Machine Learning with MLlib

MLib 是 Spark 上用于实现机器学习功能的 Spark 库。MLib 被设计成在集群上并行的运行，包含了大量的学习算法和支持 spark 所有的编程语言。本章将向您讲解如何在你的程序中使用 Mlib，并提供通用的使用方法。
机器学习本身就是一门足以填满很多书的学科，所以很抱歉，在本章中我们没有足够的空间去详细的向您阐述机器学习。如果你很熟悉机器学历，这章将向您阐述如何使用 spark；但是即使你对机器学习很陌生，你也可以将本章的材料和其他的材料结合起来。本章的内容面向想使用 spark 的有机器学习背景的数据分析师，以及与机器学习专家一起工作的工程师。

Overview

MLib 的设计和体系很简单：用 RDDs 代表所有的数据，让你在分布式的数据集上运行各种算法。MLib 引进了几个数据类型（如 labled points 和 vectors）,但是最终，它只是简单的一组在 RDDs 上调用的函数。比如，要用 MLib 去实现文本分类（如识别垃圾邮件）功能，你可能做下面的这些事情：

1．将一个包含信息的字符串类型的 RDD 作为开始。
2．运行一个类型识别算法将文本转换成数值特征（要适合机器学习算法），这将返回一个包含 vector 类型的 RDD。
3．在 RDD 的 vectors 上运行分类算法（如 logistic regression 逻辑回归）；它将返回一个用于识别新点的模型对象。
4．在一个测试数据集上运行这个模型，并用一个 MLIB 的评估函数去评估它。

关于 MLib 一个需要注意的事是，MLib 只包含在集群上运行很好的并行算法。因为一些经典的机器学习算法不是为并行平台而设计的，所以没有包含在 MLib 中。但是另一方面，Mlib 也包含了几个为集群设计的新近研究算法，如分布式随机森林（distributed random forests）、K-means||，和交替最小二乘法（alternating least squares）。这种选择意味着 MLib 最适合于在打数据集上运行算法。如果你想在很多小数据集上训练不同的学习模型，还是在每个节点上运行单节点学习库（如 Weka 或 SciKit-Learn）比较好，或许能够使用 spark map()跨节点并行调用它。同样的，为了选择最优的算法配置，通常将相同的算法在小数据集上以不同参数配置进行训练。你可以在 Spark 上用一个参数 list（作为输入）运行 parallelize（）去在不同的节点上训练不同的算法，接着在每个节点上运行单节点机器学习库。但是，当你有个一个需要训练模型的大的、分布式的数据集的时候，MLib 的表现是突出的。
最后，在 Spark1.0 和 spark1.1 中，MLib 接口相对较为低级，给你不同的函数去调用以实现不同的任务，而不像高级的工作流通常需要一个学校管道（如，将输入分为训练和测试数据，或者尝试很多的参数集合）。在 Spark1.2 中，MLib 获得了一个附加（写作本章时仍在实验）pipeLine API 去创建这个管道。这个 API 类似于像 Scikit-Learn 高级库，从而有望实现简单的去实现完整的、自调谐的管道。我们将在本章的末尾预览一下这个 API，但是我们主要还是集中（讲解）在这些低级的 API。

系统要求（System Requirements）
MLib 需要在你的机器上安装一些线性代数库（linear algebra libraries）。首先，你的操作系统上需要 gfortran 运行时库。如果 MLib 警告说缺失 gfortran，按照 MLlb website（http://bit.ly/1yCoHox）的安装说明（进行安装）。其次，用 Python 使用 MLib，你需要 NumPy。如果你的 Python 安装中没有它（或者你不能 import numpy），获取它的最简单的方法是在 Linux 上通过 package manager 安装 python-numpy 或者 numpy package，或者安装类似于 Anaconda 这样的第三方 python 科学安装包。
MLib 的支持也是随着时间进化的。我们在这里仅讨论的 Spark 1.2 上可行的算法，但是其中也有些算法在早期的版本中不存在。

机器学习基础（Machine Learning Basics）
在 context 中放入 MLib 的功能前，我们先来简单回顾一下机器学习的概念。
机器学习算法试着基于训练数据去做预测或决策，often maximizing a mathematical objective about how the algorithm should behave。有几类学习问题，包括分类、回归和聚类，每一个都有各自的目标。举个简单的例子，我们讨论下分类，它专注于确认一个元素属于几类中的哪一类（如一封电子邮件是垃圾邮件或非垃圾邮件），通过给其他的样本打标签（构建模型）作为依据。
所有的学习算法都需要为每个元素定义一套特征，（这些特征）将被喂给学习函数。如，对于一封邮件，一些特征可能包括这封邮件来自哪个服务器、或者提到过“free”这个词的次数、或者文字的颜色。在很多情况下，定义正确的特征是使用机器学习的最大挑战。例如，在一个产品推荐任务中，简单的添加一个特征（例如，意识到你应该推荐给使用者的书有可能取决于她看什么电影）有可能对结果造成很大的改进。
很多的算法被定义成只能使用数值特征（更具体的说，用一些数值代表每个特征的向量），所以，常常一个很重要的步骤是通过特征提取和转换（feature extraction and transformation）去构造这些特征向量。例如，对于文本分类（如，我们的垃圾邮件与非垃圾邮件例子），有几个方法去特征话文本，例如统计每个单词的频度。
一旦数据被表示成特征向量，大多数的机器学习算法都基于这些向量优化出一个定义良好的数学函数。比如，一个分类算法可能去定义一个最好的面（在特征向量空间中）去分割垃圾邮件与非垃圾邮件样例，（这个最好的面的建立）依据某些定义的最好（如，最多的点能被这个面能够正确进行归类）。最后，算法返回一个代表学习决策（如，选择了一个面（来划分不同类型的点））的模型。现在，这个模型能够被用来推测新的点（如，看看一个新邮件的特征向量会落到面的哪一边，从而判断它是否是垃圾邮件）。Figure 11-1 展示了一个学习管道样例。
P216

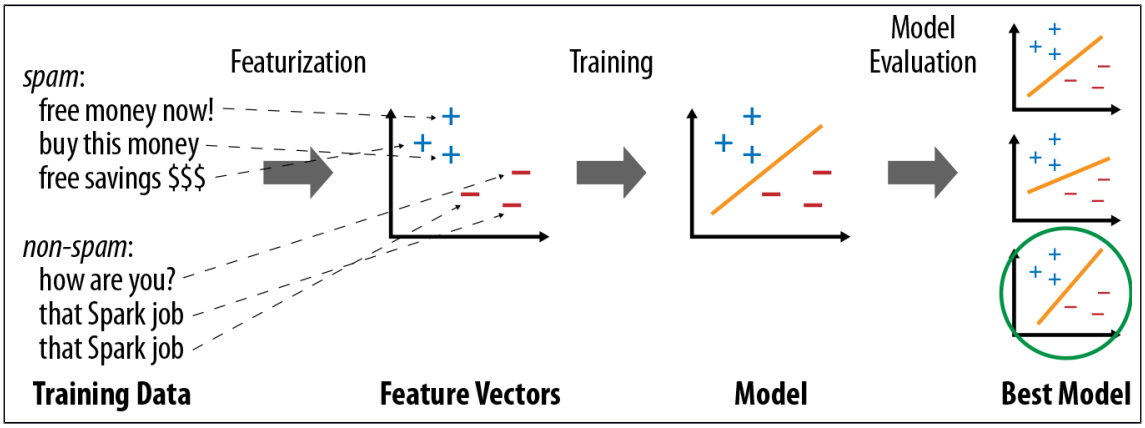

Figure 11-1. Typical steps in a machine learning pipeline

最后，大多数的学习算法有很多的参数能够影响到结果，所以真实的管道的一个模型可能会训练出很多的版本并且需要对每个版本进行评价。为了做这些，通常的方法是将输入数据分为“训练”集和“测试”集，“训练”集只用来建模，而“测试”集用来测试模型是否对“训练”数据“过拟合”（overfit）。MLib 提供了几个算法用于模型评价。

样例：垃圾邮件分类 （Example：Spam Classification）

作为 MLib 的一个快速样例，我们展示一个很简单的垃圾邮件分类器程序（从 Example 11-1 到 Example 11-3）。这个程序用到两个 MLib 的算法：HashingTF 算法用来从文本数据中构建检索词频率（term frequency）特征向量，Logistic RegressionWithSGD 算法用 chastic gradient descent（SGD 随机梯度下降法）实现了逻辑回归程序。我们假设从两个文件（spam.txt 和 normal.txt）开始，这两个文件包含了垃圾邮件和非垃圾邮件样本，每行一个。然后我们用 TF 将每个文件中文本转换成标签向量，在训练一个回归模型将信息分为两种类型。Code 和数据文件可以在本书的 Git repository 中得到。

Example 11-1. Spam classifier in Python

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.feature import HashingTF
from pyspark.mllib.classification import LogisticRegressionWithSGD
spam = sc.textFile("spam.txt")
normal = sc.textFile("normal.txt")
# Create a HashingTF instance to map email text to vectors of 10,000 features.
tf = HashingTF(numFeatures = 10000)
# Each email is split into words, and each word is mapped to one feature.
spamFeatures = spam.map(lambda email: tf.transform(email.split(" ")))
normalFeatures = normal.map(lambda email: tf.transform(email.split(" ")))
# Create LabeledPoint datasets for positive (spam) and negative (normal) examples.
positiveExamples = spamFeatures.map(lambda features: LabeledPoint(1, features))
negativeExamples = normalFeatures.map(lambda features: LabeledPoint(0, features))
trainingData = positiveExamples.union(negativeExamples)
trainingData.cache() # Cache since Logistic Regression is an iterative algorithm.
# Run Logistic Regression using the SGD algorithm.
```

```
model = LogisticRegressionWithSGD.train(trainingData)
# Test on a positive example (spam) and a negative one (normal). We first apply
# the same HashingTF feature transformation to get vectors, then apply the model.
posTest = tf.transform("O M G GET cheap stuff by sending money to ...".split(" "))
negTest = tf.transform("Hi Dad, I started studying Spark the other ...".split(" "))
print "Prediction for positive test example: %g" % model.predict(posTest)
print "Prediction for negative test example: %g" % model.predict(negTest)
```

Page 217

Example 11-2. Spam classifier in Scala

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.feature.HashingTF
import org.apache.spark.mllib.classification.LogisticRegressionWithSGD
val spam = sc.textFile("spam.txt")
val normal = sc.textFile("normal.txt")
// Create a HashingTF instance to map email text to vectors of 10,000 features.
val tf = new HashingTF(numFeatures = 10000)
// Each email is split into words, and each word is mapped to one feature.
val spamFeatures = spam.map(email => tf.transform(email.split(" ")))
val normalFeatures = normal.map(email => tf.transform(email.split(" ")))
// Create LabeledPoint datasets for positive (spam) and negative (normal) examples.
val positiveExamples = spamFeatures.map(features => LabeledPoint(1, features))
val negativeExamples = normalFeatures.map(features => LabeledPoint(0, features))
val trainingData = positiveExamples.union(negativeExamples)
trainingData.cache() // Cache since Logistic Regression is an iterative algorithm.
// Run Logistic Regression using the SGD algorithm.
val model = new LogisticRegressionWithSGD().run(trainingData)
// Test on a positive example (spam) and a negative one (normal).
val posTest = tf.transform(
"O M G GET cheap stuff by sending money to ...".split(" "))
val negTest = tf.transform(
"Hi Dad, I started studying Spark the other ...".split(" "))
println("Prediction for positive test example: " + model.predict(posTest))
println("Prediction for negative test example: " + model.predict(negTest))
```

Example 11-3. Spam classifier in Java

```
import org.apache.spark.mllib.classification.LogisticRegressionModel;
import org.apache.spark.mllib.classification.LogisticRegressionWithSGD;
import org.apache.spark.mllib.feature.HashingTF;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.regression.LabeledPoint;
JavaRDD<String> spam = sc.textFile("spam.txt");
JavaRDD<String> normal = sc.textFile("normal.txt");
```

```
// Create a HashingTF instance to map email text to vectors of 10,000 features.
final HashingTF tf = new HashingTF(10000);
// Create LabeledPoint datasets for positive (spam) and negative (normal) examples.
JavaRDD<LabeledPoint> posExamples = spam.map(new Function<String, LabeledPoint>() {
public LabeledPoint call(String email) {
return new LabeledPoint(1, tf.transform(Arrays.asList(email.split(" "))));
}
});
JavaRDD<LabeledPoint> negExamples = normal.map(new Function<String, LabeledPoint>() {
public LabeledPoint call(String email) {
return new LabeledPoint(0, tf.transform(Arrays.asList(email.split(" "))));
}
});
JavaRDD<LabeledPoint> trainData = positiveExamples.union(negativeExamples);
trainData.cache(); // Cache since Logistic Regression is an iterative algorithm.
// Run Logistic Regression using the SGD algorithm.
LogisticRegressionModel model = new LogisticRegressionWithSGD().run(trainData.rdd());
// Test on a positive example (spam) and a negative one (normal).
Vector posTest = tf.transform(
Arrays.asList("O M G GET cheap stuff by sending money to ...".split(" ")));
Vector negTest = tf.transform(
Arrays.asList("Hi Dad, I started studying Spark the other ...".split(" ")));
System.out.println("Prediction for positive example: " + model.predict(posTest));
System.out.println("Prediction for negative example: " + model.predict(negTest));
```

正如你所见，代码在所有的编程语言间都比较相似。在这个样例中，在包含 String 类型（普通文本）的 RDDs 上和 LabeledPoints（包含了特征和标签的一种 MLib 数据类型）上直接进行了操作。

数据类型（Data Types）

Mlib 包含了几个特别的数据类型，位于 org.apache.spark.mllib 包（Java/Scala）或 pyspark.mllib(Python)。主要如下：

向量(Vector)

一个数学向量。MLib 支持密集向量（每个记录都被存储）和稀疏向量（只有非零向量才被存储以便节省空间）。稍后我们将讨论这两种向量。可以用 mlib.linalg.Vectors 类来构造向量。

标记点（LabeledPoint）

一个标记数据点是为诸如分类和回归这样的监督学习算法（而准备的）。包括一个特征向量和一个标签（为浮点值）。它在 mlib.regression 包中。

等级（rating）

一个产品对某用户的等级，被用于产品推荐，被包含在 mlib.recommednation 包中。

各种模型类（Various Model classes）

每个训练算法的结果都有一个模型（model），且通常都有个 predict()方法将模型（model）用于新的数据点和包含了新点的 RDD。
大多的算法直接工作在包含了 Vectors, LabeledPoints, 或者 Ratings 的 RDD 上。如果你真的想你也可以创建这些对象，但是通常的做法是你通过转换外部的数据来创建 RDD——如加载一个文本文件或者运行 Spark SQL 命令——然后在请求一个 map()去将你的数据对象转换成 MLib 的类型。

使用向量（Working with Vectors）
在 MLib 中向量（vector）是最常用的，有限点需要注意：
首先、向量有两种风格：密集的和稀疏的。密集型向量将他们所有的条目存储在一个浮点数类型的 array。如，一个尺寸为 100 的向量将包含 100 个 double 类型的值。相比之下稀疏向量只存储非零值和它们的指数。如果至多 10%的元素为非零的话稀疏向量通常更优选（在内存使用或速度上都是）。许多的特色技术产生非常稀疏的向量，所以使用这种表示方法通常是个有效的优化方法。
其次，在不同的语言中构造向量的方法稍微有点不同。在 Python 中，你可以简单的在 MLib 的任何地方传递一个 NumPy array 去表示一个密集向量，或者用 mlib.linalg.Vectors 类去创建其他类型的向量（参看 Example 11-4）。在 java 和 scala 中，使用 mlib.linalg.Vectors 类（参看 Example 11-5 和 11-6）

Example 11-4. Creating vectors in Python

```
from numpy import array
from pyspark.mllib.linalg import Vectors
# Create the dense vector <1.0, 2.0, 3.0>
denseVec1 = array([1.0, 2.0, 3.0]) # NumPy arrays can be passed directly to MLlib
denseVec2 = Vectors.dense([1.0, 2.0, 3.0]) # .. or you can use the Vectors class
# Create the sparse vector <1.0, 0.0, 2.0, 0.0>; the methods for this take only
# the size of the vector (4) and the positions and values of nonzero entries.
# These can be passed as a dictionary or as two lists of indices and values.
sparseVec1 = Vectors.sparse(4, {0: 1.0, 2: 2.0})
sparseVec2 = Vectors.sparse(4, [0, 2], [1.0, 2.0])
```

Example 11-5. Creating vectors in Scala

```
import org.apache.spark.mllib.linalg.Vectors
// Create the dense vector <1.0, 2.0, 3.0>; Vectors.dense takes values or an array
val denseVec1 = Vectors.dense(1.0, 2.0, 3.0)
val denseVec2 = Vectors.dense(Array(1.0, 2.0, 3.0))
// Create the sparse vector <1.0, 0.0, 2.0, 0.0>; Vectors.sparse takes the size of
// the vector (here 4) and the positions and values of nonzero entries
val sparseVec1 = Vectors.sparse(4, Array(0, 2), Array(1.0, 2.0))
```

Example 11-6. Creating vectors in Java

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

```
// Create the dense vector <1.0, 2.0, 3.0>; Vectors.dense takes values or an array
Vector denseVec1 = Vectors.dense(1.0, 2.0, 3.0);
Vector denseVec2 = Vectors.dense(new double[] {1.0, 2.0, 3.0});
// Create the sparse vector <1.0, 0.0, 2.0, 0.0>; Vectors.sparse takes the size of
// the vector (here 4) and the positions and values of nonzero entries
Vector sparseVec1 = Vectors.sparse(4, new int[] {0, 2}, new double[]{1.0, 2.0});
```

最后，在 java 和 scala 中，MLib 向量类的主要意图是数据表示，而没有提供诸如加法和减法这样的算法操作的 API。（在 Python 中，你当然可以在密集向量上使用 NumPy 去实现数学运算然后将他们（密集向量）传给 MLib）。这样做是为了保持 MLib 轻便，因为实现一个完整的线性代数路超出了项目的范围。但是如果你想要在你的程序中做向量运算你可以使用第三方库（如 Scala 中的 Breeze（https://github.com/scalanlp/breeze），java 中的 MTJ（https://github.com/fommil/matrix-toolkits-java）），并从中将数据转换成 MLib 的向量。

算法（Algorithms）

在这段中，我们将浏览一下 MLib 的主要算法及他们的输入输出类型。我们没有空间去解释每个算法的数学原理。取而代之的是去关注如何调用和配置这些算法。