K-Nearest Neighbors (KNN)

Overview

K-Nearest Neighbors is a simple yet powerful supervised machine learning algorithm used for both classification and regression tasks. It makes predictions by finding the K closest data points in the training set to a new data point and using their values to make a prediction.

How KNN Works

1. **Distance Calculation**
   - For a new data point, calculate its distance to all points in the training set
   - Common distance metrics:
     - Euclidean distance (most common): $\sqrt{[(x_1-x_2)^2 + (y_1-y_2)^2]}$
     - Manhattan distance: $|x_1-x_2| + |y_1-y_2|$
     - Minkowski distance
     - Hamming distance (for categorical variables)
2. **Finding K Nearest Neighbors**
   - Sort distances in ascending order
   - Select the K closest points
   - For classification: Use majority voting
   - For regression: Take the mean/median of K neighbors

Implementation in Python

python
Copy

```python
import numpy as np
from collections import Counter

class KNN:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y
```

```
    def euclidean_distance(self, x1, x2):
        return np.sqrt(np.sum((x1 - x2) ** 2))

    def predict(self, X):
        predictions = [self._predict(x) for x in X]
        return np.array(predictions)

    def _predict(self, x):
        # Compute distances
        distances = [self.euclidean_distance(x, x_train)
                    for x_train in self.X_train]

        # Get k nearest samples, labels
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]

        # Majority vote
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]
```

Advantages

1. Simple to understand and implement
2. No training phase required (lazy learning)
3. Naturally handles multi-class classification
4. Can be used for both regression and classification
5. Non-parametric: makes no assumptions about data distribution

Disadvantages

1. Computationally expensive during prediction
2. Requires feature scaling
3. Curse of dimensionality
4. Sensitive to noisy data and outliers
5. Memory-intensive (stores all training data)

Best Practices

1. **Choose K Wisely**
   - Larger K: Smoother decision boundary, less sensitive to noise
   - Smaller K: More complex decision boundary, may overfit
   - Use odd K for binary classification to avoid ties
   - Common approach: K = √n, where n is the number of samples

Feature Scaling
 python
Copy
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

2. `X_scaled = scaler.fit_transform(X)`

3. **Dimensionality Reduction**
   - Use PCA or feature selection for high-dimensional data
   - Remove irrelevant features

Cross-Validation for K Selection
 python
Copy
```
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier

k_range = range(1, 31)
k_scores = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X, y, cv=5)
```

4. `    k_scores.append(scores.mean())`

Common Applications

1. Recommendation systems
2. Pattern recognition
3. Data imputation
4. Credit risk assessment
5. Medical diagnosis

Example Use Case

python
Copy

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load data
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.2, random_state=42)

# Create and train model
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Make predictions
predictions = knn.predict(X_test)

# Evaluate
accuracy = knn.score(X_test, y_test)
print(f"Accuracy: {accuracy:.2f}")
```