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

Tarea 4

Clustering

Integrantes: Lukas Pavez
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla
Ayudantes: Francisco Leiva C.
Gabriel Andrés Azócar Cárcamo
Giovanni Pais L.
Nicolas Cruz Brunet

Fecha de entrega: 12 de Junio de 2019
Santiago, Chile

Índice de Contenidos

Índice de Figuras

1. Introducción

Se tiene como objetivo utilizar distintos tipos de algoritmos de clustering y analizar su desempeño. Para esto se utilizará una base de datos con llamadas de 10 especies distintas de ranas, las cuales se representan en 22 características.

Para lograr esto se leerá la base de datos y se utilizarán los algoritmos de clustering kmeans, cambiando su método de inicialización, DBSCAN, cambiando su valor de épsilon y si se cuenta o no el ruido como cluster y clustering aglomerativo, luego de esto se utilizara PCA para disminuir la dimensionalidad de la base de datos (de 22 dimensiones a 2 dimensiones), y se volverá a analizar los mismos algoritmos.

2. Marco Teórico

Las características de la base de datos representan coeficientes cepstrales en frecuencias de Mel, estos coeficientes son la representación en características del contenido relevante de una señal de audio, son ampliamente utilizados en speech recognition.

Uno de los algoritmos a utilizar en la tarea es k-means, que es un algoritmo que se basa en particiones de los datos , este método separa todas las observaciones en k clusters, esto lo hace ubicando k *centroides* según un método de inicialización, y cada observación pertenece al cluster formado por el centroide más cercano, luego, se calcula el promedio de cada cluster, donde los promedios pasan a ser el nuevo centroide, y se repite el proceso hasta que los centroides se mantienen en un valor. Uno de los métodos de inicialización es llamada *kmeans++*, éste método básicamente localiza cada centroide lejos de los otros. También se utiliza el método de inicialización *random*, que como su nombre lo dice, ubica los k centroides de manera aleatoria.

Otro algoritmo que se utiliza es el DBSCAN, que es un algoritmo que genera clusters a partir de la densidad de los datos. En el algoritmo los puntos se dividen en tres tipos: core, borde y outlier/ruido. Un punto core es un punto que tiene al menos una cantidad predefinida de puntos dentro de un radio *épsilon* (por lo general 5 puntos). Los puntos borde son los que no son puntos core, y están dentro del radio de un punto core. El resto de los puntos son los outliers o ruido, ya que no están cercanos a nada. Luego de clasificar todos los puntos, los conjuntos de puntos core representan los clusters.

Finalmente, el último algoritmo utilizado es clustering aglomerativo, éste es un método jerárquico, donde se parte con que cada punto es un cluster por si solo, y luego se van combinando clusters en uno solo, hasta quedar con un número dado de clusters, o bien se puede definir una distancia mínima para que los clusters se puedan combinar en uno.

Para medir el desempeño de cada algoritmo, se utilizarán 4 métricas, : homogeneity, completeness, v-measure y silhouette, donde las 3 necesitan el ground truth o clase correcta de los datos. La métrica homogeneity mide que cada cluster contenga puntos de solo una clase. Completeness mide que todos los miembros de una clase se encuentran en el mismo cluster. V-measure es un promedio entre homogeneity y completeness dado por la Equación 1, el valor de β por default es 1.

$$\text{v-measure} = \frac{(1+\beta)\cdot \text{homogeneity} \cdot \text{completeness}}{(\beta \cdot \text{homogeneity} + \text{completeness})} \tag{1}$$

Finalmente, la métrica silhouette se utiliza sin el ground truth de los datos, se calcula con la Ecuación 2 y se calcula con 2 valores: (a) la distancia de una muestra y todas los valores en el mismo cluster, y (b)la distancia promedio entre una muestra y los puntos del próximo cluster más cercano.

$$\text{silhouette} = \frac{b-a}{max(b,a)} \tag{2}$$

3. Desarrollo

3.1. Lectura del dataset

Se utilizó la función read_csv() de pandas, luego se crearon las matrices que representan a las características (features) y a las etiquetas (species), como las etiquetas están con un string con el nombre de la clase y se necesita que estén representadas con un número, se creó un diccionario (labels_unique) donde las llaves son los nombres de las etiquetas, y los valores son un número entre 0 y 9, diferente para cada etiqueta. Luego se creó la lista labels con las etiquetas (en número) para cada muestra del dataset. El código es el siguiente:

```
if __name__ == "__main__":
    np.random.seed(42)
    df = pd.read_csv("Frogs_MFCCs.csv")
    species = df["Species"]
    features = df.drop(["RecordID", "Family", "Genus", "Species"], 1)
    feature_names = features.columns.tolist()
    labels_unique = {}
    species_unique = species.unique()
    for i in range(len(species_unique)):
        labels_unique[species_unique[i]] = i

    labels = []
    for specie in species:
        labels.append(labels_unique[specie])

    ...
```

3.2. Modificación bench_k_means()

Se tomó la función bench_k_means() del código de ejemplo y se cambió la tabla que se imprimía en la pantalla, borrándole las columnas inertia, ARI y AMI, inertia se borró ya que los otros algoritmos no tienen la variable, y los otros se borraron ya que no se pedían esas métricas. También se le agregó una columna a la tabla que representa el número de clusters. El código es el siguiente:

```
def bench_k_means(estimator, name, data, true_labels):
    n_clusters = estimator.get_params()["n_clusters"]
    t0 = time()
    estimator.fit(data)
    labels = estimator.labels_
    homo, compl, vmeas = metrics.homogeneity_completeness_v_measure(true_labels, labels)
    silhouette = metrics.silhouette_score(data, labels, metric='euclidean')
    print("{:20}\t{:<10}\t{:<10}\t{:<10}\t{:<10}\t{:<10}".format(
        name,
        round(time() - t0, 3),
        round(homo, 3),
        round(compl, 3),
        round(vmeas, 3),
        round(silhouette, 3),
        n_clusters))
```

3.3. Creación bench_DBSCAN()

Esta función recibe el épsilon del algoritmo, la matriz de características, las etiquetas correctas y un booleano extra_cluster, donde True significa que se considerará el ruido como un cluster extra, y False lo contrario. En la función, luego de calcular los clusters, si el ruido se considera como un cluster extra, se toman las etiquetas estimadas y las que son -1 (ruido) se cambian por otro número, que es el número de clusters + 1. Si el ruido no se considera como cluster extra, se busca en qué índices la lista de etiquetas estimadas tiene valores -1, luego se borran las filas con esos índices de la matriz de las características, de las etiquetas verdaderas y de las etiquetas estimadas, así ya no se tienen las muestras asociadas al ruido. Después de filtrar las muestras según si se usa el ruido como cluster o no, se cuenta el número de cluster resultante, si es menor o igual a 1, entonces se setean los valores de las métricas como un string con guiones (—-), que indica que no se pudo calcular. En caso contrario, se calculan las métricas. Finalmente se imprime en pantalla una fila de la tabla con la información del algoritmo, en el nombre del algoritmo que aparece en la tabla se muestra una combinación entre *DBSCAN*, el valor del épsilon dado y el booleano extra_cluster, todo separado con guión bajo. El código es el siguiente:

```
def bench_DBSCAN(epsilon, data, true_labels, extra_cluster):
    data_ = data.copy()
    metric_labels = true_labels[:]
    estimator = DBSCAN(eps=epsilon)
    t0 = time()
    estimator.fit(data_)
    labels = estimator.labels_
    clusters = set(labels)
    n_clusters_ = len(clusters)
    if not extra_cluster:
        est_labels = estimator.labels_
        indexs = []
        for i in range(len(est_labels)):
            if est_labels[i] == -1:
                indexs.append(i)

        for i in indexs[::-1]:
            labels = np.delete(labels, i, axis=0)
            metric_labels.pop(i)
            data_.drop(data_.index[i], inplace=True)
        if -1 in clusters:
            n_clusters_ -= 1
    else:
        last = n_clusters_
        for i in range(len(labels)):
            if labels[i] == -1:
                labels[i] = last

    if n_clusters_ <= 1:
        homo = "----"
        compl = "----"
        vmeas = "----"
        silhouette = "----"
    else:
        homo, compl, vmeas = metrics.homogeneity_completeness_v_measure(metric_labels, labels)
        homo = round(homo, 3)
        compl = round(compl, 3)
        vmeas = round(vmeas, 3)
        silhouette = round(metrics.silhouette_score(data_, labels, metric='euclidean'), 3)
    print("{:20}\t{:<10}\t{:<10}\t{:<10}\t{:<10}\t{:<10}".format("DBSCAN_" + str(epsilon)
     ↪  + "_" + str(extra_cluster),
                                                              round(time() - t0, 3),
                                                              homo,
                                                              compl,
                                                              vmeas,
                                                              silhouette,
                                                              n_clusters_))
```

3.4. Creación bench_agglomerative_clustering

Para esta función se copió la función bench_k_means() y se le agregó que si se encontraba sólo 1 cluster no se calcularan las métricas, el nombre mostrado en la tabla es *AGGC*. El código es el siguiente:

```
def bench_agglomerative_clustering(estimator, data, true_labels):
    t0 = time()
    estimator.fit(data)
    labels = estimator.labels_
    clusters = set(labels)
    n_clusters_ = len(clusters)

    if n_clusters_ <= 1:
        homo = "----"
        compl = "----"
        vmeas = "----"
        silhouette = "----"
    else:
        homo, compl, vmeas = metrics.homogeneity_completeness_v_measure(true_labels, labels)
        homo = round(homo, 3)
        compl = round(compl, 3)
        vmeas = round(vmeas, 3)
        silhouette = round(metrics.silhouette_score(data, labels, metric='euclidean'), 3)
    print("{:20}\t{:<10}\t{:<10}\t{:<10}\t{:<10}\t{:<10}\t{:<10}".format("AGGC",
                                                        round(time() - t0, 3),
                                                        homo,
                                                        compl,
                                                        vmeas,
                                                        silhouette,
                                                        n_clusters_))
```

3.5. Pruebas

Primero se escalaron los datos con la función scale(), luego se realizaron 9 pruebas con los algoritmos, 2 con kmeans, variando el método de inicialización (random y kmeans++), 6 con DBSCAN, donde las primeras 3 no consideran al ruido como cluster y las últimas 3 si, y dentro de ambos grupos se varía el épsilon, tomando los valores 0.5 (default), 0.2 y 0.7, y la última prueba se utilizó agglomerative clustering, donde se le dio el número de clusters al que debe llegar (10). Se obtuvieron los resultados que se ven en la Figura 1.

n_clusters: 10, n_samples 7195, n_features 22

init	time	homo	compl	v-meas	silhouette	n_clusters
KMeans_random	1.615	0.728	0.572	0.64	0.245	10
KMeans_k-means++	1.782	0.68	0.554	0.611	0.254	10
DBSCAN_0.5_False	5.925	1.0	0.327	0.493	0.234	8
DBSCAN_0.7_False	5.707	1.0	0.585	0.738	0.511	13
DBSCAN_0.2_False	5.68	1.0	0.0	0.0	0.517	2
DBSCAN_0.5_True	1.813	0.074	0.451	0.127	-0.227	9
DBSCAN_0.7_True	1.862	0.179	0.545	0.27	-0.333	14
DBSCAN_0.2_True	1.546	0.011	0.677	0.022	-0.113	3
AGGC	3.467	0.789	0.661	0.719	0.276	10

Figura 1: Resultados 9 pruebas

Luego se redujo la dimensionalidad de los datos a 2 dimensiones utilizando PCA, y se realizaron las mismas pruebas, obteniendo los siguientes resultados (Figura 2):

PCA

n_clusters: 10, n_samples 7195, n_features 2

init	time	homo	compl	v-meas	silhouette	n_clusters
random	1.261	0.655	0.506	0.571	0.408	10
k-means++	1.302	0.634	0.497	0.557	0.407	10
DBSCAN_0.5_False	1.316	0.002	0.306	0.004	0.12	3
DBSCAN_0.7_False	0.178	----	----	----	----	1
DBSCAN_0.2_False	1.449	0.536	0.824	0.649	0.069	23
DBSCAN_0.5_True	1.35	0.005	0.159	0.009	0.116	4
DBSCAN_0.7_True	1.353	0.003	0.157	0.005	0.454	2
DBSCAN_0.2_True	1.253	0.512	0.708	0.594	0.026	24
AGGC	2.601	0.651	0.516	0.576	0.356	10

Figura 2: Resultados 9 pruebas después de aplicar PCA

El código de las pruebas es el siguiente:

```
...

n_samples, n_features = features.shape
n_clusters = len(np.unique(labels))

features = scale(features)
features = pd.DataFrame(data=features, columns=feature_names)

print(100 * '_')
print("n_clusters: %d, \t n_samples %d, \t n_features %d"
      % (n_clusters, n_samples, n_features))
print(100 * '_')
print("{:20}\t{:10}\t{:10}\t{:10}\t{:10}\t{:10}\t{:10}".format("init", "time", "homo", "compl", "v
 ↪ -meas", "silhouette",
                                                "n_clusters"))

```

```
16  bench_k_means(KMeans(init='random', n_clusters=10), "KMeans_random", features, labels)
17  bench_k_means(KMeans(init='k-means++', n_clusters=10), "KMeans_k-means++", features,
     ↪ labels)
18  bench_DBSCAN(0.5, features, labels, False)
19  bench_DBSCAN(0.7, features, labels, False)
20  bench_DBSCAN(0.2, features, labels, False)
21  bench_DBSCAN(0.5, features, labels, True)
22  bench_DBSCAN(0.7, features, labels, True)
23  bench_DBSCAN(0.2, features, labels, True)
24  bench_agglomerative_clustering(AgglomerativeClustering(n_clusters=10), features, labels)
25
26  print(100 * '_')
27
28  pca = PCA(n_components=2)
29  pca_features = pca.fit_transform(features)
30
31  pca_features = pd.DataFrame(data=pca_features, columns=['col1', 'col2'])
32  print(48 * " " + "PCA")
33  print(100*"_")
34  n_samples, n_features = pca_features.shape
35  print("n_clusters: %d, \t n_samples %d, \t n_features %d"
36        % (n_clusters, n_samples, n_features))
37  print(100 * '_')
38  print("{:20}\t{:10}\t{:10}\t{:10}\t{:10}\t{:10}\t{:10}".format("init", "time", "homo", "compl", "v
     ↪ -meas",
39                                                          "silhouette",
40                                                          "n_clusters"))
41
42  bench_k_means(KMeans(init='random', n_clusters=10), "random", pca_features, labels)
43  bench_k_means(KMeans(init='k-means++', n_clusters=10), "k-means++", pca_features, labels)
44  bench_DBSCAN(0.5, pca_features, labels, False)
45  bench_DBSCAN(0.7, pca_features, labels, False)
46  bench_DBSCAN(0.2, pca_features, labels, False)
47  bench_DBSCAN(0.5, pca_features, labels, True)
48  bench_DBSCAN(0.7, pca_features, labels, True)
49  bench_DBSCAN(0.2, pca_features, labels, True)
50  bench_agglomerative_clustering(AgglomerativeClustering(n_clusters=10), pca_features, labels)
51
52  print(100 * '_')
```

3.6. Análisis de resultados

Partiendo con la métrica homogeneity, las pruebas que obtuvieron mayor valor fueron las 3 pruebas de DBSCAN sin contar al ruido como cluster y antes de aplicar PCA en los datos, donde los 3 obtuvieron valor 1, lo que indica que todos los clusters formados tienen solo muestras de una clase. Mientras que el que tuvo peor valor fue DBSCAN con épsilon 0.5 sin contar al ruido como cluster, después de aplicar PCA, donde se obtuvo un valor de 0.3, lo que indica que los clusters formados tenían muchas muestras de distintas clases.

En el caso de completeness, la prueba con mejores resultados fue DBSCAN con épsilon 0.2, sin considerar al ruido como cluster y después de aplicar PCA, donde se obtuvo un valor de 0.824, lo que indica que la mayoría de los elementos de una clase se encontraban en el mismo cluster. El peor caso fue DBSCAN con épsilon 0.2 sin contar al ruido como cluster, antes de aplicar PCA, donde se obtuvo un valor de 0.

Con la métrica v-measure, la mejor prueba fue DBSCAN con épsilon 0.7 sin contar al ruido como cluster, antes de aplicar PCA, donde se obtuvo un valor de 0.738, lo que indica que se obtuvieron altos valores de homogeneity y completeness, ya que esta métrica es el promedio, por lo que se puede decir que los datos fueron bien asignados a cada cluster, también cabe mencionar que el método de clustering aglomerativo obtuvo un valor cercano, igual a 0.719. El peor caso fue DBSCAN con épsilon 0.2 sin contar al ruido como cluster, antes de aplicar PCA, esto es porque se obtuvo 0 en completeness, y esta métrica se calcula con un promedio armónico entre homogeneity y completeness.

En el caso de silhouette, la mejor prueba fue DBSCAN con épsilon 0.2 sin contar al ruido como cluster, antes de aplicar PCA, con 0.517, seguida de cerca por DBSCAN con épsilon 0.7 sin contar al ruido como cluster, antes de aplicar PCA, con 0.512, sin embargo, el valor no es tan alto, por lo que se puede decir que la distancia entre los puntos de un mismo cluster es medianamente alta, o que también que la separación entre los clusters es baja. La peor prueba fue DBSCAN con épsilon 0.7, contando al ruido como cluster, antes de aplicar PCA, que obtuvo -0.333, lo que indica que los clusters en vez de estar separados, estaban juntos ya que las distancias de un punto de un cluster con los de otro eran menores que las distancias de un punto de un cluster con los puntos del mismo cluster, esto se puede deber a que el ruido forma un cluster muy grande que contiene a los otros.

En las 12 pruebas con DBSCAN, solo hay 1 en la que falla, que es con épsilon 0.7 sin contar al ruido como cluster y después de aplicar PCA, esto se puede deber a que el valor de épsilon fue demasiado grande, ya que se obtuvo sólo 1 cluster, lo que indica que la distancia entre los puntos era pequeña.

Con respecto al número de clusters obtenido por DBSCAN, antes de aplicar PCA varía entre 2 y 14, donde se considera que la mejor prueba fue DBSCAN con épsilon 0.7 sin contar al ruido como cluster, donde por su valor de silhouette, los clusters formados (13) estaban medianamente separados y algo compactos, y el valor es cercano al valor real de clases (10). En los casos de que se acepta el ruido como cluster se llego a la conclusión de que el ruido hace que el valor de silhouette baje hasta ser negativo, y en general los valores de las métricas fueron bajos, con la excepción de completeness cuando se usó épsilon 0.2, que tuvo un valor alto (0.677).

Después de aplicar PCA, ninguna prueba se acerco al número correcto de clusters, la más cercana encontró 4, lo que puede significar que al reducir la dimensionalidad hubo pérdida de información importante para la clasificación, y en general las métricas fueron bajas, con la excepción cuando se usó épsilon 0.2 sin contar al ruido como cluster y también al contarlo, pero esto se puede deber a que se encontró un número muy alto de clusters (23 y 24 respectivamente), por lo que habían clusters pequeños con los puntos que les pertenecían.

Al considerar el efecto de agregar el ruido como clase extra, en el caso antes de aplicar PCA a los datos, se considera que afecto negativamente a las métricas, ya que en este caso el ruido podía estar muy disperso, lo que hace que el cluster ruido contenga a los otros clusters. En el caso de homogeneity, al agregar el ruido los valores bajaron de 1 a valores cercanos a 0, mientras que con completeness, los valores aumentaron, esto puede deberse a que todos los valores de ruido estaban dentro de su cluster.
Después de aplicar PCA, la métrica silhouette dejó de ser negativa, por lo que se piensa que se hubo menos ruido, aún así se obtuvieron valores bajos. Con respecto a homogeneity y completeness, todos los valores disminuyeron.
Por lo que se puede decir que agregar el ruido afecto de forma negativa a las métricas en todos los casos.

4. Conclusiones

Luego de realizar todas las pruebas, se concluye que la mejor es DBSCAN con épsilon 0.7 sin contar al ruido como cluster, antes de aplicar PCA, ya que, por los valores de sus métricas, se ve que se formaron clusters separados y compactos, además de estar cerca del número correcto de clusters.
También se pudo ver como afecta a las métricas el agregar el ruido como cluster extra, y que es mejor evitar hacerlo.

Si bien el problema se puede resolver bien con k-means, éste método solo funciona bien cuando se sabe el número de clusters al que se quiere llegar, mientras que con DBSCAN se obtuvieron diversos resultados, pero también ocurre lo mismo, al no saber el número correcto de clusters habría que basarse sólo en las métricas para encontrar la separación de los datos, aunque en este caso si no se supiera el número de clusters se habría elegido el con mejores métricas, que llego a un valor cercano. En el caso de clustering agglomerativo se obtuvieron buenos resultados, pero a éste método también se le daba el número de clusters.