TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

VIỆN TRÍ TUỆ NHÂN TẠO

BÁO CÁO MÔN HỌC KỸ THUẬT VÀ CÔNG NGHỆ DỮ LIỆU LỚN

ĐỀ TÀI: XÂY DỰNG HỆ THỐNG LOG ANALYSIS TRONG MÔI TRƯỜNG BIG DATA

Nhóm sinh viên thực hiện:

1. Nguyễn Văn Việt - 23020444
2. Ngô Đinh Minh Nhật - 23020408
3. Kiều Đức Nam - 23020404

MỞ ĐẦU

Trong bối cảnh cuộc cách mạng công nghiệp 4.0 diễn ra mạnh mẽ, dữ liệu đã trở thành yếu tố cốt lõi trong việc vận hành, tối ưu hệ thống và hỗ trợ ra quyết định của các doanh nghiệp, tổ chức. Dữ liệu lớn (Big Data) không chỉ bao gồm những tập dữ liệu có khối lượng khổng lồ, tốc độ tăng nhanh mà còn mang tính đa dạng và phức tạp, đòi hỏi các phương pháp xử lý hiện đại, có khả năng mở rộng và tiết kiệm chi phí.

Hệ thống **Log Analysis** được xây dựng nhằm giải quyết nhu cầu thu thập, quản lý và phân tích dữ liệu log từ nhiều nguồn khác nhau theo thời gian thực. Log – dữ liệu dạng chuỗi sự kiện được sinh ra liên tục từ ứng dụng, máy chủ, thiết bị hoặc người dùng – đóng vai trò quan trọng trong việc giám sát hoạt động hệ thống, phát hiện bất thường, ngăn ngừa rủi ro và tối ưu hóa hiệu suất vận hành. Tuy nhiên, lượng log phát sinh mỗi ngày là vô cùng lớn, tạo ra thách thức trong việc lưu trữ, xử lý và khai thác giá trị thực tiễn từ nguồn dữ liệu này.

Dự án **"Xây dựng hệ thống Log Analysis trong Môi trường Big Data"** được thực hiện nhằm xây dựng một pipeline xử lý dữ liệu log hoàn chỉnh, kết hợp xử lý theo lô (batch) và xử lý theo thời gian thực (streaming). Đồng thời, nhóm nghiên cứu ứng dụng các mô hình học máy để phân loại log, phát hiện bất thường và hỗ trợ cảnh báo tự động. Việc tích hợp các công nghệ hiện đại như *Kafka, Spark, Hadoop HDFS, HBase* và các thuật toán học máy giúp hệ thống đạt được khả năng mở rộng, tốc độ xử lý cao và đáp ứng yêu cầu phân tích log trong thực tế.

Báo cáo này trình bày chi tiết các bước triển khai hệ thống Log Analysis , bao gồm thu thập – tiền xử lý dữ liệu, xây dựng kiến trúc xử lý Big Data, ứng dụng thuật toán học máy và đánh giá hiệu quả mô hình. Nhóm thực hiện hy vọng báo cáo sẽ mang lại cái nhìn tổng quan và thực tiễn về việc ứng dụng công nghệ dữ liệu lớn trong bài toán phân tích log, từ đó mở ra các hướng phát triển cho các hệ thống giám sát và cảnh báo tự động trong tương lai.

Nội dung báo cáo gồm các chương:

- **Chương 1:** Giới thiệu chung
- **Chương 2:** Phương pháp và thuật toán trong xử lý dữ liệu
- **Chương 3:** Quá trình thực nghiệm
- **Chương 4:** Tổng kết quá trình thực nghiệm

PHÂN CÔNG NHIỆM VỤ

Thành viên	Nhiệm vụ
Nguyễn Văn Việt	Tìm hiểu các công nghệ dữ liệu lớn, xây dựng cấu trúc báo cáo và viết nội dung mô tả kiến trúc hệ thống Log Analysis. Tìm hiểu và triển khai kiến trúc Lambda với Batch Layer , Stream Layer và Severing. Huấn luyện mô hình Machine Learning
Ngô Đinh Minh Nhật	Thu thập dữ liệu log, xây dựng pipeline tiền xử lý và tham gia thiết kế thuật toán phân tích log. Xây dựng app hiển thị kết quả bằng Flask Python và lưu trữ vào HBase
Kiều Đức Nam	Tìm hiểu và tham gia xây dựng Batch Layer. Triển khai lưu trữ trên PosterSQL và phân tích dữ liệu bằng Power-BI. Hoàn thiện báo cáo và xây dựng file md.

MỤC LỤC

Mục lục

1 Giới thiệu chung

1.1 Tổng quan về dữ liệu lớn

Theo wikipedia: Dữ liệu lớn (Big data) là một thuật ngữ chỉ bộ dữ liệu lớn hoặc phức tạp mà các phương pháp truyền thống không đủ các ứng dụng để xử lý dữ liệu này [**WikipediaBigData**].

Theo Gartner: Dữ liệu lớn là những nguồn thông tin có đặc điểm chung khối lượng lớn, tốc độ nhanh và dữ liệu định dạng dưới nhiều hình thức khác nhau, do đó muốn khai thác được đòi hỏi phải có hình thức xử lý mới để đưa ra quyết định, khám phá và tối ưu hóa quy trình.[**GartnerBigData**].

Với sự phát triển bùng nổ của công nghệ 4.0, lượng dữ liệu khổng lồ được tạo ra nhanh chóng từ tất cả các lĩnh vực:

1. Dữ liệu hành chính (phát sinh từ chương trình của một tổ chức, có thể là chính phủ hay phi chính phủ). Ví dụ: hồ sơ y tế điện tử ở bệnh viện, hồ sơ bảo hiểm, hồ sơ ngân hàng, hồ sơ định danh điện tử ...;
2. Dữ liệu từ hoạt động thương mại (phát sinh từ các giao dịch giữa hai thự thể) Ví dụ: các giao dịch thẻ tín dụng, giao dịch trên mạng, ...
3. Dữ liệu từ các thiết bị cảm biến như thiết bị chụp hình ảnh vệ tinh, cảm biến đường, cảm biến khí hậu;
4. Dữ liệu từ các thiết bị theo dõi. Ví dụ : theo dõi dữ liệu từ điện thoại di động, GPS , ảnh vệ tinh,...
5. Dữ liệu từ các hành vi Ví dụ như tìm kiếm trực tuyến (tìm kiếm sản phẩm, dịch vụ hay thông tin khác), đọc các trang mạng trực tuyến...
6. Dữ liệu từ các thông tin về ý kiến, quan điểm của các cá nhân, tổ chức, trên các phương tiện thông tin xã hội

Hình 1: Các nguồn phổ biến của bigdata.

Theo đặc trưng phổ biến, dữ liệu lớn được mô tả qua mô hình

- Volume (Khối lượng): Là đặc điểm tiêu biểu nhất, kích cỡ của big data ngày càng tăng, sử dụng công nghệ đám mây mới có khả năng lưu trữ
- Velocity (Tốc độ): Khối lượng dữ liệu gia tăng rất nhanh / Xử lý dữ liệu nhanh ở mức thời gian thực (real-time)
- Variety (Đa dạng): Đối với dữ liệu truyền thống chúng ta hay nói đến dữ liệu có cấu trúc. Ngày nay hơn 80% dữ liệu được sinh ra là phi cấu trúc (tài liệu, blog, hình ảnh, video). Big Data cho phép liên kết và phân tích nhiều dạng dữ liệu khác nhau
- Veracity (Độ tin cậy): Bài toán phân tích và loại bỏ dữ liệu thiếu chính xác và nhiễu đang là tính chất quan trọng của bigdata.
- Value (Giá trị): Khi bắt đầu triển khai xây dựng dữ liệu lớn thì việc đầu tiên chúng ta cần phải làm đó là xác định được giá trị của thông tin mang lại như thế nào, khi đó chúng ta mới có quyết định nên triển khai dữ liệu lớn hay không

Hình 2: Mô hình 5V

1.2 Tổng quan về dữ liệu Log HDFS

Hadoop Distributed File System (HDFS) là thành phần lưu trữ cốt lõi của hệ sinh thái Hadoop, và trong quá trình vận hành, hệ thống tạo ra một lượng lớn dữ liệu log phản ánh toàn bộ trạng thái, hành vi và hoạt động của các thành phần khác nhau. Dữ liệu log HDFS là nguồn thông tin quan trọng trong việc giám sát hệ thống, phân tích hiệu năng và phát hiện bất thường.

Dữ liệu log trong hệ thống HDFS được sinh ra từ nhiều tiến trình cốt lõi đảm nhiệm các chức năng khác nhau trong quá trình quản lý và vận hành hệ thống tệp phân tán. Trước hết, NameNode là tiến trình trung tâm quản lý metadata của toàn bộ hệ thống, bao gồm cấu trúc namespace, thông tin về các block, trạng thái cluster và các thao tác của client. Mọi thay đổi liên quan tới việc tạo file, xóa file, phân bổ block hay cập nhật trạng thái đều được NameNode ghi lại dưới dạng log.

Song song với NameNode, các DataNode là nguồn sinh log liên tục khi chúng thực hiện nhiệm vụ lưu trữ và phục vụ block dữ liệu. Mỗi yêu cầu đọc hoặc ghi block từ client hay NameNode đều dẫn đến nhiều dòng log phản ánh trạng thái I/O, quá trình truyền dữ liệu, replicate block hoặc các cảnh báo liên quan tới phần cứng và mạng.

Bên cạnh hai tiến trình chính này, hệ thống còn có SecondaryNameNode hoặc Checkpoint Node, những thành phần hỗ trợ trong việc tổng hợp và đồng bộ metadata nhằm giảm tải cho NameNode. Các hoạt động lấy snapshot, tổng hợp file image và checkpoint cũng được ghi lại dưới dạng log.

Ngoài ra, các log từ phía client thông qua lớp ClientProtocol cũng đóng góp đáng kể vào tổng lượng log của hệ thống. Khi người dùng gửi yêu cầu đọc, ghi hoặc thao tác file qua API HDFS, client sẽ ghi lại trạng thái kết nối, phản hồi từ server và các lỗi phát sinh. Tập hợp log từ các tiến trình này tạo nên nguồn dữ liệu đa dạng phản ánh đầy đủ trạng thái của hệ thống HDFS.

Đặc trưng của dữ liệu log HDFS

Dữ liệu log trong hệ thống HDFS mang nhiều đặc thù riêng biệt so với các dạng dữ liệu thông thường. Trước hết, log được sinh ra liên tục theo thời gian và theo từng sự kiện trong hệ thống. Mỗi thao tác đọc hoặc ghi block, quá trình replicate dữ liệu giữa các DataNode, hoạt động kết nối từ client, các gói heartbeat hay bất kỳ lỗi nào xảy ra đều tạo thành một dòng log mới. Điều này khiến log phản ánh trực tiếp toàn bộ trạng thái và hoạt động nội bộ của hệ thống phân tán.

Bên cạnh đó, log HDFS không có cấu trúc cố định mà tồn tại dưới dạng văn bản tự do theo từng dòng. Mỗi dòng thường bao gồm timestamp, mức độ log (INFO, WARN, ERROR), tên dịch vụ hoặc module phát sinh log, địa chỉ IP liên quan và nội dung thông điệp của sự kiện. Việc thiếu cấu trúc rõ ràng khiến quá trình phân tích log yêu cầu các bước trích xuất và chuẩn hóa trước khi xử lý.

Một đặc điểm nổi bật khác là sự đa dạng về loại sự kiện. Log có thể ghi lại hoạt động truyền dữ liệu của `DataXceiver`, quá trình replicate block, báo cáo block từ DataNode, các hoạt động liên quan đến file system, sự kiện khởi động hoặc tắt dịch vụ, các exception hay cảnh báo trong hệ thống. Sự đa dạng này mang lại nhiều thông tin quan trọng nhưng cũng đặt ra thách thức cho quá trình nhận dạng và phân loại sự kiện.

Cuối cùng, dữ liệu log HDFS có khối lượng rất lớn. Trong các hệ thống thực tế, log có thể đạt tới hàng chục triệu dòng mỗi ngày. Ngay cả với bộ dữ liệu HDFS_v1 được sử dụng trong dự án, số lượng log cũng đã vượt hơn 11 triệu dòng. Khối lượng lớn cùng tốc độ sinh liên tục khiến việc lưu trữ, xử lý và phân tích log trở thành một bài toán dữ liệu lớn đúng nghĩa.

Cấu trúc điển hình của một dòng log HDFS

Một dòng log HDFS thường bao gồm:

- Timestamp: thời điểm ghi log theo độ phân giải milliseconds.
- Thread / Process: tên tiến trình sinh log.

- Level: INFO, WARN, ERROR.
- Module: ví dụ `DataNode $DataXceiver`, `FSNamesystem`, `ClientProtocol`.
- Message: nội dung mô tả sự kiện.

1.3 Tổng quan về hệ sinh thái Hadoop

Hadoop là một nền tảng mã nguồn mở được thiết kế để lưu trữ và xử lý dữ liệu lớn theo mô hình phân tán. Ban đầu được phát triển bởi Apache, Hadoop nhanh chóng trở thành một trong những giải pháp phổ biến nhất trong lĩnh vực Big Data nhờ khả năng mở rộng linh hoạt, chịu lỗi tốt và phù hợp với các hệ thống sử dụng phần cứng giá rẻ.

Cốt lõi của Hadoop bao gồm ba thành phần chính: Hadoop Distributed File System (HDFS), YARN và MapReduce. Tuy nhiên, hệ sinh thái Hadoop đã phát triển vượt xa các thành phần nền tảng này, bao gồm nhiều công cụ hỗ trợ xử lý dữ liệu đa dạng, từ lưu trữ, xử lý theo lô (batch) – thời gian thực (streaming), đến truy vấn, giám sát và quản lý dữ liệu.

Hadoop Distributed File System (HDFS) là hệ thống tệp phân tán được thiết kế để lưu trữ dữ liệu ở quy mô rất lớn, có thể lên đến hàng petabyte. Dữ liệu được chia thành nhiều khối nhỏ và phân phối trên nhiều node trong cluster, nhờ đó hệ thống đạt được khả năng chịu lỗi cao và cải thiện tốc độ đọc/ghi. Khi một node gặp sự cố, HDFS tự động phục hồi dữ liệu từ các bản sao được lưu trữ trên các node khác, đảm bảo tính sẵn sàng và độ tin cậy của hệ thống.

Bên cạnh HDFS, YARN (Yet Another Resource Negotiator) là thành phần chịu trách nhiệm quản lý tài nguyên và lập lịch thực thi cho các ứng dụng chạy trong hệ sinh thái Hadoop. YARN tách riêng chức năng quản lý tài nguyên khỏi mô hình MapReduce truyền thống, cho phép Hadoop hỗ trợ linh hoạt nhiều mô hình xử lý khác nhau như Spark, Flink hay Hive, và từ đó mở rộng khả năng ứng dụng của toàn hệ thống.

Cuối cùng, MapReduce đóng vai trò là mô hình lập trình song song truyền thống của Hadoop, đặc biệt phù hợp cho các tác vụ xử lý batch trên lượng dữ liệu lớn. Mô hình này hoạt động qua hai giai đoạn chính: giai đoạn Map thực hiện phân chia và xử lý dữ liệu song song ở các node, trong khi giai đoạn Reduce tổng hợp và hợp nhất các kết quả trung gian để tạo ra đầu ra cuối cùng. Cách tiếp cận này giúp MapReduce thực thi hiệu quả các công việc tính toán quy mô lớn không yêu cầu thời gian thực.

1.4 Tổng quan về các công nghệ sử dụng

Để xây dựng hệ thống Log Analysis có khả năng xử lý log theo thời gian thực và theo lô (batch), dự án sử dụng một tập hợp các công nghệ phổ biến trong lĩnh vực dữ liệu lớn. Mỗi công nghệ đảm nhiệm một vai trò riêng biệt trong pipeline, từ thu thập dữ liệu, lưu trữ phân tán, xử lý song song cho đến thống kê và xây dựng giao diện hiển thị. Dưới đây là tổng quan chi tiết về các công nghệ chính được sử dụng trong hệ thống.

Apache Kafka

Apache Kafka là lớp đầu tiên trong pipeline, đóng vai trò tiếp nhận dữ liệu theo thời gian thực. Kafka hoạt động theo mô hình publish–subscribe, nơi các ứng dụng đóng vai trò Producer gửi log mới vào các topic, trong khi Consumer đọc dữ liệu luồng và chuyển vào hệ thống xử lý.

- Kafka Producer gửi log từ các ứng dụng hoặc file nguồn lên các topic của Kafka.
- Kafka Consumer nhận dữ liệu theo luồng và chuyển tiếp đến Spark, HBase hoặc các hệ thống downstream khác.

Nhờ khả năng chịu tải lớn, phân vùng (partition) linh hoạt và cơ chế lưu trữ bền vững, Kafka đảm bảo hệ thống Log Analysis có thể tiếp nhận log ổn định, không mất mát và hoạt động liên tục.

Hadoop Distributed File System (HDFS)

HDFS là hệ thống tệp phân tán dùng để lưu trữ dữ liệu log khối lượng lớn, đảm nhiệm vai trò lưu trữ lâu dài cho dữ liệu log. Các file log lớn khi được đưa vào HDFS sẽ được tự động chia thành các block và phân phối trên nhiều DataNode, đảm bảo độ tin cậy và khả năng phục hồi khi có lỗi phần cứng. Ngoài ra, HDFS cũng đóng vai trò như một kho dữ liệu trung tâm (data lake), nơi Spark có thể đọc trực tiếp để thực hiện các tác vụ xử lý batch.

- Lưu trữ quy mô lớn: Dữ liệu được chia thành block và sao chép (replicate) trên nhiều node, đảm bảo an toàn và khả năng phục hồi.
- Tích hợp Spark: Spark có thể đọc trực tiếp dữ liệu từ HDFS để xử lý batch.
- Data Lake: HDFS được dùng làm kho dữ liệu lịch sử để huấn luyện mô hình và phân tích.

HDFS giúp hệ thống Log Analysis lưu trữ dữ liệu lâu dài, ổn định và mở rộng dễ dàng theo nhu cầu.

Apache Spark (Batch Processing)

Trong lớp xử lý theo lô, Apache Spark được sử dụng nhờ khả năng tính toán phân tán và xử lý dữ liệu trong bộ nhớ. Spark thực hiện các bước làm sạch, chuẩn hóa và trích xuất đặc trưng từ log, đồng thời tổng hợp các bảng thống kê phục vụ phân tích hoặc huấn luyện mô hình. Với Spark SQL, dữ liệu log được biểu diễn dưới dạng bảng giúp việc truy vấn, lọc và phân tích trở nên linh hoạt và hiệu quả hơn.

- Spark Core: Xử lý dữ liệu phân tán với các thao tác chuyển đổi và hành động.
- Spark SQL: Hỗ trợ truy vấn và xử lý dữ liệu log dưới dạng bảng.

- Batch Job: Được sử dụng để làm sạch log, tạo đặc trưng (feature engineering), tổng hợp thống kê và chuẩn bị dữ liệu cho HBase và PostgreSQL.

Spark batch là xương sống của phần xử lý lô trong kiến trúc Lambda của hệ thống Log Analysis .

Apache HBase

Ở tầng dữ liệu thời gian thực, Apache HBase được dùng để lưu trữ các bản ghi đã được xử lý ngay lập tức từ Kafka hoặc từ các module streaming. Do được thiết kế theo mô hình NoSQL dạng cột, HBase hỗ trợ đọc và ghi với độ trễ thấp, đặc biệt phù hợp cho các truy vấn theo khóa như tìm kiếm log theo `BlockId` hoặc theo timestamp. Điều này giúp giao diện người dùng truy xuất log realtime nhanh chóng và ổn định. Trong dự án Log Analysis , HBase được sử dụng để lưu:

- Dữ liệu log đã chuẩn hóa
- Kết quả xử lý từ Spark Streaming hoặc Kafka Consumer;
- Dữ liệu realtime phục vụ UI.

HBase phù hợp cho các truy vấn theo khóa (row key), tốc độ đọc/ghi nhanh và lưu trữ khối lượng log rất lớn.

PostgreSQL (Lưu thống kê)

PostgreSQL được sử dụng để lưu trữ các bảng dữ liệu tổng hợp sinh ra từ quy trình batch. Đây là nơi chứa dữ liệu phục vụ phân tích, thống kê và báo cáo, nhờ khả năng truy vấn SQL mạnh mẽ và cơ chế quản lý giao dịch đảm bảo tính toàn vẹn dữ liệu. PostgreSQL bổ sung cho HBase bằng việc cung cấp một tầng dữ liệu có cấu trúc và phù hợp với các tác vụ phân tích truyền thống.

- Lưu kết quả batch: Các bảng summary từ Spark được ghi vào PostgreSQL.
- Truy vấn SQL: Phù hợp cho dashboard, thống kê, và báo cáo định kỳ.
- Tính toàn vẹn: Hỗ trợ ACID đảm bảo dữ liệu chính xác và đáng tin cậy.

PostgreSQL giúp hệ thống kết hợp giữa dữ liệu thô ở HDFS và dữ liệu realtime ở HBase để tạo ra các bảng phân tích chất lượng cao.

Flask (User Interface)

Flask được dùng để xây dựng giao diện người dùng và API. Nhờ tính linh hoạt của Python, Flask cho phép kết nối trực tiếp tới HBase để hiển thị log theo thời gian thực và tới PostgreSQL để hiển thị các thống kê batch. Giao diện dựa trên Flask cung cấp cho người dùng khả năng quan sát log, theo dõi hoạt động hệ thống và xem kết quả mô hình một cách thuận tiện và trực quan.

- Xây dựng API realtime: Kết nối đến HBase/Kafka để trả dữ liệu log trực tiếp.
- Trang UI hiển thị log: Bao gồm bảng log, thống kê, biểu đồ và kết quả mô hình.
- Dễ tích hợp: Flask có thể kết nối linh hoạt với Spark, Kafka, PostgreSQL hoặc bất kỳ API nào.

Flask tạo ra cầu nối giữa dữ liệu log và người dùng cuối, giúp giám sát hệ thống dễ dàng và trực quan.

Power-BI (Trực quan hóa dữ liệu)

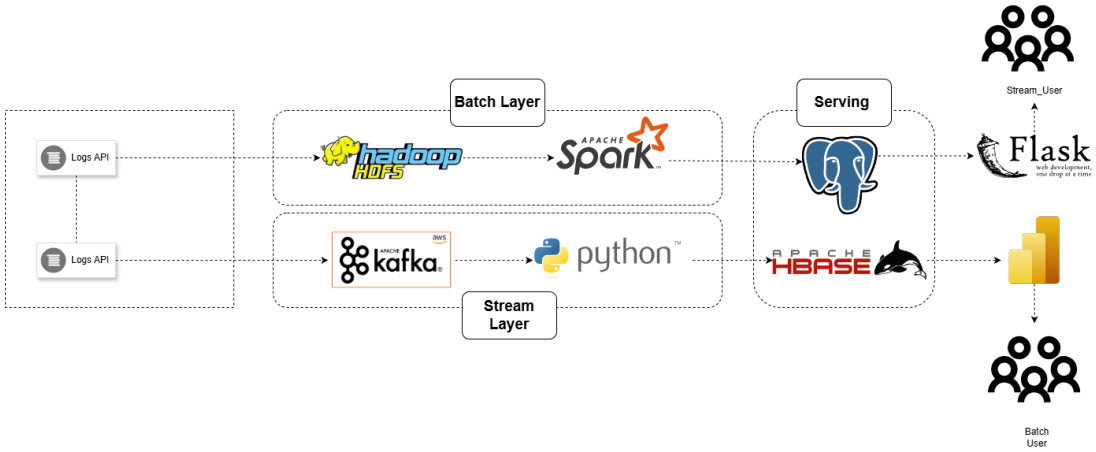
Power BI được sử dụng như công cụ trực quan hóa dữ liệu trong hệ thống Log Analysis, giúp trình bày các bảng thống kê, kết quả xử lý batch và các chỉ số phân tích một cách trực quan và dễ hiểu. Đây là công cụ rất phù hợp trong quá trình đánh giá hệ thống, hỗ trợ người dùng và nhóm phát triển quan sát xu hướng log, phát hiện bất thường và theo dõi trạng thái hệ thống một cách tổng quan.

Trong hệ thống, Power BI kết nối trực tiếp với cơ sở dữ liệu PostgreSQL – nơi lưu trữ toàn bộ các bảng tổng hợp được tạo ra từ pipeline Spark batch. Nhờ vậy, Power BI có thể khai thác dữ liệu một cách hiệu quả mà không tác động tới hệ thống xử lý realtime Nhờ khả năng trực quan hóa mạnh mẽ, Power BI giúp người dùng nhanh chóng nắm bắt được hành vi của hệ thống, phát hiện sớm các bất thường tiềm ẩn và đánh giá hiệu quả pipeline xử lý dữ liệu. Công cụ này đóng vai trò quan trọng trong tầng hiển thị (Serving Layer), bổ sung cho giao diện Flask và hoàn thiện quy trình phân tích log trong hệ thống dữ liệu lớn.

2 Phương pháp và Thuật toán trong xử lý dữ liệu

Trong hệ thống Log Analysis , việc phân tích dữ liệu log lớn yêu cầu sự kết hợp giữa các phương pháp xử lý dữ liệu truyền thống và các kỹ thuật học máy hiện đại. Phần này trình bày chi tiết pipeline xử lý theo mô hình Lambda, các bước chuyển đổi dữ liệu, cùng các thuật toán được sử dụng trong phần phân tích và phát hiện bất thường.

2.1 Kiến trúc Lambda trong trong thiết kế hệ thống Log Analysis



Hình 3: Kiến trúc Lambda.

Kiến trúc Lambda là mô hình xử lý dữ liệu được thiết kế nhằm đáp ứng đồng thời hai nhu cầu: xử lý dữ liệu theo thời gian thực và xử lý dữ liệu theo lô với độ chính xác cao. Trong hệ thống Log Analysis , kiến trúc này được áp dụng để đảm bảo rằng dữ liệu log từ HDFS và các nguồn khác được thu thập, xử lý, lưu trữ và phục vụ phân tích một cách nhất quán, nhanh chóng và đáng tin cậy.

Kiến trúc Lambda của Log Analysis bao gồm ba lớp chính: **Batch Layer**, **Speed Layer** và **Serving Layer**. Mỗi lớp đảm nhiệm một vai trò cụ thể nhưng kết hợp chặt chẽ với nhau để xây dựng pipeline xử lý dữ liệu hoàn chỉnh.[**Lambda2014**]

Batch Layer

Batch Layer là tầng chịu trách nhiệm xử lý toàn bộ dữ liệu log lịch sử với quy mô lớn. Toàn bộ log thô được đưa vào HDFS, nơi dữ liệu được lưu trữ theo dạng phân tán, đảm bảo khả năng chịu lỗi khi một phần cụm hệ thống gặp sự cố. Spark được sử dụng để thực hiện các tác vụ tiền xử lý, bao gồm chuẩn hóa timestamp, loại bỏ các dòng log bị lỗi và gom nhóm các sự kiện liên quan theo `BlockId`. Sau quá trình làm sạch, dữ liệu được sử dụng để trích xuất đặc trưng, chẳng hạn như số lượng dòng log, thời gian xử lý block hoặc các mẫu sự kiện quan trọng. Những đặc trưng này được dùng để xây dựng các bảng dữ liệu tổng hợp phục vụ huấn luyện mô hình và các phân tích sau này. Với khả năng tính toán phân tán mạnh mẽ của Apache Spark, Batch Layer đảm bảo đạt độ chính xác cao trên lượng dữ liệu lớn, tuy nhiên không đáp ứng được yêu cầu thời gian thực.

- Lưu trữ dữ liệu thô (raw logs): Toàn bộ log gốc được đưa vào HDFS, lưu ở dạng file phân tán, đảm bảo tính chịu lỗi và khả năng mở rộng khi dung lượng tăng.
- Tiền xử lý và chuẩn hóa dữ liệu: Spark được sử dụng để đọc log từ HDFS, loại bỏ các dòng không hợp lệ, chuẩn hóa timestamp và gom nhóm các dòng log theo BlockId.

- Tính toán đặc trưng (feature engineering): Batch Layer thực hiện việc trích xuất đặc trưng như số dòng log liên quan đến block, độ dài thời gian xử lý block, số lượng Event các mẫu sự kiện quan trọng.
- Tính toán dữ liệu tổng hợp (batch views): Kết quả sau xử lý được lưu lại dưới dạng bảng dữ liệu tổng hợp đã làm sạch, phục vụ việc huấn luyện mô hình và phân tích chuyên sâu.

Ưu điểm của Batch Layer là khả năng xử lý lượng dữ liệu lớn với độ chính xác cao, nhờ tận dụng sức mạnh tính toán phân tán của Apache Spark. Tuy nhiên, lớp này không đáp ứng yêu cầu thời gian thực.

Speed Layer

Không giống với Batch Layer, Speed Layer được thiết kế để xử lý dữ liệu log vừa sinh ra trong thời gian thực. Trong hệ thống Log Analysis, tầng này được triển khai dựa trên Kafka kết hợp với Python Consumer. Kafka đảm nhiệm việc tiếp nhận log realtime thông qua các topic, sau đó Consumer xử lý ngay lập tức để trích xuất các đặc trưng cơ bản và gắn nhãn sơ bộ. Dữ liệu sau khi xử lý được ghi trực tiếp vào HBase, nơi hỗ trợ đọc và ghi với độ trễ thấp, giúp giao diện người dùng có thể truy vấn ngay lập tức. Tầng này cũng có thể tích hợp các mô hình phát hiện bất thường nhẹ, cho phép đưa ra cảnh báo ngay khi log có dấu hiệu không bình thường.

- Tiếp nhận log realtime từ Kafka: Kafka Producer gửi log từ hệ thống nguồn vào các topic, sau đó Consumer nhận và chuyển vào pipeline phân tích.
- Xử lý nhanh dữ liệu mới: Consumer Python thực hiện parsing tức thời, trích xuất đặc trưng cơ bản và đánh nhãn.
- Cập nhật dữ liệu realtime vào HBase: Do HBase hỗ trợ truy cập đọc/ghi ngẫu nhiên với độ trễ thấp, dữ liệu được ghi ngay vào bảng để UI có thể truy vấn tức thời.
- Cảnh báo sự kiện bất thường: Speed Layer được tích hợp mô hình bất thường (anomaly detection) để phát hiện dấu hiệu lỗi ngay khi log xuất hiện.

Speed Layer đảm bảo hệ thống Log Analysis có khả năng phản hồi nhanh với log mới, tạo ra chế độ xem thời gian thực cho người dùng.

Serving Layer

Serving Layer là nơi cung cấp dữ liệu cho các ứng dụng trực quan hóa và phân tích. Trong hệ thống Log Analysis, tầng này được xây dựng bằng HBase, PostgreSQL và Flask. HBase giữ vai trò lưu trữ dữ liệu realtime đã xử lý; PostgreSQL lưu các bảng thống kê và dữ liệu tổng hợp từ Spark batch, phù hợp cho báo cáo, dashboard và các phân tích chuyên sâu; cuối cùng, Flask được sử dụng để phát triển giao diện người dùng và xây dựng API, cho phép truy vấn log realtime từ HBase hoặc xem các thống kê tổng hợp từ PostgreSQL. Nhờ đó, Serving Layer tạo ra cầu nối nhất quán giữa hai tầng Batch và Speed, đảm bảo người dùng có thể truy cập đầy đủ dữ liệu lịch sử lẫn dữ liệu realtime.

- HBase: lưu trữ dữ liệu realtime đã xử lý, phục vụ các truy vấn tốc độ cao cho UI.
- PostgreSQL: lưu dữ liệu tổng hợp, thống kê và các bảng kết quả batch từ Spark,phù hợp cho báo cáo, dashboard , phân tích chuyên sâu và hiển thị bằng Power-BI
- Flask: xây dựng giao diện người dùng (UI) và API, cho phép truy vấn dữ liệu realtime từ HBase .

Serving Layer giúp hợp nhất dữ liệu từ hai tầng Batch và Speed, đảm bảo UI và người dùng cuối có thể truy cập cả dữ liệu lịch sử và dữ liệu thời gian thực một cách nhất quán.

Tổng kết về kiến trúc Lambda

Sự kết hợp giữa ba tầng Batch, Speed và Serving giúp hệ thống Log Analysis đạt khả năng xử lý dữ liệu log lớn một cách chính xác, đồng thời phản hồi nhanh trước dữ liệu mới sinh ra. Kiến trúc này đem lại khả năng mở rộng linh hoạt và hỗ trợ triển khai các mô hình học máy trong cả hai chế độ batch và streaming, đáp ứng những yêu cầu phức tạp của môi trường dữ liệu lớn hiện đại.

Kiến trúc Lambda là nền tảng giúp Log Analysis vận hành như một hệ thống phân tích log hiệu quả, đáp ứng đầy đủ yêu cầu về tốc độ, độ chính xác và tính sẵn sàng trong môi trường dữ liệu lớn.

2.2 Thuật toán LightGBM

LightGBM (Light Gradient Boosting Machine) là một thuật toán thuộc nhóm Gradient Boosting Decision Tree (GBDT), được thiết kế để xử lý các bài toán phân loại và hồi quy với quy mô dữ liệu lớn. Thuật toán hướng tới mục tiêu tối ưu tốc độ huấn luyện, giảm bộ nhớ sử dụng và mở rộng tốt trên các tập dữ liệu nhiều đặc trưng. LightGBM đạt được những mục tiêu này thông qua hai kỹ thuật cốt lõi là Histogram-based Decision Tree Learning và Leaf-wise Growth Strategy, kết hợp cùng cơ chế GOSS và EFB để tăng tốc và giảm kích thước dữ liệu [**Ke2017LightGBM**]..

Nguyên lý GBDT

LightGBM xây dựng mô hình bằng cách kết hợp tuần tự nhiều cây quyết định hồi quy. Ở vòng lặp thứ t, dự đoán của mô hình được cập nhật bằng cách cộng thêm dự đoán của cây mới theo công thức:

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i),$$

trong đó $f_t(x)$ là cây được huấn luyện từ gradient và Hessian của hàm mất mát tại vòng trước. Trong các bài toán phân loại nhị phân, hàm mất mát thường được sử dụng là Logistic Loss với gradient và Hessian tương ứng:

$$l(y_i, \hat{y}_i) = -\left[y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)\right], \qquad g_i = \frac{\partial l}{\partial \hat{y}_i}, \qquad h_i = \frac{\partial^2 l}{\partial \hat{y}_i^2}.$$

Histogram-based Decision Tree Learning

Thay vì xét toàn bộ giá trị đặc trưng liên tục, LightGBM xây dựng histogram bằng cách phân hoạch giá trị của mỗi đặc trưng thành các bucket cố định. Gradient và Hessian được cộng dồn theo bucket, từ đó quá trình tìm kiếm điểm chia chỉ cần duyệt qua histogram thay vì toàn bộ dữ liệu thô. Kỹ thuật này giảm độ phức tạp tính toán từ $O(n)$ xuống $O(B)$, đồng thời tiết kiệm bộ nhớ và hỗ trợ xử lý song song rất hiệu quả.

Leaf-wise Growth Strategy

LightGBM sử dụng chiến lược tăng trưởng theo lá (leaf-wise), trong đó thuật toán luôn chọn lá có mức độ cải thiện lớn nhất để tiếp tục chia nhỏ thay vì chia theo tầng. Chiến lược leaf-wise giúp giảm nhanh giá trị hàm mất mát và đạt độ chính xác cao hơn với cùng số lượng cây; tuy nhiên cần giới hạn độ sâu bằng tham số `max_depth` để tránh overfitting.

Hàm mục tiêu

Hàm mục tiêu tổng quát của LightGBM bao gồm hàm mất mát và thành phần điều chuẩn:

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{t=1}^{T} \Omega(f_t),$$

trong đó thành phần điều chuẩn được xác định bởi:

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2.$$

LightGBM tối ưu hàm mục tiêu bằng cách khai triển Taylor bậc hai:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \Omega(f_t).$$

Trọng số lá tối ưu và Gain

Với mỗi lá chứa tập mẫu I_j, giá trị dự đoán tối ưu được tính:

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}.$$

Độ cải thiện khi tách một lá thành hai lá được tính bởi:

$$Gain = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma.$$

GOSS (Gradient-based One-Side Sampling)

Để giảm số lượng mẫu nhưng vẫn giữ lại thông tin quan trọng, LightGBM áp dụng GOSS. Thuật toán giữ toàn bộ các mẫu có gradient lớn và chỉ lấy ngẫu nhiên một phần nhỏ các mẫu có gradient nhỏ. Các trọng số sau đó được điều chỉnh để đảm bảo quá trình ước lượng không thiên lệch.

EFB (Exclusive Feature Bundling)

Trong các tập dữ liệu thưa, nhiều đặc trưng loại trừ lẫn nhau (không cùng xuất hiện trong một mẫu). LightGBM kết hợp các đặc trưng này vào chung một bundle nhằm giảm số chiều, giúp tăng tốc xây dựng histogram và giảm bộ nhớ.

Ưu điểm của LightGBM

LightGBM là một trong những thuật toán tăng cường cây quyết định hiệu quả nhất cho bài toán dữ liệu lớn nhờ tốc độ huấn luyện nhanh, khả năng mở rộng tốt và khả năng mô hình hoá các đặc trưng phức tạp. Cơ chế leaf-wise và histogram giúp mô hình đạt độ chính xác cao với chi phí tính toán thấp. LightGBM cũng hỗ trợ tốt bài toán mất cân bằng nhờ tham số `scale_pos_weight` và cung cấp khả năng phân tích tầm quan trọng đặc trưng, phù hợp với bài toán phát hiện bất thường trong log HDFS.

3 Quá trình thực nghiệm

Chương này trình bày quá trình thực nghiệm của hệ thống Log Analysis, bao gồm quy trình chuẩn bị dữ liệu, tiền xử lý, xây dựng pipeline xử lý batch và realtime, huấn luyện mô hình máy học (XGBoost) và đánh giá kết quả. Toàn bộ thực nghiệm được triển khai trên bộ dữ liệu log HDFS và hệ thống xử lý dữ liệu lớn đã được thiết kế ở các chương trước.

3.1 Chuẩn bị dữ liệu thực nghiệm

Nguồn dữ liệu sử dụng trong quá trình đánh giá hệ thống là tập log **HDFS_v1** từ bộ dữ liệu Loghub. Tập dữ liệu này bao gồm hơn 11 triệu dòng log được sinh ra từ các thành phần khác nhau của hệ thống HDFS như NameNode, DataNode, DataXceiver và ClientProtocol. File log thô ban đầu được đưa vào HDFS để lưu trữ và làm đầu vào cho quá trình xử lý batch.

Do dữ liệu log ở dạng văn bản thô (unstructured text), bước tiền xử lý là cần thiết để chuyển log thành dạng bảng có cấu trúc (structured), phục vụ cho phân tích và huấn luyện mô hình học máy.

3.2 Tiền xử lý bằng Apache Spark

Trong quá trình thực nghiệm, Apache Spark được sử dụng làm công cụ xử lý batch chính nhằm chuyển đổi file log thô của bộ dữ liệu HDFS_v1 (Loghub) thành một tập dữ liệu có cấu trúc phục vụ cho phân tích và huấn luyện mô hình. Phần này mô tả chi tiết toàn bộ quy trình tiền xử lý bám sát với đoạn mã Spark được triển khai trong hệ thống.

Đọc dữ liệu log và các bảng hỗ trợ từ HDFS

Hệ thống khởi tạo `SparkSession` kết nối trực tiếp với cụm Spark và HDFS thông qua tham số `spark.hadoop.fs.defaultFS`. Ba nguồn dữ liệu được nạp: File log thô HDFS (`HDFS.log`); Bảng template sự kiện `HDFS.log_templates.csv`; Bảng nhãn bất thường `anomaly_label.csv`.

File log được đọc bằng `spark.read.text` dưới dạng một DataFrame đơn cột (`value`) chứa toàn bộ dòng log tương ứng với cấu trúc log của HDFS.

```
+----------------------------------------------------------------------------------------------------------------------------------------------------------------+
|value                                                                                                                                                           |
+----------------------------------------------------------------------------------------------------------------------------------------------------------------+
|081109 203518 143 INFO dfs.DataNode$DataXceiver: Receiving block blk_-1608999687919862906 src: /10.250.19.102:54106 dest: /10.250.19.102:50010            |
|081109 203518 35 INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock: /mnt/hadoop/mapred/system/job_200811092030_0001/job.jar. blk_-1608999687919862906|
|081109 203519 143 INFO dfs.DataNode$DataXceiver: Receiving block blk_-1608999687919862906 src: /10.250.10.6:40524 dest: /10.250.10.6:50010               |
|081109 203519 145 INFO dfs.DataNode$DataXceiver: Receiving block blk_-1608999687919862906 src: /10.250.14.224:42420 dest: /10.250.14.224:50010           |
|081109 203519 145 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block blk_-1608999687919862906 terminating                                          |
+----------------------------------------------------------------------------------------------------------------------------------------------------------------+
only showing top 5 rows

+-------+---------------------------------------+
|EventId|EventTemplate                          |
+-------+---------------------------------------+
|E1     |[*]Adding an already existing block[*]|
|E2     |[*]Verification succeeded for[*]       |
|E3     |[*]Served block[*]to[*]                |
|E4     |[*]Got exception while serving[*]to[*]|
|E5     |[*]Receiving block[*]src:[*]dest:[*]   |
+-------+---------------------------------------+
only showing top 5 rows

+-----------------------+-------+
|BlockId                |Label  |
+-----------------------+-------+
...
|blk_7854771516489510256 |Normal |
+-----------------------+-------+
```

Hình 4: DataFrame dữ liệu ban đầu

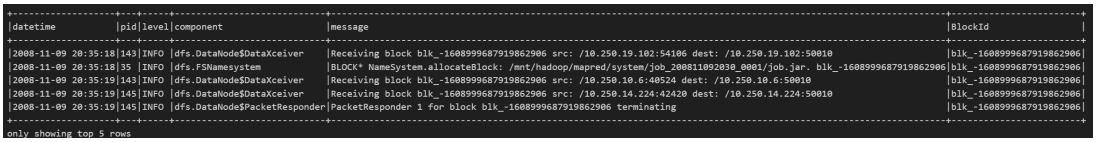
Trích xuất thông tin log bằng Regular Expressions

Do log HDFS có định dạng không cố định theo từng dòng, Spark sử dụng tập hợp các mẫu regex để trích xuất các trường quan trọng. Những trường này bao gồm:

- **timestamp**: thời gian dạng `yyMMdd HHmmss`;
- **pid**: mã tiến trình sinh log;
- **level**: mức độ log (INFO, WARN, ERROR, DEBUG);
- **component**: module sinh log, ví dụ `DataNode$DataXceiver`;
- **message**: nội dung thông điệp log;
- **BlockId**: khóa nhận dạng block (`blk_...`).

Các trường được tạo ra bằng hàm `regexp_extract` trực tiếp trên DataFrame ban đầu. Timestamp sau khi trích xuất được chuyển đổi về kiểu `timestamp` bằng hàm `to_timestamp`.

Sau bước này, DataFrame chỉ giữ lại các cột quan trọng gồm: `datetime`, `pid`, `level`, `component`, `message`, `BlockId`.



datetime	pid	level	component	message	BlockId
2008-11-09 20:35:18	143	INFO	dfs.DataNode$DataXceiver	Receiving block blk_-1608999687919862906 src: /10.250.19.102:54106 dest: /10.250.19.102:50010	blk_-1608999687919862906
2008-11-09 20:35:18	35	INFO	dfs.FSNamesystem	BLOCK* NameSystem.allocateBlock: /mnt/hadoop/mapred/system/job_200811092030_0001/job.jar. blk_-1608999687919862906	blk_-1608999687919862906
2008-11-09 20:35:19	143	INFO	dfs.DataNode$DataXceiver	Receiving block blk_-1608999687919862906 src: /10.250.10.6:40524 dest: /10.250.10.6:50010	blk_-1608999687919862906
2008-11-09 20:35:19	145	INFO	dfs.DataNode$DataXceiver	Receiving block blk_-1608999687919862906 src: /10.250.14.224:42420 dest: /10.250.14.224:50010	blk_-1608999687919862906
2008-11-09 20:35:19	145	INFO	dfs.DataNode$PacketResponder	PacketResponder 1 for block blk_-1608999687919862906 terminating	blk_-1608999687919862906

only showing top 5 rows

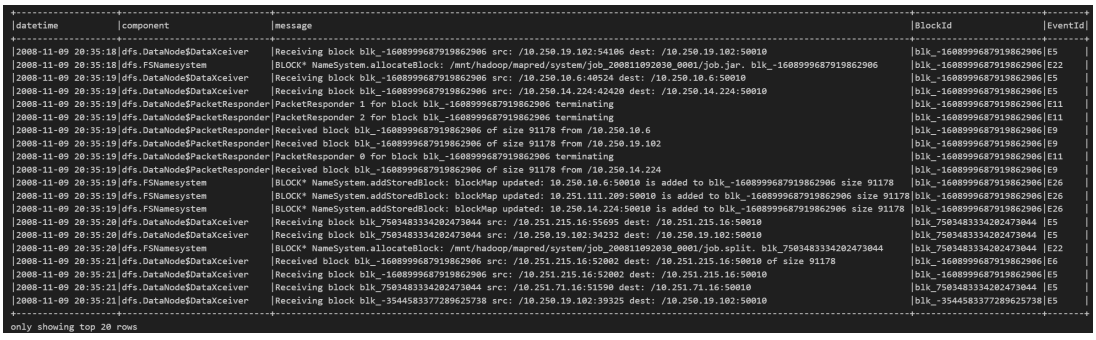
Hình 5: Kết quả trích xuất thông tin.

Gán EventId theo bảng template

Bộ dữ liệu Loghub cung cấp bảng template log dưới dạng `EventTemplate`. Để ánh xạ mỗi dòng log vào mã sự kiện (EventId), hệ thống thực hiện:

1. Chuyển mẫu template thành biểu thức regex thông qua hàm `template_to_regex()`;
2. Duyệt qua từng template và kiểm tra dòng log có khớp với mẫu hay không;
3. Nếu khớp, gán `EventId` tương ứng cho dòng log bằng `when().otherwise()`.

Cơ chế gán này tạo ra cột `EventId` thể hiện kiểu sự kiện xảy ra trong mỗi dòng log. Các dòng log không khớp với bất kỳ template nào được loại bỏ ở bước sau.



datetime	component	message	BlockId	EventId
2008-11-09 20:35:18	dfs.DataNode$DataXceiver	Receiving block blk_-1608999687919862906 src: /10.250.19.102:54106 dest: /10.250.19.102:50010	blk_-1608999687919862906	E5
2008-11-09 20:35:18	dfs.FSNamesystem	BLOCK* NameSystem.allocateBlock: /mnt/hadoop/mapred/system/job_200811092030_0001/job.jar. blk_-1608999687919862906	blk_-1608999687919862906	E22
2008-11-09 20:35:19	dfs.DataNode$DataXceiver	Receiving block blk_-1608999687919862906 src: /10.250.10.6:40524 dest: /10.250.10.6:50010	blk_-1608999687919862906	E5
2008-11-09 20:35:19	dfs.DataNode$DataXceiver	Receiving block blk_-1608999687919862906 src: /10.250.14.224:42420 dest: /10.250.14.224:50010	blk_-1608999687919862906	E5
2008-11-09 20:35:19	dfs.DataNode$PacketResponder	PacketResponder 1 for block blk_-1608999687919862906 terminating	blk_-1608999687919862906	E11
2008-11-09 20:35:19	dfs.DataNode$PacketResponder	PacketResponder 2 for block blk_-1608999687919862906 terminating	blk_-1608999687919862906	E11
2008-11-09 20:35:19	dfs.DataNode$PacketResponder	Received block blk_-1608999687919862906 of size 91178 from /10.250.10.6	blk_-1608999687919862906	E9
2008-11-09 20:35:19	dfs.DataNode$PacketResponder	Received block blk_-1608999687919862906 of size 91178 from /10.250.19.102	blk_-1608999687919862906	E9
2008-11-09 20:35:19	dfs.DataNode$PacketResponder	PacketResponder 0 for block blk_-1608999687919862906 terminating	blk_-1608999687919862906	E11
2008-11-09 20:35:19	dfs.DataNode$PacketResponder	Received block blk_-1608999687919862906 of size 91178 from /10.250.14.224	blk_-1608999687919862906	E9
2008-11-09 20:35:19	dfs.FSNamesystem	BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.250.10.6:50010 is added to blk_-1608999687919862906 size 91178	blk_-1608999687919862906	E26
2008-11-09 20:35:19	dfs.FSNamesystem	BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.111.209:50010 is added to blk_-1608999687919862906 size 91178	blk_-1608999687919862906	E26
2008-11-09 20:35:19	dfs.FSNamesystem	BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.250.14.224:50010 is added to blk_-1608999687919862906 size 91178	blk_-1608999687919862906	E26
2008-11-09 20:35:20	dfs.DataNode$DataXceiver	Receiving block blk_7503483334202473044 src: /10.251.215.16:55695 dest: /10.251.215.16:50010	blk_7503483334202473044	E5
2008-11-09 20:35:20	dfs.DataNode$DataXceiver	Receiving block blk_7503483334202473044 src: /10.250.19.102:34232 dest: /10.250.19.102:50010	blk_7503483334202473044	E5
2008-11-09 20:35:20	dfs.FSNamesystem	BLOCK* NameSystem.allocateBlock: /mnt/hadoop/mapred/system/job_200811092030_0001/job.split. blk_7503483334202473044	blk_7503483334202473044	E22
2008-11-09 20:35:21	dfs.DataNode$DataXceiver	Received block blk_-1608999687919862906 src: /10.251.215.16:52002 dest: /10.251.215.16:50010 of size 91178	blk_-1608999687919862906	E6
2008-11-09 20:35:21	dfs.DataNode$DataXceiver	Receiving block blk_-1608999687919862906 src: /10.251.215.16:52002 dest: /10.251.215.16:50010	blk_-1608999687919862906	E5
2008-11-09 20:35:21	dfs.DataNode$DataXceiver	Receiving block blk_7503483334202473044 src: /10.251.71.16:51590 dest: /10.251.71.16:50010	blk_7503483334202473044	E5
2008-11-09 20:35:21	dfs.DataNode$DataXceiver	Receiving block blk_-3544583377289625738 src: /10.250.19.102:39325 dest: /10.250.19.102:50010	blk_-3544583377289625738	E5

only showing top 20 rows

Hình 6: Kết quả sau khi gán EventId.

Tạo chuỗi Event Sequence cho từng BlockId

Để biểu diễn hành vi của block theo trình tự thời gian, hệ thống cần xây dựng chuỗi sự kiện (event trace) cho từng `BlockId`. Spark thực hiện: lọc bỏ các log không có `EventId`; gom nhóm theo `BlockId`; sử dụng `collect_list(EventId)` để tạo ra dãy sự kiện `EventSequence`.

Kết quả thu được là một DataFrame, trong đó mỗi dòng tương ứng với một block và chứa một chuỗi các sự kiện mô tả toàn bộ hoạt động của block từ khi bắt đầu đến khi kết thúc.

BlockId	EventSequence
blk_-1001138135617662562	[E22, E5, E5, E5, E11, E9, E11, E9, E26, E26, E26, E11, E9, E3, E3, E4, E3, E3, E4, E23, E23, E23, E3, E3, E4, E3, E3, E4, E21, E21, E21]
blk_-1005590426013980840	[E23, E23, E23, E21, E21, E21, E5, E5, E22, E5, E11, E9, E11, E9, E11, E9, E26, E26, E26]
blk_-1021309077469740 51	[E23, E23, E23, E21, E5, E5, E5, E22, E11, E9, E11, E9, E11, E9, E26, E26, E26, E21, E21]
blk_-1024067452279512003	[E5, E22, E5, E5, E11, E9, E11, E9, E11, E9, E26, E26, E26]
blk_-1027487181718073118	[E5, E5, E22, E5, E11, E9, E11, E9, E26, E26, E11, E9, E26]
blk_-1028418231551336995	[E4, E4, E3, E2, E23, E23, E23, E22, E5, E5, E5, E11, E9, E11, E9, E11, E9, E26, E26, E26, E21, E21, E21]
blk_-1029280765014954488	[E5, E5, E22, E5, E11, E9, E11, E9, E11, E9, E26, E26, E26]
blk_-1036632738441638662	[E5, E5, E5, E22, E11, E9, E11, E9, E11, E9, E26, E26, E26, E23, E23, E23, E21, E21, E21]
blk_-1040412692236014953	[E23, E23, E23, E21, E21, E21, E5, E5, E5, E22, E11, E9, E11, E9, E11, E9, E26, E26, E26]
blk_-1041728781629481437	[E23, E23, E23, E5, E5, E22, E5, E11, E9, E11, E9, E11, E9, E26, E26, E26, E21, E21, E21]
blk_-1043804223311329266	[E23, E23, E23, E2, E21, E21, E21, E22, E5, E5, E5, E26, E26, E26, E11, E9, E11, E9, E11, E9]
blk_-1046696409333586608	[E5, E5, E22, E5, E11, E9, E26, E11, E9, E11, E9, E26, E26, E21, E21, E21, E3, E3, E4, E3, E4, E3, E23, E23, E23]
blk_-10504368397740793	[[E23, E23, E23, E21, E21, E21, E22, E5, E5, E5, E26, E26, E26, E11, E9, E11, E9, E11, E9]
blk_-1061080714170194 86	[E5, E22, E5, E5, E11, E9, E26, E26, E11, E9, E26, E11, E9, E4, E3, E4, E23, E23, E23, E21, E21, E21]
blk_-1061729600448953014	[E4, E3, E4, E23, E23, E23, E22, E5, E5, E5, E26, E26, E11, E9, E11, E9, E26, E11, E9, E21, E21, E21]
blk_-1066159766740796836	[E23, E23, E23, E5, E22, E5, E5, E11, E9, E11, E9, E26, E26, E11, E9, E26, E21, E21, E21]
blk_-108380212683047118	[E23, E23, E23, E22, E5, E5, E5, E11, E9, E11, E9, E26, E26, E26, E11, E9, E21, E21, E21]
blk_-1089336313406296579	[E4, E3, E4, E23, E23, E23, E22, E5, E5, E5, E26, E26, E26, E11, E9, E11, E9, E11, E9, E21, E21, E21]
blk_-109386570129652225	[E5, E22, E5, E5, E11, E9, E11, E9, E11, E9, E26, E26, E26]
blk_-1096621677333075077	[E5, E5, E5, E22, E11, E9, E11, E9, E26, E26, E26, E11, E9, E23, E23, E23, E21, E21, E21]

only showing top 20 rows

Hình 7: Kết quả chuỗi EventId

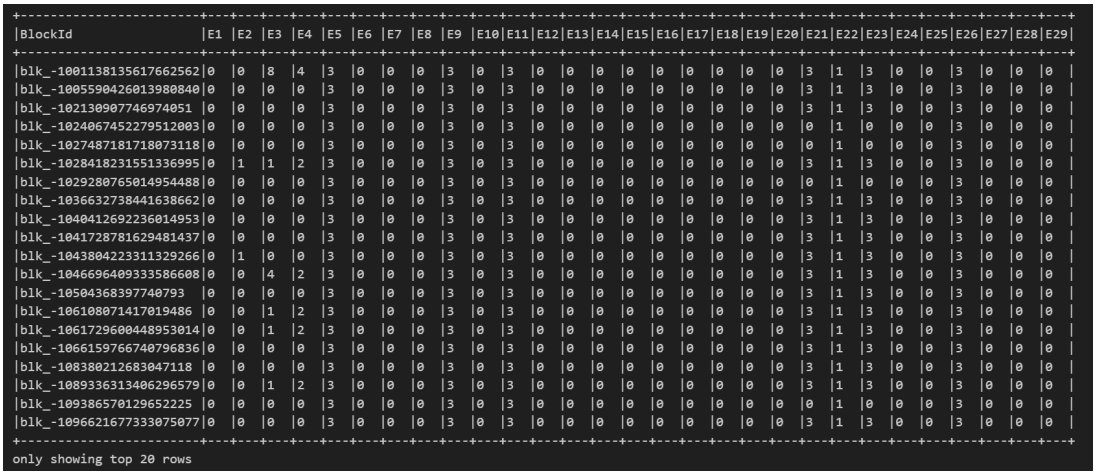
Biến đổi dãy sự kiện thành đặc trưng số (Feature Encoding)

Để tạo đặc trưng đầu vào cho mô hình học máy, hệ thống chuyển chuỗi sự kiện `EventSequence` thành ma trận đặc trưng bằng cách đếm số lần xuất hiện của từng EventId (`E1`, `E2`, ..., `E29`). Spark sử dụng hàm `aggregate()` kết hợp `filter()` để tính số lượng xuất hiện sự kiện trong từng chuỗi:

$$\text{count}(E_k) = |\{e \in EventSequence \mid e = E_k\}|.$$

Tập sự kiện được ánh xạ thành các cột số, mỗi cột biểu diễn tần suất của một EventId. Đây là dạng đặc trưng thường dùng cho các mô hình học máy truyền thống như XGBoost.

Sau khi chuyển đổi, DataFrame `df_features` chứa: `BlockId`; các cột `E1`, `E2`, ..., `E29`.



BlockId	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15	E16	E17	E18	E19	E20	E21	E22	E23	E24	E25	E26	E27	E28	E29
blk_-1001138135617662562	0	0	8	4	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-1005590426013980840	0	0	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-102130907746974051	0	0	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-1024067452279512003	0	0	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	1	0	0	0	0	3	0	0	0
blk_-1027487181718073118	0	0	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	1	0	0	0	0	3	0	0	0
blk_-1028418231551336995	0	1	1	2	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-1029280765014954488	0	0	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	1	0	0	0	0	3	0	0	0
blk_-1036632738441638662	0	0	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-1040412692236014953	0	0	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-1041728781629481437	0	0	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-1043804223311329266	0	1	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-1046696409333586608	0	0	4	2	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-10504368397740793	0	0	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-1061080714170194 86	0	0	1	2	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-1061729600448953014	0	0	1	2	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-1066159766740796836	0	0	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-108380212683047118	0	0	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-1089336313406296579	0	0	1	2	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0
blk_-109386570129652225	0	0	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	1	0	0	0	0	3	0	0	0
blk_-1096621677333075077	0	0	0	0	3	0	0	0	3	0	3	0	0	0	0	0	0	0	0	0	3	1	3	0	0	3	0	0	0

only showing top 20 rows

Hình 8: Kết quả Feature Encoding.

Kết quả tiền xử lý

Kết quả cuối cùng của giai đoạn tiền xử lý là một bảng đặc trưng có cấu trúc rõ ràng, trong đó mỗi block HDFS được biểu diễn dưới dạng một chuỗi đặc trưng số học mô tả hành vi của block trong suốt vòng đời xử lý. Các đặc trưng này được rút trích từ chuỗi sự

kiện, tần suất xuất hiện EventId, thời gian hoạt động của block và các thông tin liên quan khác. Bảng dữ liệu sau khi chuẩn hóa được lưu lại đồng thời trên HDFS và PostgreSQL, vừa phục vụ cho bước xây dựng và huấn luyện mô hình trong phần thực nghiệm, vừa hỗ trợ quá trình phân tích thống kê và trực quan hóa. Ngoài ra, bộ đặc trưng này còn được tích hợp trực tiếp vào pipeline xử lý thời gian thực, giúp mô hình dự đoán bất thường hoạt động liên tục và nhất quán với dữ liệu batch.

Quy trình này đảm bảo dữ liệu log thô được chuẩn hóa, cấu trúc hóa và tối ưu hóa theo đúng mục tiêu phân tích log của hệ thống.

3.3 Xử lý dữ liệu realtime bằng Kafka, Python và HBase

Bên cạnh pipeline xử lý batch bằng Spark, hệ thống Log Analysis còn triển khai một pipeline realtime nhằm mô phỏng quá trình log được sinh ra liên tục từ hệ thống HDFS. Pipeline realtime được xây dựng dựa trên Kafka và các module Python, kết hợp với HBase để lưu trữ và hiển thị dữ liệu tức thời trên giao diện người dùng.

Sinh dữ liệu realtime từ tập test

Để mô phỏng dữ liệu log thời gian thực, hệ thống sử dụng tập `test_stream_data.csv` (generated từ Spark batch) làm danh sách các BlockId cần mô phỏng. Hàm `generate_real_time_data()` đọc dữ liệu từ file log đã làm sạch `logs_full_labeled.csv`, lọc ra các block thuộc tập test và xây dựng một DataFrame chứa: BlockId, start_ts, end_ts, duration_sec, log_full (toàn bộ nội dung log), num_lines.

Hàm `choose_random_log_full()` chọn ngẫu nhiên một block tại mỗi lần phát sinh, tạo ra một sự kiện log realtime tương tự như khi log thực tế được sinh ra từ HDFS.

Kiểm tra trạng thái Kafka và khởi chạy Producer/Consumer

Trước khi gửi dữ liệu, chương trình gọi hàm `wait_for_kafka()` để kiểm tra Kafka hoạt động. Khi Kafka sẵn sàng, hai luồng (thread) được khởi tạo:

- Producer thread gửi lần lượt 5 bản ghi log ngẫu nhiên lên topic `log_stream_topic`.
- Consumer thread chạy nền (daemon) để đọc liên tục các bản ghi mới.

Cả hai luồng được quản lý bằng đối tượng `STOP Event` và tín hiệu `SIGINT/SIGTERM` để đảm bảo hệ thống có thể dừng an toàn khi người dùng Ctrl+C.

Gửi dữ liệu log realtime lên Kafka

Kafka Producer được cấu hình với cơ chế: `acks='all'` để đảm bảo không mất message, `retries=5` để Retry khi Kafka bận, `value_serializer` sử dụng JSON encoder.

Mỗi message được gửi lên Kafka chứa toàn bộ thông tin của block dưới dạng `dict`: Log_full, Thời gian hoạt động block, Số dòng log, BlockId, Các thông tin thống kê cơ bản. Mỗi lần gửi cách nhau 10 giây để mô phỏng tốc độ sinh log thực tế.

Consumer đọc message và thực hiện transformation

Kafka Consumer nhận message dưới dạng chuỗi JSON. Tại mỗi lần nhận, Consumer thực hiện:

1. Giải mã JSON và chuyển thành đối tượng Python;
2. Gọi hàm `transformation()` để phân tích nội dung log;
3. Chuyển `log_full` thành danh sách dòng log và tách các trường bằng regex: timestamp, level, component, message, BlockId.
4. Gán `EventId` theo bảng template `HDFS.log_templates.csv`;
5. Gom nhóm chuỗi sự kiện theo BlockId để tạo `EventSequence`;
6. Đếm tần suất 29 sự kiện E1 – E29 tạo thành vector đặc trưng;
7. Gọi mô hình `predict_block()` để dự đoán nhãn bất thường.

Kết quả trả về của hàm `transformation()` là một dict chứa: thông tin block, danh sách đặc trưng, nội dung log đầy đủ, nhãn dự đoán bất thường.

Ghi dữ liệu realtime vào HBase

Sau khi dự đoán nhãn, Consumer gọi hàm `insert_dataHbase()` để lưu thông tin vào bảng HBase `log_stream_data`.

Mỗi block được lưu thành một bản ghi có khóa chính là `BlockId`. Các trường được lưu dưới cột `info:` như: start_ts, end_ts, duration_sec, log_full, num_lines, features (vector E1..E29), prediction (kết quả dự đoán), ts_ms (timestamp hệ thống).

Việc lưu realtime vào HBase giúp giao diện Flask có thể truy vấn trực tiếp và hiển thị log mới nhất ngay khi Consumer xử lý xong message từ Kafka.

3.4 Huấn luyện mô hình LightGBM

Sau khi hoàn tất bước tiền xử lý và xây dựng bộ đặc trưng từ log HDFS, hệ thống tiến hành huấn luyện mô hình học máy để dự đoán khả năng bất thường của mỗi block. Trong thực nghiệm, mô hình `LightGBM` được lựa chọn thay vì LightBGM cho pipeline cuối cùng do tốc độ huấn luyện nhanh hơn và khả năng xử lý tốt dữ liệu có độ chênh lệch lớn giữa các lớp (class imbalance).

Chia dữ liệu theo tỉ lệ và giữ nguyên phân bố nhãn

Dữ liệu sau tiền xử lý được chia thành ba tập riêng biệt gồm *train*, *validation* và *test*. Việc chia dữ liệu được thực hiện với tham số `stratify`, nhằm đảm bảo tỉ lệ giữa hai nhãn — block bình thường và block bất thường — được giữ nguyên trong tất cả các tập con. Trước tiên, toàn bộ dữ liệu được tách thành một tập huấn luyện chiếm 60% tổng dữ liệu

và một tập tạm thời chiếm 40%. Phần dữ liệu tạm thời này được tiếp tục chia đều thành hai nửa bằng nhau để tạo thành tập validation và tập test, mỗi tập chiếm 20% tổng dữ liệu. Cách chia này giúp mô hình được huấn luyện ổn định trên tập train, trong khi tập validation đóng vai trò điều chỉnh quá trình học và xác định thời điểm dừng sớm, còn tập test được giữ lại để đánh giá khách quan hiệu năng sau khi huấn luyện hoàn tất.

Trong bộ dữ liệu log HDFS, sự mất cân bằng nhãn thể hiện rõ khi số lượng block bất thường (`label = 1`) xuất hiện thấp hơn rất nhiều so với block bình thường, với tỉ lệ xấp xỉ 1:33. Để giảm ảnh hưởng của hiện tượng này và giúp mô hình học tốt hơn các trường hợp hiếm, tham số `scale_pos_weight` được thiết lập tương ứng với tỉ lệ mất cân bằng. Việc tăng trọng số cho lớp bất thường khiến mô hình chú trọng hơn đến các mẫu có nhãn hiếm, từ đó cải thiện khả năng phát hiện bất thường trong log HDFS.

Cấu hình và huấn luyện mô hình

Mô hình LightGBM được cấu hình với số lượng cây lớn (`n_estimators = 3000`) kết hợp với `early stopping`, cho phép mô hình dừng sớm nếu tập validation không cải thiện trong 100 vòng lặp. Bộ siêu tham số được lựa chọn cân bằng giữa độ phức tạp và khả năng tổng quát:

- `num_leaves = 63`, `max_depth = 6`: giới hạn độ sâu cây để tránh overfitting;
- `subsample = 0.8`, `colsample_bytree = 0.8`: tăng khả năng khái quát;
- `reg_lambda = 1.0`: regularization L2;
- `scale_pos_weight = 33`: xử lý mất cân bằng dữ liệu.

Mô hình được huấn luyện trên tập train và được đánh giá chủ yếu dựa trên hai chỉ số quan trọng. Chỉ số đầu tiên là AUC (Area Under ROC Curve), phản ánh khả năng phân tách giữa hai lớp và là một thước đo phổ biến trong các bài toán phân loại nhị phân. Chỉ số thứ hai là Average Precision (AP), một thước đo đặc biệt phù hợp trong bối cảnh dữ liệu mất cân bằng, khi lớp bất thường chiếm tỉ lệ rất nhỏ so với lớp bình thường. Hai chỉ số này giúp đánh giá một cách toàn diện cả năng lực nhận diện đúng các mẫu dương tính lẫn khả năng hạn chế nhầm lẫn giữa hai lớp.

Trong quá trình huấn luyện, mô hình sử dụng cơ chế `early_stopping` nhằm xác định thời điểm dừng tối ưu. Khi chất lượng dự đoán trên tập validation không còn được cải thiện sau một số vòng lặp nhất định, quá trình huấn luyện sẽ tự động dừng lại để tránh hiện tượng overfitting, đồng thời giữ lại mô hình ở trạng thái tốt nhất.

3.5 Đánh giá mô hình

Hiệu năng của mô hình LightGBM được đánh giá trên cả tập *validation* và *test*. Các bảng dưới đây tổng hợp các chỉ số quan trọng thu được trong quá trình thực nghiệm, bao gồm ROC-AUC, PR-AUC, ngưỡng phân lớp tối ưu, F1-score, precision và recall.

Kết quả trên tập validation

Chỉ số	Giá trị
ROC-AUC	0.9999543
PR-AUC	0.9980548
Ngưỡng dự đoán tối ưu	0.83
F1-score tại ngưỡng tối ưu	0.99423
Precision	0.99028
Recall	0.99822

Bảng 1: Kết quả đánh giá mô hình trên tập validation

Kết quả trên tập test

Chỉ số	Giá trị
ROC-AUC	0.9999976
PR-AUC	0.9999125

Bảng 2: ROC-AUC và PR-AUC trên tập test

Confusion Matrix trên tập test

	Dự đoán 0	Dự đoán 1
Thực tế 0	111637	8
Thực tế 1	8	3360

Bảng 3: Confusion matrix trên tập test

Báo cáo phân loại trên tập test

Lớp	Precision	Recall	F1-score	Support
0	0.9999	0.9999	0.9999	111645
1	0.9976	0.9976	0.9976	3368
Accuracy		0.9999		
Macro avg	0.9988	0.9988	0.9988	115013
Weighted avg	0.9999	0.9999	0.9999	115013

Bảng 4: Kết quả phân loại chi tiết trên tập test

Nhận xét

Các kết quả cho thấy mô hình đạt hiệu năng rất cao và ổn định trên cả tập validation lẫn test. Chỉ số ROC-AUC gần tuyệt đối và PR-AUC vượt 0.999 trên tập test chứng minh mô hình phân tách hai lớp hiệu quả ngay cả khi dữ liệu bị mất cân bằng mạnh. Confusion matrix cho thấy số lượng dự đoán sai rất nhỏ (chỉ 16 mẫu trên hơn 115 000 mẫu). Điều

này xác nhận rằng LightGBM là lựa chọn phù hợp cho bài toán phát hiện bất thường trong log HDFS.

3.6 Đánh giá kết quả thực nghiệm

Phần này trình bày kết quả thực nghiệm của hệ thống Log Analysis, bao gồm đánh giá chất lượng mô hình học máy LightGBM, hiệu quả của quá trình tiền xử lý dữ liệu, và phân tích trực quan từ dashboard được xây dựng trên Power BI. Các kết quả cho thấy pipeline xử lý dữ liệu hoạt động ổn định, mô hình đạt hiệu suất cao và có khả năng phát hiện bất thường một cách chính xác.

Hiệu năng mô hình LightGBM

Mô hình được huấn luyện với tập dữ liệu đã được chuẩn hóa từ Spark và chia theo tỉ lệ 60% cho huấn luyện, 20% cho validation và 20% cho kiểm thử, đồng thời đảm bảo giữ nguyên phân bố nhãn nhờ tham số `stratify`. Dữ liệu log HDFS thể hiện sự mất cân bằng mạnh giữa hai lớp, do đó tham số `scale_pos_weight` được điều chỉnh để mô hình chú trọng hơn vào việc nhận diện các trường hợp bất thường.

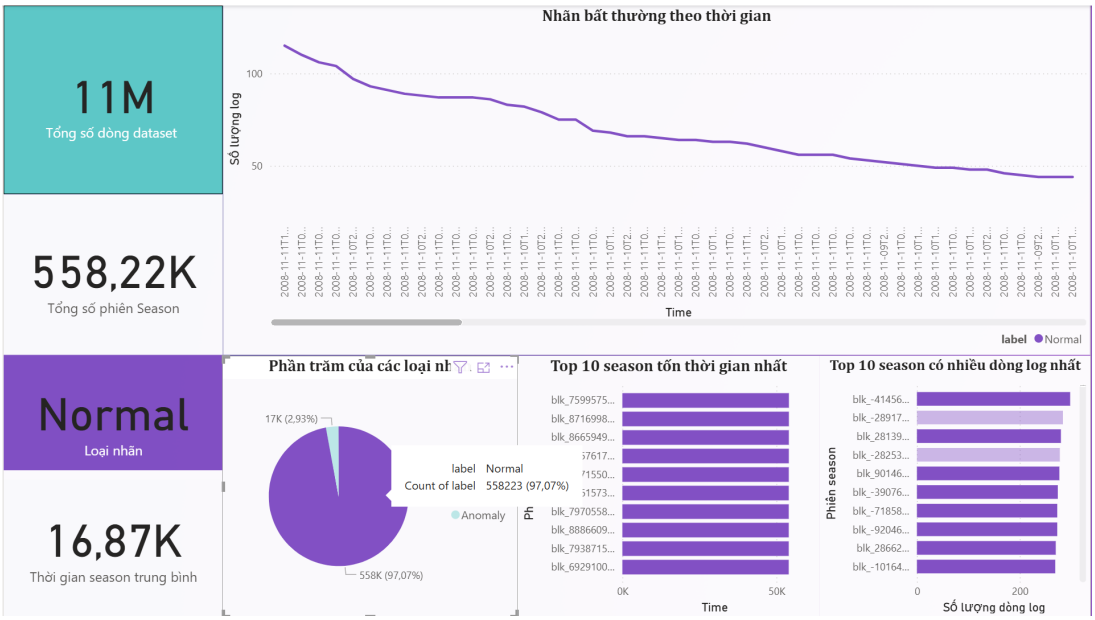
Kết quả trên tập validation cho thấy mô hình đạt hiệu năng rất cao, với ROC-AUC xấp xỉ 0.99995 và PR-AUC đạt 0.99805. Ngưỡng phân lớp tối ưu được tìm thấy vào khoảng 0.83, tại đó mô hình thu được F1-score 0.99423, cùng precision 0.99028 và recall 0.99822. Những kết quả này chứng tỏ mô hình có khả năng phân biệt hai lớp gần như tuyệt đối, đồng thời tránh bỏ sót các block bất thường.

Đánh giá trên tập kiểm thử tiếp tục cho thấy sự ổn định của mô hình khi ROC-AUC đạt 0.9999976 và PR-AUC lên tới 0.9999125. Confusion matrix thể hiện số lỗi dự đoán rất nhỏ so với quy mô tập dữ liệu — chỉ 16 sai lệch trên tổng số hơn 115 000 mẫu. Báo cáo phân loại cho thấy precision, recall và F1-score của lớp bất thường đều đạt trên 0.997, trong khi tổng độ chính xác toàn mô hình đạt 0.9999. Điều này xác nhận rằng mô hình LightGBM không chỉ phù hợp với bài toán phát hiện bất thường mà còn tổng quát hóa tốt trên dữ liệu chưa từng xuất hiện trong huấn luyện.

Đánh giá thông qua trực quan hóa dữ liệu

Dashboard tổng hợp trên Power BI cung cấp cái nhìn trực quan về đặc tính của bộ dữ liệu và sự phân bố các nhãn. Kết quả cho thấy hệ thống ghi nhận tổng cộng khoảng 11 triệu dòng log, được gom lại thành hơn 575 nghìn phiên Season tương ứng với các block HDFS. Mặc dù lớp bất thường chỉ chiếm khoảng 2.93% tổng số phiên, số lượng này vẫn đủ lớn để mô hình học được các mẫu hành vi khác biệt.

Biểu đồ theo thời gian phản ánh xu hướng số lượng bất thường có xu hướng giảm dần, cho thấy sự ổn định tương đối của hệ thống. Các biểu đồ top Season theo thời gian xử lý hoặc số lượng dòng log cho thấy những block có thời gian hoạt động dài hoặc log quá nhiều thường liên quan đến sự cố hoặc tình huống bất thường trong quá trình xử lý dữ liệu. Đây là những thông tin quan trọng giúp kiểm chứng khả năng nhận diện của mô hình và hỗ trợ công tác phân tích nguyên nhân gốc (root cause analysis).



Hình 9: Dashboard.

Kết luận về kết quả thực nghiệm

Từ các kết quả thu được, có thể thấy pipeline xử lý dữ liệu được triển khai hoạt động trơn tru và mô hình LightGBM đạt hiệu năng vượt trội trên cả tập validation và tập test. Khả năng phân tách lớp gần như tuyệt đối, số lỗi phân loại rất thấp cùng với độ chính xác cao đồng nhất trên các tập dữ liệu chứng minh tính hiệu quả của thuật toán trong bài toán phát hiện bất thường từ log HDFS. Đồng thời, các biểu đồ trực quan từ Power BI cũng củng cố độ tin cậy của mô hình khi phản ánh đúng những xu hướng bất thường trong dữ liệu. Kết hợp lại, các kết quả thực nghiệm cho thấy hệ thống Log Analysis đáp ứng tốt yêu cầu phát hiện bất thường trong môi trường dữ liệu lớn và có tiềm năng ứng dụng trong các hệ thống giám sát thực tế.

4 Tổng kết quá trình thực nghiệm

Kết quả thực nghiệm đã chứng minh rằng hệ thống Log Analysis được xây dựng có khả năng xử lý hiệu quả dữ liệu log HDFS với quy mô lớn, đồng thời đáp ứng được yêu cầu phân tích theo cả hai hướng batch và realtime. Ở lớp xử lý lô, việc sử dụng Apache Spark cho phép trích xuất đặc trưng từ hơn 11 triệu dòng log một cách ổn định và nhanh chóng, bao gồm việc chuẩn hóa dữ liệu, ánh xạ EventId và xây dựng bảng đặc trưng phục vụ mô hình học máy. Spark cho thấy khả năng mở rộng tốt và tận dụng tài nguyên tính toán phân tán, giúp quy trình tiền xử lý đạt hiệu năng cao ngay cả khi dữ liệu có kích thước lớn và đa dạng.

Ở hướng xử lý thời gian thực, pipeline dựa trên Kafka và HBase hoạt động ổn định và phản hồi nhanh. Kafka đảm nhiệm tốt vai trò tiếp nhận và truyền tải log giả lập realtime, trong khi HBase xử lý truy cập đọc/ghi tức thời và lưu trữ dữ liệu streaming phục vụ giao diện người dùng. Sự kết hợp này giúp hệ thống không chỉ phân tích log lịch sử mà còn giám sát được các block mới phát sinh theo thời gian thực.

Mô hình LightGBM là thành phần quan trọng trong quá trình phát hiện bất thường. Thông qua bộ đặc trưng được xây dựng từ chuỗi sự kiện và tần suất EventId, mô hình đạt hiệu năng rất cao, với ROC-AUC và PR-AUC gần như tuyệt đối trên cả tập validation và tập test. Số lượng dự đoán sai cực kỳ thấp, cho thấy khả năng tổng quát hóa tốt và độ tin cậy cao của mô hình ngay cả trong điều kiện dữ liệu mất cân bằng mạnh. Điều này chứng minh rằng cách tiếp cận dựa trên boosting cây quyết định phù hợp với đặc trưng của dữ liệu log HDFS.

Trực quan hóa dữ liệu bằng Power BI cũng cung cấp cái nhìn toàn diện về hành vi của hệ thống qua thời gian, bao gồm phân bố nhãn, xu hướng bất thường và các block có thời gian xử lý hoặc số dòng log cao nhất. Những biểu đồ này góp phần xác nhận tính hợp lệ của pipeline và hỗ trợ thêm trong việc phân tích nguyên nhân gốc của các bất thường quan sát được.

Tổng thể, quá trình thực nghiệm cho thấy kiến trúc Lambda được triển khai hoạt động hiệu quả, kết hợp hài hòa giữa xử lý batch, xử lý luồng và mô hình học máy. Hệ thống đáp ứng tốt yêu cầu phát hiện bất thường từ log HDFS, đồng thời có thể mở rộng linh hoạt để ứng dụng trong các hệ thống giám sát phân tán quy mô lớn. Đây là nền tảng vững chắc cho các hướng phát triển tiếp theo, bao gồm tối ưu hóa mô hình, mở rộng tập dữ liệu realtime, triển khai trong môi trường thực tế và xây dựng thêm các module phân tích chuyên sâu.

Tài liệu

[1] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T.-Y. Liu, *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*, Advances in Neural Information Processing Systems (NeurIPS), 2017.

[2] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*, Manning Publications, 2014.

[3] Wikipedia, "Big data," *Wikipedia, The Free Encyclopedia*

[4] Gartner IT Glossary, "Big Data," Gartner Inc., 2012.