3장 회귀 알고리즘과 모델 규제

3-1: k-최근접 이웃 회귀

가장 직관적인 k-최근접 이웃(k-NN) 알고리즘을 통해 회귀의 기본 원리를 이해합니다.

핵심 개념과 동작 원리

k-NN 회귀는 예측하려는 데이터 주변의 **가장 가까운 k개 이웃의 정보를 활용**하는 방식입니다.

1. **동작 방식**: 새로운 데이터 X에 대한 예측을 요청받으면, 훈련 세트에서 X와 가장 가까운 샘플 k개를 찾습니다. 그리고 그 k개 샘플의 **타겟 값(y)을 산술 평균**하여 X의 예측값으로 삼습니다.
2. **사례 기반 학습**: 별도의 수학적 모델을 구축하는 대신, 훈련 데이터 자체를 모델로 사용합니다.

모델 성능 평가: 결정 계수 (R^2)

회귀 모델의 성능은 결정 계수(R^2)로 평가하며, 이는 모델이 타겟의 분산을 얼마나 잘 설명하는지를 나타냅니다.

- **의미**: R^2 값은 1에 가까울수록 좋습니다. 만약 R^2가 0이라면, 모델이 단순히 타겟 전체의 평균으로 예측하는 것과 같은 수준임을 의미합니다. 음수가 나오면 평균으로 예측하는 것보다도 성능이 나쁘다는 뜻입니다.

$$R^2 = 1 - \frac{(\text{타깃} - \text{예측})^2}{(\text{타깃} - \text{평균})^2}$$

과소적합과 과대적합

모델의 복잡도는 이웃의 개수인 `n_neighbors(k)` 로 조절됩니다.

- **과소적합 (Underfitting)**: **k가 너무 크면** 모델이 지나치게 단순해집니다. 너무 많은 이웃의 평균을 내다보니, 예측값은 전체 데이터의 평균에 가까워지며 데이터의 지역적 특성을 놓칩니다. 예측선이 매우 부드러운 직선처럼 변합니다.
- **과대적합 (Overfitting)**: **k가 너무 작으면** 모델이 매우 복잡해집니다. 소수의 이웃에만 의존하므로 훈련 데이터의 작은 노이즈에도 민감하게 반응합니다. 이로 인해 예측선이 훈련 데이터 포인트를 하나하나 따라가려는 듯이 매우 구불구불한 형태로 나타납니다.

한계점

k-NN 회귀는 **훈련 세트의 범위를 벗어나는 데이터를 예측할 수 없습니다.** 예를 들어, 훈련 데이터에 있는 가장 큰 생선의 무게가 1kg이라면, 아무리 더 큰 생선에 대한 예측을 요청해도 예측값은 1kg을 넘을 수 없습니다. 새로운 데이터의 이웃은 항상 훈련 세트 내에 존재하기 때문입니다

```
from sklearn.neighbors import KNeighborsClassifier

# k-NN 모델 생성 및 훈련
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# 예측 수행
y_pred = knn.predict(X_test)
```

3-2: 선형 회귀와 다항 회귀

k-NN의 한계를 극복하고 데이터의 **전반적인 추세**를 학습하기 위해 **선형 회귀**를 사용합니다.

선형 회귀 (Linear Regression)

데이터를 가장 잘 대표하는 **하나의 직선**을 학습하는 알고리즘입니다.

- **학습 원리**: 최소 제곱법을 사용하여, 모든 데이터 포인트와 직선 사이의 거리(오차) 제곱의 합이 최소가 되는 계수(coefficient, 기울기)와 절편(intercept)을 찾습니다.
- **수학적 표현**: y=ax+b (특성이 하나일 때)

다항 회귀 (Polynomial Regression)

데이터가 직선이 아닌 **곡선 형태**의 관계를 보일 때 사용합니다.

- **학습 원리**: 다항 회귀는 새로운 모델이 아니라, **입력 데이터를 변환**하여 선형 회귀를 적용하는 기법입니다. 기존 특성 `x`로부터 `x²`, `x³` 와 같은 **고차항 특성을 새롭게 생성**합니다.
- **특성 변환**: `PolynomialFeatures` 변환기를 사용하여 이 과정을 자동화할 수 있습니다. 예를 들어, `[길이]`라는 특성을 `[길이, 길이²]`라는 두 개의 특성으로 확장합니다.
- **결과**: 확장된 특성(`길이` , `길이²`)을 입력으로 선형 회귀를 학습시키면, 모델은 y=a·(길이²)+b·(길이)+c 형태의 **2차 함수(포물선) 그래프**를 학습하게 됩니다.

```
# 1. 라이브러리 임포트
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

# 단순 선형 회귀
lr = LinearRegression()
lr.fit(X_train, y_train)

# 다항 회귀
# 2차항 특성 생성기
poly = PolynomialFeatures(degree=2, include_bias=False)

# 훈련 데이터를 2차 다항 특성으로 변환
train_poly = poly.fit_transform(X_train)
test_poly = poly.transform(X_test)

# 변환된 데이터로 선형 회귀 모델을 다시 훈련
lr_poly = LinearRegression()
lr_poly.fit(train_poly, y_train)
```

3-3: 특성 공학과 규제

더 복잡한 모델을 만들기 위해 특성을 늘리고(특성 공학), 이로 인해 발생하는 극심한 과대적합을 제어하기 위해 **규제** 기법을 적용합니다.

특성 공학 (Feature Engineering) & 다중 회귀

- **다중 회귀 (Multiple Regression)**: `길이` 뿐만 아니라 `높이`, `두께` 등 여러 개의 특성을 동시에 사용하여 예측하는 회귀 모델입니다.
- **특성 공학**: 다항 회귀에서처럼 기존 특성들을 조합하여(예: `길이 * 높이`, `높이²`) 훨씬 더 많은 새로운 특성을 만들어내는 과정입니다. 특성이 많아질수록 모델은 복잡해지고 훈련 데이터에 더 잘 맞출 수 있지만, 과대적합의 위험 또한 기하급

수적으로 커집니다.

규제 (Regularization)

규제는 모델의 계수(가중치) 값이 너무 커지지 않도록 **페널티**를 부과하여 과대적합을 억제하는 핵심적인 기법입니다.

- **데이터 스케일링의 중요성**: 규제는 계수의 크기에 페널티를 주므로, 특성마다 값의 범위(스케일)가 다르면 불공평한 페널티가 적용됩니다. 따라서 규제 적용 전 `StandardScaler`**를 이용해 모든 특성의 스케일을 표준화**하는 과정이 **필수적**입니다. `fit_transform()`은 훈련 세트에만, `transform()`은 테스트 세트에 적용하여 데이터 유출을 방지해야 합니다.
- **릿지(Ridge) 회귀 (L2 규제)**: 모든 계수의 **제곱**에 페널티를 부과합니다.
  - **특징**: 계수 값을 0에 매우 가깝게 만들지만 완전히 0으로 만들지는 않습니다. 여러 특성들이 예측에 비슷한 영향을 줄 때 안정적인 성능을 보입니다.
- **라쏘(Lasso) 회귀 (L1 규제)**: 모든 계수의 **절댓값**에 페널티를 부과합니다.
  - **특징**: 중요하지 않다고 판단되는 특성의 계수를 **완전히 0으로 만들어** 모델에서 제외시키는 **자동 특성 선택** 효과가 있습니다.
- **하이퍼파라미터** `alpha`: 규제의 강도를 조절하는 값입니다. `alpha`가 클수록 규제가 강해져 모델이 단순해지고(과소적합 방향), 작을수록 규제가 약해져 모델이 복잡해집니다(과대적합 방향). 최적의 `alpha` 값은 훈련 점수와 테스트 점수가 가장 가깝고 높은 지점을 찾아 결정합니다.

```
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import Ridge, Lasso

# 특성 공학
poly = PolynomialFeatures(degree=5, include_bias=False)
train_poly = poly.fit_transform(X_train)
test_poly = poly.transform(X_test)

# 스케일링 (표준화)
scaler = StandardScaler()
train_scaled = scaler.fit_transform(train_poly)
test_scaled = scaler.transform(test_poly)

# 릿지(Ridge) 모델
ridge = Ridge(alpha=0.1)
ridge.fit(train_scaled, y_train)

#라쏘(Lasso) 모델
lasso = Lasso()
lasso.fit(train_scaled, y_train)
```