Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Tarea 5

Clustering

Integrantes: Bastián Garcés
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla Z.
Ayudantes: Juan Pablo Cáceres B.
Rudy García
Rodrigo Salas O.
Sebastian Solanich
Pablo Troncoso P.

Fecha de entrega: 20 de Junio de 2021
Santiago, Chile

Índice de Contenidos

Índice de Figuras

Índice de Códigos

1. Introducción

En el presente informe se expondrá y detallará el desarrollo de la tarea número 5 del curso EL4106 Inteligencia Computacional, tarea que tenía como objetivo el utilizar distintos algoritmos de clustering para posteriormente analizar el desempeño de estos, los algoritmos de clustering a utilizar corresponden a *K-Means*, *DBSCAN* y *clustering aglomerativo*. Para realizar lo anterior se utilizará la base de datos *Anuran Call MFCCs*, que corresponde a una base de datos tomada del *UC Irvine Machine Learning Repository*, esta base de datos contiene llamadas de 10 especies de rana, las cuales son representadas mediante 22 características. Además se recurrirá a un código basado en *Scikit-Learn* para la realización de esta tarea. Las bibliotecas a utilizar corresponden a *Pandas*, *Sklearn* y *Time* mientras que el entorno de trabajo a utilizar corresponde a *Colaboratory*.

Las secciones que tendrá el presente informe se corresponden con las distintas actividades que el cuerpo docente solicitó realizar. A continuación se enumeran las distintas secciones que tendrá el documento:

- **Actividad 1:** La primera actividad a realizar corresponde a darle lectura a la base de datos entregada por el cuerpo docente, además, los datos deben dividirse en características (MFCCs_1 - MFCCs_22) y labels (Species), siendo esto último transformado a números en el rango de 0 a 9.
- **Actividad 2:** En esta actividad se debía modificar la función *bench_k_means()*, entregada en el código base, de modo que a la salida de esta se le agregue una columna extra, esta columna corresponde al número de clusters encontrados por el algoritmo en cuestión.
- **Actividad 3:** En la tercera actividad se implementará la función *bench_clustering2()*, que está basada en la función implementada en la actividad 3. Esta función deberá ser utilizada para hacer clustering mediante el algoritmo *DBSCAN* y *clustering aglomerativo*, además debe tener ciertas peculiaridades, las cuales se enumeran a continuación:
  1. Esta función no deberá imprimir la variable inertia_ pues los algoritmos *DBSCAN* y *clustering aglomerativo* no poseen dicha variable.
  2. El algoritmo *DBSCAN* no asigna todas las muestras a un cluster, hay casos donde algunas de estas no quedan asociadas ningún cluster y se les otorga el label -1. Debido a lo anterior la función tendrá un parámetro booleano que indicará cuantas muestras se deben utilizar para obtener las muestras. Si el parámetro booleano es True significa que se calcularán las métricas utilizando todas las muestras, al hacer esto las muestras con label -1 se asignarán a un cluster extra. Luego, si el parámetro booleano es False solo se utilizarán las muestras con un label distinto de -1.
  3. Debido a que el algoritmo *DBSCAN* no genera una cantidad predeterminada de clusters habrán ocasiones en que se obtendrá 0 o 1 cluster motivo por el cual no se podrán obtener las métricas, debido a lo anterior la función deberá ser capaz de dar resolución a aquellas situaciones en que las métricas no se puedan calcular o generen errores.
- **Actividad 4:** Esta actividad corresponde a hacer pruebas con distintas variantes de algoritmos y calcular las métricas de estos con las funciones implementadas anteriormente. Las pruebas a realizar corresponden a:

1. K-Means (con inicialización al azar)
2. K-Means++
3. DBSCAN con épsilon por defecto
4. DBSCAN con épsilos 0.7
5. DBSCAN con épsilon 0.2
6. DBSCAN con épsilon por defecto, agregando outliers a cluster extra
7. DBSCAN con épsilos 0.7, agregando outliers a cluster extra
8. DBSCAN con épsilon 0.2, agregando outliers a cluster extra
9. Clustering aglomerativo
10. Repetir las pruebas anteriores, después de utilizar PCA sobre los datos para reducirlos a dos dimensiones.

- **Actividad 5:** La última actividad corresponde a una actividad de análisis, donde se deberán analizar los resultados obtenidos ante las distintas pruebas realizadas. Los puntos a analizar son:

  1. Comparar los algoritmo aplicados en base a cuatro métricas (Completeness, Homogeneity, V-Measure, Silhuette) indicando que variante era mejor según cada métrica y cual la peor.
  2. Indicar en cuales casos no fue posible calcular las métricas de DBSCAN. Además se analizará el número de clusters obtenidos por DBSCAN y el efecto sobre las métricas. Para finalmente analizar el hecho de considerar o no los outliers del algoritmo *DBSCAN* como un cluster extra y el efecto de esto sobre las métricas.
  3. Analizar el efecto de usar PCA sobre las métricas obtenidas.

2. Desarrollo

En la presente sección se expondrá el código implementado para desarrollar las distintas actividades expuestas durante la introducción. Cabe destacar que en esta sección solo se expondrán los resultados y el código implementado en las distintas actividades, más no se ahondará en lo que se debía hacer en cada una, esto fue realizado en la introducción.

2.1. Actividad 1

Para subir el archivo, darle lectura, separar en características y clases, además de hacer que las clases vayan de 0 a 9 se utilizó el código 1.

Código 1: Lectura de datos y configuraciones iniciales

```
#Para subir el archivo
from google.colab import files
uploaded = files.upload()

!ls

df = pd.read_csv('Frogs_MFCCs.csv')
df

#Separamos en caracteristicas y clase
MFCCs = df.iloc[:, :22]
Species = df['Species']
tipos = []
#Hacemos que las clases vayan de 0 a 9
for i in range(len(Species)):
  if Species[i] not in tipos:
    tipos.append(Species[i])
  Species[i] = tipos.index(Species[i])
```

2.2. Actividad 2

La función a modificar es la que se puede observar en el código 2.

Código 2: Código base

```
def bench_k_means(kmeans, name, data, labels):
    """Benchmark to evaluate the KMeans initialization methods.

    Parameters
    ----------
    kmeans : KMeans instance
        A :class:`~sklearn.cluster.KMeans` instance with the initialization
        already set.
    name : str
        Name given to the strategy. It will be used to show the results in a
```

```
        table.
    data : ndarray of shape (n_samples, n_features)
        The data to cluster.
    labels : ndarray of shape (n_samples,)
        The labels used to compute the clustering metrics which requires some
        supervision.
    """
    t0 = time()
    estimator = make_pipeline(StandardScaler(), kmeans).fit(data)
    fit_time = time() - t0
    results = [name, fit_time, estimator[-1].inertia_]

    # Define the metrics which require only the true labels and estimator
    # labels
    clustering_metrics = [
        metrics.homogeneity_score,
        metrics.completeness_score,
        metrics.v_measure_score,
        metrics.adjusted_rand_score,
        metrics.adjusted_mutual_info_score,
    ]
    results += [m(labels, estimator[-1].labels_) for m in clustering_metrics]

    # The silhouette score requires the full dataset
    results += [
        metrics.silhouette_score(data, estimator[-1].labels_,
                                 metric="euclidean", sample_size=300,)
    ]

    # Show the results
    formatter_result = ("{:9s}\t{:.3f}s\t{:.0f}\t{:.3f}\t{:.3f}"
                        "\t{:.3f}\t{:.3f}\t{:.3f}")
    print(formatter_result.format(*results))
```

Luego la función modificada corresponde a la que se puede observar en el código 3, dicho código se diferencia del código 2 en que se obtuvo el vector que indica el cluster al que pertenece cada muestra, luego simplemente se utilizó la función *set()* sobre el vector obtenido para obtener los cluster existentes, posteriormente se calculó el largo de vector de clusters, obteniendo de dicha forma la cantidad de estos, continuando se añadió la cantidad de clusters obtenidos al vector de resultados. Finalmente se modificó la variable *formatter_result* de tal forma que imprimiese una variable más, logrando así que se imprimiese la cantidad de clusters.

Código 3: Función modificada en actividad 2

```
def bench_k_means(kmeans, name, data, labels):
    """Benchmark to evaluate the KMeans initialization methods.

    Parameters
    ----------
    kmeans : KMeans instance
        A :class:`~sklearn.cluster.KMeans` instance with the initialization
```

```
        already set.
    name : str
        Name given to the strategy. It will be used to show the results in a
        table.
    data : ndarray of shape (n_samples, n_features)
        The data to cluster.
    labels : ndarray of shape (n_samples,)
        The labels used to compute the clustering metrics which requires some
        supervision.
    """
    t0 = time()
    estimator = make_pipeline(StandardScaler(), kmeans).fit(data)
    fit_time = time() - t0
    results = [name, fit_time, estimator[-1].inertia_]

    # Define the metrics which require only the true labels and estimator
    # labels
    clustering_metrics = [
        metrics.homogeneity_score,
        metrics.completeness_score,
        metrics.v_measure_score,
        metrics.adjusted_rand_score,
        metrics.adjusted_mutual_info_score,
    ]
    results += [m(labels, estimator[-1].labels_) for m in clustering_metrics]

    # The silhouette score requires the full dataset
    results += [
        metrics.silhouette_score(data, estimator[-1].labels_,
                                 metric="euclidean", sample_size=300,)
    ]

    clusterizacion = estimator[-1].labels_ #Obtenemos las etiquetas de cada punto
    nro_clusters = len(set(clusterizacion)) #Vemos cuantos clusters hay

    results += [nro_clusters] #ponemos la cantidad de clusters obtenidos a los resultados
    # Show the results
    formatter_result = ("{:9s}\t{:.3f}\t{:.0f}\t{:.3f}\t{:.3f}"
                        "\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t\t{:0}")
    print(formatter_result.format(*results))
```

2.3. Actividad 3

La función implementada para obtener el benchmark tanto para el algoritmo de clustering *DBSCAN* como el *clustering aglomerativo* corresponde a la que se observa en el código 4.

Código 4: Función para obtener benchmark de clustering DBSCAN y aglomerativo

```
def bench_clustering2(kmeans, name, data, labels, bool):
    """Benchmark to evaluate the KMeans initialization methods.
```

```
3
4     Parameters
5     ----------
6     kmeans : KMeans instance
7         A :class:‘~sklearn.cluster.KMeans‘ instance with the initialization
8         already set.
9     name : str
10        Name given to the strategy. It will be used to show the results in a
11        table.
12    data : ndarray of shape (n_samples, n_features)
13        The data to cluster.
14    labels : ndarray of shape (n_samples,)
15        The labels used to compute the clustering metrics which requires some
16        supervision.
17    bool : bool
18        Indica si se utilizan todas las muestras (True) o no (False)
19    """
20    t0 = time()
21    estimator = make_pipeline(StandardScaler(), kmeans).fit(data)
22    fit_time = time() - t0
23    results = [name, fit_time, 'NAN']
24
25    # Define the metrics which require only the true labels and estimator
26    # labels
27    clustering_metrics = [
28        metrics.homogeneity_score,
29        metrics.completeness_score,
30        metrics.v_measure_score,
31        metrics.adjusted_rand_score,
32        metrics.adjusted_mutual_info_score,
33    ]
34
35    #Estos parametros son usados si bool es True y si es False:
36    clusterizacion = estimator[-1].labels_ #estimacion inicial
37    new_clustering = [] #Aca guardamos los labels de los datos finales, los labels obtenidos
38                        #dependeran de si bool es True o False
39    #-----------------------------------------------------------------------
40    #Estos otros parametros son usados solo si bool es False, para el caso
41    new_labels = [] #Aca se guardaran los labels correctos
42    new_data = pd.DataFrame() #Aca se guardaran los datos que obtuvieron un label distinto a -1
43    #-----------------------------------------------------------------------
44    #Si bool es true significa que usare todos los datos, incluyendo aquellos
45    #datos que no tienen un cluster asociado (y que por tanto estan en el
46    #cluster -1)
47    if bool == True:
48      maximo = max(clusterizacion)
49      new_labels = labels
50      new_data = data
51      for i in range(len(clusterizacion)):
52        if clusterizacion[i] == -1: #Los datos no asociados a un cluster se consideran
53                            # en un cluster propio
```

```
54          new_clustering.append(maximo+1)
55        else:
56          new_clustering.append(clusterizacion[i])
57
58    #Si bool es false significa que solo considerare los datos que tienen
59    #un cluster asociado distinto de -1
60    elif bool == False:
61      for i in range(len(clusterizacion)):
62        if clusterizacion[i] != -1: #si el elemento si esta asociado a un cluster
63          new_clustering.append(clusterizacion[i]) #Guardamos la clusterizacion del elemento
64          new_labels.append(labels[i]) #Guardamos la correcta clasificacion del elemento
65          new_data = pd.concat([new_data, data.iloc[i]], axis=1)
66        #Los elemenotos con cluster -1 no se consideran
67      new_data = new_data.transpose() #Para que las caracteristicas queden como columnas
68
69    #Cantidad de clusters
70    nro_clusters = len(set(new_clustering))
71
72    try:
73      if nro_clusters <= 1: #Solo hay 1 cluster o no hay ninguno
74        #Obtenemos las metricas
75        results += [m(new_labels, new_clustering) for m in clustering_metrics]
76        results +=['NAN'] #No hay silhuette
77        results += [nro_clusters] #ponemos la cantidad de clusters obtenidos a los resultados
78        # Show the results
79        formatter_result = ("{:9s}\t{:.3f}\t{:3s}\t{:.3f}\t{:.3f}"
80                    "\t{:.3f}\t{:.3f}\t{:.3f}\t{:3s}\t\t{:0}")
81        print(formatter_result.format(*results))
82
83      elif nro_clusters > 1: #Hay mas de 1 cluster
84        #Obtenemos las metricas
85        results += [m(new_labels, new_clustering) for m in clustering_metrics]
86        #The silhouette score requires the full dataset
87        results += [
88           metrics.silhouette_score(data, estimator[-1].labels_,
89                                    metric="euclidean", sample_size=300,)
90        ]
91        results += [nro_clusters] #ponemos la cantidad de clusters obtenidos a los resultados
92        # Show the results
93        formatter_result = ("{:9s}\t{:.3f}\t{:3s}\t{:.3f}\t{:.3f}"
94                      "\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t\t{:0}")
95        print(formatter_result.format(*results))
96
97    except:
98      print("Para el caso "+str(name)+" hubo un error al calcular las metricas")
```

El código anterior opera como sigue: primero se obtiene el vector que indica el cluster al que se asoció cada muestra, luego se definen unos vectores que ayudarán a operar la función si bool es True o False. El vector *new_clustering* guardará el cluster de los datos a utilizar, si bool es True guardará el cluster de todas las muestras, incluyendo aquellas que no quedaron con un cluster asociado. Luego el vector *new_labels* guardará la clase correcta de las distintas muestras, en el caso de que bool sea

True guardará la clase de todas las muestras y en caso de que sea False guardará la clase de las muestras cuyo cluster es distinto de -1. Finalmente se creó un dataframe llamado *new_data*, dicho dataframe guardará las muestras dependiendo de si bool es True o False, si es True simplemente guardará todas las muestras, pero si bool es False guardará las muestras cuyo cluster es distinto de -1. A modo de explicación se debe hacer hincapié en el hecho de que una muestra quede con cluster -1 no significa que su cluster es el -1 sino que indica que dicha muestra no quedó asociada a ningún cluster.

Con las distintas variables definidas se pasa a ver si bool es True o False, si es True significa que se desean utilizar todas las muestras motivo por el cual el vector *new_labels* y el dataframe *new_data* corresponderán al vector con la clase de todas las muestras y las características de todas las muestras respectivamente. Luego, se hizo que todas las muestras sin un cluster asociado quedaran asociadas al mismo cluster, dicho cluster corresponde al máximo cluster más 1, es decir, si se obtuvieron los clusters 0, 1 y 2 las muestras con cluster -1 quedarán asociadas al cluster 3, las muestras que ya tenían un cluster asociado no se ven modificadas. Para el caso donde bool era False se hizo que en el vector *new_clustering* se guardaran los clusters de las muestras que quedaron asociadas a un cluster, en *new_labels* se guardaron las clases de las muestras que tienen un cluster asociado y en *new_data* se añadió la muestra con cluster asociado, finalmente solo se traspuso el dataframe obtenido de forma tal que las características quedaran como columna en vez de fila.

La última parte del código se encarga de hacer el cálculo de las distintas métricas, cabe destacar que en ocasiones se producían errores a la hora de obtener las métricas para los casos donde habían 0, 1 o 2 clusters, por este motivo se utilizó el método *try*, de tal forma que si había problema para el cálculo de alguna métrica simplemente se mostrara en pantalla un mensaje indicando que no fue posible obtener las métricas. Asumiendo un caso donde no hay problema con el cálculo de las métricas (no se produjo un error en la ejecución del código), es decir, el programa está trabajando dentro del *try*, se debe verificar si el número de clusters es menor o igual que 1, ante dicho caso el programa tiene problemas para calcular la métrica Silhuette, pues si solo hay un cluster o no hay ninguno no es posible obtener dicha métrica, motivo por el cual esta métrica queda asociada a un "NAN" que indica que no se puede obtener dicha métrica, el resto de métricas se calculan sin problemas. Luego para el caso donde el número de clusters es mayor que 1 no se producirán errores en el cálculo de ninguna métrica, motivo por el cual se calculan de la misma forma que en la actividad 3. Se debe hacer mención en el hecho de que si alguna métrica presenta error en su cálculo simplemente se mostrará un mensaje indicando que hubo un error calculando las métricas.

2.4. Actividad 4

En esta actividad se realizaron diversas pruebas, estas pruebas consisten en realizar clustering utilizando el algoritmo *K-Means*, *DBSCAN* y *Clustering aglomerativo*, además se hizo una reducción de dimensionalidad utilizando PCA, haciendo que las dimensiones fuesen 2. Las pruebas realizadas se pueden observar en el código 5.

Código 5: Pruebas realizadas

```
1 #Pruebas:
2
3 print(98 * '_')
4 print('init\t\ttime\tinertia\thomo\tcompl\tv-meas\tARI\tAMI\tsilhouette\tn_clusters')
```

```
n_clusters_pred = 10 #Cantidad de clusters a usar en KMeans y Aglomerative
#Kmeans con inicializacion al azar
kmeans = KMeans(init="random", n_clusters=n_clusters_pred, n_init=4,
            random_state=0)
bench_k_means(kmeans=kmeans, name="random", data=MFCCs, labels=Species)

#Kmeans++
kmeans = KMeans(init="k-means++", n_clusters=n_clusters_pred, n_init=4,
            random_state=0)
bench_k_means(kmeans=kmeans, name="k-means++", data=MFCCs, labels=Species)

#DBSCAN con epsilon por defecto (0.5)
dbscan = DBSCAN(eps=0.5).fit(MFCCs)
bench_clustering2(kmeans= dbscan, name='DBSCAN-0.5-f', data=MFCCs, labels=Species, bool=
    ↪ False)

#DBSCAN con epsilon 0.7
dbscan = DBSCAN(eps=0.7).fit(MFCCs)
bench_clustering2(kmeans= dbscan, name='DBSCAN-0.7-f', data=MFCCs, labels=Species, bool=
    ↪ False)

#DBSCAN con epsilon 0.2
dbscan = DBSCAN(eps=0.2).fit(MFCCs)
bench_clustering2(kmeans= dbscan, name='DBSCAN-0.2-f', data=MFCCs, labels=Species, bool=
    ↪ False)

#DBSCAN con epsilon por defecto (0.5), agregando outliers a cluster extra
dbscan = DBSCAN(eps=0.5).fit(MFCCs)
bench_clustering2(kmeans= dbscan, name='DBSCAN-0.5-t', data=MFCCs, labels=Species, bool=
    ↪ True)

#DBSCAN con epsilon 0.7, agregando outliers a cluster extra
dbscan = DBSCAN(eps=0.7).fit(MFCCs)
bench_clustering2(kmeans= dbscan, name='DBSCAN-0.7-t', data=MFCCs, labels=Species, bool=
    ↪ True)

#DBSCAN con epsilon 0.2, agregando outliers a cluster extra
dbscan = DBSCAN(eps=0.2).fit(MFCCs)
bench_clustering2(kmeans= dbscan, name='DBSCAN-0.2-t', data=MFCCs, labels=Species, bool=
    ↪ True)

#Clustering aglomerativo
aglomerative = AgglomerativeClustering(n_clusters=n_clusters_pred)
bench_clustering2(kmeans=aglomerative, name='Aglomerative', data=MFCCs, labels=Species, bool
    ↪ =True)
print(98 * '_')
#------------------------------------------------------------------------------
#Aplicando PCA
MFCCs_red = PCA(n_components=2).fit_transform(MFCCs)
MFCCs_red_df = pd.DataFrame(MFCCs_red)

```

```
print("")
print("Al aplicar PCA se obtienen los siguientes resultados:")
print(98 * '_')
print('init\t\ttime\tinertia\thomo\tcompl\tv-meas\tARI\tAMI\tsilhouette\tn_clusters')
#Kmeans con inicializacion al azar
kmeans = KMeans(init="random", n_clusters=n_clusters_pred, n_init=10)
bench_k_means(kmeans=kmeans, name="random", data=MFCCs_red_df, labels=Species)

#Kmeans++
kmeans = KMeans(init="k-means++", n_clusters=n_clusters_pred, n_init=10)
bench_k_means(kmeans=kmeans, name="k-means++", data=MFCCs_red_df, labels=Species)

#DBSCAN con epsilon por defecto (0.5)
dbscan = DBSCAN(eps=0.5).fit(MFCCs)
bench_clustering2(kmeans= dbscan, name='DBSCAN-0.5-f', data=MFCCs_red_df, labels=Species,
    ↪ bool=False)

#DBSCAN con epsilon 0.7
dbscan = DBSCAN(eps=0.7).fit(MFCCs)
bench_clustering2(kmeans= dbscan, name='DBSCAN-0.7-f', data=MFCCs_red_df, labels=Species,
    ↪ bool=False)

#DBSCAN con epsilon 0.2
dbscan = DBSCAN(eps=0.2).fit(MFCCs)
bench_clustering2(kmeans= dbscan, name='DBSCAN-0.2-f', data=MFCCs_red_df, labels=Species,
    ↪ bool=False)

#DBSCAN con epsilon por defecto (0.5), agregando outliers a cluster extra
dbscan = DBSCAN(eps=0.5).fit(MFCCs)
bench_clustering2(kmeans= dbscan, name='DBSCAN-0.5-t', data=MFCCs_red_df, labels=Species,
    ↪ bool=True)

#DBSCAN con epsilon 0.7, agregando outliers a cluster extra
dbscan = DBSCAN(eps=0.7).fit(MFCCs)
bench_clustering2(kmeans= dbscan, name='DBSCAN-0.7-t', data=MFCCs_red_df, labels=Species,
    ↪ bool=True)

#DBSCAN con epsilon 0.2, agregando outliers a cluster extra
dbscan = DBSCAN(eps=0.2).fit(MFCCs)
bench_clustering2(kmeans= dbscan, name='DBSCAN-0.2-t', data=MFCCs_red_df, labels=Species,
    ↪ bool=True)

#Clustering aglomerativo
aglomerative = AgglomerativeClustering(n_clusters=n_clusters_pred)
bench_clustering2(kmeans=aglomerative, name='Aglomerative', data=MFCCs_red_df, labels=
    ↪ Species, bool=True)

print(98 * '_')
```

Los resultados obtenidos para las pruebas se observan en la figura 1. Las pruebas no fueron expuestas en forma de código o tabla debido a que si se mostraban como código los valores de las métricas

se desalineaban con respecto a la columna, y si se utilizaba una tabla se salía de los márgenes del documento, motivo por el cual se decidió exponer los resultados en forma de imagen.

init	time	inertia	homo	compl	v-meas	ARI	AMI	silhouette	n_clusters
random	0.316	54400	0.641	0.507	0.566	0.326	0.565	0.217	10
k-means++	0.301	51996	0.708	0.571	0.633	0.432	0.631	0.256	10
DBSCAN-0.5-f	0.537	NAN	1.000	0.327	0.493	0.256	0.483	-0.202	8
DBSCAN-0.7-f	0.675	NAN	1.000	0.585	0.738	0.450	0.730	-0.237	13
DBSCAN-0.2-f	0.335	NAN	1.000	0.000	0.000	0.000	0.000	-0.080	2
DBSCAN-0.5-t	0.541	NAN	0.074	0.451	0.127	0.049	0.123	-0.175	9
DBSCAN-0.7-t	0.673	NAN	0.179	0.545	0.270	0.133	0.265	-0.235	14
DBSCAN-0.2-t	0.323	NAN	0.011	0.677	0.022	0.006	0.021	-0.132	3
Aglomerative	2.223	NAN	0.789	0.661	0.719	0.590	0.718	0.239	10

Al aplicar PCA se obtienen los siguientes resultados:

init	time	inertia	homo	compl	v-meas	ARI	AMI	silhouette	n_clusters
random	0.444	942	0.654	0.499	0.566	0.363	0.565	0.426	10
k-means++	0.358	942	0.671	0.518	0.585	0.403	0.584	0.403	10
DBSCAN-0.5-f	0.199	NAN	0.000	1.000	0.000	0.000	0.000	NAN	1
DBSCAN-0.7-f	0.283	NAN	-0.000	1.000	-0.000	0.000	-0.000	NAN	1
DBSCAN-0.2-f	0.115	NAN	0.002	0.198	0.004	-0.001	0.003	0.001	3
Para el caso DBSCAN-0.5-t hubo un error al calcular las metricas									
Para el caso DBSCAN-0.7-t hubo un error al calcular las metricas									
DBSCAN-0.2-t	0.124	NAN	0.004	0.103	0.007	0.001	0.005	0.228	4
Aglomerative	1.610	NAN	0.649	0.490	0.558	0.350	0.557	0.360	10

Figura 1: Resultados obtenidos ante las distintas pruebas.

2.5. Actividad 5

A continuación se expondrá el análisis solicitado por el cuerpo docente, análisis que se basa en los resultados obtenidos en la actividad anterior.

1. Primero se compararán los resultados de los distintos algoritmos sobre la métrica Homogeneity, donde se logra observar que el algoritmo que tiene el mejor homogeneity es el algoritmo *DBSCAN* cuando solo se están utilizando las muestras que quedaron asociadas a un cluster (lo que es equivalente a decir un cluster distinto a -1) donde independiente del valor de épsilon utilizado el homogeneity es siempre 1, que es el máximo valor posible, luego el algoritmo con peor homogeneity es el *DBSCAN* cuando se redujo la dimensionalidad de los datos a 2, en este caso vale la pena detenerse pues se logra observar que para el caso *DBSCAN* con épsilon 0.7 utilizando las muestras asociadas a un cluster se obtiene un homogeneity negativo, lo cual es un error pues esta métrica solo puede ir de 0 a 1, sin embargo, este error se respalda en el hecho de que se tiene únicamente un cluster, donde es sabido que se presentan problemas a la hora de calcular las métricas, otra forma de interpretar ese valor negativo en la métrica podría ser en que el resultado de esta es demasiado pequeño, lo que provoca que a la hora de imprimirlo en la pantalla sea interpretado como un número negativo.

   Para el caso de la métrica Completeness se observa que el algoritmo que tiene el valor más grande en esta métrica es el *DBSCAN* con épsilon 0.5 y 0.7 utilizando los datos con cluster asociado y reducción de dimensionalidad a 2 donde se logra observar que se obtiene el valor más grande posible, luego se observa que el algoritmo *DBSCAN* con épsilon 0.2 utilizando las muestras asociadas a un cluster tiene el peor valor posible en esta métrica, teniendo un Com-

pleteness de 0, que es el mínimo valor posible.

El algoritmo que presenta el mejor valor en la métrica V-Measure es el *DBSCAN* con épsilon 0.7, luego los algoritmos que tienen la peor métrica es el *DBSCAN* con épsilon 0.2 usando los datos asociados a un cluster, *DBSCAN* con épsilon 0.5 considerando las muestras asociadas a un cluster y usando PCA, y *DBSCAN* con épsilon 0.7 usando muestras con cluster distinto de -1 y usando PCA, donde para el último caso se logra observar una métrica de valor negativo, lo cual no debería pasar, sin embargo se observa que se tiene únicamente 1 cluster motivo por el cual es probable que internamente se esté asociado esta métrica como un valor muy pequeño y que el programa lo esté asociando a un número negativo a la hora de imprimirlo en pantalla.

Finalmente el algoritmo con mejor Silhuette corresponde a *K-Means* con inicialización al azar una vez se redujo la dimensionalidad a 2, por otro lado el algoritmo cuyo Silhuette es el peor corresponde a *DBSCAN* con épsilon 0.7 utilizando los datos con cluster asociado

Se podría decir que los algoritmos que más fueron mencionados corresponden a los tipo *DBSCAN* con distintos épsilon y utilizando solo aquellas muestras asociadas a un cluster distinto de -1, lo cual tiene sentido pues si se consideran aquellas muestras que no estaban asociadas a un cluster se genera un cluster extra, provocando que el cálculo de las métricas fallé demasiado, para entender esto último se debe considerar el hecho de que si se están utilizando todas las muestras se producirá que muchas de estas no quedarán en el cluster al que realmente pertenecen, lo que provocará que existan muchas discrepancias entre los clusters predichos y los reales y que por tanto las métricas empeoren significativamente. Por otro lado se debe hacer mención al hecho de que reducir la dimensionalidad favorece a algunas métricas mientras que a otras las perjudica.

2. Los casos donde no se pudieron obtener las métricas son en el caso del algoritmo *DBSCAN* donde se usó un épsilon de 0.5 y 0.7, se utilizaron todas las muestras y se redujo dimensionalidad, también se podría decir que los casos *DBSCAN* con épsilon 0.7 y 0.5 donde se utilizaron aquellas muestras asociadas a un cluster y se redujo dimensionalidad también presentan errores en las métricas pues se obtienen algunos valores negativos en estas, lo cual tiene sentido si se tiene únicamente un cluster, motivo por el cual si bien el programa no arrojó error en su ejecución se podría decir que estas métricas están erradas.

   El efecto del número de clusters en las métricas del algoritmo *DBSCAN* reside en que a mayor cantidad de clusters hay menos errores en el cálculo de las métricas, para observar esto basta con apreciar que para los casos *DBSCAN* con dimensionalidad reducida y distintos épsilon el número de clusters obtenidos son pocos y las métricas obtenidas están erradas o tuvieron problemas a la hora de ser calculadas, mientras que para el caso *DBSCAN* donde no se redujo dimensionalidad nunca se presentaron errores en la obtención de las métricas.

   Finalmente se puede afirmar que el hecho de considerar los outliers en el algoritmo *DBSCAN*, provoca que las métricas empeoren bastante, un claro ejemplo de esto es el caso *DBSCAN* con épsilon 0.7 al considerar los outliers, donde los resultados de las métricas son inferiores a los obtenidos al usar el mismo algoritmo pero sin los outliers.

3. El hecho de utilizar PCA afecta de formas diferentes a los distintos algoritmos, para el algoritmo *K-Means* el usar PCA favorece a algunas métricas y a otras las empeora, por lo cual no se podría decir si el hecho de usar PCA favorece o no a este algoritmo. Para el caso del algoritmo *DBSCAN* se observa que reducir la dimensionalidad genera bastantes problemas a la hora de obtener las métricas, motivo por el cual se podría decir que utilizar PCA y el algoritmo *DBSCAN* no es recomendable. Finalmente el algoritmo de *clustering aglomerativo* presenta el mismo comportamiento que *K-Means*, es decir, se ve favorecido en el cálculo de algunas métricas pero otras se ven perjudicadas.

Algo que vale la pena mencionar y que enriquece el análisis anterior es el hecho de que los resultados obtenidos no son "universales", con esto se hace referencia a que entre distintas ejecuciones del código los resultados obtenidos irán variando, sin embargo es de esperar que el algoritmo *DBSCAN* seguirá presentando, en general, los mismos problemas.

3. Conclusión

A modo de conclusión se puede afirmar el cumplimiento de los objetivos impuestos durante la tarea número 5 del curso EL4106 Inteligencia Computacional, donde se logró implementar distintas funciones que permiten observar las distintas métricas de los algoritmos de clustering utilizados.

Dentro de los aprendizajes obtenidos destaca el hecho de utilizar los algoritmos para hacer clustering *K-Means*, *DBSCAN* y *Clustering Aglomerativo*, además del hecho de lograr implementar una función que permite apreciar las métricas de los algoritmos antes impuestos de forma tal que realizar una comparación entre los distintos algoritmos sea mucho más simple.

En cuanto a los resultados obtenidos se debe hacer una mención al hecho de que los resultados obtenidos no son siempre los mismos y que entre distintas ejecuciones del código los resultados obtenidos serán diferentes, sobretodo en el caso del algoritmo *DBSCAN*, pues es un algoritmo que puede arrojar distinta cantidad de clusters entre ejecuciones del código, además se destaca el hecho de que *DBSCAN* presenta muchos problemas en el cálculo de las métricas cuando se utiliza PCA para reducir la dimensionalidad.

Finalmente a modo de mejora se podría intentar implementar una función *bench_clustering2* que tarde menos tiempo arrojar las respuestas de las pruebas realizadas, pues a la hora de probar el código se necesitó de un minuto para lograr obtener los resultados de todas las pruebas realizadas. Por otra parte se podría mejorar la forma en que se calculan las métricas cuando se tiene 0 o 1 cluster, donde se podría hacer que siempre que se tenga 1 o menos clusters todas las métricas sean "incalculables", esto permitiría que no se diesen casos con métricas que tienen valores negativos.

4. Anexo

Código 6: Código implementado

```
#Para subir el archivo
from google.colab import files
uploaded = files.upload()

!ls

df = pd.read_csv('Frogs_MFCCs.csv')
df

#Separamos en caracteristicas y clase
MFCCs = df.iloc[:, :22]
Species = df['Species']
tipos = []
#Hacemos que las clases vayan de 0 a 9
for i in range(len(Species)):
  if Species[i] not in tipos:
    tipos.append(Species[i])
  Species[i] = tipos.index(Species[i])

#------

def bench_k_means(kmeans, name, data, labels):
    """Benchmark to evaluate the KMeans initialization methods.

    Parameters
    ----------
    kmeans : KMeans instance
        A :class:`~sklearn.cluster.KMeans` instance with the initialization
        already set.
    name : str
        Name given to the strategy. It will be used to show the results in a
        table.
    data : ndarray of shape (n_samples, n_features)
        The data to cluster.
    labels : ndarray of shape (n_samples,)
        The labels used to compute the clustering metrics which requires some
        supervision.
    """
    t0 = time()
    estimator = make_pipeline(StandardScaler(), kmeans).fit(data)
    fit_time = time() - t0
    results = [name, fit_time, estimator[-1].inertia_]

    # Define the metrics which require only the true labels and estimator
    # labels
    clustering_metrics = [
```

```
47          metrics.homogeneity_score,
48          metrics.completeness_score,
49          metrics.v_measure_score,
50          metrics.adjusted_rand_score,
51          metrics.adjusted_mutual_info_score,
52      ]
53      results += [m(labels, estimator[-1].labels_) for m in clustering_metrics]
54
55      # The silhouette score requires the full dataset
56      results += [
57          metrics.silhouette_score(data, estimator[-1].labels_,
58                                   metric="euclidean", sample_size=300,)
59      ]
60
61      clusterizacion = estimator[-1].labels_ #Obtenemos las etiquetas de cada punto
62      nro_clusters = len(set(clusterizacion)) #Vemos cuantos clusters hay
63
64      results += [nro_clusters] #ponemos la cantidad de clusters obtenidos a los resultados
65      # Show the results
66      formatter_result = ("{:9s}\t{:.3f}\t{:.0f}\t{:.3f}\t{:.3f}"
67                          "\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t\t{:0}")
68      print(formatter_result.format(*results))
69
70  #------
71
72  def bench_clustering2(kmeans, name, data, labels, bool):
73      """Benchmark to evaluate the KMeans initialization methods.
74
75      Parameters
76      ----------
77      kmeans : KMeans instance
78          A :class:`~sklearn.cluster.KMeans` instance with the initialization
79          already set.
80      name : str
81          Name given to the strategy. It will be used to show the results in a
82          table.
83      data : ndarray of shape (n_samples, n_features)
84          The data to cluster.
85      labels : ndarray of shape (n_samples,)
86          The labels used to compute the clustering metrics which requires some
87          supervision.
88      bool : bool
89          Indica si se utilizan todas las muestras (True) o no (False)
90      """
91      t0 = time()
92      estimator = make_pipeline(StandardScaler(), kmeans).fit(data)
93      fit_time = time() - t0
94      results = [name, fit_time, 'NAN']
95
96      # Define the metrics which require only the true labels and estimator
97      # labels
```

```
98  clustering_metrics = [
99      metrics.homogeneity_score,
100     metrics.completeness_score,
101     metrics.v_measure_score,
102     metrics.adjusted_rand_score,
103     metrics.adjusted_mutual_info_score,
104 ]
105
106 #Estos parametros son usados si bool es True y si es False:
107 clusterizacion = estimator[-1].labels_ #estimacion inicial
108 new_clustering = [] #Aca guardamos los labels de los datos finales, los labels obtenidos
109                     #dependeran de si bool es True o False
110 #--------------------------------------------------------------------------
111 #Estos otros parametros son usados solo si bool es False, para el caso
112 new_labels = [] #Aca se guardaran los labels correctos
113 new_data = pd.DataFrame() #Aca se guardaran los datos que obtuvieron un label distinto a -1
114 #--------------------------------------------------------------------------
115 #Si bool es true significa que usare todos los datos, incluyendo aquellos
116 #datos que no tienen un cluster asociado (y que por tanto estan en el
117 #cluster -1)
118 if bool == True:
119   maximo = max(clusterizacion)
120   new_labels = labels
121   new_data = data
122   for i in range(len(clusterizacion)):
123     if clusterizacion[i] == -1: #Los datos no asociados a un cluster se consideran
124                         # en un cluster propio
125       new_clustering.append(maximo+1)
126     else:
127       new_clustering.append(clusterizacion[i])
128
129 #Si bool es false significa que solo considerare los datos que tienen
130 #un cluster asociado distinto de -1
131 elif bool == False:
132   for i in range(len(clusterizacion)):
133     if clusterizacion[i] != -1: #si el elemento si esta asociado a un cluster
134       new_clustering.append(clusterizacion[i]) #Guardamos la clusterizacion del elemento
135       new_labels.append(labels[i]) #Guardamos la correcta clasificacion del elemento
136       new_data = pd.concat([new_data, data.iloc[i]], axis=1)
137     #Los elemenotos con cluster -1 no se consideran
138   new_data = new_data.transpose() #Para que las caracteristicas queden como columnas
139
140 #Cantidad de clusters
141 nro_clusters = len(set(new_clustering))
142
143 try:
144   if nro_clusters <= 1: #Solo hay 1 cluster o no hay ninguno
145     #Obtenemos las metricas
146     results += [m(new_labels, new_clustering) for m in clustering_metrics]
147     results +=['NAN'] #No hay silhuette
148     results += [nro_clusters] #ponemos la cantidad de clusters obtenidos a los resultados
```

```
149       # Show the results
150       formatter_result = ("{:9s}\t{:.3f}\t{:3s}\t{:.3f}\t{:.3f}"
151                   "\t{:.3f}\t{:.3f}\t{:.3f}\t{:3s}\t\t{:0}")
152       print(formatter_result.format(*results))
153
154     elif nro_clusters > 1: #Hay mas de 1 cluster
155       #Obtenemos las metricas
156       results += [m(new_labels, new_clustering) for m in clustering_metrics]
157       #The silhouette score requires the full dataset
158       results += [
159         metrics.silhouette_score(data, estimator[-1].labels_,
160                                  metric="euclidean", sample_size=300,)
161       ]
162       results += [nro_clusters] #ponemos la cantidad de clusters obtenidos a los resultados
163       # Show the results
164       formatter_result = ("{:9s}\t{:.3f}\t{:3s}\t{:.3f}\t{:.3f}"
165                     "\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t\t{:0}")
166       print(formatter_result.format(*results))
167
168     except:
169       print("Para el caso "+str(name)+" hubo un error al calcular las metricas")
170
171 #------
172
173 #Pruebas:
174
175 print(98 * '_')
176 print('init\t\ttime\tinertia\thomo\tcompl\tv-meas\tARI\tAMI\tsilhouette\tn_clusters')
177 n_clusters_pred = 10 #Cantidad de clusters a usar en KMeans y Aglomerative
178 #Kmeans con inicializacion al azar
179 kmeans = KMeans(init="random", n_clusters=n_clusters_pred, n_init=4,
180             random_state=0)
181 bench_k_means(kmeans=kmeans, name="random", data=MFCCs, labels=Species)
182
183 #Kmeans++
184 kmeans = KMeans(init="k-means++", n_clusters=n_clusters_pred, n_init=4,
185             random_state=0)
186 bench_k_means(kmeans=kmeans, name="k-means++", data=MFCCs, labels=Species)
187
188 #DBSCAN con epsilon por defecto (0.5)
189 dbscan = DBSCAN(eps=0.5).fit(MFCCs)
190 bench_clustering2(kmeans= dbscan, name='DBSCAN-0.5-f', data=MFCCs, labels=Species, bool=
    ↪ False)
191
192 #DBSCAN con epsilon 0.7
193 dbscan = DBSCAN(eps=0.7).fit(MFCCs)
194 bench_clustering2(kmeans= dbscan, name='DBSCAN-0.7-f', data=MFCCs, labels=Species, bool=
    ↪ False)
195
196 #DBSCAN con epsilon 0.2
197 dbscan = DBSCAN(eps=0.2).fit(MFCCs)
```

```
198 bench_clustering2(kmeans= dbscan, name='DBSCAN-0.2-f', data=MFCCs, labels=Species, bool=
        ↪ False)
199
200 #DBSCAN con epsilon por defecto (0.5), agregando outliers a cluster extra
201 dbscan = DBSCAN(eps=0.5).fit(MFCCs)
202 bench_clustering2(kmeans= dbscan, name='DBSCAN-0.5-t', data=MFCCs, labels=Species, bool=
        ↪ True)
203
204 #DBSCAN con epsilon 0.7, agregando outliers a cluster extra
205 dbscan = DBSCAN(eps=0.7).fit(MFCCs)
206 bench_clustering2(kmeans= dbscan, name='DBSCAN-0.7-t', data=MFCCs, labels=Species, bool=
        ↪ True)
207
208 #DBSCAN con epsilon 0.2, agregando outliers a cluster extra
209 dbscan = DBSCAN(eps=0.2).fit(MFCCs)
210 bench_clustering2(kmeans= dbscan, name='DBSCAN-0.2-t', data=MFCCs, labels=Species, bool=
        ↪ True)
211
212 #Clustering aglomerativo
213 aglomerative = AgglomerativeClustering(n_clusters=n_clusters_pred)
214 bench_clustering2(kmeans=aglomerative, name='Aglomerative', data=MFCCs, labels=Species, bool
        ↪ =True)
215 print(98 * '_')
216 #------------------------------------------------------------------------------
217 #Aplicando PCA
218 MFCCs_red = PCA(n_components=2).fit_transform(MFCCs)
219 MFCCs_red_df = pd.DataFrame(MFCCs_red)
220
221 print("")
222 print("Al aplicar PCA se obtienen los siguientes resultados:")
223 print(98 * '_')
224 print('init\t\ttime\tinertia\thomo\tcompl\tv-meas\tARI\tAMI\tsilhouette\tn_clusters')
225 #Kmeans con inicializacion al azar
226 kmeans = KMeans(init="random", n_clusters=n_clusters_pred, n_init=10)
227 bench_k_means(kmeans=kmeans, name="random", data=MFCCs_red_df, labels=Species)
228
229 #Kmeans++
230 kmeans = KMeans(init="k-means++", n_clusters=n_clusters_pred, n_init=10)
231 bench_k_means(kmeans=kmeans, name="k-means++", data=MFCCs_red_df, labels=Species)
232
233 #DBSCAN con epsilon por defecto (0.5)
234 dbscan = DBSCAN(eps=0.5).fit(MFCCs)
235 bench_clustering2(kmeans= dbscan, name='DBSCAN-0.5-f', data=MFCCs_red_df, labels=Species,
        ↪ bool=False)
236
237 #DBSCAN con epsilon 0.7
238 dbscan = DBSCAN(eps=0.7).fit(MFCCs)
239 bench_clustering2(kmeans= dbscan, name='DBSCAN-0.7-f', data=MFCCs_red_df, labels=Species,
        ↪ bool=False)
240
241 #DBSCAN con epsilon 0.2
```

```
242 dbscan = DBSCAN(eps=0.2).fit(MFCCs)
243 bench_clustering2(kmeans= dbscan, name='DBSCAN-0.2-f', data=MFCCs_red_df, labels=Species,
        ↪ bool=False)
244
245 #DBSCAN con epsilon por defecto (0.5), agregando outliers a cluster extra
246 dbscan = DBSCAN(eps=0.5).fit(MFCCs)
247 bench_clustering2(kmeans= dbscan, name='DBSCAN-0.5-t', data=MFCCs_red_df, labels=Species,
        ↪ bool=True)
248
249 #DBSCAN con epsilon 0.7, agregando outliers a cluster extra
250 dbscan = DBSCAN(eps=0.7).fit(MFCCs)
251 bench_clustering2(kmeans= dbscan, name='DBSCAN-0.7-t', data=MFCCs_red_df, labels=Species,
        ↪ bool=True)
252
253 #DBSCAN con epsilon 0.2, agregando outliers a cluster extra
254 dbscan = DBSCAN(eps=0.2).fit(MFCCs)
255 bench_clustering2(kmeans= dbscan, name='DBSCAN-0.2-t', data=MFCCs_red_df, labels=Species,
        ↪ bool=True)
256
257 #Clustering aglomerativo
258 aglomerative = AgglomerativeClustering(n_clusters=n_clusters_pred)
259 bench_clustering2(kmeans=aglomerative, name='Aglomerative', data=MFCCs_red_df, labels=
        ↪ Species, bool=True)
260
261 print(98 * '_')
```