# Big data: architectures and data analytics

## Spark MLlib

## Spark MLlib

- Spark MLlib is the Spark component providing the machine learning/data mining algorithms
  - Pre-processing techniques
  - Classification (supervised learning)
  - Clustering (unsupervised learning)
  - Itemset mining

# Spark MLlib

- MLlib APIs are divided into two packages:
  - org.apache.spark.mllib
    - It contains the original APIs built on top of RDDs
    - This version of the APIs is in maintenance mode and will be probably deprecated in the next releases of Spark
  - org.apache.spark.ml
    - It provides higher-level API built on top of DataFrames (i.e, Dataset<Row>) for constructing ML pipelines
    - It is recommended because the DataFrame-based API is more versatile and flexible
    - It provides the pipeline concept

## Spark MLlib – Data types

## Spark MLlib – Data types

- Spark MLlib is based on a set of basic local and distributed data types
  - Local vector
  - Labeled point
  - Local matrix
  - Distributed matrix
  - ..
- DataFrames for ML contain objects based on those basic data types

## Local vectors

- Local org.apache.spark.ml.linalg.Vector objects in MLlib are used to store vectors of double values
  - Dense and sparse vectors are supported
- The MLlib algorithms work on vectors of doubles
  - Vectors of doubles are used to represent the input records/data
    - One vector for each input record
  - Non double attributes/values must be mapped to double values

## Local vectors

- Dense and sparse representations are supported
- E.g., a vector (1.0, 0.0, 3.0) can be represented
  - in dense format as [1.0, 0.0, 3.0]
  - or in sparse format as (3, [0, 2], [1.0, 3.0])
    - where 3 is the size of the vector
    - The array [0,2] contains the indexes of the non-zero cells
    - The array [1.0, 3.0] contains the values of the non-zero cells

## Local vectors

- The following code shows how a vector can be created in Spark

```
import org.apache.spark.ml.linalg.Vector;
import org.apache.spark.ml.linalg.Vectors;

// Create a dense vector (1.0, 0.0, 3.0).
Vector dv = 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 non-zero entries
Vector sv = Vectors.sparse(3, new int[] {0, 2},
                              new double[] {1.0, 3.0});
```

# Local vectors

- The following code shows how a vector can be created in Spark

```
import org.apache.spark.ml.linalg.Vector;
import org.apache.spark.ml.linalg.Vectors;

// Create a dense vector (1.0, 0.0, 3.0).
Vector dv = 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 non-zero entries
Vector sv = Vectors.sparse(3, new int[] {0, 2},
                           new double[] {1.0, 3.0});
```

Indexes of non-empty cells

Size of the vector

Values of non-empty cells

# Labeled points

- Local org.apache.spark.ml.feature.LabeledPoint objects are local vectors of doubles associated with a label
  - The label is a double value
    - For the classification problem, each class label is associated with an integer value (casted to a double) ranging from 0 to C-1, where C is the number of distinct classes
    - For the regression problem, the label is the real value to predict
  - Both dense and sparse vectors associated with a label are supported

# Labeled points

- LabeledPoint objects are created by invoking the **LabelPoint LabeledPoint(double label, Vector features)** constructor
  - Note that label is a double and also Vector is a vector of doubles
- Given a LabeledPoint
  - The **double label()** method returns the value of its label
  - The **org.apache.spark.ml.linalg.Vector features()** method returns the vector containing the values of its attributes/features

## Labeled points

- In MLlib, labeled points are used by supervised (classification and regression) machine learning algorithms to represent records/data points
  - The **label** part represents the **target** of the analysis
  - The **features** part represents the **predictive attributes**/features that are used to predict the target attribute, i.e., the value of label

## Labeled points: classification example

- Suppose the analyzed records/data points are characterized by
  - 3 real (predictive) attributes/features
  - A class label attribute that can assume two values: 0 or 1
    - This is a binomial classification problem
- We want to predict the value of the class label attribute based on the values of the other attributes/features

## Labeled points: classification example

- Consider the following two records/data points
  - Attributes/features = [1.0,0.0,3.0] -- Label = 1
  - Attributes/features = [2.0,5.0,3.0] -- Label = 0
- Two LabeledPoint objects to represent those two data points in Spark

## Labeled points: classification example

```
import org.apache.spark.ml.linalg.Vectors;
import org.apache.spark.ml.feature.LabeledPoint;

// Create a LabeledPoint for the first record/data point
LabeledPoint record1=
	new LabeledPoint(1, Vectors.dense(1.0, 0.0, 3.0));

// Create a LabeledPoint for the second record/data point
LabeledPoint record2=
	new LabeledPoint(0, Vectors.dense(2.0, 5.0, 3.0));
```

# Labeled points: classification example

```
import org.apache.spark.ml.linalg.Vectors;
import org.apache.spark.ml.feature.LabeledPoint;

// Create a
LabeledPoint record1=
      new LabeledPoint(1, Vectors.dense(1.0, 0.0, 3.0));

// Create a LabeledPoint for the second record/data point
LabeledPoint record2=
      new LabeledPoint(0, Vectors.dense(2.0, 5.0, 3.0));
```

Value of the class label

# Labeled points: classification example

```
import org.apache.spark.ml.linalg.Vectors;
import org.apache.spark.ml.feature.LabeledPoint;

// Create a LabeledPoint
LabeledPoint record1=
        new LabeledPoint(1, Vectors.dense(1.0, 0.0, 3.0));

// Create a LabeledPoint for the second record/data point
LabeledPoint record2=
        new LabeledPoint(0, Vectors.dense(2.0, 5.0, 3.0));
```

Vector of doubles representing the values of the predictive features/attributes

## Labeled points and sparse vectors: classification example

- Consider again the following two records/data points
  - Attributes/features = [1.0,0.0,3.0] -- Label = 1
  - Attributes/features = [2.0,5.0,3.0] -- Label = 0
- Now we will use sparse vectors instead of dense ones to represent those two data points in Spark

## Labeled points and sparse vectors: classification example

```
import org.apache.spark.ml.linalg.Vectors;
import org.apache.spark.ml.feature.LabeledPoint;

// Create a LabeledPoint for the first record/data point
LabeledPoint record1= new LabeledPoint(1,
    Vectors.sparse(3, new int[] {0, 2}, new double[] {1.0, 3.0}) );

// Create a LabeledPoint for the second record/data point
LabeledPoint record2= new LabeledPoint(0,
    Vectors.sparse(3, new int[] {0, 1, 2}, new double[] {2.0, 5.0, 3.0}) );
```

## Spark MLlib - Main concepts

## Spark MLlib - Main concepts

- Spark MLlib uses DataFrames (alias of Dataset<Row>) as input data
- Hence, the input of the MLlib algorithms are structured data (i.e., tables)
- All input data must be represented by means of "tables" before applying the MLlib algorithms
  - Also the document collections must be transformed in a tabular format

## Spark MLlib - Main concepts

- The DataFrames used by the MLlib algorithms are characterized by several columns associated with different characteristics of the input data
  - label
    - Target of a classification/regression analysis
  - features
    - A vector containing the values of the attributes/features of the input record/data points
  - text
    - The original text of a document before being transformed in a tabular format
  - ..

## Spark MLlib - Main concepts

- Transformer
  - A Transformer is an ML algorithm/procedure that transforms one Dataset<Row> into another Dataset<Row>
    - E.g., A feature transformer might take a Dataset<Row>, read a column (e.g., text), map it into a new column (e.g., feature vectors), and output a new Dataset<Row> with the mapped column appended
    - E.g., a classification model is a Transformer that can be applied on a Dataset<Row> with features and transforms it into a Dataset<Row> with also the prediction column

## Spark MLlib - Main concepts

- Estimator
  - An Estimator is a ML algorithm/procedure that is applied on a Dataset<Row> to produce a Transformer (a model)
    - Each Estimator implements a method fit(), which accepts a Dataset<Row> and produces a **Model of type Transformer**
  - An Estimator abstracts the concept of a learning algorithm or any algorithm that fits or trains on an input dataset and returns a model
    - E.g., The Logistic Regression classification algorithm is an Estimator, and calling fit() on it a Logistic Regression Model is built, which is a Model and hence a Transformer

# Spark MLlib - Main concepts

- Pipeline
  - A Pipeline chains multiple Transformers and Estimators together to specify a Machine learning/Data Mining workflow
    - The output of a transformer/estimator is the input of the next one in the pipeline
  - E.g., a simple text document processing workflow aiming at building a classification model includes several steps
    - Split each document into a set of words
    - Convert each set of words into a numerical feature vector
    - Learn a prediction model using the feature vectors and the associated class labels

## Spark MLlib - Main concepts

- Parameter
  - All Transformers and Estimators share common APIs for specifying parameters

## Spark MLlib - Main concepts

- In the new APIs of Spark MLlib the use of the pipeline approach is preferred/recommended
- This approach is based on the following steps
  1. The set of Transformers and Estimators that are needed are instantiated
  2. A pipeline object is created and the sequence of transformers and estimators associated with the pipeline are specified
  3. The pipeline is executed and a model is created
  4. (optional) The model is applied on new data

## Classification algorithms

## Classification algorithms

- Spark MLlib provides a (limited) set of classification algorithms
  - Logistic regression
  - Decision trees
  - SVMs (with linear kernel)
    - Only binary classification problem are supported by the implemented SVMs
  - Naïve Bayes
  - ...

## Classification algorithms

- All the available classification algorithms are based on two phases
  - **Model generation** based on a set of **training data**
  - **Prediction** of the **class label** of new **unlabeled data**
- All the classification algorithms available in Spark work **only** on **numerical attributes**
  - Categorical values must be mapped to integer values (i.e., numerical values) before applying the MLlib classification algorithms

## Classification algorithms

- All the Spark classification algorithms are built on top of an input Dataset<Row> containing (at least) two columns
  - label
    - The class label, i.e., the attribute to be predicted by the classification model
      - It is an integer value (casted to a double)
  - features
    - A vector of doubles containing the values of the predictive attributes of the input records/data points
      - The data type of this column is org.apache.spark.ml.linalg.Vector

## Classification algorithms: Example of expected input Dataset<Row>

- Consider the following classification problem
  - We want to predict if new customers are good customers or not based on their monthly income and number of children
  - Predictive attributes
    - Monthly income
    - Number of children
  - Class Label (target attribute)
    - Customer type: Good customer/Bad customer
      - We map "Good customer" to 1 and "Bad customer" to 0

## Classification algorithms: Example of expected input Dataset<Row>

- Example of input training data
  - i.e., the set of customers for which the value of the class label is known
  - They are used by the classification algorithm to infer a classification model

| CustomerType | MonthlyIncome | NumChildren |
|---|---|---|
| Good customer | 1400.0 | 2 |
| Bad customer | 11105.5 | 0 |
| Good customer | 2150.0 | 2 |

## Classification algorithms: Example of expected input Dataset<Row>

- Input training data

| CustomerType | MonthlyIncome | NumChildren |
|---|---|---|
| Good customer | 1400.0 | 2 |
| Bad customer | 11105.5 | 0 |
| Good customer | 2150.0 | 2 |

- Input Dataset<Row> that must be generated as input for the MLlib classification algorithms

| label | features |
|---|---|
| 1.0 | [1400.0 , 2.0] |
| 0.0 | [11105.5, 0.0] |
| 1.0 | [2150.0 , 2.0] |

# Classification algorithms: Example of expected input Dataset<Row>

- Inpu

The categorical values of CustomerType (the class label column) must be mapped to integer values (finally casted to doubles)

| CustomerType | MonthlyIncome | NumChildren |
|---|---|---|
| Good customer | 1400.0 | 2 |
| Bad customer | 11105.5 | 0 |
| Good customer | 2150.0 | 2 |

- Input Dataset<Row> that must be generated as input for the MLlib classification algorithms

| label | features |
|---|---|
| 1.0 | [1400.0 , 2.0] |
| 0.0 | [11105.5, 0.0] |
| 1.0 | [2150.0 , 2.0] |

# Classification algorithms: Example of expected input Dataset<Row>

- Inpu

The values of the predictive attributes are "stored" in vectors of doubles. One single vector for each input record.

| CustomerType | MonthlyIncome | NumChildren |
|---|---|---|
| Good customer | 1400.0 | 2 |
| Bad customer | 11105.5 | 0 |
| Good customer | 2150.0 | 2 |

- Input Dataset<Row> that must be generated as input for the MLlib classification algorithms

| label | features |
|---|---|
| 1.0 | [1400.0 , 2.0] |
| 0.0 | [11105.5, 0.0] |
| 1.0 | [2150.0 , 2.0] |

# Classification algorithms: Example of expected input Dataset<Row>

- Inpu

In the generated Dataset<Row> the names of the predictive attributes are not preserved.

| CustomerType | MonthlyIncome | NumChildren |
|---|---|---|
| Good customer | 1400.0 | 2 |
| Bad customer | 11105.5 | 0 |
| Good customer | 2150.0 | 2 |

- Input Dataset<Row> that must be generated as input for the MLlib classification algorithms

| label | features |
|---|---|
| 1.0 | [1400.0 , 2.0] |
| 0.0 | [11105.5, 0.0] |
| 1.0 | [2150.0 , 2.0] |

## Logistic regression and structured data

## Logistic regression and structured data

- The following slides show how to
  - Create a classification model based on the **logistic regression algorithm**
    - The model is inferred by analyzing the training data, i.e., the example records/data points for which the value of the class label is known
  - Apply the model to new unlabeled data
    - The inferred model is applied to predict the value of the class label of new unlabeled records/data points

# Logistic regression and structured data: training data

- In the following example, the input training data is stored in a text file that contains
  - One record/data point per line
  - The records/data points are structured data with a fixed number of attributes (four)
    - One attribute is the class label
      - We suppose that the first column of each record contains the class label
    - The other three attributes are the predictive attributes that are used to predict the value of the class label
  - The input file has not the header line

## Logistic regression and structured data: training data

- Consider the following example input training data file

```
1.0,0.0,1.1,0.1
0.0,2.0,1.0,-1.0
0.0,2.0,1.3,1.0
1.0,0.0,1.2,-0.5
```

- It contains four records/data points
- This is a binary classification problem because the class label assumes only two values
    - 0 and 1

## Logistic regression and structured data: training data

- The first operation consists in transforming the content of the input training file into a Dataset<Row> containing two columns
  - label
  - features
- In the following solution, we will first define an RDD of LabeledPoint objects and then we will "transform" it into a Dataset<Row>
- However, other approaches can be used to achieve the same result

## Logistic regression and structured data: training data

- Input training file

```
1.0,0.0,1.1,0.1
0.0,2.0,1.0,-1.0
0.0,2.0,1.3,1.0
1.0,0.0,1.2,-0.5
```

- Input training Dataset<Row> to be generated

| label | features |
|---|---|
| 1.0 | [0.0,1.1,0.1] |
| 0.0 | [2.0,1.0,-1.0] |
| 0.0 | [2.0,1.3,1.0] |
| 1.0 | [0.0,1.2,-0.5] |

# Logistic regression and structured data: training data

- Input training file

```
1.0,0.0,1.1,0.1
0.0,2.0,1.0,-1.0
0.0,2.0,1.3,1.0
1.0,0.0,1.2,-0.5
```

Name of this column: label
Data type: double

- Input training Dataset<Row> to be generated

| label | features |
|---|---|
| 1.0 | [0.0,1.1,0.1] |
| 0.0 | [2.0,1.0,-1.0] |
| 0.0 | [2.0,1.3,1.0] |
| 1.0 | [0.0,1.2,-0.5] |

# Logistic regression and structured data: training data

- Input training file

```
1.0,0.0,1.1,0.1
0.0,2.0,1.0,-1.0
0.0,2.0,1.3,1.0
1.0,0.0,1.2,-0.5
```

Name of this column: features
Data type: org.apache.spark.ml.linalg.Vector

- Input training Dataset<Row> to be generated

| label | features |
|---|---|
| 1.0 | [0.0,1.1,0.1] |
| 0.0 | [2.0,1.0,-1.0] |
| 0.0 | [2.0,1.3,1.0] |
| 1.0 | [0.0,1.2,-0.5] |

## Logistic regression and structured data: unlabeled data

- The file containing the **unlabeled data** has the same format of the training data file
  - However, the **first column** is **empty** because the class label is **unknown**
- We want to predict the class label value of each unlabeled data by applying the classification model that has been inferred on the training data
- The predicted class label value is stored in a new column, called “prediction” of the returned Dataset<Row>

## Logistic regression and structured data: unlabeled data

- Consider the following example input unlabeled data file

```
,-1.0,1.5,1.3
,3.0,2.0,-0.1
,0.0,2.2,-1.5
```

- It contains three unlabeled records/data points
- Note that the first column is empty (the content before the first comma is the empty string)

## Logistic regression and structured data: unlabeled data

- Also the unlabeled data must be stored into a Dataset<Row> containing two columns
  - label
  - features
- A label value is required also for unlabeled data
  - Any numerical value can be used
    - The specified value does not impact on the prediction because the label column is not used to perform the prediction
  - Usually the value -1.0 is used

## Logistic regression and structured data: unlabeled data

- Input unlabeled data file

```
,-1.0,1.5,1.3
,3.0,2.0,-0.1
,0.0,2.2,-1.5
```

- Input unlabeled data Dataset<Row> to be generated

| label | features |
|---|---|
| -1.0 | [-1.0,1.5,1.3] |
| -1.0 | [3.0,2.0,-0.1] |
| -1.0 | [0.0,2.2,-1.5] |

# Logistic regression and structured data: unlabeled data

- Input unlabeled data file

```
,-1.0,1.5,1.3
```

A value must be specified also for the label column.
I arbitrarily set it to -1.0 for all records

- <Row> to be generated

| label | features |
|---|---|
| -1.0 | [-1.0,1.5,1.3] |
| -1.0 | [3.0,2.0,-0.1] |
| -1.0 | [0.0,2.2,-1.5] |

## Logistic regression and structured data: prediction column

- After the application of the classification model on the unlabeled data, Spark returns a new Dataset<Row> containing
  - The same columns of the input data
  - A new column called prediction
    - For each input unlabeled record, it contains the predicted class label value
  - Also other two columns, associated with the probabilities of the predictions, are returned
    - We do not consider them in the following example

# Logistic regression and structured data: prediction column

- Input unlabeled data Dataset<Row>

| label | feature |
|---|---|
| -1.0 | [-1.0,1.5,1.3] |
| -1.0 | [3.0,2.0,-0.1] |
| -1.0 | [0.0,2.2,-1.5] |

- Returned Dataset<Row> with the predicted class label values

| label | features | prediction | rawPrediction | probability |
|---|---|---|---|---|
| -1.0 | [-1.0,1.5,1.3] | 1.0 | ... | ... |
| -1.0 | [3.0,2.0,-0.1] | 0.0 | ... | ... |
| -1.0 | [0.0,2.2,-1.5] | 1.0 | ... | ... |

# Logistic regression and structured data: prediction column

- Input unlabeled data Dataset<Row>

| label | feature |
|---|---|
| -1.0 | [-1.0,1.5,1.3] |
| -1.0 | [3.0,2.0,-0.1] |
| -1.0 | [0.0,2.2 |

This column contains the predicted class label values

- Returned Dataset<Row> with the predicted class label values

| label | features | prediction | rawPrediction | probability |
|---|---|---|---|---|
| -1.0 | [-1.0,1.5,1.3] | 1.0 | … | … |
| -1.0 | [3.0,2.0,-0.1] | 0.0 | … | … |
| -1.0 | [0.0,2.2,-1.5] | 1.0 | … | … |

## Logistic regression and structured data: Example

```
package it.polito.bigdata.spark.sparkmllib;

import org.apache.spark.api.java.*;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.apache.spark.ml.Pipeline;
import org.apache.spark.ml.PipelineModel;
import org.apache.spark.ml.PipelineStage;
import org.apache.spark.ml.classification.LogisticRegression;
import org.apache.spark.ml.linalg.Vector;
import org.apache.spark.ml.linalg.Vectors;
import org.apache.spark.ml.feature.LabeledPoint;
```

# Logistic regression and structured data: Example

```
public class SparkDriver {
    public static void main(String[] args) {
        String inputFileTraining;      String inputFileTest;   String outputPath;
        inputFileTraining=args[0];
        inputFileTest=args[1];
        outputPath=args[2];

        // Create a Spark Session object and set the name of the application
        // We use some Spark SQL transformation in this program
        SparkSession ss = SparkSession.builder().
                        appName("MLlib - logistic regression").getOrCreate();

        // Create a Java Spark Context from the Spark Session
        // When a Spark Session has already been defined this method
        // is used to create the Java Spark Context
        JavaSparkContext sc = new JavaSparkContext(ss.sparkContext());
```

## Logistic regression and structured data: Example

```
// ************************
//Training step
// ************************

// Read training data from a text file
// Each line has the format: class-label, list of three numerical
// attribute values.
// E.g.,                    1.0,5.8,0.5,1.7
JavaRDD<String> trainingData=sc.textFile(inputFileTraining);
```

## Logistic regression and structured data: Example

```
// Map each input record/data point of the input file to a LabeledPoint
JavaRDD<LabeledPoint> trainingRDD=trainingData.map(record ->
        {
                String[] fields = record.split(",");
                // Fields of 0 contains the id of the class
                double classLabel = Double.parseDouble(fields[0]);

                // The other three cells of fields contain the (numerical)
                // values of the three predictive attributes
                // Create an array of doubles containing those values
                double[] attributesValues = new double[3];

                attributesValues[0] = Double.parseDouble(fields[1]);
                attributesValues[1] = Double.parseDouble(fields[2]);
                attributesValues[2] = Double.parseDouble(fields[3]);
```

## Logistic regression and structured data: Example

```
            // Create a dense vector based on the content of
            // attributesValues
            Vector attrValues= Vectors.dense(attributesValues);

            // Return a LabeledPoint based on the content of
            // the current line
            return new LabeledPoint(classLabel, attrValues);
});
```

# Logistic regression and structured data: Example

```
// Prepare training data.
// We use LabeledPoint, which is a JavaBean.
// We use Spark SQL to convert RDDs of JavaBeans
// into Dataset<Row>. The columns of the Dataset are label
// and features
Dataset<Row> training =
        ss.createDataFrame(trainingRDD, LabeledPoint.class).cache();
```

# Logistic regression and structured data: Example

```
// Prepare training data.
// We use LabeledPoint, which is a JavaBean.
// We use Spark SQL to convert RDDs of JavaBeans
// into Dataset<Row>. The columns of the Dataset are label
// and features
Dataset<Row> training =
          ss.createDataFrame(trainingRDD, LabeledPoint.class).cache();
```

The training data are represented by means of a Dataset<Row> of LabeledPoint.
Each element of this DataFrame has two columns:
-label: the class label
-features: the vector of real values associated with the attributes of the input record

This Dataset is cached because the logistic regression algorithm iterates and applies multiple actions on it.

## Logistic regression and structured data: Example

```
// Create a LogisticRegression object.
// LogisticRegression is an Estimator that is used to
// create a classification model based on logistic regression.
LogisticRegression lr = new LogisticRegression();

// We can set the values of the parameters of the
// Logistic Regression algorithm using the setter methods.
// There is one set method for each parameter
// For example, we are setting the number of maximum iterations to 10
// and the regularization parameter. to 0.0.1
lr.setMaxIter(10);
lr.setRegParam(0.01);


// Define the pipeline that is used to create the logistic regression
// model on the training data
// In this case the pipeline contains one single stage/step (the model
// generation step).
Pipeline pipeline = new Pipeline().setStages(new PipelineStage[] {lr});
```

# Logistic regression and structured data: Example

```
// Create a LogisticRegression object.
// LogisticRegression is an Estimator that is used to
// create a classification model based on logistic regression.
LogisticRegression lr = new LogisticRegression();

// We can set the values of the parameters of the
// Logistic Regression algorithm using the setter methods.
// There is one set method for each parameter
// For example, we are setting the number of maximum iterations to 10
// and the regularization parameter. to 0.0.1
lr.setMaxIter(10);
lr.setRegParam(0.01);
```

This is the sequence of Transformers and Estimators to apply on the training data. This simple pipeline contains only the logistic regression estimator

```
// In this case the pipeline contains one single stage/step (the model
// generation step).
Pipeline pipeline = new Pipeline().setStages(new PipelineStage[] {lr});
```

## Logistic regression and structured data: Example

```
// Execute the pipeline on the training data to build the
// classification model
PipelineModel model = pipeline.fit(training);

// Now, the classification model can be used to predict the class label
// of new unlabeled data
```

# Logistic regression and structured data: Example

```
// Execute the pipeline on the training data to build the
// classification model
PipelineModel model = pipeline.fit(training);

// Now, the classification model can be used to predict the class label
// of r
```

Execute the pipeline and train the model

# Logistic regression and structured data: Example

```
// *************************
// Prediction  step
// *************************

// Read unlabeled data
// For the unlabeled data only the predictive attributes are available
// The class label is not available and must be predicted by applying
// the classification model inferred during the previous phase
JavaRDD<String> unlabeledData=sc.textFile(inputFileTest);
```

## Logistic regression and structured data: Example

```
// Map each unlabeled input record/data point of the input file to
// a LabeledPoint
JavaRDD<LabeledPoint> unlabeledRDD=unlabeledData.map(record ->
{
		String[] fields = record.split(",");

		// The last three cells of fields contain the (numerical) values of the
		// three predictive attributes
		// Create an array of doubles containing those three values
		double[] attributesValues = new double[3];

		attributesValues[0] = Double.parseDouble(fields[1]);
		attributesValues[1] = Double.parseDouble(fields[2]);
		attributesValues[2] = Double.parseDouble(fields[3]);
```

## Logistic regression and structured data: Example

```
            // Create a dense vector based in the content of attributesValues
            Vector attrValues= Vectors.dense(attributesValues);

            // The class label in unknown.
            // To create a LabeledPoint a class label value must be specified
            // also for the unlabeled data. I set it to -1 (an invalid value).
            // The specified value  does not impact on the prediction because
            // the label column is not used to perform the prediction
            double classLabel = -1;

            // Return a new LabeledPoint
            return new LabeledPoint(classLabel, attrValues);
});

// Create the DataFrame based on the new test data
Dataset<Row> test =
            ss.createDataFrame(unlabeledRDD, LabeledPoint.class);
```

# Logistic regression and structured data: Example

```
// Make predictions on test documents using the transform()
// method.
// The transform will only use the 'features' columns
Dataset<Row> predictions = model.transform(test);

// The returned Dataset<Row> has the following schema (attributes)
// - features: vector (values of the attributes)
// - label: double (value of the class label)
// - rawPrediction: vector (nullable = true)
// - probability: vector (The i-th cell contains the probability that the
//                                   current record belongs to the i-th class
// - prediction: double (the predicted class label)

// Select only the features (i.e., the value of the attributes) and
// the predicted class for each record
Dataset<Row> predictionsDF=predictions.select("features", "prediction");
```

# Logistic regression and structured data: Example

```
// Make predictions on test documents using the transform()
// method.
// The transform will only use the 'features' columns
Dataset<Row> predictions = model.transform(test);

// The returned Dataset<Row> has the following schema (attributes)
```

The model is applied to new data/records and the class label is predicted for each new data/record.
The new generated Dataset<Row> has the same attributes of the input Dataset<Row> and the prediction attribute (and also some other related attributes).

```
// - prediction: double (the predicted class label)

// Select only the features (i.e., the value of the attributes) and
// the predicted class for each record
Dataset<Row> predictionsDF=predictions.select("features", "prediction");
```

# Logistic regression and structured data: Example

```
// Make predictions on test documents using the transform()
// method.
//The transform will only use the 'features' columns
Dataset<Row> predictions = model.transform(test);

//The returned Dataset<Row> has the following schema (attributes)
// - features: vector (values of the attributes)
// - label: double (value of the class label)
// - rawPrediction: vector (nullable = true)
// - probability: vector (The i-th cell contains the probability that the
//                                  current record belongs to the i-th class
```

The predictive attributes and the predicted class are selected

```
// Select only the features (i.e., the value of the attributes) and
// the predicted class for each record
Dataset<Row> predictionsDF=predictions.select("features", "prediction");
```

# Logistic regression and structured data: Example

```
            // Save the result in an HDFS file
            JavaRDD<Row> predictionsRDD = predictionsDF.javaRDD();
            predictionsRDD.saveAsTextFile(outputPath);


            // Close the Spark Context object
            sc.close();
    }
}
```

## Decision trees and structured data

## Decision trees and structured data

- The following slides show how to
  - Create a classification model based on the **decision tree algorithm**
    - The model is inferred by analyzing the training data, i.e., the example records/data points for which the value of the class label is known
  - Apply the model to new unlabeled data
    - The inferred model is applied to predict the value of the class label of new unlabeled records/data points

## Decision trees and structured data

- The same example structured data already used in the running example related to the logistic regression algorithm are used also in this example related to the decision tree algorithm

## Decision trees and structured data

- In the following example, the input training data is stored in a text file that contains
  - One record/data point per line
  - The records/data points are structured data with a fixed number of attributes (four)
    - One attribute is the class label
      - We suppose that the first column of each record contains the class label
    - The other three attributes are the predictive attributes that are used to predict the value of the class label
  - The input file has not the header line

## Decision trees and structured data: training data

- Also in this case the training data must be represented by means of a Dataset<Row> characterized by two columns:
  - label
  - features
- The code that is used to solve this problem is similar to the one we used to build a logistic regression model
  - The only difference is given by the use of the Decision Tree algorithm instead of the Logistic regression one in the defined Pipeline

## Decision trees and structured data: training data

- Input training file

```
1.0,0.0,1.1,0.1
0.0,2.0,1.0,-1.0
0.0,2.0,1.3,1.0
1.0,0.0,1.2,-0.5
```

- Input training Dataset<Row> to be generated

| label | features |
|---|---|
| 1.0 | [0.0,1.1,0.1] |
| 0.0 | [2.0,1.0,-1.0] |
| 0.0 | [2.0,1.3,1.0] |
| 1.0 | [0.0,1.2,-0.5] |

## Decision trees and structured data: unlabeled data

- The file containing the **unlabeled data** has the same format of the training data file
  - However, the **first column** is **empty** because the class label is **unknown**
- We want to predict the class label value of each unlabeled data by applying the classification model that has been inferred on the training data

## Decision trees and structured data: unlabeled data

- Input unlabeled data file

```
,-1.0,1.5,1.3
,3.0,2.0,-0.1
,0.0,2.2,-1.5
```

- Input unlabeled data Dataset<Row> to be generated

| label | features |
|---|---|
| -1.0 | [-1.0,1.5,1.3] |
| -1.0 | [3.0,2.0,-0.1] |
| -1.0 | [0.0,2.2,-1.5] |

## Decision trees and structured data: prediction column

- After the application of the classification model on the unlabeled data, Spark returns a new Dataset<Row> containing
  - The same columns of the input data
  - A new column called prediction
    - For each input unlabeled record, it contains the predicted class label value
  - Also other columns, associated with the probabilities of the predictions, are returned
    - We do not consider them in the following example

## Decision trees and structured data: prediction column

- Input unlabeled data Dataset<Row>

| label | feature |
|---|---|
| -1.0 | [-1.0,1.5,1.3] |
| -1.0 | [3.0,2.0,-0.1] |
| -1.0 | [0.0,2.2,-1.5] |

- Returned Dataset<Row> with the predicted class label values

| label | features | prediction | rawPrediction | probability |
|---|---|---|---|---|
| -1.0 | [-1.0,1.5,1.3] | 1.0 | … | … |
| -1.0 | [3.0,2.0,-0.1] | 0.0 | … | … |
| -1.0 | [0.0,2.2,-1.5] | 1.0 | … | … |

# Decision trees and structured data: Example

```
package it.polito.bigdata.spark.sparkmllib;

import org.apache.spark.api.java.*;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.apache.spark.ml.Pipeline;
import org.apache.spark.ml.PipelineModel;
import org.apache.spark.ml.PipelineStage;
import org.apache.spark.ml.classification.DecisionTreeClassifier;
import org.apache.spark.ml.linalg.Vector;
import org.apache.spark.ml.linalg.Vectors;
import org.apache.spark.ml.feature.LabeledPoint;
```

## Decision trees and structured data: Example

```
public class SparkDriver {
    public static void main(String[] args) {
        String inputFileTraining;     String inputFileTest;   String outputPath;
        inputFileTraining=args[0];
        inputFileTest=args[1];
        outputPath=args[2];

        // Create a Spark Session object and set the name of the application
        // We use some Spark SQL transformation in this program
        SparkSession ss = SparkSession.builder().
                        .appName("MLlib - Decision Tree").getOrCreate();

        // Create a Java Spark Context from the Spark Session
        // When a Spark Session has already been defined this method
        // is used to create the Java Spark Context
        JavaSparkContext sc = new JavaSparkContext(ss.sparkContext());
```

# Decision trees and structured data: Example

```
// *************************
// Training step
// *************************

// Read training data from a text file
// Each line has the format: class-label, list of three numerical
// attribute values.
// E.g.,               1.0,5.8,0.5,1.7
JavaRDD<String> trainingData=sc.textFile(inputFileTraining);
```

## Decision trees and structured data: Example

```
// Map each input record/data point of the input file to a LabeledPoint
JavaRDD<LabeledPoint> trainingRDD=trainingData.map(record ->
        {
                String[] fields = record.split(",");
                // Fields of 0 contains the id of the class
                double classLabel = Double.parseDouble(fields[0]);

                // The other three cells of fields contain the (numerical)
                // values of the three predictive attributes
                // Create an array of doubles containing those values
                double[] attributesValues = new double[3];

                attributesValues[0] = Double.parseDouble(fields[1]);
                attributesValues[1] = Double.parseDouble(fields[2]);
                attributesValues[2] = Double.parseDouble(fields[3]);
```

## Decision trees and structured data: Example

```
                // Create a dense vector based on the content of
                // attributesValues
                Vector attrValues= Vectors.dense(attributesValues);

                // Return a LabeledPoint based on the content of
                // the current line
                return new LabeledPoint(classLabel, attrValues);
        });
```

## Decision trees and structured data: Example

```
// Prepare training data.
// We use LabeledPoint, which is a JavaBean.
// We use Spark SQL to convert RDDs of JavaBeans
// into Dataset<Row>. The columns of the Dataset are label
// and features
Dataset<Row> training =
        ss.createDataFrame(trainingRDD, LabeledPoint.class).cache();
```

## Decision trees and structured data: Example

```
// Create a DecisionTreeClassifier object.
// DecisionTreeClassifier is an Estimator that is used to
// create a classification model based on decision trees.
// The algorithm infers a model that can be used to predict the value
// of the label attribute based on content of vector
// that is stored in the feature attribute
DecisionTreeClassifier dc= new DecisionTreeClassifier();

// We can set the values of the parameters of the Decision Tree
// For example we can set the measure that is used to decide if a
// node must be split
// In this case we set gini index
dc.setImpurity("gini");

// Define the pipeline that is used to create the logistic regression
// model on the training data
// In this case the pipeline contains one single stage/step (the model
// generation step).
Pipeline pipeline = new Pipeline().setStages(new PipelineStage[] {dc});
```

# Decision trees and structured data: Example

```
// Create a DecisionTreeClassifier object.
// DecisionTreeClassifier is an Estimator that is used to
// create a classification model based on decision trees.
// The algorithm infers a model that can be used to predict the value
// of the label attribute based on content of vector
// that is stored in the feature attribute
DecisionTreeClassifier dc= new DecisionTreeClassifier();

// We can set the values of the parameters of the Decision Tree
// For example we can set the measure that is used to decide if a
// node must be split
// In this case we set gini index
dc.setImpurity("gini");

// Define the pipeline that is used to create the logistic regression
// model on the training data
// In this case the pipeline contains one single stage/step (the model
// generation step).
Pipeline pipeline = new Pipeline().setStages(new PipelineStage[] {dc});
```

We use a Decision Tree Classifier in this application.

## Decision trees and structured data: Example

```
// Execute the pipeline on the training data to build the
// classification model
PipelineModel model = pipeline.fit(training);

// Now, the classification model can be used to predict the class label
// of new unlabeled data
```

# Decision trees and structured data: Example

```
// *************************
// Prediction  step
// *************************

// Read unlabeled data
// For the unlabeled data only the predictive attributes are available
// The class label is not available and must be predicted by applying
// the classification model inferred during the previous phase
JavaRDD<String> unlabeledData=sc.textFile(inputFileTest);
```

## Decision trees and structured data: Example

```
// Map each unlabeled input record/data point of the input file to
// a LabeledPoint
JavaRDD<LabeledPoint> unlabeledRDD=unlabeledData.map(record ->
{
		String[] fields = record.split(",");

		// The last three cells of fields contain the (numerical) values of the
		// three predictive attributes
		// Create an array of doubles containing those three values
		double[] attributesValues = new double[3];

		attributesValues[0] = Double.parseDouble(fields[1]);
		attributesValues[1] = Double.parseDouble(fields[2]);
		attributesValues[2] = Double.parseDouble(fields[3]);
```

## Decision trees and structured data: Example

```
          // Create a dense vector based in the content of attributesValues
          Vector attrValues= Vectors.dense(attributesValues);

          // The class label in unknown.
          // To create a LabeledPoint a class label value must be specified
          // also for the unlabeled data. I set it to -1 (an invalid value).
          // The specified value  does not impact on the prediction because
          // the label column is not used to perform the prediction
          double classLabel = -1;

          // Return a new LabeledPoint
          return new LabeledPoint(classLabel, attrValues);
});

// Create the DataFrame based on the new test data
Dataset<Row> test =
          ss.createDataFrame(unlabeledRDD, LabeledPoint.class);
```

# Decision trees and structured data: Example

```
// Make predictions on test documents using the transform()
// method.
// The transform will only use the 'features' columns
Dataset<Row> predictions = model.transform(test);

// The returned Dataset<Row> has the following schema (attributes)
// - features: vector (values of the attributes)
// - label: double (value of the class label)
// - rawPrediction: vector (nullable = true)
// - probability: vector (The i-th cell contains the probability that the
//                                  current record belongs to the i-th class
// - prediction: double (the predicted class label)

// Select only the features (i.e., the value of the attributes) and
// the predicted class for each record
Dataset<Row> predictionsDF=predictions.select("features", "prediction");
```

# Decision trees and structured data: Example

```
        // Save the result in an HDFS file
        JavaRDD<Row> predictionsRDD = predictionsDF.javaRDD();
        predictionsRDD.saveAsTextFile(outputPath);


        // Close the Spark Context object
        sc.close();
    }
}
```

## Categorical class labels

## Categorical class labels

- Frequently the class label is a categorical value (i.e., a string)
- As reported before, Spark MLlib works only with numerical values and hence categorical class label values must be mapped to integer (and then double) values

# Categorical class labels

- Input training data

| categoricalLabel | Attr1 | Attr2 | Attr3 |
|---|---|---|---|
| Positive | 0.0 | 1.1 | 0.1 |
| Negative | 2.0 | 1.0 | -1.0 |
| Negative | 2.0 | 1.3 | 1.0 |

- Input Dataset<Row> that must be generated as input for the MLlib classification algorithms

| label | features |
|---|---|
| 1.0 | [0.0, 1.1, 0.1] |
| 0.0 | [2.0, 1.0, -1.0] |
| 0.0 | [2.0, 1.3, 1.0] |

## Categorical class labels

- Inpu

The categorical values of categoricalLabel (the class label column) must be mapped to integer values (finally casted to doubles)

| categoricalLabel | Attr1 | Attr2 | Attr3 |
|---|---|---|---|
| Positive | 0.0 | 1.1 | 0.1 |
| Negative | 2.0 | 1.0 | -1.0 |
| Negative | 2.0 | 1.3 | 1.0 |

- Input Dataset<Row> that must be generated as input for the MLlib classification algorithms

| label | features |
|---|---|
| 1.0 | [0.0, 1.1, 0.1] |
| 0.0 | [2.0, 1.0, -1.0] |
| 0.0 | [2.0, 1.3, 1.0] |

# Categorical class labels

- The Estimators **StringIndexer** and **IndexToString** support the transformation of categorical class label into numerical one
  - StringIndexer maps each categorical value of the class label to an integer (finally casted to a double)
  - IndexToString is used to perform the opposite operation

# Categorical class labels

- Main steps
  1. Use **StringIndexer** to extend the input DataFrame with a new column, called "**label**", containing the numerical representation of the class label column
  2. Create a column, called "**features"**, of type vector containing the predictive features
  3. Infer a **classification model** by using a classification algorithm (e.g., Decision Tree, Logistic regression)
     - The model is built by considering only the values of features and label. All the other columns are not considered by the classification algorithm during the generation of the prediction model

## Categorical class labels

4. Apply the model on a set of **unlabeled data** to predict their **numerical class label**
5. Use **IndexToString** to convert the predicted numerical class label values to the **original categorical values**

# Categorical class labels: Example – Training data

- Input training file

  Positive,0.0,1.1,0.1

  Negative,2.0,1.0,-1.0

  Negative,2.0,1.3,1.0

- Initial training Dataset<Row>

| categoricalLabel | features |
|---|---|
| Positive | [0.0, 1.1, 0.1] |
| Negative | [2.0, 1.0, -1.0] |
| Negative | [2.0, 1.3, 1.0] |

# Categorical class labels: Example – Training data

- Input training file

  Positive,0.0,1.1,0.1

  Negative,2.0,1.0,-1.0

  Negative,2.0,1.3,1.0

- Initial training Dataset<Row>

| categoricalLabel | features |
|---|---|
| Positive | [0.0, 1.1, 0.1] |
| Negative | [2.0, 1.0, -1.0] |
| Negative | [2.0, 1.3, 1.0] |

Vector

String

# Categorical class labels: Example – Training data

- Initial training Dataset<Row>

| categoricalLabel | features |
|---|---|
| Positive | [0.0, 1.1, 0.1] |
| Negative | [2.0, 1.0, -1.0] |
| Negative | [2.0, 1.3, 1.0] |

- Training Dataset<Row> after StringIndexer

| categoricalLabel | features | label |
|---|---|---|
| Positive | [0.0, 1.1, 0.1] | 1.0 |
| Negative | [2.0, 1.0, -1.0] | 0.0 |
| Negative | [2.0, 1.3, 1.0] | 0.0 |

# Categorical class labels: Example – Training data

- Initial training Dataset<Row>

| categoricalLabel | features |
|---|---|
| Positive | [0.0, 1.1, 0.1] |
| Negative | [2.0, 1.0, -1.0] |
| Negative | [2.0, 1.3, 1.0] |

- Training Dataset<Row> after StringIndexer

| categoricalLabel | features | label |
|---|---|---|
| Positive | [0.0, 1.1, 0.1] | 1.0 |
| Negative | [2.0, 1.0, -1.0] | 0.0 |
| Negative | [2.0, 1.3, 1.0] | 0.0 |

Mapping generated by StringIndexer:
-"Positive": 1.0
-"Negative": 0.0

## Categorical class labels: Example – Unlabeled data

- Input unlabeled data file

```
,-1.0,1.5,1.3
,3.0,2.0,-0.1
,0.0,2.2,-1.5
```

- Initial unlabeled data Dataset<Row>

| categoricalLabel | features |
|---|---|
| Positive | [-1.0, 1.5, 1.3] |
| Positive | [3.0, 2.0, -0.1] |
| Positive | [0.0, 2.2, -1.5] |

# Categorical class labels: Example – Unlabeled data

- Input unlabeled data file

```
,-1.0,1.5,1.3
,3.0,2.0,-0.1
,0.0,2.2,-1.5
```

The categoricalLabel attribute must be set to a valid value also for the unlabeled data otherwise the StringIndexer component of the Pipeline raises and an error.
Select the class label value that you prefer.
It does not impact on the predicted class label value.

- Initial unlabeled data

| categoricalLabel | features |
|---|---|
| Positive | [-1.0, 1.5, 1.3] |
| Positive | [3.0, 2.0, -0.1] |
| Positive | [0.0, 2.2, -1.5] |

# Categorical class labels: Example – Unlabeled data

- Initial unlabeled data Dataset<Row>

| categoricalLabel | features |
|---|---|
| Positive | [-1.0, 1.5, 1.3] |
| Positive | [3.0, 2.0, -0.1] |
| Positive | [0.0, 2.2, -1.5] |

- Dataset<Row> after prediction + IndexToString

| categoricalLabel | features | label | prediction | predictedLabel | … |
|---|---|---|---|---|---|
| … | [-1.0, 1.5, 1.3] | … | 1.0 | Positive | |
| … | [3.0, 2.0, -0.1] | … | 0.0 | Negative | |
| … | [0.0, 2.2, -1.5] | … | 1.0 | Positive | |

# Categorical class labels: Example – Unlabeled data

- Initial unlabeled data Dataset<Row>

| categoricalLabel | features |
|---|---|
| Positive | [-1.0, 1.5, 1.3] |
| Positive | [illegible] |
| Positive | [illegible] |

Predicted label: numerical version

Predicted label: categorical/original version

- Dataset<Row> after prediction + IndexToString

| categoricalLabel | features | label | prediction | predictedLabel | … |
|---|---|---|---|---|---|
| … | [-1.0, 1.5, 1.3] | … | 1.0 | Positive | |
| … | [3.0, 2.0, -0.1] | … | 0.0 | Negative | |
| … | [0.0, 2.2, -1.5] | … | 1.0 | Positive | |

## Categorical class labels: Example

- In the following example, the input training data is stored in a text file that contains
  - One record/data point per line
  - The records/data points are structured data with a fixed number of attributes (four)
    - One attribute is the class label
      - Categorical attribute assuming two values: Positive, Negative
    - The other three attributes are the predictive attributes that are used to predict the value of the class label
  - The input file has not the header line

## Categorical class labels: Example

- The file containing the **unlabeled data** has the same format of the training data file
  - However, the **first column** is **empty** because the class label is **unknown**
- We want to predict the class label value of each unlabeled data by applying the classification model that has been inferred on the training data

# Categorical class labels: Example

```
package it.polito.bigdata.spark.sparkmllib;

import java.io.Serializable;
import org.apache.spark.ml.linalg.Vector;

@SuppressWarnings("serial")
public class MyLabeledPoint implements Serializable {

    private String categoricalLabel;
    private Vector features;

    public MyLabeledPoint(String categoricalLabel, Vector features) {
        this.categoricalLabel = categoricalLabel;
        this.features = features;
    }

    public String getCategoricalLabel() {
        return categoricalLabel;
    }
```

# Categorical class labels: Example

```
    public Vector getFeatures() {
        return features;
    }

    public void setFeatures(Vector features) {
        this.features = features;
    }
}
```

# Categorical class labels: Example

```
package it.polito.bigdata.spark.sparkmllib;

import org.apache.spark.api.java.*;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.apache.spark.ml.Pipeline;
import org.apache.spark.ml.PipelineModel;
import org.apache.spark.ml.PipelineStage;
import org.apache.spark.ml.classification.DecisionTreeClassifier;
import org.apache.spark.ml.linalg.Vector;
import org.apache.spark.ml.linalg.Vectors;
import org.apache.spark.ml.feature.IndexToString;
import org.apache.spark.ml.feature.StringIndexer;
import org.apache.spark.ml.feature.StringIndexerModel;
```

# Categorical class labels: Example

```
public class SparkDriver {
    public static void main(String[] args) {
        String inputFileTraining;       String inputFileTest;    String outputPath;
        inputFileTraining=args[0];
        inputFileTest=args[1];
        outputPath=args[2];

        // Create a Spark Session object and set the name of the application
        // We use some Spark SQL transformation in this program
        SparkSession ss = SparkSession.builder().
        .appName("MLlib - Decision Tree - Categorical label").getOrCreate();

        // Create a Java Spark Context from the Spark Session
        // When a Spark Session has already been defined this method
        // is used to create the Java Spark Context
        JavaSparkContext sc = new JavaSparkContext(ss.sparkContext());
```

## Categorical class labels: Example

```
// ************************
// Training step
// ************************

// Read training data from a textual file
// Each lines has the format: class-label,list of numerical attribute values
// The class label is a String
// E.g., Positive,1.0,5.0,4.5,1.2
JavaRDD<String> trainingData=sc.textFile(inputFileTraining);
```

## Categorical class labels: Example

```
// Map each input record/data point of the input file to a MyLabeledPoint object
// MyLabeledPoint is a class that I defined to store the information about
// records/data points  with class label of type String
        JavaRDD<MyLabeledPoint> trainingRDD=trainingData.map(record ->
                {
                        String[] fields = record.split(",");
                        // Fields of 0 contains the id of the class
                        String classLabel = fields[0];

                        // The other three cells of fields contain the values of the
                        // three predictive attributes
                        // Create an array of doubles containing those values
                        double[] attributesValues = new double[3];

                        attributesValues[0] = Double.parseDouble(fields[1]);
                        attributesValues[1] = Double.parseDouble(fields[2]);
                        attributesValues[2] = Double.parseDouble(fields[3]);
```

## Categorical class labels: Example

```
                // Create a dense vector based on the content of
                //  attributesValues
                Vector attrValues= Vectors.dense(attributesValues);

                // Return a LabeledPoint based on the content of the
                // current line
                return new MyLabeledPoint(classLabel, attrValues);
});
```

## Categorical class labels: Example

```
// Prepare training data.
// We use MyLabeledPoint, which is a JavaBean.
// We use Spark SQL to convert RDDs of JavaBeans
// into DataFrames.
// Each data point has a set of features and a label
Dataset<Row> training =
        ss.createDataFrame(trainingRDD, MyLabeledPoint.class).cache();
```

## Categorical class labels: Example

```
// The StringIndexer Estimator is used to map each class label
// value to an integer value (casted to a double).
// A new attribute called label is generated by applying
// transforming the content of the categoricalLabel attribute.
StringIndexerModel labelIndexer = new StringIndexer()
                .setInputCol("categoricalLabel")
                .setOutputCol("label")
                .fit(training);
```

## Categorical class labels: Example

```
//The StringIndexer Estimator is used to map each class label
// value to an integer value (casted to a double).
// A new attribute called label is generated by applying
// transforming the content of the categoricalLabel attribute.
StringIndexerModel labelIndexer = new StringIndexer()
                .setInputCol("categoricalLabel")
                .setOutputCol("label")
                .fit(training);
```

This StringIndexer component is used to map the categorical values of column "categoricalLabel" to a set of integer values stored in the new column called "label"

# Categorical class labels: Example

```
// Create a DecisionTreeClassifier object.
// DecisionTreeClassifier is an Estimator that is used to create a
// classification model based on decision trees
// By default, the MLlib decision tree algorithm considers only the content
// of the features attribute to predict the value of the label attribute.
// The other attributes are not considered during the generation of the
//  model (i.e., in this case the value of categoricalLabel is not considered by
//  the MLlib decision tree algorithm).
DecisionTreeClassifier dc= new DecisionTreeClassifier();

// We can set the values of the parameters of the
// DecisionTree.
// For example we can set the measure that is used to decide if a
// node must be split.
// In this case we set it to the gini index
dc.setImpurity("gini");
```

## Categorical class labels: Example

```
// At the end of the pipeline we must convert indexed labels back
// to original labels (from numerical to string).
// The content of the prediction attribute is the index of the predicted class
// The original name of the predicted class is stored in  the predictedLabel
// attribute.
// IndexToString creates a new column (called predictedLabel in
// this example) that is based on the content of the prediction column.
// prediction is a double while predictedLabel is a string
IndexToString labelConverter = new IndexToString()
  .setInputCol("prediction")
  .setOutputCol("predictedLabel")
  .setLabels(labelIndexer.labels());
```

# Categorical class labels: Example

```
// At the end of the pipeline we must convert indexed labels back
// to original labels (from numerical to string).
// The content of the prediction attribute is the index of the predicted class
// The original name of the predicted class is stored in  the predictedLabel
// attribute.
// IndexToString creates a new column (called predictedLabel in
// this example) that is based on the content of the prediction column.
// prediction is a double while predictedLabel is a string
IndexToString labelConverter = new IndexToString()
  .setInputCol("prediction")
  .setOutputCol("predictedLabel")
  .setLabels(labelIndexer.labels());
```

This IndexToString  component is used to remap the numerical predictions available in the "prediction" column to the original categorical values that are stored in the new column called "predictedLabel"

# Categorical class labels: Example

```
// Define the pipeline that is used to create the decision tree
// model on the training data.
// In this case the pipeline contains the following steps:
// - Create a new column, called label, containing the numerical
//    representation of the original column
// - Run the decision tree algorithm to infer a classification model
// - Convert the numerical predicted class label values into the original
//    categorical values (strings)
Pipeline pipeline = new Pipeline()
                .setStages(new PipelineStage[] {labelIndexer,dc,labelConverter});

// Execute the pipeline on the training data to build the
// classification model
PipelineModel model = pipeline.fit(training);

// Now, the classification model can be used to predict the class label
// of new unlabeled data
```

# Categorical class labels: Example

```
// Define the pipeline that is used to create the decision tree
// model on the training data.
// In this case the pipeline contains the following steps:
// - Create a new column, called label, containing the numerical
//   representation of th
// - Run the decision tree algorithm to infer a classification model
// - Convert the numerical predicted class label values into the original
//   categorical values (strings)
Pipeline pipeline = new Pipeline()
            .setStages(new PipelineStage[] {labelIndexer,dc,labelConverter});

// Execute the pipeline on the training data to build the
// classification model
PipelineModel model = pipeline.fit(training);

// Now, the classification model can be used to predict the class label
// of new unlabeled data
```

This Pipeline is composed of three steps

## Categorical class labels: Example

```
// ************************
// Prediction  step
// ************************

// Read unlabeled data
// For the unlabeled data only the predictive attributes are available
// The class label is not available and must be predicted by applying
// the classification model inferred during the previous phase
JavaRDD<String> unlabeledData=sc.textFile(inputFileTest);
```

## Categorical class labels: Example

```
// Map each unlabeled input record/data point of the input file to
// a MyLabeledPoint
JavaRDD<MyLabeledPoint> unlabeledRDD=unlabeledData.map(record ->
{
        String[] fields = record.split(",");

        // The last three cells of fields contain the (numerical) values of the
        // three predictive attributes
        // Create an array of doubles containing those three values
        double[] attributesValues = new double[3];

        attributesValues[0] = Double.parseDouble(fields[1]);
        attributesValues[1] = Double.parseDouble(fields[2]);
        attributesValues[2] = Double.parseDouble(fields[3]);

        // Create a dense vector based in the content of attributesValues
        Vector attrValues= Vectors.dense(attributesValues);
```

# Categorical class labels: Example

```
        // The class label in unknown.
        // To create a MyLabeledPoint a categoricalLabel value must be
        // specified also for the unlabeled data.
        // I set it to "Positive" (we must set the value of the class label to
        // a valid value otherwise IndexToString raises an error).
        // The specified value  does not impact on the prediction because
        // the categoricalLabel column is not  used to perform the
        // prediction.
        String classLabel = new String("Positive");

        // Return a new LabeledPoint
        return new MyLabeledPoint(classLabel, attrValues);
});

// Create the DataFrame based on the new test data
Dataset<Row> unlabeled =
        ss.createDataFrame(unlabeledRDD, MyLabeledPoint.class);
```

# Categorical class labels: Example

```
// Make predictions on test documents using the transform()
// method.
// The transform will only use the 'features' columns
Dataset<Row> predictions = model.transform(unlabeled);

// The returned DataFrame has the following schema (attributes)
// - features: vector (values of the attributes)
// - label: double (value of the class label)
// - rawPrediction: vector (nullable = true)
// - probability: vector (The i-th cell contains the probability that the
//                   current record belongs to the i-th class
// - prediction: double (the predicted class label)
// - predictedLabel: string (nullable = true)

// Select only the features (i.e., the value of the attributes) and
// the predicted class for each record (the categorical version)
Dataset<Row> predictionsDF=
                            predictions.select("features", "predictedLabel");
```

# Categorical class labels: Example

```
// Make predictions on test documents using the transform()
// method.
// The transform will only use the 'features' columns
Dataset<Row> predictions = model.transform(unlabeled);

// The returned DataFrame has the following schema (attributes)
// - features: vector (values of the attributes)
// - label: double (value of the class label)
// - rawPrediction: vector (nullable = true)
// - probability: vect                                              the
//                   curren
// - prediction: doub
// - predictedLabel:

// Select only the features (i.e., the value of the attributes) and
// the predicted class for each record (the categorical version)
Dataset<Row> predictionsDF=
                    predictions.select("features", "predictedLabel");
```

"predictedLabel" is the column containing the predicted categorical class label for the unlabeled data

## Categorical class labels: Example

```
            // Save the result in an HDFS file
            JavaRDD<Row> predictionsRDD = predictionsDF.javaRDD();
            predictionsRDD.saveAsTextFile(outputPath);


            // Close the Spark Context object
            sc.close();
      }
}
```

## Sparse labeled data

## Sparse labeled data: The LIBSVM format

- Frequently the training data are sparse
  - E.g., textual data are sparse
    - Each document contains only a subset of the possible words
  - Hence, sparse vectors are frequently used
- MLlib supports reading training examples stored in the LIBSVM format
  - It is a commonly used textual format that is used to represent sparse documents/data points

## Sparse labeled data: The LIBSVM format

- The LIBSVM format
  - It is a textual format in which each line represents a labeled point by using a sparse feature vector:
- Each line has the format

  label index1:value1 index2:value2 ...
- where
  - label is an integer associated with the class label
    - It is the first value of each line
  - The indexes are integer values representing the features
  - The values are the (double) values of the features

## Sparse labeled data: The LIBSVM format

- Consider the following two records/data points characterized by 4 predictive features and a class label
  - Features = [5.8, 1.7, 0 , 0 ] -- Label = 1
  - Features = [4.1, 0 , 2.5, 1.2] -- Label = 0
- Their LIBSVM format-based representation is the following

```
1 1:5.8 2:1.7

0 1:4.1 3:2.5 4:1.2
```

## Sparse labeled data: The LIBSVM format

- LIBSVM files can be loaded into Dataset<Row> by combining the following methods:
  - read(), format("libsvm"), and load(String inputpath)
- The returned Dataset<Row> (i.e., DataFrame) has two columns:
  - label: double
    - The double value associated with the label
  - features: vector
    - A sparse vector associated with the predictive features

## Sparse labeled data: The LIBSVM format

```
…
SparkSession ss = parkSession.builder()
        .appName("Test read LIBSMV file").getOrCreate();

Dataset<Row> data = ss.read().format("libsvm")
        .load("sample_libsvm_data.txt");


..
```