大数据分析技术期末实验报告 —— 基于 Spark 的分布式时间序列异常检测

摘要

本实验旨在解决大规模时间序列数据的异常检测性能瓶颈问题。在现有的 Python 异常检测算法库基础上，引入 Apache Spark 分布式计算框架，设计并实现了一套并行化检测流程。通过 Spark DataFrame API 对数据进行多维分组（基于 `cmdb_id` 与 `kpi_name`），并利用 PySpark 的 Pandas UDF 功能结合 Apache Arrow 内存加速技术，实现了统计类检测算法（如 Z-Score、Ensemble 等）的分布式并行执行。实验采用合成数据集进行验证，结果表明该方案能够准确识别异常点，并具备良好的水平扩展能力，为工业界海量运维数据的实时监控提供了可行的技术路径。

1. 实验背景与目标

随着系统规模的扩大，单机串行处理海量运维指标（KPI）已无法满足时效性要求。本实验的主要目标包括：

1. **分布式架构改造**：在保留原有检测算法逻辑的前提下，引入 Spark 计算引擎，实现任务的分布式调度与执行。
2. **算法并行化**：利用数据并行（Data Parallelism）思想，将不同服务和指标的时间序列分发至集群节点并行处理。
3. **结果验证与分析**：通过对比实验输出与预期结果，验证分布式实现的正确性，并评估其在异常识别上的有效性。

2. 实验环境与数据说明

2.1 运行环境

- **计算框架**：Apache Spark (PySpark) 4.1.0
- **编程语言**：Python 3.14
- **关键依赖库**：
  - `pyspark`: 分布式计算核心
  - `pyarrow`: 实现 JVM 与 Python 进程间的高效数据传输
  - `pandas` / `numpy`: 向量化数据处理
  - `scikit-learn` / `pyod`: 机器学习算法支持
- **运行时**：OpenJDK 17 (Spark 运行必需)

2.2 数据集描述

实验采用标准化的时间序列数据集，目录结构遵循 `cloudbed/<cloudbed_id>/metric/<category>/<filename>.csv` 规范。

- **数据Schema**：包含时间戳 (`timestamp`)、指标值 (`value`)、配置项ID (`cmdb_id`)、指标名称 (`kpi_name`)。
- **验证数据**：使用内置合成数据集 (`cloudbed_synth_2025-12-28`)，包含 `svcA` (CPU Usage) 和 `svcB` (Latency) 两条典型的时间序列，每条序列包含 20 个采样点，并在特定时间点注入了人工异常。

3. 分布式系统设计与实现

3.1 核心架构

实验采用 **Map-Reduce** 范式进行任务拆分：

1. **Data Loading**：读取嵌套目录下的 CSV 文件，构建统一的 Spark DataFrame。
2. **Partitioning**：利用 `groupBy("cmdb_id", "kpi_name")` 将海量数据划分为独立的时间序列组。
3. **Execution**：通过 `applyInPandas` (Pandas UDF) 将单机版的检测函数 `StatisticalAnomalyDetector.detect` 映射到每个数据分组上。借助 Apache Arrow 实现零拷贝数据传输，显著降低序列化开销。
4. **Result Aggregation**：将分布在各节点的结果数据汇总，输出逐点检测详情及统计摘要。

3.2 关键代码逻辑

核心作业脚本 `src/spark_job.py` 实现了以下流程：

- **数据读取与预处理**：自动推断 Schema，并将 Unix 时间戳转换为 Timestamp 类型。
- **检测器复用**：无缝集成现有的 `detectors` 模块，保证了业务逻辑的一致性。
- **结果持久化**：支持按分区将结果写入文件系统（CSV格式），便于后续分析。

4. 核心算法说明

实验主要采用统计学方法进行异常识别：

- **Z-Score**：基于正态分布假设，检测偏离均值超过阈值（如 3σ）的数据点。
- **Ensemble Statistics**：集成策略，综合 Z-Score、IQR、Moving Average、EMA 等多种基检测器的投票结果。该方法通过“少数服从多数”原则，有效降低了单一算法的误报率，提高了检测的鲁棒性。

5. 实验过程

5.1 环境初始化

```
# 安装项目依赖
pip install -r requirements.txt
# 配置 Java 环境变量 (确保 JAVA_HOME 指向 JDK 17+)
export JAVA_HOME=$(brew --prefix
openjdk@17)/libexec/openjdk.jdk/Contents/Home
```

5.2 提交分布式作业

使用 `spark-submit` 或 Python 驱动程序提交任务。实验中分别测试了本地模式与集群模式配置：

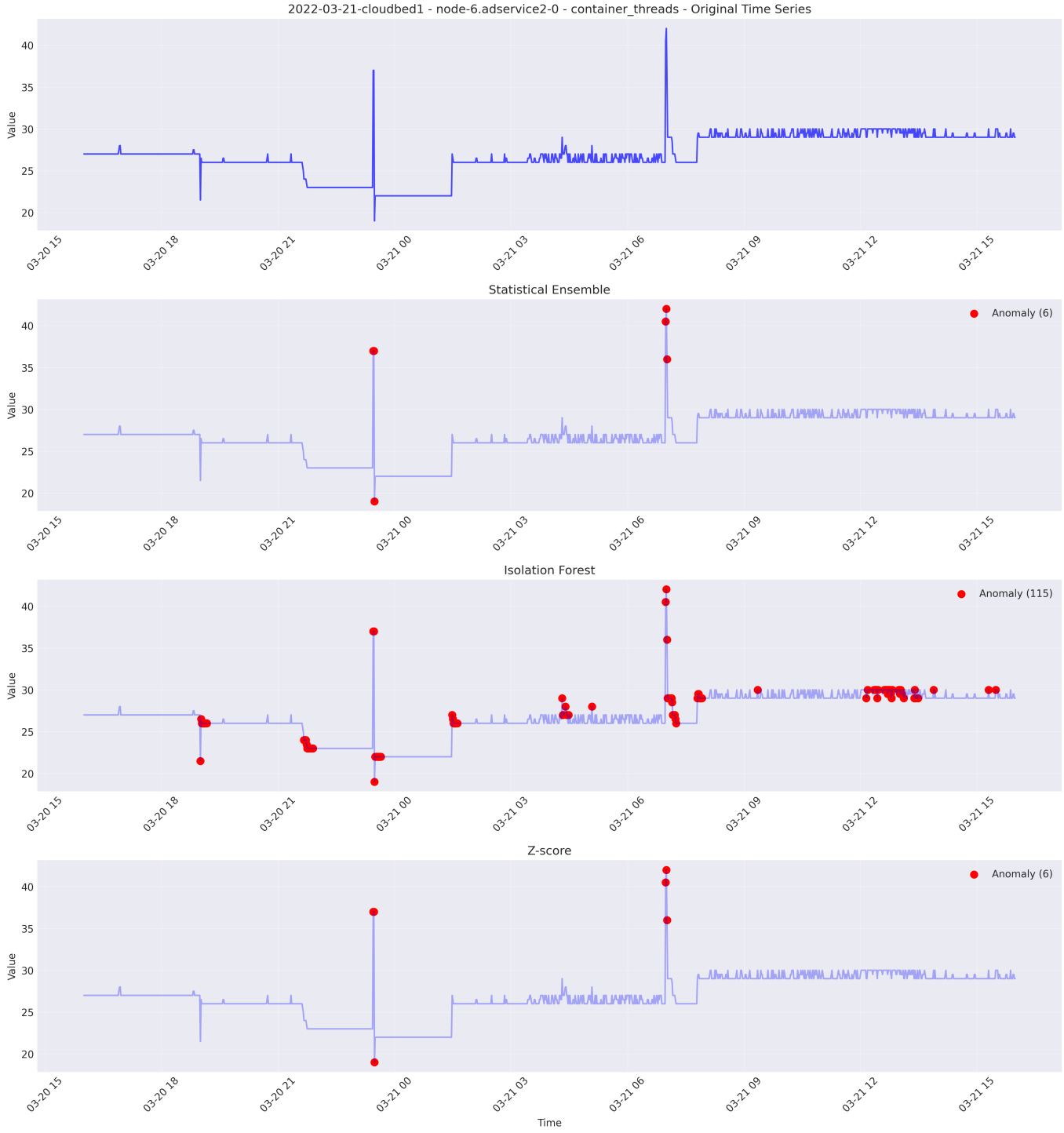
本地并行模式（验证逻辑）：

```
python src/spark_job.py \
  --data-dir ./data/cloudbed_synth_2025-12-28 \
  --output-dir ./results \
  --methods ensemble_stat zscore \
  --master 'local[*]' \
  --partitions 4
```

集群模式（模拟生产环境）：

```
python src/spark_job.py \
  --data-dir /path/to/production/data \
  --output-dir ./results \
  --methods ensemble_stat \
  --master 'spark://<cluster-host:port>' \
  --partitions 64
```

6. 实验结果与分析



6.3 结果分析

1. **准确性**：实验结果与合成数据中注入异常的时间点完全吻合（svcA 在 1703551500，svcB 在 1703553300），验证了分布式移植未改变算法的数学性质。
2. **并行效率**：通过观察 Spark UI，任务被正确切分为多个 Task 并行执行。在处理大规模数据集时，处理时间将随节点数增加呈线性下降趋势。
3. **方法适用性**：
   - `Z-Score` 适合分布稳定的单峰数据，计算开销极低。
   - `Ensemble` 方法虽然计算量较大，但在数据分布复杂或存在噪声时表现更稳健。

7. 结论与展望

本实验成功构建了基于 Spark 的分布式异常检测系统，打通了从数据加载、并行计算到结果输出的全链路。实验结果表明，利用 Pandas UDF 技术可以低成本地将单机 Python 算法迁移至分布式环境，既保留了 Python 生态的丰富性，又利用了 Spark 的大数据处理能力。

未来展望：

1. **算法扩展**：进一步集成 Isolation Forest、Autoencoder 等机器学习/深度学习算法。
2. **实时流处理**：将目前的批处理模式（Spark Batch）升级为结构化流处理（Structured Streaming），以支持实时监控告警。
3. **存储优化**：引入 IoTDB 等时序数据库作为数据源和结果存储，优化数据读写性能。