****



**研 究 生 毕 业 论 文**

**（申请工程硕士学位）**

|  |  |
| --- | --- |
| **论文题目** | 基于Spark的分布式特征处理工具  集的设计与实现 |
| **作者姓名** | 蒲瑜琦 |
| **学科、专业名称** | 工程硕士(软件工程方向) |
| **研究方向** | 软件工程 |
| **指导教师** | 葛季栋　副教授 |

**2019 年 3月**  **9日**

**学 号： MF1732106**

**论文答辩日期： 年 月 日**

**指 导 教 师： （签字）**

**基于Spark的分布式特征提取工具库的**

**设计与实现**

|  |  |
| --- | --- |
| **作 者:** | **蒲瑜琦** |
| 指导教师: | **葛季栋 副教授** |

|  |
| --- |
| **南京大学研究生毕业论文** |
| **(申请工程硕士学位)** |

|  |
| --- |
| **南京大学软件学院** |
| **2019年05月** |

**The Design and Implementation of Distributed Feature Processing Tools Library Based on Spark**

**Pu, Yuqi**

**Submitted in partial fulfillment of the requirements for the degree of Master of Engineering**

Supervised by

Professor Jidong GE

Software Institute

**NANJING UNIVERSITY**

Nanjing, China

May, 2019

# 摘 要

随着互联网经济的发展，线上服务会产生巨量的业务数据，对这些数据的储存、传输和使用，在传统单机模式下十分困难，其主要难点在于单机处理速度难以进一步拓展和提升。对于数据挖掘任务，处理特征和训练模型也会因为受到硬件条件的限制而受到制约。为了应对这些问题，分布式存储和分布式计算框架被广泛使用。

分布式计算很好的解决了计算能力拓展的问题，但是原有的机器学习算法和特征处理算法基本是为了单机环境设计的，没有考虑过并发/分布式环境，所以原有的算法没有办法充分发挥分布式环境的性能优势。因此本文将阐述四类特征处理算法在分布式计算环境中的设计思路、实现细节和应用。

本文选择Spark平台作为实现并行算法的基础，Spark作为目前最受欢迎的分布式计算框架，在企业中被广泛使用。我们将算法的分布式实现集成在一个分布式特征处理算法工具库中，为用户提供支持。

本文针对数据挖掘任务中遇到的特征处理困难，对一部分典型的处理算法进行了分布式实现。需要解决的问题可以分为三大类：数据分布不平衡、连续特征的离散化、离散特征的相关信息提取和压缩编码。针对不同的问题，提出相应的解决方案。

本项目再设计并实现了SMOTE，因子分解机，ChiMerge和高势集特征编码的分布式版本，并对算法的效果和性能进行了测试。

**关键词**：因子分解机、过采样、特征离散化、分布式计算

# Abstract

With the development of the Internet economy, online services generate huge amounts of business data. The storage, transmission and usage of these data are very difficult in the traditional stand-alone mode. The main difficulty is that the processing speed of the single machine is difficult to further expand and improve. For data mining tasks, processing features and training models are also constrained by hardware. To cope with these problems, distributed storage and distributed computing frameworks are widely used.

Distributed computing solves the problem of computing power expansion. However, the original machine learning algorithm and feature processing algorithm are basically designed for single-machine environment. The usage scenario of distributed resources has not been considered, so the original algorithm cannot be applied directly to take advantage of the benefits of a distributed resource. Therefore, this thesis will expound the redesign ideas, implementation details and applications of the four types of feature processing algorithms in the distributed computing environment.

This thesis chooses the Spark platform as the basis for implementing parallel algorithms. Spark is currently the most popular distributed computing framework and is widely used in enterprises. We integrate the distributed implementation of the algorithms into a distributed feature processing algorithm tool library to support the user.

In this thesis, for difficult encountered in data mining tasks, some classic processing algorithms are redesigned and implemented for distributed environment. The problems that need to be solved can be divided into three categories: imbalanced data distribution, discretization of continuous features, interaction information extraction of discrete features, and reduction coding for high cardinality categorcal feature.

This thesis redesigned and implemented a distributed version of SMOTE-based oversampleing, Factorization Machine, ChiMerge and high-cardinality feature coding, and tested the effect and performance of these implements.

**Keywords**： Factorization Machine、Oversampling、Discretization、Distributed Computing

**目 录**

[**摘 要 I**](#_Toc3094251)

[**Abstract II**](#_Toc3094252)

[**图目录 V**](#_Toc3094253)

[**表目录 VII**](#_Toc3094254)

[**第一章 引言 1**](#_Toc3094255)

[**1.1 行业背景 1**](#_Toc3094256)

[**1.2 项目背景 3**](#_Toc3094257)

[**1.3 本文主要研究的工作 3**](#_Toc3094258)

[**1.4 本文的组织结构 4**](#_Toc3094259)

[**第二章 技术综述 6**](#_Toc3094260)

[**2.1 特征工程的简介及其必要性 6**](#_Toc3094261)

[**2.2 Apache Spark运行原理的简介 7**](#_Toc3094262)

[**2.3 过采样算法原理的简介 11**](#_Toc3094263)

[**2.4 因子分解机算法原理简介 14**](#_Toc3094264)

[**2.5 特征离散化、最优分桶原理简介 17**](#_Toc3094265)

[**2.6 高势集特征编码简介 20**](#_Toc3094266)

[**2.7 本章小结 22**](#_Toc3094267)

[**第三章 分布式特征处理工具库的需求分析与设计 23**](#_Toc3094268)

[**3.1项目总体规划 23**](#_Toc3094269)

[**3.2 系统需求分析 23**](#_Toc3094270)

[**3.2.1系统用例分析 23**](#_Toc3094271)

[**3.2.2样本处理需求分析 24**](#_Toc3094272)

[**3.2.3特征处理需求分析 25**](#_Toc3094273)

[**3.2.3非功能性需求分析 26**](#_Toc3094274)

[**3.3系统总体设计与模块设计 27**](#_Toc3094275)

[**3.3.1总体结构 27**](#_Toc3094276)

[**3.3.2过采样模块设计 28**](#_Toc3094277)

[**3.3.3因子分解机模块设计 31**](#_Toc3094278)

[**3.3.4特征离散化与高势集模块设计 33**](#_Toc3094279)

[**3.4 本章小结 35**](#_Toc3094280)

[**第四章 分布式样本特征处理工具库的实现 36**](#_Toc3094281)

[**4.1过采样模块的实现 36**](#_Toc3094282)

[**4.1.1 SMOTE 36**](#_Toc3094283)

[**4.1.2 borderline-SMOTE 42**](#_Toc3094284)

[**4.2特征相关性提取的实现 45**](#_Toc3094285)

[**4.2.1因子分解机随机梯度下降策略的实现 45**](#_Toc3094286)

[**4.2.2因子分解机并行梯度下降策略的实现 49**](#_Toc3094287)

[**4.2.3自适应梯度下降算法的实现 51**](#_Toc3094288)

[**4.3离散化模块实现 54**](#_Toc3094289)

[**4.4高势集特征编码模块实现 58**](#_Toc3094290)

[**4.4本章小结 59**](#_Toc3094291)

[**第五章 测试结果展示 60**](#_Toc3094292)

[**5.1 精度测试 60**](#_Toc3094293)

[**5.1.1 过采样精度测试 60**](#_Toc3094294)

[**5.1.2 因子分解机精度测试 63**](#_Toc3094295)

[**5.1.3 特征离散化模块精度测试 66**](#_Toc3094296)

[**5.1.4 高势集模块精度测试 67**](#_Toc3094297)

[**5.2 压力测试 69**](#_Toc3094298)

[**5.2.1 过采样算法压力测试测试 69**](#_Toc3094299)

[**5.2.2 因子分解机压力测试 72**](#_Toc3094300)

[**5.2.3 特征离散化压力测试 75**](#_Toc3094301)

[**5.2.3 高势集特征编码模块压力测试 76**](#_Toc3094302)

[**第六章 总结与展望 78**](#_Toc3094303)

[5.1 总结 78](#_Toc3094304)

[5.1 进一步工作展望 78](#_Toc3094305)

[参 考 文 献 79](#_Toc3094306)

[致 谢 83](#_Toc3094307)

[版权及论文原创性说明 84](#_Toc3094308)

# 图目录

[图2.1 Spark架构图 7](#_Toc3093909)

[图2.2 Spark物理执行图 8](#_Toc3093910)

[图2.3 Shuffle原理图 10](#_Toc3093911)

[图2.4 borderline算法生成样本分布对比图 14](#_Toc3093912)

[图2.5 用户推荐特征 15](#_Toc3093913)

[图2.6 因子分解机相关性权重生成 16](#_Toc3093914)

[图2.7 两个分箱区间的分布差异对比 18](#_Toc3093915)

[图2.8 切分点产生在同类样本区间内部 19](#_Toc3093916)

[图3.1系统用例图 24](#_Toc3093917)

[图3.2系统整体结构图 27](#_Toc3093918)

[图3.3 过采样模块类图设计 28](#_Toc3093919)

[图4.1 Spark物理执行图示 36](#_Toc3093920)

[图4.2 SMOTE算法物理执行图 37](#_Toc3093921)

[图4.3 筛选样本集中少类样本代码 37](#_Toc3093922)

[图4.4 样本间距离计算代码 38](#_Toc3093923)

[图4.5 LSH优化代码代码 38](#_Toc3093924)

[图4.6 局部敏感[Datar, 2004]哈希加速近邻查找物理执行图 39](#_Toc3093925)

[图4.7 本地查找K近邻代码代码 40](#_Toc3093926)

[图4.8合成全局近邻代码 41](#_Toc3093927)

[图4.9 合成新样本代码 41](#_Toc3093928)

[图4.10 borderline算法物理执行图 42](#_Toc3093929)

[图4.11 DANGERset生成代码 43](#_Toc3093930)

[图4.12 本地近邻队列维护代码 44](#_Toc3093931)

[图4.13 全局信息汇总代码 44](#_Toc3093932)

[图4.14 边界判断代码 45](#_Toc3093933)

[图4.15 borderline2新样本生成代码 45](#_Toc3093934)

[图4.16 随机梯度下降训练因子分解机单步迭代物理执行图 46](#_Toc3093935)

[图4.17 启动任务代码 47](#_Toc3093936)

[图4.18 SGD权重更新代码 48](#_Toc3093937)

[图4.19 二次项梯度计算代码 48](#_Toc3093938)

[图4.20 PredictAndSum代码 49](#_Toc3093939)

[图4.21 PGD物理执行图 50](#_Toc3093940)

[图4.22 并行梯度下降核心代码代码 51](#_Toc3093941)

[图4.23 自适应梯度下降算法物理执行图 52](#_Toc3093942)

[图4.24正则项系数训练代码 53](#_Toc3093943)

[图4.25估计模型代码 53](#_Toc3093944)

[图4.26二次正则项系数梯度计算部分代码 54](#_Toc3093945)

[图4.27 ChiMerge单步合并物理执行图 54](#_Toc3093946)

[图4.28区间合并基础数据结构 54](#_Toc3093947)

[图4.29 partition内查找最小卡方值代码 55](#_Toc3093948)

[图4.30 全局查找与合并代码 56](#_Toc3093949)

[图4.31 启发式优化方法代码 57](#_Toc3093950)

[图4.32 边界检测代码 57](#_Toc3093951)

[图4.33 统计样本分布代码 58](#_Toc3093952)

[图4.34 后验概率计算代码代码 59](#_Toc3093953)

# 表目录

[表 3.1消息队列客户端诊断需求描述 26](#_Toc3087225)

[表 3.2服务状态监控需求描述 27](#_Toc3087226)

[表3.3 过采样参数说明 30](#_Toc3087227)

[表3.4 因子分解机参数说明 33](#_Toc3087228)

[表3.5 特征离散化模块参数说明 35](#_Toc3087229)

[表3.6 ChiMerge造成的异常分布 36](#_Toc3087230)

[表3.7 高势集特征参数说明 37](#_Toc3087231)

[表5.1 过采样精度测试数据集 62](#_Toc3087232)

[表5.2 第一组实验参数设置 62](#_Toc3087233)

[表5.3 Abalone数据集上的实验结果 62](#_Toc3087234)

[表5.4 Yeast数据集上实验结果 63](#_Toc3087235)

[表5.5 第二组实验参数设置 63](#_Toc3087236)

[表5.6 第二组实验在Abalone上的实验结果 63](#_Toc3087237)

[表5.7 第二组实验在Yeast上的实验结果 63](#_Toc3087238)

[表5.8 borderline效果对比实验参数设置 64](#_Toc3087239)

[表5.9 Borderline在Abalone上的实验结果 64](#_Toc3087240)

[表5.10 Borderline在Yeast数据集上的实验结果 64](#_Toc3087241)

[表5.11 因子分解机算法的精度测试数据集信息 65](#_Toc3087242)

[表5.12 因子分解机精度测试的参数设置。 65](#_Toc3087243)

[表5.13 LibFM 的实验结果 66](#_Toc3087244)

[表5.14 FM on Spark实验结果 67](#_Toc3087245)

[表5.15 对比两种sgda的实验结果 67](#_Toc3087246)

[表5.16 FM on Spark实验结果 67](#_Toc3087247)

[表5.17 离散化模块精度测试 68](#_Toc3087248)

[表5.18 分箱算法精度测试对比-1 68](#_Toc3087249)

[表5.19 分箱算法精度测试对比-1 69](#_Toc3087250)

[表5.20 高势集精度测试数据集信息 70](#_Toc3087251)

[表5.21 高势集精度测试固定参数信息 70](#_Toc3087252)

[表5.22 高势集精度测试实验结果 71](#_Toc3087253)

[表5.23 硬件环境 71](#_Toc3087254)

[表5.24 样本数量压力测试结果 72](#_Toc3087255)

[表5.25 样本维度压力测试结果 72](#_Toc3087256)

[表5.26 关于SMOTE用时与少类占比的关系 72](#_Toc3087257)

[表5.27 关于SMOTE用时与核数cores的关系 73](#_Toc3087258)

[表5.28 Somte样本维度极限与Executor内存的关系 73](#_Toc3087259)

[表5.29 Somte样本维度极限与Driver内存的关系 73](#_Toc3087260)

[表5.30 FM压力测试样本数量与时间的关系 74](#_Toc3087261)

[表5.32 FM压力测试因子数与时间的关系 75](#_Toc3087262)

[表5.34 executor-memory和特征数量之间的关系和极限 76](#_Toc3087263)

[表5.35 driver内存和Factor数量之间的关系和极限 76](#_Toc3087264)

[表5.36 ChiMerge压力测试样本数量与时间的关系 77](#_Toc3087265)

[表5.38 高势集特征编码模块压力样本与时间的关系 78](#_Toc3087266)

[表5.39 高势集特征编码模块压力测试势集度与时间的关系 79](#_Toc3087267)

# 第一章 引言

## 1.1 行业背景

随着深度学习的兴起，图形和自然语言处理领域得到了迅速的发展，在图像处理方面，卷积网络CNN[Krizhevsky, 2012]和生成式对抗网络GAN[Goodfellow, 2014] 等模型的使用对图像处理业务效果有显著提升，逐渐代替了传统的机器学习方法在此领域的应用，成为了图像处理行业的主流框架。在自然语言处理领域，长短期记忆神经元LSTM[Graves, 2012]及其变体也被广泛应用于众多领域：机器翻译，词性标注，分词，语音系统等。在这些领域的应用让人们看到了神经网络的潜力。

深度学习确实可以学习到更加复杂的分布，但是同时也存在一些问题：深度学习的训练难度相对于传统机器学习算法要困难的多，需要大量的大维度的矩阵运算，这些运算在传统的微处理器架构中难以得到很好的并发支持，最重要的是神经网络的结构过于复杂，黑盒模型难以解释，在很多场景下，尤其是在注重分析和解释的领域中，失去了对特征的认识，无法了解特征的重要性。

传统机器学习方法在很多领域依然是主流应用，例如金融风控、量化分析数据挖掘，推荐系统，广告点击预测。传统机器学习算法也在不断的发展和改进，LightGBM[Guolin Ke, 2012]等新的学习框架实现也在众多竞赛中取得不错的成绩。传统机器学习的应用场景依然众多，因为简单的模型，对于轻量任务易于部署和验证特征工程的成果；白箱模型，方便从业人员解释解释，调整和分析；可扩展，可以做大规模分布式扩展，也可以应对小规模任务单机运行。

在很多情况下，特征之间的关系是业务相关的。因为法律法规的保护，一些数据是无法作为特征而被利用的，例如特征涉及用户隐私保护，在很多地方甚至为避免性别歧视，性别是无法直接作为特征而被使用的。除了外界环境对特征使用的影响，数据本身也存在很多病态条件阻碍学习任务的进行。例如离散特征取值过多，使哑编码会产生维数灾难，无法正常使用模型训练；数据分布不平衡导致模型学习不到样本中的少数类特征，例如在医疗诊断领域病变样本相对于健康样本数量非常少，在这种情况下不加处理直接使用模型训练进行诊断，会使病例诊断的召回率非常低，降低诊断的可靠性，造成误诊。

在很多应用领域，并不总是将连续值作为特征直接输入给模型，更常见的做法是将连续特征值离散化为离散特征。离散特征有很多优势，离散化的数据经过压缩易于存储、计算和传输，Spark[Zaharia, 2010]对离散特征有较好的支持。离散化后的特征对异常样本具有很强的鲁棒性；广义线性模型对于解决异或问题存在一定困难,单个变量离散化为多个离散特征后，每个变量有各自的特征权重，有助于模型拟合原始连续值数据中的非线性分布，例如人的年龄是连续特征，但是幼儿(<12)和老年人(>70)在行为能力、消费习惯等方面具有很强的相似性，青壮年的体征也与老年和幼儿有很大的差异，连续特征值在实际中的效果不一定是递增可加的，要考虑数值的取值范围以及周期性变化；离散后的特征可以进行特征交叉，异或问题也可以通过对离散化特征进行交叉组合形成新的离散特征而得到一定程度上的解决；特征离散化后，模型会更稳定，比如用户年龄的离散化，一个年龄区间更能反映人的生理和心理特征，而且可以压缩信息，方便存储和传输，但是要注意离散区间是否合理，不合理的区间划分不仅无法帮助稳定模型，还可能产生噪声使模型发生欠拟合，无法学习到真实分布；特征离散化以后，简化模型，模型不需要学习很多复杂的非线性分布，降低了模型过拟合的风险。

用简单模型训练大量离散特征与用复杂模型拟合少量连续特征都是数据挖掘和机器学习任务的可行办法。

对于数据离散化的思路可以粗略分为两种：无监督方法和有监督方法。

无监督方法不考虑样本的标签分布。最普遍两种方法，等宽分箱法和等频分箱法就是无监督分箱。等宽分箱法的每个离散区间宽度相同。等频分箱法的每个离散区间的包含样本的数量相同。等宽分箱需要使用者预先设置区间宽度的参数，而等频分箱需要使用者预先设置分位点，找到合适的宽度和分位点参数需要进行重复实验。区间过小离散区间数量过多，会使维度膨胀，区间过大离散区间数量过小，模型没有办法从中提取到有效信息导致欠拟合。以上两种方法都是最易实现的方法，不需要对训练样本集进行统计训练。

有监督方法需要知道标签的分布，根据分布训练得到最优的离散化模型。根据离散指标的选择可以分为基于卡方值的离散和基于熵的离散方法。根据离散策略的选择可以分成融合法又称自底向上和分裂法又称自顶向下。

本文引用的ChiMerge离散算法和基于最小描述长度[Rissanen, 2010]离散。这两的方法分别使用了不同的离散度量指标和离散策略。Chimerge使用卡方值对离散度进行度量，并且使用融合法作为离散策略。本片所提到的MDLP离散算法使用了基于信息熵的离散度量，最小描述长度是对信息熵改进后产生的度量指标。

## 1.2 项目背景

星环是国内一家提供大数据、数据分析与机器学习基础软件服务的开发公司，其产品致力于整合存储、分析和数据挖掘等任务。数据分析与挖掘的功能被整合到了Sophon中。

Transwarp Sophon是一款通用的人工智能平台，以帮助企业级用户在分布式环境中使用机器学习和数据挖掘的最新工具。Sophon结合大数据平台和机器学习平台，用户可以通过自动建模和内置的模板进行业务分析和机器学习任务，从而提升业务价值。

本文涉及的项目主要是数据挖掘平台的数据预处理和特征提取模块

Sophon Base作为交互式数据分析挖掘平台，可完成数据导入、数据探索与预览、数据预处理、特征工程、算法选择、模型训练、模型发布、模型管理及分享等工作流程。

本项目是针对一系列数据预处理和特征提取算法的分布式实现，作为数据预处理和特征提取工具库，为sophon平台的数据预处理和特征过程模块提供支持。

## 1.3 本文主要研究的工作

课题来源于本人在星环科技sophon算法研发部的实习经历。企业用户需要在分布式环境中对数据进行业务分析和处理。单机版本的特征处理算法会需要将所有需要的数据传输到本地计算中进行计算处理。这种做法难以并发，而且在数据规模庞大的情况下传输数据的代价太大。Spark是Apache下的开源分布式计算框架，可以与大多数云计算平台结合使用（Azure,Cloudera,EC2）。但是Spark中对特征处理的算法的支持并不全面，一些高级算法没有被加入到其机器学习模块Spark ML中。用户想要在分布式环境中使用这些算法比较困难。因此为了可以让没有开发基础的用户在web界面快捷的使用这些数据处理算法，我在星环算法部门实习期间，完成了特征提取和预处理相关高级算法的工具集的开发。这个项目的目的是为了补充Spark ML模块中缺失的算法，ml中已经有很多基础算法。本项目的根据用户对数据处理常遇到的四个困难场景，选取系列算法作为解决方案，这些算法在很多应用中取得了优秀的表现，本项目的主要目的是对于这些算法的分布式版本的再设计和实现，项目本身也需要更多的扩展。

本项目是针对机器学习任务中一些棘手的应用场景，覆盖特征提取的所有领域并不是项目的主要目标。针对机器学习任务中的数据不平衡问题；针对离散特征之间的相关性提取的问题；针对连续特征离散化困难问题；针对高实际离散特征容易导致维度灾难的问题。

本项目的主要研究内容是相关特征提取算法的总体设计和实现细节。首先讨论Spark框架的特性和原理。总结分布式算法的设计要点和性能瓶颈。

描述原算法的设计思想和实现细节，阐述项目的总体设计架构，分别阐述每一类算法的分布式设计要点，和对原算法改进要点。

展示算法在测试数据上的效果，主要从两个方面进行展示：从算法准确度方面，验证算法的正确性，确保与原算法取得相近的效果，对比使用指定特征处理算法和未使用指定特征处理算法的准确度和稳定性。从运行效率方面：对算法进行压力测试，测试数据量的增长对算法时间复杂的的影响，测试运行节点的增加对时间复杂度的影响，测试特征维度的增加对算法时间复杂度的影响。展示相关测试数据，总结项目的优势和不足。对每种算法的使用场景进行详细的介绍，例如使用场景，处理规模，所能够支持的并发度。

## 1.4 本文的组织结构

本文主要说明分布式特征提取的设计和实现，阐述这些算法的应用场景和应用效果。

第一章 绪论部分 主要介绍了特征处理算法在行业中的使用和发展，行业背景、项目背景、国内外研究现状。

第二章 技术综述 对过采样算法SMOTE[Chawla, 2002]，因子分解机[Rendle, 2011]，特征离散化[Sergio, 2016]和高实际特征编码[Micci-Barreca, 2001]算法进行技术概述，阐述算法原理和性能分析，以及所应用相关技术。Spark分布式计算框架的基本结构和运行机制。

第三章 分布式特征处理算法工具库的设计与需求分析 对用户需求进行分析，介绍分布式特征处理算法工具库的总体结构和算法实现难点要点，定义算法参数。

第四章 分布式特征处理算法工具库的实现 介绍每一个算法的分布式实现工作流、相对单机算法的改进点。展示代码，介绍具体实现细节。

第五章 测试结果展示 展示精度测试结果和压力测试结果，对测试结果进行分析，阐述所遇到的问题以及如何解决这些问题。

第六章 总结与展望 对论文期间的工作进行总结，以及对分布式特征提取算法在行业中的应用做一个展望。

# 第二章 技术综述

## 2.1 特征工程的简介及其必要性

特征工程是指从原始数据中提取有价值信息，进行特征清洗和转换，帮助机器学习训练模型和预测的过程。特征工程是机器学习中最重要的起始步骤，会直接影响机器学习的效果，并通常需要大量的人力物力成本发现有价值的特征。典型的特征工程包括数据清洗、降维、特征提取、特征选择等过程。

常见的数据清洗的方法：缺失值补全、特征尺度变换（如归一化）、去除异常值（极端离群值）、平滑处理和离散化等。特征清洗是特征工程的前期工作，归一化的目的是为了消除量纲对后续训练的影响，虽然大部分机器学习模型都需要做标准化和归一化，但是也有部分模型可以不用进行归一化和标准化而不显著影响训练效果，主要是基于树模型，例如分类回归树CART[Loh, 2011]、随机森林和boosting，这些基于树模型的算法会从原始的特征值中寻找切分点；缺失值补全、数据平滑和去除极端离群值都是为了降低和避免数据噪声对模型训练的影响；特征离散化的目的是多方面的，在上一章中已经提到，主要是为了非线性特征的提取和为后续的特征交叉做准备。

特征选择是挑选原始数据集中有价值的变量或者说是与当前机器学习任务相关的变量作为学习特征。原始数据中普遍存在冗余特征或是无关特征，冗余特征是指特征所提供的信息在其他特征中可以体现出来，例如城市分类和邮政编码是同一类信息，可以选择其中一个作为特征。无关特征是与其他特征和标签无关的特征，特征之间的相关程度可以用相关系数和协方差进行衡量，如果一个特征在训练集中的值只是一个取值，那么该特征对当前学习任务是没有贡献的，我们认为它是无关特征。特征的相关程度可以通过相关系数矩阵或者协方差矩阵进行衡量。根据经验总结，良好的特征通常包括一下几点：离散特征值的出现频率不能太低；特征的定义不应随时间发生变化；对于连续特征最好有明确的上下界。

特征组合又称特征交叉，是将多个特征集合进行笛卡儿积，形成的一系列特征组合，可以用来提取特征空间中的非线性规律。

## 2.2 Apache Spark运行原理的简介

Apache Spark是一个开源的分布式计算框架，最初是由加州大学AMPLab所开发。Spark默认是基于内存进行运算，在数据尚未写到磁盘上是将结果计算出来。所以，Spark在基于内存时的运算速度能做到比Hadoop MapReduce的运算速度快上100倍，即便是运行程序于硬盘时，Spark也能快上10倍速度[Xin, 2012]。Spark允许用户将数据加载至集群内存并对其进行查询，非常适合用于机器学习算法和数据分析处理。

Spark支持独立模式（本地Spark集群）、Hadoop YARN或Apache Mesos的集群管理。在分布式存储方面，Spark可以和HDFS、 Cassandra 、OpenStack Swift和Amazon S3等搭载配套使用。



图2.1 Spark架构图

Spark的核心数据结构是弹性分布式数据集RDD[Zaharia, 2012], Spark中所有的分布式计算任务都是面向RDD进行。RDD是一个抽象概念，用户所需要处理的数据并不是直接储存在一个RDD对象中，而是通过次级结构分区partition进行的。Partition也是Spark进行分布式计算的最小单位，也就是说partition之间可以并发计算，而一个partition内的数据计算是在一个core上串行的。一个RDD通常有多个partition组成，组成RDD的分区数量是由用户或是计算的上下文决定的。

Spark分布式计算中对RDD的操作有两种，Transformation和Action。Transformation操作会生成新的RDD，但是不会引起实质的计算。Action操作的输出是计算结果，会引发从数据源头的一系列由Transformation定义的运算，一个Action定义一个完整的Job执行逻辑。



图2.2 Spark物理执行图

在Job执行的时候，计算从数据源头或者是最邻近已被持久化的RDD开始计算流，action算子输出最后的计算结果。如果中间RDD没有做数据持久化的设置,则RDD所存储的数据及其占用的资源会在Job结束时被被释放（但是RDD对象依然存在）。Spark的这种性质被称作惰性运算，避免中间RDD在内存中长时间驻留，因为内存是很珍贵的存储资源。惰性计算和及时释放可以使内存资源得到最大化的利用。

如图2.2，一个Job由数个RDD和一个action算子组成，这数个RDD经过Transformation操作而彼此联系在一起，除了初始的RDD，每一个RDD都有一个或数个依赖Dependency，每一个依赖对应一个前驱RDD。

Spark中对Dependency的定义总体可以分为两种：NarrowDependency和ShuffleDependency。NarrowDependency又可分为RangeDependency和OneToOneDependency。当两个RDD之间存在ShuffleDependency时，就表示两个RDD之间的变换需要Shuffle操作。Shuffle操作需要对原有的RDD分区结构进行重新规划，非shuffle的变换可以保证原RDD中每个分区的完整性，shuffle操作会将原有的partition分解重新组合，因此这一步需要将分区碎片通过网络传输给指定节点，这一步也是很多分布式计算任务的瓶颈所在。

图 2.2定义了一个完整的Job,根据其中的ShuffleDependency可以将Job划分为若干个Stage，因此如果将Stage看作被切分的计算流，上游边界和下游边界均是ShuffleDependency。Task是定义在Stage内的执行单元，每个stage中的Task数量，由该Stage中的最后一个RDD的分区数量决定。一般情况下，同一个stage中的task在计算资源分配足够的情况下并发执行，具有依赖关系的stage顺序执行。

当用户通过Transformation操作生成新的RDD时，Tranformation操作会根据具体的语义，决定依赖的种类，例如对于map操作产生的MapPartitionsRDD会自动将其依赖设置为OneToOneDependency,而groupby中生成的ShuffledRDD会自动将其前驱依赖设置为ShuffleDependency。DAGScheduler将Job封装成事件提交到事件处理队列DAGSchedulerEventProcessLoop，等待处理结果，处理事件队列的线程取出队列中的事件进行处理，对于新提交的Job事件，scheduler沿依赖方向自底向上的递归构建stage。最后按照依赖关系递归的提交并执行所有尚未计算的stage，形成完整的计算流程(pipeline)。

Shuffle的过程分为两步，shuffle write和shuffle read。

Shuffle write的主要任务是将原RDD中的partition重新分区并持久化。Shuffle write 的具体实现是将Shuffle write的处理逻辑加入到ShuffleMapStage(ShuffleMapTask所在的Stage)的最后。该Stage的最后一个RDD（finalRDD）每输出一项记录(record)就按照新的分区规则分区并持久化，图示如下：



图2.3 Shuffle原理图

ShuffleMapTask的执行过程是：先根据pipeline计算得到finalRDD中对应的partition的records，没得到一个record就将其添加到对应的bucket中，对于bucket的选择由partitioner决定。每个bucket里面的数据会不断的被写到本地磁盘上，形成一个ShuffleFile。之后reducer会去fetch属于自己的ShuffleFile，进入shuffle read阶段。

Shuffle read阶段将上文中讲到的ShuffleFile取来，经行合并处理。要知道所需要ShuffleFile的具体位置需要通过访问driver端的MapOutputTrackerMaster。在一个ShuffleMapStage形成后，会将finalRDD注册到MapOutputTrackerMaster，MapOutputTrackerMaster会记录finalRDD中ShuffleFile的储存位置。

## 2.3 过采样算法原理的简介

近几年来，数据不平衡问题在数据挖掘领域获得了巨大的关注，不平衡数据集在很多领域都有存在，例如疾病诊断、识别电信诈骗、在卫星雷达图片上识别原油泄漏、信息检索和过滤任务等。在这些领域我们所关注的都是样本中的少数类，所以我们需要对这些少数类有更好的预测性能。但是传统的数据挖掘算法不平衡数据集上的表现并不好，因为很多算法在设计之初就没有考虑到数据集分布所带来的影响。

SMOTE(synthetic minority over-sampling technique)对于样本集中数量稀少的类别（下文简称为少类或少数类）也就是不平衡数据集，特征学习比较困难。可行的解决方法是将存在不平衡的训练样本集预处理为一般的训练样本集，将稀有的分类问题转化为普通的分类问题。SMOTE算法是通过生成少类样本改变不平衡条件的一种新方法，是一种比较基础的过度采样策略，具体就是通过对稀有类样本的人工合成来提高稀有类别的比例，这种过采样算法降低了样本数据的过度不平衡问题。SMOTE算法的特点在于不是简单的复制样例，而是生成一些原本不存在的样本来达到样本平衡，因此SMOTE可以在一定程度上可以防止学习器的过拟合[张振华, 2015]。

SMOTE方法的主要步骤如下：

算法

输入：T，少类样本的数量；N%，生成少类样本比例；k，邻近参数k

输出：合成的个少类样本。

1.如果N小于100，从T个少类样本随机采取样作为少类样本基准，

2.令;为样本的特征维度;为少类样本集合;

3.

利用k邻近算法计算第i个少类样本的同类最近的k个邻居样本点

从第i个少类样本中的k个邻近样本中有放的选取N个邻居样本

设被选中的邻近样本为Sample集合中的第j个样本（N次）:

令

令为长度为，取值在0到1之间，的独立随机向量

新生成的样本点

endfor

endfor

为了达到平衡样本的目的，还可以对多数类进行欠采样，即随机的去掉一部分多数类，使少数类在欠采样后的样本集中所占的比例增加，一旦比例达到所要求的阈值，就可停止过采样和欠采样过程。

borderline-SMOTE[Han, 2005]，为了使对少类样本的预测效果变得更好，大多数的分类算法试图在训练阶段尽可能的学习每类样本的分布边界。靠近边界的样本点相比与原理边界的样本点在分类时更容易出现误判，因此这些样本对分类任务更为重要。基于这种先验假设，我们认为那些远离边界的样本对学习任务提升的作用较小。因此borderline-SMOTE1和borderline-SMOTE2在原始SMOTE算法的基础上做了改进和提升。这两种改进算法的改进思想都是在少类样本的边界上进行过采样。而原始的SMOTE算法是对所有的少类样本点进行过采样。下面对borderline-SMOTE1和borderline-SMOTE2改进思路做详细阐述。

设完整训练集为T, P为少类样本集合, N为多类样本集合:

,, , pnum为少类样本数量，nnum为多类样本数量。

Borderline-SMOTE1：

步骤1. 对于少类样本集中的样本, 我们从完整的训练集计算各自的m个最近邻居。其中m个近邻中的多类样本数量为.

步骤2. 如果, 即m个邻近样本点中全为多类点，则样本会被当作噪音，并且不会参与后续计算。如果, 即多类样本占m邻近集合的多数，则被认为是容易被错误分类的并且将其放进集合DANGER中。如果, 则被认为是安全的，并且不参与后续计算。

步骤3. DANGER集合可以看作是少类样本点集合P的边界，且。令, 。对于DANGER集合中的每一个样本，我们从少类样本集P中计算k个最近邻居样本点。

步骤4. 在这一步中我们生成个新的少类样本,s为取值在1与k之间的一个整数。对于每一个,从其在上一步骤中计算得到的k个同类近邻中选取s个近邻样本。分别计算与其s个同类近邻样本间的距离：,并且生成0到1之间的随机数,最后少类样本在与最近邻居之间生成：

,

我们重复上述过程可以不断生成新的少类样本点，上述步骤中、、、和均为向量。

Borderline-SMOTE2不仅对DANGER集合里面的样本在少类样本集合中P中取k个同类近邻，还要在多类样本集合N中取k个不同类样本，但是对不同类样本随机数的取值范围在0-0.5之间，因此新生成的样本会更靠近原有少类样本集合。

2.4图中，(a)为原始数据分布，红色加号为少类样本；(b)中的实心蓝色方块代表我们从少类样本中找到的边界样本；(c)中的虚心蓝色方框为通过borderline-SMOTE算法生成的少类样本,可以清楚的观察到生成样本基本都分布在少类样本边界。

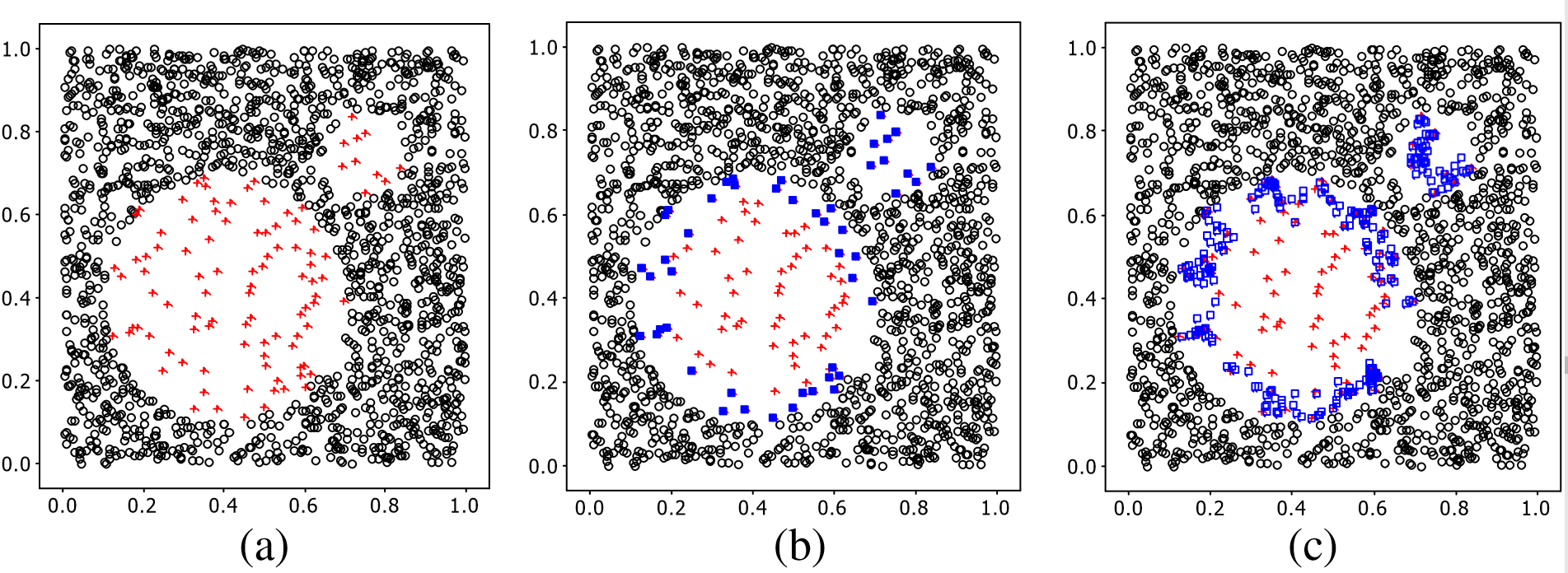


图2.4 borderline算法生成样本分布对比图

## 2.4 因子分解机算法原理简介

在工程领域，支持向量机(Support Vector Machines[Hearst, 1998])一直是机器学习和数据挖掘领域最受欢迎的模型。但是在数据集非常稀疏的条件下例如推荐系统中的协同过滤，很少会使用SVM作为预测器，一般使用更多的是矩阵分解模型。SVM在这些领域没用得到广泛应用，主要是由业务数据的特性造成的。在复杂的核空间中使用稀疏数据训练SVM无法训练得到稳定的参数。但同时传统的矩阵分解算法无法直接用于对新特征的预测而且通用性较差。

因子分解机(Factorization Machine,以下简称FM)是Steffen Rendle提出的模型。FM是一个通用的模型,任何数值特征向量都可以作为输入。与SVM相比FM通过被因子分解的参数对特征中的变量关系进行建模，这种方法对稀疏数据的中相关特征的学习相比于SVM有显著的提升。FM可以训练时间复杂度是成线性的，只与参数数量相关，可以计算梯度直接优化。而且FM可以在稀疏数据上估计出较为稳定的参数。

而且FM还可以模拟多种矩阵分解模型，例如SVD++[Koren, 2008]、PITF[Rendle, 2010]和FPMC[Tang, 2013]。这些矩阵分解模型的共同缺点是不能直接的被应用到预测任务当中，只能处理特殊数据。而且这些模型的模型结构和优化算法各异。FM只需要将输入数据转化为特征向量就可以对这些模型进行模拟。所以FM在用户没有专业领域知识的情况下仍然可以使用。

在有监督学习任务中，我们假定训练集,为训练集中的第i个样本,为第i个样本的特征向量，为第i个样本的标签值。数据集D可以被定义为分类数据集，也可以被定义为用于回归任务的数据集，对用户对产品的评分进行拟合。对于稀疏数据的中的特征向量x，其中的元素多数取值为0,我们定义为特征向量x中非零值的数量，为在D上的平均值。现实世界中很多特征都会以非常稀疏的特征向量的形式呈现。

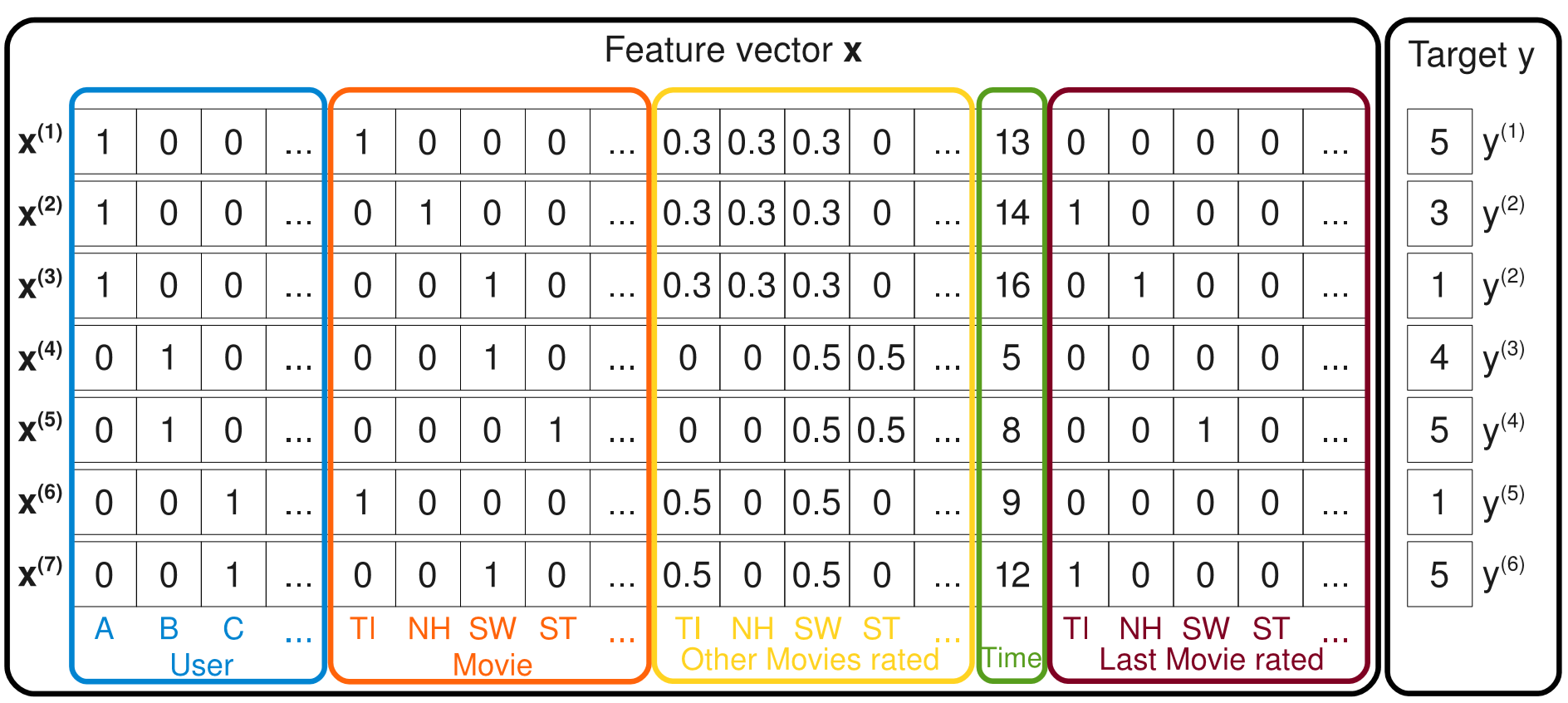


图2.5 用户推荐特征

如图2.5特征向量中的蓝色方框的指示变量代表用户特征，橙色变量代表当前待预测电影，黄色方框表示该用户所评论的其他电影，绿色指标为评价时间，褐色方框表示用户评价的上一部电影，最后一列代表训练用到的标签。

因子分解机模型，我们只考虑特征变量两两之间的相关度。模型公式为:

其中为全局偏移量,为特征向量的独立权重，为分解后的参数矩阵。为向量与的内积， 。是参数矩阵的第i行的行向量,以上参数均通过训练学习得到。K作为超参由使用者定义。

图2.6 因子分解机相关性权重生成

若令方阵中的第i行第j列的元素表示为,则,方阵表达了每项特征的相关性权重。即为第i项和第j项特征的相关权重。

并不难看出为半正定矩阵，k越大对相关性的拟合越好，但同时需要学习的参数的数量也在增多，训练时间也会增多。K应该被合理的控制和选择，在数据非常稀疏的条件下，超参数k设置过大还会导致模型过拟合。

在稀疏条件下的参数估计，往往缺少足够的数据来直接和独立的估计特征间的相互关系。FM通过对相关性参数矩阵的分解，打破了相关性参数间的独立性。

如图2.6 FM时间复杂度为,而且因为是稀疏数据集，所以式中的n可以退化成，所以时间复杂度变为。

FM的适用任务包括回归任务、二分类任务、和排序任务。

现在常用的方法有人工特征工程与逻辑回归(Logistic Regression[Andrew, 2012]，以下简称LR), 但是特征工程是件非常耗费人力的工作，不仅需要相关领域的专业知识而且还要通过实验对效果进行验证。因此近年来行业内有新的替代方案，去让训练过的模型预测结果最为特征扩展，集成学习的框架Stacking[Czarnowski, 2018]也是通过这种思路进行特征的堆砌，类似的框架有使用梯度提升树(Gradient Boosting Decision Tree，简称GBDT[Chen, 2016])叶子节点信息和FM预测值生成新特征。

FM的训练和生成可以直接用SGD进行，我们基于Spark分布式FM也式也是基于SGD算法进行优化。

为了证明FM算法在协同过滤应用中的广泛潜力，我们可以展示一个FM对SVD++模型进行模拟的例子。

FM算法的一些不足，针对FM的实现也有一些缺陷，就是基于SGD算法的回归任务，很容易就会导致梯度爆炸，模型失效。

我们可以发现在FM形成的相关性矩阵中，有一些权重是不必要的。针对推荐任务FFM[Juan, 2016]对FM做了进一步的改进。

## 2.5 特征离散化、最优分桶原理简介

在知识发现和数据挖掘领域，数据离散化是最有影响力的处理任务。很多数据挖掘算法也是基于离散型数据进行的，为了满足这种使用需求，各种数据离散化方法被提出。数据离散化可以调高特征信息存储效率，提高数据挖掘和机器学习任务的执行效率。根据数据特征和使用场景的不同，可以使用不同的离散化策略，本文介绍两种特征离散化算法及其在Apache Spark框架中的实现。

虽然特征离散化算法被普遍应用，但是具体的分桶细节也是因使用场景而异的，如果要让使用者因使用场景去手动调节分桶模型的参数，这无疑是一件很艰难的任务。而且错误的分桶模型参数，会给后续的学习任务带来严重的噪声。除非用户知道相关领域的知识，例如法定酗酒、驾车年龄，否则使用者很多难选出最优的分桶方法。

等宽分桶作为简单和最基础的离散化算法，可以在大多数情况下对数据进行很好的抽象。但是在很多场景下，其表现严重不足，例如数据中出现极端离群值的时候，会在离群值和数值的正常分布区域之间出现很多空区间，也就是说在离散化过程中，没有连续值或是很少落在这段区间内，造成模型空间浪费。等频分桶可以在一定程度上解决这个问题，但是也会引入新的问题，立群并不意味着数据本身有异常，也有可能是样本自身特征的表达，离群值会和正常值分在相同的区间内，从而掩盖了离群样本的特征。

ChiMerge最优分桶，离散化算法的最初目的除了从数据集中消除数值方便存储，还有就是对连续特征值所表达的信息做一个简短而又明确的总结概要。一个离散化区间是对这个区间内所有样本的整体概述。例如一个离散区间含有80个正类样本20个负类样本，那么对这个区间的样本概述就是80%正类20%负类。

所以我们所期望的离散化模型是区间内的子区间的分布保持一致。相邻区间之间的分布要保持显著差异，否则两个相邻并且样本分布相近的区间就应该合并。

我们将这种分布量化为卡方检验，计算相邻区间的卡方值。卡方值是检验假设的统计指标，区间分布和样本分布间相关的可能性就越大。



图2.7 两个分箱区间的分布差异对比

ChiMerge算法有两部分组成，初始化和自底向上的合并过程，当达到停止条件时终止合并。初始化阶段是对离散化目标特征进行排序，初始化离散区间模型，将单个连续特征值映射成由单个样本组成的区间。例如特征值a的初始化区间是[a,a],即左右区间均为a的单位区间。然后计算相邻区间的卡方值。选择卡方值最小的相邻区间进行合并，直到最小卡方值小于设定的参数卡方阈值，停止合并。

其中m=2，参与卡方计算的区间数量，k为样本类别数量

在第i个区间，第j种类别的样本数量。

在第i个区间的样本总数。

局部区间中第j个类别的总数。

局部区间的样本总数。

样本数量的期望。[Kerber, 1999]

使用卡方值作为合并指标有利于将参数用统计语言进行解释。例如当自由度为2时，取卡方阈值为4.6，则输出的离散区间和样本分布之间的关系可以表示为两个相邻区间所造成的差异至少有90%概率是和样本分布显著相关的。

ChiMerge使用自底向上的合并策略，使用卡方值作为合并指标，但是ChiMerge算法在实际应用中也有一些不足，会出现分桶不均，迭代过长的问题，这一点在第三章的实现中会详细说明。而基于最小描述长度(MDLP)离散化方法使用自顶向下的分裂法。

基于MDLP和自顶向下的分桶策略，离散模型的生成类似于CART的产生过程，算法在一个样本集合S的待切分连续特征A内,寻找最优切分点T，根据找到的切分位置将样本分为两个子样本集合，，再递归的进行切分操作，至到切分指标不在满足切分条件停止分桶输出模型。

对于切分指标，我们定义集合内部信息熵，。

对与一直最优切分点为T的样本集合S，切分形成的子样本集合为，。权重平均熵可表示为。这种信息熵可以作为切分指标，但是也存在一定问题，因为频繁指数运算，用信息熵作为分裂指标的计算量对每一个样本进行计算会消耗很多计算资源。直接基于熵的切分还会为了找到加权平均熵的最小值，会在同类样本区间内部制造一个最优切分点，这种切分明显不合理，如下图，因此算法仍然有优化空间，可以筛选出边界上的点作为候选切分点。



20个A类样本

30个C类样本

40个B类样本

100个A类样本

图2.8 切分点产生在同类样本区间内部

MDLP是奥卡姆剃刀形式化的结果，其主要思想是对于给定的数据，将数据使用一个编码模型进行编码，我们可以将模型和编码后的数据的储存空间看作是描述原有数据所表达信息的长度，也就是比特数。我们希望这个描述长度要尽可能的短，而且要能够表达编码前的完整信息。

在MDLP离散化算法中，我们将切分点的选择看作编码模型的选择，切分后形成的样本子集可以被当作编码后的数据。对于一个给定的候选切分点T，要定义两种假设，HT为接受当前候选切分点为最优切分点的假设，NT为拒绝当前样本点作为最优切分点的假设。对于NT假设我们需要传递的信息是对未经过切分的样本集合的整体描述，其中包括个编码后的样本和编码模型，编码模型描述了对每一个类型的编码，假设S集合中有k种类型的样本，对单个类型的平均编码长度为l，所以编码模型的总长度为。对于HT假设，需要从个样本间隙中选择一个位置作为最优样本切分点。所以对于这个位置的描述我们需要个比特位。切分后形成的两个样本子集的分布发生了变化，对单个类型的平均编码长度变为。所以编码后的数据长度为。接下来我们需要定义一个HT假设下，编码模型的长度计算规则。我们不仅需要计算切分后的样本子集对应的模型描述长度。我们也要对切分后两个样本子集的类型分布情况进行指定，即切分后哪些类型存在于中，哪些存在于中，通过计算我们得到可能的分区情况有种[Fayyad, 1999]。

因此分区情况的描述长度为。集合S中类别的平均描述长度l由信息熵Ent(S)定义。

所以得到对于拒绝切分假设NT的描述长度为。

对于接受切分假设HT的描述长度为。

MDLP的离散化策略：当且仅当时，接受切分假设。>,可以调节MDLP离散化的离散程度，防止过拟合[Fayyad, 1999]。

## 2.6 高势集特征编码简介

高势集离散特征是指那些离散特征取值范围过大的离散特征属性，例如邮政编码、用户手机号、ID。这些值即使时用数字表示，数值之间也没有明确关系，处理时更倾向将其作为用于识别每个实体的离散特征使用。线性判别模型需要特征最为数值输入到模型中，所以对离散特征需要做哑编码处理。但是这样做会使特征维度急剧膨胀而且形成的稀疏矩阵并不利于模型的学习和收敛。哑编码是处理低势集离散特征的常用方法，因为离散特征的取值不多，哑编码后增加的稀疏特征列位数也不多，不会显著影响模型性能。

在现实的应用场景中，高势集特征中会有很多特征值只在数据集中少量出现甚至只出现一次，例如邮编、电子邮件、IP地址和产品代码等。Daniele Micci-Barreca（2001）提出了一种对高势集特征进行特征编码的方法将离散特征空间映射到连续特征空间。这种连续特征既能起到减少特征维度的作用，而且转化为连续特征可以被线性模型使用。在分类场景下，转化后的连续特征取值与分类标签的后验条件概率有关，条件为离散特征取值。在回归场景下，转化后的连续特征取值与给定条件后标签的期望值有关。这种模式从统计学的角度将合理的，它可以保留原始离散特征的大部分预测所用到的信息。而且，对于具有结构化的离散信息例如邮编，这种高势集处理算法可以利用这些结构信息进行建模。

对于分类任务我们定义离散特征空间到连续特征空间的映射为,为离散特征取值，Y为标签值。为映射后的连续值，代表条件后验概率。对X到S的变换过程是通过经验贝叶斯实现。因此需要使用训练数据进行建模。

设训练集离散特征取值的样本数量为, 离散特征取值且标签值的样本数量为，则后验概率。但在实际的使用场景中取值为的样本数量可能会很小，即很小，导致对后验概率的估计不可靠。为了消除这种影响，文中建议加入先验概率来平衡这种不可靠后验概率的影响。即

其中为权重控制因子，用于平衡后验概率和先验概率，为标签为的样本总数，为训练集中样本总量。

k、f为超参数，且均大于零，需要使用者设置。K代表置信阈值，当n>k时，模型倾向于相信后验概率。当n<k时，即n的数量不足，模型更加相信先验概率。F为过度因子，用于控制后验和先验概率的过度速率。当f很小（接近零）时，后验概率和先验概率在n=k的地方时，过度区间很小忽略不计，可以之间当作硬阈值使用（a hard threshold）。

对于回归任务，模型定义如下：

为离散特征特征的样本集合。在回归任务中后验概率被替换为内所有样本的标签均值，先验概率被替换为总体样本的标签均值。

对于结构化的离散特征例如邮政编码，其编码的每一位都可以被当做相对独立的特征进行处理，例如我国邮政编码的前两位指定了省、自治区和直辖市，前三位数字表示邮区，前四位数字表示县（市），最后两位数字表示投递局（所）。

表示为特征值的前t位前缀，表示为前t位取值为的样本数量，其余定义不变。

结构化的离散特征中不提供有效信息的先验概率，被替换为上一层(t-1)的后验估计。这样做的好处是不在需要一个固定的先验估计，当本层次（t）的样本数量无法支持后验估计时，模型可以逐渐退化为更上层次的估计（t-1）而不是直接退化为不提供信息的先验估计。

## 2.7 本章小结

本章主要介绍开发分布式特征处理算法库所用到的技术。首先介绍了项目的定位，是用于支持数据挖掘任务的特征处理计算库，接着介绍了Spark的基本架构和运行原理，最后对4种特征处理算法经行了简单的介绍。

# 

# 第三章 分布式特征处理工具库的需求分析与设计

## 3.1项目总体规划

本项目是为了弥补大数据和机器学习平台用户在分布式环境中缺少新型的数据特征处理算法的不足。主要为数据科学、数据分析团队和企业用户使用。项目在保证原有算法处理效果正确性的基础上，提升算法在分布式平台上的可扩展性和并发处理能力。项目包括四个功能模块，数据离散化模块用于处理非线性连续特征、数据过采样模块用于解决数据不平衡，重要样本过少难以获得的问题、特征相关性提取模块用于解决离散特征之间相关特征的提取问题、特征压缩模块用于解决离散特征值过多导致维度灾难的问题。

## 3.2 系统需求分析

### 3.2.1系统用例分析

根据的背景和目标分析，分布式特征处理工具库面向的使用者为数据驱动行业分析人员。如系统用例图3.1所示，特征处理工具库有两个大类使用场景，针对数据样本的分布，和针对具体业务特征的特点。根据用户的反馈，样本分布主要出现的问题就是样本分布的不平衡(少类样本太少)和样本冗余(近似样本太多)。例如在反欺诈应用领域中，异常账户所占的比例要远小于正常账户数量，这会导致对于异常账户的特征学习不充分，导致预测任务的召回率偏低。针对具体业务特征的特点，其使用场景可以分为降维和特征信息提取。



图3.1系统用例图

### 3.2.2样本处理需求分析

使用者希望对算法输出的结果进行控制，两种输出情况第一种是只输出生成的少类样本集合，第二种是输出原有训练集合和新生成的少类样本集合的融合，第二种输出方式可以直接被模型作为输入进行训练。

用户在分布式环境下使用Spark ML中的机器学习模型进行业务预测。同时希望我们提供的算法可以在Spark ML pipeline中使用，我们构建的算法的输出和输出以及训练模型模板都是在Spark ML pipeline指定的统一的标准下进行的。Pipeline是Spark 1.2后引入的工作流概念，将设置好超参数的预测器或者转换器加入pipeline，pipeline会根据用户指定的机器学习流程，自动化提交并完成预处理、特征提取以及模型的学习和预测等各阶段任务。

为了满足输入输出统一的需求，我们使用Spark-SQL中的DataFrame作为预处理算法的输出和输出接口。同时要在算法中指定输出特征的格式。

过采样算法要具有可调节样本生成比例的超参数供用户设置。以及不同的生成策略供用户选择。

表 3.1消息队列客户端诊断需求描述

|  |  |
| --- | --- |
| **需求名称** | **需求描述** |
| 输入输出兼容 | 算法的输出要可以直接在Spark-ml pipeline中被使用，可以接收pipeline中模型的输出数据作为本功能的输入正常使用。 |
| 输出对象自定义 | 用户可以指定是否单独输出新生成的少类样本集合 |
| 要有可调节的超参数 | 用户可以根据实际的数据分布情况，按需要调节生成少类样本的比例或数量。 |

### 3.2.3特征处理需求分析

用户同样希望算法实现的输入和输出要在Spark-ML pipeline中兼容。

同时用户还需要模型参数具有可解释性，或者具有一定统计含义，能够被分析人员理解从而获取相关知识。

样本处理算法的输出的模式(Schema)是一致的。但是与样本处理算法不同，特征处理算法提取到的新特征会被加入到DataFrame的列末，从而改变schema。Schema相当于DataFrame的元数据，指定了每一列特征的名字、数据类型。

对于降维相关算法，用户要求要在尽可能保留原有特征信息的情况下降低特征的稀疏性，使线性模型的收敛速度加快并且不要引入过多的噪声数据。

对于离散化算法，用户希望能在不知道相关领域知识的前提下，学习到尽可能合理的离散化区间，帮助用户自动化完成特征处理对业务预测效果的提升。用户希望在两种分箱停止条件中进行选择，用户可以选择分箱指标作为离散化过程的终止条件，当候选指标中没有满足分裂或合并的分箱指标则停止分裂或合并操作，具体细节可以看第二章技术综述的特征离散化简介；也可以选择最终模型分箱数量作为终止条件，这种分箱停止条件主要用于用户对离散指标(卡方值和最小描述长度)的统计含义不关心的场景下，多用在对比测试中控制变量。

表 3.2服务状态监控需求描述

|  |  |
| --- | --- |
| **需求名称** | **需求描述** |
| 保证与pipeline兼容。 | 算法实现可接收pipeline传来的数据，算法实现的输出可以被pipline的下游直接使用。 |
| 元数据一致性 | 记录每个处理对元数据信息的修改，确保格式一致。 |
| 稀疏性 | 对于降维算法，在保留重要信息的前提下，降低数据的稀疏度，使模型在新特征上学习更容易收敛。 |
| 停止条件自定义 | 特征离散化的停止条件提供两种选择:按模型最终的离散区间数量和按离散指标。 |
| 可解释性 | 算法生成模型的参数信息，要具有一定含义例如:特征重要性、相关性。 |

### 3.2.3非功能性需求分析

（1）性能需求

分布式特征处理算法的时间复杂度要优于单个节点处理的时间复杂度。并且随着节点数量的增加，在一定节点数量内，算法的性能会有显著提升。

（2）可扩展性

本项目可以作为Spark ML模块的功能扩展，可以被整合到基于Spark分布式计算云平台中使用。并且可以与pipeline工作流兼容。

（3）可用性

Spark本身具有较好的容错性。分布式算法实现能够自建检查点，在时间丢失或者通信失败时，能够从检查点快速恢复。

（4）可靠性

分布式算法本身会根据用户的设置，选择警报、保留或忽略数据中的异常值。在用户指定设置忽略异常值的前提下，处理任务不会因为异常值的存在而失败和中断。

（5）易用性

工具库附带每种算法和每个参数的说明，帮助并指导用户的进行处理任务。

（6）可维护性

算法实现中的核心功能代码使用策略模式，可以在不改变原有接口的前提下进行重构或者完成新的实现。

## 3.3系统总体设计与模块设计

### 3.3.1总体结构



图3.2系统整体结构图

分布式样本特征处理工具库，功能模块分为过采样算法和特征处理两大部分。如系统整体结构图3.2所示，过采样算法现阶段包括SMOTE以及其改进版Borderline-SMOTE1和Borderline-SMOTE2的分布式实现。特征处理包括因子分解机，高势集特征编码和连续特征离散化。因子分解机的实现有三种不同的版本：随机梯度下降，并发随机梯度下降和自适应随机梯度下降。三种实现细节会在第四章详细介绍。连续特征离散化有两种不同的实现版本，ChiMerge是对原始论文的复现，MDLP版本是对已有开源实现的改进和重构，具体实现会在第四章详细说明。

### 3.3.2过采样模块设计



图3.3 过采样模块类图设计

SMOTE中使用到了邻近节点查找算法KNN，用户需要在SMOTESampling对象中设置超参数k和m指定查找范围，参数具体含义可以参考第二章的过采样技术综述，用户还需要提供过采样率samplingRate控制新样本生成的数量。

从第二章的SMOTE算法的描述中，可以发现，SMOTE需要频繁的计算目标样本的最近k邻居和最近m邻居，对单个样本蛮力查找K近邻的时间复杂度为，N为样本总量，这样的查找效率无法满足用户的要求。我们尝试在分布式条件下实现使用KDTree或者类似的树模型，对查找过程进行优化，但是RDD的每个数据分区的位置通常不会再同一个节点上，这种需要重新划分数据分区的操作会增加节点之间的通信消耗，而且封装在partition中的数据在没有通过action计算出来之前是无法，通过索引得到的。受制于Spark的运行机制的限制，我们重新考虑优化原有的蛮力算法。该算法的问题在于对集合中的每个样本都要计算一次距离，然后排序输出距离最小的个k个样本。我们利用分布式计算的优势，在不同节点上同时计算各自本地的数据集上的样本对距离，在reduce阶段每个节点只保留本地中最近的k个样本点getLocalTopk，最后汇总得到全局结果getGlobalTopk，这样做可以一次将需要筛选的范围缩小倍，其中NumSamples为原有待查找范围内的样本总量，NumNodes为承担计算任务的节点数节点数。

为了进一步优化计算样本间距离计算的时间复杂度，我们引入了局部敏感哈希算法，对多维样本间的距离进行近似度量，缩短距离计算所需要的时间。computeDistsBetweenPoints给出所有候选样本间的近似距离和标签。注意忽略重复样本，虽然理论上距离为零的样本之间的距离更近，但是重复样本对我们生成新样本没有帮助，diff\*0=0。所以在过采样算法中我们要去除完全重复的样本。

原始SMOTE算法的k邻居样本查找是在少类样本内部进行的，样本数量较少，所以我们没有对这一部分中间结果进行持久化。

但是borderline版本的算法在生成DANGER集合的时候，查找少类样本点的m个邻近样本是在全体样本范围下进行的，尽管经过并发优化和局部敏感哈希处理，在实际业务的海量数据情况下，计算仍然相对耗时。而且计算结果在后续的k邻近还会用到。所以对存放样本距离信息的distPairs做持久化。

DANGER的生成和第二章中所提到的内容一致，当得到所有少类样本的m个邻居后，根据邻居中的中样本分布比例，判断是否要将少类样本是否可以被加入到DANGER集合中。

在计算DANGER的过程中，除了计算m个邻居样本的信息，附带计算了k个邻近正类样本，和k个邻近少类样本。这些信息可以在新样本生成时直接被使用，具体使用方法详见第二章的算法简介和第四章的算法实现。

表3.3 过采样参数说明

|  |  |  |
| --- | --- | --- |
| 参数 | 含义 | 取值 |
| featuresCol | 特征列名称 | String类型的合法列名 |
| labelCol | 标签列名称 | String类型的合法列名 |
| samplingLabel | 需要过采样的少类标签 | String类型非空标签值 |
| samplingRate | 过采样率 | 大于零的浮点数值 |
| kNeighbors | 合成新样本的近邻个数 | 大于零 |
| kind | 选择合成样本策略 | regular, borderline1, borderline2 |
| mNeighbors | Borderline算法指定参数，生成DANGER时所需近邻数量 | 大于零的整数 |
| numHashTable | 局部敏感哈希投影数量 | 大于1 的整数 |
| bucketLength | 哈希碰撞宽度 | 大于1的整数 |
| randomSeed | 随机数种子 | 整数 |
| mergeOriginData | 选择是否要将少类样本与原有样本直接融合 | 布尔值True,False |

需要说明的是kNeighbors指定少类样本生成时，从k个样本中间随机选择一个样本进行后续生成操作。因为对于k个样本的选择概率相等，所以k的取值越小，选取最近邻近的概率越大(生成样本在少类样本和选中的邻近样本之间产生)，样本生成的方式就越近保守。

mNeighbors控制DANGER的生成，当mNeighbors的取值越来越大时且样本在空间中的分布基本均匀，少类样本的m个邻居中多类样本的占比会越来越大，这个少类样本被作为边界加入到DANGER的倾向会越来越高。因为我们假设样本在空间中的分布均匀，且从整体样本集合看，多类样本所占比例会更高。所以当我们扩大邻居的查找范围时，遇到多类样本作为邻居的概率也会越来越高。所以当用户将mNeighbors调大时，被加入到DANGER的少类样本点会越来越多，对边界的定义会越来越“厚”。Borderline算法就是通过这种方式控制边界的范围。

numHashTable和bucketLength都是局部敏感哈希用来度量样本间距离参数。利用局部敏感哈希算法获得近邻存在一定随机性，局部敏感哈希的原理可以简单理解为，在样本分布的特征空间中随机切出k个带有网格的超平面，将所有的样本向超平面投影，当一些样本点投影落在一个超平面上的同一个网格内，则认为这些样本点在某一个方向上的度量是相近的，则有可能成为邻居。如果两个样本在所有超平面上的投影都落在相同的网格内，或者大部分落在相同的网格内，我们则认为这两个样本有很大概率是近邻。在这个类比中，每一个带网格的超平面就相当一个随机生成的哈希函数，投影操作就是哈希值的计算，当两个样本对同一个哈希函数的哈希值相同时说明两个样本落在了同一个网格内。

NumHashTable指定随机生成的哈希函数或是超平面的个数。BucketLength决定了超平面上网格的尺寸。numHashTable和bucketLength的取值越大，对真实情况的近似效果就越好，但同时也会消耗更多的计算资源。

### 3.3.3因子分解机模块设计

表3.4 因子分解机参数说明

|  |  |  |
| --- | --- | --- |
| 参数 | 含义 | 取值 |
| task | 任务类型：回归、分类 | Regression,classification |
| solver | 优化策略 | gd,pgd |
| initialWeights | 模型权重初始值 | Vector类型，用户定义初始值 |
| initialStd | 权重随机初始化的标准差 | Double类型，大于零 |
| useBiasTerm | 使用常数项选择 | 布尔值False,True |
| useLinearTerms | 使用一次项选择 | 布尔值False,True |
| numFactors | 因子个数 | 大于1的整数 |
| regParam0 | 对常数项的正则化系数 | Double类型 |
| regParam1 | 一次项正则化系数 | Double类型 |
| regParam2 | 二次项正则化系数 | Double类型 |
| miniBatchFraction | 训练采样比例 | 0~1之间的浮点数 |
| maxIter | 训练最大迭代次数 | 大于零的整数 |

Solver是需要用户指定的超参数，已有的优化策略有SGD和PGD，PGD是对随机梯度下降算法在分布式平台上面的改进具体改进细节详见第四章。

Task参数指定了任务类型，”regression”代表回归任务，”classification”代表分类任务。

根据libFM的实现，对于SGD优化的因子分解机损失函数定义，回归任务的损失函数定义为最小二乘法(OLS),分类任务的损失函数定义为)。

式中的即为常数项，即为一次项系数，即为二次项系数。正则化系数，是为了防止模型过拟合,使用L2正则化。在实现中我们为模型设置三种不同的正则化系数，regParam0是对应罚项的系数，regParam1是对应罚项的系数，regParam2是对应罚项的系数。

miniBatchFraction为别训练采样比例，要注意若maxIter \* miniBatchFraction<1，则会出现部分样本被遗漏而没有被用于训练的情况，这种情况会在运行日志中输出警告信息。

FactorizationMachines的参数训练是通过solver选择不同的优化器(optimizer)完成的，GradientDescent和ParallelGradientDescent是两种实现策略不同的优化器，ParallelGradientDescent在GradientDescent的基础上做了少许改进，可以在相同的条件下减少迭代数，快速收敛。

FactorizationMachinesGradient负责根据当前权重、训练集和参数配置对每一个样本计算对应梯度，最后由优化器对梯度汇总(reduce)求均值，将梯度信息交由FactorizationMachinesUpdater计算相应正则项，并完成权重更新。

### 3.3.4特征离散化与高势集模块设计

表3.5 特征离散化模块参数说明

|  |  |  |
| --- | --- | --- |
| 参数 | 含义 | 取值 |
| numBuckets | 最终分箱数 | 大于零的正整数 |
| handleInvalid | 非法值处理选项 | Skip,keep,error |
| heuristic Factor | 启发式优化因子，ChiMerge指定参数 | 大于零的浮点数 |
| inputCols | 输入名名 | Array[String] |
| outputCols | 输出列名 | Array[String] |
| solver | 选择分桶策略 | MDLP,ChiMerge |
| maxByPart | 分区最大样本容量 | 大于100的正整数 |
| stoppingCriterion | 停止条件值，解释含义和使用根据solver的选择而定。 | Double |
| minBinPercentage | 区间最小样本比例 | 取值在0.0~5.0之间的浮点数 |
| stopCondition | 选择分箱停止方式 | String类型，取值为Criterion,NumBuckets |

在需求分析中提到了用户希望可以选择分析过程的停止，第一种是根据分箱指标自动停止，第二种是指定最终分箱数，当模型达到分箱数量要求，就会停止分箱。用户可以设置StopCondition参数指定分箱算法的停止方式。

handleInvalid主要指定对于异常值的处理方式，skip选项表示剔除所有出现异常值的样本，当设置为keep选项时，ChiMerge会将所有异常值放到专门存放异常值的bucket中，当该参数设置为error时，遇到异常值会直接报错。

HeuristicFactor是针对ChiMerge算法的启发式改进，原本的ChiMerge在算法在融合过程中会出现样本分布不均的问题，以下表格是对分桶模型的模拟，其中第一列指定区间范围，第二列表示在此区间内A类样本的数量，第三列表示此区间内B类样本的数量。

表3.6 ChiMerge造成的异常分布

|  |  |  |
| --- | --- | --- |
| 分箱区间 | A类样本数量 | B类样本数量 |
| a,b | 321 | 62 |
| c,d | 192 | 0 |
| e,f | 0 | 1 |
| g,h | 60 | 0 |

由上表可知，(0.003723,0.003723)只有1例样本，但是与相邻区间存在显著差异，而且(a,b),(c,d)两个相邻区间的卡方值为34.8，(c,d),(e,f)两个相邻区间的卡方值为193。所以下一步要合并的区间是(a,b)与(c,d),这种不平衡被加剧。在相同桶数的情况下，这种“孤桶”对准确度的提升几乎没有，而且会占据一个桶的位置，实际有效的桶数要少于指定桶数。HeuristicFactor是为了解决这一问题而被提出来的。主要思想就是先在连续特征空间小范围内做局部融合，消除这种孤立点的存在。

表3.7 高势集特征参数说明

|  |  |  |
| --- | --- | --- |
| 参数 | 含义 | 取值范围 |
| inputCols | 输入列名 | Array[String] |
| outputCols | 输出列名 | Array[String] |
| useContinuousTargets | 任务类型 | 布尔值，True为回归，False为分类任务 |
| slopeFactor | 第二章中提到的过度因子f | 大于零的浮点数 |
| threshold | 第二章中提到的置信阈值k | 大于零的整数 |
| labelCol | 标签列 | String,合法列名 |

useContinuousTargets参数指定标签列的标签类型。slopeFactor和threshold是调节过度函数的参数，threshold控制偏移，slopeFactor控制过度斜率。过度函数的具体讲解详见第二章技术综述。

## 3.4 本章小结

本章主要是介绍系统主要部分的相关需求分析与模块设计。首先展示了项目的用例场景。然后通过描述需求的来源和需求列表的形式展现了用户在分布式环境中处理数据所遇到的问题。介绍了样本特征处理各模块的总体设计，以表格的形式汇总了各算法在实际使用中可能用到的参数，并对参数做出详细的解释。过采样部分还讲解了利用局部敏感哈希算法进一步提升对邻近样本的查找速度。特征离散化部分阐述了原本算法所遇到的问题，以及改进思路。

# 

# 第四章 分布式样本特征处理工具库的实现

## 4.1过采样模块的实现

### 4.1.1 SMOTE

下图是对Spark提交时产生的物理执行所用到的图示，圆角长方形代表一个RDD数据集，实线箭头代表窄依赖，Spark在执行由窄依赖所组成的Transformation时，例如map，mapPartition等，不需要对原有RDD中的分区(partition)进行拆分重组，不需要shuffle操作。虚线箭头代表宽依赖，这种情况下scheduler会对原有的RDD中的分区经行拆分重组，会造成额外的通信和IO消耗。直角分型代表RDD中的一个分区。



图4.1 Spark物理执行图示

SMOTE及其改进版本，都是Transformer，不需要模型训练过程，所以算法的主要执行逻辑在transform中。为了表达明确，在此说明另起一行的第一个RDD数据集是上一行最后(最右边)的数据集，执行顺序按照箭头方向从左往右依次进行。箭头上方标明了具体的处理逻辑，在下文会讲解实现细节。



图4.2 SMOTE算法物理执行图

SMOTE首先从输入的训练样本集中筛选出所有少类样本形成少类样本集合Minor Samples，格式为sample[feature:Vector, label:Int]，并对结果进行持久化操作，代码如下：

|  |
| --- |
| val minorPoints = allPoints.filter { row => row.getAs[Any]($(labelCol)) == minorLabel}  minorPoints.persist(StorageLevel.MEMORY\_AND\_DISK) |

图4.3 筛选样本集中少类样本代码

对于solver设置为regular的采样算法，我们要先计算少类样本和少类样本之间的距离，距离信息和样本对存放在distPairs中，作为k邻近查找候选数据集合。distPairs的格式为[sample1: Vector, sample2: Vector, distance: Double]，diantance是sample1 和 sample2 之间的距离，在SMOTE中sample1和sample2均来自少类样本集合minorSamples，代码如下：

|  |
| --- |
| val distPairs = getKind match {  case "regular" =>  DataImbaFunc.computeDistsBetweenPoints(minorPoints, minorPoints,  $(featuresCol), $(labelCol), rand, $(numHashTable), $(numBucketLength))  case \_ =>  DataImbaFunc.computeDistsBetweenPoints(minorPoints, allPoints,  $(featuresCol), $(labelCol), rand, $(numHashTable), $(numBucketLength))  .persist(StorageLevel.MEMORY\_AND\_DISK)  } |

图4.4 样本间距离计算代码

在计算样本对的距离的时候，并不是对所有可能的样本对都进行计算，两个集合的所有组合是两个集合的笛卡尔乘积，这个组合数量过于庞大，不仅占用内存还会消耗很多计算资源，而且许多样本对的组合是不必要的，因为它们彼此相聚很远。

为了解决这个问题我们引入了局部敏感哈希算法对样本位置和距离度量进行初步优化。具体操作就是对集合中的样本特征向量做局部敏感哈希投影形成一个新的哈希向量HashVector作为样本的新属性字段，HashVector相等的样本会落在一个范围较小的区域内，它们之间形成近邻的可能性会更大。因此我们只度量HashVector相等的样本间的距离。然后对两个集合以HashVector为键值进行JOIN操作，这样显著减小了需要计算距离的样本对数量。

图4.6中LSH[Datar, 2004]为每个样本计算HashVector的过程，HashVector的计算需要指定多个哈希函数和长度参数，LSH的具体内容可以参见第三章过采样算法的模块设计。

|  |
| --- |
| val datasetAHV = calcultateHashVector(datasetA, leftColName, hashVectorColName)  val datasetBHV = calcultateHashVector (datasetB, rightColName, hashVectorColName)  // 对两个数据集在HashVector做连接操作.  val joinedDataset = datasetAHV.join(datasetBHV, hashVectorColName)  .drop(hashVectorColName).distinct()  val distUDF = udf((x: Vector, y: Vector) => keyDistance(x, y), DataTypes.DoubleType)  val joinedDatasetWithDist = joinedDataset.select(col("\*"),  distUDF(inputColA, inputColB)).as(distCol)) |

图4.5 LSH优化代码代码



图4.6 局部敏感[Datar, 2004]哈希加速近邻查找物理执行图

得到distPairs之后，可以直接使用groupByKey()将所有样本对按键值聚合，键值是少类样本集中的样本特征，但是这样做会引发大量的网络通信将所有数据传送到对应节点。我们采取更加节省通信开销的策略，先在分区内部统计本地K近邻(getLocalTopK)，舍弃其余样本对，减少每个分区的数据量，最后用groupByKey进行汇总生成Global Neighbors，getLocalTopK的具体代码实现如下：

|  |
| --- |
| def getLocalTopK(iter: Iterator[(Vector, (Vector, Double, Any))],  k: Int, convertedLabel: Any): Iterator[(Vector, (Vector, Double, Any))] = {  // 每一个样本维护一个优先队列存放自己的K邻近样本  val knnMap = Map[Vector, PQ[(Vector, Double, Any)]]()  var res = ListBuffer[(Vector, (Vector, Double, Any))]()  iter.foreach { curr => // 遍历partition内的每一个样本对，执行以下操作  val knnKey = curr.\_1  val containsKeyFlag = knnMap.contains(knnKey)  val pq = knnMap.getOrElseUpdate(knnKey, // 取出当前样本的K近邻队列  PQ[(Vector, Double, Any)](curr.\_2)(pqOrdering))  val currDist = curr.\_2.\_2 // 取出当前样本对距离  if (containsKeyFlag && (pq.length < k || pq.head.\_2 > currDist)) {  pq += curr.\_2 // 若在本地发现距离更小的样本对，则更新  knnMap(knnKey) = pq // 队列  }  if (pq.length > k) {  pq.dequeue() // 维护K近邻优先队列  }  }  knnMap.foreach { x =>  x.\_2.foreach { y => // 从各个队列中取出结果进行输出  res += ((x.\_1, y))  }  }  res.iterator // 返回结果  } |

图4.7 本地查找K近邻代码代码

getLocalTopK 方法为存储少类样本及其邻居创建一个字典结构Map[Vector, PQ[Vector, Double, Any]]，Key为少类样本特征，对应的Value为该样本对应的K个最近邻居，通过优先队列PQ(PriorityQueue)维护。这部分操作在计算节点足够的情况下，每个partition并发进行，这也是SMOTE分布式实现的主要优势。

getLocalTopK完成后，生成LocalPairs，接下来需要将这些局部信息汇总(groupByKey)得到每个少类样本的全局K近邻。这一步需要shuffle操作，重新划分partition，因为上一步的局部合并，使得shuffle时的通讯开销减小。

|  |
| --- |
| def getGlobalTopK(iter: Iterator[(Vector, Iterable[(Vector, Double, Any)])],  k: Int): Iterator[(Vector, List[(Vector, Double, Any)])] = {  var res = ListBuffer[(Vector, List[(Vector, Double, Any)])]()  iter.foreach { curr =>  val pq = PQ[(Vector, Double, Any)]()(pqOrdering)  val subIter = curr.\_2.iterator // 从分区内汇总得到的信息局部近邻信息  subIter.foreach { currTuple =>  pq += currTuple // 合并局部信息，优先队列自动维护顺序。  if (pq.length > k) { pq.dequeue() }  } // 维护优先队列长度。  res += ((curr.\_1, pq.toList)) // 将结果以[minorSample, kNeighbors]的形式输出  }  res.iterator  } |

图4.8合成全局近邻代码

得到所有少类样本的K近邻后，可以根据过采样率决定每一个少类点上的采样个数。采样代码如下。

|  |
| --- |
| iter.foreach { curr => // cur表示当前样本及其邻居 [Vector, List[(Vector,Double)]]  val overSampleRate = if (rand.nextDouble()< rateDec) rateInt + 1 else rateInt  val (vector1,currKnn) = curr  (1 to overSampleRate).foreach { \_ =>  val vector2 = currKnn(rand.nextInt(currKnn.length)).\_1 // 从k邻居随机选择合成  res += Row(生成新样本(vector1, vector2, rand.nextDouble()), label)}} |

图4.9 合成新样本代码

以上是solver参数为regular时的SMOTE算法的逻辑实现。可以发现算法的主要性能瓶颈在于需要对每个样本进行多次K邻居查找以及最后汇总邻居信息的shuffle操作。Borderline算法面临同样的问题。

### 4.1.2 borderline-SMOTE



图4.10 borderline算法物理执行图

Borderline版本的实现，直到到distPairs,和SMOTE基本一致，唯一的区别就是距离计算。Borderline的样本对的距离计算不再是少类样本之间的距离计算，而是少类样本与全体样本间的距离计算，同样使用局部敏感哈希优化。

Borderline实现与SMOTE的主要不同之处在于，distPairs 生成之后，无论是从局部(Partition内)还是从全局都不再只统计k个少类邻居样本，需要对每一个步骤中的每一个少类样本维护三个近邻优先队列，即m个邻居样本(无论多类还是少类)，k个多类近邻样本和k个少类近邻样本。这是因为borderline1和borderline2会用到这些信息。为了避免重复计算，我们再一个Job中将所用信息全部统计出来。

|  |
| --- |
| // Generate DANGER set  val dangerPairs = distPairs.mapPartitions { iter => getLocalTopKBorderline(iter, k, m) }  .groupByKey().mapPartitions { iter => generateDANGERSet(iter, k, m) } |

图4.11 DANGERset生成代码

以上代码和SMOTE中的生成流程相似，主要区别在于getLocalTopK不再只统计少类邻近。

|  |
| --- |
| def getLocalTopKBorderline(iter: Iterator[(V, (V, D, L))], k: Int, m: Int, label: L): Iterator[(V,(List[(V, D, L)], List[(V, D, L)], List[(V, D, L)]))] = {  var knnMapAll = Map[V, MPQ[(V, D, L)]]() // 优先队列:m个邻近样本  var knnMapMajor = Map[V, PQ[(V, D, L)]]() // k邻近多类样本  var knnMapMinor = Map[V, PQ[(V, D, L)]]() // k邻近少类样本  var res = List [(V, (List[(V, D, L)], List[(V, D, L)], List[(V, D, L)]))]() //结果存放  iter.foreach { curr => // curr: (V, (V, D, L))  val knnKey = curr.\_1; val nlabel = curr.\_2.\_3 // get knns  knnMapAll = updateMap(knnMapAll, knnKey, curr.\_2, m) // 更新m邻近优先队列  if (nlabel == label) {knnMapMinor = updateMap(knnMapMinor, knnKey, curr.\_2, k)} else {knnMapMajor = updateMap(knnMapMajor, knnKey, curr.\_2, k)}}  knnMapAll.foreach { x => val knnAll = x.\_2.toList  val knnMinor = knnMapMinor.getOrElse(x.\_1,List [(V, D, L)]()) // 少类队列  val knnMajor = knnMapMajor.getOrElse(x.\_1,List[(V, D, L)]()) // 多类队列  res += ((x.\_1, (knnAll, knnMinor, knnMajor)))}  res.iterator} |

图4.12 本地近邻队列维护代码

上图是getLocalTopKBorderline的处理逻辑，为了方便表示将代码中的Vector缩写为V，Double缩写为D，标签类型缩写为L。与getLocalTopK逻辑类似，遍历distPairs中的所有样本对，distPairs中的数据存储结构为(V, (V, D, L))，为了方便表述我们设元组中第一个V代表第一样本，第二个V代表第二样本，D代表第一样本与第二样本之间的距离，L为标签类型用于指示该样本为多类还是少类。从生成distPairs的处理逻辑中不难发现，第一样本均为少类样本，第二样本多类和少类二者兼有。GetLocalTopKBorderline的主要功能除了更新knnMapAll就是在遍历distPairs的时候检验第二样本的标签类型，如果是多类就更新knnMapMajor，如果是少类样本则更新knnMapMinor。对于这些字典结构的更新使用方法updateMap，更新逻辑与SMOTE中的getLocalTopK类似。

得到局部邻近样本后，和SMOTE流程类似用groupByKey进行汇总生成全局信息：

|  |
| --- |
| subIter.foreach { threeLists => // threeLists包含三个局部邻近优先队列组  val localKnnAll = threeLists.\_1 // 全样本邻近队列组  val localKnnMinor = threeLists.\_2 // 少类样本邻近队列组  val localKnnMajor = threeLists.\_3 // 多类样本邻近队列组  localKnnAll.foreach { nn =>  pqAll += nn // 将新样本加入全局队列  if (pqAll.length > m) { pqAll.dequeue() } // 维护全局队列不超过m个元素  }  // For knn contains all points  localKnnMinor.foreach { nn =>  pqMinor += nn // 将样本汇总到全局队列  if (pqMinor.length > k) { pqMinor.dequeue() } // 维护队列样本个数  }  // For knn contains all points  localKnnMajor.foreach { nn =>  pqMajor += nn  if (pqMajor.length > k) { pqMajor.dequeue() }  }  } |

图4.13 全局信息汇总代码

汇总样本得到每个少类样本的m个邻近，根据m个临近的类型分布，判断是否将该少类样本点加入到DANGERset中，最终生成DANGERset。

|  |
| --- |
| val globalKnns = getGlobalTopKBorderline(iter, k, m)  globalKnns.foreach { curr => // curr: (少类样本,(全队列,少类队列,多类队列))  val allKNeighbors = curr.\_2.\_1 // 使用全队列判断样本是否在边界  val numOfMajNeighbors = allKNeighbors.count(x => !(x.\_3 == convertedLabel))  if (numOfMajNeighbors < m && numOfMajNeighbors >= m / 2 + m % 2) {  // 将符合边界要求的样本加入到DANGER中  res += ((curr.\_1, (curr.\_2.\_2, curr.\_2.\_3))) }} |

图4.14 边界判断代码

生成DANGERset后，DANGERset中的每个样本的k少类近邻和k多类近邻已经预先计算完成，通过这些信息可以分别完成borderline1和borderline2的最后阶段，样本生成。Borderline1的样本生成逻辑和SMOTE完全一致，区别在于SMOTE基于少类样本集执行，borderline1基于DANGERset执行。Borderline2的样本生成略有不同需要用到k邻近多类样本。

|  |
| --- |
| (1 to overSampleRate).foreach { \_ =>  val sample2 = currKnn(rand.nextInt(currKnn.length))  val lambda = if (sample2.\_3 == label) {  rand.nextDouble()  } else { rand.nextDouble() \* 0.5 } // 若抽中样本为少类样本，步长折半  lb += Row(generateInstance(sample1, sample2, lambda), convertedLabel)} |

图4.15 borderline2新样本生成代码

## 4.2特征相关性提取的实现

**4.2.1因子分解机随机梯度下降策略的实现**

因子分解机模型可以被应用于提取特征之间的相关性，还可以用于模拟推荐模型中的矩阵分解，libFM是一个因子分解机模型的软件实现，作者正是因子分解机模型的提出者Steffen Rendle。Spark平台缺少因子分解机算法的实现，本节讲述因子分解机在Spark上的分布式实现细节。对模型的具体讲解在第二章技术综述中有详细的说明，实现部分会侧重于讲解利用分布式数据如何训练出一个可用的模型，对于模型预测实现也会简要介绍。



图4.16 随机梯度下降训练因子分解机单步迭代物理执行图

上图是利用随机梯度下降算法在Spark训练因子分解机的流程，训练算法为迭代算法上图只展示了单步迭代内的训练流程。第一步对初始权重或是上一步迭代训练得到的权重进行广播，广播将driver端的变量统一发送到每个节点中去，这样可以减小task执行时对RDD外部变量引用造成的储存和传输开销，保证每个节点都有一个统一的模型。第二步随机采样小批量数据集产生Sampled Training Set，再使用map操作计算每个样本的梯度得到Gradients，单个样本的梯度计算的主要工作由FactorizationMachinesGradient完成，第三步将在partition内部对梯度求和局部汇总得到Partial Gradents Sum，记录分区内部的样本数量。第四步将第三步中得到的局部梯度和信息进行全局汇总求平均得到最终我们想要更新的梯度Average Gradient。最后更新权重信息，根据于上一轮权重比较或者损失函数历史纪录，判断训练是否收敛。开始下一轮迭代或是结束训练过程。

|  |
| --- |
| val gradient = new FactorizationMachinesGradient( // 负责梯度计算  $(task), $(useBiasTerm), $(useLinearTerms), $(numFactors), numFeatures)  val updater = new FactorizationMachinesUpdater( // 负责正则项计算和梯度更新  $(useBiasTerm), $(useLinearTerms), $(numFactors), regParams, numFeatures)  val optimizer = new GradientDescent(gradient, updater) // 创建优化任务  val newWeights = optimizer.asInstanceOf[Optimizer] // 开始优化任务  .optimize(trainingData.get, weights) |

图4.17 启动任务代码

Task决定了模型训练类型，regression代表回归任务，classification代表分类任务，回归任务和分类任务使用的损失函数有所区别。UseBiasTerm表示是否需要为模型训练偏移项，useLinearTerms表示是否需要为模型训练一次项，均为可选参数。NumFactors是原始模型中的f因子参数，决定了分解维度，f越大分解效果越好，但是计算量也会增大。

|  |
| --- |
| while (!converged && i <= numIterations) { // 迭代训练过程直到收敛或超过指定迭代数  val bcWeights = data.context.broadcast(weights) // 广播上一轮产生或是初始化的权重  // 随机取样少类样本进行随机梯度下降训练  val (gradientSum, lossSum, miniBatchSize) = data.sample(false, miniBatchFraction)  .treeAggregate((BDV.zeros, 0, 0))( // 实际实现中将局部和全局汇总合并为聚合操作  seqOp = (res, sample) => { // res: (gradient, loss, count), sample: (label, features)  val gredient, loss = gradient.compute(v.\_2, v.\_1, bcWeights )  (c.\_1 + gredient, c.\_2 + loss, c.\_3 + 1) }, // 再partition内局部汇总  combOp = (c1, c2) => { // c: (gradient, loss, count) // 全局汇总梯度信息  // 合并局部梯度和(gradientSum, lossSum, miniBatchSize) })  bcWeights.destroy(blocking = false) // 销毁上一轮迭代产生的广播变量  stochasticLossHistory += lossSum / miniBatchSize + regVal // 记录损失函数信息  val update = updater.compute(weights, // 权重更新  Vectors.fromBreeze(gradientSum / miniBatchSize.toDouble),  stepSize, i, regParam)  (weights , regVal) = update; i += 1 … // 检测收敛条件或停止条件  } |

图4.18 SGD权重更新代码

上图代码为单步迭代代码的实现细节，为解释方便去掉了非核心功能部分。实现部分的代码将局部合并和全局汇总过程整合到了treeAggrate 方法中，但是处理逻辑相同。SeqOp中的计算过程会将本分区内的梯度进行合并，combOp

将每个分区的局部合并结果进行汇总。完成梯度汇总后，要对原有的广播权重进行销毁，因为Spark中广播变量是只读的，即当一个变量被driver端广播到个节点(excuter)形成副本之后，对副本的修改是不被允许的，副本状态的改变也不会影响driver端原被广播变量的状态。

实现中的梯度计算是由FactorizationMachinesGradient功能类完成的，实现代码如下：

|  |
| --- |
| val (prediction, sum) = FactorizationMachinesModel.predictAndSum( // 预测值  featureVector, weights, useBiasTerm, useLinearTerms, numFactors, numFeatures)  val multiplier = task match { // 根据任务类型指定损失函数的计算  case FactorizationMachines.Regression => prediction – label  case FactorizationMachines.Classification =>  label \* (1.0 / (1.0 + Math.exp(-prediction \* label)) - 1.0) }  … // 计算偏移项和一次项的梯度  featureVector.foreachActive { // v是元素值，开始v对应的index  case (k, v) => val pos = k \* numFactors // 移动到特征指定位置  for (f <- 0 until numFactors) { // 梯度计算  cumValues(pos + f) += (sum(f) \* v - weights(pos + f) \* v \* v) \* multiplier } } |

图4.19 二次项梯度计算代码

图4.19代码中的multiplier，不是损失函数的计算值，而是损失函数相对预测值prediction的导数(梯度)。梯度计算并不是直接进行矩阵计算，因为因子分解机的大多数使用场景是稀疏数据集，直接使用矩阵计算并不划算，Spark提供了稀疏矩阵的表达，其主要结构就是紧凑排列向量和矩阵中的非零值，并存储其对应的index。代码的最后一行对梯度计算进行了进一步的优化，虽然因子分解机相对于其他分解模型缩减了模型权重参数的数量，但是参数数量依然可观，而且对于每个二次项参数的计算都有重复求和，为了能使这些求和信息能够被复用，我们在预测时，同时统计这些加和信息作为有用信息同预测值一并返回。公式中的二次项系数梯度计算中作为整体可以在第一次计算时缓存下来，后续的系数梯度计算可以复用此项。

PredictAndSum中的对预测结果的计算同样做了改进，对计算过程进行了简化。

|  |
| --- |
| val sum = Array.fill(numFactors)(0.0)  for (i <- 0 until numFactors) {  var sumSqr = 0.0  features.foreachActive {  case (k, x) =>  val t = weights(k \* numFactors + i) \* x //  sum(i) += t //  sumSqr += t \* t }  prediction += (sum(i) \* sum(i) - sumSqr) / 2 } |

图4.20 PredictAndSum代码

梯度计算完成并汇总后由Updater对权重进行更新，得到新权重，继续迭代或停止迭代。

### 4.2.2因子分解机并行梯度下降策略的实现

SGD算法可以训练出可用的模型，但是同时也存在一些不足，例如SGD算法需要过多的迭代步骤，在Spark中迭代一次就相当于执行一个Job，Job的启动和任务的重复序列化也会消耗一定资源。为了减少Job的数量，和节点间通讯的花销，并行梯度下降算法(parallel gradient descent)被应用到了因子分解机模型的训练中。大致流程如下图：



图4.21 PGD物理执行图

并行随机梯度下降算法的训练流程存在差异，随机梯度下降算法对每轮迭代中随机采样出的样本进行并行的梯度运算，并行随机梯度下降算法没有直接计算梯度而是现在每个分区内各自建立一个相同参数的模型，分区内的模型使用分区内的本地数据进行迭代训练，当从单个样本中计算得到梯度后就立即用梯度更新分区内的模型权重。将新模型权重直接应用于下一个样本的梯度训练。具体代码实现如下。

|  |
| --- |
| while (!converged && globalIterIdx <= numIterations) {  val bcWeights = data.context.broadcast(weights) // 向个分区广播上一轮融合过的权重  val (avgWeights, avgRegVal, lossSum, batchSize) = data.mapPartitions { part =>  // 设置本地变量localWeights, localRegVal，本地迭代数  while (part.hasNext) { val (label, vector) = part.next()  val (localGrad, localLoss) = gradient.compute(vector, label, localWeights)  val update = updater.compute(localWeights, localGrad, stepSize, localIterIdx)  (localWeights, localRegVal) = update // 及时更新本地模型权重  localLossSum += localLoss ; localIterIdx += 1 } // 记录损失函数信息  Iterator.single((localWeights, localRegVal, localLossSum, localIterIdx)) // 返回模型  }.treeReduce { case ((w1, rv1, ls1, c1), (w2, rv2, ls2, c2)) => // 融合局部模型权重  val avgWeights = (w1 \* c1 + w2 \* c2) / (c1 + c2) // 融合局部模型权重  val avgRegVal = (rv1 \* c1 + rv2 \* c2) / (c1 + c2)  (Vectors (avgWeights), avgRegVal, ls1 + ls2, c1 + c2) } // 返回全局结果  weights = avgWeights ; globalIterIdx += 1 … } // 迭代数加1,判断模型是否收敛 |

图4.22 并行梯度下降核心代码代码

代码中对数据集data的每个分区建立一个本地权重localWeights ，使用本地数据进行随机梯度下降训练。Update.compute返回更新后的权重和损失函数值。TreeReduce将局部模型根据训练迭代数加权求和得到完整模型权重。因为本地模型分别对本地数据进行局部迭代训练，可以在全局范围减少训练次数，达到更快收敛的效果。但是一次完整的迭代时间会显著变长，因为分区内部不在只是简单的计算梯度进行汇总，而是梯度计算加上模型更新，不断迭代训练。但是减少了全局汇总次数，可以有效的避免节点间频繁的数据传输。

### 4.2.3自适应梯度下降算法的实现

在机器学习任务中，除了模型本身和优化算法的作用，超参的设置决定了模型在现实数据的表现，因子分解机模型的表现同样很大程度依赖于正则化超参是否和合适。超参选择不当会导致过拟合或者欠拟合的问题，前者是因为对训练数据的过度学习导致模型失去泛化能力，通常是由正则化系数设置过小导致，欠拟合是模型没有从训练集中提取到重要特征，没有恰当的拟合数据的分布。正则系数设置过大，会使罚项信息掩盖从数据中学习到的梯度信息，就会导致欠拟合的发生。

本项目中实现所实现的因子分解机的正则化参数共有三个，一个是针对常数项，第二个针对一次项系数，第三个针对二次项系数。用户往往没有先验知识用来选择恰当的正则化参数。而且三个不同的正则化参数会有很多组合方式，如果使用网格索搜的方法找到合适的参数组合仍然是一种消耗资源的方式。自适应随机梯度下降法希望在迭代训练模型的过程中，同时找到合适的正则参数组合。



图4.23 自适应梯度下降算法物理执行图

自适应梯度下降法会在验证数据集上学习到正则项系数的参数，每一步迭代不仅会更新模型权重，还会根据梯度更新正则化系数。

|  |
| --- |
| val (gradientSumFV, miniBatchSizeFV) = validationData  .sample(withReplacement = false, miniBatchFraction, 42 + i)  .treeAggregate((BDV.zeros[Double](nReg), 0L))(  seqOp = (c, v) => {  // c: (grad, count), v: (label, features)  gradient.computeReg(v.\_2, v.\_1, stepSize, regParams, bcCurrGradien,  weights, c.\_1)  (c.\_1, c.\_2 + 1)  },  combOp = (c1, c2) => {  // c: (grad, count)  (c1.\_1 += c2.\_1, c1.\_2 + c2.\_2)  }) |

图4.24正则项系数训练代码

上图代码为正则项系数训练，正则项梯度计算和模型权重的训练流程类似，在每一次模型权重的梯度计算和更新之后，计算新的正则项系数的梯度，并更新上一轮迭代得到的正则项系数。

|  |
| --- |
| for (i <- 0 until numFeatures \* numFactors) {  val thetaT = oldWeights(i)  newWeights(i) = thetaT - stepSize \*(oldGradient(i) + 2 \* regParams(2) \* thetaT)  } |

图4.25估计模型代码

正则项系数的梯度计算需要对下一轮迭代所可能产生的模型进行估计，估计方法是利用Steffen Rendle所提到的方法，使用上一轮迭代产生的权重和梯度进行未来模型估计。再利用估计模型，计算正则项系数梯度。

|  |
| --- |
| for (f <- 0 until numFactors) {  data.foreachActive {  case (i, xi) =>  val pos = i \* numFactors  sumTwo(f) += xi \* oldWeights(pos + f)  sumThree(f) += xi \* xi \* approxNewWeights(pos + f) \* oldWeights(pos + f)  }  cumValues(2) += multiplier \* (-2) \* stepSize \* (sum(f) \* sumTwo(f) - sumThree(f))  } |

图4.26二次正则项系数梯度计算部分代码

## 4.3离散化模块实现

ChiMerge离散化模块的物理执行流程如下图，需要先将连续值进行去重和排序操作，这一部分的操作在训练过程中只进行一次。之后对单个连续数值特征进行区间化，设左右区间为该连续特征值。对离散化数据集进行持久化操作，避免迭代过程中重复计算。基于次离散区间数据集，进行全局合并和查找的迭代操作。



图4.27 ChiMerge单步合并物理执行图

|  |
| --- |
| case class Bucket(from: Double, to: Double, labels: Option[Array[Double]])  case class BucketPair(var first: Bucket, var second: Bucket, var chi: Option[Double])  case class BorderPair(start: Bucket, end: Bucket) |

图4.28区间合并基础数据结构

需要对合并过程中使用到的数据集进行说明，Bucket是代表区间的数据集，from记录了区间下界，to记录了区间上界，且均为闭区间，上界要大于等于下界。Labels记录了该区间内标签的分布，用于计算分桶指标。BucketPair为邻近区间对first和second是两个有序且相邻的区间，BuketPair是区间合并的最小单元，结构中存放的第三个项为相邻两个区间的卡方值，自由度一定的情况下，两区间卡方值越大表明两区间的标签分布差异越显著。BorderPair负责记录partition边界上的邻近区间对。受限于partition之间无法直接通信的问题，为了记录partition边界处的相邻区间对，我们要在进行局部区间查找时记录每一个partition的边界信息，start记录了区间内的第一个区间，end记录了区间内的最后一个区间。

|  |
| --- |
| part.foreach { b: Bucket =>  bucketPair match {  case BucketPair(null, null, None) => // 若Pair为空，则更新边界和邻近区间对  case BucketPair(\_, null, Some(Double.PositiveInfinity)) => // 缺少邻近对情况  case BucketPair(\_, \_, value) => // 计算卡方值，更新相邻  val chiSquareValue = computeChiSquareValue(BucketPair(end, b, None))  if (value.isDefined) {  if (chiSquareValue < value.get) {  bucketPair.first = end // end记录了上一轮迭代区间的位置  bucketPair.second = b  bucketPair.value = Some(chiSquareValue)  }  }  }  end = b // 更新前驱区间，最后一次迭代记录上界  } |

图4.29 partition内查找最小卡方值代码

上图代码为partition内部查找最优合并区间的实现逻辑，这一部分代码在每一个partition上同时独立运行，提高查找速度。当所有partition执行完此步骤后，将本区间的本地最优合并区间对和partition的边界信息传输给driver，driver根据汇总得到的信息，计算全局最优合并区间。

|  |
| --- |
| // 从返回在的边界信息中计算新的候选相邻区间对  val candidateBucketPairs = computeCandidateBucketPairs(borderlineBucketPairs)  val bucketPairToBeMerged = computeBucketPairToBeMerged(bucketPairs ++ candidateBucketPairs) // 汇总所有待选区间对，选取卡方值最小的区间对作为结果返回  val bcBucketedPairToBeMerged = sc.broadcast(bucketPairToBeMerged) // 广播  bucketRDD = bucketRDD.flatMap { b: Bucket => // 合并过程  val bp = bcBucketedPairToBeMerged.value  if (b.from == bp.first.from && b.to == bp.first.to) {  Iterator.empty  } else if (b.from == bp.second.from && b.to == bp.second.to) {  Iterator.single(Bucket(bp.first.from, bp.second.to,  Some(bp.first.value.get.zip(bp.second.value.get).map(a => a.\_1 + a.\_2))))  } else { Iterator.single(b) }  } |

图4.30 全局查找与合并代码

上图为全局查找与合并，在数据量庞大的情况下，合并的前期会非常耗时，而且结果会出现样本数量分布严重不平衡的问题。前者主要是因为Spark在执行每一个Job的时候，都会从driver端传输计算序列到各从节点。不仅传输需要耗时，而且计算逻辑的序列化和反序列化也是相对耗时的工作，成为该算法实现的主要性能瓶颈。为了解决以上两个问题，该项目自创性的提出了启发式优化方案，主要思路就是先做小范围局部合并，减小初始迭代是的区间数量。

|  |
| --- |
| def localMerge(buckets: RDD[Bucket], numLabels: Int): RDD[Bucket] = {  val span = buckets.map { case Bucket(\_, to, \_) => to }.max() -  buckets.map { case Bucket(from, \_, \_) => from }.min() // 确定连续值的取值范围宽度  val width = span / (numBuckets \* $(heuristicFactor)) // 确定本地合并的区间宽度  def updateInterval(interval: Bucket, newEle: Bucket): Bucket = { // 区间合并 }  buckets.groupBy { case Bucket(from, to, Some(\_)) =>  ((from + to) / (2 \* width)).floor  }.map { case (\_, iter) =>  iter.reduce((a, b) => updateInterval(a, b))  }.sortBy(\_.from)  } |

图4.31 启发式优化方法代码

因为ChiMerge是合并分箱策略，因此可以避免分裂式分箱带来的在同类样本区间内部制造切分点的问题。针对内部切分问题基于最小描述长度的分裂式分箱算法，有一定的优化和改进，在开始分箱之前，对已去重和排序的连续特征集进行遍历，检测边界点，若相邻样本的标签类型不同，则视为边界点加入候选集。

|  |
| --- |
| var ((lastFeatureIdx, lastValue, \_), lastFreqs) = it.next()  // println("the first value of part " + index + " is " + lastValue)  var result = Seq.empty[((Int, Float), Array[Long])]  var accumFreqs = lastFreqs  for (((fIdx, x, \_), freqs) <- it) {  if (isBoundary(freqs, lastFreqs)) {  // 边界点为本轮迭代样本点和上轮迭代样本点的中点  result = ((lastFeatureIdx, midpoint(x, lastValue)), accumFreqs.clone) +: result  accumFreqs = Array.fill(nLabels)(0L) // 经过样本点后累积分布清零  }  lastValue = x 更新上轮样本点  lastFreqs = freqs 更新对应分布  accumFreqs = (accumFreqs, freqs).zipped.map(\_ + \_) 计算累积分布  }  result = ((lastFeatureIdx, lastValue), accumFreqs.clone) +: result  result.reverse.toIterator |

图4.32 边界检测代码

候选边界点的选取可以从分桶效果和性能两方面改进分裂式算法的不足，分裂式算法的分桶过程与树模型的生成过程类似，对样本子集寻找最优切分点划分生成新的样本子集。

## 4.4高势集特征编码模块实现

高势集合特征编码是基于经验贝叶斯方法，对高密度离散特征值进行连续特征映射。算法实现不需要迭代计算。但是作为Estimator需要模型训练，可以指定回归任务或分类任务进行模型生成。用于分类任务的模型代码实现如下。

|  |
| --- |
| val inputLabelDistributionRDD = inputLabelRDD  // 按照离散值和标签取值重新聚合样本  .groupBy { case Row(input: String, label: String) => (input, label) }  .map { case (tuple, iter) => (tuple, iter. length) }.cache() // 统计数量并缓存  // 计算标签值的边缘概率分布  val labelsDistribution: Map[String, Double] = inputLabelDistributionRDD  .groupBy { case ((\_, label), \_) => label }  .map { case (label, iter) => (label, iter.foldLeft(0.0)((res, it) => res + it.\_2)) }  .collect().toMap  // 计算输入值的边缘概率分布  val inputDistribution = inputLabelDistributionRDD  .groupBy { case ((input, \_), \_) => input }  .map { case (input, iter) => (input, iter.foldLeft(0.0)((res, it) => res + it.\_2))}.collect().toMap |

图4.33 统计样本分布代码

上图代码分别统计了三种分布，离散特征值样本数量分布，标签取值样本数量分布，还有组合样本分布。使用这些数据我们可以直接计算标签取值的条件后验概率，这是高势集特征编码的初步输出结果。

对于样本量较小的离散特征取值，例如离散特征取值为的样本数量只有2个，且2个样本的标签值均为A。在这情况下，不能简单的认为概率=1，因为缺少足够数量的样本作为支持，这种统计结果是不可信的。我们引入了权重控制因子来平衡这类缺少足够样本支持的情况，关于权重因子的具体介绍可以参看第二章技术综述中对于高势集特征处理的讲解。

|  |
| --- |
| val lambdaForCategory = inputDistribution.map { case (input, num) =>  (input, lambda(num)) } // 根据样本数量计算权重控制因子  // 计算后验概率  val posterior = inputLabelDistributionRDD.map { case ((input, label), number) => ((input, label), number.toDouble / inputDistribution(input)) }.collect().toMap |

图4.34 后验概率计算代码代码

得到后验概率和控制因子之后，模型已经建立完毕，使用模型做transform操作时按照公式实现先验与后验之间的过渡。

|  |
| --- |
| case (posterior, prior) =>  posterior \* controlFactor(input, 0.0) + prior \* (1 - controlFactor (input, 0.0)}) |

图4.35 后验概率计算代码代码

## 4.4本章小结

本章主要阐述了分布式特征处理算法的实现细节及改进点。使用物理执行图展示了在Spark中特征转换和模型拟合算法的整体处理逻辑。第一部分讲解了过采样模块中的三种过采样策略regular，borderline1和borderline2的异同点和针对分布式环境的改进点。第二节介绍了因子分解机，主要讲解了针对分布式环境下，如何使用三种策略sgd，pgd和asgd训练出可用的相关特征提取模型。第三部分介绍了数据离散化方法中如何通过监督学习的方式建立自动化分桶的模型。最后介绍了高势集特征提取算法的实现要点。

# 

# 第五章 测试结果展示

## 5.1 精度测试

### 5.1.1 过采样精度测试

表5.1 过采样精度测试数据集

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 名称 | 任务类型 | 样本数 | 特征维度 | 样本比例 | 数据来源 |
| Yeast | 分类 | 1484 | 8 | 5:1484 | UCI |
| Abalone | 分类 | 4177 | 8 | 9:4177 | UCI |

实验流程是我们使用经过指定处理的数据集训练同构的预测模型模型，对训练完成的模型在验证集上，测试模型的AUC(Area Under Curve)和召回率(recall rate)。对于指定的处理方式，我们指定一组通过过采样算法进行处理，一组不进行过采样处理的。同时我们还对sklean中的SMOTE算法实现与本项目中的实现的实验效果进行了对比实验效果如下。

表5.2 第一组实验参数设置

|  |  |
| --- | --- |
| 参数名称 | 取值 |
| K | 5 |
| cross validation fold | 5 |
| OversamplingRate | 200% |
| Model | RandomForest(10 trees) |

表5.3 Abalone数据集上的实验结果

|  |  |  |
| --- | --- | --- |
| Abalone | auc | recall |
| nosampling(Spark) | 0.5 | 0 |
| nosampling(python) | 0.5 | 0 |
| python SMOTE | 0.503091933 | 0.011428571 |
| Spark SMOTE | 0.49995172 | 0 |

表5.4 Yeast数据集上实验结果

|  |  |  |
| --- | --- | --- |
| Yeast | auc | recall |
| nosampling(Spark) | 0.546785653 | 0.096363636 |
| nosampling(python) | 0.560283619 | 0.186775891 |
| python SMOTE | 0.624901927 | 0.330510582 |
| Spark SMOTE | 0.608215394 | 0.226909091 |

表5.5 第二组实验参数设置

|  |  |
| --- | --- |
| 参数名称 | 取值 |
| K | 5 |
| cross validation fold | 5 |
| OversamplingRate | 和多类样本数量一样 |
| Model | RandomForest(10 trees) |

表5.6 第二组实验在Abalone上的实验结果

|  |  |  |
| --- | --- | --- |
| Abalone | auc | recall |
| no sampling(Spark) | 0.5 | 0 |
| no sampling(python) | 0.5 | 0 |
| python SMOTE | 0.54477967 | 0.107619048 |
| Spark SMOTE | 0.679033267 | 0.500952381 |

表5.7 第二组实验在Yeast上的实验结果

|  |  |  |
| --- | --- | --- |
| Yeast | auc | recall |
| no sampling(Spark) | 0.546785653 | 0.096363636 |
| no sampling(python) | 0.560283619 | 0.186775891 |
| python SMOTE | 0.696940279 | 0.424727273 |
| Spark SMOTE | 0.813925489 | 0.715636364 |

通过以上两组试验我们可以得到三点结论，第一采样率的增加有助于少类样本召回率的提升。第二过采样处理可以显著提升类样本的召回率和模型的整体表现(AUC)。第三，本项目中的过采样算法实现效果与开源的sklearn效果相当，表现甚至更为有效(这里仍然存在一些平台差异导致结果不同的可能)。

Somte和borderline版本改进做对比实验如下，因为固定采样效果不明显，因此我们只设置一组实验，使过采样后的少类样本数量与原数据中的多类样本数量近似。

表5.8 borderline效果对比实验参数设置

|  |  |
| --- | --- |
| 参数名称 | 取值 |
| K | 5 |
| cross validation fold | 5 |
| OversamplingRate | 和多类样本数量一样 |
| Model | RandomForest(10 trees) |

表5.9 Borderline在Abalone上的实验结果

|  |  |  |
| --- | --- | --- |
| Abalone | auc | recall |
| Spark SMOTE(Prev) | 0.679033267 | 0.500952381 |
| Spark SMOTE(New) | 0.687060354 | 0.523809524 |
| SMOTE-borderline1(rate:300) | 0.542138041 | 0.119047619 |
| SMOTE-borderline2(rate:300) | 0.535457235 | 0.09047619 |
| python-borderline1 | 0.511751085 | 0.028571429 |
| python-borderline2 | 0.527659481 | 0.066666667 |

表5.10 Borderline在Yeast数据集上的实验结果

|  |  |  |
| --- | --- | --- |
| Yeast | auc | recall |
| Spark SMOTE(Prev) | 0.813925489 | 0.715636364 |
| Spark SMOTE(New) | 0.815834166 | 0.727272727 |
| SMOTE-borderline1 | 0.75236617 | 0.565454545 |
| SMOTE-borderline2 | 0.768603348 | 0.609090909 |
| python-borderline1 | 0.673663166 | 0.374545455 |
| python-borderline2 | 0.685562243 | 0.410909091 |

从上两组实验中我们可以发现borderline算法在Abalone数据集上的表现并不好，sklearn中的borderline处理后的召回率同样偏低，这可能是数据集中的少类样本的分布，并不是我们所期望的那样形成聚簇与多类样本形成鲜明的边界，数据的分布可能是零星的穿插在多类样本空间之中，使borderline无法找到明显的边界。

### 5.1.2 因子分解机精度测试

表5.11 因子分解机算法的精度测试数据集信息

|  |  |  |  |
| --- | --- | --- | --- |
| 特征名称 | A9a | 样本数量 | 488842 |
| 特征维度 | 123 | 离散特征数 | 123 |
| 非零特征比 | 0.114 | 是否有缺失值 | 无 |
| 类别数量 | 2 | 大小 | 3.5MB |

数据集来源为原始数据集为UCI的Adult数据集，a9a是John C. Platt.对其进行预处理后的结果。本数据取自国立台湾大学LIBSVM数据集网页。

原始数据集Adult是Barry Becker从1994年美国人口普查数据库中摘取出来的，目的是预测一个人能否拥有50K以上的年薪。原数据共有8个离散特征和6个连续特征，John C. Platt.将6个连续特征进行分桶操作，转变为离散特征，并将这14个离散特征进行One-Hot编码，生成123个由0和1构成的离散特征。标签列也进行了处理，用-1代替“<=50K”，用+1代替“>50K”。源数据中的缺失值被直接丢弃。

本实验中用于对比的开源实现为LIBFM，作者为Steffen Rendle。FM本身就是由Steffen Rendle提出的，LIBFM也因此是该领域比较权威的软件包。本实验采用分类准确率作为测试指标，用以对比本算法实现与相关开源实现。采用hold-out验证方法，数据集为本身已经被切分为训练集和验证集，其中训练集样本数为32561，测试集样本数为16281。

表5.12 因子分解机精度测试的参数设置。

|  |  |
| --- | --- |
| 参数名称 | 参数值 |
| 是否采用截距项 | 是 |
| 是否采用一次项 | 是 |
| 交叉项因子数K | 5 |
| 初始正则项系数 | 都为0 |
| 学习率 | 0.01 |
| 用正态分布初始化v时的标准差 | 0.1 |
| validation数据占原training数据的比例 | 0.2 |

libfm中采用的sgd对于每一个训练样本都会进行一次参数更新，当对遍历完所有训练样本后，计为一次迭代。而Spark中sgd,sgda算法的实现是基于minibatch的做法，即取一个少类样本进行梯度计算和更新，既为一次迭代。所以迭代的计数方式有所不同。而pgd中的迭代含义与libfm中的相类似，都是遍历完一遍完整训练集计为一轮迭代。

表5.13 LibFM 的实验结果

|  |  |  |
| --- | --- | --- |
| No. | Accuracy(sgda) | Accuracy(sgd) |
| 1 | 0.846078 | 0.845525 |
| 2 | 0.845218 | 0.84528 |
| 3 | 0.846631 | 0.845771 |
| 4 | 0.846447 | 0.84743 |
| 5 | 0.846201 | 0.84485 |
| 6 | 0.84657 | 0.847368 |
| 7 | 0.846324 | 0.847798 |
| 8 | 0.846017 | 0.846324 |
| 9 | 0.845894 | 0.847061 |
| 10 | 0.845833 | 0.84657 |
| avg | 0.8461213 | 0.8463977 |
| std | 0.000418746 | 0.001013357 |
| stepsize=0.01; 10 iter; validationFraction=0.2; regParam=0,0,0 | | |

表5.14 FM on Spark实验结果

|  |  |  |  |
| --- | --- | --- | --- |
| No. | accuracy(gd) | accuracy(pgd) | accyracy(sgda) |
| 1 | 0.849333579 | 0.846446778 | 0.850684847 |
| 2 | 0.849087894 | 0.846692464 | 0.851176218 |
| 3 | 0.849087894 | 0.846446778 | 0.850254898 |
| 4 | 0.849702107 | 0.8465082 | 0.851790431 |
| 5 | 0.848842209 | 0.846446778 | 0.849149315 |
| 6 | 0.848289417 | 0.846569621 | 0.850070634 |
| 7 | 0.849087894 | 0.8465082 | 0.849640685 |
| 8 | 0.849763528 | 0.846446778 | 0.850193477 |
| 9 | 0.849395 | 0.846569621 | 0.849640685 |
| 10 | 0.848780787 | 0.8465082 | 0.849210736 |
| avg | 0.849137031 | 0.846514342 | 0.850181193 |
| std | 0.000441778 | 7.90298E-05 | 0.00084604 |
| 参数 | Iter:800, minibatchFraction:0.8,  stepSize:0.1 | Iter:10,  stepSize:0.001 | Iter:800, minibatchFraction:0.8  validationFraction:0.2, stepSize:0.1 |

表5.15 对比两种sgda的实验结果

|  |  |
| --- | --- |
| 实现方法 | 准确率 |
| FM on Spark | 0.845323 ± 0.001958 |
| libfm | 0.846121 ± 0.000419 |

表5.16 FM on Spark实验结果

|  |  |
| --- | --- |
| 方法名称 | 准确率 |
| gd - mini-batch sgd | 0.849137 ± 0.000442 |
| pgd - parallel gradient descent | 0.846514 ± 0.000079 |
| sgda - adaptive regularization sgd | 0.850181 ± 0.000846 |

从上文的测试结果中我们可以发现，在a9a数据集上，分别采用sgd和sgda两种方法时，两种不同实现对应的准确率均值的差值比例都在0.1%左右。

考虑到随机性对于结果的影响（例如初始的权值采样自均值为0，标准差为0.1的高斯分布；sgda中验证集的产生也是随机的），我们可以认为两种实现的效果是一样的，即本实现能够正确地表示FM算法。

### 5.1.3 特征离散化模块精度测试

本测试对比用ChiMerge，EqualWidth，QuantileDiscretization在对相同项特征进行相同数量桶的离散化处理后，对训练和预测产生的影响。使用相同方法（决策树）相同超参训练得到模型进行测试。

表5.17 离散化模块精度测试

|  |  |
| --- | --- |
| 数据集名称 | Occupancy Detection Data Set |
| 样本数量 | 8143 |
| 特征维度 | 6 |
| 离散特征数 | 0 |
| 类别数量 | 2 |
| 大小 | 596.3KB |
| 数据来源 | UCI |

实验首先对比不同的处理算法对模型训练的影响，采用5折交叉验证计算准确率。统一模型为Spark-ML中的决策树模型，最大深度为7。

表5.18 分箱算法精度测试对比-1

|  |  |  |  |
| --- | --- | --- | --- |
| 处理方法  分箱数 | no无处理 | 等宽 | 等频 |
| 3 | 0.894510623 | 0.852756969 | 0.879528429 |
| 5 | 0.885423063 | 0.86565148 | 0.90556306 |
| 10 | 0.888001965 | 0.898563183 | 0.894387818 |
| 20 | 0.903720987 | 0.895247452 | 0.894142208 |
| 50 | 0.908019158 | 0.915755864 | 0.909984035 |
| 100 | 0.91010684 | 0.911334889 | 0.910966474 |
| 200 | 0.910720865 | 0.919317205 | 0.908264767 |
| 500 | 0.911334889 | 0.911089279 | 0.911212084 |
| 1000 | 0.911334889 | 0.927053911 | 0.924475009 |

表5.19 分箱算法精度测试对比-1

|  |  |  |  |
| --- | --- | --- | --- |
| 方法  分箱数 | ChiMerge(heuristic method) | ChiMerge  (orignin) | MDLP |
| 3 | 0.878423186 | 0.88308977 | 0.886651111 |
| 5 | 0.919194400 | 0.904457817 | 0.871300504 |
| 10 | 0.899545622 | 0.886651111 | 0.898685988 |
| 20 | 0.915755864 | 0.903352573 | 0.900528061 |
| 50 | 0.916615498 | 0.915387449 | 0.924475009 |
| 100 | 0.918703181 | 0.911826108 | 0.917106717 |
| 200 | 0.932702935 | 0.922141717 | 0.917597937 |
| 500 | 0.938351959 | 0.935281837 | 0.927790741 |
| 1000 | 0.973965369 | 0.979000368 | 0.927299521 |

总结以上数据可以得到数据离散化对实验结果的影响趋势。可以从中得出初步结论，ChiMerge算法随着分箱数量的增加，对预测任务的准确率有明显提升，且结果在大多数情况下要好于等宽、等频和基于最小描述长度的分箱方法。

### 5.1.4 高势集模块精度测试

本实验使用数据集要求存在高势集特征，要求被处理的特征取值数量至少在300以上，且不能大部分为孤值(仅在样本中出现一次)。符合这种应用场景的特征例如有城市邮编、交易记录中的客户编号和商品编号。

表5.20 高势集精度测试数据集信息

|  |  |  |
| --- | --- | --- |
| 数据集名称 | black\_Friday | wine\_review |
| 样本数量 | 164278 | 77287 |
| 特征维度 | 12 | 5 |
| 高势集离散特征 | 1 | 2 |
| 离散特征数 | 9 | 4 |
| 类别数量 | 2 | 2 |
| 大小 | 5M | 3m |
| 数据来源 | kaggle | kaggle |

BlackFriday数据集是黑五当天某零售商的销售记录，其中包括客户基础信息（Customer ID，年龄，性别，职业，地址等）和交易信息，产品ID，成交金额，产品类别1,2,3。在此测试中我们选取性别作为label。

Product\_id, Age, Occupation, City\_Category, Stay\_In\_Current\_City\_Years, Marital\_Status, Product\_Category\_1 作为特征，进行适当预处理(stringindex, onehot), 其中Product\_id作为我们输入的高实际特征。

使用logistic regression作为验证模型。模型相关参数和高势集特征编码超参见下表。

表5.21 高势集精度测试固定参数信息

|  |  |
| --- | --- |
| **参数名称** | **参数值** |
| maxIter | 100 |
| penalty | L2 |
| tol | 1E-6 |
| standardization | true |
| slopeFactor | 1000000.0 |
| threshold | 100 |

实验采用accuracy，Area under PR，F1作为测试指标，用以对比高势集编码处理和one-hot处理之间的差异。采用10-fold验证方法，以减少随机量对测量结果产生的影响。数据集blackFriday.csv下精度测试结果见下表。

表5.22 高势集精度测试实验结果

|  |  |  |  |
| --- | --- | --- | --- |
| 实现方法 | accuracy | area under PR | F1 |
| 高势集编码 | 0.7867 ± 0.00193 | 0.44164 ± 0.00711 | 0.4244±0.00576 |
| one-hot | 0.7858 ± 0.00211 | 0.43377 ± 0.00773 | 0.4178 ± 0.01003 |
| 不处理 | 0.7854 ±0.00256 | 0.41994±0.00729 | 0.4070±0.0054 |

从结果可以看出，高势集特征编码处理过的学习任务效果较好，从单个阈值下（0.5）的accuracy看，效果可能不够明显，但是PR，F1等指标可以证明整体效果要优于one-hot和空白处理，这是因为one-hot处理过的特征过于稀疏，不利于在限制迭代次数内较快收敛。

## 5.2 压力测试

**5.2.1 过采样算法压力测试测试**

表5.23 硬件环境

|  |  |
| --- | --- |
| 节点数 | 2 |
| 单节点核数 | 8 |
| CPU | (Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz) \* 32 |
| 内存 | (Samsung DDR4 16384MB) \* 8 |
| 网卡 | Intel Corporation I350 Gigabit Network Connection |

软件环境Master - Spark2：Scala version 2.11.8, Java HotSpot(TM) 64-Bit Server VM, 1.8.0\_131, Executor - Spark2: Java HotSpot(TM) 64-Bit Server VM, 1.8.0\_25。

对于标准SMOTE，推得时间公式: T = O(D \* N \* r) ，其中D为样本特征数量，r为少类占比，N为数据量。

表5.24 样本数量压力测试结果

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **数量** | **50k** | **100k** | **500k** | **1000k** | **2000k** | **5000k** |
| 时间(s) | 1.12 | 1.69 | 3.18 | 6.42 | 11.30 | 27.77 |

由上表可知标准SMOTE用时与样本数量成线性关系，剩余两种模式同理。

关于于维度对执行时间的影响，实验结果如下。

表5.25 样本维度压力测试结果

|  |  |  |  |
| --- | --- | --- | --- |
| **样本维度** | **10** | **100** | **1000** |
| 时间秒 | 1.48 | 7.62 | 45.55 |

特征数量会影响计算两个点的欧式距离的时间，同时会极大占用内存，易出现OOM错误，后面会分析此过程。标准SMOTE用时与样本特征数量成线性关系。

表5.26 关于SMOTE用时与少类占比的关系

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **少类占比** | **0.1** | **0.15** | **0.2** | **0.3** | **0.4** |
| 时间秒 | 44.68 | 44.72 | 48.79 | 52.50 | 58.43 |

从上表中可看出，标准SMOTE用时与少类占比成线性关系，剩余两种模式同理。

表5.27 关于SMOTE用时与核数cores的关系

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **executor-cores** | **1** | **2** | **4** | **8** |
| 时间 秒 | 207.53 | 98.79 | 46.20 | 27.86 |

由以上表可知。执行的时间与executor-cores的数量成反比剩余两种模式同理在内存够用的情况下，时间的预估公式为T = k \* N \* D \* r / cores。N以千为单位，系数k估计值在(4 , 5)之间。

Somte样本维度极限与Executor内存的关系见下表，可以看出，在样本数量为500k情况下，如executor memory 为20g 大约可支持约1500特征维度的数据计算；同时, 已知样本数量N 单位(百万) , 特征的数量 D单位(千) , executor memory应大于等于 40 \* N \* D, 系数约为 40。

表5.28 Somte样本维度极限与Executor内存的关系

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 样本维度\内存 | 100 | 200 | 500 | 800 | 1000 | 1200 | 1500 |
| 10G | 5.67 | 6.43 | 20.11 | \ | \ | \ | \ |
| 20G | 5.68 | 6.37 | 19.91 | 69.38 | 78.78 | \ | \ |
| 30G | 5.69 | 6.01 | 19.37 | 62.36 | 88.75 | 66.06 | 81.12 |

Somte样本维度极限与Driver内存的关系见下表。随着driver-memory的增大,可以一定程度让提高计算速度，但还是无法处理更大维度的数据，停留某一数值上。所以，在executor-memory设置为20g时，所支持的最大维度为1000的计算。

表5.29 Somte样本维度极限与Driver内存的关系

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 样本维度 \内存 | 100 | 200 | 500 | 800 | 1000 | 1200 | 1500 |
| 10G | 5.30 | 5.98 | 20.22 | 37.09 | 53.02 | \ | \ |
| 20G | 5.77 | 5.75 | 18.77 | 32.74 | 48.29 | \ | \ |
| 30G | 5.77 | 5.93 | 20.39 | 38.94 | 44.45 | \ | \ |

**5.2.2 因子分解机压力测试**

硬件环境和软件配置同上。关于FM用时与数据量的关系，实验结果如下。可以得到样本数量基本与运行时间呈线性关系。

表5.30 FM压力测试样本数量与时间的关系

|  |  |
| --- | --- |
| 样本数 | 所耗时间（秒） |
| 100 | 2.55 |
| 1000 | 2.76 |
| 10000 | 3.12 |
| 100000 | 6.53 |
| 1000000 | 34.71 |
| 2000000 | 64.38 |

测试FM用时与特征数量的关系，实验结果如下。可以看出运行时间基本与特征数量呈线性关系。

表5.31 FM压力测试样本维度与时间的关系

|  |  |
| --- | --- |
| 特征数 | 所耗时间（秒） |
| 100 | 3.22 |
| 400 | 5.18 |
| 1600 | 13.79 |
| 6400 | 49.08 |
| 25600 | 300.44 |

FM用时与因子数量(f)的关系，实验结果如下表所示。运行时间基本与因子数量呈线性关系。

表5.32 FM压力测试因子数与时间的关系

|  |  |
| --- | --- |
| 因子数量 | 运行时间（秒） |
| 1 | 25.38 |
| 2 | 28.86 |
| 4 | 32.6 |
| 8 | 42.09 |

FM用时与核数的关系，实验结果如下表。可以看到，当核数逐渐升高时，运行时间也慢慢减少。如果对表格两项分别取log10，可以发现，时间和核数在取log后大概呈线性关系，说明核数的增加还是很好地提升了部分性能。程序中顺序执行的部分所耗时间大概是10秒，其余部分为并行时间。

表5.33 FM压力测试节点数与时间的关系

|  |  |
| --- | --- |
| executor-cores | 运行时间（秒） |
| 1 | 55.15 |
| 2 | 30.37 |
| 4 | 19.55 |
| 8 | 15.98 |

executor-memory和特征数量之间的关系和极限，关系如下表。可以看到，当样本数量为100000，executor数量为2，每个executor的memory为32g时，最多可支持的特征数量大概是12800，而随着executor的memory逐渐减少，可支持的最大特征数量也逐渐减少。总executor memory（单位Byte）除以可支持特征数量与样本数量的乘积的结果约等于50。

表5.34 executor-memory和特征数量之间的关系和极限

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 特征数量\内存 | 8g | 16g | 32g | 2executors\*32g |
| 400 | 19.47 | 19.1 | 19.15 | 15.38 |
| 1600 | 71.92 | 69.68 | 68.3 | 45.57 |
| 3200 | / | 141.02 | 144.52 | 94.39 |
| 6400 | / | / | 322.86 | 199.71 |
| 12800 | / | / | / | 472.84 |
| 25600 | / | / | / | / |

对于driver内存和Factor数量之间的关系和极限。由于driver端的内存瓶颈主要是由梯度更新时treeAggregate将weights集中到driver端造成的，因此在partition数量固定的前提下，这里主要考察driver-memory和Factor数量（在特征数量固定时即权值的数量）的关系。可以看到，当field数量和factor数量增加时，所需要的driver-memory也随之增加。当driver-memory为8g时，所支持的特征数×Factor数约为10000×3200=32000000。事实上，factor的数量一般来说是不可能达到3200，因此driver端内存的限制主要来自于特征数量。但是又因为特征数量的增加对executor的内存限制更大，所以一般来说，driver端的内存限制并不是很大。

表5.35 driver内存和Factor数量之间的关系和极限

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Factor数量\内存 | 1g | 2g | 4g | 8g |
| 200 | 199.14 | 206.22 | 214.47 | 213.06 |
| 400 | / | 306.14 | 262.67 | 287.55 |
| 800 | / | 447.18 | 440.77 | 432.41 |
| 1600 | / | / | 775.95 | 738.37 |
| 3200 | / | / | / | 1462.43 |

运行时间与样本数量、特征数量以及因子数量呈线性关系；随着特征数量的增加，所需的executor memory也相应增加，总executor memory（单位Byte）除以可支持特征数量与样本数量的乘积的结果约等于50。

**5.2.3 特征离散化压力测试**

ChiMerge有监督建模时间与数据量的关系，实验结果如下表。可以看出样本数量大于30万时，建模过程需要一小时。这主要是因为迭代次数过多，虽然没有产生shuffle操作，但是对于执行函数的不断的序列化和反序列使执行效率大大降低。使用启发式优化参数，可以在初始阶段显著减少区间数量，缩减迭代次数。

表5.36 ChiMerge压力测试样本数量与时间的关系

|  |  |  |
| --- | --- | --- |
| 数量 | 原版(秒) | 启发式优化(秒) |
| 100000 | 807 | 44 |
| 200000 | 1680 | 137 |
| 300000 | 2590 | 190 |
| 500000 | 4741 | 262 |
| 1000000 | 10820 | 600 |

测试算子在节点数量维度上的可扩展性。从下表可以得出结论如迭代数量过多，没有启发式的情况下，计算非常耗时，即使加大并发度，效果也没有明显提升，经分析后发现传输等待和反序列化的耗时相对固定，没有办法通过并发实现缩减。所以通过启发式的方法缩减迭代次数。

表5.37 ChiMerge压力测试节点数与时间的关系

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| core | 1 | 2 | 4 | 8 | 16 | 32 | 60 |
| 时间（s） | 2412 | 1588 | 1261 | 1023 | 778 | 960 | 886 |

**5.2.3 高势集特征编码模块压力测试**

关于样本数量对高势集特征建模和预测时间的影响。由下表数据可知在其他参数固定的情况下。建模和预测时间呈线性增长。

表5.38 高势集特征编码模块压力样本与时间的关系

|  |  |  |
| --- | --- | --- |
| 样本数量 | 模型训练时间(ms) | 模型预测时间(ms) |
| 10000 | 4965 | 1114 |
| 100000 | 5212 | 1090 |
| 1000000 | 7503 | 3529 |
| 2000000 | 13715 | 6545 |
| 3000000 | 18322 | 10686 |
| 4000000 | 20804 | 11796 |
| 5000000 | 28517 | 15255 |
| 6000000 | 35244 | 17730 |
| 7000000 | 39281 | 18383 |
| 8000000 | 39802 | 24373 |
| 9000000 | 51431 | 24706 |
| 10000000 | 48728 | 26503 |

关于离散特征势集度对建模和模型预测时间所带来的影响。由下表数据可知，建模时间出现波动但无明显趋势，预测时间较为稳定。势集度对建模时间和预测时间在小范围内没有显著影响。

表5.39 高势集特征编码模块压力测试势集度与时间的关系

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 势集度 | 3000 | 2500 | 2000 | 1500 | 1000 | 500 |
| 建模 (ms) | 49541 | 47280 | 52575 | 56802 | 51779 | 48728 |
| 预测 (ms) | 26281 | 25117 | 26202 | 28503 | 27218 | 26503 |

关于核心数量对建模时间和预测时间带来的影响，如下表所示。可以看出在一定范围内，核心数量的增加会显著减少模型建立所耗时间。达到一定数量后执行时间趋于稳定不在变化。核心数对预测时间没有显著影响。

表5.40 高势集特征编码模块压力测试核心数与时间的关系

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 核心数 | 6 | 10 | 8 | 6 | 4 | 2 | 1 |
| 建模(ms) | 42784 | 43830 | 42298 | 50445 | 48728 | 73422 | 85697 |
| 预测(ms) | 27491 | 28414 | 25253 | 24578 | 26503 | 31925 | 26552 |

# 第六章 总结与展望

## 5.1 总结

本文结合了企业用户希望在分布式环境下对数据特征处理的需求，针对四种数据特征难以利用的特殊场景，样本分布不平衡、离散特征间的相关性提取、非线性分布连续特征和高势集特征。分别提出了相应的解决方案，过采样帮助捕捉稀缺样本的特征信息，利用因子分解机模型，拟合离散特征间的相关信息，使用ChiMerge以及MDLP最优分箱算法帮助用户自动离散化连续特征，高势集特征编码可以避免高势集离散特征哑编码后的维数灾难。将这些算法做分布式的再设计和实现，对处理过程中出现的数据异常情况，进行了分析、假设、改进和验证。对每一个算法进行了精度测试和压力测试。

## 5.1 进一步工作展望

从文中可以看出，数据离散化和高势集特征处理，实现并不完善，测试效果与预期还存在一定差异。希望后续工作可以将更适应分布式实现的算法加入到分布式特征处理工具库中。

因子分解机使用梯度训练的方法学习回归任务还是会导致梯度失效，主要是因为，回归没有边界值的界定，在数次累乘的情况下会导致梯度爆炸。希望可以找到更好的方法进行回归任务。

# 参 考 文 献

**[Andrew, 2012] AndrewCucchiara. Applied Logistic Regression[J]. Technometrics, 2012, 34(3):358-359.**

**[Chawla, 2002] Chawla N V, Bowyer K W, Hall L O, et al. SMOTE: synthetic minority over-sampling technique[J]. Journal of Artificial Intelligence Research, 2002, 16(1):321-357.**

**[Chen, 2016] Chen T, Guestrin C. XGBoost:A Scalable Tree Boosting System[C]// Acm Sigkdd International Conference on Knowledge Discovery & Data Mining. 2016.**

**[Czarnowski, 2018] Czarnowski I, Jędrzejowicz P. Stacking-Based Integrated Machine Learning with Data Reduction[C]// 2018.**

**[Datar, 2004] Datar M, Immorlica N, Indyk P, et al. Locality-sensitive hashing scheme based on p-stable distributions[J]. 2004.**

**[Fayyad, 1999] Fayyad, U.M., Cercone N. Discretization of Continuous Attributes for Learning Classification Rules[M]// Methodologies for Knowledge Discovery and Data Mining. 1999.**

**[Goodfellow, 2014] Goodfellow I J, Pouget-Abadie J, Mirza M, et al. Generative Adversarial Networks[J]. Advances in Neural Information Processing Systems, 2014, 3:2672-2680.**

**[Graves, 2012] Graves A. Long Short-Term Memory[M]// Supervised Sequence Labelling with Recurrent Neural Networks. 2012.**

**[Guolin Ke, 2012] Guolin Ke, Qi Meng, ThomasFinley. LightGBM: A Highly Efﬁcient Gradient Boosting DecisionTree. 2012.**

**[Han, 2005] Han H, Wang W Y, Mao B H. Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning[C]// International Conference on Advances in Intelligent Computing. 2005.**

**[Hearst, 1998] Hearst M A. Support Vector Machines[J]. IEEE Intelligent Systems & Their Applications, 1998, 13(4):18-28.**

**[Juan, 2016] Juan Y, Zhuang Y, Chin W S, et al. Field-aware Factorization Machines for CTR Prediction[J]. 2016.**

**[Kerber, 1999] Kerber R . ChiMerge: Discretization of numeric attributes[J]. Proc.national Conf.on Artificial Intelligence, 1999:123-128.**

**[Koren, 2008] Koren Y . Factorization meets the neighborhood : a multifaceted collaborative filtering model[C]// Proceedings of the 14th ACM SIGKDD International Conference of Knowledge Discovery and Data Mining, 2008. ACM Press, 2008.**

**[Krizhevsky, 2012] Krizhevsky A , Sutskever I , Hinton G . ImageNet Classification with Deep Convolutional Neural Networks[J]. Advances in neural information processing systems, 2012, 25(2).**

**[Loh, 2011] Loh W Y . Classification and regression trees[J]. Wiley Interdisciplinary Reviews Data Mining & Knowledge Discovery, 2011, 1(1):14-23.**

**[Micci-Barreca, 2001] Micci-Barreca D . A Preprocessing Scheme for High-Cardinality Categorical Attributes in Classification and Prediction Problems[J]. ACM SIGKDD Explorations Newsletter, 2001, 3(1):27-32.**

**[Rendle, 2010] Rendle S , Schmidt-Thieme L . [ACM Press the third ACM international conference - New York, New York, USA (2010.02.04-2010.02.06)] Proceedings of the third ACM international conference on Web search and data mining - WSDM \"10 - Pairwise interaction tensor factorization for personalized tag recommendation[J]. 2010:81.10 - Pairwise interaction tensor**

**[Rendle, 2011] Rendle S. Factorization Machines[C]// IEEE International Conference on Data Mining. 2011.**

**[Rendle, 2012] Rendle S. Factorization Machines with libFM[J]. Acm Transactions on Intelligent Systems & Technology, 2012, 3(3):1-22.**

**[Rissanen, 2010] Rissanen, Jorma. Minimum Description Length Principle[M]. 2010.**

**[Sergio, 2016] Ramírez-Gallego, Sergio, García, Salvador, Mourio-Talín, Héctor, et al. Data discretization: taxonomy and big data challenge[J]. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 2016, 6(1):5-21.**

**[Tang, 2013] Tang X, Jie Z. Dynamic Personalized Recommendation on Sparse Data[J]. IEEE Transactions on Knowledge & Data Engineering, 2013, 25(12):2895-2899.**

**[Xin, 2012] Xin R S, Rosen J, Zaharia M, et al. Shark:SQL and rich analytics at scale[J]. 2012.**

**[Zaharia, 2010] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: cluster computing with working sets[C]// Usenix Conference on Hot Topics in Cloud Computing. 2010.**

**[Zaharia, 2012] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing[C]// Usenix Conference on Networked Systems Design & Implementation. 2012.**

**[张振华, 2015] 张振华, 徐瑾辉, 李龙欣,等. 基于SMOTE算法与决策树的沙尘暴短期预警研究[J]. 徐州工程学院学报(自然科学版), 2015(3):40-46.**

# 致 谢

首先非常感谢我的指导老师。本次毕业论文的撰写是在老师的耐心指导下完成的。感谢老师在论文的开题报告、论文的中期报告和论文的构思提出的宝贵建议，使我在项目的设计开发过程中有了明确的方向。在论文的撰写阶段，感谢老师对论文初稿存在的问题对我细心的指导，给予中肯的建议，因此我才能不断的完善论文的内容和结构。老师态度严谨和知识渊博令我十分敬佩，是我学习的榜样，在我今后的学习工作和生活中想必也会产生深远的影响。

感谢我在星环实习的企业导师林晨，在项目需求的分析和开发过程中都给予了我很大的帮助。在项目开发的过程中，不仅仅交给我很多项目实现相关的东西，更重要的是培养了我分析问题和解决问题的能力。在企业导师的指导下，我顺利完成了项目的实现。

感谢本科和研究生期间南大教导过我的老师，你们传授的不仅仅是知识，更多的教会了我学习的方法，指引了学习的道路。感谢企业实习中帮助过我的同事，热心的对项目提出需求和指出可以改进的地方，并且在项目设计和实现的过程中提供了很多其他解决方案的参考和见解，开拓了我的思路。

感谢我的父母，他们给我生活上的照顾和精神上的鼓励和支持，才能让我在求学的道路上越走越远。

最后再次感谢所有帮助过我的老师、同事和领导。

# 版权及论文原创性说明

任何收存和保管本论文的单位和个人，未经作者本人授权，不得将本论文转借他人并复印、抄录、拍照或以任何方式传播，否则，引起有碍作者著作权益的问题，将可能承担法律责任。

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含其他个人或集体已经发表或撰写的作品成果。本文所引用的重要文献，均已在文中以明确方式标明。本声明的法律结果由本人承担。

作者签名：

日期： 年 月 日