4주차

날짜	@2024년 10월 29일
상태	완료

Chaper.5 트리 알고리즘

▼ 5-1

1. 로지스틱 회귀로 와인 분류

- **데이터 준비**: 알코올 도수, 당도, pH 값을 입력으로, 클래스(레드 와인: 0, 화이트 와인: 1)를 출력으로 사용.
- **데이터 전처리**: `StandardScaler`로 특성 값을 표준화하고 훈련 및 테스트 세트로 나눔.
- **로지스틱 회귀 모델 훈련 및 평가**: 과소적합 문제가 발생하며, 해결을 위해 규제 매개변수 `C` 조정이나 다항 특성 추가 등을 고려.

2. 결정 트리 모델

- **결정 트리 장점**: 트리의 분할 기준이 직관적이며, 특정 특성 값을 기준으로 샘플을 분할하므로 해석이 쉬움.
- **훈련 및 평가**: `DecisionTreeClassifier`로 모델 훈련, 시각화(`plot_tree()`)를 통해 트리 구조와 분류 기준 확인.

3. 지니 불순도 (Gini Impurity)

- **불순도 계산**: 지니 불순도는 각 클래스 비율의 제곱을 더한 값에 1을 빼서 계산.
- 결정 트리는 지니 불순도를 기준으로 샘플을 최대한 순수하게 나눔.

4. 과대적합 방지 - 가지치기

- **가지치기**: 트리의 깊이를 제한(`max_depth`)해 과대적합을 방지하고, 훈련 성능은 낮아졌지만 테스트 성능은 유지.

- **특성 스케일 영향 없음**: 결정 트리는 특성의 스케일에 민감하지 않으므로 표준화 없이도 정확한 분류가 가능.

5. 특성 중요도

- **특성 중요도 계산**: `feature_importances_` 속성을 통해 각 특성이 모델 성능에 기여하는 중요도를 확인.
- **결과**: 당도가 가장 높은 특성 중요도를 가지며, 결정 트리 알고리즘은 특성 선택에 활용 가능.

결정 트리는 설명 가능한 모델로, 과대적합 방지를 위한 가지치기와 특성 중요도 제공 등 다양한 장점을 가짐.

▼ 5-2

1. 검증 세트와 교차 검증

- **검증 세트**: 훈련 세트를 나누어 검증용으로 사용하여 모델의 과대적합과 과소적합을 확인.
- **교차 검증**: 데이터를 여러 폴드로 나누어 반복적으로 검증 세트를 바꿔가며 점수를 평균해 안정적인 성능 평가를 얻음. 5-폴드, 10-폴드 등 사용 가능.

```
from sklearn.model_selection import cross_validate
scores = cross_validate(dt, train_input, train_target)
print(np.mean(scores['test_score']))  # 평균 검증 점수
```

2. 하이퍼파라미터 튜닝

- **그리드 서치**: `GridSearchCV`를 사용해 설정된 매개변수 범위를 탐색하고 최적의 조합을 찾음. 예) `min_impurity_decrease`를 조정하여 최적화.

```
from sklearn.model_selection import GridSearchCV
params = {'min_impurity_decrease': [0.0001, 0.0002, 0.00
03, 0.0004, 0.0005]}
gs = GridSearchCV(DecisionTreeClassifier(random_state=4
2), params, n_jobs=-1)
gs.fit(train_input, train_target)
print(gs.best_params_)  # 최적 매개변수 출력
```

3. 복잡한 매개변수 튜닝

- **복합 그리드 서치**: 여러 매개변수를 동시에 조정하여 최적 조합 탐색.

```
params = {
    'min_impurity_decrease': np.arange(0.0001, 0.001, 0.
0001),
    'max_depth': range(5, 20, 1),
    'min_samples_split': range(2, 100, 10)
}
gs = GridSearchCV(DecisionTreeClassifier(random_state=4
2), params, n_jobs=-1)
gs.fit(train_input, train_target)
print(gs.best_params_)  # 최적 매개변수 조합
```

4. 랜덤 서치

- **랜덤 서치**: `RandomizedSearchCV`를 사용하여 특정 매개변수 범위 내에서 임의로 선택하여 탐색함. 큰 매개변수 조합을 탐색할 때 유리.

```
from scipy.stats import uniform, randint
params = {
    'min_impurity_decrease': uniform(0.0001, 0.001),
    'max_depth': randint(20, 50),
    'min_samples_split': randint(2, 25),
    'min_samples_leaf': randint(1, 25)
}
gs = RandomizedSearchCV(DecisionTreeClassifier(random_st
ate=42), params, n_iter=100, n_jobs=-1)
gs.fit(train_input, train_target)
print(gs.best_params_)  # 최적 매개변수 조합
```

5. 최종 모델 평가

- **최적 모델 평가**: 최적의 하이퍼파라미터로 훈련된 모델의 성능을 테스트 세트에서 평가함.

```
dt = gs.best_estimator_
print(dt.score(test_input, test_target))  # 최종 테스트 세
트 성능
```

랜덤 서치를 통해 효율적으로 매개변수를 튜닝하고, 최적 모델을 테스트 세트에서 평가하여 최종 성능을 확인함.

▼ 5-3

1. 정형 데이터와 비정형 데이터

- **정형 데이터**: 구조화된 데이터로 CSV나 데이터베이스에 저장 가능. 예: 와인 데이터.
- **비정형 데이터**: 구조화하기 어려운 데이터로 텍스트, 이미지, 음성 등이 포함됨.

2. 랜덤 포레스트

- 여러 **결정 트리**를 랜덤한 데이터로 훈련하여 예측을 합산하는 **앙상블 기법**.
- **부트스트랩 샘플**로 중복을 허용하여 랜덤한 훈련 세트를 구성하며, 노드 분할 시 일부 특성을 무작위로 선택해 과대적합을 방지.
- 교차 검증을 통한 성능 평가 시 훈련 세트에 과대적합 경향을 보임.

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(oob_score=True, random_state
=42, n_jobs=-1)
rf.fit(train_input, train_target)
print(rf.oob_score_)  # OOB 점수로 교차 검증 대체 가능
```

3. 엑스트라 트리

- 랜덤 포레스트와 유사하지만, **부트스트랩 샘플을 사용하지 않고** 전체 훈련 세트를 활용.
- 노드 분할 시 무작위로 선택하여 빠른 계산 속도가 장점.

```
from sklearn.ensemble import ExtraTreesClassifier
et = ExtraTreesClassifier(n_jobs=-1, random_state=42)
et.fit(train_input, train_target)
print(et.feature_importances_)  # 특성 중요도 계산
```

4. 그레이디언트 부스팅

- 얕은 결정 트리를 연속적으로 추가하여 이전 트리의 오차를 보완. 과대적합에 강하고 높은 일반화 성능을 제공.

- `n_estimators`와 `learning_rate`를 조정하여 성능 최적화 가능.

```
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier(n_estimators=500, learni
ng_rate=0.2, random_state=42)
gb.fit(train_input, train_target)
print(gb.feature_importances_)  # 특성 중요도 계산
```

5. 히스토그램 기반 그레이디언트 부스팅 (HGB)

- 데이터를 256개 구간으로 나누어 빠르게 학습하는 **효율적인 부스팅 기법**.
- 특성 중요도는 `permutation_importance()` 함수를 사용해 계산.

```
from sklearn.ensemble import HistGradientBoostingClassif
ier
hgb = HistGradientBoostingClassifier(random_state=42)
hgb.fit(train_input, train_target)
from sklearn.inspection import permutation_importance
result = permutation_importance(hgb, train_input, train_
target, n_repeats=10, random_state=42)
print(result.importances_mean)  # 특성 중요도 계산
```

최종 평가:

- HGB는 과대적합을 잘 억제하며 높은 성능을 보임.