TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

VIỆN TRÍ TUỆ NHÂN TẠO

***

BÁO CÁO PROJECT
KỸ THUẬT & CÔNG NGHỆ DỮ LIỆU LỚN

Chủ đề: Smartphone price prediction in Bigdata

Nhóm 2:

1. Lê Vũ Hiếu
2. Đàm Lê Minh Quân
3. Nguyễn Hoàng Tú

Hà Nội, 05/2025

Mục lục

1 Giới thiệu

1.1 Đặt vấn đề

Thị trường điện thoại thông minh ngày nay vô cùng sôi động và cạnh tranh, với hàng trăm mẫu mã mới được ra mắt mỗi năm cùng sự biến động giá liên tục. Việc nắm bắt và dự đoán giá điện thoại trở thành một nhu cầu thiết yếu không chỉ đối với người tiêu dùng muốn đưa ra quyết định mua sắm thông minh, mà còn quan trọng đối với các nhà bán lẻ trong việc xây dựng chiến lược giá cả cạnh tranh, và các nhà sản xuất trong việc định vị sản phẩm trên thị trường.

Tuy nhiên, việc dự đoán giá điện thoại là một bài toán phức tạp, chịu ảnh hưởng bởi nhiều yếu tố đa dạng như: thông số kỹ thuật (vi xử lý, RAM, bộ nhớ, camera, pin), thương hiệu, thời điểm ra mắt, các chương trình khuyến mãi, xu hướng thị trường, và thậm chí là các yếu tố kinh tế vĩ mô. Hơn nữa, lượng dữ liệu liên quan đến giá cả và thông số điện thoại là rất lớn, thay đổi nhanh chóng và đến từ nhiều nguồn khác nhau. Điều này đòi hỏi một hệ thống không chỉ có khả năng phân tích dữ liệu lịch sử để tìm ra các quy luật tiềm ẩn mà còn phải xử lý được luồng thông tin mới cập nhật gần như tức thời để đưa ra những dự đoán chính xác và kịp thời. Các phương pháp truyền thống thường gặp khó khăn khi phải đối mặt đồng thời với cả hai yêu cầu về xử lý dữ liệu lớn theo lô (batch processing) và xử lý dữ liệu theo luồng thời gian thực (real-time stream processing).

Chính vì vậy, việc xây dựng một hệ thống dự đoán giá điện thoại có khả năng tích hợp và xử lý hiệu quả cả hai loại hình dữ liệu này là một thách thức công nghệ đáng kể nhưng mang lại giá trị thực tiễn cao.

1.2 Mục tiêu dự án

Dự án này được thực hiện với mục tiêu chính là nghiên cứu, thiết kế và triển khai một hệ thống có khả năng dự đoán giá điện thoại thông minh dựa trên kiến trúc Lambda. Các mục tiêu cụ thể bao gồm:

- Xây dựng luồng xử lý dữ liệu theo lô (Batch Layer): Có khả năng xử lý khối lượng lớn dữ liệu lịch sử về điện thoại (thông số kỹ thuật, giá cả) để phục vụ việc huấn luyện mô hình học máy và tạo ra các "batch views" tổng hợp.
- Xây dựng luồng xử lý dữ liệu thời gian thực (Speed Layer): Có khả năng tiếp nhận và xử lý các thông tin mới về điện thoại (ví dụ: giá mới cập nhật, sản phẩm mới ra mắt) một cách nhanh chóng để đưa ra dự đoán gần như tức thời.
- Huấn luyện và triển khai mô hình Machine Learning: Phát triển một mô hình học máy (cụ thể là XGBoost) có khả năng dự đoán giá điện thoại dựa trên các đặc trưng kỹ thuật và thông tin liên quan.
- Xây dựng tầng phục vụ (Serving Layer): Cung cấp kết quả dự đoán từ cả Batch Layer và Speed Layer cho người dùng cuối thông qua một giao diện ứng dụng web đơn giản.
- Ứng dụng các công nghệ Big Data: Sử dụng các công cụ và nền tảng phổ biến trong hệ sinh thái Big Data như Apache Kafka, Apache Spark, Apache HBase, HDFS và MongoDB để xây dựng hệ thống. Đóng gói và điều phối hệ thống: Sử dụng Docker và Docker Compose để dễ dàng triển khai và quản lý các thành phần của hệ thống.

1.3 Phạm vi dự án

Trong khuôn khổ của dự án này, các hoạt động sẽ tập trung vào:

Nguồn dữ liệu: Sử dụng dữ liệu mô phỏng (từ file CSV mobiles_dataset_2025.csv và stream_data.csv) để phục vụ cho việc phát triển và kiểm thử các luồng xử lý. Bao gồm một kịch bản (run_scraper.py) để thu thập dữ liệu thông số kỹ thuật điện thoại từ trang web gsmarena.com làm cơ sở cho việc tạo tập dữ liệu huấn luyện.

- Mô hình dự đoán: Tập trung vào việc sử dụng mô hình XGBoost cho bài toán hồi quy dự đoán giá.
- Kiến trúc hệ thống: Triển khai kiến trúc Lambda với các thành phần cốt lõi.
- Giao diện người dùng: Xây dựng một ứng dụng web Flask đơn giản để hiển thị kết quả dự đoán mới nhất từ Speed Layer.
- Công nghệ: Các công nghệ chính được sử dụng bao gồm Python, Apache Kafka, Apache Spark, Apache HBase, HDFS, MongoDB, Flask, Docker và Docker Compose.
- Giới hạn: Dự án không tập trung sâu vào việc tối ưu hóa từng mili giây độ trễ hay xử lý hàng terabyte dữ liệu trong môi trường production thực tế, mà chủ yếu nhằm mục đích minh họa khả năng và cách thức hoạt động của kiến trúc Lambda và các công nghệ liên quan trong bài toán dự đoán giá. Các khía cạnh như bảo mật nâng cao, giao diện người dùng phức tạp, hay các thuật toán Machine Learning tiên tiến khác nằm ngoài phạm vi của dự án này.

1.4 Thông tin chung

Thành viên nhóm:

Thông tin	**Công việc chính**
Lê Vũ Hiếu K68 - AI1 23020365	**Chuẩn bị Dữ liệu & Producer:** • Phát triển scraper (run_scraper.py), làm sạch dữ liệu, tạo file CSV (mobiles_dataset_2025.csv, stream_data.csv). • Viết producer.py để gửi dữ liệu mô phỏng vào Kafka. **Phát triển Serving Layer (Flask App):** • Xây dựng ứng dụng Flask (app.py) với logic truy vấn HBase/MongoDB. • Thiết kế giao diện HTML (index.html) và CSS/JS cơ bản.

	Viết báo cáo, làm slide: • Giới thiệu, Thu thập dữ liệu, Flask App, Kết quả hoạt động (ứng dụng).
Nguyễn Hoàng Tú K68 - AI2 23020428	**Xây dựng & Huấn luyện Mô hình ML (XGBoost):** • Nghiên cứu XGBoost, train model (tiền xử lý Spark, huấn luyện, đánh giá, lưu model .pkl). **Phát triển Speed Layer:** • Viết stream pipeline (đọc từ Kafka, xử lý nhanh, áp dụng model, ghi vào HBase). **Viết báo cáo, làm slide:** • Cơ sở lý thuyết (XGBoost, Lambda), Huấn luyện mô hình, Speed Layer, Hướng phát triển (ML)
Đàm Lê Minh Quân K68 - AI2 24020416	**Triển khai Hạ tầng Hệ thống (Docker & Docker Compose):** • Viết Dockerfiles và cấu hình docker-compose.yml cho các service (Zookeeper, Kafka, Hadoop, HBase, Spark, Apps). • Viết script hỗ trợ triển khai. **Phát triển Batch Layer:** • Viết batch pipeline (Kafka → HDFS). • Viết spark batch job (HDFS → XGBoost → MongoDB). **Viết báo cáo, làm slide:** • Cơ sở lý thuyết (Công nghệ Big Data), Batch Layer, Triển khai hệ thống (Docker), Đánh giá.

Giáo viên hướng dẫn:

TS. Trần Hồng Việt

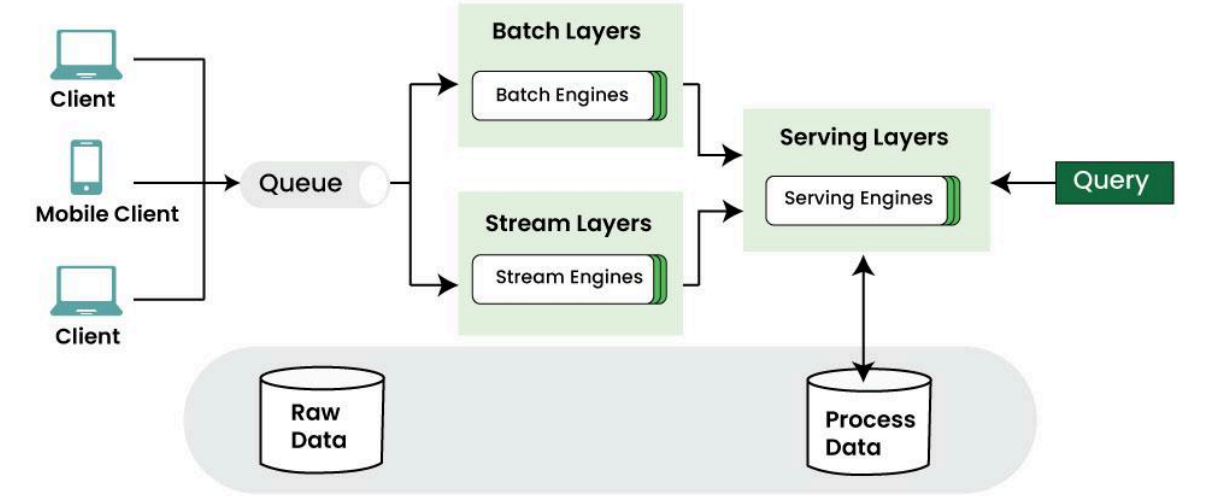
ThS. Ngô Minh Hương

---

2 Cơ sở lý thuyết

Để giải quyết những thách thức đã đề ra, cần xây dựng giải pháp dựa trên một nền tảng lý thuyết đa diện. Kiến trúc Lambda được nhóm lựa chọn làm trụ cột, cho phép hệ thống vừa xử lý khối lượng lớn dữ liệu lịch sử nhằm nhận diện các xu hướng giá dài hạn, vừa cập nhật nhanh chóng những biến động thị trường tức thời, từ đó cung cấp những dự đoán giá vừa chính xác vừa phù hợp với thời điểm hiện tại. Trọng tâm của khả năng dự đoán này là mô hình học máy XGBoost, với năng lực nắm bắt hiệu quả mối quan hệ phi tuyến giữa vô số đặc tính sản phẩm và giá cả thị trường. Việc hiện thực hóa kiến trúc Lambda và vận hành các mô hình học máy đòi hỏi một hệ sinh thái công nghệ xử lý dữ liệu lớn và thời gian thực bao gồm Apache Spark cho năng lực tính toán phân tán, Apache Kafka cho quản lý luồng dữ liệu, cùng với HBase và MongoDB cho các giải pháp lưu trữ và truy xuất dữ liệu hiệu năng cao, phù hợp với từng lớp của kiến trúc. Những lựa chọn này, được củng cố thông qua việc tham khảo và phân tích các công trình nghiên cứu liên quan trong lĩnh vực dự đoán giá và hệ thống dữ liệu, tạo nên một phương pháp luận khoa học chặt chẽ, làm tiền đề cho việc thiết kế và triển khai chi tiết hệ thống.

2.1 Kiến trúc Lambda



Layer of Lambda Architecture

Kiến trúc Lambda là kiến trúc xử lý dữ liệu được thiết kế để xử lý lượng dữ liệu lớn bằng cách tận dụng cả phương pháp xử lý hàng loạt (batch) và xử lý luồng (stream). Kiến trúc Lambda cân bằng độ trễ, thông lượng và khả năng chịu lỗi bằng cách áp dụng xử lý hàng loạt để cung cấp chế độ xem toàn diện và chính xác cho dữ liệu, đồng thời áp dụng xử lý luồng thời gian thực để cung cấp dữ liệu thời gian thực.

Hai dữ liệu batch và stream có thể được kết hợp trước khi dùng. Kiến trúc Lambda được phổ biến do sự phát triển của dữ liệu lớn, nhu cầu phân tích thời gian thực và nỗ lực giảm thiểu độ trễ của xử lý map-reduce.

Kiến trúc Lambda bao gồm ba lớp: lớp xử lý theo lô (batch), lớp xử lý tốc độ (speed) và lớp phục vụ (serving) để phản hồi các truy vấn. Các lớp xử lý lấy dữ liệu từ một bản sao của tập dữ liệu được đổ vào hệ thống.

- **Lớp xử lý dữ liệu batch:** Lớp xử lý batch tính toán trước kết quả bằng cách sử dụng hệ thống xử lý phân tán có thể xử lý lượng dữ liệu rất lớn, hướng đến độ chính xác bằng cách xử lý tất cả dữ liệu. Điều này có nghĩa là ta có thể sửa bất kỳ lỗi nào bằng cách tính toán lại dựa trên tập dữ liệu hoàn chỉnh. Kết quả thường được lưu trữ trong cơ sở dữ liệu chỉ đọc, và thay thế hoàn toàn các dữ liệu tính toán trước đó.

- **Lớp xử lý dữ liệu speed (hoặc còn gọi là real time):** Lớp Speed xử lý các luồng dữ liệu trong thời gian thực và không có yêu cầu sửa chữa hoặc hoàn thiện. Lớp này hy sinh thông lượng vì nó nhằm mục đích giảm thiểu độ trễ bằng cách xử lý dữ liệu trong thời gian thực. Về cơ bản, lớp tốc độ chịu trách nhiệm lấp đầy 'khoảng trống' do độ trễ của lớp batch. Dữ liệu của lớp speed có thể không chính xác hoặc đầy đủ như chế độ xem cuối cùng do lớp batch tạo ra, nhưng chúng có sẵn gần như ngay lập tức sau khi nhận được dữ liệu và có thể được thay thế khi bởi dữ liệu được xử lý do lớp batch. Thường hay bị nhầm lẫn với micro-batch, lớp speed hoàn toàn xử lý dữ liệu theo từng dòng dữ liệu.
- **Lớp Serving**: Dữ liệu được xử lý từ các lớp batch và speed được lưu trữ trong lớp serving, đáp ứng các truy vấn bằng cách trả về dữ liệu được tính toán trước hoặc dữ liệu từ kho dữ liệu (data warehouse).

2.2 Mô hình XGBoost

XGBoost là một thuật toán học máy thuộc nhóm học tăng cường (ensemble learning) dựa trên kỹ thuật Gradient Boosting. Cốt lõi của Gradient Boosting là xây dựng một mô hình mạnh mẽ từ sự kết hợp tuần tự của nhiều mô hình yếu (thường là cây quyết định - decision trees). Thay vì huấn luyện các mô hình yếu một cách độc lập, Gradient Boosting huấn luyện mỗi mô hình mới để sửa lỗi của các mô hình đã được xây dựng trước đó.

Nguyên lý Gradient Boosting:

- **Mô hình yếu (Weak Learner):** XGBoost thường sử dụng các cây CART (Classification and Regression Trees) làm mô hình yếu. Mỗi cây cố gắng học một phần nhỏ của dữ liệu.
- **Hàm mất mát (Loss Function):** Để đánh giá mức độ "sai" của mô hình, một hàm mất mát được định nghĩa (trong trường hợp này là Mean Squared Error cho bài toán hồi quy). Mục tiêu là tìm cách cực tiểu hóa hàm mất mát này.
- **Hướng Gradient:** Trong mỗi bước boosting, thuật toán tính toán gradient (đạo hàm) của hàm mất mát theo dự đoán của mô hình

hiện tại. Gradient này chỉ ra hướng mà dự đoán cần được điều chỉnh để giảm thiểu lỗi.

- **Huấn luyện tuần tự:** Một cây quyết định mới được huấn luyện để dự đoán phần dư (residuals) hoặc một đại lượng liên quan đến gradient âm của hàm mất mát từ bước trước. Nói cách khác, cây mới tập trung vào những điểm dữ liệu mà mô hình hiện tại đang dự đoán kém.
- **Kết hợp mô hình:** Dự đoán cuối cùng được tổng hợp từ dự đoán của tất cả các cây yếu, thường là qua một phép cộng có trọng số.

Các cải tiến chính của XGBoost so với Gradient Boosting truyền thống:

- **Regularization (Chuẩn hóa):**
    - L1 (Lasso) và L2 (Ridge) Regularization: XGBoost thêm các thành phần chuẩn hóa vào hàm mục tiêu (objective function) trong quá trình xây dựng cây. Thành phần L2 giúp ngăn chặn trọng số của các lá cây trở nên quá lớn, trong khi L1 có thể giúp loại bỏ các đặc trưng ít quan trọng (feature selection) bằng cách đưa trọng số của chúng về 0. Điều này giúp mô hình có khả năng tổng quát hóa tốt hơn và giảm thiểu overfitting.
    - Hàm mục tiêu của XGBoost có thể biểu diễn tổng quát là: $Obj(\Theta) = L(\Theta) + \Omega(\Theta)$, trong đó $L(\Theta)$ là hàm mất mát (ví dụ: training loss) và $\Omega(\Theta)$ là thành phần chuẩn hóa (regularization term). XGBoost tối ưu hóa hàm mục tiêu này.
- **Xấp xỉ Taylor bậc hai (Second-order Taylor Expansion):** Để tối ưu hóa hàm mục tiêu, XGBoost sử dụng xấp xỉ Taylor bậc hai của hàm mất mát. Điều này cung cấp thông tin chính xác hơn về hướng và độ lớn của việc cập nhật mô hình so với việc chỉ sử dụng gradient bậc nhất (như trong Gradient Boosting truyền thống). Việc này giúp thuật toán hội tụ nhanh hơn và chính xác hơn.
- **Xử lý giá trị thiếu (Sparsity-aware Split Finding):** XGBoost có một thuật toán hiệu quả để xử lý các giá trị bị thiếu trong dữ liệu. Tại mỗi nút của cây, khi tìm điểm chia tốt nhất, thuật toán sẽ xem

xét việc đặt các mẫu có giá trị thiếu vào nhánh trái hoặc nhánh phải, sau đó chọn hướng mang lại mức giảm hàm mất mát lớn nhất. Điều này cho phép XGBoost học được cách tốt nhất để xử lý giá trị thiếu mà không cần bước imputation dữ liệu trước.

- **Kỹ thuật cắt tỉa cây (Tree Pruning):** XGBoost xây dựng cây đến một độ sâu tối đa (max_depth) và sau đó thực hiện cắt tỉa ngược (post-pruning) dựa trên giá trị gamma (ngưỡng giảm mất mát tối thiểu để thực hiện một phép chia). Nếu một phép chia không mang lại sự cải thiện đủ lớn cho hàm mất mát (sau khi đã tính cả thành phần chuẩn hóa), phép chia đó sẽ bị loại bỏ.
- **Weighted Quantile Sketch:** Để tìm điểm chia tối ưu cho các đặc trưng liên tục một cách hiệu quả, XGBoost sử dụng một thuật toán xấp xỉ dựa trên quantile (quantile sketch) để đề xuất các điểm chia tiềm năng.

Những cải tiến này không chỉ giúp XGBoost đạt được độ chính xác cao mà còn tối ưu về mặt tốc độ và khả năng mở rộng, làm cho nó trở thành một trong những mô hình học máy hiệu quả và phổ biến.

2.3 Công nghệ sử dụng

Để hiện thực hóa các chức năng và phạm vi đã đề ra, dự án tận dụng một loạt các công nghệ và nền tảng tiên tiến trong lĩnh vực xử lý dữ liệu lớn và học máy.

- Ngôn ngữ lập trình chính - Python: Được sử dụng xuyên suốt dự án cho việc phát triển scraper, các pipeline xử lý dữ liệu trong Spark, huấn luyện mô hình, xây dựng ứng dụng web Flask, và các script tiện ích. Sự phong phú của các thư viện hỗ trợ (Pandas, NumPy, Scikit-learn, HappyBase, Pymongo, Kafka-Python) là một lợi thế lớn.
- Containerization và Điều phối - Docker và Docker Compose: Đóng gói các thành phần của hệ thống (Kafka, Zookeeper, HBase, Hadoop, Spark, các ứng dụng Python) vào các container độc lập, đảm bảo tính nhất quán môi trường và đơn giản hóa việc triển khai, quản lý các dịch vụ.
- Hệ sinh thái Dữ liệu lớn (Big Data Ecosystem):

- Apache Zookeeper: Là một dịch vụ điều phối tập trung, đóng vai trò nền tảng thiết yếu cho hoạt động ổn định của các hệ thống phân tán như Kafka và HBase. Zookeeper chịu trách nhiệm quản lý cấu hình, đồng bộ hóa trạng thái, phát hiện lỗi và bầu chọn leader trong các cụm Kafka và HBase, đảm bảo tính sẵn sàng cao và khả năng chịu lỗi của chúng.
- Apache Kafka: Đóng vai trò là hệ thống message queue trung tâm, tiếp nhận và phân phối luồng dữ liệu sản phẩm đến cả Batch Layer và Speed Layer, đảm bảo tính linh hoạt và khả năng mở rộng cho việc thu thập dữ liệu.
- Apache Hadoop (HDFS): Hệ thống tệp phân tán, được sử dụng làm nơi lưu trữ chính cho dữ liệu thô và dữ liệu đã qua xử lý sơ bộ của Batch Layer, phù hợp cho việc lưu trữ và truy cập các tập dữ liệu lớn.
- Apache Spark: Framework tính toán phân tán mạnh mẽ, là trái tim của các tác vụ xử lý dữ liệu:
    - Trong Batch Layer: Được sử dụng để thực hiện các phép biến đổi dữ liệu (ETL), làm sạch, trích xuất đặc trưng và áp dụng mô hình XGBoost trên toàn bộ dữ liệu lịch sử từ HDFS.
    - Trong quá trình Huấn luyện Mô hình: train_model_spark.py sử dụng Spark (cụ thể là xgboost.spark) để huấn luyện mô hình XGBoost trên tập dữ liệu lớn, tận dụng khả năng xử lý song song.
- Apache HBase: Cơ sở dữ liệu NoSQL dạng cột, được sử dụng trong Serving Layer để lưu trữ các kết quả dự đoán từ Speed Layer. HBase cung cấp khả năng đọc/ghi với độ trễ thấp, rất quan trọng cho việc phục vụ các truy vấn thời gian thực.
- MongoDB Atlas: Dịch vụ cơ sở dữ liệu NoSQL dạng tài liệu trên cloud. Được sử dụng để lưu trữ kết quả (batch views) từ Batch Layer, mang lại sự linh hoạt trong cấu trúc dữ liệu và khả năng mở rộng.

- Học máy - XGBoost: Thuật toán được lựa chọn để xây dựng mô hình dự đoán giá. XGBoost nổi bật với hiệu suất cao, khả năng xử lý giá trị thiếu, tích hợp cơ chế chuẩn hóa (regularization) để tránh overfitting và hỗ trợ tốt việc huấn luyện song song.
- Phát triển Web và Giao tiếp - Flask: Microframework web nhẹ và linh hoạt, được sử dụng để xây dựng một ứng dụng web đơn giản hiển thị kết quả dự đoán mới nhất từ HBase, minh họa khả năng truy cập dữ liệu thời gian thực của hệ thống.

---

3 Phương pháp thiết kế

3.1 Kiến trúc tổng thể hệ thống

Dựa trên lý thuyết về kiến trúc Lambda vừa phân tích, hệ thống bao gồm các thành phần chính sau:

- Luồng Dữ liệu Đầu vào (Data Ingestion):
    - Dữ liệu thô về sản phẩm điện thoại (thông số kỹ thuật, giá ban đầu, v.v.) được thu thập từ các nguồn khác nhau. Trong dự án này, một module scraper được phát triển để thu thập dữ liệu từ trang gsmarena.com.
    - Đồng thời, một luồng dữ liệu giả lập (Lambda/producer.py sử dụng Stream_data/stream_data.csv) tách ra từ tập test được sử dụng để mô phỏng việc dữ liệu sản phẩm mới liên tục được đưa vào hệ thống.
    - Apache Kafka đóng vai trò là cổng tiếp nhận dữ liệu trung tâm. Tất cả dữ liệu mới, dù là từ scraper hay luồng giả lập, đều được đẩy vào các topic Kafka (smartphoneTopic), hoạt động dưới sự điều phối của Zookeeper. Kafka cung cấp một

buffer đáng tin cậy và khả năng phân phối dữ liệu đến các lớp xử lý khác nhau.

- Lớp Xử lý theo Lô (Batch Layer):
    - Mục tiêu: Xử lý toàn bộ tập dữ liệu lịch sử để tạo ra các "batch views" – những cái nhìn tổng thể và chính xác nhất về dự đoán giá dựa trên toàn bộ kiến thức đã có.
    - Luồng hoạt động:
        - Một consumer (Lambda/Batch_layer/HDFS_consumer.py, chạy trong service batch-processor) đọc dữ liệu từ topic Kafka (smartphoneTopic) và lưu trữ dữ liệu thô này vào HDFS.
        - Một job Apache Spark (Lambda/Batch_layer/spark_tranformation.py, chạy trong service spark-batch-job) được kích hoạt định kỳ.
        - Job Spark này đọc dữ liệu từ HDFS, thực hiện các bước tiền xử lý, làm sạch, trích xuất đặc trưng và áp dụng mô hình dự đoán giá để tính toán giá dự đoán cho từng sản phẩm.
        - Kết quả dự đoán và các thông tin liên quan (batch views) sau đó được lưu trữ vào MongoDB Atlas.
- Lớp Xử lý theo Thời gian thực (Speed Layer / Stream Layer):
    - Mục tiêu: Xử lý dữ liệu mới đến với độ trễ thấp nhất có thể, cung cấp các dự đoán giá cập nhật nhanh chóng (real-time views) và bổ sung cho batch views.
    - Luồng hoạt động:
        - Một consumer (Lambda/Stream_layer/ML_consumer.py, chạy trong service stream-processor) đọc dữ liệu sản phẩm mới trực tiếp từ topic Kafka (smartphoneTopic) ngay khi chúng xuất hiện.

        - Với mỗi bản ghi dữ liệu mới, module này ngay lập tức áp dụng logic tiền xử lý và mô hình dự đoán. Giá dự đoán cùng với thông tin sản phẩm được ghi vào Apache HBase. HBase được chọn vì khả năng ghi và truy vấn nhanh với độ trễ thấp, phù hợp cho các cập nhật thời gian thực.

- Lớp Phục vụ (Serving Layer):
    - Mục tiêu: Hợp nhất kết quả từ Batch Layer và Speed Layer để cung cấp câu trả lời cho các truy vấn của người dùng hoặc ứng dụng khác.
    - Thành phần:
        - MongoDB Atlas: Lưu trữ các batch views (kết quả toàn diện từ Batch Layer).
        - Apache HBase: Lưu trữ các real-time views (kết quả cập nhật nhanh từ Speed Layer).
        - Một ứng dụng web Flask (Lambda/real_time_web_app(Flask)/app.py) được phát triển để minh họa khả năng truy vấn. Ứng dụng này hiện tại truy vấn bản ghi mới nhất từ HBase để hiển thị giá dự đoán theo thời gian thực.
- Quy trình Huấn luyện Mô hình (Model Training - Offline Process):
    - Đây là một quy trình quan trọng, diễn ra tách biệt với luồng xử lý dữ liệu chính của kiến trúc Lambda nhưng cung cấp "trí thông minh" cho hệ thống.
    - Dữ liệu được thu thập bởi run_scraper.py và lưu thành trong hệ thống.
    - Một job Spark khác (ML_operations/train_model_spark.py) được sử dụng để tiền xử lý dữ liệu này, huấn luyện mô hình XGBoost, và lưu tệp mô hình đã huấn luyện (ML_operations/xgb_model.pkl). Tệp mô hình này sau đó được cả Batch Layer và Speed Layer sử dụng để thực hiện dự đoán.

- Việc huấn luyện lại mô hình có thể được thực hiện định kỳ khi có đủ dữ liệu mới hoặc khi hiệu suất mô hình suy giảm.

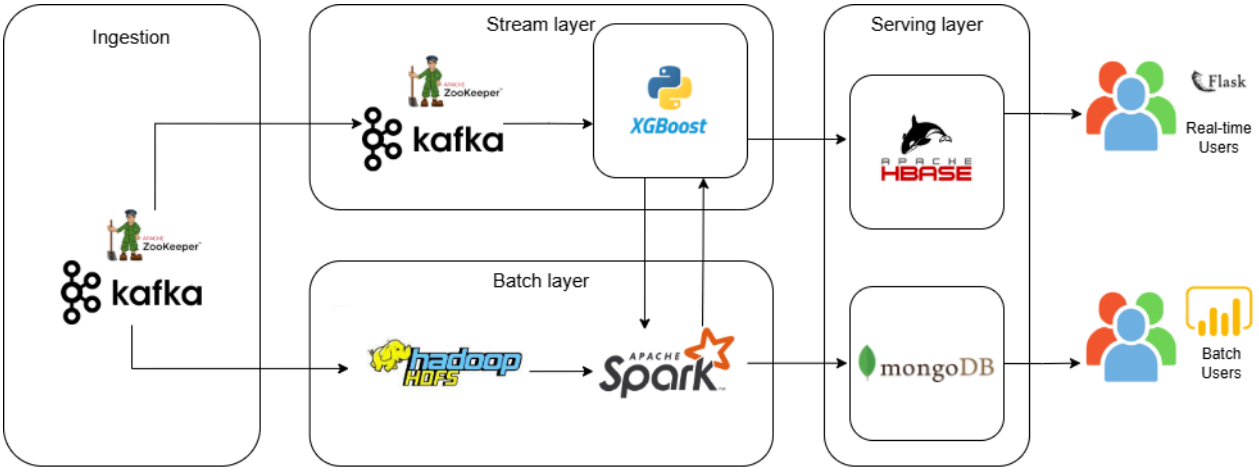
Kiến trúc này cho phép hệ thống vừa có khả năng xử lý các tập dữ liệu lớn, thực hiện các phân tích phức tạp để có được mô hình dự đoán chính xác, vừa có thể phản ứng nhanh với thông tin mới, đảm bảo tính cập nhật của các dự đoán. Các thành phần được đóng gói bằng Docker và điều phối qua Docker Compose, tạo điều kiện thuận lợi cho việc triển khai và quản lý.

Qua đó, project được nhóm triển khai theo cấu trúc file như sau:

```
.
├── .gitattributes
├── README.md
├── images/
│   ├── architecture.png
│   ├── dashboard_phone.png
│   ├── run_web_app.png
│   └── spring_boot_web_app.png
├── Main/
│   ├── create-table.sh
│   ├── docker-compose.yml
│   ├── hbase-site.xml
│   ├── spark-with-deps.Dockerfile
│   ├── start-hbase.sh
│   └── Lambda/
│       ├── ML_operations/
│       │   ├── mobiles_dataset_2025.csv
│       │   ├── run_scraper.py
│       │   ├── train_model_spark.py
│       │   └── xgb_model.pkl
│       ├── Batch_layer/
│       │   ├── HDFS_consumer.py
│       │   ├── __init__.py
│       │   ├── batch_pipeline.py
│       │   ├── put_data_hdfs.py
│       │   ├── save_data_mongodb.py
│       │   ├── spark_job_requirements.txt
│       │   └── spark_tranformation.py
│       ├── Stream_data/
│       │   ├── stream_data.csv
```

```
        │   └── stream_data.py
        ├── Stream_layer/
        │   ├── ML_consumer.py
        │   ├── __init__.py
        │   ├── insert_data_hbase.py
        │   └── stream_pipeline.py
        ├── batch_processor.Dockerfile
        ├── batch_processor_requirements.txt
        ├── create_hbase_table.py
        ├── producer.py
        ├── real_time_web_app(Flask)/
        │   ├── Dockerfile
        │   ├── app.py
        │   ├── get_Data_from_hbase.py
        │   ├── requirements.txt
        │   ├── static/
        │   │   ├── css/
        │   │   │   └── style.css
        │   │   └── js/
        │   │       └── script.js
        │   └── templates/
        │       └── index.html
        ├── stream_processor.Dockerfile
        ├── stream_processor_requirements.txt
        └── transform.py
└── bigdata_report.docx
```

Hình ảnh minh họa cho các luồng dữ liệu:



3.2 Thu thập & Tiền xử lý dữ liệu

Quá trình thu thập và tiền xử lý dữ liệu là nền tảng quan trọng, đảm bảo chất lượng đầu vào cho việc huấn luyện mô hình và hoạt động của hệ thống dự đoán.

Thu thập dữ liệu:

- Dữ liệu huấn luyện ban đầu: Nguồn dữ liệu chính cho việc huấn luyện mô hình được thu thập từ trang web gsmarena.com thông qua script Python run_scraper.py. Script này có khả năng duyệt qua các danh mục sản phẩm, trích xuất thông tin chi tiết về cấu hình, thông số kỹ thuật của từng mẫu điện thoại và lưu trữ dưới dạng tệp CSV.
- Dữ liệu đầu vào cho hệ thống (Batch & Speed Layers):
    - Để mô phỏng luồng dữ liệu sản phẩm mới liên tục đi vào hệ thống, một tệp CSV mẫu được sử dụng.
    - Script Main/Lambda/producer.py (được gọi trong cả batch_pipeline.py và stream_pipeline.py) đọc dữ liệu từ tệp CSV này và gửi dưới dạng tin nhắn JSON vào topic smartphoneTopic trên Apache Kafka. Điều này giả lập việc dữ liệu mới được cập nhật liên tục.
    - Dữ liệu từ Kafka sau đó được HDFS_consumer.py (trong batch-processor) thu thập và lưu vào HDFS dưới dạng các tệp dữ liệu thô cho Batch Layer.

Tiền xử lý dữ liệu: Quá trình tiền xử lý được thực hiện ở nhiều giai đoạn và bởi các module khác nhau, tùy thuộc vào mục đích (huấn luyện hay dự đoán) và lớp xử lý (batch hay speed):

- Trong quá trình huấn luyện mô hình (ML_operations/train_model_spark.py):
    - Làm sạch dữ liệu: Các cột chứa thông tin như RAM, dung lượng pin, kích thước màn hình thường chứa các đơn vị hoặc ký tự không cần thiết. Các hàm UDF (User Defined

Functions) trong Spark được sử dụng để chuẩn hóa và trích xuất giá trị số từ các chuỗi này (ví dụ: "8GB RAM" -> 8.0, "5000 mAh" -> 5000.0). Giá sản phẩm (ví dụ: "USD 799") cũng được làm sạch để lấy giá trị số.

- Chuyển đổi đặc trưng (Feature Transformation):
  - Mã hóa Categorical Features: Đặc trưng dạng chuỗi được chuyển đổi sang dạng số bằng cách sử dụng một bảng ánh xạ và ngược lại.
  - Các giá trị thiếu (null/NA) trong các cột quan trọng được loại bỏ.
  - Chuẩn bị dữ liệu cho mô hình: Các đặc trưng đã được xử lý sau đó được tập hợp thành một vector đặc trưng duy nhất bằng VectorAssembler của Spark MLlib, sẵn sàng cho việc huấn luyện mô hình XGBoost.

Quá trình này đảm bảo rằng dữ liệu, dù từ nguồn lịch sử hay luồng thời gian thực, đều được chuẩn hóa và biến đổi thành dạng phù hợp để mô hình học máy có thể xử lý và đưa ra dự đoán chính xác.

3.3 Xây dựng & huấn luyện mô hình

Việc xây dựng và huấn luyện mô hình dự đoán giá là một quy trình ngoại tuyến (offline process). Mô hình được lựa chọn là XGBoost do hiệu suất cao và khả năng xử lý tốt dữ liệu dạng bảng. Với dữ liệu train đã thu thập, quy trình huấn luyện được diễn ra trong như sau:

Khởi tạo Spark Session: Thiết lập môi trường Spark để thực hiện huấn luyện phân tán.

- Tiền xử lý và Trích xuất Đặc trưng (Feature Engineering):
  - Loại bỏ các hàng có giá trị thiếu ở các cột quan trọng.
  - Sử dụng pyspark.ml.feature.VectorAssembler để kết hợp các cột đặc trưng đã chọn (ví dụ: "brand\numeric", "screen_size_float", "ram_float", "battery_float") thành một cột vector đặc trưng duy nhất có tên là "features".

    - Phân chia dữ liệu: Chia tập dữ liệu đã xử lý thành hai phần: tập huấn luyện (training set) và tập kiểm thử (test set) theo một tỷ lệ nhất định (ví dụ: 80% huấn luyện, 20% kiểm thử) để đánh giá hiệu suất của mô hình sau này.
- Huấn luyện mô hình XGBoost:
    - Sử dụng xgboost.spark.XGBoostRegressor từ thư viện XGBoost dành cho Spark.
    - Cấu hình các tham số cho XGBoostRegressor (ví dụ: featuresCol="features", labelCol="label", maxDepth).
    - Gọi phương thức fit() trên tập dữ liệu huấn luyện để huấn luyện mô hình.
- Lưu trữ mô hình:
    - Sau khi huấn luyện, mô hình XGBoost (cụ thể là đối tượng booster của mô hình) được trích xuất.
    - Sử dụng thư viện pickle của Python để serialize (tuần tự hóa) đối tượng booster và lưu xuống tệp. Trong dự án này, mô hình được lưu vào ML_operations/xgb_model.pkl. Tệp .pkl này sau đó sẽ được tải bởi cả Batch Layer và Speed Layer để thực hiện dự đoán trên dữ liệu mới.
- Đánh giá mô hình: Đánh giá mô hình trên tập kiểm thử sử dụng các chỉ số như RMSE (Root Mean Squared Error), R-squared (R^2),...

Quy trình này đảm bảo rằng một mô hình dự đoán giá được huấn luyện dựa trên dữ liệu lịch sử và sẵn sàng để được triển khai, phục vụ cho việc dự đoán giá trên dữ liệu mới trong cả Batch Layer và Speed Layer. Mô hình cần được huấn luyện lại định kỳ khi có thêm dữ liệu mới hoặc khi hiệu suất của mô hình hiện tại giảm sút.

3.4 Thiết kế Batch Layer

Batch Layer (Lớp xử lý theo lô) chịu trách nhiệm xử lý toàn bộ tập dữ liệu lịch sử để tạo ra các "batch views" – những kết quả dự đoán giá

toàn diện và chính xác dựa trên tất cả dữ liệu đã biết. Lớp này hoạt động không yêu cầu độ trễ thấp và thường được chạy định kỳ.

Luồng dữ liệu và xử lý:

- Thu thập dữ liệu vào HDFS:
    - Dữ liệu sản phẩm mới được gửi đến topic smartphoneTopic trên Apache Kafka.
    - Service batch-processor (chạy batch_pipeline.py) chứa một consumer (HDFS_consumer.py) lắng nghe topic này.
    - Khi có dữ liệu mới, HDFS_consumer.py sẽ đọc và sử dụng put_data_hdfs.py để lưu trữ dữ liệu thô này vào HDFS (Hadoop Distributed File System). HDFS là nơi lưu trữ chính cho dữ liệu đầu vào của Batch Layer.
- Xử lý và Dự đoán bằng Spark:
    - Một job Apache Spark được định nghĩa trong spark_tranformation.py sẽ được kích hoạt để xử lý dữ liệu.
    - Job Spark này đọc toàn bộ dữ liệu thô từ HDFS.
    - Tiền xử lý: Áp dụng các bước làm sạch, chuẩn hóa và biến đổi đặc trưng.
    - Tải mô hình: Tải mô hình XGBoost đã được huấn luyện.
    - Dự đoán giá: Sử dụng mô hình XGBoost để dự đoán giá cho từng sản phẩm trong tập dữ liệu.
    - Tạo Batch Views: Kết quả cuối cùng bao gồm thông tin chi tiết của sản phẩm cùng với giá dự đoán.
- Lưu trữ Batch Views:
    - DataFrame chứa kết quả dự đoán từ Spark sau đó được lưu trữ vào MongoDB Atlas (cụ thể là collection smartphones trong database phone_price_pred, hoặc một collection khác dành cho dữ liệu đã biến đổi).

Đặc điểm:

- Tính toàn diện: Xử lý toàn bộ dữ liệu lịch sử.
- Độ chính xác cao: Có thời gian để thực hiện các phép tính phức tạp và sử dụng toàn bộ dữ liệu.
- Chạy định kỳ: Các job Spark trong Batch Layer thường được lên lịch để chạy định kỳ (ví dụ: hàng ngày, hàng giờ) tùy theo tần suất cập nhật dữ liệu và yêu cầu nghiệp vụ.
- Khả năng chịu lỗi: Được thiết kế để xử lý các tập dữ liệu lớn và có khả năng phục hồi sau lỗi.

Batch Layer đóng vai trò quan trọng trong việc xây dựng "master dataset" với các dự đoán giá đã được tính toán, làm cơ sở cho các phân tích sâu hơn hoặc cung cấp dữ liệu nền tảng cho Serving Layer.

3.5 Thiết kế Speed Layer

Speed Layer (Lớp xử lý theo thời gian thực), còn được gọi là Stream Layer, được thiết kế để xử lý dữ liệu mới đến với độ trễ cực thấp, cung cấp các dự đoán giá cập nhật nhanh chóng. Mục tiêu chính của lớp này là bổ sung cho Batch Layer bằng cách xử lý các dữ liệu mà Batch Layer chưa kịp cập nhật.

Luồng dữ liệu và xử lý:

- Tiếp nhận dữ liệu từ Kafka:
    - Dữ liệu sản phẩm mới được gửi đến topic smartphoneTopic trên Apache Kafka.
    - Service stream-processor (chạy stream_pipeline.py) chứa một consumer (ML_consumer.py) lắng nghe topic này.
- Xử lý và Dự đoán tức thời: Khi ML_consumer.py nhận được một bản ghi dữ liệu mới từ Kafka:
    - Tiền xử lý nhanh: Tương tự như trong Batch Layer nhưng xử lý từng bản ghi một.
    - Tải mô hình: Hàm transformation() tải mô hình XGBoost đã được huấn luyện.

    - Dự đoán giá: Áp dụng mô hình XGBoost để dự đoán giá cho bản ghi dữ liệu vừa nhận được.
    - Toàn bộ quá trình này được thiết kế để diễn ra nhanh chóng, với độ trễ thấp.
- Lưu trữ Real-time Views:
    - Kết quả dự đoán sau đó được truyền vào Apache HBase (bảng smartphone, column family info). Mỗi bản ghi được gán một row_key duy nhất (thường dựa trên timestamp) để đảm bảo thứ tự và khả năng truy xuất. HBase được chọn vì khả năng ghi và đọc nhanh, rất phù hợp cho việc lưu trữ và truy vấn các cập nhật thời gian thực.

Đặc điểm:

- Độ trễ thấp: Ưu tiên hàng đầu là tốc độ xử lý để cung cấp dự đoán gần như ngay lập tức.
- Tính cập nhật: Xử lý dữ liệu mới nhất mà Batch Layer có thể chưa xử lý.
- Khả năng mở rộng: Có thể mở rộng để xử lý lượng lớn tin nhắn từ Kafka.
- Độ chính xác có thể hy sinh một chút so với Batch Layer: Do ưu tiên tốc độ, các phép tính hoặc nguồn dữ liệu bổ sung có thể không được sử dụng như trong Batch Layer. Tuy nhiên, việc sử dụng cùng một mô hình xgb_model.pkl giúp duy trì sự nhất quán cơ bản.

Speed Layer đảm bảo rằng hệ thống có thể phản ứng nhanh với các thông tin mới nhất, cung cấp các dự đoán giá "nóng" cho người dùng hoặc các ứng dụng khác cần thông tin cập nhật tức thì.

3.6 Thiết kế Serving Layer

Serving Layer (Lớp Phục vụ) có nhiệm vụ cung cấp kết quả dự đoán giá cho người dùng cuối hoặc các ứng dụng khác. Lớp này hợp nhất hoặc cung cấp quyền truy cập vào các "views" (khung nhìn dữ liệu) đã được tính toán bởi Batch Layer và Speed Layer.

Nguồn dữ liệu cho Serving Layer:

- MongoDB Atlas (Batch Views):
    - Lưu trữ kết quả dự đoán toàn diện và chi tiết từ Batch Layer (service spark-batch-job ghi vào MongoDB).
    - Dữ liệu này phản ánh dự đoán dựa trên phân tích toàn bộ lịch sử dữ liệu, thường có độ chính xác cao nhưng có độ trễ nhất định do chu kỳ xử lý của Batch Layer.
    - Phù hợp cho các truy vấn phân tích, báo cáo tổng hợp hoặc khi cần độ chính xác tối đa trên dữ liệu lịch sử.
- Apache HBase (Real-time Views):
    - Lưu trữ các dự đoán giá mới nhất được xử lý bởi Speed Layer (service stream-processor ghi vào HBase).
    - Dữ liệu này có độ trễ rất thấp, phản ánh những thay đổi mới nhất của thị trường.
    - Phù hợp cho các truy vấn cần thông tin cập nhật tức thì, ví dụ như hiển thị giá dự đoán cho một sản phẩm vừa được quan tâm.

Truy vấn và cung cấp dữ liệu:

- Logic truy vấn:
    - Các ứng dụng hoặc người dùng có thể truy vấn trực tiếp vào HBase để lấy các dự đoán mới nhất.
    - Cũng có thể truy vấn MongoDB để lấy dữ liệu lịch sử hoặc các phân tích tổng hợp từ batch views.
- Ứng dụng minh họa (Flask Web App):
    - Dự án bao gồm một ứng dụng web đơn giản được xây dựng bằng Flask.
    - Ứng dụng này kết nối trực tiếp với HBase để lấy bản ghi dự đoán mới nhất.

    - Sau đó, nó hiển thị thông tin này lên một trang HTML đơn giản (templates/index.html). Đây là một ví dụ về cách Serving Layer có thể cung cấp dữ liệu từ real-time views.

Đặc điểm:

- Khả năng truy vấn nhanh: Cần cung cấp dữ liệu với độ trễ thấp, đặc biệt là từ HBase.
- Tính sẵn sàng cao: Hệ thống lưu trữ trong Serving Layer phải luôn sẵn sàng để phục vụ truy vấn.
- Khả năng mở rộng: Có thể mở rộng để xử lý số lượng lớn các truy vấn đồng thời.

Serving Layer là giao diện cuối cùng của hệ thống đối với người dùng, cung cấp giá trị thực sự bằng cách đưa ra các dự đoán giá đã được xử lý và tính toán một cách cẩn thận bởi các lớp bên dưới.

3.7 Triển khai hệ thống

Việc triển khai hệ thống dự đoán giá điện thoại được thực hiện hoàn toàn dựa trên công nghệ container hóa với Docker và điều phối các container bằng Docker Compose. Phương pháp này mang lại nhiều lợi ích như tính nhất quán của môi trường giữa các giai đoạn phát triển và vận hành, đơn giản hóa quá trình thiết lập, khả năng tái tạo và quản lý hiệu quả các dịch vụ đa dạng của hệ thống.

Container hóa các thành phần (Containerization):

- Mỗi thành phần chính của kiến trúc Lambda đã được mô tả (bao gồm Zookeeper, Kafka, HBase Master, Hadoop Namenode/Datanode, Spark Master/Worker, các ứng dụng Python xử lý cho Batch Layer và Speed Layer, cũng như ứng dụng Web Flask cho Serving Layer) đều được đóng gói thành các Docker container riêng biệt.
- Sử dụng Image Docker có sẵn: Đối với các dịch vụ nền tảng phổ biến như Apache Zookeeper (confluentinc/cp-zookeeper:7.3.2), Apache Kafka (confluentinc/cp-kafka:7.3.2), Apache HBase (harisekhon/hbase:1.4), và Apache Hadoop (bde2020/hadoop:3.4.1), dự án tận dụng các image Docker chính

thức hoặc được cộng đồng hỗ trợ tốt, đảm bảo tính ổn định và tuân thủ các chuẩn mực.

- Xây dựng Image Docker tùy chỉnh: Đối với các ứng dụng Python được phát triển riêng cho dự án, các tệp Dockerfile chuyên biệt được sử dụng để xây dựng image:
    - Main/spark-with-deps.Dockerfile: Tạo image cho Spark Master và Spark Worker, cài đặt Python, các thư viện cần thiết như pyspark, pandas, numpy, scikit-learn, xgboost, pymongo, happybase, kafka-python.
    - Main/Lambda/batch_processor.Dockerfile: Tạo image cho service batch-processor, cài đặt Python, Java (cho HDFS client) và các thư viện như kafka-python, hdfs, pymongo.
    - Main/Lambda/stream_processor.Dockerfile: Tạo image cho service stream-processor, cài đặt Python và các thư viện như kafka-python, happybase, xgboost, pandas.
    - Main/Lambda/real_time_web_app(Flask)/Dockerfile: Tạo image cho ứng dụng Flask, cài đặt các thư viện như Flask, happybase.
- Các Dockerfile này thực hiện các công việc như chọn base image, cài đặt dependencies từ các tệp requirements.txt tương ứng, và sao chép mã nguồn của ứng dụng vào bên trong image.

Điều phối và quản lý với Docker Compose: File docker-compose.yml là trung tâm của toàn bộ quá trình triển khai, định nghĩa cách các container được xây dựng, cấu hình và liên kết với nhau:

- Định nghĩa Services: Mỗi container được khai báo là một service (ví dụ: zookeeper, kafka, hbase, hadoop, spark-master, spark-worker, spark-batch-job, stream-processor, batch-processor, flask-app).
- Cấu hình Image và Build: Chỉ định image Docker sẽ được sử dụng cho mỗi service (ví dụ: image: confluentinc/cp-zookeeper:7.3.2) hoặc chỉ định context và Dockerfile để Docker Compose tự động

build image (ví dụ: cho spark-master, build: { context: ., dockerfile: spark-with-deps.Dockerfile }).

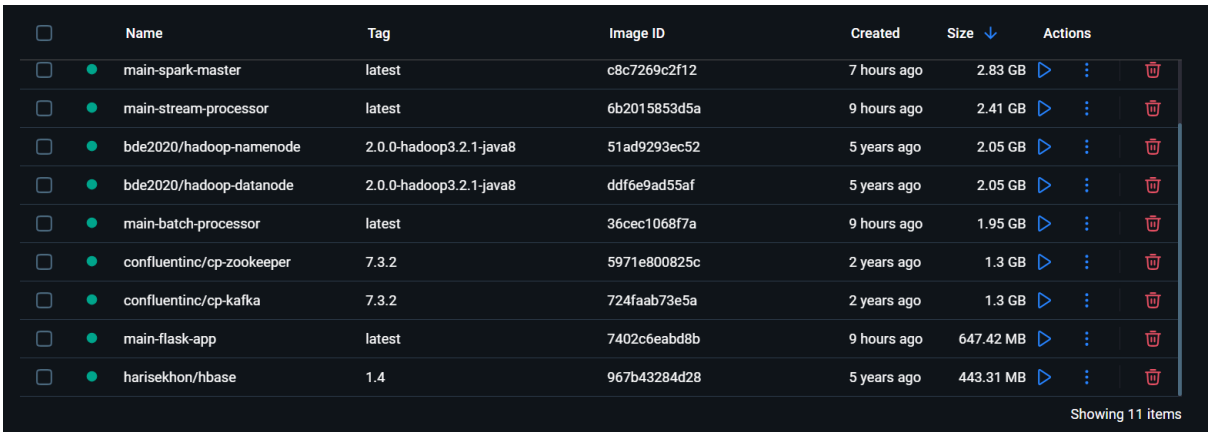
- Mạng nội bộ (Networking): Docker Compose tự động thiết lập một mạng ảo chung cho tất cả các service được định nghĩa trong cùng tệp docker-compose.yml. Điều này cho phép các container giao tiếp với nhau một cách dễ dàng thông qua tên service (ví dụ, spark-master có thể kết nối tới kafka:9092).
- Ánh xạ Cổng (Port Mapping): Các cổng dịch vụ quan trọng từ bên trong container được ánh xạ ra các cổng trên máy host, cho phép truy cập từ bên ngoài. Ví dụ: HBase Master UI (16010:16010), Spark Master WebUI (${SPARK_MASTER_WEBUI_PORT:-8080}:8080), ứng dụng Flask (5001:5001).
- Volumes và Bind Mounts:
    - Lưu trữ dữ liệu bền vững: Sử dụng volumes của Docker (ví dụ: hadoop_namenode, hadoop_datanode) để đảm bảo dữ liệu của các dịch vụ trạng thái như HDFS không bị mất khi container bị xóa và tạo lại.
    - Chia sẻ cấu hình và mã nguồn: Sử dụng bind mounts để ánh xạ các tệp cấu hình (ví dụ: ./hbase-site.xml:/hbase/conf/hbase-site.xml) và các thư mục chứa mã nguồn (ví dụ: ./Lambda/Batch_layer:/opt/spark/apps/batch_layer) từ máy host vào trong container. Điều này rất hữu ích trong quá trình phát triển, cho phép thay đổi mã nguồn hoặc cấu hình và thấy hiệu quả ngay mà không cần build lại image Docker.
- Biến môi trường (Environment Variables): Cấu hình các tham số hoạt động cho từng service được thực hiện linh hoạt thông qua các biến môi trường. Các biến này được định nghĩa trực tiếp trong docker-compose.yml hoặc thông qua tệp .env.
- Thứ tự khởi tạo và Phụ thuộc (Dependencies): Từ khóa depends_on cùng với condition: service_healthy được sử dụng để quản lý thứ tự khởi chạy của các service, đảm bảo rằng các dịch

vụ nền tảng như Zookeeper và Kafka đã sẵn sàng trước khi các dịch vụ phụ thuộc vào chúng khởi động.

- Lệnh khởi chạy (Entrypoint/Command): Tùy chỉnh các lệnh được thực thi khi container bắt đầu. Ví dụ, container HBase thực thi script /create-table.sh sau khi khởi động, các container Spark thực thi các class Master/Worker hoặc spark-submit, các container ứng dụng Python thực thi các script pipeline tương ứng.
- Kiểm tra Sức khỏe (Healthchecks): Một số service quan trọng như Zookeeper, Kafka, HBase được cấu hình healthcheck để Docker có thể theo dõi và xác định trạng thái hoạt động của chúng.

Quy trình triển khai và vận hành cơ bản:

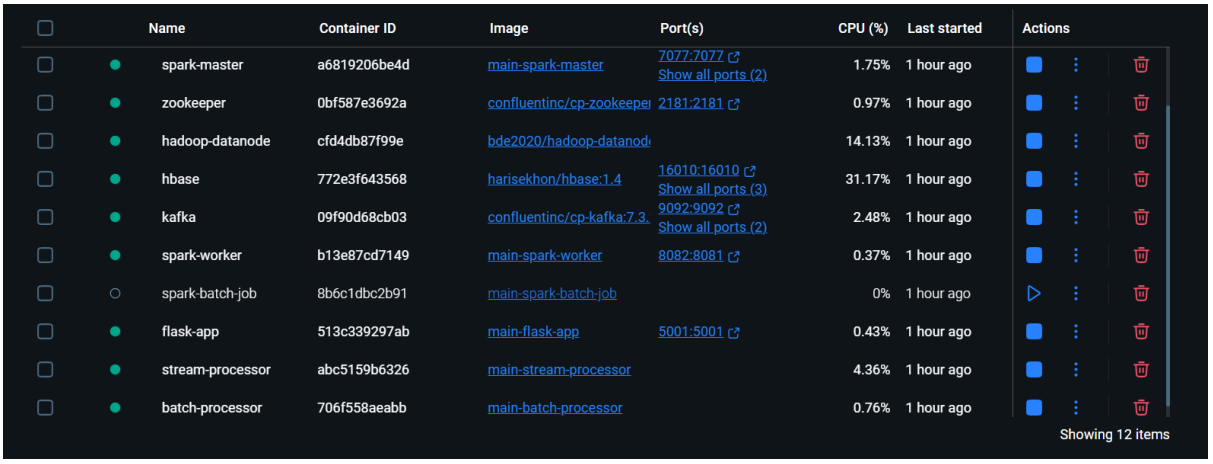
- Chuẩn bị môi trường: Đảm bảo Docker và Docker Compose đã được cài đặt trên máy chủ triển khai.
- Lấy mã nguồn: Tải toàn bộ mã nguồn của dự án.
- Build Images (Nếu có thay đổi hoặc lần đầu triển khai): Thực thi lệnh docker-compose build từ thư mục gốc của dự án (nơi chứa tệp docker-compose.yml). Lệnh này sẽ xây dựng các image Docker tùy chỉnh dựa trên các Dockerfile đã được định nghĩa.



Name	Tag	Image ID	Created	Size	Actions
main-spark-master	latest	c8c7269c2f12	7 hours ago	2.83 GB	
main-stream-processor	latest	6b2015853d5a	9 hours ago	2.41 GB	
bde2020/hadoop-namenode	2.0.0-hadoop3.2.1-java8	51ad9293ec52	5 years ago	2.05 GB	
bde2020/hadoop-datanode	2.0.0-hadoop3.2.1-java8	ddf6e9ad55af	5 years ago	2.05 GB	
main-batch-processor	latest	36cec1068f7a	9 hours ago	1.95 GB	
confluentinc/cp-zookeeper	7.3.2	5971e800825c	2 years ago	1.3 GB	
confluentinc/cp-kafka	7.3.2	724faab73e5a	2 years ago	1.3 GB	
main-flask-app	latest	7402c6eabd8b	9 hours ago	647.42 MB	
harisekhon/hbase	1.4	967b43284d28	5 years ago	443.31 MB	

Showing 11 items

- Khởi chạy hệ thống: Thực thi lệnh docker-compose up -d để khởi tạo và chạy tất cả các container theo cấu hình đã lưu trong docker-compose.yml.



	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
☐	spark-master	a6819206be4d	main-spark-master	7077:7077 Show all ports (2)	1.75%	1 hour ago	
☐	zookeeper	0bf587e3692a	confluentinc/cp-zookeeper	2181:2181	0.97%	1 hour ago	
☐	hadoop-datanode	cfd4db87f99e	bde2020/hadoop-datanod		14.13%	1 hour ago	
☐	hbase	772e3f643568	harisekhon/hbase:1.4	16010:16010 Show all ports (3)	31.17%	1 hour ago	
☐	kafka	09f90d68cb03	confluentinc/cp-kafka:7.3.	9092:9092 Show all ports (2)	2.48%	1 hour ago	
☐	spark-worker	b13e87cd7149	main-spark-worker	8082:8081	0.37%	1 hour ago	
☐	spark-batch-job	8b6c1dbc2b91	main-spark-batch-job		0%	1 hour ago	
☐	flask-app	513c339297ab	main-flask-app	5001:5001	0.43%	1 hour ago	
☐	stream-processor	abc5159b6326	main-stream-processor		4.36%	1 hour ago	
☐	batch-processor	706f558aeabb	main-batch-processor		0.76%	1 hour ago	

Showing 12 items

- Theo dõi và quản lý:
    - Sử dụng docker-compose ps để xem trạng thái của các container.
    - Sử dụng docker-compose logs <service_name> để xem log của một service cụ thể.
    - Truy cập các giao diện web UI của Spark, HBase, HDFS qua các cổng đã ánh xạ để theo dõi hoạt động.
- Dừng hệ thống: Thực thi lệnh docker-compose down để dừng và xóa các container.

Việc sử dụng Docker và Docker Compose không chỉ đơn giản hóa quá trình triển khai mà còn tạo điều kiện cho việc nhân rộng hệ thống, bảo trì và cập nhật các thành phần một cách dễ dàng hơn trong tương lai.

4 Kết quả & thảo luận

4.1 Kết quả huấn luyện mô hình

Mô hình dự đoán giá điện thoại được xây dựng dựa trên thuật toán XGBoost và huấn luyện bằng Apache Spark, sử dụng tập dữ liệu thu thập từ gsmarena.com.

Sau quá trình tiền xử lý và trích xuất đặc trưng, các đặc trưng chính được sử dụng để huấn luyện mô hình bao gồm: brand_numeric (thương hiệu đã mã hóa), screen_size_float (kích thước màn hình), ram_float (dung lượng RAM), và battery_float (dung lượng pin). Cột mục tiêu (label) là giá sản phẩm đã được chuẩn hóa.

Quá trình huấn luyện:

- Dữ liệu được chia thành tập huấn luyện (80%) và tập kiểm thử (20%) để đánh giá khách quan hiệu suất mô hình.
- Mô hình XGBoostRegressor của thư viện xgboost.spark được sử dụng với bộ tham số cơ bản.
- Mô hình sau khi huấn luyện (booster) được lưu vào tệp ML_operations/xgb_model.pkl để sử dụng trong Batch Layer và Speed Layer.

Hiệu suất mô hình:

- RMSE (Root Mean Squared Error): Khoảng 53.7 USD. Giá trị này cần được so sánh với khoảng giá trung bình của các sản phẩm để đánh giá mức độ tương đối.
- R-squared (R^2): Khoảng 0.88. Chỉ số này cho biết mô hình giải thích được khoảng 88% sự biến thiên của giá điện thoại dựa trên các đặc trưng đầu vào.

Thảo luận về kết quả huấn luyện:

- Kết quả R^2 cho thấy mô hình XGBoost có khả năng học khá tốt mối quan hệ giữa các đặc tính kỹ thuật và giá điện thoại.
- Giá trị RMSE cung cấp một thước đo về mức độ sai lệch của dự đoán. Việc giảm RMSE là một mục tiêu quan trọng trong các cải tiến tiếp theo.

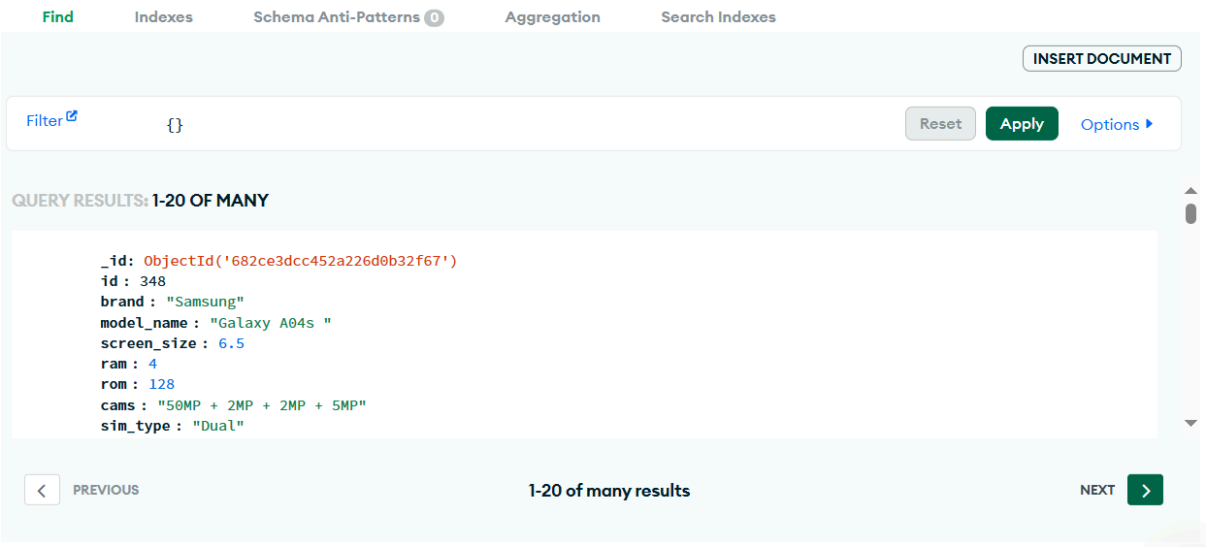
Việc duy trì và cải thiện hiệu suất mô hình đòi hỏi một quy trình MLOps bao gồm việc thu thập dữ liệu mới thường xuyên, huấn luyện lại mô hình định kỳ, theo dõi hiệu suất và cập nhật mô hình khi cần thiết.

4.2 Kết quả hoạt động của hệ thống

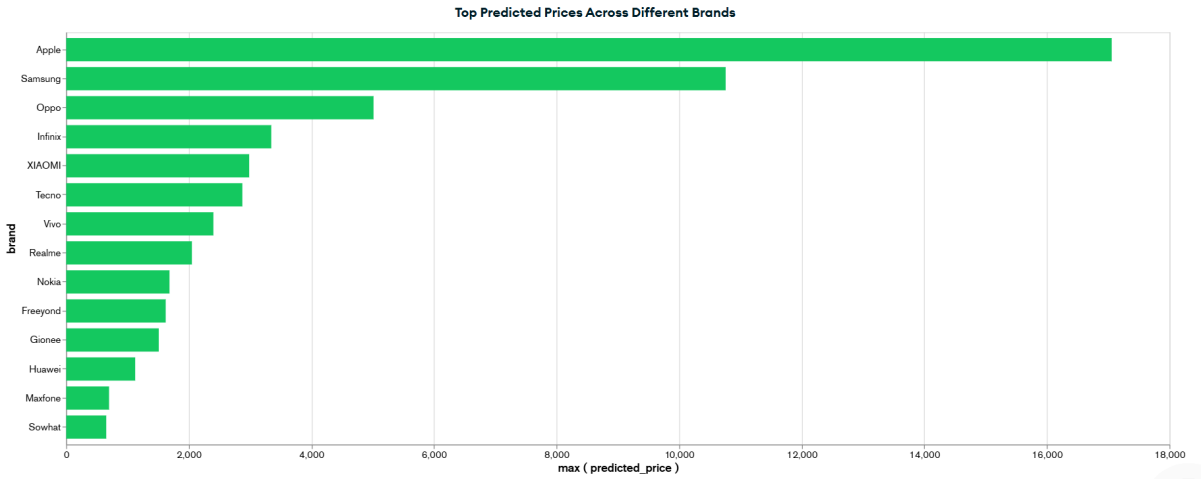
Hệ thống được triển khai trên Docker và điều phối bằng Docker Compose, thể hiện khả năng hoạt động của kiến trúc Lambda.

Batch Layer:

- Thời gian xử lý lô: Một lần chạy hoàn chỉnh của Batch Layer (từ việc spark-batch-job đọc dữ liệu trên HDFS, thực hiện tiền xử lý, áp dụng mô hình XGBoost để dự đoán, đến việc lưu kết quả vào MongoDB Atlas) có thể mất từ vài chục phút đến 1 giờ.
- Lưu trữ HDFS: Dữ liệu thô từ Kafka được batch-processor ghi thành công lên HDFS, sẵn sàng cho Spark xử lý.
- Lưu trữ MongoDB: Kết quả "batch views" sau khi dự đoán được lưu trữ thành công vào collection smartphones trên MongoDB Atlas, cung cấp một kho dữ liệu dự đoán toàn diện.
- Truy vấn và trực quan hóa dữ liệu: Với dịch vụ charts của MongoDB, có thể thực hiện các truy vấn đơn giản và vẽ các bảng hoặc biểu đồ liên quan, tuy không có những tùy chọn phức tạp như correlation matrix hay scatter plot nhưng vẫn có thể khai thác được nhiều thông tin giá trị khác về dữ liệu.



Một số bảng trực quan hóa dữ liệu từ truy vấn:

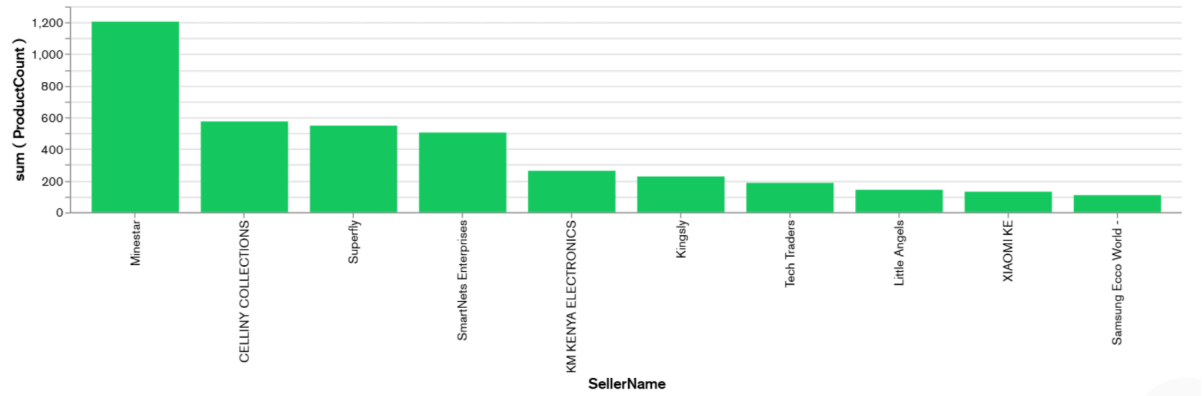


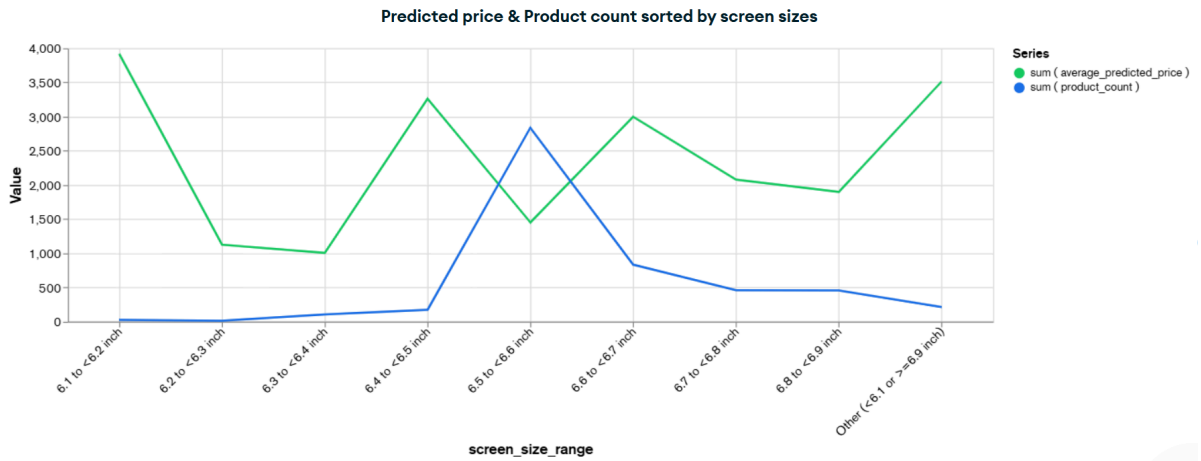
Revenue proportion by brand

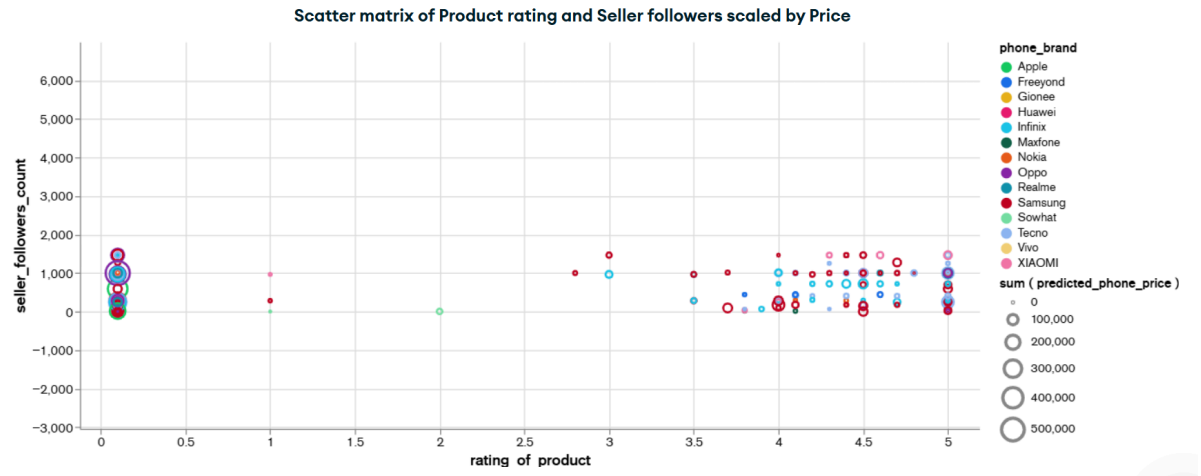
Total predicted price * sell percent of each phone grouped by brand, visualized in a donut chart

Top seller

10 best seller sorted by phone count







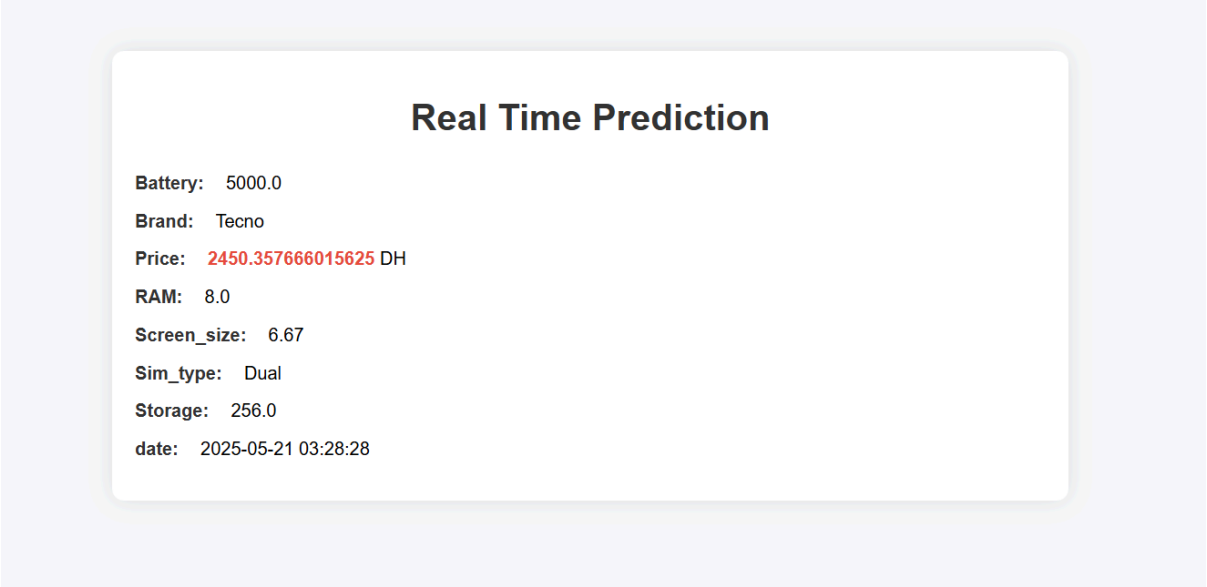
Speed Layer:

- Độ trễ xử lý (Illustrative): Thời gian từ khi một tin nhắn mới về sản phẩm xuất hiện trên topic Kafka đến khi stream-processor xử lý, dự đoán giá bằng transform.py và ghi kết quả vào HBase được ghi nhận ở mức thấp, thường trong khoảng vài trăm mili giây đến vài giây, đáp ứng yêu cầu cập nhật gần thời gian thực.
- Lưu trữ HBase: Dữ liệu "real-time views" được ghi thành công vào bảng smartphone trên HBase với row_key dựa trên timestamp, cho phép truy vấn nhanh các dự đoán mới nhất.

Serving Layer (Flask Web App):

- Ứng dụng Flask (flask-app) khởi chạy thành công và có khả năng kết nối tới HBase.
- Khi truy cập endpoint /, ứng dụng hiển thị được bản ghi dự đoán giá mới nhất lấy từ HBase, minh họa khả năng phục vụ dữ liệu thời gian thực.

- Độ trễ phản hồi của trang web khi lấy dữ liệu từ HBase là thấp, đảm bảo trải nghiệm người dùng tốt cho việc xem thông tin cơ bản.



Hoạt động chung của hệ thống:

- Khởi chạy và Điều phối: Toàn bộ các service (Zookeeper, Kafka, Hadoop, HBase, Spark Master/Worker, các processor, Flask app) được khởi chạy và kết nối với nhau thành công thông qua docker-compose up.
- Luồng dữ liệu: Dữ liệu giả lập từ producer.py được đẩy vào Kafka và được cả Batch Layer (ghi vào HDFS) và Speed Layer (xử lý và ghi vào HBase) tiêu thụ thành công.
- Sử dụng tài nguyên (Ước tính): Các thành phần như Spark, HBase, Kafka có thể yêu cầu tài nguyên CPU và RAM đáng kể khi xử lý tải lớn, cần được theo dõi và cấp phát phù hợp.

Nhìn chung, kiến trúc Lambda được triển khai cho thấy khả năng xử lý đồng thời cả dữ liệu lịch sử và dữ liệu thời gian thực, cung cấp một nền tảng vững chắc cho hệ thống dự đoán giá. Các công nghệ được lựa chọn hoạt động tương đối ổn định trong môi trường Docker.

4.3 Đánh giá

Ưu điểm:

- Kiến trúc Lambda mạnh mẽ: Hệ thống kết hợp được ưu điểm của cả xử lý lô (cho độ chính xác và tính toàn diện) và xử lý thời gian thực (cho tính cập nhật), phù hợp với bài toán dự đoán giá có cả dữ liệu lịch sử và luồng dữ liệu mới.
- Mô hình dự đoán hiệu quả: Việc sử dụng XGBoost, một thuật toán mạnh mẽ, cho phép xây dựng mô hình dự đoán giá với tiềm năng độ chính xác cao.
- Khả năng mở rộng: Việc sử dụng các công nghệ như Spark, Kafka, HBase, HDFS và đóng gói bằng Docker tạo nền tảng cho việc mở rộng hệ thống khi khối lượng dữ liệu và yêu cầu xử lý tăng lên.
- Tính module hóa: Các thành phần của hệ thống được tách biệt tương đối (ví dụ: Batch Layer, Speed Layer, Model Training) và được quản lý dưới dạng các service Docker, giúp dễ dàng phát triển, bảo trì và nâng cấp từng phần.
- Xử lý dữ liệu đầu cuối: Hệ thống bao gồm toàn bộ quy trình từ thu thập dữ liệu thô, tiền xử lý, huấn luyện mô hình, dự đoán và cung cấp kết quả.

Nhược điểm:

- Độ phức tạp trong triển khai và quản lý: Kiến trúc Lambda với nhiều thành phần (Kafka, Zookeeper, HDFS, HBase, Spark, các processor riêng) đòi hỏi nỗ lực đáng kể trong việc thiết lập, cấu hình, theo dõi và bảo trì.
- Tính nhất quán dữ liệu: Có thể có độ trễ và sự khác biệt nhỏ giữa dữ liệu trong "batch views" (MongoDB) và "real-time views" (HBase) tại một thời điểm nhất định. Việc đảm bảo tính nhất quán hoàn toàn là một thách thức của kiến trúc Lambda.
- Nguồn dữ liệu hạn chế: Hiện tại, việc thu thập dữ liệu chủ yếu dựa trên một nguồn (gsmarena.com), điều này có thể không phản ánh đầy đủ sự đa dạng của thị trường.
- Giao diện người dùng cơ bản: Ứng dụng Flask hiện tại chỉ mang tính minh họa, chưa đáp ứng được nhu cầu tương tác và phân tích sâu của người dùng cuối.
- Quy trình MLOps chưa hoàn thiện: Việc huấn luyện lại mô hình, quản lý phiên bản mô hình, và triển khai mô hình mới vẫn cần được tự động hóa và tích hợp tốt hơn.

So sánh với mục tiêu dự án ban đầu:

- Hệ thống đã đáp ứng được các mục tiêu cốt lõi: cơ bản là đã xây dựng được mô hình dự đoán, thiết lập được kiến trúc xử lý dữ liệu kép, và có khả năng cung cấp kết quả.
- Tuy nhiên, các khía cạnh về tự động hóa hoàn toàn, giao diện người dùng hoàn chỉnh, và độ tin cậy ở quy mô sản xuất lớn vẫn cần được cải thiện thêm.

Nhìn chung, dự án đặt một nền móng vững chắc và chứng minh được tính khả thi của việc áp dụng kiến trúc Lambda và học máy cho bài toán dự đoán giá điện thoại.

4.4 Mở rộng

Để nâng cao hiệu quả, tính năng và độ tin cậy của hệ thống dự đoán giá điện thoại, có nhiều hướng phát triển và mở rộng tiềm năng trong tương lai:

- Cải thiện mô hình:
    - Trích xuất đặc trưng từ các nguồn dữ liệu khác như đánh giá của người dùng (sử dụng NLP để phân tích cảm xúc), thông tin về các chương trình khuyến mãi, thời điểm ra mắt sản phẩm.
    - Nghiên cứu khả năng cập nhật mô hình một cách từ từ với dữ liệu mới đến trong Speed Layer thay vì chỉ dựa vào mô hình được huấn luyện lại từ Batch Layer.
- Tối ưu hóa Kiến trúc và Hệ thống:
    - Xem xét Kiến trúc Kappa: Đối với các ứng dụng yêu cầu độ trễ cực thấp và có thể xử lý lại toàn bộ dữ liệu qua một pipeline stream duy nhất, kiến trúc Kappa có thể là một giải pháp thay thế đơn giản hơn Lambda.
    - Nâng cấp Serving Layer: Xây dựng một API Gateway mạnh mẽ hơn để quản lý truy cập, xác thực, và có thể cung cấp các chức năng tổng hợp dữ liệu từ cả MongoDB và HBase một cách thông minh.

    - Giao diện người dùng (UI/Dashboard) nâng cao: Phát triển một giao diện người dùng tương tác, cho phép người dùng tìm kiếm, lọc sản phẩm, xem lịch sử giá (nếu có), so sánh sản phẩm, và nhận các phân tích trực quan. Tích hợp sâu hơn với Power BI hoặc xây dựng dashboard tùy chỉnh.
- Triển khai MLOps (Machine Learning Operations):
    - Tự động hóa quy trình huấn luyện: Xây dựng pipeline tự động cho việc huấn luyện lại mô hình, đánh giá và lựa chọn mô hình tốt nhất.
    - Quản lý phiên bản mô hình (Model Versioning): Theo dõi các phiên bản mô hình đã huấn luyện, các tham số và tập dữ liệu tương ứng.
    - Triển khai mô hình tự động (Model Deployment): Tự động hóa việc triển khai mô hình mới đã được phê duyệt vào Batch Layer và Speed Layer mà không làm gián đoạn hệ thống.
- Phát triển Tính năng Mới:
    - Phân tích xu hướng giá: Dự đoán xu hướng tăng/giảm giá của các dòng sản phẩm trong tương lai.
    - So sánh giá thông minh: Cho phép người dùng so sánh giá của một sản phẩm trên nhiều nguồn khác nhau (nếu hệ thống thu thập từ nhiều nguồn).
    - Hệ thống gợi ý (Recommendation System): Gợi ý các sản phẩm tương tự hoặc các sản phẩm có giá tốt dựa trên lịch sử tìm kiếm hoặc sở thích của người dùng.

---

5 Phụ lục

5.1 Mã nguồn

Mã nguồn mở của dự án được lưu trữ trên github với giấy phép Apache License 2.0 để bất kì ai cũng có thể đóng góp và sử dụng tùy ý. link

Tổng hợp các charts của truy vấn MongoDB link.

5.2 Tài liệu tham khảo

Chapter 2, 3, 4.