Practica 1: Código Huffman y Primer Teorema de Shannon

Pablo Jiménez Poyatos : U1

6 de marzo de 2024

1. Introducción

En el ámbito del análisis de datos, dos técnicas cruciales son el clustering y el diagrama de Voronói. Utilizando datos de la población "Villa Laminera", aplicaremos estas técnicas con los algoritmos KMeans y DBSCAN para agrupar a los individuos en clusters óptimos. Evaluaremos la calidad de los clusters utilizando el coeficiente de Silhouette y compararemos los resultados obtenidos con diferentes métricas y algoritmos.

2. Material usado

En esta investigación, he utilizado las plantillas proporcionadas en el campus virtual, **"GCOM2024- practica2_plantilla1.py"** y **GCOM2024-practica2_plantilla2.py"** como punto de partida para desarrollar mi código en Python. El archivo de datos, **"Personas_de_villa_laminera.txt"**, proporciona información crucial sobre las características de los individuos, con variables como el nivel de estrés y la afinidad por los dulces.

Para la manipulación y análisis de los datos, se importaron varias bibliotecas de Python. Esto incluyó **numpy** para operaciones con matrices y arrays, **matplotlib.pyplot** para la creación de gráficos y visualizaciones, y por último, las funciones y clases específicas de **scikit-learn** para implementar los algoritmos de clustering, como KMeans y DBSCAN, así como métricas de evaluación como el coeficiente de Silhouette. Además, se utilizó la biblioteca **scipy.spatial** para trabajar con el diagrama de Voronoi.

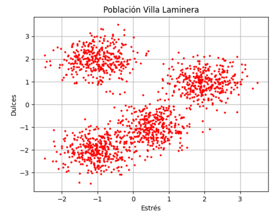


Figura 1: Grafico dispersion datos

El código **comienza** con el pre-procesamiento de los datos, donde se cargan y se muestran en un grafico de dispersión (Figura 1). En el **primer apartado**, se utiliza el algoritmo KMeans para determinar el número óptimo de clusters en el conjunto de datos. Se calcula el coeficiente de Silhouette para diferentes números de clusters (k) y se grafican en función de k para visualizar cómo varía la calidad de la agrupación. Luego, se identifica el valor máximo del coeficiente de Silhouette y se determina el número óptimo de clusters correspondiente. Con esta información, se entrena un modelo KMeans y se clasifican los datos utilizando un diagrama de Voronoi.

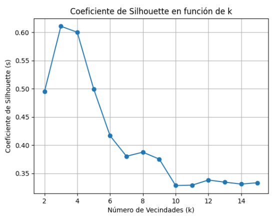


Figura 2: Relación del coeficiente de Silhouette en funcion de k

En el **segundo apartado**, se utiliza el algoritmo DBSCAN para estimar el número óptimo de clusters de manera automática. Se define un rango de valores para el parámetro eps y se establece un valor fijo para el número mínimo de puntos requeridos para formar un cluster. Luego, se itera sobre diferentes métricas de distancia y se calcula el coeficiente de Silhouette para cada combinación de parámetros. Finalmente, se grafican los resultados y se comparan con los del apartado anterior.

En el **tercer apartado**, se realizan predicciones utilizando el modelo KMeans entrenado en el primer apartado sobre dos puntos específicos en el espacio de características. Se utiliza el método predict del modelo KMeans para asignar cada punto al cluster más cercano según la estructura definida por el modelo.

3. Resultados

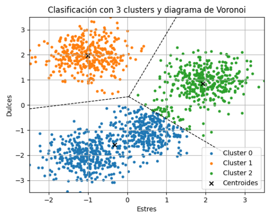


Figura 3: Diagrama de Voronoi usando KMEANS

Para el sistema A, se obtuvo el coeficiente de Silhouette (s) utilizando el algoritmo KMeans para diferentes números de vecindades k en el rango [2,15]. La Figura 2 muestra el valor de s en función de k, lo que permite determinar el número óptimo de vecindades. Basándonos en esta gráfica, se selecciona el valor de k que maximiza el coeficiente de Silhouette. En cuyo caso es $k = 3$ con coeficiente de Silhouette $s = 0{,}6108$.

A continuación, en la Figura 3, se muestra el diagrama de Voronoi que representa las regiones de influencia de cada cluster en el espacio de características. Esta visualización proporciona una representación clara de cómo los datos se agrupan en clusters y cómo se distribuyenlas regiones de influencia en función de los centroides.

En el caso del algoritmo DBSCAN, se calculó el coeficiente de Silhouette para el mismo sistema A utilizando las métricas éuclidean'y 'manhattan'. En la Figura 4, se muestra el valor de s en función del umbral de distancia (eps) para ambas métricas. En él podemos observar que el mayor coeficiente de Silhouette para DBSCAN con la métrica euclidean es $s = 0{,}5636$ que le corresponde con umbral de distancia óptimo $Eps = 0{,}32$ y se estima que haya 3 clusters. Por otra parte, para la métrica manhattan, se estima el mismo número de clusters 3, y parecido coeficiente de Silhouette $s = 0{,}5642$ pero diferente umbral de distancia, $Eps = 0{,}4$. A continuación veremos las pequeñas diferencias entre algoritmos.

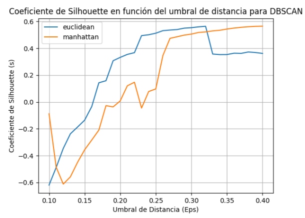
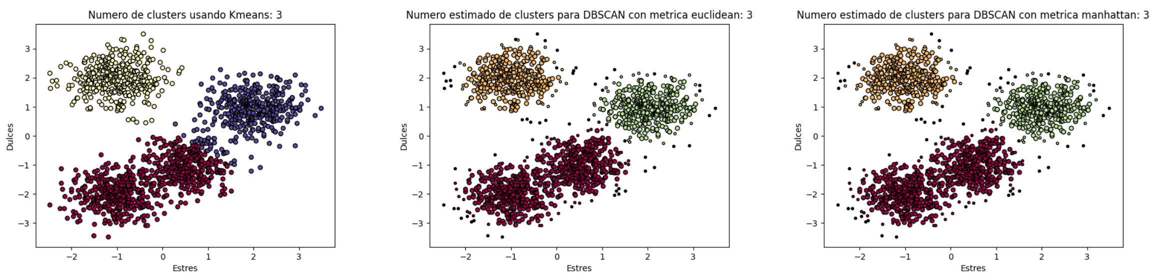


Figura 4: Relacion s en función de eps



(a) KMEANS (b) DBSCAN euclidean (c) DBSCAN manhattan

Figura 5: Comparación de las nubes de puntos dependiendo del algoritmo utilizado

Se puede observar que la única diferencia es, a parte de la distribución de los *core points* y *noise points*(se representa con el tamaño de los puntos), que los puntos que se encuentran en la frotera entre los clusters de color rojo y (verde o azul), usando KMEANS se quedan de color azul porque la distancia al centroide azul es más pequeña, pero usando DBSCAN se incluyen en el rojo porque hay más densidad de puntos en su vecindad.

Por último, para determinar la franja de edad a la que pertenecen las personas con coordenadas $a = (\frac{1}{2}, 0)$ y $b = (0, -3)$, vamos a graficarlos (Figura 6) . En él, se puede observar como la persona a pertenecería al cluster número 2 (color verde) y la persona b, al cluster numero 0 (color azul). Para comprobarlo, utilizamos la función kmeans.predict que nos muestran lo mismo: El cluster predicho para el punto [0.5, 0] es: [2], y para el punto [0, -3] es: [0]

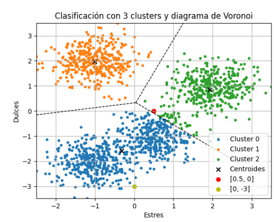


Figura 6: Predicción de la vecindad de los puntos

4. Conclusión

Nuestra experimentación demuestra que tanto KMeans como DBSCAN son herramientas útiles para analizar la distribución y agrupación de una población en función de sus características. El uso de métricas como el coeficiente de Silhouette nos permite evaluar y comparar la calidad de las clasificaciones obtenidas, lo que proporciona información valiosa para comprender mejor la estructura de los datos y las relaciones entre los individuos en el conjunto de datos de Villa Laminera

5. Anexo

5.1. Código implementado

```python
"""

    Asignatura: Geometría computacional
    Subgrupo: 1
    Curso: 2023-2024
    Alumno: Jiménez Poyatos, Pablo
    Curso: 4 CC
    Carrera: Grado en matemáicas.
    Práctica 2. Diagrama de Voronói y clustering.

"""

import os
import numpy as np
import matplotlib.pyplot as plt

from sklearn.cluster import KMeans, DBSCAN
from sklearn.metrics import silhouette_score
from scipy.spatial import Voronoi, voronoi_plot_2d




def preprocesamiento_datos(archive_name):
    """
    Función para cargar y preprocesar datos desde un archivo.

    Parameters:
        archive_name (str): Nombre del archivo que contiene los datos.

    Return:
        X (numpy.ndarray): Datos preprocesados.
    """

    ruta = os.getcwd()
    archivo = os.path.join(ruta, archive_name)
    X = np.loadtxt(archivo, skiprows=1)
    return X

def calculo_silhouette(X, rango_inf, rango_sup):
    """
    Calcula el coeficiente de silhouette para un rango dado de número de
    clusters.

    Parameters:
        X (numpy.ndarray): Datos.
        rango_inf (int): Límite inferior del rango de número de clusters.
        rango_sup (int): Límite superior del rango de número de clusters.

    Return:
        silhouette_scores (list): Lista de coeficientes de silhouette para cada
        número de clusters en el rango dado.
    """

    silhouette_scores = []
    for k in range(rango_inf, rango_sup + 1):
        kmeans = KMeans(n_clusters=k, random_state=0).fit(X)
        labels = kmeans.labels_
        silhouette = silhouette_score(X, labels)
        silhouette_scores.append(silhouette)
    return silhouette_scores

```

```python

def graficar(x,y,x_name, y_name, title_name, marker, style, metrics_l,
             grid=True, legend=False):
    """
    Función para graficar datos.

    Parámetros:
        x (list): Valores del eje x.
        y (list(list)): Valores del eje y (una lista de listas si se grafican
                                        múltiples líneas).
        x_name (str): Nombre del eje x.
        y_name (str): Nombre del eje y.
        title_name (str): Título del gráfico.
        marker (str): Tipo de marcador.
        style (str): Estilo de línea.
        metrics_l (list): Lista de nombres de las métricas.
        grid (bool): Indica si mostrar la cuadrícula en el gráfico
                    (por defecto True).
        legend (bool): Indica si mostrar la leyenda en el gráfico
                    (por defecto False).
    """

    plt.figure()
    for i in range(len(y)):
        plt.plot(x, y[i], marker=marker, linestyle=style, label=metrics_l[i])
    plt.xlabel(x_name)
    plt.ylabel(y_name)
    plt.title(title_name)
    plt.grid(grid)
    if legend:
        plt.legend()
    plt.show()

def grafica_voronoi(kmeans, opt_k, X, labels, points, x_name, y_name, title):
    """
    Función para graficar un diagrama de Voronoi.

    Parámetros:
        kmeans: Modelo de KMeans entrenado.
        opt_k (int): Número óptimo de clusters.
        X (numpy.ndarray): Datos.
        labels (numpy.ndarray): Etiquetas de los clusters.
        points (list): Lista de puntos para destacar en el gráfico.
        x_name (str): Nombre del eje x.
        y_name (str): Nombre del eje y.
        title (str): Título del gráfico.
    """

    plt.figure()
    vor = Voronoi(kmeans.cluster_centers_)
    voronoi_plot_2d(vor, show_vertices=False, show_points=False)
    for i in range(opt_k):
        plt.scatter(X[labels == i, 0], X[labels == i, 1], label=f'Cluster {i}',
                    s=10)
    plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
                marker='x', color='k', label='Centroides')

    # Adding two red points
    for i in range(len(points)):
        plt.scatter([points[i][0][0]], [points[i][0][1]], color=points[i][1],
                    label=str(points[i][0]))

    plt.xlabel(x_name)
    plt.ylabel(y_name)
    plt.title(title)
    plt.legend()
```

```python
    plt.grid(True)

    plt.xlim(min(X[:, 0]), max(X[:, 0]))  # Ajusta los límites del eje x
    plt.ylim(min(X[:, 1]), max(X[:, 1]))  # Ajusta los límites del eje y

    plt.show()


def grafica_Algoritmo(labels, X, n_clusters, title, x_title, y_title,
                      core_samples_mask = None):
    """
    Función para graficar clusters encontrados por un algoritmo de clustering.

    Parámetros:
        labels (numpy.ndarray): Etiquetas de los clusters.
        X (numpy.ndarray): Datos.
        n_clusters (int): Número de clusters.
        title (str): Título del gráfico.
        x_title (str): Nombre del eje x.
        y_title (str): Nombre del eje y.
        core_samples_mask (numpy.ndarray): Máscara de muestras centrales (opc).
    """

    unique_labels = set(labels)
    colors = [plt.cm.Spectral(each)
              for each in np.linspace(0, 1, len(unique_labels))]

    plt.figure()
    for k, col in zip(unique_labels, colors):
        if k == -1:
            # Black used for noise.
            col = [0, 0, 0, 1]

        class_member_mask = (labels == k)

        if core_samples_mask is not None:
            xy = X[class_member_mask & core_samples_mask]
            plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
                     markeredgecolor='k', markersize=5)

            xy = X[class_member_mask & ~core_samples_mask]
            plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
                     markeredgecolor='k', markersize=3)
        else:
            xy = X[class_member_mask]
            plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
                     markeredgecolor='k', markersize=5)
    plt.xlabel(x_title)
    plt.ylabel(y_title)
    plt.title(title + str(n_clusters))
    plt.show()


def calculos(metric, min_samples, X, epsilons):
    """
    Calcula el coeficiente de Silhouette y determina el número estimado de
    clusters utilizando DBSCAN para diferentes
    valores de epsilon (eps).

    Parámetros:
    metric : str
        La métrica de distancia a utilizar. Debe ser compatible con las
        métricas aceptadas por scikit-learn.
    min_samples : int
        El número mínimo de muestras requeridas para formar un cluster.
    X : array-like, shape (n_samples, n_features)
```

```
        La matriz de datos de entrada.
    epsilons : array-like
        Lista de valores de epsilon (eps) para probar.

    Retorna:
    lista : list
        Lista de coeficientes de Silhouette para cada valor de epsilon probado.
    labels : array-like, shape (n_samples,)
        Etiquetas de cluster asignadas por DBSCAN.
    core_samples_mask : array-like, shape (n_samples,)
        Máscara de muestras centrales identificadas por DBSCAN.
    n_clusters : int
        Número estimado de clusters identificados por DBSCAN.
    """

    lista = []
    for epsilon in epsilons:
        db = DBSCAN(eps=epsilon, min_samples=min_samples, metric=metric).fit(X)
        labels = db.labels_
        silhouette = silhouette_score(X, labels)
        lista.append(silhouette)

    max_silhouette = max(lista)
    optimal_epsilon = epsilons[lista.index(max_silhouette)]
    print(f"Mayor coef. de Silhouette para DBSCAN con métrica '{metric}':",
          max_silhouette)
    print(f"Eps óptimo para DBSCAN con métrica '{metric}':",
          optimal_epsilon)

    db = DBSCAN(eps=optimal_epsilon, min_samples=10, metric=metric).fit(X)
    core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
    core_samples_mask[db.core_sample_indices_] = True
    labels = db.labels_
    n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
    print(f"Número estimado de clusters para DBSCAN con métrica '{metric}':",
          n_clusters)
    return (lista,labels, core_samples_mask, n_clusters)





if __name__ == '__main__':

    #Preprocesamiento de datos
    X = preprocesamiento_datos("Personas_de_villa_laminera.txt")
    plt.plot(X[:, 0], X[:, 1], 'ro', markersize=2)
    plt.xlabel('Estrés')
    plt.ylabel('Dulces')
    plt.title('Población Villa Laminera')
    plt.grid(True)
    plt.show()


    ### APARTADO 1 ###

    # Calcular coeficiente de Silhouette para diferentes valores de k
    valores_s = calculo_silhouette(X, 2, 15)
    graficar(range(2, 16), [valores_s], 'Número de Vecindades (k)',
             'Coeficiente de Silhouette (s)',
             'Coeficiente de Silhouette en función de k', 'o', '-',
             [None], True)

    # Clasificar los datos con el número óptimo de vecindades
    max_sil_kmeans = max(valores_s)
    optimal_k = valores_s.index(max_sil_kmeans) + 2
```

```python
    print("Mayor coeficiente de Silhouette para KMeans:", max_sil_kmeans)
    print("Número óptimo de vecindades (k) para KMeans:", optimal_k)


    kmeans = KMeans(n_clusters=optimal_k, random_state=0).fit(X)
    labels = kmeans.labels_

    # Graficar la clasificación resultante con el diagrama de Voronoi
    points=[]
    grafica_voronoi(kmeans, optimal_k, X, labels, points, 'Estres', 'Dulces',
                    f'Clasificación con {optimal_k} clusters y Voronoi')

    grafica_Algoritmo(labels, X, optimal_k,
                      'Numero de clusters usando Kmeans: ', 'Estres', 'Dulces')



    ### APARTADO 2 ###:

    eps = np.arange(0.1, 0.4, 0.01)  # Rango de umbrales de distancia
    min_samples = 10  # Número mínimo de elementos en una vecindad
    metrics_list = ['euclidean', 'manhattan']

    # Calculamos Silhouette para cada combinación de parámetros
    valores_s = {}
    for metric in metrics_list:
        l,label, core_mask, n_cluster= calculos(metric, min_samples, X, eps)
        valores_s[metric] = l
        grafica_Algoritmo(label, X, n_cluster,
                          f'Estimación de clusters para DBSCAN con metrica {metric}: ',
                          'Estres', 'Dulces' , core_mask)

    # Gráfica comparativa
    graficar(eps, list(valores_s.values()), 'Umbral de Distancia (Eps)',
             'Coeficiente de Silhouette (s)',
             'Coeficiente de Silhouette en función Eps para DBSCAN',
             None, None, metrics_list, True, True)



    ### APARTADO 3 ###:

    # Coordenadas de los puntos
    a = [0.5, 0]
    b = [0, -3]
    punto_a = np.array([a])
    punto_b = np.array([b])

    # Predicción de los puntos utilizando el modelo KMeans
    cluster_a = kmeans.predict(punto_a)
    cluster_b = kmeans.predict(punto_b)

    points = [(a,'r'),(b,'y')]
    grafica_voronoi(kmeans, optimal_k, X, labels, points, 'Estres', 'Dulces',
                    f'Clasificación con {optimal_k} clusters y Voronoi')

    print(f"El cluster predicho para el punto {a} es:", cluster_a)
    print(f"El cluster predicho para el punto {b} es:", cluster_b)
```
Programa 1: Entrega2.DiagramaVoronoiClustering.py