适用于 **Apache Spark** 的**NetApp**存储解决方案

NetApp artificial intelligence solutions

NetApp
February 12, 2026

目录

适用于 Apache Spark 的NetApp存储解决方案

TR-4570：适用于 Apache Spark 的NetApp存储解决方案：架构、用例和性能结果

Rick Huang，Karthikeyan Nagalingam， NetApp

本文档重点介绍 Apache Spark 架构、客户用例以及与大数据分析和人工智能 (AI) 相关的NetApp存储产品组合。它还展示了使用行业标准 AI、机器学习 (ML) 和深度学习 (DL) 工具针对典型 Hadoop 系统进行的各种测试结果，以便您可以选择合适的 Spark 解决方案。首先，您需要一个 Spark 架构、适当的组件和两种部署模式（集群和客户端）。

该文档还提供了解决配置问题的客户用例，并讨论了与大数据分析以及 Spark 的 AI、ML 和 DL 相关的NetApp存储产品组合的概述。然后，我们得到来自 Spark 特定用例和NetApp Spark 解决方案组合的测试结果。

客户挑战

本节重点关注零售、数字营销、银行、离散制造、流程制造、政府和专业服务等数据增长行业中客户面临的大数据分析和 AI/ML/DL 挑战。

不可预测的表现

传统的 Hadoop 部署通常使用商品硬件。为了提高性能，您必须调整网络、操作系统、Hadoop 集群、生态系统组件（如 Spark）和硬件。即使您调整每一层，也很难达到所需的性能水平，因为 Hadoop 运行在并非为您的环境的高性能而设计的商用硬件上。

介质和节点故障

即使在正常条件下，商品硬件也容易出现故障。如果数据节点上的一个磁盘发生故障，则 Hadoop 主服务器默认认为该节点不健康。然后，它通过网络将该节点上的特定数据从副本复制到健康节点。此过程会减慢任何 Hadoop 作业的网络数据包速度。当不健康的节点恢复健康状态时，集群必须再次复制数据并删除过度复制的数据。

Hadoop供应商锁定

Hadoop 分销商拥有自己的 Hadoop 发行版和版本控制，从而将客户锁定在这些发行版上。然而，许多客户需要内存分析支持，而这种支持不会将客户绑定到特定的 Hadoop 发行版。他们需要自由地改变分布，同时仍然保留他们的分析能力。

缺乏对多种语言的支持

客户通常需要除了 MapReduce Java 程序之外的多种语言支持来运行他们的作业。 SQL 和脚本等选项为获取答案提供了更大的灵活性，为组织和检索数据提供了更多的选项，以及将数据移动到分析框架的更快的方式。

使用难度

一段时间以来，人们一直抱怨 Hadoop 难以使用。尽管 Hadoop 的每个新版本都变得更简单、更强大，但这种批评仍然存在。 Hadoop 要求您了解 Java 和 MapReduce 编程模式，这对数据库管理员和具有传统脚本技能的人员来说是一个挑战。

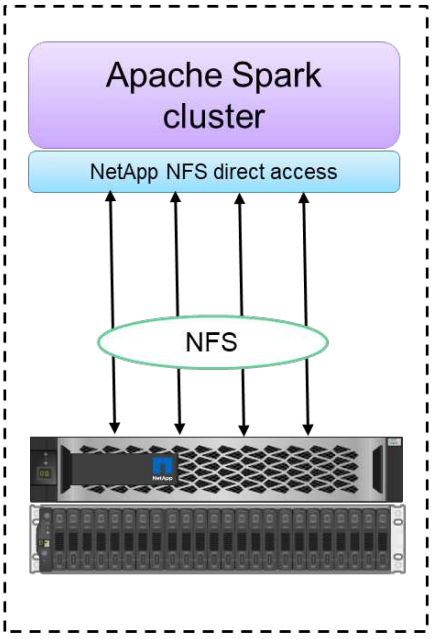
复杂的框架和工具

企业AI团队面临多重挑战。即使拥有专业的数据科学知识，不同部署生态系统和应用程序的工具和框架也可能无法简单地从一个转换到另一个。数据科学平台应该与基于 Spark 构建的相应大数据平台无缝集成，易于数据移动、可重复使用的模型、开箱即用的代码以及支持原型设计、验证、版本控制、共享、重用和快速将模型部署到生产的最佳实践的工具。

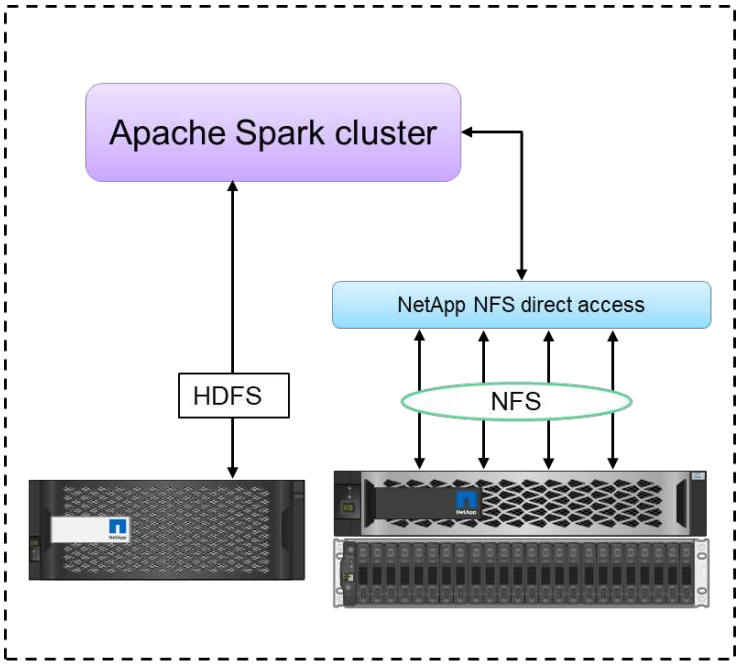
为什么选择**NetApp**?

NetApp可以通过以下方式改善您的 Spark 体验：

- NetApp NFS 直接访问（如下图所示）允许客户在其现有或新的 NFSv3 或 NFSv4 数据上运行大数据分析作业，而无需移动或复制数据。它可以防止数据的多次复制，并且无需将数据与源同步。
- 更高效的存储和更少的服务器复制。例如， NetApp E 系列 Hadoop 解决方案需要两个而不是三个数据副本，而FAS Hadoop 解决方案需要一个数据源，但不需要数据复制或副本。 NetApp存储解决方案还能减少服务器之间的流量。
- 驱动器和节点故障期间的 Hadoop 作业和集群行为更好。
- 更好的数据提取性能。



Configuration 1: NFS as primary storage



Configuration 2: HDFS and NFS in single Spark cluster

例如，在金融和医疗保健领域，数据从一个地方移动到另一个地方必须满足法律义务，这不是一件容易的事。在这种情况下， NetApp NFS 直接访问会从原始位置分析财务和医疗保健数据。另一个主要优势是，使用NetApp NFS 直接访问可以通过使用本机 Hadoop 命令简化 Hadoop 数据的保护，并利用NetApp丰富的数据管理产品组合实现数据保护工作流。

NetApp NFS 直接访问为 Hadoop/Spark 集群提供了两种部署选项：

- 默认情况下，Hadoop 或 Spark 集群使用 Hadoop 分布式文件系统 (HDFS) 进行数据存储和默认文件系统。NetApp NFS 直接访问可以用 NFS 存储替换默认的 HDFS 作为默认文件系统，从而实现对 NFS 数据的直接分析。

- 在另一个部署选项中， NetApp NFS 直接访问支持在单个 Hadoop 或 Spark 集群中将 NFS 与 HDFS 一起配置为附加存储。在这种情况下，客户可以通过 NFS 导出共享数据，并从同一个集群访问数据以及 HDFS 数据。

使用NetApp NFS 直接访问的主要优势包括：

- 从当前位置分析数据，从而避免将分析数据移动到 Hadoop 基础架构（如 HDFS）这一耗时耗能的任务。
- 将副本数量从三个减少到一个。
- 使用户能够分离计算和存储以独立扩展它们。
- 利用ONTAP丰富的数据管理功能提供企业数据保护。
- 通过 Hortonworks 数据平台认证。
- 支持混合数据分析部署。
- 利用动态多线程功能减少备份时间。

看"TR-4657： NetApp混合云数据解决方案 - 基于客户用例的 Spark 和 Hadoop"用于备份 Hadoop 数据、从云端到本地的备份和灾难恢复、对现有 Hadoop 数据进行 DevTest、数据保护和多云连接以及加速分析工作负载。

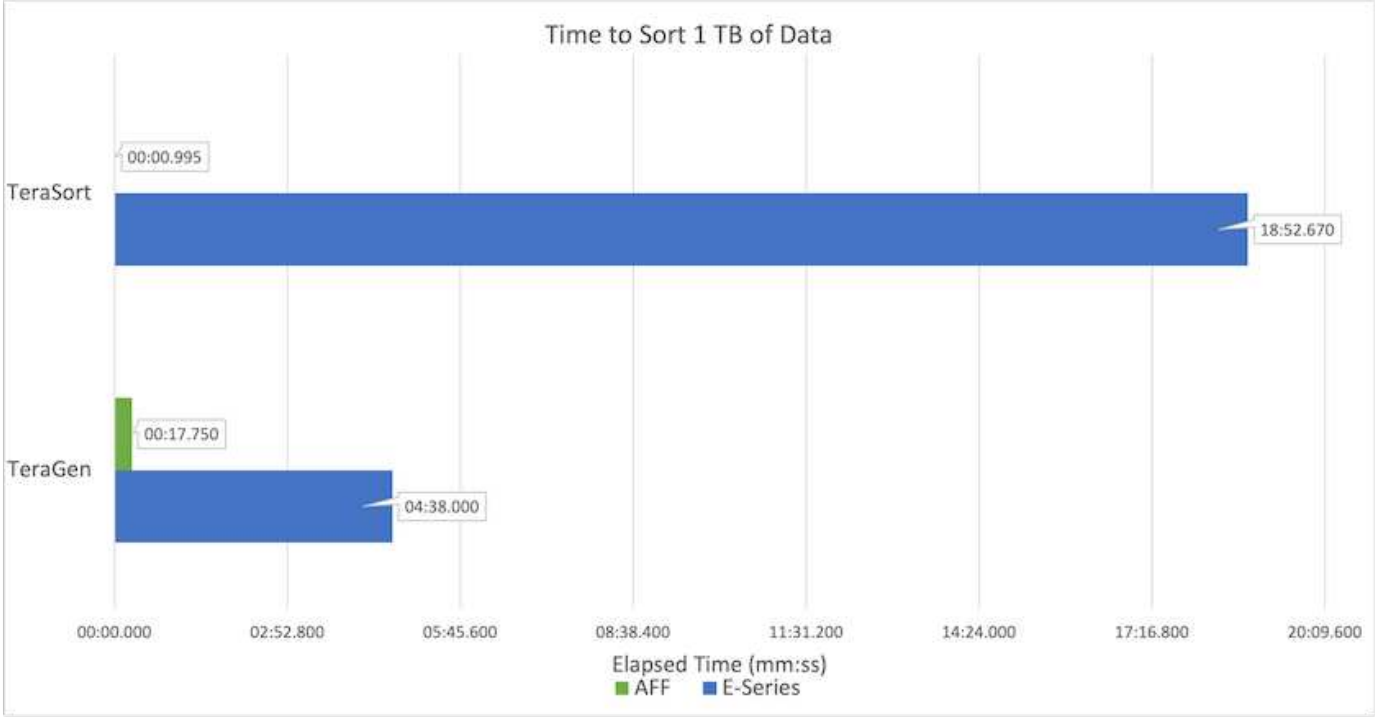
以下部分介绍了对 Spark 客户来说重要的存储功能。

存储分层

通过 Hadoop 存储分层，您可以根据存储策略存储具有不同存储类型的文件。存储类型包括 `hot`， `cold`， `warm`， `all_ssd`， `one_ssd`， 和 `lazy_persist`。

我们在NetApp AFF存储控制器和具有不同存储策略的 SSD 和 SAS 驱动器的 E 系列存储控制器上对 Hadoop 存储分层进行了验证。带有AFF-A800 的 Spark 集群有四个计算工作节点，而带有 E 系列的集群有八个。这主要是为了比较固态硬盘 (SSD) 与硬盘 (HDD) 的性能。

下图显示了NetApp针对 Hadoop SSD 的解决方案的性能。



- 基线 NL-SAS 配置使用八个计算节点和 96 个 NL-SAS 驱动器。此配置在 4 分 38 秒内生成了 1TB 的数据。看 "TR-3969 NetApp E系列Hadoop解决方案"有关集群和存储配置的详细信息。
- 使用 TeraGen，SSD 配置生成 1TB 数据的速度比 NL-SAS 配置快 15.66 倍。此外，SSD 配置使用了一半数量的计算节点和一半数量的磁盘驱动器（总共 24 个 SSd 驱动器）。根据作业完成时间，它几乎比 NL-SAS 配置快两倍。
- 使用 TeraSort，SSD 配置对 1TB 数据的排序速度比 NL-SAS 配置快 1138.36 倍。此外，SSD 配置使用了一半数量的计算节点和一半数量的磁盘驱动器（总共 24 个 SSd 驱动器）。因此，每个驱动器的速度大约比 NL-SAS 配置快三倍。
- 要点是从旋转磁盘过渡到全闪存可以提高性能。计算节点的数量不是瓶颈。借助 NetApp 的全闪存存储，运行时性能可以很好地扩展。
- 使用 NFS，数据在功能上相当于被集中在一起，这可以根据您的工作负载减少计算节点的数量。 Apache Spark 集群用户在更改计算节点数量时不必手动重新平衡数据。

性能扩展 - 横向扩展

当您需要从AFF解决方案中的 Hadoop 集群获取更多计算能力时，您可以添加具有适当数量存储控制器的数据节点。 NetApp建议从每个存储控制器阵列 4 个数据节点开始，然后根据工作负载特点将每个存储控制器的数据节点数量增加到 8 个。

AFF和FAS非常适合就地分析。根据计算要求，您可以添加节点管理器，并且无中断操作允许您按需添加存储控制器而无需停机。我们提供AFF和FAS的丰富功能，例如 NVME 媒体支持、保证效率、数据减少、QOS、预测分析、云分层、复制、云部署和安全性。为了帮助客户满足他们的需求， NetApp提供了文件系统分析、配额和机上负载平衡等功能，无需额外的许可费用。 NetApp在并发作业数量、更低的延迟、更简单的操作以及更高的每秒千兆字节吞吐量方面比我们的竞争对手表现更佳。此外， NetApp Cloud Volumes ONTAP可在三大云提供商上运行。

性能扩展 - 扩大规模

当您需要额外的存储容量时，扩展功能允许您将磁盘驱动器添加到AFF、 FAS和 E 系列系统。使用Cloud

Volumes ONTAP，将存储扩展到 PB 级别需要结合两个因素：将不常用的数据从块存储分层到对象存储，以及堆叠Cloud Volumes ONTAP许可证而无需额外的计算。

多种协议

NetApp系统支持大多数 Hadoop 部署协议，包括 SAS、iSCSI、FCP、InfiniBand 和 NFS。

运营和支持的解决方案

本文档中描述的 Hadoop 解决方案由NetApp支持。这些解决方案也经过了主要 Hadoop 分销商的认证。更多信息，请参阅 "Hortonworks"站点和 Cloudera "认证"和 "伙伴"站点。

目标受众

分析和数据科学领域涉及 IT 和商业的多个学科：

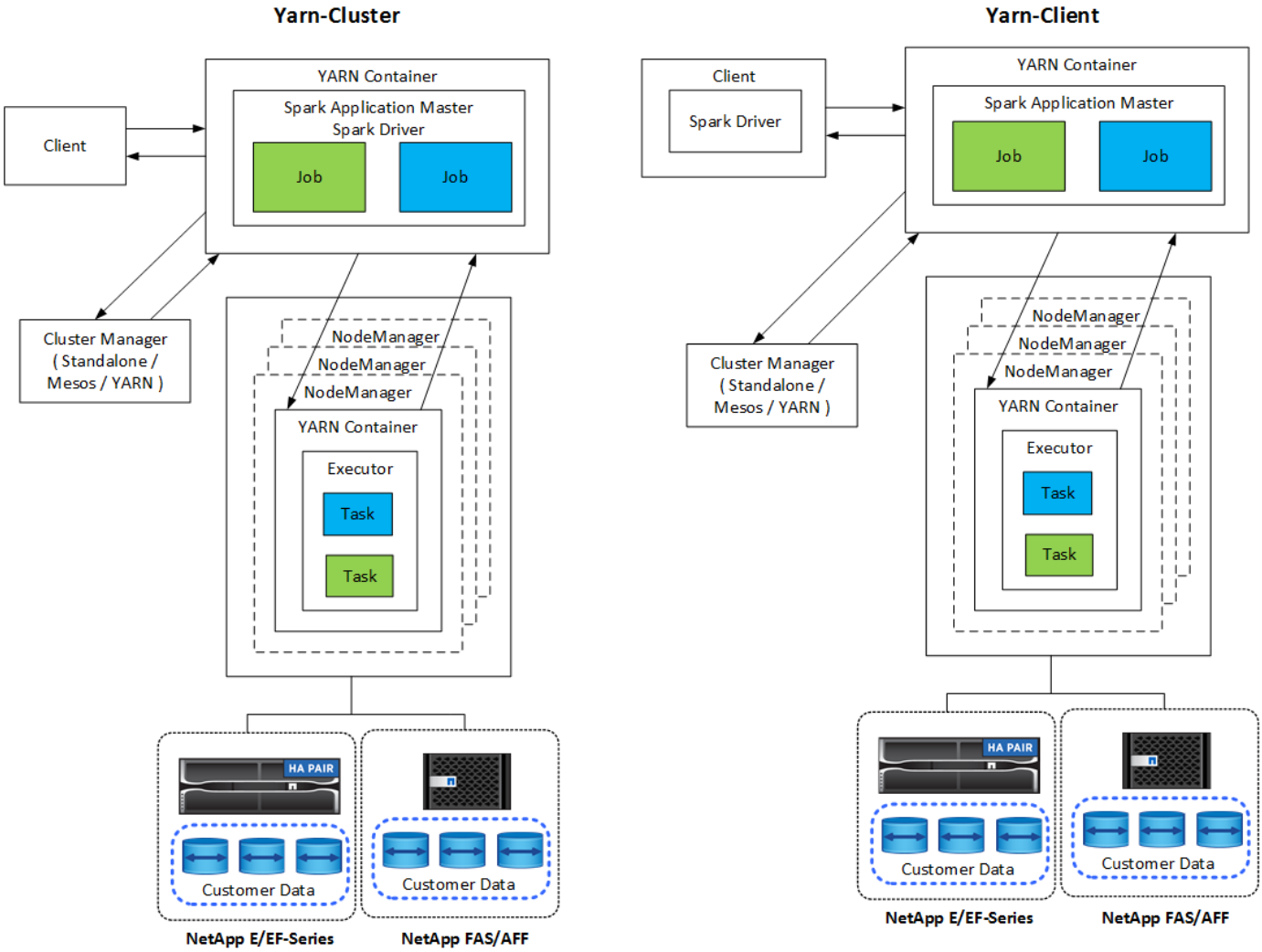
- 数据科学家需要灵活地使用他们选择的工具和库。
- 数据工程师需要知道数据如何流动以及位于何处。
- DevOps 工程师需要工具将新的 AI 和 ML 应用程序集成到他们的 CI 和 CD 管道中。
- 云管理员和架构师必须能够设置和管理混合云资源。
- 商业用户希望能够访问分析、AI、ML 和 DL 应用程序。

在本技术报告中，我们描述了NetApp AFF、E 系列、 StorageGRID、NFS 直接访问、Apache Spark、Horovod 和 Keras 如何帮助这些角色为企业带来价值。

解决方案技术

Apache Spark 是一种流行的编程框架，用于编写可直接与 Hadoop 分布式文件系统 (HDFS) 协同工作的 Hadoop 应用程序。 Spark 已准备好投入生产，支持流数据处理，并且比 MapReduce 更快。 Spark 具有可配置的内存数据缓存，可实现高效迭代，并且 Spark shell 具有交互性，可用于学习和探索数据。使用 Spark，您可以用 Python、Scala 或 Java 创建应用程序。 Spark 应用程序由一个或多个具有一个或多个任务的作业组成。

每个 Spark 应用程序都有一个 Spark 驱动程序。在 YARN-Client 模式下，驱动程序在客户端本地运行。在 YARN-Cluster 模式下，驱动程序在应用程序主机上的集群中运行。在集群模式下，即使客户端断开连接，应用程序仍继续运行。



有三个集群管理器：

- *独立。*该管理器是 Spark 的一部分，可以轻松设置集群。
- Apache Mesos。这是一个通用集群管理器，也运行 MapReduce 和其他应用程序。
- Hadoop YARN。这是 Hadoop 3 中的资源管理器。

弹性分布式数据集（RDD）是 Spark 的主要组件。 RDD 从集群内存中存储的数据中重新创建丢失和缺失的数据，并存储来自文件或以编程方式创建的初始数据。 RDD 是从文件、内存中的数据或另一个 RDD 创建的。 Spark 编程执行两种操作：转换和操作。转换基于现有 RDD 创建新的 RDD。操作从 RDD 返回一个值。

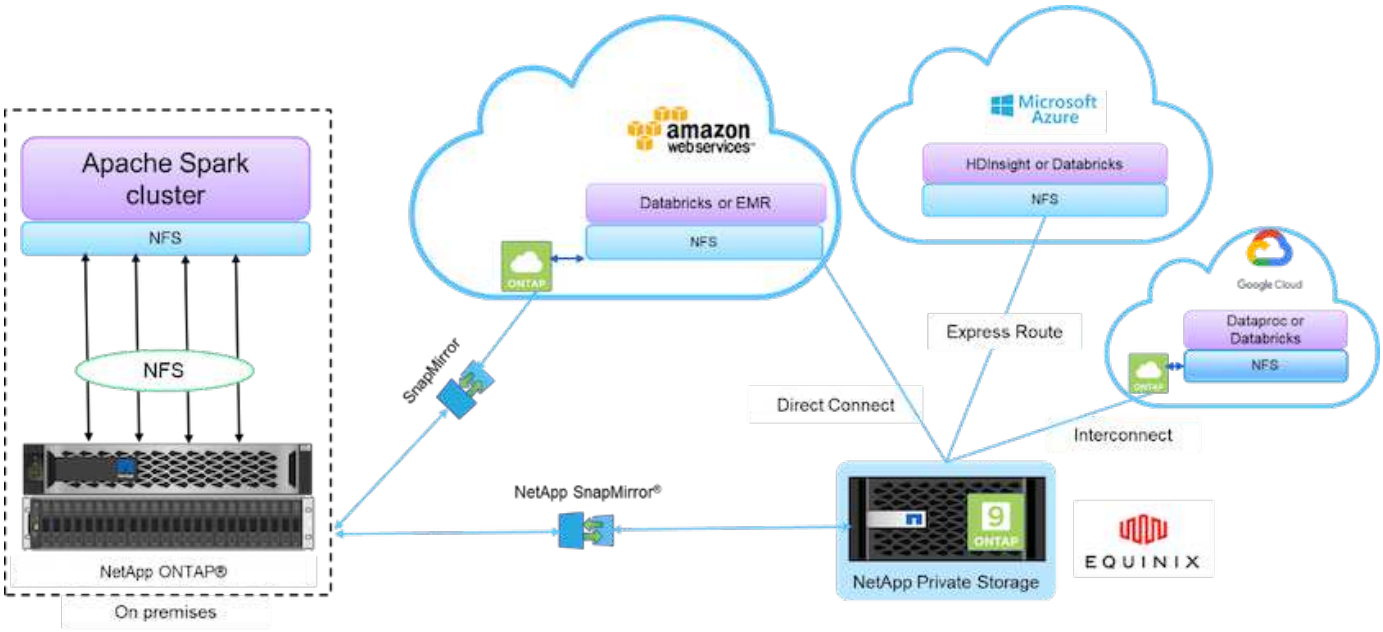
转换和操作也适用于 Spark 数据集和 DataFrames。数据集是分布式数据集合，它兼具 RDD 的优势（强类型、使用 lambda 函数）和 Spark SQL 优化执行引擎的优势。可以从 JVM 对象构建数据集，然后使用功能转换（map、flatMap、filter 等）进行操作。 DataFrame 是按命名列组织起来的数据集。它在概念上等同于关系数据库中的表或 R/Python 中的数据框。 DataFrames 可以从多种来源构建，例如结构化数据文件、Hive/HBase 中的表、本地或云端的外部数据库或现有的 RDD。

Spark 应用程序包括一个或多个 Spark 作业。作业在执行器中运行任务，执行器在 YARN 容器中运行。每个执行器都在单个容器中运行，并且执行器在应用程序的整个生命周期中都存在。应用程序启动后，执行器就固定了，YARN 不会调整已分配的容器的大小。执行器可以对内存数据同时运行任务。

NetApp Spark 解决方案概述

NetApp有三个存储产品组合： FAS/ AFF、E 系列和Cloud Volumes ONTAP。我们已经通过 Apache Spark 验证了适用于 Hadoop 解决方案的AFF和带有ONTAP存储系统的 E 系列。

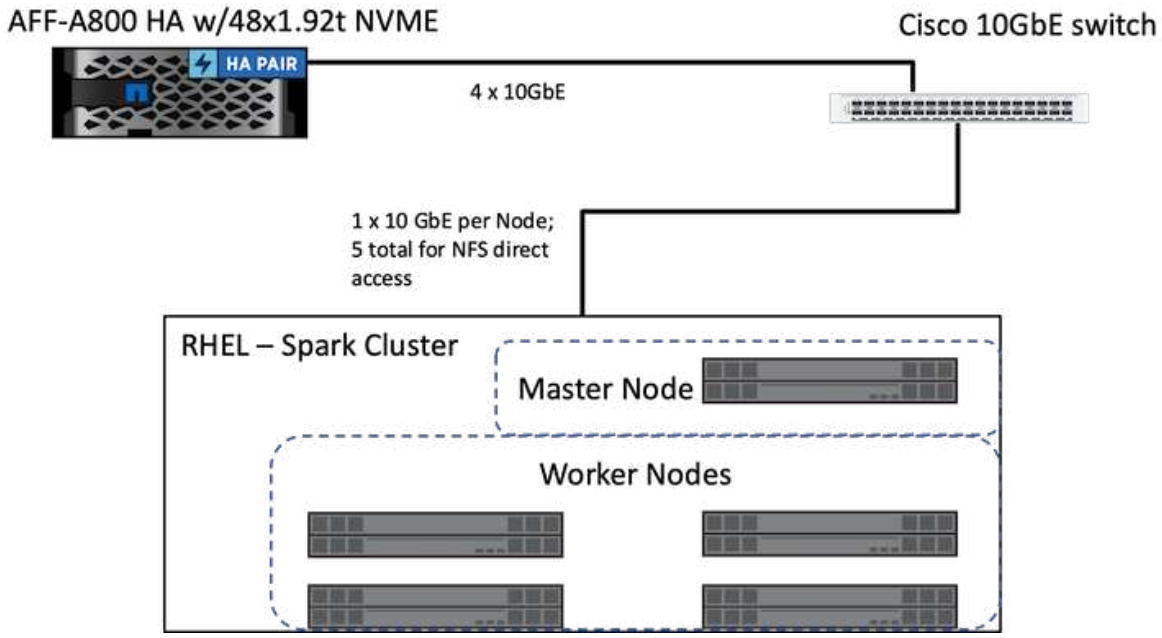
NetApp提供支持的数据结构集成了数据管理服务和应用程序（构建块），用于数据访问、控制、保护和安全，如下图所示。



上图中的构建块包括：

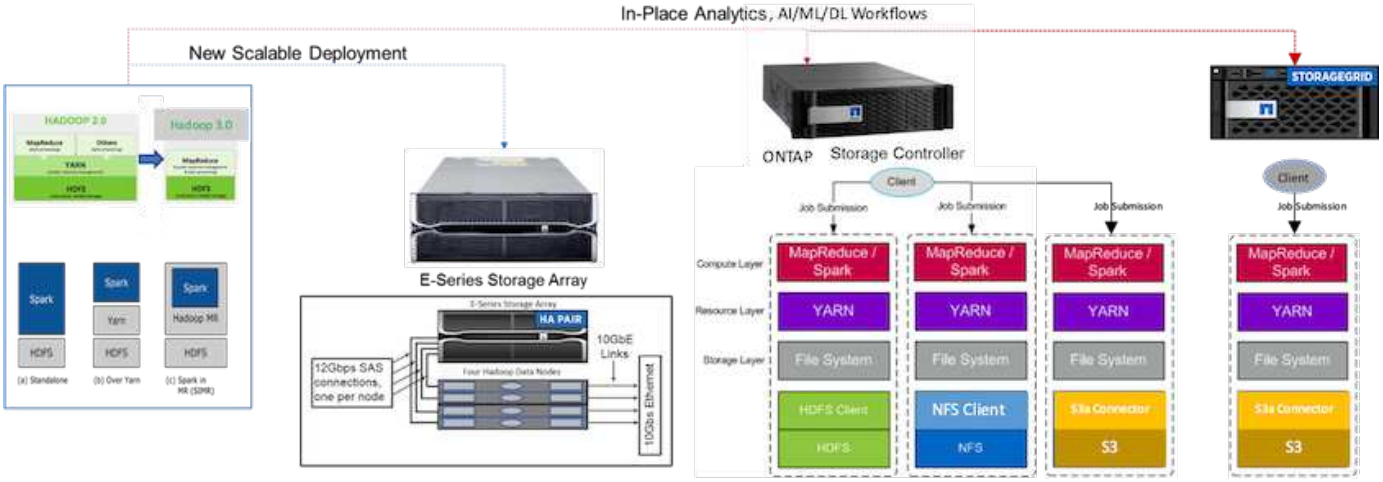
- * NetApp NFS 直接访问。*为最新的 Hadoop 和 Spark 集群提供对NetApp NFS 卷的直接访问，无需额外的软件或驱动程序要求。
- * NetApp Cloud Volumes ONTAP和Google Cloud NetApp Volumes 。*基于在 Amazon Web Services (AWS) 或 Microsoft Azure 云服务中的Azure NetApp Files (ANF) 中运行的ONTAP的软件定义连接存储。
- * NetApp SnapMirror技术。*在本地和ONTAP Cloud 或 NPS 实例之间提供数据保护功能。
- *云服务提供商。*这些提供商包括 AWS、Microsoft Azure、Google Cloud 和 IBM Cloud。
- *平台即服务 (PaaS)。*基于云的分析服务，例如 AWS 中的 Amazon Elastic MapReduce (EMR) 和 Databricks 以及 Microsoft Azure HDInsight 和 Azure Databricks。

下图描述了采用NetApp存储的 Spark 解决方案。

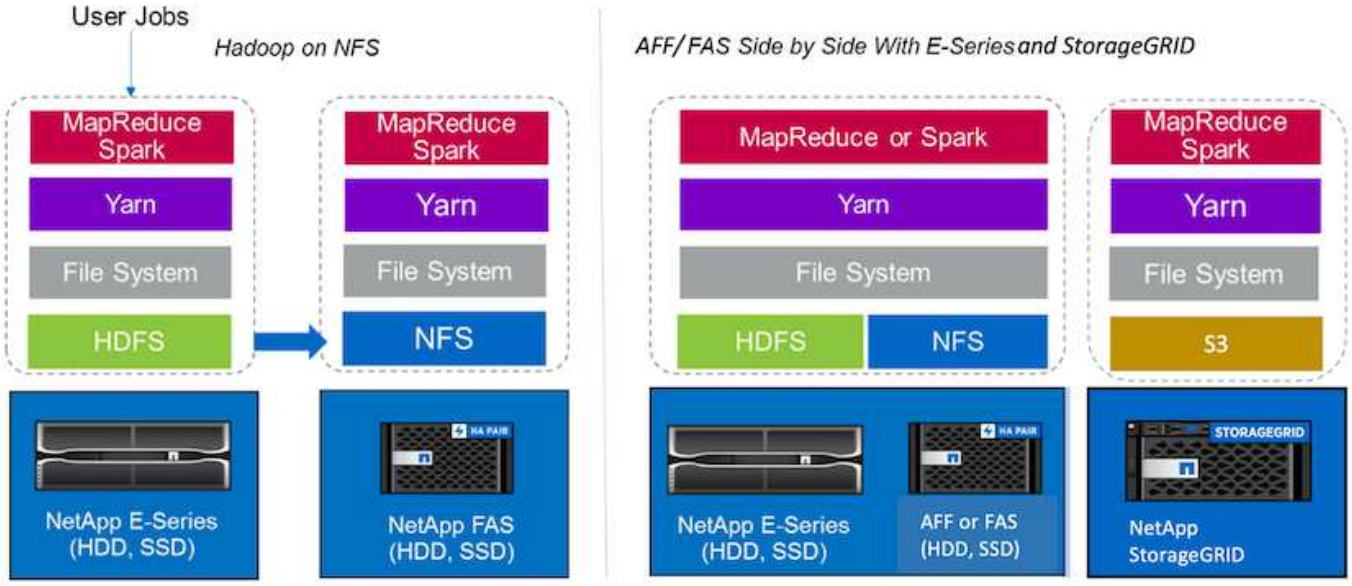


ONTAP Spark 解决方案使用NetApp NFS 直接访问协议进行就地分析以及通过访问现有生产数据来实现 AI、ML 和 DL 工作流。 Hadoop 节点可用的生产数据被导出以执行就地分析和 AI、ML 和 DL 作业。您可以使用NetApp NFS 直接访问或不使用 NetApp NFS 直接访问来访问 Hadoop 节点中要处理的数据。在 Spark 中，使用独立或 `yarn`集群管理器，您可以使用配置 `NFS 卷 `\file://<target_volume>`。我们用不同的数据集验证了三个用例。这些验证的详细信息在“测试结果”部分中介绍。 （外部参照）

下图描述了NetApp Apache Spark/Hadoop 存储定位。



我们确定了 E 系列 Spark 解决方案、 AFF/ FAS ONTAP Spark 解决方案和StorageGRID Spark 解决方案的独特功能，并进行了详细的验证和测试。根据我们的观察， NetApp建议对于绿地安装和新的可扩展部署使用 E 系列解决方案，对于使用现有 NFS 数据的就地分析、AI、ML 和 DL 工作负载使用AFF/ FAS解决方案，对于需要对象存储时的 AI、ML、DL 和现代数据分析使用StorageGRID 。



数据湖是原生形式的大型数据集的存储库，可用于分析、AI、ML 和 DL 作业。我们为 E 系列、 AFF/ FAS 和StorageGRID SG6060 Spark 解决方案构建了一个数据湖存储库。 E 系列系统提供对 Hadoop Spark 集群的 HDFS 访问，而现有生产数据则通过 NFS 直接访问协议访问 Hadoop 集群。对于驻留在对象存储中的数据集，NetApp StorageGRID提供 S3 和 S3a 安全访问。

用例摘要

本页描述了可以使用该解决方案的不同领域。

流数据

Apache Spark 可以处理流数据，用于流式提取、转换和加载 (ETL) 过程；数据丰富；触发事件检测；以及复杂的会话分析：

- 流式 ETL。*数据在被推送到数据存储之前会被不断地清理和汇总。 Netflix 使用 Kafka 和 Spark 流构建实时在线电影推荐和数据监控解决方案，每天可以处理来自不同数据源的数十亿个事件。然而，用于批处理的传统 ETL 的处理方式有所不同。首先读取该数据，然后将其转换为数据库格式，再写入数据库。
- 数据丰富。 Spark 流使用静态数据丰富实时数据，以实现更实时的数据分析。例如，在线广告商可以根据客户行为信息投放个性化、有针对性的广告。
- 触发事件检测。 Spark 流允许您检测并快速响应可能表明存在严重问题的异常行为。例如，金融机构使用触发器来检测和阻止欺诈交易，医院使用触发器来检测患者生命体征中检测到的危险健康变化。
- 复杂的会话分析。 Spark 流收集用户登录网站或应用程序后的活动等事件，然后对其进行分组和分析。例如，Netflix 使用此功能提供实时电影推荐。

有关流数据配置、Confluent Kafka 验证和性能测试的更多内容，请参阅"TR-4912： NetApp Confluent Kafka 分层存储的最佳实践指南"。

机器学习

Spark 集成框架可帮助您使用机器学习库 (MLlib) 对数据集运行重复查询。 MLlib 用于聚类、分类和降维等领域，用于一些常见的大数据功能，例如预测智能、用于营销目的的客户细分和情感分析。 MLlib 用于网络安全，对数据包进行实时检查，以发现恶意活动的迹象。它可以帮助安全提供商了解新的威胁并领先于黑客，同时实时保

护他们的客户。

深度学习

TensorFlow 是业界流行的深度学习框架。 TensorFlow支持在CPU或GPU集群上进行分布式训练。这种分布式训练允许用户在具有大量深层的数据上运行它。

直到最近，如果我们想将 TensorFlow 与 Apache Spark 一起使用，我们需要在 PySpark 中为 TensorFlow 执行所有必要的 ETL，然后将数据写入中间存储。然后，该数据将被加载到 TensorFlow 集群上，用于实际的训练过程。此工作流程要求用户维护两个不同的集群，一个用于 ETL，一个用于 TensorFlow 的分布式训练。运行和维护多个集群通常很繁琐且耗时。

早期 Spark 版本中的 DataFrames 和 RDD 不太适合深度学习，因为随机访问受到限制。在带有氢项目的 Spark 3.0 中，添加了对深度学习框架的原生支持。这种方法允许在 Spark 集群上进行非基于 MapReduce 的调度。

交互式分析

Apache Spark 的速度足够快，可以使用 Spark 以外的开发语言（包括 SQL、R 和 Python）执行探索性查询而无需采样。 Spark 使用可视化工具来处理复杂数据并以交互方式进行可视化。具有结构化流的 Spark 对网络分析中的实时数据执行交互式查询，使您能够对网络访问者的当前会话运行交互式查询。

推荐系统

多年来，随着企业和消费者对网上购物、在线娱乐和许多其他行业的巨大变化做出了反应，推荐系统给我们的生活带来了巨大的变化。事实上，这些系统是人工智能在生产中最明显的成功案例之一。在许多实际用例中，推荐系统与对话式 AI 或与 NLP 后端交互的聊天机器人相结合，以获取相关信息并产生有用的推论。

如今，许多零售商正在采用更新的商业模式，例如网上购买、店内取货、路边取货、自助结账、扫描即走等等。这些模式在新冠疫情期间尤为突出，因为它们让消费者的购物更加安全、更加便捷。人工智能对于这些日益增长的数字趋势至关重要，这些趋势受到消费者行为的影响，反之亦然。为了满足消费者日益增长的需求、增强客户体验、提高运营效率和增加收入， NetApp帮助其企业客户和企业使用机器学习和深度学习算法来设计更快、更准确的推荐系统。

有几种流行的技术用于提供推荐，包括协同过滤、基于内容的系统、深度学习推荐模型 (DLRM) 和混合技术。客户之前利用 PySpark 实现协同过滤来创建推荐系统。 Spark MLlib 实现了用于协同过滤的交替最小二乘法 (ALS)，这是 DLRM 兴起之前企业中非常流行的算法。

自然语言处理

对话式人工智能是通过自然语言处理 (NLP) 实现的，它是帮助计算机与人类交流的人工智能的一个分支。 NLP 在每个垂直行业和许多用例中都很普遍，从智能助手和聊天机器人到谷歌搜索和预测文本。根据 "Gartner"预测到2022年，70%的人将每天与对话式人工智能平台进行互动。为了实现人与机器之间的高质量对话，响应必须快速、智能且听起来自然。

客户需要大量数据来处理和训练他们的 NLP 和自动语音识别 (ASR) 模型。他们还需要在边缘、核心和云端移动数据，并且需要在几毫秒内进行推理的能力，以与人类建立自然的交流。 NetApp AI 和 Apache Spark 是计算、存储、数据处理、模型训练、微调和部署的理想组合。

情感分析是 NLP 中的一个研究领域，它从文本中提取积极、消极或中性情感。情绪分析有多种用例，从确定支持中心员工与呼叫者对话的表现到提供适当的自动聊天机器人响应。它还被用来根据公司代表和季度收益电话会议上的听众之间的互动来预测公司的股价。此外，情绪分析可用于确定客户对品牌提供的产品、服务或支持的看法。

我们使用了 "Spark NLP"来自的图书馆 "约翰·斯诺实验室"加载预训练管道和 Transformer (BERT) 模型的双向编码器表示，包括 "财经新闻情绪"和 "FinBERT"，大规模执行标记化、命名实体识别、模型训练、拟合和情感分析。 Spark NLP 是唯一一个正在生产中的开源 NLP 库，它提供最先进的转换器，例如 BERT、ALBERT、ELECTRA、XLNet、DistilBERT、RoBERTa、DeBERTa、XLM-RoBERTa、Longformer、ELMO、Universal Sentence Encoder、Google T5、MarianMT 和 GPT2。该库不仅适用于 Python 和 R，还可以通过原生扩展 Apache Spark 在 JVM 生态系统（Java、Scala 和 Kotlin）中大规模运行。

主要的 AI、ML 和 DL 用例和架构

主要的 AI、ML 和 DL 用例和方法可分为以下几部分：

Spark NLP 管道和 TensorFlow 分布式推理

以下列表包含数据科学界在不同发展水平下采用的最流行的开源 NLP 库：

- "自然语言工具包（NLTK）"。所有 NLP 技术的完整工具包。它自 21 世纪初以来一直得到维护。
- "文本块"。基于 NLTK 和 Pattern 构建的易于使用的 NLP 工具 Python API。
- "斯坦福核心 NLP"。斯坦福 NLP 小组开发的 Java NLP 服务和包。
- "Gensim"。人类主题建模最初是捷克数字数学图书馆项目的 Python 脚本集合。
- "SpaCy"。使用 Python 和 Cython 实现端到端工业 NLP 工作流程，并为 Transformer 提供 GPU 加速。
- "快文"。Facebook 的 AI 研究 (FAIR) 实验室创建的免费、轻量级、开源 NLP 库，用于学习词嵌入和句子分类。

Spark NLP 是针对所有 NLP 任务和要求的单一、统一的解决方案，可为实际生产用例提供可扩展、高性能和高精度的 NLP 软件。它利用迁移学习并在研究和跨行业中实施最新的最先进的算法和模型。由于 Spark 缺乏对上述库的全面支持，Spark NLP 建立在 "Spark 机器学习"利用 Spark 的通用内存分布式数据处理引擎作为关键任务生产工作流的企业级 NLP 库。它的注释器利用基于规则的算法、机器学习和 TensorFlow 来支持深度学习的实现。这涵盖了常见的 NLP 任务，包括但不限于标记化、词形还原、词干提取、词性标注、命名实体识别、拼写检查和情感分析。

来自 Transformer 的双向编码器表示 (BERT) 是一种基于 Transformer 的 NLP 机器学习技术。它推广了预训练和微调的概念。 BERT 中的 Transformer 架构源自机器翻译，它比基于循环神经网络 (RNN) 的语言模型更好地模拟长期依赖关系。它还引入了掩蔽语言建模 (MLM) 任务，其中随机 15% 的所有标记被掩蔽，并且模型对其进行预测，从而实现真正的双向性。

由于该领域的专业语言和缺乏标记数据，金融情绪分析具有挑战性。 FinBERT 是一种基于预训练 BERT 的语言模型，已在以下领域进行了调整： "路透社 TRC2"，一个金融语料库，并使用标记数据进行微调（ "金融短语库") 用于金融情绪分类。研究人员从包含金融术语的新闻文章中提取了 4,500 个句子。然后，16位具有金融背景的专家和硕士生将这些句子标记为肯定、中性和否定。我们构建了一个端到端的 Spark 工作流程，使用 FinBERT 和其他两个预先训练的流程来分析 2016 年至 2020 年纳斯达克十大公司收益电话会议记录的情绪， "解释文档 DL" ）来自 Spark NLP。

Spark NLP 的底层深度学习引擎是 TensorFlow，这是一个端到端的开源机器学习平台，可以轻松构建模型、在任何地方进行强大的 ML 生产以及进行强大的研究实验。因此，在 Spark 中执行管道时 `yarn cluster`模式，我们本质上是在运行分布式 TensorFlow，数据和模型在一个主节点和多个工作节点上并行化，并在集群上安装网络附加存储。

Horovod分布式训练

与 MapReduce 相关的性能的核心 Hadoop 验证是使用 TeraGen、TeraSort、TeraValidate 和 DFSIO（读写）执行的。 TeraGen 和 TeraSort 验证结果如下 "NetApp E系列Hadoop解决方案"以及AFF的“存储分层”部分。

根据客户要求，我们认为使用 Spark 进行分布式训练是各种用例中最重要的用例之一。在本文档中，我们使用了 "Spark 上的 Hovorod"使用NetApp All Flash FAS (AFF) 存储控制器、 Azure NetApp Files和StorageGRID来验证 Spark 与NetApp本地、云原生和混合云解决方案的性能。

Horovod on Spark 包为 Horovod 提供了一个便捷的包装器，使得在 Spark 集群中运行分布式训练工作负载变得简单，从而实现了紧密的模型设计循环，其中数据处理、模型训练和模型评估都在训练和推理数据所在的 Spark 中完成。

有两个用于在 Spark 上运行 Horovod 的 API：高级 Estimator API 和低级 Run API。尽管两者都使用相同的底层机制在 Spark 执行器上启动 Horovod，但 Estimator API 抽象了数据处理、模型训练循环、模型检查点、指标收集和分布式训练。我们使用 Horovod Spark Estimators、TensorFlow 和 Keras 进行端到端数据准备和分布式训练工作流程，基于 "Kaggle Rossmann 商店销售"竞赛。

脚本 `keras_spark_horovod_rossmann_estimator.py`可以在以下部分找到"每个主要用例的 Python 脚本。"它包含三个部分：

- 第一部分对 Kaggle 提供并由社区收集的一组初始 CSV 文件执行各种数据预处理步骤。输入数据被分成一个训练集， `Validation`子集和测试数据集。
- 第二部分定义了一个具有对数 S 型激活函数和 Adam 优化器的 Keras 深度神经网络 (DNN) 模型，并使用 Spark 上的 Horovod 对模型进行分布式训练。
- 第三部分使用最小化验证集总体平均绝对误差的最佳模型对测试数据集进行预测。然后创建一个输出 CSV 文件。

请参阅"机器学习"用于各种运行时比较结果。

使用 **Keras** 进行多任务深度学习以进行 **CTR** 预测

随着机器学习平台和应用的最新进展，人们将大量注意力放在了大规模学习上。点击率（CTR）定义为每百次在线广告展示的平均点击次数（以百分比表示）。它被广泛采用为各个行业垂直领域和用例的关键指标，包括数字营销、零售、电子商务和服务提供商。有关 CTR 和分布式训练性能结果的应用的更多详细信息，请参阅"CTR预测性能的深度学习模型"部分。

在本技术报告中，我们使用了 "Criteo Terabyte 点击日志数据集"（参见 TR-4904）用于多工作者分布式深度学习，使用 Keras 构建具有深度和交叉网络 (DCN) 模型的 Spark 工作流，并将其对数损失误差函数方面的性能与基线 Spark ML 逻辑回归模型进行比较。 DCN 有效地捕获有界度的有效特征交互，学习高度非线性交互，不需要手动特征工程或穷举搜索，并且计算成本低。

网络规模推荐系统的数据大多是离散的和分类的，导致特征空间庞大且稀疏，这对于特征探索来说是一个挑战。这使得大多数大型系统仅限于逻辑回归等线性模型。然而，识别经常预测的特征并同时探索看不见的或罕见的交叉特征是做出良好预测的关键。线性模型简单、可解释、易于扩展，但其表达能力有限。

另一方面，交叉特征已被证明对提高模型的表现力具有重要意义。不幸的是，通常需要手动特征工程或详尽搜索来识别这些特征。推广到看不见的特征交互通常很困难。使用像 DCN 这样的交叉神经网络可以通过以自动方式明确应用特征交叉来避免特定于任务的特征工程。交叉网络由多层组成，其中最高程度的交互可由层深度决定。每一层都会在现有交互的基础上产生更高阶的交互，并保留前几层的交互。

深度神经网络 (DNN) 有望捕捉跨特征的非常复杂的交互。然而，与 DCN 相比，它需要的参数几乎多一个数量

级，无法明确地形成交叉特征，并且可能无法有效地学习某些类型的特征交叉。交叉网络内存效率高并且易于实现。联合训练交叉和 DNN 组件可以有效地捕获预测特征交互并在 Criteo CTR 数据集上提供最先进的性能。

DCN 模型从嵌入和堆叠层开始，然后并行连接交叉网络和深度网络。接下来是最终的组合层，它将两个网络的输出组合在一起。您的输入数据可以是具有稀疏和密集特征的向量。在 Spark 中，库包含类型 `SparseVector`。因此，用户区分两者并在调用各自的函数和方法时要小心，这一点很重要。在 CTR 预测等网络规模推荐系统中，输入大多是分类特征，例如 `'country=usa'`。这些特征通常被编码为独热向量，例如，`'[0,1,0, …]'` 。独热编码（OHE） `SparseVector`在处理词汇不断变化和增长的真实世界数据集时很有用。我们修改了示例 "深度点击率"处理大型词汇表，在 DCN 的嵌入和堆叠层中创建嵌入向量。

这 "Criteo 展示广告数据集"预测广告点击率。它有 13 个整数特征和 26 个分类特征，其中每个类别都有很高的基数。对于该数据集，由于输入规模较大，对数损失 0.001 的改进实际上具有显著意义。对于庞大的用户群，预测准确度的微小提升都可能带来公司收入的大幅增加。该数据集包含 7 天内 11GB 的用户日志，相当于约 4100 万条记录。我们使用了 Spark `dataFrame.randomSplit()function`随机分割数据用于训练（80%）、交叉验证（10%），剩余 10% 用于测试。

DCN 是使用 Keras 在 TensorFlow 上实现的。使用DCN实现模型训练过程主要有四个部分：

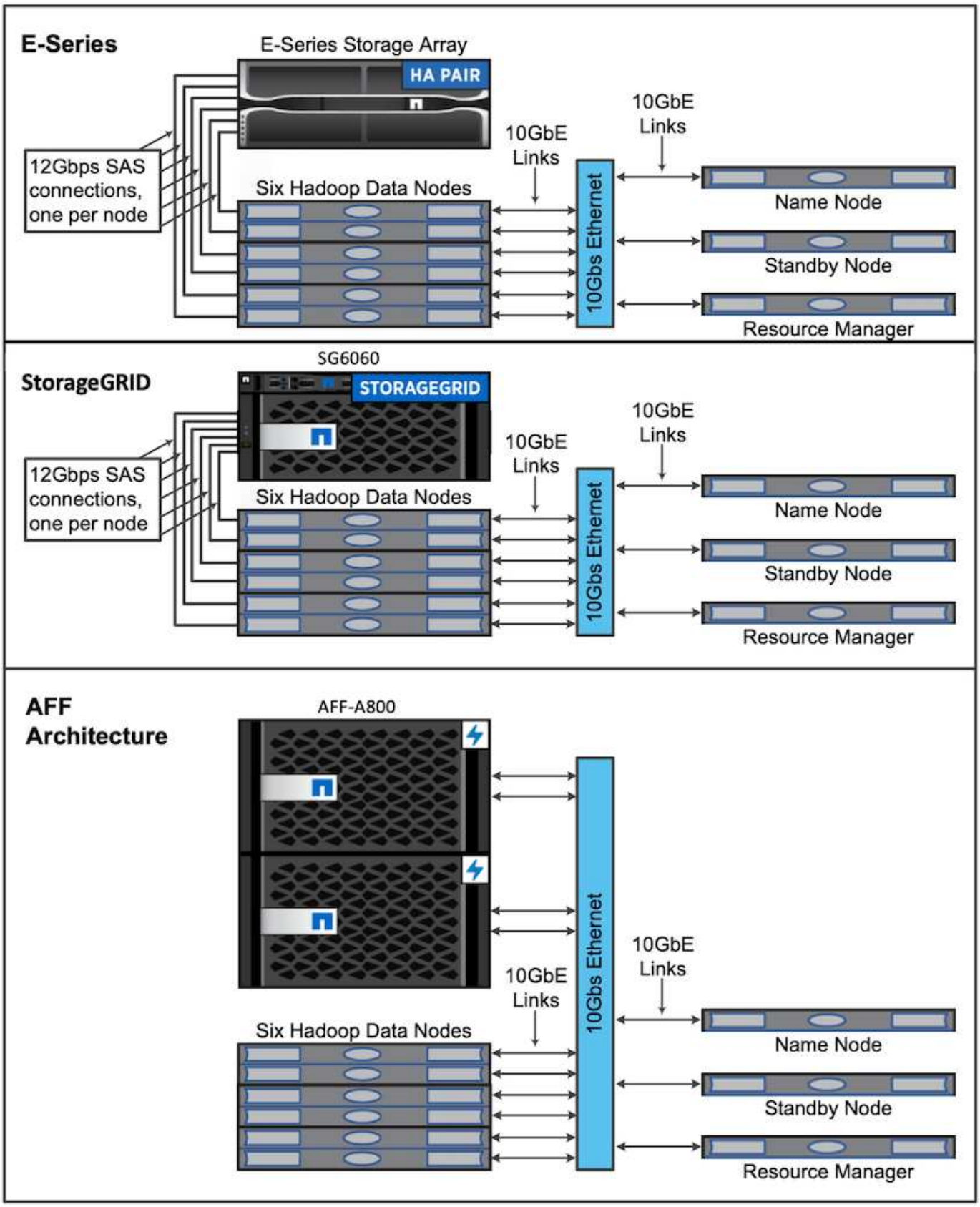
- *数据处理和嵌入。*通过应用对数变换对实值特征进行规范化。对于分类特征，我们将特征嵌入到维度为 6×(类别基数)1/4 的密集向量中。连接所有嵌入将产生一个维度为 1026 的向量。
- *优化。*我们利用 Adam 优化器进行了小批量随机优化。批次大小设置为 512。对深度网络进行批量归一化，梯度裁剪范数设为100。
- *正则化。*我们采用了提前停止的方法，因为 L2 正则化或 dropout 被发现无效。
- 超参数。我们报告基于对隐藏层数量、隐藏层大小、初始学习率和交叉层数量的网格搜索的结果。隐藏层的数量范围为 2 至 5，隐藏层大小范围为 32 至 1024。对于DCN，交叉层的数量为1至6。初始学习率从 0.0001 调整到 0.001，增量为 0.0001。所有实验均在训练步骤 150,000 时提前停止，超过该步骤后就会开始出现过度拟合。

除了 DCN 之外，我们还测试了其他流行的深度学习模型来进行 CTR 预估，包括 "DeepFM"， "自动输入" ， 和 "DCN v2"。

用于验证的架构

为了进行此验证，我们使用了四个工作节点和一个主节点以及一个AFF-A800 HA 对。所有集群成员都通过 10GbE 网络交换机连接。

为了验证NetApp Spark 解决方案，我们使用了三种不同的存储控制器：E5760、E5724 和AFF-A800。 E系列存储控制器通过12Gbps SAS连接连接到五个数据节点。 AFF HA 对存储控制器通过 10GbE 连接向 Hadoop 工作节点提供导出的 NFS 卷。 Hadoop 集群成员通过 E 系列、 AFF和StorageGRID Hadoop 解决方案中的 10GbE 连接进行连接。



测试结果

我们使用 TeraGen 基准测试工具中的 TeraSort 和 TeraValidate 脚本来测量 E5760

、E5724 和AFF-A800 配置的 Spark 性能验证。此外，还测试了三个主要用例：Spark NLP 管道和 TensorFlow 分布式训练、Horovod 分布式训练以及使用 Keras 进行 DeepFM CTR 预测的多工深度学习。

对于 E 系列和StorageGRID验证，我们使用了 Hadoop 复制因子 2。对于AFF验证，我们仅使用一个数据源。

下表列出了 Spark 性能验证的硬件配置。

类型	**Hadoop** 工作节点	驱动器类型	每个节点的驱动器	存储控制器
SG6060	4	SAS	12	单个高可用性 (HA) 对
E5760	4	SAS	60	单个 HA 对
E5724	4	SAS	24	单个 HA 对
AFF800	4	SSD	6	单个 HA 对

下表列出了软件要求。

软件	版本
RHEL	7.9
OpenJDK 运行环境	1.8.0
OpenJDK 64 位服务器虚拟机	25.302
Git	2.24.1
GCC/G++	11.2.1
火花	3.2.1
PySpark	3.1.2
SparkNLP	3.4.2
TensorFlow	2.9.0
喀拉拉	2.9.0
霍罗沃德	0.24.3

金融情绪分析

我们发表了"TR-4910：利用NetApp AI 对客户沟通进行情绪分析"，其中使用 "NetApp DataOps 工具包"、 AFF 存储和NVIDIA DGX 系统。该管道利用 DataOps Toolkit 执行批量音频信号处理、自动语音识别 (ASR)、迁移学习和情绪分析， "NVIDIA Riva SDK" ，以及 "道框架"。将情绪分析用例扩展到金融服务行业，我们构建了 SparkNLP 工作流程，为各种 NLP 任务（例如命名实体识别）加载了三个 BERT 模型，并获得了纳斯达克十大公司季度收益电话会议的句子级情绪。

以下脚本 `sentiment_analysis_spark. py`使用 FinBERT 模型处理 HDFS 中的转录本，并产生正面、中性和负面情绪计数，如下表所示：

```
-bash-4.2$ time ~/anaconda3/bin/spark-submit
--packages com.johnsnowlabs.nlp:spark-nlp_2.12:3.4.3
--master yarn
--executor-memory 5g
--executor-cores 1
--num-executors 160
--conf spark.driver.extraJavaOptions="-Xss10m -XX:MaxPermSize=1024M"
--conf spark.executor.extraJavaOptions="-Xss10m -XX:MaxPermSize=512M"
/sparkusecase/tr-4570-nlp/sentiment_analysis_spark.py
hdfs:///data1/Transcripts/
> ./sentiment_analysis_hdfs.log 2>&1
real13m14.300s
user557m11.319s
sys4m47.676s
```

下表列出了 2016 年至 2020 年纳斯达克十大公司的收益电话会议句子级情绪分析。

情绪计数和百分比	全部 **10** 家公司	苹果	**AMD**	亚马逊	思科	谷歌	国际贸易中心	微软	**NVDA**
正计数	7447	1567	743	290	682	826	824	904	417
中立计数	64067	6856	7596	5086	6650	5914	6099	5715	6189
负数	1787	253	213	84	189	97	282	202	89
未分类的计数	196	0	0	76	0	0	0	1	0
（总数）	73497	8676	8552	5536	7521	6837	7205	6822	6695

从百分比来看，首席执行官和首席财务官所说的大多数句子都是事实，因此带有中立的情绪。在收益电话会议期间，分析师提出的问题可能会传达积极或消极的情绪。值得进一步定量研究负面或正面情绪如何影响交易当天或次日的股票价格。

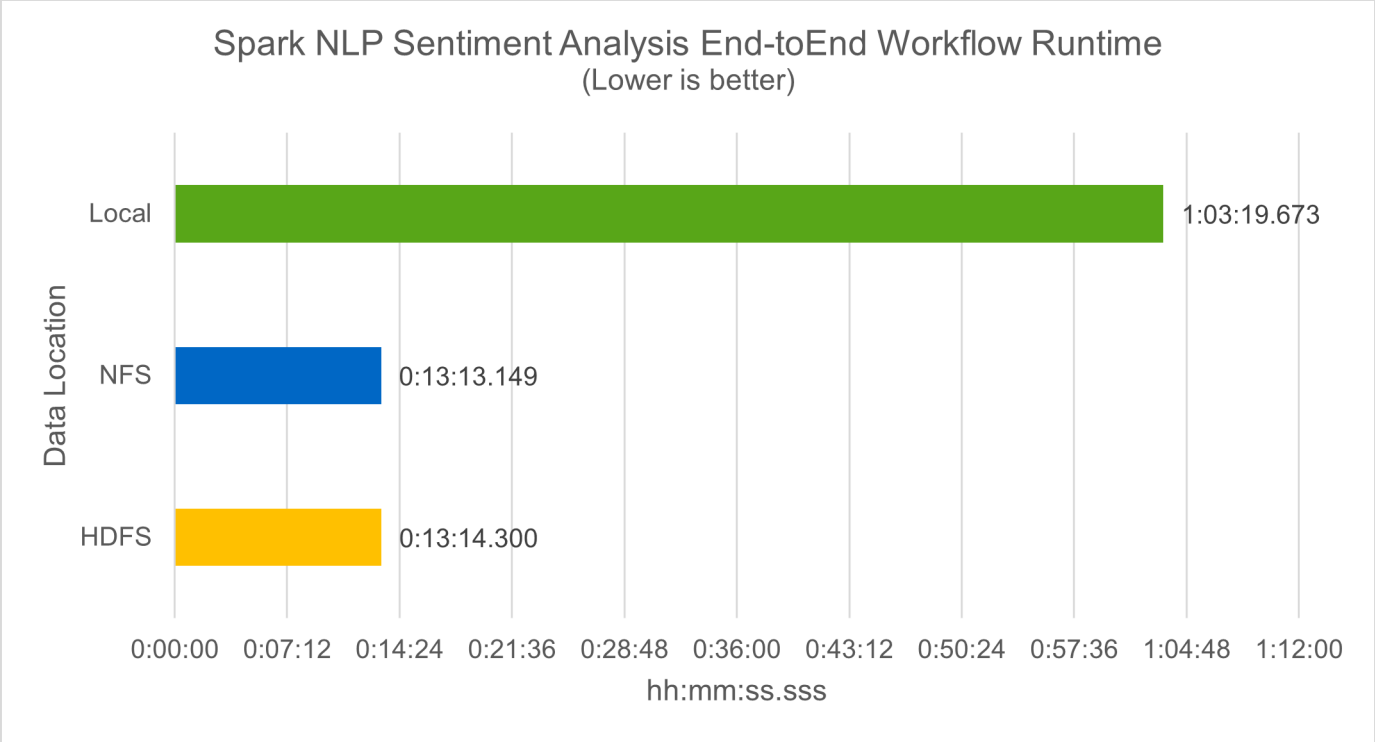
下表列出了纳斯达克十大公司的句子级情感分析，以百分比表示。

情绪百分比	全部 **10** 家公司	苹果	**AMD**	亚马逊	思科	谷歌	国际贸易中心	微软	**NVDA**
积极的	10.13%	18.06%	8.69%	5.24%	9.07%	12.08%	11.44%	13.25%	6.23%
中性的	87.17%	79.02%	88.82%	91.87%	88.42%	86.50%	84.65%	83.77%	92.44%
消极的	2.43%	2.92%	2.49%	1.52%	2.51%	1.42%	3.91%	2.96%	1.33%
未分类	0.27%	0%	0%	1.37%	0%	0%	0%	0.01%	0%

在工作流运行时方面，我们看到了显著的 4.78 倍改进 `local`模式到 HDFS 中的分布式环境，并通过利用 NFS 进一步提高 0.14%。

```
-bash-4.2$ time ~/anaconda3/bin/spark-submit
--packages com.johnsnowlabs.nlp:spark-nlp_2.12:3.4.3
--master yarn
--executor-memory 5g
--executor-cores 1
--num-executors 160
--conf spark.driver.extraJavaOptions="-Xss10m -XX:MaxPermSize=1024M"
--conf spark.executor.extraJavaOptions="-Xss10m -XX:MaxPermSize=512M"
/sparkusecase/tr-4570-nlp/sentiment_analysis_spark.py
file:///sparkdemo/sparknlp/Transcripts/
> ./sentiment_analysis_nfs.log 2>&1
real13m13.149s
user537m50.148s
sys4m46.173s
```

如下图所示，数据和模型并行提高了数据处理和分布式 TensorFlow 模型推理的速度。 NFS 中的数据位置产生了稍微更好的运行时间，因为工作流瓶颈是预训练模型的下载。如果我们增加成绩单数据集的大小，NFS 的优势就更加明显。



Horovod 性能的分布式训练

以下命令使用单个 `master`具有 `160` 个执行器的节点，每个执行器都有一个核心。执行器内存限制为 `5GB`，以避免内存不足错误。请参阅"针对每个主要用例的 `Python` 脚本"有关数据处理、模型训练和模型准确率计算的更多详细信息，请参阅 `keras_spark_horovod_rossmann_estimator.py`。

```
(base) [root@n138 horovod]# time spark-submit
--master local
--executor-memory 5g
--executor-cores 1
--num-executors 160
/sparkusecase/horovod/keras_spark_horovod_rossmann_estimator.py
--epochs 10
--data-dir file:///sparkusecase/horovod
--local-submission-csv /tmp/submission_0.csv
--local-checkpoint-file /tmp/checkpoint/
> /tmp/keras_spark_horovod_rossmann_estimator_local. log 2>&1
```

经过 10 个训练周期后，最终的运行时间如下：

```
real43m34.608s
user12m22.057s
sys2m30.127s
```

处理输入数据、训练 DNN 模型、计算准确度以及生成 TensorFlow 检查点和预测结果的 CSV 文件花费了超过 43 分钟。我们将训练周期数限制为 10，在实践中通常设置为 100，以确保令人满意的模型准确率。训练时间通常与训练次数呈线性关系。

接下来，我们使用集群中可用的四个工作节点，并在 `yarn`HDFS 中的数据模式：

```
(base) [root@n138 horovod]# time spark-submit
--master yarn
--executor-memory 5g
--executor-cores 1 --num-executors 160
/sparkusecase/horovod/keras_spark_horovod_rossmann_estimator.py
--epochs 10
--data-dir hdfs:///user/hdfs/tr-4570/experiments/horovod
--local-submission-csv /tmp/submission_1.csv
--local-checkpoint-file /tmp/checkpoint/
> /tmp/keras_spark_horovod_rossmann_estimator_yarn.log 2>&1
```

最终的运行时间改进如下：

```
real8m13.728s
user7m48.421s
sys1m26.063s
```

借助 Horovod 模型和 Spark 中的数据并行性，我们看到运行速度提高了 5.29 倍 `yarn`相对 `local`具有十个训练阶段的模式。下图中图例显示了这一点 `HDFS`和 `Local`。如果可用的话，可以使用 GPU 进一步加速

底层 TensorFlow DNN 模型训练。我们计划进行此项测试并在未来的技术报告中发布结果。

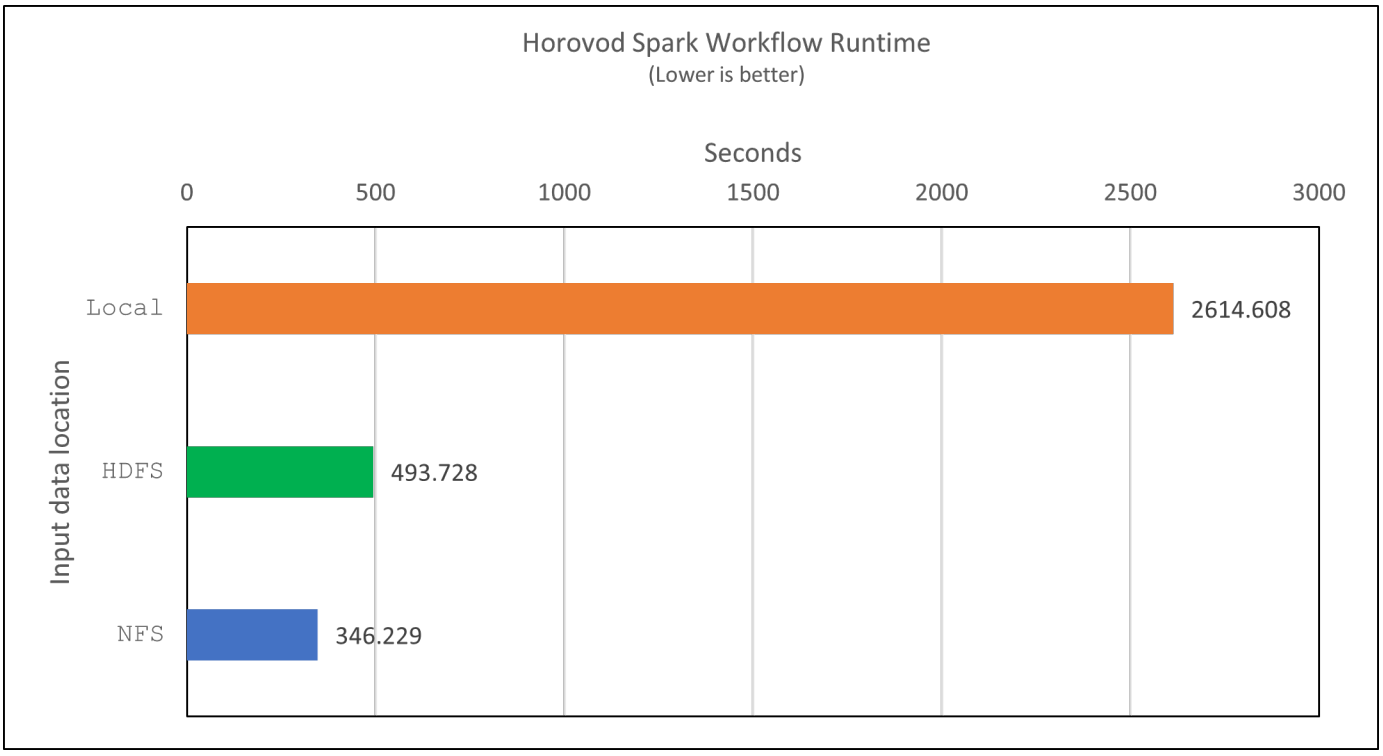
我们的下一个测试比较了 NFS 和 HDFS 中的输入数据的运行时间。 AFF A800上的 NFS 卷已安装在 `/sparkdemo/horovod`分布于 Spark 集群的五个节点（一个主节点，四个工作节点）上。我们运行了与之前的测试类似的命令， `--data- dir`参数现在指向 NFS 挂载：

```
(base) [root@n138 horovod]# time spark-submit
--master yarn
--executor-memory 5g
--executor-cores 1
--num-executors 160
/sparkusecase/horovod/keras_spark_horovod_rossmann_estimator.py
--epochs 10
--data-dir file:///sparkdemo/horovod
--local-submission-csv /tmp/submission_2.csv
--local-checkpoint-file /tmp/checkpoint/
> /tmp/keras_spark_horovod_rossmann_estimator_nfs.log 2>&1
```

使用 NFS 的运行结果如下：

```
real 5m46.229s
user 5m35.693s
sys  1m5.615s
```

速度又提高了 1.43 倍，如下图所示。因此，通过将NetApp全闪存存储连接到其集群，客户可以享受 Horovod Spark 工作流的快速数据传输和分发优势，与在单个节点上运行相比，可实现 7.55 倍的加速。



CTR预测性能的深度学习模型

对于旨在最大化点击率的推荐系统，必须学习用户行为背后复杂的特征交互，这些特征交互可以通过数学方式从低阶到高阶计算。对于良好的深度学习模型来说，低阶和高阶特征交叉应该同等重要，而不应偏向其中任何一方。深度分解机（DeepFM）是一种基于分解机的神经网络，它将用于推荐的分解机和用于特征学习的深度学习结合在一种新的神经网络架构中。

虽然传统的分解机将成对的特征交叉建模为特征之间潜在向量的内积，并且理论上可以捕获高阶信息，但在实践中，机器学习从业者通常只使用二阶特征交叉，因为计算和存储复杂度很高。深度神经网络变体，例如谷歌的 "广度与深度模型"另一方面，通过结合线性宽模型和深度模型，在混合网络结构中学习复杂的特征交互。

这个 Wide & Deep 模型有两个输入，一个用于底层的广度模型，另一个用于深度模型，后者仍然需要专家的特征工程，因此该技术不太适用于其他领域。与广度和深度模型不同，DeepFM 可以使用原始特征进行有效训练，而无需任何特征工程，因为它的广度部分和深度部分共享相同的输入和嵌入向量。

我们首先处理了 Criteo `train.txt` （11GB）文件转换为名为 `ctr_train.csv`存储在 NFS 挂载中 `/sparkdemo/tr-4570-data`使用 `run_classification_criteo_spark.py`来自部分"每个主要用例的 Python 脚本。"在此脚本中，函数 `process_input_file`执行几个字符串方法来删除制表符并插入 ’,‘`作为分隔符和 ’\n‘`作为换行符。请注意，您只需处理原始 `train.txt`一次，这样代码块就显示为注释。

为了对不同的 DL 模型进行以下测试，我们使用 `ctr_train.csv`作为输入文件。在后续的测试运行中，输入的 `CSV` 文件被读入 `Spark DataFrame`，其模式包含以下字段 ’`label'`，整数密集特征 `['I1', 'I2', 'I3', …, 'I13']`和稀疏特征 `['C1', 'C2', 'C3', …, 'C26']`。下列 `spark-submit`命令接受输入 CSV，以 20% 的比例训练 DeepFM 模型进行交叉验证，并在十个训练周期后选出最佳模型来计算测试集上的预测准确率：

```
(base) [root@n138 ~]# time spark-submit --master yarn --executor-memory 5g
--executor-cores 1 --num-executors 160
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py --data
-dir file:///sparkdemo/tr-4570-data >
/tmp/run_classification_criteo_spark_local.log 2>&1
```

请注意，由于数据文件 `ctr_train.csv`超过 11GB，则必须设置足够的 `spark.driver.maxResultSize`大于数据集大小以避免错误。

```
 spark = SparkSession.builder \
    .master("yarn") \
    .appName("deep_ctr_classification") \
    .config("spark.jars.packages", "io.github.ravwojdyla:spark-schema-
utils_2.12:0.1.0") \
    .config("spark.executor.cores", "1") \
    .config('spark.executor.memory', '5gb') \
    .config('spark.executor.memoryOverhead', '1500') \
    .config('spark.driver.memoryOverhead', '1500') \
    .config("spark.sql.shuffle.partitions", "480") \
    .config("spark.sql.execution.arrow.enabled", "true") \
    .config("spark.driver.maxResultSize", "50gb") \
    .getOrCreate()
```

在上述 `SparkSession.builder`配置我们还启用了 "阿帕奇箭"，将 Spark DataFrame 转换为 Pandas DataFrame ， `df.toPandas()`方法。

```
22/06/17 15:56:21 INFO scheduler.DAGScheduler: Job 2 finished: toPandas at
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py:96, took
627.126487 s
Obtained Spark DF and transformed to Pandas DF using Arrow.
```

随机分割后，训练数据集中有超过 3600 万行，测试集中有 900 万个样本：

```
Training dataset size =  36672493
Testing dataset size =  9168124
```

由于本技术报告专注于不使用任何 GPU 的 CPU 测试，因此必须使用适当的编译器标志构建 TensorFlow。此步骤避免调用任何 GPU 加速库，并充分利用 TensorFlow 的高级矢量扩展 (AVX) 和 AVX2 指令。这些特征是为线性代数计算而设计的，例如矢量加法、前馈中的矩阵乘法或反向传播 DNN 训练。 AVX2 提供的融合乘加 (FMA) 指令使用 256 位浮点 (FP) 寄存器，非常适合整数代码和数据类型，可实现高达 2 倍的加速。对于 FP 代码和数据类型，AVX2 比 AVX 实现了 8% 的加速。

```
2022-06-18 07:19:20.101478: I
tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow binary
is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the
following CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the
appropriate compiler flags.
```

要从源代码构建 TensorFlow， NetApp建议使用 "巴泽尔"。对于我们的环境，我们在 shell 提示符下执行以下命令来安装 `dnf`， `dnf-plugins` ，以及 Bazel。

```
yum install dnf
dnf install 'dnf-command(copr)'
dnf copr enable vbatts/bazel
dnf install bazel5
```

您必须启用 GCC 5 或更新版本才能在构建过程中使用 C++17 功能，该功能由 RHEL 通过软件集合库 (SCL) 提供。以下命令安装 `devtoolset`以及 RHEL 7.9 集群上的 GCC 11.2.1：

```
subscription-manager repos --enable rhel-server-rhscl-7-rpms
yum install devtoolset-11-toolchain
yum install devtoolset-11-gcc-c++
yum update
scl enable devtoolset-11 bash
. /opt/rh/devtoolset-11/enable
```

请注意，最后两个命令启用 `devtoolset-11`，使用 `/opt/rh/devtoolset-11/root/usr/bin/gcc`（GCC 11.2.1）。此外，请确保您的 `git`版本高于 1.8.3（随 RHEL 7.9 提供）。参考这个 "文章"用于更新 `git`至 2.24.1。

我们假设您已经克隆了最新的 TensorFlow 主仓库。然后创建一个 `workspace`目录与 `WORKSPACE`文件使用 AVX、AVX2 和 FMA 从源代码构建 TensorFlow。运行 `configure`文件并指定正确的 Python 二进制位置。"CUDA"由于我们没有使用 GPU，因此在我们的测试中被禁用。一个 `.bazelrc`文件根据您的设置生成。此外，我们编辑了文件并设置 `build --define=no_hdfs_support=false`启用 HDFS 支持。参考 `.bazelrc`在本节中"每个主要用例的 Python 脚本，"以获得完整的设置和标志列表。

```
./configure
bazel build -c opt --copt=-mavx --copt=-mavx2 --copt=-mfma --copt=
-mfpmath=both -k //tensorflow/tools/pip_package:build_pip_package
```

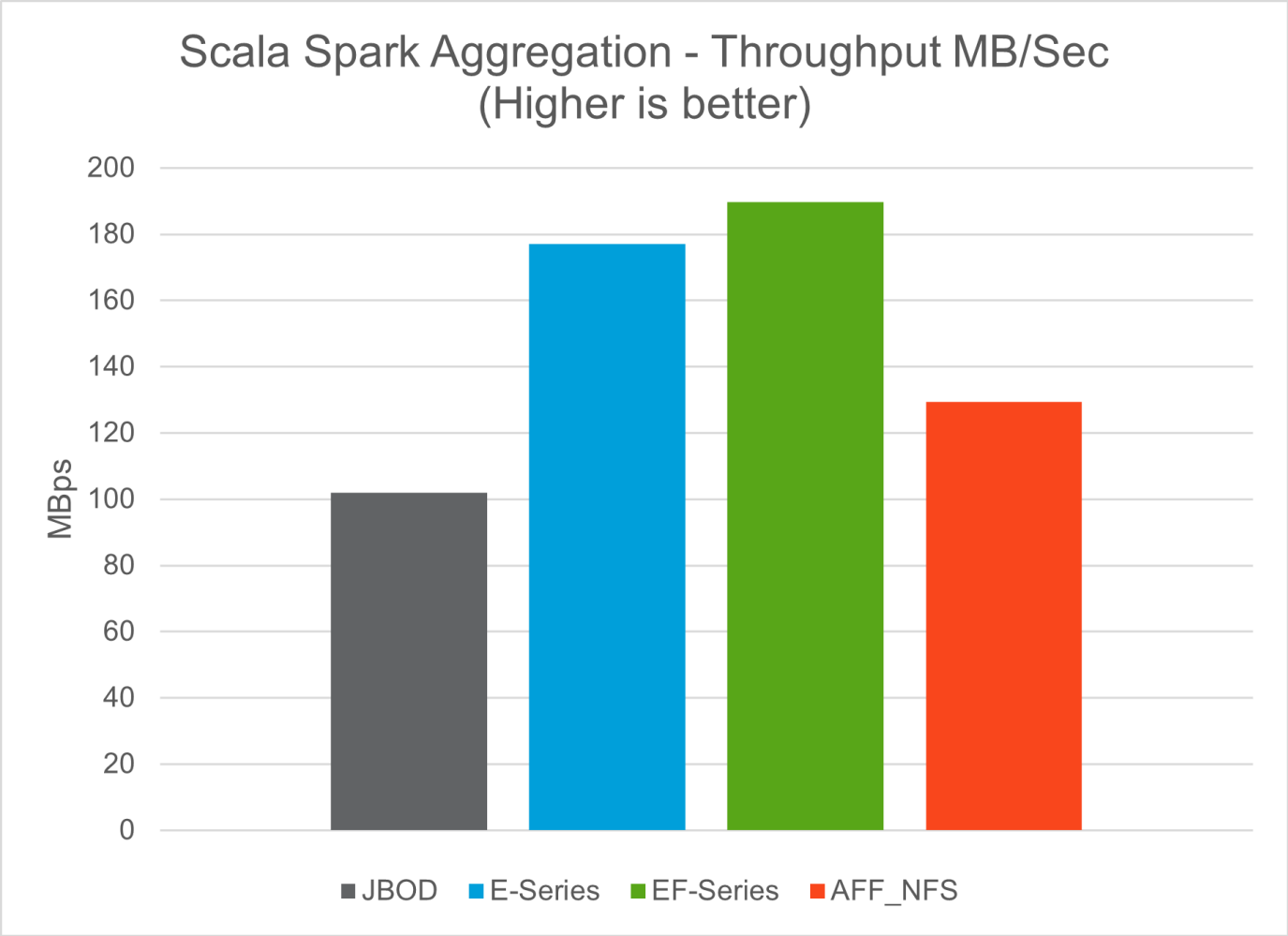
使用正确的标志构建 TensorFlow 后，运行以下脚本来处理 Criteo Display Ads 数据集，训练 DeepFM 模型，并根据预测分数计算接收者操作特征曲线下面积 (ROC AUC)。

```
(base) [root@n138 examples]# ~/anaconda3/bin/spark-submit
--master yarn
--executor-memory 15g
--executor-cores 1
--num-executors 160
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py
--data-dir file:///sparkdemo/tr-4570-data
> . /run_classification_criteo_spark_nfs.log 2>&1
```

经过十次训练后，我们获得了测试数据集上的 AUC 分数：

```
Epoch 1/10
125/125 - 7s - loss: 0.4976 - binary_crossentropy: 0.4974 - val_loss:
0.4629 - val_binary_crossentropy: 0.4624
Epoch 2/10
125/125 - 1s - loss: 0.3281 - binary_crossentropy: 0.3271 - val_loss:
0.5146 - val_binary_crossentropy: 0.5130
Epoch 3/10
125/125 - 1s - loss: 0.1948 - binary_crossentropy: 0.1928 - val_loss:
0.6166 - val_binary_crossentropy: 0.6144
Epoch 4/10
125/125 - 1s - loss: 0.1408 - binary_crossentropy: 0.1383 - val_loss:
0.7261 - val_binary_crossentropy: 0.7235
Epoch 5/10
125/125 - 1s - loss: 0.1129 - binary_crossentropy: 0.1102 - val_loss:
0.7961 - val_binary_crossentropy: 0.7934
Epoch 6/10
125/125 - 1s - loss: 0.0949 - binary_crossentropy: 0.0921 - val_loss:
0.9502 - val_binary_crossentropy: 0.9474
Epoch 7/10
125/125 - 1s - loss: 0.0778 - binary_crossentropy: 0.0750 - val_loss:
1.1329 - val_binary_crossentropy: 1.1301
Epoch 8/10
125/125 - 1s - loss: 0.0651 - binary_crossentropy: 0.0622 - val_loss:
1.3794 - val_binary_crossentropy: 1.3766
Epoch 9/10
125/125 - 1s - loss: 0.0555 - binary_crossentropy: 0.0527 - val_loss:
1.6115 - val_binary_crossentropy: 1.6087
Epoch 10/10
125/125 - 1s - loss: 0.0470 - binary_crossentropy: 0.0442 - val_loss:
1.6768 - val_binary_crossentropy: 1.6740
test AUC 0.6337
```

以与以前的用例类似的方式，我们将 Spark 工作流运行时与位于不同位置的数据进行了比较。下图显示了 Spark 工作流运行时深度学习 CTR 预测的比较。

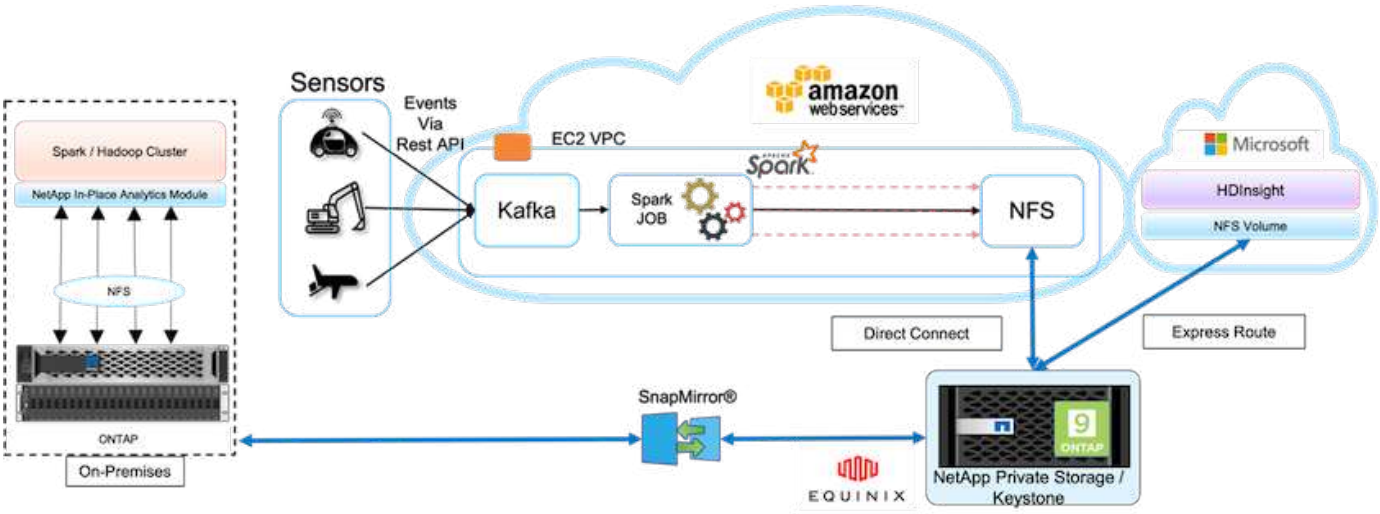


混合云解决方案

现代企业数据中心是一种混合云，它通过具有一致操作模式的连续数据管理平面在本地和/或多个公共云中连接多个分布式基础设施环境。为了充分利用混合云，您必须能够在本地和多云环境之间无缝移动数据，而无需进行任何数据转换或应用程序重构。

客户表示，他们开始混合云之旅的方法是将二级存储迁移到云端用于数据保护等用例，或者将不太重要的业务工作负载（如应用程序开发和 DevOps）迁移到云端。然后他们转向处理更重要的工作。 Web 和内容托管、DevOps 和应用程序开发、数据库、分析和容器化应用程序是最受欢迎的混合云工作负载。企业 AI 项目的复杂性、成本和风险历来阻碍 AI 从实验阶段走向生产阶段。

借助NetApp混合云解决方案，客户可以通过单一控制面板享受集成的安全性、数据治理和合规性工具，用于跨分布式环境的数据和工作流管理，同时根据其消费情况优化总体拥有成本。下图是一个云服务合作伙伴的示例解决方案，该合作伙伴负责为客户的大数据分析数据提供多云连接。



在这种情况下，AWS 从不同来源接收的 IoT 数据存储在NetApp私有存储 (NPS) 的中心位置。 NPS 存储连接到位于 AWS 和 Azure 中的 Spark 或 Hadoop 集群，使在多个云中运行的大数据分析应用程序能够访问相同的数据。此用例的主要要求和挑战包括：

- 客户希望使用多个云对相同数据运行分析作业。
- 必须通过不同的传感器和集线器从不同来源（例如本地和云环境）接收数据。
- 该解决方案必须高效且具有成本效益。
- 主要挑战是构建一个经济高效的解决方案，在不同的内部部署和云环境之间提供混合分析服务。

我们的数据保护和多云连接解决方案解决了跨多个超大规模计算平台的云分析应用程序的痛点。如上图所示，来自传感器的数据通过 Kafka 流式传输并输入到 AWS Spark 集群中。数据存储在 NPS 中的 NFS 共享中，NPS 位于 Equinix 数据中心内的云提供商之外。

由于NetApp NPS 分别通过 Direct Connect 和 Express Route 连接连接到 Amazon AWS 和 Microsoft Azure，因此客户可以利用 In-Place Analytics Module 访问来自 Amazon 和 AWS 分析集群的数据。因此，由于本地存储和 NPS 存储都运行ONTAP软件， "SnapMirror"可以将NPS数据镜像到本地集群，提供跨本地和多云的混合云分析。

为了获得最佳性能， NetApp通常建议使用多个网络接口和直接连接或快速路由来访问云实例的数据。我们还有其他数据移动解决方案，包括 "XCP"和 "BlueXP复制和同步"帮助客户构建应用感知、安全且经济高效的混合云 Spark 集群。

针对每个主要用例的 Python 脚本

以下三个 Python 脚本对应测试的三个主要用例。首先是 `sentiment_analysis_sparknlp.py`。

```
# TR-4570 Refresh NLP testing by Rick Huang
from sys import argv
import os
import sparknlp
import pyspark.sql.functions as F
from sparknlp import Finisher
```

```
from pyspark.ml import Pipeline
from sparknlp.base import *
from sparknlp.annotator import *
from sparknlp.pretrained import PretrainedPipeline
from sparknlp import Finisher
# Start Spark Session with Spark NLP
spark = sparknlp.start()
print("Spark NLP version:")
print(sparknlp.version())
print("Apache Spark version:")
print(spark.version)
spark = sparknlp.SparkSession.builder \
     .master("yarn") \
     .appName("test_hdfs_read_write") \
     .config("spark.executor.cores", "1") \
     .config("spark.jars.packages", "com.johnsnowlabs.nlp:spark-
nlp_2.12:3.4.3")\
     .config('spark.executor.memory', '5gb') \
     .config('spark.executor.memoryOverhead','1000')\
     .config('spark.driver.memoryOverhead','1000')\
     .config("spark.sql.shuffle.partitions", "480")\
     .getOrCreate()
sc = spark.sparkContext
from pyspark.sql import SQLContext
sql = SQLContext(sc)
sqlContext = SQLContext(sc)
# Download pre-trained pipelines & sequence classifier
explain_pipeline_model = PretrainedPipeline('explain_document_dl',
lang='en').model#pipeline_sa =
PretrainedPipeline("classifierdl_bertwiki_finance_sentiment_pipeline",
lang="en")
# pipeline_finbert =
BertForSequenceClassification.loadSavedModel('/sparkusecase/bert_sequence_
classifier_finbert_en_3', spark)
sequenceClassifier = BertForSequenceClassification \
        .pretrained('bert_sequence_classifier_finbert', 'en') \
        .setInputCols(['token', 'document']) \
        .setOutputCol('class') \
        .setCaseSensitive(True) \
        .setMaxSentenceLength(512)
def process_sentence_df(data):
    # Pre-process: begin
    print("1. Begin DataFrame pre-processing...\n")
    print(f"\n\t2. Attaching DocumentAssembler Transformer to the
pipeline")
    documentAssembler = DocumentAssembler() \
```

```
        .setInputCol("text") \
        .setOutputCol("document") \
        .setCleanupMode("inplace_full")
        #.setCleanupMode("shrink", "inplace_full")
    doc_df = documentAssembler.transform(data)
    doc_df.printSchema()
    doc_df.show(truncate=50)
    # Pre-process: get rid of  blank lines
    clean_df = doc_df.withColumn("tmp", F.explode("document")) \
        .select("tmp.result").where("tmp.end !=
-1").withColumnRenamed("result", "text").dropna()
    print("[OK!] DataFrame after initial cleanup:\n")
    clean_df.printSchema()
    clean_df.show(truncate=80)
    # for FinBERT
    tokenizer = Tokenizer() \
        .setInputCols(['document']) \
        .setOutputCol('token')
    print(f"\n\t3. Attaching Tokenizer Annotator to the pipeline")
    pipeline_finbert = Pipeline(stages=[
        documentAssembler,
        tokenizer,
        sequenceClassifier
        ])
    # Use Finisher() & construct PySpark ML pipeline
    finisher = Finisher().setInputCols(["token", "lemma", "pos",
"entities"])
    print(f"\n\t4. Attaching Finisher Transformer to the pipeline")
    pipeline_ex = Pipeline() \
        .setStages([
           explain_pipeline_model,
           finisher
           ])
    print("\n\t\t\t ---- Pipeline Built Successfully ----")
    # Loading pipelines to annotate
    #result_ex_df = pipeline_ex.transform(clean_df)
    ex_model = pipeline_ex.fit(clean_df)
    annotations_finished_ex_df = ex_model.transform(clean_df)
    # result_sa_df = pipeline_sa.transform(clean_df)
    result_finbert_df = pipeline_finbert.fit(clean_df).transform(clean_df)
    print("\n\t\t\t ----Document Explain, Sentiment Analysis & FinBERT
Pipeline Fitted Successfully ----")
    # Check the result entities
    print("[OK!] Simple explain ML pipeline result:\n")
    annotations_finished_ex_df.printSchema()
    annotations_finished_ex_df.select('text',
```

```
'finished_entities').show(truncate=False)
    # Check the result sentiment from FinBERT
    print("[OK!] Sentiment Analysis FinBERT pipeline result:\n")
    result_finbert_df.printSchema()
    result_finbert_df.select('text', 'class.result').show(80, False)
    sentiment_stats(result_finbert_df)
    return
def sentiment_stats(finbert_df):
    result_df = finbert_df.select('text', 'class.result')
    sa_df = result_df.select('result')
    sa_df.groupBy('result').count().show()
    # total_lines = result_clean_df.count()
    # num_neutral = result_clean_df.where(result_clean_df.result ==
['neutral']).count()
    # num_positive = result_clean_df.where(result_clean_df.result ==
['positive']).count()
    # num_negative = result_clean_df.where(result_clean_df.result ==
['negative']).count()
    # print(f"\nRatio of neutral sentiment = {num_neutral/total_lines}")
    # print(f"Ratio of positive sentiment = {num_positive / total_lines}")
    # print(f"Ratio of negative sentiment = {num_negative /
total_lines}\n")
    return
def process_input_file(file_name):
    # Turn input file to Spark DataFrame
    print("START processing input file...")
    data_df = spark.read.text(file_name)
    data_df.show()
    # rename first column 'text' for sparknlp
    output_df = data_df.withColumnRenamed("value", "text").dropna()
    output_df.printSchema()
    return output_dfdef process_local_dir(directory):
    filelist = []
    for subdir, dirs, files in os.walk(directory):
        for filename in files:
            filepath = subdir + os.sep + filename
            print("[OK!] Will process the following files:")
            if filepath.endswith(".txt"):
                print(filepath)
                filelist.append(filepath)
    return filelist
def process_local_dir_or_file(dir_or_file):
    numfiles = 0
    if os.path.isfile(dir_or_file):
        input_df = process_input_file(dir_or_file)
        print("Obtained input_df.")
```

```
        process_sentence_df(input_df)
        print("Processed input_df")
        numfiles += 1
    else:
        filelist = process_local_dir(dir_or_file)
        for file in filelist:
            input_df = process_input_file(file)
            process_sentence_df(input_df)
            numfiles += 1
    return numfiles
def process_hdfs_dir(dir_name):
    # Turn input files to Spark DataFrame
    print("START processing input HDFS directory...")
    data_df = spark.read.option("recursiveFileLookup",
"true").text(dir_name)
    data_df.show()
    print("[DEBUG] total lines in data_df = ", data_df.count())
    # rename first column 'text' for sparknlp
    output_df = data_df.withColumnRenamed("value", "text").dropna()
    print("[DEBUG] output_df looks like: \n")
    output_df.show(40, False)
    print("[DEBUG] HDFS dir resulting data_df schema: \n")
    output_df.printSchema()
    process_sentence_df(output_df)
    print("Processed HDFS directory: ", dir_name)
    returnif __name__ == '__main__':
    try:
        if len(argv) == 2:
            print("Start processing input...\n")
    except:
        print("[ERROR] Please enter input text file or path to
process!\n")
        exit(1)
    # This is for local file, not hdfs:
    numfiles = process_local_dir_or_file(str(argv[1]))
    # For HDFS single file & directory:
    input_df = process_input_file(str(argv[1]))
    print("Obtained input_df.")
    process_sentence_df(input_df)
    print("Processed input_df")
    numfiles += 1
    # For HDFS directory of subdirectories of files:
    input_parse_list = str(argv[1]).split('/')
    print(input_parse_list)
    if input_parse_list[-2:-1] == ['Transcripts']:
        print("Start processing HDFS directory: ", str(argv[1]))
```

```
        process_hdfs_dir(str(argv[1]))
    print(f"[OK!] All done. Number of files processed = {numfiles}")
```

第二个脚本是 keras_spark_horovod_rossmann_estimator.py。

```
# Copyright 2022 NetApp, Inc.
# Authored by Rick Huang
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
==========================================================================
====
# The below code was modified from: https://www.kaggle.com/c/rossmann-
store-sales
import argparse
import datetime
import os
import sys
from distutils.version import LooseVersion
import pyspark.sql.types as T
import pyspark.sql.functions as F
from pyspark import SparkConf, Row
from pyspark.sql import SparkSession
import tensorflow as tf
import tensorflow.keras.backend as K
from tensorflow.keras.layers import Input, Embedding, Concatenate, Dense,
Flatten, Reshape, BatchNormalization, Dropout
import horovod.spark.keras as hvd
from horovod.spark.common.backend import SparkBackend
from horovod.spark.common.store import Store
from horovod.tensorflow.keras.callbacks import BestModelCheckpoint
parser = argparse.ArgumentParser(description='Horovod Keras Spark Rossmann
Estimator Example',

formatter_class=argparse.ArgumentDefaultsHelpFormatter)
```

```
parser.add_argument('--master',
                    help='spark cluster to use for training. If set to
None, uses current default cluster. Cluster'
                         'should be set up to provide a Spark task per
multiple CPU cores, or per GPU, e.g. by'
                         'supplying `-c <NUM_GPUS>` in Spark Standalone
mode')
parser.add_argument('--num-proc', type=int,
                    help='number of worker processes for training,
default: `spark.default.parallelism`')
parser.add_argument('--learning_rate', type=float, default=0.0001,
                    help='initial learning rate')
parser.add_argument('--batch-size', type=int, default=100,
                    help='batch size')
parser.add_argument('--epochs', type=int, default=100,
                    help='number of epochs to train')
parser.add_argument('--sample-rate', type=float,
                    help='desired sampling rate. Useful to set to low
number (e.g. 0.01) to make sure that '
                         'end-to-end process works')
parser.add_argument('--data-dir', default='file://' + os.getcwd(),
                    help='location of data on local filesystem (prefixed
with file://) or on HDFS')
parser.add_argument('--local-submission-csv', default='submission.csv',
                    help='output submission predictions CSV')
parser.add_argument('--local-checkpoint-file', default='checkpoint',
                    help='model checkpoint')
parser.add_argument('--work-dir', default='/tmp',
                    help='temporary working directory to write
intermediate files (prefix with hdfs:// to use HDFS)')
if __name__ == '__main__':
    args = parser.parse_args()
    # ================ #
    # DATA PREPARATION #
    # ================ #
    print('================')
    print('Data preparation')
    print('================')
    # Create Spark session for data preparation.
    conf = SparkConf() \
        .setAppName('Keras Spark Rossmann Estimator Example') \
        .set('spark.sql.shuffle.partitions', '480') \
        .set("spark.executor.cores", "1") \
        .set('spark.executor.memory', '5gb') \
        .set('spark.executor.memoryOverhead','1000')\
        .set('spark.driver.memoryOverhead','1000')
```

```
    if args.master:
        conf.setMaster(args.master)
    elif args.num_proc:
        conf.setMaster('local[{}]'.format(args.num_proc))
    spark = SparkSession.builder.config(conf=conf).getOrCreate()
    train_csv = spark.read.csv('%s/train.csv' % args.data_dir,
header=True)
    test_csv = spark.read.csv('%s/test.csv' % args.data_dir, header=True)
    store_csv = spark.read.csv('%s/store.csv' % args.data_dir,
header=True)
    store_states_csv = spark.read.csv('%s/store_states.csv' %
args.data_dir, header=True)
    state_names_csv = spark.read.csv('%s/state_names.csv' % args.data_dir,
header=True)
    google_trend_csv = spark.read.csv('%s/googletrend.csv' %
args.data_dir, header=True)
    weather_csv = spark.read.csv('%s/weather.csv' % args.data_dir,
header=True)
    def expand_date(df):
        df = df.withColumn('Date', df.Date.cast(T.DateType()))
        return df \
            .withColumn('Year', F.year(df.Date)) \
            .withColumn('Month', F.month(df.Date)) \
            .withColumn('Week', F.weekofyear(df.Date)) \
            .withColumn('Day', F.dayofmonth(df.Date))
    def prepare_google_trend():
        # Extract week start date and state.
        google_trend_all = google_trend_csv \
            .withColumn('Date', F.regexp_extract(google_trend_csv.week,
'(.*?) -', 1)) \
            .withColumn('State', F.regexp_extract(google_trend_csv.file,
'Rossmann_DE_(.*)', 1))
        # Map state NI -> HB,NI to align with other data sources.
        google_trend_all = google_trend_all \
            .withColumn('State', F.when(google_trend_all.State == 'NI',
'HB,NI').otherwise(google_trend_all.State))
        # Expand dates.
        return expand_date(google_trend_all)
    def add_elapsed(df, cols):
        def add_elapsed_column(col, asc):
            def fn(rows):
                last_store, last_date = None, None
                for r in rows:
                    if last_store != r.Store:
                        last_store = r.Store
                        last_date = r.Date
```

```
                    if r[col]:
                        last_date = r.Date
                    fields = r.asDict().copy()
                    fields[('After' if asc else 'Before') + col] = (r.Date
- last_date).days
                    yield Row(**fields)
            return fn
        df = df.repartition(df.Store)
        for asc in [False, True]:
            sort_col = df.Date.asc() if asc else df.Date.desc()
            rdd = df.sortWithinPartitions(df.Store.asc(), sort_col).rdd
            for col in cols:
                rdd = rdd.mapPartitions(add_elapsed_column(col, asc))
            df = rdd.toDF()
        return df
    def prepare_df(df):
        num_rows = df.count()
        # Expand dates.
        df = expand_date(df)
        df = df \
            .withColumn('Open', df.Open != '0') \
            .withColumn('Promo', df.Promo != '0') \
            .withColumn('StateHoliday', df.StateHoliday != '0') \
            .withColumn('SchoolHoliday', df.SchoolHoliday != '0')
        # Merge in store information.
        store = store_csv.join(store_states_csv, 'Store')
        df = df.join(store, 'Store')
        # Merge in Google Trend information.
        google_trend_all = prepare_google_trend()
        df = df.join(google_trend_all, ['State', 'Year',
'Week']).select(df['*'], google_trend_all.trend)
        # Merge in Google Trend for whole Germany.
        google_trend_de = google_trend_all[google_trend_all.file ==
'Rossmann_DE'].withColumnRenamed('trend', 'trend_de')
        df = df.join(google_trend_de, ['Year', 'Week']).select(df['*'],
google_trend_de.trend_de)
        # Merge in weather.
        weather = weather_csv.join(state_names_csv, weather_csv.file ==
state_names_csv.StateName)
        df = df.join(weather, ['State', 'Date'])
        # Fix null values.
        df = df \
            .withColumn('CompetitionOpenSinceYear',
F.coalesce(df.CompetitionOpenSinceYear, F.lit(1900))) \
            .withColumn('CompetitionOpenSinceMonth',
F.coalesce(df.CompetitionOpenSinceMonth, F.lit(1))) \
```

```
            .withColumn('Promo2SinceYear', F.coalesce(df.Promo2SinceYear,
F.lit(1900))) \
            .withColumn('Promo2SinceWeek', F.coalesce(df.Promo2SinceWeek,
F.lit(1)))
        # Days & months competition was open, cap to 2 years.
        df = df.withColumn('CompetitionOpenSince',
                           F.to_date(F.format_string('%s-%s-15',
df.CompetitionOpenSinceYear,

df.CompetitionOpenSinceMonth)))
        df = df.withColumn('CompetitionDaysOpen',
                           F.when(df.CompetitionOpenSinceYear > 1900,
                                  F.greatest(F.lit(0), F.least(F.lit(360 *
2), F.datediff(df.Date, df.CompetitionOpenSince))))
                           .otherwise(0))
        df = df.withColumn('CompetitionMonthsOpen',
(df.CompetitionDaysOpen / 30).cast(T.IntegerType()))
        # Days & weeks of promotion, cap to 25 weeks.
        df = df.withColumn('Promo2Since',
                           F.expr('date_add(format_string("%s-01-01",
Promo2SinceYear), (cast(Promo2SinceWeek as int) - 1) * 7)'))
        df = df.withColumn('Promo2Days',
                           F.when(df.Promo2SinceYear > 1900,
                                  F.greatest(F.lit(0), F.least(F.lit(25 *
7), F.datediff(df.Date, df.Promo2Since))))
                           .otherwise(0))
        df = df.withColumn('Promo2Weeks', (df.Promo2Days /
7).cast(T.IntegerType()))
        # Check that we did not lose any rows through inner joins.
        assert num_rows == df.count(), 'lost rows in joins'
        return df
    def build_vocabulary(df, cols):
        vocab = {}
        for col in cols:
            values = [r[0] for r in df.select(col).distinct().collect()]
            col_type = type([x for x in values if x is not None][0])
            default_value = col_type()
            vocab[col] = sorted(values, key=lambda x: x or default_value)
        return vocab
    def cast_columns(df, cols):
        for col in cols:
            df = df.withColumn(col,
F.coalesce(df[col].cast(T.FloatType()), F.lit(0.0)))
        return df
    def lookup_columns(df, vocab):
        def lookup(mapping):
```

```
            def fn(v):
                return mapping.index(v)
            return F.udf(fn, returnType=T.IntegerType())
        for col, mapping in vocab.items():
            df = df.withColumn(col, lookup(mapping)(df[col]))
        return df
    if args.sample_rate:
        train_csv = train_csv.sample(withReplacement=False,
fraction=args.sample_rate)
        test_csv = test_csv.sample(withReplacement=False,
fraction=args.sample_rate)
    # Prepare data frames from CSV files.
    train_df = prepare_df(train_csv).cache()
    test_df = prepare_df(test_csv).cache()
    # Add elapsed times from holidays & promos, the data spanning training
& test datasets.
    elapsed_cols = ['Promo', 'StateHoliday', 'SchoolHoliday']
    elapsed = add_elapsed(train_df.select('Date', 'Store', *elapsed_cols)
                          .unionAll(test_df.select('Date', 'Store',
*elapsed_cols)),
                          elapsed_cols)
    # Join with elapsed times.
    train_df = train_df \
        .join(elapsed, ['Date', 'Store']) \
        .select(train_df['*'], *[prefix + col for prefix in ['Before',
'After'] for col in elapsed_cols])
    test_df = test_df \
        .join(elapsed, ['Date', 'Store']) \
        .select(test_df['*'], *[prefix + col for prefix in ['Before',
'After'] for col in elapsed_cols])
    # Filter out zero sales.
    train_df = train_df.filter(train_df.Sales > 0)
    print('====================')
    print('Prepared data frame')
    print('====================')
    train_df.show()
    categorical_cols = [
        'Store', 'State', 'DayOfWeek', 'Year', 'Month', 'Day', 'Week',
'CompetitionMonthsOpen', 'Promo2Weeks', 'StoreType',
        'Assortment', 'PromoInterval', 'CompetitionOpenSinceYear',
'Promo2SinceYear', 'Events', 'Promo',
        'StateHoliday', 'SchoolHoliday'
    ]
    continuous_cols = [
        'CompetitionDistance', 'Max_TemperatureC', 'Mean_TemperatureC',
'Min_TemperatureC', 'Max_Humidity',
```

```
        'Mean_Humidity', 'Min_Humidity', 'Max_Wind_SpeedKm_h',
'Mean_Wind_SpeedKm_h', 'CloudCover', 'trend', 'trend_de',
        'BeforePromo', 'AfterPromo', 'AfterStateHoliday',
'BeforeStateHoliday', 'BeforeSchoolHoliday', 'AfterSchoolHoliday'
    ]
    all_cols = categorical_cols + continuous_cols
    # Select features.
    train_df = train_df.select(*(all_cols + ['Sales', 'Date'])).cache()
    test_df = test_df.select(*(all_cols + ['Id', 'Date'])).cache()
    # Build vocabulary of categorical columns.
    vocab = build_vocabulary(train_df.select(*categorical_cols)

.unionAll(test_df.select(*categorical_cols)).cache(),
                               categorical_cols)
    # Cast continuous columns to float & lookup categorical columns.
    train_df = cast_columns(train_df, continuous_cols + ['Sales'])
    train_df = lookup_columns(train_df, vocab)
    test_df = cast_columns(test_df, continuous_cols)
    test_df = lookup_columns(test_df, vocab)
    # Split into training & validation.
    # Test set is in 2015, use the same period in 2014 from the training
set as a validation set.
    test_min_date = test_df.agg(F.min(test_df.Date)).collect()[0][0]
    test_max_date = test_df.agg(F.max(test_df.Date)).collect()[0][0]
    one_year = datetime.timedelta(365)
    train_df = train_df.withColumn('Validation',
                                   (train_df.Date > test_min_date -
one_year) & (train_df.Date <= test_max_date - one_year))
    # Determine max Sales number.
    max_sales = train_df.agg(F.max(train_df.Sales)).collect()[0][0]
    # Convert Sales to log domain
    train_df = train_df.withColumn('Sales', F.log(train_df.Sales))
    print('====================================')
    print('Data frame with transformed columns')
    print('====================================')
    train_df.show()
    print('================')
    print('Data frame sizes')
    print('================')
    train_rows = train_df.filter(~train_df.Validation).count()
    val_rows = train_df.filter(train_df.Validation).count()
    test_rows = test_df.count()
    print('Training: %d' % train_rows)
    print('Validation: %d' % val_rows)
    print('Test: %d' % test_rows)
    # ============== #
```

```
    # MODEL TRAINING #
    # ============== #
    print('==============')
    print('Model training')
    print('==============')
    def exp_rmspe(y_true, y_pred):
        """Competition evaluation metric, expects logarithic inputs."""
        pct = tf.square((tf.exp(y_true) - tf.exp(y_pred)) /
tf.exp(y_true))
        # Compute mean excluding stores with zero denominator.
        x = tf.reduce_sum(tf.where(y_true > 0.001, pct,
tf.zeros_like(pct)))
        y = tf.reduce_sum(tf.where(y_true > 0.001, tf.ones_like(pct),
tf.zeros_like(pct)))
        return tf.sqrt(x / y)
    def act_sigmoid_scaled(x):
        """Sigmoid scaled to logarithm of maximum sales scaled by 20%."""
        return tf.nn.sigmoid(x) * tf.math.log(max_sales) * 1.2
    CUSTOM_OBJECTS = {'exp_rmspe': exp_rmspe,
                      'act_sigmoid_scaled': act_sigmoid_scaled}
    # Disable GPUs when building the model to prevent memory leaks
    if LooseVersion(tf.__version__) >= LooseVersion('2.0.0'):
        # See https://github.com/tensorflow/tensorflow/issues/33168
        os.environ['CUDA_VISIBLE_DEVICES'] = '-1'
    else:

K.set_session(tf.Session(config=tf.ConfigProto(device_count={'GPU': 0})))
    # Build the model.
    inputs = {col: Input(shape=(1,), name=col) for col in all_cols}
    embeddings = [Embedding(len(vocab[col]), 10, input_length=1,
name='emb_' + col)(inputs[col])
                  for col in categorical_cols]
    continuous_bn = Concatenate()([Reshape((1, 1), name='reshape_' +
col)(inputs[col])
                                   for col in continuous_cols])
    continuous_bn = BatchNormalization()(continuous_bn)
    x = Concatenate()(embeddings + [continuous_bn])
    x = Flatten()(x)
    x = Dense(1000, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dense(1000, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dense(1000, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dense(500, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
```

```
    x = Dropout(0.5)(x)
    output = Dense(1, activation=act_sigmoid_scaled)(x)
    model = tf.keras.Model([inputs[f] for f in all_cols], output)
    model.summary()
    opt = tf.keras.optimizers.Adam(lr=args.learning_rate, epsilon=1e-3)
    # Checkpoint callback to specify options for the returned Keras model
    ckpt_callback = BestModelCheckpoint(monitor='val_loss', mode='auto',
save_freq='epoch')
    # Horovod: run training.
    store = Store.create(args.work_dir)
    backend = SparkBackend(num_proc=args.num_proc,
                           stdout=sys.stdout, stderr=sys.stderr,
                           prefix_output_with_timestamp=True)
    keras_estimator = hvd.KerasEstimator(backend=backend,
                                         store=store,
                                         model=model,
                                         optimizer=opt,
                                         loss='mae',
                                         metrics=[exp_rmspe],
                                         custom_objects=CUSTOM_OBJECTS,
                                         feature_cols=all_cols,
                                         label_cols=['Sales'],
                                         validation='Validation',
                                         batch_size=args.batch_size,
                                         epochs=args.epochs,
                                         verbose=2,

checkpoint_callback=ckpt_callback)
    keras_model =
keras_estimator.fit(train_df).setOutputCols(['Sales_output'])
    history = keras_model.getHistory()
    best_val_rmspe = min(history['val_exp_rmspe'])
    print('Best RMSPE: %f' % best_val_rmspe)
    # Save the trained model.
    keras_model.save(args.local_checkpoint_file)
    print('Written checkpoint to %s' % args.local_checkpoint_file)
    # ================ #
    # FINAL PREDICTION #
    # ================ #
    print('================')
    print('Final prediction')
    print('================')
    pred_df=keras_model.transform(test_df)
    pred_df.printSchema()
    pred_df.show(5)
    # Convert from log domain to real Sales numbers
```

```
    pred_df=pred_df.withColumn('Sales_pred', F.exp(pred_df.Sales_output))
    submission_df = pred_df.select(pred_df.Id.cast(T.IntegerType()),
pred_df.Sales_pred).toPandas()
    submission_df.sort_values(by=['Id']).to_csv(args.local_submission_csv,
index=False)
    print('Saved predictions to %s' % args.local_submission_csv)
    spark.stop()
```

第三个脚本是 `run_classification_criteo_spark.py`。

```
import tempfile, string, random, os, uuid
import argparse, datetime, sys, shutil
import csv
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
from pyspark import SparkContext
from pyspark.sql import SparkSession, SQLContext, Row, DataFrame
from pyspark.mllib import linalg as mllib_linalg
from pyspark.mllib.linalg import SparseVector as mllibSparseVector
from pyspark.mllib.linalg import VectorUDT as mllibVectorUDT
from pyspark.mllib.linalg import Vector as mllibVector, Vectors as
mllibVectors
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.classification import LogisticRegressionWithSGD
from pyspark.ml import linalg as ml_linalg
from pyspark.ml.linalg import VectorUDT as mlVectorUDT
from pyspark.ml.linalg import SparseVector as mlSparseVector
from pyspark.ml.linalg import Vector as mlVector, Vectors as mlVectors
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import OneHotEncoder
from math import log
from math import exp  # exp(-t) = e^-t
from operator import add
from pyspark.sql.functions import udf, split, lit
from pyspark.sql.functions import size, sum as sqlsum
import pyspark.sql.functions as F
import pyspark.sql.types as T
from pyspark.sql.types import ArrayType, StructType, StructField,
LongType, StringType, IntegerType, FloatType
from pyspark.sql.functions import explode, col, log, when
from collections import defaultdict
import pandas as pd
import pyspark.pandas as ps
from sklearn.metrics import log_loss, roc_auc_score
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from deepctr.models import DeepFM
from deepctr.feature_column import SparseFeat, DenseFeat,
get_feature_names
spark = SparkSession.builder \
    .master("yarn") \
    .appName("deep_ctr_classification") \
    .config("spark.jars.packages", "io.github.ravwojdyla:spark-schema-
utils_2.12:0.1.0") \
    .config("spark.executor.cores", "1") \
    .config('spark.executor.memory', '5gb') \
    .config('spark.executor.memoryOverhead', '1500') \
    .config('spark.driver.memoryOverhead', '1500') \
    .config("spark.sql.shuffle.partitions", "480") \
    .config("spark.sql.execution.arrow.enabled", "true") \
    .config("spark.driver.maxResultSize", "50gb") \
    .getOrCreate()
# spark.conf.set("spark.sql.execution.arrow.enabled", "true") # deprecated
print("Apache Spark version:")
print(spark.version)
sc = spark.sparkContext
sqlContext = SQLContext(sc)
parser = argparse.ArgumentParser(description='Spark DCN CTR Prediction
Example',

formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--data-dir', default='file://' + os.getcwd(),
                    help='location of data on local filesystem (prefixed
with file://) or on HDFS')
def process_input_file(file_name, sparse_feat, dense_feat):
    # Need this preprocessing to turn Criteo raw file into CSV:
    print("START processing input file...")
    # only convert the file ONCE
    # sample = open(file_name)
    # sample = '\n'.join([str(x.replace('\n', '').replace('\t', ',')) for
x in sample])
    # # Add header in data file and save as CSV
    # header = ','.join(str(x) for x in (['label'] + dense_feat +
sparse_feat))
    # with open('/sparkdemo/tr-4570-data/ctr_train.csv', mode='w',
encoding="utf-8") as f:
    #     f.write(header + '\n' + sample)
    #     f.close()
    # print("Raw training file processed and saved as CSV: ", f.name)
    raw_df = sqlContext.read.option("header", True).csv(file_name)
```

```
    raw_df.show(5, False)
    raw_df.printSchema()
    # convert columns I1 to I13 from string to integers
    conv_df = raw_df.select(col('label').cast("double"),
                            *(col(i).cast("float").alias(i) for i in
raw_df.columns if i in dense_feat),
                            *(col(c) for c in raw_df.columns if c in
sparse_feat))
    print("Schema of raw_df with integer columns type changed:")
    conv_df.printSchema()
    # result_pdf = conv_df.select("*").toPandas()
    tmp_df = conv_df.na.fill(0, dense_feat)
    result_df = tmp_df.na.fill('-1', sparse_feat)
    result_df.show()
    return result_df
if __name__ == "__main__":
    args = parser.parse_args()
    # Pandas read CSV
    # data = pd.read_csv('%s/criteo_sample.txt' % args.data_dir)
    # print("Obtained Pandas df.")
    dense_features = ['I' + str(i) for i in range(1, 14)]
    sparse_features = ['C' + str(i) for i in range(1, 27)]
    # Spark read CSV
    # process_input_file('%s/train.txt' % args.data_dir, sparse_features,
dense_features) # run only ONCE
    spark_df = process_input_file('%s/data.txt' % args.data_dir,
sparse_features, dense_features) # sample data
    # spark_df = process_input_file('%s/ctr_train.csv' % args.data_dir,
sparse_features, dense_features)
    print("Obtained Spark df and filled in missing features.")
    data = spark_df
    # Pandas
    #data[sparse_features] = data[sparse_features].fillna('-1', )
    #data[dense_features] = data[dense_features].fillna(0, )
    target = ['label']
    label_npa = data.select("label").toPandas().to_numpy()
    print("label numPy array has length = ", len(label_npa)) # 45,840,617
w/ 11GB dataset
    label_npa.ravel()
    label_npa.reshape(len(label_npa), )
    # 1.Label Encoding for sparse features,and do simple Transformation
for dense features
    print("Before LabelEncoder():")
    data.printSchema()  # label: float (nullable = true)
    for feat in sparse_features:
        lbe = LabelEncoder()
```

```
        tmp_pdf = data.select(feat).toPandas().to_numpy()
        tmp_ndarray = lbe.fit_transform(tmp_pdf)
        print("After LabelEncoder(), tmp_ndarray[0] =", tmp_ndarray[0])
        # print("Data tmp PDF after lbe transformation, the output ndarray
has length = ", len(tmp_ndarray)) # 45,840,617 for 11GB dataset
        tmp_ndarray.ravel()
        tmp_ndarray.reshape(len(tmp_ndarray), )
        out_ndarray = np.column_stack([label_npa, tmp_ndarray])
        pdf = pd.DataFrame(out_ndarray, columns=['label', feat])
        s_df = spark.createDataFrame(pdf)
        s_df.printSchema() # label: double (nullable = true)
        print("Before joining data df with s_df, s_df example rows:")
        s_df.show(1, False)
        data = data.drop(feat).join(s_df, 'label').drop('label')
        print("After LabelEncoder(), data df example rows:")
        data.show(1, False)
        print("Finished processing sparse_features: ", feat)
    print("Data DF after label encoding: ")
    data.show()
    data.printSchema()
    mms = MinMaxScaler(feature_range=(0, 1))
    # data[dense_features] = mms.fit_transform(data[dense_features]) # for
Pandas df
    tmp_pdf = data.select(dense_features).toPandas().to_numpy()
    tmp_ndarray = mms.fit_transform(tmp_pdf)
    tmp_ndarray.ravel()
    tmp_ndarray.reshape(len(tmp_ndarray), len(tmp_ndarray[0]))
    out_ndarray = np.column_stack([label_npa, tmp_ndarray])
    pdf = pd.DataFrame(out_ndarray, columns=['label'] + dense_features)
    s_df = spark.createDataFrame(pdf)
    s_df.printSchema()
    data.drop(*dense_features).join(s_df, 'label').drop('label')
    print("Finished processing dense_features: ", dense_features)
    print("Data DF after MinMaxScaler: ")
    data.show()

    # 2.count #unique features for each sparse field,and record dense
feature field name
    fixlen_feature_columns = [SparseFeat(feat,
vocabulary_size=data.select(feat).distinct().count() + 1, embedding_dim=4)
                              for i, feat in enumerate(sparse_features)] +
\
                             [DenseFeat(feat, 1, ) for feat in
dense_features]
    dnn_feature_columns = fixlen_feature_columns
    linear_feature_columns = fixlen_feature_columns
```

```
    feature_names = get_feature_names(linear_feature_columns +
dnn_feature_columns)
    # 3.generate input data for model
    # train, test = train_test_split(data.toPandas(), test_size=0.2,
random_state=2020) # Pandas; might hang for 11GB data
    train, test = data.randomSplit(weights=[0.8, 0.2], seed=200)
    print("Training dataset size = ", train.count())
    print("Testing dataset size = ", test.count())
    # Pandas:
    # train_model_input = {name: train[name] for name in feature_names}
    # test_model_input = {name: test[name] for name in feature_names}
    # Spark DF:
    train_model_input = {}
    test_model_input = {}
    for name in feature_names:
        if name.startswith('I'):
            tr_pdf = train.select(name).toPandas()
            train_model_input[name] = pd.to_numeric(tr_pdf[name])
            ts_pdf = test.select(name).toPandas()
            test_model_input[name] = pd.to_numeric(ts_pdf[name])
    # 4.Define Model,train,predict and evaluate
    model = DeepFM(linear_feature_columns, dnn_feature_columns,
task='binary')
    model.compile("adam", "binary_crossentropy",
                  metrics=['binary_crossentropy'], )
    lb_pdf = train.select(target).toPandas()
    history = model.fit(train_model_input,
pd.to_numeric(lb_pdf['label']).values,
                        batch_size=256, epochs=10, verbose=2,
validation_split=0.2, )
    pred_ans = model.predict(test_model_input, batch_size=256)
    print("test LogLoss",
round(log_loss(pd.to_numeric(test.select(target).toPandas()).values,
pred_ans), 4))
    print("test AUC",
round(roc_auc_score(pd.to_numeric(test.select(target).toPandas()).values,
pred_ans), 4))
```

结束语

在本文档中，我们讨论了 Apache Spark 架构、客户用例以及与大数据、现代分析、AI 、ML 和 DL 相关的NetApp存储产品组合。在我们基于行业标准基准测试工具和客户需求的性能验证测试中， NetApp Spark 解决方案表现出了相对于原生 Hadoop 系统的卓越性能。本报告中提供的客户用例和性能结果的组合可以帮助您为您的部署选择合适的 Spark 解决方案。

在哪里可以找到更多信息

本 TR 中使用了以下参考文献：

- Apache Spark 架构和组件

  "http://spark.apache.org/docs/latest/cluster-overview.html"

- Apache Spark 用例

  "https://www.qubole.com/blog/big-data/apache-spark-use-cases/"

- Spark NLP

  "https://www.johnsnowlabs.com/spark-nlp/"

- BERT

  "https://arxiv.org/abs/1810.04805"

- 用于广告点击预测的深度和交叉网络

  "https://arxiv.org/abs/1708.05123"

- FlexGroup

  https://www.netapp.com/pdf.html?item=/media/7337-tr4557pdf.pdf

- 简化 ETL

  "https://www.infoq.com/articles/apache-spark-streaming"

- NetApp E系列Hadoop解决方案

  "https://www.netapp.com/media/16420-tr-3969.pdf"

- NetApp现代数据分析解决方案

  "数据分析解决方案"

- SnapMirror

  "https://docs.netapp.com/us-en/ontap/data-protection/snapmirror-replication-concept.html"

- XCP

  https://mysupport.netapp.com/documentation/docweb/index.html?productID=63942&language=en-US

- BlueXP复制和同步

  "https://cloud.netapp.com/cloud-sync-service"

- DataOps 工具包

"https://github.com/NetApp/netapp-dataops-toolkit"