4장 평가 part2

# Ch	4
날짜	@2025년 9월 29일
카테고리	개념 정리

4.5 ~ 4.8 , 4.10~11 개념 정리 & 필사

4.5 GBM (Gradient Boosting Machine)

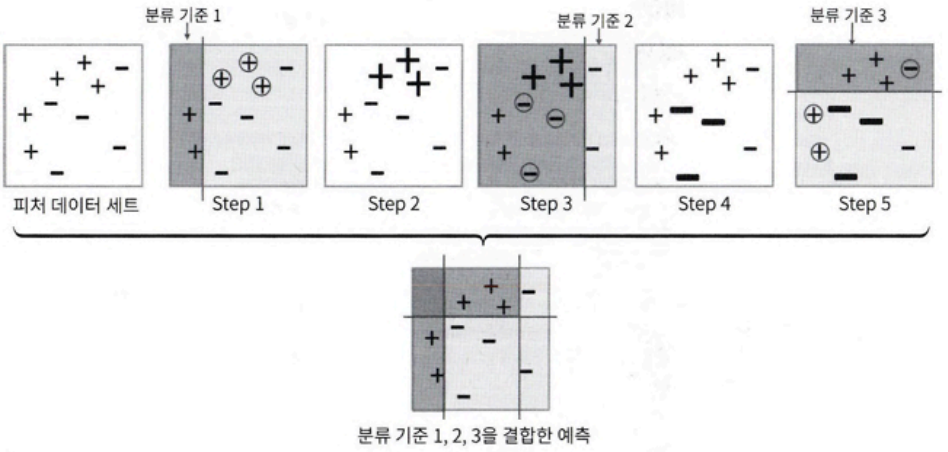
GBM 개요 및 실습

1. 개요

- **부스팅(Boosting)** 알고리즘의 한 종류
- 여러 개의 **약한 학습기(Decision Tree)**를 순차적으로 학습하면서 성능을 점진적으로 개선
- 대표적인 부스팅 알고리즘
  - **AdaBoost**: 잘못 예측한 데이터에 가중치를 부여
  - **GBM**: 가중치 업데이트를 **경사 하강법(Gradient Descent)** 으로 수행

2. GBM 학습 원리

- **오차 = 실제값 - 예측값**
- 오차를 줄이는 방향으로 **경사 하강법**을 적용해 가중치 업데이트
- 여러 트리를 순차적으로 학습하여 **잔차(Residual)를 줄여 나감**
- 분류(Classification)와 회귀(Regression) 모두 가능



✅ **장점**

- 일반적으로 랜덤 포레스트보다 예측 성능이 좋은 경우가 많음

⚠️ **단점**

- 학습 시간이 오래 걸림 (병렬 처리 불가 → 대규모 데이터셋에서 비효율적)
- 하이퍼 파라미터 튜닝 필요

3. 실습 내용

- 사이킷런 `GradientBoostingClassifier` 사용
- 사용자 행동 데이터(`get_human_dataset()`)로 분류 실험 수행
- 기본 파라미터로 약 **93.89% 정확도** 달성 (랜덤 포레스트보다 성능 ↑)
- but 학습 시간은 상대적으로 오래 걸림 (nnn sec, n min 이상)

GBM 하이퍼 파라미터 소개

- **loss**
  - 경사 하강법에서 사용할 **비용 함수**
  - 기본값: `'deviance'` (대부분 그대로 사용)
- **learning_rate**
  - Weak learner가 순차적으로 오류를 보정할 때 적용하는 **학습률 계수** (0~1)
  - 기본값: `0.1`
  - 특성:
    - 값이 작을수록 → 업데이트가 작아지고 반복(n_estimators)을 많이 해야 함 → 성능은 좋아질 수 있으나 시간이 오래 걸림
    - 값이 클수록 → 빠른 학습 가능하나 최적점을 지나쳐 성능 저하 가능
  - **n_estimators와 상호 보완적 조합** 필요
- **n_estimators**
  - **Weak learner(트리)의 개수**
  - 많을수록 일정 수준까지 성능 향상 가능
  - 기본값: `100`

- 개수가 많아질수록 수행시간 증가

- **subsample**
  - Weak learner가 학습에 사용하는 **데이터 샘플링 비율**
  - 기본값: `1.0` (전체 데이터 사용)
  - 1보다 작게 설정하면 과적합 완화 가능

<정리>

- **learning_rate ↓, n_estimators ↑** → 성능 개선 가능 (단, 수행시간 ↑)
- **GBM 장점**: 과적합에 강하고 뛰어난 예측 성능
- **GBM 단점**: 수행시간이 오래 걸림
- GBM 기반으로 발전한 고성능 라이브러리: **XGBoost, LightGBM**

4.6 XGBoost (eXtra Gradient Boost)

XGBoost 개요

- 트리 기반 앙상블 학습에서 가장 각광받는 알고리즘 중 하나
- Kaggle 등 대회에서 상위권 팀들이 많이 사용 → 널리 알려짐
- GBM(Gradient Boosting Machine) 기반이지만, **느린 수행 속도**와 **과적합 규제 부재** 문제를 개선
- 병렬 CPU 환경에서 학습 가능 → GBM보다 빠름

✅ XGBoost 주요 장점

항목	설명
뛰어난 예측 성능	분류와 회귀 영역에서 일반적으로 높은 성능
GBM 대비 빠른 수행 시간	병렬 수행과 다양한 최적화 기능으로 GBM보다 빠름 (단, 랜덤 포레스트보다 빠른 것은 아님)
과적합 규제(Regularization)	자체 규제 기능을 통해 과적합에 강함
Tree pruning (나무 가지치기)	불필요한 분할을 줄여 모델 효율성 향상
내장 교차 검증	반복 수행 중 교차 검증을 통해 최적 반복 횟수 자동 결정
조기 중단(Early Stopping)	평가 성능이 향상되지 않으면 학습을 조기에 종료
결손값 처리 지원	결손값을 자체적으로 처리 가능

구현 및 파이썬 지원

- XGBoost 핵심 라이브러리: **C/C++ 기반**
- 파이썬 패키지명: `xgboost`
- 두 가지 형태의 API 제공:
  1. **파이썬 네이티브 XGBoost API**
     - XGBoost 고유 프레임워크 기반
     - 사이킷런 호환 X 아님
     - `fit()`, `predict()` 같은 사이킷런 유틸리티 사용 불가
  2. **사이킷런 래퍼 XGBoost 모듈**

- 사이킷런과 호환
- 클래스: `XGBClassifier`, `XGBRegressor`
- 다른 사이킷런 Estimator처럼 활용 가능 (`fit()`, `predict()`, `cross_val_score`, `GridSearchCV`, `Pipeline` 등 사용 가능)

<정리>

- XGBoost = **GBM의 단점 보완 + 고성능 부스팅 알고리즘**
- 장점: 높은 성능, 빠른 수행 시간, 과적합 방지 기능, 조기 중단, 결손값 처리
- 실무에서는 **사이킷런 래퍼 모듈** (`XGBClassifier`, `XGBRegressor`)을 주로 활용

XGBoost 설치하기

`pip install xgboost=1.5.0`

```
import xgboost as xgb
from xgboost import XGBClassifier
```

파이썬 래퍼 XGBoost 하이퍼 파라미터

일반 파라미터

파라미터	설명	기본값
booster	부스팅 모델 선택: `gbtree` (트리 기반), `gblinear` (선형 모델)	gbtree
silent	출력 메시지 표시 여부 (`0`: 표시, `1`: 숨김)	0
nthread	CPU 실행 스레드 개수 (기본: 모든 코어 사용)	전체 스레드

부스터 파라미터

파라미터	설명	기본값 / 범위
eta (learning_rate)	학습률. 낮을수록 보수적인 학습 (GBM의 learning_rate)	0.3 (Python wrapper) 0.1 (Scikit-learn wrapper) 일반적으로 0.01 ~ 0.2
num_boost_rounds	부스팅 반복 횟수 (GBM의 n_estimators와 동일)	-
min_child_weight	추가 분할을 위해 필요한 최소 가중치 합. 값이 클수록 분할 억제 → 과적합 방지	1
gamma (min_split_loss)	분할을 위한 최소 손실 감소 값. 값이 클수록 과적합 방지	0
max_depth	트리 최대 깊이. 과적합 방지 위해 보통 3~10 사용	6
subsample	학습에 사용하는 데이터 샘플링 비율. 과적합 방지	1 (0.5~1 권장)
colsample_bytree	트리 학습 시 사용하는 피처 비율. 고차원 데이터일 때 유용	1
lambda (reg_lambda)	L2 정규화 항. 값이 클수록 과적합 억제	1
alpha (reg_alpha)	L1 정규화 항. 값이 클수록 과적합 억제	0
scale_pos_weight	불균형 데이터의 클래스 가중치 조정	1

학습 태스크 파라미터

파라미터	설명	주요 값
objective	손실 함수 정의	- binary:logistic (이진 분류) - multi:softmax (다중 분류, `num_class` 필요) - multi:softprob (확률 반환)
eval_metric	평가 지표	- rmse (회귀) - mae (회귀) - logloss (이진 분류) - error (분류 오류율) - merror (다중 분류 오류율) - mlogloss (다중 분류 logloss) - auc (ROC AUC)

과적합 방지 팁

- **eta** ↓ (0.01 ~ 0.1) → 대신 **num_round** ↑
- **max_depth** ↓
- **min_child_weight** ↑
- **gamma** ↑
- **subsample, colsample_bytree** ↓ (트리 복잡도 조절)

파이썬 래퍼 XGBoost 적용 - 위스콘신 유방암 예측

데이터

- **데이터셋**: 위스콘신 유방암 (Breast Cancer)
- **분류 목적**: 악성(malignant, 0) ↔ 양성(benign, 1) 종양 판별
- **구성**: 569개 샘플 (악성 212, 양성 357)
- **특징**: 종양의 크기, 모양 등 30개 feature

steps

1. 데이터 로드 (`sklearn.datasets.load_breast_cancer`)
2. DataFrame 생성 및 레이블 확인
3. 학습(80%), 테스트(20%) 분리 → 학습(90%), 검증(10%) 추가 분리
4. **XGBoost 전용 객체** `DMatrix` 변환
5. 하이퍼 파라미터 설정 및 학습 수행 (`xgb.train`)
6. **조기 중단 (early stopping)** 적용

```
# 하이퍼파라미터 설정
■ max_depth(트리 최대 깊이)는 3.
■ 학습률 eta 는 0.1(XGBClassifier 를 사용할 경우 eta 가 아니라 learning_rate 입니다).
■ 예제 데이터가 0 또는 1 이진 분류이므로 목적함수(objective)는 이진 로지스틱(binarylogistic).
■ 오류 함수의 평가 성능 지표는 logloss.
■ num_rounds(부스팅 반복 횟수)는 400회
```

```
params = {
    'max_depth': 3,
    'eta': 0.05,   # learning_rate
    'objective': 'binary:logistic',
    'eval_metric': 'logloss'
}
num_rounds = 400
```

피처 중요도 시각화

```
import matplotlib.pyplot as plt
%matplotlib inline

fig, ax = plt.subplots(figsize=(10, 12))
plot_importance(xgb_model, ax=ax)
```

- `plot_importance()` → 피처 중요도 막대그래프
- 기본 지표: **f-score** (트리 분할 시 사용 빈도)
- DataFrame 아닌 **numpy 기반 학습 시** → `f0`, `f1`, ... 순서로 표시

<정리>

- **train()**: 반복 학습 + 조기 중단 적용
- **predict()**: 확률 값 반환 → threshold 적용 필요
- **성능**: 매우 우수 (정확도 95% 이상, ROC-AUC 0.99 이상)
- **시각화**: `plot_importance()`, `to_graphviz()` 지원
- **교차 검증**: `xgb.cv()`로 최적 파라미터 탐색 가능
- 이후 실습에서는 **사이킷런 래퍼 XGBoost** 사용

사이킷런 래퍼 XGBoost의 개요 및 적용

개요

- XGBoost 개발팀은 사이킷런과의 호환을 위해 **사이킷런 전용 래퍼**를 제공
- 래퍼 클래스:
  - **분류** `XGBClassifier`
  - **회귀** `XGBRegressor`
- 사이킷런 래퍼는 기존 Estimator와 동일하게 `fit()`, `predict()`, `predict_proba()` 사용 가능
- **GridSearchCV, Pipeline** 등 사이킷런 유틸리티와 호환

하이퍼파라미터 대응

xgboost 파라미터	XGBClassifier 파라미터
eta	learning_rate

xgboost 파라미터	XGBClassifier 파라미터
sub_sample	subsample
lambda	reg_lambda
alpha	reg_alpha
n_estimators	n_estimators
num_boost_round	n_estimators

기본 학습 및 예측

```
# 사이킷런 래퍼 XGBoost 클래스인 XGBClassifier 임포트
from xgboost import XGBClassifier

# 모델 생성
 # Warning 메시지를 없애기 위해 eval.metric 값을 XGBClassifier 생성 인자로 입력.
xgb_wrapper = XGBClassifier(n_estimators=400, learning_rate=0.05, max_depth=3, eval_metric='loglo
ss')

# 학습
xgb_wrapper.fit(X_train, y_train, verbose=True)

# 예측
w_preds = xgb_wrapper.predict(X_test)
w_pred_proba = xgb_wrapper.predict_proba(X_test)[:, 1]

# 예측 성능 평가
get_clf_eval(y_test, w_preds, w_pred_proba)
```

→ 파이썬 래퍼보다 조금 더 좋은 성능 결과 관찰 가능 (학습 데이터 손실이 적음)

조기 중단 적용

- `fit()` 에 파라미터 입력으로 조기 중단 가능
  - `early_stopping_rounds` : 성능 향상 반복 수 기준
  - `eval_metric` : 평가 지표 (`logloss` 등)
  - `eval_set` : 성능 평가용 데이터 [(학습, 레이블), (검증, 레이블)]

```
import xgboost as xgb

# DMatrix로 변환
dtrain = xgb.DMatrix(X_tr, label=y_tr)
dval = xgb.DMatrix(X_val, label=y_val)
dtest = xgb.DMatrix(X_test, label=y_test)

# 하이퍼파라미터 설정
params = {
    'max_depth': 3,
    'eta': 0.05,
    'objective': 'binary:logistic',
    'eval_metric': 'logloss'
}
```

```
num_rounds = 400

# 학습 (early stopping 적용)
evals = [(dtrain, 'train'), (dval, 'eval')]
xgb_model = xgb.train(params, dtrain, num_boost_round=num_rounds,
                      evals=evals, early_stopping_rounds=10)

# 예측
pred_probs = xgb_model.predict(dtest)
preds = [1 if x > 0.5 else 0 for x in pred_probs]
get_clf_eval(y_test, preds, pred_probs)
```

피처 중요도 시각화

```
import xgboost as xgb
import matplotlib.pyplot as plt

# DMatrix 변환
dtrain = xgb.DMatrix(X_tr, label=y_tr)
dval = xgb.DMatrix(X_val, label=y_val)

# 파라미터
params = {'max_depth': 3, 'eta': 0.05, 'objective': 'binary:logistic'}

# 학습
evals = [(dtrain, 'train'), (dval, 'eval')]
booster = xgb.train(params, dtrain, num_boost_round=400, evals=evals, early_stopping_rounds=10)

# 피처 중요도 시각화
fig, ax = plt.subplots(figsize=(10, 12))
xgb.plot_importance(booster, ax=ax)
plt.show()
```

4.7 LightGBM

: XGBoost보다 **빠르고 메모리 효율적**이며, **대규모 데이터셋과 병렬 처리에 적합**

LightGBM 개요

1. LightGBM 소개

- XGBoost와 함께 **부스팅 계열 알고리즘**에서 대표적으로 사용되는 모델
- 장점: 학습 속도가 빠르고 메모리 사용량이 적음
- XGBoost 대비:
  - 학습 시간 단축
  - 메모리 효율
  - 카테고리형 피처 자동 변환 및 최적 분할 (원-핫 인코딩 불필요)

    - GPU 지원 가능
- 단점: 작은 데이터셋(약 10,000건 이하)에서는 과적합 발생 가능

2. 학습 방식

- **트리 분할 방식**:
    - 기존 GBM: **Level Wise** (균형 트리 분할) → 오버피팅 방지, 깊이 최소화
    - LightGBM: **Leaf Wise** (리프 중심 분할) → 최대 손실값을 가진 리프 노드를 반복 분할, 트리 깊이가 깊어지지만 손실 최소화 가능
- 결과: 학습 반복 시 예측 오류 손실 최소화 가능, 다소 비대칭적인 트리 생성

3. XGBoost 대비 장점

1. 학습과 예측 속도 빠름
2. 메모리 사용량 적음
3. 카테고리형 피처 자동 처리
4. 대용량 데이터에 대한 높은 예측 성능
5. 병렬 연산 및 GPU 지원

4. Python

- 패키지 명: `lightgbm`
- 두 가지 래퍼 제공:
    1. **파이썬 래퍼**: 초기 구현
    2. **사이킷런 래퍼**:
        - **LGBMClassifier**: 분류
        - **LGBMRegressor**: 회귀
- 사이킷런 기반 Estimator 상속 → `fit()`, `predict()` 가능
- 사이킷런 유틸리티(예: GridSearchCV, Pipeline) 사용 가능

LightGBM 설치

```
import lightgbm
from lightgbm import LGBMClassifier
```

LightGBM 하이퍼 파라미터

주요 파라미터

구분	파라미터	기본값	설명	사이킷런 래퍼명 / XGBoost 비교
학습 반복	`num_iterations`	100	학습할 트리 개수. 많을수록 성능 ↑, 과적합 주의	`n_estimators` (사이킷런 호환)
학습률	`learning_rate`	0.1	각 부스팅 스텝에서 가중치를 업데이트하는 학습률	동일
트리 깊이	`max_depth`	-1	트리의 최대 깊이, -1이면 제한 없음. Leaf Wise 트리 사용 시 깊이 깊음	동일

구분	파라미터	기본값	설명	사이킷런 래퍼명 / XGBoost 비교
리프 최소 데이터	min_data_in_leaf	20	리프 노드가 되기 위한 최소 데이터 수, 과적합 제어	min_child_samples
리프 개수	num_leaves	31	하나의 트리가 가질 수 있는 최대 리프 개수	동일
부스팅 방식	boosting	gbdt	트리 생성 알고리즘 지정: gbdt, rf 등	동일
샘플링 비율	bagging_fraction	1.0	과적합 방지용 학습 데이터 샘플링 비율	subsample
피처 샘플링 비율	feature_fraction	1.0	과적합 방지용 피처 무작위 선택 비율	colsample_bytree
L2 규제	lambda_l2	0.0	과적합 방지용 L2 규제	reg_lambda
L1 규제	lambda_l1	0.0	과적합 방지용 L1 규제	reg_alpha

Learning Task 파라미터

손실함수	objective	–	최소화할 손실함수 지정, 회귀/분류 등	XGBoost와 동일

하이퍼 파라미터 튜닝 방안

1. 모델 복잡도

파라미터	역할	튜닝 전략
num_leaves	하나의 트리가 가질 수 있는 최대 리프 개수	증가시키면 정확도 ↑, 트리 깊이 ↑ → 과적합 ↑ → 적절히 조정
min_data_in_leaf (min_child_samples)	리프 노드가 되기 위한 최소 데이터 수	크게 설정하면 트리 깊이 억제, 과적합 감소
max_depth	트리 최대 깊이 제한	num_leaves 와 min_data_in_leaf 와 조합해 과적합 제어

2. Learning Task - 과적합 제어

파라미터	역할	튜닝 전략
learning_rate	부스팅 단계에서 학습률	작게 설정 → n_estimators ↑ → 안정적 학습, 과적합 ↓
n_estimators	학습 트리 반복 횟수	learning_rate 와 함께 조절, 너무 크게 하면 과적합 주의
reg_lambda , reg_alpha	L2, L1 정규화	과적합 감소 용도로 조절
colsample_bytree	학습에 사용할 피처 비율	낮추면 과적합 감소
subsample	학습 데이터 샘플링 비율	낮추면 과적합 감소

파이썬 래퍼 LightGBM과 사이킷런 래퍼 XGBoost, LightGBM 하이퍼 파라미터 비교

유형	파라미터명	파이썬 래퍼 LightGBM	사이킷런 래퍼 LightGBM	사이킷런 래퍼 XGBoost
반복/트리 수	n_estimators	num_iterations	n_estimators	n_estimators
학습률	learning_rate	learning_rate	learning_rate	learning_rate
최대 깊이	max_depth	max_depth	max_depth	max_depth
최소 리프 데이터	min_data_in_leaf	min_data_in_leaf	min_child_samples	N/A
배깅 샘플 비율	bagging_fraction	bagging_fraction	subsample	subsample
피처 샘플 비율	feature_fraction	feature_fraction	colsample_bytree	colsample_bytree
L2 정규화	lambda_l2	lambda_l2	reg_lambda	reg_lambda
L1 정규화	lambda_l1	lambda_l1	reg_alpha	reg_alpha
조기 종료	early_stopping_round	early_stopping_round	early_stopping_rounds	early_stopping_rounds

유형	파라미터명	파이썬 래퍼 LightGBM	사이킷런 래퍼 LightGBM	사이킷런 래퍼 XGBoost
최대 리프 수	num_leaves	num_leaves	num_leaves	N/A
최소 헤시안 합	min_sum_hessian_in_leaf	min_sum_hessian_in_leaf	min_child_weight	min_child_weight

LightGBM 적용 - 위스콘신 유방암 예측

```
# LightGBM과 기타 라이브러리 임포트
from lightgbm import LGBMClassifier, early_stopping, log_evaluation, plot_importance
import pandas as pd
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# 데이터 로드
dataset = load_breast_cancer()
cancer_df = pd.DataFrame(data=dataset.data, columns=dataset.feature_names)
cancer_df['target'] = dataset.target

# Feature / Label 분리
X_features = cancer_df.iloc[:, :-1]
y_label = cancer_df.iloc[:, -1]

# 학습용 / 테스트용 데이터 분리
X_train, X_test, y_train, y_test = train_test_split(
    X_features, y_label, test_size=0.2, random_state=156
)

# 학습 데이터를 다시 학습/검증용으로 분리
X_tr, X_val, y_tr, y_val = train_test_split(
    X_train, y_train, test_size=0.1, random_state=156
)

# LGBMClassifier 생성
lgbm_wrapper = LGBMClassifier(n_estimators=400, learning_rate=0.05)

# 학습 및 조기 중단 적용
evals = [(X_tr, y_tr), (X_val, y_val)]
lgbm_wrapper.fit(
    X_tr, y_tr,
    eval_set=evals,
    eval_metric="logloss",
    callbacks=[early_stopping(stopping_rounds=50), log_evaluation(10)]
)

# 예측 수행
preds = lgbm_wrapper.predict(X_test)
pred_proba = lgbm_wrapper.predict_proba(X_test)[:, 1]

# 예측 성능 평가 함수(get_clf_eval)를 사용
```

```
get_clf_eval(y_test, preds, pred_proba)

# 피처 중요도 시각화
fig, ax = plt.subplots(figsize=(10, 12))
plot_importance(lgbm_wrapper, ax=ax)
plt.show()
```

4.8 베이지안 최적화 기반의 HyperOpt를 이용한 하이퍼 파라미터 튜닝

1. Grid Search 방식의 한계

- Grid Search는 모든 하이퍼파라미터 조합을 반복 학습 → 조합 수가 많으면 시간이 기하급수적으로 증가
- ex) LightGBM 6개 하이퍼파라미터

```
params = {
 'max_depth' = [10Z 20, 30, 40, 50], 'nㄴm_leaves'= [ 35, 45, 55, 65],
 'colsample_bytree'=[0.5, 0.6, 0.7, 0.8, 0.9], 1 subsample'= [0.5, 0.6, 0.7, 0.8, 0.9],
,min_child_weight'= [10, 20, 30, 40], reg_alpha=[0.01, 0.05, 0.1]
 }
```

```
max_depth = [10, 20, 30, 40, 50]      → 5개
num_leaves = [35, 45, 55, 65]         → 4개
colsample_bytree = [0.5,0.6,0.7,0.8,0.9] → 5개
subsample = [0.5,0.6,0.7,0.8,0.9]     → 5개
min_child_weight = [10,20,30,40]      → 4개
reg_alpha = [0.01,0.05,0.1]           → 3개
```

- 조합 수 = 5 × 4 × 5 × 5 × 4 × 3 = **6000회 학습 !!!**
- 결과: 실무 대규모 데이터에서는 최적화 시간이 매우 길어짐

베이지안 최적화 개요

- 블랙박스 함수(함수식 모름)에서 **최대/최소 값을 효율적으로 찾는 최적화 기법**
- 단순 반복 대입보다 훨씬 적은 시도로 최적값 탐색 가능
- 개념 예시: `f(x, y) = 2x - 3y`
  - 함수식 알면 쉽게 최댓값 찾음
  - 함수식 모르면 순차 대입으로 찾기 어려움 → 베이지안 최적화 필요

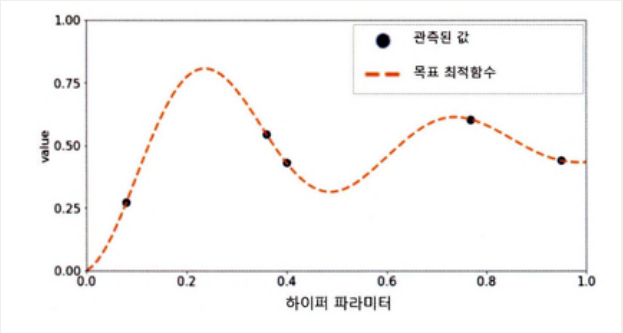
원리

- **베이지안 확률 기반** → 새로운 샘플 관측 시 사후 확률을 개선
- 하이퍼파라미터 튜닝에서는
  - 입력값 = 하이퍼파라미터
  - 목표값 = 모델 성능
- 구성 요소:

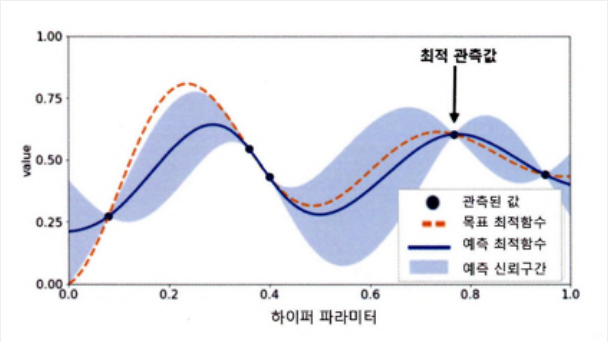
1. **대체 모델(Surrogate Model)**
    - 최적 함수 추정
    - HyperOpt는 **TPE(Tree-structured Parzen Estimator)** 사용
2. **획득 함수(Acquisition Function)**
    - 대체 모델 기반으로 다음 관측할 하이퍼파라미터 계산

⭐베이지안 최적화 단계⭐

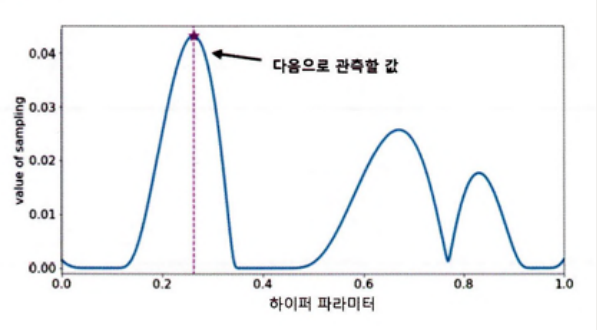
1. **랜덤 샘플링** → 초기 하이퍼파라미터 입력 및 성능 관측



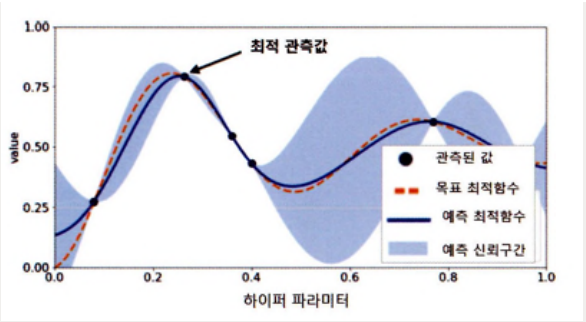
2. **대체 모델 추정** → 관측값 기반 최적 함수 추정
    - 신뢰 구간 표시 → 추정 함수의 불확실성 표현



3. **획득 함수 계산** → 다음 관측할 하이퍼파라미터 선택
    - 이전 최적 관측값보다 높은 가능성 지점 탐색



4. **대체 모델 갱신** → 관측값 반영 후 최적 함수 재추정



⇒ Step 3~4 반복 → 점점 정확한 최적 함수 추정 가능

HyperOpt 사용하기

- HyperOpt 사용 순서:
  1. 검색 공간 정의 (`hp` 모듈)
  2. 목적 함수 정의 (`fn`)
  3. `fmin()`으로 최소 반환값을 가지는 최적 입력값 탐색
- `TPE` 알고리즘 기반으로 베이지안 최적화 수행
- `max_evals`를 늘리면 탐색 성능 향상

1. HyperOpt 설치

- HyperOpt는 **베이지안 최적화**를 기반으로 머신러닝 모델 하이퍼파라미터를 효율적으로 튜닝 가능
- 목적 함수 반환값의 **최솟값**을 찾는 방식

```
pip install hyperopt
```

2. 주요 구성요소

1. **검색 공간(Search Space)**
   - 하이퍼파라미터 후보 범위를 정의
   - `hp` 모듈 사용, 딕셔너리 형태
   - 예시: x, y 두 변수 설정

```
from hyperopt import hp

search_space = {
    'x': hp.quniform('x', -10, 10, 1),
    'y': hp.quniform('y', -15, 15, 1)
}
```

2. **검색 공간 생성/제공 함수**

- `hp.quniform(label, low, high, q)` low~high까지 q 간격 값
- `hp.uniform(label, low, high)` low~high 구간 균등분포

- `hp.randint(label, upper)` 0~upper 랜덤 정수
- `hp.loguniform(label, low, high)` exp(uniform(low, high)) 값
- `hp.choice(label, options)` 문자열/숫자 혼합 리스트 선택

3. **목적 함수(Objective Function)**

- 검색 공간 딕셔너리를 입력받아 반환값 계산
- HyperOpt는 숫자 반환 또는 `{'loss': 값, 'status': STATUS_OK}` 형태 가능

```
from hyperopt import STATUS_OK

def objective_func(search_space):
    x = search_space['x']
    y = search_space['y']
    retval = x**2 - 20*y
    return retval
```

3. fmin() 함수로 최적값 탐색

```
from hyperopt import fmin, tpe, Trials
import numpy as np

trial_val = Trials()  # 관측값 저장 객체
best_01 = fmin(
    fn=objective_func,
    space=search_space,
    algo=tpe.suggest,  # TPE(Tree-structured Parzen Estimator)
    max_evals=5,       # 입력값 시도 횟수
    trials=trial_val,
    rstate=np.random.default_rng(seed=0)  # 랜덤 시드
)
print(best_01)
```

- **state**: 실행 시 동일 결과 재현용. 일반적으로는 생략 가능.
- `max_evals`를 늘리면 더 최적값에 근사

4. 베이지안 최적화 반복 효율 비교

Trials 객체

```
trial_val = Trials()
best_02 = fmin(
    fn=objective_func,
    space=search_space,
    algo=tpe.suggest,
    max_evals=20,
    trials=trial_val,
    rstate=np.random.default_rng(seed=0)
```

```
)
print(best_02)
```

- `fmin()` 수행 시 입력값과 반환값을 저장하는 객체
- 주요 속성:
  1. **results**
     - 함수 반복 수행 시 반환값 저장
     - 리스트 형태, 각 원소 = `{'loss': 반환값, 'status': 상태}`

     ```
     print(trial_val.results)
     ```

     - 예: max_evals=20 → 20개의 딕셔너리 원소
  2. **vals**
     - 반복 수행 시 입력된 하이퍼파라미터 값 저장
     - 딕셔너리 형태: `{ '입력변수명': [반복마다 입력값 리스트] }`

     ```
     print(trial_val.vals)
     ```

- `results`와 `vals`를 DataFrame으로 변환
- 각 반복 수행의 입력값과 반환값 확인 가능

```
import pandas as pd

# results에서 loss 값 추출
losses = [loss_dict['loss'] for loss_dict in trial_val.results]

# DataFrame 생성
result_df = pd.DataFrame({
    'x': trial_val.vals['x'],
    'y': trial_val.vals['y'],
    'losses': losses
})

result_df
```

- DataFrame 컬럼:
  - `x`: 반복마다 입력된 x 값
  - `y`: 반복마다 입력된 y 값
  - `losses`: 목적 함수 반환값

<정리>

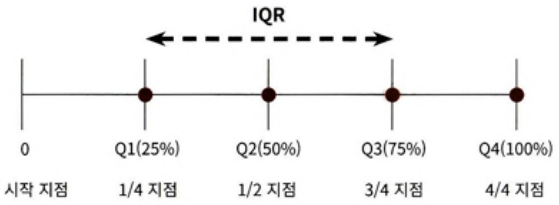
- HyperOpt는 **반복 횟수를 크게 줄이면서** 목적 함수 최솟값 근사 가능
- Trials 객체를 통해 **반복 시 입력값과 결과값 추적** 가능
- DataFrame 변환으로 반복 결과를 직관적으로 확인 가능

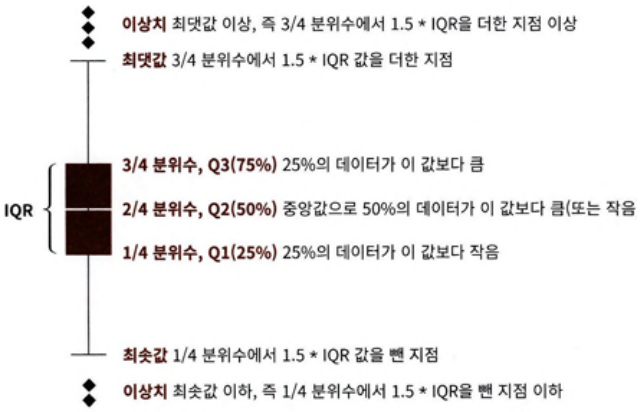
- 다음 단계: HyperOpt를 이용하여 **실제 ML 모델 하이퍼파라미터 최적화** 적용

HyperOpt를 이용한 XGBoost 하이퍼 파라미터 최적화

1. **HyperOpt 적용 방법**
   - 검색 공간(Search Space) 설정 → 목적 함수(Objective Function) 작성 → `fmin()`으로 최적 입력값 탐색
   - XGBoost 적용 시 주의:
     1. 정수형 하이퍼파라미터(max_depth, min_child_weight)는 `int()`로 형변환
     2. HyperOpt는 최소화 기준 → 정확도와 같이 클수록 좋은 값은 `1` 곱해서 반환
2. **데이터 준비**
   - 위스콘신 유방암 데이터
   - 학습/검증/테스트 분리
   - 예: 학습 80%, 테스트 20% → 학습 데이터를 다시 학습/검증 90/10 분리
3. **검색 공간 예시**
   - max_depth: 5~20 (정수)
   - min_child_weight: 1~2 (정수)
   - learning_rate: 0.01~0.2 (실수)
   - colsample_bytree: 0.5~1 (실수)
4. **목적 함수**
   - 교차검증 기반 정확도 평가
   - 정수형 파라미터는 int() 변환
   - 정확도는 클수록 좋음 → -1 곱해서 반환
5. **최적화 수행**
   - fmin(fn=objective_func, space=xgb_search_space, algo=tpe.suggest, max_evals=50, trials=Trials(), rstate=np.random.default_rng(seed=9))
6. **결과 적용**
   - 최적 하이퍼파라미터를 XGBClassifier에 적용
   - 조기 중단 적용, n_estimators=400
   - 성능 평가

4.10 분류 실습 - 캐글 신용카드 사기 검출





4.11 스태킹 앙상블

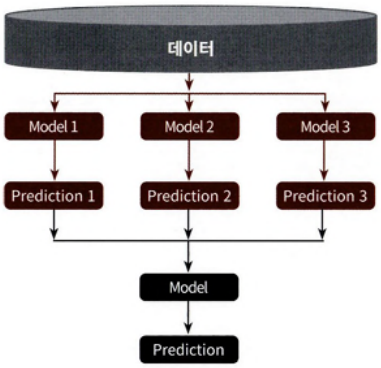
1. 스태킹 앙상블 개념

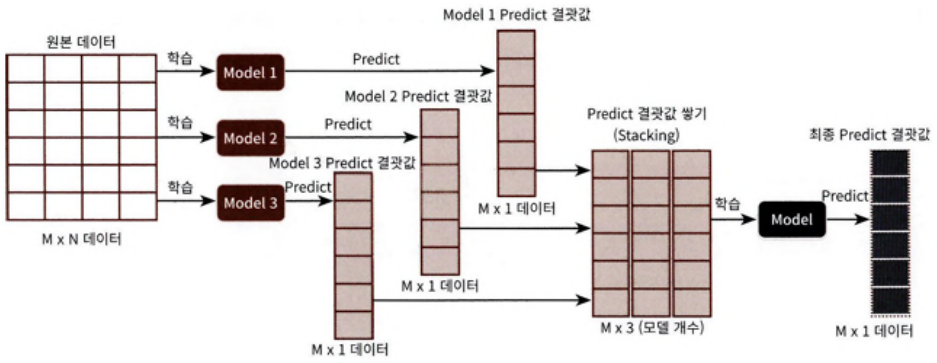
- **기본 아이디어**
  : 여러 개별 모델(Base Model)의 예측 결과를 **메타 모델(Meta Model)**의 입력으로 사용하여 최종 예측을 수행
- **차이점**
  - 배깅(Bagging), 부스팅(Boosting)과 달리 단순 결합이 아니라 **예측값을 다시 학습 데이터로 사용**
  - 여러 모델의 장점을 조합해 성능 향상을 노림
- **필요한 모델**
  1. **Base 모델**: 개별 알고리즘으로 원래 학습 데이터를 학습하고 예측
  2. **Meta 모델**: Base 모델의 예측값으로 학습하고 최종 예측 수행

2. 데이터 흐름

<가정>

- 학습 데이터: M개의 샘플 × N개의 피처
- Base 모델: 3개





Step 1: Base 모델 학습

- 각 Base 모델이 원본 학습 데이터 $(M \times N)$를 학습
- 각 모델은 학습 후 **예측값 M개** 생성 (각 샘플당 하나의 레이블 예측)

Sample	Base1	Base2	Base3
1	0	1	0
2	1	1	1
3	0	0	0
...	...	...	...
M	1	0	1

Step 2: 스태킹 데이터 생성

- Base 모델들의 예측값을 **세로로 쌓아 새로운 데이터 세트** 생성
- 이 새로운 데이터 세트의 **피처 수 = Base 모델 수, 샘플 수 = 원본 학습 데이터 M**

Step 3: Meta 모델 학습

- 새로 생성한 스태킹 데이터 세트를 **Meta 모델의 입력**으로 사용
- Meta 모델이 최종 예측 수행 → 최종 레이블 출력

> 💡 **특징 및 주의점**
>
> 1. 스태킹은 캐글 등 대회에서 **성능 최적화용**으로 자주 사용
> 2. **2~3개 모델만 결합**해서는 성능 향상이 크게 나타나지 않음
> 3. Base 모델들은 **성능이 비슷하고 다양성이 있는 모델**을 선택하는 것이 좋음
> 4. 반드시 성능 향상이 보장되는 것은 X

기본 스태킹모델

1. 데이터 준비

- 위스콘신 유방암(Breast Cancer) 데이터 세트를 사용
- 학습용 데이터와 테스트용 데이터로 분리

2. 개별 모델 생성 및 학습

- 개별 Base 모델: KNN, Random Forest, Decision Tree, AdaBoost
- 최종 Meta 모델: Logistic Regression

3. 개별 모델 예측 및 정확도 확인

4. 스태킹 데이터 생성

- 개별 모델의 예측값을 칼럼 단위로 합쳐 최종 메타 모델의 입력으로 사용

5. 최종 메타 모델 학습 및 예측

6. 결과 요약

- 개별 모델의 예측값을 스태킹으로 재구성하여 Meta 모델에서 학습 후 예측
- 예제에서는 **최종 정확도 97.37%**로 개별 모델보다 향상
- 주의: 스태킹이 항상 성능을 향상시키는 것은 아님

CV 세트 기반의 스태킹

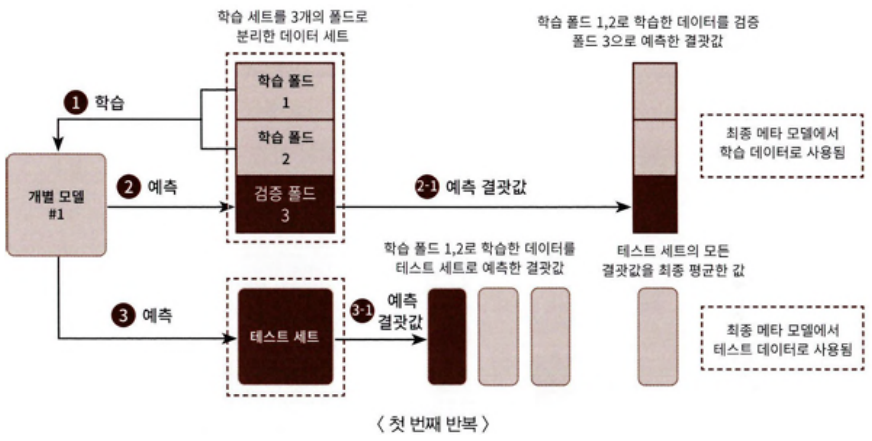
1. 개념

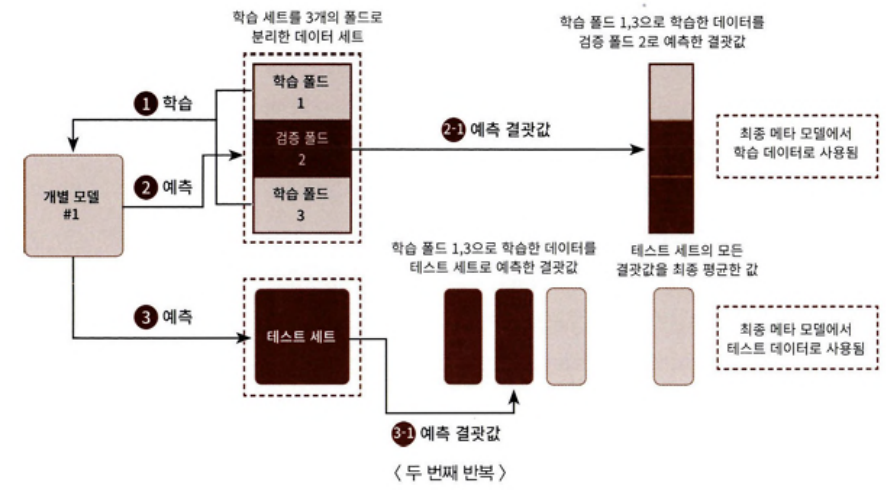
- **목적**: 과적합(overfitting)을 방지하기 위해 메타 모델 학습용 데이터를 **교차검증(CV) 기반으로 생성**
- **차이점**: 이전 기본 스태킹에서는 Meta 모델 학습에 **테스트 레이블 사용** → 과적합 발생 가능
- **CV 기반 스태킹**: Base 모델 예측값을 CV로 나누어 학습/테스트용 데이터 생성 → Meta 모델 학습

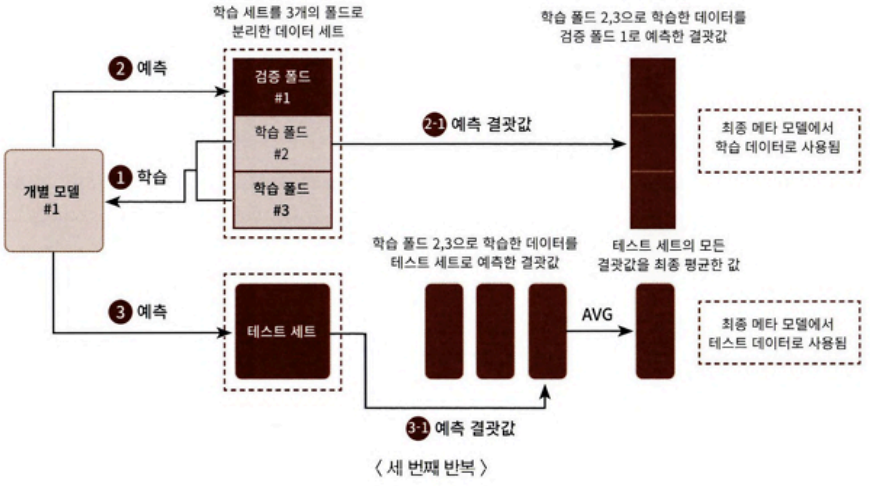
2. CV 스태킹 단계

Step 1: 개별 모델 예측 데이터 생성

- 학습 데이터를 N개의 Fold로 나눔 (예: 3-Fold)
- 각 반복에서:
  1. N-1개 Fold → Base 모델 학습
  2. 남은 1개 Fold → 검증용 예측 데이터 생성 (Meta 모델 학습용)
  3. 원본 테스트 데이터 → Base 모델 예측 → 최종 평균값 계산 (Meta 모델 테스트용)
- 반복 후: Fold별 예측 데이터를 합쳐 Meta 모델 학습용 데이터와 테스트용 데이터 생성

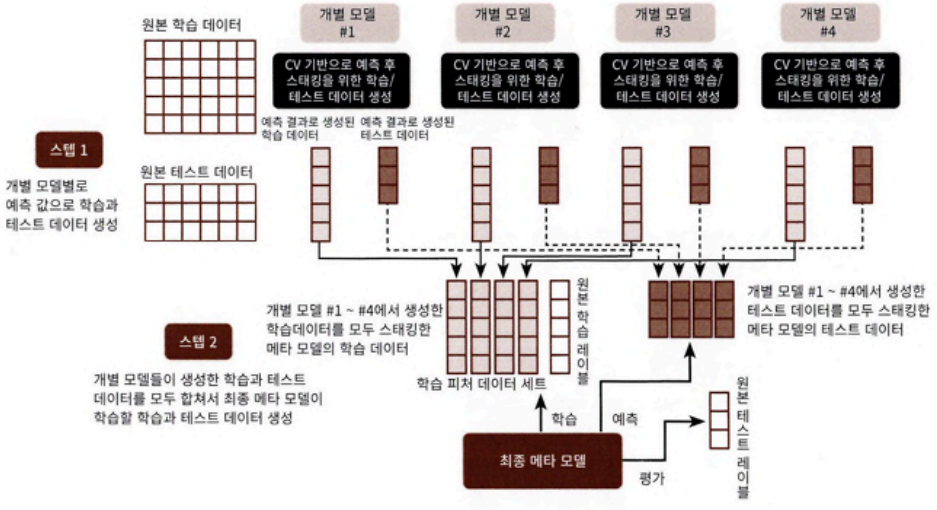






Step 2: 메타 모델 학습 및 최종 예측

- Step 1에서 생성된 **학습/테스트 데이터**를 스태킹 형태로 합침
- Meta 모델 학습: 최종 학습 데이터 + 원본 레이블
- 최종 예측: 최종 테스트 데이터 → 원본 테스트 레이블과 비교하여 정확도 평가



- 핵심: Base 모델에서 CV로 학습/검증을 반복 → Meta 모델 학습/테스트용 데이터 생성
- Step 1: Base 모델 단위로 CV 수행
- Step 2: Base 모델들의 Fold별 결과를 합쳐 최종 Meta 모델 데이터 생성