04

분류

분류 (classification)

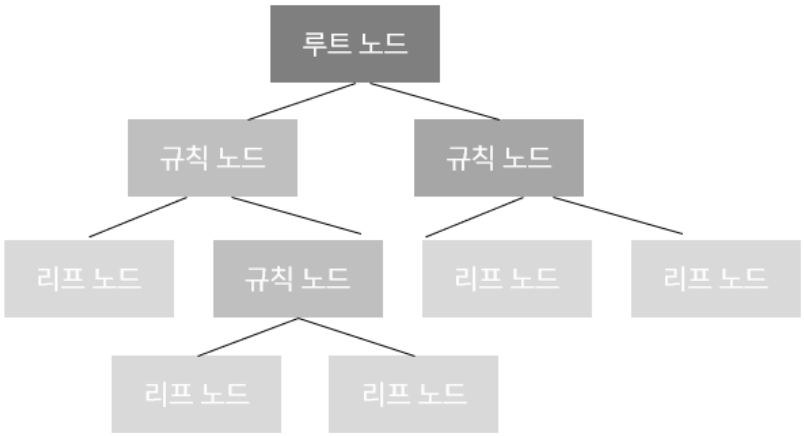
지도학습의 대표적인 유형인 분류는 학습 데이터로 주어진 데이터의 피처와 레이블 값을 machine learning 알고리즘으로 학습해 모델을 생성하고, 생성된 모델에 새로운 데이터가 주어졌을 때 미지의 레이블을 예측하는 것입니다.

분류를 구현하는 machine learning 알고리즘

- Decision tree
- Logistic regression
- Support vector machine

Decision Tree

Decision tree 는 데이터에 있는 규칙을 학습을 통해 자동으로 찾아내 tree 기반의 분류 규칙을 만드는 것입니다. 일반적으로 규칙을 가장 쉽게 표현하는 방법은 if/else 기반으로 나타내는 것입니다. 따라서 데이터를 어떤 기준(if 조건)으로 분류 해야 하는가가 알고리즘의 성능을 크게 좌우하게 됩니다.



위 그림은 결정 tree 의 구조입니다. 많은 규칙이 있다는 것은 결국 tree 가 복잡해진다는 이야기이고, 이는 overfitting 의 문제를 야기합니다. 즉, tree 의 깊이(depth)가 깊어질수록 결정 tree 의 예측 성능이 저하될 수 있습니다.

따라서 가능한 적은 결정 node 로 높은 예측 정확도를 가지려면 어떻게 tree 를 분할할 것인가가 중요한데, leaf node 에 최대한 균일한 데이터를 구성할 수 있도록 분할하는 것이 필요합니다. 다시 말해, 데이터가 모두 특정 분류에 속하게 되도록 데이터를 쪼개는 방식으로 분류를 합니다.

이렇듯 decision tree 는 정보의 균일한 분리에만 신경 쓰면 되기 때문에, 특징 스케일링/정규화 등의 데이터 전처리 과정이 필요 없다는 장점을 가지고 있습니다. 반면에 overfitting 의 발생 가능성이 높아 예측정확도가 떨어질 수 있다는 단점을 가지고 있습니다. 이를 해결하기 위해 계속하여 규칙 node 를 추가하게 되면 트리 깊이가 커지게 되어 실제 상황(평가 데이터)에 유연하게 대처하지 못하게 됩니다. 따라서 tree 의 최대 성장 가능 크기를 사전에 제한하는 것이 오히려 성능 향상에 도움이 됩니다.

Scikit-learn 의 Decision Tree 학습

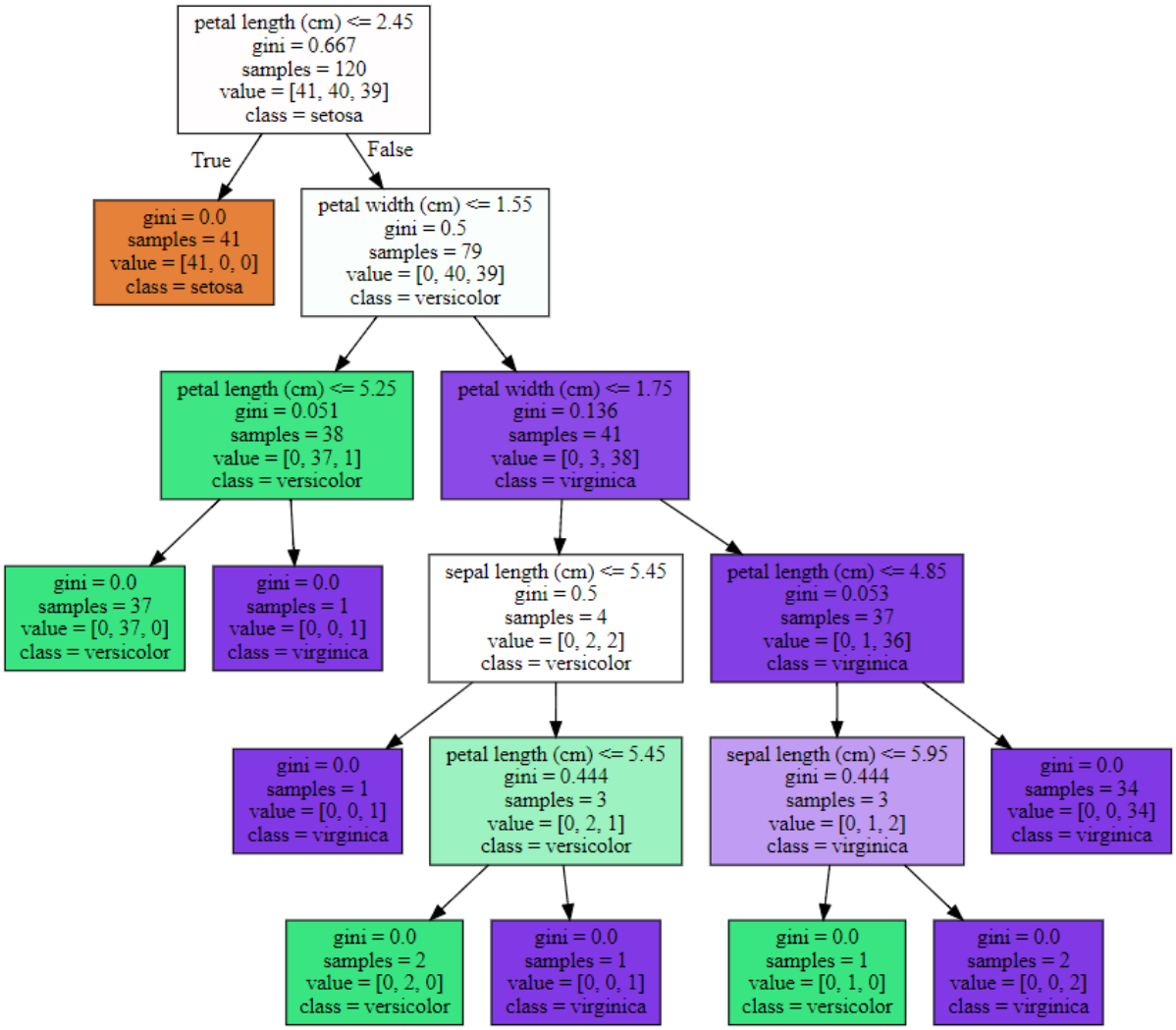
Scikit-learn 은 결정 트리 알고리즘을 구현한 DecisionTreeClassifier 를 제공합니다. 붓꽃 데이터를 이 DecisionTreeClassifier 를 이용하여 학습한 뒤 어떠한 형태로 decision tree 가 만들어지는 지 확인해 보겠습니다.

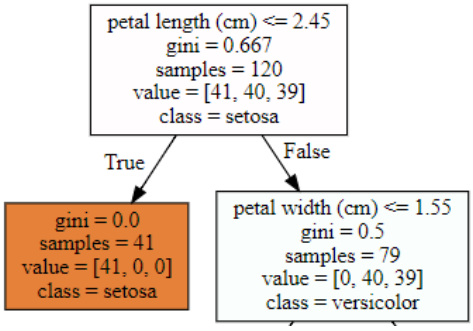
```
from sklearn.tree import DecisionTreeClassifier

dt_clf = DecisionTreeClassifier(random_state=156)
X_train, X_test, y_train,y_test = train_test_split(iris_data.data,
                                                   iris_data.target,
                                                   test_size=0.2, random_state=11)
# Decision Tree Classifier 학습
dt_clf.fit(X_train, y_train)
```

Scikit-learn 의 Decision Tree 시각화

Scikit-learn 의 트리 모듈은 시각화를 위해 export_graphviz( ) 함수를 제공합니다. export_graphviz( ) 함수를 통해 생성된 파일 'tree.dot'를 Graphviz 모듈을 호출하여 시각적으로 표현할 수 있습니다.

```
from sklearn.tree import export_graphviz
import graphviz

# tree.dot 생성
export_graphviz(dt_clf, out_file="tree.dot", class_names = iris_data.target_names,
                feature_names = iris_data.feature_names, impurity=True, filled=True)
with open("tree.dot") as f:
  dot_graph = f.read()
graphviz.Source(dot_graph)
```





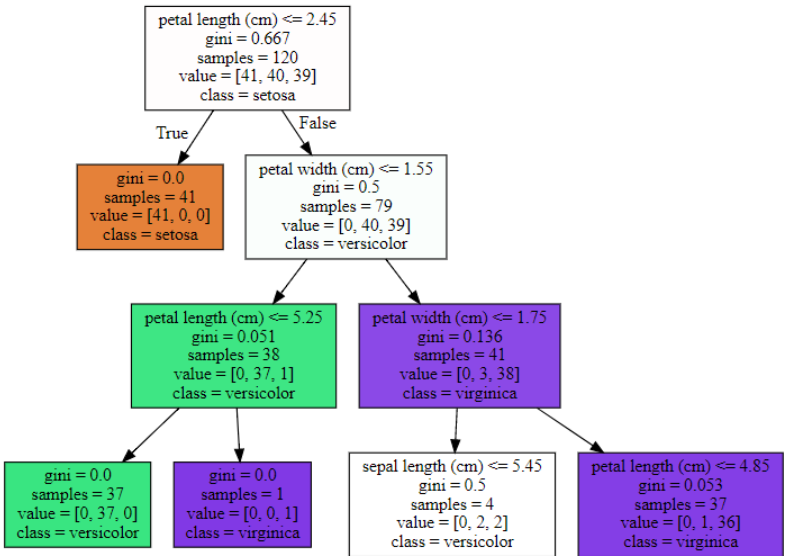
예를 들어, 위 sub-tree 에서 표시된 node 는 petal width <= 2.45 규칙이 False 인 규칙 node 입니다. 이때 petal width <=1.55 규칙으로 자식 node 를 생성하게 됩니다.

Scikit-learn 의 Decision Tree Hyper-parameter

Hyper-parameter 를 통해 트리 깊이를 줄이고, tree 를 보다 간결하게 만들 수 있습니다.

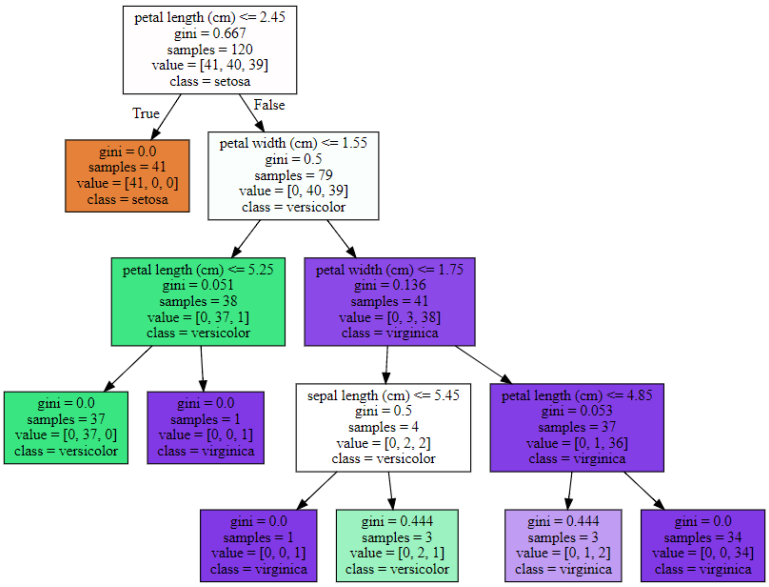
- max_depth

```
from sklearn.tree import export_graphviz
import graphviz

# tree.dot 생성
dt_clf = DecisionTreeClassifier(random_state=156, max_depth=3)
dt_clf.fit(X_train, y_train)
export_graphviz(dt_clf, out_file="tree.dot", class_names = iris_data.target_names,
                feature_names = iris_data.feature_names, impurity=True, filled=True)
with open("tree.dot") as f:
  dot_graph = f.read()
graphviz.Source(dot_graph)
```



- min_samples_split

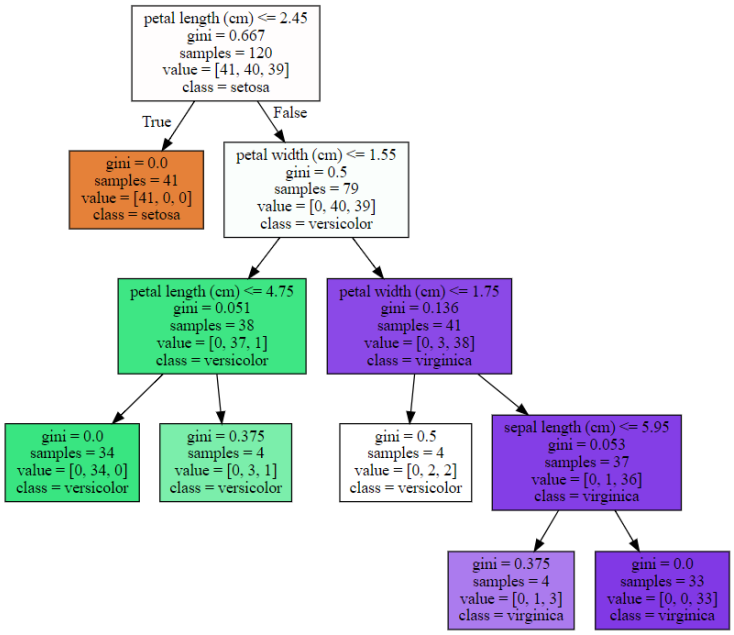
```
from sklearn.tree import export_graphviz
import graphviz

# tree.dot 생성
dt_clf = DecisionTreeClassifier(random_state=156, min_samples_split=4)
dt_clf.fit(X_train, y_train)
export_graphviz(dt_clf, out_file="tree.dot", class_names = iris_data.target_names,
                feature_names = iris_data.feature_names, impurity=True, filled=True)
with open("tree.dot") as f:
  dot_graph = f.read()
graphviz.Source(dot_graph)
```



min_sample_splits 는 자식 규칙 node 를 분할해 만들기 위한 최소한의 샘플 데이터 수입니다. 예를 들어, min_samples_splits = 4 이면, 자식 node 로 분할하려면 최소한의 샘플 개수가 4 개가 필요하다는 말입니다. 즉, 샘플 개수가 3 개 뿐이라면 더 이상 분할하지 않고 leaf node 가 됩니다.

- min_samples_leaf

```
from sklearn.tree import export_graphviz
import graphviz

# tree.dot 생성
dt_clf = DecisionTreeClassifier(random_state=156, min_samples_leaf=4)
dt_clf.fit(X_train, y_train)
export_graphviz(dt_clf, out_file="tree.dot", class_names = iris_data.target_names,
                feature_names = iris_data.feature_names, impurity=True, filled=True)
with open("tree.dot") as f:
  dot_graph = f.read()
graphviz.Source(dot_graph)
```

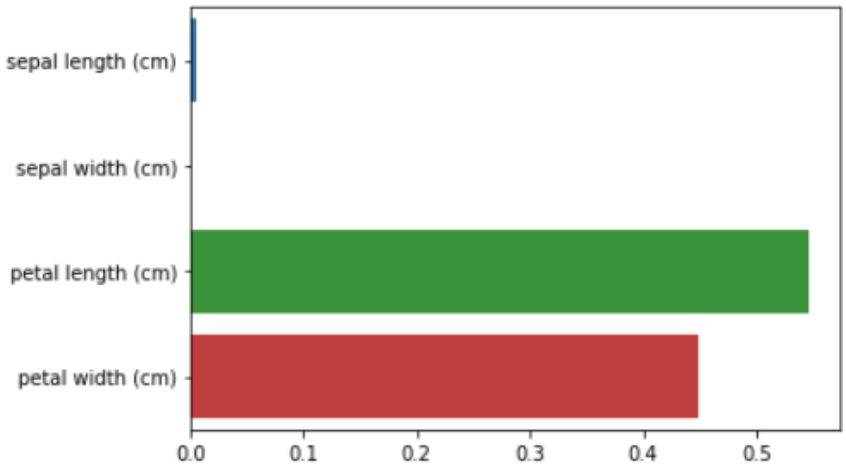


min_samples_leaf <= 지정 값 기준을 만족하면 leaf node 가 될 수 있습니다. 즉, min_samples_leaf = 4 로 설정하면 샘플이 4 이하이면 leaf node 가 됩니다.

- feature_importances_ 속성

DecisionTreeClassifier 객체의 feature_importances 속성을 통해 특징 별로 중요도 값을 확인할 수 있습니다.

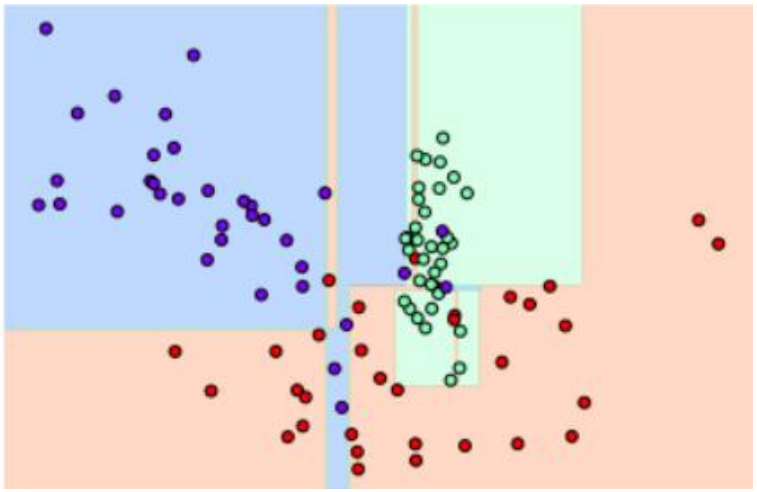
```
feature_importance = dt_clf.feature_importances_
```



Decision Tree 과적합 (Overfitting)

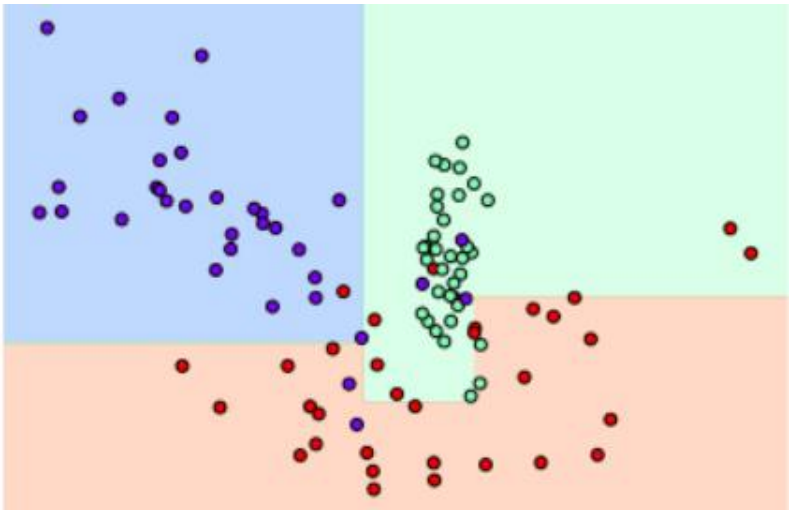
학습데이터에만 지나치게 최적화된 분류를 overfitting 이라고 합니다. 이러한 문제를 해결하기 위해서 hyper-parameter 를 적절히 조정하는 것이 필요합니다. 예를 들어 설명하겠습니다. 우선 decision tree 생성에 별다른 제약이 없도록 hyper-parameter 가 디폴트인 classifier 을 학습하고, 결정 기준 경계를 시각화한 자료입니다. 일부 이상치(outlier) 데이터까지 분류하기 위해 분할이 자주 일어나서 결정 기준 경계가 많아졌습니다.

```
# 특정한 트리 생성 제약없는 결정 트리의 Decsion Boundary 시각화.
dt_clf = DecisionTreeClassifier().fit(X_features, y_labels)
visualize_boundary(dt_clf, X_features, y_labels)
```



이번에는 min_sample_leaf = 6 을 설정해 6 개 이하의 데이터는 leaf node 를 생성할 수 있도록 node 생성 규칙을 완화하여 hyper-parameter 를 변경 후 결정 기준 경계가 바뀐 모습입니다.

```
dt_clf = DecisionTreeClassifier( min_samples_leaf=6).fit(X_features, y_labe
ls)
visualize_boundary(dt_clf, X_features, y_labels)
```



Hpyer-parameter 를 조정한 아래 분류가 이상 치에 크게 반응하지 않으면서 좀 더 일반화된 분류 규칙에 따라 분류되었음을 알 수 있습니다. 앞선 디폴트 예제와 같이 학습 데이터에만 지나치게 최적화된 분류 기준은 오히려 테스트 데이터에서 정확도를 떨어뜨릴 수 있습니다.

Decision Tree 실습 – 사용자 행동 인식 데이터

- https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones

앙상블 학습 (Ensemble Learning)

앙상블 학습을 통한 분류는 여러 개의 분류기를 생성하고 그 예측을 결합함으로써 보다 정확한 최종 예측을 도출하는 기법을 말합니다. 즉, 앙상블 학습의 목표는 다양한 분류기의 예측 결과를 결합함으로써 단일 분류기보다 신뢰성이 높은 예측 값을 얻는 것입니다.

특히나 앙상블 학습은 정형 데이터의 분류 시에 뛰어난 성능을 나타냅니다.

비정형데이터	비정형 데이터
이미지, 영상 음성	수치를 통한 의미파악이 용이한 데이터

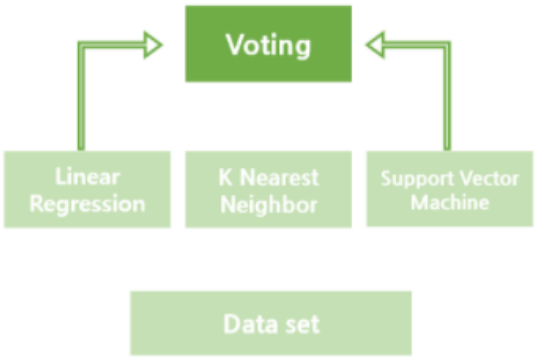
앙상블 학습의 유형은 전통적으로 1) voting, 2) bagging, 3) boosting 의 세 가지로 나눌 수 있으며, 이외에도 stacking 을 포함한 다양한 앙상블 방법이 있습니다.

Voting 과 bagging 은 여러 개의 분류기가 투표를 통해 최종 예측 결과를 결정하는 방식입니다. 둘의 차이점은

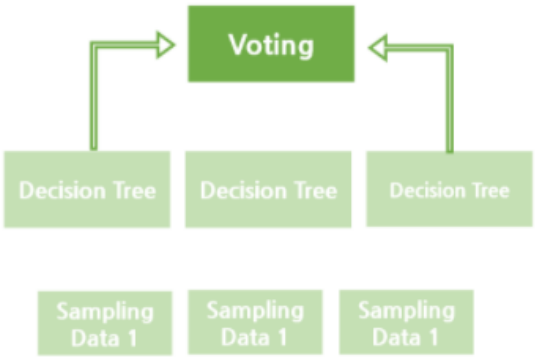
1) Voting 의 경우, 일반적으로 서로 다른 알고리즘의 분류기를 결합하는 것,

2) Bagging 은 서로 같은 알고리즘의 분류기이지만, 데이터 샘플링을 서로 다르게 가져가면서 (bootstrapping) 학습을 수행하는 것입니다.

3) Boosting 은 여러 개의 분류기가 순차적으로 학습을 수행하되, 앞에서 학습한 분류기가 예측이 틀린 데이터에 대해서는 올바르게 예측할 수 있도록 다음 분류기에게는 가중치를 부여하면서 학습과 예측을 진행하는 것입니다.



Voting



Bagging

Voting 분류기

Voting 방법에는 두 가지가 있습니다. 1) Hard voting 과 2) Soft voting 입니다.

1) Hard voting 을 통한 분류는 다수결 원칙과 비슷합니다.

2) Soft voting 은 분류기들의 레이블 결정 확률을 평균하여 가장 확률이 높은 레이블을 결과값으로 선정합니다. 일반적으로 soft voting 이 주로 사용됩니다.

Scikit-learn 은 voting 방식의 앙상블을 구현한 VotingClassifier 클래스를 제공하고 있습니다. voting 방식의 앙상블을 이용해 악성종양, 양성종양 여부를 결정하는 데이터를 예측해보도록 하겠습니다.

```
import pandas as pd

from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

cancer = load_breast_cancer()

data_df = pd.DataFrame(cancer.data, columns=cancer.feature_names)
data_df.head(3)
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst radius	worst texture	worst perimeter	worst area	worst smoothnes
0	17.99	10.38	122.8	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	...	25.38	17.33	184.6	2019.0	0.1622
1	20.57	17.77	132.9	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	...	24.99	23.41	158.8	1956.0	0.1238
2	19.69	21.25	130.0	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	...	23.57	25.53	152.5	1709.0	0.1444

이 예시에서는 logistic regression 과 k nearest neighbor 를 기반으로 하여 soft voting 방식으로 voting 분류기를 만들어 보겠습니다. Scikit-learn 의 VotingClassifier 클래스의 생성 인자로 estimators 는 voting 에 사용될 여러 classifier 객체들을 입력 받으며, voting 인자는 hard/soft voting 방식을 의미합니다.

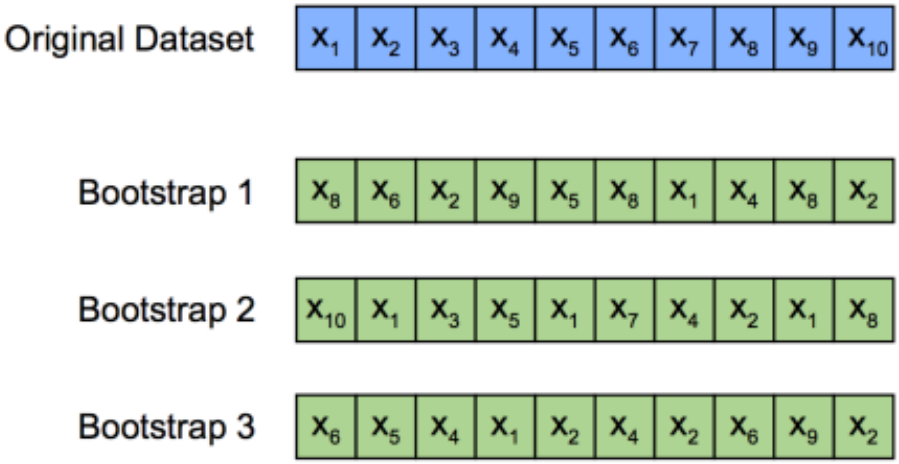
```
# 개별 모델은 로지스틱 회귀와 KNN 임.
lr_clf = LogisticRegression()
knn_clf = KNeighborsClassifier(n_neighbors=8)

# 개별 모델을 소프트 보팅 기반의 앙상블 모델로 구현한 분류기
vo_clf = VotingClassifier( estimators=[('LR',lr_clf),('KNN',knn_clf)] , voting='soft' )

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    test_size=0.2 , random_state= 156)

# VotingClassifier 학습/예측/평가.
vo_clf.fit(X_train , y_train)
pred = vo_clf.predict(X_test)
```

앙상블 방법은 전반적으로 단일 ML 알고리즘보다 뛰어난 예측 성능을 가지는 경우가 많습니다. voting 과 stacking 등은 서로 다른 알고리즘을 기반으로 하고 있지만, bagging 과 bosting 은 대부분 decision tree 알고리즘을 기반으로 합니다. 전 챕터에서 다루었던 결정 트리의 Overfitting 문제는 bagging 과 boosting 에서 수십~수천 개의 매우 많은 분류기의 결합으로 다양한 상황을 학습하게 함으로써 극복하고 있습니다.

Bagging 알고리즘 – Random Forest

Bagging 은 같은 알고리즘으로 여러 개의 분류기를 만들어서 voting 으로 최종 결정하는 알고리즘입니다. Bagging의 대표적인 알고리즘은 random forest 입니다. Random forest 의 기반 알고리즘은 결정 tree 로서, 결정 tree 의 쉽고 직관적인 장점을 그대로 가지고 있습니다.

Random forest의 개별 tree가 학습하는 데이터를 전체 데이터에서 일부가 중첩되게 샘플링 된 데이터입니다. 이렇게 여러 개의 데이터를 중첩되게 분리하는 것을 bootstrapping (bootstrapping) 이라고 합니다. 예를 들어, 원본 데이터 건수가 10 인 학습 데이터셋에 random forest 를 3 개의 결정 트리 기반으로 학습하려고 n_estimators=3 으로 hyper-parameter 를 부여하면 다음과 같이 데이터 subset 이 만들어집니다.



Scikit-learn 은 RandomForestClassifier 클래스를 통해 random forest 기반의 분류를 지원합니다. random forest 의 hyper-parameter 는 decision tree 의 hyper-parameter 와 같은 parameter 가 대부분입니다.

- n_estimators : random forest 에서 decision tree 개수, 많이 설정할수록 무조건 성능이 향상되는 것은 아니며, 학습 수행시간이 길어질 수 있음
- max_depth, min_samples_leaf 등 decision tree 의 hyper-parameter

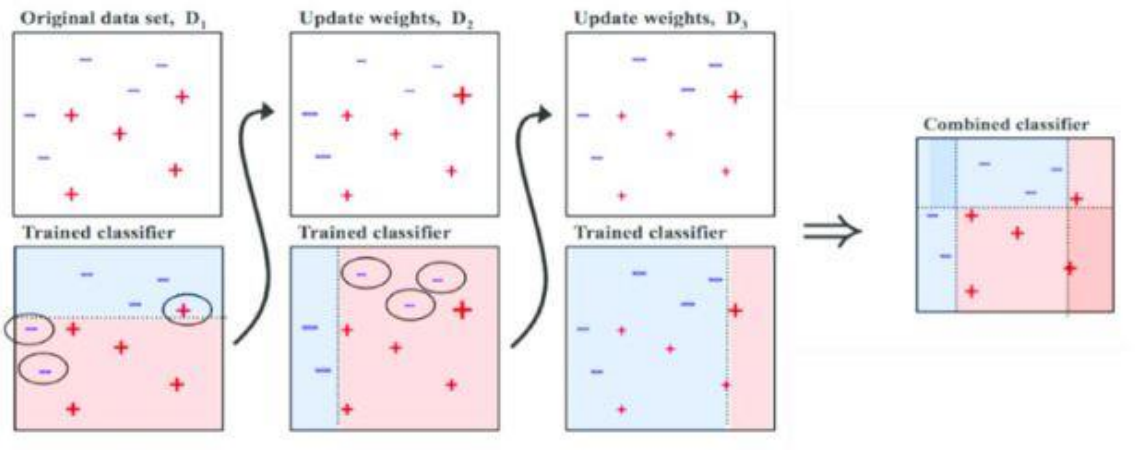
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

# 결정 트리에서 사용한 get_human_dataset( )을 이용해 학습/테스트용 DataFram
e 반환
X_train, X_test, y_train, y_test = get_human_dataset()

# 랜덤 포레스트 학습 및 별도의 테스트 셋으로 예측 성능 평가
rf_clf = RandomForestClassifier(random_state=0)
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))
```

랜덤 포레스트 정확도: 0.9108

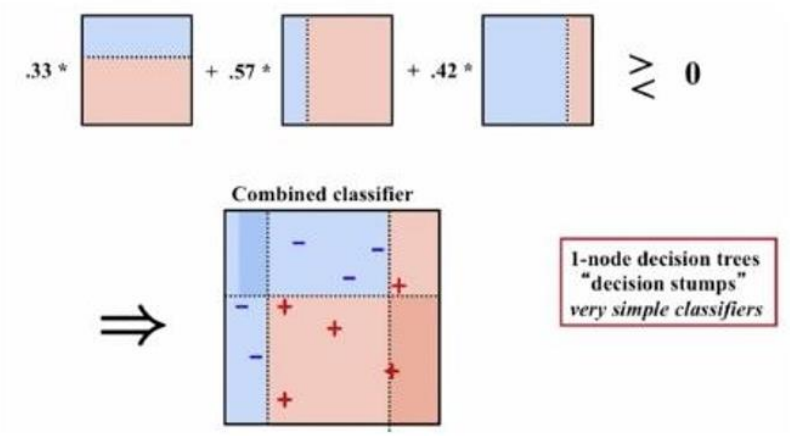
Boosting 알고리즘 -Gradient Boosting Machine, XGBoost

Boosting 알고리즘은 여러 개의 약한 학습기 (weak learner)를 순차적으로 학습-예측하면서 잘못 예측한 데이터에 가중치 부여를 통해 오류를 개선해나가면서 학습하는 방식입니다. 부스팅의 대표적인 알고리즘은 AdaBoost (Adaptive Boosting)와 gradient boost 가 있습니다.

Boosting 알고리즘의 학습 진행 방식에 대하여 자세히 알아보도록 하겠습니다. 먼저 AdaBoost 입니다.



개별 약한 학습기는 다음 그림처럼 각각의 가중치를 부여해 결합하게 되고, 모두 결합해 예측을 수행합니다.



GBM도 AdaBoost와 유사하나, 가중치 업데이트를 경사하강법 (Gradient Descent)를 이용하는 것이 큰 차이입니다. 경사하강법에 대해서는 앞으로 더 자세히 다루겠지만, 여기서는 '반복 수행을 통해 오류를 최소화할 수 있도록 가중치의 업데이트 값을 도출하는 기법'으로 이해하면 좋을 것 같습니다. Scikit-learn 은 GBM 기반 분류를 위해서 GradientBoostingClassifier 클래스를 제공합니다. Hyper-parameter 는 아래와 같습니다.

- loss: 경사하강법에서 사용할 비용함수
- learning_rate: GBM 이 학습을 진행할 때마다 적용하는 학습률, 작은 값을 적용하면 최소 오류 값을 찾아 예측 성능이 높아질 가능성 있음. 하지만 수행시간 오래 걸리고, weak learner 의 반복이 완료될 때까지 최소 오류 값 찾지 못할 가능성 존재
- n_estimators. max_depth, min_samples_leaf, max_features 등 tree 기반 parameter

```
from sklearn.ensemble import GradientBoostingClassifier
gb_clf = GradientBoostingClassifier(random_state=0)
gb_clf.fit(X_train , y_train)
gb_pred = gb_clf.predict(X_test)
```

GBM 은 overfitting 에도 강한 뛰어난 예측 성능을 가진 알고리즘입니다. 하지만 수행 시간이 오래걸리는 단점이 있으며 GBM 이후 이를 기반으로 한 많은 알고리즘들이 새롭게 만들어지고 있습니다. 대표적으로 XGBoost 와 LightGBM 이 있습니다.

Boosting 알고리즘 – XGBoost

먼저 소개할 알고리즘은 XGBoost 입니다. 일반적으로 분류와 회귀 영역에서 뛰어난 예측 성능을 발휘하며, 앞서 언급했던 GBM 의 느린 수행 시간 문제를 해결하여 매우 각광 받고 있습니다.

XGBoost 설치는 Anaconda 를 이용하면 쉽게 할 수 있습니다.

```
conda install -c anaconda py-xgboost     # Windows 기반
```

```
conda install -c conda-forge py-xgboost          #리눅스 기반
```

이제부터 XGBoost 를 통해 악성종양과 양성종양을 분류해보도록 하겠습니다. scikit-learn 에서는 분류를 위해 XGBClassifier, 회귀를 위해 XGBRegressor 를 각각 제공합니다.

먼저, XGBoost 의 hyper-parameter 를 유형별로 나누면 다음과 같습니다.

- 일반 파라미터: thread 의 개수 등의 선택 용, 대부분 default 값을 따름

- 부스트 파라미터: tree 최적화, boosting, regularization 등 관련 parameter, 대부분의 XGBoost parameter 가 속함.
  - learning_rate
  - num_estimators
  - min_child_weight [default=1]: min_child_leaf 와 유사함. overfitting 조절
  - gamma[default=0, alis: min_split_loss]: tree 의 leaf node 를 추가적으로 나눌지를 결정할 최소 loss 감소 값
  - max_depth [default=6]
  - subsample [default=1]: tree 가 overfitting 되는 것을 제어하기 위해 데이터를 샘플링하는 비율. sub_sample=0.5 이면 데이터의 절반을 트리 생성에 사용함.
  - colsample_bytree [default=1]: max_features 와 유사. 매우 많은 feature 가 있는 경우 overfitting 조정에 사용함.

- reg_lambda [default=1]: L2 regularization 적용 값. feature 개수가 많을 경우 사용하는 편이며 값이 클수록 overfitting 감소 효과
- reg_alpha [default=0]: L1 regularization 적용 값. feature 개수가 많을 경우 사용하는 편이며 값이 클수록 overfitting 감소 효과
- scale_pos_weight [default=1]: 특정 값으로 치우친 비대칭 데이터셋의 균형을 맞추기 위함.

- 학습 태스크 파라미터: 학습 수행 시의 객체함수, 평가를 위한 지표
  - objective: loss function
  - eval_metric: 검증에 사용되는 함수. 회귀는 rmse (Root Mean Square Error), 분류는 error (Binary classification error rate)가 default.

overfitting 문제를 해결하기 위해 위 hyper-parameter를 아래와 같이 조정할 수 있습니다.

- learning_rate 값을 낮춥니다(0.01 ~ 0.1). eta 값을 낮출 경우 num_round 는 높여야합니다.
- max_depth 값을 낮춥니다.
- min_child_weight 값을 높입니다.
- gamma 값을 높입니다.

XGBoost 를 통해 악성종양 여부를 분류하는 예시 코드는 아래와 같습니다. n_estimators 는 400, learning_rate는 0.1, max_depth=3으로 설정하여 모델을 정의 후, fit( ), predict( )를 통해 각각 학습, 예측을 수행합니다. 이때 early stopping 반복 횟수를 100, early stopping 평가 지표인 eval_metric은 logloss, eval_set은 성능 평가를 위한 데이터셋으로 설정하였습니다.

```
from xgboost import XGBClassifier

xgb_wrapper = XGBClassifier(n_estimators=400, learning_rate=0.1, max_depth=3)
evals = [(X_test, y_test)]
xgb_wrapper.fit(X_train, y_train, early_stopping_rounds=100, eval_metric="logloss",
                eval_set=evals, verbose=True)
ws100_preds = xgb_wrapper.predict(X_test)
```

이때, XGBoost 의 예측 성능 평가는 아래와 같습니다.

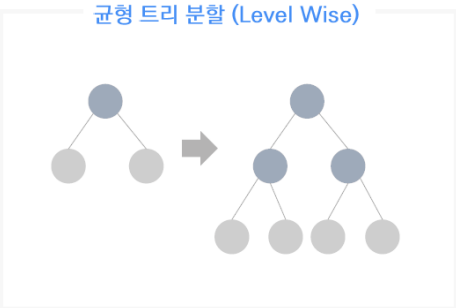
```
정확도: 0.9737, 정밀도: 0.9744, 재현율: 0.9870,    F1: 0.9806, AUC:0.9665
```

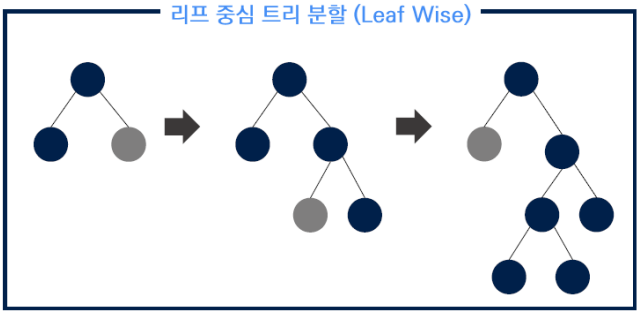
만약 early stopping 반복 횟수를 너무 적게 하면 예측 성능이 저하될 우려가 큽니다. 만일 early_stopping_rounds 를 10 으로 하면 아직 성능이 향상될 여지가 있음에도 불구하고, 10 번동안 성능 평가 지표가 향상되지 않으면 반복이 멈추어버릴 수 있습니다. 아래는 early_stopping_rounds 를 10 으로 하였을 때 정확도입니다.

```
정확도: 0.9561, 정밀도: 0.9615, 재현율: 0.9740,    F1: 0.9677, AUC:0.9465
```

Boosting 알고리즘 - LightGBM

LightGBM 은 XGBoost 와 함께 부스팅 계열 알고리즘에서 가장 각광을 받고 있습니다. LightGBM 은 XGBoost 이후 개발된 모델로, 기존 XGBoost 보다 빠른 학습시간을 갖는다는 장점이 있습니다. LightBGM 은 일반 GBM 계열 tree 분할 방법(Level Wise)과 다르게 Leaf Wise 방식을 사용합니다. Level Wise 방식은 최대한 균형 잡힌 tree 를 유지하면서 분할하는 방식으로 overfitting 에 보다 강하다고 알려져 있습니다. 반대로 Leaf Wise 방식은 tree 의 균형을 맞추기 보단, 최대 손실 값(max delta loss)을 가지는 리프 노드를 지속적으로 분할합니다. Leaf Wise 방식이 Level Wise 보다 예측 오류 손실을 최소화 할 수 있다는 것이 LightGBM 의 구현 사상입니다.





LightGBM 의 XGBoost 대비 특징을 정리하면 아래와 같습니다.

- 더 빠른 학습과 예측 수행 시간
- 더 작은 메모리 사용량
- leaf node 가 계속 분할되면서 tree 가 깊어지므로 max_depth 가 큰 편

LightGBM 의 hyper-parameter 는 XGBoost 와 유사하고, 아래와 같습니다.

- 부스트 파라미터: tree 최적화, boosting, regularization 등 관련 parameter
  - learning_rate [default=0.1]
  - num_estimators: num_boost_round
  - min_child_samples [default=20]: leaf node 가 되기 위해서 최소한으로 필요한 레코드 수. overfitting 조절
  - gamma[default=0, alis: min_split_loss]: tree 의 leaf node 를 추가적으로 나눌지를 결정할 최소 loss 감소 값
  - max_depth [default=-1] : 0 보다 작은 값을 설정하면 깊이 제한이 없음.
  - sub_sample [default=1]: tree 가 overfitting 되는 것을 제어하기 위해 데이터를 샘플링하는 비율. sub_sample=0.5 이면 데이터의 절반을 트리 생성에 사용함.
  - colsample_bytree [default=1]: max_features 와 유사. 매우 많은 feature 가 있는 경우 overfitting 조정에 사용함.

- reg_lambda [default=0.0] L2 regularization 적용 값. feature 개수가 많을 경우 사용하는 편이며 값이 클수록 overfitting 감소 효과
- reg_alpha [default=0.0]: L1 regularization 적용 값. feature 개수가 많을 경우 사용하는 편이며 값이 클수록 overfitting 감소 효과
- boosting [default=gbdt]: boosting tree 를 생성하는 알고리즘
  - gbdt : 일반적인 gradient boosting decision tree
  - rf : random forest

- 학습 태스크 파라미터: 학습 수행 시의 객체함수, 평가를 위한 지표
  - objective: loss function

LightGBM 의 기본적인 hyper-parameter tuning 방안은

1) num_leaves 는 개별 트리가 가질 수 있는 최대 리프의 개수이고, 일반적으로 이 값을 크게하면 정확도는 높아지나, overfitting 의 문제가 발생할 수 있습니다.

num_leaves 의 개수를 중심으로

2) min_child_samples 를 보통 큰 값으로 설정하면 overfitting 을 방지할 수 있습니다.
3) max_depth 도 함께 조정합니다.
4) learning_rate 를 작게 하면서 n_estimators 를 크게하는 것 또한 boosting 계열에서 가장 기본적인 튜닝방안입니다.

LightBGM 은 scikit-learn 과 호환하기 위해 분류를 위한 LGBMClassifier 와 회귀를 위한 LGBMRegressor 클래스를 제공합니다. LightGBM 을 통해 악성종양 여부를 분류하는 예시 코드는 아래와 같습니다. sckn_estimators 는 400, 나머지는 default 로 설정하여 모델을 정의 후, fit( ), predict( )를 통해 각각 학습, 예측을 수행합니다. 이때 early stopping 반복 횟수를 100, early stopping 평가 지표인 eval_metric 은 logloss 로 설정하였습니다.

```
from lightgbm import LGBMClassifier

import pandas as pd
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

dataset = load_breast_cancer()
ftr = dataset.data
target = dataset.target

# 전체 데이터 중 80%는 학습용 데이터, 20%는 테스트용 데이터 추출
X_train, X_test, y_train, y_test=train_test_split(ftr, target, test_size=0.
2, random_state=156 )

# 앞서 XGBoost와 동일하게 n_estimators는 400 설정.
lgbm_wrapper = LGBMClassifier(n_estimators=400)

# LightGBM도 XGBoost와 동일하게 조기 중단 수행 가능.
evals = [(X_test, y_test)]
lgbm_wrapper.fit(X_train, y_train, early_stopping_rounds=100, eval_metric
="logloss",
                 eval_set=evals, verbose=True)
preds = lgbm_wrapper.predict(X_test)
```

XGBoost, LightGBM 분류 실습

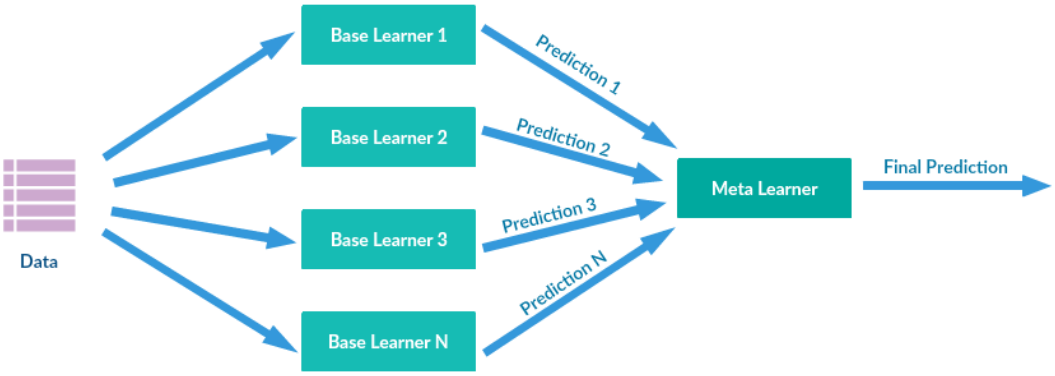
캐글 산탄데르 고객만족 예측

https://www.kaggle.com/c/santander-customer-satisfaction/data

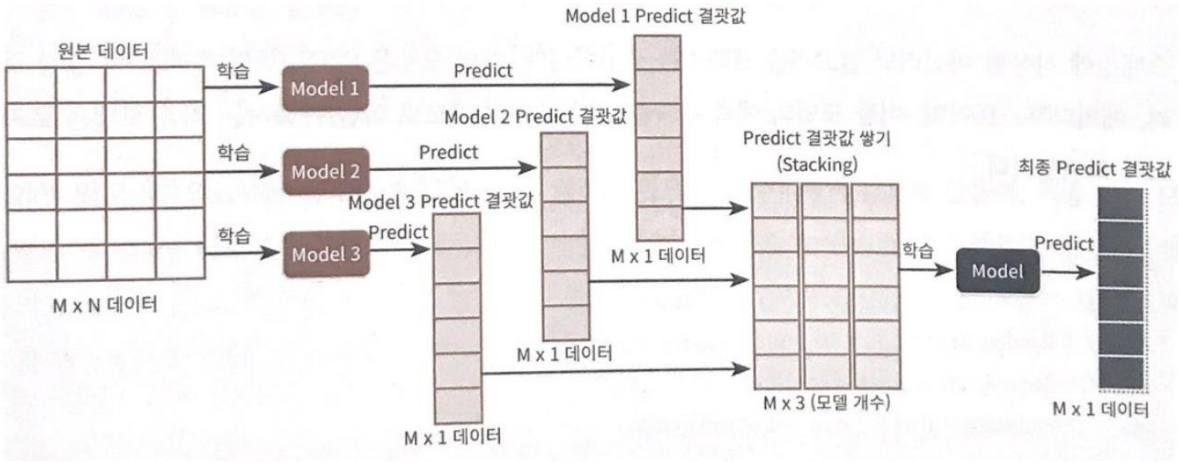
Stacking 알고리즘

stacking 은 개별 알고리즘으로 예측한 데이터를 기반으로 다시 예측을 수행한다는 점에서 앞서 언급한 알고리즘들과 차이점을 가지고 있습니다.

stacking 은 두 종류의 모델이 필요합니다. 첫 번째는 개별적인 기반 모델이고, 두 번째는 첫 번째의 개별 모델의 예측 데이터를 학습 데이터로 만들어 학습하는 최종 메타모델입니다. stacking 모델의 구조는 아래와 같습니다.



m 개의 row, n 개의 feature (column)을 가진 데이터셋에 stacking 앙상블을 적용한다고 가정하겠습니다. 그리고 학습에 사용될 level 1 모델은 3 개 입니다. 먼저 level 1 모델 별로 각각 학습 시킨 뒤 예측을 수행하면 각각 m 개의 row 를 가진 1 개의 label 값을 도출할 것입니다. 모델별로 도출된 예측 label 값을 다시 합해서 (stacking) 새로운 데이터셋을 만들고 최종 모델을 적용하여 최종 예측하는 것이 stacking 앙상블 모델입니다.



stacking 으로 악성종양을 판별하는 예제를 통해 개념을 더 파악해보도록 하겠습니다. 먼저 stacking 에 사용될 개별 알고리즘은 KNN, random forest, decision tree, adaboost 이며, 이들에 대한 최종 모델은 logistic regression 입니다.

우선 개별 모델을 정의하고, 각각을 학습 시킨 후 예측 결과들을 비교합니다.

```
# 개별 ML 모델을 위한 Classifier 생성.
knn_clf  = KNeighborsClassifier(n_neighbors=4)
rf_clf = RandomForestClassifier(n_estimators=100, random_state=0)
dt_clf = DecisionTreeClassifier()
ada_clf = AdaBoostClassifier(n_estimators=100)

# 최종 Stacking 모델을 위한 Classifier생성.
lr_final = LogisticRegression(C=10)
```

```
# 개별 모델들을 학습.
knn_clf.fit(X_train, y_train)
rf_clf.fit(X_train , y_train)
dt_clf.fit(X_train , y_train)
ada_clf.fit(X_train, y_train)
```

```
knn_pred = knn_clf.predict(X_test)
rf_pred = rf_clf.predict(X_test)
dt_pred = dt_clf.predict(X_test)
ada_pred = ada_clf.predict(X_test)
```

이때 각 모델들의 예측 결과는 KNN, random forest, decision tree, adaboost 순, 0.92, 0.96, 0.90, 0,95 입니다.

개별 알고리즘으로부터 예측된 예측값을 column 레벨로 옆으로 붙여서 feature 값으로 만들어, 최종 메타 모델의 학습데이터로 만들어보도록 하겠습니다.

```
pred = np.array([knn_pred, rf_pred, dt_pred, ada_pred])
print(pred.shape)

# transpose를 이용해 행과 열의 위치 교환. 컬럼 레벨로 각 알고리즘의 예측 결과를 피처로 만듦.
pred = np.transpose(pred)
print(pred.shape)
```

```
(4, 114)
(114, 4)
```

이렇게 예측데이터로 생성된 데이터셋을 기반으로 최종 메타모델인 로지스틱 회귀를 학습하고 예측 정확도를 측정한 결과, 개별 모델에 비해 정확도가 보다 상승하였습니다. (물론 이러한 stacking 기법으로 무조건 개별모델보다 예측 성능이 좋아진다는 보장은 없습니다.)

```
lr_final.fit(pred, y_test)
final = lr_final.predict(pred)

print('최종 메타 모델의 예측 정확도:
```

```
최종 메타 모델의 예측 정확도: 0.9737
```