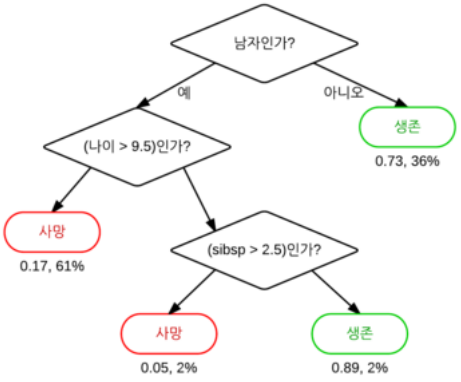
4장. 분류

💡 지도학습의 대표적인 유형인 분류는 학습 데이터로 주어진 데이터의 피처와 레이블값(결정 값, 클래스 값)을 머신러닝 알고리즘으로 학습해 모델을 생성하고, 이렇게 생성된 모델에 새로운 데이터 값이 주어졌을 때 미지의 레이블 값을 예측하는 것

2️⃣ 결정트리

결정트리는 데이터에 있는 규칙을 학습을 통해 자동으로 찾아내 트리(Tree) 기반의 분류 규칙을 만든다. 일반적으로 쉽게 표현하는 방법은 **if/else** 로 스무고개 !!

아래 그림에서 다이아몬드 모양은 규칙 노드를, 타원형은 리프 노드를 뜻한다. 데이터 세트에 피처가 있고 이러한 피처가 결합해 규칙 조건을 만들 때마다 규칙 노드가 만들어지지만, 많은 규칙이 있으면 분류를 결정하는 방식이 복잡해지고 이는 과적합으로 이어지기 쉽다. 즉 트리의 깊이(depth)가 깊어질수록 결정 트리의 예측 성능이 저하될 가능성이 높다.



결정 노드는 정보 균일도가 높은 데이터 세트를 먼저 선택할 수 있도록 규칙 조건을 만든다. 정보의 균일도를 측정하는 대표적인 방법은 정보 이득 지수와 지니 계수가 있다.

- 정보 이득은 엔트로피 개념을 기반으로 한다. 엔트로피는 주어진 데이터 집합의 혼잡도를 의미하는데, 서로 다른 값이 섞여 있으면 엔트로피가 높고, 같은 값이 섞여 있으면 엔트로피가 낮다. 정보 이득 지수는 *1 - 엔트로피 지수* 이다.
- 지니 계수는 경제학에서 불평등 지수를 나타낼 때 사용하는 계수로, 0이 가장 평등하고 1로 갈수록 불평등하다. 머신러닝에서는 지니 계수가 낮을수록 데이터 균일도가 높은 것으로 해석한다.

2-2. 결정 트리 파라미터

[사이킷런]

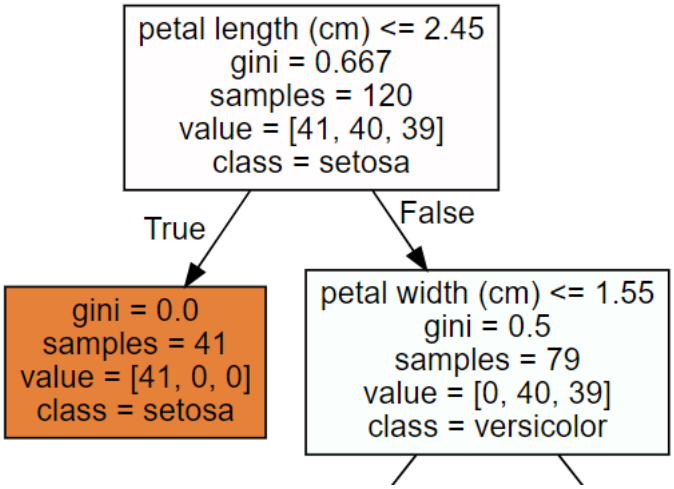
- DecisionTreeClassifier : 분류
  - `min_sample_split` : 노드 분할 위한 최소한의 샘플 데이터 수 (default = 2)
  - `min_sample_leaf` : 분할이 될 경우 왼/오 브랜치 노드에서 가져야할 최소 샘플 데이터 수

- `max_features` : 최대 피처 개수 (default = None)
- `max_depth` : 최대 깊이 규정
- `max_leak_nodes` : 말단 노드 최대 개수

- DecisionTreeRegressor : 회귀
- 결정 트리 구현 : CART (Classification And Regression Tree) 알고리즘 기

2-3. 결정 트리 모델의 시각화

- Graphviz 패키지
  - `export_graphviz()`



- petal length(cm): 피처의 조건이 있는 것은 자식 노드를 만들기 위한 규칙 조건, 조건이 없으면 리프 노드
- gini: 다음의 value=[]로 주어진 데이터 분포에서의 지니 계수
- samples: 현 규칙에 해당하는 데이터 건수
- value: 클래스 값 기반의 데이터 건수, [41, 40, 39]는 Setosa 41개, Vesicolor 40개, Virginica 39개

- `feature_importances_` : 결정 트리 알고리즘이 어떻게 동작하는지 직관적 이해가능

2-4. 결정 트리 과적합(Overfitting)

- `make_classification()` : 분류를 위한 테스트용 데이터 쉽게 만들 수 있도록
- `visualize_boundary()` : 모델이 클래스 값을 예측하는 결정 기준을 색상과 경계로 나타내 모델이 어떻게 데이터 세트를 예측 분류하는지 !

```
import numpy as np

# Classifier의 Decision Boundary를 시각화 하는 함수
def visualize_boundary(model, X, y):
    fig,ax = plt.subplots()

    # 학습 데이타 scatter plot으로 나타내기
    ax.scatter(X[:, 0], X[:, 1], c=y, s=25, cmap='rainbow', edgecolor='k',
               clim=(y.min(), y.max()), zorder=3)
    ax.axis('tight')
    ax.axis('off')
    xlim_start , xlim_end = ax.get_xlim()
    ylim_start , ylim_end = ax.get_ylim()

    # 호출 파라미터로 들어온 training 데이타로 model 학습 .
    model.fit(X, y)
    # meshgrid 형태인 모든 좌표값으로 예측 수행.
    xx, yy = np.meshgrid(np.linspace(xlim_start,xlim_end, num=200),np.linspace(ylim_start,ylim_end, num=200))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    # contourf() 를 이용하여 class boundary 를 visualization 수행.
```

```
    n_classes = len(np.unique(y))
    contours = ax.contourf(xx, yy, Z, alpha=0.3,
                           levels=np.arange(n_classes + 1) - 0.5,
                           cmap='rainbow', clim=(y.min(), y.max()),
                           zorder=1)
```

3️⃣ 앙상블 학습

3-1. 앙상블 학습 개요

앙상블은 여러 개의 분류기를 생성하고 그 예측을 결합함으로써 보다 정확한 최종 예측을 도출하는 기법이다.

[앙상블 학습의 유형]

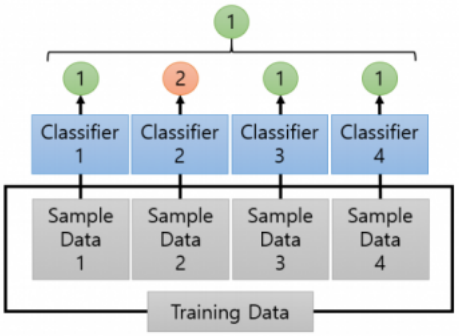
- **보팅(Voting)**
- **배깅(Bagging)**
- **부스팅(Boosting)**

보팅과 배깅은 여러 개의 분류기가 투표를 통해 최종 예측 결과를 결정하는 방식인데, 차이점은 보팅은 서로 다른 알고리즘을 가진 분류기를 결합하는 것이고, 배깅은 각각의 분류기가 모두 같은 유형의 알고리즘 기반이지만, 데이터 샘플링을 서로 다르게 가져가면서 학습을 수행해 보팅을 수행한다. 대표적인 배깅은 랜덤 포레스트 알고리즘이다.

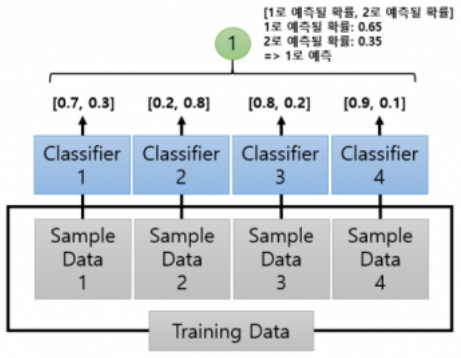
부스팅은 여러 개의 분류기가 순차적으로 학습을 수행하되, 예측이 틀린 데이터에 대해 올바르게 예측하도록 가중치를 부여하면서 학습과 예측을 진행하는 것이다.

3-1. 보팅 유형 - 하드 보팅과 소프트 보팅

일반적으호 하드 보팅보다는 소프트 보팅이 예측 성능이 좋음



Hard Voting



Soft Voting

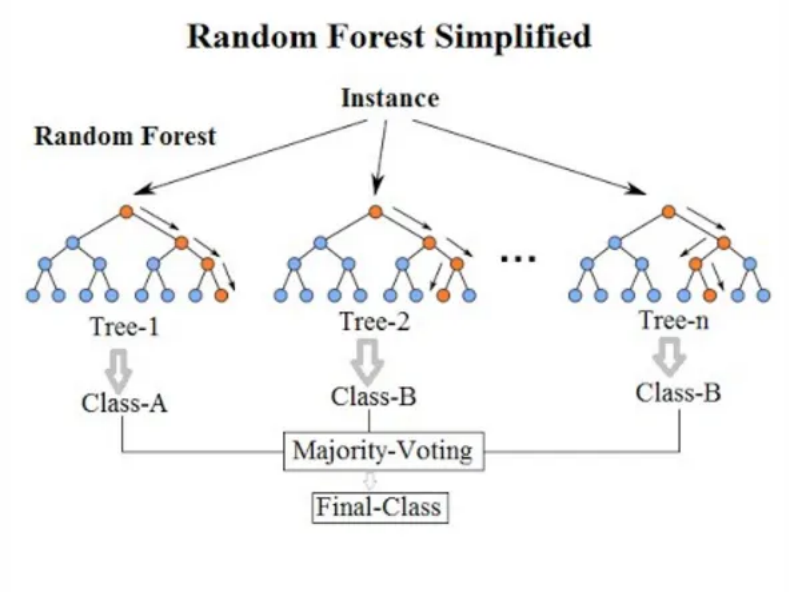
- Hard Voting : 예측한 결괏값들중 다수의 분류기가 결정한 예측값을 최종 보팅 결괏값으로 선정 (다수결의 원칙과 유사)
- Soft Voting : 분류기들의 레이블 값 결정 확률을 모두 더해 이를 평균내서 확률이 가장 높은 레이블 값을 최종 보팅 결괏값을 선정

IT위키
IT에 관한 모든 지식. 함께 만들어가는 깨끗한 위키
https://itwiki.kr/w/앙상블_기법

4️⃣ 랜덤 포레스트

4-1. 랜덤 포레스트의 개요 및 실습

랜덤 포레스트는 여러 개의 결정 트리 분류기가 전체 데이터에서 배깅 방식으로 각자의 데이터를 샘플링해 개별적으로 학습을 수행한 뒤 최종적으로 모든 분류기가 보팅을 통해 예측 결정을 한다.



- RandomForestClassifier 클래스 사용

랜덤 포레스트에서 전체 데이터에서 일부가 중첩되게 샘플링해서 데이터 세트를 만드는데 이를 **부트스트래핑(bootstrapping)** 분할 방식이라 한다. 파라미터 *n_estimators*= 에 넣는 숫자만큼 데이터 세트를 만든다.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

X_train, X_test, y_train, y_test = get_human_dataset()

rf_clf = RandomForestClassifier(random_state=0, max_depth=8)
rf_clf.fit(X_train, y_train)
pred = rf_clf.predict(X_test)
accuracy = accuracy_score(y_test, pred)
print('랜덤 포레스트 정확도:{0:.4f}'.format(accuracy))
```

4-2. 랜덤 포레스트 하이퍼 파라미터 및 튜닝

- `n_estimators` : 랜덤 포레스트에서 결정 트리의 갯수를 지정(디폴트는 10개)
- `max_features` : 결정 트리에 사용된 max_features 파라미터와 같지만, RandomForestClassifier의 기본 max_features는 None이 아닌 auto, 즉 sqrt와 같다. 예를 들어 전체 피처가 16개라면 4개 참조
- `max_depth`나 `min_samples_leaf` 와 같이 결정 트리에서 과적합을 개선하기 위해 사용되는 파라미터가 랜덤 포레스트에서도 똑같이 적용될 수 있다.

5️⃣ GBM

> 💡 여러 개의 약한 학습기를 순차적으로 학습/예측을 하면서 잘못 예측한 데이터에 가중치를 부여해서 오류를 개선해나가는 학습방식

5-1. GBM의 개요 및 실습

1. **AdaBoost**
2. **그래디언트 부스트**

   → 대표적인 차이는 그래디언트 부스트의 가중치 업데이트를 **경사 하강법**을 이용해서 한다는 것

- `GradientBoostingClassifier`

일반적으로 GBM이 랜덤 포레스트보다는 예측 성능이 뛰어난 경우가 많지만 수행시간이 오래 걸리고, 하이퍼 파라미터 튜닝 노력도 더 필요 !!

5-2. GBM 하이퍼 파라미터 소개

- `n_estimators` : weak learner의 갯수이다. 개수가 많을수록 예측 성능이 일정 수준까지는 좋아지지만 수행시간이 오래걸린다. (**디폴트=100**)
- `max_depth`
- `max_features`
- `loss` : 경사 하강법에서 사용할 비용 함수를 지정한다. **디폴트는 *deviance*** 이고 일반적으로 디폴트를 그대로 사용한다.
- `learning_rate` : GBM이 학습을 진행할 때마다 적용하는 함수이다. 0~1 사이의 값을 지정할 수 있으며, **디폴트는 0.1**이다. 너무 작은 값을 설정하면 예측 성능이 높아지지만 수행 시간이 오래걸린다. 반대로 너무 높은 값을 설정하면 예측 성능이 낮아질 가능성이 높지만 수행 시간이 짧아진다.
- `subsample` : weak learner가 학습에 사용하는 데이터 샘플링 비율이다. **디폴트는 1**이고, 예를 들어 0.5로 설정하면 학습데이터의 50%를 기반으로 학습한다는 뜻이다. 과적합이 염려되는 경우 1보다 작은 값으로 설정한다.

두 개의 그래디언트 부스팅 기반 ML 패키지 **XGBoost** 와 **LightGBM**

6️⃣ XGBoost

GBM의 단점인 느린 수행 시간과 과적합 규제 부재 등의 문제를 해결

파이썬 래퍼 XGBoost 하이퍼 파라미터

- 일반 파라미터: 일반적으로 실행 시 스레드의 개수나 silent 모드 등의 선택을 위한 파라미터, 디폴트 파라미터 값을 바꾸는 경우가 거의 없다.
  - **booster**: gbtree(트리 베이스 모델), gblinear(선형 모델) 중 선택, 디폴트는 gbtree
  - **silent**: 디폴트는 0, 출력 메시지를 나타내고 싶지 않을 때 1로 설정
  - **nthread**: CPU의 실행 스레드 개수를 조정, 디폴트는 CPU의 전체 스레드를 다 사용하는 것
- 주요 부스터 파라미터: 트리 최적화, 부스팅, regularization 등과 관련 파라미터 등을 지칭한다.
  - **eta**[default=0.3, alias: learning_rate]: GBM의 *학습률(learning_rate)* 와 같은 파라미터로 0-1 사이의 값을 지정하고 부스팅 스텝을 반복적으로 수행할 때 업데이트되는 학습률 값.
  - **num_boost_rounds**: GBM의 *n_estimators* 와 같은 파라미터
  - **min_child_weight**[default=1]: 트리에서 추가적으로 가지를 나눌지를 결정하기 위해 필요한 데이터의 weight 총합, 값이 클수록 분할을 자제하며 과적합을 조절하기 위해 사용
  - **gamma**[default=0, alias: min_split_loss]: 트리의 리프 노드를 추가로 나눌지를 결정할 최소 손실 감소 값, 해당 값보다 큰 손실이 감소된 경우 리프를 분리하며, 값이 클수록 과적합 감소 효과가 있다.
  - **max_depth**[default=6]: 트리 기반 알고리즘의 *max_depth* 와 같다. 0을 지정하면 깊이에 제한이 없고, *max_depth* 가 높으면 특정 피처 조건에 특화되어 과적합 가능성이 높아지므로 3~10 사이의 값을 적용한다.
  - **sub_sample**[default=1]: GBM의 *subsample* 과 동일하다. 트리가 커져서 과적합되는 것을 제어하기위해 데이터를 샘플링하는 비율을 지정한다. 0에서 1사이의 값이 가능하나 일반적으로 0.5에서 1사이의 값을 사용한다.
  - **colsample_bytree**[default=1]: GBM의 *max_features* 와 유사하다. 트리 생성에 필요한 피처를 임의로 샘플링 하는데 사용되며, 매우 많은 피처가 있을때 과적합을 조정하는데 적용한다.
  - **lambda**[default=1, alias: reg_lambda]: L2 Regularizaion 적용 값으로, 피처 개수가 많을 경우 적용을 검토하고, 값이 클수록 과적합 감소 효과가 있다.
  - **alpha**[default=0, alias: reg_alpha]: L1 Regularizaion 적용 값으로, 위 *lambda* 와 동일하다.
  - **scale_pos_weight**[default=1]: 특정 값으로 치우친 비대칭한 클래스로 구성된 데이터 세트의 균형을 유지하기 위한 파라미터이다.

- 학습 태스크 파라미터: 학습 수행시의 객체 함수, 평가를 위한 지표 등을 설정하는 파라미터
  - **objective**: 최솟값을 가져야할 손실 함수를 정의, 이진 분류인지 다중 분류인지에 따라 달라짐.
  - **binary:logistic**: 이진 분류일 때 적용
  - **multi:softmax**: 다중 분류일 때 적용, 손실 함수가 *multi:softmax* 면 레이블 클래스의 개수인 *num_class* 파라미터를 지정해야한다.
  - **multi:softprob**: *multi:softmax* 와 유사하나 개별 레이블 클래스의 해당하는 예측 확률을 반환한다.
  - **eval_metric**: 검증에 사용되는 함수를 정의하며, 기본값은 회귀인 경우 *rmse*, 분류는 *error* 이다.
    - rmse: Root Mean Square Error
    - mae: Mean Absolute Error
    - logloss: Negative log-likelihood
    - error: Binary classification error rate(0.5 threshold)
    - merror: Multiclass classification error rate
    - mlogloss: Multiclass logloss
    - auc: Area under the curve

만약 과적합 문제가 심각하다면 아래를 고려해 볼 수 있다.

- eta 값 낮추기 + num_round(또는 n_estimators) 높이기
- max_depth 값 낮추기
- min_child_weight 값 높이기
- gamma 값 높이기
- subsample, colsample_bytree 값 조정하기

파이썬 래퍼 XGBoost로 위스콘신 유방암 예측

파이썬 래퍼 XGBoost와 사이킷런의 차이는 학습용과 테스트용 데이터 세트를 위해 별도의 객체인 **DMatrix** 를 생성!

```
dtrain = xgb.DMatrix(data=X_train, label=y_train)
dtest = xgb.DMatrix(data=X_test, label=y_test)
```

```
params = {'max_depth': 3,
         'eta': 0.1,
         'objective': 'binary:logistic',
         'eval_metric': 'logloss',
         'early_stoppings': 100}
num_rounds = 400

# train 데이터 세트는 train, test 데이터 세트는 eval(평가용 데이터)로 표기
wlist = [(dtrain, 'train'), (dtest, 'eval')]
xgb_model = xgb.train(params=params, dtrain=dtrain, num_boost_round=num_rounds, early_stopping_rounds=100, evals=wlist)
```

평가용 데이터 세트에서 eval_metric의 지정된 평가 지표로 예측 오류 측정

예측을 위해 `predict()` 메서드를 사용하는데 여기서 사이킷런의 메서드와 차이가 있다. 사이킷런의 메서드는 0, 1을 반환하는데 ***xgboost의 predict()*** 는 확률 값을 반환한다. 이진 분류 문제이므로 예측 확률이 0.5보다 크면 1, 작으면 0으로 결정하는 로직을 추가하면 된다.

```
pred_probs = xgb_model.predict(dtest)
print('predict() 수행 결과값을 10개만 표시')
print(np.round(pred_probs[:10], 3))

preds = [1 if x > 0.5 else 0 for x in pred_probs]
print(f'예측값 10개만 표시: {preds[:10]}')
```

사이킷런 래퍼 XGBoost로 위스콘신 유방암 예측

파이썬 래퍼 XGBoost와 사이킷런 래퍼 XGBoost는 파라미터 부분에서 약간의 차이가 있다.

- eta -> learning_rate
- sub_sample -> subsample
- lambda -> reg_lambda
- alpha -> reg_alpha

7️⃣ LightGBM

> 💡 리프 중심 트리 분할 방식

- 주요 파라미터
  - n_estimators[default=100]: 반복 수행하려는 트리의 개수를 지정, 크게 지정할수록 예측 성능이 높아질 수 있으나 너무 크면 과적합으로 성능 저하될 수 있다.
  - learning_rate[default=0.1]: 0-1 사이의 값을 지정하고 부스팅 스텝을 반복적으로 수행할 때 업데이트되는 학습률 값.
  - max_depth[default=-1]: 트리 기반 알고리즘의 *max_depth* 와 같다. 0보다 작은 값을 지정하면 깊이에 제한이 없고, Depth wise 방식이 아닌 Leaf wise 기반이므로 상대적으로 더 깊다.
  - min_child_samples[default=20]: 결정 트리의 *min_samples_leaf* 와 같은 파라미터로 최종 결정 클래스인 리프 노드가 되기위해 최소한으로 필요한 레코드 수이며, 과적합을 제어하기 위한 파라미터이다.
  - num_leaves[default=31]: 하나의 트리가 가질 수 있는 최대 리프 개수이다. 개수를 높이면 정확도가 높아지지만, 트리의 깊이가 깊어지고 모델의 복잡도가 커져 과적합 영향도가 커진다.
  - boosting[default=gbdt]: 부스팅의 트리를 생성하는 알고리즘
    - gbdt: 일반적인 그래디언트 부스팅 결정 트리
    - rf: 랜덤 포레스트
  - subsample[default=1.0]: 트리가 커져서 과적합되는 것을 제어하기 위해서 데이터를 샘플링하는 비율을 지정.
  - colsample_bytree[default=1.0]: 개별 트리를 학습할 때마다 무작위로 선택하는 피처의 비율
  - reg_lambda[default=0.0]: L2 Regularizaion 적용 값으로, 피처 개수가 많을 경우 적용을 검토하고, 값이 클수록 과적합 감소 효과가 있다.
  - reg_alpha[default=0.0]: L1 Regularizaion 적용 값으로, 위 *lambda* 와 동일하다.
- 학습 태스크 파라미터
  - objective: 최솟값을 가져야할 손실 함수를 정의, 이진 분류인지 다중 분류인지에 따라 달라짐.

*lightgbm에 내장된 `plot_importance()` 역시 넘파이로 피처 데이터를 학습할 경우 피처명을 알 수 없기에 Column_ 뒤에 숫자를 붙여서 나열한다.

8️⃣베이지안 최적화 기반의 HyperOpt

- 목적 함수의 식을 제대로 알 수 없는 함수에서, 최대 또는 최소의 함수 반환 값을 만드는 최적 입력값을 가능한 적은 시도를 통해 빠르고 효과적으로 찾아주는 방식
- 베이지안 확률에 기반을 두고 있는 최적화 기법
  - 베이지안 확률이 새로운 데이터를 기반으로 사후 확률을 개선해 나가듯이, 베이지안 최적화는 새로운 데이터를 입력받았을 때, 최적 함수를 예측하는 사후 모델을 개선해 나가며 최적 함수 모델을 만들어 낸다.
- 베이지안 최적화의 주요 요소
  1. 대체 모델 (Surrogate Model)
  2. 획득 함수 (Acquisition Function)

대체 모델은 획득 함수로부터 최적 함수를 예측할 수 있는 입력값을 추천받고 이를 기반으로 최적 함수 모델 개선, 획득 함수는 개선된 대체 모델을 기반으로 최적 입력값을 계산하는 프로세스

이때, 입력값은 하이퍼 파라미터에 해당함, 즉, 대체 모델은 획득 함수로부터 하이퍼 파라미터를 추천받아 모델 개선 수행, 획득 함수는 개선된 모델을 바탕으로 더 정확한 하이퍼 파라미터를 계산함!

HyperOpt 사용법 익히기

HyperOpt는 다음과 같은 프로세스를 통해 사용할 수 있다.

1. 검색 공간 설정
2. 목적 함수 절정
3. fmin() 함수를 통해 베이지안 최적화 기법에 기반한 최적의 입력 값 찾기

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

## 입력 결과를 저장할 객체 생성
trial_val = Trials()

## 목적 함수의 최솟값을 반환하는 최적 입력 변수를 5번 시도로 찾아냄
## fmin() 함수는 아래의 주요 인자를 가짐
best_01 = fmin(fn=objective_func,    ## 목적 함수
               space=search_space,   ## 검색 공간
               algo=tpe.suggest,     ## 베이지안 최적화 적용 알고리즘
               max_evals=20,         ## 입력 시도 횟수
               trials=trial_val,     ## 시도한 입력 값 및 입력 결과 저장
               #rstate=np.random.default_rng(seed=0)   ## fmin()을 시도할 때마다 동일한 결과를 가질 수 있도록 설정하는 랜덤 시드
               )
print('best:', best_01)
```

• trial_val의 results 및 vals 확인

- `results` : 딕셔너리 형태 {'loss': 함수 반환값, 'status': 반환 상태값}
- `vals` : {'입력변수명' : 개별 수행 시마다 입력된 값 리스트}
-

HyperOpt을 사용하여 XGBoost 하이퍼 파라미터 최적화

- HyperOpt는 입력 값과 반환 값이 모두 실수형이기 때문에 정수형 하이퍼 파라미터 입력 시 형변환 필요
- HyperOpt는 목적 함수의 최솟값을 반환할 수 있도록 최적화하는 것이기 때문에 성능 값이 클수록 좋은 성능 지표일 경우 -1을 곱해주어야 함

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

import pandas as pd
import numpy as np

## 유방암 데이터셋 로드
dataset = load_breast_cancer()
features = dataset.data
labels = dataset.target

## 데이터를 Pandas DataFrame으로 로드
cancer_df = pd.DataFrame(data=features, columns=dataset.feature_names)
cancer_df['target'] = labels
cancer_df.head(5)
```

```
## 학습 및 검증 데이터셋으로 데이터 분리
X_features = cancer_df.iloc[:, :-1]
y_label = cancer_df.iloc[:, -1]
## 학습, 검증용 데이터셋 비율 8 : 2
X_train, X_test, y_train, y_test = train_test_split(X_features, y_label, test_size=0.2, random_state=156)

## X_train, y_train을 다시 9 : 1 비율로 분리
## => XGBoost가 제공하는 교차 검증 성능 평가 및 조기 중단을 수행하기 위함
X_tr, X_val, y_tr, y_val = train_test_split(X_train, y_train, test_size=0.1, random_state=156)
```

```
## 모델 성능 평가 함수 선언
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_score, f1_score, roc_auc_score

def get_clf_eval(y_test, pred=None, pred_proba=None):

    confusion = confusion_matrix(y_test, pred)
    accuracy = accuracy_score(y_test, pred)
    precision = precision_score(y_test, pred)
    recall = recall_score(y_test, pred)
    f1 = f1_score(y_test, pred)
    roc_auc = roc_auc_score(y_test, pred_proba)

    print('오차 행렬')
    print(confusion)
    print('정확도: {0:.4f}, 정밀도: {1:.4f}, 재현율: {2:.4f}, F1: {3:.4f}, AUC: {4:.4f}'.format(accuracy, precision, recall, f1, roc_auc))
```

```
## 1. 검색 공간 설정

from hyperopt import hp

xgb_search_space = {
    'max_depth':hp.quniform('max_depth', 5, 20, 1),                    ## 정수형 하이퍼 파라미터 => quniform 사용
    'min_child_weight':hp.quniform('min_child_weight', 1, 2, 1),       ## 정수형 하이퍼 파라미터 => quniform 사용
    'learning_rate':hp.uniform('learning_rate', 0.01, 0.2),
    'colsample_bytree':hp.uniform('colsample_bytree', 0.5, 1),
}
```

```
## 2. 목적 함수 설정
## 검색 공간에서 설정한 하이퍼 파라미터들을 입력 받아서 XGBoost를 학습시키고, 평가 지표를 반환하도록 구성되어야 함

from sklearn.model_selection import cross_val_score  ## 교차 검증
from xgboost import XGBClassifier
from hyperopt import STATUS_OK

def objective_func(search_space):

  xgb_clf = XGBClassifier(
      n_estimators=100,
      max_depth=int(search_space['max_depth']),               ## int형으로 형변환 필요
      min_child_weight=int(search_space['min_child_weight']),  ## int형으로 형변환 필요
      learning_rate=search_space['learning_rate'],
      colsample_bytree=search_space['colsample_bytree'],
      eval_metric='logloss'
  )

  accuracy = cross_val_score(xgb_clf, X_train, y_train, scoring='accuracy', cv=3)  ## 3개의 교차 검증 세트로 정확도 반환

  ## acc는 cv=3 개수만큼의 결과를 리스트로 가짐, 이를 평균하여 반환하되, -1을 곱함
  return {
      'loss':(-1) * np.mean(accuracy),
      'status':STATUS_OK
  }
```

```
## 3. fmin()을 사용하여 최적 하이퍼 파라미터 찾기

from hyperopt import fmin, tpe, Trials

trial_val = Trials() ## 결과 저장

best = fmin(
    fn=objective_func,
    space=xgb_search_space,
    algo=tpe.suggest,
    max_evals=50,    ## 최대 반복 횟수 지정
    trials=trial_val,
)
```

```
## 획득한 최적의 하이퍼 파라미터를 이용하여 모델 선언

xgb_wrapper = XGBClassifier(
    n_estimators=400,
    learning_rate=round(best['learning_rate'], 5),
    max_depth=int(best['max_depth']),
    min_child_weight=int(best['min_child_weight']),
```

```
    colsample_bytree=round(best['colsample_bytree'], 5)
)
```

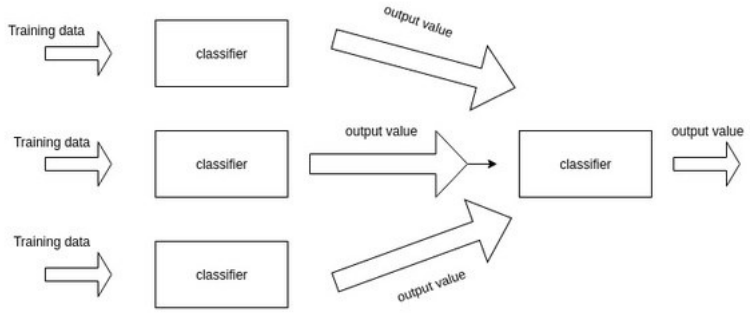
```
## early stopping
evals = [(X_tr, y_tr), (X_val, y_val)]

## model train
xgb_wrapper.fit(
    X_tr, y_tr,
    early_stopping_rounds=50,
    eval_metric='logloss',
    eval_set=evals,
    verbose=True
)
```

```
## eval

preds = xgb_wrapper.predict(X_test)
pred_proba = xgb_wrapper.predict_proba(X_test)[:, 1]

get_clf_eval(y_test, preds, pred_proba)
```

1️⃣1️⃣ 스테킹 앙상블

스태킹(Stacking)은 개별적인 여러 알고리즘을 결합해 예측 결과를 도출하는 것이 배깅(Bagging), 부스팅(Boosting)과 비슷하지만, 개별 알고리즘으로 예측한 데이터를 기반으로 다시 예측을 한다.

즉, 개별 알고리즘의 예측 결과 데이터 세트를 최종적인 메타 데이터 세트로 만들어 별도의 머신러닝 알고리즘으로 최종 학습하고 테스터 데이터를 기반으로 다시 최종 예측을 하는 방식이다.



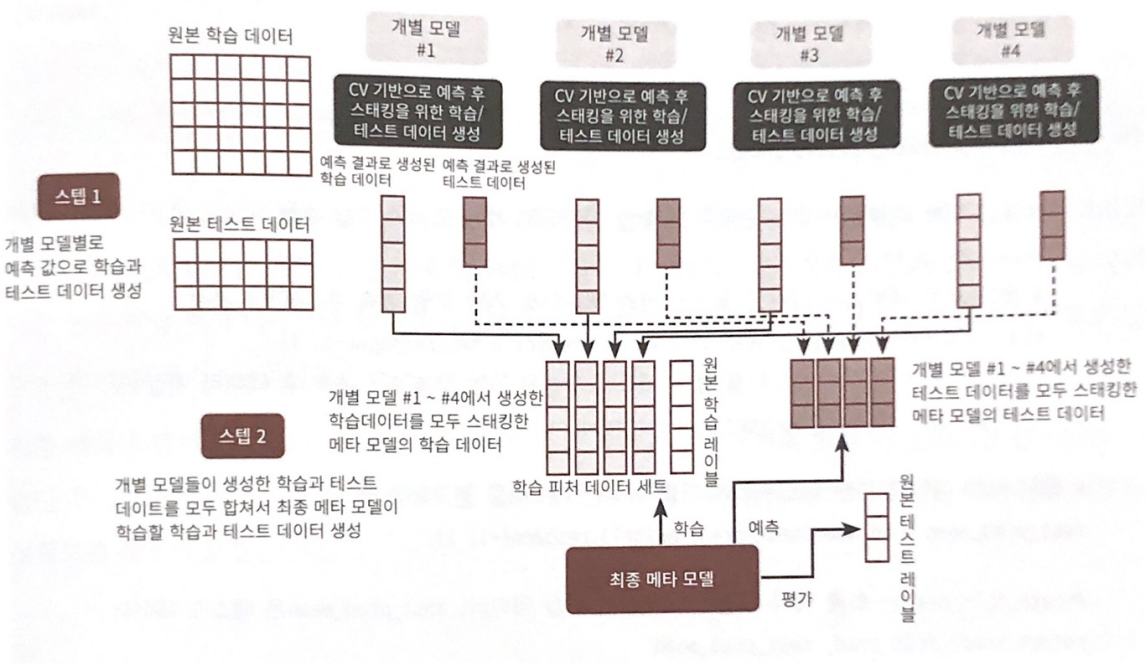
스태킹 모델의 핵심은 여러 개별 모델의 예측 데이터를 각각 스태킹 형태로 결합해 최종 메타 모델의 학습용 피처 데이터 세트와 테스트용 데이터 세트를 만드는 것

CV 세트 기반의 스태킹

CV 세트 기반의 스태킹 모델은 과적합을 개선하기 위해 최종 메타 모델을 위한 데이터 세트를 만들 때 **교차 검증 기반**으로 예측된 결과 데이터 세트를 이용한다. 개별 모델들이 각각 교차 검증으로 메타 모델을 위한 학습용, 테스트용 데이터를 생성한 뒤 이를 기반으로 메타 모델이 학습과 예측을 수행한다.

1) 각 모델별로 원본 학습/테스트 데이터를 예측한 결과 값을 기반으로 메타 모델을 위한 학습용/테스트용 데이터 생성

2) 개별 모델들이 생성한 학습용 데이터를 모두 스태킹 형태로 합쳐서 메타 모델이 학습할 최종 학습용 데이터 세트 생성. (테스트용 데이터도 마찬가지) 메타 모델은 최종적으로 생성된 학습 데이터 세트와 원본 학습 데이터의 레이블 데이터를 기반으로 학습한 뒤, 최종적으로 생성된 테스트 데이터 세트를 예측하고, 원본 테스트 데이터의 레이블 데이터를 기반으로 평가


원본 학습 데이터
개별 모델 #1
개별 모델 #2
개별 모델 #3
개별 모델 #4
CV 기반으로 예측 후 스태킹을 위한 학습/테스트 데이터 생성
CV 기반으로 예측 후 스태킹을 위한 학습/테스트 데이터 생성
CV 기반으로 예측 후 스태킹을 위한 학습/테스트 데이터 생성
CV 기반으로 예측 후 스태킹을 위한 학습/테스트 데이터 생성
예측 결과로 생성된 학습 데이터
예측 결과로 생성된 테스트 데이터
스텝 1
개별 모델별로 예측 값으로 학습과 테스트 데이터 생성
원본 테스트 데이터
개별 모델 #1 ~ #4에서 생성한 학습데이터를 모두 스태킹한 메타 모델의 학습 데이터
원본 학습 레이블
개별 모델 #1 ~ #4에서 생성한 테스트 데이터를 모두 스태킹한 메타 모델의 테스트 데이터
스텝 2
개별 모델들이 생성한 학습과 테스트 데이트를 모두 합쳐서 최종 메타 모델이 학습할 학습과 테스트 데이터 생성
학습 피처 데이터 세트
학습
예측
원본 테스트 레이블
최종 메타 모델
평가