4.5 GBM(Gradient Boosting Machine)

1) 부스팅 알고리즘이란?

• 여러 개의 약한 학습기를 순차적으로 학습-예측하면서 잘못 예측한 데이터에 가중치 부여를 통해 오류를 개선해 나가면서 학습하는 방식
• 부스팅의 대표적인 구현: **AdaBoost(Adaptive boosting)**, **그래디언트 부스트**
☞ **에이다 부스트** - 오류 데이터에 가중치를 부여하면서 부스팅을 수행하는 대표적인 알고리즘

2) 경사 부스팅의 동작 원리

1. 초기 데이터에 대해 약한 학습기 1이 분류를 수행한다. 동그라미로 표시된 점들은 오분류된 데이터다.

2. 두 번째 단계에서 이 오분류 표본들에 가중치를 부여한다. 가중치가 커진 표본은 다음 학습기에서 더 눈에 띄게 반영된다.

3. 약한 학습기 2가 다시 분류를 수행하고, 새로 잘못 맞힌 표본이 생기면 그 표본들에 또 다시 더 큰 가중치를 준다.

4. 약한 학습기 3도 같은 방식으로 학습한다. 이렇게 오류가 난 표본에 점점 더 큰 가중치를 주면서 순차적으로 보정한다.

5. 마지막에 여러 학습기의 결정 기준을 결합해 최종 예측을 만든다. 개별 학습기보다 정확도가 높아진다.

6. 결합할 때는 학습기마다 서로 다른 가중치를 줄 수 있다(예: 첫째 0.3, 둘째 0.5, 셋째 0.8 등을 부여해 합산).

2) 하이퍼파라미터 설명

- **손실 함수**: 경사 하강에서 사용할 **비용 함수**를 고른다. 특별한 이유가 없으면 **기본값**을 쓴다.

- **학습률**: 단계별로 값을 얼마나 **업데이트할지**를 정하는 비율.
  - 범위는 **0부터 1 사이**, **기본값은 0.1**.
  - **작게** 두면 한 번에 조금씩 움직이므로 **최소 오류**에 가까이 갈 가능성이 높지만, **반복이 많이** 필요해 시간이 오래 걸릴 수 있고, 너무 작으면 모든 반복을 끝내도 최소 오류를 찾지 못할 가능성도 있다.
  - **크게** 두면 빨리 진행되지만 **최소 오류를 지나쳐** 성능이 떨어질 위험이 있다.

- **약한 학습기 수**:
  순차적으로 오류를 보정. 단, **많을수록 시간이 오래** 걸린다. **기본값은 100**.

- **부분 표본 비율**: 학습에 사용할 **데이터 비율**.
  - **기본값 1.0**(전체 데이터 사용). 예를 들어 **0.5면 절반**만 사용.
  - **과적합이 걱정**되면 **1보다 작은 값**으로 설정해 사용한다.

4.6 XGBosst(eXtra Gradient Boost)

1) XGBoost 개요

- XGBoost는 트리 기반 앙상블 학습에서 가장 각광받는 알고리즘 중 하나.
- GBM을 기반으로 하지만 **느린 수행 시간**과 **과적합 문제**를 규제로 해결
- 병렬 CPU 환경에서 빠른 학습이 가능하며, 기존 GBM보다 빠른 학습 속도를 제공.

2) 주요 특징

1. **뛰어난 예측 성능**: 분류와 회귀 영역에서 우수한 성능.
2. **빠른 수행 시간**: 병렬 학습을 통해 GBM 대비 빠름.
3. **과적합 규제**: 규제 기능으로 과적합을 방지.
4. **Tree Pruning**: 불필요한 가지치기를 통해 효율적 모델 생성.
5. **자체 내장된 교차 검증**: 학습 중 자동으로 교차 검증 수행.
6. **결손값 자체 처리**: 결손값을 직접 처리 가능.

3) 파이썬 래퍼 XGBoost

- 핵심 라이브러리는 C/C++로 작성됨.
- 파이썬에서는 두 가지 방식으로 사용 가능:
    - 독자적인 프레임워크 기반 **파이썬 래퍼 XGBoost 모듈**
    - 사이킷런과 호환되는 **사이킷런 래퍼 XGBoost 모듈**

사이킷런 래퍼는 fit(), predict() 등 표준 사이킷런 방식 사용 가능.
파이썬 래퍼는 DMatrix라는 전용 데이터 객체 사용.

4) 하이퍼 파라미터

XGBoost 하이퍼 파라미터는 크게 세 가지로 나뉨:

1. **일반 파라미터**: booster, silent, nthread 등 실행 관련 설정.
2. **부스터 파라미터**: eta(학습률), num_boost_rounds, max_depth, min_child_weight, gamma, subsample, colsample_bytree, lambda, alpha 등.
3. **학습 태스크 파라미터**: objective(손실 함수), eval_metric(평가 지표).

- **주요 평가 지표**:
    - rmse, mae, logloss, error, merror, mlogloss, auc

과적합 방지 방법:

- eta 값을 줄임
- max_depth 줄임
- min_child_weight 늘림
- gamma 값을 늘림
- subsample과 colsample_bytree 조정

5) 추가 기능

- **조기 중단**: 일정 횟수 반복 후 성능 개선이 없으면 학습 종료.
- **교차 검증, 성능 평가, 피처 중요도, 시각화** 기능 지원.

6) 학습과 평가 과정

- 평가 데이터는 튜플 형태로 [데이터, 'train'], [데이터, 'eval'] 같이 입력.
- eval_metric을 지정하면 학습 도중 평가 지표 결과가 출력됨.
- 학습이 완료되면 최종 모델 객체를 반환.

적용 예시: 위스콘신 유방암 데이터

- XGBClassifier를 사용해 유방암 데이터 분류 수행.
- 주요 하이퍼 파라미터: n_estimators=400, learning_rate=0.05, max_depth=3, eval_metric=logloss.
- 학습 후 예측을 통해 정확도, 정밀도, 재현율, F1, AUC 등 평가 지표 계산.
- 사이킷런 래퍼를 사용했을 때 더 좋은 성능이 나왔음.

7) 조기 중단

- 학습을 반복할 때 평가 지표가 더 이상 개선되지 않으면 학습을 중단.
- 예시에서는 400회를 설정했으나, 176번째에서 조기 중단됨.
- 가장 좋은 성능은 126번째 반복에서 나타남.
- 이후 50회 동안 개선이 없었으므로 학습 종료.

8) 예측

- 학습 완료 후 모델은 새로운 데이터에 대한 예측 수행 가능.
- 예측은 두 가지 방법으로 제공됨:
  - **클래스 값(0 또는 1) 반환**
  - **예측 확률 반환** (이진 분류 시 확률 0.5 기준으로 분류 가능)

9) 시각화 기능

- XGBoost는 내장된 시각화 기능을 제공.
- plot_importance API를 통해 피처 중요도를 막대그래프로 표현.
- 피처가 분할에 얼마나 자주 사용되었는지를 기준으로 중요도 계산.

사이킷런의 feature_importances_ 속성과 유사하지만, XGBoost는 자체 API로 바로 시각화 가능.

NumPy 기반 학습 데이터에서는 피처명을 알 수 없으므로 순서대로 f0, f1, f2… 형태로 표시됨.

10) 트리 시각화

- to_graphviz API를 통해 학습된 트리 구조를 시각화할 수 있음.
- 트리의 분할 조건과 규칙 구조를 직접 확인 가능.

11) 교차 검증 (CV)

- XGBoost는 사이킷런의 GridSearchCV와 유사하게 교차 검증 기능 제공.
- 하이퍼 파라미터 탐색과 모델 성능 평가를 동시에 진행 가능.
- 주요 인자:
  - params: 부스터 파라미터
  - dtrain: 학습 데이터
  - num_boost_round: 반복 횟수
  - nfold: 교차 검증 폴드 수
  - stratified: 층화 추출 여부
  - metrics: 평가 지표
  - early_stopping_rounds: 조기 중단 여부

4.7 LightGBM

1) LightGBM 개요

- XGBoost와 함께 부스팅 계열 알고리즘 중 가장 각광받고 있음.
- XGBoost는 뛰어난 알고리즘이지만 학습 시간이 오래 걸림.
- 특히 GridSearchCV를 이용해 하이퍼 파라미터 튜닝을 수행하면 시간이 많이 소요되어 어려움이 있음.
- LightGBM은 이러한 한계를 극복하기 위해 설계되어 **학습 속도가 빠르고 메모리 사용량이 적음**.

2) LightGBM의 장점

1. **빠른 학습 속도**: XGBoost보다 훨씬 적은 시간 소요.
2. **메모리 효율성**: 메모리 사용량이 상대적으로 적음.
3. **예측 성능**: XGBoost와 큰 차이 없음.
4. **범용성**: 다양한 기능을 제공.

* 단점은 작은 데이터셋에서는 과적합이 발생하기 쉬우며, 공식 문서에서는 10,000건 이하 데이터에서는 권장하지 않음.

3) 트리 분할 방식

- 기존 GBM 계열은 **레벨 중심 분할** 방식을 사용.
- 균형 잡힌 트리를 유지하려고 깊이가 효율적으로 줄어듦.
- LightGBM은 **리프 중심 분할** 방식을 사용.
- 손실 감소가 큰 리프 노드를 계속 분할.
- 더 깊은 트리를 만들어 예측 성능을 높일 수 있지만, 과적합 가능성이 있음.

4) XGBoost 대비 장점 요약

- 더 빠른 학습과 예측 수행 시간.
- 더 작은 메모리 사용량.
- 범주형 피처를 원-핫 인코딩하지 않고 최적화된 방식으로 처리 가능.
- 대용량 데이터에서 성능과 효율성이 뛰어남.
- GPU 학습도 지원.

5) 파이썬 패키지

- 패키지명은 lightgbm.
- 초기에 파이썬 래퍼만 제공되었으나 이후 사이킷런 호환 클래스도 개발됨.
- 대표 클래스:
  - LGBMClassifier (분류용)
  - LGBMRegressor (회귀용)

사이킷런 기반 Estimator와 동일하게 fit, predict 사용 가능.
GridSearchCV, Pipeline 등 다양한 사이킷런 유틸리티와 연동 가능.

6) LightGBM 하이퍼 파라미터

XGBoost와 유사하나 **리프 중심 분할 특성** 때문에 일부 다른 점 존재.

주요 파라미터

- num_iterations: 반복 수행 횟수 (트리 개수).
- learning_rate: 학습률, 작게 할수록 반복 횟수를 늘려야 함.
- max_depth: 트리 깊이 제한 (기본 -1, 제한 없음).
- min_data_in_leaf: 하나의 리프에 필요한 최소 데이터 수.
- num_leaves: 하나의 트리가 가질 수 있는 최대 리프 개수.
- boosting: 트리 생성 방식 (기본: gbdt, 그 외 rf).
- bagging_fraction: 샘플링 비율 (과적합 방지).
- feature_fraction: 학습 시 사용하는 피처의 비율.
- lambda_l2: L2 규제.
- lambda_l1: L1 규제.

7) Learning Task 파라미터

- objective: 손실 함수를 정의 (분류, 회귀 등).
- eval_metric: 평가 지표 지정 가능.

8) 하이퍼 파라미터 튜닝 방법

- 모델 복잡도를 줄이기 위해 주로 조정하는 값:
  - num_leaves
  - min_child_samples(min_data_in_leaf)
  - max_depth

learning_rate를 낮추고 num_iterations를 크게 하여 안정적 성능 확보.
정규화 파라미터(lambda, alpha)와 샘플링 관련 파라미터(bagging_fraction, feature_fraction)로 과적합 방지.

9) 파이썬 래퍼와 사이킷런 래퍼 비교

- LightGBM은 사이킷런과 호환 가능.
- 사이킷런 래퍼 클래스는 하이퍼 파라미터 명칭이 일부 변경됨.
- 예시:
  - num_iterations → n_estimators
  - min_data_in_leaf → min_child_samples
  - bagging_fraction → subsample
  - feature_fraction → colsample_bytree
  - lambda_l2 → reg_lambda
  - lambda_l1 → reg_alpha

4.8 베이지안 최적화 기반의 HyperOpt를 이용한 하이퍼 파라미터 튜닝

1) 기존 Grid Search 방식의 한계

- 사이킷런에서 제공하는 **Grid Search**는 가능한 모든 하이퍼 파라미터 조합을 전부 탐색하는 방식.
- 단점:
  - 파라미터 개수가 많아질수록 시간이 기하급수적으로 증가.
  - 대용량 학습 데이터에서는 시간이 너무 오래 걸려 비효율적임.

예시: LightGBM의 6개 파라미터(max_depth, num_leaves, colsample_bytree, subsample, min_child_weight, reg_alpha)를 각각 여러 값으로 설정하면 경우의 수가 수천 번이 되어 학습과 평가에 지나치게 긴 시간이 필요함.

2) 베이지안 최적화 개요

- **목적 함수의 식을 정확히 알 수 없을 때** 최적 입력값을 찾는 방법.
- 기본 원리: 이전의 탐색 결과를 바탕으로 다음 탐색 위치를 확률적으로 선택하여 효율적으로 최적값 탐색.
- 단순한 반복 탐색이 아니라 사후 확률 모델(Surrogate Model)을 만들어 최적화.
- 두 가지 핵심 요소:
  1. **대체 모델**: 주어진 입력값으로 함수 출력을 예측.
  2. **획득 함수**: 대체 모델을 기반으로 최적의 입력값을 탐색.

3) 베이지안 최적화 절차

1. 임의의 하이퍼 파라미터를 선택해 학습 및 성능 평가 → 결과 관측.
2. 관측된 결과를 토대로 대체 모델이 성능을 예측 → 가장 유망한 영역 탐색.
3. 획득 함수를 통해 다음 탐색할 입력값을 선택.
4. 선택된 값으로 다시 학습 후 성능 평가 → 이 과정을 반복해 점점 더 정확한 최적값 탐색.

4) HyperOpt의 특징

- 베이지안 최적화 기법을 구현한 대표적 라이브러리.
- 일반적으로 트리 기반 추정기(TPE)를 사용.
- 장점:
  - Grid Search처럼 모든 경우를 탐색하지 않고 효율적으로 최적값을 찾음.
  - 불필요한 연산을 줄여 대규모 데이터에서도 적용 가능.

5) HyperOpt의 입력 공간 정의

- 입력값 탐색을 위해 범위를 지정.
- 여러 방식으로 값의 범위를 줄 수 있음: 균등 분포, 정규 분포, 랜덤 정수 등

특정 파라미터에 대해 탐색 공간을 설정하면 HyperOpt이 자동으로 최적 탐색 수행

6) 목적 함수와 최적화 과정

- 목적 함수: 특정 파라미터 조합으로 학습 후 성능 지표 반환
- 반환 값: 손실과 상태
- HyperOpt은 반환된 성능 지표를 기반으로 더 좋은 입력값을 탐색
- 핵심 함수: fmin
  - 목적 함수, 탐색 공간, 최적화 알고리즘, 반복 횟수, 기록 장치 등을 입력
  - 반복하며 점진적으로 최적의 파라미터를 탐색

7) XGBoost에 HyperOpt 적용

- XGBoost 하이퍼 파라미터 최적화도 LightGBM과 동일한 방식으로 가능
- 탐색할 파라미터 예시: max_depth, min_child_weight, learning_rate, colsample_bytree 등
- 주의할 점:
    1. HyperOpt은 연속적인 입력값을 기본으로 처리 → 정수형 파라미터는 따로 정수 변환 필요.
    2. **최적화 목표를 분명히 설정해야 함** (예: 정확도 최대화, 손실 최소화).

예시 상황:

- max_depth, min_child_weight 등은 정수형이므로 반드시 변환 후 적용
- early stopping은 별도로 지원되지 않으므로 교차 검증을 통해 성능 평가

4.10

1) 언더 샘플링과 오버 샘플링 이해

- **불균형 데이터 문제**: 레이블 분포가 불균형한 데이터셋을 학습할 경우 예측 성능이 떨어질 수 있음. 특히 이상 레이블을 가진 데이터가 매우 적으면 다양한 유형을 학습하지 못함.
- **언더 샘플링**: 정상 레이블이 많은 경우 일부를 줄여서 이상 데이터 비율을 맞춤. 하지만 정상 데이터 손실이 커질 수 있음.
- **오버 샘플링**: 이상 레이블 데이터를 복제하거나 합성해 데이터 수를 늘려 균형을 맞춤. 과적합 가능성이 있지만 상대적으로 자주 활용됨.

* 대표적인 방법: SMOTE 기법 (K개의 최근접 이웃을 활용해 새로운 데이터를 합성하여 이상 클래스 데이터를 보강함.)

2) 이상치 데이터 제거 후 학습/예측/평가

- **이상치 데이터:** 전체 데이터 패턴에서 벗어난 값. 모델 성능에 부정적 영향을 줄 수 있음.
- **이상치 탐지 방법**: 여러 가지 있으나 대표적으로 **IQR 방법**이 있음.
    - 데이터를 사분위수(Q1, Q2, Q3, Q4)로 나눈 뒤,
    - Q1 − 1.5×IQR 미만, Q3 + 1.5×IQR 초과 값을 이상치로 판단.
    - 여기서 IQR은 Q3 − Q1 범위를 의미
- **박스 플롯**을 통해 IQR과 이상치를 시각적으로 확인 가능.

☞ 데이터 간 **상관관계 히트맵**을 확인한 뒤 이상치 탐지가 필요한 변수를 선택해 제거

3) SMOTE 오버 샘플링 적용 후 학습/예측/평가

- **SMOTE 적용 전후 데이터 차이**: 학습 데이터가 두 배 가까이 증가.
- **로지스틱 회귀 결과**: 재현율은 크게 향상되었으나 정밀도가 급격히 낮아져 실제 적용이 어려움. (재현율 92% 이상, 정밀도 5%대)
- **LightGBM 결과**: 재현율은 높아졌지만 정밀도가 낮아짐. 예를 들어 원래 82.88%였던 정밀도가 84.93%로 소폭 개선되었으나 재현율이 91% 이상으로 크게 증가.

4) 데이터 가공 방법별 성능 비교

데이터 가공 유형	알고리즘	정밀도	재현율	ROC-AUC
원본 데이터 사용	로지스틱 회귀	0.8679	0.6216	0.9702
	LightGBM	0.9573	0.7635	0.9709
데이터 로그 변환	로지스틱 회귀	0.8812	0.7063	0.9727
	LightGBM	0.9587	0.7654	0.9774
이상치 제거	로지스틱 회귀	0.8750	0.8206	0.9784
	LightGBM	0.9627	0.7635	0.9727
SMOTE 오버 샘플링	로지스틱 회귀	0.0542	0.9247	0.9737
	LightGBM	0.9118	0.8493	0.9814

5) 결론

- 단순 오버샘플링은 재현율을 극적으로 높이지만 정밀도가 급격히 낮아 실제 현업 적용에는 부적합.
- LightGBM: SMOTE 적용 후 정밀도와 재현율 균형이 상대적으로 양호
- 데이터 전처리(로그 변환, 이상치 제거 등)를 통해 성능이 크게 개선됨

4.11 스태킹 앙상블

1) 스태킹 앙상블 개념

- 여러 개별 알고리즘을 결합해 예측 성능을 높이는 기법
- 개별 모델의 예측값을 기반으로 또 다른 모델(메타 모델)이 최종 예측을 수행
- 개별 모델들의 예측 결과를 모아 메타 데이터 세트를 만들고 최종 학습에 활용
- 메타 모델: 개별 모델이 만든 예측값을 입력으로 받아 최종적인 예측을 수행

2) 스태킹의 특징

1. **두 종류의 모델 필요**
   - 첫 번째는 개별 모델(기반 모델)
   - 두 번째는 개별 모델의 예측값을 학습하는 메타 모델
2. **구조**
   - 여러 개별 모델이 예측한 값을 모아 스태킹 데이터 세트를 만듦
   - 이 데이터를 이용해 최종 메타 모델이 학습 및 예측을 수행
3. **활용성**
   - 실제 모델링에서는 자주 사용되지는 않지만 캐글 같은 대회에서는 성능 향상을 위해 많이 사용됨
   - 보통 2~3개의 개별 모델만 결합해도 효과를 볼 수 있음

3) 스태킹 과정

- 데이터 → 여러 개별 모델 학습 → 각 모델의 예측값 생성 → 예측값들을 모아 새로운 데이터 세트 생성 → 메타 모델 학습 → 최종 예측 수행

4) CV 세트 기반의 스태킹

- 단순 스태킹은 과적합 문제가 발생할 수 있음. 이를 보완하기 위해 CV 세트 활용
- CV 세트 기반 스태킹은 데이터를 여러 개의 폴드로 나누어 학습/검증 과정을 반복하면서 메타 데이터 세트를 생성

* 두 가지 스텝

1스텝 1

- 각 개별 모델이 원본 데이터를 기반으로 학습/검증하여 예측값을 생성
- 이 예측값들이 모여 스태킹용 학습 데이터가 됨

스텝 2

- 스텝 1에서 만들어진 데이터를 메타 모델의 학습용 데이터로 사용
- 최종적으로 원본 테스트 데이터를 대상으로 예측을 수행

※그림 설명 (순차적 과정)

1. 첫 번째 반복

- 데이터를 여러 개의 폴드로 나눔 → 일부는 학습용, 일부는 검증용으로 사용
- 검증 폴드에서 개별 모델이 예측한 값을 모아 저장 → 메타 데이터에 활용

2. 두 번째 반복

- 폴드를 바꿔 같은 과정을 반복 → 또 다른 예측 결과 생성

3. 세 번째 반복

- 남은 폴드로 동일한 과정을 수행
- 이렇게 반복하면서 얻은 예측값들을 평균 내어 최종 메타 데이터 세트 완성

5) 스태킹 최종 구조

- 스텝 1: 개별 모델들이 CV 기반으로 학습/검증하여 예측값 생성
- 스텝 2: 이 예측값들을 합쳐 최종 메타 모델 학습 데이터 구성
- 마지막에 메타 모델을 학습시켜 최종 예측을 수행