Session: Machine Learning in Spark

ML Overview

Utilities

Transformers

Pipelines

Lesson Objectives

◆ Understand some of the basics of Machine Learning (ML)

◆ Learn how Spark MLlib supports ML
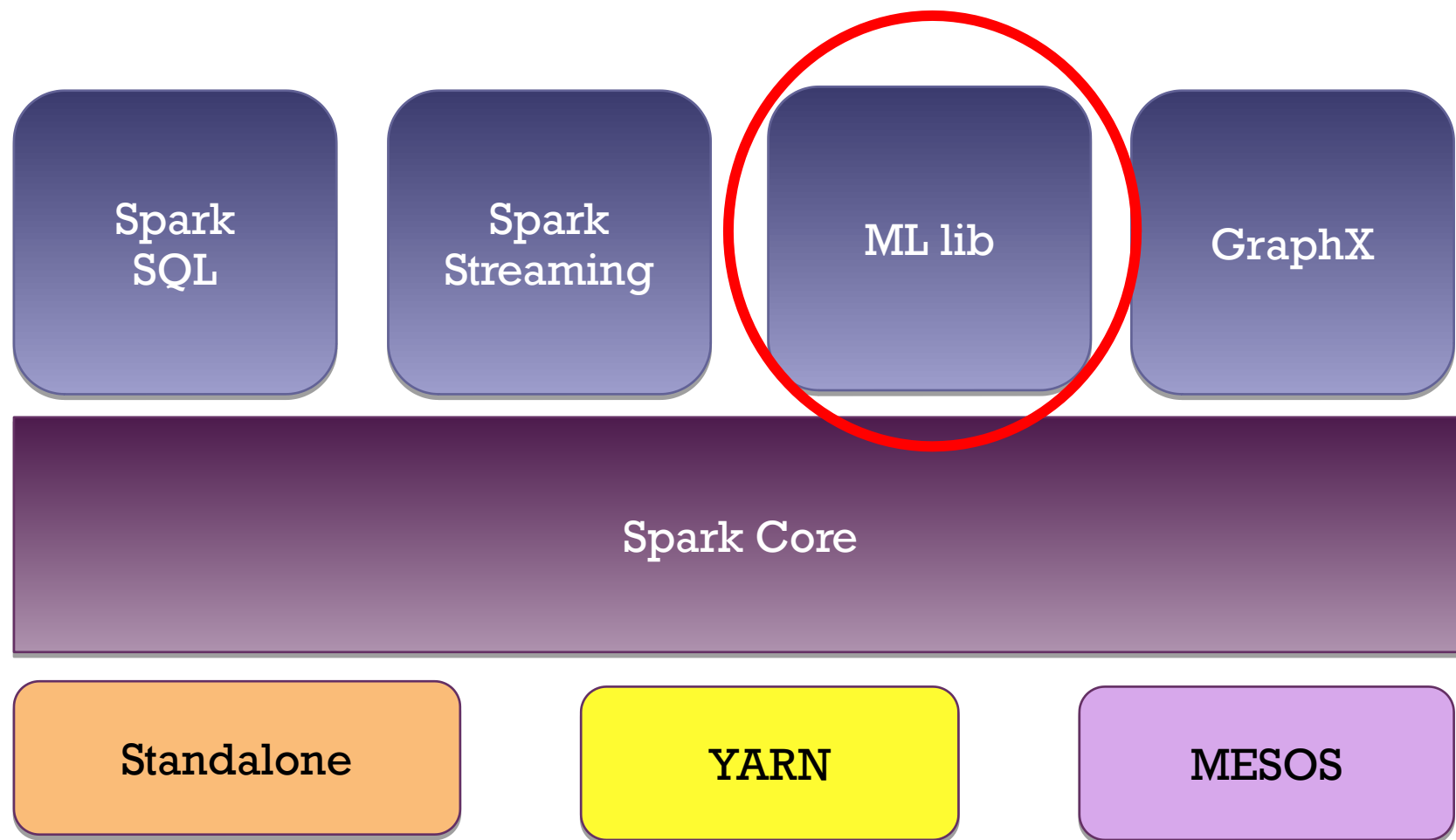
◆ Become familiar with some of the algorithms in MLlib

Introduction

➔ **ML Overview**

Utilities

Transformers

Pipelines

Spark Illustrated

| Spark SQL | Spark Streaming | ML lib | GraphX |

Spark Core

| Standalone | YARN | MESOS |

History of Machine Learning @ Scale

◆ Hadoop
- − Hadoop is the first popular distributed platform
- − MapReduce is the execution engine
- − Did great at batch computes
- − 'Mahout' is a machine learning library built on top of Hadoop's MapReduce
- − Not so great for iterative algorithms (machine learning)
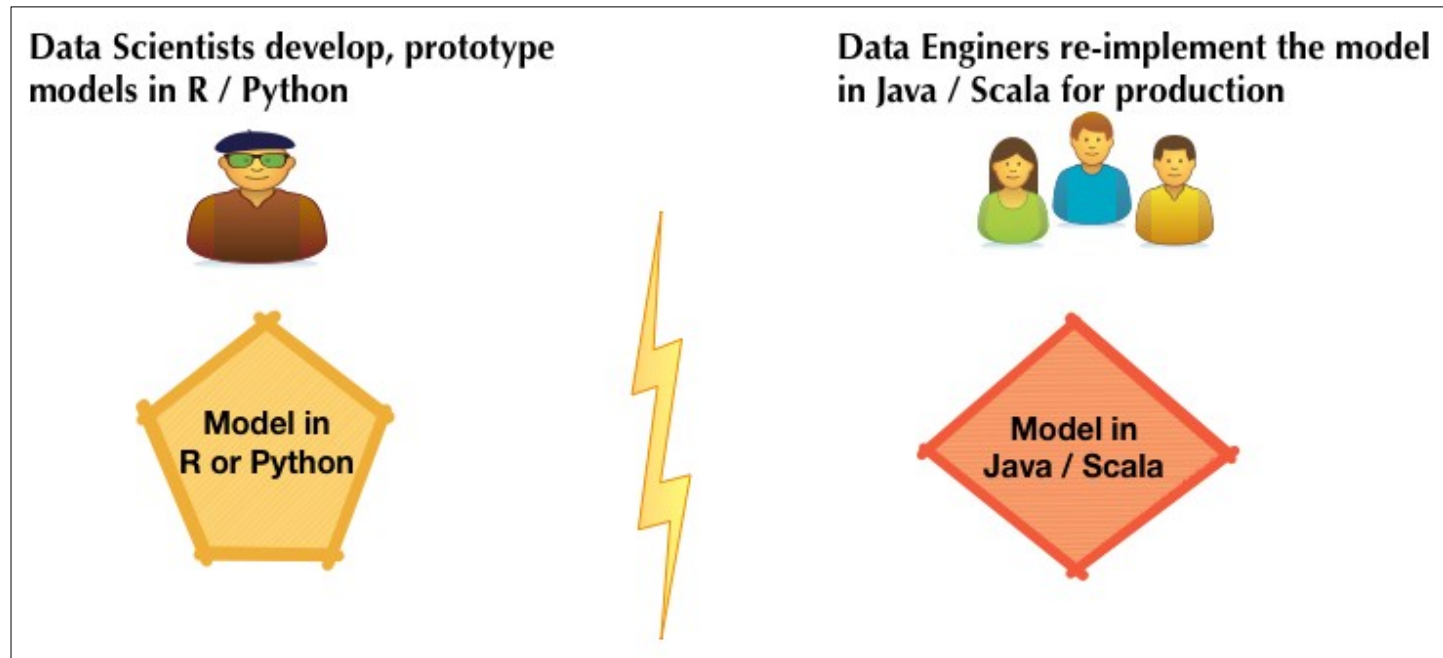
◆ Spark
- − Native Machine Learning component
- − Execution engine is faster than MapReduce (less overhead)
- − Iterative algorithms work well
- − Support in-memory computations (very fast and great for iterative computes)

Spark ML Features

- ◆ Wide variety of algorithms (classifications, clustering, regressions, collaborative filtering)
  - − All parallelized out of the box!

- ◆ Featurization: feature extraction / transformation / dimensionality reduction / selection

- ◆ Pipelines: create, evaluate and tune Pipeline

- ◆ Persistence: saving and loading of algorithms/models/pipelines

- ◆ Utilities: Linear algebra, statistics, data handling

- ◆ Multi Language support: Python / Java / Scala / R
  - − Equal coverage!

◆ Spark.mllib contains the original API built on top of RDDs

◆ MLLib has been migrating to a new API called Spark.ML

◆ **Spark.ml** provides higher-level API built on top of Dataframes for constructing ML pipelines
  − Dataframes provide faster data access, better caching ..etc

◆ As of Spark v2.0, RDD based spark.mllib package is in maintenance mode (no new features, bug fixes only)
  − Removed in Spark v3.0

- ◆ Extra work
- ◆ Different code path
  - − Possible bugs!
- ◆ Updating models is slow!

Streamlining Prototyping → deploy
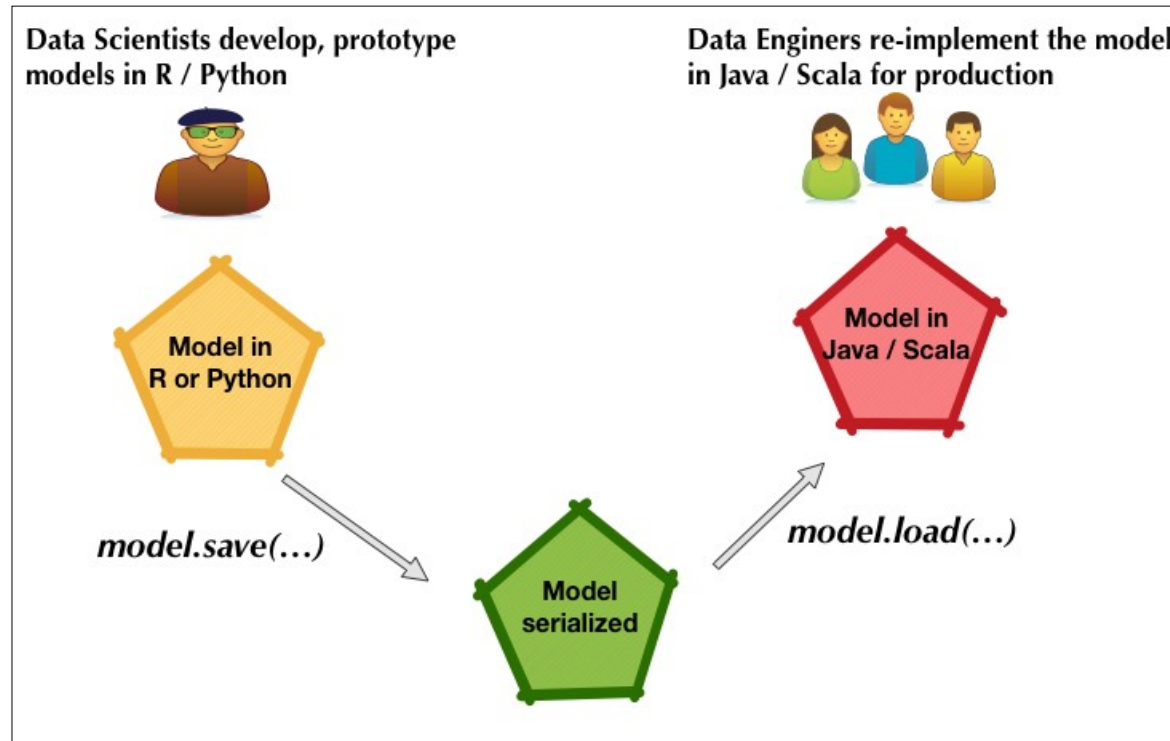
- ◆ Language neutral
- ◆ Same model – no need to re-implement
- ◆ Fast deploy!
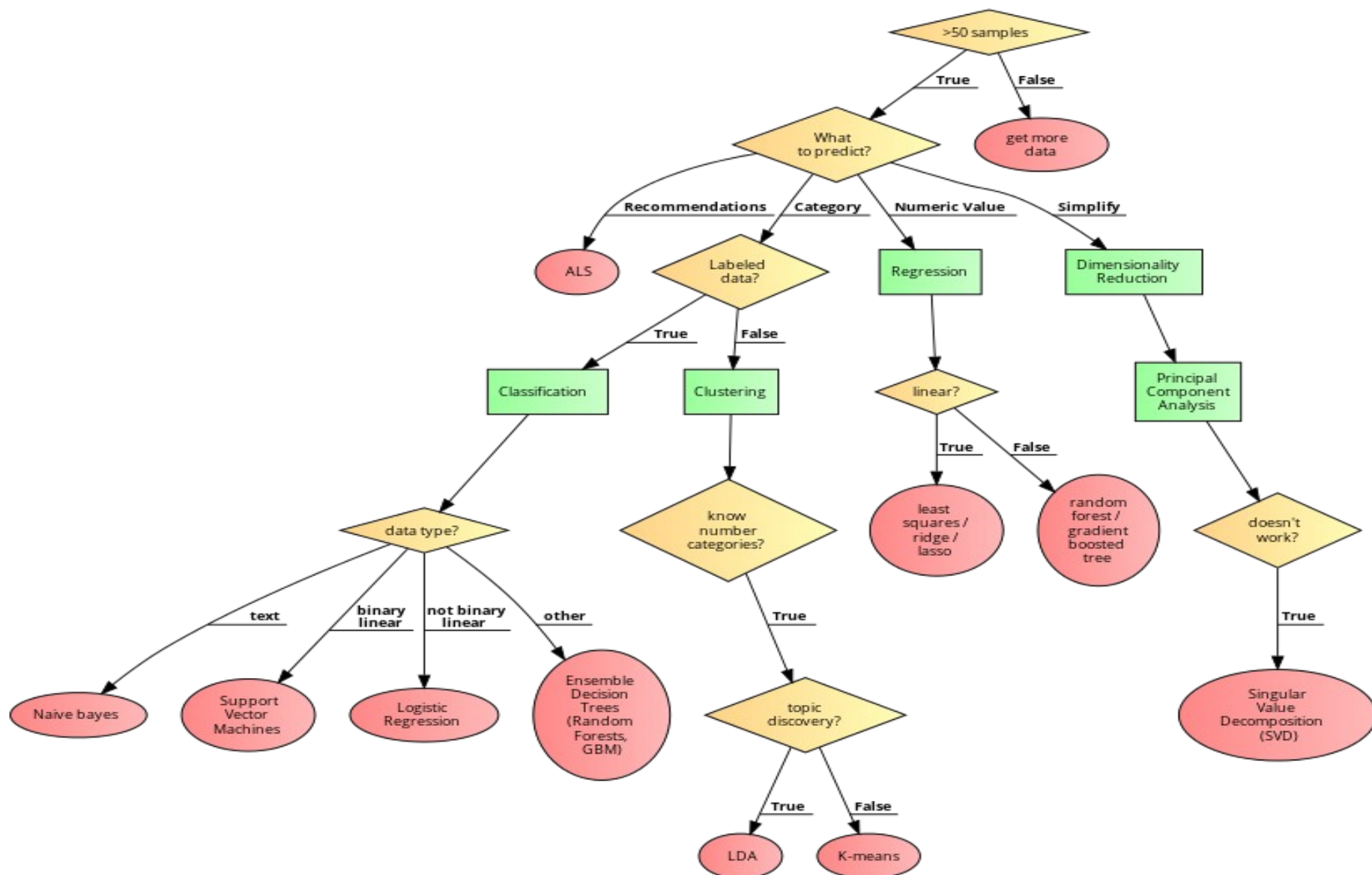
ML Data Types And Utilities

ML Overview

➔ **Utilities**

Transformers

Pipelines

ML Vectors

◆ One dimensional array of Numerics / Doubles

◆ **DenseVector**
When most positions in the vector have value

◆ **SparseVector**
When most elements have no value

org.apache.spark.ml.linalg.Vector

Vector

DenseVector   SparseVector
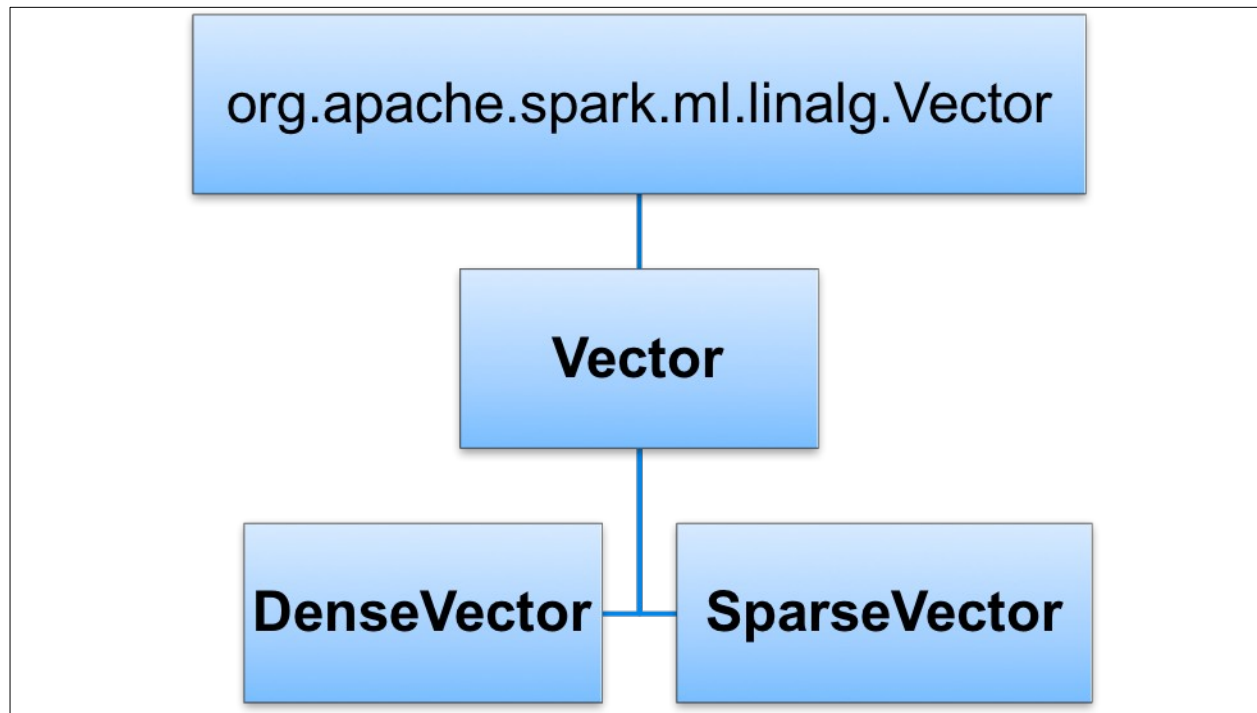
DenseVector vs. Spark Vector

◆ DenseVector = simply an array
   [1, 2, 3, 4, 5]

◆ SparseVector
   − We specify size
   − Index array
   − and value array

◆ Vectors.sparse (length,   index array,   value array)
   Vectors.sparse(10, (0,9),   (100,200))
   − Size is 10
   − 0th (first) element = 100
   − 9th (last) element = 200
   − [100. 0. 0. 0. 0. 0. 0. 0. 0. 200.]

13

Creating Vectors (Scala)

◆ We use **Vectors** class to create dense or sparse vectors

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

// Create a dense vector (1.0, 0.0, 3.0).
val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)

// Create a sparse vector (1.0, 0.0, 3.0) by specifying its
indices and values corresponding to nonzero entries.
val sv1: Vector = Vectors.sparse(3, Array(0, 2), Array(1.0,
3.0))

// Create a sparse vector (1.0, 0.0, 3.0) by specifying its
nonzero entries.
val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))
```

Creating Vectors (Python)

◆ We use **Vectors** class to create dense or sparse vectors

```python
from pyspark.ml.linalg import Vectors

v1 = Vectors.dense(3,2,1)
print(v1)
# [3.0,  2.0,   1.0]

## sparse (size of array,  indexe array,  value array)
v2 = Vectors.sparse(10, (0, 9), (100, 200))
print(v2)
# (10,[0,9],[100.0,200.0])

print(v2.toArray())
# [ 100.   0.   0.   0.   0.   0.   0.   0.   0. 200.]
```

ML Utilities

ML Overview
➔ **ML Utilities**
Transformers
Pipelines

Splitting Data Into Training / Test Subsets

◆ **Dataframe.randomSplit (weights)**

◆ **Dataframe.randomSplit(weights, seed)**
  - Use the 'seed' to consistent split

◆ Weights should add up to 1.0

```
(train, test) = df.randomSplit( [0.7, 0.3])
```

Training / Test Split Code (Python)

```python
df = spark.range(1,100)
df.show()
(train, test) = df.randomSplit([0.7, 0.3])
print("----training data set-----")
print("count: ", train.count())
train.show()

print("----testing data set-----")
print("count: ", test.count())
test.show()

common = train.intersect(test)
print("----common data set-----")
print("count: ", common.count())
common.show()
```

```
# df
Count= 100


+---+
| id|
+---+
|  1|
|  2|
|  3|
|  4|
|  5|
|  6|
|  7|
|  8|
|  9|
| 10|
```

```
# train
Count= 69


+---+
| id|
+---+
|  3|
|  4|
|  5|
|  6|
|  7|
|  9|
| 11|
| 12|
| 13|
| 17|
```

```
# test
Count= 31


+---+
| id|
+---+
|  1|
|  2|
|  8|
| 10|
| 15|
| 16|
| 18|
| 21|
| 23|
| 26|
```

```
# common
Count= 0


+---+
| id|
+---+
```

ML Transformers

ML Overview

Utilities

➔ **Transformers**

Pipelines

Transformers

◆ 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.

A	B	C

Transformer →

X	Y

VectorAssembler

◆ Transforms a Dataframe To Another Dataframe
  − By adding (or appending) to a "**features**" column

Car name	mpg	cyl	hp
Mazda RX	21	6	110
Merc 240D	24	4	62
Lincoln Continental	10.4	8	215
Toyota Corolla	33.9	4	65

VectorAssembler [mpg + cyl]

Car name	mpg	cyl	hp	Features
Mazda RX	21	6	110	**[21, 6]**
Merc 240D	24	4	62	**[24, 4]**
Lincoln Continental	10	8	215	**[10, 8]**
Toyota Corolla	34	4	65	**[34, 4]**

VectorAssembler Example Code (Python)

```python
from pyspark.ml.feature import VectorAssembler

data = spark.read.csv("mtcars_header.csv", header=True,
inferSchema=True)
mpg_cyl = data.select("model", "mpg", "cyl")
mpg_cyl.show()

assembler = VectorAssembler(inputCols=["mpg", "cyl"],
outputCol="features")
feature_vector = assembler.transform(mpg_cyl)
feature_vector.show(40)
```

```
## mpg_cyl

+-------------------+----+---+
|              model| mpg|cyl|
+-------------------+----+---+
|          Mazda RX4|21.0|  6|
|      Mazda RX4 Wag|21.0|  6|
|         Datsun 710|22.8|  4|
|      Hornet 4 Drive|21.4|  6|
|   Hornet Sportabout|18.7|  8|
```

```
## feature_vector

+-------------------+----+---+----------+
|              model| mpg|cyl|  features|
+-------------------+----+---+----------+
|          Mazda RX4|21.0|  6|[21.0,6.0]|
|      Mazda RX4 Wag|21.0|  6|[21.0,6.0]|
|         Datsun 710|22.8|  4|[22.8,4.0]|
|      Hornet 4 Drive|21.4|  6|[21.4,6.0]|
|   Hornet Sportabout|18.7|  8|[18.7,8.0]|
```

String Indexer

◆ Converts string based values into numeric values

◆ Numeric values are in [0, Number of Labels-1]

◆ Most frequently used label gets 0 and so on

id	color
1	red
2	white
3	blue
4	blue
5	white
6	yellow
7	blue
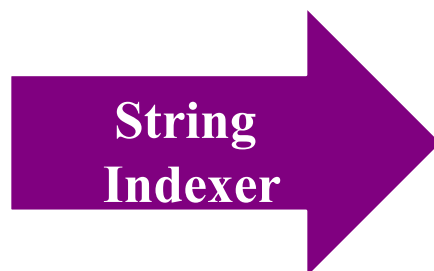
String Indexer →

id	color	Color index
1	red	3.0
2	white	1.0
3	blue	0.0
4	blue	0.0
5	white	1.0
6	yellow	2.0
7	blue	0.0

String Indexer Example Code (Python)

```python
import pandas as pd
from pyspark.ml.feature import IndexToString, StringIndexer

df_pd = pd.DataFrame({"id":[1,2,3,4,5,6,7],
            "color":['red', 'white', 'blue', 'blue', 'white' ,'yellow', 'blue'
            ]})
df_spark = spark.createDataFrame(df_pd)

str_indexer = StringIndexer(inputCol="color", outputCol="colorIndex")
model = str_indexer.fit(df_spark)
indexed = model.transform(df_spark)
```

```
+------+---+
| color| id|
+------+---+
|   red|  1|
| white|  2|
|  blue|  3|
|  blue|  4|
| white|  5|
|yellow|  6|
|  blue|  7|
+------+---+
```

String Indexer

```
+------+---+----------+
| color| id|colorIndex|
+------+---+----------+
|   red|  1|       3.0|
| white|  2|       1.0|
|  blue|  3|       0.0|
|  blue|  4|       0.0|
| white|  5|       1.0|
|yellow|  6|       2.0|
|  blue|  7|       0.0|
+------+---+----------+
```

Reverse String Indexer Example Code (Python)

```
converter = IndexToString(inputCol="colorIndex", outputCol="originalColor")
converted = converter.transform(indexed)
converted.show()
```

```
+------+---+                +------+---+----------+          +------+---+----------+-------------+
| color| id|                | color| id|colorIndex|          | color| id|colorIndex|originalColor|
+------+---+                +------+---+----------+          +------+---+----------+-------------+
|   red|  1|                |   red|  1|       3.0|          |   red|  1|       3.0|          red|
| white|  2|                | white|  2|       1.0|          | white|  2|       1.0|        white|
|  blue|  3|                |  blue|  3|       0.0|          |  blue|  3|       0.0|         blue|
|  blue|  4|                |  blue|  4|       0.0|          |  blue|  4|       0.0|         blue|
| white|  5|                | white|  5|       1.0|          | white|  5|       1.0|        white|
|yellow|  6|                |yellow|  6|       2.0|          |yellow|  6|       2.0|       yellow|
|  blue|  7|                |  blue|  7|       0.0|          |  blue|  7|       0.0|         blue|
+------+---+                +------+---+----------+          +------+---+----------+-------------+
```

One Hot Encoding

- Most ML algorithms need numeric data
- So we need to convert categorical / string data into numerical data before training the model
- Below we see two approaches of encoding 'marital status' data
- The one in middle has various indexes
- The one in right creates 'dummy variables' and assigns true / false to each.
  - Note, only one bit is on
  - This is called **ONE-HOT-Encoding**

id	status
1	married
2	single
3	married
4	Divorced
5	single

id	Status idx
1	0
2	1
3	0
4	2
5	1

id	is married	is single	is divorced
1	1	0	0
2	0	1	0
3	1	0	0
4	0	0	1
5	0	1	0

Hot Encoder Code (Python)

```python
import pandas as pd
from pyspark.ml.feature import StringIndexer, OneHotEncoder

df2_pd = pd.DataFrame({"id":[1,2,3,4,5,6,7],
                "status":['married', 'single', 'single', 'divorced', 'married'
                ,'single', 'married' ]})
df2_spark = spark.createDataFrame(df2_pd)

# first String Indexer
string_indexer = StringIndexer(inputCol="status", outputCol="statusIndex")
model = string_indexer.fit(df2_spark)
indexed = model.transform(df2_spark)

encoder = OneHotEncoder(inputCol="statusIndex", outputCol="statusVector",
dropLast=False)
encoded = encoder.transform(indexed)
encoded.show()

print(encoded.toPandas()) # print pandas df
```
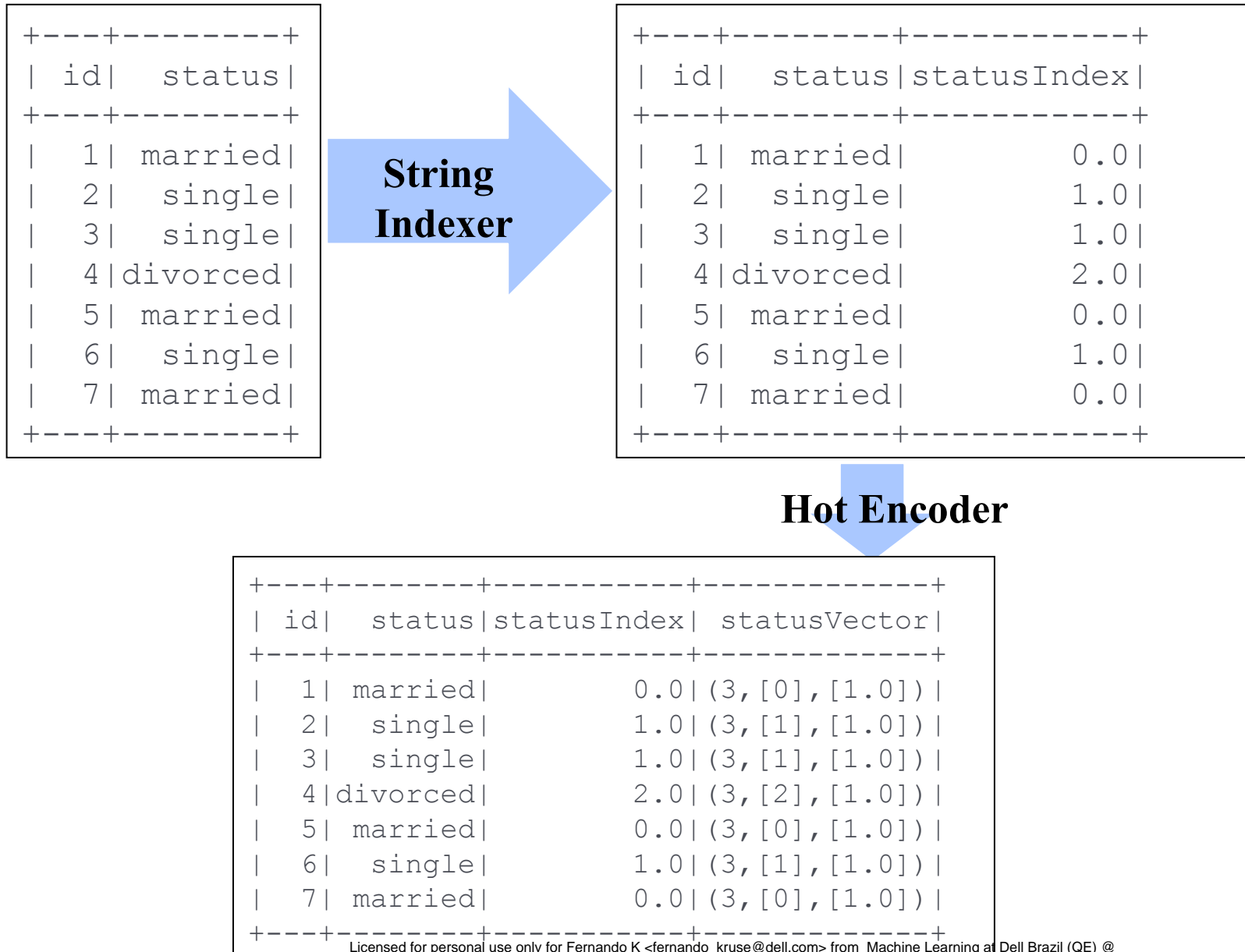
Hot Encoder Code (Python)

```
+---+--------+
| id|  status|
+---+--------+
|  1| married|
|  2|  single|
|  3|  single|
|  4|divorced|
|  5| married|
|  6|  single|
|  7| married|
+---+--------+
```

String Indexer

```
+---+--------+-----------+
| id|  status|statusIndex|
+---+--------+-----------+
|  1| married|        0.0|
|  2|  single|        1.0|
|  3|  single|        1.0|
|  4|divorced|        2.0|
|  5| married|        0.0|
|  6|  single|        1.0|
|  7| married|        0.0|
+---+--------+-----------+
```

Hot Encoder

```
+---+--------+-----------+-------------+
| id|  status|statusIndex| statusVector|
+---+--------+-----------+-------------+
|  1| married|        0.0|(3,[0],[1.0])|
|  2|  single|        1.0|(3,[1],[1.0])|
|  3|  single|        1.0|(3,[1],[1.0])|
|  4|divorced|        2.0|(3,[2],[1.0])|
|  5| married|        0.0|(3,[0],[1.0])|
|  6|  single|        1.0|(3,[1],[1.0])|
|  7| married|        0.0|(3,[0],[1.0])|
+---+--------+-----------+-------------+
```

◆ We are converting Spark DF → Pandas DF.
  So the spare vector array is displayed.

```
+---+--------+-----------+--------------+
| id|  status|statusIndex| statusVector|
+---+--------+-----------+--------------+
|  1| married|        0.0|(3,[0],[1.0])|
|  2|  single|        1.0|(3,[1],[1.0])|
|  3|  single|        1.0|(3,[1],[1.0])|
|  4|divorced|        2.0|(3,[2],[1.0])|
|  5| married|        0.0|(3,[0],[1.0])|
|  6|  single|        1.0|(3,[1],[1.0])|
|  7| married|        0.0|(3,[0],[1.0])|
+---+--------+-----------+--------------+
```

```
id    status  statusIndex     statusVector
  1   married          0.0  (1.0, 0.0, 0.0)
  2    single          1.0  (0.0, 1.0, 0.0)
  3    single          1.0  (0.0, 1.0, 0.0)
  4  divorced          2.0  (0.0, 0.0, 1.0)
  5   married          0.0  (1.0, 0.0, 0.0)
  6    single          1.0  (0.0, 1.0, 0.0)
  7   married          0.0  (1.0, 0.0, 0.0)
```
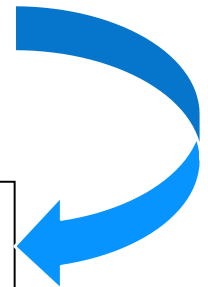
Scaling Data

- ◆ Sometimes we want to scale input data, so the algorithms produce better results
- ◆ Scaling prevents against features with very large variances exerting an overly large influence during model training.
- ◆ Consider the following data
  Salary with its larger range, might influence the outcome more
- ◆ Scaling can improve the convergence rate during the optimization process
- ◆ Spark ML has
  - − Standard Scaler
  - − MinMax Scaler

Age	Salary
20	50,000
23	65,000
40	100,000
35	86,000
30	75,000

Scaling: Standard Scalar

◆ **StandardScaler** standardizes features by scaling to unit variance and around mean (can be zeroed optionally)

◆ Uses column summary statistics on the samples in the training set

◆ This is a very common pre-processing step

```python
import pandas as pd
from pyspark.ml.feature import VectorAssembler

df_pd = pd.DataFrame({
    "home_runs": [ 30,  22,  17,  12, 44,   38,  40],
    "salary_in_k":[ 700, 450,340, 250, 1200, 800, 950 ]})
df_spark = spark.createDataFrame(df_pd)

assembler = VectorAssembler(inputCols=["home_runs", "salary_in_k"],
outputCol="features")
feature_vector = assembler.transform(df_spark)
feature_vector.show()
```

```
+---------+-----------+-------------+
|home_runs|salary_in_k|     features|
+---------+-----------+-------------+
|       30|        700| [30.0,700.0]|
|       22|        450| [22.0,450.0]|
|       17|        340| [17.0,340.0]|
|       12|        250| [12.0,250.0]|
|       44|       1200|[44.0,1200.0]|
|       38|        800| [38.0,800.0]|
|       40|        950| [40.0,950.0]|
+---------+-----------+-------------+
```

```python
from pyspark.ml.feature import StandardScaler
scaler = StandardScaler(inputCol="features",
                        outputCol="scaled_features",
                        withStd=True, withMean=False)
scalerModel = scaler.fit(feature_vector)
scaledData = scalerModel.transform(feature_vector)
scaledData.show()
```

```
+---------+-----------+------------+------------------------------------------+
|home_runs|salary_in_k|features    |scaled_features                           |
+---------+-----------+------------+------------------------------------------+
|30       |700        |[30.0,700.0]|[2.435993828823451,2.03376119068933]      |
|22       |450        |[22.0,450.0]|[1.7863954744705306,1.3074179083002835]|
|17       |340        |[17.0,340.0]|[1.3803965029999554,0.98782864049103] |
|12       |250        |[12.0,250.0]|[0.9743975315293804,0.7263432823890463]|
|44       |1200       |[44.0,1200.0]|[3.572790948941061,3.4864477554674225] |
|38       |800        |[38.0,800.0]|[3.085592183176371,2.32429850364494 8]|
|40       |950        |[40.0,950.0]|[3.2479917717646014,2.760104473078376] |
+---------+-----------+------------+------------------------------------------+
```

Scaling : MinMaxScaler

◆ **MinMax Scaler allows you to scale data at arbitrary range – 0.0 to 1.0 is default or 0 to 100)**

```python
from pyspark.ml.feature import MinMaxScaler

mmScaler = MinMaxScaler(min=1, max=100,
                        inputCol="features",
                        outputCol="scaled_features2")
scaledModel2 = mmScaler.fit(feature_vector)
scaledData2 = scaledModel2.transform(feature_vector)
scaledData2.show(10, False)
```

```
+---------+-----------+-------------+----------------------------+
|home_runs|salary_in_k|features     |scaled_features2            |
+---------+-----------+-------------+----------------------------+
|30       |700        |[30.0,700.0] |[56.6875,47.89473684210526] |
|22       |450        |[22.0,450.0] |[31.9375,21.842105263157894]|
|17       |340        |[17.0,340.0] |[16.46875,10.378947368421054]|
|12       |250        |[12.0,250.0] |[1.0,1.0]                    |
|44       |1200       |[44.0,1200.0]|[100.0,100.0]               |
|38       |800        |[38.0,800.0] |[81.4375,58.31578947368421] |
|40       |950        |[40.0,950.0] |[87.625,73.94736842105263]  |
+---------+-----------+-------------+----------------------------+
```

2019-03-12 **34**

Creating Vectors From Text

◆ TF/IDF: Term Frequency/Inverse Document Frequency
   − This essentially means the frequency of a term divided by its frequency in the larger group of documents (the "corpus")
   − Each word in the corpus is then a "dimension" – you would have thousands of dimensions.

◆ Word2Vec
   − Created at Google

Spark ML Workflow

Lab 5.1: ML Basics

Lab
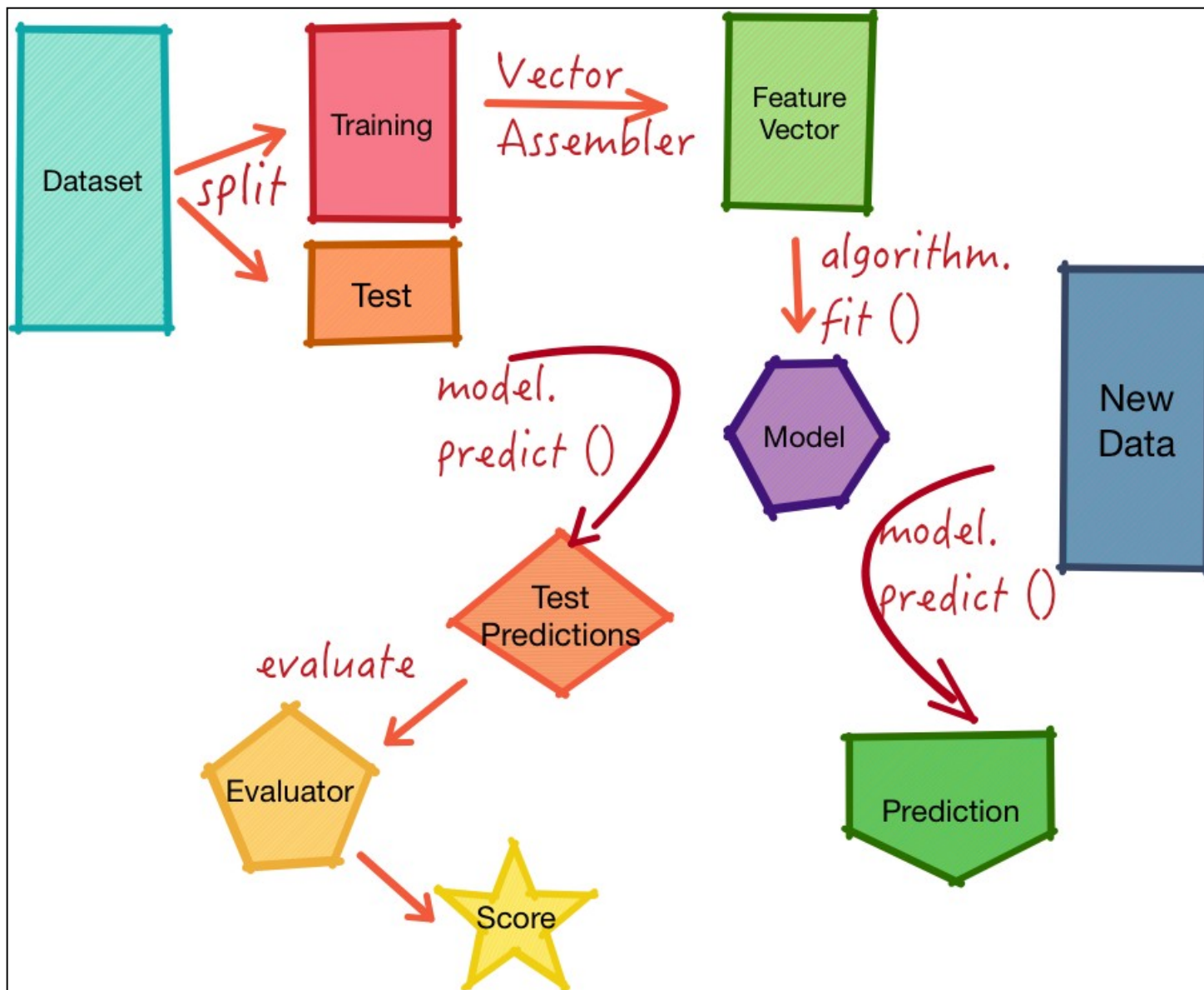
◆ **Overview**:
Get familiar with ML APIs in Spark

◆ **Approximate time**:
10 – 15 mins

◆ **Instructions**:
 − **5.1 : 'basics/spark-ml-basics'** lab for Scala / Python

Pipelines

ML Overview

Utilities

Transformers

➔ **Pipelines**

◆ Spark ML Pipelines is a powerful concept combining multiple steps to be carried out as a single unit
  − Reusable, repeatable
  − Makes very modular code

◆ This feature is modelled after the Python 'Scikit.Learn' pipeline feature

◆ Also allows tuning various parameters of the pipeline. 'Hyper Tuning'

Pipeline Example

- ◆ Imagine a text processing task.
- ◆ On left are individual steps
- ◆ On right we create a pipeline encompassing multiple steps
- ◆ This pipeline is re-useable by other programs too!

```
# text processing

df1 = spark.read(...)

# step1 - lower case the text
df2 = df1.lowercase()

# step2 - remove numbers /
punctuations
df3 = df2.removeNumbersPunct()

# step3 - break into words
df4 = df3.splitIntoWords()

# step4 - create word vectors
df5 = df4.word2Vec()

# process df5
```
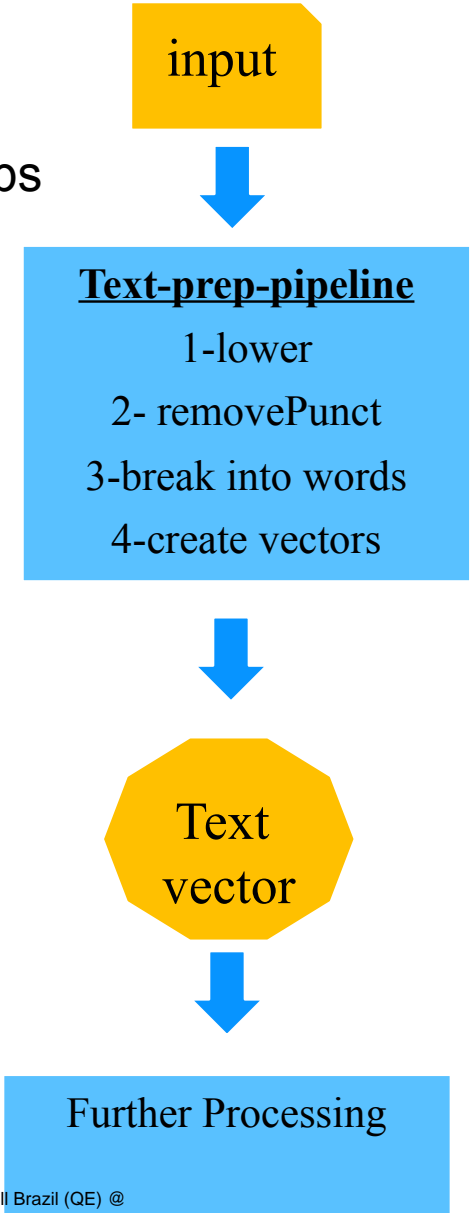
input

Text-prep-pipeline

1-lower

2- removePunct

3-break into words

4-create vectors

Text vector

Further Processing

40

- ◆ **Dataframe**: Contains data
- ◆ **Transformer**: Converts one dataframe into another

A	B	C

→ **Transformer** →

X	Y

- ◆ **Estimator**: fits the data in Dataframe to create a transformer.
  - − E.g. a learning model is an estimator

A	B	C

→ **Estimator.fit()** → **Model**
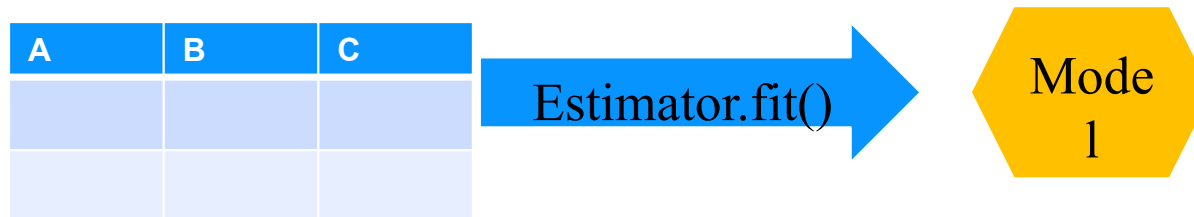
- ◆ **Pipeline**: Contains multiple Transformers and Estimators
- ◆ **Parameter**: Parameters can be passed uniformly to all components within a pipeline

Pipeline Example Code (Python)

◆ Here we are creating a pipeline consisting of 3 stages
  − Tokenizer: breaks text into words
  − HashingTF: converts words into Vector
  − And finally, a LogisticRegression model
◆ Also note, we train the model on the **entire** pipeline in one go!

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

training_data = spark.read(....)

# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol="words", outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training_data)

test_data = spark.read(...)
predicted_data = model.predict(test_data)
```
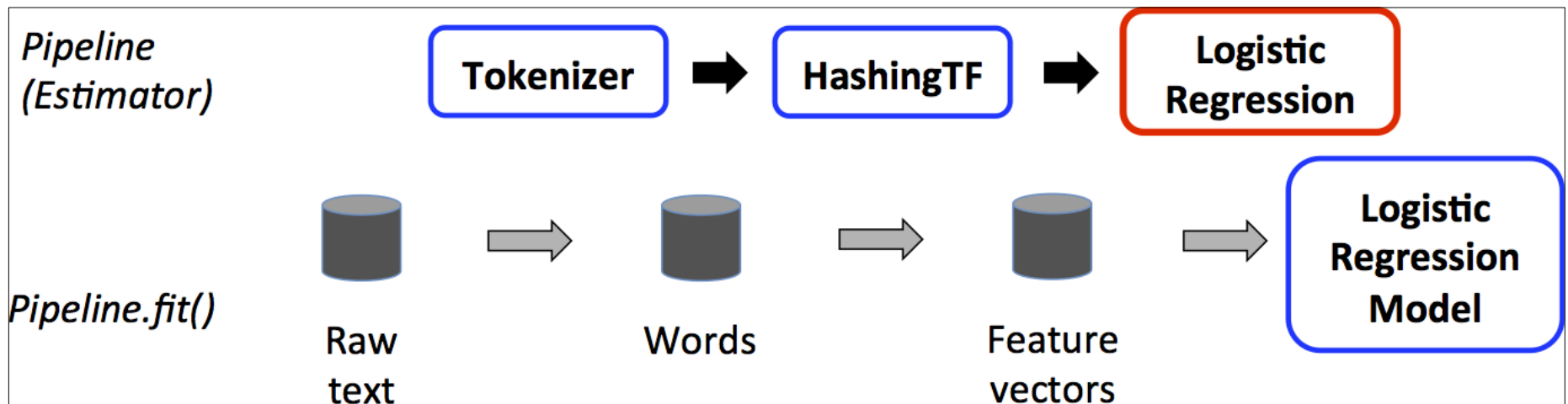
Pipeline Explained

◆ 3 stage pipeline shown

◆ First two (Tokenizer, hashingTF) are transformers (blue), third LogisticRegression is estimator (red)

◆ Pipeline executes 'transform' on first two and 'fit' on Logistic Regression
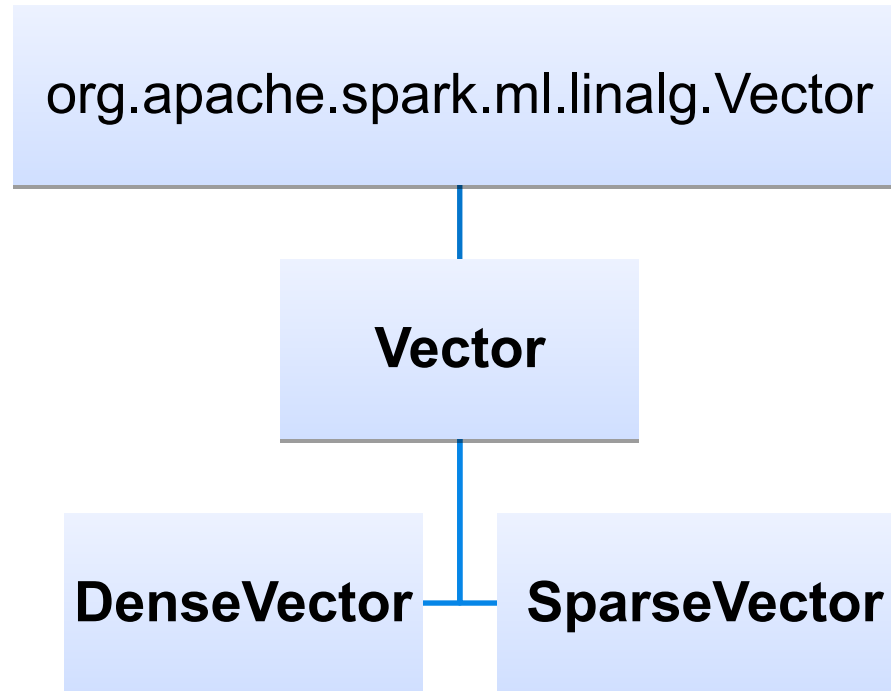
Further Reading

◆ http://spark.apache.org/docs/latest/ml-guide.html

◆ https://www.slideshare.net/julesdamji/apache-spark-mllib-2x-how-to-productionize-your-machine-learning-models

Backup Slides

Vectors in Spark

```
org.apache.spark.ml.linalg.Vector
              |
           Vector
              |
    ┌─────────┴─────────┐
DenseVector        SparseVector
```