# Spark Machine Learning

## 主要内容

* 基本统计
* 管道
* 提取、转换和选择特征
* **分类**

逻辑回归

**决策树**

随机森林

梯度提升树

多层感知器

线性支持向量机

一对多分类

朴素贝叶斯

* 回归

**线性回归**

广义线性回归

**决策树回归**

随机森林

梯度提升树回归

生存回归

保序回归

* **聚类**

**K-means**

Latent Dirichlet allocation (LDA)

Bisecting k-means

Gaussian Mixture Model (GMM)

* **协同过滤**
* 频繁模式挖掘
* 模型选择和调整
* 先进话题

## 决策树

决策树和它的集成（ensembles）在机器学习的回归和分类问题中都很常用。由于决策树良好的可解释性、可以处理分类特征、易于扩展到多分类、不需要进行特征缩放、可以捕捉到非线性和关联特征，所以它被广泛使用。基于决策树的随机森林和提升树集成算法在分类和回归问题上有极为出色的表现。

**例子**

下面这个例子加载了一个LibSVM格式的数据集，这个数据集被分为训练集和测试集。训练集被用来训练决策树模型，测试集被用来评价结果。我们使用特征转换器featureIndexer来建立分类特征索引，然后转换成DataFreme输入到决策树模型中。

<Java代码1>

**public class** DecitionTree {  
 **public static void** main(String[] args){

//windows需要添加此环境变量  
 System.*setProperty*(**"hadoop.home.dir"**,  
 **"C:\\Users\\pipi\\Downloads\\hadoop-common-2.2.0-bin-master\\hadoop-common-2.2.0-bin-master"**);  
 SparkSession spark = SparkSession  
 .*builder*()  
 .appName(**"Java Decition Tree Example"**)  
 .master(**"local"**)  
 .getOrCreate();Dataset<Row> data = spark.read().format(**"libsvm"**)  
 .load(**"./mllib/sample\_libsvm\_data.txt"**);VectorIndexerModel featureIndexer = **new** VectorIndexer()  
 .setInputCol(**"features"**)  
 .setOutputCol(**"indexedFeatures"**)  
 .setMaxCategories(4)  
 .fit(data);  
Dataset<Row>[] splits = data.randomSplit(**new double**[]{0.7, 0.3});  
 Dataset<Row> trainingData = splits[0];  
 Dataset<Row> testData = splits[1];  
  
 Dataset<Row> indexedTraingData = featureIndexer.transform(trainingData);  
 Dataset<Row> indexedTestingData = featureIndexer.transform(testData);  
DecisionTreeRegressor dt = **new** DecisionTreeRegressor()  
 .setFeaturesCol(**"indexedFeatures"**);  
DecisionTreeRegressionModel model = dt.fit(indexedTraingData);  
Dataset<Row> predictions = model.transform(indexedTestingData);  
predictions.select(**"label"**, **"features"**).show();  
RegressionEvaluator evaluator = **new** RegressionEvaluator()  
 .setLabelCol(**"label"**)  
 .setPredictionCol(**"prediction"**)  
 .setMetricName(**"rmse"**);  
 **double** rmse = evaluator.evaluate(predictions);  
 System.***out***.println(**"Root Mean Squared Error (RMSE) on test data = "** + rmse);System.***out***.println(**"Learned regression tree model:\n"** + model.toDebugString());  
 }  
}

<Java代码2使用pipline>

**public class** PiplineDecitionTree {  
 **public static void** main(String[] args) {  
 System.*setProperty*(**"hadoop.home.dir"**,  
 **"C:\\Users\\pipi\\Downloads\\hadoop-common-2.2.0-bin-master\\hadoop-common-2.2.0-bin-master"**);  
 SparkSession spark = SparkSession  
 .*builder*()  
 .appName(**"Java Decition Tree Example"**)  
 .master(**"local"**)  
 .getOrCreate();  
 *// Load the data stored in LIBSVM format as a DataFrame.* Dataset<Row> data = spark.read().format(**"libsvm"**)  
 .load(**"./mllib/sample\_libsvm\_data.txt"**);  
 *// Automatically identify categorical features, and index them.  
// Set maxCategories so features with > 4 distinct values are treated as continuous.* VectorIndexerModel featureIndexer = **new** VectorIndexer()  
 .setInputCol(**"features"**)  
 .setOutputCol(**"indexedFeatures"**)  
 .setMaxCategories(4)  
 .fit(data);  
  
*// Split the data into training and test sets (30% held out for testing).* Dataset<Row>[] splits = data.randomSplit(**new double**[]{0.7, 0.3});  
 Dataset<Row> trainingData = splits[0];  
 Dataset<Row> testData = splits[1];  
  
*// Train a DecisionTree model.* DecisionTreeRegressor dt = **new** DecisionTreeRegressor()  
 .setFeaturesCol(**"indexedFeatures"**);  
  
*// Chain indexer and tree in a Pipeline.* Pipeline pipeline = **new** Pipeline()  
 .setStages(**new** PipelineStage[]{featureIndexer, dt});  
  
*// Train model. This also runs the indexer.* PipelineModel model = pipeline.fit(trainingData);  
  
*// Make predictions.* Dataset<Row> predictions = model.transform(testData);  
  
*// Select example rows to display.* predictions.select(**"label"**, **"features"**).show(5);  
  
*// Select (prediction, true label) and compute test error.* RegressionEvaluator evaluator = **new** RegressionEvaluator()  
 .setLabelCol(**"label"**)  
 .setPredictionCol(**"prediction"**)  
 .setMetricName(**"rmse"**);  
 **double** rmse = evaluator.evaluate(predictions);  
 System.***out***.println(**"Root Mean Squared Error (RMSE) on test data = "** + rmse);  
  
 DecisionTreeRegressionModel treeModel =  
 (DecisionTreeRegressionModel) (model.stages()[1]);  
 System.***out***.println(**"Learned regression tree model:\n"** + treeModel.toDebugString());  
 }  
}

## 线性回归

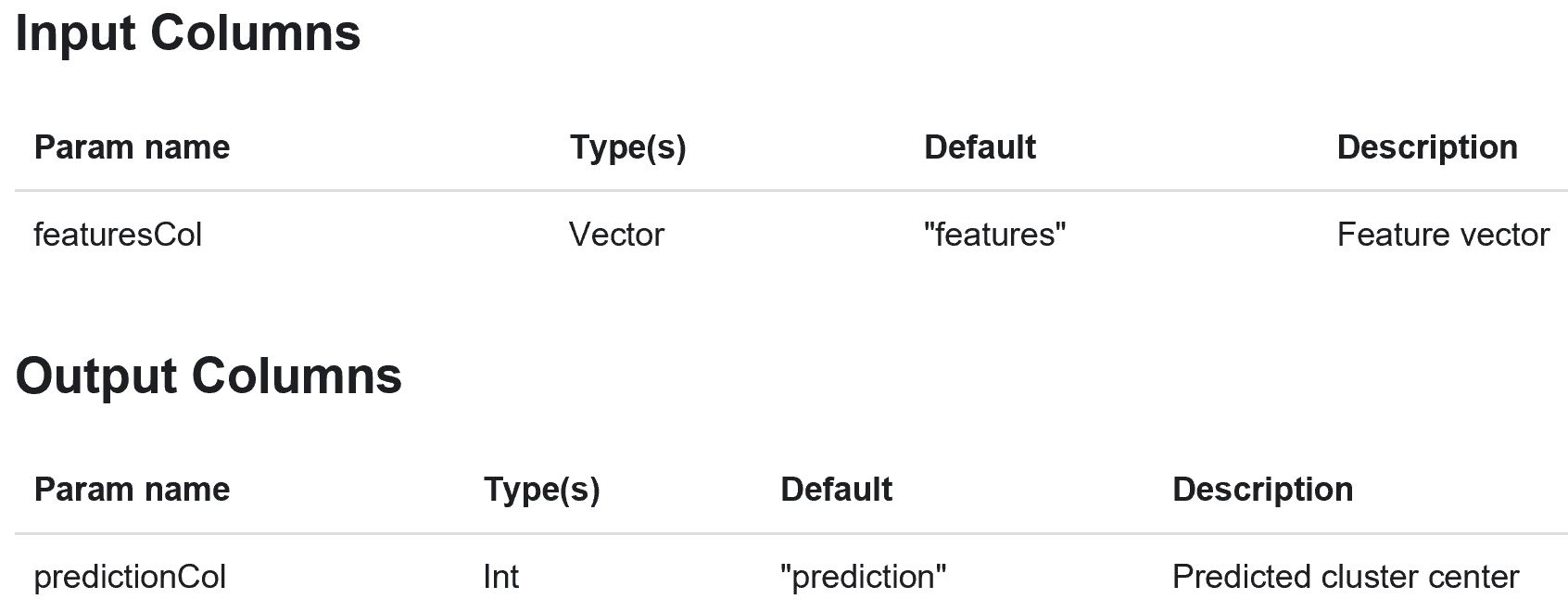
**例子**

<Java代码>

**public class** LinearRegressionTest {  
 **public static void** main(String[] args) {  
 System.*setProperty*(**"hadoop.home.dir"**,  
 **"C:\\Users\\pipi\\Downloads\\hadoop-common-2.2.0-bin-master\\hadoop-common-2.2.0-bin-master"**);  
 SparkSession spark = SparkSession  
 .*builder*()  
 .appName(**"Java Decition Tree Example"**)  
 .master(**"local"**)  
 .getOrCreate();  
 *// Load training data.* Dataset<Row> training = spark.read().format(**"libsvm"**)  
 .load(**"./mllib/sample\_linear\_regression\_data.txt"**);  
  
 LinearRegression lr = **new** LinearRegression()  
 .setMaxIter(10)  
 .setRegParam(0.3)  
 .setElasticNetParam(0.8);  
  
*// Fit the model.* LinearRegressionModel lrModel = lr.fit(training);  
  
*// Print the coefficients and intercept for linear regression.* System.***out***.println(**"Coefficients: "** + lrModel.coefficients() + **" Intercept: "** + lrModel.intercept());  
  
*// Summarize the model over the training set and print out some metrics.* LinearRegressionTrainingSummary trainingSummary = lrModel.summary();  
 System.***out***.println(**"numIterations: "** + trainingSummary.totalIterations());  
 System.***out***.println(**"objectiveHistory: "** + Vectors.*dense*(trainingSummary.objectiveHistory()));  
 trainingSummary.residuals().show();  
 System.***out***.println(**"RMSE: "** + trainingSummary.rootMeanSquaredError());  
 System.***out***.println(**"r2: "** + trainingSummary.r2());  
 }  
}

## K-means

K-means是一种广泛使用的聚类方法，可以将数据聚类为给定的数量。基于[k-means++](http://en.wikipedia.org/wiki/K-means%2B%2B)，Spark ML实现了一个并行化的方法。



<Java代码>

**public class** kmeans {  
 **public static void** main(String[] args) {  
 System.*setProperty*(**"hadoop.home.dir"**,  
 **"C:\\Users\\pipi\\Downloads\\hadoop-common-2.2.0-bin-master\\hadoop-common-2.2.0-bin-master"**);  
 SparkSession spark = SparkSession  
 .*builder*()  
 .appName(**"Java Decition Tree Example"**)  
 .master(**"local"**)  
 .getOrCreate();  
 *// Load training data.* Dataset<Row> dataset = spark.read().format(**"libsvm"**)  
 .load(**"./mllib/sample\_kmeans\_data.txt"**);  
 dataset.show();  
 *// Trains a k-means model.* KMeans kmeans = **new** KMeans().setK(3).setSeed(1L);  
 KMeansModel model = kmeans.fit(dataset);  
  
*// Evaluate clustering by computing Within Set Sum of Squared Errors.* **double** WSSSE = model.computeCost(dataset);  
 System.***out***.println(**"Within Set Sum of Squared Errors = "** + WSSSE);  
  
*// Shows the result.* Vector[] centers = model.clusterCenters();  
 System.***out***.println(**"Cluster Centers: "**);  
 **for** (Vector center: centers) {  
 System.***out***.println(center);  
 }  
 }  
}

## 协同过滤

协同过滤算法常用于推荐系统，来预测用户-物品矩阵的空缺值。Spark ML支持基于模型的协同过滤，用户和物品被分解为隐式特征因子，用来预测用户对物品的评分。Spark ML使用最小交叉二乘法来求这些特征因子。参数列表如下：

* *numBlocks* 并行化参数 (默认为 10).
* *rank* 隐因子数量 (默认为10).
* *maxIter* 最大迭代次数 (defaults to 10).
* *regParam* 最小二乘的惩罚参数 (defaults to 1.0).
* *implicitPrefs* 是否使用隐式反馈 (默认为false，使用显示反馈).
* *alpha* is a parameter applicable to the implicit feedback variant of ALS that governs the *baseline* confidence in preference observations (defaults to 1.0).
* *nonnegative* 在最小二乘中是否使用非负约束 (默认 false).

<Java代码>

**import** java.io.Serializable;  
  
**import** org.apache.spark.api.java.JavaRDD;  
**import** org.apache.spark.ml.evaluation.RegressionEvaluator;  
**import** org.apache.spark.ml.recommendation.ALS;  
**import** org.apache.spark.ml.recommendation.ALSModel;  
**import** org.apache.spark.sql.Dataset;  
**import** org.apache.spark.sql.Row;  
**import** org.apache.spark.sql.SparkSession;  
  
**public class** Rating **implements** Serializable {  
 **private int userId**;  
 **private int movieId**;  
 **private float rating**;  
 **private long timestamp**;  
  
 **public** Rating() {}  
  
 **public** Rating(**int** userId, **int** movieId, **float** rating, **long** timestamp) {  
 **this**.**userId** = userId;  
 **this**.**movieId** = movieId;  
 **this**.**rating** = rating;  
 **this**.**timestamp** = timestamp;  
 }  
  
 **public int** getUserId() {  
 **return userId**;  
 }  
  
 **public int** getMovieId() {  
 **return movieId**;  
 }  
  
 **public float** getRating() {  
 **return rating**;  
 }  
  
 **public long** getTimestamp() {  
 **return timestamp**;  
 }  
  
 **public static** Rating parseRating(String str) {  
 String[] fields = str.split(**"::"**);  
 **if** (fields.**length** != 4) {  
 **throw new** IllegalArgumentException(**"Each line must contain 4 fields"**);  
 }  
 **int** userId = Integer.*parseInt*(fields[0]);  
 **int** movieId = Integer.*parseInt*(fields[1]);  
 **float** rating = Float.*parseFloat*(fields[2]);  
 **long** timestamp = Long.*parseLong*(fields[3]);  
 **return new** Rating(userId, movieId, rating, timestamp);  
 }  
  
 **public static void** main(String[] args) {  
 System.*setProperty*(**"hadoop.home.dir"**,  
 **"C:\\Users\\pipi\\Downloads\\hadoop-common-2.2.0-bin-master\\hadoop-common-2.2.0-bin-master"**);  
 SparkSession spark = SparkSession  
 .*builder*()  
 .appName(**"Java Decition Tree Example"**)  
 .master(**"local"**)  
 .getOrCreate();  
 JavaRDD<Rating> ratingsRDD = spark  
 .read().textFile(**"./mllib/als/sample\_movielens\_ratings.txt"**).javaRDD()  
 .map(Rating::*parseRating*);  
 Dataset<Row> ratings = spark.createDataFrame(ratingsRDD, Rating.**class**);  
 Dataset<Row>[] splits = ratings.randomSplit(**new double**[]{0.8, 0.2});  
 Dataset<Row> training = splits[0];  
 Dataset<Row> test = splits[1];  
  
*// Build the recommendation model using ALS on the training data* ALS als = **new** ALS()  
 .setMaxIter(5)  
 .setRegParam(0.01)  
 .setUserCol(**"userId"**)  
 .setItemCol(**"movieId"**)  
 .setRatingCol(**"rating"**);  
 ALSModel model = als.fit(training);  
  
*// Evaluate the model by computing the RMSE on the test data  
// Note we set cold start strategy to 'drop' to ensure we don't get NaN evaluation metrics* model.setColdStartStrategy(**"drop"**);  
 Dataset<Row> predictions = model.transform(test);  
  
 RegressionEvaluator evaluator = **new** RegressionEvaluator()  
 .setMetricName(**"rmse"**)  
 .setLabelCol(**"rating"**)  
 .setPredictionCol(**"prediction"**);  
 Double rmse = evaluator.evaluate(predictions);  
 System.***out***.println(**"Root-mean-square error = "** + rmse);  
  
*// Generate top 10 movie recommendations for each user* Dataset<Row> userRecs = model.recommendForAllUsers(10);  
*// Generate top 10 user recommendations for each movie* Dataset<Row> movieRecs = model.recommendForAllItems(10);  
  
 *//spark2.3  
//// Generate top 10 movie recommendations for a specified set of users  
// Dataset<Row> users = ratings.select(als.getUserCol()).distinct().limit(3);  
// Dataset<Row> userSubsetRecs = model.recommendForUserSubset(users, 10);  
//// Generate top 10 user recommendations for a specified set of movies  
// Dataset<Row> movies = ratings.select(als.getItemCol()).distinct().limit(3);  
// Dataset<Row> movieSubSetRecs = model.recommendForItemSubset(movies, 10);* }  
}