Big data: architectures and data analytics

Spark MLlib
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
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 following parameters of the logistic regression algorithm are considered:
  - Maximum iteration: 10, 100, 1000
  - Regulation parameter: 0.1, 0.01

6 parameter configurations are evaluated (3 x 2).

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

import org.apache.spark.api.java.*;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.apache.spark.ml.Pipeline;
import org.apache.spark.ml.PipelineStage;
import org.apache.spark.ml.classification.LogisticRegression;
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator;
import org.apache.spark.ml.linalg.Vector;
import org.apache.spark.ml.linalg.Vectors;
import org.apache.spark.ml.feature.LabeledPoint;
import org.apache.spark.ml.param.ParamMap;
import org.apache.spark.ml.tuning.CrossValidator;
import org.apache.spark.ml.tuning.CrossValidatorModel;
import org.apache.spark.ml.tuning.ParamGridBuilder;
```
public class SparkDriver {
    public static void main(String[] args) {
        String inputFileTraining;  // Training data file
        String inputFileTest;     // Test data file
        String outputPath;       // Output file path

        inputFileTraining = args[0];
        inputFileTest = args[1];
        outputPath = args[2];

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

        // Create a Java Spark Context from the Spark Session
        JavaSparkContext sc = new JavaSparkContext(ss.sparkContext());

        // Training step
        JavaRDD<String> trainingData = sc.textFile(inputFileTraining);

        // Read training data from a text file
        // Each line has the format: class-label, list of three numerical attribute values.
        // E.g., 1.0, 5.8, 0.517
        JavaRDD<String> trainingData = sc.textFile(inputFileTraining);
Classification: Parameter Tuning - Example

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

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

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

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

        // Return a LabeledPoint based on the content of
        // the current line
        return new LabeledPoint(classLabel, attrValues);
    });
Classification: Parameter Tuning - Example

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

Classification: Parameter Tuning - Example

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

// Define the pipeline that is used to create the logistic regression
// model on the training data.
// In this case the pipeline contains one single stage/step (the model
// generation step).
Pipeline pipeline = new Pipeline()
    .setStages(new PipelineStage[] {lr});
We use a `ParamGridBuilder` to construct a grid of parameter values to search over.

We set 3 values for `lr.setMaxIter` and 2 values for `lr.regParam`.

This grid will evaluate $3 \times 2 = 6$ parameter settings for the input pipeline.

```java
ParamMap[] paramGrid = new ParamGridBuilder()
    .addGrid(lr.maxIter(), new int[]{10, 100, 1000})
    .addGrid(lr.regParam(), new double[]{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
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.

```java
CrossValidator cv = new CrossValidator()
  .setEstimator(pipeline)
  .setEstimatorParamMaps(paramGrid)
  .setEvaluator(new BinaryClassificationEvaluator())
  .setNumFolds(3);
```

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)

```java
CrossValidator cv = new CrossValidator()
  .setEstimator(pipeline)
  .setEstimatorParamMaps(paramGrid)
  .setEvaluator(new 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).
CrossValidatorModel model = cv.fit(training);

// Now, the classification model can be used to predict the class label
// of new unlabeled data
Classification: Parameter Tuning - Example

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

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

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

    // The last three cells of fields contain the (numerical) values of the
    // three predictive attributes
    // Create an array of doubles containing those three values
    double[] attributesValues = new double[3];
    attributesValues[0] = Double.parseDouble(fields[1]);
    attributesValues[1] = Double.parseDouble(fields[2]);
    attributesValues[2] = Double.parseDouble(fields[3]);
Classification: Parameter Tuning - Example

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

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

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

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

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

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

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

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

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