Guía de Laboratorio - Sesión 16: Clustering Avanzado

Inteligencia Artificial (100000S14F)
Ingeniería de Software

Ciclo 2 Agosto 2025

Objetivos del Laboratorio

Al finalizar esta sesión, los estudiantes serán capaces de:

- Implementar y comparar múltiples algoritmos de clustering
- Evaluar la calidad de clusters usando diferentes métricas
- Seleccionar el algoritmo apropiado según el tipo de datos
- Interpretar y visualizar resultados de clustering
- Aplicar técnicas de clustering a problemas del mundo real

1. Configuración Inicial

```
# Importaciones esenciales
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import warnings
warnings.filterwarnings('ignore')

# Configuracion de visualizacion
plt.style.use('seaborn-v0_8')
sns.set_palette("husl")
plt.rcParams['figure.figsize'] = (12, 8)
```

2. Parte 1: Fundamentos de Clustering

2.1. ¿Por qué usar clustering?

El clustering nos ayuda a:

- Descubrir patrones ocultos en los datos
- Segmentar clientes para marketing
- Detectar anomalías y outliers
- Reducir la complejidad de datos
- Preparar datos para otros algoritmos

3. Parte 2: Implementación de Algoritmos

3.1. K-Means Clustering

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, calinski_harabasz_score,
    davies_bouldin_score
from sklearn.datasets import make_blobs, make_moons, make_circles

def demostrar_kmeans():
    # Generar datos de ejemplo
    X, y_true = make_blobs(n_samples=300, centers=4, cluster_std=0.60,
        random_state=42)

    # Aplicar K-Means
    kmeans = KMeans(n_clusters=4, random_state=42, n_init=10)
    y_pred = kmeans.fit_predict(X)

    # Calcular metricas
    silhouette = silhouette_score(X, y_pred)
    calinski = calinski_harabasz_score(X, y_pred)
    davies = davies_bouldin_score(X, y_pred)

    # Visualizar resultados
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

    # Datos originales
    ax1.scatter(X[:, 0], X[:, 1], c=y_true, cmap='viridis', s=50, alpha
        =0.8)
    ax1.set_title('Datos Originales con Etiquetas Verdaderas')
    ax1.set_xlabel('Caracteristica 1')
    ax1.set_ylabel('Caracteristica 2')

    # Resultados K-Means
    ax2.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='viridis', s=50, alpha
        =0.8)
    ax2.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_
        [:, 1],
                marker='x', s=200, linewidths=3, color='red', label='
                    Centroides')
    ax2.set_title(f'K-Means Clustering\nSilhouette: {silhouette:.3f},
        Calinski: {calinski:.1f}')
    ax2.set_xlabel('Caracteristica 1')
    ax2.set_ylabel('Caracteristica 2')
    ax2.legend()
```

```
    plt.tight_layout()
    plt.show()

    print(f"Metricas de evaluacion:")
    print(f"Silhouette Score: {silhouette:.3f} (mejor cerca de 1)")
    print(f"Calinski-Harabasz: {calinski:.1f} (mayor es mejor)")
    print(f"Davies-Bouldin: {davies:.3f} (menor es mejor)")

    return kmeans, X, y_pred

# Ejecutar demostracion
kmeans_model, X_blobs, y_kmeans = demostrar_kmeans()
```

3.2. DBSCAN Clustering

```
from sklearn.cluster import DBSCAN

def demostrar_dbscan():
    # Generar datos no lineales
    X, y_true = make_moons(n_samples=300, noise=0.05, random_state=42)

    # Probar diferentes parametros de DBSCAN
    fig, axes = plt.subplots(2, 3, figsize=(18, 12))
    eps_values = [0.1, 0.2, 0.3]
    min_samples_values = [5, 10, 15]

    best_silhouette = -1
    best_dbscan = None

    for i, eps in enumerate(eps_values):
        for j, min_samples in enumerate(min_samples_values):
            dbscan = DBSCAN(eps=eps, min_samples=min_samples)
            y_pred = dbscan.fit_predict(X)

            # Calcular metricas si hay mas de un cluster
            n_clusters = len(set(y_pred)) - (1 if -1 in y_pred else 0)
            if n_clusters > 1:
                silhouette = silhouette_score(X, y_pred)
            else:
                silhouette = -1

            # Actualizar mejor modelo
            if silhouette > best_silhouette and n_clusters > 1:
                best_silhouette = silhouette
                best_dbscan = dbscan
                best_y_pred = y_pred

            # Visualizar
            ax = axes[i, j]
            scatter = ax.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='
               viridis', s=50, alpha=0.8)
            ax.set_title(f'DBSCAN: eps={eps}, min_samples={min_samples
               }\nClusters: {n_clusters}, Silhouette: {silhouette:.3f}'
               )
            ax.set_xlabel('Caracteristica 1')
            ax.set_ylabel('Caracteristica 2')
```

```
    plt.tight_layout()
    plt.show()

    # Analisis del mejor modelo
    n_clusters = len(set(best_y_pred)) - (1 if -1 in best_y_pred else
        0)
    n_noise = list(best_y_pred).count(-1)

    print(f"Mejor configuracion DBSCAN:")
    print(f"eps: {best_dbscan.eps}, min_samples: {best_dbscan.
        min_samples}")
    print(f"Numero de clusters encontrados: {n_clusters}")
    print(f"Puntos considerados ruido: {n_noise} ({n_noise/len(X)
        *100:.1f}%)")
    print(f"Silhouette Score: {best_silhouette:.3f}")

    return best_dbscan, X, best_y_pred

# Ejecutar demostracion DBSCAN
dbscan_model, X_moons, y_dbscan = demostrar_dbscan()
```

3.3. Agglomerative Clustering

```
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, linkage

def demostrar_agglomerative():
    # Generar datos
    X, y_true = make_blobs(n_samples=150, centers=3, cluster_std=0.8,
        random_state=42)

    # Crear dendrograma
    plt.figure(figsize=(15, 5))

    # Subplot 1: Dendrograma
    plt.subplot(1, 2, 1)
    linked = linkage(X, 'ward')
    dendrogram(linked, orientation='top', distance_sort='descending',
        show_leaf_counts=True)
    plt.title('Dendrograma - Agglomerative Clustering')
    plt.xlabel('Indice del Punto')
    plt.ylabel('Distancia')

    # Subplot 2: Resultados del clustering
    plt.subplot(1, 2, 2)

    # Probar diferentes numeros de clusters
    n_clusters_list = [2, 3, 4]
    colors = ['red', 'blue', 'green', 'purple']

    for idx, n_clusters in enumerate(n_clusters_list):
        agglo = AgglomerativeClustering(n_clusters=n_clusters)
        y_pred = agglo.fit_predict(X)

        silhouette = silhouette_score(X, y_pred)
```

```
        plt.scatter(X[y_pred == 0, 0], X[y_pred == 0, 1],
                    color=colors[idx], alpha=0.6, s=50, label=f'{
                        n_clusters} clusters (S: {silhouette:.3f})')
        # Para multiples clusters, necesitamos scatter separados
        for cluster in range(1, n_clusters):
            plt.scatter(X[y_pred == cluster, 0], X[y_pred == cluster,
                1],
                        color=colors[idx], alpha=0.6, s=50)

    plt.title('Agglomerative Clustering con Diferentes K')
    plt.xlabel('Caracteristica 1')
    plt.ylabel('Caracteristica 2')
    plt.legend()

    plt.tight_layout()
    plt.show()

    # Analisis con el numero optimo de clusters (3 en este caso)
    agglo_optimal = AgglomerativeClustering(n_clusters=3)
    y_pred_optimal = agglo_optimal.fit_predict(X)

    silhouette = silhouette_score(X, y_pred_optimal)
    calinski = calinski_harabasz_score(X, y_pred_optimal)

    print(f"Agglomerative Clustering con 3 clusters:")
    print(f"Silhouette Score: {silhouette:.3f}")
    print(f"Calinski-Harabasz Score: {calinski:.1f}")

    return agglo_optimal, X, y_pred_optimal

# Ejecutar demostracion Agglomerative
agglo_model, X_agglo, y_agglo = demostrar_agglomerative()
```

4. Parte 3: Métricas de Evaluación Completas

```
def evaluar_clustering_completo(X, y_pred, y_true=None, algoritmo=""):
    """Evalua exhaustivamente un resultado de clustering"""

    print(f"\n{'='*50}")
    print(f"EVALUACION COMPLETA: {algoritmo}")
    print(f"{'='*50}")

    # Metricas internas (no requieren etiquetas verdaderas)
    silhouette = silhouette_score(X, y_pred)
    calinski = calinski_harabasz_score(X, y_pred)
    davies = davies_bouldin_score(X, y_pred)

    # Informacion basica
    n_clusters = len(set(y_pred)) - (1 if -1 in y_pred else 0)
    n_noise = list(y_pred).count(-1)

    print(f"INFORMACION BASICA:")
    print(f"- Numero de clusters: {n_clusters}")
    print(f"- Puntos considerados ruido: {n_noise}")
```

```
    print(f"- Tamanio promedio del cluster: {len(X)/n_clusters:.1f}
        puntos")

    print(f"\nMETRICAS INTERNAS:")
    print(f"- Silhouette Score: {silhouette:.3f}")
    print(f"  * >0.7: Fuerte estructura de clusters")
    print(f"  * 0.5-0.7: Estructura razonable")
    print(f"  * 0.25-0.5: Estructura debil")
    print(f"  * <0.25: Sin estructura significativa")

    print(f"- Calinski-Harabasz Score: {calinski:.1f}")
    print(f"  * Mayor valor indica clusters mejor definidos")

    print(f"- Davies-Bouldin Score: {davies:.3f}")
    print(f"  * Menor valor indica mejor separacion entre clusters")

    # Metricas externas (si tenemos etiquetas verdaderas)
    if y_true is not None:
        from sklearn.metrics import adjusted_rand_score,
            normalized_mutual_info_score

        ari = adjusted_rand_score(y_true, y_pred)
        nmi = normalized_mutual_info_score(y_true, y_pred)

        print(f"\nMETRICAS EXTERNAS (vs etiquetas verdaderas):")
        print(f"- Adjusted Rand Index: {ari:.3f}")
        print(f"  * 1.0: Coincidencia perfecta")
        print(f"  * 0.0: Agrupamiento aleatorio")

        print(f"- Normalized Mutual Information: {nmi:.3f}")
        print(f"  * 1.0: Informacion mutua perfecta")
        print(f"  * 0.0: Sin informacion compartida")

    # Analisis de distribucion de clusters
    unique, counts = np.unique(y_pred, return_counts=True)
    print(f"\nDISTRIBUCION DE PUNTOS POR CLUSTER:")
    for cluster, count in zip(unique, counts):
        if cluster == -1:
            print(f"  Ruido: {count} puntos ({count/len(X)*100:.1f}%)")
        else:
            print(f"  Cluster {cluster}: {count} puntos ({count/len(X)
                *100:.1f}%)")

    return {
        'silhouette': silhouette,
        'calinski': calinski,
        'davies': davies,
        'n_clusters': n_clusters,
        'n_noise': n_noise
    }

# Evaluar todos los algoritmos anteriores
print("EVALUACION COMPARATIVA DE ALGORITMOS")
print("="*60)

# K-Means
eval_kmeans = evaluar_clustering_completo(X_blobs, y_kmeans,
```

```
                                                y_true=None, algoritmo="K-
                                                    Means")

# DBSCAN
eval_dbscan = evaluar_clustering_completo(X_moons, y_dbscan,
                                                y_true=None, algoritmo="DBSCAN
                                                    ")

# Agglomerative
eval_agglo = evaluar_clustering_completo(X_agglo, y_agglo,
                                               y_true=None, algoritmo="
                                                   Agglomerative")
```

5. Parte 4: Determinación del Número Óptimo de Clusters

```
def analizar_numero_optimo_clusters(X, max_k=10):
    """Analiza el numero optimo de clusters usando multiples metodos"""

    print("ANALISIS DEL NUMERO OPTIMO DE CLUSTERS")
    print("="*50)

    # Metricas a calcular
    wcss = []  # Within-cluster sum of squares
    silhouette_scores = []
    calinski_scores = []
    davies_scores = []

    for k in range(2, max_k + 1):
        kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
        y_pred = kmeans.fit_predict(X)

        wcss.append(kmeans.inertia_)
        silhouette_scores.append(silhouette_score(X, y_pred))
        calinski_scores.append(calinski_harabasz_score(X, y_pred))
        davies_scores.append(davies_bouldin_score(X, y_pred))

    # Crear visualizaciones
    fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(15, 10)
        )

    # Metodo del codo
    ax1.plot(range(2, max_k + 1), wcss, 'bo-')
    ax1.set_title('Metodo del Codo')
    ax1.set_xlabel('Numero de Clusters (K)')
    ax1.set_ylabel('WCSS (Within-Cluster Sum of Squares)')
    ax1.grid(True, alpha=0.3)

    # Silhouette Score
    ax2.plot(range(2, max_k + 1), silhouette_scores, 'ro-')
    ax2.set_title('Silhouette Score')
    ax2.set_xlabel('Numero de Clusters (K)')
    ax2.set_ylabel('Silhouette Score')
    ax2.grid(True, alpha=0.3)
```

```
    # Calinski-Harabasz
    ax3.plot(range(2, max_k + 1), calinski_scores, 'go-')
    ax3.set_title('Calinski-Harabasz Score')
    ax3.set_xlabel('Numero de Clusters (K)')
    ax3.set_ylabel('Calinski-Harabasz Score')
    ax3.grid(True, alpha=0.3)

    # Davies-Bouldin
    ax4.plot(range(2, max_k + 1), davies_scores, 'mo-')
    ax4.set_title('Davies-Bouldin Score')
    ax4.set_xlabel('Numero de Clusters (K)')
    ax4.set_ylabel('Davies-Bouldin Score')
    ax4.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

    # Recomendacion basada en metricas
    optimal_silhouette = np.argmax(silhouette_scores) + 2
    optimal_calinski = np.argmax(calinski_scores) + 2
    optimal_davies = np.argmin(davies_scores) + 2

    print(f"\nRECOMENDACIONES:")
    print(f"- Segun Silhouette Score: K = {optimal_silhouette}")
    print(f"- Segun Calinski-Harabasz: K = {optimal_calinski}")
    print(f"- Segun Davies-Bouldin: K = {optimal_davies}")

    # Consenso
    recommendations = [optimal_silhouette, optimal_calinski,
        optimal_davies]
    optimal_k = max(set(recommendations), key=recommendations.count)

    print(f"\nRECOMENDACION FINAL: K = {optimal_k}")

    return optimal_k

# Ejecutar analisis
k_optimo = analizar_numero_optimo_clusters(X_blobs)
```

6. Parte 5: Caso Práctico - Segmentación de Clientes

```
def caso_practico_segmentacion_clientes():
    """Caso practico completo de segmentacion de clientes"""

    print("CASO PRACTICO: SEGMENTACION DE CLIENTES")
    print("="*50)

    # Crear dataset simulado de clientes
    np.random.seed(42)
    n_clientes = 200

    data = {
        'edad': np.random.normal(45, 15, n_clientes).astype(int),
        'ingreso_anual': np.random.normal(50000, 20000, n_clientes),
        'gasto_mensual': np.random.normal(800, 300, n_clientes),
```

```
    'frecuencia_compra': np.random.poisson(8, n_clientes),
    'puntuacion_satisfaccion': np.random.uniform(1, 10, n_clientes)
}

# Ajustar relaciones realistas
data['gasto_mensual'] = data['ingreso_anual'] * 0.01 + np.random.
    normal(0, 100, n_clientes)
data['frecuencia_compra'] = (data['gasto_mensual'] / 100 + np.
    random.poisson(3, n_clientes)).astype(int)

df_clientes = pd.DataFrame(data)
df_clientes['edad'] = np.clip(df_clientes['edad'], 18, 80)
df_clientes['ingreso_anual'] = np.clip(df_clientes['ingreso_anual'
    ], 10000, 150000)
df_clientes['gasto_mensual'] = np.clip(df_clientes['gasto_mensual'
    ], 100, 2000)

print("Dataset de clientes creado:")
print(df_clientes.describe())

# Preprocesamiento
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df_clientes)

# Determinar numero optimo de clusters
k_optimo = analizar_numero_optimo_clusters(X_scaled, max_k=8)

# Aplicar K-Means con K optimo
kmeans_clientes = KMeans(n_clusters=k_optimo, random_state=42,
    n_init=10)
df_clientes['segmento'] = kmeans_clientes.fit_predict(X_scaled)

# Visualizar segmentos (usando PCA para reduccion dimensional)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

plt.figure(figsize=(15, 5))

# Subplot 1: Segmentos en espacio PCA
plt.subplot(1, 2, 1)
scatter = plt.scatter(X_pca[:, 0], X_pca[:, 1], c=df_clientes['
    segmento'],
                      cmap='viridis', s=50, alpha=0.7)
plt.xlabel('Componente Principal 1')
plt.ylabel('Componente Principal 2')
plt.title(f'Segmentacion de Clientes ({k_optimo} Segmentos)')
plt.colorbar(scatter, label='Segmento')

# Subplot 2: Caracteristicas de segmentos
plt.subplot(1, 2, 2)
segmentos_agg = df_clientes.groupby('segmento').mean()
segmentos_agg.plot(kind='bar', ax=plt.gca())
plt.title('Caracteristicas Promedio por Segmento')
plt.xlabel('Segmento')
plt.ylabel('Valor Promedio')
plt.xticks(rotation=45)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

```
plt.tight_layout()
plt.show()

# Analisis detallado de segmentos
print("\nANALISIS DETALLADO DE SEGMENTOS:")
print("="*40)

for segmento in range(k_optimo):
    segmento_data = df_clientes[df_clientes['segmento'] == segmento
        ]
    print(f"\nSEGMENTO {segmento} ({len(segmento_data)} clientes):"
        )
    print(f"- Edad promedio: {segmento_data['edad'].mean():.1f}
        anios")
    print(f"- Ingreso anual promedio: ${segmento_data['
        ingreso_anual'].mean():.0f}")
    print(f"- Gasto mensual promedio: ${segmento_data['
        gasto_mensual'].mean():.0f}")
    print(f"- Frecuencia de compra: {segmento_data['
        frecuencia_compra'].mean():.1f} compras/mes")
    print(f"- Satisfaccion: {segmento_data['puntuacion_satisfaccion
        '].mean():.1f}/10")

# Recomendaciones de marketing
print(f"\nRECOMENDACIONES DE MARKETING:")
print("="*40)

# Identificar segmentos valiosos
segmento_valor = df_clientes.groupby('segmento')['gasto_mensual'].
    mean().idxmax()
segmento_frecuente = df_clientes.groupby('segmento')['
    frecuencia_compra'].mean().idxmax()
segmento_satisfecho = df_clientes.groupby('segmento')['
    puntuacion_satisfaccion'].mean().idxmax()

print(f"- Segmento mas valioso (mayor gasto): Segmento {
    segmento_valor}")
print(f"- Segmento mas frecuente: Segmento {segmento_frecuente}")
print(f"- Segmento mas satisfecho: Segmento {segmento_satisfecho}")

# Estrategias especificas
print(f"\nESTRATEGIAS RECOMENDADAS:")
for segmento in range(k_optimo):
    seg_data = df_clientes[df_clientes['segmento'] == segmento]
    gasto_promedio = seg_data['gasto_mensual'].mean()
    frecuencia_promedio = seg_data['frecuencia_compra'].mean()

    print(f"\nSegmento {segmento}:")
    if gasto_promedio > df_clientes['gasto_mensual'].mean():
        print(f"  * Estrategia: Programas de fidelizacion premium")
        print(f"  * Accion: Ofrecer membresia VIP con beneficios
            exclusivos")
    elif frecuencia_promedio > df_clientes['frecuencia_compra'].
        mean():
        print(f"  * Estrategia: Programas de frecuencia")
        print(f"  * Accion: Sistema de puntos y recompensas por
            compras frecuentes")
    else:
```

```
            print(f"  * Estrategia: Reactivacion")
            print(f"  * Accion: Campanas de email marketing con ofertas
                especiales")

    return df_clientes, kmeans_clientes

# Ejecutar caso practico
df_clientes_final, modelo_clientes =
   caso_practico_segmentacion_clientes()
```

7. Parte 6: Comparación Final de Algoritmos

```
def comparacion_final_algoritmos():
    """Comparacion exhaustiva de todos los algoritmos en diferentes
       tipos de datos"""

    print("COMPARACION FINAL DE ALGORITMOS DE CLUSTERING")
    print("="*60)

    # Generar diferentes tipos de datos
    datasets = {
        'Blobs (esferico)': make_blobs(n_samples=300, centers=4,
           random_state=42),
        'Moons (no lineal)': make_moons(n_samples=300, noise=0.05,
           random_state=42),
        'Circulos (anillos)': make_circles(n_samples=300, noise=0.05,
           factor=0.5, random_state=42),
        'Varianza variable': make_blobs(n_samples=300, centers=4,
           cluster_std=[1.0, 2.5, 0.5, 1.5], random_state=42)
    }

    resultados_comparativos = []

    for nombre_dataset, (X, y_true) in datasets.items():
        print(f"\n{'='*40}")
        print(f"DATASET: {nombre_dataset}")
        print(f"{'='*40}")

        # Probar diferentes algoritmos
        algoritmos = {
            'K-Means': KMeans(n_clusters=4, random_state=42, n_init=10)
                ,
            'DBSCAN': DBSCAN(eps=0.3, min_samples=5),
            'Agglomerative': AgglomerativeClustering(n_clusters=4)
        }

        fig, axes = plt.subplots(1, len(algoritmos) + 1, figsize=(20,
           4))

        # Datos originales
        axes[0].scatter(X[:, 0], X[:, 1], c=y_true, cmap='viridis', s
           =50, alpha=0.8)
        axes[0].set_title(f'{nombre_dataset}\nDatos Originales')
        axes[0].set_xlabel('Caracteristica 1')
        axes[0].set_ylabel('Caracteristica 2')
```

```
        for idx, (nombre_algo, algoritmo) in enumerate(algoritmos.items
            (), 1):
            try:
                y_pred = algoritmo.fit_predict(X)

                # Calcular metricas
                if len(set(y_pred)) > 1:
                    silhouette = silhouette_score(X, y_pred)
                    calinski = calinski_harabasz_score(X, y_pred)
                else:
                    silhouette = -1
                    calinski = 0

                # Visualizar
                axes[idx].scatter(X[:, 0], X[:, 1], c=y_pred, cmap='
                    viridis', s=50, alpha=0.8)
                axes[idx].set_title(f'{nombre_algo}\nSilhouette: {
                    silhouette:.3f}')
                axes[idx].set_xlabel('Caracteristica 1')
                axes[idx].set_ylabel('Caracteristica 2')

                # Guardar resultados
                resultados_comparativos.append({
                    'Dataset': nombre_dataset,
                    'Algoritmo': nombre_algo,
                    'Silhouette': silhouette,
                    'Calinski': calinski,
                    'N_Clusters': len(set(y_pred)) - (1 if -1 in y_pred
                        else 0)
                })

            except Exception as e:
                print(f"Error con {nombre_algo} en {nombre_dataset}: {e
                    }")
                axes[idx].set_title(f'{nombre_algo}\nError')

        plt.tight_layout()
        plt.show()

    # Crear resumen comparativo
    df_comparativo = pd.DataFrame(resultados_comparativos)

    print("\n" + "="*60)
    print("RESUMEN COMPARATIVO FINAL")
    print("="*60)

    # Mejor algoritmo por dataset segun Silhouette
    mejor_por_dataset = df_comparativo.loc[df_comparativo.groupby('
        Dataset')['Silhouette'].idxmax()]
    print("\nMEJOR ALGORITMO POR DATASET (segun Silhouette):")
    print(mejor_por_dataset[['Dataset', 'Algoritmo', 'Silhouette', '
        N_Clusters']])

    # Estadisticas generales
    print(f"\nESTADISTICAS GENERALES:")
    stats = df_comparativo.groupby('Algoritmo')['Silhouette'].agg(['
        mean', 'std', 'max'])
```

```
    print(stats.round(3))

    # Recomendaciones generales
    print(f"\nRECOMENDACIONES GENERALES:")
    print("K-Means: Ideal para datos esfericos con varianza similar")
    print("DBSCAN: Mejor para datos no lineales y deteccion de outliers
        ")
    print("Agglomerative: Buen balance, permite analisis jerarquico")

    return df_comparativo

# Ejecutar comparacion final
df_comparativo_final = comparacion_final_algoritmos()
```

8. Conclusión y Recomendaciones Finales

8.1. Resumen de Aprendizajes

- **K-Means**: Excelente para clusters esféricos y cuando se conoce K
- **DBSCAN**: Ideal para datos con ruido y clusters de forma arbitraria
- **Agglomerative**: Proporciona jerarquía y es robusto con diferentes distribuciones
- **Silhouette Score**: Mejor métrica general para evaluación interna
- **Método del codo**: Útil para determinar K en K-Means

8.2. Guía de Selección de Algoritmos

Escenario	**Algoritmo Recomendado**
Datos esféricos, K conocido	K-Means
Datos con ruido/outliers	DBSCAN
Formas de cluster complejas	DBSCAN o Agglomerative
Análisis jerárquico necesario	Agglomerative
Datos de alta dimensionalidad	K-Means con PCA
Velocidad requerida	K-Means
Interpretabilidad	Agglomerative (dendrograma)

Cuadro 1: Guía de selección de algoritmos de clustering