用R玩转Spark——大规模数据分析和建模完全手册

魏博

2020-01-09

Table of Contents

致Adrian，Clara，Julian，Max，Mila和Roman。

# 序

Apache Spark是一个基于可扩展性而构建的分布式计算平台：Spark的API使来自多个数据源的输入变得很容易，并且使用多种编程语言和算法来处理输入进而构建数据应用程序。R是数据科学和统计学中最强大的语言之一，因此将R与Spark结合起来是非常有意义的。幸运的是，R丰富的语言特性使得从R调用Spark的API与在本地数据源上运行R类似。有了这两个系统的背景知识，你就能够在Spark中调用大量计算，或者从最喜欢的R编程环境中并行运行R代码。

本书详细探讨了在R中使用Spark的方法，重点介绍了sparklyr包，它支持dplyr和R社区已有的其它包。本书详细涵盖了主要用例，从使用Spark引擎查询数据到探索性数据分析、机器学习、R代码的并行执行和流式处理。它还对运行Spark和监视作业执行提供了自含的介绍。而作者正是介绍这个话题的合适人员。Javier，Kevin和Edgar一直参与sparklyr开发项目。我很兴奋地看到他们把这本关于使用Spark和R的清晰而集中的指南编排得这么好。

我希望你也喜欢这本书，并通过它来提升你的R代码能力，并将其与更广泛的Spark生态系统的功能连接起来。而且因为这里的所有基础模块都是开源的，所以你可以毫不犹豫地向开发人员反馈改进这些工具的建议。

— Matei Zaharia 斯坦福大学助理教授，Databricks首先技术专家，Apache Spark发明者

# 前言

在信息呈指数级增长的世界中，Apache Spark等先进工具为我们解决今天面临的许多相关问题提供了支持。从基于数据驱动的决策寻找改进方法的公司，到解决医疗、金融、教育和能源问题的研究机构，Spark比以往任何时候都能更快、更可靠地分析更多信息。

人们编写了各种各样的书籍来学习Apache Spark。例如，《Spark: The Definitive Guide》是一本综合性的书籍，《Learning Spark》是一本介绍性的书籍，旨在帮助用户熟悉和运行Spark（两者都来自O’Reilly）。然而，在撰写本书的时候，既没有使用R语言学习Apache Spark的书籍，也没有专门为R用户或潜在的R用户设计的书籍。

有一些在线资源可以学习R语言中的Apache Spark。最著名的是Spark.rstudio.com网站和Spark.Apache.org网站上的Spark文档。这两个网站都是很好的在线资源；但是，上边的内容并不打算让你从初学者开始阅读。它们都假设读者对Apache Spark、R和集群计算有一定的了解。

这本书的目的是帮助所有人通过R使用Apache Spark。另外，由于R编程语言的目的是为了简化数据分析，我们相信这本书也可以提供学习用Spark解决数据分析问题的最简单途径。第1章提供了一个介绍，帮助快速理解这些概念，并介绍了在自己的计算机上处理这些问题所需的工具。然后，我们会快速进入相关的数据科学主题、集群计算和高级主题。即使是最有经验的用户也会感兴趣。

因此，本书旨在成为一个对广大用户有用的资源，从渴望学习Apache Spark初学者，到寻求理解为什么和如何通过R使用Apache Spark的有经验的读者。

本书的结构如下：

* 介绍 在前两章，第1章简介和第2章开始中，你将了解Apache Spark，R以及使用Spark和R执行数据分析的工具。
* 分析 在第3章分析中，你将学习如何使用R在Apache Spark中分析、探索、转换和可视化数据。
* 建模 在第4章建模和第5章管道中，你将学习如何创建统计模型，以便在生产环境的工作流中提取信息、预测结果和自动化过程。
* 规模化 到目前为止，本书的重点是在你的个人电脑上执行操作和接受有限的数据格式。第6章集群，第7章连接，第8章数据，以及第9章调试，介绍了跨多台机器和数据格式执行分析和建模所需的分布式计算技术，以解决Apache Spark关注的大规模数据和计算问题。
* 扩展 第10章扩展，描述了适用于特定用例的可选模块和扩展功能。你将学习其它建模框架、图处理、深度学习的数据预处理、地理空间分析和大规模基因组学。
* 高级技术 本书结尾是介绍一系列高级技术的章节，包括第11章分布式R，第12章Streaming，第13章社区贡献；这些内容是高级用户最感兴趣的。但是，当你学到本节时，你会发现其实并没有那么吓人；相反，这些章节将与前面的章节一样相关、有用和有趣。

第一组章节，1-5，简单地介绍了如何大规模执行数据科学和机器学习项目。如果你打算阅读这本书，同时也跟随代码示例，这些是可以逐行执行代码的好章节。因为这些章节介绍了个人电脑上涉及的所有概念。你不会利用多台电脑，而Spark却是为使用多台电脑而设计的。但是别担心：下一章将详细介绍这一点！

第二组章节，6-9，介绍了使用Spark进行集群计算中令人激动的基本概念。事实上，它们还介绍了集群计算中一些不那么有趣的部分。但是相信我们，这些概念还是值得学习的。此外，每章中的概述部分都特别有趣、信息丰富、易于阅读，有助于你直观地了解集群计算是如何工作的。对于这些章节，我们实际上并不建议逐行执行代码，尤其是在尝试从头到尾学习Spark的情况下。在拥有一个合适的Spark集群之后，你总是可以回过头来执行代码。但是，如果你已经有一个可以工作的集群，或者你真的很想得到一个集群，那么你可能需要借助第6章来选择一个集群，然后按照第7章来进行连接。

第三组章节，10-13，展示了对大多数读者来说应该很有趣的工具，并且会使后续的操作更容易。这里有很多高级主题，一些主题自然比其它主题更有意思。例如，你可能对分析地理数据集感兴趣，或者你对处理实时数据集更感兴趣，或者你希望两者兼得！基于个人兴趣或手头的问题，我们鼓励你执行最相关的代码示例。这些章节中的所有代码都是为了在个人计算机上执行而编写的，但我们也鼓励你使用合适的Spark集群，因为你会拥有解决问题和优化大规模计算所需的工具。

## 格式化

代码生成的表格类似于以下代码的格式：

# A tibble: 3 x 2  
numbers text  
<dbl> <chr>  
1 1 one  
2 2 two  
3 3 three

表的维度（行数和列数）在第一行中有描述，第二行是列名，第三行是列类型。我们在本书中使用的tibble包还提供了各种细微的视觉改进。

大多数绘图都使用ggplot2包和附录中提供的自定义主题进行渲染。但是，由于本书不关注数据可视化，因此我们只提供基本的绘图代码，其可能与应用的格式不匹配。如果你有兴趣学习R中更多关于可视化的知识，可以考虑专门的书籍，比如《R Graphics Cookbook》（O’Reilly）。

## 致谢

感谢那些允许通过R使用Spark的程序包作者：Javier Luraschi、Kevin Kuo、Kevin Ushey和JJ Allaire（sparklyr）；Romain franois和Hadley Wickham（dbplyr）；Hadley Wickham和Edgar Ruiz（dpblyr）；Kirill Mülller（DBI）；以及Apache Spark项目本身的作者及其最初作者Matei Zaharia。

感谢为丰富Spark和R生态系统而发布扩展的程序包作者：Akhil Nair（crassy）；Harry Zhu（geospark）；Kevin Kuo（graphframes、mleap、sparktf和sparkxgb）；Jakub Hava、Navdeep Gill、Erin LeDell和Michal Malohlava（rsparkling）；Jan Wijffels（Spark.sas7bdat）；Aki Ariga（sparkavro）；Martin Studer（sparkbq）；Matt Pollock（sparklyr.nested）；Nathan Eastwood（sparkts）；Samuel Macêdo（variantspark）。

感谢我们出色的编辑Melissa Potter为我们提供指导、鼓励和无数小时的详细反馈，使这本书成为我们的著作中最好的一本。

感谢Bradley Boehmke，Bryan Adams，Bryan Jonas，Dusty Turner和Hossein Falaki，感谢你们的技术评论、及时和坦率的反馈，并与我们分享专业知识。多亏了你们，许多读者才会有更愉快的阅读体验。

感谢RStudio、JJ Allaire和Tareef Kawaf对这项工作的支持，感谢R的持续支持和鼓励。

Max Kuhn，感谢你对第4章的宝贵反馈。在他的允许下，我们采用了他的精彩著作《特征工程与选择：预测模型的实用方法》（CRC出版社）中的例子。

我们也感谢所有间接参与但未在本节中明确列出的人们；我们真正站在了巨人的肩膀上。

这本书是用Yihui Xie的bookdown，JJ Allaire和Yihui Xie的rmarkdown，Yihui Xie的knitr写成的；用Hadley Wickham和Winston Chang的ggplot2完成可视化；用Daniel Kallin和Javier Luraschi的nomnoml绘制图表；用John MacFarlane的pandoc进行文档转换。

## 行文格式

本书使用下列排版风格：

*斜体* 表示新的术语，URL，电子邮箱地址，文件名，和文件扩展名。

固定宽度 用于表示程序内容以及段落中引用的程序元素，如变量或函数名、数据库、数据类型、环境变量、语句和关键字。

**重体固定宽度** 展示用户输入的命令或其它文本。

*斜体固定宽度* 展示应该被用户提供的，或由上下文确定的值替换的文本。

**用来表示注意事项或者建议。**

用来表示一般的注释。

## 使用示例代码

补充材料（示例代码、练习等）可在https://github.com/r-spark/the-r-in-spark上下载。

这本书是来帮你完成工作的。一般情况下，如果本书提供了示例代码，你就可以在你的程序和文档中使用。你不需要为了获得许可联系我们，除非你正在复制代码的重要部分。例如，编写一个使用本书中多段代码的程序不需要获得我们的许可。出售或分发O’Reilly书籍的示例光盘就需要获得许可。引用本书以及示例代码来回答问题也不需要许可。将本书中大量的示例代码合并到你的产品文档中就需要获得许可。

虽然我们会感谢你的引用，但是我们也不强制要求你这么做。引用通常包括标题、作者、出版商和ISBN。例如：“Mastering Spark with R by Javier Luraschi, Kevin Kuo, and Edgar Ruiz (O’Reilly). Copyright 2020 Javier Luraschi, Kevin Kuo, and Edgar Ruiz, 978-1-492-04637-0.”

如果你觉得你对代码示例的使用超出了合理使用或上述许可范围，[请通过permissions@oreilly.com与我们联系](mailto:请通过permissions@oreilly.com与我们联系)。

# 第1章 引言

你什么都不知道，Jon Snow。

— Ygritte

随着信息以指数的速率增长，历史学家把这段历史时期称作信息时代也就不意外了。逐步加速的数据收集速度也创造了很多新的机会，并且必然会创造更多的机会。本章介绍一些用于解决大规模数据挑战的工具。首先，作为处理大规模数据集的流行工具，本章会介绍Apache Spark。有了这一基础，我们会介绍R编程语言，它是专门用于简化数据分析的。最后，我们会介绍sparklyr，该项目将R和Spark结合成为大家都可使用的强大工具。

*第2章 开始*会介绍用于在个人电脑上执行Spark和R运算所需的预备知识，工具和步骤。你会学到如何安装和初始化Spark，了解常见操作，处理第一份数据和完成建模任务。本章的目的就是要帮助读者掌握处理大规模数据问题所需的概念和工具，而这些问题目前只有少数组织可以接触到。

然后，你可以继续学习如何分析大规模数据，以及构建模型支持预测趋势和发现隐藏在海量信息中的知识。接着，你可以掌握一些用于执行数据分析和大规模建模的工具。后续章节会帮助你从本地计算机中跳出来，转向计算集群，以便解决许多真实世界的问题。最后几章会介绍一些其它的话题，例如实时数据处理和图分析，可以帮助你真正的掌握任何规模数据分析的最先进技能。本书的最后一章会介绍一些工具和想法，读者可以考虑在Spark和R社区中做些贡献。

我们希望，阅读本书能成为一种愉悦的经历，可以帮助你解决职业生涯的数据问题，并推动世界作出有利于所有人的更好决策。

## 概述

人类大约从公元前3000年苏美尔人时期就开始用美索不达米亚成熟的书写体系保存，获取，操作和沟通信息了。根据使用的保存和处理技术，人类的信息处理可能分成四个不同的发展时期：前机械工具时期（公元前3000至公元后1450年），机械工具时期（1450–1840），电子机械时期（1840–1940），和电子时期（1940至今）[[1]](#footnote-29)。

数学家George Stibitz在1942年使用词语digital来描述快速的电子脉冲[[2]](#footnote-30)。直到今天，我们把用电子格式存储的信息还叫做数字信息。相反，模拟信息表示用非电子格式存储的各种信息，例如手写笔记，书籍，报纸等。

世界银行发布的关于数字发展的报告给出了过去三十年保存下来的数字信息和模拟信息总量估计[[3]](#footnote-31)。这份报告指出，数字信息大概在2003年超过了模拟信息，其总量有大约一千万太字节，大约相当于今天的一千万存储驱动。然而，这份报告中更相关的发现是，我们的数字信息正在以指数的速度增长。图1-1展示了这份报告的结论。注意，每隔一年，世界上的信息都会增长10倍。

为了提供工具找出所有新的数字信息，许多公司都试图提供相关的功能。例如，我们今天熟知的用于搜索网络的搜索引擎。给定海量的数字信息，管理如此规模的信息绝对是一项挑战。搜索引擎并不能存储所有的网页信息，支持在单一电脑上搜索网页。这就意味着它必须把信息分成几个文件，并存储在多个机器上。这项技术就是*谷歌文件系统（Google File System）*，它在谷歌2003年发表的学术文献中有介绍[[4]](#footnote-32)。

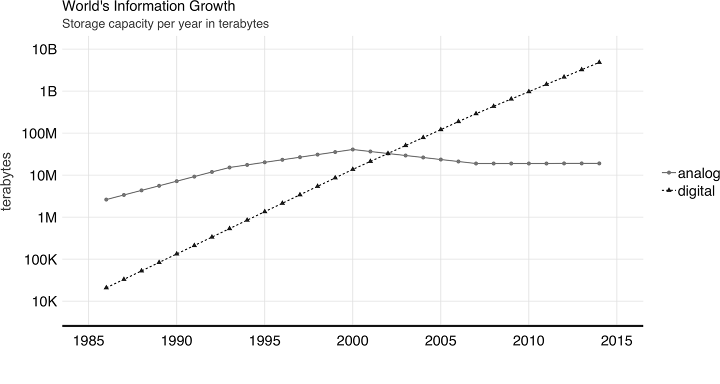


图1-1. 全球存储信息能力

## Hadoop

一年以后，谷歌发表了新的文章，描述了如何在谷歌文件系统上执行操作，也就是后来的*MapReduce*技术。[[5]](#footnote-35)。正如你可以预见到的，MapReduce中有两种操作：map和reduce。*map*操作提供任意一种方法把每个文件转化为一个新的文件，而*reduce*操作可以合并两个文件。两种操作都需要定制计算机代码，而MapReduce框架关注一次性在多个电脑上的自动执行。这两个操作足够支持网络上所有数据的处理，同时也提供足够的灵活性从数据中得到有意义的信息。

例如在图1-2中，我们可以使用MapReduce计算存放在不同机器上的两个不同文本文件的词语数量。map操作切分原始文件中的每个词语，并输出一个新的词语计数文件，其包含词语到计数的映射。reduce操作可以定义为对两个词语计数文件的合并，其通过整合每个词语的计数总数来实现。最终的文件会包含来自所有原始文件的词语计数。

词语计数通常是最基础的MapReduce示例，但是我们也可以使用MapReduce完成更加复杂和有趣的应用。例如，我们可以在谷歌的PageRank算法中对网页排序，该算法可以基于链接到一个网页的超链接数对所有网页排序，以及给出该网页的排位。

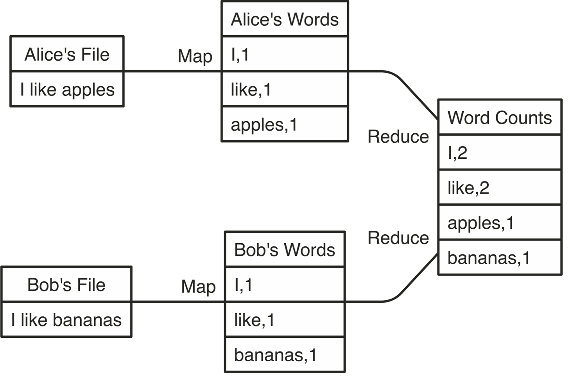


图1-2. 跨多个文件的词语计数MapReduce作业示例

在谷歌发表了这些文献之后，雅虎的一个团队致力于将谷歌文件系统和MapReduce作为一个单一的开源项目。这个项目就是2006发布的*Hadoop*，其中谷歌文件系统通过*Hadoop分布式文件系统（Hadoop Distributed File System，HDFS）*实现。Hadoop项目使得基于分布式文件的计算可以被更多的用户和组织获取，这也使得MapReduce的能力突破了网络数据处理的限制。

尽管Hadoop提供了在分布式文件系统上执行MapReduce操作的支持，但是它依然需要在每次数据分析的时候写好MapReduce操作代码。为了改进这一繁琐的过程，Facebook于2008年发布了Hive项目。它给Hadoop引入了*结构化查询语句（Structured Query Language，SQL）*。这意味着大规模数据分析不再需要为每个MapReduce操作编写代码。相反，用户可以用SQL编写通用的，更加容易理解和编写的数据分析语句。

## Spark

2009年，Apache Spark开始作为加州大学伯克利分校AMP实验室的科研项目，以期改进MapReduce。Spark针对性的提供了更加丰富的MapReduce以外的运算符集合，便于优化多个机器上的代码运行。Spark同时将数据加载到内存中，使得各种操作要比Hadoop的磁盘存储快很多。最新的一项结果显示，运行逻辑回归，一种在第4章中介绍的数据建模技术，利用内存中数据集的Spark要比Hadoop快10倍[[6]](#footnote-38)。类似于图1-3的结果在原始研究文献中也有展示。

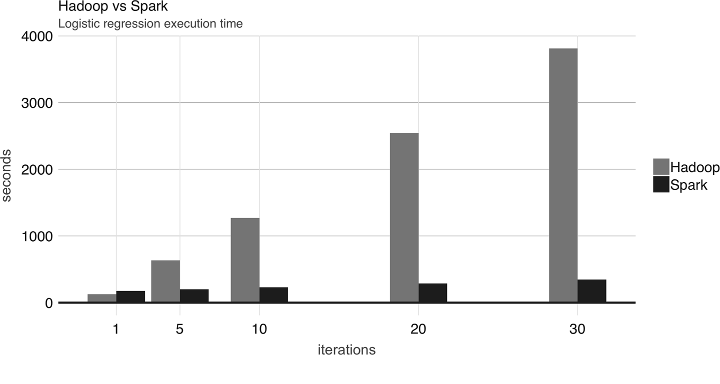


图1-3. 逻辑回归在Hadoop和Spark上的性能

尽管Spark以内存操作的性能而闻名，而当初设计时是作为支持内存和磁盘操作的一种通用执行引擎。例如，Spark创造了新的大规模排序记录，而这些数据并没有存在内存中。相反，Spark改进了网络序列化，网络改组，CPU缓存的高效利用，进而极大的提高了性能。如果你需要对大规模数据进行排序，世界上没有比Spark更快的系统了。

为了让你了解Spark有多快多高效，Hadoop使用了72分钟，对2100台计算机完成了100太字节数据的排序，但是Spark只用了23分钟和206台计算机。另外，Spark拥有云端排序记录，使其成为云端对大规模数据进行排序的最具性价比的解决方案。

|  |  |  |
| --- | --- | --- |
|  | Hadoop记录 | Spark记录 |
| 数据规模 | 102.5TB | 100TB |
| 使用时间 | 72分钟 | 23分钟 |
| 节点数量 | 2,100 | 206 |
| 内核数量 | 50,400 | 6,592 |
| 磁盘 | 3,150GB/秒 | 618GB/秒 |
| 网络 | 10GB/秒 | 10GB/秒 |
| 排序速度 | 1.42TB/分钟 | 4.27TB/分钟 |
| 排序速度/节点 | 0.67GB/分钟 | 20.7GB/分钟 |

Spark也比Hadoop易用。例如，词语计数的MapReduce示例在Hadoop中需要50行代码，而在Spark中只需要2行代码。正如所见，和Hadoop相比，Spark更快，更高效，更易用。

2010年，Spark作为开源项目发布出来，并于2013年捐赠给了Apache软件基金会。Spark执行Apache 2.0协议，允许用户免费使用，修改和分发。Spark有超过1,000名贡献者，是Apache软件基金会中最活跃的项目。

下面的表述介绍了Spark的定位。我们可以按照项目网站那样正式介绍Apache Spark：

*Apache Spark是一个面向大规模数据处理的统一分析引擎。*

为了帮助我们理解Apache Spark的概念，我们把它分成如下几个方面：

* 统一
* Spark支持许多程序包，集群技术和存储系统。
* 分析
* 分析是发现和理解数据，以期产出和沟通信息的行为。
* 引擎
* Spark是高效的和通用的。
* 大规模
* 你可以把大规模理解成集群规模，一些连接一起的，共同工作的计算机。

Spark可以看做是一种*引擎*，因为它是通用的和高效的。它的通用性是因为Spark可以优化和执行通用代码。也就是说，对于Spark中的程序代码并没有什么类型限制。它的高效性是因为Spark可以有效利用内存，网络和CPU加速计算集群中的数据处理算法。正如之前提到的，它比其它技术都要快速。

这些特点使得Spark在许多分析任务中，例如Netflix的电影排序，蛋白质序列对齐，和CERN的高能物理分析中，成为理想工具。

作为一种*统一*的平台，Spark可以支持许多集群技术和多种数据源。你可以分别在第6章和第8章中了解到。它也支持许多不同的库，例如Spark SQL，MLlib，GraphX和Spark Streaming，分别支持分析，建模，图处理和实时数据处理。总之，Spark是一种支持请求集群，数据源和各种库的平台，进而实现大规模计算，如图1-4所示。

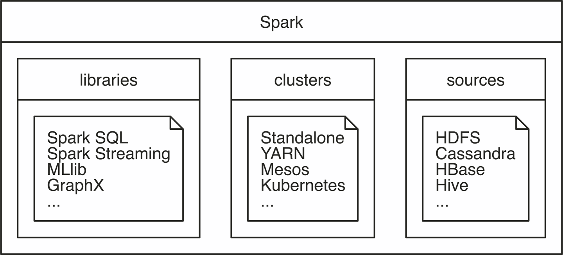


图1-4. Spark，面向大规模数据处理的统一分析引擎

Spark支持大规模运算，意味着Spark的良好用例应该是使用多台机器解决问题。例如，当数据无法在单个磁盘或内存中存放，Spark会是一种良好的备选方案。然而，你也可以在面对非大规模问题的时候考虑使用Spark。此时，多台计算机可以加速计算。例如，CPU密集型的模型和科学模拟也可以从Spark中受益。

因此，Spark不但擅长解决大规模数据处理问题，即通常所说的*大数据*（数据集比传统的要更加庞大和复杂），而且擅长解决大规模计算问题，即通常所说的*大计算*（协调使用大量CPU和内存资源的工具和技术）。大数据经常需要大计算，但是*大计算不一定需要大数据*。

大数据和大计算问题通常很容易识别。如果数据不能存放在单个机器，你就遇到了大数据问题；如果数据可以存放在单个机器，但是数据处理却需要数天，数周甚至更长的时间来完成，你就遇到了大计算问题。

但是，还有第三类问题，既不是大数据问题，也不是大计算问题，它们也可以从类似Spark的集群计算框架中受益良多。这类问题，有以下几种场景：

* 高速性
* 假设你有一个10GB的数据集，它的数据处理过程需要30分钟。这既不是一个大数据问题也不是一个大计算问题。但是，如果你突发奇想要改进模型的准确率，减少运行时间到3分钟以内，这就是一个显著的改进。这种数据分析速度的改进可以带来有意义的推进和生产力的提高。另外，你也可能需要更快的处理数据速度，比如支持股票交易。尽管三分钟已经足够快，但是这对于实时数据处理依然很慢。这种场景下，你可能需要在几秒内甚至几毫秒内完成数据处理。
* 多样性
* 你可能有了一个高效的数据收集流程，从多个数据源收集数据放在一个位置，通常是一个数据库。这个过程可能已经非常高效并且接近实时。这样的过程就是通常所说的*抽取，转换和加载（Extract，Transform，Load，ETL）*；数据从多个数据源抽取，转换成所需的格式，加载到单个数据存储结构中。尽管这一方法已经流行了多年，但是其中的权衡是添加新的数据源代价很高。因为系统是集中控制的，并且是严密操作的，任何改变都可能使整个过程暂停；因此，添加新的数据源通常要花费很长的时间才能实现。相反，你可以把全部数据按照自然格式存放，并且按照需要使用集群处理，即数据湖的架构。另外，按照自然格式存放数据支持处理图片，声音，和视频格式的新文件，而不需要明确如何把它们放进传统的结构化存储系统。
* 真实性
* 当使用多个数据源的时候，你可能发现它们之间的数据质量参差不齐，需要专门的分析方法来改进准确性。例如，假设你有一个城市列表，取值包括旧金山，西雅图，和波士顿。当数据包含拼错的单词，例如“Bston”，会发生什么呢？在传统的关系型数据库中，这类非法输入可能会被舍弃。但是，舍弃取值并不一定在所有情形下都是最好的方法；你会希望利用地理编码，交叉验证数据源，或者尽最大努力匹配来纠正这些字段。所以，理解和改进原始数据源的真实性可以带来更准确的结果。

如果我们把“海量性”当做是大数据的同义词，你就得到了方便记忆的大数据的4个V。一些人也把它扩展成5个V甚至10个V。除了几个便于记忆的符号，今天集群计算也正在以更加创新的方式使用。看到各个组织实验新的工作流，以及实验传统意义上并不适用集群计算的各种任务，已经不再稀奇。更多的源于大数据的热潮都是这类问题。严格意义上讲，你不是在解决大数据，但是依然可以从大数据和大计算工具的使用中获益。

我们希望本书可以帮助你理解集群计算的机遇和局限，特别是使用Apache Spark和R的机遇和局限。

## R

R编程语言源于诞生在贝尔实验室的S语言。Rick Becker在2016年的useR!会议上解释，在当时的贝尔实验室里，计算是通过调用Fortran语句写成的子程序而实现的，而这种程序显然不好处理。而S计算语言当时是作为一种接口语言而设计的，它可以解决特殊的问题，而不需要使用其它语言，例如Fortran。S语言的发明者，John Chambers在图1-5中介绍了S是如何设计出来的，进而提供简化数据处理的接口。它的共同发明者也在2016年的useR!上展示了这一原图，启发了S的设计。

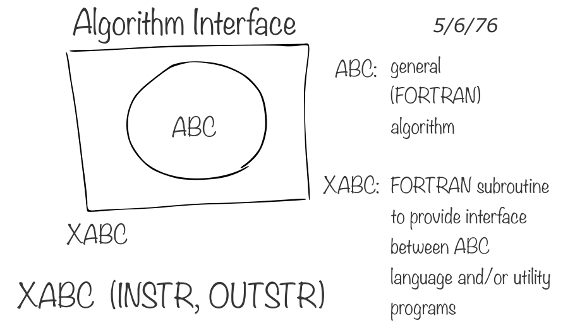


图1-5. John Chambers的接口语言图 (Rick Becker useR! 2016)

R语言是S语言的一种现代而自由的实现。特别的，根据《R Project for Statistical Computing》：

*R是一种面向统计计算和图形学的编程语言和免费软件。*

处理数据的时候，我们相信R语言有两大特点：

* R语言
* R是由统计学家设计的，也是面向统计学家使用的。这意味着R是少有的，成功的面向非编程人员的编程语言之一。所以学习R会更加自然。另外，因为R语言是其它语言和工具的接口，所以R可以让你更多的关注理解问题，而更少的花在计算机科学和工程上。
* R社区
* R社区通过*R语言综合文档网络（Comprehensive R Archive Network，CRAN）*提供了丰富的程序包，并支持安装即时可用的程序包，来执行各种任务——最享有盛名的高质量数据操作，可视化，和统计模型。许多操作只能在R中进行。另外，R社区是一个热情的，活跃的组织，这些优秀的朋友都愿意帮助你成功。R社区提供的许多程序包，到目前为止，都是统计计算的最优选择。一些下载最多的程序包包括：操作数据的dplyr，分析聚类的cluster，可视化数据的ggplot2。通过画出每天CRAN里R程序包的下载量，图1-6定量刻画了R社区的增长。

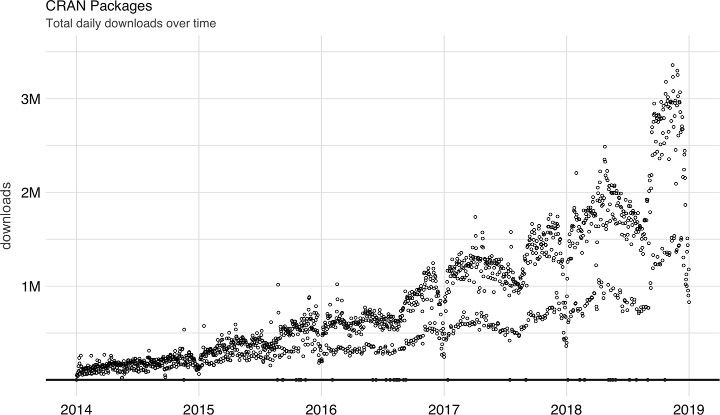


图1-6. CRAN程序包每日下载量

除了统计分析，R也用在其它领域。以下领域和本书高度相关：

* 数据科学
* 数据科学基于来自统计学和计算机科学的知识和实践，通过数据分析和建模技术把原始数据变成深刻理解[[7]](#footnote-44)。统计学的方法提供了理解世界和进行预测的坚实基础，而计算机模型中的自动化技术可以让我们简化统计分析并易于实现。一些人认为，统计学应该重新命名为数据科学[[8]](#footnote-45)，然而数据科学结合了计算机的进展，并超出了统计学的范畴[[9]](#footnote-46)。本书给出了统计学中常见的分析和建模技术，但是会应用到大数据集上，而这又需要分布式计算的成果。
* 机器学习
* 机器学习利用来自于统计学和计算机科学的实践。但是，它更多的关注自动化和预测。例如，Arthur Samuel在实现跳棋的计算机自动化程序的时候发明了词语machine learning[[10]](#footnote-47)。尽管我们可以在专门的游戏上实践数据科学，但是编写一个跳棋的程序需要我们自动化整个过程。因此，这个问题就是机器学习的范畴，而不是数据科学的范畴。机器学习使得许多用户在无意识中利用统计学的方法成为可能。机器学习中最重要的一个应用是过滤垃圾邮件。在这个例子中，只做数据分析和对每个邮箱账户建模是不可行的，因此机器学习可以自动化发现垃圾邮件的整个过程，并在用户没有参与的情况下过滤出去。本书给出了把数据科学工作流转换为完全自动化的机器学习的方法，例如，支持构建和导出Spark流程，以便容易的在自动化环境中复用此流程。
* 深度学习
* 深度学习建立在统计学，数据科学和机器学习知识的基础上，并受到生物神经系统的些许启发而定义模型。当梯度散失问题通过一次一层的训练解决后，深度学习就从神经网络模型逐步进化了[[11]](#footnote-48)，并被证明在图像识别和语音识别任务中非常强大。例如，在语音助手应用中，诸如Siri，Alexa，Cortana，或谷歌助手，实现语音到文本转换的模型大部分都是基于深度学习模型。尽管*图形处理器（Graphic Processing Units，GPUs）*已经成功的用在了训练深度学习过程中[[12]](#footnote-49)，但是一些数据集并不能在单个GPU上处理。另一个事实是，深度学习模型需要大量的数据。这也需要在多个机器上预处理一下，然后才交给单个GPU训练。本书并不会直接给出深度学习模型的任何参考文献。但是你可以使用我们在本书中给出的方法，准备深度学习数据。在未来几年里，使用深度学习大规模计算技术会变得非常普遍。事实上，Spark的最新版本已经加入了为深度学习训练而优化过的模型。

处理上述几个领域时，你要么会面对逐渐增大的数据集，要么会面对逐渐复杂的计算过程。这些流程执行起来都很慢，或者某些时候在单台计算机上变得不可能。但是，要知道，Spark并不一定是所有计算问题的答案。面对R中的计算问题时，使用以下技术会很有效：

* 采样
* 第一个要尝试的技术是通过采样，减少需要处理的数据量。然而，我们必须使用可靠的统计学原理，合适地采样数据。例如，选取有序数据集的头部数据是不够的。使用随机采样，会得到占比较少的数据集。这些都可以通过分层采样克服，但是选取合适的类别增加了算法复杂度。讲解如何进行合适的统计采样超出了本书的范畴，但是有很多有用的资源都是关于这个话题的。
* 代码分析
* 你可以自己试着理解为什么计算很慢，并且需要改进。分析器是检查代码执行情况，帮助识别瓶颈的工具。在R中，R的分析器，profvis程序包和RStudio的分析器功能可以让你轻松的重试和可视化一个分析器。然而，优化代码通常并不容易。
* 纵向扩展
* 加速计算通常可以通过购买更快或者更强劲的硬件（即提高机器内存，升级硬盘，或者使用CPU更多的计算器）来实现。这些手段叫做纵向扩展。但是，对于单个计算机的纵向扩展通常存在硬极限，即使使用大量的CPU，你还需要新的框架来有效地并行化计算。
* 横向扩展
* 最后，我们可以考虑把计算和存储扩大到多台机器上。这种手段提供了最高维度的扩展性能。因为你可以使用任意数量的机器来执行计算。这个技术通常叫做横向扩展。但是，有效的扩展计算是一项复杂的实践，特别是没有使用专门的工具和框架，例如Apache Spark的时候。

最后一点更加具体的介绍了本书的目的，即Apache Spark提供的分布式计算系统的能力，借助R解决数据科学和相关领域中有意义的计算问题。

## sparklyr

考虑到Spark提供的计算能力和R语言的易用性，我们很自然的想让二者无缝的结合起来。这也是R社区期望的：提供Spark接口的R程序包用起来很简单，并与其它R程序包兼容，而且可以在CRAN上找到。为此，我们开始开发sparklyr。第一版，sparklyr 0.4，在2016年的useR!大会上发布。第一版包括对dplyr，DBI，建模工具MLlib的支持，以及支持类似H2O的rsparkling程序包的可扩展API。 从此，许多新的功能和改进在sparklyr 0.5, 0.6, 0.7, 0.8, 0.9和1.0做了实现。

根据官方定义，sparklyr是Apache Spark的R语言接口。它可以像其它CRAN程序包一样工作通过CRAN获取。这意味着人们无需知道Spark的版本，就可以完成安装，同时服务于R社区，兼容其它R程序包，并接受来自R社区的使用反馈，等等。它托管在GitHub，遵循Apache 2.0协议，允许克隆，修改，并用户参与项目开发。使用sparklyr的用户应该包含以下角色：

* 新用户
* 对于新用户，我们坚信sparklyr提供了使用Spark的最简单的方式。我们希望本书的前面几章可以帮助你逐渐熟悉并毫无压力的使用Spark，进而为长期任务做准备。
* 数据科学家
* 对于已经使用和钟爱R的数据科学家，sparklyr整合了许多其它的R实践和程序包，例如dplyr，magrittr，broom，DBI，tibble，rlang，以及其它框架，让你使用Spark从容自如。而对于R和Spark的新手，sparklyr的高级别工作流整合和低级别扩展机制保证Spark是一个高产的环境，可以满足每个数据科学家的需求和技能。
* 专家用户
* 对于已经深度使用Spark，并可以编写Scala原生代码的用户，可以考虑让你的Spark程序包作为一个R程序包贡献给R社区。一个多样的，强大的社区可以让你的工作贡献出最好的价值，促进开源社会的发展。

我们编写这本书是为了介绍和传授Apache Spark和R融合的部分。sparklyr就是把这些社区，期望和未来方向，程序包及其扩展整合在一起的R程序包。我们相信，使用这本书可以搭起R和Spark社区的桥梁：给R社区介绍为什么Spark让人兴奋，给Spark社区介绍为什么R如此伟大。两个社区都在使用不同的技术和背景来解决相似的问题。因此，我们希望sparklyr成为创新的肥沃土壤，新手的好客之地，是数据科学达人的高产环境，也是集群计算，数据科学，机器学习一起交流的开放社区。

## 小结

本章介绍了Spark这一现代而强大的计算平台，R这一易用的，基于扎实的统计建模理论的计算语言，以及sparklyr这一连接两个技术和社区的项目。我们处在一个信息指数级增长的世界，学习如何分析大规模数据可以帮助你解决当今人类面临的问题和机遇。但是，在分析数据之前，第2章会介绍本书后面内容所需的工具。确保你可以明白每一步操作，并花点时间安装推荐的工具，因为它们会成为你经常使用和喜爱的资源。

# 第2章 开始

我一直想成为一个巫师。

— Samwell Tarly

完成第1章的学习，你应该熟悉了Spark可以解决的各类问题。显而易见，Spark利用多台计算机来解决数据无法在单个机器上运行或者计算太慢的问题。如果你是R的新手，也可以看到，把Spark和数据科学工具，例如可视化工具ggplot2，数据变换dplyr，结合起来可以给大规模数据科学项目带来美好的想象。我们也希望你因为自己成为大规模数据计算的高手而感到兴奋。

在本章中，我们会简单介绍一下成为Spark高手所需要的工具。我们鼓励你能够实现本章的代码，因为这样会让你理解分析，建模，读取和写入数据的各种动作。换句话说，你需要坚持不懈，在使用Spark前反复实践。

在第3章中，我们会深入介绍分析和建模，并给出在单个集群上，也就是你的个人电脑上运行的例子。后续章节介绍集群计算，以及成功在多台机器上运行代码所需的概念和技术。

## 概述

R使用sparklyr来打通Spark和本地集群，整个过程和安装加载sparklyr程序包一样简单，后面安装Spark也通过sparklyr完成。但是，我们假设你在使用一个全新的电脑运行Windows，macOS或者Linux，因此我们会在连接本地Spark集群前，完成预备操作。

尽管本章是为了在个人电脑上使用Spark做准备，一些读者可能已经有了Spark集群，或者喜欢使用在线Spark集群。例如，Databricks托管了一个免费的社区版的Spark，你可以很容易的从浏览器访问。如果你最终选择了这条路，可以跳过“预备操作”。但是要确保你查阅了合适的资源，支持已有的或在线的Spark集群。

完成预备操作后，你首先要学习如何连接Spark。我们会介绍本书后续章节中用到的最重要的工具和操作。我们不会过多关注教授概念或者如何使用概念。我们也不可能在一章中解释完建模或流式计算。但是，完成本章的学习让你窥见到Spark的能力，以及确保工具已经完成正确的部署，足以应付未来更多棘手的问题。

所用的工具大部分都分成R代码和Spark网络接口。所有Spark操作都在R中运行。但是，监控分布式操作的行文会在Spark的网络接口中操作，你可以通过任何一个网络浏览器加载看到。然后我们断开本地集群的连接。虽然这很容易忘记，我们还是强烈建议在使用本地集群和共享集群的时候能够记得这一操作.

本章末尾会介绍一些功能，使得在Rstudio中操作Spark更容易。更具体的，我们会介绍由sparklyr实现的RStudio扩展。但是，如果你喜欢使用Jupyter Notebooks或者你的集群已经装好了别的R用户界面，确保你可以通过纯R代码使用Spark。现在，让我们开始学习预备操作。

## 预备操作

R可以在许多平台和环境中运行。所以，不管你用Windows，Mac，还是Linux，第一步都需要从r-project.org安装R。具体信息在“安装R”中有说明。

大部分人在使用编程语言时会使用工具，以便让编程更加高效。对于R，RStudio就是这样一个工具。严格地说，RStudio是一个*集成开发环境（integrated development environment，IDE）*，现在它也支持许多平台和环境。如果还没有安装Rstudio，我们强烈推荐你安装，具体操作见“安装RStudio”。

**使用Windows的时候，我们建议不要使用路径上有空格的目录。如果运行getwd()返回了带有空格的路径，可以使用setwd("path")换成不带空格的路径，或者把RStudio项目创建到没有空格的路径中。**

另外，因为Spark是用Scala实现的，运行在Java虚拟机（Java Virtual Machine，JVM）上，所有你还需要给系统安装Java 8。你的系统可能装了Java，但是还需要检查版本，按照“安装Java”的内容升级或降级。你可以使用如下R命令检查安装在系统的版本：

system("java -version")  
java version "1.8.0\_201"  
Java(TM) SE Runtime Environment (build 1.8.0\_201-b09)  
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)

你也可以运行Sys.setenv(JAVA\_HOME = "path-to-java-8")，使用JAVA\_HOME环境变量来指定具体的Java版本。也可以在安装sparklyr之前，确保R环境可以使用Java 8版本。

### 安装sparklyr

和许多其它的R程序包一样，你可以使用如下命令安装CRAN上的sparklyr：

install.packages("sparklyr")

本书的示例假设你使用最新版本的sparklyr。你可以运行如下代码，验证你的版本是不是和我们的一样新：

packageVersion("sparklyr")  
[1] '1.0.2'

### 安装Spark

使用sparklyr安装:

library(sparklyr)

这个代码可以实现R中所有sparklyr函数，非常有用。否则，你需要带着前缀sparklyr::运行每一个sparklyr命令。

你可以简单的通过运行spark\_install()安装Spark。这个命令可以在本地电脑上下载，安装和配置最新版本的Spark。但是，因为我们这本书是根据Spark 2.3编写的，所以你也可以安装这个版本，并顺畅的运行所有示例：

spark\_install("2.3")

你可以运行如下命令，输出所有可以安装的Spark版本：

spark\_available\_versions()  
## spark 1 1.6 2 2.0 3 2.1 4 2.2 5 2.3 6 2.4

你可以使用Spark版本信息安装特定的版本，或者也可以指定Hadoop版本。例如，要安装Spark 1.6.3，你可以运行：

spark\_install(version = "1.6.3")

你也可以使用如下命令，检查已经安装的版本：

spark\_installed\_versions()  
spark hadoop dir  
7 2.3.1 2.7 /spark/spark-2.3.1-bin-hadoop2.7

Spark的安装路径通常是Spark的主目录，可以通过R代码和系统配置的SPARK\_HOME标识符来定义。当你使用安装有sparklyr的本地Spark集群时，这个路径是已知的，不需要额外的配置。

最后，要卸载具体版本的Spark，你可以指定Spark和Hadoop的版本信息，运行spark\_uninstall()，如下：

spark\_uninstall(version = "1.6.3", hadoop = "2.6")

默认的安装路径是：macOS和Linux的~/spark，Windows的%LOCALAPPDATA%/spark。要定制化安装路径，可以在spark\_install()和spark\_connect()之前运行options(spark.install.dir = "installation-path")。

## 连接

到目前为止，我们只安装了本地集群。本地集群对于初始学习，测试代码，轻松的排查故障非常有好处。后面的章节会介绍在哪里发现，安装和连接真正的有多台机器的Spark集群。但是对于开始的几章，我们只会使用本地集群。

要连接本地集群，只需运行如下代码：

library(sparklyr)  
sc <- spark\_connect(master = "local", version = "2.3")

如果你使用的是自己的或者在线的Spark集群，确保你是按照集群管理员的配置或者在线文档进行的连接。如果你需要指导，可以参考第7章，它会详细介绍如何连接Spark集群。

master参数明确了Spark集群中哪一个是主机器。这个机器通常叫做*驱动节点（driver node）*。使用多台机器的真实集群时，你会发现大部分机器是工作机器，一个是主机器。因为我们只有一个单台机器的本地集群，从现在开始我们会默认使用“local”。

建立好连接后，spark\_connect()会获取一个工作的Spark连接，通常在大部分代码中命名为sc，你可以使用sc执行Spark命令。

如果连接失败了，第7章包含有故障排查的内容，可以帮助你解决连接问题。

## 使用Spark

建立连接后，我们可以运行几个简单的命令。例如，首先使用copy\_to()把mtcars数据集复制给Apache Spark:

cars <- copy\_to(sc, mtcars)

现在数据复制给了Spark，我们可以使用cars从R中请求数据。要打印具体内容，我们可以键入\*cars\*:

cars  
# Source: spark<mtcars> [?? x 11]  
mpg cyl disp hp drat wt qsec vs am gear carb  
<dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
1 21 6 160 110 3.9 2.62 16.5 0 1 4 4  
2 21 6 160 110 3.9 2.88 17.0 0 1 4 4  
3 22.8 4 108 93 3.85 2.32 18.6 1 1 4 1  
4 21.4 6 258 110 3.08 3.22 19.4 1 0 3 1  
5 18.7 8 360 175 3.15 3.44 17.0 0 0 3 2  
6 18.1 6 225 105 2.76 3.46 20.2 1 0 3 1  
7 14.3 8 360 245 3.21 3.57 15.8 0 0 3 4  
8 24.4 4 147. 62 3.69 3.19 20 1 0 4 2  
9 22.8 4 141. 95 3.92 3.15 22.9 1 0 4 2  
10 19.2 6 168. 123 3.92 3.44 18.3 1 0 4 4  
# … with more rows

很棒！你成功的连接了Spark并加载了第一个数据集。

让我们看看copy\_to()做了什么操作。第一个参数sc，给函数提供了一个工作的Spark连接指向，这个连接是之前通过spark\_connect()创建的。第二个参数给出了要加载到Spark的数据集。现在，copy\_to()返回Spark中数据集，并由R环境自动打印。只要Spark打印数据集，它就会收集几条数据记录并在前台展示。在这个例子中，数据集只包含几行描述机动车型号的信息以及诸如马力，和每加仑可行驶英里数的说明。

### 网络接口

大部分Spark命令都在R控制台执行。但是，监控和分析动作是在Spark的网络界面完成，如图2-1所示。这个接口是Spark提供的网络应用，你可以运行如下代码进行请求：

spark\_web(sc)

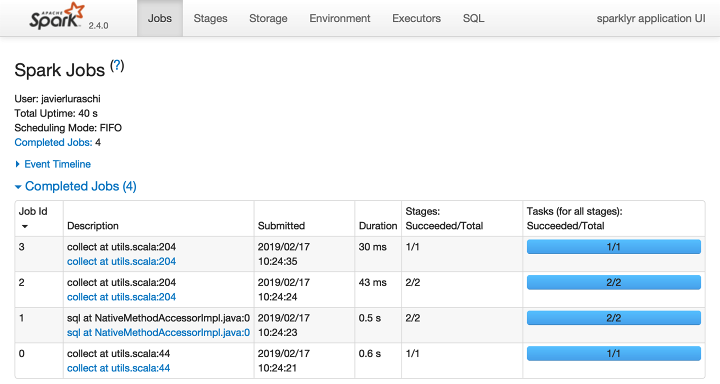


图2-1. Apache Spark网络接口

打印cars数据集会收集几行记录并展示到R控制台。你可以在Spark网络接口中看到，有一个作业用于在Spark中收集这个信息。你可以选择Storage标签，查看Spark内存中的mtcars数据集，如图2-2所示。

注意，根据Fraction Cached列100%取值，这个数据集完全加载到了内存中。因此，根据Size in Memory列，你可以准确的看到这个数据集使用了多少内存。

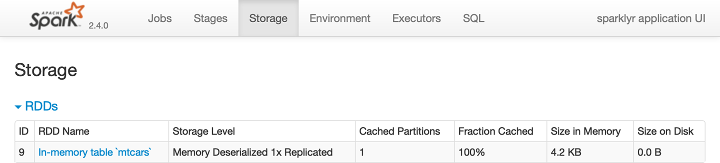


图2-2. Apache Spark网络接口的Storage标签

Executors标签，如图2-3所示，提供了集群资源的总览。对于本地连接，你只能看到一个2GB内存，384MB计算内存的Spark工作节点。在第9章中，你会看到如何请求更多的计算实例和资源，以及内存如何分配。

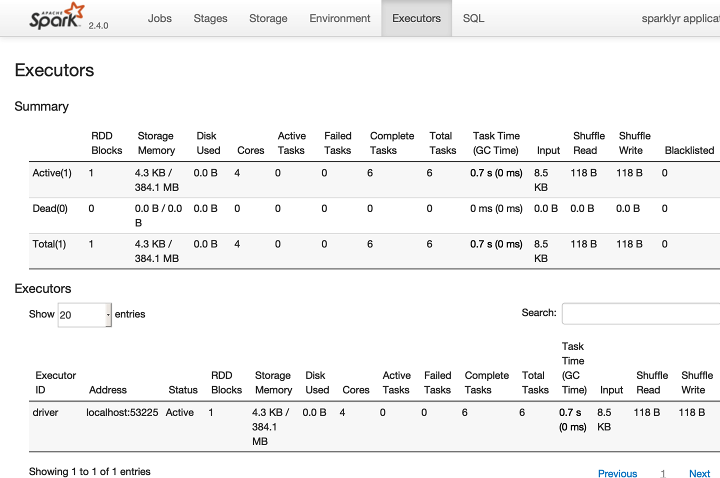


图2-3. Apache Spark网络接口的Executors标签

最后一个要介绍的标签是Environment标签，如图2-4所示；这个标签列出了所有Spark应用的设置，我们会在第9章中介绍。你可以看到，大部分设置不需要明确的配置，但是要在大规模数据上顺利运行，你还是要熟悉这些配置。

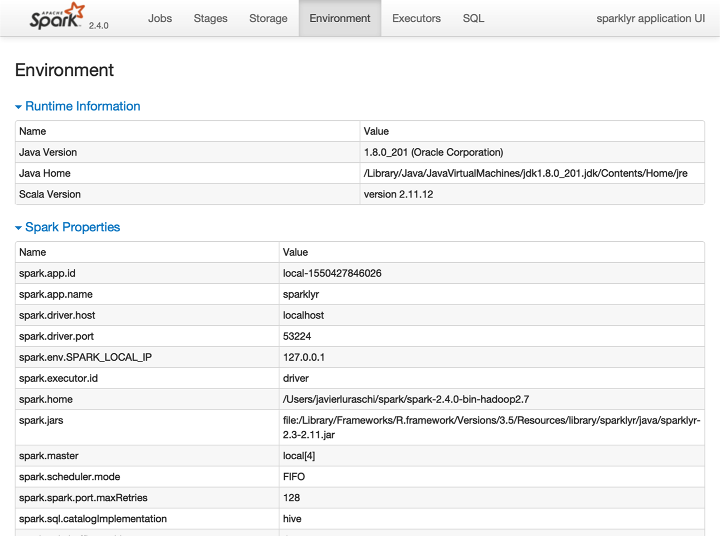


图2-4. Apache Spark网络接口的Environment标签

然后，你可以使用小的数据子集进行练习。我们会在第3章中详细介绍。

### 分析

在R中使用Spark分析数据时，可以使用SQL（结构化查询语句）或者dplyr（数据操作语法）。你可以通过DBI程序包使用SQL。例如，要计算我们的cars数据集中有多少记录，我们可以运行如下代码：

library(DBI)  
dbGetQuery(sc, "SELECT count(\*) FROM mtcars")  
count(1)  
1 32

使用dplyr的时候，需要的代码更少。而且通常比SQL容易的多。这也是为什么本书不用SQL的原因。但是，如果你对SQL很熟悉，SQL也是备选方案。例如，dplyr中计算行数要更简洁，且易于理解：

library(dplyr)  
count(cars)  
# Source: spark<?> [?? x 1]  
n  
<dbl>  
1 32

一般情况下，我们通常首先使用dplyr在Spark中做数据分析，然后进行采样和列维度的子集选取。最后一步是从Spark中收集数据，并在R中做进一步数据处理，例如数据可视化。让我们运行一个非常简单的数据分析例子，包括对Spark中cars数据集选取，采样，绘制图形:

select(cars, hp, mpg) %>% sample\_n(100) %>% collect() %>% plot()

图2-5看到，随着汽车马力的增加，它的每加仑英里数燃料效率在降低。尽管这个分析很深刻，但是要定量预测增加的马力影响多少燃料效率也并不容易。建模工作可以帮助我们解决这个问题。



图2-5. 马力vs每加仑英里数

### 建模

尽管数据分析可以让你快速理解数据，但是构建数学模型描述和泛化数据集也非常有用。在第1章中，你学习了机器学习和数据科学领域使用数学模型进行预测和发现重要的观点。例如，我们可以使用线性模型估计燃料效率和马力的关系：

model <- ml\_linear\_regression(cars, mpg ~ hp)  
model  
Formula: mpg ~ hp  
Coefficients:  
(Intercept) hp  
30.09886054 -0.06822828

现在我们可以使用这个模型预测不在原始数据集中的值。例如，我们可以添加马力大于250的数据，并可视化预测的值，如图2-6所示。

model %>% ml\_predict(copy\_to(sc, data.frame(hp = 250 + 10 \* 1:10))) %>% transmute(hp = hp,   
 mpg = prediction) %>% full\_join(select(cars, hp, mpg)) %>% collect() %>% plot()

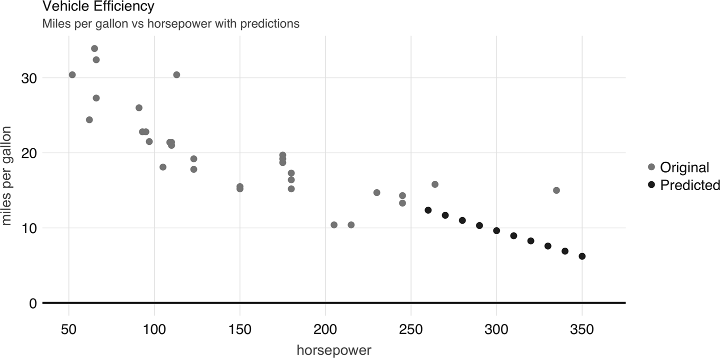


图2-6. 马力vs每加仑英里数预测结果

尽管前面的例子缺少建模中所用的诸多技术，但是也足以作为一个简单的例子简要介绍Spark的建模能力。我们会在第4章中介绍所有的Spark模型，技术和最佳实践。

### 数据

为了简单起见，我们把mtcars数据集复制给了Spark。但是，一般情况下数据通常不在Saprk中。相反，数据要从已有的数据源中读取，格式也多种多样，例如纯文本，CSV，JSON，Java数据库连接JDBC和其它。我们会在第8章中详细介绍这些内容。例如，我们可以把cars数据集输出为CSV文件：

spark\_write\_csv(cars, "cars.csv")

实际工作中，我们会从分布式存储系统例如HDFS中读取已有的数据集，但是我们也可以从本地文件系统中读取数据：

cars <- spark\_read\_csv(sc, "cars.csv")

### 扩展

R以其活跃的程序包作者社区而著名。同样的，许多Spark和R扩展已经被编程实现，并随时可用，只是数量较R社区中的少些。第10章会介绍许多有趣的程序包，它们支持高级建模，图分析，深度学习数据预处理等等。

例如，sparkly.nested扩展是R的一个程序包，允许sparklyr管理包含嵌套信息的取值。常见的用例是，包含嵌套列表的JSON文件，需要在有意义的数据分析前做预处理。要使用这个扩展，我们首先需要按照如下代码安装：

install.packages("sparklyr.nested")

然后，我们可以使用sparklyr.nested扩展，根据气缸数，对所有的马力数据点分组：

sparklyr.nested::sdf\_nest(cars, hp) %>%  
group\_by(cyl) %>%  
summarise(data = collect\_list(data))  
# Source: spark<?> [?? x 2]  
cyl data  
<int> <list>  
1 6 <list [7]>  
2 4 <list [11]>  
3 8 <list [14]>

尽管嵌套数据不太容易读取，但是在借助spark\_read\_json()和spark\_write\_json()函数处理诸如JSON的嵌套数据格式的时候，确实解决了这样的需求。

### 分布式R

对于某些情况下，个别Spark的功能不能实现，或者扩展没有开发完成，你可以考虑把R代码发布到Spark集群上。这是一个强大的工具，但是也带了额外的复杂性。所以你应该把它做为最后的方案。

假设我们需要四舍五入数据集中所有列的取值。一个方法是运行R代码，执行round()函数：

cars %>% spark\_apply(~round(.x))  
# Source: spark<?> [?? x 11]  
mpg cyl disp hp drat wt qsec vs am gear carb  
<dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
1 21 6 160 110 4 3 16 0 1 4 4  
2 21 6 160 110 4 3 17 0 1 4 4  
3 23 4 108 93 4 2 19 1 1 4 1  
4 21 6 258 110 3 3 19 1 0 3 1  
5 19 8 360 175 3 3 17 0 0 3 2  
6 18 6 225 105 3 3 20 1 0 3 1  
7 14 8 360 245 3 4 16 0 0 3 4  
8 24 4 147 62 4 3 20 1 0 4 2  
9 23 4 141 95 4 3 23 1 0 4 2  
10 19 6 168 123 4 3 18 1 0 4 4  
# … with more rows

如果你是R语言高手，你会倾向于使用spark\_apply()处理一切。但是最好别这样！spark\_apply()适用于Spark能力受限的高级情形。你会了解到如何执行合适的数据分析和建模，而不用把R代码分发给你的集群。

### Streaming

虽然处理大量静态数据是Spark的典型应用场景，但是Spark也可以处理实时动态数据集。某些应用中确实有这样的需求。你可以把一个流式数据集看做一个持续更新数据的静态数据源，例如股票市场报价。流式数据通常从Kafka（一个开源的流式数据处理软件平台）或分布式存储中读取。

要体验Streaming，首先创建一个*input/文件夹*，放入一些数据流的输入：

dir.create("input")  
write.csv(mtcars, "input/cars\_1.csv", row.names = F)

然后，定义一个作业流，处理来自*input/*文件夹的数据，执行R代码定义的转换，并输出到一个*output/*文件夹：

stream <- stream\_read\_csv(sc, "input/") %>% select(mpg, cyl, disp) %>% stream\_write\_csv("output/")

实时数据一开始输入，*input/*文件夹几乎同时开始处理，并在*output/*文件夹中存入新的转换数据文件。因为输入文件夹只有一个文件，输出文件夹也只包含一个经过spark\_apply()转换的文件。

dir("output", pattern = ".csv")  
[1] "part-00000-eece04d8-7cfa-4231-b61e-f1aef8edeb97-c000.csv"

到目前为止，这些操作和静态数据的处理很像。但是，我们可以持续添加*input/*文件夹中的文件，Spark会自动的并行处理数据。让我们添加一个新的文件，检查是否可以自动处理：

# Write more data into the stream source  
write.csv(mtcars, "input/cars\_2.csv", row.names = F)

等待几秒，确认Spark作业流处理了数据：

# Check the contents of the stream destination  
dir("output", pattern = ".csv")  
[1] "part-00000-2d8e5c07-a2eb-449d-a535-8a19c671477d-c000.csv"  
[2] "part-00000-eece04d8-7cfa-4231-b61e-f1aef8edeb97-c000.csv"

然后你应该终止作业流：

stream\_stop(stream)

你可以使用dplyr，SQL，Spark模型，或者分布式R分析实时数据流。在第12章中，我们会详细介绍分析实时数据所需的所有有意思的转换。

### 日志

日志记录肯定比实时数据处理要无聊。但是，它是你应该熟悉的工具。日志是一份文本文件，Spark可以追加集群中任务执行的相关信息。对于本地集群，我们运行如下代码，获取所有相关日志：

spark\_log(sc)  
18/10/09 19:41:46 INFO Executor: Finished task 0.0 in stage 5.0 (TID 5)...  
18/10/09 19:41:46 INFO TaskSetManager: Finished task 0.0 in stage 5.0...  
18/10/09 19:41:46 INFO TaskSchedulerImpl: Removed TaskSet 5.0, whose...  
18/10/09 19:41:46 INFO DAGScheduler: ResultStage 5 (collect at utils...  
18/10/09 19:41:46 INFO DAGScheduler: Job 3 finished: collect at utils...

我们也可以获取具体的日志记录，比如包含sparklyr的记录。使用filter参数，运行如下代码：

spark\_log(sc, filter = "sparklyr")  
## 18/10/09 18:53:23 INFO SparkContext: Submitted application: sparklyr 18/10/09  
## 18:53:23 INFO SparkContext: Added JAR... 18/10/09 18:53:27 INFO Executor:  
## Fetching spark://localhost:52930/... 18/10/09 18:53:27 INFO Utils: Fetching  
## spark://localhost:52930/... 18/10/09 18:53:27 INFO Executor: Adding  
## file:/private/var/folders/...

多数情况下，你不必操心Spark日志，除非你需要排查一个失败的计算任务。此时，日志是无价之宝。切记！

## 断开连接

对于本地集群（以及任何集群），你应该在完成数据处理后，使用如下代码断开连接：

spark\_disconnect(sc)

这些操作会终止集群连接和集群任务。如果有多个Spark连接在工作，或者连接实例sc不再可用，你也可以使用如下命令，断开所有的Spark连接：

spark\_disconnect\_all()

注意，退出R或者RStudio，或者重启R进程，也会引起Spark连接终止，进而终止Spark集群并把没有明确保存的数据放在缓存。

## 使用RStudio

既然使用RStudio运行R非常普遍，sparklyr也提供了RStudio扩展来帮助简化工作流，提高Rstudio中使用Spark的生产效率。如果你不熟悉RStudio，可以快速浏览依一下“使用Rstudio”部分。一些扩展非常值得关注。

首先，不需要Rstudio控制台的spark\_connect()建立连接，你可以用Connections标签下的New Connection操作，然后选取Spark connection，打开图2-7的对话窗口。你可以定制版本号和连接配置，生成正确的spark\_connect()命令，并在R控制台中执行。

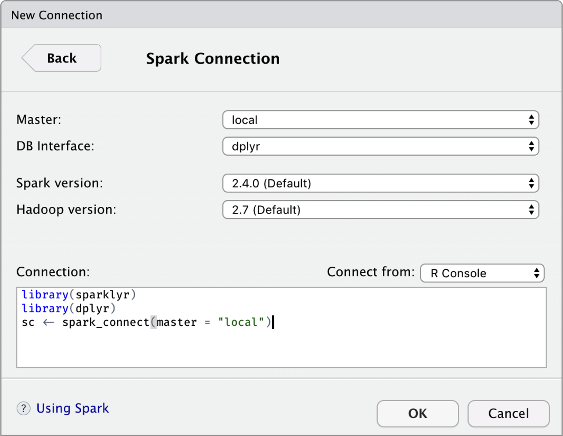


图2-7. RStudio中新建Spark连接对话窗口

连接到Spark后，RStudio在Connections标签中展示了可用的数据集，如图2-8所示。这是跟踪可用数据集很有用的方法，也提供了探查它们的简便方式。

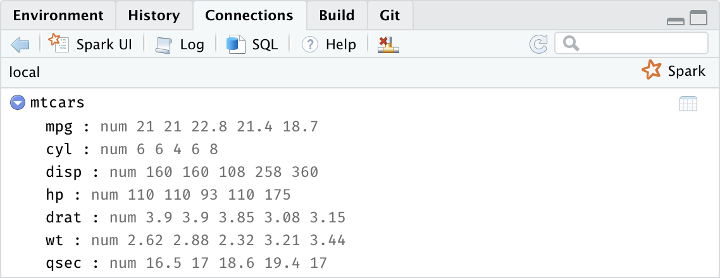


图2-8. RStudio的Connections标签

另外，工作的连接提供了如下定制的操作：

* Spark UI
* 打开Spark网络接口，spark\_web(sc)的快捷操作。
* 日志
* 打开Spark网络日志，spark\_log(sc)的快捷操作。
* SQL
* 打开新的SQL查询。更多关于DBI和SQL支持的信息，可以查看第3章。
* 帮助
* 在新的浏览器窗口中打开参考文档。
* 断开
* 断开Spark连接，spark\_disconnect(sc)的快捷操作。

本书后边的章节会使用纯R代码。本书中的代码也都在R环境中运行。是在R控制台，Rstudio，Jupyter Notebooks，还是其它支持R代码的工具中运行R代码，取决于你的个人需要。

## 资源

我们已经花了很大的精力简化上手过程，不过还有许多其它资源可以帮助你解决具体的上手问题。通常，这些资源会介绍更广泛的Spark和R社区，帮助你找到具体的答案，讨论某个话题，以及与经常使用Spark和R的用户沟通：

* 文档
* 托管在RStudio的Spark网站文档站点应该是了解更多在R中用Spark知识的首要选择。文档会对示例，引用函数和其它更多相关资源保存持续更新。
* 博客
* 要获取最新的重大sparklyr通知，你可以关注Rstudio博客。
* 社区
* 对于常见的sparklyr问题，你可以在RStudio社区发布帖子，并带上标签sparklyr。
* Stack Overflow
* 对于常见的Spark问题，Stack Overflow是一个很棒的资源，它也包含很多关于sparklyr的话题。
* GitHub
* 如果确定一些代码需要修改，你可以新建一个GitHub issue，或者给我们发送一个pull请求。
* Gitter
* 对于紧急的问题，或者需要时常保持联系，你可以通过Gitter和我们沟通。

## 小结

在本章中，你学到了使用Spark的预备操作。你已经看到如何使用spark\_connect()连接Spark，使用spark\_install()安装本地集群，使用spark\_web(sc)和spark\_log(sc)加载简单的数据集，启动网络接口和展示日志，使用RStudio的spark\_disconnect()断开连接。最后，我们介绍了sparklyr提供的Rstudio扩展。

现在，我们希望你已经准备好使用Spark和R分析真实数据和建模问题。这些内容会在接下来两章介绍。第3章会介绍数据分析，一个探查，清洗和转换数据的过程，达到发现有用信息的目的。第4章，建模，可以看做是数据分析的一部分。但是适合独立成章，以便真正阐述和利用Spark的建模功能。

# 第3章 分析

第一课：用尖的那头去刺敌人。 —Jon Snow

之前的章节主要介绍Spark和R，可以让你快速熟悉和鼓励尝试基础的数据分析流程。但是，我们并没有介绍什么是数据分析，尤其是Spark中的数据分析。 我们介绍了本书中需要用到的工具，有助于你更多的关注学习而不是排查问题。

本章会介绍使用Spark和R进行数据分析的工具和概念。需要注意的是，这些工具和你只使用R的时候一样！这并非偶然。我们只是希望数据科学家能够屏蔽一些技术细节，而只使用自己熟知和喜欢的程序包，并且确实在Spark框架下可行。现在我们离这个目标还有点距离，但是并不远。所以，你会在本章学到一些广泛使用的R程序包和实际操作来进行数据分析——dplyr，ggplot2，formulas，rmarkdown等，这些程序包在Spark框架下依然有效。

第4章会关注创建统计模型来预测，估计和描述数据集。但是首先，我们从分析工作开始！

## 概述

数据分析项目的主要目标是理解数据到底要告诉我们什么，希望能够给出一个具体问题的答案。大部分数据分析项目遵循一些步骤，如图3-1所示。

如图所示，我们首先把数据引入主要分析流程。这个流程会通过各种数据转换，例如数据合并，完成*数据操作*。然后，我们会对数据*可视化*，以便看到数据的关系和趋势。要获得更深的理解，我们可以用一个或者多个统计模型集合数据。这会帮助我们找出相关模式在数据集上是否成立。最后，结果可以以公开的形式或者私密的形式和同事和相关人员沟通。

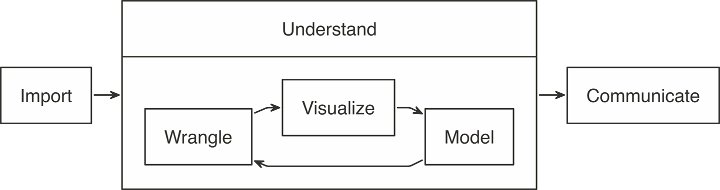


图3-1. 数据分析一般步骤

当处理小规模数据集，也就是内存允许的数据集时，我们可以按照R语言的步骤执行，不用使用Spark。但是，当数据及无法放入内存，或者计算太慢时，我们可以结合Spark优化这些步骤。但是如何优化呢？

对于数据分析师而言，理想的方法是让Spark做它擅长的事情。Spark是并行计算的引擎，可以在大数据集上工作，并提供SQL查询引擎和建模所用的各种库。你可以使用这些技术执行大部分与R相同的操作。这些操作，包括数据选取，转换，建模。

另外，Spark包含了一些执行特殊计算的工具，例如图分析，流式处理，还有其它工具。现在，我们跳过这些非表格数据集，留在后面的章节介绍。

你可以使用Spark进行数据*导入*，*操作*和*建模*。你也可以做一部分可视化工作，后边会介绍这一内容。最终目的是要使用R告诉Spark要做什么数据操作，并把结果返回给R。如图3-2所示，理想的流程可以*推动计算（pushes compute）* 到Spark集群中，并*收集结果（collects results）*返回给R。

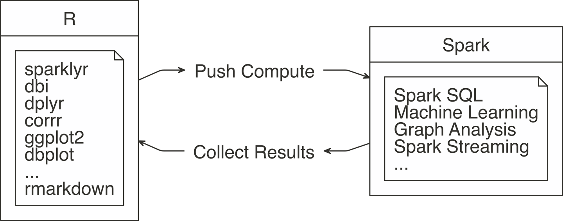


图3-2. Spark进行数据计算，R收集计算结果

sparklyr程序包助力实现“推动计算，收集结果”的原则。其大部分函数都是Spark API调用的顶层封装。这可以让我们利用Spark的分析组件，而不是R的分析组件。例如，当你需要拟合线性回归模型的时候，不要使用R的熟知的lm()函数，你应该使用Spark对应的ml\_linear\_regression()函数。 这个R函数会调用Spark创建模型。图3-3给出了具体的例子。

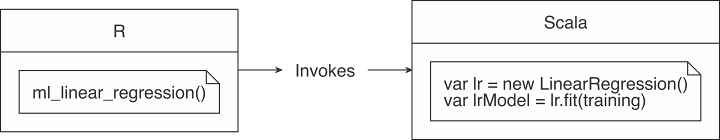


图3-3. R函数调用Spark函数

对于更常见的数据任务，sparklyr提供了dplyr的强大支持。这就意味着，你可以使用R中已经熟知的dplyr操作，然后sparklyr和dplyr会把这些操作转换成Spark SQL语句。这些语句通常比SQL语句更加紧凑，易读（见图3-4）。所以，如果你对R和dplyr很熟悉，那就无需过多学习。这样介绍可能让人觉得有点虎头蛇尾。的确有点这种感觉。但是，可以只专注学习大规模计算技能也是很棒的体验。

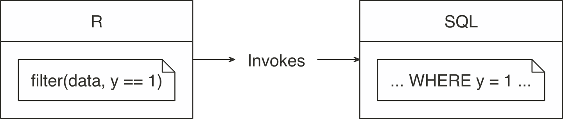


图3-4. dplyr在Spark中执行SQL语句

为了边学边练，本章后边的代码使用简单的练习，可以在本地Spark集群上运行。这样，你可以在自己的电脑上重跑代码。确保sparklyr可以使用，第2章已经有过相关介绍。

本章会使用一些你可能还没有安装的包，所以确保以下程序包通过如下命令安装成功：

install.packages("ggplot2")  
install.packages("corrr")  
install.packages("dbplot")  
install.packages("rmarkdown")

首先，加载sparklyr和dplyr程序包，并打开新的本地连接。

library(sparklyr)  
library(dplyr)  
sc <- spark\_connect(master = "local", version = "2.3")

环境已经可以用了，下面我们要导入数据，进行分析。

## 数据导入

使用Spark和R的时候，你需要使用不同的导入数据方法。通常，导入意味着R会读取文件，并加载到内存中。而使用Spark的时候，数据导入Spark，而不是R。在图3-5中，我们可以看到数据源是如何与Spark连接而不是和R连接的。

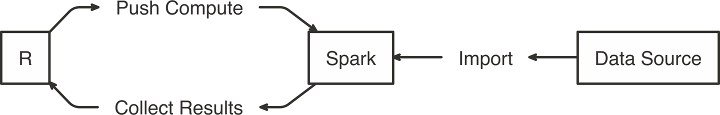


图3-5. 数据导入到Spark，而不是R

当你进行分析大规模数据集的时候，Spark集群中大部分所需的数据都可以进行操作（通常允许用户通过hive表或直接请求文件系统而实现）。第八章或详细介绍这些内容。

不要把所有的数据都导入Spark中，你可以让Spark请求数据源，而不是导入数据。你应该根据机器速度和性能做出一个决定。给Spark进程导入所有的数据会引起一次性内存支出，因为Spark需要等待所有的数据都加载好了才能分析。而如果没有导入数据，你通过会遇到每次Spark操作的支出问题，因为Spark需要从集群存储，通常是磁盘驱动器中获取数据子集。这通常要比从Spark内存中读取数据要慢得多。第九章会详细介绍这一话题。

让我们首先使用copy\_to()给Spark导入数据，准备有个带有数据的进程。你也可以从各种格式的分布式文件中导入数据。相关内容会在第八章中介绍。

cars <- copy\_to(sc, mtcars)

使用真实集群的时候，你应该只对小规模R数据表格使用copy\_to()进行转换，大数据表格转换应该使用专门的数据转换工具。

现在Spark可以请求数据了，你可以轻松的进行数据转换。下一节会介绍如何在Spark中使用dplyr转换数据。

## 数据操作

数据操作利用数据转换理解数据。它通常指的是把“原始”数据转换成其它格式的数据，并试图利于后续数据分析的过程。

异常值，缺失值，以及多重属性的列是常见的需要解决的数据问题，因为这些问题妨碍你理解数据集。例如，“name”字段包括客户的姓和名，这个列有两个属性（姓和名）。为了可用，我们需要把“name”字段转换成“first\_name”和“last\_name”。

完成数据清洗，你还需要理解数据内容的基本情况。其它转换诸如聚合，可以帮助我们完成这个任务。例如，所有客户的平均余额结果会返回一个数据表。值是所有客户的平均值。当我们观察单个客户或群体客户的余额时，这些信息会提供一些背景知识。

尽可能使用R语言来编写数据转换的代码，这才是主要目的。这样可以省去单个任务中多个计算机技术之间来回切换的认知成本。这种情况下，最好使用dplyr，而不要编写Spark的SQL语句，完成数据转换。

在R环境中，cars可以按照本地数据框处理，所以你可以使用dplyr语句。例如，我们可以使用summarise\_all()找出所有列的平均值：

summarise\_all(cars, mean)  
# Source: spark<?> [?? x 11]  
 mpg cyl disp hp drat wt qsec vs am gear carb  
 <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
1 20.1 6.19 231. 147. 3.60 3.22 17.8 0.438 0.406 3.69 2.81

虽然这些代码与非Spark环境下使用dplyr的代码相同，其内部还是做了很多运算。数据并*没有*导入到R中，相反，dplyr把这个任务变成SQL语句发送给Spark。show\_query()命令允许查看由sparklyr和dplyr创建发送的SQL语句细节。我们也可以使用管道操作符（%>%），一种来自magrittr程序包的定制操作符，把计算结果当做下一个函数的第一个参数，进而让数据分析更加已读：

summarise\_all(cars, mean) %>%  
show\_query()  
<SQL>  
SELECT AVG(`mpg`) AS `mpg`, AVG(`cyl`) AS `cyl`, AVG(`disp`) AS `disp`,  
 AVG(`hp`) AS `hp`, AVG(`drat`) AS `drat`, AVG(`wt`) AS `wt`,  
 AVG(`qsec`) AS `qsec`, AVG(`vs`) AS `vs`, AVG(`am`) AS `am`,  
 AVG(`gear`) AS `gear`, AVG(`carb`) AS `carb`  
FROM `mtcars`

可以看到，dplyr比SQL语句简洁。但是请放心，使用dplyr的时候不需要关心和理解SQL。你的关注点一直在数据理解上，而不是给出SQL下的一系列数据转换语句。下面是另一个例子，对cars数据集按照transmission类型分组：

cars %>%  
mutate(transmission = ifelse(am == 0, "automatic", "manual")) %>%  
group\_by(transmission) %>%  
summarise\_all(mean)  
# Source: spark<?> [?? x 12]  
 transmission mpg cyl disp hp drat wt qsec vs am gear carb  
 <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
1 automatic 17.1 6.95 290. 160. 3.29 3.77 18.2 0.368 0 3.21 2.74  
2 manmual 24.4 5.08 144. 127. 4.05 2.41 17.4 0.538 1 4.38 2.92

dplyr支持的大部分本地数据的转换操作也同样支持由Spark连接实现。这意味着，你可以首先关注学习dplyr，然后在Spark操作中复用。 由Hadley Wickham和Garrett Grolemund（O’Reilly）编写的《R语言数据科学》第5章是深入学习dplyr的良好资源。如果你已经熟练掌握dplyr，我们建议你花些时间，用不同的dplyr函数在cars数据表上做做实验。

有时，我们可能需要一些dplyr和sparklyr并不支持的操作。不要把数据加载到R中，而是找到一个Spark中的Hive函数完成所需的任务，下一节会介绍这个话题。

### 内置函数

Spark SQL基于Hive SQL的风格和函数而构建，并且可以使用dplyr调用所有的函数。这允许我们用任何一个Spark SQL函数来完成dplyr无法支持的数据操作。我们可以像R中的函数一样调用它们。dplyr可以把不曾认识的函数传递给查询引擎，为我们提供了使用函数的极大便利。

例如，percentile()函数返回组内一列的确切分位数。这个函数返回列名，以及一个分位数或一系列分位数。我们可以通过dplyr使用这个Spark SQL函数，如下所示:

summarise(cars, mpg\_percentile = percentile(mpg, 0.25))  
# Source: spark<?> [?? x 1]  
 mpg\_percentile  
 <dbl>  
1 15.4

R中并没有percentile()函数，所以dplyr把它原封不动的传给了SQL查询中：

summarise(cars, mpg\_percentile = percentile(mpg, 0.25)) %>%  
show\_query()  
<SQL>  
SELECT percentile(`mpg`, 0.25) AS `mpg\_percentile`  
FROM `mtcars\_remote`

要给percentile()传递多个值，我们可以调用另一个Hive函数array()。在这个例子中，array()类似于R中的list()函数。我们可以按照逗号分割，传递多个值。Spark的输出是一个数组变量，会以列表变量的形式导入到R中：

summarise(cars, mpg\_percentile = percentile(mpg, array(0.25, 0.5, 0.75)))  
# Source: spark<?> [?? x 1]  
 mpg\_percentile  
 <list>  
1 <list [3]>

你可以使用explode()函数分割Spark的数组，存到对应的记录中。可以在mutate()命令中使用explode()，并把包含分位数操作的变量传递出去：

summarise(cars, mpg\_percentile = percentile(mpg, array(0.25, 0.5, 0.75))) %>%  
mutate(mpg\_percentile = explode(mpg\_percentile))  
# Source: spark<?> [?? x 1]  
 mpg\_percentile  
 <dbl>  
Wrangle | 39  
1 15.4  
2 19.2  
3 22.8

我们在“Hive函数”的章节中罗列了所有的Hive函数。简单浏览一下，你就可以感受到Hive函数的强大运算能力。

### 相关性

计算和可视化相关性是非常常见的数据探查技术。我们经常计算后找到哪些成对的变量存在着何种统计关系。Spark提供了在整个数据集上计算相关性的函数，并返回给R，存在数据框对象中：

ml\_corr(cars)  
# A tibble: 11 x 11  
 mpg cyl disp hp drat wt qsec  
 <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
 1 1 -0.852 -0.848 -0.776 0.681 -0.868 0.419  
 2 -0.852 1 0.902 0.832 -0.700 0.782 -0.591  
 3 -0.848 0.902 1 0.791 -0.710 0.888 -0.434  
 4 -0.776 0.832 0.791 1 -0.449 0.659 -0.708  
 5 0.681 -0.700 -0.710 -0.449 1 -0.712 0.0912  
 6 -0.868 0.782 0.888 0.659 -0.712 1 -0.175  
 7 0.419 -0.591 -0.434 -0.708 0.0912 -0.175 1  
 8 0.664 -0.811 -0.710 -0.723 0.440 -0.555 0.745  
 9 0.600 -0.523 -0.591 -0.243 0.713 -0.692 -0.230  
10 0.480 -0.493 -0.556 -0.126 0.700 -0.583 -0.213  
11 -0.551 0.527 0.395 0.750 -0.0908 0.428 -0.656  
# ... with 4 more variables: vs <dbl>, am <dbl>,  
# gear <dbl>, carb <dbl>

corrr程序包专门进行相关性计算。它包含友好的准备和可视化结果的函数。程序包还包括连接Spark的后端，所以corrr中有Spark对象时，实际的计算发生在Spark中。后台计算中的correlate()函数执行sparklyr::ml\_corr()，所以运行命令前，不需要给R准备任何数据：

library(corrr)  
correlate(cars, use = "pairwise.complete.obs", method = "pearson")  
# A tibble: 11 x 12  
 rowname mpg cyl disp hp drat wt  
 <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
 1 mpg NA -0.852 -0.848 -0.776 0.681 -0.868  
 2 cyl -0.852 NA 0.902 0.832 -0.700 0.782  
 3 disp -0.848 0.902 NA 0.791 -0.710 0.888  
 4 hp -0.776 0.832 0.791 NA -0.449 0.659  
 5 drat 0.681 -0.700 -0.710 -0.449 NA -0.712  
 6 wt -0.868 0.782 0.888 0.659 -0.712 NA  
40 | Chapter 3: Analysis  
 7 qsec 0.419 -0.591 -0.434 -0.708 0.0912 -0.175  
 8 vs 0.664 -0.811 -0.710 -0.723 0.440 -0.555  
 9 am 0.600 -0.523 -0.591 -0.243 0.713 -0.692  
10 gear 0.480 -0.493 -0.556 -0.126 0.700 -0.583  
11 carb -0.551 0.527 0.395 0.750 -0.0908 0.428  
# ... with 5 more variables: qsec <dbl>, vs <dbl>,  
# am <dbl>, gear <dbl>, carb <dbl>

我们可以把结果输送给其它corrr函数。例如，shave()函数可以把所有重复值变成NA。而且，尽管这感觉很像标准的R代码，而事实上Spark确实在后端运行相关性计算。

另外，如图3-6所示，结果很容易使用rplot()可视化，如下所示:

correlate(cars, use = "pairwise.complete.obs", method = "pearson") %>% shave() %>%   
 rplot()

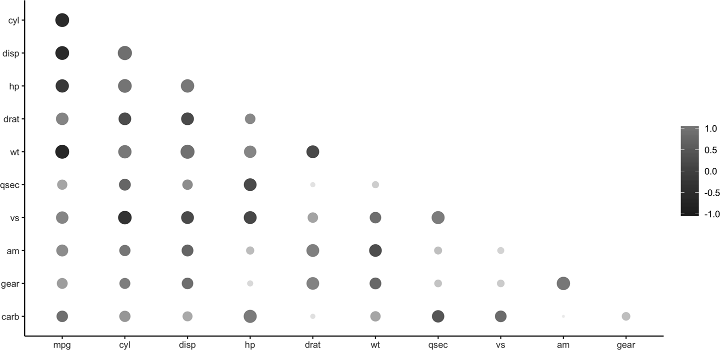


图3-6. 使用rplot()可视化相关性

很容易看到哪些关系是正向的，哪些是负向的。正向关系是灰色的，负向关系是黑色的。圆圈的大小表示关系的显著程度。可视化数据的作用在于它可以让我们更加容易的理解结果。下一节会介绍这方面的内容。

## 可视化

可视化是帮助我们发现数据模式的重要工具。与从列表中读取异常值相比，画出一个图形在1000个观测值的数据集中识别异常值会更容易。

R擅长数据可视化。R语言的绘图能力可以通过许多关注分析的程序包来扩展。不幸的是，大部分绘图的R函数都需要数据加载在本地内存中，所以使用它并不能在Spark中使用远程数据表。

然而我们还是可以使用R给Spark中的数据源做可视化。要明白其中的原理，让我们首先理解计算机是如何绘制图形的。首先，程序读入原始数据并完成一些数据转换。转换后的数据映射成一些坐标点。最后，映射出的值绘制在图中。图3-7描述了每一步。

图3-7. 一个R图形的各个阶段

图3-7. 一个R图形的各个阶段

可视化方法本质上与数据操作一样：让Saprk做计算，然后收集结果绘制图形。如图3-8，准备数据的繁重工作，例如按照分组或者分箱聚合数据可以在Spark中完成，而规模小很多的数据集可以在R中操作。R中的绘图变成了非常基本的操作。例如，一个直方图的分箱可以在Spark中计算，而R语言不用绘制直方图，而是实现简单的柱形图绘制。R语言没有必要再计算一次分箱。



图3-8. 使用Spark和R绘图

让我们借助ggplot2来应用这个概念模型。

### 使用ggplot2

要使用ggplot2绘制柱形图，只需调用一个函数：

library(ggplot2)  
ggplot(aes(as.factor(cyl), mpg), data = mtcars) + geom\_col()

在这个例子中，mtcars原始数据自动转换成3个离散的聚合数值。然后，每个结果都映射到x和y轴。接着，画出图形。作为R用户，绘图的所有阶段都非常便捷的抽象出来。

在Spark中，有两个关键的步骤来执行“推动计算，收集数据”的理念。第一个是确保在Spark中完成转换操作。在后面的例子中，group\_by()和summarise()会在Spark中运行。第二个是数据完成转换后返回给R。确保先转换数据后返回结果的顺序。如果collect()先运行，R会加载Spark中的全部数据集。根据数据集规模的不同，收集全部数据会拖慢你的系统，甚至宕机。

car\_group <- cars %>%  
group\_by(cyl) %>%  
summarise(mpg = sum(mpg, na.rm = TRUE)) %>%  
collect() %>%  
print()  
# A tibble: 3 x 2  
 cyl mpg  
 <dbl> <dbl>  
1 6 138.  
2 4 293.  
3 8 211.

在这个例子中，既然数据已经预先聚合好，并且返回给了R，只用给绘图函数传递三条记录即可：

ggplot(aes(as.factor(cyl), mpg), data = car\_group) + geom\_col(fill = "#999999") +   
 coord\_flip()

图3-9展示了绘图结果。

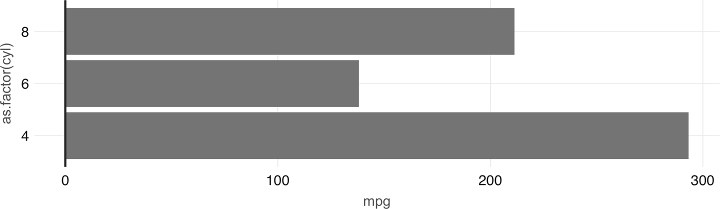


图3-9. Spark中绘制聚合结果

绘制其它ggplot2可视化都可以使用这个方法。但是，这部分内容超出了本书的范畴。我们推荐你可以查阅 Winston Chang 的《R Graphics Cookbook》，学习其它的Spark支持的可视化技术。现在，要简化可视化前的数据转换，dbplot程序包提供了一些立即可用的可视化方法，来自动化Spark中的聚合计算。

### 使用dbplot

dbplot程序包为远程数据提供了有用的绘图函数。dbplot中用于转换数据的代码已经完成，以便可以对应到Spark中。它会使用这些结果，通过ggplot2程序包创建图形，其中数据转换和绘图都是由一个函数完成的。dbplot\_histogram()函数可以让Spark计算分箱，并对每个分箱计数，产出一个ggplot对象，这个对象可以通过后续添加其它步骤进行优化。dbplot\_histogram()也接受binwidth参数来控制用于计算分箱的数据范围：

library(dbplot)  
cars %>% dbplot\_histogram(mpg, binwidth = 3) + labs(title = "MPG Distribution", subtitle = "Histogram over miles per gallon")

图3-10给出了绘图结果。

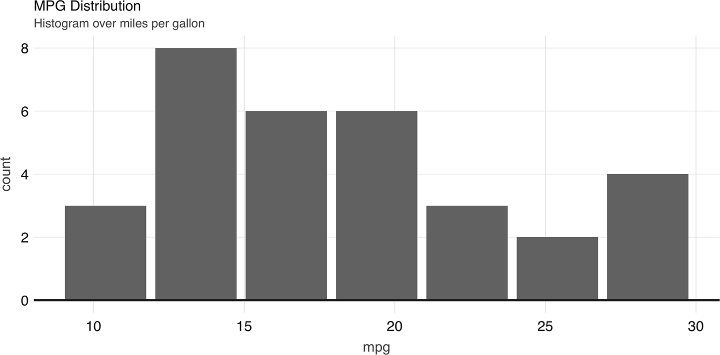


图3-10. dbplot创建的直方图

直方图是分析单变量的好方法。要分析两个变量，我们经常会用到散点图或栅格图。散点图用来对比两个连续型变量的关系。例如，一个散点图会展示汽车重量和汽油消耗之间的关系。图3-11说明，汽车越重，汽油消耗越多，因为这些点几乎靠拢成一条线，从左上角下降到右下角。下面是生成图形的代码：

ggplot(aes(mpg, wt), data = mtcars) + geom\_point()



图3-11. Spark中的散点图示例

然而，对于散点图，再多的“推动计算”也无助于解决这个问题，因为数据必须用单独的点绘制。

最好的选择是找到一种能代表x/y关系和集中度的绘图类型，且这种类型易于理解和实际绘图。栅格图可能是最好的答案。栅格图返回x/y位置的网格和给定聚集的结果，通常用带颜色的方形表示。

可以使用dbplot\_raster()在Spark中创建类似散点图的绘图，而只获取（收集）远程数据集的一个子集：

dbplot\_raster(cars, mpg, wt, resolution = 16)

如图3-12所示，生成的图返回不大于的网格。这会把需要收集到R中的记录数限制为25。

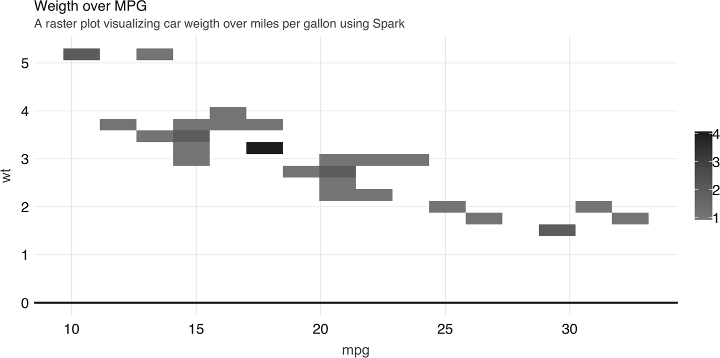


图3-12. 使用Spark绘制的

**你也可以使用dbplot获取原始数据并用其它方式做可视化。要获取聚合结果，而不是图形，使用db\_compute\_bins()，db\_compute\_count()，db\_compute\_raster()和db\_compute\_boxplot()。**

虽然可视化是必不可少的，但你还可以使用统计模型补足数据分析，以获得对数据更深入的了解。下一节将介绍如何准备数据以便使用Spark进行建模。

## 建模

下面两章完全关注建模。本章不会过多地介绍建模的细节，而是介绍如何在做数据分析的时候与模型交互。

首先，一个分析项目经过很多数据转换和模型才能找出答案。这也是为什么图3-2中的第一个数据分析图展示了一个周期，包括：可视化，数据操作和建模。我们知道，不管是在R环境中还是使用Spark的过程中，你都不会止步于建模。

所以，理想的数据分析语言可以让你快速的在“数据操作-可视化-建模”的迭代中切换。幸运的是，这就是使用Spark和R的场景。

为了说明在Spark中迭代数据操作和建模多么简单，可以考虑下面的例子。首先我们会在所有的特征上做一个线性回归，并且预测每加仑的英里数：

cars %>%  
ml\_linear\_regression(mpg ~ .) %>%  
summary()  
Deviance Residuals:  
 Min 1Q Median 3Q Max  
-3.4506 -1.6044 -0.1196 1.2193 4.6271  
Coefficients:  
(Intercept) cyl disp hp drat wt  
12.30337416 -0.11144048 0.01333524 -0.02148212 0.78711097 -3.71530393  
 qsec vs am gear carb  
0.82104075 0.31776281 2.52022689 0.65541302 -0.19941925  
R-Squared: 0.869  
Root Mean Squared Error: 2.147

现在，很容易使用不同的特征做实验。我们可以简单的改变R中的公式，把mpg ~ .改成mpg ~ hp + cyl，以便只使用马力和气缸作为特征：

cars %>%  
ml\_linear\_regression(mpg ~ hp + cyl) %>%  
summary()  
Deviance Residuals:  
 Min 1Q Median 3Q Max  
-4.4948 -2.4901 -0.1828 1.9777 7.2934  
Coefficients:  
(Intercept) hp cyl  
 36.9083305 -0.0191217 -2.2646936  
R-Squared: 0.7407  
Root Mean Squared Error: 3.021

另外，在不同的模型上迭代也非常容易。下面的例子把线性模型换成了广义线性模型：

cars %>%  
ml\_generalized\_linear\_regression(mpg ~ hp + cyl) %>%  
summary()  
Deviance Residuals:  
 Min 1Q Median 3Q Max  
-4.4948 -2.4901 -0.1828 1.9777 7.2934  
Coefficients:  
(Intercept) hp cyl  
 36.9083305 -0.0191217 -2.2646936  
(Dispersion parameter for gaussian family taken to be 10.06809)  
 Null deviance: 1126.05 on 31 degress of freedom  
Residual deviance: 291.975 on 29 degrees of freedom  
AIC: 169.56

通常，在拟合模型之前，你需要使用多个dplyr转换来让数据适合作为模型输入。要保证模型可以尽可能的高效拟合，你应该在拟合之前，把数据集缓存起来。我们会在下面的章节看到。

### 缓存

本节的例子使用非常小的数据集。在实际场景中，模型会使用大规模数据集。如果数据需要预先转换，会消耗Spark大量的进程开销。拟合模型之前，最好把所有转换的结果保存到一个Spark内存的新表中。

compute()命令会接棒一个dplyr命令把结果保存到Spark内存中：

cached\_cars <- cars %>%  
mutate(cyl = paste0("cyl\_", cyl)) %>%  
compute("cached\_cars")  
cached\_cars %>%  
ml\_linear\_regression(mpg ~ .) %>%  
summary()  
Deviance Residuals:  
 Min 1Q Median 3Q Max  
-3.47339 -1.37936 -0.06554 1.05105 4.39057  
Coefficients:  
(Intercept) cyl\_cyl\_8.0 cyl\_cyl\_4.0 disp hp drat  
16.15953652 3.29774653 1.66030673 0.01391241 -0.04612835 0.02635025  
 wt qsec vs am gear carb  
 -3.80624757 0.64695710 1.74738689 2.61726546 0.76402917 0.50935118  
R-Squared: 0.8816  
Root Mean Squared Error: 2.041

由于数据可以提炼出更多的理解，更多的问题也会随之提出来。这也是为什么我们希望通过数据操作，可视化和建模，来多次迭代。每次迭代应该增量的提供数据给我们传递的含义。我们会有一个时间点，来达到一个满意的理解水准。做到这一点，我们就可以分享分析的结果了。这就是下一章节的主题。

## 沟通

清楚的沟通分析结果非常重要，甚至和分析工作本身一样重要。公众，同事或者利益相关者需要知道你发现了什么信息，以及发现的方法。

为了有效的沟通，我们需要使用一些途径，例如报告或者展示：这些都是可以使用R环境中的*R markdown*创建的常见输出格式。R Markdown文档可以让你一起编写介绍性的文字和代码。不同的输出格式让R markdown的学习和使用变得很迫切。有很多可用的输出格式，例如HTML，PDF，PowerPoint，Word，网页幻灯片，网站，书籍等等。

大部分输出可以通过R markdown的核心程序包生成，例如knitr和rmarkdown。你可以使用其它程序包扩展R Markdown。例如，本书就是用R Markdown编写的，基于bookdown程序包提供的扩展而完成。深度了解R Markdown的最优资源是官方书籍[[13]](#footnote-103)。

在R Markdown中，一个文档可以被渲染成不同的格式。例如，你可以渲染同一份报告，通过在报告中设置不同的参数得到HTML或PDF文件。相反，许多不同类型的文档也可以渲染成同样的输出格式。例如，一个展示平台和一个报告都可以渲染成HTML格式。

使用Spark作为计算引擎，创建一个新的R Markdown报告并不难。在文件头部，R Markdown需要一个YAML标头。第一行和最后一行是三个连续短横线（---）。短横线之间的内容可以根据文档类型不同而变化。YAML标头中唯一必需的字段是output值。R Markdown需要知道它要把报告渲染成哪种类型的输出。这个YAML标头叫做*前辅文（frontmatter）*。前辅文后面是各种的代码，叫做*代码块（code chunks）*。这些代码块可以和介绍性文件交叉在一起。在Spark中使用R Markdown也并没有什么特别不同的地方：只是和平时的工作一样。

由于R Markdown文档是自含的并且是可以复现的，在润色文档之前，我们首先从Spark断开连接：

spark\_disconnect(sc)

下面的例子展示了创建一个完全可复现的有关Spark处理大规模数据集的报告是多么的容易。介绍性文字，代码，以及最重要的代码输出都记录在最后的HTML文件中。你可以把下面的代码复制粘贴到一个文件中，并用*.Rmd*后缀保存起来，取一个喜欢的文件名字：

---  
title: "mtcars analysis"  
output:  
 html\_document:  
 fig\_width: 6  
 fig\_height: 3  
---

library(ggplot2)  
cars %>%  
 group\_by(cyl) %>% summarise(mpg = mean(mpg)) %>%  
 ggplot(aes(cyl, mpg)) + geom\_bar(stat="identity")  
  
  
## Model  
The selected model was a simple linear regression that  
uses the weight as the predictor of MPG

cars %>% ml\_linear\_regression(wt ~ mpg) %>% summary()

要生成这个报告，用*.Rmd*扩展名保存，例如*report.Rmd*，并且运行R中的render()。输出结果应该和图3-13类似。

rmarkdown::render("report.Rmd")



图3-13. R Markdown HTML输出

你可以轻松的分享这个报告，并不会消耗读者的Spark或R资源。它是一份自含的HTML文件，任何网络浏览器都可以打开。

从一个报告中提炼出各种观点并生成不同的输出格式也很常见。格式的转换也很容易，只需改变输出选项为point\_presentation，pdf\_document，word\_document或类似选项。你甚至可以从一份报告中生成多个输出格式：

---  
title: "mtcars analysis"  
output:  
 word\_document: default  
 pdf\_document: default  
 powerpoint\_presentation: default  
---

最终结果是一个PowerPoint展示文档，一个Word文档和一个PDF。在原始HTML报告中展示的所有相同的信息都是通过Spark计算得出，并在R中渲染。你可能需要编辑PowerPoint模板，或者代码块的输出。

这个最小例子展示了从一种格式换成另一种格式是非常简单的。当然，R用户需要做更多的编辑工作，来保证幻灯片只包含相关信息。关键要记住，它不需要你为了从一个文档换成另一个文档，学习不同的标记语言或者代码风格。

## 小结

本章对R和Spark的数据分析进行了扎实的介绍。提出的许多技术看起来与只使用R而不使用Spark非常相似。这并非虎头蛇尾，而是让已经熟悉R的用户简单的过渡到Spark的正确设计。对于不熟悉R的用户，本章还非常简要介绍R中一些最流行的（有用的）程序包。

现在应该很清楚，R和Spark是一个强大的组合，一个大型计算平台，以及一个异常强大的R程序包生态系统，可以成为理想的分析平台。

在使用R和Spark分析时，请记住将计算推送到Spark，并在R中收集结果。这种范式应该建立了一个成功的途径，以各种输出格式分享结果，实现数据操作、可视化和沟通。

第4章将深入探讨如何使用大量更有趣的数据集（还有什么比约会数据集更有趣？）。你还将学习更多的技术，这些技术甚至没有在本章建模部分提及。

# 第4章 建模

多年来，我一直相信你的幻象，你的预言。 —Stannis Baratheon

在第3章中，你学到了如果在大规模数据集上利用Spark进行数据分析。在本章中，我们会详细介绍使用Spark构建预测模型的步骤。我们会研究MLlib，Spark中支持编写高层级代码在分布式数据上进行预测建模的模块，并在特征工程和探索性数据分析的背景下使用数据操作。

我们首先结合Spark的背景以及贯穿全章的数据集，介绍建模。然后我们会展示一个有监督的学习流程，包括探索性数据分析，特征工程，和模型构建。接着，我们会使用非结构化的文本数据继续介绍无监督的主题建模示例。切记，我们的目标是介绍大规模数据集上执行数据科学任务的各种技术，而不是指导严谨一致的分析。Spark中海油许多其它的模型并不会在本章中介绍。，但是在本章的末尾，你会看到一些不错的工具，可以试试用自己的模型做实验。

手动预测数据集通常是一个合理的手段（“手动”是指人们给Spark导入数据集，使用拟合好的模型来丰富或预测取值）。这引出一个问题，我们可以把这个过程自动化为任何人都可以使用的系统吗？例如，我们如何构建一个系统可以自动的识别垃圾邮件，而不用手动分析每一个邮箱账户？第5章会介绍自动化数据分析和建模流程的工具。但是在这之前，我们首先需要知道如何“亲手”训练模型。

## 概述

R中的Spark接口提供了R用户熟悉的建模算法。例如，我们已经用过ml\_linear\_regression(cars, mpg ~ .)，但是我们可以只用简单的运行ml\_logistic\_regression(cars, am ~ .)。

花些时间看看本书附录中长长的MLlib函数列表。简单浏览可以发现Spark支持决策树，梯度加速树，加速失效时间生产回归，保序回归，K-Means聚类，高斯混合聚类Clustering等等。

可以看到，Spark提供了大量的算法和特征转换函数。这里我们只涉及具有代表性的一些功能。完整 的预测模型概念讲解已经超出了本书的范畴，所以我们推荐阅读Hadley Wickham和Garrett Grolemund G (O’Reilly)的*《R语言数据科学》*，以及*Feature Engineering and Selection: A Practical Approach for Predictive Models*[[14]](#footnote-108)。本章中的一些例子和可视化也是从这些材料中引用过来（有时候会原封不动的引用）。

本章关注预测建模，Spark力求推动机器学习而非统计推断。通常机器学习更多关注预测未来而不是推断数据生产的过程[[15]](#footnote-109)，进而用来创建自动化系统。机器学习可以分成*监督式学习*（预测建模）和*非监督式学习*。在监督式学习中，通过数据集（X，Y）实例，我们试着学习一个函数可以把X映射到Y。在非监督式学习中，我们只有X，而没有Y标记。所以我们会试着学习X的结构。一些实际的监督式学习的用例包括预测明天的天气，确定信用卡交易是否虚假，制度汽车保险报价。使用非监督式学习，例子包括个人照片的自动聚类，根据购买历史划分顾客，文档聚类。

sparklyr中的机器学习接口设计的初衷是为了最小化认知的成本，可以轻松的从本地的，基于内存的原生R工作流转向集群计算，也包括反向的转变。然而Spark生态非常丰富，CRAN上依然有大量的程序包，帮助实现一个项目中所需要的功能。而且，你也会希望利用你在R中的技能和经验保持一定的生产力。我们在第3章中也用到了这一点。记录执行计算的位置，以及集群和R进程之间的合适转换，这很重要。

本章的例子使用*OkCupid*数据集[[16]](#footnote-110)。这个数据集包括来自在线约会网站的用户画像数据，以及一系列特征集合，例如性别职业等生平细节特征，还有有关个人兴趣的自由文本字段。这个数据集中有大约60,000份用户画像，可以轻松的导入目前笔记本的内存中， 算不上“大数据”。所以你可以很容易的在本地节点上运行Spark程序。

你可以下载这个数据集，代码如下：

download.file("https://github.com/r-spark/okcupid/raw/master/profiles.csv.zip", "okcupid.zip")  
unzip("okcupid.zip", exdir = "data")  
unlink("okcupid.zip")

我们不建议对这个数据集采样，因为最终模型的性能会不够好。但是，如果硬件资源受限，你还是需要进行采用，代码入下：

profiles <- read.csv("data/profiles.csv")  
write.csv(dplyr::sample\_n(profiles, 10^3), "data/profiles.csv", row.names = FALSE)

本章的例子使用小型数据集，以便你可以在本地节点上运行。在实际工作中，如果你的数据集可以很轻松的导入到本地内存中，可以考虑使用一个高效的，非分布式的模型算法实现。例如你可以使用ranger程序包，而不是ml\_random\_forest\_classifier()。

另外，你需要安装一些额外的程序包：

install.packages("ggmosaic")  
install.packages("forcats")  
install.packages("FactoMineR")

要运行示例，我们考虑以下问题：

对是否有人还在上班进行预测。也就是说，没有退休，不是学生，也不是失业的状态。

下面我们探索这个数据集。

## 探索性数据分析

探索性数据分析（Exploratory data analysis，EDA），在预测建模领域中是指查看部分数据，和进行数据汇总的行为。EDA阶段的具体目的由业务问题定义，但是也有一些共通的目标:

* 确认数据质量；确认缺失值的含义和缺失程度，根据当期条件确定统计量。
* 理解单变量的含义。
* 初步评估纳入的变量和需要的转换。

首先，连接到Spark，加载相关的库，读取数据：

library(sparklyr)  
library(ggplot2)  
library(dbplot)  
library(dplyr)  
sc <- spark\_connect(master = "local", version = "2.3")  
okc <- spark\_read\_csv(sc, "data/profiles.csv", escape = "\"", memory = FALSE, options = list(multiline = TRUE)) %>%   
 mutate(height = as.numeric(height), income = ifelse(income == "-1", NA, as.numeric(income))) %>%   
 mutate(sex = ifelse(is.na(sex), "missing", sex)) %>% mutate(drinks = ifelse(is.na(drinks),   
 "missing", drinks)) %>% mutate(drugs = ifelse(is.na(drugs), "missing", drugs)) %>%   
 mutate(job = ifelse(is.na(job), "missing", job))

我们给定escape = "\""和options = list(multiline = TRUE)，接受字段中自带的引号和换行符。我们把height和income列的属性转换成数值类型，并按字符型记录缺失值。需要注意，最好尝试几次给定不同的参数，保证数据的输入是正确的。有时，即使在获取数据更多数据理解的建模阶段，你也需要回头看看这些步骤。

现在我们可以使用glimpse()快速查看一下数据：

glimpse(okc)  
Observations: ??  
Variables: 31  
Database: spark\_connection  
$ age <int> 22, 35, 38, 23, 29, 29, 32, 31, 24, 37, 35…  
$ body\_type <chr> "a little extra", "average", "thin", "thin…  
$ diet <chr> "strictly anything", "mostly other", "anyt…  
$ drinks <chr> "socially", "often", "socially", "socially…  
$ drugs <chr> "never", "sometimes", "missing", "missing"…  
$ education <chr> "working on college/university", "working …  
$ essay0 <chr> "about me:<br />\n<br />\ni would love to …  
$ essay1 <chr> "currently working as an international age…  
$ essay2 <chr> "making people laugh.<br />\nranting about…  
$ essay3 <chr> "the way i look. i am a six foot half asia…  
$ essay4 <chr> "books:<br />\nabsurdistan, the republic, …  
$ essay5 <chr> "food.<br />\nwater.<br />\ncell phone.<br…  
$ essay6 <chr> "duality and humorous things", "missing", …  
$ essay7 <chr> "trying to find someone to hang out with. …  
$ essay8 <chr> "i am new to california and looking for so…  
$ essay9 <chr> "you want to be swept off your feet!<br />…  
$ ethnicity <chr> "asian, white", "white", "missing", "white…  
$ height <dbl> 75, 70, 68, 71, 66, 67, 65, 65, 67, 65, 70…  
$ income <dbl> NaN, 80000, NaN, 20000, NaN, NaN, NaN, NaN…  
$ job <chr> "transportation", "hospitality / travel", …  
$ last\_online <chr> "2012-06-28-20-30", "2012-06-29-21-41", "2…  
$ location <chr> "south san francisco, california", "oaklan…  
$ offspring <chr> "doesn&rsquo;t have kids, but might want t…  
$ orientation <chr> "straight", "straight", "straight", "strai…  
$ pets <chr> "likes dogs and likes cats", "likes dogs a…  
$ religion <chr> "agnosticism and very serious about it", "…  
$ sex <chr> "m", "m", "m", "m", "m", "m", "f", "f", "f…  
$ sign <chr> "gemini", "cancer", "pisces but it doesn&r…  
$ smokes <chr> "sometimes", "no", "no", "no", "no", "no",…  
$ speaks <chr> "english", "english (fluently), spanish (p…  
$ status <chr> "single", "single", "available", "single",…

我们添加一个响应变量作为一列，查看它的分布：

okc <- okc %>%  
mutate(  
not\_working = ifelse(job %in% c("student", "unemployed", "retired"), 1 , 0) )  
okc %>%  
group\_by(not\_working) %>%  
tally()  
# Source: spark<?> [?? x 2]  
 not\_working n  
 <dbl> <dbl>  
1 0 54541  
2 1 5405

继续讲解之前，让我们先把数据分成训练集和测试集，并把测试集放在一边。事实上，这是很关键的一步，因为我们希望有一个例外的集合，可以在建模过程结束的时候用来评估模型性能。如果我们在EDA阶段囊括了所有的数据，测试集中的信息会“泄露”到可视化和汇总统计中,让我们的模型构建带有某种偏置。即便一个学习算法没有直接使用这个数据，也会发生这种情况。这会降低性能评估的可信度。我们可以简单的使用sdf\_random\_split()函数分割数据集：

data\_splits <- sdf\_random\_split(okc, training = 0.8, testing = 0.2, seed = 42)  
okc\_train <- data\_splits$training  
okc\_test <- data\_splits$testing

我们可以快速的查看响应变量的分布：

okc\_train %>%  
group\_by(not\_working) %>%  
tally() %>%  
mutate(frac = n / sum(n))  
# Source: spark<?> [?? x 3]  
 not\_working n frac  
 <dbl> <dbl> <dbl>  
1 0 43785 0.910  
2 1 4317 0.0897

使用sdf\_describe()函数，我们可以获取具体列的数值统计信息：

sdf\_describe(okc\_train, cols = c("age", "income"))  
# Source: spark<?> [?? x 3]  
 summary age income  
 <chr> <chr> <chr>  
1 count 48102 9193  
2 mean 32.336534863415245 104968.99815076689  
3 stddev 9.43908920033797 202235.2291773537  
4 min 18 20000.0  
5 max 110 1000000.0

正如我们在第3章看到的，我们也可以使用dbplot程序包绘制这些变量的分布。在图4-1中，我们展示了一个年龄变量的分布直方图，其通过以下代码生成：

dbplot\_histogram(okc\_train, age)

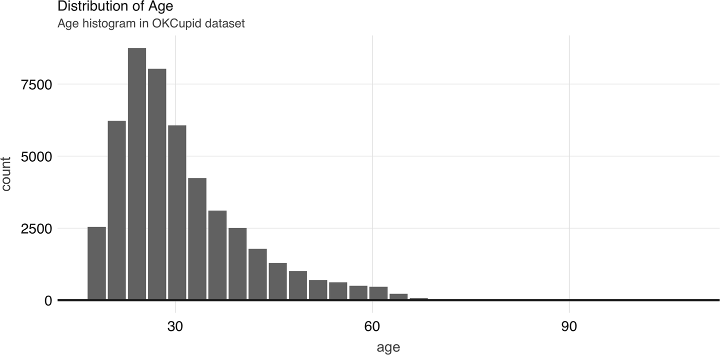


图4-1. 年龄分布

一般的EDA操作是要查看响应变量个各个预测变量之间的关系。通常，你可能有些先验知识理解这些关系的含义，所以这些知识可以用在数据质量的检查。而且，一些异常的趋势可以展示变量的交互，你也可能把这些交互信息建模到模型中。作为一个例子，我们可以探索religion变量：

prop\_data <- okc\_train %>%  
mutate(religion = regexp\_extract(religion, "^\\\\w+", 0)) %>%  
group\_by(religion, not\_working) %>%  
tally() %>%  
group\_by(religion) %>%  
summarise(  
count = sum(n),  
prop = sum(not\_working \* n) / sum(n) ) %>%  
mutate(se = sqrt(prop \* (1 - prop) / count)) %>%  
collect()  
prop\_data  
# A tibble: 10 x 4  
 religion count prop se  
 <chr> <dbl> <dbl> <dbl>  
 1 judaism 2520 0.0794 0.00539  
 2 atheism 5624 0.118 0.00436  
 3 christianity 4671 0.120 0.00480  
 4 hinduism 358 0.101 0.0159  
 5 islam 115 0.191 0.0367  
 6 agnosticism 7078 0.0958 0.00346  
 7 other 6240 0.0841 0.00346  
 8 missing 16152 0.0719 0.002  
 9 buddhism 1575 0.0851 0.007  
10 catholicism 3769 0.0886 0.00458

注意prop\_data是一个小型的数据框，它通过R进程导入到内存中，我们可以利用ggplot2创建一个丰富的可视化结果（见图4-2）：

prop\_data %>% ggplot(aes(x = religion, y = prop)) + geom\_point(size = 2) + geom\_errorbar(aes(ymin = prop -   
 1.96 \* se, ymax = prop + 1.96 \* se), width = 0.1) + geom\_hline(yintercept = sum(prop\_data$prop \*   
 prop\_data$count)/sum(prop\_data$count))

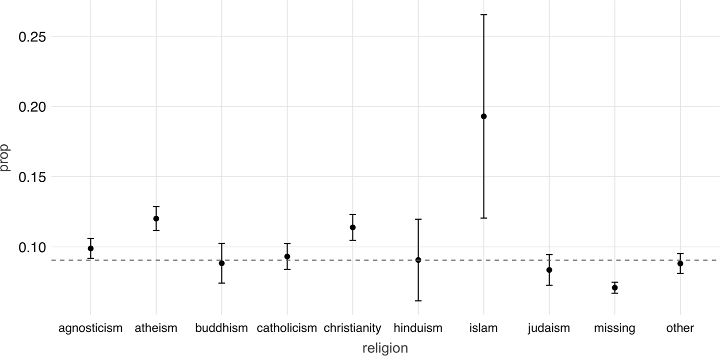


图4-2. 当期失业人群按地区占比

然后，我们看一下两个预测变量之间的关系：饮酒和毒品。我们希望它们之间有某种相关性。你可以通过sdf\_crosstab()计算出一个列联表：

contingency\_tbl <- okc\_train %>%  
sdf\_crosstab("drinks", "drugs") %>%  
collect()  
contingency\_tbl  
# A tibble: 7 x 5  
 drinks\_drugs missing never often sometimes  
 <chr> <dbl> <dbl> <dbl> <dbl>  
1 very often 54 144 44 137  
2 socially 8221 21066 126 4106  
3 not at all 146 2371 15 109  
4 desperately 72 89 23 74  
5 often 1049 1718 69 1271  
6 missing 1121 1226 10 59  
7 rarely 613 3689 35 445

我们可以使用一个马赛克图，可视化这个列联表（见图4-3）：

library(ggmosaic)  
library(forcats)  
library(tidyr)  
contingency\_tbl %>% rename(drinks = drinks\_drugs) %>% gather("drugs", "count", missing:sometimes) %>%   
 mutate(drinks = as\_factor(drinks) %>% fct\_relevel("missing", "not at all", "rarely",   
 "socially", "very often", "desperately"), drugs = as\_factor(drugs) %>% fct\_relevel("missing",   
 "never", "sometimes", "often")) %>% ggplot() + geom\_mosaic(aes(x = product(drinks,   
 drugs), fill = drinks, weight = count))



图4-3. 毒品和酒精使用的马赛克图

要进一步探索这两个变量之间的关系，我们可以使用FactoMineR程序包进行对应分析[[17]](#footnote-115)。这项技术可以通过把每个因子映射到平面上的一个点，让我们刻画高维因子水平之间的关系。首先，我们使用FactoMineR::CA()获取映射，代码如下：

dd\_obj <- contingency\_tbl %>% tibble::column\_to\_rownames(var = "drinks\_drugs") %>%   
 FactoMineR::CA(graph = FALSE)

然后，我们使用ggplot绘制结果，如图4-4所示：

dd\_drugs <- dd\_obj$row$coord %>% as.data.frame() %>% mutate(label = gsub("\_", " ",   
 rownames(dd\_obj$row$coord)), Variable = "Drugs")  
dd\_drinks <- dd\_obj$col$coord %>% as.data.frame() %>% mutate(label = gsub("\_", " ",   
 rownames(dd\_obj$col$coord)), Variable = "Alcohol")  
ca\_coord <- rbind(dd\_drugs, dd\_drinks)  
ggplot(ca\_coord, aes(x = `Dim 1`, y = `Dim 2`, col = Variable)) + geom\_vline(xintercept = 0) +   
 geom\_hline(yintercept = 0) + geom\_text(aes(label = label)) + coord\_equal()

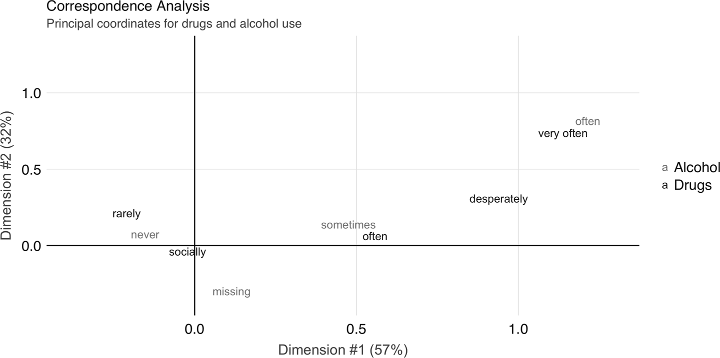


图4-4. 毒品和酒精使用的对应分析主坐标

在图4-4中，我们看到对应分析流程把因子转换成名为*主坐标*的变量，它对应因子的坐标轴并表示列联表包含了多少信息。例如，我们可以把“drinking often”和“using drugs very often”之间的距离看成是指示性关联。

这是有关EDA的全部介绍。下面让我们介绍特征工程。

## 特征工程

特征工程实践包括数据转换，以便增加模型性能。这些实践具体指，数值的中心化和缩放，字符的操作以便获取有意义的变量等。它也包括变量选取——选择哪些预测因素用到模型中。

在图4-1中，我们看到age变量的取值范围是18到60以上。一些算法，特别是神经网络在归一化输入，维持相同的数据规模情况下训练的更快。现在让我们通过移除平均值和缩放单位方差，归一化age变量。首先计算平均值和标准差：

scale\_values <- okc\_train %>%  
summarise(  
mean\_age = mean(age),  
sd\_age = sd(age) ) %>%  
collect()  
scale\_values  
# A tibble: 1 x 2  
 mean\_age sd\_age  
 <dbl> <dbl>  
1 32.3 9.44

然后，我们使用这些信息转换数据集：

okc\_train <- okc\_train %>% mutate(scaled\_age = (age - !!scale\_values$mean\_age)/!!scale\_values$sd\_age)  
dbplot\_histogram(okc\_train, scaled\_age)

在图4-5中，我们看到缩放后的age变量取值位于零附近。下面我们讨论其它类型的转换，但是在做特征工程的时候，你可能会希望对所有模型中用到的数值变量进行归一化。 Feature Engineering | 63



图4-5. 缩放后age变量的分布

由于一些特征会被多次选取，换句话说，一个人可能对一个变量关联多个选择，我们需要在构建有意义的模型前做些处理。例如，如果我们看一下ethnicity列，可以发现有许多组合：

okc\_train %>%  
group\_by(ethnicity) %>%  
tally()  
# Source: spark<?> [?? x 2]  
 ethnicity n  
 <chr> <dbl>  
 1 hispanic / latin, white 1051  
 2 black, pacific islander, hispanic / latin 2  
 3 asian, black, pacific islander 5  
 4 black, native american, white 91  
 5 middle eastern, white, other 34  
 6 asian, other 78  
 7 asian, black, white 12  
 8 asian, hispanic / latin, white, other 7  
 9 middle eastern, pacific islander 1  
10 indian, hispanic / latin 5  
# … with more rows

一种处理方式是把种族的每一种组合都看成是一个独立的水平，但是这会导致生成大量的水平，这会给许多算法带来问题。为了更好的编码这些信息，我们可以为每个种族创建哑变量，代码如下：

ethnicities <- c("asian", "middle eastern", "black", "native american", "indian",  
"pacific islander", "hispanic / latin", "white", "other")  
ethnicity\_vars <- ethnicities %>%  
purrr::map(~ expr(ifelse(like(ethnicity, !!.x), 1, 0))) %>%  
purrr::set\_names(paste0("ethnicity\_", gsub("\\s|/", "", ethnicities)))  
okc\_train <- mutate(okc\_train, !!!ethnicity\_vars)  
okc\_train %>%  
select(starts\_with("ethnicity\_")) %>%  
glimpse()  
Observations: ??  
Variables: 9  
Database: spark\_connection  
$ ethnicity\_asian <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…  
$ ethnicity\_middleeastern <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…  
$ ethnicity\_black <dbl> 0, 1, 0, 0, 0, 0, 0, 0, 0, 0…  
$ ethnicity\_nativeamerican <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…  
$ ethnicity\_indian <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…  
$ ethnicity\_pacificislander <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…  
$ ethnicity\_hispaniclatin <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…  
$ ethnicity\_white <dbl> 1, 0, 1, 0, 1, 1, 1, 0, 1, 0…  
$ ethnicity\_other <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…

对于自由文本字段，一个直接的获取特征的方法是计算字符数量。我们会把训练集存在Spark的内存中，使用compute()计算。

okc\_train <- okc\_train %>% mutate(essay\_length = char\_length(paste(!!!syms(paste0("essay",   
 0:9))))) %>% compute()  
dbplot\_histogram(okc\_train, essay\_length, bins = 100)

我们可以看到essay\_length变量的分布，如图4-6所示。

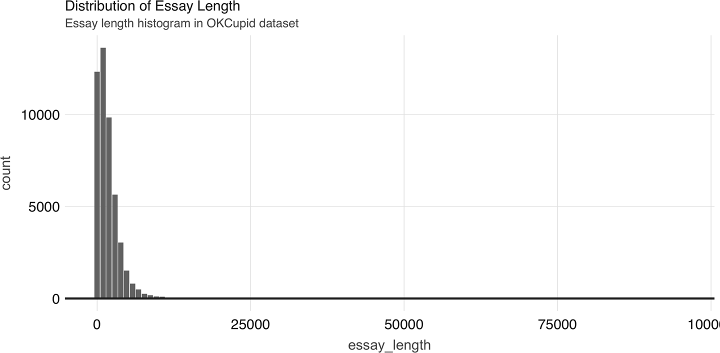


图4-6. 文章长度分布

我们会在第5章中使用这个数据集，所以让我们先存成一个Parquet文件，一种数值型数据的高效文件格式：

spark\_write\_parquet(okc\_train, "data/okc-train.parquet")

现在我们有了一些可用的特征，可以进行一些监督式的学习算法。

## 监督式学习

一旦我们有了数据集，我们就可以开始构建一些模型。在我们开始之前，需要制定一个计划来调整和验证备选模型。在建模项目中，我们经常尝试不同类型的模型和方法来进行拟合，看看哪一个表现最好。因为我们处理的是一个二分类问题，可用的指标包括准确度，精确度，灵敏度，以及接受者操作特性曲线（ROC AUC）下的面积，以及其它度量方法。要优化的指标取决于特定业务问题，但是在本次练习中，我们将重点放在ROC AUC上。

很重要的一点是，我们在最终结束前不要偷看测试集，因为我们获得的任何信息都可能影响我们的建模决策，这会反过来让模型性能估计不够可信。为了调试验证，我们进行10折交叉验证，这是模型的标准调试方法。具体方案如下：首先将数据集划分为10个大致相同的子集。我们把第2到第10组作为训练集，而在第一个集合上验证模型结果。然后，我们把第2组作为验证集，并在第1，第3到第10组上训练算法。最终，我们培训了10个模型，并对性能进行平均。如果时间和资源允许，你还可以使用不同的随机分区多次执行此过程。在我们的例子中，我们进行一次交叉验证。从现在开始，我们把与每个分割相关的训练集称为分析数据，并将验证集作为评估数据。

使用sdf\_random\_split()函数，我们可以构建一系列okc\_train数据表的子集：

vfolds <- sdf\_random\_split(okc\_train, weights = purrr::set\_names(rep(0.1, 10), paste0("fold",   
 1:10)), seed = 42)

然后，我们创建第一次分析/评估数据集分割，代码如下：

analysis\_set <- do.call(rbind, vfolds[2:10])  
assessment\_set <- vfolds[[1]]

这里我们需要仔细处理的一个问题是变量的缩放。我们需要确保不要把评估集的任何信息泄漏给分析集，所以我们只在分析集上计算平均值和标准差，并对两个集合做相同的转换。以下是我们对age变量所做的操作：

make\_scale\_age <- function(analysis\_data) {  
 scale\_values <- analysis\_data %>% summarise(mean\_age = mean(age), sd\_age = sd(age)) %>%   
 collect()  
 function(data) {  
 data %>% mutate(scaled\_age = (age - !!scale\_values$mean\_age)/!!scale\_values$sd\_age)  
 }  
}  
scale\_age <- make\_scale\_age(analysis\_set)  
train\_set <- scale\_age(analysis\_set)  
validation\_set <- scale\_age(assessment\_set)

为了简洁起见，这里我们只展示如何转换age变量。但是在实践中，你可能希望归一化每一个连续预测变量，例如前一节中导出的essay\_length 变量。

逻辑回归通常是二分类问题的合理起点，所以让我们试试看。假设我们的领域知识给我们提供了一组初始预测变量。然后，我们可以使用Formula接口拟合模型：

lr <- ml\_logistic\_regression(  
analysis\_set, not\_working ~ scaled\_age + sex + drinks + drugs + essay\_length  
)  
lr  
Formula: not\_working ~ scaled\_age + sex + drinks + drugs + essay\_length  
Coefficients:  
 (Intercept) scaled\_age sex\_m drinks\_socially  
 -2.823517e+00 -1.309498e+00 -1.918137e-01 2.235833e-01  
 drinks\_rarely drinks\_often drinks\_not at all drinks\_missing  
 6.732361e-01 7.572970e-02 8.214072e-01 -4.456326e-01  
drinks\_very often drugs\_never drugs\_missing drugs\_sometimes  
 8.032052e-02 -1.712702e-01 -3.995422e-01 -7.483491e-02  
 essay\_length  
 3.664964e-05

要获取评估集的性能度量信息，我们可以使用ml\_evaluate()函数：

validation\_summary <- ml\_evaluate(lr, assessment\_set)

你可以打印validation\_summary，查看可用的度量：

validation\_summary  
BinaryLogisticRegressionSummaryImpl  
 Access the following via `$` or `ml\_summary()`.  
 - features\_col()  
 - label\_col()  
 - predictions()  
 - probability\_col()  
 - area\_under\_roc()  
 - f\_measure\_by\_threshold()  
 - pr()  
 - precision\_by\_threshold()  
 - recall\_by\_threshold()  
 - roc()  
 - prediction\_col()  
 - accuracy()  
 - f\_measure\_by\_label()  
 - false\_positive\_rate\_by\_label()  
 - labels()  
 - precision\_by\_label()  
 - recall\_by\_label()  
 - true\_positive\_rate\_by\_label()  
 - weighted\_f\_measure()  
 - weighted\_false\_positive\_rate()  
 - weighted\_precision()  
 - weighted\_recall()  
 - weighted\_true\_positive\_rate()

我们可以使用validation\_summary$roc()的输出和ggplot2，绘制ROC曲线：

roc <- validation\_summary$roc() %>% collect()  
ggplot(roc, aes(x = FPR, y = TPR)) + geom\_line() + geom\_abline(lty = "dashed")

图4-7展示了绘制结果。

ROC曲线绘制了真阳性率（敏感性）与假阳性率（1-特异性）之间的关系，以便反映分类阈值的变化。在实践中，业务问题有助于确定曲线上设置分类阈值的位置。AUC是一种用于确定模型质量的汇总度量，我们可以通过调用area\_under\_roc()函数来计算它。



图4-7. 逻辑回归模型的ROC曲线

validation\_summary$area\_under\_roc()  
[1] 0.7872754

Spark只为广义线性模型（包括线性模型和逻辑回归）提供了评估方法。对于其它算法，你可以使用评估函数（例如，数据框的ml\_binary\_classification\_evaluator() ，或者是自己的度量算法。

现在，我们可以轻松的重复这个已经用在分割分析/评估数据集上的逻辑：

cv\_results <- purrr::map\_df(1:10, function(v) {  
analysis\_set <- do.call(rbind, vfolds[setdiff(1:10, v)]) %>% compute()  
assessment\_set <- vfolds[[v]]  
scale\_age <- make\_scale\_age(analysis\_set)  
train\_set <- scale\_age(analysis\_set)  
validation\_set <- scale\_age(assessment\_set)  
model <- ml\_logistic\_regression(  
analysis\_set, not\_working ~ scaled\_age + sex + drinks + drugs + essay\_length  
)s <- ml\_evaluate(model, assessment\_set)  
roc\_df <- s$roc() %>%  
collect()  
auc <- s$area\_under\_roc()  
tibble(  
Resample = paste0("Fold", stringr::str\_pad(v, width = 2, pad = "0")),  
Supervised Learning | 69  
roc\_df = list(roc\_df),  
auc = auc  
)  
})

这会返回10条ROC曲线：

unnest(cv\_results, roc\_df) %>% ggplot(aes(x = FPR, y = TPR, color = Resample)) +   
 geom\_line() + geom\_abline(lty = "dashed")

图4-8展示了绘图结果：

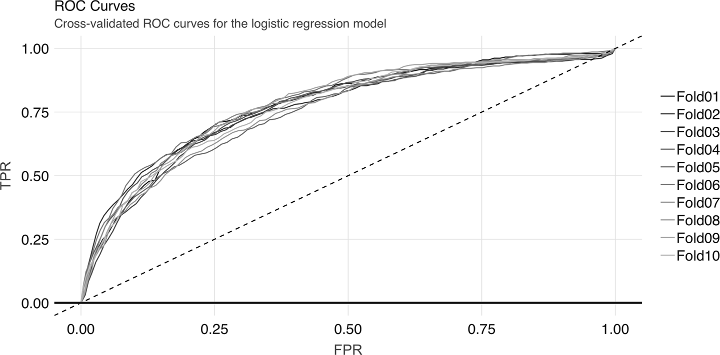


图4-8. 逻辑回归模型的交叉验证ROC曲线

我们可以获取AUC度量的平均值：

mean(cv\_results$auc)  
[1] 0.7715102

### 广义线性回归

如果你对广义线性模型（generalized linear model，GLM）诊断感兴趣，你也可以通过广义线性回归接口，给定family= "binomial"，拟合一个逻辑回归模型。因为结果是一个回归模型，ml\_predict()方法不会给出类别概率。但是它包含了置信区间，可以进行系数估计：

glr <- ml\_generalized\_linear\_regression(analysis\_set, not\_working ~ scaled\_age +   
 sex + drinks + drugs, family = "binomial")  
tidy\_glr <- tidy(glr)

我们可以把置信估计存到一个整齐的数据框中，以便进一步处理。例如，创建一个系数绘图，如图4-9所示：

tidy\_glr %>% ggplot(aes(x = term, y = estimate)) + geom\_point() + geom\_errorbar(aes(ymin = estimate -   
 1.96 \* std.error, ymax = estimate + 1.96 \* std.error, width = 0.1)) + coord\_flip() +   
 geom\_hline(yintercept = 0, linetype = "dashed")

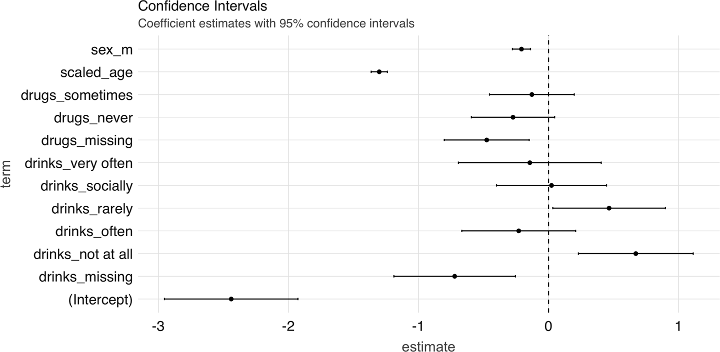


图4-9. 95%置信区间的系数估计

ml\_logistic\_regression()和ml\_linear\_regression()都支持通过reg\_param和elastic\_net\_param参数，实现弹性网正则化[[18]](#footnote-125)。reg\_param对应，而elastic\_net\_param对应。ml\_generalized\_linear\_regression()只支持reg\_param。

### 其它模型

Spark支持许多标准的建模算法，很容易为特定问题应用这些模型和超参数（控制模型拟合过程的值）。你可以在附录中找到支持ML相关函数的列表。访问这些功能的接口大部分是相同的，因此很容易进行实验。例如，为了拟合神经网络模型，我们可以运行以下命令：

nn <- ml\_multilayer\_perceptron\_classifier(analysis\_set, not\_working ~ scaled\_age +   
 sex + drinks + drugs + essay\_length, layers = c(12, 64, 64, 2))

这返回一个前馈神经网络模型，它有两个隐藏层，每层64个节点。注意，必须在layers参数中为输入和输出层指定正确的值。我们可以使用ml\_predict()在验证集上获取预测值：

predictions <- ml\_predict(nn, assessment\_set)

然后，我们可以通过ml\_binary\_classification\_evaluator()计算AUC：

ml\_binary\_classification\_evaluator(predictions)  
[1] 0.7812709

到目前为止，我们还没有查看文章字段中的非结构化文本，只是做了一些简单的字符计数。在下一节中，我们会具体探索文本数据。

## 非监督式学习

除了语音、图像和视频之外，文本数据也是大数据爆炸的表现之一。在现代文本挖掘技术和计算资源之前，公司很少使用自由格式的文本字段。今天，文本被认为是一个丰富的无处不在的信息来源，从医生的笔记到客户的投诉。在本节中，我们将展示一些sparklyr基本的文本分析能力。如果想了解文本挖掘技术的更多背景知识，我们建议阅读David Robinson和Julie Silge的《R文本挖掘》（O’Reilly）。

在本节中，我们将展示如何对OKCupid数据集中的文章数据进行基本的主题建模任务。我们的计划是连接每个用户的文章字段（其中有10个），并将每个用户看成为一个文档，然后尝试使用隐狄利克雷分布（Latent Dirichlet Allocation ，LDA）来发现主题（我们很快会定义这些主题）。

### 数据准备

和往常一样，在分析一个数据集（或其中的一个子集）之前，我们希望快速地查看它以便确定自己的方向。在这种情况下，我们关注用户在约会档案中输入的自由格式文本。

essay\_cols <- paste0("essay", 0:9)  
essays <- okc %>%  
select(!!essay\_cols)  
essays %>%  
glimpse()  
Observations: ??  
Variables: 10  
Database: spark\_connection  
$ essay0 <chr> "about me:<br />\n<br />\ni would love to think that…  
$ essay1 <chr> "currently working as an international agent for a f…  
$ essay2 <chr> "making people laugh.<br />\nranting about a good sa…  
$ essay3 <chr> "the way i look. i am a six foot half asian, half ca…  
$ essay4 <chr> "books:<br />\nabsurdistan, the republic, of mice an…  
$ essay5 <chr> "food.<br />\nwater.<br />\ncell phone.<br />\nshelt…  
$ essay6 <chr> "duality and humorous things", "missing", "missing",…  
$ essay7 <chr> "trying to find someone to hang out with. i am down …  
$ essay8 <chr> "i am new to california and looking for someone to w…  
$ essay9 <chr> "you want to be swept off your feet!<br />\nyou are …

从这些输出，我们看到：

* 文本包含HTML标签。
* 文本包含换行符（\n）。
* 数据中有缺失值。

HTML标记和特殊字符会污染数据，因为它们不是用户直接输入的，也不提供有意义的信息。类似地，由于我们已经用*missing*字符串对丢失的字符字段进行了编码，所以需要删除它。（注意，这样还将删除用户编写的单词“missing”的实例，但删除过程中丢失的信息可能很小。）

当你分析自己的文本数据时，你会很快发现并熟悉特定数据集的独有性质。与表格数值数据一样，预处理文本数据是一个迭代的过程。经过几次尝试之后，我们得到以下转换结果：

essays <- essays %>% # Replace `missing` with empty string.  
mutate\_all(list(~ifelse(. == "missing", "", .))) %>% # Concatenate the columns.  
mutate(essay = paste(!!!syms(essay\_cols))) %>% # Remove miscellaneous characters and HTML tags  
mutate(words = regexp\_replace(essay, "\\n|&nbsp;|<[^>]\*>|[^A-Za-z|']", " "))

这里注意我们使用regex\_replace()，它是一个Spark的SQL函数。下面，我们讨论LDA以及如何应用到干净的数据集上。

### 主题建模

LDA是一种主题模型，用于识别一组文档中的抽象“主题”。它是一种无监督的算法，因为我们不给输入文档提供任何标签或主题。LDA假设每个文档都是主题的混合体，并且主题是词的混合体。在训练阶段，它试图同时估计这两个变量。主题模型的典型用例包括对许多文档进行分类，这些文档由于量大使得手动方法不可行。应用领域从GitHub问题到法律文档。

在我们遵循上一节的工作流，得到一个非常干净的数据集之后，可以用ml\_lda()来拟合LDA模型：

stop\_words <- ml\_default\_stop\_words(sc) %>% c("like", "love", "good", "music", "friends",   
 "people", "life", "time", "things", "food", "really", "also", "movies")  
lda\_model <- ml\_lda(essays, ~words, k = 6, max\_iter = 1, min\_token\_length = 4, stop\_words = stop\_words,   
 min\_df = 5)

我们还需要一个stop\_words向量，由常用的英语单词和我们数据集中常见的单词组成，它会告诉算法忽略它们。在模型拟合之后，我们可以使用tidy()函数从模型中提取相关的beta，即每个主题中每个单词的概率。

betas <- tidy(lda\_model)  
betas  
# A tibble: 256,992 x 3  
 topic term beta  
 <int> <chr> <dbl>  
 1 0 know 303.  
 2 0 work 250.  
 3 0 want 367.  
 4 0 books 211.  
 5 0 family 213.  
 6 0 think 291.  
 7 0 going 160.  
 8 0 anything 292.  
 9 0 enjoy 145.  
10 0 much 272.  
# … with 256,982 more rows

然后，我们可以按主题查看单词概率来可视化输出。在图4-10和图4-11中，我们显示了1次迭代和100次迭代的结果。下面是生成图4-10的代码；要生成图4-11，你需要在运行ml\_lda()时设置max\_iter = 100，但要注意在单机上这可能需要长时间——这是一个大计算问题，而一个合适Spark集群可以很容易地解决。

betas %>% group\_by(topic) %>% top\_n(10, beta) %>% ungroup() %>% arrange(topic, -beta) %>%   
 mutate(term = reorder(term, beta)) %>% ggplot(aes(term, beta, fill = factor(topic))) +   
 geom\_col(show.legend = FALSE) + facet\_wrap(~topic, scales = "free") + coord\_flip()

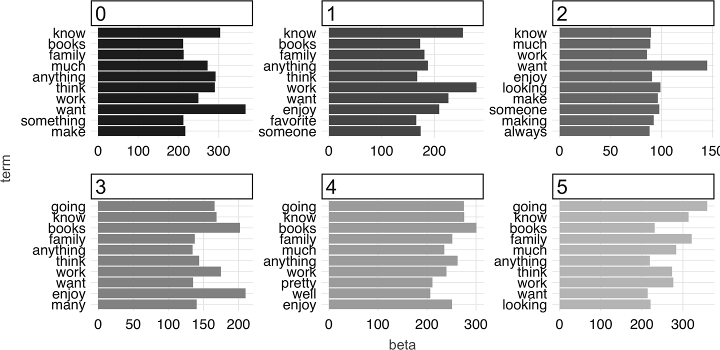


图4-10. 第1次迭代后，每个主题最常用的词语

在100次迭代中，我们可以看到“主题”开始出现。如果你正在挖掘大量不熟悉的文档，这本身就是一个有趣的信息。学到的主题也可以作为下游监督学习任务的特征；例如，我们可以考虑在模型中使用主题编号作为预测变量，在我们的预测建模例子

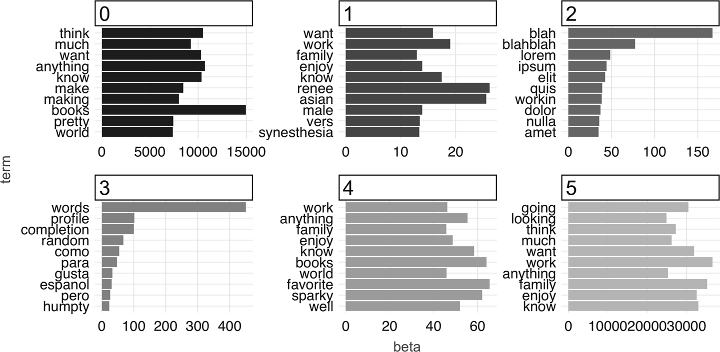


图4-11. 第100次迭代后，每个主题最常用的词语

最后，要结束本章，你应该断开Spark连接。第5章也会使用OKCupid数据集，但是我们会从零开始，介绍如何重新加载它：

spark\_disconnect(sc)

## 小结

在本章中，我们介绍了用R在Spark中构建预测模型的基础知识，包括EDA、特征工程和构建监督式模型，我们探索了逻辑回归和神经网络，模型。这只是Spark中通过MLlib提供的几十种模型中的一小部分。

然后我们探索了如何使用无监督式学习来处理原始文本。我们创建了一个主题模型，该模型自动将文档分为六个类别。我们介绍了使用一台机器构建主题模型会需要花费大量的时间。这几乎是引入全功能计算集群的完美理由。但要注意：我们首先需要考虑如何自动化数据科学工作流程。

正如我们在介绍本章时提到的，本章重点是预测建模。Spark可以助力大规模数据科学项目，也可以产品化数据科学工作流程，使之可以自动化，也就是我们熟知的机器学习。第5章介绍工具，让预测模型，以及整个训练流程，融合到自动化环境中，可以持续的运行，或者导出到web应用程序、移动应用程序和其它环境中。

# 第5章 管道操作

你再也不会走路了，但你会飞的！ —Three-Eyed Raven

在第4章中，你学习了如何使用Spark提供的高级函数以及与Spark协同工作的著名R程序包，来构建预测模型。你首先学习了监督式算法，最后学习了原始文本上的非监督式算法。

在本章中，我们将深入学习Spark管道操作，这是第四章中介绍的特征工程的动力。例如，当通过R中的公式接口调用MLlib函数时，例如ml\_logistic\_regression(cars, am ~ .)，一个*管道*就在背后构建完成了。因此，管道允许使用高级数据处理和建模流程。此外，管道还允许将其*部署*到生产系统，web应用程序、移动应用程序等,促进了跨数据科学和工程团队的协作。

本章也是鼓励使用本地计算机作为Spark集群的最后一章。你离真正的集群计算，以及数据科学或机器学习只有一章之遥，它们可以扩展到最苛刻的计算问题上。

## 概述

管道的构件是称为*转换器*和*评估器*的对象，它们统称为*管道阶段*。转换器可将转换操作应用于数据框并返回另一个数据框；生成的数据框通常包含原始数据框及其追加的新列。另一方面，估计器可以用来创建一个提供训练数据的转换器。考虑下面的例子来说明这种关系：一个“中心和扩展“估计器可以学习一些数据的平均值和标准差，并将统计信息存储在最后的转换器对象中；然后，可以使用这个转换器来归一化它所训练的数据以及任何新的、但看不见的数据。

这里是定义一个评估器的过程：

library(sparklyr)  
library(dplyr)  
sc <- spark\_connect(master = "local", version = "2.3")  
scaler <- ft\_standard\_scaler(  
sc,  
input\_col = "features",  
output\_col = "features\_scaled",  
with\_mean = TRUE)  
scaler  
StandardScaler (Estimator)  
<standard\_scaler\_7f6d46f452a1>  
 (Parameters -- Column Names)  
 input\_col: features  
 output\_col: features\_scaled  
 (Parameters)  
 with\_mean: TRUE  
 with\_std: TRUE

现在我们可以创建一些数据（我们已经知道平均值和标准差），然后使用ml\_fit()函数拟合扩展模型：

df <- copy\_to(sc, data.frame(value = rnorm(100000))) %>%  
ft\_vector\_assembler(input\_cols = "value", output\_col = "features")  
scaler\_model <- ml\_fit(scaler, df)  
scaler\_model  
StandardScalerModel (Transformer)  
<standard\_scaler\_7f6d46f452a1>  
 (Parameters -- Column Names)  
 input\_col: features  
 output\_col: features\_scaled  
 (Transformer Info)  
 mean: num 0.00421  
 std: num 0.999

在Spark ML中，许多算法和特征变换都要求输入为矢量列。函数ft\_vector\_assembler()执行此任务。你还可以使用该函数初始化要在管道中使用的转换器。

我们看到，平均值和标准差分别非常接近0和1，这正是我们期望的。借助ml\_transform()函数，我们可以使用转换器来转换数据框：

scaler\_model %>%  
ml\_transform(df) %>%  
glimpse()  
Observations: ??  
Variables: 3  
Database: spark\_connection  
$ value <dbl> 0.75373300, -0.84207731, 0.59365113, -…  
$ features <list> [0.753733, -0.8420773, 0.5936511, -0.…  
$ features\_scaled <list> [0.7502211, -0.8470762, 0.58999, -0.4…

既然你已经有了基本的评估器和转换器例子。我们继续介绍管道。

## 创建工作

一个*管道*是一系列转换器和评估器，一个*管道模型*是在数据上完成训练的一个管道。因此，所有部件都变成了转换器。

在sparklyr中，有两种方法构建管道，二者都使用ml\_pipeline()函数。

我们可以使用ml\_pipeline(sc)初始化空管道，并追加一些阶段：

ml\_pipeline(sc) %>%  
ft\_standard\_scaler(  
input\_col = "features",  
output\_col = "features\_scaled",  
with\_mean = TRUE)  
Pipeline (Estimator) with 1 stage  
<pipeline\_7f6d6a6a38ee>  
 Stages  
 |--1 StandardScaler (Estimator)  
 | <standard\_scaler\_7f6d63bfc7d6>  
 | (Parameters -- Column Names)  
 | input\_col: features  
 | output\_col: features\_scaled  
 | (Parameters)  
 | with\_mean: TRUE  
 | with\_std: TRUE

或者，我们也可以直接把阶段传给ml\_pipeline()：

pipeline <- ml\_pipeline(scaler)

我们按照拟合评估器的方式，拟合一个管道：

pipeline\_model <- ml\_fit(pipeline, df)  
pipeline\_model  
PipelineModel (Transformer) with 1 stage  
<pipeline\_7f6d64df6e45>  
 Stages  
 |--1 StandardScalerModel (Transformer)  
 | <standard\_scaler\_7f6d46f452a1>  
 | (Parameters -- Column Names)  
 | input\_col: features  
 | output\_col: features\_scaled  
 | (Transformer Info)  
 | mean: num 0.00421  
 | std: num 0.999  
pipeline

由于Spark ML的设计，即使管道只包含转换器，它也始终看成是评估器对象。这意味着，如果有一个只有转换器的管道，你仍然需要对其调用ml\_fit()，以获得一个转换器。本例中的“拟合”过程实际上不会修改任何转换器。

## 用例

既然你已经了解了ML管道的基本概念，那么让我们把它们用到上一章中的预测建模问题。在上一章中，我们试图通过查看人们的画像来预测他们目前是否有工作。我们从带有相关列的okc\_train数据框开始。

okc\_train <- spark\_read\_parquet(sc, "data/okc-train.parquet")  
okc\_train <- okc\_train %>% select(not\_working, age, sex, drinks, drugs, essay1:essay9,   
 essay\_length)

首先，我们展示管道，包括特征工程和建模步骤，然后逐步走通：

pipeline <- ml\_pipeline(sc) %>% ft\_string\_indexer(input\_col = "sex", output\_col = "sex\_indexed") %>%   
 ft\_string\_indexer(input\_col = "drinks", output\_col = "drinks\_indexed") %>% ft\_string\_indexer(input\_col = "drugs",   
 output\_col = "drugs\_indexed") %>% ft\_one\_hot\_encoder\_estimator(input\_cols = c("sex\_indexed",   
 "drinks\_indexed", "drugs\_indexed"), output\_cols = c("sex\_encoded", "drinks\_encoded",   
 "drugs\_encoded")) %>% ft\_vector\_assembler(input\_cols = c("age", "sex\_encoded",   
 "drinks\_encoded", "drugs\_encoded", "essay\_length"), output\_col = "features") %>%   
 ft\_standard\_scaler(input\_col = "features", output\_col = "features\_scaled", with\_mean = TRUE) %>%   
 ml\_logistic\_regression(features\_col = "features\_scaled", label\_col = "not\_working")

前三个阶段通过ft\_string\_indexer()为sex、drinks和drugs字符型列建立数值索引。这对于接下来的ft\_one\_hot\_encoder\_estimator()是必要的，因为它需要数值列输入。当我们所有的预测变量都是数值类型时（回想一下，age已经是数值类型），我们可以使用ft\_vector\_assembler()创建特征向量，它将其所有输入拼接成一个列向量中。然后我们可以使ft\_standard\_scaler()归一化特征列的所有元素（包括类别型变量独热编码的0/1值），最后通过ml\_logistic\_regression()使用逻辑回归。

在原型设计期间，你可能将数据框传递给ft\_和ml\_函数并检查转换后的数据框，以此*急切地*在数据的一小部分上执行这些转换。即时的反馈方便快速迭代想法；当你获得所需的处理步骤时，你可以将它们打包成一个管道。例如，你可以执行以下操作：

okc\_train %>%  
ft\_string\_indexer("sex", "sex\_indexed") %>%  
select(sex\_indexed)  
# Source: spark<?> [?? x 1]  
 sex\_indexed  
 <dbl>  
 1 0  
 2 0  
 3 1  
 4 0  
 5 1  
 6 0  
 7 0  
 8 1  
 9 1  
10 0  
# … with more rows

找到数据集的合适转换后，可以用ml\_pipeline(sc)替换数据框输入，结果是一个可以应用到任何具有合适结构的数据框的管道。在下一节中，我们将看到管道如何使我们更容易测试不同的模型配置。

### 超参调试

回到前面创建的管道，我们可以使用ml\_cross\_validator()来执行在上一章中介绍的交叉验证工作流，并轻松测试不同的超参数组合。在这个例子中，我们测试逻辑回归中变量的中心化和各种正则化手段是否可以改善预测。我们定义交叉验证器，代码如下：

cv <- ml\_cross\_validator(sc, estimator = pipeline, estimator\_param\_maps = list(standard\_scaler = list(with\_mean = c(TRUE,   
 FALSE)), logistic\_regression = list(elastic\_net\_param = c(0.25, 0.75), reg\_param = c(0.01,   
 0.001))), evaluator = ml\_binary\_classification\_evaluator(sc, label\_col = "not\_working"),   
 num\_folds = 10)

estimator参数只是我们想要调整的评估器，在本例中，它是我们定义的管道。我们通过estimator\_param\_maps参数给我们感兴趣的超参提供数值，该参数采用一个嵌套的命名列表。第一层的名称对应UID，它是与每个我们要调试的管道阶段对象关联的唯一标识符（如果只提供了部分UID，sparklyr将尝试将其与管道阶段匹配）。而第二级的名称对应于每个阶段的参数。在开始的代码片段中，我们指定要测试以下内容：

* 标准标量 with\_mean的TRUE和FALSE值，表示预测变量的值是否中心化。
* 逻辑回归 的0.25和0.75值，的1e-2和1e-3值。

我们希望这些信息可以生成2 × 2 × 2 = 8种超参数组合，我们可以通过打印cv对象来确认：

cv  
CrossValidator (Estimator)  
<cross\_validator\_d5676ac6f5>  
 (Parameters -- Tuning)  
 estimator: Pipeline  
 <pipeline\_d563b0cba31>  
 evaluator: BinaryClassificationEvaluator  
 <binary\_classification\_evaluator\_d561d90b53d>  
 with metric areaUnderROC  
 num\_folds: 10  
 [Tuned over 8 hyperparameter sets]

有了其它的评估器，我们可以使用ml\_fit()拟合交叉验证器：

cv\_model <- ml\_fit(cv, okc\_train)

并查看结果：

ml\_validation\_metrics(cv\_model) %>%  
arrange(-areaUnderROC)  
 areaUnderROC elastic\_net\_param\_1 reg\_param\_1 with\_mean\_2  
1 0.7722700 0.75 0.001 TRUE  
2 0.7718431 0.75 0.010 FALSE  
3 0.7718350 0.75 0.010 TRUE  
4 0.7717677 0.25 0.001 TRUE  
5 0.7716070 0.25 0.010 TRUE  
6 0.7715972 0.25 0.010 FALSE  
7 0.7713816 0.75 0.001 FALSE  
8 0.7703913 0.25 0.001 FALSE

因为我们看到了实际工作中的管道API，因此让我们更加正式的介绍它们在不同的问题背景下是如何运行的：

## 操作模式

截止到目前，你很可能已经注意到，管道阶段函数，例如ft\_string\_indexer()和ml\_logistic\_regression()会根据传入的第一个变量返回不同类型的对象。表5-1给出了完整的模式。

表5-1. 机器学习函数的操作模式

|  |  |  |
| --- | --- | --- |
| 第一个变量 | 返回值 | 示例 |
| Spark连接 | 评估器或转换器对象 | ft\_string\_indexer(sc) |
| 管道 | 管道 | ml\_pipeline(sc) %>% ft\_string\_indexer() |
| 数据框，不带公式 | 数据框 | ft\_string\_indexer(iris, "Species", "indexed") |
| 数据框，带公式 | 数据框 | sparklyr ML模型对象 |

这些函数是使用S3实现的，S3是R提供的最流行的面向对象编程范例。就我们的目的而言，只需知道ml\_或ft\_函数的行为是由提供的第一个参数的类决定的就可以了。这可以使我们在不引入额外函数名的情况下提供很多功能。现在我们总结一下这些函数的行为：

* 如果提供了Spark连接，则函数返回一个转换器或评估器对象，它们可以直接通过ml\_fit()或ml\_transform()调用，也可以包含在管道中。
* 如果提供了管道，函数将返回一个追加有阶段的管道对象。
* 如果将数据框提供给特征转换函数（带有前缀ft\_的函数），或者不带公式的ML算法，则该函数将实例化管道阶段对象，必要时会完成拟合（如果阶段是评估器），然后转换返回数据框。
* 如果给支持公式接口的ML算法提供了数据框和公式，那么sparklyr会在后台构建管道模型，并返回包含其它元数据信息的ML模型对象。

公式接口方法是我们在第4章中学到的，这就是我们推荐给Spark新手的入门方法，因为它的语法类似于现有的R建模程序包，并且抽象掉了一些SPARM ML特性。然而，为了充分利用Spark ML的强大功能并利用管道进行工作流组织和交互性，学习ML管道的API还是很值得的。

随着管道基础知识的深入，我们现在开始讨论本章导言中提到的协作和模型部署话题。

## 交互性

管道最强大的功能之一是，它们可以序列化到磁盘，并且可以与其它Spark API（如Python和Scala）完全可交互。这意味着你可以在使用不同语言的Spark用户之间轻松地共享成果，其中可能包括其它数据科学家、数据工程师和部署工程师。要保存管道模型，请调用ml\_save()并提供路径：

model\_dir <- file.path("spark\_model")  
ml\_save(cv\_model$best\_model, model\_dir, overwrite = TRUE)  
Model successfully saved.

让我们看一下刚才写的目录：

list.dirs(model\_dir,full.names = FALSE) %>%  
head(10)  
 [1] ""  
 [2] "metadata"  
 [3] "stages"  
 [4] "stages/0\_string\_indexer\_5b42c72817b"  
 [5] "stages/0\_string\_indexer\_5b42c72817b/data"  
 [6] "stages/0\_string\_indexer\_5b42c72817b/metadata"  
 [7] "stages/1\_string\_indexer\_5b423192b89f"  
 [8] "stages/1\_string\_indexer\_5b423192b89f/data"  
 [9] "stages/1\_string\_indexer\_5b423192b89f/metadata"  
[10] "stages/2\_string\_indexer\_5b421796e826"

我们可以再试两个文件，看看保存了什么类型的数据：

spark\_read\_json(sc, file.path(  
file.path(dir(file.path(model\_dir, "stages"),  
pattern = "1\_string\_indexer.\*",  
full.names = TRUE), "metadata")  
)) %>%  
glimpse()  
Observations: ??  
Variables: 5  
Database: spark\_connection  
$ class <chr> "org.apache.spark.ml.feature.StringIndexerModel"  
$ paramMap <list> [["error", "drinks", "drinks\_indexed", "frequencyDesc"]]  
$ sparkVersion <chr> "2.3.2"  
$ timestamp <dbl> 1.561763e+12  
$ uid <chr> "string\_indexer\_ce05afa9899"  
spark\_read\_parquet(sc, file.path(  
file.path(dir(file.path(model\_dir, "stages"),  
pattern = "6\_logistic\_regression.\*",  
full.names = TRUE), "data")  
))  
# Source: spark<data> [?? x 5]  
 numClasses numFeatures interceptVector coefficientMatr… isMultinomial  
 <int> <int> <list> <list> <lgl>  
1 2 12 <dbl [1]> <-1.27950828662… FALSE

我们看到，从dplyr转换器中的SQL语句到逻辑回归的拟合系数估计，我们已经导出了相当多的信息。然后，我们可以（在新的Spark进程中）使用ml\_load()重建模型：

model\_reload <- ml\_load(sc, model\_dir)

让我们看看我们是否可以从管道模型中得到逻辑回归阶段：

ml\_stage(model\_reload, "logistic\_regression")  
LogisticRegressionModel (Transformer)  
<logistic\_regression\_5b423b539d0f>  
 (Parameters -- Column Names)  
 features\_col: features\_scaled  
 label\_col: not\_working  
 prediction\_col: prediction  
 probability\_col: probability  
 raw\_prediction\_col: rawPrediction  
 (Transformer Info)  
 coefficient\_matrix: num [1, 1:12] -1.2795 -0.0915 0 0.126 -0.0324 ...  
 coefficients: num [1:12] -1.2795 -0.0915 0 0.126 -0.0324 ...  
 intercept: num -2.79  
 intercept\_vector: num -2.79  
 num\_classes: int 2  
 num\_features: int 12  
 threshold: num 0.5  
 thresholds: num [1:2] 0.5 0.5

注意，导出的JSON和parquet文件与导出它们的API无关。这意味着，在多语言机器学习工程团队中，你可以从使用Python的数据工程师那里获取数据预处理管道，基于此构建预测模型，然后将最终管道交给使用Scala的部署工程。在下一节中，我们会详细讨论模型的部署。

为（使用公式接口创建的）sparklyr ml模型调用ml\_save()时，将保存关联的管道模型，但不会保存任何sparklyr相关的元数据，例如索引标签。换句话说，保存一个sparklyr ml\_model对象以及后续的加载将生成一个管道模型对象，就像是通过ml 管道API创建的一样。这个行为需要与其它编程语言一起使用管道。

在我们继续讨论如何在生产环境汇总运行管道之前，确保你断开了Spark：

spark\_disconnect(sc)

我们可以重启一个全新的环境，这也是你给生产环境部署管道时所期望的。

## 部署

我们刚刚介绍的内容值得强调：通过在ML管道框架内协作，我们减少了数据科学团队中不同角色之间的摩擦。特别的，我们减少了从建模到部署的时间。

在许多情况下，一个数据科学项目并不仅仅是以一个带有见解和建议的幻灯片结束。相反，手头的业务问题可能需要实时的按计划或按需求给新的数据点打分。例如，一家银行可能希望每晚评估其抵押贷款组合风险，或就信用卡申请提供即时决策。将模型转换为其它人可以使用的服务的过程通常称为*部署*或*产品化*。历史上，构建模型的分析人员和部署模型的工程人员之间存在巨大的鸿沟。前者可能使用R并开发了大量打分机制的文档，以便后者可以在C++或Java中重新实现模型。这种做法在一些组织中很可能需要几个月的时间，也并不盛行。但在Spark ML工作流中几乎都不必要。

前面提到的每晚投资组合风险和信贷应用评分示例代表了两种ML部署模式，即批处理和实时。简单的说，批处理是指同时处理许多记录，而且只要执行时间合理（通常以分钟到小时为单位），执行时间就不重要。另一方面，实时处理意味着一次给一条或几条记录打分，但延迟是至关重要的（在小于1秒的范围内）。现在让我们来看看如何把我们的OKCupid管道模型应用到“生产”中。

### 批打分

对于批打分和实时打分两种方法，我们将以超文本传输协议（Hypertext Transfer Protocol，HTTP）上API的形式将模型发布为web服务，这是软件通信的主要桥梁。通过提供API，其它服务或最终用户可以在不了解R或Spark的情况下利用我们的模型。R程序包plumber通过注释我们的预测函数让我们能够很容易做到这一点。

你需要确保安装了plumber, callr和httr程序包，代码如下：

install.packages(c("plumber", "callr", "httr"))

callr包支持在单独的R进程中运行R代码。这不是必须的，但我们将使用它在后台启动web服务。httr包允许我们使用R的web API。

在批量打分用例中，我们只需启动一个Spark连接并加载保存的模型。把以下脚本保存为*plumber/spark-plumber.R*：

library(sparklyr)  
sc <- spark\_connect(master = "local", version = "2.3")  
spark\_model <- ml\_load(sc, "spark\_model")  
# \* @post /predict  
score\_spark <- function(age, sex, drinks, drugs, essay\_length) {  
 new\_data <- data.frame(age = age, sex = sex, drinks = drinks, drugs = drugs,   
 essay\_length = essay\_length, stringsAsFactors = FALSE)  
 new\_data\_tbl <- copy\_to(sc, new\_data, overwrite = TRUE)  
 ml\_transform(spark\_model, new\_data\_tbl) %>% dplyr::pull(prediction)  
}

然后，我们可以通过执行下列代码初始化服务：

service <- callr::r\_bg(function() { p <- plumber::plumb("plumber/spark-plumber.R") p$run(port = 8000)  
})

这个代码会启动本地web服务，然后我们可以使用要打分的数据查询服务。然而，你可能需要等待几秒钟，让Spark服务完成初始化：

httr::content(httr::POST(  
"http://127.0.0.1:8000/predict",  
body = '{"age": 42, "sex": "m", "drinks": "not at all",  
 "drugs": "never", "essay\_length": 99}'  
))  
[[1]]  
[1] 0

这个结果告知我们，这个人很可能不是失业，也就是在业。我们现在可以通过停止callr服务来终止plumber服务：

service$interrupt()

如果我们对该操作进行计时（例如，使用system.time()），我们会看到延迟大约为数百毫秒级别，这可能适用于批处理应用，但不足以满足实时操作。主要的瓶颈是R数据框和Spark数据框之间的两个方向的序列化。此外，它还需要一个工作的Spark进程，这是一个很重的运行时要求。为了改善这些问题，我们接下来讨论一种更适合实时部署的方法。

### 实时打分

对于实时生产，我们希望尽可能减少依赖关系，以便能够针对更多的平台进行部署。现在我们介绍如何使用mleap程序包（它提供了*MLeap*库的接口）来序列化和服务Spark ML模型。MLeap是开源的（Apache License 2.0），支持很多功能，诸如Spark ML转换器。在运行时，环境的唯一需求是Java虚拟机（Java Virtual Machine，JVM）和MLeap运行时库。这既避免了Spark二进制文件，也避免了将数据转换为和转换出Spark数据框的昂贵开销。

由于mleap是sparklyr扩展和R程序包，所以首先需要从CRAN安装它：

install.packages("mleap")

然后它必须在调用spark\_connect()的时候加载。所以让我们重启R进程，建立新的连接[[19]](#footnote-143)，加载之前保存的管道模型：

library(sparklyr)  
library(mleap)  
sc <- spark\_connect(master = "local", version = "2.3")  
spark\_model <- ml\_load(sc, "spark\_model")

我们将模型保存为MLeap bundle格式的方法与使用Spark ML 管道API保存模型的方法非常相似；唯一的附加参数是sample\_input，它是一个Spark数据框，带有我们希望给新数据打分的模式：

sample\_input <- data.frame(sex = "m", drinks = "not at all", drugs = "never", essay\_length = 99,   
 age = 25, stringsAsFactors = FALSE)  
sample\_input\_tbl <- copy\_to(sc, sample\_input)  
ml\_write\_bundle(spark\_model, sample\_input\_tbl, "mleap\_model.zip", overwrite = TRUE)

现在我们可以在任何运行Java并具备开源mleap运行时依赖的设备中部署刚刚创建的结果*mleap\_model.zip*，而不需要Spark或R！事实上，我们可以继续断开与Spark的连接：

spark\_disconnect(sc)

在使用MLeap模型之前，确保已经安装运行时依赖：

mleap::install\_maven()  
mleap::install\_mleap()

要测试这个模型，我们可以创建一个新的plumber API，并完成发布。脚本*plumber/mleap-plumber.R*非常类似于之前的例子：

library(mleap)  
mleap\_model <- mleap\_load\_bundle("mleap\_model.zip")  
# \* @post /predict  
score\_spark <- function(age, sex, drinks, drugs, essay\_length) {  
 new\_data <- data.frame(age = as.double(age), sex = sex, drinks = drinks, drugs = drugs,   
 essay\_length = as.double(essay\_length), stringsAsFactors = FALSE)  
 mleap\_transform(mleap\_model, new\_data)$prediction  
}

我们启动服务的方法也一样：

service <- callr::r\_bg(function() { p <- plumber::plumb("plumber/mleap-plumber.R") p$run(port = 8000)  
})

我们可以运行之前编写的完全一样的代码，在新的服务中测试对失业人员的预测：

httr::POST(  
"http://127.0.0.1:8000/predict",  
body = '{"age": 42, "sex": "m", "drinks": "not at all",  
 "drugs": "never", "essay\_length": 99}'  
) %>%  
httr::content()  
[[1]]  
[1] 0

如果我们对这个操作计时，可以看到服务现在在几十毫秒内返回预测结果。

让我们停止服务，结束这一章：

service$interrupt()

## 小结

在本章中，我们讨论了Spark管道，这是第4章中介绍的建模函数背后的引擎。你学习了如何通过将数据处理和建模算法组织到管道中来整理预测建模工作流。你学习了管道还通过共享与语言无关的序列化格式促进了多语言数据科学和工程团队成员之间的协作，也就是说，你可以从R导出Spark管道，让其它人使用Python或Scala将你的管道重新加载到Spark中，这使得他们可以在不改变语言选择的情况下进行合作。

你还学习了如何使用mleap部署管道。它是一个Java运行时，提供了另一条产品化Spark模型的方法，即你可以导出管道并将其集成到支持Java的环境中，而不要求目标环境支持Spark或R。

你可能注意到一些算法，特别是无监督的学习算法，速度很慢。甚至对于可以加载到内存中的OKCupid数据集也是如此。如果我们能够访问一个合适的Spark集群，我们可以用更多的时间建模，用更少的时间等待！不仅如此，我们还可以使用集群资源运行更广泛的超参数优化作业并处理大型数据集。为了达到这个目的，第6章介绍了什么是计算集群，并解释了值得你考虑的各种选项，比如建立自己的集群或按需使用云集群。

# 第6章 集群

我有一支很强大的军队和一条很大的龙。 —Daenerys Targaryen

前几章集中讨论了在一个计算实例上,即你的个人计算机上，使用Spark。在本章中，我们将介绍在多个计算实例（也称为计算集群）上运行Spark的技术。本章和后续章节将介绍并使用适用于计算集群的概念；但是，并不需要使用计算集群来继续学习，因此你仍然可以使用个人计算机。值得一提的是，虽然前面的章节主要关注一个计算实例，但你也可以在计算集群中使用介绍的所有数据分析和建模技术，而无需更改任何代码。

如果你的组织中已经有了Spark集群，可以考虑跳过第7章，它会教你如何连接到现有的集群。否则，如果没有集群或正在考虑对现有基础设施的改进，本章将介绍目前可用的集群趋势、管理者和供应商。

## 概述

集群计算有三大趋势值得讨论：内部部署、云计算和Kubernetes。刻画这些在时间上的趋势将有助于我们了解它们是如何形成的，它们是什么，以及它们的未来可能是什么。为了说明这一点，图6-1使用来自Google trends的数据绘制了这些随时间变化的趋势图。

对于内部群集，你或你组织中的某个人购买了用于集群计算的物理计算机。集群中的计算机是由*现成*的硬件组成的，这意味着有人下订单购买了通常在商店货架上就可以找到的计算机；或者是*高性能*硬件组成的，这意味着计算供应商提供了高度定制的计算硬件，它还针对高性能网络连接、功耗等进行了优化。

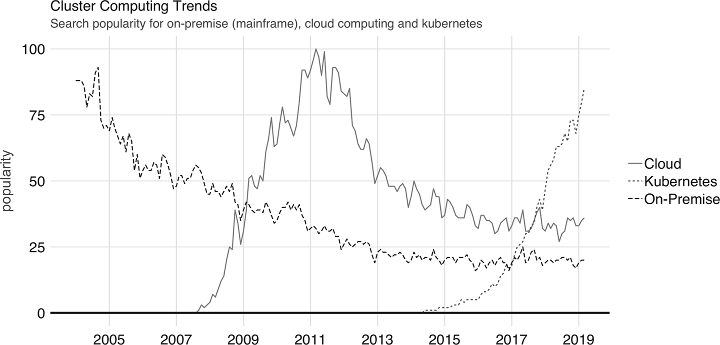


图6-1. 内部部署（主机），云计算和Kubernetes的Google trends

在购买成百上千个计算实例时，将它们放在我们熟知的常见计算机箱中是没有意义的；相反，尽可能高效地将它们堆叠在一起以最小化使用空间是有意义的。这组高效堆叠的计算实例称为*机架*。当集群增长到数千台计算机之后，你还需要托管数百个计算设备机架；在这种规模下，你还需要大量的物理空间来托管这些机架。

提供计算实例机架的建筑通常称为*数据中心*。在数据中心的规模上，你还需要找到提高建筑物效率的方法，特别是冷却系统、电源、网络连接等。由于这是一项耗时的工作，一些组织以开放计算项目计划（open Compute Project initiative）的名义联合起来开放其基础设施的源代码，该计划为任何人免费提供一组数据中心蓝图。

没有什么可以阻止你构建自己的数据中心。事实上，许多组织都已经开始了这条道路。例如，亚马逊最初是一家在线书店，但经过几年的发展，它的销售远远不止是书。随着在线商店的增长，其数据中心的规模也在增长。2002年，亚马逊考虑把数据中心的服务器出租给公众。两年后，亚马逊网络服务（Amazon Web Services，AWS）推出，让任何人都可以按需租用该公司数据中心的服务器，这意味着你你需购买、配置、维护或拆除自己的集群；相反，你可以直接从AWS租下来。

这种按需计算模型就是我们今天熟知的云计算。在云中，你使用的集群不归你所有，也不在你的物理建筑中；而是由其它人拥有和管理的数据中心。今天，这个领域有许多云提供商，包括AWS、Databricks、Google、Microsoft、Qubole和许多其它公司。大多数云计算平台通过web应用程序或命令行提供用户界面来请求和管理资源。

虽然多年来在云中处理数据的好处是显而易见的，但选择一个云提供商却产生了意想不到的副作用，即把组织锁定到一个特定提供商，使得很难在提供商之间切换或返回到本地群集。201年谷歌发布的Kubernetes是一个开源系统，用于跨多个主机管理容器化应用程序。实际上，它使跨多个云提供商和本地部署变得更容易。

总而言之，我们已经看到了从本地到云计算的转变，以及最近的Kubernetes。这些技术通常被简单地描述为私有云、公共云，以及启用混合云的编排服务之一。本章将带你了解Spark和R环境中的每一种集群计算趋势。

## 本地化

如概述部分所述，内部部署群集代表一组由组织中的工作人员采购和管理的计算实例。这些群集可以高度定制和控制；但是，它们也可能产生较高的初始费用和维护成本。

在使用本地Spark群集时，你应该考虑两个概念：

* 集群管理员 与操作系统（如Windows或macOS）允许你在同一台计算机上运行多个应用程序类似，集群管理器允许多个应用程序在同一个集群中运行。在使用内部部署集群时，你需要自己选择一个。
* Spark分发 虽然你可以从Apache Spark站点安装Spark，但是许多组织与能够为Apache Spark提供支持和增强的公司合作，我们通常称之为Spark分发。

### 管理器

要在一个计算集群上运行Spark,你需要运行能够在每一个物理机器上初始化Spark并注册所有可用计算节点的软件。这个软件就是集群管理器。Spark中可用的集群管理器包括Spark单机，YARN，Mesos和Kubernetes。

在分布式系统和集群文献中，我们经常把每一个物理机器叫做一个*计算实例*，*计算结点*或者*结点*。

#### 单机

在Spark单机中，Spark使用自己作为自己的集群管理器，这允许你无需在集群中安装其它软件的情况下使用Spark。如果你计划使用集群仅运行Spark应用程序，这可能很有用；如果该集群不只用于Spark，那么像YARN、Mesos或Kubernetes这样的通用集群管理器会更适合。Spark独立文档包含有关配置、启动、监视和启用高可用性的详细信息，如图6-2所示。

由于Spark单机包含在第2章介绍的Spark安装中，你可以使用Spark安装来初始化自己机器上的本地Spark 单机集群。实际工作中，你可能希望启动不同的机器上的工作节点，但为了简单起见，我们提供了在单个机器上启动单机集群的代码。

首先，通过运行spark\_home\_dir()来获取SPARK\_HOME目录，然后按如下代码启动主节点和工作节点：

# Retrieve the Spark installation directory  
spark\_home <- spark\_home\_dir()  
# Build paths and classes  
spark\_path <- file.path(spark\_home, "bin", "spark-class")  
# Start cluster manager master node  
system2(spark\_path, "org.apache.spark.deploy.master.Master", wait = FALSE)

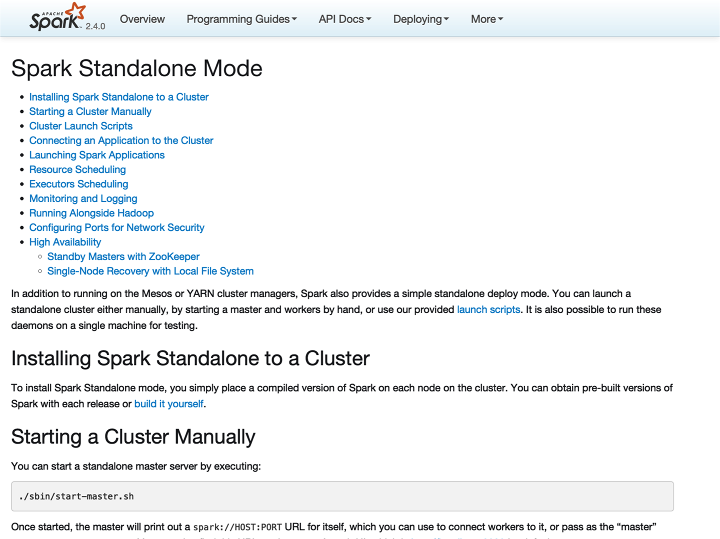
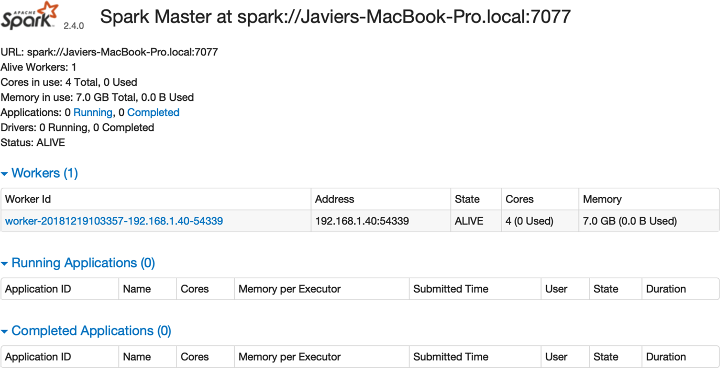


图6-2. Spark单机网站

上一个命令初始化主节点。你可以通过localhost:8080访问主节点接口，如图6-3所示。注意，Spark主URL被指定为Spark://address:port。 你需要此URL来初始化工作节点。

然后，我们可以使用主URL初始化单个工作节点；但是，你可以使用类似的方法，即多次运行代码，在不同的机器上初始化多个工作节点：

# Start worker node, find master URL at http://localhost:8080/  
system2(spark\_path, c("org.apache.spark.deploy.worker.Worker", "spark://address:port"),   
 wait = FALSE)

 Spark单机中只注册了一个工作节点。点击这个工作节点的链接，查看这个节点的细节，例如内存和处理器，如图6-4所示。 

在这个集群执行完计算之后，你需要停止主节点和工作节点。你可以使用jps命令来标识要终止的进程号。在下面的例子中，15330和15353是可以终止并结束集群的进程。要终止进程，可以在Windows中使用system("Taskkill /PID ##### /F")，在macOS或Linux中使用system("kill -9 #####")。

system("jps")  
15330 Master  
15365 Jps  
15353 Worker  
1689 QuorumPeerMain

你可以按照类似的方法配置集群，即在集群中的每台计算机上运行初始化代码。

虽然可以初始化一个简单的单机集群，但是要配置一个合适的Spark单机集群，满足从计算机重启和故障中恢复，并且支持多个用户、权限等，通常是一更加漫长的过程。这超出了本书的范围。以下各节介绍了几种更易于在本地或通过云服务进行管理的替代方案。我们首先介绍YARN。

#### YARN

Hadoop YARN，或者简称为YARN，是Hadoop项目的资源管理器。它最初是在Hadoop项目中开发的，但在Hadoop 2中被重构为自己的项目。正如我们在第1章中提到的，Spark是为了加快Hadoop上的计算速度而构建的，因此在Hadoop集群上安装Spark很常见。

YARN的一个优点是它可能已经安装在许多支持Hadoop的现有集群上，这意味着你可以轻松地使用Spark与许多现有的Hadoop集群，而不需要对现有集群的基础设施进行任何重大改变。Spark部署在YARN集群中也非常常见，因为许多集群最初是Hadoop集群，最终也会升级支持Spark。

你可以通过两种模式提交YARN应用：YARN客户端和YARN群集。在YARN集群模式下，驱动程序（可能）正在远程运行，而在YARN客户端模式下，驱动程序在本地运行。这两种模式都支持，我们将在第7章中进一步结束它们。

YARN提供了一个资源管理用户界面，可用于访问日志、监视可用资源、终止应用程序等。从R连接到Spark后，你能够用YARN管理正在运行的应用程序，如图6-5所示。

由于YARN是Hadoop项目中的集群管理器，因此可以在Hadoop.apache.org上找到YARN的文档。你还可以参考Spark.apache.org上的““Running Spark on YARN”指南。

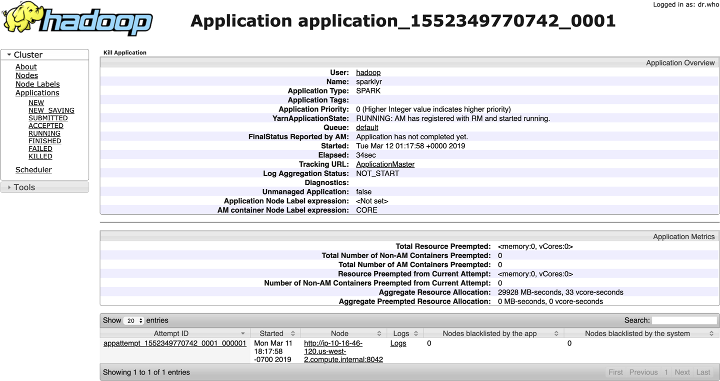


图6-5. 运行sparklyr应用的YARN资源管理器

#### Apache Mesos

Apache Mesos是一个用于管理计算机集群的开源项目。Mesos最初是加州大学伯克利分校RAD实验室的一个研究项目，它利用Linux Cgroup为CPU、内存、I/O和文件系统访问提供隔离。

与YARN一样，Mesos支持执行许多集群框架，包括Spark。然而，Mesos的一个优势是它允许像Spark这样的集群框架实现定制的任务调度器。调度器是集群中协调应用程序分配执行时间和资源的组件。Spark使用粗粒度的调度器，它根据应用程序的持续时间调度资源；但是，其它框架可能使用Mesos的细粒度调度器，它可以通过在较短的时间区间内调度任务来提高集群的整体效率，从而允许任务之间共享资源。

Mesos提供了一个web界面来管理正在运行的应用程序、资源等。从R连接到Spark后，你的应用程序像Mesos中运行的任何其它应用程序一样完成注册。图6-6显示了从R到Spark的成功连接。

Mesos是一个Apache项目，它的文档可以在Mesos.Apache.org上找到。如果你选择使用Mesos作为集群管理器，《Running Spark on Mesos》指南也是一个很好的资源。

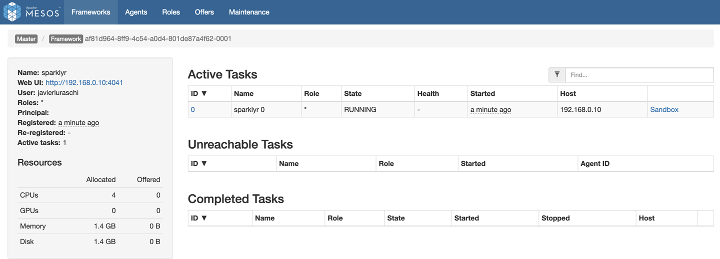


图6-6. 运行Spark和R的Mesos网络接口

### 发行版

正如前一节所介绍的，你可以在本地群集中使用群集管理器；但是，许多组织（包括但不限于Apache Spark）选择与提供附加管理软件、服务和资源的公司合作，以帮助管理其群集中的应用程序。一些本地集群提供商包括*Cloudera*、*Hortonworks*和*MapR*。下面我们会简要介绍它们。

Cloudera，Inc.是一家美国软件公司，为商业客户提供基于Apache Hadoop和Apache Spark的软件、支持、服务以及培训。Cloudera的混合开源Apache Hadoop发行版，即含Apache Hadoop的Cloudera发行版（CDH），目标成为该技术的企业级部署方案。Cloudera将其50%以上的工程输出捐赠给Apache许可的各种开源项目（Apache Hive、Apache Avro、Apache HBase等），这些项目结合在一起形成Apache Hadoop平台。Cloudera也是Apache软件基金会的赞助商。

Cloudera集群使用包。包是包含程序文件和元数据的二进制发行版。Spark恰好作为一个包安装在Cloudera上。介绍如何配置Cloudera集群超出了本书的范围，但是可以在Cloudera.com和Cloudera博客上的“Introducing sparklyr, an R Interface for Apache Spark”下找到资源和文档。

Cloudera提供Cloudera管理器网络接口来管理资源、服务、包、诊断等。图6-7显示了运行在Cloudera管理器中的Spark包，你可以稍后使用它连接R。

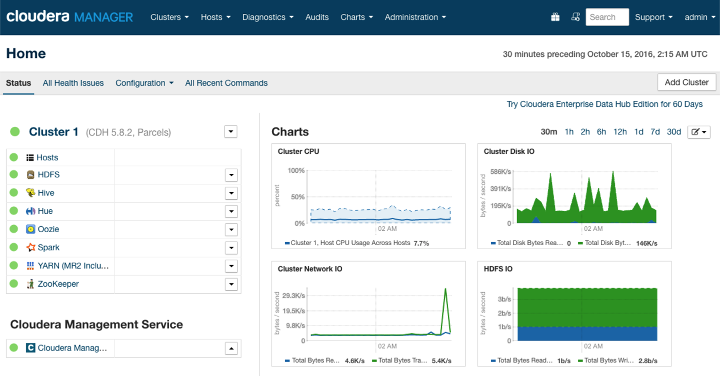
 sparklyr通过了Cloudera认证，意味着Cloudera也会支持sparklyr，可以有效帮助使用Spark和R的组织。表6-1总结了当前认证的版本。

表6-1. Cloudera认证d额sparklyr版本

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Cloudera版本 | 产品 | 版本 | 构件 | Kerberos |
| CDH5.9 | sparklyr | 0.5 | HDFS, Spark | 是 |
| CDH5.9 | sparklyr | 0.6 | HDFS, Spark | 是 |
| CDH5.9 | sparklyr | 0.7 | HDFS, Spark | 是 |

Hortonworks是一家总部位于加州圣克拉拉的大数据软件公司。公司开发、支持并提供一整套的完全开源的软件，旨在管理从物联网（IoT）到高级分析和机器学习等的数据和处理。Hortonworks相信它是一家连接云和数据中心的数据管理公司。

Hortonworks与微软合作，改善微软Windows对Hadoop和Spark的支持。这一点曾是区别于Cloudera的一个不同点；然而，自从两家公司于2019年1月合并以来，Hortonworks和Cloudera的比较在今天就不那么重要了。尽管已经合并，对Cloudera和Hortonworks的Spark发行版的支持仍然可用。在Hortonworks下配置Spark的其它资源可以在Hortonworks.com上找到。

MapR也是一家总部位于加利福尼亚州圣克拉拉的商业软件公司。MapR提供从单个计算机集群访问各种数据源的权限，包括大数据工作负载，如Apache Hadoop和Apache Spark，分布式文件系统，多模式数据库管理系统和事件流处理，实时分析与操作应用的结合。它的技术运行在商品硬件和公有云计算服务上。

## 云端

如果你既没有本地群集也没有备用计算机可重用，那么从云集群开始可能会非常方便，因为它允许你在几分钟内访问正确的集群。本节简要介绍了一些主要的云基础设施提供商，并提供了一些资源，帮助你在选择使用云提供商时上手。

在云服务中，计算实例的计费时间与Spark集群的运行时间一样长；当集群启动时开始计费，当集群停止时停止计费。此成本需要乘以为集群保留的实例数。因此，例如，如果一个云提供商每小时从每个计算实例收费1.00美元，而你启动一个三节点集群，使用一小时10分钟，那么你会收到1.00美元x2小时x 3节点=6.00美元的账单。 一些云提供商按分钟收费，但至少你可以采用他们每小时计算收费。

请注意，虽然小型集群的计算成本可能很低，但不经意地让集群一直运行可能会导致高额的帐单费用。因此，值得花额外的时间检查两次集群是否在不再需要时就可以终止。这也是每天使用集群时监测成本的优质实践，同时确保你的期望符合每天的账单。

根据以往的经验，在处理大型项目时，你还应该计划提前请求计算资源。但是在发出正式支持请求之前，各种云提供商不允许启动包含数百台计算机的集群。虽然这可能很麻烦，但它也是帮助你控制成本的一种方法。

由于集群的大小很灵活，从小型集群开始并按需扩展计算资源是一个很好的实践。即使你事先知道将需要一个很大规模的集群，从小规模开始也提供了一个以较低成本解决问题的机会，因为你的数据分析不太可能在第一次尝试大规模运行时毫无问题。经验法则是，以指数方式增长实例；如果需要在八节点群集上运行计算，需要从一个节点和整个数据集的八分之一开始，然后是两个节点和四分之一数据集，然后是四个节点和一半数据集，最后是八个节点和整个数据集。当你变得更有经验时，会对如何解决问题以及所需集群的大小有一个很好的了解，并且可以跳过中间步骤。但对于新手来说，这是一个很好的实践。

你还可以使用云提供商获取计算资源，然后自己安装上一节中介绍的内部部署发行版；例如，你可以在亚马逊的弹性计算云（Amazon Elastic Compute cloud，Amazon EC2）上运行Cloudera发行版。此模型可以避免获取同位置的硬件，但它仍然允许你密切管理和定制集群。本书只简单介绍了云提供商提供的完全托管的Spark服务。但是，你通常可以在网上轻松找到有关如何在云中安装本地发行版的说明。

云计算基础设施的主要供应商包括亚马逊、Databricks、谷歌、IBM、微软和Qubole。接下来的章节会简要介绍每一个供应商。

### 亚马逊

亚马逊通过AWS提供云服务。更具体的，它通过亚马逊EMR提供按需Spark集群。

使用R和亚马逊EMR的具体说明发表在Amazon的大数据博客中，名为“Running sparklyr on Amazon EMR”。这篇博文介绍了sparklyr的启动和通过其配置亚马逊集群的说明。例如，它推荐你可以使用亚马逊命令行界面启动三个节点的集群，代码如下：

aws emr create-cluster --applications Name=Hadoop Name=Spark Name=Hive \  
 --release-label emr-5.8.0 --service-role EMR\_DefaultRole --instance-groups \  
InstanceGroupType=MASTER,InstanceCount=1,InstanceType=m3.2xlarge \  
InstanceGroupType=CORE,InstanceCount=2,InstanceType=m3.2xlarge \  
 --bootstrap-action Path=s3://aws-bigdata-blog/artifacts/aws-blog-emr-\  
rstudio-sparklyr/rstudio\_sparklyr\_emr5.sh,Args=["--user-pw", "<password>", \  
"--rstudio", "--arrow"] --ec2-attributes InstanceProfile=EMR\_EC2\_DefaultRole

然后，你可以看到集群启动，并最终在AWS门户网站中运行，如图6-8所示。

接着，你可以导航至Master Public DNS，并在端口8787下找到RStudio，例如ec2-12-34-567-890.us-west-1.compute.amazonaws.com:8787，然后使用用户名hadoop和密码password登陆进入。

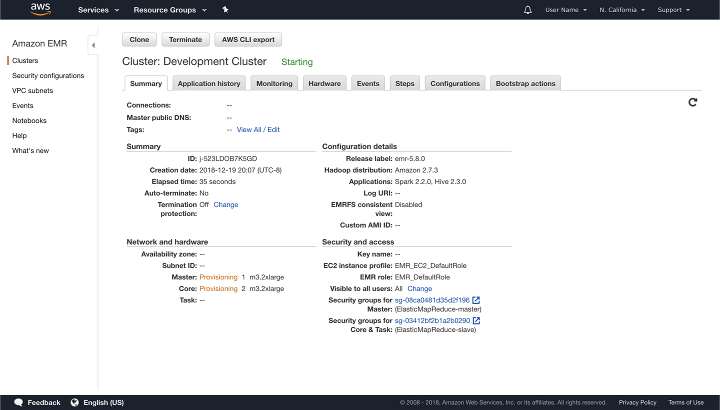


图6-8. 启动亚马逊EMR集群

你也可以使用网络界面启动亚马逊EMR集群。同一篇介绍性博文包含专门为亚马逊EMR设计的其它详细信息和步骤。

在启动亚马逊EMR集群进行敏感数据分析时，记住关闭集群以避免不必要的费用，并使用适当的安全限制。

关于成本，你可以在Amazon EMR Pricing上找到最新的信息。表6-2显示了us-west-1区域中（截至写书的时候）可用的一些实例类型；这是为了提供与云处理相关的资源和成本的简介。注意，“EMR价格是亚马逊EC2价格（底层服务器的价格）之外的价格。”

表6-2. 亚马逊EMR定价信息

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 实例 | CPU | 内存 | 存储 | EC2 | 成本 | EMR成本 |
| c1.medium | 2 | 1.7 GB | 350 GB | 0.148美元/小时 | 0.030美元/小时 |  |
| m3.2xlarge | 8 | 30 GB | 160 GB | 0.616美元/小时 | 0.140美元/小时 |  |
| i2.8xlarge | 32 | 244 GB | 6400 GB | 7.502美元/小时 | 0.270美元/小时 |  |

我们仅介绍了截至2019年亚马逊和后续云提供商提供额可用计算实例的一个子集；但是 需要注意，硬件（CPU速度、硬盘驱动器速度等）因供应商和位置而异。因此，你不能使用这些硬件表作为准确的价格比较。准确的比较需要运行特定的工作负载并考虑计算实例成本之外的其它方面。

### Databricks

Databricks是由Apache Spark的创建者创立的一家公司，其目标是帮助客户使用Spark进行基于云的大数据处理。Databricks源于加州大学伯克利分校的AMPLab项目。

Databricks提供了企业级的集群计算套餐，以及一个免费/社区版本，用于探索功能并熟悉其环境。

启动集群后，你可以按照第2章中提供的步骤或通过在数据块上安装RStudio来使用Databricks笔记本中的R和sparklyr。图6-9展示了通过sparkylyr使用Spark的Databricks笔记本。

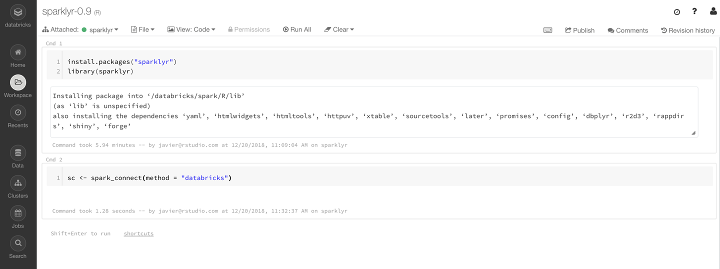


图6-9. 运行sparkylyr的Databricks社区版笔记本

其它资源可以在Databricks工程博客文章“Using sparklyr in Databricks”和sparklyr的Databricks文档下获得。

你可以在databricks.com/product/pricing上找到最新的定价信息。表6-3列出了截至本文撰写时可用的套餐。

表6-3. Databricks定价信息

|  |  |  |  |
| --- | --- | --- | --- |
| 套餐 | 基础定价 | 数据工程 | 数据分析 |
| AWS标准版 | 0.07美元/DBU | 0.20美元/DBU | 0.40美元/DBU |
| Azure标准版 |  | 0.20美元/DBU | 0.40美元/DBU |
| Azure高级版 |  | 0.35美元/DBU | 0.55美元/DBU |

注意，定价基于每小时DBU的成本。在Databricks中，“Databricks单位（DBU）是Apache Spark每小时处理能力的一个单位。对于一组不同的实例，DBU是一种更透明的查看使用情况的方法，而不是节点-小时。”

### 谷歌

谷歌提供谷歌云Dataproc，作为谷歌云平台（GCP）上提供的云端托管Spark和Hadoop的服务。Dataproc利用许多GCP技术，如谷歌计算引擎和g弍云存储，来提供运行主流数据处理框架（如Apache Hadoop和Apache Spark）的完全托管集群。

你可以从谷歌云控制台或谷歌云命令行界面（CLI）轻松创建集群，如图6-10所示。

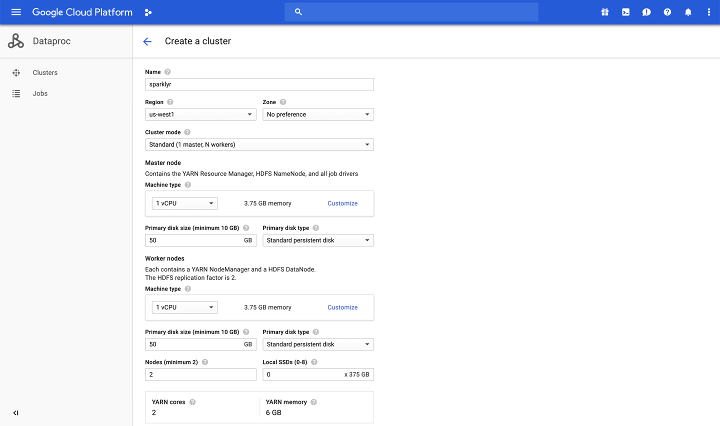


图6-10. 启动Dataproc集群

创建集群后，可以转发端口以允许从计算机访问此集群。例如，通过启动Chrome来使用此代理并安全地连接到Dataproc集群。配置此连接，代码如下：

gcloud compute ssh sparklyr-m --project=<project> --zone=<region> -- -D 1080 \  
 -N "<path to chrome>" --proxy-server="socks5://localhost:1080" \  
 --user-data-dir="/tmp/sparklyr-m" http://sparklyr-m:8088

有各种教程（cloud.google.com/dataproc/docs/tutorials）可以参考，包括配置RStudio和sparklyr的综合教程。

你可以在cloud.google.com/dataproc/pricing上找到最新的定价信息。在表6-4中，注意成本是在计算引擎和dataproc高级版之间分摊的。

Table 6-4. 谷歌云Dataproc定价信息

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实例 | CPU | 内存 | 计算引擎 | Dataproc高级版 |
| n1-standard-1 | 1 | 3.75 GB | 0.0475美元/小时 | 0.010美元/小时 |
| n1-standard-8 | 8 | 30 GB | 0.3800美元/小时 | 0.080美元/小时 |
| n1-standard-64 | 64 | 244 GB | 3.0400美元/小时 | 0.640美元/小时 |

### IBM

IBM云计算是一套面向业务的云计算服务。IBM云包括通过公共、私有和混合云交付模型提供的基础设施即服务（IaaS）的云、软件即服务的云（SaaS）和平台即服务的云（PaaS），以及组成这些云的组件。

在IBM云服务中，打开Watson Studio并创建一个数据科学项目，在项目设置下添加一个Spark集群，然后在Launch IDE菜单上启动RStudio。请注意，截至写书的时候，提供的sparklyr版本不是CRAN中提供的最新版本，因为sparklyr被修改了以便IBM云下运行。在任何情况下，请按照IBM的文档作为权威参考，在IBM云上运行R和Spark，特别是如何恰当的升级sparklyr。图6-11展示了启动Spark集群的IBM的云门户。

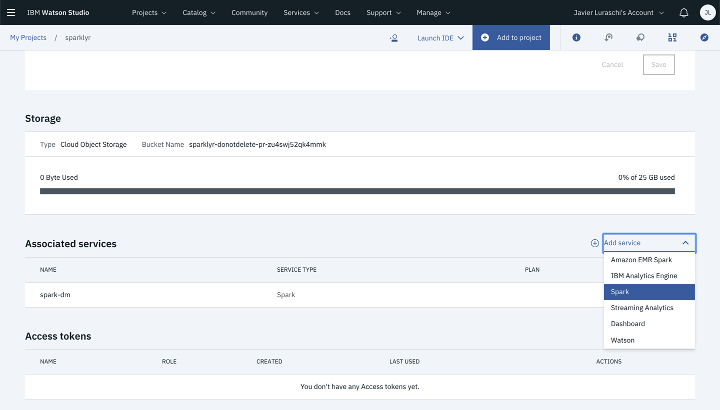


图6-11. 使用R启动Spark集群的IBM Watson

最新的定价信息可以在ibm.com/cloud/pricing获得。在表6-5中，计算成本使用每月成本对应的31天进行了归一。

表6-5. IBM云服务定价信息

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实例 | CPU | 内存 | 存储 | 成本 |
| C1.1c1x25 | 1 | 1 GB | 25 GB | 0.033美元/小时 |
| C1.4x4x25 4 4 GB | 25 GB | 0.133美元/小时 |  |  |
| C1.32x32x25 | 32 | 25 GB | 25 GB | 0.962美元/小时 |

### 微软

微软Azure是由微软创建的云计算服务，用于通过微软管理的数据中心全球网络来构建、测试、部署和管理应用程序和服务。它提供SaaS、PaaS和IaaS，并支持许多不同的编程语言、工具和框架，包括微软特定的和第三方软件和系统。

通过Azure门户，Azure HDInsight服务为定制化Spark集群提供支持。通过选择ML Services集群类型，可以轻松创建支持Spark和RStudio的HDInsight集群。注意，提供的sparklyr版本可能不是CRAN中的最新版本，因为默认的程序包仓库似乎是使用微软R应用程序网络（MRAN）而初始化的，而不是直接从CRAN初始化的。图6-12显示了使用R启动Spark集群的Azure门户。

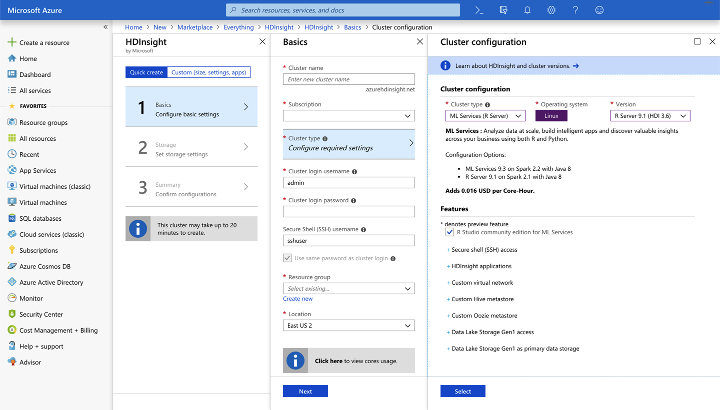
 HDInsight的最新定价信息可以在azure.microsoft.com/en-us/pricing/details/hdinsight找到。表6-6列出了写书之时的报价。

表6-6. Azure HDInsight定价信息

|  |  |  |  |
| --- | --- | --- | --- |
| 实例 | CPU | 内存 | 总成本 |
| D1 v2 | 1 | 3.5 GB | 0.074美元/小时 |
| D4 v2 | 8 | 28 GB | 0.59美元/小时 |
| G5 | 64 | 448 GB | 9.298美元/小时 |

### Qubole

Qubole成立于2013年，其使命是缩小数据的可访问性差距。Qubole为基于亚马逊、微软、谷歌和甲骨文云的大数据分析提供了一个自助服务平台。在Qubole，你可以启动Spark集群，支持通过Qubole笔记本或RStudio服务器使用。图6-13展示了一个用RStudio和sparklyr初始化的Qubole集群。

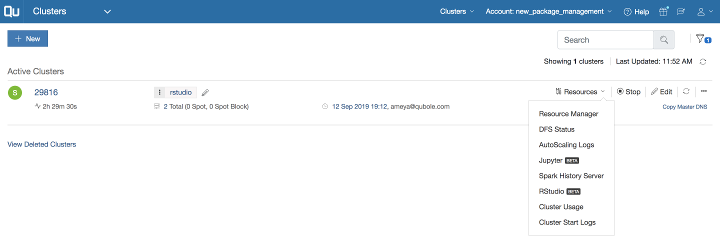


图6-13. 用RStudio和sparklyr初始化的Qubole集群

你可以在Qubole的定价页面上找到最新的定价信息。表6-7列出了截至写书的时候Qubole当前套餐的价格。注意，定价基于QCU/hr的成本，即“每小时Qubole计算单位”，而企业版本需要一份年度合同。

表6-7. Qubole定价信息

|  |  |  |
| --- | --- | --- |
| 测试驱动 | 全功能试用版 | 企业版 |

0美元 0美元 0.14美元/QCU

## Kubernetes

Kubernetes是一个开源的容器编排系统，用于容器化应用程序的自动部署、扩展和管理，最初由Google设计，现在由云原生计算基金会（Cloud Native Computing Foundation，CNCF）维护。Kubernetes最初基于Docker，而和Mesos一样，它现在也基于Linux Cgroups。

Kubernetes可以执行许多集群应用和框架，你可以通过使用具有特定资源和库的容器镜像进行高度定制。这可以让Kubernetes集群用于数据分析之外的许多不同目的，进而又帮助组织轻松地管理他们的计算资源。使用自定义镜像需要权衡增加了进一步的配置开销的问题，但又使Kubernetes集群非常灵活。然而，这种灵活性证明，在许多组织中都有助于轻松管理集群资源。正如之前“概述”中所指出的，Kubernetes正在成为一个非常流行的集群框架。

所有主要的云提供商都支持Kubernetes。它们都提供了关于如何启动、管理和拆除Kubernetes集群的大量文档；图6-14展示了创建Kubernetes集群时的GCP控制台。你可以在任何Kubernetes集群上部署Spark，并且可以使用R来完成连接、分析、建模等操作。

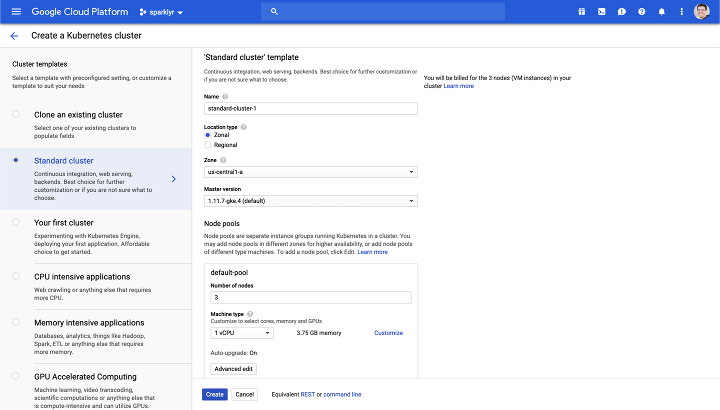


图6-14. 使用谷歌云为Spark和R创建Kubernetes集群

你可以在kubernetes.io上了解更多信息，并通过Spark.apache.org阅读《kubernetes运行Spark指南》。

严格地说，Kubernetes是一种集群技术，而不是一种特定的集群体系结构。然而，Kubernetes代表了一个更大的趋势，通常被称为混合云。混合云是一种计算环境，它利用内部部署和公有云服务，在不同平台之间进行调度。现在，要准确的对形成集群计算混合方法的主要技术进行分类还为时尚早。尽管前面提到，Kubernetes是领先的方案，但还有更多技术可能形成补充，甚至替代现有技术。

## 工具

虽然对于一些集群来说，仅使用R和Spark就足够了，但通常在集群中还会安装补充工具,例如分别使用Ganglia、Hue和Oozie等应用来改进监视、SQL分析、工作流协调等。本节并不打算涵盖所有工具，而是介绍常用的工具。

### RStudio

通过第1章的学习，你知道RStudio是R的一个著名的免费桌面开发环境；因此，很可能大家正在使用RStudio 桌面版完成本书中的示例。但是，你可能不知道RStudio也可以作为web服务在Spark集群中运行。这个版本的RStudio称为RStudio Server。你可以在图6-15中看到正在运行的RStudio Server。与Spark UI在集群中运行的方式相同，你可以在集群中安装RStudio Server，然后连接到RStudio Server，并按照RStudio Desktop完全相同的方式使用RStudio，同时支持针对Spark集群运行代码。如图6-15所示，RStudio Server不管看起来，还是感觉起来都像RStudio Desktop，但是它支持通过所在的集环境中来高效的运行命令。

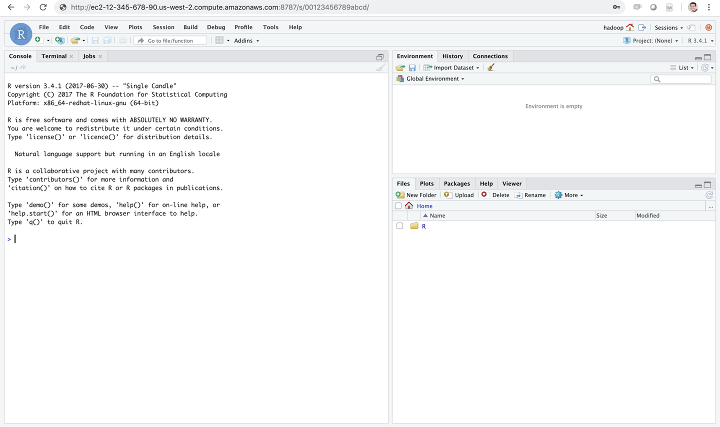


图6-15. 运行在AWS的RStudio Server专业版

如果你熟悉R的话，Shiny服务是从R构建交互式web应用程序的一个非常流行的工具。我们建议你直接在Spark集群中安装Shiny。

RStudio和Shiny服务器是免费开源的产品；但是，RStudio还提供RStudio Server、RStudio Server Pro、Shiny Server Pro和RStudio Connect等专业产品。你可以在集群内安装这些产品以支持其它R工作流程。sparklyr不需要任何额外的工具，但它们带来了值得考虑的生产力显著提高。你可以在rstudio.com/products/上了解更多相关信息。

### Jupyter

Juyter项目为开发开源软件、公开标准和跨几十种编程语言的交互式计算服务而生。Jupyter笔记本支持多种编程语言，包括R。你可以使用R内核实现sparklyr与Jupyter笔记本一起使用。图6-16展示了sparklyr在本地Jupyter笔记本中运行的情况。

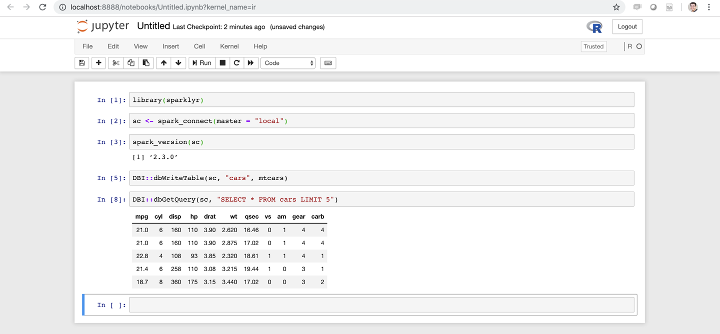


图6-16. 运行sparklyr的Jupyter笔记本

### Livy

Apache Livy是Apache中的一个孵化项目，支持通过web界面远程使用Spark集群。直接连接到Spark集群是最理想的；但是，有时直接连接到Spark集群是不可行的。当面临这些限制时，你可以考虑在集群中安装Livy并对其进行适当的安全保护，以便能够通过web协议进行远程使用。不过，请注意，在sparklyr中使用Livy会带来明显的性能开销。

为了帮助在本地测试Livy，sparklyr通过执行livy\_available\_versions()列出、安装、启动和停止本地Livy实例：

## livy 1 0.2.0 2 0.3.0 3 0.4.0 4 0.5.0

这个命令可以列出可安装的版本。我们建议安装最新的版本并完成验证，代码如下：

# Install default Livy version  
livy\_install()  
114 | Chapter 6: Clusters  
# List installed Livy services  
livy\_installed\_versions()  
# Start the Livy service  
livy\_service\_start()

你可以导航至本地http://localhost:8998 上的Livy进程。第7章会具体介绍如何通过Livy连接。连接成功后，你可以导航至Livy web应用，如图6-17所示。

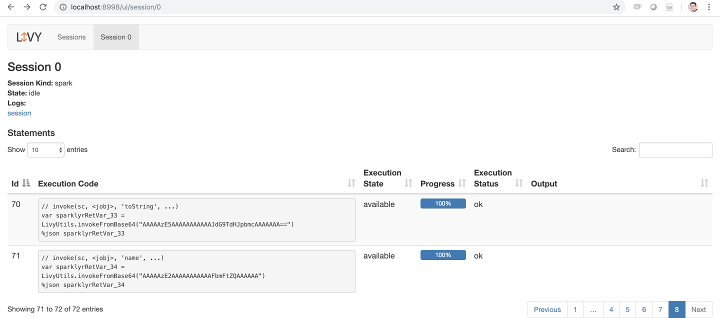


图6-17. 作为本地服务运行的Apache Livy

确保你在使用本地Livy实例的时候已经终止了Livy服务（而对于集群中运行的正常Livy服务，你无需这么做）：

# Stops the Livy service  
livy\_service\_stop()

## 小结

本章介绍了内部部署和云计算的历史和权衡，以及Kubernetes这一提供跨内部部署或多个云提供商之间灵活性的有前途的框架。本章还引入了集群管理器（Spark Standalone、YARN、Mesos和Kubernetes），作为运行Spark集群应用所需的软件。本章还简要介绍本地集群提供商，如Cloudera、Hortonworks和MapR，以及主要的Spark云提供商：亚马逊、Databricks、IBM、谷歌、微软和Qubole。

虽然本章为理解当前集群计算趋势、工具和服务提供商在处理大规模数据科学项目上提供了坚实的基础，但它并没有提供一个全面的框架来帮助你决定选择哪种集群技术。相反，可以将本章作为概述和起点来寻找其它资源，以帮助找到最适合组织需要的集群。

第7章将重点介绍如何连接到现有的集群，因此，就像我们在本章中介绍的,它假定Spark集群已经可用。

# 第7章 连接

他们没有选择的余地。 —Daenerys Targaryen

第6章介绍了主要的集群计算趋势、集群管理器、发行版和云服务提供商，以帮助你选择最适合需要的Spark集群。相比之下，本章会介绍Spark集群内部组件以及如何连接到特定Spark集群。

阅读本章时，无需执行每一行代码。这会非常困难，因为你需要准备不同的Spark环境。相反，如果你已经有了一个Spark集群，或者上一章让你有足够的动力去注册一个按需的集群，那么现在是学习如何连接集群的时候了。本章帮助你连接到已经选择的集群。如果还没有选择集群，我们建议你学习这些概念，然后回来运行相关代码。

此外，本章还提供了各种连接故障排除技术。虽然我们不希望你需要使用它们，但还是保证你可以运用这些技术解决连接问题。

虽然本章的内容可能有点枯燥，因为连接和故障排除绝对不是大规模计算中最让人兴奋的内容，但是本章还是会介绍Spark集群的构成和交互方式,通常被称为Apache Spark*体系结构*。本章以及第8章和第9章将详细介绍Spark的工作原理，这将帮助你成长为Spark的中级用户，能够真正理解使用Apache Spark进行分布式计算的精彩内容。

## 概述

Spark集群的所有连接体系结构由三种类型的计算实例组成：*驱动节点*、*工作节点*和*集群管理器*。集群管理器是一种允许Spark在集群中执行的服务。这部分内容第之前的“管理器”中有详细说明。工作节点（也称为执行器）在分区数据上执行计算任务，并将中间结果传递给其它工作节点或返回给驱动节点。驱动节点的任务是将工作委托给工作节点，同时还负责聚合它们的结果和控制计算流。在大多数情况下，聚合发生在工作节点中；然而，即使在节点聚合数据之后，驱动节点通常也需要收集工作节点的结果。因此，驱动节点通常至少配备（通常更多）一些计算资源（内存、CPU、本地存储等），可以作为工作节点。

严格地说，驱动节点和工作节点只是分配了具有特定角色的机器的名称，而驱动节点中的实际计算则根据*SparkContext*执行的。SparkContext是Spark功能的主要入口，因为它的任务是调度任务、管理存储、跟踪执行状态、指定访问配置的设置、取消作业等。在工作节点中，实际计算是在spark执行器下执行的，spark执行器是一个spark组件，其任务是针对特定的数据分区执行子任务。图7-1说明了这个概念，其中驱动节点通过集群管理器编排一个工作节点的工作。

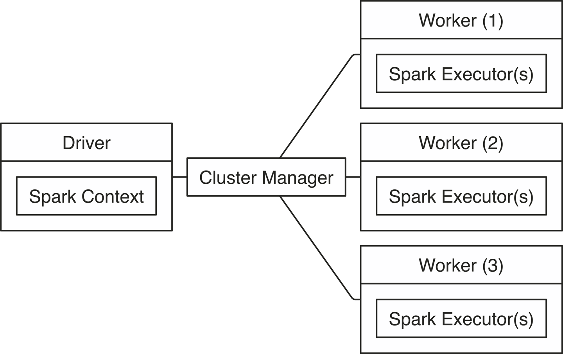


图7-1. Apache Spark连接体系结构

如果你的组织中已经有一个Spark集群，那么你应该向集群管理员请求指向该集群的连接信息，仔细阅读它们的使用策略，并遵循它们的建议。因为集群可以在许多用户之间共享，所以你需要确保只请求所需的计算资源。我们将在第9章中介绍如何请求资源。系统管理员将指定它是本地集群还是云集群、使用的集群管理器、支持的连接和支持的工具。你可以使用此信息直接跳转到本地、单机、YARN、Mesos、Livy或Kubernetes，根据这些信息可以选择适合需求的方案。

使用spark\_connect()连接后，可以借助sc连接方法使用在前面章节中描述的技术；例如，你可以使用前面章节中介绍的相同代码进行数据分析或建模。

### 边缘节点

计算集群会完成配置，支持节点之间的大带宽和快速网络连接。为了优化网络连接，集群中的节点被配置为相互信任并禁用安全功能。这会提高性能，但需要你关闭所有外部网络通信，从而使整个集群成为一个安全的整体，只有少数集群计算机在精心配置后接受来自集群外部的连接。理论上讲，这些计算机位于集群的“边缘”，称为*边缘节点*。因此，在连接到Apache Spark之前，可能首先需要连接到集群中的边缘节点。有两种连接方法：

* 终端 使用计算机终端应用程序，可以使用Secure Shell建立到集群的远程连接；连接到集群后，可以启动R，然后使用sparklyr。但是，终端对于一些任务，如探索性数据分析来说可能会很麻烦，因此它通常只在配置集群或故障排查时使用。
* 网络浏览器 虽然可以从终端使用sparklyr，但在边缘节点安装web服务器通常会更高效，该节点提供从网络浏览器运行R和sparklyr的访问权限。很可能，你会考虑使用RStudio或Jupyter，而不是从终端连接。

图7-2形象的解释了这些概念。左边部分通常是你的网络浏览器，右边部分是边缘节点。在使用网路浏览器的时候，客户端和边缘节点使用HTTP通信；在使用终端的时候，客户端和边缘节点使用Secure Shell （SSH）通信。

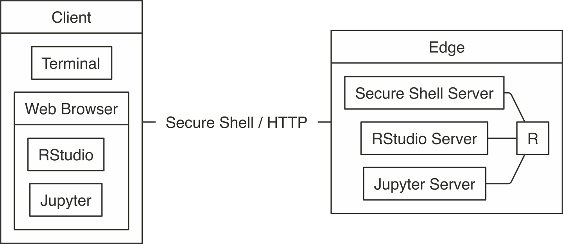


图7-2. 连接到Spark边缘节点

### Spark主目录

在连接到边缘节点之后，下一步是确定Spark安装在哪里，一个名为SPARK\_HOME的位置。在大多数情况下，集群管理器已经将SPARK\_HOME环境变量设置为正确的安装路径。如果没有，你需要得到正确的*SPARK\_HOME*路径。必须指定SPARK\_HOME路径作为环境变量，或者在运行spark\_connect()时显式地指定spark\_connect()路径。

如果你的集群提供商或集群管理员已经为提供了*SPARK\_HOME*，那么下面的代码应该返回路径而不是空字符串：

Sys.getenv("SPARK\_HOME")

如果这段代码返回了空字符串，这就意味着集群中的SPARK\_HOME环境变量没有设置。所以你需要在使用spark\_connect()的时候指定SPARK\_HOME，代码如下：

sc <- spark\_connect(master = "master", spark\_home = "local/path/to/spark")

在这个例子里，设置master，为Spark单机，YARN，Mesos，Kubernetes或Livy指向集群管理器主节点。

## 本地模式

当你以本地模式连接到Spark时，Spark会像SparkContext和单个执行一样，启动一个运行大多数集群组件的进程。这对于在大型计算集群上运行Spark、脱机工作、故障排查或测试代码非常理想。图7-3描述了Spark的本地连接。

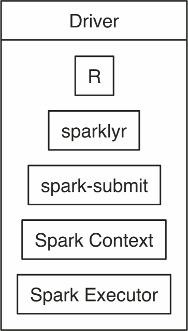


图7-3. 本地连接图

注意，本地模式下既没有集群管理器，也没有工作进程，因为在本地模式下，所有工作都在驱动应用程序中运行。另外值得注意的是，sparklyr通过spark-submit启动SparkContext 。spark-submit是每个Spark安装程序中都提供的一个脚本，允许用户向Spark提交自定义应用程序。如果你好奇的话，第13章会解释sparklyr中发生的提交此应用程序并从R正确连接的内部过程。

要执行本地连接，我们可以使用前面章节中熟悉的以下代码：

# Connect to local Spark instance  
sc <- spark\_connect(master = "local")

## 单机模式

连接到Spark单机集群需要集群管理器主节点实例的位置。就像之前在“单机”部分介绍的，你可以在集群管理器的web界面中找到该实例。你可以通过查找以spark://开头的URL来找到此位置。

单机模式下的连接从sparklyr开始，它会启动spark-submit，然后提交sparklyr应用程序并创建SparkContext。该Context从运行在给定主节点地址下的Spark单机实例中请求执行器。图7-4说明了此过程，这与图7-1中的总体连接架构非常相似，但有一些单机集群和sparklyr特有的附加细节。

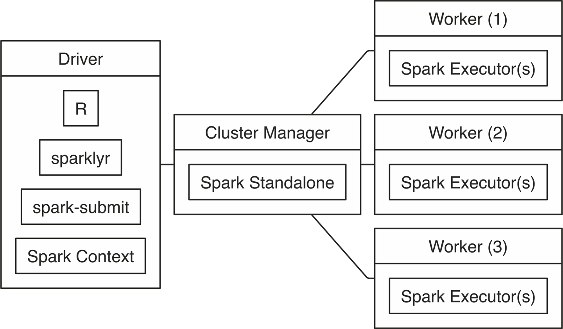


图7-4. Spark单机连接图

要完成连接，使用spark\_connect()中的master = "spark://hostname:port"，代码如下：

sc <- spark\_connect(master = "spark://hostname:port")

## YARN

Hadoop YARN是Hadoop项目中的集群管理器。这是集群中最常见的集群管理器，最初是Hadoop集群；之后是和亚马逊EMR类似的Cloudera、Hortonworks和MapR发行版。YARN支持两种连接方式：YARN客户端和YARN集群。然而，YARN客户端模式比YARN集群更常见，因为它更高效，更容易建立。

### YARN客户端

当你使用YARN客户端模式连接时，驱动实例运行R，sparklyr和SparkContext。这些操作需要YARN工作节点来运行Spark执行器，如图7-5所示。

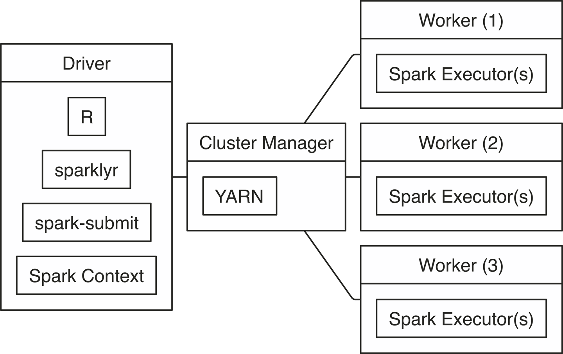


图7-5. YARN客户端连接图

要完成连接，你只需运行master = "yarn"，代码如下：

sc <- spark\_connect(master = "yarn")

当你使用客户端模式运行YARN时，集群管理器会在后台执行你希望集群管理器执行的操作：它从集群分配资源并将它们分配给Spark应用程序。SparkContext会为你管理这些应用。在之前的的“YARN”部分中有一点需要注意，SparkContext位于运行R代码的同一台机器中；这与集群模式下运行YARN有所不同。

### YARN集群

在集群模式下运行YARN与在客户端模式下运行YARN的主要区别在于，在集群模式下，不要求驱动节点是启动R和sparklyr的节点。相反，驱动节点仍然是专门指定的，通常与运行R的边缘节点不同。当边缘节点有太多并发用户、缺少计算资源或诸如RStudio或Jupyter的工具需要区别于其它集群资源管理工具进行独立管理时，考虑使用集群模式可能会有帮助。

图7-6展示了在集群模式下运行时不同组件是如何分离的。注意，仍然有一条连接客户端和集群管理器的线路，因为首先，仍然需要从集群管理器分配资源；但是，在分配资源之后，客户端直接与驱动节点通信，驱动节点与工作节点通信。从图7-6中，你可能认为集群模式看起来比客户端模式复杂得多。这种认识是正确的；因此，如果可能的话，最好避免集群模式，因为它有额外的配置开销。

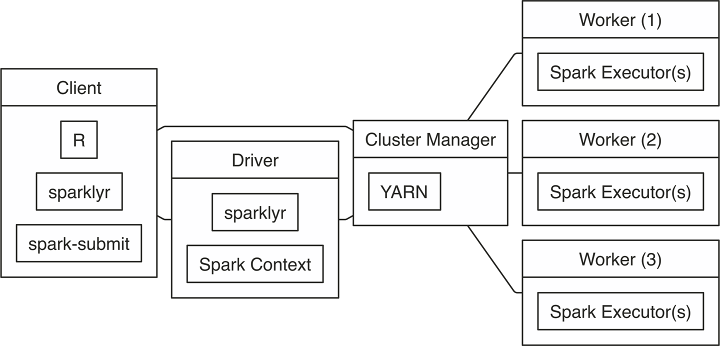


图7-6. YARN集群连接图

为了连接YARN集群节点，只需要运行下列代码：

sc <- spark\_connect(master = "yarn-cluster")

集群模式假定运行spark\_connect()的节点都正确配置好了，这意味着存在yarn-site.xml，并且YARN\_CONF\_DIR环境变量都配置正确。。当使用Hadoop作为文件系统时，还需要HADOOP\_CONF\_DIR环境变量配置正确。此外，你需要确保客户端和驱动节点之间的网络连接正确，这不仅仅是需要两台机器都可以访问，而且还需要确保它们之间有足够的带宽。此配置通常由系统管理员提供，不需要手动配置。

## Livy

与其它需要在集群中使用边缘节点的连接方法不同，Livy提供了一个web API，使Spark集群可以从集群外部访问，并且不需要在客户端中安装Spark。在通过web API连接之后，*Livy服务*通过从集群管理器请求资源并像往常一样分发工作来启动SparkContext。图7-7展示了一个Livy连接；注意，客户端通过web API远程连接到驱动节点。

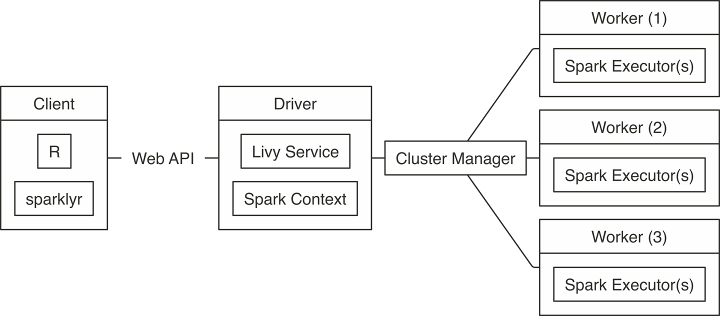


图7-7. Livy连接图

通过Livy连接需要指向Livy服务的URL，应该与https://hostname:port/livy类似。因为允许远程连接，连接通常至少需要基本的认证：

sc <- spark\_connect(master = "https://hostname:port/livy", method = "livy", config = livy\_config(spark\_version = "2.3.1",   
 username = "<username>", password = "<password>"))

要在本地机器上使用Livy，你可以想之前在“Livy”中描述的那样，安装运行Livy服务并进行连接，代码如下：

sc <- spark\_connect(master = "http://localhost:8998", method = "livy", version = "2.3.1")

通过Livy连接后，你可以使用sparklyr的任何特性；但是，Livy不适合探索性数据分析，因为执行命令会带来很大的性能开销。虽然这么说，但在长时间运行计算时，这种开销可能被认为是无关紧要的。一般来说，你应该避免使用Livy并直接在集群中的边缘节点内工作；当这种策略不可行时，使用Livy才是一种合理的方法。

通过spark\_version参数指定Spark版本是可选的；但是，当指定了版本时，部署与给定版本兼容的预编译Java二进制文件会显著提高集群性能。因此，在使用Livy连接到Spark时，最好指定Spark版本。

## Mesos

与YARN类似，Mesos支持客户端模式和集群模式；但是，sparklyr目前仅支持Mesos下的客户端模式。因此，图7-8所示的图相当于YARN客户端图，只有集群管理器从YARN更改为Mesos。

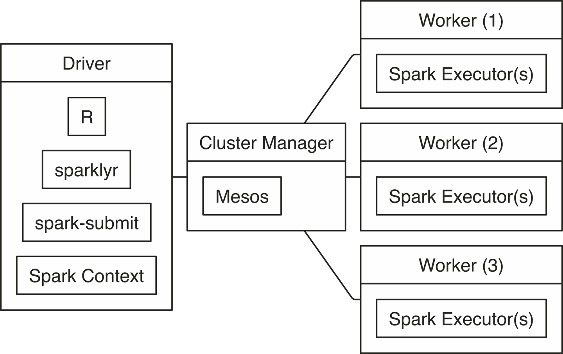


图7-8. Mesos连接图

连接需要指向Mesos主节点的网址。对于使用ZooKeeper的Mesos，通常形式是mesos://host:port或者mesos://zk://host1:2181,host2:2181,host3:2181/mesos。

sc <- spark\_connect(master = "mesos://host:port")

The MESOS\_NATIVE\_JAVA\_LIBRARY需要经过系统管理员的设置，或者在本地机器上运行Mesos的时候手动设置。例如，在MacOS中，你可以用终端安装和初始化，然后通过手动设置Mesos库，并使用spark\_connect()进行连接：

brew install mesos  
/usr/local/Cellar/mesos/1.6.1/sbin/mesos-master --registry=in\_memory  
 --ip=127.0.0.1 MESOS\_WORK\_DIR=. /usr/local/Cellar/mesos/1.6.1/sbin/mesos-slave  
 --master=127.0.0.1:5050  
  
Sys.setenv(MESOS\_NATIVE\_JAVA\_LIBRARY =  
"/usr/local/Cellar/mesos/1.6.1/lib/libmesos.dylib")  
sc <- spark\_connect(master = "mesos://localhost:5050",  
spark\_home = spark\_home\_dir())

## Kubernetes

Kubernetes并不像Mesos或者YARN支持客户端模式。相反，连接模型类似于YARN集集群，其中的驱动节点由Kubernetes指定，如图7-9所示。

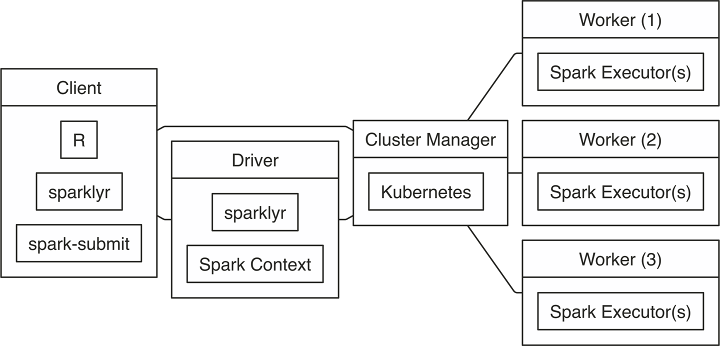


图7-9. Kubernetes连接图

要使用Kubernetes，你需要准备一个安装了Spark并完成正确配置的虚拟机；但是，本书不讨论如何创建一个虚拟机。完成创建后，Kubernetes连接的流程如下：

library(sparklyr)  
sc <- spark\_connect(config = spark\_config\_kubernetes("k8s://https://<apiserver-host>:<apiserver-port>",   
 account = "default", image = "docker.io/owner/repo:version", version = "2.3.1"))

如果你的计算机已经完成使用Kubernetes集群的配置，你可以使用下面的命令找出apiserver-host和apiserver-port：

system2("kubectl", "cluster-info")

## 云模式

当你使用云服务提供商时，有不同的连接方式。例如，从Databricks连接需要下面的方法：

sc <- spark\_connect(method = "databricks")

因为亚马逊EMR使用YARN，因此你可以使用master = "yarn"进行连接：

sc <- spark\_connect(master = "yarn")

当使用IBM的Watson Studio连接到Spark时，需要你通过IBM提供的load\_spark\_kernels()函数获取一个配置对象：

kernels <- load\_spark\_kernels()  
sc <- spark\_connect(config = kernels[2])

使用微软的Azure HDInsights和ML服务（R服务器）时，Spark连接按照以下方式初始化：

library(RevoScaleR)  
cc <- rxSparkConnect(reset = TRUE, interop = "sparklyr")  
sc <- rxGetSparklyrConnection(cc)

从Qubole连接需要使用qubole连接方法：

sc <- spark\_connect(method = "qubole")

如果需要帮助，参考你的云服务提供商文档和支持渠道。

## 批量模式

大多数情况下，你可以交互使用sparklyr；也就是说，你可以显式地与spark\_connect()连接，然后执行命令来分析和建模大规模数据。但是，你也可以通过调度使用sparklyr的spark作业来自动化进程。Spark不提供调度数据处理任务的工具；相反，你要使用其它工作流管理工具。这对于转换数据、准备模型和隔夜数据打分，或者其它系统使用Spark非常有用。

例如，你可以创建名为batch.R的文件，其中包含以下内容：

library(sparklyr)  
sc <- spark\_connect(master = "local")  
sdf\_len(sc, 10) %>% spark\_write\_csv("batch.csv")  
spark\_disconnect(sc)

然后，你可以使用spark\_submit()给Spark提交这个申请。master参数应该完成设置：

spark\_submit(master = "local", "batch.R")

你也可以从Shell直接调用spark-submit代码如下：

/spark-home-path/spark-submit  
 --class `sparklyr`.Shell '/spark-jars-path/sparklyr-2.3-2.11.jar'  
8880 12345 --batch /path/to/batch.R

最后一个参数表示端口号8880和进程号12345。你可以将其设置为任何唯一的数字标识符。使用以下R代码获得正确的路径：

# Retrieve spark-home-path  
spark\_home\_dir()  
# Retrieve spark-jars-path  
system.file("java", package = "sparklyr")

你可以通过给spark\_submit()传递额外的命令行参数定制你的脚本，并使用R的commandArgs()读取出来。

## 工具

使用Jupyter和RStudio等工具连接到Spark集群时，你可以运行本章中介绍的相同连接参数。但是，由于许多云提供商使用web代理来保护Spark的web界面，要使用spark\_web()或RStudio Connections窗口扩展，你需要正确配置sparklyr.web.spark，然后通过config参数将其传递给spark\_config()。

例如，在使用亚马逊EMR时，可以通过动态检索YARN应用程序和构建EMR代理URL来配置sparklyr.web.spark和sparklyr.web.yarn：

domain <- "http://ec2-12-345-678-9.us-west-2.compute.amazonaws.com"  
config <- spark\_config()  
config$sparklyr.web.spark <- ~paste0(domain, ":20888/proxy/", invoke(spark\_context(sc),   
 "applicationId"))  
config$sparklyr.web.yarn <- paste0(domain, ":8088")  
sc <- spark\_connect(master = "yarn", config = config)

## 多次连接

通常，我们只会连接Spark一次，而且只有一次。但是，你也可以通过连接到不同的集群或指定app\_name参数来打开多次Spark连接。这有助于在提交到集群之前比较Spark版本或验证你的分析结果。以下示例打开了与Spark 1.6.3、2.3.0和Spark单击的连接：

# Connect to local Spark 1.6.3  
sc\_16 <- spark\_connect(master = "local", version = "1.6")  
# Connect to local Spark 2.3.0  
sc\_23 <- spark\_connect(master = "local", version = "2.3", appName = "Spark23")  
# Connect to local Spark Standalone  
sc\_standalone <- spark\_connect(master = "spark://host:port")

最后，你可以断开每个连接：

spark\_disconnect(sc\_1\_6\_3)  
spark\_disconnect(sc\_2\_3\_0)  
spark\_disconnect(sc\_standalone)

你也可以一次性断开所有连接：

spark\_disconnect\_all()

## 故障排除

最后，我们介绍以下故障排除技术：日志记录、Spark Submit和Windows。如果不确定从何处开始，请在使用Windows系统时从Windows部分开始，然后是日志记录，最后是Spark Submit。当运行spark\_connect()失败并显示错误消息时，这些技术非常有用。

### 记录日志

第一项故障排除技术是直接打印Spark日志到控制台，帮助你定位额外的错误消息：

sc <- spark\_connect(master = "local", log = "console")

另外，你可以通过设定连接时sparklyr.verbose选项为TRUE来开启详细日志：

sc <- spark\_connect(master = "local", log = "console", config = list(sparklyr.verbose = TRUE))

### Spark Submit

通常，你可以通过spark\_submit()运行一个示例任务，判断一个连接问题是R导致的还是Spark导致的，并说明没有抛出错误：

# Find the spark directory using an environment variable  
spark\_home <- Sys.getenv("SPARK\_HOME")  
# Or by getting the local spark installation  
spark\_home <- sparklyr::spark\_home\_dir()

然后，使用故障排除中正确的master参数替换"local"计算圆周率例子：

# Launching a sample application to compute Pi  
system2(  
file.path(spark\_home, "bin", "spark-submit"),  
c(  
"--master", "local",  
"--class", "org.apache.spark.examples.SparkPi",  
dir(file.path(spark\_home, "examples", "jars"),  
pattern = "spark-examples", full.names = TRUE),  
100),  
stderr = FALSE  
)  
Pi is roughly 3.1415503141550314

如果前面的消息没有显示，你需要调查为什么你的Spark集群没有正确配置。这超出了本书的范围。首先，重新运行圆周率例子，但是要删除stderr = FALSE；这会将错误消息打印到控制台，然后你可以使用控制台的消息来调查问题可能的原因。当使用云提供商或Spark发行版时，你可以联系他们的支持团队帮助进一步解决此问题；另外，Stack Overflow也是一个很好的调查问题的起点。如果你确实看到消息，这意味着你的Spark群集已正确配置，但是不知道为什么故R无法使用Spark，因此你还需要详细地解决此问题，我们接下来会解释。

#### 故障排除细节

要详细排除连接过程的故障，可以手动复制两步连接过程，这通常对诊断连接问题非常有帮助。首先，从R调用spark\_submit()，将应用提交给spark；其次，R连接到正在运行的spark应用程序。

首先，通过运行以下命令来标识Spark安装目录和正确sparklyr\*.jar文件路径：

dir(system.file("java", package = "sparklyr"), pattern = "sparklyr", full.names = T)

确保你找到了匹配Saprk的正确版本。例如，sparklyr-2.1-2.11.jar匹配Spark 2.1。

然后，在终端中运行以下代码：

$SPARK\_HOME/bin/spark-submit --class `sparklyr`。Shell $PATH\_TO\_SPARKLYR\_JAR 8880 42  
18/06/11 12:13:53 INFO sparklyr: Session (42) found port 8880 is available  
18/06/11 12:13:53 INFO sparklyr: Gateway (42) is waiting for `sparklyr` client  
 to connect to port 8880

参数8880表示sparklyr中使用的默认端口，而42是进程号，这是sparklyr生成的加密安全号，但出于故障排除的目的，可以简化为42。

如果第一个连接步骤失败，这意味着集群无法接受应用程序。这通常意味着没有足够的资源，或者存在权限限制。

第二步是按如下方式从R连接（注意有60秒的超时，在运行终端命令后需要运行R命令；如果需要，你可以按照第9章中的说明配置超时参数）：

library(sparklyr)  
sc <- spark\_connect(master = "sparklyr://localhost:8880/42", version = "2.3")

如果第二个连接步骤失败，通常意味着R和驱动节点之间存在连接问题。例如，你可以尝试使用其它连接端口。

### Windows

在大多数情况下，Windows连接和从Linux和macOS连接一样简单。但是，有几个常见的连接问题需要注意：

* 防火墙和防病毒软件可能会阻止连接端口。sparklyr使用的默认端口是8880；请仔细检查此端口是否未被阻止。
* 长路径名可能会导致问题，特别是对于像Windows 7这样的旧Windows系统。使用这些系统时，请尝试与所有文件夹一起安装的Spark连接，最多使用8个字符，名称中不要有空格。

## 小结

本章介绍了Spark的体系结构、连接概念和在本地模式、单机模式、YARN模式、Mesos模式、Kubernetes模式和Livy模式下连接的示例，并介绍了边缘节点及其在连接到Spark集群时的作用。这些内容应该可以给你提供足够的信息，以便成功连接到任何Apache Spark集群。

要解决本章介绍的技术之外的连接问题，我们建议你在Stack Overflow、sparklyr问题的GitHub页面中搜索连接问题。如果需要，在sparklyr中打开一个新的GitHub问题以进一步获得帮助。

在第8章中，我们会介绍如何使用Spark读写来自从各种数据源和格式的数据，这可以让你在数据分析时更加灵活的添加新的数据源。过去需要几天，几周，甚至几个月完成的事情，现在你可以通过拥抱数据湖在数小时内完成。

# 第8章 数据

你有没有想过她可能不是一个可靠的信息来源？ —Jon Snow

根据前面章节中获得的知识，你现在可以进行大规模数据分析和建模了！然而，到目前为止，我们还没有真正解释过如何将数据读入Spark。我们已经探索了如何使用copy\_to()上传小数据集或使用函数，如spark\_read\_csv()或 spark\_write\_csv()，但没有详细解释过程和原因。

所以，你会学习到如何使用Spark读写数据。尽管这一点很重要，但本章还会向介绍数据湖——按照自然或原始格式存储的数据仓库，它提供了有别于现有存储体系结构的优点。例如，你可以轻松地集成来自外部系统的数据，而无需将其转换为通用格式，也无需假定这些源与内部数据源一样可靠。

此外，我们还将讨论如何扩展Spark的能力，使其能够处理无法直接访问的数据，并提出一些提高数据读写性能建议。读取大型数据集通常需要微调Spark集群配置，但这是第9章的主题。

## 概述

在第1章中，你学习到除了大数据和大计算之外，还可以使用Spark来提高数据任务的速度、多样性和准确性。虽然你可以将本章的知识用于任何需要加载和存储数据的任务中，但在处理各种数据源的时，本章的内容尤其有趣。为了理解原因，我们首先应该快速看一下，当今数据在许多组织中是如何处理的。

多年来，人们经常把大型数据集存储在关系*数据库*中，这是Edgar F.Codd于1970年首次提出的一种系统[[20]](#footnote-216)。你可以将数据库视为相互关联的表的集合，其中每个表都经过精心设计以保存特定的数据类型和与其它表的关系。大多数关系数据库系统使用*结构化查询语言（Structured Query Language，SQL）*来查询和维护数据库。数据库在今天仍然被广泛使用，这是有充分理由的：它们可靠且一致地存储数据；事实上，你的银行很可会将帐户余额存储在数据库中，这是一种很好的实践。

然而，数据库也被用来存储来自其它应用程序和系统的信息。例如，你的银行还可能存储其它银行生成的数据，例如传入的支票。要做到这一点，需要从外部系统中提取外部数据，转换成适合当前数据库的数据，最后加载到数据库。这称为*提取*、*转换*和*加载*（ETL），是一种将数据从一个或多个源复制到目标系统的通用过程，该目标系统以不同于源的方式表示数据。ETL过程从20世纪70年代开始流行。

除了数据库之外，数据通常还加载到*数据仓库*中，数据仓库是一个用于报告和数据分析的系统。数据通常以提高数据分析速度的格式存储和索引，但这种格式通常不适合建模或运行自定义的分布式代码。挑战在于，更改数据库和数据仓库通常是一个漫长而微妙的过程，因为需要重新建立数据索引，并且需要小心地将来自多个数据源的数据转换为单个跨数据源共享的表。

不要试图将所有数据源转换为通用格式，你可以在一个数据湖、即一个以其自然格式存储的数据系统或存储库中包含各种数据源（见图8-1）。由于数据湖允许使数据以其原始格式存储，所以不必预先对其进行仔细的转换，任何人都可以使用它进行分析，这就大大增加了ETL的灵活性。然后，你可以使用Spark通过一个可在所有数据池、数据库和数据仓库中扩展的单一接口来统一数据处理。一些组织还使用Spark替换现有ETL过程；然而，这属于数据工程领域，远远超出了这本书的范围。我们在图8-1中用虚线来说明这一点。

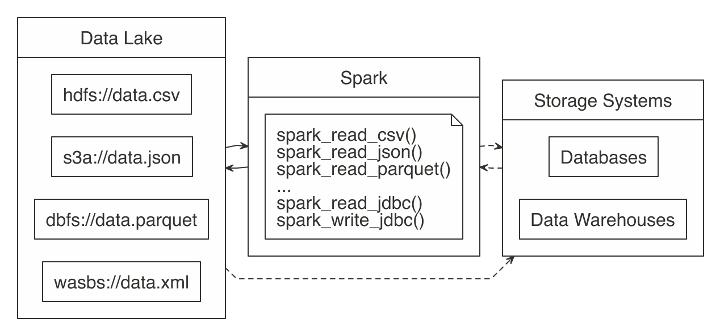


图8-1. Spark处理来自数据湖，数据库和数据仓库的原始数据

为了支持各种各样的数据源，Spark需要能够读写几种不同的文件格式（CSV、JSON、Parquet和其它格式），并访问存储在几个文件系统（HDFS、S3、DBFS等）中的数据，并且可能与其它存储系统（数据库、数据仓库等）进行交互。我们会讨论所有这些内容，但首先，我们会介绍如何使用Spark读取、写入和复制数据。

## 读取数据

如果你是Spark新手，强烈建议你在开始使用大型数据集之前查阅本节。我们将介绍几种提高数据读取速度和效率的技术。每个子章节都介绍了利用Spark读取文件的具体方法，例如可以将整个文件夹视为数据集，以及可以描述数据集以便在Spark中更快地读取。

### 路径

在分析数据时，通常将多个文件加载到单个数据对象中。在R中，我们通常使用循环或函数式编程指令来实现这一点。这是因为R必须单独加载每个文件传递到R进程中。让我们在文件夹中创建几个CSV文件，并先用R进行读取：

letters <- data.frame(x = letters, y = 1:length(letters))  
dir.create("data-csv")  
write.csv(letters[1:3, ], "data-csv/letters1.csv", row.names = FALSE)  
write.csv(letters[1:3, ], "data-csv/letters2.csv", row.names = FALSE)  
do.call("rbind", lapply(dir("data-csv", full.names = TRUE), read.csv))  
 x y  
1 a 1  
2 b 2  
3 c 3  
4 a 1  
5 b 2  
6 c 3

在Spark中，将一个文件夹作为数据集是一个重要的思想。我们只需传递包含所有文件的路径就可以，而不是枚举每个文件。Spark假设该文件夹中的每个文件都是同一数据集的一部分。这意味着目标文件夹只能用于数据目的。这一点尤其重要，因为像HDFS这样的存储系统会存储跨多台计算机的文件，但从概念上讲，它们存储在同一个文件夹中；当Spark从这个文件夹中读取文件时，实际上是执行分布式代码来读取每台计算机中的每个文件。当读取分布式文件的时候，并没有数据转移：

library(sparklyr)  
sc <- spark\_connect(master = "local", version = "2.3")  
spark\_read\_csv(sc, "data-csv/")  
# Source: spark<datacsv> [?? x 2]  
 x y  
 <chr> <int>  
 1 a 1  
 2 b 2  
 3 c 3  
 4 d 4  
 5 e 5  
 6 a 1  
 7 b 2  
 8 c 3  
 9 d 4  
10 e 5

“文件夹即表”的思想也可以在其它开源技术中找到。Hive表的后台工作原理是一样的。当你查询一个Hive表时，映射会在同一文件夹中的多个文件上完成。文件夹的名称通常与用户可见的表的名称匹配。

接下来，我们会介绍一种技术，它允许Spark更快地读取文件，并通过预先描述数据集的结构来避免读取失败。

### 模式

在读取数据时，Spark能够确定数据源的列名和列类型，也称为模式。然而，猜测模式是有代价的；Spark需要初步传递一些数据来猜测它是什么。对于大型数据集，这会给数据读取过程增加大量时间，即使对于中等大小的数据集，这也会增加成本。对于反复读取的文件，额外的读取时间会随着时间的推移而累积。

为了避免这种情况，Spark允许通过提供一个columns参数定义列，来描述数据集。你自己可以对原始文件的一小部分进行采样来创建这个模式：

spec\_with\_r <- sapply(read.csv("data-csv/letters1.csv", nrows = 10), class)  
spec\_with\_r  
 x y  
 "factor" "integer"

你也可以把列的说明来设置为一个显式包含列类型的向量。这个向量的值按照与字段名匹配的方式命名：

spec\_explicit <- c(x = "character", y = "numeric")  
spec\_explicit  
 x y  
"character" "numeric"

可接受的变量类型有：integer，character，logical，double，numeric，factor，Date和 POSIXct。

然后，在使用spark\_read\_csv()读取时，可以spec\_with\_r传递给columns参数，以匹配原始文件的名称和类型。这有助于提高性能，因为Spark不需要确定列类型。

spark\_read\_csv(sc, "data-csv/", columns = spec\_with\_r)  
# Source: spark<datacsv> [?? x 2]  
 x y  
 <chr> <int>  
1 a 1  
2 b 2  
3 c 3  
4 a 1  
5 b 2  
6 c 3

下面的示例演示如何将字段类型设置为不同的类型。新的字段类型需要与原始数据集兼容。例如，不能将character字段设置为numeric。如果使用不兼容的类型，文件读取会失败并出现错误。此外，以下示例同时会更改原始字段的名称：

spec\_compatible <- c(my\_letter = "character", my\_number = "character")  
spark\_read\_csv(sc, "data-csv/", columns = spec\_compatible)  
  
# Source: spark<datacsv> [?? x 2]  
 my\_letter my\_number  
 <chr> <chr>  
1 a 1  
2 b 2  
3 c 3  
4 a 1  
5 b 2  
6 c 3

在Spark中，格式错误的记录会在读取过程中导致错误，特别是对于非字符型字段。为了防止这种错误，我们可以使用一个文件说明，将字段作为字符型导入，然后使用dplyr将字段强制为所需的类型。本小节回顾了如何能够更快地读取文件且失败更少，从而使我们能够更快地开始分析。另一种加速分析的方法是将更少的数据加载到Spark内存中，我们会在下一节中介绍这个话题。

### 内存

默认情况下，当使用Spark和R环境读取数据时，数据被复制到Spark的分布式内存中，使得数据分析和其它操作非常快速。有些情况下，例如当数据太大时，加载所有数据可能不实际，甚至不必要。对于这些情况，Spark只需“映射”文件，而无需将数据复制到内存中。

映射在Spark中创建了一种虚拟表。这意味着，当查询该表时，Spark需要从文件中读取数据。任何后续读取都将执行相同的操作。实际上，Spark变成了高速的数据传递。这种方法的优点是几乎不需要提前“读取”文件；映射速度非常快。缺点是运行实际提取数据的查询需要更长的时间。

这由读取函数的memory参数控制。将其设置为FALSE可防止数据复制（默认值为TRUE）：

mapped\_csv <- spark\_read\_csv(sc, "data-csv/", memory = FALSE)

这个方法有一些很好的用例，其中一个用例是不需要表的所有列。例如，以包含许多列的非常大的文件为例。假设这不是你第一次与这些数据交互，那么你可能知道分析需要哪些列。当你知道需要哪些列时，可以使用memory=FALSE读取文件，然后使用dplyr选择所需的列。然后，可以使用compute()函数将生成的dplyr变量缓存到内存中。这会告诉Spark查询文件，抽取所选字段，并只将数据复制到内存中。结果是，读取时间相对较短的内存中的表：

mapped\_csv %>% dplyr::select(y) %>% dplyr::compute("test")

下一节会介绍一个小技术，使得读取导入数据的原始字段名更加容易。

### 列

Spark 1.6要求对列名进行清理，所以默认情况下R是这样做的。在某些情况下，你可能希望保持原始名称的完整性，或者在使用Spark 2.0或更高版本。为此，可以把sparklyr.sanitize.column.names选项设置为FALSE：

options(sparklyr.sanitize.column.names = FALSE)  
copy\_to(sc, iris, overwrite = TRUE)  
# Source: table<iris> [?? x 5]  
# Database: spark\_connection  
 Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 <dbl> <dbl> <dbl> <dbl> <chr>  
 1 5.1 3.5 1.4 0.2 setosa  
 2 4.9 3 1.4 0.2 setosa  
 3 4.7 3.2 1.3 0.2 setosa  
 4 4.6 3.1 1.5 0.2 setosa  
 5 5 3.6 1.4 0.2 setosa  
 6 5.4 3.9 1.7 0.4 setosa  
 7 4.6 3.4 1.4 0.3 setosa  
 8 5 3.4 1.5 0.2 setosa  
 9 4.4 2.9 1.4 0.2 setosa  
10 4.9 3.1 1.5 0.1 setosa  
# ... with more rows

学习完如何给Spark读取数据，我们继续介绍如何从Spark进程中写入数据。

## 写入数据

有些项目要求将Spark中生成的新数据写回远程的数据源。例如，数据可以是由Spark模型返回的新预测值。Spark作业会处理大量的预测生成任务，预测值需要存储。本节重点介绍如何使用Spark将数据从Spark传递到外部目的地。

许多新用户首先会把Spark数据下载到R中，然后将其上传到目标数据地址，如图8-2所示。它适用于较小的数据集，对于较大的数据集效率会变得低下。数据通常会增长到R不再可行的程度。

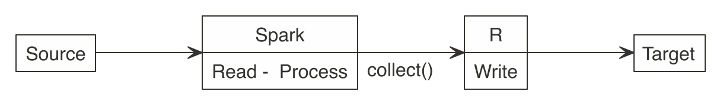


图8-2. 写入大型数据集时Spark的错误使用

应尽一切努力将Spark连接到最终目标位置。这样，读、处理和写就发生在同一个Spark进程中。如图8-3所示，更好的方法是使用Spark来读取、处理和写入目标。这种方法能够扩展到Spark集群能力所及的范围，并防止R成为瓶颈。

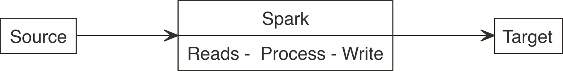


图8-3. 写入大型数据集时Spark的正确使用

考虑以下场景：一个Spark作业刚刚处理了一个大型数据集的预测任务，产生了大量的预测值。选择写入结果的方法取决于你正在使用的基础技术架构。更具体地说，它将取决于Spark和目标位置是否在同一集群中运行。

回到我们的场景，Spark中有一个需要保存的大型数据集。当Spark和目标位置在同一个集群中时，复制结果是没有问题的；数据传输是在同一集群的内存和磁盘之间进行的，或者通过高速连接高效的完成。但如果目标不在Spark集群内怎么办？有两个选项，需要根据数据大小和网络速度选择一个：

* Spark传输 在这种情况下，Spark连接到远程目标位置并复制新数据。如果这是在同一个数据中心或云提供商内完成的，那么数据传输速度可能足够快，保证Spark可以直接写入数据。
* 外部传输和其它方式 Spark可以将结果写入磁盘并通过第三方应用进行传输。Spark将结果作为文件写入，然后一个单独的作业完成文件复制。在目标位置，你可以使用单独的进程将数据传输到目标位置。

最好意识到Spark、R和任何其它技术都是工具。没有工具可以做任何事情，你也不应该期望它能做任何事。接下来我们将描述如何将数据复制到Spark或收集内存无法加载的大型数据集，这些数据集可用于跨集群传输数据或帮助初始化分布式数据集。

## 复制数据

前几章使用copy\_to()作为复制数据到Spark的便捷助手；但是，你只能使用copy\_to()来传输已经加载到内存中的内存数据集。这些数据集往往比要复制到Spark中的数据集要小得多。

例如，假设我们生成了一个3GB的数据集，如下所示：

dir.create("largefile.txt")  
write.table(matrix(rnorm(10 \* 10^6), ncol = 10), "largefile.txt/1", append = T, col.names = F,   
 row.names = F)  
for (i in 2:30) file.copy("largefile.txt/1", paste("largefile.txt/", i))

如果驱动节点中只有2GB内存，则无法使用copy\_to()将此3GB文件加载到内存中。相反，当使用HDFS作为集群中的存储时，可以使用hadoop命令行工具将文件从磁盘复制到Spark中，如下所示。请注意，以下代码仅在使用HDFS的集群中有效，而在本地环境中无效。

hadoop fs -copyFromLocal largefile.txt largefile.txt

然后你可以按照“文件格式”部分介绍的，读取上传的数据集。对于文本文件，你可以运行：

spark\_read\_text(sc, "largefile.txt", memory = FALSE)  
# Source: spark<largefile> [?? x 1]  
 line  
 <chr>  
 1 0.0982531064914565 -0.577567317599452 -1.66433938237253 -0.20095089489…  
 2 -1.08322304504007 1.05962389624635 1.1852771207729 -0.230934710049462 …  
 3 -0.398079835552421 0.293643382374479 0.727994248743204 -1.571547990532…  
 4 0.418899768227183 0.534037617828835 0.921680317620166 -1.6623094393911…  
 5 -0.204409401553028 -0.0376212693728992 -1.13012269711811 0.56149527218…  
 6 1.41192628218417 -0.580413572014808 0.727722566256326 0.5746066486689 …  
 7 -0.313975036262443 -0.0166426329807508 -0.188906975208319 -0.986203251…  
 8 -0.571574679637623 0.513472254005066 0.139050812059352 -0.822738334753…  
 9 1.39983023148955 -1.08723592838627 1.02517804413913 -0.412680186313667…  
10 0.6318328148434 -1.08741784644221 -0.550575696474202 0.971967251067794…  
# … with more rows

collect()也有类似的限制，因为它只能收集适合驱动节点内存的数据集；但是，如果必须通过驱动节点从Spark中提取大型数据集，则可以使用分布式存储提供的专用工具。对于HDFS，你可以运行以下命令：

hadoop fs -copyToLocal largefile.txt largefile.txt

或者，你也可以通过一个collect()的回调任务来收集不适合内存的数据集。回调只是一个R函数，它会在每个Spark分区上调用。然后你可以将数据集写入磁盘或通过网络推送到其它群集。即使收集数据集的驱动节点内存小于3GB，也可以使用以下代码。也就是说，正如第3章所介绍的，你应该避免将大型数据集收集到一台计算机中，因为这会造成一个明显的性能瓶颈。为了简洁起见，我们只收集前100万行；如果时间允许，可以放心移除head(10^6)：

dir.create("large")  
spark\_read\_text(sc, "largefile.txt", memory = FALSE) %>% head(10^6) %>% collect(callback = function(df,   
 idx) {  
 writeLines(df$line, paste0("large/large-", idx, ".txt"))  
})

确保你清除了这些大型文件，也清空了你的回收站：

unlink("largefile.txt", recursive = TRUE)  
unlink("large", recursive = TRUE)

在大多数情况下，数据已经存储在集群中，因此不必担心复制大型数据集；相反，你可以专注于读取和写入不同的文件格式，我们下面介绍这些格式。

## 文件格式

Spark能够“开箱”就可以与多种文件格式交互，如CSV、JSON、LIBSVM、ORC和Parquet。表8-1将文件格式映射到Spark中用于读写数据的函数。

表8-1. 用于读写文件格式的Spark函数

|  |  |  |
| --- | --- | --- |
| 格式 | 读取 | 写入 |
| CSV | spark\_read\_csv() | spark\_write\_csv() |
| JSON | spark\_read\_json() | spark\_write\_json() |
| LIBSVM | spark\_read\_libsvm() | spark\_write\_libsvm() |
| ORC | spark\_read\_orc() | spark\_write\_orc() |
| Apache Parquet | spark\_read\_parquet() | spark\_write\_parquet() |
| 文本 | spark\_read\_text() | spark\_write\_text() |

以下各节将从著名的CSV文件格式开始，介绍每种文件格式的特殊注意事项以及一些主流文件格式的优缺点。

### CSV

CSV格式可能是目前使用的最常见的文件类型。它是由一个使用给定的字符（通常是逗号）进行分隔的文本文件。读取CSV文件应该非常简单；但是，值得一提的是，有两种技术可以帮助你处理与优质格式CSV文件不完全兼容的CSV。Spark提供了以下解决解析问题的模式：

* 容忍缺失 为缺失部分插入NULL值
* 舍弃不良结构 舍弃结构不正确的行
* 快速报错 如果遇到结构不正确的行，马上放弃

你可以在sparklyr的options参数中传值，使用它们。下面的示例创建了一个条目破坏的文件，展示如何将其读入Spark：

## Creates bad test file  
writeLines(c("bad", 1, 2, 3, "broken"), "bad.csv")  
spark\_read\_csv(  
sc,  
"bad3",  
"bad.csv",  
columns = list(foo = "integer"),  
options = list(mode = "DROPMALFORMED"))  
# Source: spark<bad3> [?? x 1]  
 foo  
 <int>  
1 1  
2 2  
3 3

Spark提供了一个问题跟踪列，该列在默认情况下是隐藏的。要启用它，使用add \_corrupt\_record给columns列表。你可以把这个方法与PERMISSIVE模式相结合。所有行都会导入，无效条目将返回NA，并在\_corrupt\_record中跟踪此问题：

spark\_read\_csv(  
sc,  
"bad2",  
"bad.csv",  
columns = list(foo = "integer", "\_corrupt\_record" = "character"),  
options = list(mode = "PERMISSIVE") )  
# Source: spark<bad2> [?? x 2]  
 foo `\_corrupt\_record`  
 <int> <chr>  
1 1 NA  
2 2 NA  
3 3 NA  
4 NA broken

在大多数系统中，以csv的形式读取和存储数据是非常常见，而且是广泛支持的。对于表格式数据集，它仍然是一个流行的选项，但是对于包含嵌套结构和非表格式数据的数据集，JSON通常是首选的。

### JSON

JSON是一种源自JavaScript的文件格式。由于其灵活性和广泛支持，它已经变得独立于语言并且非常流行。读取和写入JSON文件非常简单：

writeLines("{'a':1, 'b': {'f1': 2, 'f3': 3}}", "data.json")  
simple\_json <- spark\_read\_json(sc, "data.json")  
simple\_json  
# Source: spark<data> [?? x 2]  
 a b  
 <dbl> <list>  
1 1 <list [2]>

但是，当你处理包含嵌套字段的数据集（如本例中的数据集）时，值得一提的是如何提取嵌套字段。一种方法是使用JSON路径，这是一种领域相关的语法，通常用于提取和查询JSON文件。你可以使用get\_json\_object()和 to\_json()的组合来指定感兴趣的json路径。要提取f1，你可以运行以下转换：

simple\_json %>% dplyr::transmute(z = get\_json\_object(to\_json(b), '$.f1'))  
# Source: spark<?> [?? x 3]  
 a b z  
 <dbl> <list> <chr>  
1 1 <list [2]> 2

另一个方法是使用install.packages("sparklyr.nested")从CRAN安装sparkly.nested，然后使用sdf\_unnest()p拍平嵌套的数据：

sparklyr.nested::sdf\_unnest(simple\_json, "b")  
# Source: spark<?> [?? x 3]  
 a f1 f3  
 <dbl> <dbl> <dbl>  
1 1 2 3

虽然JSON和csv使用起来非常简单且通用，但它们并没有针对性能进行优化；而其它格式（如ORC、AVRO和Parquet）则进行了优化。

### Parquet

Apache Parquet、Apache ORC和Apache AVRO都是为了性能而设计的文件格式。Parquet和ORC以列格式存储数据，而AVRO是基于行的。它们都是二进制文件格式，这会减少存储空间并提高性能。这是以外部系统和库更难读取它们为代价的；然而，当在Spark中用作中间数据存储时，这通常不是问题。

为了说明这一点，图8-4描绘了使用bench程序包运行100万行写速度基准的结果；在决定哪种格式最适合你的需要时，可以在有意义的数据集上使用自己的基准：

numeric <- copy\_to(sc, data.frame(nums = runif(10^6)))  
bench::mark(CSV = spark\_write\_csv(numeric, "data.csv", mode = "overwrite"), JSON = spark\_write\_json(numeric,   
 "data.json", mode = "overwrite"), Parquet = spark\_write\_parquet(numeric, "data.parquet",   
 mode = "overwrite"), ORC = spark\_write\_parquet(numeric, "data.orc", mode = "overwrite"),   
 iterations = 20) %>% ggplot2::autoplot()

从现在开始，确保你从Spark断开了连接：

spark\_disconnect(sc)

以上就是一些开箱即用的广泛支持的文件格式介绍。接下来，我们将介绍如何处理需要外部包和定制的格式。

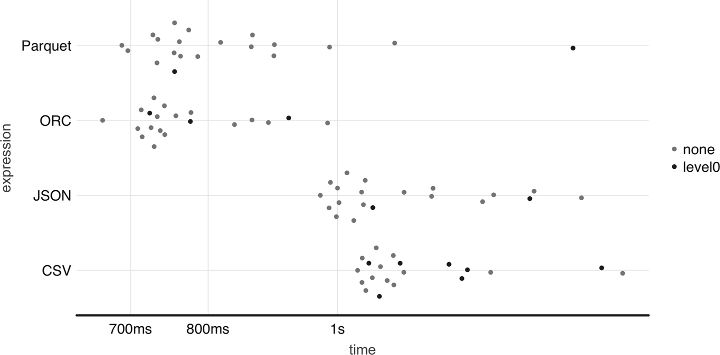


图8-4. 使用CSV，JSON，Parquet和ORC写入100万行的基准表现

### 其它

Spark是一个非常灵活的计算平台。它可以通过使用扩展程序包来添加功能。你可以使用适当的包访问新的源类型或文件系统。

连接时需要将程序包加载到Spark中。要加载包，Spark需要它的位置，它可能在集群内、共享文件中或internet中。

在sparklyr中，程序包位置传递给spark\_connect()。所有包都应列在连接配置的sparklyr.connect.packages项中。

可以访问以前未列出的数据源类型。加载Spark的默认包是两个步骤中的第一步，第二步是实际读取或写入数据。spark\_read\_source()和spark\_write\_source()函数会执行此操作。它们是通用函数，可以使用默认包导入的库。

例如，我们可以按如下方式读取XML文件：

sc <- spark\_connect(master = "local", version = "2.3", config = list(  
sparklyr.connect.packages = "com.databricks:spark-xml\_2.11:0.5.0"))  
writeLines("<ROWS><ROW><text>Hello World</text></ROW>", "simple.xml")  
spark\_read\_source(sc, "simple\_xml", "simple.xml", "xml")  
# Source: spark<data> [?? x 1]  
 text  
 <chr>  
1 Hello World

你也可以轻松的写入XML，代码如下：

tbl(sc, "simple\_xml") %>% spark\_write\_source("xml", options = list(path = "data.xml"))

此外，R社区还开发了一些扩展来加载其它文件格式，例如sparklyr.nested用于帮助嵌套数据，spark.sas7bdat用于从SAS读取数据，sparkavro用于以AVRO格式读取数据，sparkwarc用于读取WARC文件，这些扩展使用了第10章中介绍的扩展机制。第11章介绍了使用R程序包加载其它文件格式的技术，第13章介绍了使用Java库做了进一步补充。但首先，让我们探讨如何从几个不同的文件系统中检索和存储文件。

## 文件系统

Spark默认为当前运行它的文件系统。在YARN管理的集群中，默认的文件系统是HDFS。*/home/user/file.csv*的示例路径将从集群的HDFS文件夹（而不是Linux文件夹）读取。操作系统的文件系统将用于其它部署，如独立部署和sparklyr的本地部署。

文件系统协议可以在读取或写入时更改。你可以通过sparklyr函数的path参数来实现这一点。例如，[*file://home/user/file.csv*](file://home/user/file.csv) 的完整路径强制使用本地操作系统的文件系统。

还有许多其它文件系统协议，例如Databricks的文件系统的*\_dbfs://\_*、Amazon的S3服务的*\_s3a://\_*、Microsoft Azure存储的*\_wasb://\_*和Google存储的*\_gs://\_*。

Spark并没有直接为所有这些组件提供支持；相反，它们是根据需要配置的。例如，访问“s3a”协议需要向sparklyr.connect.packages配置添加一个包，而连接和指定适当的请求凭据可能需要使用AWS\_ACCESS\_KEY\_ID和 AWS\_SECRET\_ACCESS\_KEY环境变量。

Sys.setenv(AWS\_ACCESS\_KEY\_ID = my\_key\_id)  
Sys.setenv(AWS\_SECRET\_ACCESS\_KEY = my\_secret\_key)  
sc <- spark\_connect(master = "local", version = "2.3", config = list(sparklyr.connect.packages = "org.apache.hadoop:hadoop-aws:2.7.7"))  
my\_file <- spark\_read\_csv(sc, "my-file", path = "s3a://my-bucket/my-file.csv")

访问其它文件协议需要加载不同的包，但在某些情况下，提供Spark环境的供应商可能会为你加载包。请参考供应商的文件，以了解是否存在这种情况。

## 存储系统

数据池和Spark通常是同时存在的，可以选择访问数据库和数据仓库等存储系统。用适当的示例演示所有不同的存储系统非常耗时，因此我们将介绍一些常用的存储系统。

首先，Apache *Hive*是一个数据仓库软件，它使用SQL方便地读取、写入和管理驻留在分布式存储中的大型数据集。事实上，Spark将Hive的组件直接嵌入到它的源中。同时安装Spark和Hive是很常见的，因此我们首先介绍Hive，然后介绍Cassandra，再介绍JDBC连接。

### Hive

在YARN管理的集群中，Spark提供了与Apache Hive的更深入的集成。

打开Spark连接后，可以方便地访问Hive表。通过在SQL语句中引用表，使用DBI访问该表的数据：

sc <- spark\_connect(master = "local", version = "2.3")  
spark\_read\_csv(sc, "test", "data-csv/", memory = FALSE)  
DBI::dbGetQuery(sc, "SELECT \* FROM test limit 10")

另一个应用表的方法是使用dplyr和tbl()函数，它们会得到一个表的引用：

dplyr::tbl(sc, "test")

必须重申没有数据导入到R中tbl()函数只创建一个引用。然后，你可以在tbl()命令之后添加更多dplyr谓词作为管道操作：

dplyr::tbl(sc, "test") %>% dplyr::group\_by(y) %>% dplyr::summarise(totals = sum(y))

Hive表引用会假设一个默认数据库源。通常，所需的表位于元存储中的不同数据库中。如果要使用SQL访问它，请在表前面加上数据库名称当做前缀。并使用句点分隔，如下所示：

DBI::dbSendQuery(sc, "SELECT \* FROM databasename.table")

在dplyr中，可以使用in\_schema()函数。这个函数用tbl()调用中：

tbl(sc, dbplyr::in\_schema("databasename", "table"))

你还可以使用tbl\_change\_db()函数设置当前进程的默认数据库。通过DBI或dplyr的任何后续调用都将使用选定的名称作为默认数据库：

tbl\_change\_db(sc, "databasename")

以下示例需要额外的Spark包和数据库，除非你恰好可以访问JDBC驱动程序或Cassandra数据库，否则很难推进。

接下来，我们将探讨一个不太结构化的存储系统，通常称为NoSQL数据库。

### Cassandra

Apache *Cassandra*是一个免费的、开源的、分布式的、宽列存储的NoSQL数据库管理系统，旨在跨许多商业服务器处理大量数据。虽然除了Cassandra之外还有许多其它数据库系统，但快速查看Spark中Cassandra的使用方法，可以让你深入了解如何使用其它数据库和存储系统，如Solr、Redshift、Delta Lake等。

下面的示例代码演示如何使用datastax:spark-cassandra-connector包从Cassandra读取数据。关键是使用org.apache.spark.sql.cassandra库作为源参数。它提供了Spark可用的理解数据源的映射。除非有Cassandra数据库，否则跳过执行以下语句：

sc <- spark\_connect(master = "local", version = "2.3", config = list(sparklyr.connect.packages = "datastax:spark-cassandra-connector:2.3.1-s\_2.11"))  
spark\_read\_source(sc, name = "emp", source = "org.apache.spark.sql.cassandra", options = list(keyspace = "dev",   
 table = "emp"), memory = FALSE)

Spark在处理外部数据库和数据仓库时最有用的特性之一是Spark可以将计算向下推到数据库中，这一特性称为*下推谓词（pushdown predicates）*。简而言之，下推谓词通过询问远程数据库一些智能问题来提高性能。当你对通过spark\_read\_source()引用但未加载到内存中的远程表执行包含filter(age > 20)表达式的查询时，不会将整个表放入spark中，而该查询会被传递到远程数据库，并且只获取远程表的一个子集。

虽然寻找支持远程存储系统的Spark包是理想的选择，但有时包不可用，你需要考虑供应商JDBC驱动程序。

### JDBC

当Spark包不能提供连接时，可以考虑使用JDBC连接。JDBC是编程语言Java的接口，它定义了客户机访问数据库的方式。

用spark\_read\_jdbc()和spark\_write\_jdbc()连接到远程数据库非常容易；你只需要能够访问适当的JDBC驱动程序，这有时是不值一提的，而有时则存在风险。为了简单起见，我们可以简单考虑一下如何实现到远程MySQL数据库的连接。

首先，你需要从MySQL的开发人员网站下载合适的JDBC驱动程序，并这个额外的驱动程序指定为sparklyr.shell.driver-class-path连接选项。由于JDBC驱动程序是基于Java的，所以代码包含在JAR（Java存档）文你中。一旦你使用合适的驱动程序连接到Spark，就可以使用*jdbc://*协议访问特定的驱动程序和数据库。除非你愿意自己下载和配置MySQL，否则请跳过执行以下语句：

sc <- spark\_connect(master = "local", version = "2.3", config = list(`sparklyr.shell.driver-class-path` = "~/Downloads/mysql-connector-java-5.1.41/mysql-connector-java-5.1.41-bin.jar"))  
spark\_read\_jdbc(sc, "person\_jdbc", options = list(url = "jdbc:mysql://localhost:3306/sparklyr",   
 user = "root", password = "<password>", dbtable = "person"))

如果你是某个数据库供应商的客户，使用供应商提供的资源通常是选择合适驱动的最好起点。

## 小结

本章继续介绍了如何使用Spark来连接和处理各种数据源及其原因。这些连接使用一种新的数据存储模型，称为数据湖的存储模式，它提供了比标准ETL过程更大的灵活性，可以让你能够使用原始数据集更多的信息来丰富数据分析和建模。

我们还介绍了在Spark中读取、写入和复制数据的最佳实践。然后我们又回来研究数据池的组成部分：文件格式和文件系统，前者表示数据的存储方式，后者表示数据的存储位置。然后，你学习了如何处理需要额外Spark包的文件格式和存储系统，以及一些跨文件格式的性能权衡问题，并学习了在Spark中使用存储系统（数据库和仓库）所需的概念。

虽然读写数据集应该并不稀奇，但在读写大型数据集时，仍然会遇到资源限制。为了处理这些情况，第9章会向你展示Spark如何跨多台机器管理任务和数据，从而进一步提高分析和建模任务的性能。

# 第9章 调试

混乱不是深渊，而是阶梯。 —Petyr Baelish

在前面的章节中，我们假设Spark中的计算是有效的。虽然这个假设在一些情况下是成立的，但通常需要对Spark内部运行的操作有一些了解，以便微调配置，从而使计算高效运行。本章介绍Spark如何计算大型数据集上的数据，并提供有关如何优化操作的详细信息。

例如，在本章中，你将学习如何请求更多的计算节点以及增加内存，如果你还记得第2章的内容，则在本地实例中默认为只有2GB。你将了解Spark如何通过分区、重洗和缓存来统一计算。就像前几章提到的，这是介绍Spark内部机制的最后一章；在完成本章之后，我们相信你会具备使用Spark所必需的中级技能。

在第10-12章中，我们会探索一些令人兴奋的技术，可以处理具体建模、可扩展性和计算问题。首先，我们必须了解Spark如何执行内部计算，我们可以控制哪些部分，以及为什么。

## 概述

Spark通过在多台计算机上配置、分区、执行、重洗、缓存和序列化数据、任务和资源来执行分布式计算：

* 配置需要集群管理器提供资源：全部机器，内存等。
* 分区在不同的机器之间分割数据。分区可以是隐式的，也可以是显式的。
* 执行意味着在每个分区上运行任意转换。
* 重洗将数据重新分发到正确的机器。
* 缓存会跨不同的计算周期将数据保存在内存中。
* 序列化转换要通过网络发送给其它工作节点或返回到驱动节点的数据。

要说明每一个概念，让我们创建一个无序整数的三个分区，并使用arrange()进行排序：

data <- copy\_to(sc, data.frame(id = c(4, 9, 1, 8, 2, 3, 5, 7, 6)), repartition = 3)  
data %>% arrange(id) %>% collect()

图9-1显示了这个排序作业在理论上是如何跨一个机器集群工作的。首先，Spark会将集群配置为使用三台工作节点。在本例中，数字1到9被划分为三个存储实例。因为数据已经分区，所以每个工作节点都加载这个隐式分区；例如，4、9和1加载在第一个工作节点中。然后，将一个任务分发给每个工作节点，对每个工作节点中的每个数据分区应用转换；此任务用f(x)表示。在本例中，f(x)在分区内执行排序操作。由于Spark是通用的，所以分区上的执行可以根据需要变得简单或复杂。

然后将结果打乱传入正确的机器，以完成整个数据集的排序操作，从而完成一个阶段。*阶段*是一组操作，Spark可以执行这些操作，而无需在机器之间对数据重洗。在集群中对数据进行排序之后，可以选择将排序后的结果*缓存*在内存中，以避免多次重新运行这个计算任务。

最后，通过连接集群计算机的网络将一小部分结果*序列化*，并将结果返回到驱动节点以打印此排序示例的预览。

注意，虽然图9-1描述了排序操作，但类似的方法也适用于筛选或连接数据集以及大规模数据的分析和建模。Spark提供了执行自定义分区、自定义重洗等的支持，但这些较低级别的操作大多没有在sparklyr中开放；相反，sparklyr通过数据分析工具,如dplyr或DBI,建模和使用许多扩展提供的更高级别的命令使这些操作可用。对于那些可能需要实现低级操作的少数情况，你始终可以通过sparklyr扩展使用Spark的Scala API或运行自定义的分布式R代码。

为了有效地调试Spark，我们首先熟悉Spark的计算图和Spark的事件时间线。两者都可以通过Spark的web界面访问。

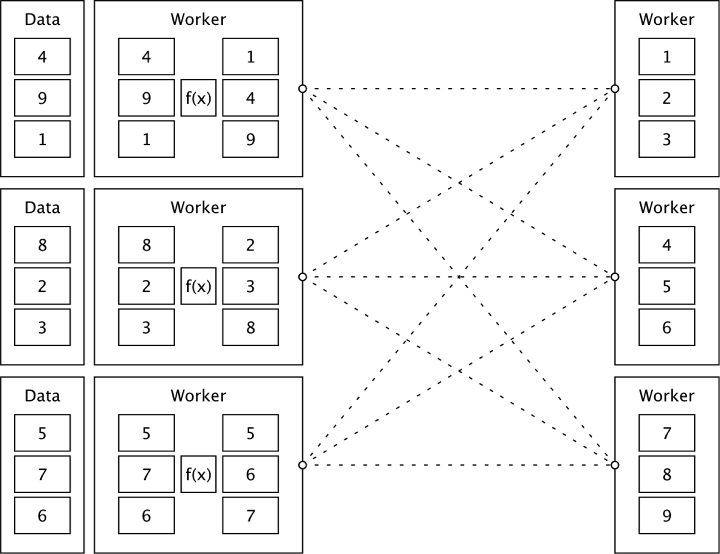


图9-1. 使用Apache Spark对分布式数据排序

### 计算图

Spark使用有向无环图（DAG）描述所有计算步骤，这意味着Spark中的所有计算都向前移动而不重复前面的步骤，这有助于Spark有效地优化计算。

在我们的示例中，理解给定操作，例如示例中的排序任务的Spark计算图的最佳方法是打开Spark的web接口中SQL选项卡上最后完成的查询。图9-2显示了排序操作的最终计算图，其中包含以下操作：

* WholeStageCodegen 此模块确定，它包含的操作用于生成高效转换至字节码的计算机代码。将操作转换为字节码通常会带来很小的成本，但这是值得付出的代价，因为在Spark中可以更快地执行操作。通常，你以忽略这个模块并将焦点放在它包含的操作上。
* InMemoryTableScan 这个意思是，原始数据集存储在内存中并一次性按行转换。
* Exchange 分区是交换过的，也就是说，是被集群中的执行器执行过重洗操作。
* Sort 当记录到达合适的执行器，它们在这个最终阶段完成排序。

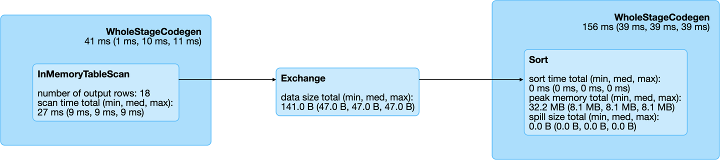


图9-2. 排序查询的Spark计算图

从查询的详细信息中，你可以打开最后一个Spark作业，到达作业详细信息页面。你可以使用“DAG Visualization”来展开该页面，以创建类似于图9-3的图形。此图显示了作业中的一些额外细节和阶段。注意，没有箭头指向之前的步骤，因为Spark使用了非循环图。

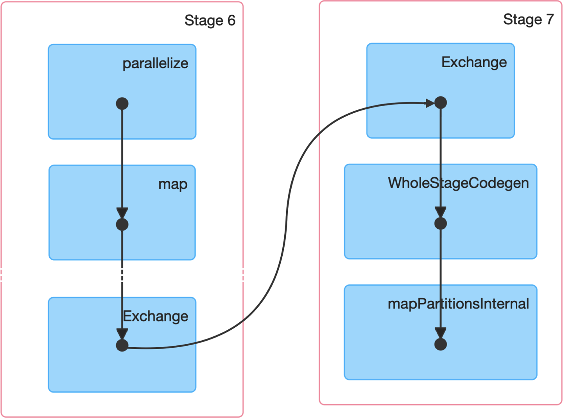


图9-3. 排序作业的Spark计算图

下面，我们会继续深入介绍Spark阶段并研究它的事件时间线。

### 时间线

*事件时间线*是有关Spark跨每个阶段的计算周期开销的一个良好汇总。理想情况下，你希望看到这个时间线主要由CPU使用量组成，因为其它任务可以被视为开销。你还希望看到Spark使用所有可用集群节点上的所有cpu。

选择当前作业中的第一个阶段并展开事件时间线，它应该类似于图9-4。注意，我们显式地请求了三个分区，它们在这个可视化中由三个道表示。

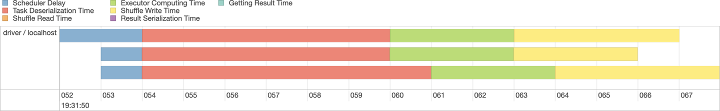


图9-4. Spark事件时间线

由于我们的机器配备了四个CPU，我们可以使用sdf\_repartition()显式地重新分区数据来进一步实现并行化计算，结果如图9-5所示：

data %>% sdf\_repartition(4) %>% arrange(id) %>% collect()

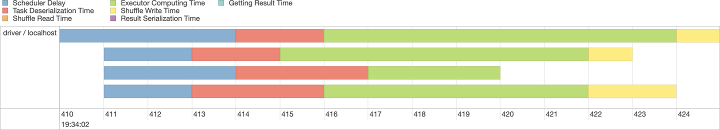


图9-5. 带有额外分区的事件时间线

图9-5展示了四个执行网关，其中在执行器计算时间上花费的时间最多，这表明这个特定的操作正在更好地利用我们的计算资源。当使用集群时，请求集群上更多计算节点应该会缩短计算时间。与此相反，对于那些展示花了大量时间进行重洗的时间线，请求更多的计算节点可能不会缩短时间，实际上还可能会使所有操作都变慢。我们并没有一套具体的规则用来优化一个阶段；但是，随着在多个操作中获得时间线的经验，你会深入了解如何正确优化Spark操作。

## 配置

在调试Spark应用程序时，最常见的配置资源是内存和内核，特别是：

* 驱动节点的内存 驱动节点所需的内存量
* 每个工作节点的内存 工作节点所需的内存量
* 每个工作节点的内核数 工作节点所需的CPU数
* 工作节点数 此会进程所需的工作节点数

建议为驱动节点请求远多于每个工作节点上可用的内存的内存量。在大多数情况下，你需要为每个工作节点请求一个内核。

本地模式中没有工作节点，但是我们依然可以使用下列代码配置内存和内核：

# Initialize configuration with defaults  
config <- spark\_config()  
# Memory  
config["sparklyr.shell.driver-memory"] <- "2g"  
# Cores  
config["sparklyr.connect.cores.local"] <- 2  
# Connect to local cluster with custom configuration  
sc <- spark\_connect(master = "local", config = config)

使用Spark单机和Mesos群集管理器时，默认情况下会分配所有可用内存和内核；因此，不需要更改其它配置，除非你希望限制资源以便允许多个用户共享群集。在这种情况下，你可以使用total-executor-cores来限制请求的执行器总数。Spark单机和Mesos管理下的Spark指南提供了有关共享集群的其它信息。

在YARN客户端下运行时，你可以按照以下方式配置内存和内核：

# Memory in driver  
config["sparklyr.shell.driver-memory"] <- "2g"  
# Memory per worker  
config["sparklyr.shell.executor-memory"] <- "2g"  
# Cores per worker  
config["sparklyr.shell.executor-cores"] <- 1  
# Number of workers  
config["sparklyr.shell.num-executors"] <- 3

在集群模式下使用YARN时，可以使用sparklyr.shell.driver-cores配置在驱动节点中请求的总内核数。Spark on YARN guide提供了额外的配置设置，可以帮助你。

有几种类型的配置：

* *连接* 这些设置用作spark\_connect()的参数。它们是连接时使用的常见设置。
* *提交* 这些设置是在sparklyr通过spark-submit提交到Spark时使用的；有些设置取决于所使用的集群管理器。
* *运行时* 这些设置会在创建Spark进程时配置Spark。它们独立于集群管理器，并且关联具体的Spark。
* *sparklyr* 使用这些来配置sparklyr行为。这些设置独立于集群管理器，并且专属于R。

下面的章节会提供所有可用设置的详细列表。在调试Spark时，并不要求你完全理解它们，但是浏览一遍对以后的故障排除有帮助。你可以跳过这些章节，根据需要把它们用作参考资料。

### 连接设置

你可以将表9-1中列出的参数与spark\_connect()一起使用。它们会配置高级设置，定义连接方法、Spark的安装路径和要使用的Spark版本。

表9-1. 连接Spark所用的参数

|  |  |
| --- | --- |
| 名称 | 取值 |
| master | 要连接的Spark集群URL。使用"local"连接至一个使用spark\_install()a安装的本地Spark实例。 |
| SPARK\_HOME | Spark的安装路径。默认为SPARK\_HOME环境变量提供的路径。如果已经定义SPARK\_HOME，这个参数会一直使用，除非给定了版本参数强制使用本地安装的版本。 |
| method | 用来连接Spark的方法。默认的连接方法是"shell"，使用spark-submit进行连接。使用"livy"通过HTTP执行远程连接，使用databricks集群时使用"databricks"，使用qubole集群时使用"qubole"。 |
| app\_name | 运行Spark集群时使用的应用名称。 |
| version | 使用的Spark版本。这个参数只对"local"和"livy"连接有效。 |
| config | 定制配置属性，生成Spark连接。详见spark\_config。 |

可以通过在config参数中指定列表来配置其它设置。现在让我们看看这些设置有什么。

### 提交设置

运行spark-submit（启动spark的终端应用程序）时必须指定某些设置。例如，由于spark-submit启动一个作为Java实例运行的驱动节点，需要将分配的内存指定为spark-submit的参数。

你可以通过运行以下命令列出所有可用的spark-submit参数：

spark\_home\_dir() %>% file.path("bin", "spark-submit") %>% system2()

为了简单易读，我们在表9-2中提供了这个命令的输出，用适当的spark\_config()设置替换spark-submit参数，并删除了不适用或本章中已经介绍的参数。

表9-2. 配置spark-submit的可用设置

|  |  |
| --- | --- |
| 名称 | 取值 |
| sparklyr.shell.jars | 作为spark\_connect()中的jars参数给定。 |
| sparklyr.shell.packages | 包含在驱动节点和执行节点类路径上的JAR的Maven坐标逗号分隔列表。将搜索本地Maven repo，然后搜索Maven Central和sparklyr.shell.repositories提供的任何其它远程仓库。坐标的格式应为groupId:artifactId:version。 |
| sparklyr.shell.exclude-packages | groupId:artifactId的逗号分割列表，排除解决sparklyr.shell.packages提供的依赖时使用，以避免依赖冲突。 |
| sparklyr.shell.repositories | 其它远程仓库的逗号分割列表，以便使用sparklyr.shell.packages搜索Maven坐标。 |
| sparklyr.shell.files | 要放在每一个执行器工作目录中文件的逗号分割列表。执行器中这些文件的路径可以通过SparkFiles.get(fileName)获取。 |
| sparklyr.shell.conf | 任意Spark配置属性，设置为PROP=VALUE。 |
| sparklyr.shell.properties-file | 要加载额外属性的文件路径。如果没有给定，会搜索*conf/spark-defaults.conf*。 |
| sparklyr.shell.driver-java-options | 传递给驱动节点的额外JAVA选项。 |
| sparklyr.shell.driver-library-path | 传递给驱动节点的额外库路径。 |
| sparklyr.shell.driver-class-path | 传递给驱动节点的额外类路径。注意，使用sparklyr.shell.jars添加的JAR会自动加到这个类路径。 |
| sparklyr.shell.proxy-user | 提交应用程序时要模拟的用户。这个参数不能和sparklyr.shell.principal/sparklyr.shell.key tab一起使用。 |
| sparklyr.shell.verbose | 打印额外的调试输出。 |

其它设置，如表9-3所示，是和YARN相关的。

表9-3. 使用YARN时配置spark-submit的可用设置

|  |  |
| --- | --- |
| 名称 | 取值 |
| sparklyr.shell.queue | 要提交的YARN队列（默认：“default”）。 |
| sparklyr.shell.archives | 抽取至每个执行器工作目录的备档的逗号分割列表。 |
| sparklyr.shell.principal | 在secure HDFS上运行时，用来登录到KDC的principal。 |
| sparklyr.shell.keytab | 包含刚才指定的principal的keytab的文件的完整路径。此keytab将通过安全分布式缓存复制到运行应用管理的节点，以便定期更新登录凭证和委派令牌。 |

通常，任何spark-submit设置都通过sparklyr.shell.X来配置，其中X是spark-submit参数不带--前缀的名称。

### runtime设置

正如提到的，一些Spark设置会配置进程的运行时。运行时设置是提交设置的超集，因为即使不能更改设置，检索当前配置通常也是有帮助的。

要列出当前Spark会话中的Spark设置，可以运行以下命令：

spark\_session\_config(sc)

表9-4 描述了运行时设置。

表9-4. 配置Spark进程时的可用设置

|  |  |
| --- | --- |
| 名称 | 取值 |
| spark.master | local[4] |
| spark.sql.shuffle.partitions | 4 |
| spark.driver.port | 62314 |
| spark.submit.deployMode | client |
| spark.executor.id | driver |
| spark.jars | */Library/…/sparklyr/java/sparklyr-2.3-2.11.jar* |
| spark.app.id | local-1545518234395 |
| spark.env.SPARK\_LOCAL\_IP | 127.0.0.1 |
| spark.sql.catalogImplementation | hive |
| spark.spark.port.maxRetries | 128 |
| spark.app.name | sparklyr |
| spark.home | */Users/…/spark/spark-2.3.2-bin-hadoop2.7* |
| spark.driver.host | localhost |

但是，Spark中还有更多的配置，如Spark配置指南中所述。这超出了本书的范围，所以，如果可能的话，可以花点时间来确定那些可能对你的特定用例有关的配置。

### sparklyr设置

除了Spark设置，还有一些专门的sparklyr设置。通常在调试Spark时不使用这些设置；而在从R排除Spark故障时很有用。例如，可以使用sparklyr.log.console = TRUE将Spark日志输出到R控制台；这在排除故障时是可取的，但在其它情况下有点烦人。下面介绍了如何列出设置（结果如表9-5所示）：

spark\_config\_settings()

表9-5. 配置sparklyr程序包的可用设置

|  |  |
| --- | --- |
| 名称 | 描述 |
| sparklyr.apply.packages | 配置spark\_apply()程序包参数的默认值。 |
| sparklyr.apply.rlang | 实验功能。启用可以改善spark\_apply()的序列化。 |
| sparklyr.apply.serializer | 配置spark\_apply()使用的版本，以便序列化闭包。 |
| sparklyr.apply.schema.infer | 当spark\_apply()给定列类型，用于推断模式的行数。 |
| sparklyr.arrow | 是否使用Apache Arrow序列化数据？ |
| sparklyr.backend.interval | sparklyr确认后端操作的总秒数。 |
| sparklyr.backend.timeout | sparklyr不再等待后端操作完成之前的总秒数。 |
| sparklyr.collect.batch | 使用批收集是要收集的总行数，默认为100,000。 |
| sparklyr.cancellable | 当R进程被打断时，是否要取消Spark作业？ |
| sparklyr.connect.aftersubmit | spark-submit执行后要调用的R函数。 |
| sparklyr.connect.app.jar | spark\_connect()中的sparklyr JAR路径。 |
| sparklyr.connect.cores.local | spark\_connect(master = "local")使用的内核数，默认Wi为parallel::detectCores()。 |
| sparklyr.connect.csv.embedded | 匹配Spark版本号的正则表达，需要程序包扩展支持CSV。 |
| sparklyr.connect.csv.scala11 | 当Spark 1.6.X使用嵌入的CSV JARS时，使用Scala 2.11 JAR]。 |
| sparklyr.connect.jars | 在给Spark提交应用时，包含的其它的JAR。 |
| sparklyr.connect.master | 作为spark\_connect()的master参数的集群主节点。注意通常选择spark.master的设置。 |
| sparklyr.connect.packages | 连接Spark时包含的Spark程序包。 |
| sparklyr.connect.ondisconnect | 完成spark\_disconnect()后调用的R函数。 |
| sparklyr.connect.sparksubmit | 连接时除使用spark-submit外的命令。 |
| sparklyr.connect.timeout | 初始化时，放弃连接sparklyr网关前的总秒数。 |
| sparklyr.dplyr.period.splits | dplyr是否应该把列名分割成数据库和表？ |
| sparklyr.extensions.catalog | 存放扩展JAR的目录PATH。默认为TRUE；可以改成FALSE。 |
| sparklyr.gateway.address | 驱动机器的地址。 |
| sparklyr.gateway.config.retries | 获取配置文件中端口和地址的重试次数。使用函数查询Kubernetes中的端口或地址时很有用。. |
| sparklyr.gateway.interval | sparklyr检查网关连接的总秒数。 |
| sparklyr.gateway.port | 驱动机器上sparklyr网关使用的端口。 |
| sparklyr.gateway.remote | sparklyr网关是否允许远程连接？例如，YARN集群中需要这个参数。 |
| sparklyr.gateway.routing | sparklyr网关服务是否要发送到其它进程？可以在Kubernetes中关闭。 |
| sparklyr.gateway.service | 当最后的连接断开后，sparklyr网关是否继续作为服务运行而不中断？ |
| sparklyr.gateway.timeout | 初始化后，放弃连接sparklyr网关前的总秒数。 |
| sparklyr.gateway.wait | 重试连接sparklyr网关前的等待时间。 |
| sparklyr.livy.auth | Livy连接的认证方式。 |
| sparklyr.livy.headers | Livy连接的其它HTTP文件头。 |
| sparklyr.livy.sources | 连接的时候，sparklyr源文件是否需要sourced？如果不需要，则手动注册sparklyr JAR。 |
| sparklyr.log.invoke | 每一次调用invoke()是否要打印到控制台？可以设置为callstack来给调用栈记录日志。 |
| sparklyr.log.console | 驱动节点的日志是否要打印到控制台？ |
| sparklyr.progress | 作业进度是否要返回给RStudio？ |
| sparklyr.progress.interval | 尝试获取Spark作业进度前的总等待秒数。 |
| sparklyr.sanitize.column.names | 部分不支持的列名是否需要清除？ |
| sparklyr.stream.collect.timeout | 停止收集sdf\_collect\_stream()中流式样本的总秒数。 |
| sparklyr.stream.validate.timeout | 确认作业流在创建过程中是否出错之前的的总秒数。 |
| sparklyr.verbose | 所有的sparklyr操作是否都使用详细日志记录？ |
| sparklyr.verbose.na | 是否在处理NA的时候使用详细日志记录？ |
| sparklyr.verbose.sanitize | 是否在清洗列和其它对象的时候使用详细日志记录？ |
| sparklyr.web.spark | Spark网络接口的URL。 |
| sparklyr.web.yarn | YARN网络接口的URL。 |
| sparklyr.worker.gateway.address | 工作机器的地址，绝大多数情况下是localhost。 |
| sparklyr.worker.gateway.port | 驱动机器上sparklyr网关使用的端口。 |
| sparklyr.yarn.cluster.accepted.timeout | 放弃等待YARN机器模式下集群资源的总秒数。 |
| sparklyr.yarn.cluster.hostaddress.timeout | 放弃等待YARN机器模式下集群分配主机地址的总秒数。 |
| sparklyr.yarn.cluster.lookup.byname | 在搜索提交的任务时，当前用户名是否要用来过滤YARN集群作业？ |
| sparklyr.yarn.cluster.lookup.prefix | 在搜索提交的任务时，用于过滤YARN集群作业的应用名称的前缀。 |
| sparklyr.yarn.cluster.lookup.username | 在搜索提交的任务时，用于过滤YARN集群作业的用户名。 |
| sparklyr.yarn.cluster.start.timeout | 放弃等待YARN集群y应用完成注册之前的总秒数。 |

## 分区

如第一章介绍的，MapReduce和Spark的设计目的是对存储在许多机器上的数据进行计算。每个计算实例上用于计算的数据子集称为分区。

默认情况下，Spark对每个现有隐式分区进行计算，因为对位置已定的数据进行计算更高效。但是，在某些情况下，你需要设置一个显式分区，以帮助Spark更有效地利用集群资源。

### 隐式分区

正如第8章介绍的，Spark可以读取以多种格式和不同存储系统存储的数据；但是，由于数据重洗是一项成本较高的操作，Spark会执行重用存储系统分区的任务。因此，这些分区对Spark是隐式的，因为它们已经很好地完成了定义，并且重新组织的话代价也很高。

Spark中由分布式存储系统定义的每一个计算都有一个隐式分区，甚至对于你可能不曾料到的创建分区，例如copy\_to()的操作也是如此。

你可以使用sdf\_num\_partitions()，来查看计算所需的分区个数：

sdf\_len(sc, 10) %>% sdf\_num\_partitions()  
[1] 2

虽然在大多数情况下，默认分区的效果已经很好，但在某些情况下，你还需要对所选分区进行显式的说明。

### 显式分区

有时候相比数据分区，你的计算实例过多或过少。在这两种情况下，重新分区数据以匹配集群资源会很有用。

各种数据函数，比如spark\_read\_csv()，已经支持repartition参数来正确地请求Spark重新分区数据。例如，我们可以创建一个由10个数字组成，分区为10的序列，如下所示：

sdf\_len(sc, 10, repartition = 10) %>% sdf\_num\_partitions()  
[1] 10

对于已经分区的数据集，我们也可以使用sdf\_repartition()：

sdf\_len(sc, 10, repartition = 10) %>%  
sdf\_repartition(4) %>%  
sdf\_num\_partitions()  
[1] 4

分区的数量通常会显著改变使用的速度和资源；例如，以下示例计算具有不同分区大小的超过1000万行的平均值：

library(microbenchmark)  
library(ggplot2)  
microbenchmark(`1 Partition(s)` = sdf\_len(sc, 10^7, repartition = 1) %>% summarise(mean(id)) %>%   
 collect(), `2 Partition(s)` = sdf\_len(sc, 10^7, repartition = 2) %>% summarise(mean(id)) %>%   
 collect(), times = 10) %>% autoplot() + theme\_light()

图9-6说明使用两个分区对数据进行排序的速度几乎是之前的两倍。这是因为程序可以使用两个CPU执行此操作。但是，不一定是更高的分区就产生更快的计算；相反，分区数据与计算集群，以及正在执行的数据分析操作有关。。

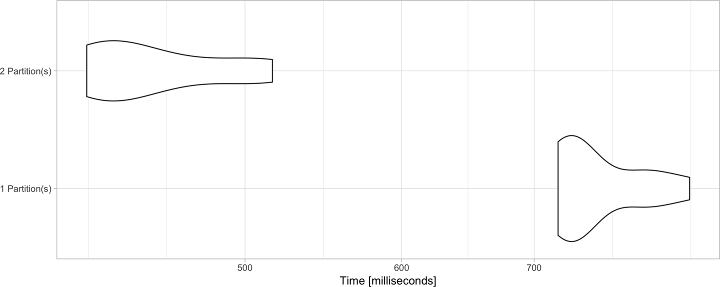


图9-6. 额外显式分区的计算速度

## 缓存

回忆一下第1章，Spark通过使用内存而不是磁盘来存储数据，从而比以前的产品更快。这就是所谓的Spark*弹性分布式数据集（resilient distributed dataset，RDD）*。RDD将相同数据的副本分布在多台机器上，这样，即使一台机器出现故障，其它机器还可以完成任务，因此，这就是术语“弹性”的含义。弹性在分布式系统中很重要，因为，虽然计算通常在一台机器上完成，但当数千台机器运行时，发生故障的可能性还是要高得多。当任务失败时，最好能够容错，以避免失去其它机器上的工作。RDD通过跟踪数据沿袭信息来实现这一点，以便在失败时自动重建丢失的数据。

在sparklyr中，可以使用tbl\_cache()和tbl\_uncache()控制何时从内存加载或卸载RDD。

大多数获取Spark数据框的sparklyr操作都会将结果缓存在内存中。例如，运行spark\_read\_parquet()或copy\_to()会提供已经缓存在内存中的spark数据框。作为Spark数据框，这个对象可用于大多数sparklyr函数，包括使用dplyr或机器学习进行数据分析：

library(sparklyr)  
sc <- spark\_connect(master = "local")  
iris\_tbl <- copy\_to(sc, iris, overwrite = TRUE)

使用spark\_web(sc)导航到Spark UI，单击“Storage”标签，然后单击具体的RDD，可以检查缓存了哪些表，如图9-7所示。

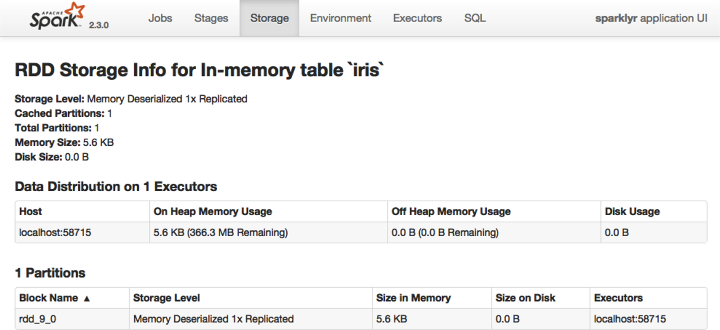


图9-7. Spark网络接口中缓存的RDD

当R进程通过重新启动或断开连接显式或隐式终止时，内存中加载的数据会释放出来；但是，要释放资源，你可以使用tbl\_uncache()：

tbl\_uncache(sc, "iris")

### 检查点

检查点是一种稍有不同的缓存类型；它在保存数据的同时，还将破坏图计算沿袭信息。例如，如果缓存的分区丢失，则可以从计算图中计算出该分区。但是由于计算源丢失，检查点不可能完成上述任务。

在创建高成本的计算图操作时，检查点可以用来保存和断开计算沿袭，以帮助Spark减少图计算资源；否则，Spark可能会尝试优化一个没有实际优化意义的计算图。

你可以通过保存CSV、Parquet和其它文件格式显式记录检查点。或者，让Spark在sparklyr中使用sdf\_checkpoint()进行检查点记录，如下所示：

# set checkpoint path  
spark\_set\_checkpoint\_dir(sc, getwd())  
# checkpoint the iris dataset  
iris\_tbl %>% sdf\_checkpoint()

注意，检查点会截断计算沿袭图，如果多次使用相同的中间结果，可以加快性能。

### 内存

Spark中的内存分为预留内存、用户内存、执行内存或存储内存：

* 预留内存 预留内存是Spark运行所需的内存，因此是必需且不应配置的开销。此值默认为300MB。
* 用户内存 用户内存是用于执行自定义代码的内存。sparklyr仅在执行dplyr表达式或建模数据集时间接使用此内存。
* 执行内存 执行内存用于Spark执行代码，主要用于处理分区的结果并执行重洗。
* 存储内存 例如，在sparklyr中使用compute()时，存储内存用于缓存RDD。

作为调试执行的一部分，你可以考虑通过创建具有不同于Spark默认值的Spark连接来调整分配给用户、执行和存储的内存量：

config <- spark\_config()  
# define memory available for storage and execution  
config$spark.memory.fraction <- 0.75  
# define memory available for storage  
config$spark.memory.storageFraction <- 0.5

例如，如果你希望使用Spark将大量数据存储在内存中，以便快速筛选和检索子集，可以让Spark只需很少的执行或用户内存。因此，为了最大化存储内存，可以按照以下方式调整Spark：

config <- spark\_config()  
# define memory available for storage and execution  
config$spark.memory.fraction <- 0.9  
# define memory available for storage  
config$spark.memory.storageFraction <- 0.9

但是，请注意，如果需要和可能的话，Spark会从存储内存中借给执行内存，反之亦然；因此，在实践中，很少需要调整内存设置。

## 重洗

重洗是一种跨机器重新分配数据的操作；它的成本很高，因此应该尽量减少这种操作。通过查看事件时间线，你可以很容易地确定是否在重洗上花费了大量的时间。通过重新组织数据分析问题或适时告诉Spark，可以减少重洗。

例如，当连接大小差异很大的多个数据框时，这就相关了；也就是说，一个集合比另一个集合小几个数量级。可以考虑使用sdf\_broadcast()将数据框标记为足够的小，以便在broadcast的连接中使用，这意味着它将一个较小的数据框推送到每个工作节点，以减少对较大数据框的重洗。以下是sdf\_broadcast()的一个示例：

sdf\_len(sc, 10000) %>% sdf\_broadcast() %>% left\_join(sdf\_len(sc, 100))

## 序列化

序列化是将数据和任务转换成一种可以在机器之间传输和在接收端重构的格式的过程。

在调试Spark时，调整序列化并不常见；但是，值得一提的是，其它序列化模块（如Kryo序列化器）可以提供基于默认Java序列化器的性能改进。你可以通过以下操作打开sparklyr中的Kryo序列化器：

config <- spark\_config()  
config$spark.serializer <- "org.apache.spark.serializer.KryoSerializer"  
sc <- spark\_connect(master = "local", config = config)

## 配置文件

在连接之前配置spark\_config()是优化Spark时最常用的方法。但是，在确定连接中的参数后，你应该考虑使用配置文件，因为它会消除连接代码中的混乱，并允许跨项目和同事共享配置信息。

例如，如果不使用如下方式连接Spark：

config <- spark\_config()  
config["sparklyr.shell.driver-memory"] <- "2G"  
sc <- spark\_connect(master = "local", config = config)

你可以使用所需的设置定义*config.yml*文件。此文件位于当前工作目录或父目录中。例如，可以创建以下*config.yml*文件来修改默认的驱动节点内存：

default:  
sparklyr.shell.driver-memory: 2G

然后，使用相同的配置进行连接要简洁的多：

sc <- spark\_connect(master = "local")

你也可以通过在spark\_config()中设置file参数来指定备用配置文件名或位置。使用配置文件的另一个好处是，系统管理员可以通过更改R\_CONFIG\_ACTIVE环境变量的值来更改默认配置。有关更多信息，请参阅GitHub的rstudio/config仓库。

## 小结

本章介绍了Spark内部机制和具体配置的诸多内容，以帮助你加速计算，并启用高计算负载。它可以为理解常见配置的瓶颈和指导提供基础。然而，调试Spark是一个广泛的话题，需要更多的章节来详尽介绍。因此，在对Spark的性能和可扩展性进行故障排除、搜索网络资源和咨询在线社区时，通常还需要对你的特定环境进行微调。

第10章会介绍R中提供的Spark扩展的生态系统。大多数扩展都是高度专业化的，但是它们对于特定情况以及有特殊需求的读者来说都是非常有用的。例如，它们可以处理嵌套数据，执行图分析，使用不同的建模库，如H20中的rsparkling。此外，接下来的几章会介绍许多高级数据分析和建模话题。这些话题是掌握R语言大规模计算所必需的。

# 第10章 扩展

我尽量多认识一些人。你永远不知道你需要哪个人。 —Tyrion Lannister

在第9章中，你学到了Spark如何通过允许用户配置集群资源、隐式或显式地分区数据、跨分布式计算节点执行命令、在需要时跨节点重洗数据、缓存数据以提高性能，以及通过网络高效地序列化数据来处理大规模数据。你还学习了如何配置连接、提交作业和运行应用程序时使用的不同Spark设置，以及仅适用于本章介绍的R和R扩展的特定设置。

第3，4章和第8章为阅读和理解大多数数据集提供了基础。然而，提供的功能限定在Spark的内置特性和表格数据集。本章突破表格数据，探讨如何通过图处理来分析和建模互联对象的网络，分析基因组学数据集，为深度学习准备数据，分析地理数据集，以及在大规模数据集上使用高级建模库（如H2O和XGBoost）。

前几章中介绍的所有内容应该能够满足你的大部分大规模计算需求。但是，对于那些仍然缺乏功能支持的少数用例，下面的章节提供了一些工具来扩展Spark，例如可以通过自定义R转换、自定义Scala代码或Spark中最新的执行模式，从而能够分析实时数据集。然而，在重新发明轮子之前，让我们来看看Spark中的一些扩展。

## 概述

在第1章中，我们把R社区呈现为一个充满活力的群体，其中的每个人在许多方面相互协作，例如，通过创建可以从CRAN安装的R程序包来推动开放科学的发展。以类似的，但规模要小得多的方式，R社区提供了扩展，扩大了Spark和R最初支持的功能。Spark本身也支持创建Spark扩展。事实上，许多R扩展都使用Spark的扩展。

人们经常创建扩展，所以本章节在某个时间点后就过时了。此外，我们甚至可能不知道许多Spark和R扩展。然而，我们至少可以通过查看CRAN中sparklyr的“反向导入”来跟踪CRAN中可用的扩展。在CRAN中发布的扩展和R程序包往往是最稳定的，因为当一个包在CRAN中发布时，它会经过一个评审过程，从而提高贡献的总体质量。

虽然我们希望能够展示所有的扩展，但是我们将本节限制在你最感兴趣的扩展。你可以在github.com/r-spark部分或通过搜索带有sparklyr标记的GitHub仓库找到其它扩展。

* rsparkling rsparkling扩展允许你使用R中的H2O和Spark。此扩展是我们将在Spark中考虑的高级建模方案。虽然Spark的内置建模库Spark MLlib在许多情况下都非常有用，但H2O的建模功能可以计算额外的统计指标，并且可以比Spark MLlib提供更多性能和可扩展性改进。我们既没有对MLlib和H2O进行详细的比较，也没有进行基准测试；这是你需要自己研究的内容，以便全面了解何时使用H2O的功能。
* graphframes graphframes扩展增加了对Spark中处理图的支持。图是描述一组对象的结构，其中某些对象对在某种意义上是相关的。正如在第1章中所看到的，对网页进行排名是开发由MapReduce驱动的Spark初期版本的早期动机；如果你将页面之间的链接视为每对页面之间的关系，则网页碰巧会形成一个图。例如，图上的PageRank等计算操作在web搜索和社交网络中非常有用。
* sparktf sparktf扩展提供了在Spark中编写TensorFlow记录的支持。TensorFlow是领先的深度学习框架之一，它经常与表示为TensorFlow记录的大量数值数据一起使用，而TensorFlow记录是一种为TensorFlow优化的文件格式。Spark通常用于将非结构化的大规模数据集处理成较小的数值数据集，以便可以很容易地导入GPU。你可以使用这个扩展保存TensorFlow记录文件格式的数据集。
* xgboost xgboost扩展将著名的XGBoost建模库带到了大规模计算领域。XGBoost是一个可扩展的、可移植的、分布式的梯度增强库。它在解决希格斯玻色子机器学习挑战中获胜后，在机器学习竞赛圈内广为人知，此后在其它kaggle竞赛中一直很受欢迎。
* variantspark variantspark扩展提供了一个使用Variant Spark这一用于全基因组关联研究（GWAS）的可扩展工具包的接口。它目前提供了构建随机林模型、估计变量重要性和读取变异判读格式（VCF）文件的功能。虽然Spark中还有其它一些随机森林实现，但大多数都没有针对处理GWAS数据集的优化，这些数据集通常包含数千个样本和数百万个变量。
* geospark geospark扩展使我们能够加载和查询大型地理数据集。通常，包含经纬度点或复杂区域的数据集是 well-known text（WKT）格式定义的，WKT是一种文本标记语言，用于表示地图上的矢量几何对象。

在学习如何以及何时使用每个扩展之前，我们应该首先简要介绍如何使用R和Spark扩展。

首先，Spark扩展是接受Spark的R程序包。与其它任何R程序包一样，你首先需要安装扩展。安装好后，一定要知道在使用扩展之前需要重新连接到Spark。所以，一般来说，你应该遵循以下模式：

library(sparkextension)  
library(sparklyr)  
sc <- spark\_connect(master = "<master>")

注意，sparklyr是在扩展之后加载的，以允许扩展正确注册。如果必须安装并加载新扩展，则首先需要使用spark\_disconnect(sc)断开连接，重新启动R进程，并对新扩展重复前面的步骤。

安装和使用R中的Spark扩展并不困难；每个扩展都可以独立掌握，所以大部分时间你需要花时间了解扩展是什么、何时使用以及如何正确使用它。你将学习的第一个扩展是rsparkling扩展，它让你能够在Spark和R中使用H2O。

## H2O

由H2O.ai创建的H2O是一个用于大规模建模的开源软件，它允许你在发现数据模式时拟合数千个潜在的模型。你可以考虑使用H2O来补充或替换Spark的默认建模算法。通常使用Spark的默认建模算法，并在Spark的算法不足或需要高级功能（如其它建模指标或自动模型选择）时过渡到H2O。

我们无法在一个段落中就公正地介绍H2O强大的建模能力；合理地解释H2O需要一本书。我们建议阅读Darren Cook的《使用H2O的实用机器学习》（O’Reilly），深入探讨H2O的建模算法和特性。同时，你可以使用这一节作为一个简单的指南，开始在R中在Spark中使用R和H2O。

要使用Spark和H2O，需要知道有四个组件：H2O，Sparkling Water，rsparkling和Spark。Sparking Water允许用户将快速、可扩展的H2O机器学习算法与Spark的功能结合起来。你可以将起Sparkling Water视为连接Spark与H2O的部分，rsparkling视为Sparkling Water的R前端，如图10-1所示。

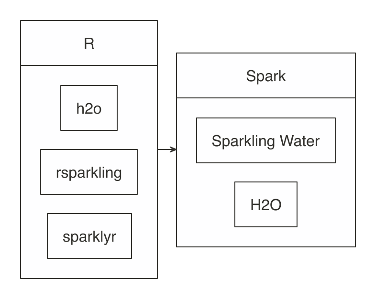


图10-1. 使用Spark和R的H2O组件

首先，按照rsparkling文档网站的介绍，安装rsparkling和h2o。

install.packages("h2o", type = "source", repos = "http://h2o-release.s3.amazonaws.com/h2o/rel-yates/5/R")  
install.packages("rsparkling", type = "source", repos = "http://h2o-release.s3.amazonaws.com/sparkling-water/rel-2.3/31/R")

需要注意的是，你需要使用文档中指定的兼容版本的Spark、Sparking Water和H2O；我们提供了Spark 2.3的说明，但使用不同的Spark版本需要你安装不同的H2O版本。因此，让我们首先通过运行以下命令来检查H2O的版本：

packageVersion("h2o")  
## [1] '3.24.0.5'  
packageVersion("rsparkling")  
## [1] '2.3.31'

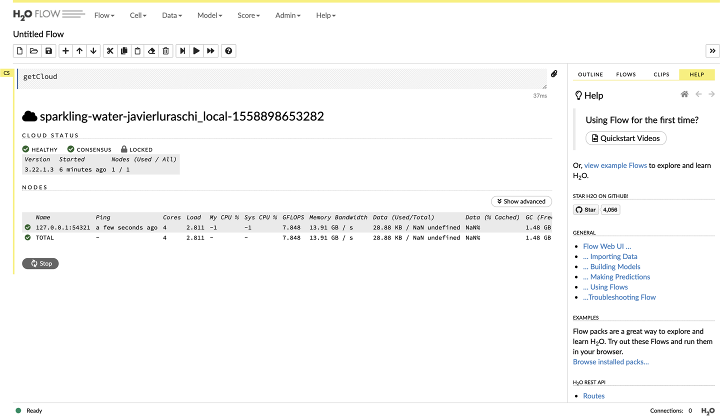
然后，我们可以按照如下代码，连接支持的Spark版本（你需要调整master参数，以便支持你的特定的集群）：

library(rsparkling)  
library(sparklyr)  
library(h2o)  
sc <- spark\_connect(master = "local", version = "2.3", config = list(sparklyr.connect.timeout = 3 \*   
 60))  
cars <- copy\_to(sc, mtcars)

H2O提供了一个web界面，可以帮助你监控训练过程并访问H2O的大部分功能。你可以通过h2o\_flow(sc)访问web界面（称为H2O流），如图10-2所示。

使用H2O时，你需要使用as\_h2o\_frame将Spark数据框转换为H2O数据数据框：

cars\_h2o <- as\_h2o\_frame(sc, cars)  
cars\_h2o  
 mpg cyl disp hp drat wt qsec vs am gear carb  
1 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4  
2 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4  
3 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1  
4 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1  
5 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2  
6 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1  
[32 rows x 11 columns]

 接着，你可以轻松地使用h2o中的许多建模函数。例如，我们可以毫不费力的拟合一个广义线性模型：

model <- h2o.glm(x = c("wt", "cyl"), y = "mpg", training\_frame = cars\_h2o, lambda\_search = TRUE)

H2O提供了一些对于Spark建模算法并不适用的额外指标。我们刚刚拟合模型提供了Residual Deviance，而使用Spark MLlib时，这不是一个标准度量。

model  
...  
MSE: 6.017684  
RMSE: 2.453097  
MAE: 1.940985  
RMSLE: 0.1114801  
Mean Residual Deviance : 6.017684  
R^2 : 0.8289895  
Null Deviance :1126.047  
Null D.o.F. :31  
Residual Deviance :192.5659  
Residual D.o.F. :29  
AIC :156.2425

然后，你可以在广义线性模型（GLM）上做预测。类似的方法在H2O的许多其它模型上都适用：

predictions <- as\_h2o\_frame(sc, copy\_to(sc, data.frame(wt = 2, cyl = 6)))  
h2o.predict(model, predictions)  
 predict  
1 24.05984  
[1 row x 1 column]

你也可以使用H2O进行模型的自动训练和调试。这意味着H2O可以通过AutoML选择你要用的模型：

automl <- h2o.automl(x = c("wt", "cyl"), y = "mpg", training\_frame = cars\_h2o, max\_models = 20,   
 seed = 1)

对于这个特定的数据集，H2O确定深度学习模型比GLM更适合[[21]](#footnote-271)。具体来说，H2O的AutoML使用XGBoost、深度学习、GLM和堆叠集成模型等模型探索数据：

automl@leaderboard  
model\_id mean\_residual\_dev… rmse mse mae rmsle  
1 DeepLearning\_… 6.541322 2.557601 6.541322 2.192295 0.1242028  
2 XGBoost\_grid\_1… 6.958945 2.637981 6.958945 2.129421 0.1347795  
3 XGBoost\_grid\_1\_… 6.969577 2.639996 6.969577 2.178845 0.1336290  
4 XGBoost\_grid\_1\_… 7.266691 2.695680 7.266691 2.167930 0.1331849  
5 StackedEnsemble… 7.304556 2.702694 7.304556 1.938982 0.1304792  
6 XGBoost\_3\_… 7.313948 2.704431 7.313948 2.088791 0.1348819

与使用排行榜不同，你可以通过automl@leader专注于最佳模型；例如，你可以按如下方式查看深度学习模型中的特定参数：

tibble::tibble(parameter = names(automl@leader@parameters),  
value = as.character(automl@leader@parameters))  
# A tibble: 20 x 2  
 parameter values  
 <chr> <chr>  
 1 model\_id DeepLearning\_grid\_1\_AutoML…  
 2 training\_frame automl\_training\_frame\_rdd…  
 3 nfolds 5  
 4 keep\_cross\_validation\_models FALSE  
 5 keep\_cross\_validation\_predictions TRUE  
 6 fold\_assignment Modulo  
 7 overwrite\_with\_best\_model FALSE  
 8 activation RectifierWithDropout  
 9 hidden 200  
10 epochs 10003.6618461538  
11 seed 1  
12 rho 0.95  
13 epsilon 1e-06  
14 input\_dropout\_ratio 0.2  
15 hidden\_dropout\_ratios 0.4  
16 stopping\_rounds 0  
17 stopping\_metric deviance  
18 stopping\_tolerance 0.05  
19 x c("cyl", "wt")  
20 y mpg

你可以使用最佳模型进行预测，代码如下：

h2o.predict(automl@leader, predictions)  
 predict  
1 30.74639  
[1 row x 1 column]

还有许多其它示例，你也可以从官方GitHub仓库请求rsparkling包的帮助。

下一个扩展graphframes允许你处理大型关系数据集。在开始使用之前，请确保使用spark\_disconnect(sc)断开连接并重新启动R进程，因为使用不同的扩展需要重新连接Spark并重新载入sparklyr。

## 图模型

图论史上的第一篇论文是由列昂哈德·欧拉于1736年写的关于柯尼斯堡七桥问题。问题是要设计一条穿过这座城市的步行道，穿过每座桥且只穿过一次。图10-3展示了原始图表。

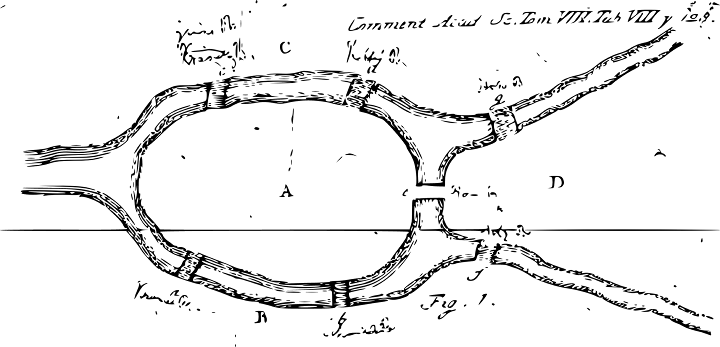


图10-3. 欧拉的柯尼斯堡七桥

今天，一个图被定义为一个有序对，其中是顶点（节点或散点），是边（连接或线）的集合。它们可以是无向图的一个无序对，也可以是有向图的有序对。前者的连接方向不重要，而而后者的连接方向是有意义的。

作为一个简单的例子，我们可以使用ggraph包中的highschool数据集，它记录了高中男生之间的朋友关系。在这个数据集中，顶点是学生，边描述了在某个年份是朋友关系的学生对：

install.packages("ggraph")  
install.packages("igraph")  
ggraph::highschool  
# A tibble: 506 x 3  
 from to year  
 <dbl> <dbl> <dbl>  
 1 1 14 1957  
 2 1 15 1957  
 3 1 21 1957  
 4 1 54 1957  
 5 1 55 1957  
 6 2 21 1957  
 7 2 22 1957  
 8 3 9 1957  
 9 3 15 1957  
10 4 5 1957  
# … with 496 more rows

虽然高中数据集可以很容易地在R中处理，但是如果不将工作分布到非常适合的Spark机器集群中，即使是中等大小的图形数据集也很难处理。Spark支持通过graphframes扩展来处理图形，而graphframes扩展又使用GraphX Spark组件。GraphX是Apache Spark用于图和图并行计算的API。它在性能上可以与最快的专用图形处理系统相媲美，并提供了一个不断增长的图算法库。

Spark中的图也表示为边和顶点的数据框；但是，我们的格式略有不同，因为我们需要为顶点构造一个数据框。首先安装graphframes扩展：

install.packages("graphframes")

接下来，我们需要连接，复制highschool数据集并将图转换为扩展所期望的格式。在这里，我们将这个数据集的范围限定在到1957年的朋友关系：

library(graphframes)  
library(sparklyr)  
library(dplyr)  
sc <- spark\_connect(master = "local", version = "2.3")  
highschool\_tbl <- copy\_to(sc, ggraph::highschool, "highschool") %>% filter(year ==   
 1957) %>% transmute(from = as.character(as.integer(from)), to = as.character(as.integer(to)))  
from\_tbl <- highschool\_tbl %>% distinct(from) %>% transmute(id = from)  
to\_tbl <- highschool\_tbl %>% distinct(to) %>% transmute(id = to)  
vertices\_tbl <- distinct(sdf\_bind\_rows(from\_tbl, to\_tbl))  
edges\_tbl <- highschool\_tbl %>% transmute(src = from, dst = to)

vertices\_tbl表需要有一个id列：

vertices\_tbl  
# Source: spark<?> [?? x 1]  
 id  
 <chr>  
 1 1  
 2 34  
 3 37  
 4 43  
 5 44  
 6 45  
 7 56  
 8 57  
 9 65  
10 71  
# … with more rows

edges\_tbl需要有src和dst列：

edges\_tbl  
# Source: spark<?> [?? x 2]  
 src dst  
 <chr> <chr>  
 1 1 14  
 2 1 15  
 3 1 21  
 4 1 54  
 5 1 55  
 6 2 21  
 7 2 22  
 8 3 9  
 9 3 15  
10 4 5  
# … with more rows

现在你可以创建一GraphFrame：

graph <- gf\_graphframe(vertices\_tbl, edges\_tbl)

现在我们可以使用这个图分析数据集。例如，我们会找出每个男孩平均有多少朋友，这被称为顶点的*度*或*价*：

gf\_degrees(graph) %>% summarise(friends = mean(degree))  
# Source: spark<?> [?? x 1]  
 friends  
 <dbl>  
1 6.94

然后我们可以找到哪一条是到某个特定顶点的最短路径（这个数据集中的一个男孩）。由于数据是匿名的，我们可以挑选一个编号为33的男孩，发现他们之间存在多少分离度：

gf\_shortest\_paths(graph, 33) %>%  
filter(size(distances) > 0) %>%  
mutate(distance = explode(map\_values(distances))) %>%  
select(id, distance)  
# Source: spark<?> [?? x 2]  
 id distance  
 <chr> <int>  
 1 19 5  
 2 5 4  
 3 27 6  
 4 4 4  
 5 11 6  
 6 23 4  
 7 36 1  
 8 26 2  
 9 33 0  
10 18 5  
# … with more rows

最后，我们也可以计算这个图的PageRank值，它在第一章讨论谷歌网页排序算法中提到过。

gf\_graphframe(vertices\_tbl, edges\_tbl) %>%  
gf\_pagerank(reset\_prob = 0.15, max\_iter = 10L)  
GraphFrame  
Vertices:  
 Database: spark\_connection  
 $ id <dbl> 12, 12, 14, 14, 27, 27, 55, 55, 64, 64, 41, 41, 47, 47, 6…  
 $ pagerank <dbl> 0.3573460, 0.3573460, 0.3893665, 0.3893665, 0.2362396, 0.…  
Edges:  
 Database: spark\_connection  
 $ src <dbl> 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 12, 12, 12,…  
 $ dst <dbl> 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17,…  
 $ weight <dbl> 0.25000000, 0.25000000, 0.25000000, 0.25000000, 0.25000000,…

为了让你理解数据集，图10-4使用ggraph画出了这个图，并高亮了下列数据集的最高PageRank得分：

highschool\_tbl %>% igraph::graph\_from\_data\_frame(directed = FALSE) %>% ggraph(layout = "kk") +   
 geom\_edge\_link(alpha = 0.2, arrow = arrow(length = unit(2, "mm")), end\_cap = circle(2,   
 "mm"), start\_cap = circle(2, "mm")) + geom\_node\_point(size = 2, alpha = 0.4)

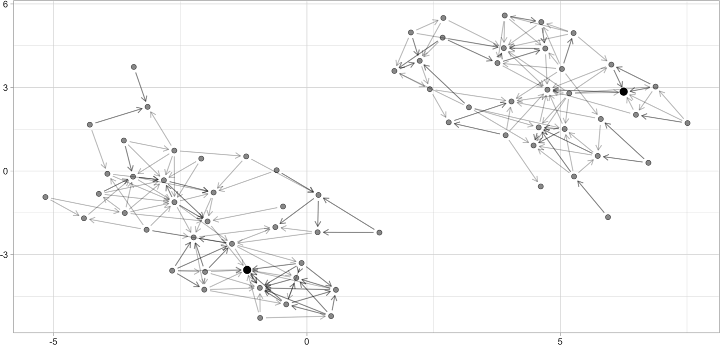


图10-4. 最高PageRank得分高亮的highschool数据集ggraph可视化

graphframes还提供了更多的图算法，例如广度优先搜索、连接分支、用于检测社区的标签传递、强连接分支和三角形计数。有关此扩展的问题，请参阅官方GitHub仓库。我们现在提供一个流行的梯度提升框架。确保在尝试下一个扩展之前断开连接并重新启动。

## XGBoost

*决策树*是一种类似流程图的结构，其中每个内部节点表示一个属性上的测试，每个分支表示测试的结果，每个叶节点表示一个类标签。例如，图10-5展示了一个决策树，该树可以帮助解决在给定一组因素（如工作满意度和加班）情况下员工是否可能离职的分类问题。当决策树被用来预测连续变量而不是离散结果时，比如某人离开公司的可能性有多大，它被称为*回归树*。

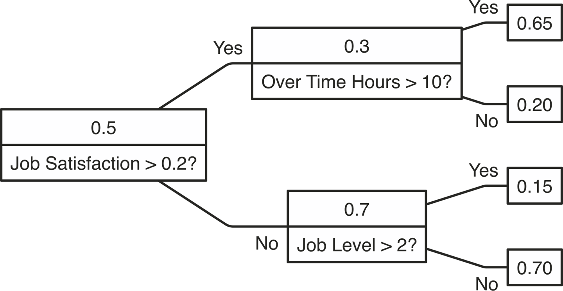


图10-5. 基于已知因素的预测离职的决策树

虽然决策树表示很容易理解和解释，但要找出树中的决策，需要像梯度下降这样的数学技术才能找到局部最小值。梯度下降与当前点处函数梯度的负方向成比例。梯度用表示，学习率用表示。你只需从给定的状态开始，然后按照梯度的方向计算下一次迭代

XGBoost是一个开源软件库，它提供了一个梯度提升框架。它旨在为训练梯度提升决策树（GBDT）和梯度提升回归树（GBRT）提供可扩展、可移植和分布式的梯度提升方法。梯度提升意味着XGBoost使用梯度下降和梯度提升，这是一种按顺序选择每个预测器的技术。

sparkxgb是一个扩展，你可以使用它在Spark中训练XGBoost模型；但是，请注意，当前不支持Windows。要使用此扩展，请首先从CRAN安装它：

install.packages("sparkxgb")

然后，你需要导入sparkxgb扩展，然后执行常用的Spark连接代码，按需调整master：

library(sparkxgb)  
library(sparklyr)  
library(dplyr)  
sc <- spark\_connect(master = "local", version = "2.3")

在本例中，我们使用rsample包中的attrition数据集，你需要使用install.packages("rsample")来安装它。这是一个虚构的数据集，由IBM数据科学家创建，以揭示导致员工离职的原因：

attrition <- copy\_to(sc, rsample::attrition)  
attrition  
# Source: spark<?> [?? x 31]  
 Age Attrition BusinessTravel DailyRate Department DistanceFromHome  
 <int> <chr> <chr> <int> <chr> <int>  
 1 41 Yes Travel\_Rarely 1102 Sales 1  
 2 49 No Travel\_Freque… 279 Research\_… 8  
 3 37 Yes Travel\_Rarely 1373 Research\_… 2  
 4 33 No Travel\_Freque… 1392 Research\_… 3  
 5 27 No Travel\_Rarely 591 Research\_… 2  
 6 32 No Travel\_Freque… 1005 Research\_… 2  
 7 59 No Travel\_Rarely 1324 Research\_… 3  
 8 30 No Travel\_Rarely 1358 Research\_… 24  
 9 38 No Travel\_Freque… 216 Research\_… 23  
10 36 No Travel\_Rarely 1299 Research\_… 27  
# … with more rows, and 25 more variables: Education <chr>,  
# EducationField <chr>, EnvironmentSatisfaction <chr>, Gender <chr>,  
# HourlyRate <int>, JobInvolvement <chr>, JobLevel <int>, JobRole <chr>,  
# JobSatisfaction <chr>, MaritalStatus <chr>, MonthlyIncome <int>,  
# MonthlyRate <int>, NumCompaniesWorked <int>, OverTime <chr>,  
# PercentSalaryHike <int>, PerformanceRating <chr>,  
# RelationshipSatisfaction <chr>, StockOptionLevel <int>,  
# TotalWorkingYears <int>, TrainingTimesLastYear <int>,  
# WorkLifeBalance <chr>, YearsAtCompany <int>, YearsInCurrentRole <int>,  
# YearsSinceLastPromotion <int>, YearsWithCurrManager <int>

要在Spark中构建XGBoost模型，使用xgboost\_classifier()。我们使用Attrition ~ .公式计算所有其它特性对离职的影响，并指定为2类，因为离职属性只记录员工是离开还是留下。然后，可以使用ml\_predict()对大规模数据集进行预测：

xgb\_model <- xgboost\_classifier(attrition,  
Attrition ~ .,  
num\_class = 2,  
num\_round = 50,  
max\_depth = 4)  
xgb\_model %>%  
ml\_predict(attrition) %>%  
select(Attrition, predicted\_label, starts\_with("probability\_")) %>%  
glimpse()  
Observations: ??  
Variables: 4  
Database: spark\_connection  
$ Attrition <chr> "Yes", "No", "Yes", "No", "No", "No", "No", "No", "No", …  
$ predicted\_label <chr> "No", "Yes", "No", "Yes", "Yes", "Yes", "Yes", "Yes", "Y…  
$ probability\_No <dbl> 0.753938094, 0.024780750, 0.915146366, 0.143568754, 0.07…  
$ probability\_Yes <dbl> 0.24606191, 0.97521925, 0.08485363, 0.85643125, 0.927375…

在Higgs机器学习挑战赛的胜出解决方案中使用XGBoost后，XGBoost在竞赛圈中广为人知，该方案使用ATLAS实验来识别Higgs玻色子。从此，XGBoost已经成为一种流行的模式，并用于大量的kaggle比赛。然而，决策树被证明能力很有限，特别是在具有非表格接数据（如图像、音频和文本）的数据集中。而这些问题使用深度学习技术会更好。提醒一下，记得断开连接并重新启动。

## 深度学习

感知器是Frank Rosenblatt提出的一种数学模型[[22]](#footnote-278)，他把感知器发展成为一种假想的神经系统的理论。感知器将刺激映射到数值输入，数值输入加权到阈值函数中，阈值函数仅在存在足够的刺激时激活，数学表示如下：

Minsky和Papert发现，一个感知器只能对线性可分的数据集进行分类；然而，他们在其著作《Perceptrons》中也揭示了分层感知器将带来额外的分类能力[[23]](#footnote-279)。图10-6给出了多层感知器最初的样子。

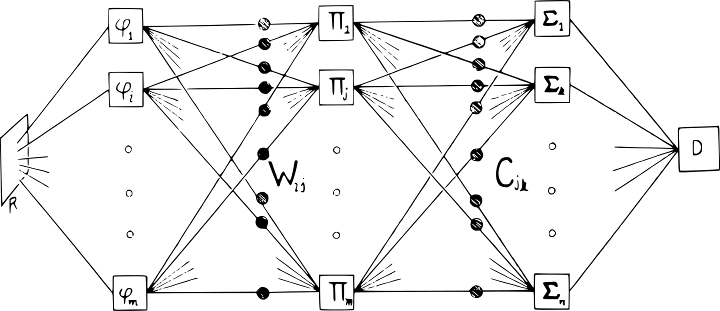


图10-6. 《Perceptrons》中描述的多层感知机

开始之前，首先安装要使用的所有程序包：

install.packages("sparktf")  
install.packages("tfdatasets")

使用Spark，我们可以创建一个多层感知器分类器，该分类器使用ml\_multilayer\_perceptron\_classifier()和梯度下降来对大型数据集进行分类和预测。Geoff Hinton将梯度下降引入分层感知器[[24]](#footnote-281)。

library(sparktf)  
library(sparklyr)  
sc <- spark\_connect(master = "local", version = "2.3")  
attrition <- copy\_to(sc, rsample::attrition)  
nn\_model <- ml\_multilayer\_perceptron\_classifier(  
attrition,  
Attrition ~ Age + DailyRate + DistanceFromHome + MonthlyIncome,  
layers = c(4, 3, 2),  
solver = "gd")  
nn\_model %>%  
ml\_predict(attrition) %>%  
select(Attrition, predicted\_label, starts\_with("probability\_")) %>%  
glimpse()  
Observations: ??  
Variables: 4  
Database: spark\_connection  
$ Attrition <chr> "Yes", "No", "Yes", "No", "No", "No", "No", "No", "No"…  
$ predicted\_label <chr> "No", "No", "No", "No", "No", "No", "No", "No", "No", …  
$ probability\_No <dbl> 0.8439275, 0.8439275, 0.8439275, 0.8439275, 0.8439275,…  
$ probability\_Yes <dbl> 0.1560725, 0.1560725, 0.1560725, 0.1560725, 0.1560725,…

注意，列必须是数值型的，因此需要使用第4章中介绍的特征转换技术手动转换它们。尝试添加更多的层来分类更复杂的数据集是很自然的；但是，添加太多的层将导致梯度散失，其它技术需要使用这些深层网络，也称为深层学习模型。

深度学习模型利用特殊的激活函数、dropout、数据增强和GPU来解决梯度散失问题。你可以使用Spark检索大型数据集，并将其预处理为只包含数值的数据集，这些数据集可以在GPU中进行深度学习训练。TensorFlow是最流行的深度学习框架之一。如前所述，它支持称为TensorFlow记录的二进制格式。

你可以使用Spark中的sparktf编写TensorFlow记录，并准备在GPU实例中使用Keras或TensorFlow等库来处理这些记录。

然后，可以在Spark中预处理大型数据集，并使用spark\_write\_tf()将它们作为TensorFlow记录写入：

copy\_to(sc, iris) %>% ft\_string\_indexer\_model("Species", "label", labels = c("setosa",   
 "versicolor", "virginica")) %>% spark\_write\_tfrecord(path = "tfrecord")

使用Keras或TensorFlow训练数据集后，可以使用tfdatasets包加载它。你还需要使用install\_tensorflow() 安装TensorFlow运行时，并自行安装Python。要了解更多关于使用Keras训练深度学习模型的信息，建议阅读使用《Deep Learning with R》[[25]](#footnote-282)。

tensorflow::install\_tensorflow()  
tfdatasets::tfrecord\_dataset("tfrecord/part-r-00000")  
<DatasetV1Adapter shapes: (), types: tf.string>

使用一个或多个GPU在单个本地节点上训练深度学习模型对于大多数应用程序来说通常已经足够了；但是，最近最先进的深度学习模型使用Apache Spark等分布式计算框架进行训练。分布式计算框架被用来实现更高的千万亿次浮点运算，支持系统每天都在训练这些模型。OpenAI分析了人工智能（AI）和集群计算领域的趋势（如图10-7所示）。从图中可以明显看出，近年来有使用分布式计算框架的趋势。

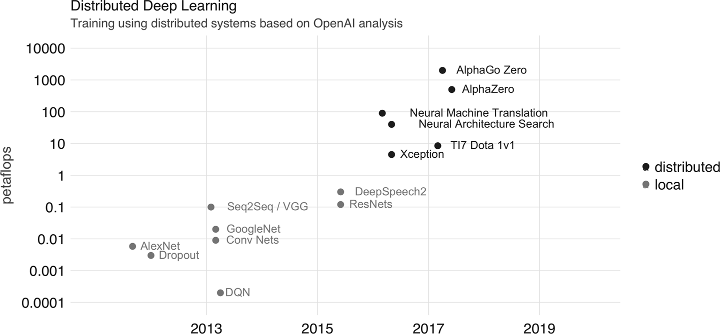


图10-7. 基于OpenAI分析使用分布式系统的训练

通过像Horovod这样的框架，Spark和TensorFlow可以训练大规模的深度学习模型。今天，可以通过reticulate在R中使用Horovod和Spark，因为Horovod需要Python和Open MPI，这超出了本书的范围。接下来，我们将介绍一个基因组学领域的不同的Spark扩展。

## 基因组学

人类基因组由23对染色体中约30亿对碱基DNA的两个拷贝组成。图10-8展示了基因组形成染色体。DNA链由核苷酸组成，每个核苷酸由四个含氮碱基中的一个组成：胞嘧啶（C）、鸟嘌呤（G）、腺嘌呤（A）或胸腺嘧啶（T）。由于所有人类的DNA几乎是相同的，我们只需要以一个变异判读格式（VCF）文件的形式存储与参考基因组的差异。

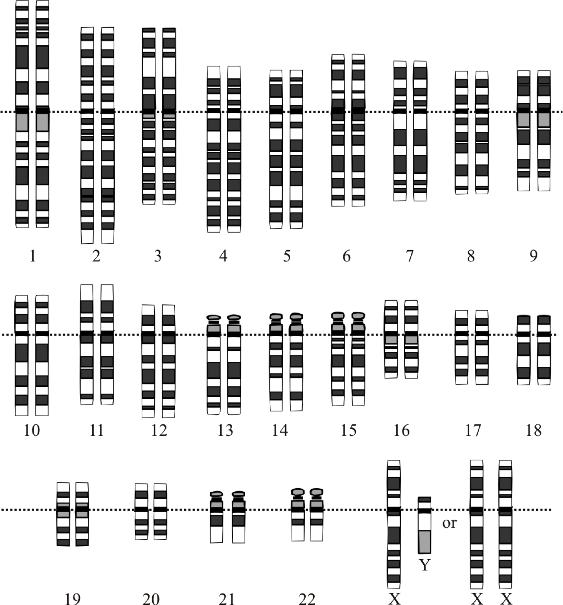


图10-8. 理想化的人类二倍体核型，显示基因组形成染色体

variantspark是一个基于Scala和Spark的分析基因组数据集的框架。它是由澳大利亚的CSIRO生物信息学团队开发的。variantspark在3000个样本的数据集上进行了测试，每个样本包含8000万个非监督聚类方法或监督应用（如分类和回归）的特征。variantspark可以读取VCF文件并运行分析，同时使用熟悉的Spark数据框。

首先，从CRAN安装variantspark，连接到Spark，然后得到variantspark的vsc连接：

library(variantspark)  
library(sparklyr)  
sc <- spark\_connect(master = "local", version = "2.3", config = list(sparklyr.connect.timeout = 3 \*   
 60))  
vsc <- vs\_connect(sc)

我们可以开始加载VCF文件：

vsc\_data <- system.file("extdata/", package = "variantspark")  
hipster\_vcf <- vs\_read\_vcf(vsc, file.path(vsc\_data, "hipster.vcf.bz2"))  
hipster\_labels <- vs\_read\_csv(vsc, file.path(vsc\_data, "hipster\_labels.txt"))  
labels <- vs\_read\_labels(vsc, file.path(vsc\_data, "hipster\_labels.txt"))

variantspark使用随机森林为每个测试的变异分配一个重要分数，反映其与兴趣表型的关联。一个重要度得分较高的变异意味着它与感兴趣的表型有更紧密的联系。你可以计算重要性并将其转换为Spark表，如下所示：

importance\_tbl <- vs\_importance\_analysis(vsc, hipster\_vcf,  
labels, n\_trees = 100) %>%  
importance\_tbl()  
importance\_tbl  
# Source: spark<?> [?? x 2]  
 variable importance  
 <chr> <dbl>  
 1 2\_109511398 0  
 2 2\_109511454 0  
 3 2\_109511463 0.00000164  
 4 2\_109511467 0.00000309  
 5 2\_109511478 0  
 6 2\_109511497 0  
 7 2\_109511525 0  
 8 2\_109511527 0  
 9 2\_109511532 0  
10 2\_109511579 0  
# … with more rows

然后，你可以使用dplyr和ggplot2 转换输出，并分析（见图10-9）：

library(dplyr)  
library(ggplot2)  
importance\_df <- importance\_tbl %>% arrange(-importance) %>% head(20) %>% collect()  
ggplot(importance\_df) + aes(x = variable, y = importance) + geom\_bar(stat = "identity") +   
 scale\_x\_discrete(limits = importance\_df[order(importance\_df$importance), 1]$variable) +   
 coord\_flip()

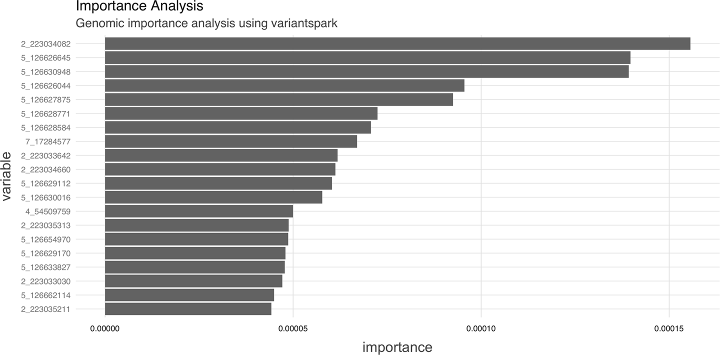


图10-9. 使用variantspark的基因组重要性分析

最后我们在Spark中使用variantspark扩展，简要介绍了基因组分析。接下来，我们将从微观基因转移到宏观数据集，这些数据集包含世界各地的地理位置。

## 空间数据

geospark使用与dplyr和sf兼容的语法支持分布式地理空间计算，该语法提供了一组用于处理地理空间矢量的工具。

你可以从CRAN安装geospark，如下所示：

install.packages("geospark")

然后，初始化geospark扩展并连接到Spark：

library(geospark)  
library(sparklyr)  
sc <- spark\_connect(master = "local", version = "2.3")

接着，加载一个包含多边形和点的空间数据集：

polygons <- system.file("examples/polygons.txt", package = "geospark") %>% read.table(sep = "|",   
 col.names = c("area", "geom"))  
points <- system.file("examples/points.txt", package = "geospark") %>% read.table(sep = "|",   
 col.names = c("city", "state", "geom"))  
polygons\_wkt <- copy\_to(sc, polygons)  
points\_wkt <- copy\_to(sc, points)

geospark中定义了各种空间操作，如图10-10所示。

这些操作允许你控制基于重叠、相交、不相交集等的地理空间数据查询过程。

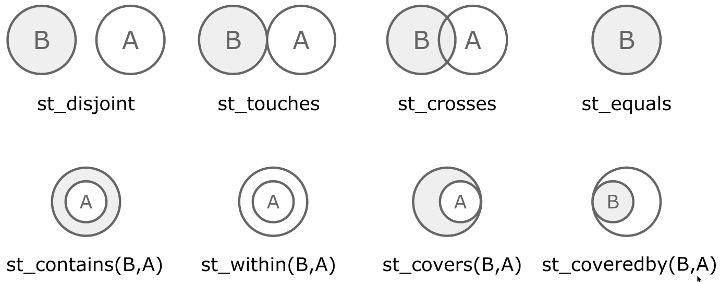


图10-10. geospark上的空间操作

例如，我们可以使用这些操作,借助st\_contains()找出包含给定点集合的多边形：

library(dplyr)  
polygons\_wkt <- mutate(polygons\_wkt, y = st\_geomfromwkt(geom))  
points\_wkt <- mutate(points\_wkt, x = st\_geomfromwkt(geom))  
inner\_join(polygons\_wkt,  
points\_wkt,  
sql\_on = sql("st\_contains(y,x)")) %>%  
group\_by(area, state) %>%  
summarise(cnt = n())  
# Source: spark<?> [?? x 3]  
# Groups: area  
 area state cnt  
 <chr> <chr> <dbl>  
1 california area CA 10  
2 new york area NY 9  
3 dakota area ND 10  
4 texas area TX 10  
5 dakota area SD 1

你还可以通过收集整个数据集的子集或在收集之前在Spark中聚合几何图形来绘制这些数据集。一个包值得学习的包是sf。

本章末尾将介绍适用于所有扩展的两种故障排除技术。

## 故障排除

当你第一次使用新扩展时，我们建议增加连接超时（考虑到Spark通常需要下载扩展依赖项），并将日志记录更改为详细记录，以帮助你在未完成下载过程时进行故障排除：

config <- spark\_config()  
config$sparklyr.connect.timeout <- 3 \* 60  
config$sparklyr.log.console = TRUE  
sc <- spark\_connect(master = "local", config = config)

此外，你应该知道Apache IVY是一种流行的依赖管理器，侧重于灵活性和简单性，并被Apache Spark用于安装扩展。当使用扩展连接失败时，考虑通过运行以下命令清除IVY缓存：

unlink("~/.ivy2", recursive = TRUE)

此外，你还可以考虑从以下扩展仓库中打开GitHub问题，以获得扩展作者的帮助：

* rsparkling
* sparkxgb
* sparktf
* variantspark
* geospark

## 小结

本章简要介绍了使用R中提供的一些Spark扩展，与安装软件包一样简单。然后，你学习了如何使用rsparkling扩展，该扩展提供了Spark中的H请求访问，进而提供了其它建模功能，如增强度量和自动选择模型的能力。然后我们跳到graphframes，这个扩展可以帮助你处理图的关系数据集。你还学习了如何计算简单的连接度量或运行复杂的算法，如PageRank。

XGBoost和深度学习部分提供了使用梯度下降的备选建模技术：前者基于决策树，后者基于多层感知器，我们可以使用Spark将数据集预处理为记录，然后使用sparktf扩展支持TensorFlow和Keras使用。最后两部分介绍了通过variantspark和geospark扩展来处理基因组和空间数据集的能力。

这些扩展，以及更多的扩展，提供了一个具备高级功能的综合库。结合所介绍的分析和建模技术，这些扩展应该涵盖在计算集群运行中所需的大多数任务。然而，当缺少功能时，你可以考虑编写自己的扩展，这是我们在第13章中讨论的，或者如马上在第11章看到的，你可以使用R代码对每个分区进行自定义转换。

# 第11章 分布式R

不是这样的，不是这样的，不是这样的。 —Cersei Lannister

在前面的章节中，你学习了如何在本地Spark实例和合适的Spark集群中执行数据分析和建模。在第10章中，我们专门研究了如何利用Spark和R社区提供的额外功能。在大多数情况下，Spark功能和扩展的结合足以执行几乎所有的计算。但是，对于Spark及其扩展缺少功能的情况，可以考虑自己将R计算分发给工作节点。

你可以在每个工作节点中运行任意R代码来运行任何计算，可以运行模拟、从web抓取内容、转换数据等。此外，你还可以使用CRAN中可用的任何包和你所在组织中可用的私有包，这将减少编写的代码量，以帮助维持生产效率。

如果你已经熟悉R，那么可能会尝试将此方法用于所有Spark操作；但是，这不在是建议使用spark\_apply()。前几章提供了更有效的技术和工具来解决众所周知的问题；相反，spark\_apply()引入了额外的认知开销、额外的故障排除步骤、性能权衡，并且通常应该避免更多的复杂性。并不是说不应该使用spark\_apply()，而是保留spark\_apply()来对付以前的工具和技术所不能满足的用例。

## 概述

第1章介绍了MapReduce可以作为一种能够处理大规模数据集的技术。它还描述了Apache Spark如何提供一个操作超集，以便更轻松、更高效地执行MapReduce计算。第9章通过对分布式数据集的每个分区应用自定义转换，深入了解了Spark是如何工作的。例如，如果我们将一个分布式数值数据集的每个元素乘以10，Spark将通过多个工作节点在每个分区上应用一个映射操作。这个过程的概念图如图11-1所示。

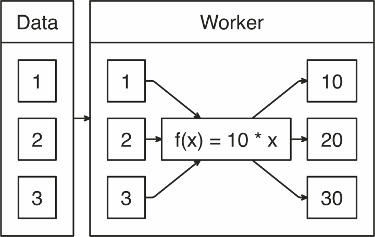


图11-1. 乘以10的映射操作

本章介绍如何使用spark\_apply()定义一个自定义的f(x)映射操作；对于之前的例子，spark\_apply()支持定义10 \* x，如下所示：

sdf\_len(sc, 3) %>% spark\_apply(~ 10 \* .x)  
# Source: spark<?> [?? x 1]  
 id  
\* <dbl>  
1 10  
2 20  
3 30

注意~ 10 \* .x是跨所有工作节点执行的普通R代码。~运算符在rlang包中定义，并提供与函数function(.x) 10 \* .x等效的函数的简洁定义；这种简洁形式也称为匿名函数或lambda表达式。

f(x)函数必须以一个R数据框（或可以自动转换为数据框一个的对象）作为输入，以及生成一个R数据框作为输出，如图11-2所示。

图11-2. spark_apply()映射中的期望函数样式

图11-2. spark\_apply()映射中的期望函数样式

我们可以参考第一章中最初的MapReduce示例，其中map操作被定义为将句子拆分为单词，并且对所有唯一单词技术当为reduce操作。

在R中，我们可以使用tidytext包中的unnest\_tokens()函数。在连接到Spark之前，你需要从CRAN安装该函数。然后，可以使用tidytext和spark\_apply()将这些句子切分为一个单词表：

sentences <- copy\_to(sc, data\_frame(text = c("I like apples", "I like bananas")))  
sentences %>%  
spark\_apply(~tidytext::unnest\_tokens(.x, word, text))  
# Source: spark<?> [?? x 1]  
 word  
\* <chr>  
1 i  
2 like  
3 apples  
4 i  
5 like  
6 bananas

我们可以通过dplyr运行reduce操作完成MapReduce，代码如下：

sentences %>%  
spark\_apply(~tidytext::unnest\_tokens(., word, text)) %>%  
group\_by(word) %>%  
summarise(count = count())  
# Source: spark<?> [?? x 2]  
 word count  
\* <chr> <dbl>  
1 i 2  
2 apples 1  
3 like 2  
4 bananas 1

本章的其余部分将详细解释通过spark\_apply()自定义映射时所需的用例、特性、注意事项、关注点和故障排除技术。

之前的句子切分示例可以使用之前章节的理念，更高效的实现，即通过sentences %>% ft\_tokenizer("text", "words") %>% transmute(word = explode(words))。

## 用例

我们已经提供了一个示例来帮助你理解spark\_apply()，现在我们介绍一些实用的例子：

* 导入 可以考虑使用R从外部数据源和格式导入数据。例如，当一个文件格式无法获得Spark或其扩展的原生支持时，可以考虑用R代码并借助R程序包实现分布式自定义解析器。
* 建模 使用Spark和R中已有的丰富建模功能是很自然的。在大多数情况下，R模型不能跨大数据使用；但是，我们会专门展示两个用例来说明其中R模型可以在大数据上使用。例如，当数据适合一台机器时，可以使用网格搜索并行优化它们的参数。在一些情况下，我们可以对数据进行分区，并在数据子集上创建多个模型，我们可以使用R中的分区建模来跨分区计算模型。
* 变换 你可以使用R的丰富数据转换功能来补充Spark。我们会展示一个通过外部系统评估数据的用例，并使用R通过web API调用它们来进行交互。
* 计算 当你需要在R中执行大规模计算，或者如第1章所述执行大型计算时，Spark是分发此计算的理想选择。我们会模拟R中大规模计算的一个特殊用例。

当我们现在详细研究每个用例时，我们会提供一个工作示例，帮助了解如何有效地使用spark\_apply()。

### 定制解析器

尽管Spark及其各种扩展提供了对许多文件格式（csv、JSON、Parquet、AVRO等）的支持，但你会在大规模数据上能需要其它格式。你可以使用spark\_apply()和许多现有的R程序包解析这些额外的格式。在本节中，我们将研究如何解析日志文件。使用类似的方法你还可以解析其它文件格式。

Spark通常用来分析日志文件，例如跟踪Amazon S3下载数据的日志。webreadr包通过支持加载存储为Amazon S3、Squid和Common格式的日志，可以简化解析日志的过程。在连接到Spark之前，应该从CRAN安装webreadr。

例如，Amazon S3日志如下所示：

#Version: 1.0  
#Fields: date time x-edge-location sc-bytes c-ip cs-method cs(Host) cs-uri-stem  
 sc-status cs(Referer) cs(User-Agent) cs-uri-query cs(Cookie) x-edge-result-type  
 x-edge-request-id x-host-header cs-protocol cs-bytes time-taken  
2014-05-23 01:13:11 FRA2 182 192.0.2.10 GET d111111abcdef8.cloudfront.net  
 /view/my/file.html 200 www.displaymyfiles.com Mozilla/4.0%20  
 (compatible;%20MSIE%205.0b1;%20Mac\_PowerPC) - zip=98101 RefreshHit  
 MRVMF7KydIvxMWfJIglgwHQwZsbG2IhRJ07sn9AkKUFSHS9EXAMPLE==  
 d111111abcdef8.cloudfront.net http - 0.001

这些内容可以使用read\_aws()快速的解析，代码如下：

aws\_log <- system.file("extdata/log.aws", package = "webreadr")  
webreadr::read\_aws(aws\_log)  
# A tibble: 2 x 18  
 date edge\_location bytes\_sent ip\_address http\_method host path  
 <dttm> <chr> <int> <chr> <chr> <chr> <chr>  
1 2014-05-23 01:13:11 FRA2 182 192.0.2.10 GET d111… /vie…  
2 2014-05-23 01:13:12 LAX1 2390282 192.0.2.2… GET d111… /sou…  
# ... with 11 more variables: status\_code <int>, referer <chr>, user\_agent <chr>,  
# query <chr>, cookie <chr>, result\_type <chr>, request\_id <chr>,  
# host\_header <chr>, protocol <chr>, bytes\_received <chr>, time\_elapsed <dbl>

要扩大这个操作，我们可以借助spark\_apply()使用read\_aws()：

spark\_read\_text(sc, "logs", aws\_log, overwrite = TRUE, whole = TRUE) %>%  
spark\_apply(~webreadr::read\_aws(.x$contents))  
# Source: spark<?> [?? x 18]  
 date edge\_location bytes\_sent ip\_address http\_method host path  
\* <dttm> <chr> <int> <chr> <chr> <chr> <chr>  
1 2014-05-23 01:13:11 FRA2 182 192.0.2.10 GET d111… /vie…  
2 2014-05-23 01:13:12 LAX1 2390282 192.0.2.2… GET d111… /sou…  
# ... with 11 more variables: status\_code <int>, referer <chr>, user\_agent <chr>,  
# query <chr>, cookie <chr>, result\_type <chr>, request\_id <chr>,  
# host\_header <chr>, protocol <chr>, bytes\_received <chr>, time\_elapsed <dbl>

纯R代码和spark\_apply()代码很相似。但是通过spark\_apply()，日志可以在集群中的所有工作节点上解析。

自定义解析器的讨论到此结束；你可以按照类似的方法用R解析大数据的许多其它文件格式。接下来，我们把当前的分区建模看作另一个用例，关注跨多个数据集的并行建模。

### 分区建模

R中有许多可以运行在大数据上的建模包。它们将数据划分成易于管理的几组数据，每组数据都可以放在单个机器上。

例如，假设你有一个1TB的数据集，包含多个城市的销售数据，并且你的任务是在每个城市构建销售预测。在这种情况下，可以考虑将原始数据集划分为10GB的单个城市数据，这些数据可以由单个计算实例管理。对于这种可分区的数据集，还可以考虑使用spark\_apply() ，为每个城市上训练一个模型。

作为分区建模的简单示例，我们可以使用按物种分区的iris数据集进行线性回归：

iris <- copy\_to(sc, datasets::iris)  
iris %>%  
spark\_apply(nrow, group\_by = "Species")  
# Source: spark<?> [?? x 2]  
 Species result  
 <chr> <int>  
1 versicolor 50  
2 virginica 50  
3 setosa 50

然后，你可以使用spark\_apply()在每个物种上执行线性回归：

iris %>%  
spark\_apply(  
function(e) summary(lm(Petal\_Length ~ Petal\_Width, e))$r.squared,  
names = "r.squared",  
group\_by = "Species")  
# Source: spark<?> [?? x 2]  
 Species r.squared  
 <chr> <dbl>  
1 versicolor 0.619  
2 virginica 0.104  
3 setosa 0.110

从r.squared结果和图11-3可以看到，versicolor的线性模型可以更好的拟合成回归曲线：

purrr::map(c("versicolor", "virginica", "setosa"), ~dplyr::filter(datasets::iris,   
 Species == !!.x) %>% ggplot2::ggplot(ggplot2::aes(x = Petal.Length, y = Petal.Width)) +   
 ggplot2::geom\_point())

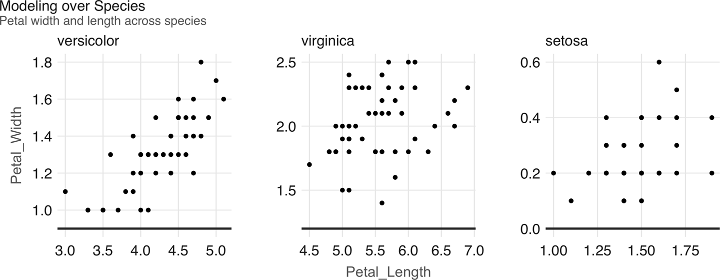


图11-3. iris数据集在不同物种上的建模结果 dataset

我们简单介绍了如何在几个不同的可分区数据集上执行建模。类似的技术可以应用于同一个数据集上的使用不同参数的建模。下面就会介绍。

### 网格搜索

许多R程序包提供了需要定义多个参数完成配置和优化的模型。当这些参数的值未知时，我们可以将一列未知参数发放到一组机器上，以找到最佳的参数组合。如果这列参数包含多个要优化的参数，则通常会通过参数A和参数B之间的所有组合进行测试，从而创建一个参数网格。在这个参数网格上搜索最佳参数的过程通常称为网格搜索。

例如，我们可以定义一个参数网格来优化决策树模型，如下所示：

grid <- list(minsplit = c(2, 5, 10), maxdepth = c(1, 3, 8)) %>%  
purrr:::cross\_df() %>%  
copy\_to(sc, ., repartition = 9)  
grid  
# Source: spark<?> [?? x 2]  
 minsplit maxdepth  
 <dbl> <dbl>  
1 2 1  
2 5 1  
3 10 1  
4 2 3  
5 5 3  
6 10 3  
7 2 8  
8 5 8  
9 10 8

使用repartition = 9来复制网格数据集，以确保每个机器包含一个分区，因为网格也有九行。现在，假设原始数据集可以放入每台计算机，我们可以把这个数据集分发给多台计算机，并执行参数搜索以找到最适合此数据的模型：

spark\_apply(  
grid,  
function(grid, cars) {  
model <- rpart::rpart(  
am ~ hp + mpg,  
data = cars,  
control = rpart::rpart.control(minsplit = grid$minsplit,  
maxdepth = grid$maxdepth) )  
dplyr::mutate(  
grid,  
accuracy = mean(round(predict(model, dplyr::select(cars, -am))) == cars$am) )  
},  
context = mtcars)  
# Source: spark<?> [?? x 3]  
 minsplit maxdepth accuracy  
 <dbl> <dbl> <dbl>  
1 2 1 0.812  
2 5 1 0.812  
3 10 1 0.812  
4 2 3 0.938  
5 5 3 0.938  
6 10 3 0.812  
7 2 8 1  
8 5 8 0.938  
9 10 8 0.812

对于这个模型，minsplit = 2和maxdepth = 8可以生成最准确的结果。现在你可以使用这些具体的参数组合来训练你的模型了。

### Web API

Web API是一个程序，它通过其它程序可重用的w网络接口来做一些有用的事情。例如，Twitter之类的服务提供了Web API，允许你用R和其它编程语言编写的程序自动读取推文。通过使用R代码向外部服务发送实际请求，可以使用spark\_apply()调用Web API。

例如，谷歌提供了一个Web API来使用深度学习技术标注图像；你可以通过R使用这个API。但是对于更大的数据集，你需要从Spark访问它的API。你可以使用Spark准备流向Web API的数据，然后使用spark\_apply()完成调用并在Spark中处理所有传入的结果。

下一个示例使用googleAuthR包向谷歌云完成身份验证，使用RoogleVision包在谷歌视觉API上执行标记，使用spark\_apply()在spark和谷歌的深度学习服务之间进行交互。要运行以下示例，首先需要断开与Spark的连接，并从谷歌开发者门户下载cloudml.json文件：

sc <- spark\_connect(  
master = "local",  
config = list(sparklyr.shell.files = "cloudml.json"))  
images <- copy\_to(sc, data.frame(  
image = "http://pbs.twimg.com/media/DwzcM88XgAINkg-.jpg"  
))  
spark\_apply(images, function(df) {  
googleAuthR::gar\_auth\_service(  
scope = "https://www.googleapis.com/auth/cloud-platform",  
json\_file = "cloudml.json")  
RoogleVision::getGoogleVisionResponse(  
df$image,  
download = FALSE)  
})  
# Source: spark<?> [?? x 4]  
 mid description score topicality  
 <chr> <chr> <dbl> <dbl>  
1 /m/04rky Mammal 0.973 0.973  
2 /m/0bt9lr Dog 0.958 0.958  
3 /m/01z5f Canidae 0.956 0.956  
4 /m/0kpmf Dog breed 0.909 0.909  
5 /m/05mqq3 Snout 0.891 0.891

要在web API上成功运行大型分布式计算，API需要能够扩展以支持来自所有Spark执行器的负载。我们相信主流服务提供商可能支持从集群传入的所有请求。但是，当你调用内部web API时，请确保API能够处理负载。此外，当使用第三方服务时，需要考虑跨集群中的所有执行器调用API的成本，以避免潜在的昂贵和意外的费用。

接下来，我们会介绍一个用于大型计算的例子，其中R用于执行分布式图像渲染。

### 模拟

你可以使用R和Spark来执行大规模计算。我们在这里探索的用例是使用rayrender包渲染计算成本高昂的图像，该包使用射线跟踪，这是电影制作中常用的保真技术。

让我们使用这个包来渲染一个简单的场景，其中包括几个使用朗博材料、漫反射材料或哑光的球体（见图11-4）。首先，使用install.packages("rayrender")安装rayrender。然后，确保已断开并重新连接Spark：

library(rayrender)  
scene <- generate\_ground(material = lambertian()) %>% add\_object(sphere(material = metal(color = "orange"),   
 z = -2)) %>% add\_object(sphere(material = metal(color = "orange"), z = +2)) %>%   
 add\_object(sphere(material = metal(color = "orange"), x = -2))  
render\_scene(scene, lookfrom = c(10, 5, 0), parallel = TRUE)

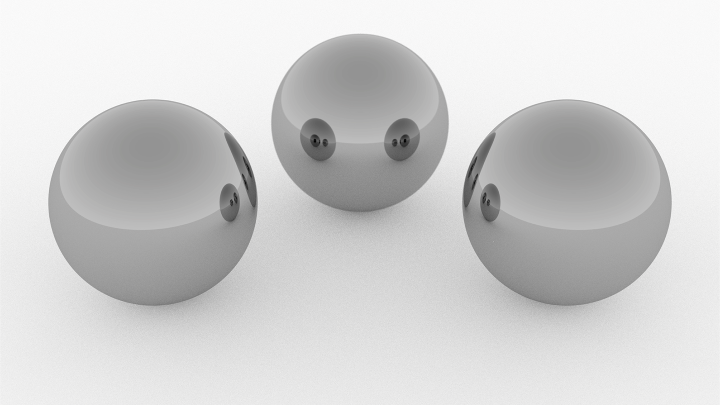


图11-4. Spark中使用R和rayrender的射线跟踪

在更高的分辨率，例如1920 x 1080中，上一个示例需要几分钟来渲染图11-4中的单个帧；单个机器以每秒30帧的速度渲染将需要数小时。但是，我们可以通过在多台机器上并行计算来减少这一时间。例如，使用10台具有相同数量CPU的计算机可以将渲染时间缩短为十分之一：

system2("hadoop", args = c("fs", "-mkdir", "/rendering"))  
sdf\_len(sc, 628, repartition = 628) %>% spark\_apply(function(idx, scene) {  
 render <- sprintf("%04d.png", idx$id)  
 rayrender::render\_scene(scene, width = 1920, height = 1080, lookfrom = c(12 \*   
 sin(idx$id/100), 5, 12 \* cos(idx$id/100)), filename = render)  
 system2("hadoop", args = c("fs", "-put", render, "/user/hadoop/rendering/"))  
}, context = scene, columns = list()) %>% collect()

完成所有的图像渲染后，最后一步是从HDFS中获得结果，并使用诸如ffmpeg的工具把单个图像转化成动画：

hadoop fs -get rendering/  
ffmpeg -s 1920x1080 -i rendering/%d.png -vcodec libx264 -crf 25  
 -pix\_fmt yuv420p rendering.mp4

本例假设HDFS用作Spark的存储技术，并在hadoop用户下运行，你需要根据特定的存储或用户进行调整。

我们已经介绍了spark\_apply()的一些常见用例，但是你还要针对具体的需求找出其它用例。下一节将介绍技术概念，达到创建其它用例和有效使用spark\_apply()的目的。

## 分区

大多数使用dplyr分析数据的任务或使用MLlib建模的Spark操作不需要了解Spark如何划分数据；它们是自动完成的。然而，对于分布式R计算，情况并非如此。要掌握这些内容，你必须学习和理解Spark是如何对数据进行分区的，并提供与之兼容的转换。这是必需的，因为spark\_apply()接受每个分区并允许执行任何转换，而不是整个数据集。你可以使用第9章中的图表和示例回忆分区和转换等概念。

为了帮助理解spark\_apply()中如何分区，可以学习以下代码：

sdf\_len(sc, 10) %>%  
spark\_apply(~nrow(.x))  
# Source: spark<?> [?? x 1]  
 result  
\* <int>  
1 5  
2 5

我们应该期望输出是总行数吗？从结果中可以看出，答案通常是否定的；Spark假设数据分布在多台机器上，因此你经常会发现它已经分好区了，即使是对于小数据集也是如此。因为我们不应该让spark\_apply()在单个分区上操作，所以让我们看看sdf\_len(sc, 10)包含多少个分区：

sdf\_len(sc, 10) %>% sdf\_num\_partitions()  
[1] 2

这解释了为什么在spark\_apply()下通过nrow()计数行会得到两行结果，因为有两个分区，而不是一个。spark\_apply()获取每个分区上的行数，每个分区包含5行，而不是所期望的那样总共10行。

对于这个特定的示例，我们可以通过重新分区来进一步聚合这些分区，然后将其相加，这类似于使用spark\_apply()的简单MapReduce操作：

sdf\_len(sc, 10) %>%  
spark\_apply(~nrow(.x)) %>%  
sdf\_repartition(1) %>%  
spark\_apply(~sum(.x))  
# Source: spark<?> [?? x 1]  
 result  
\* <int>  
1 10

因此，既然你知道在分区上使用spark\_apply()，我们会继续使用group\_by来控制分区。

## 分组

使用spark\_apply()时，我们可以从spark请求显式分区。例如，如果我们必须在一个分区中处理小于4的数字，而在另个分区中处理其余的数字，我们可以显式地创建这些组，然后请求spark\_apply()使用两个分区：

sdf\_len(sc, 10) %>%  
transmute(groups = id < 4) %>%  
spark\_apply(~nrow(.x), group\_by = "groups")  
# Source: spark<?> [?? x 2]  
 groups result  
\* <lgl> <int>  
1 TRUE 3  
2 FALSE 7

注意，spark\_apply()还是处理两个分区，但在本例中，这是我们期望的，因为我们在spark\_apply() 中显式地请求了它们；因此，你可以安全地将结果理解为“有3个整数小于4”。

只能对放在单个计算机的数据按照分区进行分组；如果其中一个组太大，会抛出异常。要在超过单个节点资源的组上执行操作，可以考虑分区为较小的单元，或者使用dplyr::do，它目前已针对大型分区进行了优化。

本节的要点是在处理spark\_apply()时始终考虑分区。接下来，我们会详细研究spark\_apply()，探索列是如何被理解的。

## 列

spark\_apply()默认自动检查生成中的数据框，以便发现列名和数据类型。例如：

sdf\_len(sc, 1) %>%  
spark\_apply(~ data.frame(numbers = 1, names = "abc"))  
# Source: spark<?> [?? x 2]  
 numbers names  
\* <dbl> <chr>  
1 1 abc

但是，这样做很低效，因为spark\_apply()需要运行两次：首先，通过对所有数据的子集计算spark\_apply()来查找列，然后计算实际所需的值。

为了提高性能，可以通过columns参数显式指定列。这个参数接受最终数据框中期望的类型的命名列表。然后，我们可以通过给numbers列指定正确的类型，重写之前的示例，并只运行一次spark\_apply()：

sdf\_len(sc, 1) %>%  
spark\_apply( ~ data.frame(numbers = 1, names = "abc"),  
columns = list(numbers = "double", names = "character"))  
# Source: spark<?> [?? x 2]  
 numbers names  
\* <dbl> <chr>  
1 1 abc

我们已经介绍了行和列如何与spark\_apply()进行交互，接下来我们将继续使用Context信息，它们有时在处理分布式数据集时需要。

## Context

如果要使用spark\_apply()处理分区，可能需要包括足以放入每个节点的小型辅助数据。网格搜索的例子中就是这样，数据集被传递到所有分区，并且自身保持未分区状态。

我们可以在本章中修改初始的f(x) = 10 \* x示例来定制乘法器。它最初设置为10，但我们可以在context参数中指定，完成配置：

sdf\_len(sc, 4) %>%  
spark\_apply(  
function(data, context) context \* data,  
context = 100  
)  
# Source: spark<?> [?? x 1]  
 id  
 <dbl>  
1 100  
2 200  
3 300  
4 400

图11-5展示了这个例子的概念图。注意到数据分区还是可用的，但是Context参数分配给了所有节点。

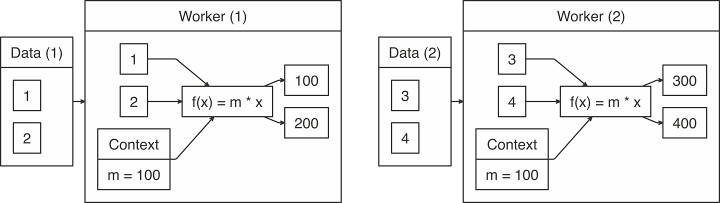


图11-5. 乘以context参数的Map操作

网格搜索示例使用这个参数将数据框传递给每个工作节点；但是，由于context参数被序列化为R对象，因此它可以包含任何内容。例如，如果需要传递多个值甚至多个数据集，你甚至可以传递包含值的列表。

下面的例子定义了一个f(x) = m \* x + b函数，并运行m = 10和b = 2：

sdf\_len(sc, 4) %>%  
spark\_apply( ~.y$m \* .x + .y$b,  
context = list(b = 2, m = 10) )  
# Source: spark<?> [?? x 1]  
 id  
 <dbl>  
1 12  
2 22  
3 32  
4 42

注意，我们已将context重命名为.y以缩短变量名。代码跑通的原因是spark\_apply()假设context是函数和表达式中的第二个参数。

你会发现context参数非常有用；例如，下一节会介绍如何正确构造函数，并且context在高级示例中用于构造依赖于其它函数的函数。

## 函数

前面我们将spark\_apply()作为函数或表达式执行自定义转换的操作。在有关编程的文献中，带有上下文的函数也被称为*闭包*。

表达式可用于定义语句较短的转换，如~ 10 \* .x。对于表达式，.x包含分区而.y包含上下文信息（如果可用）。但是，我们很难为跨多行的复杂代码定义表达式。对于这些情况，函数更合适。

函数支持复杂的多行转换，并定义为function(data, context) {}，你可以在{}中提供任何代码。我们在前面的章节中看到过使用谷歌云将图片转换为图片标题的例子。

传递给spark\_apply()的函数使用serialize()进行序列化，serialize()被描述为“一个用于序列化连接的简单低层级接口”。serialize()的一个当前限制是它不会序列化在其环境之外引用的对象。例如，以下函数由于闭包引用了external\_value而报错：

external\_value <- 1  
spark\_apply(iris, function(e) e + external\_value)

作为解决这个限制的方法，你可以将闭包所需的函数添加到context中，然后将这些函数分配到全局环境中：

func\_a <- function() 40  
func\_b <- function() func\_a() + 1  
func\_c <- function() func\_b() + 1  
sdf\_len(sc, 1) %>% spark\_apply(function(df, context) {  
for (name in names(context)) assign(name, context[[name]], envir = .GlobalEnv)  
func\_c()  
}, context = list(  
func\_a = func\_a,  
func\_b = func\_b,  
func\_c = func\_c  
))  
# Source: spark<?> [?? x 1]  
 result  
 <dbl>  
1 42

当这个方法不可行时，你还可以为所需的功能创建自己的R程序包，然后在spark\_apply()中使用该包。

你已经使用简单的R代码学习了spark\_apply()中提供的所有功能。在下一节中，我们会介绍如何在分发计算时使用包。在创建有用的转换时，R程序包是必不可少的。

## 程序包

通过spark\_apply()你可以在Spark中使用任何R程序包。例如，你可以用broom程序包创建线性回归模型输出的整洁的数据框结果：

spark\_apply(  
iris,  
function(e) broom::tidy(lm(Petal\_Length ~ Petal\_Width, e)),  
names = c("term", "estimate", "std.error", "statistic", "p.value"),  
group\_by = "Species")  
# Source: spark<?> [?? x 6]  
 Species term estimate std.error statistic p.value  
 <chr> <chr> <dbl> <dbl> <dbl> <dbl>  
1 versicolor (Intercept) 1.78 0.284 6.28 9.48e- 8  
2 versicolor Petal\_Width 1.87 0.212 8.83 1.27e-11  
3 virginica (Intercept) 4.24 0.561 7.56 1.04e- 9  
4 virginica Petal\_Width 0.647 0.275 2.36 2.25e- 2  
5 setosa (Intercept) 1.33 0.0600 22.1 7.68e-27  
6 setosa Petal\_Width 0.546 0.224 2.44 1.86e- 2

第一次调用spark\_apply()时，本地.libPaths()中的所有内容（包含所有R包）都会复制到每个Spark工作节点中。程序包只复制一次，并且只要连接一直打开，包就一直存在。R程序包大小为几十亿字节很常见，因此在将R包复制到Spark集群时，请做好单次资源调用的准备。你可以通过设置packages = FALSE来禁用程序包分发。

由于在spark\_connect()连接期间只复制一次包，因此在连接处于活动状态时不支持安装其它包。因此，如果需要安装新程序包，请用spark\_connect()断开连接，修改软件包，然后重新连接。此外，R程序包不在本地模式下复制，因为程序包已经存在于本地系统上。

尽管本节很简短，但是使用分布式R代码的程序包打开了一个全新而有趣的用例领域。本章介绍了其中一些用例，通过找寻当今的丰富的R包生态系统，你会发现更多的用例。

本节完成了用程序包分发R代码所需功能的讨论。现在，我们讨论一下集群使用spark\_apply()时的一些需求。

## 集群需求

前面章节中介绍的功能不需要特殊配置Spark集群。只要你有一个正确配置的Spark集群，就可以使用R。但是，对于下面提供的功能，你的群集管理员、云提供商， 或者你自己必须通过安装以下任一项来配置群集：

* 每个节点中的R，以便在集群中执行R代码
* 当使用Spark 2.3或更高版本时，每个节点中的Apache Arrow（Arrow提供了性能改进，使分布式R代码更接近原生Scala代码）

让我们看看每个需求，保证你可以正确考虑它们提供的利弊或益处。

### 安装R

对于第一个需求，R运行时应该预先安装在集群中的每个节点中；这是针对spark\_apply()的需求。

如果未能在每个节点中安装R，在尝试使用spark\_apply()时会触发一个Cannot run program, no such file or directory的报错。

请与群集管理员联系，保证整个群集中的R运行时可用。如果已经安装了R，可以指定要与spark.r.command配置一起使用的安装路径，如下所示：

config <- spark\_config()  
config["spark.r.command"] <- "<path-to-r-version>"  
sc <- spark\_connect(master = "local", config = config)  
sdf\_len(sc, 10) %>% spark\_apply(function(e) e)

由于驱动节点会向工作节点分发包，也可能编译包，因此我们需要同构群集。例如，驱动节点和工作节点必须具有相同的处理器体系结构、系统库等。这通常是大多数集群的情况，但是你的集群可能不是这样。

不同的集群管理器、Spark发行版和云提供商支持不同的解决方案，以便在集群中的每个节点上安装额外的软件（如R）。在每个工作节点上安装R时，请遵循说明。这里有一些示例：

* Spark单机版 需要连接到每台计算机并安装R；pssh之类的工具允许你面对多台计算机只运行一条安装命。
* Cloudera 提供了一个R包（请参阅Cloudera的博客文章“How to Distribute Your R code with sparklyr and Cloudera Data Science Workbench”），可以在每个工作节点上启用R。
* 亚马逊EMR 在启动EMR集群时预装了R，如“Amazon”第104页提到的。
* 微软HDInsight R是预装的，不需要其它步骤。
* Livy Livy连接不支持分发包，因为预编译库的客户机可能没有与集群计算机相同的处理器体系结构或操作系统。

严格地说，这些步骤完成了集群的最后一个需求。但是，我们强烈建议你使用Apache Arrow和spark\_apply()来以最小的开销支持大规模计算。

### Apache Arrow

在介绍Apache Arrow之前，我们会讨论一下如何在Spark和R之间存储和传输数据。由于R一开始的设计目的是执行快速的数值计算，因此找出存储数据的最佳方法非常重要。

一些计算系统在内部按行存储数据；然而，大多数有意义的数值运算要求按列处理数据。例如，计算列的平均值需要单独处理每一列，而不是整行。Spark默认按行存储数据，因为它更容易分区；但是，R按列存储数据。因此，当数据在Spark和R之间传输时，需要对这两种表示进行转换，如图11-6所示。

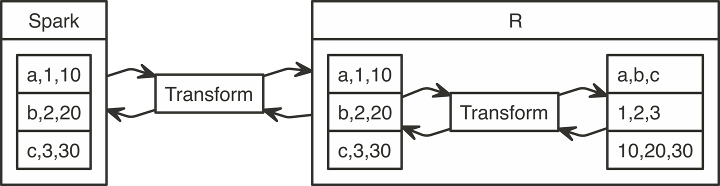


图11-6. Spark和R之间的数据转换

每个分区都需要进行从行到列的转换。此外，数据还必须从Scala的内部表示转换为R的内部表示。Apache Arrow减少了这些浪费大量CPU周期的转换。

Apache Arrow是一个跨语言的内存数据开发平台。在Spark中，它通过定义与许多编程语言兼容的通用数据格式来加速Scala和R之间的数据传输。我们不必在Scala的内部表示和R之间进行转换，而是对这两种语言使用相同的结构。此外，Spark并行地将数据从基于行的存储转换为基于列的存储，这可以通过第8章中介绍的列存储格式进一步优化。改进后的转换如图11-7所示。

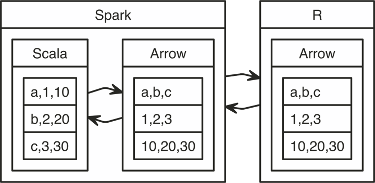


图11-7. Spark和R之间使用的Apache Arrow数据转换

虽然Apache Arrow不是必需的，但是还是强烈建议你在使用spark\_apply()时使用它。它支持Spark2.3.0以后的版本；但是，它要求系统管理员在每个节点中安装Apache Arrow运行时（请参见http://Arrow.Apache.org/install/）。

此外，要将Apache Arrow与sparklyr一起使用，还需要安装arrow程序包：

install.packages("arrow")

使用arrow之前，让我首先挑选一个验证指标：

system.time(  
sdf\_len(sc, 10^4) %>% spark\_apply(nrow) %>% collect()  
)  
 user system elapsed  
 0.240 0.020 7.957

在我们的系统中，处理10000行大约需要8秒。要启用Arrow，只需像往常一样导入库并使用spark\_apply()。让我们测一下spark\_apply()处理100万行需要多长时间：

library(arrow)  
system.time(  
sdf\_len(sc, 10^6) %>% spark\_apply(nrow) %>% collect()  
)  
 user system elapsed  
 0.317 0.021 3.922

在我们的系统中，Apache Arrow可以在一半的时间内处理100倍的数据：仅耗时4秒。

arrow中的大多数功能只是在后台工作，可以提高性能和数据序列化；但是，有一个设置需要注意。spark.sql.execution.arrow.maxRecordsPerBatch配置指定每个arrow数据传输的默认大小。它与其它Spark组件共享，默认为10000行：

218 | Chapter 11: Distributed R  
library(arrow)  
sdf\_len(sc, 2 \* 10^4) %>% spark\_apply(nrow)  
# Source: spark<?> [?? x 1]  
 result  
 <int>  
1 10000  
2 10000

你可能需要根据系统能够处理的数据量调整这个数字，使其对于大型数据集更小，或者对于需要同时处理记录的操作更大。我们可以将此设置更改为5000行，并验证分区是否也正确更改：

config <- spark\_config()  
config$spark.sql.execution.arrow.maxRecordsPerBatch <- 5 \* 10^3  
sc <- spark\_connect(master = "local", config = config)  
sdf\_len(sc, 2 \* 10^4) %>% spark\_apply(nrow)  
# Source: spark<?> [?? x 1]  
 result  
 <int>  
1 5000  
2 5000  
3 5000  
4 5000

到目前为止，我们已经介绍了用例、主要操作和集群需求。现在我们将讨论在分发R代码时有用的故障排除技术。

## 故障排除

一个定制化的转换可能因为各种原因而失败。要了解如何排除报错，让我们自己模拟一个导致报错的问题：

sdf\_len(sc, 1) %>% spark\_apply(~stop("force an error"))  
Error in force(code) :  
 `sparklyr` worker rscript failure, check worker logs for details  
 Log: wm\_bx4cn70s6h0r5vgsldm0000gn/T/Rtmpob83LD/file2aac1a6188\_spark.log  
---- Output Log ----  
19/03/11 14:12:24 INFO sparklyr: Worker (1) completed wait using lock for RScript

注意，错误信息提到要检查日志。在以本地模式运行时，你可以只运行以下代码：

spark\_log(sc, filter = "terminated unexpectedly")  
19/03/11 14:12:24 ERROR sparklyr: RScript (1) terminated unexpectedly:  
 force an error

这指向我们提到的人为stop("force an error")错误。但是，如果不在本地模式下工作，必须从集群管理器中获取工作日志。由于这个过程可能很麻烦，一种替代方法是重新运行spark\_apply()，自己返回错误消息：

sdf\_len(sc, 1) %>% spark\_apply(~tryCatch(  
stop("force an error"),  
error = function(e) e$message  
))  
# Source: spark<?> [?? x 1]  
 result  
 <chr>  
1 force an error

除其它适用于spark\_apply()的更高级故障排除技术外，以下各节按顺序介绍这些技术。你应该首先尝试使用工作日志进行故障排除，然后识别分区错误，最后尝试调试工作节点。

### 工作节点日志

每当执行spark\_apply()时，有关执行的信息都会写入每个工作节点。你可以使用这个日志编写自定义消息，以帮助诊断和调试代码。

例如，假设我们不知道df的第一个列名是什么。我们可以使用worker\_log()编写在工作节点执行的自定义日志消息，如下所示：

sdf\_len(sc, 1) %>% spark\_apply(function(df) {  
worker\_log("the first column in the data frame is named ", names(df)[[1]])  
df  
})  
# Source: spark<?> [?? x 1]  
 id  
\* <int>  
1 1

当在本地运行的时候，我们可以过滤工作节点的日志记录，如下所示：

spark\_log(sc, filter = "sparklyr: RScript")  
18/12/18 11:33:47 INFO sparklyr: RScript (3513) the first column in  
the dataframe is named id  
18/12/18 11:33:47 INFO sparklyr: RScript (3513) computed closure  
18/12/18 11:33:47 INFO sparklyr: RScript (3513) updating 1 rows  
18/12/18 11:33:47 INFO sparklyr: RScript (3513) updated 1 rows  
18/12/18 11:33:47 INFO sparklyr: RScript (3513) finished apply  
18/12/18 11:33:47 INFO sparklyr: RScript (3513) finished

注意，日志会打印我们自定义日志记录，展示的id是给定数据框中第一列的名称。

此功能在g故障排除时非常有用；例如，如果使用stop()函数强制执行错误：

sdf\_len(sc, 1) %>% spark\_apply(function(df) {  
 stop("force an error")  
})

我们会得到如下的错误：

Error in force(code) :  
 `sparklyr` worker rscript failure, check worker logs for details

根据错误提示的，我们可以查看工作节点的日志，获取如下的具体错误信息：

spark\_log(sc)

这些信息会展示包含错误和回调的记录：

18/12/18 11:26:47 INFO sparklyr: RScript (1860) computing closure  
18/12/18 11:26:47 ERROR sparklyr: RScript (1860) terminated unexpectedly:  
 force an error  
18/12/18 11:26:47 ERROR sparklyr: RScript (1860) collected callstack:  
11: stop("force and error")  
10: (function (df)  
{  
 stop("force and error")  
})(structure(list(id = 1L), class = "data.frame", row.names = c(NA,  
-1L)))

注意，spark\_log(sc)只在使用本地集群时检索工作日志。当在具有多台计算机的集群中运行时，必须使用集群管理器提供的工具和用户界面来查找这些日志条目。

### 解决超时

当你运行几百个执行器时，某些任务可能会无限期挂起。在这种情况下，工作中的大多数任务都会成功完成，但其中有少数任务仍在运行，不会失败或成功。

假设你需要计算许多网页的大小。你可以在类似如下的内容上使用spark\_apply()：

sdf\_len(sc, 3, repartition = 3) %>% spark\_apply(~download.file("https://google.com",   
 "index.html") + file.size("index.html"))

有些网页可能不存在或下载时间过长。在这种情况下，大多数任务都会成功，但少数任务会挂起。为了防止这几个任务阻塞所有计算，可以使用spark.speculation设置。启用这个设置后，一旦75%的任务成功，Spark将查找比任务执行时间中位数更长的任务，然后重试。你可以使用spark.speculation.multiplier设置时间倍数，来确定任务是否运行缓慢。

因此，在本例中，你可以配置比这个中位数长四倍的任务，让Spark重试，如下所示：

config <- spark\_config()  
config["spark.speculation"] <- TRUE  
config["spark.speculation.multiplier"] <- 4

### 检查分区

如果一个特定的分区失败了，你可以通过计算摘要信息和检索具体的分区来检测损坏的分区。通常，你可以在连接到Spark之前从CRAN安装digest：

sdf\_len(sc, 3) %>% spark\_apply(function(x) {  
 worker\_log("processing ", digest::digest(x), " partition")  
 # your code  
 x  
})

这个操作会添加一个如下的记录：

18/11/03 14:48:32 INFO sparklyr: RScript (2566)  
 processing f35b1c321df0162e3f914adfb70b5416 partition

在集群中执行此操作时，可以在日志中查找未完成的任务。一旦找到了摘要信息，你就可以取消这个作业。然后，可以使用摘要检索具体的R数据帧框，类似如下：

sdf\_len(sc, 3) %>% spark\_apply(function(x) {  
if (identical(digest::digest(x),  
"f35b1c321df0162e3f914adfb70b5416")) x else x[0,]  
}) %>% collect()  
# A tibble: 1 x 1  
 result  
 <int>  
1 1

你可以在R中运行这些代码，进一步排查故障。

### 调试工作节点

调试器是一种工具，允许你逐行执行代码；你可以使用它来解决本地连接的spark\_apply()问题。可以使用debug参数在调试模式下启动spark\_apply()，然后按照说明进行操作：

sdf\_len(sc, 1) %>% spark\_apply(function() {  
stop("Error!")  
}, debug = TRUE)  
Debugging spark\_apply(), connect to worker debugging session as follows:  
 1. Find the workers <sessionid> and <port> in the worker logs, from RStudio  
 click 'Log' under the connection, look for the last entry with contents:  
 'Session (<sessionid>) is waiting for `sparklyr` client to connect to  
 port <port>'  
 2. From a new R session run:  
 debugonce(sparklyr:::spark\_worker\_main)  
 sparklyr:::spark\_worker\_main(<sessionid>, <port>)

如这些说明所示，你需要从不同的R进程中连接“as the worker node”，然后逐步完成代码。这个方法并没有以前的方法直接，因为你还需要查看一些sparklyr代码。因此，我们只建议你最后使用这个方法。（你也可以尝试第2章中描述的在线资源。）

现在，让我们在这一章结束时简要回顾一下我们介绍的功能。

## 小结

本章介绍了spark\_apply()这一高级技术，你可以使用它来填补Spark或其许多扩展中的功能空白。我们给出了spark\_apply()解析数据、并行建模许多小数据集、执行网格搜索和调用web API的用例。你还了解到了分区与spark\_apply()的关系，以及如何创建自定义组、跨所有节点分发context信息，以及排除故障问题、功能局限和集群配置注意事项。我们还强烈建议在使用Spark和R时使用Apache Arrow作为库，并介绍了安装过程、用例和应该注意的事项。

到目前为止，我们只处理静态数据的大型数据集，这些数据集不会随时间变化，并且在我们分析、建模和可视化它们时保持不变。在第12章中，我们将介绍另一种处理数据集的技术，这些数据集除了很大之外，还像信息流一样增长。

# 第12章 数据流

我们的故事还没有结束。 —Arya Stark

回顾前几章，我们已经讨论了很多技术，但还不是全部技术。我们分析了表格数据集，对原始文本执行了无监督学习，分析了图和地理数据集，甚至用自定义的R代码转换了数据！那现在还需要学习什么？

在此之前我们一直假设数据是静态的，并且不会随着时间的推移而改变。尽管我们对此没有明确说明，d但是现在假设你的工作是分析交通模式，向交通部提出建议。一个合理的方法是分析历史数据，然后设计预测模型，在一夜之间完成预测。真的可以一夜之间完成吗？如果真的会这很有用，但是交通模式会随着时间甚至分钟的变化而变化。你可以预处理和预测的越来越快，但最终模型失效。因为你无法加载大型数据集、转换数据、对数据进行评分、卸载数据，并在每一秒重复此过程。

相反，我们需要引入一种不同的数据集，一种不是静态的而是动态的，类似于表但不断增长的数据集。我们将这些数据集称为*数据流*。

## 概述

我们知道如何处理大规模的静态数据集，但是我们如何推理大规模的实时数据集呢？具有无限条目数的数据集称为数据流。

对于静态数据集，如果我们使用预先训练的主题模型进行实时评分，记录就是文本行；对于实时数据集，我们会对无限行的文本执行相同的评分操作。在实践中，你其实永远不会处理无限多的记录。你最终会终止这条数据，何况这个世界也会终结。这两个结果哪个先到还不知道呢。不管怎样，将数据集看成无限数据集让我们更容易对它们进行推理。

数据流在处理实时数据时最常用的技术，例如分析Twitter feed或股票价格。这两个示例都有定义良好的列，如“tweet”或“price”，但总是有新的数据行要分析。

*Spark streaming*提供可扩展和容错的数据流处理。这意味着你可以使用多台计算机来处理多个流数据源，与其它流或静态源连接，并在保证至少从故障中恢复一次（每条消息都会被传递，但可能会传递多次）。

在Spark中，通过定义源、转换和sink来创建流；可以将这些步骤看作是读取、转换和编写数据流，如图12-1所示。

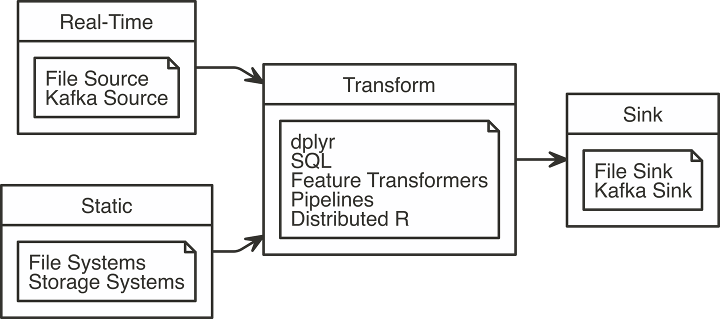


图12-1. 使用Spark Streaming

让我们具体看一下这些概念：

* 读取 数据流使用stream\_read\_\*()函数读取数据；读取操作定义流的源。你可以定义一个或多个读取的源。
* 转换 数据流可以使用dplyr、SQL、特征变换、评分管道或分布式R代码执行一个或多个转换。转换不仅可以应用于一个或多个流，还可以用于流和静态数据源的组合；例如，用spark\_read\_()函数加载到Spark中的转换意味着你可以轻松地组合静态数据和实时数据源。
* 写入 写操作是通过流的一系列stream\_write\_\*()函数来执行，而读取操作定义了流的sink。你可以指定一个或多个sink来写入数据。

你可以用几种不同的文件格式读写数据流：CSV、JSON、Parquet、行优化列式文件（ORC）和文本（见表12-1）。你也可以读写Kafka，我们稍后会介绍。

表12-1. 读写数据流的Spark函数

格式| 读取 |写入 CSV |stream\_read\_csv |stream\_write\_csv JSON |stream\_read\_json |stream\_write\_json Kafka |stream\_read\_kafka |stream\_write\_kafka ORC |stream\_read\_orc |stream\_write\_orc Parquet| stream\_read\_parquet |stream\_write\_parquet 文本| stream\_read\_text |stream\_write\_text 内存| |stream\_write\_memory

由于转换步骤是可选的，我们可以定义的最简单的流是在数据源和目标地址之间连续复制文本文件的数据流。

首先，使用install.packages("future")安装future包并连接到Spark。

因为数据流需要数据源，所以创建一个\_source\_文件夹：

dir.create("source")

现在，我们可以定义第一个数据流！

stream <- stream\_read\_text(sc, "source/") %>% stream\_write\_text("destination/")

数据流开始以stream\_write\_\*()运行；一旦执行，数据流将监视source路径，并在数据到达时处理至destination/路径。

我们可以使用stream\_generate\_test()每秒生成一个文件，其中包含遵循给定分布的文本行；你可以在附录A中阅读更多有关此内容的信息。

在实践中，你将连接到现有的源，而不必人工生成数据。然后，我们可以使用view\_stream()跟踪数据源和目标路径中每秒处理的行数（rps）及其随时间变化的最新值：

future::future(stream\_generate\_test(interval = 0.5))  
stream\_view(stream)

结果如图12-2所示。

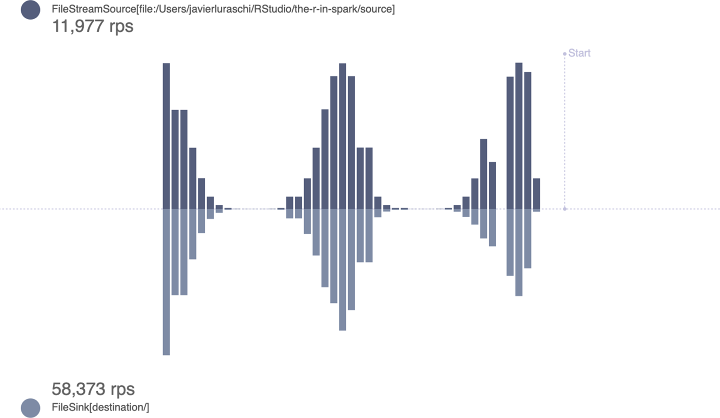


图12-2. 监控记录行服从二项分布的的数据流

注意，目标流中的rps速率高于数据源流中的rps速率。由于Spark测量来自数据源流的传入速率，而且还测量目标流中的实际行处理次数，因此这是符合预期的，也是期望的。例如，如果每秒向source/路径写入10行，则传入速率为10 rps。但是，如果Spark只需要0.01秒就可以写入这10行，那么输出速率是100 rps。

使用stream\_stop()停止处理流中的数据：

stream\_stop(stream)

上述练习介绍了如何轻松启动Spark流，并基于模拟的数据流读取和写入数据。下面让我们通过转换做一些比复制数据更有趣的事情。

## 转换

在实际场景中，数据流中传入的数据不会按原样输出。Spark Streaming作业会对数据进行转换，然后写入转换后的数据。数据流可以使用dplyr、SQL查询、ML管道或R代码进行转换。我们可以根据需要使用尽可能多的转换，就像Spark数据框可以使用sparklyr进行转换一样。

转换的数据源可以是数据流，也可以是数据框，但输出始终是数据流。如果需要，可以始终从目标流中获取快照，然后将保存输出为数据框。如果未指定目标流，sparklyr会负责这些事情。

下面的每个小节都介绍了sparklyr提供的在数据流上执行转换的选项。

### 分析

你可以使用dplyr操作和DBI的SQL分析数据流。作为一个快速示例，我们会在数据流上筛选行和添加列。我们不会显式调用stream\_generate\_test()，但是如果你觉得需要，还可以自己调用，核实数据正在持续处理：

library(dplyr)  
stream\_read\_csv(sc, "source") %>%  
filter(x > 700) %>%  
mutate(y = round(x / 100))  
# Source: spark<?> [inf x 2]  
 x y  
 <int> <dbl>  
 1 701 7  
 2 702 7  
 3 703 7  
 4 704 7  
 5 705 7  
 6 706 7  
 7 707 7  
 8 708 7  
 9 709 7  
10 710 7  
# … with more rows

你还可以在数据流的整个历史记录中执行聚合。历史记录可以经过筛选，也可以不筛选：

stream\_read\_csv(sc, "source") %>%  
filter(x > 700) %>%  
mutate(y = round(x / 100)) %>%  
count(y)  
# Source: spark<?> [inf x 2]  
 y n  
 <dbl> <dbl>  
1 8 25902  
2 9 25902  
3 10 13210  
4 7 12692

数据流中最新数据的分组聚合需要时间戳。时间戳会显示读取函数（在本例中为stream\_read\_csv()）第一次“看到”该特定记录的时间。在Spark Streaming术语中，时间戳称为*水印*。spark\_watermark()会添加时间戳。在本例中，所有记录的水印都是相同的，因为五个文件在创建后都被数据流读取。注意，只有Kafka和内存输出支持水印：

stream\_read\_csv(sc, "source") %>%  
stream\_watermark()  
# Source: spark<?> [inf x 2]  
 x timestamp  
 <int> <dttm>  
 1 276 2019-06-30 07:14:21  
 2 277 2019-06-30 07:14:21  
 3 278 2019-06-30 07:14:21  
 4 279 2019-06-30 07:14:21  
 5 280 2019-06-30 07:14:21  
 6 281 2019-06-30 07:14:21  
 7 282 2019-06-30 07:14:21  
 8 283 2019-06-30 07:14:21  
 9 284 2019-06-30 07:14:21  
10 285 2019-06-30 07:14:21  
# … with more rows

创建水印后，可以在group\_by()操作中使用它。然后，可以将其导入summarise()函数以获取数据流的一些统计信息：

stream\_read\_csv(sc, "source") %>%  
stream\_watermark() %>%  
group\_by(timestamp) %>%  
summarise(  
max\_x = max(x, na.rm = TRUE),  
min\_x = min(x, na.rm = TRUE),  
count = n()  
)  
# Source: spark<?> [inf x 4]  
 timestamp max\_x min\_x count  
 <dttm> <int> <int> <dbl>  
1 2019-06-30 07:14:55 1000 1 259332

### 建模

Spark数据流目前不支持实时数据集训练。除了技术上的挑战，即使有可能，也很难训练模型，因为模型本身需要随着时间的推移进行调整。这项技术叫做*在线学习*，这也许是Spark未来支持的技术。

尽管如此，还有一下其它的建模概念可以与数据流一起使用，如特征变换器和评分。让我们用数据流尝试一个特征转换器，并将评分留给下一节，因为我们会需要训练一个模型。

下一个示例使用ft\_bucketizer()特征转换器修改数据流，然后是常规的dplyr函数，你可以像使用静态数据集一样使用这些函数：

stream\_read\_csv(sc, "source") %>%  
mutate(x = as.numeric(x)) %>%  
ft\_bucketizer("x", "buckets", splits = 0:10 \* 100) %>%  
count(buckets) %>%  
arrange(buckets)  
# Source: spark<?> [inf x 2]  
# Ordered by: buckets  
 buckets n  
 <dbl> <dbl>  
 1 0 25747  
 2 1 26008  
 3 2 25992  
 4 3 25908  
 5 4 25905  
 6 5 25903  
 7 6 25904  
 8 7 25901  
 9 8 25902  
10 9 26162

### 管道

Spark管道可以用来对数据流进行评分，但不能用来训练数据流。前一个功能已经完全支持，后一个是Spark社区正在积极开发的特性。

要给数据流评分，必须首先创建我们的模型。因此，让我们构建、调整并保存一个简单的管道：

cars <- copy\_to(sc, mtcars)  
model <- ml\_pipeline(sc) %>% ft\_binarizer("mpg", "over\_30", 30) %>% ft\_r\_formula(over\_30 ~   
 wt) %>% ml\_logistic\_regression() %>% ml\_fit(cars)

如果你愿意，可以使用第5章中介绍的其它概念，例如对数据流进行评分之前，通过ml\_save()和ml\_load()保存和重新加载管道。

然后我们可以使用stream\_generate\_test()生成一个基于mtcars的数据流，并使用ml\_transform()给模型评分:，

future::future(stream\_generate\_test(mtcars, "cars-stream", iterations = 5))  
ml\_transform(model, stream\_read\_csv(sc, "cars-stream"))  
# Source: spark<?> [inf x 17]  
 mpg cyl disp hp drat wt qsec vs am gear carb over\_30  
 <dbl> <int> <dbl> <int> <dbl> <dbl> <dbl> <int> <int> <int> <int> <dbl>  
 1 15.5 8 318 150 2.76 3.52 16.9 0 0 3 2 0  
 2 15.2 8 304 150 3.15 3.44 17.3 0 0 3 2 0  
 3 13.3 8 350 245 3.73 3.84 15.4 0 0 3 4 0  
 4 19.2 8 400 175 3.08 3.84 17.0 0 0 3 2 0  
 5 27.3 4 79 66 4.08 1.94 18.9 1 1 4 1 0  
 6 26 4 120. 91 4.43 2.14 16.7 0 1 5 2 0  
 7 30.4 4 95.1 113 3.77 1.51 16.9 1 1 5 2 1  
 8 15.8 8 351 264 4.22 3.17 14.5 0 1 5 4 0  
 9 19.7 6 145 175 3.62 2.77 15.5 0 1 5 6 0  
10 15 8 301 335 3.54 3.57 14.6 0 1 5 8 0  
# … with more rows, and 5 more variables: features <list>, label <dbl>,  
# rawPrediction <list>, probability <list>, prediction <dbl>

尽管这个例子是几行代码组合在一起的，但是这一刚刚完成的工作实际上令人印象深刻。你把数据复制到Spark中，执行特征工程，训练模型，并在实时数据集上对模型进行评分，只需7行代码！现在让我们尝试实时环境下的自定义转换。

### 分布式R

你也可以借助spark\_apply()使用任意R代码来转换数据流。此方法遵循第11章中讨论的相同原则，spark\_apply()在集群中存储数据的每个执行器上运行R代码。这样可以处理高吞吐量数据流并满足低延迟需求：

stream\_read\_csv(sc, "cars-stream") %>% select(mpg) %>% spark\_apply(~round(.x), mpg = "integer") %>%   
 stream\_write\_csv("cars-round")

正如可以预见的，它通过运行自定义的round()函数将来自cars-stream的数据处理成cars-round。让我们看看输出sink：

spark\_read\_csv(sc, "cars-round")  
# Source: spark<carsround> [?? x 1]  
 mpg  
 <dbl>  
 1 16  
 2 15  
 3 13  
 4 19  
 5 27  
 6 26  
 7 30  
 8 16  
 9 20  
10 15  
# … with more rows

同样，在使用数据流时，请确保使用已经熟悉的有关spark\_apply()的概念；例如，你应该考虑使用arrow来显著提高性能。在我们继续介绍之前，断开与Spark的连接：

spark\_disconnect(sc)

这是数据流转换的最后内容。我们会学到如何通过Kafka使用Spark Streaming。

## Kafka

Apache Kafka是领英开发的开源数据流处理软件平台，捐赠给了Apache软件基金会。它是用Scala和Java编写的。做一个类比的话，Kafka是实时存储，Hadoop是静态存储。

Kafka将数据流存储为记录，记录由键、值和时间戳组成。它可以按主题对包含不同信息的多个流进行分类，并处理这些数据流。Kafka通常用于连接多个实时应用程序。生产者是将数据流到Kafka的应用程序，而消费者是从Kafka读取数据的应用程序；在Kafka术语体系中，消费者应用程序*订阅*主题。 因此，我们可以用Kafka完成的最基本的工作流是，包含一个生产者和一个消费者的流程，如图12-3所示。



图12-3. 一个基本的Kafka工作流

如果你是Kafka新手，我们不建议你运行本节中的代码。但是，如果你非常想继续研究，那么首先需要按照附录A中的说明安装Kafka或将其部署到集群中。

使用Kafka还需要在连接到Spark时安装Kafka包。请确保在连接的config中指定此选项：

library(sparklyr)  
library(dplyr)  
sc <- spark\_connect(master = "local", version = "2.3", config = list(sparklyr.shell.packages = "org.apache.spark:spark-sql-kafka-0-10\_2.11:2.3.1"))

连接成功后，从数据流读取数据后就很直接了：

stream\_read\_kafka(sc, options = list(kafka.bootstrap.server = "host1:9092, host2:9092",   
 subscribe = "<topic-name>"))

但是，请注意，你需要正确配置options列表。kafka.bootstrap.server需要一个kafka主机列表，而topic和subscribe分别定义了在从Kafka写入或读取时应使用哪个主题。

尽管我们已经介绍了包含一个生产者和一个消费者简单用例，Kafka也支持更复杂的交互。接下来，我们从一个主题中读取数据、处理数据，然后将结果写入另一个主题。属于同一主题的生产者和消费者的系统称为流处理器。在图12-4中，流处理器读取主题A，然后将结果写入主题B。这考虑到给定的消费者程序读取结果，而不是读取“原始”feed数据。

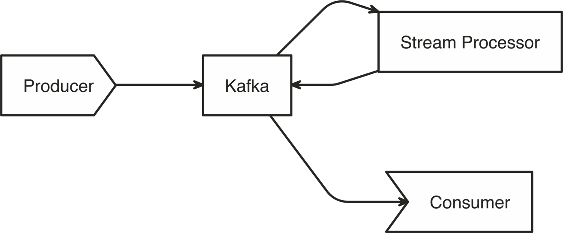


图12-4. 使用数据流处理器的Kafka工作流

在Spark中处理Kafka流时，有三种模式可用：complete，update和append。complete模式在每次有新的批量数据时为每组提供全部信息；update仅为最近批量数据中有更新的组提供全部信息；append将原始记录添加到目标主题。append模式不适用于聚合，但适用于将筛选的子集传递到目标主题。

在下一个示例中，生产者在letters主题下将随机字母流到Kafka中。然后，Spark将充当流处理器，读取字母主题并计算唯一的字母，然后在totals主题下将这些字母写回Kafka。写回Kafka时，我们将使用update模式；也就是说，只有发生改变的totals才会发送到Kafka。这个更改在letters主题的每批数据之后确定：

hosts <- "localhost:9092"  
read\_options <- list(kafka.bootstrap.servers = hosts, subscribe = "letters")  
write\_options <- list(kafka.bootstrap.servers = hosts, topic = "totals")  
stream\_read\_kafka(sc, options = read\_options) %>%  
mutate(value = as.character(value)) %>% # coerce into a character  
count(value) %>% # group and count letters  
mutate(value = paste0(value, "=", n)) %>% # kafka expects a value field  
stream\_write\_kafka(mode = "update",  
options = write\_options)

你可以读取Kafka，快速查看全部信息：

stream\_read\_kafka(sc, options = totals\_options)

通过新的终端会话，使用Kafka的命令行工具手动将单个字母添加到letters主题中：

kafka-console-producer.sh --broker-list localhost:9092 --topic letters  
>A  
>B  
>C

输入的字母被推到Kafka，被Spark读取，在Spark中聚合，然后返回给Kafka。最后，它们再次被Spark使用，提供有关totals主题简单信息。这是一个很好的流程，也是实时数据处理项目中常见的实际配置。

接下来，我们将使用Shiny框架来实时可视化数据流！

## Shiny

Shiny的reactive框架非常适合支持流式信息，你可以使用reactiveSpark()来显示来自Spark的实时数据。Shiny的知识远比我们在这里所能了解到的多得多。如果你已经熟悉Shiny，那么这个示例应该很容易理解。

我们有一个k-means的Shiny示例的修改版本，它不是从静态的iris数据集中获取数据，而是使用stream\_generate\_test()生成，被Spark使用，通过reactiveSpark()获取给Shiny，展示如图12-5。

要运行这个示例，可以把以下Shiny应用程序存储在shiny/shiny-stream.R下：

library(sparklyr)  
library(shiny)  
unlink("shiny-stream", recursive = TRUE)  
dir.create("shiny-stream", showWarnings = FALSE)  
sc <- spark\_connect(master = "local", version = "2.3", config = list(sparklyr.sanitize.column.names = FALSE))  
ui <- pageWithSidebar(headerPanel("Iris k-means clustering from Spark stream"), sidebarPanel(selectInput("xcol",   
 "X Variable", names(iris)), selectInput("ycol", "Y Variable", names(iris), selected = names(iris)[[2]]),   
 numericInput("clusters", "Cluster count", 3, min = 1, max = 9)), mainPanel(plotOutput("plot1")))  
server <- function(input, output, session) {  
 iris <- stream\_read\_csv(sc, "shiny-stream", columns = sapply(datasets::iris,   
 class)) %>% reactiveSpark()  
 selectedData <- reactive(iris()[, c(input$xcol, input$ycol)])  
 clusters <- reactive(kmeans(selectedData(), input$clusters))  
 output$plot1 <- renderPlot({  
 par(mar = c(5.1, 4.1, 0, 1))  
 plot(selectedData(), col = clusters()$cluster, pch = 20, cex = 3)  
 points(clusters()$centers, pch = 4, cex = 4, lwd = 4)  
 })  
}  
shinyApp(ui, server)

这个Shiny应用程序可以通过runApp()启动，类似于:

shiny::runApp("shiny/shiny-stream.R")

当Shiny应用程序运行时，会从同一目录启动一个新的R会话，并使用stream\_generate\_test()创建一个测试流。这会生成Spark可以处理和Shiny可以可视化的连续数据流（如图12-5所示）：

sparklyr::stream\_generate\_test(datasets::iris, "shiny/shiny-stream", rep(5, 10^3))

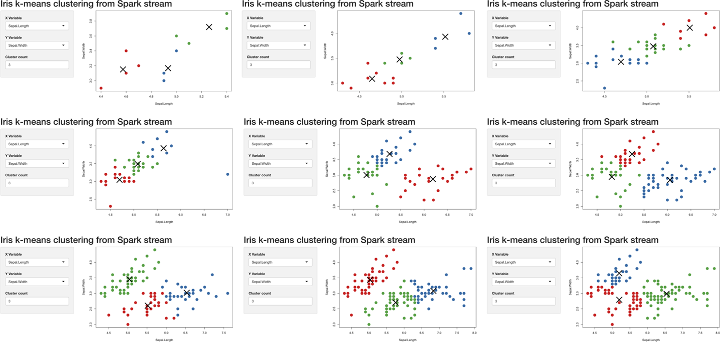


图12-5. 给Shiny程序加载数据的Spark reactive过程

在这一部分中，你了解了创建一个可用于多种不同目的（如监视和仪表板）的Shiny应用程序是非常容易的。

在更复杂的实现中，数据源可能是Kafka流。在离开之前，请断开与Spark的连接并清除我们用过的文件夹：

spark\_disconnect(sc)  
unlink(c("source", "destination", "cars-stream", "car-round", "shiny/shiny-stream"),   
 recursive = TRUE)

## 小结

从静态数据集到实时数据集，你已经真正掌握了许多大规模计算技术。具体来说，在本章中，你学习了将静态数据视为一个无限表的情况下，如何将其推广为实时数据。然后我们可以创建一个简单的数据流，无需任何数据转换的情况下，就可以将数据从A复制到B。

当你了解到不同的转换可以用于数据流时，这个简单的开始也很有用。例如，可以使用dplyr和DBI程序包完成数据分析，建模时引入的特征变换器转换、能够实时评分的成熟管道，以及使用自定义R代码转换数据集。当然，这些内容需要消化。

然后，我们将Apache Kafka作为一个可靠的、可扩展的实时数据解决方案。我们展示了如何通过消费者、生产者和主题来构建实时系统。如果将它们适当地组合在一起，就可以构建强大的体系架构来处理实时数据。

然后我们介绍了锦上添花的技术，如何在Shiny中使用Spark Streaming。由于数据流流可以转换为reactive类型（响应式编程领域的通用语言），因此其易用性是绝对让人惊喜。现在是时候进入我们的（非常简短的）最后一章了，第13章。我们会试图说服你使用新知识，让整个Spark和R社区的获益。

# 第13章 社区贡献

堵住门，堵住门。 —Hodor

在第12章中，我们提供了使用R处理Spark中大规模实时数据处理的工具。在最后一章中，我们将更少地关注学习，更多地关注回报Spark和R社区或你职业生涯中的同事。这真的需要整个社区来维持，所以我们需要你！

有很多方法可以做出贡献，从帮助社区成员和新开一个GitHub issue，到为自己、同事或R和Spark社区提供新功能。但是，这里我们只关注编写和共享扩展Spark的代码，以帮助其它人使用新的功能。你可以成为这些Spark扩展的作者。具体来说，你可以从零开始，了解什么是扩展、可以构建的不同的扩展类型、可用的构建工具，何时以及如何构建扩展。

你还会学习到如何利用Spark中提供的数百个扩展和Java中提供的数百万个易于在R中使用的组件。我们还会介绍如何构建使用Spark的Scala代码。你会看到，R是与其它语言如C++、SQL、Python等交互的伟大语言。因此，使用R中的Scala也就不足为奇了。它会遵循类似的实践，使R成为提供易用接口的理想选择。这些接口使数据处理变得高效，并且深受用户喜爱。

## 概述

当你考虑回馈更大的编码社区时，关于你所写的任何一段代码，你能问的最重要的问题是：这段代码对其它人有用吗？

让我们从本书中介绍的第一行也是最简单的一行代码开始。此代码用于加载简单的CSV文件：

spark\_read\_csv(sc, "cars.csv")

这个基础代码可能对其它人没有用处。但是，你可以将同样的示例调整为能够产生更多意义的内容，可能如下所示：

spark\_read\_csv(sc, "/path/that/is/hard/to/remember/data.csv")

这条代码与第一条很类似。但是如果你和其它人一起使用这个数据集，这条代码是有用的。有点意外！因为这意味着并非所有有用的代码都很高级或复杂。但是，为了让它对其它人有用，它确实需要以易于使用的格式打包、呈现和共享。

第一次使用时将其保存到teamdata.R文件中，并编写一个函数包装它：

load\_team\_data <- function() {  
 spark\_read\_text(sc, "/path/that/is/hard/to/remember/data.csv")  
}

这是一个改进，但需要用户一次又一次地手动共享此文件。幸运的是，这个问题很容易在R中通过R程序包解决。R程序包包含R代码，支持使用install.packages()函数安装。其中一个例子是sparklyr。还有许多其它R包可用；你也可以创建自己的R包。对于创建R包的新手，我们鼓励阅读Hadley Wickham的著作《R包》（O’Reilly）。创建R包允许你在组织中共享包文件，进而轻松地与其它人共享函数。

一旦你创建了一个包，有很多方法可以与同事或外界分享。例如，对于自己使用的包，可以考虑使用Drat或RStudio包管理器之类的产品。对于公开使用的R包，可以提供给CRAN（R综合档案网络）的R社区。

这些R包仓库允许用户通过install.packages("teamdata")安装程序包，而不必担心从何处下载包。它还允许其它包重用你的包。除了使用R包（如sparklyr、dplyr、broom和其它包）创建扩展Spark的新R包外，你还可以使用Spark API或Spark扩展中提供的所有功能，也可以编写自定义Scala代码。

例如，假设有一个类似于CSV但不完全相同的新文件格式。我们可能需要编写一个名为spark\_read\_file()的函数，该函数将接收这个新文件类型的路径，并在spark中读取它。一种方法是使用dplyr处理文本的每一行，或借助spark\_apply()的其它R包。另一种方法是使用Spark API访问Spark提供的方法。第三种方法是检查Spark社区中是否已经有人提供了支持这种新文件格式的Spark扩展。最后，你可以编写自己的定制Scala代码，使用任何Java库，包括Spark及其扩展。如图13-1所示。

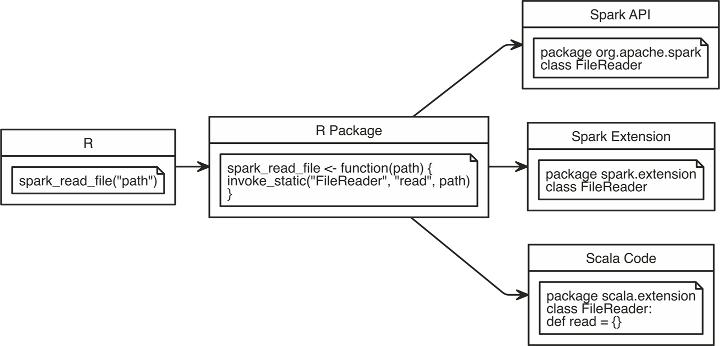


图13-1. 使用Spark API，Spark扩展，或编写Scala代码来扩展Spark

我们会首先关注使用Spark API扩展Spark，因为调用Spark API所需的技术也适用于调用Spark扩展或自定义Scala代码。

## Spark API

在介绍Spark API之前，让我们考虑一个简单而著名的问题。假设我们要计算一个分布式的、可能很大的文本文件，例如cars.csv中的行数，我们将其初始化如下：

library(sparklyr)  
library(dplyr)  
sc <- spark\_connect(master = "local")  
cars <- copy\_to(sc, mtcars)  
spark\_write\_csv(cars, "cars.csv")

现在，要计算文件中有多少行，我们可以运行如下代码：

spark\_read\_text(sc, "cars.csv") %>% count()  
# Source: spark<?> [?? x 1]  
 n  
 <dbl>  
1 33

很简单：我们使用spark\_read\_text()读取整个文本文件，然后使用dplyr的count()计数行。现在，假设spark\_read\_text()、dplyr或任何其它Spark功能都不可用。你会将怎么要求Spark计算cars.csv中的行数？如果在Scala中执行此操作，可以在Spark文档中找到答案。通过使用Spark API，可以按如下方式计算文件中的行数：

val textFile = spark.read.textFile("cars.csv")  
textFile.count()

因此，要使用R中Spark API的功能，比如spark.read.text文件，可以使用invoke()、invoke\_static()或invoke\_new()。（顾名思义，第一个从对象调用方法，第二个从静态对象调用方法，第三个创建新对象。）然后使用这些函数调用Spark的API并执行类似于Scala中提供的代码：

spark\_context(sc) %>%  
invoke("textFile", "cars.csv", 1L) %>%  
invoke("count")  
[1] 33

invoke()函数最初设计用于调用Spark代码，但它也可以调用Java中的任何代码。例如，我们可以使用以下代码创建一个Java BigInteger：

invoke\_new(sc, "java.math.BigInteger", "1000000000")  
<jobj[225]>  
java.math.BigInteger  
1000000000

如你所见，创建的对象不是一个R对象，而是一个Java对象。在R中，这个Java对象由spark\_jobj表示。这些对象应该与invoke()函数或spark\_dataframe()和spark\_connection()一起使用。spark\_dataframe()尽可能将spark\_jobj转换为Spark数据框，而spark\_connection()获取原始的Spark连接对象，这有助于避免在函数之间传递sc对象。

虽然在某些情况下调用Spark API可能很有用，但Spark中提供的大多数功能已经在sparklyr中得到了支持。因此，一个更有趣的扩展Spark的方法是使用许多现有的扩展。

## Spark扩展

在开始本节之前，请考虑导航到spark-packages.org，该网站跟踪Spark社区提供的Spark扩展。使用上一节中介绍的相同技术，你可以在R中使用这些扩展。

例如，有一个Apache Solr，它是一个用于在大型数据集上执行全文搜索的系统，Apache Spark目前并不原生支持这种功能。另外，在本文撰写之时，还没有R扩展来支持Solr。所以，让我们试着用Spark扩展来解决这个问题。

首先，如果搜索“spark packages.org”来找到Solr扩展，那么你应该能够找到spark-solr。“How to”扩展指出应该加载com.lucidworks.spark:spark-solr:2.0.1。我们可以使用sparklyr.shell.packages配置选项在R中完成配置：

config <- spark\_config()  
config["sparklyr.shell.packages"] <- "com.lucidworks.spark:spark-solr:3.6.3"  
config["sparklyr.shell.repositories"] <- "http://repo.spring.io/plugins-release/,http://central.maven.org/maven2/"  
sc <- spark\_connect(master = "local", config = config)

虽然指定sparklyr.shell.packages参数通常就足够了，但是对于这个特定的扩展，依赖项无法从Spark包仓库下载。你需要在Maven repo中手动查找失败的依赖项，并在sparklyr.shell.repositories参数下添加更多的仓库。

当使用扩展时，Spark会连接到Maven包仓库来以便获取检索扩展。这可能需要很长时间，取决于扩展和你的下载速度。在这种情况下，考虑使用sparklyr.connect.timeout配置参数来允许Spark下载所需的文件。

从spark-solr文档中，你可以找到这个扩展可以通过以下Scala代码使用:

val options = Map(  
"collection" -> "{solr\_collection\_name}",  
"zkhost" -> "{zk\_connect\_string}"  
)  
val df = spark.read.format("solr") .options(options) .load()

我们可以翻译成R代码：

spark\_session(sc) %>% invoke("read") %>% invoke("format", "solr") %>% invoke("option",   
 "collection", "<collection>") %>% invoke("option", "zkhost", "<host>") %>% invoke("load")

但是，这段代码会失败，因为它需要一个有效的Solr实例，并且配置Solr超出了本书的范围。但是这个例子帮你理解如何创建Spark扩展。值得一提的是，你可以使用spark\_read\_source()从通用源中读取，以避免编写自定义的invoke()代码。

正如“概述”中指出的，考虑与使用R包的其它人共享代码。虽然可以要求你的包的用户指定sparklyr.shell.packages，但是在R包中注册依赖项可以避免这种情况。依赖项在spark\_dependencies()函数下声明；因此，对于本节中的示例：

spark\_dependencies <- function(spark\_version, scala\_version, ...) {  
 spark\_dependency(packages = "com.lucidworks.spark:spark-solr:3.6.3", repositories = c("http://repo.spring.io/plugins-release/",   
 "http://central.maven.org/maven2/"))  
}  
.onLoad <- function(libname, pkgname) {  
 sparklyr::register\_extension(pkgname)  
}

当加载库时，onLoad函数由R自动调用。它应该调用register\_extension()，该函数再回调spark\_dependencies()，以允许扩展提供其它依赖项。本例支持Spark 2.4，但也应该支持将Spark和Scala版本映射到正确的Spark扩展版本。

你可以使用大约450个Spark扩展；此外，你还可以使用Maven仓库中的任何Java库，其中Maven Central有超过300万个组件。虽然并不是所有的Maven Central库都与Spark相关，但是Spark扩展和Maven仓储库的结合无疑为你提供了许多值得考虑的可能性！

但是，对于那些没有Spark扩展的情况，下一节将介绍如何使用来自自己R包的定制Scala代码。

## 使用Scala代码

Scala代码允许你使用Spark API、Spark扩展或Java库中的任何方法。此外，在Spark中运行时编写Scala代码可以在使用spark\_apply()的R代码时提供性能改进。通常，R包结构会包含R代码和Scala代码；但是，Scala代码需要编译为JAR包（Java归档文件）并包含在程序包中。图13-2展示了这种结构。

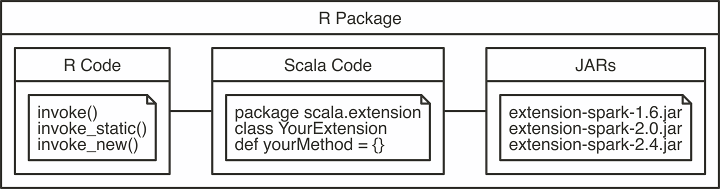


图13-2. 使用Scala代码时的R程序包结构

通常，R代码应该放在顶层R文件夹下，Scala代码放在java文件夹下，编译的JAR包放在inst/java文件夹下。尽管欢迎手动编译Scala代码，但是你可以使用帮助函数下载所需的编译器并编译Scala代码。

要编译Scala代码，需要安装Java Development Kit 8（简称JDK8）。从Oracle的Java下载页面下载JDK；这需要重新启动R进程。

你还需要Scala编译器支持Scala 2.11和2.12。使用download\_scalac()，Scala编译器会自动下载和安装：

download\_scalac()

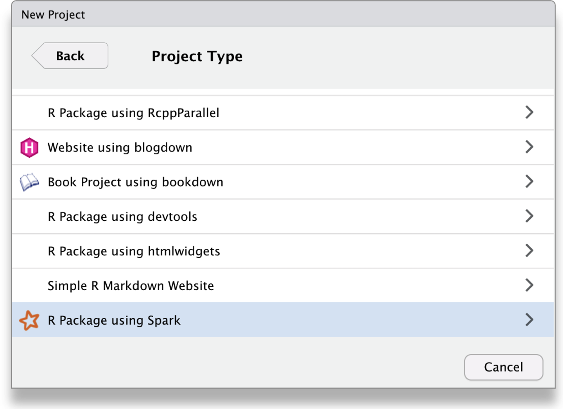
接下来，需要使用compile\_package\_jars()编译Scala源代码。默认情况下，它使用spark\_compilation\_spec()，并支持以下Spark版本：

## [1] '1.5.2' '1.6.0' '2.0.0' '2.3.0' '2.4.0'

你还可以使用spark\_compilation\_spec()创建自定义条目来定义这个配置。

虽然可以从头开始为Scala代码创建项目结构，但也可以简单地调用spark\_extension(path)在给定路径中创建扩展。 虽然这个扩展很可能是空的，但是包含调用Scala代码的项目结构。

由于spark\_extension()在RStudio中注册为自定义项目扩展，你还可以从“文件”菜单中使用Scala代码创建扩展spark的R包；单击“新建项目”，然后选择“使用spark的R包”，如图13-3所示。

 准备好编译程序包的JAR包后，你只需要运行以下代码：

compile\_package\_jars()

由于JAR包默认编译到inst/package路径中，因此在构建R包时，所有JAR包也将包含在程序包中。这意味着你可以共享或发布你的R包，为R用户提供相关功能。对于拥有Scala专业知识的Spark高级用户来说，必须考虑用Scala为R用户和R社区编写库，然后将这些库打包成易于调取、使用和共享的R包。

如果你对R环境中的Spark扩展开发感兴趣，而在开发过程中遇到困难，可以考虑加入sparklyr的Gitter频道，我们中的许多人都会在这里帮助社区成长。我们希望很快听到你的声音！

## 小结

最后一章介绍了一组新的工具，用于将Spark功能扩展到R和R包当前支持的范围之外。这个巨大的仓库包括450多个Spark扩展和数以百万计的Java组件。它们都帮助我们在R中使用Spark。除了这些资源之外，你还学习了如何使用Scala代码构建Java组件，该代码可以很容易地从R嵌入和编译。

这让我们回到了本书的初衷：虽然我们知道你已经学习了如何使用在R中使用Saprk执行大规模计算，还相信你已经掌握了通过Spark扩展帮助其它社区成员所需的知识。我们迫不及待地想看到你们新的创造，这肯定会有助于Spark和R社区的发展。

最后，我们希望前几章完成了Spark和R的简单介绍。然后，我们为了使用Spark，通过熟悉的R程序包，介绍了分析和建模基础。接着，你学习了如何在合适的Spark集群中执行大规模计算。本书的最后三分之一关注在高级主题上：使用扩展、分发R代码、处理实时数据。最后，借助R和Scala代码使用Spark扩展回馈社区。

我们已经尽所能提供了最好的内容。但是，如果你知道任何改进本书的方法，请在the-r-in-spark仓库下新开一个GitHub issue。我们会在以后的修订中关注你的建议。我们希望你喜欢读这本书，而且你学到的和我们在写这本书时学到的一样多。我们希望你值得花时间阅读本书。有你作为我们的读者是我们的荣幸。

# 附录A：补充参考代码

在整本书中，我们都提到了这个附录。这里我们包括了重要的内容（并列出了可以找到这些材料的章节）。

## 前言

### 格式化

下面的ggplot2主题用来格式化本书的所有绘图：

plot\_style <- function() {  
 font <- "Helvetica"  
 ggplot2::theme\_classic() + ggplot2::theme(plot.title = ggplot2::element\_text(family = font,   
 size = 14, color = "#222222"), plot.subtitle = ggplot2::element\_text(family = font,   
 size = 12, color = "#666666"), legend.position = "right", legend.background = ggplot2::element\_blank(),   
 legend.title = ggplot2::element\_blank(), legend.key = ggplot2::element\_blank(),   
 legend.text = ggplot2::element\_text(family = font, size = 14, color = "#222222"),   
 axis.title.y = ggplot2::element\_text(margin = ggplot2::margin(t = 0, r = 8,   
 b = 0, l = 0), size = 14, color = "#666666"), axis.title.x = ggplot2::element\_text(margin = ggplot2::margin(t = -2,   
 r = 0, b = 0, l = 0), size = 14, color = "#666666"), axis.text = ggplot2::element\_text(family = font,   
 size = 14, color = "#222222"), axis.text.x = ggplot2::element\_text(margin = ggplot2::margin(5,   
 b = 10)), axis.ticks = ggplot2::element\_blank(), axis.line = ggplot2::element\_blank(),   
 panel.grid.minor = ggplot2::element\_blank(), panel.grid.major.y = ggplot2::element\_line(color = "#eeeeee"),   
 panel.grid.major.x = ggplot2::element\_line(color = "#ebebeb"), panel.background = ggplot2::element\_blank(),   
 strip.background = ggplot2::element\_rect(fill = "white"), strip.text = ggplot2::element\_text(size = 20,   
 hjust = 0))  
}

你可以使用下列代码激活：

ggplot2::theme\_set(plot\_style())

## 第1章

### 全球存储信息能力

下列脚本用来生成图1-1：

library(ggplot2)  
library(dplyr)  
library(tidyr)  
read.csv("data/01-worlds-capacity-to-store-information.csv", skip = 8) %>% gather(key = storage,   
 value = capacity, analog, digital) %>% mutate(year = X, terabytes = capacity/1e+12) %>%   
 ggplot(aes(x = year, y = terabytes, group = storage)) + geom\_line(aes(linetype = storage)) +   
 geom\_point(aes(shape = storage)) + scale\_y\_log10(breaks = scales::trans\_breaks("log10",   
 function(x) 10^x), labels = scales::trans\_format("log10", scales::math\_format(10^x))) +   
 theme\_light() + theme(legend.position = "bottom")

### CRAN程序包每日下载量

图1-6通过以下代码生成：

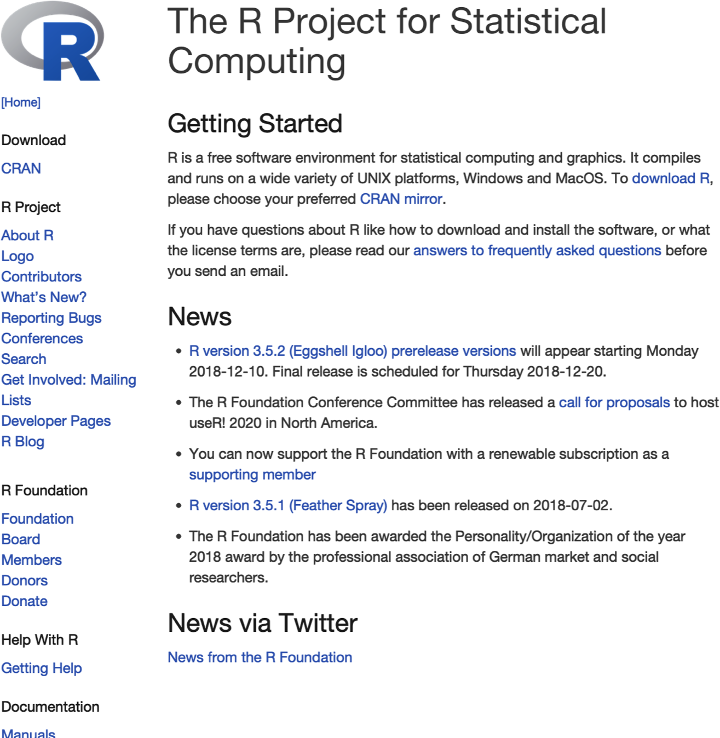
downloads\_csv <- "data/01-intro-r-cran-downloads.csv"  
if (!file.exists(downloads\_csv)) {  
 downloads <- cranlogs::cran\_downloads(from = "2014-01-01", to = "2019-01-01")  
 readr::write\_csv(downloads, downloads\_csv)  
}  
cran\_downloads <- readr::read\_csv(downloads\_csv)  
ggplot(cran\_downloads, aes(date, count)) + labs(title = "CRAN Packages", subtitle = "Total daily downloads over time") +   
 geom\_point(colour = "black", pch = 21, size = 1) + scale\_x\_date() + xlab("year") +   
 ylab("downloads") + scale\_x\_date(date\_breaks = "1 year", labels = function(x) substring(x,   
 1, 4)) + scale\_y\_continuous(limits = c(0, 3.5 \* 10^6), breaks = c(0.5 \* 10^6,   
 10^6, 1.5 \* 10^6, 2 \* 10^6, 2.5 \* 10^6, 3 \* 10^6, 3.5 \* 10^6), labels = c("",   
 "1M", "", "2M", "", "3M", ""))

## 第2章

### 预备知识

#### 安装R

下载R安装文件（见图A-1），在Windows，Mac或Linux平台上启动。



图A-1. 面向统计计算的R项目

#### 安装Java

下载Java安装文件（见图A-2），在Windows，Mac或Linux平台上启动。



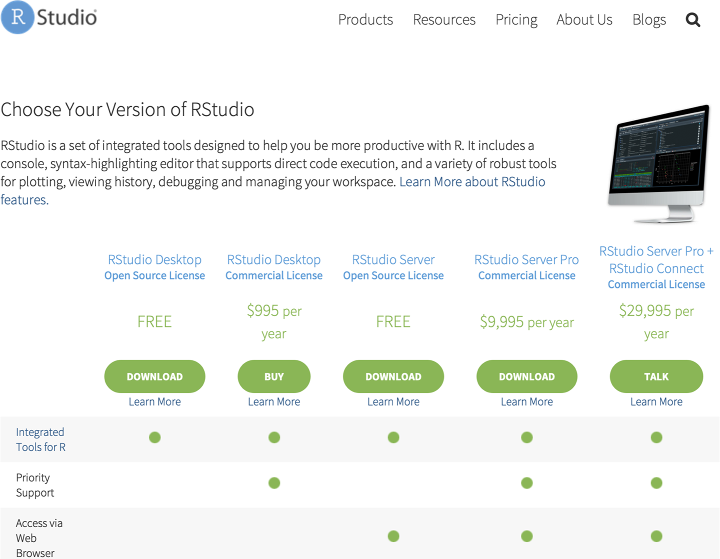
图A-2. Java下载页

Spark 2.1需要Java 8。但是，之前的Spark版本支持Java 7。不管怎么样，我们都建议安装Java Runtime Engine 8（JRE 8）。

如果你是一个已经在使用Java Development Kit（JDK）的高级读者，请注意JDK 9+目前不受支持。因此，你需要通过卸载JDK 9+或适当设置JAVA\_HOME来降级到JDK 8。

#### 安装RStudio

虽然安装RStudio在使用Spark和R并不是必须的，但它会提高工作效率。因此，我们建议还是安装Rstudio。下载RStudio安装程序（参见图A-3），然后在Windows，Mac或Linux平台上启动。

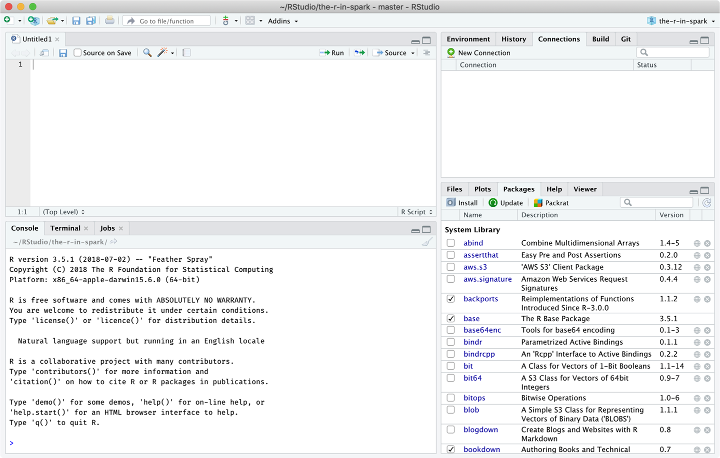


图A-3. RStudio下载页

启动RStudio后，你可以使用控制台面板来执行本章中的代码。

#### 使用RStudio

如果你不熟悉RStudio（参见图A-4），应该注意以下面板： \* 控制台 可以使用一个独立的R控制台来执行本书中介绍的所有代码。 \* 程序包 此面板允许你轻松安装sparklyr，检查其版本，导航到帮助内容，等等。 \* 连接 此面板允许你连接到Spark，管理活动连接，并查看可用的数据集。



图A-4. RStudio界面

## 第3章

### Hive函数

|  |  |
| --- | --- |
| 名称 | 描述 |
| size(Map<K.V>) | Returns the number of elements in the map type. |
| size(Array) | Returns the number of elements in the array type. |
| map\_keys(Map<K.V>) | Returns an unordered array containing the keys of the input map. |
| map\_values(Map<K.V>) | Returns an unordered array containing the values of the input map. |
| array\_contains(Array, value) | Returns TRUE if the array contains a value. |
| sort\_array(Array) | Sorts the input array in ascending order according to the natural ordering of the array elements and returns it. |
| binary(string or binary) | Casts the parameter into a binary. |
| cast(expr as a given type) | Converts the results of the expression expr to the given type. |
| from\_unixtime(bigint unixtime[, string format]) | Converts the number of seconds from Unix epoch (1970-01-01 00:00:00 UTC) to a string. |
| unix\_timestamp() | Gets current Unix timestamp in seconds. |
| unix\_timestamp(string date) | Converts a time string in the format yyyy-MM-dd HH:mm:ss to a Unix timestamp (in seconds). |
| to\_date(string timestamp) | Returns the date part of a timestamp string. |
| year(string date) | Returns the year part of a date. |
| quarter(date/timestamp/string) | Returns the quarter of the year for a date. |
| month(string date) | Returns the month part of a date or a timestamp string. |
| day(string date) dayofmonth(date) | Returns the day part of a date or a timestamp string. |
| hour(string date) | Returns the hour of the timestamp. |
| minute(string date) | Returns the minute of the timestamp. |
| second(string date) | Returns the second of the timestamp. |
| weekofyear(string date) | Returns the week number of a timestamp string. |
| extract(field FROM source) | Retrieve fields such as days or hours from source. Source must be a date,timestamp, interval, or string that can be converted into either a date or timestamp. |
| datediff(string enddate, string startdate) | Returns the number of days from startdate to enddate. |
| date\_add(date/timestamp/string startdate, tinyint/smallint/int days) | Adds a number of days to startdate. |
| date\_sub(date/timestamp/string startdate, tinyint/smallint/int days) | Subtracts a number of days to startdate. |
| from\_utc\_timestamp(\{any primitive type} ts, string timezone) | Converts a timestamp in UTC to a given time zone. |
| to\_utc\_timestamp(\{any primitive type} ts, string timezone) | Converts a timestamp in a given time zone to UTC. |
| current\_date | Returns the current date. |
| current\_timestamp | Returns the current timestamp. |
| add\_months(string start\_date, int num\_months, output\_date\_format) | Returns the date that is num\_months after start\_date. |
| last\_day(string date) | Returns the last day of the month to which the date belongs. |
| next\_day(string start\_date, string day\_of\_week) | Returns the first date that is later than start\_date and named as day\_of\_week. |
| trunc(string date, string format) | Returns date truncated to the unit specified by the format. |
| months\_between(date1, date2) | Returns number of months between dates date1 and date2. |
| date\_format(date/timestamp/string ts, string fmt) | Converts a date/timestamp/string to a value of string in the format specified by the date format fmt. |
| if(boolean testCondition, T valueTrue, T valueFalseOrNull) | Returns valueTrue when testCondition is true; returns valueFalseOrNull otherwise. |
| isnull( a ) | Returns true if a is NULL and false otherwise. |
| isnotnull( a ) | Returns true if a is not NULL and false otherwise. |
| nvl(T value, T default\_value) | Returns default value if value is NULL; else returns value. |
| COALESCE(T v1, T v2, …) | Returns the first v that is not NULL, or NULL if all vs are NULL. |
| CASE a WHEN b THEN c [WHEN d THEN e]\* [ELSE f] END | When a = b, returns c; when a = d, returns e; else returns f. |
| nullif( a, b ) | Returns NULL if a = b; otherwise returns a. |
| assert\_true(boolean condition) | Throw an exception if condition is not true; otherwise return NULL. |
| ascii(string str) | Returns the numeric value of the first character of str. |
| base64(binary bin) | Converts the argument from binary to a base64 string. |
| character\_length(string str) | Returns the number of UTF-8 characters contained in str. |
| chr(bigint double A) |  |
| concat(stringǀbinary A, stringǀbinary B…) | Returns the string or bytes resulting from concatenating the strings or bytes passed in as parameters in order. For example, concat(foo, bar) results in foobar. |
| context\_ngrams(array<array>,array, int K, int pf) | Returns the top-k contextual N-grams from a set of tokenized sentences. |
| concat\_ws(string SEP, string A, string B…) | Like concat(), but with custom separator SEP. |
| decode(binary bin, string charset) | Decodes the first argument into a string using the provided character set (one of US-ASCII, ISO-8859-1, UTF-8, UTF-16BE, UTF-16LE, or UTF-16). If either argument is NULL, the result will also be NULL. |
| elt(N int,str1 string,str2 string,str3 string,…) | Return string at index number; elt(2,hello,world) |
| encode(string src, string charset) | Encodes the first argument into a BINARY using the provided character set (one of US-ASCII, ISO-8859-1, UTF-8, UTF-16BE, UTF-16LE, UTF-16). |
| field(val T,val1 T,val2 T,val3 T,…) | Returns the index of val in the val1,val2,val3,… list or 0 if not found. |
| find\_in\_set(string str, string strList) | Returns the first occurance of str in strList where strList is a comma-delimited string. |
| format\_number(number x, int d) | Formats the number x to a format like ‘#,###,###.##’, rounded to d decimal places, and returns the result as a string. If d is 0, the result has no decimal point or fractional part. |
| get\_json\_object(string json\_string, string path) | Extracts JSON object from a JSON string based on JSON path specified, and returns JSON string of the extracted JSON object. |
| in\_file(string str, string filename) | Returns true if the string str appears as an entire line in filename. |
| instr(string str, string substr) | Returns the position of the first occurrence of substr in str. |
| length(string A) | Returns the length of the string. |
| locate(string substr, string str[, int pos]) | Returns the position of the first occurrence of substr in str after position pos. |
| lower(string A) lcase(string A) | Returns the string resulting from converting all characters of B to lowercase. |
| lpad(string str, int len, string pad) | Returns str, left-padded with pad to a length of len. If str is longer than len, the return value is shortened to len characters. |
| ltrim(string A) | Returns the string resulting from trimming spaces from the beginning (left side) of A. |
| ngrams(array<array>, int N, int K, int pf) | Returns the top-k N-grams from a set of tokenized sentences, such as those returned by the sentences() UDAF. |
| octet\_length(string str) | Returns the number of octets required to hold the string str in UTF-8 encoding. |
| parse\_url(string urlString, string partToExtract [, string keyToExtract]) | Returns the specified part from the URL. Valid values for partToExtract include HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, and USERINFO. |
| printf(String format, Obj… args) | Returns the input formatted according do printf-style format strings. |
| regexp\_extract(string subject, string pattern, int index) | Returns the string extracted using the pattern. |
| regexp\_replace(string INITIAL\_STRING, string PATTERN, string REPLACEMENT) | Returns the string resulting from replacing all substrings in INITIAL\_STRING that match the Java regular expression syntax defined in PATTERN with instances of REPLACEMENT. |
| repeat(string str, int n) | Repeats str n times. |
| replace(string A, string OLD, string NEW) | Returns the string A with all non-overlapping occurrences of OLD replaced with NEW. |
| reverse(string A) | Returns the reversed string. |
| rpad(string str, int len, string pad) | Returns str, right-padded with pad to a length of len. |
| rtrim(string A) | Returns the string resulting from trimming spaces from the end (right side) of A. |
| sentences(string str, string lang, string locale) | Tokenizes a string of natural language text into words and sentences, where each sentence is broken at the appropriate sentence boundary and returned as an array of words. |
| space(int n) | Returns a string of n spaces. |
| split(string str, string pat) | Splits str around pat (pat is a regular expression). |
| str\_to\_map(text[, delimiter1, delimiter2]) | Splits text into key-value pairs using two delimiters. delimiter1 separates text into key-value pairs, and delimiter2 splits each key-value pair. Default delimiters are , for delimiter1 and : for delimiter2. |
| substr(string binary A, int start) | Returns the substring or slice of the byte array of A starting from start position till the end of string A. |
| substring\_index(string A, string delim, int count) | Returns the substring from string A before count occurrences of the delimiter delim. |
| translate(stringǀcharǀvarchar input, stringǀcharǀvarchar from, stringǀcharǀvarchar to) | Translates the input string by replacing the characters present in the from string with the corresponding characters in the to string. |
| trim(string A) | Returns the string resulting from trimming spaces from both ends of A. |
| unbase64(string str) | Converts the argument from a base64 string to BINARY. |
| upper(string A) ucase(string A) | Returns the string resulting from converting all characters of A to uppercase. For example, upper(fOoBaR) results in FOOBAR’\. |
| initcap(string A) | Returns string, with the first letter of each word in uppercase, and all other letters in lowercase. Words are delimited by whitespace. |
| levenshtein(string A, string B) | Returns the Levenshtein distance between two strings. |
| soundex(string A) | Returns soundex code of the string. |
| mask(string str[, string upper[, string lower[, string number]]]) | Returns a masked version of str. |
| mask\_first\_n(string str[, int n]) | Returns a masked version of str with the first n values masked. |
| mask\_first\_n("1234-5678-8765-4321", 4) | Results in nnnn-5678-8765-4321. |
| mask\_last\_n(string str[, int n]) | Returns a masked version of str with the last n values masked. |
| mask\_show\_first\_n(string str[, int n]) | Returns a masked version of str, showing the first n characters unmasked. |
| mask\_show\_last\_n(string str[, int n]) | Returns a masked version of str, showing the last n characters unmasked. |
| mask\_hash(stringǀcharǀvarchar str) | Returns a hashed value based on str. |
| java\_method(class, method[, arg1[, arg2..]]) | Synonym for reflect. |
| reflect(class, method[, arg1[, arg2..]]) | Calls a Java method by matching the argument signature, using reflection. |
| hash(a1[, a2…]) | Returns a hash value of the arguments. |
| current\_user() | Returns current username from the configured authenticator manager. |
| logged\_in\_user() | Returns current username from the session state. |
| current\_database() | Returns current database name. |
| md5(string/binary) | Calculates an MD5 128-bit checksum for the string or binary. |
| sha1(string/binary) sha(string/binary) | Calculates the SHA-1 digest for string or binary and returns the value as a hex string. |
| crc32(string/binary) | Computes a cyclic redundancy check value for string or binary argument and returns bigint value. |
| sha2(string/binary, int) | Calculates the SHA-2 family of hash functions (SHA-224, SHA-256, SHA-384, and SHA-512). |
| aes\_encrypt(input string/binary, key string/binary) | Encrypt input using AES. |
| aes\_decrypt(input binary, key string/binary) | Decrypt input using AES. |
| version() | Returns the Hive version. |
| count(expr) | Returns the total number of retrieved rows. |
| sum(col), sum(DISTINCT col) | Returns the sum of the elements in the group or the sum of the distinct values of the column in the group. |
| avg(col), avg(DISTINCT col) | Returns the average of the elements in the group or the average of the distinct values of the column in the group. |
| min(col) | Returns the minimum value of the column in the group. |
| max(col) | Returns the maximum value of the column in the group. |
| variance(col), var\_pop(col) | Returns the variance of a numeric column in the group. |
| var\_samp(col) | Returns the unbiased sample variance of a numeric column in the group. |
| stddev\_pop(col) | Returns the standard deviation of a numeric column in the group. |
| stddev\_samp(col) | Returns the unbiased sample standard deviation of a numeric column in the group. |
| covar\_pop(col1, col2) | Returns the population covariance of a pair of numeric columns in the group. |
| covar\_samp(col1, col2) | Returns the sample covariance of a pair of a numeric columns in the group. |
| corr(col1, col2) | Returns the Pearson coecient of correlation of a pair of a numeric columns in the group. |
| percentile(BIGINT col, p) | Returns the exact \_p\_th percentile of a column in the group (does not work with floating-point types). p must be between 0 and 1. |
| percentile(BIGINT col, array(p1 [, p2]…)) | Returns the exact percentiles p1, p2, … of a column in the group. pi must be between 0 and 1. |
| percentile\_approx(DOUBLE col, p [, B]) | Returns an approximate \_p\_th percentile of a numeric column (including floating point types) in the group. The B parameter controls approximation accuracy at the cost of memory. Higher values yield better approximations, and the default is 10,000. When the number of distinct values in col is smaller than B, this gives an exact percentile value. |
| percentile\_approx(DOUBLE col, array(p1 [, p2]…) [, B]) | Same as previous, but accepts and returns an array of percentile values instead of a single one. |
| regr\_avgx(independent, dependent) | Equivalent to avg(dependent). |
| regr\_avgy(independent, dependent) | Equivalent to avg(independent). |
| regr\_count(independent, dependent) | Returns the number of non-null pairs used to fit the linear regression line. |
| regr\_intercept(independent, dependent) | Returns the y-intercept of the linear regression line—that is, the value of b in the equation dependent = a \* independent + b. |
| regr\_r2(independent, dependent) | Returns the coecient of determination for the regression. |
| regr\_slope(independent, dependent) | Returns the slope of the linear regression line—that is, the value of a in the equation dependent = a \* independent + b. |
| regr\_sxx(independent, dependent) | Equivalent to regr\_count(independent, dependent) \* var\_pop(dependent). |
| regr\_sxy(independent, dependent) | Equivalent to regr\_count(independent, dependent) \* covar\_pop(independent, dependent) |
| regr\_syy(independent, dependent) | Equivalent to regr\_count(independent, dependent) \* var\_pop(independent). |
| histogram\_numeric(col, b) | Computes a histogram of a numeric column in the group using b nonuniformly spaced bins. The output is an array of size b of double-valued (x,y) coordinates that represent the bin centers and heights. |
| collect\_set(col) | Returns a set of objects with duplicate elements eliminated. |
| collect\_list(col) | Returns a list of objects with duplicates. |
| ntile(INTEGER x) | Divides an ordered partition into x groups called buckets and assigns a bucket number to each row in the partition. This allows easy calculation of tertiles, quartiles, deciles, percentiles, and other common summary statistics. |
| explode(ARRAY a) | Explodes an array to multiple rows. Returns a row-set with a single column (col), one row for each element from the array. |
| explode(MAP<Tkey,Tvalue> m) | Explodes a map to multiple rows. Returns a row-set with a two columns (key,value), one row for each key-value pair from the input map. |
| posexplode(ARRAY a) | Explodes an array to multiple rows with additional positional column of int type (position of items in the original array, starting with 0). Returns a rowset with two columns (pos,val), one row for each element from the array. |
| inline(ARRAY<STRUCT<f1:T1,…,fn:Tn>> a) | Explodes an array of structs to multiple rows. Returns a row-set with N columns (N = number of top-level elements in the struct), one row per struct from the array. |
| stack(int r,T1 V1,…,Tn/r Vn) | Breaks up n values V1,…,Vn into r rows. Each row will have n/r columns. r must be constant. |
| json\_tuple(string jsonStr,string k1,…,string kn) | Takes JSON string and a set of n keys, and returns a tuple of n values. |
| parse\_url\_tuple(string urlStr,string p1,…,string pn) | Takes URL string and a set of n URL parts, and returns a tuple of n values. |

## 第4章

### MLlib函数

下表展示了sparklyr支持的机器学习算法：

#### 分类

|  |  |
| --- | --- |
| 算法 | 函数 |
| 决策树 | ml\_decision\_tree\_classifier() |
| 梯度提升树 | ml\_gbt\_classifier() |
| 线性支持向量机 | ml\_linear\_svc() |
| 逻辑回归 | ml\_logistic\_regression() |
| 多层感知机 | ml\_multilayer\_perceptron\_classifier() |
| 朴素贝叶斯 | ml\_naive\_bayes() |
| 一对多分类 | ml\_one\_vs\_rest() |
| 随机森林 | ml\_random\_forest\_classifier() |

#### 回归

|  |  |
| --- | --- |
| 算法 | 函数 |
| 加速时效时间生成回归 | ml\_aft\_survival\_regression() |
| 决策树 | ml\_decision\_tree\_regressor() |
| 广义线性回归 | ml\_generalized\_linear\_regression() |
| 梯度提升树 | ml\_gbt\_regressor() |
| 保序回归 | ml\_isotonic\_regression() |
| 线性回归 | ml\_linear\_regression() |

#### 聚类

|  |  |
| --- | --- |
| 算法 | 函数 |
| 二分k-means聚类 | ml\_bisecting\_kmeans() |
| 高斯混合聚类 | ml\_gaussian\_mixture() |
| k-means聚类 | ml\_kmeans() |
| 隐狄利克雷分布 | ml\_lda() |

#### 推荐

|  |  |
| --- | --- |
| 算法 | 函数 |
| 交替最小二乘分解 | ml\_als() |

#### 常见模式挖掘

|  |  |
| --- | --- |
| 算法 | 函数 |
| 频繁模式增长 | ml\_fpgrowth() |

#### 特征变换

|  |  |
| --- | --- |
| 变换器 | 函数 |
| 二值化器 | ft\_binarizer() |
| 分桶器 | ft\_bucketizer() |
| 卡方特征选择器 | ft\_chisq\_selector() |
| 文档集合词表 | ft\_count\_vectorizer() |
| 离散余弦转换 | ft\_discrete\_cosine\_transform() |
| 使用dplyr转换 | ft\_dplyr\_transformer() |
| Hadamard积 | ft\_elementwise\_product() |
| 特征哈希 | ft\_feature\_hasher() |
| 使用哈希的词频 | export(ft\_hashing\_tf) |
| 逆文档频率 | ft\_idf() |
| 缺失值推断 | export(ft\_imputer) |
| Index to string | ft\_index\_to\_string() |
| 特征交互转换 | ft\_interaction() |
| [–1, 1]区间缩放 | ft\_max\_abs\_scaler() |
| [min, max]区间缩放 | ft\_min\_max\_scaler() |
| 局部敏感哈希 | ft\_minhash\_lsh() |
| 转换为n-grams | ft\_ngram() |
| 给定P-norm的归一 | ft\_normalizer() |
| One-hot编码 | ft\_one\_hot\_encoder() |
| 多项式空间中的特征扩展 | ft\_polynomial\_expansion() |
| 映射到分箱的类别型特征 | ft\_quantile\_discretizer() |
| SQL变换 | ft\_sql\_transformer() |
| 使用修正的STD标准化特征 | ft\_standard\_scaler() |
| 过滤停用词 | ft\_stop\_words\_remover() |
| 映射到标签索引 | ft\_string\_indexer() |
| 按空格切分 | ft\_tokenizer() |
| 合并向量为行向量 | ft\_vector\_assembler() |
| 为类别型特征建立索引 | ft\_vector\_indexer() |
| 原始特征的子数组 | ft\_vector\_slicer() |
| 讲词转换为编码 | ft\_word2vec() |

## 第6章

### 本地机（主机），云计算和Kubernetes的谷歌趋势

用于创建图6-1的数据可以从https://bit.ly/2YnHkNI下载。

library(dplyr)  
read.csv("data/clusters-trends.csv", skip = 2) %>% mutate(year = as.Date(paste(Month,   
 "-01", sep = ""))) %>% mutate(`On-Premise` = mainframe...Worldwide., Cloud = cloud.computing...Worldwide.,   
 Kubernetes = kubernetes...Worldwide.) %>% tidyr::gather(`On-Premise`, Cloud,   
 Kubernetes, key = "trend", value = "popularity") %>% ggplot(aes(x = year, y = popularity,   
 group = trend)) + geom\_line(aes(linetype = trend, color = trend)) + scale\_x\_date(date\_breaks = "2 year",   
 date\_labels = "%Y") + labs(title = "Cluster Computing Trends", subtitle = paste("Search popularity for on-premise (mainframe)",   
 "cloud computing and kubernetes ")) + scale\_color\_grey(start = 0.6, end = 0.2) +   
 geom\_hline(yintercept = 0, size = 1, colour = "#333333") + theme(axis.title.x = element\_blank())

## 第12章

### 数据流生成器

第12章中介绍的stream\_generate\_test()函数可以创建本地测试流。此功能独立于Spark连接而工作。下面的示例将在原始子文件夹中创建五个文件。每个文件将在上一个文件完成创建后1秒创建：

library(sparklyr)  
stream\_generate\_test(iterations = 5, path = "source", interval = 1)

函数完成后，所有文件都应显示在源文件夹中。请注意，文件大小各不相同：这是为了模拟真实流的作用：

file.info(file.path("source", list.files("source")))[1]  
## size source/stream\_1.csv 44 source/stream\_2.csv 121 source/stream\_3.csv 540  
## source/stream\_4.csv 2370 source/stream\_5.csv 7236

stream\_generate\_test()默认创建一个单个数值变量的数据框。

readr::read\_csv("source/stream\_5.csv")  
## # A tibble: 1,489 x 1 x <dbl> 1 630 2 631 3 632 4 633 5 634 6 635 7 636 8 637 9  
## 638 10 639 # ... with 1,479 more rows

### 安装Kafka

本说明是根据Kafka官方网站的快速入门页面中的信息形成的。（毫无疑问，在本书出版后不久，新版本的Kafka将面世。）此处是为了给“Kafka”中的例子中使用的版本打上“时间戳”：

1. 下载Kafka。

wget http://apache.claz.org/kafka/2.2.0/kafka\_2.12-2.2.0.tgz

1. 扩展*tar*文件，并进入新的文件夹。

tar -xzf kafka\_2.12-2.2.0.tgz  
cd kafka\_2.12-2.2.0

1. 启动带有Kafka的Zookeeper服务。

bin/zookeeper-server-start.sh config/zookeeper.properties

1. 启动Kafka服务。

bin/kafka-server-start.sh config/server.properties

确保总是首先启动Zookeeper，然后启动Kafka。

# 关于作者

Javier Luraschi是一名软件工程师，在从桌面、web、移动和后端开发到增强现实和深度学习应用程序的技术方面具有丰富经验。他曾供职于微软研究院和SAP，拥有数学和软件工程双学位。他是sparklyr、cloudml、r2d3、mlflow、tfdeploy和kerasjs的作者。

Kevin Kuo为机器学习和模型部署构建了开源代码库。他曾在多个行业担任数据科学家职位，其中包括保险业，他是一名持照的精算师。Kevin是mlflow、mleap和sparkxgb的作者。他也是一个业余的调酒师和品酒师。

Edgar Ruiz具有部署企业报告和商业智能解决方案的背景。他是多篇文章和博客文章的作者，旨在分享分析见解和数据科学项目的服务器架构。Edgar是db.rstudio.com网站的作者和管理员，也是sparklyr网站的现任管理员。他还与人合著了dbplyr包，并创建了dbplot、tidypredict和modeldb包。

# 结语

《Mastering Spark with R》封面上的动物是一种玫瑰鱼（褐家鼠），一种深海岩鱼。虽然大多数岩鱼生活在北太平洋，但玫瑰鱼是生活在北大西洋的四种鱼类之一。

幼年玫瑰鱼生活在峡湾附近，而成年玫瑰鱼生活在3280英尺深的水域。成年玫瑰鱼平均有18英寸长，然后某些也可以达到3英尺。它们是生长缓慢的鱼类，在8或9岁左右达到完全成熟。玫瑰鱼是胎生的；在冬季，雌性会产下50000-350000个卵。玫瑰鱼寿命可达60岁。

玫瑰鱼是一种商业捕捞的鱼类，年捕获量为40至60千吨。人们认为是玫瑰鱼已经被严重过度捕捞了，特别是考虑到需要8或9年才能达到生殖年龄。

奥雷利岛上的许多动物正濒临灭绝，而它们对世界都很重要。

封面插图由Jose Marzan根据Histoire Naturelle的黑白雕刻而成。封面字体是Gilroy Semibold和Guardian Sans。文本字体是Adobe Minion Pro；标题字体是Adobe Myriad Condensed；代码字体是Dalton Maag的Ubuntu Mono。

1. Laudon KC, Traver CG, Laudon JP (1996). “Information technology and systems.” Cambridge, MA: Course Technology。 [↑](#footnote-ref-29)
2. Ceruzzi PE (2012). Computing: a concise history. MIT Press. [↑](#footnote-ref-30)
3. Group WB (2016). The Data Revolution. World Bank Publications. [↑](#footnote-ref-31)
4. Ghemawat S, Gobioff H, Leung S (2003). “The Google File System.” In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. ISBN 1-58113-757-5. [↑](#footnote-ref-32)
5. Dean J, Ghemawat S (2004). “MapReduce: Simplified data processing on large clusters.” In USENIX Symposium on Operating System Design and Implementation (OSDI). [↑](#footnote-ref-35)
6. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010). “Spark: Cluster computing with working sets.” HotCloud, 10(10-10), 95. [↑](#footnote-ref-38)
7. Wickham H, Grolemund G (2016). R for data science: import, tidy, transform, visualize, and model data. O’Reilly Media, Inc [↑](#footnote-ref-44)
8. Wu CJ (1997). “Statistics = Data Science?” [↑](#footnote-ref-45)
9. Cleveland WS (2001). “Data Science: An Action Plan for Expanding the Technical Areas of the Field of Statistics?” [↑](#footnote-ref-46)
10. Samuel AL (1959). “Some studies in machine learning using the game of checkers.” IBM Journal of research and development, 3(3), 210–229. [↑](#footnote-ref-47)
11. Hinton GE, Osindero S, Teh Y (2006). “A fast learning algorithm for deep belief nets.” Neural computation, 18(7), 1527–1554. [↑](#footnote-ref-48)
12. Krizhevsky A, Sutskever I, Hinton GE (2012). “Imagenet classification with deep convolutional neural networks.” In Advances in neural information processing systems, 1097–1105. [↑](#footnote-ref-49)
13. Xie Allaire G (2018). R Markdown: The Definite Guide, 1st edition. CRC Press. [↑](#footnote-ref-103)
14. Kuhn M, Johnson K (2019). Feature Engineering and Selection: A Practical Approach for Predictive Models. (CRC PRess.) [↑](#footnote-ref-108)
15. 我们也承认，这里的术语对于不同的人意味着不同的理解，二者之间也是统一的。但是，本书有一定的定义。 [↑](#footnote-ref-109)
16. Kim AY, Escobedo-Land A (2015). “OKCupid data for introductory statistics and data science courses.” Journal of Statistics Education, 23(2). [↑](#footnote-ref-110)
17. Greenacre M (2017). Correspondence analysis in practice. Chapman and Hall/CRC. [↑](#footnote-ref-115)
18. Zou H, Hastie T (2005). “Regularization and variable selection via the elastic net.” Journal of the royal statistical society: series B (statistical methodology), 67(2), 301–320. [↑](#footnote-ref-125)
19. 编写本书的时候，MLeap还不支持Spark 2.4。 [↑](#footnote-ref-143)
20. Codd EF (1970). “A relational model of data for large shared data banks.” [↑](#footnote-ref-216)
21. AutoML使用交叉验证，而GLM不使用。 [↑](#footnote-ref-271)
22. Rosenblatt F (1958). “The perceptron: a probabilistic model for information storage and organization in the brain.” Psychological review. [↑](#footnote-ref-278)
23. Minsky M, Papert SA (2017). Perceptrons: An introduction to computational geometry. MIT press. [↑](#footnote-ref-279)
24. Ackley DH, Hinton GE, Sejnowski TJ (1985). “A learning algorithm for Boltzmann machines.” Cognitive science. [↑](#footnote-ref-281)
25. Chollet F, Allaire J (2018). Deep Learning with R. Manning Publications. [↑](#footnote-ref-282)