DATA WAREHOUSE & MINING

TA COURSE #08

CLASSIFICATION

TA COURSE #08

CLASSIFICATION

ALGO EXAMPLES

Access here for more detailed info

LOGISTIC REGRESSION

- A popular method to predict a categorical response.
  - It is a special case of Generalized Linear models that predicts the probability of the outcomes.
- Using binomial logistic regression to predict a binary outcome
- …Or using multinomial logistic regression to predict a multiclass outcome
- Use the family parameter to select between these two algorithms, or leave it unset and Spark will infer the correct variant.

BINOMIAL LOGISTIC REGRESSION

```scala
import org.apache.spark.ml.classification.LogisticRegression

// Load training data
val training = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

val lr = new LogisticRegression()
  .setMaxIter(10)
  .setRegParam(0.3)
  .setElasticNetParam(0.8)

// Fit the model
val lrModel = lr.fit(training)

// Print the coefficients and intercept for logistic regression
println(s"Coefficients: ${lrModel.coefficients} Intercept: ${lrModel.intercept}")

// We can also use the multinomial family for binary classification
val mlr = new LogisticRegression()
  .setMaxIter(10)
  .setRegParam(0.3)
  .setElasticNetParam(0.8)
  .setFamily("multinomial")

val mlrModel = mlr.fit(training)

// Print the coefficients and intercepts for logistic regression with multinomial family
println(s"Multinomial coefficients: ${mlrModel.coefficientMatrix}")
println(s"Multinomial intercepts: ${mlrModel.interceptVector}")
```

MULTINOMIAL LOGISTIC REGRESSION

```scala
import org.apache.spark.ml.classification.LogisticRegression

// Load training data
val training = spark
  .read
  .format("libsvm")
  .load("data/mllib/sample_multiclass_classification_data.txt")

val lr = new LogisticRegression()
  .setMaxIter(10)
  .setRegParam(0.3)
  .setElasticNetParam(0.8)

// Fit the model
val lrModel = lr.fit(training)

// Print the coefficients and intercept for multinomial logistic regression
println(s"Coefficients: \n${lrModel.coefficientMatrix}")
println(s"Intercepts: \n${lrModel.interceptVector}")

val trainingSummary = lrModel.summary

// Obtain the objective per iteration
val objectiveHistory = trainingSummary.objectiveHistory
println("objectiveHistory:")
objectiveHistory.foreach(println)

// for multiclass, we can inspect metrics on a per-label basis
println("False positive rate by label:")
trainingSummary.falsePositiveRateByLabel.zipWithIndex.foreach { case (rate, label) =>
  println(s"label $label: $rate")
}

println("True positive rate by label:")
trainingSummary.truePositiveRateByLabel.zipWithIndex.foreach { case (rate, label) =>
```

MULTINOMIAL LOGISTIC REGRESSION

```scala
  println(s"label $label: $rate")
}

println("Precision by label:")
trainingSummary.precisionByLabel.zipWithIndex.foreach { case (prec, label) =>
  println(s"label $label: $prec")
}

println("Recall by label:")
trainingSummary.recallByLabel.zipWithIndex.foreach { case (rec, label) =>
  println(s"label $label: $rec")
}


println("F-measure by label:")
trainingSummary.fMeasureByLabel.zipWithIndex.foreach { case (f, label) =>
  println(s"label $label: $f")
}

val accuracy = trainingSummary.accuracy
val falsePositiveRate = trainingSummary.weightedFalsePositiveRate
val truePositiveRate = trainingSummary.weightedTruePositiveRate
val fMeasure = trainingSummary.weightedFMeasure
val precision = trainingSummary.weightedPrecision
val recall = trainingSummary.weightedRecall
println(s"Accuracy: $accuracy\nFPR: $falsePositiveRate\nTPR: $truePositiveRate\n" +
  s"F-measure: $fMeasure\nPrecision: $precision\nRecall: $recall")
```

RANDOM FOREST CLASSIFIER

- Ensembles of decision trees.
- Random forests combine many decision trees in order to reduce the risk of overfitting.
- The spark.ml implementation supports random forests for binary and multiclass classification and for regression, using both continuous and categorical features.

RANDOM FOREST CLASSIFIER

```scala
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.{RandomForestClassificationModel, RandomForestClassifier}
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorIndexer}

// Load and parse the data file, converting it to a DataFrame.
val data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

// Index labels, adding metadata to the label column.
// Fit on whole dataset to include all labels in index.
val labelIndexer = new StringIndexer()
  .setInputCol("label")
  .setOutputCol("indexedLabel")
  .fit(data)
// Automatically identify categorical features, and index them.
// Set maxCategories so features with > 4 distinct values are treated as continuous.
val featureIndexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexedFeatures")
  .setMaxCategories(4)
  .fit(data)

// Split the data into training and test sets (30% held out for testing).
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))

// Train a RandomForest model.
val rf = new RandomForestClassifier()
  .setLabelCol("indexedLabel")
  .setFeaturesCol("indexedFeatures")
  .setNumTrees(10)

// Convert indexed labels back to original labels.
val labelConverter = new IndexToString()
  .setInputCol("prediction")
  .setOutputCol("predictedLabel")
```

RANDOM FOREST CLASSIFIER

```scala
  .setLabels(labelIndexer.labels)

// Chain indexers and forest in a Pipeline.
val pipeline = new Pipeline()
  .setStages(Array(labelIndexer, featureIndexer, rf, labelConverter))

// Train model. This also runs the indexers.
val model = pipeline.fit(trainingData)

// Make predictions.
val predictions = model.transform(testData)

// Select example rows to display.
predictions.select("predictedLabel", "label", "features").show(5)

// Select (prediction, true label) and compute test error.
val evaluator = new MulticlassClassificationEvaluator()
  .setLabelCol("indexedLabel")
  .setPredictionCol("prediction")
  .setMetricName("accuracy")
val accuracy = evaluator.evaluate(predictions)
println(s"Test Error = ${(1.0 - accuracy)}")

val rfModel = model.stages(2).asInstanceOf[RandomForestClassificationModel]
println(s"Learned classification forest model:\n ${rfModel.toDebugString}")
```

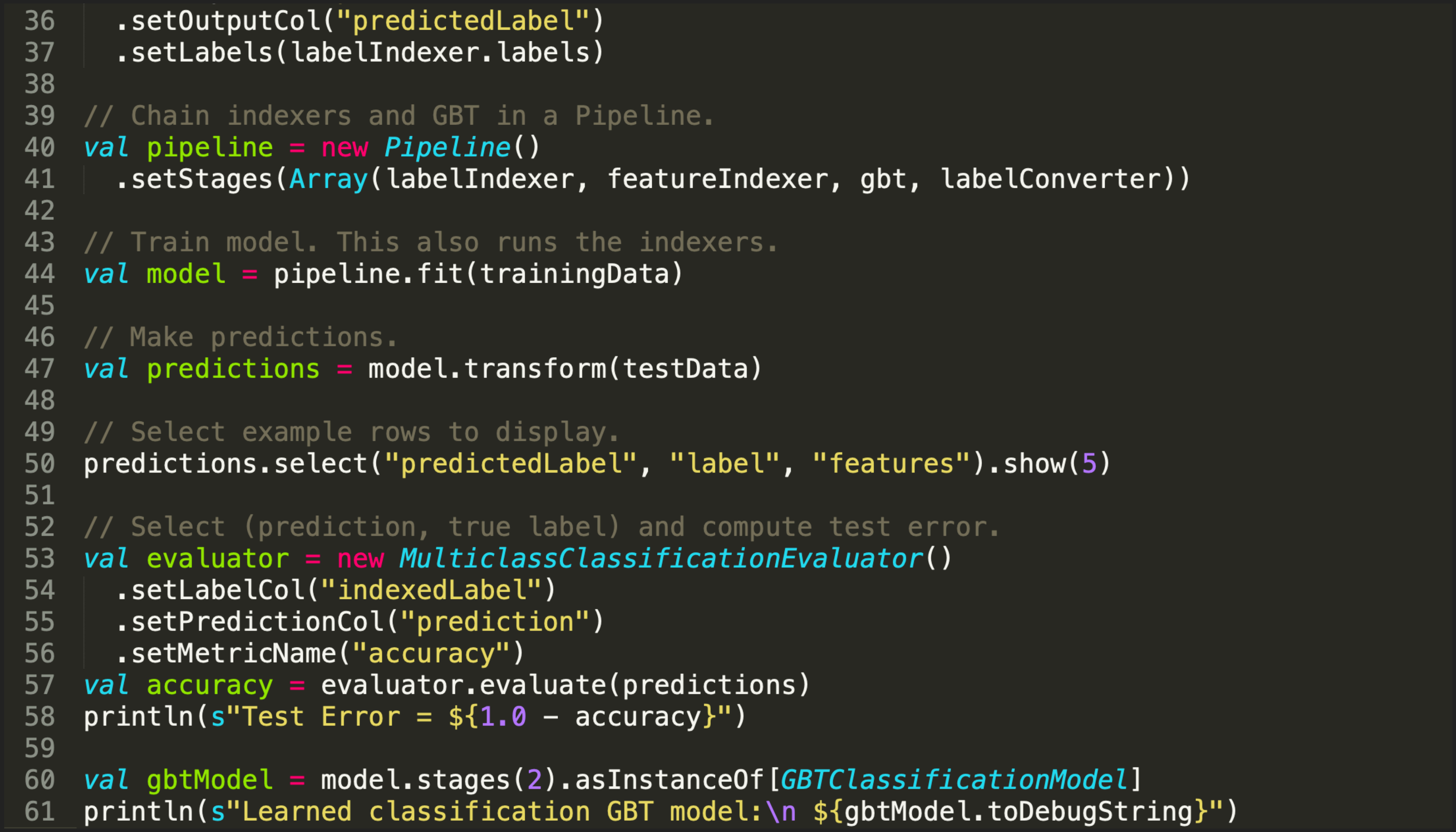
GRADIENT-BOOSTED TREE CLASSIFIER

- Ensembles of decision trees.
- GBTs iteratively train decision trees in order to minimize a loss function.
- The spark.ml implementation supports GBTs for binary classification and for regression, using both continuous and categorical features.

GRADIENT-BOOSTED TREE CLASSIFIER

```scala
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.{GBTClassificationModel, GBTClassifier}
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorIndexer}

// Load and parse the data file, converting it to a DataFrame.
val data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

// Index labels, adding metadata to the label column.
// Fit on whole dataset to include all labels in index.
val labelIndexer = new StringIndexer()
  .setInputCol("label")
  .setOutputCol("indexedLabel")
  .fit(data)
// Automatically identify categorical features, and index them.
// Set maxCategories so features with > 4 distinct values are treated as continuous.
val featureIndexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexedFeatures")
  .setMaxCategories(4)
  .fit(data)

// Split the data into training and test sets (30% held out for testing).
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))

// Train a GBT model.
val gbt = new GBTClassifier()
  .setLabelCol("indexedLabel")
  .setFeaturesCol("indexedFeatures")
  .setMaxIter(10)
  .setFeatureSubsetStrategy("auto")

// Convert indexed labels back to original labels.
val labelConverter = new IndexToString()
  .setInputCol("prediction")
```

GRADIENT-BOOSTED TREE CLASSIFIER

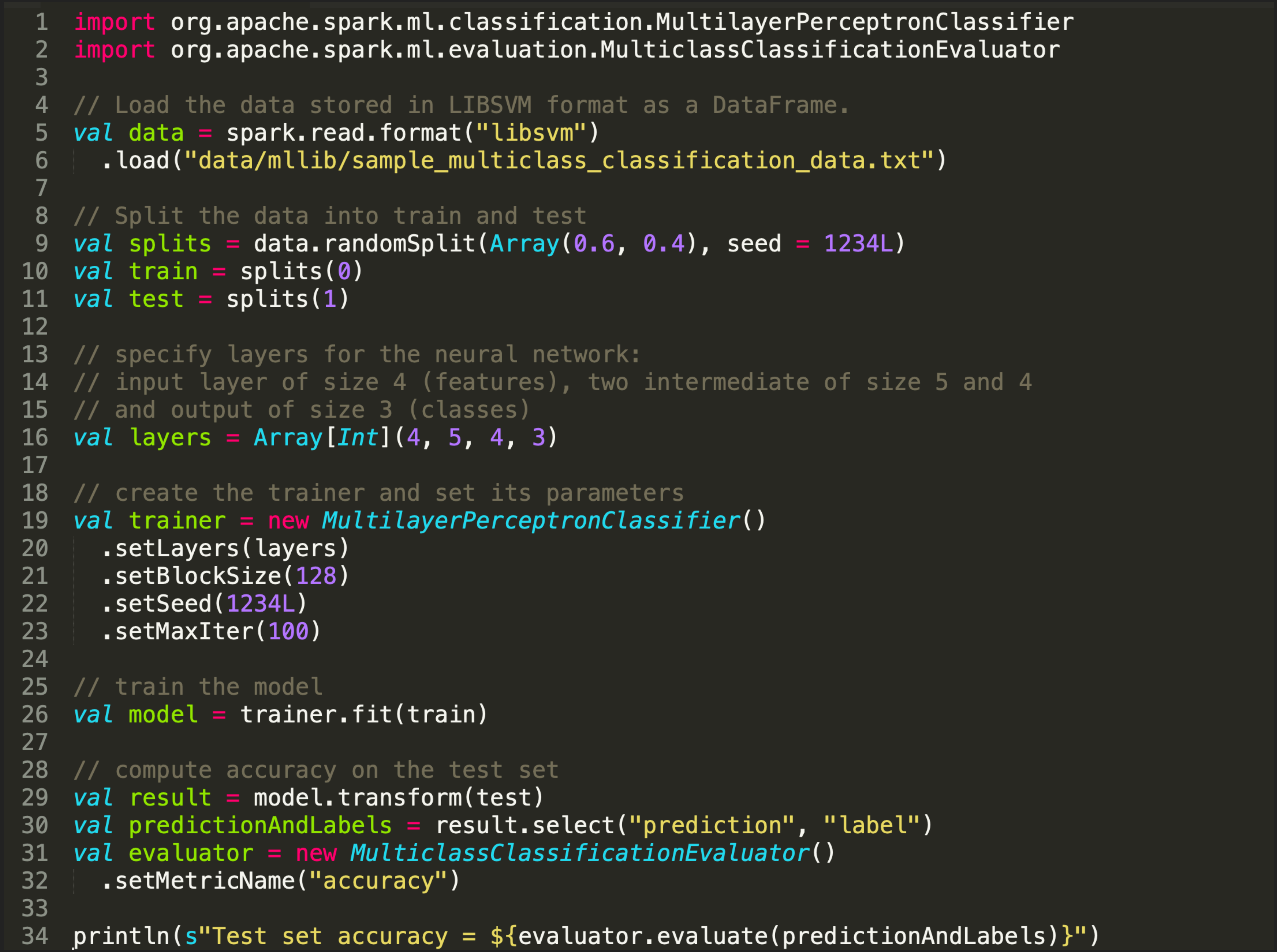


```scala
  .setOutputCol("predictedLabel")
  .setLabels(labelIndexer.labels)

// Chain indexers and GBT in a Pipeline.
val pipeline = new Pipeline()
  .setStages(Array(labelIndexer, featureIndexer, gbt, labelConverter))

// Train model. This also runs the indexers.
val model = pipeline.fit(trainingData)

// Make predictions.
val predictions = model.transform(testData)

// Select example rows to display.
predictions.select("predictedLabel", "label", "features").show(5)

// Select (prediction, true label) and compute test error.
val evaluator = new MulticlassClassificationEvaluator()
  .setLabelCol("indexedLabel")
  .setPredictionCol("prediction")
  .setMetricName("accuracy")
val accuracy = evaluator.evaluate(predictions)
println(s"Test Error = ${1.0 - accuracy}")

val gbtModel = model.stages(2).asInstanceOf[GBTClassificationModel]
println(s"Learned classification GBT model:\n ${gbtModel.toDebugString}")
```

MULTILAYER PERCEPTRON CLASSIFIER

- A classifier based on the feedforward artificial neural network.
- MLPC consists of multiple layers of nodes. Each layer is fully connected to the next layer in the network. Nodes in the input layer represent the input data. All other nodes map inputs to outputs by a linear combination of the inputs with the node's weights w and bias b and applying an activation function.
- MLPC employs backpropagation for learning the model. We use the logistic loss function for optimization and L-BFGS as an optimization routine.

MULTILAYER PERCEPTRON CLASSIFIER



```scala
import org.apache.spark.ml.classification.MultilayerPerceptronClassifier
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

// Load the data stored in LIBSVM format as a DataFrame.
val data = spark.read.format("libsvm")
  .load("data/mllib/sample_multiclass_classification_data.txt")

// Split the data into train and test
val splits = data.randomSplit(Array(0.6, 0.4), seed = 1234L)
val train = splits(0)
val test = splits(1)

// specify layers for the neural network:
// input layer of size 4 (features), two intermediate of size 5 and 4
// and output of size 3 (classes)
val layers = Array[Int](4, 5, 4, 3)

// create the trainer and set its parameters
val trainer = new MultilayerPerceptronClassifier()
  .setLayers(layers)
  .setBlockSize(128)
  .setSeed(1234L)
  .setMaxIter(100)

// train the model
val model = trainer.fit(train)

// compute accuracy on the test set
val result = model.transform(test)
val predictionAndLabels = result.select("prediction", "label")
val evaluator = new MulticlassClassificationEvaluator()
  .setMetricName("accuracy")

println(s"Test set accuracy = ${evaluator.evaluate(predictionAndLabels)}")
```

LINEAR SUPPORT VECTOR MACHINE

- A support vector machine constructs a hyperplane or set of hyperplanes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks.

- Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training-data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.

- LinearSVC in Spark ML supports binary classification with linear SVM. Internally, it optimizes the Hinge Loss using OWLQN optimizer.

LINEAR SUPPORT VECTOR MACHINE

```scala
import org.apache.spark.ml.classification.LinearSVC

// Load training data
val training = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

val lsvc = new LinearSVC()
  .setMaxIter(10)
  .setRegParam(0.1)

// Fit the model
val lsvcModel = lsvc.fit(training)

// Print the coefficients and intercept for linear svc
println(s"Coefficients: ${lsvcModel.coefficients} Intercept: ${lsvcModel.intercept}")
```

ONE-VS-REST CLASSIFIER

- OneVsRest is an example of a machine learning reduction for performing multiclass classification given a base classifier that can perform binary classification efficiently.
  - It is also known as "One-vs-All."
- OneVsRest is implemented as an Estimator. For the base classifier, it takes instances of Classifier and creates a binary classification problem for each of the k classes. The classifier for class i is trained to predict whether the label is i or not, distinguishing class i from all other classes.
- Predictions are done by evaluating each binary classifier and the index of the most confident classifier is output as label.

ONE-VS-REST CLASSIFIER

```scala
import org.apache.spark.ml.classification.{LogisticRegression, OneVsRest}
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

// load data file.
val inputData = spark.read.format("libsvm")
  .load("data/mllib/sample_multiclass_classification_data.txt")

// generate the train/test split.
val Array(train, test) = inputData.randomSplit(Array(0.8, 0.2))

// instantiate the base classifier
val classifier = new LogisticRegression()
  .setMaxIter(10)
  .setTol(1E-6)
  .setFitIntercept(true)

// instantiate the One Vs Rest Classifier.
val ovr = new OneVsRest().setClassifier(classifier)

// train the multiclass model.
val ovrModel = ovr.fit(train)

// score the model on test data.
val predictions = ovrModel.transform(test)

// obtain evaluator.
val evaluator = new MulticlassClassificationEvaluator()
  .setMetricName("accuracy")

// compute the classification error on test data.
val accuracy = evaluator.evaluate(predictions)
println(s"Test Error = ${1 - accuracy}")
```

NAIVE BAYES

- Naive Bayes classifiers are a family of simple probabilistic, multiclass classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between every pair of features.

- Naive Bayes can be trained very efficiently. With a single pass over the training data, it computes the conditional probability distribution of each feature given each label. For prediction, it applies Bayes' theorem to compute the conditional probability distribution of each label given an observation.

- MLlib supports both multinomial naive Bayes and Bernoulli naive Bayes.

NAIVE BAYES

```scala
import org.apache.spark.ml.classification.NaiveBayes
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

// Load the data stored in LIBSVM format as a DataFrame.
val data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

// Split the data into training and test sets (30% held out for testing)
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3), seed = 1234L)

// Train a NaiveBayes model.
val model = new NaiveBayes()
  .fit(trainingData)

// Select example rows to display.
val predictions = model.transform(testData)
predictions.show()

// Select (prediction, true label) and compute test error
val evaluator = new MulticlassClassificationEvaluator()
  .setLabelCol("label")
  .setPredictionCol("prediction")
  .setMetricName("accuracy")
val accuracy = evaluator.evaluate(predictions)
println(s"Test set accuracy = $accuracy")
```

THX!