Spark MLlib 研究、使用

目录

一、概要

MLlib 是一个可扩展的 Spark 机器学习库，由通用的学习算法和工具组成，包括二元分类、线性回归、聚类、协同过滤、降维、特征提取和转换以及底层优化原语。借助 Spark 分布式计算框架和弹性分布式数据集（RDD），能够高效完成机器学习工作。

MLlib 提供 scala，java，python 三种 API（spark1.4 新增了 SparkR），本文基于 Spark1.3.1 对 MLlib 的一些概念及部分算法使用方法做一些说明，同时对其中一些例子做了调整，详细代码及测试数据集见附件程序。

二、数据类型

MLlib 中数据类型主要包括：Local vector,Labeled point,Local matrix,Distributed matrix。

2.1 本地向量（Local vector）

它类似一维数组，值是 double 类型，存储在单一机器上，它有两种表示方式：dense(密集)、sparse（稀疏）。

举例：vector(1.0,0.0,3.0)

dense 格式：[1.0，0.0，3.0]

sparse 格式：（3，[0，2]，[1.0，3.0]），其中 3 表示长度，省略 0 值。

```
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.Vectors;

// Create a dense vector (1.0, 0.0, 3.0).
Vector dv = Vectors.dense(1.0, 0.0, 3.0);
// Create a sparse vector (1.0, 0.0, 3.0) by specifying its indices and values corresponding to nonzero entries.
Vector sv = Vectors.sparse(3, new int[] {0, 2}, new double[] {1.0, 3.0});
```

2.2 标记点（Labeled point）

标记点是一个本地向量与一个标记值组成的一个数据对，它被用于监督式算法中，标记值用 double 表示，所以，我们可以在回归和分类中使用它。在二元分类中，标记值是 0 或 1，多类分类中，它是以 0 开始的索引值：0，1，2，3…。

```java
import org.apache.spark.mllib.linalg.Vectors;
import org.apache.spark.mllib.regression.LabeledPoint;

// Create a labeled point with a positive label and a dense feature vector.
LabeledPoint pos = new LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0));

// Create a labeled point with a negative label and a sparse feature vector.
LabeledPoint neg = new LabeledPoint(1.0, Vectors.sparse(3, new int[] {0, 2}, new double[] {1.0, 3.0}));
```

2.3 本地矩阵（Local matrix）

本地矩阵存储在一台机器上，行列标记用 integer 表示，值用 double 类型,MLlib 支持密集矩阵,例如,一个三行两列的矩阵会被存储在一个一维数组[1.0,3.0,4.0,2.0,3.0,5.0]及大小为（3，2）的矩阵中。

（1.0，3.0

4.0，2.0

3.0，5.0）

```java
import org.apache.spark.mllib.linalg.Matrix;
import org.apache.spark.mllib.linalg.Matrices;

// Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
Matrix dm = Matrices.dense(3, 2, new double[] {1.0, 3.0, 5.0, 2.0, 4.0, 6.0});
```

2.4 分布式矩阵（Distributed matrix）

分布式矩阵由 long 类型行列索引以及 double 类型值组成,被分散的存储在一个或多个 RDD 上，目前有三种类型的分布式矩阵：RowMatrix,IndexedRowMatrix,CoordinateMarix，BlockMatrix。选择正确的存储格式去存储巨大分布式矩阵非常重要，因为，转换一个分布式矩阵的存储格式的代价是昂贵的，它会导致全局洗牌。

RowMatrix 是一个面向行的分布式矩阵，没有行索引。用一个 RDD 表示所有行，每一行都是一个本地向量，因此它的列数有限，以便于一个本地向量能够存储它的一行。

```java
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.distributed.RowMatrix;

JavaRDD<Vector> rows = ... // a JavaRDD of local vectors
// Create a RowMatrix from an JavaRDD<Vector>.
RowMatrix mat = new RowMatrix(rows.rdd());

// Get its size.
long m = mat.numRows();
long n = mat.numCols();
```

IndexedRowMatrix 类似 RowMatrix，但它具有行索引，行索引可以用于标示行及执行关联操作。每一行由一个 long 类型索引和一个本地向量组成。其中，IndexedRow 是一个被包装过的（long,vector）。

```
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.mllib.linalg.distributed.IndexedRow;
import org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix;
import org.apache.spark.mllib.linalg.distributed.RowMatrix;

JavaRDD<IndexedRow> rows = ... // a JavaRDD of indexed rows
// Create an IndexedRowMatrix from a JavaRDD<IndexedRow>.
IndexedRowMatrix mat = new IndexedRowMatrix(rows.rdd());

// Get its size.
long m = mat.numRows();
long n = mat.numCols();

// Drop its row indices.
RowMatrix rowMat = mat.toRowMatrix();
```

CoordinateMarix 是一个实体集合，每个实体有行索引（long）、列索引（long）及值（double）组成：(i:Long, j:Long, value:Double)。它常用于行、列都很大，而且值稀疏的场景。

```
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.mllib.linalg.distributed.CoordinateMatrix;
import org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix;
import org.apache.spark.mllib.linalg.distributed.MatrixEntry;

JavaRDD<MatrixEntry> entries = ... // a JavaRDD of matrix entries
// Create a CoordinateMatrix from a JavaRDD<MatrixEntry>.
CoordinateMatrix mat = new CoordinateMatrix(entries.rdd());

// Get its size.
long m = mat.numRows();
long n = mat.numCols();

// Convert it to an IndexRowMatrix whose rows are sparse vectors.
IndexedRowMatrix indexedRowMatrix = mat.toIndexedRowMatrix();
```

BlockMatrix 是 MatrixBlock 的集合，MatrixBlock 是一个元组，它的组成为：((int ,int),matrix),其中（int,int）是块索引，块大小默认为 1024*1024，这个大小可以指定。

```
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.mllib.linalg.distributed.BlockMatrix;
import org.apache.spark.mllib.linalg.distributed.CoordinateMatrix;
import org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix;

JavaRDD<MatrixEntry> entries = ... // a JavaRDD of (i, j, v) Matrix Entries
// Create a CoordinateMatrix from a JavaRDD<MatrixEntry>.
CoordinateMatrix coordMat = new CoordinateMatrix(entries.rdd());
// Transform the CoordinateMatrix to a BlockMatrix
BlockMatrix matA = coordMat.toBlockMatrix().cache();

// Validate whether the BlockMatrix is set up properly. Throws an Exception when it is not valid.
// Nothing happens if it is valid.
matA.validate();

// Calculate A^T A.
BlockMatrix ata = matA.transpose().multiply(matA);
```

三、基本统计分析

3.1 概要统计量（Summary statistics）

对 RDD[Vector]格式数据，MLlib 提供面向列的最大值、最小值、平均值、方差、非零值个数以及总数量的统计量指标，具体通过 Statistics.colStats 方法实现。

3.2 相关性（Correlations）

在统计分析中，计算两系列数据间的相关性很常见。在 MLlib 中，提供了用于计算多系列数据间两两关系的方法。目前支持的相关性算法有 Perarson 和 Speasman。

3.3 分层抽样（Stratified sampling）

在MLlib中，分层抽样方法有sambleByKey和sampleByKeyExact,不同于其它统计方法，分层抽样的输入要求是键值对格式的RDD。对分层抽样来说，keys是一个标签，值是特定属性，类似于labledpoint。sampleByKey方法会随机决定某个观察值是否需要抽样，因此需要提供一个确定的抽样数量，sampleByKeyExact比sampleByKey需要更多明确的资源。没有替换的抽样需要一个额外的RDD来确保抽样数量，而带有替换的抽样需要两个额外的RDD。

```
import java.util.Map;

import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;

JavaSparkContext jsc = ...

JavaPairRDD<K, V> data = ... // an RDD of any key value pairs
Map<K, Object> fractions = ... // specify the exact fraction desired from each key

// Get an exact sample from each stratum
JavaPairRDD<K, V> approxSample = data.sampleByKey(false, fractions);
JavaPairRDD<K, V> exactSample = data.sampleByKeyExact(false, fractions);
```

3.4 假设测定（Hypothesis testing）

MLlib目前支持Pearson’s chi-squared以测定适配度和独立性。输入数据类型决定了测定是否产生适配度或独立性，适配度测试需要输入类型为Vector，而独立性测试需要Matrix作为输入。

MLlib也支持RDD[LabeledPoint]输入类型，然后使用chi-squared独立性测试来运行特征选择。

```
JavaSparkContext jsc = ...

Vector vec = ... // a vector composed of the frequencies of events

// compute the goodness of fit. If a second vector to test against is not supplied as a parameter,
// the test runs against a uniform distribution.
ChiSqTestResult goodnessOfFitTestResult = Statistics.chiSqTest(vec);
// summary of the test including the p-value, degrees of freedom, test statistic, the method used,
// and the null hypothesis.
System.out.println(goodnessOfFitTestResult);

Matrix mat = ... // a contingency matrix

// conduct Pearson's independence test on the input contingency matrix
ChiSqTestResult independenceTestResult = Statistics.chiSqTest(mat);
// summary of the test including the p-value, degrees of freedom...
System.out.println(independenceTestResult);

JavaRDD<LabeledPoint> obs = ... // an RDD of labeled points

// The contingency table is constructed from the raw (feature, label) pairs and used to conduct
// the independence test. Returns an array containing the ChiSquaredTestResult for every feature
// against the label.
ChiSqTestResult[] featureTestResults = Statistics.chiSqTest(obs.rdd());
int i = 1;
for (ChiSqTestResult result : featureTestResults) {
    System.out.println("Column " + i + ":");
    System.out.println(result); // summary of the test
    i++;
```

3.5 随机数生成（Random data generation）

生成随机数对随机算法、性能测试都是很有用的，MLlib支持使用i.i.d生成随机RDD：均匀、标准常态、波松分布。

四、分类和回归

MLlib支持多种方法用来处理二元分类、多元分类及回归分析。具体到算法的相关推导公式，这里就不做介绍，详细可以参照相关官方文档。

4.1 二元分类

二元分类旨在将item分成两个种类：积极和消极。MLlib支持两种线性方法：线性支持向量机（SVM）和逻辑回归，这两种方法都支持L1和L2正则化变体，在MLlib中训练数据集表

示为LabeledPoint的一个RDD，在本文的数学表达式中，训练标签 y 表示为 +1 （正）和 -1 （负），然而在MLlib中使用 0 来表示负的。

MLlib支持常见的二分分类评估指标，包括精确、召回、F值、ROC、精密召回曲线和AUC，AUC在比较多个模型的性能是很常用的，精确、召回、F值可以帮助决定在预测中适当的下限值。

这里以支持向量机（SVMModel）和逻辑回归（LogisticRegressionModel）两种算法为例说明如何导入数据、训练模型、预测以及评估。

4.2 线性回归

线性回归是利用数理统计中的回归分析，来确定两种或两种以上变量间相互依赖的定量关系的一种统计分析方法，运用十分广泛。在统计学中，线性回归(Linear Regression)是利用称为线性回归方程的最小平方函数对一个或多个自变量和因变量之间关系进行建模的一种回归分析。MLlib 支持 Linear least squares(最小二乘法)、Lasso 和 ridge regression 三种算法。另外，MLlib 目前也支持基于流数据的线性回归分析，这个功能只有对应的 scala api。这里就以最小最小二乘法为例说明如何导入数据、训练模型、预测以及评估。

```java
public class SVMClassifier {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("SVM Classifier Example");
    SparkContext sc = new SparkContext(conf);
    String path = "hdfs://udh-spark-01:8020/tmp/mllib/sample_libsvm_data.txt";
    JavaRDD<LabeledPoint> data = MLUtils.loadLibSVMFile(sc, path).toJavaRDD();

    // Split initial RDD into two... [60% training data, 40% testing data].
    JavaRDD<LabeledPoint> training = data.sample(false, 0.6, 11L);
    training.cache();
    JavaRDD<LabeledPoint> test = data.subtract(training);

    // Run training algorithm to build the model.
    int numIterations = 1;
    final SVMModel model = SVMWithSGD.train(training.rdd(), numIterations);
    // Clear the default threshold.
    model.clearThreshold();

    // Compute raw scores on the test set.
    JavaRDD<Tuple2<Object, Object>> scoreAndLabels = test.map(
      new Function<LabeledPoint, Tuple2<Object, Object>>() {
        public Tuple2<Object, Object> call(LabeledPoint p) {
          Double score = model.predict(p.features());
          return new Tuple2<Object, Object>(score, p.label());
        }
      }
    );
    // Get evaluation metrics.
    BinaryClassificationMetrics metrics =
            new BinaryClassificationMetrics(JavaRDD.toRDD(scoreAndLabels));
    double auROC = metrics.areaUnderROC();

    System.out.println("----------------------begin--------------------------");
    List<Tuple2<Object, Object>> pairs = scoreAndLabels.collect();
    for(Tuple2<Object, Object> item:pairs){
        System.out.println("tuple predict:"+item._1()+" real:"+item._2);
    }

    System.out.println("model weight:"+model.weights());
    System.out.println("Area under ROC = " + auROC);
    System.out.println("----------------------end--------------------------");

    sc.stop();
  }
}
```

```java
public class MultinomialLogisticRegressionExample {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("SVM Classifier Example");
    SparkContext sc = new SparkContext(conf);
    String path = "hdfs://udh-spark-01:8020/tmp/mllib/sample_libsvm_data.txt";
    JavaRDD<LabeledPoint> data = MLUtils.loadLibSVMFile(sc, path).toJavaRDD();
    // Split initial RDD into two... [60% training data, 40% testing data].
    JavaRDD<LabeledPoint>[] splits = data.randomSplit(new double[] {0.6, 0.4}, 11L);
    JavaRDD<LabeledPoint> training = splits[0].cache();
    JavaRDD<LabeledPoint> test = splits[1];

    // Run training algorithm to build the model.
    final LogisticRegressionModel model = new LogisticRegressionWithLBFGS()
      .setNumClasses(10)
      .run(training.rdd());

    // Compute raw scores on the test set.
    JavaRDD<Tuple2<Object, Object>> predictionAndLabels = test.map(
      new Function<LabeledPoint, Tuple2<Object, Object>>() {
        public Tuple2<Object, Object> call(LabeledPoint p) {
          Double prediction = model.predict(p.features());
          return new Tuple2<Object, Object>(prediction, p.label());
        }
      }
    );

    // Get evaluation metrics.
    MulticlassMetrics metrics = new MulticlassMetrics(predictionAndLabels.rdd());
    double precision = metrics.precision();

    System.out.println("----------------------begin--------------------------");
    List<Tuple2<Object, Object>> pairs = predictionAndLabels.collect();
    for(Tuple2<Object, Object> item:pairs){
        System.out.println("tuple predict:"+item._1()+" real:"+item._2);
    }

    System.out.println("model weight:"+model.weights());
    System.out.println("Precision = " + precision);
    System.out.println("----------------------end--------------------------");
    sc.stop();
  }
}
```

4.3 决策树

决策树算法在机器学习中用到的地方很多，因为它容易学习并易于处理类别特征。MLlib 支持通过决策树处理二分、多分和回归，实现的方式是使用行来对数据进行分区，这样就允许多个实例来对数据进行分布式训练。这个具体参照官方文档说明，这里不做进一步说明。

4.4 朴素贝叶斯（Naive Bayes）

朴素贝叶斯是一种简单的多类分类算法。该算法会给问题实例分配用特征值表示的类标签，类标签取自有限集合。它不是训练这种分类器的单一算法，而是一系列基于相同原理的算法：所有朴素贝叶斯分类器都假定样本每个特征与其他特征都不相关。举个例子，如果一种水果其具有红，圆，直径大概 3 英寸等特征，该水果可以被判定为是苹果。尽管这些特征相互依赖或者有些特征由其他特征决定，然而朴素贝叶斯分类器认为这些属性在判定该水果是否为苹果的概率分布上独立的。MLlib 目前只支持 multinomial naive Bayes，最新版本

加入了 Bernoulli naive Bayes 的支持。下面例子说明如何导入数据、训练模型、预测以及评估。

```java
public class NativeBayesExample {

    public static void main(String[] args) {

        SparkConf conf = new SparkConf().setAppName("Native Bayes Example");
        SparkContext sc = new SparkContext(conf);
        String path = "hdfs://udh-spark-01:8020/tmp/mllib/sample_naive_bayes_data.txt";

        JavaRDD<String> lines = sc.textFile(path, 1).toJavaRDD();
        JavaRDD<LabeledPoint> data = lines.map(new Function<String, LabeledPoint>() {
            public LabeledPoint call(String line){
                return LabeledPoint.parse(line);
            }
        });

        JavaRDD<LabeledPoint>[] splits = data.randomSplit(new double[] {0.8, 0.2}, 11L);
        JavaRDD<LabeledPoint> training = splits[0].cache();
        JavaRDD<LabeledPoint> test = splits[1];

        final NaiveBayesModel model = NaiveBayes.train(training.rdd(), 1.0);
        JavaPairRDD<Double, Double> predictionAndLabel =
          test.mapToPair(new PairFunction<LabeledPoint, Double, Double>() {
            @Override public Tuple2<Double, Double> call(LabeledPoint p) {
              return new Tuple2<Double, Double>(model.predict(p.features()), p.label());
            }
          });
        double accuracy = predictionAndLabel.filter(new Function<Tuple2<Double, Double>, Boolean>() {
            @Override public Boolean call(Tuple2<Double, Double> pl) {
              return pl._1().equals(pl._2());
            }
          }).count() / (double) test.count();

        System.out.println("----------------------begin--------------------------");
        List<Tuple2<Double, Double>> pairs = predictionAndLabel.collect();
        for(Tuple2<Double, Double> item:pairs){
            System.out.println("tuple predict:"+item._1()+" real:"+item._2);
        }

        System.out.println("model accuracy:"+accuracy);
        System.out.println("----------------------end--------------------------");
        sc.stop();
    }
}
```

4.5 保序回归（Isotonic regression）

保序回归用法上类似线性回归，不过，它的输入是一个具有三元组的 RDD，分别为：label，feature 、 weight，模型类是 IsotonicRegressionModel，它有一个参数（isotonic）可以用来设置单调递增还是单调递减。具体使用这里就不举例说明了。

五、协同过滤（Collaborative filtering）

协同滤波可以看做是一个分类问题，也可以看做是矩阵分解问题，它常用于推荐系统。协同过滤主要是基于每个人自己的喜好都类似这一特征，它不依赖于个人的基本信息。比如电影评分的例子中，预测那些没有被评分的电影的分数只依赖于已经打分的那些分数，并不需要去学习那些电影的特征。下面的例子以 movielens 上的数据集（10m）进行的模拟测试，

上面还有 100k,1m,20m 不同大小的数据集可供下载测试。

```
String path = "hdfs://udh-spark-01:8020/tmp/mllib/ratings.dat";
JavaRDD<String> data = sc.textFile(path);
JavaRDD<Rating> ratings = data.map(
  new Function<String, Rating>() {
    public Rating call(String s) {
      String[] sarray = s.split("::");
      return new Rating(Integer.parseInt(sarray[0]), Integer.parseInt(sarray[1]),
                        Double.parseDouble(sarray[2]));
    }
  }
);
// Build the recommendation model using ALS
int rank = 10;
int numIterations = 20;
MatrixFactorizationModel model = ALS.train(JavaRDD.toRDD(ratings), rank, numIterations, 0.01);
// Evaluate the model on rating data
JavaRDD<Tuple2<Object, Object>> userProducts = ratings.map(
  new Function<Rating, Tuple2<Object, Object>>() {
    public Tuple2<Object, Object> call(Rating r) {
      return new Tuple2<Object, Object>(r.user(), r.product());
    }
  }
);
JavaPairRDD<Tuple2<Integer, Integer>, Double> predictions = JavaPairRDD.fromJavaRDD(
  model.predict(JavaRDD.toRDD(userProducts)).toJavaRDD().map(
    new Function<Rating, Tuple2<Tuple2<Integer, Integer>, Double>>() {
      public Tuple2<Tuple2<Integer, Integer>, Double> call(Rating r){
        return new Tuple2<Tuple2<Integer, Integer>, Double>(
          new Tuple2<Integer, Integer>(r.user(), r.product()), r.rating());
      }
    }
));
JavaPairRDD<Tuple2<Integer,Integer>,Tuple2<Double,Double>> ratesAndPreds =
          JavaPairRDD.fromJavaRDD(ratings.map(
            new Function<Rating, Tuple2<Tuple2<Integer, Integer>, Double>>() {
              public Tuple2<Tuple2<Integer, Integer>, Double> call(Rating r){
                return new Tuple2<Tuple2<Integer, Integer>, Double>(
                  new Tuple2<Integer, Integer>(r.user(), r.product()), r.rating());
              }
            }
        )).join(predictions);

JavaRDD<Tuple2<Double, Double>> ratesAndPredsValues = ratesAndPreds.values();
double MSE = JavaDoubleRDD.fromRDD(ratesAndPredsValues.map(
  new Function<Tuple2<Double, Double>, Object>() {
    public Object call(Tuple2<Double, Double> pair) {
      Double err = pair._1() - pair._2();
      return err * err;
    }
  }
).rdd()).mean();
```

六、聚类算法

聚类是把相似的对象通过静态分类的方法分成不同的组别或者更多的子集（subset），这样让在同一个子集中的成员对象都有相似的一些属性，常见的包括在坐标系中更加短的空间距离等。一般把数据聚类归纳为一种非监督式学习。MLlib 里包含了 K-Means、Gaussian mixture、pic、lda、streaming K-Means 五种聚类算法，这里只着重介绍下 K-Means。K-Means 最大的特别和优势在于模型的建立不需要训练数据。在日常工作中，很多情况下没有办法事先获取到有效的训练数据，这时采用 K-Means 是一个不错的选择。但 K-Means 需要预先设置

有多少个簇类（K 值），这对于像计算某省份全部电信用户的交往圈这样的场景就完全的没办法用 K-Means 进行。对于可以确定 K 值不会太大但不明确精确的 K 值的场景，可以进行迭代运算，然后找出 cost 最小时所对应的 K 值，这个值往往能较好的描述有多少个簇类。下面例子说明如何导入数据、训练模型、预测以及评估。

```
public class KMeansExample {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("K-means Example");
    JavaSparkContext sc = new JavaSparkContext(conf);

    // Load and parse data
    String path = "hdfs://udh-spark-01:8020/tmp/mllib/kmeans_data.txt";
    JavaRDD<String> data = sc.textFile(path);
    JavaRDD<Vector> parsedData = data.map(
      new Function<String, Vector>() {
        public Vector call(String s) {
          String[] sarray = s.split(" ");
          double[] values = new double[sarray.length];
          for (int i = 0; i < sarray.length; i++)
            values[i] = Double.parseDouble(sarray[i]);
          return Vectors.dense(values);
        }
      }
    );
    parsedData.cache();

    // Cluster the data into two classes using KMeans
    int numClusters = 2;
    int numIterations = 20;
    KMeansModel clusters = KMeans.train(parsedData.rdd(), numClusters, numIterations);
    // Evaluate clustering by computing Within Set Sum of Squared Errors
    double WSSSE = clusters.computeCost(parsedData.rdd());

    JavaRDD<Integer> predicts = clusters.predict(parsedData);
    List<Integer> datas = predicts.collect();

    System.out.println("----------------------begin---------------------------");
    for(Integer item:datas){
        System.out.println("分类: "+item);
    }
    System.out.println("Within Set Sum of Squared Errors = " + WSSSE);
    System.out.println("----------------------end---------------------------");
    sc.stop();
  }
}
```

七、降维（Dimensionality Reduction）

降维就是指采用某种映射方法，将原高维空间中的数据点映射到低维度的空间中。降维的本质是学习一个映射函数 f : x->y，其中 x 是原始数据点的表达，目前最多使用向量表达形式。 y 是数据点映射后的低维向量表达，通常 y 的维度小于 x 的维度（当然提高维度也是可以的）。f 可能是显式的或隐式的、线性的或非线性的。通过降维可以减少数据的冗余及噪音信息或者保持原油数据结构的情况下压缩数据，加速后续的计算速度,而且降维后的数据也便于可视化。MLlib 中提供两种降维算法：奇异值分解（SVD）、主成分分析（PCA），下面例子具体说明了在 MLlib 中如何使用这两个模型。

```java
public class SVD {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("SVD Example");
    JavaSparkContext sc = new JavaSparkContext(conf);
    String path = "hdfs://udh-spark-01:8020/tmp/mllib/dr.txt";
    JavaRDD<String> data = sc.textFile(path);

    JavaRDD<Vector> parsedData = data.map(
              new Function<String, Vector>() {
                public Vector call(String s) {
                  String[] sarray = s.split(" ");
                  double[] values = new double[sarray.length];
                  for (int i = 0; i < sarray.length; i++)
                    values[i] = Double.parseDouble(sarray[i]);
                  return Vectors.dense(values);
                }
              }
    );

    // Create a RowMatrix from JavaRDD<Vector>.
    RowMatrix mat = new RowMatrix(parsedData.rdd());
    // Compute the top 4 singular values and corresponding singular vectors.
    SingularValueDecomposition<RowMatrix, Matrix> svd = mat.computeSVD(3, true, 1.0E-9d);
    RowMatrix U = svd.U();
    Vector s = svd.s();
    Matrix V = svd.V();
    System.out.println("----------------------begin---------------------------");
    System.out.println("RowMatrix U = " + U+" NUM ROWS="+U.numRows()+" NUM COLS="+U.numCols());
    System.out.println("Vector s = " + s);
    System.out.println("Matrix V = " + V);
    System.out.println("Matrix V NUM ROWS= " + V.numRows()+" NUM COLS="+V.numCols());
    System.out.println("----------------------end---------------------------");
    sc.stop();

  }
}
```

```java
public class PCA {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("PCA Example");
    JavaSparkContext sc = new JavaSparkContext(conf);

    String path = "hdfs://udh-spark-01:8020/tmp/mllib/dr.txt";
    JavaRDD<String> data = sc.textFile(path);

    JavaRDD<Vector> parsedData = data.map(
              new Function<String, Vector>() {
                public Vector call(String s) {
                  String[] sarray = s.split(" ");
                  double[] values = new double[sarray.length];
                  for (int i = 0; i < sarray.length; i++)
                    values[i] = Double.parseDouble(sarray[i]);
                  return Vectors.dense(values);
                }
              }
    );

    // Create a RowMatrix from JavaRDD<Vector>.
    RowMatrix mat = new RowMatrix(parsedData.rdd());

    // Compute the top 3 principal components.
    Matrix pc = mat.computePrincipalComponents(3);
    RowMatrix projected = mat.multiply(pc);

    System.out.println("----------------------begin---------------------------");
    System.out.println("Matrix pc =" + pc+" NUM ROWS="+pc.numRows()+" NUM COLS="+pc.numCols());
    System.out.println("RowMatrix projected ="+projected+ " NUM ROWS="+projected.numRows()+" NUM COLS="+projected.numCols
    System.out.println("----------------------end---------------------------");
    sc.stop();
  }
}
```

八、特征提取和转换

特征提取和转换处于机器学习中数据预处理阶段，该阶段处理的好坏会直接影响最终预测模型的准确性。这块的东西 MLlib 中提供的相对还比较有限，这里做下简单说明，就不再

具体举例了。

这一块主要包含以下三部分内容：

TF-IDF、*Word2Vec* 属于文本分析算法。

StandardScaler、*Normalizer* 用于数据规范化处理。

Feature selection 是指模式识别中，对某一模式的一组测量值进行变换以突出该模式具有代表性特征的方法，降维可以理解为特征提取中的一类。

九、频繁模式挖掘（Frequent Pattern Mining）

频繁模式挖掘是通常是大规模数据分析的第一步，多年以来它都是数据挖掘领域的活跃研究主题。MLlib 支持了一个并行的 FP-growth，FP-growth 是很受欢迎的频繁项集挖掘算法。

参考资料：http://spark.apache.org/docs/1.3.1/MLlib-guide.html

具体代码请下载附件查看，由于 spark 自身类库太大，打包程序里没有包含，可以自行从 UDH SPARK 集群环境中下载。