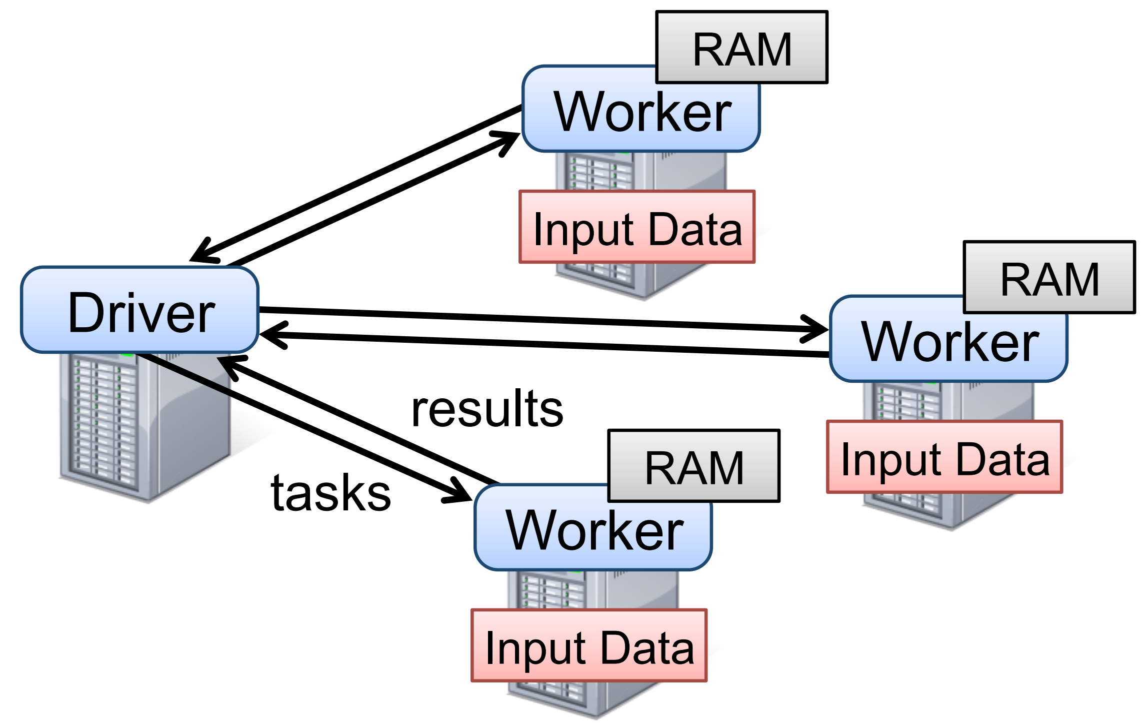
Part 3:
Data frames, SQL.

Dario Colazzo

Credits: Amir H. Payberah

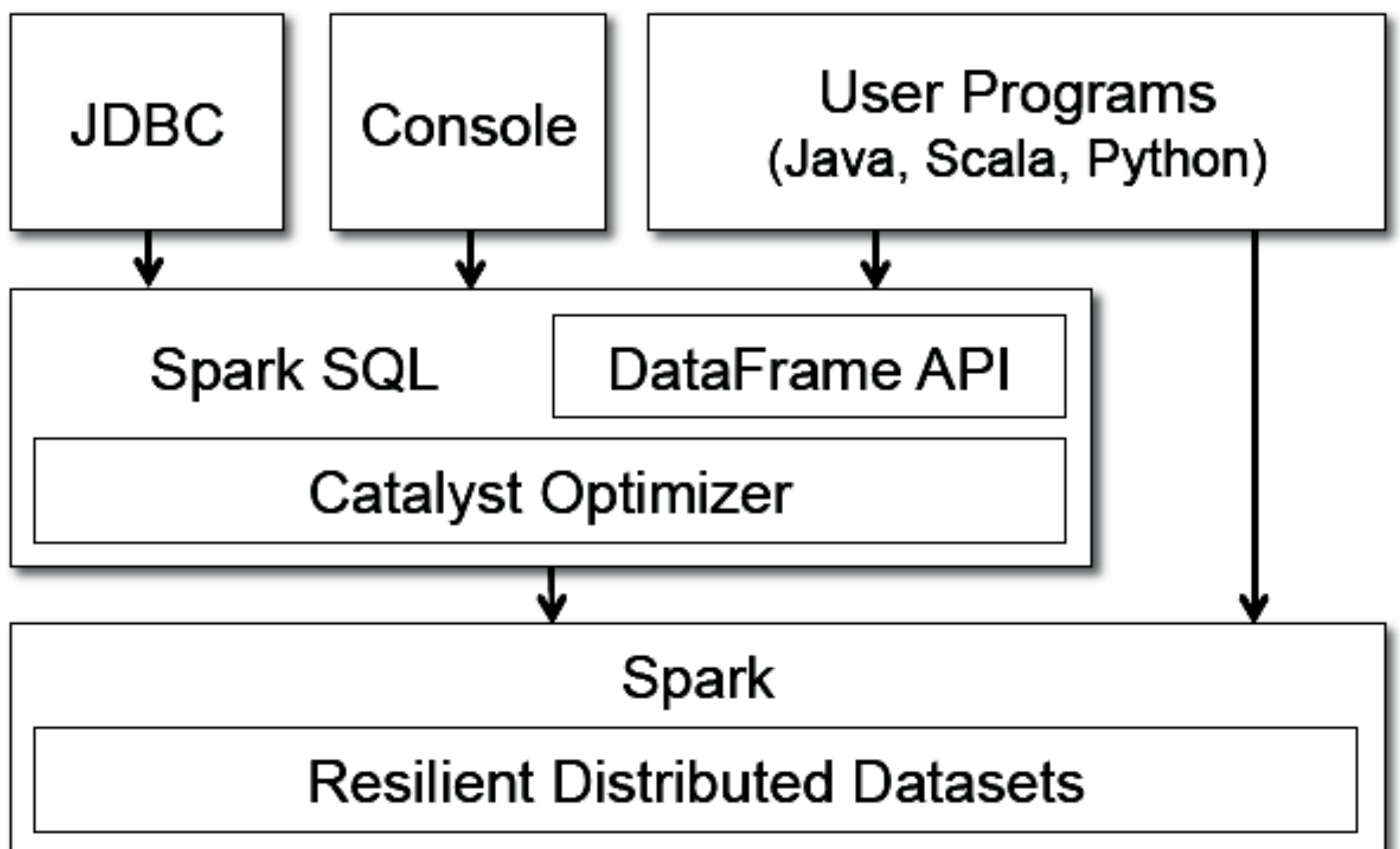
Spark programming model

- RDDs : collection of element values distributed over the cluster, mainly in main-memory (RAM)
- Transformations : lazy operators that create new RDDs from RDDs.
- Actions : lunch a computation and return a value to the program driver or write data to the external storage



Dataframes

Dataframes and Spark SQL

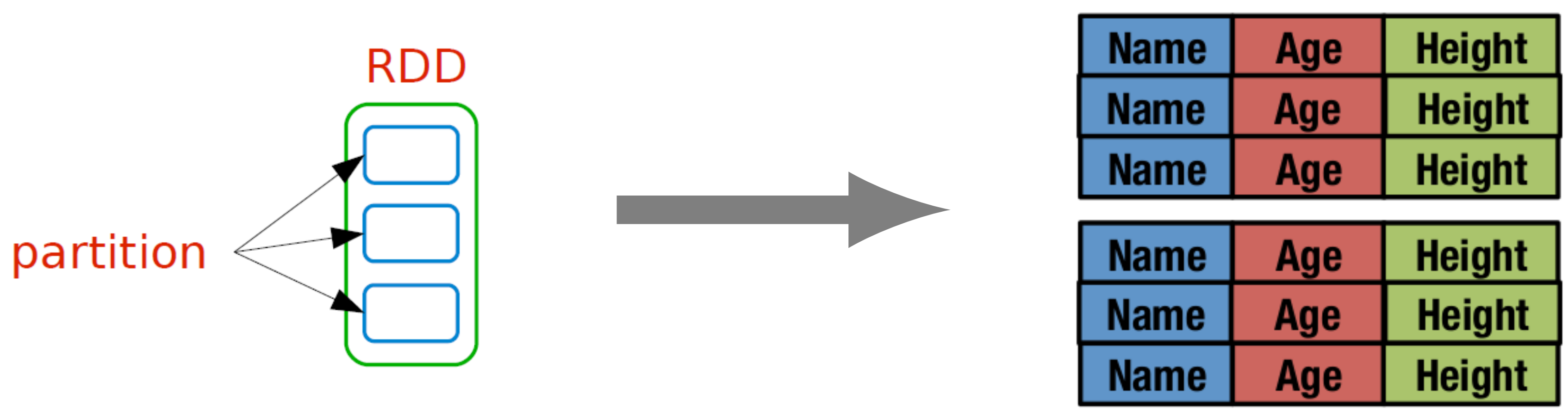


Dataframe

- A Dataframe is a distributed collection of rows
- Homogeneous schema
- Somewhat equivalent to a table in a relational database.

Adding schema to RDDs

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



Creating DataFrames

```
scala> val spark = SparkSession
           .builder()
           .appName("Spark SQL basic example")
           .config("abc", "cdf")
           .getOrCreate()

scala> import spark.implicits._
import spark.implicits._

val df = spark
    .read
    .json("/usr/local/Cellar/apache-spark/2.4.0/libexec/
    examples/src/main/resources/people.json")
```

Local path

https://databricks.com/blog/2016/08/15/how-to-use-sparksession-in-apache-spark-2-0.html

Creating DataFrames

```
scala> df.show()
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+


scala>
```

Using DataFrames

```
scala>  df.select($"name").show()
+-------+
|   name|
+-------+
|Michael|
|   Andy|
| Justin|
+-------+
scala> df.select($"name", $"age" + 1).show()

+-------+---------+
|   name|(age + 1)|
+-------+---------+
|Michael|     null|
|   Andy|       31|
| Justin|       20|
+-------+---------+
```

Select everybody, but increment the age by 1

Using DataFrames

```
scala> df.filter($"age" > 21).show()
+---+----+
|age|name|
+---+----+
| 30|Andy|
+---+----+
scala> df.groupBy($"age").count().show()
+----+-----+
| age|count|
+--+-----+
|  19|    1|
|null|    1|
|  30|    1|
+----+-----+
```

SQL on DataFrames

Register the DataFrame as a SQL temporary view

```
scala> df.createOrReplaceTempView("people")

scala> val sqlDF = spark.sql("SELECT * FROM people")

scala> sqlDF.show()
+----+-------+
| age|   name|
+——+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+


scala>
```

Converting RDDs into DataFrames

```
scala> val spark = SparkSession
          .builder()
          .appName("Spark SQL basic example”)
          .config("abc", “cdf")
          .getOrCreate()

scala> import spark.implicits._
import spark.implicits._
```

Converting RDDs into DataFrames

```
scala> val data = Array(("a",1), ("b",2) , ("a",3) , ("c",4), ("b",5))
data: Array[(String, Int)] = Array((a,1), (b,2), (a,3), (c,4), (b,5))

scala> val rdd = sc.parallelize(data)
rdd: org.apache.spark.rdd.RDD[(String, Int)] =
ParallelCollectionRDD[47] at parallelize at <console>:35

scala> val rdd_1 = rdd.reduceByKey((a, b) => a + b)

scala> rdd_1.collect
res13: Array[(String, Int)] = Array((a,4), (b,7), (c,4))



scala>
```

Converting RDDs into DataFrames

```
scala> rdd_1.collect
res13: Array[(String, Int)] = Array((a,4), (b,7), (c,4))

scala> val myDf = rdd_1.toDF("name", "val")
myDf: org.apache.spark.sql.DataFrame = [name: string, val: int]

scala> myDf.show()
+----+---+
|name|val|
+----+---+
|   a|  4|
|   b|  7|
|   c|  4|
+----+---+
scala> myDf.printSchema
root
 |-- name: string (nullable = true)
 |-- val: integer (nullable = false)
scala>
```

More details in this nice post

https://indatalabs.com/blog/convert-spark-rdd-to-dataframe-dataset

Conclusion

- Supports on a variety of data sources.

and more ...

- A DataFrame can be operated on as normal RDDs or as a temporary table.
- Registering a DataFrame as a table allows you to run SQL queries over its data.
- More details on :

http://spark.apache.org/docs/latest/sql-programming-guide.html#starting-point-sparksession

Datasets

Datasets

- Datasets offers a compromise/mix between RDD and DataFrames
    - They can be used to run SQL operations
- A Dataset is a collection of JVM objects, so objects are strongly typed

First example

```
scala> val wordsRDD = sc.parallelize(Seq("Spark I am your father", "May the spark be with
you", "Spark I am your father”))

wordsRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[59] at parallelize at
<console>:33



scala> val wordsDataset = wordsRDD.toDS()
wordsDataset: org.apache.spark.sql.Dataset[String] = [value: string]

scala> wordsDataset.printSchema
root
 |-- value: string (nullable = true)



scala> wordsDataset.show()
+--------------------+
|               value|
+--------------------+
|Spark I am your f...|
|May the spark be ...|
|Spark I am your f...|
+--------------------+
```

First example

```
scala> val groupedDataset = wordsDataset.flatMap(_.toLowerCase.split(" "))
                                  .filter(_ != "")
                                  .groupBy("value")

scala> val countsDataset = groupedDataset.count()


scala> countsDataset.show()

scala> groupedDataset.count().show
+------+-----+
| value|count|
+------+-----+
|father|    2|
|   you|    1|
|  with|    1|
|    be|    1|
|  your|    2|
|   may|    1|
| spark|    3|
```

RDD & Datasets

RDDs

```
val lines = sc.textFile("/wikipedia")

val words = lines

  .flatMap(_.split(" "))

  .filter(_ != "")
```

Datasets

```
val lines = sqlContext.read.text("/wikipedia").as[String]

val words = lines

  .flatMap(_.split(" "))

  .filter(_ != "")
```

Word-count

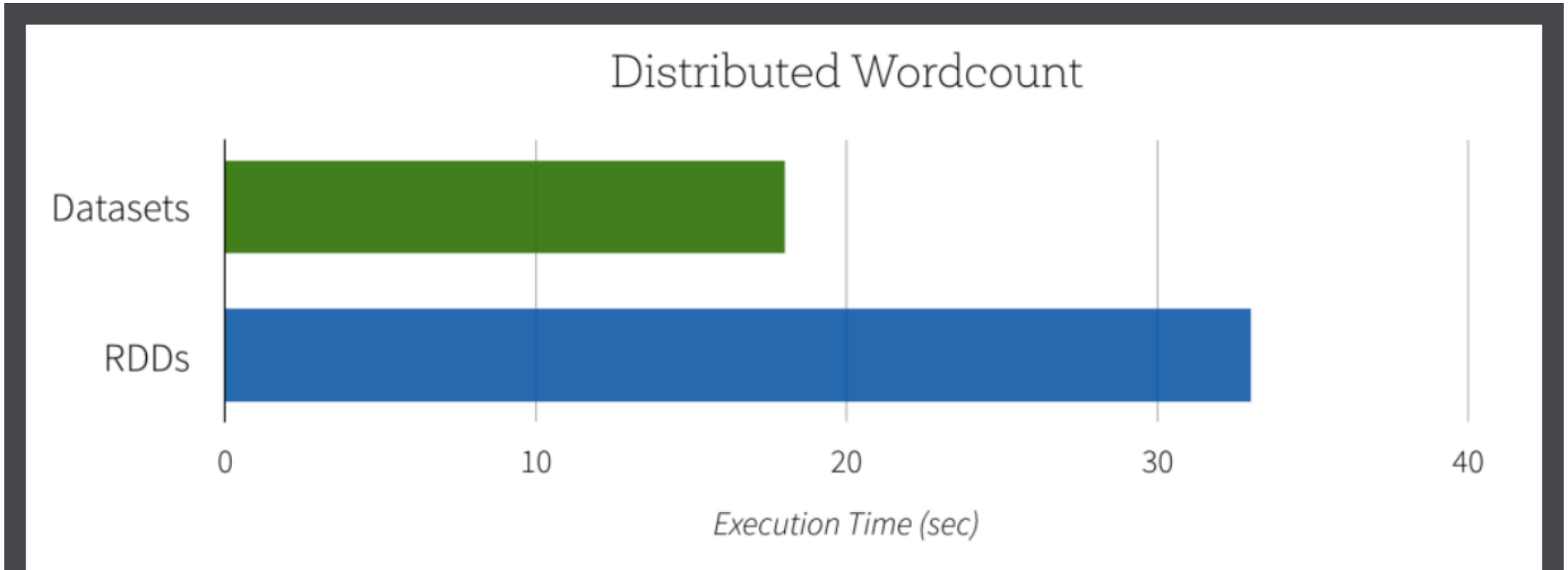
RDDs

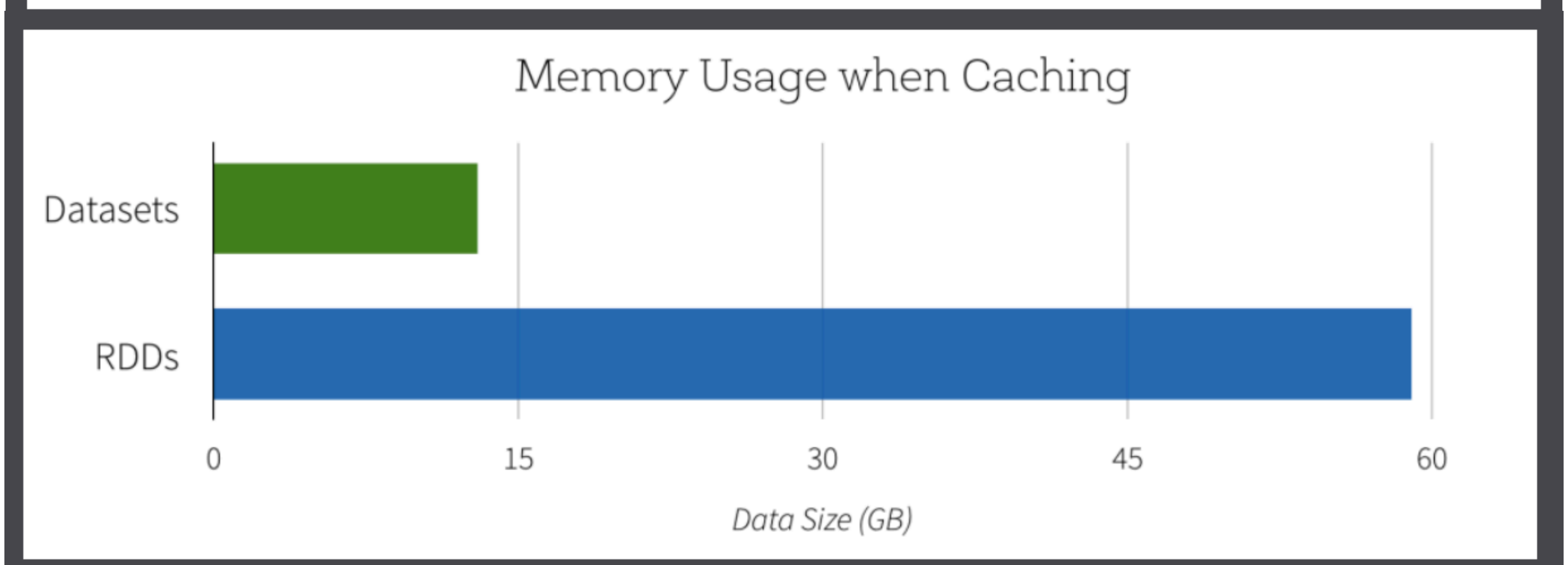
```
val counts = words

    .groupBy(_.toLowerCase)

    .map(w => (w._1, w._2.size))
```

Datasets

```
val counts = words

    .groupBy(_.toLowerCase)

    .count()
```

Performances





Readings

- Interesting post on serialisation/deserialisation
  - https://blog.xebia.fr/2017/09/27/spark-comprendre-et-corriger-lexception-task-not-serializable/
- Datasets:
  - https://spark.apache.org/docs/2.3.0/api/java/index.html?org/apache/spark/sql/Dataset.html

FETS in Spark

Spark MLib

- We will rely on Spark MLib
- Main notion it relies on: ML pipelines
- Mostly inspired by scikit-learn

Pipeline

- A pipeline chains several ML steps/stages in order to specify a ML workflow, including all the steps: preprocessing, feature extraction, model training, estimation, …

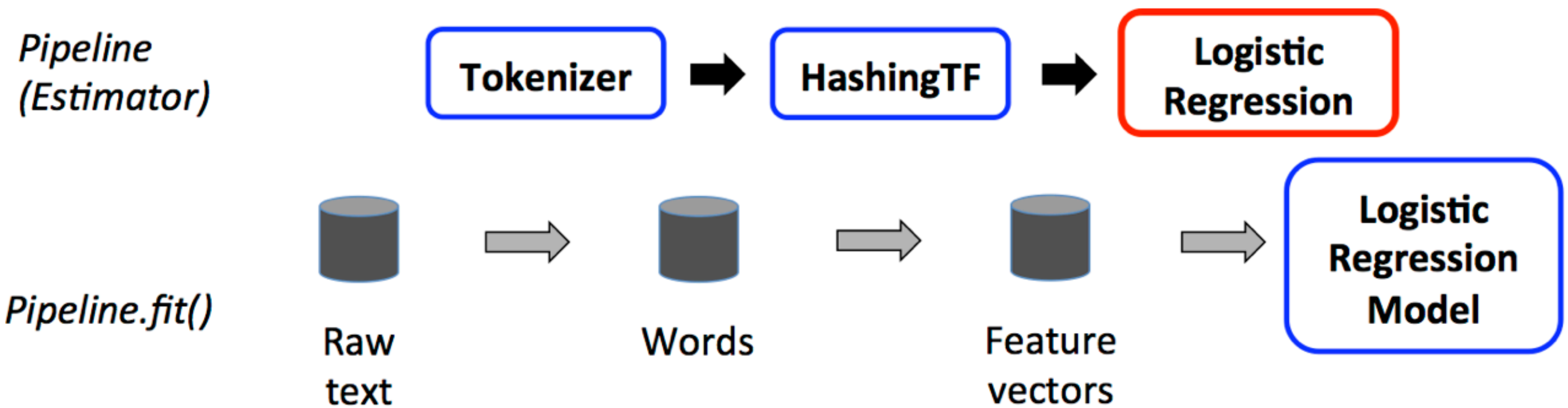
- Each step/stage is either a Transformer or an Estimator

- From Spark documentation:

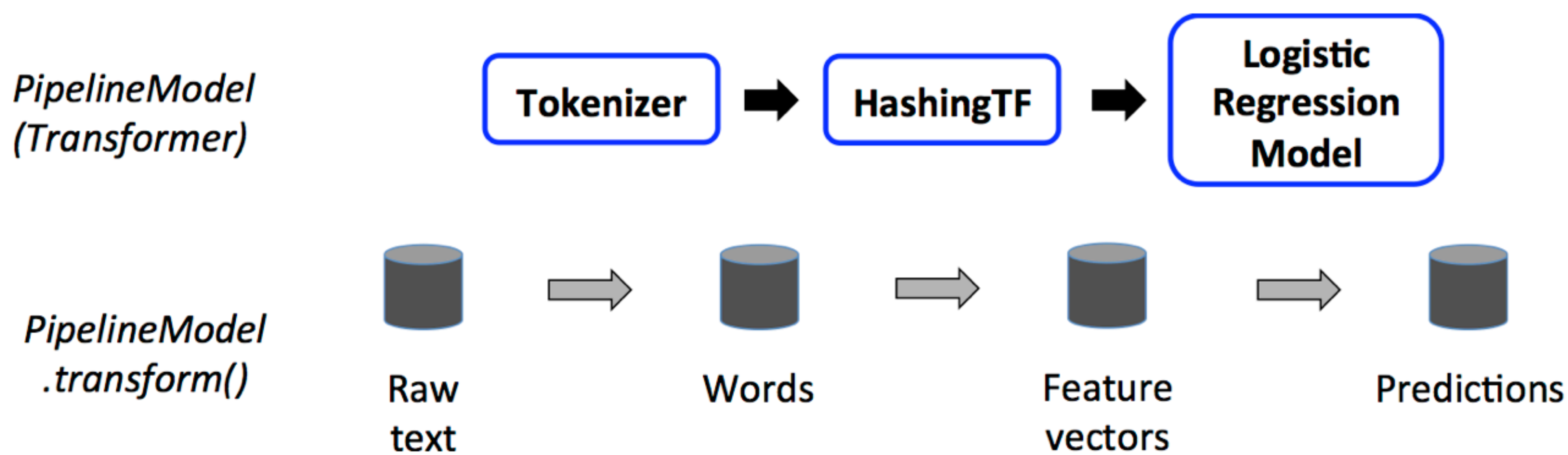
- **`Transformer`**: A `Transformer` is an algorithm which can transform one `DataFrame` into another `DataFrame`. E.g., an ML model is a `Transformer` which transforms a `DataFrame` with features into a `DataFrame` with predictions.
- **`Estimator`**: An `Estimator` is an algorithm which can be fit on a `DataFrame` to produce a `Transformer`. E.g., a learning algorithm is an `Estimator` which trains on a `DataFrame` and produces a model.

https://spark.apache.org/docs/latest/ml-pipeline.html#pipeline-components

Training Pipeline



Pipeline model



Example, a couple of interesting Transformers

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


scala>  val sentenceData = spark.createDataFrame(Seq(
     | (0, "Hi I heard about Spark about"),
     | (0, "I wish Java could case classes"),
     | (1, "Logistic regression models are neat")
     | )).toDF("label", "sentence")


scala> sentenceData.show(3,truncate=false)
+-----+-----------------------------------+
|label|sentence                           |
+-----+-----------------------------------+
|0    |Hi I heard about Spark             |
|0    |I wish Java could use case classes |
|1    |Logistic regression models are neat|
+-----+-----------------------------------+
```

HashingTF

```
scala> val tokenizer = new Tokenizer().setInputCol("sentence").setOutputCol("words")


scala> val hashingTF = new HashingTF().
     | setInputCol("words").setOutputCol("rawFeatures").setNumFeatures(100)


scala> val wordsData = tokenizer.transform(sentenceData)

scala> wordsData.show(1, truncate=false)
+-----+----------------------+----------------------------+
|label|sentence              |words                       |
+-----+----------------------+----------------------------+
|0    |Hi I heard about Spark|[hi, i, heard, about, spark]|
+-----+----------------------+----------------------------+

scala> val featurizedData = hashingTF.transform(wordsData)

scala> featurizedData.show(3,truncate=false)

+-----+----------------------------------+-----------------------------------------+---------------------------------------------+
|label|sentence                          |words                                    |rawFeatures                                  |
+-----+----------------------------------+-----------------------------------------+---------------------------------------------+
|0    |Hi I heard about Spark about      |[hi, i, heard, about, spark, about]      |(100,[56,68,73,86],[3.0,1.0,1.0,1.0])        |
|0    |I wish Java could case classes    |[i, wish, java, could, case, classes]    |(100,[7,42,56,67,80,95],[1.0,1.0,1.0,1.0,1.0,1.0])|
|1    |Logistic regression models are neat|[logistic, regression, models, are, neat]|(100,[4,59,63,71,86],[1.0,1.0,1.0,1.0,1.0])  |
+-----+----------------------------------+-----------------------------------------+---------------------------------------------+

scala> featurizedData.printSchema()
root
 |-- label: integer (nullable = false)
 |-- sentence: string (nullable = true)
 |-- words: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- rawFeatures: vector (nullable = true)
```

CountVectorizer

```
scala> import org.apache.spark.ml.feature.{CountVectorizer,Tokenizer}

scala> val tokenizer = new Tokenizer().setInputCol("sentence").setOutputCol("words")


scala> val wordsData = tokenizer.transform(sentenceData)


scala> wordsData.printSchema()
root
 |-- label: integer (nullable = false)
 |-- sentence: string (nullable = true)
 |-- words: array (nullable = true)
 |    |-- element: string (containsNull = true)

scala> val countVectorizer =
        new CountVectorizer().setInputCol("words").setOutputCol("features").setVocabSize(20)


scala> val featurizedDataModel = countVectorizer.fit(wordsData)


scala> featurizedDataModel.transform(wordsData).show(3, truncate=false)
+-----+----------------------------------+------------------------------------------+---------------------------------------------
|label|sentence                          |words                                     |features
+-----+----------------------------------+------------------------------------------+---------------------------------------------
|0    |Hi I heard about Spark            |[hi, i, heard, about, spark]              |(16,[0,7,8,9,13],[1.0,1.0,1.0,1.0,1.0])
|0    |I wish Java could use case classes|[i, wish, java, could, use, case, classes]|(16,[0,1,4,5,10,12,15],[1.0,1.0,1.0,1.0,1.0,1.0,1.0]
|1    |Logistic regression models are neat|[logistic, regression, models, are, neat] |(16,[2,3,6,11,14],[1.0,1.0,1.0,1.0,1.0])
+-----+----------------------------------+------------------------------------------+---------------------------------------------


scala> featurizedDataModel.vocabulary
res46: Array[String] = Array(i, could, regression, neat, java, case, models, spark, about, hi, wish, are
classes, heard, logistic, use)
```

StringIndexer

```
id | category
----|----------
0  | a
1  | b
2  | c
3  | a
4  | a
5  | c
```

```
id | category | categoryIndex
----|----------|---------------
0  | a        | 0.0
1  | b        | 2.0
2  | c        | 1.0
3  | a        | 0.0
4  | a        | 0.0
5  | c        | 1.0
```

StringIndexer

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

scala> val df = spark.createDataFrame(
     |   Seq((0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c"))
     | ).toDF("id", "category")

scala> val indexer = new StringIndexer().
     | setInputCol("category").
     | setOutputCol("categoryIndex")

scala> val indexed = indexer.fit(df).transform(df)

scala> indexed.show()
+---+--------+-------------+
| id|category|categoryIndex|
+---+--------+-------------+
|  0|       a|          0.0|
|  1|       b|          2.0|
|  2|       c|          1.0|
|  3|       a|          0.0|
|  4|       a|          0.0|
|  5|       c|          1.0|
+---+--------+-------------+
```

oneHotEncoder

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


scala>  val df = spark.createDataFrame(Seq(
         |   (0.0, 1.0),
         |   (1.0, 0.0),
         |   (2.0, 1.0),
         |   (0.0, 2.0),
         |   (0.0, 1.0),
         |   (2.0, 2.0)
         | )).toDF("categoryIndex1", "categoryIndex2")



scala> val encoder = new OneHotEncoder().
     | setInputCols(Array("categoryIndex1", "categoryIndex2")).
     | setOutputCols(Array("categoryVec1", "categoryVec2"))


scala> val model = encoder.fit(df)


scala> val encoded = model.transform(df).cache()


scala> encoded.show()
+--------------+--------------+-------------+-------------+
|categoryIndex1|categoryIndex2| categoryVec1| categoryVec2|
+--------------+--------------+-------------+-------------+
|           0.0|           1.0|(2,[0],[1.0])|(2,[1],[1.0])|
|           1.0|           0.0|(2,[1],[1.0])|(2,[0],[1.0])|
|           2.0|           1.0|    (2,[],[])|(2,[1],[1.0])|
|           0.0|           2.0|(2,[0],[1.0])|    (2,[],[])|
|           0.0|           1.0|(2,[0],[1.0])|(2,[1],[1.0])|
|           2.0|           2.0|    (2,[],[])|    (2,[],[])|
+--------------+--------------+-------------+-------------+


scala>
```

```
scala> encoded.printSchema()
root
 |-- categoryIndex1: double (nullable = false)
 |-- categoryIndex2: double (nullable = false)
 |-- categoryVec1: vector (nullable = true)
 |-- categoryVec2: vector (nullable = true)


scala>
```

VectorAssembler

```
scala> encoded.printSchema()
root
 |-- categoryIndex1: double (nullable = false)
 |-- categoryIndex2: double (nullable = false)
 |-- categoryVec1: vector (nullable = true)
 |-- categoryVec2: vector (nullable = true)


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

scala> val assembler = new VectorAssembler().
     | setInputCols(Array("categoryVec1", "categoryVec2")).
     | setOutputCol("features")

scala> val output = assembler.transform(encoded)


scala> output.show()

+--------------+--------------+-------------+-------------+-----------------+
|categoryIndex1|categoryIndex2| categoryVec1| categoryVec2|         features|
+--------------+--------------+-------------+-------------+-----------------+
|           0.0|           1.0|(2,[0],[1.0])|(2,[1],[1.0])|[1.0,0.0,0.0,1.0]|
|           1.0|           0.0|(2,[1],[1.0])|(2,[0],[1.0])|[0.0,1.0,1.0,0.0]|
|           2.0|           1.0|    (2,[],[])|(2,[1],[1.0])|    (4,[3],[1.0])|
|           0.0|           2.0|(2,[0],[1.0])|    (2,[],[])|    (4,[0],[1.0])|
|           0.0|           1.0|(2,[0],[1.0])|(2,[1],[1.0])|[1.0,0.0,0.0,1.0]|
|           2.0|           2.0|    (2,[],[])|    (2,[],[])|        (4,[],[])|
+--------------+--------------+-------------+-------------+-----------------+
```

VectorAssembler - how to avoid information loss

```
scala> encoded.printSchema()
root
 |-- categoryIndex1: double (nullable = false)
 |-- categoryIndex2: double (nullable = false)
 |-- categoryVec1: vector (nullable = true)
 |-- categoryVec2: vector (nullable = true)


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


scala> val encoder = new OneHotEncoderEstimator().
             | setInputCols(Array("categoryIndex1", "categoryIndex2")).
             | setOutputCols(Array("categoryVec1", "categoryVec2")).setDropLast(false)


scala>  val model = encoder.fit(df)
model: org.apache.spark.ml.feature.OneHotEncoderModel = oneHotEncoder_9715ada877c5


scala> val encoded = model.transform(df)
encoded: org.apache.spark.sql.DataFrame = [categoryIndex1: double, categoryIndex2: double ... 2 more
fields]

scala> encoded.show()
+--------------+--------------+-------------+-------------+
|categoryIndex1|categoryIndex2| categoryVec1| categoryVec2|
+--------------+--------------+-------------+-------------+
|           0.0|           1.0|(3,[0],[1.0])|(3,[1],[1.0])|
|           1.0|           0.0|(3,[1],[1.0])|(3,[0],[1.0])|
|           2.0|           1.0|(3,[2],[1.0])|(3,[1],[1.0])|
|           0.0|           2.0|(3,[0],[1.0])|(3,[2],[1.0])|
|           0.0|           1.0|(3,[0],[1.0])|(3,[1],[1.0])|
|           2.0|           2.0|(3,[2],[1.0])|(3,[2],[1.0])|
+--------------+--------------+-------------+-------------+


scala>
```

VectorAssembler - how to avoid information loss

Same as before

```
scala> val output = assembler.transform(encoded)


scala> output.show()
+--------------+--------------+-------------+-------------+-------------------+
|categoryIndex1|categoryIndex2| categoryVec1| categoryVec2|           features|
+--------------+--------------+-------------+-------------+-------------------+
|           0.0|           1.0|(3,[0],[1.0])|(3,[1],[1.0])|(6,[0,4],[1.0,1.0])|
|           1.0|           0.0|(3,[1],[1.0])|(3,[0],[1.0])|(6,[1,3],[1.0,1.0])|
|           2.0|           1.0|(3,[2],[1.0])|(3,[1],[1.0])|(6,[2,4],[1.0,1.0])|
|           0.0|           2.0|(3,[0],[1.0])|(3,[2],[1.0])|(6,[0,5],[1.0,1.0])|
|           0.0|           1.0|(3,[0],[1.0])|(3,[1],[1.0])|(6,[0,4],[1.0,1.0])|
|           2.0|           2.0|(3,[2],[1.0])|(3,[2],[1.0])|(6,[2,5],[1.0,1.0])|
+--------------+--------------+-------------+-------------+-------------------+


scala>
```

Let's now obtain an RDD with arrays corresponding to assembled vectors

```
scala> output.show()
+--------------+--------------+-------------+-------------+------------------+
|categoryIndex1|categoryIndex2| categoryVec1| categoryVec2|          features|
+--------------+--------------+-------------+-------------+------------------+
|           0.0|           1.0|(3,[0],[1.0])|(3,[1],[1.0])|(6,[0,4],[1.0,1.0])|
|           1.0|           0.0|(3,[1],[1.0])|(3,[0],[1.0])|(6,[1,3],[1.0,1.0])|
|           2.0|           1.0|(3,[2],[1.0])|(3,[1],[1.0])|(6,[2,4],[1.0,1.0])|
|           0.0|           2.0|(3,[0],[1.0])|(3,[2],[1.0])|(6,[0,5],[1.0,1.0])|
|           0.0|           1.0|(3,[0],[1.0])|(3,[1],[1.0])|(6,[0,4],[1.0,1.0])|
|           2.0|           2.0|(3,[2],[1.0])|(3,[2],[1.0])|(6,[2,5],[1.0,1.0])|
+--------------+--------------+-------------+-------------+------------------+
```

Attention: if you have both Sparse and Dense vectors, just use Vector instead of SparseVecor

```
scala> import org.apache.spark.ml.linalg.SparseVector

scala> val toArr: Any => Array[Double] = _.asInstanceOf[SparseVector].toArray

scala> val toArrUdf = udf(toArr)


scala> val outputWithArrayFeat = output.withColumn("features_arr",toArrUdf('features)).cache()

scala> outputWithArrayFeat.show(truncate=false)
+--------------+--------------+-------------+-------------+-------------------+------------------------------+
|categoryIndex1|categoryIndex2|categoryVec1 |categoryVec2 |features           |features_arr                  |
+--------------+--------------+-------------+-------------+-------------------+------------------------------+
|0.0           |1.0           |(3,[0],[1.0])|(3,[1],[1.0])|(6,[0,4],[1.0,1.0])|[1.0, 0.0, 0.0, 0.0, 1.0, 0.0]|
|1.0           |0.0           |(3,[1],[1.0])|(3,[0],[1.0])|(6,[1,3],[1.0,1.0])|[0.0, 1.0, 0.0, 1.0, 0.0, 0.0]|
|2.0           |1.0           |(3,[2],[1.0])|(3,[1],[1.0])|(6,[2,4],[1.0,1.0])|[0.0, 0.0, 1.0, 0.0, 1.0, 0.0]|
|0.0           |2.0           |(3,[0],[1.0])|(3,[2],[1.0])|(6,[0,5],[1.0,1.0])|[1.0, 0.0, 0.0, 0.0, 0.0, 1.0]|
|0.0           |1.0           |(3,[0],[1.0])|(3,[1],[1.0])|(6,[0,4],[1.0,1.0])|[1.0, 0.0, 0.0, 0.0, 1.0, 0.0]|
|2.0           |2.0           |(3,[2],[1.0])|(3,[2],[1.0])|(6,[2,5],[1.0,1.0])|[0.0, 0.0, 1.0, 0.0, 0.0, 1.0]|
+--------------+--------------+-------------+-------------+-------------------+------------------------------+
```

Let’s now obtain an RDD with arrays corresponding to assembled vectors

```
scala> val outputWithArrayOnlyFeat = outputWithArrayFeat.select("features_arr")

scala> outputWithArrayOnlyFeat.rdd.take(1)
res11: Array[org.apache.spark.sql.Row] = Array([WrappedArray(1.0, 0.0, 0.0, 0.0, 1.0, 0.0)])

scala> outputWithArrayOnlyFeat.rdd.map(x => x(0))
res18: org.apache.spark.rdd.RDD[Any] = MapPartitionsRDD[62] at map at <console>:32

scala> import scala.collection.mutable.WrappedArray

scala> val myrdd = outputWithArrayOnlyFeat.rdd.map(x => x(0).asInstanceOf[WrappedArray[Double]].toArray[Double])
myrdd: org.apache.spark.rdd.RDD[Array[Double]] = MapPartitionsRDD[64] at map at <console>:31

scala> myrdd.take(1)
res20: Array[Array[Double]] = Array(Array(1.0, 0.0, 0.0, 0.0, 1.0, 0.0))
```

Pipelines

- Once data preparation steps and model estimator are ready you can concatenate them in a pipeline

- For instance, a pipeline for data preparation can be built and used as follows

Pipeline

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

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


scala>  val df = spark.createDataFrame(Seq(
           |   (0.0, 1.0),
           |   (1.0, 0.0),
           |   (2.0, 1.0),
           |   (0.0, 2.0),
           |   (0.0, 1.0),
           |   (2.0, 2.0)
           | )).toDF("categoryIndex1", "categoryIndex2")

scala> val encoder = new OneHotEncoder().
     | setInputCols(Array("categoryIndex1", "categoryIndex2")).
     | setOutputCols(Array("categoryVec1", "categoryVec2"))


scala> val assembler = new VectorAssembler().
     | setInputCols(Array("categoryVec1", "categoryVec2")).
     | setOutputCol(“features”)


scala> val steps: Array[org.apache.spark.ml.PipelineStage] = Array(encoder,  assembler)

scala>  import org.apache.spark.ml.Pipeline


scala> val pipeline_prep = new Pipeline().setStages(steps)

scala> val output = (pipeline_prep.fit(df)).transform(df)

scala>  output.show(truncate=false)
+--------------+--------------+-------------+-------------+-----------------+
|categoryIndex1|categoryIndex2|categoryVec1 |categoryVec2 |features         |
+--------------+--------------+-------------+-------------+-----------------+
|0.0           |1.0           |(2,[0],[1.0])|(2,[1],[1.0])|[1.0,0.0,0.0,1.0]|
|1.0           |0.0           |(2,[1],[1.0])|(2,[0],[1.0])|[0.0,1.0,1.0,0.0]|
|2.0           |1.0           |(2,[],[])    |(2,[1],[1.0])|(4,[3],[1.0])    |
|0.0           |2.0           |(2,[0],[1.0])|(2,[],[])    |(4,[0],[1.0])    |
|0.0           |1.0           |(2,[0],[1.0])|(2,[1],[1.0])|[1.0,0.0,0.0,1.0]|
|2.0           |2.0           |(2,[],[])    |(2,[],[])    |(4,[],[])        |
+--------------+--------------+-------------+-------------+-----------------+
```

Data splitting

- In order to split a dataset for training and testing you can proceed as follows

```
scala> val Array(training, test) = data.randomSplit(Array(0.75, 0.25), seed = 12345 )
```

Building a model estimator and parameters

```
scala> import org.apache.spark.ml.regression.{LinearRegression}

scala> import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

scala> val lr = new LinearRegression().setLabelCol("priceOutputVar").setFeaturesCol("features")
```

Features and label to be estimated have been already prepared

```
scala> val paramGrid = new ParamGridBuilder().
             | addGrid(lr.regParam, Array(0.1, 0.01)).
             | addGrid(lr.fitIntercept).
             | addGrid(lr.elasticNetParam, Array(0.0, 1.0)).build()
paramGrid: Array[org.apache.spark.ml.param.ParamMap] =
Array({
   linReg_41a489042a50-elasticNetParam: 0.0,
   linReg_41a489042a50-fitIntercept: true,
   linReg_41a489042a50-regParam: 0.1
}, {
   linReg_41a489042a50-elasticNetParam: 0.0,
   linReg_41a489042a50-fitIntercept: true,
   linReg_41a489042a50-regParam: 0.01
}, {
   linReg_41a489042a50-elasticNetParam: 1.0,
   linReg_41a489042a50-fitIntercept: true,
   linReg_41a489042a50-regParam: 0.1
}, {
   linReg_41a489042a50-elasticNetParam: 1.0,
   linReg_41a489042a50-fitIntercept: true,
   linReg_41a489042a50-regParam: 0.01
}, {
   linReg_41a489042a50-elasticNetParam: 0.0,
   linReg_41a489042a50-fitIntercept: false,
   linReg_41a489042a50-regParam: 0.1
}, {
   linReg_41a489042a50-elasticNetParam: 0.0,
   linReg_41a489042a50-fitIntercept: false,
   linReg_41a489042a50-regPa...
scala>
```

$$\alpha\left(\lambda\|\mathbf{w}\|_1\right) + (1-\alpha)\left(\frac{\lambda}{2}\|\mathbf{w}\|_2^2\right), \alpha \in [0,1], \lambda \geq 0$$

Elastic net regularization: you have L1 for a=1 while L2 is obtained for a=0. Lambda is the reg-param.

A complete pipeline (to be used for Diamonds regression)

```
scala> val steps: Array[org.apache.spark.ml.PipelineStage] =
                 categoricalIndexers ++ categoricalEncoders ++ Array(assembler, lr)



scala> val pipeline = new Pipeline().setStages(steps)

scala> import org.apache.spark.ml.evaluation.{RegressionEvaluator}


scala>  val tvs = new TrainValidationSplit().
             | setEstimator(pipeline).
             | setEvaluator( new RegressionEvaluator().setLabelCol(“priceOutputVar”) ).
             | setEstimatorParamMaps(paramGrid).
             | setTrainRatio(0.75)


scala> val Array(training, test) = data.randomSplit(Array(0.75, 0.25), seed = 12345)

scala> val model = tvs.fit(training)

scala> val holdout = model.transform(test).select("prediction", "priceOutputVar")
```

Regression model evaluation

Regression analysis is used when predicting a continuous output variable from a number of independent variables.

Available metrics

Metric	Definition
Mean Squared Error (MSE)	$MSE = \frac{\sum_{i=0}^{N-1} (\mathbf{y}_i - \hat{\mathbf{y}_i})^2}{N}$
Root Mean Squared Error (RMSE)	$RMSE = \sqrt{\frac{\sum_{i=0}^{N-1} (\mathbf{y}_i - \hat{\mathbf{y}_i})^2}{N}}$
Mean Absolute Error (MAE)	$MAE = \sum_{i=0}^{N-1} \lvert \mathbf{y}_i - \hat{\mathbf{y}_i} \rvert$
Coefficient of Determination (R^2)	$R^2 = 1 - \frac{MSE}{\text{VAR}(\mathbf{y}) \cdot (N-1)} = 1 - \frac{\sum_{i=0}^{N-1} (\mathbf{y}_i - \hat{\mathbf{y}_i})^2}{\sum_{i=0}^{N-1} (\mathbf{y}_i - \bar{\mathbf{y}})^2}$
Explained Variance	$1 - \frac{\text{VAR}(\mathbf{y} - \hat{\mathbf{y}})}{\text{VAR}(\mathbf{y})}$

https://spark.apache.org/docs/2.2.0/mllib-evaluation-metrics.html

Regression metrics

………

………

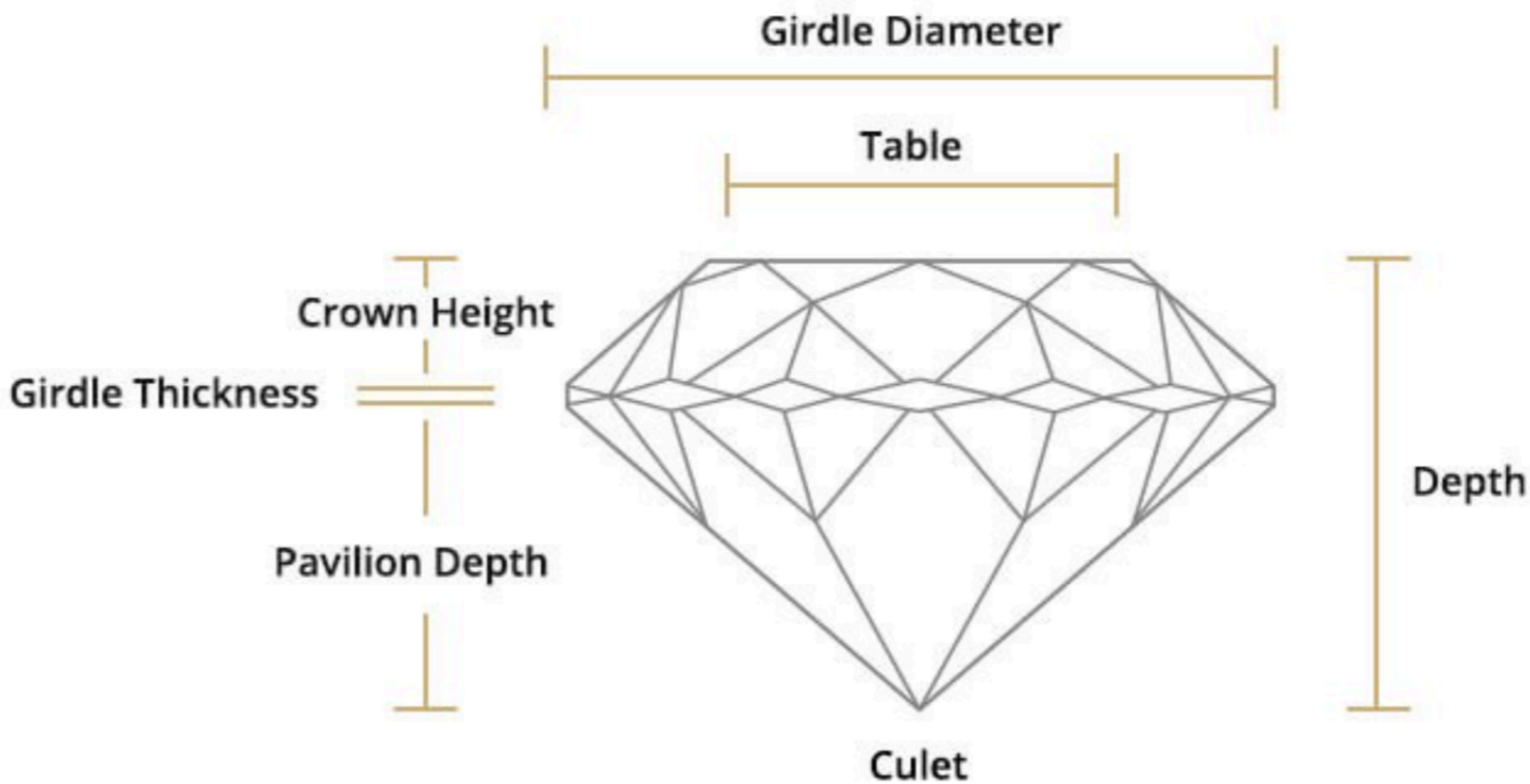
```
scala>  val tvs = new TrainValidationSplit().
             | setEstimator(pipeline).
             | setEvaluator( new RegressionEvaluator().setLabelCol(“priceOutputVar”) ).
             | setEstimatorParamMaps(paramGrid).
             | setTrainRatio(0.75)


scala> val Array(training, test) = data.randomSplit(Array(0.75, 0.25), seed = 12345)

scala> val model = tvs.fit(training)

scala> val holdout = model.transform(test).select("prediction", “priceOutputVar")

scala> import org.apache.spark.mllib.evaluation.RegressionMetrics


scala> val rm = new RegressionMetrics(holdout.rdd.map(x =>(x(0).asInstanceOf[Double], x(1).asInstanceOf[Double]))
scala> println("sqrt(MSE): " + Math.sqrt(rm.meanSquaredError))
sqrt(MSE): 1141.7457210656032

scala> println("R Squared: " + rm.r2)
R Squared: 0.9179908957599221

scala> println("Explained Variance: " + rm.explainedVariance + "\n")
Explained Variance: 1.4616693408985674E7
```

What else?

Many other operations:

- https://spark.apache.org/docs/latest/ml-features.html

Diamonds


Girdle Diameter
Table
Crown Height
Girdle Thickness
Pavilion Depth
Depth
Culet

FETS for Diamonds

- Dataset available here
  - https://www.dropbox.com/s/t6jxtilvn31e2gb/diamonds.csv
- Use spark-shell
- Loading the data:

```
val data = spark.read
  |.format(“csv").option("header", "true").option("inferSchema", true")
  |.load(datapath).withColumn(“priceOutputVar", $"price".cast("double")).cache()
```