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

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

- Inpu

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

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

- Inpu

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

- Inpu

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

```
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

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,\
               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)
```

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 0.0.1
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,\
                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 unlabled 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,\
                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 0.0.1
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,\
          format="csv", header=True, inferSchema=True)

# Make predictions on the unlabled 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 = "predictionsDT/"
```

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,\
                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 Decision Tree
# For example we can set the measure that is used to decide if a
# node must be split. In this case we set gini index
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])


#
#
c
```

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,\
            format="csv", header=True, inferSchema=True)

# Make predictions on the unlabled 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

- Inpu

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

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

- Input 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

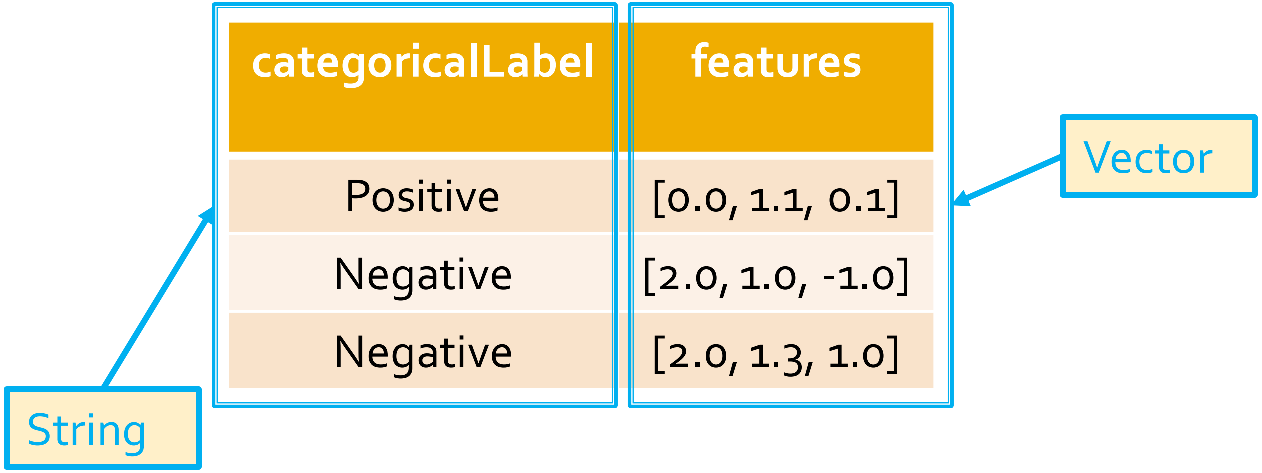
  categoricalLabel,attr1,attr2,attr3

  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]

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	
null	

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",\
                                    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 Decision Tree
# For example we can set the measure that is used to decide if a
# node must be split.
# In this case we set gini index
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",\
                              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])
```

This Pipeline is composed of four steps

```
# 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

```
# *************************
# 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 unlabled 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 predicte
# - predictedLabel: string (nullable =

# 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")
```

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

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,\
               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()\
.setInputCol("words")\
.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()\
.setInputCol("rawFeatures")\
.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)
```

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 0.0.1
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 x 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 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()
```

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 or repetitions)

```
# - an Evaluator.
cv = CrossValidator()\
.setEstimator(pipeline)\
.setEstimatorParamMaps(paramGrid)\
.setEvaluator(BinaryClassificationEvaluator())\
.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,\
               format="csv", header=True, inferSchema=True)

# Make predictions on the unlabled 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")
```