Random Forest (1)

의사결정트리를 사용하는 가장 대표적인 배깅 모델

의사결정트리의 단점(과적합이 자주 발생)을 보완하고 장점은 유지한다.

최근 XGBoost, LightGBM, CatBoost와 함께 주목받는 알고리즘 중 하나.

> 예제 진행을 위해 `imblearn` 패키지의 설치가 필요하다

#01. 패키지 참조

```
import warnings
warnings.filterwarnings('ignore')

from matplotlib import pyplot as plt
import seaborn as sb
from pandas import read_excel, DataFrame, melt
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score, cross_validate
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV

# 데이터 불균형 해소를 위한 패키지
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import RandomOverSampler
from imblearn.over_sampling import SMOTE
```

#02. 데이터 가져오기

필드이름	설명
target	타겟(종속)변수 `(Class_1~Class_9)`
feat_1 ~ feat_93	설명(독립)변수

```
origin = read_excel("https://data.hossam.kr/G02/otto_train.xlsx")
origin.head()
```

	feat_1	feat_2	feat_3	feat_4	feat_5	feat_6	feat_7	feat_8	feat_9	feat_10	...	feat_
0	1	0	0	0	0	0	0	0	0	0	...	1
1	0	0	0	0	0	0	0	1	0	0	...	0
2	0	0	0	0	0	0	0	1	0	0	...	0
3	1	0	0	1	6	1	5	0	0	1	...	0

	feat_1	feat_2	feat_3	feat_4	feat_5	feat_6	feat_7	feat_8	feat_9	feat_10	...	feat_
4	0	0	0	0	0	0	0	0	0	0	...	1

5 rows × 94 columns

#02. 데이터 전처리

타겟변수 라벨링

```
origin['target'] = origin['target'].map({
    'Class_1': 0,
    'Class_2': 1,
    'Class_3': 2,
    'Class_4': 3,
    'Class_5': 4,
    'Class_6': 5,
    'Class_7': 6,
    'Class_8': 7,
    'Class_9': 8
})

origin['target'].value_counts().sort_index()
```

```
target
0     1929
1    16122
2     8004
3     2691
4     2739
5    14135
6     2839
7     8464
8     4955
Name: count, dtype: int64
```

> 각 class별로 데이터 불균형이 보이므로 이에 대한 처리가 필요하다.

독립/종속 변수 분리

```
x = origin.drop(['target'], axis=1)
y = origin['target']
x.shape, y.shape
```

```
((61878, 93), (61878,))
```

데이터 표준화

> 이 데이터셋은 생략

훈련/검증 데이터 분할

```
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.3, random_state=777)
x_train.shape, x_test.shape, y_train.shape, y_test.shape
```

```
((43314, 93), (18564, 93), (43314,), (18564,))
```

데이터 불균형 해소

1) Under Sampling 방식 - Random Under Sampler

많은 비율을 차지하는 다수 집단에서 일부만 샘플링하는 방식

소수 집단의 데이터가 어느 정도 확보되었다고 여겨질 때, 다수 집단의 데이터를 줄여서 균형을 맞춘다.

다수 집단의 유의미한 데이터를 손실할 수 있다는 단점이 있다.

`sampling_strategy` 파라미터

값	설명
`majority`	다수 클래스만 다시 샘플링
`not majority`	`다수 아님` - 다수 클래스를 제외한 모든 클래스를 다시 샘플링
`not minority`	`소수 아님` - 소수 클래스를 제외한 모든 클래스를 다시 샘플링
`all`	모든 클래스를 다시 샘플링
`auto`	자동 처리

```
undersampler = RandomUnderSampler(sampling_strategy="majority", random_state=777)
x_under, y_under = undersampler.fit_resample(x_train, y_train)
print(x_under.shape, y_under.shape)

y_under.value_counts().sort_index()
```

```
(33336, 93) (33336,)
```

```
target
0    1339
1    1339
2    5598
3    1864
4    1914
5    9896
6    1956
7    5941
8    3489
Name: count, dtype: int64
```

2. Over Sampling - Random Over Sampler

소수 집단에서 복원 추출을 수행하는 방법.

언더 샘플링처럼 데이터 중 일부를 취하는 것은 아니기 때문에 데이터 손실은 발생하지 않지만, 동일한 데이터를 여러번 학습 데이터에 포함시키므로 학습 정확도는 높지만 과적합 리스크가 크다.

`sampling_strategy` **파라미터**

값	설명
`minority`	소수 클래스만 다시 샘플링
`not majority`	`다수 아님` - 다수 클래스를 제외한 모든 클래스를 다시 샘플링
`not minority`	`소수 아님` - 소수 클래스를 제외한 모든 클래스를 다시 샘플링
`all`	모든 클래스를 다시 샘플링
`auto`	자동 처리

```
oversampler = RandomOverSampler(sampling_strategy="minority", random_state=777)
x_over, y_over = oversampler.fit_resample(x_train, y_train)
print(x_over.shape, y_over.shape)

y_over.value_counts().sort_index()
```

```
(53292, 93) (53292,)
```

```
target
0    11317
1    11317
2     5598
3     1864
4     1914
5     9896
6     1956
7     5941
8     3489
Name: count, dtype: int64
```

3. Over Sampling - SMOTE

소수 집단의 데이터를 바탕으로 새로운 데이터를 생성.

단순히 소수 집단의 데이터를 복원 추출하는 것이 아니라 소수 집단 데이터를 분석해 어떤 특징이 있는지 살피고 그와 유사한 패턴을 갖는 가짜 데이터를 생성한다.

`sampling_strategy` **파라미터**

값	설명
`minority`	소수 클래스만 다시 샘플링
`not majority`	`다수 아님` - 다수 클래스를 제외한 모든 클래스를 다시 샘플링
`not minority`	`소수 아님` - 소수 클래스를 제외한 모든 클래스를 다시 샘플링
`all`	모든 클래스를 다시 샘플링

값	설명
`auto`	자동 처리

혹은 실수 타입으로 설정할 경우 샘플 수의 비율을 의미

`k_neighbors` **파라미터** (int)

합성 샘플을 생성하는데 사용할 샘플의 가장 가까운 이웃 수 (기본값=5)

```
# "sampling_strategy" can be a float only when the type of target is binary
smote_sampler = SMOTE(sampling_strategy="minority", random_state=777)
x_sm, y_sm = smote_sampler.fit_resample(x_train, y_train)
print(x_sm.shape, y_sm.shape)

y_sm.value_counts().sort_index()
```

```
(53292, 93) (53292,)
```

```
target
0    11317
1    11317
2     5598
3     1864
4     1914
5     9896
6     1956
7     5941
8     3489
Name: count, dtype: int64
```

#03. 랜덤포레스트 모델 적합

단일 모델 만들기

```
#rfc = RandomForestClassifier(n_estimators=20, max_depth=5, random_state=777)
#rfc = RandomForestClassifier(n_estimators=50, max_depth=30, random_state=777)
#rfc = RandomForestClassifier(n_estimators=100, max_depth=30, random_state=777)
rfc = RandomForestClassifier(n_estimators=100, max_depth=100, random_state=777)

# 원본 데이터로 학습 진행
#rfc.fit(x_train, y_train)
#print("훈련 정확도:", rfc.score(x_train, y_train))

# 언더샘플링 데이터로 학습 진행
#rfc.fit(x_under, y_under)
#print("훈련 정확도:", rfc.score(x_under, y_under))

# 오버샘플링 데이터로 학습 진행
#rfc.fit(x_over, y_over)
#print("훈련 정확도:", rfc.score(x_over, y_over))

# SMOTE 데이터로 학습 진행
```

```
rfc.fit(x_sm, y_sm)
print("훈련 정확도:", rfc.score(x_sm, y_sm))

print("테스트 정확도:", rfc.score(x_test, y_test))
```

```
훈련 정확도: 0.9999437063724387
테스트 정확도: 0.7989118724412843
```

하이퍼파라미터 튜닝

```
rfc = RandomForestClassifier(random_state=777)

params = {
    "n_estimators": [20, 50, 100],
    "max_depth": [5, 30, 100]
}

grid = GridSearchCV(rfc, param_grid=params, cv=5, n_jobs=-1)
grid.fit(x_train, y_train)

print("최적의 하이퍼 파라미터:", grid.best_params_)
print("최대 훈련 정확도:", grid.best_score_)

y_pred = grid.best_estimator_.predict(x_test)
print("최대 검증 정확도:", accuracy_score(y_test, y_pred))

result_df = DataFrame(grid.cv_results_['params'])
result_df['mean_test_score'] = grid.cv_results_['mean_test_score']
result_df.sort_values(by='mean_test_score', ascending=False)
```

```
최적의 하이퍼 파라미터: {'max_depth': 100, 'n_estimators': 100}
최대 훈련 정확도: 0.7999030982674508
최대 검증 정확도: 0.807207498383969
```

	max_depth	n_estimators	mean_test_score
8	100	100	0.799903
7	100	50	0.796832
5	30	100	0.795747
4	30	50	0.792608
3	30	20	0.785820
6	100	20	0.785751
0	5	20	0.612758
2	5	100	0.606709
1	5	50	0.606571