Práctica 2 - Clasificar textos en categorías

Joaquín Jiménez López de Castro — jo.jimenez@alumnos.upm.es
Ángel Fragua Baeza — angel.fragua@alumnos.upm.es

11 de diciembre de 2021

Índice

1. Introducción

En esta práctica se describe la implementación de un clasificador que permita identificar dado un texto nuevo o nunca visto antes, la categoría correcta entre «Deportes», «Política» y «Salud». Además, deberá proporcionar el orden de relevancia de las tres categorías mencionadas, es decir, cuál es la categoría más probable, la segunda más probable, y la menos probable. Se clasificará cada texto con la categoría que considere más probable. Esto no tendría por qué ser así, ya que en muchos modelos, no se asigna una categoría a no ser que se tenga cierto grado de seguridad, aunque esto suele hacerse solamente cuando confundir una predicción con otra tiene un alto coste, por ejemplo en medicina. Para la funcionalidad que se busca, no es necesario hacer tales consideraciones.

Este trabajo se ha distribuido en varias partes. En la Sección 2 se explicará la forma de obtención de los textos, para posteriormente generar y probar el modelo. En la Sección 3 se explicará el desarrollo del clasificador, siguiendo dos modelos distintos. Finalmente, en la Sección 4 se realizarán unas conclusiones sobre el trabajo realizado.

2. Recolección de textos (*dataset*)

En esta sección se describe el proceso de recolección de textos, así como la naturaleza de estos. El conjunto de textos o *dataset* ha sido generado a partir de noticias recolectadas de distintos periódicos digitales. En concreto, para este trabajo se han recolectado 60 artículos distintos de cada una de las tres categorías de «Política», «Deporte» y «Salud».

Para intentar obtener algo más de diversidad, para cada tema se han extraído noticias de múltiples periódicos digitales. Para la categoría de Salud, las 30 primeras noticias provienen del periódico de El País, mientras que las 30 últimas son de El Mundo. Algo similar se ha hecho con la categoría de Política, donde los 30 primeros artículos han sido extraídos de El País, y los últimos 30 de El Mundo. En el caso de la categoría de Deportes, todos los artículos se han extraído del MARCA, pero los 20 primeros están relacionados con el fútbol, los 20 siguientes con baloncesto y los 20 últimos con atletismo.

Los textos se han extraído a mano mediante una estrategia de copiar y pegar. La desventaja más obvia de este sistema es que hay que invertir más tiempo en el proceso de selección. La ventaja es que reduce el tiempo invertido después haciendo procesos de limpieza. Por ejemplo, si se descargaran los ficheros HTML directamente con algún tipo de *script*, en algún momento hay que extraer el texto de dentro de las etiquetas HTML; además, es muy probable que dentro del documento se encuentre texto no perteneciente al artículo, e.g. publicidad.

Por supuesto, si se pretendiese utilizar el clasificador posteriormente con textos en HTML, sí que habría que preocuparse de automatizar estos procesos de limpieza, para que no haya que hacer ninguna intervención a mano con textos nuevos. Para esta práctica se ha decidido focalizarse en obtener una buena funcionalidad para un texto que ya esté limpio, pues el tipo de formatos en sucio es muy variante, y se considera mejor desarrollar una herramienta de limpieza a medida que surja la necesidad.

Todos los artículos han sido almacenados en el directorio **Datos/**, existiendo un directorio específico para cada una de las categorías mencionadas previamente. Es decir, las noticias de Salud, se encuentran en el directorio **Datos/Salud**, las de Política en el directorio **Datos/Politica** y las de Deporte en el directorio **Datos/Deportes**. Cada noticia ha sido almacenada, en el directorio correspondiente a su categoría, con nombre **text*X*.txt** que corresponde con el ***X***-ésimo artículo recolectado de dicha categoría. Por ejemplo, **Datos/Deportes/text13.txt** se corresponde con el decimotercer texto de la categoría de deportes.

Adicionalmente se ha creado un archivo de *metadatos* llamado **Recopilacion.md** en el directorio **Datos/**, en donde se incluye un apartado para cada categoría, donde a su vez se enumeran los archivos por su nombre, acompañado del titular de la noticia y el enlace para acceder a la misma.

El objetivo inicial era que el orden de palabras por artículo fuera de entorno a 300 palabras, pero como se decidió emplear noticias completas y no solo una porción de estas, para trabajar lo más cerca de la realidad posible. El número medio de palabras aumentó un poco quedando las siguientes medias por categoría:

- Deportes: 371.9 palabras
- Política: 441.43 palabras
- Salud: 465.5 palabras

Esto significa que finalmente el orden medio de de palabras es del orden de 100 palabras más a las estimadas inicialmente, por lo que la media final por artículo es de 426,27 palabras. Cabe destacar que la desviación típica dentro de cada categoría queda muy similar tal y como se puede observar a continuación.

- Deportes: 134.28
- Política: 123.99
- Salud: 138.53

Estas métricas se pueden obtener con la ejecución del *script* de Python **Codigo/estadisticas_datos.py**.

El árbol del directorio que conformaría el *dataset* quedaría tal y como muestra la Figura 1.

```
Practica2/
├── Codigo/
└── Datos/
    ├── Deportes/
    │   ├── text1.txt
    │   ├── text2.txt
    │   ├── ⋮
    │   └── text60.txt
    ├── Politica/
    │   ├── text1.txt
    │   ├── text2.txt
    │   ├── ⋮
    │   └── text60.txt
    ├── Salud/
    │   ├── text1.txt
    │   ├── text2.txt
    │   ├── ⋮
    │   └── text60.txt
    └── Recopilacion.md
```

Figura 1: Organización del directorio de Datos

3. Desarrollo

En esta sección se plantea el desarrollo de un clasificador de textos que cubra las necesidades ya mencionadas.

3.1. División del *dataset*

Cuando se desarrollan modelos de clasificación, lo más frecuente es dividir el conjunto de datos en dos: entrenamiento y test. El conjunto de entrenamiento se utiliza para obtener el modelo, y para ello cualquier estrategia es válida, incluso memorizar cada texto y su categoría asociada. Naturalmente, esta última estrategia no es buena, pues no sirve para textos nuevos. Para probar la capacidad de generalización del modelo de forma definitiva, se utiliza el conjunto de test.

Sin embargo, el conjunto de test no debería emplearse hasta que se esté seguro de que se tiene el modelo preparado, puesto que si se utiliza la tasa de aciertos sobre dicho conjunto para ver qué hiperparámetros

funcionan mejor, se filtra información del conjunto de test hacia el modelo, haciendo que la prueba de la capacidad de generalización sea menos honesta. Por ello, se suele subdividir el conjunto de entrenamiento en uno de validación y otro que pasa a ser el de entrenamiento real, aunque hay otras estrategias como K-fold. El conjunto de validación actúa como un conjunto de test auxiliar para medir la capacidad de generalización con los hiperparámetros actuales, y cuando se tiene un resultado satisfactorio, se emplea finalmente el conjunto de test. La Figura 2 ilustra esta idea.

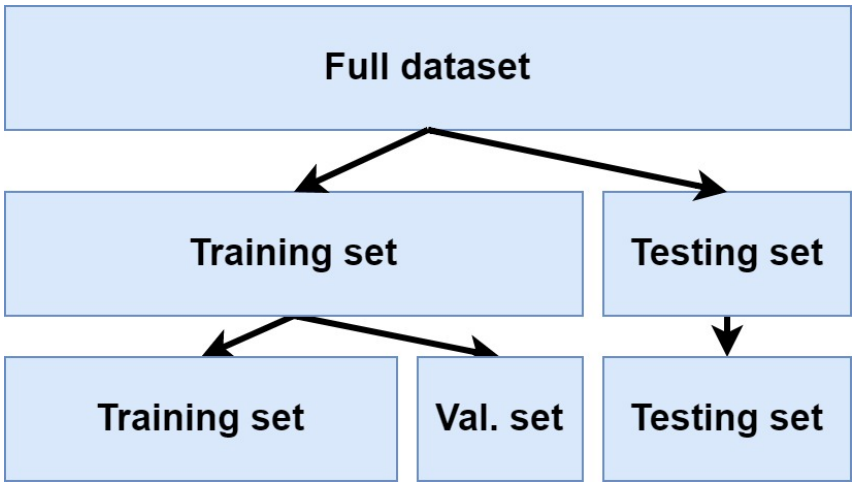

Figura 2: División del conjunto de datos, extraída de https://tinyurl.com/2p9apd46

Cuando el conjunto de datos es de textos, hay muchas formas de realizar la división de los conjuntos, por ejemplo teniendo cada uno en una carpeta. En este caso se almacenarían los textos de test de política en **Datos/test/Politica/**. Se ha decidido tomar una aproximación distinta: se genera un fichero de partición que diferencia los documentos que se corresponden con el conjunto de entrenamiento de los de test por algún tipo de identificador que permita recuperarlos. Esta alternativa es más flexible, e.g. se podrían tener los documentos de test en otro servidor y ser recuperados mediante algún protocolo; y permite realizar cambios en la partición de forma más rápida. A cambio, tiene la desventaja de que el fichero de partición debe ser consistente con la realidad.

Una de las funcionalidades de nuestro código es generar el fichero de partición. La hemos utilizado para hacer una partición del *dataset* en entrenamiento y test a proporción 50-50 de forma aleatoria. La subdivisión del conjunto de entrenamiento en validación y entrenamiento no se hace con esta utilidad, puesto que el tamaño y necesidad de un conjunto de validación depende del modelo y de si se quieren optimizar sus hiperparámetros. El resultado de la partición que hemos hecho se encuentra en **Codigo/particion_train_test.json**, y tiene un formato como el que se muestra en la Figura 3.

En este caso, no se ha codificado ningún protocolo de acceso a los archivos más allá del implícito para el sistema operativo correspondiente. Es decir, se asume que la el nombre de cada categoría coincide con el de su directorio correspondiente en el *dataset*. Por tanto, para interpretar por completo el fichero de partición será necesario conocer el directorio del *dataset*. Teniendo en cuenta que esta funcionalidad solamente se usa en la fase de pruebas del modelo, parece razonable asumir estas restricciones.

```
{
  "train": {
    "Deportes": [
      "text35.txt",
      ...
    ],
    "Politica": [
      "text19.txt",
      ...
    ],
    "Salud": [...]
  },
  "test": {
    "Deportes": [
      "text17.txt",
      ...
    ],
    "Politica": [...],
    "Salud": [...]
  }
}

```

Figura 3: Formato del fichero de partición

3.2. Clasificación con el Modelo de Espacio Vectorial (VSM)

En esta sección se detalla la implementación del modelo de clasificación como un Modelo de Espacio Vectorial (VSM). La aproximación que se tomará es una de las más populares, aunque también antiguas del estado del arte, apoyándose en la medida de relevancia TF-IDF. Este modelo permite representar documentos como vectores de números reales. Cada posición del vector nos da información sobre la presencia de un término de un determinado vocabulario en el documento. Después se pueden comparar dos documentos mediante la comparación de sus vectores asociados con alguna medida de similitud.

Siendo este modelo determinista, y teniendo pocos hiperparámetros, de los cuales se van a utilizar los más comunes, no se ha separado un conjunto de validación. Sin embargo, si quisiéramos realizar un proceso de optimización de los hiperparámetros del modelo, como la medida de similitud entre vectores o la variante TF-IDF utilizada, sí sería conveniente hacer esta separación para ver cuál es la configuración óptima para resolver el problema.

3.2.1. Generación del vocabulario

Como ya se ha mencionado, en este modelo se representan los documentos mediante vectores de números reales donde cada posición va asociada a un término. El primer paso es seleccionar el conjunto de términos $T = t_1, ..., t_n$ que deberían utilizarse. La idea para generar la estimación de T es que debería estar formado por los subconjuntos $T_i \subset T$ de términos relevantes de cada categoría i; en este caso, $T = T_P \cup T_S \cup T_D$, con T_P, T_S y T_D el conjunto de términos relevantes para política, salud y deportes respectivamente. Se busca que estos conjuntos de términos sean lo más disjuntos posibles, es decir, $\forall i, j\ T_i \cap T_j \sim \emptyset$ si $i \neq j$.

Para estimar T, se usará el conjunto de documentos de la partición de entrenamiento, para hacer una evaluación honesta del clasificador con documentos nuevos que podrían tener términos desconocidos.

Es una suposición razonable que los términos representativos de cada categoría serán sustantivos, pues son probablemente las palabras con mayor carga semántica de forma individual; siendo además este modelo uno que no evalúa dependencias semánticas entre palabras, se ha decidido seleccionar solamente los

sustantivos de cada texto como candidatos a términos. Esto no es una tarea sencilla, pues los sustantivos en español tienen múltiples variaciones.

Se ha decidido no reinventar la rueda y utilizar modelos existentes que detectan sustantivos y los lematizan. La lematización consiste en hallar para cada variante de una palabra (en género, número,...) la representante de todas ellas. Por ejemplo, la lematización de «mesas» es «mesa». De esta forma se evita tener un término distinto para cada variante, cuando en realidad todos tienen la misma carga semántica para adivinar la categoría.

En este caso, se ha usado la librería *spacy* de Python para esta tarea. La librería contiene modelos de aprendizaje automático en múltiples idiomas que permiten identificar verbos, sustantivos, etc. Un ejemplo de la salida de este parseador para la frase «Los futbolistas como Cristiano Ronaldo marcan muchos goles.» es la que se muestra en la Figura 4, donde como se ve, no solamente se reconocen sustantivos, pero también nombres propios, verbos, determinantes...

```
1   ('el', 'DET'), ('futbolista', 'NOUN'), ('como', 'SCONJ'), ('Cristiano', 'PROPN'), ('Ronaldo', 'PROPN
    '), ('marcar', 'VERB'), ('mucho', 'DET'), ('gol', 'NOUN'), ('.', 'PUNCT')
2
```

Figura 4: Salida del parseador de spacy para una oración. Se muestra (lema, tipo de palabra)

El primer paso seguido para seleccionar los términos del documento j-ésimo de la categoría i-ésima es lanzar el parseador sobre dicho documento. Se seleccionan como términos candidatos todos los sustantivos que detecta el parseador. Adicionalmente, se ha decidido tener en cuenta los nombres propios, pues en muchos casos pueden ser relevantes para la clasificación de un texto, por ejemplo «Covid-19» para salud, o «Pedro Sánchez» en política.

Aunque hay un problema evidente que se puede apreciar en la Figura 4, los nombres propios formados por varias palabras como «Cristiano Ronaldo» son detectados como dos nombres propios distintos. Para mitigar este problema, las sucesiones de varios nombres propios seguidos que saca el parseador como «Cristiano Ronaldo» en el ejemplo anterior, o nombres propios unidos por la preposición «de» y/o el determinante «el» (e.g., Ramón de la Cueva, Joaquín Jiménez López de Castro, Real Madrid...) se tratan como un solo término. Ciertamente, algunos casos escapan estas condiciones, como «Ramón y Cajal», pero es mucho más difícil distinguir Ramón y Cajal como dos personas o una sola, por lo que se consideran por defecto como dos nombres propios distintos.

Incluso con este filtrado previo, todavía pueden entrar algunas palabras que son irrelevantes por su falta de carga semántica, que podemos llamar palabras vacías. Esto se debe a que el parseador no es perfecto y se puede equivocar. Por tanto, conviene filtrar las palabras vacías del listado de términos candidatos obtenidos con el parseador. Hay muchos listados de palabras vacías distintos para el español, en este caso se ha utilizado el de *stopwords-iso*, en `https://github.com/stopwords-iso/stopwords-es` por integrar muchos otros listados de palabras vacías y su completitud, con 732 palabras. Este listado está almacenado en el fichero **Codigo/listado_palabras_vacias.json**.

Resumiendo, con este proceso se puede obtener para el texto j-ésimo de la categoría i-ésima D_{ij} (en el conjunto de entrenamiento), un listado de términos

$$T_{ij} = \texttt{FiltrarPalabrasVacias(LematizaciónSustantivosYNombresPropios(}D_{ij}\texttt{))}$$

y la cantidad de veces que aparece cada término k en dicho texto, $F_{ij}^{(k)}$, con $t_k \in T_{ij}$. Es trivial obtener las componentes análogas para cada categoría: listado de términos para la categoría i-ésima $T_i = \cup_j T_{ij}$ y cantidad de veces que aparece cada término k en dicha categoría, $F_i^{(k)} = \sum_j F_{ij}^{(k)}$; y finalmente, el listado de términos $T = \cup_i T_i$ y la cantidad de veces que aparece un término k en el conjunto de entrenamiento $F^{(k)}$.

3.2.2. Vectorización de documentos y conjuntos de documentos usando el vocabulario

Una vez conocido el vocabulario T, se usa para hacer una operación de vectorización $v(\cdot)$, que nos permita obtener dado el j-ésimo documento D_{ij} de la categoría i-ésima o conjunto de documentos asociados a la categoría i-ésima D_i los vectores asociados $V_{ij} = v(D_{ij})$ y $V_i = v(D_i)$. La operación de vectorización más básica que se nos puede ocurrir es

$$V_{ij}^{(k)} = v(D_{ij})^{(k)} = F_{ij}^{(k)} \quad V_i^{(k)} = v(D_i)^{(k)} = F_i^{(k)}$$

Es decir, el valor asociado a cada término t_k en la posición k-ésima de los vectores V_{ij}, V_i es cuántas veces ha aparecido en el texto o colección de textos de la categoría respectivamente, i.e. la frecuencia absoluta. Hay algunas desventajas en esta aproximación, siendo la más evidente que es difícil comparar vectores que se han extraído de documentos o conjuntos de documentos con distintos tamaños.

Una representación más ventajosa es TF-IDF. Tiene la siguiente definición para un documento:

$$V_{ij}^{(k)} = v_{ij}^{(k)} = TF(D_{ij})^{(k)} \times IDF(D_{ij})^{(k)}$$

Donde un valor más alto en $V_{ij}^{(k)}$ indica que el término k-ésimo es más relevante para el documento D_{ij}. El propósito de TF es ponderar la importancia de un término en el documento por cuántas veces aparece en él con respecto a otros términos, asumiendo que un término representativo para un documento debería ser frecuente en este. Se puede definir de muchas formas, la aquí escogida es normalización con el término más frecuente: $TF(D_{ij})^{(k)} = F_{ij}^{(k)} / \max_k(F_{ij}^{(k)})$.

El propósito de IDF es restar importancia a un término en un documento cuando es frecuente en el resto de documentos. Por ejemplo, el determinante «el» tendría una puntuación muy baja, pues es muy probable que aparezca en un documento, aportando poca información. Nuevamente, hay muchas formas de definirla, aquí se ha aplicado la siguiente:

$$IDF(D_{ij})^{(k)} = \log_{10}\left(\frac{N}{n^{(k)}}\right)$$

Donde N es el número de documentos del conjunto de entrenamiento, y $n^{(k)}$ es el número de documentos que contienen dicho término: $n^{(k)} = |\{D_{xy} \setminus\ t_k \in T_{xy},\ \forall(x,y)\}|$. Muchas de las variantes escogen otra base para el logaritmo; donde a mayor la base, más rápido crece la penalización por frecuencia del término.

Solamente queda definir estas operaciones para un conjunto de documentos D_i:

$$TF(D_i)^{(k)} = F_i^{(k)} / \max_k(F_i^{(k)}) \quad IDF(D_i)^{(k)} = \log_{10}\left(\frac{N}{n^{(k)}}\right)$$

Donde cabe observar que el IDF se obtiene de forma independiente para un término en un documento y el de un conjunto de documentos.

3.2.3. Clasificación mediante comparación de vectores

Ahora se tienen todos los componentes básicos para realizar la clasificación. Cuando se quiera calcular la similitud entre un documento d y un conjunto de documentos asociados a la categoría i-ésima D_i, se puede calcular como la similitud entre sus vectores asociados: $s(v(d), v(D_i))$. Existen muchas funciones de similitud posibles, se ha decidido utilizar una de las más populares, la similitud coseno, que mide el coseno del ángulo entre dos vectores,

$$s(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{||\vec{x}|| \times ||\vec{y}||}$$

y tomará valores cercanos a 1 si los vectores son muy similares, y a 0 si no están relacionados (ortogonalidad).

El proceso pues de clasificación de un documento d es obtener el listado de categorías ordenado por la similitud de la vectorización correspondiente al documento con la de los documentos de cada categoría. En concreto, se elige la categoría más similar de dicha lista.

3.2.4. Implementación del modelo

El desarrollo de las funcionalidades de librería se ha hecho en Python, y el código se encuentra en el fichero **Codigo/clasificacion_textos.py**. Este fichero contiene una programación orientada a objetos del clasificador. Sus funcionalidades son las siguientes:

1. *__init__*: Función para crear el clasificador. Carga el parseador de *spacy* y el documento con la partición de entrenamiento/test (solamente necesario si se quiere comprobar el rendimiento del clasificador sobre el conjunto de test).
2. *compilar*: Función para compilar el modelo. Se crea un fichero con una asociación de cada término T a un entero, la vectorización de cada categoría y siendo siempre la misma, la relevancia IDF de cada término, para ser utilizada con los documentos nuevos. Es necesario una llamada a este método para clasificar si aún no se dispone de un fichero de compilación existente.
3. *clasifica*: Dado un listado de las ubicaciones de unos ficheros a clasificar, genera un listado con las categorías asociadas por orden de similitud. Para ello, sigue el mismo proceso de vectorización de un documento nuevo d que el que se sigue en compilación: multiplicación de TF por IDF para cada término, donde el IDF ya está precalculado en compilación para cada término, y el TF se obtiene como ya se ha descrito: $TF(d,k) = F(d,k)/\max_k(F(d,k))$, denotando $F(d,k)$ la frecuencia absoluta del término t_k en el documento d. Después utiliza la similitud coseno para compararse con la vectorización de cada documento generada en compilación.
4. *test*: Si se ha especificado un fichero de partición, se hacen una llamada a *clasifica* sobre el conjunto de test, y se calcula la tasa de aciertos y se muestran los documentos que han sido clasificados incorrectamente, así como la salida de la llamada a *clasifica*.

3.2.5. Resultados del modelo

Para probar las funcionalidades descritas, se ha escrito un Jupyter Notebook, que se encuentra en **Codigo/prueba_concepto.ipynb**.

Para poder lanzarlo, es necesario tener Python 3 instalado y lanzar los siguientes comandos:

Command Line

```
Codigo/$ pip3 install pipenv # (pip a secas en Windows)
Codigo/$ pipenv install # Instalar librerías en entorno virtual aislado <Codigo>
(Codigo) Codigo/$ python -m spacy download es_core_news_md # Fichero auxiliar para parser de spacy
(Codigo) Codigo/$ jupyter-notebook prueba_concepto.ipynb # Lanzar el notebook
```

De forma resumida, el notebook realiza lo siguiente:

1. Compilar el clasificador. El fichero resultante se puede encontrar en **Codigo/compilacion.json**. Su formato se puede ver en la Figura 5, que incluye el listado de términos, relevancia IDF de cada uno, y vectorización TF-IDF del conjunto de documentos de cada categoría.
2. Probar el clasificador sobre la partición previamente creada en **Codigo/particion_test_train.json** mediante una llamada a *test*. Los resultados se almacenan en **Codigo/clasificacion.json**.
3. Probar el clasificador sobre un breve texto escrito por nosotros pendiente de ser clasificado mediante una llamada a *clasifica*. El texto a ser clasificado se encuentra en **Codigo/texto_prueba_concepto.txt**. Su contenido es: «Este texto trata sobre el deporte. Marcar goles es muy complicado, sobre todo, si queremos hacerlo igual de bien que Cristiano Ronaldo. Un portero que no para goles no sirve ni para sujetar la portería.»

Los resultados de la clasificación sobre el conjunto de test marcan una **tasa de aciertos del 96.67 %**. Un vistazo al fichero **Codigo/clasificacion.json** (véase Figura 6) nos permite ver qué ficheros han sido

```
{
    "terminos": {"bomba": 0, "Lesoto": 1,...},  // "<Termino>": <posicion en vector>
    "relevancia_idf": [1.95, 1.17,...] // Para cada termino del vocabulario
    "vectores_tf_idf": {
        "Politica": [0.0, 0.0, 0.05,...], // Para cada termino del vocabulario
        "Salud": [0.03, 0.3, 0.00,...],
        "Deportes": [0.0, 0.0, 0.0,...]
    }
}
```

Figura 5: Formato del fichero de compilación.

```
{
    "errores": [
        {
            "real": "Deportes/text42.txt",  // Ubicacion real
            "pred": "Salud/text42.txt"      // Prediccion
        },
        {
            "real": "Politica/text32.txt",
            "pred": "Salud/text32.txt"
        },
        {
            "real": "Salud/text23.txt",
            "pred": "Deportes/text23.txt"
        }
    ],
    "predicciones": {
        "../Datos/Deportes/text17.txt": [
            ["Deportes",0.22609791583220654], // [<Categoria>, <Similitud coseno>]
            ["Politica", 0.08074438525038288],
            ["Salud", 0.05103969365660253]
        ],
        ...,  // Resto de ficheros de test
    },
    "tasa_aciertos": "96.66666666666667%"
}

```

Figura 6: Resultados del test

clasificados incorrectamente, así como las predicciones para todos los ficheros del conjunto de test y la tasa de aciertos ya mencionada.

Un vistazo a la primera oración del fichero **Datos/Deportes/text42.txt** nos permite entender por qué se le asigna Salud: «La muerte de tres corredores en las últimas semanas en pruebas de atletismo con una clara vertiente popular, como laBehobia-San Sebastiáno en el maratón nocturno de Bilbao, han reabierto el debate de los controles médicos en este tipo de especialidades.». Donde palabras como muerte y médico pesan bastante para el tema de Salud. Algo similar ocurre con **Datos/Politica/text32** donde frecuenta la palabra «sanidad» y se confunde con un artículo de salud. Finalmente, el texto **Datos/Salud/text23.txt** habla de correr para mejorar la salud, y se confunde con uno de deporte.

En cuanto al texto escrito por nosotros para ser clasificado, produce por terminal la salida que se muestra en la Figura 7. Obviamente lo hace bien, pues el texto habla sobre porterías, marcar goles y Cristiano Ronaldo.

```
{'texto_prueba_concepto.txt': [
    ('Deportes', 0.10392268630637337),
    ('Politica', 0.011241229986113075),
    ('Salud', 0.0029481742415920034)]
}

```

Figura 7: Clasificación de **Codigo/texto_prueba_concepto.txt**

3.3. Clasificación con una red neuronal

En esta sección se incluye una breve descripción de la aplicación y resultados obtenidos al utilizar una red neuronal para esta tarea, aunque no se entrará en muchos detalles sobre el funcionamiento, solamente se discutirán a alto nivel las características del modelo.

Las redes neuronales son modelos de aprendizaje automático que en su forma básica, toman como entrada un vector de números reales y producen, para un problema categórico como este, la probabilidad de la entrada de pertenecer a cada clase. Cabe decir que es un modelo que tiene muchos más hiperparámetros, por lo que sí es conveniente separar un conjunto de validación del conjunto de entrenamiento, como se explicó en la Sección 3.1.

Naturalmente, si se quiere clasificar un texto, nuevamente hay que hacer una vectorización de algún tipo para que la red entienda la entrada. Se puede ciertamente utilizar la que ya se ha mencionado, TF-IDF. En este caso se ha decidido probar con otra, que viene descrita en el modelo que se ha tomado como base de `https://www.tensorflow.org/tutorials/keras/text_classification`. La operación de vectorización $v(\cdot)$ consiste en tres pasos: tokenización, transformación a *embedding*, y promediado global.

La tokenización es muy similar al proceso que ya hemos hecho para obtener T. Simplemente, se asigna a cada término distinto de los textos del conjunto de entrenamiento un identificador I_t, en este caso un número entero; después, cada texto se representa como un vector τ cuyo tamaño l es un hiperparámetro a determinar, donde el valor de la posición i-ésima del vector, τ_i es el identificador de la palabra i-ésima del documento tokenizado.

La transformación a *embeddings* es lo más importante. La idea es, para cada término t obtenido en la fase anterior, hacer una asignación a un vector (*embedding*) e_t de números reales de u dimensiones, donde u es un hiperparámetro a estimar. La transformación se hace de manera que el ángulo entre dos vectores asociados a dos términos similares, sea pequeño, mientras que el ángulo entre dos vectores asociados a términos sin relación sea cercano a 0. En esta práctica no se entrará en detalles sobre cómo se obtiene esta transformación, pero basta con decir que en el proceso se considera que dos términos son similares si se utilizan en contextos similares, es decir, están rodeados de términos iguales o similares en los textos del conjunto de entrenamiento.

Realizado el paso anterior, se tienen para la entrada l *embeddings* de tamaño u, es decir una matriz $l \times u$, pero la entrada debería ser un vector unidimensional. El paso final es aplastar esta matriz a un vector de tamaño l. Hay varias formas de hacerlo, pero en este caso la escogida hace un promedio, con lo que se obtiene finalmente la vectorización V de un documento. Realizada esta vectorización para cada documento, se entrena el modelo con el algoritmo de retropropagación del gradiente sobre el conjunto de entrenamiento, se validan hiperparámetros con el conjunto de validación, y finalmente, con la configuración óptima, se prueban los resultados con el conjunto de test.

Se ha seguido este proceso en un Jupyter Notebook utilizando la librería *keras* para redes neuronales, que incluye esta funcionalidad de *embeddings*. Como estas librerías son más difíciles de instalar, se ha hecho la ejecución en *Google colab*, un entorno gratuito en la nube para la ejecución de Jupyter Notebooks. Este Notebook es público y accesible mediante el siguiente enlace: https://colab.research.google.com/drive/1JkPXxnwWjPmVvZFvUmegw2PGS88uM_hY?usp=sharing. En este Notebook, también se incluyen las pruebas sobre el modelo VSM comentadas en la sección 3.2.5. Los resultados del modelo sobre el conjunto de test han dado una **tasa de aciertos del 91.11 %**.

4. Conclusiones

En esta práctica se han desarrollado dos variantes para clasificación de textos muy populares. La primera, más antigua, que es el Modelo de Espacios Vectoriales, ha dado muy buenos resultados, pese a la escasez de textos, con una tasa de aciertos del 96.66 % sobre el conjunto de test. Además, destaca su simplicidad, donde todos los procesos, excepto el parseado de sustantivos se han hecho a mano.

El segundo, usando redes neuronales ha obtenido resultados peores, aunque decentes, con una tasa de aciertos del 91.11 %. Es más complejo, menos interpretable, y se tiene que pasar más tiempo entrenando el modelo y configurando hiperparámetros. Cabe decir que tiene una representación más potente de los documentos, con el uso de *embeddings*, pero con un conjunto de textos tan pequeño como el nuestro, apenas puede ser explotada. Otra ventaja es que no se ha necesitado especificar un listado de palabras vacías, ni ha sido necesario un parseado ni de sustantivos ni de nombres propios, que es un proceso que en nuestro caso, lleva por detrás otros modelos de aprendizaje automático ya preparados. Queda para el trabajo futuro combinar los dos modelos anteriores, utilizando los vectores TF-IDF como entrada a la red, que sería similar a sustituir la similitud coseno por una red neuronal.

Como conclusión final, se puede afirmar que un modelo no es peor por ser más antiguo, y que muchas veces, particularmente cuando se tienen pocos datos, los modelos más sencillos dan mejores resultados. Sin embargo, si se dispusiera de un corpus de gran tamaño de textos clasificados, valdría la pena volver a probar el modelo con una red neuronal similar a la mencionada.