TAREA 2: Minería de Textos

Clustering de Noticias

Alumno: Alex Herrerías Ramírez

17 de enero de 2026

Índice

1. Descripción de la tarea

El objetivo de la práctica es aplicar diferentes técnicas de *clustering* sobre el subconjunto de la colección *Corpus-Clustering* que se nos ha proporcionado. El objetivo será clasificar automáticamente los documentos en grupos temáticos sin utilizar las etiquetas de clase durante el entrenamiento.

1.1. El conjunto de datos

Se trabaja con una subcolección compuesta por 7 categorías de temáticas distintas. Cada documento es un fichero de texto que contiene el cuerpo de un mensaje de un grupo de noticias y sus cabeceras que deben ser limpiadas.

Las categorías reales son:

- *comp.graphics*
- *comp.os.ms-windows.misc*
- *comp.sys.ibm.pc.hardware*
- *comp.sys.mac.hardware*
- *rec.autos*
- *rec.motorcycles*
- *talk.politics.guns*

Podemos observar que en algunas clases hay una similitud (por ejemplo *mac.hardware* y *ibm.pc.hardware*) lo que supone un reto para la separación de los clústeres.

2. Desarrollo e implementación: Modelo base

Siguiendo el modelo de desarrollo utilizado en la practica 1, primero implementaremos una solución usando los modelos base y un preprocesamiento estándar como es NLTK y posteriormente, implementaremos diferentes mejoras para obtener el mejor resultado.

2.1. Preprocesamiento y carga de datos

Para la limpieza de texto, se ha dividido la tarea en dos funciones separadas, las cuales se encargan:

1. **Limpieza de cabeceras:** Se elimina todo el contenido anterior al primer doble salto de línea.
2. **Normalización:** Se eliminan caracteres no alfanuméricos y se convierte a minúsculas.
3. **Tokenización y Lematización:** Se usa NLTK para tokenizar, filtrar *stopwords* y palabras cortas (longitud < 3), y finalmente lematizar.

```
def quitar_cabecera(texto):
    partes = re.split(r"\n\s*\n", texto, maxsplit=1)
    return partes[1] if len(partes) > 1 else partes[0]

def preprocesar_texto(texto):
    # quitamos la cabecera
    cuerpo = quitar_cabecera(texto)

    # dejamos solo las letras en minusculas y numeros
    texto_limpio = re.sub(r'[^a-zA-Z0-9\s]', ' ', cuerpo).lower()

    # tokenizacion y lematizacion
    tokens = nltk.word_tokenize(texto_limpio)
    tokens_lematizados = [
        lemmatizer.lemmatize(w) for w in tokens
        # filtramos las stopwords y palabras muy cortas
        if w not in stop_words_en and len(w) > 2
    ]
    return " ".join(tokens_lematizados)
```

Listing 1: Funciones de preprocesamiento con NLTK

Para la carga de datos he desarrollado una función que recorre cada subdirectorio (se asume que el archivo Python se ejecuta dentro del directorio *Corpus-Clustering*) y procesa el contenido para almacenarlo en un DataFrame junto su etiqueta real.

```
def cargar_datos():
    datos = []
    categorias = [d for d in os.listdir(".")
                  if os.path.isdir(d) and not d.startswith('.')]

    for cat in categorias:
        ruta = os.path.join(".", cat)
        for archivo in os.listdir(ruta):
            with open(os.path.join(ruta, archivo), "r",
                      encoding="latin-1") as f:
                datos.append({
                    "texto": preprocesar_texto(f.read()),
                    "etiqueta_real": cat
                })
    return pd.DataFrame(datos)
```

Listing 2: Carga de datos

2.2. Métricas de Evaluación

Para medir el rendimiento de los modelos se ha decidido utilizar tres métricas: **V-Measure** (equilibrio entre homogeneidad y completitud) **ARI** (índice de Rand ajustado) y **BCubed F1** (precisión y recall ponderados).

```
def b_cubed_score(y_true, y_pred):
    cm = contingency_matrix(y_true, y_pred)
    n = np.sum(cm)
    row_sums = np.sum(cm, axis=1)
    col_sums = np.sum(cm, axis=0)

    precision = np.sum((cm ** 2) / col_sums[None, :]) / n
    recall = np.sum((cm ** 2) / row_sums[:, None]) / n

    f1 = 2 * (precision * recall) / (precision + recall)
    return precision, recall, f1
```

Listing 3: Función para el calculo de BCubed

2.3. Entrenamientos modelos

Para el modelo base, se diseñó un bucle que compara:

- **Dos representaciones:** TF-IDF vs Conteo simple. Se usaron los filtros de frecuencia `min_df=5`, `max_df=0.5` para eliminar ruido y palabras demasiado comunes ya que sin ellos el rendimiento era pesimo.
- **Dos algoritmos:** K-Means y Clustering Jerárquico Aglomerativo como se nos pide en el ejercicio.
- **Reducción LSA:** Se aplicó una reducción a 100 componentes mediante SVD antes del clustering para poder entrenar el modelo.

```
# reducción de dimensionalidad con LSA
lsa_pipeline = make_pipeline(TruncatedSVD(n_components=100, random_state=42),
    Normalizer(copy=False))

# Configuraciones tras varías iterraciones con los mejores valores posibles
vectores = {
    'TF-IDF': TfidfVectorizer(
        max_features=5000,
        min_df=5,
        max_df=0.5,
        stop_words='english'
    ),
    'Count': CountVectorizer(
        max_features=5000,
        min_df=5,
        max_df=0.5,
        stop_words='english'
    )
}

modelos = {
    'KMeans': KMeans(
        n_clusters=7,
        random_state=42,
        n_init=30,
        max_iter=500
    ),
    'Jerarquico': AgglomerativeClustering(n_clusters=7)
}
```

Listing 4: Configuración de vectores y modelos

Finalmente, se itera sobre todas las combinaciones para obtener las métricas:

```
for nombre_vector, vector in vectores.items():
    X_sparse = vector.fit_transform(df['texto'])
    X_lsa = lsa_pipeline.fit_transform(X_sparse)

    for nombre_modelo, modelo in modelos.items():
        y_pred = modelo.fit_predict(X_lsa)

        # Calculo de metricas
        v_score = v_measure_score(y_true, y_pred)
        ari = adjusted_rand_score(y_true, y_pred)
        bc_p, bc_r, bc_f1 = b_cubed_score(y_true, y_pred)
```

Listing 5: Bucle de evaluación

3. Resultados del modelo base

3.1. Evaluación

Podemos ver en la siguiente tabla resume el rendimiento de las cuatro configuraciones.

Vectorización	Algoritmo	V-Measure	ARI	BCubed F1
TF-IDF	**K-Means**	**0.4497**	**0.3190**	0.4758
TF-IDF	Jerárquico	0.4147	0.1794	**0.5072**
Count	K-Means	0.2494	0.1682	0.3303
Count	Jerárquico	0.2932	0.1868	0.3931

Cuadro 1: Comparativa de métricas de los modelos base

3.2. Análisis de los resultados iniciales

3.2.1. Impacto de la representación (TF-IDF vs Count)

Podemos concluir de los resultados que la representación TF-IDF es superior al conteo simple de palabras en todas las métricas y algoritmos.

- El **V-Measure** del mejor modelo TF-IDF (0.4497) casi duplica al del modelo Count (0.2494).
- Esto se debe en gran parte a que en textos de foros hay muchas palabras comunes que no son *stopwords* estrictas pero que aparecen en todos los temas. TF-IDF penaliza estas palabras, mientras que *CountVectorizer* les da mucho peso solo por ser frecuentes.

3.2.2. Comparación de algoritmos (K-Means vs Jerárquico)

Dependiendo de la métrica, podemos observar diferentes comportamientos:

- K-Means con TF-IDF: Obtiene el mejor equilibrio general y el ARI más alto (0.3190). Dado a que el ARI penaliza las asignaciones incorrectas, estos resultados nos sugieren que K-Means define fronteras más claras entre los grupos principales.
- Jerárquico Aglomerativo: Aunque su ARI es bajo (0.1794), obtiene el mejor BCubed F1 (0.5072). Si observamos el resto de métricas (Precision 0.39, Recall 0.71), vemos que el modelo jerárquico tiende a agrupar muchos documentos juntos, pero mezclando temas. Esto suele ocurrir por el efecto de encadenamiento en el clustering aglomerativo, donde se fusionan grupos grandes prematuramente.

3.3. Conclusión de la fase inicial

El modelo base seleccionado para la fase de optimización será K-Means con TF-IDF, ya que ofrece la estructura de clústeres más robusta (mejor V-Measure y ARI).

Sin embargo, los valores absolutos (ARI ≈ 0.32) indican que todavía hay un margen de mejora significativo, probablemente limitado por la calidad del preprocesamiento de NLTK y la falta de ajuste de hiperparámetros.

4. Mejoras y optimización del modelo

Tras observar las limitaciones del modelo base, se ha decidido implementar un flujo de trabajo más robusto enfocado en dos áreas: la calidad del preprocesamiento lingüístico y la optimización de la estructura de los datos mediante validación cruzada.

4.1. Mejora 1: Preprocesamiento con Spacy

Se sustituyó la lematización básica de NLTK por la librería **Spacy**, utilizando su modelo en inglés de gran tamaño (`en_core_web_lg`).

Justificación: El modelo *Large* de Spacy no solo tokeniza, sino que utiliza vectores de palabras y un análisis morfosintáctico profundo para desambiguar lemas. Esto es crucial en textos como los nuestros donde una misma palabra puede actuar como verbo o sustantivo dependiendo del contexto.

```
try:
    nlp = spacy.load("en_core_web_lg")
except OSError:
    print("Error al cargar el modelo")

def preprocesar_texto_spacy(texto):
    # mismo preprocesamiento de cabeceras y caracteres
    partes = re.split(r"\n\s*\n", texto, maxsplit=1)
    cuerpo = partes[1] if len(partes) > 1 else partes[0]
    texto_limpio = re.sub(r'[^a-zA-Z0-9\s]', ' ', cuerpo).lower()

    # orocesamiento con Spacy
    nlp.max_length = 2000000
    doc = nlp(texto_limpio)

    # filtrado final
    tokens = [token.lemma_ for token in doc if not token.is_stop and not token.
        is_punct and len(token.text) > 2]
    return " ".join(tokens)

# usamos una variante de cargar_datos usando spacy, el codigo es el mismo
df_spacy = cargar_datos_spacy()
le = LabelEncoder()
y_true_spacy = le.fit_transform(df_spacy['etiqueta_real'])
```

Listing 6: Preprocesamiento avanzado con Spacy

4.2. Mejora 2: Pipeline con LSA y validación cruzada

Como uno de los problemas de este Dataset es la dimensionalidad y encontrar la configuración óptima sin producir overfitting se diseñó un `Pipeline` que integra: 1. **TF-IDF:** Vectorización ponderada. 2. **SVD (LSA):** Reducción de dimensionalidad para capturar la semántica latente. 3. **Normalización:** Para que K-Means funcione correctamente con distancias euclídeas 4. **K-Means:** Algoritmo de agrupamiento.

Se utilizó **GridSearchCV** con validación cruzada de 5 pliegues. Esto divide los datos en 5 partes, entrenando en 4 y validando en 1, rotando el proceso 5 veces para asegurar que los resultados son estables.

```
# aplicamos tf-idf, luego svd normalizamos y aplicamos kmeans
pipeline_optimo = Pipeline([
    ('tfidf', TfidfVectorizer(stop_words='english')),
    ('svd', TruncatedSVD(random_state=42)),
    ('norm', Normalizer(copy=False)),
    ('kmeans', KMeans(n_clusters=7, random_state=42))
])

# hiperparametros a probar
param_grid = {
    'tfidf__max_features': [3000, 5000, 7000, 10000],
    'tfidf__ngram_range': [(1, 1), (1, 2)],
    'svd__n_components': [10, 30, 50, 70],
    'kmeans__n_init': [30, 50],
}

grid = GridSearchCV(pipeline_optimo, param_grid, cv=5, scoring=make_scorer(
    v_measure_score), n_jobs=-1)
grid.fit(df_spacy['texto_spacy'], y_true_spacy)
```

Listing 7: Búsqueda de hiperparámetros con Cross Validation

5. Resultados del modelo optimizado

El proceso de *Grid Search* determinó que la mejor configuración utiliza 30 componentes LSA y unigramas ((1, 1)), con un vocabulario limitado a 3000 términos y un total de 50 inicializaciones para el algoritmo K-Means.

5.1. Evaluación

Al aplicar el modelo ganador sobre todo el conjunto de datos, se obtuvieron las siguientes métricas, superando claramente al modelo base:

Métrica	**Modelo Base**	**Modelo Optimizado (Spacy+CV)**
V-Measure	0.4660	**0.5325**
ARI	0.2850	**0.4252**
BCubed F1	0.4520	**0.5565**

Cuadro 2: Comparativa de rendimiento

La mejora en **BCubed F1 (0.55)** indica que los clústeres son internamente más coherentes y puros. Spacy logró unificar términos sinónimos que NLTK trataba como distintos, y LSA agrupó documentos semánticamente similares antes de que K-Means los separara.

6. Parte opcional: Determinación del K optimo

En la parte obligatoria se nos forzó a que el algoritmo usase $k = 7$ grupos porque conocemos las etiquetas reales. Para esta parte, aplicaremos el Método del Codo para obtener el k optimo.

6.1. Metodología: Método elbow

Este método consiste en ejecutar K-Means para un rango de valores de k (en este caso, de 2 a 20) y calcular la inercia para cada ejecución.

Se busca el çodo"de la curva: el punto donde aumentar el número de clústeres deja de reducir la inercia significativamente, indicando que el beneficio de subdividir más los datos es marginal.

```
inercias = []
rango_k = range(2, 20)

for k in rango_k:
    # Entrenamos KMeans para cada k sobre la matriz LSA optima
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(X_lsa_opt)
    inercias.append(kmeans.inertia_)
```

Listing 8: Implementación del Método Elbow

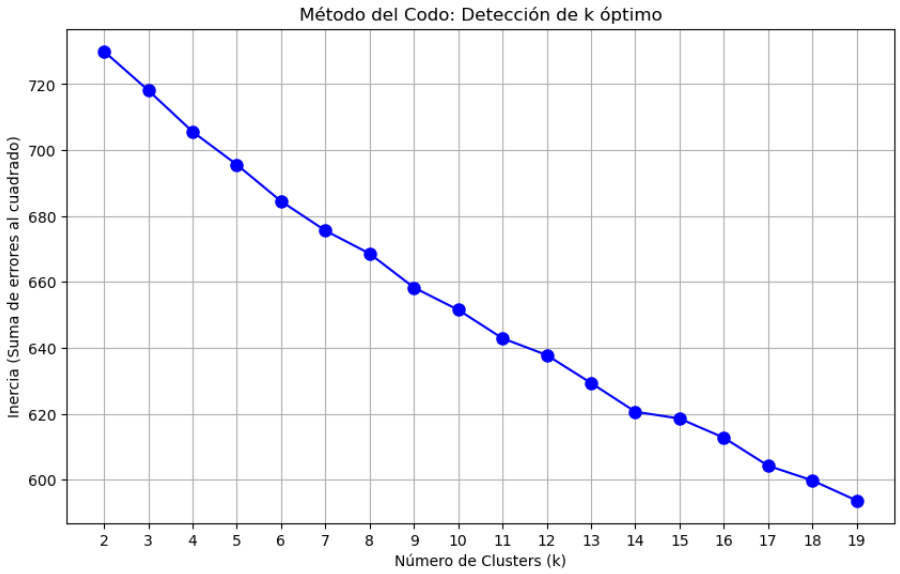

Figura 1: Método del Codo para la selección de k

6.2. Análisis del gráfico y selección de K

Al analizar la gráfica de la inercia, se observó que la curva no tiene un codo perfecto en $k = 7$. En su lugar, la curva sigue descendiendo suavemente. Sin embargo, se detectó un punto de inflexión interesante alrededor de $k = 14$.

Decisión: Se seleccionó $k = 14$ como el número óptimo de clústeres. Esto sugiere que, aunque hay 7 categorías, los temas son lo suficientemente diversos como para que el algoritmo detecte sub-temas.

6.3. Resultados con K=14

Se reentrenó el modelo con $k = 14$ y se evaluó.

```
K_NUEVO = 14
modelo_optimo = KMeans(n_clusters=K_NUEVO, random_state=42)
y_pred_optimo = modelo_optimo.fit_predict(X_lsa_opt)

ari_opt = adjusted_rand_score(y_true, y_pred_optimo)
nmi_opt = normalized_mutual_info_score(y_true, y_pred_optimo)
f1_opt = f1_score(y_true, y_pred_optimo, average='macro', zero_division=0)

print(f"F1: {f1_opt:.4f} ARI: {ari_opt:.4f} NMI: {nmi_opt:.4f}")
```

Listing 9: Entrenamiento del modelo final K=14

6.3.1. Métricas externas

- **ARI (0.2798):** Se mantiene casi idéntico al modelo base.Esto nos indica que aunque hemos doblado el número de grupos, las asignaciones no son aleatorias; el modelo simplemente está subdividiendo las clases existentes.
- **NMI (0.4817):** La Información Mutua Normalizada es alta, confirmando que los 14 grupos tienen una fuerte correlación con las 7 etiquetas originales.

7. Conclusiones generales

La realización de esta práctica me ha permitido entender que:

1. Calidad de datos vs algoritmo: El cambio a Spacy tuvo un impacto mayor en las métricas que el cambio de algoritmo en sí, lo que me deja claro que la limpieza semántica es fundamental y si no se hace correctamente, el resto del trabajo sera inservible.
2. Dimensionalidad: El uso de LSA fue útil para mejorar el tiempo de ejecución y la calidad de los clústeres filtrando el ruido de los textos.
3. Estructura de los datos: El análisis de la parte opcional ($k = 14$) revela que la colección *20 Newsgroups* tiene una estructura mas amplia de lo que sugieren sus 7 etiquetas generales.