

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:
 - `pyspark.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
 - `pyspark.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

- MLlib APIs are divided into two packages:
 - `pyspark.mllib`
 - It contains RDDs
 - This version of code and will be probably deprecated in the next releases of Spark
 - `pyspark.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

We will use the DataFrame-based version

Spark MLlib – Data types

Spark MLlib – Data types

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

Local vectors

- Local `pyspark.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 before applying MLlib algorithms

Local vectors

- Dense and sparse representations are supported
- E.g., the vector of doubles $[1.0, 0.0, 3.0]$ 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 dense and sparse vectors can be created in Spark

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

```
# Create a dense vector [1.0, 0.0, 3.0]  
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  
# by means of a dictionary  
sv = Vectors.sparse(3, { 0:1.0, 2:3.0 })
```

Local vectors

- The following code shows how dense and sparse vectors can be created in Spark

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

```
# Create a dense vector [1.0, 0.0, 3.0]  
dv = Vectors.dense([1.0, 0.0, 3.0])
```

```
# Create a sparse vector [1.0, 0.0, 3.0] by specifying  
# its indices and value  
# by means of a dictionary  
sv = Vectors.sparse(3, {0:1.0, 2:3.0})
```

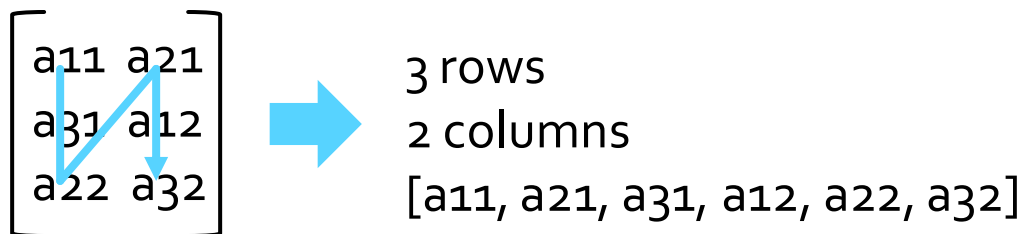
Index and value of a non-empty cell

Size of the vector

Dictionary of index:value pairs

Local matrices

- Local `pyspark.ml.linalg.Matrix` objects in MLlib are used to store matrices of double values
 - Dense and sparse matrices are supported
 - The column-major order is used to store the content of the matrix in a linear way



Local matrices

- The following code shows how dense and sparse matrices can be created in Spark

```
from pyspark.ml.linalg import Matrices
```

```
# Create a dense matrix with two rows and three columns
```

```
# 3.0 0.0 0.0
```

```
# 1.0 1.5 2.0
```

```
dm = Matrices.dense(2,3,[3.0, 1.0, 0.0, 1.5, 0.0, 2.0])
```

```
# Create a sparse version of the same matrix
```

```
sm = Matrices.sparse(2,3, [0, 2, 3, 4], [0, 1, 1, 1] , [3,1,1.5,2])
```

Local matrices

- The following code shows how dense and sparse matrices can be created in Spark

```
from pyspark.ml.linalg import Matrices
```

```
# Create a dense matrix with two rows and three columns
```

```
# 3.0 0.0 0.0
```

```
# 1.0 1.5 2.0
```

```
dm = Matrices.dense(2, 3, [3.0, 1.0, 0.0, 1.5, 0.0, 2.0])
```

Number of columns

Number of rows

Values in column-major order

```
sm = Matrices.sparse(2, 3, [0, 2, 3, 4], [0, 1, 1, 1], [3, 1, 1.5, 2])
```

Local matrices

- The following code shows how dense and sparse matrices can be created in Spark

from pyspark.m

Create a dens

3.0 0.0 0.0

1.0 1.5 2.0

dm = Matrices.dense(2,3,[3.0, 1.0, 0.0, 1.5

Create a sparse version of the same matrix

sm = Matrices.sparse(2,3,[0, 2, 3, 4],[0, 1, 1, 1],[3,1,1.5,2])

One element per column that encodes the offset in the array of non-zero values where the values of the given column start.
The last element is the number of non-zero values.

Number of columns

Array of non-zero values of the represented matrix

Number of rows

Row index of each non-zero value

Spark MLlib - Main concepts

Spark MLlib - Main concepts

- Spark MLlib uses DataFrames as input data
- 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 document collections must be transformed in a tabular format before applying the MLlib algorithms

Spark MLlib - Main concepts

- The **DataFrames** used and created by the MLlib algorithms are characterized by several columns
- Each column is associated with a different role/meaning
 - **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
 - **prediction**
 - Predicted value of a classification/regression analysis
 - ..

Spark MLlib - Main concepts

- **Transformer**

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

Spark MLlib - Main concepts

■ Estimator

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

- **Parameters**
 - Transformers and Estimators share common APIs for specifying the values of their 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 trained
 4. (optional) The model is applied on new data

Data Preprocessing

Data preprocessing

- Input data must be preprocessed before applying machine learning and data mining algorithms
 - To organize data in a format consistent with the one expected by the applied algorithms
 - To define good (predictive) features
 - To remove bias
 - E.g., normalization
 - To remove noise and missing values
 - ...

Extracting, transforming and selecting features

- MLlib provides a set of transformers than can be used to extract, transform and select features from DataFrames
 - Feature Extractors
 - TF-IDF, Word2Vec, ..
 - Feature Transformers
 - Tokenizer, StopWordsRemover, StringIndexer, IndexToString, OneHotEncoderEstimator, Normalizer, ...
 - Feature Selectors
 - VectorSlicer, ...
- Up-to-date list
 - <https://spark.apache.org/docs/latest/ml-features.html>

Feature Transformations

Feature Transformations

- Several algorithms are provided by MLlib to transform features
 - They are used to create new columns/features by combining or transforming other features
 - You can perform feature transformations and feature creations by using the standard methods you already know for DataFrames and RDDs

Vector Assembler

- VectorAssembler
(`pyspark.ml.feature.VectorAssembler`) is a transformer that combines a given list of columns into a single vector column
 - Useful for combining features into a single feature vector before applying ML algorithms

Vector Assembler

- **VectorAssembler**(inputCols, outputCol)
 - **inputCols**
 - The list of original columns to include in the new column of type Vector
 - The following input column types are accepted
 - all numeric types, boolean type, and vector type
 - Boolean values are mapped to 1 (True) and 0 (False)
 - **outputCol**
 - The name of the new output column

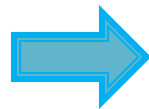
Vector Assembler

- When the transform method of VectorAssembler is invoked on a DataFrame the returned DataFrame
 - Has a new column (outputCol)
 - For each record, the value of the new column is the “concatenation” of the values of the input columns
 - Has also all the columns of the input DataFrame

Vector Assembler: Example

- Consider an input DataFrame with three columns
- Create a new DataFrame with a new column containing the “concatenation” of colB and colC in a new vector column
 - The name of the new column is set to features

colA	colB	colC
1	4.5	True
2	0.6	True
3	1.5	False
4	12.1	True
5	0.0	True



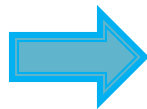
colA	colB	colC	features
1	4.5	True	[4.5,1.0]
2	0.6	True	[0.6,1.0]
3	1.5	False	[1.5,0.0]
4	12.1	True	[12.1,1.0]
5	0.0	True	[0.0,1.0]

Vector Assembler: Example

- Consider an input DataFrame with three columns
- Create a new DataFrame with a new column containing the “concatenation” of colB and colC in a new vector column
 - The name of the new column is set to features

Columns of DataFrames can also be vectors

colA	colB	colC
1	4.5	True
2	0.6	True
3	1.5	False
4	12.1	True
5	0.0	True



colA	colB	colC	features
1	4.5	True	[4.5,1.0]
2	0.6	True	[0.6,1.0]
3	1.5	False	[1.5,0.0]
4	12.1	True	[12.1,1.0]
5	0.0	True	[0.0,1.0]

Vector Assembler: Example

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

# input and output folders
inputPath = "data/exampleDataAssembler.csv"
# Create a DataFrame from the input data
inputDF = spark.read.load(inputPath,\
                           format="csv", header=True, inferSchema=True)

# Create a VectorAssembler that combines columns colB and colC
# The new vector column is called features
myVectorAssembler = VectorAssembler(inputCols = ['colB', 'colC'],\
                                     outputCol = 'features')

# Apply myVectorAssembler on the input DataFrame
transformedDF = myVectorAssembler.transform(inputDF)
```

Data Normalization

- MLlib provides a set of normalization algorithms (called scalers)
 - StandardScaler
 - MinMaxScaler
 - Normalizer
 - MaxAbsScaler

Standard Scaler

- **StandardScaler** (`pyspark.ml.feature.StandardScaler`) is an Estimator that **returns** a **Transformer** (`pyspark.ml.feature.StandardScalerModel`)
- `StandardScalerModel` transforms a vector column of an input `DataFrame` normalizing each “feature” of the input vector column to have unit standard deviation and/or zero mean

Standard Scaler

- `StandardScaler(inputCol, outputCol)`
 - `inputCol`
 - The name of the input vector column (of doubles) to normalize
 - `outputCol`
 - The name of the new output normalized vector column
- Invoke the `fit` method of `StandardScaler` on the `input DataFrame` to infer a `StandardScalerModel`
 - The returned model is a `Transformer`

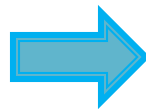
Standard Scaler

- Invoke the transform method of StandardScalerModel on the input DataFrame to create a new DataFrame that
 - Has a new column (outputCol)
 - For each record, the value of the new column is the normalized version of the input vector column
 - Has also all the columns of the input DataFrame

Standard Scaler: Example

- Consider an input DataFrame with four columns
- Create a new DataFrame with a new column containing the normalized version of the vector column features
 - Set the name of the new column to scaledFeatures

colA	colB	colC	features
1	4.5	True	[4.5,1.0]
2	0.6	True	[0.6,1.0]
3	1.5	False	[1.5,0.0]
4	12.1	True	[12.1,1.0]
5	0.0	True	[0.0,1.0]



colA	colB	colC	features	scaledFeatures
1	4.5	True	[4.5,1.0]	[0.903,2.236]
2	0.6	True	[0.6,1.0]	[0.120,2.236]
3	1.5	False	[1.5,0.0]	[0.301, 0.0]
4	12.1	True	[12.1,1.0]	[2.428,2.236]
5	0.0	True	[0.0,1.0]	[0.0 ,2.236]

Standard Scaler: Example

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler

# input and output folders
inputPath = "data/exampleDataAssembler.csv"
# Create a DataFrame from the input data
inputDF = spark.read.load(inputPath,\
                           format="csv", header=True, inferSchema=True)
# Create a VectorAssembler that combines columns colB and colC
# The new vector column is called features
myVectorAssembler = VectorAssembler(inputCols = ['colB', 'colC'],\
                                     outputCol = 'features')

# Apply myVectorAssembler on the input DataFrame
transformedDF = myVectorAssembler.transform(inputDF)
```

Standard Scaler: Example

Create a Standard Scaler to scale the content of features

```
myScaler = StandardScaler(inputCol="features", outputCol="scaledFeatures")
```

Compute summary statistics by fitting the StandardScaler

Before normalizing the content of the data we need to compute mean and

standard deviation of the analyzed data

```
scalerModel = myScaler.fit(transformedDF)
```

Apply myScaler on the input column features

```
scaledDF = scalerModel.transform(transformedDF)
```


Categorical columns

- Frequently the input data are characterized by categorical attributes (i.e., string columns)
 - The class label of the classification problem is a categorical attribute
- The Spark MLlib classification and regression algorithms work only with numerical values
- Categorical columns must be mapped to double values

StringIndexer

- StringIndexer (pyspark.ml.feature.StringIndexer) is an **Estimator** that returns a **Transformer of type** `pyspark.ml.feature.StringIndexerModel`
- StringIndexerModel encodes a string column of “labels” to a column of “label indices”
 - Each distinct value of the input string column is mapped to an integer value in $[0, \text{num. distinct values})$

StringIndexer

- `StringIndexer(inputCol, outputCol)`
 - `inputCol`
 - The name of the input string column to map to a set of integers
 - `outputCol`
 - The name of the new output column
- Invoke the `fit` method of `StringIndexer` on the input `DataFrame` to infer a `StringIndexerModel`
 - The returned model is a `Transformer`

StringIndexer

- Invoke the **transform** method of StringIndexerModel on the input DataFrame to create a new DataFrame that
 - Has a new column (outputCol)
 - For each record, the value of the new column is the integer (casted to a double) associated with the value of the input string column
 - Has also all the columns of the input DataFrame

StringIndexer : Example

- Consider an input DataFrame with two columns
- Create a new DataFrame with a new column containing the “integer” version of the string column category
 - Set the name of the new column to categoryIndex

id	category
1	a
2	b
3	c
4	c
5	a



id	category	categoryIndex
1	a	0.0
2	b	2.0
3	c	1.0
4	c	1.0
5	a	0.0

StringIndexer : Example

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import StringIndexer

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

# Create a StringIndexer to map the content of category to a set of "integers"
indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")

# Analyze the input data to define the mapping string -> integer
indexerModel = indexer.fit(df)

# Apply indexerModel on the input column category
indexedDF = indexerModel.transform(df)
```

IndexToString

- `IndexToString`(`pyspark.ml.feature.IndexToString`), which is `symmetrical` to `StringIndexer`, is a Transformer that maps a column of “`label indices`” back to a column containing the original “`labels`” as strings
 - Classification models return the integer version of the predicted label values. We must remap those values to the original ones to obtain human readable results

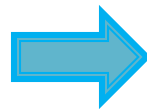
IndexToString

- `IndexToString(inputCol, outputCol, labels)`
 - `inputCol`
 - The name of the input numerical column to map to the original a set of string "labels"
 - `outputCol`
 - The name of the new output column
 - `labels`
 - The list of original "labels"/strings
 - The mapping with integer values is given by the positions of the strings inside labels
- Invoke the `transform` method of `IndexToString` on the input `DataFrame` to create a new `DataFrame` that
 - Has a new column (`outputCol`)
 - For each record, the value of the new column is the original string associated with the value of the input numerical column
 - Has also all the columns of the input `DataFrame`

IndexToString: Example

- Consider an input DataFrame with two columns
- Create a new DataFrame with a new column containing the “integer” version of the string column category and then map it back to the string version in a new column

id	category
1	a
2	b
3	c
4	c
5	a



id	category	categoryIndex	originalCategory
1	a	0.0	a
2	b	2.0	b
3	c	1.0	c
4	c	1.0	c
5	a	0.0	a

IndexToString: Example

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import IndexToString

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

# Create a StringIndexer to map the content of category to a set of "integers"
indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")

# Analyze the input data to define the mapping string -> integer
indexerModel = indexer.fit(df)

# Apply indexerModel on the input column category
indexedDF = indexerModel.transform(df)
```

IndexToString: Example

```
# Create an IndexToString to map the content of numerical attribute categoryIndex  
# to the original string value  
converter = IndexToString(inputCol="categoryIndex", outputCol="originalCategory",\  
                          labels=indexerModel.labels)  
  
# Apply converter on the input column categoryIndex  
reconvertedDF = converter.transform(indexedDF)
```

SQLTransformer

- SQLTransformer (pyspark.ml.feature.SQLTransformer) is a **transformer** that implements the transformations which are defined by SQL queries
 - Currently, the **syntax of the supported** (simplified) SQL queries is
 - “**SELECT** attributes, function(attributes)
FROM __THIS__”
 - __THIS__ represents the DataFrame on which the SQLTransformer is invoked
- SQLTransformer **executes an SQL query** on the input DataFrame and returns a new DataFrame associated with the result of the query

SQLTransformer

- SQLTransformer(statement)
 - statement
 - The SQL query to execute

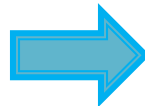
SQLTransformer

- When the transform method of SQLTransformer is invoked on a DataFrame the returned DataFrame is the result of the executed statement query

SQLTransformer : Example

- Consider an input DataFrame with two columns: "text" and "id"
- Create a new DataFrame with a new column, called "numWords", containing the number of words occurring in column "text"

id	text
1	This is Spark
2	Spark
3	Another sample sentence of words
4	Paolo Rossi
5	Giovanni



id	text	numWords
1	This is Spark	3
2	Spark	1
3	Another sample sentence of words	5
4	Paolo Rossi	2
5	Giovanni	1

SQLTransformer : Example

```
from pyspark.sql.types import *
from pyspark.ml.feature import SQLTransformer

#Local Input data
inputList = [(1, "This is Spark"),\
              (2, "Spark"),\
              (3, "Another sample sentence of words"),\
              (4, "Paolo Rossi"),\
              (5, "Giovanni")]

# Create the initial DataFrame
dfInput = spark.createDataFrame(inputList, ["id", "text"])
```


SQLTransformer : Example

```
# Define a UDF function that counts the number of words in an input string
spark.udf.register("countWords", lambda text: len(text.split(" ")), IntegerType())
```

```
# Define an SQLTransformer to create the columns we are interested in
sqlTrans = SQLTransformer(statement="""SELECT *,
countWords(text) AS numLines
FROM __THIS__""")
```

```
# Create the new DataFrame by invoking the transform method of the
# defined SQLTransformer
newDF = sqlTrans.transform(dfInput)
```

Classification algorithms

Classification algorithms

- Spark MLlib provides a (limited) set of classification algorithms
 - Logistic regression
 - Binomial logistic regression
 - Multinomial logistic regression
 - Decision tree classifier
 - Random forest classifier
 - Gradient-boosted tree classifier
 - Multilayer perceptron classifier
 - Linear Support Vector Machine

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 (one distinct value per class) before applying the MLlib classification algorithms

Classification algorithms

- All the Spark classification algorithms are trained on top of an input DataFrame 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 `pyspark.ml.linalg.Vector`
 - Both dense and sparse vectors can be used

Classification algorithms: Example of expected input DataFrame

- 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 DataFrame

- 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/train 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 DataFrame

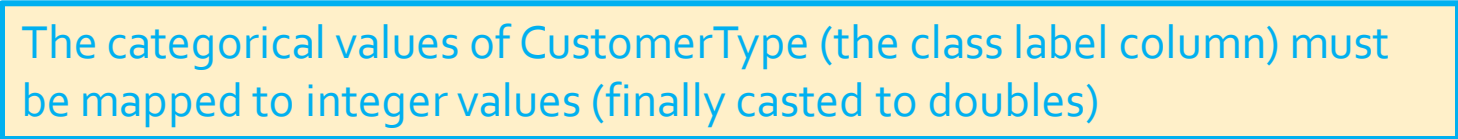
- Input training data

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

- Input training DataFrame that must be provided as input to train an MLlib classification algorithm

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 DataFrame

- Input  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 training DataFrame that must be provided as input to train an MLlib classification algorithm

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 DataFrame

- Input

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 training DataFrame that must be provided as input to train an MLlib classification algorithm

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 DataFrame

- Input In the generated DataFrame 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 training DataFrame that must be provided as input to train an MLlib classification algorithm

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

Structured data classification

Logistic regression and structured data

- The following slides show how to
 - Create a classification model based on the **logistic regression algorithm** on **structured data**
 - 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 values are already doubles (we do not need to convert them)
 - The input file has the header line

Logistic regression and structured data: training data

- Consider the following example input training data file

label,attr1,attr2,attr3

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 DataFrame containing two columns
 - label
 - The double value that is used to specify the label of each training record
 - features
 - It is a vector of doubles associated with the values of the predictive features

Logistic regression and structured data: training data

- Input training file logistic regression (classification) is different from linear one (clustering)

label,attr1,attr2,attr3

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 DataFrame to be created

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

label,attr1,attr2,attr3

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 DataFrame to be created

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

label,attr1,attr2,attr3

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: pyspark.ml.linalg.Vector

- Input training DataFrame to be created

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 trained on the training data
- The predicted class label value of the unlabeled data is stored in a new column, called “prediction”, of the returned DataFrame

Logistic regression and structured data: unlabeled data

- Consider the following example input unlabeled data file

label,attr1,attr2,attr3

,-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 DataFrame containing two columns
 - label
 - features
- A label value is required also for unlabeled data
 - Its value is set to null for all records

Logistic regression and structured data: unlabeled data

- Input unlabeled data file

label,attr1,attr2,attr3

, -1.0, 1.5, 1.3

, 3.0, 2.0, -0.1

, 0.0, 2.2, -1.5

- Input unlabeled data DataFrame to be created

label	features
null	[-1.0, 1.5, 1.3]
null	[3.0, 2.0, -0.1]
null	[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 DataFrame containing
 - The same columns of the input DataFrame
 - A new column called prediction
 - For each input unlabeled record, it contains the predicted class label value
 - Also two other 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 DataFrame

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

- Returned DataFrame with the predicted class label values

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

Logistic regression and structured data: prediction column

- Input unlabeled data DataFrame

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

This column contains the predicted class label values

- Returned DataFrame with the predicted class label values

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

Logistic regression and structured data: Example

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
```

```
# input and output folders
trainingData = "ex_data/trainingData.csv"
unlabeledData = "ex_data/unlabeledData.csv"
outputPath = "predictionsLR/"
```

Logistic regression and structured data: Example

```
# *****  
# Training step  
# *****  
  
# Create a DataFrame from trainingData.csv  
# Training data in raw format  
trainingData = spark.read.load(trainingData,\n    format="csv",\  
    header=True,\  
    inferSchema=True)
```

Logistic regression and structured data: Example

```
# Define an assembler to create a column (features) of type Vector
# containing the double values associated with columns attr1, attr2, attr3
assembler = VectorAssembler(inputCols=["attr1", "attr2", "attr3"],\
                             outputCol="features")
# Apply the assembler to create column features for the training data
trainingDataDF = assembler.transform(trainingData)
```

we should use pipeline to avoid applying this two times separately for training and testing datasets

in real application scenario, we use two API, one for training and save it to use several times, another just for prediction

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.  
lr = 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 0.01  
lr.setMaxIter(10)  
lr.setRegParam(0.01)
```

```
# Train a logistic regression model on the training data  
classificationModel = lr.fit(trainingDataDF)
```

Logistic regression and structured data: Example

```
# *****  
# Prediction step  
# *****  
# Create a DataFrame from unlabeledData.csv  
# Unlabeled data in raw format  
unlabeledData = spark.read.load(unlabeledData,\n                                format="csv", header=True, inferSchema=True)  
  
# Apply the same assembler we created before also on the unlabeled data  
# to create the features column  
unlabeledDataDF = assembler.transform(unlabeledData)  
  
# Make predictions on the unlabeled data using the transform() method of the  
# trained classification model transform uses only the content of 'features'  
# to perform the predictions  
predictionsDF = classificationModel.transform(unlabeledDataDF)
```

Logistic regression and structured data: Example

```
# The returned DataFrame has the following schema (attributes)
# - attr1
# - attr2
# - attr3
# - 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 original features (i.e., the value of the original attributes
# attr1, attr2, attr3) and the predicted class for each record
predictions = predictionsDF.select("attr1", "attr2", "attr3", "prediction")

# Save the result in an HDFS output folder
predictions.write.csv(outputPath, header="true")
```


Pipelines

- In the previous solution we applied the same preprocessing steps on both training and unlabeled data
 - We applied the same assembler on both input data
- We can use a pipeline to specify the common phases we apply on both input data sets

Logistic regression and structured data: Example based on pipelines

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel
```

```
# input and output folders
trainingData = "ex_data/trainingData.csv"
unlabeledData = "ex_data/unlabeledData.csv"
outputPath = "predictionsLR/"
```

Logistic regression and structured data: Example based on pipelines

```
# *****  
# Training step  
# *****  
  
# Create a DataFrame from trainingData.csv  
# Training data in raw format  
trainingData = spark.read.load(trainingData,\n    format="csv",\  
    header=True,\  
    inferSchema=True)
```

Logistic regression and structured data: Example based on pipelines

```
# Define an assembler to create a column (features) of type Vector
# containing the double values associated with columns attr1, attr2, attr3
assembler = VectorAssembler(inputCols=["attr1", "attr2", "attr3"],\
                             outputCol="features")
```

```
# Create a LogisticRegression object
# LogisticRegression is an Estimator that is used to
# create a classification model based on logistic regression.
lr = 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 0.01
lr.setMaxIter(10)
lr.setRegParam(0.01)
```

Logistic regression and structured data: Example based on pipelines

```
# Define a pipeline that is used to create the logistic regression  
# model on the training data. The pipeline includes also  
# the preprocessing step  
pipeline = Pipeline().setStages([assembler, lr])
```

```
# Execute the pipeline on the training data to build the  
# classification model  
classificationModel = pipeline.fit(trainingData)
```

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

Logistic regression and structured data: Example based on pipelines

```
# Define a pipeline that is used to create the logistic regression  
# model on the training data. The pipeline includes also  
# the preprocessing step  
pipeline = Pipeline().setStages([assembler, lr])
```

This is the sequence of Transformers and Estimators to apply on the input data.

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

Logistic regression and structured data: Example based on pipelines

```
# *****  
# Prediction step  
# *****  
# Create a DataFrame from unlabeledData.csv  
# Unlabeled data in raw format  
unlabeledData = spark.read.load(unlabeledData,\n                                format="csv", header=True, inferSchema=True)  
  
# Make predictions on the unlabeled data using the transform() method of the  
# trained classification model transform uses only the content of 'features'  
# to perform the predictions. The model is associated with the pipeline and hence  
# also the assembler is executed  
predictions = classificationModel.transform(unlabeledData)
```

Logistic regression and structured data: Example based on pipelines

```
# The returned DataFrame has the following schema (attributes)
# - attr1
# - attr2
# - attr3
# - 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 original features (i.e., the value of the original attributes
# attr1, attr2, attr3) and the predicted class for each record
predictions = predictionsDF.select("attr1", "attr2", "attr3", "prediction")

# Save the result in an HDFS output folder
predictions.write.csv(outputPath, header="true")
```


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** on **structured data**
 - 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
- The main steps are the same of the previous example
 - The only difference is the definition and configuration of the used classification algorithm

Decision trees and structured data: Example based on pipelines

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel

# input and output folders
trainingData = "ex_data/trainingData.csv"
unlabeledData = "ex_data/unlabeledData.csv"
outputPath = "predictionsLR/"
```

Decision trees and structured data: Example based on pipelines

```
# *****  
# Training step  
# *****  
  
# Create a DataFrame from trainingData.csv  
# Training data in raw format  
trainingData = spark.read.load(trainingData,\n    format="csv",\  
    header=True,\  
    inferSchema=True)
```

Decision trees and structured data:

Example based on pipelines

```
# Define an assembler to create a column (features) of type Vector
# containing the double values associated with columns attr1, attr2, attr3
assembler = VectorAssembler(inputCols=["attr1", "attr2", "attr3"],\
                             outputCol="features")
```

```
# Create a DecisionTreeClassifier object.
# DecisionTreeClassifier is an Estimator that is used to
# create a classification model based on decision trees.
dt = 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 gini index
dt.setImpurity("gini")
```

Decision trees and structured data: Example based on pipelines

```
# Define a pipeline that is used to create the decision tree  
# model on the training data. The pipeline includes also  
# the preprocessing step  
pipeline = Pipeline().setStages([assembler, dt])
```

```
# Execute the pipeline on the training data to build the  
# classification model  
classificationModel = pipeline.fit(trainingData)
```

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

Decision trees and structured data: Example based on pipelines

```
# Define a pipeline that is used to create the decision tree  
# model on the training data. The pipeline includes also  
# the preprocessing step  
pipeline = Pipeline().setStages([assembler, dt])
```

This is the sequence of Transformers and Estimators to apply on the input data.
A decision tree algorithm is used in this case

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


Decision trees and structured data: Example based on pipelines

```
# *****
```

```
# Prediction step
```

```
# *****
```

```
# Create a DataFrame from unlabeledData.csv
```

```
# Unlabeled data in raw format
```

```
unlabeledData = spark.read.load(unlabeledData,\n                                format="csv", header=True, inferSchema=True)
```

```
# Make predictions on the unlabeled data using the transform() method of the
```

```
# trained classification model transform uses only the content of 'features'
```

```
# to perform the predictions. The model is associated with the pipeline and hence
```

```
# also the assembler is executed
```

```
predictions = classificationModel.transform(unlabeledData)
```

Decision trees and structured data: Example based on pipelines

```
# The returned DataFrame has the following schema (attributes)
# - attr1
# - attr2
# - attr3
# - 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 original features (i.e., the value of the original attributes
# attr1, attr2, attr3) and the predicted class for each record
predictions = predictionsDF.select("attr1", "attr2", "attr3", "prediction")

# Save the result in an HDFS output folder
predictions.write.csv(outputPath, header="true")
```

Categorical class labels

Categorical class labels

- Usually 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
 - **Processing and postprocessing** steps are used to manage this transformation

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 DataFrame 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

- Input 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 DataFrame 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 Estimator **StringIndexer** and the Transformer **IndexToString** support the transformation of categorical class label into numerical one and vice versa
 - 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
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
- Initial training DataFrame

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

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

- Initial training DataFrame

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

String

Vector

Categorical class labels: Example – Training data

- Initial training DataFrame

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

- Training DataFrame 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 DataFrame

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

- Training DataFrame 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
categoricalLabel,attr1,attr2,attr3
,-1.0,1.5,1.3
,3.0,2.0,-0.1
,0.0,2.2,-1.5
- Initial unlabeled data DataFrame

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

Categorical class labels: Example – Unlabeled data

- Initial unlabeled data DataFrame

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

- DataFrame 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 DataFrame

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

Predicted label:
numerical version

Predicted label:
categorical/original version

- DataFrame 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 (categoricalLabel)
 - Categorical attribute assuming two values: Positive, Negative
 - The other three attributes (attr1, attr2, attr3) are the predictive attributes that are used to predict the value of the class label
 - The input file has 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

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import IndexToString
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel

# input and output folders
trainingData = "ex_dataCategorical/trainingData.csv"
unlabeledData = "ex_dataCategorical/unlabeledData.csv"
outputPath = "predictionsDTCategoricalPipeline/"
```

Categorical class labels: Example

```
# *****  
# Training step  
# *****  
  
# Create a DataFrame from trainingData.csv  
# Training data in raw format  
trainingData = spark.read.load(trainingData,\  
                                format="csv", header=True, inferSchema=True)  
  
# Define an assembler to create a column (features) of type Vector  
# containing the double values associated with columns attr1, attr2, attr3  
assembler = VectorAssembler(inputCols=["attr1", "attr2", "attr3"],\  
                             outputCol="features")
```

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.  
labelIndexer = StringIndexer(inputCol="categoricalLabel", outputCol="label",\  
                             handleInvalid="keep").fit(trainingData)
```

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.

```
labelIndexer = StringIndexer(inputCol="categoricalLabel", outputCol="label",\n                             handleInvalid="keep").fit(trainingData)
```

This StringIndexer estimator is used to infer a transformer that maps the categorical values of column "categoricalLabel" to a set of integer values stored in the new column called "label".
The list of valid label values are extracted from the training data

Categorical class labels: Example

```
# Create a DecisionTreeClassifier object.  
# DecisionTreeClassifier is an Estimator that is used to  
# create a classification model based on decision trees.  
dt = 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 gini index  
dt.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
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel",\
                               labels=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  
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel",\n                               labels=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”.
The mapping integer -> original string value is the one of `labelIndexer`

Categorical class labels: Example

```
# Define a pipeline that is used to create the decision tree
# model on the training data. The pipeline includes also
# the preprocessing and postprocessing steps
pipeline = Pipeline().setStages([assembler, labelIndexer, dt, labelConverter])
```

```
# Execute the pipeline on the training data to build the
# classification model
classificationModel = pipeline.fit(trainingData)
```

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

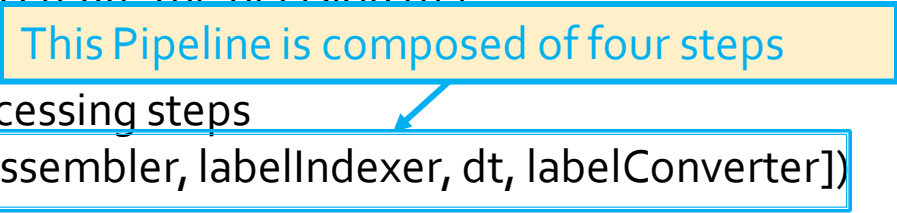
Categorical class labels: Example

```
# Define a pipeline that is used to create the decision tree
# model on the training data. The
# the preprocessing and postprocessing steps
pipeline = Pipeline().setStages([assembler, labelIndexer, dt, labelConverter])

# Execute the pipeline on the training data to build the
# classification model
classificationModel = pipeline.fit(trainingData)

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

This Pipeline is composed of four steps

A yellow callout box with a blue border contains the text "This Pipeline is composed of four steps". A blue arrow points from this box to the list of stages in the `setStages` method call of the `Pipeline` object in the code above.

Categorical class labels: Example

```
# *****  
# Prediction step  
# *****  
# Create a DataFrame from unlabeledData.csv  
# Unlabeled data in raw format  
unlabeledData = spark.read.load(unlabeledData,\  
                                format="csv", header=True, inferSchema=True)  
  
# Make predictions on the unlabeled data using the transform() method of the  
# trained classification model transform uses only the content of 'features'  
# to perform the predictions. The model is associated with the pipeline and hence  
# also the assembler is executed  
predictions = classificationModel.transform(unlabeledData)
```

Categorical class labels: Example

```
# The returned DataFrame has the following schema (attributes)
# - attr1: double (nullable = true)
# - attr2: double (nullable = true)
# - attr3: double (nullable = true)
# - 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 original features (i.e., the value of the original attributes
# attr1, attr2, attr3) and the predicted class for each record
predictions = predictionsDF.select("attr1", "attr2", "attr3", "predictedLabel")
```

Categorical class labels: Example

```
# The returned DataFrame has the following schema (attributes)
# - attr1: double (nullable = true)
# - attr2: double (nullable = true)
# - attr3: double (nullable = true)
# - 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)
```

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

```
# Select only the original features (i.e., the value of the original attributes
# attr1, attr2, attr3) and the predicted class for each record
predictions = predictionsDF.select("attr1", "attr2", "attr3", "predictedLabel")
```

Categorical class labels: Example

```
# Save the result in an HDFS output folder  
predictions.write.csv(outputPath, header="true")
```

Textual data management and classification

Textual data classification

- The following slides show how to
 - Create a classification model based on the logistic regression algorithm for **textual documents**
 - A set of specific preprocessing estimators and transformers are used to preprocess textual data
 - Apply the model to new textual documents
- The input training dataset represents a textual document collection
 - Each line contains one document and its class
 - The class label
 - A list of words (the text of the document)

Textual data classification

- Consider the following example file
Label,Text
1,The Spark system is based on scala
1,Spark is a new distributed system
0,Turin is a beautiful city
0,Turin is in the north of Italy
- It contains four textual documents
- Each line contains two attributes
 - The class label (first attribute)
 - The text of the document (second attribute)

Textual data classification

- Input data before preprocessing

Label	Text
1	The Spark system is based on scala
1	Spark is a new distributed system
0	Turin is a beautiful city
0	Turin is in the north of Italy

Textual data classification

- A set of preprocessing steps must be applied on the textual attribute before generating a classification model

Textual data classification

1. Since Spark ML algorithms work only on “Tables” and double values, the textual part of the input data must be translated in a set of attributes to represent the data as a table
 - Usually a table with an attribute for each word is generated

Textual data classification

- 2. Many words are useless (e.g., conjunctions)
 - Stopwords are usually removed

Textual data classification

- The words appearing in almost all documents are not characterizing the data
 - Hence, they are not very important for the classification problem
- The words appearing in few documents allow distinguish the content of those documents (and hence the class label) with respect to the others
 - Hence, they are very important for the classification problem

Textual data classification

3. Traditionally a weight, based on the **TF-IDF** measure, is used to assign a difference importance to the words based on their frequency in the collection

Textual data classification

- Input data after the preprocessing transformations (tokenization, stopword removal, TF-IDF computation)

Label	Spark	system	scala
1	0.5	0.3	0.75	..
1	0.5	0.3	0	...
0	0	0	0	...
0	0	0	0	...

Textual data classification

- The DataFrame associated with the input data after the preprocessing transformations must contain, as usual, the columns
 - label
 - Class label value
 - features
 - The preprocessed version of the input text
 - There are also some other intermediate columns, related to applied transformations, but they are not considered by the classification algorithm

Textual data classification

- The DataFrame associated with the input data after the preprocessing transformations

label	features	text
1	[0.5, 0.3, 0.75, ..]	The Spark system is based on scala
1	[0.5, 0.3, 0, ..]	Spark is a new distributed system
0	[0, 0, 0, ..]	Turin is a beautiful city
0	[0, 0, 0, ..]	Turin is in the north of Italy

Textual data classification

- The DataFrame associated with the input data after the preprocessing transformations

label	features	text
1	[0.5, 0.3, 0.75, ..]	The Spark system is based on scala
1	[0.5, 0.3, 0, ..]	Spark is a new distributed system
0	[0, 0, 0, ..]	Turin is a beautiful city
0	[0, 0, 0, ..]	Turin is in the north of Italy

Only "label" and "features" are considered by the classification algorithm

Textual data classification

- In the following solution we will use a set of new Transformers to prepare input data
 - Tokenizer
 - To split the input text in words
 - StopWordsRemover
 - To remove stopwords
 - HashingTF
 - To compute the (approximate) term frequency of each input term
 - IDF
 - To compute the inverse document frequency of each input word

Textual data classification

- The input data (training and unlabeled data) are stored in input csv files
 - Each line contains two attributes
 - The class label (label)
 - The text of the document (text)
- We infer a linear regression model on the training data and apply the model on the unlabeled data

Textual data classification: example

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import Tokenizer
from pyspark.ml.feature import StopWordsRemover
from pyspark.ml.feature import HashingTF
from pyspark.ml.feature import IDF
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel

# input and output folders
trainingData = "ex_dataText/trainingData.csv"
unlabeledData = "ex_dataText/unlabeledData.csv"
outputPath = "predictionsLRPipelineText/"
```

Textual data classification: example

```
# *****  
# Training step  
# *****  
  
# Create a DataFrame from trainingData.csv  
# Training data in raw format  
trainingData = spark.read.load(trainingData,\n    format="csv",\  
    header=True,\  
    inferSchema=True)
```


Textual data classification: example

```
# Configure an ML pipeline, which consists of five stages:
# tokenizer -> split sentences in set of words
# remover -> remove stopwords
# hashingTF -> map set of words to a fixed-length feature vectors (each
# word becomes a feature and the value of the feature is the frequency of
# the word in the sentence)
# idf -> compute the idf component of the TF-IDF measure
# lr -> logistic regression classification algorithm

# The Tokenizer splits each sentence in a set of words.
# It analyzes the content of column "text" and adds the
# new column "words" in the returned DataFrame
tokenizer = Tokenizer().setInputCol("text").setOutputCol("words")
```

Textual data classification: example

```
# Remove stopwords.  
# The StopWordsRemover component returns a new DataFrame with  
# a new column called "filteredWords". "filteredWords" is generated  
# by removing the stopwords from the content of column "words"  
remover = StopWordsRemover()\br/>.setInputCol("words")\br/>.setOutputCol("filteredWords")
```

Textual data classification: example

```
# Map words to a features
# Each word in filteredWords must become a feature in a Vector object
# The HashingTF Transformer can be used to perform this operation.
# This operations is based on a hash function and can potentially
# map two different words to the same "feature". The number of conflicts
# in influenced by the value of the numFeatures parameter.
# The "feature" version of the words is stored in Column "rawFeatures".
# Each feature, for a document, contains the number of occurrences
# of that feature in the document (TF component of the TF-IDF measure)
hashingTF = HashingTF()\
.setNumFeatures(1000)\
.setInputCol("filteredWords")\
.setOutputCol("rawFeatures")
```

Textual data classification: example

```
# Apply the IDF transformation/computation.  
# Update the weight associated with each feature by considering also the  
# inverse document frequency component. The returned new column  
# is called "features", that is the standard name for the column that  
# contains the predictive features used to create a classification model  
idf = IDF()\br/>  .setInputCol("rawFeatures")\br/>  .setOutputCol("features")    });
```

Textual data classification: example

```
# Create a classification model based on the logistic regression algorithm
# We can set the values of the parameters of the
# Logistic Regression algorithm using the setter methods.
lr = LogisticRegression()\
    .setMaxIter(10)\
    .setRegParam(0.01)
```

Textual data classification: example

```
# Define the pipeline that is used to create the logistic regression
# model on the training data.
# In this case the pipeline is composed of five steps
# - text tokenizer
# - stopword removal
# - TF-IDF computation (performed in two steps)
# - Logistic regression model generation
pipeline = Pipeline().setStages([tokenizer, remover, hashingTF, idf, lr])

# Execute the pipeline on the training data to build the
# classification model
classificationModel = pipeline.fit(trainingData)

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

Textual data classification: example

```
# *****  
# Prediction step  
# *****  
# Read unlabeled data  
# Create a DataFrame from unlabeledData.csv  
# Unlabeled data in raw format  
unlabeledData = spark.read.load(unlabeledData,\  
                                format="csv", header=True, inferSchema=True)
```

Textual data classification: example

```
# Make predictions on unlabeled documents by using the  
# Transformer.transform() method.  
# The transform will only use the 'features' columns  
predictionsDF = classificationModel.transform(unlabeledData)
```


Textual data classification: example

The returned DataFrame has the following schema (attributes)

|-- label: string (nullable = true)

|-- text: string (nullable = true)

|-- words: array (nullable = true)

| |-- element: string (containsNull = true)

|-- filteredWords: array (nullable = true)

| |-- element: string (containsNull = true)

|-- rawFeatures: vector (nullable = true)

|-- features: vector (nullable = true)

|-- rawPrediction: vector (nullable = true)

|-- probability: vector (nullable = true)

|-- prediction: double (nullable = false)

Select only the original features (i.e., the value of the original text attribute) and

the predicted class for each record

predictions = predictionsDF.select("text", "prediction")

Textual data classification: example

```
# Save the result in an HDFS output folder  
predictions.write.csv(outputPath, header="true")
```

Classification: Performance evaluation

Performance evaluation

- In order to test the goodness of algorithms there are some **evaluators**
- The **Evaluator** can be
 - a **BinaryClassificationEvaluator** for binary data
 - a **MulticlassClassificationEvaluator** for multiclass problems
- Provided metrics are:
 - Accuracy
 - Precision
 - Recall
 - F-measure

Performance evaluation

- Use the **MulticlassClassificationEvaluator** estimator from `pyspark.ml.evaluator` on a `DataFrame`
- The instantiated estimator has the method **evaluate()** that is applied on a `DataFrame`
 - It compares the predictions with the true label values
 - Output
 - The double value of the computed performance metric

Performance evaluation

- Parameters of MulticlassClassificationEvaluator
 - metricName
 - 'accuracy', 'f1', 'weightedPrecision', 'weightedRecall'
 - labelCol:input
 - Column with the true label/class value
 - predictionCol:
 - Input column with the predicted class/label value

Performance evaluation: Example

- In the following example, the set of labeled data is read from 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 (label)
 - The other three attributes (attr1, attr2, attr3) are the predictive attributes that are used to predict the value of the class label
 - All attributes are already double attributes
 - The input file has the header line

Performance evaluation: Example

- Consider the following example input labeled data file

label,attr1,attr2,attr3

1,0.0,1.1,0.1

0,2.0,1.0,-1.0

0,2.0,1.3,1.0

1,0.0,1.2,-0.5

.....

Performance evaluation: Example

- We initially split the labeled data set in two subsets
 - Training set: 75% of the labeled data
 - Test set: 25% of the labeled data
- Then, we infer/train a logistic regression model on the training set
- Finally, we evaluate the prediction quality of the inferred model on both the test set and the training set

Performance evaluation: Example

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel

# input and output folders
labeledData = "ex_dataValidation/labeledData.csv"
outputPath = "predictionsLRPipelineValidation/"
```

Performance evaluation: Example

```
# Create a DataFrame from labeledData.csv
# Training data in raw format
labeledDataDF = spark.read.load(labeledData,\
                                format="csv", header=True,\
                                inferSchema=True)

# Split labeled data in training and test set
# training data : 75%
# test data: 25%
trainDF, testDF = labeledDataDF.randomSplit([0.75, 0.25], seed=10)
this is good idea to cache it
```

Performance evaluation: Example

```
# Create a DataFrame from labeledData.csv
# Training data in raw format
labeledDataDF = spark.read.load(labeledData,\
                                format="csv", header=True,\
                                inferSchema=True)
```

```
# Split labeled data in training and test set
# training data : 75%
# test data: 25%
```

```
trainDF, testDF = labeledDataDF.randomSplit([0.75, 0.25], seed=10)
```

`randomSplit` can be used to split the content of an input DataFrame in subsets

Performance evaluation: Example

```
# *****  
# Training step  
# *****  
# Define an assembler to create a column (features) of type Vector  
# containing the double values associated with columns attr1, attr2, attr3  
assembler = VectorAssembler(inputCols=["attr1", "attr2", "attr3"],\  
                             outputCol="features")
```

Performance evaluation: Example

```
# Create a LogisticRegression object.  
# LogisticRegression is an Estimator that is used to  
# create a classification model based on logistic regression.  
lr = 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 0.01  
lr.setMaxIter(10)  
lr.setRegParam(0.01)
```

Performance evaluation: Example

```
# Define a pipeline that is used to create the logistic regression  
# model on the training data. The pipeline includes also  
# the preprocessing step  
pipeline = Pipeline().setStages([assembler, lr])
```

```
# Execute the pipeline on the training data to build the  
# classification model  
classificationModel = pipeline.fit(trainDF)
```

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

Performance evaluation: Example

```
# Make predictions on the test data using the transform() method of the  
# trained classification model transform uses only the content of 'features'  
# to perform the predictions. The model is associated with the pipeline and hence  
# also the assembler is executed  
predictionsDF = classificationModel.transform(testDF)
```


Performance evaluation: Example

The predicted value is column prediction

The actual label is column label

Define a set of evaluators

```
myEvaluatorAcc = MulticlassClassificationEvaluator(labelCol="label",\
                                                    predictionCol="prediction",\
                                                    metricName='accuracy')
```

```
myEvaluatorF1 = MulticlassClassificationEvaluator(labelCol="label",\
                                                    predictionCol="prediction",\
                                                    metricName='f1')
```

Performance evaluation: Example

```
myEvaluatorWeightedPrecision =  
    MulticlassClassificationEvaluator(labelCol="label",\  
                                     predictionCol="prediction",\  
                                     metricName='weightedPrecision')
```

```
myEvaluatorWeightedRecall = MulticlassClassificationEvaluator(labelCol="label",\  
                                                             predictionCol="prediction",\  
                                                             metricName='weightedRecall')
```

Performance evaluation: Example

```
# Apply the evaluators on the predictions associated with the test data  
# Print the results on the standard output
```

```
print("Accuracy on test data ", myEvaluatorAcc.evaluate(predictionsDF))  
print("F1 on test data ", myEvaluatorF1.evaluate(predictionsDF))  
print("Weighted recall on test data ",\  
      myEvaluatorWeightedRecall.evaluate(predictionsDF))  
print("Weighted precision on test data ",\  
      myEvaluatorWeightedPrecision.evaluate(predictionsDF))
```

Performance evaluation: Example

We compute the prediction quality also for the training data.

To check if the model is overfitted on the training data

Make predictions on the training data using the transform() method of the

trained classification model transform uses only the content of 'features'

to perform the predictions. The model is associated with the pipeline and hence

also the assembler is executed

```
predictionsTrainingDF = classificationModel.transform(trainDF)
```

Performance evaluation: Example

Apply the evaluators on the predictions associated with the test data
Print the results on the standard output

```
print("Accuracy on training data ",\
      myEvaluatorAcc.evaluate(predictionsTrainingDF))
print("F1 on training data ",\
      myEvaluatorF1.evaluate(predictionsTrainingDF))
print("Weighted recall on training data ",\
      myEvaluatorWeightedRecall.evaluate(predictionsTrainingDF))
print("Weighted precision on training data ",\
      myEvaluatorWeightedPrecision.evaluate(predictionsTrainingDF))
```

Classification: Parameter Tuning

Classification: Parameter Tuning

- The setting of the parameters of an algorithm is always a difficult task
- A “brute force” approach can be used to find the setting optimizing a quality index
 - The training data is split in two subsets
 - The first set is used to build a model
 - The second one is used to evaluate the quality of the model
 - The setting that maximizes a quality index (e.g., the prediction accuracy) is used to build the final model on the whole training dataset

Classification: Parameter Tuning

- One single split of the training set usually is biased
- Hence, the cross-validation approach is usually used
 - It creates **k splits** and **k models**
 - The **parameter setting** that achieves, on the average, the **best result on the k models** is selected as **final setting** of the algorithm's parameters

Classification: Parameter Tuning

- Spark supports a **brute-force grid-based approach** to evaluate a set of possible parameter settings on a pipeline
- Input:
 - An MLlib pipeline
 - A set of values to be evaluated for each input parameter of the pipeline
 - All the possible combinations of the specified parameter values are considered and the related models are automatically generated and evaluated by Spark
 - A quality evaluation metric to evaluate the result of the input pipeline
- Output
 - The model associated with the best parameter setting, in term of quality evaluation metric

Classification: Parameter Tuning - Example

- The following example shows how a grid-based approach can be used to tune a logistic regression classifier on a structured dataset
 - The pipeline that is repeated multiple times is based on the cross validation component
- The input data set is the same structured dataset used for the example of the evaluators

Classification: Parameter Tuning - Example

- The following parameters of the logistic regression algorithm are considered in the brute-force search/parameter tuning
 - Maximum iteration
 - 10, 100, 1000
 - Regulation parameter
 - 0.1, 0.01
 - 6 parameter configurations are evaluated (3×2)

Classification: Parameter Tuning - Example

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import ParamGridBuilder
from pyspark.ml.tuning import CrossValidator
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel
```

Classification: Parameter Tuning - Example

```
# input and output folders
labeledData = "ex_dataValidation/labeledData.csv"
unlabeledData = "ex_dataValidation/unlabeledData.csv"
outputPath = "predictionsLRPipelineTuning/"

# Create a DataFrame from labeledData.csv
# Training data in raw format
labeledDataDF = spark.read.load(labeledData,\
    format="csv",\
    header=True,\
    inferSchema=True)
```

Classification: Parameter Tuning - Example

```
# *****
```

```
# Training step
```

```
# *****
```

```
# Define an assembler to create a column (features) of type Vector
```

```
# containing the double values associated with columns attr1, attr2, attr3
```

```
assembler = VectorAssembler(inputCols=["attr1", "attr2", "attr3"],\
```

```
    outputCol="features")
```

Classification: Parameter Tuning - Example

Create a LogisticRegression object.

LogisticRegression is an Estimator that is used to

create a classification model based on logistic regression.

```
lr = LogisticRegression()
```

Define a pipeline that is used to create the logistic regression

model on the training data. The pipeline includes also the preprocessing step

```
pipeline = Pipeline().setStages([assembler, lr])
```

Classification: Parameter Tuning - Example

```
# We use a ParamGridBuilder to construct a grid of parameter values to
# search over.
# We set 3 values for lr.setMaxIter and 2 values for lr.regParam.
# This grid will evaluate 3 x 2 = 6 parameter settings for
# the input pipeline.
paramGrid = ParamGridBuilder()\
.addGrid(lr.maxIter, [10, 100, 1000])\
.addGrid(lr.regParam, [0.1, 0.01])\
.build()
```


Classification: Parameter Tuning - Example

We use a ParamGridBuilder to construct a grid of parameter values to search over.
We set 3 values for lr.setMaxIter and 2 values for lr.regParam.
This grid will evaluate $3 \times 2 = 6$ parameter settings for the input pipeline.

```
paramGrid = ParamGridBuilder()\n.addGrid(lr.maxIter, [10, 100, 1000])\n.addGrid(lr.regParam, [0.1, 0.01])\n.build()
```

There is one call to the addGrid method for each parameter that we want to set. Each call to the addGrid method is characterized by

- The parameter we want to consider
- The list of values to test/to consider

Classification: Parameter Tuning - Example

```
# We now treat the Pipeline as an Estimator, wrapping it in a  
# CrossValidator instance. This allows us to jointly choose parameters  
# for all Pipeline stages.  
# CrossValidator requires  
# - an Estimator  
# - a set of Estimator ParamMaps  
# - an Evaluator.  
cv = CrossValidator()\  
  .setEstimator(pipeline)\  
  .setEstimatorParamMaps(paramGrid)\  
  .setEvaluator(BinaryClassificationEvaluator())\  
  .setNumFolds(3)
```

Classification: Parameter Tuning - Example

Here, we set

- The pipeline to be evaluated
- The set of parameter values to be considered
- The evaluator (i.e., the object that is used to compute the quality measure that is used to evaluate the quality of the model)
- The number of folds to consider (i.e., the number of repetitions)

- an Evaluator.

```
cv = CrossValidator()\n.setEstimator(pipeline)\n.setEstimatorParamMaps(paramGrid)\n.setEvaluator(BinaryClassificationEvaluator())\n.setNumFolds(3)
```

Classification: Parameter Tuning - Example

```
# Run cross-validation. The result is the logistic regression model  
# based on the best set of parameters (based on the results of the  
# cross-validation operation).  
tunedLRmodel = cv.fit(labeledDataDF)  
  
# Now, the tuned classification model can be used to predict the class label  
# of new unlabeled data
```

Classification: Parameter Tuning - Example

Run cross-validation. The result is the logistic regression model
based on the best set of parameters (based on the results of the
cross-validation operation).

```
tunedLRmodel = cv.fit(labeledDataDF)
```

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

The returned model is the one associated with the best parameter setting, based on the result of the cross-validation test

Classification: Parameter Tuning - Example

```
# *****
```

```
# Prediction step
```

```
# *****
```

```
# Create a DataFrame from unlabeledData.csv
```

```
# Unlabeled data in raw format
```

```
unlabeledData = spark.read.load(unlabeledData,\n                                format="csv", header=True, inferSchema=True)
```

```
# Make predictions on the unlabeled data using the transform() method of the
```

```
# trained tuned classification model transform uses only the content of 'features'
```

```
# to perform the predictions. The model is associated with the pipeline and hence
```

```
# also the assembler is executed
```

```
predictionsDF = tunedLRmodel.transform(unlabeledData)
```

Classification: Parameter Tuning - Example

```
# 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)

# Select only the original features (i.e., the value of the original attributes
# attr1, attr2, attr3) and the predicted class for each record
predictions = predictionsDF.select("attr1", "attr2", "attr3", "prediction")

# Save the result in an HDFS output folder
predictions.write.csv(outputPath, header="true")
```

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 an input record/data 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 DataFrames by combining the following methods:
 - `read`, `format("libsvm")`, and `load(inputpath)`
- The returned 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

...

```
spark.read.format("libsvm")\  
    .load("sample_libsvm_data.txt")
```

..

Clustering algorithms

Clustering algorithms

- Spark MLlib provides a (limited) set of clustering algorithms
 - K-means
 - Bisecting k-means
 - Gaussian Mixture Model (GMM)

Clustering

- Each clustering algorithm has its own parameters
- However, all the provided algorithms identify a set of groups of objects/clusters and assign each input object to one single cluster
- All the clustering algorithms available in Spark work only with numerical data
 - Categorical values must be mapped to integer values (i.e., numerical values)

Clustering

- The **input** of the MLlib clustering algorithms is a DataFrame containing a column called **features of type Vector**
- The clustering algorithm clusters the input records by considering only the content of features
 - The other columns, if any, are not considered

Clustering: Example of input data

- Example of input data
 - A set of customer profiles
 - We want to group customers in groups based on their characteristics

MonthlyIncome	NumChildren
1400.0	2
11105.5	0
2150.0	2

Clustering: Example of input data

- Input training data

MonthlyIncome	NumChildren
1400.0	2
11105.5	0
2150.0	2

- Input DataFrame that must be generated as input for the MLlib clustering algorithms

features
[1400.0 , 2.0]
[11105.5, 0.0]
[2150.0 , 2.0]

Clustering: Example of input data

The values of all input attributes are “stored” in a vector of doubles (one vector for each input record).
The generated DataFrame contains a column called features containing the vectors associated with the input records.

- Input training

MonthlyIncome	NumChildren
1400.0	2
11105.5	0
2150.0	2

- Input DataFrame that must be generated as input for the MLlib clustering algorithms

features
[1400.0 , 2.0]
[11105.5, 0.0]
[2150.0 , 2.0]

Clustering: main steps

- Clustering with Mlib

1. Create a DataFrame with the features column
2. Define the clustering pipeline and run the fit() method on the input data to infer the clustering model (e.g., the centroids of the k-means algorithm)
 - This step returns a clustering model
3. Invoke the transform() method of the inferred clustering model on the input data to assign each input record to a cluster
 - This step returns a new DataFrame with the new column “prediction” in which the cluster identifier is stored for each input record

K-means clustering algorithm

K-means clustering algorithm

- K-means is one of the most popular clustering algorithms
- It is characterized by one important parameter
 - The number of clusters **K**
 - The choice of **K** is a complex operation
- It is able to identify only spherical shaped clusters

K-means clustering algorithm

- The following slides show how to apply the **K-means algorithm** provided by MLlib
- The input dataset is a structured dataset with a fixed number of attributes
 - All the attributes are numerical attributes

K-means clustering algorithm

- Example of input file

attr1,attr2,attr3

0.5,0.9,1.0

0.6,0.6,0.7

.....

- In the following example code we suppose that the input data are already normalized
 - I.e., all values are already in the range [0-1]
 - Scalers/Normalizers can be used to normalized data if it is needed

K-means clustering algorithm: Example

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.clustering import KMeans
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel

# input and output folders
inputData = "ex_datakmeans/dataClusteering.csv"
outputPath = "clusterskmeans/"

# Create a DataFrame from dataClusteering.csv
# Training data in raw format
inputDataDF = spark.read.load(inputData,\
                               format="csv", header=True,\
                               inferSchema=True)
```

K-means clustering algorithm: Example

```
# Define an assembler to create a column (features) of type Vector
# containing the double values associated with columns attr1, attr2, attr3
assembler = VectorAssembler(inputCols=["attr1", "attr2", "attr3"],\
                             outputCol="features")

# Create a k-means object.
# k-means is an Estimator that is used to
# create a k-means algorithm
km = KMeans()

# Set the value of k ( = number of clusters)
km.setK(2)
```

K-means clustering algorithm: Example

```
# Define the pipeline that is used to cluster
```

```
# the input data
```

```
pipeline = Pipeline().setStages([assembler, km])
```

```
# Execute the pipeline on the data to build the
```

```
# clustering model
```

```
kmeansModel = pipeline.fit(inputDataDF)
```

```
# Now the clustering model can be applied on the input data
```

```
# to assign them to a cluster (i.e., assign a cluster id)
```

```
# The returned DataFrame has the following schema (attributes)
```

```
# - features: vector (values of the attributes)
```

```
# - prediction: double (the predicted cluster id)
```

```
# - original attributes attr1, attr2, attr3
```

```
clusteredDataDF = kmeansModel.transform(inputDataDF)
```

K-means clustering algorithm: Example

```
# Define the pipeline that is used to cluster
# the input data
pipeline = Pipeline().setStages([assembler, km])

# Execute the pipeline on the data to build the
# clustering model
kmeansModel = pipeline.fit(inputDataDF)
```

The returned DataFrame has a new column (prediction) in which the “predicted” cluster identifier (an integer) is stored for each input record.

```
# - features: vector (values of the attributes)
# - prediction: double (the predicted cluster id)
# - original attributes attr1, attr2, attr3
```

```
clusteredDataDF = kmeansModel.transform(inputDataDF)
```

K-means clustering algorithm: Example

```
# Select only the original columns and the clusterID (prediction) one
# I rename prediction to clusterID
clusteredData = clusteredDataDF\
.select("attr1", "attr2", "attr3", "prediction")\
.withColumnRenamed("prediction", "clusterID")

# Save the result in an HDFS output folder
clusteredData.write.csv(outputPath, header="true")
```

Regression algorithms

Regression algorithms

- Spark MLlib provides also a set of regression algorithms
 - Linear regression
 - Decision tree regression
 - Random forest regression
 - Survival regression
 - Isotonic regression

Regression algorithms

- A regression algorithm is used to predict the value of a continuous attribute (the target attribute) by applying a model on the predictive attributes
- The model is trained on a set of training data
 - i.e., a set of data for which the value of the target attribute is known
- And it is applied on new data to predict the target attribute

Regression algorithms

- The regression algorithms available in Spark work only on numerical data
 - They work similarly to classification algorithms, but they **predict continuous numerical values** (the target attribute is a continuous numerical attribute)
- The **input** data must be transformed in a DataFrame having the following attributes:
 - **label: double**
 - The continuous numerical value to be predicted
 - **features: Vector of doubles**
 - Predictive features

Regression algorithms

- The main steps used to infer a regression model with MLlib are the same we use to infer a classification model
 - The difference is only given by the type of the target attribute to predict

Linear regression and structured data

Linear regression and structured data

- Linear regression is a popular, effective and efficient regression algorithm
- The following slides show how to instantiate a linear regression algorithm in Spark and apply it on new data
- The input dataset is a structured dataset with a fixed number of attributes
 - One attribute is the target attribute (the label)
 - We suppose the first column contains the target attribute
 - The others are predictive attributes that are used to predict the value of the target attribute

Linear regression and structured data

- Consider the following example file

label,attr1,attr2,attr3

2.0,0.0,1.1,0.1

5.0,2.0,1.0,-1.0

5.0,2.0,1.3,1.0

2.0,0.0,1.2,-0.5

.....

- Each record has three predictive attributes and the target attribute
 - The first attribute (label) is the target attribute
 - The other attributes (attr1, attr2, attr3) are the predictive attributes

Linear regression and structured data: Example

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel

# input and output folders
trainingData = "ex_dataregression/trainingData.csv"
unlabeledData = "ex_dataregression/unlabeledData.csv"
outputPath = "predictionsLinearRegressionPipeline/"
```

Linear regression and structured data: Example

```
# *****  
# Training step  
# *****  
  
# Create a DataFrame from trainingData.csv  
# Training data in raw format  
trainingData = spark.read.load(trainingData,\  
                                format="csv", header=True,\  
                                inferSchema=True)  
  
# Define an assembler to create a column (features) of type Vector  
# containing the double values associated with columns attr1, attr2, attr3  
assembler = VectorAssembler(inputCols=["attr1", "attr2", "attr3"],\  
                             outputCol="features")
```


Linear regression and structured data: Example

```
# Create a LinearRegression object.  
# LinearRegression is an Estimator that is used to  
# create a regression model based on linear regression  
lr = LinearRegression()  
  
# We can set the values of the parameters of the  
# Linear 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 0.01  
lr.setMaxIter(10)  
lr.setRegParam(0.01)
```

Linear regression and structured data: Example

```
# Define a pipeline that is used to create the linear regression
# model on the training data. The pipeline includes also
# the preprocessing step
pipeline = Pipeline().setStages([assembler, lr])

# Execute the pipeline on the training data to build the
# regression model
regressionModel = pipeline.fit(trainingData)

# Now, the regression model can be used to predict the target attribute value
# of new unlabeled data
```

Linear regression and structured data: Example

```
# Create a DataFrame from unlabeledData.csv
# Unlabeled data in raw format
unlabeledData = spark.read.load(unlabeledData,\
                                format="csv", header=True, inferSchema=True)

# Make predictions on the unlabeled data using the transform() method of the
# trained regression model transform uses only the content of 'features'
# to perform the predictions. The model is associated with the pipeline and hence
# also the assembler is executed
predictionsDF = regressionModel.transform(unlabeledData)
```

Linear regression and structured data: Example

```
# The returned DataFrame has the following schema (attributes)
# - attr1
# - attr2
# - attr3
# - original attributes
# - features: vector (values of the attributes)
# - label: double (actual value of the target variable)
# - prediction: double (the predicted continuous value of the target variable)

# Select only the original features (i.e., the value of the original attributes
# attr1, attr2, attr3) and the predicted value of the target variable for each record
predictions = predictionsDF.select("attr1", "attr2", "attr3", "prediction")

# Save the result in an HDFS output folder
predictions.write.csv(outputPath, header="true")
```

Linear regression and textual data

Linear regression and textual data

- The linear regression algorithms can be used also when the input dataset is a collection of documents/texts
- Also in this case the text must be mapped to a set of continuous attributes

Linear regression and parameter setting

Linear regression and parameter setting

- The tuning approach that we used for the classification problem can also be used to optimize the regression problem
- The only difference is given by the used evaluator
 - In this case the difference between the actual value and the predicted one must be computed

Itemset and Association rule mining

Itemset and Association rule mining

- Spark MLlib provides
 - An **itemset mining algorithm** based on the **FP-growth** algorithm
 - That extracts all the sets of items (of any length) with a minimum frequency
 - A **rule mining algorithm**
 - That extracts the association rules with a minimum frequency and a minimum confidence
 - **Only** the rules with one single item in the consequent of the rules are extracted

Itemset and Association rule mining

- The input dataset in this case is a set of transactions
- Each transaction is defined as a set of items
- Transactional dataset example
 - A B C D
 - A B
 - B C
 - A D E
- The example dataset contains 4 transactions
- The distinct items are A, B, C, D, E

The FP-Growth algorithm and Association rule mining

The FP-Growth algorithm

- FP-growth is one of the most popular and efficient itemset mining algorithms
- It is characterized by one single parameter
 - The minimum support threshold (**minsup**)
 - i.e., the minimum frequency of the itemset in the input transactional dataset
 - It is a real value in the range (0-1]
 - The minsup threshold is used to limit the number of mined itemsets
- The input dataset is a transactional dataset

Association Rule Mining

- Given a set of frequent itemsets, the frequent association rules can be mined
- An association rule is mined if
 - Its frequency is greater than the minimum support threshold (**minsup**)
 - i.e., a minimum frequency
 - The minsup value is specified during the itemset mining step and not during the association rule mining step
 - Its confidence is greater than the minimum confidence threshold (**minconf**)
 - i.e., a minimum “correlation”
 - It is a real value in the range [0-1]

The FP-Growth algorithm

- The MLlib implementation of FP-growth is based on DataFrames
- Differently from the other algorithms, the FP-growth algorithm is not invoked by using pipelines

Itemset and Association Rule Mining

- Itemset and association rule mining
 - Instantiate an FP-Growth object
 - Invoke the fit(input data) method on the FP-Growth object
 - Retrieve the sets of frequent itemset and association rules by invoking the following methods of on the FP-Growth object
 - freqItemsets()
 - associationRules()

Itemset and Association Rule Mining: Input data

- The input of the MLlib itemset and rule mining algorithm is a DataFrame containing a column called **items**
 - Data type: array of values
- Each record of the input DataFrame contains one transaction, i.e., a set of items

Itemset and Association Rule Mining: Input data

- Example of input data

transactions

A B C D

A B

B C

A D E

Itemset and Association Rule Mining: Input data

- Input data

transactions

A B C D

A B

B C

A D E

- The column items must be created before invoking FP-growth

items
[A, B, C, D]
[A, B]
[B, C]
[A, D, E]

Itemset and Association Rule Mining: Input data

Each input line is “stored” in an array of strings
The generated DataFrame contains a column called items, which is an ArrayType, containing the lists of items associated with the input transactions.

- Input data

transactions
A B C D
A B
B C
A D E

- The column items must be created before invoking FP-growth

items
[A, B, C, D]
[A, B]
[B, C]
[A, D, E]

Itemset and Association Rule Mining: Example

- The following slides show how to
 - Extract the set of frequent itemsets from a transactional dataset and the association rules from the extracted frequent itemsets
- The input dataset is a transactional dataset
 - Each line of the input file contains a transaction, i.e., a set of items

Itemset and Association Rule Mining: Example

- Example of input data

transactions

A B C D

A B

B C

A D E

Itemset and Association Rule Mining: Example

```
from pyspark.ml.fpm import FPGrowth
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel
from pyspark.sql.functions import col, split

# input and output folders
transactionsData = "ex_dataitemsets/transactions.csv"
outputPathItemsets = "Itemsets/"
outputPathRules = "Rules/"

# Create a DataFrame from transactions.csv
transactionsDataDF = spark.read.load(transactionsData,\
    format="csv", header=True,\
    inferSchema=True)
```

Itemset and Association Rule Mining: Example

```
# Transform Column transactions into an ArrayType
trsDataDF = transactionsDataDF\
.selectExpr('split(transactions, " ")')\
.withColumnRenamed("split(transactions, )", "items")
```


Itemset and Association Rule Mining: Example

```
# Transform Column transactions into an ArrayType  
trsDataDF = transactionsDataDF\  
    .selectExpr('split(transactions, " ")')\  
    .withColumnRenamed("split(transactions, )", "items")
```

This is the `pyspark.sql.functions.split()` function.
It returns an SQL `ArrayType`

Itemset and Association Rule Mining: Example

```
# Transform Column transactions into an ArrayType
trsDataDF = transactionsDataDF\
.selectExpr('split(transactions, " ")')\
.withColumnRenamed("split(transactions, )", "items")

# Create an FP-growth Estimator
fpGrowth = FPGrowth(itemsCol="items", minSupport=0.5, minConfidence=0.6)

# Extract itemsets and rules
model = fpGrowth.fit(trsDataDF)

# Retrieve the DataFrame associated with the frequent itemsets
dfItemsets = model.freqItemsets

# Retrieve the DataFrame associated with the frequent rules
dfRules = model.associationRules
```

Itemset and Association Rule Mining: Example

Save the result in an HDFS output folder
`dfItemsets.write.json(outputPathItemsets)`

Save the result in an HDFS output folder
`dfRules.write.json(outputPathRules)`

The result is stored in a JSON file because itemsets and rules are stored in columns associated with the data type Array.
Hence, CSV files cannot be used to store the result.