小白学苑

大数据技术书籍-精编系列

Spark 机器学习

Scala 预览版本

本书专为有一定 Spark 基础，但机器学习知识尚处零基础阶段的读者量身打造。我们深知，许多读者怀揣着利用 Spark 开发机器学习应用的热情，却被 Python 这一在机器学习领域广泛使用的编程语言所阻碍。市面上多数机器学习资料紧密围绕 Python 展开，使得不熟悉 Python 的读者望而却步。本书正是为解决这一痛点而生，我们将带领读者探索基于 Scala 语言的 Spark 的机器学习开发之路，无需过度依赖 Python 机器学习框架，为您扫除学习障碍。

基于 Hadoop 3.x 和 Spark 3.5.x

前　言

在数据驱动的时代，机器学习已成为挖掘数据价值、驱动业务决策的核心工具。然而，对于长期深耕大数据领域的 Spark 开发者而言，一个现实的困境始终存在—机器学习的主流生态几乎被 Python“垄断”。无论是 TensorFlow、PyTorch，还是 Scikit-learn，这些工具链的学习成本迫使许多 Spark 开发者不得不“绕道”Python，甚至陷入“学 Spark 还是学 Python”的两难选择。

为什么要写这本书？

本书的诞生源于一个简单的信念：**Spark 开发者不应被强迫学习另一门语言和框架才能掌握机器学习**。Spark 自身拥有强大的分布式计算能力（Spark MLlib、Spark ML）和日趋完善的机器学习库，完全可以在大数据生态中原生实现从特征工程到模型训练的完整流程。遗憾的是，市面上的资料要么过于分散，要么默认读者已具备 Python 机器学习经验，这让许多 Spark 用户望而却步。

本书以**零机器学习基础**的 Spark 开发者为起点，通过清晰的逻辑、直观的案例和可直接运行的代码，系统性地讲解如何仅用 Spark 构建机器学习流水线。书中所有示例均基于 Scala API 设计，为读者开辟了一条不依赖 Python 机器学习框架，直接基于 Spark 进行机器学习开发的全新路径。这种独特的视角和讲解方式，极大地降低了读者的学习门槛，满足了特定读者群体的迫切需求。

本书特色

1）三阶学习框架，步步为营

每章采用统一结构：

- ✓ 原理速览：用通俗语言解释算法核心思想（如“决策树如何做选择题”“聚类如何物以类聚”），刻意避免复杂数学推导，聚焦直觉理解。
- ✓ Spark 实现：逐行解析 Spark MLlib/ML API，对比不同算法的代码范式（。
- ✓ 实战案例：基于真实数据集（如电商用户行为数据）演示全流程开发，提供“可修改、可调试、可扩展”的代码模板。

2）零公式恐惧，代码优先

- ✓ 全书不要求读者具备高等数学或统计学背景，所有算法均通过通俗地讲解来理解。
- ✓ 例如，用“投票机制”解释集成学习，用“距离丈量”说明 K-Means 等。

3）Zeppelin 交互式开发，所见即所得

- ✓ 所有案例均在 Apache Zeppelin Notebook 中完成，支持代码、可视化图表与文字说明混合编排，降低上下文切换成本。

本书适合谁？

（1）熟悉 Spark 核心 API（RDD、DataFrame）但未接触过机器学习的开发者。

（2）希望直接在 Spark 集群上部署机器学习模型的大数据工程师。

（3）厌倦了在 Python 与 Spark 之间频繁切换，渴望统一技术栈的团队。

你将获得什么？

（1）无需 Python 的机器学习路径：

- ✓ 从线性回归到集成学习算法，再到自然语言处理，再到深度学习，所有算法均使用 Spark MLlib/ML 实现，代码可直接嵌入现有 Spark 作业。
- ✓ 基于 Spark 独有的分布式机器学习特性，突破单机局限。

（2）开箱即用的开发环境：

- ✓ 提供预配置的虚拟机镜像环境，读者无需耗时搭建环境，5 分钟即可运行全书案例。通过本书提供的环境，您可以将更多精力专注于知识的学习与实践，快速上手 Spark 机器学习开发。
- ✓ 无需再为自行搭建复杂的运行环境而苦恼，也不必面对程序运行过程中各种不可测的故障原因，告别“跑不通代码”的挫败感。

（3）聚焦实战的渐进式学习：

- ✓ 每章以真实业务场景切入（如用户分群、异常检测），通过“问题定义-数据准备-模型迭代-部署优化”的全流程演练，培养工程化思维。

序

近年来，大数据与机器学习的融合已从趋势变为刚需。然而，当开发者试图将两者结合时，往往会发现一个割裂的生态—大数据框架（如 Spark）负责数据的“搬运”与“清洗”，而机器学习任务却被“外包”给 Python 生态。这种割裂不仅增加了技术栈的复杂度，更让许多企业面临“数据管线”与“模型管线”难以协同的挑战。

Spark 机器学习的价值，远未被充分挖掘。Spark 的分布式内存计算、原生 DataFrame API 以及对流批一体的支持，本应成为机器学习落地的理想平台。但受限于教程资源的匮乏，许多团队仍停留在“用 Spark 跑 SQL，用 Python 跑模型”的初级阶段。

本书的革新性在于，它重构了机器学习的学习路径：

- ✓ 以 Spark 为母语：从第一行代码开始，所有逻辑均通过 Spark API 表达，无需切换语法思维。
- ✓ 以理解为导向:用通俗的语言和比喻解释抽象的概念,让抽象概念落地为工程直觉。
- ✓ 以交互为桥梁：Zeppelin Notebook 的即时反馈机制，让算法调参从“黑箱操作”变为“可视化实验”，大幅提升学习心流。

更值得称赞的是，本书对“用户体验”的极致关注。预置的环境配置、避坑指南与调试技巧，大幅降低了学习门槛；而贯穿全书的“以数据工程师视角看机器学习”的叙事方式，则让复杂理论变得直观可操作。

如果你是一名 Spark 开发者，渴望在不切换技术栈的前提下解锁机器学习能力；或是一名数据团队负责人，希望统一大数据与机器学习的技术底座—这本书将成为你的“地图”与“工具箱”。它不仅教你如何用 Spark 实现机器学习，更在传递一种理念：真正的数据智能，应生于数据原生之地。

小白学苑大数据平台负责人

2025-4-17

目录

第 1 章 Spark 机器学习库概述

机器学习是人工智能的核心研究领域之一，其主要研究内容是如何利用数据或经验进行学习，改善具体算法的性能。而在大数据分析和大数据挖掘领域，则利用机器学习界提供的技术来分析海量数据，同时机器学习也是数据挖掘中的一种重要工具。

本章将简要介绍机器学习的概念和分类，并解释 Spark 是如何将机器学习由单机扩展到分布式运行，最后介绍 Spark 机器学习库和机器学习管道。

1.1 机器学习概述

机器学习源自人工智能，它不是数据科学的分支。在数据科学领域，机器学习是一种数据分析方法，它可以自动建立分析模型。数据科学只是使用机器学习作为工具，统计和数学才是机器学习算法的基础。实际上，机器学习已经成为计算机数据分析技术的创新源头之一。

1.1.1 什么是机器学习？

机器学习是实现人工智能的一种重要方法，是人工智能领域的一个子领域。人工智能、机器学习（ML）和深度学习（DL）之间的关系：

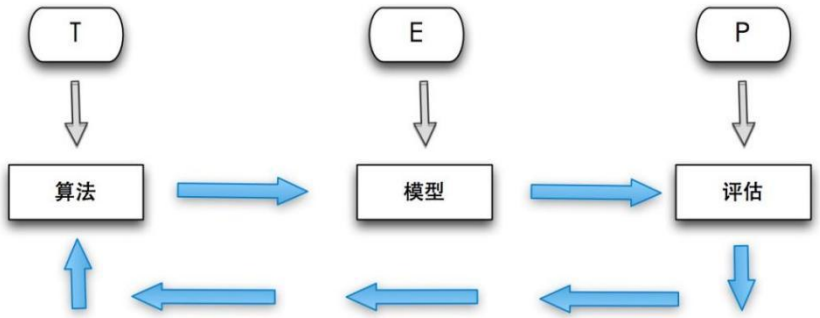
机器学习是一种能够赋予机器学习的能力以此让它完成直接编程无法完成的功能的方法。但从实践的意义上来说，机器学习是一种通过利用数据，训练出模型，然后使用模型预测的一种方法。

一种经常引用的英文定义是：“A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E”。

这段话是机器学习领域中对 “学习” 的一个经典定义，通俗地讲就是：如果一个计算机程序，在某类任务（用 T 表示）上的表现（用 P 衡量）能随着经验（用 E 表示）的增加而提升，那就可以说这个程序能从经验 E 中学习。

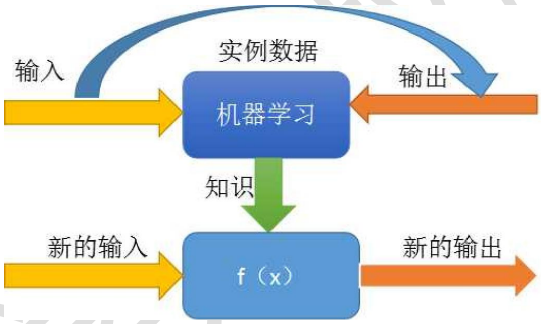
举个例子来帮助理解。假设你要训练一个程序来识别手写数字（这就是任务 T），衡量这个程序性能的指标 P 可以是它识别数字的准确率。为了让程序学习，你会给它大量手写数字的图片和对应的正确数字标签（这就是经验 E）。随着程序不断接触这些图片和标签，

它识别手写数字的准确率（也就是它在任务 T 上的表现，由 P 来衡量）会逐渐提高。当出现这种情况时，就意味着这个程序在从经验 E 中学习。其处理过程可以用下图表示。



从以上解释可以看出，机器学习强调三个关键词：算法、经验、性能。因此，机器学习的过程，就是在数据的基础上，通过算法构建出模型并对模型进行评估。评估的性能如果达到要求，就用该模型来测试其他的数据；如果达不到要求，就要调整算法来重新建立模型，再次进行评估。如此循环往复，最终获得满意的经验来处理其他的数据。

机器学习的本质就是“重现人认识世界的过程”。机器学习就是通过实例数据学习。对于给定的输入，产生一个特定的输出。



机器学习技术和方法已经被成功应用到多个领域，比如数据挖掘、个性化推荐系统、计算机视觉、自然语言处理和机器翻译、生物特征识别、搜索引擎、医学诊断、检测信用卡欺诈、证券市场分析、DNA 序列测序、语音和手写识别、模式识别、智能控制、战略游戏和机器人等领域。

1.1.2 机器学习算法分类

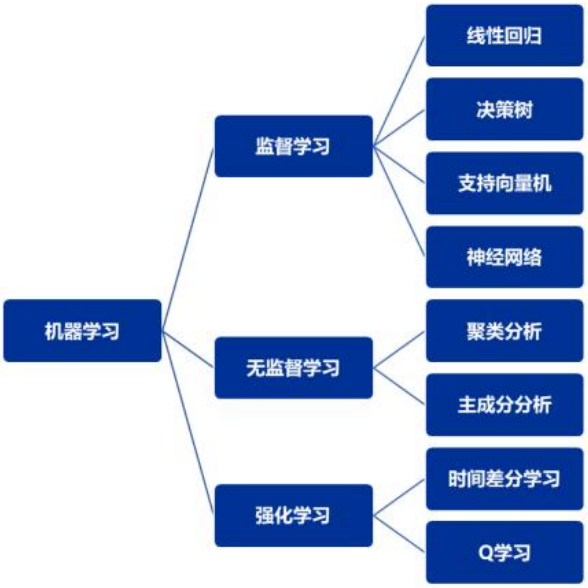
机器学习算法是一类从数据中自动分析获得规律，并利用规律对未知数据进行预测的算法。机器学习算法通常被分为三类：

(1) Supervised Learning（监督学习）：监督算法可以用于预测、估计、分类和其他类似的需求。

(2) Unsupervised Learning（无监督学习）：这样的算法可以从数据学习关系和结构。使用无监督学习技术的例子有聚类和关联规则学习。

(3) Reinforcement learning（强化学习）：与前两种类型的学习不同，这一种学习不会从数据中学习。相反，它通过一系列的操作来学习与环境的交互，而反馈循环提供了它用来进行调整的信息。换句话说，它从自己的经验中学习。

机器学习算法分类表示如下图：



本书主要讲解基于 Spark 集群的监督学习和无监督学习，不涉及强化学习。

注意： Spark 不支持强化学习，所以本书不包括此内容。

按照样本数据是否带有标签值，可以将机器学习算法分为有监督学习和无监督学习。

1. **监督学习**（Supervised Learning）：

监督学习是机器学习中的一个重要分支。它指的是在给定一组包含输入特征（通常是数据）和与之对应的目标标签的训练数据的情况下，构建一个模型，使得该模型能够学习到输入特征和目标标签之间的映射关系，从而对新的、未见过的输入数据预测其对应的目标标签。

监督学习依赖于带有标签的训练数据。例如，在一个图像分类任务中，训练数据可能包含大量的猫和狗的图片，并且每张图片都有一个标签（label），明确指出这张图片是猫还是狗。输入特征就是图片的像素值，目标标签就是“猫”或者“狗”。

模型通过对训练数据的学习，尝试找出输入特征和目标标签之间的规律。例如，在上述图像分类的例子中，模型可能会学习到猫和狗在颜色、形状、纹理等方面的差异，从而建立起从图片像素值到“猫”或“狗”标签的映射关系。

一旦模型学习到了这种映射关系，就可以对新的、未在训练数据中出现过的输入数据进行预测。比如，给模型一张新的猫或狗的图片，它就可以根据学习到的规律判断这张图片是猫还是狗。预测模型训练和预测的过程如下图：

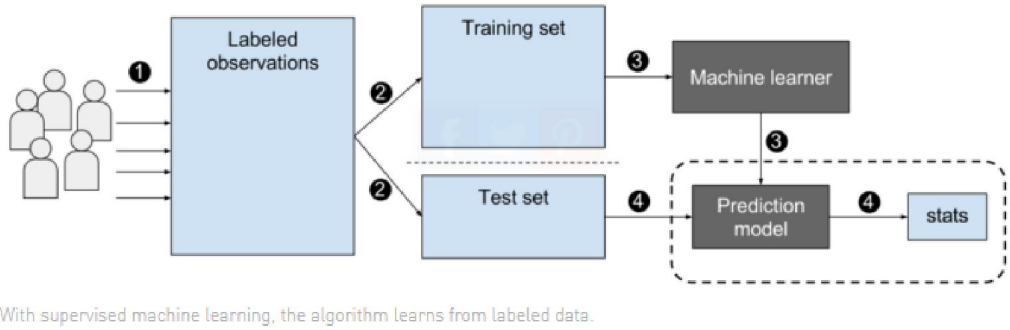
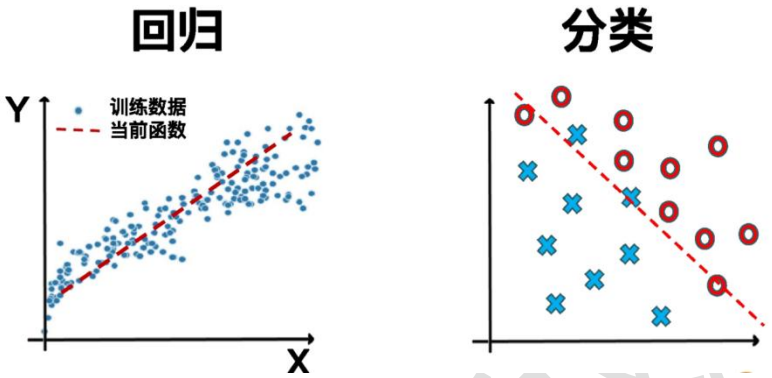


图 1-1　监督式学习：预测模型，“标记”的数据

监督学习算法是机器学习算法家族的一个子集，主要用于预测建模。预测模型基本上是由机器学习算法和训练数据的特征或属性构建的模型，这样我们就可以使用从输入数据中获得的其他值来预测一个值。

常见的监督学习算法包括线性回归（用于预测连续数值，如预测房价）、逻辑回归（用于分类问题，如判断邮件是否为垃圾邮件）、决策树、支持向量机等。

按照标签值的类型，可以将监督学习算法进一步细分为分类（Classification）问题和回归（Regression）问题。



1）分类（Classification）

分类问题是监督学习中的一类任务，在这类任务里，模型的目标是将输入数据划分到预先定义好的不同类别中。这些类别是明确且有限的。例如在手写数字识别任务中，类别就是从 0 到 9 这 10 个数字；在垃圾邮件检测任务中，类别通常分为“垃圾邮件”和“非垃圾邮件”。

分类模型的训练过程就是让模型学习输入特征与类别标签之间的关联。以鸢尾花分类为例，输入特征可能是花瓣长度、花瓣宽度、萼片长度、萼片宽度等，目标是将鸢尾花样本分为不同的品种（如 Setosa、Versicolour、Virginica）。模型会根据大量带有标签的鸢尾花样本数据进行学习，找出特征与品种之间的规律。

简单来说，分类问题要求算法预测离散值。输入数据是一组特征，而输出是一个离散的类别标签。

分类问题在生活中有广泛的应用，像疾病诊断（判断患者是否患有某种疾病）、图像识别（识别图片中的物体类别）、情感分析（判断文本的情感倾向是积极、消极还是中性）等。

2）回归（Regression）

回归问题同样属于监督学习，不过它的目标是预测一个连续的数值。输入是一组特征，输出是一个具体的数值。

与分类问题的离散类别标签不同，回归问题的输出是一个连续的值。例如预测房屋价格，价格可以是任意的实数；预测股票的收盘价，也是一个连续的数值。

在训练回归模型时，模型会尝试找到输入特征与连续输出值之间的函数关系。比如，在预测房价的问题中，输入特征可能包括房屋面积、房间数量、地理位置等，模型会根据大量房屋的这些特征以及对应的实际价格数据进行学习，构建一个能根据输入特征预测房价的函数。

回归问题的应用场景包括金融领域的股票价格预测、气象领域的气温预测、工业领域的产品质量预测等。

2. 无监督学习（Unsupervised Learning）：

无监督学习是机器学习的一个重要分支，它是指在没有明确目标标签（label）的情况下，对输入数据进行分析和建模，旨在发现数据中的内在结构、模式和规律，主要用于模式检测和描述建模。

与监督学习不同，无监督学习中用于训练机器学习算法的数据不带有预先定义好的目标标签（即无标签数据）。例如，在分析客户购买行为数据时，没有预先给每个客户贴上“高价值客户”、“低价值客户”等标签，模型需要自己从数据中去发现潜在的模式，给这些客户的标签将由机器自己设计。

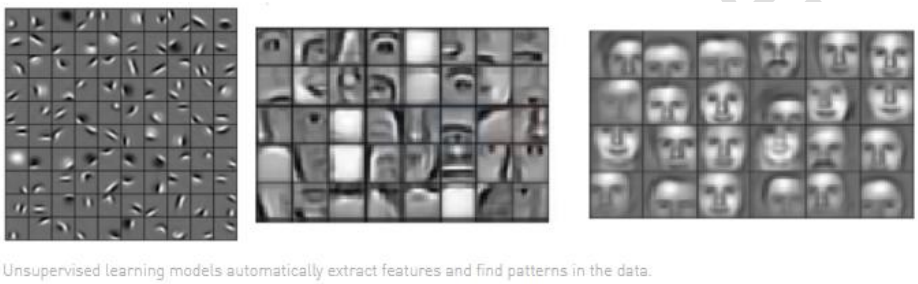


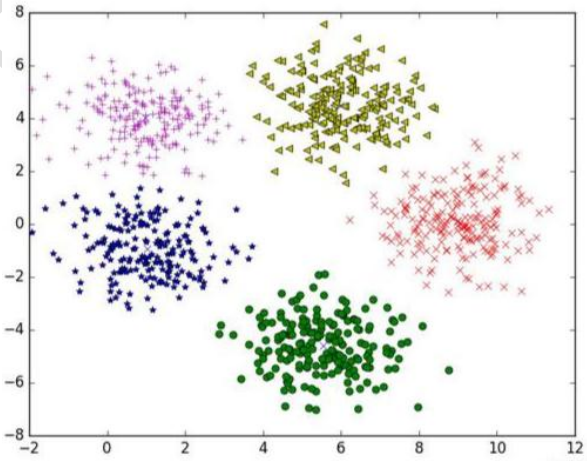
图 1-1　无监督学习：描述性模型，“未标记”的数据

无监督学习对事务性数据的处理效果很好，在很多领域都有广泛的应用。在市场营销中，可以用于客户细分，了解不同客户群体的特点；在生物学中，可以对基因数据进行聚类分析，发现不同基因的功能和关系；在信息检索中，可以对文档进行聚类，提高搜索效率；在金融系统中，银行或保险公司通过寻找客户行为中的异常模式来检测欺诈交易。

常见的无监督学习任务主要包括聚类、降维和关联规则等。

1）聚类

聚类是将数据集中相似的数据点划分为不同的组，也称为簇。如下图所示：

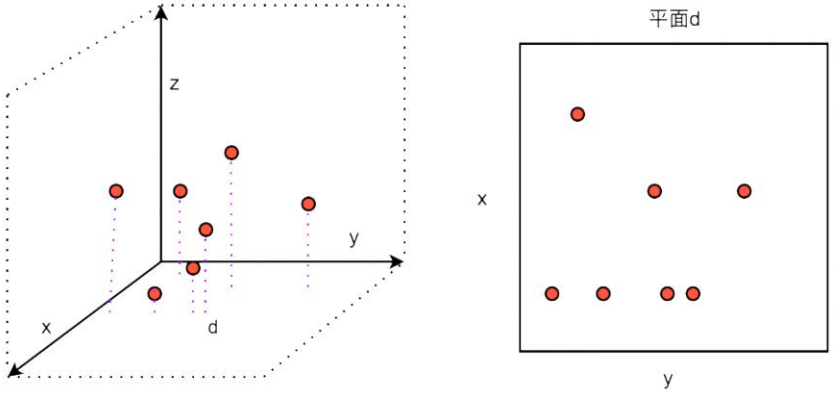


例如，在电商平台中，将具有相似购买偏好的用户聚为一类，以便为不同类别的用户提供个性化的推荐服务。通过聚类，模型能够自动识别出数据中的自然分组，而无需事先知道这些分组的定义。

2）降维

降维是减少数据特征的数量，同时保留数据的主要信息。在处理高维数据时，数据的维度可能非常高，这会增加计算复杂度和过拟合的风险。降维可以帮助简化数据，提高算法的效率和性能。例如，在图像识别中，将高分辨率的图像数据进行降维处理，减少特征数量，同时保留图像的关键信息。

如下图所示，我们将位于三维空间（实例有三个特征 x，y，z）的实例映射到了二维平面，此时在二维平面上实例点通过降低维度抛弃了特征 y，只有两个特征 x，y。这样通过投影就实现了降维。



3）关联规则

关联规则反映了一个事物与其他事物之间的相互依存性和关联性。如果两个或多个事物之间存在一定的关联关系，那么其中一个事物的出现可能会暗示其他事物的出现。用数学语言表达，关联规则通常表示为 X⇒Y 的形式，其中 X 称为前件，Y 称为后件。例如，在超市购物篮分析中，“购买面包⇒ 购买牛奶” 就是一条关联规则，它表明如果顾客购买了面包，那么他很可能也会购买牛奶。

例如，美国沃尔玛超市对一年多的原始交易数据进行了详细的分析，得到一个意外发现：与尿布一起被购买最多的商品竟然是啤酒。这是关联规则在客户关系管理系统中的经典案例：数据样本的某些特性与其他特性相关联。

在数据挖掘领域，关联规则分析是一种发现变量之间有趣关系的技术。这种关系通常被描述为“如果...那么...”的模式，例如，“如果一个顾客购买了方便面，那么他们也很可能会购买火腿肠”。

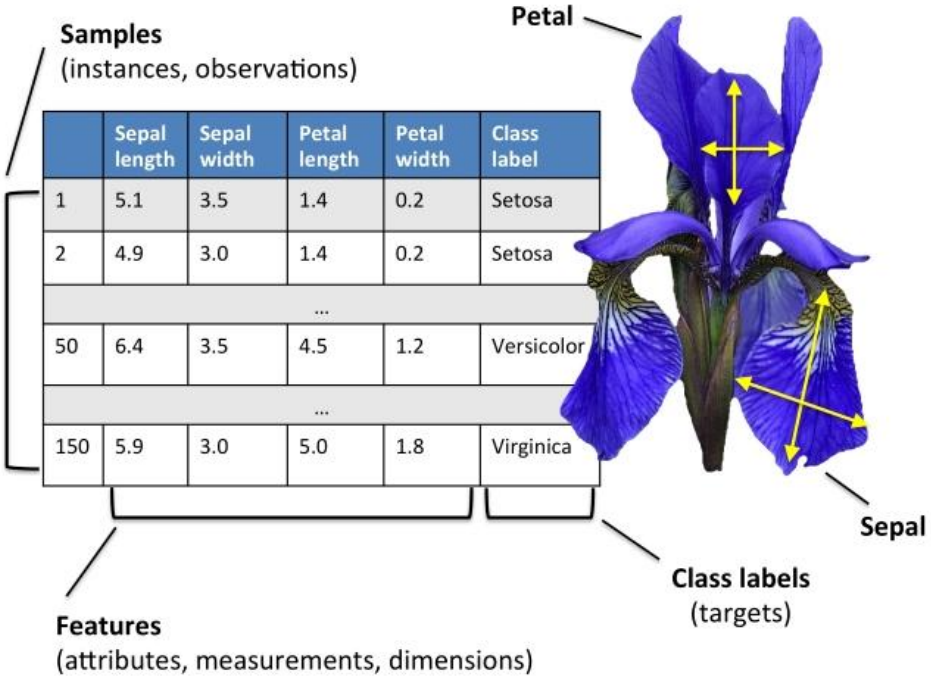
1.1.3 机器学习数据描述

在机器学习里，为了能让机器理解训练示例，通常采用属性来描述数据。属性是数据的特征或者特性，它能够从不同方面刻画数据对象。例如，在描述一个人的时候，可以使用年龄、身高、体重、性别等属性。每个属性都有其特定的值域，比如年龄的值域是正整数，性别的值域通常是“男”或“女”。

buyer_id	buyer_prov	gender	age_range	zodiac
00047f4d5e5c	陕西省	男	20-30岁	白羊座
0014a6246fe1	福建省	男	20岁以下	巨蟹座
0016b8240d96	河北省	女	40-50岁	金牛座
003c9438d00e	辽宁省	女	20岁以下	巨蟹座
0058fe948474	云南省	女	30-40岁	射手座
007cf71ad360	山东省	女	40-50岁	狮子座
00f9ef8aa19e	福建省	男	20-30岁	白羊座

带标签的数据集是指数据集中的每个样本除了包含用于描述其特征的属性外，还带有一个对应的标签（label），这个标签代表了该样本所属的类别或者具有的某种属性。

例如，鸢尾花数据集是一个经典的带标签数据集，它被广泛用于机器学习的教学和研究。该数据集包含了 150 个鸢尾花样本，分为 3 个不同的品种（类别），每个品种有 50 个样本。如下图所示：



在这个数据集中，每一行代表一个鸢尾花样本，前四列的属性描述了鸢尾花的特征，最后一列的标签表明了该鸢尾花所属的品种。具体属性如下：

（1）萼片长度（sepal length）：指鸢尾花萼片的长度，用厘米作为度量单位。

（2）萼片宽度（sepal width）：即鸢尾花萼片的宽度，单位同样是厘米。

（3）花瓣长度（petal length）：表示鸢尾花花瓣的长度，以厘米为单位。

（4）花瓣宽度（petal width）：也就是鸢尾花花瓣的宽度，单位为厘米。

标签则是鸢尾花的品种（class label），具体分为山鸢尾（Setosa）、变色鸢尾（Versicolor）和维吉尼亚鸢尾（Virginica）。

通过这些属性和标签，机器学习模型可以学习到不同品种鸢尾花在特征上的差异，从而对新的鸢尾花样本进行分类预测。

1.1.4 机器学习过程

为了有效地将机器学习应用于智能应用程序的开发，应该考虑采用一组大多数机器学习实践者所遵从的最佳实践，它包含了一系列可重复性和一致性的步骤。

一般来说，机器学习任务包括以下步骤：

(1) 读取数据。

(2) 对数据进行预处理或准备。

(3) 提取特征。

(4) 为训练、验证和测试拆分数据。

(5) 使用训练数据集训练模型。

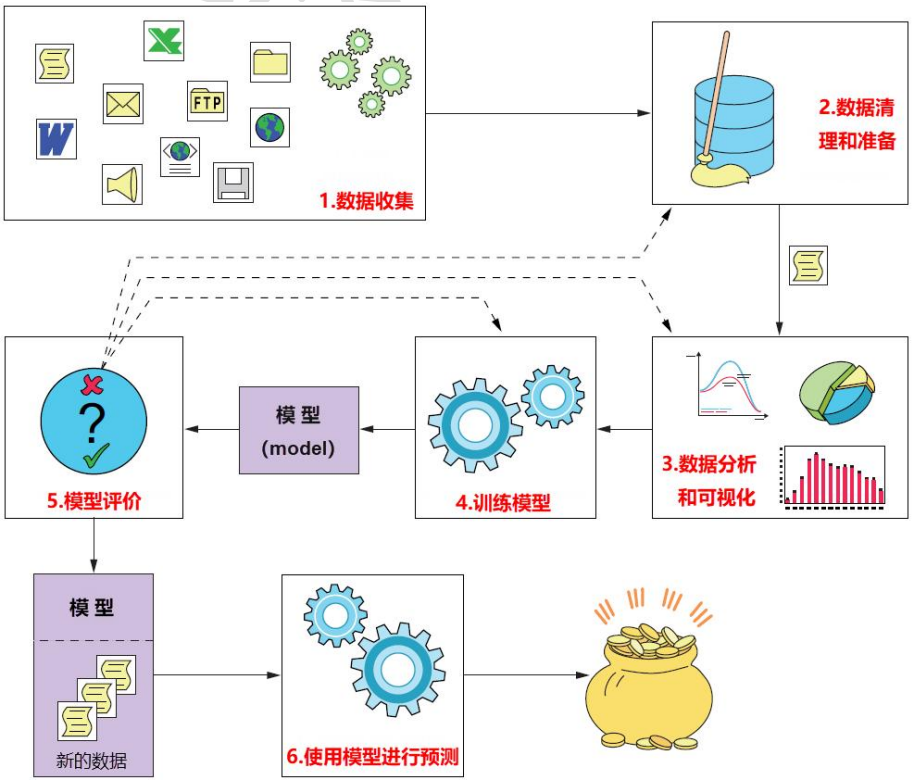
(6) 使用交叉验证技术优化模型。

(7) 在测试数据集上评估模型。

(8) 部署模型。

每个步骤代表机器学习工作流（俗称管道）中的一个阶段。Spark ML 为这些阶段提供了抽象表示。Spark ML 引入的关键抽象包括 Transformer、Estimator、Pipeline、Parameter Grid、CrossValidator 和 Evaluator。

这个例子说明了一个机器学习项目由多个步骤组成。典型的步骤如下图所示：



机器学习应用程序开发过程通常包括以下步骤：

(1) 数据收集：典型的机器学习管道收集的输入数据来自多个数据源，通常有着不同的数据格式。这些源可以包括文件、数据库(RDBMS、NoSQL、Graph 等)、Web 服务(例如 REST 端点)、Kafka 和 Amazon Kinesis 流等。

(2) 数据清洗和预处理：数据清洗是整个数据分析管道中的关键步骤。通常包括过滤数据，处理丢失、不完整或损坏的数据，处理潜在的异常、错误和异常值，将不同的数据源连接在一起，聚合数据等。该预处理步骤修复了数据质量问题，使之适合机器学习模型使用。

(3) 特征工程：在这一步中，我们使用各种技术从输入数据中提取和生成特定的特征，然后将它们组合成一个特征向量，并传递到流程的下一个步骤。通常，使用 VectorAssembler 从指定的 DataFrame 列创建特征向量。大多数机器学习算法都是基于特征的，这些特征通常是用于模型的输入变量的数值表示。

(4) 模型训练：机器学习模型训练包括指定一个算法和一些训练数据(模型可以从中学习)。通常，我们将输入数据集分为训练数据集和测试数据集，为每个数据集随机选择一定比例的输入记录。模型通过在训练数据集上调用 fit()方法进行训练。

(5) 模型验证：这一步包括评估和优化 ML 模型，以评估预测的好坏。在此步骤中，使用 transform()方法将模型应用于测试数据集，并计算模型的合适的性能度量，例如精度、错误等。

(6) 模型选择：在此步骤中，我们不断尝试和调整 Transformer 和 Estimator 的超参数，以生成最优的 ML 模型。通常，我们创建一个参数网格，并使用称为交叉验证的过程对给定模型最合适的参数集执行网格搜索。交叉验证器返回的最佳模型可以保存并稍后加载到生产环境中。

(7) 模型部署：最后，我们部署用于生产的最优模型。部署的模型需要在生产环境中持续维护、升级、优化等等。

1.2 Spark 机器学习库

机器学习模型是使用算法和数据的组合来建立的。使用强大的算法是至关重要的，但同样重要（有些人可能认为更重要）的是使用大量高质量数据进行训练。

一般来说，数据越多，机器学习的表现就更好。2001 年，微软研究人员 Michele Banko 和 Eric Brill 在他们颇具影响力的论文《自然语言消歧扩展到非常非常大的语料库》中首次提出了这个概念。谷歌的研究主任 Peter Norvig 在他的论文“数据的不合理有效性”中进一步推广了这个概念。

Apache Spark 的所有优势也由此扩展到了机器学习领域。Apache Spark 最重要的是它的分布式特性，它使用户能够以足够的速度在非常大的数据集上训练和应用机器学习算法。Apache Spark 的另一个优势是它的统一性：它提供了一个执行大多数任务的平台。用户可以在同一个系统中，并使用相同的 API，进行数据的收集、准备和分析，并对模型进行训练、评估和使用。

Apache Spark 提供了最流行的机器学习算法的分布式实现，并且不断添加新的算法，称为 Spark 机器学习库。Spark 机器学习库通过提供一组常用的机器学习算法和一组工具来简化机器学习的可扩展性和易用性，以方便构建和评估机器学习模型的过程。

Apache Spark 当前有两个机器学习库：基于 RDD 的 MLlib 库，和基于 DataFrame 的 ML 库。MLlib 是基于 RDD 的，从 Spark 0.8 开始被包含进来，并被开源社区不断扩展和开发。Spark MLlib 目前处于维护状态。

从 Spark 1.2 开始，引入了一个名为 Spark ML 的新机器学习库。新 Spark ML 库广泛地使用 DataFrame 对象来表示数据集，并且提供了许多内置的功能，包括特征提取器、转换器、选择器和机器学习算法，如分类、回归和聚类算法。为了使实际的机器学习变得简单，Spark ML 库提供了一组抽象来帮助简化数据清理、特征工程、模型训练、模型调优和评估的步骤，并将它们组织成一个管道，使其易于理解、维护和重复。

注意：Apache Spark 依赖于几个低级的库来执行优化的线性代数操作。这些库包括用于 Scala 和 Java 的 Breeze 和 jblas，以及 Python 的 NumPy。

Apache Spark ML 机器学习库支持如下这些机器学习算法：

1）协同过滤

Alternating Least Squares (ALS)算法：协同过滤是推荐系统中常用的一种方法。这些技术旨在填补用户项关联矩阵中缺失的条目。spark.mllib 目前支持基于模型的协同过滤。在这个实现中，user 和 item 由一组可以用来预测缺失条目的潜在因素来描述。sprk.mllib 使用 ALS 算法学习这些潜在因素。

2）聚类

在 Spark 中实现了以下聚类技术：

(1) k-means：这是一种常用的聚类算法，它将数据点聚类到预定义的集群数量中。集群的数量由用户决定。spark.mllib 实现包括 k-means++方法的并行变体。

（参考 http://theory.stanford.edu/~sergei/papers/vldb12-kmpar.pdf）

(2) Gaussian mixture：高斯混合模型（Gaussian Mixture Model，GMM）表示从 k 个高斯子分布中选取点的复合分布。每个分布都有它自己的概率。spark.mllib 实现使用期望最大化算法在给定一组样本的情况下归纳出最大似然模型。

(3) Power Iteration Clustering (PIC)：这是一个可伸缩的算法，用于根据给定的一对相似的边属性来聚类图的顶点。它使用权力迭代（power iteration）计算图的(归一化的关联矩阵)伪特征向量。spark.mllib 包括一个使用 GraphX 的 PIC 实现。它采用元组的 RDD 并输出带有集群分配的模型。相似性必须是非负的。PIC 假设相似度量是对称的(在统计学中，相似性度量或相似性函数是一个实值函数，它量化了两个对象之间的相似性。这些度量是距离度量的逆;余弦相似度就是一个例子)。无论顺序如何，一对(srcId、dstId)在输入数据中最多出现一次。

(4) Latent Dirichlet Allocation (LDA)：这是主题模型的一种形式，它从文本文档集合中推断主题。LDA 是一种聚类算法的形式。以下几点说明了主题：a) 主题是集群中心，文档对应于数据集中主题和文档都存在于特征空间中的示例，其中特征向量是单词计数的向量(也称为单词包)；b) LDA 不是使用传统的距离方法估计集群，而是使用基于文本文档生成模型的函数。

(5) Bisecting k-means：这是一种层次聚类。层序聚类分析（Hierarchical Cluster Analysis，HCA）是一种聚类分析方法，它构建了一个自顶向下的集群层次结构。在这种方法中，所有的观察都从一个集群开始，当一个集群向下移动时递归地执行分割。层次聚类是聚类分析中常用的方法之一，其目的是构建一个聚类的层次。

(6) Streaming k-means：当数据到达流中时，我们希望动态地估计集群，并在新数据到达时更新它们。spark.mllib 支持 streaming k-means 聚类，参数控制估计的衰减。该算法采用了微批 k-means 更新规则的泛化。

3）分类

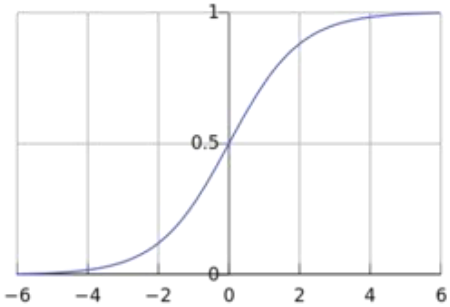
在 Spark 中实现了以下分类技术：

(1) Decision Trees：决策树及其集合是分类和回归的方法之一。决策树很受欢迎，因为它们易于解释、处理分类特性并扩展到多类分类设置。它们不需要特征缩放，也能够捕获非线性和特征交互。树集成算法、随机森林和增强算法在分类和回归场景中表现最好。spark.mllib 实现了用于二元和多类分类以及回归的决策树。它支持连续和分类特征。该实现按行分区数据，允许使用数百万个实例进行分布式训练。

(2) Naive Bayes：朴素贝叶斯分类器是基于应用贝叶斯定理(查看贝叶斯定理)，在特征之间具有强(朴素)独立性假设的一类简单概率分类器。Naive Bayes 是一种多类分类算法，假设每一对特征之间都是独立的。该算法在一次训练数据的传递中，计算出给定标签的每个特征的条件概率分布，然后应用贝叶斯定理计算出给定观察值的标签的条件概率分布，然后用于预测。spark.mllib 支持多项式朴素贝叶斯和伯努利朴素贝叶斯。这些模型通常用于文档分类。

(3) Probability Classifier：在机器学习中，Probability Classifier（概率分类器）是这样一种分类器：给定一个输入，它可以预测一组类的概率分布，而不是输出样本应该属于的最有可能的类。概率分类器提供了一些确定性的分类，这些确定性本身或在将分类器组合成整体时非常有用。

(4) Logistical Regression：这是一种用来预测二元响应的方法。Logistic 回归通过使用逻辑函数估计概率来度量分类因变量和自变量之间的关系。这个函数是一个累积逻辑分布。逻辑函数如下图所示：



它是广义线性模型(GLM)的一种特殊情况，用于预测结果的概率。GLM 被认为是线性回归的一种泛化，它允许有误差分布而非正态分布的响应变量。

(5) Random Forest：该算法利用决策树的集合来确定决策边界。随机森林结合了许多决策树。这降低了过拟合结果的风险。Spark ML 支持用于二元分类和多类分类以及回归的随机森林。它可以用于连续值或分类值。

4）降维

降维（Dimensionality reduction）是一个减少机器学习变量数量的过程。它可以用于从原始特征中提取潜在特征，或者在维护整体结构的同时压缩数据。MLlib 在 RowMatrix 类的基础上提供了降维支持。

(1) Singular value decomposition (SVD)：矩阵 M 的奇异值分解：M x n(实数或复数)是形式 U Σ V *的一个因式分解，其中 U 是一个 m x R 矩阵。 Σ 是一个 R x R 矩形对角矩阵（对角非负实数），V 是一个 n x r 单式矩阵。r 等于矩阵 M 的秩。

(2) Principal component analysis (PCA)：这是一种统计方法，用于在第一个坐标中找到最大方差的旋转。每一个后续坐标都有最大的可能方差。旋转矩阵的列称为主分量。主成分分析在降维中得到了广泛的应用。MLlib 使用 RowMatrix 支持对存储在面向行格式中的所有瘦矩阵应用 PCA。

5）频繁模式挖掘

频繁模式挖掘（Frequent pattern mining）。

(1) FP-growth：FP 代表频繁模式。算法首先计算数据集中出现的项(属性和值对)，并将它们存储在头表中。在第二步中，该算法通过插入实例(由 items 组成)来构建 FP 树结构。每个实例中的 items 项按照它们在数据集中出现的频率降序排序；这确保了可以快速处理树。每个实例中不满足最小覆盖阈值的 items 项将被丢弃。对于许多实例共享最频繁 items 项的用例，FP 树提供了接近树根的高压缩。

(2) Association rules：关联规则学习是在大型数据库中发现变量之间有趣关系的一种机制。它实现了一个并行规则生成算法，用于构造以单个 item 项作为结果的规则。

6）PrefixSpan

PrefixSpan：这是一个顺序模式挖掘算法。

7）Evaluation metrics

Evaluation metrics：spark.mllib 提供了一套用于评估算法的度量标准。

8）PMML 模型导出

PMML model export：预测模型标记语言（Predictive Model Markup Language，PMML）是一种基于 xml 的预测模型交换格式。PMML 为分析应用程序描述和交换机器学习算法生成的预测模型提供了一种机制。spark.mllib 允许将其机器学习模型导出到 PMML 及其等价的 PMML 模型。

9）优化

Optimization (Developer)

Stochastic Gradient Descent：这是用来优化梯度下降，以最小化一个目标函数；这个函数是可微函数的和。梯度下降法和随机次梯度下降法(Stochastic Subgradient Descent，SGD)是 MLlib 中的低级原语，在此基础上发展了各种 ML 算法。

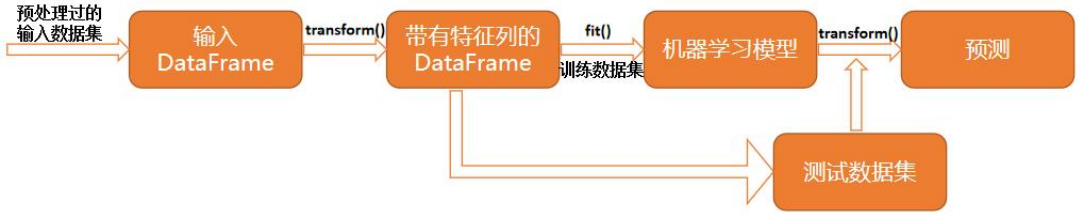
10）L-BFGS

Limited-Memory BFGS (L-BFGS)：这是一种优化算法，它使用有限的计算机内存。用于机器学习中的参数估计。它属于近似 Broyden-Fletcher-Goldfarb-Shanno(BFGS)算法的准牛顿方法族。BFGS 方法近似于牛顿方法，牛顿方法是一类寻找函数平稳点的爬山优化技术。对于这类问题，一个必要的最优条件是梯度应该为零。

此外，Spark 还支持使用 TF-IDF、ChiSquare、Selector、Normalizer 和 Word2Vector 进行特征提取与转化。

1.3 Spark 机器学习管道

机器学习流程本质上是一个由一系列以顺序方式运行的步骤组成，相同的步骤经常以相同的顺序重复出现，通常需要重复几次才能到达一个最佳的模型。实现机器学习模型的步骤以及每个阶段的操作和输入/输出如下图所示：



Spark 1.2 引入了机器学习管道 API，包括 Transformer、Estimator 和 Evaluators。Spark 使用 Transformer 来实现将一个数据集转换成另一个数据集的机器学习组件。Estimators 通过在数据集上进行拟合来生成 Transformer。Evaluator 用来评估模型的性能。它们可以组合成管道（Pipeline）。机器学习管道 API 简化了数据分析和机器学习应用程序的实现。

1）机器学习管道

在 Spark 中，使用 Transformer 来实现将一个数据集转换成另一个数据集。Spark ML 中的机器学习模型就是一个 Transformer，它们通过添加预测来转换数据集。实现这种转换的主要方法是 transform()，它接受一个 DataFrame 和一个可选的参数集。

Estimator 通过在数据集上进行拟合来生成 Transformer。例如，线性回归算法产生了一个线性回归模型，该模型具有拟合的权重和截距，它就是一个 Transformer。使用 Estimator 的主要方法是 fit()，该方法也需要一个 DataFrame 和一组可选的参数。

Evaluator 根据一个度量来评估模型的性能。例如，RegressionEvaluator（回归模型的 Evaluator）可以使用 RMSE 和 MSE 等作为模型的评价指标。

下图显示了 Transformer、Estimator 和 Evaluator 的操作方法以及它们的输入和输出。

在 Spark ML 中，机器学习工作流被实现为由一系列阶段组成的管道。每个管道阶段由按指定顺序执行的 Transformer 或 Estimator 组成。下图描述了机器学习过程中的核心步骤和 Spark MLlib 提供的主要组件之间的相似性。

机器学习主要步骤　　Spark ML管道主要步骤

在机器学习中，通常运行一系列步骤来清理和转换数据，然后训练一个或多个 ML 算法来从数据中学习，最后调整模型以获得最佳的性能。Spark ML 中的管道（pipeline）抽象设计是为了使这个工作流更易于开发和维护。

从技术的角度来看，Spark ML 有一个名为 Pipeline 的类，它被设计用来管理一系列的阶段，每一个阶段都由 PipelineStage 来表示。一个 PipelineStage 既可以是 Transformer，也可以是 Estimator。抽象 Pipeline 是一种 Estimator。管道以指定的顺序连接多个 Transformer 和 Estimator，形成机器学习工作流。从概念上讲，它将机器学习工作流中的数据预处理、特征提取和模型训练步骤链接在一起。

2）Transformer

Transformer 是一个算法，用来将一个输入 DataFrame 转换到另一个 DataFrame，并向其添加一个或多个特征。Transformers 的设计目的是通过在特征工程步骤和模型评估步骤中操作一个或多个列来转换 DataFrame 中的数据。转换过程是在构建特征的上下文中，这些特征将被 ML 算法用于学习。这个过程通常涉及添加或删除列（特征），将列值从文本转换为数值，或者规范化某一列的值。

从技术的角度来看，Transformer 有一个 transform()方法，它在的原始数据的输入列上执行转换，结果存储在输出列中。在 Transformer 的构造过程中可以指定输入列和输出列名称。如果没有指定，则使用默认列名("inputCol","outputCol")。

Spark ML 提供两种类型的 Transformers：特征 Transformer 和机器学习模型 Transformer。

一个特征 Transformer 通过对输入数据集中的一个列应用转换来创建一个或多个新列，并返回一个附加了新列的新 DataFrame。例如，如果输入数据集有一个包含句子的列，则可以使用特征 Transformer 将句子拆分为单词，并创建一个新的列，该列将单词存储在数组中。

一个模型 Transformer 代表一个机器学习模型。它接受一个 DataFrame 作为输入，并输出带有每个输入特征向量的预测标签的新 DataFrame。输入数据集必须具有包含特征向量的列。模型读取包含特征向量的列，为每个特征向量预测一个标签，并返回一个新的 DataFrame，其中预测标签作为新列附加。

下图描述了 Transformer 的工作模式，DF1 中的阴影部分表示输入列，DF2 的阴影部分表示输出列。

Transformer 以某种方式转换原始数据。这可能是创建一个新的交互变量，或者规范化一列，或者简单地将一个整数转换为 Double 类型，以便输入到模型中。

Transformer 主要用于机器学习流程中的数据预处理和特征工程阶段。

3）Estimator

一个 Estimator 代表了一种机器学习算法，用来在训练数据集上训练或拟合机器学习模型。它实现了一个名为 fit()的方法，该方法接受一个 DataFrame 作为参数并返回一个机器学习模型。

Estimator 所代表的算法可分为两类，一类是用于机器学习的算法，另一类是一种用数据初始化的 Transformer。例如，称为 LinearRegression 的线性回归算法就属于第一种类型，它的 fit()方法返回一个 LinearRegressionModel 类的实例，可用于诸如预测房价等回归任务。而 StringIndexer 就属于第二种类型，它用来将一列的分类值编码成索引，这样每个分类值的索引值都是基于它出现在 DataFrame 的整个输入列中的频率。

从技术的角度来看，一个 Estimator 有一个名为 fit()的方法，它在输入列上应用一个算法，结果被封装在一个叫做 Model 的对象类型中，它是一个 Transformer 类型。输入列和输出列名称可以在 Estimator 的构造过程中指定。下图描述了一个 Estimator 及其输入和输出。

4）模型评估

人们需要评价各种机器学习算法和模型的好坏，以进行比较，因此，需要定义衡量模型精度的指标。比较算法的优劣需要使用算法的评价指标有：

（1）对于分类问题，常用的评价指标是准确率，它定义为测试样本集中被正确分类的样本数与测试样本总数的比值；

（2）对于回归问题，常用的评价指标是回归误差，定义为预测函数输出值与样本标签值之间的均方误差。二分类问题由于其特殊性，我们为它定义了精度与召回率指标，在此基础上可以得到 ROC 曲线。对于多分类问题，常用的评价指标是混淆矩阵。

对于二分类问题，我们可以调整分类器的阈值（即灵敏度）从而得到不同的分类结果（计算各种假阳率下对应的真阳率）。将各种灵敏度下的准确率指标连成一条曲线，就是 ROC 曲线（Receiver Operator Characteristic Curve，接收者操作曲线）。

随着阈值的增大，被判定为正样本的样本数会增加，因此，真阳率会提高；但同时负样本被判定为正样本的数量也会增加，假阳率也会上升。

多分类问题的准确率可以用混淆矩阵定义。对于 k 分类问题，混淆矩阵为 $k \times k$ 的矩阵，它的元素 c_{ij} 表示第 i 类样本被分类器判定为是 j 类的数量：

$$\begin{bmatrix} c_{11} & \cdots & c_{1k} \\ \vdots & \vdots & \vdots \\ c_{k1} & \cdots & c_{kk} \end{bmatrix}$$

如果所有样本都被正确分类，则该矩阵为对角矩阵。因此，对角线的值越大，分类器的准确率越高。

除了上面定义的这些通用指标，对于某此特定的问题还有特定的精度指标，例如，机器翻译、图像分割等问题都有自己的评价指标，在此就不展开了。

5）模型调优

监督学习训练模型的目标模型是在训练集上误差最小化。但是在训练集上学习得到的模型能否有效地用于测试集，衡量指标为泛化能力（泛化能力是指模型从训练集推广到测试集的能力）。泛化能力是衡量监督学习算法的核心标准。

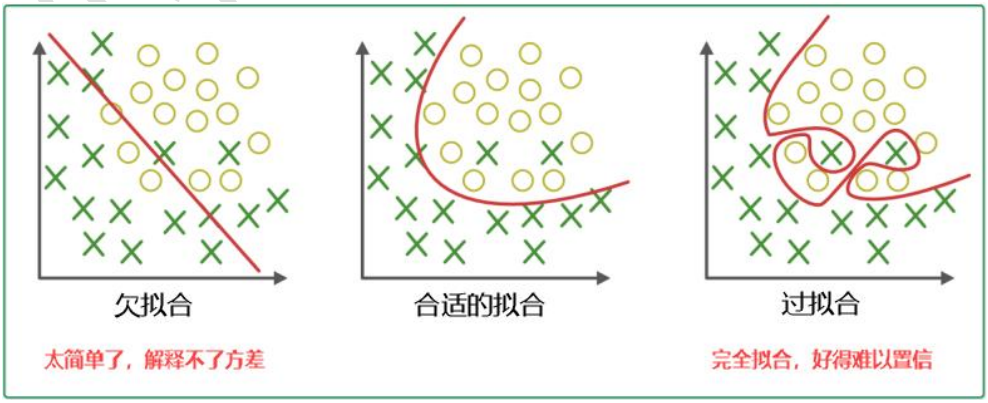
泛化能力好的模型，能够捕捉到数据中的规律和模式，以便对新的、未见过的数据进行准确的预测，这称为“拟合（fitting）”。这种模型能够尽可能准确地描述输入数据和输出标签之间的关系。本质上，就是通过调整模型的参数，使得模型对训练数据的预测结果与实际标签尽可能接近。

模型泛化能力差的情况下又分为欠拟合和过拟合两种情况。

（1）欠拟合是指模型没有足够的能力捕捉到数据中的复杂关系，导致模型在训练数据和新数据上的表现都不佳。简单来说，就是模型过于简单，无法充分学习到数据中的规律。

（2）过拟合是指模型在训练数据上表现得非常好，能够几乎完美地拟合训练数据，但在新的、未见过的数据上表现却很差。也就是说，模型过度学习了训练数据中的噪声和细节，而没有真正掌握数据的一般规律。表现为模型在训练集上的准确率很高，但在测试集或实际应用中的准确率明显下降。

下图列举了欠拟合、拟合、过拟合这三种情况。



模型的泛化误差可以分解成偏差（bias）和方差（variance）。模型的总体误差可以分解为偏差的平方与方差之和。

偏差是模型本身导致的误差，即错误的模型假设所导致的误差，它是模型的预测值的数学期望和真实值之间的差距。高偏差意味着模型本身的输出值与期望值差距很大，因此会导致欠拟合问题。

方差是由于对训练样本集的小波动敏感而导致的误差。它可以理解为模型预测值的变化范围，即模型预测值的波动程度。高方差意味着算法对训练样本集中的随机噪声进行建模，从而出现过拟合问题。

如果模型过于简单，一般会有大的偏差和小的方差；反之，如果模型复杂则会有大的方差但偏差很小。这是一对矛盾，因此需要在偏差和方差之间做一个折中。

数学期望是加权平均值的抽象，是随机变量在概率意义下的均值，即试验中每次可能的结果乘以其结果概率的总和。对于离散型随机变量 x，数学期望定义为：

$$\mathrm{E}(x) = \sum x_i p(x_i)$$

方差定义为：

$$D(x) = \sum (x_i - E(x))^2 p(x_i)$$

推广到连续型的情况，假设有一个连续型随机变量 x 的概率密度函数是 f(x)，其数学期望定义为：

$$\mathrm{E}(x) = \int_{-\infty}^{+\infty} x\mathrm{f}(x)\mathrm{d}x$$

根据定积分的定义，可以看到，连续型就是离散型的极限情况。对于连接型随机变量，方差定义为：

$$\mathrm{D}(x) = \int_{-\infty}^{+\infty} (x - E(x))^2 f(x)dx$$

方差反映的是随机变量取值变化的程度。方差越小，随机变量的变化幅度越小，反之亦然。

在训练过程中，经常会出现过拟合的问题，就是模型可以很好的匹配训练数据，却不能很好在预测训练集外的数据。过拟合是监督学习算法长期以来需要解决的一个问题。

正则化技术是解决过拟合问题的一种常见方法，例如岭回归算法。

监督学习算法训练的目标是最小化损失函数（也叫误差函数）。在损失函数的类型选定之后，人们能控制的只有函数的参数。为了防止过拟合，可以为损失函数加上一个惩罚项，对复杂的模型进行惩罚，强制让模型的参数值尽可能小以使得模型更简单。惩罚项部分被称为正则化项。

正则化项可以使用 L2 范数（即平方和），也可以使用其他范数（如 L1 范数，即绝对值之和）。L2 范数在求解最优化问题时计算简单，而且有更好的数学性质。与 L2 相比，L1 正则化能更有效地让参数趋向于 0，产生的结果更稀疏。例如，岭回归算法就是带 L2 正则化项的线性回归算法，而 LASSO 回归算法就是带 L1 正则化项的线性回归算法。

除了直接加上正则化项之外 ，还有其他强制让模型变简单的方法，如决策树的剪枝算法、神经网络训练中的 dropout 技术、提前终止技术等。

第 2 章 Spark 数据探索和预处理

现实生活中，数据并非完美的，需要进行清洗才能进行后面的数据分析。数据清洗主要是删除原始数据集中的无关数据、重复数据，平滑噪声数据，处理缺失值、异常值等。

数据清洗是整个数据分析过程的第一步，也是整个数据分析项目中最耗费时间的一步。数据清洗的过程决定了数据分析的准确性。

2.1 读取数据源

Apache Spark 提供了一个接口 DataFrameReader，用来以各种格式（如 JSON、CSV、Parquet、Text、Avro、ORC 等）从众多的数据源读取数据到 DataFrame。同样，要将 DataFrame 以特定格式写回数据源，Spark 使用 DataFrameWriter 接口。

加载存储系统中的文件到 DataFrame，可通过 SparkSession 的 read 字段，它是 DataFrameReader 的一个实例。它有个 load()方法，可直接从配置的数据源加载数据。另外它还有五个快捷方法：text()、csv()、json()、orc()和 parquet()，相当于先调用 format()方法再调用 load()方法。

为了演示如何加载外部数据源到 DataFrame，本节使用 Spark 自带的各种格式的数据源文件。这些数据源文件位于$SPARK_HOME/examples/src/main/resources/目录下。首先将它们上传到 HDFS 的/data/spark/resources/目录下。在命令行终端执行 hdfs shell 命令，将文件上传到 HDFS，命令如下：

```
$ hdfs dfs -put ~/bigdata/spark3/examples/src/main/resources
/data/spark/resources
```

注意： 本书中所有代码及相关资源、路径，均基于小白学苑的 PBLP（个人大数据学习环境）进行测试和运行。如果读者在自己的大数据环境下运行，请自行修改代码中相关内容。

2.1.1 文本文件数据源

文本文件是最常见的数据存储文件。Spark DataFrame API 允许开发者读取文本文件的内容到一个 DataFrame 中。例如，读取 Spark 自带的文本文件 people.txt 到一个 DataFrame 中，实现代码如下。

```
val file = "/data/spark/resources/people.txt"
val txtDF = spark.read.format("text").load(file)        // 加载文本文件
// val txtDF = spark.read.text(file)            // 等价于上一句，快捷方法

txtDF.printSchema                                       // 打印 schema
txtDF.show                                              // 输出
```

执行以上代码，输出内容如下：

```
root
 |-- value: string (nullable = true)

+-----------+
|      value|
+-----------+
|Michael, 29|
```

```
|   Andy, 30|
| Justin, 19|
+-----------+
```

Spark 会自动推断出模式，并相应地创建一个单列（列名为 value）的 DataFrame。因此，没有必要为文本数据定义模式。不过，当加载大数据文件时，显式定义一个 Schema 要比让 Spark 来进行推断效率更高。

2.1.2 CSV 文件数据源

Spark 支持从 CSV 格式的文件中读取数据到一个 DataFrame 中。在 Spark 3.x 中，加载 CSV 文件是非常简单的。例如，要将 Spark 自带的 people.csv 文件加载到一个 DataFrame 中，实现代码如下。

```
// 数据源文件
val file = "/data/spark/resources/people.csv"

val peopleDF = spark.read.format("csv")
     .option("sep", ";")                          // 字段使用;分隔符
     .option("inferSchema", "true")               // 指定自动推断模式
     .option("samplingRatio", 0.001)              // 根据抽样进行模式推断
     .option("header", "true")                    // 说明有标题行
     .load(file)

// 或者使用快捷方法
/*
val peopleDF = spark.read
     .option("sep", ";")                          // 字段使用;分隔符
     .option("inferSchema", "true")               // 指定模式自动推断
     .option("samplingRatio", 0.001)              // 根据抽样进行模式推断
     .option("header", "true")                    // 说明有标题行
     .csv(file)
*/

peopleDF.printSchema                              // 打印 schema
peopleDF.show()                                   // 显示
```

执行以上代码，输出内容如下：

```
root
 |-- name: string (nullable = true)
 |-- age: string (nullable = true)
 |-- job: string (nullable = true)

+-----+---+---------+
| name|age|      job|
+-----+---+---------+
|Jorge| 30|Developer|
|  Bob| 32|Developer|
+-----+---+---------+
```

在上面的代码中，使用了模式自动推断。对于大型的数据源，指定一个 schema 要比让 Spark 来进行模式推断效率更高。修改上面的代码，明确提供一个 schema，实现代码如下。

```
import org.apache.spark.sql.types._

// 数据源文件
```

```
val file = "/data/spark/resources/people.csv"

// 构造 schema
val fields = Seq(
    StructField("p name", StringType, nullable = true),
    StructField("p age", LongType, nullable = true),
    StructField("p job", StringType, nullable = true)
)
val schema = StructType(fields)

// 读取数据源，创建 DataFrame
val peopleDF = spark.read
     .option("sep", ";")                          // 字段使用;分隔符
     .option("header", "true")                    // 说明有标题行
     .schema(schema)                              // 指定使用的 schema
     .csv(file)

peopleDF.printSchema()                            // 打印 schema
peopleDF.show()                                   // 显示
```

执行以上代码，输出内容如下：

```
root
 |-- p_name: string (nullable = true)
 |-- p_age: long (nullable = true)
 |-- p_job: string (nullable = true)

+------+-----+---------+
|p_name|p_age|    p_job|
+------+-----+---------+
| Jorge|   30|Developer|
|   Bob|   32|Developer|
+------+-----+---------+
```

可以看出，它返回了由行和命名列组成的 DataFrame，该 DataFrame 具有模式中指定的

在使用 SparkSessionc 对象的 read 字段读取 CSV 文件时，可指定的 option 选项参数和值的列表很长，见表 2-1。

表 2-1 加载 CSV 文件时可指定的 option 选项

Option	默认值	说明	引入版本
sep	,	设置单个字符作为每个字段和值的分隔符	v2.0.0
encoding	UTF-8	根据给定的编码类型解码 CSV 文件	v2.0.0
quote	"	设置用于转义引用值的单个字符，其中分隔符可以是值的一部分。如果要关闭引号，需要设置的不是 null，而是一个空字符串	v2.0.0
escape	\	设置用于转义已引用值中的引号的单个字符	v2.0.0
comment	空字符串	设置用于跳过以该字符开头的行的单个字符。默认情况下，它是禁用的	v2.0.0
header	false	是否使用第一行作为列的名称。不支持两行标题。	v2.0.0
inferSchema	false	告诉 Spark 是否尝试基于列值推断列类型（即从数据自动推断输入模式）。它需要额外传递一次数据。	v2.0.0
ignoreLeadingWhiteSpace	false	指示是否跳过正在读取的值中的前导空格	v2.0.0
ignoreTrailingWhiteSpace	false	指示是否跳过正在读取的值中的尾部空格	v2.0.0

nullValue	空字符串	设置 null 值的字符串表示形式。从 2.0.1 开始，这适用于所有受支持的类型，包括字符串类型。	v2.0.0
nanValue	NaN	设置非数字"值"的字符串表示形式	v2.0.0
positiveInf	Inf	设置正无穷值的字符串表示形式	v2.0.0
negativeInf	-Inf	设置负无穷值的字符串表示形式	v2.0.0
dateFormat	yyyy-MM-dd	设置指示日期格式的字符串。自定义日期格式遵循 java.text.SimpleDateFormat 的格式。这适用于日期(date)类型	v2.0.0
timestampFormat	yyyy-MM-dd'T'HH:mm:ss.SSSXXX	设置指示时间戳格式的字符串。自定义日期格式遵循 java.text.SimpleDateFormat 的格式。这适用于时间戳(timestamp)类型	v2.1.0
maxColumns	20480	定义一个记录可以有多少列的硬限制	
maxCharsPerColumn	-1	定义允许读取任何给定值的最大字符数。默认值是 1000000	
mode	PERMISSIVE	允许在解析期间处理损坏记录的模式。它支持以下不区分大小写的模式。 - PERMISSIVE：当遇到损坏的记录时，将其他字段设置为 null，并将格式错误的字符串放入由 columnNameOfCorruptRecord 配置的字段中。要保存损坏的记录，用户可以在用户定义的模式中设置一个名为 columnNameOfCorruptRecord 的字符串类型字段。如果一个模式没有该字段，它将在解析期间删除损坏的记录。当解析后的 CSV tokens 长度小于模式的预期长度时，它会为额外字段设置 null。 - DROPMALFORMED：忽略整个损坏的记录。 - FAILFAST：当遇到损坏的记录时抛出异常	v2.0.0
columnNameOfCorruptRecord		允许重命名新字段，该字段具有由 PERMISSIVE 模式创建的格式错误字符串。这将覆盖 spark.sql.columnNameOfCorruptRecord。 默认值是在 spark.sql.columnNameOfCorruptRecord 中指定的值	v2.2.0
multiLine		解析一条记录，它可能跨越多行	v2.2.0
wholeFile	false	解析一条记录，它可能跨越多行	

2.1.3 JSON 文件数据源

Spark 支持从纯文本文件中读取 JSON 格式的行数据到一个 DataFrame 中。Spark SQL 可以自动推断 JSON 数据集的模式，并将其加载为 DataFrame。这种转换可以在 DataFrame 或 JSON 文件上使用 SparkSession.read.json()完成。

注意： 作为 JSON 文件提供的文件实际上并不是标准意义上的 JSON 文件，而是要求每一行必须包含一个单独的、自包含的有效 JSON 对象。对于常规的多行 JSON 文件，将 multiLine 选项设置为 true。

读取 JSON 数据源文件时，Spark 会从 key 中自动推断模式，并相应地创建一个 DataFrame。因此，没有必要为 JSON 数据定义模式。此外，Spark 极大地简化了访问复杂

JSON 数据结构中的字段所需的查询语法。例如，读取 Spark 自带的 people.json 文件内容到一个 DataFrame 中，实现代码如下。

```
// 数据源文件
// JSON 数据集路径可以是单个文件，也可以是存储文件的目录
val file = "/data/spark/resources/people.json"

// 读取数据源，创建 DataFrame
val df = spark.read.json(file)         // json 解析；列名和数据类型隐式地推断

// schema
df.printSchema()

// 显示
df.show()
```

执行以上代码，输出内容如下：

```
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)

+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

当然，也可以明确指定一个 schema，覆盖 Spark 的推断 Schema，实现代码如下。

```
import org.apache.spark.sql.types._

// 数据源文件
// JSON 数据集路径可以是单个文件，也可以是存储文件的目录
val file = "/data/spark/resources/people.json"

// 创建 schema。字段名称要与 json 对象的 key 名称保持一致
val fields = Seq(
    StructField("name",StringType,nullable = true),
    StructField("age",IntegerType,nullable = true)
)
val schema = StructType(fields)

// 读取数据源，创建 DataFrame，使用自定义的 schema
val df = spark.read.schema(schema).json(file)

// schema
df.printSchema()

// 显示
df.show()
```

执行以上代码，输出内容如下：

```
root
 |-- name: string (nullable = true)
```

```
 |-- age: integer (nullable = true)

+-------+----+
|   name| age|
+-------+----+
|Michael|null|
|   Andy|  30|
| Justin|  19|
+-------+----+
```

如果在加载 JSON 文件时 JSON 格式解析错误，那么默认创建的 DataFrame 的相应各行各列值都设为 null。在下面的示例代码中，在 Schema 中将姓名（name）这一列对应的数据类型错误地设为 boolean 类型，因此 SparkSession 在解析时出现解析错误，因而整个 DataFrame 中的各行各列值均为 null，实现代码如下。

```
import org.apache.spark.sql.types._

// 数据源文件
// JSON 数据集路径可以是单个文件，也可以是存储文件的目录
val file = "/data/spark/resources/people.json"

// 创建 schema。字段名称要与 json 对象的 key 名称保持一致
// 注意下面代码中错误的类型 BooleanType
val fields = Seq(
    StructField("name",BooleanType,nullable = true),
    StructField("age",IntegerType,nullable = true)
)
val schema = StructType(fields)

// 读取数据源，创建 DataFrame，使用自定义的 schema
val df = spark.read.schema(schema).json(file)

// schema
df.printSchema()

// 显示
df.show()
```

执行以上代码，并不会出现任何异常，输出结果如下所示。

```
root
 |-- name: boolean (nullable = true)
 |-- age: integer (nullable = true)

+----+----+
|name| age|
+----+----+
|null|null|
|null|  30|
|null|  19|
+----+----+
```

可以看到，name 列值全部解析为 null，但是这样处理 JSON 格式解析错误并不是一种好的方式，因为它经常会掩盖错误事实，让用户迷惑。所以最好的方式是，如果 JSON 格式解析错误，那么就直接抛出异常（快速失败），而不是全都设为 null 值。例如，在读取 Spark 自带的 people.json 文件到一个 DataFrame 时，指定快速失败的处理方式，实现代码如下。

```
import org.apache.spark.sql.types.

// 数据源文件
// JSON 数据集路径可以是单个文件，也可以是存储文件的目录
val file = "/data/spark/resources/people.json"

// 创建 schema。字段名称要与 json 对象的 key 名称保持一致
val fields = Seq(
   StructField("name",BooleanType,nullable = true), // 错误的类型
   StructField("age",IntegerType,nullable = true)
)

// 读取数据源，创建 DataFrame，使用自定义的 schema
val df = spark.read
    .option("mode","failFast")    // 指定 failFast 模式
    .schema(StructType(fields))
    .json(file)

// schema
df.printSchema()

// 显示
df.show()              // 当执行一个 action 时，Spark 将抛出一个 RuntimeException
```

在上面的代码中，指定 mode 为 failFast 模式，这是用来告诉 Spark，当面对解析错误时，快速失败，而不是生成 null 值。执行以上代码，会抛出异常信息，异常信息如下：

```
root
 |-- name: boolean (nullable = true)
 |-- age: integer (nullable = true)

org.apache.spark.SparkException: Job aborted due to stage failure: Task 0 in
stage 7.0 failed 4 times, most recent failure: Lost task 0.3 in stage 7.0 (TID
10) (192.168.190.133 executor 1): org.apache.spark.SparkException:
Malformed records are detected in record parsing. Parse Mode: FAILFAST. To
process malformed records as null result, try setting the option 'mode' as
'PERMISSIVE'.
     at ......
Caused by: org.apache.spark.sql.catalyst.util.BadRecordException:
java.lang.RuntimeException: Failed to parse a value for data type boolean
(current token: VALUE_STRING).
     at ......
     ... 25 more
Caused by: java.lang.RuntimeException: Failed to parse a value for data type
boolean (current token: VALUE_STRING).
     at ......
     ... 27 more
```

可以看到，在 JSON 格式解析错误时会立即抛出异常信息，从而不会对用户产生误导。

2.1.4 Parquet 文件数据源

Apache Parquet 是一种高效的、压缩的、面向列的开源数据存储格式。它提供了多种存储优化，允许读取单独的列而非整个文件，这不仅节省了存储空间而且提升了读取效率。

Parquet 是 Spark 默认的文件格式，支持非常有效的压缩和编码方案，也可用于 Hadoop 生态系统中的任何项目，可以大大提高这类应用程序的性能。

Apache Spark 提供了对读取和写入 Parquet 文件的支持，这些文件自动保存原始数据的模式。Parquet 是一种非常流行的格式，在 Spark 中有一些额外的选项可以用于读写 Parquet 文件。在写入 Parquet 文件时，出于兼容性考虑，所有列都会自动转换为 nullable。

例如，读取 Spark 自带的 Parquet 文件 users.parquet 文件内容到一个 DataFrame 中，然后打印其 Schema 并输出数据，实现代码如下。

```
// 读取 Parquet 文件
val parquetFile = "/data/spark/resources/users.parquet"

// Parquet 是默认的格式，因此当读取时不需要指定格式
val usersDF = spark.read.load(parquetFile)

// 如果想要更加明确，也可以指定 parquet 函数
// val usersDF = spark.read.parquet(parquetFile)

// 输出模式和内容
usersDF.printSchema()
usersDF.show()
```

执行以上代码，输出结果如下所示。

```
root
 |-- name: string (nullable = true)
 |-- favorite_color: string (nullable = true)
 |-- favorite_numbers: array (nullable = true)
 |    |-- element: integer (containsNull = true)

+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null  |  [3, 9, 15, 20]|
|   Ben|          red   |              []|
+------+--------------+----------------+
```

2.1.5 ORC 文件数据源

ORC （Optimized Row Columnar）是一种流行的大数据文件存储格式。ORC 文件是一种自描述的、类型感知的列存储格式数据文件，它针对大型数据的读写进行了优化，具有高性能、可压缩性强等特点，被许多顶级 Apache 产品支持，比如 Hive、Crunch、Cascading、Spark 等等，是大数据中常用的文件格式。

Apache Spark 提供了对读取和写入 ORC 文件的支持。例如，读取 Spark 自带的 ORC 文件 users.orc 到一个 DataFrame 中，实现代码如下。

```
// 数据源文件
val orcFile = "/data/spark/resources/users.orc"

// 读取 ORC 文件，构造 DataFrame
val orcDF = spark.read.format("orc").load(orcFile)
// val orcDF = spark.read.orc(orcFile)   // 简洁写法

// 输出模式和内容
```

```
orcDF.printSchema()
orcDF.show()
```

执行以上代码，输出结果如下：

```
root
 |-- name: string (nullable = true)
 |-- favorite color: string (nullable = true)
 |-- favorite numbers: array (nullable = true)
 |    |-- element: integer (containsNull = true)

+------+--------------+----------------+
|  name|favorite color|favorite numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+
```

从 Spark 2.3 开始，Spark 支持一个带有新的 ORC 文件格式的向量化 ORC 读取器，配置见表 2-2。

表 2-2 读取 ORC 文件时的配置

属性名	默认值	含义
spark.sql.orc.impl	native	The name of ORC 实现的名称。可以是 native 或 hive。native 意味着本地 ORC 支持，它构建在 Apache ORC 1.4 之上。`hive`意味着 Hive 1.2.1 中的 ORC 库
spark.sql.orc.enableVectorizedReader	true	在 native 实现中启用向量化 orc 解码。如果为 false，则在 native 实现中使用新的非向量化 ORC reader

当 spark.sql.orc.impl 设置为 native 以及 spark.sql.orc.enableVectorizedReader 设置为 true 时，会将向量化的读取器用于本地 ORC 表（例如，使用 USING ORC 子句创建的表）。对于 Hive ORC serde 表，当 spark.sql.hive.convertMetastoreOrc 也被设置为 true 时，也会使用向量化的读取器。

2.1.6 JDBC 数据源

Spark SQL 还包括一个可以使用 JDBC 从其他关系型数据库读取数据的数据源。开发人员可以使用 JDBC 创建来自其他数据库的 DataFrame，只要确保预定数据库的 JDBC 驱动程序是可访问的（需要在 Spark 类路径中包含特定数据库的 JDBC 驱动程序）。

注意： Spark 安装程序默认是没有提供数据库驱动的，所以在使用前需要将相应的数据库驱动上传到$SPARK_HOME 目录下的 jars 目录中。本书示例使用的是 MySQL 数据库，所以使用前需要将相应的 mysql-connector-java-x.x.x.jar 包上传到$SPARK_HOME/jars 目录下。

如果是在 Maven 项目中开发，则需要在 pom.xml 文件中添加 mysql 驱动程序依赖，内容如下：

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.18</version>
</dependency>
```

如果是在 STB 项目中开发，则需要在 build.sbt 文件中添加 mysql 驱动程序依赖，内容如下：

```
libraryDependencies += "mysql" % "mysql-connector-java" % "8.0.18"
```

注意： 作者本地安装的 MySQL 数据库版本为“8.0.18”，请读者在应用时修改为自己的本地数据库版本。

在下面的示例中，通过 JDBC 读取 MySQL 数据库中的一个 peoples 数据表，并创建 DataFrame。为此，请先在 MySQL 中执行如下脚本，创建一外名为 xueai8 的数据和一个名为 peoples 的数据表，并向表中插入一些样本记录，命令如下：

```
mysql> create database xueai8;
mysql> use xueai8;
mysql> create table peoples(id int not null primary key, name varchar(20),
age int);
mysql> insert into peoples values(1,"张三",23),(2,"李四",18),(3,"王老五",35);
mysql> select * from peoples;
```

然后编写代码，从 MySQL 中读取 peoples 表中的数据到 DataFrame 中，实现代码如下。

```
// 定义 mysql 8 的数据库连接 URL，数据库名为 xueai8
val DB_URL= "jdbc:mysql://localhost:3306/xueai8?useSSL=false" +
     "&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true"

// 读取 JDBC 数据源
val peoplesDF = spark.read.format("jdbc")
    .option("driver", "com.mysql.cj.jdbc.Driver") // mysql8 数据库驱动程序类名
    .option("url", DB_URL)                        // 连接 url
    .option("dbtable", "peoples")                 // 要读取的表
    .option("user", "root")                       // 连接账户，需修改为自己的
    .option("password","123456")                  // 连接密码，需修改为自己的
    .option("fetchsize","50")                     // 每轮读取多少行
    .load()

// 输出模式和内容
peoplesDF.printSchema()
peoplesDF.show()
```

注意： 上面的连接配置是 mysql 8 数据库的配置。如果要连接的是 mysql5 数据库，则连接 URL 不需要指定时区信息，并且驱动程序名称为“com.mysql.jdbc.Driver”。另外，数据库连接账号和密码请修改为自己本机数据的账号和密码。

执行以上代码，输出结果如下：

```
root
 |-- id: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)

+---+------+---+
| id|  name|age|
+---+------+---+
|  1|   张三| 23|
|  2|   李四| 18|
|  3| 王老五| 35|
+---+------+---+
```

也可以直接使用 jdbc()快捷方法从关系型数据库中加载数据。例如，改写上面的示例，实现代码如下。

Scala 代码如下：

```
// 定义 mysql 8 的数据库连接 URL，数据库名为 xueai8
val DB_URL= "jdbc:mysql://localhost:3306/xueai8?useSSL=false" +
     "&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true"

// 创建一个 Properties()对象来保存参数
import java.util.Properties
val props = new Properties()

props.put("user", "root")                          // 账号
props.put("password", "admin")                     // 密码

val driverClass = "com.mysql.cj.jdbc.Driver"    // mysql 8 驱动
// val driverClass = "com.mysql.jdbc.Driver"    // mysql 5 驱动
props.put("Driver", driverClass)
// props.setProperty("Driver", driverClass)    // 等价上一句

// 使用快捷方式
val peoplesDF = spark.read.jdbc(DB_URL, "peoples", props)

// 输出模式和内容
peoplesDF.printSchema()
peoplesDF.show()
```

注意： 根据 MySQL 数据库的版本不同（主要是 MySQL 8 有较大的变化），相应的驱动程序名和 JDBC 的连接 URL 也发生变化。

如果读写的是 MySQL 5.x：

(1) 驱动程序为：com.mysql.jdbc.Driver。

(2) 连接 URL 为：jdbc:mariadb://localhost:3306/db?useSSL=false"。

如果读写的是 MySQL 8.x：

(1) 驱动程序为：com.mysql.cj.jdbc.Driver。

(2) 连接 URL 为：

jdbc:mysql://localhost:3306/db?useSSL=false&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true"

还可以使用 query 选项指定用于将数据读入 Spark 的查询语句。指定的查询将被圆括号括起来，并在 FROM 子句中用作子查询。Spark 还将为子查询子句分配一个别名。例如，Spark 将向 JDBC 源发出如下形式的查询：

```
SELECT <columns> FROM (<user_specified_query>) spark_gen_alias
```

使用此选项时有一些限制：

(1) 不允许同时指定 dbtable 和 query 选项。

(2) 不允许同时指定 query 和 partitionColumn 选项。当需要指定 partitionColumn 选项时，可以使用 dbtable 选项指定子查询，分区列可以使用作为 dbtable 的一部分提供的子查询别名进行限定。

例如，通过 PySpark SQL 读取 MySQL 中的数据，使用 query 选项，实现代码如下。

```
// 定义 mysql 8 的数据库连接 URL，数据库名为 xueai8
val DB URL= "jdbc:mysql://localhost:3306/xueai8?useSSL=false" +
     "&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true"

// 定义查询语句
val query = "select name,age from peoples where age>20"

// 读取 JDBC 数据源
val peoplesDF = spark.read.format("jdbc")
    .option("driver", "com.mysql.cj.jdbc.Driver") // mysql8 数据库驱动程序类名
    .option("url", DB URL)                        // 连接 url
    .option("query",query )                       // 要读取的表
    .option("user", "root")                       // 连接账户，需修改为自己的
    .option("password","123456")                  // 连接密码，需修改为自己的
    .option("fetchsize","50")                     // 每轮读取多少行
    .load()

// 输出 schema 和内容
peoplesDF.printSchema()
peoplesDF.show()
```

执行以上代码，输出结果如下：

```
root
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)

+------+---+
|  name|age|
+------+---+
|   张三| 23|
| 王老五| 35|
+------+---+
```

2.2 数据探索

收集、读取或加载到样本数据集后，接下来要考虑的问题是：样本数据集的数量和质量是否满足数据分析和模型构建的要求？有没有出现从未设想过的数据状态？其中有没有明显的规律和趋势？各因素（或属性特征）之间是否有关联性？有什么样的关联性？

通过检验数据集的数据质量、绘制图表、计算某些特征量等手段，对样本数据集的结构和规律进行分析的过程就是数据探索。数据探索有助于选择合适的数据预处理和建模方法，甚至可以完成一些通常由数据挖掘解决的问题。

本节介绍 Spark 中数据探索的方法，并从数据质量分析和数据特征分析两个角度对一个电商数据集进行探索。

2.2.1 Spark 数据简单探索

首先准备一个样本数据集。这里我们使用某著名电商平台双十一美妆销售数据集。该数据集存储在一个 CSV 文件“双十一淘宝美妆数据.csv”中。由于是真实的商业数据，所以做了脱敏处理，数据集中对店名的引用被处理为产品的品牌名以保护店家隐私。部分数据如下：

```
update_time,id,title,price,sale_count,comment_count,店名
2016/11/14,A18164178225,CHANDO/自然堂 雪域精粹纯粹滋润霜 50g 补水保湿 滋润水润面
霜,139,26719,2704,自然堂
```

```
2016/11/14,A18177105952,CHANDO/自然堂凝时鲜颜肌活乳液 120ML 淡化细纹补水滋润专柜正品,194,8122,1492,自然堂
2016/11/14,A18177226992,CHANDO/自然堂活泉保湿修护精华水（滋润型 135ml 补水控油爽肤水,99,12668,589,自然堂
2016/11/14,A18178033846,CHANDO/自然堂 男士劲爽控油洁面膏 100g 深层清洁  男士洗面奶,38,25805,4287,自然堂
...
```

可以看到，该数据带有一个标题行，标题行中各个字段的含义如表 2-3 所示：

表 2-3 双十一淘宝美妆数据字段说明

字段名	含义	数据类型
update_time	统计时间	字符串
id	产品编号	字符串
title	商品名称	字符串
price	商品单价	小数
sale_count	销售数量	整数
comment_count	评论数量	整数
店名	店铺名称	字符串（脱敏处理为品牌名以保护店家隐私）

实际上，该数据集包括 27599 行和 7 个特征变量，每一行对应一个产品的销售情况。将该数据集上传到 HDFS 的指定位置，例如/data/spark/beauty/目录下，命令如下：

```
$ hdfs dfs -put 双十一淘宝美妆数据.csv /data/spark/beauty/
$ hdfs -ls /data/beauty/
```

然后，在 Zeppelin Notebook 中，编写 Spark 代码读取该文件数据源到一个 DataFrame 中，代码实现如下。

```
// 指定要抽取的文件数据源
val filePath = "/data/spark/beauty/双十一淘宝美妆数据.csv"

// 指定 Schema
import org.apache.spark.sql.types._

// 定义字段: update_time,id,title,price,sale_count,comment_count,店名
val fields = Seq(
       StructField("update_time", StringType, true),
       StructField("id", StringType, true),
       StructField("title", StringType, true),
       StructField("price", DecimalType(10,2), true),
       StructField("sale_count", IntegerType, true),
       StructField("comment_count", IntegerType, true),
       StructField("shop", StringType, true)
   )

// 定义 schema
val schema = StructType(fields)

// 读取文件数据源，创建 DataFrame
val df = spark.read
     .option("header",true)
     .option("inferSchema",false)
     .option("sep",",")
     .schema(schema)
     .csv(filePath)
```

Spark 提供了一些方法，如 printSchema()、show()、count()、first()、take()等，用于对数据进行简单地探索分析。在得到一个 DataFrame 后，最常用的两个方法是 printSchema()和 show()，其中 printSchema()方法用来查看表结构模式，show()方法用来查看数据记录，应用代码如下。

```
// 查看前 5 条数据
df.show(5,true)

// 查看数据集的模式(schema)
df.printSchema()
```

在 show()方法中可以指定第 2 个参数，用来指定当列内容较长时，是否截断显示，false 为不截断显示，true 为截断显示。默认为 true。

执行以上代码，输出内容如下：

```
+-----------+------------+-----------------------------+------+---------
-+-------------+------+
|update_time|         id|                        title|
price|sale_count|comment_count|  shop|
+-----------+------------+-----------------------------+------+---------
-+-------------+------+
| 2016/11/14|A18164178225| CHANDO/自然堂 雪域精粹纯粹...|139.00|    26719|
2704|自然堂|
| 2016/11/14|A18177105952|CHANDO/自然堂凝时鲜颜肌活乳...|194.00|     8122|
1492|自然堂|
| 2016/11/14|A18177226992|CHANDO/自然堂活泉保湿修护精...| 99.00|    12668|
589|自然堂|
| 2016/11/14|A18178033846| CHANDO/自然堂 男士劲爽控油...| 38.00|    25805|
4287|自然堂|
| 2016/11/14|A18178045259|CHANDO/自然堂雪域精粹纯粹滋...|139.00|     5196|
618|自然堂|
+-----------+------------+-----------------------------+------+---------
-+-------------+------+
only showing top 5 rows

root
 |-- update_time: string (nullable = true)
 |-- id: string (nullable = true)
 |-- title: string (nullable = true)
 |-- price: decimal(10,2) (nullable = true)
 |-- sale_count: integer (nullable = true)
 |-- comment_count: integer (nullable = true)
 |-- shop: string (nullable = true)
```

如果要查看数据集的大小（数据集中记录的数量），可以使用 count()方法。例如，统计 df 中销售记录的数量，实现代码如下。

```
// 返回数据集中记录的数量
println("数据集中记录的数量: " + df.count)
```

执行以上代码，输出内容如下：

```
数据集中记录的数量: 27598
```

如果只想查看数据集中前一条数据或几条数据，可以使用 first()、head()或 take()方法。实现代码如下。

```
// 返回数据集中第 1 条数据
println("first()方法: ")
println(df.first())

// 等价于 first 方法
println("\nhead()方法: ")
println(df.head())

// 返回数据集中前 3 条数据，以 Array 形式
println("\nhead(3)方法: ")
for(line <- df.head(3)){
    println(line)
}

// 返回数据集中前 3 条数据，以 Array 形式
println("\ntake(3)方法: ")
for(line <- df.take(3)){
    println(line)
}

// 返回数据集中前 3 条数据，以 List 形式
println("\ntakeAsList(3)方法: ")
df.takeAsList(3).forEach(println)
```

执行以上代码，输出内容如下：

```
first()方法:
[2016/11/14,A18164178225,CHANDO/自然堂 雪域精粹纯粹滋润霜 50g 补水保湿 滋润水润面霜,139.00,26719,2704,自然堂]

head()方法:
[2016/11/14,A18164178225,CHANDO/自然堂 雪域精粹纯粹滋润霜 50g 补水保湿 滋润水润面霜,139.00,26719,2704,自然堂]

head(3)方法:
[2016/11/14,A18164178225,CHANDO/自然堂 雪域精粹纯粹滋润霜 50g 补水保湿 滋润水润面霜,139.00,26719,2704,自然堂]
[2016/11/14,A18177105952,CHANDO/自然堂凝时鲜颜肌活乳液 120ML 淡化细纹补水滋润专柜正品,194.00,8122,1492,自然堂]
[2016/11/14,A18177226992,CHANDO/自然堂活泉保湿修护精华水（滋润型 135ml 补水控油爽肤水,99.00,12668,589,自然堂]

take(3)方法:
[2016/11/14,A18164178225,CHANDO/自然堂 雪域精粹纯粹滋润霜 50g 补水保湿 滋润水润面霜,139.00,26719,2704,自然堂]
[2016/11/14,A18177105952,CHANDO/自然堂凝时鲜颜肌活乳液 120ML 淡化细纹补水滋润专柜正品,194.00,8122,1492,自然堂]
[2016/11/14,A18177226992,CHANDO/自然堂活泉保湿修护精华水（滋润型 135ml 补水控油爽肤水,99.00,12668,589,自然堂]

takeAsList(3)方法:
[2016/11/14,A18164178225,CHANDO/自然堂 雪域精粹纯粹滋润霜 50g 补水保湿 滋润水润面霜,139.00,26719,2704,自然堂]
[2016/11/14,A18177105952,CHANDO/自然堂凝时鲜颜肌活乳液 120ML 淡化细纹补水滋润专柜正品,194.00,8122,1492,自然堂]
[2016/11/14,A18177226992,CHANDO/自然堂活泉保湿修护精华水（滋润型 135ml 补水控油爽肤水,99.00,12668,589,自然堂]
```

如果想查看数据集所有列的列名和数据类型，也可以使用 dtypes 方法（注意，这是一个无括号 0 参函数，调用时不能加括号），它以数组形式返回所有列名及其数据类型。实现代码如下。

```
// 返回数据集的数据类型，以 Array 形式
for(e <- df.dtypes){
   println(e)
}
```

执行以上代码，输出内容如下：

```
(update_time,StringType)
(id,StringType)
(title,StringType)
(price,DecimalType(10,2))
(sale_count,IntegerType)
(comment_count,IntegerType)
(shop,StringType)
```

如果只是想获取所有的列名，可以使用 columns 方法（注意，这也是一个无括号 0 参函数，调用时不能加括号），它以数组形式返回所有列名。实现代码如下。

```
// 返回数据集的列名，以 Array 形式
val cols = df.columns
cols
```

执行以上代码，输出内容如下：

```
cols: Array[String] = Array(update_time, id, title, price, sale_count,
comment_count, shop)
```

2. 2. 2 Spark 数据质量分析

数据质量分析是数据准备过程的重要一环，是数据预处理的前提，也是数据分析和数据挖掘结论有效性和准确性的基础。

数据担分析的主要任务是检查原始数据中是否存在脏数据。所谓脏数据，一般是指不符合要求以及不能直接进行相应分析的数据。在数据分析和挖掘工作中，脏数据包括：缺失值、异常值、不一致的值、重复数据以及含有特殊符号（如#、*、?）的数据。

1. 缺失值分析

数据的缺失主要包括记录的缺失和记录中某个或某几个字段值的缺失。数据缺失会造成分析结果不准确，出现偏差，导致模型中蕴含的规律更难把握，会使数据建模过程陷入混乱，产生不可靠的输出。

对缺失值的分析主要从以下两方面进行：

(1) 使用简单的统计分析，可以得到含有缺失值的属性的个性以及每个属性的未缺失数、缺失数与缺失率等。

(2) 对于缺失值的处理，从总体上来说分为三种情况，包括：删除存在缺失值的记录、对可能值进行插补、不处理缺失值。

在 Spark 中，可以聚合函数 count()来统计分组数据中的非空值数量。聚合函数需要和 agg()函数一起使用，实现代码如下。

```
// count(): 这是一个 Action，返回数据集中的行数
println("数据集总记录数: " + df.count())

println("price、sale count 和 comment count 字段的非空值数量: ")
df.agg(count("price").as("price notnull"),
       count("sale count").as("sale count notnull"),
       count("comment_count").as("comment_count_notnull")
  ).show()
```

执行以上代码，输出内容如下：

```
数据集总记录数: 27598
price、sale count 和 comment count 字段的非空值数量:
+-------------+------------------+---------------------+
|price_notnull|sale_count_notnull|comment_count_notnull|
+-------------+------------------+---------------------+
|        27598|             25244|                25244|
+-------------+------------------+---------------------+
```

由以上输出结果可以看出，price 字段没有缺失值，而 sale_count 和 comment_count 两个字段均有缺失值，缺失值数量=27598-25244=2354 条，缺失率约为 8.5%。

2. 异常值分析

异常值是指样本中的个别值，其数值明显偏离其他的观测值。异常值分析是检验数据是否有录入错误，是否含有不合常理的数据。忽视异常值会对分析结果造成不良影响。异常值也称为离群点，因此异常值分析也称为离群点分析。

在进行异常值分析时，可以先对变量做一个描述性统计，进而查看哪些数据是不合理的。最常用的统计量是最大值和最小值，用来判断这个变量的取值是否超出了合理范围。例如，一个用户的身高是 1780cm，则判断该变量的取值存在异常。

如果数据服从正态分布，则可以按 3σ（发音“西格玛”）原则进行检查。在 3σ原则下，异常值被定义为一组测定值中与平均值的偏差超过 3 倍标准差的值。在正态分布的假设下，距离平均值 3σ之外的值出现的概率小于 0.003，属于极个别的小概率事件。

如果数据不服从正态分布，也可以用远离平均值的标准差倍数来描述。

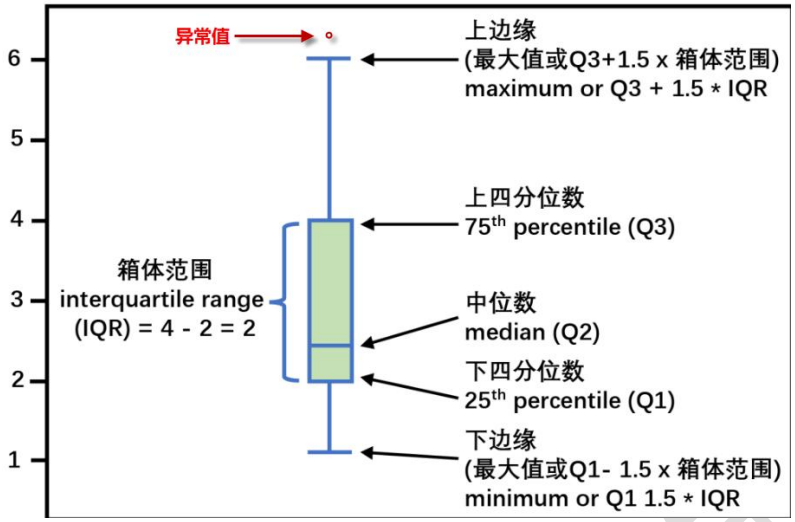
另外，也可以使用箱形图进行异常值分析。箱型图提供了识别异常值的一个标准，即当一个值小于 $Q_L - 1.5IQR$ 或大于 $Q_U + 1.5IQR$ 时，则该值为异常值，其中：

(1) Q_L 代表下四分位数，表示全部观察值中有四分之一的数据取值比它小。

(2) Q_U 代表上四分位数，表示全部观察值中有四分之一的数据取值比它大。

(3) IQR 代表四分位数间距，是上四分位数 Q_U 与下四分位数 Q_L 之差，其间包含了全部观察值的一半。

箱型图及其异常值差别标准如图 2-4 所示。



因为箱形图判断异常值的标准是以四分位数和四分位距为基础，而四分位数具有一定的健壮性：多达 25%的数据可以变得任意远而不会严重扰动四分位数，所以异常值不能对这个标准施加影响。因此箱型图识别异常值的结果也比较客观。

Spark 中提供了多个方法用来计算极值、均值、标准差，如 max()、min()、mean()、stddev()、stddev_samp()等。例如，按 3σ原则来分析 comment_count（用户评论数）字段是否存在异常值，首先计算 comment_count 列的最大值、最小值、均值和标准差。实现代码如下。

```
val outlier_detection = df.agg(
      max("comment count").as("max comm count"),
      min("comment count").as("min comm count"),
      mean("comment count").as("mean comm count"),
      stddev("comment count").as("stddev comm count")
   )
outlier_detection.show()
```

执行以上代码，输出内容如下：

```
+--------------+--------------+----------------+-----------------+
|max_comm_count|min_comm_count| mean_comm_count|stddev_comm_count|
+--------------+--------------+----------------+-----------------+
|        202930|             0|1121.14181587704|5271.059821573427|
+--------------+--------------+----------------+-----------------+
```

然后从这个结果 DataFrame 中分别提取最小值、最大值和标准差，并在 df 中增加一个新列 is_outlier，用 1 表示是异常值，用 0 表示是正常值。实现代码如下。

```
// 先提取第一行，返回一个 org.apache.spark.sql.Row 对象
val first_row = outlier_detection.first()

// 分别提取各个字段的值
val max_comm_count = first_row.getAs[Int](0)
val min_comm_count = first_row.getAs[Int](1)
val mean_comm_count = first_row.getAs[Double](2)
val stddev_comm_count = first_row.getAs[Double](3)

// 计算上边缘值（即 Q3 + 1.5*箱体范围）
val df_with_outlier = df.withColumn("is_outlier",
   when(col("comment_count") - lit(mean_comm_count ) >
lit(stddev_comm_count)*3, 1).otherwise(0))
```

```
println("异常值的数量: " +
df_with_outlier.where(col("is_outlier")===1).count())
df_with_outlier
    .select("id","title","comment_count","is_outlier")
    .where(col("is_outlier")===1)
    .show(10)
```

执行以上代码，输出内容如下：

```
异常值的数量: 213
+-------------+------------------------------+-------------+----------+
|           id|                         title|comment_count|is_outlier|
+-------------+------------------------------+-------------+----------+
| A21625571512|   CHANDO/自然堂水润保湿面部护...|        28585|         1|
| A24612968810|   CHANDO/自然堂雪域精粹纯粹冰...|        17389|         1|
| A26487404578|   CHANDO/自然堂水润保湿洗颜霜...|        28333|         1|
|A526274580793|   CHANDO/自然堂多重隔离防晒乳...|        38066|         1|
|A532236636799|   CHANDO/自然堂喜马拉雅补水嫩...|        27516|         1|
| A21301819561|   CHANDO/自然堂雪域精粹水乳套...|        16952|         1|
| A21625571512|   CHANDO/自然堂水润保湿面部护...|        28640|         1|
| A24612968810|   CHANDO/自然堂雪域精粹纯粹冰...|        17456|         1|
| A26487404578|   CHANDO/自然堂水润保湿洗颜霜...|        28333|         1|
|A526274580793|   CHANDO/自然堂多重隔离防晒乳...|        38242|         1|
+-------------+------------------------------+-------------+----------+
only showing top 10 rows
```

从输出结果可以看出，按 3σ原则分析出的异常值数量有 213 条。不过需要注意的是，因为 comment_count 字段并不一定是服从正态分布的，因此这里按三倍标准差来分析主要是演示的目的。读者朋友可以尝试按 10 倍标准差或 30 倍标准差来分别测试一下结果。

下面换用箱形图方法分析异常值。Spark 提供了一个 percentile_approx()函数来计算指定列的近似分位数，该方法的签名如下：

```
percentile_approx(e: Column, percentage: Column, accuracy: Column): Column
```

这是一个聚合函数，用于计算 e 这个字段的几个近似分位点，其中：

(1) percentage 参数如果是一个数组，则每个值必须在 0.0 到 1.0 之间；如果是单个浮点值，则它必须在 0.0 到 1.0 之间。

(2) accuracy 参数是一个正的数字字面值，它以内存为代价控制近似准确度，准确度值越高，准确度越好，1.0/accuracy 为近似值的相对误差。

```
// 计算最小值、最大值、下四分位数、中位数和上四分位数
val percentile_df = df_with_outlier.agg(
   min(col("comment_count")).as("minimum"),
   max(col("comment_count")).as("maximum"),
   percentile_approx(col("comment_count"), lit(0.25),lit(10000)).as("Q1"),
   percentile_approx(col("comment_count"), lit(0.50),lit(10000)).as("Q2"),
   percentile_approx(col("comment_count"), lit(0.75),lit(10000)).as("Q3")
)

// 显示结果
percentile_df.show()
```

执行以上代码，输出内容如下：

```
+-------+-------+---+---+---+
```

```
|minimum|maximum| Q1| Q2| Q3|
+-------+-------+---+---+---+
|      0| 202930| 21|153|669|
+-------+-------+---+---+---+
```

然后从这个结果 DataFrame 中分别提取最小值、最大值、下四分位数、中位数和上四分位数，并在 df 中增加一个新列 is_outlier，用 1 表示是异常值，用 0 表示是正常值。实现代码如下。

```
// 先提取第一行，返回一个 org.apache.spark.sql.Row 对象
val first_row = percentile_df.first()

// 分别提取各个字段的值
val minimum = first_row.getAs[Int](0)
val maximum = first_row.getAs[Int](1)
val q1 = first_row.getAs[Int](2)
val q2 = first_row.getAs[Int](3)
val q3 = first_row.getAs[Int](4)

// 计算上边缘值（即 Q3 + 1.5*箱体范围）
val df with outlier = df.withColumn("is outlier",
   when(col("comment count") > lit(q3 + 1.5 * (q3-q1)), 1).otherwise(0))

println("异常值的数量: " +
df with outlier.where(col("is outlier")===1).count())
df with outlier
   .select("id","title","comment count","is outlier")
   .where(col("is_outlier")===1)
   .show(10)
```

执行以上代码，输出内容如下：

```
异常值的数量: 3404
+-----------+--------------------------------+-------------+----------+
|         id|                           title|comment_count|is_outlier|
+-----------+--------------------------------+-------------+----------+
|A18164178225|      CHANDO/自然堂 雪域精粹纯粹...|         2704|         1|
|A18178033846|      CHANDO/自然堂 男士劲爽控油...|         4287|         1|
|A18178129035|     自然堂 雪域纯粹滋润洗颜霜 110...|         8426|         1|
|A18919906680|       自然堂 水润保湿柔肤水 135ml...|        10210|         1|
|A18934397934|      CHANDO/自然堂水润保湿霜 50...|         3152|         1|
|A19008350212|     CHANDO/自然堂雪润皙白水乳套...|         2918|         1|
|A19009618209|     CHANDO/自然堂凝时鲜颜套装水...|         1786|         1|
|A19743978078|      CHANDO/自然堂水润保湿面膜 5...|         4394|         1|
|A19829582834|     CHANDO/自然堂雪域精粹密集修...|         1645|         1|
|A20173935321|     CHANDO/自然堂雪域精粹纯粹滋...|         5763|         1|
+-----------+--------------------------------+-------------+----------+
only showing top 10 rows
```

从输出结果可以看出，按箱形图方法分析出的异常值数量有 3404 条。当然了，异常值的定义通常还需要结合具体的业务数据性质来判定。

3. 一致性分析

数据不一致性指的是数据相互矛盾，不能保持一致。不一致数据的产生主要发生在数据集成的过程中，可能是由于数据来自于不同的数据源，或者是对于重复存放的数据未能进行

一致性更新造成的。例如，两个数据集中都存储了客户的地址信息，但是用户地址变更时只更新了一个数据集中的地址数据，那么这两张数据集中就有了不一致的数据。

数据一致性是评估数据质量的一个关键点，通常我们说的一致性是指用来描述同一信息主体在不同的数据集中信息属性是否相同，各实体、属性是否符合一致性约束关系。

数据一致性维度大类下可细分为以下维度小类：

(1) 等值一致性依赖约束：描述检核对象之间数据取值的约束规则。一个检核对象数据取值必须与另一个或多个检核对象在一定规则下相等。等值一致性依赖约束一般指外键关联的场景。例如：理赔表的保单号存在于保单主表中，保单主表中的保单号和理赔表中的保单号是关联关系，两者的数据类型和值要保持一致。

(2) 存在一致性依赖约束：描述检核对象之间数据值存在关系的约束规则。一个检核对象的数据值必须在另一个检核对象满足某一条件时存在。存在一致性依赖约束主要是强调业务的关联性，一个状态发生了则某个值一定会如何。例如：投保状态为已投保，则投保日期不应为空。

(3) 逻辑一致性依赖约束：描述检核对象之间数据值逻辑关系的约束规则。一个检核对象上的数据值必须与另一个检核对象的数据值满足某种逻辑关系（如大于、小于等）。逻辑一致性依赖约束主要强调的是字段间的互相约束关系。例如：投保开始时间小于等于投保结束时间。

例如，我们可以通过相关性分析来进一步了解 comment_count 字段与 sale_count 字段的相关性，以分析两者之间是否保持逻辑一致性，即评论人数多的商品，其销售量也相应较高。

2. 2. 3 Spark 数据特征分析

对数据进行质量分析之后，接下来可进一步对数据进行特征分析，这可以通过计算某些特征量、绘制图表等方式来实现。

1. 数据分布分析

分布分析指的是对数据的分布情况进行描述，从而对事件的发生规律有准确的认识。分布分析能揭示数据的分布特征和分布类型。

对于定量数据，例如销售量（sale_count 字段）、评论数量（comment_count 字段），要想了解其分布形式是对称的还是非对称的、发现某些特大或特小的可疑值，可做出频率分布表、绘制分布直方图等进行直观分析。

对于定性数据，常常根据变量的分类类型来分组，例如店名（shop 字段，美妆数据集脱敏为品牌名称）、定单时间（update_time 字段），可用饼图和条形图直观地显示其分布情况。饼图的每一个扇形部分代表每种类型的占比或频数，根据定性变量的类型数目将饼图分成几个部分，每一部分的大小与每种类型的频数成正比。条形图的高度则代表每种类型的百分比或频数，条形图的完度没有意义。

Spark 提供有 histogram()函数，用于对一个 RDD 统计频率分布。

2. 数据对比分析

对比分析是指把两个相互联系的指标进行比较，从数量上展示和说明研究对象规模的大小、水平的高低、速度的快慢、以及各种关系是否相关。对比分析特别适用于指标间的横纵向比较、时间序列的比较分析。

例如，在美妆销售数据集中，将订单时间（update_time 字段）与销售量（sale_count 字段）进行对比分析，可以找出销售量与时间的关系；将销售量（sale_count 字段）与评论数量（comment_count 字段）进行对比分析，可以揭示商品热度和商品销量之间的关系。

3. 统计量分析

用统计指标对定量数据进行统计描述，通常从集中趋势和离中趋势两个方面进行分析。

平均水平指标是对个体集中趋势的试题，使用最广泛的是均值和中位数；反映变异程序的指标则是对个体离开平均水平的度量，使用较广泛的是标准差（均方差）、四分位间距。

1）集中趋势度量

集中趋势度量通常包括均值、中位数和众数。

(1) 均值

均值是所有数据的平均值，例如美妆商品的平均价格、不同品牌的平均价格和平均销量等。作为一个统计量，均值对极端值很敏感。如果数据中存在极端值或者数据是偏态分布的，那么均值就不能很好地度量数据的集中趋势。截断均值是去掉高、低极端值之后的平均值。

(2) 中位数

中位数是将一组观察值从小到大按顺序排列，位于中间的那个数据。即在全部数据中，小于和大于中位数的数据个数相等。

(3) 众数

众数是指数据集中出现最频繁的值。众数更适用于定性变量，经常用来试题定性变量的中心位置。众数不具有唯一性。众数一般用于离散型变量而非连续型变量。

2）离中趋势度量

离中趋势度量通常包括极差、标准差、变异系数和四分位数间距。

(1) 极差

极差指的是最大值和最小值之间的差值（极差=最大值-最小值）。极差对数据集的极端值非常敏感，并且忽略了位于最大值与最小值之间的数据是如何分布的。

(2) 标准差

标准差是方差的算术平方根，用 σ （希腊语字母西格马，英语发音 sigma）表示。标准差也被称为标准偏差，用它来度量数据偏离均值的程度。标准差的计算公式如下（其中 $\bar{x}$ 代表平均值）：

$$s=\sqrt{\frac{\sum(x_i-\bar{x})^2}{n}}$$

(3) 变异系数

变异系数度量标准差相对于均值的离中趋势。变异系数的计算公式如下（其中 $\bar{x}$ 代表平均值）：

$$CV = \frac{s}{\bar{x}} \times 100\%$$

变异系数主要用来比较两个或多个具有不同单位或不同波动幅度的数据集的离中趋势。

(4) 四分位数间距

四分位数包括上四分位数和下四分位数。将所有数值由小到大排列并分成 4 等份，处于第一个分割点位置的数值是下四分位数（Q1），处于第二个分割点位置（中间位置）的数据是中位数（Q2），处于第三个分割点位置的数值是上四分位数（Q3）。

四分位数间距是指上四分位数 Q3 与下四分位数 Q1 之差，其间包含了全部观察值的一半。其值越大，说明数据的变异程度越大；反之 ，说明数据的变异程度越小。

4. 周期性分析

周期性分析是探索某个变量（或属性、字段）是否随时间的变化而呈现出某种周期变化趋势。时间尺度相对较长的周期性趋势有年度周期性趋势季节性周期性趋势；时间尺度相对较短的有月度周期性趋势、周度周期性趋势、天周期性趋势和小时周期性趋势。

例如，要分析美妆商品的销售量或评论量是否和时间周期有关，可以按天统计销售总量或评论问题，形成按天的时序图，来直观地观察到其销量趋势或评论量趋势。

5. 数据贡献度分析

贡献度分析又称为帕累托分析，它的原理是帕累托法则，又称为 20/80 定律。例如，对一个电商平台来说，其 80%的利润常常来自于 20%最畅销的商品，而其他 80%的商品仅仅贡献了其余的 20%利润。

例如，美妆商品销售数据中，20%的品牌贡献了 80%的销量；同一品牌的美妆，20%的商品贡献了该品牌 80%的销量。

6. 数据相关性分析

相关分析指的是分析连续变量（属性、字段）之间线性相关程度的强弱，并用适当的统计指标表示出来的过程。

如果要判断两个变量是否具有线性相关关系，最直观的方法是绘制二维散点图。

如果要同时考察多个变量间的相关关系时，可利用散点图矩阵来同时绘制各变量间的散点图，从而快速发现多个变量间的主要相关性，这在进行多元线性回归时显得尤为重要。

为了更加准确地描述变量之间的相关程度，可以通过相关系数的计算来进行相关性分析。在二元变量的相关分析过程中，比较常用的有 Pearson 相关系数、Spearman 秩相关系数和判定系数。

1）Pearson 相关系数

Pearson 相关系数，又名皮尔逊相关系数，一般用于分析两个连续性变量之间的关系，其计算公式如下：

$$r=\frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2\sum_{i=1}^{n}(y_i-\bar{y})^2}}$$

相关系数 r 的取值范围为：$-1 \le r \le 1$，并且：

(1) 当 $r > 0$ 时为正相关关系，当 $r < 0$ 时为负相关关系。

(2) $|r|=0$ 表示不存在线性关系。

(3) $|r|=1$ 表示完全线性相关。

当 r 的取值在 $0 < |r| < 1$ 范围内时，表示存在不同程度的线性相关：

(1) $|r| \le 0.3$，为极弱线性相关或不存在线性相关。

(2) $0.3 < |r| \le 0.5$，为低度线性相关。

(3) $0.5 < |r| \le 0.8$，为显著线性相关。

(4) $|r| > 0.8$，为高度线性相关。

2）Spearman 秩相关系数

Pearson 相关系数要求连续变量的取值服从正态分布。对于不服从正态分布的变量、分类或等级变量之间的关联性，可采用 Spearman 秩相关系数（又名等级相关系统、斯皮尔曼相关系数）来描述。

只要两个变量具有严格单调的函数关系，那么它们就是完全 Spearman 相关的，这与 Pearson 相关系数不同，后者只有在变量具有线性关系时才是完全相关的。

研究表明，在正态分布情况下，Spearman 秩相关系数与 Pearson 相关系数在效率上是等价的，而对于连续测量数据，更适合用 Pearson 相关系数来进行分析。

3）判定系数

判定系统是相关系数的平方，用 r^2 表示，用来衡量回归方程对因变量 y 的解释程度。判定系统的取值范围为 $0 \le r^2 \le 1$。r^2 越接近于 1，表明 x 与 y 这间的相关性越强；r^2 越接近于 0，越表明这两个变量之间几乎没有直线相关关系。

4）使用 Spark corr()函数

Spark 提供了一个 corr()函数用来计算两列的 Pearson 相关系数，其定义如下：

```
def corr(columnName1: String, columnName2: String): Column
def corr(column1: Column, column2: Column): Column
```

这是一个聚合函数，要在 agg()函数中使用，其示例应用的实现代码如下。

```
// 创建一个 DataFrame
val df = sc.parallelize(0 until 10).toDF("id")
    .withColumn("rand1", rand(seed=1))
    .withColumn("rand2", rand(seed=50))

// 计算 rand1 和 rand2 这两列的皮尔逊相关系数
```

```
df.agg(corr("rand1","rand2").as("pearson_corr")).show()
```

执行以上代码，输出内容如下：

```
+------------------+
|      pearson corr|
+------------------+
|0.6720576967812109|
+------------------+
```

从计算的 Pearson 相关系数值可以得知，rand1 和 rand2 这两列是显著相关的。读者朋友可以尝试修改 rand()函数的 seed 种子值，观察相关性的变化。

注意： rand()函数的功能是，生成一个随机列，其样本均匀分布在[0.0,1.0)中，且独立同分布（independent and identically distributed，i.i.d.）。如果想要从标准正态分布中生成具有独立和同分布（i.i.d.）样本的列，使用 randn()函数。

计算两组数据之间的相关性是统计中常见的操作。在 spark.ml 包中提供了计算多组数据之间两两相关关系的灵活性。目前支持的相关方法是 Pearson 和 Spearman 相关性。

Correlation 使用指定的.corr()方法计算向量输入数据集的关联矩阵。输出将是一个 DataFrame，其中包含向量列的相关矩阵。

```
import org.apache.spark.ml.linalg.{Matrix, Vectors}
import org.apache.spark.ml.stat.Correlation
import org.apache.spark.sql.Row

// 构造数据
val data = Seq(
  Vectors.sparse(4, Seq((0, 1.0), (3, -2.0))),  // [1,0,0,-2.0]
  Vectors.dense(4.0, 5.0, 0.0, 3.0),            // [4,5,0, 3.0]
  Vectors.dense(6.0, 7.0, 0.0, 8.0),            // [6,7,0, 8.0]
  Vectors.sparse(4, Seq((0, 9.0), (3, 1.0)))    // [9,0,0, 1.0]
)

// 构造 DataFrame
val df = data.map(Tuple1.apply).toDF("features")

// df.show

val Row(coeff1: Matrix) = Correlation.corr(df, "features").head
println(s"\n 皮尔森相关矩阵:\n $coeff1")

val Row(coeff2: Matrix) = Correlation.corr(df, "features", "spearman").head
println(s"斯皮尔曼相关矩阵:\n $coeff2")
```

执行以上代码，输出结果如下所示：

```
皮尔森相关矩阵:
1.0                   0.055641488407465814  NaN  0.4004714203168137
0.055641488407465814  1.0                   NaN  0.9135958615342522
NaN                   NaN                   1.0  NaN
0.4004714203168137    0.9135958615342522    NaN  1.0

斯皮尔曼相关矩阵:
1.0                  0.10540925533894532  NaN  0.40000000000000174
0.10540925533894532  1.0                  NaN  0.9486832980505141
NaN                  NaN                  1.0  NaN
```

```
0.40000000000000174   0.9486832980505141    NaN  1.0
```

2. 2. 4 Spark 统计 API

除上前面几节已经了解到的一些 Spark 用于数据探索的函数外，Spark 的 Spark SQL 模块还为 Dataset/DataFrame 提供了一些专门用于信息统计的函数，如 describe()、summary()，并提供了一个 DataFrameStatFunctions 统计类，用于封装 DataFrame 的统计信息。下面分别进行介绍。

1）describe()函数

Spark 为 Dataset/DataFrame 提供了一个 describe()函数，用来计算数字列和字符串列的基本统计信息，包括 count、mean（均值）、stddev（标准差）、min 和 max。如果没有给定列，则此函数计算所有数值列或字符串列的统计信息。该函数返回的也是一个 DataFrame，其方法签名如下：

```
def describe(cols: String*): DataFrame
```

这个方法是一个 Action 类型操作，经常用于对数据集执行探索性数据分析。例如，对上面的美妆数据集 df 执行描述性统计分析，实现代码如下。

```
val descDF = movies.describe()

descDF.show()
```

执行以上代码，输出内容如图 2-1 所示。

```
// 执行描述性统计
val descDF = df.describe()

descDF.show()
```

```
+-------+-----------+------------+-----------------------------------+-----------------+-----------------+-----------------+------+
|summary|update_time|          id|                              title|            price|       sale_count|    comment_count|  shop|
+-------+-----------+------------+-----------------------------------+-----------------+-----------------+-----------------+------+
|  count|      27598|       27598|                              27598|            27598|            25244|            25244| 27598|
|   mean|       null|        null|                               null|       362.830600|12301.77416415782| 1121.14181587704|  null|
| stddev|       null|        null|                               null|614.1733203621806|52336.92628409459|5271.059821573427|  null|
|    min| 2016/11/10|A10027317366|  2件*德国妮维雅唇膏男女润唇膏唇膜...|             1.00|                0|                0|  SKII|
|    max|  2016/11/9| A9768255247|（限量）蜜丝佛陀彩虹遮瑕笔膏熊猫眼...|         11100.00|          1923160|           202930|雪花秀|
+-------+-----------+------------+-----------------------------------+-----------------+-----------------+-----------------+------+
```

图 2-1　添加系统变量

从统计结果可以得出以下信息：

(1) sale_count 列和 comment_count 列有缺失值，因为它们的统计值小于 27598。

(2) 商品销售价格，最低 1 元，最高 11100 元，均价 362.83 元。

(3) 商品销售量，最低 0 个/套，最高 1923160 个/套。

(4) 商品评论数量，最少的 0 条，最高的 202930 条。

(5) 因为 update_time 列是字符串类型，且不是标准日期格式（即 yyyy-MM-dd），所以统计的最小日期和最大日期是不正确的。

(6) 对于 id 列、title 列和 shop 列，除了 count 统计值，其它统计值无意义。

如果只是想计算 price、sale_count 和 comment_count 这三列的基本统计信息，实现代码如下。

```
val descDF = df.describe("price","sale_count","comment_count")
```

```
descDF.show()
```

执行以上代码，输出内容如下：

```
+-------+-----------------+-----------------+-----------------+
|summary|            price|       sale count|   comment count|
+-------+-----------------+-----------------+-----------------+
|  count|            27598|            25244|            25244|
|   mean|       362.830600|12301.77416415782|1121.14181587704|
| stddev|614.1733203621806|52336.92628409459|5271.059821573427|
|    min|             1.00|                0|                0|
|    max|         11100.00|          1923160|           202930|
+-------+-----------------+-----------------+-----------------+
```

注意： 这个函数用于探索性数据分析，它不能保证结果数据集模式的向后兼容性。如果希望以编程方式计算汇总统计信息，请使用 agg()函数。

2）summary()函数

Spark 还提供了一个与 describe()函数类似的 summary()函数，来扩展统计信息，并控制计算哪些统计信息。如果没有给出统计信息，这个函数将计算 count、mean、stddev、min、近似四分位数（25%、50%和 75%的百分位数）和 max 值。

例如，对 df 调用 summary()函数，实现代码如下。

```
val summaryDF = df.summary()
summaryDF.show()
```

执行以上代码，输出内容如图 2-2 所示。

```
val summaryDF = df.summary()
summaryDF.show()
```

```
+-------+-----------+-----------+--------------------------------+-----------------+-----------------+-----------------+-------+
|summary|update_time|         id|                           title|            price|       sale_count|    comment_count|   shop|
+-------+-----------+-----------+--------------------------------+-----------------+-----------------+-----------------+-------+
|  count|      27598|      27598|                           27598|            27598|            25244|            25244|  27598|
|   mean|       null|       null|                            null|       362.830600|12301.77416415782|1121.14181587704|   null|
| stddev|       null|       null|                            null|614.1733203621806|52336.92628409459|5271.059821573427|   null|
|    min| 2016/11/10|A10027317366|  2件*德国妮维雅唇膏男女润唇膏唇膜...|             1.00|                0|                0|   SKII|
|    25%|       null|       null|                            null|             99.0|            279|             21|   null|
|    50%|       null|       null|                            null|            205.0|             1444|            153|   null|
|    75%|       null|       null|                            null|            390.0|             6354|            669|   null|
|    max|  2016/11/9| A9768255247|（限量）蜜丝佛陀彩虹遮瑕笔膏熊猫眼...|          11100.00|          1923160|           202930|雪花秀|
+-------+-----------+-----------+--------------------------------+-----------------+-----------------+-----------------+-------+
```

图 2-2　添加系统变量

第二四分位数（50%），又称“中位数”，等于该样本中所有数值由小到大排列后第 50%的数字。可以看出，商品价格（price 字段）的中位数是 205.0 元，销售量（sale_count 字段）的中位数是 1444，评论数量（comment_count 字段）的中位数是 153，普遍是偏向于价格较低的价格区间。

也可以指定想要的统计信息。例如，只统计 count、min、50%、max 这几个摘要信息，实现代码如下。

```
val summaryDF = df.summary("count", "min", "50%", "max")

summaryDF.show()
```

执行以上代码，输出内容如图 2-4 所示。

```
// 也可以指定想要的统计信息
val summaryDF = df.summary("count", "min", "50%", "max")

summaryDF.show()
```

```
+-------+-----------+-----------+--------------------------------+--------+----------+-------------+------+
|summary|update_time|         id|                           title|   price|sale_count|comment_count|  shop|
+-------+-----------+-----------+--------------------------------+--------+----------+-------------+------+
|  count|      27598|      27598|                           27598|   27598|     25244|        25244| 27598|
|    min| 2016/11/10|A10027317366|  2件*德国妮维雅唇膏男女润唇膏唇膜...|    1.00|         0|            0|  SKII|
|    50%|       null|       null|                            null|   205.0|      1444|          153|  null|
|    max|  2016/11/9| A9768255247|（限量）蜜丝佛陀彩虹遮瑕笔膏熊猫眼...|11100.00|   1923160|       202930|雪花秀|
+-------+-----------+-----------+--------------------------------+--------+----------+-------------+------+
```

图 2-1　添加系统变量

如果想要对指定的列做一个摘要统计，则首先选择这些列，然后再执行 summary()方法。实现代码如下。

```
// 对指定的列做一个摘要统计
val summaryDF = df.select("price","sale_count","comment_count").summary()

summaryDF.show()
```

执行以上代码，输出内容如下：

```
+-------+-----------------+-----------------+-----------------+
|summary|            price|       sale count|    comment count|
+-------+-----------------+-----------------+-----------------+
|  count|            27598|            25244|            25244|
|   mean|       362.830600|12301.77416415782|1121.14181587704|
| stddev|614.1733203621806|52336.92628409459|5271.059821573427|
|    min|             1.00|                0|                0|
|    25%|             99.0|              279|               21|
|    50%|            205.0|             1444|              153|
|    75%|            390.0|             6354|              669|
|    max|          11100.00|          1923160|           202930|
+-------+-----------------+-----------------+-----------------+
```

3）DataFrameStatFunctions 函数类

DataFrameStatFunctions 是 DataFrame 的统计函数，它位于 org.apache.spark.sql 包中。该类提供了许多统计函数用来统计 DataFrame 的信息。在一个 DataFrame 上调用 stat 无参方法，可以获得该 DataFrame 的 DataFrameStatFunctions 对象实例。实现代码如下：

```
df.stat    // 返回一个 DataFrameStatFunctions 用于工作统计函数的支持
```

下面来了解一些常用的 DataFrameStatFunctions 统计方法。

(1) corr()函数

DataFrameStatFunctions 提供了 corr()函数用来计算 DataFrame 中两列的 Pearson 相关系数。该 corr()函数有两个重载版本，定义如下：

```
def corr(col1: String, col2: String): Double
def corr(col1: String, col2: String, method: String): Double
```

在这两个方法中，参数 col1 是列名，col2 是要计算相关性的列的名称。第二个重载版本多了个 method 参数，目前只支持 Pearson 相关系数。对于 Spearman 秩相关系数，考虑使用在 MLlib 的 Statistics 中找到的 RDD 方法。这两个方法都返回 Double 类型的 Pearson 相关系数。其用法实现代码如下。

```
val df = sc.parallelize(0 until 10).toDF("id")
```

```
    .withColumn("rand1", rand(seed=1))
    .withColumn("rand2", rand(seed=50))

df.show()

println("皮尔逊相关系数" + df.stat.corr("rand1", "rand2", "pearson"))
```

执行以上代码，输出内容如下：

```
+---+-------------------+--------------------+
| id|              rand1|               rand2|
+---+-------------------+--------------------+
|  0| 0.6363787615254752|  0.6248339318714432|
|  1| 0.5993846534021868|  0.3095650587587385|
|  2|  0.134842710012538|0.014611374754349371|
|  3|0.07684163905460906| 0.12000822622067098|
|  4| 0.8539211111755448|  0.6904828655239911|
|  5| 0.5311207224659675| 0.05778277112153141|
|  6| 0.2861372051669987| 0.42446338738757383|
|  7|0.49443063728956627|  0.7290635083080451|
|  8|0.45537077449713226|  0.2114766264584158|
|  9| 0.8792399632068049|  0.5842625408772919|
+---+-------------------+--------------------+

皮尔逊相关系数 0.6720576967812107
```

从计算的 Pearson 相关系数值可以得知，rand1 和 rand2 这两列是显著相关的。读者朋友可以尝试修改 rand()函数的 seed 种子值，观察相关性的变化。

(2) approxQuantile()函数

DataFrameStatFunctions 提供了 approxQuantile()函数用来计算 DataFrame 的数值列的近似分位数。该 approxQuantile()函数有两个重载版本，定义如下：

```
// 同时对多个数值列计算近似分位数
def approxQuantile(cols: Array[String], probabilities: Array[Double],
relativeError: Double): Array[Array[Double]]

// 对单个数值列计算近似分位数
def approxQuantile(col: String, probabilities: Array[Double], relativeError:
Double): Array[Double]
```

上述方法中的 probabilities 参数是分位数概率的列表，其中每个数字必须属于[0,1]。例如 0 是最小值，0.5 是中值，1 是最大值。参数 relativeError 指定要实现的相对目标精度（大于或等于 0）。如果设置为 0，将计算准确的分位数，这可能非常昂贵。请注意，大于 1 的值可以接受，但给出的结果与 1 相同。

需要注意的是，数值列中的 null 和 NaN 值将在计算之前被忽略。对于只包含 null 值或 NaN 值的列，将返回一个空数组。

例如，要计算美妆销售数据集中的评论数量（comment_count 列）的四分位数，可以通过 approxQuantile()函数实现，实现代码如下。

```
val quantiles = df.stat
     .approxQuantile("comment_count", Array(0.25, 0.5, 0.75), 0.0001)

println(quantiles.toList)
println("下四分位值: " + quantiles(0))
```

```
println("中位数: " + quantiles(1))
println("上四分位值: " + quantiles(2))
```

执行以上代码，输出内容如下：

```
List(21.0, 153.0, 669.0)
下四分位值: 21.0
中位数: 153.0
上四分位值: 669.0
```

(3) cov()函数

DataFrameStatFunctions 提供了 cov()函数，用于计算一个 DataFrame 的两个数值列的样本协方差。该函数的定义如下：

```
def cov(col1: String, col2: String): Double
```

协方差表示的是两个变量的总体的误差，这与只表示一个变量误差的方差不同。如果两个变量的变化趋势一致，也就是说如果其中一个大于自身的期望值，另外一个也大于自身的期望值，那么两个变量之间的协方差就是正值。如果两个变量的变化趋势相反，即其中一个大于自身的期望值，另外一个却小于自身的期望值，那么两个变量之间的协方差就是负值。如果两个变量是统计独立的，那么二者之间的协方差就是 0。协方差为 0 的两个随机变量称为是不相关的。

例如，要计算美妆销售数据集中的销售量（sale_count 列）和评论数量（comment_count 列）的协方差，可以通过 cov()函数实现，实现代码如下。

```
val covariance = df.stat.cov("sale_count","comment_count")
println("销售量和评论数量的协方差为: " + covariance)
```

执行以上代码，输出内容如下：

```
销售量和评论数量的协方差为: 2.1758380631914788E8
```

由计算结果可知，美妆销售数据集中的销售量（sale_count 列）和评论数量（comment_count 列）是正相关的。

(4) freqItems()函数

DataFrameStatFunctions 提供了 freqItems()函数，用于查找列的频繁项，可能有误报。该函数有多个重载的版本，定义如下：

```
def freqItems(cols: Seq[String], support: Double): DataFrame
def freqItems(cols: Array[String], support: Double): DataFrame
```

这个函数用于探索性的数据分析，其中参数 cols 代表要搜索频繁项的列的名称，参数 support 代表一个 item 项被认为是频繁的最低频率，它应该大于 1e-4，默认是 0.01。该函数返回一个本地 DataFrame，每个列都包含频繁项的数组。它的示例代码实现如下。

```
// Scala tableulate 方法方法可以用来创建和填充 List
val rows = Seq.tabulate(100) { i =>
  if (i % 2 == 0) (1, -1.0) else (i, i * -1.0)
}

// 构造一个 DataFrame
val df = spark.createDataFrame(rows).toDF("a", "b")
df.show(10)
/*
```

```
+---+----+
|  a|   b|
+---+----+
|  1|-1.0|
|  1|-1.0|
|  1|-1.0|
|  3|-3.0|
|  1|-1.0|
|  5|-5.0|
|  1|-1.0|
|  7|-7.0|
|  1|-1.0|
|  9|-9.0|
+---+----+
*/

// 找出“a”和“b”列中频率大于 0.4(观察 40%的时间)的项
val freqSingles = df.stat.freqItems(Array("a", "b"), 0.4)
freqSingles.show()
/*
+-----------+-------------+
|a_freqItems|  b_freqItems|
+-----------+-------------+
|    [1, 99]||[-1.0, -99.0]|
+-----------+-------------+
*/

// 找出“a”和“b”列中频率大于 0.1 的 items 对
val pairDf = df.select(struct("a", "b").as("a-b"))
pairDf.show(10)
/*
+---------+
|      a-b|
+---------+
|{1, -1.0}|
|{1, -1.0}|
|{1, -1.0}|
|{3, -3.0}|
|{1, -1.0}|
|{5, -5.0}|
|{1, -1.0}|
|{7, -7.0}|
|{1, -1.0}|
|{9, -9.0}|
+---------+
*/

val freqPairs = pairDf.stat.freqItems(Array("a-b"), 0.1)
freqPairs.select(explode($"a-b_freqItems").as("freq_ab")).show()
/*
+-----------+
|    freq_ab|
+-----------+
|{93, -93.0}|
|{95, -95.0}|
|{45, -45.0}|
|{99, -99.0}|
```

```
|{49, -49.0}|
|{47, -47.0}|
|{43, -43.0}|
|{97, -97.0}|
|{91, -91.0}|
|  {1, -1.0}|
+-----------+
*/
```

(5) crosstab()函数

DataFrameStatFunctions 提供了 crosstab()函数，用于计算给定列的成对频率表，也称为列联表。每个列的唯一值的数量应该小于 1e4。最多返回 1e6 对非零频率。每一行的第一列将是 col1 的唯一值，而列名将是 col2 的唯一值。第一个列的名称将是 col1_col2。计数将作为 Long 值返回。没有出现的配对计数将为 0。Null 元素将被替换为“null”，如果元素存在反勾号，反勾号将被从元素中删除。该函数定义如下：

```
def crosstab(col1: String, col2: String): DataFrame
```

其中参数 col1 是第一列的名称，它的不同项（唯一值项）将构成每行的第一项。参数 col2 是第二列的名称，它的不同项（唯一值项）将构成 DataFrame 的列名。该函数将返回一个包含列联表的 DataFrame。使用 crosstab()函数的示例代码如下。

```
// 构造一个 DataFrame
val df = spark
    .createDataFrame(Seq((1, 1), (1, 2), (2, 1), (2, 1), (2, 3), (3, 2), (3,
3)))
    .toDF("key", "value")
df.show()

// 计算给定列的列联表
val ct = df.stat.crosstab("key", "value")
ct.show()
```

执行以上代码，输出内容如下：

```
+---+-----+
|key|value|
+---+-----+
|  1|    1|
|  1|    2|
|  2|    1|
|  2|    1|
|  2|    3|
|  3|    2|
|  3|    3|
+---+-----+

+---------+---+---+---+
|key value|  1|  2|  3|
+---------+---+---+---+
|        2|  2|  0|  1|
|        1|  1|  1|  0|
|        3|  0|  1|  1|
+---------+---+---+---+
```

4）Summarizer 向量列摘要统计信息

还可以通过 org.apache.spark.ml.stat 包中的 Summarizer 类为 DataFrame 提供向量列摘要统计信息。可用的度量包括按列排列的最大值、最小值、平均值、方差和非零个数，以及总计数。请看下面的示例。

```
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.ml.stat.Summarizer

// 构造 DataFrame，带有一个特征向量列
val data = Seq(
  (Vectors.dense(2.0, 3.0, 5.0), 1.0),
  (Vectors.dense(4.0, 6.0, 7.0), 2.0)
)

val df = data.toDF("features", "weight")

val (meanVal, varianceVal) = df.select(metrics("mean", "variance")
  .summary($"features", $"weight").as("summary"))
  .select("summary.mean", "summary.variance")
  .as[(Vector, Vector)].first()

println(s"with weight: mean = ${meanVal}, variance = ${varianceVal}")

val (meanVal2, varianceVal2) = df
    .select(mean($"features"), variance($"features"))
    .as[(Vector, Vector)].first()

println(s"without weight: mean = ${meanVal2}, sum = ${varianceVal2}")
```

执行以上代码，输出结果如下所示：

```
with weight: mean = [3.333333333333333,5.0,6.333333333333333], variance =
[2.0,4.5,2.0]
without weight: mean = [3.0,4.5,6.0], variance = [2.0,4.5,2.0]
```

5）Spark SQL 统计指标

也可以使用 Spark SQL 进行指标统计，Spark SQL 已经自带了相关的内置函数：mean 均值，variance 方差，stddev 标准差，corr(Pearson 相关系数)，skewness 偏度，kurtosis 峰度。

```
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.ml.stat.Summarizer

// 构造 DataFrame，带有一个特征向量列
val data = Seq(
  (Vectors.dense(2.0, 3.0, 5.0), 1.0),
  (Vectors.dense(4.0, 6.0, 7.0), 2.0)
)

val df = data.toDF("features", "weight")
df.show()

// 注册为临时表
df.createOrReplaceTempView("tmp view")

// 进一步改写**********************************************
```

```
val df4=spark.sql("SELECT mean(age), variance(age), stddev(age),
corr(age,yearsmarried), skewness(age), kurtosis(age) FROM Affairs")
```

执行以上代码，输出结果如下所示：

```

```

2. 2. 5 Spark 数据探索示例

本小节将继续对美妆销售数据集进行探索，进一步分析其数据分布特征和属性相关性，以深入洞察该销售数据中蕴含的规律和特征。为了读者阅读更清晰，首先列出数据加载代码如下。

```
// 指定要抽取的文件数据源
val filePath = "/data/spark/beauty/双十一淘宝美妆数据.csv"

// 指定 Schema
import org.apache.spark.sql.types._

// 定义字段: update_time,id,title,price,sale_count,comment_count,店名
val fields = Seq(
       StructField("update_time", StringType, true),
       StructField("id", StringType, true),
       StructField("title", StringType, true),
       StructField("price", DecimalType(10,2), true),
       StructField("sale_count", IntegerType, true),
       StructField("comment_count", IntegerType, true),
       StructField("shop", StringType, true)
   )

// 定义 schema
val schema = StructType(fields)

// 读取文件数据源，创建 DataFrame
val df = spark.read
     .option("header",true)
     .option("inferSchema",false)
     .option("sep",",")
     .schema(schema)
     .csv(filePath)
```

接下来对美妆销售数据集进行探索性分析。

(1) 查看美妆商品销售价格分布情况

首先分析美妆商品销售集中商品销售的最低价格、最高价格、四分位价格等。这里使用summary()函数，实现代码如下。

```
df.select("price").summary().show
```

执行以上代码，输出内容如下：

```
+-------+-----------------+
|summary|            price|
+-------+-----------------+
|  count|            27598|
|   mean|       362.830600|
| stddev|614.1733203621806|
```

```
|    min|             1.00|
|    25%|             99.0|
|    50%|            205.0|
|    75%|            390.0|
|    max|         11100.00|
+-------+-----------------+
```

从统计结果可以看出，美妆商品销售价格区域跨度比较大，最低价格可以低至 1.00 元，最高可高至 11100.00 元，平均价格约 363.00 元左右。大部分美妆商品的价格位于 99.00~390.00 元之间。

用同样的方法，可以分析销售量（sale_count 字段）和用户评论数量（comment_count 字段）的统计信息，请读者自行尝试。

(2) 统计美妆商品销售时间跨度

数据集中的 update_time 字段为销售统计时间，按该字段分组，然后统计每天的销售总量。实现代码如下。

```
df.groupBy("update_time").sum("sale_count").show()
```

执行以上代码，输出内容如下：

```
+-----------+---------------+
|update_time|sum(sale_count)|
+-----------+---------------+
| 2016/11/12|       28821197|
|  2016/11/7|       32011063|
|  2016/11/8|       32666229|
|  2016/11/6|       32617979|
| 2016/11/14|       29958174|
|  2016/11/9|       33323546|
| 2016/11/11|       27820235|
|  2016/11/5|       32246405|
| 2016/11/10|       31779471|
| 2016/11/13|       29301688|
+-----------+---------------+
```

可以看到，该数据集包含 10 天的美妆销售数据，从 2016 年 11 月 5 日到 2016 年 11 月 14 日。请注意，因为 update_time 字段是字符串类型，我们目前尚未对其进行预处理（转换为标准日期格式 yyyy-MM-dd），因此无法按时序数据进行处理。在下一节将对该数据集进行预处理，包括日期标准格式转换。

如果想同时统计多个聚合指标，则可以通过 agg()函数实现。例如，统计每天的总销售量和总评论数量，以及每天的平均销售量和平均评论数量，实现代码如下。

```
df.groupBy("update time")
  .agg(
     sum("sale count").as("sum sale count"),
     mean("sale count").as("mean sale count"),
     sum("comment count").as("sum comment count"),
     mean("comment_count").as("mean_comment_count")
  ).show()
```

执行以上代码，输出内容如下：

```
+-----------+--------+-----------------+-----------+------------------+
|update_time|sum_sale|         mean_sale|sum_comment|      mean_comment|
```

```
+----------+--------+-----------------+----------+-----------------+
| 2016/11/12|28821197|15034.531559728743|    2586529|1349.2587376108502|
|  2016/11/7|32011063|11149.795541623127|    2946896|1026.4353883664228|
|  2016/11/8|32666229| 11429.75122463261|    2978503| 1042.163400979706|
|  2016/11/6|32617979|11251.458778889271|    3016238|1040.4408416695412|
| 2016/11/14|29958174|15578.873634945397|    2693889|1400.8783151326054|
|  2016/11/9|33323546|11590.798608695652|    2999176| 1043.191652173913|
| 2016/11/11|27820235|12838.133364097832|    2503262| 1155.173973234887|
|  2016/11/5|32246405|11115.617028610824|    2988385|1030.1223715960014|
| 2016/11/10|31779471|10988.752074688797|    2916942|1008.6244813278008|
| 2016/11/13|29301688| 15096.18134981968|    2672284|1376.7563111798042|
+----------+--------+-----------------+----------+-----------------+
```

(3) 找出店铺/品牌频繁项

因为美妆销售数据集经过脱敏处理，将店铺名全部替换品牌名（shop 字段）。要找出最频繁的店铺/品牌，实现代码如下。

```
df.stat.freqItems(Seq("shop"), 0.8).show()
```

执行以上代码，输出内容如下：

```
+--------------+
|shop_freqItems|
+--------------+
|       [佰草集]|
+--------------+
```

从结果可以看出，出现最多的品牌是“佰草集”，这是国产品牌，上海家化旗下的产品，非常棒！

为了更加直观地进行分析，接下来将 df 注册为一个临时视图，然后使用 Spark SQL 语句进行探索性分析，并利用 Zeppelin Notebook 对 SQL 语句的可视化支持，将分析结果以图表的形式展示出来。

将 df 注册到名为 beauty_tb 的临时视图，代码实现如下。

```
df.createOrReplaceTempView("beauty_tb")
```

(4) 店铺/品牌数量 Top 10 统计

接下来统计总数量位居前 10 位的店铺/品牌，实现代码如下。

```
spark.sql("""
   select shop,count(*) as cnt
   from beauty_tb
   group by shop
   order by cnt desc
   limit 10
 """).show
```

执行以上代码，输出内容如下：

```
+--------+----+
|    shop| cnt|
+--------+----+
|  悦诗风吟|3021|
|   佰草集|2265|
|   欧莱雅|1974|
|  雅诗兰黛|1810|
|    倩碧|1704|
```

```
|    美加净|1678|
|    欧珀莱|1359|
|    妮维雅|1329|
|  相宜本草|1313|
|      兰蔻|1285|
+--------+----+
```

Zeppelin Notebook 通过 sql 解释器支持对 SQL 查询结果的可视化展示。例如，我们想要以饼状图显示 Top 10 个美妆品牌，在 notebook 的代码单元格首行调用 sql 解释器（%sql），后跟要执行的 SQL 语句即可。代码如下：

```
%sql
select shop,count(*) as cnt
from beauty_tb
group by shop
order by cnt desc
limit 10
```

执行以上语句，点击饼状图显示图片，查询结果将以饼状图呈现，结果如图所示。

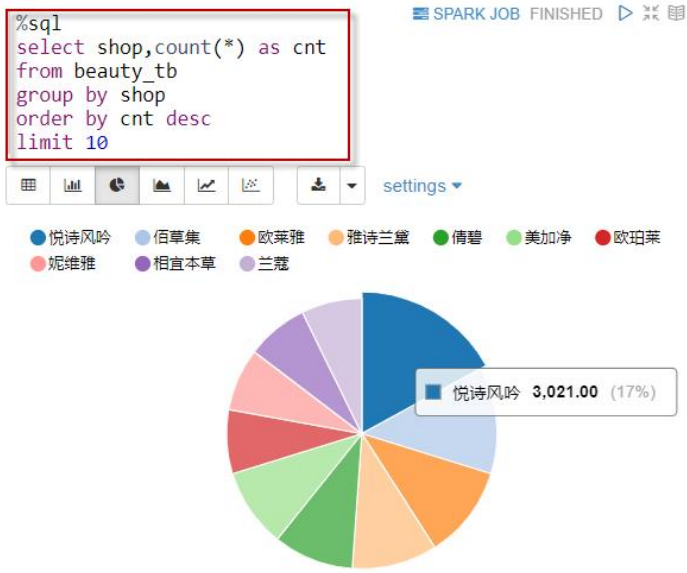


图 2-1　添加系统变量

(5) 品牌总销量 Top 10 统计

接下来按总销量统计 Top 10 的品牌，实现代码如下。

```
spark.sql("""
  select shop as `品牌`, sum(sale_count) as `品牌总销量`
  from beauty_tb
  group by shop
  order by `品牌总销量` desc
  limit 10
""").show
```

执行以上代码，输出内容如下：

```
+--------+----------+
|      品牌|品牌总销量|
+--------+----------+
| 相宜本草|  65462947|
|   美宝莲|  39358088|
| 悦诗风吟|  39070496|
```

```
|   妮维雅|   38421702|
|   欧莱雅|   33997797|
|   自然堂|   18081479|
| 蜜丝佛陀|   15391247|
|   佰草集|   14998206|
|     兰芝|    9135514|
|   美加净|    8825906|
+--------+-----------+
```

由结果可知，销量排名前 5 位的分别是：相宜本草、美宝莲、悦诗风吟、妮维雅、欧莱雅。以饼状图可视化显示总销量 Top 10 的品牌，SQL 语句如下：

```
%sql
select shop as `品牌`, sum(sale_count) as `品牌总销量`
from beauty_tb
group by shop
order by `品牌总销量` desc
limit 10
```

执行以上语句，点击饼状图显示图片，查询结果将以饼状图呈现，结果如图所示。

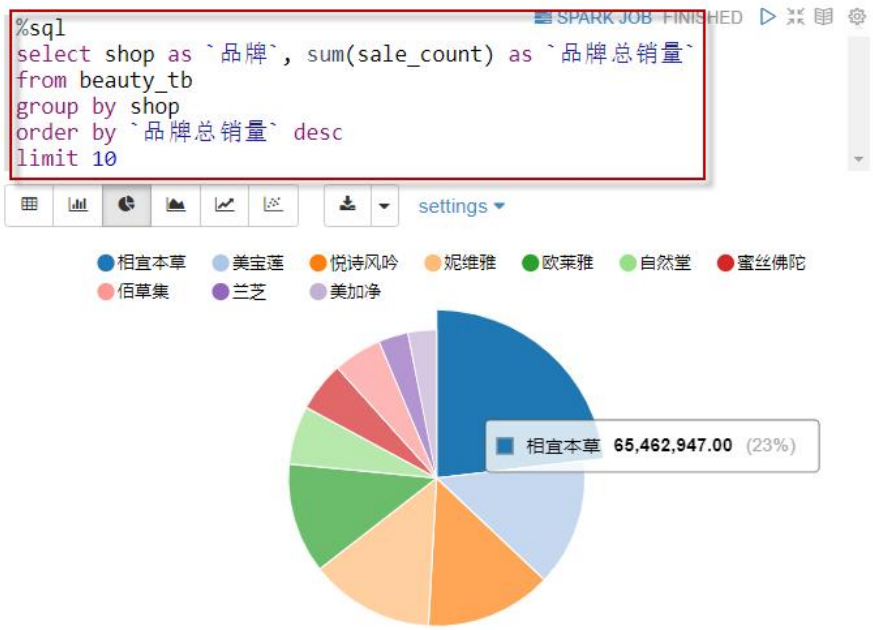


图 2-1　添加系统变量

(6) 品牌均价统计

接下来分析不同品牌的销售均价，SQL 查询语句如下：

```
%sql
select shop as `品牌`, avg(price) as `均价`
from beauty_tb
group by shop
order by `均价` desc
```

执行以上语句，点击柱状图显示图片，查询结果将以柱状图呈现，结果如图所示。

```
%sql
select shop as `品牌`, avg(price) as `均价`
from beauty_tb
group by shop
order by `均价` desc
```

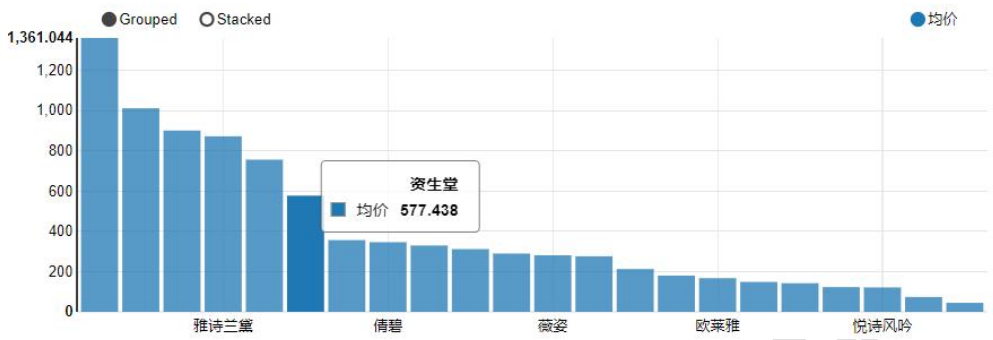


图 2-1　添加系统变量

(7) 各品牌均价与销量的关系

接下来分析不同品牌各品牌均价与销量的关系，SQL 查询语句如下：

```
%sql
select shop as `品牌`,
       avg(price) as `均价`,
       sum(sale_count)/50000 as `销量`
from beauty_tb
group by shop
order by `均价`
```

在上面的语句中，因为均价和销售量在量纲上有巨大的差距，为此将销量除以 50000 以便进行比较。执行以上语句，点击柱状图显示图片，查询结果将以柱状图呈现，结果如图所示。

```
%sql
select shop as `品牌`, avg(price) as `均价`, sum(sale_count)/50000 as `销量`
from beauty_tb
group by shop
order by `均价`
```

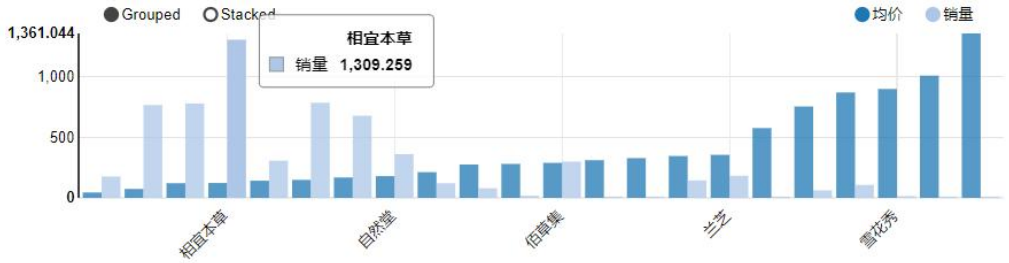


图 2-1　添加系统变量

由图 1 和图 2 可知：高于全品牌平均价格的 6 个大品牌（娇兰、SKII、雪花秀、雅诗兰黛、兰蔻、资生堂）在活动期间销量、热度都比较低；平均价格较低的美宝莲、蜜丝佛陀、相宜本草、妮维雅等品牌在销量上有不错的表现；百元左右的美妆销售最好。尽管大品牌价格远高于全品牌平均价格，但中低价格的品牌多，销量大，使得整个全品牌平均价格低于 400 元。

以上是可以进行的一些分析，但是还有一些分析是无法做的，比如，销量和评论量与时间的关系分析，或者说时序分析或趋势分析无法执行，因为当前的日期字段是字符串类型。为此，需要进行相应的数据预处理任务才能进行这样的分析。

2. 3 Spark 数据预处理

海量的原始数据中存在着大量不完整（有缺失值）、不一致、有异常的数据，这将影响到数据分析和挖掘建模的执行效率，并有可能导致结果偏差，所以进行数据清洗就非常有必要。在数据清洗完成后，在正式数据分析和挖掘建模之前，还需要对数据进行数据集成、属性转换和规约等一系列处理。数据清洗及之后的数据处理过程就是数据预处理。统计发现，在数据分析和挖掘过程中，数据预处理工作量占到了整个过程的 60%。

数据预处理的主要内容包括数据清洗、数据集成、数据变换和数据归约等。

2. 3. 1 Spark 数据清洗

数据清洗主要是删除原始数据集中的无关数据、重复数据，处理缺失值、异常值等。

1. 数据去重

因为某些原因，数据集中可能会存在重复的数据记录。有可能是整条记录的完全重复，也有可能是某些关键字段（比如 id）的重复。对于整条记录的完全重复，一般的处理原则是去重，即只保留其中一条即可。对于某些关键字段重复的记录，则根据业务需求来判断是否要执行去重操作。

在 Spark 中，去重类型的操作主要包括 dropDuplicates()和 distinct()。

(1) dropDuplicates()

方法描述：使用 dropDuplicates()删除某些列中的重复行。示例代码如下。

```
// 构造一个简单 DataFrame
val df = spark.sparkContext.parallelize(List(
    ("张老三", 5, 80),
    ("张老三", 5, 80),
    ("张老三", 10, 80)
 )).toDF("name","age","height")

// 去掉重复的行
df.dropDuplicates().show()
```

执行上面的代码，输出内容如下：

```
+------+---+------+
|  name|age|height|
+------+---+------+
| 张老三|  5|    80|
| 张老三| 10|    80|
+------+---+------+
```

上面这个示例比较数据集中的两行，只有当两行中各个字段都完全相同的情况下才认为是重复数据。如果只想比较其中若干个字段是否重复，则可将比较字段作为参数传入。修改上面的代码，当 name 和 height 字段重复时，就删除重复的行，实现代码如下。

```
// 删除 name 列和 height 列重复的行
```

```
df.dropDuplicates("name", "height").show()
```

执行以上代码，输出内容如下：

```
+------+---+------+
|  name|age|height|
+------+---+------+
| 张老三|  5|    80|
+------+---+------+
```

(2) distinct()

方法描述：当 dropDuplicates()中没有指定列名时，意味着去重所有的列。dropDuplicates()方法也有一个别名，称为 distinct()。

因此，使用 distinct 也可以去重复，并且只能根据所有列去重。重构前面的代码，实现代码如下。

```
// 构造一个简单 DataFrame
val df = spark.sparkContext.parallelize(List(
    ("张老三", 5, 80),
    ("张老三", 5, 80),
    ("张老三", 10, 80)
 )).toDF("name","age","height")

// 去掉重复的行
df.distinct().show()
```

执行以上代码，输出内容如下：

```
+------+---+------+
|  name|age|height|
+------+---+------+
| 张老三|  5|    80|
| 张老三| 10|    80|
+------+---+------+
```

2. 缺失值处理

对于数据集中缺失值的处理，一般有三种方法：删除带有缺失值的记录、缺失数据填充以及不处理。

为了演示 Spark 中缺失值处理方法，首先构造一个带有缺失值的 DataFrame，代码如下。

```
import org.apache.spark.sql.types._
import org.apache.spark.sql._

// 定义模式 schema
val fields = List(
  StructField("emp_id", IntegerType, nullable = true),
  StructField("name", StringType, nullable = true),
  StructField("age", IntegerType, nullable = true),
  StructField("role", StringType, nullable = true),
  StructField("salary", DoubleType, nullable = true)
)

val schema = StructType(fields)

// 数据
val employees = Seq(
```

```
    Row(1, "陈柯宇", 25, "合伙人", 10000.00),
    Row(2, "陶心瑶", 35, "经理", 12000.00),
    Row(3, "楼一萱", 24, "经理", 12000.00),
    Row(4, "张希", 28, "合伙人", null),
    Row(5, "王心凌", 26, "经理", 120.00),
    Row(6, "庄妮", 35, "高级经理", 22000.00),
    Row(7,"何洁", 38, "高级经理", 20000.00),
    Row(8, "成方圆", 32, "经理", 12000.00),
    Row(9, "孙玉", 29, "经理", 10000.00),
    Row(10, "黄渤",null, "合伙人", null),
    Row(11, "刘珂矣", 29, "合伙人", 8000.00),
    Row(12, "林忆莲", 28, "经理", 12000.00),
    Row(13, "蓝琪儿", 25, "经理", 12000.00),
    Row(14, "白安",31, "经理", 120000.00),
    Row(15, "黄渤",null, null, null)
)

// 创建 DataFrame
val employeesDF = spark.createDataFrame(sc.parallelize(employees),  schema)

// 查看
employeesDF.show()
```

执行以上代码，输出内容如下：

```
+------+------+----+--------+--------+
|emp_id|  name| age|    role|  salary|
+------+------+----+--------+--------+
|     1| 陈柯宇|  25|   合伙人| 10000.0|
|     2| 陶心瑶|  35|     经理| 12000.0|
|     3| 楼一萱|  24|     经理| 12000.0|
|     4|   张希|  28|   合伙人|    null|
|     5| 王心凌|  26|     经理|   120.0|
|     6|   庄妮|  35|  高级经理| 22000.0|
|     7|   何洁|  38|  高级经理| 20000.0|
|     8| 成方圆|  32|     经理| 12000.0|
|     9|   孙玉|  29|     经理| 10000.0|
|    10|   沈腾|null|   合伙人|    null|
|    11| 刘珂矣|  29|   合伙人|  8000.0|
|    12| 林忆莲|  28|     经理| 12000.0|
|    13| 蓝琪儿|  25|     经理| 12000.0|
|    14|   白安|  31|     经理|120000.0|
|    15|   黄渤|null|    null|    null|
+------+------+----+--------+--------+
```

Spark 中提供了一个处理 DataFrames 中缺失数据的函数类 DataFrameNaFunctions，它位于 org.apache.spark.sql 包中。要获取一个 DataFrame 的 DataFrameNaFunctions 对象，可以调用 na 零参函数。例如，要获取 employeesDF 的 DataFrameNaFunctions 对象，实现代码如下。

```
employeesDF.na
```

1）删除包含缺失值的行

这处理缺失值的三种方法中，直接删除含有缺失值的记录这种方法最简单。一般实践中，当含有缺失值的记录数不超过总记录数的 5%时，可直接将其删除。

DataFrameNaFunctions 提供了多个重载的 drop()方法，用来在不同情况下删除带有缺失值的行，其定义如下：

```
def drop(): DataFrame
def drop(how: String): DataFrame
def drop(cols: Array[String]): DataFrame
def drop(how: String, cols: Array[String]): DataFrame

def drop(minNonNulls: Int): DataFrame
def drop(minNonNulls: Int, cols: Array[String]): DataFrame
```

例如，要删除 employeesDF 中带有缺失值（null 值或 NaN 值）的行，可以调用 drop()方法或 drop(how="any")方法（这两个方法等价），实现代码如下。

```
employeesDF.na.drop().show
employeesDF.na.drop(how="any").show   // 与上一句等价
```

执行以上代码，输出内容如下：

```
+------+------+---+--------+--------+
|emp_id|  name|age|    role|  salary|
+------+------+---+--------+--------+
|     1| 陈柯宇| 25|   合伙人| 10000.0|
|     2| 陶心瑶| 35|    经理| 12000.0|
|     3| 楼一萱| 24|    经理| 12000.0|
|     5| 王心凌| 26|    经理|   120.0|
|     6|  庄妮| 35|  高级经理| 22000.0|
|     7|  何洁| 38|  高级经理| 20000.0|
|     8| 成方圆| 32|    经理| 12000.0|
|     9|  孙玉| 29|    经理| 10000.0|
|    11| 刘珂矣| 29|   合伙人|  8000.0|
|    12| 林忆莲| 28|    经理| 12000.0|
|    13| 蓝琪儿| 25|    经理| 12000.0|
|    14|  白安| 31|    经理|120000.0|
+------+------+---+--------+--------+
```

如果参数 how 的值是"any"，则删除包含任何 null 或 NaN 值的行。如果 how 的值是"all"，则只有当行的每一列为 null 或 NaN 时才删除该行。

另外也可以指定列名，当省略 how 参数时或 how="any"时，该 DataFrame 删除指定列中包含任何 null 或 NaN 值的行；当 how="all"时，则仅当指定的列为空或 NaN 时才删除行。例如，要删除 role 或 salary 中包含 null 或 NaN 值的行，实现代码如下。

```
employeesDF.na.drop(how="any", Array("role","salary")).show()
employeesDF.na.drop(Array("role","salary")).show()  // 等价
```

执行以上代码，输出内容如下：

```
+------+------+---+--------+--------+
|emp_id|  name|age|    role|  salary|
+------+------+---+--------+--------+
|     1|陈柯宇| 25|  合伙人| 10000.0|
|     2|陶心瑶| 35|   经理| 12000.0|
|     3|楼一萱| 24|   经理| 12000.0|
|     5|王心凌| 26|   经理|   120.0|
|     6|  庄妮| 35|高级经理| 22000.0|
|     7|  何洁| 38|高级经理| 20000.0|
|     8|成方圆| 32|   经理| 12000.0|
|     9|  孙玉| 29|   经理| 10000.0|
```

```
|    11|刘珂矣| 29|  合伙人|  8000.0|
|    12|林忆莲| 28|    经理| 12000.0|
|    13|蓝琪儿| 25|    经理| 12000.0|
|    14|  白安| 31|    经理|120000.0|
+------+------+---+--------+--------+
```

如果要删除 role 和 salary 列值均为 null 或 NaN 的行，实现代码如下。

```
employeesDF.na.drop(how="all", Array("role","salary")).show()
```

执行以上代码，输出内容如下：

```
+------+------+----+--------+--------+
|emp_id|  name| age|    role|  salary|
+------+------+----+--------+--------+
|     1|陈柯宇|  25|  合伙人| 10000.0|
|     2|陶心瑶|  35|    经理| 12000.0|
|     3|楼一萱|  24|    经理| 12000.0|
|     4|  张希|  28|  合伙人|    null|
|     5|王心凌|  26|    经理|   120.0|
|     6|  庄妮|  35|高级经理| 22000.0|
|     7|  何洁|  38|高级经理| 20000.0|
|     8|成方圆|  32|    经理| 12000.0|
|     9|  孙玉|  29|    经理| 10000.0|
|    10|  黄渤|null|  合伙人|    null|
|    11|刘珂矣|  29|  合伙人|  8000.0|
|    12|林忆莲|  28|    经理| 12000.0|
|    13|蓝琪儿|  25|    经理| 12000.0|
|    14|  白安|  31|    经理|120000.0|
+------+------+----+--------+--------+
```

如果在 drop()方法中指定 minNonNulls 参数，则表示删除包含小于 minNonNulls 个非 null 和非 NaN 值的行。例如，删除 employeesDF 中少于 4 个非空（非 null 或非 NaN）字段的行，实现代码如下。

```
employeesDF.na.drop(minNonNulls=4).show()
```

执行以上代码，输出内容如下：

```
+------+------+---+--------+--------+
|emp_id|  name|age|    role|  salary|
+------+------+---+--------+--------+
|     1|陈柯宇| 25|  合伙人| 10000.0|
|     2|陶心瑶| 35|    经理| 12000.0|
|     3|楼一萱| 24|    经理| 12000.0|
|     4|  张希| 28|  合伙人|    null|
|     5|王心凌| 26|    经理|   120.0|
|     6|  庄妮| 35|高级经理| 22000.0|
|     7|  何洁| 38|高级经理| 20000.0|
|     8|成方圆| 32|    经理| 12000.0|
|     9|  孙玉| 29|    经理| 10000.0|
|    11|刘珂矣| 29|  合伙人|  8000.0|
|    12|林忆莲| 28|    经理| 12000.0|
|    13|蓝琪儿| 25|    经理| 12000.0|
|    14|  白安| 31|    经理|120000.0|
+------+------+---+--------+--------+
```

带有 minNonNulls 参数的 drop()方法可以变相实现“删除至少包含 n 个缺失值的行”，此时设 minNonNulls=df.columns.length-n+1。注意，在 Spark 中没有直接删除至少包含 n 个缺失值的行的函数。

也可以指定将 minNonNulls 参数应用在哪些列上。例如，从 employeesDF 中删除 name、age、role、salary 这四个列中包含不少于 2 个非空字段的行（即删除这 4 列中至少包含 3 个空值的行）。实现代码如下。

```
employeesDF.na
    .drop(minNonNulls=2, Array("name","age","role","salary"))
    .show()
```

执行以上代码，输出内容如下：

```
+------+------+----+--------+--------+
|emp_id|  name| age|    role|  salary|
+------+------+----+--------+--------+
|     1| 陈柯宇|  25|   合伙人| 10000.0|
|     2| 陶心瑶|  35|    经理| 12000.0|
|     3| 楼一萱|  24|    经理| 12000.0|
|     4|   张希|  28|   合伙人|    null|
|     5| 王心凌|  26|    经理|   120.0|
|     6|   庄妮|  35|  高级经理| 22000.0|
|     7|   何洁|  38|  高级经理| 20000.0|
|     8| 成方圆|  32|    经理| 12000.0|
|     9|   孙玉|  29|    经理| 10000.0|
|    10|   黄渤|null|   合伙人|    null|
|    11| 刘珂矣|  29|   合伙人|  8000.0|
|    12| 林忆莲|  28|    经理| 12000.0|
|    13| 蓝琪儿|  25|    经理| 12000.0|
|    14|   白安|  31|    经理|120000.0|
+------+------+----+--------+--------+
```

2）填充缺失值

直接删除包含缺失值的行的方法虽然简单，但该方法有着一定的局限性。它有可能丢弃了大量隐藏在这些记录中的信息，尤其在数据信本来就包含很少记录的情况下，删除少量记录就可能严重影响分析结果的客观性和正确性。一般实践中，当含有缺失值的记录数超过总记录数的 5%时，我们就需要对缺失值进行填充而不是直接删除掉。

DataFrameNaFunctions 提供了多个重载的 fill()方法，用来将所有列或指定列中的 null 或 NaN 值替换为一个常量。最经常使用的一个 fill()版本，其定义如下：

```
def fill(valueMap: Map[String, Any]): DataFrame
```

其使用方法如下：

```
df.na.fill(Map("列名"->填充值))
```

例如，使用平均薪资填充 employeesDF 中薪资（salary 字段）的缺失值，使用最频繁项填充 employeesDF 中职位（role 字段）的缺失值，使用中位数填充 employeesDF 中年龄（age 字段）的缺失值。实现代码如下。

```
// 统计摘要信息
val employeesSummary = employeesDF
    .select("age","salary")
```

```
    .summary("mean","50%")

// employeesSummary.show()
/*
+-------+-----------------+-----------------+
|summary|              age|           salary|
+-------+-----------------+-----------------+
|   mean|29.615384615384617|20843.333333333332|
|    50%|               29|          12000.0|
+-------+-----------------+-----------------+
*/

// 计算平均薪资
val salary_avg = employeesSummary
    .where(col("summary")==="mean")
    .first.getAs[String]("salary")
    .toDouble.formatted("%.2f")

// 计算 age 中位数
val age_median = employeesSummary
    .where(col("summary")==="50%")
    .first.getAs[String]("age")
    .toInt

// 计算 role 的最频繁项
val role_freq = employeesDF.stat
    .freqItems(Array("role"),0.8)
    .first.get(0)
    .asInstanceOf[scala.collection.mutable.WrappedArray[String]].head
// 或者
/*
val role_freq = employeesDF.stat
    .freqItems(Array("role"),0.8)
    .select(explode(col("role_freqItems")))
    .first.getAs[String](0)
*/

// 填充
employeesDF.na.fill(Map(
       "age"->age_median,
       "role"->role_freq,
       "salary"->salary_avg
   )).show()
```

执行以上代码，输出内容如下：

```
+------+------+---+--------+--------+
|emp_id|  name|age|    role|  salary|
+------+------+---+--------+--------+
|     1| 陈柯宇| 25|  合伙人| 10000.0|
|     2| 陶心瑶| 35|    经理| 12000.0|
|     3| 楼一萱| 24|    经理| 12000.0|
|     4|   张希| 28|  合伙人|20843.33|
|     5| 王心凌| 26|    经理|   120.0|
|     6|   庄妮| 35|高级经理| 22000.0|
|     7|   何洁| 38|高级经理| 20000.0|
|     8| 成方圆| 32|    经理| 12000.0|
|     9|   孙玉| 29|    经理| 10000.0|
```

```
|     10|   沈腾| 29|   合伙人|20843.33|
|     11| 刘珂矣| 29|   合伙人|  8000.0|
|     12| 林忆莲| 28|    经理| 12000.0|
|     13| 蓝琪儿| 25|    经理| 12000.0|
|     14|   白安| 31|    经理|120000.0|
|     15|   黄渤| 29|    经理|20843.33|
+------+------+---+--------+--------+
```

3. 异常值处理

异常值也叫离群值，是指那些偏离正常范围的值，不是错误值，例如月薪超平均月薪三倍、温度远超正常温度范围等。异常值出现频率较低，但又会对实际项目分析造成偏差。

在数据预处理时，异常值是否剔除，需视具体情况而定，因为有些异常值可能蕴含着有用的信息。异常值处理常用方法见下表：

表 2-4 异常处理常用方法

异常值处理方法	方法描述
删除含有异常值的记录	直接将含有异常值的记录删除
视为缺失值	将异常值视为缺失值，利用缺失值处理的方法进行处理
平均值修正	可用前后两个观测值的平均值修正该异常值
不处理	直接在具有异常值的数据集上进行数据挖掘和建模

例如，找出 employeesDF 中的异常薪资（异常薪资设定为超过平均薪资两个标准差的薪资），并用平均薪资来替换。实现代码如下。

```
// 计算每一行的偏差
val devs = employeesDF
       .select((
            ($"salary" - salary_avg) * ($"salary" - salary_avg)
          ).alias("deviation")
         )

// 计算标准差
val stddev = devs.select(sqrt(avg("deviation"))).first().getDouble(0)

// UDF: 用平均工资替换超过 2 个标准差范围内的异常值
val outlierfunc = udf((value: Long, mean: Double) => {
   if (value > mean+(2*stddev) || value < mean-(2*stddev))
      mean
   else
      value
})

// 应用自定义函数，替换异常值
val no_outlier = employeesDF
     .withColumn("updated_salary",
                 outlierfunc(col("salary"),lit(salary_avg)))

// 观察修改后的值
no_outlier.filter($"salary" =!= $"updated_salary").show()
```

执行以上代码，输出内容如下：

```
+------+----+---+----+--------+--------------+
|emp_id|name|age|role|  salary|updated_salary|
```

```
+------+----+---+----+--------+--------------+
|    14| 白安| 31|经理|120000.0|      20843.33|
+------+----+---+----+--------+--------------+
```

从输出内容可知，工号为 14 的员工薪资异常，因此使用了全体员工的平均薪资来进行替换。

2. 3. 2 Spark 数据集成

在企业中，由于开发时间或开发部门的不同，往往有多个异构的、运行在不同的软硬件平台上的信息系统同时运行，这些系统的数据源彼此独立、相互封闭，使得数据难以在系统之间交流、共享和融合，从而形成了“信息孤岛”。

随着信息化应用的不断深入，企业内部、企业与外部信息交互的需求日益强烈，急切需要对已有的信息进行整合，联通“信息孤岛”，共享信息，这些信息数据整合的一系列方案被称为数据集成。

通俗地理解，数据集成，也就是指将多份数据进行合并成数据集的过程和方法。数据集成最常见的两种方法是：数据关联与数据合并。数据关联代表两份数据表做左右联接，而数据合并代表两份数据表做上下（垂直）联接。如图 2-4 所示。

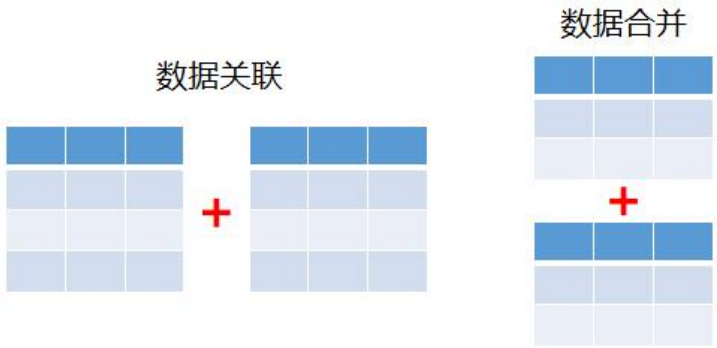


图 2-4　数据关联与数据合并

接下来分别了解在 Spark 中实现数据关联和数据合并的方法。

1. 数据关联

一旦数据从不同的来源获得，接一来就是将它们全部合并，以便将数据作为一个整体进行清理、格式化，并转换为分析所需的格式。Spark SQL 支持对两个或多个 DataFrame/Dataset 执行各种类型的 join 连接操作。

执行两个数据集的连接需要指定两个内容：

(1) 第一个是连接表达式，它指定来自每个数据集的哪些列应该用于确定来自两个数据集的哪些行将被包含在连接后的数据集中（确定连接列/等值列）。

(2) 第二种是连接类型，它决定了连接后的数据集中应该包含哪些内容。

为了演示如何在 Spark SQL 中使用 join 连接，需要先准备两个小型的 DataFrame。第一个 DataFrame 代表一个员工列表，每一行包含员工姓名和所属部门。第二个 DataFrame 包含一个部门列表，每一行包含一个部门 ID 和部门名称。代码如下。

```
// 首先创建两个 case class，分别代表员工类和部门类：
// 员工
```

```
case class Employee(emp name:String, dept no:Long)

// 部门
case class Dept(id:Long, name:String)

// 然后创建两个测试 DataFrame
val employeeDF = Seq(
    Employee("刘宏明", 31),
    Employee("赵薇", 33),
    Employee("黄海波", 33),
    Employee("杨幂", 34),
    Employee("楼一萱", 34),
    Employee("龙梅子", null.asInstanceOf[Int])
).toDF

val deptDF = Seq(
    Dept(31, "销售部"),
    Dept(33, "工程部"),
    Dept(34, "财务部"),
    Dept(35, "市场营销部")
).toDF
```

对 employeeDF 和 deptDF 这两个数据集按部门编号执行连接，代码如下。

```
// 执行该 join
employeeDF.join(deptDF, col("dept_no") === col("id"), "inner").show

// 也可以不需要指定 join 类型，因为"inner"是默认的
// employeeDF.join(deptDF, col("dept_no") === col("id")).show
```

执行上面的代码，输出结果如下：

```
+--------+-------+---+------+
|emp_name|dept_no| id|  name|
+--------+-------+---+------+
|    刘宏明|     31| 31|  销售部|
|      赵薇|     33| 33|  工程部|
|    黄海波|     33| 33|  工程部|
|      杨幂|     34| 34|  财务部|
|    楼一萱|     34| 34|  财务部|
+--------+-------+---+------+
```

前面我们使用的是关系函数执行联接。也可以将这两个 DataFrame 注册为临时视图，然后使用 SQL 语句进行联接，代码如下。

```
// 注册为临时视图
employeeDF.createOrReplaceTempView("employees")
deptDF.createOrReplaceTempView("departments")

// 使用 SQL 执行 join 连接
spark
  .sql("select * from employees JOIN departments on dept_no == id")
  .show
```

执行以上代码，输出内容如下：

```
+--------+-------+---+------+
|emp name|dept no| id|  name|
+--------+-------+---+------+
|    刘宏明|     31| 31|  销售部|
```

```
|      赵薇|     33| 33|  工程部|
|    黄海波|     33| 33|  工程部|
|      杨幂|     34| 34|  财务部|
|    楼一萱|     34| 34|  财务部|
+--------+-------+---+------+
```

在实际应用中，更多地是组合从各种数据源获得的数据。假设这样的场景：员工数据分散存储在本地的 RDD、JSON 文件和关系型数据库中。下面演示了从不同数据源（内存集合、文件等）加载数据到 DataFrame 中，并将这些数据关联在一起。

```
// 创建一个 RDD 并转换为 DataFrame
val data = List(
              (1, "陈柯宇", 25), (2, "陶心瑶", 35),(3, "楼一萱", 24),
              (4, "张希", 28), (5, "王心凌", 26), (6, "庄妮", 35),
              (7,"何洁", 38), (8, "成方圆", 32), (9, "孙玉", 29),
              (10, "刘珂矣", 29),(11, "林忆莲", 28),
              (12, "蓝琪儿", 25),(13, "白安",31)
            )

val employeesDF = sc.parallelize(data).toDF("emp id","name","age")

// 加载 json 格式的数据源文件
val salaryFilePath = "/data/spark/data analyze/salary.json"
val salaryDF = spark.read.json(salaryFilePath)

val designationFilePath = "/data/spark/data analyze/designation.json"
val designationDF = spark.read.json(designationFilePath)

// 数据整合：组合从各种数据源获得的数据。
val final data = employeesDF
    .join(salaryDF,$"emp id"===$"e id")
    .join(designationDF,$"emp id"===$"id")
    .select("emp_id","name","age","role","salary")

final_data.show
```

执行以上代码，输出内容如下：

```
+------+------+---+--------+------+
|emp_id|  name|age|    role|salary|
+------+------+---+--------+------+
|     1| 陈柯宇| 25|   合伙人| 10000|
|     2| 陶心瑶| 35|    经理| 12000|
|     3| 楼一萱| 24|    经理| 12000|
|     4|  张希| 28|   合伙人|  null|
|     5| 王心凌| 26|    经理|   120|
|     6|  庄妮| 35|  高级经理| 22000|
|     7|  何洁| 38|  高级经理| 20000|
|     8| 成方圆| 32|    经理| 12000|
|     9|  孙玉| 29|    经理| 10000|
|    10| 刘珂矣| 29|   合伙人|  8000|
|    11| 林忆莲| 28|    经理| 12000|
|    12| 蓝琪儿| 25|    经理| 12000|
|    13|  白安| 31|    经理|120000|
+------+------+---+--------+------+
```

2. 数据合并

数据合并，也称为数据追加，是指对多份数据字段基本完全相同的数据进行上下连接。在 Spark 中，这可以通过 union()方法或 unionAll()方法实现。这两个方法等价，都是按列位置合并两个数据集的行。

下面这个示例演示了具有相同 Schema 的数据集的合并操作。

```
// 创建具有相同 Schema 的三个数据集
val df1 = Seq(("张三",23,"工程师"),
              ("李四",27,"会计师"),
              ("王老五",28,"医生")
             ).toDF("name","age","profession")
val df2 = Seq(("张三",23,"工程师"),
              ("赵小六",27,"律师")
             ).toDF("name","age","profession")
val df3 = Seq(("周扒皮",25,"数据分析师")).toDF("name","age","profession")

 // df1 和 df2 是两个具有相同模式，执行 uinon 合并
df1.union(df2).show
```

执行以上代码，输出内容如下：

```
+------+---+----------+
|  name|age|profession|
+------+---+----------+
|   张三| 23|      工程师|
|   李四| 27|      会计师|
| 王老五| 28|        医生|
|   张三| 23|      工程师|
| 赵小六| 27|        律师|
+------+---+----------+
```

从输出结果中，我们观察到重复的记录没有被删除。如果合并后需要删除重复项，则需要使用 distinct()或 dropDuplicates()。代码如下：

```
df1.union(df2).distinct().show
// df1.union(df2).dropDuplicates().show  // 等价上一句
```

执行以上代码，输出内容如下：

```
+------+---+----------+
|  name|age|profession|
+------+---+----------+
| 王老五| 28|        医生|
|   张三| 23|      工程师|
| 赵小六| 27|        律师|
|   李四| 27|      会计师|
+------+---+----------+
```

当有多个 DataFrame 需要合并时，可以首先创建了一个 DataFrame 的序列集合，然后联合 reduce()方法来执行合并操作。例如，将 df1、df2 和 df3 合并在一起，代码如下。

```
val dfs = Seq(df1,df2,df3)
val df123 = dfs.reduce(_ union _)   // 或者 df1.union(df2).union(df3)
df123.distinct().show()
```

执行以上代码，输出内容如下：

```
+------+---+----------+
```

```
|  name|age|profession|
+------+---+----------+
|  王老五| 28|      医生|
|  周扒皮| 25| 数据分析师|
|    张三| 23|    工程师|
|  赵小六| 27|      律师|
|    李四| 27|    会计师|
+------+---+----------+
```

以上执行合并的数据集都具有相同的 Schema。那么具有不完全相同 Schema 的数据集该如何合并呢？下面的示例演示了这种场景下的合并示例。

为了代码更加简洁，我们首先创建一个自定义函数，它在 DataFrame 上施加模式演化。也就是说，它从 total_cols 中添加额外的列及其值到当前的 Dataframe 中。代码如下。

```
import org.apache.spark.sql.functions._

/**
 * myCols: 当前 DataFrame 的所有列
 * allCols: 合并后的 DataFrame 的所有列
 */
def expr(myCols: Set[String], allCols: Set[String]) = {
  allCols.toList.map(x => x match {
      // 如果是当前 DataFrame 具有的列，则是取当前列值
      case x if myCols.contains(x) => col(x)
      // 否则，取 null 值
      case   => lit(null).as(x)
  })
}
```

前面的 df123 是合并了 df1、df2 和 df3 后的 DataFrame，下面创建一个具有不同列的 DataFrame df4，然后将它们列进行统一，再进行合并。

```
// 首先，创建一个具有不同 Schema 的数据集
val df4 = Seq(("张三","人力资源",50000),
            ("李四","会计师",60000),
            ("钱多多","软件工程师",45000),
            ("赵小六","律师",70000),
            ("大 boss","ceo",100000)
        ).toDF("name","profession","salary")

// 创建一个包含 df4 和 df123 列的集合 total cols
val cols123 = df123.columns.toSet
val cols4 = df4.columns.toSet
val total cols = cols123 ++ cols4  // 所有列的集合

// 分别扩展两个 DataFrame 的列
val modified df4 = df4.select(expr(cols4, total cols): *)
val modified_df123 = df123.select(expr(cols123, total_cols):_*)

// 然后执行合并操作
val final_df = modified_df4.union(modified_df123).distinct()

// 查看
final_df.show()
```

执行以上代码，输出内容如下：

```
+------+----+----------+------+
|  name| age|profession|salary|
+------+----+----------+------+
|大boss|null|       ceo|100000|
| 王老五|  28|      医生|  null|
| 赵小六|null|      律师| 70000|
| 周扒皮|  25|  数据分析师|  null|
| 钱多多|null|  软件工程师| 45000|
|   张三|null|    人力资源| 50000|
|   李四|null|     会计师| 60000|
|   张三|  23|     工程师|  null|
| 赵小六|  27|      律师|  null|
|   李四|  27|     会计师|  null|
+------+----+----------+------+
```

从输出结果可以观察到，df123 和 df4 的不常见列，年龄和工资将在结果数据集中用空值（null）填充。

2. 3. 3 Spark 属性转换

原始数据集并不是所有的属性都能直接使用，有的属性列需要进一步的转换，以便于后续的分析。例如，不规范的日期列，需要将日期进行格式化转换；带有货币符号的金额列，需要去除货币符号以便进行数值计算。再例如，订单表中只有一个日期列，但是统计指标中有按年、按月或按日进行统计的要求，那么就需要从日期列通过抽取计算派生出新的列。诸如此类，都需要对原始数据集的属性进行相应的转换计算。

实际生产环境中，存在有各种各样的数据转换需求，这里我们将讨论一些基本类型的转换，如下所示：

(1) 将两列合并成一列。

(2) 将字符/数字添加到现有的字符/数字。

(3) 从现有的字符/数字中删除或替换字符/数字。

(4) 更改日期的格式。

下面的程序代码演示了如何对数据进行一些基本的转换。例如，合并已经存在的两个列，创建一个新的列，代码如下。

```
import org.apache.spark.sql.functions._

// 创建 DataFrame
val df1 = Seq(("张三",23,"工程师"),
              ("李四",27,"会计师"),
              ("王老五",28,"医生"),
              ("赵小六",27,"律师"),
              ("周扒皮",25,"数据分析师")
).toDF("name","age","profession")

// 增加一个新的列，列值来自于 name 和 age 列的合并
val concat_df = df1
       .withColumn("name_age",concat($"name", lit("-"), $"age"))

// 查看结果
concat_df.show
```

执行以上代码，输出内容如下：

```
+------+---+----------+---------+
|  name|age|profession| name age|
+------+---+----------+---------+
|   张三| 23|      工程师|   张三-23|
|   李四| 27|      会计师|   李四-27|
| 王老五| 28|        医生| 王老五-28|
| 赵小六| 27|        律师| 赵小六-27|
| 周扒皮| 25|  数据分析师| 周扒皮-25|
+------+---+----------+---------+
```

下面的代码实现向数据添加常量：所有人的年龄增加 1 岁。

```
df1.withColumn("age", $"age"+1).show
```

执行以上代码，输出内容如下：

```
+------+---+----------+
|  name|age|profession|
+------+---+----------+
|   张三| 24|      工程师|
|   李四| 28|      会计师|
| 王老五| 29|        医生|
| 赵小六| 28|        律师|
| 周扒皮| 26|  数据分析师|
+------+---+----------+
```

如果要替换一个列中的值，使用 na.replace()方法。例如，要将 profession 列中的“会计师”职业修改为“会计”，“数据分析师”职业修改为“商业分析师”，代码如下。

```
df1.na.replace("profession",
                Map("会计师" -> "会计", "数据分析师" -> "商业分析师")
   ).show
```

执行以上代码，输出内容如下：

```
+------+---+----------+
|  name|age|profession|
+------+---+----------+
|   张三| 23|      工程师|
|   李四| 27|        会计|
| 王老五| 28|        医生|
| 赵小六| 27|        律师|
| 周扒皮| 25|  商业分析师|
+------+---+----------+
```

如果在 replace 中列名参数是"*"，那么替换应用到所有的列。

现在我们已经熟悉了一些基本的例子，让我们来看一个比较复杂的例子。在下面的示例中，我们看到的是日期列有许多不同日期格式的情况。现在需要将所有不同的日期格式标准化成一种格式。首先构造出这样的一个数据集。

```
// 构造数据集
case class Book (title: String, author: String, pubtime: String)

val books = Seq(Book("浮生六记","[清]沈复","2018/7/1"),
                Book("云边有个小卖部","张嘉佳","2018/07/12"),
                Book("菊与刀","[美]本尼迪克特",null),
                Book("苏菲的世界","乔斯坦·贾德","2017,10 12"),
                Book("罗生门",null,null)
               )
```

```
val booksDF = sc.parallelize(books).toDF()
booksDF.show
```

执行以上代码，输出内容如下：

```
+--------------+--------------+----------+
|         title|        author|   pubtime|
+--------------+--------------+----------+
|        浮生六记|        [清]沈复|  2018/7/1|
|   云边有个小卖部|          张嘉佳|2018/07/12|
|          菊与刀|   [美]本尼迪克特|      null|
|      苏菲的世界|     乔斯坦·贾德|2017,10 12|
|          罗生门|          null|      null|
+--------------+--------------+----------+
```

接下来创建一个用户定义的函数（udf），它可以处理不同的日期格式，将它们转换为一种标准日期格式。然后在属性转换中应用这个 udf 函数。

```
// 定义 udf：将传入的字符串转换成 YYYY-MM-DD 格式
def toDateUDF = udf((s: String) => {
    var (year, month, day) = ("","","")

    // 格式化日期
    if(s != null) {
        var x = s.split(" |/|,")    // 拆分
        year = "%04d".format(x(0).toInt)
        month = "%02d".format(x(1).toInt)
        day = "%02d".format(x(2).toInt)

        year + "-" + month + "-" + day
    } else {
        null
    }
})

// 应用 udf 并将日期字符串转换为标准的形式 YYYY-MM-DD
booksDF.withColumn("pubtime",toDateUDF($"pubtime")).show
```

执行以上代码，输出内容如下：

```
+--------------+--------------+----------+
|         title|        author|   pubtime|
+--------------+--------------+----------+
|        浮生六记|       [清]沈复|2018-07-01|
|   云边有个小卖部|         张嘉佳|2018-07-12|
|          菊与刀|  [美]本尼迪克特|      null|
|      苏菲的世界|    乔斯坦·贾德|2017-10-12|
|          罗生门|          null|      null|
+--------------+--------------+----------+
```

对日期格式的转换，也可以利用日期和时间处理函数。例如，下面示例中得到的日期是不规范的，这就需要将这些不规范的日期转换为规范的表示。对不规范日期的转换，代码如下。

```
// 第 5 章/functions_date.scala

import spark.implicits._
```

```
// 转换不规范的日期:
val df = Seq(
     "Nov 05, 2018 02:46:47 AM",
     "Nov 5, 2018 02:46:47 PM"
  ).toDF("times")

df.withColumn("times2",
     from unixtime(
       unix timestamp($"times", "MMM d, yyyy hh:mm:ss a"),
       "yyyy-MM-dd HH:mm:ss.SSSSSS"
     )
).show(false)
```

执行以上代码，输出结果如下：

```
+------------------------+--------------------------+
|times                   |times2                    |
+------------------------+--------------------------+
|Nov 05, 2018 02:46:47 AM|2018-11-05 02:46:47.000000|
|Nov 5, 2018 02:46:47 PM |2018-11-05 14:46:47.000000|
+------------------------+--------------------------+
```

在处理时间序列数据（time-series data）时，经常需要提取日期或时间戳值的特定字段（如年、月、小时、分钟和秒）。例如，当需要按季度、月或周对所有股票交易进行分组时，就可以从交易日期提取该信息，并按这些值分组。从日期或时间戳中提取字段，代码如下。

```
// 提取日期或时间戳值的特定字段（如年、月、小时、分钟和秒）
// 从一个日期值中提取指定的字段
val valentimeDateDF = Seq("2019-02-14 13:14:52").toDF("date")
valentimeDateDF.select(
     year('date).as("year"),                    // 年
     quarter('date).as("quarter"),              // 季
     month('date).as("month"),                  // 月
     weekofyear('date).as("woy"),               // 周
     dayofmonth('date).as("dom"),               // 日
     dayofyear('date).as("doy"),                // 天
     hour('date).as("hour"),                    // 小时
     minute('date).as("minute"),                // 分
     second('date).as("second")                 // 秒
   ).show()
```

执行以上代码，输出结果如下：

```
+----+-------+-----+---+---+---+----+------+------+
|year|quarter|month|woy|dom|doy|hour|minute|second|
+----+-------+-----+---+---+---+----+------+------+
|2019|      1|    2|  7| 14| 45|   13|    14|    52|
+----+-------+-----+---+---+---+----+------+------+
```

2.4 Spark 数据预处理示例

前面已经了解了 Spark 数据探索和预处理的方法和思路。本节通过几个案例掌握如何在真实数据集上执行数据预处理任务。

2.4.1 京东股票历史数据预处理

现有一份从网上采集到的京东股票历史数据，保存在名为 JDHistoricalData.csv 的 CSV 文件中。我们将运用前面几节所掌握的预处理技术，对该数据集执行数据探索和预处理工作。

首先，加载该数据集到 DataFrame 中，代码如下。

```
// 加载数据文件到 DataFrame
val jdFile = "/data/spark/JDHistoricalData.csv"
val jdDf = spark.read.option("header","true").csv(jdFile)

// 查看
jdDf.printSchema()
jdDf.show(5)
```

执行以上代码，输出内容如下：

```
root
 |-- Date: string (nullable = true)
 |-- Close/Last: string (nullable = true)
 |-- Volume: string (nullable = true)
 |-- Open: string (nullable = true)
 |-- High: string (nullable = true)
 |-- Low: string (nullable = true)

+----------+----------+-------+-------+------+------+
|      Date|Close/Last| Volume|   Open|  High|   Low|
+----------+----------+-------+-------+------+------+
|02/15/2022|    $76.13|6766205| $75.35|$76.35| $74.8|
|02/14/2022|    $74.45|5244967| $73.94|$74.62|$73.01|
|02/11/2022|    $73.98|6673354| $75.97|$76.55|$73.55|
|02/10/2022|     $76.4|6432184|$75.955|$78.39|$75.24|
|02/09/2022|    $78.29|7061571| $76.83|$78.67|$76.61|
+----------+----------+-------+-------+------+------+
only showing top 5 rows
```

观察输出内容，可以看到其中 Date 列的日期并不是标准日期格式（即 yyyy-MM-dd），同时所有价格都带有美元符号（$）。因此，我们的预处理任务包括以下几个方面：

(1) Date 字段：对日期进行格式化，转换为标准日期格式（即 yyyy-MM-dd）。

(2) Close/Last 字段：去掉$符号，转换数据类型到 Double，并重命名列。

(3) Open、High、Low 三个字段：去掉$符号。

对于 Date 字段的日期标准化转换，使用 to_date()函数即可。而要去掉金额前面的$符号，则可以使用正则表达式替换方法 regexp_replace()。具体实现如下。

```
import org.apache.spark.sql.functions._

val jdDf2 = jdDf
    .select(
       to_date(col("Date"),"MM/dd/yyyy").as("Date"),
       regexp_replace(col("Close/Last"), "\\$", "").as("Close"),
       col("Volume"),
       regexp_replace(col("Open"), "\\$", "").as("Open"),
       regexp_replace(col("High"), "\\$", "").as("High"),
       regexp_replace(col("Low"), "\\$", "").as("Low")
    )
    // 转换数据类型
```

```
    .selectExpr(
           "Date",
           "cast(Close as double) Close",
           "cast(Volume as bigint) Volume",
           "cast(Open as double) Open",
           "cast(High as double) High",
           "cast(Low as double) Low"
     )

// 查看转换后的结果
jdDf3.printSchema()
jdDf3.show(5)
```

执行以上代码，输出内容如下：

```
root
 |-- Date: date (nullable = true)
 |-- Close: double (nullable = true)
 |-- Volume: long (nullable = true)
 |-- Open: double (nullable = true)
 |-- High: double (nullable = true)
 |-- Low: double (nullable = true)

+----------+-----+-------+------+-----+-----+
|      Date|Close| Volume|  Open| High|  Low|
+----------+-----+-------+------+-----+-----+
|2022-02-15|76.13|6766205| 75.35|76.35| 74.8|
|2022-02-14|74.45|5244967| 73.94|74.62|73.01|
|2022-02-11|73.98|6673354| 75.97|76.55|73.55|
|2022-02-10| 76.4|6432184|75.955|78.39|75.24|
|2022-02-09|78.29|7061571| 76.83|78.67|76.61|
+----------+-----+-------+------+-----+-----+
only showing top 5 rows
```

将预处理后的数据存储到 HDFS 上，以备后续数据处理使用。实现代码如下。

```
// 写出到 parquet 文件
jdDf3.coalesce(1).write.parquet("/data/spark/jd2")

// 测试
spark.read.parquet("/data/spark/jd2").printSchema
```

执行以上代码，输出内容如下：

```
root
 |-- Date: date (nullable = true)
 |-- Close: double (nullable = true)
 |-- Volume: long (nullable = true)
 |-- Open: double (nullable = true)
 |-- High: double (nullable = true)
 |-- Low: double (nullable = true)
```

2.4.2 电商美妆销售数据预处理

在 2.2 节我们使用了某著名电商平台双十一美妆销售数据集，该数据集存储在一个 CSV 文件“双十一淘宝美妆数据.csv”中，并对其进行了数据探索。在这里我们在已经探索了解的基础上，进一步进行该数据的预处理任务。

首先加载该数据集到 DataFrame 中，并指定 Schema。代码如下。

```
// 指定要抽取的文件数据源
val filePath = "/data/spark/beauty/双十一淘宝美妆数据.csv"

// 指定 Schema
import org.apache.spark.sql.types.

// 定义字段: update time,id,title,price,sale count,comment count,店名
val fields = Seq(
       StructField("update time", StringType, true),
       StructField("id", StringType, true),
       StructField("title", StringType, true),
       StructField("price", DecimalType(10,2), true),
       StructField("sale count", IntegerType, true),
       StructField("comment count", IntegerType, true),
       StructField("shop", StringType, true)
   )

// 定义 schema
val schema = StructType(fields)

// 读取文件数据源，创建 DataFrame
val df = spark.read
     .option("header",true)
     .option("inferSchema",false)
     .option("sep",",")
     .schema(schema)
     .csv(filePath)

// 数据集大小
println("记录数量: " + df.count())

// 查看前 5 条数据
df.show(5,true)

// 查看数据集的模式(schema)
df.printSchema()

// 执行描述性统计，进行数据探索
df2.describe().show
```

执行以上代码，输出内容如下：

```
记录数量: 27598
+-----------+------------+------------------------------+------+---------
-+-------------+------+
|update_time|         id|                         title|
price|sale_count|comment_count|  shop|
+-----------+------------+------------------------------+------+---------
-+-------------+------+
| 2016/11/14|A18164178225| CHANDO/自然堂 雪域精粹纯粹...|139.00|    26719|
2704|自然堂|
| 2016/11/14|A18177105952|CHANDO/自然堂凝时鲜颜肌活乳...|194.00|     8122|
1492|自然堂|
| 2016/11/14|A18177226992|CHANDO/自然堂活泉保湿修护精...| 99.00|    12668|
589|自然堂|
| 2016/11/14|A18178033846| CHANDO/自然堂 男士劲爽控油...| 38.00|    25805|
4287|自然堂|
```

```
| 2016/11/14|A18178045259|CHANDO/自然堂雪域精粹纯粹滋...|139.00|     5196|
618|自然堂|
+-----------+------------+-----------------------------+------+---------
-+-------------+------+
only showing top 5 rows

root
 |-- update time: string (nullable = true)
 |-- id: string (nullable = true)
 |-- title: string (nullable = true)
 |-- price: decimal(10,2) (nullable = true)
 |-- sale count: integer (nullable = true)
 |-- comment count: integer (nullable = true)
 |-- shop: string (nullable = true)

+-------+-----------+------------+-------------------------------------+-
---------------+-----------------+------------------+------+
|summary|update time|         id|                              title|
price|       sale_count|    comment_count|  shop|
+-------+-----------+------------+-------------------------------------+-
---------------+-----------------+------------------+------+
|  count|     27512|      27512|                              27512|
27512|           25162|           25162| 27512|
|   mean|      null|       null|                               null|
363.423512|12316.054208727446|1121.7411970431604|  null|
| stddev|      null|       null|
null|614.8761530891369| 52412.36413498439| 5277.781581003796|  null|
|    min| 2016/11/10|A10027317366|  2 件*德国妮维雅唇膏男女润唇膏唇膜...|
1.00|              0|              0|  SKII|
|    max|  2016/11/9| A9768255247|（限量）蜜丝佛陀彩虹遮瑕笔膏熊猫眼...|
11100.00|         1923160|          202930|雪花秀|
+-------+-----------+------------+-------------------------------------+-
---------------+-----------------+------------------+------+
```

从上面的输出结果可以看出，sale_count 列和 comment_count 列有空值，update_time 列的日期值是字符串类型，并且不是标准日期格式。

1. 数据去重

首先去掉重复的数据。代码如下。

```
val df2 = df.distinct
println("去重后的记录数量: " + df2.count)
```

执行以上代码，输出内容如下：

```
去重后的记录数量: 27512
```

可以看到，原始数据量是 27598 条，去重之后数据量是 27512 条。

2. 缺失值填充

因为 sale_count、comment_count 两列均有 null 值，所以我们选择使用各自列的众数进行缺失值填充。首先找出这两个列的众数（众数指的是一组数据中出现次数最多的数值）。

```
// 找出 sale_count 列的众数 = 0
val mode_sale_count = df2
   .where(col("sale_count").isNotNull)
```

```
   .select("sale count")
   .groupBy("sale count")
   .count()
   .orderBy(col("count").desc)
   .first
   .get(0)

println("sale count 列的众数是: " + mode sale count)

// 找出 comment count 列的众数 = 0
val mode comment count = df2
   .where(col("comment count").isNotNull)
   .select("comment count")
   .groupBy("comment count")
   .count()
   .orderBy(col("count").desc)
   .first
   .get(0)

println("comment_count 列的众数是: " + mode_sale_count)
```

执行以上代码，输出内容如下：

```
sale_count 列的众数是: 0
comment_count 列的众数是: 0
```

从统计结果可知，sale_count、comment_count 两列的众数均为 0，且结合实际情况销量、评论数都可能存在为 0 的情况，因此使用 0 来填充这两列的空值。

```
// 使用众数填充 comment_count 列和 sale_count 列的空值
val df3 = df2.na.fill(Map(
    "sale_count"->mode_sale_count,
    "comment_count"->mode_comment_count
))

// 再一次执行描述性统计
df3.describe().show()
```

执行以上代码，输出内容如下：

```
+-------+-----------+------------+-------------------------------------+-
---------------+-----------------+------------------+------+
|summary|update_time|          id|                                title|
price|       sale_count|    comment_count|  shop|
+-------+-----------+------------+-------------------------------------+-
---------------+-----------------+------------------+------+
|  count|      27512|       27512|                                27512|
27512|            27512|           27512| 27512|
|   mean|       null|        null|                                 null|
363.423512|11264.050450712417|1025.9251235824368|  null|
| stddev|       null|        null|
null|614.8761530891368| 50241.93085149619| 5057.064300635684|  null|
|    min| 2016/11/10|A10027317366|  2 件*德国妮维雅唇膏男女润唇膏唇膜...|
1.00|              0|              0|  SKII|
|    max|  2016/11/9| A9768255247|（限量）蜜丝佛陀彩虹遮瑕笔膏熊猫眼...|
11100.00|         1923160|          202930|雪花秀|
+-------+-----------+------------+-------------------------------------+-
---------------+-----------------+------------------+------+
```

观察输出结果，各列均已不存在空值了。

3. 抽取商品类别

原始数据集中并没有包含商品类别信息，但是统计需求里要求“按商品类别统计...”或“按商品子类别统计...”，因此我们需要对 title 列执行分词操作，以便根据商品描述提取商品主类别和子类别。这里我们使用结巴中文分词工具。

在下面的代码中，首先自定义分词函数 cutUDF()，并在 Spark 中注册为 cutTitleUDF。然后在 title 列上应用该 UDF 函数，分词后的数组作为新增加的列 title_words 的值。实现代码如下。

```
// 1）对商品名称 title 进行分词处理
import com.huaban.analysis.jieba.JiebaSegmenter
import scala.jdk.CollectionConverters.

// 自定义分词函数
......

// 查看
df4.printSchema()
df4.select("title_words").show(5,false)
```

执行以上代码，输出内容如下：

```
root
 |-- update_time: string (nullable = true)
 |-- id: string (nullable = true)
 |-- title: string (nullable = true)
 |-- price: decimal(10,2) (nullable = true)
 |-- sale_count: integer (nullable = false)
 |-- comment_count: integer (nullable = false)
 |-- shop: string (nullable = true)
 |-- title_words: array (nullable = true)
 |    |-- element: string (containsNull = true)

+-----------------------------------------------------------------------
----------------------+
|title_words
|
+-----------------------------------------------------------------------
----------------------+
|[CHANDO, /, 自然, 堂娇颜, 粉嫩, 保湿, 眼部, 精华,  , 四件套, 乳液, 精华]
|
|[CHANDO, /, 自然, 堂水润, 保湿, 洗颜霜,  , 专柜, 正品,  , 洗面奶, 女,  , 深层, 清
洁, 不, 紧绷]|
|[CHANDO, /, 自然, 堂, 【, 预售, 】, 凝, 时鲜, 颜, 眼部, 精华, 滋养, 套装,  , 保湿,
滋润, 精华]|
|[CHANDO, /, 自然, 堂弹, 嫩, 紧致, 抗皱, 滋养, 乳液, /, 霜,  , 早安, 面霜,  , 润泽,
肌肤]     |
|[资生堂,  , 盼丽, 风姿, 抗皱, 日乳, SPF15,  , 75ml, 补水, 保湿,  , 乳液,  , 滋润]
|
+-----------------------------------------------------------------------
----------------------+
```

为了从这些描述性属性中抽取出商品所性的主类别和子类别，接下来定义一个从商品描述属性以商品类别的倒排索引。实现代码如下。

```
// 定义美妆标准分类，每一元素包含(商品主类别,商品子类别,商品描述属性)
......

baseDataMap.foreach(println)
```

执行以上代码，输出内容如下：

```
(乳液,(护肤品,乳液类))
(唇釉,(化妆品,口红类))
(散粉,(化妆品,底妆类))
(防晒喷雾,(护肤品,防晒类))
(洗面,(护肤品,清洁类))
(眉粉,(化妆品,眼部彩妆))
(菁华霜,(护肤品,面霜类))
(其他,(其他,其他))
(高光,(化妆品,修容类))
(润肤乳,(护肤品,乳液类))
(底霜,(护肤品,面霜类))
(亮肤水,(护肤品,化妆水))
(舒缓喷雾,(护肤品,化妆水))
(腮红,(化妆品,修容类))
(美白乳,(护肤品,乳液类))
(蜜粉,(化妆品,底妆类))
(润肤水,(护肤品,化妆水))
(睫毛膏,(化妆品,眼部彩妆))
(精华素,(护肤品,精华类))
(柔肤水,(护肤品,化妆水))

......

(粉饼,(化妆品,底妆类))
(柔肤液,(护肤品,化妆水))
(眼影,(化妆品,眼部彩妆))
(染眉膏,(化妆品,眼部彩妆))
(凝乳,(护肤品,乳液类))
(套装,(护肤品,套装))
(日霜,(护肤品,面霜类))
(修护霜,(护肤品,面霜类))
(亮肤乳,(护肤品,乳液类))
(晚间霜,(护肤品,面霜类))
(面霜,(护肤品,面霜类))
(精粹水,(护肤品,化妆水))
(眼膜,(护肤品,眼部护理))
(精华液,(护肤品,精华类))
(去角质,(护肤品,清洁类))
(唇彩,(化妆品,口红类))
(精华露,(护肤品,精华类))
(磨砂,(护肤品,清洁类))
(凝霜,(护肤品,面霜类))
(洗颜,(护肤品,清洁类))
(定妆粉,(化妆品,底妆类))
(面膜,(护肤品,面膜类))
(眼部精华,(护肤品,眼部护理))
(遮瑕,(化妆品,底妆类))
(粉底膏,(化妆品,底妆类))
(保湿霜,(护肤品,面霜类))
(滋润霜,(护肤品,面霜类))
(粉底霜,(化妆品,底妆类))
```

```
(口红,(化妆品,口红类))
(卸妆,(护肤品,清洁类))
(亮肤霜,(护肤品,面霜类))
```

接下来，定义并注册一个 UDF 函数 getFirstAndSecondType()，它实现的功能是根据 title_words 列的描述属性，从 baseDataMap 中检索出对应的商品主类和商品子类。

```
// 自定义函数 UDF，用于从 title_words 中提取商品的主类别和子类别
def getFirstAndSecondType(words:Array[String]): (String,String) = {
    var flag = true
    var firstAndSecondType = ("其他","其他")
    ......
    firstAndSecondType
}

// 注册 UDF
......
```

执行以上代码，输出内容如下：

```
root
 |-- update_time: string (nullable = true)
 |-- id: string (nullable = true)
 |-- title: string (nullable = true)
 |-- price: decimal(10,2) (nullable = true)
 |-- sale_count: integer (nullable = false)
 |-- comment_count: integer (nullable = false)
 |-- shop: string (nullable = true)
 |-- title_words: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- firstType: string (nullable = true)
 |-- secondType: string (nullable = true)

+-------------+-----------------------------------+---------+----------+
|           id|                              title|firstType|secondType|
+-------------+-----------------------------------+---------+----------+
|A539970643513|        CHANDO/自然堂娇颜粉嫩保湿眼...|    护肤品|    乳液类|
| A26487404578|        CHANDO/自然堂水润保湿洗颜霜...|    护肤品|    清洁类|
| A24613924461|        CHANDO/自然堂【预售】凝时鲜...|    护肤品|     套装|
|A539857501037|        CHANDO/自然堂弹嫩紧致抗皱滋...|    护肤品|    乳液类|
| A41347420374|       资生堂 盼丽风姿抗皱日乳 SPF15...|    护肤品|    乳液类|
|A526986535177|    【双 II 预售】资生堂新透白美肌光感...|      其他|      其他|
|A527621891313|     【双 II 预售】资生堂 新透白祛斑精...|      其他|      其他|
|A529008131128|       【11-11】植村秀五角海绵 粉扑...|      其他|      其他|
|A539909535729|      【11-11 套装】植村秀晶萃溢采肌...|    护肤品|     套装|
|A537380192775|       【11-11】植村秀自动砍刀眉笔 ...|      其他|      其他|
|A528472872135|       【11-11】植村秀塑颜水感亲肤粉...|    化妆品|    底妆类|
| A21199824014|           innisfree/悦诗风吟橄榄油...|    护肤品|    清洁类|
|A521773707465|           innisfree/悦诗风吟济州石...|    护肤品|  眼部护理|
|A524388759760|           innisfree/悦诗风吟乐活自...|      其他|      其他|
|A527110296708|           innisfree/悦诗风吟雪耳晶...|    护肤品|    化妆水|
| A26099952811|           innisfree/悦诗风吟济州岛...|    护肤品|    清洁类|
| A45724828592|           innisfree/悦诗风吟自然关...|    护肤品|    防晒类|
| A45152404336|           innisfree/悦诗风吟亲肤透...|    护肤品|    面膜类|
|A521788336803|           innisfree/悦诗风吟济州石...|      其他|      其他|
| A41596173473|           innisfree/悦诗风吟滋养护...|      其他|      其他|
+-------------+-----------------------------------+---------+----------+
only showing top 20 rows
```

4. 增加性别判断列

新增一列 man，用于判断是否推荐男士使用。这也是一个计算列，基于 title_words 列。如果 title_words 列中的商品描述信息中包含有“男”、“男士”、“男生”这样的词语描述，则将 man 列的值设为“是”，否则设为“否”。实现代码如下。

```
// 定义一个转换函数
def convertToGender(words: Array[String]): String = {
    ......
}

// 注册 UDF
val forMan = udf(convertToGender( : Array[String]): String)

// 应用自定义函数，计算 man 列的值
val df6 = df5.withColumn("man", forMan(col("title words")))

// 查看 Schema
df6.printSchema()

// 随机抽取部分样本，查看转换结果
df6.sample(false,0.01).select("title","man").show(20,false)
```

执行以上代码，输出内容如下：

```
root
 |-- update_time: string (nullable = true)
 |-- id: string (nullable = true)
 |-- title: string (nullable = true)
 |-- price: decimal(10,2) (nullable = true)
 |-- sale_count: integer (nullable = false)
 |-- comment_count: integer (nullable = false)
 |-- shop: string (nullable = true)
 |-- title_words: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- firstType: string (nullable = true)
 |-- secondType: string (nullable = true)
 |-- man: string (nullable = true)

+------------------------------------------------------------+---+
|title                                                       |man|
+------------------------------------------------------------+---+
|CHANDO/自然堂活泉秋冬滋养控油护肤套装 补水控油保湿 化妆品女          |否 |
|兰蔻 新清滢洁面乳洗面奶 125ml 保湿温和深入清洁 丰富泡沫不刺激       |否 |
|L'oreal 欧莱雅复颜密集修护精华液 基层提拉素科技 积雪草修护          |否 |
|L'OREAL 欧莱雅男士护肤控油周全理肤露 补水保湿祛痘印              |是 |
|新【卸妆洁面 2 合 1】妮维雅透美白深层清洁洗面奶乳温和保湿收毛孔     |否 |
|美加净银耳珍珠滋养霜 80g*3+120g*2 嫩肤霜 补水保湿面霜护肤品       |否 |
|美宝莲 晴采造型眉粉 告别染眉膏 防水防汗 不晕染不脱妆            |否 |
|Herborist/佰草集典萃水润活颜三部曲                           |否 |
|兰芝水衡清盈精华水 200ml 补水保湿 水油平衡 全新升级            |否 |
|雅诗兰黛口红 花漾倾慕丰唇膏 玻尿酸持久保湿滋养                 |否 |
|Herborist/佰草集养肤悦色隔离慕斯 50ml                        |否 |
|CHANDO/自然堂凝时休眠霜（凝时鲜颜肌活霜淡化细纹面霜             |否 |
|【11-11 预售】Shu uemura 植村秀专业睫毛夹 自然卷曲 明星款工具   |否 |
|Olay 玉兰油水漾动力莹眸走珠精华笔去淡化眼袋保湿补水眼部精华液     |否 |
|【全新上市】雅诗兰黛香水 缪斯淡香氛-风尚格调 淡香持久 女士       |否 |
|专柜正品相宜本草红石榴鲜活透亮眼凝霜淡化黑眼圈细纹女保湿眼霜       |否 |
```

```
|兰芝臻白焕颜修护面霜 50ml 美白亮肤 保湿改善暗黄  深层滋补                |否 |
|德国妮维雅男士美白洗面奶控油祛痘印保湿去黑头去油洁面乳                   |是 |
|CHANDO/自然堂自然堂 嫩白保湿乳霜礼装套装持久保湿水嫩肌肤补水             |否 |
|L'OREAL欧莱雅美眸深邃巨星眼线水笔 纯黑流畅女神同款                      |否 |
+--------------------------------------------------------------+---+
only showing top 20 rows
```

5. 日期格式化

我们注意到，update_time 列的日期值并不是标准格式，因此需要对该列执行日期格式化转换，从 yyyy/MM/dd 格式转换为 yyyy-MM-dd 格式。实现代码如下。

```
val df7 = ......

df7.printSchema()
df7.select("update_time").show(5, false)
```

执行以上代码，输出内容如下：

```
root
 |-- update_time: date (nullable = true)
 |-- id: string (nullable = true)
 |-- title: string (nullable = true)
 |-- price: decimal(10,2) (nullable = true)
 |-- sale_count: integer (nullable = false)
 |-- comment_count: integer (nullable = false)
 |-- shop: string (nullable = true)
 |-- title_words: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- firstType: string (nullable = true)
 |-- secondType: string (nullable = true)
 |-- man: string (nullable = true)

+-----------+
|update_time|
+-----------+
|2016-11-13 |
|2016-11-12 |
|2016-11-08 |
|2016-11-08 |
|2016-11-08 |
+-----------+
only showing top 5 rows
```

6. 增加销售额列

在原始数据集中，只有销售数量列（sale_count）和单价列（price），如果要计算销售额（比如，统计不同类别美妆商品的销售额），则需要增加一个销售额列，它是通过对 sale_count 和 price 执行相乘计算而来的。实现代码如下。

```
val df8 = ......

df8.printSchema()
df8.select("id","price","sale_count","sales_amount").show(5)
```

执行以上代码，输出内容如下：

```
root
 |-- update time: date (nullable = true)
 |-- id: string (nullable = true)
 |-- title: string (nullable = true)
 |-- price: decimal(10,2) (nullable = true)
 |-- sale count: integer (nullable = false)
 |-- comment count: integer (nullable = false)
 |-- shop: string (nullable = true)
 |-- title words: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- firstType: string (nullable = true)
 |-- secondType: string (nullable = true)
 |-- man: string (nullable = true)
 |-- sales amount: decimal(21,2) (nullable = true)

+-------------+------+----------+------------+
|          id| price|sale count|sales amount|
+-------------+------+----------+------------+
|A539970643513|695.00|       734|   510130.00|
| A26487404578| 38.00|    189078|  7184964.00|
| A24613924461|438.00|     14950|  6548100.00|
|A539857501037|280.00|         4|    1120.00|
| A41347420374|480.00|       320|   153600.00|
+-------------+------+----------+------------+
only showing top 5 rows
```

7. 保存预处理结果

最后，将预处理后的数据保存到 HDFS 文件中，以便进行下一步的数据分析任务。保存数据的代码如下。

```
// 写出到 parquet 文件
df8.coalesce(1).write.parquet("/data/spark/beauty")
```

第 3 章 Spark ML 特征工程

特征工程，是组合或转换现有变量以创建新特征的过程。例如，我们可以创建一个新变量，它是一些其他数据的平均值，例如用户观看电影的平均次数。

数据转换和特征提取过程中常见的问题包括：

(1) 转换分类数据，将其编码为数字表示，如国家、电影类别、商品大类等。

(2) 从文本数据中提取有用的特征。

(3) 处理图像或音频数据。

(4) 将数值数据转换为分类数据，以减少变量可以取值的数量。例如，将年龄变量转换为 bucket（如 25-35、45-55 等等）。

(5) 数值转换功能；例如，对数值变量应用对数变换可以帮助处理取值范围非常大的变量。

(6) 规范化和标准化数值特征，以确保模型的所有不同输入变量具有一致的比例。

(7) 降维。如果特征向量的维数过高，不仅计算量大，还会影响算法的精度，此时需要对数据进行降维，把向量变换到更低维的空间中之后再做处理。

3.1 特征工程概念

一旦完成了数据的清理，我们就可以着手从数据中提取实际的特征，用这些特征可以训练我们的机器学习模型。所谓特征，指的是我们用来训练模型的变量。每一行数据都包含我们想要提取到模型训练中的信息。

变量（特征）以不同的形式捕获数据信息。主要有两类被广泛使用的变量：数值变量和分类变量。如图 3-1 所示。

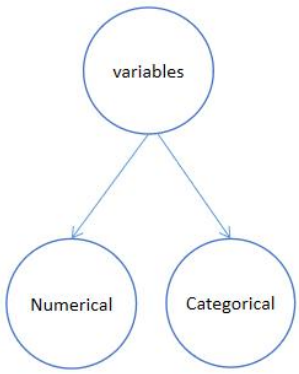


图 3-1 数据值和分类变量

数值变量是那些本质上是定量的值，比如数字（整数/浮点数）。例如，工资记录、考试成绩、年龄或身高以及股票价格都属于数值变量的范畴。数值变量也称为连续变量。

分类变量本质上是定性的，主要代表被测量数据的类别。例如，颜色、结果（是/否）、评级（好/中/差）。

为了建立机器学习模型，我们需要输入和输出变量。输入变量是那些用于构建和训练机器学习模型以预测输出或目标变量的值。

在 MLlib 中使用 ML 算法是有严格要求的：它们要求所有的特征都是 Double 数据类型，包括标签。几乎所有的机器学习模型最终都以向量的形式进行数值表示；因此，我们需要将原始数据转换为数字。

3.2 特征分类

特征大致可以分为以下几类：

(1) 数值特征：这些特征通常是实数或整数，例如，用户年龄、商品价格等。

(2) 分类特征：这些特征指的是在任何给定时间都可以采用一组可能状态中的一种的变量。例如用户的性别、职业、电影类别等。

(3) 文本特征：这些特征源自数据中的文本内容，例如电影标题、描述或评论。

(4) 其他特征：大多数其他类型的特征最终都是用数字表示的。例如，图像、视频和音频可以表示为一组数值数据。地理位置可以表示为纬度和经度或 geohash 数据（GeoHash 将二维的经纬度转换成字符串）。

下面分别来理解这几类特征。

1）数值特征

任何数值数据都可以作为输入变量。然而，在机器学习模型中，我们需要了解每个特征的权重向量。权重在将特征值映射到结果或目标变量（在监督学习模型的情况下）中发挥作用。

因此，我们希望使用有意义的特征，即模型可以学习特征值和目标变量之间的关系。例如，年龄可能是一个合理的特征。也许年龄的增长和某种结果有直接关系。例如，随着年龄的增长，收入也随之增长。

我们经常看到，数值特征在其原始形式中用处不大，但可以转换成更有用的表示形式，例如位置。使用原始位置（比如纬度和经度）可能没有那么有用，除非我们的数据非常密集，因为我们的模型可能无法了解原始位置和结果之间的有用关系。但是，位置（例如，城市或国家）的一些聚合或分箱表示与结果之间可能存在关系。

2）分类特征

分类特征不能作为原始形式的输入，因为它们通常不是数字，而是一组可能的值。例如，用户的职业，可能是程序员、学生、数据分析师、销售人员等。

为了将分类变量转换成数字表示形式，我们可以使用一种称为 1-of-k 编码的常见方法。需要一种如 1-of-k 编码的方法来表示。

假设这个变量有 k 个可能的值。如果我们给每个可能的值分配一个索引，从 0 到 k-1，那么我们可以用长度为 k 的二元向量来表示变量的给定状态；在这个向量中，除了索引处对应于给定变量状态的项被设置为 1，其余的项都设为 0。

例如，职业有"学生、程序员、数据分析师"，对应的索引分别为"0,1,2"，那么我们可以用这样的二元向量来表示各个职业：学生=>[1,0,0]；程序员=>[0,1,0]；数据分析师=>[0,0,1]。

注意： 分类变量可以进一步划分如下：

(0) Norminal Data，定类数据：也称作名义数据，是对事物的类别或属性的一种划分，按照事物的某种属性对其进行分类或分组。其不同取值仅仅代表了不同类别的事物，仅能表示类别差异，不能比较各类别之间的大小，各类别之间没有顺序或等级，这样的变量叫定类变量。

例如，人口特征中的“性别”，就是定类变量。对于定类变量，加减乘除等运算是没有实际意义的，只能用于计算频数和频率。

(1) Ordinal Data，定序数据：变量的值不仅能够代表事物的分类，还能代表事物按某种特性的排序，这样的变量叫定序变量。例如，人口特征中的“教育程度”，就是定序变量。定序变量的值之间可以比较大小，或者有强弱顺序，但两个值的差一般没有什么实际意义。

(2) Interval Data，定距数据：变量的值之间可以比较大小，两个值的差有实际意义，这样的变量叫定距变量。例如，人口特征中的“年龄”和“每月平均收入”，都是定距变量。

(3) Ratio Data，定比数据：有绝对 0 点，如质量，高度。定比变量与定距变量的差别在于，定距变量取值为 0 时，不表示“没有”，仅仅是取值为 0。定比变量取值为 0 时，则表示“没有”。

3）派生特征

正如我们前面提到的，从一个或多个可用变量计算派生特征通常很有用。我们希望派生的特征能够添加更多的信息，而不仅仅是使用原始形式的变量。从原始数据派生出特征的例子包括计算平均值、中值、方差、和、差、最大值或最小值以及计数。

通常，使用这些转换背后的思想是以某种方式汇总数值数据，从而使模型更容易学习特征。例如，给定一个订单日期列 2022-01-15 22:30:30，我们可以从中提取年、月、日、小时等值，以这些值为派生特征，统计每月订单数、每年订单数等。

4）文本特征

在某些方面，文本特征是分类特征和派生特征的一种形式。例如，在一个电影评分数据集中，假定包含对一个电影的描述。在这里，原始文本（电影描述）不能直接使用，也无法作为分类特征，因为有无限可能的组合的单词，我们的模型几乎不会看到相同特征会出现两次，所以不能有效地学习。因此，我们希望将原始文本转换成一种更适合机器学习的形式。

处理文本的方法有很多，自然语言处理领域致力于处理、表示和建模文本内容。完整的处理超出了本书的范围，但是我们将介绍一种简单和标准的文本特征提取方法，这种方法称为“词袋表示（bag-of-words）”。词袋方法将文本内容视为文本中的单词（可能还有数字，这些通常称为“术语”）的集合。

词袋方法的过程如下：

(1) 分词：首先，对文本应用某种形式的分词法，将其分割为一组标记（一般是单词、数字等）。

(2) 删除停止词：接下来，通常删除非常常见的单词，如 the、and 和 but（这些被称为停止）。

(3) 词干提取：下一步可以包括词干提取，即提取一个术语并将其还原为其基本形式或词干。一个常见的例子是复数变成单数（例如，dogs 变成 dog，等等）。词干提取有很多方法，文本处理库通常包含各种词干提取算法，例如 OpenNLP、NLTK 等。

(4) 向量化：最后一步是将处理过的项转换成向量表示形式。最简单的形式可能是二元向量表示，如果文本中存在项，则赋值为 1；如果文本中不存在项，则赋值为 0。这与我们前面遇到的分类 1-of-k 编码本质上是相同的。像 1-of-k 编码一样，这需要一个词汇字典，将给定的词汇映射到索引号。通常使用稀疏向量。

5）标准化特征

将特征提取为向量后，通常的预处理步骤是对数值数据进行标准化（也叫归一化或规范化）。其背后的思想是以一种将每个数值特征缩放到标准大小的方式进行转换。

如果特征向量各分量的取值范围相关很大，则会影响算法的精度，导致值小的特征被值大的特征淹没，在计算时也可能会导致浮点数的溢出。例如，身高特征的范围是 1.0~2.4m，体重特征的范围是 35~100kg，则身高特征可能会被体重特征所淹没。有些模型使用了指数函数、对数函数、多次乘积，过大或过小的输入数据可能会导致浮点数溢出。

处理这一问题的手段是数据归一化，常用的方案有三种：

(1) 将所有特征分量缩放到某一范围内，如 0∽1。

(2) 用方差和均值进行归一化。

(3) 将特征向量归一化到单位长度。

Spark 在其机器学习库中提供了一些内置的功能，用于特征扩展和标准化。这些包括 StandardScaler，它应用标准正态变换，以及 Normalizer，它应用相同的特征向量标准化。优先使用这些 MLlib 的内置方法，因为它们肯定比编写我们自己的函数更方便和高效!

3.2 特征转换-连续特征

连续特征类似数轴上的值，是从正无穷到负无穷的连续数值。Spark ML 提供了两种用于连续特征的 Transformer 转换器：

(1) 一种是通过一个称为“分桶”（或分箱）的过程，将连续特征转换为分类特征；

(2) 或者，根据不同的需求对特征进行缩放或规范化（归一化）。

这些 Transformer 只能作用在 Double 类型上，所以确保要转换的列值为 Double 类型。

3.2.1 连续特征离散化

在 Spark ML 中，提供了两个 Transformer 用于实现连续特征的离散化，它们分别是：

(1) Binarizer Transformer。

(2) Bucketizer Transformer。

下面分别介绍这两种 Transformer 的用法。

1. Binarizer Transformer

Binarizer Transformer 简单地将输入列的值转换为两组。第一组包含小于或等于指定阈值的值，输出列中的值将为 0。第二组包含大于指定阈值的值，输出列中的值将是 1。也就是说，Binarizer 是将数值特征阈值化为二值（0/1）特征的过程，将大于阈值的特征值二值化为 1.0，等于或小于阈值的值被二值化为 0.0。输入列必须是 Double 类型或 Vector。

下面这个示例中，Binarizer Transformer 基于给定的阈值，将连续值变量转换为两个离散的值。

```
import org.apache.spark.ml.feature.Binarizer
import scala.util.Random

// 生成一个随机数序列
val nums = Seq.fill(10)(Random.nextDouble*100)
```

```
// 创建一个 DataFrame
val numdf = spark.createDataFrame(nums.map(Tuple1.apply)).toDF("raw nums")

// 应用 Binarizer Transformer 进行输入列转换
val binarizer = new Binarizer()        // 构造实例对象
    .setInputCol("raw nums")           // 设置输入列
    .setOutputCol("binary vals")       // 设置输出列
    .setThreshold(50.0)                // 设置阈值

binarizer.transform(numdf).show()
```

执行以上代码，输出结果如下：

```
+------------------+-----------+
|          raw_nums|binary_vals|
+------------------+-----------+
| 49.76634665589662|        0.0|
| 63.31988173337968|        1.0|
| 74.37634020301995|        1.0|
| 86.29882090510866|        1.0|
|55.395809640280966|        1.0|
|  80.2439323330318|        1.0|
| 80.27148620271457|        1.0|
| 11.41027545273987|        0.0|
| 47.46819061651881|        0.0|
| 41.82512660087531|        0.0|
+------------------+-----------+
```

下面的示例代码中，使用 Binarizer Transformer 将 temperature 列值转换到两个桶。

```
// 使用 Binarizer transformer 将温度值转换到两个桶
import org.apache.spark.ml.feature.Binarizer

// 构造原始数据集(DataFrame)
val arrival_data = spark.createDataFrame(
    Seq(("北京大兴国际机场", "B737", 18, 95.1, "late"),
        ("广州白云国际机场", "A319", 5, 65.7, "ontime"),
        ("贵阳龙洞堡国际机场", "B747", 15, 31.5, "late"),
        ("青岛流亭国际机场", "A319", 14, 40.5, "late")
    )).toDF("origin", "model", "hour", "temperature", "arrival")


// Binarizer Transformer
val binarizer = new Binarizer()        // 构造实例对象
    .setInputCol("temperature")        // 设置输入列
    .setOutputCol("freezing")          // 设置输出列
    .setThreshold(35.6)                // 设置阈值

binarizer.transform(arrival_data).show
```

执行以上代码，输出结果如下：

```
+------------------+-----+----+-----------+-------+--------+
|            origin|model|hour|temperature|arrival|freezing|
+------------------+-----+----+-----------+-------+--------+
|      北京大兴国际机场| B737|  18|       95.1|   late|     1.0|
|      广州白云国际机场| A319|   5|       65.7| ontime|     1.0|
|    贵阳龙洞堡国际机场| B747|  15|       31.5|   late|     0.0|
|      青岛流亭国际机场| A319|  14|       40.5|   late|     1.0|
```

```
+------------------+-----+----+----------+-------+--------+
```

注意观察输出结果中的 freezing 列，会发现已经转换为二元变量值，凡是 temperature 列低于 35.6 的值都被转换为 0.0，而高于 35.6 的值都被转换为 1.0。为了看得更清楚，可以以输出时只选择输入列和输出列，代码如下。

```
binarizer.transform(arrival_data)
        .select("temperature", "freezing")
        .show
```

执行以上代码，输出结果如下：

```
+-----------+--------+
|temperature|freezing|
+-----------+--------+
|       95.1|     1.0|
|       65.7|     1.0|
|       31.5|     0.0|
|       40.5|     1.0|
+-----------+--------+
```

可使用 Binarizer 中的 explainParams 方法查看各个方法和参数的说明。例如，查看前面的 binarizer 这个 Transformer 的方法和参数解释，代码如下。

```
binarizer.explainParams
```

执行以上代码，输出结果如下：

```
res7: String =
inputCol: input column name (current: temperature)
inputCols: input column names (undefined)
outputCol: output column name (default: binarizer_10fad3434ebe__output,
current: freezing)
outputCols: output column names (undefined)
threshold: threshold used to binarize continuous features (default: 0.0,
current: 35.6)
thresholds: Array of threshold used to binarize continuous features. This is
for multiple columns input. If transforming multiple columns and thresholds
is not set, but threshold is set, then threshold will be applied across all
columns. (undefined)
```

从输出结果可以看出，除了指定单个输入列和单个输出列外，还可以同时指定多个输入列和多个输出列、阈值。例如，同时转换 hour 和 temperature 两列，其中 hour 的阈值设为 12，temperature 的阈值设为 35.6。代码如下。

```
// 因为所有特征都必须是 double 类型，所以将 hour 列值改为 double
val arrival_data2 = arrival_data.withColumn("hour", $"hour"*1.0)

// Binarizer Transformer
val binarizer2 = new Binarizer()
    .setInputCols(Array("hour","temperature"))
    .setOutputCols(Array("meridiem","freezing"))
    .setThresholds(Array(12.0,35.6))

binarizer2
    .transform(arrival_data2)
    .show
```

执行以上代码，输出结果如下：

```
+------------------+-----+----+-----------+-------+--------+--------+
|            origin|model|hour|temperature|arrival|meridiem|freezing|
+------------------+-----+----+-----------+-------+--------+--------+
|      北京大兴国际机场| B737|18.0|       95.1|   late|     1.0|     1.0|
|      广州白云国际机场| A319| 5.0|       65.7|ontime|     0.0|     1.0|
|    贵阳龙洞堡国际机场| B747|15.0|       31.5|   late|     1.0|     0.0|
|      青岛流亭国际机场| A319|14.0|       40.5|   late|     1.0|     1.0|
+------------------+-----+----+-----------+-------+--------+--------+
```

2. Bucketizer Transformer

Bucketizer Transformer 是 Binarizer 的通用版本，它可以将列值转换成我们所选择的桶。通过指定一个 double 值数组的桶边界列表，它可以控制桶的数量以及每个桶的取值范围。例如，有一个列，其中包含每个人的收入，每个人都生活在一个特定的地区，现在把他们的收入分成以下几个部分：高收入、中等收入、低收入，等等。

该桶的值边界数组必须是 double 类型，并且他们必须遵守以下要求：

(1) 最小的桶边值必须小于 DataFrame 中输入列中的最小值。

(2) 最大的桶边值必须大于 DataFrame 中输入列的最大值。

(3) 在输入数组中必须至少有三个桶边界，这会创建两个桶。

在上面的一个人的收入情况下，很容易知道最小的收入是 0；那么最小的桶边值就可以是小于 0 的数。如果不可能预测最小列值，那么指定负无穷。类似地，如果不可能预测最大列值，那么指定正无穷大。

下面的示例代码使用 Bucketizer Transformer 将连续值变量转换为所需的一组离散的值。

```
import org.apache.spark.ml.feature.Bucketizer

// 定义桶边界列表
val split_vals:Array[Double] = Array(0,20,50,80,100)

// 创建 Bucketizer 转换器实例，并指定参数
val bucketizer = new Bucketizer()
    .setInputCol("raw_nums")
    .setOutputCol("binned_nums")
    .setSplits(split_vals)

// 转换
bucketizer.transform(numdf).show()
```

执行以上代码，输出结果如下：

```
+------------------+-----------+
|          raw nums|binned nums|
+------------------+-----------+
| 49.76634665589662|        1.0|
| 63.31988173337968|        2.0|
| 74.37634020301995|        2.0|
| 86.29882090510866|        3.0|
|55.395809640280966|        2.0|
|  80.2439323330318|        3.0|
| 80.27148620271457|        3.0|
| 11.41027545273987|        0.0|
```

```
| 47.46819061651881|       1.0|
| 41.82512660087531|       1.0|
+------------------+----------+
```

在只给定两个间隔的桶边界情况下，则 Bucketizer 实际上等价于 Binarizer。请看下面这个示例代码。

```
new Bucketizer()
    .setInputCol("raw_nums")
    .setOutputCol("binned_nums")
    .setSplits(Array(0,50.0,100.0))  // 指定两个桶
    .transform(numdf)
    .show()
```

执行以上代码，输出结果如下：

```
+------------------+----------+
|          raw_nums|binned_nums|
+------------------+----------+
| 49.76634665589662|       0.0|
| 63.31988173337968|       1.0|
| 74.37634020301995|       1.0|
| 86.29882090510866|       1.0|
|55.395809640280966|       1.0|
|  80.2439323330318|       1.0|
| 80.27148620271457|       1.0|
| 11.41027545273987|       0.0|
| 47.46819061651881|       0.0|
| 41.82512660087531|       0.0|
+------------------+----------+
```

下面的代码示例使用 Bucketizer 转换器将温度列放入三个桶中，这意味着 bucket 边界数组必须包含至少 4 个值。输出按温度列排序，使其更容易查看。

```
// 使用 Bucketizer transformer 将温度值转换到三个桶
import org.apache.spark.ml.feature.Bucketizer

// 指定桶范围列表
val bucketBorders = Array(-1.0, 32.0, 70.0, 150.0)

// 构造 Bucketizer 实例，并指定输入列、输出列和桶范围
val bucketer = new Bucketizer()
    .setInputCol("temperature")
    .setOutputCol("intensity")
    .setSplits(bucketBorders)

// 执行转换
val output = bucketer.transform(arrival_data)
output.show
```

执行以上代码，输出结果如下：

```
+------------------+-----+----+-----------+-------+---------+
|            origin|model|hour|temperature|arrival|intensity|
+------------------+-----+----+-----------+-------+---------+
|    北京大兴国际机场| B737|  18|       95.1|   late|      2.0|
|    广州白云国际机场| A319|   5|       65.7| ontime|      1.0|
|  贵阳龙洞堡国际机场| B747|  15|       31.5|   late|      0.0|
|    青岛流亭国际机场| A319|  14|       40.5|   late|      1.0|
```

```
+------------------+-----+----+----------+-------+---------+
```

3.2.2 连续特征标准化

当不同特征之间的值差距较大，分布很离散时，那么可能就需要统一这些数据的量纲，以便后期的处理，这就需要对数据进行标准化处理。

数据标准化（归一化）处理是数据挖掘的一项基础工作，不同评价指标往往具有不同的量纲和量纲单位，这样的情况会影响到数据分析的结果，为了消除指标之间的量纲影响，需要进行数据标准化处理，以解决数据指标之间的可比性。原始数据经过数据标准化处理后，各指标处于同一数量级，适合进行综合对比评价。

在 Spark 机器学习库中，提代的数据标准化处理的主要方法如下：

(1) StandardScaler。

(2) MinMaxScaler。

(3) MaxAbsScaler。

(4) Normalizer。

下面分别介绍这些数据标准化处理方法。

1. StandardScaler

StandardScaler 将一组特征标准化，使其均值为 0，标准差为 1。StandardScaler 处理的对象是每一列，也就是每一维特征，将特征标准化为单位标准差（标准差为 1）或是 0 均值。它主要有两个参数可以设置：

(1) withSt：默认为 true。将数据标准化（缩放）到单位标准差。

(2) withMean：默认为 false。是否变换为 0 均值，即在缩放之前是否对数据分布居中处理。如果设为 true，则在缩放前对数据分布居中处理 (此种方法将产出一个密集输出，所以不适用于稀疏输入)。

StandardScaler 是一种 Estimator，它可以 fit() 一个数据集来生成一个 StandardScalerModel，这相当于计算汇总统计数据。然后，该模型可以转换数据集中的向量列，使其具有单位标准差和/或零均值特征。如果一个特征的标准差为零，它将返回该特征在向量中的默认 0.0 值。

注意：

这种方法根据原始数据的均值（mean）和标准差（standard deviation）进行数据的标准化，通常称为 z-score 标准化，也称为中心标准化。z-score 标准化方法适用于属性的最大值和最小值未知的情况,或有超出取值范围的离群数据的情况。经过处理的数据符合标准正态分布，即均值为 0，标准差为 1。转换函数如下：

$$x^* = \frac{x - \mu}{\sigma}$$

其中 μ 为所有样本数据的均值，σ 为所有样本数据的标准差，计算时对每个属性/每列分别进行。得到的结果是，对于每个属性/每列来说所有数据都聚集在 0 附近，方差为 1。

图 3-2 是一个散点序列的标准化过程：原图->减去均值->除以标准差。这种将特征的值域重新缩放到 0-1 之间的技巧对于优化算法是很有用的，诸如在回归和神经网问题中应用到的“梯度下降”。缩放也适用于基于距离测量的算法，如 K 近邻算法（KNN）。

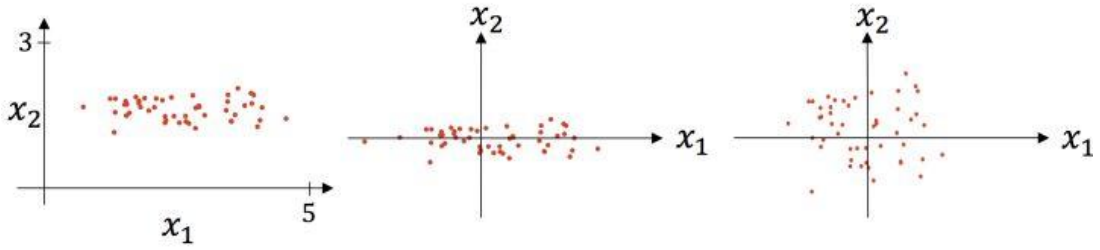


图 3-2 一个散点序列的标准化过程

下面的示例使用 StandardScaler Estimator 对数据集的特征进行标准放缩放处理，代码如下。

```
// 使用 StandardScaler Estimator 标准化围绕均值 0 的特征
import org.apache.spark.ml.feature.StandardScaler
import org.apache.spark.ml.linalg.Vectors

// 构造 DataFrame
val employee_data = Seq(
    (1, Vectors.dense(1.0, -1.0,  2.0)),
    (1, Vectors.dense(2.0,  0.0,  0.0)),
    (1, Vectors.dense(0.0,  1.0, -1.0)),
    (1, Vectors.dense(3.0, -5.0,  6.0))
 ).toDF("empId", "features")

// 将单位标准差设置为 true 并围绕平均值
val standardScaler = new StandardScaler()
    .setWithStd(true)                    // 缩放到单位标准差
    .setWithMean(true)                   // 缩放前对数据分布居中处理
    .setInputCol("features")
    .setOutputCol("scaledFeatures")

// 拟合数据，建立模型 - 返回一个 transformer
val standardMode = standardScaler.fit(employee_data)

// 使用学习到的模型对数据集进行转换
val standardData = standardMode.transform(employee_data)

// 显示结果:标准化后的数据，均值为 0，方差为 1
standardData.show(false)
```

执行以上代码，输出结果如下：

```
+-----+--------------+------------------------------------------------------------
----------+
|empId|features      |scaledFeatures
|
+-----+--------------+------------------------------------------------------------
----------+
|1
|[1.0,-1.0,2.0]|[-0.3872983346207417,0.09505863757867168,0.08075728530872
482]|
```

```
|1    |[2.0,0.0,0.0]
|[0.3872983346207417,0.4752931878933584,-0.5653009971610737]  |
|1
|[0.0,1.0,-1.0]|[-1.161895003862225,0.8555277382080452,-0.888330138395973
1]  |
|1
|[3.0,-5.0,6.0]|[1.161895003862225,-1.4258795636800752,1.372873850248322]
|
+-----+--------------+-----------------------------------------------------
----------+
```

下面的例子使用一个简单的员工数据集。对该数据集的特征列进行标准化转换，转换后这些特征的值以 0 为中心，有一个单位的标准差。

```
   // 使用 StandardScaler estimator 标准化围绕均值 0 的特征
import org.apache.spark.ml.feature.StandardScaler
import org.apache.spark.ml.linalg.Vectors

// 构造 DataFrame
val employee_data = spark.createDataFrame(
    Seq(
        (1, Vectors.dense(125400, 5.3)),
        (2, Vectors.dense(179100, 6.9)),
        (3, Vectors.dense(154770, 5.2)),
        (4, Vectors.dense(199650, 4.11))
    )).toDF("empId", "features")

// 将单位标准偏差设置为 true 并围绕平均值
val standardScaler = new StandardScaler()
    .setWithStd(true)
    .setWithMean(true)
    .setInputCol("features")
    .setOutputCol("scaledFeatures")

// 拟合数据，建立模型
val standardMode = standardScaler.fit(employee_data)

// 使用学习到的模型对数据集进行转换
val standardData = standardMode.transform(employee_data)

// 显示结果
standardData.show(false)
```

执行以上代码，输出结果如下：

```
+-----+---------------+-----------------------------------------+
|empId| features      |     scaledFeatures                      |
+-----+---------------+-----------------------------------------+
|1    |[125400.0,5.3] | [-1.2290717420781212,-0.06743742573177587]|
|2    |[179100.0,6.9] | [0.4490658767775897,1.3248191055048935]   |
|3    |[154770.0,5.2] | [-0.3112523404805006,-0.15445345893406737]|
|4    |[199650.0,4.11]|  [1.091258205781032,-1.102928220839048]    |
+-----+---------------+-----------------------------------------+
```

2. MinMaxScaler

另一种标准化方式是，将特征值缩放到一个指定的最大和最小值之间，通常是[0-1]之间。这样，每个特征的最大绝对值为 1。为此，Spark 提供了 MinMaxScaler 或者 MaxAbsScaler 来实现这个标准化转换。

MinMaxScaler 同样是作用在每一列，即每一维特征。将每一维特征线性地映射到指定的区间，通常是[0, 1]。它需要参数:

(1) min：默认 0.0。转换后的区间的下限，被所有特征共享。

(2) max：默认 1.0。转换后的区间的上限，被所有特征共享。

MinMaxScaler 计算数据集的汇总统计信息，并生成 MinMaxScalerModel。然后，模型可以对每个特征进行单独的转换，使其位于给定的范围内。

注意：

MinMaxScaler 也称为离差标准化，是对原始数据的线性变换，使结果值映射到[0 - 1]之间。转换函数如下：

$$x^* = \frac{x - \min}{\max - \min}$$

其中 max 为样本数据的最大值，min 为样本数据的最小值。这种方法有个缺陷就是当有新数据加入时，可能导致 max 和 min 的变化，需要重新定义。

使用这种方法的目的如下：

(1) 对于方差非常小的属性可以增强其稳定性。

(2) 维持稀疏矩阵中为 0 的项。

此变换通常用作零均值、单位方差缩放的替代方案。相比于中心标准化，离差标准化后的标准差比较小。在使用离差标准化后，数据的数值更加接近平均值。

但是如果特征列中含有异常值，离差标准化只能将所有特征统一比例，并不能很好地解决异常值问题。中心标准化在异常值方面则有更好的表现，因此它比离差标准化应用更广。

下面的示例使用 MinMaxScaler Estimator 对一个简单数据集的特征进行标准缩放处理。实现代码如下。

```
// 下面的示例演示如何将每个特征重新缩放到[0,1]
import org.apache.spark.ml.feature.MinMaxScaler
import org.apache.spark.ml.linalg.Vectors

val dataFrame = spark.createDataFrame(Seq(
     (0, Vectors.dense(1.0, 0.1, -1.0)),
     (1, Vectors.dense(2.0, 1.1, 1.0)),
     (2, Vectors.dense(3.0, 10.1, 3.0))
   )).toDF("id", "features")

val scaler = new MinMaxScaler()
   .setInputCol("features")
   .setOutputCol("scaledFeatures")

// 计算汇总统计信息并生成 MinMaxScalerModel
val scalerModel = scaler.fit(dataFrame)

// 将每个特征缩放到范围[min, max]
```

```
// 每维特征线性地映射，最小值映射到 0，最大值映射到 1
val scaledData = scalerModel.transform(dataFrame)

println(s"将每个特征缩放到范围：[${scaler.getMin}, ${scaler.getMax}]")
scaledData.select("features", "scaledFeatures").show()
```

执行以上代码，输出结果如下：

```
将每个特征缩放到范围：[0.0, 1.0]
+--------------+--------------+
|      features|scaledFeatures|
+--------------+--------------+
|[1.0,0.1,-1.0]|     (3,[],[])|
| [2.0,1.1,1.0]| [0.5,0.1,0.5]|
|[3.0,10.1,3.0]| [1.0,1.0,1.0]|
+--------------+--------------+
```

下面这个示例使用带有工资和身高信息的 employee_data 数据集，并将这两个特征转换到[0-5]范围之内。这两个特征的值之间的量级差别是相当大的，但是在通过 MinMaxScaler 进行特征转换之后，它们的量纲就统一了。

```
// 使用 MinMaxScaler estimator 来重新调节特征
import org.apache.spark.ml.feature.MinMaxScaler
import org.apache.spark.ml.linalg.Vectors

// 构造 DataFrame
val employee_data = spark.createDataFrame(
    Seq(
        (1, Vectors.dense(125400, 5.3)),
        (2, Vectors.dense(179100, 6.9)),
        (3, Vectors.dense(154770, 5.2)),
        (4, Vectors.dense(199650, 4.11))
    )).toDF("empId", "features")

// MinMaxScaler Estimator
val minMaxScaler = new MinMaxScaler()
    .setMin(0.0)
    .setMax(5.0)
    .setInputCol("features")
    .setOutputCol("scaledFeatures")

// 拟合数据，建立模型
val scalerModel = minMaxScaler.fit(employee_data)

// 使用学习到的模型对数据集进行转换
val scaledData = scalerModel.transform(employee_data)

// 输出特征缩放到的范围
println(s"特征缩放到：[${minMaxScaler.getMin},${minMaxScaler.getMax}]")

// 显示结果
scaledData.select("features", "scaledFeatures").show(false)
```

执行以上代码，输出结果如下：

```
特征缩放到：[0.0,5.0]
+---------------+----------------------------------------+
| features      |    scaledFeatures                      |
```

```
+---------------+--------------------------------------+
|[125400.0,5.3] |   [0.0,2.1326164874551963]            |
|[179100.0,6.9] |   [3.616161616161616,5.0]             |
|[154770.0,5.2] |[1.9777777777777776,1.9534050179211468]|
|[199650.0,4.11]|     [5.0,0.0]                        |
+---------------+--------------------------------------+
```

3. MaxAbsScaler

MaxAbsScaler 转换一个数据集的向量行，通过除以每个特征维的最大绝对值缩放每个特征，将每个特征重新调整到范围[- 1,1]。它不会移动数据到中心位置，因此不会破坏任何稀疏性。MaxAbsScaler 是专门处理稀疏数据而设计的。

MaxAbsScaler 计算数据集的汇总统计信息，并生成 MaxAbsScalerModel。然后，该模型可以将每个特征单独转换为闭区间[- 1,1]。下面的例子演示了如何将每个特征行重新缩放到[- 1,1]。

```
import org.apache.spark.ml.feature.MaxAbsScaler
import org.apache.spark.ml.linalg.Vectors

val dataFrame = spark.createDataFrame(
    Seq(
      (0, Vectors.dense(1.0, 0.1, -8.0)),
      (1, Vectors.dense(2.0, 1.0, -4.0)),
      (2, Vectors.dense(4.0, 10.0, 8.0))
    )).toDF("id", "features")

val scaler = new MaxAbsScaler()
    .setInputCol("features")
    .setOutputCol("scaledFeatures")

// 计算汇总统计信息并生成 MaxAbsScalerModel
val scalerModel = scaler.fit(dataFrame)

// 将每个特征缩放到范围[-1, 1]
val scaledData = scalerModel.transform(dataFrame)
scaledData.select("features", "scaledFeatures").show(false)
```

执行以上代码，输出结果如下：

```
+--------------+-------------------------------+
| features     |   scaledFeatures              |
+--------------+-------------------------------+
|[1.0,0.1,-8.0]|[0.25,0.010000000000000002,-1.0]|
|[2.0,1.0,-4.0]|   [0.5,0.1,-0.5]              |
|[4.0,10.0,8.0]|   [1.0,1.0,1.0]               |
+--------------+-------------------------------+
```

4. Normalizer

Normalizer（正态分布化）是一个 Transformer，它转换一个向量行数据集，将每个向量归一化为具有单位范数：即 Normalizer 的作用范围是每一行，使每一个行向量的范数变换为一个单位范数。它接受参数 p（默认 p = 2），该参数指定用于标准化的 p-norm。

注意：

Normalization 又叫正则化，用来将各个样本归一化为范数为 1 的正态分布。这种标准化可以帮助我们标准化输入数据，并改进学习算法的行为，该方法主要应用于文本分类和内容聚类中。例如，对于两个 TF-IDF 向量的 L2 范数进行点积，就可以得到这两个向量的余弦相似性。

正则化的过程是将每个样本缩放到单位范数（每个样本的范数为 1），如果后面要使用如二次型（点积）或者其它核方法计算两个样本之间的相似性这个方法会很有用。Normalization 主要思想是对每个样本计算其 p-范数，然后对该样本中每个元素除以该范数，这样处理的结果是使得每个处理后样本的 p-范数（L1-norm,L2-norm）等于 1。

(1) p = 1 指的是曼哈顿范数(或曼哈顿距离)，p = 2 指的是欧几里得范数。

(2) L1 范数是指向量中各个元素绝对值之和。

(3) L2 范数是指向量各元素的平方和然后求平方根。

(4) L1 范数可以进行特征选择，即让特征的系数变为 0。

(5) L2 范数可以防止过拟合，提升模型的泛化能力。L2 对大数，对 outlier 离群点更敏感。

(6) L1 会趋向于产生少量的特征，而其他的特征都是 0，而 L2 会选择更多的特征，这些特征都会接近于 0。

(7) L1 主要是用来特征选择，能够将含有信息量小的特征权重优化为 0，从而降低特征的维度，从而简化模型。

(8) L2 偏向于将特征的权重都调整的比较小，分布相对比较均匀，而不是将特征权重调整为 0，这样也可以简化模型。L2 范数不但可以防止过拟合，还可以让我们的优化求解变得稳定和快速。

总得来说，L1 可以让一部分特征的系数缩小到 0，从而间接实现特征选择。所以 L1 适用于特征之间有关联的情况。L2 让所有特征的系数都缩小，但是不会减为 0，它会使优化求解稳定快速。所以 L2 适用于特征之间没有关联的情况。

下面的例子演示了如何将每一行规范化为单位 L1 范数和单位 L∞范数。

```
import org.apache.spark.ml.feature.Normalizer
import org.apache.spark.ml.linalg.Vectors

val dataFrame = Seq(
       (0, Vectors.dense(1.0, 0.5, -1.0)),
       (1, Vectors.dense(2.0, 1.0, 1.0)),
       (2, Vectors.dense(4.0, 10.0, 2.0))
   ).toDF("id", "features")

// 使用 L1 范数对每个向量进行标准化(即所有值绝对值之和为 1)
val normalizer = new Normalizer()
   .setInputCol("features")
   .setOutputCol("normFeatures")
   .setP(1.0)

val p1Data = normalizer.transform(dataFrame)

println("用 L1 范数标准化:")
p1Data.show()

// 使用最大范数对每个向量进行标准化(即向量中所有值中的最大值调整为 1)
```

```
val maxData = normalizer.transform(dataFrame, normalizer.p ->
Double.PositiveInfinity)
println("用 L^inf 范数标准化:")

maxData.show()
```

执行以上代码，输出结果如下：

```
用 L1 范数标准化:
+---+--------------+------------------+
| id|      features|      normFeatures|
+---+--------------+------------------+
|  0|[1.0,0.5,-1.0]|    [0.4,0.2,-0.4]|
|  1| [2.0,1.0,1.0]|   [0.5,0.25,0.25]|
|  2|[4.0,10.0,2.0]|[0.25,0.625,0.125]|
+---+--------------+------------------+

用 L^inf 范数标准化:
+---+--------------+--------------+
| id|      features|  normFeatures|
+---+--------------+--------------+
|  0|[1.0,0.5,-1.0]|[1.0,0.5,-1.0]|
|  1| [2.0,1.0,1.0]| [1.0,0.5,0.5]|
|  2|[4.0,10.0,2.0]| [0.4,1.0,0.2]|
+---+--------------+--------------+
```

5. ElementwiseProduct

ElementwiseProduct 使用元素乘法将每个输入向量乘以一个提供的“权重”向量。

换句话说，它通过标量乘法器缩放数据集的每一列。这表示输入向量 v 和转换向量 w 之间的 Hadamard 乘积，从而得到一个结果向量。ElementwiseProduct 允许我们用任意值缩放向量中的每个值。

```
import org.apache.spark.ml.feature.ElementwiseProduct
import org.apache.spark.ml.linalg.Vectors

// 创建带有特征向量的 DataFrame
val dataFrame = spark.createDataFrame(
    Seq(
        ("a", Vectors.dense(1.0, 2.0, 3.0)),
        ("b", Vectors.dense(4.0, 5.0, 6.0)))
    ).toDF("id", "vector")

// 定义一个权重向量
val transformingVector = Vectors.dense(0.0, 1.0, 2.0)

// 执行转换
val transformer = new ElementwiseProduct()
  .setScalingVec(transformingVector)
  .setInputCol("vector")
  .setOutputCol("transformedVector")

// 批量变换矢量创建新列
transformer.transform(dataFrame).show()
```

执行以上代码，输出结果如下：

```
+---+-------------+-----------------+
| id|       vector|transformedVector|
+---+-------------+-----------------+
|  a|[1.0,2.0,3.0]|    [0.0,2.0,6.0]|
|  b|[4.0,5.0,6.0]|   [0.0,5.0,12.0]|
+---+-------------+-----------------+
```

6. 几种标准化方法应用场景

使用这些Estimators的原因是为了确保使用距离作为度量的学习算法不会因具有较大值的特征更施加更大的权重（比另一个具有较小值的特征相比）。有使用过程中，有一些实践经验可供借鉴：

(1) 在分类、聚类算法中，需要使用距离来度量相似性的时候、或者使用 PCA 技术进行降维的时候，StandardScaler 表现更好（丢失权重信息，保留距离信息）。

(2) 在不涉及距离度量、协方差计算、数据不符合正态分布的时候，可以使用 MinMaxScaler（丢失距离信息，只留权重信息）。比如图像处理中，将 RGB 图像转换为灰度图像后将其值限定在[0 255]的范围。

3. 3 特征转换-分类特征

分类特征最常见的任务是索引。索引将列中的分类变量转换为可以插入到机器学习算法中的数字变量。一般来说，为了保持一致性，建议在预处理时重新索引每个分类变量，这有助于长期维护使用的模型。

在 Spark 机器学习库中，建立索引最简单的方法是通过 StringIndexer，它是一个 Estimator，用于将字符串映射到不同的数字 ID。Spark 的 StringIndexer 还创建附加到 DataFrame 的元数据，该元数据指定什么输入对应什么输出。这样带来的便利之处在于，我们以后可以反过来从它们各自的索引值得到对应的输入值。

3. 3. 1 StringIndexer

StringIndexer Estimator 将一个分类值编码成一个基于其频率的索引，这样最频繁的分类值就会得到 0 的索引值，以此类推。为了让这个 Estimator 为分类值提供一个索引值，它首先必须计算每个值的频率，最后分配一个索引值；换句话说，它必须查看 DataFrame 中输入列的所有值。如果该输入列是数值，这个 Estimator 在计算它的频率之前会将它转换为字符串类型。

下面的示例中提供了一个使用 StringIndexer Estimator 来对电影类型进行编码的示例。

```
// 使用 StringIndexer estimator 来对电影类型进行编码
import org.apache.spark.ml.feature.StringIndexer

// 构造一个 DataFrame
val movie data = spark.createDataFrame(
   Seq(
       (1, "Comedy"),
       (2, "Action"),
       (3, "Comedy"),
       (4, "Horror"),
       (5, "Action"),
```

```
        (6, "Comedy")
    )).toDF("id", "genre")

// StringIndexer EEstimator
val movieIndexer = new StringIndexer()
    .setInputCol("genre")
    .setOutputCol("genreIdx")

// 首先拟合数据
val movieIndexModel = movieIndexer.fit(movie data)

// 使用返回的 transformer 来转换该数据
val indexedMovie = movieIndexModel.transform(movie data)

// 查看结果
indexedMovie.orderBy("genreIdx").show()
```

执行以上代码，输出结果如下：

```
+---+------+--------+
| id| genre|genreIdx|
+---+------+--------+
|  6|Comedy|     0.0|
|  3|Comedy|     0.0|
|  1|Comedy|     0.0|
|  2|Action|     1.0|
|  5|Action|     1.0|
|  4|Horror|     2.0|
+---+------+--------+
```

如前所述，StringIndexer Estimator 根据频率的降序分配索引。这种默认行为可以很容易地更改为频率的升序；事实上，它支持另外两种排序类型：降序字母表和升序字母表。为了改变默认的排序类型，只需简单调用 setStringOrderType("<ordering type>")函数，它具有下列值之一：frequencyDesc、frequencyAsc、alphabetDesc、alphabetAsc。例如，修改上面代码中创建 StringIndexer 对象实例的方法如下。

```
...
// StringIndexer EEstimator
val movieIndexer = new StringIndexer()
    .setInputCol("genre")
    .setOutputCol("genreIdx")
    .setStringOrderType("alphabetAsc")

...
```

执行以上代码，输出结果如下：

```
+---+------+--------+
| id| genre|genreIdx|
+---+------+--------+
|  1|Comedy|     2.0|
|  2|Action|     1.0|
|  3|Comedy|     2.0|
|  4|Horror|     0.0|
|  5|Action|     1.0|
|  6|Comedy|     2.0|
+---+------+--------+
```

也可以对非字符串的列应用 StringIndexer，在这种情况下，它们会先被转换为字符串，然后被索引。

因为 StringIndexer 是一个 Estimator，因此它必须拟合输入数据（调用 fit()方法）。这意味着它必须查看所有输入，以选择输入到 IDs 的映射。如果我们在输入“a”、“b”和“c”上训练 StringIndexer，然后在输入“d”上使用它，默认情况下它会抛出一个错误。可以在 StringIndexer 或 StringIndexerModel 上调用 setHandleInvalid()方法，通过传入的参数来指示如何处理无效的数据（包括训练时未曾见过的数据或 NULL 值）。传入的参数可以是"skip"（如果输入无效数据，则跳过整个行）、"error"（抛出一个错误，默认值）、"keep"（将无效数据放在一个特殊的附加桶中，索引为 numLabels）。代码如下：

```
movieIndexer.setHandleInvalid("skip")
movieIndexer.fit(simpleDF).setHandleInvalid("skip")
```

3. 3. 2 IndexToString

在检查机器学习结果时，可能想要将索引映射回原始值。由于 MLlib 分类模型使用索引值进行预测，这种转换对于将模型预测(索引)转换回原始类别非常有用。我们可以用 IndexToString 来实现这个转换。

与 StringIndexer 对称，IndexToString 将一列标签索引映射回包含原始标签作为字符串的列。一个常见的应用场景是使用 StringIndexer 从标签生成索引，使用这些索引训练模型，并使用 IndexToString 从预测索引的列检索原始标签。

下面这个示例中，先使用 StringIndexer Estimator 来对电影类型进行编码，然后再使用 IndexToString 将索引标签转换回原始的字符串分类。请注意，IndexToString 是一个 Transformer。

```
import org.apache.spark.ml.feature.{StringIndexer,IndexToString}

val movie data = Seq(
       (1, "Comedy"),
       (2, "Action"),
       (3, "Comedy"),
       (4, "Horror"),
       (5, "Action"),
       (6, "Comedy")
   ).toDF("id", "genre")

// movie data.show

val movieIndexer = new StringIndexer()
    .setInputCol("genre")       // 输入列
    .setOutputCol("genreIdx")  // 输出列

// 首先拟合数据，得到模型
val movieIndexModel = movieIndexer.fit(movie_data)

// 使用返回的 transformer 来转换该数据
val indexedMovie = movieIndexModel.transform(movie_data)

// 显示结果
// indexedMovie.orderBy("genreIdx").show()
```

```
// 将索引标签转换回原始的字符串分类(originalGenre 列)
var idx2str = new IndexToString()
    .setInputCol("genreIdx")         // 输入索引列
    .setOutputCol("originalGenre")   // 输出字符串列
val origin movie data = idx2str.transform(indexedMovie)

origin_movie_data.show
```

执行以上代码，输出结果如下：

```
+---+------+--------+-------------+
| id| genre|genreIdx|originalGenre|
+---+------+--------+-------------+
|  1|Comedy|     0.0|       Comedy|
|  2|Action|     1.0|       Action|
|  3|Comedy|     0.0|       Comedy|
|  4|Horror|     2.0|       Horror|
|  5|Action|     1.0|       Action|
|  6|Comedy|     0.0|       Comedy|
+---+------+--------+-------------+
```

3.3.3 VectorIndexer

VectorIndexer 是一个有用的工具，用于在 Vector（向量）数据集中索引分类特征列。默认情况下，此工具将自动找到输入向量中的分类特征，并将它们转换为具有从零开始的分类索引的分类特征。这有助于将未知向量的数据集处理为具有一些连续特征和一些分类特征的数据集。连续和分类之间的选择基于 maxCategories 参数。

将 maxCategories 设置为任何分类特征应该具有的类别的最大数量。通过在 VectorIndexer 中将 maxCategories 设置为 n，将指示 Spark 接受向量中具有 n 个或 n 个以下不同值的任何列，并将其转换为分类变量。例如，特征 0 具有值{-1.0, 0.0}，特征 1 具有值{1.0, 3.0, 5.0}。如果 maxCategories = 2，那么特征 0 将被声明为分类的，并使用索引{0,1}，特征 1 将被声明为连续的。

在下面的示例中，构造了一个向量数据集，并使用 VectorIndexer Estimator 将输入向量中的分类特征自动转换为使用基于 0 的索引。

```
import org.apache.spark.ml.feature.VectorIndexer
import org.apache.spark.ml.linalg.Vectors

// 构造 DataFrame,元素为 Vector 向量
val idxIn = spark.createDataFrame(
    Seq((Vectors.dense(1, 2, 3),1),
        (Vectors.dense(2, 5, 6),2),
        (Vectors.dense(1, 8, 9),3)
    )
).toDF("features", "label")

// 构造 VectorIndexer Estimator
val indxr = new VectorIndexer()
    .setInputCol("features")   // 输入特征列
    .setOutputCol("idxed")     // 输出特征列
    .setMaxCategories(2)       // 指定最大类别数
```

```
// 拟合数据，返回模型
val vectorIndexerMode = indxr.fit(idxIn)

// 对数据进行转换
val inxOut = vectorIndexerMode.transform(idxIn)

// 显示结果
inxOut.show
```

执行以上代码，输出结果如下：

```
+-------------+-----+-------------+
|     features|label|        idxed|
+-------------+-----+-------------+
|[1.0,2.0,3.0]|    1|[0.0,2.0,3.0]|
|[2.0,5.0,6.0]|    2|[1.0,5.0,6.0]|
|[1.0,8.0,9.0]|    3|[0.0,8.0,9.0]|
+-------------+-----+-------------+
```

请注意观察输出结果中 features 特征的第一列和 idxed 特征的第一列，因为第一列只有两个分类值，所以会被转换为 0 和 1。而另外两列的分类值都超过了两个，所以会被当作连续变量，不进行分类特征转换。

3.3.4 OneHotEncoder

在特征转换过程中，索引分类变量只是完成了事情的一半。独热编码（one-hot encoding）是在索引分类变量之后执行的一种非常常见的数据转换。这是因为索引并不总是以下游模型处理的正确方式表示分类变量。比如，在某个数据集中有一个表示颜色的 color 列，当我们为 color 列建立索引时，会注意到一些颜色的值（即索引编号）比其他颜色的值更高，例如，蓝色是 1，绿色是 2，等等。这是不正确的，因为它给出的数学表示，机器学习算法的输入似乎指定了"绿色>蓝色"，这在当前类别的情况下没有意义。

为了避免这种情况，我们使用 OneHotEncoder 把每个不同的分类值转换为布尔标志（1 或 0），作为向量中的一个组成部分。当对颜色值进行编码时，可以看到它们不再是有序的，这使得下游模型（例如线性模型）更容易处理。

Label Encoding

Food Name	Categorical #	Calories
Apple	1	95
Chicken	2	231
Broccoli	3	50

→

One Hot Encoding

Apple	Chicken	Broccoli	Calories
1	0	0	95
0	1	0	231
0	0	1	50

例如，下面的数据代表学生专业，每个专业被分配一个顺序值，这似乎表明某个专业比其他专业要高。为了在机器学习训练步骤中消除这种无意的偏差，我们使用 OneHotEncoder Estimator 将序数值转换成向量。请看下面的示例代码。

```
// 使用 OneHotEncoder Transformer 将序数值转换为分类值
import org.apache.spark.ml.feature.OneHotEncoder

// 创建 DataFrame
```

```
val student_major_df = spark.createDataFrame(
    Seq(("张三", "数学", 3),
        ("李四", "机械工程", 2),
        ("王老五", "哲学", 7),
        ("赵小六", "数学", 3),
        ("钱多多", "护理学", 4)
    )).toDF("user", "major", "majorIdx")

// 构造 OneHotEncoder，指定输入特征列和输出特征列
val oneHotEncoder = new OneHotEncoder()
    .setInputCol("majorIdx")        // 输入列
    .setOutputCol("majorVect")      // 输出列

// 拟合数据，返回模型
val oneHotEncoderModel = oneHotEncoder.fit(student_major_df)

// 使用模型转换数据
val student_major_data = oneHotEncoderModel.transform(student_major_df)

// 查看
student_major_data.show()
```

执行以上代码，输出结果如下：

```
+------+--------+--------+-------------+
|  user|   major|majorIdx|    majorVect|
+------+--------+--------+-------------+
|  张三|    数学|       3|(7,[3],[1.0])|
|  李四|机械工程|       2|(7,[2],[1.0])|
|王老五|    哲学|       7|    (7,[],[])|
|赵小六|    数学|       3|(7,[3],[1.0])|
|钱多多|  护理学|       4|(7,[4],[1.0])|
+------+--------+--------+-------------+
```

如果类别值是字符串类型的，那么首先应用 StringIndexer Estimator 将它们转换成数值类型。OneHotEncoder Estimator 将一个数字的分类值映射到一个二元向量中，以有意地删除数字分类值的隐式排序。在下面的示例中，使用 OneHotEncoder Estimator 进一步处理 StringIndexer Estimator 的输出。

```
import org.apache.spark.ml.feature.OneHotEncoder
import org.apache.spark.ml.feature.StringIndexer

// 构造电影类别数据集
val movie_data = Seq(
       (1, "Comedy"),
       (2, "Action"),
       (3, "Comedy"),
       (4, "Horror"),
       (5, "Action"),
       (6, "Comedy")
   ).toDF("id", "genre")

// movie_data.show

// 使用 StringIndexer estimator 来对电影类型进行编码
val movieIndexer = new StringIndexer()
    .setInputCol("genre")
```

```
    .setOutputCol("genreIdx")

// 首先拟合数据
val movieIndexModel = movieIndexer.fit(movie data)

// 使用返回的 transformer 来转换该数据
val indexedMovie = movieIndexModel.transform(movie data)

// 输入列 genreIdx 是 StringIndex 的输出列
val oneHotEncoderEst = new OneHotEncoder()
    .setInputCol("genreIdx")
    .setOutputCol("genreIdxVector")

// 拟合并转换 indexedMovie 数据（在上一步中产生的）
val oneHotEncoderModel = oneHotEncoderEst.fit(indexedMovie)
val oneHotEncoderVect = oneHotEncoderModel.transform(indexedMovie)

// 显示
oneHotEncoderVect.orderBy("genre").show()
```

执行以上代码，输出结果如下：

```
+---+------+--------+--------------+
| id| genre|genreIdx|genreIdxVector|
+---+------+--------+--------------+
|  5|Action|     1.0| (2,[1],[1.0])|
|  2|Action|     1.0| (2,[1],[1.0])|
|  1|Comedy|     0.0| (2,[0],[1.0])|
|  6|Comedy|     0.0| (2,[0],[1.0])|
|  3|Comedy|     0.0| (2,[0],[1.0])|
|  4|Horror|     2.0|     (2,[],[])|
+---+------+--------+--------------+
```

OneHotEncoder Estimator 可以转换多个列，为每个输入列返回一个 one-hot-encoded 的输出向量列。通常使用 VectorAssembler 将这些向量合并成单个特征向量。

OneHotEncoder Estimator 支持 handleInvalid 参数来选择在转换数据（即执行 transform()方法）期间如何处理无效输入。可用选项包"keep"（将任何无效的输入分配给额外的分类索引）和"error"（抛出错误）。

需注意的是最后一个分类被编码为全 0 向量，若希望该分类也占有一个二进制特征，则可在创建 OneHotEncoder 时指定 setDropLast(false)。

3.4 特征转换-文本数据

在 MLlib 中使用 ML 算法是有严格要求的：它们要求所有的特征都是 Double 数据类型，包括标签。因此，数据集中的文本数据通常需要进行大量操作才能映射到机器学习模型能够有效使用的格式。通常会看到两种文本：自由格式的文本和字符串分类变量。本节主要关注自由格式文本（上一节已经讨论了分类变量）。

3.4.1 文本分词

Tokenization 是将自由格式的文本转换为一系列“tokens”或单个单词的过程。最简单的方法是使用 Tokenizer 类。这个转换器将获取由空格分隔的单词字符串，并将它们转换为单词数组。它将输入字符串转换为小写字母，然后用空格分隔。

在很多机器学习用例中，其输入是自由格式的文本，因此需要一些转换来将自由格式的文本转换成 ML 算法可以使用的数字表示。其中包括分词和统计词频。

Tokenizer Transformer 对由空格分隔的字符串进行分词，并返回一个单词数组。请看下面这个使用 Tokenizer transformer 的示例。

```
// 使用 Tokenizer Transformer 执行分词
import org.apache.spark.ml.feature.Tokenizer
import org.apache.spark.sql.functions._

// 构造一个 DataFrame
val text_data = spark.createDataFrame(
    Seq((1, "Spark is a unified data analytics engine"),
        (2, "It is fun to work with Spark"),
        (3, "There is a lot of exciting sessions at upcoming Spark summit"),
        (4, "mllib transformer estimator evaluator and pipelines")
    )).toDF("id", "line")

// 创建分词器，指定输入特征列和输出特征列
val tokenizer = new Tokenizer().setInputCol("line").setOutputCol("words")

// 转换数据集
val tokenized = tokenizer.transform(text_data)

// 查看结果
tokenized
    .select("words")
    .withColumn("tokens", size(col("words")))
    .show(false)
```

执行以上代码，输出结果如下：

```
+-----------------------------------------------------------------+------+
|words                                                            |tokens|
+-----------------------------------------------------------------+------+
|[spark, is, a, unified, data, analytics, engine]                 |7     |
|[it, is, fun, to, work, with, spark]                             |7     |
|[there,is,a,lot,of, exciting, sessions,at,upcoming,spark,summit]|11    |
|[mllib, transformer, estimator, evaluator, and, pipelines]       |6     |
+-----------------------------------------------------------------+------+
```

如果需要用不同的分隔符执行分词，那么可以使用 RegexTokenizer。可以使用 RegexTokenizer 创建一个基于正则表达式的 Tokenizer。正则表达式的格式应该符合 Java 正则表达式语法。请看下面的示例。

```
import org.apache.spark.ml.feature.RegexTokenizer
import org.apache.spark.sql.functions.

// 创建一个 DataFrame
val textDF = spark.createDataFrame(
    Seq((1, "Spark is a unified data analytics engine"),
        (2, "It is fun to work with Spark"),
        (3, "There is a lot of exciting sessions at upcoming Spark summit"),
        (4, "mllib transformer estimator evaluator and pipelines")
    )).toDF("id", "line")

// 创建分词器
val rt = new RegexTokenizer()
```

```
    .setInputCol("line")    // 输入列
    .setOutputCol("words")  // 输出列
    .setPattern(" ")       // 分隔符
    .setToLowercase(true)   // 转换为小写

// 转换数据集
val tokenized = rt.transform(textDF)

// 查看
tokenized
    .select("words")
    .withColumn("tokens", size($"words"))
    .show(false)
```

执行以上代码，输出结果如下：

```
+-----------------------------------------------------------------------+------+
|words                                                                  |tokens|
+-----------------------------------------------------------------------+------+
|[spark, is, a, unified, data, analytics, engine]                       |7     |
|[it, is, fun, to, work, with, spark]                                   |7     |
|[there,is,a,lot,of, exciting, sessions, at, upcoming, spark, summit]|11    |
|[mllib, transformer, estimator, evaluator, and, pipelines]             |6     |
+-----------------------------------------------------------------------+------+
```

使用 RegexTokenizer 的另一种方法是用它来输出与提供的模式匹配的值，而不是使用它作为分隔符。为此，需要将 gaps 参数设置为 false。

例如，我们要捕获每行文本中的单词 spark，代码如下。

```
import org.apache.spark.ml.feature.RegexTokenizer
import org.apache.spark.sql.functions._

// 创建 DataFrame
val textDF = spark.createDataFrame(
    Seq((1, "Spark is a unified data analytics engine"),
        (2, "It is fun to work with Spark"),
        (3, "There is a lot of exciting sessions at Spark summit"),
        (4, "mllib transformer estimator evaluator and pipelines")
    )).toDF("id", "line")

// 创建分词器，指定各个参数
val rt = new RegexTokenizer()
    .setInputCol("line")     // 输入列
    .setOutputCol("words")   // 输出列
    .setPattern("spark")     // 要捕获的单词
    .setGaps(false)         // 设为 false
    .setToLowercase(true)    // 匹配前转换为小写

// 转换数据集
val tokenized = rt.transform(textDF)

// 查看结果
tokenized.show(false)
```

执行以上代码，输出结果如下：

```
+---+-----------------------------------------------------+--------+
|id |line                                                 |words   |
```

```
+---+----------------------------------------------------+-------+
|1  |Spark is a unified data analytics engine           |[spark]|
|2  |It is fun to work with Spark                       |[spark]|
|3  |There is a lot of exciting sessions at Spark summit|[spark]|
|4  |mllib transformer estimator evaluator and pipelines |[]     |
+---+----------------------------------------------------+-------+
```

3.4.2 停止词

停止词（stop words）指的是一种在许多类型的分析中不相关、因此应该删除的常见词。停止词是应该从输入中排除的词，通常是因为这些词出现频率高，没有太多意义。英语中经常出现的停止词包括 the、and 和 but 等。在自然语言处理或机器学习的背景下，停止词往往会增加不必要的噪音，而不是提供任何有意义的贡献。因此，通常在分词步骤之后立即执行停止词删除步骤。

Spark 机器学习库中提供了 StopWordsRemover Transformer，它是设计用来实现这个目的的。StopWordsRemover 接受一个字符串序列作为输入（例如 Tokenizer 的输出），并删除输入序列中的所有停止词。停止词列表由列表由 stopWords 参数指定。除非显式地将 null 添加到 stopWords 中，否则将保留输入数组中的 null 值。

通过调用 StopWordsRemover.loadDefaultStopWords(language)可以访问某些语言的默认停止词。支持的语言有“danish”、“dutch”、“english”、“finnish”、“french”、“german”、“hungarian”、“italian”、“norwegian”、“portuguese”、“russian”、“spanish”、“swedish”以及“turkish”。

要使用特定语言的停止词，首先调用 StopWordsRemover.loadDefaultStopWords(language)加载它们（Array[String]形式），然后将它们提供给 StopWordsRemover 的一个实例。此外，如果需要，还可以请求该 Transformer 执行停止词过滤，并在必要时不区分大小写。

在下面的示例中，使用 StopWordsRemover Transformer 来删除 Tokenization 分词后单词中的英语停止词。

```
import org.apache.spark.ml.feature.Tokenizer
import org.apache.spark.sql.functions.
import org.apache.spark.ml.feature.StopWordsRemover

// 构造一个 DataFrame
val text data = spark.createDataFrame(
    Seq((1, "Spark is a unified data analytics engine"),
        (2, "It is fun to work with Spark"),
        (3, "There is a lot of exciting sessions at Spark summit"),
        (4, "mllib transformer estimator evaluator and pipelines")
    )).toDF("id", "line")

// 创建分词器，指定输入特征列和输出特征列
val tokenizer = new Tokenizer().setInputCol("line").setOutputCol("words")

// 转换数据集
val tokenized = tokenizer.transform(text_data)
tokenized.select("words").show(false)

// 加载英语中的停止词
val enStopWords = StopWordsRemover.loadDefaultStopWords("english")
```

```
// 创建 StopWordsRemover Transformer 实例
val remover = new StopWordsRemover()
    .setStopWords(enStopWords)
    .setInputCol("words")
    .setOutputCol("filtered")

// 转换：过滤停止词
val cleanedTokens = remover.transform(tokenized)

// 查看结果
cleanedTokens.select("filtered").show(false)
```

执行以上代码，输出结果如下：

```
+----------------------------------------------------------------+
|words                                                           |
+----------------------------------------------------------------+
|[spark, is, a, unified, data, analytics, engine]                |
|[it, is, fun, to, work, with, spark]                            |
|[there, is, a, lot, of, exciting, sessions, at, spark, summit]|
|[mllib, transformer, estimator, evaluator, and, pipelines]      |
+----------------------------------------------------------------+

+-----------------------------------------------------+
|filtered                                             |
+-----------------------------------------------------+
|[spark, unified, data, analytics, engine]            |
|[fun, work, spark]                                   |
|[lot, exciting, sessions, spark, summit]             |
|[mllib, transformer, estimator, evaluator, pipelines]|
+-----------------------------------------------------+
```

从输出结果可以看到，分词以后的停止词（如 is、a、the、it、there、of、at 等）已经被过滤掉了。

可进一步使用 HashingTF transformer 将单词转换为数字表示，实现代码如下。

```
import org.apache.spark.ml.feature.HashingTF

// 创建 HashingTF 实例
val tf = new HashingTF()
    .setInputCol("filtered")  // 指定输入列
    .setOutputCol("TFOut")    // 指定输出列
    .setNumFeatures(4096)     // 指定索引空间大小

// 将单词转换为索引数字表示
val tfResult = tf.transform(cleanedTokens)

// 查看
tfResult.select("filtered", "TFOut").show(false)
```

执行以上代码，输出结果如下：

```
+-----------------------------------------------------+------------------
-------------------------------------+
|filtered                                             |TFOut
|
```

```
+-----------------------------------------------------+-----------------
-------------------------------------+
|[spark, unified, data, analytics, engine]
|(4096,[296,991,1526,2252,3992],[1.0,1.0,1.0,1.0,1.0])  |
|[fun, work, spark]
|(4096,[1526,1575,2607],[1.0,1.0,1.0])                 |
|[lot, exciting, sessions, spark, summit]
|(4096,[1526,2620,2966,3023,3935],[1.0,1.0,1.0,1.0,1.0])|
|[mllib, transformer, estimator, evaluator,
pipelines]|(4096,[705,1151,2097,2973,3694],[1.0,1.0,1.0,1.0,1.0]) |
+-----------------------------------------------------+-----------------
-------------------------------------+
```

```
+-----------------------------------------------+-----------------------------------------------------+
|filtered                                       |TFOut                                                |
+-----------------------------------------------+-----------------------------------------------------+
|[spark, unified, data, analytics, engine]      |(4096,[296,991,1526,2252,3992],[1.0,1.0,1.0,1.0,1.0])|
|[fun, work, spark]                             |(4096,[1526,1575,2607],[1.0,1.0,1.0])                |
|[lot, exciting, sessions, spark, summit]       |(4096,[1526,2620,2966,3023,3935],[1.0,1.0,1.0,1.0,1.0])|
|[mllib, transformer, estimator, evaluator, pipelines]|(4096,[705,1151,2097,2973,3694],[1.0,1.0,1.0,1.0,1.0])|
+-----------------------------------------------+-----------------------------------------------------+
```

3.4.3 创建单词组合 n-gram

对字符串进行分词和过滤停止词为我们提供了一组可用作特征的干净的单词。观察单词的组合通常是很有意义的，经常是通过观察与之搭配的单词。单词组合在技术上称为 n-grams，即长度为 n 的单词序列。创建 n-gram 的目的是为了更好地捕捉句子结构，这比简单地单独查看所有单词能获得更多信息。

在创建 n-gram 时，顺序很重要，因此将一个包含三个单词的句子转换为两个单词的组合（称为 bigram）表示将产生两个 bigram。例如，将语句“Big Data Processing Made Simple”转换为 bigram，结果如下：

(1)“Big Data”

(2)“Data Processing”

(3)“Processing Made”

(4)“Made Simple”

如果转换为三个单词的组合（称为 trigrams），则结果如下：

(1)“Big Data Processing”

(2)“Data Processing Made”

(3)“Procesing Made Simple”

有了 n-gram，我们就可以查看经常同时出现的单词序列，并将它们作为机器学习算法的输入。这可以创造出比简单地单独查看所有的单词更好的特征。请看下面的示例。

```
import org.apache.spark.ml.feature.NGram

// 创建 DataFrame
val wordDataFrame = spark.createDataFrame(Seq(
  (0, Array("Hi", "I", "heard", "about", "Spark")),
  (1, Array("I", "wish", "Java", "could", "use", "case", "classes")),
  (2, Array("Logistic", "regression", "models", "are", "neat"))
)).toDF("id", "words")

// 创建 NGram 实例
```

```
val ngram = new NGram().setN(2).setInputCol("words").setOutputCol("ngrams")

// 转换数据
val ngramDataFrame = ngram.transform(wordDataFrame)

// 查看结果
ngramDataFrame.select("words","ngrams").show(false)
```

执行以上代码，输出结果如下：

```
+----------------------------------------+----------------------------
----------------------------------+
|words                                   |ngrams
|
+----------------------------------------+----------------------------
----------------------------------+
|[Hi, I, heard, about, Spark]            |[Hi I, I heard, heard about, about
Spark]                       |
|[I, wish, Java, could, use, case, classes]|[I wish, wish Java, Java could,
could use, use case, case classes]|
|[Logistic, regression, models, are, neat] |[Logistic regression, regression
models, models are, are neat]    |
+----------------------------------------+----------------------------
----------------------------------+
```

```
+----------------------------------------+------------------------------------------------------------------------+
|words                                   |ngrams                                                                  |
+----------------------------------------+------------------------------------------------------------------------+
|[Hi, I, heard, about, Spark]            |[Hi I, I heard, heard about, about Spark]                               |
|[I, wish, Java, could, use, case, classes]|[I wish, wish Java, Java could, could use, use case, case classes]|
|[Logistic, regression, models, are, neat] |[Logistic regression, regression models, models are, are neat]          |
+----------------------------------------+------------------------------------------------------------------------+
```

3.4.4 将单词转换为数字表示形式

一旦有了单词特征，就该开始计算在我们的模型中使用的单词和单词组合的实例了。最简单的方法是在给定的文档（在我们的例子中是一行文本）中包含一个单词的二进制计数。本质上，我们度量的是每一行是否包含给定的单词。这是对文档大小和出现次数规范化并获得文档分类数字特征的简单方法。Spark 机器学习库中提供了 CountVectorizer 对单词进行计数，使用 TF-IDF 转换根据给定单词在所有文档中的流行程度并重新对它们进行加权。

1. CountVectorizer Estimator

CountVectorizer 从文档集合中提取词汇表并生成 CountVectorizerModel。它和它返回的模型 CountVectorizerModel 旨在帮助将文本文档集合转换为 token 计数的向量，用来进行特征抽取。当一个先验字典不可用时，CountVectorizer 可以作为一 Estimator 估计器来提取词汇表，并生成一个 CountVectorizerModel。该模型为词汇表上的文档生成稀疏表示，然后可以将其传递给其他算法，如 LDA。

CountVectorizer 主要做两件事：

(1) 在 fit()过程中，它在所有文档中查找单词集，然后计算这些单词在这些文档中的出现次数。

(2) 然后，它会对在转换过程中 DataFrame 列的每一行中出现的给定单词进行计数，并输出一个向量，向量中包含该行中出现的单词项。

从概念上讲，这种转换将每一行视为一个文档（document），将每一个单词视为一项（term），并将所有项的汇总作为词汇表（vocabulary）。

CountVectorizer 可以指定一些参数，如 minTF、minDF、vocabSize 等。这些都是可调参数，这意味着用户可以为词汇表中包含的词汇设置最小词汇频率（minTF，这可以有效去除词汇表中少见的单词）；在包含在词汇表中之前，一个词汇必须出现在其中的最小文档数量（minDF，另一种从词汇表中删除少见词汇的方法）；以及总的最大词汇（vocabSize）。最后，默认情况下，CountVectorizer 将输出文档中某个单词项的计数。要返回文档中是否存在某个单词，可以使用 setBinary(true)方法。

下面这个示例演示了 CountVectorizer Estimator 的用法。

```
import org.apache.spark.ml.feature.{CountVectorizer, CountVectorizerModel}

// 假设有以下文档
val document = Seq(
    (0, Array("a", "b", "c", "e")),
    (1, Array("a", "b", "b", "c", "a"))
)

// 构造为带有列 id 和 texts 的 DataFrame
val df = spark.createDataFrame(document).toDF("id", "words")

// df.show
/*
+---+---------------+
| id|         words|
+---+---------------+
|  0|   [a, b, c, e]|
|  1|[a, b, b, c, a]|
+---+---------------+
*/

// 从语料库中拟合 CountVectorizerModel
// 设定词汇表中的词至少要在 2 个文档中出现过，以过滤那些偶然出现的词汇
val cv = new CountVectorizer()
       .setInputCol("words")
       .setOutputCol("features")
       .setVocabSize(3)
       .setMinDF(2)

// 拟合数据，返回模型
val cvModel = cv.fit(df)

// 使用这一模型对 DataFrame 进行变换，可以得到文档的向量化表示
cvModel.transform(df).show(false)

// 获得到模型的词汇表
println("词汇表: " + cvModel.vocabulary.toList)
```

在 CountVectorizerModel 的训练过程中，CountVectorizer 将根据语料库中的词频排序从高到低进行选择，词汇表的最大词汇量由 vocabSize 超参数来指定（方法 setVocabSize(3)）。超参数 minDF 则指定词汇表中的词语至少要在多少个不同文档中出现。

执行以上代码，输出结果如下：

```
+---+---------------+-------------------------+
|id |words          |features                 |
+---+---------------+-------------------------+
|0  |[a, b, c, e]   |(3,[0,1,2],[1.0,1.0,1.0])|
|1  |[a, b, b, c, a]|(3,[0,1,2],[2.0,2.0,1.0])|
+---+---------------+-------------------------+

词汇表: List(b, a, c)
```

输出结果虽然输出看起来有点复杂，但它实际上只是一个稀疏向量，包含词汇表的总大小、词汇表中单词的索引，以及特定单词的计数。

在训练结束后，可以通过 CountVectorizerModel 的 vocabulary 成员获得到模型的词汇表。可以看到，词汇表中有 a、b、c 三个词，且这三个词都在两个文档中出现过（代码中设定了 minDF 为 2）。

和其他 Transformer 不同，CountVectorizerModel 可以通过指定一个先验词汇表来直接生成。例如，在以下示例中，直接指定词汇表的成员是 a、b、c 三个词。

```
val cvm = new CountVectorizerModel(Array("a", "b", "c"))
       .setInputCol("words")
       .setOutputCol("features")

cvm.transform(df).show(false)
```

执行以上代码，输出结果如下：

```
+---+---------------+-------------------------+
|id |words          |features                 |
+---+---------------+-------------------------+
|0  |[a, b, c, e]   |(3,[0,1,2],[1.0,1.0,1.0])|
|1  |[a, b, b, c, a]|(3,[0,1,2],[2.0,2.0,1.0])|
+---+---------------+-------------------------+
```

2. HashingTF Transformer

处理将文本转换为数字表示的问题的另一种方法是使用词频-逆文件频率（TF-IDF）。用最简单的术语来说，TF-IDF 度量一个单词在每个文档中出现的频率，根据该单词出现在多少文档中进行加权。其结果是，出现在少数文档中的单词比出现在许多文档中的单词具有更大的权重。在实践中，像 a、the、of 这样的词由于其普遍性而权重很低，而象 streaming 这样更专业的词出现在较少的文件中，因此权重较高。在某种程度上，TF-IDF 帮助查找具有类似主题的文档。

Spark 机器学习库中的 HashingTF Transformer 用于通过计算每行中每个单词的频率来将单词转换成数字表示。每个单词都通过一个名为 MurmurHash3 的哈希函数映射到一个索引中。这种方法虽然有效，但是它可能会遇到潜在的哈希冲突（也叫哈希碰撞），即多个单词可能映射到同一个索引。一种最小化碰撞的方法是通过指定大量的桶（桶数量是 2 的幂）来帮助均匀地分布单词。下面是一个简单的 HashingTF Transformer 使用示例。

```
// 简单 HashingTF 示例
import org.apache.spark.ml.feature.HashingTF

// 文档数据
val document = Seq(
```

```
    (0, Array("a", "b", "c", "e")),
    (1, Array("a", "b", "b", "c", "a"))
)

// 构造为带有列 id 和 texts 的 DataFrame
val df = spark.createDataFrame(document).toDF("id", "words")

// 创建 HashingTF 实例
val hashingTF = new HashingTF()
    .setInputCol("words")
    .setOutputCol("rawFeatures")
    .setNumFeatures(100)

// 转换数据集
val featurizedData = hashingTF.transform(df)

// 查看转换结果
featurizedData.show(false)
```

执行以上代码，输出结果如下：

```
+---+---------------+-------------------------------------+
|id |words          |rawFeatures                          |
+---+---------------+-------------------------------------+
|0  |[a, b, c, e]   |(100,[50,65,67,68],[1.0,1.0,1.0,1.0])|
|1  |[a, b, b, c, a]| (100,[65,67,68],[2.0,2.0,1.0])      |
+---+---------------+-------------------------------------+
```

可以看出，结果向量（rawFeatures 列）使用三个不同的值表示：向量空间的大小、文档中出现的每个单词项在向量空间中的索引位置以及这些单词项出现的次数。

补充：我们通过下面的代码来理解 HashingTF 的转换过程。

```
// 有一个单词数组，计算该数组中单词的频率向量
val words = Array("the", "dog", "jumped", "over", "the")

// 第一步是计算数组中每个元素的哈希码。可以使用内置的##方法来计算任何对象的哈希码
val hashCodes = words.map { _.## }

// 为了将哈希码转换为有效的向量索引，取每个哈希的模(哈希值除以向量的大小，本例中为 16)：
val indices = hashCodes.map { code => Math.abs(code % 16) }

// 然后可以创建一个从索引到该索引出现次数的映射：
val indexFrequency = indices.groupBy(identity).mapValues {_.size.toDouble}

// 最后，可以把这个映射转换成一个稀疏向量，其中向量中每个元素的值就是这个特定索引出现的频率：
import org.apache.spark.ml.linalg._

val termFrequencies = Vectors.sparse(16, indexFrequency.toSeq)
```

执行以上代码，输出结果如下所示：

```
words: Array[String] = Array(the, dog, jumped, over, the)
hashCodes: Array[Int] = Array(114801, 99644, -1148867251, 3423444, 114801)
indices: Array[Int] = Array(1, 12, 3, 4, 1)
indexFrequency: scala.collection.immutable.Map[Int,Double] = Map(4 -> 1.0, 1 -> 2.0, 3 -> 1.0, 12 -> 1.0)
import org.apache.spark.ml.linalg._
termFrequencies: org.apache.spark.ml.linalg.Vector = (16,[1,3,4,12],[2.0,1.0,1.0,1.0])
```

注意，稀疏向量的.tostring()方法的输出由三个元素组成：向量的总大小，然后是两个列表，第一个是一系列索引，第二个是这些索引上的一系列值（频次）。使用稀疏向量提供了一种紧凑而有效的方法来表示消息中单词出现的频率，这正是 HashingTF 的工作原理。缺点是，从单词到索引的映射不一定是惟一的：按向量长度截断哈希代码将把不同的字符串映射到相同的索引。这就是所谓的"哈希碰撞"。解决办法是使 n 足够大，使碰撞的频率最小化。

3. IDF Estimator

IDF 是用于处理文本的常用的 Estimators 之一。它的名字是“inverse document frequency（反转文档频率）”的缩写。这个 Estimator 经常在文本被分词和术语频率被计算之后立即使用。

IDF 背后的思想是通过计算它出现在的文档数量来计算每个单词的重要性或权重。这个想法背后的直觉是，一个高发生率和广泛流行的词就不那么重要了，例如，the 这个单词。相反，一个高频率出现的词，只出现在少数几个文档中，就会显示出更高的重要性，例如，单词 classification。在 DataFrame 上下文中，一个文档是指一行。为了计算每个单词的重要性，它需要遍历每一行，因此 IDF 是一个 Estimator，而不是 Transformer。

下面的示例中，先使用 HashingTF 对文档中的术语频次向量化，然后使用 IDF 来对单纯的词频特征向量进行修正，使其更能体现不同词汇对文本的区别能力。完整的代码如下：

```
// 简单 HashingTF 和 IDF 示例
import org.apache.spark.ml.feature.{HashingTF,IDF}

// 文档
val document = Seq(
    (0, Array("a", "b", "c", "e")),
    (1, Array("a", "b", "b", "c", "a"))
)

// 构造为带有列 id 和 texts 的 DataFrame
val df = spark.createDataFrame(document).toDF("id", "words")

// HashingTF Transformer 转换
val hashingTF = new HashingTF()
    .setInputCol("words")
    .setOutputCol("rawFeatures")
    .setNumFeatures(100)

val featurizedData = hashingTF.transform(df)

// IDF 是一个 Estimator，调用 fit()方法并将词频向量传入，即产生一个 IDFModel
val idf = new IDF().setInputCol("rawFeatures").setOutputCol("features")
val idfModel = idf.fit(featurizedData)


// IDFModel 是一个 Transformer
// 调用它的 transform()方法，即可得到每一个单词对应的 TF-IDF 度量值
val rescaledData = idfModel.transform(featurizedData)
rescaledData.show(false)
```

执行以上代码，输出结果如下：

```
+---+---------------+-----------------------------------+-----------------------------------------------------+
|id |words          |rawFeatures                        |features                                             |
+---+---------------+-----------------------------------+-----------------------------------------------------+
|0  |[a, b, c, e]   |(100,[50,65,67,68],[1.0,1.0,1.0,1.0])|(100,[50,65,67,68],[0.4054651081081644,0.0,0.0,0.0])|
|1  |[a, b, b, c, a]|(100,[65,67,68],[2.0,2.0,1.0])     |(100,[65,67,68],[0.0,0.0,0.0])                       |
+---+---------------+-----------------------------------+-----------------------------------------------------+
```

可以看出，结果向量（features 列）使用三个不同的值表示：特征向量空间的总大小、文档中出现的每个术语在向量空间中的索引位置以及这些术语的权重。这类似于 CountVectorizer 的输出。可以看到，特征向量已经对其在语料库中出现的总次数进行了修正，修正为了权重表示。通过 TF-IDF 得到的特征向量，接下来可以被应用到相关的机器学习方法中。

下面这个示例综合使用了 Tokenizer、HashingTF 和 IDF，来提取文档的 TF-IDF 度量值。

```
import org.apache.spark.ml.feature._

// 文档数据，每一个句子代表一个文档
val document = Seq(
  (0.0, "Hi I heard about Spark"),
  (0.0, "I wish Java could use case classes"),
  (1.0, "Logistic regression models are neat")
)

// 构造 DataFrame
val sentenceData = spark
     .createDataFrame(document)
     .toDF("label", "sentence")

// 在得到文档集合后，即可用 Tokenizer Transformer 对句子进行分词。
val tokenizer = new Tokenizer()
     .setInputCol("sentence")
     .setOutputCol("words")
val wordsData = tokenizer.transform(sentenceData)
// wordsData.show(false)

// 得到分词后的文档序列后，即可使用 HashingTF 的 transform()方法把句子哈希成特征向量
// 这里设置哈希表的桶数为 100
val hashingTF = new HashingTF()
    .setInputCol("words")
    .setOutputCol("rawFeatures")
    .setNumFeatures(100)
val featurizedData = hashingTF.transform(wordsData)

// 最后，使用 IDF 来对单纯的词频特征向量进行修正，使其更能体现不同词汇对文本的区别能力
// IDF 是一个 Estimator，调用 fit()方法并将词频向量传入，即产生一个 IDFModel
val idf = new IDF().setInputCol("rawFeatures").setOutputCol("features")
val idfModel = idf.fit(featurizedData)

// IDFModel 是一个 Transformer
// 调用它的 transform()方法，即可得到每一个单词对应的 TF-IDF 度量值
val rescaledData = idfModel.transform(featurizedData)

// 查看结果
rescaledData.select("label", "words", "features").show(false)
```

执行以上代码，输出结果如下：

```
+-----+----------------------------------------+-----------------------------------------------------------------------------------------------------------------------------------------------------------+
|label|words                                   |features                                                                                                                                                   |
+-----+----------------------------------------+-----------------------------------------------------------------------------------------------------------------------------------------------------------+
|0.0  |[hi, i, heard, about, spark]            |(16,[0,1,2,3,10],[0.28768207245178085,0.6931471805599453,0.6931471805599453,0.6931471805599453,0.6931471805599453])                                        |
|0.0  |[i, wish, java, could, use, case, classes]|(16,[0,4,5,7,11,14,15],[0.28768207245178085,0.6931471805599453,0.6931471805599453,0.6931471805599453,0.6931471805599453,0.6931471805599453,0.6931471805599453])|
|1.0  |[logistic, regression, models, are, neat]|(16,[6,8,9,12,13],[0.6931471805599453,0.6931471805599453,0.6931471805599453,0.6931471805599453,0.6931471805599453])                                       |
+-----+----------------------------------------+-----------------------------------------------------------------------------------------------------------------------------------------------------------+
```

4. Word2Vec Estimator

Word2Vec 是在文本中使用的另一个有意思的 Estimator，它代表“words to vector”（单词到向量）。Word2Vec 利用了一种众所周知的技术，叫做“word embeddings”，它将单词 token 转换成数字向量表示，这样语义相似的单词就会被映射到临近的点。这种技术背后的直觉是，类似的词往往会同时出现，并且有相似的上下文。换句话说，有相似的相邻词的两个不同的词在意义上或与之相关的可能是非常相似的。这种技术已经被证明在许多自然语言处理应用程序中非常有效，例如单词类比、单词相似、实体识别和机器翻译。

Word2Vec 是一个基于深度学习的工具，用于计算一组单词的向量表示。我们的目标是在这个向量空间中让相似的词彼此接近，这样我们就可以对这些词本身进行概括。该模型易于训练和使用，并已被证明在许多自然语言处理应用程序中非常有用，包括实体识别、消歧、解析、标记和机器翻译。

值得注意的是，Word2Vec 可以根据语义捕获单词之间的关系。为此，Word2Vec 使用一种称为“skip-grams”的技术将一个由单词组成的句子转换为向量表示（可选特定大小）。它通过构建一个词汇表来实现这一点，然后对于每个句子，它删除一个 token，并训练模型预测在“n-gram”表示中缺失的 token。Word2Vec 最适合使用 tokens 形式的连续、自由格式的文本。

Word2Vec Estimator 有一些重要的配置参数，需要提供适当的值来控制基于输入的输出。表 3.1 描述了这样的配置：

表 1-1 消息格式

配置项	默认值	描述
vectorSize	100	这是输出向量的大小
windowSize	5	这是用作上下文的单词的数量
minCount	5	这是 token 必须出现在输出中的最小次数
maxSentenceLength	1000	这指定了其他术语之间的交互来创建新特征。数值采用乘法，分类值采用二值化。

下面的例子演示了如何使用 Word2Vec Estimator，并演示了如何找到相似的单词。

```
// 使用 Word2Vec Estimator 来计算单词的嵌入和发现类似的单词
import org.apache.spark.ml.feature.Word2Vec

// 构造一个 DataFrame
// 每一行是一个句子或文档中的单词包
val documentDF = spark.createDataFrame(
    Seq("Unified data analytics engine Spark".split(" "),
        "People use Hive for data analytics".split(" "),
        "MapReduce is not fading away".split(" ")
    ).map(Tuple1.apply)
  ).toDF("word")

// Word2Vec Estimator
val word2Vec = new Word2Vec()
```

```
    .setInputCol("word")
    .setOutputCol("feature")
    .setVectorSize(3)
    .setMinCount(0)

// 拟合并转换数据
val model = word2Vec.fit(documentDF)
val result = model.transform(documentDF)

// 查看结果
result.show(false)
```

代码中的超参含义如下：

(1) vectorSize：向量维度，它决定了训练过程中计算的维度，一般情况下 200 维够用。

(2) minCount：最小词频训练阈值，这个值根据训练语料大小设置，只有词频超过这个阈值的词才能被训练。可以通过设置相对大一点的 minCount 过滤掉切错的词。

执行以上代码，输出结果如下：

```
+----------------------------------------+-----------------------------------------------------------------+
|word                                    |feature                                                          |
+----------------------------------------+-----------------------------------------------------------------+
|[Unified, data, analytics, engine, Spark]|[-0.04907700642943383,-0.0399535870179534,-0.0047751694917678835]|
|[People, use, Hive, for, data, analytics]|[-0.019951647768417992,-0.002224805454413096,0.04896689318896581]|
|[MapReduce, is, not, fading, away]      |[0.09043671563267708,0.02406979613006115,0.005196916684508324]   |
+----------------------------------------+-----------------------------------------------------------------+
```

也可以遍历结果集，像下面这样查看。

```
import org.apache.spark.sql.Row

result.collect().foreach {
    case Row(text: Seq[_], features: Vector) =>
      println(s"Text: [${text.mkString(", ")}] => \nVector: $features\n")
}
```

执行以上代码，输出结果如下：

```
Text: [Unified, data, analytics, engine, Spark] =>
Vector: [-0.031449429923668504,0.005661628767848015,0.03859300017356873]

Text: [People, use, Hive, for, data, analytics] =>
Vector: [-0.06259135529398918,-0.013038620226628456,0.03549020915913085]

Text: [MapReduce, is, not, fading, away] =>
Vector: [0.007724904966016766,-0.04348084554076195,0.026679459214210513]
```

可以看到，文档被转变为了一个 3 维的特征向量，这些特征向量就可以被应用到相关的机器学习方法中。

进一步，我们可以利用计算的特征向量，来寻找同义词。例如：找出与单词 Spark 相似的其他 3 个单词，不包括 Spark 本身。实现代码如下。

```
model.findSynonyms("Spark", 3).show
```

执行以上代码，输出结果如下：

```
+---------+------------------+
|     word|        similarity|
+---------+------------------+
|   fading|0.8346112966537476|
```

```
|MapReduce|0.6870409846305847|
|     data|0.3523484468460083|
+---------+------------------+
```

3.5 特征操作

虽然机器学习库中的几乎每个 Transformer 都以某种方式操作特征空间，但以下算法和工具能够自动扩展输入特征向量或将其降至更低维数。

3.5.1 主成分分析(PCA)

主成分分析（PCA）是一种寻找数据中最重要特征（主成分）的数学技术。它通过创建一组新的特征来改变数据的特征表示；每个新特征都是原始特征的组合。PCA 的强大之处在于，它可以创建更小的、更有意义的特征集，并将其输入到模型中，但是这可能会以可解释性为代价。

注意： PCA 是一种统计过程，它使用正交变换将一组潜在相关的变量转换为一组线性不相关的简化变量。不相关变量的结果集称为主成分。PCA 类训练一个模型，使用 PCA 将向量投影到低维空间。

如果有一个大型的输入数据集，并且想要减少其所拥有的特征的总数，那么就需要使用 PCA。这在文本分析中经常出现，在这种情况下，整个特征空间非常庞大，而许多特征在很大程度上是不相关的。使用 PCA，我们可以找到最重要的特征组合，并且只将它们包含在我们的机器学习模型中。PCA 接受一个参数 K，指定要创建的输出特征的数量。一般来说，它应该比输入向量的维数小得多。

下面的示例展示了如何将 5 维特征向量投影到 3 维主成分中。

```
// 导入相关的机器学习库
import org.apache.spark.ml.feature.PCA
import org.apache.spark.ml.linalg.Vectors

// 创建一个 DataFrame
val data = Seq(
  Tuple1(Vectors.sparse(5, Seq((1, 1.0), (3, 7.0)))),
  Tuple1(Vectors.dense(2.0, 0.0, 3.0, 4.0, 5.0)),
  Tuple1(Vectors.dense(4.0, 0.0, 0.0, 6.0, 7.0))
)

val df = data.toDF("features")

df.printSchema()
df.show()

// 主分成分析
val pca = new PCA().setInputCol("features")
                .setOutputCol("pcaFeatures")
                .setK(3)
                .fit(df)

val result = pca.transform(df)
result.show(false)
```

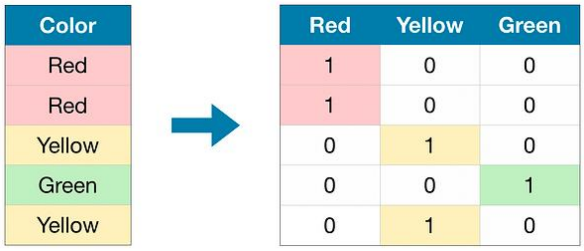
执行以上代码，输出结果如下：

```
root
 |-- features: vector (nullable = true)

+--------------------+
|            features|
+--------------------+
| (5,[1,3],[1.0,7.0])|
|[2.0,0.0,3.0,4.0,...|
|[4.0,0.0,0.0,6.0,...|
+--------------------+

+---------------------+-----------------------------------------------------
---------+
|features             |pcaFeatures                                          |
+---------------------+-----------------------------------------------------
---------+
|(5,[1,3],[1.0,7.0])
|[1.6485728230883807,-4.013282700516296,-5.524543751369388] |
|[2.0,0.0,3.0,4.0,5.0]|[-4.645104331781534,-1.1167972663619026,-5.5245437
51369387]|
|[4.0,0.0,0.0,6.0,7.0]|[-6.428880535676489,-5.337951427775355,-5.52454375
1369389] |
+---------------------+-----------------------------------------------------
---------+
```

3.5.2 特征交互 Interaction

在某些情况下，用户可能拥有关于数据集中特定变量的专业领域知识。例如，他可能知道两个变量之间的某种交互是下游 estimator 中要包含的一个重要变量。

Interaction 是一个 Transformer，它允许用户手动创建两个变量之间的交互，实现特征交互转换。该转换器接受 Double 和 Vector 类型列，并输出它们的特征交互的扁平向量。为了处理交互，首先要对任何名义特征（nominal features）进行 one-hot 编码。然后，得到特征交叉积的向量。例如，如果有 2 个向量类型的列，每个列都有 3 个维度作为输入列，那么就会得到一个 9 维向量作为输出列。



Color
Red
Red
Yellow
Green
Yellow

Red	Yellow	Green
1	0	0
1	0	0
0	1	0
0	0	1
0	1	0

例如，给定输入特征值 Double(2)和 Vector(3,4)，如果所有输入特征都是数字，则输出将是 Vector(6,8)。如果第一个特征是有四个类别的名义特征，则需要先进行 one-hot 编码，那么输出将是 Vector(0,0,0,0,3,4,0,0)。

```
// 导入相关包
import org.apache.spark.ml.feature.Interaction
import org.apache.spark.ml.feature.VectorAssembler

// 创建 DataFrame
val df = spark.createDataFrame(Seq(
```

```
  (1, 1, 2, 3, 8, 4, 5),
  (2, 4, 3, 8, 7, 9, 8),
  (3, 6, 1, 9, 2, 3, 6),
  (4, 10, 8, 6, 9, 4, 5),
  (5, 9, 2, 7, 10, 7, 3),
  (6, 1, 1, 4, 2, 8, 4)
)).toDF("id1", "id2", "id3", "id4", "id5", "id6", "id7")

df.printSchema()
df.show()

// 装配第一组特征向量
val assembler1 = new VectorAssembler()
    .setInputCols(Array("id2", "id3", "id4"))
    .setOutputCol("vec1")

val assembled1 = assembler1.transform(df)

// 装配第二组特征向量
val assembler2 = new VectorAssembler()
    .setInputCols(Array("id5", "id6", "id7"))
    .setOutputCol("vec2")

val assembled2 = assembler2.transform(assembled1)

val assemble = assembled2.select("id1", "vec1", "vec2")

// 两组特征向量交互
val interaction = new Interaction()
    .setInputCols(Array("id1", "vec1", "vec2"))
    .setOutputCol("interactedCol")

val interacted = interaction.transform(assemble)

interacted.show(truncate = false)
```

执行以上代码，输出结果如下：

```
root
 |-- id1: integer (nullable = false)
 |-- id2: integer (nullable = false)
 |-- id3: integer (nullable = false)
 |-- id4: integer (nullable = false)
 |-- id5: integer (nullable = false)
 |-- id6: integer (nullable = false)
 |-- id7: integer (nullable = false)

+---+---+---+---+---+---+---+
|id1|id2|id3|id4|id5|id6|id7|
+---+---+---+---+---+---+---+
|  1|  1|  2|  3|  8|  4|  5|
|  2|  4|  3|  8|  7|  9|  8|
|  3|  6|  1|  9|  2|  3|  6|
|  4| 10|  8|  6|  9|  4|  5|
|  5|  9|  2|  7| 10|  7|  3|
|  6|  1|  1|  4|  2|  8|  4|
+---+---+---+---+---+---+---+
```

```
+---+--------------+-------------+--------------------------------------
----------------+
|id1|vec1          |vec2         |interactedCol
|
+---+--------------+-------------+--------------------------------------
----------------+
|1  |[1.0,2.0,3.0] |[8.0,4.0,5.0]
|[8.0,4.0,5.0,16.0,8.0,10.0,24.0,12.0,15.0]           |
|2  |[4.0,3.0,8.0] |[7.0,9.0,8.0]
|[56.0,72.0,64.0,42.0,54.0,48.0,112.0,144.0,128.0]    |
|3  |[6.0,1.0,9.0] |[2.0,3.0,6.0]
|[36.0,54.0,108.0,6.0,9.0,18.0,54.0,81.0,162.0]       |
|4  |[10.0,8.0,6.0]|[9.0,4.0,5.0]
|[360.0,160.0,200.0,288.0,128.0,160.0,216.0,96.0,120.0]|
|5  |[9.0,2.0,7.0]
|[10.0,7.0,3.0]|[450.0,315.0,135.0,100.0,70.0,30.0,350.0,245.0,105.0] |
|6  |[1.0,1.0,4.0] |[2.0,8.0,4.0]
|[12.0,48.0,24.0,12.0,48.0,24.0,48.0,192.0,96.0]      |
+---+--------------+-------------+--------------------------------------
----------------+
```

这个转换器目前只能在 Scala 中直接使用，但是可以使用 RFormula 从任何语言调用。建议用户使用 RFormula 而不是手动创建 Interaction。

3.5.3 多项式展开

多项式展开是将特征扩展到一个多项式空间的过程，它是由原始维度的 n 次组合来表示的。多项式展开用于生成所有输入列的交互变量。通过多项式展开，我们指定了想看到的各种相互作用的程度。例如，对于一个 degree-2 多项式，Spark 取特征向量中的每一个值，将其与特征向量中的其他每一个值相乘，然后将结果存储为特征。当希望查看特定特征之间的交互，但又不确定应该考虑哪些交互时，这种转换非常有用。

注意： 多项式展开可以极大地增加特征空间，导致高计算成本和过拟合。使用时要小心，特别是对于更高的 degrees。

下面的示例展示了如何将特征扩展到 3 次多项式空间。

```
// 导入相关包
import org.apache.spark.ml.feature.PolynomialExpansion
import org.apache.spark.ml.linalg.Vectors

// 创建 DataFrame
val data = Seq(
  Tuple1(Vectors.dense(2.0, 1.0)),
  Tuple1(Vectors.dense(0.0, 0.0)),
  Tuple1(Vectors.dense(3.0, -1.0))
)

val df = data.toDF("features")

df.printSchema()
df.show()

// 多项式展开
val polyExpansion = new PolynomialExpansion()
  .setInputCol("features")
```

```
  .setOutputCol("polyFeatures")
  .setDegree(3)

// 转换
val polyDF = polyExpansion.transform(df)

polyDF.show(false)
```

执行以上代码，输出结果如下：

```
root
 |-- features: vector (nullable = true)

+----------+
|  features|
+----------+
| [2.0,1.0]|
| [0.0,0.0]|
|[3.0,-1.0]|
+----------+

+----------+-----------------------------------------+
|features  |polyFeatures                             |
+----------+-----------------------------------------+
|[2.0,1.0] |[2.0,4.0,8.0,1.0,2.0,4.0,1.0,2.0,1.0]    |
|[0.0,0.0] |[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]    |
|[3.0,-1.0]|[3.0,9.0,27.0,-1.0,-3.0,-9.0,1.0,3.0,-1.0]|
+----------+-----------------------------------------+
```

3.5.4 RFormula 转换

RFormula 生成特征向量列和标签的 Double 列或 String 列，实现根据 R 模型公式拟合数据集所需的转换。

RFormula 选择由 R 模型公式指定的列。目前，它支持有限的 R 操作符（算子）子集，包括'~'、'.'、':'、'+'、'-'、'*'和'^'。基本算子如下：

（1）~　将目标和术语分开（即用来分开 label 标签列和其他列）。

（2）+　连接一项，“+ 0”表示删除截距。

（3）-　删除一项，“- 1”表示删除截距。

（4）:　交互（数值或二值化分类值的乘法）。

（5）.　除目标列外的所有列。

（6）*　因子交叉，包括术语和它们之间的交互。

（7）^　因子交叉到指定的 degree。

假设 a 和 b 是 double 列，我们用下面的简单例子来说明 RFormula 的效果：

（1）y ~ a + b，表示模型 y = w0 + w1 * a + w2 * b，其中 w0 为截距，w1、w2 为系数，y 是目标变量。

（2）y ~ a + b + a:b - 1，表示模型 y = w1 * a + w2 * b + w3 * a * b，其中 w1、w2、w3 为系数，y 是目标变量。

（3）y ~ a * b，表示模型 y = w0 + w1 * a + w2 * b + w3 * a * b，其中 w0 是截距，w1、w2、w3 为系数，y 是目标变量。

（4）y ~ (a + b)^2，表示模型 y = w0 + w1 * a + w2 * b + w3 * a * b，其中 w0 是截距，w1、w2、w3 为系数，y 是目标变量。

对于线性回归的转换原则：

（1）数值列将被转换为 Double，但不会是 one-hot 编码（注意这一点，这可能会保留隐含顺序。如果不想有隐含顺序，则要转换为 String）。

（2）字符串输入列，首先使用 StringIndexer 转换，使用 stringOrderType 确定排序，然后删除排序后的最后一个类别，然后对其进行 one-hot 编码。stringOrderType 可取的值：'frequencyDesc'、'frequencyAsc'、'alphabetDesc'、'alphabetAsc'。

（3）如果 label 列是 String 类型，则首先使用 frequencyDesc 排序，用 StringIndexer 将其转换为 Double。

（4）如果 DataFrame 中不存在 label 列，则将从公式中指定的响应变量创建输出标签（label）列。

请看下面使用 RFormula 的示例。

```
// 使用 RFormula estimator 来创建一个特征向量
import org.apache.spark.ml.feature.RFormula

val arrival_data = spark.createDataFrame(
   Seq(("SFO", "B737", 18, 95.1, "late"),
       ("SEA", "A319", 5, 65.7, "ontime"),
       ("LAX", "B747", 15, 31.5, "late"),
       ("ATL", "A319", 14, 40.5, "late") )
   ).toDF("origin", "model", "hour", "temperature", "arrival")

arrival_data.show

// 设置 label 标签列为"arrival",
// 特征列为剩余的所有列("origin", "model", "hour", "temperature")以及"hour"列和"temperature"列的乘积
val formula = new RFormula()
       .setFormula("arrival ~ . + hour:temperature")
       .setFeaturesCol("features")
       .setLabelCol("label")

// 首先调用 fit 函数，它返回一个模型(model，类型为 transformer),然后调用 transform
// val output = formula.fit(arrival_data).transform(arrival_data)
val model = formula.fit(arrival_data)   // estimator 拟合数据，返回的结果也是一个 Transformer
val output = model.transform(arrival_data)

// output.select("*").show(false)
output.show(false)
```

执行以上代码，输出结果如下：

```
+------+-----+----+-----------+-------+
|origin|model|hour|temperature|arrival|
+------+-----+----+-----------+-------+
|   SFO| B737|  18|       95.1|   late|
|   SEA| A319|   5|       65.7| ontime|
|   LAX| B747|  15|       31.5|   late|
|   ATL| A319|  14|       40.5|   late|
+------+-----+----+-----------+-------+
```

```
+------+-----+----+-----------+-------+-----------------------------------
---+-----+
|origin|model|hour|temperature|arrival|features
|label|
+------+-----+----+-----------+-------+-----------------------------------
---+-----+
|SFO   |B737 |18  |95.1       |late   |(8,[4,5,6,7],[1.0,18.0,95.1,1711.8])
|0.0  |
|SEA   |A319 |5   |65.7       |ontime |[0.0,0.0,1.0,1.0,0.0,5.0,65.7,328.5]
|1.0  |
|LAX   |B747 |15  |31.5       |late   |(8,[1,5,6,7],[1.0,15.0,31.5,472.5])
|0.0  |
|ATL   |A319 |14  |40.5       |late
|[1.0,0.0,0.0,1.0,0.0,14.0,40.5,567.0]|0.0  |
+------+-----+----+-----------+-------+-----------------------------------
---+-----+
```

3.5.5 特征装配

Spark 机器学习库中提供了一个 VectorAssembler Transformer 类，它是将多个列合并为一个向量列的特征转换器。

VectorAssembler Transformer 简单地将一组列合并成一个 vector 列。在机器学习术语中，这相当于将单个特征组合成单向量特征，以供机器学习算法学习。单个输入列的类型必须是下列类型之一：数值、布尔值或向量。输出向量列包含以指定顺序的所有列的值。

这个 Transformer 实际上几乎在每一个机器学习管道中都被使用，它的输出将被传递到一个 Estimator 中。请看下面这个使用 VectorAssembler Transformer 的示例。

```
// 导入相关类
import org.apache.spark.ml.feature.VectorAssembler

// 创建 DataFrame
val arrival_features = spark.createDataFrame(Seq(
       (18, 95.1, true),
       (5, 65.7, true),
       (15, 31.5,false),
       (14, 40.5, false)
   )).toDF("hour", "temperature","on_time")

// 使用 VectorAssembler transformer 来组合特征到一个 Vecotr 特征向量
val assembler = new VectorAssembler()
    .setInputCols(Array("hour","temperature", "on_time"))
    .setOutputCol("features")

// 转换
val output = assembler.transform(arrival_features)

output.show
```

执行以上代码，输出结果如下：

```
+----+-----------+-------+--------------+
|hour|temperature|on_time|      features|
+----+-----------+-------+--------------+
|  18|       95.1|   true|[18.0,95.1,1.0]|
```

```
|  5|       65.7|   true| [5.0,65.7,1.0]|
| 15|       31.5|  false|[15.0,31.5,0.0]|
| 14|       40.5|  false|[14.0,40.5,0.0]|
+----+-----------+-------+---------------+
```

3.6 特征选择

在构建预测模型时，选择合适的特征对模型的成功至关重要。使用太多的特征，特别是当它们与预测无关时，会增加模型的复杂性并导致过拟合。因此，应该优化输入变量的数量，以降低计算成本并提供高性能的模型。

特征选择是特征工程中的重要一环，其主要目的是从所有特征中选出相关特征，或者说在不引起重要信息丢失的前提下去除掉无关特征和冗余特征。

进行特征选择的好处主要有以下几种：

(1) 降低过拟合风险，提升模型效果。

(2) 提高训练速度，降低运算开销。

(3) 更少的特征通常意味着更好的可解释性。

既然特征选择的目的是去除无关特征，那么什么是无关特征？对于分类问题，在过滤式方法中一般假设与标签独立的特征为无关特征，而卡方检验恰好可以进行独立性检验，所以其适用于特征选择。如果检验结果是某个特征与标签独立，则可以去除该特征。

注意：卡方检验最基本的思想就是通过观察实际值与理论值的偏差来确定理论的正确与否。具体做的时候常常先假设两个变量确实是独立的（行话就叫做“原假设”），然后观察实际值（也可以叫做观察值）与理论值（这个理论值是指“如果两者确实独立”的情况下应该有的值）的偏差程度，如果偏差足够小，我们就认为误差是很自然的样本误差，是测量手段不够精确导致或者偶然发生的，两者确确实实是独立的，此时就接受原假设；如果偏差大到一定程度，使得这样的误差不太可能是偶然产生或者测量不精确所致，我们就认为两者实际上是相关的，即否定原假设，而接受备择假设。

3.6.1 卡方假设检验

Spark 的 spark.ml 包目前支持皮尔逊卡方（Pearson's Chi-squared）用于特征独立性检验。该检验主要用于检查两个分类变量在影响检验统计量方面是否独立。

ChiSquareTest 位于 org.apache.spark.ml.stat 包中，实现了对分类数据的卡方假设检验。它针对标签对每个特征进行 Pearson 独立性测试。对于每个特征，(特征、标签)对被转换成一个列联矩阵（contingency matrix），计算其卡方统计量。所有的标签和特征值必须是分类的。该方法返回一个 DataFrame，包含针对标签的每个特征的测试结果。下面是使用卡方假设检验的示例。

```
// 导入相关包
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.ml.stat.ChiSquareTest

// 构造 DataFrame
val data = Seq(
  (0.0, Vectors.dense(0.5, 10.0)),
  (0.0, Vectors.dense(1.5, 20.0)),
  (1.0, Vectors.dense(1.5, 30.0)),
  (0.0, Vectors.dense(3.5, 30.0)),
```

```
  (0.0, Vectors.dense(3.5, 40.0)),
  (1.0, Vectors.dense(3.5, 40.0))
)

val df = data.toDF("label", "features")

df.printSchema()
df.show()

// 卡方假设检验
val chi = ChiSquareTest.test(df, "features", "label")
chi.printSchema()
chi.show(false)

println(s"pValues = ${chi.first.getAs[Vector](0)}")
println(s"degreesOfFreedom = ${chi.first.getSeq[Int](1).toList}")
println(s"statistics = ${chi.first.getAs[Vector](2)}")
```

执行以上代码，输出结果如下所示：

```
root
 |-- label: double (nullable = false)
 |-- features: vector (nullable = true)

+-----+----------+
|label|  features|
+-----+----------+
|  0.0|[0.5,10.0]|
|  0.0|[1.5,20.0]|
|  1.0|[1.5,30.0]|
|  0.0|[3.5,30.0]|
|  0.0|[3.5,40.0]|
|  1.0|[3.5,40.0]|
+-----+----------+

root
 |-- pValues: vector (nullable = true)
 |-- degreesOfFreedom: array (nullable = true)
 |    |-- element: integer (containsNull = false)
 |-- statistics: vector (nullable = true)

+----------------------------------------+----------------+----------+
|pValues                                 |degreesOfFreedom|statistics|
+----------------------------------------+----------------+----------+
|[0.6872892787909721,0.6822703303362126]|  [2, 3]        |[0.75,1.5]|
+----------------------------------------+----------------+----------+
```

3.6.2 卡方选择器

Spark 有一些工具来实现在模型训练之前预先做一些粗略的过滤，比如 ChiSqSelector。

ChiSqSelector 使用独立性卡方检验来选择特征。它利用统计检验来识别与我们试图预测的标签不独立的特征，并去掉不相关的特征。它通常与分类数据一起使用，目的是减少将输入到模型中的特征数量，以及减少文本数据的维数（以频率或计数的形式）。

由于这种方法是基于卡方检验的，所以我们可以通过几种不同的方法来选择“最好的”特征。它支持五种选择方法：numTopFeatures，percentile，fpr，fdr，fwe。

(1) numTopFeatures：根据卡方检验选择固定数量的 top feature。这类似于生成具有最强预测能力的特征。它是按 p 值排序的（P 值即概率，反映某一事件发生的可能性大小）。

(2) percentile，类似于 numTopFeatures，但它选择所有特征的一部分，而不是一个固定的数量。

(3) fpr：选择所有 p 值低于阈值的特征，从而控制选择的假阳性率。

(4) fdr：使用 Benjamin-Hochberg 过程选择错误发现率低于阈值的所有特征。

(5) fwe：选择 p 值低于阈值的所有特征。阈值按 1/numFeatures 进行缩放，从而控制了选择的 family-wise 错误率。

默认情况下，选择方法是 numTopFeatures，将 top features 的默认数量设置为 50。用户可以使用 setSelectorType 选择一个选择方法。

假设我们有一个 DataFrame，包含 id、features 和 click 三列，其中列 clicked 是要预测的目标（即 label 列）：

```
Id   |      features          | clicked
----|-----------------------|---------
 7   | [0.0, 0.0, 18.0, 1.0] | 1.0
 8   | [0.0, 1.0, 12.0, 0.0] | 0.0
 9   | [1.0, 0.0, 15.0, 0.1] | 0.0
```

如果使用带有 numTopFeatures = 1 的 ChiSqSelector，那么根据标签"clicked"，"features"中的第三列被选择为最有用的特征:

```
+---+---------------------+----------+----------------------+
|id |       features      | clicked  |   selectedFeatures   |
+---+---------------------+----------+----------------------+
|  7|    [0.0,0.0,18.0,1.0]|       1.0|                [18.0]|
|  8|    [0.0,1.0,12.0,0.0]|       0.0|                [12.0]|
|  9|    [1.0,0.0,15.0,0.1]|       0.0|                [15.0]|
+---+---------------------+----------+----------------------+
```

示例代码如下：

```
// 首先，引入卡方选择器所需要使用的类
import org.apache.spark.ml.feature.ChiSqSelector
import org.apache.spark.ml.feature.ChiSqSelectorModel
import org.apache.spark.ml.linalg.Vectors

// 构造 DataFrame，我们将在此数据集上进行卡方选择
// 这是一个具有三个样本，四个特征维度的数据集，标签有 1，0 两种
val data = Seq(
  (1, Vectors.dense(0.0, 0.0, 18.0, 1.0), 1.0),
  (2, Vectors.dense(0.0, 1.0, 12.0, 0.0), 0.0),
  (3, Vectors.dense(1.0, 0.0, 15.0, 0.1), 0.0)
)

val df = data.toDF("id", "features", "label")

df.printSchema()
df.show()

// 现在，用卡方选择进行特征选择器的训练
val selector = new ChiSqSelector()
  .setNumTopFeatures(1)    // 为了观察更明显，设置只选择和标签关联性最强的一个特征
```

```
  .setFeaturesCol("features")   // 输入特征向量列
  .setLabelCol("label")         // 标签列
  .setOutputCol("selectedFeatures") // 输出特征向量列

val result = selector.fit(df).transform(df)

// 输出
println(s"top ${selector.getNumTopFeatures} 个特征被选取。")
result.show(false)
```

执行以上代码，输出结果如下：

```
root
 |-- id: integer (nullable = false)
 |-- features: vector (nullable = true)
 |-- label: double (nullable = false)

+---+------------------+-----+
| id|         features|label|
+---+------------------+-----+
|  1|[0.0,0.0,18.0,1.0]|  1.0|
|  2|[0.0,1.0,12.0,0.0]|  0.0|
|  3|[1.0,0.0,15.0,0.1]|  0.0|
+---+------------------+-----+

top 1 个特征被选取。
+---+------------------+-----+----------------+
|id |features          |label|selectedFeatures|
+---+------------------+-----+----------------+
|1  |[0.0,0.0,18.0,1.0]| 1.0  | [18.0]         |
|2  |[0.0,1.0,12.0,0.0]| 0.0  | [12.0]         |
|3  |[1.0,0.0,15.0,0.1]| 0.0  | [15.0]         |
+---+------------------+-----+----------------+
```

不过，从 Spark 3.1.1 开始，ChiSqSelector 类已被标记为弃用状态，官方建议改用另一个新的特征选择器 UnivariateFeatureSelector 类。

3.6.3 单变量特征选择器

从 Spark 3.1.1 开始，Spark 决定不再使用之前版本中单独的选择器类（例如，ChiSqSelector，ANOVASelector 等），而是将它们集中在一个名为 UnivariateFeatureSelector 的类中。它是基于标签单变量统计测试的特征选择器。单变量特征选择是根据响应变量单独评估每个特征以确定它们之间关系的过程。

UnivariateFeatureSelector 类接受用户的选择模式和标准，并在内部应用适当的测试。目前，Spark 支持三种单变量特征选择器：chi-squard（卡方）、ANOVA F-test（方差分析 f 检验）和 F-value（f 值）。UnivariateFeatureSelector 对具有分类/连续特征的分类/连续标签进行操作。给定 featureType 和 labelType 信息，Spark 然后根据这些特定类型选择评分函数。

支持表 3-1 所示的 featureType 和 labelType 组合：

表 3-1 消息格式

featureType	labelType	评分函数
categorical	categorical	chi-squared（chi2）
continuous	categorical	ANOVA F-test（f_classif）

continuous	continuous	F-value（f_regression）

UnivariateFeatureSelector 支持五种选择模式：numTopFeatures、percentile、fpr、fdr、fwe。

(1) numTopFeatures：根据假设选择固定数量的 top feature。这也是默认选择模式。选择 setSelectionThreshold 参数给出的前 N 个特征。该参数的默认值为 50。

(2) percentile：类似于 numTopFeatures，但不是选择固定数量的特征，而是通过给定的百分比（默认情况下，前 10%）选择其中的一部分。由于我们对每个客户的模型有不同数量的输入参数，因此与 numTopFeatures 相比，该方法更具动态性。

(3) fpr：假阳性率（False Positive Rate）。该模式选择 p 值低于某一阈值的所有特征，从而控制了选择的误报率。默认情况下，该阈值为 0.05。因此，通过选择< 0.05，有强有力的证据可以排除零假设。当可能存在关系时，我们选择特征。

(4) fdr：假阳性发现率（False Discovery Rate）。该模式遵循与 fpr 相似的方法，但在选择进行切割的特征之前，使用一种特殊的方法来调整 p 值分布。这种方法被称为 Benjamini-Hochberg 过程。这样调整的原因是为了减少低于 p 值的假阳性数。

(5) fwe：Familywise Error Rate。通过使用多重假设检验，我们总是有可能在不应该拒绝零假设的情况下拒绝零假设。从统计学上讲，100 次测试中有 5 次我们做出了错误的决定。这种模式是解决这一挑战的一种方式。这种模式在选择特征时，它不收集具有”p 值 < 阈值”的特征，而是收集具有“pval < threshold / nummofeatures”的特征。即先对阈值按 1/numFeatures 进行缩放，再选择 p 值低于缩放后阈值的所有特征，从而控制了选择的 family-wise 错误率。

下面的示例中，使用 UnivariateFeatureSelector 选择一个最重要的特征。

```
// 首先，导入所需要使用的类
import org.apache.spark.ml.feature.UnivariateFeatureSelector
import org.apache.spark.ml.feature.UnivariateFeatureSelectorModel
import org.apache.spark.ml.linalg.Vectors

// 构造 DataFrame，我们将在此数据集上进行卡方选择
// 这是一个具有三个样本，四个特征维度的数据集，标签有 1，0 两种
val data = Seq(
  (1, Vectors.dense(0.0, 0.0, 18.0, 1.0), 1.0),
  (2, Vectors.dense(0.0, 1.0, 12.0, 0.0), 0.0),
  (3, Vectors.dense(1.0, 0.0, 15.0, 0.1), 0.0)
)

val df = data.toDF("id", "features", "label")

df.printSchema()
df.show()

// 现在，用单变量特征选择选择进行特征选择器的训练
val selector = new UnivariateFeatureSelector()
  .setFeatureType("categorical")
  .setLabelType("categorical")
  .setSelectionMode("numTopFeatures")  // 选择模式
  .setSelectionThreshold(1)            // 设置只选择和标签关联性最强的一个特征
  .setFeaturesCol("features")          // 输入特征向量列
  .setLabelCol("label")                // 标签列
  .setOutputCol("selectedFeatures")    // 输出特征向量列
```

```
val result = selector.fit(df).transform(df)

// 输出
result.show(false)
```

执行以上代码，输出内容如下：

```
root
 |-- id: integer (nullable = false)
 |-- features: vector (nullable = true)
 |-- label: double (nullable = false)

+---+------------------+-----+
| id|          features|label|
+---+------------------+-----+
|  1|[0.0,0.0,18.0,1.0]|  1.0|
|  2|[0.0,1.0,12.0,0.0]|  0.0|
|  3|[1.0,0.0,15.0,0.1]|  0.0|
+---+------------------+-----+

+---+------------------+-----+----------------+
|id |features          |label|selectedFeatures|
+---+------------------+-----+----------------+
|1  |[0.0,0.0,18.0,1.0]| 1.0 | [18.0]         |
|2  |[0.0,1.0,12.0,0.0]| 0.0 | [12.0]         |
|3  |[1.0,0.0,15.0,0.1]| 0.0 | [15.0]         |
+---+------------------+-----+----------------+
```

可以尝试修改代码中 featureType 和 labelType 的参数值，并观察的选择结果。

第 4 章 Spark ML 分类算法基础

分类（classification）是一种监督机器学习方法，其中模型试图预测给定输入数据的正确标签。在分类中，使用训练数据对模型进行充分训练，然后在测试数据上对模型进行评估，然后用于对新的未知数据进行预测。

在本章中，我们将首先定义什么是机器学习中的分类，然后澄清机器学习中的两种类型的学习器以及分类和回归之间的区别。然后，我们将介绍一些可以使用分类的真实应用场景。

4.1 分类任务概述

分类是有监督机器学习算法的一个子集，目标变量是一个分类变量。分类的任务是将输入样本分类为几个类别。分类算法可以学习预测给定的电子邮件是垃圾邮件（Spam）还是 Ham（无垃圾邮件），如下图所示。

分类是最广泛被研究和使用的机器学习任务之一，因为它能够帮助解决许多现实生活中的与分类相关的问题。例如，在信用卡交易记录中，判断这是一种欺诈性的信用卡交易吗？根据病人的症状，判断病人是否串有某类疾病？给出一个照片，判断这是猫、狗或鸟的图像吗？

4.1.1 懒惰的学习者 Vs. 渴望的学习者

在机器学习分类中有两种类型的学习器：懒惰学习器和渴望学习器。

渴望学习器是这样一种机器学习算法：它首先从训练数据集中建立一个模型，然后对未来的数据集进行任何预测。他们在训练过程中花费了更多的时间，因为他们渴望在训练过程中通过学习权重来获得更好的泛化，但他们需要更少的时间来进行预测。

大多数机器学习算法都是渴望学习的，例如：

（1）Logistic 回归算法。

（2）支持向量机算法。

（3）决策树算法。

（4）人工神经网络算法。

另一方面，惰性学习器或基于实例的学习器不会立即从训练数据创建任何模型，这就是惰性方面的来源。他们只是记住训练数据，每次需要进行预测时，他们从整个训练数据中寻找最近的邻居，这使得他们在预测过程中非常慢。这样的算法有：

（1）KNN 近邻算法。

（2）基于案例的推理。

Spark 机器学习库中实现了大多数的渴望学习器算法，但是由于大数据分布式计算的原因，未实现类似 KNN 这样的懒惰学习器算法。

4.1.2 不同类型的分类任务

机器学习中有三种主要的分类任务：二分类、多类分类、多标签分类。

1. 二分类（Binary Classification）

在二分类任务中，目标是将输入数据分为两个相互排斥的类别。例如，交易是欺诈或不欺诈，会议论文被接受或不接受，肿瘤是良性的或恶性的。这种情况下的训练数据被标记为二元格式：true 和 false，积极和消极，0 和 1，垃圾邮件和非垃圾邮件等等，这取决于要解决的问题。二分类如图 4-所示：

2. 多类分类（Multi-Class Classification）

多类分类至少有两个相互排斥的类标签，其目标是预测给定输入示例属于哪个类。例如，例如，图像是狗、猫还是鸟，手写字母识别等。图 4-表示的是三分类的情况。

3. 多标签分类（Multi-Label Classification）

在多标签分类任务中，我们尝试为每个输入样例预测 0 个或更多的类。在这种情况下，没有互斥，因为输入样例可以有多个标签。

这样的场景可以在不同的领域中观察到，例如自然语言处理中的自动标记，其中给定的文本可以包含多个主题。另外，电影类型就是一个很好的例子，例如，一部电影既可以标记为动作片，也可以标记为喜剧片。

多类分类和多标签分类的区别：

不可能使用多类或二分类模型来执行多标签分类。Spark 机器学习库并不支持这种类型的分类。

Spark ML 机器学习库为分类任务提供了一些机器学习算法的分布式实现，这些算法实现类都位于 org.apache.spark.ml.classification 包中，其中包括：

（1）Logistic 回归算法（Logistic regression）。

（2）决策树算法（Decision tree）。

（3）随机森林算法（Random forest）。

（4）梯度提升树算法（Gradient-boosted tree）。

（5）线性支持向量机算法（Linear support vector machine）。

（6）One versus rest 算法。

（7）朴素贝叶斯算法（Naïve Bayes）。

Logistic 回归是预测分类的一种常用方法。它是预测结果概率的广义线性模型的一个特例。在 spark.ml 机器学习库中，Logistic 回归可以用二元 Logistic 回归预测二分类结果，也可以用多项 Logistic 回归预测多类结果。使用 family 参数在这两种算法之间进行选择，或者不设置它，这时 Spark 将推断出正确的变体。

4.2 Logistic 回归实现二分类任务

Logistic 回归（逻辑回归）是一种很常见的用于解决分类问题的监督学习算法，它主要是通过寻找最优参数来正确地分类原始数据。在分类问题中，我们有二进制或离散格式的因

变量，如 0 或 1。目前，Logistic 回归在流行病学、实验研究、临床试验评价及疾病的预后因素分析等方面均有广泛应用。

4.2.1 基本原理

逻辑回归（Logistic Regression）是一种预测概率的线性分类器，以其易用性和快速的训练速度而受到欢迎，经常被用于二分类和多类分类。

逻辑回归算法是一种基于概率的预测分析算法，它适用于分类变量，如 0 或 1、是或否、真或假、垃圾邮件或非垃圾邮件等。

当数据具有清晰的决策边界时，可以使用 Logistic 回归线性分类器，如下图所示：

需要注意的是，虽然其名字包含回归二字，但本质上是一种分类学习方法。这是因为，Logistic 回归假设输入和目标变量之间存在线性关系，线性回归方程可以找到输入变量和目标变量之间的关系，从而找出截距和系数的最优值来捕捉这种关系，然后用线性回归模型的预测结果取逼近真实标记的对数几率。因此，其对应的模型被称为 "对数几率回归"（Logistic Regression），这条曲线也被称为逻辑曲线（Logistic Curve）。

逻辑回归和线性回归的主要区别在于，逻辑回归使用 sigmoid 非线性函数将其预测输出转换为可映射到两个（二项）类的概率值，并将其限制在 0 和 1 之间（Sigmoid 函数的输出范围是 0 到 1）。使用 sigmoid 函数的优点是，无论输入值是多少，输出总是转换的概率，通常用在逻辑回归场景中将预测概率作为输出来确定类发生的概率。

注意： *Sigmoid 函数也被称为 Logistic 函数，因为它最初是与 Logistic 回归算法一起引入的。*

Sigmoid 函数可以表示为：

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

其中，f(x)=函数输出（0 和 1 值），x=函数输入，e=自然对数的底。

当我们向函数提供输入值(数据)时，它给出如下的 s 曲线。它使用阈值水平的概念，高于阈值水平的值四舍五入到 1，低于阈值水平的值四舍五入到 0。如图 6-所示：

Logistic 回归在许多业务应用中得到了广泛的应用，适用于各项广义上的分类任务，例如：预测用户是否会购买产品（二分类）、评论信息的正负情感分析（二分类）、用户点击率（二分类）、用户违约信息预测（二分类）、垃圾邮件预测（二分类）、疾病预测（二分类）、用户等级分类（多分类）等场景。在这些情况下，模型将为每个样本返回一个概率。

4.2.2 示例：客户购买预测

在 Spark 机器学习库中，提供了逻辑回归算法的分布式实现类 LogisticRegression。这一节，我们使用 LogisticRegression 来构建逻辑回归模型，并预测目标类标签。

这个示例中使用的数据集是一个虚拟数据集，此数据集包含有关某虚拟体育零售商品网站在线用户的信息。这些数据包括用户的国家、使用的购物渠道、年龄、是否是回头客，以及网站上浏览的网页数量，另外它还包含客户最终是否购买产品的信息，如表 4-1 所示。

表 4-1 零售商品网站在线用户数据说明

序号	字段	数据类型	字段含义
1	country	字符串	客户所在国家
2	age	整数	客户的年龄
3	is_returned	整数	是否是回头客。0-不是回头客，1-是回头客
4	channel	字符串	客户使用的搜索引擎
5	page_views	整数	客户在网站上浏览的网页数量
6	is_purchase	整数	客户是否购买了产品。0-没有购买，1-购买

从上表中可知，其中的 country 和 channel 字段是分类变量，需要在特征工程中对其进行转换和编码。因为我们是预测一个客户是否会购买商品，因此 is_purchase 字段是我们的目标字段（也是训练集中的标签列）。

【示例】使用 Spark 机器学习库的逻辑回归算法实现类 LogisticRegression 来构建逻辑回归模型，并预测客户购买意向。

1）读取数据集

首先，加载数据集，代码如下：

```
// 因为是测试，所以使用了本地路径
val filePath = "file:///home/hduser/data/ml/purchase data.csv"

// 读取到 DataFrame 中
val df = spark.read
    .option("header",true)
    .option("inferSchema",true)
    .csv(filePath)
```

2）探索性数据分析

接下来，执行探索性分析。先查看数据模式，代码如下：

```
// 查看数据模式
df.printSchema()
```

执行以上代码，输出结果如下：

```
root
 |-- country: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- is returned: integer (nullable = true)
 |-- channel: string (nullable = true)
 |-- page_views: integer (nullable = true)
 |-- is_purchase: integer (nullable = true)
```

再取前 10 条数据看看，以便对数据有一个基本的认识，代码如下：

```
// 取前 10 条数据看看
df.show(10)
```

执行以上代码，输出结果如图 4-所示。

从以上两个输出内容可以看出，该数据集中的 country 和 channel 两列是自然分类列，而要做为特征来使用，需要先转换为数值形式才可以。

再查看数据集的规模，有多少行，多少列。代码如下：

```
// 数据有多少行多少列
println("数据集记录数量: " + df.count())
println("数据集字段数量: " + df.columns.length)
```

执行以上代码，输出结果如下：

```
数据集记录数量: 20000
数据集字段数量: 6
```

从以上输出内容可以看出，数据集总共包含 20,000 行和 6 列。结合前面的探索可知，其中前 5 列是特征列，最后 1 列 is_purchase 是标签列。

接下来执行描述性统计方法，查看数据集的统计度量情况，代码如下：

```
df.describe().show()
```

执行以上代码，输出内容如图 4-所示。

从以上输出内容可以观察到，客户的平均年龄在 28 岁和 29 岁左右，他们在访问网站时平均大约浏览 9 到 10 个网页。

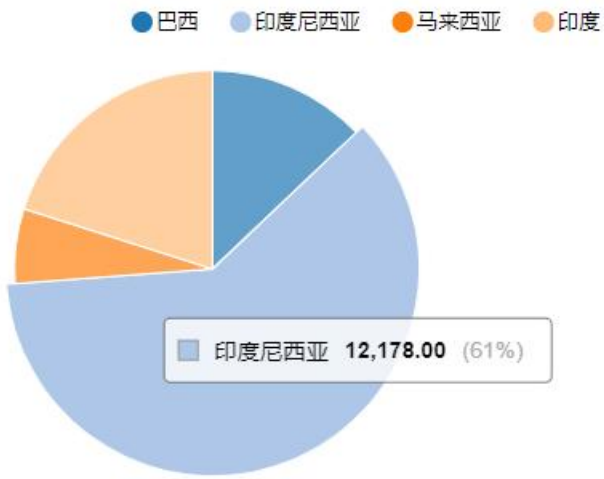
下面从不同的维度来看一些统计值。首先将 df 注册为一个临时视图，代码如下：

```
// 将 df 注册为一个临时视图 purchase_tb
df.createOrReplaceTempView("purchase_tb")
```

统计不同国家的访问人数，执行如下 SQL 语句：

```

```

执行结果如图 4-所示：

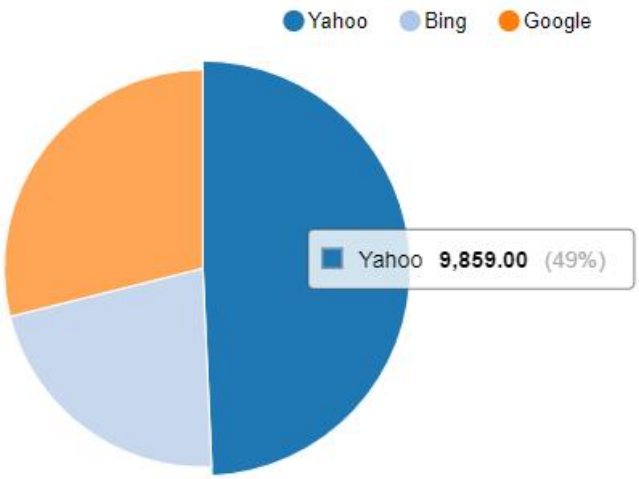


从结果图 4-中可以看到，超过 60 的客户来自印尼。

统计使用不同搜索引擎的人数，执行如下 SQL 语句：

```

```

执行结果如图 4-所示：



从结果图 4-中可以看到，将近一半的客户使用的搜索引擎是 Yahoo。

统计所有客户中，有多少转化为了商品购买者。执行如下 SQL 语句：

```

```

执行结果如图 4-所示：

is_purchase	购买人数
1	10000
0	10000

3）特征工程

在上一步的探索性数据分析中可知，数据集中的 country 和 channel 两列属于分类变量，需要将它们转换为数值形式，然后创建特征向量，作为 LogisticRegression 逻辑回归类的特征输入参数。

首先，导入特征转换所需的依赖包，代码如下：

```
import org.apache.spark.ml.feature.StringIndexer
import org.apache.spark.ml.feature.OneHotEncoder
import org.apache.spark.ml.feature.VectorAssembler
```

然后对分类变量 country 和 channel 两列执行转换，代码如下：

```
// ......
```

执行以上代码，输出结果如图 4-所示：

接下来一步，对 country_idx 和 channel_idx 两列进行 one-hot 编码，将它们表示为一个 one-hot 编码向量的形式（表示值及值在向量中的位置）。代码如下：

```
// one-hot 编码
......
```

执行以上代码，输出结果如图 4-所示：

然后，创建特征向量列 features，代码如下：

```
// ......
```

最后，仅保留特征向量列 features 和标签列 is_purchase（训练逻辑回归模型仅需要这两个列）。代码如下：

```
// ......
```

执行以上代码，输出内容如图 4-所示：

4）分割数据集

为了避免过拟合，在执行有监督的机器学习时，常见的方法是将已有数据集中保留一部分数据作为测试集，用来评估模型的训练效果。这也是在机器学习建模过程中通行的做法：将数据分为训练集和测试集。测试集是与训练独立的数据，完全不参与训练，用于最终模型的评估。

将上一步经过特征工程后的数据集按 75/25 的比例，分割为训练集和测试集，这可能通过调用 randomSplit()方法来实现。代码如下：

```
// ......
```

执行以上代码，输出内容如下：

```
训练集记录数量: 15102
测试集记录数量: 4898
总记录数量: 20000
```

要确保拆分后的训练集和测试集中目标类的平衡（即 is_purchase 列中购买者和未购买都的比例在训练集和测试集中要有相似的比例，以保证平衡），这可以通过在训练集和测试集中分别统计来检验。代码如下：

```
// ......
```

执行以上代码，输出结果如图 4-所示。

可以看出，不管是在拆分后的训练集中，还是在测试集中，目标类都是平衡的。

5）构建和训练逻辑回归模型

LogisticRegression 是一个 Estimator，它拟合一个训练数据集来创建一个训练过的模型。然后可以使用该模型转换测试数据集。LogisticRegression Estimator 默认需要一个名为"features"的特征列和一个名为"label"的标签列（目标列），使用 features 作为输入特征列（默认输入特征列名就是 features，因此也可以省略），使用 is_purchase 作为标签列，构建和训练逻辑回归模型。代码如下：

```
// ......
```

执行以上代码，观察到交互输出的信息如图 4-所示：

可以看到，在执行 fit()方法拟合数据后，返回的是 LogisticRegressionModel 类，它代表一个训练后的逻辑回归模型类。

6）将模型应用到测试集上

使用以上训练的模型对测试集进行预测，这需要使用 LogisticRegressionModel 的 transform()方法。代码如下：

```
// ......
```

执行以上方法，输出结果如图 4-所示：

LogisticRegressionModel 的 transform()方法，通过从 featuresCol 中读取数据集，并根据参数添加新列来转换数据集：

（1）prediction 列：预测的标签，类型为 Double。

（2）rawPrediction 列：原始预测(概率，反映了对某一类别的信心)，类型为 Vector。

（3）probability 列：每个类别的概率，类型为 Vector。

对比原始标签列 is_purchase 和预测目标列 prediction，可以看出，有的预测结果和原始标签是一致的，而也有的预测结果和原始标签是不一致的（即预测错误）。

4.3 Logistic 回归实现多分类任务

逻辑回归分类器是机器学习领域著名的分类模型。其常用于解决二分类问题。但是在工作/学习/项目中，我们经常要解决多分类问题。多分类任务意味着一个分类器将输入示例分为几个类别。

许多机器学习算法可用于训练多类分类器，但有些算法，如感知机、逻辑回归和支持向量机等算法是为二值分类而设计的，它们本身不支持超过两个类的分类任务。然而，在面临多类分类任务时，可以使用一些策略在多类别分类问题上使用二元分类器进行分类。

4.3.1 one-vs-rest 和 one-vs-one 策略

将二元分类算法用于多分类问题的一种方法是将多类分类数据集拆分为多个二元分类数据集，并在每个数据集上拟合一个二元分类模型。这种方法的两个不同的例子是 One-vs-One 策略和 One-vs-Rest 策略。

假设我们要解决一个分类问题，该分类问题有三个类别，分别用△、□和×表示。数据集中的每个样本有两个属性，如果把属性 1 作为 X 轴，属性 2 作为 Y 轴，数据集的分布可以表示为图 4-：

下面分别了解 One-vs-One 策略和 One-vs-Rest 策略下的解决过程。

1. One-vs-One (OvO)

One-vs-One（简称 OvO）是利用二元分类算法进行多类分类的一种启发式方法。

One-vs-One 将多类分类数据集分解为二分类问题。One-vs-One 方法将数据集分为每个类的一个数据集，然后让不同类别的数据两两组合训练分类器，因此对于三分类问题，可以得到 3 个二元分类器，如图 4-所示：

假如我们要预测的一个数据在图 4-中红色圆圈的位置，那么第一个分类器会认为它是 x，第二个分类器会认为它偏向三角形，第三个分类器会认为它是 x，经过三个分类器的投票之后，可以预测红色圆圈所代表的数据的类别为 x。

任何一个测试样本都可以通过分类器的投票选举出预测结果，这就是 One-Vs-One 的运行方式。

当然这一方法也有显著的优缺点。其缺点是训练出更多的分类器，会影响预测时间。其优点是，它在一定程度上规避了数据集不平衡的情况，性能相对稳定，并且需要训练的模型数虽然增多，但是每次训练时训练集的数量都降低很多，其训练效率会提高。

2. One-vs-Rest (OvR)

One-Vs-Rest 策略的思想是把一个多分类的问题变成多个二分类的问题。转变的思路就如同方法名称描述的那样，选择其中一个类别为正类（Positive），使其他所有类别为负类（Negative）。其过程如下：

（1）第一步，可以将△所代表的样例全部视为正类，其他样例全部视为负类，得到第一个分类器。

（2）第二步，把×所代表的样例全部视为正类，其他样例全部视为负类，可以得到第二个分类器。

（3）第三步，把□所代表的样例全部视为正类，其他样例全部视为负类，可以得到第三个分类器。

如图 4-所示：

对于一个三分类问题，最终得到 3 个二元分类器。在预测阶段，每个分类器模型可以根据测试样本，预测一个类成员概率或类似概率的分数，然后使用这些分数的 argmax（分数最大的类索引）来预测一个类。

One-Vs-Rest 是一种最为常用的二分类拓展方法，其优缺点也十分明显。其优点是普适性还比较广，可以应用于能输出值或者概率的分类器，同时效率相对较好，有多少个类别就训练多少个分类器。其缺点是很容易造成训练集样本数量的不平衡，尤其在类别较多的情况下，经常容易出现正类样本的数量远远不及负类样本的数量，这样就会造成分类器的偏向性。

3. Spark One-vs-Rest 分类器

假设对于有 K 个类别数据的多分类问题：

（1）OvR 策略总共会训练 K 个二元分类器。需要为每一个类别分别建立一个唯一的二分类基分类器，属于此类的所有样本均为正例，其余的全部为负例。

（2）OvO 策略则会训练 K (K － 1) / 2 个二分类分类器。每一个二分类分类器从初始多分类训练集中收集其中两个类别的所有样本，并学习去区分这两个类别。在预测时，会有一个投票：所有 K (K － 1) / 2 个二分类分类器被应用于一个未知样本，并且那个得到最多正例（Positive）预测的类别会成为最终的多分类预测结果。

尽管 One-vs-Rest 方法不能处理多个数据集，但它训练的分类器数量较少，使其成为更快的选择，通常是首选。这种方法通常用于自然预测数字类成员概率或分数的算法，因此是逻辑回归首选的策略。

Spark 机器学习库中提供了 OneVsRest 类，为了进行多类分类任务，它需要给出一个能有效进行二分类的基分类器（当然是 LogisticRegression 分类器）。OneVsRest 产生一个 OneVsRestModel，可以用于数据集转换。

OneVsRest 被实现为一个 Estimator。对于基分类器，它接受 Classifier 类的实例，并为 k 个类中的每一个类创建一个二元分类问题。类 i 的分类器被训练来预测标签是否为 i，从而将类 i 与所有其他类区分开来。

预测是通过评估每个二元分类器来完成的，最自信的分类器的索引作为标签输出。

4.3.2 示例：鸢尾花分类预测

下面的示例演示了如何加载 Iris 数据集，将其解析为 DataFrame，并使用 OneVsRest 执行多类分类。通过计算测试误差来衡量算法的精度。

【示例】使用 Spark ML 中的 Logistic 回归算法实现对鸢尾花数据集进行分类。

在下面这个示例中，我们将展示如何在使用被称为 one-vs-rest 的策略来利用 Logistic 回归（二分类模型）执行多分类的任务。我们的目标是根据一组特征（即花萼长度、花萼宽度、花瓣长度、花瓣宽度 4 个属性）来预测鸢尾花卉属于 Setosa（山鸢尾）、Versicolour（杂色鸢尾）、Virginica（维吉尼亚鸢尾）三个种类中的哪一类。

1）理解鸢尾花数据集

鸢尾花数据集共收集了三类鸢尾花，即 Iris Setosa（山鸢尾花）、Iris Versicolour（变色鸢尾花）和 Iris Virginica（维吉尼亚鸢尾花），植物学家 Edgar Anderson 测量了这些鸢尾花的花萼长度、花萼宽度、花瓣长度、花瓣宽度，并标注了每个鸢尾花所属的类别，如图 4-所示（其中 petal 表示花瓣，sepal 表示花萼）。

在鸢尾花数据集中，每一类鸢尾花收集了 50 条样本记录，共计 150 条。它共有 4 个属性列，分别是：sepal length（萼片长度）、sepal width（萼片宽度）、petal length（花瓣长度）、petal width （花瓣宽度），单位都是厘米。最后一列是品种类别列，其值为通过前面四列所确定的鸢尾花所属的类别名称，分别是 Setosa、Versicolour 和 Virginica。如图 4-所示：

鸢尾花数据集的字段说明如表 4-所示。

表 4-1 鸢尾花数据集字段说明

序号	字段	数据类型	字段说明
1	SepalLength	float	花萼长度
2	SepalWidth	float	花萼宽度
3	PetalLength	float	花瓣长度
4	PetalWidth	float	花瓣宽度
5	Class	int	标签列，类别变量。其中，0 代表 Iris Setosa，1 代表 Iris Versicolor，2 代表 Iris Virginica

2）读取数据集

首先，加载数据集，代码如下：

```
import org.apache.spark.sql.types.

// ......
```

3）探索性数据分析

接下来，执行探索性分析。先查看数据规模和数据模式，代码如下：

```
// ......
```

执行以上代码，输出内容如下：

```

```

从输出内容可以看出，该数据集的四个特征属性的取值都是数值型的，它们具有相同的量纲，因此不需要做任何标准化的处理。

随机抽取 10%数据进行查看数据内容，代码如下：

```
dataDF.sample(false, 0.1).show()
```

执行以上代码，输出内容如下：

```
// ......
```

看看整个数据集中，每种鸢尾花类型的样本各有多少，代码如下：

```
dataDF.groupBy("species").count().show
```

执行以上代码，输出内容如下：

```

```

获取数据的描述性统计数据,这有助于了解数据的分布，代码如下：

```
dataDF.describe().show(5)
```

执行以上代码，输出内容如图 4-所示：

4）特征工程

在上一步的探索性数据分析中可知，数据集中的标签列 species 目前类型为 string，属于分类变量，需要将其转换为数值形式。

首先，导入特征转换所需的依赖包，代码如下：

```
import org.apache.spark.ml.feature.StringIndexer
import org.apache.spark.ml.feature.VectorAssembler
```

使用 StringIndexer 将 species 列编码为 double 类型，并存储到名为 label 的新列中，代码如下：

```
// ......
```

执行以上代码，输出内容如下：

```

```

使用 VectorAssembler（它是一个 transformer）将这些特征合并到单个的特征向量列，代码如下：

```
// 定义特征列
// ......
```

执行以上代码，输出内容如图 4-所示：

用皮尔逊相关来测量各个特征和 label 目标列之间的统计相关性，代码如下：

```
// ......
```

执行以上代码，输出内容如下：

```
0.9490425448523336
0.9564638238016178
0.7825612318100821
-0.41944620026002677
```

从以上输出内容可以看出，前 3 个属性与目标列高度线性相关，但第 4 个属性 sepal_width 与目标列似乎为低度负线性相关。

5）分割数据集

将上一步经过特征工程后的数据集按 80/20 的比例，分割为训练集和测试集，这可能通过调用 randomSplit()方法来实现。代码如下：

```
// 分割数据集（其中训练集 80%，测试集 20%）
// ......
```

执行以上代码，输出内容如下：

```
训练集数量: 113
测试集数量: 37
```

查看数据分布是否均衡，代码如下：

```
// ......
```

执行以上代码，输出内容如下：

```
// ......
```

从统计结果看，各类别数据在训练集和测试集中是均衡分布的。

6）构建和训练 OneVsRest 分类器

接下来，使用 OneVsRest 类在训练数据集上拟合一个模型，并指定基分类器为 LogisticRegression。代码如下：

```
// ......
```

执行以上代码。

7）在测试集上测试模型

使用上一步训练出来的 ovrModel 对测试集进行预测，代码如下：

```
// ......
```

执行以上代码，输出内容如下：

```
// ......
```

从以上输出结果可以看出，使用 ovrModel 模型对测试集进行转换后，输出结果集中增加了两个新列：rawPrediction 和 prediction。其中 prediction 列就是模型对测试集中每个样本的预测类别。选取 4 个特征列以及 label 列、prediction 列查看，代码如下：

```
predictions
    .drop("features", "rawPrediction")
    .show()
```

执行以上代码，输出内容如图 4-所示：

对比上面的输出内容，会发现有些预测结果和实际类别是不一致的，说明这些是预测错误的项。检查 rawPrediction 和 prediction 列，代码如下：

```
predictions.select("rawPrediction","prediction").show(false)
```

执行以上代码，输出内容如下：

```

```

其中 rawPrediction 列中每个数组值是原始预测结果，反映了该样本属于索引对应的类的信心，最后选其中最大值对应的类为预测结果。

4. 4 多项 Logistic 回归实现多分类任务

多项逻辑回归（Multinomial Logistic Regression），也叫做多元 Logistic 回归或多类别 Logistic 回归。在统计学里，它是一个将逻辑回归一般化成多类别问题得到的分类方法。用更加专业的话来说，它就是一个用来预测一个具有类别分布的因变量不同可能结果的概率的模型。

4. 4. 1 Softmax 函数

Sigmoid 和 SoftMax 函数定义了机器学习中使用的激活函数，更具体地说，是用于分类方法的激活函数。两个函数都从实数 R 的范围内取一个值 X，并输出一个介于 0 到 1 之间的数字，表示 X 属于某个类的概率。

上图表明，Sigmoid 函数和 SoftMax 函数的输出，是在给定输入 X 的情况下 Y 是 k 类的概率。

Sigmoid 用于二元分类方法，而 SoftMax 适用于多类问题。实际上，SoftMax 函数是 Sigmoid 函数的扩展。Softmax 是一个更广义的 Logistic 激活函数，可以把它看作是 Sigmoid 函数的向量泛化，主要用于有多个类别的分类问题，而不仅仅是二元分类。它能够以总和等于 1 的方式映射每个输出（即输出是一个概率分布）。

因此，这两个函数的输入和输出略有不同，因为 Sigmoid 只接收一个输入，并且只输出一个表示属于 class1 的概率的数字（在二元分类中，只有两个可能的类，因此属于 class2 的概率= 1 - P(class1)）。而另一方面，SoftMax 是向量化的，这意味着它接受一个与我们拥有的类具有相同数量条目的向量，并输出一个概率向量，其中每个组件表示属于该类的概率 P ∈[0,1]（索引 i 处的项对应于属于类别 i 的特定输入的概率）。输出向量本质上是所有预测

类的概率分布，即向量的所有项之和必须为 1。这个限制可以解释为每个输入必须属于一个类，而且只能属于一个类。

在多项逻辑回归中，我们希望基于两个以上的类别进行分类。为了能够在两个以上的类之间进行分类，我们需要一个函数，该函数为每个类返回一个概率值，并且所有概率的和必须等于 1。Softmax 函数可以满足这些需求。因此，多项 Logistic 回归也称为 Softmax 回归。

4.4.2 示例：手写数字识别

在 spark.ml 中的 LogisticRegression 类可以通过二项逻辑回归（binomial logistic regression）来预测二分类结果，也可以通过多项逻辑回归（multinomial logistic regression）来预测多分类结果。使用 family 参数在这两种算法之间进行选择，或者不设置 family 参数，Spark 将推断出正确的变体。

多项逻辑回归可以通过将 family 参数设置为"multinomial"来进行二元分类。它会产生两组系数和两个截距。

【示例】手写数字分类识别。

我们将通过一个示例，来演示如何使用 spark.ml 中的 LogisticRegression 分类器对手写数字进行识别分类。

1）理解手写数字数据集

// .....

2）读取数据集

首先，加载数据集，代码如下：

```
// ......
```

3）探索性数据分析

查看数据数量和模式 Schema，代码如下：

```
println(s"共有数据${dataDF.count}条")
dataDF.printSchema
```

执行以上代码，输出内容如下：

```
共有数据 9912 条
// ......
```

查看前 10 条数据，代码如下：

```
dataDF.show(10)
```

执行以上代码，输出内容如图 4-所示：

统计数据集中，每种标签类型各有多少样本，代码如下：

```
dataDF.groupBy("label").count().orderBy("label").show
```

执行以上代码，输出内容如下：

```
```

4）特征工程

在上一步的探索性数据分析中可知，这里的数据不需要做更多处理，仅将各特征列组合到特征向量中即可。代码如下：

```
import org.apache.spark.ml.feature.VectorAssembler

// ......
```

执行以上代码，输出内容如图 4-所示：

5）分割数据集

将上一步经过特征工程后的数据集按 80/20 的比例，分割为训练集和测试集，这可能通过调用 randomSplit()方法来实现。代码如下：

```
// ......
```

执行以上代码，输出内容如下：

```
训练集记录数量: 7956
测试集记录数量: 1956
总记录数量: 9912
```

查看数据分布是否均衡，代码如下：

```
// ......
```

执行以上代码，输出内容如下：

```
```

可以看出，不管是在拆分后的训练集中，还是在测试集中，目标类都是平衡的。

6）构建和训练逻辑回归模型

在这里，将使用 LogisticRegression 分类器构建和训练逻辑回归模型，使用 features 作为输入特征列、label 列作为标签列（默认输入特征列名称就是 features，默认标签列名称就是 label，因此这两者可以省略）。通过指定 family 参数值为 multinomial，应用多项 Logistic 回归实现多分类任务。代码如下：

```
```

7）在测试集上测试模型

使用上一步训练出来的 mlrModel 对测试集进行预测，代码如下：

```
```

从以上输出结果可以看出，使用 mlrModel 模型对测试集进行转换后，输出结果集中增加了三个新列：rawPrediction、probability 和 prediction。其中 probability 列是模型预测输出的概率分布，prediction 列就是模型对测试集中每个样本的预测类别。选取 probability 列、prediction 列以及 label 列查看，代码如下：

执行以上代码，输出内容如图 4-所示：

4.5 分类模型评估

在为新数据部署模型之前，有一个重要的事情是在测试数据集中对该模型进行评估。既然我们已经了解了使用 Logistic 回归模型来实现二分类和多分类任务，那么为这些模型选择正确的评估指标就变得至关重要了。

分类模型常用的一些模型评价方法和指标如下：

（1）混淆矩阵。

（2）ROC (receiver operating characteristic，接收者操作特征) 下面积曲线。

4.5.1 混淆矩阵

为了理解逻辑回归模型的性能，我们应该使用一个混淆矩阵（Confusion Matrix），它由预测值与实际值的值计数组成。

对于二分类问题，假设分类类别为正例（Positive）和负例（Negative），则可将样例根据其真实类别与分类器预测类别的组合划分为真正例（True Positive，TP）、假正例（False Positive，FP）、真负例（True Negative，TN）、假负例（False Negative，FN）四种情形。

(1) 真正例（True Positive，TP）：对于一个正例样本，如果预测也是正的，则它是一个真正例；

(2) 假负例（False Negative，FN）：对于一个正例样本，如果预测它是负的，则它是一个假负例；

(3) 真负例（True Negative，TN）：对于一个负例样本，如果预测也是负的，则它是一个真负例；

(4) 假正例（False Positive，FP）：对于一个负例样本，如果将其预测为正的，则它是一个假正例。

在二分类问题中有两种类型的错误，即假正例（FP）和假负例（FN）。

令 TP、FP、TN、FN 分别表示真正例、假正例、真负例、假负例对应的样例数，则显然有这样的一种关系：TP+FP+TN+FN=样例总数。分类结果的混淆矩阵如下图 4-所示：

n=165	**Predicted: NO**	**Predicted: YES**	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

使用混淆矩阵对分类器或分类模型进行评估时，常用的指标有 accuracy（准确率）、precision（精确率）、recall（召回率）和 F1-score（F1 评分数）。

1. 准确率和错误率

Accuracy（准确率），是真正例（TP）和真负例（TN）之和除以样例总数，它表示分类器对给定的一组示例进行正确分类的频率，是最直观的性能度量。其计算公式如下：

$$Acc = \frac{N_{TP} + N_{TN}}{N_{FP} + N_{FN} + N_{TP} + N_{TN}}$$

Error（错误率），是分类器在给定的一组示例上出错的频率，它等于 1-准确率。其计算公式为：

$$E = \frac{N_{FP} + N_{FN}}{N_{FP} + N_{FN} + N_{TP} + N_{TN}}$$

对于对称的数据集，accuracy（准确率）和错误率是一个很好的衡量标准，其中 FP 和 FN 的值几乎是相同的。但当目标类不平衡时，它并不总是首选的度量标准。

大多数情况下，目标类的频率是倾斜的，即 TN 例子远比 TP 例子要多。例如，一个用于欺诈检测的数据集包含 99%的真实交易，只有 1%属于欺诈交易。那么，如果我们的逻辑回归模型预测所有的交易均为真实的交易，没有欺诈案例，那么它的准确率（accuracy）仍然是 99%。很明显，如果一个分类器不能识别一个正例，则它是毫无用处的。事实上，这种不平衡类的域非常常见。

注意： 这里找出欺诈交易是 TP（真正），找出非欺诈交易是 TN（真负）。上述例子中，虽然分类模型一个欺诈交易也没有找出，但由于它预测所有交易都是真实非欺诈的，也就是说 100 个交易中，它预测的真负（TN）99 个，而真正（TP）0 个，但根据准确度计算公式，该分类模型的准确度仍然达到 99%。

在目标类不平衡的情况下，错误率和分类准确率很难告诉我们分类器的实际用途。我们需要的标准不是将两个（或所有）类的性能平均起来，而是将重点放在一个类上，这个类虽然很重要，但只有几个样例。因此，这种情况下评价分类模型的重点是找出在识别正类方面的表现；此时，需要使用的评估标准是 precision（精确率）和 recall（召回率）。

2. 精确率和召回率

Precision（精确率），也叫查准率，是指模型预测的所有正例中实际正例（TP）的数量。是一个模型分类中 TP 占分类出的所有正例的比例。换句话说，precision（精确率）是分类器在标记一个样例为正时正确的概率。其计算公式如下：

$$Pr = \frac{N_{TP}}{N_{TP} + N_{FP}}$$

Recall（召回率），也叫查全率，指的是全部正例中分类模型能够正确预测出的实际正例（TP）的百分比。这有助于从正类的角度评估模型的性能，它告诉全部正例中该模型能够正确预测出的实际正例的百分比（TPR）。当预测正类时，它讨论了该机器学习模型的质量：在所有正类中，模型能正确预测多少？该指标被广泛地用作分类模型的评价标准。其计算公式如下：

$$Re = \frac{N_{TP}}{N_{TP} + N_{FN}}$$

例如，在下面这个例子中，虽然有很高的 accuracy，但 precision 和 recall 的比率都很低：

		Labels returned by the classifier	
		pos	neg
True labels:	pos	20	50
	neg	30	900

From here, the following values of precision, recall, and accuracy are obtained:

$$\text{precision} = \frac{20}{50} = 0.40; \quad \text{recall} = \frac{20}{70} = 0.29; \quad \text{accuracy} = \frac{920}{1{,}000} = 0.92$$

查准率和查全率是一对矛盾的度量。一般来说，查准率高时，查全率往往偏低；而查全率高时，查准率往往偏低。在一些应用中，对查准率和查全率的重视程度有所不同。例如，在商品推荐系统中，为了尽可能少打扰用户，更希望推荐内容的确是用户感兴趣的，此时更重视查准率；而在逃犯信息检索系统中，则更希望尽可能少漏掉逃犯，此时更重视查全率。

注意： Precision（查准率）和 Recall（查全率）的计算公式仅在分母上有所不同。这是有意义的。Precision 是分类器认为是正例中 TP 的比率，recall 是集合中所有正例中 TP 的比率。

3. F1 评分数

F1-score（F1 评分数），是查准率（precision）和查全率（recall）的加权平均值。因此，这个分数同时考虑了 FP 和 FN。其计算公式如下：

$$\text{F1 Score} = 2 * \frac{(Precision * Recall)}{(Precision + Recall)}$$

分类模型的 F1 评分取 0 到 1 之间的值。最好的模型 F1 评分为 1，而 F1 评分为 0 的分类模型是最差的。

从直觉上看，F1-score 不像准确率（accuracy）那样容易理解，但 F1 通常比准确率更有用，特别是在类分布不均匀的情况下。如果 FP 和 FN 造成的代价相似，则使用准确率评价指标最好。如果 FP 和 FN 造成的代价非常不同，那么最好同时考虑查准率和查全率。

4. 5. 2 ROC 曲线和 AUC 值

在实际商业应用中，我们一般不会以准确度作为模型的评估标准，因为准确度很多时候并不可靠。以银行客户违约预测模型为例，倘若 100 个客户中有 10 人违约，即便模型预测所有客户都不会违约，模型的预测准确度也能达到 90%，但显然这个较高的准确度并不能反映模型的优劣。在实际商业应用中，一般更关心以下两个指标：

(1) 真正率（命中率）：True Positive Rate（TPR），TPR=TP/(TP+FN)。

(2) 假正率（假报警率）：False Positive Rate（FPR），FPR=FP/(FP+TN)。

其中，TP、FP、TN、FN 的含义如下，该表也称为“混淆矩阵”。

表 4-1 评估银行客户违约预测模型的混淆矩阵表

	1（预测违约）	0（预测不违约）	合计
1（实际违约）	TP（Ture Positive） 正确肯定	FN（False Negative） 漏报	TP+FN
0（实际不违约）	FP（False Positive） 虚报	TN（True Negative） 正确否定	FP+TN

真正率（TPR）计算的是在所有实际违约的人中，预测为违约的比例，也称命中率或召回率；而假正率计算的是在所有实际没有违约的人中，预测为违约的比例，也称假警报率。

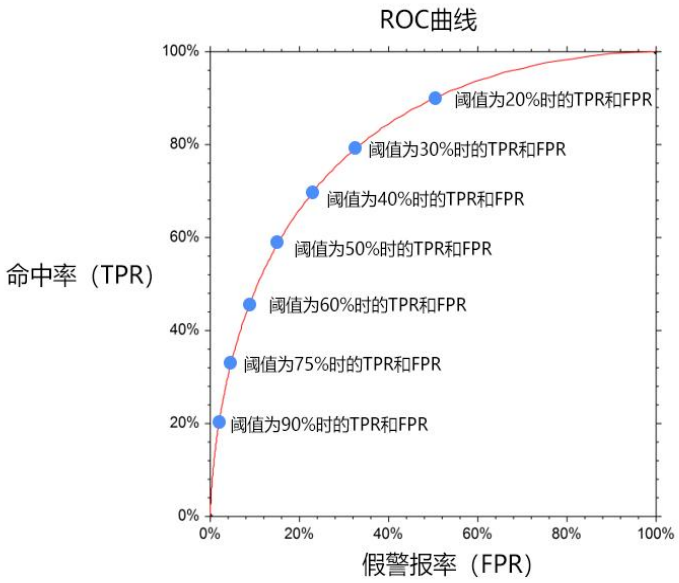
在上述例子中，100 个客户中有 10 人违约，模型预测所有客户都不会违约，如表所示，那么模型的假正率为 0，即没有误伤一个“好人”，但此时模型的真正率也为 0，即没有揪出一个“坏人”，由此可知即使是高达 90%的预测准确度也没有任何意义。

表 4-1 银行客户违约预测模型的预测结果

	1（预测违约）	0（预测不违约）	合计
1（实际违约）	TP=0 正确肯定	FN=10 漏报	TP+FN=10
0（实际不违约）	FP=0 虚报	TN=90 正确否定	FP+TN=90

由于我们知道 Logistic 回归模型的输出是概率分数，因此确定预测概率的截止或阈限是非常重要的。默认情况下，概率阈值设置为 50%。这意味着如果模型的概率输出小于 50%，模型将预测它属于一个负类(0)，如果它等于且大于 50%，则将被分配一个正类(1)。

一个优秀的客户违约预测模型，命中率应尽可能高，即能尽量揪出“坏人”，同时假警报率应尽可能低，即不要误伤“好人”。然而，这两者往往呈正相关性，因为一旦调高概率阈值，比如认为违约概率超过 90%才认定为违约，那么就会导致假警报率很低，命中率也很低；而降低概率阈值，如将违约概率超过 10%认定为违约，那么命中率就会高，但同时假警报率也会很高。因此，为了衡量一个模型的优劣，数据科学家们根据不同概率阈值下的命中率和假警报率绘制了 ROC 曲线（Receiver Operator Characteristic Curve，接收者操作特征曲线）图，如图 4-所示。



ROC 曲线的横坐标为假警报率（FPR），纵坐标为命中率（TPR）。可将 ROC 曲线用于确定分类模型的概率阈值：在某个概率阈值条件下，我们希望所构建模型的命中率尽可能高，而假警报率尽可能低。

如果把假警报率理解为代价，那么命中率就是收益，所以也可以说在相同阈值的情况下，我们希望代价（假警报率）尽量小，收集（命中率）尽量高，反映在图形上就是该曲线尽可能陡峭，曲线越靠近左上角说明在同样的阈值条件下，命中率越高，假警报率越低，模型越完善。换一个角度来理解，一个完善的模型是在不同的阈值下，假警报率接近于 0，而命中率接近于 1，该特征反映在图形上，就是曲线非常接近（0,1）这个点，即曲线非常陡峭。

在同一个分类器之内，阈值的不同设定对 ROC 曲线的影响，其规律如下：

(1) 当概率阈值设定为最高时，亦即所有样本都被预测为负例，没有样本被预测为正例，此时在假正率 FPR = FP / (FP + TN)算式中的 FP = 0，所以 FPR = 0%。同时在真正率（TPR）算式中 TPR = TP / (TP + FN) 算式中的 TP = 0，所以 TPR = 0%。因此，可得出结论：当概率阈值设定为最高时，必得出 ROC 座标系左下角的点 (0, 0)。

(2) 当概率阈值设定为最低时，亦即所有样本都被预测为正例，没有样本被预测为负例，此时在假正率 FPR = FP / (FP + TN) 算式中的 TN = 0，所以 FPR = 100%。同时在真正率 TPR = TP / (TP + FN) 算式中的 FN = 0，所以 TPR=100%。因此，可得出结论：当阈值设定为最低时，必得出 ROC 座标系右上角的点 (1, 1)。

(3) 因为 TP、FP、TN、FN 都是累积次数，TN 和 FN 随着阈值调低而减少（或持平），TP 和 FP 随着阈值调低而增加（或持平），所以 FPR 和 TPR 皆必随着阈值调低而增加（或持平）。随着阈值调低，ROC 点往右上（或右 / 或上）移动，或不动；但绝不会往左下（或左 / 或下)）移动。

在比较不同的分类模型时，可以将每个模型的 ROC 曲线都画出来，比较曲线下面积做为衡量模型优劣的指标。曲线下面积(Area under Curve，AUC)又称 ROC 下面积(Area under ROC)，是一种通常用于评价二分类器的度量。曲线下的面积越大，分类器越好。

因为 AUC 值是指 ROC 曲线下面的面积（在 1x1 的方格里求面积），因此 AUC 值必在 0~1 之间。AUC 值越大的分类器，正确率越高。从 AUC 判断分类模型优劣的标准如下：

(1) AUC = 1，是完美分类器，采用这个预测模型时，存在至少一个阈值能得出完美预测。绝大多数预测的场合，不存在完美分类器。

(2) 0.5 < AUC < 1，优于随机猜测。这个分类器模型妥善设定阈值的话，能有预测价值。

(3) AUC = 0.5，跟随机猜测一样，模型没有预测价值。

(4) AUC < 0.5，比随机猜测还差；但只要总是反预测而行，就优于随机猜测。

简单来说，AUC 面积的取值范围通常在 0.5~1 之间，0.5 表示随机判断，1 则代表完美的模型。在实际商业应用中，因为存在很多扰动因子，因此，AUC 值能达到 0.75 以上就已经可以接受了，如果能达到 0.85 以上，就是非常好的模型。

4.5.3 正确选择评价指标的策略

前面讲了各种对分类模型进行评估的指标，那么如何正确选择评价指标？现在正确选择评价指标的策略总结如下。

当模型预测 FP 的代码和 FN 的代价大致相等，或者预测 FP 和 TN 的收益大致相等时，选择 accuracy 评价指标。例如，评估预测客户会购买或不会购买商品的分类模型时，预测错误的 FP 和 FN 并不会造成多么严重不同的后果，这时选择 accuracy 作为我们分类模型的评价指标就比较合适。

当模型预测 FP 造成的后果远高于预测 FN 的后果，或者模型预测 TP 的收益远大于预测 TN 的收益时，选择 precision 评价指标。例如，在评估预测垃圾邮件的分类模型时，如果把正常的邮件误报为垃圾邮件，就有可能造成用户无法收到该邮件而错失某些重要信息的严重后果。这时对模型评估应选择 precision 评价指标。

当模型预测 FN 造成的后果远高于预测 FP 的后果，或者模型预测 TN 的收益远大于预测 TP 的收益时，选择 recall 评价指标。例如，评估预测病人是否患有恶性肿瘤的类型模型时，如果将本来患有恶性肿瘤的病人预测为未患有恶性肿瘤，则可能会耽误病人的及时治疗，造成严重的后果。

当处理平衡数据集时，使用 ROC 和 AUC 指标；当处理不平衡数据集时，使用 F1 评分指标（调合 precision 和 recall）。

4.5.4 示例：客户购买预测模型评估

本节继续 4.2.2 节中的示例，对构建的 LogisticRegression 模型进行评估。

【示例】使用 Spark 机器学习库的 LogisticRegression 类来构建逻辑回归模型，并对该模型进行评估。

首先，加载数据集，执行特征转换和合并，构建并训练逻辑回归模型。代码如下：

```
// ......
```

然后，将训练的得到模型 purchase_model（它是 LogisticRegressionModel 的一个实例）应用在测试集上进行预测，这通过调用 LogisticRegressionModel 的 transform()方法来实现，代码如下：

```
// 模型的 transform 方法输出 DataFrame
```

```
val test_df_prediction = purchase_model.transform(test_df)
test_df_prediction.drop("features").show(false)
```

执行以上代码，输出内容如图 4-所示：

其中 probability 列是预测输出的各类比的概率值，prediction 是对概率执行 sigmod 函数后输出的预测目标值。

由于本示例是一个分类问题，因此使用混淆矩阵来衡量模型的性能。这里我们将手工为 TP、TN、FP 和 FN 创建变量，以便更好地理解它们。代码如下：

```
// ......
```

执行以上代码，输出内容如下：

```
真正例（TP）：2358
真负例（TN）：2299
假正例（TP）：163
假负例（TP）：2358
```

然后，根据以上计算结果，计算准确率、精确率和召回率。代码如下：

```

```

执行以上代码，输出内容如下：

```

```

逻辑回归的 spark.ml 实现还支持在训练集上提取模型的摘要，这些摘要信息封装在 LogisticRegressionSummary 的实例对象中。因此，也可以使用 evaluate()方法在测试数据集上评估模型，该方法以优化的方式执行所有步骤，并返回一个 LogisticRegressionSummary 类，它是对预测结果进行总结和概括的类，包含许多对模型的评价指标。代码如下：

```

```

LogisticRegressionTrainingSummary 为 LogisticRegressionModel 提供了一个摘要。在二分类的情况下，某些额外的指标是可用的，例如 ROC 曲线。二分类摘要可以通过 binarySummary 方法访问，得到 BinaryLogisticRegressionTrainingSummary。例如，从前面训练的返回的 LogisticRegressionModel 实例中提取摘要，代码如下：

```

```

4.5.5 示例：鸢尾花分类预测模型评估

本节继续 4.3.2 节中的示例，对构建的 LogisticRegression 模型进行评估。

【示例】使用 Spark ML 中的 Logistic 回归算法实现对鸢尾花数据集进行分类，并对训练的分类模型进行评估。

首先，加载数据集，执行特征转换和合并，构建并训练逻辑回归模型。代码如下：

```
// ......
```

在 spark.ml 机器学习库中，提供了 MulticlassClassificationEvaluator 评估类，它是用于多类分类的评估器。接下来，应用训练出来的模型在测试集上做预测，并使用这个多类分类的评估器对测试集测试结果进行评估。代码如下：

```
//
```

4.5.6 示例：手写数字识别预测模型评估

本节继续 4.4.2 节中的示例，对构建的 LogisticRegression 模型进行评估。

【示例】使用 Spark ML 中的 Logistic 回归算法实现对鸢尾花数据集进行分类，并对训练的分类模型进行评估。

首先，加载数据集，执行特征转换和合并，构建并训练逻辑回归模型。代码如下：

```

```

执行以上代码，输出内容如下：

```
准确率（Accuracy）: 0.9460648730198642
FPR: 0.0059794463728614295
TPR: 0.9460648730198641
F-measure: 0.9460738861899096
Precision: 0.9462484022026725
Recall: 0.9460648730198641
```

4.6 模型选择与调优

机器学习中的一个重要任务是模型选择，或者使用数据为给定任务找到最佳模型或参数。这也被称为调优（tuning）。模型调优步骤的目标是用正确的参数集训练一个模型，以获得最佳性能。这个步骤通常是冗长的、重复的和耗时的，因为它可能涉及到尝试不同的机器学习算法或一些参数。

4.6.1 参数调优

机器学习的各个模型中都有内置参数，例如，上面提到的逻辑回归模型中的一个很重要的参数 regParam，这是用来控制过度拟合的正则化参数。这种参数也称为“超参数”。

机器学习模型的性能或质量取决于模型训练过程中提供给机器学习算法的超参数。例如，使用逻辑回归算法训练的模型的有效性取决于梯度下降迭代的步长和次数。但是，找到能够训练出最佳模型的超参数组合是比较困难的。

寻找最佳超参数的技术之一是在超参数空间上进行网格搜索。网格搜索（Grid Search）是一种调参手段，它在参数列表中进行穷举搜索，对每种情况进行训练，找到最优的参数；在网格搜索中，使用来自超参数空间的指定子集的每个超参数组合来训练模型。它是一种调优方法。

注意： 穷举搜索：在所有候选的参数选择中，通过循环遍历，尝试每一种可能性，表现最好的参数就是最终的结果。其原理就像是在数组里找最大值。

例如，假设有一个机器学习算法需要两个实值超参数 p1 和 p2。我们不需要猜测 p1 和 p2 的最优值，而是对 p1 取值 0.01、0.1、1 以及 p2 取值 20、40、60 的组合进行网格搜索。这就产生了 9 种不同的 p1 和 p2 组合。用 p1 和 p2 的每个组合来训练和评估一个模型。选择产生最优训练模型的那一组超参数组合。

网格搜索的代价很高，参数越多，候选值越多，耗费时间越长。但它是一种比猜测每个超参数的最优值更好的超参数优化方法。通常，这是找到性能良好模型是必需步骤。

4.6.2 交叉验证

在处理机器学习任务时，我们必须正确识别问题，这样我们才能选择最合适的算法，给出最好的分数。但是我们如何比较这些模型呢?

大多数学习算法都需要一些参数调优。在选择不同的超参数组合进行参数调优的过程中，如何选择最优的参数组合？比如说，我们已经用某个超参数组合在数据集中训练了模型，现在想知道模型的性能如何。其中一种方法是在训练过的数据集上测试模型，但这肯定不是一个好的做法，因为这将不可避免地产生过拟合问题。

所以要知道模型的真实评分，就需要对模型从未见过的数据进行测试，这组数据通常被称为测试集。但是，如果我们把数据分成训练数据和测试数据，难道我们不会丢失测试数据集可能包含的一些重要信息吗? 这可以通过 K-fold（K 折）交叉验证法来解决。这种方法保证了我们模型的评分不依赖于我们选择训练和测试集的方式。

在 K-fold 交叉验证中，数据集被等比例划分成 K 份，以其中的一份作为测试数据，其他的 K-1 份数据作为训练数据。例如，将数据集分成 10 份，其中 9 份作为训练集，用于运行算法，保留 1 份子集作为测试集。这个过程重复 10 次，每次换一个不同的测试集。这称为"K-fold 交叉验证"，其中 K 是数据集被分割的子集数量。

K-fold 交叉验证法的执行步骤如下：

(1) 将整个数据集随机分成 k 个折叠（k-fold，k 个子集）。

(2) 对于数据集中的每一次折叠，在数据集的 k - 1 次折叠上构建模型。然后，对模型进行测试，以检查第 k 次折叠的有效性。

(3) 重复这个过程，直到每一个 k 折叠都作为测试集被测试过。

K 个记录准确度（accuracy）的平均值称为交叉验证准确度（cross-validation accuracy），并将作为模型的性能度量。K-fold 交叉验证的整个过程如图 4-所示：

图 4- K-fold 交叉验证法

图 4-中红色的部分为每次从样本数据集中抽取出来作为测试集的部分。

因为 K-fold 交叉验证保证了原始数据集的每个观测值都有机会出现在训练集和测试集中，所以与其他方法相比，这种方法通常会得到一个较小的偏差模型。这种方法的缺点是训练算法必须从头开始运行 k 次，这意味着需要 k 倍的计算才能进行评估。在实践中，k 一般取 10，取值依不同项目情况而定。

交叉验证经常与网格搜索进行结合，作为参数评价的一种方法，这种方法叫做带交叉验证的风格搜索（grid search with cross validation）。

4.6.3 Spark ML 模型选择

在 Spark ML 中，调优可以针对单独的 Estimators（如 LogisticRegression）进行，也可以针对包含多个算法、特征和其他步骤的整个机器学习管道进行。用户可以一次调优整个机器学习管道，而不是单独调优机器学习管道中的每个元素。

1）Spark ML 调优工具

Spark ML 提供了一些工具来帮助模型调优步骤。Spark ML 中来帮助进行模型调优的两个常用的类是 CrossValidator 和 TrainValidationSplit，它们都是 Estimator 类型。这些类也称为验证器。它们都需要以下输入：

(1) Estimator：需要调优的内容，它可以是一个机器学习算法或一个 Pipeline 实例。换句话说，它必须是 Estimator 类型。

(2) ParamMaps：可供 Estimator 选择的一组参数。这些参数也被称为参数网格，用于搜索寻找最优模型。

(3) Evaluator：评估器，用于度量一个拟合模型在测试数据上的表现。对于每一个不同的机器学习任务，Spark ML 都提供了一个特定的 Evaluator，它可以生成一个或多个评估指标，以了解模型的性能。

不管是 CrossValidator 还是 TrainValidationSplit，它们都按以下过程进行模型选择：

(1) 它们将输入数据分成单独的训练数据集和测试数据集。

(2) 对于每个(训练集，测试集)对，它们遍历 ParamMaps 集合。对于每个 ParamMap，他们使用这些参数拟合 Estimator，获得拟合的模型，并使用 Evaluator 评估模型的性能。

(3) 它们选择由表现最好的一组参数产生的模型。

TrainValidationSplit 将给定的输入数据分割成一个基于指定比例的训练和验证数据集，然后根据每个参数组合对数据集对进行训练和评估。例如，如果给定的参数集有 6 个组合，那么给定的 Estimator 就会被训练和评估 6 次，每次都使用不同的参数组合。

CrossValidator 使用 k-折交叉验证和网格搜索来进行超参数选择和模型调优。它从用户指定的不同超参数集生成所有超参数的组合。对于每个组合，它使用一个 Estimator 使用训练数据集训练模型，并使用一个 Evaluator 在测试数据集上评估生成的模型。它对所有 k 对训练和测试数据集重复这个步骤，并为每组计算特定评估指标的平均值。它会选取平均评价指标最优的超参数作为最优超参数。最后，交叉验证器使用最佳超参数在整个数据集上训练模型。

CrossValidator 采用 K-fold 交叉验证法来执行交叉验证，从而使训练和验证的数据量最大化。需要注意的是，使用交叉验证器的代价可能会很高，因为它尝试来自指定参数网格的每个超参数组合。例如，如果 k 是 4，并且参数组合的数量是 6，那么 Estimator 将被训练和评估的总次数是 24。

不过，这是一种成熟的选择最优超参数值的方法。它是一种统计上优于启发式手工调优的方法。CrossValidator 是在实际中应用较多的验证器，是在机器学习从业者社区中广泛使用的技术实现。但必须要注意的是，使用这个验证器（有相当数量的参数组合）的开销。

对于回归问题，Spark ML 提供的评估器是 RegressionEvaluator。对于二分类问题，Spark ML 提供的评估器是 BinaryClassificationEvaluator。对于多类分类问题，Spark ML 提供的评估器是 MulticlassClassificationEvaluator。对于多标签分类问题，Spark ML 提供的评估器是 MultilabelClassificationEvaluator。用于选择最佳 ParamMap 的默认度量可以被这些求值器中的 setMetricName()方法覆盖。

为了帮助构造参数网格，用户可以使用 ParamGridBuilder 工具。默认情况下，参数网格中的参数集以串行方式计算。在使用 CrossValidator 或 TrainValidationSplit 运行模型选择之前，可以通过将 parallelism 设置为 2 或更多的值（1 将是串行的）来并行地进行参数评估。应该仔细选择 parallelism 的值，以在不超过集群资源的情况下最大化并行性，较大的值可能并不总是导致性能的提高。一般来说，对于大多数集群来说，最多 10 的值就足够了。

2）CrossValidator 调优示例

CrossValidator 首先将数据集分成一组折叠，这些折叠用作单独的训练和测试数据集。例如，当 k=3 次折叠时，CrossValidator 将生成 3 个(训练，测试)数据集对，每个数据对使用 2/3 的数据用于训练，1/3 用于测试。为了评估一个特定的 ParamMap，CrossValidator 通过在 3 个不同的(训练、测试)数据集对上拟合 Estimator 来计算 3 个模型的平均评估度量。

在确定最优 ParamMap 之后，CrossValidator 最终使用最优 ParamMap 和整个数据集重新拟合 Estimator。

下面的示例演示使用 CrossValidator 从参数网格中进行选择。

【示例】通过交叉验证选择最优逻辑回归模型，判断一行文本中是否包含单词 spark。

首先，导入相关的依赖包，代码如下：

```
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.ml.linalg.Vector
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.sql.Row
```

然后，从一个(id, text, label)元组列表中准备训练数据。代码如下：

```

```

执行以上代码，输出内容如下：

```
res32: org.apache.spark.ml.param.ParamMap =
{
    logreg_789319ccbe67-aggregationDepth: 2,
    logreg_789319ccbe67-elasticNetParam: 0.0,
    logreg_789319ccbe67-family: auto,
    logreg_789319ccbe67-featuresCol: features,
    logreg_789319ccbe67-fitIntercept: true,
    logreg_789319ccbe67-labelCol: label,
    logreg_789319ccbe67-maxBlockSizeInMB: 0.0,
    logreg_789319ccbe67-maxIter: 10,
    logreg_789319ccbe67-predictionCol: prediction,
    logreg_789319ccbe67-probabilityCol: probability,
    logreg_789319ccbe67-rawPredictionCol: rawPrediction,
    logreg_789319ccbe67-regParam: 0.1,
    logreg_789319ccbe67-standardization: true,
    logreg_789319ccbe67-threshold: 0.5,
    logreg 789319ccbe67-tol: 1.0E-6
}
```

可以看到，当取(maxIter=10, regParm=0.1)这组参数组合时，模型最优。

3）TrainValidationSplit 调优示例

除了 CrossValidator 之外，Spark 还提供了 TrainValidationSplit 用于超参数调优。TrainValidationSplit 只对每个参数组合求值一次，而 CrossValidator 则需要求值 k 次。因此，它调优成本更低，但当训练数据集不够大时，它不会产生可靠的结果。

与 CrossValidator 不同，TrainValidationSplit 创建单个(训练、测试)数据集对。它使用 trainRatio 参数将数据集分成这两部分。例如，当 trainRatio=0.75 时，TrainValidationSplit 将生成一个训练和测试数据集对，其中 75%的数据用于训练，25%用于验证。

像 CrossValidator 一样，TrainValidationSplit 最终使用最优的 ParamMap 和整个数据集来拟合 Estimator。下面的示例演示使用 TrainValidationSplit 从参数网格中进行选择。

【示例】通过 TrainValidationSplit 选择最优逻辑回归模型，判断一行文本中是否包含单词 spark。

数据集和特征转换与上一节相同，构造逻辑回归模型，并使用 TrainValidationSplit 和网格搜索来训练模型，代码如下：

```
// ......
```

执行以上代码，输出内容如下：

```

```

最后，提取出最优模型，查看最优参数组合，代码如下：

```
// 提取最优模型
val bestModel = cvModel.bestModel

// 查看最优模型的参数
bestModel.extractParamMap()
```

执行以上代码，输出内容如下：

```

```

可以看到，当取(maxIter=10, regParm=0.1)这组参数组合时，模型最优。

4. 7 模型保存和加载

在 org.apache.spark.ml.util 包中有两个抽象类 MLWriter 和 MLReader，通过它们可以保存和加载模型。

4. 7. 1 MLWriter 抽象类

MLWriter 抽象类自带 save(path: String)方法，用于以 Spark 的内部格式将 ML 组件保存到给定的 path 路径。该方法的定义如下：

```
def save(path: String): Unit
```

4. 7. 2 MLReader 抽象类

MLReader 抽象类自带 load(path: String)方法，用于从给定 path 路径加载 ML 组件。该方法的定义如下：

```
abstract defload(path: String): T
```

Spark 机器学习库中所有的 Estimator 和 Transformer 的伴生对象中，都有一个 read()方法，调用该方法，会返回一个 MLReader 的实例。例如，读取之前存储的未拟合机器学习管道，代码如下：

```
import org.apache.spark.ml._

val pipeline = Pipeline.read.load("sample-pipeline")

val stageCount = pipeline.getStages.size   // 0
```

读取之前存储的机器学习管道模型，代码如下：

```
val pipelineModel = PipelineModel.read.load("sample-model")
pipelineModel.stages
```

4.7.3 示例：存储和调用最优模型

本节将对 4.6.3 小节中 CrossValidator 调优示例进行改写，将使用交叉验证和网络搜索后选择的最优模型存储起来，并在稍后加载该最优模型来对新的数据集进行预测。

【示例】训练一个逻辑回归分类器，判断一行文本中是否包含单词 spark。通过交叉验证选择最优逻辑回归模型，保存该最优模型，并在稍后加载使用。

```

```

4.8 应用机器学习管道

Spark ML Pipeline 提供了一组统一的高级 API，构建在 DataFrame 之上，帮助用户创建和调优实用的机器学习管道。

4.8.1 管道相关的 API

Spark 机器学习库中 标准化了机器学习算法的 API，使其更容易将多个算法组合到单个管道或工作流中。

1）DataFrame

机器学习可以应用于各种各样的数据类型，如向量、文本、图像和结构化数据。Spark ML API 使用 Spark SQL 中的 DataFrame 作为 ML 数据集，以支持多种数据类型。例如，一个 DataFrame 可以有不同的列来存储文本、特征向量、真实标签和预测。

DataFrame 支持许多基本和结构化类型。除此之外，DataFrame 还可以使用 ML Vector 类型。可以隐式或显式地从常规 RDD 创建 DataFrame，也可以读取各种数据源来创建 DataFrame。

注意： 有关 Spark SQL 和 DataFrame 的数据类型、数据源、API 等使用，可以参考清华出版社的《Spark 原理深入与编程实战》一书中第 4 章和第 5 章的内容。

DataFrame 中的列被命名，例如"text"、"features"和"label"之类的名称。

2）Transformer

Transformer 是包含特征转换器和学习模型的抽象。从技术上讲，每个 Transformer 都实现了一个 transform()方法，该方法通常通过附加一个或多个列将一个 DataFrame 转换为另一个 DataFrame。例如：

(1) 一个特征转换器可以接受一个 DataFrame，读取一个列(例如，文本)，将它映射到一个新列(例如，特征向量)，然后输出一个附加了映射列的新 DataFrame。

(2) 一个学习模型可能会接受一个带有特征的 DataFrame，读取包含特征向量的列，预测每个特征向量的标签，然后输出一个新的 DataFrame，并将预测的标签附加为列。

简单来说，Transformer 是一种可以将一个 DataFrame 转换为另一个 DataFrame 的算法。

3）Estimator

Estimator 是一种算法，可以拟合或训练 DataFrame 数据以产生 Transformer。从技术上讲，每个 Estimator 都实现了一个方法 fit()，它接受一个 DataFrame 并产生一个 Model，这是一个 Transformer。例如，像 LogisticRegression 这样的学习算法是一个 Estimator，调用其 fit()方法，传入带有特征向量的 DataFrame，训练出一个 LogisticRegressionModel，它是一个 Model，因此是一个 Transformer。

Transformer.transform()和 Estimator.fit()都是无状态的。在未来，有状态算法可能会通过其他概念得到支持。

Transformer 或 Estimator 的每个实例都有一个惟一的 ID，这在指定参数时很有用。

4）Pipeline

在机器学习中，运行一系列算法来处理和学习数据是很常见的，这些阶段构成了一个机器学习工作流程。Spark ML 将一个机器学习工作流表示为一个 Pipeline（管道），它由一系列以特定顺序运行的 PipelineStages（Transformer 和 Estimator）组成。

换句话说，一个 Pipeline 被指定为一系列阶段，每个阶段要么是 Transformer，要么是 Estimator。这些阶段按顺序运行，输入的 DataFrame 在经过每个阶段时进行转换。对于 Transformer 阶段，在 DataFrame 上调用 transform()方法。对于 Estimator 阶段，调用 fit()方法来生成一个 Transformer（它成为 PipelineModel，或已拟合的 Pipeline 的一部分），并且在 DataFrame 上调用该 Transformer 的 transform()方法。

例如，在一个简单的文本数据处理工作流中，Pipeline 在训练时的用法如图 4-所示：

在图 4-中，最上面一行表示有三个阶段的 Pipeline。前两个（Tokenizer 和 HashingTF）是 Transformer，第三个（LogisticRegression）是 Estimator。底部一行表示流经管道的数据，其中的圆柱体表示 DataFrame。Pipeline.fit()方法在原始 DataFrame 上调用，该 DataFrame 具有原始文本文档和标签。Tokenizer.transform()方法将原始文本文档拆分为单词，并在 DataFrame 中添加一个包含 words 的新列。HashingTF.transform()方法将 words 列转换为特征向量，并将具有这些向量的新列添加到 DataFrame 中。现在，由于 LogisticRegression 是一个 Estimator，Pipeline 首先调用 LogisticRegression.fit()来生成一个 LogisticRegressionModel。

如果该 Pipeline 有更多的 Estimator，那么在将 DataFrame 传递到下一个阶段之前，它将在 DataFrame 上调用 LogisticRegressionModel 的 transform()方法。

一个 Pipeline 是一个 Estimator。因此，在 Pipeline 的 fit()方法运行之后，它会生成一个 PipelineModel，这是一个 Transformer。这个 PipelineModel 在测试时使用，用法如图 4-所示。

在图 4-中，PipelineModel 具有与原始 Pipeline 相同数量的阶段，但是原始 Pipeline 中的所有 Estimators 都变成了 Transformer。当在测试数据集上调用 PipelineModel 的 transform()方法时，数据将按顺序通过拟合的管道传递。每个阶段的 transform()方法更新数据集并将其传递给下一个阶段。

Pipeline 和 PipelineModel 有助于确保训练和测试数据经过相同的特征处理步骤。

5）Parameter

所有的 Transformer 和 Estimators 现在都使用统一的 API 来指定参数。其中 Param 是带有自包含文档的命名参数。ParamMap 是一组(参数, 值)对。

向算法传递参数主要有两种方式：

(1) 为实例设置参数。例如，如果 lr 是 LogisticRegression 的实例，则可以调用 lr.setMaxIter(10)来使 lr.fit()最多使用 10 次迭代。

(2) 向 fit()或 transform()传递一个 ParamMap。ParamMap 中的任何参数都将覆盖先前通过 setter 方法指定的参数。

参数属于 Estimator 和 Transformer 的特定实例。例如，如果我们有两个 LogisticRegression 实例 lr1 和 lr2，那么我们可以用指定的两个 maxIter 参数构建一个 ParamMap：

```
ParamMap(lr1.maxIter -> 10, lr2.maxIter -> 20)
```

如果在一个 Pipeline 中有两个带有 maxIter 参数的算法，这是很有用的。

4.8.2 管道的存储和加载

通常情况下，将模型或管道保存到磁盘以供以后使用是值得的。在 Spark 1.6 中，模型导入/导出功能被添加到 Pipeline API 中。从 Spark 2.3 开始，模型导入/导出功能已经覆盖了 spark.ml 和 pyspark.ml 中基于 DataFrame 的 API。

Spark ML Pipeline 的持久化可以跨 Scala、Java 和 Python 工作。这就意味着，我们可以将 Scala API 开发的 Pipeline 存储到磁盘中，然后在 Java 或 Python API 中加载它使用，反之亦然。

一般来说，Spark 维护 ML 持久性的向后兼容性。也就是说，如果我们在一个版本的 Spark 中保存了一个 ML 模型或 Pipeline，那么我们应该能够将它加载回来并在未来的 Spark 版本中使用它。然而，也有一些罕见的例外，如下所述。

(1) 模型持久化：在 Spark 版本 X 中使用 Apache Spark ML 持久化保存的模型或管道是否可以由 Spark 版本 Y 加载？答案是：对于主版本来说，不保证，但尽最大努力；对于次要和补丁版本来说，它们是向后兼容的。

(2) 模型行为：Spark 版本 X 中的模型或管道在 Spark 版本 Y 中的行为相同吗？答案是：对于主版本来说，不保证，但尽最大努力；对于次要和补丁版本来说，保持相同的行为，除了 bug 修复。

注意： 对于模型持久性和模型行为，任何跨小版本或补丁版本的破坏性更改都会在 Spark 版本发布说明中报告。如果一个漏洞没有在发布说明中报告，那么它应该被视为一个需要修复的 bug。

4.8.3 机器学习管道示例

在本节中，我们将以 Logister 回归为例，通过两个示例来理解 Spark 机器学习管道中各个组成阶段 Estimator、Transformer、Param 的概念、用法，以及机器学习管道应用。

1）简单的参数传递示例

【示例】一个简单的 Logister 回归示例，演示不同的参数传递方法。

首先，构造一个训练数据集。从(label, features)元组列表中准备训练数据，代码如下：

```
```

注意，model2.transform()输出一个 myProbability 列，而不是通常的 probability 列，因为我们前面重命名了 lr.probabilityCol 参数。

2）典型的机器学习管道示例

在下面另一个示例中，构建一个典型的机器学习过程。在这个机器学习过程中，我们构造一个机器学习管道，包含特征工程处理和 Logister 回归模型训练，目的是查找出所有包含 spark 的句子，即将包含 spark 的句子的标签设为 1，没有 spark 的句子的标签设为 0。

【示例】构建一个机器学习管道，判断一条语句中是否包含 spark 这个单词。

首先，构造一个训练数据集，包含字符串文本行列和 label 标签列。代码如下：

```
```

3）机器学习管道和交叉验证

CrossValidator 需要一个 Estimator，一组 Estimator ParamMaps 和一个 Evaluator。我们现在也可将 Pipeline 视为一个 Estimator，将其封装在一个 CrossValidator 实例中。然后我们就可以共同选择所有管道阶段的参数。

【示例】构建一个机器学习管道，判断一条语句中是否包含 spark 这个单词。并使用交叉验证和参数网格来选择最优模型。

首先，准备数据集。这里出于测试的目的，从一个 (id, text, label)元组列表构建一个训练数据集，代码如下：

```
```

4.8.4 示例：垃圾邮件分类

本节我们演示如何构建一个机器学习管道来训练一个垃圾邮件分类器，使用逻辑回归算法来估计给定消息是垃圾邮件的概率。构建的机器学习管道如图 4-所示：

本示例中使用 Ling-Spam 数据集。其中 ham 目录下为收集到的非垃圾邮件，spam 目录下为收集到的垃圾邮件。

【示例】通过构建一个机器学习管道来训练一个垃圾邮件过滤器，进行垃圾邮件分类。

首先，导入相关的依赖包，并使用 SparkContext 的 wholeTextFiles 方法分别加载 ham 下的非垃圾邮件和 spam 目录下的垃圾邮件。代码如下所示：

```

```

从输出内容可以看出，在测试集上我们训练的模型质量还是不错的，ROC 值达到 0.997，非常接近于 1。当然了，读者的运行结果可能与此稍有不同，因为每次运行对数据集的分割都是随机的。这里得到的 ROC 值可以解释为，给定从测试集中随机抽取的任意两条消息，其中一条是 ham，另一条是 spam，模型有 99.7%的概率为 spam 消息分配了比 ham 消息更大的为垃圾邮件的可能性。

4.8.5* 本章需要修改的地方

交叉验证部分的导入描述，以及图示。

k-fold 交叉验证将训练数据集分解为 k-fold 而不进行替换，即任何给定的数据点只属于其中一个子集，其中 k-1 个 fold 用于模型训练，1 个 fold 用于测试。该过程重复 k 次，从而获得 k 个模型和性能估计。

然后，我们根据单个 fold 计算模型的平均性能，以获得相比 holdout 或单 fold 方法对训练数据细分较不敏感的性能估计。

交叉验证允许我们避免这种特殊情况，减少结果方差，并为我们的模型生成更真实的分数。k-fold 交叉验证的通常步骤如下:

1. 将数据集划分为 k 个不同的子集。
2. 通过对 k-1 子集的培训和对其余子集的测试，创建 k 个不同的模型。
3. 测量 k 个模型中每种模型的性能并取平均值。

默认情况下，交叉验证使用准确性(accuracy)作为性能度量，但是我们可以通过传递任何记分函数作为参数来选择度量。

第 5 章 Spark ML 分类算法进阶

Spark 机器学习库为分类任务提供了许多机器学习算法的分布式实现。在第 4 章中，我们详细讲解了其中的 Logistic 回归算法原理，以及使用它完成分类任务的完整流程。这一章，我们将讲解更多的分类算法原理和应用，如决策树分类器、朴素贝叶斯分类器等。

5.1 决策树分类器

决策树用于机器学习中的分类或回归任务。它们使用分层树结构，其中内部节点代表一个特征，分支代表决策规则，每个叶节点代表数据集的结果。

决策树通过学习从输入变量推断出的决策规则来预测输出变量的值。决策树算法使用一组树状的用户定义或学习规则，根据输入示例的特征值对它们进行分类。因为它不是基于复杂的数学，所以很容易使用和理解。它可以使用简单的决策规则（从训练数据集中学习的）执行分类和回归分析。学习的决策规则可以被可视化，并且提供了算法的内部工作的一个直观的解释。

5.1.1 决策树算法简介

决策树算法是机器学习的各种算法中比较容易理解的一个算法，其基本原理是通过对一系列问题进行 if-else 推导，最终实现相关决策。

决策树算法的核心原理可以通过一个典型的决策树模型-员工离职预测模型-来简单演示，如图 5-1 所示。

图 5-1　决策树模型原理

该决策树首先判断员工满意度是否小于 5，如果答案为“是”则认为该员工会离职，答案为“否”则继续判断其收入是否少于 10000 元，若答案为“是”则认为该员工会离职，答案为“否”则认为该员工不会离职。

决策树算法涉及几个重要的概念：父节点、子节点、根节点和叶子节点。父节点与子节点是相对的，子节点由父节点根据某一规则分裂而来，子节点作为新的父节点继续分裂，直至不能分裂为止。根节点则与叶子节点是相对的，根节点是没有父节点的节点，即初始节点。叶子节点是没有子节点的节点，即最后的节点。决策树模型的关键就是如何选择合适的节点进行分裂。例如，图 5-1 中的“满意度<5”就是根节点，但它同时也是一个父节点，从它分裂出来两个子节点“离职”和“收入<10000”。其中子节点“离职”因为不再分裂，不再有子节点，所以也称为叶子节点；另一个子节点“收入<10000”同时也是一个父节点，它又分裂出两个子节点“离职”和“不离职”，这两个子节点因为不再有自己的子节点（不再分裂），所以也是叶子节点。

在实际应用中，企业会通过已有的数据来判断离职员工符合何种特征，如员工满意度、收入、工龄、月工时、项目数等，然后选择相应的特征进行节点分裂，便可以构建出类似图 5-1 所示的决策树模型。利用该模型就可以对员工离职情况进行预测，并根据预测结果采取相应的应对措施。

决策树算法的概念本身并不复杂，主要就是通过连续的逻辑判断得到最后的结论，其关键在于如何建立一棵这样的“树”。例如，根节点应该选择哪一个特征？选择“满意度<5”作为根节点还是选择“收入<10000”作为根节点，最后的结果和效果是不同的。再例如，收入作为一个连续变量，是选择“收入<10000”作为一个节点，还是选择“收入<50000”作为一个节点，结果也会有一定的差别。

总体来说，决策树算法作为机器学习的经典算法，有其独特优势，如对异常值不敏感、、可解释性强等。与逻辑回归等线性模型相比，决策树不需要特征缩放，不需要数据标准化。它能够处理缺失的值（丢失的特征），并同时处理连续和分类特征。不过它也存在一些缺点，如结果不稳定、容易造成过拟合等。更重要的是，决策树模型是很多重要集成模型的基础，如随机森林模型、GDBT 模型和 XGBoost 模型，因此需要读者很好地去学习和理解。

5.1.2 决策树算法原理

决策树算法的建树依据主要有基尼（gini）系数和信息熵（entroy）。

1. 基尼系数

基尼系数用于计算一个系统中的失序现象，即系统的混乱程度。基尼系数越高，系统混乱程度就越高。建立决策树模型的目的就是降低系统的混乱程度，从而得到合适的数据分类效果。基尼系统的计算公式如下：

$$gini(T)=1-\sum p_i^2$$

其中，p_i 为类别 i 在样本 T 中出现的概率，即类别为 i 的样本占总样本个数的比率；$\sum$ 为求和公式，即对所有的 p_i^2 进行求和。

例如，对于一个全部都是离职员工的样本而言，其中只有一个类别-离职员工，其出现的概率是 100%，所以该系统基尼系统为 $1-1^2=0$，表示该系统没有混乱，或者该系统的“纯度”很高，很纯粹。如果一个样本中有一半是离职员工，一半是未离职员工，那么类别个数为 2，每个类别出现的概率都为 50%，所以基尼系数为 $1-(0.5^2+0.5^2)=0.5$，即其混乱程度很高。

当引入某个用于进行分类的变量（如“满意度<5”）时，分割后的基尼系数公式如下：

$$gini(T)=\frac{S_1}{S_1+S_2}gini(T_1)+\frac{S_2}{S_1+S_2}gini(T_2)$$

其中，S_1、S_2 表示划分成两类后各自的样本量；$gini(T_1)$ 和 $gini(T_2)$ 表示划分后的两类各自的基尼系数。

例如，一个初始样本中有 1000 名员工，已知其中有 400 人离职，600 人未离职。划分前该系统的基尼系数为 $1-(0.4^2+0.6^2)=0.48$，那么下面采用两种不同的划分方式来决定初始节点：

(1) 根据“满意度<5”进行分类。

(2) 根据“收入<10000”进行分类。

对于第一种方式，以“满意度<5”为根节点进行划分，划分后的基尼系数为 0.3，如图 5-2 所示。

图 5-2　决策树模型原理

对于第二种方式，以“收入<10000”为根节点进行划分，划分后的基尼系数为 0.45，如图 5-2 所示。

图 5-3　决策树模型原理

可以看到，在根节点划分前的基尼系数为 0.48。当以“满意度<5”为根节点进行划分后基尼系数为 0.3，而以“收入<10000”为根节点进行划分后基尼系数为 0.45。基尼系数越小表示系统的混乱程度越低，区分度越主，能够较好地作为一个分类预测模型，因此应选择“满意度<5”作为初始节点进行划分。根节点下面的子节点也是用类似的方法来进行选择。

同理，对于“收入”这一属性变量，是选择“收入<10000”还是选择“收入<50000”进行划分，也是通过礼教上在这两种情况下划分后的基尼系数进行判断。如果还有其他变量，如工龄、月工时等，也是通过类似的方法计算划分后系统的基尼系数来确定如何进行节点的划分，从而构建一个较为完善的决策树模型。

采用基尼系数进行运算的决策树也称为 CART 决策树。

2. 信息熵

除了基尼系数，还有另一个衡量系统混乱程度的经典指标“信息熵”。信息熵的作用与基尼系数基本一致，都是用来衡量系统的混乱程度，从而选择合适的节点进行划分。信息熵 H(X)的公式如下：

$$\mathrm{H(X)} = -\sum_{\mathrm{i=1}}^{n} \mathrm{p}_i \log_2(\mathrm{p}_i)$$

其中，X 表示随机变量，取值为$(X_1, X_2, X_3, ...)$，在 n 分类问题中，便有 n 个取值。例如，在员工离职预测模型案例中，X 的取值就有两个：“离职”和“未离职”；p_i 表示随机变量 X 取值为 X_i 时发生的概率，且 $\sum \mathrm{p}_i = 1$。此外，注意这里的对数函数是以 2 为底的，即 $\log_2(\mathrm{p}_i)$。

同样，对于一个全部都是离职员工的样本而言，其中只有一个类别-离职员工，其出现的概率是 100%，所以该系统信息熵为 $-1 \times \log_2(1) = 0$，表示该系统没有混乱，或者该系统的“纯度”很高，很纯粹。如果一个样本中有一半是离职员工，一半是未离职员工，那么类别个数为 2，每个类别出现的概率都为 50%，所以信息熵为 $-(0.5 \times \log_2(0.5) + 0.5 \times \log_2(0.5)) = 1$，即其混乱程度很高。

当引入某个用于进行分类的变量 A（如“满意度<5”）时，则根据该变量划分后的信息熵也称为条件熵，其公式为：

$$H_A(X)=\frac{S_1}{S_1+S_2}H(X_1)+\frac{S_2}{S_1+S_2}H(X_2)$$

其中，S_1、S_2表示划分成两类后各自的样本量；$H(X_1)$和$H(X_2)$表示划分后的两类各自的信息熵。

与前面计算基尼系数的减少值类似，这里同样需要计算信息熵的减少值（即原系统熵值-划分后的系统熵值），该减少值称为“熵增益”或“信息增益”，该值越大越好，越大表明分类后的混乱程度越低，即分类越准确。信息增益的计算公式如下：

$$Gain(A)=H(X)-H_A(X)$$

假设初始样本中有 1000 名员工，已知其中有 400 人离职，600 人未离职。划分前该系统的信息熵为$-(0.4\times\log_2 0.4+0.6\times\log_2 0.6)=0.97$，可见混乱程度较高。下面采用两种不同的划分方式来决定初始节点：

(1) 根据“满意度<5”进行划分。

(2) 根据“收入<10000”进行划分。

对于第一种方式，以“满意度<5”为根节点进行划分，划分后的信息熵为 0.65，信息增益为 0.32。如图 5-2 所示。

图 5-4　决策树模型原理

对于第二种方式，以“收入<10000”为根节点进行划分，划分后的信息熵为 0.924，信息增益为 0.046。如图 5-2 所示。

图 5-5　决策树模型原理

根据方式一划分后的信息增益为 0.32，大于根据方式二划分后的信息增益 0.046，因此应根据方式一进行决策树的划分，这样能更好地降低系统的混乱程度，从而进行更加合理的分类。这与之前用基尼系数运算得出的最终结论相同。

在决策树模型构建中，因为基尼系数涉及平方运算，而信息熵涉及的是一些复杂的对数函数运算，所以目前决策树模型默认使用基尼系数进行运算，运算速度会较快。

在实际商业应用中数据量通常很大，计算不同情况下的基尼系数或者信息熵不是人力所能完成的，这就需要利用机器不停地训练来找到最佳分裂节点。而在 Spark 中，则有相应的机器学习库来帮助快速建立决策树模型。

5.1.3 Spark 决策树分类器

决策树学习算法既可以用来做分类分析（即预测分类变量值），也可以用来做回归分析（即预测连续变量值）。在 Spark MLlib 库中提供了对决策树学习算法的实现，分别是决策树分类器 DecisionTreeClassifier 和决策树回归器 DecisionTreeRegressor，它们的学习结果（输出）分别是决策树分类模型 DecisionTreeClassificationModel 和决策树回归模型 DecisionTreeRegressionModel。本节主要关注 DecisionTreeClassifier 决策树分类器的使用。

1. 决策树分类器的超参数

DecisionTreeClassifier()模型常用的一些超参数及其解释如下。

(1) criterion：特征选择标准。取值为 entropy（信息熵）和 gini（基尼系数），默认选择 gini。

(2) splitter：取值为 best 和 random，其中 best 是指在特征的所有划分点中找出最优的划分点，适合样本量不大的情况；而 random 是指在部分划分点中随机寻找局部最优的划分点，适合样本量非常大的情况。默认选择 best。

(3) max_depth：决策树最大深度，取值为 int 或 None，一般数据或特征比较少时可以不设置。如果数据或特征比较多，可以设置最大深度进行限制。默认取 None。

(4) min_sample_split：子节点向下划分所需的最小样本数，默认取 2，如果子节点中的样本数小于该值则停止分裂。

(5) min_sample_leaf：叶子节点的最少样本数，默认取 1，如果小于该数值，该叶子节点会与兄弟节点一起被剪枝（即剔除该叶子节点和其兄弟节点，并停止分裂）。

(6) min_weight_fraction_leaf：叶子节点最小的样本权重和，默认取 0，即不考虑权重问题，如果小于该数值，该叶子节点会与兄弟节点一起被剪枝。如果较多样本有缺失值或者样本的分布类别偏差很大，则需要考虑样本权重问题。

(7) max_features：在划分节点时所考虑的特征值数量的最大值，默认取 None，可以传入 int 型或 float 型数据。如果是 float 型数据，则表示百分数。

(8) max_leaf_nodes：最大叶子节点数，默认取 None，可以传入 int 型数据。

(9) class_weight：类别权重，默认取 None，可以取 balanced，代表样本量少的类别所对应的样本权重更高，也可以传入字典指定权重。该参数主要是为防止训练集中某些类别的样本过多，导致训练的决策树过于偏向这些类别。除了此处指定 class_weight，还可以使用过采样和欠采样的方法处理样本类别不均衡的问题。

(10) random_state：决策树算法会优先选择使整个系统基尼系数下降程度最大的划分方式来进行节点划分，但是当数据量较大或特征变量较多时，可能在某个节点划分时出现两个特征变量的信息熵增益或基尼系数减少量一样的情况，此时决策树模型默认从中随机选取一个特征变量进行划分，这样可能导致每次运行程序后生成的决策树不一致。如果设定 random_state 参数（可以设置成 0 或任意数字），则可以保证每次运行代码时各个节点的分裂结果一致，使每次运行的结果相同。这在特征变量较多、树的深度较大时较为重要。

在大多数情况中下，使用模型的默认参数也能取得较好的结果及预测准确度。如果想要精益求精，就需要对模型的超参数进行调优，例如，max_depth 是取 3 还是取默认值 None（默认值情况下不设置最大深度，即分裂到所有叶子节点的基尼系数都为 0）是有考究的。max_depth 取值过小可能会导致模型欠拟合，而取值过大则容易导致模型过拟合，因此便需要采用某种方法来调节这些参数。

2. 决策树的前剪枝和后剪枝

决策树剪枝的目的是防止构建的决策树出现过拟合。决策树剪枝分为前剪枝和后剪枝，两者的定义如下。

(1) 前剪枝：从上向下剪枝，通常利用超参数进行剪枝。例如，限制决策树的最大深度（max_depth）便可以减去该最大深度下面的节点。

(2) 后剪枝：从下向上剪枝，大多是根据业务需求剪枝。例如，在舞弊预测模型中，认为 45%和 50%舞弊概率的两个叶子节点都是高危人群，那么就把这两个叶子节点合并成一个节点。

在实际中，往往前剪枝的应用更加广泛。参数调优其实也起到了一定的前剪枝的作用。

5.1.4 示例：使用决策树预测鸢尾花分类

下面的示例演示了如何加载 Iris 数据集，将其解析为 DataFrame，并使用 Spark ML 中的决策树分类器 DecisionTreeClassifier 执行多类分类。本示例中使用的鸢尾花数据集与 4.3.2 节中的相同。

【示例】使用 Spark ML 中的 DecisionTreeClassifier 决策树算法实现对鸢尾花数据集进行分类。

5.2 朴素贝叶斯分类器

朴素贝叶斯分类器是一类基于贝叶斯定理的简单概率多类分类器，每对特征之间具有很强的(朴素)独立性假设。这是一个监督型的机器学习算法，对一个预测性问题进行概率建模。

朴素贝叶斯算法使用每个属性（特征）属于某个类的概率做出预测，训练模型的过程可以看做是对条件概率的计算，通过计算每个类别的相应条件概率来估计分类结果。这个算法基于一个假设：所有特征相互独立，任意特征的值和其他特征的值没有关联关系。这种假设在实际生活中几乎不存在，但是朴素贝叶斯算法在很多领域，尤其是自然语言处理(NLP)领域很成功，因此它通常用于基于文本的数据集，以学习和理解有关文本的内容。朴素贝叶斯算法的典型应用还有垃圾邮件过滤等等。

注意： NLP 是一个与机器学习密切相关的领域，因为它的许多问题可以被表述为一个分类任务。通常，NLP 问题有大量以文本文档形式标记的数据。这些数据可以作为机器学习算法的训练数据集。

为了使用朴素贝叶斯算法对数据集进行分类，数据必须是线性可分的；也就是说，数据中的类必须能被类边界线性划分。下图通过虚线显示了三个数据集和两个类边界，直观地解释了这一点:

朴素贝叶斯是一种计算效率高、可扩展的算法，只需要对数据集进行一次传递。但它在处理非线性分类问题时表现很差。

5.2.1 贝叶斯定理

18 世纪，托马斯·贝叶斯（Thomas Bayes）牧师提出了最初的研究，后来成为贝叶斯定理，它用来描述两个条件概率之间的关系，比如 P(A|B) 和 P(B|A)。皮埃尔-西蒙·拉普拉斯扩展了贝叶斯的研究，得出了我们今天所知道的结果。

贝叶斯定理可以表述为：

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

我们把 P(A)称为"先验概率"（prior probability），即在 B 事件发生之前，我们对 A 事件概率的一个判断。P(A|B)称为"后验概率"（posterior probability），即在 B 事件发生之后，我们对 A 事件概率的一个判断。

(1) 先验：表示数据对象在观察之前的知识或不确定性的概率分布。

(2) 后验：条件概率分布，表示观察数据对象后可能的参数。

贝叶斯定理实际上就是计算"条件概率"的公式。所谓"条件概率"，就是指在事件 B 发生的情况下，事件 A 发生的概率，用 P(A|B)来表示，读作“在 B 发生的条件下 A 发生的概率”。

根据文氏图，可以很清楚地看到在事件 B 发生的情况下，事件 A 发生的条件概率 P(A|B)就是 P(A∩B)/P(B)。

其中 P(A∩B)表示 A 与 B 的联合概率。联合概率表示两个事件共同发生（数学概念上的交集）的概率。

所以条件概率公式为：

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

因此，

$$P(A \cap B) = P(A \mid B)P(B)$$

同理可得，

$$P(A \cap B) = P(B \mid A)P(A)$$

所以，

$$P(A \mid B)P(B) = P(B \mid A)P(A)$$

即

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

这就是条件概率的计算公式。

5.2.2 贝叶斯推断

对条件概率公式进行变形，可以得到如下形式：

$$P(A \mid B) = P(A)\frac{P(B \mid A)}{P(B)}$$

P(B|A)/P(B)称为"可能性函数"（Likelyhood，似然函数），这是一个调整因子，使得先验概率更接近真实概率。(似然，Likelihood，属于某一特定类别或分类的可能性)

所以，贝叶斯定理可以理解成下面的公式：后验概率 = 先验概率 * 似然。这就是贝叶斯推断的含义。我们先预估一个"先验概率"，然后加入实验结果，看这个实验最终增强或削弱"先验概率"，由此得到更接近事实的"后验概率"。

请注意以下几点：

(1) 如果"可能性函数"P(B|A)/P(B)>1，意味着"先验概率"被增强，事件 A 的发生的可能性变大；

(2) 如果"可能性函数"=1，意味着 B 事件无助于判断事件 A 的可能性；

(3) 如果"可能性函数"<1，意味着"先验概率"被削弱，事件 A 的可能性变小。

5.2.3 全概率公式

由于后面要用到全概率，所以除了条件概率以外，这里还要推导全概率公式。

假定样本空间 S，是两个事件 A 与 A'的和，它们共同构成了样本空间 S，如图 5-所示。

在这种情况下，事件 B 可以划分成两个部分，如图 5-所示。

即：

$$P(B) = P(B \cap A) + P(B \cap A^{'})$$

在上一节的推导当中，我们已知：

$$P(B) = P(B \cap A) + P(B \cap A^{'})$$

所以：

$$P(B) = P(B \mid A)P(A) + P(B \mid A^{'})P(A^{'})$$

这就是全概率公式。它的含义是，如果 A 和 A'构成样本空间的一个划分，那么事件 B 的概率，就等于 A 和 A'的概率分别乘以 B 对这两个事件的条件概率之和。

将这个公式代入上一节的条件概率公式，就得到了条件概率的另一种写法：

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B \mid A)P(A) + P(B \mid A^{'})P(A^{'})}$$

5.2.4 贝叶斯定理应用

朴素贝叶斯分类器广泛应用于文本分类、垃圾邮件过滤、医疗诊断等领域。它是一种简单而强大的算法，可用于快速准确地对数据进行分类。为了进一步理解该算法，下面应用贝叶斯定理回答如下几个问题。

1. 水果糖问题

假设有两个一模一样的碗，一号碗有 30 颗水果糖和 10 颗巧克力糖，二号碗有水果糖和巧克力糖各 20 颗。现在随机选择一个碗，从中摸出一颗糖，发现是水果糖。请问这颗水果糖来自一号碗的概率有多大？

我们假定，H1 表示一号碗，H2 表示二号碗。由于这两个碗是一样的，所以 P(H1)=P(H2)，也就是说，在取出水果糖之前，这两个碗被选中的概率相同。因此，P(H1)=0.5，我们把这个概率就叫做"先验概率"，即没有做实验之前，来自一号碗的概率是 0.5。

再假定，E 表示水果糖，所以问题就变成了在已知 E 的情况下，来自一号碗的概率有多大，即求 P(H1|E)。我们把这个概率叫做"后验概率"，即在 E 事件发生之后，对 P(H1)的修正。

根据条件概率公式，得到：

$$P(H_1|E)=P(H_1)\frac{P(E|H_1)}{P(E)}$$

已知，P(H1)等于 0.5，P(E|H1)为一号碗中取出水果糖的概率，等于 0.75，那么求出 P(E)就可以得到答案。根据全概率公式，

$$P(E)=P(E|H_1)P(H_1)+P(E|H_2)P(H_2)$$

所以，

$$P(E)=0.75\times0.5+0.5\times0.5=0.625$$

将数字代入原方程，得到

$$P(H1|E)=0.5\times\frac{0.75}{0.625}=0.6$$

这表明，来自一号碗的概率是 0.6。也就是说，取出水果糖之后，H1 事件的可能性得到了增强。

2. 文本分类问题

假设有两个人，张三和李四。张三经常挂在口头的话是"ok, supprise, good"，李四经常挂在口头的话是"shit, ok, good"。现在我们偶尔听到两个人中有人说"ok good"，请问，是张三说的可能性有多大？

张三和李四之间的概率分布是对半分割，即各占 50%（即先验概率）。

$$P(\text{张三})=0.5$$

$$P(\text{李四})=0.5$$

张三和李四都有 3 个词，每个词下面有一个频率加权：这个人使用这个词的频率。例如，张三用户 50%的时候会说“ok”，40%的时候会说“surprise”，最后 10%的时候会说“good”。

根据我们所知道的，我们现在需要找出每个人说“ok good”这个短语的概率。这里我们用"E"表示"ok good"短语，则：

$$P(E\,|\,\text{张三})=0.5\times0.5\times0.1=0.025$$
$$P(E\,|\,\text{李四})=0.5\times0.2\times0.1=0.01$$

根据条件概率公式，

$$P(\text{张三}\,|\,E)=P(\text{张三})\frac{P(E\,|\,\text{张三})}{P(E)}$$

用全概率公式改写分母，则

$$P(\text{张三}\,|\,E)=P(\text{张三})\frac{P(E\,|\,\text{张三})}{P(E\,|\,\text{张三})P(\text{张三})+P(E\,|\,\text{李四})P(\text{李四})}$$

将数字代入，

$$P(\text{张三}\,|\,E)=0.5\times\frac{0.025}{0.025\times0.5+0.01\times0.5}\approx0.71$$

可以看到，张三说这个短语（"ok good"）的概率是 71%，相应地，李四说这个短语（"ok good"）的概率是 29%。这里没有被考虑顺序或单词组合，只有每个单词和它的频率。这就是名称“Naive（朴素）”背后的原因：因为每个特征（在本示例中是 word）都被独立地对待。

3. 假阳性问题

已知某种疾病的发病率是 0.001，即 1000 人中会有 1 个人得病。现有一种试剂可以检验患者是否得病，它的准确率是 0.99，即在患者确实得病的情况下，它有 99%的可能呈现阳性。它的误报率是 5%，即在患者没有得病的情况下，它有 5%的可能呈现阳性。现有一个病人的检验结果为阳性，请问他确实得病的可能性有多大？

假定 A 事件表示得病，那么 P(A)为 0.001。这就是"先验概率"，即没有做试验之前，我们预计的发病率。再假定 B 事件表示阳性，那么要计算的就是 P(A|B)。这就是"后验概率"，即做了试验以后，对发病率的估计。

根据条件概率公式，

$$P(A\,|\,TP)=P(A)\frac{P(TP\,|\,A)}{P(TP)}$$

用全概率公式改写分母，

$$P(A\,|\,TP)=P(A)\frac{P(TP\,|\,A)}{P(TP\,|\,A)P(A)+P(TP\,|\,\overline{A})P(\overline{A})}$$

将数字代入，

$$P(A\,|\,TP)=0.001\times\frac{0.99}{0.99\times0.001+0.05\times0.999}\approx0.019$$

我们得到了一个惊人的结果，P(A|TP)约等于 0.019。也就是说，即使检验呈现阳性，病人得病的概率，也只是从 0.1%增加到了 2%左右。这就是所谓的"假阳性"，即阳性结果完全不足以说明病人得病。

为什么会这样？为什么这种检验的准确率高达 99%，但是可信度却不到 2%？答案是与它的误报率太高有关。

注意： 如果误报率从 5%降为 1%，请问病人得病的概率会变成多少？请自行练习计算。

有兴趣的朋友，还可以算一下"假阴性"问题，即检验结果为阴性，但是病人确实得病的概率有多大。然后问自己，"假阳性"和"假阴性"，哪一个才是医学检验的主要风险？

3. 礼品卡欺诈问题

下面这个示例使用朴素贝叶斯算法发现虚假订单。假设我们注意到欺诈性订单经常使用礼品卡和多个促销代码。利用这些知识，我们如何确定什么是欺诈，什么不是欺诈？也就是说，我们如何计算在购买者使用礼品卡的情况下发生欺诈的可能性?

在我们的欺诈示例中，假设我们想要度量使用礼品卡的订单的欺诈概率。这将是：

$$P(\text{欺诈}\,|\,\text{礼品卡}) = \frac{P(\text{欺诈} \cap \text{礼品卡})}{P(\text{礼品卡})}$$

推导贝叶斯定理过程：

$$P(B \mid A) = \frac{\frac{P(A \cap B)P(B)}{P(B)}}{P(A)} = \frac{P(A \cap B)}{P(A)}$$

$$P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A)}$$

利用贝叶斯定理，我们现在计算:

$$P(\text{欺诈}\,|\,\text{礼品卡}) = \frac{P(\text{礼品卡}\,|\,\text{欺诈})P(\text{欺诈})}{P(\text{礼品卡})}$$

根据统计，欺诈的概率是 10%。假设礼品卡被使用的概率是 10%，根据我们的研究在欺诈订单中使用礼品卡的概率是 60%。那么，在使用礼品卡的情况下，订单被欺诈的概率是多少?

$$P(\text{欺诈}\,|\,\text{礼品卡}) = \frac{60\%*10\%}{10\%} = 60\%$$

5. 2. 5 Spark 贝叶斯分类器

朴素贝叶斯分类器通过计算给定一组特征的每个类标签的概率来工作。然后，它使用最大后验估计（MAP，maximum a posteriori）以最高的概率分配类标签。MAP 估计基于所有特征彼此独立的假设，这就是为什么它被称为“朴素”分类器。

朴素贝叶斯可以非常有效地训练。通过对训练数据的一次遍历，它计算给定每个标签的每个特征的条件概率分布。对于预测，它应用贝叶斯定理计算给定观测值的每个标签的条件概率分布。Apache Spark MLlib 的朴素贝叶斯分类器是一个强大的机器学习工具，可用于根据数据进行预测。

朴素贝叶斯分类器用于分类和回归任务。它是情感分析和垃圾邮件过滤等文本分类任务的流行选择。它还用于医疗诊断、预测客户流失和其他任务。朴素贝叶斯分类器易于实现，计算效率高。它需要最少的训练数据，并且能够处理大型数据集。它对噪声数据也具有鲁棒性，并且不受无关特征的影响。

要使用朴素贝叶斯分类器，必须对数据进行预处理并将其转换为算法可以使用的格式。这包括将数据分割为训练集和测试集，将分类变量转换为数值变量，以及对数据进行规范化。

一旦数据准备好了，朴素贝叶斯分类器就可以用来进行预测。该算法利用贝叶斯定理计算给定输入特征的每一类的概率。然后选择概率最高的类作为预测。

Spark 机器学习库支持多项朴素贝叶斯、补朴素贝叶斯、伯努利朴素贝叶斯和高斯朴素贝叶斯。

多项朴素贝叶斯、补朴素贝叶斯、伯努利朴素贝叶斯通常用于文档分类。在这个上下文中，每个观察都是一个文档，每个特征代表一个术语。特征的值是术语的频率(在多项式朴素贝叶斯或补朴素贝叶斯中)，或者是表示该术语是否在文档中找到的 0 或 1(在伯努利朴素贝叶斯中)。多项式和伯努利模型的特征值必须是非负的。对于模型类型的选择，使用可选参数“multinomial”、“complement”、“bernoulli”或“gaussian”，默认为“multinomial”。对于文档分类，输入特征向量通常应该是稀疏向量。由于训练数据只使用一次，所以没有必要缓存它。

可以通过设置参数（默认为 1.0）来使用加性平滑（additive smoothing）。

注意： 在统计学中，加性平滑,也称为拉普拉斯平滑或 Lidstone 平滑，是一种用于平滑分类数据的技术。

5.2.6 示例：Spark 朴素贝叶斯分类器

朴素贝叶斯假设数据集中的特征(或维度)是相互独立的；也就是说，它们彼此之间没有影响。下面的示例使用朴素贝叶斯算法实现对鸢尾花数据集的多分类任务。

【示例】使用朴素贝叶斯算法实现对鸢尾花数据集的多分类任务。

鸢尾花数据集的字段说明如表 5-所示。

表 5-1 鸢尾花数据集字段说明

序号	字段	数据类型	字段说明
1	SepalLength	float	花萼长度
2	SepalWidth	float	花萼宽度
3	PetalLength	float	花瓣长度
4	PetalWidth	float	花瓣宽度
5	Class	int	标签列，类别变量。其中，0 代表 Iris Setosa，1 代表 Iris Versicolor，2 代表 Iris Virginica

1）读取数据集

首先，加载数据集，代码如下：

```

```

2）探索性数据分析

接下来，执行探索性分析。先查看数据规模和数据模式，代码如下：

```

```

执行以上代码，输出内容如图 4-所示：

3）分割训练集和测试集

将数据集随机分为训练集(70%)和测试集(30%)。训练集将用于构建我们的模型，测试集将用于评估模型。代码如下：

4）创建机器学习管道

首先定义机器学习管道中的 Estimator 和 Transformer。其中使用 StringIndexer 将其编码为 double 类型，并存储到名为 label 的新列中；使用 VectorAssembler transformer 将这些特征合并到单个的向量列；创建一个朴素贝叶斯多类分类器。请记住，朴素贝叶斯算法假定特征之间是独立的，并且要求特征取非负值。代码如下：

注意，代码中朴素贝叶斯分类器的 smoothing 参数指的是拉普拉斯平滑，用来解决某个文本在训练集中没有出现但在测试集中出现导致概率为零的问题。另外，模型类型支持的选项有"multinomial"、"complement"、"bernoulli"和"gaussian"，默认是"multinomial"。

5）训练模型

定义好管道后，使用其拟合训练数据集，代码如下：

```
val pipelineModel = pipeline.fit(trainDF)
```

执行以上代码，返回的是学习后的管道模型 PipelineModel。

6）模型评估

现在可以从模型进行预测并查看结果。对测试数据进行预测，这样我们就可以测量模型对新数据的准确性。代码如下：

7）模型调优

前面训练模型时采用的拉普拉斯平滑超参数值为 1.0。接下来我们通过参数风格和交叉验证方法实验各种平滑参数，以找出最优模型（参数集）来。代码如下：

8）模型持久化

可以将调优后的模型存储起来，并在稍后使用的时候将其加载进来。以下代码演示了这种用法：

5.3 多层感知机分类器

多层感知机是一个前馈人工神经网络，它由几个完全连接的层节点组成。输入层中的节点对应于输入数据集。在学习多层感知机分类器之前，首先来了解什么是感知机。

5.3.1 感知机算法原理

众所周知，深度神经网络的最基本单元被称为人工神经元/感知机。我们今天使用的感知机来自 1943 年由麦卡洛克和皮茨对生物神经元的功能的模仿。

1. 生物神经元

基本上，一个神经元接受输入信号（树突），像 CPU（细胞体）一样处理它，将输出信号通过电缆状结构传递给其他连接的神经元（轴突到突触，再到其他神经元的树突）。在更高的层面上，这就是我们大脑中的神经元所发生的事情—接受输入，处理它，然后输出。

图 5-1　生物神经元

在图 5-1 中，各部分的功能如下：

(1) 树突：接收来自其他神经元的信号。

(2) 细胞体：处理输入的信息。

(3) 轴突：传递这个神经元的输出。

(4) 突触：连接其他神经元的点。

我们的感觉器官与外部世界相互作用，并将视觉和声音信息发送给神经元。假设你正在看《老友记》。现在，你的大脑接收到的信息被“笑或不笑”神经元吸收，这将帮助你决定是否要笑。每个神经元只有在满足各自的标准时才会被触发/激活，如下所示。

麦卡洛克和皮茨将这种神经细胞描述为具有二进制输出的简单逻辑门。

2. MCP 神经元

第一个神经元的计算模型是由神经学家 Warren MuCulloch（沃伦·麦卡洛克）和逻辑学家 Walter Pitts（沃尔特·皮茨）在 1943 年提出的。这被称为 McCullock-Pitts（MCP）神经元。

它可以分为 2 部分。第一部分，g 接受一个输入（类似树突），执行聚合；第二部分，基于聚合值，f 做出决策。

M-P 神经元的一个简明表示如图 5-所示。

图 5-表示，对于布尔输入 x_1、 x_2 和 x_3，如果 g(x)，即求和 sum≥θ，神经元就会触发，否则就不会。

注意我们的输入都是布尔值输出也是布尔值，所以本质上，神经元只是在学习一个布尔函数。基于适当的输入变量，比如是否继续阅读这篇文章，看完这篇文章后是否看《老友记》等，很多布尔决策问题都可以用 M-P 神经元来表示。

例如，用 M-P 神经元来表示 AND 函数，以及其几何意义，如图 5-所示。

用 M-P 神经元来表示 OR 函数，以及其几何意义，如图 5-所示。

麦卡洛克·皮特的神经元模型是人类第一次尝试模仿人脑。因此它有很多局限性：

(1) 该模型无法捕获和计算非二元输入的情况。它受限于只能计算 0 和 1 的所有情况。

(2) 阈值必须事先确定，并且需要人工计算，而不是模型自己决定。

(3) 线性可分函数无法计算。

3. 人工神经元

人工神经元（感知机）是 Frank Rosenblatt 在 1957 年提出的，作为一种能够学习和执行二元分类任务的监督学习算法。Frank Rosenblatt 是一位心理学家和计算机科学家，他提出了一种基于原始 MCP 神经元的感知机学习规则。感知机与 MCP 的区别如下：

(1) McCulloh/Pitt 的模型只接受布尔输入，而感知机模型可以处理各种实际形式的输入。

(2) 在 McCulloh/Pitt 的模型中，输入没有加权，这意味着这个模型不灵活，然而与这个模型相比，感知机模型接受相对于提供的输入的权重，这使得它更加灵活。

感知机算法是基于一个简单的计算单元的概念，它接受一个或多个输入（样本的特征向量），并产生一个单一的输出（样本的类别），模仿大脑中神经元的结构和功能。如图 5-所示。

图 5-1　人工神经元

人工神经元（感知机）是一种基于生物神经元模型的数学函数，其中每个神经元接受输入，分别对它们进行加权，将它们相加，并将这些总和传递给非线性函数以产生输出。感知器学习规则指出算法将自动学习最优权重系数。然后将输入特征与这些权重相乘，以确定神经元是否触发。在感知器学习规则中，将预测输出与已知输出进行比较。如果不匹配，则将错误向后传播以允许进行权重调整。

图 5-1　感知机规则

感知器被设计成能够从样本中学习并调整其参数以提高其分类新样本的准确性，其基本组成部分包括：

(1) 输入层：输入层由一个或多个输入神经元组成，接收来自外部世界或来自神经网络其他层的输入信号。

(2) 权重：每个输入神经元都有一个权重，权重表示输入神经元和输出神经元之间的连接强度。

(3) 偏差：在输入层中添加一个偏差（bias）项，为感知器在建模输入数据中的复杂模式时提供额外的灵活性。 偏差类似于线性方程中的截距。它是一个附加参数，用于调整输出和神经元输入的加权和。

(4) 激活函数：激活函数根据输入和偏置项的加权和确定感知器的输出。感知器中常用的激活函数包括阶跃函数、sigmoid 函数和 ReLU 函数。

(5) 输出：感知器的输出是一个单一的二进制值，0 或 1，表示输入数据所属的类或类别。

(6) 训练算法：感知器通常使用监督学习算法进行训练，例如感知器学习算法或反向传播算法。在训练过程中，感知器的权重和偏差被调整，以最小化给定训练示例集的预测输出与真实输出之间的误差。

感知器模型首先将所有输入值及其权重相乘（请注意，输入的权重表示节点的强度），然后将这些值相加以创建加权和。进一步，将这个加权和应用于激活函数 f 以获得所需的输出。这个激活函数也被称为阶跃函数，用 f 表示。这个阶跃函数或激活函数对于确保输出映射到(0,1)或(-1,1)之间至关重要。

注意： 作为一种二元分类器的监督学习算法，我们也可以把感知机看作一个单层神经网络，它有四个主要参数:输入值、权重和偏置、净和和一个激活函数。感知机是一个单层的神经网络，或者我们可以说神经网络是一个多层的感知机。

感知机接收多个输入信号，如果输入信号的总和超过一定的阈值，它要么输出信号，要么不返回输出。在监督学习和分类的背景下，这可以用来预测样本的类别。感知机使用的激活函数是阈值阶跃函数。阶跃函数在神经元输出的某个值以上被触发；否则输出为 0。除了阶跃函数外，激活函数还可以使用符号函数输出+1 或-1（取决于神经元输出是否大于零），以及 Sigmoid 函数，它是 S 型曲线，输出的值在 0 到 1 之间。这三种激活函数如图 5-所示。

5.3.2 感知机分类实现

首先看一个分类的例子。

1. 分类示例过程

假设我们是一所高校的招生办人员，工作就是接受或拒绝申请入学的学生。我们可以利用两方面的信息来评估提出入学申请的学生，即入学测试成绩和在校期间的平时成绩（假设均为 10 分制）。表 5-列出了录取和拒绝录取的学生信息。

表 5-1 入学申请信息和录取信息

学生编号	入学测试成绩	在校平时成绩	是否录取
学生 1	9 分	8 分	录取
学生 2	3 分	4 分	拒绝录取

假设学生 3 的入学测试成绩是 7 分，平时成绩是 6 分。请问，我们是否录取学生 3？

我们在下图中表示过往学生们的成绩和录取情况。坐标系 x 轴表示入学测试成绩，y 轴表示在校平时成绩。图中实心圆点表示录取，空心圆圈表示拒绝录取。

我们可以在这个二维平面上划一条线，对这两类学生做个区分，如图 5-所示。可以看出，学生 3 会被录取。

问题的关键在于，我们怎么找出这条直线（边界线）呢？假设在坐标系中标记横轴为 x1，纵轴为 x2，那么这个线性边界线可以用下图中的线性方程表示：

把图 5-中的方程 $2x_1 + x_2 - 18$ 的结果当作评分：Score = 2Test + Grades -18（Test 代表测试成绩，Grades 代表平时成绩，Score 代表总评分）。每当输入一个学生的成绩，我们就代入这个方程计算其得分。如果得分 Score>0，则接受入学申请；如果得分 Score<0，则拒绝入学申请。这称为“预测”。这里得到的线性方程 $2x_1 + x_2 - 18 = 0$，就是我们的模型。

将这个线性方程（模型）进行泛化，可以得到泛化模型：$w_1x_1 + w_2x_2 + b = 0$。将其进一步向量简化，得到模型：$Wx + b = 0$。其中 $W = (w_1, w_2)$，即 W 代表权重向量；$x = (x_1, x_2)$，即 x 代表输入的特征向量。

考虑一下，如果我们在考虑学生的入学申请时，不只是考虑入学测试成绩和在校平时成绩，还需要考虑其他因素，比如该学生在班级的排名，这时该怎么办？

这个时候需要在三个数据列（多维度）中来拟合数据。这时的边界不再是一条线，而是一个平面，如图 5-所示。

该界面方程可表示为：$w_1x_1 + w_2x_2 + w_3x_3 + b = 0$。对其进行向量简化，得到的简化方程如下：$Wx + b = 0$。其中 $W = (w_1, w_2, w_3)$，即 W 代表权重向量；$x = (x_1, x_2, x_3)$，即 x 代表输入的特征向量。

如果有更多列呢？比如 n 列（n 维特征）。方法是相同的，这时数据分布在 n 维空间 $(x_1, x_2, x_3, ...x_n)$ 中，如表 5-所示，其中 y 为标签列，1 代表录取，0 代表拒绝录取。

表 5-1 当数据分布在 n 维空间时

	x_1 (测试成绩)	x_2 (班级排名)	x_3 (平时成绩)		x_n (论文答辩)	y(是否录取)
学生 1	9	6	5	...	6	1
学生 2	8	4	8	...	3	0
...	...	...	...	...	...	...
学生 n	6	7	2	...	8	1

然后分界线为 n-1 维的超平面，相当于二维的直线和三维的平面。该超平面方程可表示为：$w_1x_1 + w_2x_2 + ... + w_nx_n + b = 0$。对其进行向量简化，得到的简化方程如下：$Wx + b = 0$。

其中$W=(w_1,w_2,...,w_n)$，即 W 代表权重向量；$x=(x_1,x_2,...,x_n)$，即 x 代表输入的特征向量，包含 n 项。

2. 感知机学习过程

从上面的示例过程可知，线性方程$Wx+b=0$对应于 n 维特征空间中的一个超平面 S，其中 W 是超平面的法向量，b 是超平面的截距。这个超平面将特征空间分为了两个部分。位于两部分的点（即特征向量）分别被分为正、负两类。所以，超平面 S 也被称之为分离超平面（separating hyperplane）。

感知机学习是有监督学习过程。感知机算法通过学习输入信号的权重，通过有标签的训练数据集求模型的参数 w 和 b，以绘制线性决策边界。例如，针对前面的学生入学申请预测示例，我们使用感知机算法构造一种二元分类器。当学生 3 提交入学申请时，它要做的是把该学生的成绩(7,6)绘制到二维平面图中，并检查这个点位于正区域还是负区域。 如果位于正区域，就返回 yes（录取）；如果位于负区域，就返回 no（拒绝录取）。如图 5-所示。

这个线性边界的方程式为：Score = 2*Test + 1*Grade－18。其中权重为 2、1 和-18，它们定义了这个线性方程。我们使用它们作为标记，标记了节点。把偏差考虑成输入的一部分，把偏差 B 标记为 1。可以把输入节点的值，乘以对应边的值（权值），然后相加得到最终结果值。最后检查结果是否大于等于 0。如果是，那么节点返回 yes（录取）；如果否，那么节点返回 no（拒绝录取）。这时可用图 5-表示。

可以将上面的方法泛化为如下形式：使用一个阶跃函数将返回的结果 yes 或 no 转换为 1 或 0。可以用以下图形进行表示：在第一个节点中使用求和符号西格玛代表线性函数，在第二个节点中绘图表示一个阶跃函数。如图 5-所示。

感知机模型简单、易学习，是基本的机器学习模型，很多机器学习模型（如逻辑回归模型、支持向量机、前馈神经网络（多层感知机）、线性判别分析等）是在感知机模型的基础上变化扩展而得到的。

3. 感知机模型限制

对于具有线性可分类的数据集，感知器总是可以使用决策线（用于 2 维空间）、决策平面（用于 3 维空间）或决策超平面（用于 n 维空间）来解决分类问题。适当的突触权值可以通过训练感知器获得。然而，感知器正常工作的一个假设是两个类应该是线性可分的，即两个类应该彼此充分分离。否则，如果类别是非线性可分的，则感知器无法解决分类问题。

要得到正确的模型，感知机要求数据集本身线性可分：

(1) 在二维平面上，线性可分意味着能用一条直线将正、负样本分开；

(2) 在三维空间中，线性可分意味着能用一个平面将正、负样本分开；

(3) 在 n 维空间中，线性可分意味着能用 n-1 维超平面将正、负样本分开。

感知机模型由于其简单，在应用时也有一定的局限，主要体现在以下几点：

(1) 感知机只能处理线性可分的数据集，线性不可分的数据集会引起分离超平面的震荡而无法收敛。

(2) 感知机是误分类驱动的模型，模型的解不唯一。离群点对结果影响很大。

(3) 感知机的判别模型由符号函数给出，无法表示更复杂的非线性映射。

5.3.3 多层感知机算法

继续之前的学生申请入学预测示例。假设现在有学生 4 申请入学，他的入学测试成绩为 9 分，平时成绩为 1 分。那么我们是录取学生 4 呢？还是拒绝录取他？

按照上一节的感知机算法，学生 4 位于正区域，所以他会被录取。如图 5-所示。

但是实际情况是，如果平时成绩很低，那么不管入学测试分数有多高，该生都不应该被录取；同样地，如果学生平时成绩很好，但入学测试成绩低于 6 分，也不应该被录取。如图 5-所示。

但是我们遇到了一个问题：这些数据无法被一条直线分隔开来，即这些数据不是线性可分的。例如，对于下面图 5-所示的这个例子，其中包含了一些无法线性区分的数据。我们无法用直线区分这些红点和蓝点。这种情况下，使用曲线是较为可行的一种解决方案。我们需要开发更加复杂的算法来解决这个问题。

简而言之，对于这些无法用直线区分的数据，我们将创建一个概率函数，其中实心圆圈区域的点，更有可能是实心圆点，空心圆圈区域的点，更有可能是空心圆点。这条分界曲线上的点，成为实心圆点或空心圆点的概率相同。

这些概念和之前都一样，只是我们要寻找的决策边界将不再是一个线性方程了。例如，下面的图 5-中，左侧为二值分类的数据集，我们希望找到一条曲线，将两个类别如右侧所示分隔开来。

我们可以把两个线性模型组合成一个非线性模型。这里是将两个感知机组合成第三个更复杂的感知机，来构建多层感知机（也叫做神经网络）。

假设我们有下面这样两个线性模型，用相应的感知机表现出来，如图 5-所示。

我们可以使用感知机，利用线性方程把这两个模型组合起来，如图 5-所示。

当把两个感知机合并在一起时，就得到了多层感知机（MLP，Multilayer Perceptron）。当把感知机看成单个神经元时，该神经元的堆叠就形成了神经网络。如图 5-所示。

利用符号法绘制，在节点内部增加了一个偏差，如图 5-所示。

多层感知机（神经网络）具有某些特殊的层级结构：

(1) 第一层称为输入层，它包含了输入数据，这个例子中的输入就是 x_1 和 x_2。

(2) 第二层称为隐藏层，也就是针对输入层的一系列线性模型。

(3) 最后一层是输出层，多个线性模型组合成一个非线性模型。

如图 5-所示。

在多层感知机模型中，输入层到隐藏层以及隐藏层之间使用 sigmoid 激活函数，用于隐藏层神经元输出，取值范围为(0,1)，它可以将一个实数映射到(0,1)区间，可以用来做二分类。隐藏层到输出层可以看成是一个多类别的逻辑回归，使用 softmax 激活函数，返回分类类别的概率。例如，二分类的多层感知机模型如图 5-所示。

5.3.4 Spark 多层感知机分类器

多层感知机分类器 MLPC（Multilayer Perceptron Classifier），是一种基于前馈人工神经网络（ANN）的分类器。

注意： 所谓前馈型神经网络，指其从输入层开始只接收前一层的输入，并把计算结果输出到后一层，并不会给前一层有所反馈，整个过程可以使用有向无环图来表示。该类型的神经网络由三层组成，分别是输入层、一个或多个隐层、输出层。

MLPC 由多层节点组成。每一层都与网络中的下一层全连接。输入层中的节点表示输入数据，对应于输入数据集。中间层中的节点使用 sigmoid 函数，而最终输出层中的节点使用 softmax 函数来支持多类分类。输出层中的节点数量必须与类的数量匹配。所有节点通过输入与节点权重 w 和偏差 b 的线性组合并应用一个激活函数将输入映射到输出。

MLPC 采用反向传播（BP，Back Propagation）算法学习优化模型。BP 算法的学习目的是对网络的连接权值进行调整，使得调整后的网络对任一输入都能得到所期望的输出。BP 算法名称里的反向传播指的是该算法在训练网络的过程中逐层反向传递误差，逐一修改神经元间的连接权值，以使网络对输入信息经过计算后所得到的输出能达到期望的误差。

Spark 中目前仅在 org.apache.spark.ml.classification 包中支持此种分类算法，其实现类名为 MultilayerPerceptronClassifier。Spark 的多层感知器隐层神经元使用 sigmoid 函数作为激活函数，输出层的节点使用 softmax 函数。输出层的节点的数目表示分类器有几类。使用逻辑损失函数进行优化，并使用 L-BFGS 进行优化。

MLPC 可调的几个重要参数解释如下：

(1) featuresCol：输入数据 DataFrame 中指标特征列的名称。

(2) labelCol：输入数据 DataFrame 中标签列的名称。

(3) layers：这个参数是一个整型数组类型，第一个元素需要和特征向量的维度相等，最后一个元素需要训练数据的标签数相等，如 2 分类问题就写 2。中间的元素有多少个就代

表神经网络有多少个隐层，元素的取值代表了该层的神经元的个数。例如 val layers = (5,6,5,2)。

(4) maxIter：优化算法求解的最大迭代次数。默认值是 100。

(5) predictionCol：预测结果的列名称。

(6) tol：允许误差，一般取 0.001~0.00001，当迭代结果的误差小于该值时，结束迭代计算，给出结果。

MLPC 对数据源有严格要求，只能是以下两种：

(1) DataFrame。使用 DataFrame 作为数据源时必须指定 DataFrame 中的标签列和特征列。

(2) LIBSVM 格式文本文件。

5.3.5 示例：Spark 多层感知机分类器

这一节通过一个示例展示如何使用 Spark 多层感知机分类器来完成分类任务。

本示例中使用 Spark 自带的样本数据集 sample_multiclass_classification_data.txt。该数据集以 libsvm 格式存储可用于分类和回归的样本数据。libsvm 是一种机器学习中常见的数据保存格式，它有如下数据格式：

```
[label] [index1]:[value1] [index2]:[value2] …
[label] [index1]:[value1] [index2]:[value2] …
```

每一行的说明如下：

(1) label：标签列（目标值），就是类别的标识，通常是一些整数。可以自己随意定，比如-10，0，15。在分类问题里通常为［0，1］。

(2) index：是有顺序的索引，通常是连续的整数。就是指特征编号，必须按照升序排列。

(3) value：就是特征值，用来训练的数据，通常是一堆实数组成。

可理解为如下格式：

```
目标值    第一维特征编号：第一维特征值    第二维特征编号：第二维特征值 …
目标值    第一维特征编号：第一维特征值    第二维特征编号：第二维特征值 …
……
目标值    第一维特征编号：第一维特征值    第二维特征编号：第二维特征值 …
```

Spark 自带的样本数据集 sample_multiclass_classification_data.txt 位于 Spark 安装目录的 data/mllib 文件夹下。使用如下命令查看数据格式：

```
$ head -10 sample_multiclass_classification_data.txt
```

执行以上命令，输出内容如下：

```
1 1:-0.222222 2:0.5 3:-0.762712 4:-0.833333
1 1:-0.555556 2:0.25 3:-0.864407 4:-0.916667
1 1:-0.722222 2:-0.166667 3:-0.864407 4:-0.833333
1 1:-0.722222 2:0.166667 3:-0.694915 4:-0.916667
0 1:0.166667 2:-0.416667 3:0.457627 4:0.5
1 1:-0.833333 3:-0.864407 4:-0.916667
2 1:-1.32455e-07 2:-0.166667 3:0.220339 4:0.0833333
2 1:-1.32455e-07 2:-0.333333 3:0.0169491 4:-4.03573e-08
1 1:-0.5 2:0.75 3:-0.830508 4:-1
0 1:0.611111 3:0.694915 4:0.416667
```

使用 wc 命令查看文件中的数据行数，命令如下：

```
$ wc -l sample_multiclass_classification_data.txt
```

执行以上命令，输出内容如下：

```
150  sample_multiclass_classification_data.txt
```

以上输出内容表明，该数据集中的记录数有 150 行。

【例】Spark 多层感知分类机应用示例。

首先，导入相关的类，并加载数据源。代码如下：

```

```

执行以上代码，输出结果如下：

```
accuracy = 0.9523809523809523
```

5.4 线性支持向量机分类器

支持向量机（Support Vector Machines，SVM）是最流行的监督学习算法之一，既可以用于回归任务，也可以用于分类任务。但一般来说，它们在机器学习中的分类问题上效果最好。

5.4.1 支持向量机算法简介

支持向量机算法是一个有监督的机器学习问题，它的目标是创建最佳的线或决策边界，可以将 n 维空间划分为不同类别，以便我们将来可以轻松地将新的数据点放在正确的类别中。这个最佳决策边界称为超平面。

注意： 不要混淆 SVM 和逻辑回归。这两种算法都试图找到最佳的超平面，但主要区别在于逻辑回归是一种概率方法，而支持向量机是基于统计方法。

支持向量机选择有助于创建超平面的极值点/向量，这些是离超平面最近的点。在这些数据点的帮助下，将定义一条分隔线。支持向量机通过找到超平面之间的最大边界来实现这一点，这意味着两个类之间的最大距离。这些极端情况被称为支持向量（support vectors），因此算法被称为支持向量机。考虑下面的图 5-，其中有两个不同的类别使用决策边界或超平面进行分类：

在 n 维空间中可以有多条线/决策边界来隔离类，但我们需要找到有助于对数据点进行分类的最佳决策边界。这种最佳边界称为支持向量机的超平面。超平面的维度取决于数据集中存在的特征，这意味着如果有 2 个特征，那么超平面将是一条直线。如果有 3 个特征，那么超平面就是一个二维平面。我们总是创建一个有最大边距的超平面，这意味着数据点之间的最大距离。

离超平面最近且影响超平面位置的数据点或向量称为支持向量（Support vector）。由于这些向量支持超平面，因此称为支持向量（Support vector）。

例如，假设我们看到一只奇怪的猫，它也有一些狗的特征，那么如果我们想要一个能准确识别它是猫还是狗的模型，那么可以使用 SVM 算法来创建这样的模型。我们将首先用大量的猫和狗的图像来训练我们的模型，这样它就可以了解猫和狗的不同特征，然后我们用这个奇怪的生物来测试它。因此，当支持向量在这两个数据（猫和狗）之间创建决策边界并选

择极端情况（支持向量）时，它将看到猫和狗的极端情况。在支持向量的基础上，将其分类为 cat。考虑下面的图表：

支持向量机（SVM）算法可用于人脸检测、图像分类、文本分类等。SVM 有两种类型：

(1) 线性支持向量机：线性支持向量机用于线性可分的数据，即如果一个数据集可以通过一条直线分为两类，那么这种数据被称为线性可分数据，使用的分类器被称为线性支持向量机分类器。

(2) 非线性支持向量机：非线性支持向量机用于非线性分离的数据，即如果一个数据集不能用直线进行分类，则该数据称为非线性数据，所使用的分类器称为非线性支持向量机分类器。

SVM 最大的优点之一是，它在处理高维空间时非常有效，也就是说，在处理有很多特征需要学习的问题时非常有效。当数据稀疏时，它们也非常有效（例如一个实例很少的高维空间）。此外，它们在内存存储方面非常有效，因为学习空间中只有一小部分点用于表示决策面。

当然，SVM 也存在一些缺点。虽然 SVM 模型在训练过程中计算量非常大，但是它并没有返回一个数值指标来显示对预测的信心程度。然而，我们可以使用一些技术，如 K-fold 交叉验证来避免这种情况，不过以增加计算成本为代价。

5.4.2 线性支持向量机

通过实例可以理解 SVM 算法的工作原理。假设我们有一个数据集，它有两个标签（方块和圆点），数据集有两个特征 x1 和 x2。我们想要一个分类器，它可以将坐标对(x1, x2)分类为绿色或蓝色。考虑下面的图片:

因为这是二维空间所以只要用一条直线，我们就可以很容易地把这两类分开。现在的问题是它会选择哪个超平面?可以有无数个超平面经过一个点，并将这两个类完美地分类。那么，哪一个是最好的呢？考虑下面的图片：

支持向量机通过找到超平面之间的最大边界来实现这一点，这意味着两个类之间的最大距离。因此，SVM 算法有助于找到最佳分界线或决策边界；这种最佳边界或区域称为超平面（hyperplane）。SVM 算法从两个类中找到最近的点，这些点被称为支持向量（support vectors）。向量和超平面之间的距离称为边距（margin），它是超平面和最接近超平面的观测值（支持向量）之间的距离。支持向量机的目标就是最大化这个边距。具有最大边距的超平面称为最优超平面（optimal hyperplane）。

最优超平面是与两个类之间的距离最大的平面，这是支持向量机的主要目标。这是通过寻找不同的超平面来完成的，这些超平面以最好的方式对标签进行分类，然后它会选择离数据点最远的那个或有最大边界的那个。

5.4.3 非线性支持向量机

如果数据是线性排列的，那么我们可以用一条直线将其分开，但对于非线性数据，我们不能画一条直线。考虑下面的图片：

SVM 最有趣的特征是它甚至可以处理非线性数据集。为了分离这些数据点，我们需要增加一个维度。对于线性数据，我们已经使用了两个维度 x 和 y，所以对于非线性数据，我们将增加第三个维度 z，它可以通过以下公式计算：

$$z = x^2 + y^2$$

通过添加第三维，样本空间将变成如下图所示：

那么现在，SVM 将按照以下方式对数据集进行分类。考虑下面的图片：

因为我们在三维空间中，所以它看起来像一个平行于 x 轴的平面。如果在二维空间中把它转换成 z=1，那么它将变成如图 5-所示：

因此，在非线性数据的情况下，我们得到半径为 1 的周长。

SVM 处理非线性数据集中使用了“核技巧”，这使得对点进行分类更容易。假设我们有一个这样的数据集：

这里我们看到我们不能画出一条线或者说是超平面来正确地对点进行分类。所以我们要做的就是尝试用一些二次函数把这个低维空间转换成高维空间这样我们就能找到一个决策边界可以清楚地划分数据点。这些帮助我们做到这一点的函数被称为核（Kernel），使用哪个内核完全由超参数调优决定。

这种方法的主要优点是它可能会降低泛化误差，使这个模型抵抗过拟合，这实际上已经在几个不同的分类任务中得到验证。

该方法不仅适用于二维空间，而且适用于高维空间和无限维空间。此外，我们可以使用非线性曲面，如多项式或径向基函数，通过所谓的核技巧，隐式地将输入映射到高维特征空间。

5.4.4 示例：Spark SVM 实现分类任务

支持向量机在高维或无限维空间中构造一个超平面或一组超平面，可用于分类、回归或其他任务。直观地说，一个很好的分离是通过超平面来实现的，它与任何类中最近的训练数据点的距离最大（所谓的函数边界），因为一般来说，边界越大，分类器的泛化误差就越小。

Spark ML 中的 org.apache.spark.ml.classification.LinearSVC 仅支持线性向量机的二元分类。在内部，它使用 OWLQN 优化器来优化 Hinge Loss。

【示例】创建机器学习模型，对泰坦尼克号沉船事件中的乘客进行生存预测。

在接下来的示例中，我们使用 Spark ML 的线性支持向量机来创建一个模型，预测哪些乘客在泰坦尼克号沉船中幸存下来。这是一个二元分类机器学习的问题。

泰坦尼克号的沉没是历史上最臭名昭著的沉船事件之一。1912 年 4 月 15 日，在泰坦尼克号的处女航中，被广泛认为是“永不沉没”的皇家邮轮泰坦尼克号在与冰山相撞后沉没。不幸的是，船上没有足够的救生艇，导致 2224 名乘客和船员中有 1502 人死亡。

虽然幸存者中有一些运气因素，但似乎有些群体比其他群体更有可能生存下来。在这个示例中，我们将使用乘客资料（即姓名、年龄、性别、社会经济阶层等）建立一个预测模型来回答这个问题：“什么样的人更有可能幸存下来?”。

在本示例中，将使用两个类似的数据集 train.csv 和 test.csv，其中包括乘客信息，如姓名（Name 字段）、年龄（Age 字段）、性别（Sex 字段）、船票等级（Pclass 字段）等。

我们的目标是，使用在 train.csv 数据中找到的模式，预测船上的其他 418 名乘客（在 test.csv 中找到）是否幸存。

1）读取数据集

首先，导入相关的依赖包，并加载数据集，代码如下：

2）探索性数据分析

执行以上代码，输出内容如下：

执行以上语句，输出结果如图 5-所示：

可以看出，891 名乘客中，65%是男性，35%是女性。

查看生存者与遇难者的乘客比例，代码如下：

执行以上语句，输出结果如图 5-所示：

可以看出，891 名乘客中，仅有 38%的幸存者（1 代表幸存，0 代表死亡）。进一步统计这些幸存者中的男女比例，代码如下：

执行以上语句，输出结果如图 5-所示：

可以看出，在幸存者中，女性占比达到了 68%。

3）分割训练集和测试集

将数据集随机分为训练集(80%)和测试集(20%)。训练集将用于构建我们的模型，测试集将用于评估模型。代码如下：

```

```

执行以上代码，输出内容如下：

```
训练数据量：577，测试数据量：137
```

4）创建机器学习管道

首先定义机器学习管道中的 Estimator 和 Transformer。其中使用 StringIndexer 将其编码为 double 类型，并存储到名为 label 的新列中；使用 VectorAssembler transformer 将这些特征合并到单个的向量列；创建一个朴素贝叶斯多类分类器。请记住，朴素贝叶斯算法假定特征之间是独立的，并且要求特征取非负值。代码如下：

```

```

5）训练模型

定义好管道后，使用其拟合训练数据集，代码如下：

```

```

执行以上代码，返回的是学习后的 CrossValidatorModel。CrossValidatorModel 包含跨折叠的平均交叉验证度量最高的模型，并使用该模型转换输入数据。CrossValidatorModel 还跟踪评估的每个参数映射的度量。

6）模型评估

现在可以从模型进行预测并查看结果。对测试数据进行预测，这样我们就可以测量模型对新数据的准确性。代码如下：

```

```

执行以上代码，绘制的 ROC 曲线如图 5-所示：

7）预测结果

最后，我们用上一步获得的最优模型，来预测 test.csv 数据集中每个乘客的生存结果。代码如下：

```

```

5.5 因子分解机分类器

SVM 分类器是机器学习和数据挖掘领域的翘楚，但是其在协同过滤这样的推荐领域就无能为力了，因为在这个领域的数据往往具有稀疏这个特点，SVM 在稀疏数据中往往不能得到一个优秀的分类超平面。

在现实世界中，许多应用问题（如文本分析、推荐系统等）会产生高度稀疏的（特征）数据，即特征向量中几乎所有的分量都为 0。在高度稀疏的数据场景中，由于数据量的不足，样本中普遍会出现未交互的特征分量，因而不能对相应的参数进行估计。在这种场景下，可以使用因子分解机来客服这种缺陷。

5.5.1 因子分解机算法简介

因子分解机（Factorization Machine, FM）是由斯特凡·伦德勒（Steffen Rendle）提出的一种基于矩阵分解的机器学习算法。这个模型结合了支持向量机（SVM）和因式分解模型的优点。与 SVM 类似的是，FM 是一个泛预测器，可以兼容任意实值的特征向量。而与 SVM 不同的是，FM 对于所有变量之间的联系使用了分解的参量去建模，解决了数据稀疏的情况下，特征怎样组合的问题。

普通的线性模型都是将各个特征独立考虑的，并没有考虑到特征之间的相互关系。但是实际上，特征之间可能具有一定的关联，比如男生更喜欢购买电子设备，女生更喜欢购买衣服化妆品。如果能找出这类的特征，是非常有意义的。FM 模型本质上是 SVM 模型的拓展，主要考虑到多维特征之间的交叉关系，其中参数的训练使用的是矩阵分解的方法。

FM 甚至能够在具有巨大稀疏性的问题（如广告和推荐系统）中估计交互。FM 公式为：

$$\hat{y} = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle v_i, v_j \rangle x_i x_j$$

前两项表示截距和线性项（与线性回归相同），最后一项表示成对相互作用项。v_i 用 k 个因子描述第 i 个变量。

FM 可用于回归任务，优化准则为均方误差。FM 也可以通过 sigmoid 函数进行二分类。优化准则为 logistic 损失。

两两相互作用可以重新表述为：

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle v_i, v_j \rangle x_i x_j = \frac{1}{2} \sum_{f=1}^{\mathrm{k}} \left(\left(\sum_{i=1}^{n} v_i, f^{x_i} \right)^2 - \sum_{i=1}^{n} {v_i}^2, f^{{x_i}^2} \right)$$

该方程在 k 和 n 上只有线性复杂度，即其计算复杂度为(O(kn))。

一般情况下，为了防止梯度爆炸问题，最好是将连续特征缩放到 0 到 1 之间，或者将连续特征打包并进行 one-hot 编码。

FM 模型的优点主要有三点：

(1) FM 模型可以在非常稀疏的数据中进行合理的参数轨迹（而 SVM 做不到这点）。

(2) FM 模型具有线性时间复杂度，可以直接在原始的模型公式上进行学习，优化效果很好，而且不需要像 SVM 一样依赖于支持向量。

(3) FM 是一个通用模型，它可以用于任何特征值为实值的情况。而其他因式分解模型只能用于一些输入数据比较固定的情况。

FM 模型可以用于回归任务、二分类任务、排名任务，特别是在数据稀疏场景下，效果明显，目前，FM 被广泛应用于推荐系统、广告系统等领域。目前来看，FM 模型是推荐系

统、广告系统的三大基础模型之一，如果想从事推荐、广告相关的工作、研究、学习，是必须要掌握的。

5.5.2 因子分解机特征工程

FM 模型关键是，假设特征之间两两相关。FM 模型对特征两两自动组合，不需要人工参与，类别型特征 One-Hot 化后才能参与计算。

举例说明如下。假设一个电影评分系统，根据用户和电影的特征，预测用户对电影的评分。系统记录了用户（UserID）在特定时间（Date）对电影（MovieID）的评分（Score），评分为 1，2，3，4，5。其中评分是 label，用户 id、电影 id、评分时间是特征。由于用户 id 和电影 id 是类别类型的，需要经过 one-hot 编码转换成数值型特征。因为是类别型特征，所以经过 one-hot 编码以后，导致样本数据变得很稀疏。该电影数据集如图 5-所示：

其中：

(1) 类别特征为：UserID、MovieID、Gender。

(2) 集合特征为：Genres、Friends、ItemsWatched。

(3) 连续特征为：Date、Age。

该电影数据集在特征化（数值化）之后，得到如图 5-所示的数据集，每行表示目标（即 Score 评分）与其对应的特征向量：

其中：

(1) 类别特征 UserID、MovieID、Gender 等执行了 one-hot 编码。

(2) 集合特征 Genres、Friends、ItemsWatched 等执行了单位化（.5=0.5，是简化表示）。

(3) 连续特征 Date 按天数进行了离散化分桶，Age 保持不变。

上面电影数据集的第一行样本，其参与计算时，FM 特征的二次项交互如图 5-所示：

FM 的某个类别特征 one-hot 编码后，只是 1 位起作用，0 位不起作用，便于计算和工程实现。但 one-hot 编码也带来了数据稀疏性和特征空间扩大。

5.5.2 示例：Spark FM 分类器实现二分类任务

Spark 机器学习库 spark.ml 的实现支持因子分解机用于二分类任务和回归任务。其中分类器实现为 org.apache.spark.ml.classification.FMClassifier，它支持正常梯度下降和 AdamW 求解。FM 分类模型使用 logistic 损失，可以通过梯度下降法求解，通常在损失函数中加入 L2 等正则化项以防止过拟合。

下面的示例以 LibSVM 格式加载 Spark 自带的数据集 sample_libsvm_data.txt，将其分成训练集和测试集，在第一个数据集上进行训练，然后在测试集上进行评估。我们将特征缩放到 0 到 1 之间，以防止梯度爆炸问题。

第 6 章 Spark ML 回归算法基础

回归和分类算法都被称为监督学习算法，用于机器学习和标记数据集的预测。这两个问题的目标都是构建一个简洁的模型，可以从属性变量中预测依赖属性的值。这两个任务之间的区别在于，依赖属性对于回归是数值属性，而对于分类是类别属性。回归问题是指输出变量是实数或连续值，如“工资”或“权重”。

回归算法的任务是找到映射函数，这样我们就可以将输入变量 x 映射到连续输出变量 y。回归分析是估计因变量和自变量之间关系的过程。简单地说，回归发现因变量和自变量之间的相关性。因此，回归算法有助于预测连续变量，如房价、市场趋势、天气模式、石油和天然气价格等。回归分析是用于预测的机器学习领域最基本的工具之一。

6.1 回归任务概述

使用回归，我们可以在可用数据上拟合一个函数，并尝试预测未来的结果或保留数据点。这种函数拟合有两个目的：

(1) 可以在数据范围内估计丢失的数据(插值)。

(2) 可以估计数据范围之外的未来数据(外推)。

应用回归分析的一些现实场景包括预测给定房屋特征的房屋价格、预测 SAT/GRE 分数对大学录取的影响、根据输入参数预测销售、预测天气等。

6.1.1 回归分析介绍

回归分析是一种用一个或多个自变量对因变量（目标变量）和自变量（预测变量）之间的关系进行建模的统计方法。更具体地说，回归分析帮助我们了解当其他自变量保持固定时，因变量的值如何对应于自变量变化。它预测连续值（实值），如温度，年龄，工资，价格等。

我们可以用下面的例子来理解回归分析的概念。

假设有一家营销公司 A，它每年做各种各样的广告并从中获得销售额。表 6-是该公司近 5 年的广告及销售情况：

表 6-1 消息格式

广告投入	销售额
￥90	￥1000
￥120	￥1300
￥150	￥1800
￥100	￥1200
￥130	￥1380
￥200	？？

现在，A 公司想在 2023 年做 200 元的广告，想知道今年的销售预测。因此，为了解决机器学习中的这类预测问题，我们需要回归分析。

回归是一种监督学习技术，它有助于发现变量之间的相关性，并使我们能够基于一个或多个预测变量预测连续输出变量。它主要用于预测、预报、时间序列建模，以及确定变量之间的因果关系。

在回归中，我们在最拟合给定数据点的变量之间绘制图表，使用该图表，机器学习模型可以对数据进行预测。简单来说，回归是指一条线或曲线，它通过目标预测图上的所有数据点，使数据点与回归线之间的垂直距离最小。数据点和线之间的距离表明模型是否捕获了强关系。

下面是一些回归的例子：

(1) 利用气温及其他因素预测雨量。

(2) 确定市场趋势。

(3) 预测因鲁莽驾驶而导致的交通事故。

与回归分析相关的术语解释如下。

(1) 因变量：在回归分析中，想要预测或理解的主要因素称为因变量。又称目标变量。

(2) 自变量：影响因变量或用来预测因变量值的因素称为自变量，也称为预测因子。

(3) 异常值：异常值是与其他观测值相比包含非常低或非常高值的观测值。异常值可能会影响结果，因此应该避免。

(4) 多重共线性：如果自变量之间的相关性高于其他变量，则称为多重共线性。它不应该出现在数据集中，因为它会在对影响最大的变量进行排序时产生问题。

(5) 欠拟合和过拟合:如果我们的算法可以很好地处理训练数据集，但不能很好地处理测试数据集，那么这样的问题被称为过拟合。如果我们的算法即使在训练数据集上也不能很好地执行，那么这种问题就被称为欠拟合。

如上所述，回归分析有助于预测连续变量。在现实世界中有各种各样的场景，我们需要一些未来的预测，如天气状况，销售预测，营销趋势等，对于这种情况，我们需要一些技术，可以做出更准确的预测。因此，对于这种情况，我们需要回归分析，这是一种统计方法，用于机器学习和数据科学。以下是使用回归分析的其他一些原因：

(1) 回归估计目标变量和自变量之间的关系。

(2) 它是用来发现数据的趋势。

(3) 它有助于预测实值/连续值。

(4) 通过执行回归，我们可以自信地确定最重要的因素，最不重要的因素，以及每个因素如何影响其他因素。

6.1.2 实现回归的不同方法

在数据科学和机器学习中使用了各种类型的回归。每种类型在不同的场景下都有自己的重要性，但在核心上，所有回归方法都是分析自变量对因变量的影响。在这里，我们将讨论一些重要的回归类型，

基于函数族和所使用的损失函数，可以将回归分为以下几类。

1. 线性回归

线性回归是一种用于预测分析的统计回归方法。它是一种非常简单和容易的算法，用于解决机器学习中的回归问题，并显示连续变量之间的关系。

线性回归表示自变量(x 轴)和因变量(y 轴)之间的线性关系，因此称为线性回归。如果只有一个输入变量(x)，则这种线性回归称为简单线性回归。如果有一个以上的输入变量，

那么这样的线性回归被称为多元线性回归。线性回归模型中变量之间的关系可以用图 6-来解释。在这里，我们是根据工作经验的年数来预测员工的工资。

在线性回归中，目标是通过最小化每个数据点的均方误差（mean-squared error，MSE）之和来拟合超平面（对于二维数据点是直线）。线性回归的数学方程如下：

$$Y = aX + b$$

在该方程中，Y=因变量（目标变量），X=自变量（预测变量），a 和 b 是线性系数。

线性回归的一些常用应用包括：

(1) 分析趋势和销售预测。

(2) 收入预测。

(3) 房地产价格预测。

(4) 到达目的地的交通堵塞预测。

2. 多项式回归

线性回归假设因变量(y)和自变量(x)之间的关系是线性的。当数据点之间的关系不是线性时，它无法拟合数据点。而多项式回归是一种使用线性模型对非线性数据集进行建模的回归。它类似于多元线性回归，但它扩展了线性回归的拟合能力，拟合 x 的值与 y 的相应条件值之间的非线性曲线。

由于因变量和自变量之间的数据点不存在线性关系，因此线性回归无法估计出良好的拟合函数。而多项式回归能够捕捉非线性关系。在多项式回归中，将原始特征转化为给定次数的多项式特征，然后用线性模型建模。这意味着数据点最好使用多项式线拟合。二次多项式回归如图 6-所示：

多项式回归方程也由线性回归方程导出，即线性回归方程 $Y = b_0 + b_1 x$，转化为多项式回归方程 $Y = b_0 + b_1 x + b_2 x^2 + b_3 x^3 + ... + b_n x^n$。这里 Y 是预测的值（目标输出），$b_0$、$b_1$、...、$b_n$ 是回归系数，x 是自变量（输入变量）。模型仍然是线性的，因为系数仍然是线性的。

多项式回归通过使用自变量的多项式函数来拟合非线性方程。考虑的函数越丰富，其拟合能力(通常)就越好。实际上，线性回归是二阶多项式回归的一种特例。

注意： 与多元线性回归的不同之处在于，在多项式回归中，单个元素具有不同的次数，而不是具有相同的次数的多个变量。

---

3. 岭回归

岭回归（ridge regression）解决了回归分析中的过拟合问题。岭回归是线性回归中最稳健的版本之一，它引入了少量的偏差，这样我们就可以得到更好的长期预测。添加到模型中的偏差量称为岭回归惩罚。我们可以通过使用 λ 乘以每个特征权值的平方来计算这个惩罚项。岭回归方程如下：

$$L(x,y)=Min(\sum_{i=1}^{n}(y_i-w_ix_i)^2+\lambda\sum_{i=1}^{n}(w_i)^2)$$

函数 f(x)可以是线性的，也可以是多项式的。如果自变量之间存在高度共线性，一般的线性或多项式回归将失败，因此可以使用 Ridge 回归来解决这类问题。

如果我们的参数比样本多，这有助于解决问题。例如，当对有 10 个训练点的数据拟合一个 25 次多项式时，可以看到它完美地拟合了红色数据点(下图中)。但这样做会损害中间的其他点(最后两个数据点之间的峰值)。这可以从图 6-中看到。岭回归试图解决这个问题。它试图通过牺牲训练点的拟合来最小化泛化误差。

岭回归是一种正则化技术,用于减少模型的复杂性。它也被称为 L2 正则化。在没有岭回归的情况下，当函数过拟合数据点时，学到的权重往往相当高。岭回归通过在损失函数中引入加权的缩放 L2 范数(beta)来限制正在学习的权重范数，从而避免了过度拟合。因此，训练模型在完美拟合数据点(学习权值的大范数)和限制权值的范数之间进行权衡。缩放常数 alpha>0 用于控制这种权衡。较小的 alpha 值将导致较高的范数权重和训练数据点的过拟合。另一方面，较大的 alpha 值将导致函数与训练数据点的拟合较差，但权重的范数非常小。仔细选择 alpha 的值将产生最佳的权衡。

4. LASSO 回归

LASSO 回归是另一种降低模型复杂性的正则化技术。LASSO 回归类似于岭回归，因为它们都被用作正则化器来防止训练数据点的过拟合。但是 LASSO 还有一个额外的好处。它强化了学习权重的稀疏性。

它类似于 Ridge 回归，只是惩罚项只包含绝对权重，而不是权重的平方。由于它取绝对值，因此，它可以将斜率缩小到 0，而岭回归只能将其缩小到接近 0。岭回归强制学习权值的范数较小，产生一组总范数减小的权值。大多数权重（如果不是全部）将是非零的。而 LASSO 回归试图找到一组权值通过使它们中的大多数都接近于零。这产生了一个稀疏权重矩阵，其实现比非稀疏权重矩阵更节能，同时在拟合数据点方面保持相似的精度。

LASSO 回归也被称为 L1 正则化。Lasso 回归方程如下：

$$L(x,y)=Min(\sum_{i=1}^{n}(y_i-w_ix_i)^2+\lambda\sum_{i=1}^{n}|w_i|)$$

下图使用 Ridge 和 Lasso 回归对数据点进行拟合，其相应的拟合和权重按升序绘制。可以看出，LASSO 回归中的大多数权重确实接近于零。

LASSO 和 Ridge 回归的区别在于 LASSO 使用权重的 L1 范数而不是 L2 范数。损失函数中的 L1 范数倾向于增加学习权重的稀疏性。常数 alpha>0 用于控制学习权值的拟合性和稀疏性之间的权衡。较大的 alpha 值会导致较差的拟合，但会使学习到的权重集更稀疏。另一方面，较小的 alpha 值会导致训练数据点的紧密拟合（可能导致过度拟合），但权重集的稀疏性较差。

5. ElasticNet 回归

ElasticNet 回归是 Ridge 和 LASSO 回归的结合。损失项包括权重的 L1 范数和 L2 范数及其相应的标度常数。它通常用于解决 LASSO 回归的局限性，例如非凸性。ElasticNet 增加了二次的权重惩罚，使其主要是凸的。

6. 贝叶斯回归

对于上面讨论的回归方法，目标是找到一组解释数据的确定性权重值。在贝叶斯回归中，我们不是为每个权重找到一个值，而是试图找到假设先验的这些权重的分布。

我们从权重的初始分布开始根据数据利用贝叶斯定理将先验分布和后验分布联系起来根据可能性和证据将分布推向正确的方向。当我们有无限个数据点时，权值的后验分布在普通最小二乘解即方差趋于零的解处变成一个脉冲。

寻找权重的分布而不是单一的确定性值集有两个目的：

(1) 它自然地防止过拟合的问题，因此充当正则化器。

(2) 它提供了置信度和权重范围，这比只返回一个值更有逻辑意义。

7. 决策树回归

决策树是一种监督学习算法，可用于解决分类和回归问题。它既可以解决分类数据问题，也可以解决数值数据问题。决策树回归构建了一个树状结构，其中每个内部节点表示一个属性的“测试”，每个分支表示测试的结果，每个叶节点表示最终的决策或结果。

这种回归的主要目的是将数据集划分为更小的子集。从根节点/父节点（数据集）开始构建决策树，该决策树分为左右子节点（数据集子集）。这些子节点进一步划分为它们的子节点，并且它们自己成为这些节点的父节点。图 6-显示了决策树回归的例子：

在这里，模型试图预测一个人在跑车或豪华车之间的选择。

8. 随机森林回归

随机森林是最强大的监督学习算法之一，它既能执行回归任务，也能执行分类任务。

随机森林回归是一种集成学习方法，它将多棵决策树组合在一起，根据每棵决策树输出的平均值来预测最终的输出。组合决策树称为基本模型，可以更正式地表示如下：

$$g(x) = f_0(x) + f_1(x) + f_2(x) + ...$$

随机森林使用装袋法或自助聚合技术的集成学习，其中聚合并行运行且不相互交互的决策树。在随机森林回归的帮助下，我们可以通过创建数据集的随机子集来防止模型中的过拟合。随机森林预测如图 6-所示：

随机森林回归在机器学习中被大量使用。它使用多个决策树来预测输出。从给定的数据集中随机选择数据点，并通过该算法构建决策树。

9. 支持向量回归

支持向量机是一种监督学习算法，可以用于回归和分类问题。所以如果我们用它来解决回归问题，那么它就被称为支持向量回归（SVR）。这种回归类型解决线性和非线性模型。它使用非线性核函数，如多项式，来寻找非线性模型的最优解。

支持向量回归是一种适用于连续变量的回归算法。以下是支持向量回归中使用的一些关键字：

(1) 核（Kernel）：一个将低维数据映射到高维数据的函数。

(2) 超平面（Hyperplane）：在一般支持向量机中，它是两个类别之间的分隔线，而在支持向量回归中，它是一条有助于预测连续变量并覆盖大部分数据点的线。

(3) 边界线（Boundary line）：边界线是超平面之外的两条线，它为数据点创建了一个边距。

(4) 支持向量（Support vectors）：支持向量是最接近超平面和相对类的数据点。

在支持向量回归（SVR）中，我们总是尝试确定一个具有最大边距的超平面，以便在该边距中覆盖最大数量的数据点。SVR 的主要目标是考虑边界线内的最大数据点，超平面（最佳拟合线）必须包含最大数量的数据点。考虑下面的图片：

在这里，中间蓝色的线被称为超平面，另外两条线被称为边界线。

6.1.3 回归与分类的区别

图 6-展示了分类与回归算法。

表 6-显示了回归算法与分类算法之间的具体差异。

表 6-1 回归算法与分类算法之间的具体差异

回归算法	分类算法
输出变量必须是连续性质或实值	输出变量必须是离散值
回归算法的任务是将输入值(x)映射到连续输出变量(y)	分类算法的任务是将输入值 x 与离散输出变量 y 进行映射
它们与连续数据一起使用	它们与离散数据一起使用
它试图找到最佳拟合线，从而更准确地预测输出	分类试图找到决策边界，将数据集划分为不同的类
回归算法解决回归问题，如房价预测和天气预测	分类算法解决分类问题，如识别垃圾邮件、发现癌细胞，以及语音识别
可以进一步将回归算法分为线性回归和非线性回归	可以进一步将分类算法分为二元分类器和多类分类器

6.2 线性回归算法原理

从历史上看，线性回归一直是最广泛使用的回归方法之一，也是统计学中最基本的分析方法之一，至今仍在广泛使用。这是因为对线性关系建模要比对非线性模型建模要容易得多。

对结果模型的解释也更容易。线性回归的理论也为机器学习中更先进的方法和算法奠定了基础。

6.2.1 线性回归算法简介

线性回归是一种有监督的机器学习算法，用于预测两个变量之间关系的统计分析。它假设自变量和因变量之间存在线性关系，旨在找到描述这种关系的最佳拟合线。这条线是通过最小化预测值和实际值之间的平方差的总和来确定的。

1. 线性回归

与其他类型的回归一样，线性回归可以使用一组自变量来预测目标变量（因变量）并量化它们之间的关系。线性回归假设在自变量和因变量之间有一个线性关系（因此得名）。当具有多个自变量和一个因变量时，称为多元线性回归。当只有一个自变量和一个因变量时，这叫做简单线性回归。

线性回归主要用于预测数值型的输出变量。回归是用来预测一条符合输入数据的直线，以最好的方式指出数据点，并可以帮助预测未知数据。但这里需要注意的是，输入变量和输出变量之间需要某种关系。变量关系主要有两种类型:

(1) 线性。

(2) 非线性。

任何两个变量之间的线性关系的概念表明，两者在某些方面是成比例的。任何两个变量之间的相关性都表明了它们之间的线性关系。相关系数可以从-1 到+ 1。负相关是指通过增加一个变量，另一个变量减小。例如，动力和车辆行驶里程可能是负相关的，因为当我们增加动力时，车辆行驶里程就会下降。而薪资和工作年限可能是正相关的，因为随着工作年限的增长，薪资也随着增加。

非线性关系本质上比较复杂，因此需要额外的细节来预测目标变量。例如，在一辆自动驾驶汽车中，地形、信号系统和行人等输入变量与车速之间的关系是非线性的。

2. 简单线性回归

线性回归算法的目标是在自变量的基础上找到能够预测因变量值的最佳线性方程。该方程提供了一条表示因变量和自变量之间关系的直线。在简单的线性回归中，有一个自变量和一个因变量。该模型估计最佳拟合线的斜率和截距，表示变量之间的关系。斜率表示自变量每单位变化时因变量的变化，截距表示自变量为零时因变量的预测值。使用简单线性回归，可以在两个维度中绘制问题：x 轴是自变量，y 轴是因变量。如图 6-所示。

在上图中，X（输入）是工作经验，Y（输出）是一个人的工资，它显示了输出变量 Y 和预测变量 X 之间的线性关系。线性回归的任务是根据给定的自变量 x 来预测因变量值 y。根据给定的数据点，我们尝试绘制一条最适合这些点的线。图中回归线就是我们模型的最佳拟合直线。

为了计算最佳拟合直线线性回归，使用传统的斜截式，如下所示：

$$Y_i = B_0 + B_1 X_i$$

其中 Y_i=因变量， B_0=截距， B_1=斜率， X_i=自变量。

该算法使用直线 $Y = B_0 + B_1X$ 来解释因变量 Y（输出）和自变量 X（预测变量）之间的线性关系。如图 6-所示：

但是线性回归如何找出哪条直线是最佳拟合线呢?

线性回归算法的目标是得到 B_0 和 B_1 的最佳值，从而找到最佳拟合线。最佳拟合线是一条误差最小的线，这意味着预测值与实际值之间的误差应该最小。

在回归中，因变量的观测值 Y_i 与预测值 $Y_{predicted}$ 之间的差称为残差（residuals）。计算公式如下：

$$E_i = Y_{predicted} - Y_i$$

其中 $Y_{predicted} = B_0 + B_1X_i$ 。

简单来说，最佳拟合线是一条以最佳方式拟合给定散点图的线。数学上，通过最小化残差平方和（RSS）来获得最佳拟合线。

3. 线性回归的成本函数

成本函数（cost function，也叫做损失函数-loss function）有助于计算出 B_0 和 B_1 的最优值，从而为数据点提供最佳拟合线。

在线性回归中，通常使用均方误差（Mean Squared Error，MSE）成本函数，它是 $Y_{predicted}$ 和 Y_i 之间发生的平方误差的平均值。我们使用简单的线性方程 y=mx+b 来计算 MSE：

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(y_i - (B_1x_i + B_0))^2$$

使用 MSE 函数，我们将更新 B_0 和 B_1 的值，使 MSE 值稳定在最小值。这些参数可以用梯度下降法确定，使成本函数的值最小。

4. 线性回归的梯度下降

梯度下降算法是一种优化成本函数（目标函数）以达到最优最小解的优化算法。为了找到最优解，我们需要降低所有数据点的成本函数（MSE）。这是通过迭代更新 B_0 和 B_1 的值来完成的，直到我们得到一个最优解。

回归模型优化梯度下降算法，通过随机选取系数值降低成本函数，然后迭代更新值达到最小成本函数，更新直线系数。

让我们举个例子来理解这一点。想象一个 U 形的坑。你站在坑的最高点，你的动机是到达坑的底部。假设在坑的底部有一个宝藏，你只能走离散的台阶才能到达底部。如果你选择一次走一步，你最终会到达坑的底部，但这将花费更长的时间。如果你决定每次都采取更大的步伐，你可能会更快地到达底部，但是，也可能会越过底部，甚至不会接近底部。在梯度下降算法中，所采取的步数可以被认为是学习率（learning rate），这决定了算法收敛（converges）到最小值的速度。

为了更新 B_0 和 B_1，我们从成本函数中取梯度。为了求出这些梯度，我们对 B_0 和 B_1 求偏导数。

我们需要最小化成本函数 J。实现这一目标的方法之一是应用批量梯度下降算法。在批量梯度下降中，值在每次迭代中更新（后两个方程显示值的更新）。

偏导数是梯度，它们用来更新 B_0 和 B_1 的值。α（Alpha）是学习率。

5. 线性回归的假设

线性回归是理解和预测变量行为的强大工具，然而，它需要满足一些条件才能得到准确可靠的解决方案：

(1) 线性：自变量和因变量之间呈线性关系。这意味着因变量的变化以线性方式遵循自变量的变化。

(2) 独立性：数据集中的观测值彼此独立。这意味着一个观测值的因变量值不依赖于另一个观测值的因变量值。

(3) 均方差性：在自变量的所有水平上，误差的方差是恒定的。这表明自变量的数量对误差的方差没有影响。

(4) 正态性：模型误差呈正态分布。

(5) 无多重共线性：自变量之间没有高度的相关性。这表明自变量之间很少或没有相关性。

6.2.2 线性回归算法理解

让我们以年龄和工资为例来深度理解（简单）线性回归。

在这个简单线性回归的例子中，可以根据每个人的年龄来预测收入。这两个变量（年龄、收入）之间显然存在相关性：基本上随着年龄的增长，收入也相应地增加。让我们假设我们有 4 个人的记录，记录了他们的年龄和各自的薪资收入，如表 6-所示。

表 6-1 消息格式

No（编号）	Age（年龄）	Salary（年收入，万元）
1	20	5
2	30	10

3	40	15
4	50	22

使用散点图显示如下：

通过线性回归能够找到一条穿过这些数据点的中间线，这样，给定年龄，就可以估算出所能期望的最可能的收入。

1. 线性回归方程计算

一般来说，如果想在二维空间中画一条线，需要两个值：直线的斜率和直线与 y 轴相交的值，也称为截距。如果把年龄变量表示为 x，计算收入的函数表示为 h（代表 hypothesis），以及截距表示为 B_0，斜率(slope)表示为 B_1，相应地，这条线可以用如下的公式来描述：

$$h(x) = B_0 + B_1 x$$

线性回归的目标是找到最适合该数据的系数 B_0 和 B_1。寻找合适的系数的线性回归方法是最小化所谓的成本函数（cost function，也有人称之为损失函数）。该成本函数（cost function）返回一个单独的值，该值可以用来衡量由系数确定的直线是否与数据集中的所有示例相匹配。可以使用不同的成本函数。线性回归中使用的一种方法是计算在数据集中的所有 m 个例子中，目标变量的预测值和实际值之间的平方差的平均值（均方误差)）

在线性回归中使用“均值-平方误差”成本函数，因为单个偏差的平方不能相互抵消（总是正数），即使对应的偏差可能是负的；函数是凸的，意思是没有局部最小值，只有全局最小值；以及用来找到它的最小值存在的一个分析解。

现在，如果我们要基于这些早期员工的收入来预测第 5 个人（新人）的收入，最好的预测方法是取现有薪资值的平均值。根据这些信息，这将是最好的预测（这就像建造一个机器学习模型，但没有任何输入数据，因为使用输出数据作为输入数据）。

让我们来计算这些给定工资值的平均工资。计算公式如下：

$$\text{平均薪资} = \frac{(5+10+15+22)}{4} = 13$$

所以，对第 5 个人的薪资收入最好的预测是 13（万元）。在下面图 6-中显示了每个人的薪资收入以及平均值（在只使用一个变量的情况下，最合适的线）。

如果仔细看一下，前四个人的工资都不在这条最合适的线上；似乎与平均工资值有一定的差距，如下面的图 6-（残差值分布图）所示。如果我们计算这个误差距离的和，它们加起来，结果是 0。所以，我们不能简单地把它们相加，而是把每个误差求平方后再相加。

计算结果为：误差平方和= 64 + 9 + 4 + 81 = 158。

将残差平方求和，得到 158，这就是误差平方和（sum of squared errors，SSE）。

接下来，暂时把这个 SSE 值放在一边。考虑输入变量 Age（这个人的年龄）的影响并预测这个人的薪资收入。将 Age 变量作为 x 轴，Salary 变量作为 y 轴，可视化年龄和工资之间的关系，如图 6-中所示：

从图 6-中可以观察到，工作经验和薪资收入之间呈现正相关的关系，它表明，由于输入（年龄)）和输出（薪资）之间的强线性关系，该模型能够预测目标值（薪资）。如前所述，线性回归的总体目标是找到一条与数据点匹配的直线，使实际目标值与预测值的平方差最小。因为它是一条直线，我们知道在线性代数中直线的方程是 y= mx + c，如图 6-所示：

因为线性回归也是求这条直线的，所以线性回归方程如下（因为只用了一个输入变量，如年龄）：

$$y = B_0 + B_1 * x$$

其中：

(1) y=薪资收入（预测的）

(2) B_0 =截距（当年龄为 0 时的薪资收入值）

(3) B_1 =斜率或薪资的系数

(4) x=年龄

现在，大家可能会问，是否可以绘制多条通过数据点的线（如图 6-所示），以及如何计算出哪个是最佳拟合的线。

寻找最优拟合直线的第一个准则是通过数据点的质心，如图 6-所示。

在本例中，质心值为：

$$\text{平均年龄} = \frac{(20+30+40+50)}{4} = 35$$

$$\text{平均薪资} = \frac{(5+10+15+22)}{4} = 13$$

第二个标准是它应该能够最小化误差的平方和。我们知道回归线方程等于：

$$y = B_0 + B_1 * x$$

现在使用线性回归的目的是找出截距 B_0 和系数 B_1 的最优值，使残差/误差最大限度地减小。可以使用以下公式很容易地为我们的数据集找到 B0 和 B1 的值。

$$B_1 = \frac{\sum (x_i - x_{均值}) * (y_i - y_{均值})}{\sum (x_i - x_{均值})^2}$$

$$B_0 = y_{均值} - B_1 * x_{均值}$$

下表展示了使用输入数据进行线性回归时斜率和截距的计算。

表 6-1 线性回归时斜率和截距的计算

Age	Salary	Age 方差(与均值)	Salary 方差(与均值)	协方差(乘积)	Age 方差(平方)
20	5	-15	-8	120	225
30	10	-5	-3	15	25
40	15	5	2	10	25
50	22	15	9	135	225

任何两个变量（Age 和 Salary）之间的协方差被定义为每个变量与其均值之间距离的乘积。简而言之，Age 和 Salary 的方差的乘积称为协方差。现在我们有协方差积和 Age 方差的平方值，我们可以继续计算线性回归线的斜率和截距的值，计算过程如下：

$$B_1 = \frac{\sum(\text{协方差})}{\sum(Age\text{方差的平方})} = \frac{280}{500} = 0.56$$

$$B_0 = 13 - (0.56 * 35) = -6.6$$

最后的线性回归方程变成：

$$y = B_0 + B_1 * x$$

$$Salary = -6.6 + (0.56 * Age)$$

现在可以用这个方程预测任意年龄的薪资收入值。例如，模型会像下面这样预测第一个人的薪资收入：

$$Salary(\text{张三}) = -6.6 + (0.56 * 20) = 4.6(\text{万元})$$

2. 线性回归方程解释

前面我们计算出最后的线性回归方程为：

$$y = B_0 + B_1 * x$$

$$Salary = -6.6 + (0.56 * Age)$$

斜率（$B_1 = 0.56$）在这里的意思是一个人每增加 1 岁，工资也相应增加了 5600 元。

截距从其值的含义来看，并不总是有意义。就像在这个例子中，- 6.6 的值表明，如果这个人还没有出生（Age=0），这个人的工资将是- 66,000 元。

图 6-显示了我们的数据集的最终回归线。

下面让我们用该回归方程来预测数据集中所有四条记录的薪资收入，并与实际工资进行比较，如表 6-所示。

表 6-1 预测所有四条记录的薪资收入

Age	Salary	预测的 Salary	误差值
20	5	4.6	-0.4
30	10	10.2	0.2
40	15	15.8	0.8
50	22	21.4	-0.6

简而言之，线性回归给出了截距（B_0）和系数（$B_1, B_2, ...$）的最优值，使预测值与目标变量的差值（方差）最小。

线性回归被用于许多不同的领域，包括金融、经济学和心理学，以理解和预测特定变量的行为。例如，在金融领域，线性回归可以用来理解公司的股票价格和收益之间的关系，或者根据过去的表现来预测货币的未来价值。

6.2.3 多元线性回归

虽然我们看到一个只有一个输入变量来理解线性回归的简单例子，但现实中其实很少出现这种情况。在大多数情况下，数据集包含多个变量，因此在这种情况下构建多变量回归模型更有意义。

1. 多元线性回归方程

多元线性回归方程如下所示：

$$y = B_0 + B_1 * x_1 + B_2 * x_2 + B_3 * x_3 + ...$$

对于多元线性回归方程，如何找到最小成本函数？通常使用梯度下降法找到最小值。

梯度下降法（gradient-descent method）是迭代地进行工作的。它从某一点（代表对系数参数值的最佳猜测（这一点也可以随机选择））开始，对于每一个系数参数 w_j，计算成本函数对该系数参数的偏导数。偏导数告诉算法如何改变系数参数以尽可能快地下降到成本函数的最小值。该算法然后根据计算的偏导数对系数参数进行更新，并计算了新点的成本函数值。如果新值小于某个公差值（tolerance value），我们说算法已经收敛（converged），该过程停止。梯度下降法请参见图 6-。

多元线性回归是一种理解单个因变量和多个自变量之间关系的技术。多元线性回归的公式也类似于简单线性回归，但有一个小的变化是，不再只有一个变量，现在包含所有的变量。公式如下：

$$Y = B_0 + B_1 X_1 + B_2 X_2 + ... + B_p X_p + \varepsilon$$

2. 多元线性回归的若干考虑

对于简单线性回归的所有四个假设以及一些新的附加假设仍然适用于多元线性回归。

(1) 过拟合。当越来越多的变量被添加到模型中时，模型可能会变得过于复杂，并且通常最终会记住训练集中的所有数据点。这种现象被称为模型的过拟合（overfitting）。这通常会导致高训练精度和非常低的测试精度。

(2) 多重共线性。这是一种现象，在一个具有几个自变量的模型中，可能有一些相互关联的变量。

(3) 特征选择。有了更多的变量，从给定的特征池（其中许多可能是冗余的）中选择最优的预测器集成为构建相关的和更好的模型的重要任务。

多重共线性

由于多重共线性使得很难找出哪个变量实际上对响应变量的预测有贡献，它导致人们得出错误的结论，即变量对目标变量的影响。虽然它不影响预测的精度，但正确检测和处理模型中存在的多重共线性是至关重要的，因为从模型中随机删除任何这些相关变量都会导致系数值剧烈摆动甚至改变符号。

多重共线性可以用以下方法检测：

(1) 成对相关性：检查不同自变量对之间的成对相关性可以在检测多重共线性方面提供有用的见解。

(2) 方差膨胀因子(VIF)：两两相关性可能并不总是有用的，因为可能只有一个变量可能无法完全解释其他变量，但一些变量组合起来可能会这样做。因此，要检查变量之间的这些关系，可以使用 VIF（Variance Inflation Factor）。VIF 基本上解释了一个自变量与所有其他自变量的关系。VIF 由以下公式给出，

$$VIF = \frac{1}{1-R^2}$$

其中 i 指的是第 i 个变量，它被表示为自变量的其余部分的线性组合。

对于 VIF 值通常遵循的启发式原则是，如果 VIF > 10，则该值肯定很高，应该将其删除。如果 VIF=5，那么它可能是有效的，但应该先检查。如果 VIF < 5，则认为它是一个良好的 VIF 值。

6.3 Spark ML 回归算法实现

Apache Spark 附带了一个名为 ML 的库，用于使用 Spark 框架执行机器学习任务。Spark ML 为回归任务提供了一些分布式机器学习算法，包括：

(1) Linear regression，线性回归。

(2) Generalized linear regression，广义线性回归。

(3) Decision tree regression，决策树回归。

(4) Random forest regression，随机森林回归。

(5) Gradient-boosted tree regression，梯度增强树回归。

(6) Survival regression，生存回归。

(7) Isotonic regression，等渗回归。

(8) Factoriztion machines regression，等渗回归。

在 Spark ML 中，我们可以使用 Spark 构建各种机器学习回归模型，如线性回归、广义回归、决策树回归、随机森林回归、梯度增强回归等。

注意： 请注意 Spark MLlib 和 Sppark ML 的区别。spark.mllib 是基于 RDD 的传统机器学习 API，而 spark.ml 是新的基于 DataFrame 的机器学习 API。MLlib 这个名称既包括基于 RDD 的 API，也包括基于 DataFrame 的 API。基于 RDD 的 API 现在处于维护模式，因此将不会再向基于 RDD 的 API 添加新特性。

6.4 示例：线性回归实现二手房价格预测

本节中，我们将实现一个 Spark ML 线性回归模型来预测二手房价格。

6.4.1 Spark 回归算法类 LinearRegression

Spark MLlib 中实现了线性回归算法类 org.apache.spark.ml.regression.LinearRegression。学习目标是最小化指定的损失函数，并进行正则化。它支持两种类型的损失：

(1) squaredError（即平方损失）。

(2) huber（相对较小误差的平方误差和相对较大误差的绝对误差的混合，我们从训练数据中估计规模参数）。

它支持多种类型的正则化：

(1) none（也就是普通最小二乘）。

(2) L2（岭回归，ridge regression）。

(3) L1（Lasso 回归）。

(4) L2 + L1（弹性网络）。

注意，huber 损失拟合只支持 none 正则化和 L2 正则化。

从 Spark 3.1.0 开始，它支持将实例堆叠到块中，并使用 GEMV 来获得更好的性能。如果参数 maxBlockSizeInMB 设置为 0.0（默认），则块大小将为 1.0 MB。

LinearRegression 算法的模型超参数子集如下所示：

(1) regParam：这是用来控制过拟合的正则化参数。默认值是 0.0。

(2) fitIntercept：这个参数用于确定是否拟合截距。默认值为 true。

6.4.2 示例：二手房价格预测

下面的例子将试图根据一组关于房子的信息来预测房价。使用的数据集来自于 Kaggle 上的住房信息数据。数据以 CSV 格式提供，有两个文件，即 train.csv 和 test.csv。其中 train.csv 文件中的标签列被称为 SalePrice，表示以美元出售的房产价格，这也是我们要预测的目标列。而在 test.csv 中，则不包含 SalePrice 列。

在本示例中，我们只使用 train.csv 的一个子集，如表 6-所示。

表 6-1 示例中使用到的房屋信息字段

字段	含义	备注
SalePrice	以美元出售的房产价格	目标列
LotArea	地块面积(以平方英尺计)	
RoofStyle	屋顶类型	
Heating	暖气方式	
1stFlrSF	一楼面积	
2ndFlrSF	二楼面积	
BedroomAbvGr	地下室层以上的卧室数	
KitchenAbvGr	厨房数量	
GarageCars	车库容量大小	
TotRmsAbvGrd	总房间（不包括浴室）	
YearBuilt	原施工日期	

我们的目标是训练一个线性回归模型，来预测房价。

【示例】应用 Spark LinearRegressionl 回归算法构建预测模型，用来根据一组关于房子的信息来预测房价。

6.4 回归模型评估

任何线性回归模型都可以用各种评价指标来评估。这些评估指标通常提供了一种衡量由模型生成的观察到的输出效果的方法。最常用的指标是：

(1) 决定系数或 R 平方（R-Squared，R2）。

(2) 均方根误差（Root Mean Squared Error，RSME）和残差标准误差（Residual Standard Error，RSE）。

6.4.1 决定系数或 R 平方（R2）

R-Squared 是一个数字，用于解释已开发模型所解释/捕获的变化量。它的范围总是在 0 和 1 之间。总体而言，R 平方值越高，模型与数据的拟合越好。

数学上可以表示为：

$$R^2 = 1 - \frac{RSS}{TSS}$$

残差平方和（RSS，Residual sum of square）被定义为样本集（数据集）中每个数据点的残差平方和。它是对预期输出和实际观察输出之间差异的度量。

$$SSR = \sum_{i=1}^{n}(y_i - b_0 - b_1 x_i)^2$$

总平方和（TSS）定义为数据点与响应变量均值的误差之和。数学上 TSS 是，

$$TSS = \sum(y_i - \bar{y}_i)^2$$

其中 $\bar{y}$ 是样本数据点的均值。

请回顾 6.2.2 节中的以年龄和工资为例来深度理解（简单）线性回归示例。记得当我们只使用输出变量本身时，我们计算过误差的平方和（sum of squared errors，SSE），它的值是 158。现在已经使用一个输入变量构建了这个模型，让我们重新计算这个模型的 SSE。表 6-为使用线性回归以后新 SSE 的计算结果。

表 6-1 使用线性回归后新的 SSE 计算结果

Age	Salary	预测的 Salary	误差值	平方差	原 SSE
20	5	4.6	-0.4	0.16	64
30	10	10.2	0.2	0.04	9
40	15	15.8	0.8	0.64	4
50	22	21.4	-0.6	0.36	81

正如我们所看到的，由于使用了线性回归，平方差的总和从 158 显著地减少到只有 1.2。目标变量（薪资收入）的方差可以通过回归（由于使用输入变量-年龄）来解释。因此，线性回归致力于减少总体误差平方和。总误差平方和 TSS 是两种类型的组合：TSS = SSE + SSR。

总误差平方和（Total Sum of Squared Errors，TSS）是实际值与平均值之差的平方和，并且总是固定的。在我们的例子中这个值等于 158。

SSE（Sum of squared errors）是目标变量实际值与预测值的平方差（即残差平方和 RSS），从上表可知，经线性回归后降为 1.2。

SSR（Sum of square regressional）是回归解释的平方和，可用(TSS - SSE)计算。计算过程如下：

$$SSR = 158 - 1.2 = 156.8$$

$$r^2(\text{决定系数}) = \frac{SSR}{TSS} = \frac{156.8}{158} = 0.99$$

这一比例表明，我们的线性回归模型在预测给定年龄的工资数额方面具有 99%的准确性。另外 1%可以归因于模型无法解释的错误。R 平方的含义如下图所示：

6.4.2 均方根误差（RMSE）

均方根误差是残差方差的平方根。它指定了模型与数据的绝对拟合，即观测数据点与预测值的接近程度。数学上可以表示为，

$$RMSE = \sqrt{\frac{RSS}{n}} = \sqrt{\sum_{i=1}^{n}\left(y_i^{Actual} - y_i^{\Pr edicted}\right)^2 \Big/ n}$$

为了使这个估计无偏，必须将残差的平方之和除以自由度，而不是模型中数据点的总数。这一项被称为残差标准差(RSE，Residual Standard Error)。数学上可以表示为，

$$RSE = \sqrt{\frac{RSS}{df}} = \sqrt{\sum_{i=1}^{n}\left(y_i^{Actual} - y_i^{\Pr edicted}\right)^2 \Big/ (n-2)}$$

R 平方是比 RSME 更好的度量。因为均方根误差的值取决于变量的单位（即它不是一个标准化的度量），它可以随着变量单位的变化而变化。

6.4.3 过拟合与欠拟合

总是存在这样的情况：模型在训练数据上表现良好，但在测试数据上却表现不佳。当在数据集上训练模型时，过拟合和欠拟合是人们面临的最常见问题。

在理解过拟合和欠拟合之前，必须先了解偏差（bias）和方差（variance）。

1. 偏差和方差

偏差（bias）是一种衡量方法，用来确定模型对未来未知数据的准确性。复杂的模型，假设有足够的训练数据可用，可以做准确的预测。然而，过于幼稚的模型，很可能在预测方面表现不佳。简单地说，偏见是由训练数据造成的错误。

一般来说，线性算法有很高的偏差，这使得它们易于学习和理解，但通常不太灵活。这意味着在复杂问题上较低的预测性能，因而无法达到预期结果。

方差（variance）是模型对训练数据的敏感性，也就是说，它量化了当输入数据发生变化时模型的反应程度。

理想情况下，从一个训练数据集到下一个训练数据集，模型不应该改变太多，这将意味着算法善于在输入和输出变量之间挑选出隐藏的潜在模式。

理想情况下，模型应该具有较低的方差，这意味着模型在改变训练数据后不会发生剧烈变化(它是可泛化的)。即使在训练数据集中发生很小的变化，较高的方差也会使模型发生巨大的变化。

让我们来理解什么是偏差-方差权衡。

2. 偏差-方差权衡

任何监督机器学习算法的目标都是实现低偏差和低方差，因为它更稳固。从而使算法达到更好的性能。

在机器学习中，偏差和方差之间的关系是不可避免的。偏差和方差之间呈反比关系：

(1) 偏差的增加会使方差减小。

(2) 方差的增加将减小偏差。

如-6 所示：

这两个概念之间存在权衡，算法必须在偏差和方差之间找到平衡。

事实上，我们无法计算出真正的偏差和方差误差项，因为我们不知道实际的潜在目标函数。

现在来看看过拟合和欠拟合。

3. 过拟合

当一个模型学习数据中的每一个模式和噪声，以至于它影响模型在未知的未来数据集上的性能时，它被称为过拟合（overfitting）。该模型与数据吻合得非常好，以至于它将噪声解释为数据中的模式。

当一个模型具有低偏差和高方差时，它最终会记住数据并导致过拟合。过拟合导致模型变得特定而非通用。这通常会导致高训练精度和非常低的测试精度。

检测过拟合是有用的，但它并不能解决实际问题。有几种方法可以防止过拟合，具体如下：

(1) 交叉验证。

(2) 如果训练数据太小而无法训练，则添加更多相关且干净的数据。

(3) 如果训练数据太大，做一些特征选择，去掉不需要的特征。

(4) 正则化。

4. 欠拟合

欠拟合不像过拟合那样经常被讨论。当模型不能从训练数据集中学习，也不能泛化测试数据集时，称为欠拟合（underfitting）。这种类型的问题可以很容易地通过性能指标检测出来。

当一个模型具有高偏差和低方差时，它最终不会泛化数据并导致欠拟合。它无法从数据中找到隐藏的底层模式。这通常会导致较低的训练精度和非常低的测试精度。防止欠拟合的方法如下：

(1) 增加模型的复杂性。

(2) 增加训练数据中的特征数量。

(3) 从数据中去除噪声。

在上一节的示例中，我们的线性回归曲线拟合这个模型非常好，但也可能是过度拟合的情况。当模型对训练数据的预测精度很高，但对不可见/测试数据的性能下降时，就会发生过拟合。解决过拟合问题的技术称为正则化，有不同类型的正则化技术。在线性回归方面，可以使用 Ridge、Lasso 或 Elasticnet 正则化技术来处理过拟合。

岭回归（Ridge Regression）又称 L2 正则化，主要是将输入特征的系数值限制在接近于零的范围内，而 Lasso 回归(L1)将部分系数设置为零以提高模型的泛化性。Elasticnet 是这两种技术的结合。

归根到底，回归仍然是一种参数驱动的方法，并且假设关于输入数据点分布的基本模式很少。如果输入数据不符合这些假设，线性回归模型就不能很好地执行。因此，为了在使用线性回归模型之前了解这些假设，快速地复习这些假设是很重要的。

6.4.4 线性回归中的假设

回归是一种参数方法，这意味着它对数据进行假设以进行分析。对于成功的回归分析，验证以下假设是必不可少的。

(1) 残差的线性：因变量和自变量之间需要有线性关系。

(2) 残差的独立性：误差项不应该相互依赖（就像在时间序列数据中，下一个值依赖于前一个值）。残差项之间不应有相关性。这种现象的缺失被称为自相关。在误差项中不应该有任何可见的模式。

(3) 残差的正态分布：残差均值应服从正态分布，均值等于零或接近于零。这样做是为了检查所选的线是否实际上是最佳拟合线。如果误差项是非正态分布的，则表明有一些不寻常的数据点必须仔细研究才能建立更好的模型。

(4) 残差的相等方差：误差项必须有恒定的方差（constant variance）。这种现象被称为同方差（Homoscedasticity）。误差项中非恒定方差的存在称为异方差（Heteroscedasticity）。通常，非恒定方差出现在异常值（outliers）或极端杠杆值（extreme leverage values）的存在下。

6.4.5 示例：二手房价格预测模型评估

在 6.4 节，我们已经构建了一个二手房价格预测的线性回归模型，完整代码如下：

执行以上代码，输出内容如下：

在机器学习中，回归评估器是用于评估回归模型的性能和准确度的工具。它通过比较模型预测的值与实际观测值之间的差异，来判断模型的拟合程度。在 Spark 机器学习库中，提供有一个 org.apache.spark.ml.evaluation.RegressionEvaluator 评估器，用于对回归模型进行评估。它提供了一些常用的回归评估指标，并且可以方便地计算这些指标。请注意，模型评估指标是指测试集的评估指标，而不是训练集的评估指标。

评估指标支持以下几种：

(1) rmse：默认评估指标，root mean squared error（均方根误差）。均方根误差是预测值与真实值偏差的平方与观测次数 n 比值的平方根。 衡量的是预测值与真实值之间的偏差，并且对数据中的异常值较为敏感。

(2) mse：mean squared error（均方误差）。在统计学和机器学习中，均方误差是一种衡量预测值与真实值之间差异的度量。它计算预测值与真实值之间差异的平方的平均值。均方误差越小，表示预测值与真实值越接近。

(3) r2：R-squared（R 平方）。R 平方又称为决定系数（coefficient of determination，R2）是反映模型拟合优度的重要的统计量，为回归平方和与总平方和之比。R2 取值在 0 到 1 之间，且无单位，其数值大小反映了回归贡献的相对程度，即在因变量 Y 的总变异中回归关系所能解释的百分比。R2 是最常用于评价回归模型优劣程度的指标，R2 越大（接近于 1），所拟合的回归方程越优。

(4) mae：mean absolute error（平均绝对误差）。在统计学和机器学习中，用于衡量预测值与真实值之间差异的指标。它计算预测值与真实值之间的绝对差异的平均值。

接下来使用 RegressionEvaluator 评估器对前面训练的线性回归模型进行评估，代码如下：

执行以上代码，输出结果如下：

```
模型的均方根误差(RMSE)=33920.53031807887
```

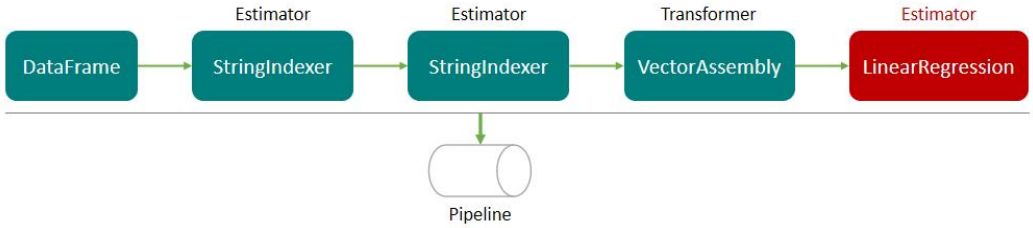
RMSE 代表的是均方根误差（root-mean-square error）。在这种情况下，RMSE 值就大约在$33, 920 美元左右，这表明模型有较大的改进空间。

因为线性回归实际上是拟合出一个最优线性方程，因此可以从学习得到的线性回归模型中获得所拟合的最优线性方程的系数和截距。代码如下：

通过调用线性模型的 summary 方法，可获取模型在训练集上的摘要（如残差 residuals、均方差 mse、等 r 平方）。如果 hasSummary 为 false，则抛出异常。对 summary 方法的调用将返回一个 LinearRegressionTrainingSummary 实现，它封装了线性回归训练结果。目前，训练摘要忽略了除目标轨迹外的训练权值。下面的代码在训练集上总结模型并打印出一些指标：

6.5 应用机器学习管道

管道将多个 Transformer 转换器和 Estimator 估计器连接在一起，以指定用于训练和使用模型的机器学习工作流。如图 10-所示。



重构上一节的示例代码，把 StringIndexer、VectorAssembler 和 LinearRegressor 放在一个机器学习管道中。代码如下：

6.6 回归模型调优

Spark ML 支持 k-fold 交叉验证的技术，可以尝试不同的参数组合，以确定机器学习算法的哪个参数值产生最优模型。使用 k-fold 交叉验证，数据被随机分成 k 个分区。每个分区使用一次作为测试数据集，其余部分用于训练。然后使用训练集生成模型并使用测试集进行评估，从而得到 k 个模型精度（accuracy）测量值。导致最高精度（accuracy）测量值的模型参数产生最优模型。

Spark ML 通过 Pipeline 管道支持 k-fold 交叉验证，该管道使用一个称为网格搜索的过程来尝试不同的参数组合，可以在其中设置参数以在交叉验证工作流中进行测试。

下面的代码使用 ParamGridBuilder 来构造用于模型训练的参数网格。我们定义了一个 RegressionEvaluator，它将通过比较预测结果集中的标签列预测列（prediction 列）来评估模型。我们使用 CrossValidator 进行模型选择。CrossValidator 使用管道、参数网格和评估器来拟合训练数据集并返回最优模型。CrossValidator 使用 ParamGridBuilder 迭代 LinearRegressor 估计器的 maxIter 和 regParam 参数，并评估模型，每个参数值重复三次以获得可靠的结果。

6.7 模型保存和加载

现在，我们可以将拟合的最优模型保存到分布式文件存储中，以便以后在生产中使用。这样既保存了特征提取阶段，又保存了通过模型调优选择的线性回归模型。代码如下：

第 7 章 Spark ML 回归算法进阶

本章主要讲解 Spark ML 中关于其他回归算法（线性回归除外）的原理、实现和应用。包括：广义线性回归、决策树回归、生存回归、等渗回归等。而像随机森林回归、梯度提升树回归等算法的原理、实现和应用则放在第 10 章“集成学习技术”中。

7.1 广义线性回归

广义线性模型（generalized linear models，GLMs）估计回归模型的结果遵循指数分布。除了高斯分布（即正态分布），这些分布还包括泊松分布、二项分布和伽马分布。每个都有不同的用途，根据分布和链接函数的选择，可以用于预测或分类。

7.1.1 什么是广义线性回归

大家都知道，线性回归是通过建立了一个线性模型来预测样本的值。线性回归模型就是通过数据去拟合出一条直线，其公式可以写成如下形式：

$$y = B_0 + B_1 * x$$

其中，y 为回归值，即输出；B_1 为参数矩阵（即我们所估计的参数）；x 为输入数据；B_0 为截距（偏置）。

我们知道了”回归“一般是用于预测样本的值，这个值通常是连续的。但是受限于其连续的特性，一般用它来进行分类的效果往往很不理想。为了保留线性回归”简单效果有不错“的特点，又想让它能够进行分类，因此需要对预测值再做一次处理。这个多出来的处理过程，就是 GLM 所做的最主要的事。而处理过程的这个函数，我们把它叫做连接函数。

二分类逻辑回归就是在线性回归的基础上，通过 sigmoid 函数对线性回归的输出 y 进行映射，使映射后的数据符合伯努利分布，即输出值从连续值 y，变为 g(y)，其值仅为 0 和 1。

$$g(y) = \frac{1}{1+e^{-y}}$$

说到这里，大家可能已经大致明白了，广义线性回归，其实就是通过某个激活函数，将线性回归的输出值映射成不同的分布，比如说伯努利分布、泊松分布、伽马分布等。不同的分布对应不同的激活函数，并且通过不同的损失函数进行迭代训练，从而得到不同应用场景下的广义线性回归模型（可以根据目标值的分布情况选择不同的模型）。

如下图是一个广义模型的流程：

图中，当一个处理样本的回归模型是线性模型，且连接函数满足一定特性（特性下面说明）时，我们把模型叫做广义线性模型。因为广义模型的最后输出可以为离散，也可以为连续，因此，用广义模型进行分类、回归都是可以的。

但是为什么线性回归是广义线性模型的子类呢，因为连接函数是 f(x) = x 本身的时候，也就是不做任何处理时，它其实就是一个线性回归啦。

所以模型的问题就转化成获得合适的连接函数？以及有了连接函数，怎么求其预测函数？在广义线性模型中，为了提高可操作性，因此限定了 Y 和 X 的分布必须满足指数族分布。上面提到的激活函数和损失函数都可以通过指数分布族来描述，对于广义线性回归模型来说，指数分布族就是其内核，其公式如下：

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

其中θ是感兴趣的参数，τ是一个离散参数。在一个 GLM 中响应变量Y_i假设是从自然指数族分布中得出的：

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

与假设输出服从高斯分布的线性回归相反，广义线性模型是线性模型的规范，其中响应变量Y_i服从指数分布族的一些分布。通过给出线性预测器（链接函数）的符号描述和误差分布（族）的描述来拟合广义线性模型，Spark 的 GeneralizedLinearRegression 接口允许灵活地指定 GLMs，它可用于各种类型的预测问题，包括线性回归、泊松回归、逻辑回归等。

注意： Spark 目前通过它的 GeneralizedLinearRegression 接口只支持多达 4096 个特性，如果超过这个约束将抛出异常。

GLMs 需要指数族分布，这些指数族分布可以写成“canonical”或“natural”形式，也就是自然指数族分布。目前在 spark.ml 中，仅支持指数族分布的子集，在下面表 7-列出。

表 7-1 spark.ml 支持的指数族分布

分布族	响应变量类型	支持的链接
Gaussian	连续变量	Identity*, Log, Inverse
Binomial	二元变量	Logit*, Probit, CLogLog
Poisson	计数变量	Log*, Identity, Sqrt
Gamma	连续变量	Inverse*, Identity, Log
Tweedie	零膨胀连续变量	Power link function

其中带星号的连接为规范连接（Canonical Link）。

Spark 的广义线性回归接口还提供了用于诊断 GLM 模型拟合的汇总统计，包括残差、p 值、偏差、Akaike 信息准则等。

7.1.2 广义线性回归示例

下面的示例演示了如何使用高斯响应和 identity link 函数训练 GLM，并提取模型汇总统计信息。本示例中使用的数据集来自 Spark 安装包自带的 sample_linear_regression_data.txt 的数据文件。

接下来的示例中，重构上一节的二手房价格预测示例，使用 Spark ML 中的广义线性回归实现 GeneralizedLinearRegression 来进行预测。代码如下：

执行以上代码，输出内容如下：

7.2 决策树回归

决策树是一种有监督的机器学习算法。

7.2.1 决策树回归算法

决策树回归算法的概念与决策树分类算法的概念基本一致，最大的不同就是其划分标准不再是信息熵或基尼系数，而是均方差（MSE），其计算公式如下：

$$\mathrm{MSE}=\frac{1}{\mathrm{n}}\sum_{\mathrm{i}=1}^{n}(y^{(i)}-\hat{y}^{(i)})^2$$

其中，n 为样本数据，$y^{(i)}$ 为实际值，$\hat{\mathrm{y}}^{(i)}$ 为拟合值（预测值）。对于决策树回归算法而言，其目的就是使系统最终的 MSE 最小，其节点划分依据也是基于这个理念。

在决策树回归算法中，节点中所有数据的均值作为该节点的拟合值，对于最终的叶子节点来说，其拟合值就是最终的回归模型预测值。

决策树学习算法既可以用来做分类分析（即预测分类变量值），也可以用来做回归分析（即预测连续变量值）。在 Spark MLlib 库中提供了对决策树学习算法的实现，分别是决策树分类器 DecisionTreeClassifier 和决策树回归器 DecisionTreeRegressor，它们的学习结果（输出）分别是决策树分类模型 DecisionTreeClassificationModel 和决策树回归模型 DecisionTreeRegressionModel。本节主要关注 DecisionTreeRegressor 决策树回归器的使用。

7.2.2 决策树回归示例

想象一下，你正在为一家共享单车公司工作，该公司在某个城市的某个地区运营。这家公司有一个自行车共享计划，用户可以在一个特定的地点租一辆自行车，然后使用停车场的机器在不同的地点还车。你负责预测未来会有多少辆自行车被使用。这对于预测公司收入和规划基础设施改进显然是非常重要的。

不幸的是，你对共享单车了解不多。但是，公司可以提供一个 csv 文件，里面有每天租用的自行车数量。因此，我们可以使用决策树创建回归模型，以预测共享单车方案中租用的自行车数量。通过回归分析，可以在一个模型中捕获温度、湿度以及风速和租用的自行车数量之间的关系。

首先，导入相关的依赖包，然后加载该数据文件，代码如下：

```

```

执行以上代码，输出内容如下：

在该数据集中，各字段的含义如下：

(1) date：记录的日期。很明显，这个字段不作为特征列。

(2) temperaturel：当天的天气温度。

(3) humidity：当天的天气湿度。

(4) windspeed：当天的风速。

查看数据集的大小，代码如下：

```

```

执行以上代码，输出内容如下：

7.3 生存回归

对于长期纵向观察的随访研究，我们常常会用到生存分析。生存分析的一个很大优点在于，它不仅考虑到终点事件的发生与否，同时也将终点事件出现的时间考虑在内，有效的解决了在研究中时间因素所带来的影响。

7.3.1 什么是生存分析

生存分析是用来估计所研究的特定人群的寿命。它也被称为“事件时间-Time to Event”分析，因为目标是估计个人或一群人经历感兴趣的事件的时间。这个时间估计是出生和死亡事件之间的持续时间。

生存分析最初是由医学研究人员和数据分析师开发和使用的，用于测量特定人群的寿命。生存分析在医学研究中占有很大的比例。在医学中，生存分析是非常常见和重要的分析方法之一，是一种将事件的结果和出现这种结果所经历的时间结合起来一起分析的统计方法。但是，多年来，它已被用于各种其他应用，如预测流失的客户/员工，估计机器的使用寿命等。出生事件可以被认为是客户开始成为公司会员的时间，死亡事件可以被认为是客户离开公司的时间。

通常除了想要使用的持续时间之外，还有其他数据。这种技术被称为生存回归（survival regression）-这个名字意味着我们将协变量（例如，年龄、国家等）与另一个变量进行回归-通常是持续时间。

1. 数据

生存分析主要关注于处理一种特殊的时间数据，并且时间数据可能带有部分删失属性。

生存分析的数据和分类数据有点类似。但是生存分析多了一列关于时间数据，这列时间数据记录整个观察期间研究对象多久才发生失效事件（例如死亡事件）。生存分析就是研究生存时间和结局与众多影响因素间的关系。

在生存分析中，我们不需要确切的起点和终点。所有的观测并不总是从零开始。受试者可以在任何时间进入研究。所有持续时间都是相对的。所有受试者都被带到一个共同的起点，在那里时间 t 为 0 (t = 0)，所有受试者的生存概率都等于 1，即他们不经历感兴趣的事件（死亡，流失等）的几率是 100%。

可能会出现这样的情况，即数据量使其无法完全用于生存分析。对于这种情况，分层抽样可能会有所帮助。在分层抽样中，你的目标是从整个人口中每一组受试者中获得相等或接近相等数量的受试者。每一组被称为一个阶层。根据某些特征，整个人口被分成若干组。现在，为了从每组中选择一定数量的受试者，可以使用简单随机抽样。受试者的总人数是在开始时指定的，然后将所需的总人数分成每一组，然后从每一组中随机选择相应数量的受试者。

2. 删失

重要的是要明白，不是每个人都会在研究期间经历感兴趣的事件（死亡，流失等）。例如，在观察/研究期间，会有仍然是公司成员的客户，或者仍然为公司工作的员工，或者仍然在运行的机器。我们不知道他们什么时候会经历感兴趣的事件作为研究的时间。我们只知

道他们还没有经历过。他们的生存时间比他们在研究中的时间更长。因此，他们的生存时间被标记为“Censored（删失）”，这表明它们的生存时间被切断了。我们把这种类型的数据称之为删失数据。因此，删失制度允许我们测量尚未经历感兴趣事件的人口的寿命。

值得一提的是，没有经历感兴趣事件的人/受试者需要成为研究的一部分，因为完全删除他们会使结果偏向于研究中经历感兴趣事件的每个人。因此，我们不能忽略这些成员，而将他们与经历过感兴趣事件的人区分开来的唯一方法是使用一个变量来表示删失（censorship）或死亡（death，感兴趣的事件）。根据生存结局的发生情况，生存分析的数据常常分为终点事件（如死亡）和删失（其他生存结局）两类。

删失数据无法明确的观察记录到发生终点事件的生存时间，即真实的生存时间未知，只知道比观察到的删失时间要长。在生存分析中，发生终点事件记为 1，删失记为 0。

产生删失的可能原因有很多，包括：

(1) 到达研究截止日期时，终点事件仍然没有发生，研究对象依然存活。

(2) 研究对象因为搬迁、更换电话号码等原因失去联系造成失访，无法明确观察到研究对象是否发生了终点事件，以及具体的发生时间。

(3) 研究对象不配合、或医生改变治疗方案等其他原因，造成研究对象中途退出研究，无法继续进行随访观察。

(4) 研究对象死于其他事件，例如死于交通事故，或者因其他疾病造成死亡。

在生存分析中有不同类型的删失，如下所述：

(1) 右删失（right censored）

在进行随访观察中，研究对象观察的起始时间已知，但终点事件发生的时间未知，无法获取具体的生存时间，只知道生存时间大于观察时间，这种类型的生存时间称为右删失。右删失是实际研究中最常见的数据删失类型。

(2) 左删失（left censored）

假设研究对象在某一时刻开始进入研究接受观察，但是在该时间点之前，研究所感兴趣的时间点已经发生，但无法明确具体时间，这种类型即为左删失数据。

例如，某项关于脑卒中复发危险因素的研究，生存时间规定为从第一次脑卒中发病到下一次脑卒中发病之间的时间间隔。在研究起始时刻对研究对象进行问卷调查，询问是否发生过脑卒中，以及第一次脑卒中发病的时间，如果研究对象回答“发生过脑卒中，但不记得发病的具体时间了”，此时无法明确获取第一次脑卒中发病时间，该数据即为左删失。

(3) 区间删失（interval censored）

在实际的研究中，如果不能够进行连续的观察随访，只能预先设定观察时间点，研究人员仅能知道每个研究对象在两次随访区间内是否发生终点事件，而不知道准确的发生时间，这种删失类型称为区间删失。这种情况发生在随访期间，即两次观察之间的时间不是连续的。可以是每周、每月、每季度等等。

例如某个研究对象在第一次随访时间点未观察到终点事件发生，在下一次随访时已经发生了终点事件，但研究人员无法获取发生时间的具体时间，只知道在这两次随访间隔的中间，生存时间并不明确，因此则认为该研究对象的生存时间在两次随访间隔内是区间删失。

(4) 左截断（left truncation）

这被称为晚入（ late entry）。研究对象在进入研究之前可能已经经历过感兴趣的事件。如果我们填充截短的区域那么我们就会对诊断后早期的情况过于自信。这就是我们截断它们的原因。

简而言之，在研究期间没有经历过感兴趣事件的研究对象是右删失，出生未被看到的研究对象是左删失。生存分析的发展主要是为了解决右删失问题。

例如下面表格记录了 6 位癌症患者在 2 年观测期间中的存活时间和感兴趣的影响因素。这里的失效事件是死亡事件，是我们关心的。生存时间对应表格的生存时间。研究的影响因素有：年龄、性别、是否吸烟、肿瘤大小。

表 7-1 癌症患者在 2 年观测期间中的存活时间和感兴趣的影响因素统计表

年龄	性别	是否吸烟	肿瘤大小(mm)	生存时间(月)	事件
55	男	是	17	23	存活
65	女	是	20	24	存活
36	女	否	16	19	死亡
38	男	是	25	21	存活
69	男	否	16	23	存活
70	男	否	28	20	死亡

在表格中有个患者生存时间只有 21 个月，结局却是存活，可能由于是患者中途退出观察或者换了电话号码，不能联系患者，无法继续进行研究，这种数据属于删失数据，可见在现实生活中生存分析数据通常会含有删失数据。再例如在表格中有个患者生存时间是 19 个月，结局是死亡，这种数据属于完整数据。不过这些删失数据在进行生存分析时，也是会一起进行分析。

3. 生存函数

在生存分析中，生存函数用于估计在某个时间点之后的生存概率。生存函数通常用 S(t)表示：S(t) = P(T>t)，其中 t 表示时间。生存函数定义了目标事件在时间 t 未发生的概率，也可以解释为时间 t 后的生存概率。这里，T 是从总体中取的随机寿命，它不可能是负的。注意 S(t)在 0 到 1 之间(含 1)，S(t)是 t 的非递增函数。

4. 风险函数

风险函数（hazard function）也称为强度函数（intensity function），定义为受试者在短时间间隔内经历感兴趣事件的概率，前提是该个体存活到该时间间隔开始时。它是在一段时间内计算的瞬时速率，该速率被认为是恒定的。它也可以被认为是在时刻 t 经历感兴趣事件的风险。它是在时间 t 开始的区间内经历事件的研究对象人数除以时间 t 存活的研究对象人数与区间宽度的乘积。

因为连续随机变量等于某一特定值的概率为零。这就是为什么我们考虑事件在特定时间间隔内发生的概率从 T 到(T + ΔT)。因为我们的目标是找到事件的风险，我们不希望风险随着时间间隔ΔT 变大而变大。因此，为了进行调整，我们将等式除以ΔT。这将方程缩放为ΔT。危险率方程为：

$$h(t)=\lim_{\delta t->0}\frac{\Pr(t\le T\le t+\delta t\mid T>t)}{\delta t}$$

极限ΔT趋近于零意味着我们的目标是度量事件在特定时间点发生的风险。因此，取极限ΔT趋近于零，可以得到一个无限小的时间段。

这里需要指出的一点是，风险不是一个概率。这是因为，尽管分子上有概率，但分母上的ΔT可能会得到一个大于1的值。

5. 生存回归

生存回归不仅包括使用持续时间和删失变量，还使用其他数据（性别、年龄、工资等）作为协变量。我们将这些协变量与持续时间变量进行“回归”。

用于生存回归的数据集需要采用DataFrame的形式，其中一列表示研究对象的持续时间，一列表示是否观察到感兴趣的事件，以及需要回归的其他协变量。与其他回归技术一样，我们需要在将数据提供给模型之前对其进行预处理。

6. 生存分析方法

生存分析是指根据试验或调查得到的数据对生物或人的生存时间进行分析和推断，研究生存时间和结局与众多影响因素间关系及其程度大小的方法，也称生存率分析或存活率分析。

生存分析方法大体上可分为三类：非参数法、半参数方法和参数法。用Kaplan-Meier曲线（也称乘积极限法）和寿命表法估计生存率和中位生存时间等是非参数的方法，半参数方法指Cox比例风险模型，参数方法指指数模型、Weibull模型、Gompertz模型等分析方法。

参数方法要求观察的生存事件服从某一特定的分布，采用估计分布中参数的方法获得生存率的估计值。生存事件的分布可能为指数分布、weibull分布、对数正态分布等，这些分布曲线都有相应的生存率函数形式，只需求的相应参数的估计值，即可获得生存率的估计值和生存曲线。

7.3.2 生存回归模型

生存模型将某些事件发生之前经过的时间与一个或多个可能与该时间量相关的协变量联系起来。

在生存回归中有几种流行的模型：Cox模型、加速失效模型（accelerated failure models）和Aalen加性模型（Aalen's additive model）。所有模型都试图将风险率h(t|x)表示为t和某些协变量x的函数。

这里我们重点讨论加速失效时间回归模型（Accelerated Failure Time Regression Model），通常简称为AFT模型。因为Spark的机器学习库中实现的是AFT模型。

如果我们有两个独立的种群A和B，每个种群都有自己的生存函数，分别由SA(t)和SB(t)给出，并且它们通过加速失效率λ相互关联，公式如下：

$$S_A(t) = S_B\left(\frac{t}{\lambda}\right)$$

它可以减缓或加速沿生存函数的移动。λ可以建模为协变量的函数。它将生存时间的延长或缩短描述为预测变量的函数，公式如下：

$$S_A(t) = S_B\left(\frac{t}{\lambda(x)}\right)$$

其中：

$$\lambda(x) = \exp\left(b_0 + \sum_{i=1}^{n} b_i x_i\right)$$

根据受试者的协变量，该模型可以加速或减速失效时间。xi 的增加意味着平均/中位生存时间发生 exp(bi)因子的变化。然后我们为生存函数选择一个参数形式。

AFT 模型将线性回归模型的建模方法引入到生存分析的领域， 将生存时间的对数作为反应变量，研究多协变量与对数生存时间之间的回归关系，在形式上，模型与一般的线性回归模型相似。对回归系数的解释也与一般的线性回归模型相似。相较于其他生存模型， AFT 模型对分析结果的解释更加简单、直观且易于理解，并且可以预测个体的生存时间。

7.3.3 生存回归示例

在 spark.ml 的 org.apache.spark.ml.regression 包中，提供了 AFTSurvivalRegression 类，它实现了 AFT（Accelerated failure time，加速失效时间）模型，该模型是截尾数据的参数生存回归模型。它描述了一个生存时间的对数模型，所以它通常被称为生存分析的对数线性模型。与为相同目的而设计的比例风险（Proportional hazards）模型不同，AFT 模型更容易并行化，因为每个实例对目标函数的贡献是独立的。

在生存分析的统计领域，加速失效时间模型（AFT 模型）是一种参数化模型，它为常用的比例风险模型提供了一种替代方案。比例风险模型假设协变量的作用是将风险乘以某个常数，而 AFT 模型假设协变量的作用是通过某个常数加速或减慢疾病的生命过程。这在技术背景下尤其具有吸引力，因为“疾病”是一些具有已知中间阶段序列的机械过程的结果。

一般情况下，加速失效时间模型可表示为：

$$\lambda(t \mid \theta) = \theta\lambda_0(\theta t)$$

在该公式中， θ 表示协变量的联合效应，

最常用的 AFT 模型是基于生存时间的威布尔分布（Weibull distribution）。寿命的威布尔分布对应于寿命对数的极值分布。AFT 模型可以表述为一个凸优化问题，即寻找一个凸函数的最小值的任务。实现的优化算法是 L-BFGS。

下面是 Spark 自带的一个使用 AFT 模型来处理生存数据的示例程序。

【示例 7-】使用 AFT 模型来处理生存数据。

首先，创建一个 DataFrame，代码如下：

```

```

在上面的数据集中，各个列的含义如下：

(1) label 列表示的是存活的时间。

(2) censor 列是事件列，表示被观察对象的结局，1 表示死亡，0 表示删失数据（病历失访或者尚存活）。

(3) features 是特征向量列，即其他协变量，最终都要转化为数据的形式（归一化）。

然后创建 AFT 生存回归模型实例对象，并指定分位数概率数组参数。分位数概率数组的值应在范围内（0，1），数组应该是非空的。通过学习训练数据集，基于生存时间的威布尔分布拟合参数化生存回归模型（即加速失效时间(AFT)模型）。代码如下：

返回的 model 对象是 AFTSurvivalRegressionModel 类型的实例。下面打印 AFT 生存回归模型的系数、截距和比例参数，代码如下：

在下面这个示例中，对 KMsurv 库提供的 bfeed 数据集执行生存分析。bfeed 数据集包括 927 名选择母乳喂养孩子的母亲的第一个孩子的信息。该数据集记录了母乳喂养的时间长短以及与母亲怀孕和生活方式有关的几个因素。数据由 9 个变量和 922 个观测值组成。其中部分数据如下：

```
label,censor,race,poverty,smoke,alcohol,agemth,ybirth,yschool,pc3mth
16,1,1,0,0,1,24,82,14,0
1,1,1,0,1,0,26,85,12,0
4,0,1,0,0,0,25,85,12,0
3,1,1,0,1,1,21,85,9,0
36,1,1,0,1,0,22,82,12,0
36,1,1,0,0,0,18,82,11,0
16,1,1,1,1,0,20,81,9,0
8,0,1,0,1,0,24,85,12,0
20,1,1,1,0,0,24,85,12,0
44,1,1,0,0,0,24,82,14,0
......
```

该数据集包含以下列：

【示例】使用 bfeed 数据集，根据不同的变量，如种族、贫困、是否吸烟、是否饮酒以及三个月后是否接受产前护理等，对母乳喂养时间的生存概率进行分析。

首先导入相关的依赖包，代码如下：

7.4 保序回归

保序回归（isotonic regression），是一种非参数回归方法，用于在给定自变量的条件下，找到一个单调递增或递减的因变量估计值。保序回归可用于估计变量之间的单调关系。

7.4.1 什么是保序回归

在统计学和数值分析中，保序回归（isotonic regression）或单调回归（monotonic regression）是一种将自由形式的线拟合到一系列观测值上的技术，使拟合的线在任何地方都是非递减的（或非递增的），并尽可能靠近观测值。

保序回归，正如它的名字，是一种对预测值施加了“保序”约束的一种回归分析方法。保序回归（红色实线）与线性回归在相同数据上的比较如图 7-所示：

两者都适合最小化均方误差（MSE）。保序回归的自由形式特性意味着当数据更陡峭时，直线可以更陡峭；保序性约束意味着线不减小。

保序回归可应用多个不同的场景，包括：

(1) 保序回归在统计推断中有应用。例如，人们可以用它来拟合一组实验结果的均值的等渗曲线，当这些均值按照某种特定的顺序增加时。保序回归的一个好处是，只要函数是单调递增的，它就不受任何函数形式的约束，例如线性回归所施加的线性。

(2) 另一个应用是非度量的多维尺度，其中寻求数据点的低维嵌入，使得嵌入点之间的距离顺序与点之间的不相似度顺序相匹配。保序回归迭代拟合理想距离，以保持相对不同的顺序。

(3) 保序回归也用于概率分类，以校准监督机器学习模型的预测概率。

保序回归的思路是将线性回归模型分成多段，这个需要提前设置一个超参数 n，相当于将区域分成 n 段，每段一个预测器，预测该区域内的点所对应的值，最后得到模型函数，模型加入了函数必须是单调递增的先验条件。

7.4.2 保序回归模型

保序回归模型的定义如下：

$$y_i = f(x_i) + \varepsilon$$

其中 x_i 为估计值， y_i 为真实值，f 为估计值到真实值的映射， ε 为误差。

形式上保序回归是一个这样的问题：给定表示观测响应的有限实数集 $Y = y_1, y_2, ..., y_n$，拟合未知的响应值 $X = x_1, x_2, ..., x_n$，求一个使其误差最小的函数：

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

要求遵循 $x_1 \le x_2 \le ... \le x_n$ 的规则，其中 w_i 是正权重。得到的函数称为保序回归，它是唯一的。它可以看作是有顺序限制的最小二乘问题。保序回归本质上是一个最拟合原始数据点的单调函数。保序回归的一种求解算法称为 PAV 算法，时间复杂度为 o(n)。

为了理解保序回归算法，我们假设给定一个无序数字序列，要求不改变每个元素的位置，但可以修改每个元素的值，修改后得到一个非递减序列，问如何使误差（该处取平方差）最小？保序回归法从该序列的首元素往后观察，一旦出现乱序现象停止该轮观察，从该乱序元素开始逐个吸收元素组成一个序列，直到该序列所有元素的平均值小于或等于下一个待吸收的元素。

例如，有一个原始序列<9, 14, 10>，从前往后观察，发现从 14 开始乱序。为此，从 14 开始，逐个吸收元素组成一个序列，直到该序列所有元素的平均值小于或等于下一个待吸收的元素 12。最后生成的结果序列为<9, 12, 12>。

而假设有另一个原始序列<9, 10, 14>，从 9 往后观察，到最后的元素 14 都未发现乱序情况，因此不用处理。

7.4.3 保序回归示例

Spark 机器学习库提供了保序回归算法类 IsotonicRegression，它位于 spark.ml 的 org.apache.spark.ml.regression 包中。它实现了一个 PAV（pool adjacent violators）算法，该算法使用并行化保序回归的方法，只支持单变量（单个特征）算法。训练输入是一个包含 label、features 和 weight 三列的 DataFrame。此外，IsotonicRegression 算法有一个可选参数 isotonic，默认为 true。此参数指定保序回归是单调递增还是单调递减。

训练返回一个 IsotonicRegressionModel，该模型可用于预测已知和未知特征的标签。保序回归的结果被视为分段线性函数。因此，预测的规则是：

(1) 如果预测输入与训练特征完全匹配，则返回相关的预测。如果有多个具有相同特征的预测，则返回其中一个。

(2) 如果预测输入低于或高于所有训练特征，则分别返回具有最低或最高特征的预测。如果有多个具有相同特征的预测，则分别返回最低或最高的预测。

(3) 如果预测输入落在两个训练特征之间，则将预测作为分段线性函数处理，并从最接近的两个特征的预测计算插值值。如果有多个值具有相同的特征，则使用与前一点相同的规则。

下面是 Spark 自带的一个使用保序回归模型的示例程序。

【示例 7-】使用 IsotonicRegression 模型来处理数据。

在该示例中，训练了一个 IsotonicRegression 回归模型，并进行预测。代码如下：

```
```

7.5 因子分解机回归

因子分解机（Factorization machine，FM）是一种在高稀疏度下估计参数的预测模型。该模型结合了支持向量机的优点，采用因子化参数代替了支持向量机的密集参数化。FM 是一种监督学习算法，可用于机器学习中的分类、回归和推荐系统任务。在 5.5 节中，我们已经学习并了解了因子分解机。因子分解机算法原理，请参考 5.5.1 节“因子分解机算法简介”。因子分解机特征处理，请参考 5.5.2 节“因子分解机特征工程”。

7.5.1 因子分解机模型

我们知道，即使在具有巨大稀疏性的问题中（如广告和推荐系统），因子分解机也能够估计特征之间的相互作用。Spark 机器学习库中提供了一个 FMRegressor 类来实现用于回归任务的分解机。下面的示例以 LibSVM 格式加载数据集，将其分成训练集和测试集，在第一个数据集上进行训练，然后在测试集上进行评估。我们将特征缩放到 0 到 1 之间，以防止梯度爆炸问题。

```
```

因为输出内容过多，这里不予展示。请读者执行以上代码并自行查看结果。

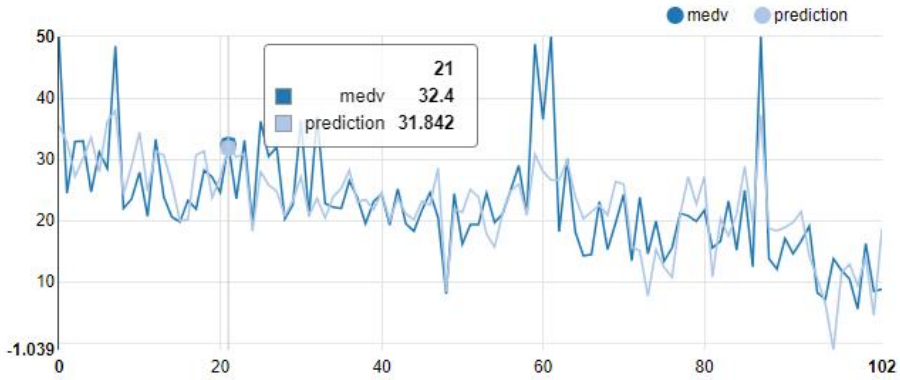
7.5.2 因子分解机示例

在下面这个示例中，我们将简要学习如何在使用 Spark FMRegressor 拟合和预测回归数据。本示例使用波士顿房价数据集。

首先，导入相关依赖包，读取数据集构造为 DataFrame。代码如下：

```
```

执行以上代码，可视化图形如下：



第 8 章 Spark ML 聚类算法

聚类的任务是基于一些相似度指标将一组示例分组到几个组中。目前，从客户细分到异常检测，在各种用例中都使用了集群。企业广泛使用机器学习驱动的集群来分析客户和细分，从而围绕这些结果创建市场策略。聚类通过在一个聚类中发现相似的对象和距离较远的不相似的对象来驱动许多搜索引擎的结果。它根据搜索查询推荐最近的相似结果。

8.1 聚类任务概述

所谓聚类问题，就是给定一个元素集合 D，其中每个元素具有 n 个可观察属性，使用某种算法将 D 划分成 k 个子集，要求每个子集内部的元素之间相异度尽可能低，而不同子集的元素相异度尽可能高。其中每个子集叫做一个簇（cluster）。聚类模型在许多方面都是分类模型的无监督等效物。通过分类，我们试图学习一个模型，该模型可以预测给定的训练示例属于哪个类别。模型本质上是一组特征到类别的映射。

注意： 聚类（clustering）与分类（classification）的不同之处在于：分类是一种示例式的有监督学习算法，它要求必须事先明确知道各个类别的信息，并且断言所有待分类项都有一个类别与之对应--很多时候这个条件是不成立的，尤其是面对海量数据的时候。而聚类是一种观察式的无监督学习算法，在聚类之前可以不知道类别甚至不给定类别数量，由算法通过对样本数据的特征进行观察，然后进行相似度或相异度的分析，从而达到“物以类聚”的目的。

在聚类中，我们希望对数据进行分段，以便将每个训练示例分配给称为"集群"的分段。除了真正的类分配是未知的以外，集群的作用与类很相似。例如，有一群人，每个人的胸牌上都写着“性别：男/女；年龄：XX”，我们可以根据性别把这堆人分为男、女两个子集，或者根据年龄把他们分为老、中、青、少四个子集。

聚类模型有许多与分类相同的应用，这些包括：

(1) 客户细分：根据行为特征和元数据将用户或客户划分为不同的组。
(2) 内容分类：将网站上的内容或零售业务中的产品分组。
(3) 基因分组：发现相似基因簇。
(4) 生态群落：生态学中群落的分割。
(5) 物品检测：创建用于图像分析应用程序(如对象检测)的图像段。

8.2 理解 K-Means 聚类算法原理

聚类算法是一种无监督的学习方法，这意味着与监督学习分类不同，样本示例在被分组（分类）之前没有被事先标记：聚类算法会学习标签本身。

聚类数据集缺少标签的原因可能是因为它太耗费金钱和时间来标记的所有数据（例如，当分组搜索结果时）或群组（类别）事先不知道（例如，市场细分），我们想要该算法帮我们找到群组，这样我们就可以更好地理解数据了。

例如，一种分类算法会得到一组标记为猫和狗的图像做为训练数据，并学习如何在未来的图像中识别猫和狗。聚类算法可以尝试识别不同图像之间的差异，并自动将它们分为两组，但不知道每个组的名称（是猫还是狗）。最多可以给他们贴上标签“第一组”和“第二组”。

聚类算法可以用于许多目的：

(1) 划分数据（例如，客户细分或通过类似的习惯对客户进行分组）。

(2) 图像分割（识别图像中的不同区域）。

(3) 检测异常。

(4) 文本分类或在一组文章中识别主题。

(5) 分组搜索结果（例如，有的搜索引擎会根据结果类别自动对其进行分组）。

K-Means 算法试图将一组数据点划分为 K 个不同的集群（其中 K 是模型的输入参数）。更正式地说，k-means 算法试图找到集群，以便最小化每个集群内的平方误差（或距离）。这个目标函数称为“集群内误差平方和(WCSS)”，它是每个集群上每个点和集群中心之间误差平方的总和。

1. K-Means 聚类算法实现过程

使用 K-Means 算法进行聚类，过程非常直观：

(a) 给定集合 D，有 n 个样本点。

(b) 随机指定两个点，作为两个子集的质心。

(c) 根据样本点与两个质心的距离远近，将每个样本点划归最近质心所在的子集。

(d) 对两个子集重新计算质心。

(e) 根据新的质心，重复操作(c)。

(f) 重复操作(d)和(e)，直至结果足够收敛或者不再变化。

整个过程如图 8-1 所示：

图 8-1 使用 K-Means 算法进行聚类过程

聚类的整体思想是最小化集群内距离，即数据点与集群质心的内部距离，最大化集群间距离，即两个不同集群的质心之间的距离。

当刚开始聚类时，每个数据点都是不同的，不属于任何集群（组）我们将它们分组的方式是，每个集群（组）包含最相似的记录，并且在任何两个集群（组）之间存在尽可能多的差异。那么，我们如何衡量两个数据点是相似的还是不相似的呢?

计算任意两个数据点之间的距离有多种方法。首先，我们将任何数据点都表示为包含该数据点的向量形式，如表 8-1 所示。

Age（年龄）	Salary（薪资，千）	Weight（体重，千克）	Hight（身高，英尺）
32	8	65	6

现在，假设我们想计算这个数据点（A）到任何其他数据点（B）的距离，数据点（B）也包含类似的属性，如表 8-2 所示。

Age（年龄）	Salary（薪资，千）	Weight（体重，千克）	Hight（身高，英尺）
40	15	90	5

我们可以用欧几里德方法来测量距离，它也被称为笛卡尔距离。我们计算任意两点之间的直线距离，如果这些点之间的距离很小，则它们很可能是相似的；而如果这些点之间的距离很大，则它们之间就是不相同的。如图 8-2 所示。

图 8-2 使用欧式距离计算两个数据点之间的相似度

任意两点之间的欧氏距离可用图 8-3 中所示公式计算：

图 8-3 两个向量间的欧式距离计算过程

因此，数据点 A 与数据点 B 之间的欧氏距离为 27.18。

2. K-Means 聚类算法实现示例

接下来通过一个示例来理解 K-means 聚类算法是如何工作的。在本示例中，我们将考虑一个样本数据集来，数据集包含多个用户，他们的年龄（Age）和体重（Weight）值如表 8-3 所示。

用户ID(UserID)	年龄(Age)	体重（Weight）
1	18	80
2	40	60
3	35	100
4	20	45
5	45	120
6	32	65
7	17	50
8	55	55
9	60	90
10	90	50

现在我们将使用 K-means 聚类来提出有意义的聚类并理解算法。

如果我们将这些用户绘制在二维空间中，我们可以看到没有一个点最初属于任何集群（组），我们的目的是在这个用户组中找到集群（可以尝试使用两个或三个），以便每个集群（组）包含相似的用户。每个用户由年龄和体重表示，如图 8-4 所示。

图 8-4

1）步骤一：决定 K 的值

从决定集群（组）的数量 K 开始。大多数情况下，我们一开始并不确定正确的集群（组）数，但是我们可以使用基于“Elbow 方法”的方法找到最佳的集群（组）数。对于本例，我们从 K=2 开始，以保持简单。因此，我们的目标是在这个示例数据中寻找两个集群（组）。

2）步骤二：随机初始化质心

接下来是随机选择任意两个点作为集群（组）的质心。这些可以随机选择，所以我们选择用户编号 5 和用户编号 10 作为新集群（组）上的两个中心点，如表 8-所示。

用户ID(UserID)	年龄(Age)	体重（Weight）
1	18	80
2	40	60
3	35	100
4	20	45
5（质心1）	**45**	**120**
6	32	65
7	17	50
8	55	55
9	60	90
10（质心2）	**90**	**50**

质心可以用体重和年龄值表示，如图 8-所示。

图 8-5 两个向量间的欧式距离计算过程

3）步骤三：为每个值分配集群（组）编号

在这一步中，我们计算每个数据点到质心的距离。在本例中，我们计算每个用户到两个质心点的欧式距离。根据距离值，决定用户属于哪个特定的集群（组 1 或组 2）。无论用户靠近哪个质心（更小的距离），都将成为该集群（组）的一部分。每个用户到两个质心的欧式距离的计算如表 8-所示。用户 5 和用户 10 到各自的质心的距离将为零，因为它们与质心是相同的点。

UserID	Age	Weight	到质心1的欧式距离	到质心2的欧式距离	集群（组）
1	18	80	48	78	1
2	40	60	60	51	2
3	35	100	22	74	1
4	20	45	79	70	2
5（质心1）	45	120	0	83	1
6	32	65	57	60	1
7	17	50	75	73	2
8	55	55	66	35	2
9	60	90	34	50	1
10（质心2）	90	50	83	0	2

因此，根据到质心的距离，我们将每个用户分配到集群（组）1 或集群（组）2。集群（组）1 包含 5 个用户，集群（组）2 也包含 5 个用户。初始集群如下图所示。

正如前面所讨论的，在包含或排除集群（组）中的新数据点之后，集群的中心质点必然会发生变化。由于早期的中心质点(C1, C2)不再位于集群（组）的中心，我们将在下一步计算新的中心质点。

4）步骤四：计算新的质心并重新分配集群（组）

K-Means 聚类的最后一步是计算聚类的新中心点，并根据与新中心点的距离将聚类编号重新赋值到每个值。我们来计算集群（组）1 和集群（组）2 的新质心。为了计算集群（组）1 的质心，我们只需取属于集群（组）1 的 Age 值和 Weight 的平均值，如下表所示。

用户ID(UserID)	年龄(Age)	体重（Weight）
1	18	80
3	35	100
5	45	120
6	32	65
9	60	90
均值	**38**	**91**

对集群（组）2 的质心计算也采用类似的方法，如下表所示。

用户ID(UserID)	年龄(Age)	体重（Weight）
2	40	60
4	20	45
7	17	50
8	55	55
10	90	50
均值	**44.4**	**52**

现在我们为每个集群（组）都计算了新的质心值，用一个叉符号表示，如下图所示。箭头表示质心在集群（组）中的移动。

对于每个集群（组）的质心，我们重复第 3 步，计算每个用户到新质心的欧氏距离，求出最近的质心。然后，根据距离质心的距离将用户重新分配到集群（组）1 或集群（组）2。在本例中，只有一个值（用户 6）将其集群（组）从 1 更改为 2，如下表所示。

UserID	Age	Weight	到质心1的欧式距离	到质心2的欧式距离	集群（组）
1	18	80	23	38	1
2	40	60	31	9	2
3	35	100	9	49	1
4	20	45	49	25	2
5	45	120	30	68	1
6	32	65	27	18	**2**
7	17	50	46	27	2
8	55	55	40	11	2
9	60	90	22	41	1
10	90	50	66	46	2

现在，集群（组）1 只剩下 4 个用户，而集群（组）2 根据与每个集群质心的距离包含 6 个用户，如下图所示。

我们不断重复上面的步骤，直到集群（组）的分配不再发生变化。新集群（组）的质心如表 8-和表 8-所示。

用户ID(UserID)	年龄(Age)	体重（Weight）
1	18	80
3	35	100
5	45	120
9	60	90
均值	**39.5**	**97.5**

用户ID(UserID)	年龄(Age)	体重（Weight）
2	40	60
4	20	45
6	32	65
7	17	50
8	55	55
10	90	50
均值	**42.33**	**54.17**

当我们执行这些步骤时，质心的移动会变得越来越小，这些值几乎成为该特定集群（组）的一部分，如下图所示。

我们可以看到，即使在质心改变之后，点也没有变化，这就完成了 K-Means 聚类。结果可能会有所不同，因为它是基于第一组随机选择的中心点。为了重现结果，我们也可以自己设置起点。带有值的最终集群（组）如下图所示。

从上述聚类结果可以看出，集群（组）1 包含的用户在 Age（年龄）属性上处于平均水平，但是在 Weight（体重）变量上似乎很高，而集群（组）2 似乎将这些用户分在了一组--这些用户 Age（年龄）平均较大，但是在 Weight（体重）上似乎很接近，如图 8-所示。

3. 如何决定集群（组）的数量 K

大多数时候，选择最优集群（组）数量是相当棘手的，因为我们需要对数据集和业务问题的背景有深入的理解。此外，当涉及到无监督学习时，并没有正确或错误的标准答案。如果换另一种方法，那么可能导致不同数量的集群（组）。我们必须尝试找出哪种方法最有效，以及所创建的集群（组）是否与决策足够相关。

每个集群（组）都可以用一些重要的属性表示，这些属性表示或能给出关于特定集群的信息。但是，有一种可以使用数据集选择最佳可能数的集群方法，这种方法被称为手肘法。

所谓手肘法，指的是我们初始先选择一个集群（组）的数量，比如 2，然后逐渐增加集群（组）的数量。对于每个选择的数量都训练出一个模型，并查看每个模型的 SSE 值（平方差的和）。SSE 值将不可避免地降低，因为随着群集数量的增加，方差会越来越小，而且距离中心的距离也会减少。极端情况下，如果集群（组）的数量与数据集中记录的数量相等，那么变异性将为零，因为每个点到自身的距离为零。如果把 SSE 作为一个群集数 K 的函数来绘制，则不同“K”值对应的 SSE 值如图 8-所示。

从图 8-中，我们可以观察到，在 K 值等于 3 和 4 之间有一种肘部的形成，在这些肘部处函数的斜率会突然改变，总方差（集群（组）内差异）突然减小，之后方差下降非常缓慢。事实上，当 K=9 时，曲线变平。因此，K =3 的值是最有意义的。

因此，对于 K 值的选择，我们使用手肘法，因为它捕捉的变异性最大，同时集群的数量较少。

8. 3 Spark ML 聚类算法实现

Spark 机器学习库中提供了以下聚类算法的实现：

(1) K-means clustering：K-means 聚类。

(2) Gaussian mixture model：高斯混合模型。

(3) Power-iteration clustering：谱聚类。

其中 K-means 聚类是三种方法中最简单的，也是最常用的。但是它也有缺点：它在处理非球形的群组和不均匀的群组（不均匀的密度或半径）方面有困难。它也不能有效地利用 one-hot-encoded 特征。它通常用于对文本文档进行分类，连同术语 frequency-inverse document frequency（TF-IDF）特征向量化方法。

而高斯混合模型（或混合高斯模型）是一种基于模型的聚类技术，这意味着每个群组都由高斯分布表示，而模型是这些分布的混合。它执行软聚类（soft clustering，建模一个例子属于一个群组的概率），不像 k-means，它执行硬聚类（hard clustering，建模一个例子是否属于一个群集）。因为高斯混合模型不能很好地扩展到有很多维度的数据集，因此我们不会在这里讨论它。

Power-iteration 聚类是一种谱聚类形式，它在 Spark 中的实现是基于 GraphX 库的。

K-means 是最常用的聚类算法之一，它将数据点聚到预定义数量的聚类中。Spark 机器学习库实现了包括 k-means++方法的并行化变体 kmeans‖。在 Spark 机器学习库中，KMeans 被实现为一个 Estimator，并生成一个 KMeansModel 作为基础模型。其输入列和输出列如表 8-所示。

表 8-1 KMeans 算法实现所需的输入列和生成的输出列说明

参数类型	参数名称	参数类型	默认值	描述
输入列	featuresCol	Vector	"features"	特征向量
输出列	predictionCol	Int	"prediction"	预测集群（组）中心

下面通过 Spark 自带的一个示例，来初步了解其 KMeans 算法实现类的使用方法。

8.4 示例：应用聚类算法对鸢尾花进行分类

Spark 机器学习库目前提供 K-Means 聚类实现。本节我们通过应用聚类算法对鸢尾花进行分类的示例，掌握 Spark 中 K-Means 聚类的使用。

本示例使用 UCI 数据集中的鸢尾花数据 Iris 进行分类。鸢尾花数据集 Iris 的字段说明如表 8-所示。

表 8-1 鸢尾花数据集字段说明

序号	字段	数据类型	单位	缺失值	字段说明
1	SepalLength	float	cm	无	花萼长度，连续变量
2	SepalWidth	float	cm	无	花萼宽度，连续变量
3	PetalLength	float	cm	无	花瓣长度，连续变量
4	PetalWidth	float	cm	无	花瓣宽度，连续变量
5	Class	int		无	标签列，类别变量。其中，0 代表 Iris Setosa，1 代表 Iris Versicolor，2 代表 Iris Virginica

Iris 数据的样本容量为 150，有四个实数值的特征，分别代表花朵四个部位的尺寸，以及该样本对应鸢尾花的亚种类型（共有 3 种亚种类型），如下所示：

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
......
```

注意： 详细了解鸢尾花数据集 Iris，可参考本书 4.3.2 节。

详细实现步骤及实现代码如下。

8.5 示例：应用聚类算法对学生的知识水平进行分类

美国当代著名教育家、心理学家布鲁姆的研究认为：“在学生前提行为（知识基础）和教学质量有利时，所有的学习结果将达到高的或积极的水平，学习结果的差异微不足道。如果学生前提行为（知识基础）存在很大差异，教学质量不能适应每个学生，那么学习结果之间存在很大差异。”他这里所说的教学质量是指教学适宜学习者的程度。

为此，我们需要了解学生的知识水平在哪个层次，进而制定合适的教学内容并采用合适的教学方法。在下面的例子使用 K-Means 聚类算法对学生的知识水平进行分类。这个例子所需要的数据集位于"源代码\data\sks\"目录下。数据以 CSV 格式提供，有两个文件：train.csv 和 test.csv。其中 train.csv 文件包含 label 列。数据集中包含对学生多维度评价的数据，如下所示：

```
......
```

第 9 章 Spark ML 推荐算法

推荐系统可以利用机器学习算法预测用户对商品的评分，是最直观、最著名的机器学习应用程序之一。推荐系统通过对用户历史行为的学习，从海量数据中自动推荐给用户可能感兴趣的内容。几乎每个人都在亚马逊、京东、淘宝和今日头条等热门网站（和 APP）上看到了推荐系统的例子。优秀的推荐系统可以极大地改善用户体验或者提升电子商务网站的收益，例如，亚马逊总收入中大约 30%是通过推荐系统获得的。

推荐引擎在以下两种不相互排斥的情况下最有效：

（1）有大量可供用户选择的选项：当有大量可用选项时，用户就会越来越难找到他们想要的东西。只有当用户知道他们在寻找什么时，搜索才是有帮助的，但是通常，正确的选项可能是用户以前不知道的。在这种情况下，推荐对于用户来说可能还不知道的相关选项可以帮助他们发现新的选项。

（2）涉及到很大程度上的个人品味：当个人品味在选择中扮演重要角色时，推荐模型可以帮助发现基于具有相似品味特征的其他人的行为的选项。

常用的推荐方法按照大类别划分有以下几种：基于群体统计的推荐、基于内容的推荐、基于协同过滤的推荐以及基于社交的推荐。

在各种推荐方法中，协同过滤是最常用也是在各种应用场景下非常有效的方法。本章讲介绍协同过滤算法的原理，以及 Spark 机器学习库中对协同过滤算法的实现类，并演示如何使用 Spark 提供的协同过滤算法类来构建一个电影推荐系统。

9.1 协同过滤算法介绍

在协同过滤中，推荐是基于其他用户的喜好。协同过滤分析用户之间的关系和产品之间的相互依赖关系，以识别新的“用户-产品（user-item）关联”。协同过滤仅依赖于过去的行为，如以前的评分、评级或交易。这背后的思想是相似性的概念。它的基本思想是，用户（users）可以隐式或显式地对产品（items）进行评价。该方法背后的直觉是“群体智慧”的概念，即过去有相似品味的用户在未来也会有相似的品味。

协同过滤方法是在收集和分析大量用户行为、活动或偏好信息的基础上，根据用户与其他用户的相似性来预测用户喜欢什么。协同过滤推荐依赖 user 与 item 间的联系，与物品自身特征没有太多关系。协同过滤方法的一个关键优势是它不依赖于机器可分析的内容，因此它能够准确地推荐复杂的 item，如电影，而不需要“了解”item 本身。

在协同过滤方法中，如果两个 user 表现出相似的偏好，即以大致相同的方式与相同的 item 进行交互的模式，那么我们假设他们在品味上是相似的。要为给定的 user 生成未知 items 的建议，我们可以使用具有类似行为的其他 user 的已知偏好，方法是通过选择一组相似的用户并根据他们所显示的偏好计算某种形式的综合得分来实现这一点。总的逻辑是，如果其他人的口味与一组产品相似，那么这些产品往往是很好的推荐对象。

尽管协作过滤通常比基于内容的技术更准确，但由于它无法处理系统的新产品和用户，因此存在所谓的冷启动问题。在这方面，基于内容的过滤更有优势。

协同过滤可进一步细分为基于用户的协同过滤（User-based KNN）、基于物品的协同过滤（Item-based KNN）、以及矩阵分解方法。其中，基于物品的协同过滤简单有效，而矩阵分解方法则具有推荐精度高等特点。

9.1.1 基于用户的协同过滤（UserCF）

基于用户的协同过滤（UserCF）的算法思想：人们往往会“趣味相投”，即兴趣相似的用户往往有相同的物品喜好。当目标用户需要个性化推荐时，可以先找到和目标用户有相似兴趣的用户群体，然后将这个用户群体喜欢的、而目标用户没有听说过的物品推荐给目标用户。例如，张三、李四是“臭味相投”的好友，张三喜欢抽烟、喝酒和烫头，李四喜欢抽烟和喝酒，那么根据此算法，推荐系统可向李四推荐烫头项目。

此算法的缺点：基于用户的协同过滤推荐非常不稳定。单个新评分或新用户可能会极大地影响所给出的建议。

9.1.2 基于物品的协同过滤（ItemCF）

基于物品的协同过滤（ItemCF）的算法思想：给目标用户推荐那些和他们之前喜欢的物品相似的物品。此算法并不利用物品的内容属性计算物品之间的相似度，而主要通过分析用户的行为记录来计算物品之间的相似度。该算法基于的假设是：物品 A 和物品 B 具有很大的相似度是因为喜欢物品 A 的用户大多也喜欢物品 B。例如，如果有两个用户对 A 书和 B 书给出了很高的评价，那么如果你购买了 A 书，你也会得到购买 B 书的推荐。

实现此算法的关键步骤：计算物品与物品间的相似度。此算法计算物品相似度是通过建立用户到物品倒排表（每个用户喜欢的物品的列表）来计算物品相似度矩阵。

9.1.3 选择 UserCF 还是 ItemCF？

UserCF 和 ItemCF 算法的主要区别：UserCF 算法推荐的是那些和目标用户有共同兴趣爱好的其他用户所喜欢的物品，ItemCF 算法则推荐那些和目标用户之前喜欢的物品类似的其他物品。因此，UserCF 算法的推荐更偏向于社会化，而 ItemCF 算法的推荐更偏向于个性化。

UserCF 算法适用于新闻推荐、微博话题推荐等应用场景，其推荐结果在新颖性方面有一定的优势，但推荐结果相关性较弱，容易受大众影响而推荐热门物品，同时此算法也很难对推荐结果作出解释。此外，新用户或低活跃用户会遇到"冷启动"问题，即无法的找到足够有效的相似用户来计算出合适的推荐结果。

ItemCF 算法适用于电子商务、电影、图书、音乐等应用场景，并且可以利用用户的历史行为给推荐结果作出解释。但此算法倾向于推荐与用户已购买商品相似的商品，往往会出现多样性不足、推荐新颖度较低的问题。

9.1.4 基于矩阵分解的协同过滤

在推荐系统中，我们常常遇到的问题是这样的，我们有很多用户和物品，也有少部分用户对少部分物品的评分，我们希望预测目标用户对其他未评分物品的评分，进而将评分高的物品推荐给目标用户。比如下面的用户物品评分表：

对于每个用户，我们希望较准确的预测出用户对未评分物品的评分。对于这个问题我们有很多解决方法，本文我们关注于用矩阵分解的方法来做。如果将 m 个用户和 n 个物品对应的评分看做一个矩阵 M，我们希望通过矩阵分解来解决这个问题。

1. 理解矩阵分解

矩阵分解是一种基于模型的协同过滤系统，使用 user-item 关联矩阵提出推荐。矩阵分解方法又称为因子模型，其算法思想是：其假设用户的特征可以使用若干潜在因子来进行刻画。同样，商品（也可以是新闻、电影、音乐等其他类型的内容）的特征可以用同样的潜在因子来进行刻画。当用户的特征和商品的特征比较吻合时，我们假设用户会给这个商品较高的评分，也即可以将这种商品推荐给用户。

基于矩阵分解的协同过滤的标准方法将 users-items 矩阵中的项视为 user 对 item 的显式首选项，例如，用户对电影进行评级。矩阵分解模型的目标是将这个稀疏 users-items 矩阵分解成一个用户矩阵 U（每一行代表该用户一个潜在的因素表示，也称为 user-embeddings）和业务矩阵 B（每一列表示该业务一个的潜在因素表示，也称为 business-embeddings），如图 8-所示：

构建协同过滤模型的一种广泛采用的方法是基于 ALS 算法的矩阵分解模型的协同过滤方法。ALS 是求解矩阵分解问题的一种优化技术，它代表的是 alternate-least-square（交替最小二乘）。该技术功能强大，实现了良好的性能，并且被证明相对容易以并行方式实现。ALS 的工作原理是迭代求解一系列最小二乘回归问题。在每次迭代中，一个用户（user）或物品（item）因子矩阵被视为固定的，而另一个则使用固定因子和评级数据进行更新。然后，解出的因子矩阵依次被视为固定的，而另一个则被更新。这个过程一直持续到模型聚合（或者对于固定数量的迭代）。这个算法需要的唯一输入是用户-物品评分矩阵，它用于通过一个称为矩阵分解的过程来发现用户的偏好和物品属性。一旦找到这两个信息，它们就被用来预测用户对未见过的物品的偏好。Spark 的机器学习库提供了 ALS 算法的实现。

2. 显式矩阵分解

当我们处理由用户自己提供的用户首选物品组成的数据时，我们引用显式首选物品数据。这包括，例如，用户给物品（items）的评分、喜欢、点赞等等。

我们可以使用这些评分，形成一个二维矩阵，用户 users 是行，物品 items 是列。每个单元项表示用户 user 对某一项 item 的评分。由于在大多数情况下，每个用户 user 只与相对较小的一组物品 items 项进行交互，所以这个矩阵非常稀疏。

不好的一面是，与最近邻模型相比，因子分解模型理解和解释起来相对更复杂，而且在模型的训练阶段通常计算量更大。

3. 隐式矩阵分解

然而，在实际场景中，很可能我们能够收集到的许多偏好数据是隐式反馈的，其中 user 和 item 之间的偏好项不是显式给定我们的，而是隐含在用户 users 可能与物品 items 的交互中的。这种情况通常包括两类数据：

（1）二元数据，例如用户是否观看了电影，是否购买了产品，等等；

（2）计数数据，例如一个用户观看一个电影的次数。

处理隐式数据有许多不同的方法。Spark 的机器学习库实现了一种特殊的方法，它将输入评分矩阵视为两个矩阵：二元偏好矩阵 P 和置信权重矩阵 C。

简而言之，矩阵分解方法通过从评分模式中推断出的因子向量来表示用户 user 和物品 item。用户 user 和物品 item 因子之间的高度信任或对应关系会导致推荐。两种主要的数据类型是显式反馈（如评分，由稀疏矩阵表示）和隐式反馈（如购买历史、搜索模式、浏览历史和点击流数据，由密集矩阵表示）。

9.2 Spark 协同过滤算法实现

Spark ML 机器学习库包含一个使用基于 ALS（交替最小二乘）算法的矩阵分解的协同过滤模型的实现。其中用户 user 和物品 item 通过一小组隐性因子进行表达，并且这些因子也用于预测缺失的元素。Spark 机器学习库使用交替最小二乘法（ALS）来学习这些隐性因子。

Spark MLlib 提供了一种协作过滤算法，可用于训练矩阵分解模型，该模型预测用户对推荐项目的显式或隐式评分。

9.2.1 Spark ALS 算法实现

Spark.ml 机器学习库目前支持基于矩阵分解（因子模型）的协同过滤实现，其中用户 user 和产品 item 由一组潜在因子描述，这些潜在因子可用于预测丢失的项。矩阵分解假设：

（1）每个用户 user 可以由一组 n 个属性或特征来描述。例如，用户的特征一可能是表示每个用户有多喜欢动作片的数字。

（2）每个物品 item 可以由一组 n 个属性或特征来描述。例如，电影的特征一可能是一个数字，表示电影与纯动作的距离有多近。

（3）如果我们将用户 user 的每个特征与物品 item 的相应特征相乘，并将所有特征相加，这将是用户 user 对该物品 item 的评分的一个很好的近似值。

另外，Spark.ml 中的 ALS 模型需要特定格式的数据（user, item, rating）。

9.2.2 模型超参数

Spark.ml 使用交替最小二乘（ALS）算法来学习这些潜在因子。Spark.ml 中的 ALS 算法实现有几个重要的超参数，其中包括：

（1）numUserBlocks：是为了并行化计算而将 users 划分为的块的数量（默认为 10）。

（2）numItemBlocks：是为了并行化计算而将 items 划分为的块的数量（默认为 10）。

（3）rank：是模型中潜在因子的数量（默认为 10），即使用的特征的数量。

（4）maxIter：是要运行的最大迭代数（默认为 10）。

（5）regParam：在 ALS 中指定正则化参数（默认为 1.0），用于解决稀疏矩阵的过拟合问题。

（6）implicitPrefs：是一个布尔类型的参数，决定是否使用隐式偏好（隐式反馈数据），默认为 false，即使用显式反馈数据。

（7）alpha：是一个 Double 类型的参数，隐式偏好公式中 alpha 参数的值（非负的）。默认值为 1.0。适用于 ALS 的隐式反馈变量，它控制偏好观察的基线置信度。

（8）nonnegativity：指定是否对最小二乘使用非负约束(默认为 false)。

（9）regParam：是一个 Double 类型的参数，用于正则化参数（>=0）。

（10）coldStartStrategy：冷启动策略。默认值是“drop”，将删除带有 NAN 值的预测，以便在非 NAN 值上计算评估度量。

（11）userCol、itemCol、ratingCol：分别对应 user、item、rating 输入列的名称。

可以调整这些参数，不断优化结果，使均方差变小。比如：imaxIter 越大，regParam 越小，均方差会越小，推荐结果较优。

注意： 基于 DataFrame 的 ALS API 目前只支持整数形式的 user id 和 item id。user 和 item id 列支持其他数字类型，但是 id 必须在整数值范围内。

9.2.3 冷启动策略

当使用训练出的推荐模型 ALSModel 进行预测时，通常会遇到在训练模型时训练数据集中没有出现的 user 和/或 item 的情况。这通常发生在两种情况下：

（1）在生产中，对于没有评分历史的新 users 或 items，并且模型没有在这些 users 或 items 上进行训练（这是“冷启动问题”）。

（2）在交叉验证期间，数据分割为训练集和评估集。当在 Spark 的 CrossValidator 或 TrainValidationSplit 中使用简单的随机分割时，在评估集中遇到不属于训练集中的 users 和/或 items。这实际上是非常常见的。

默认情况下，当一个 user 和/或 item 因子没有出现在模型中时，Spark 在 ALSModel.transform()期间分配 NaN 预测。这在生产系统中很有用，因为它指示了一个新 user 或 item，因此系统可以对一些反馈进行决策，以用作预测。

然而，在交叉验证期间，这是不可取的，因为任何 NaN 预测值都会导致评估度量的 NaN 结果（例如使用 RegressionEvaluator 时）。这将导致无法进行模型选择。

Spark 允许用户将 coldStartStrategy 参数设置为“drop”，以便在预测集 DataFrame 中删除任何包含 NaN 值的行。然后将对非 NaN 数据计算评估度量，该度量将是有效的。

注意： 目前支持的冷启动策略是 nan（上面提到的默认行为）和 drop。今后可能支持进一步的策略。

9.2.4 Spark.ml ALS 算法应用示例

在下面的示例中，我们使用 Spark 自带的电影评分数据集（来自著名的 MovieLens 数据集），每行包含一个用户 ID、一部电影 ID、该用户对该电影的一个评分以及一个时间戳。部分数据如下：

```
userId,movieId,rating,timestamp
0::2::3::1424380312
0::3::1::1424380312
0::5::2::1424380312
0::9::4::1424380312
0::11::1::1424380312
...
```

然后我们训练一个 ALS 模型，默认情况下，该模型假设评分是显式的（即超参数 implicitPrefs 为 false）。我们通过测量评分预测的均方根误差来评估推荐模型。步骤如下。

```

```

9.3 示例：构建幽默故事推荐系统

矩阵分解是一类用于推荐系统的协同过滤算法。矩阵分解将给定的等级矩阵近似为两个低秩矩阵的乘积。它将评级矩阵 R（n x m）分解为两个矩阵 W（n x d）和 U（m x d）的乘积。公式如下：

$$\mathrm{R}_{n\times m} \approx \hat{R} = V_{n\times k} \times V_{m\times k}^{T}$$

在 Spark 的机器学习库中，实现了基于矩阵分解的协同过滤算法 ALS。在本示例中，我们将学习使用 Scala 和 Apache Spark 构建推荐系统。使用来自 Kaggle 的 Jester1.7M 笑话评级数据集在 Spark 中构建推荐系统。该数据集包含来自 59,132 名用户的 150 个笑话的 170 多万个连续评分（分值从-10.00 到+10.00）。完整的数据集包含有两个 CSV 文件：

（1）jester_ratings.csv。部分内容如下：

```
......
```

数据集中每一行的格式为[User ID] [Item ID] [Rating]。

（2）jester_items.csv。部分内容如下：

```
......
```

数据集中每一项包含笑话 ID 和笑话内容的映射。注意，笑话内容（即 jokeText）部分包含有换行符和双引号等特殊符号，在加载时需要注意。

数据包含用户的阅读和点评历史记录。我们将建立一个推荐系统，向用户推荐新的幽默笑话故事。实现过程如下。

9.3.1 加载数据集

首先，导入本推荐系统中要用到的依赖库。代码如下：

```

```

执行以上代码，输出内容如下：

9.3.2 数据探索

查看评分数据集的一些基本统计数据。代码如下：

```
```

执行以上代码，输出内容如下：

9.3.3 拆分数据集

将数据集按 90/10 的比例拆分为训练集和测试集。代码如下：

```
```

9.3.4 模型训练和预测

接下来，构建一个 ALS 模型。代码如下：

```
```

执行以上代码，可以获得 NaN 预测的总数为 181 条。

9.3.5 模型评估

对于推荐模型的评估，我们使用 Spark 机器学习库中的 RegressionEvaluator 评估器。评估指标采用 RMSE。代码如下：

```
```

正常情况下，测试集上的 RMSE 是要高于训练集上的 RMSE 值的，正如以上输出所示。

9.3.6 模型调优

在机器学习领域，模型调优是提升模型性能的关键环节，而超参数调整是模型调优的核心环节。本示例中，我们使用交叉验证法对推荐模型进行调优。代码如下：

```
```

9.3.7 模型存储与加载

在 Spark 中，我们可以很容易地将调优后的模型持久存储，例如，存储到 HDFS 中。在稍后的时候，由其他人在其他应用中再加载该持久存储的模型，并部署应用。

上一节中，调优后获得的 cvModel 是一个 CrossValidatorModel 类的实例，我们可以将其直接存储到 HDFS 中指定的路径下。代码如下：

```
```

执行以上代码，输出内容如下：

```
```

9.3.8 推荐幽默故事

ASLModel 自带了推荐用户和推荐物品的方法，可以直接调用进行推荐。例如，为所有用户推荐预测评分最高的前 3 个幽默故事，代码如下：

```

```

执行以上代码，输出内容如下：

9.4 频繁模式挖掘

挖掘频繁项、项集、子序列或其他子结构通常是分析大规模数据集的第一步，多年来一直是数据挖掘领域的一个活跃研究课题。频繁模式挖掘在不同领域有各种实际应用，包括购物篮分析、客户行为分析、网络挖掘、生物信息学和网络流量分析。

9.4.1 理解频繁模式挖掘算法

频繁模式挖掘（又名关联规则挖掘）是一种分析过程，它从各种数据库（如关系数据库、事务数据库和其他数据存储库）中的数据集中发现频繁的模式、关联或因果结构。给定一组交易，此过程旨在找到规则，使我们能够根据交易中其他商品项的出现来预测特定商品项的出现。

支持度和置信度分别反映了所发现规则的有用性和确定性。通常，如果关联规则同时满足最小支持阈值和最小置信度阈值，则认为它们是有意义的。这些阈值可以由用户或领域专家设置。可以执行其他分析来发现相关项之间有意义的统计相关性。

9.4.2 频繁模式挖掘支持业务分析

这种分析方法可用于评估各种业务功能和行业的数据，可用于确定各种产品和服务的购买行为中的常见模式，可用于分析各种数据点之间的关系，以交叉销售和捆绑产品和服务，并了解目标受众。下面列出了常用的几种应用场景：

（1）购物篮数据分析

分析单个购物篮或单次购买中所购买物品的关联。

（2）交叉营销与销售

与其他企业（与自己的企业互补，而不是竞争对手）合作。例如，经常见到的汽车经销商和制造商与石油和天然气公司开展交叉营销活动。

（3）目录设计

企业目录中的商品选择通常被设计为相互补充，因此购买一种商品会导致购买另一种商品，因此这些商品通常是补充或密切相关的。

（4）医疗

将每个病人表示为包含有序疾病集的事务，并且可以预测哪些疾病可能同时/顺序发生。

9.4.3 Spark 实现 FPGrowth

有几种算法用于实现关联规则方法，通过查找频繁的项集来确定大型数据库中变量或项之间最感兴趣的关系。FP-growth（频繁模式增长）是一种高效且可扩展的算法，用于更有

可能在大型事务数据库中一起出现的提取项。此外还有其他算法，如 Apriori、MAFIA（maximum frequency Itemset Algorithm）和 Eclat。但 FP-growth 是最快的。

1. FP-Growth 算法

Han 等人的论文《挖掘不生成候选的频繁模式》描述了 FP-growth 算法，其中“FP”代表频繁模式（frequent pattern）。给定一个交易数据集，FP-growth 的第一步是计算项目频率并识别频繁项目。它的第二步使用后缀树（FP-tree）结构来编码事务，而不显式地生成候选集（这在内存和时间上都更昂贵）-这也是为什么它是最快的。在第二步之后，可以从 FP-tree 中提取频繁项集。

2. Spark PFP 算法实现

Spark 机器学习库（Spark .ml）中提供了 PFP 算法的内存实现。PFP 根据交易的后缀分配生长 FP-tree 的工作，因此比单机实现更具可伸缩性，便于在分布式计算环境中使用关联规则技术。Spark 提供的 PFP 支持 4 种编程语言，包括 Scala、Java、Phyton。除了传统 FP-growth 的默认设置（最小支持和最小置信度阈值）之外，spark.ml 包还接受 numPartitions 参数，该参数指定用于拆分作业的分区数量。该算法的优点是在执行时间和可伸缩性方面提供了更好的性能。

3. Spark PFP 使用示例

下面是 Spark 自带的一个 PFP 频繁项挖掘示例。

```
```

9.4.4 Spark 实现 PrefixSpan

PrefixSpan 是一种频繁顺序模式挖掘算法，是 Pei 等人（2001）提出的一种在序列数据库中发现序列模式的算法。

1. PrefixSpan 算法

上一节讲到的 FP-growth 算法都是挖掘频繁项集的，而 PrefixSpan 算法也是关联算法，但是它是挖掘频繁序列模式的。

首先我们看看项集数据和序列数据有什么不同，如下图所示。

图中左表的数据集就是项集数据，每个项集数据由若干项组成，这些项没有时间上的先后关系。而图中右表的序列数据则不一样，它是由若干数据项集组成的序列。比如第一个序列<a(abc)(ac)d(cf)>，它由 a、abc、ac、d、cf 共 5 个项集数据组成，并且这些项有时间上的先后关系。对于多于一个项的项集要加上括号，以便和其他的项集分开。同时由于项集内部是不区分先后顺序的，为了方便数据处理，我们一般将序列数据内所有的项集内部按字母顺序排序。

了解了序列数据的概念，再来看看什么是子序列。子序列和我们数学上的子集的概念很类似，也就是说，如果某个序列 A 所有的项集在序列 B 中的项集都可以找到，则 A 就是 B

的子序列。而频繁序列则和频繁项集很类似，也就是频繁出现的子序列。比如对于下图，支持度阈值定义为 50%，也就是需要出现两次的子序列才是频繁序列。而子序列<(ab)c>是频繁序列，因为它是图中的第一条数据和第三条序列数据的子序列，对应的位置用蓝色标示。

PrefixSpan 运行时最大的消耗在递归的构造投影数据库。如果序列数据集较大，项数种类较多时，算法运行速度会有明显下降。因此有一些 PrefixSpan 的改进版算法都是在优化构造投影数据库这一块。比如使用伪投影计数。因此使用大数据平台的分布式计算能力也是加快 PrefixSpan 运行速度一个好办法。

2. Spark 的 PrefixSpan 算法实现

Spark 的机器学习库内置了 PrefixSpan 算法的并行实现类 PrefixSpan。

由 spark.ml 提供的 PrefixSpan 实现接受以下参数：

（1）minSupport：被认为是一个频繁顺序模式所需的最低支持度。

（2）maxPatternLength：频繁顺序模式的最大长度。任何超过此长度的频繁模式将不包括在结果中。

（3）maxLocalProjDBSize：在开始对投影数据库进行局部迭代处理之前，前缀投影数据库中允许的最大项数。这个参数应该根据执行程序的大小进行调整。

（4）sequenceCol：数据集中序列列的名称（默认为“sequence”），该列中带空的行将被忽略。

3. Spark PrefixSpan 使用示例

下面是 Spark 自带的一个频繁顺序模式挖掘示例。

```

```

执行以上代码，输出内容如下：

```

```

9.5 使用 FP-Growth 的市场购物篮分析

当向购物者提供购买建议时，经营者通常会寻找经常一起购买的商品（例如方便面和火腿肠、洗发水和沐浴露等）。揭示不同商品之间关联的一项关键技术被称为购物篮分析。在您的推荐引擎工具箱中，由市场购物篮分析生成的关联规则（例如，如果一个人购买方便面，那么他可能会购买方便面）是一项重要而有用的技术。随着电子商务数据的快速增长，有必要对越来越大的数据量执行购物篮分析等模型。也就是说，拥有在分布式平台上生成关联规则所需的算法和基础设施非常重要。

9.5.1 什么是购物蓝分析

频繁项目集挖掘的一个典型例子是市场购物篮分析。使用市场购物篮分析方法来分析客户的购买模式，以识别商品之间的联系并增强销售策略。这个过程通过寻找顾客放在“购物篮”中的不同商品之间的联系来分析顾客的购买习惯（图 1）。

在购物篮分析中，事务库记录了顾客的购买行为。事务库中的每条记录称为一笔事务，而事务中的每样商品称为一个项，项的集合称为项集。

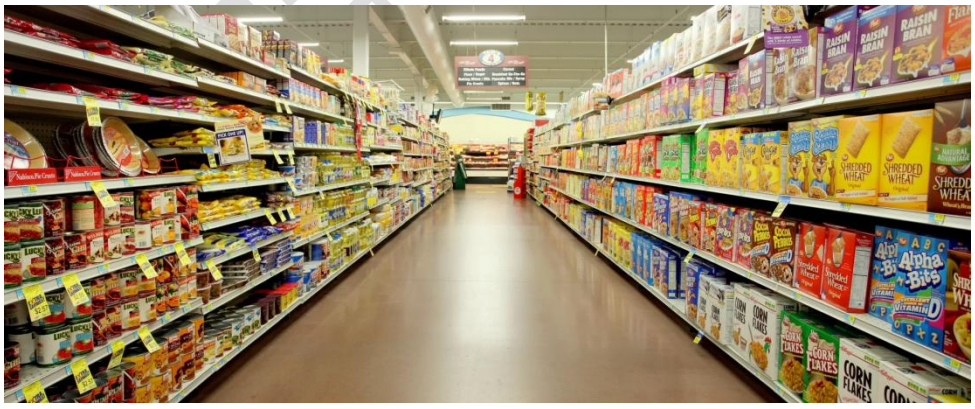
这种关联的发现可以帮助零售商通过洞察哪些商品经常被顾客一起购买来制定营销策略。例如，如果顾客购买了《Spark 原理深入与编程实战》这本书，那么他同时购买《PySpark 原理深入与编程实战》以及《Flink 原理深入与编程实战》这两本书的可能性有多大？

这实际上代表了这一类商业问题：一个零售商店经理想要进行市场购物篮分析，以提出更好的产品植入和产品捆绑策略。

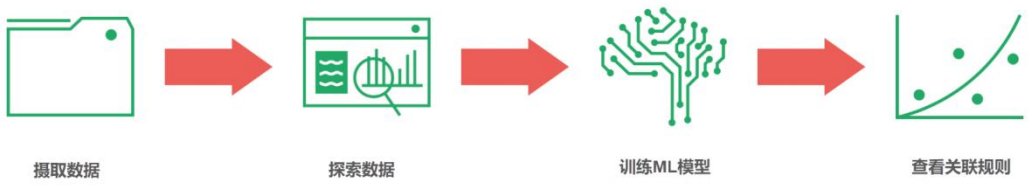
通过市场购物蓝分析，可获得的商业利益：根据生成的规则，商店经理可以战略性地将产品放在一起或按顺序放置，从而增加销售额，进而增加商店的收入。诸如“买这个送那个”或“买这个打 9 折”之类的优惠可以根据生成的规则来设计。

9.5.2 示例流程和数据说明

在本示例中，我们将演示如何探索购物车数据，并学习如何使用 Apache Spark MLlib FP-growth 算法执行购物篮分析，以识别经常一起购买的商品以及生成关联规则。



这个示例的流程如图 9-所示。



各个步骤说明如下：

（1）摄取数据。从源系统引入数据；通常涉及 ETL 过程。

（2）使用 Spark SQL 探索数据：在清理了数据之后是探索数据，以便洞察业务。

（3）使用 FP-growth 训练 ML 模型：执行 FP-growth 来执行频繁模式挖掘算法。

（4）查看由 ML 模型生成的关联规则以获得推荐。

```
...
```

9.5.3 数据摄取

数据分析的第一步，是摄取数据。将数据上传到 HDFS 的合适位置（为简单起见，本示例代码中使用本地文件系统，读者可根据情况调整）。

```

```

执行以上代码，输出结果如图 9-所示。

9.5.4 数据探索

现在既然已经创建了 dataframe，就可以使用 Spark SQL 执行探索性数据分析了。为了便于使用 SQL 执行数据探索，我们首先将以上加载的数据集注册到临时表中。代码如下：

```

```

执行以上代码，可视化结果如图 9-所示。

9.5.5 整理购物篮

为了为下游处理准备数据，我们将按购物篮组织数据。也就是说，DataFrame 的每一行都表示一个 order_id，每个 items 列都包含一个产品项数组。代码如下：

```

```

执行以上代码，输出结果如图 9-所示：

9.5.6 训练 ML 模型

为了了解商品相互关联的频率（例如，花生酱和果冻一起购买多少次），我们将使用关联规则挖掘进行市场购物篮分析。Spark MLlib 实现了两种与频率模式挖掘（FPM）相关的算法：FP-growth 和 PrefixSpan。区别在于 FP-growth 不使用项目集中的顺序信息（如果有的话），而 PrefixSpan 是为顺序模式挖掘而设计的，其中项目集是有序的。我们将使用 FP-growth，因为顺序信息对于这个应用示例并不重要。

```

```

执行以上代码，创建了 mostPopularItemInABasket DataFrame 后，我们可以使用 Spark SQL 查询购物篮中最受欢迎的商品项-包含 2 个以上商品的项。首先将计算出的频繁项集注册到临时表中，代码如下：

```

```

执行以上代码，输出结果如图 9-所示。

9.5.7 查看关联规则

除了 freqItemSets 之外，FP-growth 模型还生成 associationRules。例如，如果一个购物者买了花生酱（peanut butter），那么他还会买果冻（jelly）的概率（或信心）是多少？获得关联规则的代码如下：

```

```

执行以上代码，输出结果如图 9-所示。

从结果可以看出，如果购物者的篮子里有有机覆盆子（organic raspberries）、有机牛油果（organic avocados）和有机草莓（organic strawberries），那么推荐有机香蕉（organic bananas）也是有意义的，这是相对较强的信心。有意思的是，前 10 条关联规则（基于置信度下降）—即购买建议—均与有机香蕉或香蕉有关。

第 10 章 Spark ML 集成学习算法

集成学习（ensemble learning）是机器学习中的一种思想，它通过多个模型的组合形成一个精度更高的模型，参与组合的模型被称为弱学习器，最终形成的模型被称为强学习器。在预测时使用这些弱学习器模型联合进行预测：训练时需要用训练样本集依次训练出这些弱学习器。

在了解随机森林算法在机器学习中的工作原理之前，我们必须了解集成学习技术。

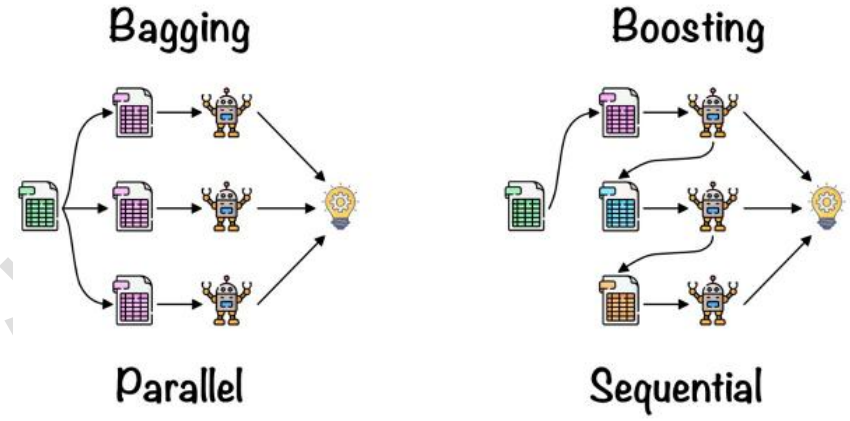
10.1 集成学习算法概述

在日常生活中人们会遇到这样的情况：对一个决策问题，如果一个人拿不定主意，可以组织多个人来集体决策。如果要判断一个病人是否患有某种疑难疾病，可以组织一批医生来会诊，会诊的做法是让每个医生做一个判断，然后收集他们的判断结果进行投票协商，得票最多的那个判断结果作为最终的结果。这种思想在机器学习领域的应用就是集成学习算法。

集成（ensemble）意味着组合多个模型。因此，用于预测的是一组模型，而不是单个模型。集成（ensemble）使用两种类型的方法：

(1) 装袋（bagging）：它使用有放回抽样从从样本训练数据中创建了一个不同的训练子集，最终的结果基于多数投票。例如，随机森林算法。

(2) 提升（boosting）：它通过创建顺序模型将弱学习器组合成强学习器，从而使最终模型具有最高的准确性。例如，梯度提升树分类器、XGBoost 算法等。



随机森林和提升算法都是流行的集成学习技术，它们使用单个学习器树来提高预测性能。随机森林的工作原理是 Bagging 原则，独立训练每棵树以组合它们的预测，而提升算法使用加法方法，其中弱学习器按顺序训练以纠正先前模型的错误。

10.1.1 装袋（Bagging）

装袋（Bagging），也称为 Bootstrap Aggregation（自举聚合），是在 Bootstrap 抽样的基础上构造出的 Bagging 算法。

注意： Bootstrap 抽样是一种数据抽样方法。抽样是指从一个样本数据集中随机选取一些样本，形成新的数据集。这里有两种选择：有放回抽样和无放回抽样。对于前者，一个样本被

抽中之后会被放回去，在下次抽样时还有机会被抽中。对于后者，一个样本被抽中之后就从抽样集中去除了，下次不会再参与抽样，因此，一个样本最多只会被抽中一次。在这里Botstrap抽样使用的是有放回抽样。

以下是装袋的步骤：

（1）选择子集。Bagging 首先从整个数据集中选择一个随机样本或子集。

（2）Bootstrap 抽样。然后从这些样本（称为 Bootstrap 样本，这些样本使用有放回抽样取自原始数据，这个过程称为行抽样）中创建每个模型。

（3）Bootstrapping：自举。用有放回行抽样的步骤称为自举。

（4）独立模型训练：每个模型在其相应的 Bootstrap 样本上独立训练。这个训练过程为每个模型生成结果。

（5）多数投票：最终的输出是通过多数投票将所有模型的结果结合起来决定的。选择模型中最普遍的预测结果。

（6）Aggregation：聚合。这一步涉及到组合所有结果并基于多数投票生成最终输出，这一步被称为聚合（aggregation）。

以上步骤可表达如图 10-所示。

现在让我们在下图的帮助下分析一个示例。这里的 bootstrap 样本取自实际数据（bootstrap 样本 01、bootstrap 样本 02 和 bootstrap 样本 03），并进行了替换（即有放回样本），这意味着每个样本很可能不包含唯一的数据。从这个 bootstrap 样本中得到的模型（model 01、model 02、model 03）是独立训练的。每个模型生成的结果如图 10-所示。现在，与悲伤表情相比，快乐表情占多数。因此，基于多数投票的最终输出是快乐表情。

10.1.2 提升（Boosting）

提升方法（Boosting）是一种机器学习中的集成学习启发算法，主要用来减小监督式学习中偏差并且也减小方差，以及一系列将弱学习器转换为强学习器的机器学习算法。弱学习器一般是指一个分类器，它的结果只比随机分类好一点点；强学习器指分类器的结果非常接近真值。

Boosting 是使用集成学习概念的技术之一。Boosting 算法结合多个简单模型（也称为弱学习器（weak learners）或基估计器（base estimators））来生成最终输出。Boosting 是一种递进的组合方式，每一个新的分类器都在前一个分类器的预测结果上改进。它是通过串联弱模型建立模型来实现的。

决策树容易过拟合，集成方法（例如增强方法）通常可用于创建更为稳健的模型。提升将多个单独的弱树（即性能略好于随机机会的模型）组合在一起，以形成强学习器。每个弱学习器都经过顺序训练，以纠正先前模型所犯的错误。经过数百次迭代后，弱学习者转换为强学习者。Boosting 方法背后的直观想法是：

（1）我们需要串行训练模型，而不是并行训练。

（2）每个模型需要重点关注之前的分类器表现不佳的地方。

Boosting 的算法过程如下：

（1）对于训练集中的每个样本建立权值 w_i，表示对每个样本的关注度。当某个样本被误分类的概率很高时，需要加大对该样本的权值。

（2）进行迭代的过程中，每一步迭代都是一个弱分类器。我们需要用某种策略将其组合，作为最终模型。

除此之外还有其他 Boosting 技术：GBM、XGBoost、LightGBM 和 CatBoost。

10.2 随机森林算法

随机森林（Random Forest）是由 Leo Breiman 和 Adele Cutler 开发的一种广泛使用的机器学习算法，它建立在多棵决策树作为基础模型之上，每棵决策树都在数据的不同自举子集上并行训练，最终将多个决策树的输出组合在一起以获得单个结果。它的易用性和灵活性，加上它作为随机森林分类器的有效性，以及它可同时用于处理分类和回归问题，推动了它的流行。

10.2.1 随机森林算法简介

随机森林的广泛流行源于其用户友好的特性和适应性，使其能够有效地解决分类和回归问题。对于分类问题，最终的类别是通过多数投票决定的。单个决策树产生的类的模式（最经常出现的）成为最终的类。对于回归问题，单个决策树输出的平均值成为模型的最终输出。

随机森林算法最重要的特点之一是它可以处理包含连续变量（continuous variables）的数据集，比如在回归的情况下，也可以处理包含分类变量（categorical variables）的数据集，比如在分类的情况下。它在分类和回归任务中表现更好。

由于随机森林使用决策树作为它的基础模型，因此它继承了决策树的大部分特性。它能够处理连续的和分类的特征，不需要特征缩放和 one-hot 编码。随机森林在不平衡数据上也表现得很好，因为它的层次化本质迫使它们同时处理两个类。最后，随机森林可以捕捉因变量和自变量之间的非线性关系。

由于其可解释性、准确性和灵活性，随机森林是最流行的基于树的分类和回归集成算法之一。该算法的优势在于其处理复杂数据集和减轻过拟合的能力，使其成为机器学习中各种预测任务的宝贵工具。然而，随机森林模型的训练需要大量的计算（这使得它非常适合在多核和分布式环境--如 Hadoop 或 Spark--上并行化）。与逻辑回归或朴素贝叶斯等线性模型相比，它需要更多的内存和计算资源。此外，随机森林往往在文本或基因组数据等高维数据上表现不佳。

10.2.2 随机森林算法原理

随机森林是一种强大的分类和回归算法，可以在不进行太多调优的情况下提供出色的结果。

1. 随机森林算法概述

随机森林算法名字中的“随机”是有原因的，因为树是由随机的一组特征和随机的一组训练示例组成的。每个用不同数据点训练的决策树都试图学习输入和输出之间的关系，最终与其他决策树的预测相结合，其他决策树使用其他数据点集合进行训练，从而得到随机森林。

随机森林算法涉及的步骤如下：

1）步骤 1：在该模型中，选择一个数据点子集和一个特征子集来构建每个决策树。简单地说，就是从有 k 条记录的数据集中随机抽取 n 条记录和 m 个特征。

2）步骤 2：为每个样本构建单独的决策树。

3）步骤 3：每个决策树将生成一个输出。

4）步骤 4：最终输出分别基于分类的多数投票或回归的平均。

例如，将果篮视为如下图所示的数据。现在每次从果篮中取出 n 个样本，并为这 n 个样本构造一个单独的决策树。每个决策树将生成一个输出，如图所示。最终的输出是基于多数投票的。在下面的图中，可以看到多数决策树给出的输出是苹果，而不是香蕉，因此最终的输出是苹果。

泛化以上示例，比如创建一个包含 4 个决策树的随机森林，它可能类似于图 10-。

决策树往往会过拟合。随机森林通过使用 Bagging（装袋）的技术来解决过拟合问题，用随机选择的数据子集训练每棵决策树。每个决策树都使用了数据子集和特征子集来进行训练，这被称为“装袋”技术。Bagging 在不增加偏差的情况下减少了模型的方差，有助于避免过拟合。随机森林还进一步使用特征装袋（feature bagging），即为每个决策树随机选择特征。特征装袋的目标是降低单个树之间的相关性。

在处理分类任务时，每棵树都对预测结果进行投票，获得最大投票的类是随机森林分类器的最终预测结果。在处理回归任务时，每棵树都对预测结果进行输出，单个决策树输出的平均值是随机森林分类器的最终预测结果。单个决策树输出的平均值成为模型的最终输出。

2. 随机森林的假设

为了有效地使用随机森林，理解算法的基本假设是很重要的：

（1）树的独立性：林中的决策树应该彼此独立。这是通过自举采样和特征随机性实现的。

（2）充足的数据：随机森林需要大量的数据来构建不同的树，达到最优的性能。

（3）平衡树：算法假设单个树长得足够深，可以捕获数据中的底层模式。

（4）噪声数据处理：随机森林可以处理噪声数据，但它假设噪声是随机分布的，而不是系统的。

3. 随机森林的重要特征

随机森林的几个关键特征有助于其有效性和多功能性：

（1）多样性：随机森林中的每个决策树都是由不同的数据和特征子集构建的。这种多样性有助于减少过拟合和提高模型的泛化能力。

（2）稳健性：通过对多棵树的结果进行平均，随机森林减少了方差，提高了预测的稳健性。

（3）缺失值的处理：它可以通过使用代理分割或对相同数据点没有缺失值的其他树的结果求平均值来内部处理缺失值。

（4）特征重要性：它提供了对预测过程中每个特征重要性的见解。这对于特征选择和理解底层数据模式尤其有用。

（5）可扩展性：随机森林可以并行化，因为每棵树都是独立于其他树构建的。这使得它可以扩展到大型数据集和高维数据。

（6）通用性：它可以用于分类和回归任务。该算法对于涉及分类变量和连续变量的任务也很有效。

（7）稳定性：由于集成的性质，与单一决策树相比，它对训练数据的变化不太敏感。

（8）袋外误差估计：随机森林提供了一种内部机制来估计模型误差，而不需要单独的验证集。这是使用袋外样本（OOB，out-of-bag）完成的，这些样本不用于构建每棵树。

4. 随机森林的优缺点

随机森林提供的一些优势如下：

（1）特征重要性：一个随机森林可以根据预测能力给出每个被用于训练的特征的重要性。这提供了一个很好的机会来选择相关的特征，并删除较弱的特征。所有特征的重要性的总和总是等于 1。

（2）提高准确性：由于随机森林从单个决策树中收集选票，因此相对于单个决策树，随机森林的预测能力更高。

（3）较少的过拟合：单个分类器的结果得到平均或最大投票，因此减少了过拟合的机会。

如前所述，随机森林是一种使用决策树集合进行分类和回归的集合算法。它是一种训练大量决策树的集成方法，并通过对所有决策树的平均结果来选择最佳结果。这使得算法能够避免过度拟合，并找到一个全局最优。

与决策树相比，随机森林提供更好的结果，减少过拟合，并且通常易于训练和使用，它们非常受欢迎，有时是监督机器学习的首选方法。

随机森林的缺点之一是，与决策树相比，它很难形象化，不像决策树那样容易理解和可视化，而且因为要构建多个单独的分类器，所以会涉及更多计算。

表 10-中总结了决策树与随机森林的区别。

决策树	随机森林
如果允许决策树在没有任何控制的情况下生长，它通常会遭受过拟合的问题。	随机森林从数据子集中创建，最终输出基于平均或多数排序；因此，过拟合的问题得到了解决。
单一决策树的计算速度更快。	它相对较慢。
当一个带有特征的数据集被决策树作为输入时，它会制定一些规则来进行预测。	随机森林随机选择观测值，构建决策树，取结果平均值。它不用任何公式。

因此，只有当随机森林具有多样性和可接受性时，它才比决策树成功得多。

5. 随机森林中的重要超参数

随机森林相对容易调优。在随机森林中使用超参数来提高模型的性能和预测能力，或者使模型更快。

其中提高预测能力的超参数有：

（1）numTrees：要训练的树的数量（至少 1）。如果是 1，则不使用自举。如果大于 1，则完成自举。增加树的数量减少了方差，通常提高了准确性。增加树数量可以增加训练时间。超过某一点上添加更多的树可能无法提高准确性。

（2）maxDepth：树的最大深度（非负）。例如，深度 0 表示 1 个叶节点；深度 1 表示 1 个内部节点+ 2 个叶节点。为其设置一个较高的值可以使模型更具表现力，但是将其设置得过高可能会增加过度拟合的可能性并使模型更加复杂。

（3）maxBins：用于离散连续特征和选择如何在每个节点上分割特征的最大箱数。

（4）impurity：如何分割每棵树中的节点（Entropy/Gini）。

（5）minInstancesPerNode：拆分后每个子节点必须拥有的最小实例数。如果分裂导致左或右子节点的个数小于 minInstancesPerNode，那么分裂将被视为无效而丢弃。必须至少为 1。

（6）minWeightFractionPerNode：每个子节点在分割后必须拥有的加权样本数的最小分数。如果拆分导致左或右子节点中总权重的比例小于 minWeightFractionPerNode，则拆分将被视为无效而丢弃。应该在[0.0,0.5]之间。

（7）thresholds：多类分类中的阈值参数，用于调整预测每一类的概率。

而增加模型速度的超参数有：

（1）featureSubsetStrategy：指定在每个节点中使用的特征的比例。设置这个参数可以提高训练速度。

（2）subsamplingRate：用于学习每个决策树的训练数据的百分比，在（0,1）范围内。设置这个参数可以提高训练速度，并有助于防止过拟合。设置太低可能会导致欠拟合。

推荐执行参数网格搜索以确定这些参数的最佳值。

6. 随机森林的应用

随机森林算法适用于分类和回归任务。以下列举了其中几个常用的情况：

（1）客户流失预测：企业可以使用随机森林来预测哪些客户可能会流失（取消他们的服务），这样他们就可以采取措施留住他们。例如，电信公司可能会使用随机森林模型来识别使用手机频率较低或有延迟付款历史的客户。

（2）欺诈检测：随机森林可以实时识别欺诈交易。例如，银行可能会使用随机森林模型来发现在异常的地点进行的交易或涉及异常的大笔资金。

（3）股价预测：可以预测未来的股票价格。然而，重要的是要注意，股票价格预测是一项非常困难的任务，没有一个模型是完全准确的。

（4）医学诊断：这些可以帮助医生诊断疾病。例如，医生可能会使用随机森林模型来帮助他们诊断癌症患者。

（5）图像识别：可以识别图像中的物体。例如，自动驾驶汽车可能会使用随机森林模型来识别道路上的行人和其他车辆。

10.2.3 Spark 随机森林算法实现

Spark DataFrame API 支持两种主要的树集成算法：随机森林和梯度提升树（Gradient-Boosted Trees，GBTs）。两者都使用 spark.ml 决策树作为他们的基本模型。

随机森林是决策树的集合。随机森林结合了许多决策树，以减少过拟合的风险。而 spark.ml 实现支持随机森林用于二元和多类分类以及回归，使用连续和分类特征。

随机森林分别训练一组决策树，因此训练可以并行进行。Spark 中可以并行训练多棵树，因为每棵树都是在随机森林中独立训练的。该算法将随机性注入到训练过程中，使得每个决策树都略有不同。将来自每个树的预测结合起来，减少了预测的方差，提高了测试数据的性能。

在训练过程中注入的随机性包括：

（1）在每次迭代中对原始数据集进行子采样，以获得不同的训练集（也称为自举）。

（2）在每个树节点上考虑不同的随机特征子集进行分割。

除了这些随机化之外，决策树的训练与单个决策树的训练方式相同。

为了对新实例进行预测，随机森林必须将其决策树集合中的预测集合起来。这种聚合在分类和回归中是不同的。

（1）分类：采用多数投票。每棵树的预测被算作对一个类别的投票。该标签被预测为获得最多选票的类别。

（2）回归：平均。每棵树预测一个真实的值。标签被预测为树预测的平均值。

1. 一些重要参数

Spark.ml 中的随机森林是由类 RandomForestClassifier 和 RandomForestRegressor 实现的。它们有两个参数是最重要的，调优它们通常可以提高性能：

（1）numTrees：是要训练的树的数量。增加树的数量将减少预测的方差，提高模型的测试时间准确性。默认值是 20。注意，训练时间在树的数量上大致呈线性增长。

（2）maxDepth：森林中每棵树的最大深度。增加深度使模型更具表现力和强大。然而，深树需要更长的时间来训练，也更容易过拟合。一般来说，与使用单一决策树相比，使用随机森林训练更深的树是可以接受的。一棵树比随机森林更容易过拟合（因为平均森林中多棵树的方差减少）。

接下来的两个参数通常不需要调优。然而，他们可以调整，以加快训练：

（1）subsamplingRate：该参数指定用于训练森林中每棵树的数据集的大小，作为原始数据集大小的因子。建议使用默认值（1.0），但是减少这个因子可以加快训练速度。

（2）featureSubsetStrategy：用于在每个树节点上进行分割的候选特征的数量。这个数字被指定为特征总数的因子或函数。用它来确定特征装袋（feature bagging）是如何完成的。减少这个数字会加快训练速度，但如果太低有时会影响性能。它的值可以是下面的一个：all（使用所有特征），onethird（随机选择三分之一的特征），sqrt（随机选择 sqrt（特征的数

量）），log2（随机选择 log2（特征的数量）），或者 auto。如果为 auto，则意味着分类用 sqrt，回归用 onethird。默认是 auto。

在大多数情况下，用默认值就好了。如果想要训练大量的树，就需要确保给 Spark 驱动程序（即 driver）分配足够的内存，因为训练的决策树被保持在 driver 的内存中。

2. 模型输入和输出

训练 Spark 随机森林模型，需要指定输入特征和输出预测。我们在表 10-中列出输入列类型，在表 10-中列出输出（预测）列类型。所有输出列都是可选的；若要排除输出列，请将其对应的参数设置为空字符串。

参数名称	类型	默认值	描述
labelCol	Double	"label"	用于预测的标签
featuresCol	Vector	"features"	特征向量

参数名称	类型	默认值	描述
predictionCol	Double	"prediction"	预测的标签
rawPredictionCol	Vector	"rawPrediction"	分类数的长度向量，在进行预测的树节点上具有训练实例标签的计数（仅用于分类）
probabilityCol	Vector	"probability"	分类数的长度向量，等于归一化到多项分布的 rawPrediction（仅用于分类）

3. 训练 Spark 随机森林分类模型

下面的示例以 LibSVM 格式加载数据集，将其分成训练集和测试集，在第一个数据集上进行训练，然后在测试集上进行评估。我们使用两个特征转换器来准备数据；这些有助于索引标签和分类特征的类别，向 DataFrame 添加基于树的算法可以识别的元数据。

4. 训练 Spark 随机森林回归模型

下面的示例以 LibSVM 格式加载数据集，将其分成训练集和测试集，在第一个数据集上进行训练，然后在搁置测试集上进行评估。我们使用特征转换器来索引分类特征，向 DataFrame 中添加基于树的算法可以识别的元数据。

随机森林因其良好的性能和易用性从而成为最受欢迎的算法之一。它们在高维数据集上也表现得同样出色，这在其他算法中是做不到的。

10.2.4 示例：使用 Spark 随机森林分类算法预测婚外情

随机森林是决策树的集合。随机森林是最成功的分类和回归机器学习模型之一。为了减少过拟合的风险，他们结合了许多决策树。与决策树一样，随机森林处理分类特征，扩展到多类分类设置，不需要特征缩放，并且能够捕获非线性和特征交互。

本节我们将通过一个示例，进一步了解如何使用 Spark 随机森林分类算法进行预测。本示例使用使用《今日心理学》和《红皮书》的调查数据来研究婚外情的决定因素。数据经过整理，存储在文件 affairs.csv 中。部分数据如下所示：

其中各字段的含义如下：

【示例】使用随机森林算法，分析和预测婚外恋数据。
请按以下步骤执行。

1. 加载数据文件

执行以上代码，输出内容如下：

2. 探索性数据分析

接下来，对数据进行探索以了解数据特性。代码如下：

执行以上代码，输出内容如下：

3. 特征工程

在这一部分中，我们使用 VectorAssembler 创建一个特征向量，它组合了所有输入特征。代码如下：

执行以上代码，输出内容如下：

4. 分割数据集

将数据集按 75/25 的比例分割为训练集和测试集，并确保我们在训练和测试集中为目标类 affairs 设置了均衡的数量。代码如下：

执行以上代码，输出内容如下：

从分割结果来看，目标 affairs 类的分布在训练集和测试集中都是较为均衡的，值 1 和 0 的比例均大致为 1:2。

5. 构建和训练随机森林模型

现在，可以构建和训练随机森林分类模型了。我们设置标签列为 affairs，设置独立决策树的数量为 50，然后在训练集上学习，返回学习后的模型。代码如下：

执行以上代码，输出内容如下：

6. 模型评估

要对测试集上的预测进行评估，我们需要导入分类评估器，这里使用 spark.ml 包中的 MulticlassClassificationEvaluator 评估器类。我们分别评估不同的指标：accuracy、precision 和 AUC。代码如下：

由此可知，索引为 0 的特征变量是 rate_marriage。

7. 保存和加载模型

有时，在训练模型之后，我们只需要调用预测模型，因此持久化模型对象并将其用于预测非常有意义。下面的代码演示了如何保存和加载模型。

10.2.5 示例：使用 Spark 随机森林回归算法预测房价

在本例中，我们将使用来自 StatLib 存储库的加利福尼亚房价数据集。此数据集包含 20,640 条基于 1990 年加州人口普查数据的记录，每条记录代表一个地理块。下面的列表提供了数据集属性的描述。

表 10-1 消息格式

字段名	含义
Median House Value	房价中位数。 一个街区内家庭的房价中值(以千美元计)
Longitude	经度。 东/西测量，越往西值越高
Latitude	纬度。 北/南测量，数值越高越靠北
Housing Median Age	住房中位数年龄。 一个街区内的房屋年龄中位数，越低越新
Total Rooms	一个街区内的房间总数
Total Bedrooms	一个街区内的卧室总数
Population	居住在一个街区内的总人数
Households	一个街区的住户总数
Median Income	一个街区内的家庭收入中位数(以万美元计)

本示例的目标是使用随机森林回归算法预测房价中位数（Median House Value）。因此 median house value 就是 Label 标签列，也是我们要预测的目标列，而特征列则包括：median age, median income, rooms per house, population per house, bedrooms per room, longitude, latitude。

要构建模型，需要提取对预测最有贡献的特征。为了使某些特征与预测房屋中位数价值（Median House Value）更相关，我们将计算并使用以下这些比率（而不是使用总数）：

(1) 每栋房屋的房间数：rooms per house=total rooms/households。

(2) 每栋房屋的人数：people per house=population/households。

(3) 每栋房屋的卧室数：bedrooms per rooms=total bedrooms/total rooms。

请按以下步骤操作。

1. 加载数据文件

第一步是将数据加载到 DataFrame 中。在下面的代码中，我们指定要加载到数据集中的数据源和模式。

```
```

执行以上代码，输出内容如下：

2. 概要统计

Spark DataFrame 包括一些用于统计处理的内置函数。description()函数对数值列执行汇总统计计算，并将其作为 DataFrame 返回。下面的代码显示了标签和一些特征的一些统计信息。

```
df.describe("medincome","medhvalue","roomsPhouse","popPhouse").show
```

执行以上代码，输出内容如下：

3. 数据集分割

下面的代码使用 DataFrame randomSplit()方法将数据集随机分成两部分，其中 80%用于训练，20%用于测试。

```
val Array(trainingData, testData) = df.randomSplit(Array(0.8, 0.2), 1234)
```

4. 特征提取和管道

下面的代码创建了一个 VectorAssembler（一个 Transformer 转换器），它将在管道中用于将给定的列列表组合为单个特征向量列。

管道将多个 Transformer 转换器和 Estimator 估计器连接在一起，以指定用于训练和使用模型的机器学习工作流。如图 10-所示。

5. 训练模型

Spark ML 支持一种称为 k-fold 交叉验证的技术，可以尝试不同的参数组合，以确定机器学习算法的哪个参数值产生最优模型。使用 k-fold 交叉验证，数据被随机分成 k 个分区。每个分区使用一次作为测试数据集，其余部分用于训练。然后使用训练集生成模型并使用测试集进行评估，从而得到 k 个模型精度（accuracy）测量值。导致最高精度（accuracy）测量值的模型参数产生最优模型。

Spark ML 通过 transformation/estimation 管道支持 k-fold 交叉验证，该管道使用一个称为网格搜索的过程来尝试不同的参数组合，可以在其中设置参数以在交叉验证工作流中进行测试。

下面的代码使用 ParamGridBuilder 来构造用于模型训练的参数网格。我们定义了一个 RegressionEvaluator，它将通过比较测试 medhvalue 列和测试 prediction 列来评估模型。我们使用 CrossValidator 进行模型选择。CrossValidator 使用管道、参数网格和评估器来拟合训练数据集并返回最优模型。CrossValidator 使用 ParamGridBuilder 迭代 RandomForestRegressor 估计器的 maxDepth、maxBins 和 numbTrees 参数，并评估模型，每个参数值重复三次以获得可靠的结果。

```

```

6. 预测和模型评估

接下来，我们使用测试 DataFrame 来衡量模型的准确性，它是原始 DataFrame 的 20%随机分割，不用于训练。

在下面的代码中，我们对管道模型调用 transform()，传入测试 DataFrame，根据管道步骤，通过特征提取阶段，使用模型调优选择的随机森林模型进行估计，然后在新 DataFrame 的一列中返回预测。

```

```

7. 保存模型

现在，我们可以将拟合的管道模型保存到分布式文件存储中，以便以后在生产中使用。这样既保存了特征提取阶段，又保存了通过模型调优选择的随机森林模型。

```
pipelineModel.write.overwrite().save(modeldir)
```

保存管道模型的结果是用于元数据的 JSON 文件和用于模型数据的 Parquet 文件。我们可以用 load 命令重新加载模型；原始模型和重新加载的模型是一样的：

```
val sameModel = CrossValidatorModel.load("modeldir")
```

10.3 梯度提升决策树算法

这一节，我们将学习另一种流行的模型集成方法，称为梯度提升树。

梯度提升决策树和随机森林一样，也是通过组合单个树的输出来执行回归或分类的集成方法。它是一种强大的机器学习技术，用于回归和分类问题，尤其以处理复杂的非线性数据集的有效性而闻名。它结合了决策树的简单性和集成学习的鲁棒性，为预测建模提供了一种强大的方法。

由于在 Kaggle 上的机器学习竞赛中的表现，梯度提升模型变得流行起来。梯度提升决策树（Gradient-Boosted Decision Trees，GBDT）已被证明优于其他模型。这是因为提升涉及实现多个模型并汇总它们的结果。

10.3.1 什么是梯度提升？

Boosting 是机器学习中一种强大的集成技术。与从数据中独立学习的传统模型不同，boosting 将多个弱学习器的预测结合起来，创建一个更准确的强学习器。

弱学习器是一种机器学习模型，它比随机猜测模型略好。例如，假设我们将蘑菇分为可食用和不可食用。如果随机猜测模型的准确率为 40%，那么弱学习者的准确率将略高于它：50% -60%。

Boosting 将数十个或数百个这样的弱学习器组合在一起，构建一个强大的学习器，在同一个问题上具有超过 95%的准确率。

最流行的弱学习器是决策树，选择决策树是因为它几乎可以处理任何数据集。

梯度提升（Gradient Boosting）是一种基于函数空间提升的机器学习技术，其目标是伪残差，而不是传统提升中的残差。它给出了一个弱预测模型集合形式的预测模型，即对数据做出很少假设的模型，通常是简单的决策树。在梯度提升中，使用弱学习器集合来提高机器学习模型的性能。

梯度提升有以下三个关键组件：

（1）决策树作为基础学习者：在梯度增强中，主要的构建块是决策树，它是基于特定条件拆分数据以进行预测的简单模型。与随机森林（另一种集成方法）相比，梯度提升按顺序构建这些树，每棵树都试图纠正前一棵树的错误。

（2）梯度下降算法：术语“梯度提升”来自于使用梯度下降算法来最小化误差。梯度下降是一种通过迭代地向损失函数的最小值移动来找到模型最佳参数的方法（损失函数测量模型的预测值与实际值的距离）。

（3）损失函数优化：模型的目标是减少损失函数，它量化了预测值与实际值之间的差异。梯度增强通过不断改进其预测，有效地最小化了这种损失函数。

梯度提升迭代训练一系列决策树。在每次迭代中，算法使用当前集合预测每个训练实例的标签，然后将预测结果与真实标签进行比较。数据集被重新标记，以更加强调具有较差预测的训练实例。因此，在下一次迭代中，决策树将帮助纠正以前的错误。

重新标记实例的具体机制由损失函数定义。随着每次迭代，GBT 进一步减少了训练数据上的损失函数。梯度提升使用梯度下降方法来最小化操作的损失函数。梯度下降是一阶优化过程，用于定位函数的局部最小值（可微函数）。梯度增强连续训练多个模型，并可用于拟合创新模型，以提供更准确的响应近似。

10.3.2 什么是梯度提升树？

为了更好地理解梯度提升算法是如何工作的，我们首先需要简要解释梯度提升树。

梯度提升树（Gradient-Boosted Trees，GBTs）是一种用于回归和分类问题的机器学习算法。它们是梯度提升法的组成部分，而梯度提升法是一种集成技术。

在梯度提升中，树是一次构建一个。每个新树都是为了解决现有树集合的错误或缺点而创建的。GBT 中使用的树通常是决策树，它是通过以最小化预定义损失函数的方式划分输入空间（特征）来绘制的。

在每棵树被添加到模型中之后，算法计算损失函数，它量化当前集合的预测与实际值的距离。残差，即预测值和实际值之间的差异，然后用于构建下一个树。下一棵树旨在预测这些残差。

梯度提升树和随机森林都结合了许多决策树，以减少每个树面临的过拟合风险。然而，它们在构建单个树的方式和组合结果的方式上有所不同。

随机森林利用装袋法建立独立的决策树，并将它们并行组合。而梯度增强树使用一种称为提升或增强（boosting）的方法。Boosting 将弱学习器（通常是只有一个分裂的决策树，称为决策树桩）按顺序组合在一起，这样每个新树都会纠正前一个树的错误。

GBTs 使用了一种被称为“boosting（提升）”的技术，从“弱学习者”（浅树）中培养出“强学习者”。提升背后的主要概念是，一群弱学习者在结合起来时可以变得强大。GBTs 对决策树集合进行连续训练，每棵后续树减少前一棵树的误差。这是通过使用前一个

模型的残差（residuals）来适应下一个模型来实现的。这个残差校正过程执行一组迭代次数，迭代次数由交叉验证确定，直到残差完全最小化。

10.3.3 梯度提升树算法原理

梯度增强算法适用于具有一组特征(X)和目标(y)的表格数据。与其他机器学习算法一样，其目的是从训练数据中学习足够的知识，以很好地推广到未见过的数据点。

为了理解梯度提升的底层过程，我们将使用一个包含四行的简单销售数据集。使用三个特征-客户年龄、购买类别和购买重量，我们想要预测购买金额：

年龄	类别	购买重量（kg）	购买金额（元）
25	电子设备	2.5	123.45
34	服装	1.3	56.78
42	电子设备	5.0	345.67
19	家居用品	3.2	98.01

1. 梯度增强中的损失函数

在机器学习中，损失函数是一个关键的组成部分，它可以让我们量化模型预测和实际值之间的差异。本质上，它度量模型的执行情况。

以下是它的作用：

（1）计算误差：获取模型的预测输出，并将其与真实值（实际观测值）进行比较。它如何比较，即计算差异，因函数而异。

（2）指导模型训练：模型的目标是最小化损失函数。在整个训练过程中，模型不断更新其内部架构和配置，以使损失尽可能小。

（3）评估度量：通过比较训练、验证和测试数据集上的损失，我们可以评估模型的泛化和避免过拟合的能力。

最常见的两种损失函数是：

1）均方差（Mean Squared Error，MSE）

这个流行的回归损失函数测量预测值和实际值之间的平方差的总和。梯度增强通常使用它的变体：

$$\frac{1}{2}(\text{观测值}-\text{预测值})^2$$

平方值乘以 1/2 的原因与微分有关。当我们对这个损失函数求导时，根据幂法则，1/2 和平方约掉了。因此，最终结果将只是（观察值-预测值），使数学更容易，计算成本更低。

2）交叉熵（cross-entropy）

这个函数测量两个概率分布之间的差。因此，它通常用于目标具有离散类别的分类任务。

因为我们将要做的任务是回归，所以将使用 MSE 损失函数。

2. 步骤一：做一个初步预测

梯度增强是一种逐步提高精度的算法。要开始这个过程，我们需要一个初步的猜测或预测。最初的猜测总是目标的平均值。换句话说，对于第一轮，我们的模型预测所有的购买都是一样的—156 元：

$$\frac{123.45+56.78+345.67+98.01}{4}=156$$

之所以选择平均值，与我们选择的损失函数及其导数有关。每一步，我们都在寻找一个值来找到最小损失函数。换句话说，我们正在寻找一个值使损失函数的导数（梯度）为 0。当我们对每个观测值对预测值求损失函数的导数并求和时，我们就得到了目标值的平均值。因此，我们最初的预测是平均 156 元。

3. 步骤二：计算伪残差

下一步是找出每个观测值和我们的初始预测之间的差异，即观察到的值-156。为了便于说明，我们将把这些差异放在一个新列中：

年龄	类别	购买重量（kg）	购买金额（元）	伪残差
25	电子产品	2.5	123.45	-32.55
34	服装	1.3	56.78	-99.22
42	电子设备	5.0	345.67	189.45
19	家居用品	3.2	98.01	-58.01

记住，在线性回归中，观测值和预测值之间的差称为残差（esiduals）。为了区分线性回归和梯度增强，我们称它们为伪残差（pseudo-residuals）。

4. 步骤三：建立一个弱学习器

接下来，我们将构建一个决策树（弱学习器），它使用我们拥有的三个特征（年龄、类别、购买重量）来预测残差。对于这个问题，我们将决策树限制为只有四个叶子（终端节点），但在实践中，人们通常在 8 到 32 之间选择叶子。

在树与数据拟合之后，我们对数据中的每一行进行预测。下面是如何做第一个：

第一行具有以下特征：电子产品类别（根节点的左侧）和年龄小于 30 岁的客户（子节点的左侧）。这将-32.55 放入叶节点。为了做出最终的预测，我们在第一次预测的基础上加上-32.55 元，这与观测值（123.45 元）完全相同！

我们刚刚做出了一个完美的预测，那么为什么还要建造其他的树呢？现在，我们严重过拟合训练数据。我们希望模型能够泛化。为了缓解这个问题，梯度增强有一个参数叫做学习率（learning rate）。

梯度增强中的学习率只是一个介于 0 和 1 之间的乘数，用于扩展每个弱学习器的预测。当我们在混合计算中添加任意 0.1 的学习率时，我们的预测变成了 152.75，而不是完美的 123.45。

$$\text{预测的购买金额}=156+0.1\times(-32.55)=152.75$$

我们再来预测一下第二行：

我们在树中运行该行，得到 146.08 作为预测值。我们对所有行继续这种方式，直到对四行分别有四个预测：152.75、146.08、174.945、150.199。现在将它们作为一个新列添加到表中，如表 10-所示。

年龄	类别	购买重量（kg）	购买金额（元）	伪残差	新预测值
25	电子产品	2.5	123.45	-32.55	152.75
34	服装	1.3	56.78	-99.22	146.08
42	电子设备	5.0	345.67	189.45	174.945
19	家居用品	3.2	98.01	-58.01	150.199

接下来，我们通过从购买金额中减去新的预测金额来找到新的伪残差。将新的伪残差作为一个新列添加到表中，如表 10-所示。

年龄	类别	购买重量（kg）	购买金额（元）	新伪残差
25	电子产品	2.5	123.45	-29.295
34	服装	1.3	56.78	-89.298
42	电子设备	5.0	345.67	170.725
19	家居用品	3.2	98.01	-52.189

如表中所见，新的伪残差更小了，这意味着我们的损失在减小。

5. 步骤四：迭代

在接下来的步骤中，我们迭代步骤三，即构建更多的弱学习器。唯一要记住的是，我们必须不断地将每棵树的残差加到初始预测中，以生成下一个预测。

例如，如果我们构建 10 棵树，每棵树的残差记为 r_i (1 <= i <= 10)，则下一个预测将变为 p_10 = 156 + eta * (r_1 + r_2 +⋯)+ r_10)，其中 p_10 表示第十轮的预测，eta 代表学习率。

在实践中，专业人士通常从 100 棵树开始，而不仅仅是 10 棵。在这种情况下，该算法据说要训练 100 个增强回合。

如果不知道自己特定问题所需的树的确切数量，那么可以使用一种简单的技术，称为提前停止（early stopping）。

在提前停止时，我们选择大量的树，如 1000 或 10000。然后，我们监控损失，而不是等待算法完成所有树的构建。如果在一定数量的增强回合（例如 50 轮）中损失没有改善，我们就会提前停止训练，从而节省时间和计算资源。

10.3.4 配置梯度增强模型

在机器学习中，选择模型的设置被称为“超参数调优”。这些设置被称为“超参数”，是机器学习工程师必须自己选择的选项。与其他参数不同，模型不能简单地通过数据训练来学习超参数的最佳值。

梯度增强模型有许多超参数，我将在下面概述其中的一些。

1）目标（objective）

该参数设置算法的方向和损失函数。如果目标是回归，则选择 MSE 作为损失函数，而对于分类，则选择交叉熵。

2）学习率（learning rate）

梯度增强最重要的超参数可能是学习率。它通过调整收缩因子来控制每个弱学习器的贡献。较小的值（趋向于 0）减少了每个弱学习器在集合中的发言权。这将要求建造更多的树，因此需要更多的时间来完成训练。但是，最终的强学习者确实是强大的，并且不受过拟合的影响。

3）树的数量（the number of trees）

这个参数，也称为增强轮数或 n_estimators，控制要构建的树的数量。构建的树越多，整个系统就会变得越强大、性能越好。随着更多的树允许模型捕获数据中的更多模式，它也变得更加复杂。然而，更多的树显著提高了过拟合的可能性。为了减轻这种情况，采用早期停止和低学习率的组合。

4）最大深度（max depth）

该参数控制每个弱学习器（决策树）中的层数。最大深度为 3 意味着树中有三层，包括叶子层。树越深，模型就越复杂，计算成本越高。选择接近 3 的值以防止过拟合。最大深度应该是 10。

5）每片叶子的最小样本数（minimum number of samples per leaf）

该参数控制决策树中的分支如何分割。将终端节点（叶子）的样本个数设置得较低，使得整个算法对噪声比较敏感。

6）子抽样率（subsampling rate）

该参数控制用于训练每棵树的数据的比例。在上面的示例中，我们使用 100%的行，因为我们的数据集中只有 4 行。但是，现实世界的数据集通常有更多，需要抽样。因此，如果将子抽样率设置为低于 1 的值，例如 0.7，则每个弱学习器都会在随机抽样的 70%的行上进行训练。较小的子样本率可以导致更快的训练，但也可能导致过拟合。

7）特征抽样率（feature sampling rate）

该参数与子抽样完全相似，但它对特征进行抽样。对于具有数百个特征的数据集，建议选择 0.5 到 1 之间的特征抽样率，以降低过拟合的可能性。

梯度增强是我们对于表格监督学习任务最有能力的模型。所以，大多数时候，你不必担心它不够好。当我们使用梯度增强时，我们的时间几乎总是花在如何正则化它上，这样它就不会在遇到看不见的数据时变得无能为力。

10.3.5 Spark 梯度提升树实现

Gradient Boosting 是一种强大的机器学习技术，它结合了多个弱学习器（通常是决策树）来创建一个强预测器。该算法首先对数据拟合一个简单的模型，例如具有一层或两层的决策树。然后使用该模型的残差来训练第二个模型，该模型被添加到集成中。这个过程重复很多次，每个新模型都是在前一个模型的残差上训练的。最后的预测器是集合中所有模型的总和。

在 spark.ml 中的实现支持将 GBT 用于二分类和回归，使用连续和分类特征。Spark.ml 中的梯度提升树是由类 GBTClassifier（用于分类任务）和 GBTRegressor（用于回归任务）实现的。

1. 一些重要参数

Spark 中的梯度提升模型有三个参数是最重要的，调优它们通常可以提高性能：

（1）maxBins：最大箱数。用于离散连续特征，选择如何在每个节点上分割特征的最大箱数。更多的箱子提供更高的粒度。在任何分类特征中必须至少有 2 个类别。（默认值= 32）

（2）maxDepth：树的最大深度（非负的）。例如，深度 0 表示 1 个叶节点；深度 1 表示 1 个内部节点+ 2 个叶节点。（默认值= 5）。

（3）maxIter：最大迭代数的参数（>= 0）。

接下来的两个参数通常不需要调优。然而，他们可以调整，以加快训练：

（1）subsamplingRate：用于学习每个决策树的训练数据的百分比，在(0,1]范围内。默认= 1.0，但是减少这个因子可以加快训练速度。

（2）featureSubsetStrategy：用于在每个树节点上进行分割的候选特征的数量。这个数字被指定为特征总数的因子或函数。减少这个数字会加快训练速度，但如果太低有时会影响性能。支持的选项有：all（使用所有特征），onethird（随机选择三分之一的特征），sqrt（随机选择 sqrt（特征的数量）），log2（随机选择 log2（特征的数量）），n 或者 auto。如果为 n，则当 n 的值在(0,1.0]范围内时，使用 n*特征数；当 n 的值在(1,特征数）范围内时，使用 n 个特征。如果为 auto，则意味着为任务自动选择：如果 numTrees==1，则设为 all；如果 numTrees>1，则分类用 sqrt，回归用 onethird。默认是 auto。

另外还有些参数也需要大致了解：

（1）lossType：GBT 试图最小化的损失函数。（不区分大小写）支持：logistic（默认）。

（2）impurity：用于信息增益计算的标准（不区分大小写）。支持：variance（默认）。

在大多数情况下，用默认值就好了。

2. 模型输入与输出

训练 Spark 梯度提升模型，需要指定输入特征和输出预测。我们在表 10-中列出输入列类型，在表 10-中列出输出（预测）列类型。所有输出列都是可选的；若要排除输出列，请将其对应的参数设置为空字符串。

参数名称	类型	默认值	描述
labelCol	Double	"label"	用于预测的标签
featuresCol	Vector	"features"	特征向量

请注意，GBTClassifier 当前只支持二分类。

参数名称	类型	默认值	描述
predictionCol	Double	"prediction"	预测的标签

将来，GBTClassifier 还将输出 rawPrediction 和 probability 的列，就像 RandomForestClassifier 一样。

注意： GTBs 还不支持多类分类。对于多类问题，请使用决策树或随机森林。。

3. 训练 Spark 梯度提升树分类模型

下面的示例以 LibSVM 格式加载数据集，将其分成训练集和测试集，在训练数据集上进行训练，然后在测试数据集上进行评估。我们使用两个特征转换器来准备数据；这些有助于索引标签和分类特征的类别，向 DataFrame 添加基于树的算法可以识别的元数据。

4. 训练 Spark 梯度提升树回归模型

下面的 GBT 回归示例以 LibSVM 格式加载数据集，将其分成训练集和测试集，在第一个数据集上进行训练，然后在搁置测试集上进行评估。我们使用特征转换器来索引分类特征，向 DataFrame 中添加基于树的算法可以识别的元数据。

```
```

10.3.6 示例：CTR 广告点击预测

在网络广告中，点击率（click-through rate，CTR）是评价广告效果的一个非常重要的指标。点击率（Click Through Rate，CTR）预估用来判断一条广告被用户点击的概率，对每次广告的点击做出预测，把用户最有可能点击的广告找出来，是广告技术最重要的算法之一。计算点击率 CTR 的公式如下：

$$CTR = \frac{\text{广告点击次数}}{\text{广告总印象数}} \times 100(\%)$$

注意：

广告印象（Advertising Impression）是指广告信息接触受众成员的一次机会。总印象数（或称接触人次）指媒介计划中整个媒介投放的亮相总次数，或指一个媒介排期计划所接触的总人次。

点击率衡量的是一个比值，有多少人看到一个广告链接，然后又有多少人点进去看，用这两个数字之比来衡量一个网络广告的受欢迎程度和影响程度。此外，CTR 不重复计算 24 小时之内相同 IP 的点击行为，因为相同的 IP 意味着相同的人，相同的人看一万次和看一次没有什么区别。目前，点击率一般都小于 1%。

因此，点击预测系统是必不可少的，广泛用于赞助搜索和实时竞价。

在本节中，我们将探讨如何使用 Spark MLlib 构建和评估 GBT 分类模型来执行 CTR 广告点击预测。我们的目标是使用 GBT 模型预测一个移动广告是否会被点击。我们将采用如下方法来实现：

（1）使用 OneHotEncoder 与 StringIndexer 提取分类特征的特征。

（2）用梯度提升树（GBT）和 CrossValidator 训练一个分类模型。

（3）CrossValidator 的评估器 Evaluator 是 BinaryClassificationEvaluator 的默认值。

1. 理解数据集和目标

```
本案例使用
```

我们的任务是训练一个 GBT 模型，预测移动广告是否会被点击，最终给出测试集中每个广告效果 id 所对应的 CTR 预估值（预测点击概率），格式如下：

```
id, 点击
60000000,0.384
63895816,0.5919
759281658,0.1934
895936184,0.9572
...
```

2. 构建 ETL 过程

首先，导入所有的依赖包和类。代码如下：

```

```

3. 探索广告日志

在训练集上创建一个名为 impression 的 Spark SQL 临时视图，代码如下：

4. 特征工程管道

一旦我们熟悉了要处理的数据，我们就可以进入机器学习阶段，在该阶段把数据转换成特征输入机器学习算法，并产生一个训练过的模型，我们可以使用该模型进行预测。由于 Spark MLlib 算法以一列双精度特征向量作为输入，因此一个典型的特征工程工作流包括：

（1）识别数字和分类特征。

（2）字符串索引。

（3）将它们组合成一个稀疏向量。

在特征提取阶段，我们将训练数据与测试数据合并。因为，测试数据包含了训练数据中不存在的值。因此，在提取测试数据的特征时，我们需要避免每个变量缺失值的错误。

构建特征工程工作流的实现代码如下：

```

```

执行以上代码，这样我们就得到了一个学习过的特征转换管道模型，它是一个 Estimator。我们使用它来对训练集和测试集进行特征转换处理，代码如下：

```

```

5. 训练 GBT 分类模型

我们应用交叉验证法和参数网格，学习（训练）出一个最优模型。使用 CrossValidator 训练一个 GBT 模型，使用 BinaryClassificationEvaluator 作为模型的评估器。代码如下：

```

```

请注意，这个学习和训练的过程会比较耗时，因此请适当修改代码中的网格参数以完成训练过程。

6. 持久保存模型

对于上一步训练出来的模型，为了便于后续的使用，我们可以将其持久存储到 HDFS 中，在需要的时候加载即可。代码如下：

```
// 将模型存储到指定位置
cvModel.write.save("/data/ctr/cvmodel")
```

7. 预测点击

有了训练出的最优模型，我们就可以将它应用于点击预测。首先，从 HDFS 加载存储的模型，代码如下：

```

```

10.3.7 示例：波士顿房价预测

在本小节中，我们将探讨如何使用 Spark MLlib 构建和评估 Gradient Boosting 回归模型，包括超参数调优和变量选择。

```
...
```

我们的目标是构建并学习一个 GBT 回归模型，用于预测波士顿住房价格。

1. 加载数据集

首先，导入所需的库。代码如下：

```
import org.apache.spark.ml.regression.GBTRegressor
```

执行以上代码，输出内容如下：

2. 数据准备

在构建模型之前，我们需要使用 VectorAssembler 类将输入特征组装成单个特征向量。然后，将数据集分成训练集（70%）和测试集（30%）。代码如下：

```

```

执行以上代码，输出内容如下：

将数据分成训练集和测试集，代码如下：

```

```

3. 训练 GBT 模型

现在，我们可以使用 GBTRegressor 类创建一个梯度增强模型。代码如下：

```

```

4. 模型评估

利用该模型对测试数据进行预测。代码如下：

```

```

5. 超参数调优和模型选择

超参数调优是为机器学习模型的参数选择最佳值的过程。

在梯度增强中，主要的超参数是树的数量、学习率和每棵树的最大深度。我们可以使用交叉验证和网格搜索来找到最佳的超参数。实现代码如下：

```

```

在这个例子中，我们使用了一个 ParamGridBuilder 来定义一个要搜索的超参数网格。

然后，我们使用 CrossValidator 执行 5 折交叉验证并选择最佳超参数。numFolds 参数指定用于交叉验证的折叠数。

6. 分析特征的重要性

变量选择是为机器学习模型选择最重要特征的过程。在梯度增强中，我们可以使用特征重要性分数来确定哪些特征是最重要的。

```
```

以上代码中的 featureImportances 方法返回一个特征重要性得分向量。然后我们可以将这些分数映射回特征名称，以确定哪些特征是最重要的。

7. 保存并加载模型（可选）

如果您望将来重用该模型，则可以将其保存到磁盘上，并在需要时重新加载它。存储和加载模型的代码如下：

```
```

10.3.8 示例：共享单车租赁数量预测

需求预测是跨业务的一个常见问题，良好的预测使企业或服务能够优化库存，匹配供需，使客户满意并最大化盈利能力。本示例使用自行车共享数据集，应用 Spark 机器学习管道和 GBT 学习算法，来预测每小时的自行车租赁数量。

1. 数据描述

自行车共享系统是新一代传统的自行车租赁，从入会、租赁到归还的整个过程都是自动化的。通过这些系统，用户可以很容易地从一个特定的位置租一辆自行车，并返还到另一个位置。目前，全球约有 500 多个共享单车项目，由 50 多万辆自行车组成。今天，由于这些系统在交通、环境和健康问题上的重要作用，人们对它们产生了极大的兴趣。

除了自行车共享系统在现实世界中的有趣应用外，这些系统生成的数据的特征也使它们对研究具有吸引力。与公共汽车或地铁等其他交通服务不同，这些系统明确记录了旅行时间、出发和到达位置。这一功能将共享单车系统变成了一个虚拟传感器网络，可以用于感知城市的交通。因此，预计城市中的大多数重要事件都可以通过监测这些数据来检测。

本示例使用的自行车共享数据集包含 2011 年至 2012 年某城市自行车共享系统中每小时和每天的租赁自行车数量，以及相应的天气和季节信息。

数据集中包括以下列：

.....

部分数据如下：

```
...
```

该数据集已经为机器学习算法做好了充分的准备。数值输入列（temp、atemp、hum 和 windspeed）已经被归一化，分类值（season、yr、mnth、hr、holiday、weekday、workingday、weathersit）已经被转换为索引，除了日期（dteday）之外的所有列都是数字。

2. 数据加载和预处理

首先，加载数据到 Spark DataFrame 中。代码如下：

3. 训练机器学习管道

现在我们已经查看了数据并将其准备为带有数值的 DataFrame，接下来我们需要训练一个模型来预测未来的自行车共享租赁。

大多数机器学习算法需要单个输入列，其中包含一个特征向量和单个目标列。DataFrame 目前对每个特性都有一列，因此我们需要将这些特征列组合到一个特征向量列中。而 Spark 机器学习管道将多个步骤组合到单个工作流中，使开发模型时更容易进行迭代。在这个例子中，我们使用以下对象创建一个机器学习管道：

（1）VectorAssembler：将特征列组装成特征向量。

（2）VectorIndexer：标识应被视为分类的列。这是启发式地完成的，将具有少量不同值的任何列标识为分类列。在本例中，以下列被认为是分类的：yr(2 个值)，season(4 个值)，holiday(2 个值)，workingday (2 个值)，以及 weathersit (4 个值)。

（3）GBTRegressor：使用 GBT 算法来学习如何从特征向量中预测租金计数。

（4）CrossValidator：GBT 算法有几个超参数。本示例演示了如何在 Spark 中使用超参数调优。该对象应用交叉验证法自动测试超参数网格并选择最佳结果模型。

4. 做出预测并评估结果

最后一步是使用拟合模型对测试数据集进行预测，并评估模型的性能。计算评估度量对于理解预测的质量，以及比较模型和调优参数非常重要。模型在测试数据集上的性能提供了它在新数据上可能表现的近似值。例如，如果我们有下周的天气预报，就可以预测下周的自行车租金。

执行以上代码，输出内容如下：

5. 模型改进

对于本示例，我们还可以从几个方面进行改进。以下是改进这一模式的一些建议：

（1）租金是 registered 租金和 casual 租金的总和。这两种情况可能有不同的行为，因为经常骑自行车的人和偶尔骑自行车的人可能出于不同的原因租用自行车。尝试分别为 registered 训练一个 GBT 模型和为 casual 训练一个模型，然后将它们的预测加在一起以获得完整的预测。

（2）为了提高效率，本例中只使用了几个超参数设置。可以通过测试更多设置来改进模型。一个好的开始是通过设置 maxIter=200 来增加树的数量；这需要更长的训练时间，但可能更准确。

（3）本示例使用的是数据集的特征，但是我们可以通过一些特征工程来提高性能。例如，与工作日相比，天气可能对周末和节假日的租赁数量有更大的影响。可以尝试通过组合这两列来创建一个新特征。

10.4 XGBoost 算法

XGBoost（eXtreme Gradient Boosting）由华盛顿大学的陈天奇开发的一个开源机器学习项目，高效地实现了 GBDT 算法并进行了算法和工程上的许多改进，被广泛应用在 Kaggle 竞赛及其他许多机器学习竞赛中并取得了不错的成绩。

XGBoost 是目前最快最好的开源提升树工具包，比常见的工具包快 10 倍以上。XGBoost 已经成为主流的分类和回归机器学习算法。在数据科学方面，有大量 Kaggle 选手选用它进行数据挖掘比赛，其中包括两个以上 Kaggle 比赛的夺冠方案。在工业界规模方面，XGBoost 的分布式版本有广泛的可移植性，支持在 YARN、MPI、Sungrid Engine 等各个平台上面运行，并且保留了单机并行版本的各种优化，使得它可以很好地解决于工业界规模的问题。

10.4.1 XGBoost 算法简介

XGBoost 本质上还是一个 GBDT，使用了梯度提升决策树（这是一种利用梯度下降的监督学习提升算法）。它以其速度、效率和可很好地扩展大型数据集的能力而闻名，在处理结构化数据方面表现优异。可以理解为 XGBoost 是相同的梯度提升的高级实现；也就是说，它通过将残差相加，将弱学习器树组合成强学习器。为了提高效率和可伸缩性，它的并行树提升功能使它比其他基于树的集成算法要快得多。

1. XGBoost 的核心思想

XGBoost 的核心算法思想是：

（1）不断地添加树，并不断地进行特征分裂来生长一棵树，每次添加一个树，其实是学习一个新函数 f(x)，去拟合上次预测的残差，逐次迭代来提高模型性能。

（2）当训练完成得到 k 棵树，我们要预测一个样本的分数，其实就是根据这个样本的特征，在每棵树中会落到对应的一个叶子节点，每个叶子节点就对应一个分数。

（3）最后只需要将每棵树对应的分数加起来就是该样本的预测值。

举个例子，假设我们要预测一家人对电子游戏的喜好程度，考虑到年轻和年老相比，年轻更可能喜欢电子游戏，以及男性和女性相比，男性更喜欢电子游戏，故先根据年龄大小区分小孩和大人，然后再通过性别区分开是男是女，逐一给各人在电子游戏喜好程度上打分，如图 10-所示。

然后，再生成第二颗树，按日常是否使用电脑来区分。就这样，训练出了 2 棵树 tree1 和 tree2，如图 10-所示：

类似之前 GBDT 的原理，两棵树的结果累加起来便是最终的预测值，所以小男孩的预测分数就是两棵树中小男孩所落到的叶子节点的分数相加：2 + 0.9 = 2.9。爷爷的预测分数同理：-1 +（-0.9）= -1.9。

在上面的例子中，我们用两棵树来进行预测。我们对于每个样本的预测结果就是每棵树预测分数的和。上面的图例只是举了两个分类器，其实还可以有更多更复杂的弱分类器，一起组合成一个强分类器。

构建最优模型的一般方法是最小化训练数据的损失函数，XGBoost 采用结构风险最小化损失函数。该损失函数由两部分构成，第一部分用来衡量预测分数和真实分数的差距，另一部分则是正则化项。正则化项同样包含两部分，即叶子结点的个数和叶子节点的分数，生成树时考虑了树的复杂度。

另外，XGBoost 在选取最佳切分点时可以开启多线程进行，大大提高了运行速度。

2. XGBoost 与 GDBT 的区别

XGBoost 算法与 GDBT 算法的区别，可归纳为以下几点：

（1）XGBoost 生成 CART 树考虑了树的复杂度，GDBT 未考虑，GDBT 在树的剪枝步骤中考虑了树的复杂度。

（2）XGBoost 是拟合上一轮损失函数的二阶导展开，GDBT 是拟合上一轮损失函数的一阶导展开，因此，XGBoost 的准确性更高，且满足相同的训练效果，需要的迭代次数更少。

（3）XGBoost 与 GDBT 都是逐次迭代来提高模型性能，但是 XGBoost 在选取最佳切分点时可以开启多线程进行，大大提高了运行速度。

3. XGBoost 的优点

相比其他算法，XGBoost 能够处理大量特征和样本，并且支持通过正则化控制模型的复杂度。XGBoost 也可以自动进行特征选择并对缺失值进行处理。

1）正则化

XGBoost 在损失函数里加入了正则项，用于控制模型的复杂度。正则项里包含了树的叶子节点个数、每个叶子节点上输出的分数的 L2 模的平方和。正则项降低了模型的方差，使学习出来的模型更加简单，防止过拟合，这也是 XGBoost 优于传统 GBDT 的一个特性。

2）并行处理

XGBoost 工具支持并行。注意 XGBoost 的并行不是树粒度的并行，XGBoost 也是一次迭代完才能进行下一次迭代的（第 t 次迭代的损失函数里包含了前面 t-1 次迭代的预测值）。XGBoost 的并行是在特征粒度上的。

我们知道，决策树的学习最耗时的一个步骤就是对特征的值进行排序（因为要确定最佳分割点），XGBoost 在训练之前，预先对数据进行了排序，然后保存为块结构，后面的迭代中重复地使用这个结构，大大减小计算量。这个块结构也使得并行成为了可能，在进行节点的分裂时，需要计算每个特征的增益，最终选增益最大的那个特征去做分裂，那么各个特征的增益计算就可以开多线程进行。

3）灵活性

XGBoost 支持用户自定义目标函数和评估函数，只要目标函数二阶可导就行。

4）缺失值处理

对于特征的值有缺失的样本，XGBoost 可以自动学习出它的分裂方向。

5）剪枝

XGBoost 先从顶到底建立所有可以建立的子树，再从底到顶反向进行剪枝，这样不容易陷入局部最优解。

6）内置交叉验证

XGBoost 允许在每一轮 boosting 迭代中使用交叉验证。因此，可以方便地获得最优 boosting 迭代次数。

4. XGBoost 算法参数

XGBoost 算法涉及到的参数比较多，下面按分类进行简单介绍 。

1）常规参数

（1）booster [default=gbtree]：选择基分类器。gbtree：基于树的模型，gblinear：线性模型。

（2）silent [default=0]：设置成 1 则没有运行信息输出，最好是设置为 0.

（3）nthread [default to maximum number of threads available if not set]：线程数

2）模型参数

（1）eta [default=0.3]：学习率。这是一个收缩参数，用于更新叶子节点权重时，乘以该系数，避免步长过大。参数值越大，越可能无法收敛。把学习率 eta 设置的小一些，小学习率可以使得后面的学习更加仔细。

（2）min_child_weight [default=1]：这个参数默认是 1，是每个叶子里面 h 的和至少是多少，对正负样本不均衡时的 0-1 分类而言，假设 h 在 0.01 附近，min_child_weight 为 1 意味着叶子节点中最少需要包含 100 个样本。这个参数非常影响结果，控制叶子节点中二阶导的和的最小值，该参数值越小，越容易过拟合。

（3）max_depth [default=6]：每颗树的最大深度，树高越深，越容易过拟合。

（4）max_leaf_nodes：最大叶结点数。

（5）gamma [default=0]：后剪枝时，用于控制是否后剪枝的参数。

（6）max_delta_step [default=0]：这个参数在更新步骤中起作用，如果取 0 表示没有约束，如果取正值则使得更新步骤更加保守。可以防止做太大的更新步子，使更新更加平缓。

（7）subsample [default=1]：样本随机采样，较低的值使得算法更加保守，防止过拟合，但是太小的值也会造成欠拟合。

（8）colsample_bytree [default=1]：列采样，对每棵树的生成用的特征进行列采样。一般设置为： 0.5-1。

（9）lambda [default=1]：控制模型复杂度的权重值的 L2 正则化项参数，参数越大，模型越不容易过拟合。

（10）alpha [default=0]：控制模型复杂程度的权重值的 L1 正则项参数，参数值越大，模型越不容易过拟合。

（11）scale_pos_weight [default=1]：如果取值大于 0 的话，在类别样本不平衡的情况下有助于快速收敛。

3）学习任务参数

（1）objective [default=reg:linear]：定义最小化损失函数类型，常用参数有：

- ✓ binary:logistic - 逻辑回归二元分类，返回预测概率（不是类别）。

- ✓ multi:softmax - 多类分类使用 softmax 损失函数，返回预测类（不是概率）。还需要设置一个额外的 num_class（类别的数量）参数来定义惟一类的数量。
- ✓ multi:softprob - 与 softmax 相同，但返回属于每个类的每个数据点的预测概率。

（2）eval_metric [默认根据 objective 而定]：用于验证数据的度量。用于回归的默认值是 rmse，用于分类的默认值是 error。典型的值有：

- ✓ rmse - root mean square error，均方根误差。
- ✓ mae - mean absolute error，平均绝对误差。
- ✓ logloss - 负对数似然。
- ✓ error - 二值分类错误率 (0.5 阈值)。
- ✓ merror - 多类分类错误率。
- ✓ mlogloss - 多类 logloss。
- ✓ auc - 曲线下的面积。

（3）seed [default=0]： 随机种子，可用于生成可重复的结果，也可用于参数调优。

10.4.2 XGBoost4J-Spark 项目

XGBoost4J-Spark 是一个旨在通过将 XGBoost 适配到 Apache Spark 的 MLLIB 框架来无缝集成 XGBoost 和 Apache Spark 的项目。通过集成，用户不仅可以使用 XGBoost 的高性能算法实现，还可以利用 Spark 强大的数据处理引擎

XGBoost4J-Spark 项目于 2016 年底启动，将 XGBoost 移植到了 Spark。XGBoost4J-Spark 利用了 Spark 的高度可伸缩的分布式处理引擎，并与 Spark MLlib 的 DataFrame/Dataset 抽象完全兼容。Xgboost4j-spark 继承了 Spark 和 XGBoost 两个开源项目的优点，可以在 Spark 上高效地进行大规模的模型训练。

XGBoost4J-Spark 还可以无缝地嵌入到 Spar 机器学习管道中，并与 Spark 机器学习库中的 transformers 和 estimators 相集成。

注意： XGBoost4J-Spark 需要 Apache Spark 3.0+。最新版本的 XGBoost4J-Spark 使用了 org.apache.spark.ml.param.shared 的功能，以提供与 Spark MLLIB 框架的紧密集成，而这些功能在早期版本的 Spark 中并不完全可用。另外，为了在 Spark 集群上利用分布式训练，XGBoost4J-Spark 包可以在 Scala 管道中使用，但在 Python 管道中存在问题。

10.4.3 用 XGBoost4J-Spark 构建一个 ML 应用程序

在使用 xgboost4j-spark 进行模型训练时，首先需要准备训练数据和测试数据。训练数据用于训练模型，测试数据用于评估模型的性能。然后，需要进行特征工程，将原始数据转换为适合 xgboost4j-spark 处理的格式。接下来，根据数据集的大小和集群的配置，选择合适的参数进行模型训练。在训练过程中，xgboost4j-spark 会根据数据的分布情况将数据分配到不同的节点上进行并行处理，以充分利用集群的计算能力。

1. 引用 XGBoost4J-Spark 依赖

.....

2. 数据准备

在本节中，我们将以 Iris 数据集为例，展示如何使用 Spark 对原始数据集进行转换，使其适合 XGBoost 的数据接口。

3. 处理缺失值

XGBoost 默认支持缺失值。如果给定一个 SparseVector，XGBoost 将把 SparseVector 中不存在的任何值视为缺失。还可以指定 XGBoost 来处理数据集中的特定值，就好像它是一个缺失的值一样。默认情况下，XGBoost 将把 NaN 视为表示缺失的值。

4. 模型训练

XGBoost 支持回归、分类和排名。虽然我们在本示例中使用 Iris 数据集来展示如何使用 XGBoost4J-Spark 来解决多类分类问题，但在回归和排名中的使用与分类非常相似。

5. 预测

XGBoost4j-Spark 支持两种模型服务方式：批量预测和单实例预测。

1）批量预测

当我们得到一个模型时，无论是 XGBoostClassificationModel 还是 XGBoostRegressionModel，它都接受一个 DataFrame，读取包含特征向量的列，预测每个特征向量，并输出一个默认包含以下列的新 DataFrame：

2）单实例预测

XGBoostClassificationModel 或 XGBoostRegressionModel 也支持对单个实例进行预测。它接受单个 Vector 向量作为特征，并输出预测标签。示例代码如下：

6. 模型持久化

数据科学家生成 ML 模型，并将其交给工程团队在生产环境中进行部署。挨过来，经过训练的模型也可能被数据科学家使用，例如作为数据探索过程中的基线。因此，支持模型持久性以使模型在使用场景和编程语言之间可用是非常重要的。

10.4.4 用 XGBoost4J-Spark 构建一个 ML 管道

Spark ML 管道可以将多个算法或函数组合到一个管道中。它涵盖了从特征提取、转换、选择到模型训练和预测。XGBoost4j-Spark 可以将 XGBoost 无缝嵌入到这样的机器学习管道中。下面的示例展示了如何构建这样一个由 Spark MLlib 特征转换器和 XGBoostClassifier 估计器组成的管道。

我们仍然使用 Iris 数据集和 rawInput DataFrame。首先，我们需要将数据集分成训练数据集和测试数据集。代码如下：

```
val Array(training, test) = rawInput.randomSplit(Array(0.8, 0.2), 123)
```

我们构建的机器学习管道包括 4 个阶段：

（1）将所有特征组装成单个向量列。

（2）从字符串标签到索引 double 标签。

（3）使用 XGBoostClassifier 训练分类模型。

（4）将索引 double 标签转换回原始字符串标签。

```

```

执行以上代码，输出内容如下：

10.4.5 在生产环境中运行 XGBoost4J-Spark

XGBoost4J-Spark 是将 XGBoost 更轻松地引入生产环境的最重要步骤之一。在生产环境中运行 XGBoost4J-Spark 有三个关键特性。

1. 并行/分布式训练

训练数据集的庞大规模是生产环境中最重要的特征之一。为了确保 XGBoost 中的训练能够随着数据规模的扩大而扩展，XGBoost4J-Spark 将 Spark 的分布式/并行处理框架与 XGBoost 的并行/分布式训练机制连接在一起。

10.5 LightGBM 算法

XGBoost 算法虽然利用了预排序和近似算法可以降低寻找最优分裂点的计算量，但在节点分裂过程中仍需要遍历整个数据集。并且预排序过程的空间复杂度过高，不仅需要存储特征值，还需要存储特征对应样本梯度统计值的索引，相当于消耗了两倍的内存。所以在内存和计算方面还是有很大的优化空间的。那么 XGBoost 还可以在哪些角度进行优化呢？这就是 LightGBM。

10.5.1 LightGBM 算法简介

LightGBM 于 2016 年 10 月 17 日发布，是微软分布式机器学习工具包（DMTK）项目的一部分。它被设计为是快速的和分布式的，从而提高了训练速度和降低了内存使用。它支持 GPU 和并行学习以及处理大型数据集的能力。

LightGBM 是一个开源、分布式、高性能梯度增强框架，类似于 XGBoost。这个框架专门为排序、分类和许多其他机器学习任务创建高质量和支持 GPU 的决策树算法。为了解决 XGBoost 的缺点问题，LightGBM 提出了以下几点解决方案：

（1）单边梯度抽样算法；

（2）直方图算法；

（3）互斥特征捆绑算法；

（4）基于最大深度的 Leaf-wise 的垂直生长算法；

（5）类别特征最优分割；

（6）特征并行和数据并行；

（7）缓存优化。

在公共数据集上的几个基准测试和实验表明，LightGBM 甚至比 XGBoost 更快、更准确。LightGBM 已经成为了 XGBoost 新的挑战者。

与 XGBoost 相比，LightGBM 有几个优点。

（1）内存更小。

（2）速度更快。在某些任务中，LightGBM 比 XGBoost 快 11 到 15 倍。

（3）LightGBM 通常在精度方面优于 XGBoost，因为它是按树叶方向生长的。训练决策树有两种主要的策略，水平方向的（level-wise growth）和叶子方向的（leaf-wise growth）。对于大多数基于树的集成系统（包括 XGBoost）来说，水平树生长是传统的决策树生长方式。LightGBM 引入了叶向增长策略。与水平生长相比，叶向生长收敛更快，损失更小。这两种策略如图 10-所示：

10.5.2 SynapseML 中的 LightGBM

SynapseML（以前称为 MMLSpark）是一个开源库，它简化了大规模可扩展机器学习（ML）管道的创建。SynapseML 为各种不同的机器学习任务（如文本分析、视觉、异常检测等）提供了简单、可组合和分布式的 API。SynapseML 建立在 Apache Spark 分布式计算框架上，并与 SparkML/MLLib 库共享相同的 API，我们可以无缝地将 SynapseML 模型嵌入到现有的 Apache Spark 工作流中。

10.5.3 LightGBM 模型参数

与随机森林等其他算法相比，LightGBM 的调优稍微复杂一些。

LightGBM 有 100 多个参数。刚开始，关注最重要的参数就足以开始使用它了。

（1）max_depth：设置此参数以防止树长得太深。浅的树不太可能过拟合。如果数据集很小，则设置此参数尤为重要。

（2）num_leaves：控制树模型的复杂性。该值应该小于 2^(max_depth),以防止过拟合。

（3）min_data_in_leaf：将此参数设置为较大的值可以防止树长得太深。

（4）max_bin：LightGBM 使用直方图将连续特征值分组到离散桶中。设置 max_bin 以指定将这些值分组到的箱子的数量。

（5）feature_fraction：此参数启用特征子采样。此参数指定将在每次迭代中随机选择的特征的比例。

（6）bagging_fraction：指定将在每次迭代中选择的数据的比例。

（7）num_iteration：设置增强迭代的数量。默认值是 100。

（8）objective：与 XGBoost 一样，LightGBM 支持多个目标。默认的目标设置为 regression。设置此参数以指定模型试图执行的任务类型。对于回归任务，选项有 regression_l2、regression_l1、poisson、quantile、mape、gamma、huber、fair 或 tweedie。对于分类任务，选项有 binary、multiclass、或 multiclassova。

10.5.4 LightGBM on Spark 应用

对于分类任务，SynapseML 提供了 LightGBMClassifier 类，用于建立分类模型。例如，为了预测一家公司是否破产，我们可以使用 LightGBMClassifier 构建一个二元分类模型。对于回归任务，SynapseML 提供了 LightGBMRegressor 类，用于构建回归模型。例如，为了预测房价，我们可以使用 LightGBMRegressor 构建回归模型。

1. 使用 LightGBMClassifier 训练分类模型

在这个例子中，我们使用 LightGBM 来建立一个分类模型来预测一家公司是否会破产。实现步骤如下。

2. 使用 LightGBMRegressor 训练回归模型

在本例中，我们将展示如何使用 LightGBM 构建用于药物发现的分位数回归模型。请按以下步骤操作。

第 11 章 Spark 自然语言处理

Spark NLP 是一个开源的自然语言处理库，建立在 Apache Spark 和 Spark ML 之上。它提供了一个简单的 API 来与 ML pipeline 集成，并且由 John Snow Labs 提供商业支持。Spark NLP 的注解利用基于规则的算法、机器学习和其中一些在底层运行的 Tensorflow 来支持特定的深度学习实现。它提供了自然语言处理领域最新研究的生产级、可扩展和可训练版本，最终目标是让观众在短时间内开始使用这个惊人的库，并使学习曲线平滑。

注意：John Snow Labs 是一家屡获殊荣的人工智能和自然语言处理公司，提供最先进的软件、模型和数据，帮助医疗保健和生命科学组织充分利用人工智能。其主导产品 Spark NLP 是世界上在企业中使用最广泛的 NLP 库。

11.1 什么是 NLP?

自然语言处理（Natural language processing，NLP）是许多数据科学系统中必须理解或推理文本的关键组成部分。常见的应用场景包括问答、释义或总结、情感分析、自然语言 BI、语言建模和消歧。

NLP（自然语言处理）就是以非结构化文本或录音的形式分析人类语言，以便能够理解数据中包含的语言上下文的细微差别。通过 NLP，我们可以将存储在书籍、博客文章和音频文件中的信息转化为有价值的见解。

NLP 在越来越多的人工智能应用中至关重要。在构建聊天机器人、搜索专利数据库、将患者与临床试验相匹配、为客户服务或销售电话评分、从财务报告中提取事实等应用场景中，从免费文本中提取准确信息都是必须的。

与机器学习的所有其他领域一样，NLP 近年来越来越受欢迎。从非结构化文本或音频数据中提取信息对于聊天机器人、高级文本搜索、提取有关数据的事实、分类等应用程序至关重要。

由于近年来 NLP 在数据科学中的流行和炒作，开发了许多很棒的 NLP 库，甚至新手数据科学爱好者也开始使用这些开源库来玩各种 NLP 技术。以下是最流行的 NLP 库，它们在社区中被大量使用，并且处于不同的开发水平：

（1）NLTK：Natural Language Toolkit，所有 NLP 技术的完整工具包。

（2）TextBlob：易于使用的 NLP 工具 API，构建在 NLTK 和 Pattern 之上。

（3）SpaCy：具有工业实力的 NLP，支持 Python 和 Cython。

（4）Gensim：人类主题建模。

（5）Stanford Core NLP：斯坦福 NLP 项目组提供的 NLP 服务和包。

（6）Fasttext：用于学习词嵌入和句子分类的 NLP 库，由 Facebook 的人工智能研究（FAIR）实验室创建。

显然，在一般的 NLP 领域中还有更多的库—但我们在这里关注的是通用的库，而不是那些迎合特定应用的库。

NLP 库提供的常用功能有：

（1）句子检测（sentence detection）：这是任何 NLP 库的基本模块之一，它决定句子的开始和结束。

（2）标记化（tokenization）：将一段文本分成更小的单元的方法。

（3）词干化（stemming）：将一个单词缩减为词干的过程，词干包括后缀和前缀的词缀。

（4）词根化（lemmatization）：类似词干化，但要考虑到单词的意思，例如，将单词 better 词根化为单词 good。

（5）词性标注（POS tagging，或 Part Of Speech tagging）：依据单词的定义及其上下文，根据特定的词性对文本中的单词进行分类。

（6）命名实体识别（NER，或 Named Entity Recognition）：将非结构化文本中的命名实体定位并分类为预定义的类别，如医疗代码、货币价值等。

（7）依存句法分析（dependency parsing）：通过分析句子中单词之间的依赖关系来找出句子的语法结构。

（8）文本匹配（text matching）：估计两个文本片段之间的语义相似度。

（9）分块（chunking）：从非结构化文本中提取短语以识别成分（名词组，动词），使用词性标记（POS）作为输入。

（10）拼写检查（spell-checking）：帮助发现可能的拼写问题。

（11）情感检测（sentiment detection）：分析数据是否有积极、消极或中性的意义。

11.2 Spark NLP 库简介

Spark NLP 是建立在 Apache Spark 之上的最先进的自然语言处理库。它为机器学习管道提供了简单，高性能和准确的 NLP 注释，可以在分布式环境中轻松扩展。Spark-NLP 用 Scala 编写，在 JVM 中运行，并利用了 Spark 的优化和执行计划。

Spark NLP 配备了超过 200 多种语言的 37000 多个预训练管道和模型。它还提供任务，如标记化、分词、词性标记、词和句子嵌入、命名实体识别、依赖解析、拼写检查、文本分类、情感分析、标记分类、机器翻译（+180 种语言）、摘要、问答、表问答、文本生成、图像分类、图像到文本（字幕）、自动语音识别、零样本学习，以及更多的 NLP 任务。

11.2.1 Spark NPL 库的特点

Spark NLP 是提供可扩展 LLM 的开源库。它是唯一在 Apache Spark 上本地构建的 NLP 库。支持 Python、Scala、Java 和 R 语言。企业中使用最广泛的 NLP 库。Spark NLP 是生产中唯一的开源 NLP 库，提供最先进的转换器，如 BERT、CamemBERT、ALBERT、ELECTRA、XLNet、DistilBERT、RoBERTa、DeBERTa、XLM-RoBERTa、Longformer、ELMO、通用句子编码器、Llama-2、M2M100、BART、Instructor、E5、谷歌 T5、MarianMT、OpenAI GPT2, Vision transformers (ViT)、OpenAI Whisper、Llama、Mistral、Phi、Qwen2 等等，不仅适用于 Python 和 R，还通过本地扩展 Apache Spark 适用于大规模的 JVM 生态系统（Java、Scala 和 Kotlin）。

Spark NPL 库的主要特点有：

（1）分布式处理：基于 Apache Spark 构建，能够充分利用 Spark 的分布式计算能力，在集群环境下处理大规模文本数据，大大提高处理效率。

（2）丰富的 NLP 功能：提供了广泛的 NLP 工具和算法，涵盖了从基础的文本处理（如分词、词性标注、命名实体识别等）到高级的文本分析（如情感分析、文本分类、主题建模等）。

（3）预训练模型：包含大量经过预训练的模型，这些模型可以直接用于各种 NLP 任务，无需从头开始训练，节省了时间和计算资源。

（4）多语言支持：支持多种语言的处理，能够处理不同语言的文本数据，满足全球化的应用需求。

（5）易用性：提供了简洁的 API，方便用户使用和集成到现有的 Spark 应用程序中。同时，与 Python、Scala 等编程语言兼容，方便不同背景的开发者使用。

（6）可扩展性：允许用户自定义和扩展其功能，例如自定义模型、特征提取器等，以满足特定的业务需求。

最重要的是，与大多数其他 NLP 库不同，Spark NLP 是在 Apache Spark 和 TensorFlow 上原生构建的。Spark NLP 提供了许多 NLP 功能、预训练模型和管道，开箱即用。

11.2.2 为什么我们需要 Spark NPL 库？

考虑到已经有了这么多的 NLP 库，大家可能会有这样的疑问：为什么我们还需要另一个 NLP 库？主要基于以下几点原因。

1）用于所有 NLP 需求的一个单一的统一的解决方案。

当我们想为实际生产使用交付可伸缩的、高性能的、高精度的 NLP 驱动的软件时，这些库都不能提供统一的解决方案。如图 11-所示。

请记住，任何 NLP 管道始终只是更大的数据处理管道的一部分：例如，问题回答涉及加载训练数据、转换训练数据、应用 NLP 注释器、构建特征、训练值提取模型、评估结果（训练/测试分割或交叉验证）以及超参数估计。我们需要一个一体化的解决方案来减轻文本预处理的负担，并将使用 NLP 将解决数据科学问题的各个步骤之间的点连接起来。因此，我们可以说，一个好的 NLP 库应该能够正确地将自由文本转换为结构化特征，并允许用户训练自己的 NLP 模型，这些模型可以轻松地输入下游机器学习（ML）或深度学习（DL）管道。

2）利用迁移学习，实现 NLP 研究中最新、最好的算法和模型

迁移学习是一种从源设置中提取知识并将其应用于不同目标设置的方法，它是一种非常有效的方法，可以不断提高 NLP 模型的准确性，并且通过利用一些相关任务或领域的已有标记数据，即使是小数据也能获得可靠的准确性。因此，不需要收集数百万个数据点来训练最先进的模型。

在过去的几年里，自然语言处理领域正在发生巨大的变化，现代工业规模的自然语言处理库应该能够实现最新、最好的算法和模型—这并不容易，因为自然语言处理正处于其 ImageNet 时刻，最先进的模型每月被超越两次。

词向量作为 NLP 的核心表示技术的长期统治已经出现了一系列令人兴奋的新挑战者，如 ELMo、BERT、RoBERTa、ALBERT、XLNet、Ernie、ULMFiT、OpenAI transformer，它们都是开源的，包括预训练模型，并且可以在不需要大量计算工作的情况下进行调优或重用。这些作品通过证明预训练的语言模型可以用于在广泛的 NLP 任务中获得最先进的结果而成为头条新闻，有时甚至超过了人类水平的基准。

3）缺乏任何由 Spark 完全支持的 NLP 库

作为一个通用的内存分布式数据处理引擎，Apache Spark 获得了业界的广泛关注，并且已经有了自己的机器学习库（SparkML）和一些其他模块来处理某些 NLP 任务，但它并没有涵盖所有的 NLP 任务，而这些任务需要一个完整的解决方案。当用户尝试在管道中使用 Spark 时，通常需要使用其他 NLP 库来完成某些任务，然后尝试将中间步骤反馈给 Spark。但是，将用户的数据处理框架与 NLP 框架分开意味着用户的大部分处理时间都花在序列化和来回复制字符串上，并且效率非常低。

4）交付关键任务的企业级 NLP 库

如今，许多最流行的 NLP 软件包都有其学术根源—这体现在设计上的权衡中，它们倾向于简化原型而不是运行时性能，倾向于选择的广度而不是简单的极简 API，轻视可伸缩性、错误处理、节省内存消耗和代码重用。

该库已经在企业项目中使用—这意味着第一级的 bug、重构、意外瓶颈和序列化问题已经得到解决。单元测试覆盖率和参考文档的水平使我们能够轻松地将代码开放源代码。

Spark NLP 已经在企业项目中用于各种用例

总而言之，我们迫切需要一个易于学习的 API 的 NLP 库，它可以用你最喜欢的编程语言提供，支持你需要的人类语言，速度非常快，并且可以扩展到包括流和分布式应用在内的大型数据集。

考虑到所有这些问题，流行的 NLP 库的局限性和行业的最新趋势，John Snow Labs，一家帮助医疗保健和生命科学组织更快地将人工智能投入工作的全球人工智能公司，领导并赞助了 Spark NLP 库的开发。该库涵盖了许多常见的 NLP 任务，包括标记化、词干提取、词源化、词性标注、情感分析、拼写检查、命名实体识别等等。

11.2.3 Spark NPL 库的应用

Spark NLP 库 是一个功能强大、易于使用的 NLP 库，适合处理大规模文本数据，在各种 NLP 应用场景中具有广泛的应用前景：

（1）信息提取：从大量文本数据中提取有用的信息，如命名实体识别、关系抽取等。

（2）文本分类：对文本进行分类，如新闻分类、情感分类等。

（3）问答系统：构建智能问答系统，自动回答用户的问题。

（4）舆情分析：分析社交媒体、新闻等文本数据，了解公众对特定事件或产品的看法和态度。

11.3 安装 Spark NLP 库

Spark NLP 是一个开源的自然语言处理库，建立在 Apache Spark 和 Spark ML 之上。它是用 Scala 编写的，它包括 Scala 和 Python API，供 Spark 使用。它不依赖于任何其他 NLP 或 ML 库。对于那些没有使用过 Apache Spark 本身的人来说，安装部分会有点令人厌烦和棘手。

11.3.1 安装 Spark 和 Java

Spark NLP 构建在 Apache Spark 3.x 之上。要使用 Spark NLP，需要：

（1）Java 8 和 11。

（2）Apache Spark 3.3.x, 3.2.x, 3.1.x, 3.0.x。

11.3.2 安装 Spark NLP

根据使用 Scala 还是 Python，有不同的安装方法。

我们可以通过在命令行运行一个 spark-shell 来用 Scala 启动一个 Spark REPL，命令如下（注意，这些步骤需要互联网连接。）：

```

```

3. GPU 支持

Spark NLP 5.5.2 是基于 ONNX 1.17.0 和 TensorFlow 2.7.1 深度学习引擎构建的。至少需要以下 NVIDIA®软件才能支持 GPU：

（1）NVIDIA® GPU 驱动程序版本 450.80.02 或更高。

（2）CUDA® Toolkit 11.2。

（3）cuDNN SDK 8.1.0。

如果你想在 GPU 机器上开始使用 Spark NLP，你可以用这个来启动 Spark 会话

```
sparknlp.start(gpu=True)
```

或者手动启动

```
# Install the necessary Nvidia drivers, CUDA, etc
$ sbt -Dis gpu=true assembly,

# Get FAT-JAR from repo
# start a spark-shell --jars with that jar
$ spark-shell --jars spark-nlp.jar
```

4. 开始使用 Spark NLP

让我们从一个简单的示例开始，以预览 Spark NLP 并确保安装步骤。

我们将使用预训练管道。这里使用的管道处理大多数可以应用于带有几个 Spark NLP 注释器的文本的 NLP 任务。

```

```

11.4 基本组件和底层技术

Spark NLP 引入了 NLP 注释器（Annotator），合并到了 Spark ML 框架内。Annotator 是 spark-nlp 中的核心组件，用于执行各种 NLP 任务，如分词（tokenization）、规范化（normalization）和依赖项解析（dependency parsing）。每个 Annotator 都有特定的功能，例如 Tokenizer 用于分词，NerDLModel 用于命名实体识别等。

现在，让我们从详细解释每个组件开始。

1. 注释器 Annotator

Spark NLP 中的注释器（annotator）是一个组件，它在文本文档上执行特定的 NLP 任务并向其添加注释（annotation）。注释器接受输入文本文档，并生成带有附加元数据的输出文档，这些元数据可用于进一步处理或分析。例如，命名实体识别器注释器可以识别和标记文本文档中的人员、组织和位置等实体，而情感分析注释器可以将文本的情感分类为积极、消极或中立。

注释器是 Spark NLP 中 NLP 功能的主要实现。注释器有两种形式：

（1）Approach：扩展自 Spark ML Estimators，这意味着需要通过 fit()函数基于一些数据来训练模型。训练结果是输出第二种类型的注释器，即注释器模型（annotator model）或转换器（transformer）。

（2）Model：从 Transformer 扩展而来，旨在通过 transform()函数转换 DataFrame。该函数接受一个 DataFrame 作为输入，并向该 DataFrame 添加一个包含当前注释的结果的新列。所有 transformer 都是叠加的，这意味着它们附加到当前数据，永远不会替换或删除以前的信息。

在 Spark NLP 中，所有注释器要么是 Estimators，要么是 Transformers，这两种形式的注释器都可以包含在 Spark 机器学习管道（pipeline）中。管道中包含的所有注释器都将按照定义的顺序自动执行，并相应地转换数据。在 fit()阶段之后，Pipeline 被转换为 PipelineModel。可以将 Pipeline 保存到磁盘并随时重新加载。

2. 注释 Annotation

在 Spark NLP 中，所有操作的基本结果是一个注释（annotation）。其结构包括：

（1）annotatorType：生成当前注释的注释器（annotator）类型。

（2）begin：与原始文本相关的匹配内容的开始位置。

（3）end：与原始文本相关的匹配内容的结束位置。

（4）result：注释的主要输出。

（5）metadata：匹配结果的内容和附加信息。

（6）embeddings：如果有必要，包含向量映射。

例如，图 11-中的 DocumentAssembler 负责将原始文本转换为符合 Spark NLP 处理的结构。通过对输入的 text 列进行转换，输出一个名为 document 的注释（annotation），其直观结构如图 11-所示：

该对象由注释器在转换过程后自动生成，不需要手工操作。然而，清楚地理解注释的结构是非常重要的，这样才能有效地使用它。

3. 预训练的模型

前面小节提到，在 Spark NLP 中，训练过的注释器称为 AnnotatorModel。通过指定的模型（训练过的注释器）将一个 DataFrame 转换为另一个 DataFrame。Spark NLP 支持超过 200 种语言的 31000+的预训练模型，我们所需要做的就是通过指定模型名称将预训练模型加载到磁盘中，然后根据自己的应用场景和数据集配置模型参数。这样就不必从头开始训练新模型，而是可以使用 transform()方法直接将预训练的 SOTA 算法应用于自己的数据。因此用户可以快速注入目标句子并在无需训练模型的情况下返回结果，包括分词、词条、词类（POS）、相似性、实体识别等。

4. 转换器 Transformer

前面我们讨论过，每个 Annotator 需要输入或输出特定类型的列。如果 DataFrame 没有这些类型的列，我们该怎么办？这就需要用到转换器（transformer）。在 Spark NLP 中，有五种不同的转换器，主要用于获取数据或将数据从一个 AnnotatorType 转换为另一个 AnnotatorType。下面列出了这些转换器：

（1）DocumentAssembler：为了完成 NLP 过程，我们需要对原始数据进行注释。这是一个特殊的 transformer，可以帮我们做到这一点。它创建了第一个 Document 类型的注释，以后的注释器可能会使用它。

（2）TokenAssembler：该转换器通常在 token 经过规范化、词序化、拼写检查等操作之后，从 token 重新构造 Document 类型注释，以便在进一步的注释器中使用该 Document 注释。

（3）Doc2Chunk：将 DOCUMENT 类型注释转换为 CHUNK 类型，其中包含 chunkCol 的内容。

（4）Chunk2Doc：将 CHUNK 类型列转换回 DOCUMENT。在尝试重新标记或对 CHUNK 结果进行进一步分析时非常有用。

（5）Finisher：一旦 NLP 管道准备就绪，我们可能希望在其他易于使用的地方使用注释结果。Finisher 将注释值输出到一个字符串中。

5. 管道 Pipeline

什么是管道（Pipeline）？在机器学习中，运行一系列算法来处理和学习数据是很常见的。例如，一个简单的文本文档处理工作流程可能包括以下几个阶段：

（1）将每个文档的文本拆分为句子和标记（token，单词）。

（2）通过应用一些文本预处理技术（清理、词序化、词干提取等）对标记进行规范化。

（3）将每个标记（token）转换为数字特征向量（例如词嵌入，tfidf 等）。

（4）使用这些特征向量和标签学习一个预测模型。

简单地说，管道将多个 Transformer 和 Estimator 链接在一起以指定 ML 工作流。我们使用 Pipeline 将多个 Transformer 和 Estimators 链接在一起，以指定我们的机器学习工作流。

Spark NLP 提供了一个简单的 API 来与 Spark ML pipeline 集成，所有的 Spark NLP 注释器和转换器都可以在 Spark ML pipeline 中使用。图 11-是管道的训练过程使用情况：

Spark NLP 使用管道处理数据，管道结构包含要在输入数据上运行的所有步骤，每个步骤都包含一个 Annotator 组件，该组件执行特定的任务。每个 Annotator 都有输入注释并输出新的注释。管道用于将多个 Annotator 组合在一起，形成一个完整的 NLP 处理流程。用户可以根据需要定义不同的 Pipeline，以实现不同的 NLP 任务。如图 11-所示：

现在让我们看看如何在 Spark NLP 中使用 Annotators 和 Transformer 来实现管道。假设我们需要在一个 DataFrame 上逐一应用以下步骤：

（1）将文本分成句子。

（2）分词（标记字符串）。

（3）规范化（normalize）。

（4）获取词嵌入。

在 Spark NLP 中编写这个管道的代码如下：

```

```

以上工作流可以表示为下面的图 11-所示的流程中。从流程图 11-中可以看到，根据输入列规范，每个生成（输出）列都指向下一个注释器（annotator）作为其输入：

下面是一个管道示例：

```

```

11.5 使用注释器

在 Spark NLP 中，每个注释器只接受特定类型的输入列，并输出另一种类型（我们称之为 AnnotatorType）的新列。在 Spark NLP 中，提供有以下类型：document、token、chunk、pos、word_embeddings、date、entity、sentiment、named_entity、dependency、labeled_dependency。也就是说，在 Spark NLP 中，所有要处理的的 DataFrame 都需要有这些类型之一的列作为注释器的输入。

注释器通常带有如下两个常用方法：

（1）setInputCols(column_names)：获取此注释器所需的注释的列名列表。它们由管道中位于当前注释器前面的注释器生成。

（2）setOutputCol(column_name)：定义包含当前注释器结果的列的名称。使用此名称作为管道中其他注释器的输入，这些注释器需要当前注释器生成的输出。

每个注释器都有一个类型。那些共享一个类型的注释器可以互换使用，这意味着我们可以在需要时使用它们中的任何一个。例如，当一个注释器（如情感分析注释器）需要 token 类型注释器时，可以提供 normalized token 或 lemma，因为两者都是 token 类型。

接下来，我们将介绍 Spark NLP 中一些常用的注释器。

11.5.1 DocumentAssembler

为了完成 NLP 过程，我们需要对原始数据进行注释。有一个特殊的转换器可以为我们做这件事：DocumentAssembler，它将输入的文本数据转换为 Document 对象（即 Document 类型的注释），作为后续处理的基础，下游的注释器以后可能会使用它。

DocumentAssembler 来自 sparknlp.base 类并具有以下可设置的参数。

参数	值	描述
setInputCol()	String	要转换的文本列的名称。我们在这里只能指定一列。它可以读取一个String列或Array。
setOutputCol()	可选	生成的Document类型的列的名称。我们在这里只能指定一列。默认为'document'。
setIdCol()	可选	用于行引用的id列，它是具有id信息的String类型列。
setMetadataCol()	可选	用于document列的元数据，它是具有元数据信息的Map类型列。
setCleanupMode()	可选	清理选项。

DocumentAssembler 读取 String 列。此外，可使用 setCleanupMode 用于预处理文本（默认：disabled）。它可能的取值有：

- ✓ disabled：保持原始资料。如果稍后需要返回源头，会很有用。这是默认设置。
- ✓ inplace：删除换行和制表符。
- ✓ inplace_full：删除换行和制表符，包括已转换为字符串的换行符（即\n）和制表符。
- ✓ shrink：删除换行和制表符，加上合并多个空格和空白行到一个空格。
- ✓ shrink_full：删除换行和制表符，包括字符串化的值，以及缩小的空格和空白行。
- ✓ each：换行和制表符各留一个空格。
- ✓ each_full：换行和制表符（也包括字符串化的）各留一个空格。
- ✓ delete_full：删除字符串化的换行符和制表符（不替换）。

使用 com.johnsnowlabs.nlp.DocumentAssembler 加载待处理文档，这是每个 Spark NLP 管道的入口点。使用它将数据准备为 Spark NLP 可处理的格式。使用它的模板代码如下：

```

```

在上面的代码中，首先，我们用所需的参数定义 DocumentAssembler，然后用它来转换 DataFrame。这里需要注意的最重要的一点是，需要在.setInputCol()中使用 String 或 String[Array]类型的列。所以它不需要必须命名为 text。只需按原样使用列名。

经过转换后，输出列 document 的结构如图 11-所示：

新列是一个 struct 类型的数组，具有如上所示的参数。注释器(annotators)和转换器(transformers)都带有通用元数据，这些元数据将根据所使用的注释器来填充。除非想将其他 Spark NLP 注释器附加到 DocumentAssembler()中，否则现在不需要知道所有这些参数的含义。

上面的示例处理的仅有一行文本。下面这个示例演示了如何处理一个多行文本，以及更详细地查看转换后的 document 列。代码如下：

```

```

执行以上代码，输出内容如下：

11. 5. 2 SentenceDetector

在自然语言处理（NLP）中，Sentence Detection（句子检测，也称为句子分割）是指将一段连续的文本分割成一个个独立句子的过程。这是 NLP 里的基础任务，后续很多处理，像句法分析、信息提取、机器翻译等，都依赖准确的句子分割结果。

句子检测的难点在于不同语言有不同的句子结束标识，且存在一些特殊情况。比如英语中，常见的句子结束标识有句号（.）、问号（?）和感叹号（!），但缩写词里的句号就不能作为句子结束的标志（如 “Mr.” “Dr.” ）。中文句子结束通常用句号（。）、问号（？）、感叹号（！），不过一些引用、省略等情况也会增加分割的难度。

Spark NLP 库提供了 SentenceDetector 组件用于识别输入文档中的句子。它是一个是基于规则的句子检测方法，使用正则表达式检测句子边界的注释器（annotator），需要由 DocumentAssembler 输出所提供的 Document 注释作为输入。

SentenceDetector 类位于 com.johnsnowlabs.nlp.annotators.sbd.pragmatic 包中，它本身就是一个 Document 类型的 token。在该注释器中，以下字符被检查为句子边界：

- ✓ Lists ("(i), (ii)", "(a), (b)", "1., 2.")
- ✓ 数字
- ✓ 缩写
- ✓ 标点符号
- ✓ 多个句号
- ✓ 地理位置/坐标 ("N°. 1026.253.553.")
- ✓ 省略号("...")
- ✓ 中间标点符号
- ✓ 引号
- ✓ 感叹号
- ✓ 基本的分隔符 (".", ";")

如果想要添加额外的自定义边界，可以使用数组设置参数 customBounds，示例如下：

```
val sentence = new SentenceDetector()
  .setInputCols("document")
  .setOutputCol("sentence")
  .setCustomBounds(Array("\n\n"))
```

如果只使用自定义边界，则应设置参数 useCustomBoundsOnly 为 true。

每个提取的句子可以在一个 Array 数组中返回；如果 explodeSentences 参数被设置为 true，则分解为单独的行。

请看下面的示例：

```

```

执行以上代码，可以看到语句检测后返回结果的模式如下：

11.5.3 Tokenizer

分词，是自然语言处理（NLP）中的一项基础任务，其主要目标是将连续的文本序列按照一定的规则和语言知识切分成有意义的词语单元。词语分割在 NLP 的多个应用场景中都起着关键作用，例如信息检索、机器翻译、文本分类、命名实体识别等。准确的分词结果能够提高后续 NLP 任务的性能。

com.johnsnowlabs.nlp.annotators.Tokenizer 用于将文档类型（document type）列中的原始文本标记为 TokenizedSentence。可将其理解为一个分词器，它使用标记化开放标准标识一个文本语句中的单词。这个类表示一个非拟合 tokenizer。拟合它将导致内部 RuleFactory 从输入配置构造用于标记（tokenizing）的规则。

Spark NLP 提供了用于词语分割的组件，其中 Tokenizer 是一个常用的分词器，它可以将文本分割成单词或标记。Tokenizer 需要 Document 注释类型，这意味着它既适用于 DocumentAssembler 输出，也适用于 SentenceDetector 输出。例如，在下面的模板代码中，使用 SentenceDetector 输出：

```
// scala
val sentenceDetector = new SentenceDetector()
    .setInputCols(Array("document"))
    .setOutputCol("sentence")

val regexTokenizer = new Tokenizer()
    .setInputCols(Array("sentence"))
    .setOutputCol("token")
```

Tokenizer 具有以下可设置的参数：

参数	值	描述
setCaseSensitiveExceptions()	Boolean	是否关心异常中的大小写敏感（默认值：true）。
setContextChars()	Array[String]	用于从单词边界分离的字符列表，默认：Array(".", ",", ";", ":", "!", "?", "*", "-", "(", ")", "\"", "'")。
setMaxLength()	Int	设置每个单词允许的最大长度。
setMinLength()	Int	设置每个单词允许的最小长度。
setSplitChars()	Array[String]	用于从单词内部进行分隔的字符串列表，例如连字符。

下面这个示例演示了 Tokenizer 标注器的用法。代码如下：

```
```

执行以上代码，输出内容如下：

11.5.4 RegexTokenizer

按正则表达式模式分割文本的 tokenizer。需要使用 setPattern 设置模式，这将设置分隔模式或 tokens 应该如何分割。默认情况下，此模式为\s+，这意味着 tokens 应该被 1 个或多个空白字符分割。它接收的输入注释类型（Annotator Type）为 DOCUMENT，输出注释类型（Annotator Type）为 TOKEN。

执行以上代码，输出内容如下：

Spark NLP 没有像处理英文那样有专门针对中文设计的、经过大量优化和验证的分词器，但可以使用 RegexTokenizer 配合合适的正则表达式进行简单的中文分词。不过这种方式通常是基于字符或简单规则的，分词效果有限。

```
```

执行以上代码，输出内容如下：

此示例中的正则表达式 (?U)会将每个汉字作为一个单独的分词结果。如果需要更复杂的分词规则，可以根据具体需求修改正则表达式。

11.5.4 TextMatcher

TextMatcher 位于 com.johnsnowlabs.nlp.annotators 包中，它是 Spark NLP 的注释器，这意味着它通过将文件中提供的精确短语（通过 token）与 Document 进行匹配来进行操作。因此，必须使用 setEntities()方法来提供一个预定义短语的文本文件。该文本文件也可以直接设置为 ExternalResource。

ExternalResource（位于 com.johnsnowlabs.nlp.util.io 包中）代表了一个外部源，其中包含了 Spark-NLP 的资源助手如何读取外部资源的信息：

- ✓ ReadAs.TEXT：将文件配置为在本地作为文本读取。
- ✓ ReadAs.BINARY：将文件配置为在本地以二进制格式读取。
- ✓ ReadAs.SPARK：将配置该文件由 Spark 读取。需要在 options 中定义 format，其你在值可以是 text 或 json 或 parquet。

例如：

```
ExternalResource(
  "src/test/resources/regex-matcher/rules.txt",
  ReadAs.TEXT,
  Map("delimiter" -> ",")
)

ExternalResource(
  "src/test/resources/regex-matcher/rules.txt",
  ReadAs.SPARK,
  Map("format" -> "text", "delimiter" -> ",")
)
```

TextMatcher 需要 DOCUMENT 和 TOKEN 作为输入，然后提供 CHUNK 作为输出。它具有以下可设置的参数：

参数	值	描述
setBuildFromTokens()	Boolean	是否应该从TOKEN中获取CHUNK（默认值：false）。
setCaseSensitive()	Boolean	是否区分大小写（默认值：true）。
setEntities()	ExternalResource	实体的外部资源，例如文本文件，其中每行是一个实体的字符串
setEntityValue()	String	为实体元数据字段设置值（默认：entity）。
setMergeOverlapping()	Boolean	设置是否合并重叠的匹配块（默认：false）。
setTokenizer()	TokenizerModel	设置要执行tokenization的Tokenizer。

假设我们有一个包含实体资源的文本文件 test-phrases.txt，其内容如下所示：

```
...
dolore magna aliqua
```

```
lorem ipsum dolor. sit
Laborum
...
```

其中每行表示一个要提取的实体短语。我们现在想从一段文本中按上述实体资源定义提取相关的内容，下面是提取实体的示例代码：

```
\
```

执行以上代码，输出结果如下：

11.5.5 BigTextMatcher

BigTextMatcher（位于 com.johnsnowlabs.nlp.annotators.btm 包中）是一个注释器，用于将文件中提供的精确短语（通过 token）与 Document 进行匹配。它是 Spark NLP 中 TextMatcher 组件的扩展，它允许用户在非常大的文档或语料库中匹配和提取文本模式。它的设计目的是处理太大而无法装入内存的数据集，并提供了一种使用 Spark NLP 执行分布式模式匹配和文本提取的有效方法。

BigTextMatcher 的工作原理是将一组单词或短语作为输入（必须使用 setStoragePath()方法提供预定义短语的文本文件，文本文件也可以直接设置为ExternalResource，与 TextMatcher 组件类似），并构建一个有效的数据结构来存储它们。这种数据结构允许 BigTextMatcher 根据大量的单词或短语快速匹配输入文本，使其在处理大量数据时比 TextMatcher 快得多。

下面让我们研究一个类似于 TextMatcher 的例子，使用 BigTextMatcher 从文本中提取人物实体。首先，让我们定义要匹配的名称，并将它保存为文本文件。

person_matches_big.txt：

```
Karen Smith
John Johnson
Jane Kim
Michael Brown
Emily Zhang
```

然后在代码中从一个待处理的示例文本创建一个 DataFrame，并创建一个数据处理管道对该数据集进行转换和处理。代码如下：

```

```

执行以上代码，输出结果如下：

11.5.6 RegexMatcher

在 Spark NLP 中，RegexMatcher 是一个用于使用正则表达式对文本数据执行模式匹配的组件。Spark NLP 中的正则匹配是指根据用户定义的模式和规则，使用正则表达式（regex）对文本数据进行搜索、提取和操作的过程。

在正则表达式匹配中，用户使用文字字符和在正则表达式引擎中具有特殊含义的特殊字符或元字符的组合来定义模式。RegexMatcher 然后在输入文本数据中搜索与所定义模式匹配的字符序列，并返回匹配结果。

RegexMatcher 的工作方式是将一组正则表达式作为输入，并将它们应用于文本数据。当找到匹配项时，RegexMatcher 输出匹配的文本及其在输入文本中的起始和结束位置。

RegexMatcher 使用规则匹配一组正则表达式，并将它们与提供的标识符关联。规则必须由正则表达式模式和该模式的标识符组成。正则表达式模式和标识符必须用一个字符分隔，该字符也必须用 setDelimiter()方法设置。例如正则表达式“\d{4}\/\d\d\/\d\d,date”，它将匹配像“1970/01/01”这样的字符串到标识符“date”。规则必须由 serules 方法（后跟 setDelimiter 方法）或外部文件提供。要使用外部文件，必须使用 setExternalRules 方法提供预定义正则表达式的字典。可以将字典设置为带分隔符的文本文件。

让我们使用一个示例文本来查看 RegexMatcher 注释器的性能，然后对 DataFrame 进行拟合和转换以获得结果.

以上代码中，在 setRegexPatterns()方法 中定义了日期匹配的正则表达式 r'\d{2,4}/\d{1,2}/\d{1,2}'，表示匹配 2-4 位数字、斜杠、1-2 位数字、斜杠、1-2 位数字的格式。setStrategy("MATCH_ALL") 表示匹配所有符合条件的结果。

执行以上代码，输出内容如下：

11.5.7 ChunkTokenizer

ChunkTokenizer（位于 com.johnsnowlabs.nlp.annotators 包中）是 Tokenizer 类的子类。它用来分词并扁平化抽取 NER 块。ChunkTokenizer 将对提取的 NER CHUNK 类型标注进行拆分，并将创建 TOKEN 类型标注。然后将结果平摊，从而得到单个数组。

请看下面的使用示例。

执行以上代码，输出结果如下：

11.5.8 DateMatcher

DateMatcher 是 Spark NLP 中的一个组件，用于在文本中识别和提取日期信息。它可以识别多种日期格式，并且可以将提取的日期转换为指定的标准格式。每个 document 文档只提取一个日期。使用句子检测器来查找每个句子中的匹配项。例如，“2008 年 4 月 31 日”将被转换为 2008/04/31。

DateMatcher 的主要方法：

（1）setInputCols：设置输入列的名称，通常是经过 DocumentAssembler 处理后的 document 列。

（2）setOutputCol：设置输出列的名称，用于存储提取的日期信息。

（3）setInputFormats：设置多个输入日期的格式，参数为字符串数组。

（4）setOutputFormat：设置输出日期的格式，例如 yyyy/MM/dd。

（5）setAnchorDateYear：为相对日期添加一个锚年份。如果未设置，则默认使用当前年份，例如：2025。

（6）setAnchorDateMonth：为相对日期添加一个锚月份。默认情况下，它将使用当前月份。月份的值从 1 开始，所以 1 代表一月份。

（7）setAnchorDateDay：为相对日期添加一个锚日。默认情况下，它将使用当前日期。一个月的第一天的值是 1。

（8）setSourceLanguage：设置输入文本的语言，默认为 en（英语）。

DateMatcher 可以读取如下类型的日期：

```
"1978-01-28", "1984/04/02,1/02/1980", "2/28/79", "The 31st of April in the
 year 2008", "Fri, 21 Nov 1997", "Jan 21, `97", "Sun", "Nov 21", "jan 1st",
 "next thursday", "last wednesday", "today", "tomorrow", "yesterday", "nex
t week", "next month", "next year", "day after", "the day before", "0600h",
 "06:00 hours", "6pm", "5:30 a.m.", "at 5", "12:59", "23:59", "1988/11/23
6pm", "next week at 7.30", "5 am tomorrow"
```

请看下面的这个从文本中识别和提取日期信息的示例。

```

```

执行以上代码，输出内容如下：

```
+------------------+----------+
|text              |date      |
+------------------+----------+
|Fri, 21 Nov 1997  |1997-11-21|
|next week at 7.30 |2025-03-20|
|see you a day after|2025-03-14|
+------------------+----------+
```

要从文档中提取多个日期，可以使用 MultiDateMatcher。MultiDateMatcher 是 DateMatcher 的扩展版本，它可以同时处理多种语言的日期识别，并且能够更灵活地处理复杂的日期表达。与 DateMatcher 相比，MultiDateMatcher 支持更多的日期格式和语言。

下面是使用 MultiDateMatcher 提取多个日期值的示例。

执行以上代码，输出内容如下：

11.5.9 Normalizer

Normalizer（位于 com.johnsnowlabs.nlp.annotators 包中）是一种清除 tokens 的注释器。需要词根（stems），因此需要 tokens。从文本中删除遵循 regex 模式的所有脏字符，并根据提供的字典转换单词。

请看下面的使用示例。

```

```

执行以上代码，输出内容如下：

11.5.10 Finisher

Spark NLP 还包括另一个特殊的转换器，称为 Finisher（位于 com.johnsnowlabs.nlp 包中），用于将注释结果转换为更易于使用的格式。它主要用于从 Spark NLP Pipeline 中提取结果。Finisher 将注释值输出到 String 中。

在 Spark NLP 完成的每个管道或任何阶段结束时，我们可能希望将结果输出到另一个管道或简单地将它们写入磁盘上。Finisher 注释器可以帮助我们清理元数据（如果它被设置为 true）并将结果输出到一个数组中。其基本用法如下：

```
// scala
val finisher = new Finisher()
    .setInputCols("token")
    .setIncludeMetadata(true)
```

```
# python
finisher = Finisher() \
    .setInputCols(["token"]) \
    .setIncludeMetadata(True)
```

如果我们需要从任何注释中获得一个扁平的 DataFrame（每个子数组在一个新的列中），而非结构类型的列，则可以使用 Spark SQL 中的 explode()函数。还可以使用 Apache Spark 函数（SQL）以需要的任何方式操作输出 DataFrame。这里我们将 tokens 和 NER 结果结合在一起。请看下面的示例。

```

```

执行以上代码，输出内容如下：

11.5.11 NGram

org.apache.spark.ml.feature.NGram

将输入字符串数组转换为 n-gram 数组的特征转换器。输入数组中的 Null 值将被忽略。它返回一个 n-gram 数组，其中每个 n-gram 由空格分隔的单词字符串表示。

当输入为空时，返回一个空数组。当输入数组长度小于 n（每 n-gram 的元素数）时，不返回 n-gram。

下面的示例结合 spark-nlp 和 Spark 的 NGram，从一个原始文本中提取 3-gram 短语。代码如下：

```

```

执行以上代码，输出内容如下：

11.5.12 NGramGenerator

com.johnsnowlabs.nlp.annotators.NGramGenerator，是一个将字符串（annotatorType TOKEN）的输入数组转换为 n-grams（annotatorType CHUNK）的数组的特征转换器。输入数组中的 Null 值将被忽略。它返回一个 n-gram 数组，其中每个 n-gram 由空格分隔的单词字符串表示。

当输入为空时，返回一个空数组。当输入数组长度小于 n（每 n-gram 的元素数）时，不返回 n-gram。

执行以上代码，输出内容如下：

11.5.13 SentenceEmbeddings

SentenceEmbeddings 是 Spark NLP 中的一个组件，用于将句子转换为固定长度的向量表示，也就是句子嵌入（sentence embeddings）。句子嵌入是自然语言处理中的重要概念，它可以将文本信息转换为数值向量，方便后续的机器学习和深度学习模型进行处理，比如文本分类、语义相似度计算等任务。

SentenceEmbeddings 通常会基于词嵌入来生成句子嵌入，常见的做法是将句子中各个词的词嵌入通过某种方式组合起来，例如汇总、取平均值、加权平均等。Spark NLP 提供了多种预训练的词嵌入模型供 SentenceEmbeddings 使用，如 WordEmbeddingsModel、Bert Embeddings 等。

SentenceEmbeddings 设置主要参数的方法有：

（1）setInputCols：设置输入列的名称，一般需要一个包含词嵌入的列和一个包含句子信息的列。

（2）setOutputCol：设置输出列的名称，用于存储生成的句子嵌入。

（3）setPoolingStrategy：设置句子嵌入的池化策略，常见的有 AVERAGE（取词嵌入的平均值）、SUM（取词嵌入的总和）等。

请看下面实现句子嵌入的示例程序。

执行以上代码，输出内容如下：

11.5.14 StopWordsCleaner

这个注释器接受一个字符串序列（例如 Tokenizer、Normalizer、Lemmatizer 和 Stemmer 的输出），并从输入序列中删除所有停止词。

默认情况下，它使用来自 Spark 机器学习库中 StopWordsRemover 的停止词。停止词也可以通过使用 setStopWords（value: Array[String]）显式设置它们来定义，或者使用其伴生对象的 pretrained ()方法从预训练的模型中加载。模板代码如下：

```
val stopWords = StopWordsCleaner.pretrained()
  .setInputCols("token")
  .setOutputCol("cleanTokens")
  .setCaseSensitive(false)
```

这将加载默认的预训练模型"stopwords_en"。下面是其使用示例。

```

```

11.6 使用预训练模型

Spark NLP 具有超过 250 种语言的 37,000 多个预训练管道和模型。它支持大多数 NLP 任务，并提供可以在集群中无缝使用的模块。

Spark NLP 支持多语言 NER 模型，包括：阿拉伯语、孟加拉语、中文、丹麦语、荷兰语、英语、芬兰语、法语、德语、希伯来语、意大利语、日语、韩语、挪威语、波斯语、波兰语、葡萄牙语、俄语、西班牙语、瑞典语、乌尔都语等等。

我们所需要做的就是通过指定模型名称将预训练模型加载到磁盘中，然后根据自己的用例和数据集配置模型参数。我们不必担心从头开始训练新模型，并且可以使用 transform() 直接将预训练的 SOTA 算法应用于自己的数据。

使用预训练模型具有以下优势：

（1）节省时间和资源：预训练模型是在大规模语料库上进行训练的，用户无需自己收集和标注大量数据，也无需花费大量时间和计算资源来训练模型。

（2）性能较好：由于预训练模型在大规模数据上进行了训练，通常能够在各种词性标注任务中取得较好的性能。

（3）方便使用：spark-nlp 提供了简单的 API 来加载和使用预训练模型，用户可以轻松地将其集成到自己的 NLP 处理流程中。

预训练模型的用法示例如下：

11.6.1 查看可用的预训练模型

Spark NLP 中有一些函数可以列出特定注释器和语言的所有可用模型。显示有哪些可用的模型，代码如下：

```
import com.johnsnowlabs.nlp.pretrained.ResourceDownloader

ResourceDownloader.showPublicModels(annotator = "NerDLModel", lang = "en
")
```

11.6.2 词性标注

词性标注（Part-of-Speech Tagging，简称 POS 标注）是自然语言处理（NLP）中的一项基础且重要的任务，它的目标是为文本中的每个词语标注其对应的词性，例如名词、动词、形容词等。

词性信息有助于分析句子的语法结构，确定词语之间的关系。例如，在分析句子“I love you.” 时，通过词性标注知道 “I” 是代词作主语，“love” 是动词作谓语，“you” 是代词作宾语，从而清晰地把握句子的基本结构。

词性标注是许多高级 NLP 任务的基础，如句法分析、语义理解、信息提取等。例如，在命名实体识别任务中，词性信息可以辅助判断一个词语是否可能是人名、地名等实体；在机器翻译中，词性标注有助于准确地进行词汇的选择和语法的转换。

```

```

11.6.3 词性还原

词形还原（Lemmatization）是自然语言处理中的一种技术，将词汇还原为其基本形式（词元），以便在文本分析中进行统一处理。Spark NLP 提供了多个预训练模型可用于词性还原。用于词性还原的预训练模型模板代码如下：

```
val lemmatizer = LemmatizerModel.pretrained()
  .setInputCols(Array("token"))
  .setOutputCol("lemma")
```

这将加载默认的预训练模型，即“lemma_antbnc”。这是一个基于英国国家语料库（BNC）进行训练的词性还原模型。它能够将单词还原为其基本形式，对于处理英语文本的词性还原任务非常有效。使用该预训练模型的示例如下：

```

```

执行以上代码，输出内容如下：

```

```

11.6.4 句子检测

Spark NLP 提供了 SentenceDetectorDLModel 预训练模型来进行句子检测。它是基于深度学习的预训练模型，能更准确地处理复杂的句子分割情况。可以使用其伴生对象的 pretrained()方法加载该预训练模型，模板代码如下：

```
val sentenceDL = SentenceDetectorDLModel.pretrained()
    .setInputCols("document")
    .setOutputCol("sentencesDL")
```

如果没有提供名称，则默认模型为“sentence_detector_dl”。每个提取的句子以数据形式返回；如果 explodeSentences 设置为 true，则分解为单独的行。

在下面的示例中，将普通的 SentenceDetector 与 SentenceDetectorDLModel 进行比较。在管道中，SentenceDetectorDLModel 可用作 SentenceDetector 的替代品。代码如下：

```

```

执行以上代码，输出内容如下：

```

```

11.6.5 嵌入

在自然语言处理（NLP）中，Embeddings（嵌入）是一种将离散的符号（如单词、字符或句子）映射到连续向量空间的技术。这些向量（即嵌入向量）能够捕捉到符号之间的语义和句法关系。例如，意思相近的单词在向量空间中的距离会比较近。

常见的嵌入类型包括：

（1）词嵌入：将单词转换为固定长度的向量，如 Word2Vec、GloVe、FastText 等。

（2）句子嵌入：将整个句子转换为单个向量，用于表示句子的语义信息，如 Universal Sentence Encoder（USE）、BERT 生成的句子嵌入等。

Spark NLP 提供了多种预训练的嵌入模型，以下介绍一些常见的模型。

1. WordEmbeddingsModel

WordEmbeddingsModel（glove_100d）是 Spark NLP 库中一个基于 GloVe（Global Vectors for Word Representation）算法训练的 100 维词嵌入模型。词嵌入是自然语言处理（NLP）中的一种技术，它将文本中的每个词映射到一个固定长度的向量空间中，使得语义相近的词在向量空间中的距离也相近。glove_100d 中的 “100d” 表示每个词被表示为一个 100 维的向量。

在 Spark NLP 中，该模型可用于以下场景：

（1）文本分类：在将文本分类到不同的类别（如新闻分类、情感分析等）时，glove_100d 可以将文本中的词转换为向量表示，然后作为特征输入到分类模型中，帮助模型更好地理解文本的语义信息，从而提高分类的准确性。

（2）命名实体识别：在识别文本中的命名实体（如人名、地名、组织机构名等）时，词嵌入可以为每个词提供更丰富的特征信息，使得模型能够更准确地判断词是否属于某个实体类别。

（3）文本相似度计算：通过计算两个文本中词向量的相似度（如余弦相似度），可以判断两个文本在语义上的相似程度，用于信息检索、文本匹配等任务。

下面是一个使用 glove_100d 词嵌入的示例程序。

```

```

执行以上代码，输出内容如下：

```

```

2. BertEmbeddings

BertEmbeddings 是基于 BERT（Bidirectional Encoder Representations from Transformers）架构的词嵌入预训练模型，可用于生成词和句子的嵌入。BERT 是由 Google 开发的预训练语言模型，采用了 Transformer 架构中的编码器部分，能够捕捉文本中词与词之间的双向上下文信息，这与传统的单向语言模型有很大不同，能更好地理解文本语义。

BertEmbeddings 预训练模型可以通过其伴生对象的 pretrained()方法加载：

```
val embeddings = BertEmbeddings.pretrained()
  .setInputCols("token", "document")
```

```
  .setOutputCol("bert_embeddings")
```

如果未提供名称，则默认模型为“small_bert_L2_768”。

在 Spark NLP 中，该模型可用于以下场景：

（1）文本分类：在将文本分类到不同类别（如新闻分类、情感分析等）时，可以将文本转换为向量表示，为分类模型提供丰富的语义特征，有助于提高分类的准确性。

（2）命名实体识别（NER）：能够为每个词生成上下文相关的嵌入向量，帮助模型更准确地识别文本中的命名实体，如人名、地名、组织机构名等。例如，在处理包含人名的句子时，该模型能根据上下文判断出哪些词构成了一个人名实体。

（3）问答系统：在问答任务中，将问题和候选答案分别转换为向量表示，通过计算向量之间的相似度来找到最匹配的答案。small_bert_L2_768 的上下文感知能力可以更好地理解问题和答案的语义，提高问答的准确性。

下面是一个使用 BertEmbeddings 词嵌入的示例程序。

```

```

3. UniversalSentenceEncoder

UniversalSentenceEncoder（通用句子编码器）是 Spark NLP 库中一个强大的句子嵌入模型，它的核心目标是将输入的句子转换为固定长度的向量表示，也就是嵌入向量。这种向量表示能够捕捉句子的语义信息，使得语义相近的句子在向量空间中的距离较近，从而可以方便地用于各种自然语言处理任务。

该模型基于深度学习技术，它的设计理念是能够跨语言和不同的任务场景，提供一种通用的句子表示方法。其具体实现结合了多种技术，通常会利用 Transformer 架构或者类似的神经网络结构，在大规模的文本数据上进行训练，学习到句子的语义特征和模式。

UniversalSentenceEncoder 预训练模型可以通过其伴生对象的 pretrained()方法加载：

```
val useEmbeddings = UniversalSentenceEncoder.pretrained()
  .setInputCols("sentence")
  .setOutputCol("sentence_embeddings")
```

如果未提供名称，则默认模型为“tfhub_use”。

通用句子编码器将文本编码为高维向量，在 Spark NLP 中，可用于以下的任务场景：

（1）文本分类：将句子转换为嵌入向量后，可以将这些向量作为特征输入到分类模型（如支持向量机、神经网络等）中，用于判断句子所属的类别，如情感分析（积极、消极、中性）、新闻分类（体育、科技、娱乐等）。

（2）语义相似度计算：通过计算两个句子嵌入向量之间的相似度（如余弦相似度），可以判断它们在语义上的相似程度。这在信息检索、问答系统、文本匹配等任务中非常有用。例如，在问答系统中，可以根据问题和候选答案的语义相似度来选择最合适的答案。

（3）聚类分析：将大量句子的嵌入向量进行聚类，把语义相近的句子归为一类。这有助于对文本数据进行组织和分析，发现数据中的潜在模式和主题。

下面是一个使用 UniversalSentenceEncoder 句子嵌入的示例程序。

```

```

执行以上代码，输出内容如下：

11.6.6 文本分类

文本分类（Text Classification）是自然语言处理（NLP）中的一项基础且重要的任务，它旨在将文本数据划分到一个或多个预定义的类别中。

文本分类的核心是为给定的文本找到其所属的类别标签。这些类别可以是多种多样的，例如新闻文章的主题分类（政治、体育、娱乐等）、情感分析中的情感极性分类（积极、消极、中性）、垃圾邮件检测中的分类（垃圾邮件、正常邮件）等。

文本分类通常应用在以下场景：

（1）新闻分类：新闻媒体平台每天会产生大量的新闻文章，通过文本分类技术可以自动将这些文章归类到不同的主题类别下，方便用户快速找到自己感兴趣的新闻内容。

（2）情感分析：在社交媒体、电商评论等场景中，分析用户文本的情感倾向，有助于企业了解用户对产品或服务的满意度，及时调整经营策略。

（3）垃圾邮件过滤：邮件服务提供商利用文本分类技术识别并过滤垃圾邮件，提高用户的邮件使用体验，减少垃圾信息的干扰。

（4）法律文档分类：在法律领域，大量的法律文档可以根据其内容进行分类，如合同、判决书、法律法规等，便于法律工作者快速查找和使用相关文档。

Spark NLP 提供了多种用于文本分类的预训练模型，以下是一些常见的模型：

（1）classifierdl_use_trec6：它主要用于文本的主题分类，具体是在 TREC - 6（Text REtrieval Conference - 6）数据集上训练的。这个数据集包含了不同主题的问题，所以该模型擅长将文本划分到像描述、实体、数字、地点等不同的主题类别中。

（2）classifierdl_use_emotion：可用于社交媒体监控、客服聊天记录分析、内容审核等场景，通过识别文本的情感，了解用户的情绪状态，从而采取相应的措施。

（3）classifierdl_use_sarcasm：这是一个用于检测文本是否具有讽刺意味的分类模型。在自然语言处理任务中，讽刺检测是一个具有挑战性的任务，因为讽刺往往依赖于上下文、语气和隐含的语义，而这个模型旨在自动化地完成这一检测工作。

这些预训练的模型可以使用伴生对象 ClassifierDLModel 的 pretrained()方法加载。模板示例代码如下：

```
val classifierDL = ClassifierDLModel.pretrained()
    .setInputCols("sentence_embeddings")
    .setOutputCol("classification")
```

如果没有提供名称，则默认模型为“classifierdl_use_trec6”。ClassifierDLModel 使用最先进的通用句子编码器（Universal Sentence Encoder）的嵌入作为文本分类的输入，并在 TREC-6 数据集上进行训练。ClassifierDLModel 注释器使用在 TensorFlow 内部构建的深度学习模型（DNNs），支持多达 100 个类。

在下面的示例中，使用 classifierdl_use_sarcasm 模型对示例文本数据进行讽刺检测。该模型使用了 Universal Sentence Encoder 作为特征提取器。在提取文本的特征向量之后，模型使用深度学习分类器（ClassifierDL）来判断输入文本是否具有讽刺意味。代码如下：

```

```

11.6.7 情感分析

情感分析（Sentiment Analysis），也被称为意见挖掘（Opinion Mining），是自然语言处理（NLP）中的一个重要任务。其核心目标是通过对文本内容进行分析，判断文本所表达的情感倾向，通常分为积极（Positive）、消极（Negative）和中性（Neutral）三种类型，当然也可以进行更细致的分类，比如分为非常积极、比较积极、中性、比较消极、非常消极等。

情感分析在多个领域都有广泛的应用，包括：

（1）商业领域：企业可以通过分析用户在社交媒体、电商平台上的评论，了解消费者对产品或服务的满意度，从而改进产品、优化服务，制定营销策略。

（2）舆情监测：政府和相关机构可以通过分析新闻报道、社交媒体言论等，了解公众对特定事件或政策的态度和情绪，及时进行舆情引导和危机公关。

（3）金融领域：分析财经新闻、股民的评论等，预测股票市场的走势和投资者的情绪，辅助投资决策。

Spark NLP 库提供了用于多类情感分析的注释器 SentimentDL，其实例模型是 SentimentDLModel。可通过其伴生对象的 pretrained()方法加载这个预训练模型，模板代码如下：

```
val sentiment = SentimentDLModel.pretrained()
  .setInputCols("sentence_embeddings")
  .setOutputCol("sentiment")
```

如果没有提供名称，则默认模型为“sentimentdl_use_imdb”。这是在 IMDB 数据集上训练的英语情感分析模型。

除此之外，Spark NLP 还提供了一些其他用于情感分析的预训练模型，例如 sentimentdl_use_twitter，它基于通用句子编码器（Universal Sentence Encoder，USE），在 Twitter 数据上进行训练，可以对文本的情感进行分类。请看下面的示例：

```

```

11.6.8 文本摘要

在自然语言处理（NLP）领域，文本摘要（Summarization）是一种将长文本内容精简为较短、更凝练版本的技术，同时保留原文的核心信息和关键要点。它主要有两种类型：

（1）抽取式摘要（Extractive Summarization）：从原文中挑选出最重要的句子、短语或段落，直接组合形成摘要，不改变原文的表述。例如，在一篇新闻报道中，选取关键的段落或句子来构成摘要。

（2）生成式摘要（Abstractive Summarization）：在理解原文语义的基础上，使用全新的表述来生成摘要，不局限于原文中的词句。这需要模型具备更强的语义理解和语言生成能力，类似于人类阅读文章后用自己的话概括内容。

文本摘要在很多场景中都有重要应用，如新闻资讯平台为用户快速提供文章要点、信息检索时对文档进行预览、智能客服自动生成对话摘要等。

Spark NLP 从 4.2.0 版本开始提供了基于 T5 架构的文本摘要模型，可用于生成式文本摘要任务。所谓 T5，指的是“the Text-To-Text Transfer Transformer（文本到文本传输转换器）”。T5 将所有 NLP 任务重新考虑为统一的文本到文本格式，其中输入和输出始终是

文本字符串，而 bert 风格的模型只能输出类标签或输入的一个范围。文本到文本框架能够在任何 NLP 任务上使用相同的模型、损失函数和超参数，包括机器翻译、文档摘要、问题回答和分类任务（例如情感分析）。T5 甚至可以应用于回归任务，通过训练它来预测数字的字符串表示，而不是数字本身。

可以使用 T5Transformer 伴生对象的 pretrained()方法来加载该预训练模型，模板代码如下：

```
val t5 = T5Transformer.pretrained()
    .setTask("summarize:")
    .setInputCols("document")
    .setOutputCol("summaries")
```

如果没有提供名称，则默认模型为“t5_small”。下面这个示例演示了如何加载默认模型来提取一段英文文本的摘要信息，代码如下：

```

```

执行以上代码，输出内容如下：

```

```

注意： 截至 2024 年 7 月，Spark NLP 本身没有专门针对中文的预训练摘要模型。

11.6.9 依存句法分析

在自然语言处理（NLP）里，依存句法分析器（Dependency Parser）是一种用于分析句子中词语之间依存关系的工具。依存关系描述了句子中各个词语之间的语法和语义联系，它将句子表示为一个有向图，其中节点是词语，边表示词语之间的依存关系。

例如，在句子 “The cat chased the mouse.” 中，“chased” 是核心动词，“The cat” 是动作的执行者（主语），“the mouse” 是动作的承受者（宾语）。依存句法分析会揭示出 “cat” 依赖于 “chased”（表示主语 - 谓语关系），“mouse” 也依赖于 “chased”（表示谓语 - 宾语关系）。

依存句法分析在许多 NLP 任务中都有重要应用，如语义理解、信息提取、机器翻译等。通过分析词语之间的依存关系，能够更准确地理解句子的结构和语义。

Spark NLP 提供了用于依存句法分析的预训练模型 DependencyParserModel。这是一个无标记解析器，可用于查找句子中两个单词之间的语法关系的。依赖解析器提供有关单词关系的信息。例如，依赖关系解析可以告诉我们动词的主语和宾语是什么，以及哪些单词在修饰（描述）主语。这可以帮助我们找到特定问题的精确答案。

可以使用 DependencyParserModel 伴生对象的 pretrained()方法加载，模板代码如下：

```
val dependencyParserApproach = DependencyParserModel.pretrained()
    .setInputCols("sentence", "pos", "token")
    .setOutputCol("dependency")
```

如果没有提供名称，则默认模型为“dependency_conllu”。这个默认模型名称中的 conllu 更多地与数据格式相关。CONLL - U 是一种用于表示句法分析结果的标准格式，它以文本文件形式存储，每行代表一个词元（token），每列包含该词元的不同信息，如词形、词性、依存关系等。下面是一个简单的 CONLL - U 格式示例：

```
1   The   the   DET   DT   Definite=Def|PronType=Art   2   det
2   dog   dog   NOUN  NN   Number=Sing            3   nsubj
3   runs  run   VERB  VBZ  Mood=Ind|Number=Sing|Person=3|Tense=Pres|VerbFor
m=Fin   0   root   _   _
```

在这个示例中，展示了“The dog runs”这句话的依存句法分析结果，每一行对应一个词，各列分别记录了词的编号、词形、词干、词性等信息，同时还标注了词语之间的依存关系（如 det 表示限定词关系，nsubj 表示名词主语关系）。

例如 dependency_conllu 和 dependency_graph 等，。以下分别给出 Python 和 Scala 的示例代码。

执行以上代码，输出内容如下：

11.6.10 命名实体识别

命名实体识别（Named Entity Recognition，简称 NER）是自然语言处理（NLP）中的一项基础且关键的任务，旨在从文本中识别出具有特定意义的命名实体，并将其分类到预先定义好的类别中。

命名实体的常见类别有：

（1）人物（PER）：指真实世界中的个体，例如“李白”、“乔布斯”、“奥黛丽·赫本”等。

（2）地点（LOC）：包括地理区域、城市、国家等，例如“北京”、“珠穆朗玛峰”、“美国”等。

（3）组织机构（ORG）：各类公司、政府机构、社会团体等，例如“苹果公司”、“联合国”、“世界卫生组织”。

（4）时间（TIME）：具体的日期、时间段、年份等，比如、“2024 年”、“周五”、“唐朝”。

（5）货币（MONEY）：涉及各种货币金额，像“100 美元”、“5000 元”。

（6）百分比（PERCENT）：文本中出现的百分比数值，例如“20%”、“5.5%”。

命名实体识别主要应用于如下这些 NLP 任务场景：

（1）信息提取：从大量非结构化文本中精准提取关键信息，帮助快速定位和收集所需的特定信息。例如，在新闻报道中自动提取事件涉及的人物、地点和组织机构等信息，方便构建新闻数据库和进行信息检索。

（2）问答系统：能够准确识别问题和文本中的命名实体，从而更好地理解问题意图，从知识库中找到更精准的答案。比如，当用户询问“阿里巴巴的创始人是谁”时，系统通过 NER 识别出“阿里巴巴”这个组织机构实体，进而准确给出答案。

（3）机器翻译：在翻译过程中，正确识别命名实体可以避免翻译错误，提高翻译质量。例如，对于人名、地名等专有名词，采用固定的译法能使翻译更准确和自然。

（4）文本理解：为计算机理解文本的语义提供重要线索，帮助其更好地把握文本的核心内容和逻辑关系。例如，在分析一篇商业报道时，识别出其中的公司名称、产品名称和人物等实体，有助于理解报道所涉及的商业活动和事件。

Spark NLP 提供了丰富的用于命名实体识别（NER）的预训练模型，这些模型涵盖了多种语言和不同的应用场景。以下介绍一些常用的预训练 NER 模型。

1. NerDLModel

这是一种基于神经网络的通用 NER 模型。它使用 Char cnn - BiLSTM - CRF 神经网络架构进行训练，在大多数数据集上达到了最先进的水平。它可以识别多种常见的命名实体类别，如人物（PER）、地点（LOC）、组织机构（ORG）和杂项（MISC）。该模型适用于一般的英语文本处理任务，如新闻文章、博客等文本的实体识别。

NerDLModel 需要词嵌入作为输入，从而让模型可以学习单词的语义表示。可以使用其伴生对象的 pretrained()方法加载该预训练模型，模板代码如下：

```
val nerModel = NerDLModel.pretrained()
  .setInputCols("sentence", "token", "embeddings")
  .setOutputCol("ner")
```

如果没有提供名称，则默认模型为“ner_dl”。请注意，一些预训练模型需要特定类型的嵌入，这取决于它们被训练的对象。例如，默认的模型“ner_dl”需要 WordEmbeddings “glove_100d”。

以下是一个使用 ner_dl 模型对英语文本进行命名实体识别的示例代码：

```

```

执行以上代码，输出内容如下：

```

```

2. BertForTokenClassification

BertForTokenClassification 可以加载顶部带有 token 分类头的 Bert 模型，用于命名实体识别任务。其输入和输出标签分别是：

（1）输入标签：[document, token]。

（2）输出标签：[ner]

这是一个基于 BERT 架构的命名实体识别模型。BERT（Bidirectional Encoder Representations from Transformers）是一种预训练的语言模型，它采用了 Transformer 架构，能够捕捉文本中的上下文信息，从而在多种自然语言处理任务中取得很好的效果。

可以使用伴生对象的 pretrained()方法加载预训练的模型，模板代码如下：

```
val tokenClassifier = BertForTokenClassification.pretrained()
  .setInputCols("token", "document")
  .setOutputCol("label")
```

如果没有提供名称，则默认模型为“bert_base_token_classifier_conll03”。请看下面的使用示例。

```

```

3. 中文命名实体识别

BertForTokenClassification 还提供了用于中文命名实体识别（NER）的预训练中文模型 product_name_ner_model，改编自 Hugging Face，可用于提取商品名称中的各种属性，如品牌、名称、颜色等，旨在使用 Spark NLP 提供可扩展性和生产准备。

请看下面的示例代码。

```
```

11.7 使用预训练管道

除了定制管道之外，Spark NLP 还提供有大量的预训练管道。这些预训练管道已经根据各种应用场景使用特定的注释器和转换器进行了拟合。

11.7.1 预训练管道介绍

Spark NLP 提供了超过 200 种语言的 6000+以上的预训练管道。下表中列举了其中一些常用的预训练管道。

例如，在 explain_document_dl 这个预训练的管道中，包含有以下 NLP 注释器：

（1）DocumentAssembler

（2）SentenceDetector

（3）Tokenizer

（4）LemmatizerModel

（5）Stemmer

（6）PerceptronModel

（7）ContextSpellCheckerModel

（8）WordEmbeddings (GloVe 6B 100)

（9）NerDLModel

（10）NerConverter (chunking)

所有这些注释器都已经经过 SOTA 算法的训练和调优，随时可用。当用户调用这个管道时，这些注释器会在底层运行，用户会通过这些注释器得到一堆新列。要使用预训练的管道，用户只需要指定管道名称，然后调用 transform()方法。用户还可以自己设计和训练这种管道，然后将其保存到磁盘以供以后使用。

在 Spark NLP 中有一些函数会列出特定语言的所有可用管道。显示可用管道：

```
import com.johnsnowlabs.nlp.pretrained.ResourceDownloader
```

或者，如果我们想检查一个特定的版本：

```
import com.johnsnowlabs.nlp.pretrained.ResourceDownloader
```

在 Spark NLP 库中，使用 PretrainedPipeline 对象代表一个完全构建和训练的 Spark NLP 管道，随时准备使用。这样，整个管道可以在一行中定义。

下面介绍一些 Spark NLP 中常用的预训练管道。

11.7.2 explain_document_ml

Explain Document ML（explain_document_ml）是 Spark NLP 中一个非常实用的预训练管道，主要用于对文本进行全面的自然语言处理（NLP）分析。其中的“ml” 代表机器学习（Machine Learning），该管道运用了多种机器学习技术来完成一系列的 NLP 任务，适用于英文文本的处理。

Explain Document ML 包含以下组件：

（1）文档装配器（Document Assembler）。作为管道的起始组件，它将输入的原始文本转换为 Spark NLP 可以处理的文档格式。在处理文本数据时，首先需要将文本包装成文档对象，以便后续组件进行处理。

（2）分词器（Tokenizer）。将文档中的文本分割成单个的词语（tokens）。分词是 NLP 中的基础步骤，它为后续的词性标注、命名实体识别等任务提供了基本的处理单元。

（3）词性标注器（Part - of - Speech Tagger）。为每个分词后的词语标注其词性，例如名词（Noun）、动词（Verb）、形容词（Adjective）等。词性标注有助于理解词语在句子中的语法角色，对于语义分析和信息提取有重要作用。

（4）命名实体识别器（Named Entity Recognizer）。识别文本中的命名实体，如人物、组织、地点、日期等。通过命名实体识别，可以从文本中提取出有意义的实体信息，用于构建知识图谱、信息检索等应用。

（5）依存句法分析器（Dependency Parser）。分析句子中词语之间的依存关系，将句子表示为一个有向图，其中节点是词语，边表示词语之间的依存关系。依存句法分析有助于理解句子的语法结构和语义关系。

explain_document_ml 可适用于以下场景：

（1）信息提取：从文本中提取命名实体和关键信息，用于构建知识库或进行信息检索。

（2）文本挖掘：通过词性标注和依存句法分析，深入理解文本的语法结构和语义关系，为文本分类、聚类等任务提供支持。

（3）自然语言理解：辅助机器理解文本的含义，在智能客服、问答系统等应用中发挥作用。

下面是 Document ML 管道的演示应用程序（请注意，第一次运行下面的代码可能需要更长的时间，因为它从官方的服务器下载了预训练的管道）：

```
// scala
import com.johnsnowlabs.nlp.pretrained.PretrainedPipeline

val explainDocumentPipeline = PretrainedPipeline("explain_document_ml")
```

执行以上代码，输出内容如下：

正如上面所看到的，explain_document_ml 能够注释任何 document 文档，提供一个词干、检查拼写、引理、词性标记、分词 token 和句子边界检测以及所有这些“开箱即用”的列表作为输出。

11.7.3 explain_document_dl

这是 Spark NLP 提供的一个强大的预训练管道，主要用于对英文文本进行全面且深入的自然语言处理（NLP）分析。这里的 “dl” 代表深度学习（Deep Learning），意味着该管道运用了深度学习技术来完成一系列 NLP 任务，相较于传统机器学习方法，在性能和准确性上往往有更好的表现。

它是一个综合性的管道，可完成多项基本 NLP 任务，包括文档装配、分词、词性标注（POS）、命名实体识别（NER）、依存句法分析和共指消解等。适用于需要对英文文本进行全面分析的场景，如文本挖掘、信息提取等。具体包含的组件和功能如下：

（1）文档装配器（Document Assembler）。作为整个管道的起点，它负责将输入的原始文本转换为 Spark NLP 能够处理的文档格式。这一步是必要的预处理操作，为后续组件的处理奠定基础。

（2）句子分割器（Sentence Detector）。将文档中的文本按句子进行分割，识别出文本中的各个句子边界。这有助于后续对每个句子进行独立的分析和处理。

（3）分词器（Tokenizer）。把每个句子进一步拆分成单个的词语（tokens）。分词是 NLP 中的基础步骤，为后续的词性标注、命名实体识别等任务提供了基本的处理单元。

（4）词性标注器（Part - of - Speech Tagger）。为每个分词后的词语标注其词性，例如名词（Noun）、动词（Verb）、形容词（Adjective）等。词性标注有助于理解词语在句子中的语法角色，对于语义分析和信息提取有重要作用。

（5）命名实体识别器（Named Entity Recognizer）。利用深度学习模型识别文本中的命名实体，如人物、组织、地点、日期等。通过命名实体识别，可以从文本中提取出有意义的实体信息，用于构建知识图谱、信息检索等应用。

（6）共指消解器（Co - reference Resolution）。处理文本中的指代关系，确定代词（如“he”、“she”、“it” 等）所指代的具体实体。共指消解有助于理解文本的连贯性和语义完整性。

（7）依存句法分析器（Dependency Parser）。分析句子中词语之间的依存关系，将句子表示为一个有向图，其中节点是词语，边表示词语之间的依存关系。依存句法分析有助于理解句子的语法结构和语义关系。

explain_document_dl 可适用于以下场景：

（1）信息提取：从文本中提取命名实体和关键信息，用于构建知识库、智能搜索系统等。

（2）文本理解：通过词性标注、共指消解和依存句法分析，深入理解文本的语法结构和语义关系，在智能客服、问答系统等自然语言理解应用中发挥重要作用。

（3）知识图谱构建：利用命名实体识别和共指消解的结果，构建实体之间的关系网络，形成知识图谱。

在 explain_document_dl 上也可以调用 annotate()方法，实现对文本的分析注释任务，与 explain_document_ml 类似。请看下面的代码：

```
import com.johnsnowlabs.nlp.pretrained.PretrainedPipeline

// 示例文本
val text = "Barack Obama was born in Hawaii. He is a former US president."
```

11.7.4 onto_recognize_entities_sm

onto_recognize_entities_sm 是 Spark NLP 库中一个预训练的命名实体识别（Named Entity Recognition, NER）管道。这个管道使用了 OntoNotes 数据集进行训练，专注于英文文本的命名实体识别任务，能够识别英文文本中的多种实体类型，例如人物（PERSON）、组织（ORG）、地点（LOC）等。在信息提取、知识图谱构建等领域有广泛应用。sm 通常表示 “small”，意味着该模型是相对轻量级的版本，适合在资源有限的环境中使用。

请看下面的使用示例。

```

```

11.7.5 将预训练管道用于 DataFrame

还可以将管道与 Spark DataFrame 一起使用。只需要首先创建一个 Spark DataFrame，其中包含一个名为 text 的列，该列将作为管道的输入，然后使用.transform()方法在该 DataFrame 上运行管道，并将不同组件的输出存储在 Spark DataFrame 中。

1. explain_document_ml 用于 DataFrame

下面的示例使用了预定义管道 explain_document_ml，对整个 DataFrame 进行转换。代码如下：

执行以上代码，输出内容如下：

2. explain_document_dl 用于 DataFrame

下面这个示例使用的是 explain_document_dl 预定义管道，过程类型。代码如下：

```
annotation.select("entities.result").show(false)
```

执行以上代码，输出内容如下：

11.7.5 使用 LightPipeline

LightPipeline 是一个 Spark NLP 特定的 Pipeline 类，相当于 Spark ML Pipeline。不同之处在于，它的执行不遵循 Spark 原则，而是在本地（但并行）计算所有内容，以便在处理少量数据时获得快速结果。这意味着，不输入 Spark DataFrame，而是输入一个字符串或字符串数组，以进行注释。

LightPipelines 是 Spark ML 管道转换为单个机器但多线程任务，对于较小的数据量（小是相对的，但上限大致是 5 万个句子），速度要快 10 倍以上。要使用它们，只需插入一个已经训练（拟合）过的 Spark ML 管道。它的 transform()阶段被转换成 annotate()来代替。

请看下面这个示例。

执行以上代码，输出内容如下：

11.7.6 使用 RecursivePipeline

RecursivePipeline 是 Spark NLP 库中的一个组件，它继承自 Spark ML 的 Pipeline。RecursivePipeline 允许在管道中嵌套子管道，这使得用户能够构建更为复杂的自然语言处理（NLP）工作流。借助 RecursivePipeline，用户可以把多个 NLP 组件组合成子管道，然后将这些子管道集成到主管道里，从而实现对文本数据的逐步处理。

RecursivePipeline 的行为与普通的 Spark ML 管道完全相同，因此它们可以用于相同的目的。

例子:

11.8 案例：影评数据情感分析

IMDB（Internet Movie Database）即互联网电影数据库，是全球知名且规模庞大的电影、电视节目等相关信息数据库网站。IMDB 成立于 1990 年，最初是作为一个 Usenet 新闻组出现，供电影爱好者交流信息。后来逐渐发展成为一个商业网站，并在 1998 年被亚马逊收购。如今，它已经成为电影行业和影迷们不可或缺的信息来源。

IMDB 涵盖了海量的影视信息，包括电影、电视剧、纪录片、短片等。截至 2023 年，它拥有超过 1000 万个影视标题和 8000 多万个人物资料。用户可以在上面找到每部作品的详细信息，如剧情简介、演职员表、上映日期、制作团队、获奖情况等。同时 IMDB 也拥有庞大的用户群体，每月有数十亿次的页面访问量。全球各地的电影爱好者都在 IMDB 上分享自己的观点、评分和评论。用户可以注册账号，对影片进行评分（1 - 10 分）、撰写影评、收藏喜欢的作品、关注影人动态等。

11.8.1 背景和任务目的

IMDB 积累了数以亿计的电影评论。既有普通用户的主观评价，也有专业影评人的深入分析。普通用户的评论通常比较简短、直接，表达了他们对电影的直观感受，如喜欢或不喜欢的原因、影片的亮点和不足等。专业影评人的评论则更具专业性和深度，会从电影的艺术价值、技术层面、社会意义等多个角度进行剖析。这些评论来自全球不同地区、不同背景的用户，涵盖了各种类型和年代的电影。对于热门电影，往往有成千上万条评论，为分析电影的受众反应提供了丰富的数据基础。

对于电影制作方来说，可以通过分析这些评论了解观众的喜好和需求，为电影的制作、宣传和发行提供参考。例如，如果某类题材的电影在评论中受到广泛好评，制作方可能会考虑增加此类题材的投入。

对于研究人员来说，利用 IMDB 电影评论数据进行学术研究，如分析电影的文化传播、社会影响、观众心理等。也可以用于自然语言处理领域的研究，如情感分析、文本分类等。

对于观众来说，则可以在选择观看电影时，参考其他用户的评论来判断电影是否符合自己的口味。

本案例的目的，就是使用 IMDB 网站的 5 万条电影评论数据，提取用户评论信息，将这些电影评论分为两类：正面意见和负面意见。这是一个 NLP 任务，也是一个情感分类任务，我们可以使用 Spark NLP 库来完成。

11.8.2 了解影评数据

这是一个二元情绪分类的数据集，包含比以前的基准数据集多得多的数据。其中提供 25000 条电影评论用于训练，25000 条用于测试。还有额外的未标记数据可供使用。并提供了原始文本和已处理的字包格式。

11.8.3 准备影评数据

由于我们使用 Spark 集群来进行计算，所以需要把待处理的文本数据上传到 HDFS 共享存储中去。请按以下步骤操作。

11.8.4 选择预训练模型

接下来，我们要考虑在 Spark NLP 库中选择合适的组件来完成任务。基本上，我们有两种方式来完成该任务：

（1）逐个创建 Annatation 组件，最后组装成一个 Pileline 管道。

（2）加载预训练好的模型或管道。

在 Spark NLP 库中已经提供了用于多类情感分析的注释器 SentimentDL，其实例模型是 SentimentDLModel，默认模型为“sentimentdl_use_imdb”，这恰好是在 IMDB 数据集上训练的英语情感分析模型，这正是本案例中需要的。

可通过 SentimentDLModel 伴生对象的 pretrained()方法加载这个预训练模型，代码如下：

```
val sentiment = SentimentDLModel.pretrained()
  .setInputCols("sentence_embeddings")
  .setOutputCol("sentiment")
```

该模型是基于 IMDB（Internet Movie Database）影评数据集进行训练的。IMDB 数据集包含大量的电影评论，这些评论有积极和消极的情感倾向，因此这个模型在电影评论的情感分析任务上表现良好。

该模型使用了 Universal Sentence Encoder 作为特征提取器。USE 可以将输入的文本转换为固定长度的向量表示，这个向量能够捕捉到文本的语义信息。在提取文本特征后，模型使用深度学习分类器来判断文本的情感极性，即判断评论是积极的还是消极的。该模型主要用于对电影评论、影评文章等文本进行情感分析，判断其表达的是积极情感还是消极情感。

以下是加载预训练模型 sentimentdl_use_imdb 并对上一节数据进行分析的代码：

执行以上代码，输出内容如下：

11.8.5 保存分析结果

接下来，我们需要将以上分析的结果，持久保存下来。同样地，将所有情感分析结果为 positive 的评论数据存储到 HDFS 的/data/spark-nlp/unsup-pos 目录下，将所有情感分析结果为 positive 的评论数据存储到 HDFS 的/data/spark-nlp/unsup-neg 目录下。实现代码如下：

11.8.6 完整示例代码

以下是完整的代码示例：

11.8.7 换一种模型

ViveknSentimentModel 是 Spark NLP 里的另一个预训练情感分析模型。它以 Vivek Narayanan 提出的基于词性标注（POS）和启发式规则的情感分析算法为基础构建。该模型可用于判断文本所表达的情感极性，如积极、消极或中性。

ViveknSentimentModel 主要应用到了以下几种技术：

（1）词性标注（POS）。模型会先对输入文本进行词性标注，明确每个词语在句子里的语法角色，像名词、动词、形容词等。词性信息对理解词语在句子中的语义和情感倾向十分关键。

（2）启发式规则。利用一系列预先定义好的启发式规则来确定文本的情感。这些规则综合考虑了词性、词语的情感极性以及词语间的依赖关系等因素。例如，某些形容词往往带有明显的积极或消极情感，模型会根据这些特征来判断整个句子的情感。

（3）机器学习方法。在规则的基础上，可能还运用了机器学习技术来优化情感分析的结果。通过对大量标注数据的学习，模型能够不断调整规则和参数，以提高情感分析的准确性。

ViveknSentimentModel 具有以下几个特点：

（1）轻量级：相较于一些基于深度学习的情感分析模型，ViveknSentimentModel 计算资源需求较低，运行速度较快。这使得它在资源受限的环境或者对实时性要求较高的场景中具有一定优势。

（2）无需大量训练数据：由于其基于规则和词性标注，在没有大量标注数据的情况下也能进行有效的情感分析。这对于一些特定领域或者数据量较少的任务来说是一个重要的优点。

（3）可解释性强：基于规则的方法使得模型的决策过程具有较高的可解释性。用户可以理解模型是如何根据文本的词性和规则来判断情感的，这在一些对结果解释要求较高的应用场景中非常有用。

VivеknSentimentModel 通常应用在以下场景中：

（1）社交媒体情感分析：可用于分析社交媒体上的用户评论、帖子等内容的情感倾向，帮助企业了解用户对产品、服务或事件的看法。

（2）客户反馈分析：对客户的反馈意见、评价等文本进行情感分析，以便企业及时发现客户的满意度和潜在问题。

（3）舆情监测：监测新闻、论坛等渠道的文本信息，了解公众对特定话题的情感态度，为政府、企业等提供决策参考。

在前面的实现方案中，我们选择使用的是预训练模型 sentimentdl_use_imdb，它使用深度学习分类器来判断文本的情感极性，相对来说对计算资源的需求比较高，运行速度较慢。现在我们尝试改用 VivеknSentimentModel 模型来重构代码。重构后的代码如下：

第 12 章 基于 Spark 的分布式深度学习

传统的机器学习算法，依赖人工特征工程，难以处理图像、文本等高维度复杂特征的非结构化数据。而近些年深度学习的崛起，突破了这种限制。深度学习通过堆叠多个非线性变换层进行多层非线性变换，从原始数据中逐层学习抽象特征，从而实现自动特征提取，而无需人工设计特征，输入原始数据即可输出最终结果。

传统单机深度学习无法处理超大规模数据（如 TB 级用户日志）。将深度学习与 Spark 相结合，可以充分利用 Spark 的分布式计算能力，将数据拆分到多个节点，加速训练。并且利用 Spark 集群的容错能力，自动处理节点故障，保障长时间任务稳定性。

表 12-1 传统机器学习 vs. 深度学习

维度	传统机器学习	深度学习
特征工程	依赖人工设计和选择特征	自动学习多层次特征
数据需求	小数据即可有效	需要海量数据（百万级样本以上）
可解释性	模型简单，易于解释（如决策树）	黑箱模型，解释性差
计算资源	CPU 即可训练	需要 GPU/TPU 加速
适用场景	结构化数据（表格数据）	非结构化数据（图像、文本、语音）

12.1 深度学习基础

深度学习是人工智能（AI）的一个分支，它使机器能够从大量数据中学习。它使用多层神经网络来自动发现模式并进行预测。它对于图像识别、语言翻译和语音处理等任务非常有用。深度学习在包括自动驾驶汽车、聊天机器人、医学图像分析和推荐系统等领域的应用非常流行。

12.1.1 什么是深度学习？

深度学习（Deep Learning）是机器学习的一个子领域，特指基于深层神经网络模型和方法的机器学习。其核心是通过多层次非线性变换（即“深度”网络）自动从数据中学习特征表示，并完成分类、回归、生成等任务。与传统机器学习不同，深度学习无需人工设计特征，而是通过模型自身逐层提取从低级到高级的抽象特征。

深度学习最重要的技术特点是具有自动提取特征的能力，深度学习模型直接从数据中学习，不需要手动提取特征。其核心思想是通过多层非线性变换自动提取数据特征，所提取的特征也称为深度特征或深度特征表示。相比于人工设计的特征，深度特征的表示能力更强、更稳健。因此，深度学习的本质是特征表征学习。

深度学习的实质是构建具有多个隐藏层的机器学习模型，通过海量的训练数据来学习更有用的特征，从而最终提升分类或预测的准确性。在此过程中，各层学习模型的最优特征，其优点是特征不需要预先确定。然而，这有一个缺点，即模型的决策是不可解释的。因为解释决策可能很重要，研究人员正在开发新的方法来理解深度学习的黑箱。

12.1.2 深度学习核心概念

神经网络是模仿人脑复杂功能的机器学习模型。这些模型由相互连接的节点或神经元组成，这些节点或神经元处理数据、学习模式，并实现模式识别和决策等任务。

1. 多层神经网络

深度学习是多层神经网络（Multi-Layer Neural Network）的通俗名称，多层神经网络是由输入节点和输出节点之间的几个“隐藏层”组成的网络，其结构可以用下图表示：

通常，人工神经网络有输入层、输出层和隐藏层。输入层接收来自外部世界的数据，神经网络需要对这些数据进行分析或学习。然后该数据通过一个或多个隐藏层，这些隐藏层将输入转换为对输出层有价值的数据。最后，输出层以人工神经网络对所提供的输入数据的响应的形式提供输出。

在梯度下降的过程中进行反向传播，错误再次通过网络发送回来，并调整权重，从而改进模型。这个过程重复数千次，根据模型产生的误差调整模型的权重，直到误差无法再减小为止。

在大多数神经网络中，单元从一层连接到另一层。这些连接中的每一个都有权重，这些权重决定了一个单元对另一个单元的影响，这意味着通过为每个输入分配不同的权重来或多或少地优化前一层输入的效果，并在训练过程中通过优化这些权重来调整效果，以提高模型性能。当数据从一个单元传输到另一个单元时，神经网络对数据的学习越来越多，最终得到输出层的输出。

2. 多层感知器（MLP）

多层神经网络中最基础的形式是多层感知器（Multi-Layer Perceptron, MLP）。MLP 是多层神经网络的一种具体实现形式，特指由全连接层（Dense Layer）构成的简单前馈网络。其结构特点是，每一层的神经元与下一层的所有神经元全连接，无跨层或反向连接。其结构表示如下图：

多层感知器（MLP）关键组件说明如下：

（1）输入层（Input Layer）：这一层的每个神经元（或节点）对应一个输入特征。例如，如果有三个输入特征，那么输入层将有三个神经元。

（2）隐藏层（Hidden Layers）：一个 MLP 可以有任意数量的隐藏层，每一层包含任意数量的节点。这些层处理从输入层接收到的信息。

（3）输出层（Output Layer）：输出层生成最终的预测或结果。如果有多个输出，则输出层将具有相应数量的神经元。

图中的每个连接都是 MLP 的完全连接性质的表示。这意味着一层中的每个节点都连接到下一层的每个节点。当数据在网络中移动时，每一层都会对其进行转换，直到在输出层中生成最终输出。

MLP 由完全连接的密集层组成，这些层将输入数据从一个维度转换到另一个维度。它被称为“多层”，因为它包含一个输入层、一个或多个隐藏层和一个输出层。MLP 的目的是为输入和输出之间的复杂关系建模，使其成为各种机器学习任务的强大工具。

3. 前向传播

在前向传播（forward propagation）中，数据从输入层流向输出层，经过任何隐藏层。隐藏层中的每个神经元对输入的处理如下：

（1）加权和。神经元计算输入的加权和：层中的每个神经元接收输入，这些输入乘以与连接相关的权重。将这些乘积加在一起，并在总和中加上一个偏差。公式如下：

$$z = \sum_i w_i x_i + b$$

其中，x_i 是输入特征，w_i 是相应的权重，b 是偏置项。

（2）激活函数。加权和 z 通过激活函数传递以引入非线性。在计算出加权和后，然后将线性变换的结果传递给激活函数。激活函数是至关重要的，因为它将非线性引入系统，使网络能够学习更复杂的模式。常用的激活函数包括：Sigmoid、ReLU（修正线性单元）和 Tanh（双曲正切函数）。

4. 损失函数

一旦网络产生了输出，下一步就是使用损失函数来计算损失。在监督学习中，将预测输出与实际标签进行比较。对于分类问题，常用的二元交叉熵损失函数为：

$$\mathrm{L} = -\frac{1}{\mathrm{N}}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1-y)\log(1-\hat{y}_i)\right]$$

其中，y_i 是实际标签，$\hat{y}_i$ 是预测标签，N 是样本数量。

对于回归问题，常用均方误差（mean squared error，MSE）来计算损失。MSE 的计算公式为：

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

5. 反向传播

训练 MLP 的目标是通过调整网络的权重和偏置来最小化损失函数（即最小化错误或误差）。这是通过反向传播（backpropagation）实现的：

（1）梯度计算：使用微积分的链式法则计算损失函数相对于每个权重和偏差的梯度。

（2）错误传播：错误通过网络一层一层地传播回来。

（3）梯度下降：网络通过与梯度相反的方向移动来更新权重和偏差，以减少损失。新的权重计算公式为：

$$w = w - \eta \times \frac{\partial L}{\partial w}$$

其中，w 是权重，η 是学习率，$\frac{\partial L}{\partial w}$ 是损失函数对权值的梯度。

6. 最优化

MLP 依靠优化算法在训练过程中迭代地细化权重和偏差。常用的优化方法包括：

（1）随机梯度下降（Stochastic Gradient Descent，SGD）：基于单个样本或小批量数据更新权重： $w = w - \eta \times \frac{\partial L}{\partial w}$。

（2）Adam 优化器（Adam Optimizer）：SGD 的扩展，结合了动量和自适应学习率，以实现更有效的训练。

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) \cdot g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) \cdot g_t^2$$

这里，g_t 表示时间 t 的梯度，$\beta_1 \beta_2$ 是衰减率。

12.2 深度学习算法

MLP 有着严格的全连接结构，参数数量随层数指数增长（参数量=输入维度×输出维度）。例如，输入层 1000 维，隐藏层 500 维，那么参数量为 1000×500+500=500,500。因此，MLP 的训练速度可能很慢，特别是在具有许多层的大型数据集上。如果没有适当的正则化技术，MLP 还可能会过拟合训练数据，导致较差的泛化。

另外，MLP 处理结构化数据（如表格数据）时表现良好，但对高维非结构化数据（如图像）效率低下。因此，MLP 往往需要适当规范化或缩放数据以获得最佳性能。

所以，在 MLP 基础上，深度学习算法演变出不同的变体，可用于构建不同场景下数据驱动的应用程序。这些多层神经网络通过特定连接方式减少参数量或捕捉特定模式，通过稀疏连接降低复杂度，并针对特定数据类型优化。因此，多层神经网络广义上指包含多个层（输入层、隐藏层、输出层）的任何神经网络结构，是一个总称。其核心特征是，通过堆叠非线性变换层，从数据中学习复杂的特征表示。常见的其他深度学习算法包括：

1. 用于改进传统算法的深度神经网络（DNN）

（1）金融：通过识别更复杂的模式加强欺诈检测。

（2）制造：基于更深层次的异常检测，增强缺陷识别。

2. 用于图像处理的卷积神经网络（CNN）

（1）零售：店内活动分析视频来衡量流量。

（2）卫星图像：标记地形，分类物体。

（3）自动驾驶：识别道路和障碍物。

（4）医疗保健：x 光、扫描等的诊断机会。

（5）保险：根据照片估计索赔严重程度。

3. 用于序列数据的递归神经网络（RNN）

（1）客户满意度：将语音数据转录为文本进行 NLP 分析。

（2）社交媒体：社交和产品论坛帖子的实时翻译。

（3）图片说明：搜索图像档案以获得新的见解。

（4）金融：基于时间序列分析预测行为（也是增强的推荐系统）。

其核心特征是，通过堆叠非线性变换层，从数据中学习复杂的特征表示。

12.2.1 深度神经网络（DNN）

多层神经网络（MLP）层数较少（通常 2-4 层），结构简单。在 2000 年代之前，多层神经网络因以下问题难以训练深层结构，实际应用多限于浅层结构：

（1）梯度消失/爆炸：误差反向传播时，梯度随层数指数级衰减或激增。

（2）过拟合：参数过多时，模型易记忆训练数据而泛化能力差。

在 2010 年后，以下现代训练技术的出现解决了深层网络的训练难题：

（1）ReLU 激活函数：相比 Sigmoid/Tanh 激活函数，ReLU 可缓解梯度消失问题。

（2）批量归一化（Batch Normalization）：加速训练并稳定梯度。

（3）残差连接（ResNet）：通过跨层跳跃传递梯度，允许训练数百层的网络。

（4）Dropout：随机屏蔽神经元，防止过拟合。

深度神经网络（Deep Neural Network，DNN）泛指具有多个隐藏层的神经网络（通常≥5 层），通过堆叠全连接层和非线性激活函数实现复杂映射，提升模型表达能力。

在 DNN 中，数据从输入层开始，经过隐藏层的逐层计算，最终到达输出层。每一层神经元的输出都作为下一层神经元的输入，通过激活函数实现非线性变换。DNN 的训练过程依赖于反向传播算法和梯度下降算法，通过计算输出层与真实标签之间的误差，并将误差反向传播到每一层神经元，更新神经元的权重和偏置项，以最小化预测误差。

深度神经网络层数多（几十至上千层），模型复杂度高，在图像、语音、NLP 等领域远超传统多层网络。

表 12-1 MLP 与 DNN 的区别

维度	多层神经网络（MLP）	深度神经网络（DNN）
层数	通常≤4 层	通常≥5 层
训练技术	依赖基础反向传播，无特殊优化	需残差连接、批量归一化、Dropout 等稳定训练
应用场景	简单分类/回归任务（如表格数据）	复杂任务（图像识别、自然语言处理、语音合成）
计算资源	CPU 或低配 GPU 即可训练	需高性能 GPU/TPU 支持
典型模型	传统 MLP、浅层 CNN	ResNet、Transformer、BERT

12.2.2 卷积神经网络（CNN）

卷积神经网络（Convolutional Neural Networks, CNN）是一类包含卷积计算且具有深度结构的前馈神经网络，是深度学习的代表算法之一。卷积神经网络（CNN）是专门处理网格结构数据（如图像、视频、音频频谱图）的神经网络，通过卷积层（Convolutional Layer）和池化层（Pooling Layer）高效提取局部空间特征。

CNN 的基本架构如下图所示，我们使用图像作为输入。执行一系列卷积（Conv）操作和池化（Pool）操作，然后是一些全连接层（FC）。 如果执行多类分类则输出为 softmax 激活函数。

CNN 的基本结构包括：卷积层、激活函数、池化层和全连接层和输出层。

（1）输入层：保留原始空间结构（如 28×28 的二维图像）。

（2）卷积层：通过滑动卷积核提取局部特征（如边缘检测）。卷积神经网路中每层卷积层由若干卷积单元组成，每个卷积单元的参数都是通过反向传播算法优化得到的。卷积运算的目的是提取输入的不同特征，第一层卷积层可能只能提取一些低级的特征如边缘、线条和角等层级，更多层的网络能从低级特征中迭代提取更复杂的特征。

（3）池化层：降低特征图维度（如最大池化保留显著特征）。通常在卷积层之后会得到维度很大的特征，池化层将特征切成几个区域，取其最大值或平均值，得到新的、维度较小的特征。

（4）全连接层：末端用于分类或回归。 把所有局部特征结合变成全局特征，用来计算最后每一类的得分。

（5）激活函数。即线性整流层，这一层神经的激活函数使用线性整流（ReLU）。

CNN 结构的优势如下：

（1）参数共享：同一卷积核在整个输入上滑动（即同一卷积核在不同位置重复使用），共享权重（如检测“边缘”的卷积核在所有位置生效），从而减少参数量。例如 5×5 卷积核仅需 25 个参数）。

（2）局部连接：也称为稀疏连接，即每个神经元仅连接输入区域的局部窗口（如 3×3 区域），而非全连接，减少计算量。例如，输入图像为 100×100，卷积核为 5×5，则每个输出神经元仅连接输入的 5×5 区域，而非全连接（10000 参数）。

（3）平移不变性。卷积核的滑动特性使模型对目标的平移（如左移、右移）不敏感。例如，在图像检测中，无论“猫”出现在图像的左上角还是右下角，CNN 均可识别。

（4）特征提取能力。CNN 通过卷积层逐级提取层次化特征：低级特征（边缘、角点）→ 中级特征（形状、纹理） → 高级特征（物体部件）。适合捕捉局部相关性（如图像中相邻像素的关系）。

CNN 适用于图像处理（图像分类，目标检测，图像分割）、图像分析（动作识别）、语音识别（声谱图分类）等应用场景。若输入是图像、视频或语音频谱，必选 CNN。

表 12-1 DNN 与 CNN 的区别

维度	深度神经网络（DNN）	卷积神经网络（CNN）
网络结构	全连接层堆叠	卷积层+ 池化层
参数数量	参数量大（输入维度×输出维度）	参数量少（卷积核共享参数）
数据适应性	适合结构化数据（表格、特征向量）	适合非结构化数据（图像、语音、视频）
特征提取方式	依赖全局特征组合	自动提取局部空间特征（如边缘、纹理）
平移不变性	无	通过卷积核滑动实现（对输入位置变化不敏感）

12.2.3 循环神经网络（RNN）

循环神经网络（Recurrent Neural Networks，RNNs）是用于建模序列数据，如时间序列或自然语言的一类神经网络，其核心特点是利用循环结构捕捉序列中的时序依赖关系。

循环神经网络（RNNs）的工作原理与常规神经网络略有不同。在神经网络中，信息从输入到输出是单向流动的。然而，在 RNN 中，信息在每一步之后都被反馈到系统中。把它

想象成读一个句子，当你试图预测下一个单词时，你不仅要看当前的单词，还需要记住之前出现的单词来做出准确的猜测。

RNNs 允许神经网络通过将一步的输出输入到下一步来“记住”过去的信息。这有助于神经网络理解已经发生的事情的背景，并在此基础上做出更好的预测。例如，当预测句子中的下一个单词时，RNN 使用前面的单词来帮助确定下一个单词最有可能出现。

下面的图片展示了 RNN 的基本架构和反馈循环机制，其中输出作为下一个时间步骤的输入传递回来。

RNN 的基本处理单元是循环单元（Recurrent Unit）。循环单元保持一种隐藏状态，保持序列中先前输入的信息。循环单元可以通过反馈其隐藏状态来“记住”先前步骤中的信息，从而允许它们捕获跨越时间的依赖关系。RNN 循环单元如下图所示（其中 X 代表输入，h 代表隐藏状态，v、u 和 w 是权重矩阵，L 代表损失函数）：

RNN 展开是在时间步长上扩展循环结构的过程。在展开过程中，序列的每个步骤都表示为一个系列中的单独层，说明信息如何在每个时间步骤中流动。这种展开实现了时间反向传播（BackPropagation Through Time，BPTT），这是一种学习过程，其中错误在时间步长上传播，以调整网络的权重，增强 RNN 学习序列数据中依赖关系的能力。RNN 展开过程如下图所示：

RNNs 与其他深度学习架构在输入和输出结构上有相似之处，但在信息从输入流向输出的方式上有很大不同。与传统的深度神经网络不同，在传统的深度神经网络中，每个连接层都有不同的权重矩阵，而 RNNs 在时间步长上使用共享的权重，从而允许它们记住序列上的信息。在每个时间步（time step），RNNs 处理具有固定激活函数的单元。这些单元具有内部隐藏状态，充当存储器，保留以前时间步骤的信息。这种记忆使神经网络能够存储过去的知识，并根据新的输入进行调整。

基于网络中输入和输出的数量，有四种类型的 RNNs。分别介绍如下。

1）One-to-One RNN

这是最简单的神经网络结构，只有一个输入和一个输出。它用于直接的分类任务，如不涉及顺序数据的二进制分类。

2）One-to-Many RNN

在 One-to-Many（一对多）RNN 中，随着时间的推移，神经网络处理单个输入以产生多个输出。这在一个输入触发一系列预测（输出）的任务中很有用。例如，在图像标题中，可以使用单个图像作为输入来生成作为标题的单词序列。

3）Many-to-One RNN

Many-to-One（多对一）RNN 接收一系列输入并生成单个输出。当需要输入序列的整个上下文来进行预测时，这种类型非常有用。在情感分析中，模型接收一个单词序列（比如一个句子），并产生一个单一的输出，比如积极、消极或中性。

4）Many-to-Many RNN

Many-to-Many（多对多）RNN 类型处理一系列输入并生成一系列输出。在语言翻译任务中，将一种语言的单词序列作为输入，并生成另一种语言的相应序列作为输出。

RNNs 的典型应用场景包括：

（1）自然语言处理（NLP）：文本生成、机器翻译。

（2）时间序列预测：股票价格预测、天气预测。

（3）语音识别：将音频信号转换为文本。

表 12-中列出了 RNNs 与 MLP 和 CNN 的区别。

表 12-3 消息格式

维度	RNN	CNN	MLP
网络结构	循环结构，时间步间共享权重，按时间维度展开	卷积层+池化层，空间维度局部连接	全连接层堆叠，无时序连接
数据处理	处理序列数据（如文本、时间序列）	处理空间数据（如图像、网格化数据）	处理静态数据（如表格、特征向量）
参数共享	时间步间共享参数（减少参数量）	空间位置间共享卷积核	每层独立参数（参数量大）
依赖关系建模	捕捉时间依赖（如句子中的上下文）	捕捉空间依赖（如图像的局部特征）	无法建模时序关系
应用场景	机器翻译、文本生成、语音识别	图像分类、目标检测	房价预测、用户分类

12.3 深度学习框架

若手动实现深度学习算法，难度比较大。例如，要实现卷积神经网络（CNN）的反向传播，需编写数百行代码计算梯度，且难以优化性能。幸运的是，业界开源了很多深度学习框架，框架将这些复杂操作封装为几行代码，直接调用即可，从而避免重复造轮子。

业界推出的深度学习框架非常多，并且还在不断迭代发展。下面列出目前几个最流行的深度学习框架：

1）Caffee

该框架发布于 2013 年，由加州大学伯克利分校的贾扬清主导开发，使用 C++语言编写。Caffe 主要在图像数据上构建深度学习模型，但在循环神经网络和语言模型上落后于其他深度学习框架，并且扩展性差，对新算法（如 Transformer）支持不足，社区活跃度下降。Caffe 主要用于建立和部署移动电话和其他计算受限平台的深度学习模型。

2）Deeplearning4j

该框架发布于 2014 年，使用 Java、Scala 语言编写，由 Skymind 开发并维护，现为 Eclipse 基金会项目。Deeplearning4j 是面向企业级应用的分布式深度学习框架，强调生产环境部署和与 Java 生态的无缝集成，适合企业级 Java/Scala 项目。Deeplearning4j 覆盖主流深度学习模型，包括 CNN、RNN、LSTM、深度自动编码器、受限玻尔兹曼机（RBM）等，支持监督、无监督和强化学习。

3）TensorFlow

TensorFlow 是一个开源的机器学习框架，使用 C++、Python 语言编写，由 Google Brain 团队开发并在 2015 年发布。它提供了灵活的计算模型，支持大规模分布式训练和跨平台部署，适用于从研究到生产的各种场景。广泛用于学术研究和生产环境，尤其在图像识别、自然语言处理领域。缺点是早期版本接口复杂，代码臃肿（接近百万行），调试难度较高。

4）Keras

该框架作为独立库发布于 2015 年，使用 Python 语言编写。目前 Keras 集成在 TensorFlow 中，支持多后端（TensorFlow、Theano、CNTK），现为 TensorFlow 官方高级接口。适合快速原型开发，但灵活性和底层控制较弱。

5）PyTorch

PyTorch 框架由 Facebook 开发并在 2017 年发布，使用 Python、C 语言编写。它采用动态计算图设计，灵活支持实时调试，适合研究场景。PyTorch 高度模块化，与 Python 生态无缝集成（如 NumPy、SciPy），支持分布式训练和移动端部署（TorchScript）。PyTorch 社区活跃，在学术界占据主导地位，尤其在自然语言处理（如 Transformer）和生成模型（如 GAN）中广泛应用。

6）百度飞桨（PaddlePaddle）

百度飞桨（PaddlePaddle）的开发语言以 Python 为核心，同时支持 C++作为底层扩展和高性能计算的补充。该框架是国内首个功能完备的开源深度学习平台，支持全流程开发（训练、推理、部署）。由百度公司在 2016 年发布，针对中文场景优化，内置丰富的预训练模型（如文心大模型）。行业解决方案覆盖医疗、金融、制造业等，国产化替代趋势显著。

表 12-中列出了框架所提供的核心功能以及支持的算法：

表 12-1 深度学习框架的核心功能和支持算法

框架功能	对算法的支持
自动求导	自动计算梯度，无需手动推导反向传播公式。
硬件加速	自动调用 GPU/TPU 加速矩阵运算。
预定义层与损失函数	提供现成的网络层（如全连接层、卷积层）和损失函数（如交叉熵）。
分布式训练	支持多 GPU 或多节点并行训练。

开源的深度学习框架不仅提供了对深度学习算法的自动求导、GPU 加速、预训练模型的支持，并且还提供高层 API 简化模型构建，极大地提高了开发效率。

12.4 Spark 与深度学习结合

从线性回归到深度学习，本质是模型从简单线性拟合向复杂非线性表示的进化，核心驱动力是数据、算力和算法的协同突破。深度学习的核心优势在于自动特征提取和端到端学习，通常需要大量数据训练。而 Spark 本身是处理大数据的利器，能够分布式处理 PB 级数据。Spark 的分布式计算框架可以扩展深度学习训练，比如数据并行，把数据分片到多个节点，加速训练过程。另外，Spark 的生态系统丰富，比如 Spark SQL、MLlib，能够和深度学习流程无缝衔接，减少数据在不同系统间的迁移成本。

目前很多企业已经有 Spark 集群，如果可以直接在上面跑深度学习，不需要额外搭建 GPU 集群，节省成本。特别是针对 CPU 优化过的框架（如 BigDL），可能更适合现有基础

设施。另外，容错性和易用性也是优势，Spark 本身有容错机制，训练过程中节点失败可以恢复，而使用 Spark 的 API 可以让熟悉 Spark 的开发者更容易上手深度学习，不需要学习新工具。

分布式计算和机器学习的结合正在加速下一代智能应用程序的发展，这一演变历程为现代 AI 应用（如自动驾驶、智能推荐）奠定了技术基础。

并不是所有的深度学习框架都可以直接在 Spark 集群上运行。可以运行在 Spark 上的深度学习框架目前主要包括以下几种：

（1）BigDL：BigDL 是由英特尔公司开发的，是专门为 Spark 设计的深度学习库，可以与 Spark 生态系统无缝集成，用户可以直接在 Spark 应用中使用深度学习功能，无需额外的复杂配置，适合 CPU 集群。BigDL 能充分利用 Spark 的分布式计算能力，在大规模数据上进行高效的深度学习训练和推理。并且可以支持 TensorFlow、PyTorch 等多种深度学习框架的模型，方便用户在不同框架之间进行切换和使用。但是，相比于原生的深度学习框架，BigDL 的一些高级功能可能不够完善，在某些特定场景下可能无法满足用户的需求。另外，与 TensorFlow 和 PyTorch 等主流框架相比，BigDL 的社区活跃度相对较低，用户在遇到问题时可能较难找到相关的解决方案。

（2）TensorFlow On Spark：TensorFlow 本身是一个功能强大且广泛使用的深度学习框架，而 TensorFlow On Spark 并非由某一个特定的公司单独开发，它是社区合作的成果。TensorFlow on Spark 允许用户在 Spark 集群上利用 TensorFlow 的丰富功能，如支持多种深度学习模型（CNN、RNN 等），并且可以使用 TensorFlow 的高级 API 进行快速模型开发。它可以在 Spark 集群上实现分布式训练，利用集群的计算资源加速训练过程，提高训练效率。得益于 TensorFlow 庞大的生态系统，用户可以方便地使用各种工具和库进行模型评估、可视化等操作。但是，在 Spark 集群上配置和运行 TensorFlow 需要一定的技术门槛，涉及到多个组件的协调和配置，可能会遇到各种兼容性问题。另外，在分布式环境下，对资源的管理和调度相对复杂，需要合理分配计算资源和内存资源，否则可能会导致性能下降。

（3）Deeplearning4j：是由 Skymind 公司（现在的 Konduit.ai）推出的。这是一个专门为 Java 和 Scala 设计的开源深度学习库，对于 Java 开发者来说，Deeplearning4j 是一个很好的选择，它提供了 Java API，方便 Java 开发者在 Spark 集群上进行深度学习开发。它支持在 Spark 集群上进行分布式训练，能够有效利用集群的计算资源。可扩展性强，可以与其他 Java 生态系统的工具和库进行集成，方便进行系统的扩展和开发。但是，Deeplearning4j 的学习曲线较陡，对于没有深度学习基础的 Java 开发者来说，Deeplearning4j 的学习曲线相对较陡，需要花费一定的时间来学习和掌握。另外，相比于 TensorFlow 和 PyTorch 等主流框架，Deeplearning4j 的社区规模较小，资源和支持相对有限。

（4）PyTorch with Horovod and Spark：PyTorch 由 Facebook 的人工智能研究团队（FAIR）开发的一个开源深度学习框架。PyTorch 以其简洁易用的 API 和动态图机制受到广泛欢迎，用户可以更方便地进行模型开发和调试。Horovod 是由 Uber 开发的一个用于分布式深度学习训练的框架，它能够在多种深度学习框架（包括 PyTorch）上实现高效的分布式训练。将 PyTorch、Horovod 与 Spark 结合，这也是社区中开发者们探索出的一种组合方案，利用 Spark 的分布式计算能力、Horovod 的分布式训练优化以及 PyTorch 的强大深度学习功能，在 Spark 集群上实现分布式的深度学习训练任务。但是，虽然 Horovod 简化了分布式训练的

过程，但在 Spark 集群上部署和管理仍然需要一定的技术经验，尤其是在处理大规模数据和复杂模型时。并且，在分布式环境下，要充分发挥 PyTorch 的性能优势，需要进行精细的性能优化，如数据加载、通信优化等。

在很多情况下，选用哪种框架无关紧要，我们会发现每个框架可用的模型基本相同。只有在某些特定的情况下，可能某个框架会优于另一个。考虑到我们是在讲基于 Spark 的机器学习和深度学习技术，所以本书选择由英特尔公司专门为 Spark 设计开发的深度学习库 BigDL，这将使得我们基于已有 Spark/Hadoop 集群，低成本地扩展深度学习能力。

12.5 BigDL on Spark

BigDL 是 Intel 开发的基于 Apache Spark 的分布式深度学习框架。它可以直接在现有的 Spark 集群上运行，适合处理存储在 Hadoop 中的大数据集。BigDL 的特点包括高性能、使用 Intel MKL 和多线程编程、支持横向扩展、并且支持加载预训练的 Caffe 和 Torch 模型到 Spark 程序中。BigDL 比较适合企业在无 GPU 环境下的低成本深度学习部署。。

BigDL on Sjpark 可以应用在多个领域，如金融风控中的分布式训练欺诈检测模型，以及结合 Spark ETL 与深度学习 CTR 预测进行广告推荐等。

12.5.1 BigDL 简介

BigDL 是 Intel 与 2016 年发布并开源的 Spark 原生分布式深度学习库，专为大数据场景设计，可直接在现有 Spark/Hadoop 集群上运行，无需额外修改集群配置。BigDL 以 Scala 为核心开发语言（也支持 Python 语言，通过 PySpark 调用），通过 Spark RDD/DataFrame/DataSet API 实现数据交互，可与 Spark ML Pipelines 集成，并利用 Spark 任务调度进行分布式计算，使其成为 Hadoop/Spark 生态中高效的深度学习解决方案。

美中不足的是，BigDL 仅支持 CPU 计算，对 GPU 加速不支持，依赖高内存配置（易引发 OOM 错误）。不过通过 Intel MKL 库和多线程优化，在 CPU 集群上性能显著优于传统框架（如 Caffe、TensorFlow），单节点性能提升达 48 倍。

目前 BigDL 的版本是 BigDL2，它已经发展出了包括大语言模型在内的庞大的 AI 框架和库，如下图所示：

BigDL 2 使数据科学家和数据工程师更容易构建端到端的分布式人工智能应用程序。BigDL 2 版本提供了以下特性：

（1）DLlib：用于 Apache Spark 的分布式深度学习库（即原始的 BigDL 框架，具有 Keras 风格的 API 和 Spark 机器学习管道支持）。

（2）Orca：为分布式大数据无缝扩展 TensorFlow 和 PyTorch 管道。

（3）Friesian：一个大规模的端到端推荐框架。

（4）Chronos：使用 AutoML 的可扩展时间序列分析。

（5）PPML：隐私保护大数据分析和机器学习。

BigDL 深度集成 Apache Spark，使其可以直接在 Spark 集群上运行，无需额外修改集群配置。具体集成方式包括：

1）数据交互

支持 RDD/DataFrame。BigDL 可以直接处理 Spark RDD 或 DataFrame，无需额外数据转换。与 Hadoop 生态兼容，支持从 HDFS、HBase、Hive 直接加载数据，适用于大数据场景。

2）计算调度

Spark 任务并行化。BigDL 将深度学习计算任务分解为 Spark 任务，每个任务使用 Intel MKL 和多线程优化计算。参数同步，采用同步随机梯度下降（SGD）和 AllReduce 通信优化，减少 Spark 集群的通信开销。

3）与 Spark ML Pipeline 集成

BigD 的模型可以嵌入 Spark ML Pipeline，与 Spark MLlib 的机器学习流程无缝结合。

4）预训练模型支持

支持加载 Caffe/Torch 预训练模型到 Spark 环境进行推理或迁移学习。

需要注意的是，目前 BigDL 支持的平台有 Linux 和 macOS，但是不支持 Windows。本书使用的是 Linux 平台（基于小白学苑的 PBCP 个人大数据学习平台）。

本书主要关注用于 Apache Spark 的分布式深度学习库，即 DLlib。使用 DLlib，我们可以使用相同的 Spark DataFrames 和 ML Pipeline APIs，将分布式深度学习应用程序编写为标准（Scala 或 Python） Spark 程序。

*12. 5. 2 BigDL 集成 Spark

1. 安装 BigDL

首先下载 BigDL 的预编译包。下载地址为：https://bigdl.readthedocs.io/en/latest/doc/release.html。选择要运行的 Spark 版本（2 或 3），如图 12-所示：

因为我们的 Spark 版本是 3，所以点击图中红框所示链接，下载 nightly build 版本：bigdl-assembly-spark_3.1.3-2.5.0-SNAPSHOT-jar-with-dependencies.jar。

2. 配置环境变量

提取预构建包后，需要设置 BIGDL_HOME 和 SPARK_HOME 环境变量，如下所示：

```
export SPARK_HOME=/home/hduser/bigdata/spark3
export BIGDL_HOME=/home/hduser/bigdata/BigDL
```

2. 使用 Spark 交互 shell

可以使用 Spark 交互 shell 尝试 BigDL，如下所示：

```
${BIGDL_HOME}/bin/spark-shell-with-dllib.sh
```

然后，将看到如下所示的欢迎信息：

在尝试 BigDL API 之前，应该使用 initNNcontext 来验证环境：

```
scala> import com.intel.analytics.bigdl.dllib.NNContext

scala> val sc = NNContext.initNNContext("Run Example")
```

执行以上代码，如下图所示：

NNContext.initNNContext()是 BigDL 中用于初始化 SparkContext 的便捷方法。SparkContext 是 Apache Spark 应用程序的入口点，它负责与集群管理器通信，分配资源，并在集群上执行并行操作。在使用 BigDL 进行分布式深度学习任务时，需要一个 SparkContext 来协调集群中的计算资源。

该方法会根据当前环境自动创建一个合适的 SparkContext 对象，并返回该对象。它会根据不同的运行环境（如本地模式、YARN 模式、Mesos 模式等）进行相应的配置，以确保 BigDL 能够在分布式环境中正常运行。NNContext.initNNContext() 主要用于快速初始化 SparkContext，适合初学者和简单的应用场景。

一旦成功启动了环境，我们就可以使用 DLlib API 了。例如，要尝试使用 DLlib 中的 dllib.keras API,可以试试下面的代码：

```
scala> import com.intel.analytics.bigdl.dllib.keras.layers._
scala> import com.intel.analytics.bigdl.numeric.NumericFloat
scala> import com.intel.analytics.bigdl.dllib.utils.Shape

scala> val seq = Sequential()
       val layer = ConvLSTM2D(32, 4, returnSequences = true, borderMode = "
same", inputShape = Shape(8, 40, 40, 32))
       seq.add(layer)
```

退出 spark-shell，执行如下命令：

```
scala> :q
```

3. 运行 BigDL 示例

可以运行一个 bigdl-dllib 程序，例如，语言模型，将其作为一个标准的 Spark 程序在本地机器或分布式集群上运行。

首先，准备数据集（http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz）。然后，在主目录（home）下提取 PTB 数据集，如下所示：

```
tar xvfz simple-examples.tgz -C $HOME
```

这里我们需要的数据集是$HOME/simple-examples/data。

这个例子引用了 tensorflow ptb 的例子，它展示了如何在一个具有挑战性的语言建模任务上训练一个循环神经网络（RNN）。模型的核心是一次处理一个单词，并计算句子中下一个单词可能值的概率。在这里，我们使用 Penn Tree Bank （PTB）作为训练数据集，这是衡量这些模型质量的流行基准，同时规模小且训练速度相对较快。

```
spark-submit \
--master spark://... \
--driver-memory 40g  \
--executor-memory 100g  \
--executor-cores cores_per_executor \
--total-executor-cores total_cores_for_the_job \
--class com.intel.analytics.bigdl.dllib.example.languagemodel.PTBWordLM \
dist/lib/bigdl-VERSION-jar-with-dependencies.jar \
```

```
-f $HOME/simple-examples/data -b 40 --checkpoint $HOME/model --numLayers 2
 --vocab 10001 --hidden 650 --numSteps 35 --learningRate 0.005 -e 20 --lea
rningRateDecay 0.001 --keepProb 0.5 --overWrite --withTransformerModel
```

然后，运行如下命令：

```

```

12.5.2 了解 DLlib 库

DLlib 是 Apache Spark 的分布式深度学习库。使用 DLlib，用户可以将深度学习应用程序编写为标准的 Spark 程序（使用 Scala 或 Python APIs）。

DLlib 库包含了原始 BigDL 项目的功能，并为 Spark 上的分布式深度学习提供了以下高级 API：

（1）类 Keras API。

（2）Spark ML 管道支持。

1. 类 Keras API

DLlib 提供了基于 Keras 1.2.2 的类 Keras API，用于在 Apache Spark 上进行分布式深度学习。用户可以很容易地使用类 Keras 的 API 来创建一个神经网络模型，并在 Spark 上以分布式的方式对它进行训练、评估或调优。

要使用类 Keras 的 API 在 Scala 中定义一个模型，只需要导入以下包：

```
import com.intel.analytics.bigdl.dllib.keras.layers._
import com.intel.analytics.bigdl.dllib.keras.models._
import com.intel.analytics.bigdl.dllib.utils.Shape
```

关于类 Keras API 的突出特性之一是形状推断。用户只需要为模型的第一层指定输入形状（一个不包括批维度的 shape 对象，例如，用于 3D 输入的 inputShape=Shape(3,4)），对于其余层，输入维度将被自动推断。

2. Spark ML 管道支持

DLlib 中的 NNFrames 为 Spark 上的分布式深度学习提供了 Spark DataFrame 和 ML Pipeline 支持。它包括 Python 和 Scala 接口，并与 Spark 2.x 和 Spark 3.x 兼容。

NNFrames 主要包含以下 API。

1）NNEstimator 和 NNModel

BigDL DLLib 提供了使用 Spark DataFrame 进行模型训练的 NNEstimator，它提供了使用 Apache Spark Estimator 和 Transformer 模式训练 BigDL 模型的高级 API，因此用户可以方便地将 BigDL DLLib 安装到 ML 管道中。NNEstimator 的拟合结果是一个 NNModel，它是一个 Spark ML Transformer。

2）NNClassifier 和 NNClassifierModel

NNClassifier 和 NNClassifierModel 扩展了 NNEstimator 和 NNModel，专注于分类任务，其中标签列和预测列都是 Double 类型。

3）NNImageReader

NNImageReader 将图像加载到 Spark DataFrame 中。

12.6 BigDL 类 Keras 的 API

在本节，我们详细了解 DLlib 库中所提供的类 Keras 网络的 API 及其使用。

12.6.1 输入和输出形状

类 Keras API 中模型的输入和输出形状是由 Shape 对象描述的。在深度学习框架里，Shape 类通常用于表示张量（Tensor）的形状，张量的形状决定了其维度和每个维度的大小。

Shape 可以分为 SingleShape 和 MultiShape。SingleShape 只是一个指示形状尺寸的 Int 列表，而 MultiShape 本质上是一个形状列表。

创建形状的示例代码：

```
// 创建一个 SingleShape
val shape1 = Shape(3, 4)

// 创建一个 MultiShape，由两个 SingleShape 组成
val shape2 = Shape(List(Shape(1, 2, 3), Shape(4, 5, 6)))
```

在上例中，Shape(3,4)构建了一个形状为(3,4)的对象，这意味着张量是一个二维数组，第一维大小为 3，第二维大小为 4。

执行以上代码，结果如下所示：

可以使用 toSingle()方法将 Shape 转换为 SingleShape。类似地，使用 toMulti()将 Shape 转换为 MultiShape。

12.6.2 定义模型

可以使用 Sequential API 或 Functional API 来定义模型。在定义模型时，要指定第一层的输入形状。

1. Sequential API

BigDL 提供了用于构建顺序模型的 Sequential 类。顺序模型是一种线性堆叠的模型结构，层可以逐个添加到 Sequential 容器中，模型中层的顺序与插入顺序相同。各层按照添加的顺序依次执行计算。

在 BigDL 里，Sequential 是一种常用的模型构建方式，它能让用户把多个层依次堆叠起来，从而构建出神经网络。要创建一个 Sequential 容器，代码如下：

```
Sequential(name="seq1")   // 创建了一个空的顺序模型对象
```

参数 name 是指定顺序模型名称的字符串。默认为 None。

创建顺序模型的示例代码如下：

```
import com.intel.analytics.bigdl.dllib.keras.layers.{Dense, Activation}
import com.intel.analytics.bigdl.dllib.keras.models.Sequential
import com.intel.analytics.bigdl.dllib.utils.Shape

// 创建模型，使用 Sequential API
val model = Sequential[Float]()
```

```
/* 向模型中添加输入层和激活函数 */
// 构建一个全连接层，其输入形状为 128，输出维度是 32，将其添加到模型的末尾
model.add(Dense[Float](32, inputShape = Shape(128)))
// 添加一个 ReLU 激活函数
model.add(Activation[Float]("relu"))
```

在上面的代码中，有这么一行：

```
model.add(Dense[Float](32, inputShape = Shape(128)))
```

这行代码的用途是给模型添加一个全连接层（也被叫做密集层）。这里构建的全连接层，其输入形状为 128，输出维度是 32。Dense 是 BigDL 里用于构建全连接层的类，其第一个参数 32 是全连接层的输出维度，也就是该层输出神经元的数量，这里表示输出一个长度为 32 的向量。inputShape 是 Dense 层的第二个参数，其作用是明确该层输入数据的形状。Shape(128) 构建了一个 Shape 对象，此对象代表输入数据是一个一维向量，长度为 128。

另一行代码：

```
model.add(Activation[Float]("relu"))
```

这行代码的作用是往神经网络模型里添加一个激活层，具体使用的激活函数是修正线性单元（ReLU）。Activation 是 BigDL 中用于构建激活层的类。“relu”是传递给 Activation 类构造函数的参数，指定了要使用的激活函数类型。“relu”是代表修正线性单元激活函数，是一种常见的激活函数，它具有计算简单、收敛速度快等优点，在很多深度学习任务中都有广泛应用。ReLU 的数学表达式为：f(x)=max(0,x)。也就是说，对于输入的每个元素 x，如果 x 大于 0，则输出 x；如果 x 小于等于 0，则输出 0。

激活层在神经网络中扮演着关键角色，它能够给神经网络引入非线性因素，在神经网络的前向传播过程中，输入数据会经过一系列的线性变换与非线性激活，最终输出一个指定输出形状（这里长度为 32） 的向量。

2. Functional API

在 Functional API 中，模型被描述为一个图。在定义一些复杂的模型（例如，具有多个输出的模型）时，它比 Sequential API 更方便。

要创建一个输入节点，代码如下：

```
Input(inputShape = null, name = null)
```

其中参数的含义如下：

（1）inputShape：一个 Shape 对象，表示输入节点的形状，不包括 batch。

（2）name：设置输入节点名称的字符串。如果未指定，其名称将默认为生成的字符串。

创建图容器的示例代码如下：

```
Model(input, output)
```

其中参数的含义如下：

（1）input：一个输入节点或输入节点数组。

（2）output：一个输出节点或输出节点数组。

要合并输入节点列表（不是层），需遵循 Functional API 中的一些合并模式：

```
import com.intel.analytics.bigdl.dllib.keras.layers.Merge.merge

merge(inputs, mode = "sum", concatAxis = -1) // 这将返回一个输出节点
```

其中参数的含义如下：

（1）inputs：节点实例列表。必须多于一个节点。

（2）mode：合并模式。字符串类型，必须是"sum"、"mul"、"concat"、"ave"、"cos"，"dot"、"max"这些值之一，默认是"sum"。

（3）concatAxis：Int 类型，连接节点时使用的轴。仅当合并模式为"concat"时指定此参数。默认值是-1，表示输入的最后一个轴。

创建一个图模型的示例代码如下：

```
import com.intel.analytics.bigdl.dllib.keras.layers.{Dense, Input}
import com.intel.analytics.bigdl.dllib.keras.layers.Merge.merge
import com.intel.analytics.bigdl.dllib.keras.models.Model
import com.intel.analytics.bigdl.dllib.utils.Shape

// 实例化输入节点
val input1 = Input[Float](inputShape = Shape(8))
val input2 = Input[Float](inputShape = Shape(6))

// 调用 inputs(), 传入一个输入节点, 获得一个输出节点
val dense1 = Dense[Float](10).inputs(input1)
val dense2 = Dense[Float](10).inputs(input2)

// 按照某些合并模式合并两个节点
val output = merge(inputs = List(dense1, dense2), mode = "sum")

// 创建一个图容器
val model = Model[Float](Array(input1, input2), output)
```

12.6.3 核心层 API

深度神经网络包括很多输入层、多个隐藏层和输出层。在 DLlib 库中也相应地提供了创建这些层的 API。

1. Dense

它是一个密集连接的 NN（神经网络）层。最常见的输入是 2D。其定义如下：

```
Dense(outputDim,
      init = "glorot uniform",
      activation = null,
      wRegularizer = null,
      bRegularizer = null,
      bias = true,
      inputShape = null
   )
```

各个参数的含义如下：

（1）outputDim：输出维度的大小。

（2）init：层权值的初始化方法。默认是 Xavier。也可以传入相应的字符串表示，如 glorot_uniform 或 normal 等，用于工厂方法中的简单初始化方法。

（3）activation：使用的激活函数。默认为 null。也可以传入相应的字符串表示，如 relu 或 sigmoid 等，用于工厂方法中的简单激活。

（4）wRegularizer：Regularizer 的一个实例，应用于输入权重矩阵。默认为 null。

（5）bRegularizer：Regularizer 的一个实例，应用于偏置（bias）。默认为 null。

（6）bias：是否包含偏置（即使层仿射而不是线性）。默认为 true。

（7）inputShape：只有当使用此层作为模型的第一层时才需要指定此参数。对于 Scala API，它应该是一个 Shape 对象。对于 Python API，它应该是一个形状元组。应排除 batch 维度。

下面这段代码构建了一个简单的包含单个全连接层（带有 ReLU 激活函数）的神经网络模型，生成了随机输入数据，并将输入数据传入模型进行前向传播计算，最终得到输出结果。通过这种方式，可以对模型的基本功能进行验证和测试。

```
// scala
```

Tensor 是 BigDL 中用于表示多维数组的类，类似于 NumPy 中的 ndarray 或 PyTorch 中的 Tensor。[Float]指定了张量的数据类型为 Float。(2, 4)表示创建一个形状为 (2, 4) 的二维张量，即该张量有 2 行 4 列。.randn()是 Tensor 类的一个方法，用于生成服从标准正态分布（均值为 0，标准差为 1）的随机数填充该张量。所以，input 是一个包含随机数的 2x4 张量。

调用模型的 forward()方法进行前向传播计算。前向传播是指将输入数据依次通过模型的每一层，进行计算并得到最终输出的过程。将之前生成的随机输入张量 input 作为参数传入 forward()方法。保存模型前向传播的输出结果到 output，其形状由模型的结构决定。由于输入是一个 2x4 的张量，经过一个输出维度为 5 的全连接层后，输出 output 是一个 2x5 的张量。

执行以上代码，输出内容如下：

2. SparseDense

SparseDens 是层 Dense 的稀疏版本。SparseDense 与 Dense 有两个不同：首先，SparseDense 的输入张量是一个 SparseTensor。其次，默认情况下，SparseDense 在反向传播中不会反向梯度到下一层，因为 SparseDense 的 gradInput 在大多数情况下是无用的并且非常大。

但是，考虑到像 Wide&Deep 这样的模型，BigDL 提供了 backwardStart 和 backwardLength 来反向部分梯度到下一层。

最常见的输入是 2D。其定义如下：

```
SparseDense(outputDim,
            init = "glorot_uniform",
            activation = null,
            wRegularizer = null,
            bRegularizer = null,
            backwardStart = -1,
            backwardLength = -1,
            initWeight = null,
            initBias = null,
            initGradWeight = null,
```

```
            initGradBias = null,
            bias = true,
            inputShape = null
        )
```

各个参数的含义如下：

（1）outputDim：输出维度大小。

（2）init：层权重的初始化方法的字符串表示形式。默认是 glorot_uniform。

（3）activation：要使用的激活函数的字符串表示形式。默认为 null。

（4）wRegularizer：Regularizer 的一个实例，应用于输入权重矩阵。默认为 null。

（5）bRegularizer：Regularizer 的一个实例，应用于偏置（bias）。默认为 null。

（6）bias：是否包含偏置（即使层仿射而不是线性）。默认为 true。

（7）backwardStart：反向开始索引，从 1 开始计数。

（8）backwardLength：反向长度。

（9）inputShape：只有当使用此层作为模型的第一层时才需要指定此参数。对于 Scala API，它应该是一个 Shape 对象。对于 Python API，它应该是一个形状元组。应排除 batch 维度。

（10）name：设置层名称的字符串。如果未指定，其名称将默认为生成的字符串。

下面这段代码主要演示了如何使用 BigDL 中的 SparseDense 层处理稀疏输入数据。SparseDense 层可以高效地处理稀疏张量，将稀疏输入转换为密集输出。

```
// scala
```

这段代码的主要目的是创建一个 SparseDense 层，将随机生成的密集张量转换为稀疏张量，然后将其作为输入传递给该层进行前向传播，最终得到输出张量。通过使用稀疏张量，可以在处理大规模稀疏数据时提高计算效率。

执行以上代码，输出内容如下：

3. MaxoutDense

一个密集的 maxout 层，它取线性层中元素的最大值。这允许层在输入上学习一个凸的、分段的线性激活函数。

该层的输入应为 2D。其定义如下：

```
MaxoutDense(outputDim,
            nbFeature = 4,
            wRegularizer = null,
            bRegularizer = null,
            bias = true,
            inputShape = null
        )
```

各个参数的含义如下：

（1）outputDim：输出维度大小。

（2）nbFeature：内部使用的密集层数。整数。默认值是 4。

（3）wRegularizer：Regularizer 的一个实例（例如，L1 或 L2 正则化），应用于输入权重矩阵。默认为 null。

（4）bRegularizer：Regularizer 的一个实例，应用于偏置（bias）。默认为 null。

（5）bias：是否包含偏置（即使层仿射而不是线性）。默认为 true。

（6）inputShape：只有当使用此层作为模型的第一层时才需要指定此参数。对于 Scala API，它应该是一个 Shape 对象。对于 Python API，它应该是一个形状元组。应排除 batch 维度。

下面这段代码借助 BigDL 构建了一个简单的神经网络模型，使用 Sequential 容器添加 MaxoutDense 层，接着生成随机输入数据并进行前向传播以获取输出结果。

```
// scala
```

这段代码构建了一个包含 MaxoutDense 层的简单顺序模型，生成了形状为 (2, 3) 的随机输入数据，然后将输入数据传入模型进行前向传播，得到形状为 (2, 2) 的输出结果。

在上面的代码中：

（1）使用 Sequential[Float]()构建了一个顺序模型，该模型会按照层添加的先后顺序依次执行。[Float] 明确了模型中数据的类型为单精度浮点数（Float）。

（2）model.add(...)方法用于把一个新的层添加到顺序模型中。MaxoutDense 是一种特殊的全连接层，其核心思想是在每个神经元的输出处取多个线性组合的最大值，其中参数 2 代表该层的输出维度，也就是该层会输出一个大小为 2 的向量；参数 inputShape = Shape(3) 指定输入数据的形状。这里表示输入数据应为一个长度为 3 的向量。

（3）Tensor[Float](2, 3)创建一个形状为 (2, 3) 的单精度浮点型张量，其中，第一个维度 2 通常代表批大小，意味着每次处理 2 个样本；第二个维度 3 与 MaxoutDense 层指定的输入形状相匹配。方法.randn()使用标准正态分布（均值为 0，标准差为 1）的随机数填充该张量。

（4）model.forward(input)：把输入张量 input 传入模型，进行前向传播计算。前向传播的过程就是输入数据依次经过模型中的各个层，最终得到输出结果。将前向传播的结果存储在 output 变量中。由于 MaxoutDense 层的输出维度为 2，且输入的批量大小为 2，所以 output 张量的形状为 (2, 2)。

执行以上代码，输出内容如下：

4. Masking

使用掩码值跳过序列的时间步长。其定义如下：

```
Masking(maskValue = 0.0, inputShape = null)
```

各个参数的含义如下：

（1）maskValue：掩码值。对于输入中的每个时间步长（第二维度），如果该时间步长的所有输入值都等于'maskValue'，则该时间步长将在所有下游层中被屏蔽（跳过）。

（2）inputShape：只有当使用此层作为模型的第一层时才需要指定此参数。对于 Scala API，它应该是一个 Shape 对象。对于 Python API，它应该是一个形状元组。应排除 batch 维度。

下面这段代码使用 BigDL 构建了一个简单的顺序模型，该模型仅包含一个 Masking 层。然后生成随机输入数据并将其传入模型进行前向传播，得到输出结果。Masking 层的主要作用是屏蔽输入序列中的特定值，常用于处理变长序列数据。

```
// scala

```

这段代码构建了一个只包含 Masking 层的简单顺序模型，生成了形状为 (2, 3) 的随机输入数据，然后将输入数据传入模型进行前向传播，得到形状同样为 (2, 3) 的输出结果。Masking 层在实际应用中通常用于处理变长序列数据，屏蔽掉填充值，使得后续层能够正确处理这些序列。在这段代码中，由于输入数据是随机生成的，没有特定的需要屏蔽的值，所以输入和输出的形状保持一致。即 output 张量的形状仍然为 (2, 3)。

执行以上代码，输出内容如下：

5. Reshape

将输出重塑为特定形状。通过在目标形状中允许一个-1 来支持形状推断。例如，如果输入形状为（2,3,4），目标形状为（3,-1），则输出形状为（3,8）。

其定义如下：

```
Reshape(targetShape, inputShape = null)
```

各个参数的含义如下：

（1）targetShape：想要的目标形状。应排除 batch 尺寸。

（2）inputShape：只有当使用此层作为模型的第一层时才需要指定此参数。对于 Scala API，它应该是一个 Shape 对象。对于 Python API，它应该是一个形状元组。应排除 batch 维度。

下面这段代码使用 BigDL 构建了一个简单的顺序模型，该模型仅包含一个 Reshape 层。接着生成符合特定形状要求的随机输入张量，将其传入模型进行前向传播，最终得到经过形状重塑后的输出张量。Reshape 层的主要作用是改变输入张量的形状，而不改变其元素数量。

```

```

这段代码构建了一个包含 Reshape 层的顺序模型，生成了形状为 (2, 2, 3, 4) 的随机输入张量，将其传入模型进行前向传播后，得到形状为 (3, 8) 的输出张量。Reshape 层的作用是在不改变元素数量的前提下，重新组织输入张量的形状。需要注意的是，这里输入张量的形状应为 (2, 3, 4)，并且输入张量的元素总数（2 * 3 * 4 = 24）必须与目标形状的元素总数（3 * 8 = 24）相等，因为 Reshape 操作不会改变张量的元素数量，只是重新排列元素的布局。

执行以上代码，输出内容如下：

6. Merge

用于按照某种合并模式将输入列表合并为单个输出。

Merge 必须至少有两个输入层。其定义如下：

```
Merge(layers = null, mode = "sum", concatAxis = -1, inputShape = null)
```

各个参数的含义如下：

（1）layers：层实例的列表。必须不止一层。

（2）mode：合并模式。字符串，必须为以下名称之一：sum、mul、concat、ave、cos、dot、max。默认是 sum。

（3）concatAxis：整型，连接层时使用的轴。仅当合并模式为 concat 时指定此参数。默认值是-1，表示输入的最后一个轴。

（4）inputShape：只有当使用此层作为模型的第一层时才需要指定此参数。对于 Scala API，它应该是一个 MultiShape 对象。对于 Python API，它应该是一个形状元组。应排除 batch 维度。

下面这段代码使用 BigDL 构建了一个简单的神经网络模型，该模型主要功能是将两个输入张量按元素求和。具体步骤包括创建一个顺序模型，定义两个输入层，使用 Merge 层将这两个输入层合并，生成两个随机输入张量，最后将输入张量传入模型进行前向传播得到输出结果。

```

```

这段代码构建了一个简单的神经网络模型，该模型包含一个合并层，用于将两个输入张量按元素求和。通过生成随机输入张量并进行前向传播，最终得到求和后的输出张量。这种模型结构在处理需要对多个输入进行合并操作的场景中非常有用，例如多模态数据融合等。

执行以上代码，输出内容如下：

7. Select

在给定的维度中选择输入的索引并返回子集部分。batch 维度需要保持不变。例如，如果输入是：

```
[[1, 2, 3], [4, 5, 6]]
```

Select（1,1）将给出输出[2 5]，Select（1, -1）将给出输出[3 6]。其定义如下：

```
Select(dim, index, inputShape = null)
```

各个参数的含义如下：

（1）dim：要选择的维度。基于 0 的索引。无法选择 batch 维度。-1 表示输入的最后一个维度。

（2）index：要选择的维度的索引。基于 0 的索引。-1 表示输入的最后一个维度。

（3）inputShape：只有当使用此层作为模型的第一层时才需要指定此参数。对于 Scala API，它应该是一个 Shape 对象。对于 Python API，它应该是一个形状元组。应排除 batch 维度。

下面这段代码使用 BigDL 构建了一个简单的顺序模型，模型中仅包含一个 Select 层。代码先生成一个符合特定形状要求的随机输入张量，然后将其传入模型进行前向传播，最终通过 Select 层从输入张量中选取特定部分得到输出结果。

```
```

在上面的代码中，Select[Float]是一个用于从输入张量中选取特定部分的层。其中：

（1）第一个参数 1：表示要在哪个维度上进行选择操作。在张量的维度索引中，维度从 0 开始计数，这里的 1 表示在第二个维度上进行选择。

（2）第二个参数 2：表示在指定维度（这里是第二个维度）上选择的索引位置。即选择该维度上索引为 2 的元素。

（3）inputShape = Shape(3, 1, 3)：指定输入张量的形状为 (3, 1, 3)。Select 层会根据这个输入形状和选择参数来确定如何从输入张量中选取元素。

这段代码构建了一个包含 Select 层的顺序模型，生成了形状为 (1, 3, 1, 3) 的随机输入张量，将其传入模型进行前向传播后，通过 Select 层在第二个维度上选择索引为 2 的元素，最终得到形状为 (1, 1, 3) 的输出张量。Select 层在处理需要从张量中提取特定部分数据的场景中非常有用。

执行以上代码，输出内容如下：

8. Permute

根据给定的模式排列输入的维度。用于将 RNNs 和 convnets 连接在一起。其定义为：

```
Permute(dims, inputShape = null)
```

各个参数的含义如下：

（1）dims：Int 数组。排列模式，不包括 batch 维度。索引从 1 开始。

（2）inputShape：只有当使用此层作为模型的第一层时才需要指定此参数。对于 Scala API，它应该是一个 Shape 对象。对于 Python API，它应该是一个形状元组。应排除 batch 维度。

下面代码利用 BigDL 构建了一个简单的顺序模型，该模型仅包含一个 Permute 层。代码先生成一个符合特定形状的随机输入张量，再将其传入模型进行前向传播，最后通过 Permute 层对输入张量的维度进行重新排列得到输出结果。

```
```

在上面的代码中，Permute[Float]是一个用于对输入张量的维度进行重新排列的层。其中：

（1）Array(2, 1, 3)：指定了维度的排列顺序。在张量的维度索引中，维度从 1 开始计数。这里的 (2, 1, 3) 表示将输入张量的第二个维度放到新张量的第一个位置，第一个维度放到新张量的第二个位置，第三个维度保持在新张量的第三个位置。

（2）inputShape = Shape(2, 2, 3)：明确了输入张量的形状。即输入张量应该是 (2, 2, 3) 这种形状的，Permute 层会依据这个输入形状和指定的维度排列顺序来对输入张量进行操作。

这段代码构建了一个包含 Permute 层的顺序模型，生成了形状为 (2, 2, 2, 3) 的随机输入张量，将其传入模型进行前向传播后，通过 Permute 层对输入张量的维度进行重新排列，最终得到输出张量。本例中由于 Permute 层只对除批量维度外的维度进行重排，所以输出张量的形状为 (2, 2, 2, 3)，但内部元素的维度顺序发生了变化，变为 (批大小, 原第二维, 原第一维, 原第三维)。

执行以上代码，输入数据如下所示：

输出数据如下：

Permute 层在处理需要调整张量维度顺序的场景中非常有用，比如在不同数据格式转换或适应特定网络架构时。

关于更多层 API 的说明，请参考官方文档。

12.6.4 持久化模型

要保存一个 Keras 模型，可以调用模型的方法 saveModel(path)。示例代码如下：

```
```

要加载保存的 Keras 模型，需要调用 Models.load_model(path)方法。示例代码如下：

```
```

12.7 BigDL 支持 Spark ML 管道的 API

BigDL DLlib 库中的 NNFrames 提供了与 Spark 机器学习管道兼容的 API，可以很方便地将深度学习集成到 Spark 机器学习流中。

12.7.1 NNEstimator

NNEstimator 继承自 org.apache.spark.ml.Estimator，支持用 Spark DataFrame/DataSet 数据训练 BigDL 模型（简单调用 fit()方法）。它可以集成到一个标准的 Spark ML Pipeline 中，允许用户组合 BigDL 和 Spark MLlib 的组件。支持使用 Spark 的 DataFrame 进行训练。

默认情况下，使用 SeqToTensor 将数组或向量转换为一维张量（Tensor）。使用预处理（Preprocessing）允许 NNEstimator 只缓存原始数据，并减少特征转换和训练期间的内存消耗，它还使模型能够消化 DataFrame 目前不支持的额外数据类型。

NNEstimator 可以为不同的场景使用不同的参数来创建。例如：

1）NNEstimator(model, criterion)

只接收 model 和 criterion 参数，并使用 SeqToTensor 作为特征和标签预处理（Preprocessing）。NNEstimator 将从特征和标签列中提取数据（仅支持标量、Array[_]或 Vector 数据类型），并将每个特征/标签转换为一维张量（Tensor）。这些张量将被组合到 BigDL Sample 中并发送给模型进行训练。示例代码如下：

```
val estimator = NNEstimator(model, criterion)
```

2）NNEstimator(model, criterion, featureSize: Array[Int], labelSize: Array[Int])

接收 model、criterion、featureSize（Int 数组）和 labelSize（Int 数组）输入参数。NNEstimator 将从特征和标签列中提取数据（仅支持标量、Array[_]或 Vector 数据类型），并根据指定的张量大小将每个特征/标签转换为张量。

3）NNEstimator(model, criterion, featureSize: Array[Array[Int]], labelSize: Array[Int])

这是多输入模型的接口。它接受 model、criterion, featureSize（Int 数组的数组）和 labelSize（Int 数组）输入参数。NNEstimator 将从特征和标签列中提取数据（仅支持标量、Array[_]或 Vector 数据类型），并根据指定的张量大小将每个特征/标签转换为张量。

4）NNEstimator(model, criterion, featurePreprocessing: Preprocessing[F, Tensor[T]], labelPreprocessing: Preprocessing[F, Tensor[T]])

接收 model、criterion、featurePreprocessing 和 labelPreprocessing 输入参数。NNEstimator 将从特征和标签列中提取数据，并通过 featurePreprocessing 和 labelPreprocessing 将每个特征/标签转换为张量（Tensor）。此构造函数在支持额外数据类型方面提供了更大的灵活性。

同时，对于高级用例（例如具有多个输入张量的模型），NNEstimator 支持：setSamplePreprocessing(value: Preprocessing[(Any, Option[Any]), Sample[T]])，根据用户指定的预处理直接组成 Sample。

请看下面的示例代码。

```

```

执行以上代码，输出内容如下：

12.7.2 NNModel

NNModel 继承自 Spark ML 的 Transformer。用户可以在 NNEstimator 上调用 fit()方法来获得一个 NNModel，或者直接从 BigDLModel 中合成一个 NNModel。BigDLModel 使用户能够将预训练的 BigDL 模型包装到 NNModel 中，并将其用作 Spark ML 管道中的 Transformer 来预测 DataFrame（DataSet）的结果。

针对不同的场景，可用不同的参数来创建 NNModel：

1）NNModel(model)

只接收 model（模型）参数，使用 SeqToTensor 作为特征预处理。NNModel 将从特征列中提取数据（仅支持标量、Array[_]或 Vector 数据类型），并将每个特征转换为一维张量（Tensor）。张量将被发送到模型中进行推理。例如：

```
val nnModel = NNModel(bigDLModel)
```

2）NNModel(model, featureSize: Array[Int])

接收 model 和 featureSize（Int 数组）参数。NNModel 将从特征列中提取数据（仅支持标量、Array[_]或 Vector 数据类型），并根据指定的 Tensor 大小将每个特征转换为 Tensor。对于多输入模型，用户也可以设置 featureSize 为 Array[Array[Int]]。

3）NNModel(model, featurePreprocessing: Preprocessing[F, Tensor[T]])

接收 model 和 featurePreprocessing 作为传入参数。NNModel 将从特征列中提取数据，并通过特征预处理将每个特征转换为张量。此构造函数在支持额外数据类型方面提供了更大的灵活性。

同时，对于高级用例（例如具有多个输入张量的模型），NNModel 支持：setSamplePreprocessing(value: Preprocessing[Any, Sample[T]])，根据用户指定的预处理直接组成 Sample。

我们可以通过以下方式从 NNModel 获取模型：

```
val model = nnModel.getModel()
```

12.7.3 NNClassifier

NNClassifier 是一个专门的 NNEstimator，它简化了标签空间离散的分类任务的数据格式。它只支持 DoubleType 类型的标签列，拟合的 NNClassifierModel 将具有 DoubleType 的预测列。

对于不同的场景，可以用不同的参数来创建 NNClassifier。

1）NNClassifier(model, criterion)

只接收参数 model 和 criterion，使用 SeqToTensor 作为特征和标签预处理。NNClassifier 将从特征和标签列中提取数据（仅支持标量、Array[_]或 Vector 数据类型），并将每个特征/标签转换为一维张量。这些张量将被组合成 BigDL Sample 并发送给模型用于训练。

参数 model 和 criterion 的含义如下：

（1）model：在 fit()方法中对 BigDL 模型进行优化。

（2）criterion：用于计算损失和梯度的准则。

示例代码如下：

```
val classifer =  NNClassifer(model, criterion)
```

2）NNClassifier(model, criterion, featureSize: Array[Int])

接受 model、criterion、featureSize（Int 数组）参数。NNClassifier 将从特征和标签列中提取数据，并根据指定的张量大小将每个特征转换为张量。ScalarToTensor 用于转换标签列。对于多输入模型，用户也可以设置 featureSize 为 Array[Array[Int]]。

3）NNClassifier(model, criterion, featurePreprocessing: Preprocessing[F, Tensor[T]])

接受 model、criterion、featurePreprocessing 参数。NNClassifier 将从特征和标签列中提取数据，并通过 featurePreprocessing（特征预处理）将每个特征转换为张量。此构造函数在支持额外数据类型方面提供了更大的灵活性。

同时，对于高级用例（例如具有多个输入张量的模型），NNClassifier 支持 setSamplePreprocessing(value: Preprocessing[(Any, Option[Any]), Sample[T]])，直接使用用户指定的预处理组成 Sample。

下面是使用 NNClassifier 来完成分类任务的示例。

```

```

执行以上代码，输出内容如下：

12.7.4 NNClassifierModel

NNClassifierModel 是一个专门用于分类任务的 NNModel。标签列和预测列的数据类型都是 Double。

可以为不同的场景使用不同的参数来创建 NNClassifierModel。

1）NNClassifierModel(model)

只接受 model 参数，使用 SeqToTensor 作为特征预处理。NNClassifierModel 将从特征列中提取数据（仅支持标量、Array[_]或 Vector 数据类型），并将每个特征转换为一维张量。张量将被发送到模型中进行推理。示例代码如下：

```
val nnClassifierModel = NNClassifierModel(model)
```

2）NNClassifierModel(model, featureSize: Array[Int])

接受 model 和 featureSize（Int 数组）参数。NNClassifierModel 将从特征列中提取数据（仅支持标量、Array[_]或 Vector 数据类型），并根据指定的张量（Tensor）大小将每个特征转换为 Tensor。对于多输入模型，用户也可以设置 featureSize 为 Array[Array[Int]]。

3）NNClassifierModel(model, featurePreprocessing: Preprocessing[F, Tensor[T]])

接受 model 和 featurePreprocessing（Int 数组）参数。NNClassifierModel 将从特征列中提取数据，并通过 featurePreprocessing（特征预处理）将每个特征转换为张量。此构造函数在支持额外数据类型方面提供了更大的灵活性。

同时，对于高级用例（例如具有多个输入张量的模型），NNClassifierModel 支持：set SamplePreprocessing(value: Preprocessing[Any, Sample[T]])，根据用户指定的预处理直接组成 Sample。

例如，我们把上面的代码重构，直接使用 NNClassifierModel，代码如下：

```

```

由于 NNModel/NNClassifierModel 继承自 Spark 的 Transformer 抽象类，所以只需在 NN Model/NNClassifierModel 上调用 transform()方法来进行预测。

12.7.5 超参数设置

在训练过程开始之前，可以修改优化算法、批处理大小、训练历元数（epoch number）和学习率来满足我们的目标。NNEstimator 和 NNClassifier 可以用同样的方式设置，否则 NNEstimator/NNClassifier 将使用默认值。

例如，在上面的示例代码中，我们创建的 NNEstimator 或 NNClassifier 均设置了一定的超参数。

```
// 用于 esitmator
estimator
    .setBatchSize(4)
    .setMaxEpoch(10)
    .setLearningRate(0.01)
    .setOptimMethod(new Adam())

// 用于 classifier
classifier
    .setBatchSize(4)
```

```
    .setMaxEpoch(10)
    .setLearningRate(0.01)
    .setOptimMethod(new Adam())
```

12.7.6 NNImageReader

NNImageReader 是主要的基于 DataFrame 的图像加载接口，定义了将图像读取到 DataFrame 的 API。

```
val imageDF = NNImageReader.readImages(imageDirectory, sc)
```

输出 DataFrame 包含单个名为“image”的列。“image”列的模式可以从 com.intel.analytics.bigdl.dllib.nnframes.DLImageSchema.byteSchema 访问。“image”列中的每条记录表示一条图像记录，格式为 Row(origin, height, width, num of channels, mode, data)，其中 origin 包含图像文件的 URI，data 保存图像文件的原始文件字节。mode 表示 OpenCV 兼容类型：CV_8UC3，在大多数情况下 CV_8UC1。模式结构如下：

```
val byteSchema = StructType(
    StructField("origin", StringType, true) ::
    StructField("height", IntegerType, false) ::
    StructField("width", IntegerType, false) ::
    StructField("nChannels", IntegerType, false) ::
    // OpenCV 兼容类型: CV_8UC3, CV_32FC1(大多数时候)
    StructField("mode", IntegerType, false) ::
    // 按 OpenCV 兼容顺序排列字节: 大多数时候是 row-wise BGR
    StructField("data", BinaryType, false) :: Nil)
```

加载图像后，用户可以使用在 com.intel.analytics.bigdl.dllib.feature.image 中定义的 Preprocessing 组合预处理步骤。

12.8 优化器

BigDL 中的优化器（Optimizer）是 BigDL 里用于优化模型参数的类。在深度学习训练过程中，模型的参数（如权重和偏置）需要不断调整，以使得模型的预测结果尽可能接近真实标签，而 Optimizer 就承担了这个参数优化的任务。

Optimizer 能够依据选定的优化算法（像随机梯度下降 SGD、Adagrad、Adam 等），对模型的参数进行更新。在每次迭代里，它会计算损失函数关于参数的梯度，接着按照优化算法的规则更新参数，从而降低损失函数的值。

Optimizer 允许用户设置各类超参数，例如学习率、批大小、迭代次数、正则化（L1/L2）、梯度裁剪等超参数。这些超参数会对模型的训练效果和收敛速度产生影响，通过调整它们可以找到最优的训练配置。

在训练过程中，Optimizer 可以定期对模型进行评估，计算模型在验证集上的性能指标（如准确率、损失值等）。这有助于监控模型的训练进度，防止过拟合或欠拟合。

因为 BigDL 是基于 Apache Spark 构建的，Optimizer 能够利用 Spark 的分布式计算能力，在集群上并行地开展模型训练。这样可以显著加快训练速度，特别是在处理大规模数据集和复杂模型时。

12.8.1 Adam 优化器

Adam（Adaptive Moment Estimation，自适应矩估计）是一种常用的优化算法，它结合了 AdaGrad 和 RMSProp 的优点，同时使用梯度的一阶矩估计（均值）和二阶矩估计（方差）来自适应地动态调整每个参数的学习率。Adam 在大多数情况下都能取得较好的效果，是目前使用最广泛的优化算法之一。适用于各种类型的深度学习任务，包括图像识别、语音识别、自然语言处理等。

这是 Adam 优化算法的实现，一阶基于梯度的随机目标函数优化。其定义如下：

```
val optim = new Adam(learningRate=1e-3, learningRateDecay=0.0, beta1=0.9,
beta2=0.999, Epsilon=1e-8)
```

其中各个参数的含义如下：

（1）learningRate：学习率。默认值是 1e-3。

（2）learningRateDecay：学习率衰减。默认值为 0.0。

（3）beta1：一阶矩系数。默认值为 0.9。

（4）beta2：二阶矩系数。默认值为 0.999。

（5）Epsilon：用于数值稳定性。默认值为 1e-8。

下面这段代码的演示了如何使用 Adam 优化器求解 Rosenbrock 函数的最小值。

```

```

上面的代码中创建了一个 Adam 优化器的实例 optm，并指定了优化器的学习率为 0.002。然后定义了目标函数 rosenBrock 有梯度计算。这个函数 rosenBrock 实现了 Rosenbrock 函数及其梯度的计算（Rosenbrock 函数是一个经典的非线性优化测试函数，也被称为“香蕉函数”），接受一个 Tensor[Float]类型的输入 x，返回一个元组(Float, Tensor[Float])，其中第一个元素是函数在 x 处的值，第二个元素是函数在 x 处的梯度。

然后创建了一个长度为 2 的输入张量 Tensor，并将其所有元素初始化为 0。这个向量 x 将作为优化算法的初始点。接着调用优化器 optm 的 optimize 方法，对目标函数 rosenBrock 进行优化，初始点为 x。优化过程会不断更新 x 的值，直到满足停止条件（例如达到最大迭代次数）。最后，将优化得到的最优解打印输出。

执行以上代码，输出内容如下：

12.8.2 SGD 优化器

在机器学习和深度学习里，梯度下降是用于最小化损失函数的一种迭代优化算法。传统的梯度下降在每次迭代时会使用整个训练数据集来计算损失函数的梯度，这在处理大规模数据集时计算成本非常高。

而随机梯度下降（SGD，Stochastic Gradient Descent）每次迭代仅随机选取一个训练样本或者一小批（mini-batch）样本计算梯度，然后依据这个梯度更新模型的参数。由于只使用部分样本，计算效率高，所以能更快地更新参数，特别适合处理大规模数据集。常用于处理大规模数据的训练任务，如大规模图像分类、自然语言处理等。不过它的收敛速度可能较慢，并且容易陷入局部最优解。

SGD 优化器的定义如下：

```
val optimMethod = new SGD(learningRate= 1e-3,learningRateDecay=0.0,
          weightDecay=0.0,momentum=0.0,dampening=Double.MaxValue,
          nesterov=false,learningRateSchedule=Default(),
          learningRates=null,weightDecays=null)
```

重构上一节的示例代码，演示如何使用 SGD 优化器求解 Rosenbrock 函数的最小值。

```
// 损失函数定义相同
// ...

// 只切换优化器
val optm = new SGD(learningRate=0.002)

// 初始化输入向量
val x = Tensor(2).fill(0)

// 执行优化
println(optm.optimize(rosenBrock, x))
```

执行以上代码，输出内容如下：

BigDL DLlib 库中 SGD 算法的简单实现提供了 optimize()方法。在创建 Optimize 时设置了优化方法后，Optimize 将在每次迭代结束时调用优化方法。

示例代码如下：

```

```

12.8.3 Adagrad 优化器

Adagrad（Adaptive Gradient ，自适应梯度）算法是一种在机器学习和深度学习领域用于优化的算法，属于自适应学习率算法的范畴。在传统的梯度下降算法中，所有参数都使用相同的固定学习率进行更新。而 Adagrad 会为每个参数自适应地调整学习率，其核心思想是：对于那些在训练过程中频繁更新的参数，减小其学习率；对于那些很少更新的参数，则增大其学习率。这样可以使得模型在训练过程中更加关注那些不经常出现的特征。

Adagrad 能够自动适应不同参数的更新频率，对于稀疏数据表现出色，例如文本数据中的词频统计，因为在稀疏数据中，不同特征的更新频率差异较大。在处理稀疏特征时，那些不常出现的特征对应的参数由于累积的梯度平方和较小，会得到较大的学习率，从而可以更有效地学习这些特征。

在 BigDL DLLib 库中，仅提供了 Adagrad 优化器的 Scala 版本实现，其定义如下：

```
val adagrad = new Adagrad(learningRate = 1e-3,
                      learningRateDecay = 0.0,
                      weightDecay = 0.0)
```

重构上一节的示例代码，演示如何使用 Adagrad 优化算法来最小化一个目标函数。重复的部分省略，重构部分代码如下：

```
// 损失函数定义相同
```

```
// ...
```

这段代码的核心是使用 Adagrad 优化器对目标函数 rosenBrock 进行 10 次迭代优化，每次迭代都会依据目标函数的梯度更新输入张量 x 的值，从而逐步找到目标函数的最小值。

执行以上代码，输出内容如下：

```

```

关于更多优化器 API 的说明，请参考官方文档。

12.9 正则化器

在机器学习和深度学习中，过拟合是一个常见问题。当模型在训练数据上表现良好，但在未见过的测试数据上表现不佳时，就可能出现了过拟合。过拟合通常是由于模型过于复杂，学习到了训练数据中的噪声和异常值，而不是数据的真实模式。正则化就是为了解决这个问题而提出的。

在 BigDL 的 DLlib 库中，正则化器（Regularizer）是一种用于防止模型过拟合的技术手段。Regularizer 是一种对模型参数进行约束的机制，它通过在损失函数中添加一个额外的正则化项（惩罚项），来限制模型参数的取值范围或复杂度。这样可以使模型更加简单、泛化能力更强，从而减少过拟合的风险。

在每次更新模型参数时，优化器不仅要考虑原始损失函数的梯度，还要考虑正则化项的梯度。这样，模型在学习数据特征的同时，也会受到正则化项的约束，避免参数取值过大或过于复杂。

在 BigDL DLlib 库中，提供了 L1 正则化器、L2 正则化器和 L1L2 正则化器。

12.9.1 L1 正则化器

L1 正则化器用于在 gradWeight 中添加惩罚，以避免过拟合。在 BigDL DLlib 库的代码实现中，gradWeight = gradWeight + alpha * abs（weight）。L1 正则化会使权重向量中的一些元素变为零，从而实现特征选择的效果。

L1 正则化器的定义如下：

```
val l1Regularizer = L1Regularizer(rate)
```

可以在定义模型的层时指定 L1 正则化器。下面这段代码演示了如何创建一个线性层，进行前向传播得到输出，以及进行反向传播计算输入的梯度。同时，还展示了如何使用 L1 正则化来约束线性层的权重和偏置，以防止模型过拟合。代码如下：

```
//
```

12.9.2 L2 正则化器

L2 正则化器用于在 gradWeight 中添加惩罚，以避免过拟合。在 BigDL DLlib 库的代码实现中，gradWeight = gradWeight + alpha * weight * weight。L2 正则化会使权重向量的元素值变小，但不会使其变为零。

L1 正则化器的定义如下：

```
val l2Regularizer = L2Regularizer(rate)
```

可以在定义模型的层时指定 L2 正则化器。下面这段代码展示了如何创建一个带有 L2 正则化的线性层，进行前向传播得到输出，以及进行反向传播计算输入的梯度。通过使用 L2 正则化，可以约束线性层的权重和偏置，避免模型过拟合，从而提高模型的泛化能力。代码如下：

```
//
```

12.9.3 L1L2 正则化器

L1L2 正则化器用于在 gradWeight 中添加惩罚，以避免过拟合。在 BigDL DLlib 库的代码实现中，将依次应用 L1 正则化器和 L2 正则化器。

L1L2 正则化器的定义如下：

```
val l1l2Regularizer = L1L2Regularizer(l1rate, l2rate)
```

可以在定义模型的层时指定 L1L2 正则化器。下面这段代码展示了如何创建一个带有 L1L2 正则化的线性层，进行前向传播得到输出，以及进行反向传播计算输入的梯度，是神经网络训练过程中的基本步骤。代码如下：

```
//
```

以上代码中，L1L2Regularizer(0.2, 0.2) 是一种结合了 L1 和 L2 正则化的正则化器。L1 正则化（也称为 Lasso 正则化）会在损失函数中添加权重的绝对值之和作为惩罚项，倾向于使部分权重变为零，从而实现特征选择的效果；L2 正则化（也称为 Ridge 正则化）会添加权重的平方和作为惩罚项，使权重的值尽量小但不会为零。这里的两个参数 0.2 分别表示 L1 和 L2 正则化的系数，即惩罚项的权重。

12.10 学习率调度器

学习率是深度学习优化算法中的一个关键超参数，它决定了在每次参数更新时，模型参数移动的步长大小。若学习率过大，模型参数更新幅度会很大，可能导致模型无法收敛，在最优解附近来回震荡，甚至发散；若学习率过小，模型参数更新幅度小，收敛速度会变得缓慢，训练时间大幅增加，还可能陷入局部最优解。因此，动态调整学习率十分必要。

在 BigDL dllib 库中，Learning Rate Scheduler（学习率调度器）是一个重要的组件，它能够根据训练的不同阶段动态地调整学习率。

Learning Rate Scheduler（学习率调度器）可以依据预定义的规则或训练过程中的某些指标（如训练轮数、验证集损失等），在训练过程中动态地改变学习率。这样做的好处是，在训练初期可以使用较大的学习率，让模型快速收敛；而在训练后期，逐渐减小学习率，使模型能够更精确地逼近最优解。

接下来我们介绍 BigDL 中一些常用的学习率调度器（更多的调度器说明，请参考官方文档）。

12.10.1 Default 学习率调度器

这是默认的学习率计划。对于每次迭代，学习率将按照以下公式更新：

$$l\_\{n + 1\} = l / (1 + n * learning\_rate\_decay)$$

其中 l 是初始学习率。

default 学习率调度器的定义如下：

```
Default()
```

以下这段代码展示了如何使用 BigDL 的 SGD 优化器对一个简单的目标函数进行优化。代码如下：

```
//
```

在优化迭代过程中，在 optimize()方法内部，优化器会多次调用 feval 函数来获取目标函数值和梯度，然后根据 SGD 算法的规则，结合学习率和动量，对参数向量 x 进行更新。经过多次迭代后，模型参数会逐渐收敛到一个较优的解。

12.10.2 Poly 学习率调度器

一种学习率衰减策略，其中有效学习率遵循多项式衰减，通过 max_iteration 衰减为零。poly 学习率调度器基于多项式函数来调整学习率。其核心思想是在训练开始时使用较大的学习率，使得模型能够快速地朝着最优解的大致方向前进；随着训练的进行，逐渐减小学习率，以便模型能够更精细地调整参数，从而更准确地收敛到最优解。

多项式学习率调度器的学习率调整公式通常为：

$$iter\_lr = base\_lr\ (1 - iter/maxIteration)\ ^\wedge\ (power)$$

其中：

（1）iter_lr：是第 t 轮训练时的学习率。

（2）base_lr：是初始学习率，即训练开始时设定的学习率。

（3）iter：当前的训练轮数 t。

（4）maxIteration：总的训练轮数。

（5）power：是多项式的幂次，它控制着学习率下降的速度。当 p 值较大时，学习率下降得更快；当 p 值较小时，学习率下降得相对较慢。

poly 学习率调度器的定义如下：

```
Poly(power=0.5, maxIteration=1000)
```

其中各参数的含义如下：

（1）power：衰减系数（即多项式的幂次），请参照计算公式。

（2）maxIteration：当学习率 lr 为零时的最大迭代次数。

下面这段代码展示了如何使用 BigDL dllib 中的 SGD 优化器和 Poly 学习率调度器进行参数优化。不过，打印出的学习率为负数可能表示代码中存在问题，需要检查学习率调度器的参数设置和计算逻辑。代码如下：

```
//
```

poly 学习率调度器的主要作用可总结为以下两点：

（1）快速收敛：在训练初期，较大的学习率可以使模型参数快速更新，加快模型的收敛速度，使模型能够迅速学习到数据的大致特征。

（2）精细调整：随着训练的推进，较小的学习率有助于模型在接近最优解时进行更精细的参数调整，避免因为学习率过大而错过最优解，从而提高模型的泛化能力。

12.10.3 Plateau 学习率调度器

Plateau 学习率调度器是 BigDL dllib 中一种用于动态调整学习率的机制。其核心思想是当模型在验证集上的性能（如损失值、准确率等）在一段时间内没有显著提升时，降低学习率，以此帮助模型跳出局部最优解，继续向全局最优解收敛。

Plateau 学习率调度器会监控一个指定的指标（通常是验证集上的损失值），当该指标在连续多个训练周期（称为 patience number of epochs）内没有改善时，就会将学习率乘以一个衰减因子（通常将学习率降低 2-10 倍）。这种策略能够在模型训练进入瓶颈期时，通过降低学习率来进行更精细的参数调整，避免模型陷入局部最优。

Plateau 学习率调度器的定义如下：

```
Plateau(monitor="score", factor=0.1, patience=10, mode="min", epsilon=1e-
4f, cooldown=0, minLr=0)
```

其中各参数的含义如下：

（1）monitor：需要监控的指标，例如验证集的损失值或者准确率。

（2）factor：学习率衰减因子，当指标没有改善时，学习率会乘以这个因子：new_lr = lr * factor。

（3）patience：容忍的周期数，即指标没有改善的连续周期数达到这个值时，才会降低学习率。

（4）mode：{min， max}中的一个。在 min 模式下，当被监测的指标停止下降时，lr 将减小；在 max 模式下，当监控的指标停止增加时，它将减少。

（5）epsilon：判断指标是否改善的阈值。

（6）cooldown：学习率调整后，需要等待的冷却周期数，在此期间不进行学习率的调整。

（7）minLr：学习率的下限，当学习率降低到这个值时，不再继续降低。

下面这段代码展示了如何使用 BigDL dllib 中的 SGD 优化器和 Plateau 学习率调度器进行参数优化。代码如下：

```
//
```

12.10.4 Exponential 学习率调度器

在 BigDL dllib 中，Exponential（指数）学习率调度器是一种动态调整学习率的机制，它能让学习率随训练轮数的增加呈指数级衰减。这种方式有助于模型在训练初期快速收敛，而在后期进行更精细的参数调整，避免错过最优解。

指数学习率调度器依据以下公式来调整学习率：

$$lr_{n + 1} = lr * decayRate ^ (iter / decayStep)$$

其中：

（1）lr_{n + 1}：第 n+1 轮训练时的学习率。

（2）lr：初始学习率。

（3）decayRate：衰减速率，有时也称为衰减因子，取值范围通常在 0 到 1 之间。

（4）iter：当前的训练轮次数。

（5）decayStep：衰减的时间间隔。

Exponential（指数）学习率调度器的定义如下：

```
Exponential(10, 0.96)
```

其中参数的含义如下：

（1）第一个参数 10 通常代表一个步长或者一个周期的长度，不过在不同的实现中含义可能会有差异。在指数学习率调度的典型场景下，它可能意味着每经过 10 个训练周期（epoch），学习率就会进行一次调整。

（2）第二个参数 0.96 是衰减因子。每到调整学习率的时机，学习率就会乘以这个衰减因子。例如，初始学习率是 0.05，经过 10 个训练周期后，新的学习率会变为 0.05 * 0.96；再经过 10 个周期，学习率会变为 0.05 * 0.96 * 0.96，以此类推。

下面这段代码创建了一个初始学习率为 0.05 的 SGD 优化器，并且为其设置了一个指数学习率调度器，该调度器会每 10 个训练周期将学习率乘以 0.96 进行衰减。代码如下：

```
//
```

12.11 模型冻结

“冻结”一个模型意味着从训练中排除模型的某些层。它指的是在训练过程中，固定模型中部分或全部参数，使其在训练时不进行更新。通常，深度学习模型的训练是通过反向传播算法来计算损失函数相对于模型参数的梯度，然后根据这些梯度更新参数。而当对某些层或整个模型进行冻结后，在反向传播过程中，这些被冻结的参数对应的梯度将被忽略，从而不会更新这些参数的值。

在处理大规模模型时，训练所有参数会消耗大量的计算资源和时间。通过冻结部分参数，可以减少需要计算梯度和更新的参数数量，从而显著降低计算开销，加快训练速度。另外，在某些情况下，模型的部分层可能已经学习到了稳定的特征表示，继续训练这些层可能会导致过拟合。通过冻结这些层，可以避免过拟合的发生，使模型更加稳定和泛化能力更强。

在迁移学习场景中，通常会使用预训练模型。预训练模型已经在大规模数据集上进行了训练，学习到了一些通用的特征表示。当将预训练模型应用到新的任务时，可以冻结模型的部分层（如特征提取层），只训练与新任务相关的层（如分类层）。这样可以利用预训练模型的知识，同时避免从头开始训练整个模型，提高模型在新任务上的性能和训练效率。

可以通过调用 freeze()方法来“冻结”模型。如果一个模型被冻结，它的参数（权重/偏差，如果存在）在训练过程中不会改变。如果传递了层的名称，那么与给定名称匹配的层将被冻结。用法如下：

```
model.freeze("layer1", "layer2")
```

相对应的，整个模型可以通过调用 unFreeze()方法来“解冻”。如果提供了层名称，那么与给定名称匹配的层将被解冻。用法如下：

```
model.unFreeze("layer1", "layer2")
```

下面这个示例中，使用原始的未被“冻结”的模型。代码如下：

```

```

执行以上代码，此时的输出内容如下：

```
```

现在，如果“冻结”模型的 fc2 层，那么 fc2 的参数将不会被改变。代码如下：

```
```

执行以上代码，此时的输出内容如下：

```
```

如果“解冻”模型（或模型的 fc2 层），那么 fc2 的参数就会被改变，代码如下：

```
```

执行以上代码，此时的输出内容如下：

```
```

12.12 使用 TensorBoard 可视化训练

有了生成的摘要信息，我们就可以使用 TensorBoard 来可视化 BigDL 程序的行为。

1. 安装 TensorBoard

需要满足以下前提条件：

（1）Python 版本：2.7，3.4，3.5，3.6。

（2）Pip 版本>= 9.0.1。

要使用 Python 2 安装 TensorBoard，可以运行以下命令：

```
pip install tensorboard==2.19.0
```

要使用 Python 3 安装 TensorBoard，可以运行以下命令：

```
pip3 install tensorboard==2.19.0
```

2. 启动 TensorBoard

可以使用下面的命令启动 TensorBoard：

```
tensorboard --logdir=/home/hduser/bigdl-log/bigdl_summaries --bind_all
```

之后，使用浏览器导航到 TensorBoard 仪表板。TensorBoard 成功启动后，可以在控制台输出中找到 URL；默认 URL 为 http://localhost:6006。

3. 在 TensorBoard 中的可视化

在 TensorBoard 仪表板中，将能够读取每次运行的可视化，包括 SCALARS 选项卡下的“Loss”和“Throughput”曲线（如下图所示）：

以及 DISTRIBUTIONS（分布）和 HISTOGRAMS（直方图）选项卡下的“weights”、“bias”、“gradientWeights”和“gradientBias”（如下图所示）：

12.13 案例：预测糖尿病的发病

在本案例中，我们将描述如何使用 Apache Spark Dataframes 来扩展分布式深度学习的数据处理。

12.13.1 创建 Scala 项目

要使用 BigDL DLLib 构建自己的深度学习应用程序，可以使用 Maven 或 SBT 创建项目并将 bigdl-dllib 添加到依赖项中。

1. 使用 Maven 构建项目

如果使用 Maven 创建项目，则需要将如下代码添加到 pom.xml 中以添加 BigDL DLLib 依赖：

```
<dependency>
   <groupId>com.intel.analytics.bigdl</groupId>
   <artifactId>bigdl-dllib-spark_3.1.3</artifactId>
   <version>2.4.0</version>
</dependency>
```

目前，BigDL 的每日构建（Nightly Build）版本是托管在 SonaType 上的。要将我们的应用程序与最新的 BigDL 每日构建版本链接起来，应该添加一些像正式版本一样的依赖项，但将 2.4.0 更改为快照版本（例如 2.5.0-snapshot），并将以下存储库添加到 pom.xml 中：

```
<repository>
   <id>sonatype</id>
   <name>sonatype repository</name>
   <url>https://oss.sonatype.org/content/groups/public/</url>
   <releases>
      <enabled>true</enabled>
   </releases>
   <snapshots>
      <enabled>true</enabled>
   </snapshots>
</repository>
```

2. 使用 SBT 构建项目

如果使用 SBT 创建项目，则需要将如下代码添加到 build.sbt 中以添加 BigDL DLLib 依赖：

```
libraryDependencies += "com.intel.analytics.bigdl" %% "bigdl-dllib-spark"
 % "2.4.0"
```

如果使用 BigDL 的每日构建（Nightly Build）版本，开发人员可以使用下面这样的配置：

```
resolvers += "ossrh repository" at "https://oss.sonatype.org/content/repo
sitories/snapshots/"
```

3. 使用 Zeppelin 开发项目

如果使用 Zeppelin 来开发的话，则需要额外的配置。建议使用小白学苑提供的 PBLP（个人大数据学习平台），所有环境已经配置好了，读者只需要关注应用即可。本书随附的参考代码均在 PBLP 上调试运行过，并且以 Zeppelin 类型的文件提供，读者直接加载到 Zeppelin 中打开即可。

12.13.2 初始代码

建议在程序开始时初始化 NNContext，它的作用是在 BigDL 中封装一个 SparkContext：

```
import com.intel.analytics.bigdl.dllib.NNContext
import com.intel.analytics.bigdl.dllib.keras.Model
import com.intel.analytics.bigdl.dllib.keras.models.Models
import com.intel.analytics.bigdl.dllib.keras.optimizers._
import com.intel.analytics.bigdl.dllib.keras.layers._
import com.intel.analytics.bigdl.dllib.nn.ClassNLLCriterion
import com.intel.analytics.bigdl.dllib.utils.Shape
import com.intel.analytics.bigdl.numeric.NumericFloat
import com.intel.analytics.bigdl.dllib.tensor.TensorNumericMath.TensorNum
eric.NumericFloat

import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.sql.functions.
import org.apache.spark.sql.types.DoubleType

// 创建或获取具有 BigDL 性能优化配置的 SparkContext。
val sc = NNContext.initNNContext("dllib_demo")
```

NNContext.initNNContext() 方法用来创建或获取具有 BigDL 性能优化配置的 SparkContext。该方法还将初始化 BigDL 引擎。

NNContext.initNNContext()方法的源码定义如下：

```
  /**
   * @param conf User defined Spark conf
   * @param appName name of the current context
   * @return Spark Context
   */
  def initNNContext(conf: SparkConf, appName: String): SparkContext = {
    val zooConf = createSparkConf(conf)
    initConf(zooConf)

    if (appName != null) {
      zooConf.setAppName(appName)
    }
    if (zooConf.getBoolean("spark.analytics.zoo.versionCheck", defaultValu
e = false)) {
      val reportWarning =
        zooConf.getBoolean("spark.analytics.zoo.versionCheck.warning", def
aultValue = false)
      checkSparkVersion(reportWarning)
      checkScalaVersion(reportWarning)
    }
    val sc = SparkContext.getOrCreate(zooConf)
    Engine.init
    sc
```

```
  }
```

注意：

如果使用 spark-shell 或 Jupyter 笔记本，因为 Spark 上下文（SparkContext）创建在你的 Spark 代码之前，因此必须通过命令行选项或属性文件设置 Spark 配置值，并手动初始化 BigDL 引擎。因为本示例使用 Zeppelin Notebook 开发，所以我们的初始代码如下：

```
import com.intel.analytics.bigdl.dllib.keras.Model
import com.intel.analytics.bigdl.dllib.keras.models.Models
import com.intel.analytics.bigdl.dllib.keras.optimizers._
import com.intel.analytics.bigdl.dllib.keras.layers._
import com.intel.analytics.bigdl.dllib.nn.ClassNLLCriterion
import com.intel.analytics.bigdl.dllib.utils.Shape
import com.intel.analytics.bigdl.numeric.NumericFloat
import com.intel.analytics.bigdl.dllib.tensor.TensorNumericMath.TensorNum
eric.NumericFloat
import com.intel.analytics.bigdl.dllib.utils.Engine

import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.sql.functions.
import org.apache.spark.sql.types.DoubleType

// 初始化 BigDL 引擎
Engine.init(3,2,onSpark=true) // 3 个节点，每个节点有 2 个核心，以集群模式运行
// 或
// Engine.init
```

在 BigDL dllib 应用程序里，Engine.init() 是个关键的初始化方法，调用它是为了对运行环境和相关资源进行初始化配置。Engine.init() 方法的主要功能之一就是初始化 Spark 上下文（SparkContext），它可以对分布式计算环境进行自动配置，确保 BigDL dllib 应用程序能在集群环境下高效运行。它会根据系统的资源情况和用户的配置，对分区数量、内存分配等参数进行合理设置，以此提升计算性能和资源利用率。

当调用 Engine.init(numNodes, coresPerNode, onSpark = true)方法时，BigDL 会读取 Spark 的配置，将 spark.executor.instances 值赋予参数 numNodes，将 spark.executor.cores 值赋予参数 coresPerNode。

12.13.3 分布式数据加载

DLlib 支持 Spark DataFrame 作为分布式训练的输入和分布式推理的输入/输出。因此，用户可以使用 Apache Spark 轻松处理大规模数据集，并直接在分布式（可能在内存中）数据框架上应用 AI 模型，而无需进行数据转换或序列化。

皮马印第安人糖尿病数据集最初来自国家糖尿病、消化和肾脏疾病研究所，包含来自美国亚利桑那州凤凰城附近人口的 768 名妇女的信息。结果检测为糖尿病，258 例检测阳性，500 例检测阴性。因此，有 8 个属性和一个目标（因变量）变量，说明如下：

```

......
```

我们可以使用 Spark API 将数据加载到 Spark DataFrame 中，例如，读取皮马印第安人糖尿病数据集（csv 文件）到 Spark DataFrame 中，代码如下：

```
```

执行以上代码，输出内容如下：

查看数据量及数据模式，代码如下：

```
print("数据量: ")
println(df.count)

println("数据模式: ")
df.printSchema
```

执行以上代码，输出内容如下：

模型的特征列应该是一个 Spark ML Vector 向量。请将相关的列组合成一个向量并传递给模型。如。

```
```

执行以上代码，输出内容如下：

然后将 Spark DataFrame 拆分为训练部分和验证部分，代码如下：

```
```

12.13.4 模型定义

下面这段代码的主要功能是使用 Keras 风格的 API 构建一个简单的三层全连接神经网络模型。全连接神经网络也被称作多层感知机（Multilayer Perceptron, MLP），由多个全连接层（Dense Layer）组成。这个模型接收一个长度为 8 的一维输入向量，经过两层带有 ReLU 激活函数（来引入非线性特性）的隐藏层，最后输出一个长度为 2 的向量。代码如下：

```
// x1 是一个代表输入层的对象，后续的层会以它为基础构建
val x1 = Input(Shape(8))

// 分别创建具有 12 个神经元、8 个神经元和 2 个神经元的全连接层
// 前两个隐藏层使用 ReLU 作为激活函数
// dense3 没有指定激活函数，意味着使用线性激活函数
val dense1 = Dense(12, activation="relu").inputs(x1)
val dense2 = Dense(8, activation="relu").inputs(dense1)
val dense3 = Dense(2).inputs(dense2)

// 创建一个完整的神经网络模型
// 该模型的输入是 x1（输入层），输出是 dense3（最终的全连接层）
val dmodel = Model(x1, dense3)
```

在构建好的完整神经网络模型 dmodel 之后，还必须决定在训练中使用哪个损失函数。然后可以使用模型的 compile()方法来设置损失函数和优化方法。compile()是 BigDL dllib 中模型对象的一个方法，它的功能是配置模型的训练参数。代码如下：

```
dmodel.compile(optimizer = new Adam(), loss = ClassNLLCriterion())
```

在深度学习里，构建好神经网络模型之后，需要对其进行编译，目的是配置训练过程所需的各种参数，像优化器、损失函数等。compile()方法的作用就是完成这个配置工作，为后续的模型训练做好准备。

优化器在深度学习训练过程中扮演着关键角色，其主要任务是根据损失函数的梯度来更新模型的参数，从而使损失函数的值不断减小，让模型的预测结果更加准确。Adam（Adaptive Moment Estimation）是一种常用的优化算法，在很多深度学习任务中，Adam 优化器都能表现出较好的性能，收敛速度较快，并且对超参数的选择相对不那么敏感。

损失函数用于衡量模型预测结果与真实标签之间的差异。在训练过程中，优化器的目标就是最小化这个损失函数的值，以此来提高模型的性能。ClassNLLCriterion 是负对数似然损失函数（Negative Log Likelihood Criterion），通常用于分类任务。在分类问题中，模型的输出一般是每个类别的概率分布，而 ClassNLLCriterion 会计算模型预测的概率分布与真实标签之间的负对数似然损失。这个损失函数能够鼓励模型对真实类别的预测概率尽可能接近 1，而对其他类别的预测概率尽可能接近 0。

完成编译之后，就可以使用 fit()方法对模型进行训练了。

12.13.5 分布式模型训练

现在可以使用 fit()方法开始训练。如果拟合的数据是使用 Spark API 生成的 DataFrame，还需要设置 featureCols 列。代码如下所示：

```
dmodel.fit(
    x=trainDF,
    batchSize=6,
    nbEpoch = 2,
    featureCols = Array("features"),
    labelCols = Array("label"),
    valX=valDF
)
```

以下 fit()方法中各参数的含义如下：

（1）x = trainDF： x 代表训练数据，通常是一个包含特征和标签的数据集合。

（2）batchSize = 6：batchSize 指的是每次训练时使用的样本数量，也就是一个批次（batch）中包含的样本数。注意，这个值应该是你的 Spark 集群的 spark.executor.instances 配置值的整数倍。

（3）nbEpoch = 2：nbEpoch 表示训练的轮数，也就是将整个训练数据集完整地输入到模型中进行训练的次数。

（4）eatureCols = Array("features")：featureCols 是一个数组，用于指定 DataFrame 中哪些列包含了模型的输入特征。

（5）labelCols = Array("label")：labelCols 是一个数组，用于指定 DataFrame 中哪些列包含了模型的真实标签。

（6）valX = valDF：valX 代表验证数据，通常是一个包含特征和标签的 DataFrame 对象。

在深度学习训练过程中，通常不会一次性将所有的训练数据都输入到模型中进行训练，而是将数据分成若干个批次，每次只使用一个批次的数据来计算损失函数和更新模型参数。

设置 batchSize = 6 表示每次训练时使用 6 个样本组成一个批次。较小的批次大小可以增加模型的随机性，有助于跳出局部最优解，但可能会导致训练速度较慢；较大的批次大小可以加快训练速度，但可能会使模型陷入局部最优。

每一轮训练都会让模型对训练数据进行一次全面的学习和参数更新。设置 nbEpoch = 2 意味着会将整个 trainDF 数据集输入到模型中进行 2 次训练。通常，增加训练轮数可以让模型更好地学习数据中的模式，但也可能会导致过拟合，即模型在训练数据上表现很好，但在新数据上表现不佳。

在训练过程中，除了使用训练数据来更新模型参数外，还会使用验证数据来评估模型在未见过的数据上的性能。通过在训练过程中定期对验证数据进行评估，可以监控模型的泛化能力，避免过拟合。在每一轮训练结束后，模型会在 valDF 上进行验证，计算相应的评估指标（如准确率、损失值等），以帮助用户判断模型的训练效果。

dmodel.fit()方法通过指定训练数据、批次大小、训练轮数、特征列、标签列和验证数据等参数，对模型进行训练和验证。这些参数的合理设置对于模型的训练效果和性能至关重要。

执行以上代码，此时会输出执行过程信息。如下图所示：

12.13.6 分布式评估与推断

训练结束后，可以使用训练好的模型进行预测或评估。

1）预测

预测也称为推断（inference），使用训练好的模型在验证数据集上进行。对于 Spark API 生成的 DataFrame 数据集，需要设置 featureCols 列，代码如下：

```
val result = dmodel.predict(
    trainDF,
    featureCols = Array("features"),
    predictionCol = "predict")

result.show(10, false)
```

执行以上代码，输出内容如下：

获得分类结果：

```
import org.apache.spark.sql.functions._

val resultWithClass = result.withColumn("predictedClass",
  when(size(col("predict")) > 0,
    argmax(col("predict")+1)
  ).otherwise(null)
)

resultWithClass.show(10, false)
```

withColumn 方法用于在 result DataFrame 中添加一个新列 predictedClass。

when(size(col("predict")) > 0, ...) 用来检查 predict 列的数组长度是否大于 0。

argmax(col("predict")) 函数会找出 predict 数组中最大值的索引，这个索引就是预测的类别。

otherwise(null) 表示如果 predict 数组长度不大于 0，就将 predictedClass 设为 null。

2）评估（evaluation）

根据验证结果，可以对模型进行评估。对于 Spark API 生成的 DataFrame，代码如下：

```
val ev_result = dmodel.evaluate(
    trainDF,
    batchSize = 6,
    featureCols = Array("features"),
    labelCols = Array("label")
)

println("评估结果: ", ev_result(0)._1)

println("平均损失值 loss 和样本数: " + ev_result(0)._1.result())
```

执行以上代码，输出内容如下：

```
(评估结果: ,(Loss: -45196.406, count: 631, Average Loss: -71.62663))
平均损失值 loss 和样本数: (-71.62663,631)
```

12.13.7 模型保存和加载

训练完成后，可能需要保存最终模型以供以后使用。BigDL 允许将 BigDL 模型保存在本地文件系统、HDFS 或 Amazon s3 上。

例如，将训练好的模型 dmodel 保存到 HDFS 的/tmp/demo/keras.model 目录下，使用如下的代码：

```
val modelPath = "hdfs:///tmp/demo/keras.model"
dmodel.saveModel(modelPath)
```

如果后续需要使用该模型，则可以从保存的位置加载到内存中进行应用，代码如下：

```
// 加载存储的模型
val loadModel = Models.loadModel(modelPath)

// 使用该模型对验证数据进行预测
val predf = loadModel.predict(
    valDF,
    featureCols = Array("features"),
    predictionCol = "predict"
)

// 查看预测结果
predf.show(10, false)
```

执行以上代码，输出内容如下：

12.13.8 检查点和恢复训练

在 BigDL dllib 里，通过设置检查点和恢复训练功能，你可以在深度学习训练过程中更加灵活地管理模型的训练进度，避免因意外情况导致的训练进度丢失。

1. 设置检查点

检查点是在模型训练过程中定期保存模型的参数和状态的一种机制。其主要作用有：

（1）防止训练中断丢失进度：在训练过程中，可能会因为各种原因（如硬件故障、停电等）导致训练中断。通过保存检查点，在中断后可以从最近的检查点继续训练，避免从头开始训练浪费大量时间和计算资源。

（2）模型评估和选择：可以在不同的训练阶段保存检查点，然后对这些检查点对应的模型进行评估，选择性能最优的模型用于后续的预测任务。

在深度学习训练过程中，检查点（Checkpoint）和恢复训练是非常重要的功能。可以通过设置检查点来实现定期对模型进行快照。BigDL dllib 也提供了相应的机制来支持这些操作。可以通过调用模型对象上的 setCheckpoint()方法来设置检查点的保存路径和相关参数，代码如下：

```
val cpPath = "hdfs:///tmp/demo/cp"
dmodel.setCheckpoint(cpPath, overWrite=false)
```

其中，cpPath 参数是一个字符串类型的参数，表示检查点文件的保存路径。本例中指定 HDFS 分布式文件系统路径，模型的检查点文件将被保存到这个路径下。overWrite 参数是一个布尔类型的参数，默认值为 false。如果设置为 false，当检查点文件已经存在时，不会覆盖原文件；如果设置为 true，则会覆盖原文件。

也可以通过设置检查点保存频率来配置定期对模型进行快照。检查点保存频率指定每隔多少个批次（batch）或者轮次（epoch）保存一次模型快照。例如，设置每 1 个 epoch 保存一次检查点，示例代码如下：

```
// 设置检查点保存路径
val cpPath = "hdfs:///path/to/checkpoints"

// 设置每 1 个 epoch 保存一次检查点
val cpInterval = 1

// 设置是否覆盖已有检查点文件
val overwrite = false

// 配置模型定期保存快照
model.setCheckpoint(cpPath, overwrite, Trigger.everyEpoch(cpInterval))
```

除了按 epoch 保存检查点，还能按批次（batch）保存，示例如下：

```
// 设置每 100 个 batch 保存一次检查点
model.setCheckpoint(cpPath, overwrite, Trigger.everyBatch(100))
```

2. 恢复训练

训练停止后，可以从任何保存点恢复。恢复训练是指在训练中断后，从之前保存的检查点文件中加载模型的参数和状态，然后继续进行训练。这样可以在之前的训练进度基础上继续优化模型，而不需要从头开始。

选择一个要恢复的模型快照（保存在检查点路径中）。使用 Models.loadModel(path)将模型快照加载到模型对象中，示例代码如下：

```
val loadModel = Models.loadModel(path)
```

加载后的模型具有与保存时相同的参数和状态，可以继续进行训练或进行预测。在实际应用中，我们可以根据需要定期保存检查点，并在需要时从检查点恢复训练。

12.13.9 监控训练

在 BigDL dllib 中，TensorBoard 是一个强大的工具，用于监控和可视化模型的训练进度。它写入在训练/验证期间收集的统计信息。保存的摘要可以通过 TensorBoard 查看。

TensorBoard 原本是 TensorFlow 中的可视化工具，它可以将深度学习训练过程中的各种指标（如损失值、准确率等）、模型结构、参数分布等以直观的图表形式展示出来，帮助开发者更好地理解模型的训练情况，及时发现问题并进行调整。

在训练模型之前，需要指定 TensorBoard 日志的保存目录。这个目录将存储训练过程中的各种指标数据，供 TensorBoard 读取和可视化。为了使其生效，需要在 fit()之前调用它，代码如下：

```
// 指定 TensorBoard 日志保存目录
dmodel.setTensorBoard("/bigdl-log/bigdl_summaries", "mlp_demo")
```

在训练完成或训练过程中，可以启动 TensorBoard 服务器来查看可视化结果。在终端中执行以下命令：

```
tensorboard --logdir=/bigdl-log/bigdl_summaries --bind_all
```

其中，/bigdl-log/bigdl_summaries 是之前指定的 TensorBoard 日志保存目录。执行该命令后，会启动一个本地服务器，默认端口为 6006。

然后在浏览器中，访问 http://master-ip:6006，即可进入 TensorBoard 界面。在该界面中，可以看到常见的可视化内容，例如，loss 值和吞吐量（Throughput）随时间的变化曲线，如下图所示：

也可以切换到 Scalars（标量）标签页，查看训练过程中的各种标量指标，如损失值、准确率等随训练轮次或批次的变化曲线。通过观察这些曲线，可以判断模型是否收敛、是否过拟合等。

12.13.10 使用 Spark ML 管道

重构以上代码，使用 NNEstimator 和 Spark ML 管道来训练、预测和评估模型。

```

```

执行以上代码，输出内容如下：

12.14 案例：构建 CNN 模型识别图像

这里我们使用 BigDL 中 Keras 风格 API 来定义一个简化版的 CNN 模型，训练该模型识别猫狗图片。

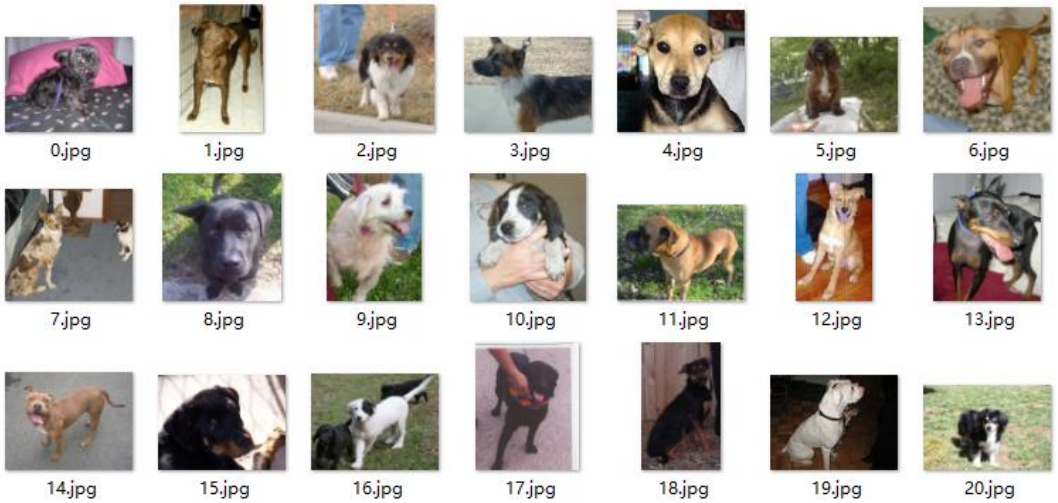
12.14.1 训练用图像数据集

Dogs vs. Cats 数据集是一个在计算机视觉领域具有重要地位的公开数据集，常被用于图像分类任务的研究与实践，尤其适合初学者入门深度学习中的图像分类问题。

该数据集最初是由 Kaggle 平台在 2013 年举办的一场图像分类竞赛中提供的，竞赛要求参与者构建一个算法，对给定的图像进行分类，判断图像中的主体是狗还是猫。此竞赛吸引了众多数据科学家和机器学习爱好者参与，推动了图像分类技术的发展。

整个数据集共包含 25000 张图像，其中狗和猫的图像各有 12500 张，图像主要为 JPEG 格式。这些图像来源广泛，涵盖了不同品种的狗和猫，以及它们在各种环境中的姿态和表情，并且具有不同的分辨率、尺寸、拍摄角度和光照条件，这增加了模型训练的挑战性。

以下为部分狗的图像：



以下为部分猫的图像：

读者可以从 Kaggle 官方网站下载该数据集，但需要注册 Kaggle 账号并遵守相关的使用条款。此外，一些开源的机器学习库和平台也可能提供该数据集的下载接口。本书配套提供有该数据集，如果需要，可以向作者索要。另外，小白学苑的 PBLP2005 平台中已经预置了该数据集，使用 PBLP2005 平台学习的读者可直接使用。

在使用数据集时，需要进行适当的数据预处理，如调整图像大小、归一化等，以提高模型的训练效果。同时，要注意数据集的划分，一般将训练集进一步划分为训练集和验证集，用于模型的训练和评估。

12.13.2 初始代码

本案例我们使用 Zeppelin Notebook 开发。因为 Spark 上下文（SparkContext）创建 Spark 代码之前，因此必须通过命令行选项或属性文件设置 Spark 配置值，并手动初始化 BigDL 引擎。初始代码如下：

```
```

在 BigDL dllib 应用程序里，Engine.init() 是个关键的初始化方法，调用它是为了对运行环境和相关资源进行初始化配置。Engine.init() 方法的主要功能之一就是初始化 Spark 上下文（SparkContext），它可以对分布式计算环境进行自动配置，确保 BigDL dllib 应用程序能在集群环境下高效运行。它会根据系统的资源情况和用户的配置，对分区数量、内存分配等参数进行合理设置，以此提升计算性能和资源利用率。

当调用 Engine.init(numNodes, coresPerNode, onSpark = true)方法时，BigDL 会读取 Spark 的配置，将 spark.executor.instances 值赋予参数 numNodes，将 spark.executor.cores 值赋予参数 coresPerNode。

12. 14. 3 分布式数据加载

对于图像数据，可以使用 com.intel.analytics.bigdl.dllib.nnframes 包中的 NNImageReader 类来加载。对于 NNImageReader 生成的 DataFrame，不需要设置 featureCols 列。如果需要对图像进行预处理，可以设置 transformers。

我们使用 BigDL 的 NNImageReader API 将数据加载到 Spark DataFrame 中。因为图像的标签值在文件名中，所以我们需要将其解析出来，并添加到 DataFrame 的 label 列。为此，首先定义一个用来解析文件名和标签值的 UDF，代码如下：

```
```

执行以上代码，输出内容如下：

12. 14. 4 定义和训练 CNN 模型

在这里，我们使用类 Keras 的 API 来定义一个简化版的 CNN 模型，并在猫狗数据集上训练它。

我们把模型创建封装到一个方法中，代码如下：

```
// 定义创建模型的方法

```

代码说明：

（1）Conv2D：这表示要添加的是一个二维卷积层。参数 32 指定了该卷积层中卷积核的数量，表示该卷积层的输出通道数，意味着该层会提取 32 种不同的特征。后面两个参数（5,5）分别指定了卷积核的高度和宽度。这里卷积核的大小是 5x5，即每个卷积核是一个 5

行 5 列的矩阵。inputShape 指定输入数据的形状，它应该是一个预先定义好的元组，例如 (channels, height, width,) 表示图像的通道数、高度和宽度。

（2）Activation("relu")：添加一个 ReLU（Rectified Linear Unit）激活函数层。

（3）MaxPooling2D：二维最大池化层，用于降低特征图的空间尺寸，减少计算量，并增强模型的平移不变性。参数 poolSize = (2, 2)指定池化窗口的大小为 2x2，即在每个 2x2 的区域内取最大值作为输出。

（4）Flatten()：将前面卷积层和池化层输出的多维特征图展平为一维向量，以便输入到后续的全连接层。

（5）Dropout(0.5)：Dropout 层用于防止过拟合。0.5 表示在训练过程中，每个神经元有 50% 的概率被随机丢弃，即不参与当前训练步骤的计算。

（6）Dense(1)：最后一个全连接层有 1 个神经元，通常用于二分类任务。

（7）Activation("sigmoid")：sigmoid 激活函数，将输出转换为概率分布，使得所有输出值之和为 1，每个值表示样本属于对应类别的概率。

这个 CNN 模型通过卷积层提取图像特征，池化层降低特征图尺寸，全连接层进行分类决策，最终输出样本属于 2 个类别的概率。

接下来，我们调用这个 buildModel()方法，创建 CNN 模型，并打印模型摘要信息。代码如下：

```
// 定义模型
val model = buildMode(Shape(3, 150, 150))

// 在模型定义后打印摘要
println(model.summary())
```

执行以上代码，输出内容如下：

然后定义损失函数和优化方法，编译模型并拟合训练集数据对模型进行训练，代码如下：

```

```

12.14.5 预测和评估 LeNet CNN 模型

训练结束后，可以使用训练好的模型进行预测或评估。

1. 预测

预测也称为推断（inference），使用训练好的模型在验证数据集上进行。

对于 NNImageReader 生成的 DataFrame，不需要设置 featureCols。如果需要对图像进行预处理，可以设置 transform。下面使用训练过的模型 model 对测试集进行预测，代码如下：

```
val result = model.predict(trainingDF,  predictionCol = "predict", transfo
rm = transformers)
result.show()
```

执行以上代码，输出内容如下：

2. 评估

可以调用模型对象的 evaluate 方法，对模型进行评估。与预测类型，对于 NNImageReader 生成的 DataFrame，不需要设置 featureCols。如果需要对图像进行预处理，可以设置 transform。代码如下：

```
model.evaluate(testImageDF,
    batchSize = 12,
    labelCols = Array("label"),
    transform = transformers)

println(metrics(0)._2 + " : " + metrics(0)._1)
println(metrics(1)._2 + " : " + metrics(1)._1)
```

执行以上代码，输出内容如下：

```
Loss : (Loss: 303.94122, count: 21, Average Loss: 14.473392)
Top1Accuracy : Accuracy(correct: 10, count: 21, accuracy: 0.47619047619047616)
```

附录

本附录中将补充 Spark 开发环境搭建等内容。

附录 1. 配套代码和数据下载

本书配套提供了全部的示例代码及所用到的数据。购买本书的读者请联系作者获取。需要注意的是，本书配套代码均使用 Zeppelin Notebook 开发，因此需要加载到 Zeppelin Notebook 中查看和运行。当然，如果想在其它开发环境中测试，请自行实现。

附录 2. Spark 练习环境下载

要验证配所讲的示例代码，读者需要有 Spark 的运行环境以及机器学习代码的开发环境。为了让读者尽快上手练习，免除搭建环境的各种繁琐与困难，本书配套提供了“个人大数据学习平台-PBLP”，下载即可直接进行本书中所有示例代码的运行和练习。该平台可从小白学苑网站下载。