Table of contents

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

All the available classification algorithms are based on two phases:

1. Model generation based on a set of training data
2. 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.

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`, and both dense and sparse vectors can be used

Example

Consider the following classification problem: the goal is to predict if new customers are good customers or not based on their monthly income and number of children.
The predictive attributes are

- Monthly income
- Number of children

The class label (target attribute) is "Customer type":

- "Good customer", mapped to 1
- "Bad customer", mapped to 0

Example of input training data
The training data is 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

Example of input training DataFrame
The input training DataFrame that must be provided as input to train an MLlib classification algorithm must have the following structure

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

Notice that

- The categorical values of "CustomerType" (the class label column) must be mapped to integer data values (then casted to doubles).
- The values of the predictive attributes are stored in vectors of doubles. One single vector for each input record.
- In the generated DataFrame the names of the predictive attributes are not preserved.

1 Structured data classification

1.1 Example of logistic regression and structured data

The following paragraphs 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.

1.1.1 Training data

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: it assumed 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 (no need to convert them), and the input file has the header line.

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.

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.

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	$[0.0, 1.2, -0.5]$

- Data type of "label" is double
- Data type of "features" is `pyspark.ml.linalg.Vector`

1.1.2 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. The goal is 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.

Consider the following 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. Notice that the first column is empty (the content before the first comma is the empty string).

Also the unlabeled data must be stored into a DataFrame containing two columns: "label" and "features". So, "label" column is required also for unlabeled data, but its value is set to null for all records.

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

1.1.3 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, that, for each input unlabeled record, contains the predicted class label value
- Two columns, associated with the probabilities of the predictions (these columns are not considered in the example)

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	...	...

The "prediction" column contains the predicted class label values.

1.1.4 Example code

```python
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/"

# *************************
# 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"
)

# Apply the assembler to create column features for the training data
trainingDataDF = assembler.transform(trainingData)

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

# It is possible to 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, the number of maximum iterations is set to 10
# and the regularization parameter is set to 0.01
lr.setMaxIter(10)
lr.setRegParam(0.01)

```

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

# *************************
# 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)

# 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")
```

1.2 Pipelines

In the previous solution the same preprocessing steps were applied on both training and unlabeled data (the same assembler on both input data). It is possible to use a pipeline to specify the common phases we apply on both input data sets.

Example

```python
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/"

# *************************
# 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"
)

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

# 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)

# 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]) # <1>

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

1. `assembler`: the sequence of transformers and estimators to apply on the input data

1.3 Decision trees and structured data

The following paragraphs 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.

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.

Example

```python
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/"

# *************************
# 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"
)

# 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")

# 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]) # <1>

# 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

```

1. `assembler`: the sequence of transformers and estimators to apply on the input data. A decision tree algorithm is used in this case.

2 Categorical class labels

Usually the class label is a categorical value (i.e., a string). However, 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.

Consider the following 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

A modified input DataFrame must be generated as input for the MLlib classification algorithms

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

Notice that the categorical values of "categoricalLabel" (the class label column) must mapped to integer data values (finally casted to doubles).

2.1 `StringIndexer` and `IndexToString`

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 (then casted to a double);
- `IndexToString` is used to perform the opposite operation.

All in all, these are the 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);
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.

Notice that 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.

2.1.1 Training data

Given the following 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
```

The initial training DataFrame will be

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

- The type of “categoricalLabel” is String
- The type of “features” is Vector

After applying `StringIndexer`, the training DataFrame will be

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

“label” contains the mapping generated by `StringIndexer`:

- “Positive”: 1.0
- “Negative”: 0.0

2.1.2 Unalabeled data

Given the 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
```

The initial unlabeled DataFrame will be

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

After performing the prediction, and applying `IndexToString`, the output DataFrame will be

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	Negative	

- "prediction" contains the predicted label, expressed as a number
- "predictedLabel" contains the predicted label, expressed as a category (original name)

Example

In this example, the input training data is stored in a text file that contains one record/data point per line, and the records/data points are structured data with a fixed number of attributes (four)

- One attribute is the class label ("categoricalLabel"): this is a categorical attribute that can assume two values, "Positive" or "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.
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.
The goal is to predict the class label value of each unlabeled data by applying the classification model that has been inferred on the training data.

```python
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/"

```

```
# ************************
# 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"
)

# 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( # <1>
    inputCol="categoricalLabel"
    outputCol="label",
    handleInvalid="keep"
).fit(trainingData)

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

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

# 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( # <2>
    inputCol="prediction",
    outputCol="predictedLabel",
    labels=labelIndexer.labels
)

# 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]) # <3>

# 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

# *************************
# 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)

# 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") # <4>

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

1. 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.
2. 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 of integer to original string value is the one of “labelIndexer”.
3. This `Pipeline` is composed of four steps.
4. The “predictedLabel” field is the column containing the predicted categorical class label for the unlabeled data.

3 Textual data management and classification

The following paragraphs 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, where each line contains one document and its class

- The class label
- A list of words (the text of the document)

Example

Given the following example training 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, and each line contains two attributes, that are the class label (first attribute) and the text of the document (second attribute).
The input data before preprocessing, represented as a DataFrame, is

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

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

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.
2. Many words are useless (e.g., conjunctions): stopwords are usually removed. In general,
    - the words appearing in almost all documents are not characterizing the data, and so they are not very important for the classification problem;
    - the words appearing in few documents allow to distinguish the content of those documents (and hence the class label) with respect to the others, and so they are very important for the classification problem.
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.

Example

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	...

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.

Example

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.

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.

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.

Example

```python
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"
```

4 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

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, and the output is the double value of the computed performance metric.

The parameters of `MulticlassClassificationEvaluator` are

- `metricName`: type of metric to compute. It can assume the following values
  - `"accuracy"`
  - `"f1"`
  - `"weightedPrecision"`
  - `"weightedRecall"`
- `labelCol`: input column with the true label/class value
- `predictionCol`: input column with the predicted class/label value

Example

In this example, the set of labeled data is read from a text file that contains one record/data point per line, and 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, and the input file has the header line.
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
```

Follow these steps

1. Split the labeled data set in two subsets

  - Training set: 75% of the labeled data
  - Test set: 25% of the labeled data

2. Infer/train a logistic regression model on the training set
3. Evaluate the prediction quality of the inferred model on both the test set and the training set

```python
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/"

# 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) # <1>

# ************************
# 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"
)

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

# 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)

# 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])
```

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

5 Hyperparameter 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, by splitting the training data 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.

Using one single split of the training set usually leads to biased results, so the cross-validation approach is normally used

- Create 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 parameters

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

Example

This 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. The following parameters of the logistic regression algorithm are considered in the brute-force search/parameter tuning

- Maximum iteration: $[10, 100, 1000]$
- Regulation parameter: $[0.1, 0.01]$

In total, 6 parameter configurations are evaluated $(3 * 2)$.

```
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

# 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
)

# *************************
# 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"
)

# 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])

# 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() # <1>
```

1. 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, and the list of values to test/to consider.
2. Here the characteristics of the cross validation are set: the pipeline to be evaluated, the set of parameters to be considered, the evaluator (i.e., the object that is used to evaluate the quality of the model), and the number of folds to consider (i.e., the number of repetitions).
3. The returned model is the one associated with the best parameter setting, based on the result of the cross-validation test

6 Sparse labeled data

Frequently the training data are sparse (e.g., textual data are sparse: each document contains only a subset of the possible words), so sparse vectors are frequently used. MLlib supports reading training examples stored in the LIBSVM format: this is a commonly used textual format that is used to represent sparse documents/data points.

The LIBSVM format 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.
- `index#` are integer values representing the features
- `value#` are the (double) values of the features

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
```

LIBSVM files can be loaded into DataFrames by combining the following methods

- `read()`
- `format("libsvm")`
- `load(inputpath)`

The returned DataFrame has two columns:

- label: the double value associated with the label
- features: the sparse vector associated with the predictive features

Example

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