December 10, 2025

▣ **강의명:** CSCI E-103: 재현 가능한 머신러닝

▣ **주차:** Lecture 08

▣ **교수명:** Anindita Mahapatra
Eric Gieseke

▣ **목적:** Lecture 08의 핵심 개념 학습

Contents

1 개요: 이 노트의 핵심

□ 핵심 요약

이 문서는 머신러닝(ML) 프로젝트의 '재현성(Reproducibility)'을 달성하는 방법을 다룹니다.
재현성이란, 동일한 데이터와 코드로 언제든 동일한 모델과 결과를 만들 수 있음을 의미합니다. 하지만 ML 프로젝트는 코드 외에도 데이터, 환경, 파라미터 등 수많은 변수로 인해 재현이 극히 어렵습니다. ('숨겨진 기술 부채' 문제)
이 노트는 재현성을 가로막는 도전 과제(모델 드리프트, 단편화된 도구)를 살펴보고, 이에 대한 해결책으로 MLOps(엠엘옵스) 문화와 MLflow(엠엘플로우)라는 강력한 도구를 소개합니다.
특히 MLflow의 4가지 핵심 구성요소(Tracking, Projects, Models, Registry)가 어떻게 실험 관리, 환경 패키징, 모델 배포, 버전 관리를 자동화하여 이 모든 문제를 해결하는지 상세히 해설합니다.

□ 예제: title

이 노트를 가장 효과적으로 학습하기 위한 추천 순서입니다.

1. **용어 정리(2장)**: 먼저 핵심 용어들의 '쉬운 정의'를 훑어봅니다.
2. **문제 인식(3, 4장)**: ML 재현성이 '왜' 어렵고 '왜' 중요한지(숨겨진 기술 부채, 드리프트) 공감합니다.
3. **핵심 개념(5, 6, 7장)**: 문제 해결에 필요한 피처 스토어, ML 파이프라인 등의 개념을 이해합니다.
4. **솔루션(8, 9장)**: '어떻게' 해결하는지 MLflow의 4가지 기능을 중심으로 학습하고, 실습 코드를 살펴봅니다.
5. **복습(10, 11장)**: 체크리스트와 한 페이지 요약으로 스스로 점검합니다.

2 주요 용어 정리

본격적인 학습에 앞서, 이 문서에서 반복적으로 사용되는 핵심 용어들을 쉬운 설명과 함께 정리합니다.

Table 1: 재현 가능한 *ML* 핵심 용어

용어	쉬운 설명 (직관적 이해)	원어 (English)	비고
재현성 (Reproducibility)	"요리 레시피(코드, 데이터, 환경)만 있으면 언제 누가 따라 해도 똑같은 맛(결과)의 요리(모델)가 나오는 상태."	Reproducibility	ML에서는 극히 어려움.
숨겨진 기술 부채 (Hidden Technical Debt)	"ML 시스템은 'ML 코드'라는 작은 얼음덩이 밑에 '데이터 관리, 모니터링, 인프라'라는 거대한 빙산이 숨어 있다는 개념."	Hidden Technical Debt	구글 논문에서 유래.
데이터 엔지니어 (Data Engineer)	"데이터라는 '원자재'를 수집, 정제, 저장하고 '공급망(파이프라인)'을 구축/관리하는 인프라 설계자."	Data Engineer	(DE)
데이터 사이언티스트 (Data Scientist)	"정제된 '원자재(데이터)'를 분석하고 '피처'를 추출하여 '제품(모델)'을 개발하고 통찰력을 도출하는 연금술사."	Data Scientist	(DS)
ML 엔지니어 (ML Engineer)	"개발된 '제품(모델)'을 '대량 생산(배포)'하고, '품질 관리(모니터링)'하며, '자동화 공정(MLOps)'을 구축하는 전문가."	ML Engineer	(MLE)
MLOps (엠엘옵스)	"DevOps(개발+운영)의 ML 버전. 모델 개발(DS)과 배포/운영(MLE)을 자동화된 파이프라인으로 통합하는 문화이자 기술."	ML Operations	DataOps + ModelOps
피처 스토어 (Feature Store)	"모델 훈련과 실시간 추론에 사용될 '피처(특성)'들을 중앙에서 관리, 저장, 공유하는 전용 데이터베이스."	Feature Store	재사용성, 일관성 확보.
모델 드리프트 (Model Drift)	"시간이 흘러 현실 세계의 데이터 패턴이 변하면서, 과거 데이터로 학습한 모델의 성능이 저하되는 현상."	Model Drift	모델은 '살아있는 자산'.

Table 1 – 이어서

용어	쉬운 설명 (직관적 이해)	원어 (English)	비고

MLflow (엠엘플로우)	"단편화된 ML 작업을 하나로 통합 관리해주는 'ML 프로젝트 만능 도구(Swiss Army Knife)'. 오픈소스 프레임워크."	MLflow	재현성의 핵심 솔루션.
MLflow Tracking	"모든 ML 실험의 '실험 노트'. 사용한 파라미터, 성능(지표), 산출물(모델 파일, 그래프)을 자동으로 기록."	MLflow Tracking	"내가 뭘 했는지" 추적.
MLflow Projects	"실험 환경(코드 + 라이브러리)을 '밀키트'처럼 통째로 패키징하여, 어디서든 동일한 환경을 재현하게 함."	MLflow Projects	'requirements.txt' 포함.
MLflow Models	"훈련된 모델을 '표준 규격'으로 저장. '파이썬 함수', '스파크 UDF' 등 다양한 '맛(Flavor)'으로 서빙 가능."	MLflow Models	배포 유연성.
MLflow Model Registry	"완성된 모델을 등록하고 '버전 관리' 하는 '모델의 Git 저장소'. 'Staging', 'Production' 단계를 관리."	MLflow Model Registry	모델 거버넌스.
하이퍼파라미터 튜닝	"모델이 '학습'하는 값(가중치)이 아닌, 개발자가 '설정'하는 값(e.g., 학습률, 트리 깊이)의 최적 조합을 찾는 과정."	Hyperparameter Tuning	'Optuna', 'Hyperopt' 사용.
A/B 테스팅 (A/B Testing)	"기존 모델(챔피언)과 새 모델(도전자)에 실제 트래픽을 나눠 보내어, 어떤 모델이 더 나은 성과를 내는지 비교 검증."	A/B Testing	e.g., 80% vs 20%
Medallion 아키텍처	"데이터를 'Bronze(원본)' → 'Silver(정제)' → 'Gold(집계)' 3단계로 정제하며 가치를 높이는 데이터 관리 패턴."	Medallion Architecture	Silver는 ML용, Gold는 BI용.

Table 1 – 이어서

용어	**쉬운 설명 (직관적 이해)**	**원어 (English)**	**비고**

3 문제: 왜 ML 재현성은 어려운가?

3.1 숨겨진 기술 부채 (Hidden Technical Debt)

초심자들은 흔히 'ML = 모델링 코드'라고 생각하지만, 이는 큰 오해입니다. 유명한 "숨겨진 기술 부채" 개념에 따르면, 실제 ML 시스템에서 순수 ML 코드가 차지하는 비중은 매우 작습니다.

오해: ML은 코드다 vs. 진실: ML은 시스템이다

잘못된 직관: "모델 성능을 높이려면 복잡한 알고리즘(ML 코드)을 짜는 것이 전부다."
올바른 이해: "실제 ML 시스템의 90% 이상은 ML 코드가 아니라, 그 코드를 둘러싼 **방대한 인프라**다." 이 인프라(빙산의 수면 아래)를 관리하지 못하면 시스템 전체가 무너집니다.

ML 코드를 둘러싼 인프라 (빙산의 아랫부분):

- **Data Collection**: 데이터 수집
- **Data Verification**: 데이터 검증 (e.g., 데이터 품질)
- **Configuration**: 수많은 설정값 (e.g., 어떤 피처를 쓸지, 파라미터는 뭔지)
- **Feature Extraction**: 피처 추출 및 공학
- **Infrastructure**: 자원 관리 (e.g., CPU, GPU, 분산 처리)
- **Monitoring**: 모델 성능 및 데이터 드리프트 감시
- **Serving**: 모델을 실제 서비스로 배포

이것이 재현성과 무슨 관계인가? 모델을 재현하려면, 그 작은 'ML 코드'뿐만 아니라 이 모든 '주변 인프라'의 상태(설정값, 데이터 버전, 라이브러리 버전 등)를 **정확히 동일하게** 복원해야 합니다. 이는 사실상 불가능에 가깝습니다.

3.2 단편화된 생태계 (Fragmented Ecosystem)

ML 프로젝트는 하나의 도구로 끝나지 않습니다. 데이터 전처리, 훈련, 튜닝, 배포에 사용되는 언어와 프레임워크가 모두 다릅니다.

- **언어**: Python, R, SQL, Spark
- **프레임워크**: Scikit-learn, PyTorch, TensorFlow, XGBoost
- **배포 환경**: Docker, Kubernetes, Batch, Streaming

이처럼 파편화된 도구들을 수동으로 연결하다 보면, "내 노트북에서는 잘 됐는데, 동료 컴퓨터나 서버에서는 왜 안 되지?"라는 '의존성 지옥(Dependency Hell)'에 빠지게 됩니다. 각 도구의 버전이 조금만 달라져도 결과가 바뀌거나 오류가 발생합니다.

3.3 모델은 '살아있는 자산'이다

모델은 한 번 만들고 끝나는 '조각상'이 아닙니다. 시간이 지남에 따라 성능이 변하는 '살아있는 자산'이며, 지속적인 '건강검진(모니터링)'과 'TLC(Tender Loving Care, 애정 어린 돌봄)'가 필요합니다.

□ **예제: title**

모델을 훈련시키는 것은 '운전면허 시험(과거 데이터)'에 합격한 것과 같습니다. 하지만 면허를 땄다고 해서 실제 '도로(현실 세계)'에서 완벽하게 운전할 수 있는 것은 아닙니다.
시간이 지나면 '새로운 교통 법규(패턴 변화)'가 생기고, '못 보던 형태의 교차로(신규 데이터)'가 등장합니다. 과거의 지식(훈련된 모델)만으로는 사고(잘못된 예측)를 낼 수 있습니다. 이것이 바로 **모델 드리프트**입니다.

결국, 6개월 전의 '최고 성능 모델'을 오늘날 똑같이 재현하는 것 자체가 무의미할 수 있습니다. 중요한 것은 "6개월 전에 '어떻게' 그 모델을 만들었는지" 그 과정을 정확히 추적(Tracking)하고, "오늘의 데이터로 '빠르게' 재훈련"하여 새 모델을 배포할 수 있는 **자동화된 파이프라인**입니다.

이것이 바로 MLOps와 MLflow가 해결하려는 핵심 문제입니다.

4 ML 수명 주기와 다양한 역할

ML 프로젝트는 여러 전문가의 협업으로 이루어집니다. 각자의 역할과 전체 흐름을 이해하는 것이 중요합니다.

4.1 ML 수명 주기 (Lifecycle)

모든 ML 프로젝트는 다음과 같은 단계를 반복하는 순환 고리(Loop)입니다.

□ 핵심 정보

ML 프로젝트의 6단계 수명 주기
순방향 프로세스:
1. **원본 데이터 (Raw Data)** →
2. **ETL (정제)** →
3. **피처화 (Featurize)** →
4. **훈련 (Train)** →
5. **서빙/추론 (Serve)** →
6. **모니터링 (Monitor)**

피드백 루프: 모니터링 결과 성능 저하 감지 시 → 1단계로 돌아가 재훈련

1. **데이터 준비 (Raw Data → ETL)**: 원본 데이터를 수집하고 정제합니다.
2. **피처 엔지니어링 (Featurize)**: 정제된 데이터로 모델이 학습할 '특성(Feature)'을 만듭니다.
3. **모델 훈련 (Train)**: 피처를 입력받아 모델을 학습시킵니다. (전체 과정 중 아주 작은 부분!)
4. **모델 서빙 (Serve/Inference)**: 훈련된 모델을 배포하여, 새로운 데이터에 대한 '예측(추론)'을 제공합니다. (Batch, 실시간, HTTP 등)
5. **모니터링 (Monitor)**: 모델의 성능이 잘 유지되는지, 데이터가 변하지는 않았는지(Drift) 감시합니다.
6. **피드백 루프 (Feedback)**: 모니터링 결과, 성능이 저하되면 1, 2, 3단계로 돌아가 새 데이터로 모델을 재훈련합니다.

4.2 프로젝트의 세 가지 핵심 역할

이 수명 주기는 보통 세 가지 주요 역할의 협업으로 완성됩니다.

Table 2: 데이터 프로젝트의 *3*가지 핵심 역할

역할	데이터 엔지니어 (DE)	데이터 사이언티스트 (DS)	ML 엔지니어 (MLE)
핵심 임무	데이터 인프라 구축	비즈니스 통찰력 도출	모델의 안정적인 배포/운영
주요 작업	- 데이터 수집 (Ingest) - 데이터 저장 (Store) - 데이터 파이프라인 (ETL) - 데이터 정제 (Curate)	- 피처 추출 (Extract Features) - 모델 코딩 (Code) - 모델 훈련 (Train) - 모델 검증 (Validate)	- 모델 평가 (Evaluate) - 모델 패키징 (Package) - 모델 배포 (Deploy / Serve) - MLOps 파이프라인 구축
비유	원자재 공급 및 공장 설계	시제품(Prototype) 개발	대량 생산 라인 구축/운영

협업의 어려움 (Silo 문제)

전통적으로 이 세 역할은 분리되어(Silo) 일했습니다.

- DS가 노트북(Jupyter)에서 모델을 개발 → MLE에게 코드를 전달
- MLE는 그 코드를 서버 환경(e.g., Docker)에서 다시 작성
- 이 과정에서 버전이 꼬이고, 환경이 달라 결과가 재현되지 않으며, 배포에 수개월이 소요됩니다.

MLOps와 **MLflow**는 이 장벽을 허물고, DS가 개발한 모델을 '그대로' '빠르게' 배포할 수 있도록 돕습니다.

5 핵심 개념 1: 피처 엔지니어링과 피처 스토어

5.1 피처 엔지니어링 (Feature Engineering)

- **한 줄 요약**: 데이터에 대한 도메인 지식을 사용해, ML 알고리즘이 더 잘 학습할 수 있도록 데이터를 '가공'하는 과정입니다.
- **직관적 비유**: "날고기(Raw Data)를 그대로 먹을 순 없습니다. 요리사(DS)가 먹기 좋게 손질하고, 양념하고(Feature Engineering), '스테이크(Feature)'로 만드는 것과 같습니다."
- **기술적 설명**: 원본 데이터 컬럼을 그대로 사용하기보다, 특정 의미를 갖도록 집계(Aggregate)하거나 변환하는 작업을 말합니다.

□ 예제: title

원본 데이터 (Raw Data):
- 고객 ID, 상품명, 구매 일시, 구매액

가공된 피처 (Features):
- 'avg$_t$ransaction$_a$mount' : ′ ′( )
- 'purchase$_f$requency$_7$days' :′ 7 ′( )
- 'weekday$_o$r$_w$eekend' : ′ ′vs′ ′ ( )
- 'weather$_o$n$_p$urchase' :′ ′( )

모델은 '구매액' 자체보다 '평균 거래액'이나 '최근 구매 빈도' 같은 가공된 피처로부터 훨씬 더 많은 정보를 학습할 수 있습니다.

Andrew Ng의 통찰

"응용 머신러닝은 기본적으로 피처 엔지니어링이다."
(Applied machine learning is basically feature engineering.)
이는 모델의 성능을 결정하는 가장 중요한 작업이 복잡한 알고리즘을 선택하는 것이 아니라, **데이터를 얼마나 의미 있는 피처로 잘 만드느냐**에 달려있음을 의미합니다.

5.2 피처 스토어 (Feature Store)

피처 엔지니어링은 '고객 7일간 평균 구매액'처럼 **컴퓨팅 비용이 매우 비싼** 작업을 포함합니다. 또한, 여기서 심각한 재현성 문제가 발생합니다.

치명적 문제: 훈련-서빙 스큐 (Training-Serving Skew)

증상: "모델을 훈련시킬 땐(Offline) 성능이 99%였는데, 실제 서비스(Online)에 배포하니 60%로 떨어졌다!"
원인:
- **훈련 시**: 'SELECT AVG(purchase) ...' (SQL로 느긋하게 배치 계산)
- **서빙 시**: 'feature = (sum$_v$al + new$_v$al)/(n + 1)'(*Python*)

두 계산 로직이 미묘하게 다르거나(e.g., 반올림, 누락 값 처리), 데이터의 시점이 달라 동일한 피처가 아니게 됩니다. 이 미세한 차이가 모델의 성능을 급격히 저하시킵니다.

피처 스토어(Feature Store)는 이 문제를 해결합니다.

피처 스토어의 2가지 핵심 역할

피처 스토어는 피처를 위한 중앙 저장소(Repository) 입니다.

1. **일관성 보장 (Consistency)**: 피처를 **단 한 번만 계산**하여 저장소에 저장합니다. 모델 훈련 시와 모델 서빙 시에 **정확히 동일한 피처**를 가져다 쓰게 하여 '훈련-서빙 스큐'를 원천 봉쇄합니다.
2. **재사용성 및 효율성 (Reusability & Efficiency)**: A팀이 비싼 돈 들여 계산한 '유저 프로필 피처'를 B팀, C팀도 즉시 가져다 쓸 수 있습니다. 중복 계산을 방지하고, 모든 팀이 검증된 피처를 공유하게 됩니다.

6 핵심 개념 2: ML 파이프라인 (Transformers vs. Estimators)

ML 작업을 코드로 구현할 때, 특히 Spark ML 같은 프레임워크에서는 '파이프라인' 개념을 사용합니다. 이때 초심자들이 가장 혼동하는 두 가지 구성요소가 **Transformer**와 **Estimator**입니다.

□ 예제: title

데이터(재료)가 컨베이어 벨트를 따라 이동하며 요리(모델)가 완성되는 과정을 상상해 보세요.

- **Transformer (변환기)**: 재료를 '손질'하는 기계. (e.g., 'StringIndexer' = "양파 껍질 까기", 'StandardScaler' = "재료 무게 맞추기")
- **Estimator (추정기)**: 재료를 '학습'하여 '레시피(모델)'를 만드는 기계. (e.g., 'LinearRegression' = "재료 조합을 학습해 최고의 스테이크 레시피 만들기")

이 둘의 가장 결정적인 차이는 '학습(Learning)'의 유무입니다.

Table 3: *Transformer vs. Estimator* 비교

특징	Transformer (변환기)
목적	데이터 변환 (Preprocessing)
핵심 메서드	`.transform()`
입력 → 출력	DataFrame → DataFrame
설명	데이터를 입력받아 규칙에 따라 변환된 데이터를 반환합니다.
예시	- 'StringIndexer' (문자 → 숫자) - 'OneHotEncoder' (숫자 → 벡터) - 'StandardScaler' (정규화) - 'Ve

□ 핵심 요약

중요한 점은, **Estimator**의 `.fit()` 메서드를 실행하면 그 결과물로 **Model**이 나온다는 것입니다. 그리고 이 **Model**은 그 자체로 **Transformer**입니다. (새로운 데이터에 대해 `.transform()` 또는 `.predict()`를 수행할 수 있으므로)
즉, **Estimator**는 '모델을 만드는 객체'이고, **Transformer**는 '데이터를 변환하는 객체'입니다.

7 핵심 개념 3: 모델 드리프트 (Model Drift)

7.1 모델은 왜 성능이 저하되는가?

모델 드리프트(Model Drift)는 "배포된 모델의 성능이 시간이 지남에 따라 저하되는 현상"을 말합니다. 모델은 '오래된(stale)' 데이터로 학습했기 때문에, '현재(fresh)'의 데이터 패턴을 따라가지 못하게 됩니다.

우리의 목표는 모델 성능이 완전히 나빠지기(벼랑 끝) 전에, 성능 저하의 '징후(inflection)'를 미리 감지하고 모델을 '재훈련(refresh)'하는 것입니다.

7.2 드리프트의 4가지 유형

드리프트가 발생했을 때, "왜?" 성능이 떨어졌는지 원인을 파악하는 것이 중요합니다.

Table 4: 모델 드리프트의 *4*가지 유형과 대응

드리프트 유형	설명 (무엇이 변했는가?)	대응 전략
Feature Drift (피처 드리프트)	**입력 데이터(X)의 분포**가 변했습니다. 예: 새로운 카테고리의 상품이 등장하여 '카테고리' 피처의 분포가 바뀜.	피처 생성 프로세스 점검, 새로운 데이터로 재훈련
Label Drift (레이블 드리프트)	**정답 데이터(Y)의 분포**가 변했습니다. 예: 갑자기 '사기 거래(정답)'의 비율이 급증함.	레이블 생성 프로세스 점검, 재훈련
Prediction Drift (예측 드리프트)	**모델의 예측(Y-hat) 분포**가 변했습니다. 예: 모델이 갑자기 모든 거래를 '정상'으로만 예측하기 시작함.	모델 훈련 과정 점검, 비즈니스 영향도 평가
Concept Drift (개념 드리프트)	**X와 Y의 '관계' 자체**가 변했습니다. (가장 심각) 예: *COVID-19* 발생. 이전에는 '주말'과 '매장 방문'이 강한 양의 관계였으나, 팬데믹 이후 그 관계가 완전히 깨짐.	**단순 재훈련으로 해결 불가.** 새로운 피처 엔지니어링, 근본적인 모델 재설계 필요.

이러한 드리프트를 감지하기 위해 **지속적인 모니터링**이 필수적이며, 드리프트가 감지되면 **자동으로 재훈련** 파이프라인이 실행되도록 하는 것이 MLOps의 핵심 목표입니다.

8 해결책: MLOps와 MLflow

지금까지 살펴본 모든 문제(숨겨진 기술 부채, 단편화된 생태계, 수동 작업, 모델 드리프트)를 해결하기 위한 문화적, 기술적 접근 방식이 바로 **MLOps**입니다.

- **DevOps** = Development (개발) + Operations (운영) → 코드 빌드/배포 자동화
- **DataOps** = Data (데이터) + Operations (운영) → 데이터 파이프라인 자동화
- **MLOps** = Model (모델) + Operations (운영) → 모델 훈련/배포/모니터링 자동화

MLOps는 이 모든 것을 통합하여, 데이터 수집부터 모델 배포 및 모니터링까지의 **전체 수명 주기를 자동화**하는 것을 목표로 합니다.

8.1 MLflow: MLOps를 위한 만능 도구

MLflow는 MLOps를 구현하기 위한 가장 인기 있는 오픈소스 프레임워크입니다. 단편화된 ML 수명 주기를 관리하기 위한 4가지 강력한 구성요소를 제공합니다.

MLflow의 4가지 핵심 구성요소

1. **MLflow Tracking (실험실 노트)**
2. **MLflow Projects (환경 밀키트)**
3. **MLflow Models (표준 규격 모델)**
4. **MLflow Model Registry (모델 Git 저장소)**

8.1.1 1. MLflow Tracking (트래킹)

- **문제점**: 데이터 사이언티스트가 수백 번의 실험을 하며 파라미터를 바꿔봅니다. "어제 돌렸던 그 모델이 성능이 제일 좋았는데... 파라미터가 뭐였더라?" → **기억 상실**
- **솔루션**: 모든 실험을 '실험 노트'에 자동으로 기록합니다.
- **무엇을 기록하는가?**
  - **Parameters (입력)**: 실험에 사용한 하이퍼파라미터 (e.g., 'learning$_rate$ = 0.1')
  - **Metrics (출력)**: 실험 결과(성능 지표) (e.g., 'RMSE=0.85')
  - **Artifacts (산출물): 모델 파일 그 자체**, 성능 그래프(PNG), 피처 중요도 등
  - **Source (출처)**: 이 실험을 실행한 소스 코드(e.g., 노트북 버전)
- **핵심 API**:
  - 'mlflow.start$_run$()' :
  - 'mlflow.log$_param$()' :
  - 'mlflow.log$_metric$()' :
  - 'mlflow.log$_artifact$()' : ( )
  - `mlflow.autolog()`: **(강력 추천)** 이 함수 하나만 호출하면, Scikit-learn, TensorFlow 등이 자동으로 위 모든 것을 기록해줍니다.

8.1.2 2. MLflow Projects (프로젝트)

- **문제점**: "제 노트북에선 잘 됐는데, 서버에선 라이브러리 버전이 안 맞아서 오류 나요." → **환경 의존성**

- **솔루션**: 코드를 실행하는 데 필요한 '환경'까지 통째로 패키징합니다.
- **어떻게?**: 프로젝트 폴더 안에 `MLproject`라는 명세 파일을 둡니다.
  - 이 파일은 "이 코드를 실행하려면 'python train.py' 명령을 써야 하고, 'requirements.txt' (또는 'conda.yaml') 파일에 명시된 **정확한 라이브러리 버전들**이 필요하다"라고 정의합니다.
- **결과**: 'mlflow run <프로젝트 경로>' 명령 하나로, MLflow가 자동으로 격리된 환경(e.g., conda)을 만들고 그 안에서 코드를 실행하여 **완벽한 재현성**을 보장합니다.

8.1.3 3. MLflow Models (모델)

- **문제점**: "DS가 Scikit-learn으로 만든 모델을 MLE가 Spark 환경에서 서빙하려니 호환이 안 됩니다." → **배포 비호환성**
- **솔루션**: 모델을 '표준화된 포맷'으로 저장합니다.
- **어떻게?**: MLflow는 모델을 저장할 때, 다양한 '맛(Flavor)'으로 저장합니다.
  - 'python$_{function}$'($pyfunc$) : **가장 중요한 맛.** $Python$ ($wrapping$) .
  - 'sklearn': Scikit-learn 네이티브 포맷
  - 'spark': Spark UDF로 바로 로드할 수 있는 포맷
- **결과**: 모델을 'pyfunc' 맛으로 저장하면, 배포팀은 이 모델이 원래 Scikit-learn인지 PyTorch인지 신경 쓸 필요 없이, 그냥 표준 Python 함수를 호출하듯 모델을 서빙할 수 있습니다.

8.1.4 4. MLflow Model Registry (모델 레지스트리)

- **문제점**: "어떤 모델이 최신 버전이죠? 이 모델은 테스트는 끝난 건가요? 지금 서비스 중인 모델은 뭐죠?" → **모델 관리/거버넌스 부재**
- **솔루션**: 모델을 위한 중앙 'Git 저장소' 역할을 합니다.
- **어떻게?**: Tracking Server에 기록된 수백 개의 모델 중, "최고의 모델"을 선택하여 **Registry에 등록(Register)**합니다.
- **핵심 기능: Stages (단계 관리)**
  - **Staging**: "이 모델을 'Staging' 환경에 배포해서 A/B 테스트를 시작하세요."
  - **Production**: "테스트 결과가 좋으니, 이 모델을 'Production' (실제 서비스)으로 승격시키세요." (e.g., v3 모델)
  - **Archived**: "이 모델(e.g., v2)은 이제 사용하지 않으니 'Archived' (보관) 상태로 변경하세요."
- **결과**: 모델의 전체 수명 주기(개발 → 테스트 → 배포 → 폐기)를 체계적으로 추적하고 관리할 수 있습니다.

9 실습: MLflow 사용하기 (주요 코드 해설)

강의 자료의 실습은 MLflow의 핵심 기능을 실제로 사용하는 방법을 보여줍니다. 여기서는 두 가지 주요 실습(XGBoost, Scikit-learn)의 핵심 코드와 **중요 수정 사항**을 해설합니다.

9.1 실습 1: XGBoost + Optuna (하이퍼파라미터 튜닝)

이 실습은 하이퍼파라미터 튜닝 라이브러리인 `Optuna`와 MLflow를 **중첩(Nested) 실행**하는 방법을 보여줍니다.

```python
import mlflow
import optuna
from sklearn.metrics import mean_squared_error

# 1. 가MLflow Unity Catalog (UC)를 레지스트리로사용하도록설정
mlflow.set_registry_uri("databricks-uc")

# 2. 가Optuna 최적화할목적 ' 함수' 정의
def objective(trial):
    # 3. 각을 trial 중첩된'(Nested) Run으로' 기록
    # 이것이핵심입니다 !
    with mlflow.start_run(nested=True):
        # 가Optuna 제안하는하이퍼파라미터
        params = {
            "max_depth": trial.suggest_int("max_depth", 3, 10),
            "n_estimators": trial.suggest_int("n_estimators", 50, 500),
            "learning_rate": trial.suggest_float("learning_rate", 0.01, 0.3)
        }

        # MLflow Tracking: 파라미터기록
        mlflow.log_params(params)

        # 모델훈련  (XGBoost)
        model = XGBRegressor(**params)
        model.fit(X_train, y_train)

        # 예측및평가
        preds = model.predict(X_val)
        rmse = mean_squared_error(y_val, preds, squared=False)

        # MLflow Tracking: 성능지표 (Metric) 기록
        mlflow.log_metric("rmse", rmse)

        return rmse # 는Optuna 이값을최소화하려고시도

# 4. 부모 Run: 전체튜닝 ' 작업을' 기록
with mlflow.start_run(run_name="XGBoost Tuning"):
    # 5. Optuna Study 생성및최적화실행
    # 주의(: n_trials은=50 시간이오래걸리므로으로    20 줄여서테스트 )
    study = optuna.create_study(direction="minimize")
```

```
    study.optimize(objective, n_trials=20) # 50 -> 20

    # 6. 가장성능이좋았던 Trial (Run)의 모델을찾음
    best_run = mlflow.search_runs(
        experiment_ids=study.study_name,
        order_by=["metrics.rmse ASC"],
        max_results=1
    ).iloc[0]

    # 7. 최고의모델을 Model 에Registry 등록''
    best_model_uri = f"runs:/{best_run.run_id}/model"
    mlflow.register_model(best_model_uri, "xgboost_best_model")
```

Listing 1: Optuna와 MLflow 중첩 실행

이 코드의 의미

- 전체 튜닝 작업(e.g., 20회)이 하나의 '부모 Run'으로 기록됩니다.
- 튜닝의 각 시도(Trial)가 '자식 Run'으로 기록되어, 어떤 파라미터가 어떤 성능을 냈는지 모두 추적할 수 있습니다.
- 튜닝이 끝나면, MLflow의 `search_runs` API를 사용해 가장 좋았던(RMSE가 가장 낮은) 자식 Run을 찾습니다.
- 이 최고의 모델을 찾아 **Model Registry**에 등록하여 'Staging' 또는 'Production' 단계로 넘길 준비를 합니다.

9.2 실습 2: Scikit-learn + Hyperopt (중요 수정 사항)

이 실습은 또 다른 튜닝 라이브러리인 `Hyperopt`와 `autolog()` 기능을 사용합니다. 특히, 강의 환경(서버리스)에서 발생하는 문제를 해결하기 위한 **중요 수정 사항**이 포함됩니다.

실습 환경 중요 수정 사항

강의 자료의 원본 코드는 두 가지 수정이 필요합니다.

1. **카탈로그명 변경**: Unity Catalog 사용 시, `main` 카탈로그가 아닌 본인 소유의 카탈로그 (e.g., `cscie103_catalog`)를 사용해야 합니다.
2. **SparkTrials → Trials**: `Hyperopt`의 `SparkTrials`는 분산 튜닝을 지원하지만, 일부 서버리스 환경에서는 작동하지 않을 수 있습니다. 이 경우, 단일 노드에서 실행되는 기본 `Trials`로 변경해야 합니다.

```
import mlflow
from hyperopt import fmin, tpe, hp, Trials # SparkTrials 대신 Trials
from sklearn.ensemble import GradientBoostingClassifier

# --- 1. 중요수정사항카탈로그 ( 설정) ---
CATALOG_NAME = "cscie103_catalog"
SCHEMA_NAME = "default"
mlflow.set_registry_uri("databricks-uc")
# 모델저장시이경로를사용 : f"{CATALOG_NAME}.{SCHEMA_NAME}.my_model"
```

```
10
11 # --- 2. MLflow autolog() 활성화 ---
12 # 이한줄이   log_param, log_metric, 을log_model 자동으로수행 !
13 mlflow.autolog(log_models=True)
14
15 # 3. Hyperopt 탐색공간 (Search Space) 정의
16 search_space = {
17     'n_estimators': hp.quniform('n_estimators', 50, 500, 10),
18     'learning_rate': hp.loguniform('learning_rate', -3, 0),
19     'max_depth': hp.quniform('max_depth', 2, 5, 1)
20 }
21
22 # 4. Hyperopt 목적함수정의
23 def train_model(params):
24     # autolog()가 켜져있으므로 , start_run()만 호출하면됨
25     with mlflow.start_run(nested=True):
26         model = GradientBoostingClassifier(**params)
27         model.fit(X_train, y_train)
28
29         # ... 평가( 로직) ...
30         accuracy = model.score(X_val, y_val)
31
32         # autolog()가 지표를자동으로기록하지만  ,
33         # 가hyperopt 반환받을값을명시적으로지정
34         return {'loss': -accuracy, 'status': 'ok'}
35
36 # --- 5. 중요수정사항   (Trials 객체) ---
37 # SparkTrials 대신로컬  Trials 사용
38 # spark_trials = SparkTrials(parallelism=4) # <- 원본오류 ( 가능)
39 local_trials = Trials() # <- 수정본안정적 ()
40
41 # 6. Hyperopt 실행 (fmin)
42 best_params = fmin(
43     fn=train_model,
44     space=search_space,
45     algo=tpe.suggest,
46     max_evals=10, # 테스트를위해회만  10 실행
47     trials=local_trials # 수정된 trials 객체전달
48 )
49
50 # 7. autolog() 덕분에모든이  Run 기록되었음
51 # search_runs()를 사용해최고의모델을찾아에    Registry 등록
52 # 이후( 과정은실습과  1 유사)
```

Listing 2: Hyperopt와 MLflow autolog() 사용 (수정 사항 적용)

10 학습 체크리스트

이 노트를 통해 다음 질문들에 답할 수 있는지 스스로 점검해 보세요.

ML의 '재현성'이 왜 중요한지, 그리고 왜 어려운지(e.g., 숨겨진 기술 부채) 설명할 수 있나요?

데이터 엔지니어(DE), 데이터 사이언티스트(DS), ML 엔지니어(MLE)의 역할을 구분하고, 이들의 협업이 왜 중요한지 이해했나요?

'피처 스토어(Feature Store)'가 무엇이며, 이것이 '훈련-서빙 스큐(Training-Serving Skew)' 문제를 어떻게 해결하는지 설명할 수 있나요?

ML 파이프라인에서 `Transformer`와 `Estimator`의 근본적인 차이(`transform` vs. `fit`)를 설명할 수 있나요?

'모델 드리프트(Model Drift)'가 무엇이며, 4가지 유형(Feature, Label, Prediction, Concept)을 구분할 수 있나요?

MLflow의 4가지 구성요소 각각의 목적을 명확히 설명할 수 있나요?

Tracking: "모든 실험을 기록하는 **실험 노트**"

Projects: "환경(라이브러리)까지 패키징하는 **밀키트**"

Models: "다양한 맛(Flavor)으로 배포하는 **표준 규격**"

Registry: "모델 버전을 관리(Staging/Prod)하는 **Git 저장소**"

MLflow의 `autolog()` 기능이 개발자에게 어떤 편리함을 주는지 이해했나요?

Model Registry의 'Stage'(e.g., Staging, Production) 개념이 A/B 테스트 및 모델 배포 관리에 왜 유용한지 이해했나요?

11 한 페이지 요약 (Quick Look)

문제: ML의 재현성 위기

1. 숨겨진 기술 부채: ML 코드는 빙산의 일각. 데이터 관리, 인프라, 모니터링 등 수면 아래의 거대한 인프라 관리가 더 복잡하고 중요함.
2. 단편화된 생태계: Python, R, Spark, TensorFlow 등 수많은 도구가 파편화되어 "내 컴퓨터에선 됐는데..."라는 '의존성 지옥' 발생.
3. 모델 드리프트: 모델은 '살아있는 자산'. 배포 후에도 현실 데이터가 변하면서(Drift) 성능이 계속 저하되므로 지속적인 관리가 필요함.

솔루션: MLOps 문화와 MLflow 도구

MLOps (문화): 데이터, 모델, 운영을 하나로 통합. DataOps (데이터 파이프라인) + ModelOps (모델 훈련/배포) + DevOps (인프라/운영) → **모델 개발부터 배포, 모니터링까지 전 과정을 자동화**

MLflow (도구): MLOps를 구현하기 위한 오픈소스 프레임워크. 단편화된 ML 수명 주기를 4가지 구성요소로 통합 관리.

MLflow Component 1: Tracking (실험 노트)

목적: "내가 뭘 했는지 잊어버리는" 문제 해결.
기능: 모든 실험의 파라미터, 성능 지표, 모델/그래프(Artifacts), 소스 코드를 자동으로 기록. `mlflow.autolog()`로 간편화.

MLflow Component 2: Projects (환경 밀키트)

목적: "내 PC에선 됐는데..." (의존성 지옥) 문제 해결.
기능: 코드 + `requirements.txt` 등 환경 정보를 통째로 패키징. 어디서든 `mlflow run`으로 동일 환경, 동일 결과 재현.

MLflow Component 3: Models (표준 규격)

목적: "훈련과 배포 환경이 다른" (호환성) 문제 해결.
기능: 모델을 다양한 '맛(Flavor)' (e.g., `pyfunc`, `spark`)으로 저장. 훈련(Python)과 배포(Spark/Docker)가 달라도 유연하게 서빙.

MLflow Component 4: Model Registry (모델 저장소)

목적: "지금 서비스 중인 모델이 뭐죠?" (버전 관리) 문제 해결.
기능: 검증된 모델을 '등록'하고 '버전' 관리. **Stage** ('Staging' → 'Production' → 'Archived')를 통해 A/B 테스트 및 릴리즈를 통제.