Classification and regression

This page covers algorithms for Classification and Regression. It also includes sections discussing specific classes of algorithms, such as linear methods, trees, and ensembles.

Table of Contents

Classification

Logistic regression

Logistic regression is a popular method to predict a categorical response. It is a special case of Generalized Linear models that predicts the probability of the outcomes. In `spark.ml` logistic regression can be used to predict a binary outcome by using binomial logistic regression, or it can be used to predict a multiclass outcome by using multinomial logistic regression. Use the `family` parameter to select between these two algorithms, or leave it unset and Spark will infer the correct variant.

> Multinomial logistic regression can be used for binary classification by setting the `family` param to "multinomial". It will produce two sets of coefficients and two intercepts.

»

> When fitting LogisticRegressionModel without intercept on dataset with constant nonzero column, Spark MLlib outputs zero coefficients for constant nonzero columns. This behavior is the same as R glmnet but different from LIBSVM.

Binomial logistic regression

For more background and more details about the implementation of binomial logistic regression, refer to the documentation of logistic regression in `spark.mllib`.

Examples

The following example shows how to train binomial and multinomial logistic regression models for binary classification with elastic net regularization. `elasticNetParam` corresponds to α and `regParam` corresponds to λ.

Scala **Java** **Python** **R**

More details on parameters can be found in the Python API documentation.

```python
from pyspark.ml.classification import LogisticRegression

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

lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Fit the model
lrModel = lr.fit(training)

# Print the coefficients and intercept for logistic regression
print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))

# We can also use the multinomial family for binary classification
mlr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8, family="multinomial")

# Fit the model
mlrModel = mlr.fit(training)

# Print the coefficients and intercepts for logistic regression with multinomial family
print("Multinomial coefficients: " + str(mlrModel.coefficientMatrix))
print("Multinomial intercepts: " + str(mlrModel.interceptVector))
```

Find full example code at "examples/src/main/python/ml/logistic_regression_with_elastic_net.py" in the Spark repo.

The `spark.ml` implementation of logistic regression also supports extracting a summary of the model over the training set. Note that the predictions and metrics which are stored as `DataFrame` in `BinaryLogisticRegressionSummary` are annotated `@transient` and hence only available on the driver.

Scala Java **Python**

»

`LogisticRegressionTrainingSummary` provides a summary for a `LogisticRegressionModel`. Currently, only binary classification is supported. Support for multiclass model summaries will be added in the future.

Continuing the earlier example:

```
from pyspark.ml.classification import LogisticRegression

# Extract the summary from the returned LogisticRegressionModel instance trained
# in the earlier example
trainingSummary = lrModel.summary

# Obtain the objective per iteration
objectiveHistory = trainingSummary.objectiveHistory
print("objectiveHistory:")
for objective in objectiveHistory:
    print(objective)

# Obtain the receiver-operating characteristic as a dataframe and areaUnderROC.
trainingSummary.roc.show()
print("areaUnderROC: " + str(trainingSummary.areaUnderROC))

# Set the model threshold to maximize F-Measure
fMeasure = trainingSummary.fMeasureByThreshold
maxFMeasure = fMeasure.groupBy().max('F-Measure').select('max(F-Measure)').head()
bestThreshold = fMeasure.where(fMeasure['F-Measure'] == maxFMeasure['max(F-Measure)']) \
    .select('threshold').head()['threshold']
lr.setThreshold(bestThreshold)
```

Find full example code at "examples/src/main/python/ml/logistic_regression_summary_example.py" in the Spark repo.

Multinomial logistic regression

Multiclass classification is supported via multinomial logistic (softmax) regression. In multinomial logistic regression, the algorithm produces K sets of coefficients, or a matrix of dimension $K \times J$ where K is the number of outcome classes and J is the number of features. If the algorithm is fit with an intercept term then a length K vector of intercepts is available.

> Multinomial coefficients are available as `coefficientMatrix` and intercepts are available as `interceptVector`.

> `coefficients` and `intercept` methods on a logistic regression model trained with multinomial family are not supported. Use `coefficientMatrix` and `interceptVector` instead.

The conditional probabilities of the outcome classes $k \in 1, 2, \ldots, K$ are modeled using the softmax function.

$$P(Y = k|\mathbf{X}, \boldsymbol{\beta}_k, \beta_{0k}) = \frac{e^{\boldsymbol{\beta}_k \cdot \mathbf{X} + \beta_{0k}}}{\sum_{k'=0}^{K-1} e^{\boldsymbol{\beta}_{k'} \cdot \mathbf{X} + \beta_{0k'}}}$$

We minimize the weighted negative log-likelihood, using a multinomial response model, with elastic-net penalty to control for overfitting.

$$\min_{\beta, \beta_0} - \left[\sum_{i=1}^{L} w_i \cdot \log P(Y = y_i|\mathbf{x}_i)\right] + \lambda \left[\frac{1}{2}(1 - \alpha)\, ||\boldsymbol{\beta}||_2^2 + \alpha ||\boldsymbol{\beta}||_1\right]$$

For a detailed derivation please see here.

Examples

The following example shows how to train a multiclass logistic regression model with elastic net regularization.

Scala Java **Python** R

```python
from pyspark.ml.classification import LogisticRegression

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

lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Fit the model
lrModel = lr.fit(training)

# Print the coefficients and intercept for multinomial logistic regression
print("Coefficients: \n" + str(lrModel.coefficientMatrix))
print("Intercept: " + str(lrModel.interceptVector))
```

Find full example code at "examples/src/main/python/ml/multiclass_logistic_regression_with_elastic_net.py" in the Spark repo.

Decision tree classifier

Decision trees are a popular family of classification and regression methods. More information about the `spark.ml` implementation can be found further in the section on decision trees.

Examples

The following examples load a dataset in LibSVM format, split it into training and test sets, train on the first dataset, and then evaluate on the held-out test set. We use two feature transformers to prepare the data; these help index categories for the label and categorical features, adding metadata to the `DataFrame` which the Decision Tree algorithm can recognize.

Scala Java **Python**

More details on parameters can be found in the Python API documentation.

»

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.feature import StringIndexer, VectorIndexer
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Load the data stored in LIBSVM format as a DataFrame.
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.
labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(data)
# Automatically identify categorical features, and index them.
# We specify maxCategories so features with > 4 distinct values are treated as continuous.
featureIndexer =\
    VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data)

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

# Train a DecisionTree model.
dt = DecisionTreeClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures")

# Chain indexers and tree in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, dt])

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

# Make predictions.
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("prediction", "indexedLabel", "features").show(5)

# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g " % (1.0 - accuracy))

treeModel = model.stages[2]
# summary only
print(treeModel)
```

Find full example code at "examples/src/main/python/ml/decision_tree_classification_example.py" in the Spark repo.

Random forest classifier

Random forests are a popular family of classification and regression methods. More information about the `spark.ml` implementation can be found further in the section on random forests.

Examples

The following examples load a dataset in LibSVM format, split it into training and test sets, train on the first dataset, and then evaluate on the held-out test set. We use two feature transformers to prepare the data; these help index categories for the label and categorical features, adding metadata to the `DataFrame` which the tree-based algorithms can recognize.

Scala **Java** **Python** **R**

»

Refer to the Python API docs for more details.

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.feature import IndexToString, StringIndexer, VectorIndexer
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Load and parse the data file, converting it to a DataFrame.
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.
labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(data)

# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as continuous.
featureIndexer =\
    VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data)

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

# Train a RandomForest model.
rf = RandomForestClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures", numTrees=10)

# Convert indexed labels back to original labels.
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel",
                               labels=labelIndexer.labels)

# Chain indexers and forest in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, rf, labelConverter])

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

# Make predictions.
predictions = model.transform(testData)

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

# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
```

```
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g" % (1.0 - accuracy))

rfModel = model.stages[2]
print(rfModel)  # summary only
```

Find full example code at "examples/src/main/python/ml/random_forest_classifier_example.py" in the Spark repo.

Gradient-boosted tree classifier

Gradient-boosted trees (GBTs) are a popular classification and regression method using ensembles of decision trees. More information about the `spark.ml` implementation can be found further in the section on GBTs.

Examples

The following examples load a dataset in LibSVM format, split it into training and test sets, train on the first dataset, and then evaluate on the held-out test set. We use two feature transformers to prepare the data; these help index categories for the label and categorical features, adding metadata to the `DataFrame` which the tree-based algorithms can recognize.

Scala Java **Python** R

Refer to the Python API docs for more details.

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import GBTClassifier
from pyspark.ml.feature import StringIndexer, VectorIndexer
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Load and parse the data file, converting it to a DataFrame.
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.
labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(data)
# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as continuous.
featureIndexer =\
    VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data)

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

# Train a GBT model.
gbt = GBTClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures", maxIter=10)

# Chain indexers and GBT in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, gbt])

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

# Make predictions.
```

```
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("prediction", "indexedLabel", "features").show(5)

# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g" % (1.0 - accuracy))

gbtModel = model.stages[2]
print(gbtModel)  # summary only
```

Find full example code at "examples/src/main/python/ml/gradient_boosted_tree_classifier_example.py" in the Spark repo.

Multilayer perceptron classifier

Multilayer perceptron classifier (MLPC) is 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 $\mathbf{w}$ and bias $\mathbf{b}$ and applying an activation function. This can be written in matrix form for MLPC with $K+1$ layers as follows:

$$\mathrm{y}(\mathbf{x}) = \mathrm{f_K}(\ldots \mathrm{f_2}(\mathbf{w}_2^T \mathrm{f_1}(\mathbf{w}_1^T \mathbf{x} + b_1) + b_2) \ldots + b_K)$$

Nodes in intermediate layers use sigmoid (logistic) function:

$$\mathrm{f}(z_i) = \frac{1}{1 + e^{-z_i}}$$

Nodes in the output layer use softmax function:

$$\mathrm{f}(z_i) = \frac{e^{z_i}}{\sum_{k=1}^N e^{z_k}}$$

The number of nodes N in the output layer corresponds to the number of classes.

MLPC employs backpropagation for learning the model. We use the logistic loss function for optimization and L-BFGS as an optimization routine.

Examples

Scala **Java** **Python** **R**

Refer to the Python API docs for more details.

```
from pyspark.ml.classification import MultilayerPerceptronClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

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

```
# Split the data into train and test
splits = data.randomSplit([0.6, 0.4], 1234)
train = splits[0]
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)
layers = [4, 5, 4, 3]

# create the trainer and set its parameters
trainer = MultilayerPerceptronClassifier(maxIter=100, layers=layers, blockSize=128, seed=1234)

# train the model
model = trainer.fit(train)

# compute accuracy on the test set
result = model.transform(test)
predictionAndLabels = result.select("prediction", "label")
evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print("Test set accuracy = " + str(evaluator.evaluate(predictionAndLabels)))
```

Find full example code at "examples/src/main/python/ml/multilayer_perceptron_classification.py" in the Spark repo.

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.

Examples

Scala Java **Python** R

Refer to the Python API docs for more details.

```
from pyspark.ml.classification import LinearSVC

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

lsvc = LinearSVC(maxIter=10, regParam=0.1)

# Fit the model
lsvcModel = lsvc.fit(training)

# Print the coefficients and intercept for linearsSVC
print("Coefficients: " + str(lsvcModel.coefficients))
print("Intercept: " + str(lsvcModel.intercept))
```

Find full example code at "examples/src/main/python/ml/linearsvc.py" in the Spark repo.

One-vs-Rest classifier (a.k.a. One-vs-All)

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.

Examples

The example below demonstrates how to load the Iris dataset, parse it as a DataFrame and perform multiclass classification using `OneVsRest`. The test error is calculated to measure the algorithm accuracy.

Scala Java **Python**

Refer to the Python API docs for more details.

```python
from pyspark.ml.classification import LogisticRegression, OneVsRest
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

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

# generate the train/test split.
(train, test) = inputData.randomSplit([0.8, 0.2])

# instantiate the base classifier.
lr = LogisticRegression(maxIter=10, tol=1E-6, fitIntercept=True)

# instantiate the One Vs Rest Classifier.
ovr = OneVsRest(classifier=lr)

# train the multiclass model.
ovrModel = ovr.fit(train)

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

# obtain evaluator.
evaluator = MulticlassClassificationEvaluator(metricName="accuracy")

# compute the classification error on test data.
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g" % (1.0 - accuracy))
```

Find full example code at "examples/src/main/python/ml/one_vs_rest_example.py" in the Spark repo.

Naive Bayes

Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features. The `spark.ml` implementation currently supports both multinomial naive Bayes and Bernoulli naive Bayes. More information can be found in the section on Naive Bayes in MLlib.

Examples

Scala Java **Python** R

Refer to the Python API docs for more details.

```python
from pyspark.ml.classification import NaiveBayes
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

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

# Split the data into train and test
splits = data.randomSplit([0.6, 0.4], 1234)
train = splits[0]
test = splits[1]

# create the trainer and set its parameters
nb = NaiveBayes(smoothing=1.0, modelType="multinomial")

# train the model
model = nb.fit(train)

# select example rows to display.
predictions = model.transform(test)
predictions.show()

# compute accuracy on the test set
evaluator = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction",
                                              metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test set accuracy = " + str(accuracy))
```

Find full example code at "examples/src/main/python/ml/naive_bayes_example.py" in the Spark repo.

Regression

Linear regression

The interface for working with linear regression models and model summaries is similar to the logistic regression case.

> When fitting LinearRegressionModel without intercept on dataset with constant nonzero column by “l-bfgs” solver, Spark MLlib outputs zero coefficients for constant nonzero columns. This behavior is the same as R glmnet but different from LIBSVM.

»

Examples

The following example demonstrates training an elastic net regularized linear regression model and extracting model summary statistics.

Scala Java **Python**

More details on parameters can be found in the Python API documentation.

```
from pyspark.ml.regression import LinearRegression

# Load training data
training = spark.read.format("libsvm")\
    .load("data/mllib/sample_linear_regression_data.txt")

lr = LinearRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Fit the model
lrModel = lr.fit(training)

# Print the coefficients and intercept for linear regression
print("Coefficients: %s" % str(lrModel.coefficients))
print("Intercept: %s" % str(lrModel.intercept))

# Summarize the model over the training set and print out some metrics
trainingSummary = lrModel.summary
print("numIterations: %d" % trainingSummary.totalIterations)
print("objectiveHistory: %s" % str(trainingSummary.objectiveHistory))
trainingSummary.residuals.show()
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
print("r2: %f" % trainingSummary.r2)
```

Find full example code at "examples/src/main/python/ml/linear_regression_with_elastic_net.py" in the Spark repo.

Generalized linear regression

Contrasted with linear regression where the output is assumed to follow a Gaussian distribution, generalized linear models (GLMs) are specifications of linear models where the response variable Y_i follows some distribution from the exponential family of distributions. Spark’s `GeneralizedLinearRegression` interface allows for flexible specification of GLMs which can be used for various types of prediction problems including linear regression, Poisson regression, logistic regression, and others. Currently in `spark.ml`, only a subset of the exponential family distributions are supported and they are listed below.

NOTE: Spark currently only supports up to 4096 features through its `GeneralizedLinearRegression` interface, and will throw an exception if this constraint is exceeded. See the advanced section for more details. Still, for linear and logistic regression, models with an increased number of features can be trained using the `LinearRegression` and `LogisticRegression` estimators.

GLMs require exponential family distributions that can be written in their "canonical" or "natural" form, aka natural exponential family distributions. The form of a natural exponential family distribution is given as:

$$f_Y(y|\theta, \tau) = h(y, \tau) \exp\left(\frac{\theta \cdot y - A(\theta)}{d(\tau)}\right)$$

where θ is the parameter of interest and τ is a dispersion parameter. In a GLM the response variable Y_i is assumed to be drawn from a natural exponential family distribution:

$$Y_i \sim f(\cdot|\theta_i, \tau)$$

where the parameter of interest θ_i is related to the expected value of the response variable μ_i by

$$\mu_i = A'(\theta_i)$$

Here, $A'(\theta_i)$ is defined by the form of the distribution selected. GLMs also allow specification of a link function, which defines the relationship between the expected value of the response variable μ_i and the so called *linear predictor* η_i:

$$g(\mu_i) = \eta_i = \vec{x_i}^T \cdot \vec{\beta}$$

Often, the link function is chosen such that $A' = g^{-1}$, which yields a simplified relationship between the parameter of interest θ and the linear predictor η. In this case, the link function $g(\mu)$ is said to be the "canonical" link function.

$$\theta_i = A'^{-1}(\mu_i) = g(g^{-1}(\eta_i)) = \eta_i$$

A GLM finds the regression coefficients $\vec{\beta}$ which maximize the likelihood function.

$$\max_{\vec{\beta}} \mathcal{L}(\vec{\theta}|\vec{y}, X) = \prod_{i=1}^{N} h(y_i, \tau) \exp\left(\frac{y_i\theta_i - A(\theta_i)}{d(\tau)}\right)$$

where the parameter of interest θ_i is related to the regression coefficients $\vec{\beta}$ by

$$\theta_i = A'^{-1}(g^{-1}(\vec{x_i} \cdot \vec{\beta}))$$

Spark's generalized linear regression interface also provides summary statistics for diagnosing the fit of GLM models, including residuals, p-values, deviances, the Akaike information criterion, and others.

See here for a more comprehensive review of GLMs and their applications.

Available families

Family	Response Type	Supported Links
Gaussian	Continuous	Identity*, Log, Inverse
Binomial	Binary	Logit*, Probit, CLogLog
Poisson	Count	Log*, Identity, Sqrt

* Canonical Link

Family	Response Type	Supported Links
Gamma	Continuous	Inverse*, Idenity, Log
Tweedie	Zero-inflated continuous	Power link function

* Canonical Link

Examples

The following example demonstrates training a GLM with a Gaussian response and identity link function and extracting model summary statistics.

Scala Java **Python** R

Refer to the Python API docs for more details.

```python
from pyspark.ml.regression import GeneralizedLinearRegression

# Load training data
dataset = spark.read.format("libsvm")\
    .load("data/mllib/sample_linear_regression_data.txt")

glr = GeneralizedLinearRegression(family="gaussian", link="identity", maxIter=10, regParam=0.3)

# Fit the model
model = glr.fit(dataset)

# Print the coefficients and intercept for generalized linear regression model
print("Coefficients: " + str(model.coefficients))
print("Intercept: " + str(model.intercept))

# Summarize the model over the training set and print out some metrics
summary = model.summary
print("Coefficient Standard Errors: " + str(summary.coefficientStandardErrors))
print("T Values: " + str(summary.tValues))
print("P Values: " + str(summary.pValues))
print("Dispersion: " + str(summary.dispersion))
print("Null Deviance: " + str(summary.nullDeviance))
print("Residual Degree Of Freedom Null: " + str(summary.residualDegreeOfFreedomNull))
print("Deviance: " + str(summary.deviance))
print("Residual Degree Of Freedom: " + str(summary.residualDegreeOfFreedom))
print("AIC: " + str(summary.aic))
print("Deviance Residuals: ")
summary.residuals().show()
```

Find full example code at "examples/src/main/python/ml/generalized_linear_regression_example.py" in the Spark repo.

Decision tree regression

Decision trees are a popular family of classification and regression methods. More information about the `spark.ml` implementation can be found further in the section on decision trees.

Examples

The following examples load a dataset in LibSVM format, split it into training and test sets, train on the first dataset, and then evaluate on the held-out test set. We use a feature transformer to index categorical features, adding metadata to the `DataFrame` which the Decision Tree algorithm can recognize.

»

Scala Java **Python**

More details on parameters can be found in the Python API documentation.

```
from pyspark.ml import Pipeline
from pyspark.ml.regression import DecisionTreeRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

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

# Automatically identify categorical features, and index them.
# We specify maxCategories so features with > 4 distinct values are treated as continuous.
featureIndexer =\
    VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data)

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

# Train a DecisionTree model.
dt = DecisionTreeRegressor(featuresCol="indexedFeatures")

# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, dt])

# Train model.  This also runs the indexer.
model = pipeline.fit(trainingData)

# Make predictions.
predictions = model.transform(testData)

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

# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)

treeModel = model.stages[1]
```

```
# summary only
print(treeModel)
```

Find full example code at "examples/src/main/python/ml/decision_tree_regression_example.py" in the Spark repo.

Random forest regression

Random forests are a popular family of classification and regression methods. More information about the `spark.ml` implementation can be found further in the section on random forests.

Examples

The following examples load a dataset in LibSVM format, split it into training and test sets, train on the first dataset, and then evaluate on the held-out test set. We use a feature transformer to index categorical features, adding metadata to the `DataFrame` which the tree-based algorithms can recognize.

Scala Java **Python** R

Refer to the Python API docs for more details.

```
from pyspark.ml import Pipeline
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

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

# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as continuous.
featureIndexer =\
    VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data)

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

# Train a RandomForest model.
rf = RandomForestRegressor(featuresCol="indexedFeatures")

# Chain indexer and forest in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, rf])

# Train model.  This also runs the indexer.
model = pipeline.fit(trainingData)

# Make predictions.
predictions = model.transform(testData)

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

# Select (prediction, true label) and compute test error
```

```
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)

rfModel = model.stages[1]
print(rfModel)  # summary only
```

Find full example code at "examples/src/main/python/ml/random_forest_regressor_example.py" in the Spark repo.

Gradient-boosted tree regression

Gradient-boosted trees (GBTs) are a popular regression method using ensembles of decision trees. More information about the `spark.ml` implementation can be found further in the section on GBTs.

Examples

Note: For this example dataset, `GBTRegressor` actually only needs 1 iteration, but that will not be true in general.

Scala Java **Python** R

Refer to the Python API docs for more details.

```
from pyspark.ml import Pipeline
from pyspark.ml.regression import GBTRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

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

# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as continuous.
featureIndexer =\
    VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data)

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

# Train a GBT model.
gbt = GBTRegressor(featuresCol="indexedFeatures", maxIter=10)

# Chain indexer and GBT in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, gbt])

# Train model.  This also runs the indexer.
model = pipeline.fit(trainingData)

# Make predictions.
predictions = model.transform(testData)

# Select example rows to display.
```

```
predictions.select("prediction", "label", "features").show(5)

# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)

gbtModel = model.stages[1]
print(gbtModel)  # summary only
```

Find full example code at "examples/src/main/python/ml/gradient_boosted_tree_regressor_example.py" in the Spark repo.

Survival regression

In `spark.ml`, we implement the Accelerated failure time (AFT) model which is a parametric survival regression model for censored data. It describes a model for the log of survival time, so it's often called a log-linear model for survival analysis. Different from a Proportional hazards model designed for the same purpose, the AFT model is easier to parallelize because each instance contributes to the objective function independently.

Given the values of the covariates $x^{`}$, for random lifetime t_i of subjects i = 1, …, n, with possible right-censoring, the likelihood function under the AFT model is given as:

$$L(\beta,\sigma)=\prod_{i=1}^n[\frac{1}{\sigma}f_0(\frac{\log t_i-x^{'}\beta}{\sigma})]^{\delta_i}S_0(\frac{\log t_i-x^{'}\beta}{\sigma})^{1-\delta_i}$$

Where δ_i is the indicator of the event has occurred i.e. uncensored or not. Using $\epsilon_i=\frac{\log t_i-x^{`}\beta}{\sigma}$, the log-likelihood function assumes the form:

$$\iota(\beta,\sigma)=\sum_{i=1}^{n}[-\delta_i\log\sigma+\delta_i\log f_0(\epsilon_i)+(1-\delta_i)\log S_0(\epsilon_i)]$$

Where $S_0(\epsilon_i)$ is the baseline survivor function, and $f_0(\epsilon_i)$ is the corresponding density function.

The most commonly used AFT model is based on the Weibull distribution of the survival time. The Weibull distribution for lifetime corresponds to the extreme value distribution for the log of the lifetime, and the $S_0(\epsilon)$ function is:

$$S_0(\epsilon_i)=\exp(-e^{\epsilon_i})$$

the $f_0(\epsilon_i)$ function is:

$$f_0(\epsilon_i)=e^{\epsilon_i}\exp(-e^{\epsilon_i})$$

The log-likelihood function for AFT model with a Weibull distribution of lifetime is:

$$\iota(\beta,\sigma)=-\sum_{i=1}^n[\delta_i\log\sigma-\delta_i\epsilon_i+e^{\epsilon_i}]$$

Due to minimizing the negative log-likelihood equivalent to maximum a posteriori probability, the loss function we use to optimize is $-\iota(\beta,\sigma)$. The gradient functions for β and $\log\sigma$ respectively are:

$$\frac{\partial(-\iota)}{\partial\beta}=\sum_{1=1}^{n}[\delta_i-e^{\epsilon_i}]\frac{x_i}{\sigma}$$

$$\frac{\partial(-\iota)}{\partial(\log\sigma)} = \sum_{i=1}^{n}[\delta_i + (\delta_i - e^{\epsilon_i})\epsilon_i]$$

The AFT model can be formulated as a convex optimization problem, i.e. the task of finding a minimizer of a convex function $-\iota(\beta, \sigma)$ that depends on the coefficients vector β and the log of scale parameter $\log\sigma$. The optimization algorithm underlying the implementation is L-BFGS. The implementation matches the result from R's survival function survreg

> When fitting AFTSurvivalRegressionModel without intercept on dataset with constant nonzero column, Spark MLlib outputs zero coefficients for constant nonzero columns. This behavior is different from R survival::survreg.

Examples

Scala Java **Python** R

Refer to the Python API docs for more details.

```python
from pyspark.ml.regression import AFTSurvivalRegression
from pyspark.ml.linalg import Vectors

training = spark.createDataFrame([
    (1.218, 1.0, Vectors.dense(1.560, -0.605)),
    (2.949, 0.0, Vectors.dense(0.346, 2.158)),
    (3.627, 0.0, Vectors.dense(1.380, 0.231)),
    (0.273, 1.0, Vectors.dense(0.520, 1.151)),
    (4.199, 0.0, Vectors.dense(0.795, -0.226))], ["label", "censor", "features"])
quantileProbabilities = [0.3, 0.6]
aft = AFTSurvivalRegression(quantileProbabilities=quantileProbabilities,
                            quantilesCol="quantiles")

model = aft.fit(training)

# Print the coefficients, intercept and scale parameter for AFT survival regression
print("Coefficients: " + str(model.coefficients))
print("Intercept: " + str(model.intercept))
print("Scale: " + str(model.scale))
model.transform(training).show(truncate=False)
```

Find full example code at "examples/src/main/python/ml/aft_survival_regression.py" in the Spark repo.

Isotonic regression

Isotonic regression belongs to the family of regression algorithms. Formally isotonic regression is a problem where given a finite set of real numbers $Y = y_1, y_2, ..., y_n$ representing observed responses and $X = x_1, x_2, ..., x_n$ the unknown response values to be fitted finding a function that minimises

$$f(x) = \sum_{i=1}^{n} w_i(y_i - x_i)^2 \qquad (1)$$

with respect to complete order subject to $x_1 \le x_2 \le ... \le x_n$ where w_i are positive weights. The resulting function is called isotonic regression and it is unique. It can be viewed as least squares problem under order restriction. Essentially isotonic regression is a monotonic function best fitting the original data points.

We implement a pool adjacent violators algorithm which uses an approach to parallelizing isotonic regression. The training input is a DataFrame which contains three columns label, features and weight. Additionally IsotonicRegression algorithm has one optional parameter called $isotonic$ defaulting to true. This argument specifies if the isotonic regression is isotonic (monotonically increasing) or antitonic (monotonically decreasing).

»

Training returns an IsotonicRegressionModel that can be used to predict labels for both known and unknown features. The result of isotonic regression is treated as piecewise linear function. The rules for prediction therefore are:

- If the prediction input exactly matches a training feature then associated prediction is returned. In case there are multiple predictions with the same feature then one of them is returned. Which one is undefined (same as java.util.Arrays.binarySearch).
- If the prediction input is lower or higher than all training features then prediction with lowest or highest feature is returned respectively. In case there are multiple predictions with the same feature then the lowest or highest is returned respectively.
- If the prediction input falls between two training features then prediction is treated as piecewise linear function and interpolated value is calculated from the predictions of the two closest features. In case there are multiple values with the same feature then the same rules as in previous point are used.

Examples

Scala **Java** **Python** **R**

Refer to the `IsotonicRegression` Python docs for more details on the API.

```
from pyspark.ml.regression import IsotonicRegression

# Loads data.
dataset = spark.read.format("libsvm")\
    .load("data/mllib/sample_isotonic_regression_libsvm_data.txt")

# Trains an isotonic regression model.
model = IsotonicRegression().fit(dataset)
print("Boundaries in increasing order: %s\n" % str(model.boundaries))
print("Predictions associated with the boundaries: %s\n" % str(model.predictions))

# Makes predictions.
model.transform(dataset).show()
```

Find full example code at "examples/src/main/python/ml/isotonic_regression_example.py" in the Spark repo.

Linear methods

We implement popular linear methods such as logistic regression and linear least squares with L_1 or L_2 regularization. Refer to the linear methods guide for the RDD-based API for details about implementation and tuning; this information is still relevant.

We also include a DataFrame API for Elastic net, a hybrid of L_1 and L_2 regularization proposed in Zou et al, Regularization and variable selection via the elastic net. Mathematically, it is defined as a convex combination of the L_1

and the L_2 regularization terms:

$$\alpha\left(\lambda\|\mathbf{w}\|_1\right) + (1-\alpha)\left(\frac{\lambda}{2}\|\mathbf{w}\|_2^2\right), \alpha \in [0, 1], \lambda \geq 0$$

By setting α properly, elastic net contains both L_1 and L_2 regularization as special cases. For example, if a linear regression model is trained with the elastic net parameter α set to 1, it is equivalent to a Lasso model. On the other hand, if α is set to 0, the trained model reduces to a ridge regression model. We implement Pipelines API for both linear regression and logistic regression with elastic net regularization.

Decision trees

Decision trees and their ensembles are popular methods for the machine learning tasks of classification and regression. Decision trees are widely used since they are easy to interpret, handle categorical features, extend to the multiclass classification setting, do not require feature scaling, and are able to capture non-linearities and feature interactions. Tree ensemble algorithms such as random forests and boosting are among the top performers for classification and regression tasks.

The `spark.ml` implementation supports decision trees for binary and multiclass classification and for regression, using both continuous and categorical features. The implementation partitions data by rows, allowing distributed training with millions or even billions of instances.

Users can find more information about the decision tree algorithm in the MLlib Decision Tree guide. The main differences between this API and the original MLlib Decision Tree API are:

- support for ML Pipelines
- separation of Decision Trees for classification vs. regression
- use of DataFrame metadata to distinguish continuous and categorical features

The Pipelines API for Decision Trees offers a bit more functionality than the original API.
In particular, for classification, users can get the predicted probability of each class (a.k.a. class conditional probabilities); for regression, users can get the biased sample variance of prediction.

Ensembles of trees (Random Forests and Gradient-Boosted Trees) are described below in the Tree ensembles section.

Inputs and Outputs

We list the input and output (prediction) column types here. All output columns are optional; to exclude an output column, set its corresponding Param to an empty string.

Input Columns

Param name	Type(s)	Default	Description
labelCol	Double	"label"	Label to predict
featuresCol	Vector	"features"	Feature vector

Output Columns

Param name	Type(s)	Default	Description	Notes

Param name	Type(s)	Default	Description	Notes
predictionCol	Double	"prediction"	Predicted label	
rawPredictionCol	Vector	"rawPrediction"	Vector of length # classes, with the counts of training instance labels at the tree node which makes the prediction	Classification only
probabilityCol	Vector	"probability"	Vector of length # classes equal to rawPrediction normalized to a multinomial distribution	Classification only
varianceCol	Double		The biased sample variance of prediction	Regression only

Tree Ensembles

The DataFrame API supports two major tree ensemble algorithms: Random Forests and Gradient-Boosted Trees (GBTs). Both use `spark.ml` decision trees as their base models.

Users can find more information about ensemble algorithms in the MLlib Ensemble guide.
In this section, we demonstrate the DataFrame API for ensembles.

The main differences between this API and the original MLlib ensembles API are:

- support for DataFrames and ML Pipelines
- separation of classification vs. regression
- use of DataFrame metadata to distinguish continuous and categorical features
- more functionality for random forests: estimates of feature importance, as well as the predicted probability of each class (a.k.a. class conditional probabilities) for classification.

Random Forests

Random forests are 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.

For more information on the algorithm itself, please see the `spark.mllib` documentation on random forests.

Inputs and Outputs

We list the input and output (prediction) column types here. All output columns are optional; to exclude an output column, set its corresponding Param to an empty string.

Input Columns

Param name	Type(s)	Default	Description
labelCol	Double	"label"	Label to predict
featuresCol	Vector	"features"	Feature vector

Output Columns (Predictions)

Param name	Type(s)	Default	Description	Notes
predictionCol	Double	"prediction"	Predicted label	
rawPredictionCol	Vector	"rawPrediction"	Vector of length # classes, with the counts of training instance labels at the tree node which makes the prediction	Classification only
probabilityCol	Vector	"probability"	Vector of length # classes equal to rawPrediction normalized to a multinomial distribution	Classification only

Gradient-Boosted Trees (GBTs)

Gradient-Boosted Trees (GBTs) are 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.

For more information on the algorithm itself, please see the `spark.mllib` documentation on GBTs.

Inputs and Outputs

We list the input and output (prediction) column types here. All output columns are optional; to exclude an output column, set its corresponding Param to an empty string.

Input Columns

Param name	Type(s)	Default	Description
labelCol	Double	"label"	Label to predict
featuresCol	Vector	"features"	Feature vector

Note that `GBTClassifier` currently only supports binary labels.

Output Columns (Predictions)

Param name	Type(s)	Default	Description	Notes
predictionCol	Double	"prediction"	Predicted label	

In the future, `GBTClassifier` will also output columns for `rawPrediction` and `probability`, just as `RandomForestClassifier` does.