Tarea 2 de Minería de Textos: Clustering

Sergio Soler Rocha
Universidad Nacional de Educación a Distancia

1. Introducción

El objetivo de la tarea es aplicar algoritmos de clustering a un conjunto de mensajes clasificados por temática y comparar los resultados obtenidos con las agrupaciones ya clasificadas. Los datos son un subconjunto de la colección 20_newsgroup del CMU Text Learning Group. El subconjunto está compuesto por 7 grupos temáticos, cada uno en una carpeta, que contiene una serie de archivos con comentarios sobre noticias.

2. Preparación y preprocesamiento de la colección de textos

En total contamos con 7 carpetas de cuyos nombres ya se puede deducir la temática: comp.sys .ibm.pc.hardware, comp.sys.mac.hardware, rec.autos, rec.sport.hockey, sci.electronics, talk.politics .guns, talk.politics.mideast. La primera carpeta aborda la temática del hardware de IBM, con un total de 124 archivos; la segunda trata sobre el hardware de Mac, con 146 archivos. La tercera carpeta está relacionada con automóviles y contiene 61 archivos; la cuarta, sobre el deporte de hockey, cuenta con 50 archivos. La quinta carpeta aborda temas de electrónica y contiene 211 archivos, mientras que la sexta, sobre armas en un contexto político, tiene 141 archivos. Finalmente, la séptima carpeta trata la política en Oriente Medio y cuenta con 72 archivos.

Cada archivo cuenta con una cabecera de la cual se puede inducir la temática, por lo que nos desharemos de ella sabiendo que siempre entre la cabecera y el texto hay una linea de margen. El código es el siguiente:

```python
import os

# Directorios que contienen los archivos
carpetas = [
    'comp.sys.ibm.pc.hardware',
    'comp.sys.mac.hardware',
    'rec.autos',
    'rec.sport.hockey',
    'sci.electronics',
    'talk.politics.guns',
    'talk.politics.mideast'
]

# Función para limpiar los mensajes
def limpiar_mensaje(ruta):
    with open(ruta, 'r', encoding='latin-1') as archivo:
        contenido = archivo.read()
        # Encontramos la posición del primer espacio en blanco
        indice = contenido.find('\n\n')
        if indice != -1:
            # Eliminamos la cabecera del mensaje
            mensaje_limpio = contenido[indice + 2:]  # +2 para omitir el espacio en blanco
            return mensaje_limpio
        else:
            return contenido  # No se encontró una cabecera

# Procesamiento de los archivos
for carpeta in carpetas:
    ruta_carpeta = f'Corpus-Clustering/{carpeta}'
    for archivo_nombre in os.listdir(ruta_carpeta):
        ruta_archivo = os.path.join(ruta_carpeta, archivo_nombre)
        mensaje_limpio = limpiar_mensaje(ruta_archivo)
        # Guardamos el mensaje limpio de vuelta al archivo
        with open(ruta_archivo, 'w', encoding='latin-1') as archivo:
            archivo.write(mensaje_limpio)
```

Figura 1: Quitar la cabecera

Una vez que nos hemos deshecho de la cabecera pasamos a la tokenización y lematización del texto resultante. La tokenización fragmenta el texto en unidades significativas (palabras), mientras que la lematización normaliza estas unidades para facilitar su análisis y comprensión, disminuyendo la complejidad y permitiendo la comparación más efectiva entre ellas. Para eso usaremos la librería spacy de la siguiente manera:

```python
import os
import spacy

# Cargamos el modelo de idioma en inglés
nlp = spacy.load('en_core_web_lg')

# Directorios que contienen los archivos
carpetas = [
    'comp.sys.ibm.pc.hardware',
    'comp.sys.mac.hardware',
    'rec.autos',
    'rec.sport.hockey',
    'sci.electronics',
    'talk.politics.guns',
    'talk.politics.mideast'
]

# Procesamiento de los archivos
for carpeta in carpetas:
    ruta_carpeta = f'Corpus-Clustering/{carpeta}'
    for archivo_nombre in os.listdir(ruta_carpeta):
        ruta_archivo = os.path.join(ruta_carpeta, archivo_nombre)

        with open(ruta_archivo, 'r', encoding='latin-1') as archivo:
            contenido = archivo.read()
            doc = nlp(contenido)
            tokens_procesados = [token.lemma_ for token in doc if token.is_alpha and not token.is_stop]

        # Guardamos los tokens procesados en un nuevo archivo
        ruta_tokens = f'Tokenizado/{carpeta}/{archivo_nombre}_tokens'
        with open(ruta_tokens, 'w', encoding='utf-8') as archivo_tokens:
            archivo_tokens.write(' '.join(tokens_procesados))
```

Figura 2: Tokenización y Lematización

Ahora pasamos a hacer un recuento de palabras por cada carpeta, es decir, por cada grupo, para Calcular la media y la desviación estándar por documento. Los cálculos aparecen en la siguiente tabla:

Carpeta	Número medio de palabras	Desviación estándar
comp.sys.ibm.pc.hardware	92.85	197.99
comp.sys.mac.hardware	62.12	45.00
rec.autos	75.57	66.52
rec.sport.hockey	106.54	111.82
sci.electronics	100.30	383.32
talk.politics.guns	161.95	225.77
talk.politics.mideast	298.94	504.04
Promedio general	**119.31**	**286.07**

Cuadro 1: Información sobre el número medio de palabras y desviación estándar por carpeta

Por lo general, observamos que la desviación estándar es alta en comparación con la media. Esto indica que, en general, los documentos dentro de cada carpeta tienen tamaños muy diferentes en cuanto a número de palabras.

Una vez que hemos procesado las palabras, es importante establecer cómo vamos a crear y estructurar el 'gold standard'. En este caso, usaremos el nombre de cada carpeta como referencia, asignándole un número del 0 al 6 según su contenido y la cantidad de archivos que contiene. Crearemos un vector de tamaño 805, que representa el número total de documentos. En las primeras 124 posiciones del vector, asignaremos un número que represente al grupo 'comp.sys.ibm.pc.hardware'; en las siguientes 146, asignaremos otro número que corresponderá a 'comp.sys.mac.hardware', y así sucesivamente hasta completar el vector. El orden seguirá el mismo que se muestra en el Cuadro 2, de arriba hacia abajo. Es importante destacar que el número asignado a cada grupo será determinado por el algoritmo de clustering de manera aleatoria, así que nos veremos obligados a permutarlos según el resultado para que concuerden con nuestra elección, como veremos más adelante.

Carpeta	Grupo
comp.sys.ibm.pc.hardware	Grupo 0
comp.sys.mac.hardware	Grupo 1
rec.autos	Grupo 2
rec.sport.hockey	Grupo 3
sci.electronics	Grupo 4
talk.politics.guns	Grupo 5
talk.politics.mideast	Grupo 6

Cuadro 2: Asignación de carpetas a grupos

3. Representaciones

En primer lugar, utilizaremos la representación TF-IDF (Frecuencia de Términos-Inversa de Frecuencia en los Documentos), ya que es beneficiosa para el procesamiento de texto. Esta técnica asigna ponderaciones a las palabras según su importancia en un documento dentro de un corpus. Esto conlleva a reducir la relevancia de palabras comunes que aparecen en muchos documentos, pero que no contribuyen con información relevante. Al mismo tiempo, aumenta el peso de aquellas palabras específicas de un documento que no son comunes en todo el corpus, otorgándoles una mayor relevancia.

Como segunda opción usaremos HashingVectorizer, una opción útil en algunos casos debido a su capacidad para generar representaciones vectoriales de texto de manera eficiente, especialmente cuando trabajas con conjuntos de datos grandes o cuando la memoria y el tiempo de procesamiento son limitados. A diferencia de TfidfVectorizer no requiere almacenar un vocabulario completo en memoria. En lugar de asignar un identificador único a cada palabra, aplica una función hash directamente a las palabras y utiliza estas funciones hash para representar las palabras. Esto puede ser beneficioso cuando el tamaño del vocabulario es enorme y supera los límites de memoria. Al no almacenar el vocabulario completo, HashingVectorizer utiliza una cantidad fija de memoria predefinida, controlada por el parámetro 'n_features'. Esto permite trabajar con grandes conjuntos de datos sin agotar la memoria. La implementación de hashing puede ser más rápida en la etapa de vectorización en comparación con otras técnicas que requieren cálculos más complejos. tiene algunas limitaciones, como la pérdida de información sobre las palabras individuales debido al proceso de hash, lo que dificulta la interpretación directa de los vectores resultantes.

Para la implementación de TfidfVectorizer seguimos el siguiente procedimiento:

```python
import os
from sklearn.feature_extraction.text import TfidfVectorizer

# Directorios que contienen los archivos procesados
carpetas_procesadas = [
    'Tokenizado/comp.sys.ibm.pc.hardware',
    'Tokenizado/comp.sys.mac.hardware',
    'Tokenizado/rec.autos',
    'Tokenizado/rec.sport.hockey',
    'Tokenizado/sci.electronics',
    'Tokenizado/talk.politics.guns',
    'Tokenizado/talk.politics.mideast'
]

textos_procesados = []  # Se almacenarán los textos procesados de todas las carpetas

# Leemos los textos procesados de cada carpeta
for carpeta in carpetas_procesadas:
    for archivo_nombre in os.listdir(carpeta):
        ruta_archivo = os.path.join(carpeta, archivo_nombre)
        with open(ruta_archivo, 'r', encoding='utf-8') as archivo:
            contenido = archivo.read()
            textos_procesados.append(contenido)

# Inicializamos el vectorizador TF-IDF
tfidf_vectorizer = TfidfVectorizer()
# Ajustamos y transformamos los textos procesados
tfidf_matrix = tfidf_vectorizer.fit_transform(textos_procesados)

# tfidf_matrix contiene ahora la representación TF-IDF de los textos procesados
```

Figura 3: Implementación de TfidfVectorizer

Para la implementación de HashingVectorizer seguimos el siguiente procedimiento:

```python
import os
from sklearn.feature_extraction.text import HashingVectorizer

# Directorios que contienen los archivos procesados
carpetas_procesadas = [
    'Tokenizado/comp.sys.ibm.pc.hardware',
    'Tokenizado/comp.sys.mac.hardware',
    'Tokenizado/rec.autos',
    'Tokenizado/rec.sport.hockey',
    'Tokenizado/sci.electronics',
    'Tokenizado/talk.politics.guns',
    'Tokenizado/talk.politics.mideast'
]

textos_procesados = []  # Se almacenarán los textos procesados de todas las carpetas

# Leemos los textos procesados de cada carpeta
for carpeta in carpetas_procesadas:
    for archivo_nombre in os.listdir(carpeta):
        ruta_archivo = os.path.join(carpeta, archivo_nombre)
        with open(ruta_archivo, 'r', encoding='utf-8') as archivo:
            contenido = archivo.read()
            textos_procesados.append(contenido)

# Aplicamos HashingVectorizer
hash_vectorizer = HashingVectorizer(n_features=20000)
hash_matrix = hash_vectorizer.fit_transform(textos_procesados)
```

Figura 4: Implementación de HashingVectorizer

4. Algoritmos de Clustering

4.1. Kmeans

K-means es un algoritmo de clustering muy popular y ampliamente utilizado en aprendizaje no supervisado. Su objetivo principal es dividir un conjunto de datos en un número predeterminado de grupos (clusters) de manera que los puntos dentro de un mismo cluster sean similares entre sí y diferentes de los puntos en otros clusters.

El funcionamiento básico del algoritmo es el siguiente: Comienza seleccionando aleatoriamente k centroides, donde k es el número de clusters especificado. Cada punto del conjunto de datos se asigna al centroide más cercano (utilizando, por ejemplo, la distancia euclidiana). Una vez que todos los puntos están asignados a clusters, se recalcula el centroide de cada cluster tomando la media de todos los puntos asignados a ese cluster. Los pasos de asignación de puntos y de actualización de centroides se repiten alternativamente hasta que no haya cambios significativos en la asignación de puntos o se alcance un número máximo de iteraciones.

Uno de los inconvenientes que puede tener es su sensibilidad a la inicialización de los centroides: los resultados pueden variar dependiendo de cómo se inicializan. Por tanto, vamos a probar con diferentes inicializaciones y nos quedaremos con aquella que haya producido los mejores resultados. Para comparar las distintas salidas, utilizaremos el índice de Rand, una medida que oscila entre 0 y 1 y evalúa cuán similares son dos conjuntos con diferentes agrupaciones, ignorando las permutaciones. Para eso creamos la siguiente función:

```python
from sklearn.cluster import KMeans
from sklearn.metrics import adjusted_rand_score

# Función que nos proporciana la inicialización más óptima
def Best_seed_Kmeans(matrix, num_clusters):
    best_rand_index = float('-inf')
    best_random_seed = None

    for random_seed in range(200):  # Iterar sobre diferentes semillas aleatorias

        # Inicializamos el modelo K-means
        kmeans = KMeans(n_clusters=num_clusters, random_state=random_seed)

        # Ajustamos el modelo a los datos obtenidos con la Representación usada
        kmeans.fit(matrix)

        # Obtenemos las etiquetas de cluster po cada documento
        etiquetas_clusters = kmeans.labels_

        # Calculamos el índice de Rand ajustado
        rand_index = adjusted_rand_score(etiquetas_reales, etiquetas_clusters)

        # Actualizamos el mejor índice de Rand y la mejor semilla aleatoria
        if rand_index > best_rand_index:
            best_rand_index = rand_index
            best_random_seed = random_seed

    print("Mejor índice de Rand ajustado:", best_rand_index)
    print("Random seed óptima:", best_random_seed)
```

Figura 5: Función que busca la mejor inicialización

Tras este proceso obtenemos una semilla de valor 22 con el ídice de Rand de 0.38178 para el caso de TF-IDF, y una semilla de 187 con el índice de Rand de 0.28663 para HashingVectorizer.

Ahora vamos a estudiar los valores y estructura obtenida a la hora de hacer el clustering y mirar si existen similitudes con el 'gold standard'. Comenzaremos haciendo un histograma de cada uno de los subconjuntos dividiendo el vector de clustering obtenido según la proporción que se comentó anteriormente y estudiaremos si hay algún tipo de patrón que se asemeje al del *golden standard*

4.1.1. TF-IDF

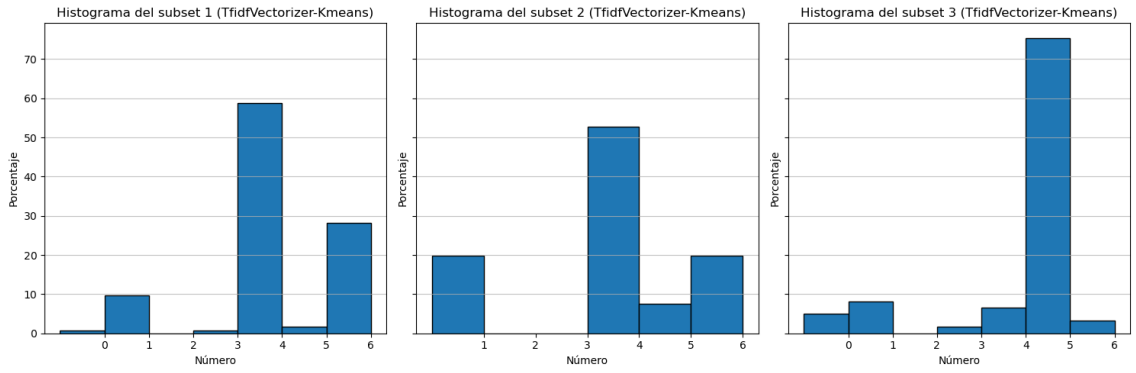

Figura 6: Histogramas de los subsets 1, 2 y 3

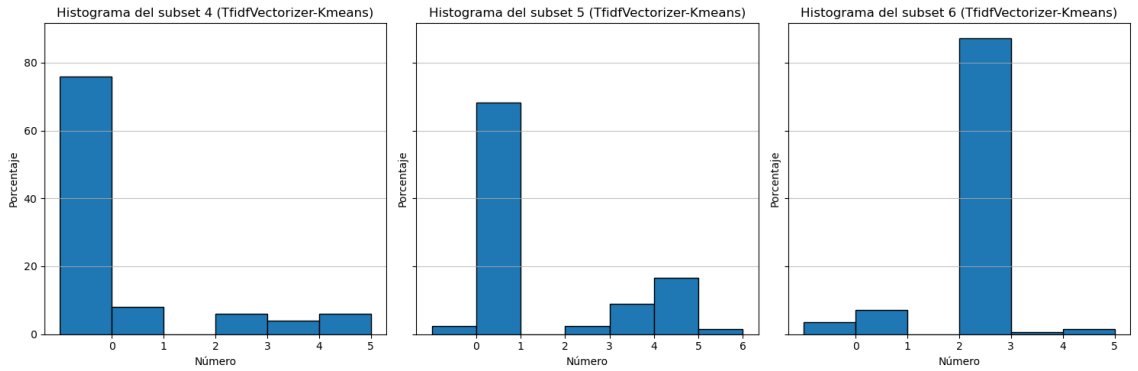

Figura 7: Histogramas de los subsets 4, 5 y 6

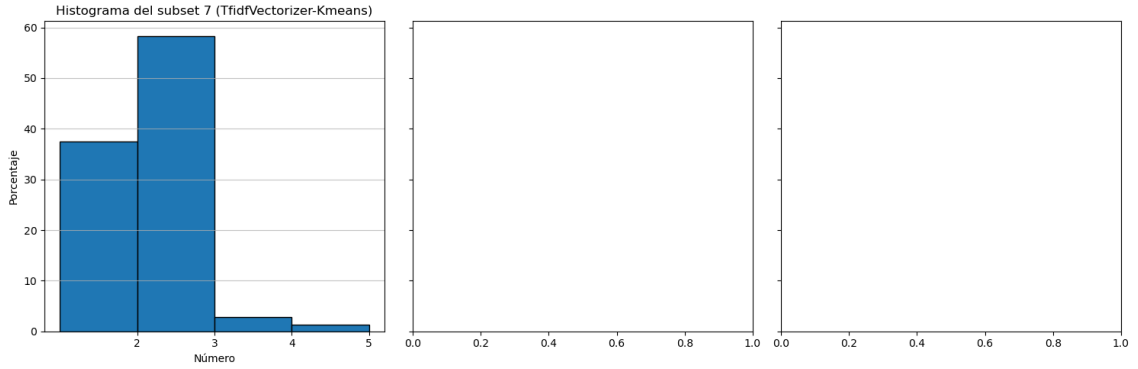

Figura 8: Histogramas del subset 7

En el primer histograma, el número que más se repite es el 4, por lo que supondremos que el número 4 representa al grupo 0. En el primer histograma, el grupo más repetido es el 4 pero como ya en siguiente lo vamos a elegir tomaremos al 6 como representante del grupo 0. En el segundo histograma, el número que más se repite es el 4, por lo que supondremos que el número 4 representa al grupo 1. En el tercer histograma, elegiremos al 5 como representante del grupo 2.

En el cuarto histograma, la elección clara es el número 0 para representar al grupo 3. En el quinto histograma, la elección recae sobre el número 1 para representar al grupo 4. En el sexto, elegimos el número 3 como representante del grupo 5. En el séptimo, vamos a elegir el 2, que aunque es la segunda opción más repetida, está muy cercana a la primera opción más repetida y es además la última opción disponible e incluso es el único grupo donde prácticamente tiene presencia.

Una vez realizadas las permutaciones correspondientes, vamos a comparar la solución obtenida por el algoritmo con respecto al *gold standard*. Calculamos las medidas de evaluación externa, las cuales son: Precisión, Cobertura y Medida-F.

Métrica	**Valor**
Precisión	0.634
Cobertura	0.609
Medida-F	0.599

Cuadro 3: Métricas obtenidas para el algoritmo TfidfVectorizer-Kmeans

La matriz de confusión obtenida es la siguiente:

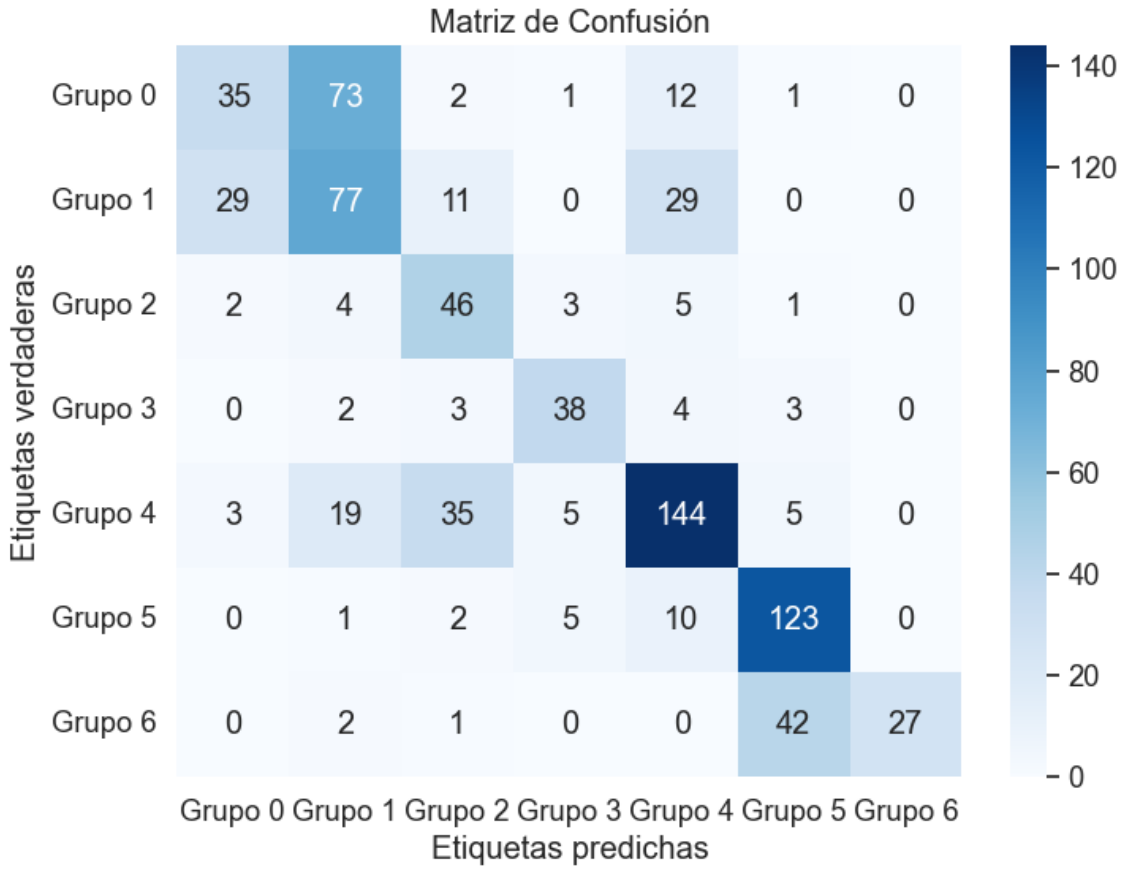

Figura 9: Matriz confusión obtenida para el algoritmo TfidfVectorizer-Kmeans

4.1.2. HashingVectorizer

Ahora realizamos el mismo procedimiento con la representación HashingVectorizer. Obtenemos los siguientes histogramas de distribución:

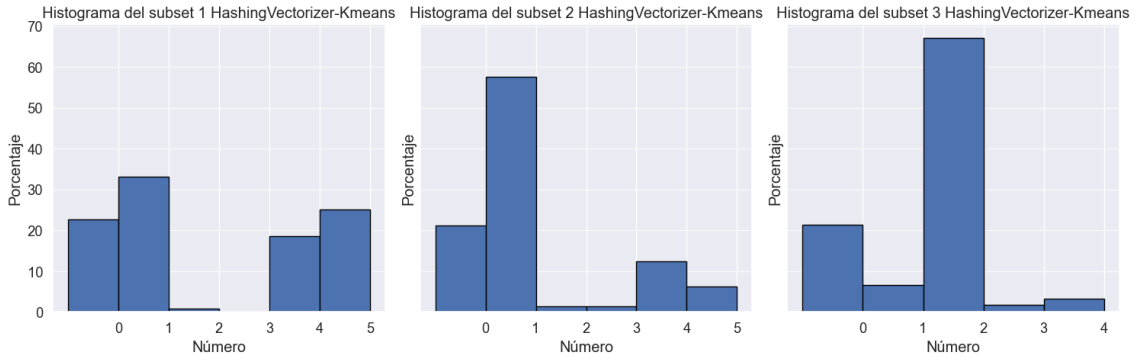

Figura 10: Histogramas de los subsets 1, 2 y 3

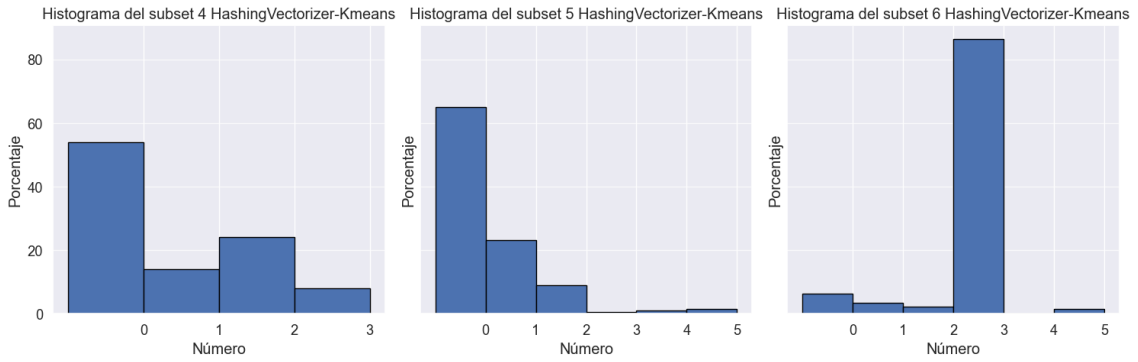

Figura 11: Histogramas de los subsets 4, 5 y 6

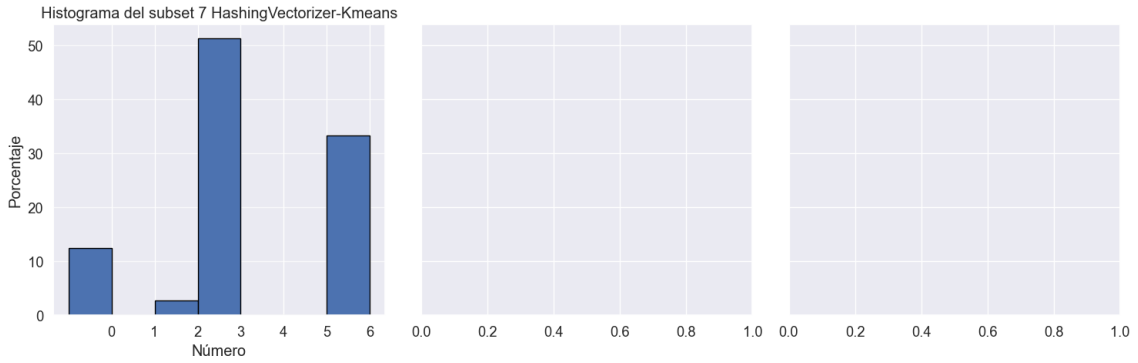

Figura 12: Histogramas del subset 7

En el primer histograma, observamos una proximidad entre los números 0, 1, 4 y 5; seleccionaremos el 5 como representante del grupo 0. En el segundo histograma, el número 1 es mayoritario, por lo tanto, lo elegimos para representar al grupo 1. En el tercer histograma, el número 2 es el más frecuente, pasando a representar al grupo 2.

En el cuarto histograma, asignamos al número 4 como representante del grupo 3. Esta elección se basa en la eliminación de otras opciones, ya que los números más comunes aparecerán en los histogramas siguientes. En el quinto histograma, optamos por el 0 como representante del grupo 4 debido a su predominancia. En el sexto, el número 3 es claramente mayoritario, convirtiéndose en nuestra elección para representar al grupo 5. En el último histograma, aunque el número 3 sigue siendo predominante, seleccionamos el 6 como representante del grupo 6, ya que ocupa el segundo lugar y es el único grupo en el que tiene presencia significativa.

Una vez realizadas las permutaciones correspondientes, vamos a comparar la solución obtenida por el algoritmo con respecto al *gold standard*. Calculamos las medidas de evaluación externa, las cuales son: Precisión, Cobertura y Medida-F.

Métrica	**Valor**
Precisión	0.584
Cobertura	0.545
Medida-F	0.529

Cuadro 4: Métricas obtenidas para el algoritmo HashingVectorizer-Kmeans

La matriz de confusión obtenida es la siguiente:

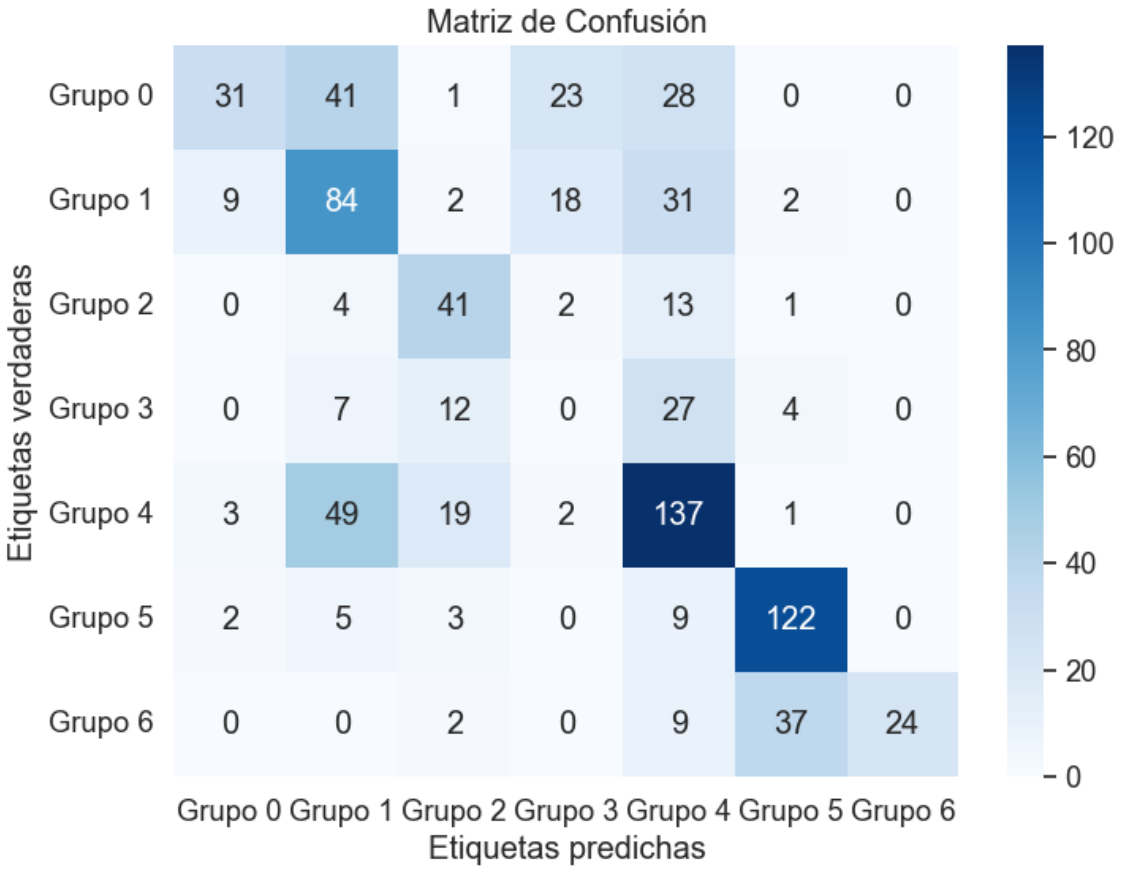

Figura 13: Matriz confusión obtenida para el algoritmo HashingVectorizer-Kmeans

4.2. Alternativa a Kmeans: Gaussian Mixture Model

Como alternativa a Kmeans vamos a usar Gaussian Mixture Model (GMM), A diferencia de K-Means, que asigna cada punto de datos a un solo cluster, GMM es más flexible, ya que permite que los puntos de datos pertenezcan a múltiples clusters con diferentes grados de pertenencia. Funciona asumiendo que los datos se generan a partir de una mezcla de distribuciones gaussianas. La idea es encontrar estas distribuciones gaussianas, cada una representando un cluster. Cada cluster se caracteriza por su centro (media) y su dispersión (covarianza).

A diferencia de K-Means, que asume clusters con formas geométricas esféricas y de igual tamaño, GMM puede manejar clusters con diferentes formas y tamaños, ya que se basa en distribuciones gaussianas. Mientras que K-Means asigna cada punto de datos a un solo cluster, GMM calcula la probabilidad de que cada punto pertenezca a cada cluster. Esto permite una asignación suave, lo que significa que un punto puede pertenecer a múltiples clusters con diferentes grados de certeza.

La implementación es la siguiente:

```python
from sklearn.mixture import GaussianMixture
from sklearn.decomposition import TruncatedSVD

data_dense = tfidf_matrix.toarray()

random_seed = 22
num_clusters = 7

# Reducción de la dimensionalidad con TruncatedSVD
svd = TruncatedSVD(n_components=1000)
data_reduced = svd.fit_transform(data_dense)

# Inicializamos el modelo GMM
gmm = GaussianMixture(n_components=num_clusters, random_state=random_seed)

# Ajustamos el modelo a los datos reducidos
gmm.fit(data_reduced)

# Obtenemos las etiquetas de cluster
etiquetas_clusters_3 = gmm.predict(data_reduced)
```

Figura 14: Implementación del algoritmo Gaussian Mixture

Al igual que antes probamos con diferentes semillas con ambas representaciones obteniendo para el caso de TF-IDF un índice de Rand de 0.264 y para el caso de HashingVectorizer un índice de 0.231.

4.2.1. TF-IDF

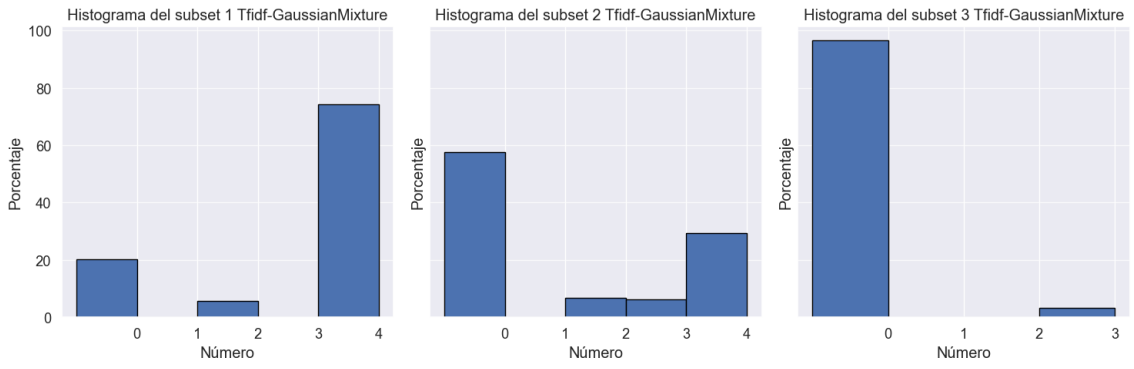

Figura 15: Histogramas de los subsets 1, 2 y 3

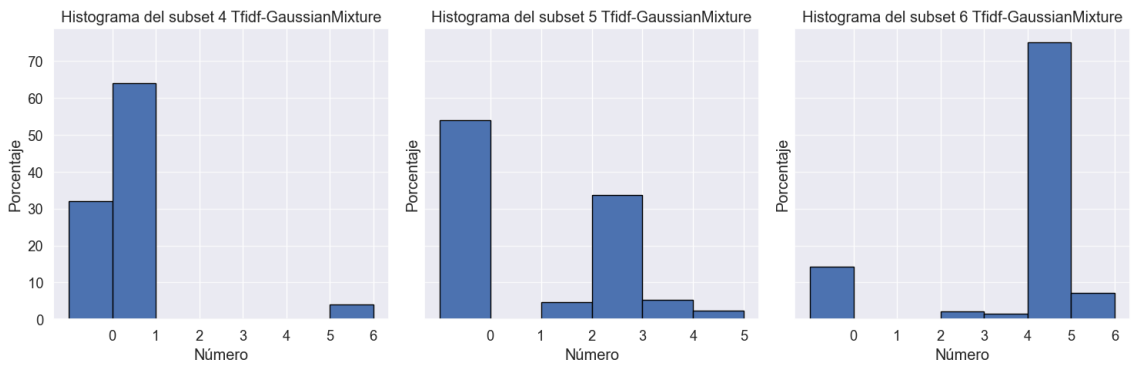

Figura 16: Histogramas de los subsets 4, 5 y 6

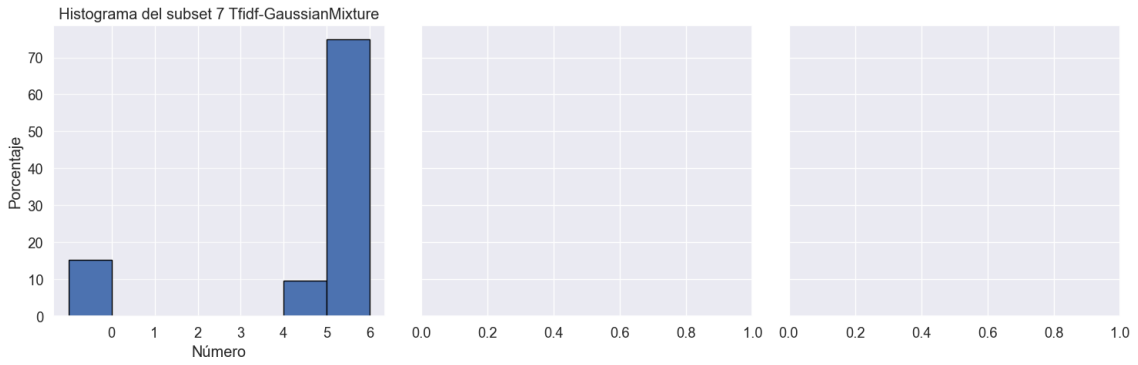

Figura 17: Histogramas del subset 7

En el primer histograma, elegimos el número 4 como representante del grupo 0. En el segundo histograma, optamos por el número 2 por descarte, ya que, aunque tiene poca presencia, es la opción más recurrente. En el tercer histograma, seleccionamos el 0 para representar al grupo 2 debido a su abrumadora mayoría.

En el cuarto histograma, tomamos el número 1 como representante del grupo 3, ya que es la opción mayoritaria. En el quinto histograma, escogemos el número 3 para representar al grupo 4, ya que es la segunda opción más repetida y tiene una presencia notable en este grupo. En el sexto histograma, elegimos el 5 como representante del grupo 5 debido a que es mayoritario. Finalmente, en el séptimo histograma, seleccionamos el número 6 como representante del grupo 6, ya que es la opción más frecuente en este caso.

Una vez realizadas las permutaciones correspondientes, vamos a comparar la solución obtenida por el algoritmo con respecto al *gold standard*. Calculamos las medidas de evaluación externa, las cuales son: Precisión, Cobertura y Medida-F.

Métrica	**Valor**
Precisión	0.688
Cobertura	0.527
Medida-F	0.536

Cuadro 5: Métricas obtenidas para el algoritmo TF-IDF-Gaussian Mixture

La matriz de confusión obtenida es la siguiente:

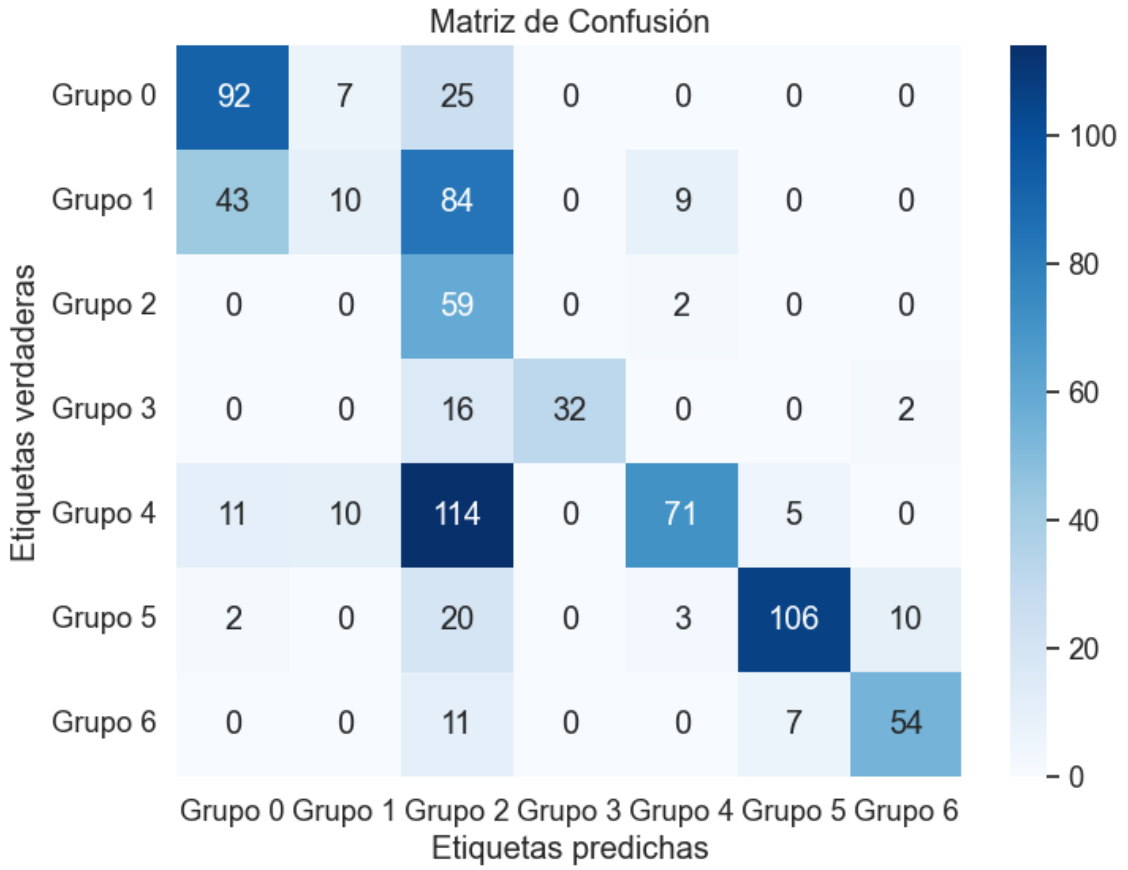

Figura 18: Matriz confusión obtenida para el algoritmo TF-IDF-Gaussian Mixture

4.2.2. HashingVectorizer

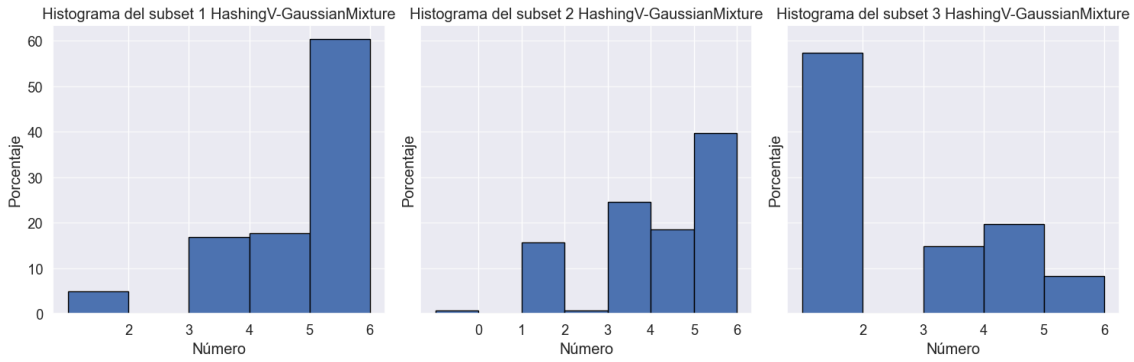

Figura 19: Histogramas de los subsets 1, 2 y 3

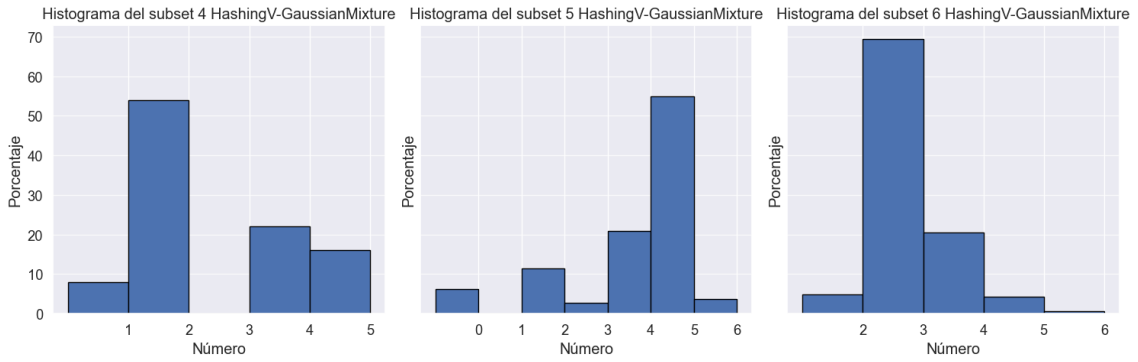

Figura 20: Histogramas de los subsets 4, 5 y 6

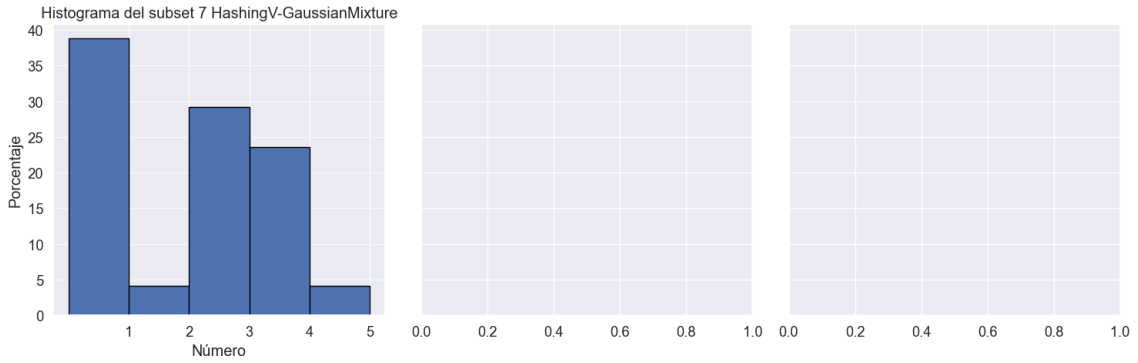

Figura 21: Histogramas del subset 7

En el primer histograma, seleccionamos la opción 6 para representar al grupo 0, ya que es la más frecuente. En el segundo histograma, optamos por el número 4 para representar al grupo 1, siendo la segunda opción más común después del 6. En el tercer histograma, elegimos el número 2 como representante del grupo 2, al ser la opción mayoritaria.

En el cuarto histograma, nos vemos obligados a elegir el 0 como representante del grupo 3, aunque no tenga presencia en ningún grupo. En el quinto histograma, escogemos el 5 como representante del grupo 4 debido a su alta repetición. En el sexto histograma, seleccionamos el 3 para representar al grupo 5, siendo el más frecuente. En el último histograma, optamos por el 1 como representante del grupo 6 por ser la opción mayoritaria.

Una vez realizadas las permutaciones correspondientes, vamos a comparar la solución obtenida por el algoritmo con respecto al *gold standard*. Calculamos las medidas de evaluación externa, las cuales son: Precisión, Cobertura y Medida-F.

Métrica	**Valor**
Precisión	0.510
Cobertura	0.482
Medida-F	0.482

Cuadro 6: Métricas obtenidas para el algoritmo HashingVectorizer-Gaussian Mixture

La matriz de confusión obtenida es la siguiente:

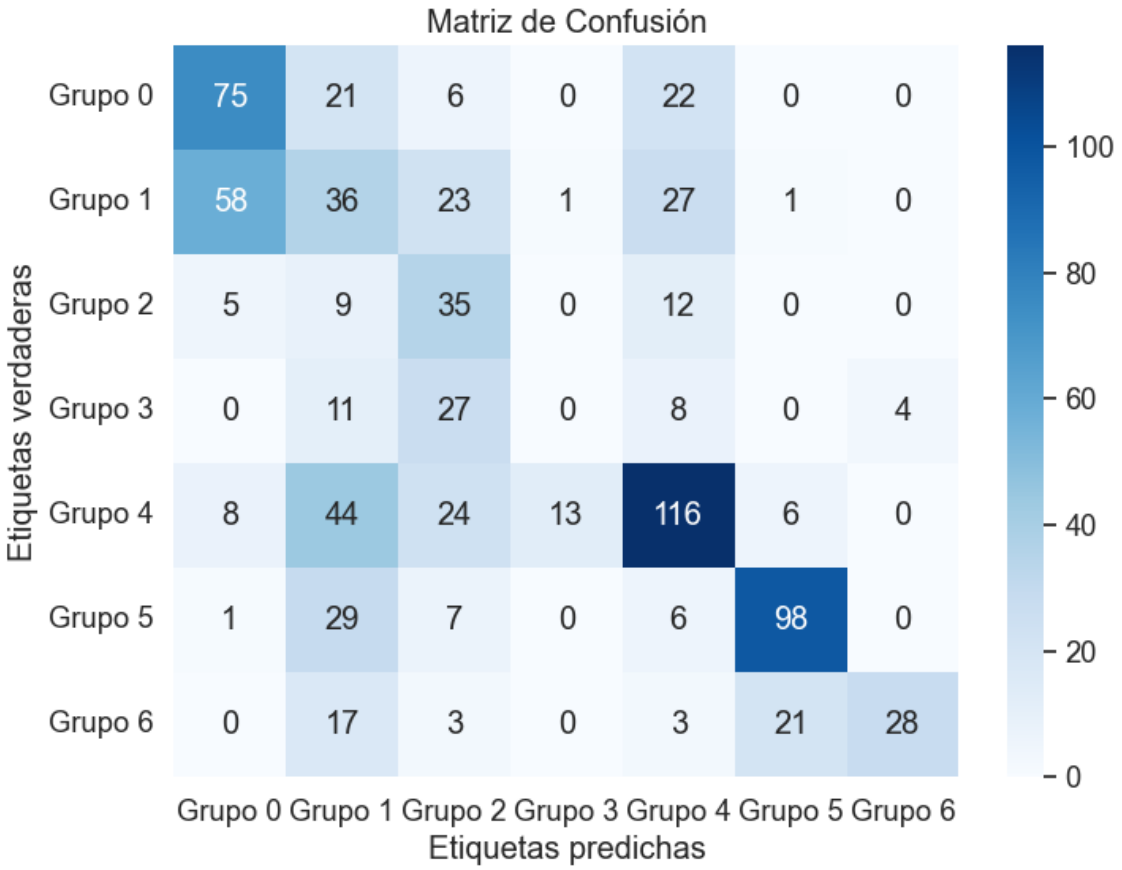

Figura 22: Matriz confusión obtenida para el algoritmo HashingVectorizer-Gaussian Mixture

5. Resultados conjuntos e interpretación

A continuación presentamos los resultados conjuntos de los cuatro métodos llevados a cabo en el apartado anterior:

Método	**Precisión**	**Cobertura**	**Medida-F**
TfidfVectorizer-Kmeans	0.634	0.609	0.599
HashingVectorizer-Kmeans	0.584	0.545	0.529
TfidfVectorizer-Gaussian Mixture	0.688	0.527	0.536
HashingVectorizer-Gaussian Mixture	0.510	0.482	0.482

Cuadro 7: Resultados de los diferentes métodos de vectorización y clustering

El primer método ha mostrado el mejor rendimiento y también ha sido el más sencillo de interpretar y predecir a qué grupo podría pertenecer un cluster dado. Al examinar la matriz de confusión del primer método, se observa que, en el caso del grupo cero, el algoritmo tiende a confundirlo con el grupo 1, asignando más elementos a este último que al propio grupo 0. Además, asigna algunas unidades al grupo 4. Respecto al grupo 1, acierta en la mayoría de los casos, pero a veces se confunde con el grupo 0 y el grupo 4. Una posible explicación de estas mezclas entre los grupos radica en su relación: los tres están vinculados con la tecnología, especialmente el grupo 0 y el grupo 1, que se centran en hardware. El grupo 4 trata sobre electrónica y podría compartir términos comunes con los dos grupos anteriores.

El grupo 2, centrado en autos, tiene un rendimiento generalmente sólido, acertando alrededor del 75 %. Por otro lado, el grupo 3, relacionado con hockey, muestra un rendimiento bueno, acertando casi el 80 % de las veces. Respecto al grupo 4, que se enfoca en electrónica, presenta un rendimiento sólido, acertando casi el 70 % de las veces; sin embargo, el modelo a veces lo confunde con los grupos 1 y 2; también se asigna erróneamente valores que deberían corresponder al grupo 1 y 0. El grupo 5,

que aborda armas en contextos políticos, destaca con un rendimiento superior al 85 %. Por último, el grupo 6 muestra un rendimiento más pobre: aunque acierta en un número significativo de casos, suele ser confundido mayormente con el grupo 5. Esto puede explicarse porque el grupo 6 trata sobre política en Oriente Medio, y los conflictos bélicos pueden ser malinterpretados como parte del grupo 5.

En el segundo método, esencialmente observamos lo mismo que en el primer método. La diferencia más significativa radica en su rendimiento nulo con respecto al grupo 3, lo que impacta negativamente en el rendimiento general.

El tercer método es más complicado de interpretar. Tiene bastante buen rendimiento con los grupos 5 y 6. El grupo 2 está sobrerrepresentado lo que produce muchos falsos negativos. Se puede también apreciar confusiones en el grupo 0 y 1. En términos de rendimiento presenta unos resultados similares al anterior aun siendo más complicado de interpretar.

El último método es el que peor rendimiento general posee. Se sigue evidenciando las confusiones entre el grupo 0 y 1. El grupo 5 presenta un buen rendimiento, el grupo 6 es confundido con el 5 a veces. Existe una sobrerrepresentación del grupo 1 a costa del grupo 3 que prácticamente es inexistente.

6. Parte opcional: método Elbow

El método del codo (Elbow method) se utiliza para determinar el número óptimo de clústeres en un algoritmo de agrupamiento, como K-Means. Este método grafica el número de clústeres en el eje x y la variabilidad dentro de cada clúster en el eje y.

Se basa en la idea de que al aumentar el número de clústeres, generalmente disminuye la variabilidad dentro de cada clúster, ya que los puntos están más cercanos entre sí y se ajustan mejor a sus centros. Sin embargo, llega un momento en que agregar más clústeres no mejora significativamente la cohesión de los datos. En este punto, el beneficio de agregar más clústeres se reduce, lo que se refleja en el gráfico.

La técnica consiste en calcular el valor de alguna métrica de evaluación (como la suma de las distancias cuadradas dentro de los clústeres) para diferentes números de clústeres y representar esta métrica en función del número de clústeres. Se observa que al principio, al aumentar el número de clústeres, la métrica decrece rápidamente, lo que significa que se mejora la cohesión dentro de los clústeres. Sin embargo, llega un punto donde añadir más clústeres tiene un impacto menor en la reducción de la métrica, formándose un 'codo' en la gráfica.

El punto en el que se produce este codo se considera una indicación del número óptimo de clústeres, ya que representa el equilibrio entre la cohesión dentro de los clústeres y la simplicidad del modelo. En ese punto, agregar más clústeres no ofrece suficiente beneficio en la estructuración de los datos, por lo que se elige el número de clústeres justo antes del codo como la mejor opción para la agrupación.

Vamos a implementar el método deElbow aplicando el primer método de clustering, ya que era el que mejor resultados ofrecía. La implementación es la siguiente:

```python
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

inertia_values = []
k_values = range(1, 13)

for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=22)
    kmeans.fit(tfidf_matrix)
    inertia_values.append(kmeans.inertia_)

# Representamos la inercia en función del número de clusters
plt.figure(figsize=(8, 6))
plt.plot(k_values, inertia_values, marker='o')
plt.xlabel('Número de Clusters')
plt.ylabel('Inercia')
plt.title('Método Elbow')
plt.show()
```

Figura 23: Implementación del método de Elbow

En este caso la métrica de evaluación es la inercia, que es la suma de las distancias al cuadrado de cada punto con respecto a su centroide más cercano divido el total de datos:

$$\text{inercia} = \frac{\sum \text{distancia(punto, centroide)}^2}{N} \tag{1}$$

Al implementarlo obtenemos la siguiente gráfica:

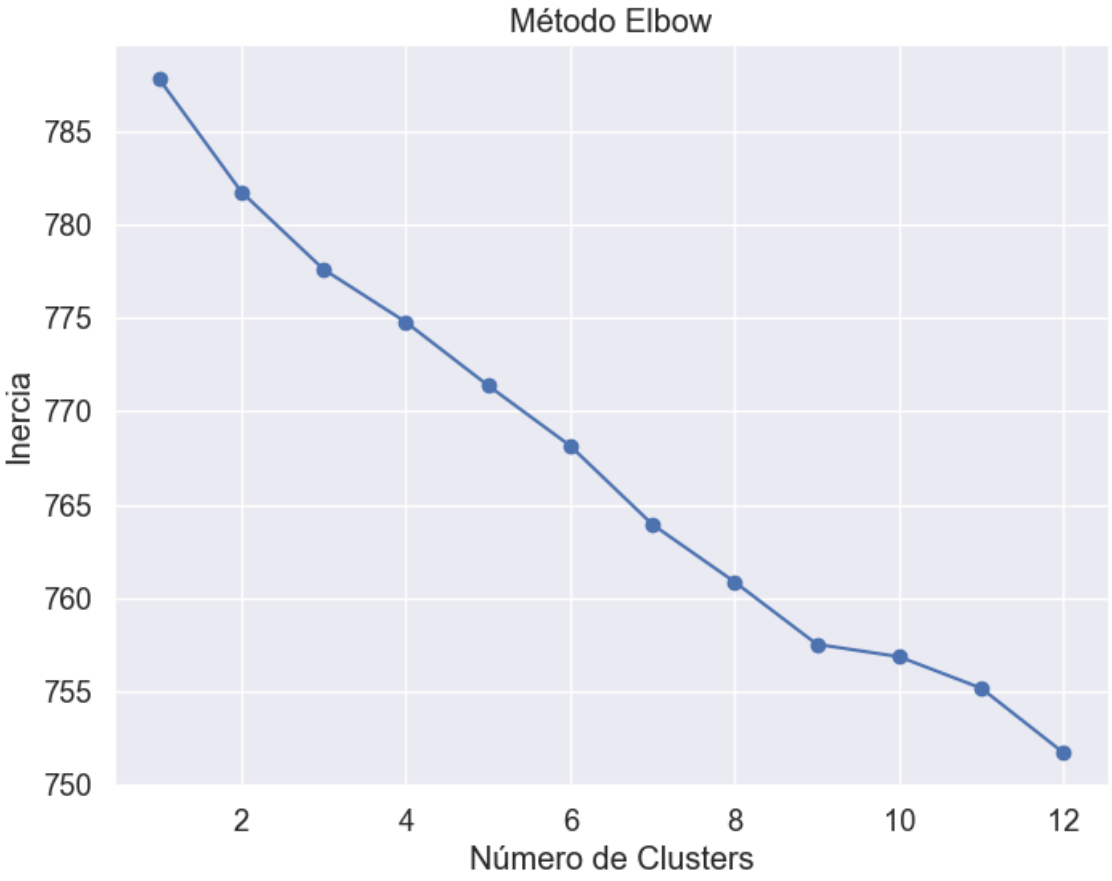

Figura 24: Representación del número de clusters frente a la Inercia

El primer punto donde vemos que suaviza el decrecimiento es de $k = 9$, lo que implica que según el método de Elbow el número optimo de clusters para este problema sería 9.

Ahora pasamos realizar el clustering con $k = 9$ y vamos a comparar los resultados con los grupos categorizados inicialmente. El resultado es el siguiente:

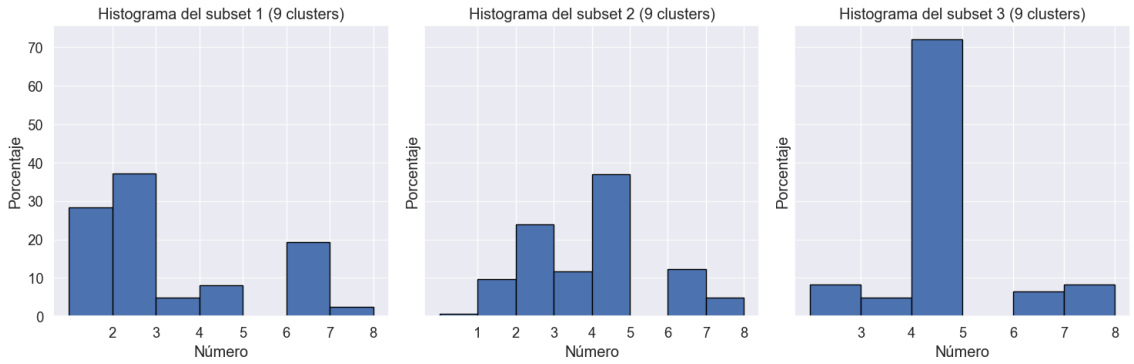

Figura 25: Histogramas de los subsets 1, 2 y 3

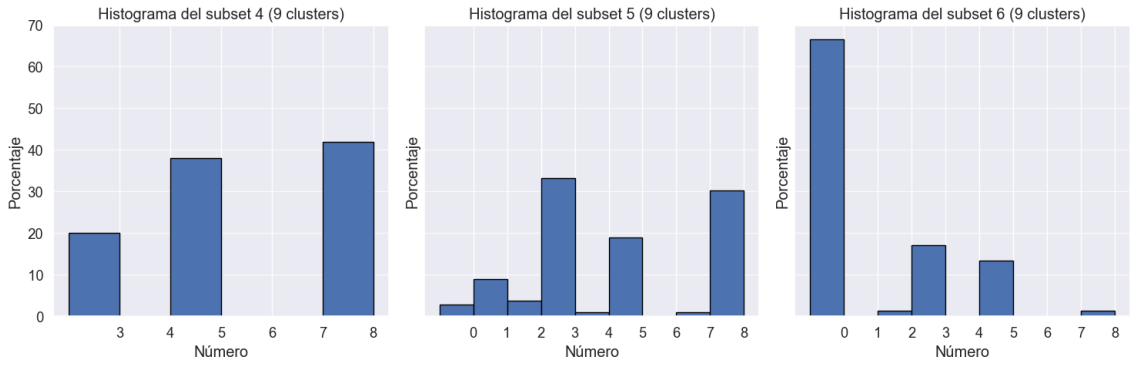

Figura 26: Histogramas de los subsets 4, 5 y 6

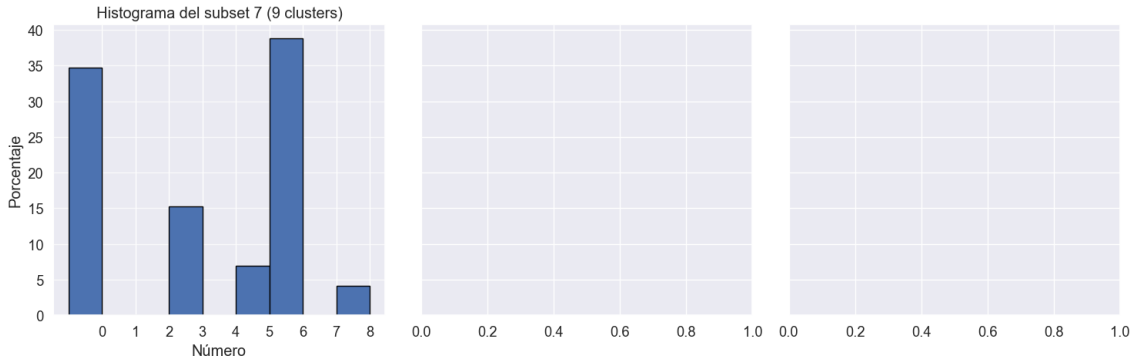

Figura 27: Histogramas del subset 7

La interpretación ahora se hace aun más complicada, vemos que el número 5 es muy representativo del grupo 2, el número 0 solo aparece en el grupo 5 y 6, el número 6 solo aparece en el grupo 6. Con respecto a los demás números, al aumentar el número de clusters se hace una tarea complicada poder interpretar que significan.

7. Conclusiones

Entre los métodos de clustering empleados, el primero que hemos presentado es el que más se asemeja a la agrupación previa realizada (gold standard), tanto en términos numéricos de rendimiento como en interpretabilidad, aunque aún tiene margen de mejora. Hay que destacar la complejidad del problema, equiparable a la tarea de solicitar a una persona que organice un conjunto de mensajes en 7 grupos y, posteriormente, pedir a otra persona que, con los mismos mensajes, elija 7 grupos bajo su criterio y los organice según ese criterio. Es altamente probable que al comparar los resultados, no sean completamente similares. Por lo tanto, al abordar un problema de esta envergadura, es fundamental que los mensajes sean representativos de cada grupo, ya que de lo contrario, el algoritmo podría agruparlos según otras características.