CSC 735 – Data Analytics

Chapter 24

Machine Learning Overview 2

Spark’s Advanced Analytics Toolkit

- Sparks includes several core packages and many external packages for performing advanced analytics.
- **MLlib** is primary package, which provides an interface for building machine learning pipelines

What Is MLlib?

- It provides interfaces for
    - gathering and cleaning data
    - feature engineering and feature selection
    - training and tuning large scale supervised and unsupervised machine learning models, and using this models in production

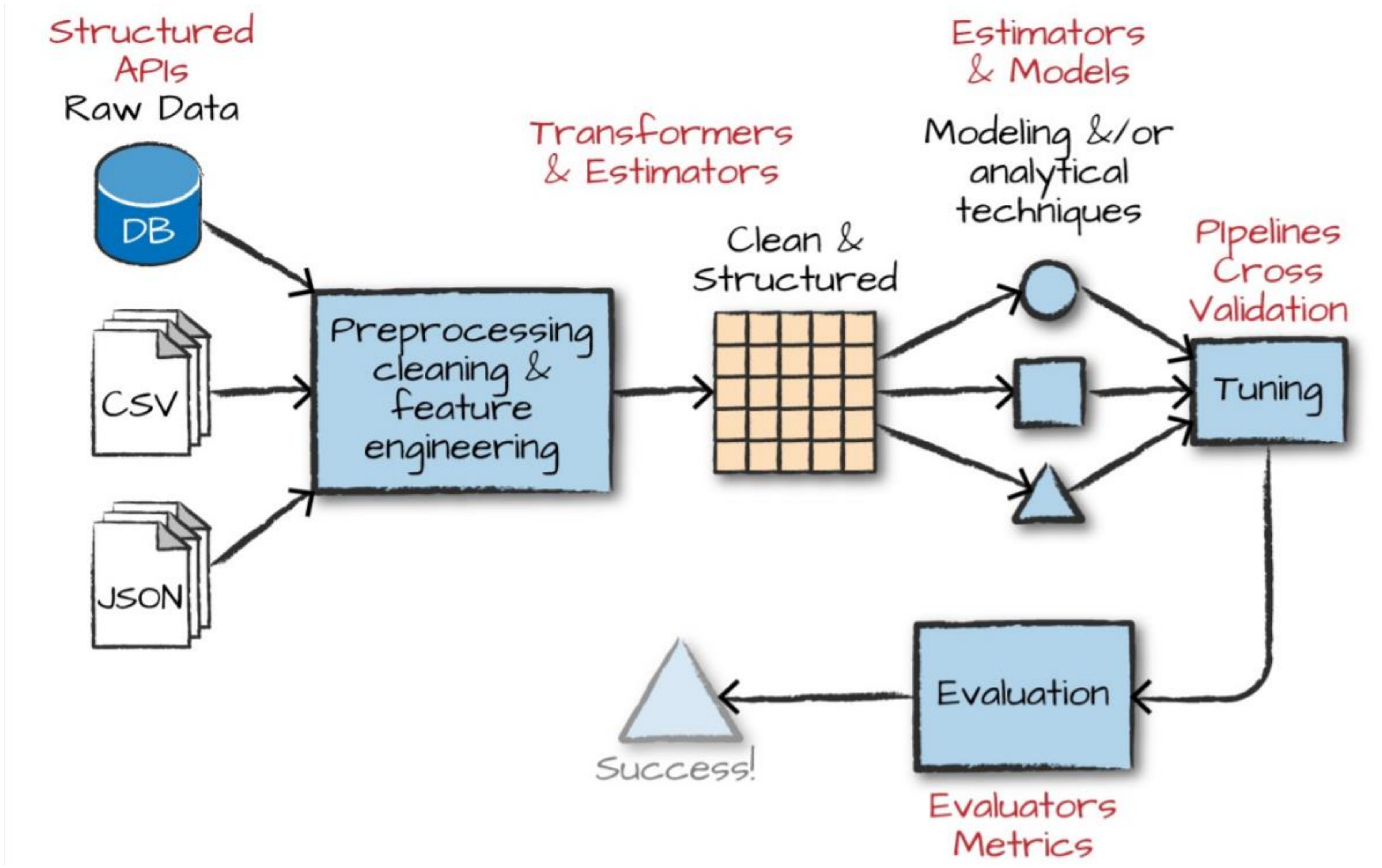
What Is MLlib?

- It provides interfaces for
    - gathering and cleaning data
    - feature engineering and feature selection
    - training and tuning large scale supervised and unsupervised machine learning models, and using this models in production
- MLlib consists of two package that leverage different core data structure
    - org.apache.spark.ml (include an interface for using Data Frame and building ML pipelines)
    - org.apache.spark.mllib (Spark’s low-level RDD APIs)

When and Why Should you use Mllib?

- Limitations of a single machine learning tools
    - Size of data
    - Processing time
- We use Spark
    - Preprocessing and feature generation to reduce the amount of time it might take to produce training and test sets from a large amount of data
    - When input data or model size become to difficult or inconvenient to put on one machine

High-Level MLlib Concept


Structured APIs
Raw Data
DB
CSV
JSON
Preprocessing cleaning & feature engineering
Transformers & Estimators
Clean & Structured
Estimators & Models
Modeling &/or analytical techniques
Pipelines Cross Validation
Tuning
Evaluation
Evaluators Metrics
Success!

High-Level MLlib -Transformers

- Transformers are functions that convert raw data in some way
- Used in processing and feature engineering

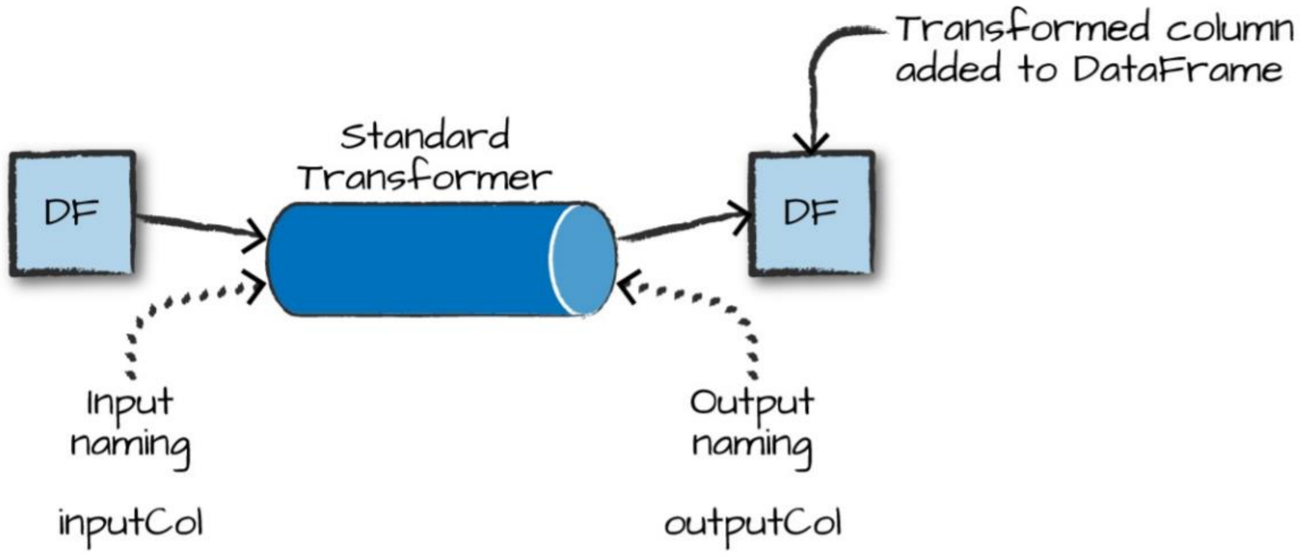


Figure 24-3. A standard transformer

High-Level MLlib - Estimator &Evaluator

- **Estimator**
  - Can be a kind of transformer that is initialized with data
  - Applies the generated function over the data
  - Algorithms that allow user to train a model from data are also referred to as estimators

High-Level MLlib - Estimator &Evaluator

- **Estimator**
  - Can be a kind of transformer that is initialized with data
  - Applies the generated function over the data
  - Algorithms that allow user to train a model from data are also referred to as estimators
- An **evaluator** allows us to see how a given model performs according to criteria
  - For example, a receiver operating characteristic (ROC) curve

Low-level Data Types

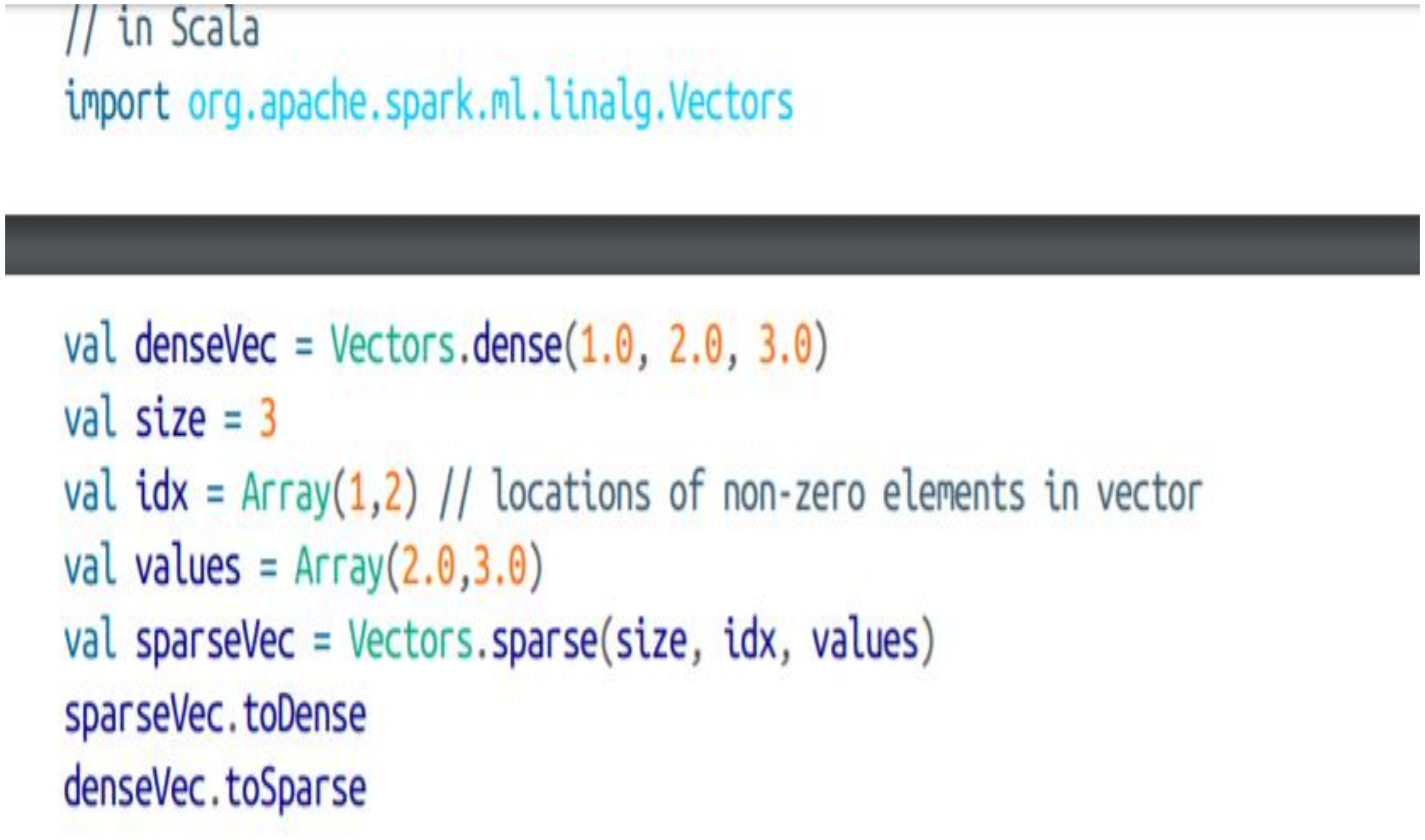
- Feature into machine learning model as a vector consisting **Double**

Low-level Data Types

- Feature into machine learning model as a vector consisting **Double**

- Types of vector
  - Sparse (where most of elements are zero)
  - Dense (where there are many unique values)

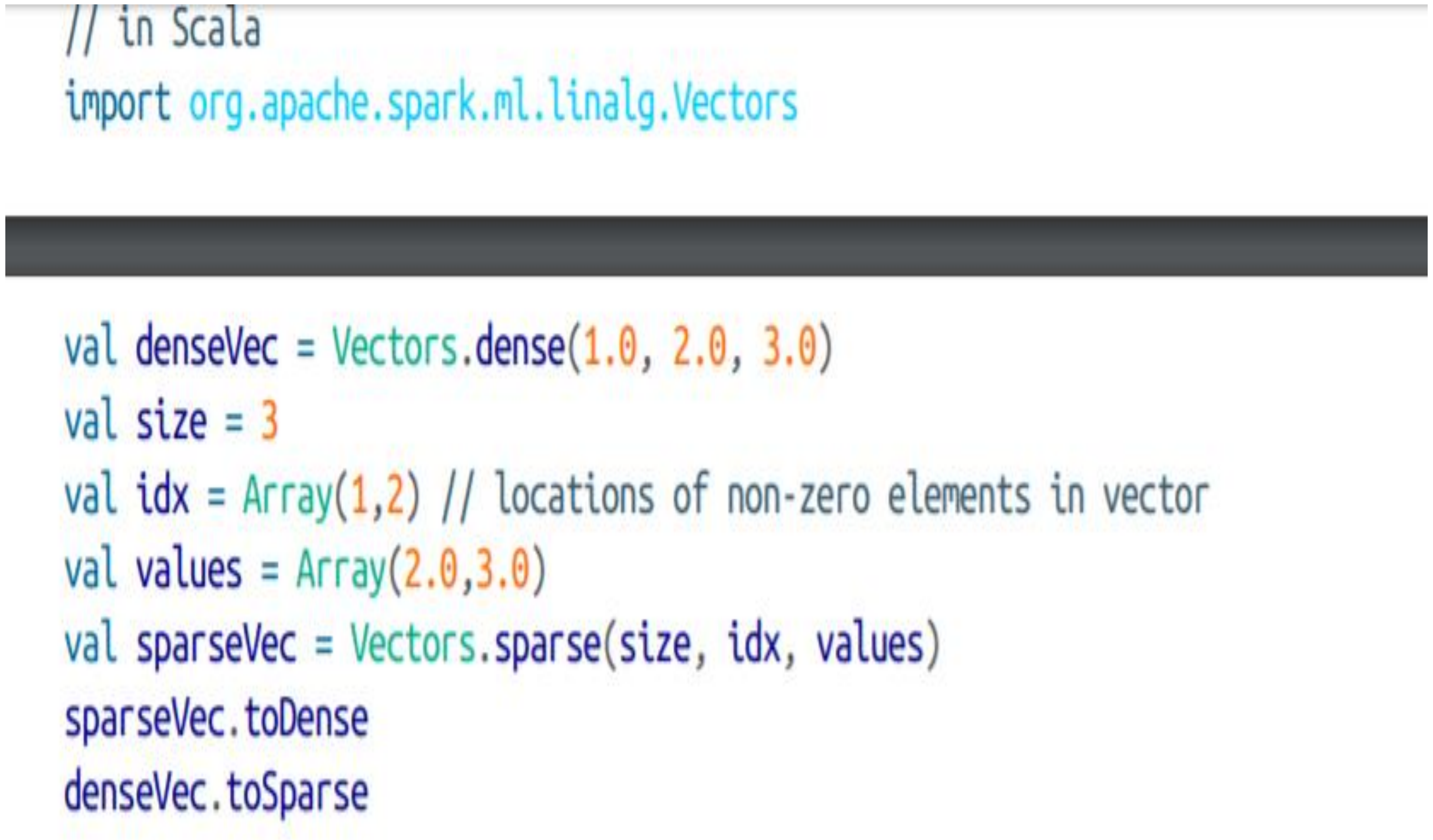
Low-level Data Types



```
// in Scala
import org.apache.spark.ml.linalg.Vectors
```

```
val denseVec = Vectors.dense(1.0, 2.0, 3.0)
val size = 3
val idx = Array(1,2) // locations of non-zero elements in vector
val values = Array(2.0,3.0)
val sparseVec = Vectors.sparse(size, idx, values)
sparseVec.toDense
denseVec.toSparse
```

Low-level Data Types



```
// in Scala
import org.apache.spark.ml.linalg.Vectors
```

```
val denseVec = Vectors.dense(1.0, 2.0, 3.0)
val size = 3
val idx = Array(1,2) // locations of non-zero elements in vector
val values = Array(2.0,3.0)
val sparseVec = Vectors.sparse(size, idx, values)
sparseVec.toDense
denseVec.toSparse
```

WARNING

Confusingly, there are similar datatypes that refer to ones that can be used in DataFrames and others that can only be used in RDDs. The RDD implementations fall under the mllib package while the DataFrame implementations fall under ml.

MLlib in Action

```
// in Scala
var df = spark.read.json("/data/simple-ml")
df.orderBy("value2").show()

# in Python
df = spark.read.json("/data/simple-ml")
df.orderBy("value2").show()
```

Here's a sample of the data:

```
+-----+----+------+------------------+
|color| lab|value1|            value2|
+-----+----+------+------------------+
|green|good|     1|14.386294994851129|
...
|  red| bad|    16|14.386294994851129|
|green|good|    12|14.386294994851129|
+-----+----+------+------------------+
```

Feature Engineering with Transformers

- All inputs to machine leaning algorithms in Spark must consist of type **Double**(for label) and **Vector[Double]**(for features)
- RFormula
  - ❑ ~ Separate target and terms
  - ❑ + Concat terms; "+ 0" means removing the intercept (this means that the y-intercept of the line that we will fit will be 0)
  - ❑ - Remove a term; "- 1" means removing the intercept (this means that the y-intercept of the line that we will fit will be 0—yes, this does the same thing as "+ 0"
  - ❑ : Interaction (**multiplication** for numeric values, or binarized categorical values)
  - ❑ . All columns except the target/dependent variable

```
// in Scala
import org.apache.spark.ml.feature.RFormula
val supervised = new RFormula()
  .setFormula("lab ~ . + color:value1 + color:value2")

# in Python
from pyspark.ml.feature import RFormula
supervised = RFormula(formula="lab ~ . + color:value1 + color:value2")
```

```scala
// in Scala
import org.apache.spark.ml.feature.RFormula
val supervised = new RFormula()
  .setFormula("lab ~ . + color:value1 + color:value2")
```

```python
# in Python
from pyspark.ml.feature import RFormula
supervised = RFormula(formula="lab ~ . + color:value1 + color:value2")
```

```scala
// in Scala
val fittedRF = supervised.fit(df)
val preparedDF = fittedRF.transform(df)
preparedDF.show()
```

```python
# in Python
fittedRF = supervised.fit(df)
preparedDF = fittedRF.transform(df)
preparedDF.show()
```

```
// in Scala
import org.apache.spark.ml.feature.RFormula
val supervised = new RFormula()
  .setFormula("lab ~ . + color:value1 + color:value2")
```

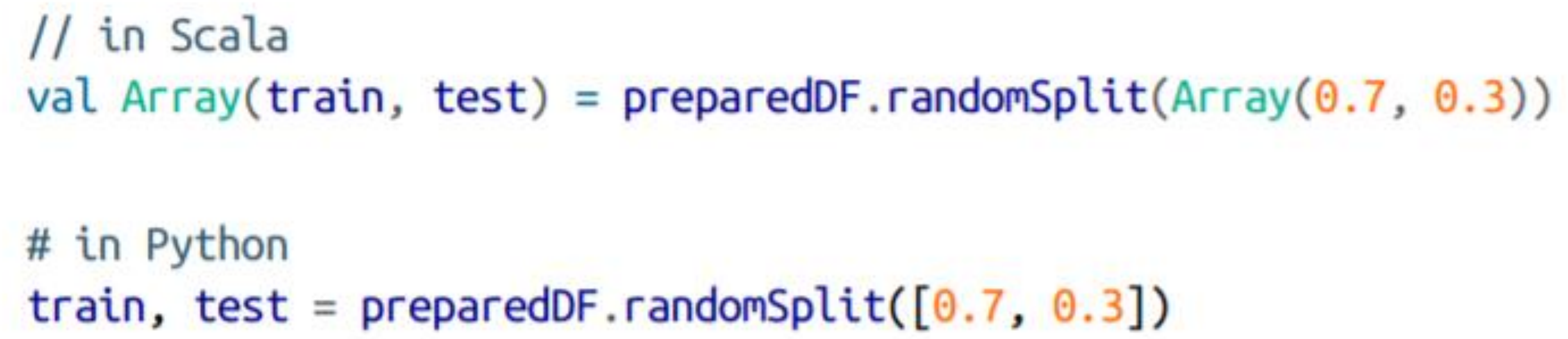
```
# in Python
from pyspark.ml.feature import RFormula
supervised = RFormula(formula="lab ~ . + color:value1 + color:value2")
```

```
// in Scala
val fittedRF = supervised.fit(df)
val preparedDF = fittedRF.transform(df)
preparedDF.show()
```

```
# in Python
fittedRF = supervised.fit(df)
preparedDF = fittedRF.transform(df)
preparedDF.show()
```

```
+-----+----+------+------------------+--------------------+-----+
|color| lab|value1|            value2|            features|label|
+-----+----+------+------------------+--------------------+-----+
|green|good|     1|14.386294994851129|(10,[1,2,3,5,8],[...|  1.0|
...
|  red| bad|     2|14.386294994851129|(10,[0,2,3,4,7],[...|  0.0|
+-----+----+------+------------------+--------------------+-----+
```

Create a Test Set

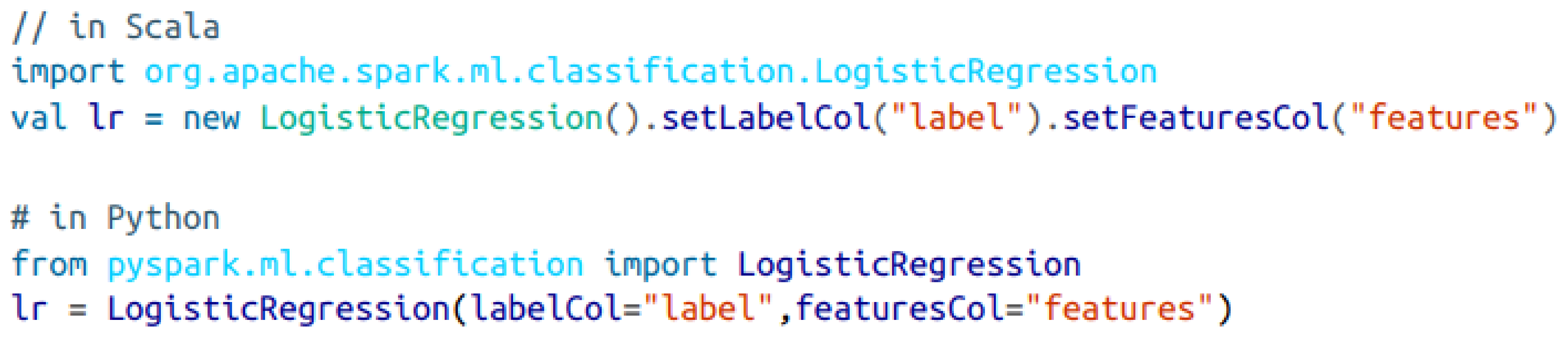


```
// in Scala
val Array(train, test) = preparedDF.randomSplit(Array(0.7, 0.3))

# in Python
train, test = preparedDF.randomSplit([0.7, 0.3])
```

Estimators

We use a classifier algorithm called logistic regression to fit our model.

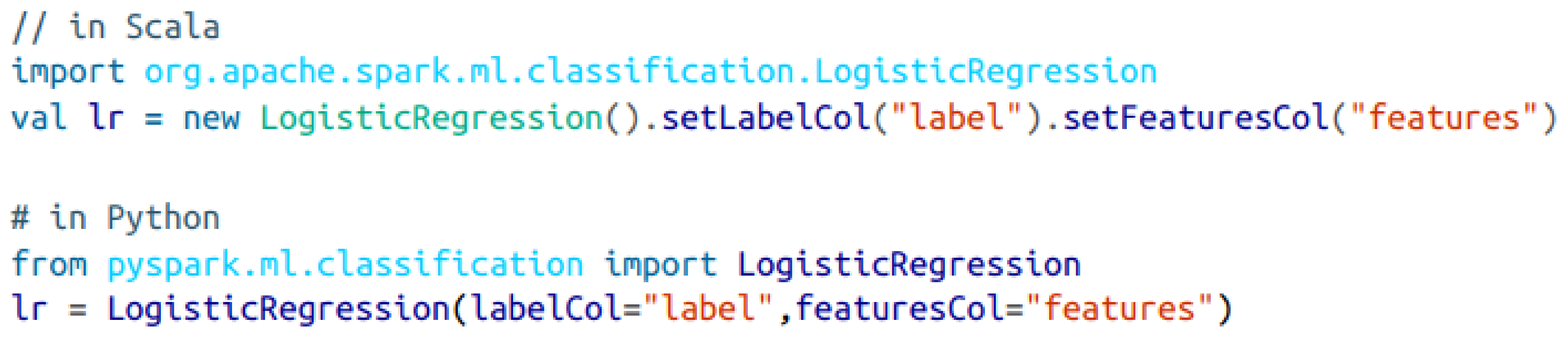


```
// in Scala
import org.apache.spark.ml.classification.LogisticRegression
val lr = new LogisticRegression().setLabelCol("label").setFeaturesCol("features")

# in Python
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(labelCol="label",featuresCol="features")
```

Estimators

We use a classifier algorithm called logistic regression to fit our model.



```
// in Scala
import org.apache.spark.ml.classification.LogisticRegression
val lr = new LogisticRegression().setLabelCol("label").setFeaturesCol("features")

# in Python
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(labelCol="label",featuresCol="features")
```

- Inspecting the parameters is a great way to remind yourself of the options available for every ML algorithm.
  - println(lr.explainParams()) // in Scala
  - print lr.explainParams() # in Python

Estimators

  - val fittedLR = lr.fit(train) // in Scala
  - fittedLR = lr.fit(train) # in Python
- This will launch a Spark job to train the model.

Estimators

  - val fittedLR = lr.fit(train) // in Scala
  - fittedLR = lr.fit(train) # in Python
- This will launch a Spark job to train the model.
- Previously, as in our transformations, they are lazy executions. This is not a lazy execution, and is performed immediately.
- Once this has been completed, you are now ready to start making predictions.
- Predictions are made with the "**transform**" method.

Estimators

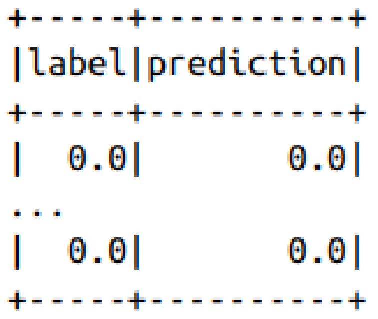
- val fittedLR = lr.fit(train) // in Scala
- fittedLR = lr.fit(train) # in Python

- This will launch a Spark job to train the model.
- Previously, as in our transformations, they are lazy executions. This is not a lazy execution, and is performed immediately.
- Once this has been completed, you are now ready to start making predictions.
- Predictions are made with the “**transform**” method.
  - fittedLR.transform(train).select("label", "prediction").show()

Estimators

- val fittedLR = lr.fit(train) // in Scala
- fittedLR = lr.fit(train) # in Python

- This will launch a Spark job to train the model.
- Previously, as in our transformations, they are lazy executions. This is not a lazy execution, and is performed immediately.
- Once this has been completed, you are now ready to start making predictions.
- Predictions are made with the “**transform**” method.
  - fittedLR.transform(train).select("label", "prediction").show()



```
+-----+----------+
|label|prediction|
+-----+----------+
|  0.0|       0.0|
...
|  0.0|       0.0|
+-----+----------+
```

Estimators cont.

- Now we can manually evaluate our model.
  - Using metrics to check performance:
    - True positives
    - False positives
    - True negatives
    - False negatives

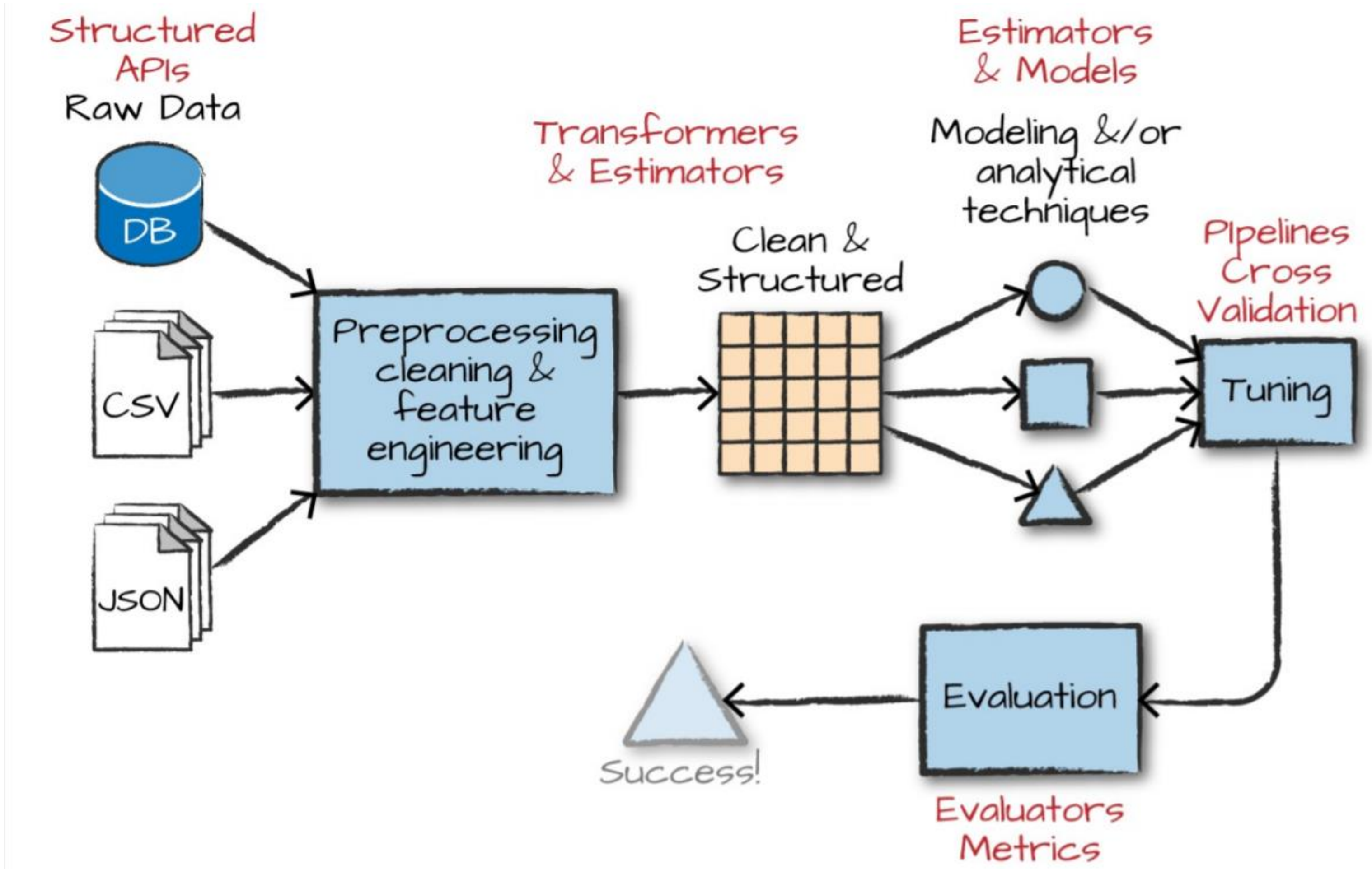
Estimators cont.

- Experiment with other parameters and check to see if our model is better/worse.
- This is helpful, but can be tedious.
- Spark lets you specify your workload as a declarative pipeline of work which includes all your transformations, including tuning hyperparameters.
  - Hyperparameters – Configuration parameters that affect the training process.
    - Ex. Model architecture, Regularization
    - Set prior to starting training
    - Again, println(<model name>.explainParams()) is a good way to see which parameters are available

Pipelining Our Workflow cont.

- With lots of transformations, writing all the steps in code, and keeping track of the different DFs, it can become tedious.
- Spark helps by including a Pipeline concept.
- A pipeline sets up a dataflow for the transformations, which gives you an estimator that is already tuned to your specifications, yielding a ready to use tuned model.

Pipelining Our Workflow


Structured APIs
Raw Data
DB
CSV
JSON
Preprocessing cleaning & feature engineering
Transformers & Estimators
Clean & Structured
Estimators & Models
Modeling &/or analytical techniques
Pipelines Cross Validation
Tuning
Evaluation
Evaluators Metrics
Success!

Pipelining Our Workflow cont.

- To avoid overfitting, create a “holdout” **test** set.
- We will tune hyperparameters based on a validation set later.
- Ex.

```
val Array(train, test) = df.randomSplit(Array(0.7, 0.3))
```

Pipelining Our Workflow cont.

- Once you have a holdout test set, you can now create the base stages in the pipeline.
- A stage represents a transformer or an estimator.
- Ex. RFormula() https://spark.apache.org/docs/2.2.0/ml-features.html#rformula
- This will analyze the data to understand the types of input features, and then transform them to create new features.

Pipelining Our Workflow cont.

- Following the RFormula, we could call the LogisticRegression() algorithm, which we will use to produce a model.

```
val rForm = new RFormula()
val lr = new
LogisticRegression().setLabelCol("label").setFeaturesCol("features")
```

Pipelining Our Workflow cont.

- Instead of manually coding transformations and tuning, we make them stages in the overall pipeline.
- import org.apache.spark.ml.Pipeline
- val stages = Array(rForm, lr)
- val pipeline = new Pipeline().setStages(stages)

Training and Evaluation

- Now we can begin training.
- We don’t want to train just one model, but to train several variations of the model, specifying different combinations of hyperparameters, to test.
- From there, we will select the best model using an Evaluator, testing their predictions.

Training and Evaluation cont.

Ex.

```
import org.apache.spark.ml.tuning.ParamGridBuilder
val params = new ParamGridBuilder()
.addGrid(rForm.formula, Array( "lab ~ . + color:value1",
"lab ~ . + color:value1 + color:value2")) .addGrid(lr.elasticNetParam,
Array(0.0, 0.5, 1.0)) .addGrid(lr.regParam, Array(0.1, 2.0))
.build()
```

Training and Evaluation cont.

Ex.

```
import org.apache.spark.ml.tuning.ParamGridBuilder
val params = new ParamGridBuilder()
.addGrid(rForm.formula, Array( "lab ~ . + color:value1",
"lab ~ . + color:value1 + color:value2")) .addGrid(lr.elasticNetParam,
Array(0.0, 0.5, 1.0)) .addGrid(lr.regParam, Array(0.1, 2.0))
.build()
```

2 different versions of RFormula

- 3 different options for ElasticNet parameter
- 2 different options for regularization parameter

- Thus 12 different versions of logistic regression.

Training and Evaluation cont.

- Evaluation process is next.
- The *Evaluator* allows us to automatically compare multiple models.
- There are different evaluators within Classification, and Regression.
- In this example, we use BinaryClassificationEvaluator and its "areaUnderROC" (Receiver Operating Characteristic).

Training and Evaluation cont.

```
import
org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
val evaluator = new BinaryClassificationEvaluator()
.setMetricName("areaUnderROC")
.setRawPredictionCol("prediction") .setLabelCol("label")
```

Training and Evaluation cont.

- We now have a pipeline specifing how our data should be transformed, performing model selecting (which tries out different hyperparameters in the regression model), and compares their performance using the areaUnderROC metric.
- Remember, it’s best practice to fit hyperparameters on a **validation** split, and not the “holdout” **test** set.

Training and Evaluation cont.

```
import org.apache.spark.ml.tuning.TrainValidationSplit
val tvs = new TrainValidationSplit()
  .setTrainRatio(0.75) // also the default.
  .setEstimatorParamMaps(params).setEstimator(pipeline)
  .setEvaluator(evaluator)
```

Training and Evaluation cont.

- Once you run the entire pipeline, this will test every version of the model against the **validation** set.
- Note: tvsFitted has a return type TrainValidationSplitModel.

```
val tvsFitted = tvs.fit(train)
```

- And now test how it performs on the “holdout” test set.

```
evaluator.evaluate(tvsFitted.transform(test)) // 0.916666
```

Training and Evaluation cont.

Some models let you see a training summary. You must extract it from the pipeline, cast to a proper type, and print the results.

```
import org.apache.spark.ml.PipelineModel
import org.apache.spark.ml.classification.LogisticRegressionModel
val trainedPipeline = tvsFitted.bestModel.asInstanceOf[PipelineModel]
val TrainedLR = trainedPipeline.stages(1).asInstanceOf[LogisticRegressionModel]
val summaryLR = TrainedLR.summary
summaryLR.objectiveHistory // 0.6751425885789243, 0.5543659647777687,
0.473776...
```

Training and Evaluation cont.

- Objective history provides details related to how the algorithm performed over each training iteration.
- It could have large jumps in the beginning, but should become smaller with small variations, over time.

Persisting and Applying Models

- Once you have a trained model, we can persist it to **disk**, for later use and prediction.

  tvsFitted.write.overwrite().save(“<model location>”)

Persisting and Applying Models

- Once you have a trained model, we can persist it to **disk**, for later use and prediction.

  tvsFitted.write.overwrite().save(“<model location>”)

- To load the model into another Spark program, we need to use a “model” version of the particular algorithm to load our persisted model from disk.
- Note: Since we used a tvsFitted, with a return type TrainValidationSplitModel, we need to use that type to load the model.

```
import org.apache.spark.ml.tuning.TrainValidationSplitModel
val model = TrainValidationSplitModel.load("/tmp/modelLocation")
model.transform(test)
```

Deployment Patterns

- Spark has several different deployment patters for putting machine learning models into production.

Deployment Patterns

- Spark has several different deployment patters for putting machine learning models into production.

1. Train your ML model **offline** and then supply it with **offline** data.
   - Spark is well suited for this deployment
2. Train your model **offline** and then put the results into a database.
   - Good for recommendation.
   - Bad for classification or regression.
3. Train your ML algorithm **offline**, persist the model to disk, and then use it for serving.
   - Not a low-latency solution if using Spark for the serving part.
   - Does not parallelize well.
4. Train your ML algorithm **online** and use it **online**.
   - Uses Structured **Streaming**, which can be complex for some models.

Conclusion

- While not every workflow or pipelining process has been covered, the big picture has been explained.