K-최근접 이웃 (K-Nearest Neighbors, KNN)

- 가장 간단하고 직관적인 머신러닝 알고리즘 중 하나로, '유유상종'이라는 개념에 기반
- KNN은 새로운 데이터가 주어졌을 때, 기존 훈련 데이터 중에서 가장 가까운 **K개의 이웃**을 찾고, 그 이웃들의 클래스를 참고하여 새로운 데이터의 클래스를 결정
- **동작 방식 (분류)**:
    1. 새로운 데이터 포인트와 모든 훈련 데이터 포인트 사이의 거리를 계산합니다.
    2. 계산된 거리 중 가장 짧은 K개의 이웃을 선택합니다.
    3. K개의 이웃들이 가장 많이 속해 있는 클래스(다수결 투표)를 새로운 데이터의 클래스로 예측합니다.

KNN은 별도의 모델을 훈련하는 과정이 없기 때문에 **게으른 학습(Lazy Learning)** 또는 사례 기반 학습(Instance-based Learning)이라고도 불립니다. 모델 파라미터를 학습하는 대신, 훈련 데이터셋 자체를 모델로 사용합니다.

적용 가능한 상황

- 데이터의 차원이 낮고, 데이터 간의 경계가 복잡하지 않은 경우에 간단하게 적용해볼 수 있습니다.
- 모델의 해석이 필요 없고, 빠른 프로토타이핑이 필요할 때 유용합니다.
- 데이터의 분포에 대한 특별한 가정이 필요 없는 비모수(non-parametric) 모델이 필요할 때 사용합니다.

구현 방법

`scikit-learn`의 `neighbors` 모듈에 있는 `KNeighborsClassifier` 클래스를 사용합니다.

주의사항

- **특성 스케일링 필수**
    - KNN은 거리를 기반으로 동작하므로, 특성들의 스케일이 다르면 큰 값을 가진 특성이 거리에 큰 영향을 미침
    - 따라서 **반드시 `StandardScaler` 등으로 스케일링**하여 모든 특성의 단위를 맞춰줘야함
- **최적의 K 찾기**: 이웃의 수 `K`는 모델의 성능에 결정적인 영향을 미칩니다.
    - `K`가 너무 작으면: 이상치나 노이즈에 민감해져 모델이 과적합(Overfitting)될 수 있습니다.
    - `K`가 너무 크면: 다른 클래스의 이웃들이 많이 포함되어 모델이 과소적합(Underfitting)될 수 있습니다.
    - 보통 K는 홀수로 설정하여 동점(tie) 상황을 방지하며, 교차 검증을 통해 최적의 K를 찾는 것이 중요합니다.
- **계산 비용**: 예측 시, 모든 훈련 데이터와의 거리를 계산해야 하므로 데이터가 매우 클 경우 예측 속도가 느려질 수 있습니다.

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, GridSearchCV
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# 1. 데이터 준비 및 전처리
iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42, stratify=y)

# KNN은 스케일링이 매우 중요
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 2. KNN 모델 학습
# KNeighborsClassifier 주요 하이퍼파라미터
# n_neighbors: 이웃의 수 (K). (기본값=5)
# weights: 예측에 사용될 가중치. 'uniform'(동일 가중치), 'distance'(거리에 반비례하는
가중치).
# metric: 거리 측정 방법. 'minkowski'(기본값, p=2이면 유클리드 거리), 'euclidean',
'manhattan', 'chebyshev' 등.
# p: minkowski 거리의 파라미터. (기본값=2)
knn_clf = KNeighborsClassifier(n_neighbors=5)
knn_clf.fit(X_train_scaled, y_train)
knn_pred = knn_clf.predict(X_test_scaled)
print(f"KNN (K=5) 정확도: {accuracy_score(y_test, knn_pred):.3f}") # 0.911

# 3. GridSearchCV를 이용한 최적 K 찾기
param_grid = {'n_neighbors': np.arange(1, 30)}

grid_knn = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5,
scoring='accuracy')
grid_knn.fit(X_train_scaled, y_train)

print(f"\nGridSearchCV 최적 K: {grid_knn.best_params_['n_neighbors']}")     # 14
print(f"최적 K 적용 시 정확도: {grid_knn.best_score_:.3f} (교차검증 평균)") # 0.971

# 최적 모델로 예측
best_knn = grid_knn.best_estimator_
best_pred = best_knn.predict(X_test_scaled)
print(f"Best KNN 정확도: {accuracy_score(y_test, best_pred):.3f}") # 0.956
```

결과 해석 방법

- KNN은 별도의 모델 파라미터를 학습하지 않으므로, 로지스틱 회귀나 결정 트리처럼 모델 자체를 해석하기는 어렵습니다.
- 모델의 성능은 정확도, 혼동 행렬 등 일반적인 분류 평가지표를 통해 평가합니다.
- 어떤 K값에서 가장 좋은 성능이 나오는지를 확인하여 데이터의 복잡도(결정 경계의 복잡성)를 간접적으로 유추할 수 있습니다.

장단점 및 대안

- **장점**:
  - 모델이 매우 간단하고 직관적이어서 이해하기 쉽습니다.
  - 별도의 훈련 과정이 없어 빠릅니다 (단, 예측 시에는 느릴 수 있음).
  - 데이터 분포에 대한 가정이 없어 유연하게 사용할 수 있습니다.
- **단점**:
  - 데이터가 많아지면 예측 시 모든 데이터와의 거리를 계산해야 하므로 매우 느려집니다.
  - 특성의 차원이 커질수록(고차원 데이터) 거리 계산의 의미가 모호해져 성능이 급격히 저하되는 '차원의 저주(Curse of Dimensionality)'에 취약합니다.
  - 특성 스케일링에 매우 민감하며, 최적의 K값을 찾는 것이 중요합니다.
- **대안**:
  - **로지스틱 회귀**: 선형적으로 분류가 가능한 문제에 더 효율적입니다.
  - **서포트 벡터 머신 (SVC)**: 고차원 데이터에서 KNN보다 일반적으로 더 좋은 성능을 보이며, 커널을 통해 비선형 분류도 가능합니다.
  - **랜덤 포레스트 / 그래디언트 부스팅**: 데이터의 양이나 차원 수에 관계없이 전반적으로 뛰어난 성능을 보이는 강력한 분류기입니다.