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.

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.

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

**Output:**
- A quality evaluation metric to evaluate the result of the input pipeline
- 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
public class SparkDriver {
  public static void main(String[] args) {
    String inputFileTraining;
    String inputFileTest;
    String outputPath;

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

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

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

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

    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
      double[] attributesValues = new double[3];
      attributesValues[0] = Double.parseDouble(fields[1]);
      attributesValues[1] = Double.parseDouble(fields[2]);
      attributesValues[2] = Double.parseDouble(fields[3]);
      return new LabeledPoint(classLabel, Vectors.dense(attributesValues));
    });

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

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

// 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 x 2 = 6 parameter settings for
// the input pipeline.
ParamMap[] paramGrid = new ParamGridBuilder()
        .addGrid(lr.maxIter, new int[]{10, 100, 1000})
        .addGrid(lr.regParam, new double[]{0.1, 0.01})
        .build();

// 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 Evaulator.
CrossValidator cv = new CrossValidator()
        .setEstimator(pipeline)
        .setEstimatorParamMaps(paramGrid)
        .setEvaluator(new BinaryClassificationEvaluator())
        .setNumFolds(3);
// 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
CrossValidatorModel model = cv.fit(training);

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

// 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]);
    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).
    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" column
// The returned Dataset<Row> has the following schema (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");

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();
```