INTRODUCTION TO APACHE SPARK

4. SPARK MLLIB APIS

REMINDER: HOMEWORKS

- TP3 expected before 12/03
- TP4 expected before 19/03
- Submit the exported notebook (.ipynb extension)
- File name should include both student name and no space (FirstName1LastName1_FirstName2LastName2.py) for instance

OVERVIEW

- Spark ML is the name given to Spark's Machine Learning library
- Since Spark 1.6, there are two APIs for Spark ML:
  - RDD API: The older, original one (MLlib)
  - DataFrame API: The new, stable one (Spark ML)
- Since Spark 2.0, it's recommended to use only the **DataFrame API**
- A quick overview of the RDD API is still useful to understand legacy code

RDD API - INTRO

- As the name hints, it uses RDDs
- All functionality is in the package `pyspark.mllib`

RDD API - INTRO

- Most models follow basically the same pattern:
  - `import ModelName`
  - `trained_model = ModelName.train(training_data, params)`
  - `model.predict(some_features)`

RDD API - FUNCTIONALITIES

- General Purpose
  - summary, correlation, random data generation
- Dimensionality Reduction
- Feature extraction and transformation
- Evaluation Methods

RDD API - MODEL CATEGORIES

- Classification and Regression
  - linear models, Bayes, decision trees, random forests, etc.
- Clustering
  - k-means, gaussian mixture, LDA, etc.
- Frequent Pattern Mining
  - fp-growth, association rule mining, etc.

RDD API - DATA TYPES

- To apply Machine Learning algorithims, MLlib comes with some new data types:
  - Local vectors (dense and sparse)
  - Labeled points
  - Local matrix
  - Distributed matrix
- They can be imported from the package `pyspark.mllib.linalg`

LOCAL VECTORS

- Local because they are not distributed
- For dense vectors, Spark accepts python lists or numpy arrays
- For sparse vecctors, Spark accepts single-column csc_matrices (from SciPy) -**But** it's recomended to use `Vector.sparse`

```
import numpy as np
import scipy.sparse as sps
from pyspark.mllib.linalg import Vectors

# Use a NumPy array as a dense vector.
dv1 = np.array([1.0, 0.0, 3.0])

# Use a Python list as a dense vector.
dv2 = [1.0, 0.0, 3.0]

# Create a SparseVector.
sv1 = Vectors.sparse(3, [0, 2], [1.0, 3.0])
```

LABELED POINT

- Used in supervised algorithms
- For binary classification, the label is either 0 (negative) or 1 (positive)
- For multi-class classification, the label represents a class index, starting from 0

```
from pyspark.mllib.linalg import SparseVector
from pyspark.mllib.regression import LabeledPoint

# Create a labeled point with a positive label and a dense fea
pos = LabeledPoint(1.0, [1.0, 0.0, 3.0])

# Create a labeled point with a negative label and a sparse fe
neg = LabeledPoint(0.0, SparseVector(3, [0, 2], [1.0, 3.0]))
```

LOCAL MATRIX

- Local because it's not distributed
- Can be either dense or sparse
- Should be created with Matrix.dense or or Matrix.sparse

```
from pyspark.mllib.linalg import Matrix, Matrices

# Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
dm2 = Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])

# Create a sparse matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))
sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 2, 1], [9, 6, 8])
```

DISTRIBUTED MATRIX

- For "big-data" matrices
- There are four types:
  - RowMatrix: RDD of rows (each row is a Vector)
  - IndexedRowMatrix: Similar to RowMatrix, but each row has an index
  - CoordinateMatrix: RDD of (i, j, value); Should be used only when both dimensions are "big"
  - BlockMatrix: RDD of MatrixBlocks (sub-matrices of the type (i, j, Matrix))

EXAMPLES

```
from pyspark.mllib.linalg.distributed import *

# RowMatrix
rows = sc.parallelize([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 1
r_mat = RowMatrix(rows)

# IndexedRowMatrix
indexedRows = sc.parallelize([
    (0, [1, 2, 3]),
    (1, [4, 5, 6]),
    (2, [7, 8, 9]),
    (3, [10, 11, 12])
])
ir_mat = IndexedRowMatrix(indexedRows)
```

EXAMPLES (CONT.)

```
from pyspark.mllib.linalg.distributed import *

# CoordinateMatrix
entries = sc.parallelize([
    (0, 0, 1.2), (1, 0, 2.1), (2, 1, 3.7)
])
coo_mat = CoordinateMatrix(entries)

# BlockMatrix
blocks = sc.parallelize([
    ((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
    ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))
])
mat = BlockMatrix(blocks, 3, 2)
```

RDD API - GENERAL PURPOSE

- MLlib contains some general-purpose functionality
- Some examples:
  - Column-wise summary
  - Correlations
  - Full list can be found here
  - And the full docs, here

COLUMN-WISE SUMMARY

- The function colStats() can be applied to a matrix to find some stats for each column:
  - total count
  - min, max, mean and variance
  - number of non-zeroes

```
import numpy as np
from pyspark.mllib.stat import Statistics

mat = sc.parallelize([
  np.array([1.0, 10.0, 100.0]),
  np.array([2.0, 20.0, 200.0]),
  np.array([3.0, 30.0, 300.0])
])

summary = Statistics.colStats(mat)

mean = summary.mean()
variance = summary.variance()
# etc.
```

CORRELATIONS

- Can be applied to multiple series (RDD of float) in order to find correlation between them
- Can use either "spearman" or "pearson" (default) methods

```
from pyspark.mllib.stat import Statistics

seriesX = sc.parallelize([1.0, 2.0, 3.0, 3.0, 5.0])
seriesY = sc.parallelize([11.0, 22.0, 33.0, 33.0, 555.0])

corr = Statistics.corr(seriesX, seriesY, method="pearson")
```

RDD API: CLASSIFICATION AND REGRESSION

- Most common ML algorithms
- Docs:
  - Guide
  - Classification package
  - Regression package

AVAILABLE METHODS

Type	Available methods
Regression	linear least squares, Lasso, ridge regression, decision trees, random forests, gradient-boosted trees, isotonic regression
Binary Classification	linear SVMs, logistic regression, decision trees, random forests, gradient-boosted trees, naive Bayes
Multiclass Classification	logistic regression, decision trees, random forests, naive Bayes

EXAMPLE: DECISION TREES

- Can be used for both classification and regression
- In spark, 3 strategies for impurity are accepted: "gini", "entropy" and "variance"
- Details of the algorithm can be found here

EXAMPLE

```
# split data
(trainingData, testData) = data.randomSplit([0.7, 0.3])
# train model
model = DecisionTree.trainClassifier(trainingData,
  numClasses=2, categoricalFeaturesInfo={}, impurity='gini',
  maxDepth=5, maxBins=32)
# predict
predictions = model.predict(testData.map(lambda x: x.features))
# compute error
labelsAndPreds = testData.map(lambda lp: lp.label)\
                         .zip(predictions)
testErr = labelsAndPreds.filter(lambda lp: lp[0] != lp[1])\
                        .count() / float(testData.count())
```

RDD API

- For more details:
  - Guide
  - Full method documentation

DATAFRAME API - INTRO

- Instead of RDDs of vectors or matrices, it uses DataFrames
- DataFrames are much easier to manipulate, so this API is becoming the main one
- Complete parity between both APIs since Spark 2.3

DATAFRAME API - COMPONENTS

- The DataFrame API tries to simplify things by defining some few global components:
  - Transformer: transforms a DataFrame into another one.
  - Estimator: transforms a DataFrame into a Transformer
  - Pipeline: chains multiple Transformers and Estimators

DF API: TRANSFORMERS

- DataFrame -> DataFrame
- Example 1: Tokenizer (e.g. break text into words)

```
from pyspark.ml.feature import Tokenizer, RegexTokenizer

sentenceDataFrame = spark.createDataFrame([
    (0, "This is a sentence"),
    (1, "This is another one"),
], ["id", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
tokenized = tokenizer.transform(sentenceDataFrame)

tokenized.select("sentence", "words").show(5, False)
```

```
+------------------+-----------------------+
|sentence          |words                  |
+------------------+-----------------------+
|This is a sentence |[this, is, a, sentence] |
|This is another one|[this, is, another, one]|
+------------------+-----------------------+
```

DF API: TRANSFORMERS

- DataFrame -> DataFrame
- Example 2: Applying a Model (transform in this case means predict)

```
training_data = ...
test_data = ...

dt = DecisionTreeClassifier(labelCol="indexedLabel",
        featuresCol="indexedFeatures")
model = dt.fit(training_data)

# 'model' is a Transformer, so 'predictions'
# will be a DataFrame
predictions = model.transform(test_data)
```

DF API: ESTIMATORS

- DataFrame -> Transformer
- Example: Learning a Model

```
training_data = ...
test_data = ...

dt = DecisionTreeClassifier(labelCol="indexedLabel",
        featuresCol="indexedFeatures")

# 'dt' is an Estimator, so the returned value
# will be a Transformer
model = dt.fit(training_data)

predictions = model.transform(test_data)
```

DF API: PARAMETERS

- Common interface for configuring Transformers and Estimators
- Three main approaches:
  - Passing them directly in the call
  - Using setters
  - Using dictionaries
- You can mix the approaches!

DF API: PARAMETERS

- Passing in the call

```
training_data = ...
test_data = ...

dt = DecisionTreeClassifier(labelCol="indexedLabel",
        featuresCol="indexedFeatures", impurity="gini")

model = dt.fit(training_data)

predictions = model.transform(test_data)
```

DF API: PARAMETERS

- Using setters

```
training_data = ...
test_data = ...

dt = DecisionTreeClassifier(labelCol="indexedLabel",
        featuresCol="indexedFeatures")

# Using a setter:
dt.setImpurity('gini')

model = dt.fit(training_data)

predictions = model.transform(test_data)
```

DF API: PARAMETERS

- Using dict (Note this overrides previous parameters)

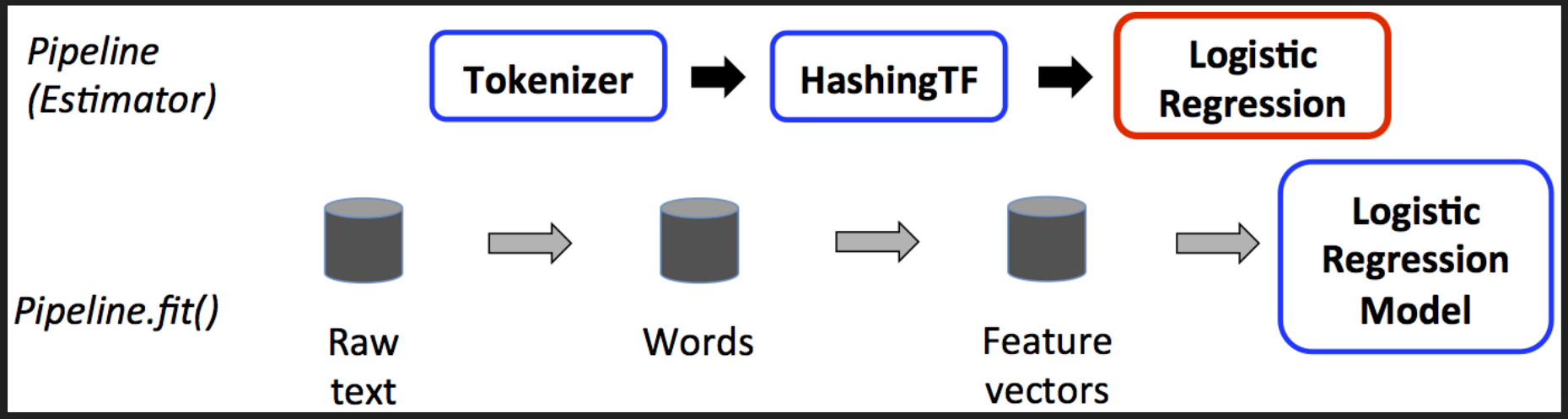
```
training_data = ...
test_data = ...
dt = DecisionTreeClassifier()

# Using a dictionary:
params = dict {
  dt.labelCol: 'indexedLabel',
  dt.featuresCol: 'indexedFeatures'
  dt.impurity: 'gini'
}

model = dt.fit(training_data, params)

predictions = model.transform(test_data)
```

DF API: PIPELINES

- Allows chaining multiple transformers and estimators
- Very useful when we need to execute multiple data transformations before fitting a model

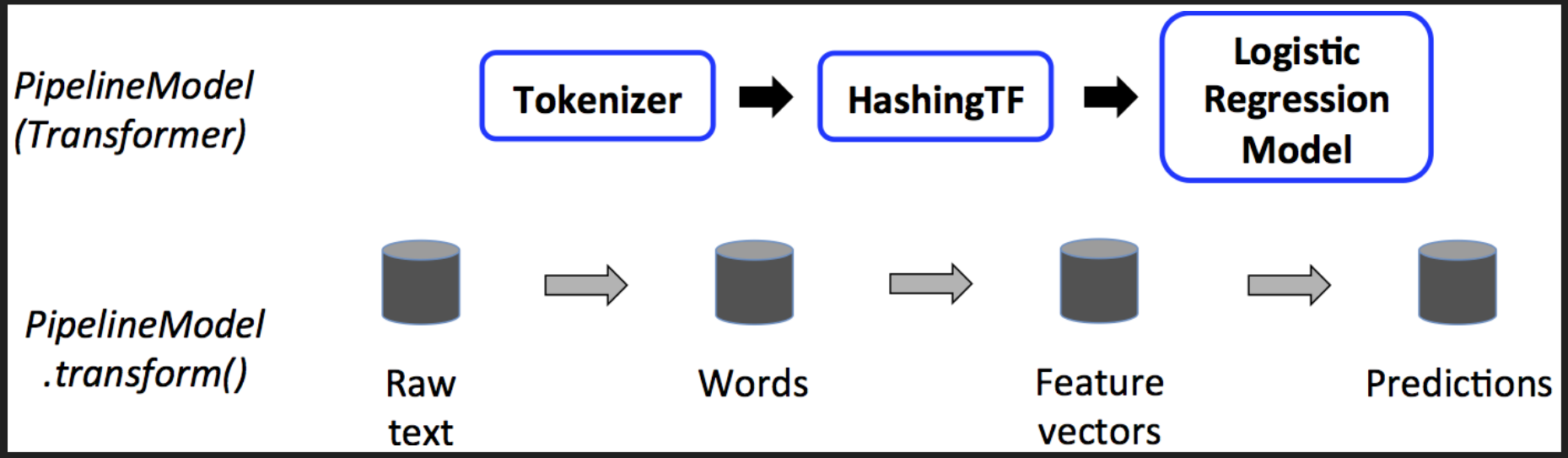
BASIC EXAMPLE:

- Suppose we have the three following tasks:
  1. Split a document's text into words;
  2. Convert each document's words into a numerical feature vector;
  3. Learn a Logistic Regression model.
- With the DF API we can make a Pipeline that looks like the following:



BASIC EXAMPLE:

- When we create a Pipeline, it's an Estimator, which means it returns a Transformer
- Then, the returned Transformer can be used to predict:



BASIC EXAMPLE:

- Code: Preparation

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer

training_data = spark.createDataFrame([
  (0,    "a b c d e hello",  1.0),
  (1,    "b d",              0.0),
  (2,    "hello f g h",      1.0),
  (3,    "oh god",           0.0)
],["id", "text",            "label"])
```

BASIC EXAMPLE:

- Code: building the Pipeline

```
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(),
                      outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)

pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
model = pipeline.fit(training_data)
```

BASIC EXAMPLE:

- Code: predicting test_data

```
test_data = spark.createDataFrame([
    (4,    "hello i j k"),
    (5,    "l m n"),
    (6,    "hello god hello"),
    (7,    "apache god")
], ["id", "text"])

prediction = model.transform(test_data)
prediction.select("id", "text", "probability", "prediction").s
```

```
+---+---------------+----------------+----------+
|id |text           |probability     |prediction|
+---+---------------+----------------+----------+
|4  |hello i j k    |[0.1596,0.8403] |1.0       |
|5  |l m n          |[0.8378,0.1621] |0.0       |
|6  |hello god hello|[0.0692,0.9307] |1.0       |
|7  |apache god     |[0.9821,0.0178] |0.0       |
+---+---------------+----------------+----------+
```

FEATURE EXTRACTION AND MANIPULATION

- There are many transformers and estimators for extracting and manipulating features
- Some useful examples include:
  - Tokenizer (as we have already seen)
  - StopWordsRemover
  - Binarizer
  - Bucketizer, etc.
- The full list with explanation and examples can be found here

STOP WORDS REMOVER

- Stop words are words that should not be considered by our models because they are common or irrelevant
- Examples: "a", "an", "all", "to", "from", etc.
- StopWordsRemover can be used to remove "stop words" from a column of tokens (words)
- The stop words list can be overridden if passed as parameter

STOP WORDS REMOVER

```
from pyspark.ml.feature import StopWordsRemover

sentence_data = spark.createDataFrame([
  (0, ["I", "saw", "the", "red", "balloon"]),
  (1, ["Mary", "had", "a", "little", "lamb"])
], ["id", "raw"])

remover = StopWordsRemover(inputCol="raw",
             outputCol="filtered")
remover.transform(sentence_data).show()
```

```
+---+--------------------+--------------------+
| id|                 raw|            filtered|
+---+--------------------+--------------------+
|  0|[I, saw, the, red...| [saw, red, balloon]|
|  1|[Mary, had, a, li...|[Mary, little, lamb]|
+---+--------------------+--------------------+
```

BINARIZER

- Can be used to convert numeric features into a binary value (0 or 1), based on a threshold

BINARIZER

```
from pyspark.ml.feature import Binarizer

continuous_df = spark.createDataFrame([
  (0, 0.1),
  (1, 0.8),
  (2, 0.2)
], ["id", "feature"])

binarizer = Binarizer(threshold=0.5, inputCol="feature",
    outputCol="binarized_feature")

binarized = binarizer.transform(continuous_df)
binarized.show()
```

```
+---+-------+-----------------+
| id|feature|binarized_feature|
+---+-------+-----------------+
|  0|    0.1|              0.0|
|  1|    0.8|              1.0|
|  2|    0.2|              0.0|
+---+-------+-----------------+
```

STRING INDEXER

- Can be used to find numeric indexes for strings

STRING INDEXER

```
from pyspark.ml.feature import StringIndexer

df = spark.createDataFrame(
[(0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c")],
["id", "category"])

indexer = StringIndexer(inputCol="category",
              outputCol="categoryIndex")
indexed = indexer.fit(df).transform(df)
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|
+---+--------+-------------+
```

BUCKETIZER

- Can be used to transform a continuous column into bucket indexes
- A vector of bucket splits must be defined and passed as parameter

BUCKETIZER

```
from pyspark.ml.feature import Bucketizer

splits = [-float("inf"), -0.5, 0.0, 0.5, float("inf")]

data = [(-999.9,), (-0.5,), (-0.3,), (0.0,), (0.2,), (999.9,)]
df = spark.createDataFrame(data, ["features"])

bucketizer = Bucketizer(splits=splits, inputCol="features",
  outputCol="bucketedFeatures")

bucketed_data = bucketizer.transform(df)
bucketed_data.show()
```

```
+--------+----------------+
|features|bucketedFeatures|
+--------+----------------+
|  -999.9|             0.0|
|    -0.5|             1.0|
|    -0.3|             1.0|
|     0.0|             2.0|
|     0.2|             2.0|
|   999.9|             3.0|
+--------+----------------+
```

WRAP-UP

- There are two APIs for Machine Learning in Spark: RDD and DataFrame
- It's important to know an overview of both
- But the DataFrame API is the main one and should be chosen

WRAP-UP

- It's not necessary to memorize all existing Transformers, Estimators, Params, etc
- But it's very important to know where to look and how to find
- Two main resources:
  - Guide with explanations and examples
  - pyspark docs for DataFrame API
  - pyspark docs for RDD API

(: !DNE EHT

ANY QUESTIONS?