


# ML\_06\_2. 앙상블과 부스팅

🕒 생성일	@2022년 6월 9일 오전 1:03
📁 유형	머신러닝/딥러닝
👤 작성자	 동훈 오

## Intro

No free lunch theorem

- Can we expect any classification method to be superior or inferior overall? → No.
- If one algorithm seems to outperform another in a particular situation, it is a consequence of its fit to a particular pattern recognition problem.
- In practice, experience with a broad range of techniques is the best insurance for solving arbitrary new classification problems.

---

## 혼공머신: 5장 트리 알고리즘 - 트리의 앙상블

- 랜덤 포레스트
- 엑스트라 트리
- 그레디언트 부스팅 머신 (gradient boosting machine; GBM)
- XGBoost
- LightGBM

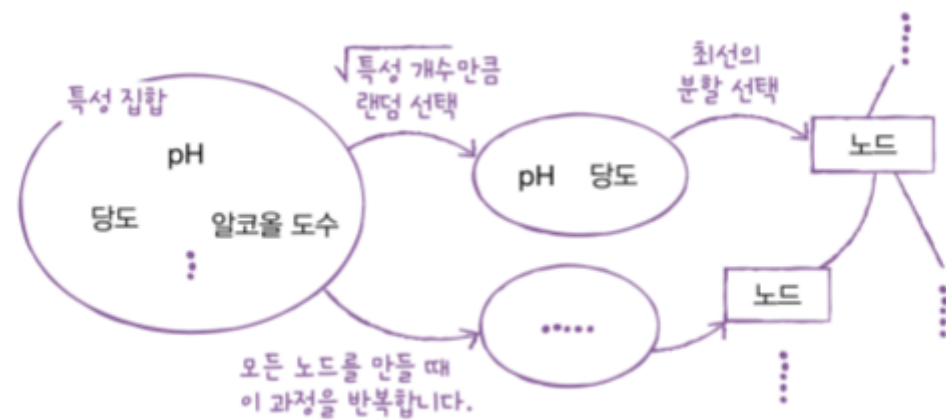
### 랜덤 포레스트

랜덤 포레스트는 결정 트리를 랜덤하게 만들어 결정 트리의 숲을 만든다. 그리고 각 결정 트리의 예측을 사용해 최종 예측을 만든다.

- 랜덤 포레스트는 훈련 데이터에서 랜덤하게 샘플을 추출하는데, 복원 추출 방식이다.
- 만들어진 샘플을 부트스트랩 샘플(bootstrap sample) 이라고 부른다.
- 기본적으로 부트스트랩 샘플은 훈련 세트의 크기와 같게 만든다.

분류 모델인 RandomForestClassifier 는 기본적으로 전체 특성 개수의 제곱근만큼의 특성을 선택한다. 즉 4개의 특성이 있다면 노드마다 2개를 랜덤하게 선택하여 사용한다.

사이킷런의 랜덤포레스트는 기본적으로 100개의 결정트리를 이런 방식으로 훈련한다. 그다음 분류일 때는 각 트리의 클래스별 확률을 평균하여 가장 높은 확률을 가진 클래스를 예측으로 삼는다.회귀일 때는 단순히 각 트리의 예측을 평균한다.



사이킷런에서 랜덤 부스트를 사용하는 방식은 다음과 같다.

```
# 데이터는 이전에 만든 train set과 test set을 사용한다.
# 랜덤 포레스트 객체 생성
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_jobs=-1, random_state=42)

# 교차 검증
scores = cross_validate(rf, train_input, train_target,
                        return_train_score=True, n_jobs=-1)
print(np.mean(scores['train_score']), np.mean(scores['test_score']))

>>> 0.9973541965122431 0.8905151032797809
```

결정트리 분류모델 대신 랜덤포레스트 분류 모델을 넣은 것과 같은 구조이다.

## 엑스트라 트리(Extra tree)

엑스트라 트리는 랜덤 포레스트 보다 조금 더 무작위성이 강하다.

- 엑스트라 트리는 랜덤 포레스트와 유사, 기본적으로 100개의 결정 트리를 훈련
- 엑스트라 트리는 부트스트랩 샘플을 사용하지 않고 전체 훈련 세트를 사용한다.
- 노드를 분할 할 때 무작위로 분할 한다.
- 무작위 분할은 성능이 낮아지는 대신 많은 트리를 앙상블하기 때문에 과대적합을 막고 검증 세트의 점수를 높이는 효과가 있다.

```
from sklearn.ensemble import ExtraTreesClassifier
et = ExtraTreesClassifier(n_jobs=-1, random_state=42)

scores = cross_validate(et, train_input, train_target,
                        return_train_score=True, n_jobs=-1)
print(np.mean(scores['train_score']), np.mean(scores['test_score']))
>>> 0.9974503966084433 0.8887848893166506
```

결과를 보면 랜덤 포레스트와 비슷하다. 하지만 코드 실행 시간은 랜덤포레스트보다 훨씬 빠르게 계산된다.

## gradient boosting machine

- 그레디언트 부스팅은 깊이가 얇은 결정 트리를 사용하여 이전 트리의 오차를 보완하는 방식으로 앙상블 하는 방법
- 사이킷런의 GradientBoostingClassifier 는 기본적으로 깊이가 3인 결정트리를 100개 사용.
- 그레디언트 부스팅은 경사 하강법을 사용하여 트리를 앙상블에 추가한다.
- 분류에서는 로지스틱 손실함수를 사용한다.

회귀 : 평균제곱오차함수(MSE)

- 결정 트리를 계속 추가하면서 극소점을 찾는것이 모델의 목적이다.
- 학습률 매개변수로 속도를 조절한다.

```
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier(random_state=42) # 학습률 default = 0.1
scores = cross_validate(gb, train_input, train_target,
                        return_train_score=True, n_jobs=-1)
print(np.mean(scores['train_score']), np.mean(scores['test_score']))
>>> 0.8881086892152563 0.8720430147331015
```

결과를 보면 과대적합이 많이 줄어들었다.

## XGBoost

XGBoost 는 그레디언트 부스팅 알고리즘을 구현한 대표적인 라이브러리이다. 구체적인 설명은 다른 자료를 활용할 예정이다. 여기선느 라이브러리의 구현만 보면 된다.

```
from xgboost import XGBClassifier
xgb = XGBClassifier(tree_method='hist', random_state=42)
scores = cross_validate(xgb, train_input, train_target,
                        return_train_score=True)
print(np.mean(scores['train_score']), np.mean(scores['test_score']))
>>> 0.8824322471423747 0.8726214185237284
```

결과를 보면 트리 모델보다 과대적합이 많이 줄어들었다. GradientBoostingClassifier 을 사용했을 때보다도 줄어들었다.

## LightGBM

MS 에서 제공하는 히스토그램 기반 그레디언트 부스팅 라이브러리이다.

```
from lightgbm import LGBMClassifier
lgb = LGBMClassifier(random_state=42)
scores = cross_validate(lgb, train_input, train_target,
                        return_train_score=True, n_jobs=-1)
print(np.mean(scores['train_score']), np.mean(scores['test_score']))
>>> 0.9338079582727165 0.8789710890649293
```

다른 그레디언트 부스팅 라이브러리 보다 성능이 조금 아쉬운 결과이다.`

# 핸즈온: 7. 앙상블 학습과 랜덤 포레스트

## 투표 기반 분류기

더 좋은 분류기를 만드는 매우 간단한 방법은 각 분류기의 예측을 모아서 가장 많이 선택된 클래스를 예측하는 것이다. 이렇게 다수결 투표로 정해지는 분류기를 hard voting 분류기 라고 한다. 놀랍게도 이 다수결 투표 분류기가 앙상블에 포함된 개별 분류기 중 가장 뛰어난 것보다도 정확도가 높을 경우가 많다. 이것이 가능한 이유는 큰 수의 법칙 때문이다.

다음은 사이킷런의 투표 기반 분류기를 만들고 훈련시키는 코드이다. 데이터는 moons 데이터셋을 사용했다.

```
# dataload
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC(probability=True)

# 투표 기반 분류기 객체 생성.
voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='soft')
voting_clf.fit(X_train, y_train)

# 각 분류기의 정확도를 확인해보자
from sklearn.metrics import accuracy_score
for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))

>>> LogisticRegression 0.864
      RandomForestClassifier 0.904
      SVC 0.896
      VotingClassifier 0.92
```

## 배깅과 페이스팅

다양한 분류기를 만드는 또 다른 방법은 같은 알고리즘을 사용하고 훈련 세트의 서브셋을 무작위로 구성하여 분류기를 각기 다르게 학습시키는 것이다.

훈련 세트에서 중복을 허용하여 샘플링하는 방식을 Bagging (Bootstrap aggregating) 이라 하며, 중복을 허용하지 않고 샘플링하는 방식을 pasting 이라고 한다.

배깅과 페이스팅 방식에서는 같은 훈련 샘플을 여러 개의 예측기에 걸쳐 사용할 수 있다. 하지만 배깅만이 한 예측기를 위해 같은 훈련 샘플을 여러 번 샘플링할 수 있다.

모든 예측기가 훈련을 마치면 앙상블은 모든 예측기의 예측을 모아서 새로운 샘플에 대한 예측을 만든다. 수집 함수는 전형적으로 분류일 때는 통계적 최빈값(statistical mode) 이고 회귀에 대해서는 평균(mean) 을 계산한다. 개별 예측기는 원본 훈련 세트로 훈련시킨 것보다 훨씬 크게 편향되어 있지만 수집 함수를 통과하면 편향과 분산이 모두 감소한다. 일반적으로 앙상블의 결과는 원본 데이터셋으로 하나의 예측기를 훈련시킬 때와 비교해 편향은 비슷하지만 분산은 줄어든다.

사이킷런은 BaggingClassifier, BaggingRegressor 을 제공한다. 다음은 결정 트리 분류기 500개의 앙상블을 훈련시키는 코드이다. bootstrap=True 로 지정하면 배깅, False로 지정하면 페이스팅을 사용할 수 있다. n\_jobs 는 CPU 코어 수를 지정하며 -1 설정 시 가용 가능한 모든 코어를 사용한다.

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, random_state=42)
bag_clf.fit(X_train, y_train)
```

## 랜덤 패치와 랜덤 서브스페이스

사이킷런 배깅분류/회귀 클래스는 특성 샘플링도 지원한다. `max_features`, `bootstrap_features` 두 매개 변수로 조절되며, 각 예측기는 무작위로 선택한 입력 특성의 일부분으로 훈련된다.

훈련 특성과 샘플을 모두 샘플링하는 것을 `random patches method` 라고 한다. 훈련 샘플을 모두 사용하고, 특성은 샘플링하는 것을 `random subspaces method` 라고 한다. 특성 샘플링은 더 다양한 예측기를 만들며 편향을 늘리는대신 분산을 낮춘다.

## 랜덤 포레스트 (random forest)

랜덤 포레스트는 일반적으로 배깅 방법을 적용한 결정 트리의 앙상블이다. 전형적으로 `max_samples`를 훈련 세트의 크기로 지정한다. 사이킷런은 랜덤포레스트 분류와 회귀 클래스를 지원한다.

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, random_state=42)
rnd_clf.fit(X_train, y_train)
rnd_clf.predict(X_test)
```

랜덤 포레스트 알고리즘은 트리의 노드를 분할할 때 전체 특성 중에서 최선의 특성을 찾는 대신 무작위로 선택한 특성 후보 중에서 최적의 특성을 찾는 식으로 무작위성을 더 주입한다. 이는 결국 트리를 더욱 다양하게 만들고 편향을 손해보는 대신 분산을 낮춘다.

## 엑스트라 트리

극단적으로 무작위한 트리의 랜덤 포레스트를 `extra tree` 라고 한다. 랜덤 포레스트에서 트리를 더욱 무작위하게 만들기 위해 최적의 임계값을 찾는 대신 후보 특성을 사용해 무작위로 분할한 다음 그 중에서 최상의 분할을 선택한다. 랜덤성을 강하게 하는 전략은 분산을 최대한 낮추는데 목적이 있다. 일반적으로 엑스트라 트리는 랜덤 포레스트보다 훨씬 빠르다.

## 특성 중요도

사이킷런은 어떤 특성을 사용한 노드가 평균적으로 불순도를 얼마나 감소시키는지 확인하여 특성의 중요도를 측정한다. 무작위성이 주입된 랜덤 포레스트는 거의 모든 특성에 대해 평가할 기회를 가진다.

```
from sklearn.datasets import load_iris
iris = load_iris()
rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
rnd_clf.fit(iris['data'], iris['target'])
for name, score in zip(iris['feature_names'], rnd_clf.feature_importances_):
    print(name, score)

>>> sepal length (cm) 0.10430913018658267
      sepal width (cm) 0.02427124402411902
      petal length (cm) 0.43576694741324595
      petal width (cm) 0.4356526783760523
```

## 부스팅(boosting)

부스팅은 약한 학습기(learner)를 여러 개 연결하여 강한 학습기를 만드는 앙상블 방법을 말한다. 부스팅 방법의 아이디어는 앞의 모델을 보완해나가면서 일련의 예측기를 학습시키는 것이다.

## AdaBoost

adaboost 분류기를 만들 때 먼저 알고리즘이 기반이 되는 첫 번째 분류기(ex. 결정 트리)를 훈련 세트에서 훈련시키고 예측을 만든다. 그다음에 알고리즘이 잘못 분류된 훈련 샘플의 가중치를 상대적으로 높인다. 두 번째 분류기는 업데이트된 가중치를 사용해 훈련 세트에서 훈련하고 다시 예측을 만든다. 이렇게 계속 가중치가 업데이트 된다.

이런 연속 학습 기법은 경사 하강법과 비슷한 면이 있다. 경사 하강법은 비용 함수를 최소화하기 위해 한 예측기의 모델 파라미터를 조정해가는데 반면 adaboost는 점차 더 좋아지도록 앙상블에 예측기를 추가한다.

다음은 adaboost 알고리즘의 학습 과정을 설명한 것이다.

각 샘플 가중치  $w^i$ 는 초기에  $1/m$ 으로 초기화 된다. 첫 번째 예측기가 학습되고 가중치가 적용된 에러율  $r_1$ 이 훈련 세트에 대해 계산된다.

$$r_j = \frac{\sum_{i=1}^m w^{(i)} \mathbb{I}_{\hat{y}_j^{(i)} \neq y^{(i)}}}{\sum_{i=1}^m w^{(i)}} \quad \text{여기서 } \hat{y}_j^{(i)} \text{는 } i\text{번째 샘플에 대한 } j\text{번째 예측기의 예측}$$

위 수식을 통해 구한 에러율과 학습률 하이퍼파라미터  $\eta$ 를 사용해서 예측기 가중치  $\alpha_j$ 를 구할 수 있다. 수식은 아래와 같으며 예측기가 정확할수록 가중치가 높게 측정된다.

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

그 다음 알고리즘이 다음의 식을 사용해 샘플의 가중치를 업데이트 한다. 즉, 잘못 분류된 샘플의 가중치가 증가한다.

### 식 7-3 가중치 업데이트 규칙

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \hat{y}_j^{(i)} = y^{(i)} \text{일 때} \\ w^{(i)} \exp(\alpha_j) & \hat{y}_j^{(i)} \neq y^{(i)} \text{일 때} \end{cases}$$

여기서  $i = 1, 2, \dots, m$

그런 다음 모든 샘플의 가중치를 정규화한다.  $w^i$ 의 summation으로 나누는 것과 같다.

마지막으로 새 예측기가 업데이트된 가중치를 사용해 훈련되고 전체 과정이 반복된다. 이 알고리즘은 지정된 예측기 수에 도달하거나 완벽한 예측기가 만들어지면 중지된다.

adaboost는 가중치 합이 가장 큰 클래스가 예측 결과가 된다.

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x})=k}}^N \alpha_j$$

여기서  $N$ 은 예측기 수

다음 코드는 사이킷런에서 제공하는 adaboost 분류기 클래스이다. 200 개의 아주 얇은(max\_depth=1) 결정 트리를 기반으로 하는 adaboost 분류기를 훈련시킨다.

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)
```

## 그레디언트 부스팅 (gradient boosting)

adaboost 처럼 그레디언트 부스팅은 앙상블에 이전까지의 오차를 보정하도록 예측기를 순차적으로 추가한다. 하지만 adaboost 처럼 반복마다 샘플의 가중치를 수정하는 대신 이전 예측기가 만든 잔여 오차에 새로운 예측기를 학습시킨다.

결정 트리를 기반 예측기로 사용하는 그레디언트 부스팅 회귀 모델을 사이킷런에서는 GradientBoostingRegressor 으로 구현할 수 있다.

```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(
    max_depth=2, n_estimators=3, learning_rate=1.0)
gbrt.fit(X, y)
```

learning rate 매개 변수가 각 트리의 기여 정도를 조절한다. 이를 0.1 처럼 낮게 설정하면 앙상블을 훈련 세트에 학습시키기 위해 많은 트리가 필요하지만 일반적으로 예측의 성능은 좋아진다. 이는 shrinkage 라고 부르는 규제 방법이다.

## XGBoost

extreme gradient boosting 의 약자이다. 이 패키지의 목표는 매우 빠른 속도, 확장성, 이식성이다.

### XGBoost 개념 이해

조대협 (<http://bcho.tistory.com>) XGBoost는 Gradient Boosting 알고리즘을 분산환경에서도 실행할 수 있도록 구현해 놓은 라이브러리이다. Regression, Classification 문제를 모두 지원하며, 성능과 자원 효율이 좋아서, 인기 있게 사용되는 알고리즘이다. XGBoost는 여러개의 Decision Tree를 조합해서 사용하는 Ensemble 알고리즘이다. 먼저 Decision

🔗 <https://bcho.tistory.com/1354>

