Implementasi Algoritma K-means dan DBSCAN Tanpa Library Machine Learning

Nugroho Adi Susanto

Mei 2025

Contents

1 Pendahuluan

Dalam dokumen ini, kami mengimplementasikan algoritma K-means dan DBSCAN dari awal tanpa menggunakan library machine learning. Implementasi ini menggunakan dataset Iris untuk demonstrasi. Tujuan utama adalah memahami mekanisme internal dari kedua algoritma clustering populer ini serta membandingkan performa dan karakteristik keduanya.

2 Loading Dataset dan Persiapan Awal

Dataset Iris adalah salah satu dataset klasik dalam pembelajaran mesin. Dataset ini berisi 150 sampel dari tiga spesies Iris (Setosa, Versicolor, dan Virginica) dengan empat fitur: panjang sepal, lebar sepal, panjang petal, dan lebar petal.

2.1 Import Library yang Diperlukan

Berikut adalah library yang digunakan:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
from matplotlib.colors import ListedColormap
import random
from collections import defaultdict

# Set random seed untuk reproducibility
np.random.seed(42)
random.seed(42)
```

2.2 Loading Dataset Iris

Pada tahap ini, kita memuat dataset Iris dan menampilkan informasi dasar tentang dataset tersebut:

```
# Load dataset Iris
iris = datasets.load_iris()
X = iris.data  # Fitur
y = iris.target  # Label sebenarnya
feature_names = iris.feature_names
target_names = iris.target_names

# Membuat DataFrame untuk memudahkan visualisasi
iris_df = pd.DataFrame(X, columns=feature_names)
iris_df['species'] = [iris.target_names[i] for i in y]

# Tampilkan informasi dataset
print(f"Dataset shape: {X.shape}")
print(f"Feature names: {feature_names}")
print(f"Target names: {target_names}")
```

3 Visualisasi Data Awal

Sebelum menerapkan algoritma clustering, penting untuk memvisualisasikan data untuk mendapatkan pemahaman intuitif tentang distribusi data.

3.1 Visualisasi dengan Label Asli

Fungsi berikut digunakan untuk memvisualisasikan dataset Iris, dengan warna menunjukkan label asli atau hasil clustering:

```
# Fungsi untuk visualisasi dataset
def plot_iris(X, y, title):
    plt.figure(figsize=(12, 5))

    # Plot untuk 2 fitur pertama (sepal length vs sepal width)
    plt.subplot(1, 2, 1)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=ListedColormap(['#FF0000', '#00FF00', '#0000FF']))
    plt.xlabel(feature_names[0])
    plt.ylabel(feature_names[1])
    plt.title(f"{title} (Sepal Features)")

    # Plot untuk 2 fitur terakhir (petal length vs petal width)
    plt.subplot(1, 2, 2)
    plt.scatter(X[:, 2], X[:, 3], c=y, cmap=ListedColormap(['#FF0000', '#00FF00', '#0000FF']))
    plt.xlabel(feature_names[2])
    plt.ylabel(feature_names[3])
    plt.title(f"{title} (Petal Features)")

    plt.tight_layout()
```

3.2 Visualisasi Data Tanpa Label

Dalam praktik clustering yang sebenarnya, kita biasanya tidak memiliki label yang benar dan bekerja dengan data tanpa label:

```
# Visualisasi data awal tanpa label (seperti yang akan kita lihat saat melakukan clustering)
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.scatter(X[:, 0], X[:, 1], c='gray')
plt.xlabel(feature_names[0])
plt.ylabel(feature_names[1])
plt.title("Iris Dataset tanpa Label (Sepal Features)")

plt.subplot(1, 2, 2)
plt.scatter(X[:, 2], X[:, 3], c='gray')
plt.xlabel(feature_names[2])
plt.ylabel(feature_names[3])
plt.title("Iris Dataset tanpa Label (Petal Features)")
```

4 Implementasi Algoritma K-means

K-means adalah algoritma clustering yang membagi data menjadi k kelompok berdasarkan jarak ke centroid terdekat. Berikut adalah implementasi K-means dari awal:

```
class KMeans:
    def __init__(self, n_clusters=3, max_iterations=300, tolerance=0.0001):
        self.n_clusters = n_clusters
        self.max_iterations = max_iterations
        self.tolerance = tolerance
        self.centroids = None
        self.labels_ = None
        self.inertia_ = None

    def _initialize_centroids(self, X):
        # Inisialisasi centroid secara acak dari data
        idx = np.random.choice(X.shape[0], self.n_clusters, replace=False)
        return X[idx, :]

    def _compute_distance(self, X, centroids):
        # Menghitung jarak euclidean antara setiap titik dengan semua centroid
        distances = np.zeros((X.shape[0], self.n_clusters))
        for k in range(self.n_clusters):
            # Jarak euclidean kuadrat untuk mempercepat perhitungan
            distances[:, k] = np.sum(np.square(X - centroids[k, :]), axis=1)
        return distances

    def _get_labels(self, distances):
        # Menentukan cluster untuk setiap data berdasarkan jarak terdekat
        return np.argmin(distances, axis=1)

    def _update_centroids(self, X, labels):
        # Update centroid berdasarkan rata-rata data di setiap cluster
        centroids = np.zeros((self.n_clusters, X.shape[1]))
        for k in range(self.n_clusters):
            if np.sum(labels == k) > 0:  # Hindari pembagian dengan nol
                centroids[k, :] = np.mean(X[labels == k, :], axis=0)
        return centroids

    def _compute_inertia(self, X, labels, centroids):
        # Menghitung inertia (jumlah kuadrat jarak data ke centroid terdekat)
        inertia = 0
        for k in range(self.n_clusters):
            if np.sum(labels == k) > 0:
                inertia += np.sum(np.square(X[labels == k, :] - centroids[k, :]))
        return inertia

    def fit(self, X):
        # Inisialisasi centroid secara acak
        self.centroids = self._initialize_centroids(X)
        prev_centroids = np.copy(self.centroids)
```

```
47
48          for _ in range(self.max_iterations):
49              # Hitung jarak dan tentukan cluster
50              distances = self._compute_distance(X, self.centroids)
51              self.labels_ = self._get_labels(distances)
52
53              # Update centroid
54              self.centroids = self._update_centroids(X, self.labels_)
55
56              # Cek konvergensi
57              if np.all(np.abs(self.centroids - prev_centroids) < self.
    tolerance):
58                  break
59
60              prev_centroids = np.copy(self.centroids)
61
62          # Hitung inertia akhir
63          self.inertia_ = self._compute_inertia(X, self.labels_, self.
    centroids)
64
65          return self
66
67      def predict(self, X):
68          # Prediksi cluster untuk data baru
69          distances = self._compute_distance(X, self.centroids)
70          return self._get_labels(distances)
```

4.1 Menjalankan K-means

Kita menjalankan K-means dengan k=3 (sesuai dengan jumlah spesies di dataset Iris):

```
1 # Implementasi K-means dengan k=3 (jumlah spesies di dataset Iris)
2 kmeans = KMeans(n_clusters=3)
3 kmeans.fit(X)
4
5 # Hasil clustering
6 y_kmeans = kmeans.labels_
```

4.2 Evaluasi Hasil K-means

Untuk mengevaluasi hasil clustering, kita menggunakan Adjusted Rand Index dan Silhouette Score:

```
1 from sklearn.metrics import adjusted_rand_score, silhouette_score
2
3 # Adjusted Rand Index - mengukur kesamaan antara dua pengelompokan
4 ari = adjusted_rand_score(y, y_kmeans)
5
6 # Silhouette Score - mengukur kualitas cluster yang terbentuk
7 silhouette = silhouette_score(X, y_kmeans)
8
9 print(f"Adjusted Rand Index: {ari:.4f}")
10 print(f"Silhouette Score: {silhouette:.4f}")
```

4.3 Elbow Method untuk K-means

Elbow Method digunakan untuk menentukan nilai k yang optimal. Metode ini melibatkan plot nilai inertia terhadap jumlah cluster:

```
# Implementasi Elbow Method untuk menentukan nilai k yang optimal
inertia_values = []
silhouette_values = []
k_values = range(2, 11)

for k in k_values:
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(X)
    inertia_values.append(kmeans.inertia_)
    silhouette = silhouette_score(X, kmeans.labels_) if k > 1 else 0
    silhouette_values.append(silhouette)
```

5 Implementasi Algoritma DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) adalah algoritma clustering berbasis densitas yang dapat menemukan cluster dengan bentuk tidak beraturan dan mengidentifikasi noise points.

```
class DBSCAN:
    def __init__(self, eps=0.5, min_samples=5):
        self.eps = eps
        self.min_samples = min_samples
        self.labels_ = None

    def _get_neighbors(self, X, point_idx):
        # Mengembalikan indeks dari semua titik yang merupakan tetangga
   dari titik yang diberikan
        distances = np.sum(np.square(X - X[point_idx, :]), axis=1)
        return np.where(distances <= self.eps**2)[0]

    def fit(self, X):
        n_samples = X.shape[0]
        # Inisialisasi label (-1 untuk noise, cluster diberi label 0, 1,
    2, ...)
        self.labels_ = np.full(n_samples, -1)

        # Cluster saat ini
        cluster = 0

        # Untuk setiap titik di dataset
        for point_idx in range(n_samples):
            # Jika titik sudah diberi label, lanjut ke titik berikutnya
            if self.labels_[point_idx] != -1:
                continue

            # Temukan tetangga dari titik saat ini
            neighbors = self._get_neighbors(X, point_idx)

            # Jika jumlah tetangga kurang dari min_samples, titik adalah
    noise
            if len(neighbors) < self.min_samples:
```

```
                self.labels_[point_idx] = -1  # Mark as noise
                continue

            # Titik adalah core point, mulai cluster baru
            self.labels_[point_idx] = cluster

            # Expand cluster: proses semua titik yang dapat dijangkau dari core point ini
            # Inisialisasi seeds dengan tetangga dari core point
            seeds = set(neighbors) - {point_idx}
            seeds = list(seeds)

            # Proses semua titik dalam seeds
            seed_idx = 0
            while seed_idx < len(seeds):
                # Ambil titik dari seeds
                current_point = seeds[seed_idx]

                # Jika titik adalah noise, tambahkan ke cluster saat ini
                if self.labels_[current_point] == -1:
                    self.labels_[current_point] = cluster

                # Jika titik belum di-assign ke cluster apapun
                if self.labels_[current_point] == -1:
                    # Tandai titik sebagai bagian dari cluster saat ini
                    self.labels_[current_point] = cluster

                    # Temukan tetangga dari titik saat ini
                    current_neighbors = self._get_neighbors(X, current_point)

                    # Jika titik adalah core point, tambahkan tetangga ke seeds
                    if len(current_neighbors) >= self.min_samples:
                        # Tambahkan tetangga yang belum diproses ke seeds
                        for neighbor in current_neighbors:
                            if self.labels_[neighbor] == -1 or self.labels_[neighbor] == -1:
                                if neighbor not in seeds:
                                    seeds.append(neighbor)

                seed_idx += 1

            # Lanjutkan dengan cluster selanjutnya
            cluster += 1

        return self
```

5.1 Menjalankan DBSCAN

Kita menjalankan DBSCAN dengan parameter awal:

```
# Menjalankan DBSCAN dengan parameter awal
dbscan = DBSCAN(eps=0.8, min_samples=5)
dbscan.fit(X)

```

```
# Hasil clustering dengan DBSCAN
y_dbscan = dbscan.labels_
```

5.2 Tuning Parameter DBSCAN

DBSCAN memerlukan dua parameter utama: eps (radius tetangga) dan min_samples (jumlah minimum tetangga). Kita perlu melakukan tuning untuk menemukan nilai optimal dari kedua parameter tersebut:

```
# Fungsi untuk menghitung jarak rata-rata ke k-tetangga terdekat
def compute_knn_distances(X, k):
    # Menghitung jarak antara semua pasangan titik
    distances = np.zeros((X.shape[0], X.shape[0]))
    for i in range(X.shape[0]):
        for j in range(X.shape[0]):
            distances[i, j] = np.sqrt(np.sum(np.square(X[i] - X[j])))

    # Untuk setiap titik, ambil jarak k-tetangga terdekat (kecuali dirinya sendiri)
    knn_distances = []
    for i in range(X.shape[0]):
        sorted_distances = np.sort(distances[i])
        knn_distances.append(sorted_distances[k])

    return np.array(knn_distances)

# Menghitung jarak rata-rata ke 5-tetangga terdekat
knn_distances = compute_knn_distances(X, 5)
```

5.3 Grid Search untuk Parameter DBSCAN

Grid search digunakan untuk secara sistematis mencoba berbagai kombinasi parameter dan memilih yang terbaik berdasarkan metrik evaluasi:

```
# Grid search untuk parameter DBSCAN
eps_values = np.linspace(0.2, 1.5, 14)  # Range dari 0.2 hingga 1.5 dengan 14 nilai
min_samples_values = [3, 5, 7, 10, 15]  # Nilai minPts yang akan dicoba

results = []

for eps in eps_values:
    for min_samples in min_samples_values:
        dbscan = DBSCAN(eps=eps, min_samples=min_samples)
        dbscan.fit(X)

        labels = dbscan.labels_
        n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
        n_noise = list(labels).count(-1)

        if n_clusters > 1:  # Hanya hitung skor jika ada lebih dari satu cluster
            try:
                silhouette = silhouette_score(X, labels)
            except:
                silhouette = float('nan')
```

```
            ari = adjusted_rand_score(y, labels)
        else:
            silhouette = float('nan')
            ari = float('nan')

        results.append({
            'eps': eps,
            'min_samples': min_samples,
            'n_clusters': n_clusters,
            'n_noise': n_noise,
            'silhouette': silhouette,
            'ari': ari
        })
```

5.4 Parameter Optimal DBSCAN

Setelah melakukan grid search, kita dapat menentukan parameter optimal untuk DBSCAN:

```
# Konversi hasil ke DataFrame
results_df = pd.DataFrame(results)

# Filter hasil dengan jumlah cluster > 1 dan tanpa nilai nan
valid_results = results_df[(results_df['n_clusters'] > 1) & (~results_df
    ['silhouette'].isna())].copy()

if not valid_results.empty:
    # Temukan parameter terbaik berdasarkan silhouette score
    best_silhouette_idx = valid_results['silhouette'].idxmax()
    best_params_silhouette = valid_results.loc[best_silhouette_idx]

    # Temukan parameter terbaik berdasarkan ARI
    best_ari_idx = valid_results['ari'].idxmax()
    best_params_ari = valid_results.loc[best_ari_idx]
```

5.5 Menjalankan DBSCAN dengan Parameter Optimal

Kita menjalankan DBSCAN dengan parameter optimal yang ditemukan dari grid search:

```
# Menggunakan parameter terbaik dari grid search (berdasarkan silhouette
    score)
if 'best_params_silhouette' in locals():
    optimal_eps = best_params_silhouette['eps']
    optimal_min_samples = int(best_params_silhouette['min_samples'])
else:  # Jika tidak ada hasil yang valid, gunakan default
    optimal_eps = 0.6
    optimal_min_samples = 5

# Jalankan DBSCAN dengan parameter optimal
optimal_dbscan = DBSCAN(eps=optimal_eps, min_samples=optimal_min_samples
    )
optimal_dbscan.fit(X)
y_dbscan_optimal = optimal_dbscan.labels_
```

6 Perbandingan K-means dan DBSCAN

Pada bagian ini, kita membandingkan hasil clustering dari K-means dan DBSCAN:

```
# Visualisasi perbandingan hasil clustering K-means dan DBSCAN
plt.figure(figsize=(15, 10))

# Visualisasi label asli
plt.subplot(3, 2, 1)
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=ListedColormap(['#FF0000', '#00FF00', '#0000FF']))
plt.xlabel(feature_names[0])
plt.ylabel(feature_names[1])
plt.title("Label Asli (Sepal Features)")

# Visualisasi hasil K-means
plt.subplot(3, 2, 3)
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap=ListedColormap(['#FF0000', '#00FF00', '#0000FF']))
plt.xlabel(feature_names[0])
plt.ylabel(feature_names[1])
plt.title("Hasil K-means (Sepal Features)")

# Visualisasi hasil DBSCAN optimal
cmap_dbscan = ListedColormap(['#000000', '#FF0000', '#00FF00', '#0000FF', '#FFFF00', '#00FFFF', '#FF00FF'])
plt.subplot(3, 2, 5)
plt.scatter(X[:, 0], X[:, 1], c=y_dbscan_optimal, cmap=cmap_dbscan)
plt.xlabel(feature_names[0])
plt.ylabel(feature_names[1])
plt.title(f"Hasil DBSCAN (eps={optimal_eps:.2f}, minPts={optimal_min_samples}) (Sepal Features)")
```

7 Kesimpulan

Pada implementasi clustering dengan dataset Iris, kita telah melakukan perbandingan antara algoritma K-means dan DBSCAN dengan pengembangan dari awal (tanpa library machine learning).

7.1 Hasil K-means

- K-means berhasil dengan baik mengelompokkan dataset Iris menjadi 3 cluster, sesuai dengan jumlah spesies asli.
- Dengan Elbow Method, kita dapat mengonfirmasi bahwa k=3 memang merupakan nilai optimal untuk dataset ini.

7.2 Hasil DBSCAN

- DBSCAN memerlukan tuning parameter eps dan min_samples untuk mendapatkan hasil yang optimal.
- Setelah melakukan grid search, parameter terbaik ditemukan.

- DBSCAN memiliki keunggulan mampu mengidentifikasi noise points dan mendeteksi cluster dengan bentuk tidak beraturan.

7.3 Perbandingan

- K-means cenderung bekerja lebih baik pada dataset Iris karena cluster-nya bersifat konveks dan terpisah dengan baik.
- DBSCAN dapat menemukan cluster dengan bentuk yang lebih kompleks, tetapi memerlukan lebih banyak upaya dalam penentuan parameter.
- Evaluasi menggunakan Adjusted Rand Index dan Silhouette Score menunjukkan seberapa baik hasil clustering dibandingkan dengan label aslinya.

8 Referensi

1. Jain, A. K. (2010). Data clustering: 50 years beyond K-means. Pattern Recognition Letters, 31(8), 651-666.
2. Ester, M., Kriegel, H. P., Sander, J., & Xu, X. (1996, August). A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96) (Vol. 96, No. 34, pp. 226-231).
3. Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.