LTR MODEL API

[검색 시스템 고도화 - LTR 2.0]

기간 : 2024.11 ~ 2025.01

목적 : 검색 시스템의 LTR 모델 서빙 아키텍처 개선

인원 : 5명 (백엔드 2, 추천 모델러 1, PM 2)

담당 : 모델 서빙 백엔드 API 개발

Elasticsearch 플러그인 기반 LTR 시스템을 독립적인 모델 서빙 아키텍처로 전환하는 프로젝트입니다.
Kotlin과 Spring Boot 기반으로 구축했으며, gRPC를 도입하여 성능을 크게 개선했습니다.
Blue-Green 배포 전략으로 무중단 운영을 실현했으며, 영역 CTR 2.44%p 증가 등 가시적인 성과를 달성했습니다.

LTR 2.0 프로젝트 개요

프로젝트 배경

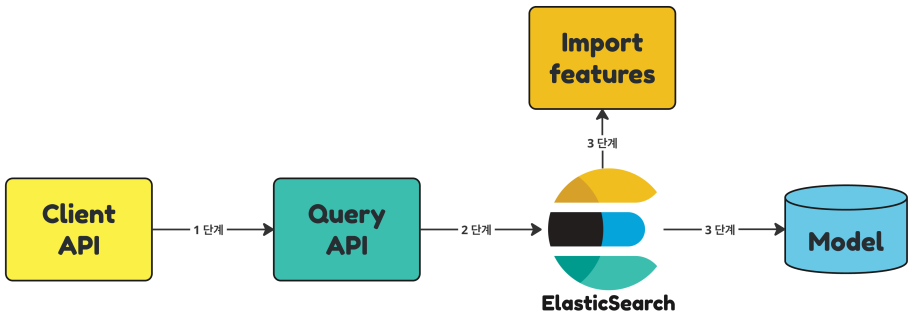
LTR 이란?

> LTR(Learning to Rank)은 머신러닝을 활용하여 검색 결과의 순위를 최적화하는 기술
> 문서나 상품의 텍스트 관련성, 사용자 행동 데이터(클릭률, 구매율), 메타데이터(가격, 리뷰 수, 평점) 등 다양한 특성을 입력으로 활용하여 가장 관련성 높은 결과를 상위에 노출시키는 것을 목표로 한다
> Elasticsearch에서는 이를 플러그인 형태로 제공하여 feature 저장, 모델 저장, 로깅 등의 기능을 지원한다

LTR 1.0

기존 프로세스

1. 사용자 검색 요청 접수
2. Query API에서 ES 쿼리 생성 (Top K 상품 선정, feature 계산, 모델 지정)
3. Elasticsearch(ES)에서 검색 및 기본 LTR 플러그인을 통해 순위(랭킹) 결정
    - 관련 상품 검색
    - feature 계산 및 LTR 모델 적용
    - 최종 점수 계산 및 정렬



기존 시스템의 한계

기존 LTR 1.0 시스템은 Elasticsearch(ES)의 외부 플러그인 기반으로 운영되고 있었으나, 다음과 같은 주요 한계점들이 존재했습니다:

1. **알고리즘 제약**
    - ES 플러그인에서 제공하는 기본 LTR 알고리즘으로만 제한
    - 새로운 알고리즘 적용 및 실험의 어려움
2. **기술적 제약**
    - feature 엔지니어링의 제한
        - 모든 feature를 ES 쿼리 형태로 변환해야 하는 번거로움
        - 복잡한 feature 생성의 어려움
    - 인프라 부하
        - ES 클러스터 용량 증가
        - 시스템 부하 가중

이러한 문제점들로 인하여 다음 방향으로 LTR 프로젝트를 개선하고자 한다

- 알고리즘 및 랭킹 모델 선택의 자유도 확보
- ES 클러스터 부하 감소
- 시스템 확장성 개선

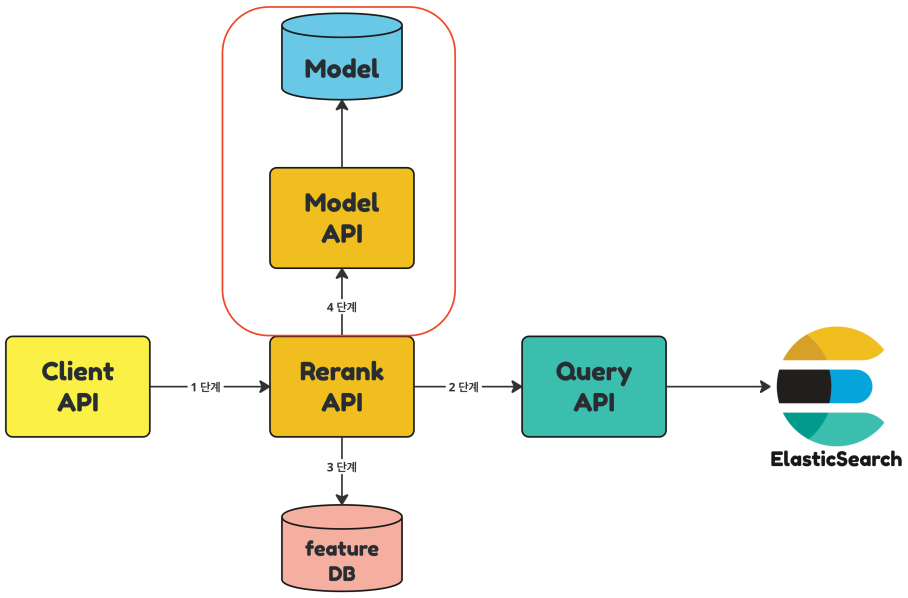
LTR 2.0

핵심 목표

- **ES 플러그인 의존도 제거:** 독립적인 모델 서빙 시스템 구축
- **유연한 모델 운영:** 모델과 알고리즘의 자유로운 교체 가능
- **데이터 기반 개선:** AB 테스트를 통한 지속적인 성능 검증
- **검색 품질 향상:** 랭킹 최적화를 통한 사용자 경험 개선

개선 프로세스

1. **검색 요청 접수**
    - 사용자가 검색어 입력 후 상품 목록 요청
2. **기본 검색 수행**
    - Query API를 통해 ES 상품 목록 조회
    - 기본 검색 파라미터 적용
3. **feature 추출**
    - 조회된 상품들의 특성(feature) 정보를 DB에서 수집
4. **LTR Model API를 통해 feature 점수 계산**
    - 학습된 모델을 호출하여 각 상품의 최종 feature score 계산
5. **최종 결과 제공**
    - 계산된 점수를 기준으로 상품 순서 재정렬
    - 최종 결과로 반환



기대 효과

- **기술적 성과**
    - 학습 모델 교체가 용이한 아키텍쳐 도입으로 다양한 알고리즘 적용
    - ES 클러스터 부하 감소 및 시스템 자원 최적화
- **시스템 운영 효율성**
    - 실시간 모니터링 체계 구축
    - 독립적인 확장성 확보
- **검색 품질 및 사용자 경험 개선**
    - 고객에게 더 매력적인 상품 리스트 제공
    - CTR(클릭률) 2%p 상승

적용 범위

- 서비스: 통합 검색결과
- 플랫폼: 모바일 앱, 모바일 웹
- 단계적 적용: AB 테스트를 통한 검증 후 확대

프로젝트 구성

팀 구성

- **백엔드 개발자 (2명)**
    - 검색 결과 re-rank 시스템 개발
    - 모델 서빙 API 개발
- **추천 모델러 (1명)**
    - LTR 모델 개발 및 성능 최적화
- **PM (2명)**
    - 프로젝트 기획 및 관리
    - 성과 지표 정의 및 측정

협업 체계

- 주 1회 정기 프로젝트 전체 회의를 통한 진행 상황 공유 및 리뷰
- 백엔드 개발자 간 수시 대면 미팅으로 요구사항 상세화
- 장애 상황 발생 시 즉각적인 공동 대응 체계 운영
- Confluence를 통한 체계적인 문서화 및 지식 공유

담당 역할

모델 서빙을 위한 백엔드 API 개발을 담당하였으며, 주요 책임은 다음과 같습니다:

- LTR 모델 서빙 시스템 설계 및 구현
- 분산 모델 인스턴스 관리 시스템 구축
- gRPC 통신 환경 구축 (rerank-api, ltr-model-api)
- Jenkins 기반 CI/CD 파이프라인 구축 및 무중단 배포 환경 구성

기술 설계 및 구현

시스템 아키텍처 개요

기술 스택

백엔드 개발 환경

서버 : Kotlin, Spring Boot

모델 : XGBoost

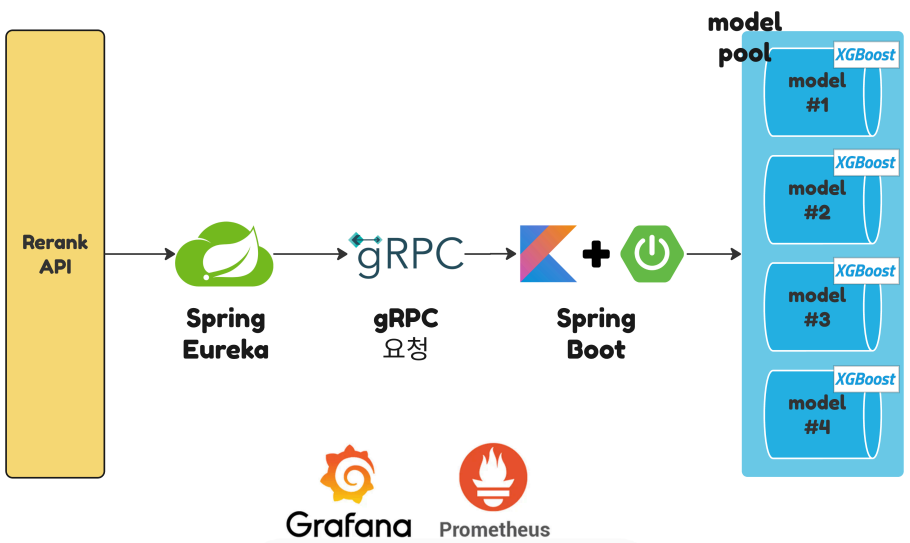
통신 : gRPC

모니터링 : Grafana, Prometheus

Service Discovery : Spring Eureka

CI/CD : Jenkins

부하테스트 : nGrinder, k6



예상 트래픽 및 목표 QPS

- **예상 트래픽 :** 초당 700 QPS (peak time 기준)
- **목표 응답 시간 :** 10ms 이내

주요 설계 결정사항

모델 인스턴스 복제 및 교체 전략

- **인스턴스 구성**
  - 서버당 4개의 모델 인스턴스를 운영하도록 구성
  - nGrinder 성능 테스트 결과:
    - 4개 인스턴스가 가장 높은 TPS(883.0) 기록
    - 인스턴스 수를 늘려도 성능이 선형적으로 증가하지 않음
    - 모델이 차지하는 메모리 크기를 고려했을 때 4개가 가장 효율적
  - Round-robin 방식으로 요청을 균등하게 분산

nGrinder 테스트	소요 시간	TPS	에러율	Vuser
model-1	10분	624.2	0.0%	99
model-2	10분	794.0	0.0%	99
model-4	**10분**	**883.0**	**0.0%**	**99**
model-8	10분	830.0	0.0%	99
model-16	10분	827.0	0.0%	99

- **모델 교체 프로세스 개선**
  - 기존 Rolling 배포 방식의 한계
    - 4대의 서버 순차 배포로 10분 이상 소요
    - 실시간으로 모델 변경 반영 불가능
    - 배포 중 트래픽 처리 불안정성 존재
    - 기존 요청 처리중에 model dispose 하여 에러 발생 위험
  - Blue-Green 배포 도입 효과
    - Scheduler를 통해 실시간으로 모델 변경을 감지하여 운영 중에도 교체
    - 신규 요청은 새 모델로, 진행 중인 요청은 기존 모델로 처리
    - 무중단 배포로 서비스 안정성 확보

gRPC 도입

- 결정 사항: 기존 REST 요청은 목표 응답시간을 달성하지 못하여 gRPC로 전환 (응답시간 10ms 초과)
- 근거:
  - 대용량 데이터 전송 최적화 필요 (200개 상품 × 40개 Feature Matrix)
  - 낮은 레이턴시 요구사항 충족
  - Client-Server 간 주고받는 데이터 형식이 명확하여 Protocol Buffers 정의에 적합

1. 메시지 크기 비교

지표	REST	gRPC	개선율
요청 크기	455.15 KB	177.73 KB	**60.95% 감소**
응답 크기	44.53 KB	22.48 KB	**49.52% 감소**

2. 단일 요청 응답시간

통계 지표	REST	gRPC	개선율
평균 응답시간	7.11ms	1.69ms	**76.2% 감소**
최소 응답시간	6.38ms	1.54ms	**75.9% 감소**
최대 응답시간	9.59ms	3.18ms	**66.8% 감소**
표준편차	0.62ms	0.19ms	**69.4% 감소**
95% 백분위수	8.25ms	1.87ms	**77.3% 감소**
99% 백분위수	9.36ms	2.66ms	**71.6% 감소**

3. nGrinder 부하 테스트 (100명의 동시 사용자)

지표	REST	gRPC	비교
QPS (초당 처리량)	894.4	2,196.4	**2.46배 향상**
총 처리 요청 수	8,948	21,966	**2.46배 많은 처리량**
평균 응답시간	10.68ms	4.05ms	**62.1% 감소**
에러 발생률	0%	0%	**동일**

주요 성능 향상 포인트

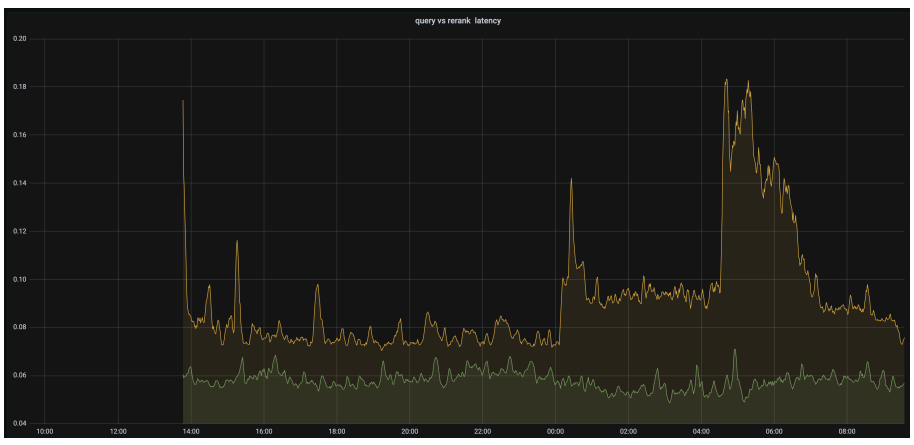
1. **메시지 효율성**
   - Protocol Buffer의 효율적인 직렬화로 메시지 크기 대폭 감소
2. **응답 속도**
   - 모든 응답시간 지표에서 60% 이상의 개선
   - 평균 응답시간 기준 62.1% 향상
   - 목표 응답속도 달성 (10ms 이하)
   - 안정적인 성능 (낮은 표준편차)
3. **처리량**
   - 2.46배 높은 QPS
   - 동일 시간, 동일 리소스로 2.46배 많은 요청 처리
   - 에러 없는 안정적인 처리

운영 및 모니터링

AB 테스트를 통한 단계적 검증

1차 AB 테스트

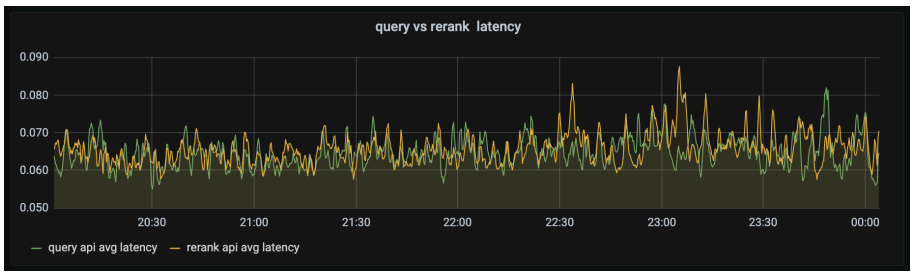
- **테스트 설계**
  - 운영 환경의 실제 트래픽으로 시스템 안정성과 성능을 검증
  - 트래픽 분배 비율을 점진적으로 증가시키며 시스템 수용 능력 측정
  - 기존 LTR과 신규 LTR의 응답 속도 비교 측정
- **테스트 결과**
  - 트래픽이 증가할수록 캐시 적중률 상승하면서 응답속도 개선
  - 미사용중인 여유 몽고디비 세트로 전환하면 부하 문제 없이 시스템을 운영할 수 있을 것으로 예상



차수	비율		TOP K (feature)		속도 비교		
	A(LTR 1.0)	B(LTR 2.0)	A(LTR 1.0)	B(LTR 2.0)	query api	rerank api	DIFF
1	9	1	1000(20)	400(40)	57ms	83ms	26ms
2	2	8	1000(20)	400(40)	57ms	71ms	14ms
3	6	4	1000(20)	400(40)	57ms	75ms	18ms
4	9	1	1000(20)	1000(40)	60ms	107ms	47ms

2차 AB 테스트

- **테스트 설계**
  - 신규 LTR 2.0와 기존 LTR 1.0과 서비스를 5:5 비율로 트래픽 분배
  - 동일한 검색 환경에서 두 서비스의 성능 비교 평가
  - CTR을 핵심 지표로 선정하여 사용자 만족도 측정 및 비교



- **테스트 결과**
  - **응답속도 측정**

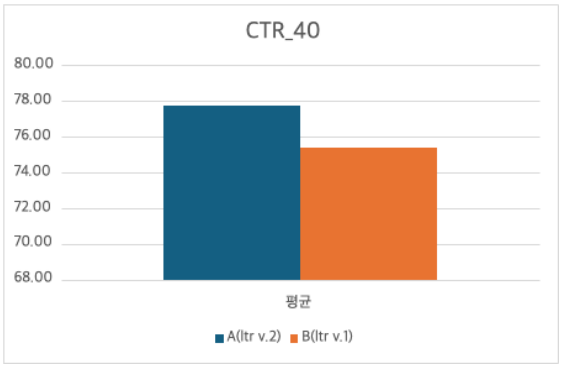
차수	비율		TOP K (feature)		속도 비교	
	A(LTR 2.0)	B(LTR 1.0)	A(LTR 2.0)	B(LTR 1.0)	rerank api	query api
1	5	5	200(40)	1000(20)	66ms	65ms

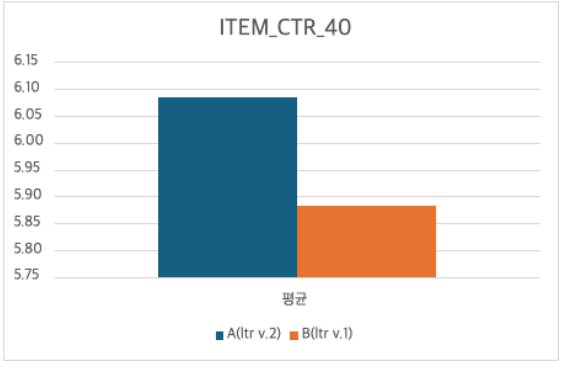
  - **CTR 측정**

CTR_40				ITEM_CTR_40			
VISIT_DT	A(LTR 2.0)	B(LTR 1.0)	DIFF %	VISIT_DT	A(LTR 2.0)	B(LTR 1.0)	DIFF %
2024-12-19	74.44	72.09	2.35	2024-12-19	5.95	5.75	0.2
2024-12-20	75.01	72.73	2.28	2024-12-20	6.05	5.82	0.23
2024-12-21	78.46	76.03	2.43	2024-12-21	6.08	5.88	0.2
2024-12-22	81.06	78.52	2.54	2024-12-22	6.19	6	0.19
2024-12-23	78.41	76.23	2.18	2024-12-23	6.12	5.92	0.2
2024-12-24	79.04	76.8	2.24	2024-12-24	6.12	5.92	0.2
평균	**77.74**	**75.40**	**2.34**	평균	**6.09**	**5.88**	**0.20**

최종 AB 결과 [전체]

- **영역 CTR ▲ 2.34%p 상승 (A : 77.74% / B : 75.40%)**
- **상품 CTR ▲ 0.20%p 상승 (A : 6.09% / B : 5.88%)**
  - 영역 CTR : 상품 클릭 / 영역 노출 (검색결과의 검색 횟수만큼 카운트)
    상품 CTR : 상품 클릭 / 상품 노출 (검색결과의 상품 노출만큼 카운트)





비즈니스 성과

LTR 2.0 프로젝트를 통해 **검색 품질이 크게 개선**되었습니다.

A/B 테스트 결과, **영역 CTR이 2.34%p 증가**했으며, **상품 CTR도 0.20%p 상승**했습니다.

이는 사용자들이 **검색 결과에서 더 많은 클릭을 수행**하고 원하는 상품을 더 쉽게 찾을 수 있게 되었음을 의미합니다.

운영 효율성 향상

Blue-Green 배포 방식 도입으로 **서비스 무중단 모델 업데이트**가 가능해졌습니다.

Grafana, Prometheus를 통해 **체계적인 모니터링 시스템 구축**으로 안정적인 서비스 운영 기반을 마련했습니다.

특히 장애 상황에 서킷이 열리어 기존 LTR 1.0 서비스를 호출하도록 대응 체계를 구축하여 **시스템 안정성을 크게 향상**시켰습니다.

기술적 성과

검색 시스템의 구조를 기존의 ES 플러그인 의존적인 구조 기반에서 **독립적인 모델 서빙 구조로 전환**하는데 성공했습니다.

이를 통해 **모델 및 LTR 알고리즘 선택의 자유도 확보**로 확장성이 높아졌고, **ES 클러스터의 부하**도 줄일 수 있었습니다.

gRPC 도입과 로컬 캐시 최적화를 통해 **시스템의 응답 속도도 개선**되었습니다.

현재는 ES 버전 업그레이드 프로젝트도 병렬적으로 진행되어 기존 **평균 65ms 응답시간에서 55ms 응답시간 까지 감소**하였습니다.

Troubleshooting

모델 갱신 과정의 안정성 확보

Issue

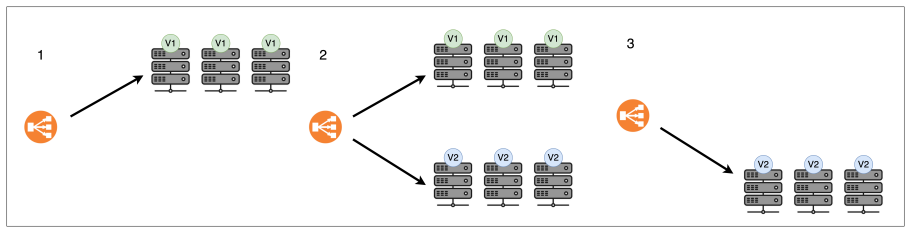
- 운영 중인 모델의 버전 교체 과정에서 진행 중이던 요청들이 완료되지 않은 채 모델이 dispose 되어 inference 실패 발생

Approach

- Blue-Green 배포 전략 기반의 모델 서빙 아키텍처 설계
- 서버당 두 개의 모델 인스턴스 그룹을 운영하며 트래픽 점진적 전환
- 기존 모델의 Graceful Shutdown을 통한 안전한 요청 처리 보장

Outcome

- 무중단 모델 교체 시스템 구현으로 서비스 안정성 확보



gRPC 서비스 디스커버리 문제

Issue

- gRPC 서비스를 구축한 후 테스트 과정에서 Eureka가 서비스를 제대로 Discover 하지 못하는 문제 발생

Approach

- 서버 로그 분석을 통해 application.yml의 설정이 예상과 다르게 적용
- spring.profiles.active 설정이 다른 환경 설정까지 영향을 미치는 점 파악
- 환경별로 yml 파일을 완전히 분리하고 spring.config.activate.on-profile 방식으로 구조를 변경

Outcome

- Eureka Service Discovery 정상적으로 동작
- 환경별 설정 관리 용이성 증가

```
spring:
  profiles:
    active: prod
```

```
spring:
  config:
    activate:
      on-profile: prod
```

로컬 캐시 메모리 최적화

Issue

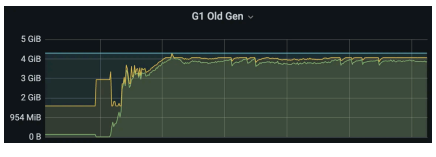
- 로컬 캐시의 과도한 힙 메모리 사용으로 OOM 위험 발생
- GC 부하 증가로 인한 전반적인 시스템 성능 저하

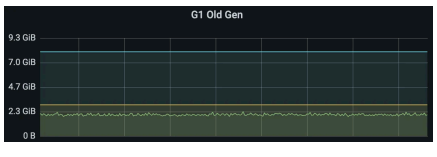
Approach

- 캐시 엔트리 개수 및 힙 메모리 재설정
- JVM 프로파일링을 통한 메모리 사용량 상세 분석
- 대부분의 메모리를 차지하는 List<Float> 자료구조를 FloatArray로 최적화

Outcome

- 적용 후 메모리 사용량이 크게 감소하여 안정적인 운영 가능





XGBoost 호환성 문제

Issue

- Java와 Python 환경 간 XGBoost 라이브러리 호환성 문제 발생
- feature score를 계산하는 함수의 Missing value 처리 로직 차이로 인한 예측 결과 불일치

Approach

- 단기적으로는 null 대신 적절한 default value을 설정하여 대응
- 다양한 입력값에 대한 Cross-validation을 통해 두 환경의 결과값 일치 여부 검증

Outcome

- 근본적인 해결을 위해 Python 기반의 FastAPI로의 전환을 검토

회고 및 향후 과제

기술적 성장

시스템 설계 및 운영 경험

- 초당 700 QPS 처리가 가능한 모델 서빙 시스템 설계
- Round-robin 방식의 모델 인스턴스 로드밸런싱 구현
- gRPC 기반 MSA 환경에서의 서비스 간 통신 체계 구축
- A/B 테스트 기반의 점진적 트래픽 전환 및 Grafana/Prometheus 모니터링 체계 확립

성능 최적화 달성

- 복잡한 연산 추가에도 기존 응답 속도 15% 개선(65ms → 55ms)
- gRPC 도입, 로컬 캐시 최적화, 모델 인스턴스 복제 등 다양한 최적화 기법 적용

팀 문화 기여

지식 공유 활동

- 트러블슈팅 경험을 팀 내에 공유하여 유사 문제 해결에 기여
- 프로젝트 진행 과정과 기술적 의사결정 사항을 상세히 문서화
- 정기적인 팀 회의를 통한 프로젝트 현황 및 기술 지식 공유

협업 문화 개선

- 백엔드 개발자 간 긴밀한 협업으로 문제 해결 속도 향상
- 장애 상황에서의 공동 대응을 통한 팀 책임감 강화
- 상호 코드 리뷰를 통한 코드 품질 개선 및 지식 공유

당면 과제

기술 호환성 개선

- XGBoost의 Java-Python 환경 간 missing value 처리 불일치 해결
- Python 기반 FastAPI 전환 검토를 통한 안정성 확보
- 유지보수성 향상을 위한 기술 스택 개선

성능 개선 영역

- 레거시 ES Query API가 전체 응답 시간의 80%를 차지하는 문제 해결 필요
- 모델의 feature 개수 증가
- 피크 타임 트래픽 처리를 위한 시스템 확장성 확보

향후 발전 계획

시스템 고도화

- 레거시 시스템 의존성 감소
- 모델의 feature 개수 40개에서 80개로 확장
- 모델 처리 가능한 상품 수 200개에서 1,000개로 확장

MLOps 역량 강화

- 모델 서빙 시스템의 자동화 수준 향상
- 모델 갱신 파이프라인 고도화
- 타 도메인 랭킹 시스템으로의 확장 가능성 모색